Transfer learning in TCAD enabled Machine Learning

Abdul Hanan Khan (122301), abdul.hanan.khan@uni-weimar.de

July 2023

I dedicate this thesis to Ami, Hania & Minahil.

Declaration of authorship

I, Abdul Hanan Khan, hereby declare that I am the sole author of this thesis. To the best of my knowledge, this thesis does not contain any material previously published by any other author except where due acknowledgment has been made. The work and the corresponding results presented in a confidential manner in this thesis have been conducted in the work environment of GlobalFoundries.

This is the true copy of the thesis, including any final revisions implemented. This thesis has not been presented previously to any other examination board nor has it been published.

Date:

Signature:

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This introductory chapter will cover briefly what machine learning is, its role in the semiconductor industry, what is TCAD, the problem statement, and the topic of this study.

1.1 Machine learning (ML)

Machine learning is a part of artificial intelligence (AI) which consists of complex algorithms that are developed to identify and understand the relationships and patterns within a set of data. Based on this understanding/training, the ML model is used to make predictions or take actions. ML can be implemented with various training techniques. The one that is utilized in this study is supervised learning.

1.1.1 Supervised learning

In this training approach, The data that is used to train the ML model is labeled, meaning that the input data is paired with the corresponding outcomes. The model trains from this labeled data and recognizes how to map input variables to the corresponding output variables.
It is important to point out that the performance of these ML models is only as good as the data they are trained on irrespective of the implemented training approach.

1.2 Machine learning in semiconductor industry

The semiconductor industry is one of the most influential and critical industries in the world whose impact and importance is increasing by the day, as it provides the key component necessary for the development of a vast range of electronic devices mainly computers, smartphones, automobiles, and medical equipment. ML has the potential to play a major part in this industry in various ways, for example, by optimizing semiconductor design and manufacturing, and improving product performance and quality while reducing costs. Some of the important applications of ML in the semiconductor industry are as follows:

1.2.1 Defect detection in semiconductor

Defect detection is a part of ML's image processing section. This is where ML is critically used in the semiconductor industry. With the help of an excessive amount of data from images of semiconductor wafers, ML algorithms can be used to analyze and detect defects as well as classify them with respect to their impact, with considerably high accuracy. This process can be implemented in an early production stage to improve device performance/yield as well as reduce costs, resulting in an overall high-quality product development, which can then be provided to customers.[Yang Li, 2021]

1.2.2 Predictive Maintenance

ML also plays a critical and impactful part in predicting equipment failures and this role is utilized in multiple fields outside of the semiconductor industry as well. ML models can detect patterns and predict which equipment is likely to fail, by analyzing data from various sensors and additional sources. Based on this prediction before the failure actually takes place, early preventive measurements can be taken to reduce downtime resulting in an overall reduction of costs associated with equipment maintenance and utilization, leading to improvement in product quality.[Dyd Pradeep, 2023]

1.3 Technical Computer-Aided Design (TCAD)

With the passage of time, semiconductor devices are becoming more complex and smaller in size leading to difficulties in device design, manufacturing, and testing. TCAD is a computer simulation that is utilized to model as well as simulate the processing and electrical behavior of semiconductor devices. With its ability to model and optimize device performance, lower design costs, and quicken the design process, TCAD has grown to be a crucial tool in semiconductor design and manufacturing.
With TCAD software, engineers can model a variety of physical phenomena, such as doping profiles, stress effects, and electrical, thermal, and optical behavior. Based on these models, engineers can tune these physical events and can forecast device performance, improving device architectures, which in turn eliminates the need for costly and time-consuming physical testing.
Simulating and optimizing intricate device structures is one of TCAD's main benefits in the semiconductor sector. As mentioned before, physical testing gets more

challenging and expensive as device structures get more intricate and device sizes get smaller. TCAD makes it possible for engineers to precisely model and mimic device behavior.
The capacity to model manufacturing processes is a crucial benefit of TCAD in the semiconductor sector. TCAD technologies enable engineers to optimize the process and raise yield by simulating the full manufacturing process, from material deposition to device creation. Manufacturers can save costs, enhance device performance, and boost production efficiency by optimizing the manufacturing process.
In conclusion, TCAD is a crucial tool for the semiconductor sector as it enables engineers to precisely model and simulate the behavior of semiconductor devices and circuits. [Neophytos Lophitis, 2018, Synopsys,]

1.3.1 Machine learning in TCAD

ML has grown in significance as a tool in Technology Computer-Aided Design (TCAD), which is used to create cutting-edge semiconductor devices. ML has multiple applications in TCAD, such as device simulation, optimizing design, modeling process and testing of device.

Modeling device process

Process modeling includes creating mathematical representations of how a semiconductor manufacture process behaves. Large volumes of process data can be analyzed using ML techniques, which can then be used to create precise models that can be used to forecast the behavior of the process.

Device simulation

The electrical and physical characteristics of semiconductor devices are modeled during the device simulation process. By examining vast volumes of data from simulations and experimental results, ML techniques can be utilized to increase the precision of these models.

Optimizing device

Using ML techniques to automatically create designs that satisfy particular performance requirements is known as design optimization. This can be especially helpful when designing complicated semiconductor devices because it may take too long or cost too much to employ conventional design techniques.

Digital Twin generation

A digital twin is a virtual duplicate/twin of a physical semiconductor device, which once trained, can be used to simulate its behavior under different conditions. Digital twins can be utilized in TCAD to forecast semiconductor device performance without the requirement for physical testing.
By feeding advanced and significant TCAD FEM simulations of a certain semiconductor device into ML algorithms, a digital twin can be generated which can mimic the performance of semiconductor devices under various operating situations and predict potential problems before they arise.
In order for an ML model to be labeled as a digital twin, it must pass certain evaluation thresholds. The evaluation criteria used for generating these digital twins is discussed in detail in Chapter 3.
Using a digital twin can provide several benefits in TCAD, such as optimizing device design, improving performance as well as reducing time and cost in physical testing and evaluation.[Raphael Wagner, 2019]
Using a digital twin has several advantages over TCAD FEM simulations. TCAD simulation can only represent a semiconductor device with fixed input variables values. If the same device needs to be tested with slightly different input variable values, then that would require a re-run of TCAD simulation leading to time and computational costs. This can be overcome with a digital twin which once trained, can be utilized to mimic the behavior of a semiconductor device within a defined range.
In summary, the effectiveness and efficiency of TCAD can be considerably increased with the help of ML. [Paul Jungmann, 2023, Neophytos Lophitis, 2018]

1.4 Problem statement

1.4.1 Broader context

The amount of data required for training an ML model depends on a number of different factors, including the complexity of the task itself, the quality of the data being utilized, and the type of ML model used. In general, an ML model is more likely to perform well the more data if it is available. However, gathering a sufficient amount of data to get accurate results out of an ML model, can be sometimes difficult.

1.4.2 Specific problem

In order to create an accurate digital twin for a semiconductor device, sufficient amount of TCAD FEM simulations are required to train an ML model for it to be accurate enough, based on the evaluation criteria, to be called a digital twin.

These TCAD FEM simulations, due to their complex nature, take quite some processing time. For a simpler/ standard semiconductor device with around 30-40 input plus output variables, A single TCAD simulation can take up to 2 CPU hours to complete, for complex devices with more than 50 input plus output variables, it can take up to 8 CPU hours as shown in Figure 1.1

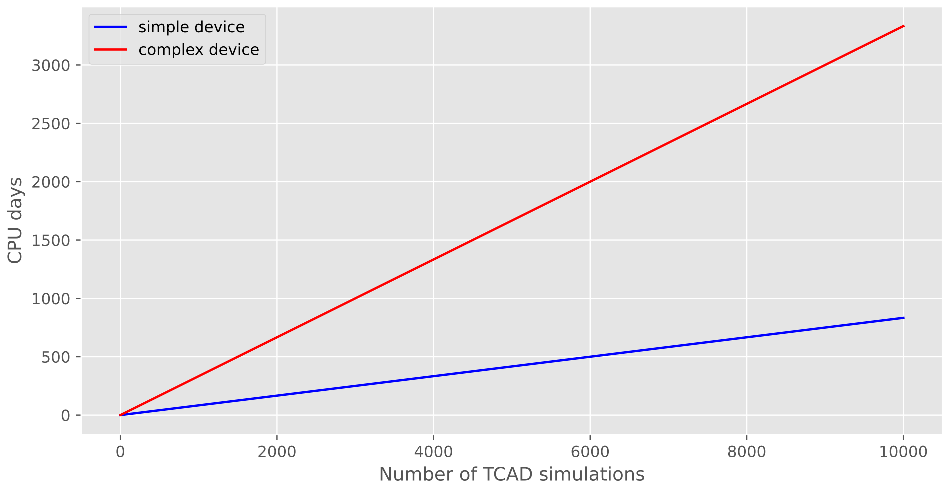


Figure 1.1: TCAD FEM simulations run-time comparison between a simple and a complex device

1.5 Scope of study

The main objective of this thesis is to develop digital twins for multiple semiconductor devices efficiently and swiftly, by overcoming the bottleneck, that is the number of TCAD FEM simulations required, in result, reducing processing time and cost.

The initial step for this process is to attain a highly optimized artificial neural network for the first device, which can perform accurately with the least amount of training data/TCAD FEM simulations required. This step involves hyperparameter optimization techniques as well as smart data selection, which will be covered in detail in Chapter 3.

The next step, which is the main focus of this study, is to conduct transfer learning, to reduce the number of TCAD FEM simulations required for training and generating these digital twins for other devices. In the semiconductor industry, there exists a vast variety of semiconductor devices. For each of these devices, there exist multiple variants, with most of the device physics being the same (target area for this study). In these particular cases, there is no need to train ML models from scratch for each one of these device variants, rather transfer learning can be used to transfer device characteristics learned information from a trained ML model of one device to another similar device's untrained ML model. This will be covered in more detail in Chapter 3 as well.

1.6 Related research questions

There are multiple research related questions/topics that can emerge from this study.

- Effectiveness of ML in semiconductor industry.
- Influence of ML in TCAD.
- Advantages of transfer learning in TCAD.
- Dependency of ML models on TCAD FEM simulations.
- Potential of digital twins.
- Predicting feature importance for device characteristics w.r.t its electrical performance.
- Predicting device behavior under different conditions using a digital twin.
- Fault/abnormality detection in device behavior using a digital twin.

Chapter 2

Review of literature

2.1 Introduction

The topic of ML as well as transfer learning has been extensively researched upon. Research related to this study will be highlighted in this chapter.

2.2 Literature split

2.2.1 Artificial Neural networks

Artificial neural network (ANN), a computational model, is modeled after how the human brain operates. It is a series of algorithms that seek to understand the patterns and relationships in a set of data. An ANN is a collection of linked processing units (neurons) that can learn from examples and generalize from them. Predictive analytics, speech recognition, image recognition, and natural language processing are just a few of the many applications that ANNs are utilized for.

Structure

An input layer, one or more hidden layers, and an output layer make up artificial neural networks, as shown in Figure 2.1. These layers are built up of neurons which are the most fundamental processing units of a neural network.
<u>Neurons</u>:
As mentioned before, the fundamental units of artificial neural networks (ANNs) are neurons. They are mathematical operations that mimic the actions of real-world brain neurons. A neuron in an ANN receives inputs, processes those inputs, and then generates an output signal that is delivered to further neurons or output nodes. In an ANN, a typical neuron can contain multiple inputs, each of which is assigned a weight value. The output of these neurons is determined with an activation function which typically consists of a linear and non-linear part. The output of the linear part is simply the sum of the product of input variables along with their weights vectors, biases are also added into this linear part if they exist. The

output from the linear part is fed into the non-linear part which determines the final output of the neuron, more details about the activation function are shared a bit further into this Chapter.[Nielsen, 2015b]

Input layer:

Input data is delivered to the input layer, typically in the form of vectors or matrices. The neurons in the input layer are solely responsible for just carrying and merely transferring the input data to the neurons in the subsequent layer without performing any calculations on it.

Hidden layers:

An ANN can have one or more connected hidden layers, based on the complexity of the task. Numerous neurons make up the hidden layers, which process the input data. Each neuron in a hidden layer gets input from the neurons in the previous layer, processes the weighted sum of those inputs using a nonlinear activation function, and then produces an output that is sent to the neurons in the layer ahead. In the case of a deeply connected neural network, each of the neurons in its hidden layers are connected to all the neurons in the previous and next hidden layers. Each of these connections has a weight assigned to it which is learned and updated during the training process. More in depth coverage of the training process is provided in Chapter 4.

Output layer:

The network's final output is produced by the output layer. The type of problem being solved determines how many neurons are present in the output layer. One neuron, for instance, would be present in the output layer for a binary classification problem, whereas several neurons would be present in the output layer for a multi-class classification or a regression problem. This final output is generated by neurons using an activation function on the weighted sum of their inputs.[Nielsen, 2015b]

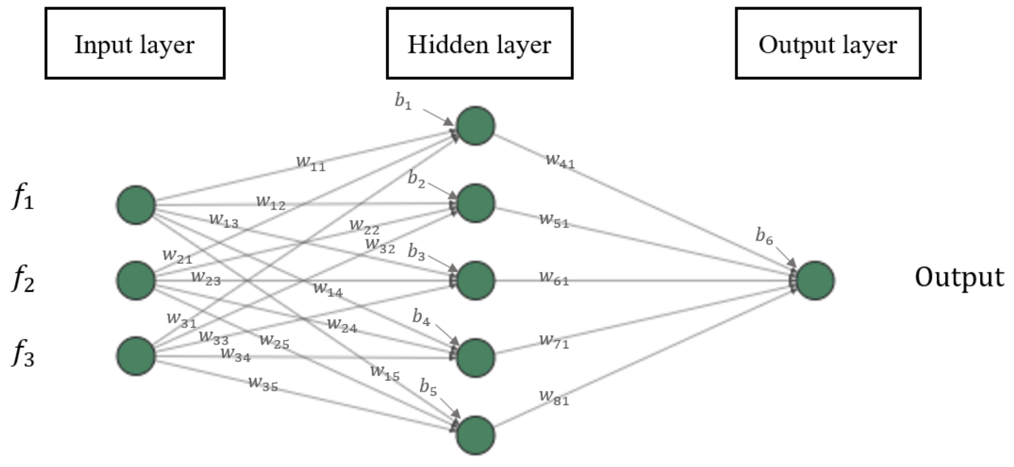


Figure 2.1: A fully connected ANN structure with weights w_{ij} and biases b_i

Activation function:

Artificial neural networks (ANNs) require an activation function if the task at hand is non-linear and complex. Activation function gives a neuron's output non-linearity and enables the network to simulate intricate connections between inputs and outputs. The network's performance can be significantly impacted by the choice of activation function. The type of activation function to select depends upon the task on hand as well as the architecture of ANN. There are various activation functions to choose from, some of the common ones are mentioned below.

Sigmoid function: This function is helpful in binary classification jobs since it scales any input value to a value between 0 and 1, in other words, normalization, as shown in Figure 2.2.

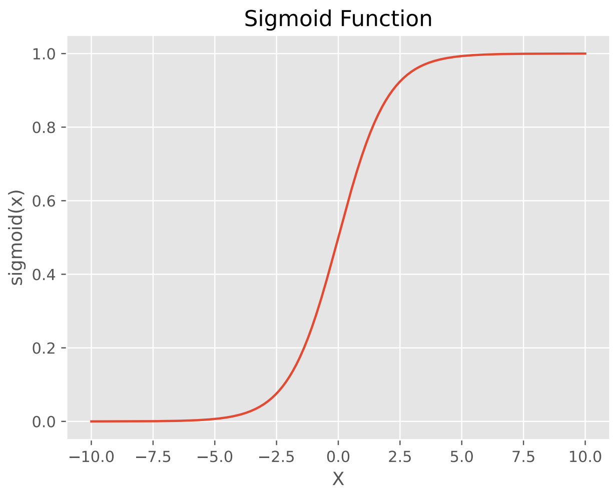


Figure 2.2: Sigmoid function

Tanh function: For cases where the output can take on negative values, the Tanh (hyperbolic tangent) function is helpful because it scales input values to values between -1 and 1, as shown in Figure 2.3.

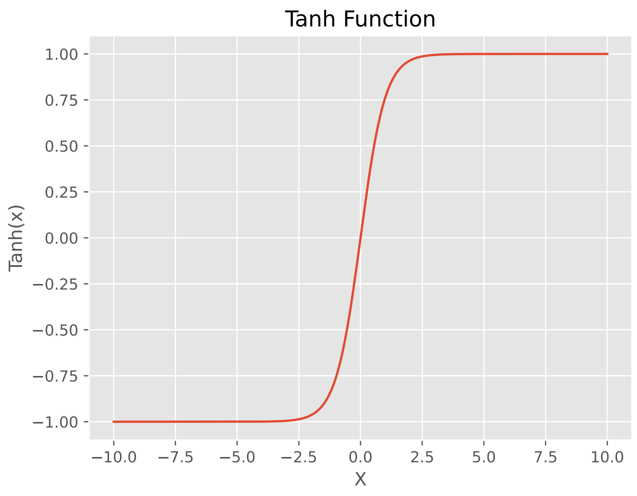


Figure 2.3: Tanh function

Softmax function: For multi-class classification issues, the softmax function is frequently utilized as the activation function in the output layer of a neural network. A probability distribution over the classes is returned. It is quite similar to the sigmoid function, the main difference is that the sigma function is used for binary classification, 2 classes, while softmax is used for multi-class tasks, as shown in Figure 2.4.

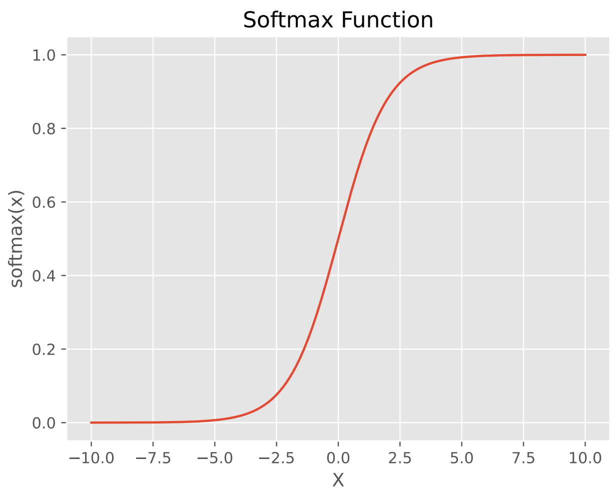


Figure 2.4: Softmax function

ReLU function: If the input value is positive, the ReLU (Rectified Linear Unit) function returns the value; if the value is negative, it returns zero. It is a well-liked option because of how easy it is to use due to its straightforwardness and how well it solves the vanishing gradient issue, as shown in Figure 2.5.

