ECOLE POLYTECHNIQUE DE BRUXELLES

Project in physics engineering

Neural network for critical heat flux predictions

Author :

Dumont Jules

Supervisors :

Sabater Adrian, Haedens Maxime

Coordinator :

Labeau Pierre-Etienne

July 2023

Abstract

Critical heat flux prediction is a crucial element for the safety, efficiency, and longevity of nuclear power plants. The complexity of the phenomenon makes it challenging to model analytically. For this reason, the 2006 Groeneveld Lookup Table, which lists more than 25,000 experiments, is used to employ linear interpolation to make CHF predictions, resulting in a mean absolute percent error of 8%. Thanks to recent developments in machine learning algorithms and computational power, new data-driven prediction methods can be developed to increase accuracy. This project explored regression neural network algorithms and their hyperparameters optimization using an object-oriented python code. The investigations led to a robust model capable of predictions with a Mean Absolute Percentage Error of 16 percent. This project establishes a foundation for further improvement in hyperparameters optimization or the implementation of transfer learning to specific geometries.

Contents

1. Introduction

When designing a nuclear reactor cooling system, an important phenomenon that must be taken in account is the dry-out at critical heat flux (CHF). When the heat flux goes over a critical value, a stable vapor film starts forming. It then leads to a decrease in heat transfer capabilities and sudden increase in temperature of the heated surface. It can obviously lead to core damages and safety problems. It is important to note that before CHF occurs and while the coolant is boiling more as the flux increase, the efficiency of heat transfer process increase. For those reasons it is necessary to make accurate CHF predictions and modelings whether for security or economic reasons. Unfortunately, it isn't an easy task since no unifying model or theory has been found yet. The most common method is the linear interpolation of the 2006 Groeneveld Lookup table (LUT) compiling more than 25000 experiments with diverse parameters but this method possess limitations. In recent years, the increase of computational power and improvement in machine learning algorithm lead to new perspectives. Those methods can help to build new models that have the potential to improve the prediction accuracy. One of the most promising type of machine learning algorithm according to [3] is the neural network(NN) and is the method explored in this project. The objective is to realise a python code that implement an adapted NN for CHF predictions using *TensorFlow* library. The code focus on 3 aspects. First it focus on the data that need to be extract filtered and format to be used by the NN. The second emphasize the structure and hyperparameters chosen to set the NN. The last part center on hyperparameters optimization.

The remainder of this paper is organized as followed: section 2 provides details about CHF theoretical aspects the LUT, section 3 introduces the background of NN algorithm, section 4 the code implementation , section 5 a discussion over the results, section 6 the concept of transfer learning and a conclusion 7 summarizing the key findings and suggesting directions for future research.

2. Theoritical aspects

2.1 Physics of CHF

In a tube liquid cooling system, the flow regime of the coolant changes as the heat flux at the wall's surface increases, as can be seen in Figure 1. The steam quality factor increases, and the liquid starts boiling more intensely. Beyond a critical value known as the critical heat flux (CHF), the liquid at the heated surface vaporizes faster than it can be replaced by the incoming coolant. A stable vapor film starts forming along the heated surface, creating an insulating layer. Consequently, a sudden change in heat transfer capabilities will occur, and the temperature of the heated surface will rise, as illustrated in Figure 2.

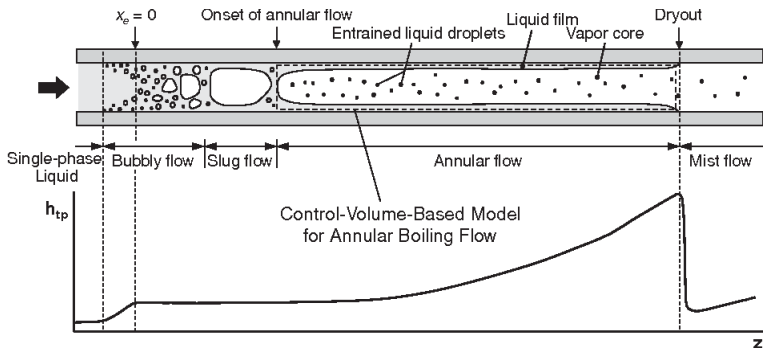

Figure 1: Flow regime from [8]

We now refer to CHF (as described in [9]) as a thermo-hydraulic phenomenon that occurs when the imposed heat flux is too large for the cooling liquid to effectively remove the heat. This phenomenon should not be confused with flow boiling, where, for a given heat flux, the temperature of the coolant increases as it travels along the heated surface. This will initiate a phase change if the tube is sufficiently long. These are two distinct phenomena, and only the first one is studied here.

