Computacional Intellingence: ASHRAE Kaggle Competition 2019.

How much energy will a building consume?

Members:
Alejandra López de Aberasturi Gómez
Bernat Martínez Gómez
Eric Cañas Tarrasón
Rafael Mesa Hernández

Course: 2019/2020

Contents

Abstract

Generating the best *Machine Learning* based *Regressor* model for predicting the value of a relevant continuous variable can be the difference between millionaire benefits or losses for an enterprise. This is the main reason why every day more enterprises are publishing well rewarded public competitions through *Kaggle* platform. In this work we show a real case about how to deal with this competitions. Concretely, facing the *ASHRAE* problem: The definition of a model for estimating a building data consumption. Driven by this problem, we analyze different procedures for analysing and processing the data as well as defining and parameterizing a *Neural Network* based architecture. Finally we will show where is most important to focus when proposing related experiments and how to analyze their results.

Keywords: *Kaggle, Neural Network, Regressor, ASHRAE, Dataset, Energy.*

1 Introduction

Kaggle is known as one of the most relevant places where *Data Scientists* meet, sharing and developing their knowledge through periodically *Machine Learning* competitions. These ones are organised by different enterprises interested on developing machine learning models and will grant lush awards to the best models. As *Computational Intelligence* students, our purpose is to involve ourselves in this kind of competitions determining our start point before ending this first semester.

We have selected the *ASHRAE - Great Energy Predictor* competition due our special interest as a group in *Regressors*. We also aim to deal with real life non-academic problems with the purpose of learning a *Machine Learning* framework in the process, in this case, *Pytorch*.

There are several approaches for creating a prediction model for a problem like the one proposed. In order to comply with the objectives and lessons of the subject as good as possible, we decided to center our efforts in the following models: *Neural Network* architectures, focused on *regression* problems which are optimized through *gradient descent* based methods with *backpropagation*.

More specifically, the *ASHRAE* competition proposes a regression problem with an enormous quantity of data as input (more than 20 millions of samples for training). This data combines categorical and numerical features, as well as variables with missing data. We needed to decide how to process these last cases. Throughout our work, this data will be precisely analyzed in order to see which prepossessing methodologies yield the best results. A series of *Neural Network* architectures proposals will also be analyzed in order to find a best possible approach. Given enough information and issues diagnostics, we expect to place ourselves in the competition ranking.

2 Community knowledge

There are two relevant fields of knowledge that are important to keep into account during the development of this model. The specific contextual knowledge of the *ASHRAE Competition*, and the information regarding *Regression Neural Networks* models (since we are focusing on them, they are the most interesting to us). In order to avoid redundancy in the present work, *Regression Neural Network* researches, and references about state of the art has been included through experiment proposals developed on section 5.

Inside the *ASHRAE Competition*, there are a big amount of notebook researches explaining how data is structured and distributed [1] as well as how different models are able to fit it or not. Despite most relevant of them about results were not released until the close of the competition, important and gentle researches about the data [2] could be found earlier.

Finally, the *leaderboard* [3] shows blindly what is the real state of the art on this problem resolution, so it is important to take it in account for seeing how far we are from it.

3 Data

The ASHRAE dataset [4] contains the following files:

1. **[Train.csv—Test.csv]**: More than 20 millions of examples containing the output variable for Training and more than 40 millions without it for Testing:
   (a) *Building ID* - Foreign key for the *building metadata file.*
   (b) *Meter* - The meter id code. Read as {0: electricity, 1: chilledwater, 2: steam, 3: hotwater}. Not every building has all meter types.
   (c) *TimeStamp* - When the measurement was taken.
   (d) *Meter Reading* - The target variable (Only in training set). Energy consumption in kWh (or equivalent). Note that this is real data with measurement error, which we expect will impose a baseline level of modeling error.
2. **Building Meta.csv**: General information about each of the buildings in the dataset
   (a) *Site ID* - Foreign key for the *weather file.*
   (b) *Building ID* - Foreign key for *training.csv.*
   (c) *Primary Use* - Indicator of the primary category of activities. for the building, based on *EnergyStar* property type definitions. There are 16 different possibilities present in the dataset.
   (d) *Square Feet* - Gross floor area of the building.
   (e) *Year Built* - Year building was opened.
   (f) *Floor Count* - Number of floors of the building.
3. **Weather [Train/Test].csv**: Information about the weather nearby to the building, coming from meteorological stations.
   (a) *Site ID* - Identification of the place.
   (b) *Air Temperature* - Degrees Celsius.
   (c) *Cloud Coverage* - Portion of the sky covered in clouds, in oktas.
   (d) *Precipitation Depth in 1h* - Millimeters.
   (e) *Sea Level Pressure* - Millibar/Hectopascals.
   (f) *Wind Direction* - Compass direction (0-360).
   (g) *Wind Speed* - Meters per second.

3.1 Our Composition

In order to present the data in the optimal way for the different Neural Networks generated we have done the following pre-processing:

1. **TimeStamps**: Dates has been transformed into separatedly: Month, Day of the week, hour (In first approaches Day of the year and hour were tried, but demonstrated to give poor results). This decision have been based in the aiming of give in the best possible way, the implicit information to the network (There is an implicit relation between the month, the day of the week and the hour, with the energy consumption)

2. **Categorical Values**: *Meter*, and *Primary Use* variables have a categorical nature. By this fact they have been implemented through *one hot coding* [5]. After some deliveries and initial attempts, time related variables have been decided to also be *one hot coded* in order to let the biases of the network decide which are the best influence of the different months, days and hours.

