COMPARISON OF UNSUPERVISED PRE-TRAINING METHODS

Jifu Zhao

University of Illinois at Urbana-Champaign
Department of Nuclear, Plasma, and Radiological Engineering
Urbana, Illinois 61801, USA

ABSTRACT

Unsupervised per-training methods are very important steps for many applications. In this work, we focused on pre-training feature extraction methods. More specifically, we focus on Principal Component Analysis (PCA), kernel PCA and auto-encoders. By comparing their effect for a multiclass digits classification problem, we give an direct comparison of the performance of different methods.

***Index Terms*—** Unsupervised pre-training, PCA, kernel PCA, Auto-encoder

1. INTRODUCTION

As one important part of artificial intelligence (AI), machine learning has been widely used in our daily life, such as natural language processing (NLP), zip code recognition, self-driving cars and so on. Among those applications, supervised learning is the most common and perhaps the most powerful form of machine learning [1]. For supervised learning, the choice of appropriate representations of original features is very important [2]. For a long time, most machine learning applications require us to specify the input $\boldsymbol{x}$ and target $\boldsymbol{y}$. The choice of feature representation $\boldsymbol{x}$ is usually determined by experts. And sometimes the input $\boldsymbol{x}$ may be just the raw feature without pre-processing.

Researchers have been working on discovering the hidden structures of the data for years. In this process, several methods have been developed and found to be useful in some domains, such as PCA, Independent Component Analysis (ICA), Non-negative Matrix Factorization (NMF), restricted Boltzmann machine (RBM), Auto-encoders and so on. These methods works in a unsupervised way and the output from these methods are thought to be a transformation of the raw input features. For supervised learning, some supervised learning mechanism will be built based on the output from these pre-training methods rather than the raw features. As expected, a good representation can help the supervised learning algorithms work better.

As one of the most widely used method, PCA has proved to be a powerful technique for extracting structures from possibly high-dimension data sets [3]. PCA is essentially a eigen-decomposition problem [4]. As an orthogonal transformation, however, PCA doesn't work well when the latent variables of the data are non-linearly related to the original data. To extract useful information in a non-linearly way, many methods has been proposed, such as kernel PCA and auto-encoders.

Kernel PCA was proposed by Schölkopf *et al.*[3]. In their work, starting from the traditional PCA algorithm, they found that, after mapping the data from original feature space $\boldsymbol{x}$ into a new feature space through some nonlinear function $\boldsymbol{\phi}(\boldsymbol{x})$, PCA algorithm requires huge amount of computations when the dimensionality of the new feature space is extremely high. Instead of solving this problem explicitly in the new feature space, they extracted the principal components through the so-called kernel matrix. In this way, this problem can be transformed into solving the eigendecomposition problem of the kernel matrix, whose dimensionality is determined by the size of the data set rather than the new feature space. This method is extremely useful when the new feature space is extremely high (for example, 10^{10}). Using this method, the authors chose the US Postal Service database for a multicalss classification problem. Using only a subset of 3000 training examples and a soft margin hyperplane classifier, they got a very good accuracy, which was compatible with support vector machine and convolutional neural networks [3].

In addition to kernel PCA, auto-encoder is another commonly used non-linear feature extraction method. Auto-encoders use the input as the target, trying to find a reconstruction $\boldsymbol{r}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{h}(\boldsymbol{x}))$ through encoder $\boldsymbol{h}(\cdot)$ and decoder $\boldsymbol{g}(\cdot)$ [2]. In most cases, an auto-encoder system is a multi-layer deep neural network which has a bottleneck structure. Auto-encoders can be trained through back-propagation. However, it works well only if we have a good initialization, especially when there are many hidden layers. To get a good initialization, Hinton and Salakhutdinov [5] then proposed an effective way which utilize a stack of RBMs for pre-training. They first built a stack of RBMs, each having only one layer of feature detectors. The learned feature representations from one RBM are used as the input for the next RBM. Since each RBM was trained separately, it was much easier than training a deep auto-encoder. After pre-training, the learned weights of the RBMs were used as the initialization for the corresponding deep auto-encoders. The auto-encoder was

then trained further through back-propagation. Following the above steps, they applied this method on several data sets, including MNIST data set [6], Olivetti face data set [7] and Reuters Corpora [8], which showed good results.

