Learning A Wafer Feature With One Training Sample

Yueling (Jenny) Zeng, Li-C. Wang
University of California, Santa Barbara
Santa Barbara, California 93106

Chuanhe (Jay) Shan
IE3A, Inc.
Los Angeles, California 90017

Nik Sumikawa
NXP Semiconductor
Chandler, AZ 85224

***Abstract*—In this work, we consider learning a wafer plot recognizer where only one training sample is available. We introduce an approach called Manifestation Learning to enable the learning. The underlying technology utilizes the Variational AutoEncoder (VAE) approach to construct a so-called Manifestation Space. The training sample is projected into this space and the recognition is achieved through a pre-trained model in the space. Using wafer probe test data from an automotive product line, this paper explains the learning approach, its feasibility and limitation.**

1. Introduction

Analysis of wafer plots derived from test results is an area of study in test data analytics. One capability desired in practice is to automatically recognize wafer plots exhibiting a certain feature. Such a *recognizer* can be deployed to monitor a production line and identify wafers containing the specific feature. In this context, the work in [1][2] developed an automatic wafer plot recognition software.

In [1], a wafer plot feature is called a *concept*. A *concept recognizer* is a model to identify wafer plots containing the feature. In contrast to "classifier", the term "recognizer" is to emphasize that the model is for separating (a few) in-class plots from (many) out-of-class plots. In [1], the approach to learn a recognizer is based on the popular Generative Adversarial Network (GAN) [3][4][5].

One key reason to use GAN was because it enabled training with very few in-class samples. In [1], training a recognizer can be done with five in-class samples. GAN training can be tricky though [4][5], and the robustness of a GAN-based recognizer can be a concern in practice. Hence, the work in [2] employs tensor computation based techniques [6][7][8] to improve robustness of the recognition.

Once deployed, the recognition software [2] would automatically decide the need to train a new recognizer, select 5 training samples to represent the concept, and train a recognizer. However, if there were less than five training samples found for a feature, the software would simply not act. Initially, this limitation motivated us to pursue this work.

For example, Figure 1 shows a wafer plot with a "ring" feature which was not captured by the recognition software as a concept. Note that in this plot, a yellow dot shows the location of a failing die. The data was from the same automotive SoC line as that reported in [2].

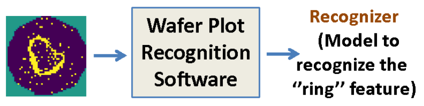

Wafer Plot Recognition Software
Recognizer
(Model to recognize the "ring" feature)

Figure 1. Goal: Learning a recognizer with one training sample

1.1. The learning problem

The learning problem can be stated as the following: We are given a single sample s exhibiting a *feature* of interest. The goal is to build a model M_s such that given future samples $s_1, \ldots, s_n$, $\forall s_i$, $1 \leq i \leq n$, $M_s(s_i)$ would return "yes" if s_i exhibits the same feature as s.

In machine learning, a closely-related problem is the *one-shot learning* [9] which also intends to learn with one training sample. One-shot learning is a form of *transfer learning* [10]. To apply one-shot learning, we start with a model $M()$ which is already learned to classify wafer plots into different categories based on their feature. The single training sample s is given to represent a *new* category. Then, the model $M()$ is enhanced based on the new sample.

In our work, the requirement is different though. For each category, we intend to build a recognizer independent of other recognizers. Hence, the learning has to "start from scratch". The reason we prefer independent models is to facilitate debugging and maintenance of the recognition software. If there is a deficiency in recognition of one category of plots, we can simply focus on that particular recognizer without concerning other recognizers.

1.2. Two elements in a learning algorithm

In the above problem statement, the exact meaning of the word "feature" is to be defined by a learning algorithm. Typically, s_i and s are treated as having the same feature if they are *similar*. However, different algorithms might measure this *similarity* differently.

A learning algorithm of this sort essentially comprises two elements. The first can be thought of as a projection method that maps each sample to a score. For example, a score indicates how close a given sample s_i is to be considered as the same category of the training sample s. For the most part, this projection method is the learning algorithm. However, just obtaining the scores is not enough. To decide a sample s_i should be treated as in-class or out-of-class, a *threshold* would also be needed.

978-1-7281-9113-3/20/$31.00 ©2020 IEEE

In practice, threshold can be decided by extensive experiment. However, this is only feasible if sufficient data are available. If not, the problem can become challenging.

1.3. Challenge on deciding a threshold

With only one training sample, it is not surprised that threshold selection is a challenge. Take the wafer plot shown in Figure 1 as an example. To accomplish the first element, i.e. the projection method, we can simply choose a *similarity measure* that computes a similarity score between two wafer plots. For example, suppose each plot is represented as a 48×48 matrix. Each entry contains one of the three values -1 (purple), 0 (green), and $+1$ (yellow).

Given two wafer plot matrices W_1, W_2, a simple way is to use their square distance, $\| W_1 - W_2 \|^2$, i.e. subtracting each entry individually and take the sum of their square. Another way is to focus on the yellow entries ("+1" entries) where the feature is defined. First we replace every "−1" with "0" to make the matrix binary. Let U be the number of "1" entries in $W_1 \vee W_2$. U captures the size of the union set of the yellow dots. Then, let I be the number of "1" entries in $W_1 \wedge W_2$. I captures the size of the intersection set of the yellow dots. We let the similarity score be $\frac{I}{U}$. We use $IoU(W_1, W_2)$ to denote this score.

In our dataset, there are six plots with the "ring" feature. These six plots are shown in Figure 2. We treat them as the *in-class*. Then, if any other plot considered by a recognizer as in-class, it is treated as a false positive.

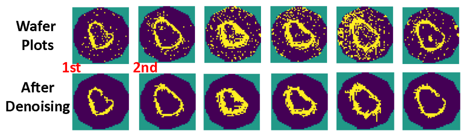


Figure 2. The six plots with "ring", before and after denoising

Before applying $IoU()$, each wafer plot goes through a denoising step which removes random and scatter yellow dots while maintaining the feature. This is typical in image processing. Figure 2 shows the images before and after the denoising step. In the rest of the paper, all wafer plots shown are after the denoising.

Let W be the first wafer plot shown in Figure 2. Suppose it is the one training sample given to us. First, we have $IoU(W, W) = 1$. For any other W_i, unless $W_i = W$ exactly we have $IoU(W, W_i) < 1$. The minimum IoU score is 0.

Suppose we have 25 other wafers to help in finding a threshold. Figure 3-(a) shows the IoU values for the 25 wafers. The values are shown along the y-axis. Note that for ease of viewing, the blue dots are spread out along the x-axis direction which has no physical meaning.

Suppose we know all blue dots are out-of-class samples. There can be two (reasonable) options to set the threshold. Threshold $T_1 = 0.0653$ is set by the blue dot with the highest score. Another option is to set the threshold at the middle point, i.e. $T_2 = \frac{1+0.0653}{2} = 0.53265$. Note that setting a threshold close to the red dot is not an option in this plot because we would not know how close is "close".

