Optimization of Industrial Solar Cell Fabrication with Machine Learning

Casper Anton Eijkens
Optimization of Industrial Solar Cell Fabrication with Machine Learning

by

Casper Anton Eijkens

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday September 23, 2019 at 10:00 AM.

Student number: 4748913
Project duration: Jan 1, 2019 – Sep 1, 2019
Supervisors: Dr. Olindo Isabella, TU Delft
           Ziv Hameiri, UNSW
           Yoann Buratti, UNSW
Thesis committee: Dr. Olindo Isabellea, PVMD, TU Delft, supervisor
                  Prof. Miro Zeman, PVMD, TU Delft
                  Dr. Paul Procel, PVMD, TU Delft
                  Dr. Simon Tindemans, IEPG, TU Delft

This thesis is confidential and cannot be made public until September 23, 2019.

An electronic version of this thesis is available at
https://repository.tudelft.nl/.
Thank you Ziv and Yoann, for facilitating my complete stay with the university of New South Wales.

Casper Anton Eijkens
Delft, September 2019

“And set a watcher upon her, great and strong Argus, who with four eyes looks every way. And the goddess stirred in him unwearying strength: sleep never fell upon his eyes; but he kept sure watch always.”

Hesoid, Aigimios
Abstract

The present thesis project approaches the improvement of performance metrics of industrially fabricated solar cells. It aims to develop an optimization method that uses machine learning algorithms to predict cell efficiency based on process configurations. Conventional frameworks for production line improvement rely on modeling design of experiment data with low-degree polynomials. This existing method allows modeling of a few processes, but cannot include all inter-dependencies between the processes in the production line. In contrast, machine learning algorithms can model all processes at once and create more complete and accurate models of the production line. As part of this study, a virtual production line was developed to test and compare different machine learning algorithms. Based on experiments with this virtual production line, two novel approaches are proposed that can optimize production lines using a combination of artificial neural networks and genetic algorithm. It was demonstrated that the first approach improved the photovoltaic cell efficiency of cells from a simulated production line from $17.9 \pm 0.3\%$ to above $19\%$. 
In the previous decade, the wholesale photovoltaic (PV) module price was reduced by a factor of 15 thanks to research and development, learning-by-doing and economies of scale [1, 2]. Today, PV is one of the cheapest forms of electricity [3] and has become a cornerstone in the transition to renewable energy generation. This progress is partly thanks to fine-tuning of PV production lines with *process optimization*, which improved photovoltaic cell efficiency (PCE) and reduced production costs.

The manufacturing of PV devices is a combination of consecutive processes that turns silicon, chemicals, gasses and metals into a solar cell. The focus of this study is the production of aluminium-back surface field (Al-BSF) single-crystal silicon (Si) solar cells. Al-BSF fabrication processes include material selection, sawing, saw damage removal, etching, texturing, cleaning, diffusion, edge isolation, surface passivation, front and rear screen printing and co-firing. The operation of each process is determined by multiple variables and strong inter-dependency between processes exists [4–6]. Each process needs to be tuned to maximize the performance of the fabricated solar cells. This is a complex optimization problem every PV factory faces.

The conventional framework of process improvement is called response surface methodology (RSM) [7]. RSM improves processes in three steps: first, design of experiment (DoE) data is collected about the process(es) of interest. Next, a polynomial is fitted on the DoE data to establish a prediction model. Finally, the prediction model is used in some optimization algorithm to identify the input variables for which the process is improved.

The present thesis project approaches the improvement of PV fabrication with machine learning (ML). ML refers to a broad class of algorithms that performs specific tasks without explicit instructions. There exist many different ML algorithms and depending on the complexity and nature of the problem, a different ML algorithm should be used. Some ML algorithms have a low capacity to fit a wide variety of functions and should be used for simpler problems. Other ML algorithms have a high capacity to fit a wide variety of functions and should be used for harder problems. When a low-capacity ML algorithm models a complicated relationship, it will underfit as it cannot capture the complexity of the data. Likewise, when a high-capacity ML algorithm models a simple relationship, it starts overfitting to model all
meaningless noise present in the data. The RSM approach is capable of modeling only a few processes at once, because the number of DoE experiments grows quickly with the amount of variables included. Furthermore, polynomial fitting can only model a few variables at once [7, 8], as it starts overfitting the data around greater than ten variables. Both these reasons lead to the fact that in practice, processes must be improved individually or a few at the time.

When the production line is modeled as individual processes, the inter-dependencies between processes are not taken into account. Since these inter-dependencies are important for accurate prediction of the solar cell efficiency, optimization using these individualistic models are expected to be sub-optimal. In this study, the limitations of conventional process modeling are addressed by using ML, which can accurately model complex patterns in high-dimensional data. ML can simultaneously learn multiple processes and the interactions among their variables that normally remain unexplored. ML is expected to outperform polynomial fitting of manufacturing processes in both scope and accuracy [9, 10].

Although application of ML in PV and semiconductor manufacturing has been previously investigated [9, 11–16], it is generally for optimization of a limited number of processes. Furthermore, in these studies, a comparison between different algorithms is missing and only small data sets are used for verification. Several of these studies implement ANN, however, the motivation for choosing this algorithm is lacking. In this study, we aim to provide new insights into the performance of various ML algorithms in process optimization.

The proposed research is an effort to make an extensive comparison between different ML algorithms for modeling PV manufacturing. This work could be a guide for factories trying to implement ML algorithms in their process optimization. By using better predictive models, optimization can be improved and factories could improve their product drastically while using the same tools.

The research questions of the present thesis are:

1. What ML algorithm is most suitable for modeling a production line?
2. Can a production line be improved by modeling production data with ML?
3. Can ML model individual processes by modeling DoE data?
4. Is it possible to model a complete production line using only DoE experiments of individual processes?

The first aim of this research is to develop a simulation of a production line to compare ML algorithms with. The second aim is to make an extensive comparison between well-known ML algorithms and to analyze the advantages and disadvantages of every algorithm. The third aim is to find a optimization method that is practical and that can be used in factories today to improve their production line. The fourth aim is to compare different ML algorithms on small DoE data sets. It is conjured that an ML-based optimization approach outperforms the conventional RSM framework to improve a complete PV production line.
First, chapter 2 provides background about the production of Al-BSF single-crystal Si solar cells, RSM process optimization and ML. Chapter 3 discusses various tools for analyzing ML performance. The same chapter also explains the four experiments done in this thesis to answer the four research questions. Chapter 4 contains the results of these experiments and a discussion of their implications.
Machine Learning Approach to Process Optimization

First, the industrial production of aluminium-back surface field silicon solar cells is discussed to highlight the complexity of photovoltaic manufacturing. Next, response surface methodology is discussed to demonstrate how these processes are improved in practice. Next, some background is given on machine learning and various well-known machine learning algorithms.

2.1. Industrial Solar Cell Manufacturing

In the following section, a brief summary is given of the industrial production of n-type emitter, single crystal silicon (Si) solar cell with aluminium-back surface field (Al-BSF) design. In fig 2.1, a schematic overview of the industrial fabrication of Al-BSF cells is shown.

The manufacturing processes are controlled by controllable factors and influenced by uncontrollable factors [9]. Each process has multiple input variables that determine the operation of the process, and delivers multiple outputs such as wafer properties. The set of operations that are performed in a process are referred to as a recipe. The configuration of each recipe affects the properties of the wafers so each process needs to be tuned to maximize the performance metrics of the solar cell. In this thesis, photovoltaic cell efficiency (PCE) is selected as the performance metric to be optimized.

In solar cell manufacturing, certain operations strongly affect subsequent processes, due to inter-dependencies between processes [4–6]. For this reason, recipes need not only to be optimized individually, but also in coordination with the other processes in the production line. In this section, every process will be discussed briefly to highlight the interactions between variables of PV manufacturing.

Wafer making: the choice of starting material for the Si wafers has a large impact on the performance metrics of the produced solar cell [5]. Process variables include boron (B) doping of the silicon ingot, ingot quality, wafer thickness and wafer slicing method. n-type, single crystal PV fabrication starts with a boron-doped silicon ingot.
of Czochralski (CZ), or float zone (FZ), quality [6]. CZ and FZ ingots are made differently: CZ is grown using quartz crucibles with $10^{18}$ cm$^{-3}$ oxygen [6], which can form complexes with B that reduce the carrier lifetime [17]. Reduction of carrier lifetime lowers the voltage, current and efficiency of the device since less charge carriers can be collected. Additionally, the oxygen can become active at higher temperatures [6], causing problems during high-temperature thermal processes such as diffusion and co-firing. FZ wafers are much purer than CZ as the impurities are removed using a technique called ‘zone melting’ [18]. The production of FZ silicon is costly, so it is not standard in industrial Si solar cell manufacturing [19].

Increasing the boron doping concentration is beneficial for the open-circuit voltage of the solar cell, because the boron doping lowers the wafer resistivity and the dark current density [5]. However, for increased boron concentration, the minority carrier lifetime reduces as Auger recombination starts to dominate [5].

The silicon ingot is sliced into wafers using wire-saws [20], which damages the surface of the wafers. The saw damage can be reduced by using a diamond-tipped wire-saw [21].

**Etching**: the saw-damaged regions on the wafer surface cause high minority carrier recombination [22]. The saw damage can be removed with a sodium hydroxide (NaOH) etch. This alkaline etch also removes any residual contaminants like metal particles, dust particles and organic molecules. The concentration of NaOH and the etch temperature determine the speed at which the wafers are etched. After the etch, the wafers are rinsed in purified water. Too rigorous etching conditions result in unnecessary loss of silicon, and the wafer may break as it becomes too thin, resulting in lower yield of the manufacturing line.

**Texturing**: Texturing applies a pyramidal structure on the Si wafer surface, which reduces optical losses of the solar cell by reflecting the incoming light sideways [23]. Reducing reflection is one of the main ways to improve the efficiency of the solar cell, as a lower reflection leads to higher generation of charge carriers. The introduction of pyramids on the surface of the wafer lowers the reflection from 36% to 11-12% [5, 24].

After texturing, the wafers are rinsed in purified water. Addition of propanol boosts uniform texturing across the surface of the wafer [25]. Ideally, the texturing process results in 100% pyramid coverage. The quality of the pyramids is determined by the proportions of NaOH, propanol, temperature, time and flow rate. The crystal struc-
ture of the silicon is preferred to be in the <100> orientation, as it is beneficial for the forming of pyramids [6].

**Cleaning**: before the Si wafers enter the diffusion furnace, the impurities on their surfaces need to be cleaned [26]. Texturing leaves silicate residues and metal impurities in the surface caused by texturing [26]. For this step, commonly a mixture between HCl (hydrogen chloride) and HF (hydrogen fluoride) is used. The chloride ions in HCl form complexes with metal ions, removing them from the wafer surface. The chloride ions also react with the sodium or potassium remaining after the previous texturing etch. If the metal impurities are not removed, they will diffuse into the wafer and affect the charge carrier lifetime of the solar cell. In general, cleaning should be thorough, but a factory naturally wants to limit the amount of chemicals that are used to reduce production costs. Minimizing the costs of chemicals without compromising the quality of the cell is also part of optimizing a production line.

**Diffusion**: in the case of n-type Si solar cells, the diffusion process introduces phosphorus (P) atoms in the Si wafer to create the emitter layer [27]. The emitter layer is essential for the working of the solar cell, as it creates the p-n junction to separate the charge carriers. Diffusion of P atoms creates an n-type emitter in a p-type wafer. In order for the P atoms to diffuse into the wafer, the diffusion furnace needs to be heated up to around 800-850°C [5].

It is crucial for the cell performance that the emitter has the right thickness. The emitter layer needs to be thin, because most charge carriers are generated close to the cell surface. When the emitter is too thick, the generated charge carriers will recombine before they can be collected. The thickness of the emitter layer is determined by the temperature and period of diffusion. After charge separation in the p-n-junction, the electrons diffuse laterally though the emitter to the electric contact [5]. The sheet resistance refers to the resistance in the emitter layer and is a good measure of quality of the emitter layer [28]. The lower the sheet resistance, the better the performance of the solar cell.

**PSG removal**: during diffusion, also a layer of phosphosilicate glass (PSG) [29] is created which has to be removed because it contains traces of metals [24]. The PSG is removed with 5% \(\text{HF}\) and is rinsed with purified water.

**Edge isolation**: the diffusion process introduces an emitter layer around the wafer, including the rear surface and the edges. The diffused layer on the edges needs to be removed as they form shunting pathways for the electrically separated charge carriers. For example, the edges can be removed with highly reactive plasma gas [30]. The power and period of edge isolation should not be too high or long as the plasma introduces damage to the silicon, causing increased recombination [24]. Too low power or too brief removes too little of the diffused layer, causing low shunt resistance. More modern production techniques use wet etching or lasers, which leads to higher shunt resistance [24].

**Surface passivation**: a thin film of SiN\(_x\) can be deposited on the wafer surface with plasma-enhanced chemical vapour deposition (PECVD) process [5, 31]. The SiN\(_x\) has multiple functions for the solar cell. First, the SiN\(_x\) layer functions as an anti-reflective coating (ARC), further lowering the reflection of the textured surface. Secondly, SiN\(_x\) acts as a passivating layer that lowers surface recombination losses and
insulates the emitter. The quality of the ARC and passivation depends on the composition of SiNₓ, which in turn depends on the parameters of the PECVD process. Parameters include pressure, temperature, the proportions of the gasses and the power of the plasma [32]. The SiNₓ also diffuses hydrogen during the subsequent thermal processes [24]. Hydrogen can passivate recombination sites in the silicon, which improves minority carrier lifetime and cell voltage [33].

