Dimension-Reduction Methods for Image Classification

10-605 Final Project Group 21: Yuwei Zhu, Vincent Mai, Amy Lee, Anthony Wu

I. Introduction

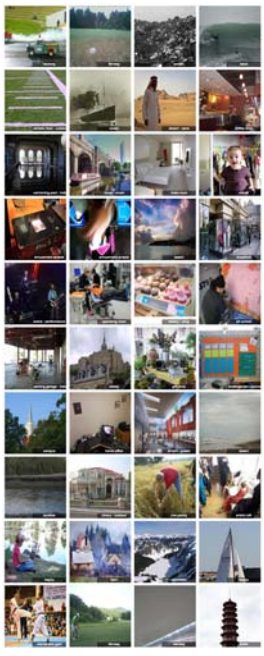
Motivation

Dimension reduction has important applications, such as data visualization in data science and data pre-processing in machine learning. In the context of image classification, images are often represented as high-dimensional samples composed of a matrix of thousands of pixels with multiple color channels each. These high-dimensional samples can be difficult to classify due to their large number of features, and they can be computationally expensive to work with due to their large size. Therefore, it is often helpful to reduce the image dimensionality to facilitate the classification process.

Although a large number of nonlinear techniques for dimensionality reduction have been proposed in the last decade, there are only a few studies focusing on implementing dimension reduction methods on a large scale image dataset. Therefore, this project presents a systematic comparison of dimensionality reduction techniques in the setting of large scale computing.

Dataset

A good example of the difficulties involved in high-dimensional image classification is the **365Places** image dataset. This dataset contains approximately 1.8 million images of places and scenery, evenly distributed among 365 different categories. The images in this dataset have been resized to have a minimum dimension of 512 while preserving the aspect ratio of the images. Original images that had a dimension smaller than 512 were left unchanged. The images were stored as a matrix of dimension (N, 512, h, 3), where:



- N = the number of images
- 512 = the width, in pixels, of each image
- h = the height, in pixels, of each image (varies)
- 3 = the number of color channels for each image

Each image is also labelled with its category.

The full dataset is further split into 3 sub-partitions of training (105 GB), validation (2.1 GB), and testing data (19 GB). In total, the dataset is roughly 126 GB large.

- The training dataset contains 1,803,460 images, with between 3,068 and 5,000 per category.
- The validation dataset contains 36,500 images, with 100 images per category.
- The testing dataset contains 328,500 images, with 900 images per category.[1]

Goals

Given the nature of the above dataset, it is not immediately clear what the best way to reduce the dimension of the images would be. Therefore, we chose to implement 3 different dimension-reduction techniques and compare their performance with respect to selected metrics. We guided our analysis with the following questions:

1) *Scalability*: How do different dimension-reduction methods scale to handle our 130 GB dataset?
2) *Complexity*: What is the complexity (computation, memory usage, run time, model tuning complexity, etc.) of each dimension reduction method?
3) *Reduction Quality*: What is the quality of the reduced images for each method?
4) *Classification Performance*: What is the performance of a generic classifier on the dimension-reduced dataset?

By answering the above questions, we hope to present a systematic comparison of the 3 chosen dimensionality reduction methods that may provide insight for researchers who are interested in image data dimension reduction.

II. Methods & Metrics

Methods

We chose to implement the following 3 methods: **Principal Component Analysis (PCA)**, **Kernel PCA (KPCA)**, and **Deep Autoencoder**. We based our choice on quantitative metrics (e.g. storage, computation, communication requirements, etc) and qualitative metrics (e.g. compatibility with given hardware and frameworks, classification accuracy on the reduced images, etc).

Frameworks

- We chose to use the **Spark** framework because it is easily able to distribute large datasets across worker nodes and perform parallel computations. We therefore believed that it would be able to handle the large 365Places dataset and the matrix operations involved in PCA and KPCA. We also believed that it would be suitable for an Autoencoder

[1] Note: the testing dataset labels were not made public in order to avoid bias during training.

implementation, but ultimately were unable to achieve this, so we switched to just using Tensorflow.

- We also chose to use **Tensorflow** because it is able to take advantage of GPUs, which we believed would be suitable for the matrix computations required for PCA, KPCA, and Autoencoding.

PCA

We chose PCA because it is relatively simple to implement in a distributed setting and is able to capture linear feature relations. In practice, PCA is also a common method used for image compression. Therefore, we believed that PCA would serve as a good baseline method.

