Bachelor's Thesis Computing Science

Radboud University Nijmegen

A feasibility study on analyzing and predicting client-side energy consumption of web browsing

---

Author:
Stefan Weijers
s1032454

First supervisor/assessor:
Dr. Bernard van Gastel

Second assessor:
Dr. Twan van Laarhoven

August 22, 2023

Acknowledgements

Firstly, I want to express my gratitude towards professor Bernard van Gastel for his support during the project, it was ambitious, but we got through it. I also want to thank my friend Wim Selles, for providing support when I struggled to figure out JavaScript.

Finally, I want to thank my family, my friends and my girlfriend for supporting me from the start and always believing in me, even when I doubted myself.

Abstract

This paper attempts to prove the feasibility of predicting the client side energy consumption of loading webpages on Google Chrome, Mozilla Firefox and Apple Webkit. We achieve this by collecting energy consumption data and a collection of features of the measured websites and then training a neural network using the energy consumption as classes, treating this as a classification problem. We use K-Fold cross-validation in order to validate the neural networks performance, showing us that while changes are needed when it comes to features and data amount, it is in fact possible to predict this energy consumption with reasonable accuracy.

Contents

Chapter 1

Introduction

In today's trend of sustainability, we want to find ways we can improve our energy consumption when doing everyday tasks. Web browsing, among others, stands out as one of the most prevalent activities performed by users. In 2022, China alone had over 1 billion users using the internet, and the US had 307 million. [1]. Consequently, users are engaged in various online activities, such as browsing the web, checking their emails, looking up the weather or browsing through news websites. For instance, the website ad.nl, the website of the Dutch newspaper "Algemeen Dagblad", is the 11th most visited website in the Netherlands according to data published by DataForSEO.[2]

This research aims to establish a proof of concept of estimating the client-side energy consumption of loading a website, by making use of abstract metrics and a neural network. Any metric we use should be easily adjustable for new browsers and hardware. In order to gather data for this purpose, we limit ourselves to the Top 1000 websites by traffic in the Netherlands, as published by DataForSEO.[2] The browsers we will be analyzing are Google Chrome, Mozilla Firefox and Apple Webkit. We chose these three browsers because of their development history and their technical differences. This, in all, leads us to the following research question:

> ***Is it feasible to predict client-sided energy usage of loading a webpage?***

In order to establish a compelling proof of concept, this research will develop a measuring methodology to evaluate client-side energy consumption on desktops and then generate a dataset by automating this method on the top 1000 websites mentioned before, making use of GitLab CI/CD. Then, this dataset will be used to train a neural network, which can then serve as a proof of concept, provided the accuracy meets the desired criteria.

In order to tackle our research question, it is important we first discuss important considerations that have to be made with respect to validity of our research (see Chapter 2). Keeping these considerations in mind, we

establish our methodology and prediction methods making sure we adhere to our conditions established in Chapter 2 (see Chapter 3 and 4). We then discuss our results and possible improvements, giving suggestions on what to improve in order to achieve a higher accuracy (see Chapter 5. In chapter 6 we explore related work, comparing our conclusions to what other researchers achieved and elaborating how we expand on their research. After this, we argue how our research can be applied to daily use, showing why our research is important (see Chapter 7).

Chapter 2

Validity

In this Chapter, we will be discussing the challenges with respect to validity. We attempt to prove that our research is, in fact, a valid method of answering the research question posed by us in Chapter 1.

2.1 Browsing emulation

In order to answer the research question, we need to obtain data on client-sided energy consumption. However, since we want to automate this process, it will be running on a server. This already leads us to a problem; There is no screen to display the page on. Given we want to run the data-collection script on this server, it is crucial that what we use accurately depicts a user's behavior when loading a webpage, in a display-less environment. In this section, we will justify that our implementation is an adequate method for depicting this behavior.

2.1.1 Playwright

Playwright[1] is a library developed by Microsoft with the purpose of end-to-end testing for web apps. It provides a headless browser instance, allowing automated loading of webpages through scripting. Playwright exclusively supports are Google Chrome, Firefox and Apple Webkit. A headless browser instance refers to a browser without any graphical interface. With Playwright, we can efficiently emulate browser behavior and automatically load our webpages. In Playwright, you begin by creating a new "page" and then proceed to navigate to the desired site. This process mirrors the typical user experience when opening a new tab and navigating to a website. Therefore, we believe using playwright is an accurate depiction of this part of the browsing process.