3. **Standarization**: In order to give the variables in the best possible way, the ones which are not *One Hot Coded* has been standardized. Standardization is a way to put re-scaling them in the same range ($\mu = 0, \sigma = 1$) through the following equation:

$$Z_i = \frac{X_i - \overline{X}}{S} \tag{1}$$

With this pre-processing our training vectors has been composed with the following structure:

0-3 : Meter variable *One Hot Coded* (2: steam, would be represented as [0,0,1,0] vector).

4-16 : Month of the Year *One Hot Coded.*

17-23 : Day of the Week *One Hot Coded.*

23-47 : Hour of the Day *One Hot Coded.*

47-63 : Primary Use variable *One Hot Coded.*

63-66 : Square feet, Year Built and Floor Count of the building.

66-72 : Information about weather of the day (Air Temperature, Cloud Coverage, Precipitation Depth in 1h, Sea Level Pressure, Wind Direction and Wind Speed).

3.1.1 Dimensionality Curse?

Taking a first look at the input data, a 73-dimensions vector could looks as an excessive dimension, entailing all of the usual dimensionality known problems. However there is important to note the fact that 64 first dimensions correspond with only 5 *One Hot Coded* variables. It implies that from the 73 input neurons, only 14 if them will be activated each time, and five of them getting always their bias weight (The weight of be measuring in Sunday or in Wednesday, or be measuring at 8pm or 2am). It is also important to note that *One Hot Coding* is a technique that may entail risk when datasets are short and we could have no examples for concrete values of a variables, however, it is not the case of this dataset containing more than 20 millions of measurements.

3.2 Analysis of Correlation

In order to determine the potential of the variables we have when making a regression prediction, the best approach is to make Correlation analysis between each one of them and the objective variable to predict.

In table 1 we present the *Pearson Correlation Coefficient* from each one of the input variables in relation with the output (Meter Reading), when extracting the missing values rows.

As seen here there are strong correlations inside the dataset, specially with the variables which refers to the following facts:

1. **Dimensions of the building**: *Floor Count* and *Square Feet* (Both positives, since as bigger is the building, bigger will be his energy consumption).

2. **Use of the building**: *Primary Use Descriptor* (There is different the energy consumption for a House, than for a School).

Pearson Correlation	
Variable *(X)*	Meter Reading *(Y)*
Floor Count	0.7290
Primary Use	-0.5431
Square Feet	0.5415
Meter Code	-0.5275
Precipitation Depth	-0.5200
Dew Temperature	0.2473
Hour	-0.2175
Air Temperature	0.1221
Cloud Coverage	-0.1210
Wind Direction	0.1238
Sea Level Pressure	0.1153
Month	-0.0813
Year Built	0.0730
Wind Speed	0.0344
Day Of Week	-0.0244

Table 1: Pearson Correlation between each one of the inputs variables with the objective variable. Ordered by his relevance

3. **Measure Taken**: There is different the expected value when taking an electricity or water measure.
4. **Weather**: The *Precipitation Depth* have a specially strong correlation with the consumed energy, as well as, in lower measure, the *Dew temperature*. Measures like the *Air Temperature*, *Cloud Coverage*, *Wind Direction*, and *Sea Level Pressure* have have a specially weak relation.
5. **Time**: In this case the most related Time measure is the *Hour* of the day.

With this information we usually could consider to erase from the dataset the weakest related variables, like the ones with $|Correlation| < 0.1$, but due it is a *Neural Network* approach, the Network would be capable of multiply this inputs by a so lower value or .0 if needed. Also is important to have in account that there are possible relations between pairs of variables like, *Month* having higher relations when measuring *Electricity*, than when measuring *Water* consumption. By favoring learning this relationships the implementation of all discrete or categorical variables has been *One Hot Coded* as last mentioned.

3.3 Missing Data

One of the greater problems about this dataset is how to deal with the high amount of missing values, present in Train and Test Set. For taking a first look at them, tables 2 and 3 shows the amount of them present in the given sets.

As seen here, there are four variables which are dangerous in terms of missing values data. We can divide them in two types:

1. **Heavy Dangerous**: *Floor Count.* It is the one which besides of having a large amount of missing values have a highest *Pearson Correlation* with the objective variable (0.729). Is important to have in account the fact that loosing this information could damage heavily the performance of the Network, but also interpolate it when missing could introduce noise in more than half of the dataset samples.

Missing Data (*Building Data*)	
Variable	Missing Data
Floor Count	0.7550
Year Built	0.5341
Primary Use	0.0
Square Feet	0.0

Table 2: Amount of missing values on *Buiding Data file*.

Missing Data (*Weather FIles*)		
Variable	Over Train	Over Test
Cloud Coverage	0.4949	0.5066
Precipitation Depth	0.3597	0.3447
Sea Level Pressure	0.0760	0.0767
Wind Direction	0.0448	0.0462
Wind Speed	0.0022	0.0017
Dew Temperature	0.0008	0.0012
Air Temperature	0.0004	0.0037

Table 3: Amount of missing values on *Weather files* over *Train* and *Test*.