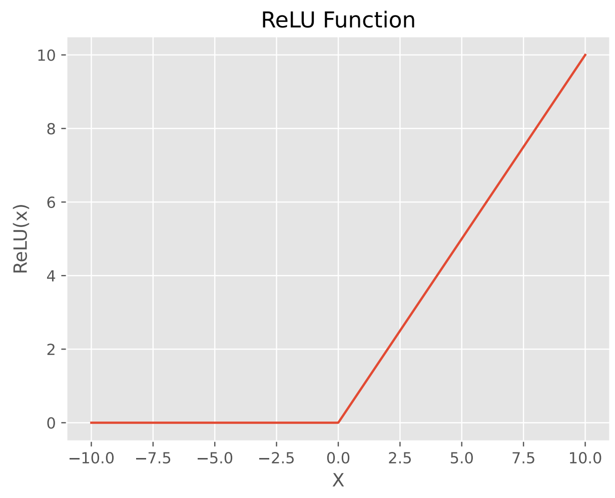


Figure 2.5: ReLU function

The "dying ReLU" problem, on the other hand, is an issue that ReLU may encounter. When the input to a ReLU neuron turns negative and remains negative for a long time, it creates the "dying ReLU" problem. In this scenario, the neuron's output is always 0, rendering it functionally "dead" and preventing it from contributing to the output of the neural network. The input to a ReLU neuron may turn negative during training when the weights are modified, or when the ReLU function is utilized with high learning rates. In order to avoid this "dying ReLU" problem, there are multiple modified versions of ReLU.

Leaky ReLU: In this ReLU variation, a small negative value is used in place of the zero output for negative inputs. As a result, the neuron never truly "dies" and continues to contribute to the output of the network.

Parametric ReLU (PReLU): This ReLU variation adds a learnable parameter (α) that controls the function's slope for negative inputs, as shown in Figure 2.6 & Equation 2.1. Performance can be enhanced by allowing the network to adjust the slope of the activation function to the input data. [Bing Xu, 2015]

$$f(x) = \{\alpha x \; if \; x \; < \; 0, \; x \; if \; x \; \geq 0\} \tag{2.1}$$

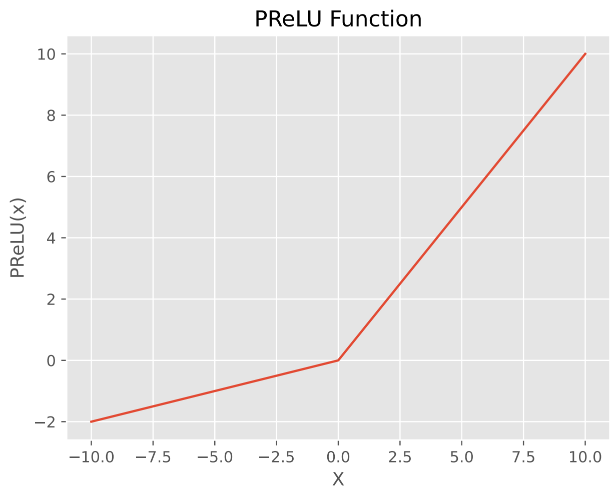


Figure 2.6: PReLU function (alpha = 0.2)

Loss function:

A loss function in artificial neural networks (ANN) is a mathematical function that measures the difference between the network's predicted output and the actual output value obtained from test data. The loss function seeks to measure the network's performance for a given task, such as classification or regression. In a standard ANN's training process, the loss function is used to guide the model's optimization process and reduce overall loss, by updating the model's weight vectors and biases.

Finding the weights and biases in the network that minimize the loss function throughout the full training dataset is the aim of the training procedure. Gradient descent, or one of its variants, is commonly used to do this. [Nielsen, 2015a] This method entails computing the gradient of the loss function relative to the model parameters and updating the parameters in the direction of the negative gradient. There are multiple loss functions to choose from, the choice depends upon the data set as well as the task at hand.

Mean squared error(MSE): If the task at hand is a regression task, then mean squared error is mostly preferred. [Scikit-learn, c] Over all samples in the training data set, it calculates the average of the squared difference between the network's predicted output and the actual output, as shown in Equation 2.2.

$$\text{MSE} = 1/n \times \textstyle\sum_{i=1}^{n}(y_i - \bar{y}_i) \tag{2.2}$$

Example: for a data set with n = 100 samples with their predicted output values being in a uniform range (-5,5) with interval 0.1 between each predicted value (-5, -4.9, -4.8, ... , 4.8, 4.9) and true output value = 2.5. We obtain an MSE value of around 9.5, as shown in Figure 2.7

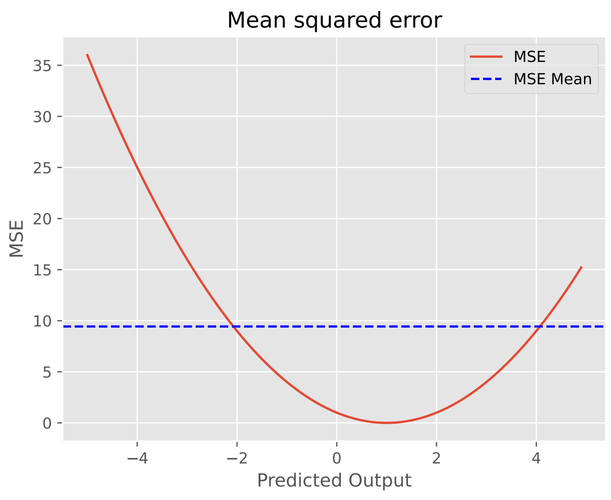


Figure 2.7: Plot displaying MSE and mean MSE values for the stated example

cross-entropy: cross-entropy is a popular loss function for classification tasks. In order to encourage the network to assign higher probabilities to the proper class and lower probabilities to the incorrect classes, it assesses the difference between the predicted probabilities of each class and the true probabilities, as shown in Equation 2.3. For classification issues, the cross-entropy loss function is chosen over other loss functions like the mean squared error (MSE) because it can handle imbalanced data sets better and is more resistant to outliers due to its higher sensitivity to class imbalance. It can also be extended to multiclass classification problems by using softmax function to calculate the predicted probabilities of each available class. [Anqi Mao, 2023]

$$\text{Cross-entropy} = -\left[y\log(p) + (1-y)\log(1-p)\right] \tag{2.3}$$

Example: for a binary classification problem with a data set with n = 100 samples with their predicted probabilities output value being in a uniform range (0,1) with interval 0.01 between each predicted value (-1, -0.09, -0.08, ... , 0.08, 0.09) and true output value = 1. We obtain a binary cross entropy with a minimum value (0) at true output/label value (1) and swiftly increasing values as predicted probability values deviate from the true label, as shown in Figure 2.8

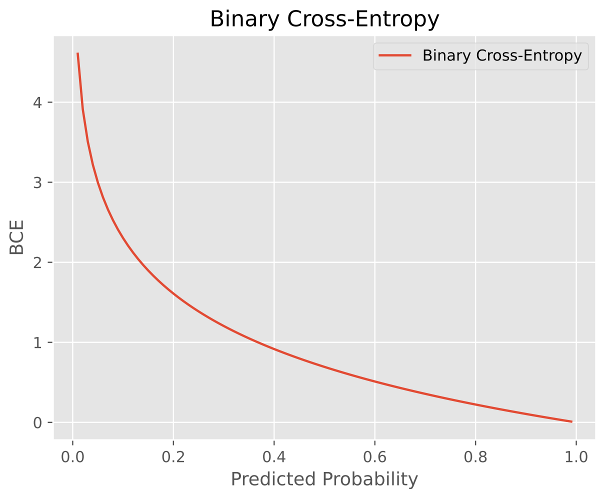


Figure 2.8: Binary cross-entropy values for the stated example

Apart from mean squared error and cross-entropy, there are other loss functions that can also be used depending upon the data set and task, such as Huber loss function, mean absolute error, etc. [Scikit-learn, c, PyTorch, a]

<u>ANN training:</u>

The training process of an ANN is discussed in detail in Chapter 4.

Evaluation metric:

The difference between the anticipated output of the ANN and the actual target output is measured using the loss function. During training, the ANN's objective is to reduce this error or loss. In general, the loss function is used to optimize the training process, while the evaluation metric is used to evaluate models performance during training. The performance of ML model is tested using the evaluation metric on unseen data/test data. Usually, it is a more intuitive measure than loss function. There are many evaluation metric to choose from, the choice again depends upon the data type and the task at hand. Some of the commonly used ones are mentioned below.

Precision metric: This metric is used for classification tasks when the cost of false positive predictions, by the model is high. Precision metric measures the number of true predictions by the model (True positives) among all the results predicted by the model as positive (True positives + false positives), as shown in equation 2.4 .

$$\text{Precision} = \text{TP}/(\text{TP}+\text{FP}) \tag{2.4}$$

where,
TP = True positives, predictions correctly predicted by ML model as true
FP = False positives, predictions falsely predicted by ML model as true

Recall metric: This metric is usually used when the cost of false negative predictions, by the model is high. Recall metric measures the amount of true predictions by the model (True positives) among all the actual positive instances (True positives + false negatives), as shown in equation 2.5 .

$$\text{Recall} = \text{TP}/(\text{TP}+\text{FN}) \tag{2.5}$$

where,
TP = True positives, predictions correctly predicted by ML model as true
FN = False negatives, predictions falsely predicted by ML model as negative

Accuracy metric: This is one of the most basic evaluation metrics which measures the correct classifications by ML model, in percentage as shown in equation 2.6

$$\text{Accuracy} = \text{NCP}/\text{NTP} \tag{2.6}$$

where,
NCP = Number of correct predictions my ML model
NTP = Total number of predictions by ML model

Mean absolute error (MAE): MAE is one of the most common evaluation measures used in ANNs, for regression tasks. It measures the absolute difference between the predicted value, from the ML model, and the actual value, from the test data, of a variable. Because it gives an indication of how far on average the predictions are from the actual values, MAE is an important metric for regression issues. ML model is more effective at predicting the target variable when the MAE value is lower. It is important to state that MAE is less sensitive to outliers than MSE. Equation 2.7 represents the formula for calculating MAE. [PyTorch, b, Keras, a]

$$\text{MAE} = (1/n) \times \textstyle\sum_{i=1}^{n} |TV - PV| \tag{2.7}$$

where,
n = Number of instances in the dataset
TV = True value/actual value, from the dataset
PV = Predicted value, from ML model

2.2.2 Hyper-parameter optimization

As mentioned before, artificial neural networks (ANN) are powerful ML models, which once tuned and trained, can be used to solve a variety of complex tasks such as image analysis, natural language processing, speech recognition etc. However, in order to utilize the full power of an ANN, it is required to set/tune its hyper-parameters with respect to the data set. [James Bergstra, 2011, James Bergstra, 2013]

Hyper-parameters

Unlike internal parameters of an ANN such as weights and biases which are learned and upgraded throughout the training process. Hyper-parameters are parameters that the ML model cannot change but they can have a significant effect on how the model performs. These hyper-parameters need to be setup/tuned properly by the user, before the training process in order to ensure optimized learning. Selecting the right set of hyper-parameters w.r.t to the training data and the task at hand, is very crucial in order to ensure proper training leading to high model performance as well as shorter training time.
Generally, hyper-parameters can be categorized into two types, Architectural and training hyper-parameters. architectural hyper-parameters are related to the structure of ANN such as the number of layers, number of neurons in these layers, and activation functions. Training hyper-parameters are directly related to the optimization algorithm involved in the training process of ANN, these can be the learning rate, epochs (training steps) and batch size. [Smith, 2018] Some of the most common and important hyper-parameters that require careful selection are mentioned as follows:

Number of hidden layers: This is a crucial hyper-parameter as it controls the depth of an ANN, based on the number of layers selected, an ANN can be a deep neural network if the number of hidden layers is large, as shown in Figure 2.9. Such a deep neural network can be used for complex tasks. Although such a model may be quite effective, deep neural networks can be computationally expensive due to their complexity. If the number of hidden layers selected is small, then it generates a shallow network, as shown in Figure 2.10 which, based on the number of neurons it has, can be quite fast to train but at the same time, computationally less expensive.

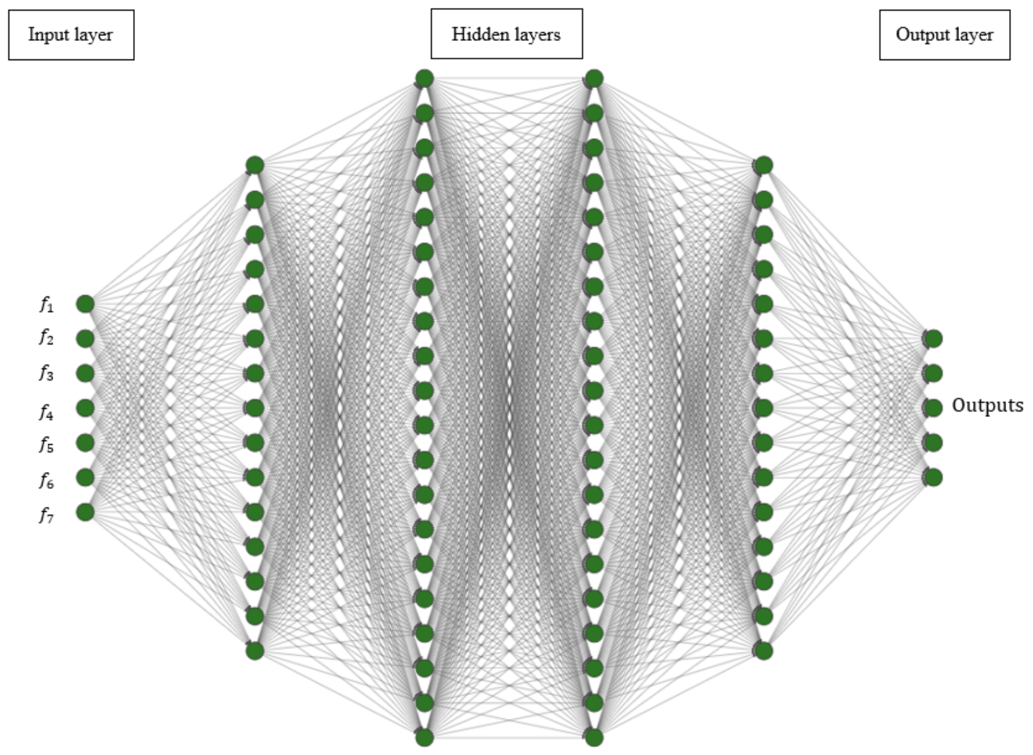


Figure 2.9: Deep ANN

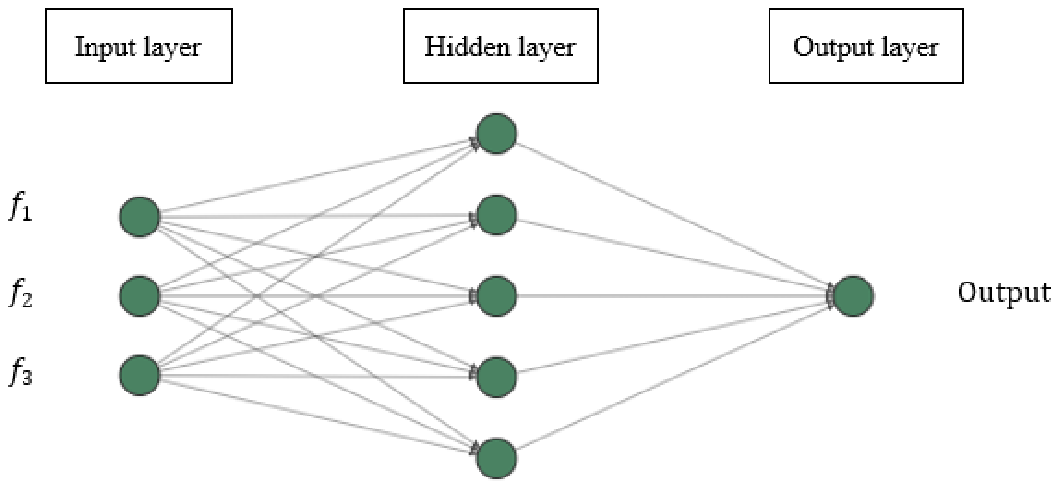


Figure 2.10: Shallow ANN

Number of neurons: This hyper-parameter dictates the number of neurons per layer. A high number of neurons per layer allows the model to learn complex relations between inputs and outputs, as shown in Figure 2.11, however high complexity results in higher computational expense, resulting in longer training times. a small number of neurons results in a simple NN structure which will have faster training but will be less complex, as shown in Figure 2.12. The number of neurons as well as the number of hidden layers should be chosen based on task complexity and the data set available. A combination of a small number of hidden layers with a large number of neurons or vice versa can also turn out to be beneficial.

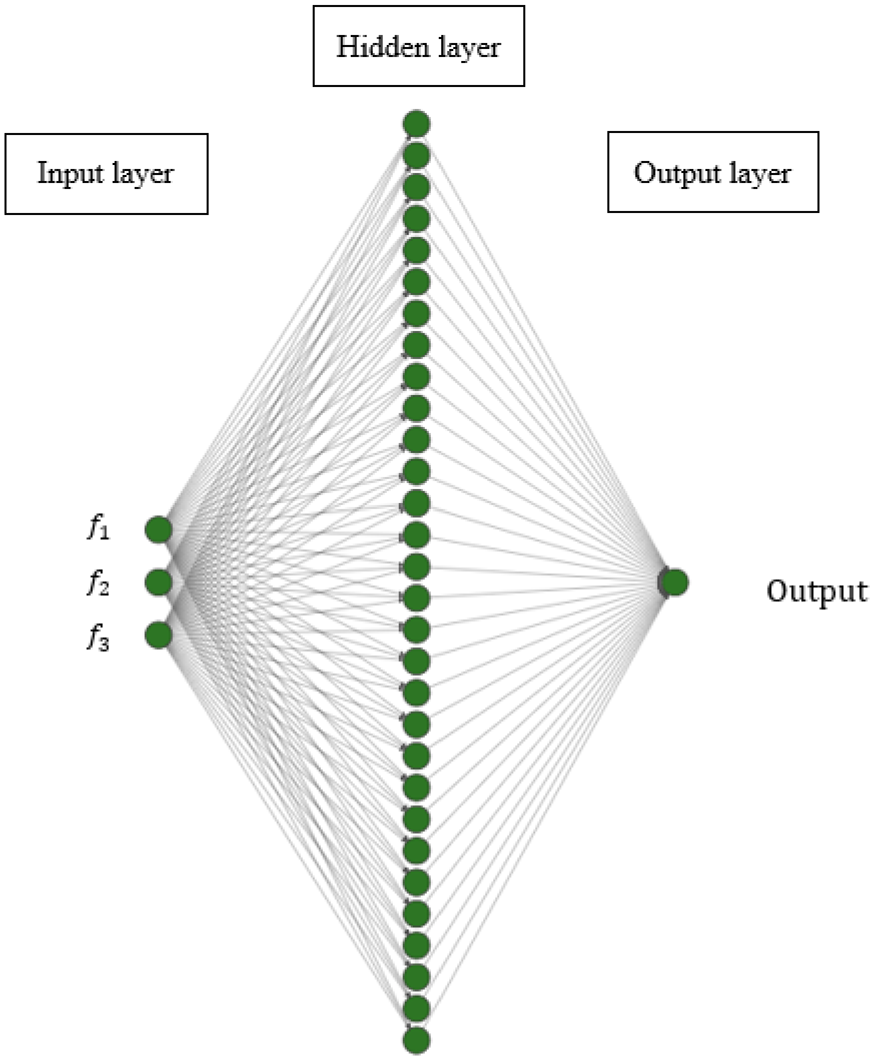


Figure 2.11: high number of neurons

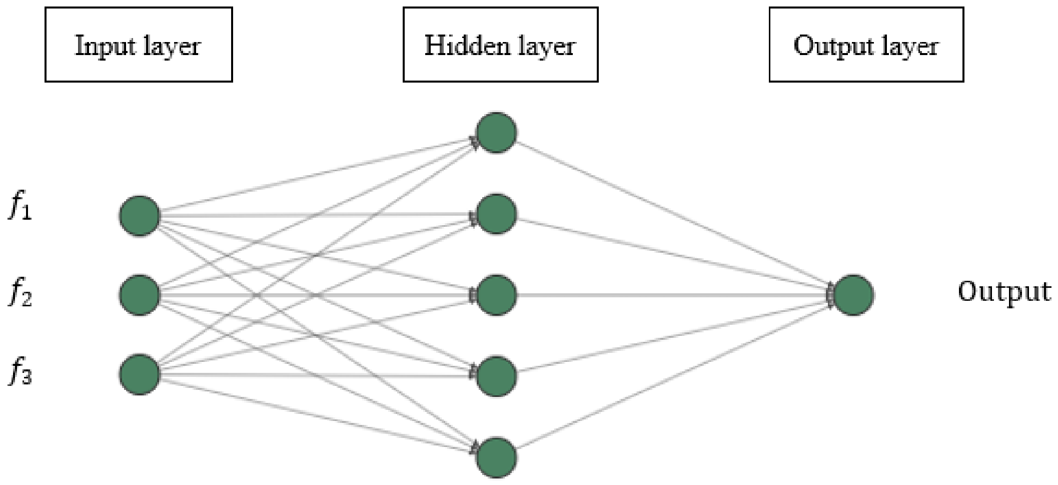


Figure 2.12: low number of neurons

Activation function: Activation function, as discussed in detail in Section 2.2.1 and shown in Figure 2.13, gives a neuron's output non-linearity and enables the network to simulate intricate connections between inputs and outputs. The network's performance can be significantly impacted by the choice of activation function. The type of activation function to select depends upon the task on hand as well as the architecture of ANN. The activation function needs to be selected before the training process is initiated.