In this work, we focus on the comparison of several unsupervised per-training methods. More specifically, we will focus on PCA, kernel PCA and auto-encoders (with and without RBM initialization). In section 2, we gave some brief review for PCA, kernel PCA and auto-encoders. In section 3, we conducted experiments of MNIST data set using PCA kernel PCA and auto-encoder methods. Finally, in section 4, we gave some conclusions and potential future works.

2. METHODS

2.1. PCA

The goal of PCA is to find a lower-dimension space such that the variance of the orthogonal projection of the original data is maximized [9]. PCA is essentially performed by solving the eigendecomposition problem of the covariance matrix. The PCA algorithm can be summarized in Algorithm 1.

Algorithm 1 PCA in Feature Space

1: **procedure** PCA(*X)*
2: *given input:* $X_{n\times m} \leftarrow \left[\boldsymbol{x}_1; \boldsymbol{x}_2; \cdots; \boldsymbol{x}_n\right]^T$
3: *de-mean:* $x_{ij} \leftarrow x_{ij} - \bar{x}_j$ *or* $x_{ij} \leftarrow \frac{x_{ij}-\bar{x}_j}{s_j}$
4: *covariance matrix:* $C \leftarrow \frac{1}{n}X^TX$
5: *eigendecomposition:* $\lambda\boldsymbol{u} = C\boldsymbol{u}$
6: *first k eigenvectors:* $U_{m\times k} \leftarrow \left[\boldsymbol{u}_1; \boldsymbol{u}_2; \cdots; \boldsymbol{u}_k\right]$
7: *project the test data* $\boldsymbol{x}$: $\boldsymbol{p} \leftarrow U^T\boldsymbol{x}$
8: **finish**

2.2. Kernel PCA

Suppose that we first map the input $\boldsymbol{x}$ from original feature space $\mathbb{R}^m$ into a new feature space $\mathbb{F}^d$ through some function $\boldsymbol{\phi(x)}$ (usually $d > m$ or even $d \gg m$):

$$\boldsymbol{\phi} : \mathbb{R}^m \rightarrow \mathbb{F}^d \tag{1}$$

For simplicity, let's first suppose that the projected data is centered, that is $\sum_{i=1}^n \boldsymbol{\phi}(\boldsymbol{x}_i) = \boldsymbol{0}$. Then the covariance matrix in this space is:

$$C = \frac{1}{n}\sum_{i=1}^n \boldsymbol{\phi}(\boldsymbol{x}_i)\boldsymbol{\phi}(\boldsymbol{x}_i)^T \tag{2}$$

and its eigendecomposition can be written as:

$$C\boldsymbol{v} = \lambda\boldsymbol{v} \tag{3}$$

Note that $\boldsymbol{v}$ can be written as a linear combination of $\boldsymbol{\phi}(\boldsymbol{x}_i)$:

$$\boldsymbol{v} = \sum_{i=1}^n \alpha_i\boldsymbol{\phi}(\boldsymbol{x}_i) \tag{4}$$

From equation (2), (3)and (4)

$$\frac{1}{n}\sum_{i=1}^n \boldsymbol{\phi}(\boldsymbol{x}_i)\boldsymbol{\phi}(\boldsymbol{x}_i)^T\sum_{j=1}^n \alpha_j\boldsymbol{\phi}(\boldsymbol{x}_j) = \lambda\sum_{i=1}^n \alpha_i\boldsymbol{\phi}(\boldsymbol{x}_i) \tag{5}$$