2. **With Avoidable Impact**: For the weather values (table 3) we have less problems due we can manage two valid solutions:
   (a) **Search for the data in nearby hours when is missing for this moment**: Since Weather files are indexed by day an hour, we can expect that the difference between weather in nearby hours will be low, so we can use them when required.
   (b) **Not using variables with high amount of missing values**: Given, we have in this weaker correlations with the objective variable (except from the *Precipitation Depth* case), the damaging in the performance of the model will be lower if we decide to not use some of this variables. *Year of Built*, would be included in this solution cause his low *Pearson Correlation*.

4 Implementation

We have generated a flexible environment where proceed as easier as possible with the generation and evaluation of different experiments. For this purpose the implementation of the executed Neural Networks has been done in python, using *pytorch* library. The key points of the code structured are the followings:

1. **ASHRAEDataset.py**: This file contains the ASHRAEDataset class, inheriting from the *pytorch Dataset* class. It overrides the __init__, __len__ and __get_item__ methods in order to make it iterable and compatible with the *Batch Readers* offered by *pytorch*. This class reads all the *csv files*, and generates each one of the inputs for the Neural Network, keeping an eye on not making unnecessary merges of the original data in order to not overflow the *RAM* due to the large amount of data the *datasets* are composed. It also takes care of the missing values in datasets, assigning them an interpolated value (*mean* or *median*), and standardize the non *One Hot Coded* data.
2. **Train.py**: Train a network in GPU. This file have two important functions:
   (a) **train()**: Takes the Training set of the ASHRAEDataset, split it in to 95% for Training and 5% of validation and wraps it inside a batch reader. Generate a new model or charges an

existing one. Execute iteratively (for N epochs) a training phase, followed by a Validation phase. During each one of this phases shows:

i. The *Mean Loss* for a set of sample *batches*.
ii. The *Pearson Correlation* between the output and the *Ground Truth* for a set of sample *batches*.
iii. The estimated *Remaining Time* until ending the *Epoch*.
iv. The total *Mean Loss* and *Pearson Correlation* for the complete *Epoch*.

Moreover, during each M epochs is saving the trained model and plotting a graphic of the results in a *Tensorboard file*.

(b) **get_predictions()**: Charge a pre-trained model and the test datasets. Then generates a *submission csv* in the correct format for being updated to *Kaggle*.

3. **NetworkUtils.py**: Contains the set of useful functions for saving and charging models, as well as the function for calculating the *Pearson Correlation Coefficients*, extracting them and the definition of the *RMLSE Loss Function*.

4. **Models.py**: Define the different Network Models for being used in different tests.

The generated source code is annexed to this report.

5 Experiments

In this section will be defined the key experiments performed. Determining how hyper-parameters has been decided, which architectures has been tested and why. Is important to remark here, that the amount of experiments could not be as extensive as desired (There is impossible to execute grid searches or keep non promising models learning a big amount of epochs). This is due to the size of the training set provokes extremely high learning times (rounding the 10 minutes per epoch) and due his resources consumption there is impossible to run experiments in parallel under the same machine.

5.1 Deciding hyper-parameters

The definition of the hyper-parameters of the Network is a decision that could imply the difference between converging in the winner model or diverging, not allowing your network to fit the data, or giving a totally useless results.

5.1.1 Loss function

The loss function defines which shape will take the cost function. Which is what is going to be used to apply gradient descent. Some functions could be considered in the case of regression models, a nice summary of the most important five can be found in the approach written by Grover Prince [6].

In this case *MSE*, *RMSE* and *L1* losses were tried in first tests, but definitely, since this work is based in the *ASHRAE* competition, the best approach is to use the loss function used as competition score in order to optimize this score, *RMSLE Loss* .

RMSLE is defined as follows:

$$RMSLE = \sqrt{\frac{1}{N}\sum_{i=1}^{n}(log(p_i+1) - log(a_i+1))^2} \tag{2}$$

Being N the total number of observations, p_i the prediction for the target a_i, and $log(x)$ the natural logarithm of x.

5.1.2 Optimizer

Selecting an optimization algorithm is another important parameter what will decide the overall efficacy of our regression system. The most two relevant options that have been considered for this task were *SGD* and *ADAM* optimizers.

1. **Stochastic Gradient Descent (SGD)**: *Stochastic Gradient Descent* [7][8], is the most popular Gradient Descent method for Neural Networks since it has a longer history among the rest of machine learning algorithms. Compared to the Adam optimizer, the hyperparameters we need to adjust are mainly the *Learning Rate* and *Momentum*.
2. **Adaptive Moment Estimation (Adam)**: *Adam* [10] is an adaptive method which computes an adaptive *Learning Rate* for each single epoch of the training phase. It is more modern than *SGD* (2015) and relieves the user from selecting the *Momentum* value. The spread of acceptable values in the *Learning Rate* is extended, giving the designer more freedom and less margin of error. This is due to the adaptive nature of the learning rate, which will be automatically adjusted across the training process.

In this case, there are a lot of studies comparing both algorithms (as well as proposing new ones) in order to find the best possible general approach. However, the conclusion usually comes to be that in the majority of Machine Learning cases: *results will depend on the specific problem, and the best decision should be based in experience and specially: Trial and error.*

In this case we have decided that, in spite of *SGD* method should be tried, it will only be used in the most promising cases, when a fixed architecture will be decided. Since, due to the expensive cost of each experiment (hours of computation), its an enormous resources optimization to avoid as much as possible grid searches over *Learning Rate* and *Momentum* parameters.

