Memory-efficient and GPU-oriented visual anomaly detection with incremental dimension reduction

Teng-Yok Lee Yusuke Nagai
Akira Minezawa
Information Technology R&D Center, Mitsubishi Electric Corporation
{lee.teng-yok@ap,nagai.yusuke@cw,minezawa.akira@ds}.mitsubishielectric.co.jp

Abstract

Recent studies show that the image features from pre-trained convolution neural network (CNN) can be used for anomaly detection, even without fine-tuning. A common type of methods divides the image space into patches, and estimates the distribution of CNN-based features per patch of all training data. While this types of methods can achieve high accuracies, the high dimensionality of CNN features causes overhead to both computing and storage. In this paper, we present an incremental algorithm to reduce the dimensionality of CNN features during the training. As our algorithm ultimately computes the Truncated PCA of the features, it only maintains the truncated singular values and vectors during the training. Besides, to efficiently update the truncated singular values/vectors of all patches, we further optimize the algorithm in order to fully utilize GPUs for parallel execution. We show that with our approach, we can achieve high accuracies on the texture classes of MVTec AD with small memory footprint and extreme high speed (around 200FPS) on a single GPU.

1. Introduction

Given an image, visual anomaly detection aims to decide whether the image is an anomaly with respect to the known normal cases, and it will be ideal if the area of the anomaly can be detected too. Recently, several methods have shown that by using a deep convolution neural network (CNN) pre-trained on ImageNet [17], we can extract highly effective features for visual anomaly detection [5, 15, 16]. Given an image, these methods forward the image through a CNN and combine the extracted image features of multiple CNN layers into a long vector. With the vectors of normal cases in the training set, these methods estimate the corresponding distribution, and use the distribution to compare against the features of test images. Even though the CNN is not fine-tuned for the target scenario, on datasets like MVTec [2], these methods already can achieve higher accuracies that other fine-tuning-based approaches.

On the other hand, since each combined feature can have 1000s of elements, the dimensionality imposes several challenges. The first issue is the overhead to store the features and to estimate related parameters. PaDiM algorithm [5], for instance, divides the image domain into 54×54 patches, uses Wide ResNet (WRN) [19] to extract the features, and computes the sample covariance matrix per patch. As the original feature length of WRN is 1792, storing all sample covariance matrices requires $54 \times 54 \times 1792 \times 1792$ numbers, meaning 37 GB of space in single precision and thus impractical.

It is thus natural to ask whether we can apply conventional dimensional reduction techniques to shorten the feature vectors. Although it has been empirically shown that using dimension reduction techniques like Principal Component Analysis (PCA) could lead to lower accuracies than using random sampling [5], random sampling still requires hundreds of dimensions to achieve high accuracies, which is still memory consuming. If dimension reduction could achieve acceptable accuracies on certain scenario, it could be considered as a trade-off between accuracies and other engineering factors such as speed and memory consumption.

While it is trivial to apply PCA once the features or sample covariance matrices of all training data have been computed, as they could take GBs of memory to store, the memory usage is still challenging during the training. While there are methods to incrementally apply singular value decomposition [3, 4], these methods need to call complex linear algebra routines like QR decomposition multiple times, which could be complicated to implement on GPUs. Such a kind of routines could be sensitive to the numerical errors during the iterative update, and thus double precision arithmetic operations are often required. Consequently, although the CNN part can be accelerated by GPU in single precision, these methods might still require CPUs to execute, which lose the benefit of GPUs.

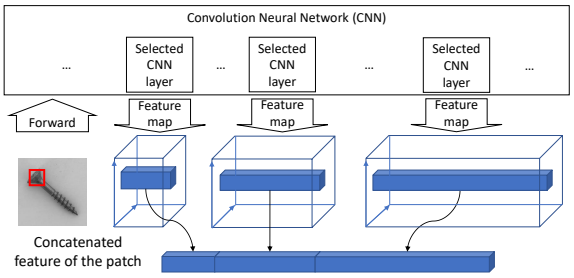

Figure 1. Feature extraction with CNN. After resizing the feature maps of pre-specified layers into the same size, we can concatenate the features of each pixel across these layers as the feature of the corresponded image patch.

In this paper, we present a GPU-oriented algorithm to incrementally reduce the dimensionalities of multiple matrices in parallel. Other than storing the features of all images first, we examine the images in batches and udpate the singular values and vectors during the iteration. Our algorithm is designed for sample covariance matrices, which is symmetric and thus can be computed with a single SVD, other than executing QR decomposition multiple times. To fully utilize GPUs, which could have limits on the matrix sizes when solving SVD for multiple in parallel, we optimize the update algorithm by limiting the matrix size, and deferring unnecessary operations until all batches have been examined. With our incremental algorithm, we can efficiently compute the decomposition with fixed and low memory footprint during the training.

This paper is organized as follows. After reviewing the terminologies of visual anomaly detection in Section 2, we describe our algorithm in Section 3. Section 4 shows the accuracies of our algorithm on the MVTec AD dataset [2], which shows that our algorithm is effectively on its classes of texture types when being combined with recent CNN like EfficientNet [18]. We discuss various issues about the speed, memory usage, and parameters of our algorithm in Section 5 and conclude with future work in Section 6.

