Technical Report: Autoencoder Kernel PCA

Jeremy Cohen, Elad Hazan, Tengyu Ma

September 25, 2016

1 Introduction

Dimensionality reduction is a widely-studied problem in data analysis and machine learning, with applications to compression, de-noising, and classification. Given a sample of n data points $\{\mathbf{x}_i\}_{i=1}^n \subset \mathbb{R}^d$ drawn from some distribution $\mathcal{D}$, the goal is to learn an "encoder" $f : \mathbb{R}^d \to \mathbb{R}^k$ which preserves "informative" structure while discarding noise. What makes for informative structure? A number of popular dimensionality reduction techniques can be motivated by what might be termed the "autoencoder principle": an encoder f learns informative structure in the data if there exists a decoder $g : \mathbb{R}^k \to \mathbb{R}^d$ which can accurately reverse the encoding to "reconstruct" the original data, so that $g(f(\mathbf{x})) \approx \mathbf{x}$ for all $\mathbf{x} \sim \mathcal{D}$ [1].

Principal component analysis (PCA) is a linear dimensionality reduction technique which finds the k-dimensional subspace "closest to" the data. From an autoencoding perspective, PCA finds the linear encoder $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ that minimizes the error in reconstructing the data points from their encodings. On the one hand, PCA is simple to implement and computationally efficient: the basis vectors for this subspace are simply the eigenvectors of the empirical covariance matrix, and these can be computed via the singular value decomposition of the data matrix. On the other hand, the underlying assumption that the data lie near a low-dimensional linear subspace is quite restrictive.

Kernel PCA [6] uses some nonlinear mapping $\phi : \mathbb{R}^d \to \mathcal{F}$ to transform the data points into a higher-dimensional "feature space" $\mathcal{F}$, and then performs PCA on the transformed data points $\phi(\mathbf{x}_1) \dots \phi(\mathbf{x}_n)$. For example, with a degree-ℓ polynomial kernel, the feature space is $\mathcal{F} = \mathbb{R}^{d^\ell}$ and the feature map $\phi(\mathbf{x}) = \mathbf{x}^{\otimes \ell}$ takes the product of every group of ℓ dimensions. The so-called "kernel trick" conveniently allows us to do PCA *implicitly* in $\mathcal{F}$ without ever actually computing any $\phi(\mathbf{x}_i)$, so the resource requirements of the algorithm do not depend on $\dim(\mathcal{F})$, which can even be infinite. Kernel PCA can find non-linear structure in the data, but the catch is that it doesn't optimize an autoencoding objective. Rather than try to reconstruct the *original* data points $\mathbf{x}$ from their encoding $f(\phi(\mathbf{x}))$, kernel PCA tries to reconstruct the *feature-mapped* data points $\phi(\mathbf{x})$ from their encodings $f(\phi(\mathbf{x}))$. As a result, kernel PCA learns an encoder that retains information maximally informative of $\phi(\mathbf{x})$, whereas PCA learns an encoder that retains information maximally informative of $\mathbf{x}$ itself.

In this technical report, we present "autoencoding kernel PCA" (AKPCA), a cousin of PCA and kernel PCA which combines the best of both worlds: like kernel PCA, it can learn non-linear structure, but like PCA, its autoencoding objective tries to reconstruct the *original data points* from a lower-dimensional encoding.

The objective function of autoencoding kernel PCA is almost identical to that of kernel PCA with a degree 2 polynomial kernel, except that it penalizes deviations from $\phi(\mathbf{x}_i) = \mathbf{x}^{\otimes 2}$ in the spectral norm rather than in the Frobenius norm. This simple modification leads autoencoding kernel PCA to do a better job than kernel PCA in reconstructing the data points, but at the sacrifice of computational tractability: whereas kernel PCA can be solved exactly with an eigendecomposition, autoencoding kernel PCA is a challenging non-convex optimization problem that we can only tackle with heuristics like stochastic gradient descent. Moreover, the time and space complexity of autoencoding kernel PCA is $O(kd^2)$, which precludes its use in settings where the dimension d is large. For this reason, we also propose a "data-span" variant in which we restrict the decision variable to lie within the span of the data points mapped into feature space. This hack brings the space complexity to $O(kn)$ and the time complexity to $O(nd)$, and is therefore feasible in settings where the dimension d is large, but the number of data points n is comparatively small.

We compare autoencoding kernel PCA (and its data-span variant) to PCA and kernel PCA on a reconstruction task, a denoising task and a classification task. On the MNIST handwritten digits dataset, we find that AKPCA performs considerably better than the alternatives at reconstruction and denoising (at least when the noise level is small to moderate), while its performance on the classification task is mixed. Unfortunately, AKPCA fails outperform the alternatives on any task on either the CIFAR-10 object recognition dataset or the Labeled Faces in the Wild face dataset.

We begin this technical report with a review of PCA and kernel PCA. Then we introduce autoencoding kernel PCA and its data-span variant. Finally, we report results of experiments on MNIST.

2 Principal Components Analysis

Principal component analysis (PCA), the simplest linear dimensionality reduction algorithm, finds a subspace that lies "close" to the data. More specifically, given a set of data points $\mathbf{x}_1 \ldots \mathbf{x}_n \subset \mathbb{R}^d$ stacked into the rows of a data matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, PCA finds a k-dimensional linear subspace which minimizes the sum of the squared distances between the original data points and their projections onto the subspace.

To formulate PCA, let $\mathbf{a}_1 \ldots \mathbf{a}_k \subset \mathbb{R}^d$ be an orthonormal basis for the optimal subspace, and let $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \ldots & \mathbf{a}_k \end{bmatrix}^T$. Since the $\{\mathbf{a}_j\}_{j=1}^k$ are orthogonal, the projection $\Pi_{\mathbf{A}}(\mathbf{x})$ of any point $\mathbf{x} \in \mathbb{R}^d$ onto the subspace they span is:

$$\Pi_{\mathbf{A}}(\mathbf{x}) = \sum_{j=1}^{k} (\mathbf{a}_j^T \mathbf{x})\, \mathbf{a}_j = \mathbf{A}^T \mathbf{A} \mathbf{x}$$

So, to find the orthonormal vectors $\mathbf{a}_1 \ldots \mathbf{a}_k$ which minimize the sum of squared distances between $\mathbf{x}_i$ and $\Pi_{\mathbf{A}}(\mathbf{x}_i)$, we solve:

$$\hat{\mathbf{A}} = \operatorname*{arg\,min}_{\mathbf{A} \in \mathbb{R}^{k \times d}} \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{A}^T \mathbf{A} \mathbf{x}_i\|_2^2 \quad \text{subject to} \quad \mathbf{A}\mathbf{A}^T = \mathbf{I} \tag{1}$$

It turns out that the solution to (1) is to set $\mathbf{a}_1 \ldots \mathbf{a}_k$ to the k eigenvectors of the empirical covariance matrix $\mathbf{X}^T\mathbf{X}$ with the largest eigenvalues, or equivalently, the k right singular vectors of the data matrix $\mathbf{X}$.

Therefore, to perform PCA, one need only compute the singular value decomposition of $\mathbf{X}$ and set $\mathbf{A}$ to the k leading right singular vectors. Then the encoder is $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ and the decoder is $g(\mathbf{y}) = \mathbf{A}^T\mathbf{y}$.

2.1 Centering

PCA is often performed on centered data (i.e. where the mean of each dimension, over all data points, is zero). The steps to perform PCA on centered data are:

1. Compute the mean $\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$
2. Compute the SVD of the centered data matrix,

$$\tilde{\mathbf{X}} = \begin{bmatrix} \mathbf{x}_1 - \bar{\mathbf{x}} \\ \vdots \\ \mathbf{x}_n - \bar{\mathbf{x}} \end{bmatrix}$$

   and let $\mathbf{A}$ be the matrix whose rows are the k leading right singular vectors.

3. The encoder is $f(\mathbf{x}) = \mathbf{A}(\mathbf{x} - \bar{\mathbf{x}})$ and the decoder is $g(\mathbf{y}) = \mathbf{A}^T\mathbf{y} + \bar{\mathbf{x}}$.

3 Kernel PCA

Kernel PCA [6] is a dimensionality reduction algorithm which maps the data points to a higher-dimensional "feature space" using a non-linear mapping, and then performs PCA on the mapped data points.

3.1 Motivating Example

Suppose that we observe data in $\mathbb{R}^2$ from two different classes: red and blue. The red data lies on the circle centered at the origin with radius $\frac{1}{2}$, while the blue data lies on the circle with radius 1.

Notice that we cannot learn a linear classifier to distinguish red from blue points — the two classes are not linearly separable. Moreover, even if we use PCA to learn an encoding $\mathbb{R}^2 \to \mathbb{R}$, the classes are still not separated in the encoded data.

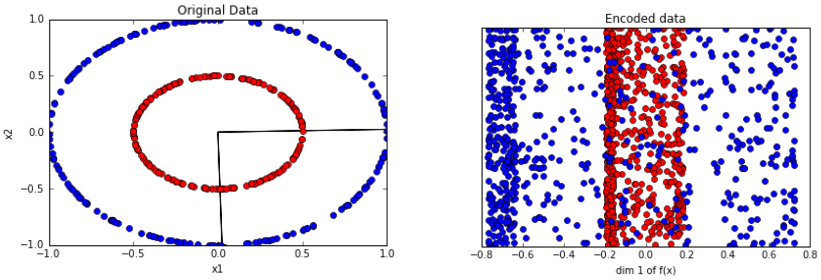

Figure 1: Original data in $\mathbb{R}^2$ (left) and the same data encoded into $\mathbb{R}$ using PCA (right).

However, suppose that we map each data point to degree-2 polynomial "feature space" with the map:

$$\phi\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} x_1^2 \\ 2x_1x_2 \\ x_2^2 \end{bmatrix}$$

In feature space, the red circle $x_1^2 + x_2^2 = \frac{1}{4}$ lies completely within the plane $\{a, b, c : a + c = \frac{1}{4}\}$, while the blue circle $x_1^2 + x_2^2 = 1$ lies within the parallel plane $\{a + c = 1\}$; hence, the two classes are separated. Sure enough, if we project the data onto the first principal component in feature space (the green arrow in figure 2, left), then we verify that the data are separated by class.