CHF is also defined as the maximum amount of heat energy that can be transferred to a unit area of a surface from the fluid before it leads to the formation of a vapor layer. This value will naturally be the focus of our predictions.

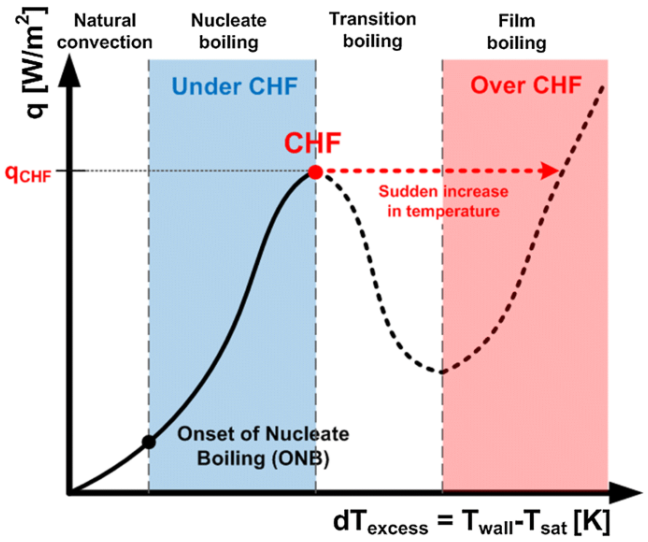

Figure 2: General boiling curve presenting boiling regimes and the effect of critical heat flux from [7]

2.2 Safety

CHF is a major concern in the cooling process of nuclear reactors. Beyond the CHF point, the deterioration of the heat transfer rate might cause an uncontrollable increase in the core temperature. This could lead to potential cladding damage or, in the worst-case scenario, fuel melting. Usually, if CHF is reached, the reactor is shut down and opened for inspection, leading to a loss in production. For these reasons, accurate CHF predictions are essential in the development of nuclear power plants.

2.3 Efficiency

In nuclear power plants, safety is a major concern. For this reason, margins are established to ensure safe values based on prediction accuracy. In our context, the maximum authorized heat flux of the fuel is configured to avoid CHF. Since these margins depend on the accuracy of our predictions, improved precision could lead to a reduction in these margins. This would permit higher heat flux or power density of the fuel, achieving higher burnup levels. In other words, it would enable the extraction of more energy from the same amount of fuel. This underscores another motivation for developing new models.

2.4 Analytic Model

As described in section 2.1, CHF is a complex phenomenon influenced by various factors. Furthermore, the scientific literature [9] indicates that multiple types of CHF exist. For this reason,

no unified model has been developed in the past 40 years. Different models have been proposed, they will not be elaborated in this paper. Instead, they will guide us in identifying the primary parameters to consider. Most of these models depend on Pressure (P), mass flux (G), the diameter and length of the tube (D & L), and the quality factor (X):

$$CHF = f(P, G, D, X, L)$$

These parameters are also present in the Lookup Table developped in the next section.

2.5 Lookup Table

Because of the vast number of correlations used to predict CHF, a lookup table has been created, compiling results from over 25,000 experiments. This has been realized with the aim to standardize the prediction methodology. These experiments were conducted using the same set of parameters each experiment build with different parameters values. The experiments were carried out exclusively with vertical water-cooled tubes. The data was normalized for a tube diameter of 8 mm as follows:

$$\text{CHF} = \text{CHF}_{D8\text{mm}} \times \left(\frac{D_{\text{real}}}{8}\right)^{-\frac{1}{2}}$$

$\text{CHF}_{D8\text{mm}}$ represent the CHF value in the table and D_{real} the real diameter. Here a look on the Lookup table representation:

Number	Data	D	L	P	G	Xchf	DHin	CHF	Tin	Reference
		m	m	kPa	kg m^{-2}s^{-1}		kJ/kg	kW m-2	°C	
1	25	0.004	0.396	100	77.5	0.84	317	442	23.94	*Lowdermilk 1958
2	26	0.004	0.396	100	142.7	0.79	317	757	23.94	*Lowdermilk 1958
3	27	0.004	0.396	100	203.9	0.7	317	978	23.94	*Lowdermilk 1958

Figure 3: 2006 Groeneveld Lookup table (LUT) from [2]

This table will be the support to create a solid database to train our neural network after some manipulation as describe in section 4.1.

Table 1: Parameters and their ranges

Parameter	Minimum	Step-size	Maximum
P [kPa]	100	200 - 3000	21000
G [kg/m^{2}s]	0	200 - 500	8000
X	-0.5	0.05 - 0.1	1
L [m]	-0.5	0.05 - 0.1	1
D [m]	-0.5	0.05 - 0.1	1

3. Neural Network (NN)

NN are a type of machine learning theorized and developed in the second half of the 20th century. They were inspired by our prior understanding of brain cells, and researchers tried to simulate their interconnections, information transmission, and learning processes. The main goal was to create a mathematical function that could 'capture' underlying complex patterns and physics by learning from observable data. In the 2010s, practical applications proliferated in many fields due to the increase in computational power, exploding the technique's popularity. Thus, neural networks represent a potential solution for creating CHF models.

As their name suggests, NN are built as a network of neurons in the form of interconnected layers, as can be seen in Figure 4.

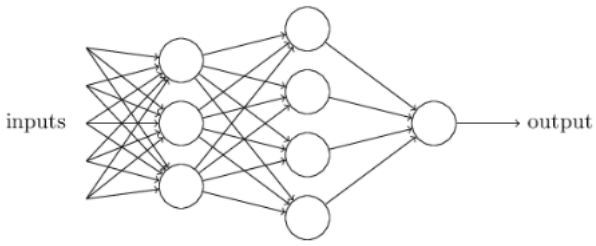

Figure 4: Caption

This all section is mostly based on [6]

3.1 Algorithm Description

Mathematically, NN can be viewed as a large function with thousands of parameters, with the learning technique as an optimization process. In this paper, we implemented a fully connected feed-forward neural network, also known as a multilayer perceptron. Each neuron is connected to all the neurons of the previous and next layer. Every connection has a weight, and each neuron possesses a bias. The value of a neuron is defined as the sum of its own bias and the

weighted sum of all the neurons from the previous layer (x_i). This is then passed through an activation function f, as follows:

$$y = f\left(b + \sum_{i=1}^{n} w_{iy} x_i\right)$$

with i designing the ith previous layer neuron, x_i its associated value and w_{iy} the weight of its connection between the neuron y

3.1.1 Activation Function

The all point of making activation function is to introduce non-linear properties into the model. Without it, the model would behave like a single-layer perceptron. This is also why the activation function should not be linear.It enables the capture of more complex and non-linear patterns. Additionally, it must also include a threshold concept to determine whether a neuron should be activated or not. The most commonly used activation functions are the ReLu and sigmoid functions:

$$Sigmoid = \frac{1}{1 + e^{-x}} \qquad ReLu = max(0, x) \tag{1}$$

The ReLu function has the advantage of being computationally efficient, meaning the model will require less time to train and predict. Another notable difference is the fact that, unlike the sigmoid function which has a maximum value of 1 at $x = \infty$, ReLu does not saturate at high x values. However, this characteristic can lead to excessively large values and unstable training. One solution to this issue is batch normalization (ref). In our implementation, the architecture will designate the list of neurons number present at each layer.

3.1.2 Architecture

The architecture of the NN is defined by its depth (number of layers) and width (number of neurons in a layer). Both of these parameters are crucial in the construction of an NN. The network needs to be deep to capture more complex underlying physics and various levels of abstraction. On the other hand, an excessively deep network may be computationally challenging. For the width, an overly wide model might lead to memorization rather than generalization. This phenomenon is called overfitting, and it can also happen for reasons discussed in Section 4.1.3.