2. Background

We first describe the basic idea to use CNN to extract image features for visual anomaly detection, which is also illustrated in Figure 1. Here we will use a single CNN to extract the features from different locations in an image. As each CNN layer leads to a 2D map of feature vectors, and different CNN layer corresponds to different semantics in the image, it is common to combine the features of multiple pre-specified layers. Once these maps are resized to the same spatial size, each pixel in the maps corresponds to a patch in the image, and we can concatenate the features of the same pixel in all layers to form the final feature vector of this patch.

Given n normal images in the training set, we can use the corresponded feature vectors per patch to model the distribution of normal data. When testing an image, we compare the feature vector of each patch in the test image against the distribution of normal data. By mapping the comparison result to a numerical score per patch, we obtain a map of anomaly scores, which can be used to infer whether the image contains anomaly and where the anomaly could be.

A simple and yet effective distribution is multivariate Gaussian distribution, which has been demonstrated by Rippel *et al.* [15] and PaDiM by Defard *et al.* [5]. By denoting the training features of a patch as $x_1, \dots x_n$, each of which is a m-dimensional vector, the multivatiate Gaussian distribution can be estimated by its sample mean μ and sample covariance Σ per Equations 1 and 2, respectively:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$

$$\Sigma = \frac{1}{n-1}\sum_{i=1}^{n} x_i x_i^\intercal - \frac{n}{n-1}\mu\mu^\intercal \tag{2}$$

When testing a sample x, its Mahalanobis distance from the multivariate Gaussian distribution can be used as its anomaly score, which is $\sqrt{(x-\mu)^\intercal \Sigma^{-1}(x-\mu)}$. However, the inverse of Σ cannot be stably computed since the rank of Σ is lower than the feature length. One solution is adding a weighted identity matrix to the covariance Σ before computing the inverse. Another approach is applying truncated singular value decomposition (TruncatedSVD) [6] to decompose Σ, and using the dominated singular values and vectors to compute Mahalanobis distance.

More precisely, TruncatedSVD approximates Σ by $U_k \text{diag}(\sigma_1, \dots, \sigma_k) U_k^\intercal$, where $\sigma_1, \dots, \sigma_k$ are the top k singular values, $\text{diag}(\sigma_1, \dots, \sigma_k)$ as a diagonal matrix with the singular values along the diagonal, and the columns of U_k are the corresponding singular vectors. According to Equation 3, the Mahalanobis distance can be computed by first projecting the vector $x - \mu$ as $\text{diag}(\sigma_1^{-1/2}, \dots, \sigma_k^{-1/2}) U_k^\intercal (x-\mu)$, and the computing the L2 norm of this projected vector.

$$\begin{aligned} & (x-\mu)^\intercal && \Sigma^{-1} && (x-\mu) \\ \sim\ & (x-\mu)^\intercal U_k && \text{diag}(\sigma_1^{-1}, \dots, \sigma_k^{-1}) && U_k^\intercal (x-\mu) \\ =\ & (x-\mu)^\intercal U_k && \text{diag}(\sigma_1^{-1/2}, \dots, \sigma_k^{-1/2}) && \\ & && \times\ \text{diag}(\sigma_1^{-1/2}, \dots, \sigma_k^{-1/2}) && U_k^\intercal (x-\mu) \end{aligned} \tag{3}$$

Another benefit of TruncatedSVD is that it can reduce the storage overhead of Σ, which is $O(m^2)$, to $O(m \times k)$ when using k singular values. While this is easy to achieve once Σ has been computed, the computing of Σ could be

storage-consuming. For small datasets like MVTec AD [2], it is common to first compute the features of all data samples and then the sample covariance, but this approach is not scalable for large datasets. Another approach is incrementally updating the Gram matrix, which is $\sum_{i=1}^{n} x_i x_i^\intercal$. This approach requires a $O(m^2)$ buffer to store the sum, which is data-independence but still challenge when the feature length m is large.

3. Algorithm

This section describes our incremental dimension reduction algorithm to compute the TruncatedSVD of Σ. Our algorithm divides the training feature vectors into B batches where the feature vectors of the b-th batch are represented by a matrix X_b of n_b feature vectors as the columns. As the sample covariance matrix Σ is equal to $\frac{1}{n-1}\sum_{b=1}^{B} X_b X_b^\intercal - \frac{n}{n-1}\mu\mu^\intercal$, our algorithm updates and stores the top singular values and vectors of the Gram matrix, which is equal to $\sum_{b=1}^{B} X_b X_b^\intercal$, when iterating through the batches. It should be noted that the term *batch* here can be different from the number of images to forward through the neural networks, which is often called batches in literature too. Hereafter we call the unit to update the dimensions as one *step*.