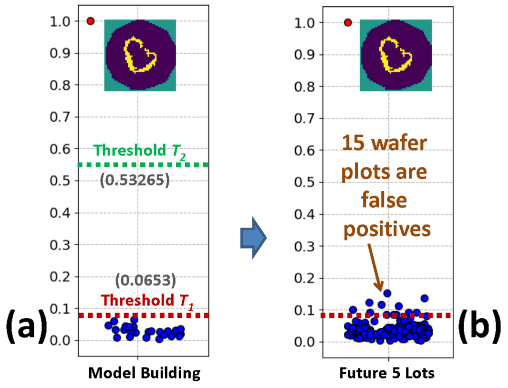


Figure 3. Model building: Selecting a threshold

Suppose we apply the model to check on future 125 wafers where none is in-class. Figure 3-(b) shows the result. If T_1 is taken, it will result in 15 false positives. If T_2 is taken, it will have no false positive. But result in Figure 4 below shows that taking T_2 will miss all the in-class plots.

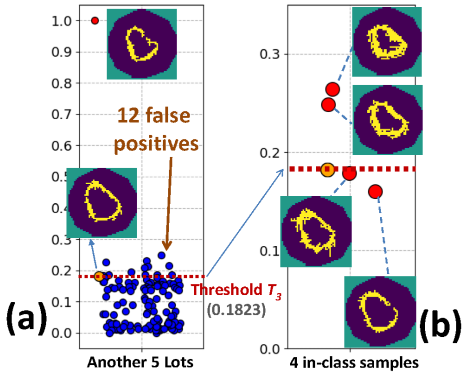


Figure 4. What if adding a second training sample

Suppose the second wafer plot in Figure 2 is also available to us. Figure 4-(a) shows its IoU score. This score, 0.1823, now can be our third threshold option, T_3. The figure shows that if a different set of 125 wafer plots is considered, T_3 would result in 12 false positives.

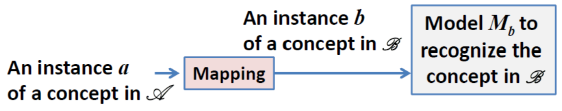
Figure 4-(b) shows the IoU scores for the remaining four in-class plots. Observe that even with T_3, the model would have treated two in-class plots as out-of-class (below the threshold line). If threshold $T_2 = 0.53265$ from Figure 3 is used, the model would have missed all in-class plots.

The above example illustrates the challenge for deciding a threshold when limited data is available. Unless the data show where the highest blue dot (out-of-class) or the lowest red dot (in-class) might be, there seems no effective way to optimize a model against false positive or false negative.

In the above example, we assume there are out-of-class samples to help decide the threshold. Even this assumption is considered unrealistic in our work. Suppose our software is deployed already. Then, the learning depicted in Figure 1 needs to be carried out online. Without acquiring labels for other samples through another means, the software would not know if they are in-class or out-of-class, i.e. the learning cannot simply treat unlabeled samples as out-of-class.

2. Manifestation Learning

Suppose a sample s is given to represent a concept in domain $\mathscr{A}$. The idea of *Manifestation Learning* is that, we would try to recognize the concept by using a pre-trained recognizer M_b for a different concept in a different domain $\mathscr{B}$. The recognizer M_b is trained with a dataset originated from the alternative domain, where there are plenty of training samples available.



An instance b of a concept in $\mathscr{B}$
Model M_b to recognize the concept in $\mathscr{B}$
An instance a of a concept in $\mathscr{A}$
Mapping

Figure 5. The basic idea of Manifestation Learning

Figure 5 illustrates the idea. Given an instance to be recognized for a concept in $\mathscr{A}$, we simply *manifest* the instance into a concept in $\mathscr{B}$ domain and use M_b for the recognition. To realize this idea, at minimal requires building a mapping function that maps every instance in $\mathscr{A}$ to an instance in $\mathscr{B}$. Note that like one-shot learning, manifestation learning can also be thought of as a special form of *transfer learning* [10].

2.1. The manifestation space

Recent advances in the AutoEncoder (AE) approach [11], in particular the Variational AutoEncoder (VAE) [12], provide a means for realizing the proposed idea. An AE has three essential components: an encoder function, a probabilistic decoder model, and a coded *latent space*.

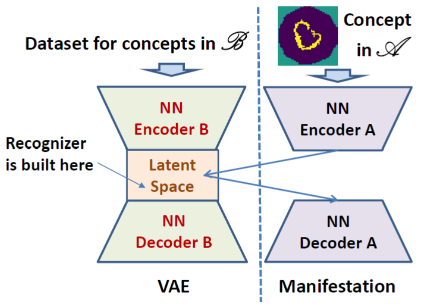


Figure 6. Setup for Manifestation Learning

Figure 6 illustrates our setup to realize the manifestation learning idea. Given a dataset for learning about a concept in $\mathscr{B}$ (which contains many in-class and out-of-class samples), an encoder and a decoder are trained. The encoder and decoder in the setup are both neural network models. For an input sample x, the encoder maps x onto a point in the latent space. This point essentially can be seen as a *code* $\vec{z}$ for x. For example, if the latent space is defined with 16 dimensions, $\vec{z}$ is a vector of size 16. Then, the decoder reconstructs the input x from the code $\vec{z}$. A main training objective is to minimize the reconstruction error.

After the training, each sample in the dataset is mapped to a code in the latent space. In manifestation learning, one sample s for a concept in $\mathscr{A}$ is given. This sample is used in training an encoder and a decoder separately. This manifestation encoder maps s to a selected code in the latent space. This selected code corresponds to a particular sample from a selected concept in $\mathscr{B}$. The decoder then reconstructs the sample from the code. The training objective for the decoder is also to minimize the reconstruction error. However, the training objective for the encoder is different from before, i.e. now the objective is to map s to a selected code.

For a selected concept in $\mathscr{B}$, its recognizer is built in the latent space where each sample is represented as a vector. The recognizer can be built using a traditional learning technique such as SVM [13]. More importantly, there are sufficient data for learning the model. Detail will be discussed later in Section 3.

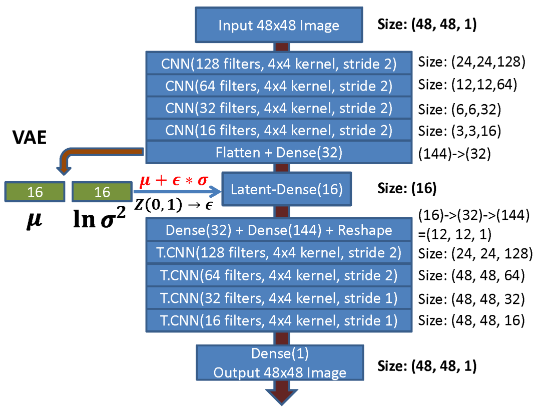
In Figure 6, we call the latent space as a *Manifestation Space*. This is to differentiate our use of the space from that of traditional AE and VAE.

