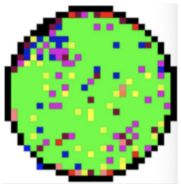
MVI - Semestral project Documentation

Görke Océane (gorkeoce@fit.cvut.cz)

This is the documentation of my semestral project you can find details from the Jupyter notebook WaferMaps_Pattern_Recognition.ipynb.
The subject being : WAFER BIN MAPS PATTERN RECOGNITION *(Create an augmented dataset of defects on wafer bin maps and train a map size independent CNN classifier to recognize them.)*
My work is divided on different parts from which details and explanation can be found here. Some things are directly explained on the notebook so they may be repeated here.

• Wafer Map Introduction

Wafers are used in manufacturing process (semiconductor essentially).
A wafer bin map (WBM) is the result of an electrical die-sorting test. It provides information on which bins failed what tests as well as the performance of the semi conductor that has been tested.



Here is what a wafer bin map can look like.

There is different colours representing default levels (dies) detected (green part is like the normal surface of the bin), but usually, WBM patterns are transformed into binary values for visualisation and analysis.
The dies that pass the functional test = 0 and the defective dies = 1.
The rest of the work will be based on it.

• Imports

All the imports needed in the work are reunited here.

• Dataset / Data import

I first checked the working directory and the elements in it.
Then I imported the dataset **LSWMD.pkl**
The .pkl extension file represents a file created by pickle which is a Python module enabling objects to be serialised to files.

Some information and data displays were also done in this part.
Also some data modifications were done here in order to simplify the analysis later. For example, for the features trainTestLabel and failureType, the data was like [[x]] so I applied a function array_values I implemented in order to only have the value x.

• Data explanation

For a good study on the data, someone has to first understand what is the data to deal with.

For each features, some explanations are given and the data can be displayed.
Also, some modifications on it can be done directly in this part.

waferMap

For this feature, the study was focused on what a wafer looks like in this dataset (array) and how it can be displayed (plot).
Moreover, I created a new feature *waferDim* that represents the dimension of the waferMap arrays. Again, it will be easier to process with data later thanks to that, especially in the data augmentation part.

dieSize

This data represents the product of waferDims for each wafer. Indeed, the distribution is not equal for all the wafers. Different dieSizes are displayed.
Besides, some outliers are in this feature (use of boxplot()). They have to be deleted since it may represent noise in the image. This deletion is done here, I will check if they are still there after cleaning the data.

waferIndex

Each lot (=reunion of wafers) contains usually 25 wafers. For each row in this dataset, it is assigned the number of the lot it belongs to and the place it has in this lot (waferIndex). It is remarkable to see that not all lots have exactly 25 wafers (cf distribution).

lotName

It represents the lot the wafer belongs to. Some modifications were done on it in order to only keep the number of the lot as an int and not as a string.

trainTestLabel

The wafers have already been part of some tests and training but there is no need to take it into consideration for this work.

failureType

This is one of the most important feature for this project : the pattern of the wafer bin registered as a failure type.
There is 10 types of failures including 0 (no label) and none (no pattern = no failure); they represent almost 97% of the dataset which is quite huge. Nevertheless, both are not necessary for the pattern recognition model.
A dictionary of the sorted patterns and one number assigned to it has been created in order to help displays and manipulations later.
In order to have an idea of the shapes, I created 3 independent datasets corresponding to wafers with labels (df_with_label : none, Center, Donut…), wafers with patterns (df_withpattern : Center, Donut…) and wafers without pattern (df_non_pattern : none). It resulted that only 15% were wafers with patterns.
Some graphs were used to display this data (distribution and plots of wafer maps with each failure types)

All dataset

The point of this part is to study if there is any correlation between the features : dieSize, lotName, trainTestLabel and failureType. Some features such as trainTestLabel and failureTypes had to be represented by numbers in order to study correlation.
However, none of the features are correlated to each other which was foreseeable.

• Data cleaning

On this part, the main task was to detect none values or missing values and deal with them.
Luckily, there were not any none values but some features had empty list or values with 0 that corresponded to nothing. This kind of values are kind of problematic especially for failureType feature so they have been deleted.

Then, considering the outliers in dieSize detected earlier, not all of them have disappeared. In this way, data has to be more precisely deleted. After going step by step, the final condition is df_dieSize['dieSize'] < 2450.

On the whole, **790 119 lines** have been deleted.

- **Data transformation (reshape)**

I chose to consider only wafers with dimensions (26,26) because it the first squared dimensions in the value_counts and I will proceed with data augmentation in further parts.
This approach will delete some data once again but a squared dimension is needed for the CNN model. Therefore, the dataset will be augmented later.