3.1. Incremental dimension reduction

For the first step, we collect more than k features to form X_1, and compute the initial TruncatedSVD such that $X_1 \sim U_k \text{diag}(s_1 \dots s_k) V_k^\intercal$. For each of the following steps, says b, our algorithm computes the TruncatedSVD of $\sum_{i=1}^{b-1} X_i X_i^\intercal + X_b X_b^\intercal$, which is approximated by Equation 4 since we only keep the dominate k singular values and vectors of $\sum_{i=1}^{b-1} X_i X_i^\intercal$:

$$\sum_{i=1}^{b-1} X_i X_i^\intercal + X_b X_b^\intercal \sim U_k \text{diag}(s_1^2, \dots, s_k^2) U_k^\intercal + X_b X_b^\intercal \quad (4)$$

By adding n_b ones to the diagonal in the middle matrix, Equation 4 can be rewritten by Equation 5:

$$[U_k, X_b]\text{diag}(s_1^2, \dots s_k^2, \underbrace{1, \dots, 1}_{n_b})[U_k, X_b]^\intercal \quad (5)$$

Since the right matrix is the transpose of the left matrix, based on the squared roots of the diagonal elements in the middle matrix, we can define a matrix W by Equation 6, add Equation 5 is equal to $WW^\intercal$:

$$\begin{aligned} W &= [U_k, X_b]\text{diag}(s_1, \dots s_k, \underbrace{1, \dots, 1}_{n_b}) \\ &= [U_k \text{diag}(s_1, \dots s_k), X_b] \end{aligned} \quad (6)$$

In summary, in every step, our algorithm computes the matrix W, and applies TruncatedSVD to decompose W into $U'_k \text{diag}(s'_1, \dots, s'_k) V'_k$. U'_k and $s'_i, \dots, s'_k$, respectively, become the updated singular vectors U_k and singular values $s_1, \dots, s_k$, which will be used in the next step.

3.2. Finalization

It should be noted that during the iteration, we are computing the singular values and vectors of Gram matrix, not the sample covariance yet. Thus at the end, we need one more step to complete the calculation. When iterating the data, we also accumulate all features into a $m \times 1$ vector x_a. Once all steps are iterated, our algorithm computes the mean $\mu = \frac{1}{n} x_a$ of all features, and cancels the mean per Equation 7:

$$\begin{aligned} &1/(n-1)\sum_{i=1}^{B} X_i X_i^\intercal - n/(n-1)\mu\mu^\intercal \\ \sim &U_k\{1/(n-1)\text{diag}(s_1^2, \dots, s_k^2)\}U_k^\intercal - n/(n-1)\mu\mu^\intercal \end{aligned} \quad (7)$$

By projecting the mean vector μ to the space of U and then reconstructing back to the original space, which becomes $UU^\intercal\mu$, Equation 7 can be rewritten by Equation 8:

$$\begin{aligned} U_k\{ &\quad 1/(n-1)\text{diag}(s_1^2, \dots, s_k^2) \quad \}U_k^\intercal \\ - &\quad n/(n-1)U_k U_k^\intercal \mu\mu^\intercal U_k U_k^\intercal \\ = U_k\{ &\quad 1/(n-1)\text{diag}(s_1^2, \dots, s_k^2) \quad \}U_k^\intercal \\ -U_k\{ &\quad n/(n-1)U^\intercal\mu\mu^\intercal U \quad \}U_k^\intercal \end{aligned} \quad (8)$$

Since the left matrices of both terms of Equation 8 are identical, and so are the right matrices, we can combine the middle terms of both, as shown in Equation 9:

$$1/(n-1)\text{diag}(s_1^2, \dots, s_k^2) - n/(n-1)U_k^\intercal \mu\mu^\intercal U_k \quad (9)$$

After solving the SVD of the matrix in Equation 9, which becomes $R\text{diag}(s_1^2, \dots, s_k^2)R^\intercal$, the singular values are essentially those of the sample covariance matrix, and we can compute the singular vectors of the sample covariance matrix by $U_k R$.

It can be seen that when iterating through the steps, we only maintain the accumulated vector x_a, features of the current step, and the TruncatedSVD matrices U_k and $s_1 m \dots, s_k$, all of which have fixed size. Because the sizes of U_k and S_k are $m \times k$ and $k \times k$, respectively, which are independent to the number of samples n. The final step needs to compute the matrix inside the bracket of Equation 8, which is a $k \times k$ matrix and still data-independent. As a result, our algorithm can be executed with a small and fixed memory space, even when be applied to a large dataset.

3.3. GPU-oriented implementation

When implementing the algorithm in Section 3.1 on GPUs, several factors should be considered. First, it could be tricky to implement robust SVD routines from scratch on GPUs, and thus we aim to fully utilize libraries like nVidia cuSolver [8]. cuSolver provides routines that are similar to LAPACK's *gesvd* [9] to compute the SVD of multiple matrices in parallel. Given a high dimensional tensor, these routines consider the last two dimensions as those of the matrices, and apply SVD to all matrices in a single function call. Nevertheless, these APIs could fail when the matrix size exceed 32×32 [8]. In Equation 6, the size of W is $m \times (k + n_b)$, which can easily exceed 32×32.

To resolve this issue, other than computing TruncatedSVD on W, we compute it on $W^\intercal W$ instead. The singular vectors of $W^\intercal W$ are essentially the right singular vectors V' of W, and we can obtain $U'_k \text{diag}(s'_1, \ldots, s'_k)$ by computing $WV'^\intercal$. While the singular values $s'_1, \ldots, s'_k$ are essentially the L2 norm of the projected matrix, and the singular vectors U'_k are its normalized column vectors, we defer the normalization till the finalization step. This is because that the calculation of L2 norm is based on the computing of squared root, which can accumulate numerical errors during the iteration. As our algorithm uses $U_k \text{diag}(s_1, \ldots s_k)$ together during the iteration, as shown in Equation 6, there is no need to separate the singular values $s_1, \ldots s_k$ and singular vectors U_k before the finalization step.