3.2 Training Process

The major steps of the training process are as follows: It starts with a random set of weights and biases, called parameters. The training data input go through the network layer by layer and activation is calculated at each neuron with the parameters. Subsequently, the difference between the network's output and the expected value is evaluated using a function known as the loss function. The next phase, called backpropagation, is an optimization process using an optimization function. It will slightly adjusts all the parameters, from the output to the input. This process is repeated across all the training data. Once all training data has been processed by the network, it is said that one 'epoch' has been passed. The procedure occurs over several epochs.

3.2.1 Batch Normalization

During NN training, the network is fed with a subset of training data rather than a single value. This subset is referred to as a batch. It allow a better computational efficiency, especially when running on GPUs. It also aids in stabilizing learning and a faster convergence. An additional enhancement can be introduced, known as batch normalization. As the name suggests, it normalizes the batches to obtain a mean of 0 and a standard deviation of 1, while also storing the scaling factors. This acts like an additional hidden layer that can be placed after any chosen layer. With batch normalization, the training benefits from improved regularization (see [4]) and ensures no value blowup, especially when the ReLu activation function is employed. When the model is used for predictions, even if neurons receive a single value, the batch normalization layer continues to normalize using the stored scaling factors.

3.2.2 Loss Function

The loss function, also known as the cost function is designed to evaluate the distance between the measured value (the target) and the output value (the prediction). The most used Loss functions are the mean squared error (MSE) and the mean squared logarithmic error (MSLE):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2}$$

$$\text{MSLE} = \frac{1}{n}\sum_{i=1}^{n}(\log(1+Y_i) - \log(1+\hat{Y}_i))^2 \tag{3}$$

with Y_i and $\hat{Y}_i$ respectively the measured and predicted value.

3.2.3 Optimization Function

The optimization function or optimizer represents the way the parameters (weights and bias) are modified during backpropagation. The two most used optimizer used are the Adaptive moment estimation (ADAM) and the stochastic gradient descent (SGD) but they won't be developed in this paper.

3.2.4 Regularization

Regularization is the technique that prevents the NN from overfitting. Overfitting happens when the model learns/memorizes too closely the training data along with noise and outliers points. It can lead to a lack of generalization, giving poor performances on unseen data. Consequently, the concept of regularization is to ensure the model does not adapt to the noise by adding some constraints on the optimization process. Dropout is a regularization technique that will remove randomly some neurons during training. It will prevent any neuron from becoming overly specialized. Dropout parameter is a value set for a given layer and defines the occurrence or probability that each neuron 'drop out'.

3.2.5 Learning Rate

The training process can be visualized as travelling a path leading to a global minimum of the loss function. Each step along this path corresponds to a change of the weights and bias. The size of the steps or the change rapidity of parameters in response to the estimated error is called the learning rate (LR). If the learning rate is too big, the system might miss the minimum or not converge. On the opposite, if the learning rate is too small, the system might get stuck in a local minimum or converge too slowly. For this reason, the LR is an important value to adjust during training. As detailed in the implementation, a learning rate decay will be set in order to obtain convergence.

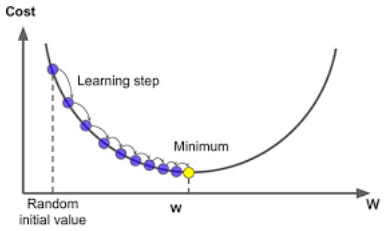

Figure 5: Learning rate visualization

3.2.6 Training Time

Another way to prevent overfitting is to avoid keeping training for too long. An implementation of a maximum number of epochs and early stopping will be set (see 4.2.2).

3.3 Hyperparameters

The weights and biases change value over training. They are designated as parameters. Hyperparameters (HP) represent all the values and general characteristics that embody a NN model. A different set of HP will create a different NN model with different performances. There are a lot of different HP that can be tweaked, and some have been introduced in the previous section: learning rate, architecture, dropout rate, optimizer, activation function, ... The implementation section will develop all the HP of interest.