**Rear screen-print:** The rear electrical connection is commonly made using the screen-printing process [5, 32, 34]. During screen-printing, a liquid metallic paste is squeezed through a screen, producing the desired metallization pattern on the surface of the Si wafer. The rear side of the wafer is completely covered in aluminum (Al) paste.

**Front screen-print:** The front electrical connection is made with silver (Ag) paste in a similar fashion to the rear screen-print [35]. However, the design of the front metal grid is more important since a trade-off exists between series resistance losses and shading losses. For optimal charge conduction, the fingers and busbars need to be as thick as possible, but with thicker fingers or busbars, shading losses increase [5]. Initially, the silver metal is not in contact with the Si wafer, and needs to be etched through the SiNₓ during the co-firing process.

**Co-firing:** After the metal contacts are placed, the firing process will finalize the wafer [36]. The device enters a belt furnace at about 800-850°C. When both rear and front-side contacts are fired at the same time, the process is called co-firing. During co-firing, the silver paste etches away the underlying SiNₓ layer, which creates a direct contact between the contact and the emitter layer [5]. The insulating SiNₓ between Ag and emitter layer needs to be etched since that would block collection of charge carriers by the contacts. The lack of passivation in the metal-semiconductor interface causes a large amount of defects [5]. To reduce recombination at this interface, the contact area is minimized and the region around the contacts is heavily doped with n⁺⁺ [37]. The heavy doping increases band bending and blocks minority holes from entering the defects and creating traps.

Simultaneously, the Al atoms diffuse into the rear side and form a p⁺ layer at the rear of the device [5, 38]. Al forms a p⁺ layer between the aluminum electrode and the p-type wafer, which acts as a back surface field (BSF). The BSF reduces recombination at the silicon-metal contact by repelling minority carriers and increasing band-bending [24].

**Testing:** After the solar cell is finished, its performance metrics can be tested. Uncontrollable factors in the production line create small noise in the variables, which leads to small deviations in the solar cell. For instance for the wafer slicing process, there is a small error of about 5% on the thickness of the Si wafers [39]. For this reason, the PCE of the produced solar cells are distributed with a small standard deviation.

The process parameters can interact in drastic and unforeseen ways. As was shown on multiple instances, the design of a solar cell needs to carefully balance different types of losses. Progress in industrial PV was achieved by finding innovative ways to combine the processes such that high efficiency solar cells are produced. Naturally, the manufacturer would like to figure out which process recipes yield him
the best solar cell for the lowest cost. A crude way of searching for this optimum is by trial-and-error, also known as the Edisionian approach. The Edisionian approach is slow and might never find the optimum [7], as is illustrated in fig 2.2. fig 2.2(a) shows the result of the Edisionian optimization of two independent variables, which successfully finds the global optimum marked with a cross. However, in fig 2.2(b) the Edisionian approach finds a sub-optimal operating point for two a process with dependent variables. Since there are typically over ten processes with several interdependencies, the Edisionian method is ineffective for production line improvement. The following chapter will discuss response surface methodology, a more effective process improvement method.

Figure 2.2: Illustration of the pitfall of Edisionian (trial-and-error) optimization, which first optimizes variable 1 and then variable 2. (a) When variables are independent, Edisionian approach might find the optimum, (b) when variables are dependent, Edisionian optimization does not lead to optimal process configuration, (c) design of experiment finds the optimum effectively.

2.2. Response Surface Methodology

There exist many methods that improve upon the Edisionian approach. Process improvement is a statistical discipline that refers to methods for efficient optimization of processes. An important method for process improvement is response surface methodology (RSM) [9, 40]. RSM was first conceived by Box and Wilson in 1951 [41], but it is still the standard method for process improvement today [40]. RSM consists of three steps [7]:

1. Experimental data is collected containing sets of inputs and outputs of a process. Design of experiment minimizes the amount of observations needed in order to fit a predictive model.
2. A second or third degree polynomial function is fitted to the observations.
3. The predictive model is used as an objective function for some optimization algorithm that searches for the inputs that produce the optimal response.
In process optimization, *design of experiment* (DoE) data is used to efficiently extract information about a system or process [42]. DoE data is much more informative about the underlying system than haphazard trial and error [43] as is shown in fig 2.2(c). DoE is an open research area [7] but there is a consensus that DoE should be space-filling. This way, the optimum is found more effectively as less experiments are needed.

In DoE, controllable variables are called *factors* and their values are called *levels*. In DoE, factors are commonly measured at either two levels or three levels. Two levels measures primary effects and interactions and three levels measure main effects, quadratic effects and their interactions [7, 43]. Three levels gives more information, but comes at a cost of more experiments. Typically, the number of levels for all factors in the design is chosen to be the same [9] and since processes in PV manufacturing are typically complex, three levels per factor are preferred.

The most basic but rigorous DoE method is full factorial design (full FD) [7]. In full FD, an experiment is conducted for every possible combination of levels. Full FD reveals a lot of information about the observed process, but comes at a high cost. In the case of three-levels, the number of experiments (‘runs’) in full FD grows exponentially with $3^n$. Especially when multiple processes are analyzed, full FD quickly becomes impractical. Four factors and three levels needs $3^4 = 81$ runs, seven factors and three levels needs $3^7 = 2187$ runs and ten factors and three levels needs $3^{10} = 59,049$ runs. Since PV manufacturing processes typically have several variables of interest, full FD of individual processes seldom used to improve production lines.

Luckily, there are more efficient DoEs that need less experimental runs than full FD to extract similar information about the process [40]. One of the most widely used DoEs in process improvement are central composite design (CCD) and Box-Behnken design (BBD) [7, 8]. Instead of doing a full FD with three levels, CCD is created by combining a 2 level full FD and an extra set of points for the interactions as illustrated in fig 2.3. This way, the number of runs grows much slower. For CCD of two factors, this DOE is shown in fig 2.3 [44] and for three factors this is shown in fig 2.4(b).

BBD is generated by taking experiments at the edges of the process space and at the center, as is shown in fig 2.4(c) for three factors. Both these DoEs need less experiments than full FD, as is shown in fig 2.4.

However, even with these more efficient DoEs, sampling a complete production line is infeasible, since the number of experiments is still too big for today’s factories. For this reason, sampling of production lines is broken down into individual processes or a few processes at the time.

The second step in RSM is the fitting of a polynomial to the DoE data to establish
2.2. Response Surface Methodology

Figure 2.4: Three methods of DoE: (a) full factorial design with $3^3 = 27$ runs, (b) central composite design with 15 runs, (c) Box-Behnken design with 15 runs.

the relationship between responses and process configurations [9]. This is done with linear regression (LR), a method that is quick and well-understood. LR contains many elements that can also explain regression machine learning algorithms. For this reason, the theory behind LR is included in this chapter.

The collected DoE data can be represented as a table of $m$ input-output pairs $(x^{(i)}, y^{(i)})$ for $i = 1, ..., m$. Each pair $(x^{(i)}, y^{(i)})$ is called a training example and the set of $m$ training examples that are used for modeling the LR is called a training set, denoted as $(X, Y)$.

RSM assumes that there exists some mathematical relationship $f$ that represents the process which maps every input $x$ to a response $y$, defined as eq 2.1.

$$y = f(x) \quad (2.1)$$

The goal of LR is to approximate this functional relationship $f$ with the function $h$ such that $h(x)$ is a good prediction of the corresponding value $y$ [45]. This mathematical model can then be used to estimate the response $y$ of a process with a prediction $\hat{y}$, defined as eq 2.2. The quality of these predictions $\hat{y}$ depends on how much information about the process the model was able to extract from the experimental data.

To find the configuration of the process factors that improves the response, $h$ can then be the objective function for some optimization algorithm.

$$\hat{y} = y + \epsilon = h(x) \quad (2.2)$$

Where $\epsilon$ is the error of prediction. Naturally, the lower $\epsilon$, the better $h$ approximates the process of interest.

The most basic form of LR is univariate linear regression, in which a single response value $y$ is predicted based on a single feature $x$. In statistical literature, the function that needs to be fit on the data is called the hypothesis function $h$, which in univariate linear regression is represented as:

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x \quad (2.3)$$

This hypothesis contains one constant term and one variable term, creating a function with a straight line. The goal of univariate linear regression is then to fit this
straight line to the data, i.e. finding the values of $\theta = (\theta_0, \theta_1)$ that fit the trend of the data best, as is shown in fig 2.5. The bias $\theta_0$ helps to move the best fit line closer to the data. Without the bias term, the hypothesis would come from the origin.

The *goodness of fit*, a measure for how well a model describes the data, can be assessed with the cost function $J(\theta)$. $J(\theta)$ is defined as the average error $|y - \hat{y}| = \epsilon$ between the predictions $\hat{y}$ of the hypothesis and the true value of $y$:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (e^{(i)})^2 = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2 = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

(2.4)

Where $m$ is the number of data points. This value is also known as the mean squared error (MSE). The best values for $\theta$ are the ones for which the total sum of squared errors is minimized, or where the cost function is minimized.

The optimal values for $\theta$ can be found using the gradient descent algorithm, which minimizes the cost function. Starting at some initial point $\theta_{ini} = (\theta_0, \theta_1)_{ini}$, the algorithm updates $\theta$ by step-wise subtracting the partial derivative in every direction $j = 0, 1$, as in eq 2.5. The algorithm continues until convergence, i.e. until the cost function does not decrease more than a certain threshold value. Learning rate $\alpha$ determines the rate of the gradient descent and needs to be adjusted when the algorithm does not converge. For $\alpha$ too low, the algorithm takes too long until convergence and for $\alpha$ too large, the algorithm might not be stable enough to converge, as it takes too big steps and misses the optimum.

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

(2.5)
In the case of univariate linear regression, the algorithm would iterate until convergence for $\theta_0$ and $\theta_1$ according to eq 2.6:

$$\theta_0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta) = \theta_0 - \frac{2\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta) = \theta_1 - \frac{2\alpha}{m} \sum_{i=1}^{m} ((h_\theta^{(i)}) - y^{(i)}) x_i$$

(2.6)

Where the summation arises from the derivation of the cost function $J(\theta)$. The gradient descent of the cost function $J(\theta)$ of linear regression never gets trapped in a local minimum because $J(\theta)$ for LR is convex. Convexity is a nice property of real-valued functions which implies that the function has a single optimum and there are very effective methods of solving such problems.

LR can be extended to **multivariate linear regression**, which uses multiple variables to predict the value of $y$. The training examples $(x^{(i)}, y^{(i)})$ are denoted as $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, ..., x_n^{(i)})$ containing some value $x_j^{(i)}$ for each feature $j$. The features of the training set can be represented as a matrix $X$ as in eq 2.7.

$$X = \begin{bmatrix}
1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\
1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)}
\end{bmatrix}$$

(2.7)

The hypothesis for multivariate linear regression for training examples $x \in X$ is represented as eq 2.8 and can be abbreviated in matrix notation.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$= \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = x \theta^T$$

(2.8)

The hypothesis of the training set in eq 2.7 is denoted as an $m \times 1$ column vector $h_\theta(X)$:

$$h_\theta(X) = X \theta^T$$

(2.9)

Similar to univariate LR, the goodness of fit can be expressed with a cost function shown in eq 2.10. This quantity is also known as the mean squared error, an important performance measure for comparing regression ML algorithms. This summation can be further simplified in matrix notation, with $X$ as in eq 2.7 and $\hat{y} = [y^{(1)}, y^{(2)}, ..., y^{(m)}]^T$:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

$$= \frac{1}{m} (X \theta^T - \hat{y})^T (X \theta^T - \hat{y})$$

(2.10)
Then the gradient descent algorithm would be just like eq 2.5, repeating the following until convergence for the complete array of $\theta$:

$$
\theta := \theta^T - \alpha \nabla J(\theta)^T = 
\begin{bmatrix}
\theta_0 \\
\theta_1 \\
\vdots \\
\theta_n
\end{bmatrix} - \alpha 
\begin{bmatrix}
\frac{\partial J(\theta)}{\partial \theta_0} \\
\frac{\partial J(\theta)}{\partial \theta_1} \\
\vdots \\
\frac{\partial J(\theta)}{\partial \theta_n}
\end{bmatrix}
$$

(2.11)

For example, if we are predicting the PCE based on various process variables and $x_i$ represents ‘diffusion temperature’, then $\theta_i$ is in units of [PCE(°Cdiff)$^{-1}$]. The weights of all parameters $\theta_j$ will be adjusted by the gradient descent according to their relative importance for predicting $y$. For example, if the feature ‘diffusion temperature’ has a strong negative effect on the cell efficiency, the corresponding weight will be a large negative number.

The gradient descent finishes quicker when each feature is scaled or normalized [45], because for a large range of $x_j$, $\theta_j$ tends to change slower as a longer distance needs to be traversed. For a short range of feature $x_j$, $\theta_j$ tends to change quicker as a shorter distance needs to be traversed. Since learning rate $\alpha$ determines the learning rate for every feature, this might cause uneven rates of learning in each dimension, which in some cases could lead to slow convergence and in other cases no convergence. Two commonly used scaling methods: min-max scaling and mean normalization, defined in eq 2.12 [46].

$$
x_{j,minmax} = \frac{x_j - \min(x_j)}{\max(x_j) - \min(x_j)}
$$

$$
x_{j,mean} = \frac{x_j - \text{mean}(x_j)}{\text{std}(x_j)}
$$

(2.12)

Where $\min(x_j)$ the minimum value, $\max(x_j)$ the maximum value, $\text{mean}(x_j)$ the mean value and $\text{std}(x_j)$ the standard deviation of feature $j$. In min-max scaling, the minimum value of the feature is subtracted from each value in the feature columns and divided by the maximum value of the feature, scaling the range of values to [0,1]. Mean normalization subtracts the average from the feature vector and divides all values by the standard deviation, setting the mean of the feature column to 0 and its standard deviation to 1. There are numerous other scaling methods which each have their own niche advantage [46], but the analysis of their differences is beyond the scope of this research. In most cases, mean standardization is used, but for artificial networks min-max scaling is preferred.