Expected performance:

- Quantitative Metrics

  For a dataset with n samples and k features, an iterative and distributed implementation of PCA would require $O(k)$ local storage and $O(k)$ computation for each worker, with a $O(kr)$ communication complexity for some communication constant r. Since these requirements do not depend on n, we therefore expect PCA to have high scalability.

- Qualitative Metrics

  In order to perform well, PCA generally assumes that the features have a linear relationship. But in the context of image classification, where our images are represented as matrices of colored pixels, we believe that the features of each image are highly non-linear and hard to capture using PCA. Therefore, we don't expect a classifier using PCA-reduced images as input to perform as well as a classifier using KPCA-reduced or Autoencoder-reduced input.

KPCA

Kernel PCA is a reformulation of traditional linear PCA in a high-dimensional space that is constructed using a kernel function. KPCA computes the principal eigenvectors of a given kernel matrix, rather than those of the covariance matrix as in PCA. The application of PCA in the kernel space gives Kernel PCA the ability to construct nonlinear mappings. Therefore, we chose KPCA in order to capture nonlinear relations in images. We chose the radial basis function (RBF) as our kernel function since it is commonly used in practice.

Expected performance:

- Quantitative Metrics

  For a dataset with n samples and k features, the size of the kernel matrix is $O(n^2)$ and would require that much storage. Matrix computations involving the kernel matrix would also have $O(n^3)$ complexity, and communication would have $O(n^2r)$ complexity for a

constant *r* since we need to migrate the full kernel to each worker. Therefore, running KPCA on a large dataset could be a great challenge due to the high storage, computation, and communication requirements. We expect KPCA to have the least scalability in comparison with other two methods.

- Qualitative Metrics

  We believe that images have highly non-linear features. KPCA is able to capture non-linear features, and KPCA has been successfully applied to face recognition and image segmentation in previous research. Therefore, we expect a classifier using KPCA-reduced images as input to perform better than a classifier using PCA-reduced input. However, we do not expect it to perform as well as Autoencoder-reduced input.

Deep Autoencoder

And finally, we chose to implement a Deep Autoencoder because we believed it would capture complex and non-linear relations to a greater degree than the other two methods. Mathematically speaking, a three-MLP-layer-autoencoder that minimizes reconstruction MSE has an optimal solution similar to maximizing variance in PCA[2]. To differentiate between the two methods, we used a deep autoencoder with five hidden layers to capture the non-linearity in the data. Given the size of our dataset, to speed up training time, we simplified the architecture for the autoencoder by keeping it symmetrical and tied-weights between the encoder and the decoder. This halved the number of parameters to be trained.

Expected performance:

- Quantitative Metrics

  The number of training examples was expected to be a major bottleneck because our autoencoder could only process a small batch of *b* images at a time. The autoencoder also runs mini-batch SGD for *e* epochs to determine encodings for each batch, which further increases the computation time. Overall, for a dataset with *n* samples and *k* features, the expected requirements are storage: O(*bk*), computation: O(n^2ek), communication: O(*n*). As long as we can select a reasonable mini-batch size and perform mini-batch SGD, we are confident that it will be able handle large scale data, although the runtime will not be very good. We expected that we would need an extra boost from GPU parallel processing to get reasonable runtime, but we thought that memory should not be an issue as long as we don't try to process too many training examples in a batch.

- Qualitative Metrics

  Our autoencoder had more tunable parameters (e.g. number of epochs, optimization method, activation function, size and number of layers, etc) compared to PCA and KPCA,

[2] https://www.cs.toronto.edu/~urtasun/courses/CSC411/14_pca.pdf p.16

so it was able to capture more non-linear features and complexities. Thus, we expected a classifier using Autoencoder-reduced images as input to perform better than a classifier using PCA or KPCA-reduced input. But, we suspect that it will not achieve the same performance compared to vision-specialized autoencoders such as convolutional autoencoders.

Metrics

We chose to evaluate the methods with respect to the following 7 metrics:

Metric	*Description*
Quantitative	
Runtime	The amount of time needed to generate the reduced images.
Memory Usage	The amount of memory needed to run the method.
Reconstruction Error	The RMSE between the original and reduced images.
Scalability	How large dataset size influences the performance.
Classification Performance	How well a basic Multiclass Logistic Regression classifier is able to classify the reduced images.
Qualitative	
User-friendliness	How complex (e.g. number of tuning parameters, evaluation time, etc.) is the implementation process of the method?
Reconstructed Image	How similar the reconstructed image is with the original image, i.e. how well the reconstructed image captures salient features of the original image.

