Paper Review: Kernel Analysis of Deep Networks

Jiahao Yang
Department of Statistics
University of Washington
Seattle, WA 98195
yangjh39@uw.edu

Abstract

In this paper review, I analyze the layer-wise evolution of the representation in two deep networks, convolutional neural network and multilayer perceptron by building a sequence of deeper and deeper kernels. Using kernel principal component analysis, I find that deep networks create better and better representations layer after layer and the structure of the deep network has influence on how fast the representations are built.

Keywords: deep network, kernel principal component analysis, representations

1 Introduction

The goal of this paper review is to study how exactly the representation is built in a deep network, in particular, how the representation evolves as we map the input through more and more layers of the deep network by using kernel principal component. Here, the kernel PCA is used as an abstraction tool for modeling the deep network.

My analysis takes a trained deep network and then get the representer of the dataset in each layer. After extracting the representer out of deep network, I use kernel PCA to analysis how good the representations yielded by these deeper and deeper layers are. Since the simpler and more accurate the representation is, the better the representation is and using kernel PCA, the number of kernel principal components can control the simplicity of the model, I quantify for each kernel how good the representation with respect to the learning problem is by measuring how much task relevant information is contained in the leading principal components of the kernel feature space. In this part, deep network, multilayer perceptron(MLP) is explored.

In addition to exploring how good the representations generated by deeper and deeper layer, I also analyze the statement that the structure of the deep network controls how fast the representation of the task is formed layer after layer. In this part, two deep network, convolutional neural network(CNN) and multilayer perceptron(MLP) are analyzed.

2 Kernel and kernel based method

2.1 Gaussian RBF kernel

2.1.1 Definition

The definition of a Gaussian RBF kernel with hyper-parameter $\sigma > 0$ on two sample $x, x' \in \mathbb{R}^d$ is

$$k(x, x') = \exp(-\frac{\|x - x'\|^2}{\sigma^2}).$$

2.1.2 Positive semi-definite

Now we begin to prove that Gaussian RBF kernel $k(\cdot,\cdot)$ is a positive semi-definite kernel.

Without loss of generality, assume $\sigma = \sqrt{2}$. Then, $\forall x_1, \cdots, x_n \in \mathbb{R}^d$, and we will have

$$\|x_i - x_j\|^2 = x_i^T x_i - 2x_i^T x_j + x_j^T x_j,$$
$$k(x_i, x_j) = \exp(-\frac{x_i^T x_i}{2}) \exp(x_i^T x_j) \exp(-\frac{x_j^T x_j}{2}).$$

By Taylor expansion,

$$\exp(x_i^T x_j) = \sum_{k=0}^{\infty} \frac{(x_i^T x_j)^k}{k!}.$$

Here, I need to prove two facts which will be needed to prove my goal that Gaussian RBF kernel is positive semi-definite.

First, if $k_i, i = 1, 2, \cdots$ are positive semi-definite kernels, then $\lim_{i\to\infty} k_i$ is also a positive semi-definite kernel if the limit exists.

This conclusion follows by the definition of positive semi-definite kernel. For all i,

$$s_i = \sum_{k,\ell} \alpha_k \alpha_\ell k_i(x_k, x_\ell) \geq 0,$$

then

$$\lim_{i\to\infty} s_i \geq 0.$$

Second, if g is a positive semi-definite kernel, and $f : \mathbb{R}^d \to \mathbb{R}$ is a function. Then $k(x, x') = f(x)g(x, x')f(x')$ is also a positive semi-definite kernel.

It follows from the fact that for all $\alpha_1, \cdots, \alpha_n \in \mathbb{R}$,

$$\sum_{i,j} \alpha_i \alpha_j f(x_i) g(x_i, x_j) f(x_j) = \sum_{i,j} \beta_i \beta_j g(x_i, x_j) \geq 0,$$

where $\beta_i = \alpha_i f(x_i) \in \mathbb{R}$.

Then, we can let $g_n(x, y) = \sum_{k=0}^{n} \frac{(x^T y)^k}{k!}$. Since for each fixed n, $g_n(x, y)$ is a finite sum of polynomial kernels, $g_n(x, y)$ are positive semi-definite. $\exp(x_i^T x_j)$ is a positive semi-definite kernel because the first fact and then RBF kernel is also positive semi-definite based on the second fact.