Our algorithm can be implemented by deep learning frameworks by tensorflow [1] and pytorch [14], which can easily utilize GPUs. Besides, by implementing our incremental algorithm in the same framework as the CNN, we can pass the feature vectors of CNN to our algorithm without extra memory copying between CPU and GPUs, which can otherwise dominate the computation. Another benefit of using these frameworks is that they already utilize cuSolver's APIs to compute SVD to multiple matrices in parallel when GPUs are available. The functions of tensorflow and pytorch also can handle matrix larger than 32×32, although the speed can degrade. More details are discussed in Section 5.

4. Experiments

This sections descries our experiment results. Our algorithm is implemented by pytorch and torchvision. We use the CNNs provided by torchvision, which were pre-trained on ImageNet [17]. We used the MVTec AD dataset [2] to benchmark. Our experiments first resized each image to 256×256 pixels by nearest-neighbor sampling, crop the central 224×224 pixels of the resized images, and used the cropped images to test the algorithms.

Our experiments were conducted on one Ubuntu16 computer, and the speed was measured on one Intel Xeon Gold 6230 CPU (2.10GHz) core alone and one nVidia QuadroRTX6000 GPU. The batch size was 8 for all experiments, and we used single precision arithmetic operations. The performance of an experiment was measured by running all classes of MVTec AD, and using the total number of frames of all classes divided by the total time as the FPS. Note that here the timing excludes the I/O time to load images.

Regarding the step size, as the ideal matrix to compute SVD on GPU is 32×32, 32 is the upper limit of step size. It should be noted that during the incremental update, the length of the matrix to computer SVD is the step size plus the dimension to reduce, as shown in Equation 6. To find a balance between the step size and the dimension to keep, our experiments used 16 dimensions, which is the half of 32, and set the step size to 16 too. The exceptions are the last batch, which might not have 16 images, and the first one, which can collect 32 images to run the first SVD.

4.1. Impact of dimension reduction

Before we show the result of our algorithm, we first find a case where dimension reduction can help. While existing works [5, 15] show that dimension reduction methods like PCA can lead to sub-optimal results on MVTec AD when considering all classes, we wonder whether PCA could be beneficial to certain types of objects. If yes, our algorithm could have value. We are especially interested in the case with extremely low dimensions, as the number of dimensions should be lower than or equal to 32 in order to fully utilize GPUs.

Table 1 shows the result of regular PCA on three CNNs, which are ResNet18 [7], WideResNet50 [19], and EfficientNet-b0 [18]. For ResNet18, we used the first three layers to extract the features and reduced to 100 dimensions, which are the same settings used by Defard *et al.* [5]. Our implementation can achieve similar pixel-level ROCAUC as reported by Defard *et al.*, which are 93.7% and 93.5% for texture and object types, respectively.

Table 1 also shows the result of 16 dimensions. Here we show the results with regular PCA, meaning that we collected all features into memory first and then applied PCA in a single function call. It could be seen that when with 16 dimensions, the pixel-level ROCAUC dropped by 7 - 9%. This is also true with WideResNet50, which achieved high pixel-level ROCAUC with 100 dimensions but low ROCAUC with 16 dimensions. It should be noted that for WideResNet50, the features were first sampled to 512 dimensions before applying PCA. Otherwise, the original dimension length is 1792, which requires $56 \times 56 \times 1792 \times 1792$ values and cannot be computed.

Fortunately, when testing on EfficinetNet-b0, we found that with only 16 dimensions, it still can achieve high RO-

Table 1. ROCAUC of PCA on the MTVec AD dataset with various CNNs. Each tuple of numbers represents the accuracies of all classes, texture classes, and object classes of the MVTec AD dataset.

Backbone	ResNet18		WideResNet50 (reduced to 512 dims. first)		EfficentNet-b0	
Layers (1-based)	1,2,3		1,2,3		3,4,5,6	
# Dimensions	100	16	100	16	100	16
Image-level (%)	84.6, 93.8, 79.9	76.6, 89.7, 71.1	81.1, 92.7, 78.3	77.3, 90.7, 69.9	89.9, 99.7, 85.0	87.6, 99.2, 81.8
Pixel-level (%)	94.3, 95.7, 93.8	79.4, 83.3, 85.5	95.3, 96.0, 94.9	84.8, 75.4, 81.4	96.4, 97.2, 96.0	93.4, 95.7, 92.4

Table 2. Results of EfficientNet-b0 on the MTVec AD dataset with different algorithm parameters. The features were reduced to 16 dimensions. The values of Incremental?" mean whether our incremental approach (Y) or regular PCA (N) was used. The values of "GPU Optimized?" mean whether Equation 6 (N) or Section 3.3 (Y) was used to update.