3.4 Performances

To measure the performance of our model, the available data will be split into 2 sets as it will be developed in section 4.1. One to train our model, called the training set, used with backpropagation and optimizer as described earlier, and another to evaluate the performance, called the validation set. Once the validation set is passed through the network and predictions are made at the output, different mathematical tools called metrics can be used. The metrics used in this project are the following:

- mean squared error (MSE) 2
- mean squared logarithmic error (MSLE) 3
- mean absolute percent error (MAPE):

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{Y_i - \hat{Y}_i}{Y_i}\right| \times 100\% \tag{4}$$

- normalized root mean squared error (NRMSE)

$$\text{NRMSE} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2}}{Y_{mean}} \tag{5}$$

- mean of the fraction Measured over Predict (Mean MP)

$$\text{Mean MP} = \frac{1}{n}\sum_{i=1}^{n}\frac{Y_i}{\hat{Y}_i} \tag{6}$$

- standart deviation from 1 of MP (STD MP):

$$\text{STD MP} = \sqrt{\frac{\sum_{i=1}^{n}(1-\frac{Y_i}{\hat{Y}_i})^2}{n}} \tag{7}$$

4. Implementation

The feed-forward neural network capable of predicting CHF has been implemented using Python. The code is object-oriented with the presence of different classes to clarify the different steps of the process. Furthermore, everything has been grouped into a package in order to simplify its usage (manual + doc). The architecture of the package is presented in Figure 6

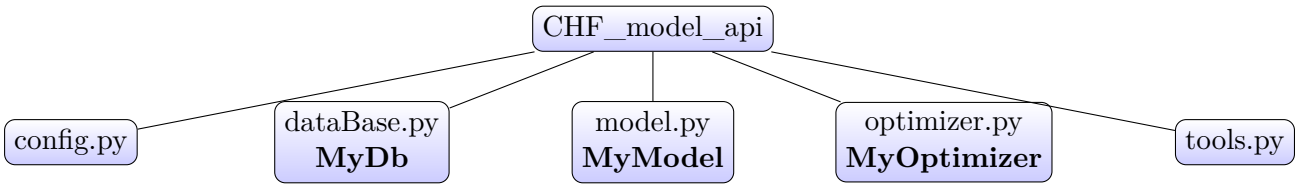

Figure 6: Architecture of the package

MyDb, **MyModel**, **MyOptimizer** are classes developed respectively in section 4.1, 4.2 and section 4.3. *tools.py* contains diverse useful functions that don't have their place in the classes such as the metrics ans *config.py* is composed of main paths and configuration values.

The primary references for developing the code were the documentations of the utilized libraries. The *Tensorflow* library [5] was employed to design the **MyModel** class and the *Optuna* library [1] was utilized for the **MyModel** class.

4.1 Database

The first part of the implementation is the creation of robust data in order to train correctly the model. For this purpose, a class called **MyDb** is created in the file *dataBase.py*. Each object of the class will correspond to a specific database adapted to a given model which is an object

of the class **MyModel**. The different steps to obtain the final database are the extraction, SI units transformation, filtration, normalization and splitting.

4.1.1 Extraction

The available resource is the 2006 Groenevel LUT [2] introduced in section 3. It comes in the form of a pdf file. Consequently, the library *tabula* and the function *tabula.read_pdf()* are used in the method *extractSortFromPdf(self,path)*. The pdf file being not perfect and the *read_pdf()* function not all-powerful, a lot of little manipulations are made in order to remove headers, unnecessary columns, incorrect lines and irregularities. The remaining columns are {D,L,P,G,Xchf,DHin,CHF,Tin}.

4.1.2 SI units & Filtration

Each column is then transformed in SI units. Then, a new column containing value L/D will be added to the data. Next, the following filters are applied according to [2]:

$$\text{CHF} > 1 \qquad 10^5 \leq P \leq 21.10^6 \qquad 0 \leq G \leq 8000 \qquad X > 0 \text{ and } L/D > 50$$

$$0.003 \leq D \leq 0.025 \qquad Xchf < 1 \qquad X < 0 \text{ and } L/D > 25$$

Those filters will exclude outliers points and physical nonsense. The data will be then limited to this set of columns: L/D, Xchf, G, P, DHin or this set: L/D, Xchf, G, P in function of the input number we desire to train on. This will be the subject of a point in section 5 with the intention to compare which set will give the best predictions.

