AI and Machine Learning, Homework9

Author: Ying Yiwen
Number: 12210159

Contents

1 Introduction **2**
1.1 PCA . . . 2
1.2 Autoencoder . . . 2
1.3 Dataset . . . 2

2 PCA **2**
2.1 Principle . . . 2
2.2 Code Implementation . . . 3
2.3 Result . . . 4

3 Linear Autoencoder **4**
3.1 Principle . . . 4
3.2 Code Implementation . . . 6
3.3 Result . . . 6

4 Nonlinear Autoencoder **7**
4.1 Principle . . . 7
4.2 Code Implementation . . . 7
4.3 Result . . . 9

5 Comparison and Analysis **10**
5.1 Reconstruction Errors . . . 10
5.2 Discussion . . . 10
5.3 Conclusion . . . 10

1 Introduction

1.1 PCA

PCA (Principal Component Analysis) is a classical linear dimensionality reduction method, which aims to retain the most important features in the data by linearly transforming the data, allowing the low-dimensional representation to maximize the retention of the variance of the original data.The core idea of PCA is to find the eigenvalues and eigenvectors by calculating the covariance matrix of the data.The eigenvalues represent the magnitude of the data's variance in the direction, and the eigenvectors determines the direction of data variation. By selecting the first k eigenvectors with the largest eigenvalues, the data is projected into a new low-dimensional space.PCA is suitable for dealing with data with linear relationships, which can effectively reduce the dimensionality of the data and reduce the burden of computation, but it is weak in the expression of nonlinear data or complex relationships.

1.2 Autoencoder

Autoencoder is a deep learning model widely used in unsupervised learning, especially in tasks such as dimensionality reduction, feature extraction and data reconstruction. An auto-encoder consists of two parts: an encoder that compresses the input data into a low-dimensional potential space and a decoder that reconstructs the original data from the low-dimensional representation. By minimizing the error between the input data and the reconstructed data, the auto-encoder learns valid features from the data. Unlike PCA, self-encoders are able to capture the nonlinear relationships of the data and are suitable for dealing with complex data structures and patterns. Self-encoders are also widely used in areas such as denoising, anomaly detection and generative modeling. Despite its ability to capture complex patterns, the training cost is high and the model is less interpretable due to the large amount of data and computation required.

1.3 Dataset

The Wine Dataset (WD) is a classical multi-class categorical dataset derived from the UCI Machine Learning Repository containing 178 samples, each describing 13 chemical characteristics of a wine, such as alcohol content, malic acid, ash, magnesium content, total phenolics, flavonoids, anthocyanins, and color intensity. These characteristics were extracted from different wines by chemical analysis and were intended to be used to predict the type of wine. The dataset has three categories (usually labeled 1, 2, and 3), each corresponding to a wine type. The dataset is suitable for supervised learning, especially for applications in multi-class classification tasks, and can help researchers to evaluate and compare the performance of various machine learning algorithms, especially in data preprocessing, feature selection, and classification model construction. The wine dataset has no missing values and there may be some correlation between the features, making it a typical dataset for experiments and model training.

2 PCA

2.1 Principle

Principal Component Analysis (PCA) is a technique commonly used for dimensionality reduction. It linearly transforms the data from a high-dimensional space to a lower-dimensional space, preserving the most important variations in the data. The core idea of PCA is to find the directions in which the data has the largest variance, i.e., the directions where the data spreads the most.

Data Centering
The first step in PCA is centering the data by subtracting the mean of each feature from the data, so that the dataset has a mean of zero.
Let the original data matrix be X, where each row represents a sample and each column represents a feature. The centered data matrix X' is given by:

$$X' = X - \mu \tag{1}$$

where μ is the mean of each feature in the dataset.

Covariance Matrix Calculation
Next, we calculate the covariance matrix of the data. The covariance matrix reflects the relationships between the features and the distribution of the data. The covariance matrix Σ is computed as:

$$\Sigma = \frac{1}{n-1} X'^T X' \tag{2}$$

where X' is the centered data matrix and n is the number of samples.