Together, we believe that these metrics capture the effectiveness and performance of each method.

III. Computation

The full ML pipeline for our project is pictured below:

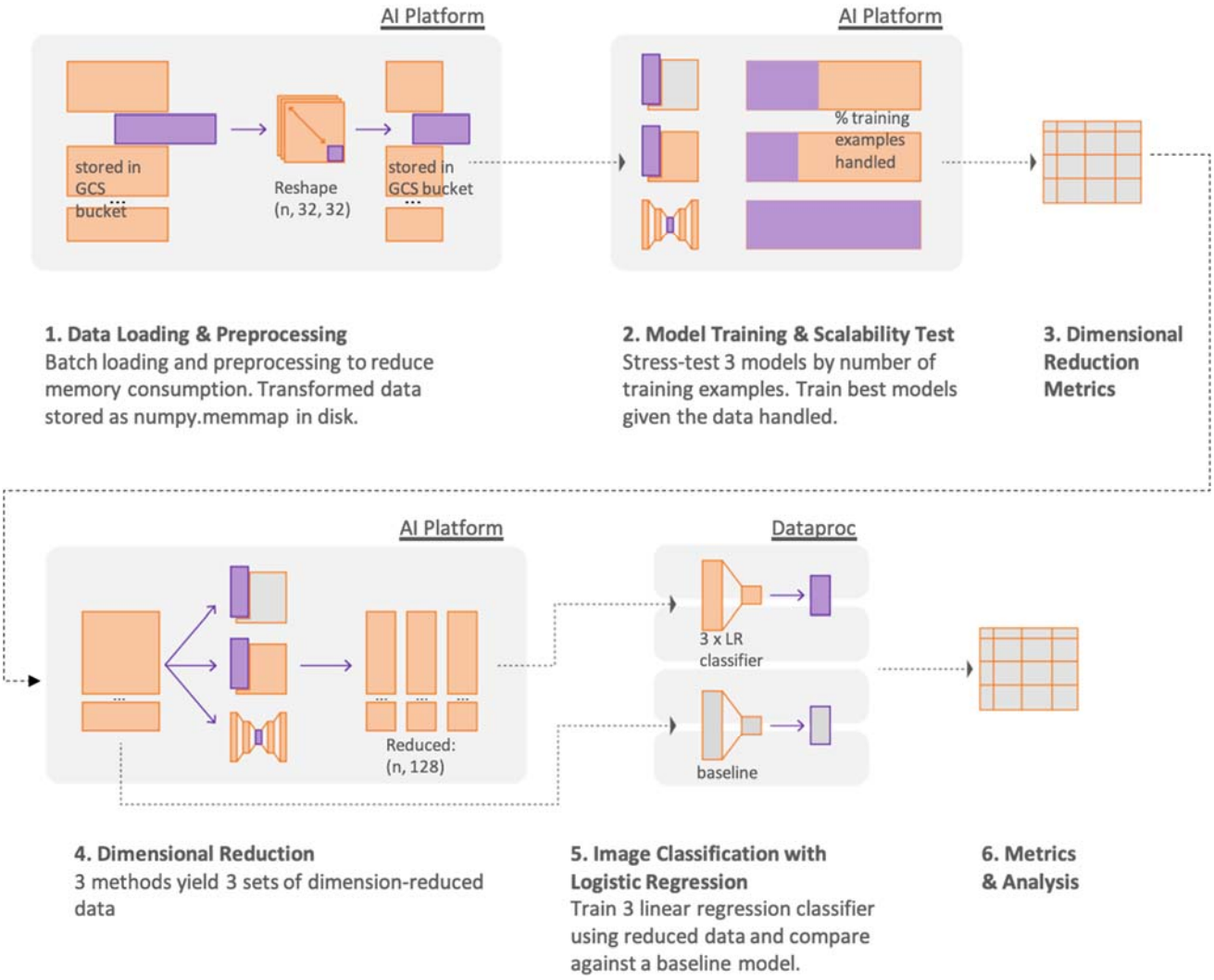


Fig. 1: Project Pipeline

Data Preprocessing

We decided to pre-process the data to make dimension-reduction easier, since the large number of samples and a large image dimension required a prohibitively large amount of memory and computation.