[1] playwright.dev: Playwright, developed by Microsoft

2.1.2 Rendering

Playwright has a lot of built in functions that can be used on loaded pages. Two relevant options for the purpose of rendering the page were `.screenshot({fullPage: true})` and `.pdf`. We decided against `.pdf`, as making a pdf of a webpage is an entirely different process to the user loading up the website. In order to load a page, we have to use `page.goto()`. This loads the webpage and renders the webpage in the viewport as defined. We defined the viewport as 1920x1080, as this the most common desktop resolution worldwide. [2]. If goto is given the option `waitUntil: networkidle`, it will only finish when every element of the page is loaded. When calling `.screenshot({fullPage: true})` over a page, playwright manually scrolls through the page, making screenshots of every slice and then stiches these together. We decided to only use `goto` using the option "networkidle", as we believe this is the closest to what a normal user would do when loading a webpage. Using `screenshot` would result in unrepresentative CPU usage relative to a normal user, which would make our method unrepresentative.

2.2 Exclusion and Measurement

With so many steps to keep in mind, it is also important that steps are excluded from our final energy measurement. Starting up the browser, for example, should not be included in the final measurement. A comprehensive list of all excluded processes include:

- Browser launch
- New Tab Launch
- Energy calculation
- Screenshot verification
- Idle energy consumed by the machine

More information on how these processes are excluded from measuring can be found in Chapter 3

2.2.1 Attributes

In order to predict the energy usage, we need attributes in our dataset. We will be discussing all 6 attributes we are using, explaining why we chose them for our final dataset.

[2] As retrieved from statcounter: https://gs.statcounter.com/screen-resolution-stats

Time

We make use of the time it took the browser to load the page, we use this because of the implication that if a website took long to load, it will probably cause a higher energy consumption.

Body size

We also make use of the response body size, this will give us an indication of the size of the webpage that is loaded by the browser, a larger body size should imply higher energy consumption, as there is a bigger file to load.

Scripts

Even though the content of scripts within a webpage can vary massively, we believe the presence of scripts can still indicate either higher or lower energy consumption. For example, lack of images on a webpage but presence of scripts could imply the images are loaded through a script. Therefore we decided on including the amount of scripts in the data provided to the neural network.

Images

Images on a webpage could, in theory, lead to higher energy consumption when loading a webpage, since a picture needs to be rendered and the browser has to download the pictures and then display them.

DIVs

The amount of DIVs on a page should give us an indication of how many elements there are on the webpage, which could help the neural network predict the energy usage.

Buttons

When manually analyzing the gathered data, there was a big variety in amount of buttons between webpages. Therefore, we decided on including this data in the data we provide to the neural network.

2.2.2 Idle energy consumption

Something important to keep in mind is that a computer has background processes that also take up energy. These processes and their energy consumption should by all means be accounted for. We want to make sure the idle energy consumption is stable for at least 10 seconds before measuring. This way we assure that it is stable during measurement and we can account

for it in the final calculation. Technical details on how this is achieved can be found in Chapter 3

2.3 Prediction

The second step of answering our research question involves researching methods for predicting this energy usage. One of these methods is using neural networks in order to predict the energy consumption

2.3.1 What is TINN?

TINN (Tiny Neural Network), is a 200 line dependency-free neural network written in C, developed by Gustav Louw[3]. It has one hidden layer, and uses sigmoidal activation for its neurons. Sigmoidal activation is one of many ways to calculate whether a neuron activates or not. The exact mathematical formula can be found on sites such as wikipedia[3], but in TINN's source code it is defined as follows:

```
// Activation function.
static float act(const float a)
{
    return 1.0f / (1.0f + expf(-a));
}
}
```

A TINN network takes arrays of floats in order to train itself. It performs this training process using forward and backwards propagation. Backwards propagation is the process of performing a backward pass through the model to adjust neuron weights according to the target prediction and actual prediction. Forward propagation is simply giving the network data and having it generate a prediction. By going back and forth enough times, we train the neural network, which gives us a trained neural network that, in theory, predicts the energy usage accordingly.

