Machine Learning

Homework Assignment 4

Name: Ahmadreza Farvardin
Student ID: 610301221

June 22, 2025

Problems

Problem 1

Part (a)

Explanation of the Vanishing Gradient Problem and Why ReLU Avoids It

Vanishing Gradient Problem

The vanishing gradient problem occurs during the training of deep neural networks when the gradients of the loss function with respect to the weights become extremely small as they are backpropagated through the layers. This causes the weights in earlier layers to update very slowly or not at all, effectively stalling the learning process for those layers.

Why Sigmoid and Tanh Suffer from Vanishing Gradients

1. Sigmoid Function:

- Equation: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Derivative: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- The maximum value of the derivative is **0.25** (when $\sigma(x) = 0.5$). Since the derivative is always ≤ 0.25, during backpropagation, repeated multiplication of small gradients leads to exponentially shrinking updates for deeper layers.

2. Tanh (Hyperbolic Tangent) Function:

- Equation: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- Derivative: $\tanh'(x) = 1 - \tanh^2(x)$
- The maximum value of the derivative is **1.0** (when $x = 0$), but for inputs far from zero, the derivative quickly approaches **0**. Thus, in deep networks, gradients can still vanish when propagated through multiple layers.

Why ReLU Avoids the Vanishing Gradient Problem

ReLU (Rectified Linear Unit) Function:

- Equation: $\text{ReLU}(x) = \max(0, x)$

- Derivative:
  - $\text{ReLU}'(x) = 1$ if $x > 0$
  - $\text{ReLU}'(x) = 0$ if $x \leq 0$

Key Advantages:

1. **No Gradient Saturation for Positive Inputs:**
   - For $x > 0$, the gradient is always **1**, meaning there is no decay during backpropagation. This helps maintain stable gradient flow in deep networks.
2. **Avoids Multiplicative Gradient Shrinkage:**
   - Unlike sigmoid and tanh, ReLU does not force gradients to be multiplied by small values (≤ 1) at every layer. Thus, gradients can propagate effectively without vanishing.
3. **Computationally Efficient:**
   - ReLU is simple to compute compared to sigmoid and tanh, making training faster.

Drawback of ReLU (Dead Neurons)

If a neuron's output is consistently negative (e.g., due to large negative bias), its gradient will be **0** (since $\text{ReLU}'(x) = 0$ for $x \leq 0$), causing the neuron to never activate again ("dying ReLU" problem). Variants like **Leaky ReLU** or **Parametric ReLU (PReLU)** help mitigate this.

Part (b)

Choosing Between ReLU and Sigmoid for Binary Classification

The choice between ReLU and sigmoid activation functions for binary classification depends on where in the network they are used. For the output layer, sigmoid is the correct choice, while ReLU should be used in hidden layers.

Why Sigmoid for Output Layer

- **Probability Output:** Sigmoid outputs values between 0 and 1, providing direct probability interpretation for binary classification
- **Loss Function Compatibility:** Works naturally with binary cross-entropy loss
- **Bounded Output:** Ensures predictions stay within meaningful probability range

Why Not ReLU for Output Layer

- **Unbounded Output:** Produces values from [0, ∞), which cannot represent probabilities
- **Loss Function Incompatibility:** Not suitable for binary cross-entropy loss
- **No Probabilistic Interpretation:** Cannot interpret output as class probabilities

Optimal Architecture

For binary classification networks:

- **Output Layer:** Sigmoid activation
- **Hidden Layers:** ReLU activation

Activation	Best For	Range	BCE Compatible
Sigmoid	Output Layer	[0,1]	Yes
ReLU	Hidden Layers	[0,∞)	No

Table 1.1: Activation Function Comparison for Binary Classification

Part (c)

Difference Between ReLU and Leaky ReLU

Mathematical Definitions

$$\text{ReLU:} \quad f(x) = \max(0, x)$$

$$\text{Leaky ReLU:} \quad f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

where α is typically 0.01

Key Differences

When to Use Leaky ReLU

- **Dead Neuron Prevention:** Allows small updates even for negative inputs
- **Deep Networks:** Maintains gradient flow better than ReLU
- **Negative Data:** Better handles meaningful negative values
- **Unstable Training:** Can improve convergence when ReLU fails

Feature	ReLU	Leaky ReLU
Negative Input Output	0	αx
Negative Input Gradient	0	α
Dead Neuron Risk	High	Low
Training Stability	Lower	Higher