Initially, we considered two preprocessing methods to extract features from each image: spatial histogram and visual bag-of-words. But since our goal was to compare dimensional reduction methods, we decided not to over-preprocess the data so that correlations in the raw data could

be more purely captured by our reduction methods. Ultimately, we scaled down each image and eliminated the original RGB color channels by converting to grayscale, thus reducing the amount of memory required to store the data during processing. We also normalized the images across the dataset.

Computations

Below, we analyze 3 selected computations that were performed as part of the ML pipeline: PCA, KPCA, and multi-class logistic regression.

PCA

We implemented our own versions of PCA in both Spark and Tensorflow since built-in PCA functions did not have the functionality we wanted. The main computation involved the following steps: 1) centering the data (using optimized tensor operations), 2) computing the covariance matrix of the centered data (using optimized tensor operations), 3) computing the eigenvalues and eigenvectors of the covariance matrix (using built-in functions), and finally 4) multiplying the eigenvectors with the data to project the data onto a lower dimension.

Although this method is very basic and commonly used, we found it interesting because, despite its simplicity compared to KPCA and Deep Autoencoding, it ultimately performed *better* than KPCA and about the same as the Deep Autoencoder. It was also much harder to implement than we originally thought it would be due to memory limitations (which we discuss in-depth later on in this report).

Kernel (RBF) PCA

In this project, we implemented Kernel PCA both in PySpark and Tensorflow since there is no available function for Kernel PCA provided in PySpark or Tensorflow. The difference between PCA and Kernel PCA is that KPCA replaces the covariance matrix with the kernel matrix. One of the most interesting parts of implementing Kernel PCA is calculating the kernel matrix in a distributed setting[3].

Our approach to calculate the kernel matrix is divided into three steps. First, we create an index matrix to store the index of data pairs that distance need to be calculated by computing the Cartesian product of the dataset index and filtering only the upper triangle matrix. After this step, we get a distributed element matrix with vectors of the form: ((1,2), (1,3), (1,4), ...). Next, we map each element in the element matrix to a distance calculation function to get the distance for each pair. The intermediate distance matrix has the form: (1 , (2, d12), (3, d13), …). In the end, we use the `groupByKey` and `map` functions to get the final distance matrix (1 , [d1, d2, ...]), (2, [d1, d2, ...]).

[3] Although in the final pipeline, we implement KPCA in tensorflow instead of PySpark. Distributing the calculation of kernel matrix is still one of the most challenging and interesting parts that we believe worth sharing.

The reason why we think this computation is interesting is that this approach uses index instead of directly calculating distance between each data point, which makes the computation process possible to distribute. For most of the distributed computing we discussed in class, we change the matrix multiplication order to make the computation distributed. Using index to create an intermediate matrix to facilitate the distribution process is a brand new technique for us.

Multi-class Logistic Regression

We wanted to test whether our dimension-reduction methods were useful for making classification tasks easier, so we decided to feed the dimension-reduced results of each method to a basic multi-class logistic regression classifier. A separate SVM was trained using each of: 1) the original pre-processed images, 2) the PCA-reduced images, 3) the KPCA-reduced images, and 4) the Autoencoder-reduced images. The SVM is trained by iteratively sampling a new data point, applying the current (linear) predictor to the data point, and updating the predictor's weight and bias vectors based on the accuracy of the prediction.

The computation was made more difficult due to the sequential nature of the SVM training, so we couldn't fully leverage distributed systems. We also had some doubts about using multi-class logistic regression as a classifier since it works best when features are linear, and we expected the images to have highly non-linear features. However, we thought it best to keep things simple so we could focus on the performance of each dimension-reduction method.

Challenges

We faced a number of implementation challenges, most of which stemmed from the massive size of the dataset and the learning curve associated with using Google Cloud Platform (GCP).

- We initially struggled with learning how to use GCP. For example, when we first set up our GCP accounts, we were unable to create Projects because our email accounts were restricted under the CMU domain. Eventually, we were able to work around this by using our personal email accounts instead.
- Once we had GCP accounts set up, we ran into problems with Dataproc. We were often unable to open Jupyter Notebooks (to set up environments for running our code) because the clusters kept crashing or freezing. Shutting down a cluster would also cause us to lose the data and progress made on that cluster, so we were forced to work around this by leaving the cluster running and saving out all of our data to a GCS bucket. Leaving the cluster running incurred a lot of costs, and saving out the data was time-consuming due to the large scale of the dataset. We eventually migrated over to Google AI Platform since it seemed more stable and had better hardware (GPUs).
- Although we intended to implement all 3 dimension reduction methods in Spark, we were unable to implement the Deep Autoencoder in Spark. So, we eventually migrated every method to Tensorflow for the sake of fair comparison.