5.1.3 Batch Size

The *Batch Size* is another hyper-parameter than determines the efficacy of the model. It can both make the system prone for better training or dissuade completely the odds of converging.

Fortunately, the dataset has a large amount of samples (more than 20 million) which does not occupy excessive memory space. This provides us with more freedom to select different batch sizes and see the effects they have on the training. The following sizes have been tested :

1. **An extremely high *batch size* (20.000 samples)**: In this case the performed training were, as expected, the fastest one of the 4 configurations. The mean time was about 6 minutes and 52 seconds per epoch using a *GTX 1060 GPU*. However, all test performed by this train (Whose *csv* could not be included in this report due it's weight greater than *700MB*) converged rapidly to a constant result ($f(x) = C$), implying the impossibility of being considered useful.
2. **A medium-large *batch size* (256 samples)**: In spite of having larger convergence times, when using this *batch size* value (Rounding the mean time of 8 minutes and 30 seconds per epoch) the networks using this *batch size* achieved a good fitness, their architectures were not deeper than five layers. In terms of the trade-off between efficiency (and not increasing the error), this *batch size* could be considered the best between the ones tested.

3. **A medium *batch size* (128 samples)**: This *batch size* gave results equivalents to the case were *256 samples* are used. However, it extended the mean time of computing an epoch up to 17 minutes and 30 seconds. In some cases, it could be considered as a replacement of the *256 samples batch size.*

4. **A low *batch size* (32 samples)**: Experiments using this *batch size* fitted well in all tested architectures, arriving to equivalent results in less epochs than the aforementioned *batch sizes.* However the learning times were always superior to 1 hour per epoch. This made it harder for evolving the model and trying different architectures. We think that although is not efficient for our purposes, it could be considered as a valid *batch size* when trying extreme cases of deep architectures. For those configurations, the tested larger batch sizes eventually converged to constant results.

Table 4 shows the best scores obtained by Models of *Two Hidden Linear Layers* using each described *batch size.*

Performance influenced by Batch Size		
Batch Size	RMSLE (Test)	Epoch Time
20.000 *	3.994	6 min 50 sec.
256	2.373	8 min 30 sec.
128	2.403	17 min 30 sec.
32	2.174	> 1 hour.

Table 4: Performance influenced by batch size (*Batch Sizes* converging to a constant value are marked with *.) The scores displayed come from equivalent architectures composed by *Two Hidden Linear Layers* and followed by a linear output (All of them using *ReLu* activation). All models were trained to a maximum of 10 hours.

As a conclusion, is almost impossible to select an optimal batch size that works properly for all the possible architectures. It is an aspect of the whole design that entirely depends on the specific architecture used at the time. We still think that the case of extreme batch sizes can be directly discarded. With the obtained results shown in table 4 in mind, as well as being guided by the studies like the ones made by Godfellow et al. [11] and the experience, we conclude that the batch size will have to be adjusted for each individual experiment. When searching for a promising structure, the first attempt should be the largest one with acceptable behaviour: 256 samples. Once its found, then we proceed to lower the *batch sizes* gradually with the intent of finding an optimum size that suits the architecture being optimized at the time. We stop testing smaller sizes whenever we notice that the network is performing way worse than it did with the previous batch size.

5.1.4 Train-Validation-Test Splits

For this competition, we are provided of two separated datasets. *Training* and *Test*, which are divided as follows:

1. **Training Set**: *20,216,100 samples* (whose measurements date back to *2016*) for the same buildings that will be measured in test. This dataset provides us with all the values for the target variable (complete supervised problem).

2. **Test Set**: *41,697,600 samples* (where the measurements date back to *2017* and *2018*). For this case, the ground-truth is not available. The idea then is that an external software located in a server of Kaggle will judge our model obtaining values such as *Precision*, *Recall* and *Accuracy.*

The data is splitted in the following proportions. 32,65% of it will belong to the train (and validation) set. The remaining 67,35% will belong to the testing dataset that will be used to make the final evaluation of our model.

Since the training dataset is extremely large, we allowed ourselves to select a small percentage of it ($5\% \simeq 1$ Million of samples) to be used as Validation set. The main purpose behind this is to establish a baseline for overfitting detection. Initially it is easy to presume that in order to quantify the performance of our model we just need to check the predictions against the groundtruth of the testing. This is a dangerous false intuition commonly seen in novice machine learning students, and its important to address it. Thinking more carefully, although we might avoid overfitting in our model per se, but we are not free of any of it. If we test multiple structures and select the one that generates the least error, we are overfitting the test dataset. Keeping a third validation dataset unmixed with the original test ensures that if we use it for selecting our model, we are not overfitting the test groundtruth.

Moreover, its important to be careful when deciding how to split Train and Validation sets, due the offered dataset has an implicit order (By date and types). Is very important to apply a shuffling in order to ensure that there is no biases between the data included in both sets. As an example, if we did not proceed to this, it could happen that validation was composed only of samples from summer dates.

5.1.5 How to feed the network

There are several studies which explore the best preorders of data when feeding a Neural Network. Active Learning [9] for example, is a wide research field focused on this.

This aspects could be relevant in some cases, specially in the computer vision field, where there is possible to define orders which will decrease the difficulty of the problem to fit. For this case however, all data could be considered equally relevant and with equivalent difficulty. We decided that the best approach is stick with random sampling and discard the initial data ordering. This will increase the expected variance within batches favouring a faster and a more stable generalization.