2.2. VAE implementation

VAE replaces the deterministic encoder function in AE with a posterior model $q(\vec{z}|x)$. The effect is that, instead of learning a code for a sample as a vector in the latent space, VAE learns codes represented as probabilistic distributions in the space. When each sample is represented as a probability distribution, effectively the latent space becomes continuous. This enables information to be learned between two sample points and allows samples to be drawn randomly from the latent space.

Note that our VAE implementation did not follow the original VAE [12] which uses KL divergence to regularize learning of the latent space. As indicated in a number of works [14][15], this can lead to an uninformative latent code and also the tendency to over-fit the samples, especially on a small dataset. We use the InfoVAE proposed in [16] which uses Maximum Mean Discrepancy (MMD) [17] to measure divergence and regularize the latent space.

2.3. The neural networks



Input 48x48 Image
Size: (48, 48, 1)
CNN(128 filters, 4x4 kernel, stride 2)
Size: (24,24,128)
CNN(64 filters, 4x4 kernel, stride 2)
Size: (12,12,64)
CNN(32 filters, 4x4 kernel, stride 2)
Size: (6,6,32)
CNN(16 filters, 4x4 kernel, stride 2)
Size: (3,3,16)
Flatten + Dense(32)
(144)->(32)
VAE
16
16
μ
$\ln \sigma^2$
$\mu + \epsilon * \sigma$
$Z(0,1) \rightarrow \epsilon$
Latent-Dense(16)
Size: (16)
Dense(32) + Dense(144) + Reshape
(16)->(32)->(144) =(12, 12, 1)
T.CNN(128 filters, 4x4 kernel, stride 2)
Size: (24, 24, 128)
T.CNN(64 filters, 4x4 kernel, stride 2)
Size: (48, 48, 64)
T.CNN(32 filters, 4x4 kernel, stride 1)
Size: (48, 48, 32)
T.CNN(16 filters, 4x4 kernel, stride 1)
Size: (48, 48, 16)
Dense(1) Output 48x48 Image
Size: (48, 48, 1)

Figure 7. Convolutional Neural Networks for VAE

Figure 7 illustrates the architecture of the neural networks for implementing VAE. Convolutional neural networks (CNN) are used to implement the encoder and decoder. The encoder comprises four convolutional layers denoted as "CNN()". The decoder comprises four transposed convolutional layers denoted as "T.CNN()". The input image is a $48 \times 48 \times 1$ tensor, i.e. a 48×48 image with one channel. Unlike a RGB image with three channels, our image has one channel with three possible values, -1, 0, and +1.

The first CNN layer essentially converts the input image into a 24×24 image with 128 channels, i.e. a $24 \times 24 \times 128$ tensor. Each 24×24 matrix is the result of applying a *filter* on the original input image. Each filter uses a 4×4 kernel (window) to "scan" the input image. The stride length is 2, i.e. during scanning the window moves every 2 pixels. The effect is to reduce the original image size by half.

Another way to understand the CNN layer's operation is by thinking in terms of a *convolutional matrix* [18], which is defined by the kernel and the stride length. In the example, the convolutional matrix can be denoted as $C_{576 \times 2304}$ where $576 = 24 * 24$ and $2304 = 48 * 48$. The 48×48 input image matrix is flatten into a column vector $\vec{v}_{2304}$. Then the convolution layer computes the product $C_{576 \times 2304} * \vec{v}_{2304}$. This product is a vector of 576 values which can then be reshaped into a 24×24 matrix.

Similarly, the other three CNN layers below the first CNN layer further reduces the image size, from 24×24 to 12×12, then to 6×6, and then to 3×3. The number of channels is equal to the number of filters used in each layer. Each filter is a $i \times i \times k$ tensor where $i \times i$ is the kernel size and k is the number of channels from the previous layer.

After four CNN layers, the result is a 16-channel 3×3 image, i.e. a $3 \times 3 \times 16$ tensor. This tensor is flatten into a vector of size 144. The last layer of the encoder is a fully connected (FC) layer that connects 144 inputs to 32 outputs.

The latent space has 16 dimensions. If it were AE, another FC layer would have been used, to reduce the size from 32 to 16. In training, each sample would be mapped into a vector of size 16 in the latent space.

In VAE, two 16-dimensional vectors are used, denoted as μ and $\ln \sigma^2$. In each training cycle, a multivariate constant ϵ is randomly drawn from the distribution $Z(0, 1)$, the 16-dimensional Gaussian distribution with mean 0 and standard deviation 1. Then, $\mu + \epsilon * e^{\frac{\ln \sigma^2}{2}}$ is used as the resulting vector in the latent space (Note that $e^{\frac{\ln \sigma^2}{2}} = \sigma$). Effectively, each sample would correspond to a probability distribution in the latent space. This distribution is assumed to be Gaussian, represented by its mean vector μ and the vector $\ln \sigma^2$. Note that the reason to use $\ln \sigma^2$ instead of its standard deviation σ directly is a trick to enable the training [12].

The decoder takes a 16-dimensional vector from the latent space and produces a 48×48 image. First, it uses a 16-to-32 FC layer, followed by a 32-to-144 FC layer to expand the 16-dimensional vector to a vector of size 144. The vector is then reshaped into an 12×12 image with 1 channel. A transposed convolutional layer is similar to a convolutional layer except that instead of using a convolutional matrix, a T.CNN uses the transpose of a convolutional matrix.

For example, in CNN suppose we multiply a convolutional matrix $C_{144 \times 576}$ to the flatten vector of a 24×24 matrix. This gives a vector of size 144, equivalent to a 12×12 matrix. In T.CNN we would multiply $(C_{144 \times 576})^T$ to the flatten vector of a 12×12 matrix. This gives a vector of size 576, equivalent to a 24×24 matrix.

The last T.CNN layer produces a 48×48 image with 16 channels. A FC layer is then used to convert the 16 channels into 1 channel. In VAE training, the reconstruction error, i.e. the difference between the input image and the output image, is minimized.

2.4. Example of manifestation space

TensorFlow [19] is used to implement the neural networks shown in Figure 7. One of the datasets we experimented was from the MNIST handwritten digit database [20]. The dataset comprises samples for individual digits "0" to "9". In our study, for each digit we had >5000 samples. Figure 8 shows ten examples for digits "0", "1", and "2".

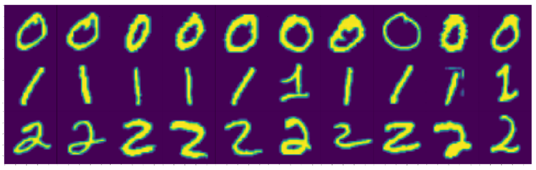


Figure 8. Examples of "0", "1", and "2" from the MNIST dataset

We took 5000 samples for each of the three digits to form a dataset of 15K images. We trained a VAE model with these 15K images. We were interested in how the samples distributed in the latent space.