- We often ran into memory issues when running PCA and KPCA because the dataset was so large that we were unable to load it onto our worker nodes, which we needed in order to compute the covariance matrix (in PCA) or kernel matrix (in KPCA). This was also an issue during the pre-processing stage, because the combined memory of the worker nodes was not enough to process the full 100+ GB dataset. Taking inspiration from mini-batch SGD methods, we worked around this problem by loading data in smaller batches that *could* fit on each worker node, and performing pre-processing simultaneously to avoid extra communication. We set the batch size to 1000 and the run time was about 8 seconds per batch. The total runtime was a little over 4 hours but without any memory issues.

- To deal with the memory issues, we also heavily relied on `numpy`'s `memmap` function, which allows us to directly map data on disk to an array object in our code. Memory-mapped files can be used to access small segments of large files on disk without reading the entire file into memory. Each time we load and write data, we first create a `memmap` object and then read or write data by small segments of the entire dataset.

- We had been using `memmap` for loading all of the data for training since it is memory efficient, but it has a downside: while training the autoencoder, we noticed that the mini-batch SGD utilized by the autoencoder was updating the weight matrix one training example at a time. We realized that `memmap` uses a generator for data loading, and can only load one example at a time. After some searching we could not find a good solution for solving this issue; we were stuck with interfacing between `numpy`'s `memmap` and TensorFlow's mini-batch SGD implementation. This increased the autoencoder's training time by a large factor and we weren't able to take advantage of the full GPU parallel processing power available to us.

Resources & Tools

We implemented the entire project in Google Cloud Platform and used three products: **AI Platform, Dataproc,** and **Storage**. We also used **Google Colab** for prototyping.

We implemented data preprocessing and 3 dimension reduction methods in Google AI Platform using Tensorflow with specialized hardware for ML. The instance we used was: N1: 8x vCPU, 30 GB RAM, 1x NVIDIA Tesla T4 GPU. For the multiclass classification task, we implemented the logistic regression on three dimension reduced datasets and one preprocessed dataset (without applied dimension reduction methods) on Dataproc using Pyspark to utilize distributed computing. We used 1 master node (N1: 16x vCPU, 60 GB RAM) and 6 worker nodes (N1: 8x vCPU, 30 GB RAM) on Dataproc.

This project was implemented in Python entirely. The primary libraries we used are: Keras, TensorFlow, Pyspark, Numpy.

The entire project cost us around $500 (Educational coupon + free GCP credits) over two weeks.

In terms of time commitment, the running time for data preprocessing was about 2.5 hours and running each dimension reduction method took around 3.5 hours. However, this running time does not account for the time spent on failed attempts at running PCA and KPCA that terminated due to memory issues. Overall, it took three of us spending around two weeks to make the entire pipeline work.

IV. Results & Analysis

Quantitative Metrics

Methodology

To compare the scalability of the three dimension reduction methods, we randomly shuffled the training dataset and then sliced the entire dataset into subsets with different sizes. We gradually increased the size of the subsets to stress-test the maximum dataset size that each method could handle. Meanwhile, we recorded the runtime and memory usage[4] of each method running on different sizes of the dataset.

Next, we trained each method on the maximum dataset size it could handle to generate a dimension-reduction model. Then we calculated reconstruction error[5] after using the models to dimension-reduce the validation dataset. Finally, we fed the dimension-reduced images for each method to a Multiclass Logistic Regression model. We trained each classifier using 10-fold cross validation and calculated the accuracy and F1 score as the classification performance metric.