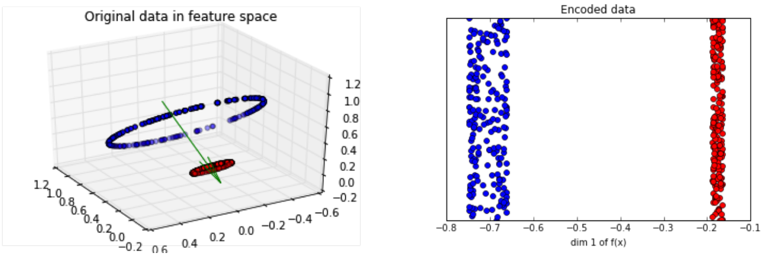

Figure 2: Data from figure 1 mapped in $\mathbb{R}^3$ "feature space" using the degree-2 polynomial map (left), and the projection of this transformed data onto the first principal component (right).

3.2 The Kernel Trick

The naive way to perform PCA on data in feature space would be to explicitly map each data point into feature space, form the $n \times \dim(\mathcal{F})$ matrix $\mathbf{\Phi} = \begin{bmatrix}\phi(\mathbf{x}_1) & \dots & \phi(\mathbf{x}_n)\end{bmatrix}^T$, and compute its top k right singular vectors.

Unfortunately, this approach scales poorly with the dimension of $\mathcal{F}$: on a dataset with 1,000 images of size 100×100, this approach would require over 370 GB of memory just to form and store the transformed data matrix $\mathbf{\Phi}$![1]

Fortunately, the so-called "kernel trick" is a shortcut that allows us to perform PCA in feature space without actually mapping the points into $\mathcal{F}$. In fact, we will see that the runtime of kernel PCA does not depend at all on $\dim(\mathcal{F})$. The "kernel trick" is as follows: for certain feature maps $\phi : \mathbb{R}^d \to \mathcal{F}$, one can compute the inner product $\phi(\mathbf{x})^T \phi(\mathbf{y})$ between two points in feature space without computing either $\phi(\mathbf{x})$ or $\phi(\mathbf{y})$. Since PCA only accesses the data points through their pairwise inner products, we can do PCA in feature space without explicitly mapping the data points there. The function $K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^T \phi(\mathbf{y})$ is called the *kernel* associated with the feature map ϕ [5].

For the degree-ℓ polynomial feature map $\phi(\mathbf{x}) = \mathbf{x}^{\otimes \ell}$, the corresponding kernel is

$$\phi(\mathbf{x})^T \phi(\mathbf{y}) = (\mathbf{x}^T \mathbf{y})^\ell \tag{2}$$

We now show that the kernel trick allows us to solve kernel PCA via the eigendecomposition of an $n \times n$ matrix.

3.3 Derivation

Let $\phi : \mathbb{R}^d \to \mathcal{F}$ be a feature map with corresponding kernel $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$.

Let $\mathbf{\Phi}$ be the matrix whose rows are the data points mapped into feature space:

$$\mathbf{\Phi} = \begin{bmatrix} \phi(\mathbf{x}_1) \\ \vdots \\ \phi(\mathbf{x}_n) \end{bmatrix}$$

To solve PCA in feature space, we need to find the k leading eigenvectors of the covariance matrix of $\mathbf{\Phi}$,

$$\mathbf{C} = \frac{1}{n} \sum_{i=1}^{n} \phi(\mathbf{x}_i)\, \phi(\mathbf{x}_i)^T \tag{3}$$

Each such eigenvector/value pair $(\mathbf{v},\, \lambda)$ will satisfy the eigenvector equation

$$\mathbf{C}\mathbf{v} = \lambda \mathbf{v}$$

It follows that for any data point $\mathbf{x}_\ell$, an eigenvector/value pair $(\mathbf{v}, \lambda)$ must also satisfy

$$\phi(\mathbf{x}_\ell)^T \mathbf{C}\mathbf{v} = \lambda\, \phi(\mathbf{x}_\ell)^T \mathbf{v} \tag{4}$$

This gives us n equations, one for each data point $\ell \in \{1 \ldots n\}$.

Now, since an eigenvector of $\mathbf{C}$ is by definition a vector $\mathbf{v}$ that satisfies $\sum_{i=1}^{n} \phi(\mathbf{x}_i)\phi(\mathbf{x}_i)^T \mathbf{v} = \lambda \mathbf{v}$, we know that any eigenvector of $\mathbf{C}$ is a linear combination of the data points in feature space:

$$\mathbf{v} = \sum_{i=1}^{n} \alpha_i\, \phi(\mathbf{x}_i) \tag{5}$$

with $\alpha_i = \frac{\phi(\mathbf{x}_i)^T \mathbf{v}}{\lambda}$.

Therefore, rather than solving for $\mathbf{v} \in \mathcal{F}$ directly, we may instead solve for the linear combination coefficients $\boldsymbol{\alpha} \in \mathbb{R}^n$.

[1] as a matrix of 32-bit floating point numbers

Plugging (5) into (4), we see that each $\boldsymbol{\alpha}$ must satisfy:

$$\phi(\mathbf{x}_\ell)^T \mathbf{C} \left(\sum_{i=1}^n \alpha_i \, \phi(\mathbf{x}_i) \right) = \lambda \, \phi(\mathbf{x}_\ell)^T \sum_{i=1}^n \alpha_i \, \phi(\mathbf{x}_i) \tag{6}$$

Plugging in the definition of $\mathbf{C}$,

$$\phi(\mathbf{x}_\ell)^T \left(\frac{1}{n} \sum_{j=1}^n \phi(\mathbf{x}_j) \, \phi(\mathbf{x}_j)^T \right) \left(\sum_{i=1}^n \alpha_i \, \phi(\mathbf{x}_i) \right) = \lambda \, \phi(\mathbf{x}_\ell)^T \sum_{i=1}^n \alpha_i \, \phi(\mathbf{x}_i) \tag{7}$$

Rearranging both sides separately,

$$\frac{1}{n} \sum_{i=1}^n \alpha_i \sum_{j=1}^n \phi(\mathbf{x}_\ell)^T \phi(\mathbf{x}_j) \, \phi(\mathbf{x}_j)^T \phi(\mathbf{x}_i) = \lambda \sum_{i=1}^n \alpha_i \, \phi(\mathbf{x}_\ell)^T \phi(\mathbf{x}_i) \tag{8}$$

The inner products in feature space may be computed using the kernel function:

$$\frac{1}{n} \sum_{i=1}^n \alpha_i \sum_{j=1}^n k(\mathbf{x}_\ell, \mathbf{x}_j) \, k(\mathbf{x}_j, \mathbf{x}_i) = \lambda \sum_{i=1}^n \alpha_i \, k(\mathbf{x}_\ell, \mathbf{x}_i) \tag{9}$$

Which may be written using the kernel matrix $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ as:

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n \alpha_i \langle \mathbf{K}_{\ell,:}, \mathbf{K}_{:,i} \rangle &= \lambda \langle \mathbf{K}_{\ell,:}, \boldsymbol{\alpha} \rangle \\ \frac{1}{n} \langle \mathbf{K}_{\ell,:}, \mathbf{K}\boldsymbol{\alpha} \rangle &= \end{aligned}$$

Since this relation must hold true for all $\ell \in \{1 \dots n\}$, we may stack these n equations on top of each other into the matrix equation:

$$\mathbf{K}\mathbf{K}\boldsymbol{\alpha} = n\lambda \mathbf{K}\boldsymbol{\alpha} \tag{10}$$

Unless $\mathbf{K}$ has a zero eigenvalue, we may pre-multiply both sides by $\mathbf{K}^{-1}$ to obtain an eigenvalue problem:

$$\mathbf{K}\boldsymbol{\alpha} = \nu \boldsymbol{\alpha} \quad \text{where} \quad \nu = n\lambda \tag{11}$$

3.4 Normalization

Finally, to ensure the uniqueness of the solution, we scale each principal component $\mathbf{v} \in \mathcal{F}$ to have unit norm. It turns out that we can do this without ever computing $\mathbf{v}$ explicitly. Let $\tilde{\boldsymbol{\alpha}}$ be a unit-norm eigenvector of $\mathbf{K}$ with eigenvalue ν. We seek a scaling $\boldsymbol{\alpha} \propto \tilde{\boldsymbol{\alpha}}$ such that $\|\mathbf{v}\| = 1$.

Since $\mathbf{v} = \sum_{i=1}^n \alpha_i \, \phi(\mathbf{x}_i) = \Phi^T \boldsymbol{\alpha}$,

$$\mathbf{v}^T \mathbf{v} = \boldsymbol{\alpha}^T \boldsymbol{\Phi} \boldsymbol{\Phi}^T \boldsymbol{\alpha} = \boldsymbol{\alpha}^T \mathbf{K} \boldsymbol{\alpha} = \nu \, \boldsymbol{\alpha}^T \boldsymbol{\alpha}$$

where the last equality is because $\boldsymbol{\alpha}$ is an eigenvector of $\mathbf{K}$, c.f. (11). Therefore, to ensure that $\|\mathbf{v}\| = 1$, it suffices to have $\|\boldsymbol{\alpha}\| = \sqrt{\frac{1}{\nu}}$. We therefore take $\boldsymbol{\alpha} = \tilde{\boldsymbol{\alpha}} / \sqrt{\nu}$.

3.5 Procedure

In summary, to perform kernel PCA on data $\mathbf{X} \in \mathbb{R}^{n \times d}$, we form the $n \times n$ kernel matrix $\mathbf{K}$, compute its k leading eigenvectors, and divide each one by the square root of the corresponding eigenvalue. Each resulting

$\boldsymbol{\alpha}_j \in \mathbb{R}^n$ are the coefficients of a kernel principal component expressed as a linear combination of the input data in feature space.