The latent space has 16 dimensions. Hence, we used the t-SNE technique [21] to project the latent space onto a 2D space for ease of viewing. Figure 9 shows the result. For clarity, only 100 samples from each class are randomly selected to be shown. It is interesting to observe that samples from each digit exhibit a cluster in the latent space.

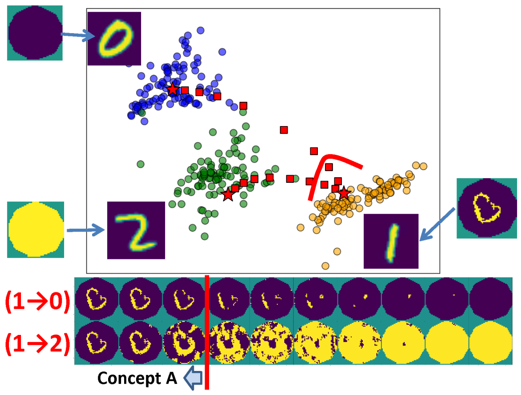


Figure 9. Manifestation space example

Recall that in VAE, each sample in the latent space corresponds to a probability distribution. Hence, it is important to note that the points shown in Figure 9 is the result by randomly drawing an ϵ (see Figure 7) for each sample. As a result, if we tried to re-generate the plot in Figure 9 with another run, the exact location of a sample's point could change due to the random drawing. However, exhibition of the three clusters would always be there.

It is also important to note that during VAE training, the images were used without their label. Hence, the training is unsupervised. Therefore, the clustering property shown in Figure 9 is the result of the VAE training. Note that this property might not be there if a different dataset or a different learning setup is used. Later in Section 3 we will discuss this property in more detail.

Suppose we want to map a wafer plot to a sampled point in the latent space. As shown in Figure 6, we train a separate encoder and decoder (Encoder A and Decoder A in the figure). As shown in Figure 9, this is done for three wafer plots: the first "ring" wafer plot shown in Figure 2, a generic no-fail ("blank") plot, and a generic all-fail ("full") plot. Each plot is mapped to the "center" of a digit cluster. In Figure 9, the centers are marked by a red star "⋆".

To check how well the manifestation decoder works, Figure 9 also shows the sequence of images produced by two *walks*. The first walk moves from digit "1" to digit "0". Observe that the resulting images gradually change from "ring" to "blank". The second walk moves from digit "1" to digit "2". Also observe that the resulting images gradually change from "ring" to "full".

In the two walks, observe that the first three plots all exhibit the "ring" feature. They can all be treated as examples of concept $\mathscr{A}$, i.e. concept "ring". In the latent space, those "ring" plots correspond to points closer to the center of the digit "1" cluster. If we could define a region around the digit "1" cluster, in Manifestation Learning this region could also be used to define a region for concept $\mathscr{A}$. Then, recognizing concept $\mathscr{A}$ becomes simply to check if an input image, after going through the manifestation encoder, results in a point inside the concept region or not.

2.5. Another example of manifestation space

Before we discuss how to model a *concept region* in a manifestation space, in this section we show another example of manifestation space. This example shows some generality of the Manifestation Learning idea.

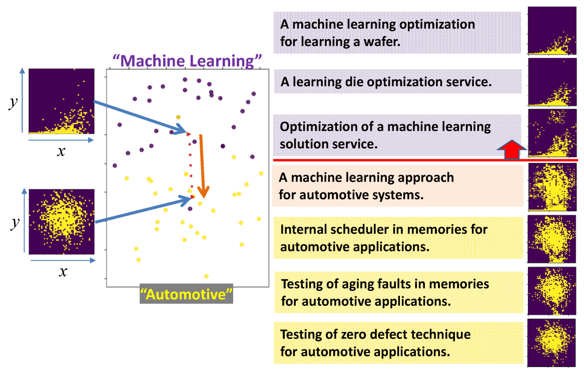

"Machine Learning"
"Automotive"
y
x
y
x
A machine learning optimization for learning a wafer.
A learning die optimization service.
Optimization of a machine learning solution service.
A machine learning approach for automotive systems.
Internal scheduler in memories for automotive applications.
Testing of aging faults in memories for automotive applications.
Testing of zero defect technique for automotive applications.

Figure 10. A sentence-based manifestation space

Figure 10 shows a manifestation space learned based on 60 paper titles from past International Test Conference (ITC), where 29 titles are categorized as "Machine Learning" and 31 titles are categorized as "Automotive" (two tracks of ITC in recent years). We use the VAE approach proposed in [22] to train a VAE model, where the CNNs in Figure 7 are replaced with LSTMs [23].

Then, we take two plots for training the manifestation encoder and decoder, which follow the network architecture shown in Figure 7. The first plot (on the top) is the same correlation plot shown as Figure 3-(a) in paper [1] before. The x-axis is the average e-test value and the y-axis is the number of fails. Each yellow dot represents a wafer lot. It is a correlation plot example used in [1] to show that, as the e-test value becomes larger, it tends to result in more fails. The second plot is a manually-created, generic counter example showing no correlation between x and y.

The manifestation encoder is trained to map the correlation plot to the center of the "Machine Learning" cluster. The no-correlation plot is mapped to the center of the "Automotive" cluster.

As before, by walking a path from "Machine Learning" to "Automotive" in the manifestation space, we can generate various plots using the manifestation decoder. In addition, we can generate various paper titles using the VAE's decoder. Note that in the figure, none of the generated titles is the same as the original titles used in training the VAE.

It is interesting to observe that the top three titles are about "Machine Learning". The bottom three are about "Automotive". The middle one is a hybrid. Also observe that the top three plots all show a similar correlation trend. Suppose we have a model that can recognize a paper title of "Machine Learning" category. Then, this model could also be used to recognize plots with a correlation trend.

3. Concept Region

To model a *concept region* in a manifestation space, we need to decide on the following four aspects:
(*Dataset*): We need to choose a labeled dataset that comprises multiple classes of samples, where for each class there are sufficient samples for the VAE training.
(*Learning Setup*): To learn a model in the manifestation space, we need to choose to employ a supervised learning or an unsupervised learning setup.
(*Algorithm*): Based on the learning setup, we choose a particular learning algorithm.
(*Target Class*): Through experiment, we choose a particular class as our *target class* for modeling the concept.

3.1. Selection of the dataset

To choose a dataset, we considered various options. Because our goal is to capture a concept based on a wafer plot, it would seem intuitive to consider using a dataset comprising multiple classes of wafer plots. For example, the work in [2] shows 11 classes of wafer plots found in the dataset with 8300 wafers. After investigating the possibility, we decided not to use the dataset because the numbers of wafer plots across the 11 classes differ significantly. Some have much fewer samples while the largest class has over a hundred. The imbalance and insufficiency in terms of the number of samples are of the major concern.