Table 1.2: Comparison of ReLU and Leaky ReLU

When to Use ReLU

- Shallow networks
- Predominantly positive activations
- When computational efficiency is crucial

Notable Variants

- **Parametric ReLU (PReLU):** Learns α during training
- **Randomized Leaky ReLU (RReLU):** Uses random α in training

Part (d)

Impact of Weight and Bias Initialization on Neural Network Performance

Initial weight and bias values significantly impact neural network learning and performance under several key conditions:

1. Network Architecture Conditions

- **Deep Networks:**
  - Poor initialization causes vanishing/exploding gradients
  - Affects gradient flow through multiple layers
- **Wide Networks:**
  - Large number of neurons requires balanced initialization
  - Affects signal propagation across neurons

2. Activation Function Dependencies

- **Sigmoid/Tanh Networks:**
  - Sensitive to initialization due to saturation regions
  - Requires Xavier initialization: $W \sim \mathcal{N}(0, \sqrt{\frac{2}{n_{in}+n_{out}}})$
- **ReLU Networks:**
  - Risk of dead neurons with poor initialization
  - Requires He initialization: $W \sim \mathcal{N}(0, \sqrt{\frac{2}{n_{in}}})$

3. Training Dynamics Impact

- **Convergence Speed:**
  - Good initialization leads to faster training
  - Poor initialization may require more epochs
- **Final Performance:**
  - Affects quality of learned features
  - Influences model's generalization ability

Conclusion

Initialization is particularly crucial when:

1. Network is deep or wide
2. Using sensitive activation functions
3. Training without batch normalization
4. Optimizing for faster convergence
5. Aiming for better final performance

Proper initialization ensures stable training dynamics and better model performance, while poor initialization can lead to training failures or suboptimal results.

Part (e)

Impact of Hidden Layer Count on Neural Network Performance

The effect of adding hidden layers in a neural network (like MLP) depends on several key conditions:

Conditions Where Adding Layers May Improve Performance

- **Complex Data Patterns:**
  - When data has hierarchical features
  - For tasks requiring multiple levels of abstraction
- **Non-Linear Relationships:**
  - When simpler architectures cannot capture complex patterns
  - For problems requiring sophisticated feature transformations

Conditions Where Adding Layers May Decrease Performance

- **Simple Problems:**
  - Linear or near-linear relationships
  - Problems solvable with simpler architectures
- **Limited Data:**
  - Small datasets relative to model complexity
  - Risk of overfitting increases with depth

Conditions Where Adding Layers May Have No Impact

- **Universal Approximation:**
  - Single hidden layer may already be sufficient
  - Additional complexity doesn't add value
- **Bottleneck Architecture:**
  - When information flow is constrained by narrow layers
  - When earlier layers capture all relevant features

Problem 2

Part (a)

1D Convolution Calculation

Given:

- Input vector: [6, 3, 6, 2]
- Filter: [2, 2, 1]
- Stride: 1
- Activation: ReLU

Solution Steps

1. First Window: [6, 3, 6]

$$\begin{aligned}\text{Calculation} &= (6 \times 2) + (3 \times 2) + (6 \times 1) \\ &= 12 + 6 + 6 \\ &= 24 \\ \text{After ReLU} &= \max(0, 24) = 24\end{aligned}$$

2. Second Window: [3, 6, 2]

$$\begin{aligned}\text{Calculation} &= (3 \times 2) + (6 \times 2) + (2 \times 1) \\ &= 6 + 12 + 2 \\ &= 20 \\ \text{After ReLU} &= \max(0, 20) = 20\end{aligned}$$

Output Size Verification For input length 4 and filter length 3 with stride 1:

$$\text{Number of outputs} = 4 - 3 + 1 = 2$$

Final Result

The output after convolution and ReLU activation is [24, 20].

Note

Using 'valid' convolution (no padding), which results in output length shorter than input. ReLU activation preserved these values as they were positive.

Part (b)

Calculating Output Size for 1D Convolution with Padding

Given Parameters

- Input size (L): 16
- Filter size (K): 3
- Stride (S): 2
- Padding (P): 1 (per side)

Formula

The output size for 1D convolution is given by:

$$\text{Output Size} = \left\lfloor \frac{L + 2P - K}{S} \right\rfloor + 1$$

where $\lfloor \cdot \rfloor$ denotes the floor function.