Sometimes, the trend in the data is not a straight line, but for instance quadratic or cubic. In this case, linear regression does not have the capacity to fit the trend in the data [10] and polynomial regression is more suitable for modeling. In polynomial regression, not only the features $x_j$ are used for prediction, but also combinations of features $x_j \cdot x_i$ and squared or cubed features. Adding polynomial features to the hypothesis function $h_\theta(x)$ increases its capacity to fit a variety of trends, since we can now make quadratic hypotheses $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$ or cubic hypotheses: $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$, where $x^3$ and $x^2$ are additional features that are based
on $x$. With polynomial features, scaling is even more important, as the magnitude of the features is increased by the order of the polynomial. It is important to note, that even though a polynomial regression uses polynomial features of degree larger than 1, it is still a \textit{linear} combination! The term ‘linear’ is an overloaded term, as it often is understood as ‘straight’, but in this case it is meant as a linear combination.

Most importantly, the hypothesis needs to generalize well. Fig 2.6 shows three univariate polynomial fittings to a data set that was generated with a quadratic function with some noise. When a hypothesis is too simplistic or uses not enough features, it does not have the capacity to fit the trend of the data and we have a case of underfitting, as is shown in fig 2.6(a). For both the training examples and its predictions of new data, it would make large errors. When a model does not generalize well, it suffers from overfitting, caused by a too complicated function that creates a lot of unnecessary curves [45]. An example of overfitting is shown in fig 2.6(c). In this case, the model has memorized the data including the noise, and it will be useless for making predictions. Fig 2.6(b) shows a model with the right capacity, which fits the experimental data well and makes good predictions.

Overfitting can be solved by regularization or reducing the number of features that are used for fitting of the model. Regularization is a method that adds an extra penalty to the cost function $J(\theta)$ that is based on the magnitude of the parameters $\theta$, as is shown in 2.13. Regularization works because the lower the parameters $\theta$, the less complicated and curvy it becomes [45, 47]. If the regularization is too low, the hypothesis will not only fit the trend of the data but also the noise in the data [10]. Too large regularization pushes the weights to zero, leading to an oversimplified model. With regularization, minimizing the cost function of eq. 2.10 becomes:

$$
\text{minimize} \frac{1}{n} \left[ \sum_{i=1}^{n} \left( h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^{k} \theta_j^2 \right] 
$$

(2.13)

Here, $\lambda$ is the regression parameter, which determines the size of regularization. For large values of $\lambda$, the penalty for high parameter values is high and the function becomes simpler. This way, there will be less unnecessary use of parameters and the function generalizes better. For a too large $\lambda$, the model starts to underfit. Regu-
larization is used in several ML algorithms, such as artificial neural networks and support vector regression.

After a predictive model of the process is established, it can be used for process improvement. A way of obtaining some basic information about the model is by inspecting its shape and the value of its weights. Given that the data is normalized, a large weight for a linear term indicates a strong primary effect of a factor on the response. A large weight for a quadratic term indicates a nonlinear response. A large weight for a cross product term is interpreted as a change in the (linear) effect of one factor as a function of the other factor and vice versa [48]. The shape of the model, such as linear, quadratic or cubic, can tell us some information about possible ways to improve the process.

A more advanced application of the predictive model to optimize it. It is important that this model is very accurate, otherwise this inaccuracy will also affect the quality of the optimization. Given that the model is an accurate approximation of the true relationship between process variables and responses, we can optimize the process to produce cells with improved PCE. There exist many optimization algorithms, but not all are equally useful. Traditional optimization algorithms rely on gradient descent algorithm. However, gradient descent can get stuck in a local minimum when the objective function is non-convex. This is a very common problem for optimizing processes, as convex multivariate hypotheses are rare [9]. For this reason, there is a preference for using so-called metaheuristic optimization algorithms [49], which are more stochastic in nature and therefore have less tendency to get stuck in local optima. The metaheuristic that will be used in this thesis is genetic algorithm (GA). GA has gained much attention over the last two decades [9]. AG is very useful when conventional optimization methods are not suitable such as high dimensional spaces with many local optima [9]. GA does not rely on gradient-based information, so it does not require the continuity or convexity of the design space. It can be used on both continuous and discrete functions.

GA is based on Darwinian evolution: you start out with a set of random data points drawn from the variable space, called a set of individuals in the population. Then, the fitness of each individual is predicted with your model \( g(\mathbf{x}) = \hat{y} \), which is in this study the PCE of a production line. In GA, the principle of survival of the fittest is applied: the fittest individuals of your population are allowed to survive and pass their DNA on to the next generation. Each individual in the new generation has a small chance of mutation from its parents, giving it slightly different variables and thus a different fitness. Ever so often, a random mutation happens that produces individuals with a better fitness than its parents, and then this individual will survive and pass on its DNA to the next generation. By repeating this algorithm for many generations, eventually new process configurations can be found to improve the PCE.

Even though many processes have been improved with RSM, LR is not always the most efficient way to model experimental data. A hypothesis of three variables with quadratic terms, would give us 10 features:

\[
h_\theta(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_1^2 + \theta_5 x_2^2 + \theta_6 x_3^2 + \theta_7 x_1 x_2 + \theta_8 x_2 x_3 + \theta_9 x_1 x_3
\]
In general, the number of features for an order r linear regression with n features can be expressed with the function [45]:

$$N = 1 + \sum_{k=1}^{r} \frac{(n + k - 1)!}{k! (n - 1)!}$$  \hspace{1cm} (2.14)

For a process with 50 factors and cubic terms, this would need 23,426 features to be trained. The number of features quickly becomes impractical: the large amount of features tends to overfit and it is computationally expensive. Furthermore, there is typically not enough data to train all those features with. If you would only include a subset of these features, you risk losing the capacity to fit the data.

Other problems with linear regression include [7–9, 43, 48]:

- (Polynomial) linear regression has a strong tendency to overfit when modeling greater than 10 variables.
- Use of higher order polynomials leads to poor predictions when you move away from the experimental observations, as illustrated in fig 2.6.
- Linear regression often demands a priori assumptions regarding functional relationships of the data that is hard to estimate, especially for new and complex processes.

To reduce the complexity of the problem, in practice production lines are improved by optimizing a few processes at once [48]. However, using this approach, the interdependencies between processes are not included into the predictive model, and the optimization search space is significantly limited. However, as we have seen at the first section of this chapter, there exist many interesting interactions between processes which influence the PCE.

Recent RSM research has focused on sequential optimization, in which the last machine is optimized first, and then the inputs of the machine before are optimized according to the settings of that machine [40]. However, there is one main problem with this approach: it requires many assumptions about which variables are important, which is not always known and can be subjective [9]. Another option is making a Taylor series to model the data, but as processes get more complicated, this option becomes too difficult to compute [43].

### 2.3. Machine Learning

Linear regression is part of a broader class of machine learning (ML) algorithms. In general, ML refers to algorithms that perform a specific task without explicit instructions. Typically, ML algorithms infer patterns and rules from data sets with examples. There exists a large variety of ML algorithms with each their own advantages and disadvantages. For every type of problem, a different ML algorithm is more suitable. The shortcomings of linear regression can be overcome by using another ML algorithm.

There are several categories of ML algorithms, but this thesis focuses on supervised
**machine learning.** In supervised ML, the ML algorithm establishes a predictive model based on a data set that contains both features $X$ (input variables) and their corresponding targets $Y$ (response values). From this point in the thesis, ML refers to ‘supervised machine learning’. Just like in linear regression, the ML algorithm learns a hypothesis $h : X \rightarrow Y$ that makes good predictions of $Y$ based on $X$ [45].

Supervised ML can be used for both regression and classification problems. In regression problems, a continuous value is predicted and in classification problems a discrete value is predicted. Most of the algorithms that we analyze in this thesis are all regression algorithms since the goal is to predict PCE based on the process recipes used to produce it. An example of a supervised classification problem is learning whether a picture contains a cat or a dog, based on thousands of photos that are labeled as containing cats or dogs.

From now, linear regression will be referred to as POLY, to prevent confusion with another well known ML algorithm called logistic regression. The ML algorithms that were tested next to POLY in this thesis are:

- Artificial neural network (ANN)
- Support vector machine regression (SVR)
- Decision tree (DT)
- Random forests (RF)
- Adaptive boosting (AB)

The reason these algorithms were chosen is because they work well on a large variety of problems and seem generally preferred among ML researchers [50]. These algorithms outperform linear regression in many ways. They are able to model a broader class of relationships and can handle more features than POLY [10]. The goal of this section to provide some intuitive insight into how these algorithms work.

Although applications of ML in PV and semiconductor manufacturing have been previously investigated [9, 11–16], it is generally for optimization of a limited number of processes. Furthermore, in these studies, a comparison between different algorithms is missing and only small data sets are used for verification. Several of these studies implement ANN, however, the motivation for choosing this algorithm is lacking. In this study, we aim to provide new insights into the performance of various ML algorithms in process optimization and explore their capacity to model a complete solar cell manufacturing line.

Central to several ML algorithms is the sigmoid function. The sigmoid function is defined as in eq 2.15:

$$ g(z) = \frac{1}{1 + e^{-z}} $$ (2.15)

Fig 2.7 shows that the sigmoid function has an S-shape. The sigmoid function maps real-valued numbers $z$ to a value in the range $(0,1)$. The sigmoid is a powerful tool for transforming linear regression estimation into binary classification. In eq 2.16,
the linear regression hypothesis is put in the sigmoid by using $z = x\theta^T$ to create a classification hypothesis.

$$h_\theta(x) = g(x\theta^T) = \frac{1}{1 + e^{-x\theta}}$$

(2.16)

This hypothesis is called **logistic regression**, which is a confusing name since it is a classifier. The outcome of the sigmoid can be interpreted as the probability that some observation is from class 1. For instance, if $x\theta^T = 0$, $h_\theta(x) = 0.5$ and if $x\theta^T = 1$, $h_\theta(x) \approx 0.73$. Logistic regression classifies examples based on a simple rule: when $x\theta^T \geq 0$ or $h_\theta(x) \geq 0.5$ the sample is classified as ‘+’, and when $x\theta^T < 0$ or $h_\theta(x) < 0.5$, it is classified as ‘-’. In other words, at $\theta^Tx = 0$, we have a decision boundary between classification of ‘+’ and ‘-’.

To highlight the working of the LR hypothesis, consider an example where the weights are trained such that $\theta = [4, -2, 0]$. Then, $h_\theta(x) = 1$ if:

$$4 - 2x_1 + 0x_2 \geq 0 \iff x_1 \leq 2$$

In this case, the decision boundary is a vertical line at $x_1 = 2$ and all samples with $x_1 > 2$ will be classified as +. So the location of this decision boundary in the feature space $X$ is dictated by the weights $\theta$.

The hypothesis makes a prediction error when it classifies a sample as ‘+’ when it is ‘-’ or it classifies ‘-’ when it should be ‘+’. This is illustrated in fig 2.8(a) for both cases. When the logistic regression hypothesis is trained, the weights $\theta$ are adjusted such that the decision boundary makes a minimum amount of classification errors are made. However, like fig 2.8(a) shows, sometimes prediction errors are hard to prevent. To reach minimal error, a gradient descent algorithm is used such that all training examples are classified with the least overall cost. This gradient descent algorithm is slightly different than for linear regression, as taking the derivative of the sigmoid function would result in a non-convex function. Instead, the function shown in 2.8(b) is used, which is low for ‘good’ predictions and increases sharply for ‘bad’ predictions.

Another example is a logistic regression hypothesis that approximates the OR logic gate. This hypothesis takes two binary values $x_1$ and $x_2$, and can be visualized in a network structure shown in fig 2.9. The weights of this hypothesis are trained such that $\theta = [-20,30,30]$. When $x_1 = 0$ and $x_2 = 1$, the outcome of the sigmoid will be...
Machine Learning Approach to Process Optimization

Figure 2.8: Illustration of (a) misclassification and (b) the cost function of the sigmoid hypothesis.

\[ g(-20 \cdot 1 + 30 \cdot 1 + 30 \cdot 0) = g(10) \approx 1. \] Similarly, when \( x_1 = 1 \) and \( x_2 = 0 \), \( h_\theta(x_1, x_2) \approx 1 \), when \( x_1 = 1 \) and \( x_2 = 1 \), \( h_\theta(x_1, x_2) \approx 1 \) and when \( x_1 = 0 \) and \( x_2 = 0 \), \( h_\theta(x_1, x_2) \approx 0 \).

Several other logic gates can be created using logistic functions, however problems arise when trying to model more advanced logic gates like the XOR (“exclusive or”) gate, which takes two binary inputs and is 1 when exactly one input is 1 and 0 otherwise. When learning the XOR function, we want the network to correctly classify the examples \( x = [x_1, x_2] \in X = \{[0, 0], [1, 0], [0, 1], [1, 1]\} \). However, you cannot place one decision boundary between these four points that separates all points correctly, as is shown in fig 2.10. Instead, gradient descent will create a hypothesis that simply always outputs 0.5, which minimizes the overall cost function [10]. The crux of this problem is that when \( x_1 = 0 \), the model’s output should increase as \( x_2 \) increases. However, when \( x_1 = 1 \), the model’s output should decrease as \( x_2 \) increases. Since the model needs to assign a fixed value to weight \( \theta_1 \), it minimizes the total loss by assigning 0.5 to it. This problem can be addressed by expanding the network with an extra function, which leads us to artificial neural networks.