2.2 Kernel principal component

2.2.1 Theory

In this paper, Kernel PCA is used to analyze the performance and the structure of deep networks. If we want to find the first kernel principal component(KPC1), the optimization problem that Kernel PCA aims to solve is like the problem below.

$$f_1 = \arg\max_{f\in\mathcal{H}} \frac{\langle f, \hat{S} f\rangle_{\mathcal{H}}}{\|f\|_{\mathcal{H}}^2},$$

where $\mathcal{H}$ is RKHS and $\hat{S}$ is the covariance operator defined as

$$\hat{S} = \frac{1}{n} \sum_{i=i}^{n} [k(x_i, \cdot) - \hat{\mu}] \otimes [k(x_i, \cdot) - \hat{\mu}],$$
$$\langle f, \hat{S} g\rangle_{\mathcal{H}} = \frac{1}{n} \sum_{i=1}^{n} \langle f, k(x_i, \cdot) - \hat{\mu}\rangle_{\mathcal{H}} \langle g, k(x_i, \cdot) - \hat{\mu}\rangle_{\mathcal{H}},$$

where $\hat{\mu} = \frac{1}{n}\sum k(x_i, \centerdot)$ is the empirical mean element.

Suppose we have centered the kernel matrix, then we have

$$f_1 = \underset{f\in\mathcal{H}}{\arg\max} \frac{1}{n\|f\|_{\mathcal{H}}^2} \sum_i^n (f(x_i))^2.$$

The solution f_1 is the same for the following optimizing problem that can be solved by using **Representer Theorem**:

$$min \;\; \psi(f(x_1), \cdots, f(x_n), \|f\|_{\mathcal{H}}^2)$$

.

The solution has the form $f(x) = \sum_{i=1}^n \alpha_i k(x_i, x)$, where $\alpha_i \in \mathbb{R}$. Therefore, by using **Representer Theorem**, we transform a problem of finding functions in $\mathcal{H}$ to an equivalent on of finding parameters in $\mathbb{R}^n$.

Thus, the optimization problem becomes

$$\alpha^{(1)} = \underset{\alpha\in\mathbb{R}^n}{\arg\max} \frac{\alpha^T \tilde{K}^2 \alpha}{n\alpha^T \tilde{K}\alpha},$$

where $\tilde{K}$ is the centered kernel matrix.

Similar to KPC1, the j-th Kernel Principal Component is

$$f_j = \underset{f\perp span\{f_1, f_2, \cdots, f_{j-1}\}}{\arg\max} \frac{1}{n\|f\|_{\mathcal{H}}^2} \sum_i^n (f(x_i))^2,$$

where the solution f_j belongs to the function space that is orthogonal to all preceding Kernel Principal Components $f_1, f_2, \cdots, f_{j-1}$ and can also be transformed into finding the corresponding parameter $\alpha^{(j)}$.

Now we show how the problem is equivalent to finding the eigenvector correspond to the largest eigenvalue. By the Spectral Theorem,

$$\tilde{K} = \sum_{p=1} \lambda_p e_p e_p^T,$$

where e_p are unit-length eigen vectors and λ_p are eigenvalues in decreasing order.

Let $\alpha = \sum_{j=1}^n \beta_j e_j$, since e_i are orthogonal and unit-length,

$$\underset{\alpha\in\mathbb{R}^n}{\arg\max} \frac{\alpha^T \tilde{K}^2 \alpha}{n\alpha^T \tilde{K}\alpha} = \underset{\alpha\in\mathbb{R}^n}{\arg\max} \frac{\sum_j \beta_j^2 \lambda_j^2}{n\sum_j \beta_j^2 \lambda_j} (\tilde{K} e_j = \lambda_j e_j),$$

That is to say, the maximums of the objective function are the eigenvalues. Then

$$\underset{\alpha\in\mathbb{R}^n}{\arg\max} \frac{\sum_j \beta_j^2 \lambda_j^2}{n\sum_j \beta_j^2 \lambda_j} \Rightarrow \alpha^{(j)} = \frac{1}{\sqrt{\lambda_j}} e_j.$$

Where, e_j is the eigenvector correspond to λ_j. To normalize $\alpha^{(j)}$ so that $\alpha^T \tilde{K}\alpha = 1$, we have $\alpha^{(j)} = \frac{1}{\sqrt{\lambda_j}} e_j$.