Step-by-Step Solution

1. Calculate Total Padding

$$\text{Total padding} = 2P = 2 \times 1 = 2$$
$$\text{Padded input size} = L + 2P = 16 + 2 = 18$$

2. Apply Formula

$$\begin{aligned} \text{Output Size} &= \left\lfloor \frac{18 - 3}{2} \right\rfloor + 1 \\ &= \left\lfloor \frac{15}{2} \right\rfloor + 1 \\ &= \lfloor 7.5 \rfloor + 1 \\ &= 7 + 1 \\ &= 8 \end{aligned}$$

Verification

Valid filter positions (stride = 2):

- Position 0: [0, 1, 2]
- Position 2: [2, 3, 4]
- Position 4: [4, 5, 6]
- Position 6: [6, 7, 8]
- Position 8: [8, 9, 10]
- Position 10: [10, 11, 12]
- Position 12: [12, 13, 14]
- Position 14: [14, 15, 16]

Final Answer

The output size is $\boxed{8}$

Part (c)

Max Pooling Layer Weight Calculation

Given Parameters

- Input Image Size: 2048×1024
- Pooling Kernel Size: 3×3
- Stride (S): 2
- Padding (P): 7 pixels per side

Key Observation

Max pooling layers have **no learnable weights**. They perform a fixed operation of selecting the maximum value in each kernel window.

Explanation

- Unlike convolutional layers, max pooling:
  - Does not perform weighted combinations
  - Has no trainable parameters
  - Simply selects maximum value in each window

Output Size Calculation

Although not relevant for weight count, the output dimensions are:

$$\begin{aligned}\text{Output Height} &= \left\lfloor \frac{2048 + 14 - 3}{2} \right\rfloor + 1 \\ &= \left\lfloor \frac{2059}{2} \right\rfloor + 1 = 1030 \\ \text{Output Width} &= \left\lfloor \frac{1024 + 14 - 3}{2} \right\rfloor + 1 \\ &= \left\lfloor \frac{1035}{2} \right\rfloor + 1 = 518\end{aligned}$$

Final Answer

The number of learnable weights is $\boxed{0}$

Note

While max pooling affects the spatial dimensions of the feature maps, it does so without introducing any trainable parameters, making it a parameter-free layer in neural networks.

Part (d)

Calculating Required Padding for Desired Output Size

Given Parameters

- Input Image Size: 256×256
- Filter Size: 32×32
- Stride (S): 2
- Desired Output Size: 128×128

Method

Using the convolution output size formula and solving for padding P:

$$\text{Output Size} = \left\lfloor \frac{H + 2P - K}{S} \right\rfloor + 1$$

1. Set up equation

$$\begin{aligned} 128 &= \left\lfloor \frac{256 + 2P - 32}{2} \right\rfloor + 1 \\ 128 &= \left\lfloor \frac{224 + 2P}{2} \right\rfloor + 1 \\ 128 &= \lfloor 112 + P \rfloor + 1 \end{aligned}$$

2. Solve for P Since P must be an integer, we can remove the floor function:

$$\begin{aligned} 128 &= 112 + P + 1 \\ P &= 128 - 113 \\ P &= 15 \end{aligned}$$

3. Verification Plugging P = 15 back into original formula:

$$\begin{aligned} \text{Output Size} &= \left\lfloor \frac{256 + 30 - 32}{2} \right\rfloor + 1 \\ &= \left\lfloor \frac{254}{2} \right\rfloor + 1 \\ &= 127 + 1 \\ &= 128 \end{aligned}$$

Final Answer

The required padding is $\boxed{15}$ pixels per side.

Part (e)

Calculating Parameters in 1D Convolution Layer

Given Parameters

- Number of filters (N): 12
- Filter size (K): 3
- No bias used

Calculation

For 1D convolution without bias:

$$\begin{aligned} \text{Total Parameters} &= N \times K \\ &= 12 \times 3 \\ &= 36 \end{aligned}$$

Note

Parameter count depends only on:

- Number of filters
- Filter size

It does not depend on input size, stride, or padding.