4.1.3 Splitting

As introduced previously, the data must be split in 2 parts so that the model can train but also be evaluated on its performances. Usually, the training set represent 70 or 80% of the available data and will be arbitrarily 80% in our case. Furthermore, the split must not be 'too random' and so for 3 reasons. The first is that the 2 set must have the same distribution of CHF. For example, it avoids the model to be trained on too low values and not being able to make correct predictions on high CHF values. This is called stratified sampling. The second reason to not conduct the split 'too randomly' is the reproducibility. Indeed the program must be able to reproduce an exact same model based on its HP to make comparison between the models. To do so, a seed will be used . A seed is defined as the starting point of the (pseudo) random

generator (*random()* function on the *random* library in our case) sequence of operation leading to a number that appear random. The seed will be what fundamentally differentiate a MyDb object from another. It also means that only the seed number need to be saved in order to reproduce the database exactly the same. As it will be developed in section 4.2, it allows to save a model without its database, and retrain it. For this reason the seed is integral part of the model HP.

4.1.4 Normalization

As previously mentioned, NN algorithm can be data scale dependent. It helps avoids saturation. Normalized data will also avoid the error landscape to be elongated in some direction, slowing down the convergence in some case. Even if batch normalization is used it is good practice to normalize the data going into our NN. In this project, the standard normalization is applied on the input data with *StandardScaler* from the *sklearn* library.

$$X_{i,normalized} = \frac{X_i - X_{mean}}{X_{std}}$$

The letter X to illustrate that input parameters (features / X) are normalized (independently). X_{mean} and X_{std} being respectively the mean and standard deviation of the **training set**. Indeed with the aim to evaluate the performance of the NN correctly, the validation set must represent unknown data and not influence our model in any way. That's why they aren't involved in the computation of the mean and Std. Nevertheless, any data going into the model need to be normalized. The validation set is then also standardized with X_{mean} and X_{std}.

Now, the database is prepared. The following values are stored in a dictionary, which is an attribute of the MyDb class.

- 'validation_targets': a list containing CHF values of the validation set
- 'validation_features': a list of lists with the input parameters of the validation set
- 'train_features': a list of lists with the input parameters of the training set
- 'train_targets': a list containing CHF values of the training set
- 'mean': list of mean values for the different input parameters
- 'std': list of standard deviations for the different input parameters

Indeed, the mean and std are saved in order to pass them as attributes of the associated model. The purpose is to normalize new incoming data if real predictions need to be made. Now that we have a robust database, it can be used to test the performances of the LUT table with the aim of making a comparison with the NN. This will be developed in the section 5.

4.2 NN Foundation

The second part of the project focused on the CHF NN models. This part is implemented in the file *model.py* and take the form of a class named MyModel. Each object is associated with a different database (if the seed is different), and they will correspond to different NNs. In the sens that they comes with a different set of HP or level of training. The class has been designed to facilitate the creation of new models or the use of previously saved ones. The various steps to obtain a model capable of making predictions are Structure Creation, Callback Initialization, Data Loading, Training, and Saving.

To create a new model from scratch, a dictionary containing HP must be passed to the MyModel arguments. The dictionary is analogous to a human ID card. It contains its name and its characteristics. The HP dictionary will also be used to store the performances. This section will elaborate on the different HPs, helping to understand the scope of the model and what can vary from one model to another.

4.2.1 Structure Creation

Before setting anything, the *Tensorflow* seed is fixed. The *Tensorflow* library also uses functions with its own randomness. This is the case, for example, when the weights and biases are initialized to random values. As mentioned previously, the seed needs to be fixed to be able to reproduce the results. The 'architecture' HP, which is a list containing the number of neurons per layer, is then used to create the base structure. An activation layer with the ReLu function 1, a batch normalization layer, and a dropout layer (only for the first two layers as is common practice) with a set dropout rate are added to each hidden layer. The optimizer is set to *Adam*, and the network is compiled with a learning rate, a loss function, and a choice of other metrics selected for additional insights during training.