2.2.2 Algorithm

The Kernel PCA algorithm can be summarized as follow,

Algorithm 1 Kernel PCA

```
Input: Kernel Matrix $K$;
Output: Principal component $\alpha$;
1: Center kernel matrix $K = (I - \frac{1}{n}\mathbf{1}\mathbf{1}^T)K(I - \frac{1}{n}\mathbf{1}\mathbf{1}^T)$;
2: Compute eigenvectors $e_j$, eigenvalues $\lambda_j$ of $K$;
3: The $j$-th principal component is the normalized eigenvector $\alpha_{(j)} = \frac{1}{\sqrt{\lambda_j}} e_j$.
```

2.3 Power Iteration

2.3.1 The convergence of Power Iteration Algorithm

For simplicity, we assume that A has eigenvalues $\lambda_1, \cdots, \lambda_n$ such that

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$$

We also assume that A is diagonalizable, meaning that it has n linearly independent eigenvectors $x_1, x_2, \cdots, x_n$ such that $Ax_i = \lambda_i x_i$ for $i = 1, 2, \cdots, n$.

Suppose that we continually multiply a given vector $x^{(0)}$ by A, generating a sequence of vectors $x^{(1)}, x^{(2)}, \cdots$ defined by

$$x^{(k)} = Ax^{(k-1)} = A^k x^{(0)}, \;\; k = 1, 2, \cdots$$

.

Because A is diagonalizable, any vector in $\mathcal{R}^n$ is a linear combination of the eigenvectors, and therefore we can write

$$x^{(0)} = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

.

We then have

$$\begin{aligned} x^{(k)} &= A^k x^{(0)} \\ &= \sum_{i=1}^{n} c_i A^k x_i \\ &= \sum_{i=1}^{n} c_i A^{k-1} \lambda_i x_i \\ &= \sum_{i=1}^{n} c_i A^{k-2} \lambda_I^2 x_i \\ &= \;\cdots \\ &= \sum_{i=1}^{n} c_i \lambda_i^k x_i \\ &= \lambda_1 [c_1 x_1 + \sum_{i=2}^{n} c_i (\frac{\lambda_i}{\lambda_1})^k x_i]. \end{aligned}$$

Because $|\lambda_1| > |\lambda_i|$ for $i = 2, \cdots, n$, it follows that the coefficients of x_i, for $u = 2, \cdots, n$,converge to zero as $k \to \inf$. Therefore, the direction of $x^{(k)}$ converges to that of x_1 which is the eigenvector we want to find.

2.3.2 Algorithm

To compute the eigen decomposition for the kernel matrix K, we have the following algorithm.

Algorithm 2 Power Iteration

```
Input: Matrix $A$;
Output: Eigenvectors $v$ corresponding to $\lambda$ and eigenvalues $\lambda$;
    Randomly initialize $v_0$;
 2: repeat
        $z = Av$;
 4:     $v = \frac{z}{\|z\|_2}$;
        $\lambda = v^T Av$;
 6: until Max iteration
    Project $A$ orthogonal to previous $v$: $A = (I - vv^T)A(I - vv^T)$
 8: Repeat previous steps to get other eigenvectors and values.
```

3 Methodology

The necessary theory and algorithm have been presented in Section 2 to realize my two goal in this paper review. In this section, I will report the procedure of analysis. In short, the main idea of the analysis is to build a set of kernels that subsume the mapping performed at each layer of the deep network and for each kernel, compute how good the representation is by measuring how many leading components of the kernel feature space are necessary in order to model the learning problem well.

3.1 Kernel's application on deep network

The kernel principal component method is applied on deep network in the following way. Let $f(x) = f_L \circ \cdots \circ f_1(x)$ be a trained deep network of L layers. Then the following "deep kernels" are defined as

$$\begin{aligned}
k_0(x, x') &= k_{RBF}(x, x') \\
k_1(x, x') &= k_{RBF}(f_1(x), f1(x')) \\
&\vdots \\
k_L(x, x') &= k_{RBF}(f_L \circ \cdots \circ f_1(x), f_L \circ \cdots \circ f_1(x')).
\end{aligned}$$

These kernels are used to subsume the mapping performed by more and more layers of the deep network. After defining "deep kernels", I use kernel PCA to find the first d kernel principal components and train multi-class logistic regression based on them. Using these trained logistic regression model, I predict the labels for each input and compute the error $e(d, \sigma)$ of the logistic regression model.