2.3.2 Why TINN?

In order to answer the research question, we need to obtain a proof of concept that some framework is capable of predicting the energy usage of webpages on the client-side. Given TINN's minimalism, it is ideal to research if it is, in fact, feasible to predict energy usage with TINN. If it is possible with TINN's tiny architecture (only one hidden layer), or at least promising, it is very plausible that a more fleshed out neural network library would be able to more accurately predict the energy usage. However, if TINN proves to

[3]Wikipedia page for Sigmoid Function

be a reliable library for prediction, it can even be used for applications in embedded systems. More info on this can be found in Chapter 7.

2.3.3 Difficulties

While TINN offers a lightweight and efficient solution for neural network implementations, it also comes with some limitations. Firstly, due to its focus on being compact, TINN lacks some advanced features present in more comprehensive neural network frameworks, like having multiple hidden layers, being able to predict actual values, making use of loss functions, supporting mini-batching, among others. TINN is limited to class prediction, using neuron activation. This, we believe, is adequate for providing a feasibility proof. If TINN can predict the range the energy usage will be in, it would be a positive result.

2.3.4 Validation

In order to validate that our neural network is adequate, we need to apply some validation technique to our network. We decided on using K-Fold cross-validation for this purpose, given its simplicity and reliabilty.[4][5] K-Fold validation splits the dataset into a given k amount of "folds", which are basically subsets of the dataset. For our dataset, we split into 10, given its size. We iterate over the 5 folds, where every iteration our testing set is the fold we iterate on, and the training set is the remaining 4 folds. We keep track of two performance metrics in every iteration; the average error, and the percentage that the network guessed correctly. At the end, we take the average of every iterations performance metric in order to assess the performance of the network. In order to prove feasibility, we want an accuracy of around 50%. Exact implementations can be found in Chapter 4.

Chapter 3

Measuring

3.1 Testing Setup

In order to measure the energy, we make use of a specialized setup using an ODROID-H3+ and a INA260. The INA260 sits between the power supply of the ODROID and the ODROID itself. The ODROID-H3+ is a x86 64-bit single board computer, which we utilize as a server. Specifications can be found in Appendix C.

A note on the INA260

We want to make sure that the INA260 is a proper method of measuring energy usage so it is important to confirm the accuracy. From the datasheet provided by Texas Instruments[1], we can derive that the maximum error is 0.15%, while it usually is around 0.02%. We believe this a sufficient percentage for us to derive valid conclusions from the output of the INA260.

The server that runs on the Odroid is a Debian server that runs on the latest version. The server makes it's energy data available by responding to HTTP post requests. The information is returned in JSON format, an example can be found in the appendix.

3.2 How do we actually measure

In order to measure the energy usage of a browser rendering a webpage, we have to consider many factors involved in the process. A few of note:

- The browser starting up should not be part of the measuring process.
- Processes in the background might make the data invalid

[1]Datasheet found at https://www.ti.com/product/INA260

- Idle consumption of energy should not be included in the total energy consumed.

The browser is started before any measurements are done. We also make sure to make the program sleep after starting the browser in order to let the CPU come to a 'resting' state. After trial and error, we found that 10 seconds of sleep was enough to keep the idle energy consumption consistent. In order to make sure that our data is not affected by background processes, we make sure to do idle energy checks before and after the render is made, to make sure the idle consumption is consistent, this will be described in more detail later on. In order to make sure that idle energy consumption is not included, we make use of the number of measurements between the measurement captured before the render and the measurement captured after the render. Before rendering, we capture two measurements of the (hopefully) idle energy consumption at that point. In the system, this is called `electricity_consumed_current`. We then take the difference in amount of measurements between these two points, and then multiply the idle energy consumed by this amount. This should give us the total amount of idle energy consumed by the machine during the render. If we then simply substract the overal total found by this number of idle energy, we get the energy consumed by the browser.

