NOTE ON MACHINE LEARNING

A PREPRINT

Haocheng Dai

Contents

1 Linear Algebra

Finite dimension domain	Infinite dimension domain
matrix	operator
covariance matrix	covariance operator (kernel)
vector	function

1.1 Understand a matrix

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

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

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

$$\begin{bmatrix}1\\ 0\end{bmatrix} \to \begin{bmatrix}3\\ 1\end{bmatrix}, \begin{bmatrix}0\\ 1\end{bmatrix} \to \begin{bmatrix}1\\ 2\end{bmatrix}$$

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

$$\mathbf{AB} = \begin{bmatrix}\langle \mathbf{A}_{1,:}, \mathbf{B}_{:,1}\rangle & \langle \mathbf{A}_{1,:}, \mathbf{B}_{:,2}\rangle & \cdots & \langle \mathbf{A}_{1,:}, \mathbf{B}_{:,n}\rangle \\ \langle \mathbf{A}_{2,:}, \mathbf{B}_{:,1}\rangle & \langle \mathbf{A}_{2,:}, \mathbf{B}_{:,2}\rangle & \cdots & \langle \mathbf{A}_{2,:}, \mathbf{B}_{:,n}\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{A}_{n,:}, \mathbf{B}_{:,1}\rangle & \langle \mathbf{A}_{n,:}, \mathbf{B}_{:,2}\rangle & \cdots & \langle \mathbf{A}_{n,:}, \mathbf{B}_{:,n}\rangle \end{bmatrix} \quad \triangleright \text{ inner product}$$
$$= \sum_{i=1}^{n} \mathbf{A}_{:,i} \otimes \mathbf{B}_{i,:} \quad \triangleright \text{ outer product}$$

1.3 Conjugate transpose (Adjoint)

Definition 1.2. **Conjugate transpose** $\mathbf{A}^*$ (or Hermitian transpose $\mathbf{A}^H$ or **adjoint**) of a matrix $\mathbf{A}$ is defined as

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

Remark 1.1. *For an operator like a matrix:*

$$\textit{adjoint} = \textit{(conjugate) transpose}$$
$$\textit{classic adjoint} = \textit{adjugate}$$

In most linear algebra discussions, "adjoint" refers to the transpose of a matrix.

Proof. To prove the adjugate of a real matrix $\mathbf{A}$ is transpose, we have

$$\begin{aligned}\langle \mathbf{A}f, g\rangle &= (\mathbf{A}f)^\top g = f^\top \mathbf{A}^\top g \\ \langle f, \mathbf{A}^* g\rangle &= \langle f, \mathbf{A}^\top g\rangle = f^\top \mathbf{A}^\top g \\ \Rightarrow \langle \mathbf{A}f, g\rangle &= \langle f, \mathbf{A}^* g\rangle\end{aligned}$$

□

Example 2.

- Adjoint of the gradient is the negative divergence: $\langle \nabla f, g\rangle = \langle f, -\nabla \cdot g\rangle$
- Adjoint of the Fourier transform is its inverse: $\langle \mathcal{F}(f), g\rangle = \langle f, \mathcal{F}^{-1}(g)\rangle$
- Adjoint of the Laplacian is itself: $\langle \Delta f, g\rangle = \langle f, \Delta g\rangle$
- Adjoint of the linear interpolation is the splatting: $\langle f \circ \phi, g\rangle = \langle f, \phi^* g\rangle$

1.4 Orthogonality

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

Example 3. The rotation matrix is an orthogonal matrix:

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

Remark 1.2. *If* $\mathbf{A}$ *is a orthogonal matrix, then* $\mathrm{Det}(\mathbf{A})$ *is either 1 or -1.*

Remark 1.3. $\mathbf{A}$ *is orthogonal means rows (columns) of* $\mathbf{A}$ *are orthonormal (orthogonal and maganitude is 1) in* $\mathbb{R}^{n\times n}$*, namely* $|\mathrm{Det}(\mathbf{A})| = 1, \|\mathbf{A}\mathbf{x}\| = \|\mathbf{x}\|$*. When* $\mathbf{A}$ *is orthogonal and* $\mathrm{Det}(\mathbf{A}) = 1$*, it's called rotation matrix.*

Remark 1.4. *The orthogonal matrices are of interest because their inverse is very cheap to compute.*

1.5 Eigen

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

$$\begin{aligned}\mathbf{A}\mathbf{x} &= \lambda\mathbf{x} \\ \mathbf{A}\mathbf{x} - \lambda\mathbf{x} &= 0 \\ (\mathbf{A} - \lambda\mathbf{I})\mathbf{x} &= 0 \\ |\mathbf{A} - \lambda\mathbf{I}| &= 0\end{aligned}$$

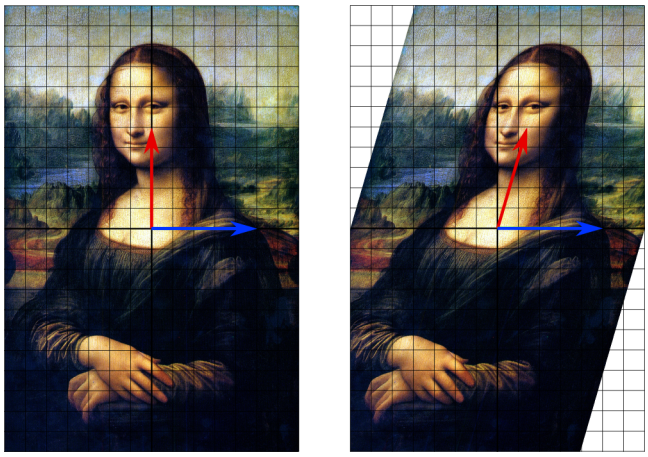


Figure 1: In this shear mapping the red arrow changes direction, but the blue arrow does not. The blue arrow is an eigenvector of this shear mapping because it does not change direction, and since its length is unchanged, its eigenvalue is 1.

Remark 1.5. *The effect of the transformation brought by* $\mathbf{A}$ *is that the space is scaled by* $\lambda^{(i)}$ *in direction of* $\mathbf{x}^{(i)}$.

Remark 1.6.

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

Remark 1.7. *Let* $\mathbf{A}$ *be a square* $n \times n$ *matrix with* n *linearly independent eigenvectors* $\mathbf{x}^{(i)}$. *Then* $\mathbf{A}$ *can be factorized as*

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1},$$

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

1.6 Inverse

- $(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$
- $(\mathbf{ABC}...)^{-1} = ...\mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$
- $(\mathbf{A}^{-1})^{\top} = (\mathbf{A}^{\top})^{-1}$
- $(r\mathbf{A})^{-1} = r^{-1}\mathbf{A}^{-1}$

1.7 Transpose

- $(\mathbf{AB})^{\top} = \mathbf{B}^{\top}\mathbf{A}^{\top}$
- $(\mathbf{ABc}...)^{\top} = ...\mathbf{c}^{\top}\mathbf{B}^{\top}\mathbf{A}^{\top}$
- $(\mathbf{A} + \mathbf{B})^{\top} = \mathbf{A}^{\top} + \mathbf{B}^{\top}$
- $(\mathbf{A}^{-1})^{\top} = (\mathbf{A}^{\top})^{-1}$
- $(r\mathbf{A})^{\top} = r\mathbf{A}^{\top}$

1.8 Trace

- $\mathrm{Tr}(\mathbf{AB}) = \mathrm{Tr}(\mathbf{BA})$
- $\mathrm{Tr}(\mathbf{ABC}) = \mathrm{Tr}(\mathbf{BCA}) = \mathrm{Tr}(\mathbf{cAB})$
- $\mathrm{Tr}(\mathbf{A} + \mathbf{B}) = \mathrm{Tr}(\mathbf{A}) + \mathrm{Tr}(\mathbf{B})$
- $\mathrm{Tr}(\mathbf{A}^\top) = \mathrm{Tr}(\mathbf{A})$
- $\mathrm{Tr}(r\mathbf{A}) = r\,\mathrm{Tr}(\mathbf{A})$
- $\mathrm{Tr}(\mathbf{P}^{-1}\mathbf{AP}) = \mathrm{Tr}(\mathbf{A})$

1.9 Determinant

- $\mathrm{Det}(\mathbf{AB}) = \mathrm{Det}(\mathbf{A})\,\mathrm{Det}(\mathbf{B})$
- $\mathrm{Det}(\mathbf{A}^{-1}) = \mathrm{Det}(\mathbf{A})^{-1}$, if $\mathbf{A}$ is invertable
- $\mathrm{Det}(\mathbf{A}^\top) = \mathrm{Det}(\mathbf{A})$
- $\mathrm{Det}(r\mathbf{A}) = r^n\,\mathrm{Det}(\mathbf{A})$
- $\mathrm{Det}(\mathbf{A}) = \prod \lambda^{(i)}$
- $\mathrm{Det}(\mathbf{A}) = \prod \mathbf{A}_{i,i}$, and $\mathbf{A}_{i,i}$ are the eigenvalues of $\mathbf{A}$, if $\mathbf{A}$ is triangular or diagonal
- $\mathrm{Det}(\mathbf{A})\mathbf{A}^{-1} = \mathbf{A}^*$

Definition 1.5. **Singular matrix** is the matrix

non-invertible $\Leftrightarrow \mathrm{Det}(\mathbf{A}) = 0 \Leftrightarrow$ rank deficient $\Leftrightarrow$ singular $\Leftrightarrow$ each column not linearly independent

Definition 1.6. **Nonsingular matrix** is the matrix

invertible $\Leftrightarrow \mathrm{Det}(\mathbf{A}) \neq 0 \Leftrightarrow$ full rank $\Leftrightarrow$ nonsingular $\Leftrightarrow$ each column linearly independent

1.10 Logarithm

- $\log(\mathbf{AB}) = \log \mathbf{A} + \log \mathbf{B}$

1.11 Matrix exponential and logarithm

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

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

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

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

where $\mathbf{A}^i$ is the matrix multiplication of i $\mathbf{A}$.

Proof. Given the following ODE

$$\frac{d}{dt} f(t) = x f(t),$$

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 $\mathbf{A} \in \mathbb{R}^{n \times n}$ and $f(\cdot) : \mathbb{R} \to \mathbb{R}^n$, we still expect the matrix to satisfy the form above, namely

$$\begin{aligned} \frac{d}{dt} f(t) &= \mathbf{A} f(t) \\ f(t) &= e^{\mathbf{A}t} \cdot \mathbf{c}. \end{aligned}$$

To prove it, we substitute the definition of the matrix exponential into the equation above, we have

$$\begin{aligned} \frac{d}{dt} f(t) &= \frac{d}{dt} e^{\mathbf{A}t} \cdot \mathbf{c} \\ &= \frac{d}{dt} \sum_{i=0}^{\infty} t^i \frac{\mathbf{A}^i}{i!} \cdot \mathbf{c} \\ &= \sum_{i=0}^{\infty} i t^{i-1} \frac{\mathbf{A}^i}{i!} \cdot \mathbf{c} \\ &= \sum_{i=0}^{\infty} t^{i-1} \frac{\mathbf{A} \cdot \mathbf{A}^{i-1}}{(i-1)!} \cdot \mathbf{c} \\ &= \mathbf{A} \cdot \sum_{i=0}^{\infty} t^{i-1} \frac{\mathbf{A}^{i-1}}{(i-1)!} \cdot \mathbf{c} \\ &= \mathbf{A} \cdot e^{\mathbf{A}t} \cdot \mathbf{c} \\ &= \mathbf{A} f(t) \end{aligned}$$

which proves the definition of matrix exponential is true and consistent with the original exponential. □

Definition 1.9. **Real logarithm** $\ln : \mathbb{R} \to \mathbb{R}$ is defined by

$$\begin{aligned} \ln(a) &= \sum_{i=1}^{\infty} (-1)^{i+1} \frac{(a-1)^i}{i} \\ &= (a-1) - \frac{(a-1)^2}{2} + \frac{(a-1)^3}{3} - \frac{(a-1)^4}{4} + \cdots. \end{aligned}$$

Definition 1.10. **Matrix logarithm** $\ln : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n}$ is likewise defined by

$$\begin{aligned} \ln(\mathbf{A}) &= \sum_{i=1}^{\infty} (-1)^{i+1} \frac{(\mathbf{A} - \mathbf{I})^i}{i} \\ &= (\mathbf{A} - \mathbf{I}) - \frac{(\mathbf{A} - \mathbf{I})^2}{2} + \frac{(\mathbf{A} - \mathbf{I})^3}{3} - \frac{(\mathbf{A} - \mathbf{I})^4}{4} + \cdots, \end{aligned}$$

where $(\mathbf{A} - \mathbf{I})^i$ is the matrix multiplication of i $\mathbf{A} - \mathbf{I}$.

1.12 Positive Definite

If $\mathbf{A}, \mathbf{B}$ are two positive definite matrices, $\lambda^{(i)}$ is a eigenvalue of $\mathbf{A}$, then

- $\mathbf{A} + \mathbf{B}$ is still positive definite.
- $\forall \lambda^{(i)} > 0$.
- $\mathrm{Tr}(\mathbf{A}) = \prod_i \lambda^{(i)} > 0$.
- $\mathrm{Det}(\mathbf{A}) = \sum_i \lambda^{(i)} > 0$.
- Any principal submatrix of a positive definite matrix is positive definite.

1.13 Decomposition

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

$$\mathbf{A} = \mathbf{L}\mathbf{L}^\top$$

Remark 1.8. *Using the Cholesky decomposition, we solve* $\mathbf{L}\mathbf{L}^\top\mathbf{x} = \mathbf{b}$ *instead of* $\mathbf{A}\mathbf{x} = \mathbf{b}$ *by*

1. *Solving* $\mathbf{L}\mathbf{z} = \mathbf{b}$ *for* $\mathbf{z}$ *using forward substitution*
2. *Solving* $\mathbf{L}^\top\mathbf{x} = \mathbf{z}$ *for* $\mathbf{x}$ *using backward substitution*

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

$$\mathbf{A} = \mathbf{L}\mathbf{U}$$

Remark 1.9. *Using the LU decomposition, we solve* $\mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{b}$ *instead of* $\mathbf{A}\mathbf{x} = \mathbf{b}$ *by*

1. *Solving* $\mathbf{L}\mathbf{z} = \mathbf{b}$ *for* $\mathbf{z}$ *using forward substitution*
2. *Solving* $\mathbf{U}\mathbf{x} = \mathbf{z}$ *for* $\mathbf{x}$ *using backward substitution*

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

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1},$$

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

Remark 1.10. *When* $\mathbf{A}$ *is a* $n \times n$ *real symmetric matrix can be decomposed as*

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top,$$

since the eigenvalues are real and the eigenvectors are orthonormal, therefore $\mathbf{Q}$ *is an orthogonal matrix.*

Remark 1.11. *A square* $n \times n$ *matrix* $\mathbf{A}$ *is called diagonalizable or non-defective if there exists an invertible matrix* $\mathbf{P}$ *s.t.* $\mathbf{P}^{-1}\mathbf{A}\mathbf{P}$ *is a diagonal matrix. Only diagonalizable matrices can be eigendecomposed.*

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

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top,$$

where $\sigma^{(i)}$ are called singular values.

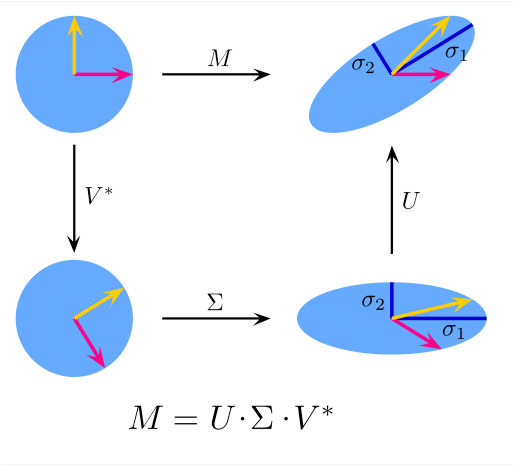


Figure 2: Illustration of the singular value decomposition $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ of a real 2×2 matrix $\mathbf{M}$. Down left: The action of $\mathbf{V}^\top$, is a rotation. Down right: The action of $\mathbf{\Sigma}$, a scaling by the singular values $\sigma^{(1)}$ horizontally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

Remark 1.12. *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* $\mathbf{A}$ *can be written as the weighted sum of outer products of vectors. Specifically, the matrix* $\mathbf{A}$ *can be decomposed as*

$$\mathbf{A} = \sum_{i=1}^{n} \sigma^{(i)} \mathbf{U}_{:,i} \mathbf{V}_{:,i}^\top,$$

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

Remark 1.13. *For real positive definite symmetric matrix*

- *The columns of* $\mathbf{U}$ *are eigenvectors of* $\mathbf{A}\mathbf{A}^\top$.
- *The columns of* $\mathbf{V}$ *are eigenvectors of* $\mathbf{A}^\top\mathbf{A}$.
- *The non-zero singular values from* $\mathbf{\Sigma}$ *are the square roots of the non-zero eigenvalues of* $\mathbf{A}\mathbf{A}^\top$ *or* $\mathbf{A}^\top\mathbf{A}$.

Remark 1.14. *Using the SVD decomposition, we solve* $\mathbf{A}\mathbf{x} = \mathbf{b}$ *by*

$$\mathbf{x} = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^\top\mathbf{b}$$

Example 4. SVD in image compression The figure below is of the size 450×333, so we can store this image as a matrix $\mathbf{A} \in \mathbb{R}^{450\times333}$.

Figure 3: Original image

Now we can decompose $\mathbf{A}$ by SVD as below

$$\mathbf{A} = \sigma^{(1)}\mathbf{U}_{:,1}\mathbf{V}_{:,1}^{\top} + \sigma^{(2)}\mathbf{U}_{:,2}\mathbf{V}_{:,2}^{\top} + \cdots + \sigma^{(n)}\mathbf{U}_{:,n}\mathbf{V}_{:,n}^{\top},$$

where $n = \min\{450, 333\}$, $u_i v_i^{\top}$ 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 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 $\mathbf{A} \in \mathbb{R}^{n \times n}$, if there exist invertible matrix $\mathbf{P} \in \mathbb{R}^{n \times n}$ s.t.

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{P} = Diag,$$

then $\mathbf{A}$ 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

$$\begin{aligned}\Pr(\text{reason}|\text{observation}) &= \frac{\Pr(\text{observation}|\text{reason})}{\Pr(\text{observation})} \times \Pr(\text{reason}) \\ &\propto \text{Likelihood} \times \text{Prior probability}\end{aligned}$$

Definition 2.4. **Bayes' rule** is as follows:

$$\Pr(R|O) = \frac{\Pr(R,O)}{\Pr(O)} = \frac{\Pr(O) \times \Pr(O|R)}{\Pr(R)}$$

Remark 2.1. *Below are the two variants of Bayes' rule:*

- $\Pr(A|B,C) = \frac{\Pr(A,B|C)}{\Pr(B|C)}$

  Proof.

$$\begin{aligned}\Pr(A|B,C) &= \frac{\Pr(A,B,C)}{\Pr(B,C)} \\ &= \frac{\Pr(A,B,C)/\Pr(C)}{\Pr(B,C)/\Pr(C)} \\ &= \frac{\Pr(A,B|C)}{\Pr(B|C)}\end{aligned}$$

  □

- $\Pr(A,B|C) = \Pr(A|B,C) \times \Pr(B|C)$

  Proof.

$$\begin{aligned}\Pr(A,B|C) &= \frac{\Pr(A,B,C)}{\Pr(C)} \\ &= \frac{\Pr(A|B,C) \times \Pr(B|C) \times \Pr(C)}{\Pr(C)} \\ &= \Pr(A|B,C) \times \Pr(B|C)\end{aligned}$$

  □

Definition 2.5. **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 2.6. **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 5. 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 2.2. *A PDF must be integrated over an interval to yield a probability.*

Definition 2.7. **The convolution of two probability distributions**, denoted as $Z = X + Y$, is given by the following integral:

$$f_Z(z) = \int_{-\infty}^{\infty} f_X(x)\, f_Y(z - x)\, dx$$

Definition 2.8. **Marginal probability** can be yield by sum rule

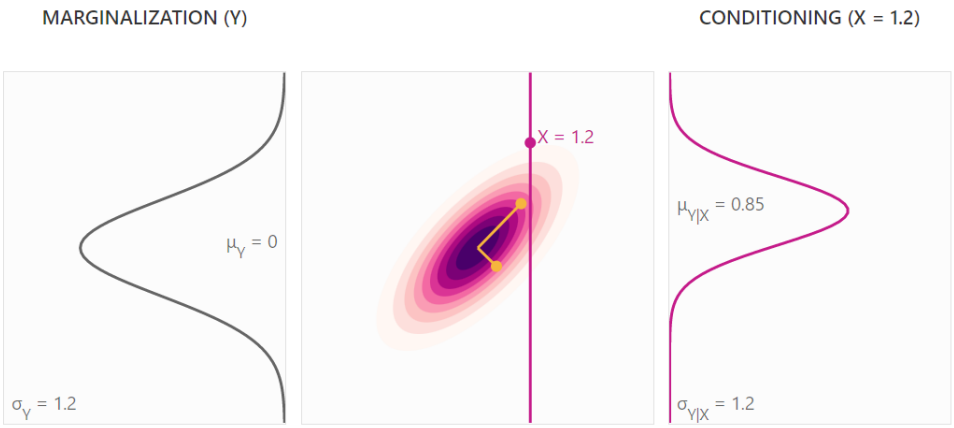
$$\Pr(X = x) = \sum_Y \Pr(X = x, Y = y) \qquad \triangleright \text{Discrete}$$

$$\Pr(x) = \int \Pr(x, y) dy \qquad \triangleright \text{Continuous}$$

Definition 2.9. **Conditional probability** can be yield by

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

Example 6. 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 2.10. **Chain rule of conditional probability** is in form of

$$\Pr(X^{(1)}, \cdots, X^{(n)}) = \Pr(X^{(1)}) \prod_{i=2}^{n} \Pr(X^i | X^{(1)}, \cdots, X^{(i-1)}).$$

Definition 2.11. **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 2.3. *PDF is the derivative of the CDF.*

Definition 2.12. **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 7. 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 $0 \leq p \leq 1$.

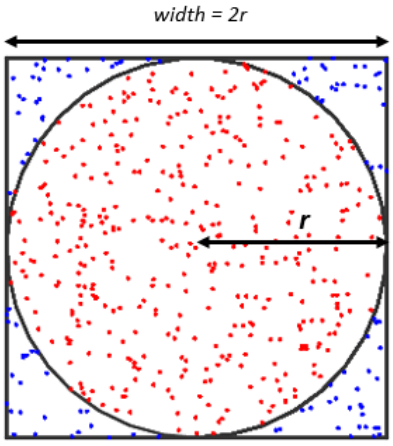
By differentiating $\mathcal{L}(p)$ with respect to p and setting it to zero, we yield that the maximum likelihood estimator for p is $\frac{49}{80}$.

Definition 2.13. **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 8. Consider a quadrant (circular sector) inscribed in a unit square. Given that the ratio of their areas is $\frac{\pi}{4}$, the value of π 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 π.



Definition 2.14. **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 μ is the mean value and σ is the standard deviation.

Remark 2.4. *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 $\{X^{(1)}, \cdots, X^{(n)}\}$ is an infinite sequence of independent identically distributed random variables with expectation $E(X^{(1)}) = \cdots = E(X^{(n)}) = \mu$, we can have*

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

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

Remark 2.5. *We can approximate the expectation of an unknown distribution by performing a sufficient number of trials.*

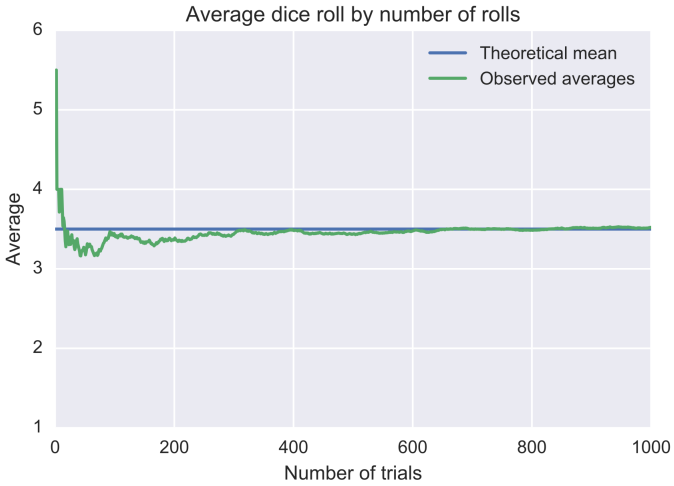


Figure 5: Law of large numbers

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 2.6. *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 9. Given a sequence of i.i.d. random variables $\{X^{(1)}, \cdots, X^{(n)}\}$, 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 2.15. **Student's t-distribution** is defined as

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

where ν is the number of degrees of freedom and Γ is the gamma function.

Definition 2.16. A non-degenerate distribution is a **stable distribution** if it satisfies the following property:

Let X_1 and X_2 be independent realizations of a random variable X. Then X is said to be stable if for any constants $a > 0$ and $b > 0$ the random variable $aX_1 + bX_2$ has the same distribution as $cX + d$ for some constants $c > 0$ and d. The distribution is said to be strictly stable if this holds with $d = 0$.

Example 10. Normal distribution, the Cauchy distribution, and the Lévy distribution are stable distributions.

Remark 2.7. *The α-stable distribution, often referred to simply as the stable distribution, is a family of symmetric probability distributions that generalize the normal (Gaussian) distribution. Unlike the normal distribution, stable distributions are characterized by four parameters:*

1. ***Alpha** (α)**: The stability index, which determines the shape of the distribution. Values of α can range from 0 to 2. If $\alpha = 2$, the distribution is Gaussian. As α decreases, the tails of the distribution become heavier.*
2. ***Beta** (β)**: The skewness parameter, which measures the asymmetry of the distribution. When $\beta = 0$, the distribution is symmetric. Non-zero values of β introduce skewness.*
3. ***Gamma** (γ)**: The scale parameter, which determines the scale of the distribution. It is a positive real number.*
4. ***Delta** (δ)**: The location parameter, which represents a shift along the x-axis. It is also a real number.*

Definition 2.17. The distribution of a random variable X with distribution function F is said to have a heavy (right) tail if

$$\overline{F}(x) \equiv \Pr[X > x] \lim_{x \to \infty} e^{tx} \overline{F}(x) = \infty \quad \text{for all } t > 0.$$

Example 11. The Lévy distribution, Cauchy distribution, and t-distribution are heavy tail distributions.

Definition 2.18. **Moment** of a function are quantitative measures related to the shape of the function's graph.

Example 12.

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

Definition 2.19. **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 2.20. **Variance** is the expectation of the squared deviation of a random variable from its population mean or sample mean.

$$\begin{aligned} \mathrm{Var}(X) &= \int \Pr(x)(x-\mu)^2 dx \\ &= E\left((X-\mu)^2\right) \\ &= E\left((X - E(X))^2\right) \\ &= E\left(X^2 - 2XE(X) + E(X)^2\right) \\ &= E\left(X^2\right) - 2E(X)E(X) + E(X)^2 \\ &= 2E\left(X^2\right) - 2E(X)^2 \end{aligned}$$

Definition 2.21. **t-value** (also known as the t-statistic) is a measure that quantifies the difference between the sample mean and a hypothesized population mean in terms of the variability of the data.

$$t = \frac{\bar{x} - \mu}{\sigma / \sqrt{n}},$$

where $\bar{x}$ is the sample mean, μ is the hypothesized population mean, σ is the sample standard deviation, n is the sample size.

Remark 2.8. *Note that when testing a single sample mean against a known or hypothesized population mean, there's only one sample standard deviation (s) in the denominator, and the formula simplifies accordingly.*

A two-sample t-test, which involves comparing the means of two groups $\{x_1\}, \{x_2\}$, reads as

$$t = \frac{\bar{x_1} - \bar{x_2}}{\sqrt{\sigma_1^2/n_1 + \sigma_2^2/n_2}},$$

Remark 2.9. *The t-value measures how much evidence the data provide against the null hypothesis (typically a statement of no effect or no difference). A larger absolute t-value suggests stronger evidence against the null hypothesis — the sample mean is relatively far from the null hypothesis.*

Definition 2.22. **Null hypothesis H_0** is a statement that suggests there is no significant effect, relationship, or difference in a population.

Example 13. In a medical study: H_0 might state that a new drug has no effect on a certain condition, meaning there is no difference in outcomes between patients who take the drug and those who don't.

Definition 2.23. **p-value** is a measure that helps us assess the strength of evidence against a null hypothesis.

Remark 2.10. *In simpler terms, p-value tells us how likely it is to get the results we've observed if the null hypothesis is true. A small p-value suggests that the observed results are unlikely to have occurred by chance alone, which could lead us to reject the null hypothesis in favor of an alternative hypothesis that suggests a real effect or relationship.*

Example 14. p-values for continuous sample data. Suppose a pharmaceutical company conducts a clinical trial to test a new blood pressure medication. They recruit two groups of patients: Group A receiving the new drug and Group B receiving a placebo. Each group consists of 20 patients. The goal is to determine if the new drug has a statistically significant effect on reducing blood pressure. Here's a more detailed breakdown of the steps and calculations involved:

1. Collect Data: Blood pressure measurements are taken for both groups before and after treatment. The data are summarized as follows:

Table 1: Group A (New Drug)

Before Treatment	150	145	160	155	148	152	158	150	147	...
After Treatment	138	132	140	142	136	142	147	135	140	...

Table 2: Group B (Placebo)

Before Treatment	148	152	145	150	146	144	155	158	153	...
After Treatment	146	150	143	148	145	142	154	156	152	...

2. Calculate Test Statistic: We will use a two-sample t-test to compare the means of the blood pressure measurements between the two groups. The formula for the t-test is: $t = \frac{(\bar{x}_1 - \bar{x}_2)}{\sqrt{(\sigma_1^2/n_1) + (\sigma_2^2/n_2)}}$, where $\bar{x}_1$ and $\bar{x}_1$ are the sample means, σ_1^2 and σ_1^2 are the sample standard deviations, and n_1 and n_2 are the sample sizes.
   For Group A:
   - Mean (Before): 153.6, Standard Deviation (Before): 5.45
   - Mean (After): 143.25, Standard Deviation (After): 4.37

   For Group B:
   - Mean (Before): 150.45, Standard Deviation (Before): 3.34
   - Mean (After): 149.55, Standard Deviation (After): 3.10

   Calculated t-value: $\frac{143.25 - 149.55}{\sqrt{(4.37^2/20) + (3.10^2/20)}} = -5.25$ [1]

[1] The sign of the t-value indicates the direction of the difference between the means. In this example, a negative t-value suggests that Group A (the new drug) has a lower mean blood pressure after treatment compared to Group B (the placebo).

3. Determine Degrees of Freedom (df): $df = n_1 + n_2 - 2 = 20 + 20 - 2 = 38$
4. Lookup in t-Distribution Table: Using the calculated t-value (1.61) and degrees of freedom (38), we find the corresponding area under the t-distribution curve. For this hypothetical example, let's assume the area corresponds to a p-value of approximately 0.058.
5. Calculate p-Value: The derived p-value from the t-distribution is 0.058.
6. Compare p-Value to Significance Level: Assuming a significance level (α) of 0.05, since the calculated p-value (0.058) is slightly higher than α, researchers may choose not to reject the null hypothesis. This suggests that there is not enough evidence to conclude that the new drug has a significant effect on reducing blood pressure.