Artificial neural networks [51–53] is a supervised ML algorithm that can be used for both classification and regression. It consists of a network of functions similar to the sigmoid function that allows it to learn a large variety of functional relationships. It can even learn non-linear patterns, such as the XOR gate.
ANN can approximate the XOR operator by transforming the feature space with another layer, as is shown in fig 2.11. This extra hidden layer between the input nodes and the output node is called a hidden layer. The hidden layer consists of activation nodes, which work very similar to the sigmoid function. Their function is to transform the linear combination of inputs into a new representation. Four important activation functions are the identity function (fig 2.12(a)), the logistic function (2.12(b)), the tanh function (fig 2.12(c)) and the rectified linear (ReLU) function (fig 2.12(d)). With the right weights and activation function, both $x = [0, 1]$ and $x = [1, 0]$ can be collapsed into a single point. For example, for $\theta_{11} = +10, \theta_{12} = -10, \theta_{21} = -10, \theta_{22} = +10, \theta_{1} = +5, \theta_{2} = +5$ and sigmoid activation functions, we get the truth table shown in table 2.1 [10].

The ANN network architecture can be extended to any amount of layers with any amount of nodes per layer. Too many hidden layer neurons lead to overfitting, while too little neurons reduces the capacity of the hypothesis. Furthermore, more hidden layers is computationally more expensive. Networks with many hidden layers are called **deep**, which enters the domain of deep learning.
more training data is needed as there are extra weights to be trained. A suitable hidden layer structure is typically found by trial-and-error.

The ANN trains with a special type of gradient descent to train its weights to improve its estimations. For different problems, different solvers should be used. For smaller data sets with less than 1,000 examples, limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) is the preferred solver \[50\]. For larger data sets with thousands of examples, stochastic gradient descent (SGD) and Adam are preferred [50]. These solvers all have stochastic properties, which prevents the cost function from getting stuck in a local minimum. Just like the gradient descent for linear regression, the ANN regularization parameter \( \alpha \) controls overfitting by penalizing the weights of the network.

**Support vector machines** [54–56] are typically used for (binary) classification, but SVM can also be used for regression. In support vector regression (SVR) [57], the goal is to find a hypothesis \( h(x) \) that for all training examples \( x \) deviates at most \( \epsilon \) from the targets \( y \), and meanwhile is as flat as possible \[47\]. It does not matter how large the errors are, as long as the hypothesis predictions do not deviate too much from the target values.

The basic form of SVR uses a linear function that is denoted as the product between weights \( \theta \) and the features of sample \( x \): \( h_\theta(x) = x\theta^T \). To make this hypothesis as flat as possible, an optimization is solved to find a hypothesis that minimizes the weights \( \theta \) while its predictions deviate not more than \( \epsilon \) from the target values. When it is impossible to find this hypothesis, slack values \( \xi_1 \) and \( \xi_2^* \) are added to cope with infeasible constraints of the optimization problem [47]. The formal definition of the
Figure 2.13: An illustration of the SVR with linear kernel. The training targets are allowed to deviate at most $\epsilon$ from the hypothesis. For infeasible examples, slack values $\xi_i$ and $\xi_i^*$ can be used [47].

SVR optimization problem is shown in eq. 2.17.

$$\begin{align*}
\text{minimize} & \quad \frac{1}{2} \|\theta\|^2 + C \sum_{i=1}^{m} (\xi_i + \xi_i^*) \\
\text{subject to} & \quad y_i - x_i^T \theta \leq \epsilon + \xi_i \\
& \quad x_i^T \theta - y_i \leq \epsilon + \xi_i^* \\
& \quad \xi_i + \xi_i^* \geq 0
\end{align*}$$

(2.17)

Where $C > 0$ is a regularization parameter that determines the trade-off between the flatness of the function and the amount up to which deviations larger than $\epsilon$ (outliers) are tolerated [47]. Large $C$ leads to small regularization with low tolerance for outliers. Small $C$ leads to strong regularization with high tolerance for outliers. The SVR is illustrated in fig. 2.13. The choice of $\epsilon$ is important, as too large $\epsilon$ creates too loose optimization constraints which leads to high bias. Too small $\epsilon$ places too strict optimization constraints which leads to high variance.

The most basic form of SVR uses a linear function, but the SVR can be made nonlinear by using a different kernel $\phi(X) \rightarrow F$. The kernels map the original training examples $X$ to a new feature space $F$, which enable to make nonlinear estimator [47]. Commonly used kernels include the polynomial kernel, the radial base function and the sigmoid kernel, which all map the training examples to a feature space in a different way. The radial base function is expected give the estimator the largest capacity to learn functions.

Random forests [58–60] in contrast to linear regression, SVR and ANN, RF relies on inferring decision rules about the data [61] by using decision trees for prediction. The working of a decision tree (DT) is similar to a game of ‘Guess Who?’: decision rules like ‘man’ or ‘female’, ‘old’ or ‘young’ will eventually lead to a good estimation of which person the opponent had in mind. DTs are trained such that the decision rules split the data up as pure as possible. Not every decision rule is equally informative: some decision rules split up the data better than others. In binary classification, a decision rule that splits up the data in 95% ‘+’ and 5% ‘−’ training examples is very informative, while a decision rule that splits up the data in 50% ‘+’ and 50% ‘−’ is very uninformative. Normally, DTs are used for classification, but by dividing the continuous line in small bins, the hypothesis can be used for regression.

The working of RF is based on bagging, which uses a large number of sufficiently
good DT estimators and then takes the average of their predictions. For effective RF prediction, there should be enough variance in the estimators. Otherwise, when all estimators would be equal, there is no point in having an ensemble of them. An important technique in RF is *bootstrapping*, which draws a random selection of examples from the training data with replacement, to introduce randomness to the different estimators. Other methods include leaving out some of the features to train with.

**Adaptive boosting** [62] is similar to RF. AB turns a group of weak estimators, for instance decision trees, into one strong learning algorithm. AB uses *boosting*, which differs from bagging in the way the weak estimators are used. Rather than training weak estimators in parallel and then taking the average of their prediction, boosting takes the weighted sum of all weak estimators. Boosting can be used for image recognition. For instance, the boosting algorithm can combine the estimator for horizontal lines, vertical lines and circles into one estimator. The individual estimators could not recognize much, but taken together they can accurately recognize eyes in a photo for example.

**Conclusion**

The manufacturing of PV devices can be viewed as a combination of consecutive processes that transform silicon, chemicals, gasses and metals into a solar cell. Each process follows a recipe of chemical and mechanical operations that collectively determine the properties of the solar cell. If a manufacturer wants to improve the quality of his solar cells, he is faced with a complex optimization problem: the processes are highly inter and intra-dependent. Any operation in any process could have some hidden trade-off with an operation in another process. In order to improve the quality of the produced solar cells, each process needs to be optimized in coordination with all other processes. This requires advanced predictive models that are capable of including all possible intra and inter-dependencies of the processes.

Process improvement is an important discipline that aims to identify which of the variable inputs optimize a process. One of the most important methods for process improvement is response surface methodology (RSM). RSM consists of three steps: (1) collection of DoE data, (2) modeling the DoE data with linear regression and (3) using the linear regression model in a optimization to improve the process. Using RSM, processes can be improved quickly as linear regression is a well understood and fast prediction model. However, the use of linear regression for process improvement of solar cell manufacturing is debatable. The quality of the optimization algorithm depends on the quality of the mathematical model, and linear regression seems unfit for modeling PV manufacturing. First of all, linear regression lacks the capacity to fit a wide variety of functions, which means that some relationships are too complicated to model. Secondly, linear regression is unfit for modeling a large number of features. Thirdly, linear regression has bad extrapolation properties, because the model typically bends off when moving away from the data. Extrapolation properties are quite advantageous as it allows to optimize a little bit beyond the data that was
modeled. Because of these shortcomings, in practice, RSM is used for optimization of small groups of processes, neglecting the interesting inter-dependencies between processes.
Various machine learning (ML) algorithms are discussed that could improve RSM by replacing linear regression. The ML algorithms that are discussed in this chapter have nice properties that make them very suitable for modeling solar cell manufacturing processes. For example, they can model more complex relationships or allow more features to be modeled. This enables better process optimization as ML models can model a larger range of processes in the production line.
The major objective of this thesis is to study production line improvement with machine learning (ML) algorithms. This has been done by training several ML algorithms on simulated production data to predict the photovoltaic cell efficiency (PCE) based on process variables. ML algorithms are known to model complex relationships in high dimensional data, so the aim of this study is to test whether ML algorithms can model the complex inter-dependencies between processes in the production line. To compare the performance of different ML algorithms, an important aim for this thesis was to develop software to simulate photovoltaic (PV) production data. Using this production data, several ML algorithms were tuned, trained and tested and their performance was measured. This chapter explains the framework in which several ML algorithms were tuned, trained and tested using simulated factory data.

### 3.1. Data Simulation

The main challenge of this thesis was to obtain useful data to test different ML algorithms with. Companies are reluctant to share their production data and dependency on a third party for data allows very little control in the quality of the data. For these reasons, software was developed to simulate production data with. The PV Factory simulation software was developed in collaboration between the university of New South Wales and PV Lighthouse [24, 63]. PV Factory is an online educational tool for learning about industrial manufacturing of Si solar cells. The virtual production line (VPL) that was used for this thesis uses the equations from PV Factory to model all processes with. The VPL simulates the industrial production of single-crystal aluminum-back surface field (Al-BSF) solar cells, the design that is still used in many factories today. Some processes that VPL uses are outdated, but that does not matter here: the goal is to create production data for ML testing, not to simulate a state of the art PV manufacturing. Moreover, the results are expected to be same when different equations for process modelling are used. The VPL includes the processes that were discussed in chapter 1, shown in fig 2.1. Each process has several input variables which determine various properties of the
solar cell. An overview of all 50 process variables is shown in table 3.1. The boundaries of the process variables guarantee that no non-physical values can be input and no non-physical solar cell properties are simulated. The first process determines various wafer properties like bulk resistivity, doping and carrier lifetime. The processes are operated sequentially and solar cell properties that are modelled in early processes also affect later processes, just like real solar cell manufacturing. The performance metrics of the solar cell are modelled with PC1D.

The original VPL software was written in the programming language C and part of research is devoted to converted to Python for easier use. Since the equations that model the VPL processes are protected, the are not shared in this thesis.

Fig 3.1 shows an example of the realistic interactions between process variables in the VPL. The etching process variables imitate the dependency that exists in the real etching process. For too low temperature and/or short etching time, the etching process will be ineffective and the saw-damage cannot be removed. For too high temperature and/or too long etching time, the etching process will be too rigorous and the Si wafer breaks.

The VPL models fourteen solar cell properties that are used to simulate the performance metrics in PC1D [64]. In order to simulate large amounts of data, a modified version of PC1D, modPC1D6-2, was used that was developed by Halvard Haug et al. [65]. modPC1D6-2 enables automatic modelling of solar cell characteristics from a scripting environment. The VPL can fabricate and measure 5,000 virtual solar cells per hour, or 120,000 virtual cells per day. The wafer properties that VPL uses for measurement with PC1D are shunt resistance, shunt diode saturation current, peak emitter concentration, emitter thickness, bulk doping concentration, back surface field peak doping, back surface field thickness, anti-reflective coating (ARC) thickness, the refractive index of the ARC, the bulk lifetime, the emitter sheet resistance and the base resistance. Furthermore, an external reflection file is created that contains reflection properties of the ARC for each wavelength from 5-1200 [nm]. All other
The VPL can produce data randomly and according to an experimental design. To produce data randomly, the user selects which variables to change. Then, the user chooses uniform random or normal distribution. Then, for every variable \( j \), the user chooses a nominal value \( M_j \) and a boundary percentage \( p_j \). In the case of uniform random distribution, the random values for variable \( j \) will be drawn uniformly random from the range \([M_j - M_j \cdot p_j, M_j + M_j \cdot p_j]\). In the case of normal distribution, the random values will be drawn from a normal distribution with mean \( M_j \) and standard deviation \( p_j \cdot M_j \).

The VPL can also produce data according to DoE. For this, you can select which process variables to include in the DoE, which design you want to use and some \( M_j \) and \( p_j \) for every variable \( j \). Then for every variable, DoE is performed on the range \([M_j - M_j \cdot p_j, M_j + M_j \cdot p_j]\). One can select full factorial, Box-Behnken and central composite design DoEs. Fig 3.1 shows that some processes have very strong PCE deviation on the edge cases. For most variables, the DoE is ineffective when analyzing on a large range. The DoE is often more effective for limited range.

### 3.2. Evaluation of Machine Learning Algorithms

The performance of machine learning algorithms generally increases when adding more data, but not always is extra data the solution [45]. Often using different algorithm settings, called hyperparameters, will drastically increase the performance of an algorithm. Other ways of improving the algorithm are for example using more (or less) features or changing the regularisation. There are various methods available to test what needs to be changed about the algorithm. In general, an algorithm will suffer from either overfitting or underfitting, as is illustrated in fig 2.6.

An ML hypothesis is good when it generalizes well. If an algorithm accurately fits the training set, it does not automatically imply that it is a good hypothesis [45]. When the algorithm performs good on examples that it was not trained on, it is a good estimator. When an algorithm is overfitting, the algorithm has a low error for predicting the training examples but it is still inaccurate for data that was not seen during training. When an algorithm is making large errors for both the training set and the test set, it is underfitting. Whether an algorithm overfits or underfits is determined by its capacity to fit a wide variety of functions. When a model is too simplistic to capture the trend of the data, it has low capacity. An example of this is a straight line that tries to model a quadratic function. In this case, the model makes large prediction errors on both the training examples and the testing examples. When the capacity of the model is too high, it is too complex and starts fitting all meaningless noise that is present in the data.