Having learned $\mathbf{A} = \begin{bmatrix}\boldsymbol{\alpha}_1 & \dots & \boldsymbol{\alpha}_k\end{bmatrix}^T$, we can encode any data point $\mathbf{x}$ without ever computing $\phi(\mathbf{x})$. The inner product of $\phi(\mathbf{x})$ with the j-th kernel principal component is

$$\begin{aligned}\phi(\mathbf{x})^T\mathbf{v}_j &= \phi(\mathbf{x})^T\left(\sum_{i=1}^{n}\alpha_i^j\,\phi(\mathbf{x}_i)\right)\\ &= \sum_{i=1}^{n}\alpha_i^j\,k(\mathbf{x},\mathbf{x}_i)\\ &= \mathbf{k}^T\boldsymbol{\alpha}^j \quad \text{where} \quad k_i = k(\mathbf{x},\mathbf{x}_i)\end{aligned}$$

Therefore,

$$f(\mathbf{x}) = \mathbf{A}\mathbf{k} \quad \text{where} \quad k_i = k(\mathbf{x},\mathbf{x}_i) \tag{12}$$

For the degree-ℓ polynomial kernel, it takes $O(n^2d)$ time to compute the kernel matrix and $O(n^2k)$ time to compute its k leading eigenvectors and eigenvalues. Therefore, the overall runtime is $O(n^2d)$, since when doing dimensionality reduction we have $k < d$.

3.6 Centering

Recall that PCA is often performed on centered data. It turns out that there is a way to perform kernel PCA on data that is centered in feature space without ever explicitly mapping the data to feature space, much less computing the mean of the data there. Let $\bar{\phi}(\mathbf{x})$ be a "centered" data point in feature space:

$$\tilde{\phi}(\mathbf{x}) = \phi(\mathbf{x}) - \frac{1}{n}\sum_{j=1}^{n}\phi(\mathbf{x}_j)$$

and let $\tilde{\mathbf{K}}$ be the kernel matrix of the centered data, i.e. $\tilde{K}_{ij} = \tilde{\phi}(\mathbf{x}_i)^T\tilde{\phi}(\mathbf{x}_j)$ Simple algebra can show that the centered kernel matrix may be computed as:

$$\tilde{\mathbf{K}} = \mathbf{K} - \mathbf{1}_n\mathbf{K} - \mathbf{K}\mathbf{1}_n + \mathbf{1}_n\mathbf{K}\mathbf{1}_n \quad \text{where} \quad (\mathbf{1}_n)_{ij} = \frac{1}{n}$$

Having computed the centered kernel matrix, kernel PCA on centered data entails solving the eigenvector problem:

$$\tilde{\mathbf{K}}\boldsymbol{\alpha} = \nu\boldsymbol{\alpha}$$

3.7 Nystrom Approximation

Both the space and time complexity of kernel PCA scale quadratically with the number of examples n. For large datasets, we can instead use the Nystrom method [7] to solve kernel PCA approximately. A sample of m data points is selected uniformly without replacement and the k leading eigenvectors and eigenvalues $(\hat{\mathbf{v}}_1, \hat{\lambda}_1)\dots(\hat{\mathbf{v}}_k, \hat{\lambda}_k)$ of the resulting $m\times m$ kernel matrix are computed. Then the eigenvectors and eigenvalues of the full kernel matrix are approximated as:

$$\mathbf{v}_j \approx \frac{\sqrt{m}}{n\,\hat{\lambda}_j}\mathbf{K}\hat{\mathbf{v}}_j \tag{13}$$

$$\lambda_j \approx \frac{n}{m}\hat{\lambda}_j \tag{14}$$

where $\mathbf{K}$ is the $n \times m$ kernel matrix between the full data set and the sample.

4 Autoencoding Kernel PCA

Kernel PCA is a richer model than PCA. However, PCA's objective function has an easy autoencoding interpretation: compress $\mathbf{x}$ into a lower-dimensional representation and then try to reconstruct $\mathbf{x}$.

$$\operatorname*{arg\,min}_{(f,g)\in\{f(\mathbf{x})=\mathbf{A}\mathbf{x},\, g(\mathbf{x})=\mathbf{A}^T\mathbf{x}\}} \sum_{i=1}^{n} \|g(f(\mathbf{x}_i)) - \mathbf{x}_i\|_2^2 \qquad \text{(PCA objective)}$$

In contrast, kernel PCA's objective function compresses $\phi(\mathbf{x})$ into a lower-dimensional representation, and then tries to reconstruct $\phi(\mathbf{x})$.

$$\operatorname*{arg\,min}_{(f,g)\in\{f(\mathbf{x})=\mathbf{A}\mathbf{x},\, g(\mathbf{x})=\mathbf{A}^T\mathbf{x}\}} \sum_{i=1}^{n} \|g(f(\phi(\mathbf{x})_i)) - \phi(\mathbf{x}_i)\|_2^2 \qquad \text{(kernel PCA objective)}$$

We would expect PCA to learn an encoding function that is maximally informative of $\mathbf{x}$ itself, whereas we would expect kernel PCA to learn an encoding function that is maximally informative of the degree-2 polynomial expansion $\phi(\mathbf{x})$.

In this section, we introduce "autoencoding kernel PCA" (AKPCA), a hybrid method which compresses $\phi(\mathbf{x})$ into a lower-dimensional representation, and then tries to reconstruct $\mathbf{x}$.

We will derive the AKPCA objective function from the same "autoencoder" principle as PCA: choose parameters of an encoder f and a decoder g so as to minimize the empirical reconstruction error,

$$\sum_{i=1}^{n} \|g(f(\mathbf{x}_i)) - \mathbf{x}_i\|_2^2$$

Our encoder will take the same form as in kernel PCA:

$$f(\mathbf{x}) = \mathbf{A}\phi(\mathbf{x})$$

parameterized by some matrix $\mathbf{A} \in \mathbb{R}^{k\times d^2}$.

This raises the question: what is the natural decoder g to decode kernel PCA encodings? This is known as the "pre-image problem" in the kernel methods literature [2, 4], and it turns out that for the special case of a degree-2 polynomial kernel, the natural decoder involves solving an eigenvalue problem.

Given an encoding $f(\mathbf{x}) = \mathbf{A}\phi(\mathbf{x}) = \mathbf{y} \in \mathbb{R}^k$, how can we recover some $\hat{\mathbf{x}}$ such that $f(\hat{\mathbf{x}}) = \mathbf{A}\phi(\hat{\mathbf{x}}) \approx \mathbf{y}$? A natural way to start would be to premultiply $\mathbf{y}$ by $\mathbf{A}$, since the kernel PCA objective function guarantees that $\mathbf{A}^T\mathbf{y} = \mathbf{A}^T\mathbf{A}\phi(\mathbf{x})$ is close to $\phi(\mathbf{x})$. But then how do we get from $\mathbf{A}^T\mathbf{y} \approx \phi(\mathbf{x})$ to $\hat{\mathbf{x}} \approx \mathbf{x}$? We solve for the $\hat{\mathbf{x}}$ such that $\phi(\hat{\mathbf{x}})$ is closest to $\mathbf{z} := \mathbf{A}^T\mathbf{y}$:

$$\begin{aligned} \hat{\mathbf{x}} &= \operatorname*{arg\,min}_{\mathbf{x}\in\mathbb{R}^d} \|\phi(\mathbf{x}) - \mathbf{z}\|_2 \\ &= \operatorname*{arg\,min}_{\mathbf{x}\in\mathbb{R}^d} \|\mathbf{x}\mathbf{x}^T - \mathcal{M}(\mathbf{z})\|_F \end{aligned}$$

By the Eckert-Young theorem, the best rank-1 approximation to $\mathcal{M}(\mathbf{y})$ is given by $\lambda_1\mathbf{v}_1\mathbf{v}_1^T$, where $(\mathbf{v}_1, \lambda_1)$ are the leading eigenvector and eigenvalue of $\mathcal{M}(\mathbf{y})$. Therefore, the solution is $\hat{\mathbf{x}} = \sqrt{\lambda_1}\mathbf{v}_1$ if $\lambda_1 > 0$, or $\hat{\mathbf{x}} = \mathbf{0}$ if $\lambda_1 \leq 0$.

Therfore, our decoder $g : \mathbb{R}^k \to \mathbb{R}^d$ should be defined as

$$g(\mathbf{y}) = \max(\sqrt{\lambda_1}, 0)\mathbf{v}_1 \quad \text{where} \quad \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T = \mathcal{M}(\mathbf{A}^T\mathbf{y})$$

Composing together the decoder and encoder, the natural autoencoding loss function is:

$$\operatorname*{arg\,min}_{\mathbf{A}\in\mathbb{R}^{k\times d^2}} \sum_{i=1}^{n} \|\max(\sqrt{\lambda_1}, 0)\, \mathbf{v}_1(\mathcal{M}(\mathbf{A}^T\mathbf{A}\phi(\mathbf{x}))) - \mathbf{x}_i\|_2^2$$

where we abuse notation by letting λ_1 and $\mathbf{v}_1$ denote the largest eigenpair of the matrix $\mathcal{M}(\mathbf{A}^T\mathbf{A}\phi(\mathbf{x}))$.

To make the model more flexible, we remove the restriction that the decoding matrix is the same as the encoding matrix, and optimize instead over two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{k\times d^2}$:

$$\operatorname*{arg\,min}_{\mathbf{A},\mathbf{B}\in\mathbb{R}^{k\times d^2}} \sum_{i=1}^{n} \| \max(\sqrt{\lambda_1}, 0)\, \mathbf{v}_1(\mathcal{M}(\mathbf{B}^T\mathbf{A}\phi(\mathbf{x}))) - \mathbf{x}_i\|_2^2 \tag{15}$$

It is not immediately clear how to optimize the objective function (15) — or even how to compute its gradient! Therefore, we will instead optimize a surrogate objective function which comes out of the following inequality[2]: for any $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{M} \in \mathbb{R}^{d\times d}$,

$$\| \max(\sqrt{\lambda_1}, 0)\, \mathbf{v}_1(\mathbf{M}) - \mathbf{x}\|_2^2 \leq c\|\mathbf{M} - \mathbf{x}\mathbf{x}^T\|_2 \tag{16}$$

for some small constant c, where $\|\cdot\|_2$ on the RHS of (16) denotes the spectral norm of a matrix.

