Dimensionality Reduction for Classification

Shang Chen
School of Electrical and Electronic Engineering
Nanyang Technological University
Singapore, Singapore
CHEN1609@e.ntu.edu.sg

***Abstract*—Utilizing deep learning techniques for dimensionality reduction has proven to be highly effective in accurately extracting salient features, showing superior performance compared to traditional methods like PCA. This not only streamlines computations by reducing complexity but also significantly boosts the generalization potential of the associated classifiers, ensuring better performance across diverse datasets.**

***Index Terms*—deep learning, PCA, classification, dimensionality reduction**

I. Introduction

In the vast expanse of data-driven domains, the curse of dimensionality often stands as a challenge, potentially undermining the efficiency and effectiveness of the machine learning models deployed. High-dimensional datasets, which consist of a multitude of features, not only increase computational demands but can also lead to overfitting, where models become too specialized on training data, consequently failing to generalize effectively to unseen samples. Dimensionality reduction, the process of mapping data from a high-dimensional space to a lower-dimensional one, emerges as a cornerstone in addressing these challenges, allowing for the extraction of salient features that better represent the underlying structure and patterns in the data [1].

Traditional methods, such as Principal Component Analysis (PCA), have been extensively utilized for dimensionality reduction, yielding reasonable success across a myriad of applications. However, as we advance into an era dominated by deep learning, the potential of neural networks to capture complex, nonlinear relationships in data sets them apart. Deep learning-based dimensionality reduction techniques have ushered in an age of enhanced feature extraction, showcasing remarkable ability in isolating the most pertinent attributes of data.

This paper delves into the strides made in utilizing deep learning for dimensionality reduction, highlighting its superiority over conventional methods in enhancing classification performance. Through the lens of its benefits, such as computational streamlining and heightened generalization, we aim to provide a comprehensive understanding of its transformative impact on handling diverse datasets, thereby championing the cause of better, more accurate classifications.

II. Methodology

A. Principal Component Analysis

Principal Component Analysis [2], commonly abbreviated as PCA, is a statistical method that is widely utilized for dimensionality reduction. At its core, PCA aims to transform the original features of a dataset into a new set of orthogonal features known as principal components. These principal components encapsulate the maximum variance present in the data, with the first principal component capturing the most variance and each subsequent component accounting for a diminishing amount of the variance, all while being orthogonal to its predecessors.

Given a dataset X with n samples and m features, the primary objective of PCA is to determine the principal components that capture the most significant information about the data's variance.

Step 1: Standardize the dataset such that each feature has a mean of 0 and a standard deviation of 1.

$$Z = \frac{X - \mu}{\sigma}$$

Where: μ = mean of the dataset, σ = standard deviation of the dataset.

Step 2: Compute the covariance matrix of the standardized data Z.

$$C = \frac{1}{n-1} Z^T Z$$

Step 3: Calculate the eigenvalues and eigenvectors of the covariance matrix C. The eigenvectors represent the directions of the new feature space, and the eigenvalues determine their magnitude or importance.

Step 4: Sort the eigenvectors based on the descending order of their corresponding eigenvalues. The sorted eigenvectors form the columns of the transformation matrix P.

Step 5: Transform the original data using the top k eigenvectors to obtain the first k principal components of the data.

$$Y = ZP_k$$

Where: Y = Data projected into the new feature space, P_k = Matrix with columns as the top k eigenvectors.

PCA is particularly potent when there exists linear correlation among the features in a dataset. By transforming the data into a set of orthogonal principal components, it ensures that

only the most significant features, which account for most of the variance, are retained, thus efficiently reducing dimensions.

However, one should be aware that PCA, being a linear method, might not perform as effectively on data with intricate, nonlinear relationships. In such cases, nonlinear dimensionality reduction techniques, like those offered by deep learning models, might be more suitable.

In conclusion, while PCA provides an intuitive and computationally efficient means of dimensionality reduction, its effectiveness is contingent upon the inherent structure and relationships present within the dataset.