5.2 Deciding the network architecture

Once the hyper-parameters of the network have been revisioned, some of them have been kept fixed (as *Loss Function*). However, most of them (as *Batch Size* or *Optimizer*) different possibilities has been analyzed or experimented without fixing a general optimal. In these cases we have defined ranges that include which values are the most well suited for the majority of cases. There are also others that should be tried only in specific or most promising ones. For example, with the *Batch Size* parameter we have found that *medium-large* values such as *256 samples* are well suited for Neural networks with less than 4 hidden layers. When the architecture is widened by adding more layers, only *low Batch Size* values allow the network to evolve in a controlled way.

In this section we will show the experiments carried over with the different Network Architectures. The intent behind them is to search for the ones that are better suited for the ASHRAE problem. As a reminder, its important to recall that since the training times of each network generally exceeds the 10 hours (on average), time is one of the most important factors we need to optimize. Using a grid search wouldn't prove being feasible for working on such a high space of combinations of parameter values. Our search then, is carried over manually and bases it's decisions on our personal experience.

This method of searching optimal hyper-parameters tends to be common in real case scenarios. Specially, considering that *Big Data* contains an overwhelming amount of features and samples. Which also extends the ranges of possibly valid hyper-parameters. The dataset used in this exercise can be considered an industry standard since it adds up to a total of 60 million samples (Shared between *Train* and *Test*).

5.2.1 Types of neurons to use

First to fix in the model is the kind of Neurons we will be using during the experiments. We are facing a regression problem, so the best suited kind of neurons to use are the *Linear* ones. *Linear* neurons trains generally the following function:

$$h(x) = \theta x^T + b \tag{3}$$

Being x the *n-dimensional* input of the neuron, θ the *n-dimensional* vector of weights that we are aiming to learn and b a bias *weight* that could be seen as $\theta_{n+1} * x_{n+1}$ if we would always append a 1. at the end of the input vector.

There are three principal kind of *Linear* layers:

1. **Identity**: *Identity* defines all θ values to 1. and b as 0. Clearly non useful and disposable for our case.
2. **Linear**: The purely *Linear* kind layer, applying and fitting the function defined in equation 3. Clearly it is the most used and suited when solving regression problems as our case.
3. **Bilinear**: Extends the linear description for two input vectors instead of one, replacing equation 3 with the following one:

   $$h(x) = x_1 \theta x_2 + b \tag{4}$$

   This kind of neurons could be considered for connecting some points of the *Network* architecture if we define one with two entry channels. However, this kind of experiments should not be considered as a priority in our problem, because we are only allowed to invest resources in the most promising architectures.

In the same way there are other kind of neurons that could be considered if we could transform the initial formulation of the problem, such as *Transformer Layer Neurons* [12] for *Encoder-Decoder* architectures, *Recurrent* type neurons (such as *RNN*, *GRU* or *LSTM*) or the practically impossible to use here: *Convolution Kernels*. This kind of units were mostly focused on *Computer Vision* based problems, but ultimately are starting to make an entrance in other *Machine Learning* fields as the *Natural Language Processing* [13] where is possible to reinterpret some input kernels.

With all this information the final decision has been to fix *Linear Neurons* as the kind of layers where focus the research. We are aware that exists other kinds of architectures that could be experimented giving possible good results, such as, layers of *Recursive Neurons* where save the information about not only one moment of time, but last hours or days information about the weather. However, as the resources required by each training are expensive we need to take decisions about give priority only to the most promising architectures, that are in this kind of regressions the *Linear Layer* based ones.

5.2.2 Depth of the Network and Activation Functions

Once the definition of the kind of Neurons to use is done, there is specially important to define the Depth of the network as well as which activation function we will use. In this section we will show how equivalent kind of architectures using different activation functions are better or worse suited for the resolution of the *ASHRAE* problem, making different experiments and iterating at same time over the depth of the network. Concretely we will use *ReLU* and *Sigmoid* activation functions, and iterate from *Multilayer Perceptrons* with a single hidden layer to five hidden layers *Perceptrons*.

5.2.2.1 *ReLU* Based Architectures

Rectified Linear Units [14] are between the most efficient non-linear activation functions used nowadays. *ReLU* function, as its names claim, propose a Rectified Linearity (Linear output on positives, constant in negatives) and is defined as follows:

$$f(x) = min(0, x) \tag{5}$$

By their computational efficiency and results, these units have proven to be an extremely good approach when facing a big amount of *Neural Network* problems, such as regression, classification or even in image generation problems for *Computer Vision.*

With the purpose of testing how *Neural Networks* architectures using *ReLU* activation are capable of solving the *ASHRAE* problem, we have trained five different models by no more than 15 hours each one (Non working models were early stopped and re-parametrized). Table 5 shows the *Test* score obtained by each model, and the definition of each entrance is given below:

ReLU activation models score	
Model	*RMSLE* Loss (Over *Test*)
1 Hidden Layer	3.738
2 Hidden Layer	2.174
3 Hidden Layer	1.639
4 Hidden Layer	1.651
5 Hidden Layer	1.622
6 Hidden Layer	(Converges to a constant)

Table 5: *RMSLE* score obtained over *Test* data by 5 different models using ReLU activation functions. Entrances defined in enumeration below.