Final Answer

The total number of parameters is $\boxed{36}$

Part (f)

Calculating 2D Convolution Output Size

Given Parameters

- Input Size: 48×48
- Filter Size: 5×5
- Stride (S): 2
- Padding (P): 0

Calculation

Using formula: $\left\lfloor \frac{H+2P-K}{S} \right\rfloor + 1$

$$\begin{aligned} \text{Output Size} &= \left\lfloor \frac{48+0-5}{2} \right\rfloor + 1 \\ &= \left\lfloor \frac{43}{2} \right\rfloor + 1 \\ &= 21 + 1 \\ &= 22 \end{aligned}$$

Final Answer

The output size is $\boxed{22 \times 22}$

Part (g)

Calculating Required Padding for Output Size

Given Parameters

- Input Size: 42×42
- Filter Size: 7×7
- Stride (S): 2
- Desired Output: 28×28

Calculation

Using formula: $28 = \left\lfloor \frac{42+2P-7}{2} \right\rfloor + 1$

$$
\begin{aligned}
28 &= \left\lfloor \frac{35+2P}{2} \right\rfloor + 1 \\
27 &= \lfloor 17.5 \rfloor + P \\
9.5 &\leq +P < 10.5 \\
P &= 10 \text{ (P is an integer value)}
\end{aligned}
$$

Verification

$$
\begin{aligned}
\text{Output Size} &= \left\lfloor \frac{42+20-7}{2} \right\rfloor + 1 \\
&= \left\lfloor \frac{55}{2} \right\rfloor + 1 = 28
\end{aligned}
$$

Final Answer

The required padding is $\boxed{10}$ pixels per side

Part (h)

Can Pooling Layer Output be Larger than Input?

Yes, the output size can be larger than the input size in a pooling layer, though this is uncommon in practice.

Possible Through Two Methods

1. **Fractionally Strided Pooling:**
   - Using stride < 1 (e.g., 0.5)
   - Example: 2×2 pooling with stride 0.5 on 2×2 input $\rightarrow$ 4×4 output
2. **Large Padding:**
   - When padding exceeds kernel size
   - Example: 3×3 pooling with padding 4 on 5×5 input:

$$\text{Output size} = \left\lfloor \frac{5 + 8 - 3}{1} \right\rfloor + 1 = 11$$

Note

While mathematically possible, pooling layers in CNNs typically reduce spatial dimensions. Upsampling is more commonly achieved through transposed convolutions or interpolation.

Problem 3

Part (a)

First PC and first LD

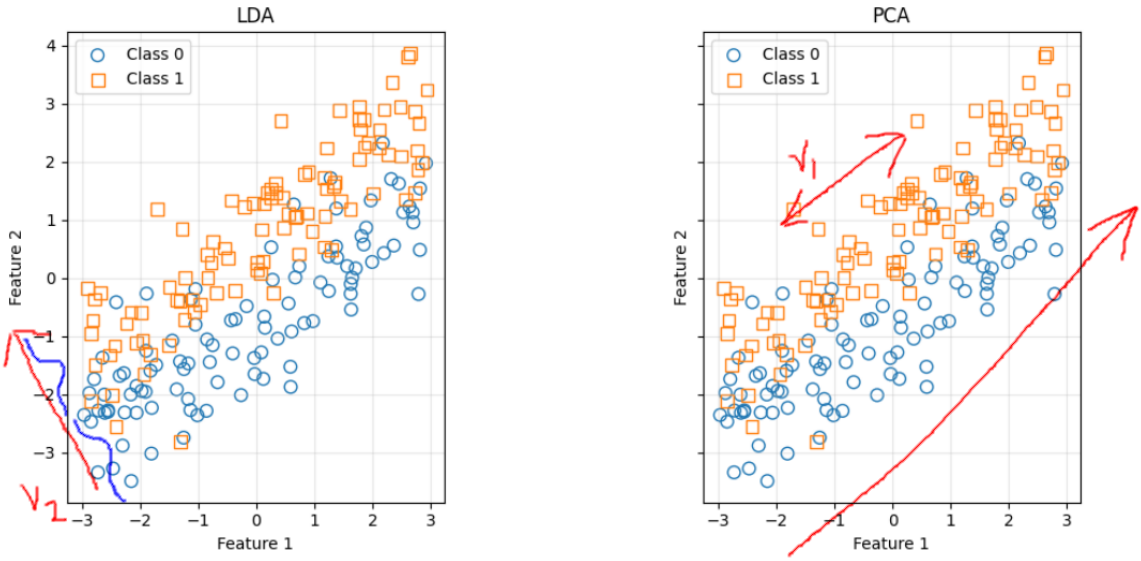

Figure 3.1: First LD and first PC

Note

The first PC is in direction of the eigenvector corresponded to the maximum eigenvalue, which denotes the biggest variance of data points regardless of the lables (v_1). While the first LD is in direction of the vector which by projecting the datapoints on it, it has maximum mean distance of classes and also the lowest variance in each data points of the class (v_2).