Results

Scalability

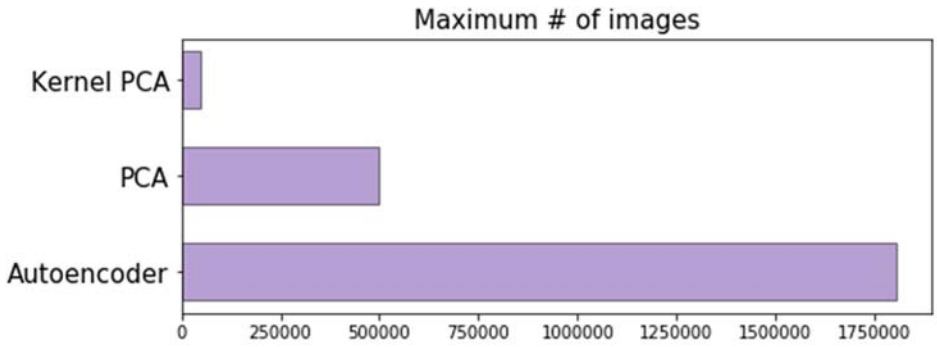


Fig. 2: The maximum number of images that each reduction method is able to handle.

To measure scalability, we stress-tested all three models to determine the maximum amount of training data that the model could handle. While the autoencoder was able to handle the entire

[4] Memory usage is measured by the changes in Resident Set Size before and after running each method.

[5] We were unable to figure out how to compute reconstruction error for Kernel PCA.

dataset, which contains about 1.8 million images, PCA and kernel PCA were only able to handle approximately 500k and 50k images, respectively, due to memory limitations. This aligns with our expectations (as discussed earlier in the report).

Complexity: Runtime

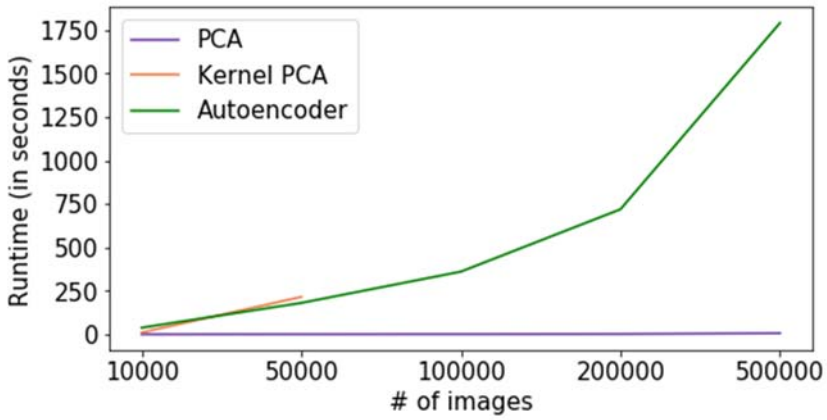


Fig. 3: The amount of time each reduction method took to process datasets of varying sizes.

While stress-testing each reduction method, we measured the runtime. In terms of runtime, we see that PCA performs best: its runtime increase is almost undetectable in comparison to the other two methods. We hypothesize that the drastic increase in the Autoencoder's runtime might be due to the fact that we can only load a few samples from memory at a time. We could not take advantage of distributed batching, which greatly increased the Autoencoder training time.

Complexity: Memory Usage

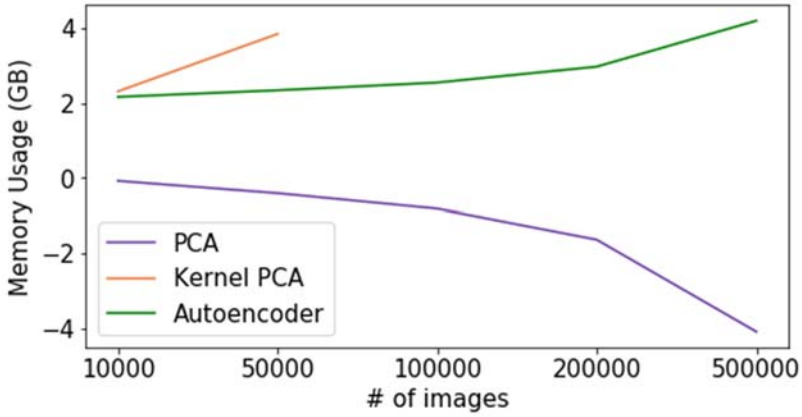


Fig. 4: The memory usage of each reduction method for datasets of varying sizes.

While stress-testing each reduction method, we measured the memory usage by recording the Resident Set Size (RSS), which is the amount of memory occupied by a process that is held in main memory before and after computation. We see relatively linear increases for both KPCA and Autoencoder. However, we see a decrease in RSS for PCA. We are unsure of the reason behind

this. We hypothesize that our measurement method may be faulty and requires further investigation.