By employing inequality (16) with $\mathbf{M} = \mathcal{M}(\mathbf{B}^T\mathbf{A}\phi(\mathbf{x}))$, we can see that the following objective function is an upper bound on (15) to within a constant factor of c:

$$\operatorname*{arg\,min}_{\mathbf{A},\mathbf{B}\in\mathbb{R}^{k\times d^2}} \sum_{i=1}^{n} \|\mathcal{M}(\mathbf{B}\mathbf{A}\phi(\mathbf{x}_i) - \phi(\mathbf{x}_i))\|_2 \tag{17}$$

An alternative way to write the objective function (17) is:

$$\operatorname*{arg\,min}_{\mathbf{A},\mathbf{B}\in\mathbb{R}^{k\times d^2}} \sum_{i=1}^{n} \left\| \sum_{j=1}^{k} (\mathbf{x}_i^T\mathbf{A}_j\mathbf{x}_i)\mathbf{B}_j - \mathbf{x}_i\mathbf{x}_i^T \right\|_2 \tag{18}$$

where $\mathbf{A} = \begin{bmatrix}\mathrm{vec}(\mathbf{A}_1) & \ldots & \mathrm{vec}(\mathbf{A}_k)\end{bmatrix}^T$ and $\mathbf{B} = \begin{bmatrix}\mathrm{vec}(\mathbf{B}_1) & \ldots & \mathrm{vec}(\mathbf{B}_k)\end{bmatrix}^T$.

The AKPCA objective function (18) is *almost* identical to the KPCA objective function (with a degree-2 polynomial kernel), but not quite: both methods try to choose $\{\mathbf{A}_j\}_{j=1}^k$ and $\{\mathbf{B}_j\}_{j=1}^k$ so as to approximate $\mathbf{x}\mathbf{x}^T$ with $\sum_k(\mathbf{x}^T\mathbf{A}_k\mathbf{x})\mathbf{B}_k$. The only difference is that AKPCA penalizes the residual $\mathbf{x}\mathbf{x}^T - \sum_k(\mathbf{x}^T\mathbf{A}_k\mathbf{x})\mathbf{B}_k$ in spectral norm, while KPCA does so in Frobenius norm. Intuitively, reconstructing $\mathbf{x}\mathbf{x}^T$ to high accuracy in the spectral norm implies one can reconstruct $\mathbf{x}$ to high accuracy in ℓ_2 norm.

4.1 Optimization

The AKPCA objective function (18) is not convex, so we cannot hope to find a global optimizer. Nevertheless, we found that running stochastic (sub)gradient descent or SVRG [3] from a random starting point usually yielded a decent solution.

In what follows, we compute a subgradient of the AKPCA objective function

$$\begin{aligned} f(\mathbf{A},\mathbf{B}) &= \left\| \sum_{j=1}^{k} \mathbf{x}^T\mathbf{A}_j\mathbf{x}\mathbf{B}_j - \mathbf{x}\mathbf{x}^T \right\|_2 \\ &= \|\mathbf{M}\|_2 \quad \text{where} \quad \mathbf{M} = \sum_{j=1}^{k} \mathbf{x}^T\mathbf{A}_j\mathbf{x}\mathbf{B}_j - \mathbf{x}\mathbf{x}^T \end{aligned}$$

The spectral norm is non-differentiable, so f does not have a gradient. Nevertheless, a subgradient of $f(\mathbf{M}) = \|\mathbf{M}\|_2$ is $\mathbf{u}\mathbf{v}^T$, where $\mathbf{u}$ and $\mathbf{v}$ are the leading left and right singular vectors of $\mathbf{M}$.

Derivative w.r.t. $\mathbf{A}_j$

[2] We have not proved this inequality, but numerical simulations lead us to suspect that it is true.

The derivative of $\mathbf{M}$ with respect to element (r,s) of $\mathbf{A}_j$ is $x_r x_s \mathbf{B}_j$. Therefore, by the chain rule,

$$\frac{\partial f}{\partial A_{j,rs}} = \mathrm{Tr}(\mathbf{v}\mathbf{u}^T x_r x_s \mathbf{B}_j) = x_r x_s \mathbf{u}^T \mathbf{B}_j \mathbf{v}$$

and

$$\frac{\partial f}{\partial \mathbf{A}_j} = (\mathbf{u}^T \mathbf{B}_j \mathbf{v})\mathbf{x}\mathbf{x}^T \tag{19}$$

Derivative w.r.t. $\mathbf{B}_j$

Similarly, the derivative of $\mathbf{M}$ with respect to element (r,s) of $\mathbf{B}_j$ has $\mathbf{x}^T \mathbf{A}_j \mathbf{x}$ at element (r,s) and zeros everywhere else. Therefore, by the chain rule,

$$\frac{\partial f}{\partial B_{j,rs}} = \mathbf{x}^T \mathbf{A}_j \mathbf{x}\, (\mathbf{u}\mathbf{v}^T)_{rs}$$

and

$$\frac{\partial f}{\partial \mathbf{B}_j} = (\mathbf{x}^T \mathbf{A}_j \mathbf{x})\, \mathbf{u}\mathbf{v}^T \tag{20}$$

The time and space complexity of AKPCA scale quadratically with the dimension d of the data: it takes $O(kd^2)$ memory to store $\mathbf{A}$ and $\mathbf{B}$, and computing a stochastic subgradient estimator over a single data point takes $O(kd^2)$ time (see (19) and (20)). This quadratic dependence on d renders AKPCA infeasible for high-dimensional data. In the following section, we will present a "hack" that cuts the memory requirement to $O(kn)$ and the time complexity of computing a stochastic gradient estimator to $O(nd)$ — at the cost of reducing the modeling power.

5 Data-span Autoencoding KPCA

Recall that even though kernel PCA was defined as an optimization problem in $\dim(\mathcal{F})$ dimensions, it boiled down to an eigenvector problem in an $n \times n$ matrix because of the crucial fact that the eigenvectors of the covariance matrix of the data in feature space (i.e. the rows of $\mathbf{A}$) could be exactly represented as linear combinations of the data points in feature space.

The same fact does not hold true for the AKPCA objective function: the rows of $\mathbf{A}$ and $\mathbf{B}$ at the optimal solution do not necessarily lie within the span of the data points in feature space. Nevertheless, as a "hack" to render AKPCA tractable for datasets with large dimensionality d but only small or moderate number of datapoints n, we may pretend as if this were true, and restrict each row of $\mathbf{A}$ to be a linear combination of the data points in feature space. Rather than optimize over the d^2 entries of each row of $\mathbf{A}$ directly, we optimize over the n coefficients of the linear combination. This "data-span" variant of AKPCA requires $O(kn)$ memory for storage and takes $O(nd)$ time to compute a stochastic gradient estimator.

5.1 Derivation

Let h_{ij} be the contribution of data point i to row j of $\mathbf{A}$, so that:

$$\mathbf{A}_{(j,:)} = \sum_{i=1}^{n} h_{ij}\, \mathbf{x}_i$$

Define g_{ij} in the same way for $\mathbf{B}$.

Let $\mathbf{h}_i = \begin{bmatrix} h_{i1} & \dots & h_{ik} \end{bmatrix}^T$ and $\mathbf{g}_i = \begin{bmatrix} g_{i1} & \dots & g_{ik} \end{bmatrix}^T$ be the contributions of data point i to all rows of $\mathbf{A}$ and $\mathbf{B}$, respectively.

Let $\mathbf{H} = \begin{bmatrix} \mathbf{h}_1 \dots \mathbf{h}_n \end{bmatrix} \in \mathbb{R}^{k \times n}$ and $\mathbf{G} = \begin{bmatrix} \mathbf{g}_1 \dots \mathbf{g}_n \end{bmatrix} \in \mathbb{R}^{k \times n}$, so that the AKPCA parameters written as linear combinations of the data points in feature space are:

$$\mathbf{A} = \mathbf{H}\mathbf{\Phi} \tag{21}$$
$$\mathbf{B} = \mathbf{G}\mathbf{\Phi} \tag{22}$$

Under this new "data-span" parameterization, the term of the AKPCA objective function (17) for a data point $\mathbf{x}$ is:

$$\begin{aligned} f(\mathbf{H},\mathbf{G}) &= \left\| \mathcal{M}(\mathbf{B}^T\mathbf{A}\phi(\mathbf{x})) - \mathbf{x}\mathbf{x}^T \right\|_2 \\ &= \|\mathcal{M}(\mathbf{\Phi}^T\mathbf{G}^T\mathbf{H}\mathbf{\Phi}\phi(\mathbf{x})) - \mathbf{x}\mathbf{x}^T\|_2 && \text{(by (21) and (22))} \\ &= \|\mathcal{M}(\mathbf{\Phi}^T\mathbf{G}^T\mathbf{H}\mathbf{k}) - \mathbf{x}\mathbf{x}^T\|_2 && \text{(by the kernel trick)} \\ &= \left\| \sum_{i=1}^{n} (\mathbf{G}^T\mathbf{H}\mathbf{k})_i \mathbf{x}_i\mathbf{x}_i^T - \mathbf{x}\mathbf{x}^T \right\|_2 && (23) \\ &= \|\mathbf{M}\|_2 \quad \text{where} \quad \mathbf{M} = \sum_{i=1}^{n} z_i\mathbf{x}_i\mathbf{x}_i^T - \mathbf{x}\mathbf{x}^T \quad \text{and} \quad \mathbf{z} = \mathbf{G}^T\mathbf{H}\mathbf{k} \end{aligned}$$

where $\mathbf{k} \in \mathbb{R}^n$ denotes the vector obtained by applying the kernel function between $\mathbf{x}$ and each of $\mathbf{x}_1 \dots \mathbf{x}_n$:

$$k_i = \phi(\mathbf{x})^T\phi(\mathbf{x}_i) = (\mathbf{x}^T\mathbf{x}_i)^2$$

5.2 Optimization

Even though the data-span AKPCA objective function (23) is not convex, we were able to attain a reasonable solution using stochastic gradient descent initialized from a random point. In what follows, we compute a subgradient of (23).