Definition 2.24. **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 15. Suppose $\{s^{(1)}, \cdots, s^{(n)}\}$ is a sequence of independent samples from a normally distributed population with unknown parameters mean μ and variance σ^2. Let

$$\bar{s} = (s^{(1)} + \cdots + s^{(n)})/n,$$
$$\mathrm{Var}(s) = \frac{1}{n-1}\sum_{i=1}^{n}(s^{(i)} - \bar{s}\,)^2,$$

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

$$T = \frac{\bar{s} - \mu}{\sqrt{\mathrm{Var}(s)}/\sqrt{n}}$$

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 μ and σ^2; i.e., it is a pivotal quantity. Suppose we wanted to calculate a 95% confidence interval for μ. Then, denoting c^2 as the 97.5th percentile of this distribution,

$$\Pr(-c \le T \le c) = 0.95$$

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,

$$\Pr\left(\bar{s} - \frac{c\sqrt{\mathrm{Var}(s)}}{\sqrt{n}} \le \mu \le \bar{s} + \frac{c\sqrt{\mathrm{Var}(s)}}{\sqrt{n}}\right) = 0.95$$

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

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

$$\left(\bar{s} - \frac{c\sqrt{\mathrm{Var}(s)}}{\sqrt{n}}, \bar{s} + \frac{c\sqrt{\mathrm{Var}(s)}}{\sqrt{n}}\right).$$

Definition 2.25. **Covariance** for two random variables X and Y, each with sample size n is defined by

$$\mathrm{Cov}(X, Y) = E((X - E(X))(Y - E(Y)))$$
$$= \sum_{i=1}^{n}\frac{(x_i - \bar{x})(y_i - \bar{y})}{n}$$

[2] In practice, we find the c in a t-table

Definition 2.26. **Covariance matrix** for a set of random variables $\{X^{(1)}, \cdots, X^{(n)}\}$, each with sample size m is defined by

$$\mathbf{\Sigma} = \begin{bmatrix} \mathrm{Cov}(X^{(1)}, X^{(1)}) & \cdots & \mathrm{Cov}(X^{(1)}, X^{(n)}) \\ \vdots & \ddots & \vdots \\ \mathrm{Cov}(X^{(n)}, X^{(1)}) & \cdots & \mathrm{Cov}(X^{(n)}, X^{(n)}) \end{bmatrix}$$

Remark 2.11. ***Covariance matrix and Riemannian metric.[Arvanitidis et al., 2016]*** *A local covariance matrix can be used to represent the local structure of the data: the inverse of a local diagonal covariance matrix Σ^{-1} can be treated as the metric tensor, as the eigenvector corresponds to the smallest eigenvalue of the metric tensor is the direction of the geodesic. The geodesic are "pulled" towards the data where the metric is small.*

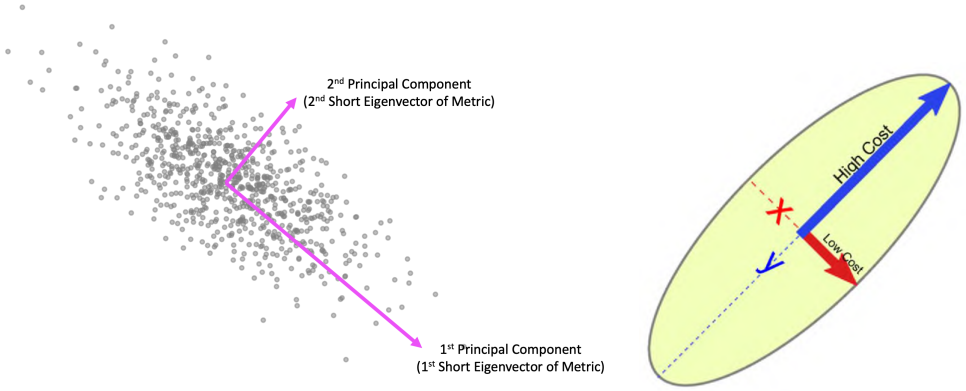


Figure 6: Left: Correspondance between eigenvectors of covariancce matrix and the Riemannian metric. Right: Visualization of metric tensor.

Remark 2.12. *The kernel function in the Gaussian process can be regarded as an infinite-dimensional covariance matrix.*

Definition 2.27. The population correlation coefficient $\rho_{X,Y}$ between two random variables X and Y with expected values μ_X and μ_Y and standard deviations σ_X and σ_Y is defined as:

$$\rho_{X,Y} = \mathrm{corr}(X,Y) = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{\mathrm{E}[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}, \quad \text{if } \sigma_X \sigma_Y > 0.$$

where E is the expected value operator, cov means covariance.

Definition 2.28. **Gaussian random field (GRF)** is a random field involving Gaussian probability density functions of the variables.

Definition 2.29. **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 2.13. *A Gaussian process is an infinite-variate Gaussian distribution.*

Remark 2.14. *A Gaussian process is a one-dimensional Gaussian random field.*

Remark 2.15. *How to visualize samples from a Gaussian process:*[3]

1. *Define the mean function and kernel for the GP* $\mu = 0, k(\mathbf{x}, \mathbf{x}') = \exp\left(\frac{-(\mathbf{x}-\mathbf{x}')^2}{2\gamma^2}\right)$;

[3] `https://www.cs.cmu.edu/~16831-f14/notes/F12/16831_lecture20_venkatrn.pdf`

2. *Sample inputs* $\mathbf{x}_i = \epsilon \times i, i = \{0, 1, \cdots, 1/\epsilon\}$*;*
3. *Compute the kernel matrix* Σ*. For the example with* $\epsilon = 0.1$*, we would have an* 11×11 *matrix;*
4. *Sample from the multivariate Gaussian distribution* $\mathcal{N}(0, \Sigma)$*.*

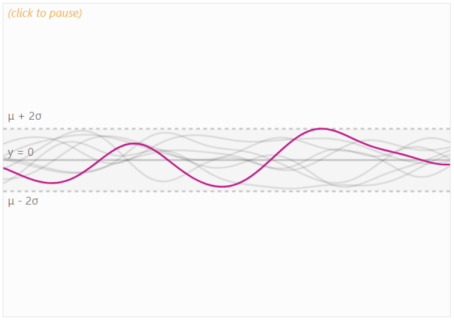


Figure 7: Gaussian process sampling

Remark 2.16. *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:*[4]

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

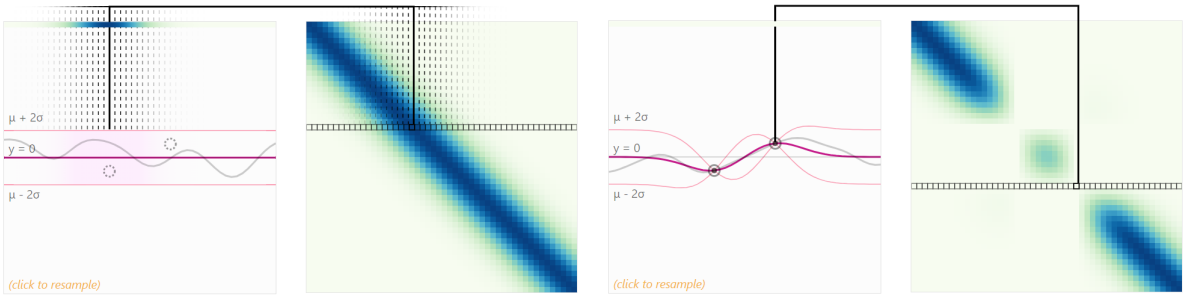


Figure 8: Darker color indicates higher value. (a) At first, no training points have been observed. Accordingly, the mean prediction remains at 0 and the standard deviation is the same for each test point. (b) By hovering over the covariance matrix you can see the influence of each point on the current 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.

[4] `https://distill.pub/2019/visual-exploration-gaussian-processes/#DimensionSwap`

Definition 2.30. **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 $\mathbf{x}, \mathbf{y} \in \mathcal{X}$, we have

$$k(\cdot,\cdot) : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$
$$k(\mathbf{x},\mathbf{y}) = \langle \phi(\mathbf{x}), \phi(\mathbf{y}) \rangle_{\mathcal{H}}$$

Remark 2.17. *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 $\mathbf{x}, \mathbf{y}$ represent two different books, we can't take an inner product between books, but we can take an inner product between the feature maps $\phi(\mathbf{x}), \phi(\mathbf{y})$ corresponding to $\mathbf{x}, \mathbf{y}$.*

Remark 2.18. *The kernel gives a way to compute inner products in some feature space without even knowing what this space is and what is ϕ. 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 $k(\mathbf{x},\mathbf{y}) = \exp(-\gamma\|\mathbf{x}-\mathbf{y}\|^2)$. If we Taylor-expand this function, we'll see that it corresponds to an infinite-dimensional codomain of ϕ.*

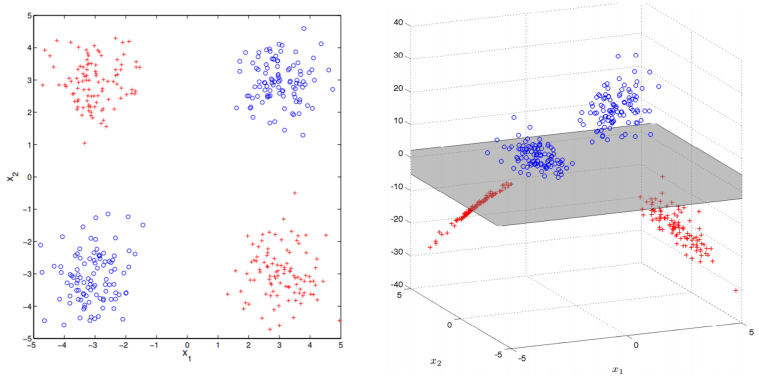


Figure 9: $\phi(\mathbf{x}) = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_1\mathbf{x}_2]^\top$ example of the kernel: on the left, the points are plotted in the original space; on the right, the points are plotted into a higher dimensional feature space by ϕ.

Remark 2.19.

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

Definition 2.32. **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 $\{\mathbf{e}^{(1)}, \cdots, \mathbf{e}^{(k)}\}$ of a set of points $\{\mathbf{s}^{(1)}, \cdots, \mathbf{s}^{(n)}\} \in \mathbb{R}^d$, and the new basis is of the same dimension d as the original space, which satisfies the re-

cursive relationship

$$\begin{aligned}
\mathbf{e}^{(1)} &= \arg\max_{\|\mathbf{e}\|=1} \sum_{i=1}^{n} \langle \mathbf{e}, \mathbf{s}^{(i)} \rangle^2 \\
\mathbf{e}^{(2)} &= \arg\max_{\|\mathbf{e}\|=1} \sum_{i=1}^{n} [\langle \mathbf{e}^{(1)}, \mathbf{s}^{(i)} \rangle^2 + \langle \mathbf{e}, \mathbf{s}^{(i)} \rangle^2] \\
&\vdots \\
\mathbf{e}^{(k)} &= \arg\max_{\|\mathbf{e}\|=1} \sum_{i=1}^{n} [\langle \mathbf{e}^{(1)}, \mathbf{s}^{(i)} \rangle^2 + \cdots + \langle \mathbf{e}^{(k-1)}, \mathbf{s}^{(i)} \rangle^2 + \langle \mathbf{e}, \mathbf{s}^{(i)} \rangle^2]
\end{aligned}$$

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

Remark 2.20. *Concatenating all the feature vectors together as a matrix, and using SVD to decompose the matrix is a common way of performing PCA.*

Example 16. Given a dataset $\{\mathbf{s}^{(1)}, \cdots, \mathbf{s}^{(n)}\}$ of n samples, where $\mathbf{s}^{(i)} \in \mathbb{R}^d$ and $\bar{\mathbf{s}}$ is the mean of the dataset

$$\mathbf{s}^{(i)} = \begin{bmatrix} \mathbf{s}_1^{(i)} \\ \vdots \\ \mathbf{s}_d^{(i)} \end{bmatrix}, \bar{\mathbf{s}} = \begin{bmatrix} \bar{\mathbf{s}}_1 \\ \vdots \\ \bar{\mathbf{s}}_d \end{bmatrix}.$$

Then we can have a $d \times d$ covariance matrix $\mathbf{\Sigma}$

$$\mathbf{\Sigma} = \begin{bmatrix} \text{Cov}(\mathbf{s}_1, \mathbf{s}_1) & \cdots & \text{Cov}(\mathbf{s}_1, \mathbf{s}_d) \\ \vdots & \ddots & \vdots \\ \text{Cov}(\mathbf{s}_d, \mathbf{s}_1) & \cdots & \text{Cov}(\mathbf{s}_d, \mathbf{s}_d) \end{bmatrix} = \begin{bmatrix} \text{cov of } 1^{st} \text{ and } 1^{st} \text{ dim} & \cdots & \text{cov of } 1^{st} \text{ and } d^{th} \text{ dim} \\ \vdots & \ddots & \vdots \\ \text{cov of } d^{th} \text{ and } 1^{st} \text{ dim} & \cdots & \text{cov of } d^{th} \text{ and } d^{th} \text{ dim} \end{bmatrix},$$

where

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

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 $\mathbf{s}'$ is formed as below

$$\underbrace{\begin{bmatrix} \mathbf{s}'^{(1)} & \cdots & \mathbf{s}'^{(n)} \end{bmatrix}}_{k \times n} = \underbrace{\begin{bmatrix} \mathbf{e}^{(1)\top} \\ \vdots \\ \mathbf{e}^{(k)\top} \end{bmatrix}}_{k \times d} \cdot \underbrace{\begin{bmatrix} \mathbf{s}^{(1)} & \cdots & \mathbf{s}^{(n)} \end{bmatrix}}_{d \times n}$$

Example 17. Given a dataset $\{\mathbf{s}^{(1)}, \cdots, \mathbf{s}^{(n)}\}$ of n samples, where $\mathbf{s}^{(i)} \in \mathbb{R}^d$. Concatenating all the $\mathbf{s}^{(i)}$ together, we can have a matrix $\mathbf{S}$ of size $n \times d$.

- If we use the SVD to solve PCA, we simply decompose $\mathbf{S}$ to $\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where the $\mathbf{V}$ is the eigenvector matrix regarding the dimension covariance, and $\mathbf{U}$ is the eigenvector matrix regarding the sample size covariance.
- For PCA, we first yield the covariance matrix of $\mathbf{S}$ by $\mathbf{S}^\top\mathbf{S} \in \mathbb{R}^{d \times d}$, then we perform the eigendecomposition on $\mathbf{S}^\top\mathbf{S}$, as $\mathbf{S}^\top\mathbf{S} = (\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top)^\top\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{V}\mathbf{\Sigma}\mathbf{U}^\top\mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^\top = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$ to acquire the largest k eigenvalues and their corresponding eigenvectors.

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

Definition 2.33. **Multidimensional Scaling (MDS)** [Kruskal, 1964] is a dimensionality reduction technique that aims to represent high-dimensional data in a lower-dimensional space while preserving the pairwise distances between data points as much as possible. The goal of MDS is, given

$$\mathbf{D} = \begin{bmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,M} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,M} \\ \vdots & \vdots & & \vdots \\ d_{M,1} & d_{M,2} & \cdots & d_{M,M} \end{bmatrix},$$

to find M vectors $x^{(1)}, \ldots, x^{(M)} \in \mathbb{R}^N$ such that $\|x^{(i)} - x^{(j)}\| \approx d_{i,j}$ for all $i, j \in 1, \ldots, M$, where $\|\cdot\|$ is a vector norm. In classical MDS, this norm is the Euclidean distance, but, in a broader sense, it may be a metric or arbitrary distance function.

Definition 2.34. **Isomap** [Tenenbaum et al., 2000] is a dimensionality reduction algorithm that seeks to preserve the global structure of the data.

1. Input: The algorithm takes as input a high-dimensional dataset, typically represented as a matrix of feature vectors.
2. Compute pairwise distances: Calculate the pairwise distances $d_E(i, j)$ between all data points using a distance metric such as Euclidean distance or some domain-specific metric.
3. Construct the neighborhood graph: Create an adjacency matrix to represent the neighborhood graph by connecting points i and j if they are closer than ϵ (ϵ-Isomap), or if i is one of the k nearest neighbors of j (k-Isomap). Set edge lengths equal to $d_E(i, j)$
4. Compute shortest paths: Apply shortest path algorithm to find the shortest path distances between all pairs of points in the neighborhood graph, represented by adjacency matrix $\mathbf{D}_S = \{d_s(i, j)\}$
5. Construct the low-dimensional embedding: Use eigenvalue decomposition to find a lower-dimensional representation of the data. Let λ_p be the p-th eigenvalue (in decreasing order) of the matrix $\tau(\mathbf{D}_S)$, and $\mathbf{v}_i^{(p)}$ be the i-th component of the p-th eigenvector. Then set the p-th component of the d-dimensional coordinate vector $y^{(i)}$ equal to $\sqrt{\lambda^{(p)}}\mathbf{v}_i^{(p)}$
6. Output: The result of the Isomap algorithm is a lower-dimensional representation of the data, where each data point is now represented by a reduced set of coordinates.

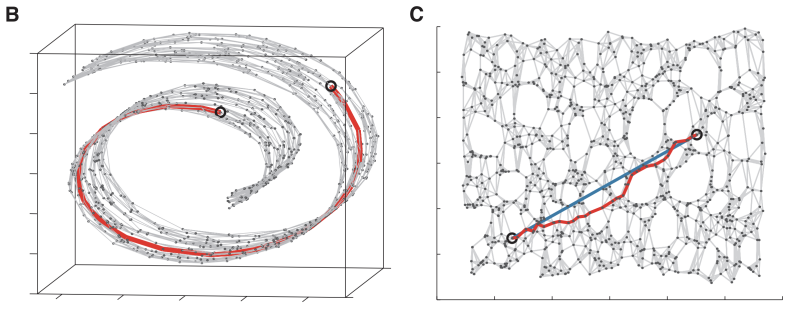


Figure 10: Finding the shortest path between two point in a Swiss row.

Definition 2.35. **Linear Discriminant Analysis (LDA)** projects the input data to a linear subspace consisting of the directions which maximize the separation between classes.

Remark 2.21. *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 2.36. **Mahalanobis distance** is a measure of the distance between a point $\mathbf{s}^{(i)}$ and a distribution, can be written as

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

where μ is the mean vector and $\mathbf{\Sigma}$ 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 $\mathbf{\Sigma}$:

$$d_M(\mathbf{s}^{(i)}, \mathbf{s}^{(j)}) = \sqrt{(\mathbf{s}^{(i)} - \mu)^\top \mathbf{\Sigma}^{-1} (\mathbf{s}^{(j)} - \mu)}.$$

Remark 2.22. *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 the Mahalanobis distance, you can view it as calculating the norm of $\mathbf{s}^{(i)} - \mu$ w.r.t. covariance matrix. Assuming $\mathbf{v} = \mathbf{s}^{(i)} - \mu$, Mahalanobis distance $d_M(\mathbf{s}^i) = \sqrt{\sum_{ij} \mathbf{v}_i \mathbf{\Sigma}^{-1}_{i,j} \mathbf{v}_j}$,

Remark 2.23. *The Riemannian metric $g(\mathbf{x})$ acts on tangent vectors may be interpreted as a standard Mahalanobis metric restricted to an infinitesimal region around $\mathbf{x}$.*

Remark 2.24. ***Another interpretation of Mahalanobis distance*** *d_M. Due to the positive definiteness of covariance matrix $M = \Sigma^{-1} = \mathbf{J}^\top \mathbf{J}$, we can have*

$$\begin{aligned} d_M^2(\mathbf{s}^{(i)}, \mathbf{s}^{(j)}) &= (\mathbf{s}^{(i)} - \mathbf{s}^{(j)})^\top M (\mathbf{s}^{(i)} - \mathbf{s}^{(j)}) \\ &= (\mathbf{s}^{(i)} - \mathbf{s}^{(j)})^\top \mathbf{J}^\top \mathbf{J} (\mathbf{s}^{(i)} - \mathbf{s}^{(j)}) \\ &= (\mathbf{J}(\mathbf{s}^{(i)} - \mathbf{s}^{(j)}))^\top \mathbf{J} (\mathbf{s}^{(i)} - \mathbf{s}^{(j)}) \\ &= \|\mathbf{J}(\mathbf{s}^{(i)} - \mathbf{s}^{(j)})\|_2^2 \end{aligned}$$

Learning a Mahalanobis distance is equivalent to learning a linear mapping L that transforms the data into a new space. And the corresponding distance is basically the Euclidean distance.

Definition 2.37. **Metric** is a mapping $d : X \times X \to \mathbb{R}$ over a vector space X if for all $\mathbf{x}^{(i)}, \mathbf{x}^{(j)}, \mathbf{x}^{(k)} \in X$, it satisfies the properties:

- Triangular inequality: $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) + d(\mathbf{x}^{(j)}, \mathbf{x}^{(k)}) \geq d(\mathbf{x}^{(i)}, \mathbf{x}^{(k)})$
- Non-negativity: $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \geq 0$
- Symmetry: $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = d(\mathbf{x}^{(j)}, \mathbf{x}^{(i)})$
- Distinguishability: $d(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = 0 \Leftrightarrow \mathbf{x}^{(i)} = \mathbf{x}^{(j)}$

The ordered pair (X, d) is called a **metric space**. Strictly speaking, if a mapping satisfies the first three properties but not the fourth, it is called **pseudometric**.

Remark 2.25. *Metric tensor M can be decomposed to $\mathbf{J}^\top\mathbf{J}$ due to its positive definiteness, where $\mathbf{J}$ is namely the Jacobian matrix representing the diffeomorphism (exponential map) maps the tangent space to the manifold.*

Remark 2.26. *From a Mahalanobis distance perspective $d_M(\mathbf{x}^{(i)},\mathbf{x}^{(j)})=\sqrt{(\mathbf{x}^{(i)}-\mathbf{x}^{(j)})^\top M(\mathbf{x}^{(i)}-\mathbf{x}^{(j)})}$, if d_M is a pseudometric (namely $d(\mathbf{x}^{(i)},\mathbf{x}^{(j)})=0 \not\leftrightarrow \mathbf{x}^{(i)}=\mathbf{x}^{(j)}$), M is not full-rank[Suárez et al., 2021], which arises to the problem of dimensionality reduction.*

Definition 2.38. **t-distributed stochastic neighbor embedding (t-SNE)** is a statistical method for visualizing high-dimensional data by giving each datapoint a location in a two or three-dimensional map.

Example 18. Given a set of n high-dimensional objects $\{\mathbf{s}^{(1)},\cdots,\mathbf{s}^{(n)}\}$, t-SNE first computes probabilities p_{ij} that are proportional to the similarity of objects $\mathbf{s}^{(i)}$ and $\mathbf{s}^{(i)}$, as follows.

For $i\neq j$, define

$$p_{j|i}=\frac{\exp(-\|\mathbf{s}^{(i)}-\mathbf{s}^{(j)}\|^2/2\sigma_i^2)}{\sum_{k\neq i}\exp(-\|\mathbf{s}^{(i)}-\mathbf{s}^{(k)}\|^2/2\sigma_i^2)}$$

and set $p_{i|i}=0$. Note that $\sum_j p_{j|i}=1$ for all i.

As Van der Maaten and Hinton explained: "The similarity of datapoint S^j to datapoint S^i is the conditional probability, $p_{j|i}$, 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^i."

Now define

$$p_{ij}=\frac{p_{j|i}+p_{i|j}}{2n}$$

and note that $p_{ij}=p_{ji}$, $p_{ii}=0$, and $\sum_{i,j}p_{ij}=1$.

The bandwidth of the Gaussian kernels σ_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 σ_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[5], 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.

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 $\mathbf{Q}$, that is:

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

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.

[5]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.

3 Machine Learning

3.1 Non-Deep Learning Models

3.1.1 Classification and clustering

Definition 3.1. **Logistic regression** is a statistical model that uses a logistic function to model a binary dependent variable. Given a dataset of $\{(\mathbf{x}^{(i)}, y^{(i)})\}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ 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.

$$h_\theta(\mathbf{x}) = g(\theta^\top \mathbf{x}), g(z) = \frac{1}{1 + e^{-z}}$$

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

Remark 3.1. *The conditional distribution $y \mid x$ is a Bernoulli distribution rather than a Gaussian distribution.*

Remark 3.2. *Logistic regression is indeed non-linear in terms of Odds and Probability, however, it is linear in terms of Log Odds.*

- *Probability of* $(Y = 1)$*:* $p = \frac{1}{1+\exp(-\theta_0 - \theta_1 x^{(1)} - \theta_2 x^{(2)})}$
- *Odds of* $(Y = 1)$*:* $\frac{p}{1-p} = \exp(-\theta_0 - \theta_1 x^{(1)} - \theta_2 x^{(2)})$
- *Log Odds of* $(Y = 1)$*:* $\log\left(\frac{p}{1-p}\right) = -\theta_0 - \theta_1 x^{(1)} - \theta_2 x^{(2)}$

Remark 3.3. *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[6], the likelihood of the parameters θ of the model $q(X = c_i; \theta)$ on the training set is

$$P_{\text{model}}(x; \theta) = q(X = c_i)$$

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

$$\begin{aligned}
\theta_{\text{ML}} &= \arg\max_\theta \left(\underbrace{q(X = c_1) \times \cdots \times q(X = c_1)}_{p(X=c_1) \text{ times}} \times \underbrace{q(X = c_2) \times \cdots \times q(X = c_2)}_{p(X=c_2) \text{ times}} \right) \\
&= \arg\max_\theta \left(\prod_i^2 q(X = c_i)^{p(X=c_i)} \right) && \triangleright \text{ 2: category number} \\
&= \arg\max_\theta \left(\sum_i^2 \log\left(q(X = c_i)^{p(X=c_i)} \right) \right) \\
&= \arg\max_\theta \left(\sum_i^2 (p(X = c_i) \log(q(X = c_i)) \right) && \triangleright \text{ Opposite of cross-entropy}
\end{aligned}$$

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. $H(\mathbb{S})$ and $H(\mathbb{S}_v)$ are the entropy of $\mathbb{S}$ and $\mathbb{S}_v$ w.r.t. final class.[7] □

[6] The training set we have is actually derived from real-world sampling.

[7] For real example, please refer to this https://towardsdatascience.com/decision-trees-for-classification-id3-algorithm-explained-89df76e72df1

Definition 3.2. **Hyperplane** $\mathbf{w}^\top \mathbf{x} - b = 0$ of an n-dimensional space V is a subspace of dimension $n - 1$.

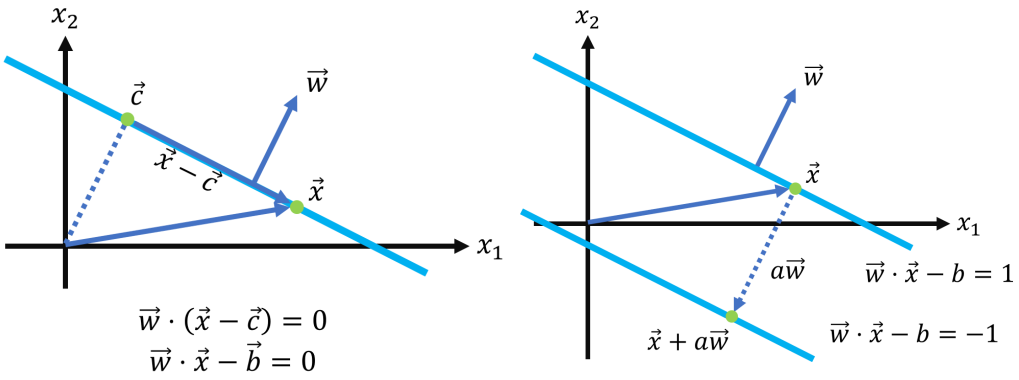


Figure 11: Left: Hyperplane of an 2D space. Right: distance between two hyperplanes.

Remark 3.4. *The distance between the origin and the hyperplane is* $\left(\frac{\mathbf{w}}{\|\mathbf{w}\|}\right)^\top \mathbf{x} = \frac{\mathbf{w}^\top \mathbf{x}}{\|\mathbf{w}\|} = \frac{b}{\|\mathbf{w}\|}$*, namely the projection of* $\mathbf{x}$ *on* $\mathbf{w}/\|\mathbf{w}\|$.

Remark 3.5. *The distance* $|a|$ *between* $\mathbf{w}^\top \mathbf{x} - b = c_1$ *and* $\mathbf{w}^\top \mathbf{x} - b = c_2$ *can be derived as below: Assuming* $\mathbf{w}$ *is a unit vector and* $a\mathbf{w}$ *is distance vector between* $\mathbf{w}^\top \mathbf{x} - b = c_1$ *and* $\mathbf{w}^\top \mathbf{x} - b = c_2$*,* $\mathbf{x}'$ *is a point on* $\mathbf{w}^\top \mathbf{x} - b = c_1$*, hence* $\mathbf{x}' + a\mathbf{w}$ *is located on* $\mathbf{w}^\top \mathbf{x} - b = c_2$*, namely*

$$\begin{aligned} \mathbf{w}^\top (\mathbf{x}' + a\mathbf{w}) - b &= c_2 \\ \mathbf{w}^\top \mathbf{x}' + a\mathbf{w}^\top \mathbf{w} - b &= c_2 \\ c_1 + a\mathbf{w}^\top \mathbf{w} &= c_2 \\ a\mathbf{w}^\top \mathbf{w} &= c_2 - c_1 \\ a &= -\frac{c_2 - c_1}{\|\mathbf{w}\|}. \end{aligned}$$

Definition 3.3. **Support vector machine (SVM)** tries to find a hyper-plane to classify data samples.

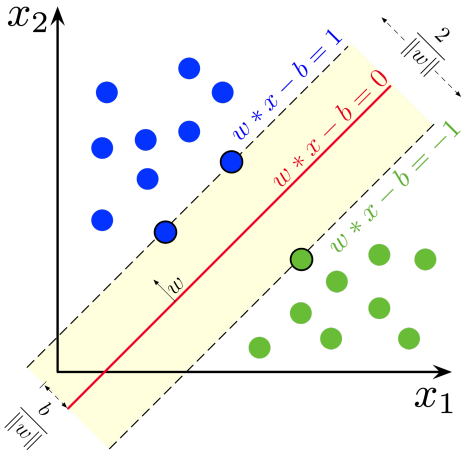


Figure 12: Support vector machine.