1. **1 Hidden Layer**: *Multilayer Perceptron* with 1 *Hidden Linear Laye*r of size $InputSize * 2$, trained with $BatchSize = 256$. One output Neuron (also with *ReLU* activation).
2. **2 Hidden Layer**: *Multilayer Perceptron* with 2 *Hidden Linear Layer* of sizes $(InputSize, InputSize/2)$, trained with $BatchSize = 256$. One output Neuron (also with *ReLU* activation).
3. **3 Hidden Layer**: *Multilayer Perceptron* with 3 Hidden Linear Layers of sizes $(InputSize * 2, InputSize, InputSize/2)$, trained with $BatchSize = 256$. One output Neuron (also with *ReLU* activation).
4. **4 Hidden Layer**: *Multilayer Perceptron* with 4 Hidden Linear Layers of sizes $(InputSize * 2, InputSize, InputSize/2, InputSize/4)$, trained with $BatchSize = 32$. One output Neuron (also with *ReLU* activation). Higher batch sizes practically precluded the correct training making the network converge to a constant value.
5. **5 Hidden Layer**: *Multilayer Perceptron* with 5 Hidden Linear Layers of sizes $(InputSize * 4, InputSize * 2, InputSize, InputSize/2, InputSize/4)$, trained with $BatchSize = 16$. One output Neuron (also with *ReLU* activation). Higher batch sizes practically precluded the correct training making the network converge to a constant value.

As seen in table 5, there is a relevant difference on the results between lower than three Hidden Layer models and the ones with three or more *Hidden Layers.* From this point onward models obtained extremely similar results and the small variance between them could be associated to the implicit randomness inherent to this kind of approaches. There is also an important difference between the difficulties and resources consumption of training the three first models (1, 2 and 3 *Hidden Layers*) than the following ones.

Due the results of this tests is possible to conclude that the best found approach when using architectures based on *Linear Layers* and *ReLU Activation*, is to use a model of 3 *Hidden Layer* with sizes: $(InputSize * 2, InputSize, InputSize/2)$ followed by a *ReLU* activated output Neuron. This model is able to fit well with a *256 samples Batch Size* being able to complete each epoch in approximately 10 minutes as a mean (When working over a *Nvidia GTX 1060 GPU*).

5.2.2.2 *Sigmoid* Based Architectures

Sigmoid function is one of the most popular activation functions, widely used since the first *Perceptron* definitions. It is the most recommended function to use as output in *Binary Classifiers* as well as highly recommended as activation function for *Hidden Layers* on regression problems. It is defined as follows:

$$sig(t) = \frac{1}{1+e^{-t}} \tag{6}$$

As before, with the purpose of testing how *Neural Networks* architectures using *Sigmoid* activation are capable of solving the *ASHRAE* problem, as well as compare it with *ReLU* activation based models, we have trained another five different models (by no more than 15 hours each one). Table 6 shows the *Test* score obtained by each model. The definition of each model inherits from the one seen for *ReLU* activation analysis:

Sigmoid activation models score	
Model	*RMSLE* Loss (Over *Test*)
1 Hidden Layer	(Converges to a constant)
2 Hidden Layer	(Converges to a constant)
3 Hidden Layer	1.620
4 Hidden Layer	1.632
5 Hidden Layer	(Converges to a constant)

Table 6: *RMSLE* score obtained over *Test* data by 5 different models using *Sigmoid* activation functions. Entrances equivalent to the ones defined in table 5

As can be seen, results obtained when using *sigmoid* activation functions were extremely similar to the ones obtained in *ReLU* cases. In spite of there were models a bit more difficult to converge (1 and 5 *Hidden Layers* were not able to it, even changing the **Batch Size** and starting *Learning Rate*). One more time is possible to determine that 3 *Hidden Layer* model is the best possible approach due it is quite close to 4 *Hidden Layer* results, but being more shallow (therefore, more resistant to overfitting and simpler). Generally, in cases where two different models obtain same or equivalent results, best approach is to take simplest one.

5.2.3 Mixed architecture

Given the last results obtained, we can determine that best possible approach is to use a 3 *Hidden Layer* model, but there is not a clear difference between using *Sigmoid* or *ReLU* functions. Figure 1 shows a comparison between the convergence process of both architectures.

As seen, there are small differences between boths activation functions, each one giving his strengths and weaknesses in different fields. While *ReLU* activated model starts to converge faster and obtaining a better *Pearson Correlation* value *Sigmoid* activated model is more stable and obtains a better *RMSLE* Loss value over train. Moreover, *Sigmoid* activated model shows a more stable behaviour, producing a less noisy evolution over the validation set (Ligh-Orange line behind the Dark one).

Given that both are very similar in final results, and each one of them give some special strengths and weaknesses, there is possible to imagine that a model combining both could extract the best of them. Therefore is important to, before concluding try to generate some experiments in order to see how a mixed model combining both would work. Table 7 shows the result of these combinations:

As shown here, there are no relevant differences in final results when using mixes of *ReLU* and *Sigmoid* activation. Due this fact, best possible approach is to use only an unique activation function, which seeing