The derivative wrt z:

Since $\mathbf{M}$ is symmetric, the derivative of f w.r.t $\mathbf{M}$ is:

$$\frac{\partial f}{\partial \mathbf{M}} = \mathbf{v}\mathbf{v}^T$$

where $\mathbf{v}$ is the leading eigenvector of $\mathbf{M}$.

Moreover, from the definition of $\mathbf{M}$ it is clear that

$$\frac{\partial \mathbf{M}}{\partial z_i} = \mathbf{x}_i\mathbf{x}_i^T$$

Therefore, by the chain rule,

$$\begin{aligned} \frac{\partial f}{\partial z_i} &= \sum_{\text{elements of } \mathbf{M}} \frac{\partial f}{\partial \mathbf{M}} \frac{\partial \mathbf{M}}{\partial \mathbf{z}_i} \\ &= \mathrm{Tr}((\mathbf{v}\mathbf{v}^T)^T(\mathbf{x}_i\mathbf{x}_i^T)) \\ &= (\mathbf{x}_i^T\mathbf{v})^2 \end{aligned}$$

Equivalently, in vector form,

$$\frac{\partial f}{\partial \mathbf{z}} = (\mathbf{X}\mathbf{v})^2 \quad \text{where} \quad (\cdot)^2 \text{ is the element-wise square} \tag{24}$$

The derivative wrt $\mathbf{G}$:

Since $\mathbf{z} = \mathbf{G}^T\mathbf{H}\mathbf{k}$, the ith element is given by $z_i = \mathbf{g}_i^T\mathbf{H}\mathbf{k}$. Therefore, the derivative of z_i w.r.t. the ith column of $\mathbf{G}$ is $\frac{\partial z_i}{\partial \mathbf{g}_i} = \mathbf{H}\mathbf{k}$. Therefore, by the chain rule, the derivative of f w.r.t. the ith column of $\mathbf{G}$ is:

$$\frac{\partial f}{\partial \mathbf{g}_i} = \frac{\partial f}{\partial z_i}\frac{\partial z_i}{\partial \mathbf{g}_i} = \frac{\partial f}{\partial z_i}\mathbf{H}\mathbf{k}$$

Equivalently, in matrix form,

$$\frac{\partial f}{\partial \mathbf{G}} = \mathbf{H}\mathbf{k}\frac{\partial f}{\partial \mathbf{z}}^T \tag{25}$$

The derivative wrt $\mathbf{H}$:

Since $z_i = \mathbf{g}_i^T\mathbf{H}\mathbf{k}$, we have:

$$\frac{\partial z_i}{\partial \mathbf{H}} = \mathbf{g}_i\mathbf{k}^T$$

Therefore, by the chain rule:

$$\begin{aligned}\frac{\partial f}{\partial \mathbf{H}} &= \sum_{i=1}^{n}\frac{\partial f}{\partial z_i}\frac{\partial z_i}{\partial \mathbf{H}} \\ &= \sum_{i=1}^{n}\frac{\partial f}{\partial z_i}\left(\mathbf{g}_i\mathbf{k}^T\right) \\ &= \left(\sum_{i=1}^{n}\frac{\partial f}{\partial z_i}\mathbf{g}_i\right)\mathbf{k}^T \\ &= \mathbf{G}\frac{\partial f}{\partial \mathbf{z}}\mathbf{k}^T \end{aligned} \tag{26}$$

5.3 Efficiently computing the gradient

It turns out that we can evaluate the DS-AKPCA objective function (23) and compute its subgradient in $O(dn)$ time. How could this be? After all, forming $\mathbf{M}$ explicitly in memory would take $O(d^2)$ space and $O(nd^2)$ time!

The trick is to use the Lanczos algorithm to compute the leading eivenvector and eigenvalue of $\mathbf{M}$ without ever forming $\mathbf{M}$ explicitly. The Lanczos algorithm only accesses $\mathbf{M}$ only through matrix-vector products $\mathbf{M}\mathbf{w}$; Lanczos requires $O(\frac{1}{\epsilon})$ of these matrix-vector multiplications to converge to an ϵ-approximate solution.

The key matrix-vector multiplication takes the form:

$$\mathbf{M}\mathbf{w} = \sum_{i=1}^{n} z_i\mathbf{x}_i\mathbf{x}_i^T\mathbf{w} - \mathbf{x}\mathbf{x}^T\mathbf{w}$$

which can be evaluated in $O(nd)$ time.

Of course, we first need to compute $\mathbf{z} = \mathbf{G}^T\mathbf{H}\mathbf{k}$ in advance. This takes $O(nd)$ time to compute $\mathbf{k}$, followed by $O(nk)$ time to compute $\mathbf{z}$.

Therefore, in total, computing the leading eigenvector and eigenvalue of $\mathbf{M}$ takes $O(nk + nd + \frac{nd}{\epsilon})$ time, where ϵ is the tolerance given to the Lanzcos algorithm.

Once we have the leading eigenvector, computing the subgradient of f wrt $\mathbf{G}$ and $\mathbf{H}$ takes $O(n(d + k)$ time.

Since for dimensionality reduction the number of components k is less than the dimensionality of the data d, the whole thing takes $O(nd(1 + \frac{1}{\epsilon}))$ time.

6 Experiments

We compared our two algorithms, AKPCA and DS-AKPCA, to PCA and kernel PCA (with a degree-2 polynomial kernel) on the MNIST handwritten digits dataset. MNIST consists of 70,000 examples, each a 28x28 grayscale image of a digit with an integer label of 0 through 9.

We compared these four algorithms on several metrics:

1. Reconstruction error on the training set: $\sum_{i \in \text{train}} \|\mathbf{x}_i - g(f(\mathbf{x}_i))\|_2^2$, a measure of how large/flexible the model's hypothesis class is, and, for those algorithms that rely on non-convex optimization, a measure of how well the optimization succeeded.
2. Reconstruction error on the held-out set: $\sum_{i \notin \text{train}} \|\mathbf{x}_i - g(f(\mathbf{x}_i))\|_2^2$, a measure of how well the encoder/decoder learned on the training set generalizes to held-out data.
3. Classification accuracy: take the encoder f learned on the training set and use it to encode all data points in the held-out set. Then use a portion of the encoded held-out set (the "classification training set") to train a linear classifier $h : \mathbb{R}^k \to \{0, \ldots, 9\}$, and measure the accuracy of h in predicting the class of the remainder of the encoded held-out set (the "classification testing set").
4. Denoising error: take an image $\mathbf{x}$ from the held-out set, "corrupt" it to make $\tilde{\mathbf{x}}$, denoise the corrupted image with $\hat{\mathbf{x}} = g(f(\tilde{\mathbf{x}}))$, and measure the difference $\|\hat{\mathbf{x}} - \tilde{\mathbf{x}}\|_2^2$. We use two kinds of noise:
   (a) Gaussian noise:
   $$\tilde{\mathbf{x}} = \mathbf{x} + \boldsymbol{\epsilon} \qquad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$
   (b) Speckle noise:
   $$(\tilde{\mathbf{x}})_i = \begin{cases} (\mathbf{x})_i & \text{w.p.} \quad 1-p \\ 0 & \text{w.p.} \quad p \end{cases}$$

We ran all of these experiments on three versions of the MNIST dataset:

1. a "short" 28x28 version consisting of 300 examples from each class (3,000 in total).
2. a "short" 14x14 version consisting of 300 examples from each class (3,000 in total). We downsampled each example from 28x28 to 14x14 by extracting one out of every four pixels.
3. a "long" 14x14 version consisting of 3,500 examples from each class (35,000 in total).

Ideally, we would have just used a single dataset: a "long" 28x28 dataset. However, the dimension $d = 28^2$ is too large to run AKPCA, and the size $n = 35,000$ is too large to run DS-AKPCA.

We scaled each dataset by the norm of the example with maximum norm:

$$\mathbf{x}_i \leftarrow \frac{\mathbf{x}_i}{\max_j \|\mathbf{x}_j\|}$$

After this scaling, all examples had norm in $[0, 1]$.

AKPCA and DS-AKPCA require an initial step size and a step size decay schedule as hyperparameters. We set these by hand by trying a range of values and choosing the ones that led to the quickest decrease in the loss function. We found that the performance of these two algorithms was highly sensitive to the initial step size.

To apply kernel PCA to the long dataset, we used the Nystrom approximation with a sample size of $m = 15,000$.

We ran each of the four algorithms to find $k = \{5, 10, 15, 20, 25\}$ components.

6.0.1 Reconstuction Error on Training Set

For all three datasets, PCA attained a lower reconstruction error than did kernel PCA. Our algorithms AKPCA and DS-AKPCA did better still, with AKPCA the best.