Example 19. 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 $\mathbf{x}^{(i)}$ 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 $\mathbf{x}^{(i)}$ from either group is maximized. Any hyperplane can be written as the set of points $\mathbf{x}$ satisfying $\mathbf{w}^T\mathbf{x} - b = 0$, where $\mathbf{w}$ 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

$\mathbf{w}^\top\mathbf{x} - b = 1$ anything on or above this boundary is of one class, with label 1

$\mathbf{w}^\top\mathbf{x} - b = -1$ anything on or below this boundary is of the other class, with label -1

Geometrically, the distance between these two hyperplanes is $\frac{2}{\|\mathbf{w}\|}$, so to maximize the distance between the planes we want to minimize $\|\mathbf{w}\|$. 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

$$\begin{aligned} &\mathbf{w}^\top\mathbf{x}^{(i)} - b \geq 1\,, && \text{if } y^{(i)} = 1, \\ &\mathbf{w}^\top\mathbf{x}^{(i)} - b \leq -1\,, && \text{if } y^{(i)} = -1. \end{aligned}$$

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

$$y^{(i)}(\mathbf{w}^\top\mathbf{x}^{(i)} - b) \geq 1, \quad \text{for all } 1 \leq i \leq n.$$

We can put this together to get the optimization problem:

$$\begin{aligned} &\underset{\mathbf{w},\, b}{\text{minimize}} && \|\mathbf{w}\|_2^2 \\ &\text{subject to} && y^{(i)}(\mathbf{w}^\top\mathbf{x}^{(i)} - b) \geq 1, \forall i \in \{1, \ldots, n\} \end{aligned}$$

The $\mathbf{w}$ 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 $\mathbf{x}^{(i)}$ that lie nearest to it. These $\mathbf{x}^{(i)}$ 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:

$$\underset{\mathbf{w},\, b}{\text{minimize}} \qquad \lambda\|\mathbf{w}\|^2 + \left(\frac{1}{n}\sum_{i=1}^{n} \max\left(0, 1 - y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} - b)\right)\right),$$

where the parameter $\max\left(0, 1 - y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} - b)\right)$ is the hinge loss (Def. 3.71), 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:

$$\mathcal{L}(\mathbf{w}, b, \lambda) = \frac{1}{2}\|\mathbf{w}\|_2^2 + \sum_i \lambda_i \left(1 - y^{(i)}(\mathbf{w}^\top\mathbf{x}^{(i)} - b)\right)$$

The primal-dual problem[8] would be

$$\text{Primal: } \underset{\mathbf{w},\, b}{\text{minimize}}\ \|\mathbf{w}\|_2^2, \text{ subject to } y^{(i)}(\mathbf{w}^\top\mathbf{x}^{(i)} - b) \geq 1, \forall i \in \{1, \ldots, n\}$$

$$\text{Dual: } \max_{\lambda} \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \lambda) = \frac{1}{2}\|\mathbf{w}\|_2^2 + \sum_i \lambda_i \left(1 - y^{(i)}(\mathbf{w}^\top\mathbf{x}^{(i)} - b)\right)$$

[8] max dual = min primal

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

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum_i y^{(i)}\mathbf{x}^{(i)} = 0$$
$$\mathbf{w} = \sum_i \lambda_i y^{(i)}\mathbf{x}^{(i)}$$

Substituting $\mathbf{w}$ back into the Lagrangian we get:

$$\begin{aligned}
\mathcal{L}(b,\lambda) &= \frac{1}{2}\left(\sum_i \lambda_i y^{(i)}\mathbf{x}^{(i)}\right)^\top \left(\sum_j \lambda_j y^{(j)}\mathbf{x}^{(j)}\right) + \sum_i \lambda_i \left(1 - y^{(i)}\left(\left(\sum_j \lambda_j y^{(j)}\mathbf{x}^{(j)}\right)^\top \mathbf{x}^{(i)} - b\right)\right) \\
&= \frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}{\mathbf{x}^{(i)}}^\top\mathbf{x}^{(j)} + \sum_i \lambda_i \left(1 - y^{(i)}\left(\left(\sum_j \lambda_j y^{(j)}{\mathbf{x}^{(j)}}^\top\mathbf{x}^{(i)}\right) - b\right)\right) \\
&= \frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}{\mathbf{x}^{(i)}}^\top\mathbf{x}^{(j)} + \sum_i \lambda_i \left(1 - \sum_j \lambda_j y^{(i)}y^{(j)}{\mathbf{x}^{(j)}}^\top\mathbf{x}^{(i)} + y^{(i)}b\right) \\
&= \frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}{\mathbf{x}^{(i)}}^\top\mathbf{x}^{(j)} + \sum_i \lambda_i - \sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}{\mathbf{x}^{(j)}}^\top\mathbf{x}^{(i)} + \sum_i \lambda_i y^{(i)}b \\
\mathcal{L}(b,\lambda) &= -\frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}{\mathbf{x}^{(i)}}^\top\mathbf{x}^{(j)} + \sum_i \lambda_i + \sum_i \lambda_i y^{(i)}b \\
\mathcal{L}(\lambda) &= -\frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}{\mathbf{x}^{(i)}}^\top\mathbf{x}^{(j)} + \sum_i \lambda_i
\end{aligned}$$

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

Remark 3.6. *Why are we solving the dual problem instead of the primal one?*[9]

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

$$\begin{aligned}
\textit{Without kernel trick: } \max_\lambda \mathcal{L}(\lambda) &= -\frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}{\mathbf{x}^{(i)}}^\top\mathbf{x}^{(j)} + \sum_i \lambda_i \\
\textit{and } \mathbf{w} &= \sum_i \lambda_i y^{(i)}\mathbf{x}^{(i)} \\
\textit{With kernel trick: } \max_\lambda \mathcal{L}(\lambda) &= -\frac{1}{2}\sum_i\sum_j \lambda_i\lambda_j y^{(i)}y^{(j)}k(\mathbf{x}^{(i)},\mathbf{x}^{(j)}) + \sum_i \lambda_i \\
\textit{where } k(\mathbf{x}^{(i)},\mathbf{x}^{(j)}) &= \langle\phi(\mathbf{x}^{(i)}),\phi(\mathbf{x}^{(j)})\rangle \\
\textit{and } \mathbf{w} &= \sum_i \lambda_i y^{(i)}\phi(\mathbf{x}^{(i)}), b = \mathbf{w}^\top\mathbf{x}^{(i)} - y^{(i)}
\end{aligned}$$

Remark 3.7. ***What's the sparseness of SVM?*** *The SVM solution is sparse in the sense that it depends only on a few critical training instances rather than the entire dataset. This property makes SVMs efficient in terms of memory and computational requirements, especially when dealing with large datasets.*

By relying on a subset of support vectors, SVMs can generalize well to new, unseen data. The support vectors capture the essential characteristics of the data distribution and provide robust decision boundaries. This sparsity also helps to alleviate overfitting, as the SVM solution focuses on the most informative instances.

[9] `https://www.cs.cmu.edu/~aarti/Class/10315_Fall20/lecs/svm_dual_kernel.pdf`

It's important to note that while the sparsity property of SVMs is desirable, it is not guaranteed in all cases. The level of sparseness depends on various factors, including the dataset, the choice of kernel, and the regularization parameters of the SVM. In some cases, SVM solutions may have a larger number of support vectors, resulting in a less sparse representation.

Remark 3.8. ***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 3.4. **Naive Bayes** is an algorithm that assigns a class to a new sample by the votes from k nearest neighbors.

Definition 3.5. **k-nearest neighbors algorithm** is an algorithm that assigns a class to a new sample by the votes from k nearest neighbors.

Definition 3.6. **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 3.9. ***Difference between kNN and k-means.***

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

Example 20. 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 3.10. *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.*

3.1.2 Ensembles

Definition 3.7. **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 21. Information gain for a feature column F is calculated as:

$$IG(\mathbb{S}, F) = H(\mathbb{S}) - \sum_v \left(\frac{|\mathbb{S}_v|}{|\mathbb{S}|} \times H(\mathbb{S}_v) \right)$$

where $\mathbb{S}_v$ is the set of sample (row) in $\mathbb{S}$ for which the feature (column) F has value v, $|\cdot|$ stands for the number of set element.

Definition 3.8. **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 22. 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 23. 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:

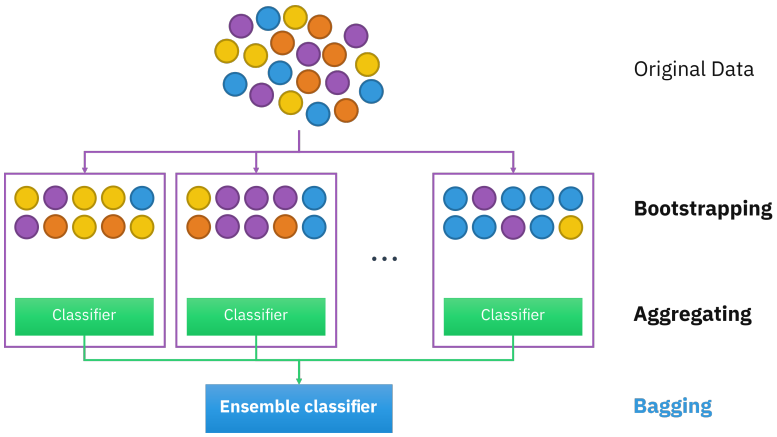
1. Check for the above base cases.
2. For each attribute a, find the normalized information gain ratio from splitting on a.
3. Let F_{best} be the attribute with the highest normalized information gain.
4. Create a decision node that splits on F_{best}.
5. Recurse on the sublists obtained by splitting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

Remark 3.11. *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 3.9. **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 24. 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).



Definition 3.10. **Random forests** is an extension over bagging. In addition to taking the random subset of data, it also takes the random selection of features[10] rather than using all features to grow trees.

Remark 3.12. *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:

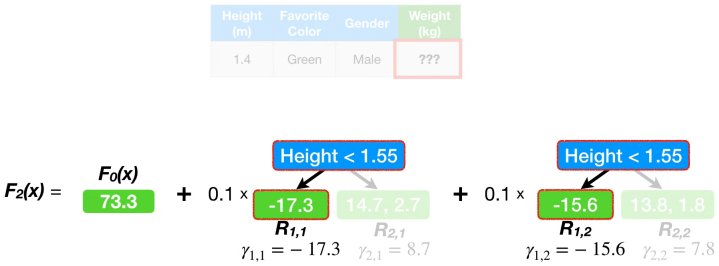
- *Random forests are incredibly dependent on their data sets, changing these can drastically change the individual trees' structures;*
- *Requires much more time to train the data compared to decision trees, make decisions, and vote within the random forest classifier.*

Remark 3.13. *The difference between boosting and random forest comes from* $\text{error} = \text{bias} + \text{variance}$*:*

- *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 3.11. **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.

[10] 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.



The **Predicted Weight** = 73.3 + (0.1 × -17.3) + (0.1 × -15.6) = 70

Example 25. Input: training set $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^n$, a differentiable loss function $\mathcal{L}(y, F(\mathbf{x})) = \frac{1}{2}(y - F(\mathbf{x}))^2$, where F is the tree model, M is the number of iterations. Algorithm[11]:

1. Initialize model with a constant value:
$$F_0(\mathbf{x}) = \arg\min_{\gamma} \sum_{i=1}^{n} \mathcal{L}(y^{(i)}, \gamma).$$
2. For $m = 1$ to M, compute so-called pseudo-residuals:
   (a)
$$r_{im} = -\left(\frac{\partial \mathcal{L}(y^{(i)}, F(\mathbf{x}^{(i)}))}{\partial F(\mathbf{x}^{(i)})}\right)_{F(\mathbf{x})=F_{m-1}(\mathbf{x})} = y_i - F(\mathbf{x}_i) \quad \text{for } i = 1, \ldots, n.$$
   (b) Fit a base learner (making a new tree) closed under scaling $h_m(\mathbf{x})$ to pseudo-residuals, i.e. train it using the training set $\{(\mathbf{x}^{(i)}, r_{im})\}_{i=1}^n$.
   (c) Compute multiplier γ_m by solving the following one-dimensional optimization problem[12]:
$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} \mathcal{L}\left(y^{(i)}, F_{m-1}(\mathbf{x}^{(i)}) + \gamma h_m(\mathbf{x}^{(i)})\right).$$
   (d) Update the model[13]:
$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \gamma_m h_m(\mathbf{x}).$$
3. Output $F_M(\mathbf{x})$.

3.1.3 Regression

Definition 3.12. **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 $\{(\mathbf{x}^{(i)}, y^{(i)})\}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^n$ 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.

$$\mathbf{y} = \mathbf{X}\theta + \varepsilon$$

where

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)T} \\ \mathbf{x}^{(2)T} \\ \vdots \\ \mathbf{x}^{(2)T} \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{x}_1^{(1)} & \cdots & \mathbf{x}_p^{(1)} \\ 1 & \mathbf{x}_1^{(2)} & \cdots & \mathbf{x}_p^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \mathbf{x}_1^{(n)} & \cdots & \mathbf{x}_p^{(n)} \end{bmatrix}, \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix}, \quad \varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}.$$

[11] Intuitive explaination.
[12] If there are multiple values in one leaf, take the average as the leaf value.
[13] $h_m(x)$ is the scaling factor, which prevents overfitting.

Definition 3.13. **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

$$f'\left(x^{(n)}\right) = \frac{f\left(x^{(n)}\right) - 0}{x^n - x^{(n+1)}}$$

Solving for $x^{(n+1)}$ gives

$$\begin{aligned} f'\left(x^{(n)}\right) &= \frac{f\left(x^{(n)}\right)}{x^n - x^{(n+1)}} \\ x^n - x^{(n+1)} &= \frac{f\left(x^{(n)}\right)}{f'\left(x^{(n)}\right)} \\ x^{(n+1)} &= x^{(n)} - \frac{f(x^{(n)})}{f'\left(x^{(n)}\right)}. \end{aligned}$$

Definition 3.14. **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:

$$x^{(n+1)} = x^{(n)} - \frac{f'\left(x^{(n)}\right)}{f''\left(x^{(n)}\right)}$$

- By Taylor expansion, we can have $f\left(x^{(n)} + t\right)$ decomposed as following:

$$f\left(x^{(n)} + t\right) \approx f(x^{(n)}) + f'\left(x^{(n)}\right) t + \frac{1}{2} f''\left(x^{(n)}\right) t^2.$$

  By taking the derivative of $f\left(x^{(n)} + t\right)$ w.r.t. t, namely find the t can make $f\left(x^{(n)} + t\right)$ as small as possible, we have

$$\begin{aligned} \frac{d}{dt} f\left(x^{(n)} + t\right) &\approx \frac{d}{dt}\left(f(x^{(n)}) + f'\left(x^{(n)}\right) t + \frac{1}{2} f''\left(x^{(n)}\right) t^2\right) \\ &= f'\left(x^{(n)}\right) + f''\left(x^{(n)}\right) t. \end{aligned}$$

  Making the derivative equal to 0 to find the stationary point

$$\begin{aligned} 0 &= f'\left(x^{(n)}\right) + f''\left(x^{(n)}\right) t \\ t &= -\frac{f'\left(x^{(n)}\right)}{f''\left(x^{(n)}\right)} \end{aligned}$$

  Hence we can have the following update rule:

$$x^{(n+1)} = x^{(n)} + t = x^{(n)} - \frac{f'\left(x^{(n)}\right)}{f''\left(x^{(n)}\right)},$$

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

Remark 3.14. *For more than one dimension case, the update rule remains similar but replacing f' with ∇f and $1/f''$ with $(\nabla^2 f)^{-1}$, namely Hessian matrix inverse H_f^{-1}.*

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

Remark 3.15. *We now consider the implications of an indefinite Hessian at a critical point.*[14] *Suppose that $f(\mathbf{x})$ has continuous second partial derivatives on a set $\mathbb{D} \subseteq \mathbb{R}^n$. Furthermore, let $\mathbf{x}^*$ be an interior point of $\mathbb{D}$ that is a critical point (could be either a local maximum, a local minimum, or a saddle point) of $f(\mathbf{x})$. If $\mathbf{H}_f(\mathbf{x}^*)$ is indefinite, then there exist vectors $\mathbf{u}, \mathbf{v}$ such that*

$$\mathbf{u}^\top \mathbf{H}_f(\mathbf{x}^*)\mathbf{u} > 0, \mathbf{v}^\top \mathbf{H}_f(\mathbf{x}^*)\mathbf{v} < 0$$

By continuity of the second partial derivatives, there exists an $\epsilon > 0$ such that

$$\mathbf{u}^\top \mathbf{H}_f(\mathbf{x}^* + t\mathbf{u})\mathbf{u} > 0, \mathbf{v}^\top \mathbf{H}_f(\mathbf{x}^* + t\mathbf{v})\mathbf{v} < 0$$

for $|t| < \epsilon$. If we define

$$\begin{aligned}
U(t) &= f(\mathbf{x}^* + t\mathbf{u}), \\
U'(t) &= \frac{\partial f(\mathbf{x}^* + t\mathbf{u})}{\partial t} = \frac{\partial f(\mathbf{x}^* + t\mathbf{u})}{\partial (\mathbf{x}^* + t\mathbf{u})} \cdot \frac{\partial (\mathbf{x}^* + t\mathbf{u})}{\partial t} = \nabla f(\mathbf{x}^* + t\mathbf{u})^\top \mathbf{u}, \\
U''(t) &= \frac{\partial \nabla f(\mathbf{x}^* + t\mathbf{u})^\top \mathbf{u}}{\partial t} = \mathbf{u}^\top \cdot \frac{\partial \nabla f(\mathbf{x}^* + t\mathbf{u})}{\partial (\mathbf{x}^* + t\mathbf{u})} \cdot \frac{\partial (\mathbf{x}^* + t\mathbf{u})}{\partial t} = \mathbf{u}^\top \nabla^2 f(\mathbf{x}^* + t\mathbf{u})\mathbf{u} \\
V(t) &= f(\mathbf{x}^* + t\mathbf{v}), \\
V'(t) &= \frac{\partial f(\mathbf{x}^* + t\mathbf{v})}{\partial t} = \frac{\partial f(\mathbf{x}^* + t\mathbf{v})}{\partial (\mathbf{x}^* + t\mathbf{v})} \cdot \frac{\partial (\mathbf{x}^* + t\mathbf{v})}{\partial t} = \nabla f(\mathbf{x}^* + t\mathbf{v})^\top \mathbf{v}, \\
V''(t) &= \frac{\partial \nabla f(\mathbf{x}^* + t\mathbf{v})^\top \mathbf{v}}{\partial t} = \mathbf{v}^\top \cdot \frac{\partial \nabla f(\mathbf{x}^* + t\mathbf{v})}{\partial (\mathbf{x}^* + t\mathbf{v})} \cdot \frac{\partial (\mathbf{x}^* + t\mathbf{v})}{\partial t} = \mathbf{v}^\top \nabla^2 f(\mathbf{x}^* + t\mathbf{v})\mathbf{v}
\end{aligned}$$

then we have $U'(0) = \nabla f(\mathbf{x}^*)^\top \mathbf{u} = 0, V'(0) = \nabla f(\mathbf{x}^*)^\top \mathbf{v} = 0$ (as $\mathbf{x}^*$ is a critical point of $f(\mathbf{x})$), while $U''(0) = \mathbf{u}^\top \nabla^2 f(\mathbf{x}^* + t\mathbf{u})\mathbf{u} > 0$ and $V''(0) = \mathbf{v}^\top \nabla^2 f(\mathbf{x}^* + t\mathbf{v})\mathbf{v} < 0$ (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 $\mathbf{x}^*$ is a saddle point of f.

Remark 3.16. *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.*
- *It can converge to a saddle point instead of to a local minimum if the Hessian is not a positive definite matrix.*

[14] `https://www.math.usm.edu/lambers/mat419/lecture3.pdf`

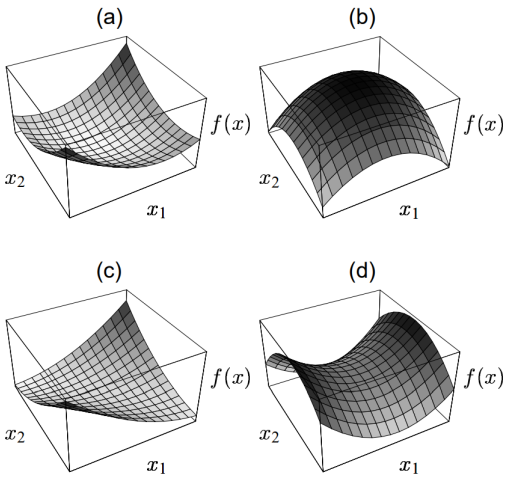


Figure 13: (a) Quadratic form for a positive-definite matrix. (b) For a negative-definite matrix. (c) For a singular (and positive-indefinite) matrix. A line that runs through the bottom of the valley is the set of solutions. (d) For an indefinite matrix. Because the solution is a saddle point, the steepest descent and conjugate gradient will not work. In three dimensions or higher, a singular matrix can also have a saddle.

Definition 3.15. **Broyden–Fletcher–Goldfarb–Shanno (BFGS)** algorithm is an iterative method for solving unconstrained nonlinear optimization problems. From an initial guess $\mathbf{x}^{(0)}$ and an approximate Hessian matrix $\mathbf{H}^{(0)}$ the following steps are repeated as $\mathbf{x}^{(k)}$ converges to the solution:

1. Obtain a direction $\mathbf{p}^{(k)}$ by solving $\mathbf{p}^{(k)} = -{\mathbf{H}^{(k)}}^{-1}\nabla f(\mathbf{x}^{(k)})$ (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 $\alpha^{(k)} = \arg\min f(\mathbf{x}^{(k)} + \alpha\mathbf{p}^{(k)})$. In practice, an inexact line search usually suffices, with an acceptable $\alpha^{(k)}$ satisfying Wolfe conditions.
3. $\mathbf{s}^{(k)} = \alpha^{(k)}\mathbf{p}^{(k)}$ and update $\mathbf{x}_{k+1} = \mathbf{x}^{(k)} + \mathbf{s}^{(k)}$.
4. $\mathbf{y}^{(k)} = \nabla f(\mathbf{x}^{(k+1)}) - \nabla f(\mathbf{x}^{(k)})$.
5. $\mathbf{H}^{(k+1)} = \mathbf{H}^{(k)} + \frac{\mathbf{y}^{(k)}{\mathbf{y}^{(k)}}^\top}{{\mathbf{y}^{(k)}}^\top\mathbf{s}^{(k)}} - \frac{\mathbf{H}^{(k)}\mathbf{s}^{(k)}{\mathbf{s}^{(k)}}^\top{\mathbf{H}^{(k)}}^\top}{{\mathbf{s}^{(k)}}^\top\mathbf{H}^{(k)}\mathbf{s}^{(k)}}$.

3.2 Misc in Deep Learning

3.2.1 Techniques

Definition 3.16. **Transfer learning** aims to help improve the learning of the target predictive function $f_T(\cdot)$ in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$, where a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}_S$, a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, where $\mathcal{D}_S \neq \mathcal{D}_T$, or $\mathcal{T}_S \neq \mathcal{T}_T$.

Definition 3.17. **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 3.18. **Cross-validation** is a resampling procedure used to evaluate machine learning models on a limited data sample.

Example 26. 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 3.17. *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.*

3.2.2 Common terms

Definition 3.19. **Representation** refers to the way data is encoded or structured to be processed by a model. It captures the relevant aspects or features of the data that are essential for solving a specific task.

Example 27. A representation can be in various forms, such as numerical features, symbolic representations, or even raw data like pixels. For example, in image classification, an image representation can be a set of numerical features extracted from the image, or it can be the raw pixel values themselves.

Definition 3.20. **Feature** is a measurable property or characteristic of the data that is relevant to the task at hand. Features are derived from the raw data and serve as input variables for a learning algorithm.

Example 28. Features are often represented as numerical values but can also be categorical or binary. Features can be simple, such as individual measurements or attributes, or they can be more complex, such as combinations of multiple measurements or derived from domain knowledge.

Definition 3.21. **Embedding** is a learned representation of data in a lower-dimensional space, which high-dimensional data into a more compact and meaningful representation. Embeddings are commonly used in natural language processing (NLP) tasks, where words or sentences are mapped to numerical vectors in a continuous vector space.

Example 29. Word embeddings such as `Word2Vec` or `GloVe` represent words as dense vectors, capturing semantic relationships between words based on their context. Embeddings can also be used for other types of data, such as images or user behavior in recommendation systems.

Remark 3.18. ***Two different perspectives of manifold learning: embedding and metric. [Arvanitidis et al., 2016]***

Traditionally, manifold learning is seen as an embedding problem where a low-dimensional representation of the data is sought. In embedding approaches, the relation between a new point and the embedded points are less well-defined, and consequently these approaches are less suited for building generative models.

In contrast, the Riemannian metric approach gives the ability to measure continuous geodesics that follow the structure of the data. This makes the learned Riemannian manifold a suitable space for a generative model.

Definition 3.22. One **batch** of samples is a group of samples concatenated together to go through the network before updating the model weights.

Types	Batch size
Stochastic gradient descent	1
Mini-Batch gradient descent	$(1, \text{size of training dataset})$
Batch gradient descent	size of training dataset

Example 30. 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 uninterruptedly.

Definition 3.23. One **iteration** is a time span when a batch of training data is passed forward and backward through the neural network.

Definition 3.24. One **epoch** is a time span when an entire training dataset is passed forward and backward through the neural network.

Definition 3.25. In **convolutional layer**, multiple channels of feature maps were extracted by sliding the trainable convolutional kernels across the input feature maps.

Example 31. 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.

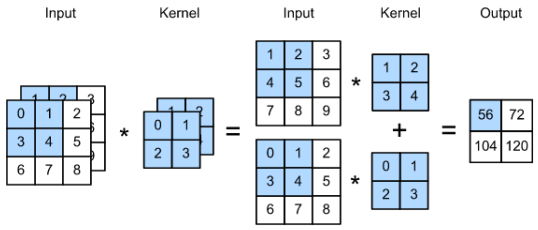


Figure 14: Assuming the input has already been flipped both horizontally and vertically (but unchanged in channel dimension), therefore the correlation above is equivalent to convolution. Courtesy of [Zhang et al., 2021].

Definition 3.26. **Low pass filtering** (smoothing), is employed to remove high spatial frequency noise from a digital image.

Definition 3.27. **High pass filtering** (edge detection, sharpening) can be used to make an image appear sharper. These filters emphasize fine details in the image - the opposite of the low-pass filter.

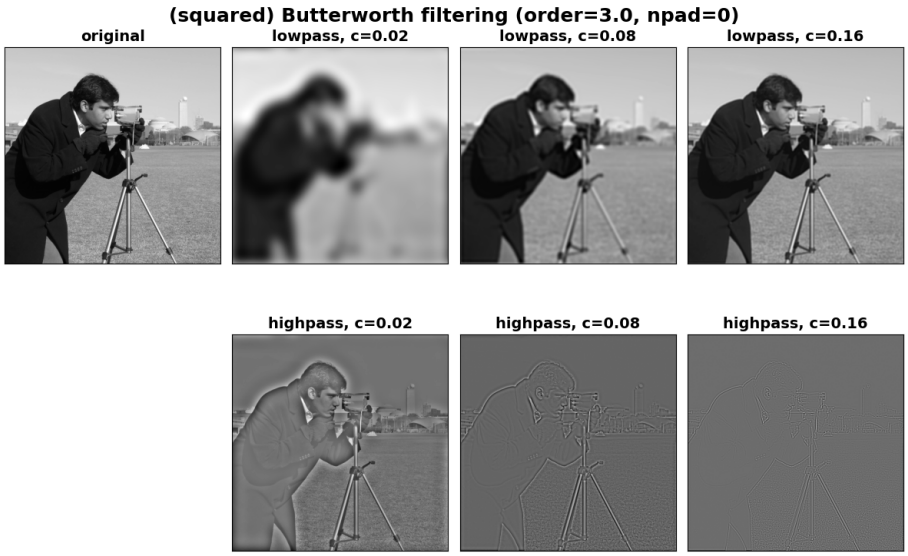


Figure 15: Examples of low and high pass filter

Definition 3.28. **Morphing** is a special effect in motion pictures and animations that changes (or morphs) one image or shape into another through a seamless transition.

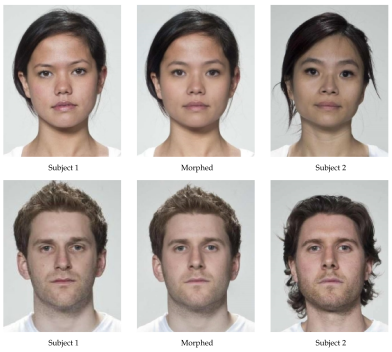


Figure 16: Morphing example

Definition 3.29. In **transposed convolutional (deconvolution) layer**[15], multiple channels of feature maps were extracted by sliding the trainable convolutional kernels across the input feature maps with the special setting of stride and padding. Pixels in input serve as the weight to the convolution filter, with the overlapping region adding up the values from differently scaled filter.

[15]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.

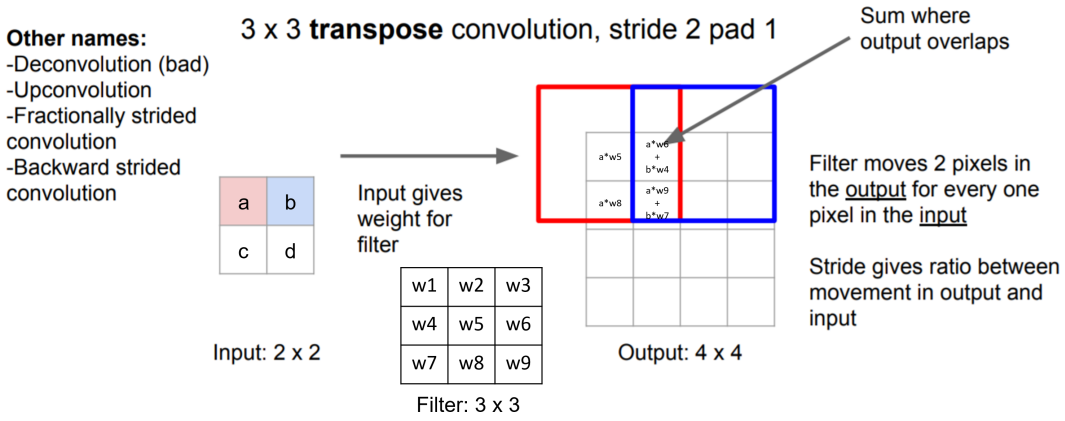


Figure 17: Deconvolution: Input tensor provides coefficients for a convolutional filter. `http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture11.pdf`

Remark 3.19. *Deconvolution can easily have "uneven overlap", putting more of the metaphorical paint in some places than others. In particular, deconvolution has uneven overlap when the kernel size (the output window size) is not divisible by the stride (the spacing between points on the top). While the network could, in principle, carefully learn weights to avoid this, in reality neural networks struggle to avoid it completely.*

The overlap pattern also forms in two dimensions. The uneven overlaps on the two axes multiply together, creating a characteristic checkerboard-like pattern of varying magnitudes.

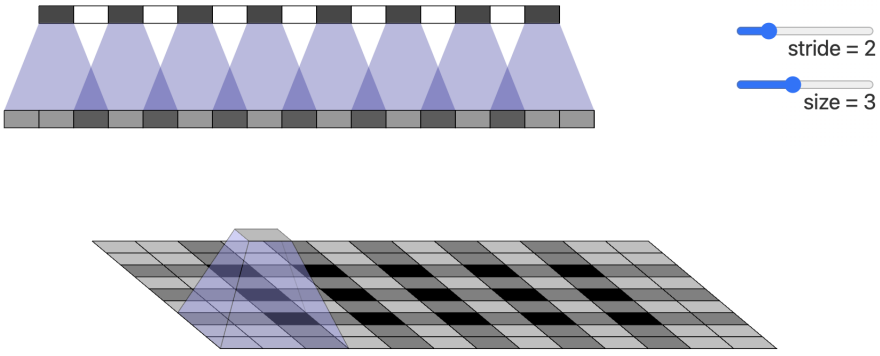


Figure 18: Uneven overlapping in deconvolution. [Odena et al., 2016]

Definition 3.30. **Resize-convolutional layer** interpolates the low resolution feature via nearest-neighbor or bilinear interpolation, then convoluting the interpolated tensor with learnable convolution filter.