B. Linear Discriminant Analysis

Linear Discriminant Analysis (LDA) [3] is another popular technique for dimensionality reduction, especially in the context of supervised classification. Unlike PCA, which focuses on maximizing variance, LDA aims to find the feature subspace that optimizes class separability. Essentially, LDA looks for directions where classes are best separated, making it particularly effective when the end goal is classification.

Let's assume a dataset X with n samples, m features, and C distinct classes.

Step 1: Calculate the mean vectors for each class.

$$\mu_i = \frac{1}{n_i} \sum_{x \in C_i} x$$

Where n_i is the number of samples in class i and μ_i is the mean vector of class i.

Step 2: Compute the within-class scatter matrix S_W and the between-class scatter matrix S_B.

$$S_W = \sum_{i=1}^{C} \sum_{x \in C_i} (x - \mu_i)(x - \mu_i)^T$$

$$S_B = \sum_{i=1}^{C} n_i (\mu_i - \mu)(\mu_i - \mu)^T$$

Where μ is the overall mean of the dataset.

Step 3: Determine the eigenvectors and eigenvalues for the generalized eigenvalue problem:

$$S_W^{-1} S_B v = \lambda v$$

Step 4: Sort the eigenvectors based on the descending order of their corresponding eigenvalues. The top k eigenvectors form the new transformation matrix W.

Step 5: Transform the original data to obtain the LDA features:

$$Y = XW_k$$

Where Y = Data projected into the LDA feature space, W_k = Matrix with columns as the top k eigenvectors.

LDA, in its design, specifically targets the separability of classes, making it a powerful tool for classification problems. By maximizing the distance between the means of different classes and minimizing the scatter (variance) within each class, LDA ensures that classes are well-separated in the transformed space.

However, some limitations accompany LDA. Firstly, the number of features that can be extracted from LDA is at most $C - 1$, where C is the number of classes. Secondly, LDA assumes that classes have identical covariance matrices, which may not always hold true in real-world datasets. Lastly, just like PCA, LDA is linear in nature and may not capture complex, nonlinear class boundaries.

In summary, while LDA offers class-specific dimensionality reduction with enhanced classification performance, its inherent assumptions and linear characteristics necessitate a thorough understanding and careful application in diverse scenarios.

C. Deep Learning

In the realm of deep learning for dimensionality reduction, one might first think of autoencoders, which are particularly adept at capturing intricate data patterns and compressing them. However, in scenarios where labeled data is available, it becomes more beneficial to leverage architectures designed for supervised learning, which can utilize these labels for better feature extraction.

Given the presence of labels in our dataset, a more direct approach would be to employ deep convolutional neural networks (CNNs), like ResNet [4]. ResNet, or Residual Network, is renowned for its deep architecture and the ability to learn residual functions, preventing the vanishing gradient problem and enabling the training of much deeper networks.

By training a ResNet on a classification task, we can utilize the penultimate layer (just before the final classification layer) as a feature extractor, giving us a compressed representation of the original data. These features, being derived in the context of the task, are likely to be highly discriminative and beneficial for downstream tasks.

Utilizing ResNet for dimensionality reduction offers several distinct advantages. Leveraging labeled data, ResNet adeptly extracts features that are both discriminative and directly relevant to the classification task, ensuring that vital information is retained in the compressed representation. The deep architecture of ResNet provides the capability to capture a spectrum of features, from nuanced details to overarching patterns. Furthermore, the potential for transfer learning with ResNet is notable; when initialized with weights from models pre-trained on extensive datasets, such as ImageNet, it can be fine-tuned to specific tasks, often accelerating the training process and enhancing overall performance.

III. EXPERIMENTS

We conduct all our experiments on image dataset CIFAR-10. It is not a high-resolution dataset with every image is $32 \times 32 \times 3$.

A. Principal Component Analysis

We use Python library sklearn.decomposition to operation dimensionality reduction, and get the following result. It shows training and test accuracy in terms of the number of components to keep n.

TABLE I
PCA CLASSIFICATION ACCURACY

n	Training	Testing
10	32.20%	32.46%
100	36.52%	32.68%
1000	52.18%	23.22%

B. Linear Discriminant Analysis

When using linear discriminant analysis, unlike principal component analysis, you do not need to choose the number of components to retain. This is because LDA is a supervised method that aims to find the optimal feature space for the classification task. The maximum number of features in LDA is determined by the number of categories: specifically, the maximum number of features is the number of categories - 1. On our CIFAR-10 dataset, it is 9.