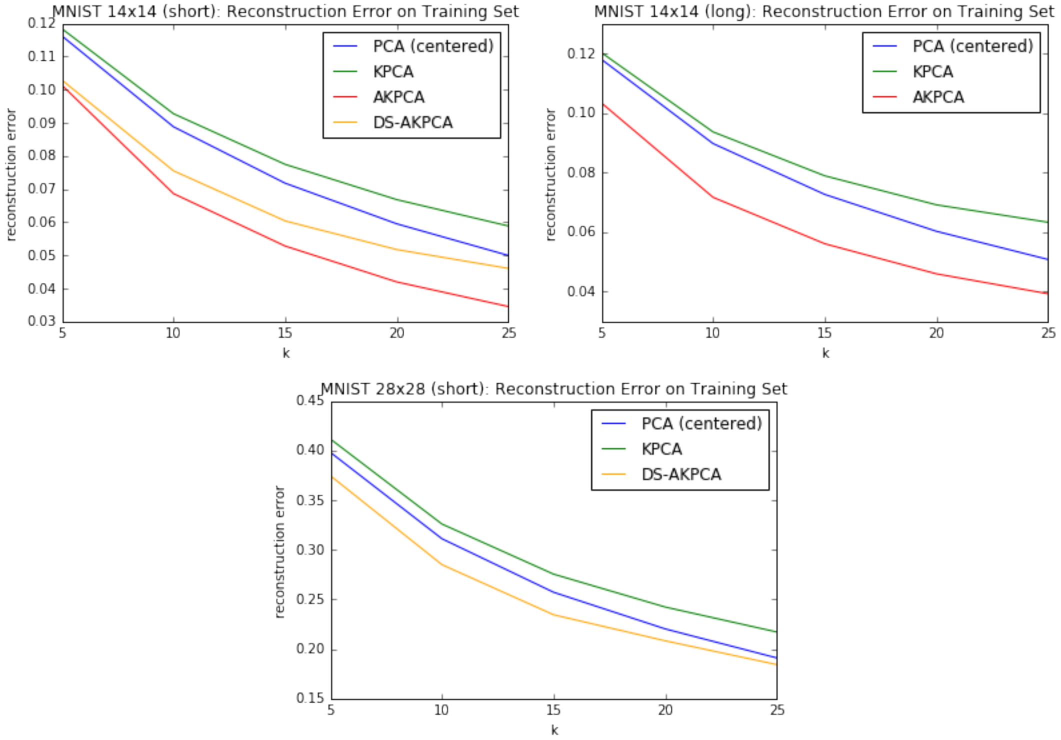

Figure 3: Average reconstruction error on the training set plotted as a function of number of components. Top left: 14x14, short training set. Top right: 14x14, long training set. Bottom: 28x28, short training set.

Across all k, PCA with centering attained a lower reconstruction error than PCA without centering, so we do not report the performance of latter at all.

That DS-AKPCA did not perform as well as AKPCA suggests that the "data-span" assumption is incorrect: at the optimal solution, the rows of **A** and **B** do not lie within the span of the data points in feature space.

6.0.2 Reconstuction Error on Held-Out Set

The relative performance of all four algorithms on the training set remained the same on the held-out set, implying that none of the algorithms overfit too much.

Figure 5 compares reconstruction error on the training set and the held-out set for each algorithm. The difference between the training and held-out reconstruction errors is a measure of how much the algorithm overfit to the training set. As expected, overfitting is much greater with a short training set (left) than with a long training set (right). In the small training set regime, AKPCA appears to overfit a bit more than the other three algorithms.

Figure 6 illustrates the effect of training set size on generalization. Across all algorithms, the larger the training set, the lower the reconstruction error on held-out data.

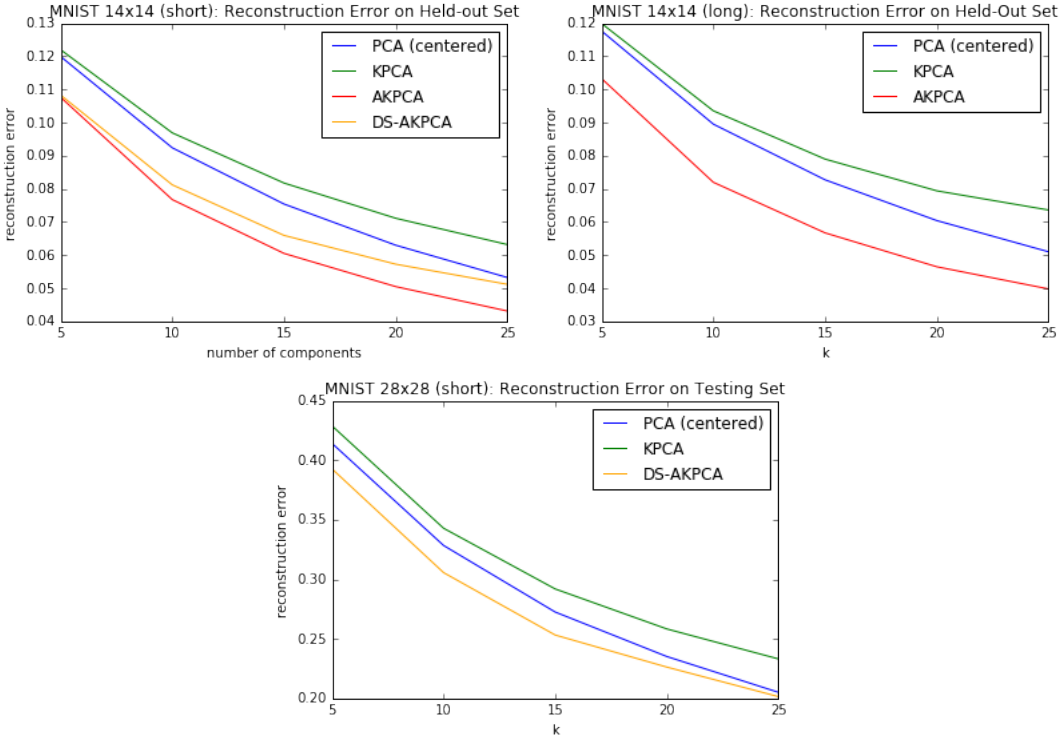

Figure 4: Average reconstruction error on the held-out set, plotted as a function of number of components. Top left: 14x14, short training set. Top right: 14x14, long training set. Bottom: 28x28, short training set.

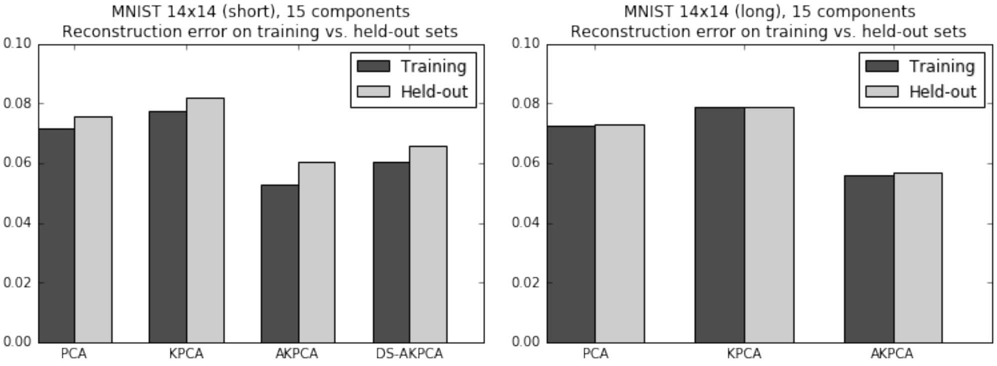

Figure 5: Average reconstruction error on the training vs. held-out set. Left: short training set (300 examples per class). Right: long training set (3,500 examples per class).

6.0.3 Denoising: Gaussian Noise

In the first denoising experiment, we pick an example $\mathbf{x}$ from the held-out set, add Gaussian noise $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$, and try to denoise $\mathbf{x} + \boldsymbol{\epsilon}$ by encoding with f and then decoding with g. We then measure the difference between the denoised version and the original example:

$$\text{denoising error} = \frac{1}{n} \sum_{i=1}^{n} \|g(f(\mathbf{x}_i + \boldsymbol{\epsilon}_i)) - \mathbf{x}_i\|_2^2 \quad \text{where } \{\mathbf{x}_i\}_{i=1}^{n} \text{ are the held-out data}$$

Figure 7 shows several MNIST digits corrupted with increasing levels of Gaussian noise.

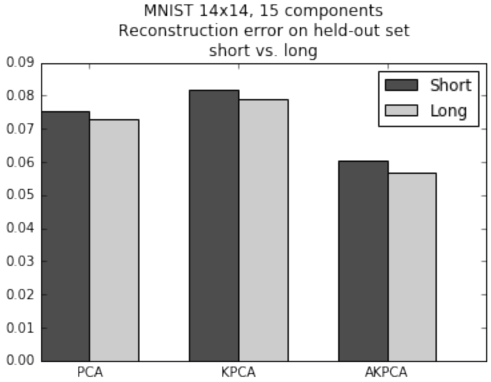

Figure 6: Comparing reconstruction error on held-out data for a model trained on a "short" training set (300 examples per class) vs. a "long" training set (1750 examples per class). Across all algorithms, the model trained on a longer dataset exhibited lower (better) reconstruction error on held-out data.

The performance of all four algorithms on this denoising task is plotted in figure 8.

Figure 8 shows the performance of all four algorithms at denoising MNIST digits corrupted by Gaussian noise. The results were consistent across all three datasets, and across a model order of both 10 (top) and 20 (bottom). Our algorithm AKPCA always performed the best at denoising so long as the standard deviation of the noise was small to moderately large. As is clear from figure 7, a noise level of $\sigma = 0.04$ (third column from the left) is still a considerable amount of noise.

Additionally, the data-span variant of AKPCA outperformed PCA and KPCA in the high noise regime (though it did not perform as well as the standard AKPCA).

For Gaussian noise with $\sigma > 0.04$, kernel PCA generally gave the best denoising results.

6.0.4 Denoising: Speckle Noise

In the second denoising experiment, we pick a sample $\mathbf{x}$ from the held-out set, corrupt it with "speckle" noise [4] by independently setting each pixel to 0 with probability p, and try to denoise the corrupted digit by encoding and then decoding.

Figure 9 shows three MNIST digits corrupted by increasing levels of "speckle" noise.

Figure 10 shows the denoising error of all four algorithms at denoising digits corrupted by speckle noise, for models of 10 (left) and 20 (right) components. For the 14x14 datasets, our algorithms, AKPCA and DS-AKPCA, outperform PCA and kernel PCA under all conditions studied. For the 28x28 dataset, DS-AKPCA outperforms PCA and KPCA so long as the noise is low to moderate.

