Image Compression and Recognition—PCA and TensorFlow

Wanfang Zhang[1]

[1]Department of Epidemiology and Biostatistics, University of South Carolina

Abstract

Image data analysis is used in a wide range of fields, including computer vision, medical imaging, remote sensing, and astronomy. In each of these fields, image data analysis plays a critical role in understanding complex phenomena and making informed decisions. In this project we will focus on two subject: Image compression with feature reduction and Image recognition with classification method.

Keywords: PCA, TensorFlow, MNIST data, Keras, neural network model

1 INTRODUCTION

Principal component analysis (PCA) is a popular technique for analyzing large datasets containing a high number of dimensions/features per observation, increasing the interpretability of data while preserving the maximum amount of information, and enabling the visualization of multidimensional data. Formally, PCA is a statistical technique for reducing the dimensionality of a dataset (Jolliffe and Cadima, 2016). This is accomplished by linearly transforming the data into a new coordinate system where (most of) the variation in the data can be described with fewer dimensions than the initial data. Many studies use the first two principal components in order to plot the data in two dimensions and to visually identify clusters of closely related data points. Principal component analysis has applications in many fields such as population genetics, microbiome studies, and atmospheric science. In the case of image data, each pixel can be considered as a feature, and the total number of pixels in an image can be quite large. PCA can be used to compress this image data by reducing the number of features. A neural network model is a type of machine learning model inspired by the structure and function of the human brain. It is composed of multiple interconnected layers of artificial neurons that work together to process and analyze complex input data (Charu, 2018). The basic unit of a neural network is the artificial neuron, which receives input data and computes an output value based on a set of weights and biases. These weights and biases are adjusted during training using an optimization algorithm, such as backpropagation, to minimize the difference between the predicted output and the true output. Neural network models are powerful because they can automatically learn to extract meaningful features from input data without being explicitly programmed to do so. This makes them well-suited for a wide range of tasks, including classification, regression, and pattern recognition. We will use neural network to recognize handwriting digit as classification.

2 METHODOLOGY

2.1 Image Compression

2.1.1 Convert the image into a matrix

The first step is to convert the image into a matrix where each pixel is represented by a value. For grayscale images, each pixel will have a single value, while for color images, each pixel will have three values (one for each color channel).

2.1.2 Standardize the data

To perform PCA, we need to standardize the data so that each pixel has a mean of 0 and a standard deviation of 1. This ensures that the principal components are based on the variations in the image and not on differences in scale.

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \text{ and } \bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$$

2.1.3 Calculate the covariance matrix

The covariance matrix is calculated from the standardized data. This matrix represents the relationships between the pixels and the direction of their variation.

$$C = \begin{pmatrix} \operatorname{cov}(x,x) & \operatorname{cov}(x,y) \\ \operatorname{cov}(y,x) & \operatorname{cov}(y,y) \end{pmatrix}$$

where

$$\operatorname{cov}(x,y) = \operatorname{cov}(y,x) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{n-1}$$

$$\operatorname{cov}(x,x) = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(x_i - \bar{x})}{n-1} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1} = \operatorname{var}(x)$$

$$\operatorname{cov}(y,y) = \frac{\sum_{i=1}^{n} (y_i - \bar{y})(y_i - \bar{y})}{n-1} = \frac{\sum_{i=1}^{n} (y_i - \bar{y})^2}{n-1} = \operatorname{var}(y)$$

2.1.4 Find the eigenvectors and eigenvalues

The eigenvectors and eigenvalues of the covariance matrix represent the principal components of the image. The number of eigenvalues and eigenvectors that exits is equal to the number of dimensions the data set has. The eigenvectors are the directions of the variation, and the eigenvalues represent the amount of variation along each direction.

$$\begin{pmatrix} a & c \\ c & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \lambda \begin{pmatrix} x \\ y \end{pmatrix}$$

2.1.5 Choose the number of principal components

The next step is to choose the number of principal components to retain. This will determine the level of compression. A higher number of principal components will result in a higher quality image but will also require more storage space. In order to decide, We can analyze the percentage of variances explained by each principal component.