Eigenvalues and Eigenvectors
We then perform eigenvalue decomposition on the covariance matrix Σ to obtain the eigenvalues and eigenvectors.

$$\Sigma v = \lambda v \tag{3}$$

where v is the eigenvector of the covariance matrix and λ is the corresponding eigenvalue. The eigenvalue represents the variance along the direction of the corresponding eigenvector, and the eigenvector represents the direction of the data's distribution.

Selecting Principal Components
We select the eigenvectors corresponding to the largest eigenvalues as the principal components. Suppose we choose the top k eigenvectors, these eigenvectors form the matrix V_k:

$$V_k = \begin{bmatrix} v_1 & v_2 & \cdots & v_k \end{bmatrix} \tag{4}$$

where $v_1, v_2, \ldots, v_k$ are the eigenvectors corresponding to the largest eigenvalues, sorted in descending order.

Data Projection
Finally, we project the original data onto the selected principal components. The projection of the data onto the lower-dimensional space is given by:

$$X_{\text{new}} = X' V_k \tag{5}$$

where X_{new} is the data in the reduced dimension and V_k is the matrix of selected principal components.

2.2 Code Implementation

```python
def fit(self, X):
  # Compute the covariance matrix of the data (X.T is the transpose of X)
  cov_matrix = np.cov(X.T)

  # Compute eigenvalues and eigenvectors of the covariance matrix
  eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

  # Create pairs of eigenvalue and corresponding eigenvector, and sort by
      eigenvalue in descending order
  eigen_pairs = [(np.abs(eigenvalues[i]), eigenvectors[:, i]) for i in range(len(
      eigenvalues))]
  eigen_pairs.sort(key=lambda k: k[0], reverse=True)

  # Select the top 'n_components' eigenvectors based on sorted eigenvalues
  self.eigenvalues = eigenvalues
  self.eigenvectors = eigenvectors
```

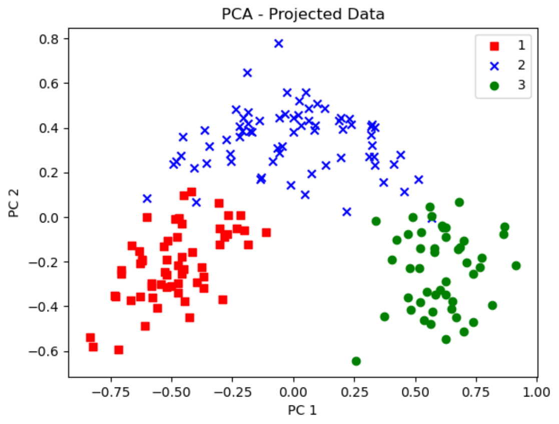
```
    self.projection_matrix = np.hstack([eigen_pairs[i][1][:, np.newaxis] for i in
        range(self.n_components)])
```

The `fit` method computes the covariance matrix of the input data, then performs eigenvalue decomposition to obtain the eigenvalues and eigenvectors. The eigenvectors corresponding to the largest eigenvalues are selected to form the `projection_matrix`, which will be used to project the data into a lower-dimensional space.

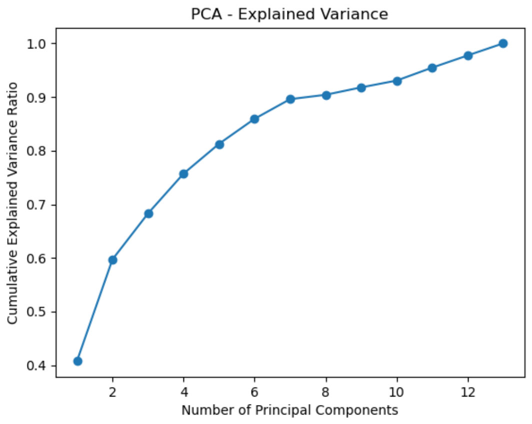
```
def transform(self, X):
  # Project the data onto the principal components
  return X.dot(self.projection_matrix)
```

The `transform` method projects the input data `X` onto the principal components by multiplying it with the `projection_matrix`, which reduces the dimensionality of the data.

2.3 Result



(a) PCA Result



(b) Explained Variance

Fig. 1: Principle Component Analysis

Reconstruction Error: 0.016641162609763827

From the picture, it can be seen that the downscaling of the two principal components has been able to basically differentiate the data and the reconstruction error has been very low and within the acceptable range. And comparing the different principal components, it is found that about 7 principal components are able to cover the vast majority of the information when they are used.