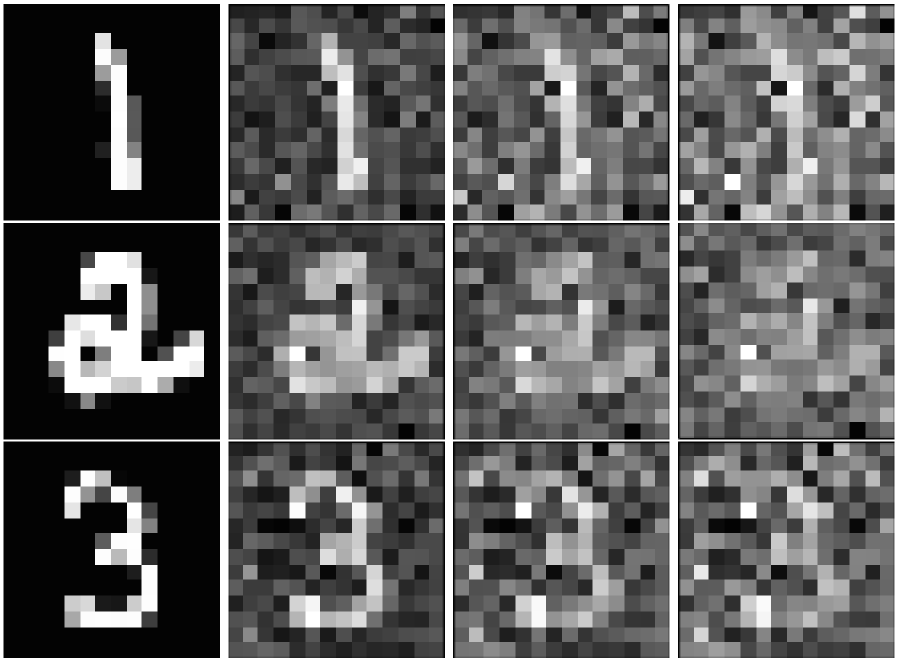

Figure 7: Three MNIST digits corrupted by increasing levels of Gaussian noise. From left to right, the standard devaiation of the noise is: $\{0, 0.02, 0.04, 0.06\}$.

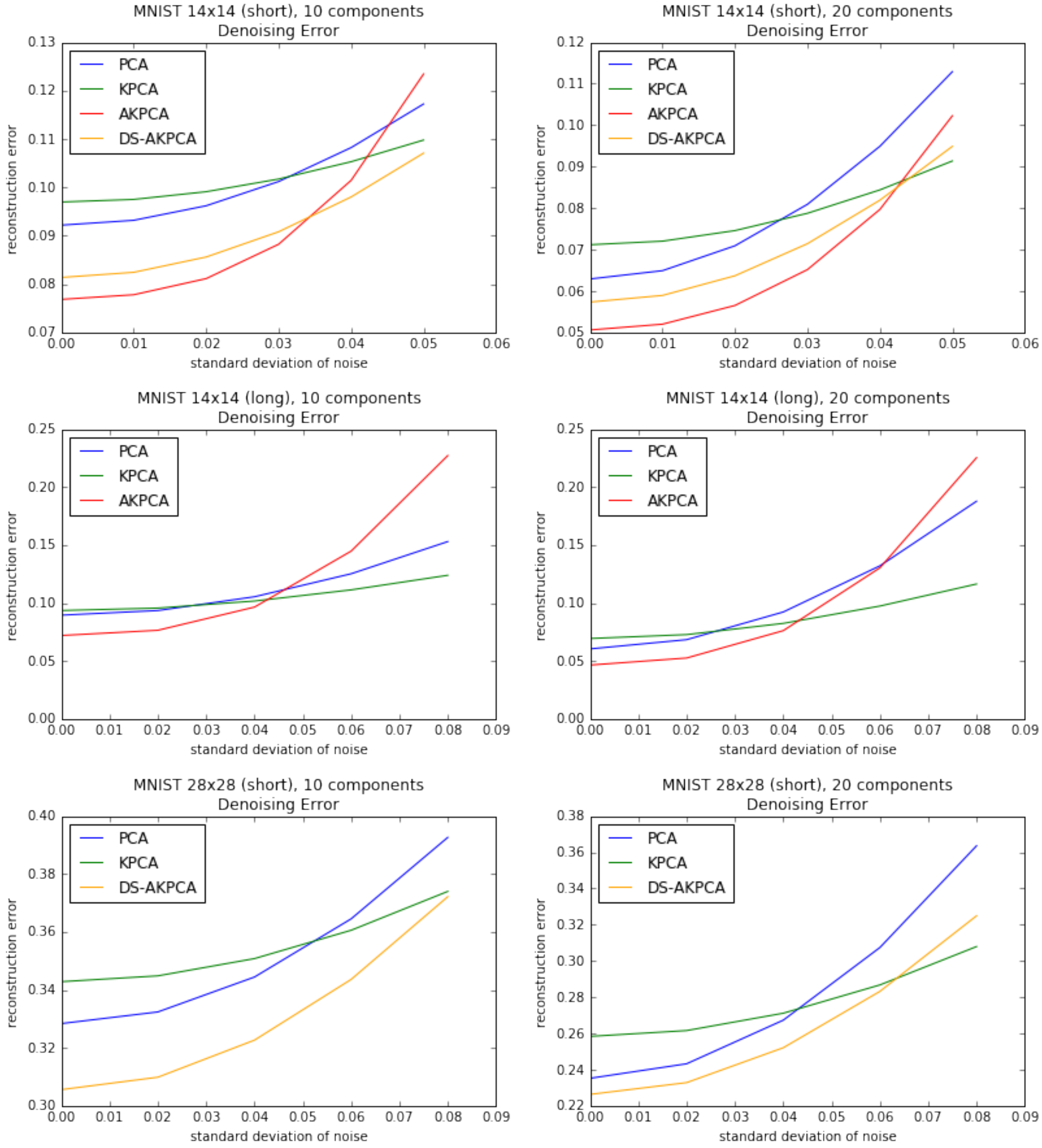

Figure 8: Denoising error of all four algorithms in denoising MNIST digits that have been corrupted by Gaussian noise, with 10 (left) and 20 (right) components in the model.

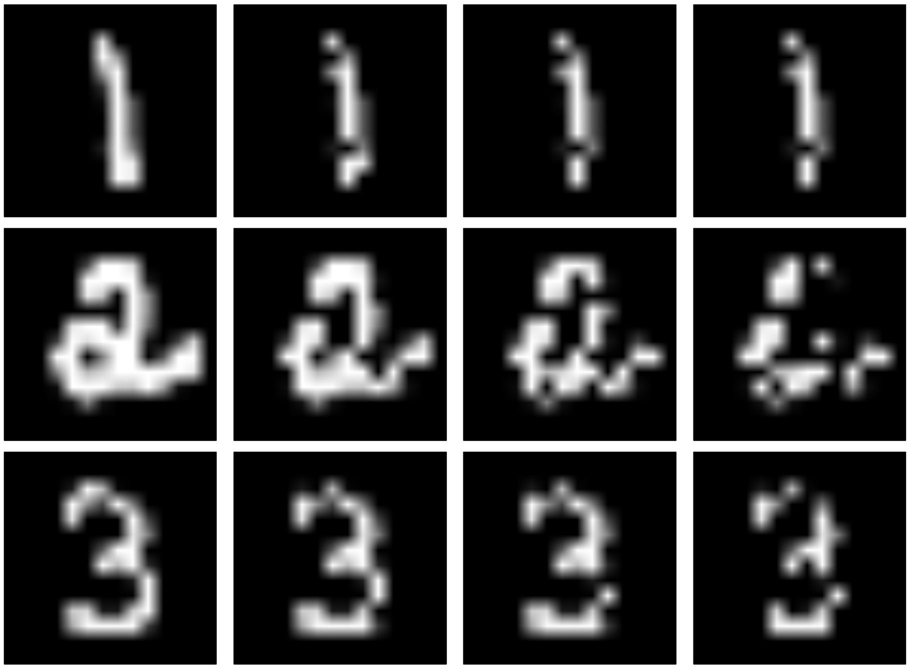

Figure 9: Three MNIST digits corrupted by increasing levels of speckle noise. From left to right: p, the probability that a pixel is set to zero, is varied in $\{0, 0.2, 0.3, 0.4\}$.