Reconstruction Error

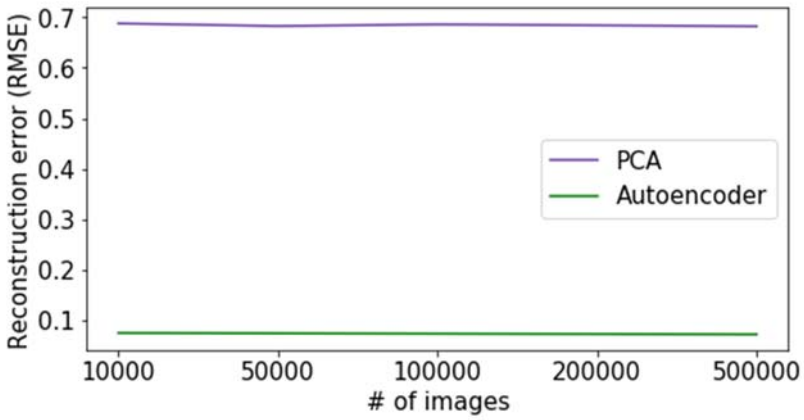


Fig. 5: The reconstruction error for each reduction method.

To determine reconstruction error, we computed the RMSE between the images reconstructed by each method versus the original pre-processed image. In reconstruction error, we see that Autoencoder is generally able to obtain a very low RMSE whereas PCA has higher RMSE. We didn't plot RMSE for KPCA here because the reconstruction for KPCA is not straightforward.

Classification Performance

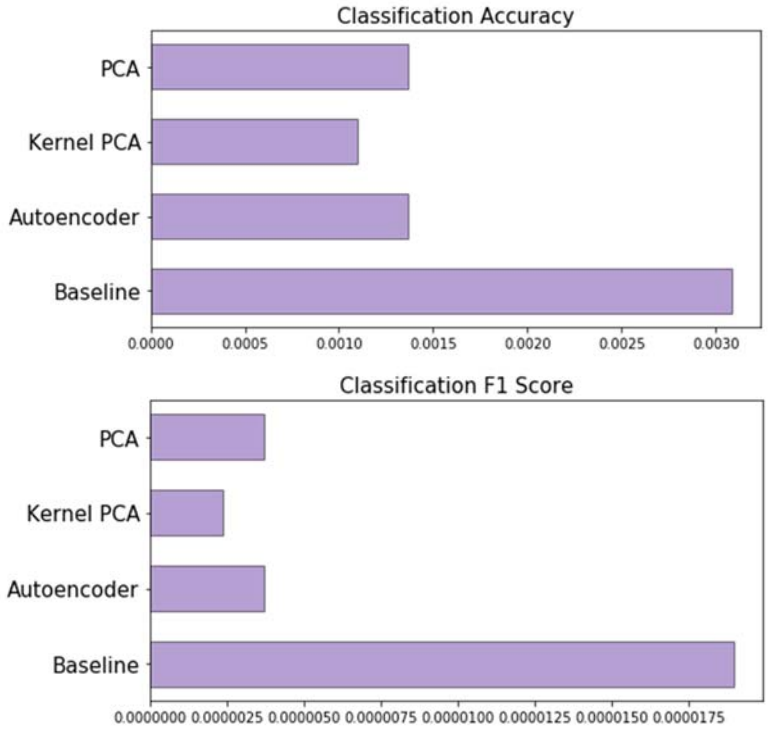


Fig. 6: The classification accuracy and F1 score of each reduction method.

Our last quantitative metric is the multiclass classification performance. We measured the performance of a logistic regression classifier using reduced images from each method as input, with the original pre-processed images as the baseline input. We found that Autoencoder performs best, PCA performs about the same as the Autoencoder, and Kernel PCA achieves the worst result. None of the three reduction methods perform as well as the baseline, but this is expected since dimension-reduction causes a loss of information that could influence the prediction.