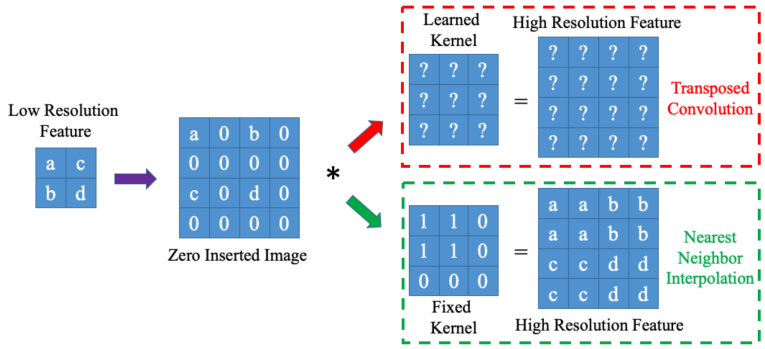


Figure 19: Resize-convolution. [Zhang et al., 2019]

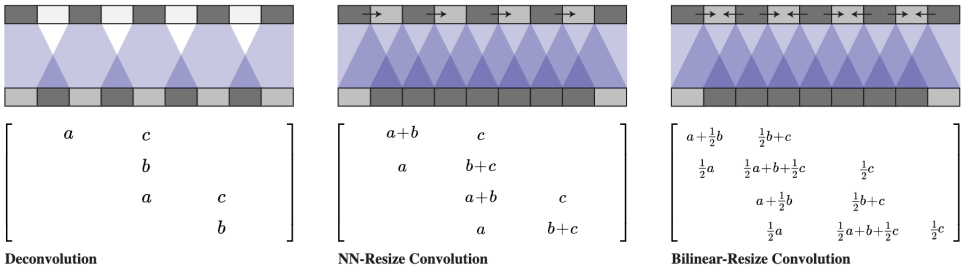


Figure 20: Deconvolution vs. resize-convolution. [Odena et al., 2016]

Figure 21: Simply switching out the standard deconvolutional layers for nearest-neighbor resize-convolution causes artifacts of different frequencies to disappear. [Odena et al., 2016]

Definition 3.31. **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 3.20. *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 3.21. *The pooling layer can also have configurations of padding and stride.*

Definition 3.32. **Global average pooling (GAP)** [Lin et al., 2013] operates by taking the average of all the values in each channel of the feature map. This is done by computing the mean value along the spatial dimensions (e.g., height and width) while preserving the number of channels.

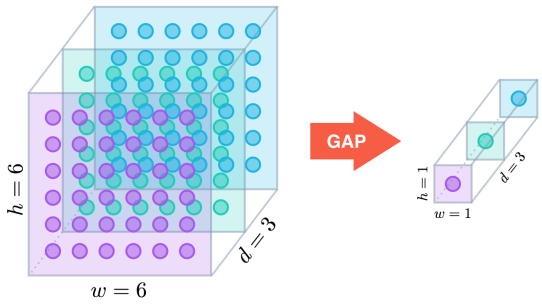


Figure 22: Illustration of global average pooling

Remark 3.22. *Global Average Pooling (GAP) vs. fully connected (FC) layers in the context of convolutional neural networks classification task:*

- *Parameter Efficiency: GAP significantly reduces the number of parameters compared to fully connected layers. In a fully connected layer, the number of parameters grows rapidly with the spatial dimensions of the feature map. In contrast, GAP computes a fixed number of parameters per channel, regardless of the input size. This parameter efficiency makes GAP more computationally efficient and helps prevent overfitting, especially in models with limited training data.*
- *Spatial Invariance: GAP promotes translation invariance in the feature representation. By averaging the activations over the entire spatial extent, GAP generates a fixed-length representation that is insensitive to small translations of the objects within the input image. This property makes the model more robust to slight shifts in object positions and improves generalization.*
- *Interpretability: GAP provides a more interpretable representation compared to fully connected layers. By taking the average of the feature map activations, GAP retains channel-wise information while discarding the spatial information. This encourages the network to focus on the most important and discriminative features rather than the precise spatial arrangement. The resulting feature representation is more compact and meaningful, making it easier to interpret and analyze.*
- *Spatial Localization: GAP can be used to obtain class activation maps, which provide insights into the important regions of an input image for a given class. By applying weights to the feature maps and combining them through GAP, it is possible to generate a heatmap that highlights the regions that contribute most to the prediction. This spatial localization information is valuable in tasks such as object detection and image segmentation.*
- *Computational Efficiency: GAP is computationally more efficient than fully connected layers. While fully connected layers require connecting every neuron to every other neuron, GAP operates on a per-channel basis, reducing the computational complexity significantly. This efficiency is especially beneficial in large-scale CNN architectures with millions of parameters, as it speeds up training and inference times.*

Remark 3.23. *Global Average Pooling (GAP) vs. Global Max Pooling (GMP):*

- *Robustness to Noise: Global Average Pooling is generally more robust to noise compared to Global Max Pooling. In Global Max Pooling, only the maximum activation value in each channel is considered, which can be sensitive to outliers or noise in the feature map. On the other hand, Global Average Pooling calculates the*

average activation value, which provides a more robust representation by considering the overall distribution of activations in the channel.

- *Spatial Information Preservation: Global Average Pooling preserves more spatial information than Global Max Pooling. In Global Average Pooling, the average activation value is calculated across the spatial dimensions, resulting in a spatially smoothed representation. This can be beneficial when spatial localization is important, as the pooled representation retains a coarse understanding of the spatial layout of the features. Global Max Pooling, however, only retains the maximum activation value, discarding information about the precise spatial location of the most activated feature.*
- *Gradient Propagation: Global Average Pooling tends to provide better gradient propagation compared to Global Max Pooling. When computing gradients during backpropagation, Global Average Pooling distributes the gradient equally across the spatial dimensions, which can help with smoother and more stable gradient flow. In Global Max Pooling, the gradient is only backpropagated to the locations that correspond to the maximum activation values, which can lead to sparse and less stable gradients.*
- *Class Activation Mapping: Global Average Pooling is more suitable for generating class activation maps (CAMs) compared to Global Max Pooling. CAMs provide visualizations of the regions in an image that are most important for a particular class prediction. Since Global Average Pooling computes a weighted average of the feature map activations, it naturally provides a heatmap that highlights the regions of importance. Global Max Pooling, however, does not directly provide spatial information about the contributing regions.*

Definition 3.33. **Class Activation Mapping (CAM)** [Zhou et al., 2016] is a technique used in computer vision tasks, particularly in the context of convolutional neural networks (CNNs), to visualize the regions of an input image that contribute the most to the prediction of a particular class. CAM is calculate by

$$\text{CAM}_c = \sum_k \mathbf{W}_{ck}\mathbf{A}_k$$

where $\mathbf{W}_{ck}$ is the learned weight from GAP layer to fully connected layer (theoretically $\mathbf{W}_{ck} = \sum_i \sum_j \frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}$), $\mathbf{A}_k$ is the k-th channel of the final activation feature map.

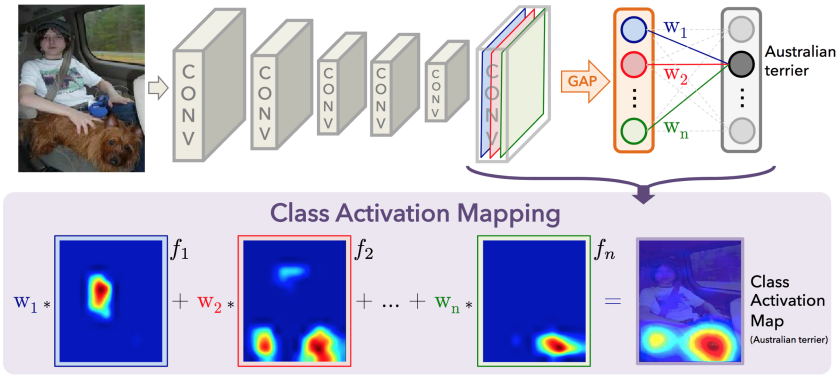


Figure 23: Illustration of class activation mapping, w_1, w_2, w_3 are the weights (blue, red, green line) w.r.t. category "Australian terrier".[Zhou et al., 2016]

Definition 3.34. **Gradient-weighted Class Activation Mapping (GradCAM)** [Selvaraju et al., 2019] is a technique similar to CAM. Instead of average the feature map by the GAP-yielded weight, it calculate the gradients flowing into the final convolutional layer of the CNN to produce a class activation map by

$$\text{GradCAM}_c = \text{ReLU}(\sum_k \mathbf{W}'_{ck}\mathbf{A}_k)$$

where $\mathbf{W}'_{ck} = \frac{1}{z}\sum_i \sum_j \frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}$, $\mathbf{A}_k$ is the k-th channel of the final activation feature map.

Remark 3.24. ***CAM vs. GradCAM.*** *CAM is based on global average pooling (GAP) and fully connected layers, while Grad-CAM generalize it to skip GAP layer by leveraging the average of the gradient map $\frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}$ to weight the feature map channels. CAM can be only applied to CNN with a GAP layer, while GradCAM does not require that.*

Remark 3.25. *GradCAM argues that the weight $\mathbf{W}_{ck}$ in standard CAM is nothing but the average of the gradient of $\mathbf{y}_c$ w.r.t. activation feature map $\mathbf{A}_{kij}$, which is how GradCAM calculate the weights.*

Proof. $\mathbf{y}_c \in \mathbb{R}$ represents the score of class c before softmax layer; $\mathbf{W}_{ck} \in \mathbb{R}$ is the weight from feature map k to class c; $\mathbf{A}_{kij} \in \mathbb{R}$ is the activation value (i, j) in the k-th channel of the final activation map; z is the total number of activation value in one channel of activation map. Based on the definition of CAM, we have

$$\mathbf{y}_c = \sum_k \mathbf{W}_{ck} \cdot \frac{1}{z} \sum_i \sum_j \mathbf{A}_{kij}$$

$$\mathbf{y}_c = \sum_k \mathbf{W}_{ck} \mathbf{F}_k \qquad \text{where } \mathbf{F}_k = \frac{1}{z} \sum_i \sum_j \mathbf{A}_{kij}$$

To take derivative of $\mathbf{y}_c$ w.r.t. $\mathbf{F}_k$, we have

$$\frac{\partial \mathbf{y}_c}{\partial \mathbf{F}_k} = \mathbf{W}_{ck} = \frac{\frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}}{\frac{\partial \mathbf{F}_k}{\partial \mathbf{A}_{kij}}} = \frac{\frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}}{\frac{1}{z}}$$

$$\mathbf{W}_{ck} = z \cdot \frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}$$

$$\sum_i \sum_j \mathbf{W}_{ck} = \sum_i \sum_j z \cdot \frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}$$

$$z\mathbf{W}_{ck} = z \sum_i \sum_j \frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}$$

$$\mathbf{W}_{ck} = \sum_i \sum_j \frac{\partial \mathbf{y}_c}{\partial \mathbf{A}_{kij}}$$

Namely the weight $\mathbf{W}_{ck}$ is equivalent to the sum of the gradient of $\mathbf{y}_c$ w.r.t. activation feature map $\mathbf{A}_{kij}$. This tells us that we can omit global average pooling (GAP) layer in calculation of CAM, which in a way accommodates more kinds of CNN architectures. □

Definition 3.35. **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 3.26. *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 first flatten the last feature map ($C \times H \times W$) into a 1D vector of length $C \times H \times W$ in both channel and spatial dimensions, while the final output is also a 1D vector which typically matches the number of the classification category (if in classification task), then it's actually the same as the FCN in the MLP.*

Remark 3.27. ***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 3.28. *Always remember the CNN or MLP are the model doing the calculation point-wisely, forget about the shape of the input tensor.*