Incremental?		N	N	Y	Y
GPU Optimized?		N	Y	N	Y
Image ROCAUC (%)	Texture	99.2	99.2	99.2	99.2
	Object	81.8	82.5	83.0	81.8
Pixel ROCAUC (%)	Texture	95.7	95.7	95.7	95.7
	Object	92.3	92.3	92.6	92.4
Training Speed (FPS)	ALL	10.1	7.7	7.9	192.8
Inference Speed (FPS)	ALL	187.9	188.7	180.1	192.3

Table 3. Image-level ROCAUC (%) of EfficientNet family on the MTVec AD dataset. The features were reduced to 16 dimensions with our incremental approaches and GPU-oriented optimization.

EfficientNet	0	1	2	3	4	5	6
All	87.6	88.3	89.1	87.4	86.3	85.4	85.7
Texture	99.2	98.4	98.9	97.9	98.3	98.1	98.2
Object	81.8	83.3	84.3	82.2	80.2	79.0	79.4
carpet	99.6	99.8	99.8	99.5	99.6	99.8	99.6
grid	98.8	94.5	97.1	92.7	98.7	97.6	98.3
leather	100.0	100.0	100.0	100.0	100.0	99.7	99.3
tile	98.7	98.3	98.7	98.2	97.9	96.1	97.3
wood	99.0	99.3	98.8	99.0	95.5	97.3	96.6
bottle	96.0	99.1	98.2	98.7	84.4	96.1	94.8
cable	85.8	86.9	86.0	83.9	83.1	86.9	86.2
capsule	87.4	91.4	90.5	90.5	91.1	89.2	89.6
hazelnut	49.5	51.2	56.0	48.8	48.2	49.5	46.4
metal nut	88.4	89.3	90.3	91.4	87.0	59.8	73.9
pill	78.5	84.9	81.0	81.1	83.0	74.4	76.8
screw	53.5	51.9	51.9	46.8	45.6	47.9	46.7
toothbrush	87.2	87.8	96.7	94.2	91.4	91.7	89.4
transitor	97.2	99.2	98.4	96.1	97.3	96.4	95.5
zipper	94.9	91.7	93.8	90.7	91.5	98.1	94.6

Table 4. Pixel-level ROCAUC (%) of EfficientNet family on the MTVec AD dataset. The features were reduced to 16 dimensions with our incremental approaches and GPU-oriented optimization.

EfficientNet	0	1	2	3	4	5	6
All	93.5	93.0	92.6	92.2	91.9	88.1	88.6
Texture	95.7	95.1	94.8	94.8	95.4	89.8	90.9
Object	92.4	91.9	91.5	90.9	90.2	87.2	87.5
carpet	98.9	98.6	98.7	99.1	99.3	95.8	94.8
grid	92.7	90.6	89.4	88.7	92.9	81.8	85.0
leather	99.4	99.4	99.2	99.2	99.2	97.8	98.2
tile	94.2	92.9	92.8	93.5	93.1	85.1	87.9
wood	93.2	94.1	93.9	93.6	92.5	88.3	88.4
bottle	97.3	97.5	96.7	97.0	96.8	94.5	93.5
cable	89.0	88.2	86.8	86.9	84.8	79.5	79.2
capsule	97.7	97.6	97.4	97.5	97.8	95.3	95.2
hazelnut	86.8	85.7	86.0	83.8	81.2	75.6	73.6
metal nut	91.8	89.3	89.6	87.8	87.0	82.5	86.5
pill	84.8	85.2	84.8	81.7	79.8	78.9	80.2
screw	87.5	86.3	85.3	85.7	84.7	82.9	82.3
toothbrush	97.2	97.1	97.2	97.8	98.0	95.3	95.9
transitor	95.4	95.3	95.3	94.6	94.8	93.8	93.8
zipper	96.9	96.6	96.4	96.7	97.0	93.7	94.8

CAUC. Here we use layers 3, 4, 5, and 6 (1-based) because these layers have the features maps in 56×56, 28×28, 28×28, and 14×14 pixels, respectively, which are the same as the first 3 layers of ResNet18 and WideResNet50. Later sections show the results on EfficinetNet-b0 of these layers with our incremental algorithm.

4.2. Speed and accuracy of incremental update

Table 2 show the accuracies and speed of different settings of PCA. The columns with "Incremental?" as "N" mean that the results were measured when the PCA was computed in a single shot after iterating through all data. The columns with "GPU Optimized?" as "N" list the results that were measured by directly applying SVD to matrix in Equation 6, other than using the optimized algorithm in Section 3.3. The results of our proposed method are listed in the rightmost column.