To determine whether an algorithm is overfitting or underfitting, we can split the data up in a training set and a testing set, as is shown in fig 3.2(a). Then, we can train the algorithm to minimize the cost function \( I_{\text{train}}(\theta) \) according to eq 3.1 using the training data as is shown in fig 3.2(b). Then, the generalization of the algorithm is assessed using the test data, to make a non-optimistic measurement of the predictive qualities of the model, fig 3.2. This generalization can be measured by measuring the cost function \( I_{\text{test}}(\theta) \) on the testing data in eq 3.1. Without testing, we make
Figure 3.2: Procedure of regression machine learning: (a) splitting examples in training and testing set, (b) fitting the model on the training set, (c) predicting 'new' examples from the testing set.

too optimistic measurements of the performance of an ML algorithm. $J_{test}(θ)$ is an important performance measure to compare ML algorithms with. It is also known as the mean squared error (MSE). In most regression ML algorithms, the performance is measured with the root mean squared error (RMSE), shown in 3.2.

$$J_{train}(θ) = \frac{1}{m_{train}} \sum_{i=1}^{m_{train}} \left( h_θ(x_{train}^{(i)}) - y_{train}^{(i)} \right)^2$$

$$J_{test}(θ) = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} \left( h_θ(x_{test}^{(i)}) - y_{test}^{(i)} \right)^2$$

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i)^2}$$ (3.2)

An important part of ML research involves trying out different hyperparameters for the ML algorithms. The hyperparameters represent the different settings a ML algorithm has. Some hyperparameters are very suitable for small data sets, or specialize in non-linear patterns. Some hyperparameters improve the speed of the algorithm, or change regularization. During hyperparameter tuning, we split the data in a training set, a testing set and a cross-validation (CV) set, as is demonstrated in fig 3.3.

Typically, hyperparameters adjust the capacity of the ML hypothesis. For instance, changing the regularization parameter of the hypothesis will penalize weights more or less. In general, there is a sweet-spot for every hyperparameter where it has the best generalization. Here, we use the principle of Ockham’s razor [66]: if you can choose between two models with equal performance, choose the simplest model. In general, your model improves by increasing its capacity. However, after some point, Ockham’s razor is crossed and the model becomes complexer than the problem. From this point, prediction increase again. In general, the cross-validation error $J_{CV}(θ)$ has a shape like shown in fig 3.4 A prediction error constitutes of bias (underfitting) and variance (overfitting), and in order to improve the hypothesis, we need to know whether it suffers from bias or variance. The intuition for these names comes from the fact that complex models are sensitive to which data you put in, which on a linear
3.2. Evaluation of Machine Learning Algorithms

Figure 3.3: Schematic overview of five-fold CV, image from [50]. First, the data is split randomly in training and testing data. Then, the training data is split in five folds. Then for five splits, each time the training data is split into four folds for training and one fold for testing. During these five splits, the ML algorithm is trained and tested to determine its parameters. Then, the ML algorithm is tested a final time on the testing data to measure its generalization.

Figure 3.4: Illustration of the difference between underfitting (high bias) and overfitting (high variance): (a) According to Ockham’s razor, the predictive model should be as simple as possible; (b) an hypothesis that suffers from high bias will not be improved by training on more examples (c) an hypothesis that suffers from high variance does improve with more training examples.
line would be high variance and low bias. Simpler models are more rigid and do not change that much by changes in the data, so they have high bias and low variance. The more complex the hypothesis becomes, the lower its error on the training set as it gains the capacity to model finer patterns in the data. The test error will also decrease as you increase complexity, but at some point the test error starts increasing again. At this point, the model becomes too complex for the problem, and Ockham’s razor is disregarded. For high bias, the prediction error of both training set and CV set will be high and $J_{CV}(\theta) \approx J_{train}(\theta)$. In general, more training examples will not improve the model as its capacity is exhausted. This is because the model is both too simplistic for the training and testing data. For high variance, the prediction error of the CV will be greater than the test error and the learning curve shows that the learning curve still can improve for more data.

Bias and variance are useful signals for diagnosis of the hypothesis. A high variance means overfitting, which could be helped with reducing the number of features, adding more data or adding more regularization. Feature reduction could help the algorithm to ignore useless variance in the data, more data will reduce the importance of single outliers and reduction of regularization allows the model to use bigger weights. A high bias cannot be helped with adding more data as the hypothesis already does not have the capacity to model the current examples. Instead, a more complex hypothesis is needed, by adding more features or reducing the regularization. After you know which problem the algorithm is suffering from, you can take steps by looking at the learning curves as a function of data size, regularization $\lambda$ and amount of features.

So, more training examples will fix the high variance, lower amount of features fixes high variance, adding features fixes high bias, decreasing $\lambda$ fixes high bias, increasing $\lambda$ fixes high variance. Adding polynomial features fixes high bias. Much of machine learning revolves around finding the right type of algorithm for your problem. Since every algorithm has many possible settings, this is often a long procedure of trial and error. Machine learning also involves the development of sophisticated features, which are not originally present in the data. It is advised to start with a simple algorithm and test it out of the box. Then you plot learning curves to decide if you need more data, more features, etc. Then you examine the errors on the examples in the CV set and try to spot a trend.

### 3.3. Experiment I: Comparison Between Algorithms

The first experiment tests which ML algorithm can model the VPL most accurately. The steps in this experiment are visualized in fig 3.5. The VPL is used to simulate random process data for all 50 variables, creating a wide distribution of PCE between 0% and 21%. This data set is then used to train five ML algorithms with. The ML algorithms are trained to predict PCE of a solar cell based on its process variables. The ML algorithms that are tested in this research are linear regression (POLY), artificial neural networks (ANN), support vector regression (SVR), random forests (RF) and adaptive boosting (AB). For machine learning, we used the Python module ‘scikit-learn’ [50]. From scikit-learn we used the function ‘MLPRegressor’ for ANN, ‘SVR’ for SVR, ‘AdaBoostRegressor’ for AB, ‘RandomForestRegressor’ for RF and
3.4. Experiment II: Modeling of Manufacturing Data

Figure 3.5: In experiment I, five algorithms are compared by tuning their hyperparameters in five-fold CV on 30,000 data points, then train them on an increasing range of set sizes from 10 to 100,000 data points and then they are tested on a testing set.

‘Ridge’ for LR. For standard scaling we used ‘StandardScaler’, for min-max scaling we used ‘MinMaxScaler’. In all cases, the scikit-learn function ‘make_pipeline’ was used to prevent data leakage due to feature scaling. The performance of all ML algorithms is assessed with the root mean squared error (RMSE) and the coefficient of determination ($R^2$) that are scored on the testing data. These five different ML algorithms are first tuned. During tuning, the settings of each ML algorithm are optimised for the data in order to make a fair comparison between algorithms. If an ML algorithm has advantageous default hyperparameters, it would have an unfair advantage and trouble the comparison. The hyperparameter optimization is performed in a grid search with five-fold cross validation.

Using the VPL, a data set of 30,000 random process configurations is simulated and their performance metrics are modeled with PC1D. This data set is then used for tuning the hyperparameters of each algorithm to make a fair comparison. The tolerance is the stopping criterion for convergence and determines how long the training should continue. For too low tolerance, the algorithm starts overfitting. The following hyperparameters are optimized. For POLY, regularization and tolerance. For ANN, hidden layer structure, regularization, tolerance, polynomial features, solver and activation function. For SVR, $\epsilon$, C, kernel function. For AB, number of estimators, learning rate and various decision tree hyperparameters. For RF, number of estimators, DT hyperparameters, amount of features, and whether to bootstrap.

After the hyperparameters are tuned, each algorithm is analyzed in a learning curve to analyze the underlying bias and variance of each ML algorithm. The VPL is used to simulate a data set of 100,000 random process configurations and all algorithms are trained on increasing portions of the data set. First, all ML algorithms are trained and tested on 10 data points, then 100, 1,000, 10,000 and finally 100,000. This is called a learning curve, which highlights bias and variance problems for every ML algorithm. The testing RMSE of these experiments are then plotted in a learning curve. Using the learning curve, various analyses are performed to investigate underlying bias or variance problems of each algorithm. Based on this experiment, it will be determined which algorithm is most suitable for modeling PV manufacturing processes. This best performing algorithm will be denoted as ML$^*$.  

3.4. Experiment II: Modeling of Manufacturing Data

The data set simulated in experiment I has a broad distribution of PCE and many examples. In reality, the type of data set used in experiment I is not available to companies because experiments are expensive and time is limited. In contrast, factories typically have very small deviation in their production line data. An industrial
Methodology

The data set for the second experiment was produced by operating each process around a nominal operating \( \mu_j \) with standard deviation \( \sigma_j \). The genetic algorithm then optimizes on \( \mu_j \pm 5\% \) (purple), \( \mu_j \pm 10\% \) (blue) and then \( \mu_j \pm 20\% \) (green) and also \( \mu_j \pm 30\% \) (not shown).

Figure 3.6: The data set for the second experiment was produced by operating each process around a nominal operating \( \mu_j \) with standard deviation \( \sigma_j \). The genetic algorithm then optimizes on \( \mu_j \pm 5\% \) (purple), \( \mu_j \pm 10\% \) (blue) and then \( \mu_j \pm 20\% \) (green) and also \( \mu_j \pm 30\% \) (not shown).

The goal of experiment II is to model an ML algorithm on this narrowly distributed data set and then to optimize the process using this predictive model. The steps in this experiment are visualized in fig 3.7. This data set is created by allowing small random noise on each process variable. For most variables, this standard noise is modeled with a 5% standard deviation. As illustrated in fig 3.6, in the simulation variable \( j \) operates around the nominal value \( \mu_j \) with a normal random noise with standard deviation \( \sigma_j = \mu_j \cdot 0.05 \). This nominal value \( \mu_j \) is denoted in table 3.1 as the default values. However, for some variables like 'diffusion temperature' and 'co-firing temperature', a standard deviation of \( \sigma_j = \mu_j \cdot 0.01 \) is more appropriate since their parameters are in the range of 800-1000. In reality, this data set could be obtained by having a fully tracked and sensed production line measuring the actual value of every process variable.

The random noise on each variable leads to small variance in the distribution of PCE of the produced wafers. The data set is pre-processed by scaling both the features and the targets. This scales all targets to a range around \([-1,1]\), which improves accuracy and reduces training time. Then, ML* is tuned and trained on this data set. After this model is established, it is used in a genetic algorithm optimization in an effort to improve the complete production line. For genetic algorithm the Python package ‘deap’ is used. For GA, a population size of 100 individuals is used, a mutation rate of 0.05, a cross-over rate of 0.6, tournament selection, best 5. For every variable \( j \), the optimization constraints of the genetic algorithm are set 5% below and above \( \mu_j \). This is shown in figure 3.6 in purple. Next, the optimization constraints of the genetic algorithm are increased to 10%, 20% and 30% above and below \( \mu_j \), to see how well ML* predicts examples from a larger space than the training space.
3.5. Experiment III: Modeling Individual Processes

The dataset from experiment II is not available to most companies today, as tracking of every wafer and advanced sensing is not commonly used. For this reason, the third experiment focuses on a more common data set: the VPL is used to simulate ten central composite DoE data sets of every individual process. The steps in this experiment are visualized in fig 3.8. Th This created around 100 examples per process, which is feasible for most factories. All ML algorithms from experiment I are tested on these data sets to compare their performance against polynomial fitting, which is standard in process optimization. The central composite is designed such that for every variable \( j \), observations are taken 30% above and below the default value \( M_j \) denoted in table 3.1. This ensures that the parameter space is not too difficult to model. Then, all ML algorithms are trained on these DoE data sets and they are tested on random data that is generated in the same range, i.e. \([M_j - 0.3 \cdot M_j, M_j + 0.3 \cdot M_j]\). The testing RMSE of each algorithm is then plotted for each algorithm to demonstrate which algorithm performs best.

3.6. Experiment IV: Learning-By-Doing

In experiment IV, all DoE data sets of the previous experiment are combined into a single data set and ML’ is trained on this data set to demonstrate whether it can model this data set. The steps in this experiment are visualized in fig 3.9. Since this data set does not include interactions between processes but only within processes, it is expected that this data set will not give the ML algorithm enough information about the production line. For this reason, the experiment is conducted as follows: after establishing the ML model on the DoE data sets, it is optimized with a GA and the results are then confirmed by doing another experiment and measuring the PCE of the resulting solar cell. This way, we validate whether the algorithm indeed found a better process configuration or whether the ML algorithm provided the GA with poor predictions. Then, the newly found data will be added to the data set, containing essential new information about interactions between processes in the production line. Then, the ML algorithm will be trained again on the new data set and another
GA will be produced. This was done for about 1000 repetitions, to see how many extra measurements one has to do in order to improve the algorithm above a certain point.

Figure 3.9: First, DoE data of every process from experiment III is collected and merged into one data set. Then, an ML algorithm is trained on this data set to establish a predictive model between production line configuration and PCE. Then, this predictive model is used in a GA optimization to predict an improved configuration. Then this new configuration is validated and added to the data set. Repeat until the production line is improved.

**Conclusion**

In this thesis, four research questions are posed that explore whether production lines could be improved using machine learning. For this thesis, a simulation of a production line is developed to simulate a data set of process configurations and their corresponding PCEs. The first research question is which ML algorithm can establish the most accurate predictive model of a production line. The ML algorithms that will be compared are LR, ANN, SVR, RF and AB. These ML algorithms will be tuned, trained and tested on a data set containing random process configuration with a wide range of PCEs. The ML algorithm that can model the production line most accurately is then selected for the subsequent experiments.