2.1.6 Project the data onto the new basis

Finally, the data is projected onto the new basis defined by the chosen principal components. This results in a compressed version of the image that can be stored and reconstructed later.

2.2 Image Recognition

2.2.1 Logistic regression

There are detailed instruction about how it works in textbook Neural networks and deep learning Charu (2018):
suppose x is n-dimension, then the weight matrix W is of size n by K with each column W_k representing the coefficients associated with class k; similarly, the bias vector b is of length K, with each element b_k served as the bias for class k. For simplicity, the bias b can be viewed as an additional row in the weight matrix W. So the probability of x being class k can be expred mathematically as:

$$P(y = k \mid x, W) = \text{ softmax }_k(Wx) = \frac{\exp(w_k x)}{\sum_{j=1}^{j=k} \exp(w_j x)}$$

Where softmax() denotes the softmax function and that is why multinomial logistic regression is often called softmax regression.
Given a set of training samples:

$$\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots \left(x^{(i)}, y^{(i)}\right) \ldots, \left(x^{(m)}, y^{(m)}\right), \text{where } y \in 1, 2, \ldots, K.$$

The optimal model w is obtained by minimizing the cost (also called log loss), which is defined as:

$$J(W) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{k} 1\left\{y^{(i)} = k\right\} \log \frac{\exp(w_k x)}{\sum_{j=1}^{j=k} \exp(w_j x)}\right], \text{ where } 1\left\{y^{(i)} = k\right\} = \begin{cases} 1, & \text{if } y^{(i)} = k \\ 0, & \text{otherwise} \end{cases}$$

As usual we resort to gradient descent, an iterative optimization algorithm, to solve for the optimal w. In each iteration, w moves a step that is proportional to the negative derivative Δw of the objective function at the current point. That is, $w := w - \eta \Delta w$, where η is the learning rate. Each column Δw_k of Δw can be computed as:

$$\Delta w_k = \frac{\partial}{\partial w_j} J(w) = -\sum_{i=1}^{m} \left[x^{(i)}\left(1\left\{y^{(i)} = k\right\} - P\left(y^{(i)} = k \mid x^{(i)}, W\right)\right)\right]$$

The well trained model, the optimal w will be used to classify a new sample $w\prime$ by:

$$y' = \underset{k}{\text{argmax}} \frac{\exp(\boldsymbol{w}_k \boldsymbol{x}')}{\sum_{j=1}^{j=K} \exp(\boldsymbol{w}_j \boldsymbol{x}')} = \underset{k}{\text{argmax}} \left(\boldsymbol{w}_k \boldsymbol{x}'\right)$$

Armed with the mechanics of the multinomial logistic regression we just reviewed, we can then apply it as the first solution to our digit classification project.

2.2.2 *Going from logistic regression to single-layer neural networks*

Basically, logistic regression is a feed-forward neural network without a hidden layer, where the output layer directly connects with the input layer. In other words, logistic regression is a single neuron that maps the input to the output layer. Theoretically, the neural networks with an additional hidden layer between the input and output layer should be able to learn more about the relationship underneath. A multinomial logistic regression as single-layer neural network can be represented graphically as follows:

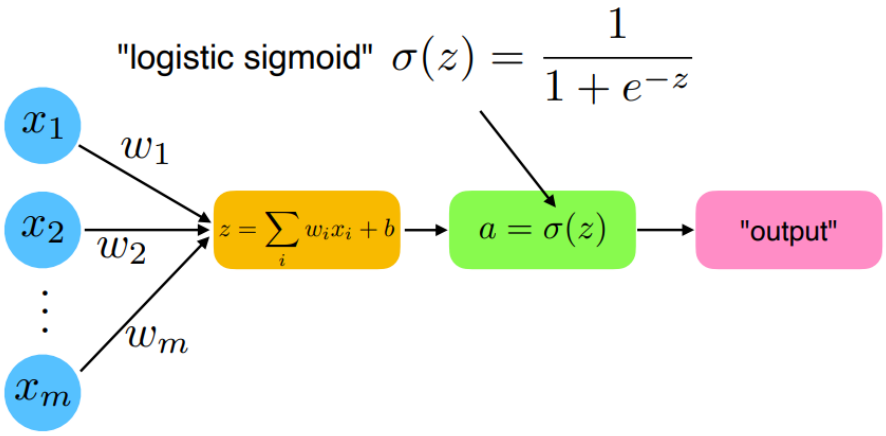


Figure 1. Logistic regression as single-layer neural network model

Suppose x is n-dimension, and there are H hidden units in the hidden layer, then the weight matrix $w^{(1)}$ connecting the input layer to the hidden layer is of size n by H with each column