Table 2 lists the accuracies and performance of these

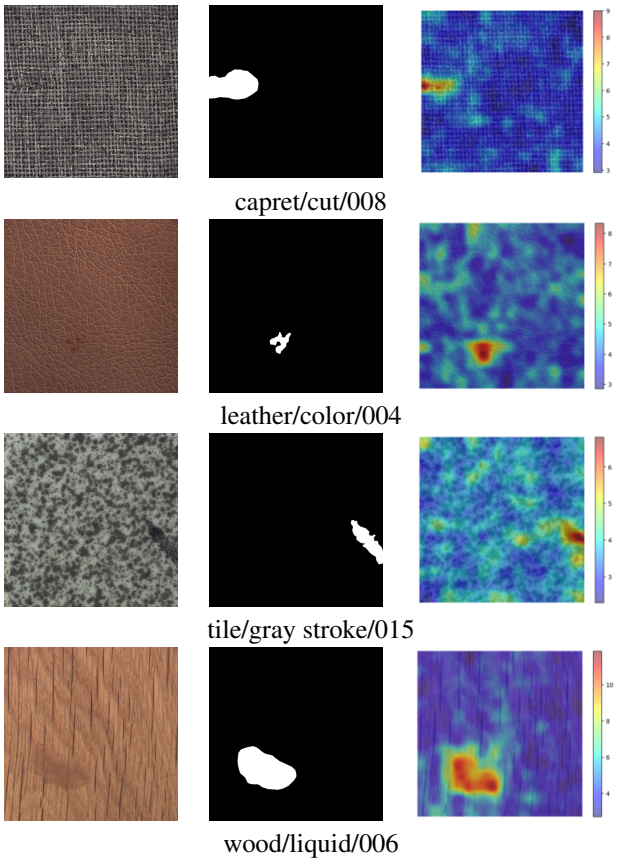

Figure 2. Detected anomaly of the MVTec AD dataset. The columns from left to right show the input images, pixel-wise ground truths, and the heat maps. Each color bar indicates the color mapping of the corresponding heat map. Each caption indicates the object type, anomaly type, and the image name, of the corresponding row.

combinations. While all can achieve similar and high accuracies on texture types, the difference on object types can be apparent. The difference is related to numerical precision, which will be discussed in Section 5. Regarding performance, as suggested by the rightmost column of Table 2, combining incremental update and the GPU-based optimization can achieve highest speed (200FPS), which is 20 times faster than regular PCA (10.1 FPS).

In contrast, collecting all features together and applying the GPU-oriented optimization actually slow down the computation. This is because that the matrices now becomes large than 32×32. This is also true when incrementally updating the singular values and vectors without the GPU-oriented optimization, which is the slowest combination here since the SVD routines were called by multiple steps.

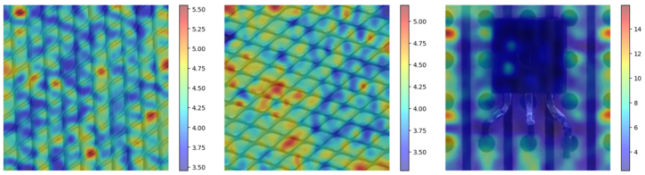

Figure 3. Visualization of false positives of the MVTec AD dataset. The color bars next to the heat maps indicate the color mapping. Each caption indicates the object type, the type of anomaly, and the image name. Here *good* means normal cases.

4.3. Backbone complexity

We also tested different backbones of EfficientNet, as shown in Tables 3 and 4. While all achieved high accuracies on the texture classes, we found that stronger backbones (b4-b6) achieved lower accuracies than weaker ones (b0-b3). If we check the accuracies of texture type alone, b0 actually performed best, as the image-level ROCAUC of all classes are higher than 98.7%. We hypothesize that because these strong CNNs generate longer features, more dimensions are required to preserve the information. In contrast, for MVTec AD, the combination of efficienenet b0 and dimension reduction is a good choice for texture type objects.

4.4. Qualitative study

Figure 2 shows the visualization when testing on samples of the texture types of the MVTec AD dataset. As our algorithm can achieve high accuracies on texture types, the visualizations verifies that the location of the anomaly can be correctly located. On the other hand, as the accuracies on object types were low, we found that the current algorithm is sensitive to changes of local patterns. The left and middle of Figure 3, for instance, show the results of two normal images of type *grid*. Although this is one of the texture types of MVTec AD, the two images were incorrectly detected as anomaly, and the heat maps incorrectly highlight multiple locations in the images. One more example is the right one of Figure 3, which shows the heat map of an abnormal case of the transistor. Although this abnormal image was correctly detected, it was because of the background, not the real anomaly. For instance, areas near the holes in the background were incorrectly highlighted.

We hypothesize that these kinds of false positive can be reduced by using data augmentation during the training. To verify this hypothesis, we tested data augmentation on the object type *grid* with EfficientNet-b0 as follows. During the training, after resizing each image into 256×256 pixels, we randomly cropped 224×224 pixels to extract the features. Table 5 shows the statistics of 100 trials. It can seen that

Table 5. Accuracies with EfficientNet-b0 on class *grid* of the MVTec AD dataset with random cropping. The statistics were measured by 100 trials.

Random cropping?	N	Y		
		Min.	Avg.	Max.
Image ROCAUC (%)	98.8	97.7	98.9	99.7
Pixel ROCAUC (%)	92.7	92.3	92.7	93.2

by average, the image-level ROCAUC was improved from 98.8% to 98.9%.