Next, the VPL is used to generate virtual solar cells with a narrow distribution of PCE, caused by small random noise in each process. This industrial data set is then used to train an ML algorithm with to establish a predictive model between process inputs and PCE. Then, this model is used in an optimization algorithm to test whether it can improve the production line.

Next, the VPL is used to simulate small DoE data about individual processes. These data sets are then used to train different ML algorithms with to test whether they can improve individual processes. Then, a data set will be composed of the individual DoE data sets to see whether an ML algorithm could establish an accurate predictive model using this limited data.

The tolerance determines the stopping criterion for the algorithm: when the loss does not improve more than $10^{-6}$ in 10 iterations, the algorithm stops.
## 3.6. Experiment IV: Learning-By-Doing

### Manufacturing Process

<table>
<thead>
<tr>
<th>Variable</th>
<th>Allowed Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Wafer making</strong></td>
<td></td>
</tr>
<tr>
<td>Cell thickness [μm]</td>
<td>[160, 180, 200, 250]</td>
</tr>
<tr>
<td>Grade</td>
<td>{CZ, CZ+, FZ}</td>
</tr>
<tr>
<td>Bulk Quality</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>Saw type</td>
<td>{standard, diamond}</td>
</tr>
<tr>
<td><strong>Etching</strong></td>
<td></td>
</tr>
<tr>
<td>Etch temperature [°C]</td>
<td>{0, ..., 50, ..., 90}</td>
</tr>
<tr>
<td>Etch time [min]</td>
<td>{0, ..., 10, ..., 40}</td>
</tr>
<tr>
<td>Previous etch baths [-]</td>
<td>\geq 0</td>
</tr>
<tr>
<td><strong>Texturing</strong></td>
<td></td>
</tr>
<tr>
<td>NaOH conc. [%]</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>Propanol conc. [%]</td>
<td>{1, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Etch temperature [°C]</td>
<td>{50, ..., 70, ..., 100}</td>
</tr>
<tr>
<td>Etch time [min]</td>
<td>{0, ..., 20, ..., 30}</td>
</tr>
<tr>
<td>Exhaust flow rate [-]</td>
<td>{0, 1, 2, 3, 4}</td>
</tr>
<tr>
<td><strong>Cleaning</strong></td>
<td></td>
</tr>
<tr>
<td>HF conc. [%]</td>
<td>{0, ..., 2, ..., 30}</td>
</tr>
<tr>
<td>HCl conc. [%]</td>
<td>{0, ..., 2, ..., 30}</td>
</tr>
<tr>
<td>Clean time [min]</td>
<td>{0, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Rinse time [min]</td>
<td>{0, ..., 5, ..., 10}</td>
</tr>
<tr>
<td><strong>Diffusion</strong></td>
<td></td>
</tr>
<tr>
<td>Phosphorus conc. [%]</td>
<td>{10, ..., 100}</td>
</tr>
<tr>
<td>Belt speed [cm/s]</td>
<td>{0.1, ..., 0.5, ..., 1}</td>
</tr>
<tr>
<td>Drying temp. [°C]</td>
<td>{100, ..., 200, ..., 300}</td>
</tr>
<tr>
<td>Zone 2 temp. [°C]</td>
<td>{700, ..., 980, ..., 1000}</td>
</tr>
<tr>
<td><strong>PSG removal</strong></td>
<td></td>
</tr>
<tr>
<td>HF conc. [%]</td>
<td>{0, ..., 2, ..., 30}</td>
</tr>
<tr>
<td>HF etch time [min]</td>
<td>{0, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Rinse time [min]</td>
<td>{0, ..., 5, ..., 10}</td>
</tr>
<tr>
<td><strong>Edge isolation</strong></td>
<td></td>
</tr>
<tr>
<td>Power [W]</td>
<td>{150, ..., 700, ..., 1500}</td>
</tr>
<tr>
<td>Time [min]</td>
<td>{2, ..., 10, ..., 20}</td>
</tr>
<tr>
<td><strong>Surface passivation</strong></td>
<td></td>
</tr>
<tr>
<td>Deposition time</td>
<td>{1, ..., 15, ..., 30}</td>
</tr>
<tr>
<td>Deposition temp. [°C]</td>
<td>{0, ..., 150, ..., 400}</td>
</tr>
<tr>
<td>Silane gas flow [-]</td>
<td>{1, 2, 3, 4, 5}</td>
</tr>
<tr>
<td>Ammonia gas flow [-]</td>
<td>{1, 2, 3, 4, 5}</td>
</tr>
<tr>
<td><strong>Rear screen print</strong></td>
<td></td>
</tr>
<tr>
<td>Mesh density [strands/cm]</td>
<td>{50, ..., 100, ..., 250}</td>
</tr>
<tr>
<td>Emulsion above mesh [μm]</td>
<td>{0, ..., 15, ..., 30}</td>
</tr>
<tr>
<td>Emulsion below mesh [μm]</td>
<td>{0, ..., 15, ..., 30}</td>
</tr>
<tr>
<td>Strand diameter [μm]</td>
<td>{20, 30, 50, 80, 120}</td>
</tr>
<tr>
<td>Squeegee pressure [-]</td>
<td>{1, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Paste viscosity [-]</td>
<td>{1, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Print speed [-]</td>
<td>{1, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Finger width [μm]</td>
<td>{50, 100, 150, 200, 250}</td>
</tr>
<tr>
<td>Finger pitch [μm]</td>
<td>{1, 2, 3, 4, 5, 6}</td>
</tr>
<tr>
<td><strong>Front screen print</strong></td>
<td></td>
</tr>
<tr>
<td>Mesh density [strands/cm]</td>
<td>{50, ..., 100, ..., 250}</td>
</tr>
<tr>
<td>Emulsion above mesh [μm]</td>
<td>{0, ..., 15, ..., 30}</td>
</tr>
<tr>
<td>Emulsion below mesh [μm]</td>
<td>{0, ..., 15, ..., 30}</td>
</tr>
<tr>
<td>Strand diameter [μm]</td>
<td>{20, 30, 50, 80, 120}</td>
</tr>
<tr>
<td>Squeegee pressure [-]</td>
<td>{1, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Paste viscosity [-]</td>
<td>{1, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Print speed [-]</td>
<td>{1, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Finger width [μm]</td>
<td>{50, 100, 150, 200, 250}</td>
</tr>
<tr>
<td>Finger pitch [μm]</td>
<td>{1, 2, 3, 4, 5, 6}</td>
</tr>
<tr>
<td><strong>Co-firing</strong></td>
<td></td>
</tr>
<tr>
<td>Oxygen conc. [%]</td>
<td>{0, ..., 21, ..., 100}</td>
</tr>
<tr>
<td>Nitrogen gas flow [l/s]</td>
<td>{0, ..., 10, ..., 20}</td>
</tr>
<tr>
<td>Belt speed [cm/s]</td>
<td>{1, ..., 5, ..., 10}</td>
</tr>
<tr>
<td>Drying temp. [°C]</td>
<td>{100, ..., 250, ..., 300}</td>
</tr>
<tr>
<td>Zone 2 temp. [°C]</td>
<td>{700, ..., 830, ..., 1000}</td>
</tr>
</tbody>
</table>

Table 3.1: Overview of the variable inputs for each manufacturing process. The bold values are used in the standard recipe.
Simulation Results

In this chapter, the comparison between machine learning (ML) algorithms for modeling PV manufacturing processes is presented. First, the ML algorithms are trained on large data sets to analyze their capacity to model a complete PV manufacturing line. The best performing algorithm in this first experiment is then trained on a examples with a narrower distribution of cell efficiencies. Once this ML model is established, it will be used as the objective function in a genetic algorithm optimization to maximize the production line efficiency. Finally, all algorithms will be trained on small design of experiment (DoE) data sets of individual processes.

4.1. Machine Learning on Large and Balanced Data

First, five ML algorithms are modeled on input-output data of a complete production line. The distribution of performance metrics is shown in fig 4.1. Before modeling the data, the ML algorithms are tuned to optimize their hyperparameters using a data set of 30,000 randomly chosen examples from the data set that corresponds to fig 4.1. For each algorithm, its hyperparameters are optimized using five-fold cross-validation (CV). Validation curves are plotted for ANN to demonstrate how analysis was performed, but these graphs are omitted for the other algorithms as their analysis were identical to ANN.

For artificial neural networks, the hyperparameters that are optimized are shown in table 4.1. Validation curves of this optimization are shown in fig 4.2. The bold values in table 4.1 indicate which hyperparameters were selected for the comparison with other ML algorithms. According to the principle of Ockham’s razor [66], two hidden layers and regularization $10^{-3}$ are selected for the ANN as it creates the simplest model without sacrificing accuracy.

For support vector regression, the hyperparameters that are optimized in a grid search are shown in table 4.2. The addition of polynomial features did not improve the testing loss while sharply increasing the training time. The SVR has minimal testing loss for the RBF kernel. This was expected, since the RBF kernel gives the
Figure 4.1: Performance metrics of a data set that was produced by simulating random production line configurations and modeling their (a) photovoltaic cell efficiency, (b) fill factor, (c) open-circuit voltage and (d) short-circuit current.

Figure 4.2: Validation curves for six hyperparameters for ANN. Plotted are the mean training and testing loss and plus/minus one standard deviation of a 5-fold CV. (a) Adding quadratic features does not improve the performance of the ANN which is highlighted by an increase in both testing and training loss; (b) a network architecture with two, three or four hidden layers achieved the lowest testing loss. One hidden layer seems to underfit while five hidden layers starts to overfit; (c) there is a slight performance increase when using the rectified linear activation function; (d) the Adam solver performs best, as was expected since it is generally preferred for sets with greater than 1000 examples [50]; (e) some strange behavior for the regularization term: the validation curve does not have the expected convex shape and suddenly drops for $10^{-7}$; (f) a minimum testing loss for a low tolerance of $10^{-6}$. This low tolerance for convergence forces the algorithm to train extensively on the data and thus indicates that the algorithm does not overfit the data.
4.1. Machine Learning on Large and Balanced Data

Hyperparameter | Values (default value, best value)
--- | ---
Polynomial features | 1, 2
Hidden layer structure | (100), (100,100), (100,100,100), (100,100,100,100)
Activation function | identity, logistic, tanh, relu
Solver | LBFGS, SGD, Adam
Alpha | 1e-6, 1e-5, 1e-4, \textbf{1e-3}, 1e-2, 1e-1
Tolerance | 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1

Table 4.1: Results of a five-fold CV gridsearch of ANN on 30,000 data points.

SVR the largest capacity [47, 50]. Regularization parameter C strongly influences the performance of SVR and minimizes the testing loss for C=10. Above this value SVR overfits while below this value SVR underfits. The choice of epsilon does not affect the testing RMSE, although for $\epsilon > 1$ predictions became significantly worse. The default value in scikit-learn of $\epsilon = 0.1$ is selected for the comparison. The computation time of SVR grew with an incredible rate with increasing number of training examples. As the SVR training time is slow and its performance is not great, SVR seems impractical and not appropriate for modeling the production line.

Hyperparameter | Values (default, best)
--- | ---
Polynomial features | 1, 2
Kernel | linear, poly, \textbf{RBF}, sigmoid
C | 1e-2, 1e-1, 1, 10, 100
Epsilon | 0.001, 0.01, 0.1, 0.99

Table 4.2: Results of a gridsearch of ANN

For decision tree, performs best for a tree structure with 16 nodes, but beyond that the algorithm overfits. The testing loss is minimal when the DT uses all features for building the decision tree. This indicates that the hypothesis does not overfit, since the hypothesis performs better for a larger number of features.

Hyperparameter | Values (default, best)
--- | ---
Polynomial features | 1, 2
Max depth | None, 2, 4, 8, \textbf{16}, 32, 64
Min samples per split | 2, 8, 16, 32, \textbf{64}, 128, 256
Min samples per leaf | 1, 2, 4, 8, 16, \textbf{32}, 64
Max features | \textbf{m}, sqrt(m), log2(m)

Table 4.3: Results of a five-fold CV gridsearch of DT on 30,000 data points.

For random forests, the hyperparameters that are optimized in a grid search are shown in table 4.4. The hypothesis performs best for about 30 estimators, which is the optimal value between overfitting and underfitting. The maximum depth of the estimator trees is 'None', which means that the decision trees decide their own length. This indicates that the RF is not overfitting since it enables RF to create strong estimators. The testing loss is minimal when the RF hypothesis uses all features for
building its estimators.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Values (default, best)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial features</td>
<td>1, 2</td>
</tr>
<tr>
<td>Number of estimators</td>
<td>10, 20, 30, 40, 50</td>
</tr>
<tr>
<td>Max depth</td>
<td>None, 2, 3, 4, 5</td>
</tr>
<tr>
<td>Min samples per split</td>
<td>1, 2, 4, 8, 16</td>
</tr>
<tr>
<td>Min samples per leaf</td>
<td>1, 2, 4, 8, 16</td>
</tr>
<tr>
<td>Max features</td>
<td>m, sqrt(m), log2(m)</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>True, False</td>
</tr>
</tbody>
</table>

Table 4.4: Results of a five-fold CV gridsearch of RF on 30,000 data points, where ‘m’ is the number of features of the training data.

For adaptive boosting, the hyperparameters that are optimized in a grid search are shown in table 4.5. For AB the same conclusions are drawn as for RF: the hyperparameters that minimize this algorithm’s testing loss all point towards high bias and as all options for increasing capacity are exhausted, this algorithm seems unfit for modeling the complete manufacturing line.

<table>
<thead>
<tr>
<th>Hyperparameter</th>
<th>Values (default, best)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial features</td>
<td>1, 2</td>
</tr>
<tr>
<td>Loss</td>
<td>linear, square, exponential</td>
</tr>
<tr>
<td>Number of estimators</td>
<td>50, 100, 150, 200, 250</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1</td>
</tr>
</tbody>
</table>

Table 4.5: Results of a five-fold CV gridsearch of AB on 30,000 data points.