We also use Python library sklearn to operate the decomposition, getting training accuracy 50.46% and testing accuracy 37.09%.

C. Deep Learning

We utilize the ResNet architecture pretrained on ImageNet for feature extraction. To evaluate the decomposition efficacy, we consider two distinct model variants:

- Configure 1: Utilizes layer 1 of ResNet18, yielding an output dimension of 64.
- Configure 2: Utilizes layer 1 of ResNet50, yielding an output dimension of 256.

Conventionally, the terminal layer of ResNet is omitted, retaining the majority of its structure. However, for our specific application, the conventional dimensionality (2048) surpasses our requirements, especially given that our individual images have a total dimensionality of 3072. While such configurations might be advantageous for high-resolution datasets like ImageNet, our task necessitates truncating the model at an earlier stage to obtain a more compact feature representation.

The results are presented as follows:

TABLE II
DEEP LEARNING CLASSIFICATION ACCURACY

Model	Traing	Testing
Configure 1	69.22%	62.20%
Configure 2	79.61%	49.57%

IV. DISSCUSION

Deep learning-driven dimensionality reduction techniques have demonstrated profound efficacy. Their merits are readily discernible. While gauging the computational expense of such techniques can be intricate—given the variability in training costs across models and the option to leverage pretrained models—one undeniable advantage is the capacity for parallel computation on GPUs. In contrast, traditional dimensionality reduction approaches are predominantly CPU-bound.

Our experimentation, while straightforward, is predicated on the relatively simple CIFAR-10 dataset. This constraint inadvertently limits the full potential of our model. Yet, when we turn our gaze to more intricate datasets, such as ImageNet, where image resolutions commonly exceed $256 \times 256 \times 3$, the prowess of architectures like ResNet50—with an output size of 2048—becomes evident. In such scenarios, even more complex transformer-based models can truly manifest their capabilities.

ACKNOWLEDGMENT

Thanks Google for their free T4 GPU on colab.

REFERENCES

[1] R. Zebari, A. Abdulazeez, D. Zeebaree, D. Zebari, and J. Saeed, "A comprehensive review of dimensionality reduction techniques for feature selection and feature extraction," *Journal of Applied Science and Technology Trends*, vol. 1, no. 2, pp. 56–70, 2020.

[2] A. Maćkiewicz and W. Ratajczak, "Principal components analysis (pca)," *Computers & Geosciences*, vol. 19, no. 3, pp. 303–342, 1993.

[3] S. Balakrishnama and A. Ganapathiraju, "Linear discriminant analysis-a brief tutorial," *Institute for Signal and information Processing*, vol. 18, no. 1998, pp. 1–8, 1998.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