$$a_h^{(2)} = f\left(z^{(2)}\right) = f\left(w_h^{(1)} x\right)$$

For example, for the outputs of the first, second and the last hidden unit:

$$a_1^{(2)} = f\left(w_1^{(1)} x\right); a_H^{(2)} = f\left(w_H^{(1)} x\right); a_H^{(2)} = f\left(w_H^{(1)} x\right),$$

where $f(z)$ is an activation function.Typical choices for the activation function in simple networks include logistic function (more often called sigmoid function) and tanh function (which can be considered a re-scaled version of logistic function):

$$\text{sigmoid}(z) = \frac{1}{1+e^{-z}}; \ \tanh(z) = \frac{e^{-z} - e^{-z}}{e^{z} + e^{-z}} = \frac{2}{1+e^{-2z}} - 1$$

We will be using the logistic function in our single-layered networks for now. For a case of K possible classes, the weight matrix $w^{(2)}$ connecting the hidden layer to the output layer is of size H by K. Each column $w_k(2)$ represents the coefficients associated with class k. The input to the output layer is the output of the hidden layer $a^{(2)} = \left\{a_1{}^{(2)}, a_2(2), \ldots, a^{(2)}\right\}$, the probability of x being class k can be expressed mathematically as (for consistency, we denote it

$$a_k^{(3)} = f\left(z^{(3)}\right) = \text{softmax}\left(W^{(2)} a^{(2)}\right)$$

Similarly, given m training samples, to train the neural network, we learn all weights $w = \left\{w^{(1)}, w^{(2)}\right\}$ using gradient descent with the goal of minimizing the mean squared error cost $J(w)$. Computation

of the gradients Δw can be realized through the backpropagation algorithm. The idea of the backpropagation algorithm is the following, we first travel through the network and compute all outputs of the hidden layers and output layer; then moving backward from the last layer, we calculate how much each node contributed to the error in the final output and propagate it back to the previous layers. In our single-layer network, the detailed steps are:
1. Compute $a^{(2)}$ for the hidden layer and feed them to the output layer to compute the outputs $a^{(3)}$
2. For the output layer, compute the derivative of the cost function of one sample $j(W)$ with regards to each unit,

$$\delta_k^{(3)} = \frac{\delta}{\delta z_k^{(3)}} j(W) = -\left(y_k - a_k^{(3)}\right) \cdot f'\left(z_k^{(3)}\right)$$

3. For the hidden layer, we compute the error term $\delta^{(2)}$ based on a weighted average of

$$\delta^{(3)} : \delta^{(2)} = \left(\left(W^{(2)}\right)^T \delta^{(3)}\right) \cdot f^{(2)}\Big)$$

4. Compute the gradients applying the chain rule:

$$\triangle W^{(2)} = \frac{\partial J(W)}{\partial z_k^{(3)}} \frac{\partial z_k^{(3)}}{\partial W^{(2)}} = \delta^{(3)} \left(a^{(2)}\right)^T$$

$$\triangle W^{(1)} = \frac{\partial J(W)}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial W^{(1)}} = \delta^{(2)} (x)^T$$

We repeatedly update all weights by taking these steps until the cost function converges.