3.2.1 Idle Energy Consumption

As mentioned in Chapter 2, we want to make sure that the idle energy consumption is monitored throughout the measurement. This is especially important because fluctuations in background consumption can wildly affect results. If idle consumption were to rise or drop massively during the rendering process, the results would be skewed, and end up higher or lower. In order to avert this risk, we make sure the idle is stable for 10 seconds before doing our measurements. This is done as following:

```
async function stabilize(baseline){
  var attempts = 0;
  var last_bound = Date.now();
  while ((Date.now() - last_bound) < 10000){
    await sleep(100);
    var idle = (await getEnergy(energyStats)).cur

    if (Math.abs(idle - baseline) > 1){
      last_bound = Date.now();
    }
    if (attempts >= 600){
      return -1;
    }
```

```
    attempts = attempts + 1;
  }
  return 0;
}
```

This function checks every 0.1 seconds if the idle has gone out of bounds, if it does not go out of bounds for 10 seconds, we call it stable and continue with the render.

This returns -1 if the idle energy did not stabilize within 60 seconds, and 0 otherwise. This way, we can assure that the idle energy consumption is stable before rendering, making us able to use it. We do one extra check after rendering, to make sure that the idle consumption after is not too far from the idle consumption before rendering.

3.2.2 Data Gathering

The main interest is of course the energy consumption. However, in order to give the neural network enough data to predict the energy consumption, we need to give it certain attributes. We keep track of the amount of scripts within a page, the amount of images, the amount of divs, and the amount of buttons. We believe these elements give a good rough indication of whats apparent on a page. Manual analysis also showed that in sample date these variables varied a lot between different pages. We also measure the time it took to load the page, and the body size of the response. An example of how a data entry looks can be found in Appendix B.

Page Content

We collect page content (SCRIPT,IMG,DIV,BUTTON) using a javascript library called JSDOM[2]. We take the page content from the webpage using the Playwright API, then we load this into a dom object using the library. Then, we can simply count how many elements that are that match the tagnames we provide. We turn this information into a dictionary, to then transform this into a JSON string using `JSON.stringify()`

```
function createElementList(document, elementlist){
  results = {}
  for (let i in elementlist){
    var list = document.getElementsByTagName(elementlist[i])
    results[elementlist[i]] = list.length
  }
  return results;
}
```

[2]JSDOM Github Page can be found at https://github.com/jsdom/jsdom

```
var content = await page.content();
const { document } = (new JSDOM(content)).window;

var elements = createElementList(document, ["SCRIPT", "IMG", "DIV", "BUTTON"])
var pageContent = JSON.stringify(elements);
```

Body size and Time

In order to measure time, we simply make use of `Date.now()` right before and after loading the page. For body size, we use a listener on page. This listener calls its defined function when the request is finished (recieved a response). We then use an inbuilt function on the response data, which gives us the body size. Of course, this can error, in which case we set it to 0.

```
var bodysize;

page.on("requestfinished", async data => {
    try {
      bodysize = (await data.sizes()).responseBodySize;
    } catch (e) {
      console.log('Failed to get body size, error: ', e)
      bodysize = 0
    }
})
```

Data structure

An example of the structuring can be found in Appendix B.1, every url is an entry in a JSON file, with its attributes being as follows:

- `screenshot`: Succesful/Failed - Shows whether taking screenshot was successful or not
- `bodySize`: Integer - Response body size in bytes
- `time`: Integer - Loading time in milliseconds
- `adjusted_total`: Float - Raw total energy consumed subtracted by the idle energy consumed
- `total`: Float - Raw total energy consumed
- `idle1`: Float - Idle energy consumption before render

- `idle2`: Float - Idle energy consumption after render
- `pageContent`: Collection - Page content as described in section 3.2.3

Of course, as mentioned before, things can go wrong during the measuring process, if this is the case the entry is changed, depending on what happened. In the case of a connection error, the url is simply paired with a simple string saying "errored" (see Appendix B.2).

In the case of the script not being able to stabilize the idle, the script ends the measurement and outputs the baseline idle used, and the last idle measured during stabilizing. An example can be found in Appendix B.3.

In the last case, the idle energy consumption after measuring differs too much from the idle energy consumption before measuring. In this case, the script outputs the site with status "fluctuated" along with the baseline idle used, the idle energy consumption before measuring and the idle energy consumption after measuring. An example can be found in Appendix B.4.