3 Linear Autoencoder

3.1 Principle

Linear Autoencoder is a self-supervised learning method that aims to compress data and learn features by learning a low-dimensional linear representation of the data. It consists of two main parts: the encoder and the decoder. The encoder maps the input data to a lower-dimensional latent space, and the decoder reconstructs the data from the latent space back to the original data space. The overall goal of the model is to learn the key features of the data while compressing it by minimizing the reconstruction error.

The **encoder**'s goal is to map the input data $\mathbf{X}$ to a lower-dimensional latent space $\mathbf{Z}$ through a linear transformation. The mathematical expression of the encoder is:

$$\mathbf{Z} = f(\mathbf{X}) = \mathbf{X}\mathbf{W_1} + \mathbf{b_1} \tag{6}$$

- $\mathbf{X} \in \mathbf{R}^{m\times n}$ is the input data matrix, where m is the number of samples and n is the number of features.
- $\mathbf{W_1} \in \mathbf{R}^{n\times k}$ is the encoder's weight matrix, with k being the dimension of the encoded representation (the latent space dimension).
- $\mathbf{b_1} \in \mathbf{R}^{1\times k}$ is the bias term.
- $\mathbf{Z} \in \mathbf{R}^{m\times k}$ is the encoded data representation, where the dimension has been reduced from n to k.

The encoder compresses the original data into a lower-dimensional representation $\mathbf{Z}$, which captures the main features of the data.

The **decoder**'s task is to map the encoded data $\mathbf{Z}$ back to the original data space. The mathematical expression of the decoder is:

$$\hat{\mathbf{X}} = g(\mathbf{Z}) = \mathbf{Z}\mathbf{W_2} + \mathbf{b_2} \tag{7}$$

- $\mathbf{W_2} \in \mathbf{R}^{k\times n}$ is the decoder's weight matrix.
- $\mathbf{b_2} \in \mathbf{R}^{1\times n}$ is the bias term.
- $\hat{\mathbf{X}} \in \mathbf{R}^{m\times n}$ is the reconstructed input data.

The decoder projects the encoded data $\mathbf{Z}$ back to the original space in an attempt to reconstruct the input data $\mathbf{X}$, recovering the key information.

The **loss function** for training the autoencoder is designed to minimize the difference between the input data and the reconstructed data. A common loss function is the Mean Squared Error (MSE):

$$L = \frac{1}{m}\sum_{i=1}^{m}\left\|\mathbf{X}_i - \hat{\mathbf{X}}_i\right\|_2^2 \tag{8}$$

- L is the loss function, representing the average reconstruction error.
- $\mathbf{X}_i$ is the i-th sample's original input data, and $\hat{\mathbf{X}}_i$ is the i-th sample's reconstructed data.
- $\|\cdot\|_2^2$ represents the squared Euclidean distance, or ℓ_2 norm squared.

The objective of the loss function is to optimize the weights and biases so that the reconstructed data $\hat{\mathbf{X}}$ is as close as possible to the original data $\mathbf{X}$, thereby enabling the encoder to learn a useful low-dimensional representation of the data.

The **training process** involves **backpropagation** to minimize the reconstruction error and update the weights and biases. The core idea of backpropagation is to compute the gradient of the loss function with respect to the parameters $\mathbf{W_1}$, $\mathbf{b_1}$, $\mathbf{W_2}$, and $\mathbf{b_2}$, and update them using gradient descent. The gradients are computed as follows:

- Gradient of the encoder weight $\mathbf{W_1}$:

$$\frac{\partial L}{\partial \mathbf{W_1}} = \frac{1}{m}\mathbf{X}^T\left(\mathbf{Z} - \mathbf{X}\right)\mathbf{W_2}^T \tag{9}$$

- Gradient of the decoder weight $\mathbf{W_2}$:

$$\frac{\partial L}{\partial \mathbf{W_2}} = \frac{1}{m}\mathbf{Z}^T\left(\mathbf{Z} - \mathbf{X}\right) \tag{10}$$

- Gradients of the bias terms:

$$\frac{\partial L}{\partial \mathbf{b_1}} = \frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{Z}_i - \mathbf{X}_i\right) \tag{11}$$

$$\frac{\partial L}{\partial \mathbf{b_2}} = \frac{1}{m}\sum_{i=1}^{m}(\mathbf{Z}_i - \mathbf{X}_i) \tag{12}$$

Through iterative updates, the model learns a low-dimensional representation that retains the essential features of the original data.