- **Data augmentation**

After deleting a lot of data and in order to have better performance, data augmentation was needed. Data augmentation is similar to collecting new data (though it doesn't bring as much new data) because it adds more data on the dataset. This data is based on the first dataset we had : a convolutional autoencoder is used since wafer maps are similar to images.
So, in order to augment the data it is interesting to use a convolutional autoencoder. Also, a simple flip of the image can augment the data.
Both methods will be used.

First of all, the encoder/decoder is implemented to transform the image and by adding some noise, it will generate a new data with the same pattern for the failure.
The convolutional encoder is composed of 5 layers : Conv2D - MaxPool2D - Conv2DTranspose - UpSamping2D - Conv2DTranspose
The first try was done on 20 epochs but with this small number of epochs the maps were really different from each other. Therefore, I chose a bigger number (other numbers such as 50 and 100 have been tested before but in order to not have more inputs commands and outputs I deleted them from the notebook).
I chose to change the number of epochs because one epoch is one complete pass through the training data, so the pass have to been augmented in order to have more precise data. Whereas the batch size is the number of iterations to complete one epoch.
On the model with epochs = 200, the loss is quite low (<0.04) so this is a good parameter to take it into account.
Finally, this encoder model has been applied to the data already in the dataset in order to create new data similar and concatenate them to the new dataset.

Besides, the new data is augmented by adding some flip maps saved in the previous part.

- **Model**

Whenever it is not specified, I worked with epochs = 200 and batch_size=1024. I considered this number from the previous part and new wafer maps generated.

Model 1 : Flatten
This layer is an important one when we are dealing with object detection and image recognition. Therefore I chose to use it in first as a very simple CNN.
114/114 - 0s - loss: 0.2381 - accuracy: 0.9657 - 115ms/epoch - 1ms/step
I think better results are possible and this neural network here is too simple anyway.

Model 2 : CONV - RELU - FC
As the wafer bin maps are considered as images, the convolutional layer has to be in the model. This is the first layer dealing with the inputs in order to extract the various features.
RELU is the activation of this layer. It is specified in the model pattern because other may be used but it wasn’t necessary to precise it. The activation relu is used for filtering information that is propagating through the network especially in the convolutional layer where nonlinearity can be.

Besides, softmax is used for the last Dense layer because the problem here is a multi-class classification.
114/114 - 5s - loss: 0.2770 - accuracy: 0.9652 - 5s/epoch - 40ms/step
This model is worse because the accuracy didn't improve that much for a greater loss.

Model 3 : CONV - RELU - FC - FC
Since the Full connected layer seems to be useful in this work I tried to add one more layer of it. Indeed, it could optimise some objectives such as scores.
114/114 - 0s - loss: 0.2536 - accuracy: 0.9655 - 495ms/epoch - 4ms/step
Adding a Flatten layer didn't change a lot, let's try with Pooling Layer now.

Model 4 : CONV - RELU - POOL - FC
A Pooling Layer is introduced now by following a Convolutional Layer. This Pooling Layer can reduce the computational costs but more specifically here I used MaxPooling because I wanted to reach the maximum value of the current view. It helps indeed to preserve the detected features.
114/114 - 0s - loss: 0.2491 - accuracy: 0.9668 - 483ms/epoch - 4ms/step
The accuracy is still the same but the loss is much better.

Model 5 : CONV - POOL - CONV - POOL - FC
I tried the previous model with one combination CONV/POOL more. It can help to deal with the inputs with double efficacy.
114/114 - 0s - loss: 0.2386 - accuracy: 0.9674 - 304ms/epoch - 3ms/step
This model is the best one so far. Let's focus on it and change some parameters.

Model 6 : CONV - POOL - CONV - POOL - FC
Epochs = 500
114/114 - 0s - loss: 0.3820 - accuracy: 0.9660 - 338ms/epoch - 3ms/step
*T*he accuracy has indeed improved but not the loss so augmenting the number of epochs here doesn't seem to be a good solution.

Model 7 : CONV - POOL - CONV - POOL - CONV - FC
If we consider the previous models, adding a CONV Layer improved the results. Let's try it again then.
114/114 - 0s - loss: 0.2608 - accuracy: 0.9638 - 303ms/epoch - 3ms/step
This model gives better result than model 6 but they are similar to model 5 which was considered as the best one so far.
Since we know epochs won't improve the model, let's try to change the batch_size.

Model 8 : CONV - POOL - CONV - POOL - CONV - FC
batch_size = 2048
14/114 - 0s - loss: 0.1647 - accuracy: 0.9613 - 273ms/epoch - 2ms/step => changing the batch didn't give better results either.
It seems to be the best model, I will do some comparisons in the next part.

- **Validation model**

After studying all the previous models, four were quite similar : model 5 - model 6 - model 7 and model 8. In order to find the best one, I plotted accuracy and loss plots.
It resulted that the best model was the model 8. Indeed the train and test lines were the most similar for all epochs.
That-is-to-say CONV - POOL - CONV - POOL - CONV - FC with epoch = 200 and batch_size = 2048.