Now, if we multiply both sides of equation (5) by $\boldsymbol{\phi}(\boldsymbol{x}_k)^T$ and express $\boldsymbol{\phi}(\boldsymbol{x}_i)^T\boldsymbol{\phi}(\boldsymbol{x}_j)$ as $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, we can get:

$$\frac{1}{n}\sum_{i=1}^n k(\boldsymbol{x}_k, \boldsymbol{x}_i)\sum_{j=1}^n \alpha_j k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \lambda\sum_{i=1}^n \alpha_i k(\boldsymbol{x}_k, \boldsymbol{x}_i) \tag{6}$$

Now, let's define $n \times n$ kernel matrix K to be:

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_i) = \boldsymbol{\phi}(\boldsymbol{x}_i)^T\boldsymbol{\phi}(\boldsymbol{x}_j) \tag{7}$$

In matrix form, this is:

$$K^2\boldsymbol{\alpha} = n\lambda K\boldsymbol{\alpha} \tag{8}$$

where $\boldsymbol{\alpha}$ is a n-dimensional column vector. We can find $\boldsymbol{\alpha}_i$ and λ_i by solving the following eigendecomposition problem:

$$K\boldsymbol{\alpha}_i = n\lambda_i\boldsymbol{\alpha}_i \tag{9}$$

Also, note that we require $\boldsymbol{v}_i^T\boldsymbol{v}_i = 1$, so we need to normalize $\boldsymbol{\alpha}_i$ such that:

$$\lambda_i n\boldsymbol{\alpha}_i^T\boldsymbol{\alpha}_i = 1 \tag{10}$$

From above equations, the projection of any given data $\boldsymbol{x}$ onto the ith eigenvector $\boldsymbol{v}v_i$ can be written as:

$$p_i(\boldsymbol{x}) = \boldsymbol{\phi(x)}^T\boldsymbol{v}_i = \sum_{j=1}^n \alpha_{ij}k(\boldsymbol{x}, \boldsymbol{x}_j) \tag{11}$$

At the begining, we assume that the projected data is centered. However, there is no guarantee that the projected data set given by $\phi(\boldsymbol{x}_n)$ has zero mean, so we need manually centralize the projected data by [9]:

$$K' = K - \mathbb{I}_nK/n - K\mathbb{I}_n/n + \mathbb{I}_nK\mathbb{I}_n/n^2 \tag{12}$$

where $\mathbb{I}_n$ stands for $n \times n$ matrix with all values equal to 1.

The above kernel PCA algorithm can be summarized in Algorithm 2.

Some commonly used kernels includes:

$$\textit{Polynomial kernel: } k(\boldsymbol{x}, \boldsymbol{y}) = (\gamma\boldsymbol{x}^T\boldsymbol{y} + c)^d \tag{13}$$

$$\textit{Gaussian kernel: } k(\boldsymbol{x}, \boldsymbol{y}) = exp(-\gamma||\boldsymbol{x} - \boldsymbol{y}||^2) \tag{14}$$

2.3. Auto-encoders

As mentioned in section 1, auto-encoders use the input as the target, trying to reconstruct the input through encoder and decoder [2]. A simple 5 layer auto-encoder system is shown in Figure 1.

Algorithm 2 Kernel PCA

procedure KERNEL PCA(X)
  given input: $X_{n\times m} \leftarrow \left[\boldsymbol{x}_1; \boldsymbol{x}_2; \cdots; \boldsymbol{x}_n\right]^T$
  kernel matrix $K_{n\times n}$: $k_{ij} \leftarrow k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
  $K' \leftarrow K - \mathbb{I}_n K/n - K\mathbb{I}_n/n + \mathbb{I}_n K \mathbb{I}_n/n^2$
  eigendecomposition: $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \cdots \leftarrow n\lambda\boldsymbol{\alpha} = K'\boldsymbol{\alpha}$
  normalization: $\boldsymbol{\alpha}_i \leftarrow n\lambda_i \boldsymbol{\alpha}_i^T \boldsymbol{\alpha}_i = 1$
  project the test data $\boldsymbol{x}$
finish

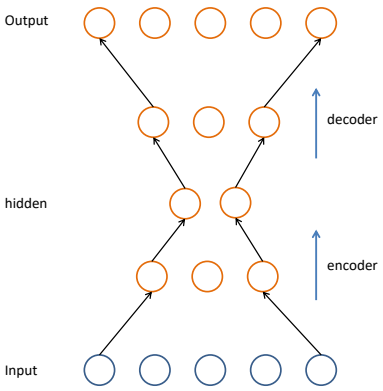


Fig. 1. Illustration of auto-encoder system.