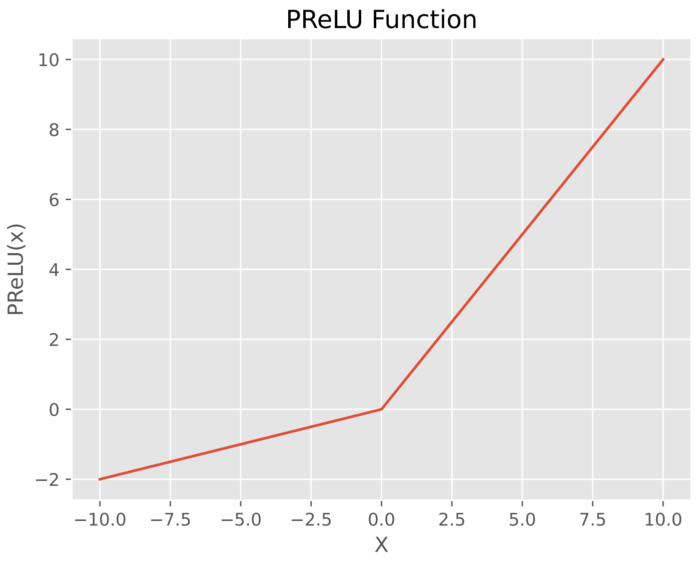


Figure 2.13: Activation function (PReLU)

Learning rate: The gradient descent optimization process, which is used to update the weights and biases of the network during training, is controlled by the learning rate. While a low learning rate can result in slow convergence, a high learning rate can cause the algorithm to overshoot the ideal solution. Based on the nature of the issue and the network's architecture, the learning rate ought to be determined.

Figures 2.14 & 2.15 show the difference between using a high and an adequate learning rate.

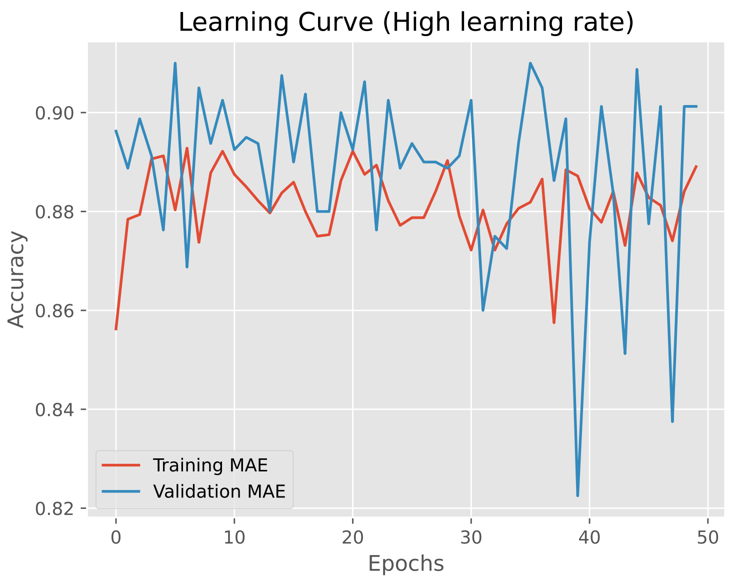


Figure 2.14: High learning rate (0.1)

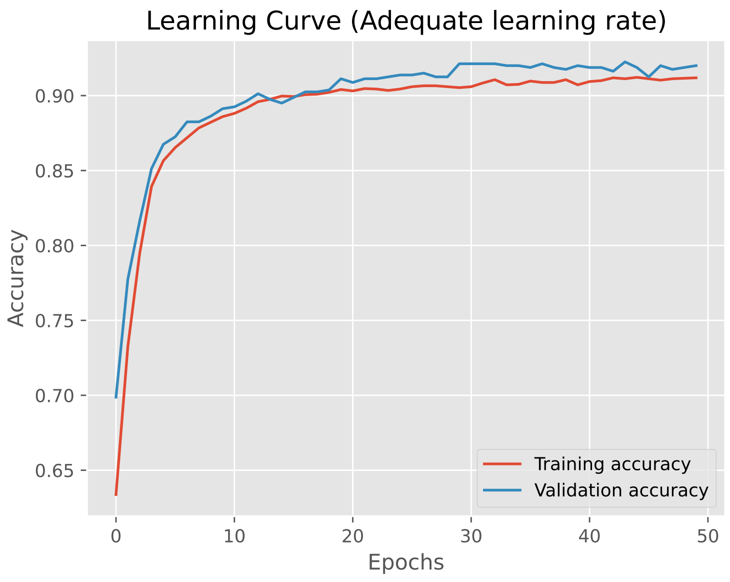


Figure 2.15: Adequate learning rate (0.001)

Batch size: The amount of samples utilized to update the network's weights and biases throughout each iteration of the training process is determined by the batch size. Large batch sizes can hasten convergence but also increase memory requirements and degrade generalization. Although it may result in slower convergence, a small batch size can also produce better generalization. The batch size should be determined based on the nature of the problem and the available computational resources.

Apart from these hyper-parameters, regularization parameters can also be taken into account which add a penalty term to the loss function to prevent overfitting, in case the data is too noisy which can cause the model to memorize the training data too well and later fail to generalize to unseen new data.

Hyper-parameter optimization techniques

Hyper-parameter optimization is the process of finding the most ideal set of hyper-parameters with respect to the data set and complexity of the task, in order to enhance ML model's performance. This process is therefore quite necessary to implement, however, it can be quite a challenging task and therefore time-consuming, as it involves trying out different combinations and testing them (trial and error method). There are various types of techniques used for hyper-parameter optimization. [James Bergstra, 2011] These techniques can also be used in combination to produce more effective results. Some of these techniques are mentioned below.
Random search: Random search is one of the most trivial methods for finding the best set of hyper-parameters for an ML model. In its implementation, a search space consisting of ranges of values for hyper-parameters is predefined. This method randomly selects combinations of values for hyper-parameters from this search space and tests them out. In case of a wide and high dimensional search space, this method can be effective, as it would be computationally expensive to search through all the search space. However, since this method involves random selection. Depending on the number of random searches, it is quite possible that this method does not select the most optimal set of hyper-parameters for a given task. [Scikit-learn, d]
A small example of random search algorithm is shown in Figure 2.16, in which a random search algorithm is used to select and test combinations (cyan colored points) from a predefined search space, of two hyper-parameters, learning rate and number of layers. It can be seen that although random search fails to select the most optimal combination, it does however generate a few combinations which lie close to it, highlighted within the red circle.

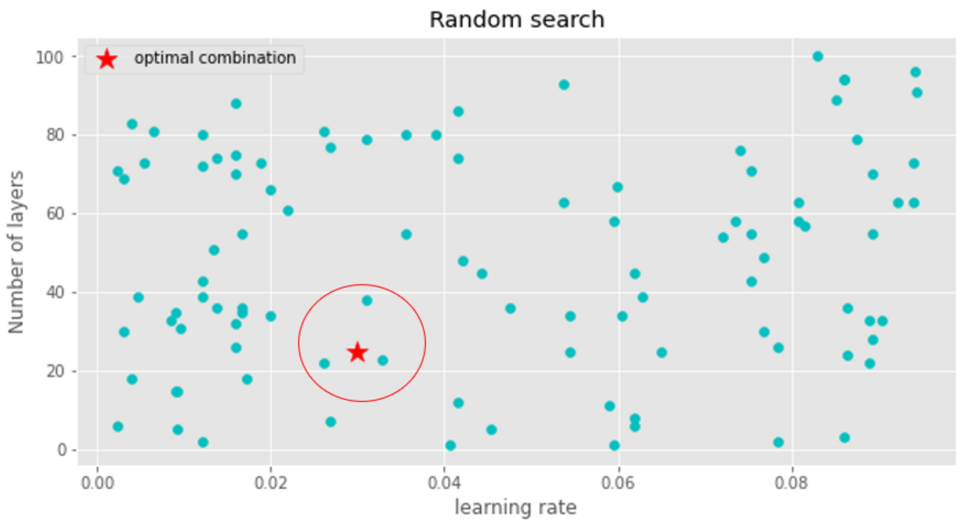


Figure 2.16: Random search

Grid search: Grid search method is one of the brute force algorithms utilized to search for the optimal set of hyper-parameters. Similar to random search, a search space consisting of ranges of values for hyper-parameters, is predefined. Grid search method selects and tests out each of the possible combinations for these hyper-parameters. As this method goes through all possible combinations, depending upon how the search space is defined, it ensures the selection of the most optimal set of hyper-parameters. Grid search is simple to implement and is quite effective, however, it is also computationally expensive. In case of a wide and high dimensional search space, this method would not be ideal to use. For a simple and non complex case, grid search is one of the most favourable choices.[Scikit-learn, b] A small example of a grid search algorithm is shown in Figure 2.17, where it is used to test out all combinations of values, from a predefined search space, for two hyper-parameters, epochs and number of layers. It can be seen, that as grid search generates all possible combinations of hyper-parameters (cyan colored points) from the predefined search space, it manages to also obtain the most optimal combination, highlighted within the red circle.

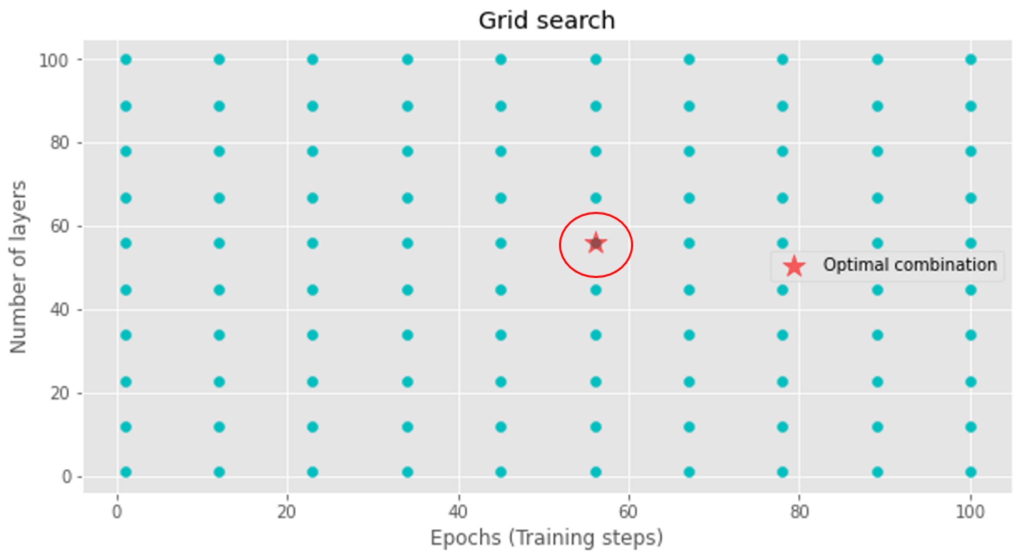


Figure 2.17: Grid search

Bayesian Optimization: Bayesian optimization algorithm is one of the popular approaches utilized for finding optimal combination of hyper-parameters for a given task. In Bayesian optimization, the search for the ideal hyper-parameters is guided by a probabilistic model of the objective function. As new evaluations of the function are acquired, this model is updated to reflect both the observed data and prior understanding of the function. The next set of hyper-parameters to examine are then suggested by the model based on a trade-off between discovering new regions of the parameter space and utilizing promising ones. More details on the methodology of Bayesian optimization will be presented in Chapter 3.
Compared to other optimization techniques like grid search and random search, Bayesian optimization has a number of benefits. It can handle noisy and non-convex functions more efficiently and be more effective at finding good solutions with fewer evaluations of the objective function. However, because it necessitates creating and maintaining a probabilistic model of the function, it is more computationally expensive.[Nguyen, 2019, Scikit-learn, a]
Considering Equation 2.8, which represents a simple function with three variables x_1, x_2 and x_3 with each having a search space of (-5,5), it takes Bayesian optimization algorithm less than 20 trials to find a global minimum for this function, this can be seen from the convergence plot shown in Figure 2.18 [Scikit-optimize,]

$$f(x) = x_1^2 + x_2^2 + x_3^3 \tag{2.8}$$

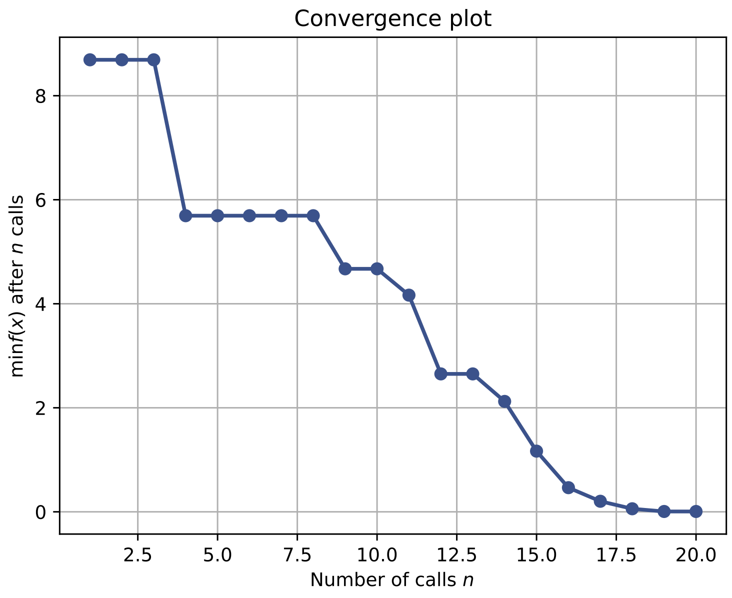


Figure 2.18: Bayesian optimization

Apart from the techniques presented, there exist other methods for hyper-parameter optimization such as evolutionary algorithms and gradient-based optimizations.

2.2.3 Transfer learning

Transfer learning is a subsection of ML that utilizes learned information from the ML model of one task, to aid the training process and thus improve performance for another ML model for a different but related task. Transfer learning can come in very handy especially when the amount of data required to train a ML model is limited/insufficient, or when training from scratch can be computationally very expensive in terms of resources or time. [Sinno Jialin Pan, 2009]
Transfer learning has a lot of important use cases in many fields. Some of them are mentioned below.

Application

Speech Recognition
Transfer learning in this field has been used to aid/improve performance on various tasks such as speaker identification as well as speech-to-text conversion. This is done by using a pre-existing highly trained ML model trained on a large amount of speech recognition data and fine-tuning it on comparatively smaller data more specific to the task at hand.
Computer Vision
One of the fields that transfer learning dominates in is computer vision. Within this field ML is used for many tasks such as the classification of images, detection of objects, face recognition, image enhancement, and semantic segmentation. One example of the use case of transfer learning in computer vision would be for the task of detecting a specific breed of dogs, let's say, German shepherds. For this task, a pre-existing ML model highly trained on a dataset of dog images can be used and then fine-tuned on a comparatively smaller set of data specifically containing various images of German shepherds, leading to higher accuracy and reduction in the requirement of specific training data.
Natural language processing
ML has multiple applications in the field of natural language processing such as question answering, classification of text, text translation, sentiment analysis, word/sentence prediction, and named entity recognition. Transfer learning can help improve performance for these tasks. For example, one of the famous NLP models BERT which is trained on predicting words/sentences in a high context environment can be finetuned on a low context small dataset for it to be compatible for low context word prediction. [Jacob Devlin, 2018]

Methodology

Using a pre-existing model

The first step for using transfer learning is to select a pre-existing highly trained ML model. It is important to take into consideration that the task for which this pre-existing model was created, should be similar to the new task at hand, in order to ensure the transfer of relevant knowledge. Once this criterion is satisfied, then this pre-existing data can be utilized by fine-tuning it to a smaller dataset specific to the new task at hand.

Freezing layers

The idea behind this method is that, instead of fine-tuning the entirety of the pre-existing model, some of the layers can be frozen, meaning that they do not take part in the training process and the information they hold is kept intact, while the remaining layers are trained and updated on the dataset relevant to the new task. This methodology can be implemented when some of the layers of the pre-existing model carry important information that requires to be kept intact and not updated. It also accelerates the training process.

Apart from this, it can also be possible that transfer learning might be required to transfer knowledge/information from a task of one field to a new task of a different field. In such cases, transfer learning may not be as effective but can still aid in the training process. In a use case like this, there would be a need for a comparatively larger set of training data relevant to the new task as well as using some techniques such as domain adaptation.

Transfer learning can be effective for many applications. By reducing the amount of training data required, it further reduces the resources and time required for tasks. This in general leads to improvement in overall performance. The amount of influence that transfer learning can make depends on the relation/similarity between the previous task, from which information is transferred, and the new task, to which information is being transferred. Its performance is also affected by what type of information/knowledge is transferred between ML models. It also depends on the amount as well as quality of the training data available for the new task, for finetuning. [TensorFlow, , Keras, b].

Transfer learning implementation is discussed in detail in Chapter 4.