3.2 Code Implementation

```
def encode(self, X):
  # Encode the input data X by applying the first weight matrix and bias
  return np.dot(X, self.W1) + self.b1
```

The `encode` method applies a linear transformation to the input data `X` by multiplying it with the weight matrix `W1` and adding the bias `b1`, producing the encoded representation.

```
def decode(self, Z):
  # Decode the encoded data Z by applying the second weight matrix and bias
  return np.dot(Z, self.W2) + self.b2
```

The `decode` method applies a linear transformation to the encoded data `Z` by multiplying it with the weight matrix `W2` and adding the bias `b2`, reconstructing the original input data.

```
def fit(self, X):
  # Train the autoencoder using gradient descent
  for epoch in range(self.epochs):
  encoded = self.encode(X)   # Get encoded data
  decoded = self.decode(encoded)  # Reconstruct the input
  loss = np.mean((X - decoded) ** 2)  # Calculate reconstruction loss
  self.losses.append(loss)  # Store loss value

  # Backpropagation (gradient descent)
  dL_ddecoded = 2 * (decoded - X) / X.shape[0]
  dL_dW2 = np.dot(encoded.T, dL_ddecoded)
  dL_db2 = np.sum(dL_ddecoded, axis=0, keepdims=True)
  dL_dencoded = np.dot(dL_ddecoded, self.W2.T)
  dL_dW1 = np.dot(X.T, dL_dencoded)
  dL_db1 = np.sum(dL_dencoded, axis=0, keepdims=True)

  # Update weights and biases
  self.W1 -= self.learning_rate * dL_dW1
  self.b1 -= self.learning_rate * dL_db1
  self.W2 -= self.learning_rate * dL_dW2
  self.b2 -= self.learning_rate * dL_db2
```

The `fit` method trains the autoencoder using gradient descent. In each epoch, it first encodes and decodes the input data, calculates the reconstruction loss, and then performs backpropagation to compute gradients. Finally, the weights and biases are updated using the gradients to minimize the loss over time.

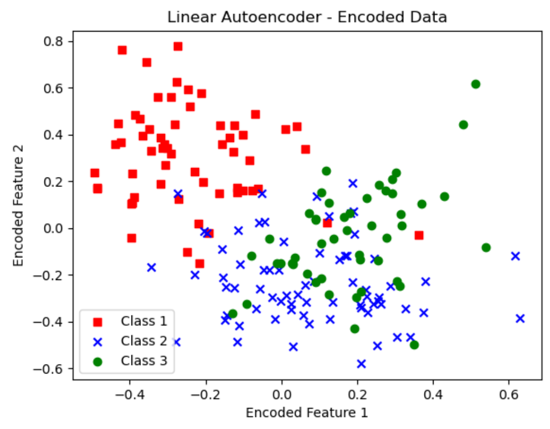
3.3 Result

The superparameters we use are:

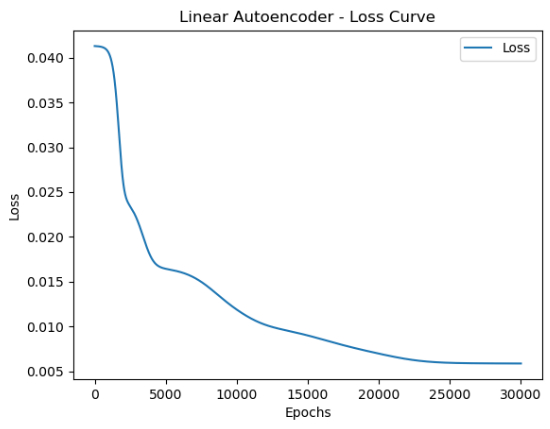
```
linear_autoencoder = LinearAutoencoder(input_dim=X.shape[1], encoding_dim=6,
    learning_rate=0.005, epochs=30000)
```

Reconstruction Error: 0.005856843049329647

As seen in the figure, Autoencoder is able to realize the dimensionality reduction of the data and gradually converges to the fit as the number of iterations increases.