Suppose the auto-encoder has L layers, where 1_{st} layer is the input $\boldsymbol{x}$ and L_{th} layer is the output layer. In this work, we use sigmoid function as the activation function:

$$sigmoid(z) = \frac{1}{1 + exp(-z)} \tag{15}$$

For the output layer, we use the squared-error as the cost function:

$$E = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=0}^{m}(y_{ij}^{(L)} - x_{ij})^2 \tag{16}$$

Followed the back-propagation rule, the update rule can be easily derived.

3. EXPERIMENTS

3.1. Data

In this work, we used the MNIST dataset [6] to compare the performance of different pre-training methods. The MNIST data set contains a huge amount of handwirtten digits, including 55000 training examples, 5000 validation examples and 10000 test examples. There were 10 classes which corresponding to digits from 0 to 9. Each example is a 28×28 pixel grayscale image and the pixel values range from 0 to 1.

Since the main goal of this work was to compare different pre-training methods, in this work, we only implemented a simple 10-way multiclass logistic regression neural network for final classification.

3.2. PCA vs. kernel PCA

In this part, we focus on the comparison of PCA and kernel PCA. More specifically, we apply PCA and kernel PCA model with different polynomial kernels (as denoted in equation 13). Due to huge computation requirement for kernel PCA, we select a subset of $20,000$ training examples to build the kernel PCA model, then transform training and test data set according to this model. After choosing different number of transformed features, we perform multiclass classification. The test accuracy are shown in Table 1.

With polynomial kernels, we have much more features can be chosen. As a result, the test accuracy gets improved a lot, which is shown clear in Table 1. Our best result is with 10-degree polynomial kernel and 16384 features. The training accuracy and test accuracy are 0.9915 and 0.9769 respectively (as a comparison, the training accuracy and test accuracy is 0.9340 and 0.9267 respectively when classified using the original 784 features as the input).

3.3. PCA vs. Auto-encoder

In this part, we focus on bottleneck auto-encoders. More specifically, we focused on the 5 layer structure which is shown in Figure 1.

Each MNIST example has 784 features. Choosing 784 features as the input, we can build the auto-encoder with different number of hidden nodes, for example, 784-512-256-512-784, where the decoded feature representation has 256 dimensions. Then, the decoded feature representations are used as the input into a multiclass classification neural network as described in Section 3.1 and 3.2. The test accuracy is shown in Table 2. The number of first hidden layer nodes are shown in horizontal direction while the number of second hidden layer nodes are shown in vertical direction. For comparison purpose, the results from PCA are also shown in Table 2.

Using auto-encoders, the training and testing accuracy is sensitive to the overall structures. With the structure of 784-512-256-512-784, we can reach 0.9437 test accuracy, which are better than using PCA. However, when we reduce the original feature heavily, for example, using the structure of 784-128-16-128-784, the corresponding test accuracy is only 0.7662. So, when using auto-encoder, it is important to find an appropriate structure such as the number of hidden layers and the number of nodes for each layer.