4.2.2 Callbacks Initialisation

The callbacks are actions taken during each step or after each epoch of training with the aim of monitoring progress. Logging is a type of callback. Another is early stopping. It is

set up to check every 'patience' (another HP) epochs if the loss changes by a value smaller than 'loss_delta_stop', and if so, to stop the training. The last one is called a learning rate scheduler, and its purpose is to multiply the learning rate by a value smaller than 1, termed 'learning_rate_decay', every 'patience' epoch. For example, if 'learning_rate_decay' is equal to 0.96 and 'patience' is 20, the learning rate will decrease by 4% every 20 epochs.

4.2.3 Data Loading

The MyModel object will then check if a compatible MyDb object exists (if the seed and input number are the same) or create its own.

4.2.4 Training

At this point, the model is ready to be trained with the method *train()* and can already be saved using *save()*. The *Tensorflow* library will automatically detect if a GPU is available to speed up the training. As previously mentioned, input data are provided in batches, and the batch size can be modified to adjust the level of parallelization or to test for convergence. This is why 'batch_size' is part of the HP. A maximum number of epochs can be set to limit training with the HP 'max_epochs'. At the end of the training, the program will automatically record the number of epochs the model was trained on and its performance. To achieve this, it will use the metrics functions in *tools.py* and store the results in the HP dictionary.

4.2.5 Saving

Upon creation, each model is assigned a unique name/ID based on the date and time when the MyModel object is initialized. The model can be saved in a directory labeled 'saved_models'. Inside this directory, the model is divided into two files, placed in sub-directories named 'hparams' and 'models'. The first is a .json file containing the dictionary of HP, and the second is a .h5 file with the binary information for the weights and biases. Both files are named after the model's ID. To retrieve a saved model, you simply pass the model's ID as an argument to MyModel. The loaded model can then be further trained, resuming from where it left off, or used to make predictions with the *predict()* method.

4.3 Hyperparameters optimization

Now that a model generator for CHF predictions has been implemented in the program, we can also introduce an automated generator of models with the goal of testing different sets of

hyperparameters (HP). With the aid of the Optuna and multiprocessing libraries, we can now develop a script that tests multiple models concurrently, aiming to find the optimal combination of hyperparameters that lead to the best predictions. In simpler terms, we are now equipped to establish a hyperparameter optimizer. This functionality is encapsulated in the optimizer.py file within a class named MyOptimizer. Each instance of this class will represent a unique optimization study.

4.3.1 Objective Function

A single study, or a MyOptimizer object, will execute multiple trials. Each trial calls an objective function that constructs a specific model, trains it, and evaluates its performances. This is the core of the Optuna optimization process. Optuna suggests hyperparameters (HP) to the objective function aiming to minimize the value it returns. These suggestions are found by exploring the hyperparameter space, using the Tree-structured Parzen Estimator (TPE) algorithm, which is a form of Bayesian Optimization.

4.3.2 Choice of Hyperparameters

MyOptimizer objects are easily manageable to configure which HPs need to be optimized. For now, only the choice of 'architecture', 'learning_rate', 'dropout_rate', 'optimizer' and 'learning_rate_decay' is possible.

4.3.3 Multiprocessing

To multiply the number of trials that can run at the same time, the program use multiprocessing. It will use different processes or cores that are going to run a study each with the purpose to accelerate the Optimization process that can be very long since it consists basically to train a lot of neural networks

4.3.4 SQL Database

The collection of the data resulting from the optimization process will be put in an SQL Database. It offers the benefits to be easy to implement on the computer and simple to pass to Optuna study function. The SQL database can also deal automatically with multiprocessing where processes might try to access the optimization database at the same time. The database can also be reused once the program shuts down, to improve the optimization process continuously.

5. Results

To make comparisons between LUT and the NN, a script for interpolating the LUT has been made in MyDb class using the training set and tested with the validation set. The results are reported in the table 2

MAPE	**Mean MP**	**Std MP**	**NRMSE**	**MSLE**
8.51%	0.998	0.136	0.126	0.018

Table 2: Performances of the LUT table

After some tries of optimization process using MyOptimizer, one of the most performing hyperparameters set is reported on the table 3 and the performances on the created model on the table 4

input_number	4	architecture	[4, 60, 56, 25, 42, 23, 22, 16, 19, 1]	learning_rate	0.00342
dropout_rate	0.335	lr_decay	0.96	rythm	20
alpha_acti	0	batch_size	32	optimizer	adam
max_epochs	1000	loss_function	msle	loss_d_stop	0.001
batch_norm	true	patience	50	trainedEpochs	1000

Table 3: Hyperparameters of the model

MAPE	**Mean MP**	**Std MP**	**NRMSE**	**MSLE**
16.5617%	1.0029	0.1977	0.2312	0.0416

Table 4: Metrics and their respective values.