2.2.3 Multiple layer neural network model

Theoretically, we can obtain a better one with more than one hidden layer. The difference between single layer neural network model and multiple layer neural network model can be viewed:

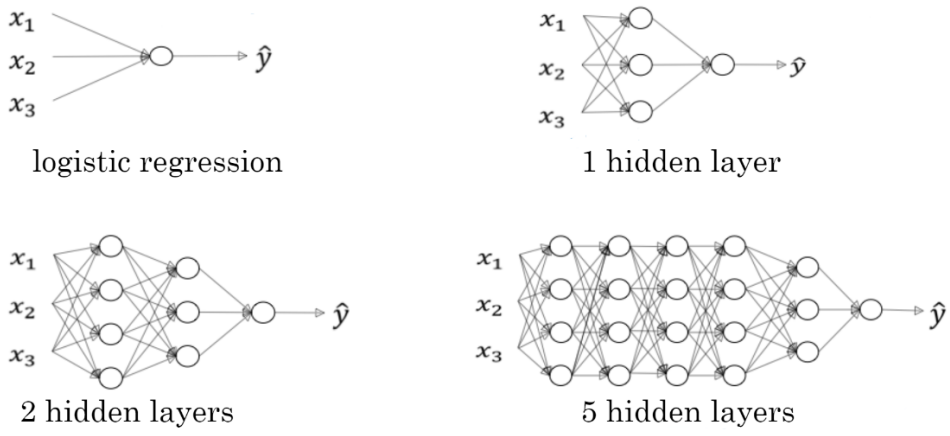


Figure 2. Difference between single layer and multiple layer neural network model

Weight optimization in feed-forward deep neural networks is also realized through the backpropagation algorithm, which is identical to single-layer networks. However, the more layers, the higher

the computation complexity, and the slower the model convergence. One way to accelerate the weight optimization, is to use a more computational efficient activation function. The most popular one in recent years is the rectified linear unit (ReLU):

$$\text{relu}(z) = z^+ = \max(0, z)$$

Thanks to the properties of its derivative, which is

$$\text{relu}'(z) = \begin{cases} 0, z < 0 \\ 1, z \geq 0, \end{cases}$$

there are two main advantages of using the ReLU activation function over sigmoid:
1. Faster learning because of constant value of relu'(z), compared to that of the logistic function

$$\text{sigmoid}(z) = \text{sigmoid}(z) * (1 - \text{sigmoid}(z))$$

2. Less likely to have the vanishing gradient problem, exponential decrease of gradient, which can be found in networks with multiple stacked sigmoid layers. As we multiply the derivative of the activation function when calculating errors δ for each layer, and the maximal value of $\text{sigmoid}'(z)$ is $1/4$, the gradients will decrease exponentially as we stack more and more sigmoid layers.

We represent the network in the data type symbol. Begin with the input layer data, the input data, and follow up with the first hidden layer fc1 with 256 nodes, which fully connects with the input layer. We then attach the ReLU function to fc1 and output the activations act1 for this layer. Similarly, we chain another hidden layer fc2, with 128 nodes this time, and output ReLU-based activates act2, chain another hidden later fc3 with 64 nodes similar to last move. Finally, we end up with the output layer with a softmax function, generating 10 probabilities corresponding to 10 classes. The overall structure looks like this:

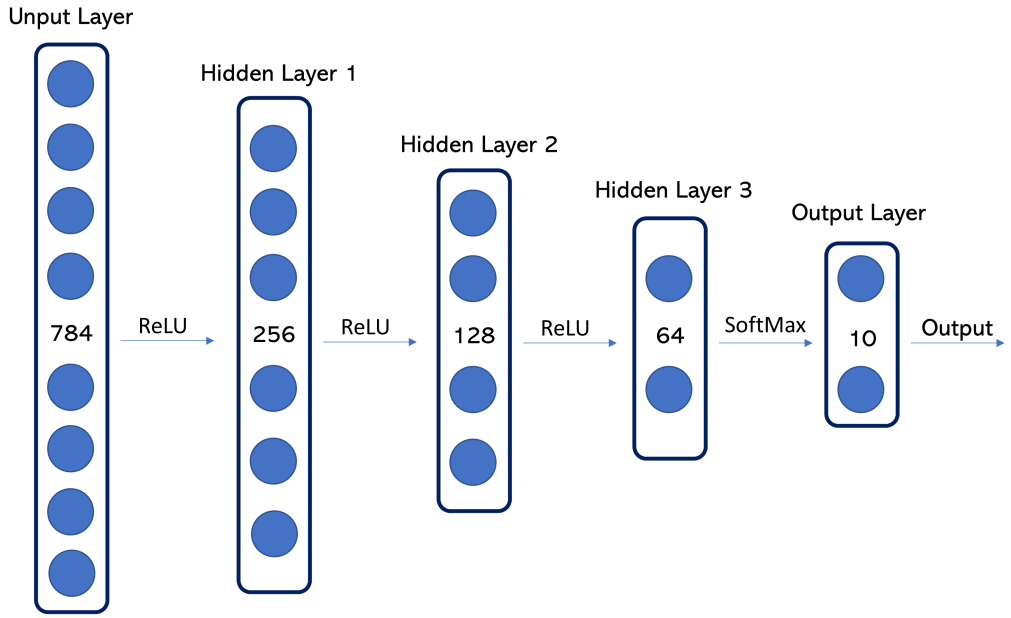


Figure 3. Our multiple layer neural network model

2.2.4 Cross-validation

Cross-validation (Browne, 2000) is a technique that is commonly used to evaluate the performance of machine learning models. The basic idea behind cross-validation is to partition the dataset into training and testing sets, and then use the training set to train the model and the testing set to evaluate its performance. However, a single split of the dataset may lead to overfitting or underfitting of the model.

To overcome this issue, we use the cross-validation technique, which involves dividing the dataset into several non-overlapping subsets, called folds. Each fold is used as a testing set once, and the remaining folds are used for training the model. This process is repeated for all folds, and the performance of the model is averaged over all the folds. In this project we used two types of cv: 5-fold cross-validation and 10-fold cross-validation.

3 RESULT AND CONCLUSION

3.0.1 Image Compression

P.S. Results on other page

3.0.2 Image Recognition

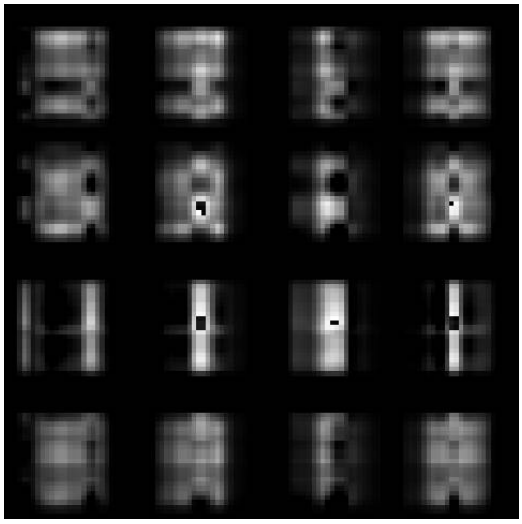
	Multinomial logistic regression	Multiple layer neural network model
One-time	0.9166	0.9836
5-fold CV	0.9179	0.9811
10-fold CV	0.9082	0.9779

REFERENCES

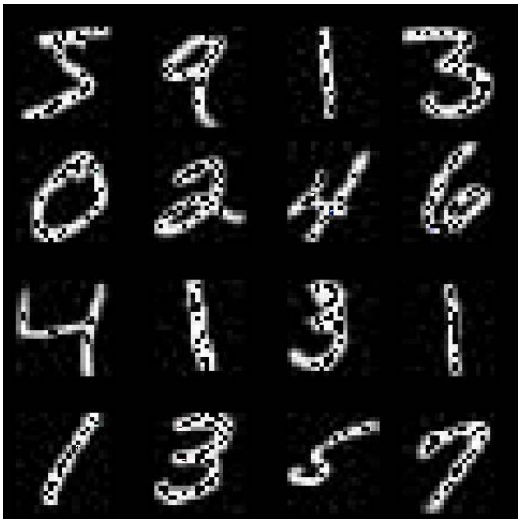
Browne, M. W. (2000). Cross-validation methods. *Journal of mathematical psychology*, 44(1):108–132.

Charu, C. A. (2018). Neural networks and deep learning: a textbook.