Another important issue with auto-encoder is the initialization. A good initialization is very important for the final result. In this work, we initialize the auto-encoder with small

Table 1. Test Accuracy for Different Kernels

	Polynomial Kernel Degrees									
Components	1	2	3	4	5	6	7	8	9	10
32	0.9005	0.8777	0.8816	0.8846	0.8877	0.8901	0.8912	0.8932	0.8938	0.8944
64	0.9155	0.8952	0.8992	0.9013	0.9044	0.9075	0.9088	0.9115	0.9130	0.9146
128	0.9222	0.8984	0.9062	0.9122	0.9161	0.9197	0.9239	0.9273	0.9302	0.9329
256	0.9235	0.9058	0.9153	0.9255	0.9320	0.9365	0.9404	0.9437	0.9462	0.9488
512	0.9255	0.9220	0.9316	0.9391	0.9445	0.9483	0.9520	0.9557	0.9584	0.9597
784	0.9242	0.9262	0.9385	0.9451	0.9498	0.9536	0.9578	0.9602	0.9626	0.9641
1024	N.A.	0.9276	0.8403	0.9475	0.9530	0.9569	0.9604	0.9632	0.9656	0.9672
2048	N.A.	0.9291	0.9436	0.9510	0.9587	0.9629	0.9662	0.9690	0.9702	0.9716
4096	N.A.	0.9293	0.9454	0.9525	0.9610	0.9652	0.9676	0.9712	0.9728	0.9753
8192	N.A.	0.9305	0.9466	0.9514	0.9630	0.9672	0.9693	0.9722	0.9739	0.9763
16384	N.A.	0.9314	0.9473	0.9534	0.9635	0.9683	0.9709	0.9730	0.9746	0.9769

Table 2. Test Accuracy for Auto-encoder and PCA

	Without RBMs (1st layer)				With RBMs (1st layer)		
2nd layer	128	256	512	PCA	128	256	512
16	0.7698	0.8339	0.8524	0.8551	0.8550	0.8687	0.8742
32	0.8160	0.8758	0.8778	0.9005	0.8914	0.8937	0.8917
64	0.8710	0.8937	0.9047	0.9155	0.9152	0.9159	0.9115
128	0.9056	0.9164	0.9202	0.9222	0.9293	0.9247	0.9174
256	N.A.	0.9367	0.9409	0.9235	N.A.	0.9458	0.9412

random values and train it with back-propagation. After running it several times, we notice that the final test accuracy can be really low for several times. To solve this problem, followed the steps proposed by Hinton and Salakhutdinov [5], we first train the stacked RBMs with the same number of hidden nodes and use the learned weights as the initialization for auto-encoders and the final test accuracy is also shown in Table 2.

From Table 2, the effect of RBM initialization is clear, especially when the number of hidden nodes are small. For example, with the structure of 784-128-16-128-784, the test accuracy without RBMs initialization is only 0.7698 while the test accuracy with RBMs initialization can reach 0.8550. However, this difference is not significant when we have more hidden nodes.

4. CONCLUSION AND FUTURE WORK

This work compared several pre-training method, including PCA, kernel PCA with polynomial kernels and auto-encoders. With appropriate kernels, kernel PCA can give us a better representation as a non-linear transformation in much higher feature space, which can finally help us to get better results. However, kernel PCA requires the computation of kernel matrix, which will require huge computations when the number of samples are very large. Bottleneck auto-encoders, on the other hand, although don't increase the feature space, but it reduce the dimensionality in a non-linear way. With appropriate structures, auto-encoder can also give us better result than linear PCA. For deep auto-encoders, stacked RBMs can give us a better initialization.

In this work, we focus on the comparison of PCA, kernel PCA and auto-encoders as pre-training step. It helps us get some insight on the effect of different pre-training steps. But these methods is only a small part of pre-training methods. Other methods, such as ICA, NMF, denosing auto-encoders, contractive auto-encoders and so on, are also widely used in different fields. Research on these methods and their variants are very important. In addition, initialization and training methods for neural network based methods, such as unfolding stacked RBMs for auto-encoders, are also very important for final result. Finding better initialization and training methods to avoid local minimum and overfitting problems are also important for future works.

5. REFERENCES

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] Yoshua Bengio et al., "Deep learning of representations for unsupervised and transfer learning.," *ICML Unsupervised and Transfer Learning*, vol. 27, pp. 17–36, 2012.

[3] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller, "Nonlinear component analysis as a kernel eigenvalue problem," *Neural computation*, vol. 10, no. 5, pp. 1299–1319, 1998.