One crucial property considered for a manifestation space is that multiple classes of samples exhibit clustering as that shown in Figure 9. This is important because it enables us to model a target class region more easily.

We also considered using the CIFAR-10 dataset [24]. However, while we could train a VAE model with a good reconstruction accuracy, we found it more difficult to observe the desired clustering property in the latent space. This was true even with using only two classes of samples.

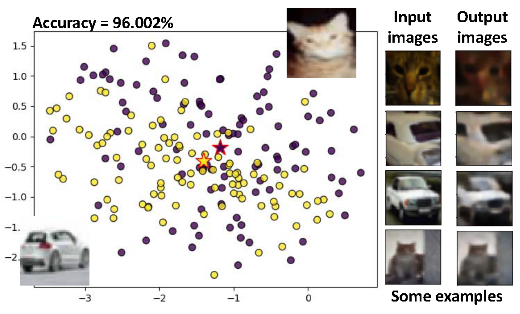


Figure 11. Latent space shows no clustering with CIFAR-10 samples

For example, Figure 11 shows the latent space with 100 samples each from two classes of CIFAR-10: cat and automobile. The training is based on 5000 samples in each class. The reconstruction accuracy is 96.002%. The dimension of the latent space is set at 1024 to achieve the accuracy. As seen, points from the two classes show no clustering.

Figure 10 shows another option based on a sentence-based manifestation space. While it is interesting, we found that training with the sentence-based VAE [22] could be much trickier than training a CNN-based VAE.

After all the experiments, we settled with the MNIST handwritten digit dataset [20]. The dataset we used comprises 10 classes of image samples. Each class has 5000 samples. The dataset is balanced and also has a sufficiently large number of samples from each class.

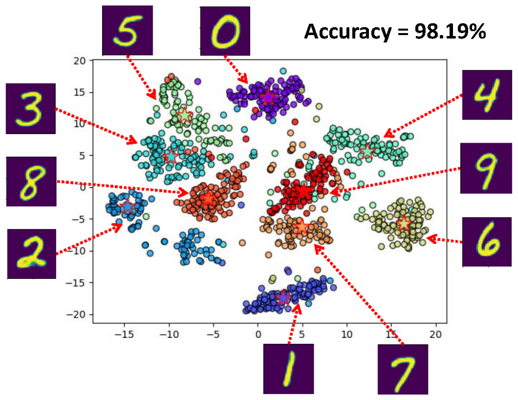


Figure 12. Clusters in the manifestation space with MNIST samples

Similar to Figure 9, Figure 12 shows the 2D projection plot of the latent space. Again, the figure shows 100 samples for each digit. As seen, samples of the same digit are clustered. In terms of the clustering property, this was the best result observed in all of our experiments. Hence, we chose this space as our manifestation space.

3.2. Learning setup and the algorithm

Each sample in the MNIST dataset is labeled by its digit. From the manifestation space, we can obtain a dataset where each sample is represented as a vector of size 16. Each vector has a label and hence intuitively, to learn a concept region for a target class, say digit “1”, we can treat it as a supervised binary classification problem, i.e. to treat “1” as one class and the rest of digits as the other class.

After some initial study, we decided not to follow this approach. The main reason is that we desire our model for a concept region to have no false positive (This property will be utilized later in Section 5). A supervised learning algorithm usually is not designed to ensure no error with respect to a given class. Hence, in our implementation we chose to treat it as an unsupervised learning problem.

Suppose we want to learn a model to capture the concept region for the digit “1” in the manifestation space. With an unsupervised setting, when learning the model we would only use the samples of digit “1”. The learning algorithm we chose was the SVM one-class algorithm [13], in particular the implementation from the Scikit-Learn package [25].

To run SVM one-class, user needs to supply value to the parameter called ν (“nu”). The ν parameter provides an upper bound on the fraction of samples that are classified as out-of-class by the model. Note that in an actual implementation, this bound is not strictly followed because of the use of a soft margin, i.e. if a sample falls outside but very close to the decision boundary (within a margin), it would not be considered as a violation of the bound.

For example, Figure 13 shows the result by running SVM one-class on the 5000 samples of digit “1”. We set ν = 0.01. This means that the model should classify no more than 50 samples as outliers. The model comprises two elements: (1) for each sample a *score* is calculated, and (2) a threshold is provided to classify samples into in-class (above the threshold) and out-of-class (below). Note that in SVM this threshold is always set at the value 0. However, the Scikit-Learn implementation moves this threshold by a positive bias to facilitate plotting with a log scale.

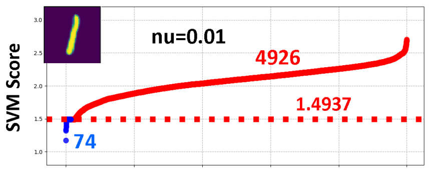


Figure 13. Resulting SVM scores on digit “1” samples

As seen in Figure 13, 74 samples are considered out-of-class, i.e. below the threshold. In our context, we can interpret this as a recognition model capturing a concept region that includes 4926 samples of digit “1”.

Next, we apply the digit “1” model to 5000 samples of digit “0”. Figure 14 shows the result. While the model does leave 4977 samples outside the concept region, it includes 23 samples as in-class. These 23 samples are false positives. As mentioned above, we desire no false positive for a model. Hence, this model would not work.

3.3. Target class selection

To search for a model that works, we use all the out-of-class samples as the *validation set*. The search is based

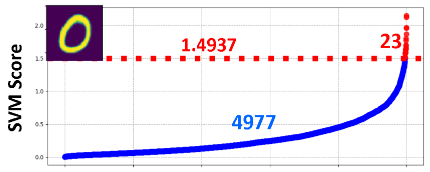


Figure 14. Applying the SVM model on digit “0” samples

on changing ν. We would search for a minimal ν value that results in a model with no false positive, i.e. the model treating all samples in the validation set as out-of-class.

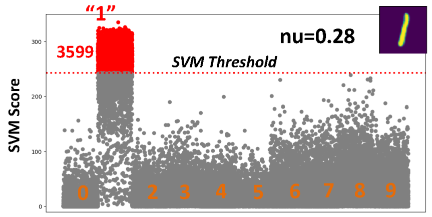


Figure 15. SVM scores by the digit “1” model, across all samples

Figure 15 shows the result based on the model found with ν = 0.28 for digit “1”. The model includes 3599 samples of digit “1” as in-class. The model classifies *all* samples from other digits as out-of-class.