(a) Linear Autoencoder Result



(b) Loss Trend

Fig. 2: Linear Autoencoder

4 Nonlinear Autoencoder

4.1 Principle

The ReLU (Rectified Linear Unit) activation function is widely used in neural networks, especially in deep learning models, due to its simplicity and effectiveness. It transforms the input by zeroing out negative values and keeping positive values unchanged.

$$\text{ReLU}(x) = \max(0, x) \tag{13}$$

If x is positive, it passes through unchanged; if x is negative, it becomes 0. This function is computationally efficient and helps avoid the vanishing gradient problem, making it ideal for deep networks.
During backpropagation, the derivative of ReLU is required to compute gradients for weight updates. The derivative of ReLU is:

$$\frac{\partial \text{ReLU}(x)}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases} \tag{14}$$

4.2 Code Implementation

The forward function computes the activations of the network layer by layer, from the input layer to the output layer.

```
def forward(self, X):
  A = X
  activations = [A]
  pre_activations = []

  for W, b in zip(self.weights[:-1], self.biases[:-1]):
    Z = np.dot(A, W) + b
    A = self.relu(Z)
    pre_activations.append(Z)
    activations.append(A)

  Z = np.dot(A, self.weights[-1]) + self.biases[-1]
  activations.append(Z)

  return activations
```

In the forward pass, the data X is passed through the network layer by layer. For each layer, the data undergoes a linear transformation followed by the ReLU activation function:

$$Z_i = A_{i-1}W_i + b_i \tag{15}$$

- A_{i-1} is the activation from the previous layer.
- W_i is the weight matrix for the current layer.
- b_i is the bias for the current layer.
- Z_i is the pre-activation (the result of the linear transformation).

After the linear transformation, the ReLU activation is applied:

$$A_i = \text{ReLU}(Z_i) \tag{16}$$

This process continues for all hidden layers, and the final output is the reconstruction of the input data $\hat{X}$, which is returned by the function.

The backward function computes the gradients of the loss function with respect to the weights and biases using backpropagation. These gradients are then used to update the weights to minimize the loss.

```
def backward(self, X, activations):
  m = X.shape[0]
  dZ = activations[-1] - X
  dW = np.dot(activations[-2].T, dZ) / m
  db = np.sum(dZ, axis=0, keepdims=True) / m

  self.weights[-1] -= self.learning_rate * dW
  self.biases[-1] -= self.learning_rate * db

  for i in range(len(self.hidden_dims)-1, -1, -1):
    dA = np.dot(dZ, self.weights[i+1].T)
    dZ = dA * self.relu_derivative(activations[i+1])
    dW = np.dot(activations[i].T, dZ) / m
    db = np.sum(dZ, axis=0, keepdims=True) / m

    self.weights[i] -= self.learning_rate * dW
    self.biases[i] -= self.learning_rate * db
```

In the backward pass, we first compute the gradient of the loss with respect to the output layer:

$$dZ_L = A_L - X \tag{17}$$

- A_L is the output of the final layer (the reconstruction of the input).
- X is the original input.

Then, we compute the gradients for the weights and biases at the output layer:

$$dW_L = \frac{1}{m} A_{L-1}^T \cdot dZ_L \tag{18}$$

$$db_L = \frac{1}{m}\sum dZ_L \tag{19}$$

- A_{L-1} is the activation of the previous layer.
- dZ_L is the gradient of the loss with respect to the output.

Next, we propagate the gradients backward through the hidden layers. For each hidden layer, the error dZ_i is computed by multiplying the error from the next layer dA with the derivative of the ReLU function:

$$dA = dZ_{i+1} \cdot W_{i+1}^T \tag{20}$$

$$dZ_i = dA \cdot \mathrm{ReLU}'(A_i) \tag{21}$$

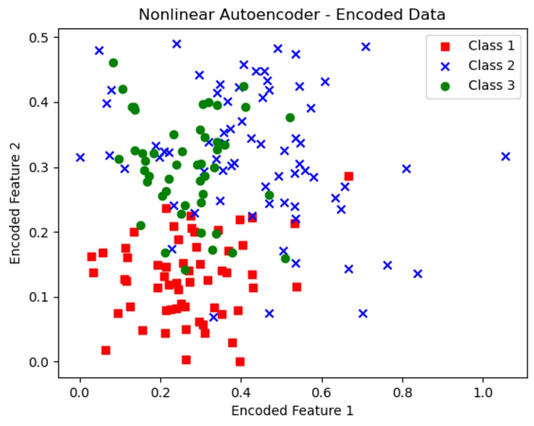
Finally, we compute the gradients for the weights and biases at each hidden layer and update them using the learning rate:

$$dW_i = \frac{1}{m} A_{i-1}^T \cdot dZ_i \tag{22}$$

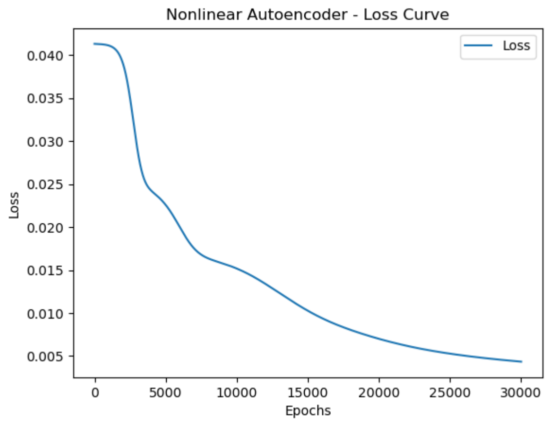
$$db_i = \frac{1}{m}\sum dZ_i \tag{23}$$

This process continues for all layers, updating the weights and biases to minimize the reconstruction error.

4.3 Result



(a) Nonlinear Autoencoder Result



(b) Loss Trend

Fig. 3: Nonlinear Autoencoder

The superparameters we use are:

```
nonlinear_autoencoder = NonlinearAutoencoder(input_dim=X.shape[1], hidden_dims
    =[16], output_dim=X.shape[1], learning_rate=0.007, epochs=30000)
```

Reconstruction Error: 0.004347043143699931

The nonlinear model can fit the nonlinear data better and does achieve better results in terms of reconstruction error.

5 Comparison and Analysis

5.1 Reconstruction Errors

- **PCA (Principal Component Analysis)**: Reconstruction Error: `0.0166`

  PCA performs dimensionality reduction by projecting the data onto a set of orthogonal axes (principal components). While it captures the largest variance in the data, it may struggle to capture non-linear relationships, especially if the data has a complex structure that cannot be well-represented in a linear subspace.

- **Linear Autoencoder**: Reconstruction Error: `0.0059`

  A linear autoencoder, similar to PCA, learns a lower-dimensional representation of the data. However, unlike PCA, it is trained to minimize reconstruction error, which may allow it to capture better representations in the latent space. It is still limited by the fact that the encoder and decoder are both linear, meaning it can only model linear relationships in the data.

- **Non-linear Autoencoder**: Reconstruction Error: `0.0043`

  Non-linear autoencoders, typically built using neural networks (such as multi-layer perceptrons), are more flexible in modeling complex relationships in the data. They can learn non-linear mappings between the input and the latent representation, which often results in better reconstruction accuracy compared to both PCA and linear autoencoders, especially for non-linear data.

5.2 Discussion

- **PCA** is a linear method, and it performs well when the data primarily lies on a linear subspace. However, if the data has complex, non-linear patterns, PCA might fail to capture these, leading to a higher reconstruction error.

- **Linear Autoencoder** performs better than PCA because it is trained explicitly to minimize the reconstruction error, and the model is optimized for this purpose. Still, since it is also a linear model, its performance is not much better than PCA when data is non-linear in nature.

- **Non-linear Autoencoder** outperforms both PCA and the linear autoencoder. It can capture complex, non-linear relationships in the data by using neural networks in the encoder and decoder. This flexibility allows it to achieve the best reconstruction accuracy.

5.3 Conclusion

The **non-linear autoencoder** provides the best reconstruction accuracy because it can model complex, non-linear relationships in the data, making it more suitable for datasets where such patterns exist. PCA, on the other hand, is limited by its linear nature, which is why it shows the highest reconstruction error. The linear autoencoder sits in between, offering better performance than PCA due to its training-based approach but still constrained by its linear structure.