[4] Jerome Friedman, Trevor Hastie, and Robert Tibshirani, *The elements of statistical learning*, vol. 1, Springer series in statistics Springer, Berlin, 2001.

[5] Geoffrey E Hinton and Ruslan R Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

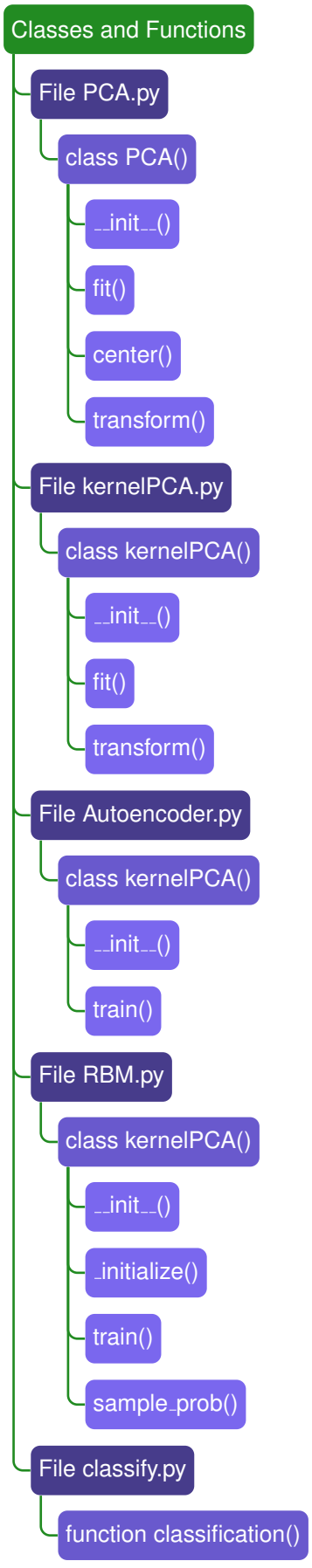
[6] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

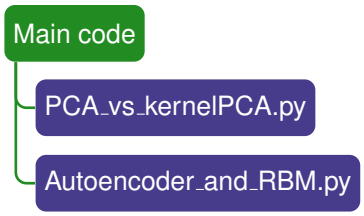
[7] Ferdinando S Samaria and Andy C Harter, "Parameterisation of a stochastic model for human face identification," in *Applications of Computer Vision, 1994., Proceedings of the Second IEEE Workshop on*. IEEE, 1994, pp. 138–142.

[8] David D Lewis, Yiming Yang, Tony G Rose, and Fan Li, "Rcv1: A new benchmark collection for text categorization research," *Journal of machine learning research*, vol. 5, no. Apr, pp. 361–397, 2004.

[9] Christopher Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics), 1st edn. 2006. corr. 2nd printing edn*, Springer, New York, 2007.

6. CODE DESCRIPTION





In this part, we will give some brief description of the code written for this project. The main part of this project are included in the file named: PCA_vs_kernelPCA.py and Autoencoder_and_RBM.py.

PCA_vs_kernelPCA.py :
Contains all the required part for the comparison of PCA and kernel PCA.

Autoencoder_and_RBM.py :
Contains all the required part for the comparison of auto-encoder and RBMs.

In this project, we implemented the PCA, kernel PCA, Auto-encoder and RBM, all of these algorithms were packaged into separate classes and saved in separate files. The overall structure is shown in the left figure.

PCA.py :
Main part for class PCA(): train and transform data according to built PCA model.

kernelPCA.py :
Main part for class kernelPCA(): train and transform data according to built kernel PCA model.

Autoencoder.py :
Main part for class Autoencoder(): train and transform data according to built auto-encoder model.

RBM.py :
Main part for class RBM(): train and transform data according to built RBM model.

classify.py :
Main part for function classification(): perform multi-class logistic regression for classification.

All codes are implemented in Python. The used package including: numpy, scipy, tensorflow, tflearn, matplotlib, sklearn.