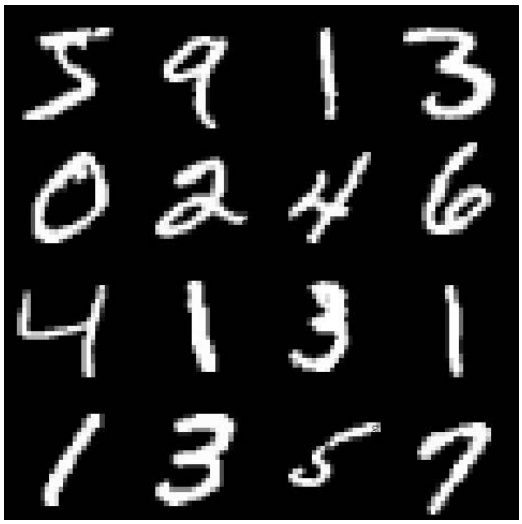
Jolliffe, I. T. and Cadima, J. (2016). Principal component analysis: a review and recent developments. *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150202.



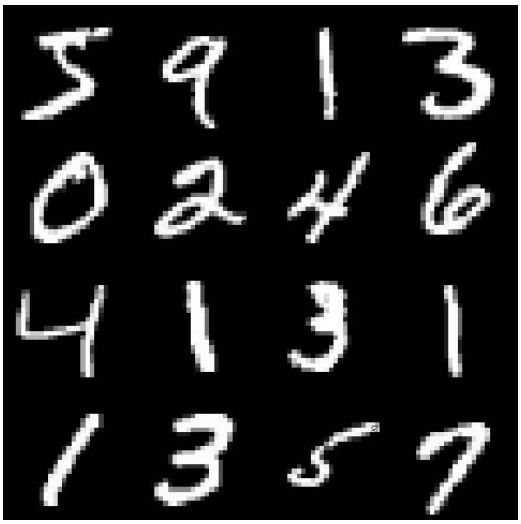
(a) Keep 2 PC in reconstruction



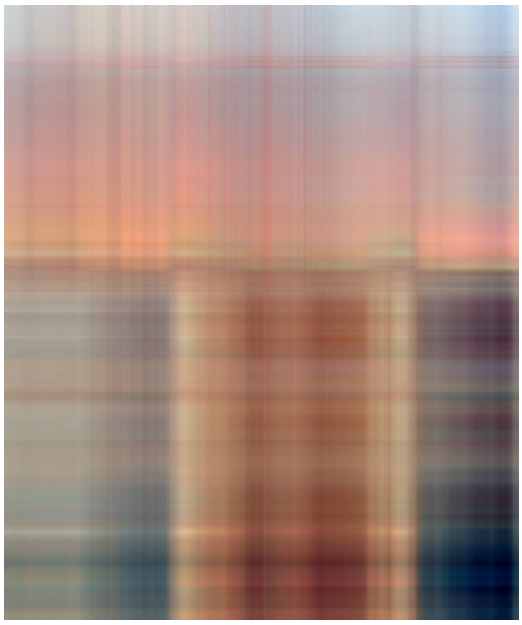
(b) Keep 30 PC in reconstruction



(c) Keep 200 PC in reconstruction

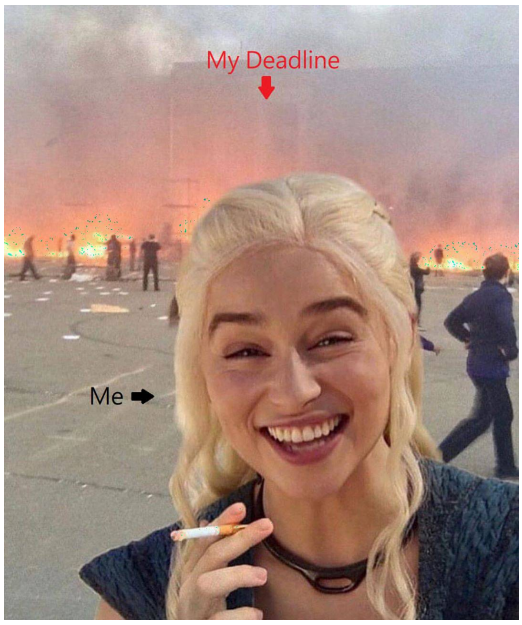


(d) Keep 300 PC in reconstruction



(a) Keep 2 PC in reconstruction

(b) Keep 30 PC in reconstruction



(c) Keep 200 PC in reconstruction

(d) Keep 600 PC in reconstruction