Remember, a representation is good when it is possible to build models of the learning problem on top of it that are both simple and accurate. These errors will show how accurate the representations generated by deep network are and how many kernel principal components we use will show the simplicity of representations.

The following algorithm summarizes the main computational steps for my analysis.

Algorithm 3 Main computational steps of layer-wise analysis of deep network

```
Input: Data set $(x_1, y_1), \cdots, (x_n, y_n)$
       Deep network $f : x \mapsto f_L \circ \cdots \circ f_1(x)$;
Output: The error curves $e(d)$ for each layer $l$;
    for $l \in 1, \cdots L$ do
 3:     for $\sigma \in \Sigma$ do
            $k(x, x') = k_{RBF}(f_L \circ \cdots \circ f_1(x), f_L \circ \cdots \circ f_1(x'))$;
            compute the kernel matrix $K$ associated to $k(x, x')$ and $(x_1, \cdots, x_n)$
 6:         do the eigendecomposition of $K$
            for $d \in 0, 1, 2, \cdots$ do
                build a low rank approximation of the input $\hat{U} = (u_1, \cdots, u_d)$
 9:             fit the multi-class logistic regression model that predict $(y_1, \cdots, y_n)$ from $\hat{U}$
                compute the error $e(d, \sigma)$ of the model for predicting $(y_1, \cdots, y_n)$ based on $\hat{U}$
            end for
12:     end for
        $e(d) = \min_\sigma e(d, \sigma)$
        plot the curve $e(d)$
15: end for
```

3.2 Experimental setup

3.2.1 Dataset

The test part of MINIST data set is used in my analysis. In this data set, there are 10000 samples from the large MINIST handwritten digits image classification data set. This data set consists of 10000 grayscale images of 28×28 pixels representing handwritten digits and their associated label, a number between 0 and 9.

3.2.2 Structure of deep networks

In my analysis, I consider two deep networks, convolutional neural network(CNN) and multilayer perceptron(MLP).

The MLP is built by alternating linear transformations and nonlinearities applied element-wise to the output of the linear transformations. On the MNIST-10K data set, we apply successively the functions

$$f_1(x) = sigm(w_1 \cdot x + b_1),$$
$$f_2(x) = sigm(w_2 \cdot x + b_2),$$
$$f_3(x) = softmax(\nu \cdot x),$$

where $sigm(x) = e^x/(1 + e^x)$ and $softmax(x) = e^x / \sum_j e^{x_j}$. The weight matrices w_1, w_2, ν and biases b_1, b_2 are learned from data and the size of hidden layers is set to 1600.

The convolutional neural network is built by several convolutional layers $y = w \otimes x + b$ and pooling layers which are used to subsampling each feature map by a given factor. On the MNIST-10K data set, we apply successively the functions

$$f_1(x) = pooling(sigm(w_1 \otimes x + b_1)),$$
$$f_2(x) = pooling(sigm(w_2 \otimes x + b_2)),$$
$$f_3(x) = softmax(\nu \cdot x),$$

where weight tensors w_1, w_2, weight matrix ν and biases b_1, b_2 are learned from the data. Convolution kernels have size 5×5, pooling layers downsample the input by a factor 2 and the number of feature maps in each layer is 100.

3.2.3 Training method and parameter set

Using Tensorflow to bulid CNN and MLP, I train the deep networks on the 10000 samples of the data set until a train error reaches 2.5%. I estimate the kernel principal components with the 10000 samples used for training the deep network. Therefore, the empirical estimate of the d leading kernel principal components takes the form of d 10000-dimensional vectors. $e(d)$ is obtained by fitting and evaluating the linear model with the 10000 mapped samples.

And since it is proved in the paper that the effect of the kernel scale is rather small, I go through 20 σs between 1 and 100, then calculate the test error rate for each number of d which represents the number of kernel principal component used in logistic regression model. After doing this, I will have 20 $e(d)$s for each value of d and choose the smallest one as our $e(d)$.