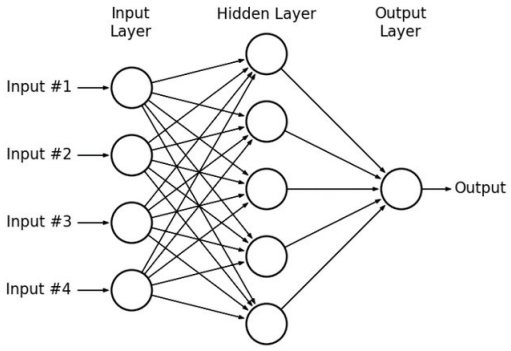


Figure 24: Simple illustration of a multilayer perceptron model. The first column of perceptrons in the figure stand for the four channels of one sample vector, rather than a flattened input tensor, which may consist of many sample points. The input tensor of four channels of any shape would fits this network, as the framework would do the calculation point-wisely. When it comes to a 3×3 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 3.29. ***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 32. An MLP implementation for images, by leveraging `Conv` layer and processing the pixels in an MLP manner all at once.

```
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. els

    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.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 3.30. *`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.*

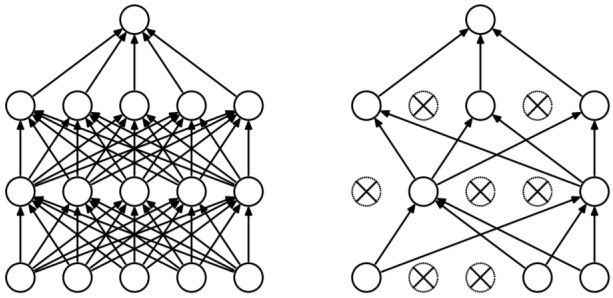
The batch size should be larger than the number of GPUs used.

Definition 3.36. **Dropout layer** is a technique to randomly drop units (along with their connections) from the neural network during training, which alleviates data overfitting.

Remark 3.31. *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.*

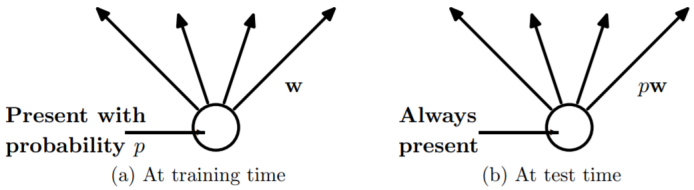
Remark 3.32. *Dropout has a tunable hyperparameter p (the probability of retaining a unit in the network). For any layer l, $\mathbf{r}^{(l)}$ is a vector of independent Bernoulli random variables each of which has probability p of being 1. Namely*

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



(a) Standard Neural Net (b) After applying dropout.

Figure 25: 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 [Srivastava et al., 2014].



(a) At training time (b) At test time

Figure 26: Left: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. 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 [Srivastava et al., 2014].

Definition 3.37. **Unsupervised learning algorithms** experience a dataset containing many features, then learn useful properties of the structure of this dataset.

Definition 3.38. **Supervised learning algorithms** experience a dataset containing features, but each example is also associated with a label or target.

Definition 3.39. **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 3.33. *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 3.40. **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 33. 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[hyun Lee,] proposes using weights:

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

Definition 3.41. **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 3.42. **Overfitting** is the phenomenon when the gap between the training error and testing error is too large. Over fitting 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 34. Methods to avoid overfitting: regularization, dropout, pruning, and data augmentation.

Definition 3.43. **Bias** is a learner's tendency to consistently learn the same wrong thing [Domingos, 2012]. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting), e.g. decision tree.

Definition 3.44. **Variance** is the tendency to learn random things irrespective of the real signal [Domingos, 2012]. The high variance may result from an algorithm modeling the random noise in the training data (overfitting), e.g. random forest.

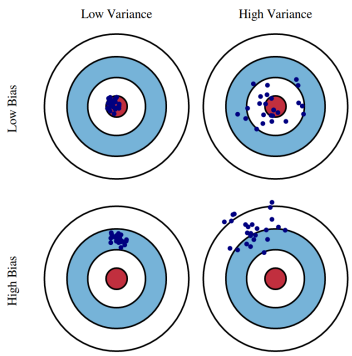


Figure 27: Variance and bias

Definition 3.45. **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 3.46. **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 3.47. **L^1 regularization** adds L^1 penalty to the loss function:

$$\mathcal{L} = \text{error}(\mathbf{v}_{\text{true}} - \mathbf{v}_{\text{pred}}) + \lambda \sum \|\theta\|$$

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

Definition 3.48. **L^2 regularization** adds L^2 penalty to the loss function:

$$\mathcal{L} = \text{error}(\mathbf{v}_{\text{true}} - \mathbf{v}_{\text{pred}}) + \lambda \sum \|\theta\|^2$$

Remark 3.34. ***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 35. Weight decay is one kind of regularizer in linear regression, which expresses a preference for the weights to have smaller squared norm:

$$L(\theta) = \text{MSE}_{\text{train}} + \lambda \theta^\top \theta$$

Definition 3.49. **Generative model** is a class of statistical models that can generate new data instances. These models are used in unsupervised machine learning.

Example 36. Naïve Bayes, Bayesian networks, Markov random fields, hidden Markov models (HMMs), latent Dirichlet allocation (LDA), and generative adversarial networks (GANs).

Definition 3.50. **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 37. Logistic regression, SVM, neural networks, k-NN, CRF, decision trees, and random forest.

Definition 3.51. **Gradient vanishing** occurs when the derivative or slope will get smaller and smaller as we go backward with every layer during backpropagation.

Remark 3.35. *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 3.52. **Gradient explosion** occurs when the derivatives or slope will get larger and larger as we go backward with every layer during backpropagation.

Remark 3.36. *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 3: Parameter number calculation for different types of layer

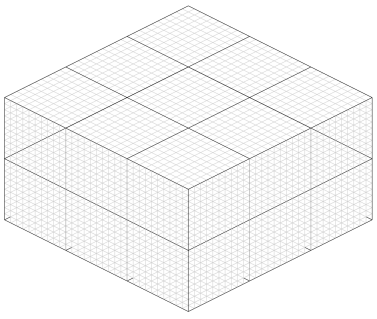
Layer	Parameters number
(Transposed-)Convolutional Layer	$(\text{input channel \#} \times \text{kernel size} + 1(bias)) \times \text{output channel \#}$
Pooling Layer	0
Dropout Layer	0
Activation Layer	0(others) or 1(α in ELU)
Linear Layer	$(\text{input channel \#} \times 1 + 1(bias)) \times \text{output channel \#}$
Batch Normalization Layer	$\text{input channel \#} \times 4$

Definition 3.53. **Curse of dimensionality** [Domingos, 2012] 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 3.37. *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 3.38. *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 38. In high dimensions, all examples look alike. Suppose, for instance, that examples are laid out on a regular grid, and consider a test example $\mathbf{s}^{(t)}$. If the grid is d-dimensional, $\mathbf{s}^{(t)}$'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 $\mathbf{s}^{(t)}$, until the choice of nearest neighbor (and therefore of class) is effectively random.



3.2.3 Loss Functions

Definition 3.54. **Confusion matrix** is a 2×2 table that contains 4 outputs provided by the binary classifier.

Ground truth \ Prediction	Positive	Negative
Positive	True positive(TP)	False negative(FN)
Negative	False positive(FP)	True negative(TN)

$$recall = \frac{TP}{TP + FN}$$

$$precision = \frac{TP}{TP + FP}$$

Definition 3.55. **Precision** is the percentage of positive identifications that were actually correct.

$$\text{precision} = \frac{TP}{TP + FP}$$

Definition 3.56. **Recall** is the percentage of actual positive that was identified correctly.

$$\text{recall} = \frac{TP}{TP + FN}$$

Remark 3.39. *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.*

$$PPV = \frac{TP}{TP + FP}$$

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

$$NPV = \frac{TN}{TN + FN}$$

Remark 3.40. *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.*

$$\text{Sensitivity} = \frac{TP}{TP + FN}$$

- ***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.*

$$\text{Specificity} = \frac{TN}{FP + TN}$$

Definition 3.57. **Accuracy** is the fraction of predictions identified correctly.

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

Definition 3.58. **F-score** is the harmonic mean of precision and recall:

$$\mathrm{F}_\beta = \frac{(1 + \beta^2) \cdot TP}{(1 + \beta^2) \cdot TP + \beta^2 \cdot FN + FP}$$

where β is chosen s.t. recall is considered β times as important as precision.

Example 39. F_1 score is a special case of F-score

$$\mathrm{F}_1 = \frac{2 \times TP}{2 \times TP + FN + FP}$$

Definition 3.59. **Receiver Operating Characteristic curve (ROC curve)** plots true positive rate(TPR) vs. false positive rate (FPR) at different classification thresholds, where

$$\text{TPR} = \frac{TP}{TP + FN} = \text{recall of positive samples}$$
$$\text{FPR} = \frac{FP}{FP + TN} = 1 - \frac{TN}{TN + FP} = 1 - \text{recall of negative samples}$$

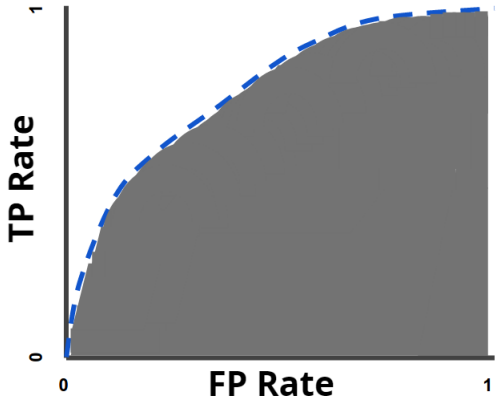


Figure 28: ROC and AUC.

Remark 3.41. *Lowering the classification threshold (corresponds to the point on the top right end of the dashed line in Figure 28) means classifying more samples into positive category (increasing the quantity of the TP and FP), which is equivalent to higher TPR (positive recall) and higher FPR (1 - negative recall).*

Definition 3.60. **Area under the ROC Curve (AUC)** measures the entire two-dimensional area underneath the entire ROC curve. AUC also represents the probability that a random positive (green) example is positioned to the right of a random negative (red) example, which can be calculated explicitly by the formula below

$$\text{AUC}(f) = \frac{\sum_{\mathbf{t}_0 \in \mathcal{D}^F} \sum_{t_1 \in \mathcal{D}^P} \mathbf{1}[f(t_0) < f(t_1)]}{|\mathcal{D}^F| \cdot |\mathcal{D}^P|},$$

where $\mathbf{1}[f(s_0) < f(s_1)]$ denotes an indicator function which returns 1 if $f(t_0) < f(t_1)$ otherwise return 0; $\mathcal{D}^F$ is the set of negative examples, and $\mathcal{D}^P$ is the set of positive examples.

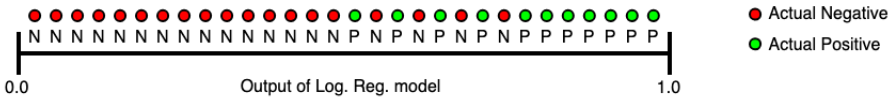


Figure 29: Predictions ranked in ascending order of logistic regression score. AUC represents the probability that a random positive (green) example is positioned to the right of a random negative (red) example.

Remark 3.42. *AUC is desirable for the following two reasons:*[16]

- *AUC is scale-invariant. It measures how well predictions are ranked, rather than their absolute values.*
- *AUC is classification-threshold-invariant. It measures the quality of the model's predictions irrespective of what classification threshold is chosen.*

[16] `https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc#:~:text=An%20ROC%20curve%20(receiver%20operating,False%20Positive%20Rate`

However, both these reasons come with caveats, which may limit the usefulness of AUC in certain use cases:

- *Scale invariance is not always desirable. For example, sometimes we really do need well calibrated probability outputs, and AUC won't tell us about that.*
- *Classification-threshold invariance is not always desirable. In cases where there are wide disparities in the cost of false negatives vs. false positives, it may be critical to minimize one type of classification error. For example, when doing email spam detection, you likely want to prioritize minimizing false positives (even if that results in a significant increase of false negatives). AUC isn't a useful metric for this type of optimization.*

Remark 3.43. *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.*

Definition 3.61. **Intersection of Union (Jaccard index, IoU)** is defined as

$$\text{IoU} = \frac{|\mathbb{A} \cap \mathbb{B}|}{|\mathbb{A} \cup \mathbb{B}|} = \frac{TP}{TP + FP + FN}$$

Definition 3.62. **Dice loss function** is defined as

$$\mathcal{L}_{\text{dice}} = -\frac{2|\mathbb{A} \cap \mathbb{B}|}{|\mathbb{A}| + |\mathbb{B}|} = \frac{TP}{2 \times TP + FP + FN} = -\frac{2 \times \langle \mathbf{v}_{\text{true}}, \mathbf{v}_{\text{pred}} \rangle}{\|\mathbf{v}_{\text{true}}\|_2^2 + \|\mathbf{v}_{\text{pred}}\|_2^2 + \epsilon} \in (-1, 0],$$

where $\mathbf{v}_{\text{true}}, \mathbf{v}_{\text{pred}} \in \mathbb{R}^{h \times w \times d}$ are **one-hot vectors**, and ϵ is a small constant to avoid zero division.

Remark 3.44. *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 3.63. L^1 **loss function** is defined as

$$\begin{aligned} \mathcal{L}_{L^1} &= \|\mathbf{v}_{\text{true}} - \mathbf{v}_{\text{pred}}\|_1, \\ &= \sum_{i=1}^{n} |\mathbf{v}_i^{\text{pred}} - \mathbf{v}_i^{\text{true}}|, \end{aligned}$$

where $\mathbf{v}_{\text{true}}, \mathbf{v}_{\text{pred}}$ are vectors.

Remark 3.45. *In image translation tasks, the choice between L^1 and L^2 loss depends on the desired characteristics of the generated images.*

- *If preserving fine details and sharpness is important, L^1 loss is often preferred as it tends to produce sharper results. Since the L^1 loss computes the absolute difference between pixel values, it is less affected by outliers or large errors. This property allows L^1 loss to preserve fine details in the generated image.*
- *If obtaining smoother or more globally consistent results is desired, L^2 loss can be used, as it calculates the squared difference between pixel values, which amplifies the impact of larger errors. This means that even small differences in pixel values can lead to a relatively higher loss, making L^2 loss more likely to produce smoother or blurry results.*

On the other hand,

Definition 3.64. L^2 **loss function** is defined as

$$\begin{aligned} \mathcal{L}_{L^2} &= \|\mathbf{v}_{\text{true}} - \mathbf{v}_{\text{pred}}\|_2^2, \\ &= \sqrt{\sum_{i=1}^{n} |\mathbf{v}_i^{\text{pred}} - \mathbf{v}_i^{\text{true}}|^2}^2, \\ &= \sum_{i=1}^{n} |\mathbf{v}_i^{\text{pred}} - \mathbf{v}_i^{\text{true}}|^2, \end{aligned}$$

where $\mathbf{v}_{\text{true}}, \mathbf{v}_{\text{pred}}$ are vectors.

Remark 3.46. *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 $\Pr_{\text{model}}(X^{(i)} = x^{(i)}; \theta) = \mathcal{N}(\hat{x^{(i)}}, \sigma^2)$, the mean of which is the true value $\hat{\mathbf{x}^{(i)}}$. Then we can have

$$\begin{aligned}
\theta_{\text{ML}} &= \arg\max_{\theta} \prod_{i}^{n} P_{\text{model}}(X = \hat{x^{(i)}}; \theta) \\
&= \arg\max_{\theta} \sum_{i}^{n} \log(P_{\text{model}}(X = \hat{x^{(i)}}; \theta)) \\
&= \arg\max_{\theta} \sum_{i}^{n} \log\left(\frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x^{(i)} - \hat{x^{(i)}})^2}{2\sigma^2}\right)\right) \\
&= \arg\max_{\theta} \sum_{i}^{n} \left(-\log(\sqrt{2\pi}\sigma) - \frac{(x^{(i)} - \hat{x^{(i)}})^2}{2\sigma^2}\right) \\
&= \arg\min_{\theta} \left(\sum_{i}^{n} \left((x^{(i)} - \hat{x^{(i)}})^2\right)\right) \qquad \triangleright\ L^2 \text{ loss}
\end{aligned}$$

□

Definition 3.65. **Mean squared error (MSE) loss function** is defined as

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{v}_i^{\text{pred}} - \mathbf{v}_i^{\text{true}})^2.$$

Remark 3.47.

$$\mathcal{L}^2 \textit{ norm} \xrightarrow{squared} \mathcal{L}^2 \textit{ loss} \xrightarrow{averaged} \textit{MSE loss}$$

Definition 3.66. The **least-squares method** finds the optimal parameter values by minimizing the sum of squared residuals

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

where $\mathbf{r}^{(i)} = \mathbf{y}^{(i)} - f(\mathbf{x}^{(i)}, \theta)$.

Remark 3.48. *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 3.67. **Hausdorff distance** is defined as

$$d_{\text{H}}(\mathcal{X}, \mathcal{Y}) = \max\left\{ \sup_{\mathbf{x} \in \mathcal{X}} \inf_{\mathbf{y} \in \mathcal{Y}} d(\mathbf{x}, \mathbf{y}),\ \sup_{\mathbf{y} \in Y} \inf_{\mathbf{x} \in X} d(\mathbf{x}, \mathbf{y}) \right\}$$

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

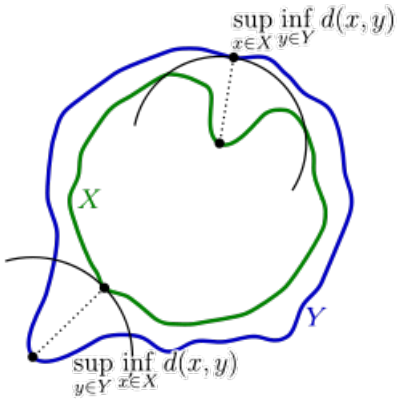


Figure 30: Components of the calculation of the Hausdorff distance between the green line $\mathcal{X}$ and the blue line $\mathcal{Y}$.

Definition 3.68. **Entropy** (a state of disorder, randomness, or uncertainty) of the **discrete probability distribution** p is defined as

$$H(p) = -E(\log p(x)) = -\sum_{x\in\mathcal{X}} p(x)\log p(x) \in [0,1]$$

Remark 3.49. *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).*

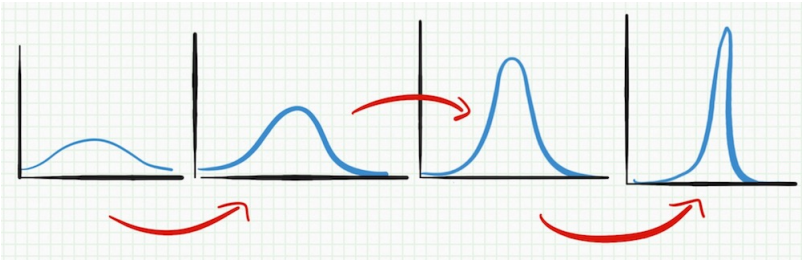


Figure 31: From left to right: high entropy to low entropy.

Definition 3.69. **Cross-entropy loss function** of the **discrete probability distribution** p and q over a given set is defined as

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

which measures the distance between the distributions.

Remark 3.50. *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 3.51. *If there are only two categories, then it is called binary cross-entropy (BCE).*

Remark 3.52. *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[17], the likelihood of the parameters θ of the model $q(X = c_i; \theta)$ on the training set is

$$P_{\text{model}}(x; \theta) = q(X = c_i)$$

We want to find parameter θ s.t. to maximize the likelihood, then we have

$$\begin{aligned}
\theta_{\text{ML}} &= \arg\max_{\theta} \left(\underbrace{q(X = c_1) \times \cdots \times q(X = c_1)}_{p(X=c_1) \text{ times}} \times \underbrace{q(X = c_2) \times \cdots \times q(X = c_2)}_{p(X=c_2) \text{ times}} \times \cdots \right) \\
&= \arg\max_{\theta} \left(\prod_i^C q(X = c_i)^{p(X=c_i)} \right) && \triangleright\ C\text{: category number} \\
&= \arg\max_{\theta} \left(\sum_i^C \log \left(q(X = c_i)^{p(X=c_i)} \right) \right) \\
&= \arg\max_{\theta} \left(\sum_i^C (p(X = c_i) \log(q(X = c_i)) \right) && \triangleright\ \text{Opposite of cross-entropy}
\end{aligned}$$

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 40. As for the two discrete distributions below:

Distribution	Class A	Class B	Class C
p	0	0	1
q	0.1	0.2	0.7

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 3.53. *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.*

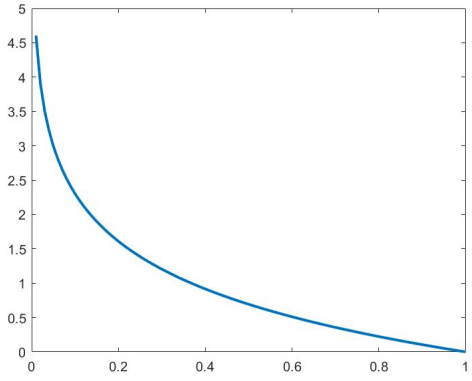


Figure 32: $y = -\log(x)$

[17] The training set we have is actually derived from real-world sampling.

Definition 3.70. **Triplet loss function** is defined as

$$\mathcal{L}\left(\mathbf{x}^A, \mathbf{x}^P, \mathbf{x}^N\right) = \max\left(\left\|f\left(\mathbf{x}^A\right) - f\left(\mathbf{x}^P\right)\right\|^2 - \left\|f\left(\mathbf{x}^A\right) - f\left(\mathbf{x}^N\right)\right\|^2 + \alpha, 0\right)$$

where $\mathbf{x}^A$ is an anchor input, $\mathbf{x}^P$ is a positive input of the same class as $\mathbf{x}^A$, $\mathbf{x}^N$ is a negative input of a different class from $\mathbf{x}^A$, α is a margin between positive and negative pairs, and f is an embedding function.

Definition 3.71. **Hinge loss function** is defined as

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

When y_{true} and y_{pred} have the same sign (meaning y_{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_{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 3.72. **Kullback–Leibler(KL) divergence** is defined as

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

Remark 3.54. *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 41. As for the two discrete distributions below:

Distribution	Class A	Class B	Class C
p	0	0	1
q_1	0.1	0.2	0.7
q_2	$1 - 1^{-\infty}$	0	$1^{-\infty}$

we can have the KL divergence

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

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

Definition 3.73. **Jensen–Shannon(JS) divergence** is defined as

$$D_{\text{JS}}(p, q) = \frac{1}{2} D_{\text{KL}}\left(p, \frac{1}{2}(p + q)\right) + \frac{1}{2} D_{\text{KL}}\left(q, \frac{1}{2}(p + q)\right)$$

Definition 3.74. **Wasserstein distance** is defined as

$$\begin{aligned} W_p(\mu, \nu) &= \left(\inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} \text{dist}(x, y)^p \, \mathrm{d}\gamma(x, y)\right)^{1/p}, \\ &= \left(\inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} \text{dist}(x, y)^p \gamma(x, y) \mathrm{d}x \mathrm{d}y\right)^{1/p}, \\ W_2(\mu, \nu) &= \left(\inf_{\gamma \in \Gamma(\mu, \nu)} \int_{M \times M} \text{dist}(x, y)^2 \gamma(x, y) \mathrm{d}x \mathrm{d}y\right)^{1/2}, \end{aligned}$$

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

In the discrete case, the distance reads as

$$W_p(\mu, \nu) = \min_{T \in \Gamma(\mu,\nu)} \langle T, M_{\mu\nu} \rangle = \min_{T \in \Gamma(\mu,\nu)} \mathrm{Tr}(T^\top M_{\mu\nu}),$$

where $M_{\mu\nu} = [\mathrm{dist}(x^{(i)}, y^{(j)})^p]_{ij} \in \mathbb{R}^{m \times n}$ and $\Gamma(\mu, \nu) = \{T \in \mathbb{R}_+^{m \times n} | T\mathbb{1}_m = a, T^\top \mathbb{1}_n = b\}$, 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 $\mathrm{dist}(x, y)$ and $\gamma(x, y)$ in continuous form.

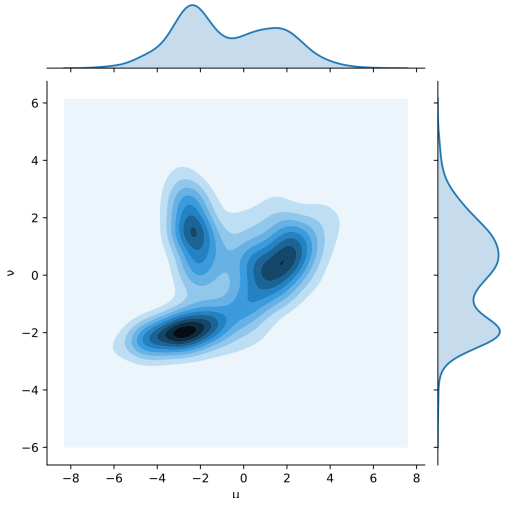


Figure 33: Wasserstein distance, credits to Wikipedia

Remark 3.55. *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 μ and ν on $\mathbb{R}^1$ has the following closed-form expression:*

$$\begin{aligned} W_p(\mu, \nu) &= \left(\int_{-\infty}^{+\infty} |U(s) - V(s)|^p ds \right)^{1/p} && \triangleright \int \textit{quantity} \times \textit{unit distance} \\ &= \left(\int_0^1 |U^{-1}(t) - V^{-1}(t)|^p dt \right)^{1/p}, && \triangleright \int \textit{distance} \times \textit{unit quantity} \end{aligned}$$

where U and V are the CDFs of μ, ν respectively.

Proof. Assuming $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)})\} \subset \text{supp}(\gamma^*)$[18], $x^{(1)} < x^{(2)}$, where γ^* 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[19]

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

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 γ^*, as it indicates that γ^* is no cyclically monotone.

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

$$\gamma^*(\mathbb{R}, (-\infty, y]) = \gamma^*((-\infty, x], (-\infty, y]) = \gamma^*((-\infty, x], \mathbb{R})$$

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

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

□

Example 42. For one dimensional discrete case, to transport ν to μ,

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 $(\sum |U(s) - V(s)|^p ds)^{1/p} = (\sum 4^p \times 1 + 3^p \times 1 + 2^p \times 1 + 1^p \times 1)^{1/p}$

[18]The 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.

[19]For 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`

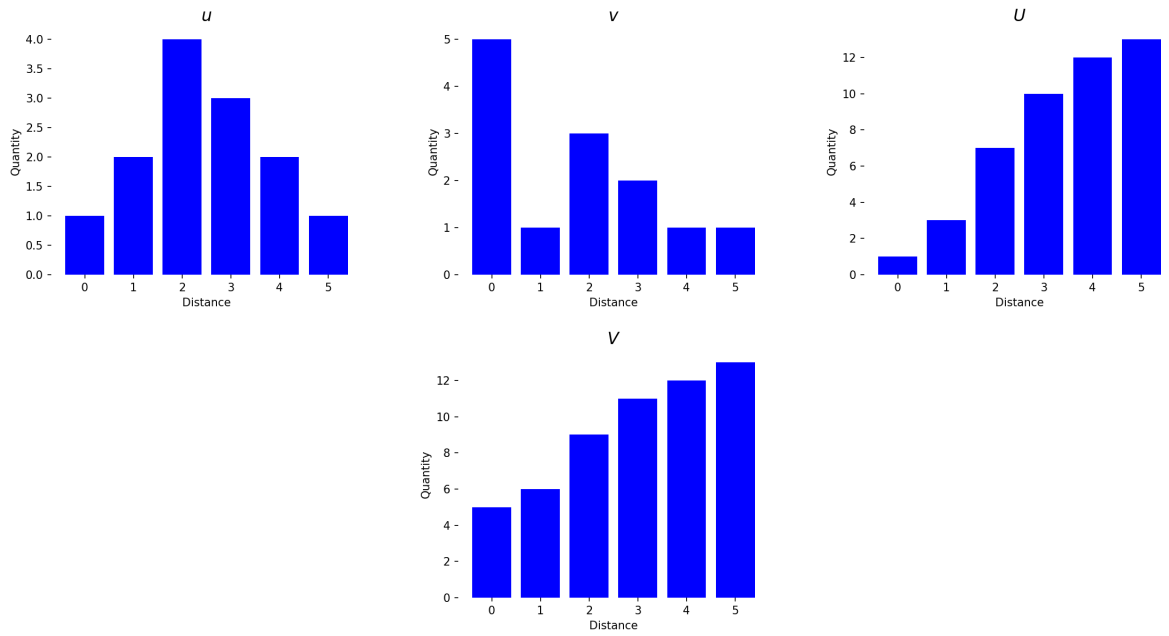


Figure 34: Two distribution μ and ν.

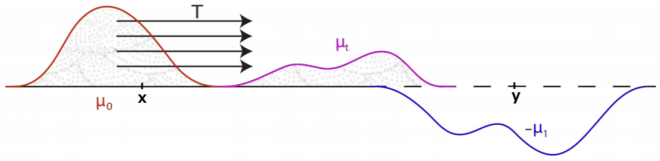


Figure 35: Optimal transport measures the minimal effort required for filling $-\mu_1$ with μ_0, i.e. transporting one distribution to another.

Remark 3.56. *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 3.57. *Advantages of Wasserstein distance:*[20]

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

[20]Figures in this remark credit to `https://www.stat.cmu.edu/~larry/=sml/Opt.pdf`

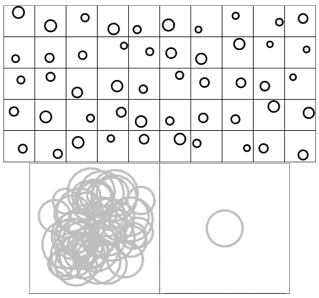


Figure 36: Top: Some random circles. Bottom left: Euclidean average of the circles. Bottom right: Wasserstein barycenter.

- *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.*

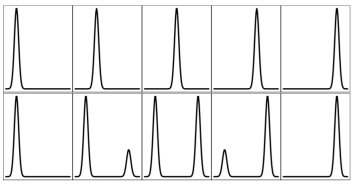


Figure 37: Top row: Geodesic path from P_0 to P_1. Bottom row: Euclidean path($P_t = tP_0 + (1-t)P_1$) from P_0 to P_1.

- *Wasserstein distance is insensitive to small wiggles.*

3.2.4 Normalization

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

Definition 3.75. **Batch normalization** [Ioffe and Szegedy, 2015] is defined as

$$\mu_c = \frac{1}{HWN}\sum_n^N\sum_w^W\sum_h^H x_{cnhw}$$
$$\sigma_c^2 = \frac{1}{HWN}\sum_n^N\sum_w^W\sum_h^H (x_{cnhw} - \mu_c)^2$$
$$\hat{x}_c = \frac{x_c - \mu_c}{\sqrt{\sigma_c^2 + \epsilon}}$$

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 3.58. *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 3.59. ***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, [Teye et al., 2018] and [Luo et al., 2018] 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 3.60. ***BN in training and testing differentiates slightly.*** *Typically, during the training, we use the statistics (mean and variance) from the specific mini-batch. During inference time, the statistics of the population is instead used in the batch normalization. Recall that dropout also exhibits this characteristic. Hence always remember to enable evaluation mode of the model during the inference time via* `model.eval()`.

Remark 3.61. *BN is typically implemented after the convolutional or fully-connected layer and before the activation function.*

Definition 3.76. **Layer normalization** is defined as

$$\begin{aligned}\mu_n &= \frac{1}{HWC}\sum_c^C\sum_w^W\sum_h^H x_{cnhw}\\ \sigma_n^2 &= \frac{1}{HWC}\sum_c^C\sum_w^W\sum_h^H (x_{cnhw}-\mu_n)^2\\ \hat{x}_n &= \frac{x_n-\mu_n}{\sqrt{\sigma_n^2+\epsilon}}\end{aligned}$$

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 3.77. **Instance normalization** is defined as

$$\begin{aligned}\mu_{cn} &= \frac{1}{HW}\sum_w^W\sum_h^H x_{cnhw}\\ \sigma_{cn}^2 &= \frac{1}{HW}\sum_w^W\sum_h^H (x_{cnhw}-\mu_{cn})^2\\ \hat{x}_{cn} &= \frac{x_{cn}-\mu_{cn}}{\sqrt{\sigma_{cn}^2+\epsilon}}\end{aligned}$$

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 3.78. **Group normalization** [Wu and He, 2018] is a middle ground between layer and instance normalization, which is defined as

$$\begin{aligned}\mu_{gn} &= \frac{1}{HW|G_i|}\sum_{c\in G_i}^{|G_i|}\sum_w^W\sum_h^H x_{cnhw}\\ \sigma_{gn}^2 &= \frac{1}{HW|G_i|}\sum_{c\in G_i}^{|G_i|}\sum_w^W\sum_h^H (x_{cnhw}-\mu_n)^2\\ \hat{x}_{gn} &= \frac{x_n-\mu_n}{\sqrt{\sigma_n^2+\epsilon}}\\ G &= \{G_1,\cdots\}\\ G_i &= \{c_i,\cdots,c_{i'}\}\end{aligned}$$

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.

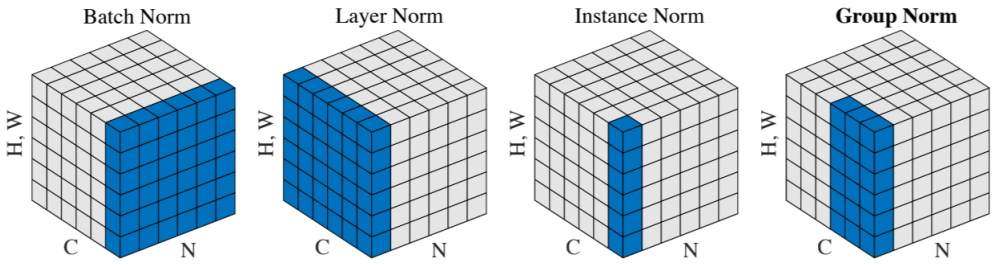


Figure 38: Normalization methods. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

3.2.5 Activation Function

Types	Functions	Range	Order of continuity
Sigmoid(logistic)	$\sigma(x) = \frac{1}{1+e^{-x}}$	$(0, 1)$	C^∞
Softplus	$\zeta(x) = \log(1 + e^x)$	$(0, \infty)$	C^∞
Tanh(hyperbolic tangent)	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$(-1, 1)$	C^∞
ReLU(rectified linear unit)	$\mathrm{ReLU}(x) = \max\{0, x\}$	$[0, +\infty)$	C^0
Leaky ReLU	$\mathrm{LeakyReLU}(x) = \max\{0.01x, x\}$	$(-\infty, +\infty)$	C^0
ELU(exponential linear unit)	$\begin{cases} \alpha\left(e^x - 1\right) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$(-\alpha, +\infty)$	$\begin{cases} C^1 & \text{if } \alpha = 1 \\ C^0 & \text{otherwise} \end{cases}$
GELU(gaussian error linear unit)	$\frac{1}{2}x\left(1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$	$(-0.17, +\infty)$	C^∞
SiLU(Sigmoid linear unit)	$x \cdot \sigma(x)$	$(-0.28, +\infty)$	C^∞
Smooth ReLU(Softplus)	$\ln(1 + e^x)$	$(0, +\infty)$	C^∞

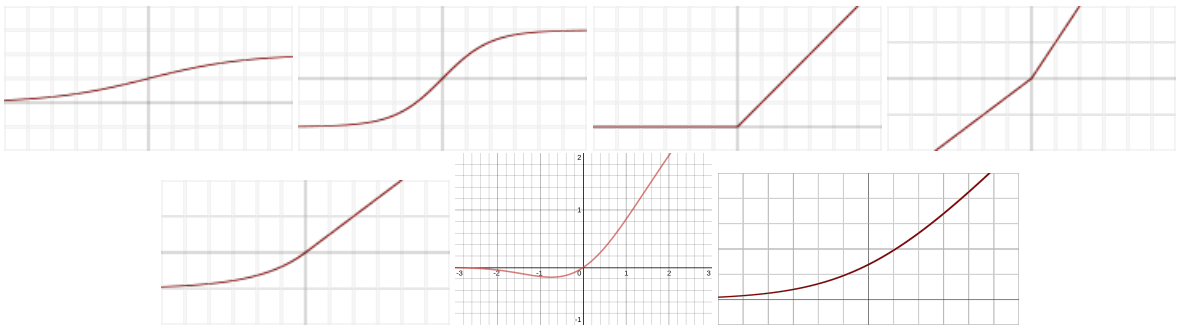


Figure 39: Plots of sigmoid, tahn, ReLU, Leaky ReLU, ELU, GELU, softplus activation functions, correspondingly.

The following table lists activation functions that are not functions of a single fold x from the previous layer or layers:

Types	Functions	Range	Order of continuity
Softmax	$\frac{e^{x^{(i)}}}{\sum_{j=1}^{J} e^{x^{(j)}}}$	$(0, 1)$	C^∞
Maxout	$\max_i x^{(i)}$	$(-\infty, \infty)$	C^0

Remark 3.62. *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 3.63. *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.*

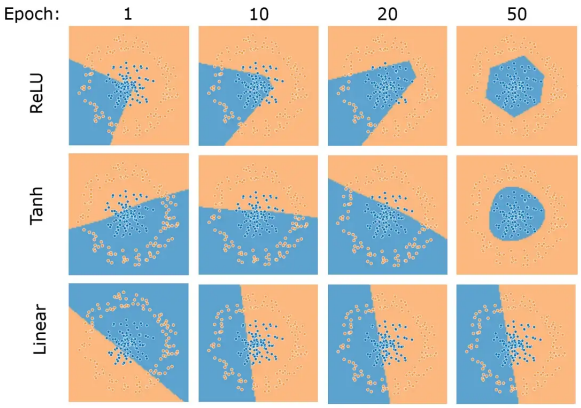


Figure 40: Whereas tanh, a smooth, curved function, draws a clean envelope around the circle (and linear fails completely), ReLU draws a hexagon, with several pointed corners.

3.2.6 Optimizer for gradient descent

Remark 3.64. *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 3.79. **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.

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla_\theta \mathcal{L}(\theta),$$
$$\theta = \theta - \mathbf{v}_t,$$

where γ is called the momentum term and is typically set to 0.9.

Remark 3.65. *If the current gradient is parallel to the previous speed $\mathbf{v}_{t-1}$, the previous speed can speed up this search. If the current gradient is opposite to the previous speed $\mathbf{v}_{t-1}$, the value of $\mathbf{v}_t$ will have a deceleration effect on this search.*

Definition 3.80. **NAG (Nesterov Accelerated Gradient)** is a method similar to the momentum, but calculates the gradient w.r.t. the approximate future position of parameter θ, rather than the current parameter.

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \nabla_\theta \mathcal{L}(\theta - \gamma \mathbf{v}_{t-1})$$
$$\theta = \theta - \mathbf{v}_t$$

where $\theta - \gamma v_{t-1}$ is the approximation of the next position of the parameters.

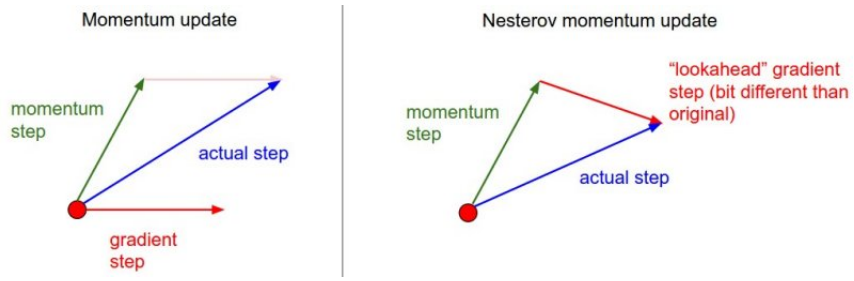


Figure 41: Nesterov momentum. Instead of calculating the gradient at the red circle, NAG looks ahead to the tip of the green arrow, which is slightly different from the original momentum method. This anticipatory update prevents us from going too fast and results in increased responsiveness.

Definition 3.81. **AdaGrad (Adaptive subgradient)** is a method that adjusts the learning rate for every parameter θ (not a vector by a scalar) at every time step t dynamically based on the historical gradient in previous iterations:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\sum_i^t \|\mathbf{g}^{(i)}\|_2^2 + \epsilon}} \cdot \mathbf{g}^{(t)}$$

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

Remark 3.66. *Without the square root operation, the algorithm performs much worse.*

Remark 3.67. *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 3.82. **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):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E(\|\mathbf{g}\|_2^2)_t + \epsilon}} \cdot \mathbf{g}^{(t)}$$
$$E^{(t)}(\|\mathbf{g}\|_2^2) = \gamma E_{t-1}(\|\mathbf{g}^{(t)}\|_2^2) + (1-\gamma)\|\mathbf{g}^{(t)}\|_2^2$$

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

Definition 3.83. **Adam (Adaptive moment)** [Kingma and Ba, 2017] is a method that combines adaptive learning rate and momentum methods:

$$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1-\beta_1)\mathbf{g}^{(t)}, \hat{\mathbf{m}}^{(t)} = \frac{\mathbf{m}^{(t)}}{1-\beta_1^t}$$

$$\mathbf{v}^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1-\beta_2)\|\mathbf{g}^{(t)}\|_2^2, \hat{\mathbf{v}}^{(t)} = \frac{\mathbf{v}^{(t)}}{1-\beta_2^t}$$

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

where $\mathbf{m}^{(t)}$ and $\mathbf{v}^{(t)}$ 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 β_1, 0.999 for β_2 and 10^{-8} for ϵ.

3.3 Representation Learning with Deep Learning

3.3.1 Autoencoder

Definition 3.84. **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.

$$\phi : \mathcal{X} \to \mathcal{F}$$

$$\psi : \mathcal{F} \to \mathcal{X}$$

$$\phi, \psi = \arg\min_{\phi,\psi} \|\mathbf{x} - (\psi \circ \phi)(\mathbf{x})\|^2 \qquad \triangleright \text{ loss function}$$

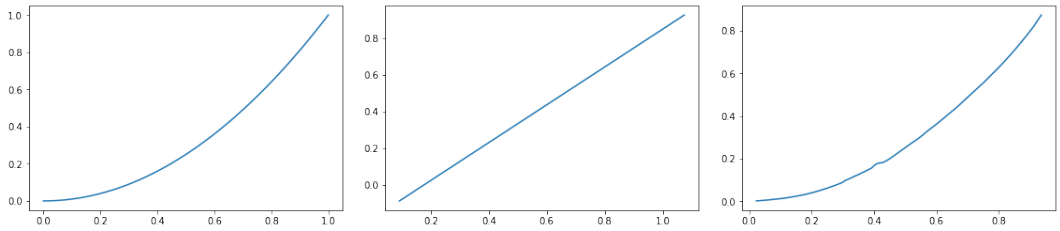


Figure 42: From left to right: 2D original distribution, distribution recovered by PCA, and distribution recovered by autoencoder. Since the original feature space is 2D, namely the covariance matrix of the distribution is 2×2, which means the distribution would be 1D after dimension reduction, and this is the reason why the distribution recovered by PCA is totally linear.

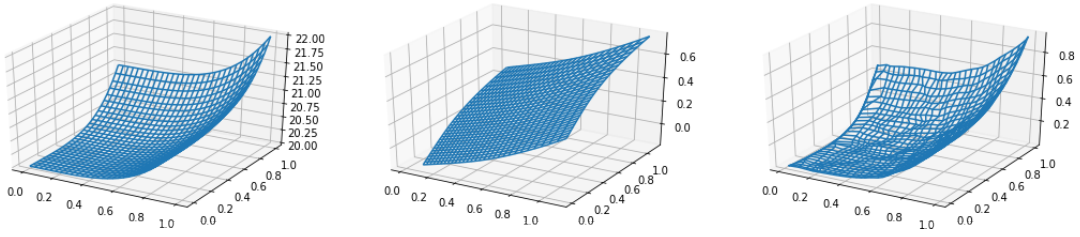


Figure 43: From left to right: 3D original distribution, distribution recovered by PCA, and distribution recovered by autoencoder. Since the original feature space is 3D, which means the distribution would be 2D after dimension reduction, and this is the reason why the distribution recovered by PCA looks like a plane.

Remark 3.68. *Relationship between **PCA** and **autoencoder**[21]:*

- *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 43. `PyTorch` implementation of autoencoder:

```
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 44. Different types of autoencoders:

[21] `https://github.com/muaz-urwa/PCA-vs-AutoEncoders/blob/master/PCAvsAE.ipynb`

- **Sparse autoencoder**[22] constraints most of the hidden units inactive (close to 0) by adding a regularization term to the loss function

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) + \lambda \sum_i \|a_i^{(h)}\| \qquad \triangleright\ L^1 \text{ regularization}$$

$$\mathcal{L}(\mathbf{x}, \hat{\mathbf{x}}) + \sum_j D_{\mathrm{KL}}(\rho, \rho_j) \qquad \triangleright\ \text{KL divergence regularization}$$

  where $a_i^{(h)}$ is the activation output of h-th layer's hidden unit i, ρ is the sparsity parameter which is typically preset as a small value close to 0, and ρ_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.

3.3.2 Contrastive Learning

Definition 3.85. **SimCLR** [Goodfellow et al., 2014] is a simple framework for contrastive learning of visual representations. The loss function for a positive pair of examples (i, j) is defined as

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

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

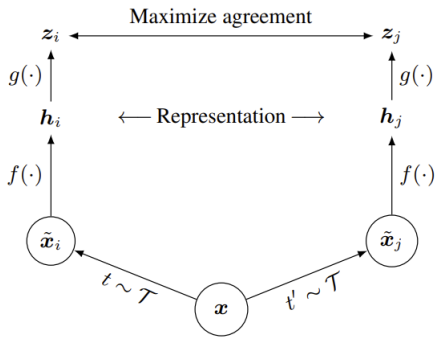


Figure 44: Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim \mathcal{T}$ and $t' \sim \mathcal{T}$) and applied to each data example to obtain two correlated views. A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained to maximize 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 3.86. **CLIP (Contrastive Language–Image Pre-training)** [Radford et al., 2021] is a pre-training framework to generate a representation for both image and text.

[22] `https://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf`

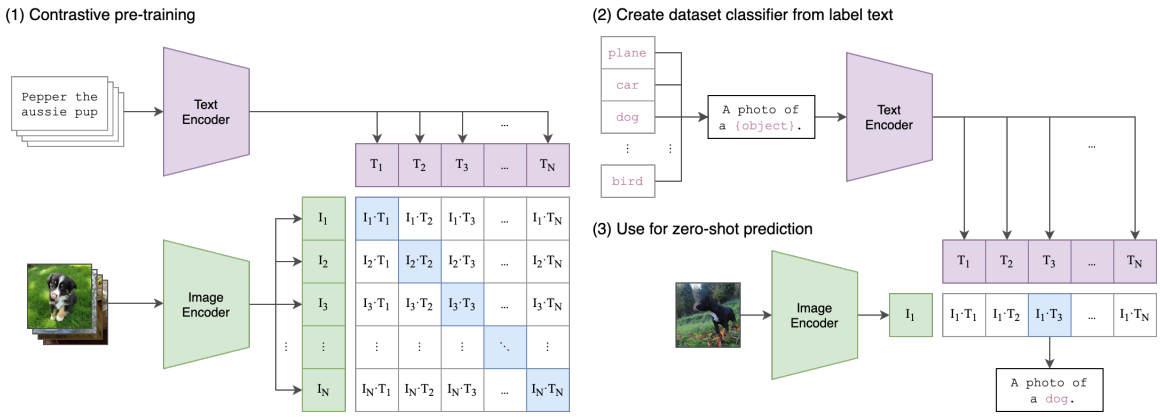


Figure 45: Summary of our approach. While standard image models jointly train an image feature extractor and a linear classifier to predict some label, CLIP jointly trains an image encoder and a text encoder to predict the correct pairings of a batch of (image, text) training examples. At test time the learned text encoder synthesizes a zero-shot linear classifier by embedding the names or descriptions of the target dataset's classes.

Example 45. 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)
```

3.4 Generative Models with Deep Learning

3.4.1 Techniques

Definition 3.87. **Fréchet Inception Distance (FID)** is a metric used to evaluate the quality and diversity of generated images by comparing them to a set of real images. For two multidimensional Gaussian distributions $\mathcal{N}(\mu, \Sigma)$ and $\mathcal{N}(\mu', \Sigma')$, it is explicitly solvable as

$$d_F(\mathcal{N}(\mu, \mathbf{\Sigma}), \mathcal{N}(\mu', \mathbf{\Sigma}'))^2 = \|\mu - \mu'\|_2^2 + \operatorname{Tr}\left(\mathbf{\Sigma} + \mathbf{\Sigma}' - 2\left(\mathbf{\Sigma}^{\frac{1}{2}} \cdot \mathbf{\Sigma}' \cdot \mathbf{\Sigma}^{\frac{1}{2}}\right)^{\frac{1}{2}}\right)$$

Definition 3.88. **Classifier-Free Guidance Scale (CFG Scale)** is a number (typically somewhere between 7.0 to 13.0) that refers to the ability to increase or decrease the amount of influence the text description has on the image generation.

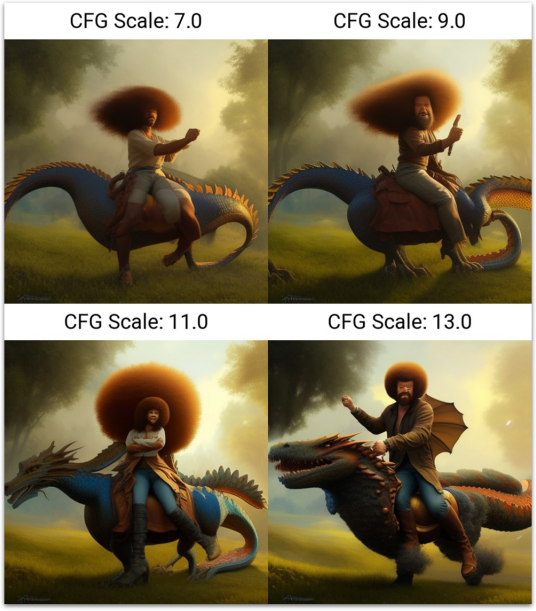


Figure 46: CFG can sometimes improve the quality of the generated result. In the example of "Bob Ross riding a dragon", it's not till a scale of 13 that we get something reasonable.

3.4.2 Datasets

Definition 3.89. **The Large-scale Artificial Intelligence Open Network (LAION)** [Schuhmann et al., 2022] is a German non-profit releasing a number of large datasets of images and captions scraped from the web which have been used to train a number of high-profile text-to-image models, including Stable Diffusion and Imagen.

Definition 3.90. **The Large-scale Scene Understanding (LSUN)** [Yu et al., 2015] challenge aims to provide a different benchmark for large-scale scene classification and understanding. The LSUN classification dataset contains 10 scene categories, such as dining room, bedroom, chicken, outdoor church, and so on. For training data, each category contains a huge number of images, ranging from around 120,000 to 3,000,000. The validation data includes 300 images, and the test data has 1000 images for each category.

Definition 3.91. **ShapeNet** [Chang et al., 2015] is an 3D shapes dataset. Each meaningful concept in WordNet, possibly described by multiple words or word phrases, is called a "synonym set" or "synset". There are more than 100,000 synsets in WordNet [Miller, 1995], the majority of them being nouns (80,000+). ShapeNet is made of several different subsets:

1. ShapeNetCore is a subset of the full ShapeNet dataset with single clean 3D models and manually verified category and alignment annotations. It covers 55 common object categories with about 51,300 unique 3D models.

2. ShapeNetSem is a smaller, more densely annotated subset consisting of 12,000 models spread over a broader set of 270 categories.

3.4.3 Variational Autoencoder

Definition 3.92. **Variational autoencoder** [Kingma and Welling, 2014] 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 46. The workflow of variational autoencoder:

1. We feed a real data sample $\mathbf{x}^{(i)}$ into the encoder, the output of which is the parameters $\log \sigma_i^2, \mu_i$ of $q_\phi(\mathbf{z}|\mathbf{x})$;
2. We sample a $\mathbf{z}^{(i)}$[23] from the Gaussian distribution $q_\phi(\mathbf{z}|\mathbf{x}) \sim \mathcal{N}(\mu, \sigma)$;
3. We feed the sampled $\mathbf{z}^{(i)}$ into the decoder, the output of which is the parameters σ_i', μ_i' of $p(\mathbf{x}|\mathbf{z})$;
4. We sample a $\mathbf{x}'$ from the Gaussian distribution $p(\mathbf{x}'|\mathbf{z}) \sim \mathcal{N}(\mu', \sigma')$.

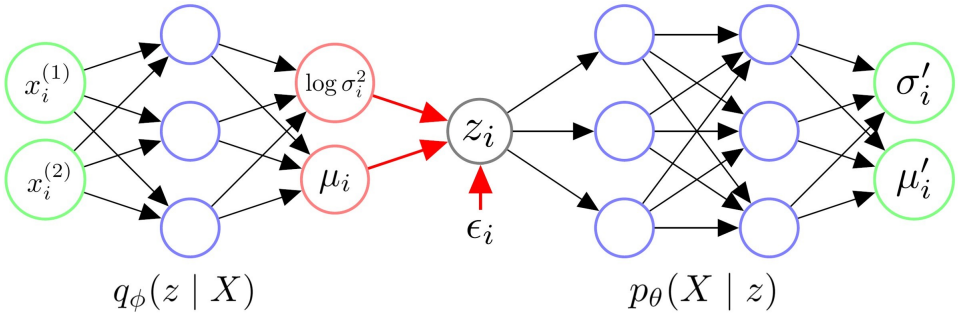


Figure 47: Variational autoencoder

Example 47. Pseudocode[24] of VAE:[25]

```
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):
    # encoder part
    h = F.relu(self.fc1(x))
    h = F.relu(self.fc2(h))
```

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

[24] `https://github.com/lyeoni/pytorch-mnist-GAN/blob/master/pytorch-mnist-VAE.ipynb`

[25] The reason of the output of encoder is $\log(\sigma^2)$ rather than σ is that taking the logarithm helps prevent numerical underflow or overflow issues that can occur when dealing with very small or very large values.