5. Discussion

5.1. Memory usage

With our dimension reduction algorithm, we can reduce the memory overhead to calculate the anomaly score. When using layers 3-6 of EfficientNet-b0, which leads to features of 256 dimensions, storing th sample covariance matrices in single precision as PaDiM requires $56\times56\times256\times256\times4 = 0.82$GB of memory space. While 0.82GB seems small nowadays, we should remind that this is mainly for texture types of MVTec AD dataset. For other object types, more dimensions are required, and thus a memory-efficient algorithm like ours is still crucial. By reducing the dimensions to 20, our method can reduce the memory usage to $56 \times 56 \times 256 \times 20 \times 4 = 64$MB.

5.2. Impact to the entire system

When considering the entire system, it should be noted that the CNN model also requires memory space to forward the images. Namely, the overall memory usage depends on the model and number of images to forward. In our benchmark, the WideResNet50 of torchvision required 1.3GB of memory space when processing a single image. For computers with limited memory space, even with our incremental algorithm, we still suggest to use light-weight models. Also, since no training of the CNN is involved, it is recommended to set the batch size to 1 when forwarding images through the CNN.

Regarding the impact to speed, we found that with the settings of the rightmost columns of Table 2, our algorithm used 1/3 of the training computation time. This means that without our incremental algorithm and GPU-oriented optimization, the dimension reduction can totally dominate the computation time, making the entire system tens of times slower, as listed by the left three columns of Table 2.

5.3. Numerical precision

As Table 2 shows different accuracies with different settings, we empirically found that the differences were due to numerical precision. Table 6 shows results of the same settings as Table 2 but in double precision instead. It can be seen that now the experiments with and without GPU-oriented optimization can have the same accuracies. It is noteworthy that even in double precision, our incremental algorithm with the GPU-oriented optimization still can achieve 60 FPS for training, while the speed of the other combinations drops to 3 FPS.

Table 6. Results of EfficientNet-b0 with different numerical precision and SVD algorithms. The other experiment settings are the same as those of Table 2.

Incremental?		N	N	Y	Y	Y
GPU Optimized?		N	Y	N	Y	Y
Precision		double				single
SVD Algorithm		gesvdj				gesvd
Image ROCAUC (%)	Texture	99.2	99.2	99.2	99.2	99.2
	Object	82.8	82.8	83.0	83.0	83.0
Pixel ROCAUC (%)	Texture	95.7	95.7	95.7	95.7	95.7
	Object	92.4	92.4	92.6	92.6	92.6
Training Speed (FPS)	ALL	0.8	0.5	1.2	63.8	3.5
Inference Speed (FPS)	ALL	158.3	151.1	156.8	154.5	198.7

Besides numerical precision, the difference is also related to the underlying SVD algorithm. While pytorch by default uses the *gesvdj* routine of cuSOLVER, which is essentially the Jacobi eigenvalue algorithm [11], since pytorch 1.13, it is possible to choose other SVD routines such as *gesvd*[1]. The rightmost column of Table 6 shows the results with *gesvd* as the SVD algorithm in single precision. While the difference between experiments with and without GPU-oriented optimization was also eliminated, the speed became extremely slow.

Based on the findings above, if numerical precision is crucial, we recommend using double precision, which still can achieve 60 FPS with our algorithm. On the other hand, as our method can achieve better accuracies for texture types even with very few dimensions, single precision should be sufficient in such a case.

5.4. Dimensions and step sizes

We also analyzed the impact of dimensions and step sizes, which are summarized in Tables 7 and 8, respectively. It can be seen that when the step size is fixed to 16, the accuracies increase when dimensions increase. In contrast, the impact of step size is less apparent. Based on this finding, we tested with more dimensions to preserve, as shown in Table 9. For each number of dimension, the step size is adjusted so the sum of step size and dimension is fixed to 32. While using 16 dimensions leads to the highest image-level ROCAUC, pixel-level ROCAUC can be further improved

[1] https://pytorch.org/docs/stable/generated/torch.linalg.svd.html

Table 7. Results of EfficientNet-b0 with different numbers of dimensions with identical step size (16).

# Dimensions		4	8	12	16
Image ROCAUC (%)	Texture	99.0	99.2	99.3	99.2
	Object	81.5	80.1	81.7	81.8
Pixel ROCAUC (%)	Texture	95.6	95.5	95.5	95.7
	Object	89.8	90.9	91.8	92.4
Training Speed (FPS)	ALL	190.2	190.8	191.8	192.8
Inference Speed (FPS)	ALL	190.4	188.5	190.4	192.3

Table 8. Results of EfficientNet-b0 with different step sizes with identical number of dimensions (16).

Step size		4	8	12	16
Image ROCAUC (%)	Texture	99.2	99.2	99.2	99.2
	Object	81.5	82.4	82.2	81.8
Pixel ROCAUC (%)	Texture	95.7	95.7	95.7	95.7
	Object	92.4	92.4	92.4	92.4
Training Speed (FPS)	ALL	163.1	178.3	181.2	192.8
Inference Speed (FPS)	ALL	188.9	190.5	186.2	192.3

Table 9. Results of EfficientNet-b0 with different numbers of dimensions. The sum of dimensions and batch size is fixed to 32.