Part (b)

Computing LDA Components

Given Data

Three classes with points:

- Class 1: $\binom{-1}{2}, \binom{1}{2}$
- Class 2: $\binom{-3}{0}, \binom{-3}{-2}$
- Class 3: $\binom{3}{0}, \binom{3}{-2}$

Solution Steps

1. **Class Means:**

$$m_1 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}, \quad m_2 = \begin{pmatrix} -3 \\ -1 \end{pmatrix}, \quad m_3 = \begin{pmatrix} 3 \\ -1 \end{pmatrix}, \quad m_t = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

2. **Within-Class Scatter Matrix:**

$$S_w = S_1 + S_2 + S_3 = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}$$

3. **Between-Class Scatter Matrix:**

$$S_B = \begin{bmatrix} 36 & 0 \\ 0 & 12 \end{bmatrix}$$

4. **Eigenvalue Problem:**

$$S_w^{-1} S_B = \begin{bmatrix} 18 & 0 \\ 0 & 3 \end{bmatrix}$$

   Eigenvalues: $\lambda_1 = 18, \lambda_2 = 3$

Final Answer

The first Linear Discriminant (LD) is in the direction of $\boxed{W = \begin{pmatrix} 1 \\ 0 \end{pmatrix}}$, corresponding to the largest eigenvalue $\lambda_1 = 18$. This aligns with the horizontal separation of classes in the data.

Problem 4

Part (a)

Kernel PCA Computational Challenge

The main computational challenge in Kernel PCA arises when working in high-dimensional feature spaces.

Problem

When applying PCA in a feature space mapped by $\phi : \mathbb{R}^D \to \mathbb{R}^M$ where $M \gg D$, the covariance matrix becomes:

$$S = \frac{1}{N} \sum_{i=1}^{N} \phi(x_i)\phi(x_i)^T$$

This is computationally intractable because:

- S is $M \times M$ dimensional
- M can be extremely large or infinite
- Direct eigenvalue decomposition becomes infeasible

The kernel trick resolves this by:

- Working with kernel matrix K where $K_{ij} = \phi(x_i)^T\phi(x_j)$
- Solving eigenvalue problem for K $(N \times N)$ instead of S $(M \times M)$
- Avoiding explicit computation of $\phi(x_i)$

This makes the computation tractable when $N \ll M$.

Part (b)

Proof: Principal Components as Linear Combinations in Feature Space

In Kernel PCA, we prove that the j-th principal component v_j in the high-dimensional feature space can be expressed as a linear combination of the mapped data points $\phi(x_i)$.

1. Eigenvalue Problem in Feature Space

The covariance matrix in feature space is:

$$S = \frac{1}{N} \sum_{i=1}^{N} \phi(x_i)\phi(x_i)^T$$

The j-th principal component v_j satisfies:

$$Sv_j = \lambda_j v_j$$

2. Express v_j as a Linear Combination

We claim that v_j can be written as:

$$v_j = \sum_{i=1}^{N} w_{ij}\phi(x_i)$$

where $w_j = (w_{1j}, \ldots, w_{Nj})^T$ is an N-dimensional weight vector.

Proof:

1. Substitute v_j into the eigenvalue equation:

$$Sv_j = \left(\frac{1}{N} \sum_{k=1}^{N} \phi(x_k)\phi(x_k)^T \right) \left(\sum_{i=1}^{N} w_{ij}\phi(x_i) \right) = \lambda_j \sum_{i=1}^{N} w_{ij}\phi(x_i)$$

2. Expand the left-hand side:

$$\frac{1}{N} \sum_{k=1}^{N} \sum_{i=1}^{N} w_{ij}\phi(x_k) \underbrace{\phi(x_k)^T\phi(x_i)}_{K_{ki}} = \frac{1}{N} \sum_{k=1}^{N} \phi(x_k) \left(\sum_{i=1}^{N} w_{ij}K_{ki} \right)$$

3. Equate to the right-hand side:

$$\frac{1}{N} \sum_{k=1}^{N} \phi(x_k) \left(\sum_{i=1}^{N} w_{ij}K_{ki} \right) = \lambda_j \sum_{i=1}^{N} w_{ij}\phi(x_i)$$

4. Compare coefficients of $\phi(x_k)$:

$$\frac{1}{N} \sum_{i=1}^{N} w_{ij}K_{ki} = \lambda_j w_{kj} \quad \forall k$$

This simplifies to the kernel eigenvalue problem:

$$Kw_j = N\lambda_j w_j$$

where K is the kernel matrix ($K_{ki} = \phi(x_k)^T\phi(x_i)$).