```
        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 3.69. *The ultimate goal of VAE is to find the true probabilistic model $p(\mathbf{x})$, the distribution of real-world samples $\mathbf{x}$. 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! $\sum_i \log p(\mathbf{x}_i)$ 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:

$$\begin{aligned}
\log p(\mathbf{x}) &= 1 \cdot \log p(\mathbf{x}) \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \cdot \log p(\mathbf{x}) \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \log p(\mathbf{x}) d\mathbf{z} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p(\mathbf{x},\mathbf{z})}{p(\mathbf{z}|\mathbf{x})} d\mathbf{z} && \triangleright \frac{p(\mathbf{x},\mathbf{z})}{p(\mathbf{z}|\mathbf{x})} = \frac{p(\mathbf{x},\mathbf{z})}{\frac{p(\mathbf{x},\mathbf{z})}{p(\mathbf{x})}} = p(\mathbf{x}) \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \log \left(\frac{p(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \cdot \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})} \right) d\mathbf{z} \\
&= \underbrace{\int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z}}_{ELBO} + \underbrace{\int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z}|\mathbf{x})} d\mathbf{z}}_{D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z}|\mathbf{x}))}
\end{aligned}$$

As $\log p(\mathbf{x})$, a real-world probabilistic distribution of $\mathbf{x}$, is constant regardless of ϕ, to minimize $D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z}|\mathbf{x}))$, namely making the two distribution $q_\phi(\mathbf{z}|\mathbf{x})$ and $p(\mathbf{z}|\mathbf{x})$ as close as possible, is equivalent to maximize ELBO. The more we optimize the ELBO, the closer our approximate posterior gets to the true posterior.

$$\log p(\mathbf{x}) = \underbrace{\int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z}}_{ELBO} + D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z}|\mathbf{x}))$$

$$\log p(\mathbf{x}) - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z}|\mathbf{x})) \downarrow = \underbrace{\int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z}}_{ELBO} \uparrow$$

where

$$
\begin{aligned}
\text{ELBO} &= \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p(\mathbf{x},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z} && \triangleright\ q_\phi(\mathbf{z}|\mathbf{x}) \textit{ can be viewed as encoder} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p_\theta(\mathbf{x}|\mathbf{z}) \cdot p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z} && \triangleright\ p_\theta(\mathbf{x}|\mathbf{z}) \textit{ can be viewed as decoder} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \log p_\theta(\mathbf{x}|\mathbf{z}) d\mathbf{z} + \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z} \\
&= E_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})) && \triangleright\ D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})) = \int q_\phi(\mathbf{z}|\mathbf{x}) \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} d\mathbf{z} \\
&= E_{q_\phi}[\log p_\theta(\mathbf{x}|\mathbf{z})] + \frac{1}{2} \sum_i^d (1 - \mu_i^2 - \sigma_i^2 + \log(\sigma_i^2)) && \triangleright\ \textit{See deviation below} \\
&\Rightarrow -\sum_i^d (\mathbf{x}_i - \mu_i')^2 + \frac{1}{2} \sum_i^d (1 - \mu_i^2 - \sigma_i^2 + \log(\sigma_i^2)) && \triangleright\ \textit{See deviation below}
\end{aligned}
$$

Since we assume $p(\mathbf{x}|\mathbf{z}) \sim \mathcal{N}(\mu', \sigma'^2)$, the maximization of $= E_{q_\phi}[\log p(\mathbf{x}|\mathbf{z})]$ (MLE) is equivalent to minimizing mean squared error (MSE), see Remark 3.46. Therefore, the loss function of VAE can be written as

$$
L = \sum_i^d (\mathbf{x}_i - \mu_i')^2 - \frac{1}{2} \sum_i^d (1 - \mu_i^2 - \sigma_i^2 + \log(\sigma_i^2)), \tag{1}
$$

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

$$
\begin{aligned}
-D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})) &= -\int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \log \frac{q_\phi(\mathbf{z}|\mathbf{x})}{p(\mathbf{z})} d\mathbf{z} \\
&= -\int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \log q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} + \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \log p(\mathbf{z}) d\mathbf{z} \\
&= \frac{1}{2} + \frac{1}{2} \log 2\pi\sigma^2 - \frac{1}{2}(\mu^2 + \sigma^2) - \frac{1}{2} \log 2\pi && \triangleright\ \textit{See deviation below} \\
&= \frac{1}{2}(1 - \mu^2 - \sigma^2 + \log(\sigma^2))
\end{aligned}
$$

For one-dimensional distribution, recalling $q_\phi(\mathbf{z}|\mathbf{x}) \sim \mathcal{N}(\mu, \sigma^2)$ and $p(\mathbf{z}) \sim \mathcal{N}(0, 1)$, we have

$$
\begin{aligned}
\int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \log q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} &= \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \log \left(\frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{(\mathbf{z} - \mu)^2}{2\sigma^2} \right) \right) d\mathbf{z} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \left(-\frac{(\mathbf{z} - \mu)^2}{2\sigma^2} \right) d\mathbf{z} - \int q_\phi(\mathbf{z}|\mathbf{x}) \log \sqrt{2\pi\sigma^2} d\mathbf{z} \\
&= -\frac{1}{2\sigma^2} \underbrace{\int q_\phi(\mathbf{z}|\mathbf{x}) \cdot (\mathbf{z} - \mu)^2 d\mathbf{z}}_{\textit{Definition of variance}} - \log \sqrt{2\pi\sigma^2} \int q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \\
&= -\frac{1}{2\sigma^2} \sigma^2 - \log \sqrt{2\pi\sigma^2} \\
&= -\frac{1}{2} - \frac{1}{2} \log 2\pi\sigma^2
\end{aligned}
$$

$$
\begin{aligned}
\int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \log p(\mathbf{z}) d\mathbf{z} &= \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \log \left(\frac{1}{\sqrt{2\pi \cdot 1^2}} \cdot \exp\left(-\frac{(\mathbf{z}-0)^2}{2 \cdot 1^2} \right) \right) d\mathbf{z} \\
&= \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \left(-\frac{\mathbf{z}^2}{2} \right) d\mathbf{z} - \int q_\phi(\mathbf{z}|\mathbf{x}) \log \sqrt{2\pi} d\mathbf{z} \\
&= -\frac{1}{2} \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \mathbf{z}^2 d\mathbf{z} - \log \sqrt{2\pi} \int q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \\
&= -\frac{1}{2} \left(\underbrace{\int q_\phi(\mathbf{z}|\mathbf{x}) \cdot (\mathbf{z}-\mu)^2 d\mathbf{z}}_{\textit{Definition of variance}} + \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot 2\mu\mathbf{z} d\mathbf{z} - \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \mu^2 d\mathbf{z} \right) - \log \sqrt{2\pi} \int q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \\
&= -\frac{1}{2} \left(\sigma^2 + 2\mu \int q_\phi(\mathbf{z}|\mathbf{x}) \cdot \mathbf{z} d\mathbf{z} - \mu^2 \int q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \right) - \log \sqrt{2\pi} \int q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \\
&= -\frac{1}{2} \left(\sigma^2 + 2\mu \cdot \mu - \mu^2 \cdot 1 \right) - \log \sqrt{2\pi} \int q_\phi(\mathbf{z}|\mathbf{x}) d\mathbf{z} \\
&= -\frac{1}{2} (\mu^2 + \sigma^2) - \frac{1}{2} \log 2\pi
\end{aligned}
$$

Remark 3.70. *Below are the assumptions that we made in the formulation of VAE's loss function:*

Prior:	$p(\mathbf{z}) \sim N(0,1)$
Posterior:	$q_\phi(\mathbf{x}\|\mathbf{z}) \sim \mathcal{N}(\mu, \sigma^2)$
Likelihood:	$p_\theta(\mathbf{z}\|\mathbf{x}) \sim \mathcal{N}(\mu', \sigma'^2)$

Normal distribution is not the only distribution used for latent variables in VAEs. There are also works using von Mises-Fisher distribution, and there are VAEs using Gaussian mixtures.

But normal distribution has many nice properties, such as analytical evaluation of the KL divergence in the variational loss, and also we can use the reparametrization trick for efficient gradient computation.

Remark 3.71. *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.*

3.4.4 Variational Diffusion Models

Definition 3.93. **Denoising Diffusion Probabilistic Models (DDPM)** [Ho et al., 2020] are generative models used in the field of computer vision and image processing. DDPM is a variant of the Diffusion Probabilistic Model, which is based on the idea of iteratively applying a diffusion process to the noise added to an image. In DDPM, the model is trained to estimate the likelihood of a clean image, given its noisy version.

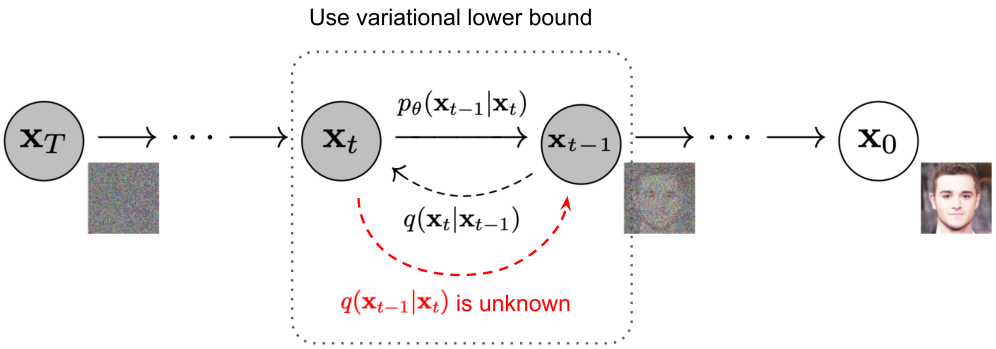


Figure 48: The Markov chain of forward (reverse) diffusion process of generating a sample by slowly adding (removing) noise.

Remark 3.72. *The ultimate goal of VDM is to find the true probabilistic model $p(\mathbf{x})$, the distribution of real-world samples $\mathbf{x}$. 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! $\sum_i \log p(\mathbf{x}_i)$ 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 VDM [Luo, 2022]:

$$
\begin{aligned}
\log p(\mathbf{x}) &= \log \int p\left(\mathbf{x}_{0:T}\right) d\mathbf{x}_{1:T} \triangleright p\left(\mathbf{x}_{0:T}\right) \iff p\left(\mathbf{x}_0, \cdots, \mathbf{x}_T\right) \\
&= \log \int \frac{p\left(\mathbf{x}_{0:T}\right) q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)} d\mathbf{x}_{1:T} \\
&= \log \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\frac{p\left(\mathbf{x}_{0:T}\right)}{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\right] \\
&\geq \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \frac{p\left(\mathbf{x}_{0:T}\right)}{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\right] \triangleright \textit{Jensen's Inequality } f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2). \\
&= \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \frac{p\left(\mathbf{x}_T\right) \prod_{t=1}^T p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{\prod_{t=1}^T q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)}\right] \\
&= \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right) \prod_{t=2}^T p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{q\left(\mathbf{x}_T \mid \mathbf{x}_{T-1}\right) \prod_{t=1}^{T-1} q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)}\right] \\
&= \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right) \prod_{t=1}^{T-1} p_\theta\left(\mathbf{x}_t \mid \mathbf{x}_{t+1}\right)}{q\left(\mathbf{x}_T \mid \mathbf{x}_{T-1}\right) \prod_{t=1}^{T-1} q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)}\right] \\
&= \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)}{q\left(\mathbf{x}_T \mid \mathbf{x}_{T-1}\right)}\right] + \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \prod_{t=1}^{T-1} \frac{p_\theta\left(\mathbf{x}_t \mid \mathbf{x}_{t+1}\right)}{q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)}\right] \\
&= \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right] + \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \frac{p\left(\mathbf{x}_T\right)}{q\left(\mathbf{x}_T \mid \mathbf{x}_{T-1}\right)}\right] + \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\sum_{t=1}^{T-1} \log \frac{p_\theta\left(\mathbf{x}_t \mid \mathbf{x}_{t+1}\right)}{q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)}\right] \\
&= \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right] + \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \frac{p\left(\mathbf{x}_T\right)}{q\left(\mathbf{x}_T \mid \mathbf{x}_{T-1}\right)}\right] + \sum_{t=1}^{T-1} \mathbb{E}_{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\left[\log \frac{p_\theta\left(\mathbf{x}_t \mid \mathbf{x}_{t+1}\right)}{q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)}\right] \\
&= \mathbb{E}_{q\left(\mathbf{x}_1 \mid \mathbf{x}_0\right)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right] + \mathbb{E}_{q\left(\mathbf{x}_{T-1}, \mathbf{x}_T \mid \mathbf{x}_0\right)}\left[\log \frac{p\left(\mathbf{x}_T\right)}{q\left(\mathbf{x}_T \mid \mathbf{x}_{T-1}\right)}\right] + \sum_{t=1}^{T-1} \mathbb{E}_{q\left(\mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1} \mid \mathbf{x}_0\right)}\left[\log \frac{p_\theta\left(\mathbf{x}_t \mid \mathbf{x}_{t+1}\right)}{q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)}\right] \\
&\quad \triangleright \textit{Only } \mathbf{x}_t, \mathbf{x}_{t+1}, \mathbf{x}_{t-1}, \mathbf{x}_0 \textit{ involved in the last two terms.} \\
&= \underbrace{\mathbb{E}_{q\left(\mathbf{x}_1 \mid \mathbf{x}_0\right)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right]}_{\textit{reconstruction term}} \\
&\quad - \underbrace{\mathbb{E}_{q\left(\mathbf{x}_{T-1} \mid \mathbf{x}_0\right)}\left[D_{\mathrm{KL}}\left(q\left(\mathbf{x}_T \mid \mathbf{x}_{T-1}\right) \| p\left(\mathbf{x}_T\right)\right)\right]}_{\textit{prior matching term}} \\
&\quad - \sum_{t=1}^{T-1} \underbrace{\mathbb{E}_{q\left(\mathbf{x}_{t-1}, \mathbf{x}_{t+1} \mid \mathbf{x}_0\right)}\left[D_{\mathrm{KL}}\left(q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right) \| p_\theta\left(\mathbf{x}_t \mid \mathbf{x}_{t+1}\right)\right)\right]}_{\textit{consistency term}}
\end{aligned}
$$

1. *The **reconstruction term** predicts the log probability of the original data sample given the first-step latent. This term also appears in a vanilla VAE, and can be trained similarly.*

2. *The **prior matching term** is minimized when the final latent distribution matches the Gaussian prior. This term requires no optimization, as it has no trainable parameters; furthermore, as we have assumed a large enough T such that the final distribution is Gaussian, this term effectively becomes zero.*

3. *The **consistency term** endeavors to make the distribution at $\mathbf{x}_t$ consistent, from both forward and backward processes. That is, a denoising step from a noisier image should match the corresponding noising step from a cleaner image, for every intermediate timestep; this is reflected mathematically by the KL Divergence. This term is minimized when we train $p_\theta(\mathbf{x}_t|\mathbf{x}_{t+1})$ to match the Gaussian distribution $q(\mathbf{x}_t|\mathbf{x}_{t-1})$.*

In this derivation, all components of the Evidence Lower Bound (ELBO) are calculated as expectations, making it possible to approximate them using Monte Carlo estimates. However, optimizing the ELBO using the derived terms may not be the most efficient approach. This inefficiency stems from the fact that the consistency term involves expectations over two random variables, namely, $\{\boldsymbol{x}_{t-1}, \boldsymbol{x}_{t+1}\}$ for each time step. Consequently, the Monte Carlo estimate for this term may exhibit higher variance compared to terms that rely on only one random variable per time step. Given that the ELBO involves summing up $T-1$ consistency terms, when dealing with large values of T, the final estimated value of the ELBO could have considerable variance.

$$
\begin{aligned}
\log p(\mathbf{x}) &\geq \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_{0:T}\right)}{q\left(\mathbf{x}_{1:T} \mid \mathbf{x}_0\right)}\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right) \prod_{t=1}^{T} p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{\prod_{t=1}^{T} q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)}\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right) \prod_{t=2}^{T} p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{q\left(\mathbf{x}_1 \mid \mathbf{x}_0\right) \prod_{t=2}^{T} q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{x}_0\right)}\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)}{q\left(\mathbf{x}_1 \mid \mathbf{x}_0\right)} + \log \prod_{t=2}^{T} \frac{p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{x}_0\right)}\right] \\
&\triangleright \textit{Bayes' rule } \Pr(A|B,C) = \frac{\Pr(A,B|C)}{\Pr(B|C)} = \frac{\Pr(B|A,C)\Pr(A|C)}{\Pr(B|C)} \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)}{q\left(\mathbf{x}_1 \mid \mathbf{x}_0\right)} + \log \prod_{t=2}^{T} \frac{p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{\frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}}\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)}{q\left(\mathbf{x}_1 \mid \mathbf{x}_0\right)} + \log \prod_{t=2}^{T} \frac{p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)} + \log \prod_{t=2}^{T} \frac{q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)}{q(\mathbf{x}_t \mid \mathbf{x}_0)}\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)}{q\left(\mathbf{x}_1 \mid \mathbf{x}_0\right)} + \log \prod_{t=2}^{T} \frac{p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)} + \log \frac{q(\mathbf{x}_1 \mid \mathbf{x}_0)}{q(\mathbf{x}_T \mid \mathbf{x}_0)}\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right) p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)}{q(\mathbf{x}_T \mid \mathbf{x}_0)} + \log \prod_{t=2}^{T} \frac{p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)}\right] \\
&= \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right] + \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right)}{q\left(\mathbf{x}_T \mid \mathbf{x}_0\right)}\right] + \mathbb{E}_{q(\mathbf{x}_{1:T}|\mathbf{x}_0)}\left[\sum_{t=2}^{T} \log \frac{p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right)}\right] \\
&= \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right] + \mathbb{E}_{q(\mathbf{x}_T|\mathbf{x}_0)}\left[\log \frac{p\left(\mathbf{x}_T\right)}{q\left(\mathbf{x}_T \mid \mathbf{x}_0\right)}\right] + \sum_{t=2}^{T} \mathbb{E}_{q(\mathbf{x}_t,\mathbf{x}_{t-1}|\mathbf{x}_0)}\left[\log \frac{p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)}{q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right)}\right] \\
&\triangleright \textit{Only } \mathbf{x}_t, \mathbf{x}_{t-1}, \mathbf{x}_0 \textit{ involved in the last two terms.} \\
&= \underbrace{\mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right]}_{\textit{reconstruction term}} \\
&- \underbrace{D_{\mathrm{KL}}\left(q\left(\mathbf{x}_T \mid \mathbf{x}_0\right) \| p\left(\mathbf{x}_T\right)\right)}_{\textit{prior matching term}} \\
&- \sum_{t=2}^{T} \underbrace{\mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[D_{\mathrm{KL}}\left(q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right) \| p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)\right)\right]}_{\textit{consistency term}}
\end{aligned}
$$

One observes that in the process of both ELBO derivations, only the Markov assumption is used; as a result these formula will hold true for any arbitrary Markovian HVAE. Furthermore, when we set $T = 1$, we have

$$\begin{aligned}
&\underbrace{\mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right]}_{\textit{reconstruction term}} - \underbrace{D_{\mathrm{KL}}\left(q\left(\mathbf{x}_T \mid \mathbf{x}_0\right) \| p\left(\mathbf{x}_T\right)\right)}_{\textit{prior matching term}} - \sum_{t=2}^{T} \underbrace{\mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[D_{\mathrm{KL}}\left(q\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0\right) \| p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right)\right)\right]}_{\textit{consistency term}} \\
&= \mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}\left[\log p_\theta\left(\mathbf{x}_0 \mid \mathbf{x}_1\right)\right] - D_{\mathrm{KL}}\left(q\left(\mathbf{x}_1 \mid \mathbf{x}_0\right) \| p\left(\mathbf{x}_1\right)\right) \\
&= \underbrace{\mathbb{E}_{q(\mathbf{z}|\mathbf{x})}\left[\log p_\theta\left(\mathbf{x} \mid \mathbf{z}\right)\right]}_{\textit{reconstruction term}} - \underbrace{D_{\mathrm{KL}}\left(q\left(\mathbf{z} \mid \mathbf{x}\right) \| p\left(\mathbf{z}\right)\right)}_{\textit{prior matching term}}
\end{aligned}$$

which is exactly the ELBO equation of a vanilla VAE, when treating $\mathbf{x}_1$ as the latent variable $\mathbf{z}$ and $\mathbf{x}_0$ as the real world sample $\mathbf{x}$ in VAE.

Remark 3.73. *Note that our encoder distributions $q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right)$ are no longer parameterized by ϕ, but a fixed linear Gaussian model,*

$$\begin{aligned}
q\left(\mathbf{x}_t \mid \mathbf{x}_{t-1}\right) &= \mathcal{N}(\mathbf{x}_t; \mu = \sqrt{\alpha}\mathbf{x}_{t-1}, \Sigma = (1-\alpha_t)\mathbf{I}) \\
\textit{where } \mathbf{x}_t &= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\epsilon_{t-1}^*
\end{aligned}$$

Proof. Suppose $\{\epsilon_t^*, \epsilon_t\}_{t=0}^T \overset{\text{iid}}{\sim} \mathcal{N}(\epsilon; \mathbf{0}, \mathbf{I})$

$$\begin{aligned}
\mathbf{x}_t &= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\epsilon_{t-1}^* \\
&= \sqrt{\alpha_t}\left(\sqrt{\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_{t-1}}\epsilon_{t-2}^*\right) + \sqrt{1-\alpha_t}\epsilon_{t-1}^* \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{\alpha_t - \alpha_t\alpha_{t-1}}\epsilon_{t-2}^* + \sqrt{1-\alpha_t}\epsilon_{t-1}^* \\
&\triangleright \text{ Gaussian summation rule:} \\
&\epsilon_1 \sim \mathcal{N}(\mu_{\epsilon_1}, \sigma_{\epsilon_1}^2), \epsilon_2 \sim \mathcal{N}(\mu_{\epsilon_2}, \sigma_{\epsilon_2}^2), \epsilon_1 + \epsilon_2 \sim \mathcal{N}(\mu_{\epsilon_1} + \mu_{\epsilon_2}, \sigma_{\epsilon_1}^2 + \sigma_{\epsilon_2}^2) \\
&a\epsilon_{t-2} \sim \mathcal{N}(\mathbf{0}, a^2\mathbf{I}), b\epsilon_{t-1} \sim \mathcal{N}(\mathbf{0}, b^2\mathbf{I}), a\epsilon_{t-2} + b\epsilon_{t-1} \sim \mathcal{N}(\mathbf{0}+\mathbf{0}, (a^2+b^2)\mathbf{I}) \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{\sqrt{\alpha_t - \alpha_t\alpha_{t-1}}^2 + \sqrt{1-\alpha_t}^2}\epsilon_{t-2} \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{\alpha_t - \alpha_t\alpha_{t-1} + 1 - \alpha_t}\epsilon_{t-2} \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1 - \alpha_t\alpha_{t-1}}\epsilon_{t-2} \\
&= \dots \\
&= \sqrt{\prod_{i=1}^{t}\alpha_i}\mathbf{x}_0 + \sqrt{1 - \prod_{i=1}^{t}\alpha_i}\epsilon_0 \\
&= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_0 \\
&\sim \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I})
\end{aligned}$$

□

Hence, we have

$$\begin{aligned}
q\left(\mathbf{x}_t \mid \mathbf{x}_0\right) &= \mathcal{N}(\mathbf{x}_t; \mu = \sqrt{\alpha}\mathbf{x}_0, \Sigma = (1-\alpha_t)\mathbf{I}) \\
\textit{where } \mathbf{x}_t &= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon_0^* \qquad \triangleright \textit{ Reparameterization trick}
\end{aligned}$$

Unlike the $q_\phi\left(\mathbf{z} \mid \mathbf{x}\right)$ in VAE, they are completely modeled as Gaussians with defined mean and variance parameters at each timestep. Therefore, in a VDM, we are only interested in learning conditionals

$$\log p_\theta\left(\mathbf{x}_{t-1} \mid \mathbf{x}_t\right),$$

so that we can simulate new data.

Remark 3.74. *How do we calculate the consistency term in the loss function? The first unknown in the consistency term is*

$$
\begin{aligned}
q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) &= \frac{q(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{x}_0) q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)}{q(\mathbf{x}_t \mid \mathbf{x}_0)} \\
&\triangleright\ q(\mathbf{x}_t \mid \mathbf{x}_{t-1}, \mathbf{x}_0) = q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) \textit{ due to the Markov property.} \\
&= \frac{q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)}{q(\mathbf{x}_t \mid \mathbf{x}_0)} \\
&= \frac{\mathcal{N}(\mathbf{x}_t; \sqrt{\alpha_t}\mathbf{x}_{t-1}, (1-\alpha_t)\boldsymbol{I}) \mathcal{N}(\mathbf{x}_{t-1}; \sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0, (1-\bar{\alpha}_{t-1})\boldsymbol{I})}{\mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I})} \\
&\propto \mathcal{N}(\mathbf{x}_{t-1}; \underbrace{\frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t}}_{\mu_q(\mathbf{x}_t, \mathbf{x}_0)}, \underbrace{\frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\boldsymbol{I}}_{\boldsymbol{\Sigma}_q(t)}) \\
&\propto \mathcal{N}(\mathbf{x}_{t-1}; \mu_q(\mathbf{x}_t, \mathbf{x}_0), \sigma_q^2(t)\mathbf{I}) \\
\mu_q(\mathbf{x}_t, \mathbf{x}_0) &= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t} \\
\sigma_q^2(t) &= \frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}
\end{aligned}
$$

In order to match the approximate denoising transition step $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ to ground-truth denoising transition step $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ as closely as possible, we can also model it as a Gaussian. We utilize the fact that the KL Divergence between two Gaussian distributions is:

$$
\mathcal{D}_{KL}(\mathcal{N}(\mathbf{x}; \mu_x, \boldsymbol{\Sigma}_x) \mid\mid \mathcal{N}(\mathbf{y}; \mu_y, \boldsymbol{\Sigma}_y)) = \frac{1}{2}\left[\log \frac{\mid \boldsymbol{\Sigma}_y \mid}{\mid \boldsymbol{\Sigma}_x \mid} - d + \operatorname{Tr}(\boldsymbol{\Sigma}_y^{-1}\boldsymbol{\Sigma}_x) + \underbrace{(\mu_y - \mu_x)^T \boldsymbol{\Sigma}_y^{-1}(\mu_y - \mu_x)}_{\textit{Mahalanobis distance}}\right]
$$

Hence, we have

$$
\begin{aligned}
&\arg\min_\theta \mathcal{D}_{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \mid\mid p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)) \\
&= \arg\min_\theta \mathcal{D}_{KL}\left(\mathcal{N}\left(\mathbf{x}_{t-1}; \mu_q, \boldsymbol{\Sigma}_q(t)\right) \mid\mid \mathcal{N}\left(\mathbf{x}_{t-1}; \mu_\theta, \boldsymbol{\Sigma}_q(t)\right)\right) \\
&= \arg\min_\theta \frac{1}{2\sigma_q^2(t)}\left[\|\mu_\theta - \mu_q\|_2^2\right],
\end{aligned}
$$

where we have written μ_q as shorthand for $\mu_{q(\mathbf{x}_{t-1}\mid\mathbf{x}_t,\mathbf{x}_0)}$, and $\mu_{p_\theta(\mathbf{x}_{t-1}\mid\mathbf{x}_t)}$ as shorthand for $\mu_\theta(\mathbf{x}_t, t)$ for brevity. In other words, we want to optimize a $\mu_{p_\theta(\mathbf{x}_{t-1}\mid\mathbf{x}_t)}$ that matches $\mu_{q(\mathbf{x}_{t-1}\mid\mathbf{x}_t,\mathbf{x}_0)}$, takes the form:

$$
\mu_{q(\mathbf{x}_{t-1}\mid\mathbf{x}_t,\mathbf{x}_0)} = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t}
$$

As p_θ is the probability model that we want to construct, we can formulate μ_θ as a function of $\mathbf{x}_t, t$, which matches $\mu_{q(\mathbf{x}_{t-1}\mid\mathbf{x}_t,\mathbf{x}_0)}$ closely by setting it to the following form:

$$
\mu_{p_\theta(\mathbf{x}_{t-1}\mid\mathbf{x}_t)} = \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\hat{\mathbf{x}}_\theta(\mathbf{x}_t, t)}{1-\bar{\alpha}_t}
$$

where $\hat{\mathbf{x}}_\theta(\mathbf{x}_t, t)$ is parameterized by a neural network that seeks to predict $\mathbf{x}_0$ from noisy image $\mathbf{x}_t$ and time index t. Then, the optimization problem simplifies to:

$$
\begin{aligned}
&\arg\min_\theta \mathcal{D}_{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \mid\mid p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)) \\
&= \arg\min_\theta \mathcal{D}_{KL}\left(\mathcal{N}\left(\mathbf{x}_{t-1}; \mu_q, \boldsymbol{\Sigma}_q(t)\right) \mid\mid \mathcal{N}\left(\mathbf{x}_{t-1}; \mu_\theta, \boldsymbol{\Sigma}_q(t)\right)\right) \\
&= \arg\min_\theta \frac{1}{2\sigma_q^2(t)} \frac{\bar{\alpha}_{t-1}(1-\alpha_t)^2}{(1-\bar{\alpha}_t)^2}\left[\|\hat{\mathbf{x}}_\theta(\mathbf{x}_t, t) - \mathbf{x}_0\|_2^2\right]
\end{aligned}
$$

Therefore, optimizing a VDM boils down to learning a neural network to predict the original ground truth image from an arbitrarily noisified version of it. Furthermore, minimizing the summation term of our derived ELBO objective across all noise levels can be approximated by minimizing the expectation over all timesteps:

$$\begin{aligned}
&\arg\min_{\theta} \sum_{t=2}^{T} \mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)} \left[\mathcal{D}_{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \mid\mid p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t))\right] && \textit{Consistency term} \\
&= \arg\min_{\theta} \mathbb{E}_{t\sim U\{2,T\}} \left[\mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)} \left[\frac{1}{2\sigma_q^2(t)} \frac{\bar{\alpha}_{t-1}(1-\alpha_t)^2}{(1-\bar{\alpha}_t)^2} \left[\|\hat{\mathbf{x}}_\theta(\mathbf{x}_t, t) - \mathbf{x}_0\|_2^2\right]\right]\right]
\end{aligned}$$

Remark 3.75. *The consistency term in loss has another equivalent parameterization. In this parameterization, we try to get rid of $\mathbf{x}_0$, to see if the consistency term can be written as a function of other variables. In our derivation of the form of $q(\mathbf{x}_t \mid \mathbf{x}_0)$, we can rearrange Equation* **??** *to show that:*

$$\mathbf{x}_0 = \frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\epsilon_0}{\sqrt{\bar{\alpha}_t}}$$

Plugging this into our previously derived true denoising transition mean μ_q, we can derive the following alternate parameterization:

$$\begin{aligned}
\mu_{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)} &= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t} \\
&= \frac{1}{\sqrt{\alpha_t}}\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}\sqrt{\alpha_t}}\epsilon_0
\end{aligned}$$

Likewise, we want to formulate our approximate denoising transition mean $\mu_{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}$ as close as possible to μ_q, hence we have:

$$\mu_{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} = \frac{1}{\sqrt{\alpha_t}}\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}\sqrt{\alpha_t}}\hat{\epsilon}_\theta(\mathbf{x}_t, t)$$

Namely using the neural network to modeling the noise in time step t and the corresponding optimization problem becomes:

$$\begin{aligned}
&\arg\min_{\theta} \mathcal{D}_{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \mid\mid p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)) \\
&= \arg\min_{\theta} \mathcal{D}_{KL}\left(\mathcal{N}\left(\mathbf{x}_{t-1}; \mu_q, \mathbf{\Sigma}_q(t)\right) \mid\mid \mathcal{N}\left(\mathbf{x}_{t-1}; \mu_\theta, \mathbf{\Sigma}_q(t)\right)\right) \\
&= \arg\min_{\theta} \frac{1}{2\sigma_q^2(t)} \frac{(1-\alpha_t)^2}{(1-\bar{\alpha}_t)\alpha_t} \left[\|\epsilon_0 - \hat{\epsilon}_\theta(\mathbf{x}_t, t)\|_2^2\right]
\end{aligned}$$

Here, $\hat{\epsilon}_\theta(\mathbf{x}_t, t)$ is a neural network that learns to predict the source noise $\epsilon_0 \sim \mathcal{N}(\epsilon; \mathbf{0}, \boldsymbol{I})$ that determines $\mathbf{x}_t$ from $\mathbf{x}_0$. We have therefore shown that learning a VDM by predicting the original image $\mathbf{x}_0$ is equivalent to learning to predict the noise; empirically, however, some works have found that predicting the noise resulted in better performance.