2.2.4 Transfer learning in TCAD

Section 1.3.1 showcases multiple applications of ML in TCAD such as modeling device process, device simulation, optimizing devices, and digital twin generation. [Paul Jungmann, 2023, Neophytos Lophitis, 2018, Synopsys,] Transfer learning is very suitable in the semiconductor industry, specifically, TCAD as these computer simulations that are generated to simulate a semiconductor device's behavior are computationally quite expensive. Transfer learning reduces the amount of these TCAD FEM simulations required for different applications and therefore is quite useful. Some of the ways in which transfer learning can be applied in TCAD are as follows:

Extraction of features for process optimization

From vast datasets of process parameters and performance, important features can be extracted via transfer learning. A model can learn features important for process optimization by being trained on a sizable dataset of process parameters. The best process parameters for achieving desired performance goals can then be predicted by models created using these extracted important features.

Defect detection using pre-trained ML models

The accuracy and effectiveness of fault detection in TCAD can be increased by using pre-trained ML models. These models can learn features/attributes that are similar across several devices since they are trained on big datasets of device attributes. The accuracy of defect identification can be increased by fine-tuning these pre-trained models on certain device types and defects, at the same time reducing the overall computational cost.

Process monitoring

Pre-trained models can be utilized for usage in particular process monitoring domains via transfer learning. Pre-trained models can be improved by applying fine-tuning with particular datasets of sensor data from the manufacturing process. By doing so, the models can learn to recognize abnormalities and anticipate possible problems in the given area. This can decrease the quantity of labeled data needed for training while increasing the accuracy and effectiveness of process monitoring.

Device modeling

The accuracy and effectiveness of device simulations in TCAD can be increased by using pre-trained models for device modeling. These models can learn properties that are similar across several devices as they are trained on big datasets of device characteristics and performance. The accuracy of the simulations can be increased while the computing cost is decreased by fine-tuning these pre-trained models on particular device types.[Yang Li, 2021, Dyd Pradeep, 2023]

As mentioned before, TCAD simulation run-time is the main bottleneck in most of the applications TCAD is used for due to their complexity and required processing time. Transfer learning, if applied efficiently, can help in overcoming this bottleneck. More details are shown in Chapter 4

2.3 Summary of literature

To summarise the literature review of Chapter 2. There has been a lot of research and development done for:
Artificial neural networks: in terms of their structural properties such as the number of hidden layers, number of neurons, etc., as well as its parameters and hyper-parameters, activation functions, loss functions, and evaluation metrics, optimization techniques for its hyper-parameters.
Transfer learning: in terms of what transfer learning is, its different applications and use cases as well as different methods to apply it efficiently and effectively.
Machine learning and transfer learning in TCAD: in terms of different use cases and applications where ML has aided in different TCAD processes, as well as the improvement in overall performance by using transfer learning in TCAD.

2.4 What is missing

Although there has been a decent amount of research and development done with respect to ML in TCAD. There is still very little to no research done on utilizing transfer learning to generate Digital twins(Trained and optimized ML models) for semiconductor devices. There still exists the need to utilize large amounts of TCAD FEM simulations to generate these digital twins, which are, as mentioned before, computationally very expensive and time-consuming.
As mentioned in Section 1.5 , this is the main objective of this study, to come up with an efficient and effective way to utilize the power of transfer learning to develop these digital twins while reducing the bottleneck of heavily depending on a large amount of computationally expensive TCAD FEM simulations.

Chapter 3

Prerequisites

This chapter showcases the prerequisites that are crucial for implementing transfer learning to aid in the development of digital twins.

3.1 Data compilation

3.1.1 Details of data

The data used for training and testing ML models which act as digital twins to a semiconductor device, are TCAD FEM simulations which simulate the behavior of a semiconductor device with specific feature details. In these FEM simulations, based on the semiconductor devices considered for this study, the number of input variables ranges from 9-10 while the number of output variables ranges from 24-27.

3.1.2 Training data

As mentioned in the previous chapters, these TCAD FEM simulations are computationally quite expensive. Therefore, it is necessary to have a smart way of selecting these simulations to ensure that the number of simulations required is as low as possible without affecting ML models' performance.
One of the methodologies implemented to aid in the training process is to ensure that axial and feature pairwise corner samples are included in the training data.

Axial samples

Axial samples are samples generated by one-factor-at-a-time method (OFAT) which is a part of design of experiments. It involves testing out variations in the value of one feature while keeping the rest of the features at their optimum/target values. Using OFAT method while generating TCAD FEM simulations can lead to finding an optimum for the entire process [John Wiley & Sons, 2017]. Axial samples are added to the training data to ensure that optimum values as well as values close to the optimum values, of features and output variables are a part of it. Figure

3.1 shows axial samples generated for two features f_1 & f_2, having a mean value of 5 and range (0,10).

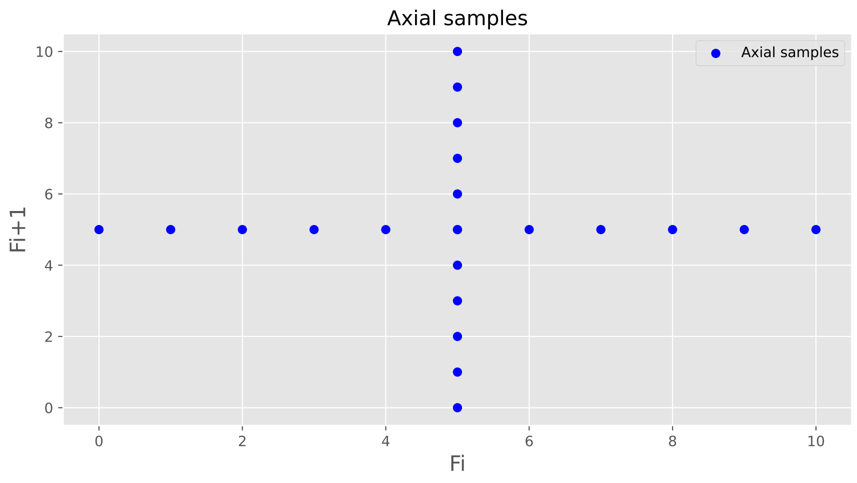


Figure 3.1: Axial samples

Adding axial samples into the training dataset leads to a slight improvement in model's performance without impacting training time.

Feature pair wise corner samples

Adding feature pair wise corner samples into the training data ensures that extreme data points, which might be mistaken for outliers in a standard case, are included in the training process. Figure 3.2 shows feature pairwise corner points along with axial samples previously shown in Figure 3.1.

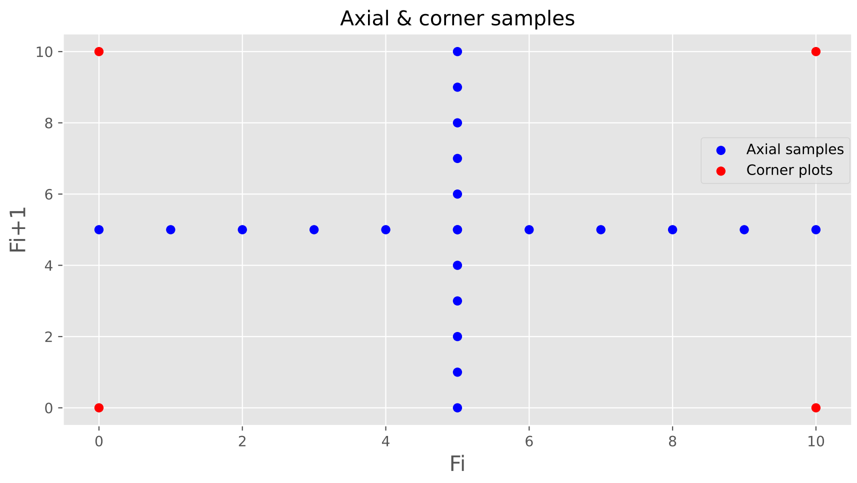


Figure 3.2: Axial & corner samples

The approach for training data selection is to first add axial and feature pairwise corner samples into the training data and then add randomly selected TCAD FEM simulations generated by Latin hypercube sampling [McKay, 1979], into the training dataset, as shown in Figure 3.3, till an adequate performance from the ML model is achieved, based on the evaluation criteria.

This further reduces the amount of TCAD FEM simulations required in the training process without having a negative effect on the model's performance.

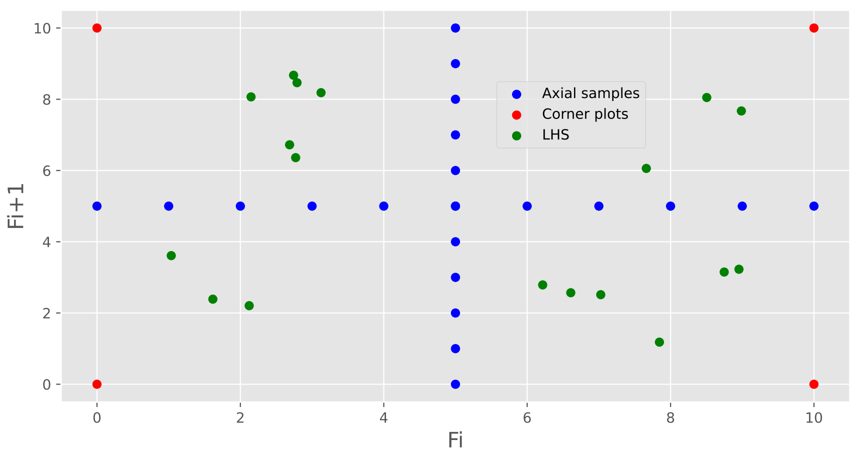


Figure 3.3: Axial, corner & LHS samples

3.1.3 validation & test data

Latin hypercube samples are uniformly selected throughout the available TCAD FEM simulations for validation and evaluation. Uniform selection from the entire range of samples, separate from the ones used for training, ensures a non-biased evaluation measure. 40 percent of the LHS uniformly selected are used during the training process for validation while the rest are used for evaluating the ML model's performance after the training process has been completed.

3.2 Hyper-parameter optimization

As mentioned in the literature review, hyper-parameter optimization involves tuning hyper-parameters with respect to the task and dataset at hand. Hyper-parameters are parameters that the ML model cannot learn and update during the training process. These are external parameters that require careful selection for achieving optimum performance from ML model
As stated in Chapter 2, there are multiple ways in which hyper-parameter optimization has been carried out in the past. The methodology implemented in this study involves a combination of random and Bayesian optimization searches. Multiple random but unique combinations of hyper-parameters are generated by random search which are fed into ML models for testing. The combinations which provide the best ML models, based on their evaluation scores, are used as a starting point to conduct a Bayesian optimization search, which after the search process provides optimal hyper-parameters with respect to the given dataset. The reason behind conducting a random search before Bayesian optimization is to have a good starting point for Bayesian optimization, in other words, reducing the chances of

having the model converge to a local minimum. The other benefit is to reduce the computational overhead generated by Bayesian optimization search by providing a good starting point.
Hyper-parameter optimization is carried out with the following steps.

3.2.1 Selection of hyper-parameters

The first step for implementing hyper-parameter optimization is to carefully select the hyper-parameters requiring optimization. There are multiple hyper-parameters to consider, as mentioned in Chapter 2. In this study, the number of hidden layers and the number of neurons within these hidden layers were selected for the optimization process because of the vast number of combinations that can be selected in between them which would be quite time consuming to check manually through the hit and trial method, while the rest of the hyper-parameters such as activation function, epochs, batch size, and learning rate were selected based on their performance with respect to the TCAD FEM simulation data. The hyper-parameter, number of hidden layers, was later fixed to a value of one based on ML model results w.r.t to transfer learning. This is covered in detail in Chapter 4.
Activation function:
After conducting experiments with different activation functions, the Parametric leaky rectified linear unit (PReLU) results in the best performance on TCAD FEM simulation data, because of its learnable parameter (alpha) that controls the slope of its negative region in opposition to how a standard rectified linear unit (ReLU) activation function performs as shown in Figure 2.6 and Equation 3.1. This learnable parameter alpha is updated by the ML model during the training process while other weights and biases are updated.
Using the standard leaky rectified linear unit activation function with a fixed parameter defining the negative slope, also gave adequate results.

$$f(x) = \{\alpha x \; if \; x \; < \; 0, \; x \; if \; x \; \geq 0\} \tag{3.1}$$

Epochs & batch size:
Different ranges of epochs were tested and it was concluded that using 500 epochs, along with an early stopping criterion that monitors validation data loss with a patience value between 20 - 30, was the most suitable option w.r.t TCAD FEM simulations data.
Along with the selected number of epochs, using a batch size of 32 during training resulted in the most suitable performance, although a batch size of 64 also gave adequate results.

Learning rate:
By testing out different values of learning rate ranging from 0.00001 to 0.1, the best performance was achieved by using the default value of 0.001 for initial training and 0.0001 for fine-tuning during transfer learning.

3.2.2 Search space definition

After the selection of hyper-parameters to optimize, a search space is defined consisting of ranges of values for these hyper-parameters, selected based on initial testing and suitability w.r.t to the dataset. The initial search space defined for the number of hidden layers is (4,9), while (40-90) is for the number of neurons within these hidden layers. This search space is fed into the random search for space exploration and testing.

3.2.3 Evaluation criteria

Training validation metrics

Loss metric:
The function of the loss metric is to optimize the training process by measuring the difference between ML models prediction and the actual output, which in this case, is defined by TCAD FEM simulations.
The loss function opted for this task was mean squared error which is usually the most popular choice for regression tasks based on its performance. It calculates the average of the squared difference between the network's predicted output and the actual output, as shown in Equation 3.2.

$$\text{MSE} = 1/n \times \textstyle\sum_{i=1}^{n}(y_i - \bar{y}_i) \tag{3.2}$$

Evaluation metric:
While the loss metric is used to optimize the training process, the evaluation metric is used to evaluate models performance during training.
The evaluation metric opted for this task is mean absolute error, which is one of the most commonly used evaluation metrics for regression tasks. It measures the absolute difference between the predicted value, from ML model, and actual value, from test data, of a variable. Equation 3.3 represents the formula for calculating MAE.

$$\text{MAE} = (1/n) \times \textstyle\sum_{i=1}^{n} |TV - PV| \tag{3.3}$$

Post training evaluation

The final evaluation measure which determines whether a ML model is suitable for being used as a digital twin, is a combination of absolute error scores of the 99^{th} percentile, per output variable, and absolute error threshold values provided by semiconductor experts. The 99^{th} percentile absolute error per output variable is compared with its respective threshold, an ML model is considered to pass the evaluation criteria if all its output variables' 99^{th} percentile scores lie within the threshold, in other words, have evaluation score less than or equal to 1, as shown in Equation 3.4. 99^{th} percentile is chosen instead of the maximum absolute error per label to avoid outliers.

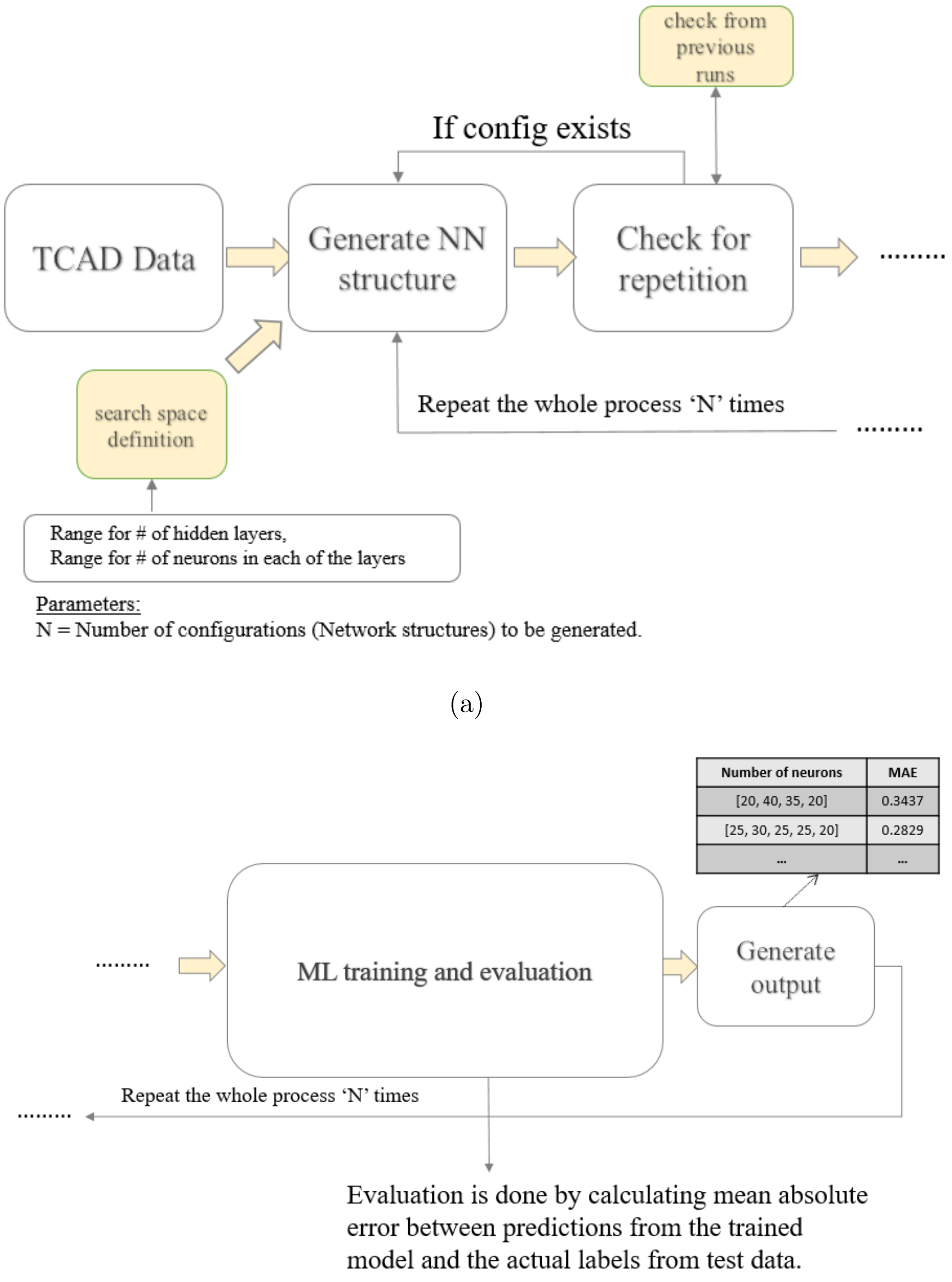
$$\text{evaluation score per output} = \frac{\text{Absolute error per output}(99^{th}\ \text{percentile})}{\text{absolute error threshold per output}} \tag{3.4}$$

$$\text{evaluation} = \begin{cases} \text{passed,} & \text{if } 0 < \text{ evaluation score of each output variable} \leq 1 \\ \text{failed,} & \text{if evaluation score for any output variable} > 1 \end{cases}$$

3.2.4 Implementation

Random search

The defined search space is fed into the random search, based on which, random search selects random but unique combinations of hyper-parameters to be optimized. Since the selection process is random and does not depend upon the previous selection, multiple ML models, with unique configurations generated from random search, are trained and validated in parallel. All the random and unique hyper-parameters configurations trained and tested by ML models along with the evaluation scores are stored. Figure 3.4 shows the flow diagram split for random search process.


check from previous runs
If config exists
TCAD Data
Generate NN structure
Check for repetition
....
search space definition
Repeat the whole process 'N' times
....
Range for # of hidden layers,
Range for # of neurons in each of the layers
Parameters:
N = Number of configurations (Network structures) to be generated.
(a)
....
ML training and evaluation
Generate output
Repeat the whole process 'N' times
....
Evaluation is done by calculating mean absolute error between predictions from the trained model and the actual labels from test data.
(b)

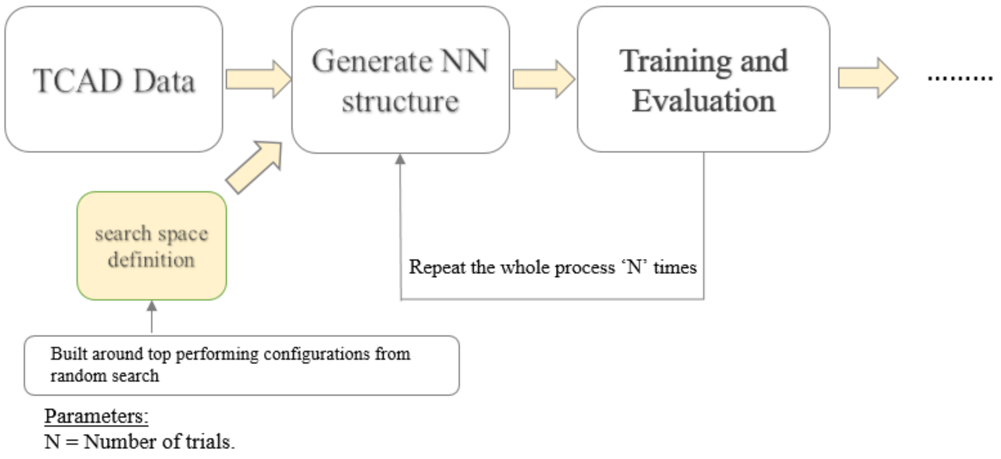
Number of neurons	MAE
[20, 40, 35, 20]	0.3437
[25, 30, 25, 25, 20]	0.2829
...	...