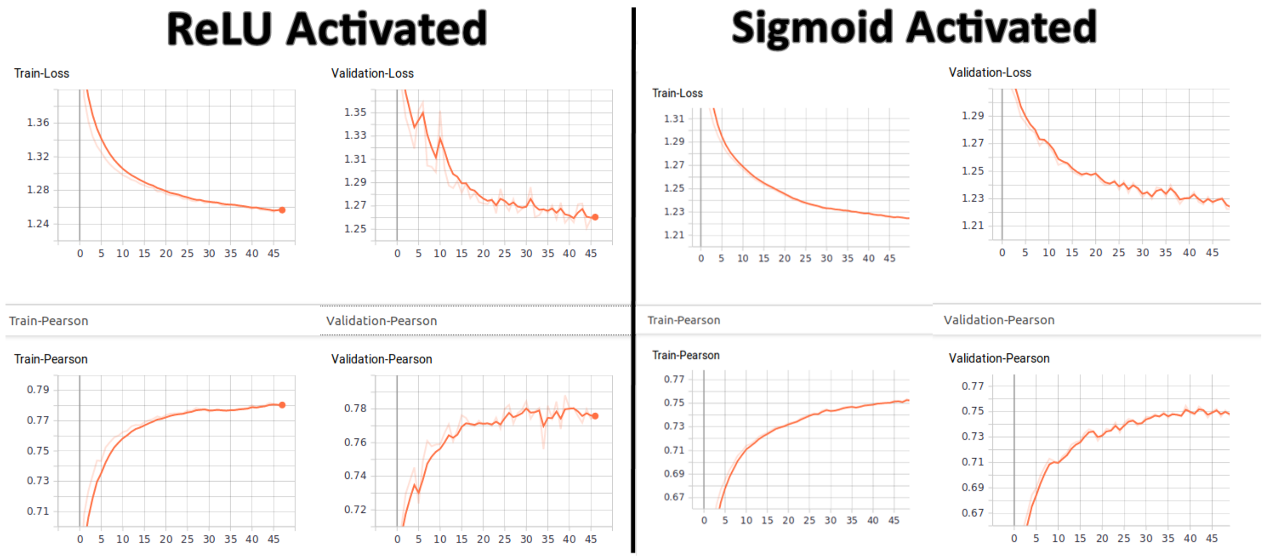

Figure 1: Comparison between *RMSLE* Loss decreasing and *Pearson Correlation* between 3 *Hidden Layers* model using *ReLU* and *Sigmoid* activation functions respectively. X axis refers to runned epochs while Y axis refers to the *RMSLE* Loss or *Pearson Correlation* value. Dark-Orange line is a .6 smoothed version of the real one, shown as the Light-Orange line behind.

Sigmoid activation models score	
Model	*RMSLE* Loss (Over *Test*)
Fully *ReLU*	1.639
Fully *Sigmoid*	1.620
Sigmoid-ReLu-ReLU	1.635
ReLU-Sigmoid-ReLU	(Converges to a constant)
Sigmoid-Sigmoid-ReLU	1.676

Table 7: *RMSLE* score obtained over *Test* data by 5 different models of 3 *Hidden Layers* using combined *ReLU* and *Sigmoid* activation.

results of the table such as figure 1 should be the *Sigmoid* activated model since is the one with more stable results on validation curves.

5.3 Final Decision

After performing the experiments, We are sure to determine that the best model we have tested is composed of 3 hidden layers with *Sigmoid* activation in *Hidden Layers* and *ReLU* in the output layer. The best hyper-parameters found for the training is to use a *Adaptive Learning Rate* algorithm such as *Adam*, with a medium *Batch Size* (*128 samples*). Also is important define *RMSLE* as *Loss* function in order to optimize the competition score. Finally we have decide to feed the *Network* with the 95% (*19 million samples*) of the *Train* data while using 5% for validate during training (*1 million samples*). Always shuffling the sets before feeding the network in order to replace the pre-existing preordering.

6 Future Work

By the limited extension of the present work and the extensive resource consumption of each experiment, there are a some promising configurations that have had to be left out of the research. The justification for this is that we needed to center our endeavors and resources only in the most promising ones. We propose the following ones as the most important future researches in order to continue the work accomplished during the subject:

1. **The exhaustive research through the optimal number of neurons on each layer**: During the shown work we have decided to fix these parameters in an balanced commonly working way. We have defined architectures which expands the number of neurons by some factor over the input neurons, in order to maximize the initial amount of learnable information about each parameter, and start decreasing progressively until ends with a final unique output value. Searching for a better approximation of the number of neurons to use in each layer should be an important research field for future works about this problem.
2. **A more extensive search through other kind of activation functions**: Some activation functions like *tanh* or *softplus* or *leaky ReLU* are also promising functions that were discarded in this work iteration. The intention for this was to avoid overflowing the extension of the present work. Researching how they damage or benefit the model generation would be a nice starting point for the future.
3. **Training secondary models for estimating the values of the missing data**: As seen in tables 2 and 3 in comparison with the table 1 there are some highly related variables that could enhance greatly the performance of the model, but have had to be deleted or poorly estimated because more than the 50% of them were missing values. Generate a great model for estimating their values in a non linear way when it is missing could include highly valuated information in the network, provoking great improvements in the estimations.
4. **Construct a *Recurrent* based path in the network architecture for encoding temporal information**: It's a fact that same building will have clear tendencies on his consumption and we could encode this knowledge in *Recurrent Neuron* states. In the same way, also the encoding of the weather progression could introduce a valuate information about the energy consumption (Is the first hour it is raining? or has not stopped in last 3 days?). There is possible to construct a two (or more) paths structured, combined by *bilinear* layers or concatenation of outputs which would encode and take profit from this information.
5. **Research through SGD based optimization techniques**: Since there was an strict requirement about the extension of the work force to optimizing the investment of resources, *Adam* optimization

technique has been to be fixed, due its faster optimization and aiming to avoid extensive parameter researches. Searching about this one and others more recent optimization techniques could also be an important way for facing future researches.