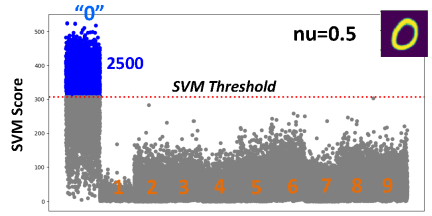


Figure 16. SVM scores by the digit “0” model, across all samples

In contrast, Figure 16 shows the model for digit “0”. In order to achieve no false positive, the ν is set at 0.5, resulting in excluding half of the samples of digit “0” from the concept region (below the threshold).

TABLE 1. THE RESULTING ν FOR THE TEN DIGIT CLASSES

0	1	2	3	4	5	6	7	8	9
0.50	0.28	0.82	0.89	0.88	0.94	0.64	0.88	0.79	0.79

Table 1 shows the ν values found across the ten digit classes where the resulting model has no false positive. The search started from ν =0.01 with an increment of 0.01. The result shows that samples of digit “1” provide the best model, meaning that the model captures the largest number of in-class samples in the concept region.

This result is consistent to the fact that the cluster of digit “1” is more separated from others, which can be somewhat observed in Figure 12. Suppose for each cluster we calculate its “center” which is the average of all the sample vectors in the manifestation space. For each digit, we calculate the average (Euclidean) distance from its center to the other 9 centers. Table 2 shows such average distance for each digit. As seen, digit “1” has the largest average distance to others.

More separation of a cluster to other clusters enables building a better model, i.e. capturing more in-class samples inside the concept region. Because of this observation, we chose digit “1” as our *target class* for manifestation. In other words, for a given concept $\mathscr{A}$ to be learned we would map its training sample onto a sample of digit “1” in the manifestation space.

TABLE 2. AVERAGE DISTANCE TO THE OTHER 9 DIGITS

0	1	2	3	4	5	6	7	8	9
5.64	6.16	5.37	5.01	5.27	4.87	5.28	5.35	4.46	4.47

3.4. Manifestation to a digit “1” sample

Refer back to Figure 6. To manifest the “ring” wafer plot (concept $\mathscr{A}$), we use the plot to train a manifestation encoder and a manifestation decoder. When training the encoder, we need to select a point in the concept region of digit “1”. In our implementation, we select the point that has the largest SVM score. We train the encoder to map the wafer plot to the point. After that, we train the decoder to reconstruct the wafer plot from the point. After manifestation training, we then check how the resulting recognizer performs on the other 8299 wafer plots.

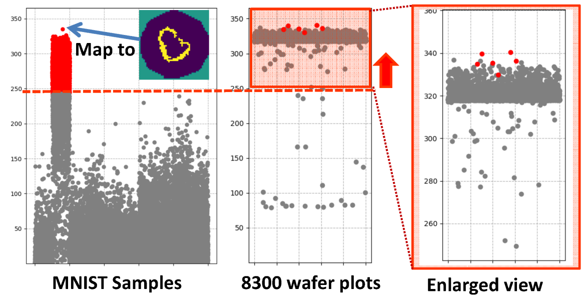


Figure 17. Initial manifestation that does not work

Figure 17 shows the result for the 8300 wafer plots. Each wafer plot corresponds to a point in the manifestation space. The SVM model calculates a score for the point. The six “ring” plots in Figure 2 are shown as six red dots “●”. All others are shown as grey dots.

What we desire to see is that the six red dots are above the threshold and none of the grey dots are above it. In contrast, we see that majority of the grey dots are also above the threshold, an undesirable result. In the enlarged view, however, we do see that the scores of the six wafer plots are higher. The problem is that the scores for other plots are also high. Hence, we need to find a way to lower the scores for all the other plots.

4. Generic Counter Examples

Figure 18 shows three wafer plot examples and the corresponding plots generated by the manifestation decoder. It is interesting to see that regardless of the input image, the output image always shows a similar “ring” feature. As explained below, the problem, though, is not with the decoder, but is with the encoder.

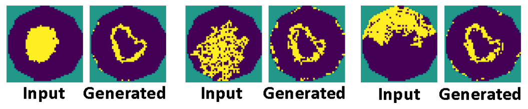


Figure 18. Checking the decoder’s output - 3 examples

The manifestation encoder is trained with only one wafer plot of concept $\mathscr{A}$. Hence, the encoder lacks information on where to map a wafer image different from concept $\mathscr{A}$. To improve the encoder's mapping, we use a *generic* counter example to differentiate from concept $\mathscr{A}$.

One generic counter example is the no-fail wafer plot, i.e. the "blank" wafer image. Figure 19 shows the result by adding this image in training the manifestation model. The training now has two wafer images. The blank image is mapped to a point of digit "0" in the manifestation space. Instead of mapping to the point with the largest SVM score, it maps to the point with the medium SVM score. The reason "0" is chosen is because in the manifestation space, the distance between the center of digit "1" and the center of digit "0" is the largest (e.g. see Figure 12). Through experiment, we observed that the further away the digit we chose for this mapping, the better the result would be.

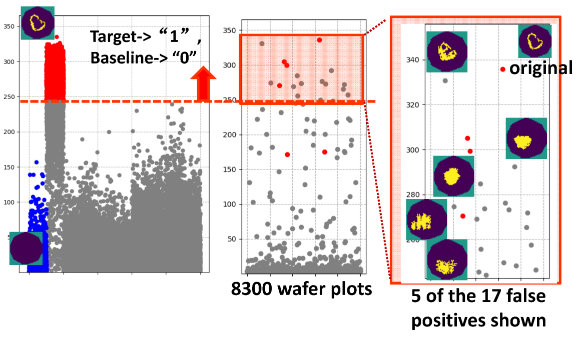


Figure 19. Add a no-fail wafer image, mapped to digit "0"

Comparing to Figure 17, observe in Figure 19 the significant effect by adding the blank wafer image. The resulting model classifies 21 wafer plots as in-class. Four out of the 21 plots are among those six containing a "ring", i.e. those shown in Figure 2. Hence, we consider the other 17 as false positives. The images for five of these 17 false positives are shown as examples.

To improve the result further, we consider adding a second generic counter example. This time we add an all-fail wafer plot, i.e. the "full" wafer image. This wafer image is mapped to digit "2" whose center is the 2nd farthest to the center of digit "1". Figure 20 shows the effect.

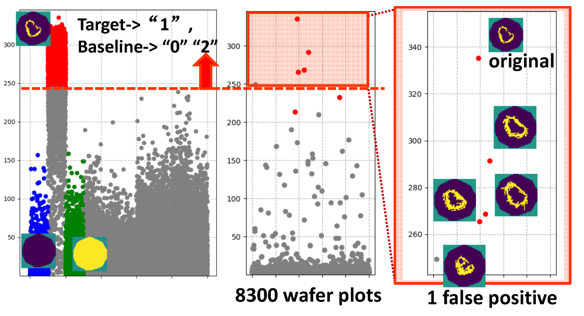


Figure 20. Add a no-pass wafer image, mapped to digit "2"

The model classifies five images as in-class with only 1 false positive left. The four recognized in-class plots in Figure 19 are also kept as in-class. Their images are shown in Figure 20. The image of the one false positive is also shown in Figure 20. Effectively, adding the "full" wafer image removes 16 of the 17 false positives in Figure 19.