After these results were collected, all algorithms were fitted with their optimal hyperparameters. Then, a learning curve for these algorithms was made by increasing number of training sets ranging from 10 to 80,000 examples with five-fold CV. The results are shown in fig 4.3. There are many interesting observations to be made. Until 5,000 examples, all algorithms perform equally, and the ANN performs worst. This makes sense, since hyperparameters were tuned for a data set of 30,000 examples. With a training set greater than 5,000 examples the ANN scores the lowest test RMSE. The learning curve of the ANN demonstrates that it suffers from high variance for small set sizes, as the algorithm improves drastically with training set size. Around 80,000 examples the ANN scores an RMSE that is half of the second-best algorithm AB.

Moreover, while the other algorithms flatten out with increasing training examples, ANN shows the least signs of slowing down and judging from the testing loss trend, the algorithm seems capable of decreasing its RMSE even further on a larger data set. Fig 4.3 shows that most other algorithms suffer from high bias errors. This demonstrates that the limit of their capacity is reached.

The plotted standard deviation demonstrates the consistency of most algorithms, however ANN shows some interesting behavior between 700 and 9000 training examples: its standard deviation increases with more training examples. The reason
4.1. Machine Learning on Large and Balanced Data

Figure 4.3: The dots represent the mean loss of five-fold CV and the colored areas represent plus/minus the standard deviation.

for this is thought to be as follows: the ANN has hyperparameters that are optimized for a data set of 30,000 examples, so it is expected that the algorithm performs bad for small data sets. In the case of very small data sets (10-1000), the ANN minimizes its training loss by making a completely biased estimator, which scores minimal error for small data sets. For training set size greater than 1000 examples, the algorithm has enough examples to train its weights and make a better estimator and reduce its bias. However, at this stage, the algorithm is still strongly affected by the randomness of the data, as five-fold CV splits the data up in random sets. When the training set is greater than 10,000 examples, most of this randomness will be balanced, as is confirmed by a diminishing standard deviation in fig 4.3.

Finally, the performance of POLY is worst of all regression models that were compared. This was expected, since linear regression has the lowest capacity of all ML algorithms in this comparison. This problem has too many variables which quickly lead to overfitting. Due to the low capacity of POLY, the hypothesis has high bias and its testing loss therefore does not improve much for greater set sizes.

Based on these results, ANN seems the most appropriate algorithm for modeling a complete PV production line. To investigate the capacity of the ANN even further, a learning curve with both training and testing loss was produced to highlight bias and variance problems of this ANN hypothesis. The learning curve of testing loss flattens out after 300k examples and coincides with the training loss, which indicates that the limit of capacity of this hypothesis is reached and there is no variance error.

Even though the ANN validation curve shows that the ANN starts to flatten out, the capacity of this ANN hypothesis is not exhausted yet. The ANN could be improved by adding more layers and decreasing regularization further. However, these improvements are beyond the scope of this research: since these type of data sets are purely hypothetical, this endeavour is not further pursued.

Fig 4.4 shows the predictions of an ANN model trained on 300k examples against the actual values. The result of this ANN ML algorithm is impressive, however it should be noted that this ML algorithm is trained on a very large number of examples with
4. Simulation Results

Figure 4.4: Scatterplot of predictions of the ANN of 20,000 examples that the model did not see before.

large variety of examples to learn from. In practice, production lines only have two types of data available: small DoE sets of individual processes with broad variance and large production data with narrow variance. To further investigate the ML algorithms for PV manufacturing, we should therefore investigate how ML can model the production line using these data set sizes.

4.2. Experiment II: Production Line Improvement Using Production Data

To test how ANN models industrial data, a large data set of narrowly distributed production data was created, that was shown in fig 4.5. The VPL in this case simulates a VPL with a mean PCE of 17.86 ± 0.31% and a maximum PCE of 18.74%.

First, the ANN hyperparameters are tuned to have optimal hyperparameters for this problem. The validation curves of this hyperparameter optimization are shown in fig 4.6. Just as in experiment I, adding quadratic polynomial features does not improve the testing loss of the ANN. There seems to be a small improvement for using two hidden layers and the tanh activation seems to have a slight improvement. Again, the Adam solver performs best and a regularization of $10^{-3}$ performs best. The algorithm minimizes its testing loss for a tolerance of $10^{-5}$. These results are very similar to experiment I.

One of the objectives for this thesis is to optimize a PV manufacturing line using production line data. Using this ANN predictive model as an objective function, a GA was used to optimize the production line. The results shown in fig 4.7 demonstrate that it is indeed possible to improve a manufacturing line by modeling an ML algorithm on production data. For 20 repeated tries for every search space, boxplots of the results are shown. For small search space, the ANN makes very accurate predictions for the GA optimization and is able to improve the production line in a constant way. This could have large impacts on they way production lines can be improved in the future. For a DoE-based approach, production needs to be halted, but this method could be used on-the-go, without having to stop industrial operation. Furthermore, fig 4.7 demonstrates that for a larger optimization search space,
4.2. Experiment II: Production Line Improvement Using Production Data

Figure 4.5: Performance metrics of a data set that was produced by simulating random production line configurations and modeling their (a) photovoltaic cell efficiency, (b) fill factor, (c) open-circuit voltage and (d) short-circuit current.

Figure 4.6: Validation curves of the hyperparameter gridsearch for ANN for industrial type of data. For most hyperparameters, the same results were similar to experiment I.
better process configurations can be found. Fig 4.7 shows that the ANN makes has good extrapolation properties. So by training on a narrowly distributed data set the ANN allows to predict quite far beyond what was trained on. However, for increased search space, the ANN accuracy decreases and for ±30% a tipping point is crossed: the predictions deteriorate. This makes sense, as extrapolating too far out of the training space.

The second experiment demonstrates that ANN can be used to model a complete production line by training on narrowly distributed production data. Furthermore, it is demonstrated that a GA can then optimize the production line using this predictive ANN model. When the GA is allowed to search in a larger space, it will find significantly higher cell efficiencies. However, when allowed to look too far, the predictions of the ANN and thus the GA optimization deteriorates.

![Figure 4.7: Comparison between genetic algorithm optimization results.](image)

Figure 4.7: Comparison between genetic algorithm optimization results. The GA optimization was performed on ±5%, ±10%, ±20% and ±30% around the nominal value of each process variable. For each search space, 20 GA optimizations were performed. Their resulting process configuration was confirmed with VPL and PC1D. The boxplots shows the spread of each GA optimization and the confirmed values.

4.3. Experiment III: ML on Design of Experiment Data

ML is infamous for being data hungry [10, 50], but most ML algorithms can be adapted to small data sets by changing their hyperparameters. In the third experiment, individual processes are sampled with central composite design and ML are trained on this data. The ML algorithms will be compared against a polynomial fitting, which is the standard modeling for response surface methodology.

First, the hyperparameters are optimized with ten-fold CV for DoE data. Ten-fold CV, which uses 90% of the data for training, was used because with these small data sets we want to use as much training data as possible. The validation curves for ANN and the silver front metallization process are shown in fig 4.8.

Fig 4.8 shows some drastic changes compared to fig 4.2 and 4.6, as the data from
single processes instead of a complete manufacturing line. The standard deviation lines suggest that negative losses were reached but this is not the case: these ± standard deviations are from a non-symmetrical distribution. According to the principle of Ockham’s razor \[66\], we select the model with one hidden layer, regularization of $10^{-1}$ and a tolerance of $10^{-4}$ as this creates the simplest model.

The hyperparameters of every ML algorithm are tuned for each process. Then, these

ML algorithms are trained on DoE data of each process and tested on random data where every variable takes random values in the same range as the DoE. Fig 4.9 shows the result of this comparison. In all cases, ANN performs best, but in three out of five cases it performs similar to POLY. POLY shows very variable performance, as it performs similar to ANN in three out of five processes but other times scores poorly, like in the silver front metallization process. This is a surprising result, since POLY is the standard way to optimize PV manufacturing processes. Furthermore, since these processes contain much less complex interactions than the complete production line, it was expected that the simpler polynomial regression would have a more suitable capacity than ANN.

Experiment III shows that even in smaller data sets, ANN is the preferred modeling approach to model processes. This is a very useful result, as it impacts not only factories but also laboratories. Researchers optimizing their experiment with RSM could use ANN for improved model building.
4. Simulation Results

Figure 4.9: Comparison of ML algorithms on DoE data of individual processes. In three out of five processes, ANN performs equal to POLY. However, in the texturing and Ag metallization process, the performance of POLY deteriorates.

4.4. Experiment IV: Production Line Improvement using DoE Data

Then, all DoE data sets of each individual process are combined in a large data set and ANN is trained on these examples. This predictive ANN model is then used in a GA optimization and the results are confirmed with the VPL. This is repeated for a 1000 times, and the improvement of the production line is tracked. Fig 4.10 shows the result of this experiment. These results are promising: fig 4.10(a) shows that after only 6 confirmation steps, the data set contains enough information about the interactions between process variables to improve its predictions. After 6 repetitions, this method improved the production line from 16.9% PCE to 18.6% PCE. 19.7% PCE was reached within 74 repetitions and 20.2% PCE was reached within 483 repetitions. Fig 4.10(b) shows that increasing the GA search space accelerates the production line improvement. A larger search space leads to improvement from 16.9% to 19.2% in 13 repetitions and 20.2% was reached within 76 repetitions. This result is very exciting for existing and starting manufacturing lines. The ANN was modeled on about 1,200 DoE data points from the complete factory, and for less than 100 extra experiments, the production line PCE could be improved from 16.9% to over 20%.

4.5. Discussion

The results achieved in this thesis indicate a promising role for ML in PV manufacturing. According to the international energy agency, in 2017 almost 100 GWp of PV installations were added to the globally installed PV capacity [67] and in 2018, the barrier of 100 GWp was crossed. With these quantities, even a tiny improvement of 0.01% becomes interesting.

Experiment I demonstrated that ANN is the most suitable ML algorithm for modelling a PV production line. Most ML algorithms tested in this thesis lacked the capacity to model the VPL, so they will probably also lack the capacity to model a real manufacturing line. For this reason, we can expect ANN to still be most appropriate for
modeling in PV manufacturing. The capacity of the algorithm could be extended by adding extra layers or tuning the hyperparameters. Moreover, in this thesis, the most basic form of ANN was used, which could be greatly improved by including more advanced deep learning algorithms [10]. This provides an interesting future direction for this project.

Since VPL is only a simulation, some results obtained in this thesis cannot be directly extrapolated to a real production line. Experiment II demonstrated that RSM could be avoided by modeling the production data with ANN. This could save a lot of time and money, since the factory could be improved while operating. Not every factory has this level of tracking and sensing, but since most industries are moving into ‘smart manufacturing’ [68], this assumption is not unrealistic. This experiment also has another implication: factories could improve themselves ‘on-the-go’ by applying this ML strategy. Moreover, using ML-based methods, companies do not have to rely on expert knowledge, since they can rely on the objectiveness of ML. An algorithm will never quit, never forget and only improve over time.

Most practical are the results of experiment III, which demonstrate that ANN is also very suitable for modeling individual processes using DoE data. This result is not only promising for factories, but also for laboratories, where DoE is commonly used. The result from this thesis suggests that a simple ANN could compete with polynomial fitting. An interesting future endeavor would be to compare an ANN based improvement method with professional statistical packages and see how well they perform.

The optimization method from experiment IV is most accessible to factories today. This DoE based approach does not need a fully tracked and sensored production line, and does not need many extra experiments. The combination between GA and ANN is a powerful method to automatically obtain the missing experiments of some production line. By allowing the GA to make extreme predictions, the weak points of the ANN model are quickly tracked down and added to the knowledge of the algorithm. Moreover, this result motivates to explore the field of reinforcement learning, which is a category of ML that is separate from supervised ML. Reinforcement learning is
learning-by-doing, in which some agent takes actions and is rewarded based on the outcome. Reinforcement learning could be a promising direction for factories to go, for instance where all production variables are controlled by an agent. As the results of experiment IV point out, it does not take many tries to improve the production line.

**Conclusion**

Four experiments were done to explore machine learning-based optimization of production lines. In the first experiment, several ML algorithms are tested on data from the VPL, with ANN scoring significantly better than the other algorithms. The ANN was then trained on a large data set of 300,000 examples, which demonstrated that ANN can accurately model the VPL given enough training examples. This result answers the first research question of this thesis.

In the second experiment, a narrow distribution of PCE is simulated with the VPL by allowing small random noise on every process variable. This created a data set of 100,000 examples with a mean PCE of 17.86 ± 0.31% and a maximum PCE of 18.74%. An ANN was hyperparameter-tuned and trained using this data set to establish a predictive model. The ANN model was then used in a GA optimizer, which was able to maximize the production line efficiency to greater than 19%. Furthermore, the production line could be improved even further when the GA optimized in a larger search space. However, the accuracy of the ANN declined as the parameter search space increased, and at some point, even the ANN prediction deteriorated. This result answers the second research question.

In the third experiment, several ML algorithms were tuned and trained on DoE data from individual processes. The results showed that in all cases, ANN is the best regression model. This is a surprising result, since ANN typically needs large training data sets. This experiment demonstrated the versatility of the ANN: with the right hyperparameters, the algorithm can model both large and small data sets. This result answers the third research question.