7 Conclusions

During this work we have seen the development of a standard *Kaggle* competition. We've gained intuition about how a workflow for good experimentation works in the specific case where you have yo deal with enormous amount of samples. Factors such as parameters, architectures and data optimization has been faced and analyzed. However the main aspects that we wanted to approach correctly are improving execution times and the optimization of resources. That was about deciding which are the most important and promising experiments to execute, in a clever way, when the cost of each one could exceed the 10 hours of computational time. Many of our design decisions were based on what looked to be more promising at the time. Since, again, it takes large spans to measure how well is performing a particular configuration on the problem, them our view scope is very limited. We have considered optimal focusing our efforts in a smaller set of structures rather than exploring many different and particular network configurations, but doing so very shallow. Also, the neural activation functions and methods applied correspond with the ones studied in the subject. Linear functions yield a deeper intuition since their behaviour is more predictable than others. Exploring a path more related to the contents of the subject was a conscious choice that aimed to

We have learned how to analyze the data and how to deal with it when feeding a *Neural Network*: *Standarization* methods, *One Hot Coding*, *date* and *time* information, missing data and correlation analysis. For each case, we have correlated this information in order to fix it as early as possible, pondering which potential affectations could it have to the training.

Moreover, we have done a bounded but precise research through different *Network* architectures and parameters. Our main focus was specially on finding the best *Batch Size* parameters, depth of the *Network* and activation functions. The methdology includes accurate explanations over the parameters that have been fixed why we have selected them (Validation partitions, kind of *Neurons* to use, optimizer or *Loss* function). And finally we have implemented all this work in a *Pytorch* based package that can be found annexed to this report.

In terms of results, we have obtained a 1.620 final score in the best of the cases. This corresponds to being above the 21,76% of participants (the best score has been 0.930 and mean score 1.593). It is a more than acceptable starting point taking into account that we have condensed our research to just *Regression Neural Networks* using *Linear* functions in the *Neurons* and applying a stop rule when training time exceeds the 10 hours of computing. It's also important to mention that we are using just the power of a typical student modest house *PC* and a *Nvidia GTX 1060 GPU* instead of a research supercomputer as the Mare Nostrum, our hardware is limited.

References

[1] Kaggle, ASHRAE Notebook Researches.

Available at: `https://www.kaggle.com/c/ashrae-energy-prediction/notebooks`

[2] Macedo, Crisliano. 'ASHRAE - Start Here: A Gentle Introduction'.

Available at: `https://www.kaggle.com/caesarlupum/ashrae-start-here-a-gentle-introduction`

[3] Kaggle, Leaderboard.

Available at: `https://www.kaggle.com/c/ashrae-energy-prediction/leaderboard`

[4] Kaggle, ASHRAE Dataset.

Available at: `https://www.kaggle.com/c/ashrae-energy-prediction/data`

[5] Kedar Potdar, Taher S. Pardawala, Chinmay D. Pai, 'A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers'.

Available at: `https://www.researchgate.net/publication/320465713_A_Comparative_Study_of_Categorical_Variable_Encoding_Techniques_for_Neural_Network_Classifiers`

[6] Grover Prince, '5 Regression Losses All Machine Learners Should Know'. Online Resource.

Available at: `https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-kn`

[7] Robbins, Herbert and Monro, Sutton 'A Stochastic Approximation Method' on 'The Annals of Mathematical Statistics', Vol. 22, No. 3. (Sep., 1951), pp. 400-407.

Available at: `https://pdfs.semanticscholar.org/34dd/d8865569c2c32dec9bf7ffc817ff42faaa01.pdf`

[8] Krefer, J. and Wolfowitz, J. 'Stochastic Estimation of the Maximum of a Regression Function'

Available at: `https://projecteuclid.org/download/pdf_1/euclid.aoms/1177729392`

[9] Konyushkova Ksenia and Sznitman Raphael 'Learning Active Learning From Data'.

Available at: `https://papers.nips.cc/paper/7010-learning-active-learning-from-data.pdf`

[10] P. Kingma, Diederick and Lei Ba, Jimmy 'A Method for Stochastic Optimization' at ICLR 2015.

Available at: `https://arxiv.org/pdf/1412.6980.pdf`

[11] Goodfellow Ian, Bengio Yoshua, Courville Aaron. 'Deep Learning' (2016), Chapter 8.

Available at: `http://www.deeplearningbook.org/contents/optimization.html`

[12] Vaswani Ashish, Shazeer Noam, Parmar Niki, Uszkoreit Jakob, Jones Llion, Gomez Aidan N., Kaiser Lukasz and Polosukhin Illia. 'Attention Is All You Need'.

Available at: `https://arxiv.org/pdf/1706.03762.pdf`

[13] Collins, Michael and Duffy, Niggle. 'Convolution Kernels for Natural Language'.

Available at: `http://papers.nips.cc/paper/2089-convolution-kernels-for-natural-language.pdf`

[14] Hinton, Geoffrey E. and Nair, Vinod. 'Rectified Linear Units Improve Restricted Boltzmann Machines'.

Available at: `http://www.cs.toronto.edu/~fritz/absps/reluICML.pdf`