4.1. The best manifestation setup

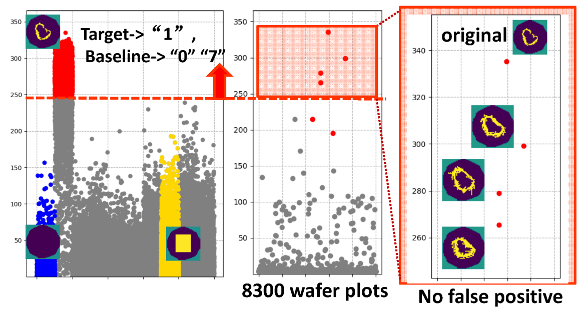


Figure 21. Add a bounding-box wafer image, mapped to digit "7"

Figure 21 shows the best manifestation setup found in our study. Instead of using the all-fail wafer image, we use an image with all-fail in a bounding box. The bounding box is the smallest box containing the "ring" feature. Then, we fill the box with yellow pixels (excluding the area outside the wafer boundary if any). Because this bounding box image depends on the wafer plot, we map this image to digit "7" (instead of digit "2" as that in Figure 20). In Figure 12, we observe that digit "7" is the closest digit to digit "1".

As seen in Figure 21, the best model keeps the four in-class plots as in-class while removing the 1 false positive concerning us in Figure 20. We consider this as the most desirable result as it can recognize wafer plots similar to the given training wafer plot with no false positive.

5. Iterative Recognition

As shown in Figure 2, we consider six wafer plots having the same "ring" feature. The first one is used to train the manifestation model shown in Figure 21. The model recognizes three of the other five plots. This leaves two wafer plots unrecognized, i.e. two false negatives.

Because our manifestation setup is designed to achieve zero false positive, the false negative issue can be mitigated rather easily. The idea is that we would simply expand the recognition by iteratively building additional manifestation models based on those recognized wafer plots. Suppose in iteration i, a set S_i of wafers $\{W_1, W_2, \ldots\}$ are recognized where no W_j is recognized in a previous iteration. For each W_j, we build a manifestation model and apply it in the next iteration to see if there is any new wafer plot recognized.

This process continues until either (1) we reach a *closure* recognition set, or (2) the recognized plots are "too far" from the original training wafer plot. A closure set $\{W_1, \ldots, W_k\}$ means that if we build a manifestation model for any W_i in the set, the recognized wafers are all in the set.

We found that iterative recognition can reach a closure set in a few iterations (e.g. three iterations) but not always. To avoid too many iterations, we can impose rules to prevent building a model from some recognized plots.

For example, such rules can be implemented by checking the size and location of a bounding box from a recognized plot, as comparing to the bounding box from the original wafer plot. In our implementation, if the size or location of the bounding box differs from the original bounding box by more than 50%, we would not build a manifestation model for the recognized plot.

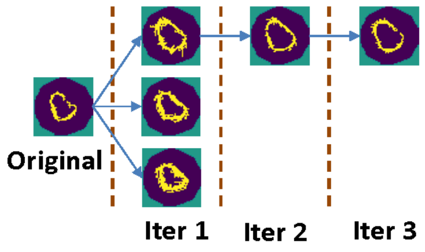


Figure 22. Iterated recognition for the "ring" concept

Figure 22 shows the result of this iterative recognition process. The first iteration shows the same result as shown in Figure 21. Then, the two missing in-class plots are recognized in the 2nd and the 3rd iterations. With iterative recognition, all six wafer plots shown in Figure 2 are included. There is no false positive in the iterative recognition process, i.e. Figure 22 shows all the recognized plots. Furthermore, the process stops because it reaches a closure set.

5.1. Other classes of wafer plots

To validate the generality of the manifestation setup, we consider two additional classes of wafer plots: "center" and "edge" which were among the 11 classes of wafer plots reported in [2]. Figure 23 shows the iterative recognition result starting with a "center" wafer plot.

Again, in each iteration only those newly recognized wafer plots are shown. A manifestation model is built and applied based on each recognized plot, except those marked with a "▲". Those marked wafer plots were excluded by the rule checking mentioned before.

For those unmarked plots, if there is no outgoing line, it means all the recognized plots by the manifestation model are already included in the previous and current iteration(s). The figure shows all plots recognized in the process.

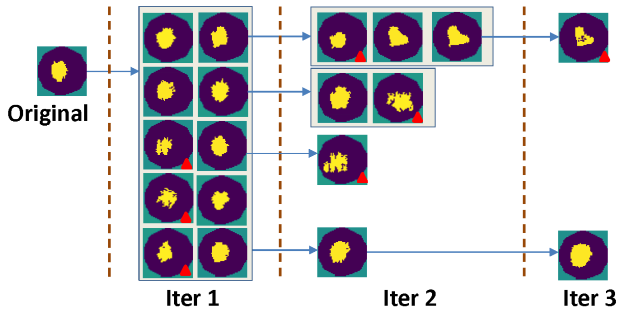


Figure 23. Iterated recognition for the "center" concept

Similarly, Figure 24 shows the result starting with a wafer plot with an "edge" feature.

Note that in the work [2], out of the 8300 wafers, in total there were 80 wafer plots reported as the "center" class and 108 wafer plots reported as the "edge" class. In contrast,

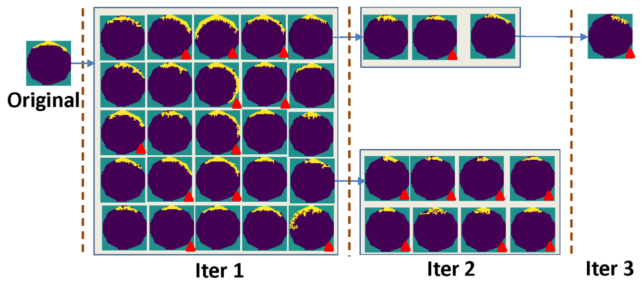


Figure 24. Iterated recognition for the "edge" concept

Figure 23 shows 20 wafer plots and Figure 24 shows 38 wafer plots. None of them would be considered as a false positive from the classification reported in [2]. However, the manifestation models recognize much fewer wafer plots in each case. This is understandable because the recognition process starts with one particular wafer plot, and the process only tries to recognize plots similar to the given wafer plot.

6. Summary Of Manifestation Learning

In Section 1.3, we begin the paper by discussing a similarity scoring method $IoU()$. Given a wafer plot, the method calculates a score for every wafer plot. As discussed in Section 3.2, in manifestation a SVM score is calculated for every wafer plot. Figure 25 shows a comparison for the top ranked plots using these two scoring methods.