Unfortunately, the performances of the model aren't as good as the LUT but the metrics give good signs of convergence. A more visual proof of convergence is the graph of the predicted value over the measured one at figure 7.

6. Transfer Learning

Although the results of the NN don't catch up with the LUT table, this project can serve as a good base for transfer learning to new geometries.

Transfer learning consists of involving a neural network, pre-trained on a large dataset, that is adapted for a different but related task. Rather than training a model from scratch, transfer

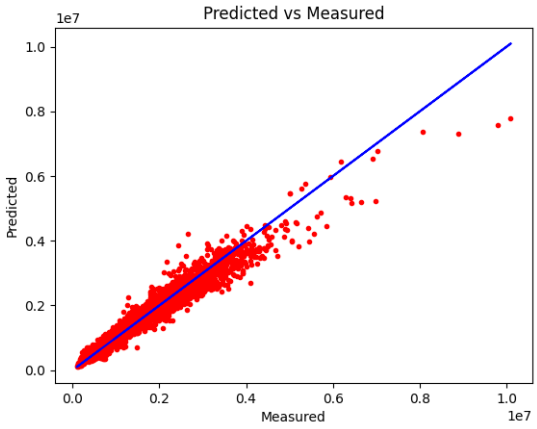

Figure 7: Predict vs Measured

learning benefits the patterns and knowledge gained from the initial training task to achieve improved performance and faster convergence on the new task. It will use the pre-trained NN and add 2 or more layers and fine-tune the entire model to new data, in the context of this project it can be the data of another geometry. The fine tuning has the advantage of requiring less data since the model is already pre-trained. This type of model could probably gain a significant advantage over the LUT table that is limited to a single geometry.

7. Conclusion

In our endeavor to modernize Critical Heat Flux (CHF) predictions, neural networks were explored as alternatives to the traditional 2006 Groeneveld Lookup Table (LUT) interpolation method, which had a MAPE of 8%. The newly developed model, implemented using the TensorFlow and Optuna libraries, resulted in a Mean Absolute Percentage Error (MAPE) of 16%. Although this doesn't surpass the LUT, the evident signs of convergence in the model's performance offer encouragement. Taking into account the importance of accurate CHF predictions, for both reactor safety and operational efficiency, future research could deepen and and refine model hyperparameters and investigating other advanced machine learning techniques to bridge the accuracy gap.

References

[1] Takuya Akiba et al. “Optuna: A Next-generation Hyperparameter Optimization Framework”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 2623–2631.

[2] D. Groeneveld et al. “The 2006 CHF look-up table”. In: *Nuclear Engineering and Design* 237.15 (2007). NURETH-11, pp. 1909–1922.

[3] Emil Helmryd Grosfilley. *Investigation of Machine Learning Regression Techniques to Predict Critical Heat Flux*. 2022.

[4] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[5] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015.

[6] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[7] JY. Park, L. Peng, and JW. Choi. “Critical heat flux limiting the effective cooling performance of two-phase cooling with an interlayer microchannel”. In: *Microsystem Technologies* 25 (2019), pp. 2831–2840. DOI: 10.1007/s00542-018-4270-y.

[8] Issam Mudawar Sung-Min Kim. “Theoretical model for local heat transfer coefficient for annular flow boiling in circular mini/micro-channels”. In: *International Journal of Heat and Mass Transfer* 73 (2014), pp. 731–742.

[9] Bao-Wen Yang et al. “Progress in rod bundle CHF in the past 40 years”. In: *Nuclear Engineering and Design* 376 (2021), p. 111076. ISSN: 0029-5493. DOI: https://doi.org/10.1016/j.nucengdes.2021.111076.