Remark 3.76. *In English, Tweedie's Formula states that the true mean of an exponential family distribution, given samples drawn from it, can be estimated by the maximum likelihood estimate of the samples (aka empirical mean) plus some correction term involving the score of the estimate. In the case of just one observed sample, the empirical mean is just the sample itself. It is commonly used to mitigate sample bias; if observed samples all lie on one end of the underlying distribution, then the negative score becomes large and corrects the naive maximum likelihood estimate of the samples towards the true mean.*

Mathematically, for a Gaussian variable $\mathbf{z} \sim \mathcal{N}(\mathbf{z}; \mu_z, \mathbf{\Sigma}_z)$, Tweedie's Formula states that:

$$\mathbb{E}\left[\mu_z \mid \mathbf{z}\right] = \mathbf{z} + \mathbf{\Sigma}_z \nabla_{\mathbf{z}} \log p(\mathbf{z})$$

In this case, we apply it to predict the true posterior mean of $\mathbf{x}_t$ given samples. As we previously defined: $\mathbf{x}_t \sim \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I})$. Then, by Tweedie's Formula, we have:

$$\mathbb{E}\left[\mu_{x_t} \mid \mathbf{x}_t\right] = \mathbf{x}_t + (1-\bar{\alpha}_t)\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t)$$

According to Tweedie's Formula, the best estimate for the true mean that $\mathbf{x}_t$ is generated from, $\mu_{x_t} = \sqrt{\bar{\alpha}_t}\mathbf{x}_0$, is defined as:

$$\sqrt{\bar{\alpha}_t}\mathbf{x}_0 = \mathbf{x}_t + (1-\bar{\alpha}_t)\nabla \log p(\mathbf{x}_t)$$
$$\therefore \mathbf{x}_0 = \frac{\mathbf{x}_t + (1-\bar{\alpha}_t)\nabla \log p(\mathbf{x}_t)}{\sqrt{\bar{\alpha}_t}}$$

Then, we can plug this into our previously derived ground-truth denoising transition mean of $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ once again and derive a new parameterization:

$$\begin{aligned}\mu_{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)} &= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\mathbf{x}_t + \sqrt{\bar{\alpha}_{t-1}}(1-\alpha_t)\mathbf{x}_0}{1-\bar{\alpha}_t} \\ &= \frac{1}{\sqrt{\alpha_t}}\mathbf{x}_t + \frac{1-\alpha_t}{\sqrt{\alpha_t}}\nabla \log p(\mathbf{x}_t)\end{aligned}$$

Therefore, we can also set our approximate denoising transition mean of $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ as:

$$\mu_{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)} = \frac{1}{\sqrt{\alpha_t}}\mathbf{x}_t + \frac{1-\alpha_t}{\sqrt{\alpha_t}}\mathbf{s}_\theta(\mathbf{x}_t, t)$$

and the corresponding optimization problem becomes:

$$\begin{aligned}&\arg\min_\theta \mathcal{D}_{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \;||\; p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)) \\ =&\arg\min_\theta \mathcal{D}_{KL}\left(\mathcal{N}\left(\mathbf{x}_{t-1}; \mu_q, \mathbf{\Sigma}_q(t)\right) \;||\; \mathcal{N}\left(\mathbf{x}_{t-1}; \mu_\theta, \mathbf{\Sigma}_q(t)\right)\right) \\ =&\arg\min_\theta \frac{1}{2\sigma_q^2(t)}\frac{(1-\alpha_t)^2}{\alpha_t}\left[\|\mathbf{s}_\theta(\mathbf{x}_t, t) - \nabla \log p(\mathbf{x}_t)\|_2^2\right]\end{aligned}$$

Here, $\mathbf{s}_\theta(\mathbf{x}_t, t)$ is a neural network that learns to predict the score function $\nabla_{\mathbf{x}_t} \log p(\mathbf{x}_t)$, which is the gradient of $\mathbf{x}_t$ in data space, for any arbitrary noise level t. The astute reader will notice that the score function $\nabla \log p(\mathbf{x}_t)$ looks remarkably similar in form to the source noise ϵ_0. This can be shown explicitly by combining Tweedie's Formula (Equation 79) with the reparameterization trick (Equation 71):

$$\begin{aligned}\mathbf{x}_0 = \frac{\mathbf{x}_t + (1-\bar{\alpha}_t)\nabla \log p(\mathbf{x}_t)}{\sqrt{\bar{\alpha}_t}} &= \frac{\mathbf{x}_t - \sqrt{1-\bar{\alpha}_t}\epsilon_0}{\sqrt{\bar{\alpha}_t}} \\ \therefore (1-\bar{\alpha}_t)\nabla \log p(\mathbf{x}_t) &= -\sqrt{1-\bar{\alpha}_t}\epsilon_0 \\ \nabla \log p(\mathbf{x}_t) &= -\frac{1}{\sqrt{1-\bar{\alpha}_t}}\epsilon_0\end{aligned}$$

Remark 3.77. *Assume $\mathbf{x}_T$ is the noise perturbation of $\mathbf{x}_0$, then we can represent $p(\mathbf{x}_T|\mathbf{x}_0)$ as*

$$\mathbf{x}_T = \mathbf{x}_0 + \sigma\epsilon, \text{ where } \epsilon \sim \mathcal{N}(0,1)$$
$$p(\mathbf{x}_T|\mathbf{x}_0) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(\mathbf{x}_T - \mathbf{x}_0)^2}{2\sigma^2}\right)$$

By the convolution rule of two probability distribution, we have

$$\begin{aligned}p(\mathbf{x}_T) &= \int p(\mathbf{x}_0)p_{Gaussian}(\mathbf{x}_T - \mathbf{x}_0)d\mathbf{x}_0 \\ &= \int p(\mathbf{x}_0)\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(\mathbf{x}_T - \mathbf{x}_0)^2}{2\sigma^2}\right)d\mathbf{x}_0\end{aligned}$$

To calculate the score function $\nabla_{\mathbf{x}_T} \log p(\mathbf{x}_T)$, we do

$$\begin{aligned}
\nabla_{\mathbf{x}_T} \log p(\mathbf{x}_T) &= \frac{1}{p(\mathbf{x}_T)} \nabla_{\mathbf{x}_T} p(\mathbf{x}_T) \\
&= \frac{1}{p(\mathbf{x}_T)} \nabla_{\mathbf{x}_T} \int p(\mathbf{x}_0) \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\mathbf{x}_T - \mathbf{x}_0)^2}{2\sigma^2}\right) d\mathbf{x}_0 \\
&= \frac{1}{p(\mathbf{x}_T)\sqrt{2\pi\sigma^2}} \int p(\mathbf{x}_0) \nabla_{\mathbf{x}_T} \exp\left(-\frac{(\mathbf{x}_T - \mathbf{x}_0)^2}{2\sigma^2}\right) d\mathbf{x}_0 \\
&= \frac{1}{p(\mathbf{x}_T)\sqrt{2\pi\sigma^2}} \int p(\mathbf{x}_0) \exp\left(-\frac{(\mathbf{x}_T - \mathbf{x}_0)^2}{2\sigma^2}\right) \nabla_{\mathbf{x}_T} \left(-\frac{(\mathbf{x}_T - \mathbf{x}_0)^2}{2\sigma^2}\right) d\mathbf{x}_0 \\
&= \frac{1}{p(\mathbf{x}_T)\sqrt{2\pi\sigma^2}} \int p(\mathbf{x}_0) p(\mathbf{x}_T|\mathbf{x}_0) \left(-\frac{2(\mathbf{x}_T - \mathbf{x}_0)}{2\sigma^2}\right) d\mathbf{x}_0 \\
&= -\frac{1}{p(\mathbf{x}_T)\sigma^2\sqrt{2\pi\sigma^2}} \int p(\mathbf{x}_0) p(\mathbf{x}_T|\mathbf{x}_0) (\mathbf{x}_T - \mathbf{x}_0) d\mathbf{x}_0 \\
&= -\frac{1}{\sigma^2\sqrt{2\pi\sigma^2}} \int \frac{p(\mathbf{x}_0) p(\mathbf{x}_T|\mathbf{x}_0)}{p(\mathbf{x}_T)} (\mathbf{x}_T - \mathbf{x}_0) d\mathbf{x}_0 \\
&= -\frac{1}{\sigma^2\sqrt{2\pi\sigma^2}} \int p(\mathbf{x}_0|\mathbf{x}_T) (\mathbf{x}_T - \mathbf{x}_0) d\mathbf{x}_0 \\
&= -\frac{1}{\sigma^2\sqrt{2\pi\sigma^2}} \left(\int p(\mathbf{x}_0|\mathbf{x}_T) \mathbf{x}_T d\mathbf{x}_0 - \int p(\mathbf{x}_0|\mathbf{x}_T) \mathbf{x}_0 d\mathbf{x}_0\right) \\
&= -\frac{1}{\sigma^2\sqrt{2\pi\sigma^2}} \left(\mathbf{x}_T \int p(\mathbf{x}_0|\mathbf{x}_T) d\mathbf{x}_0 - \int p(\mathbf{x}_0|\mathbf{x}_T) \mathbf{x}_0 d\mathbf{x}_0\right) \\
&= -\frac{1}{\sigma^2\sqrt{2\pi\sigma^2}} (\mathbf{x}_T - \mathbb{E}_{p(\mathbf{x}_0|\mathbf{x}_T)}(\mathbf{x}_0))
\end{aligned}$$

Hence, we have $\nabla_{\mathbf{x}_T} \log p(\mathbf{x}_T) \propto \mathbf{x}_T - \mathbb{E}_{p(\mathbf{x}_0|\mathbf{x}_T)}(\mathbf{x}_0)$. We use a neural network $s_\theta(\mathbf{x}_T)$ to parameterize $\mathbb{E}_{p(\mathbf{x}_0|\mathbf{x}_T)}(\mathbf{x}_0)$, which gives us $s_\theta(\mathbf{x}_T) = \mathbf{x}_T - \nabla_{\mathbf{x}_T} \log p(\mathbf{x}_T) = \mathbf{x}_T + \nabla_{\mathbf{x}_T}(-\log p(\mathbf{x}_T))$. It states that to denoise the image, we should move along with the gradient of the likelihood $-\log p(\mathbf{x}_T)$ which will give us a higher likelihood. In practice, we use known noise residuals between each step as the surrogate of $\nabla_{\mathbf{x}_T} \log p(\mathbf{x}_T)$.

Remark 3.78. *The key challenge is the fact that the estimated score functions are inaccurate in low-density regions, where few data points are available for computing the score-matching objective. This is expected as score matching minimizes the Fisher divergence*

$$\mathbb{E}_{p(\mathbf{x})}[\|\nabla_{\mathbf{x}} \log p(\mathbf{x}) - \mathbf{s}_\theta(\mathbf{x})\|_2^2] = \int p(\mathbf{x}) \|\nabla_{\mathbf{x}} \log p(\mathbf{x}) - \mathbf{s}_\theta(\mathbf{x})\|_2^2 \mathrm{d}\mathbf{x}.$$

Since the l_2 differences between the true data score function and score-based model are weighted by $p(\mathbf{x})$, they are largely ignored in low-density regions where $p(\mathbf{x})$ is small. This behavior can lead to subpar results, as illustrated by the figure below: When sampling with Langevin dynamics, our initial sample is highly likely in low-density regions when data reside in a high dimensional space. Therefore, having an inaccurate score-based model will derail Langevin dynamics from the very beginning of the procedure, preventing it from generating high-quality samples that are representative of the data. How can we bypass the difficulty of accurate score estimation in regions of low data density? Our solution is to perturb data points with noise and train score-based models on the noisy data points instead. When the noise magnitude is sufficiently large, it can populate low data density regions to improve the accuracy of estimated scores. For example, here is what happens when we perturb a mixture of two Gaussians perturbed by additional Gaussian noise.

Definition 3.94. **Latent diffusion models (stable diffusion)** are based on the idea of iteratively applying a diffusion process to the latent variables. The model is trained to estimate the likelihood of the next element in the sequence, given the previous elements and the current state of the latent variables.

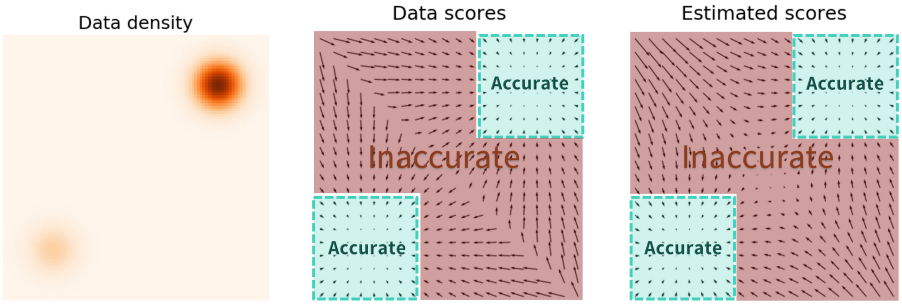


Figure 49: Estimated scores are only accurate in high density regions.

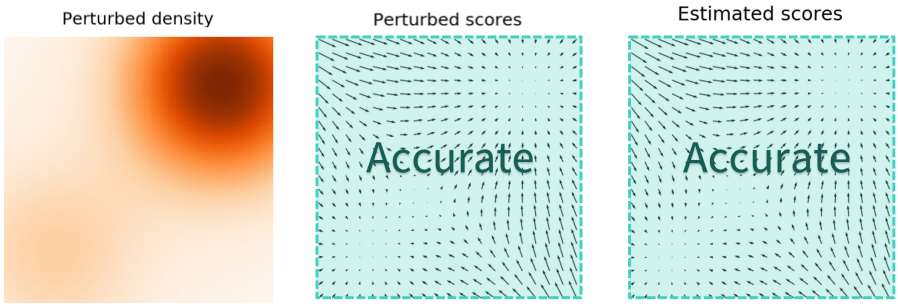


Figure 50: Estimated scores are accurate everywhere for the noise-perturbed data distribution due to reduced low data density regions.

3.4.5 Generative Adversarial Networks

Definition 3.95. **Generative Adversarial Networks** '[Goodfellow et al., 2014] generative part generates candidates while the discriminative part evaluates them.

Example 48. The workflow of GAN:

1. Optimize discriminator D with generator G initialized and fixed:
   (a) Sample $\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(n)}$ from p_{data}, which are positive samples;
   (b) Sample $\mathbf{z}^{(1)}, \cdots, \mathbf{z}^{(n)}$ from p_{prior}, which give negative samples;
   (c) Generate $\mathbf{x}^{(1)'}, \cdots, \mathbf{x}^{(n)'}$ from $G(\mathbf{z}) = \mathbf{x}^{(i)'}$;
   (d) Maximize $V(G, D) = \frac{1}{n}\sum_{i=1}^{n} \log D(\mathbf{x}^{(i)}) + \frac{1}{n}\sum_{i=1}^{n} \log(1 - D(G(\mathbf{z}^{(i)})))$ 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 $\{\mathbf{x}^{(i)}\}$ given, sample $\mathbf{z}^{(1)}, \cdots, \mathbf{z}^{(n)}$ from p_{prior}, which give negative samples;
   (b) Minimize $V(G, D) = \frac{1}{n}\sum_{i=1}^{n} \log D(\mathbf{x}^{(i)}) + \frac{1}{n}\sum_{i=1}^{n} \log(1 - D(G(\mathbf{z}^{(i)})))$ (equivalent to minimizing $\frac{1}{n}\sum_{i=1}^{n} \log(1 - D(G(\mathbf{z}^{(i)})))$) by gradient descent $\theta_G \leftarrow \theta_G - \eta \nabla V(G)$ for a few iterations.
3. Iterate from the beginning.

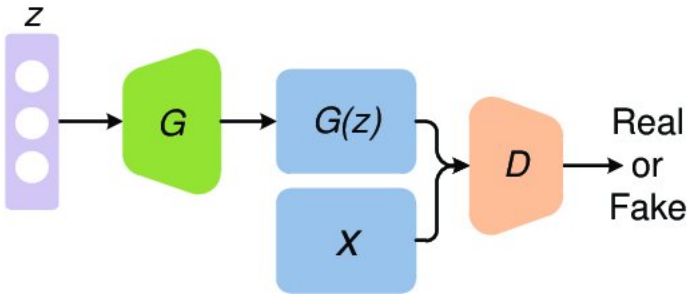


Figure 51: Generative Adversarial Networks

Example 49. Pseudocode[26] of GAN:

```
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))

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))

def discriminator_train(x_real):
```

[26] https://github.com/lyeoni/pytorch-mnist-GAN/blob/master/pytorch-mnist-GAN.ipynb

```
        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 3.79. *The workflow of GAN above can be interpreted as:*

1. *Find the JS divergence between the p_{data} and current p_G;*

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

Remark 3.80.

- *The input and output of G are vector $\mathbf{z}$ (noise) and vector $\mathbf{x}$ (negative sample);*

- *The input of D is vector $\mathbf{x}_n$ (negative sample) and vector $\mathbf{x}_p$ (positive sample), while the output is simply a scalar between 0 and 1.*

Remark 3.81. *Both maximizing $\frac{1}{n}\sum_{i=1}^{n}\log D(\mathbf{x}^{(i)})+\frac{1}{n}\sum_{i=1}^{n}\log(1-D(G(\mathbf{z}^{(i)})))$ and minimizing $\frac{1}{n}\sum_{i=1}^{n}\log(1-D(G(\mathbf{z}^{(i)})))$ 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 $\prod_{i=1}^{n}p_{data}(x)$, where x is the real-world sample. However, the true p_{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}.

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

Remark 3.82. *The objective optimization equation reads as:*

$$\min_G \max_D V(G,D) = \min_G \max_D \left(\mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))] \right)$$

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

Proof. First, we have

$$\begin{aligned}
V(G,D) &= E_{\mathbf{x}\sim p_{data}}[\log D(\mathbf{x})] + E_{\mathbf{z}\sim p_{prior}}[\log(1 - D(G(\mathbf{z})))] \\
&= E_{\mathbf{x}\sim p_{data}}[\log D(\mathbf{x})] + E_{\mathbf{x}\sim p_G(\mathbf{x})}[\log(1 - D(\mathbf{x}))] \\
&= \int p_{data}(\mathbf{x}) \log D(\mathbf{x}) d\mathbf{x} + \int p_G(\mathbf{x}) \log(1 - D(\mathbf{x})) d\mathbf{x} \\
&= \int [p_{data}(\mathbf{x}) \log D(\mathbf{x}) + p_G(\mathbf{x})(\log(1 - D(\mathbf{x})))] d\mathbf{x}
\end{aligned}$$

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

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

Therefore, $\max_D V(G,D)$ can be written as

$$\begin{aligned}
\max_D V(G,D) = V(G,D^*) &= \int p_{data}(\mathbf{x}) \cdot \log\left(\frac{p_{data}}{p_{data} + p_G}\right) + p_G(\mathbf{x}) \cdot \log\left(\frac{p_G}{p_{data} + p_G}\right) d\mathbf{x} \\
&= \int p_{data}(\mathbf{x}) \cdot \log\left(\frac{p_{data}}{(p_{data} + p_G)/2}\right) d\mathbf{x} - \log 2 + \int p_G(\mathbf{x}) \cdot \log\left(\frac{p_G}{(p_{data} + p_G)/2}\right) d\mathbf{x} - \log 2 \\
&= D_{KL}\left(p_{data} \| \frac{p_{data} + p_G}{2}\right) + D_{KL}\left(p_G \| \frac{p_{data} + p_G}{2}\right) - 2\log 2 \\
&= D_{JS}(p_{data} \| p_G) - 2\log 2
\end{aligned}$$

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

□

Remark 3.83. *An individual element of $\mathbf{z}$ in a vanilla GAN does not necessarily correspond to semantic features of data, like age and gender etc.*

Remark 3.84. *In the task of image translation, the input of the GAN is not solely a random variable, but also with an image in the field A (edges) - the output is an image in the field B (realistic photos). The loss function comparison between vanilla and conditional GAN:*

$$\textit{Vanilla GAN: } V(G,D) = \frac{1}{n}\sum_{i=1}^{n}\log D(\mathbf{x}_B^{(i)}) + \frac{1}{n}\sum_{i=1}^{n}\log(1 - D(G(\mathbf{z}^{(i)}, \mathbf{x}_A^{(i)})))$$

$$\textit{Conditional GAN: } V(G,D) = \frac{1}{n}\sum_{i=1}^{n}\log D(\mathbf{x}_A^{(i)}, \mathbf{x}_B^{(i)}) + \frac{1}{n}\sum_{i=1}^{n}\log(1 - D(\mathbf{x}_A^{(i)}, G(\mathbf{z}^{(i)}, \mathbf{x}_A^{(i)})))$$

The main difference is that the discriminator in the conditional GAN would take both input and output into consideration. Another primary difference between conventional generative task and image translation task is that the image translation is supervised with ground-truth, hence the goal of the generator is not only to fool the discriminator but also mimic the real sample as close as possible.

3.4.6 Flow-based Generative Models

3.5 Classification and Regression Models with Deep Learning

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.

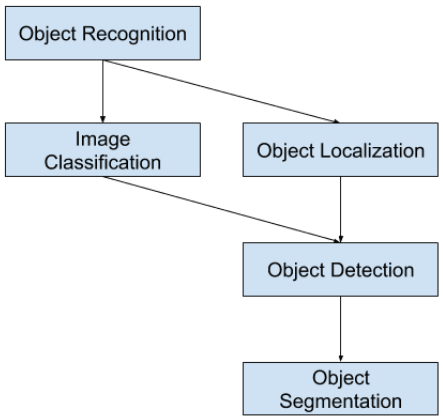


Figure 52: Overview of object recognition computer vision tasks

3.5.1 Datasets

Definition 3.96. **CIFAR-10** (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 32×32 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 3.97. **ImageNet** 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.

3.5.2 LeNet

At a high level, LeNet (LeNet-5)[LeCun et al., 1998] 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:

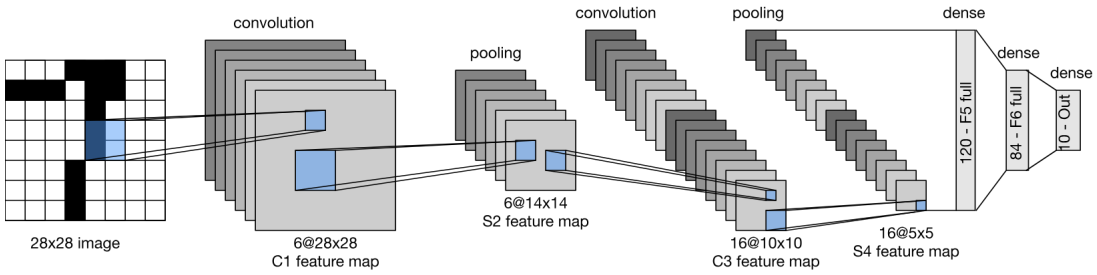


Figure 53: Data flow in LeNet. The input is a handwritten digit, the output a probability over 10 possible outcomes. Courtesy of [Zhang et al., 2021], section 6.6.

Remark 3.85. *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:

```
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))
```

3.5.3 AlexNet

The design philosophies of AlexNet[Krizhevsky et al., 2012] 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.
3. AlexNet used the max pooling instead of the average pooling as its activation function.

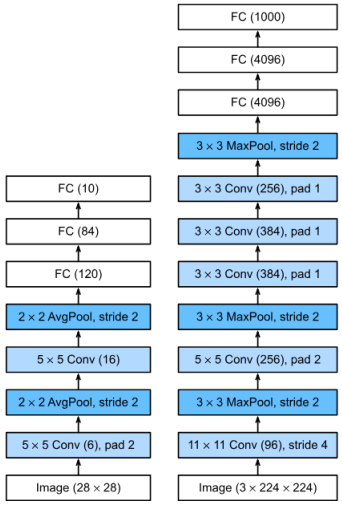


Figure 54: From LeNet (left) to AlexNet (right). Courtesy of [Zhang et al., 2021], section 7.1.

Remark 3.86. *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:

```
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))
```

3.5.4 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[Simonyan and Zisserman, 2014], for the first time, proposed the idea of using blocks to build deep learning frame work.

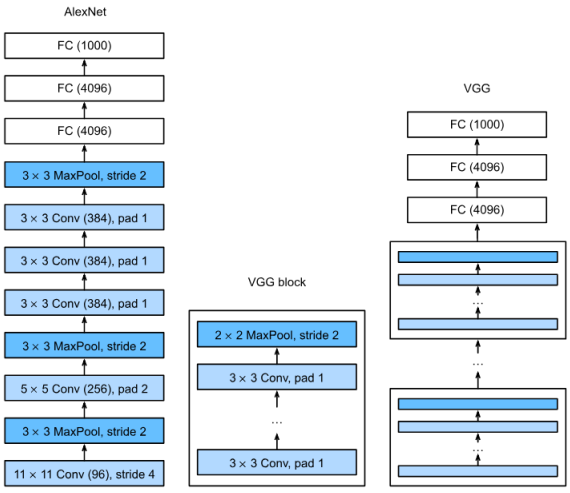


Figure 55: From AlexNet to VGG that is designed from building blocks. Courtesy of [Zhang et al., 2021], section 7.2.

Remark 3.87. *In their VGG paper, Simonyan and Ziserman 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:

```
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
    # The convolutional part
    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, 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 3.88. *Evolution path of LeNet, AlexNet, and VGG:*

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

3.5.5 DenseNet

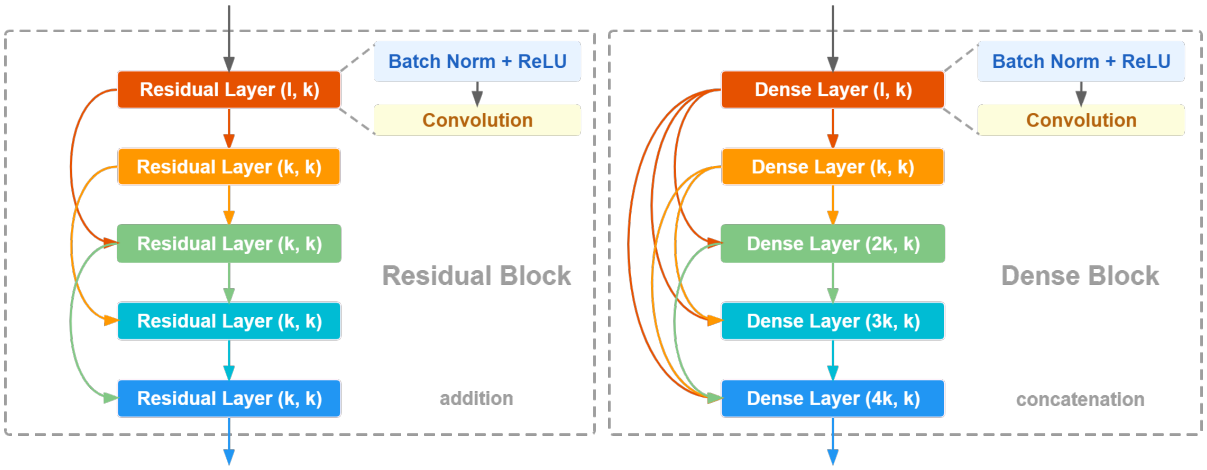


Figure 56: Comparison between Residual block and dense block, parenthesis reads as (input channel , output channel)

3.5.6 R-CNN series

Example 50. 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.

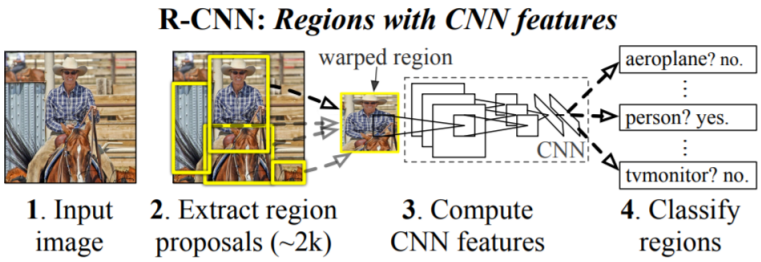


Figure 57: Regions with CNN features[Girshick et al., 2014]

Remark 3.89. *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 3.98. **Selective Search** is a region proposal algorithm used in object detection.

Example 51. 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.