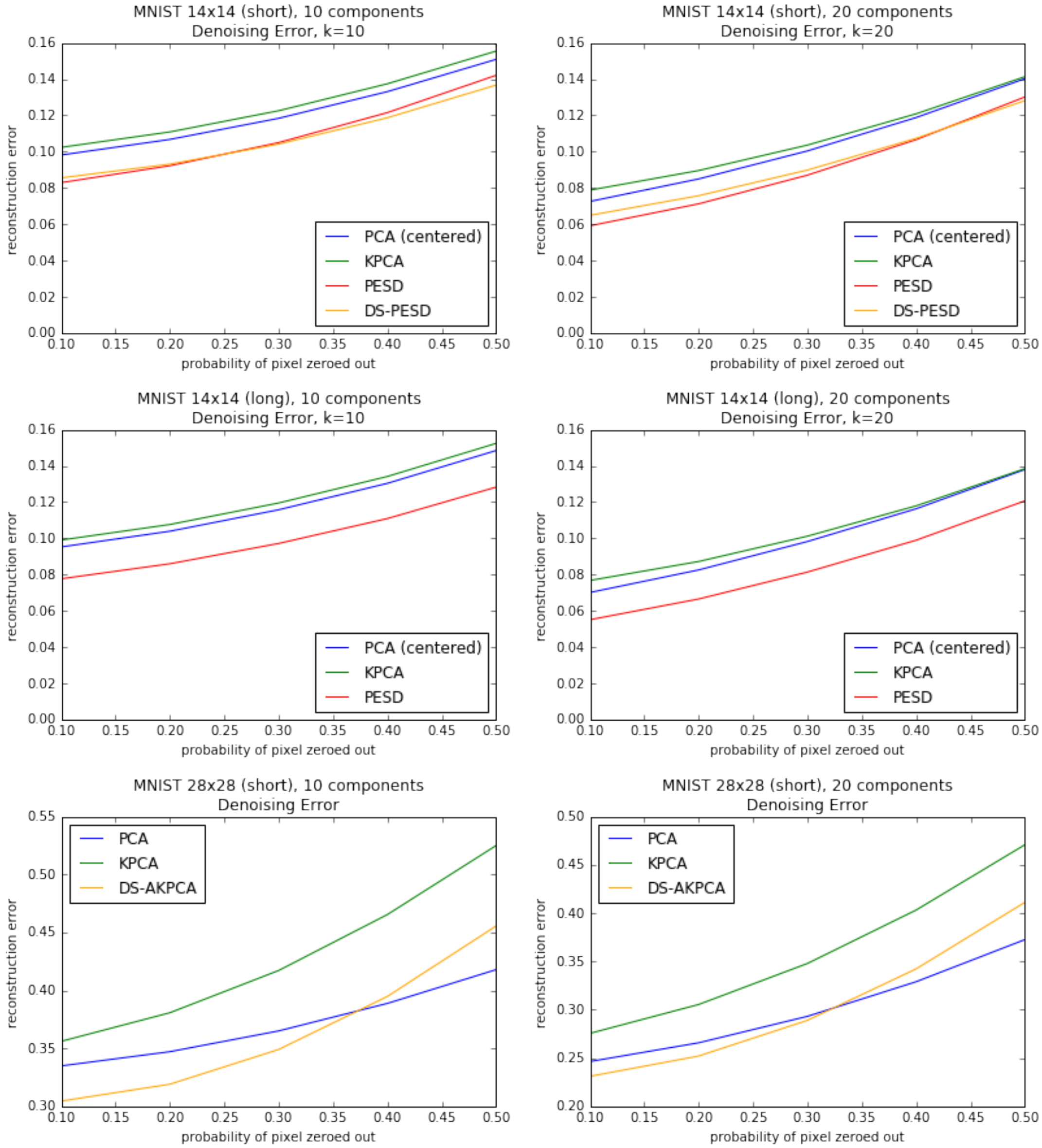

Figure 10: Denoising error of all four algorithms in denoising MNIST digits that have been corrupted with speckle noise, as a function of p, the probability that a pixel is set to zero. Plots shown for models of 10 (left) and 20 (right) components.

6.0.5 Classification

The goal of the classification experiment was to evaluate whether the encoding function learned by each algorithm serves well as a preprocessing step for the 10-way classification task of discriminating digits. For each of PCA, KPCA, AKPCA, and DS-AKPCA, we learned an encoder f on the training set and then used f to encode the held-out set. We trained a linear classifier on a portion of the encoded held-out set and measured its accuracy on the remainder, using cross-validation to average the classification accuracy over several training / testing splits.

For the linear classifier, we used logistic regression with a “multinomial” or “softmax” loss. The probability that a data point $\mathbf{x} \in \mathbb{R}^k$ belongs to class c is modeled as:

$$p(y = c|\mathbf{x}) = \frac{\mathbf{w}_c^T \mathbf{x}}{\sum_{j=1}^{C} \mathbf{w}_j^T \mathbf{x}}$$

and the parameters $\mathbf{w}_1 \ldots \mathbf{w}_C \subseteq \mathbb{R}^k$ are learned by maximizing the likelihood of the training data labels.

We tried training the classifier on both a small training set (18 examples per class) and a large training set (1750 examples per class).

Figure 6.0.5 plots cross-validated classification accuracy as a function of number of components, for short (left) and long (right) dimensionality reduction training set sizes, and small (top) and large (bottom) classification training set sizes. The only regime where AKPCA outperformed its competitors across all model orders was when both the dimensionality reduction training set and the classification training set were small (upper left corner). In all other settings, KPCA and occasionally PCA outperformed AKPCA for large enough model order ($k \geq 20$).

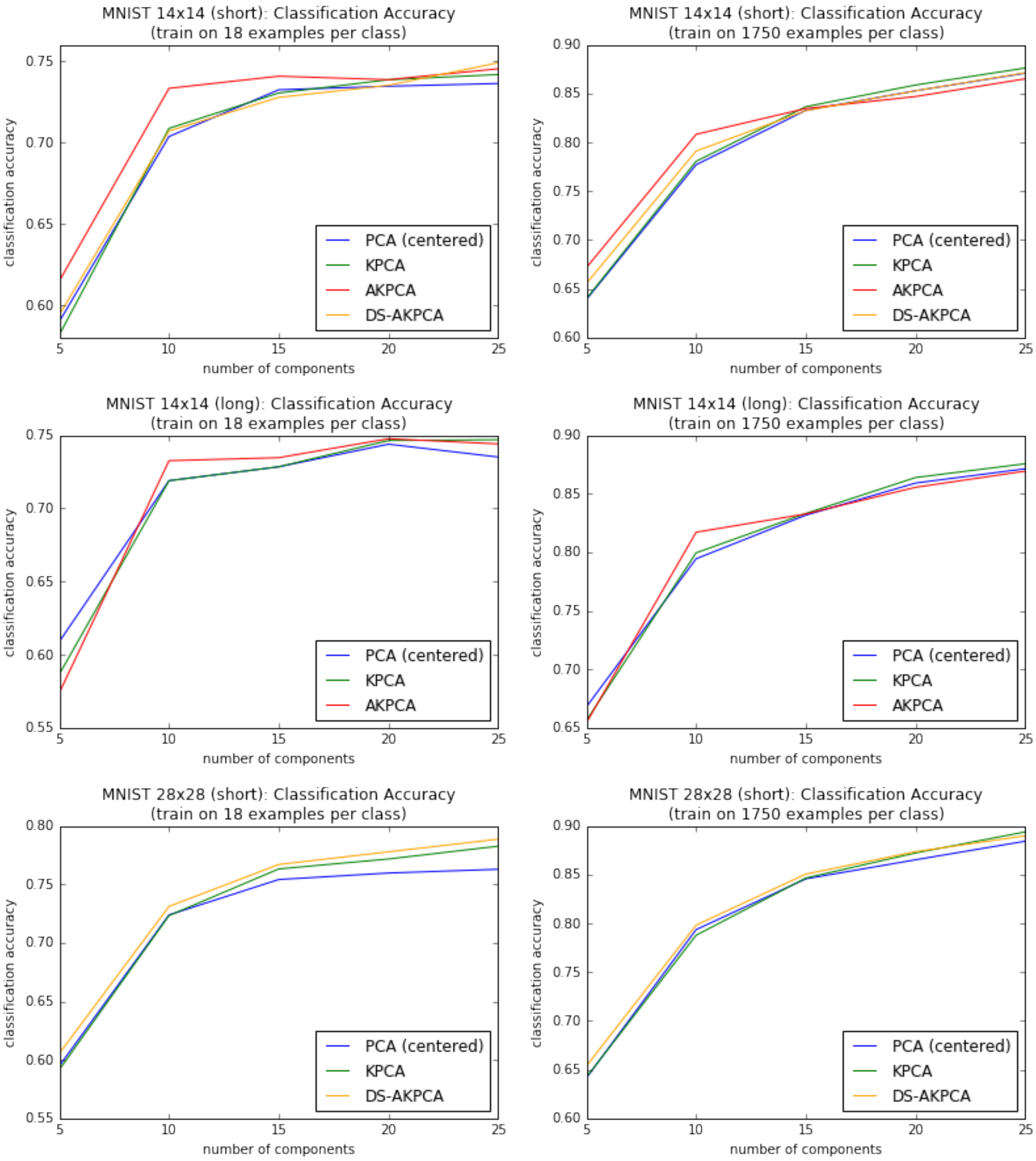

Figure 11: Cross-validated classification accuracy as a function of number of components, for small (left) and large (right) classification training set sizes.

7 Conclusion

The “autoencoder principle” is that a dimensionality reduction algorithm learns informative structure if it learns an *encoder* that compresses data into a low-dimensional encoding, and a *decoder* that reconstructs the original data from the encodings. Principal component analysis can be derived from the autoencoder principle, yet it is limited by the underlying assumption that the data lie close to a linear subspace. On the other hand, kernel PCA can discover nonlinear structure, yet does not have an autoencoder interpretation. In this technical report, we presented autoencoding kernel PCA (AKPCA), an algorithm derived from the autoencoder principle which can learn non-linear structure. AKPCA learns to compress the degree-2 polynomial expansion of a data point into a low-dimensional encoding, and to reconstruct the original data point from there. We also presented a “data-span” variant of AKPCA which scales better to datasets with high dimensionality, provided that the number of data points is small.

We showed that AKPCA and DS-AKPCA outperform PCA and kernel PCA with a degree-2 polynomial kernel at a denoising task on the MNIST dataset. Unfortunately, results on a classification task were mixed — our two algorithms generally matched PCA and KPCA in classification accuracy, but failed to consistently outperform them. Likewise, our algorithms did not perform well at compressing the CIFAR-10 or LFW datasets.

A final drawback to the two algorithms we propose is that they are computationally expensive: whereas PCA takes seconds on the datasets we consider, and kernel PCA takes minutes at most, our algorithms require a runtime on the scale of hours, since they perform stochastic gradient descent on a non-convex objective.

References

[1] Elad Hazan and Tengyu Ma. “A non-generative framework and convex relaxations for unsupervised learning.” *Advances in Neural Information Processing Systems.* 2016.

[2] James Kwok and Ivor Tsang. “The Pre-Image Problem in Kernel Methods.” *International Conference on Machine Learning.* 2003.

[3] Rie Johnson and Tong Zhang. “Accelerating Stochastic Gradient Descent using Predictive Variance Reduction.” *Advances in Neural Information Processing Systems.* 2013.

[4] Sebastian Mika, Bernhard Scholkopf, Alex Smola, Klaus-Robert Muller, Matthias Scholz, and Gunnar Ratsch. “Kernel PCA and De-Noising in Feature Spaces.” *Advances in Neural Information Processing Systems.* 1999.

[5] Bernhard Scholkopf and Alexander Smola. *Learning with Kernels.* 2002.

[6] Bernhard Scholkopf, Alexander Smola, and Klaus-Robert Muller. “Kernel Principal Component Analysis.” *International Conference on Artificial Neural Networks.* 1997.

[7] Christopher Williams and Matthias Seeger. “Using the Nystrom Method to Speed Up Kernel Machines.” *Advances in Neural Information Processing Systems.* 2001.