The fourth experiment showed that a combination between ANN and GA can improve the VPL when allowed to try a few extra experiments. With each try, the process configuration is confirmed and added to the data set. After 13 tries, an optimization from 16.9% to 19.2% could be realized and 20.2% was reached within 76 repetitions. This result answers the fourth research question.
Production line improvement is an expensive and time-consuming endeavor in photovoltaics (PV) manufacturing. In this thesis, novel methods for production line optimization are tested that could improve conventional frameworks. These innovative methods are based on machine learning (ML) algorithms, which can effectively model complex relationships with many variables. For testing these ML algorithms, a virtual production line was developed to simulate the industrial production of solar cells. The findings of this research show promising possibilities for ML in PV manufacturing. By using artificial neural networks (ANN) and genetic algorithms (GA), a virtual production line could be improved by using either production line data or DoE data.

The main challenge of this thesis project was to collect production line data because companies are hesitant to share their data. For this reason, the first aim was to develop a simulation of a PV production line to produce data with. The virtual production line (VPL) software was developed to simulate the industrial manufacturing of aluminium-back surface field single-crystal silicon solar cells. The VPL models wafer properties with 50 input variables divided over eleven processes and simulates performance metrics using PC1D. The VPL software that was developed for producing the results in this thesis could be very useful for testing future production line improvement ideas.

The second aim of this research was to compare the capacity of ML algorithms to model a PV manufacturing line. The contenders in this comparison were linear regression, artificial neural networks (ANN), support vector regression, random forests and adaptive boosting. The VPL was used to simulate random process configurations with photovoltaic cell efficiency (PCE) between 0-20%. The ML algorithms were trained on this data set to predict the PCE of the produced wafers based on 50 process inputs used to fabricate it. ANN outperformed the other ML contenders by a large margin and was able to predict PCE with a root mean squared error of 0.08. The other algorithms in this comparison lacked the capacity to model the production line and their performance could not be improved by training on a larger data set.

The third aim of this research was to develop a ML-based method for improving solar cell production lines. For this, two alternative methods were developed: method 1
uses production line data and method 2 uses design of experiment data of the individual processes in the production line. To test method 1, the VPL was used to simulate a data set with mean PCE of $17.86 \pm 0.31\%$ and a maximum PCE of $18.74\%$. An ANN was hyperparameter-tuned and trained on this data set and used in GA optimization, which increased the PCE of the production line to greater than $19\%$. Furthermore, when the GA optimized on a larger search space, the production line could be improved even further, utilizing the extrapolation properties of the ANN model.

To test method 2, the VPL was used to simulate design of experiment (DoE) data of each individual process according to central composite design. Then, these DoE data sets were combined into one data set and used for hyperparameter-tuning and training an ANN to model all processes at once. Once this ANN model was established, it was used in GA to improve the production line. The result of this optimization was then confirmed with the VPL and it was added to the training data. After 6 repetitions of ANN training and GA optimization, this method improved the production line from $16.9\%$ to $18.6\%$. $19.7\%$ was reached within 74 repetitions and $20.2\%$ was reached within 483 repetitions. The speed and performance of improvement were boosted by increasing the GA search space. A larger search space lead to improvement from $16.9\%$ to $19.2\%$ in 13 repetitions and $20.2\%$ was reached within 76 repetitions.

The fourth aim of this research was to test different ML algorithms on small DoE data sets of individual processes. Several ML algorithms were tuned and trained on these data sets and their performance was assessed with root mean squared error. Surprisingly, ANN performed best in this comparison, outperforming the conventional polynomial fitting. While polynomial fitting and ANN performed equally in three out of five tests, in two processes ANN performed significantly better.

These novel improvement methods enable factories to produce better solar cells with the same production line. It was demonstrated in simulation that ML could improve the conventional response surface methodology for process optimization. This can drastically reduce the costs of producing solar cells and the transition to renewable energy can be accelerated.

A limitation of this research is the fact that these results were obtained in simulation and they cannot be extrapolated yet to real factories. Moreover, the process improvement methods that are suggested in this thesis need either a fully tracked and sensored production line or around 1,000 DoE experiments.

Since not every production line is fully tracked and sensored, the DoE-based approach is probably the most promising method of this thesis. Future directions of this project could focus on reducing the amount of DoE data needed to train the ML algorithms. Moreover, tests with real data would solidify the evidence found in this thesis. The DoE-based approach encourages to further explore reinforcement learning, a field in ML which has seen some recent breakthroughs. Reinforcement learning leads agents to perform some task by rewarding them based on their performance. This agent could be in control of all variables of a production line and be rewarded for making high efficiency solar cells. As the experiments in this thesis shows, it does not take many tries to improve the process for an artificial intelligence.
List of Abbreviations
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>Adaptive Boosting</td>
</tr>
<tr>
<td>ARC</td>
<td>Anti Reflective Coating</td>
</tr>
<tr>
<td>AI-BSF</td>
<td>Aluminium-Back Surface Field</td>
</tr>
<tr>
<td>ANN</td>
<td>Artificial Neural Networks</td>
</tr>
<tr>
<td>B</td>
<td>Boron</td>
</tr>
<tr>
<td>BBD</td>
<td>Box-Behnken Design</td>
</tr>
<tr>
<td>CCD</td>
<td>Central-Composite Design</td>
</tr>
<tr>
<td>CV</td>
<td>Cross-Validation</td>
</tr>
<tr>
<td>DoE</td>
<td>Design of Experiment</td>
</tr>
<tr>
<td>DT</td>
<td>Decision Tree</td>
</tr>
<tr>
<td>full FD</td>
<td>Full Factorial Design</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>LBFGS</td>
<td>Limited-memory Broyden–Fletcher–Goldfarb–Shanno</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Squared Error</td>
</tr>
<tr>
<td>P</td>
<td>Phosphorus</td>
</tr>
<tr>
<td>PCE</td>
<td>Photovoltaic Cell Efficiency</td>
</tr>
<tr>
<td>POLY</td>
<td>Linear Regression with polynomial features</td>
</tr>
<tr>
<td>PV</td>
<td>Photovoltaics</td>
</tr>
<tr>
<td>ReLU</td>
<td>Rectified Linear Unit</td>
</tr>
<tr>
<td>RF</td>
<td>Random Forests</td>
</tr>
<tr>
<td>RSM</td>
<td>Response Surface Methodology</td>
</tr>
<tr>
<td>RSME</td>
<td>Root Mean Squared Error</td>
</tr>
<tr>
<td>Si</td>
<td>Silicon</td>
</tr>
<tr>
<td>SGD</td>
<td>Stochastic Gradient Descent</td>
</tr>
<tr>
<td>SVR</td>
<td>Support Vector Regression</td>
</tr>
<tr>
<td>VPL</td>
<td>Virtual Production Line</td>
</tr>
</tbody>
</table>

Table A.1: List of abbreviations used in this thesis.
I would like to share a simple way to operate PC1D on Windows from a Python script. The first section explains where to download the files you need, the second section explains how to run PC1D from the command prompt, the third section explains how to create PC1D-readable files and the last section contains the code to run PC1D in Python. Note, that this is a tutorial for Windows only.

### B.1. Downloading PC1D

For this tutorial you need the command version of PC1D, which can be downloaded from PVlighthouse [link 1] by clicking on **PC1Dmod and cmd PC1D v6.2.2.zip**. In this zip-file, you find two versions of cmd-PC1D: **cmd-pc1d5.exe** and **cmd-PC1D6-2.exe**. PC1Dmod6-2 is an updated version of PC1D5 with more realistic solar cell simulations. One limitation of version 6.2 is that it does not allow batch mode operation. However, this will not form any limitation for most users. In general, **cmd-PC1D6-2.exe** is the recommended version of PC1D.

### B.2. PC1D from command prompt

In order to run PC1D from the command prompt, open **cmd** and navigate to the folder where you saved the PC1D folder. Then, simply type:

```plaintext
cmd-PC1D6-2.exe filename.prm
```

The .prm-file stores all information about a PC1D solar cell. Writing this line in **cmd** is equal to loading a parameter file into PC1D and running it. In the PVlighthouse zip-file, there are already some .prm-files available. When you saved your folder in C:\Users\username\PC1Dmod, you can run PVcell_simple.prm as follows. The result is shown in the appendix.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\username\PC1Dmod>cmd-PC1D6-2.exe PVcell_simple.prm
```
B.3. Creating . prm-files

Of course, you want to test your own solar cell designs. In order to change specific parameters of the solar cell, the . prm-file first needs to be decoded. The developer of cmd-PC1D, Halvard Haug, was so kind to create a tool for converting . prm-files to ascii. This program can also be found in the zip-folder under the name convert prm_to_ascii.exe. In order to execute this program, we go back to cmd and type:

```bash
convert_prm_to_ascii.exe filename.prm filename.txt
```

In our command prompt this looks as follows. If there is no file called example.txt in the working directory, it will be created upon execution of the program.

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

`~\PC1Dmod>convert_prm_to_ascii.exe PVcell_simple.prm example.txt`

Understanding the . prm format

When opening text file example.txt, we get a deeper insight into PC1D. After conversion of a .prm-file to ascii, we can open the text in Notepad and read it. The format might seem somewhat cryptic at first sight, but once your eyes are adjusted we discern different PC1D related parameters. For instance, the following lines are copied from the textfile and belong to the front reflection window in PC1D. When both bCoated and bExternal are 0, the Reflectance “Fixed” option is selected with a value of 0.055. This part of the .prm file is linked to the reflectance window in PC1D, shown in figure B.1.

```plaintext
CDevice::m_FrontRfl:
CReflectance::m_bCoated=0
CReflectance::m_bExternal=1
CReflectance::m_Filename=example reflectance.ref
CReflectance::m_Fixed=0.055
```

As you can see, even the location of the external reflection file is specified. I suggest that you write the whole path of the file, otherwise PC1D might get confused where the file is located. So write:

```plaintext
CReflectance::m_Filename= ~\PC1Dmod\example reflectance.ref
```

The same is true for other files that PC1D uses, like the mat-file, the abs-file, the inr-file, et cetera. If your program does not work, this is the first place I would look. In this text file, we can change every PC1D parameter. I encourage you to read through the text file yourself to find the location of every parameter.

After you have changed your desired hyperparameters, you can save the text file

Now you might wonder why I explained this cumbersome method of running PC1D. This method is very useful because it enables us to operate PC1D from a program script. There are multiple ways of achieving this, but the meta code looks like:

1. Calculate your desired PC1D-parameters from a script
2. Write a text file according to the PC1D format
3. Convert the text file to a .prm file
4. Run cmd-PC1D with this .prm file
5. Collect the output from cmd

There are different ways to achieve step 2. I wrote a Python dictionary with a key for every section. After all values are included into the text strings I join them and write them into a text file, see the code in the Appendix. If you have trouble achieving this, do not hesitate to contact me! To operate step 3, 4 and 5 from a Python script, please see the code in the Appendix. I want to thank Mattias Klaus Juhl for helping me with this part. I learned a lot from his Python wrapper for PC1D (Link 2). This is basically a stripped-down version of that. Also I would like to thank Yoann and Harsh for testing.

B.5. Links
2. https://github.com/MK8J/PC1D_wrapper

B.6. Appendix
B.6.1. Execute output
The output of running PV_cell_simple.prm:
<table>
<thead>
<tr>
<th>Base Voltage</th>
<th>Base Current</th>
<th>Base Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.011271</td>
<td>-0.0394364</td>
<td>0.00044515</td>
</tr>
<tr>
<td>-0.011271</td>
<td>-0.0394364</td>
<td>0.00044515</td>
</tr>
<tr>
<td>0.0047278</td>
<td>-0.0394364</td>
<td>-0.000186449</td>
</tr>
<tr>
<td>0.0207274</td>
<td>-0.0394364</td>
<td>-0.000817412</td>
</tr>
</tbody>
</table>
B.6.2. Python code

```python
import os
from io import BytesIO
import subprocess

# Join path of current file with the folder “files” containing the
# executable programs
dir_path = os.path.join(os.path.dirname(os.getcwd()), "files")

# Assign names to the executable programs that you need
PC1D_exe = os.path.join(dir_path, "cmd-PC1D6-2.exe")
asc2prm = os.path.join(dir_path, "convert_ascii_to_prm.exe")

# Create the PC1D text file from a dictionary
ascii = "".join("{0}".format(v) for v in str_dic.values())

# Choose a folder where you want to save the text file.
# IMPORTANT! Save the file locally and not on a dropbox/ onedrive
# folder since this will crash upon syncing.
textfile = "~\Desktop\dataSim\ascii.txt"

# Choose a folder where you want to save the prm file
prmfile = "~\Desktop\dataSim\out.prm"

# Write the text file
with open(textfile, "w") as f:
    f.write(ascii)

# Execute conversion file
txt2prm = subprocess.run([asc2prm, textfile, prmfile],
                          stdout=subprocess.PIPE)

# Execute cmd-PC1D using the new .prm-file
pc1d = subprocess.Popen([PC1D_exe, prmfile],
                        stdout=subprocess.PIPE)

# Take the output of the of the command prompt and turn it into a
# Python-readable file
(output, err) = pc1d.communicate()
data_str = BytesIO(output)

# Kill PC1D
pc1d.kill()

# Extract Pmpp, Vmpp, Impm
if data_str is not None:
    try:
        data = np.genfromtxt(data_str, delimiter="\t", names=True)
        index = np.argmax(-data["Base_Power"])  
        Pmpp = -data["Base_Power"][index]
        Vmpp = -data["Base_Voltage"][index]
        Impm = -data["Base_Current"][index]
    except:
        print(BytesIO(output).read())
        data = None
        Pmpp = np.nan
        Vmpp = np.nan
        Impm = np.nan
```

---

B.6. Appendix


[67] Gaëtan Masson, Izumi Kaizuka, Johan Lindahl, Arnulf Jaeger-Waldau, Gregory Neubourg, Peter Ahm, José Donoso, and Francesca Tilli. A snapshot of global pv markets-the latest survey results on pv markets and policies from the iea pvps