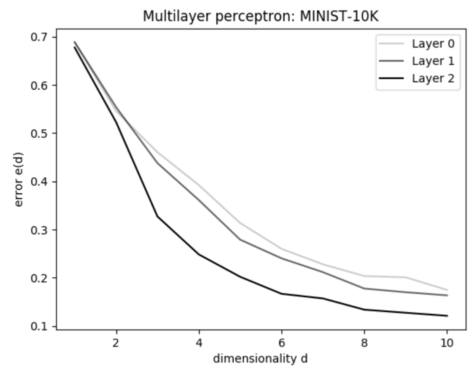
The layer of interest here are the input data($l = 0$) and the output of each layer($l = 1, 2, \cdots$)

4 Results

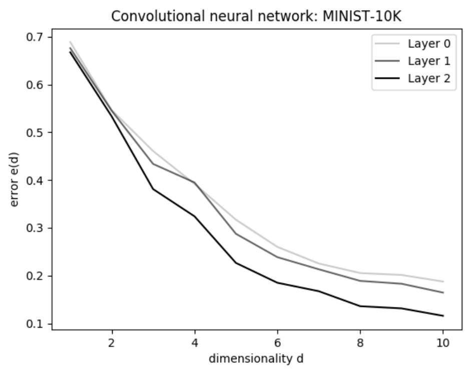
In this section, I present the results of my analysis on the evolution of the representation in deep networks. Section 4.1 will discuss the result of my first goal, how good the representations yielded by these deeper and deeper layers are. And section 4.2 will compare the evolution of the representations in different deep networks and discuss if the structure of the deep network controls how fast the representation of the task is formed layer after layer

4.1 Better representations are built layer after layer

After running the experiment like I described in previous sections, I get the follow two figures for two deep networks.



(a)



(b)

Figure 1: Layer-wise evolution of the error as a function of the number of dimensions when trained on the target task. The left one is the result of multilayer perceptron and the right one is the result of convolutional neural network. As we move from the first to the last layers, more and more data's information is captured by the representations.

We can observe in Figure 1 that as the input is propagated through more and more layers of the deep network, simpler and more accurate representations of the learning problem are obtained. These figures are very similar to those in the original paper and we can get the same conclusion that better representations are built layer after layer.

4.2 Role of the structure of deep networks

Choosing $d = 10$, I explore different deep network structures' effect on the representations' evolution trace. The result is shown in the figure below.

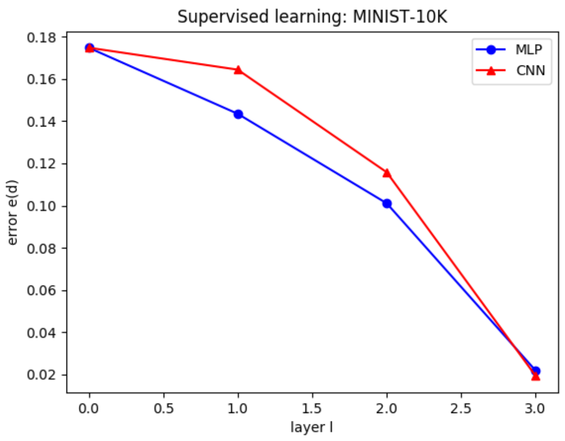


Figure 2: Layer-wise evolution of the error obtained for each training procedure for $d = 10$. The different trend of the evolution of representations for different deep networks are displayed here.

Figure 2 shows that the evolution of the representations withe respect to the learning problem when the deep network has been trained on the target task. We can observe that the evolution of the representation of the MLP follows a different trend from the representation built by the CNN. And comparing the result in original paper, we can find that the trend of CNN is similar but the trend of MLP is a little different. This difference maybe due to the different training parameters, such as learning rates, training times are used. But both the result of my own analysis and the one of original paper can lead to the same conclusion that the structure of the deep network controls how fast the representation of the task is formed layer after layer.

5 Conclusion

In this paper review, I reimplement a method for analyzing deep networks based on kernel principal component analysis in order to quantify the layer-wise evolution of the representation in deep networks. After this analysis, I get two conclusions consistent with the results in the original paper.

The first conclusion is as the network gets deeper, representations become simpler and more accurate. And the second conclusion is different deep network structures will lead to different evolution traces for the representation.

This analysis shows again that this kernel framework can be used to abstracts deep networks as a sequence of deeper and deeper kernels and to express the relation between the representation built in the deep network and the structure of the deep network.

References

[1] Gregoire Montavon, Mikio L. Braun, & Klaus-Robert Muller (2011) Kernel Analysis of Deep Networks, *Journal of Machine Learning Research* **12**:2563-2581.

[2] Convolutional Neural Networks (2018, January 27). Retrieved from `https://www.tensorflow.org/tutorials/deep_cnn`