Figure 3.4: Random search flow diagram

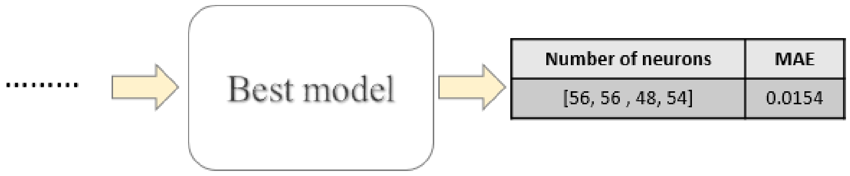
From a highly parallelized and fast random search, multiple random and unique combinations of hyper-parameters are fed into a ML model and tested out. Out of these tested combinations, one's with top evaluation scores are selected and used as a starting point for the Bayesian optimization search. A good starting point leads to a good optimization performance from the search algorithm.

Bayesian optimization search

Configurations of top performing ML models generated using random search, serve as a good starting point for Bayesian search. Bayesian algorithm develops a narrow search space around these configurations and builds an objective function where the hyper-parameters to be optimized, are updated in order to minimize the loss, that is the mean absolute error.
Bayesian optimization uses Tree structured Parzen estimator (TPE) to select hyper-parameters based on its previous selection of hyper-parameters. TPE algorithm generates probability density distributions of hyper-parameter combinations, both good and bad based on the evaluation scores, using kernel density estimators. The algorithm then performs exploitation step by sampling hyper-parameters combinations that are likely to generate good scores, based on the current density estimations. These density estimations are refined based on performance of hyper-parameters on each iteration, which guides the search process in the direction of better hyper-parameters combinations. [Meng Zhao, 2018]
Figure 3.5 shows the flow diagram split for Bayesian optimization search.



(a)



Number of neurons	MAE
[56, 56 , 48, 54]	0.0154

(b)

Figure 3.5: Bayesian optimization search flow diagram

Evaluation

The evaluation after the Bayesian search determines whether an ML model is good enough to be considered as a digital twin for a semiconductor device. Each ML model trained and tested as a result of random and Bayesian search generates absolute error (99^{th} percentile) values for each of the outputs variables. These absolute error values for each electrical output variable are compared with thresholds provided by semiconductor experts. As explained in section 3.2.3, A ML model which has MAE values lying below the threshold is selected as a 'Digital Twin'. More details about the evaluation process are shown in Chapter 5.

Optimizer

Keeping in mind the main objective of this study, which is to develop digital twins with the least amount of TCAD FEM simulations, the evaluation procedure shown in Section 3.2.3 is applied to ML models trained with different amounts of training data controlled by the optimizer.
Initially, the optimizer selects the maximum amount of training data available. The optimizer on each iteration reduces the amount of training data by a defined step size value followed by training and evaluation of the ML model. It keeps on reducing the training data on each iteration until the ML model fails the evaluation criteria. After reaching failure, the optimizer selects the ML model from its previous iteration which just passed the evaluation criteria. This ensures that the least amount of training data is utilized.

Chapter 4

Transfer learning

There are multiple semiconductor devices as well as multiple variants of each of these semiconductor devices for which a digital twin needs to be developed. Considering once again the main goal of this study, which is to reduce the amount of TCAD FEM simulations, required for training ML models as much as possible. Transfer learning aids this goal substantially by reducing the amount of TCAD FEM simulations required for training by more than half while maintaining ML models' performance. There are multiple device variants having majority of device characteristics in common, transfer learning is utilized here to transfer learned knowledge of these characteristics, their importance, and their impact on corresponding electrical output variables, from a trained ML model of one device to an untrained ML model of a different yet similar device. This learned information is carried by the weight vectors within the ML model (ANN). These weights are assigned to the connections between input variables and the neighboring hidden layer. Details involved with ML model training and updating weight vectors are shown in the subsequent sections.

4.1 ML model configuration

4.1.1 Single hidden layer configuration

A single hidden layer ANN configuration was opted for generated digital twins as it was observed through experimentation that a single hidden layer with a high number of neurons, selected from hyperparameter optimization shown in Chapter 3, performed equally as good as an ANN with multiple hidden layers with lesser neurons, for generating digital twins.
An additional reason for selecting a single hidden layer configuration was to have more transferrable weight vectors. In the case of a multi-hidden layer configuration, adding one extra input variable would lead to re-training of all the weight vectors existing in between the hidden layers as the ANN model that is used is deeply connected meaning that the additional input variable would influence the

calculation of outputs of all the neurons in the succeeding layers. This would result in the requirement of a higher number of training data for re-training the model as compared to a single hidden layer configuration consisting of lesser weight vectors that require re-training.

4.1.2 Transfer learning methadology

To explain the training process and transfer learning methodologies used in this study, in detail without breaching confidentiality, synthetic data is artificially generated based on the quadratic equation shown in Equation 4.1. This can be considered as an artificial device 1.

$$Y = 0.5 \times x_1 \times x_2^2 \times x_3^{x_2} \tag{4.1}$$

It consists of three input variables x_1, x_2 and x_3 and one output variable Y. An example of generated data from Equation 4.1 can be seen in Table 4.1.

x_1	x_2	x_3	Y
2.26	2.80	9.28	512.36
9.25	4.74	9.48	42984
1.46	7.53	4.79	131804
4.10	1.10	1.21	3.73

Table 4.1: Synthetic data

An artificial neural network (ANN) used to understand and mimic the patterns and relationships existing in the dataset shown in Table 4.1 is shown in Figure 4.1. Majority of the parameters and hyper-parameters are identical to the ones used in this study for generating digital twins.

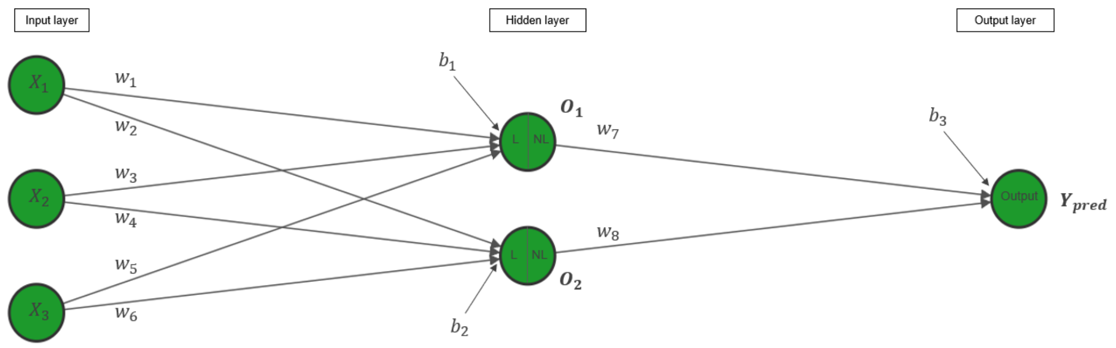


Figure 4.1: ANN for artificial device 1

This ANN consists of the following structural characteristics:

- Input layer with 3 neurons, one for each input variable x_1 & x_2 & x_3
- Hidden layer with 2 neurons. The activation function used for these neurons is sigmoid function for the sake of simplicity. PReLU is not necessary in such a low-dimensional and simple case.
- Output with 1 neuron representing output variable Y
- Cost function to minimize is mean squared error.
- learning rate η is set to 0.01.

Each neuron in the hidden layer consist of two parts, Linear and non-linear, as shown in Figure 4.2. The linear part takes sum of product of input variables, connected to this neuron from the input layer, and corresponding weights along with addition of any associated biases, as shown in Equation 4.2.

$$Z_i = x_1 w_1 \ + \ x_2 w_2 \ + \ x_3 w_3 \ + ... \ + \ b_i \tag{4.2}$$

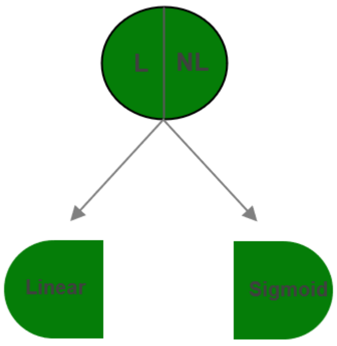


Figure 4.2: Hidden layer's neurons parts

The non-linear part is the activation function which in this case is sigmoid function as shown in 4.3 that takes in the output of linear part as its input, as shown in Equation 4.3. The output of the non-linear part serves as an input to the output layer's neuron or to the succeeding hidden layer in case of multi hidden layers configuration.

$$O_i = \frac{1}{1 \ + \ e^{-Z_i}} \tag{4.3}$$

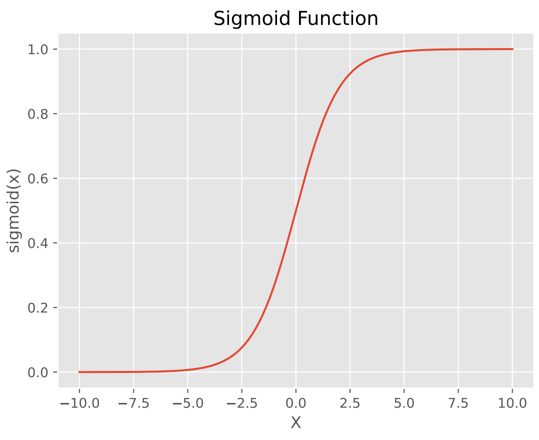


Figure 4.3: Sigmoid function

4.2 ML model training

Training of an artificial neural network involves two main stages, Forward and backward propagation. The forward propagation step is responsible for generating outputs while the backward propagation step is utilized to update weights and biases in order to improve ML models' performance. To properly explain the training process, the last row of Table 4.1 was used, also shown in Table 4.2

x_1	x_2	x_3	Y
4.10	1.10	1.21	3.73

Table 4.2: Synthetic data point

4.2.1 Forward propagation

Training of a neural network starts with forward propagation, where the ML model assigns random starting weights and biases to all the connections, and based on those values it predicts the output. For detailed explanation purposes, initial weights and biases selected are shown in Figure 4.4.

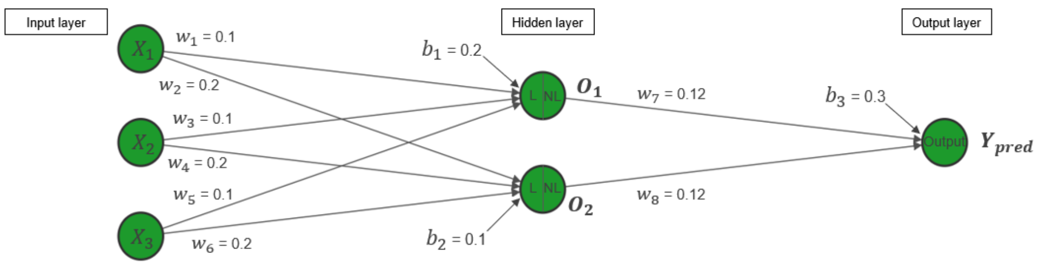


Figure 4.4: ANN with initial weights and biases

First step includes moving from input layer to hidden layer and calculating outputs O_1 & O_2. These outputs are calculated in two steps, as explained previously.

- Linear part: Calculating sum of products of features with their weights and sum of associated biases.
- Non-linear part: Calculating the output of activation function.

Output O_1

O_1 = Output of 1st neuron of hidden layer, as shown in Figure 4.5

Z_1 : output of linear part

$Z_1 = x_1w_1 \ + \ x_2w_3 \ + \ x_3w_5 \ + \ b_1$

$Z_1 = (4.1 \times 0.1) \ + \ (1.10 \times 0.1) \ + (1.21 \times 0.1) \ + \ 0.2 \ = \ 0.841$

$O_1 = \frac{1}{1 \ + \ e^{-Z_1}} = 0.698$

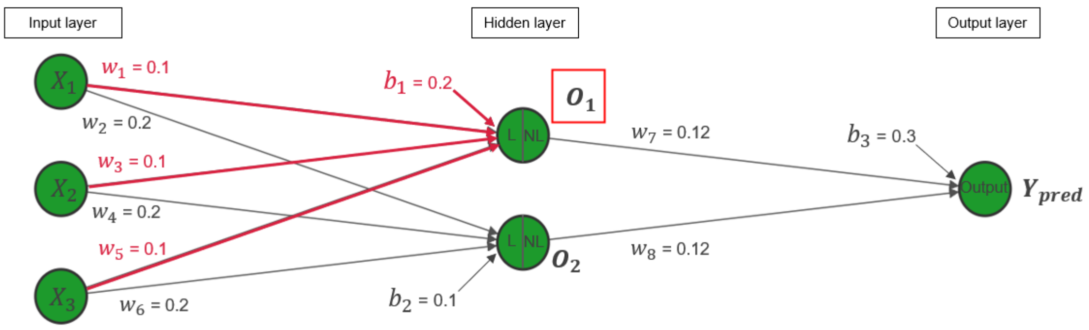


Figure 4.5: Output O_1

Output O_2

O_2 = Output of 2$^n d$ neuron of hidden layer, as shown in Figure 4.6

Z_2 : output of linear part

$Z_2 = x_1w_2 \ + \ x_2w_4 \ + \ x_3w_6 \ + \ b_2$

$Z_2 = (4.1 \times 0.2) \ + \ (1.10 \times 0.2) \ + (1.21 \times 0.2) \ + \ 0.1 \ = \ 1.382$

$O_2 = \frac{1}{1 \ + \ e^{-Z_2}} = 0.799$

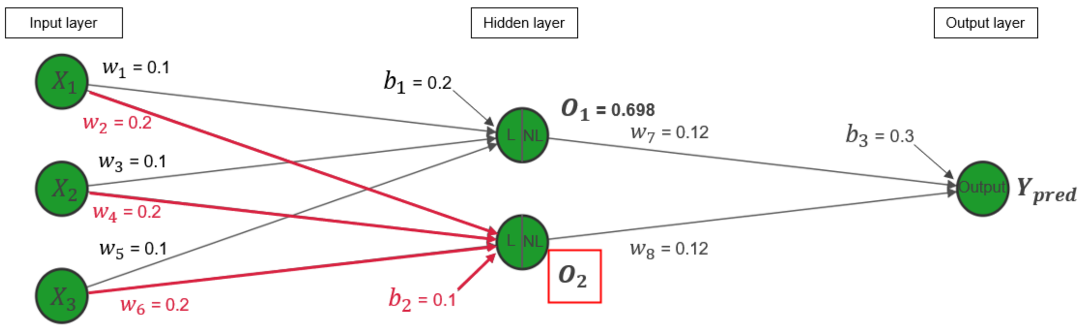


Figure 4.6: Output O_2

Final output Y_{pred}

Y_{pred} = Final output from output layer, as shown in Figure 4.7

$Y_{pred} = O_1 w_7 \ + \ O_2 w_8 \ + \ b_3$

$Y_{pred} = (0.698 \times 0.12) \ + \ (0.799 \times 0.12) \ + \ 0.3 \ = \ 0.48$

Cost function: MSE $= (Y_{pred} \ - \ Y_{true})^2 = (0.48 - \ 3.73)^2 = 10.56$

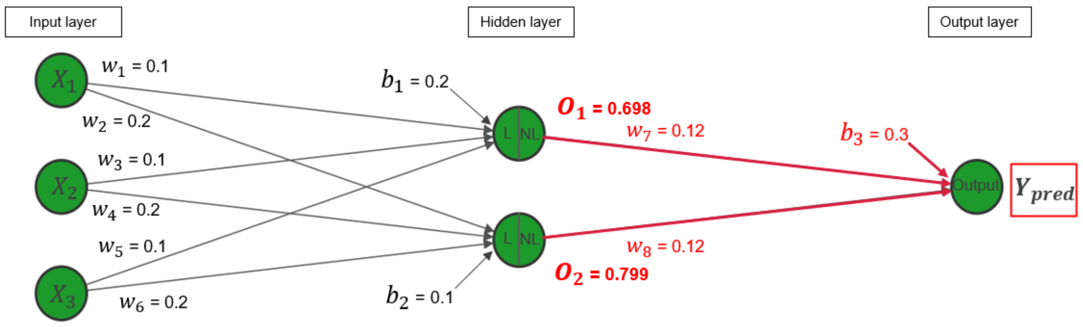


Figure 4.7: Final output Y_{pred}

The cost function value from the first iteration will almost always be large which is justifiable as the ML model has not been trained to adjust weights and biases. This adjustment takes place in the second part of the training process, which is backward propagation.

4.2.2 Backward propagation

As mentioned before, backward propagation is a crucial part of ANN training process. Weights and biases are adjusted in this training step based on the model's performance (cost function). In the first step of training, that is forward propagation, training starts from the input layer and reaches the output layer after which the output variable is predicted by the model. In the second step that is backward propagation, the direction is reversed. Weight and bias updating starts from the output layer and eventually reaches the weights connected to the input layer.
The objective is to perform gradient descent [Nielsen, 2015a] and move along the negative direction of the slope of cost function, until a minimal value is found or for the number of predefined training steps (epochs). The amount of update for each of the weights and biases is controlled by the defined learning rate ($\eta = 0.01$). Partial derivatives are calculated w.r.t the cost function to update weights and biases to improve models' performance.