# Dimensions Step size		8 24	16 16	24 8	31 1
Image ROCAUC (%)	Texture	99.2	99.2	99.3	99.3
	Object	81.4	81.8	79.5	80.5
Pixel ROCAUC (%)	Texture	95.5	95.7	96.2	96.6
	Object	90.9	92.4	93.2	93.7
Training Speed (FPS)	ALL	201.0	192.8	177.1	84.6
Inference Speed (FPS)	ALL	192.2	192.3	184.5	188.9

by increasing the dimensions. Meanwhile, it can be seen that when the dimension increases, implying a smaller step size, more steps were required and thus the training speed became slower.

To further verify the impact of dimensions, we tested another dataset Magnetic Tile [10]. Similar to MVTec AD, we resized the images to 256×256 pixels and used the central 224×224 pixels to extract the features. Among the normal images, we use the images with labels *exp1* for testing and the others for training. We fixed the step size to 16. Table 10 shows the accuracies and speed with different numbers of dimensions. It can be seen that when using only 16 dimensions, both image-level and pixel-level ROCAUC were limited to around 75%. When using more dimensions, the image-level ROCAUC was apparently improved. On the other hand, the training speed was hugely reduced since the sum of dimension number and step size exceeded 32. A future work is optimizing our algorithm to support more dimensions without impacting the speed on GPUs.

Table 10. Results of EfficientNet-b0 on the dataset Magnetic Tile [10].

# Dimensions Step size	16 16	32 16	48 16	64 16
Image ROCAUC (%)	75.8	80.5	86.9	89.6
Pixel ROCAUC (%)	75.0	74.3	74.9	75.8
Training Speed (FPS)	267.8	5.4	5.3	3.3
Inference Speed (FPS)	221.2	215.7	212.8	203.3

6. Conclusion

This paper presents an incremental dimension reduction method to compute the truncated PCA. This algorithm is designed when using CNN-based features for visual anomaly detection and especially optimized for GPUs. When the dimensionality is low, which is still sufficient to detect anomaly in certain types of objects, we can achieve 100s of FPS on a single GPU, and our method is memory efficient during both the training and testing stages. In the future, we would consider other types of backbones like transformers [12] and non-linear score calculation such as K-nearest-neighbor. A target is applying our dimension reduction approach to reduce the memory bank of PatchCore [16], which is required when a single Gaussian distribution is insufficient to describe the normal data. We would also like to extend our method to supervised or semi-supervised cases [20], and integrate with other error metrics [13].

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 4

[2] Paul Bergmann, Kilian Batzner, Michael Fauser, David Sattlegger, and Carsten Steger. The mvtec anomaly detection dataset: A comprehensive real-world dataset for unsuper-

vised anomaly detection. *International Journal of Computer Vision*, 129:1038–1059, 2021. 1, 2, 3, 4

[3] M. Brand. Incremental singular value decomposition of uncertain data with missing values. In *ECCV*, 2002. 1

[4] M. Brand. Fast low-rank modifications of the thin singular value decomposition. *Linear Algebra and Its Applications*, 415(1):20–30, 2006. 1

[5] Thomas Defard, Aleksandr Setkov, Angelique Loesch, and Romaric Audigier. Padim: a patch distribution modeling framework for anomaly detection and localization. *arXiv:2011.08785*, 2020. 1, 2, 4

[6] Per Christian Hansen. The truncatedsvd as a method for regularization. *BIT Numerical Mathematics*, 1987. 2

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 4

[8] https://docs.nvidia.com/cuda/cusolver/index.html. *cuSOLVER API Reference*. nVidia Corporation. 4

[9] https://netlib.org/lapack/. *LAPACK: Linear Albegra PACKage*. 4

[10] Yibin Huang, Congying Qiu, Yue Guo, Xiaonan Wang, and Kui Yuan. Surface defect saliency of magnetic tile. In *CASE: Proceedings of the IEEE International Conference on Automation Science and Engineering*, 2018. 8

[11] C.G.J. Jacobi. *Crelle's Journal*, 1846(30):51–94, 1846. 7

[12] Pankaj Mishra, Riccardo Verk, Daniele Fornasier, Claudio Piciarelli, and Gian Luca Foresti. Vt-adl: A vision transformer network for image anomaly detection and localization. In *ISIE 2021: Proceedings of the IEEE International Symposium on Industrial Electronics*, 2021. 8

[13] Ibrahima J. Ndiour, Nilesh A. Ahuja, and Omesh Tickoo. Subspace modeling for fast out-of-distribution and anomaly detection. In *ICIP*, 2022. 8

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeruIPS*. 2019. 4

[15] Oliver Rippel, Patrick Mertens, and Dorit Merhof. Modeling the distribution of normal data in pre-trained deep features for anomaly detection. *arXiv:2005.14140*, 2020. 1, 2, 4

[16] Karsten Roth, Latha Pemula, Joaquin Zepeda, Bernhard Schölkopf, Thomas Brox, and Peter V. Gehler. Towards total recall in industrial anomaly detection. *arXiv:2106.08265*, 2021. 1, 8

[17] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. 1, 4

[18] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling forconvolutional neural networks. In *ICML*, 2019. 2, 4

[19] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *BMVC*, 2016. 1, 4

[20] Yang Zou, Jongheon Jeong, Latha Pemula, Dongqing Zhang, and Onkar Dabeer. Spot-the-difference self-supervised pretraining for anomaly detection and segmentation. In *ECCV*, 2022. 8