Qualitative Metrics

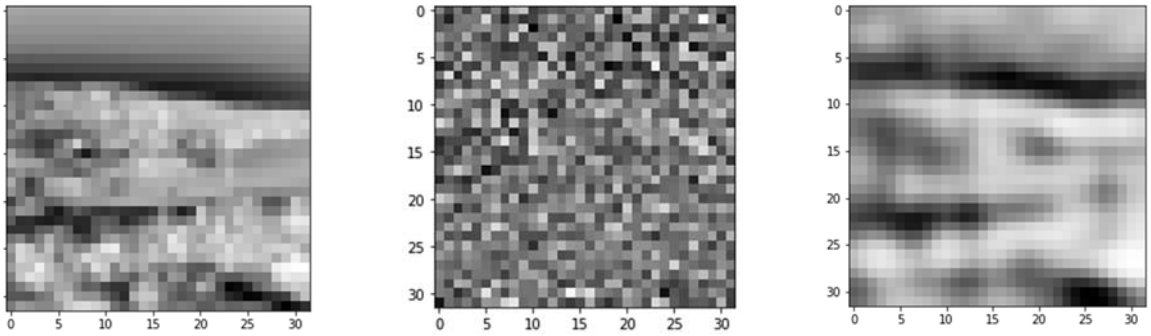
User-friendliness

We measured the user-friendness in terms of the difficulty of tuning the methods. We can see that PCA and Kernel PCA are easy to tune since PCA has no parameters to tune while Kernel PCA only has one parameter (i.e. *gamma*, a parameter of the radial basis function) to tune. However, to train an Autoencoder, there are many parameters when designing the architecture of the neural network model. For example: the number of epochs, the optimization method, optimization algorithm, activation function, number of layers and size, etc. all need to be considered.

Reconstruction Quality

We gauged reconstruction quality based on the human-perceived similarity between the reconstructed image and the original image. In general, we desire higher similarity because it indicates that the reconstructed image has captured more of the salient features of the original image.

Fig. 7: Sample original raw image



Preprocessed Image | PCA Reconstructed Images | AE Reconstructed Images

Fig. 8: Reconstruction visualizations.

As mentioned before, PCA and Autoencoder achieve similar performance in the multiclass classification task. We are curious about how the reconstructed images look like for these two methods and why they perform similarly (as for KPCA, we were unable to determine a method of reconstruction). For the Autoencoder, we can see that it captures the main structure (sky, mountain and lake) of this specific image. However, the reconstructed image for PCA looks like noise. We cannot directly see the image structure anymore. We were surprised that the PCA performance was comparable to that of the Autoencoder despite its noise-like reconstruction, but we hypothesize that the reconstruction dissimilarity may be due to the fact that PCA and Autoencoder encode information differently.

Further Discussion: Cosine Similarity

After reconstructing the images for both PCA and Autoencoder, we realized that PCA and Autoencoder applied very different approaches to decompose the same image. This led us to investigate the similarity between the reduced images for each method (rather than their reconstructions). To do so, we computed the cosine similarity between the dimension-reduced validation dataset for each method. The result is shown below:

Methods to Compare	Cosine Similarity
PCA - KPCA	5.87742
PCA - Autoencoder	17.96488
KPCA - Autoencoder	16.786083

We found that images reduced using PCA and Kernel PCA are more similar to each other than those of the Autoencoder. This slightly differs from our original expectations, which were that PCA and Kernel PCA would have dissimilar results because PCA focuses on linear features while KPCA other captures non-linear features.

Combining the similarity score result with the classification task performance, we might conclude that PCA and Autoencoder have different approaches to decomposing images but both can achieve satisfying results in the context of image classification.

Conclusion

Overall, based on each of the methods we evaluated, we conclude that PCA performs the best in terms of computation time, memory usage, and accuracy. However, the Deep Autoencoder appears to be the most scalable, and could be improved if we could determine a way to leverage a distributed system to reduce the computation time.

V. Next Steps

We have developed a systematic comparison among three different dimension reduction methods in this project. Based on current results, we believe there are several potential avenues of further research:

- Testing scalability in Spark

  Currently, all the three methods are implemented in Tensorflow which means we cannot leverage the full benefits of distributed computing. Moving all implementations to Spark could benefit especially in terms of scalability. However, we need to develop an implementation of a Deep Autoencoder in Spark.

- Investigate some of the unexpected results we had obtained

  We recorded negative memory usage for the PCA implementation, which is very strange. Believing that there was an error, we tried to restart the kernel instance and rerun the entire pipeline several times but still saw negative memory usage. We are still not sure why this happens. It might be an interesting topic to investigate in the future.

- Research on calculating reconstruction error for KPCA

  In this project, we were unable to figure out the proper process of reconstructing images after applying Kernel PCA, which led to an incomplete comparison of reconstruction error and reconstructed images. Further study on Kernel PCA image reconstruction could add more insight to this project.