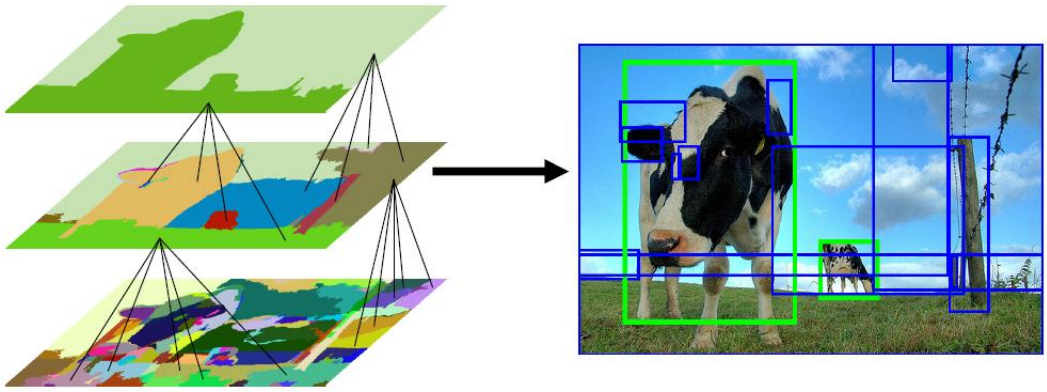


Figure 58: Selective search[Uijlings et al., 2013]

Example 52. 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.

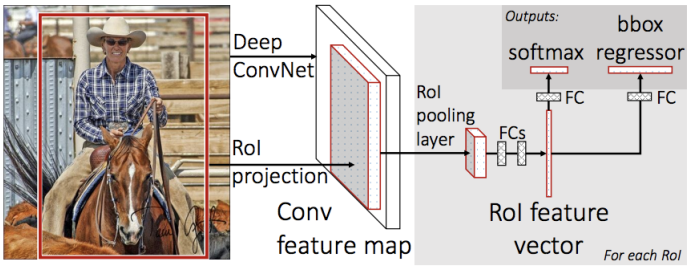


Figure 59: Fast CNN[Girshick, 2015]

Remark 3.90. *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 53. 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[27], 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.

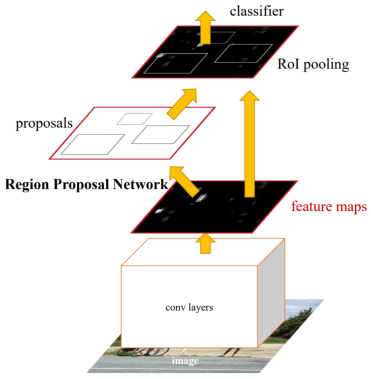


Figure 60: Faster CNN [Ren et al., 2015]

Definition 3.99. **Region proposal network (RPN)** is a region proposal generator used in object detection.

Example 54. The workflow of RPN:

1. Feed the 256D feature map generated from VGG into PRN;

[27]ROI 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.

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.

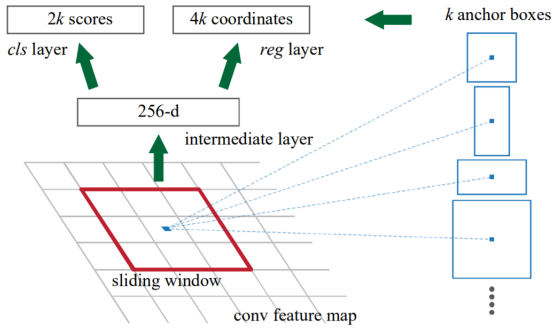


Figure 61: RPN: for each anchor box, it features negative and positive scores (after softmax), and four coordinates of x, y, h, w (after regression). For a convolutional feature map of a size $W \times H$ (typically ~2,400), there are WHk anchors in total. [Ren et al., 2015]

3.5.7 Transformers

Definition 3.100. **Self-attention function** [Vaswani et al., 2017] is formulated as

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

where d is the dimension of embedding vectors.

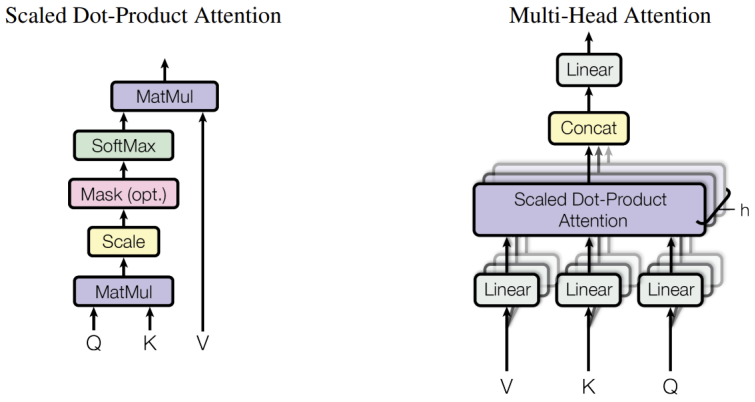


Figure 62: Left: Scaled Dot-Product Attention. Right: Multi-Head attention consists of several attention layers running in parallel.

Remark 3.91. *Scaling the attention score by $1/\sqrt{d}$ helps to control the magnitude of the value and keep the gradient size reasonable, which leads to easier training.*

Remark 3.92. *Softmax is conducted in each row in the attention filter in Figure 63*

Remark 3.93. *Multi-head attention is concatenated by a single-head attention tensor along the feature vector dimensions.*

Remark 3.94. *In practice,* $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{n \times d}$ *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,* $\mathrm{softmax}(\mathbf{Q}\mathbf{K}^\top/\sqrt{d}) \in \mathbb{R}^{n \times n}$ *represents the attention score inside of the sequence.*

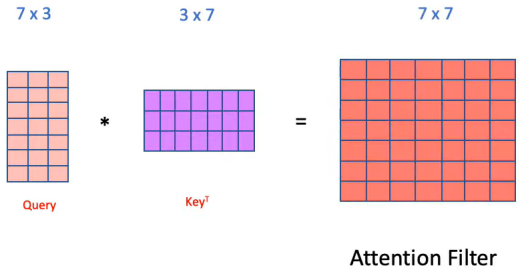


Figure 63: Illustration of how attention score is calculated. 7 is the sequence length and 3 is the dimension of token representation.

Example 55. Below shows the structure of a vanilla transformer encoder layer:

```
import torch
import torch.nn.functional as F

class SelfAttention(nn.Module):
    def __init__(self, dim):
        super(SelfAttention, self).__init__()
        self.heads = heads
        self.dim = d_model

        # Define weight matrices for queries, keys, and values
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)

        # Output linear layer
        self.W_out = nn.Linear(d_model, d_model)

    def forward(self, x):
        batch_size, seq_len, _ = x.size()

        # Linearly transform queries, keys, and values
        queries = self.W_q(x)
        keys = self.W_k(x)
        values = self.W_v(x)

        # Reshape queries, keys, and values to have 'self.heads' as the new dimension
        queries = queries.view(batch_size, seq_len, self.dim)
        keys = keys.view(batch_size, seq_len, self.dim)
        values = values.view(batch_size, seq_len, self.dim)
```

```
        # Transpose dimensions for matrix multiplication
        queries = queries.transpose(1, 2)
        keys = keys.transpose(1, 2)
        values = values.transpose(1, 2)

        # Calculate self-attention scores
        scores = torch.matmul(queries, keys.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.dim, dtyp
        attention_weights = F.softmax(scores, dim=-1)
        attended_values = torch.matmul(attention_weights, values)
        output = self.W_out(attended_values)

        return output

# Example usage
sequence_length = 5
d_model = 64  # Dimension of the input and output vectors
num_heads = 8  # Number of attention heads

# Create a random input tensor
x = torch.rand(2, sequence_length, d_model)

# Create a self-attention module
self_attention = SelfAttention(d_model, num_heads)

# Apply self-attention to the input tensor
output = self_attention(x)

print("Input Tensor:")
print(x)
print("Output Tensor after Self-Attention:")
print(output)
```

3.5.8 Graph Neural Networks[Sanchez-Lengeling et al., 2021]

Definition 3.101. **Graph neural network (GNN)** is an optimizable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances).

GNNs adopt a "graph-in, graph-out"[28] architecture meaning that these model types accept a graph as input, with information loaded into its nodes, edges, and global context, and progressively transform these embeddings, without changing the connectivity of the input graph.

[28] Because a GNN does not update the connectivity of the input graph, we can describe the output graph of a GNN with the same adjacency list and the same number of feature vectors as the input graph. But, the output graph has updated embeddings, since the GNN has updated each of the node, edge and global-context representations.

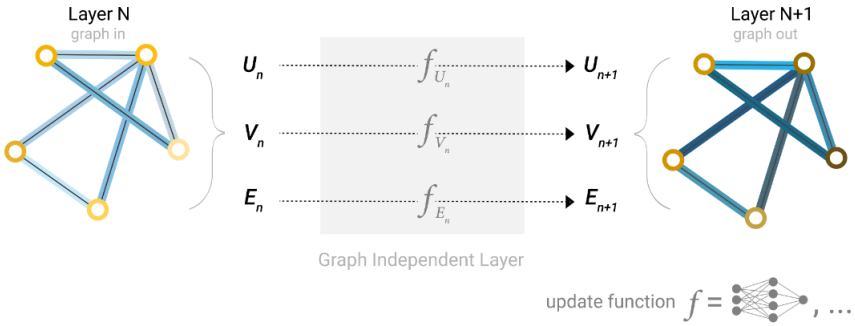


Figure 64: A single layer of a simple GNN. A graph is the input and each component (V, E, U) gets updated by an MLP f to produce a new graph. Each function subscript indicates a separate function for a different graph attribute at the n-th layer of a GNN model.

Example 56. Application of GNNs:

- Graph-level task: predicting the property of an entire graph. For example, for a molecule represented as a graph, we might want to predict what the molecule smells like, or whether it will bind to a receptor implicated in a disease.
- Node-level task: predicting the identity or role of each node within a graph.
- Edge-level task: predicting the identity or role of each edge within a graph. One example of edge-level inference is in image scene understanding. Beyond identifying objects in an image, deep learning models can be used to predict the relationship between them.

Sometimes, you want to make predictions on nodes when information in edges is available, but the one in nodes is unavailable. We need a way to collect information from edges and give them to nodes for prediction. We can do this by pooling.

Definition 3.102. **Pooling** proceeds in two steps:

1. For each item to be pooled, gather each of their embeddings and concatenate them into a matrix.
2. The gathered embeddings are then aggregated, usually via a sum operation.

We represent the pooling operation by the letter ρ, and denote that we are gathering information from edges to nodes as $\rho_{E_n \to V_n}$.

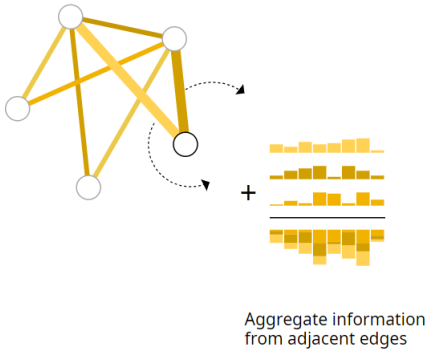


Figure 65: Aggregate information for nodes from adjacent edges.

Example 57. So If we only have edge-level features, and are trying to predict binary node information, we can use pooling to route (or pass) information to where it needs to go. The model looks like below:

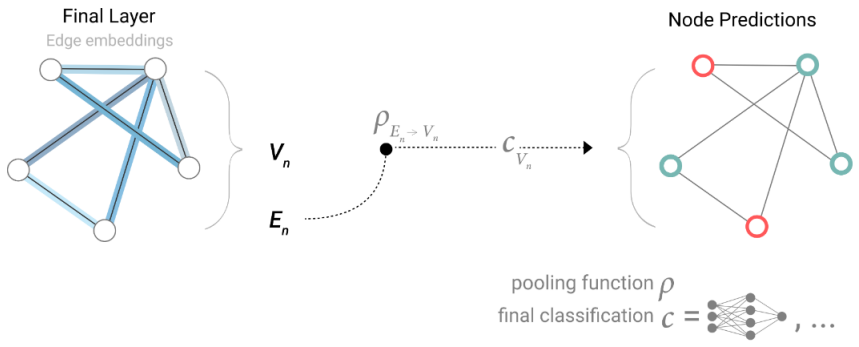


Figure 66: Aggregate information for nodes from adjacent edges to perform node prediction.

In order to make our learned embeddings aware of graph connectivity, we can use message passing, where neighboring nodes or edges exchange information and influence each other's updated embeddings.

Definition 3.103. **Message passing** works in three steps:

1. For each node in the graph, gather all the neighboring node embeddings (or messages), which is the g function described above;
2. Aggregate all messages via an aggregate function (like sum);
3. All pooled messages are passed through an update function, usually a learned neural network.

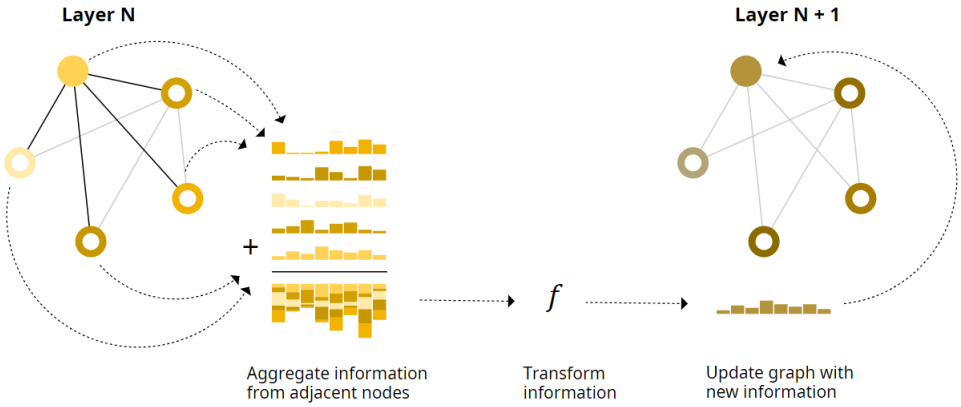


Figure 67: This sequence of operations, when applied once, is the simplest type of message-passing GNN layer.

Remark 3.95. ***This is reminiscent of standard convolution:*** *In essence, message passing and convolution are operations to aggregate and process the information of an element's neighbors in order to update the element's value.*

In graphs, the element is a node, and in images, the element is a pixel. However, the number of neighboring nodes in a graph can be variable, unlike in an image where each pixel has a set number of neighboring elements.

Remark 3.96. *The node and edge information stored in a graph are not necessarily the same size or shape, so it is not immediately clear how to combine them. One way is to learn a linear mapping from the space of edges to the space of nodes, and vice versa. Alternatively, one may concatenate them together before the update function.*

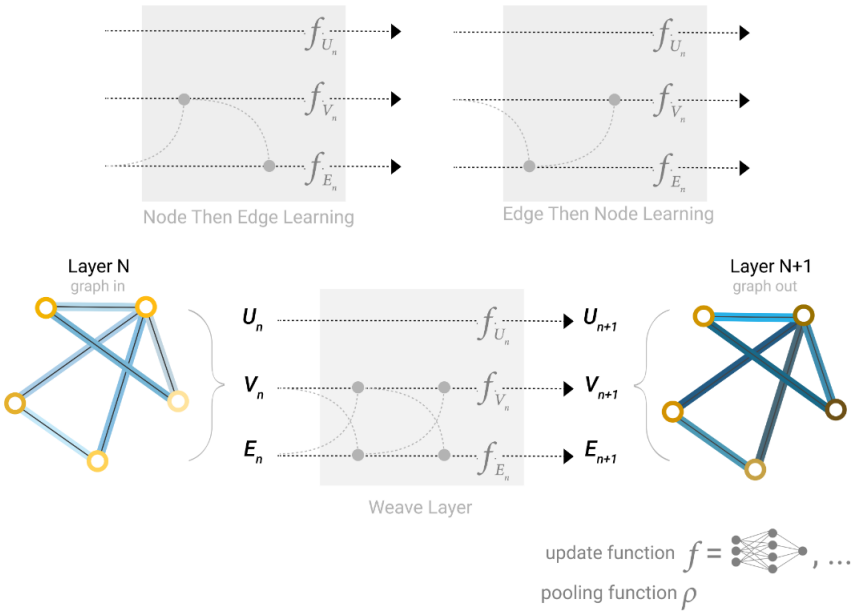


Figure 68: Some of the different ways we might combine edge and node representation in a GNN layer.

3.6 Physics Informed Machine Learning

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 3.104. In an **inverse problem**, you are given the effect and the task is to recover the cause.

Definition 3.105. **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 $a_0(x), ..., a_n(x)$ 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 58. 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 3.106. **Neural ODE** can be represented by the following continuous-time formulation:

$$\frac{dy(x)}{dx} = f(y(x), x, \theta) \rightarrow \text{Neural network}$$

Definition 3.107. **Runge–Kutta methods(RK4)** Let an initial value problem be specified as follows: The only things we know are the slope of the tangent line to the curve y at any point

$$\frac{dy}{dt} = f(t, y)$$

and the initial condition (t_0, y_0), namely

$$y(t_0) = y_0.$$

We want to approximate the original function y, which equals to $\int f(t,y)$. For the $n+1^{th}$ iteration, we have the approximation

$$y_{n+1} = y_n + \frac{1}{6}h(k_1^n + 2k_2^n + 2k_3^n + k_4^n),$$
$$t_{n+1} = t_n + h,$$

where h is the step size and

$$k_1^n = f(t_n, y_n) \quad \triangleright \text{ The slope at the beginning of the interval}$$
$$k_2^n = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1^n}{2}\right) \quad \triangleright \text{ The slope at the midpoint of the interval, using } y \text{ and } k_1^n$$
$$k_3^n = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2^n}{2}\right) \quad \triangleright \text{ The slope at the midpoint of the interval, using } y \text{ and } k_2^n$$
$$k_4^n = f(t_n + h, y_n + hk_3) \quad \triangleright \text{ The slope at the end of the interval, using } y \text{ and } k_3^n$$

Remark 3.97. *When $k_4^n = k_3^n = k_2^n = k_1^n = f(t_n, y_n)$, it's the simplest Runge–Kutta method, namely the Euler method.*

Remark 3.98. *In an ideal world, we can integrate a function by integration rules, like $\int e^x = e^x + c$. However, in the real world, we can only use a numerical procedure to integrate a random function. That's why we call the result of RK4 or Euler methods as integral curve, which is simply the result after integration.*

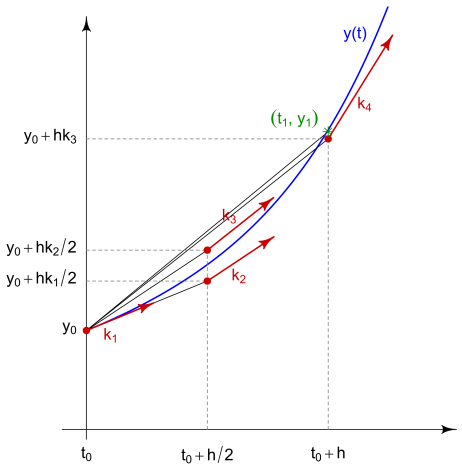


Figure 69: RK4

Definition 3.108. **Integral curve** is a parametric curve that represents a specific solution to an ordinary differential equation or system of equations. If the differential equation is represented as a vector field or slope field, then the corresponding integral curves are tangent to the field at each point.

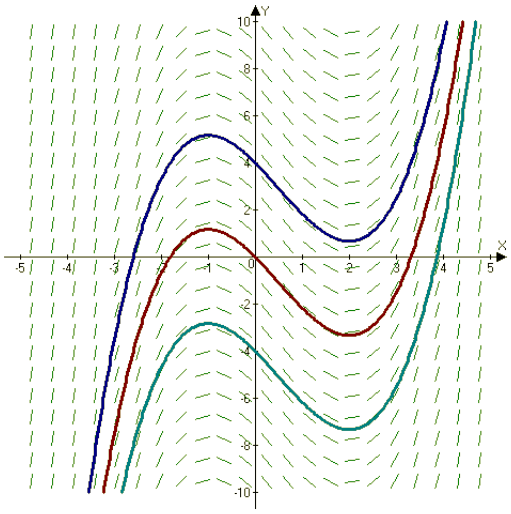


Figure 70: Integral curve

Definition 3.109. **Partial differential equation** is an equation that imposes relations between the various partial derivatives of a multivariable function. For example:

$$Au_{xx} + 2Bu_{xy} + Cu_{yy} + \cdots (\text{lower order terms}) = 0,$$

where the coefficients $A, B, C \cdots$ may depend upon x and y.

Remark 3.99. *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 59. When the equation involves a gradient ∇ or Laplacian Δ, it is a partial differential equation.

3.6.1 PINN

[Raissi et al., 2019] In one space dimension, Burger's equation along with Dirichlet boundary conditions reads as

$$\begin{aligned} u_t + uu_x - \frac{0.01}{\pi}u_{xx} &= 0 \qquad\qquad x \in [-1,1], t \in [0,1] \\ u(0,x) &= -\sin(\pi x) \\ u(t,-1) &= u(t,1) = 0 \end{aligned}$$

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

```
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

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

$$\mathrm{MSE}_u = \frac{1}{N_u}\sum_{i=1}^{N_u} |u(t_u^i, x_u^i) - u^i|^2$$

$$\mathrm{MSE}_f = \frac{1}{N_u}\sum_{i=1}^{N_u} |f(t_u^i, x_u^i) - 0|^2$$

Here, $\{t_u^i, x_u^i, u^i\}_{i=1}^{N_u}$ denotes the initial and boundary(cross mark in Figure 71) training data on $u(t,x)$ and $\{t_f^i, x_f^i\}_{i=1}^{N_f}$ specify the collocations points for $f(t,x)$.

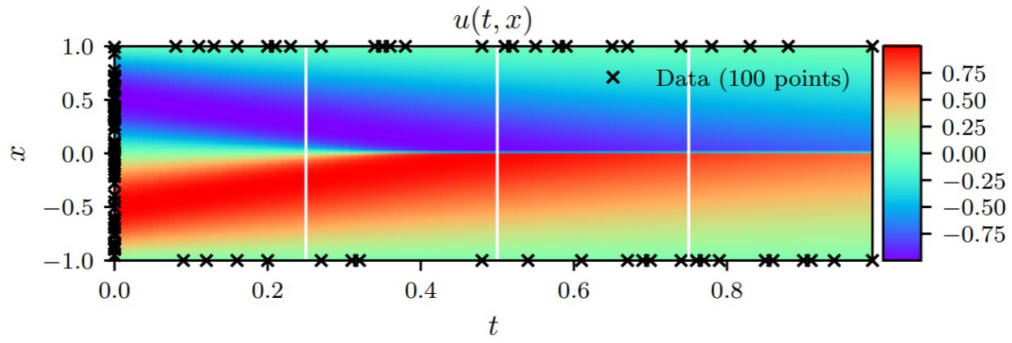


Figure 71: Solved two variables function: $u(t,x)$, courtesy of [Raissi et al., 2019].

Remark 3.100. *The neural network presents function u in a **map**($\mathbb{R}^2 \to \mathbb{R}$) manner, which requires an input and an output, while Figure 71 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($\mathbb{R}^n \to \mathbb{R}, n \leq 3$), we can regard the deep neural network for solving PDE as a n dimensional **"image"**.*

Example 60. The 2D Poisson's equation reads as

$$\Delta u = f$$

Consequently, we can have the code snippet below

```
    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.

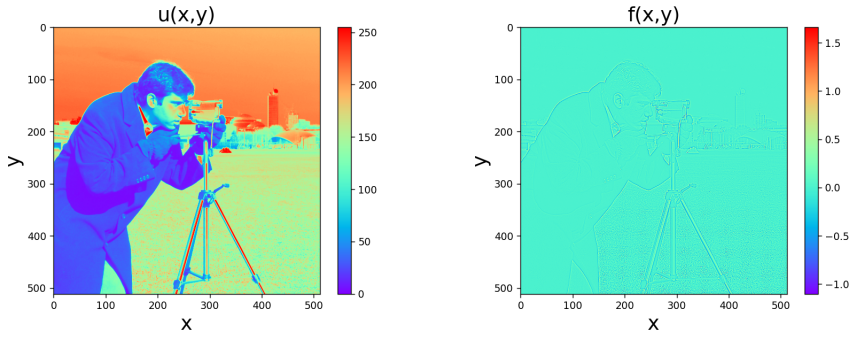


Figure 72: The recovered function $u(x, y)$ is a meaningful image.

3.7 Meta Learning

Definition 3.110. **Support set** is a very small set of labeled images.

Example 61. 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 3.111. **Few-shot learning** is the problem of making predictions based on a support set.

Remark 3.101. *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 62. 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 63. 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.

3.8 PyTorch - Software in Deep Learning

3.8.1 `torch.nn`

Function 3.1. `L1Loss()` creates a criterion that measures the mean absolute error (MAE) between each element in the prediction and target.

Example 64. The unreduced (i.e. with reduction set to `none`) loss can be described as:

$$l(x,y) = [l_1, \cdots, l_N]^\top = L,$$
$$l_n(x[n,0,h,w], y[n,0,h,w]) = |x[n,0,h,w] - y[n,0,h,w]|,$$

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

Example 65. The unreduced (i.e. with reduction set to `none`) loss can be described as:

$$l(x,y) = [l_1, \cdots, l_N]^\top = L,$$
$$l_n(x[n,0,h,w], y[n,0,h,w]) = |x[n,0,h,w] - y[n,0,h,w]|^2,$$

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 3.3. `BCELoss()` creates a criterion that measures the Binary Cross Entropy between the prediction and the target.

Example 66. The unreduced (i.e. with reduction set to `none`) loss can be described as:

$$l(x,y) = [l_1, \cdots, l_N]^\top,$$
$$l_n(x[n,0,h,w], y[n,0,h,w]) = -w_n(y[n,0,h,w]\log(x[n,0,h,w]) + (1 - y[n,0,h,w])\log(1 - x[n,0,h,w])),$$

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 3.102. *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 3.103. *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 3.4. `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 67. The unreduced (i.e. with reduction set to `none`) loss can be described as:

$$l(x,y) = [l_1, \cdots, l_N]^\top,$$
$$l_n(x[n,0,h,w], y[n,0,h,w]) = -w_n(y[n,0,h,w]\log(\sigma(x[n,0,h,w])) + (1-y[n,0,h,w])\log(1-\sigma(x[n,0,h,w]))),$$

where N is the batch size, σ 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 3.104. *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 3.5. `LogSoftmax()` can be simplified as:

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

Function 3.6. `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 68. The loss can be described as:

$$l(x,y) = [l_1, \cdots, l_N]^\top,$$
$$\text{loss}(x[n,:,h,w], \text{class}) = -w[\text{class}] \cdot x[n, \text{class}, h, w],$$

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 3.7. `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 69. The loss can be described as:

$$l(x,y) = [l_1, \cdots, l_N]^\top,$$
$$\text{loss}(x[n,:,h,w], \text{class}) = -w[\text{class}]\log\left(\frac{\exp(x[n,\text{class},h,w])}{\sum_j \exp(x[n,j,h,w])}\right),$$

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 3.105. *The second dimension of x should always be the class dimension.*

Function 3.8. `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_target` set to True. This criterion expects a target Tensor of the same size as the input Tensor.

Example 70. The unreduced (i.e. with reduction set to `none`) loss can be described as:

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

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

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

Function 3.9. `AdaptiveAvgPool2d(output_size)` applies a 2D adaptive average pooling (automatically set stride and kernel size) over an input signal composed of several input planes. The output is of size (H_{out}, W_{out}), for any input size. The number of output features is equal to the number of input planes.

$$\text{Input:}(N, C, H_{in}, W_{in})$$
$$\text{Output:}(N, C, H_{out}, W_{out})$$

Example 71. The adaptive pooling operation is particularly useful when you have a model with fully connected layers following the convolutional layers. The output size of the convolutional layers may vary depending on the input size, but the fully connected layers require a fixed-size input.

In a 1000-class classification task, the final output of the network can be any shape, for example (N, C, H', W'). we can use a `AdaptiveAvgPool2d` layer to condense it as $(N, C, 1, 1)$, then send it through a linear layer to project it as a tensor with shape $(N, 1000)$.

References

[Arvanitidis et al., 2016] Arvanitidis, G., Hansen, L. K., and Hauberg, S. r. (2016). A locally adaptive normal distribution. In Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc.

[Chang et al., 2015] Chang, A. X., Funkhouser, T., Guibas, L., Hanrahan, P., Huang, Q., Li, Z., Savarese, S., Savva, M., Song, S., Su, H., Xiao, J., Yi, L., and Yu, F. (2015). ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago.

[Domingos, 2012] Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87.

[Girshick, 2015] Girshick, R. (2015). Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448.

[Girshick et al., 2014] Girshick, R., Donahue, J., Darrell, T., and Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc.

[Ho et al., 2020] Ho, J., Jain, A., and Abbeel, P. (2020). Denoising diffusion probabilistic models.

[hyun Lee,] hyun Lee, D. Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France. PMLR.

[Kingma and Ba, 2017] Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

[Kingma and Welling, 2014] Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

[Kruskal, 1964] Kruskal, J. B. (1964). Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, volume 86, pages 2278–2324.

[Lin et al., 2013] Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

[Luo, 2022] Luo, C. (2022). Understanding diffusion models: A unified perspective.

[Luo et al., 2018] Luo, P., Wang, X., Shao, W., and Peng, Z. (2018). Towards understanding regularization in batch normalization.

[Miller, 1995] Miller, G. A. (1995). Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41.

[Odena et al., 2016] Odena, A., Dumoulin, V., and Olah, C. (2016). Deconvolution and checkerboard artifacts. *Distill*.

[Radford et al., 2021] Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. (2021). Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning*, pages 8748–8763. PMLR.

[Raissi et al., 2019] Raissi, M., Perdikaris, P., and Karniadakis, G. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707.

[Ren et al., 2015] Ren, S., He, K., Girshick, R., and Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99.

[Sanchez-Lengeling et al., 2021] Sanchez-Lengeling, B., Reif, E., Pearce, A., and Wiltschko, A. B. (2021). A gentle introduction to graph neural networks. *Distill*. https://distill.pub/2021/gnn-intro.

[Schuhmann et al., 2022] Schuhmann, C., Beaumont, R., Vencu, R., Gordon, C., Wightman, R., Cherti, M., Coombes, T., Katta, A., Mullis, C., Wortsman, M., Schramowski, P., Kundurthy, S., Crowson, K., Schmidt, L., Kaczmarczyk, R., and Jitsev, J. (2022). Laion-5b: An open large-scale dataset for training next generation image-text models.

[Selvaraju et al., 2019] Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. (2019). Grad-CAM: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

[Suárez et al., 2021] Suárez, J. L., García, S., and Herrera, F. (2021). A tutorial on distance metric learning: Mathematical foundations, algorithms, experimental analysis, prospects and challenges. *Neurocomputing*, 425:300–322.

[Tenenbaum et al., 2000] Tenenbaum, J. B., de Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323.

[Teye et al., 2018] Teye, M., Azizpour, H., and Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks.

[Uijlings et al., 2013] Uijlings, J. R. R., van de Sande, K. E. A., Gevers, T., and Smeulders, A. W. M. (2013). Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

[Wu and He, 2018] Wu, Y. and He, K. (2018). Group normalization. *CoRR*, abs/1803.08494.

[Yu et al., 2015] Yu, F., Seff, A., Zhang, Y., Song, S., Funkhouser, T., and Xiao, J. (2015). Lsun: Construction of a large-scale image dataset using deep learning with humans in the loop. *arXiv preprint arXiv:1506.03365*.

[Zhang et al., 2021] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2021). Dive into deep learning. *arXiv preprint arXiv:2106.11342*.

[Zhang et al., 2019] Zhang, X., Karaman, S., and Chang, S.-F. (2019). Detecting and simulating artifacts in gan fake images. In *2019 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6.

[Zhou et al., 2016] Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A. (2016). Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929.