APPENDIX

```
# pca/lda.py
import numpy as np
import torch
import torchvision.transforms as transforms
from torchvision.datasets import CIFAR10
from sklearn.discriminant_analysis import
    LinearDiscriminantAnalysis as LDA
from scipy.spatial import distance
from tqdm import tqdm

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5),
        (0.5, 0.5, 0.5))
])

train_dataset = CIFAR10(root='./data', train=
    True, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=len(
    train_dataset), shuffle=False)

test_dataset = CIFAR10(root='./data', train=
    False, download=True, transform=transform)
```

```
test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=len(test_dataset)
    , shuffle=False)

train_images, train_labels = next(iter(
    train_loader))
train_images, train_labels = train_images.
    numpy(), train_labels.numpy()

test_images, test_labels = next(iter(
    test_loader))
test_images, test_labels = test_images.numpy()
    , test_labels.numpy()

train_images_flat = train_images.reshape(
    train_images.shape[0], -1)
test_images_flat = test_images.reshape(
    test_images.shape[0], -1)

lda = LDA()
train_lda = lda.fit_transform(
    train_images_flat, train_labels)
test_lda = lda.transform(test_images_flat)

class_means = {}
class_cov_matrices = {}
classes = np.unique(train_labels)

for c in classes:
    class_data = train_lda[train_labels == c]
    class_means[c] = np.mean(class_data, axis
        =0)
    class_cov_matrices[c] = np.cov(class_data,
         rowvar=False)

def mahalanobis_distance(x, mean, cov_matrix):
    return distance.mahalanobis(x, mean, np.
        linalg.inv(cov_matrix))

def classify_sample(x):
    distances = {}
    for c in classes:
        distances[c] = mahalanobis_distance(x,
             class_means[c],
            class_cov_matrices[c])
    return min(distances, key=distances.get)

train_pred = np.array([classify_sample(x) for
    x in tqdm(train_lda, desc="Classifying␣
    train␣samples")])
test_pred = np.array([classify_sample(x) for x
     in tqdm(test_lda, desc="Classifying␣test␣
    samples")])

train_accuracy = np.mean(train_pred ==
    train_labels)
test_accuracy = np.mean(test_pred ==
    test_labels)
print(f"Training␣accuracy␣using␣minimum␣
    Mahalanobis␣distance␣classifier␣after␣LDA:
    ␣{train_accuracy:.4f}")
print(f"Test␣accuracy␣using␣minimum␣
    Mahalanobis␣distance␣classifier␣after␣LDA:
    ␣{test_accuracy:.4f}")
```

```
# cnn.py
import torch
import torchvision
import torchvision.transforms as transforms
import numpy as np
from scipy.spatial.distance import mahalanobis
from tqdm import tqdm
import timm

device = torch.device("cuda" if torch.cuda.
    is_available() else "cpu")

transform = transforms.Compose([transforms.
    ToTensor()])
trainset = torchvision.datasets.CIFAR10(root='
    ./data', train=True, download=True,
    transform=transform)
testset = torchvision.datasets.CIFAR10(root='
    ./data', train=False, download=True,
    transform=transform)

trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=512, shuffle=True)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=512, shuffle=False)

model = timm.create_model('resnet18',
    pretrained=True).to(device)
feature_extractor = torch.nn.Sequential(*list(
    model.children())[:-4]).to(device)

feature_extractor.eval()

def extract_features(loader, model):
    features = []
    labels = []
    with torch.no_grad():
        for data in tqdm(loader, desc="
            Extracting␣features"):
            inputs, label = data
            inputs = inputs.to(device)
            feature = model(inputs)
            feature = torch.nn.functional.
                adaptive_avg_pool2d(feature,
                (1, 1))
            feature = feature.reshape(feature.
                size(0), -1)
            features.append(feature.cpu())
            labels.append(label)
    features = torch.cat(features, 0)
    labels = torch.cat(labels, 0)
    return features, labels

train_features, train_labels =
    extract_features(trainloader,
    feature_extractor)
test_features, test_labels = extract_features(
    testloader, feature_extractor)

class_means = {}
class_inv_cov_matrices = {}
for c in range(10):
    data_c = train_features[train_labels == c]
    class_means[c] = torch.mean(data_c, dim=0)
    cov_matrix = torch.tensor(np.cov(data_c,
        rowvar=False))
    class_inv_cov_matrices[c] = torch.linalg.
        inv(cov_matrix)
```

```
50
51 def classify(features, means, inv_cov_matrices
       ):
52     pred_labels = []
53     for feature in tqdm(features, desc="
           Classifying"):
54         distances = []
55         for c in range(10):
56             m_distance = mahalanobis(feature,
                   means[c], inv_cov_matrices[c])
57             distances.append(m_distance)
58         pred_labels.append(distances.index(min
               (distances)))
59     return torch.tensor(pred_labels)
60
61 train_pred = classify(train_features,
       class_means, class_inv_cov_matrices)
62 test_pred = classify(test_features,
       class_means, class_inv_cov_matrices)
63
64 train_acc = (train_pred == train_labels).float
       ().mean().item()
65 test_acc = (test_pred == test_labels).float().
       mean().item()
66
67 print(f"Training␣Accuracy:␣{train_acc␣*␣100:.2
       f}%")
68 print(f"Testing␣Accuracy:␣{test_acc␣*␣100:.2f
       }%")
```