3. Note

- The eigenvector w_j is in $\mathbb{R}^N$, not $\mathbb{R}^M$ (where $M \gg N$), making computation tractable.
- The projection of a new point x onto v_j is:

$$v_j^T \phi(x) = \sum_{i=1}^{N} w_{ij} K(x_i, x)$$

  No explicit $\phi(x)$ is needed—just the kernel K.

Why Non-Zero Eigenvalues Matter

Only eigenvectors with $\lambda_j > 0$ are meaningful:

- Zero eigenvalues correspond to directions with no variance
- Non-zero eigenvalues capture the principal components that retain data structure

Part (c)

Deriving the Kernel PCA Eigenvalue Equation

We prove that for a kernel matrix K, the weight vector w_j (corresponding to non-zero eigenvalue λ_j) satisfies:

$$Kw_j = N\lambda_j w_j$$

Step 1: Setup in Feature Space

The covariance matrix in feature space $\mathbb{R}^M$ is:

$$S = \frac{1}{N} \sum_{i=1}^{N} \phi(x_i)\phi(x_i)^T$$

The j-th principal component v_j satisfies:

$$Sv_j = \lambda_j v_j$$

From previous part, v_j is a linear combination:

$$v_j = \sum_{i=1}^{N} w_{ij} \phi(x_i) = \Phi w_j$$

where $\Phi = [\phi(x_1), \ldots, \phi(x_N)]$ and $w_j = (w_{1j}, \ldots, w_{Nj})^T$.

Step 2: Substitution

Substitute $v_j = \Phi w_j$ into $Sv_j = \lambda_j v_j$:

$$\left(\frac{1}{N}\Phi\Phi^T\right)\Phi w_j = \lambda_j \Phi w_j$$

Simplify left-hand side:

$$\frac{1}{N}\Phi \underbrace{(\Phi^T\Phi)}_{K} w_j = \frac{1}{N}\Phi K w_j$$

Thus:

$$\frac{1}{N}\Phi K w_j = \lambda_j \Phi w_j$$

Step 3: Elimination of Φ

Multiply both sides by Φ^T:

$$\frac{1}{N}\Phi^T\Phi K w_j = \lambda_j \Phi^T\Phi w_j$$

Using $\Phi^T\Phi = K$:

$$\frac{1}{N}K^2 w_j = \lambda_j K w_j$$

For invertible K, multiply by K^{-1}:

$$\frac{1}{N}K w_j = \lambda_j w_j$$

Finally:

$$K w_j = N\lambda_j w_j$$

Key Points

1. **Kernel Matrix Duality:** Problem reduces from $M \times M$ to $N \times N$
2. **Non-Zero Eigenvalues:** Only $\lambda_j > 0$ are meaningful
3. **Interpretation:** w_j defines v_j as optimal linear combination

Part (d)

Projection in Kernel PCA Without Explicit Principal Components

Key Idea

We compute the projection of a mapped point $\phi(x)$ onto the j-th principal component v_j:

$$\phi(x)^T v_j$$

without explicitly calculating v_j or $\phi(x)$.

Step 1: Express v_j as Linear Combination

From earlier:

$$v_j = \sum_{i=1}^{N} w_{ij}\phi(x_i) = \Phi w_j$$

where:

- $\Phi = [\phi(x_1), \ldots, \phi(x_N)]$ (mapped data matrix)
- $w_j = (w_{1j}, \ldots, w_{Nj})^T$ (weight vector from $Kw_j = N\lambda_j w_j$)

Step 2: Compute Projection

Project new point $\phi(x)$ onto v_j:

$$\begin{aligned}\phi(x)^T v_j &= \phi(x)^T \left(\sum_{i=1}^{N} w_{ij}\phi(x_i) \right) \\ &= \sum_{i=1}^{N} w_{ij}\phi(x)^T\phi(x_i) \\ &= \sum_{i=1}^{N} w_{ij}k(x, x_i)\end{aligned}$$

using kernel trick $\phi(x)^T\phi(x_i) = k(x, x_i)$

Step 3: Vectorized Form

Define kernel vector for x:

$$k_x = \begin{bmatrix} k(x, x_1) \\ k(x, x_2) \\ \vdots \\ k(x, x_N) \end{bmatrix}$$

Then projection is:

$$\phi(x)^T v_j = w_j^T k_x$$