Bias b_3

Starting from the output layer, first to update is bias b_3, as shown in Figure 4.8

$$b_3^{'} = b_3 + \eta \frac{\partial Costfunction}{\partial b_3}$$

$$\frac{\partial Costfunction}{\partial b_3} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial b_3}$$

$$\frac{\partial Costfunction}{\partial b_3} = 2(Y_{pred} - Y_{true}) \times 1 = 2(0.48 \times 3.73)^2 = 21.12$$

$$b_3^{'} = b_3 + \eta \frac{\partial Costfunction}{\partial b_3} = 0.3 + (0.01 \times 21.12) = 0.511$$

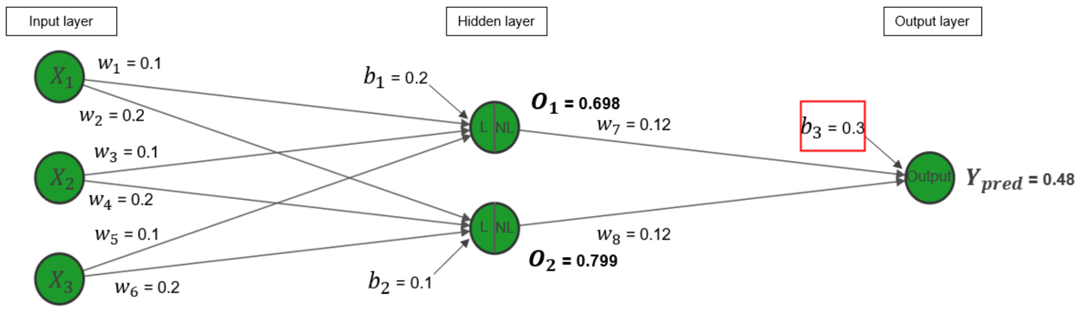


Figure 4.8: Bias b_3

Weight w_8

Next update is for weight w_8, as shown in Figure 4.9

$$w_8^{'} = w_8 + \eta \frac{\partial Costfunction}{\partial w_8}$$

$$\frac{\partial Costfunction}{\partial w_8} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial w_8}$$

$$\frac{\partial Costfunction}{\partial w_8} = 2(Y_{pred} - Y_{true}) \times O_2 = 2(0.48 \times 3.73)^2 \times 0.799 = 16.88$$

$$w_8^{'} = w_8 + \eta \frac{\partial Costfunction}{\partial w_8} = 0.12 + (0.01 \times 16.88) = 0.289$$

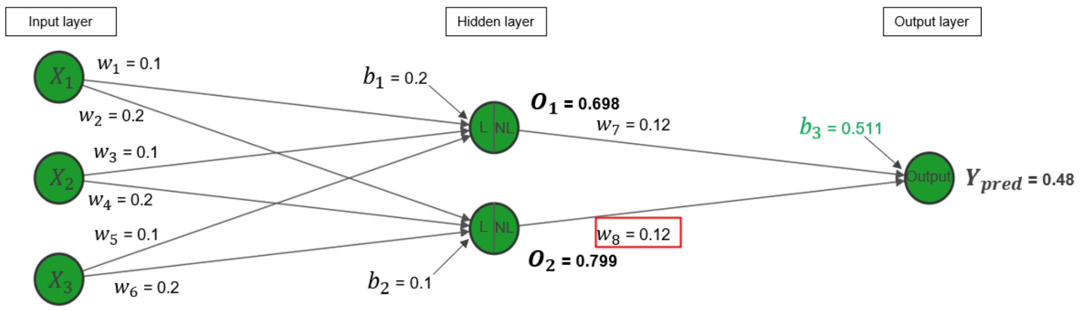


Figure 4.9: Weight w_8

Weight w_7

Next update is for weight w_7, as shown in Figure 4.10

$$w_7^{'} = w_7 + \eta\frac{\partial Costfunction}{\partial w_7}$$

$$\frac{\partial Costfunction}{\partial w_7} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial w_7}$$

$$\frac{\partial Costfunction}{\partial w_7} = 2(Y_{pred} - Y_{true}) \times O_1 = 2(0.48 \times 3.73)^2 \times 0.698 = 14.74$$

$$w_7^{'} = w_7 + \eta\frac{\partial Costfunction}{\partial w_7} = 0.12 + (0.01 \times 14.74) = 0.267$$

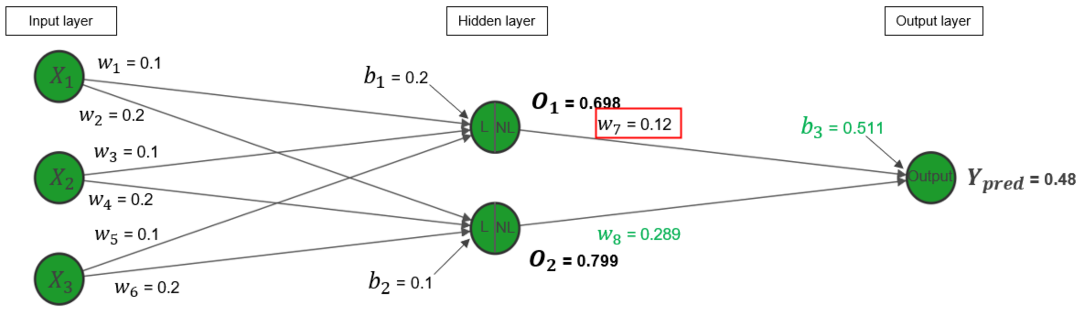


Figure 4.10: Weight w_7

Bias b_2

Next update is for bias b_2, In order to update weights and biases before the hidden layer, It requires backtracking from the output layer, as shown in Figure 4.11

$$b_2^{'} = b_2 + \eta\frac{\partial Costfunction}{\partial b_2}$$

$$\frac{\partial Costfunction}{\partial b_2} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial O_2} \times \frac{\partial O_2}{\partial Z_2} \times \frac{\partial Z_2}{\partial b_2}$$

$$\frac{\partial Costfunction}{\partial b_2} = 2(Y_{pred} - Y_{true}) \times w_8 \times O_2(1 - O_2) \times 1 = 2(0.48 \times 3.73)^2 \times 0.12 \times 0.799(1 - 0.799) \times 1 = 0.407$$

$$b_2^{'} = b_2 + \eta\frac{\partial Costfunction}{\partial b_2} = 0.1 + (0.01 \times 0.407) = 0.104$$

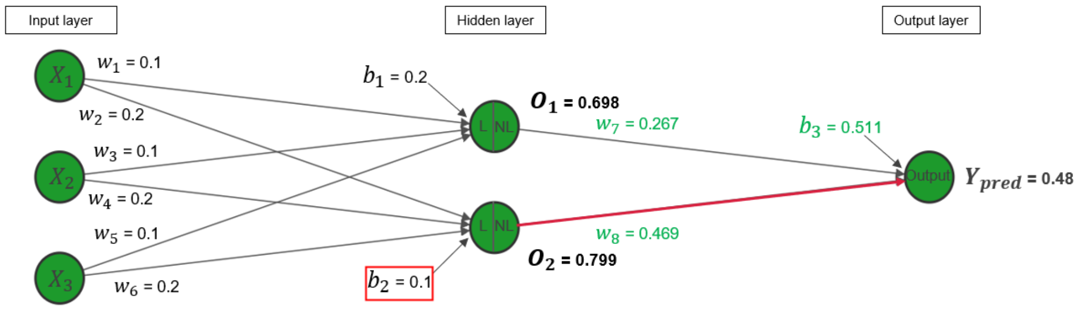


Figure 4.11: Bias b_2

Bias b_1

Next update is for bias b_1, as shown in Figure 4.12

$b_1' = b_1 + \eta \frac{\partial Costfunction}{\partial b_1}$

$$\frac{\partial Costfunction}{\partial b_1} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial O_1} \times \frac{\partial O_1}{\partial Z_1} \times \frac{\partial Z_1}{\partial b_1}$$

$\frac{\partial Costfunction}{\partial b_1} = 2(Y_{pred} - Y_{true}) \times w_7 \times O_1(1 - O_1) \times 1 = 2(0.48 \times 3.73)^2 \times 0.12 \times 0.689(1 - 0.689) \times 1 = 0.534$

$b_1' = b_1 + \eta \frac{\partial Costfunction}{\partial b_1} = 0.2 + (0.01 \times 0.534) = 0.205$

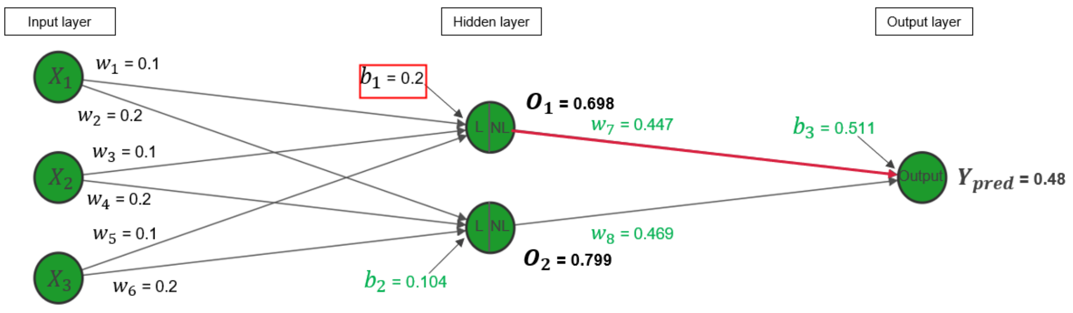


Figure 4.12: Bias b_1

Weight w_6

Next update is for weight w_6, as shown in Figure 4.13

$w_6' = w_6 + \eta \frac{\partial Costfunction}{\partial w_6}$

$$\frac{\partial Costfunction}{\partial w_6} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial O_2} \times \frac{\partial O_2}{\partial Z_2} \times \frac{\partial Z_2}{\partial w_6}$$

$\frac{\partial Costfunction}{\partial w_6} = 2(Y_{pred} - Y_{true}) \times w_8 \times O_2(1 - O_2) \times X_3 = 2(0.48 \times 3.73)^2 \times 0.12 \times 0.799(1 - 0.799) \times 1.21 = 0.493$

$w_6' = w_6 + \eta \frac{\partial Costfunction}{\partial w_6} = 0.2 + (0.01 \times 0.493) = 0.205$

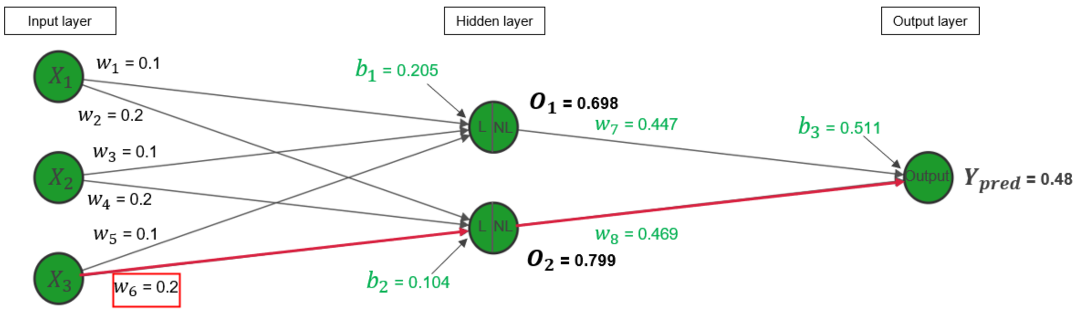


Figure 4.13: Weight w_6

Weight w_5

Next update is for weight w_5, as shown in Figure 4.14
$w_5^{'} = w_5 + \eta \frac{\partial Costfunction}{\partial w_5}$

$$\frac{\partial Costfunction}{\partial w_5} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial O_1} \times \frac{\partial O_1}{\partial Z_1} \times \frac{\partial Z_1}{\partial w_5}$$

$\frac{\partial Costfunction}{\partial w_5} = 2(Y_{pred} - Y_{true}) \times w_7 \times O_1(1 - O_1) \times X_3 = 2(0.48 \times 3.73)^2 \times 0.12 \times 0.698(1 - 0.698) \times 1.21 = 0.646$

$$w_5^{'} = w_5 + \eta \frac{\partial Costfunction}{\partial w_5} = 0.1 + (0.01 \times 0.646) = 0.106$$

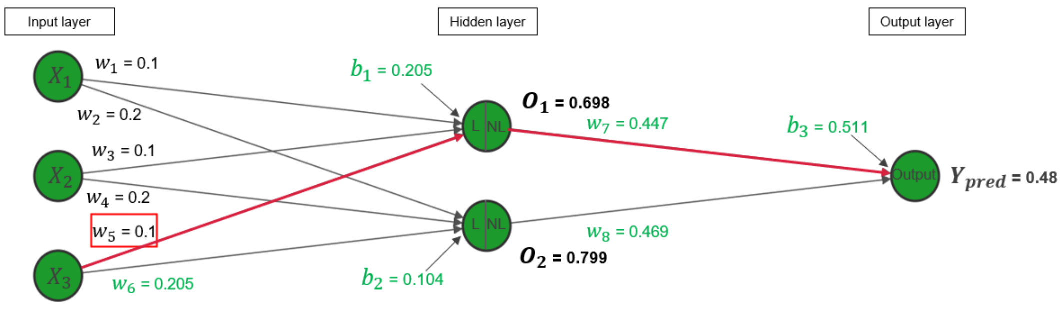


Figure 4.14: Weight w_5

Weight w_4

Next update is for weight w_4, as shown in Figure 4.15
$w_4^{'} = w_4 + \eta \frac{\partial Costfunction}{\partial w_4}$

$$\frac{\partial Costfunction}{\partial w_4} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial O_2} \times \frac{\partial O_2}{\partial Z_2} \times \frac{\partial Z_2}{\partial w_4}$$

$\frac{\partial Costfunction}{\partial w_4} = 2(Y_{pred} - Y_{true}) \times w_8 \times O_2(1 - O_2) \times X_2 = 2(0.48 \times 3.73)^2 \times 0.12 \times 0.799(1 - 0.799) \times 1.10 = 0.448$

$$w_4^{'} = w_4 + \eta \frac{\partial Costfunction}{\partial w_4} = 0.2 + (0.01 \times 0.448) = 0.204$$

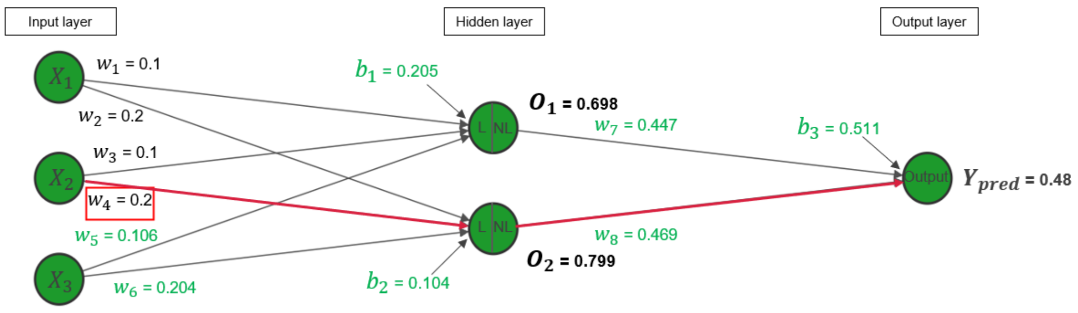


Figure 4.15: Weight w_4

Weight w_3

Next update is for weight w_3, as shown in Figure 4.16

$w_3' = w_3 + \eta \frac{\partial Costfunction}{\partial w_3}$

$$\frac{\partial Costfunction}{\partial w_3} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial O_1} \times \frac{\partial O_1}{\partial Z_1} \times \frac{\partial Z_1}{\partial w_3}$$

$\frac{\partial Costfunction}{\partial w_3} = 2(Y_{pred} - Y_{true}) \times w_7 \times O_1(1 - O_1) \times X_2 = 2(0.48 \times 3.73)^2 \times 0.12 \times 0.698(1 - 0.698) \times 1.10 = 0.588$

$$w_3' = w_3 + \eta \frac{\partial Costfunction}{\partial w_3} = 0.1 + (0.01 \times 0.588) = 0.106$$

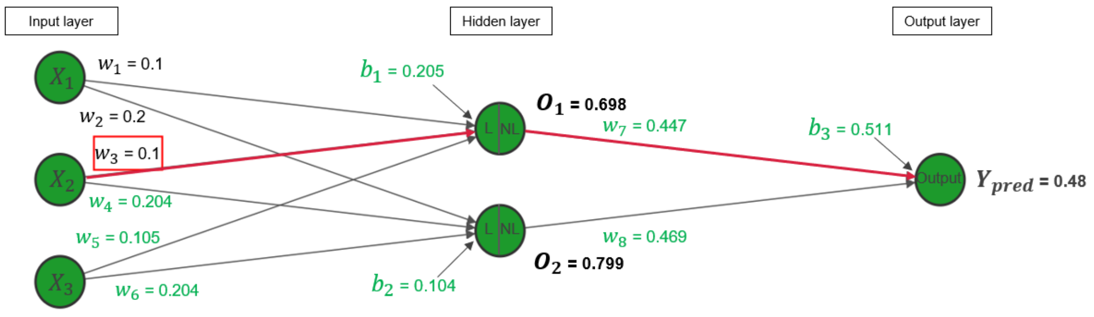


Figure 4.16: Weight w_3

Weight w_2

Next update is for weight w_2, as shown in Figure 4.17

$w_2' = w_2 + \eta \frac{\partial Costfunction}{\partial w_2}$

$$\frac{\partial Costfunction}{\partial w_2} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial O_2} \times \frac{\partial O_2}{\partial Z_2} \times \frac{\partial Z_2}{\partial w_2}$$

$\frac{\partial Costfunction}{\partial w_2} = 2(Y_{pred} - Y_{true}) \times w_8 \times O_2(1 - O_2) \times X_1 = 2(0.48 \times 3.73)^2 \times 0.12 \times 0.799(1 - 0.799) \times 4.10 = 1.67$

$$w_2' = w_2 + \eta \frac{\partial Costfunction}{\partial w_2} = 0.2 + (0.01 \times 1.67) = 0.217$$