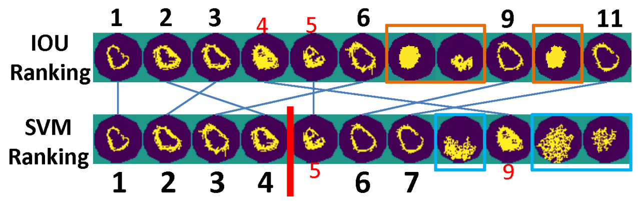


Figure 25. Ranking comparison: SVM scores Vs. IoU scores

For the six "ring" plots, the SVM scoring ranks them as 1,2,3,4,6, and 7. The 1st plot is the given plot. The model in Figure 21 puts the threshold between the 4th and the 5th plots. Hence, the top 4 plots are recognized as in-class. The IoU ranks the six plots as 1,2,3,6,9, and 11.

It is interesting to observe that if we assume the threshold is set to include all the top 11 plots as in-class, with both rankings, they both can capture all six "ring" plots as in-class. Also, both have exactly five false positives. The false positive sets, however, are different by three plots (the three plots highlighted in each ranking).

In view of the $IoU()$-based method discussed in Section 1.3, we see that the proposed Manifestation Learning effectively does the following four things:
(1) It maintains the trend of an IoU ranking.
(2) It fine-tunes the IoU ranking.
(3) It automatically decides a threshold.
(4) It ensures no false positive to enable implementation of an iterative recognition process.

It is interesting to note that if we consider IoU as intuitive and interpretable to a person, we see that Manifestation Learning keeps this intuition to a large extent.

7. Wafer Plot Generation

Figure 9 shows the possibility to use manifestation learning for sample generation. For example, a walk from a given wafer plot for a target concept to a (generic) counter example can produce a sequence of new wafer plots to represent the target concept. This can be particularly useful if a user intends to obtain many wafer plots of the same category from one wafer plot example.

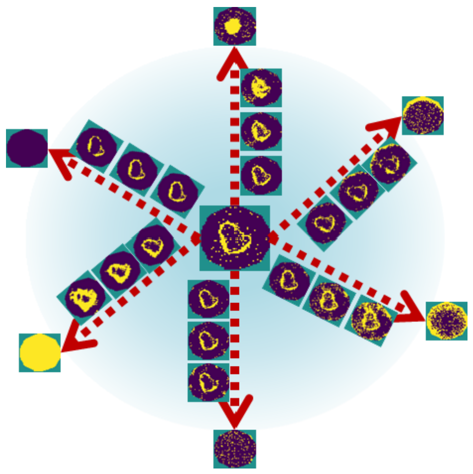


Figure 26. Multi-directional wafer plot generation

Fig. 26 illustrates how this wafer plot generation scheme can be used flexibly with different counter examples, two generic ones and four selected from other categories of wafer plots. As seen, by using different counter examples, a walk can generate different "ring" wafer plots. In practice, a counter example can be selected randomly from a set of unlabeled wafer plots. Because each sample is generated based on the given plot and a counter example, it is straightforward for a person to trace each generated sample back to its sources. This traceability can be helpful when using such a generated dataset in practice.

8. Conclusion

In test data analytics, one obstacle frequently encountered is the lack of sufficient samples for training a reliable model. This work focuses on an extreme scenario where only one training sample can be used. The proposed approach is by manifesting the given training sample into a space where a recognition model is already pre-trained with a large dataset originated from another domain. We call this approach Manifestation Learning and in this work, its feasibility is shown in the context of wafer plot recognition. Although this work focuses on plot recognition, the preliminary result in Section 7 also shows the potential of using manifestation learning for plot generation.

Acknowledgment This work is supported in part by National Science Foundation Grant No. 2006739.

References

[1] M. Nero, J. Shan, L. Wang, and N. Sumikawa, "Concept recognition in production yield data analytics," *IEEE International Test Conference*, 2018.

[2] C. Shan, A. Wahba, L.-C. Wang, and N. Sumikawa, "Deploying a machine learning solution as a surrogate," in *IEEE International Test Conferecel*. IEEE, 2019, pp. 1–10.

[3] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and J. Bengio, "Generative adversarial networks," *arXiv:1406.2661*, 2014.

[4] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training GANs," *arXiv:1606.03498v1*, 2016.

[5] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv:1511.06434v2*, 2016.

[6] A. Wahba, L.-C. Wang, Z. Zhang, and N. Sumikawa, "Wafer pattern recognition using tucker decomposition," in *VLSI Test Symposium (VTS), 2019 IEEE 37th*. IEEE, 2019, pp. 1–6.

[7] A. Wahba, J. Shan, L.-C. Wang, and N. Sumikawa, "Wafer plot classification using neural networks and tensor methods," in *ITC-Asia*. IEEE, 2019, pp. 79–84.

[8] A. Wahba, C. Shan, L.-C. Wang, and N. Sumikawa, "Measuring the complexity of learning in concept recognition," in *Int. Symposium on VLSI Design, Automation and Test*. IEEE, 2019, pp. 1–4.

[9] F.-F. Li, "Knowledge transfer in learning to recognize visual object classes," *International Conference on Development and Learning (ICDL)*, 2006.

[10] I. Goodfellow, Y. Benjio, and A. Courville, "15.2 transfer learning and domain adaptation," *Deep Learning*, 2016.

[11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, pp. 318–362, 1986. [Online]. Available: https://doi.org/10.21105

[12] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2013. [Online]. Available: https://arxiv.org/abs/1312.6114

[13] B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. The MIT Press, 2001.

[14] e. a. Xi Chen, "Variational lossy autoencoder," 2016. [Online]. Available: https://arxiv.org/abs/1611.02731

[15] D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, "Ladder variational autoencoders," *Advances in Neural Information Processing Systems*, vol. 29, pp. 3738–3746, 2016.

[16] S. Zhao, J. Song, and S. Ermon, "Infovae: Information maximizing variational autoencoders," 2017. [Online]. Available: https://arxiv.org/abs/1706.02262

[17] A. Gretton, K. M. Borgwardt, M. Rasch, B. Schölkopf, and A. J. Smola, "A kernel method for the two-sample-problem," *Advances in Neural Information Processing Systems*, pp. 513–520, 2007.

[18] N. Shibuya, "Up-sampling with transposed convolution," 2017. [Online]. Available: https://medium.com/activating-robotic-minds/up-sampling-with-transposed-convolution-9ae4f2df52d0

[19] M. Abadi and et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[20] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[21] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[22] e. a. Samuel R. Bowman, "Generating sentences from a continuous space," 2015. [Online]. Available: https://arxiv.org/abs/1511.06349

[23] ——, "Long short-term memory," *Neural Comput.*, vol. 9, pp. 1735–1780, 8 1997.

[24] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009. [Online]. Available: https://www.cs.toronto.edu/ kriz/cifar.html

[25] F. Pedregosa and et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://scikit-learn.org/stable/