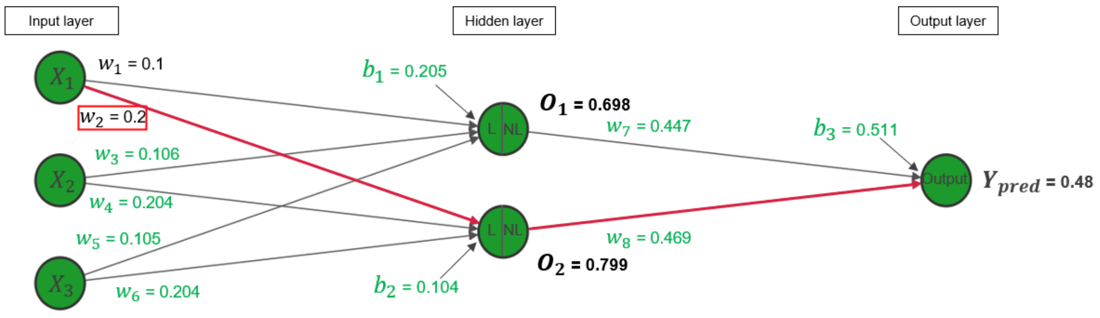


Figure 4.17: Weight w_2

Weight w_1

Next update is for weight w_1, as shown in Figure 4.18

$w_1' = w_1 + \eta \frac{\partial Costfunction}{\partial w_1}$

$$\frac{\partial Costfunction}{\partial w_1} = \frac{\partial Costfunction}{\partial Y_{pred}} \times \frac{\partial Y_{pred}}{\partial O_1} \times \frac{\partial O_1}{\partial Z_1} \times \frac{\partial Z_1}{\partial w_1}$$

$\frac{\partial Costfunction}{\partial w_1} = 2(Y_{pred} - Y_{true}) \times w_7 \times O_1(1 - O_1) \times X_1 = 2(0.48 \times 3.73)^2 \times 0.12 \times 0.698(1 - 0.698) \times 4.10 = 2.19$

$w_1' = w_1 + \eta \frac{\partial Costfunction}{\partial w_1} = 0.1 + (0.01 \times 2.19) = 0.122$

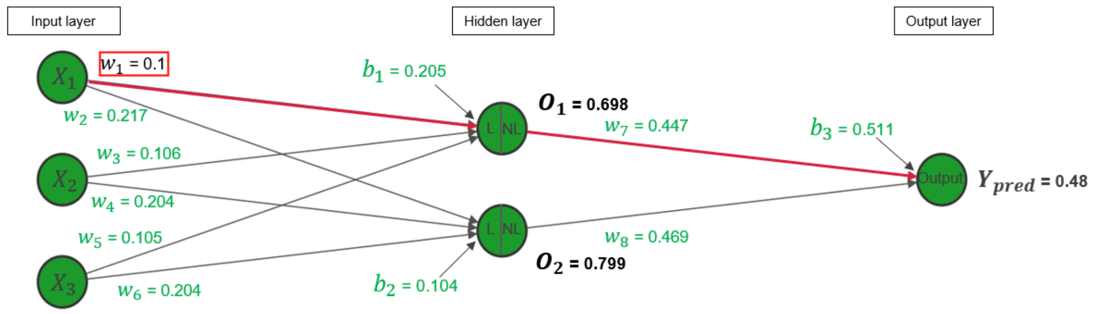


Figure 4.18: Weight w_1

Backward propagation is complete once all the weights and biases are updated. Completion of a single cycle of forward and backward propagation represents a single training step/epoch as shown in Figure 4.19. A ML model goes through multiple epochs, minimizing cost function along the way, to generate accurate results.

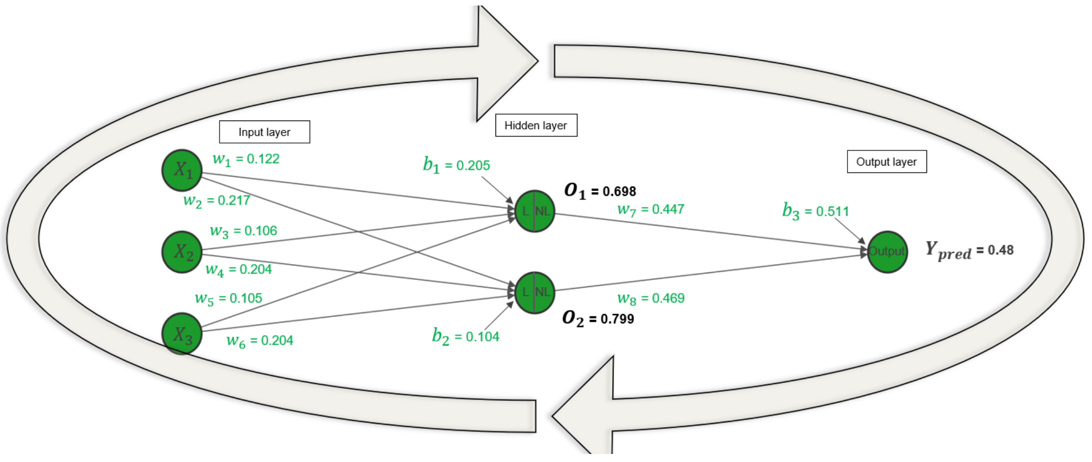


Figure 4.19: Single epoch

4.2.3 Transfer learning

A trained ML model of one semiconductor device, with updated weights and passing evaluation criteria, is used to transfer input variable information to an untrained ML model of another yet similar device. An optimizer, explained in Section 3.2.4 is used to minimize the amount of training data. Again for the sake of confidentially, simulated data is utilized from a quadratic equation with an additional input variable (x_4) as compared to Equation 4.1, as shown in Equation 4.4 and Figure 4.20. This can be considered as artificial device 2.

$$Y = 0.5 \times x_1 \ \times \ x_2^2 \ \times \ x_3^{x_2} \ + \ x_4 \tag{4.4}$$

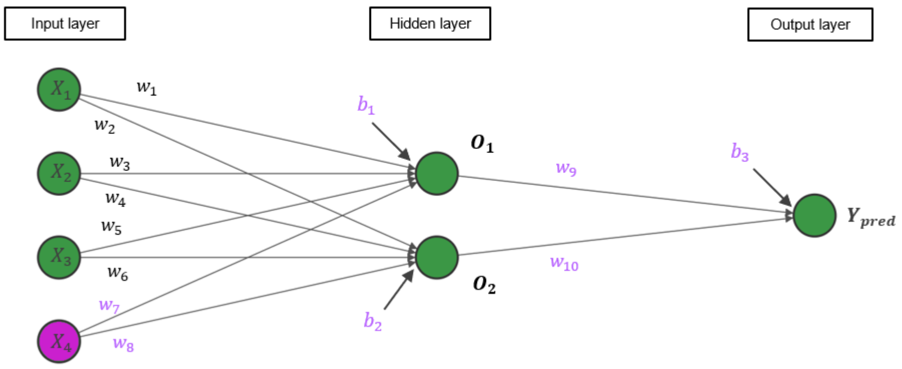


Figure 4.20: ANN for artificial device 2

Selecting base model

It is important to select a base ML model which is efficiently trained with good performance and similar input and output variables, in this case, artificial device 1. For the sake of simplicity, the device 1 ML model trained with just one epoch, as shown previously, is considered as a highly trained model ready for transferring information.

Transferring feature weights

There are three input variables x_1 , x_2 & x_3 that are common between both artificial devices 1 and 2. Transfer learning can be used to transfer learned weight vectors of these features from a base model to a new model of artificial device 2, as shown in Figures 4.21, 4.22 and 4.23.

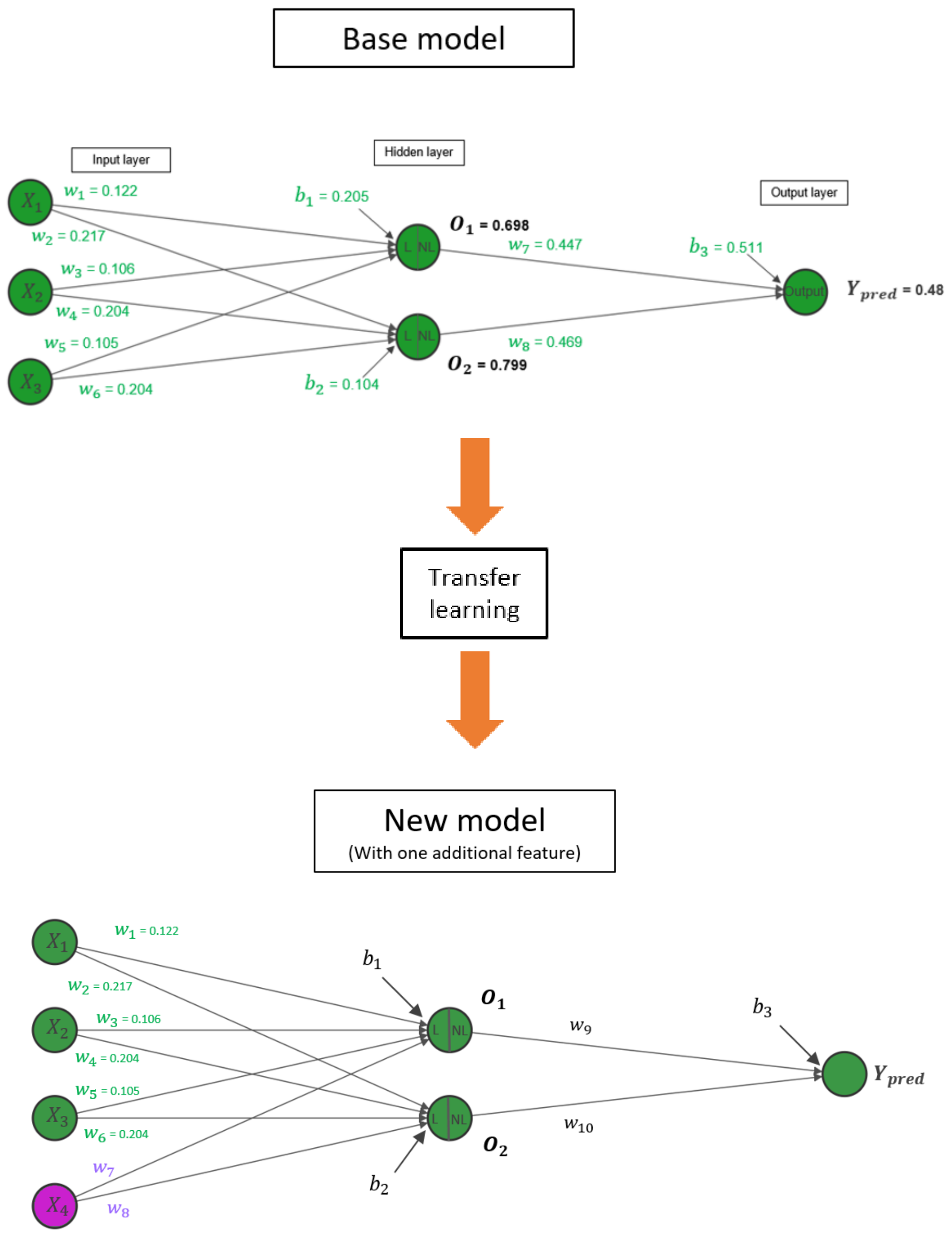


Figure 4.21: Transfer learning

$$\boxed{\text{H.L (linear part)} = \boxed{(W^T * \ Input\ layer)} + bias}$$

Hidden Layer | Weight Matrix (Transposed) | Input layer

$$\begin{pmatrix} O_1 \\ O_2 \end{pmatrix} = \begin{pmatrix} w_1 & w_3 & w_5 & w_7 \\ w_2 & w_4 & w_6 & w_8 \end{pmatrix} \times \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{pmatrix}$$

Figure 4.22: Weight matrix

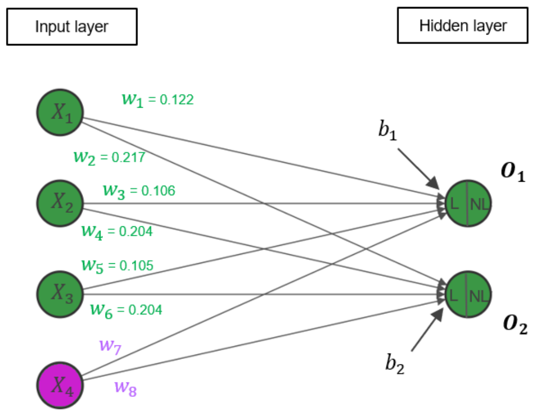


Figure 4.23: Transferred weights

Training for new input variable

After weight vectors of common features are transferred, partial training is implemented to update weight vectors of input variables unique to artificial model 2. This is done by freezing weight vectors of common features denoting that they are not updated during the training process. The weight vectors of new features, on the other hand, are updated with respect to the transferred weight vectors of common features. Weights and biases that are updated in the partial training, in this case, are highlighted in Figure 4.24

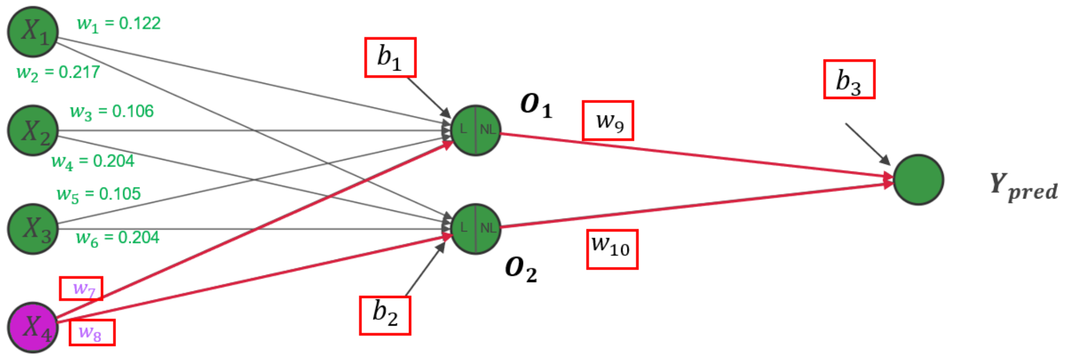


Figure 4.24: Updating new weights

Fine-tuning

Once the new feature weights are trained, the entire model is re-trained with a low learning rate ($\eta = 0.001$) to improve the new model's performance and ensure compatibility w.r.t to the dataset of artificial device 2. A low learning rate is used to re-calibrate the transferred knowledge from device 1 to fit the inherent dependencies of device 2 and to avoid overfitting, as the main objective of transfer learning is to reduce training data, so in practice, the dataset available for the second device will always be smaller than the first one and training with the same complexity as for first device would introduce overfitting.

Theory vs practical implementation

A major part of the methodology explained in this chapter was also implemented in practice. The only difference was in Section 4.2.3 'training for new features'. In theory, it was mentioned that transferred weights vectors of common features/input variables between the two devices were frozen while the weight vectors for features/input variables unique to device 2 were updated. This was not possible in practice based on what was found in the conducted literature review. Due to the limitation of methodologies provided by available ML-related libraries which mainly allow freezing of entire layers, it was not possible to partially freeze neurons within a single layer. Modifying the code base of the existing ML libraries could provide that capability, but this was not practical enough for real-world applications where the code base of a library should stay untouched as the custom changes may not be publically available. This however was overcome in practice by allowing the whole ML model to be trained, including the transferred weights, and once the training was completed, weight vectors of common input variables were reset to their initial transferred values while at the same time conserving updated weight vectors for input variables unique to the second device. This can be seen in Figure 4.25, 4.26 & 4.27, After which, fine-tuning was implemented as described previously. The results obtained at the end of this implementation were quite substantial, which are shown in Chapter 5.

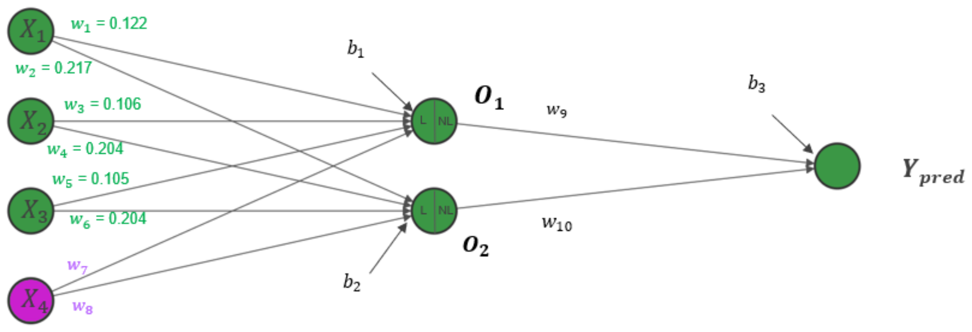


Figure 4.25: ANN pre-training

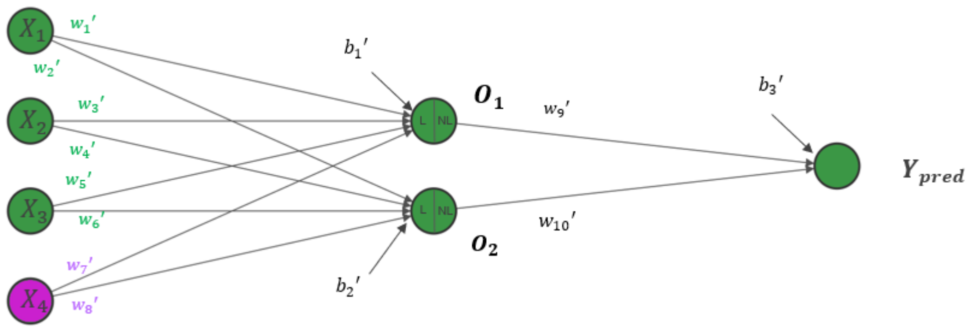


Figure 4.26: ANN post-training

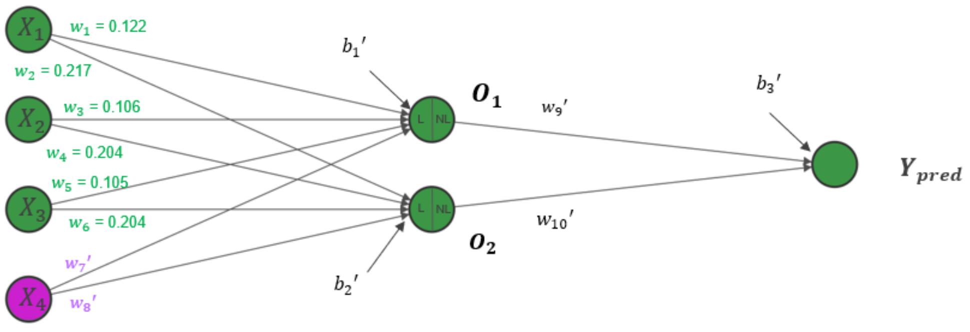


Figure 4.27: ANN resetting transferred weights

where,

w_i = pre-training weights

w_i' = post-training weights

Chapter 5

Results

Real-world application of transfer learning showcased in Chapter 4 was implemented for two different semiconductor devices named positive field-effect transistor (PFET) and negative field-effect transistor (NFET), having multiple variants. The PFET device had 4 different variants named super low threshold voltage (SLVT), low threshold voltage (LVT), regular voltage (RVT), and high voltage (HVT). Transfer learning was implemented for all possible pairs of these variants leading to 12 test cases. 3 variants, LVT, RVT, and HVT, were considered for device NFET, leading to 6 test cases for transfer learning from it. In total, 18 different test cases for transfer learning were considered.
The results are shown in this chapter in a tabular form with rows representing different training techniques along with the average value of the evaluation score of all the outputs, shown in Section 3.2.3. It is to be noted that the average evaluation score is shown instead of showing the evaluation score for all output variables only for a clearer representation of the difference in performance with different training techniques. In practice, the evaluation criteria is tested for each output variable as discussed in Section 3.2.3. The first row showcases the results of an ML model trained with the standard training procedure (without transfer learning). The rest of the rows showcase the effect of transfer learning with different base models. Out of these, the one utilizing the least amount of training data is selected as the final model. The least amount of training data required to pass the evaluation criteria for these training techniques is selected by the optimizer explained in Section 3.2.4. The difference in the amount of training data required to pass the evaluation criteria for each of these variations of transfer learning is based on the similarity between the device variants. The variants with the highest resemblance require the least amount of training data during transfer learning, to pass the evaluation criteria.

5.1 PFET device

As mentioned before, 12 different test cases were considered for the PFET device. Results for all these test cases are displayed below.

5.1.1 SLVT

Train type	Training data	Average evaluation score ($99^{th} percentile$)
Standard	730	0.335
Transfer learning LVT to SLVT	300	0.337
Transfer learning RVT to SLVT	340	0.34
Transfer learning HVT to SLVT	**160**	**0.33**

Table 5.1: PFET SLVT transfer learning results

It can be seen from Table 5.1 that the best result (least amount of training data) was achieved using transfer learning with the HVT device as a base model. The amount of training data was reduced by almost 78 % while maintaining the ML model's performance (passing the evaluation criteria).

5.1.2 LVT

Train type	Training data	Average evaluation score ($99^{th} percentile$)
Standard	800	0.422
Transfer learning SLVT to LVT	340	0.360
Transfer learning RVT to LVT	**270**	**0.376**
Transfer learning HVT to LVT	350	0.378

Table 5.2: PFET LVT transfer learning results

It can be seen from Table 5.2 that the best result (least amount of training data) was achieved using transfer learning with the RVT device as a base model. The amount of training data was reduced by almost 66 % while maintaining the ML model's performance (passing the evaluation criteria).

5.1.3 RVT

Train type	Training data	Average evaluation score ($99^{th} percentile$)
Standard	790	0.338
Transfer learning SLVT to RVT	230	0.354
Transfer learning LVT to RVT	240	0.326
Transfer learning HVT to RVT	**210**	**0.366**

Table 5.3: PFET RVT transfer learning results

It can be seen from Table 5.3 that the best result (least amount of training data) was achieved using transfer learning with the HVT device as a base model. The amount of training data was reduced by almost 73 % while maintaining the ML model's performance (passing the evaluation criteria).

5.1.4 HVT

Train type	Training data	Average evaluation score ($99^{th} percentile$)
Standard	660	0.358
Transfer learning SLVT to HVT	330	0.392
Transfer learning LVT to HVT	**240**	**0.391**
Transfer learning RVT to HVT	350	0.401

Table 5.4: PFET HVT transfer learning results

It can be seen from Table 5.4 that the best result (least amount of training data) was achieved using transfer learning with the LVT device as a base model. The amount of training data was reduced by almost 64 % while maintaining the ML model's performance (passing the evaluation criteria).

5.2 NFET device

As mentioned before, 6 different test cases were considered for the NFET device. Results for all these test cases are displayed below.

5.2.1 LVT

Train type	Training data	Average evaluation score ($99^{th} percentile$)
Standard	580	0.253
Transfer learning RVT to LVT	220	0.254
Transfer learning HVT to LVT	**150**	**0.262**

Table 5.5: NFET LVT transfer learning results

It can be seen from Table 5.5 that the best result (least amount of training data) was achieved using transfer learning with the HVT device as a base model. The amount of training data was reduced by almost 74 % while maintaining the ML model's performance (passing the evaluation criteria).

5.2.2 RVT

Train type	Training data	Average evaluation score ($99^{th} percentile$)
Standard	520	0.208
Transfer learning LVT to RVT	240	0.226
Transfer learning HVT to RVT	**280**	**0.235**

Table 5.6: NFET RVT transfer learning results

It can be seen from Table 5.6 that the best result (least amount of training data) was achieved using transfer learning with the HVT device as a base model. The amount of training data was reduced by almost 46 % while maintaining the ML model's performance (passing the evaluation criteria).

5.2.3 HVT

Train type	Training data	Average evaluation score ($99^{th} percentile$)
Standard	800	0.257
Transfer learning LVT to HVT	**280**	**0.263**
Transfer learning RVT to HVT	300	0.266

Table 5.7: NFET HVT transfer learning results

It can be seen from Table 5.7 that the best result (least amount of training data) was achieved using transfer learning with the LVT device as a base model. The amount of training data was reduced by almost 65 % while maintaining the ML model's performance (passing the evaluation criteria).

Further details about the results are mentioned in Chapter 6 and in Appendix A.

Chapter 6

Conclusion

6.1 Study conclusion

It can be seen in Chapter 5 from the results of all test cases that transfer learning reduced the amount of training data required by a substantial amount, between 46-78 % while maintaining if not slightly improving device performance. This can also be seen in the bar plots shown in Figure 6.1 for the PFET device and in Figure 6.2 for the NFET device.

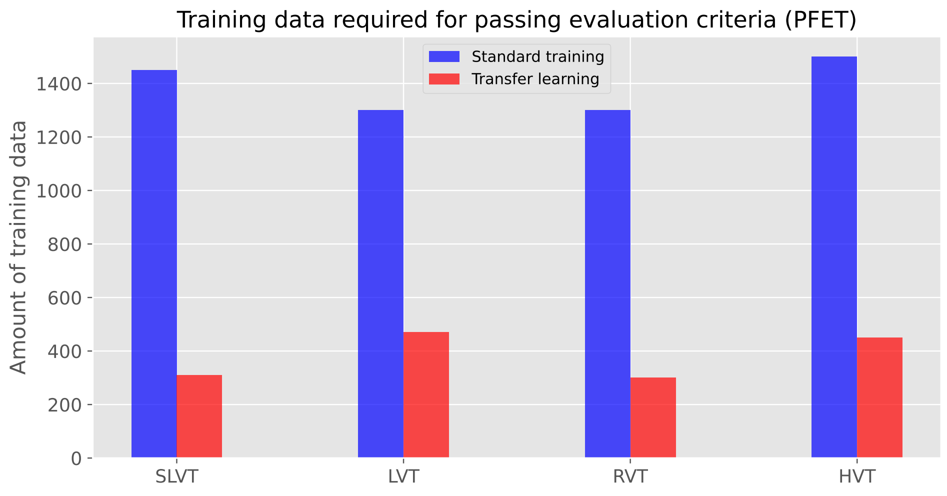


Figure 6.1: Transfer learning improvement PFET

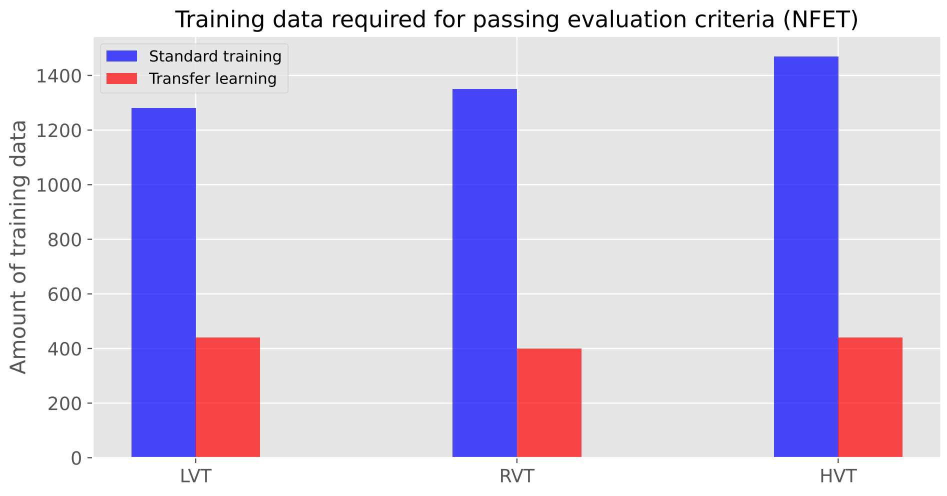


Figure 6.2: Transfer learning improvement NFET

One thing to note from the results shown in Chapter 5 is that the average evaluation score will never be close to 1 which would ideally be the representation of a ML model barely passing the evaluation criteria with minimum training data. The reason behind this is that some electrical parameters fail the evaluation criteria much earlier than the rest. So the average score does not accurately depict that rather it simply shows the overall status of the model's performance when comparing the standard training method to the transfer learning approach. As mentioned in Chapter 5, the average evaluation score is considered only for a clearer representation of results from different training techniques.

6.2 Limitations

As mentioned in Chapter 4, the only difference between theory vs practical implementation was the methodology of transfer learning. In theory, transferred weight vectors for common input variables between two device variants were frozen while the input variables unique to the second device were allowed to be updated during the training process. This, however in practice, was not possible based on the conducted research as mentioned in Section 4.2.3. An alternative approach that was adopted was to let all the weight vectors get updated during the training process and then manually resetting the weight vectors of common input variables back to their transferred values. This led to achieving great results which can be seen in Chapter 5 and Section 6.1.

6.3 Recommendations/ Further steps

6.3.1 Partial freezing of hidden layers

As mentioned earlier in Section 6.2, Currently, partial freezing of hidden layers was not possible based on the conducted research. However, this can be possibly overcome by making custom changes to ML libraries and adding self-made functions allowing partial freezing. However, this can lead to limitations in terms of utilizing the transfer learning methodology as it is not an ideal approach for real-world problems. To be more specific, it can lead to ML model deployment issues in terms of compatibility with available services.

6.3.2 Wider scope of transfer learning

In this study, Transfer learning was mainly conducted within the environment of a single semiconductor device, that is, between one device variant to another. This was limited mainly due to time and resource constraints. However, it is also possible to conduct transfer learning between two separate devices and not just in between their variants. The amount of improvement achieved from transfer learning, in that case, will depend upon the similarity between the two devices, and the number of input and electrical output variables parameters they share.

6.3.3 Targeted training

It was noticed from the results that some electrical output variables predicted by the ML model, based on the evaluation criteria mentioned in Section 3.2.3, were failing the evaluation criteria much earlier than the rest, in other words, some output variables, had evaluation score greater than 1 while the rest of the output variables had scores very close to 0 which would lead to overall evaluation failure. This can be further worked on by specifically training ML models for these electrical parameters or by assigning higher training weights to them during the training process. This approach would rely on repeating the training process for these electrical parameters rather than increasing the training data, which will lead to higher performance by the ML model and an even lower requirement of training data.

Appendix A

Transfer learning results detail

A.1 PFET device

Evaluation scores and threshold values are shown for important electrical output variables in this appendix. The purpose of this appendix is to share further details about the evaluation procedure that was used while conducting transfer learning. Tables A.1, A.2, A.3, and A.4 display evaluation scores, based on Section 3.2.3, for important electrical output variables for PFET semiconductor device variants. The average of these scores along with the evaluation scores of the rest of the output variables makes up the average evaluation score shown multiple times in Chapter 5.

A.1.1 SLVT

Best model achieved by transfer learning from HVT to SLVT

Training data utilized: 160

Average evaluation score (99^{th} *percentile*): 0.33

Output variable	Output threshold value	Output abs value	Evaluation score
Saturated drain current (Idsat)	20	15.94	0.797
Overdrive saturated drain current (Iodsat)	20	0.0114	0.00057
Linear drain current (Idlin)	10	4.794	0.479
Overdrive linear drain current (Iodlin)	10	$4.62\ e^{-6}$	$4.62\ e^{-7}$
Effective drain current (Ideff)	15	13.13	0.875
Drain cutoff current (Idoff)	10	0.078	0.0078

Output variable	Output threshold value	Output abs value	Evaluation score
Saturation threshold voltage (Vtsat)	0.008	0.0055	0.697
Linear threshold voltage (Vtlin)	0.008	0.0035	0.4375
Drain-induced barrier lowering (DIBL)	4	0.0157	0.039

Table A.1: PFET SLVT extended results

A.1.2 LVT

Best model achieved by transfer learning from RVT to LVT

Training data utilized: 270

Average evaluation score (99th *percentile*): 0.376

Output variable	Output threshold value	Output abs value	Evaluation score
Saturated drain current (Idsat)	20	15.5	0.775
Overdrive saturated drain current (Iodsat)	20	0.014	0.00071
Linear drain current (Idlin)	10	5.11	0.511
Overdrive linear drain current (Iodlin)	10	4.65 e^{-6}	4.65 e^{-7}
Effective drain current (Ideff)	15	11.31	0.75
Drain cutoff current (Idoff)	10	0.145	0.0145
Saturation threshold voltage (Vtsat)	0.008	0.0073	0.92
Linear threshold voltage (Vtlin)	0.008	0.0062	0.775
Drain-induced barrier lowering (DIBL)	4	0.0157	0.105

Table A.2: PFET LVT extended results

A.1.3 RVT

Best model achieved by transfer learning from HVT to RVT

Training data utilized: 210

Average evaluation score (99th *percentile*): 0.366

Output variable	Output threshold value	Output abs value	Evaluation score
Saturated drain current (Idsat)	20	18.98	0.949
Overdrive saturated drain current (Iodsat)	20	0.0157	0.000785
Linear drain current (Idlin)	10	5.4	0.54
Overdrive linear drain current (Iodlin)	10	5.63 e^{-6}	5.63 e^{-7}
Effective drain current (Ideff)	15	14.84	0.989
Drain cutoff current (Idoff)	10	0.089	0.089
Saturation threshold voltage (Vtsat)	0.008	0.0056	0.7
Linear threshold voltage (Vtlin)	0.008	0.0037	0.465
Drain-induced barrier lowering (DIBL)	4	0.0156	0.004

Table A.3: PFET RVT extended results

A.1.4 HVT

Best model achieved by transfer learning from LVT to HVT

Training data utilized: 240

Average evaluation score (99th *percentile*): 0.931

Output variable	Output threshold value	Output abs value	Evaluation score
Saturated drain current (Idsat)	20	18.58	0.929
Overdrive saturated drain current (Iodsat)	20	0.0175	0.000877
Linear drain current (Idlin)	10	5.366	0.536
Overdrive linear drain current (Iodlin)	10	$5.11\ e^{-6}$	$5.11\ e^{-7}$
Effective drain current (Ideff)	15	12.71	0.847
Drain cutoff current (Idoff)	10	0.129	0.0129
Saturation threshold voltage (Vtsat)	0.008	0.0056	0.7
Linear threshold voltage (Vtlin)	0.008	0.00747	0.935
Drain-induced barrier lowering (DIBL)	4	0.0173	0.0043

Table A.4: PFET HVT extended results

A.2 NFET device

The same evaluation procedure was used for conducting transfer learning for NFET device variants.

Bibliography

[Anqi Mao, 2023] Anqi Mao, e. a. (2023). Cross-entropy loss functions: Theoretical analysis and applications. *arXiv preprint arXiv:2304.07288.*

[Bing Xu, 2015] Bing Xu, e. a. (2015). Empirical evaluation of rectified activations in convolutional network.

[Dyd Pradeep, 2023] Dyd Pradeep, Bitragunta Vivek Vardhan, e. a. (2023). Optimal predictive maintenance technique for manufacturing semiconductors using machine learning. *3rd International Conference on Intelligent Communication and Computational Techniques (ICCT), IEEE.*

[Jacob Devlin, 2018] Jacob Devlin, e. a. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805.*

[James Bergstra, 2011] James Bergstra, e. a. (2011). Algorithms for hyper-parameter optimization.

[James Bergstra, 2013] James Bergstra, e. a. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures.

[John Wiley & Sons, 2017] John Wiley & Sons, p.-. (2017). *Design and analysis of experiments.*

[Keras, a] Keras. Metrics. `https://www.tensorflow.org/api_docs/python/tf/keras/metrics`. Accessed: 04/20/2023.

[Keras, b] Keras. Transfer learning and fine-tuning. `https://keras.io/guides/transfer_learning/`. Accessed: 03/03/2023.

[McKay, 1979] McKay, e. a. (1979). *A comparison of three methods for selecting values of input variables in the analysis of output from a computer code, pg: 56-61.*

[Meng Zhao, 2018] Meng Zhao, J. L. (2018). Tuning the hyper-parameters of cma-es with tree-structured parzen estimators. *Tenth International Conference on Advanced Computational Intelligence (ICACI),IEEE.*

[Neophytos Lophitis, 2018] Neophytos Lophitis, Anastasios Arvanitopoulos, e. a. (2018). *TCAD Device Modelling and Simulation of Wide Bandgap Power Semiconductors.* PhD thesis, Faculty of Engineering, Environment and Computing, Coventry University, UK.

[Nguyen, 2019] Nguyen, V. (2019). Bayesian optimization for accelerating hyperparameter tuning. *IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE).*

[Nielsen, 2015a] Nielsen, M. (2015a). *Neural Networks and Deep Learning, pg:15-24.*

[Nielsen, 2015b] Nielsen, M. (2015b). *Neural Networks and Deep Learning, pg:2-12.*

[Paul Jungmann, 2023] Paul Jungmann, e. a. (2023). Tcad-enabled machine learning—an efficient framework to build highly accurate and reliable models for semiconductor technology development and fabrication. *IEEE Transactions on Semiconductor Manufacturing.*

[PyTorch, a] PyTorch. Loss functions. `https://pytorch.org/docs/stable/nn.html#loss-functions`. Accessed: 04/26/2023.

[PyTorch, b] PyTorch. Metrics. `https://pytorch.org/docs/stable/nn.html#metrics`. Accessed: 04/20/2023.

[Raphael Wagner, 2019] Raphael Wagner, e. a. (2019). Challenges and potentials of digital twins and industry 4.0 in product design and production for high performance products. *29th CIRP Design 2019, ScienceDirect.*

[Scikit-learn, a] Scikit-learn. Bayesian optimization. `https://scikit-optimize.github.io/stable/modules/generated/skopt.BayesSearchCV.html`. Accessed: 04/29/2023.

[Scikit-learn, b] Scikit-learn. Grid search. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html`. Accessed: 04/29/2023.

[Scikit-learn, c] Scikit-learn. Loss functions. `https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics`. Accessed: 04/18/2023.

[Scikit-learn, d] Scikit-learn. Randomsearch. `https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html`. Accessed: 04/29/2023.

[Scikit-optimize,] Scikit-optimize. Bayesian optimization. `https://github.com/scikit-optimize/scikit-optimize`. Accessed: 04/29/2023.

[Sinno Jialin Pan, 2009] Sinno Jialin Pan, e. a. (2009). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering.*

[Smith, 2018] Smith, L. (2018). A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay.

[Synopsys,] Synopsys. Synopsys tcad. `https://www.synopsys.com/silicon/tcad.html`. Accessed: 01/15/2023.

[TensorFlow,] TensorFlow. Transfer learning and fine-tuning. `https://www.tensorflow.org/tutorials/images/transfer_learning`. Accessed: 03/03/2023.

[Yang Li, 2021] Yang Li, J. W. (2021). A defect detection method based on improved mask r-cnn for wafer maps. *International Conference on Computer Network, Electronic and Automation (ICCNEA), IEEE.*