Chapter 4

Predicting

In this chapter we will describe how we attempt to assess the performance of TINN on our dataset, making use of K-Fold cross-validation. We will also describe how we can then use TINN in order to predict energy usage on websites that are not present in the dataset.

It is important to note that the base for the code used in this paragraph is taken from the example program contained within official TINN GitHub Repo.[3]

4.1 Data Input

4.1.1 Data formatting

In order to properly input our data into the neural network for training, we have a secondary output from the script which is simply a .txt file with the data in a slightly different format from the JSON file. It simply puts all data points on a single line, without labels. In order, they are: time, body size, scripts, images, divs, buttons, energy usage. (see Appendix (B.1)). However, this is not quite enough, as TINN is only capable of class prediction. Therefore, we divide the possibilities for every entries' energy consumption into 6.

- Below 10 - "A"
- Between 10 and 20 - "B"
- Between 20 and 30 - "C"
- Between 30 and 40 - "D"
- Between 40 and 50 - "E"
- Above 50 - "F"

We give them the labels A - F in order to resemble the European energy label (**Not** accurately represented)[1].

Using a python script, which can be found in Appendix D, we transform our raw data to data that the neural network can use in order to train and predict the energy usage.

4.1.2 Data object

In order to properly store the data within C, we make use of a C struct called Data. This was taken from the example provided in the github repository of TINN. [3]

```
typedef struct
{
    // 2D floating point array of input.
    float** in;
    // 2D floating point array of target.
    float** tg;
    // Number of inputs to neural network.
    int nips;
    // Number of outputs to neural network.
    int nops;
    // Number of rows in file (number of sets for neural network).
    int rows;
}
Data;
```

The input are the attributes, the target is the actual class of the attributes (the energy usage). `nips` is the number of attributes we input into the network, `nops` the amount of possible classes.

4.2 Training and Validation

4.2.1 Generating K-Folds

In order to perform K-Fold cross validation we need to generate 10 folds, where 9 are used for testing. We want to use all folds for testing at least once. We do this by looping the following code 10 times.

```
int start_index = fold * fold_size;

const Data train_fold = ndata(train_data.nips, train_data.nops, train_data.rows
- fold_size);
```

[1] More info to be found at: commision.europa.eu

```
const Data test_fold = ndata(train_data.nips, train_data.nops, fold_size);

int train_i = 0, test_i = 0;
for (int i = 0; i < train_data.rows; i++){
    if (i >= start_index && i < start_index + fold_size){
        test_fold.in[test_i] = train_data.in[i];
        test_fold.tg[test_i] = train_data.tg[i];

        test_i++;
    } else {
        train_fold.in[train_i] = train_data.in[i];
        train_fold.tg[train_i] = train_data.tg[i];

        train_i++;
    }
}
```

Where `fold_size` is the amount of rows divided by the amount of folds. We store the data within the test fold in a Data object called `test_fold`, and all other data is stored in a training dataset.

4.2.2 Training

We then train for 1024 iterations, and every iteration we anneal our learning rate by multiplying it by the anneal rate, which we set to 0.99. We shuffle our data every iteration, and then use TINN's xttrain function to train the network. This is implemented as follows:

```
for(int i = 0; i < iterations; i++) {
    shuffle(train_fold);
    float error = 0.0f;
    for(int j = 0; j < train_fold.rows; j++)
    {
        const float* const in = train_fold.in[j];
        const float* const tg = train_fold.tg[j];
        error += xttrain(tinn, in, tg, rate);
    }
    printf("error %.12f :: learning rate %f\n",
        (double) error / train_fold.rows,
        (double) rate);
    rate *= anneal;
}
```

We shuffle the train data before hand, making sure we vary our data, then we train our network using the built-in function, and then at the end

we anneal the learning rate at a rate such that at the end of all learning cycles, the learning rate is close to 0. We do this because if the learning rate stays too high, the network fails to properly train and the accuracy will drop. This was discovered using trial-and-error. We discovered that an anneal rate of 0.99 and 1024 served as a good amount, since this gave us the highest accuracy that we could find.

4.2.3 Validation

In order to validate the trained network, we use the network to predict our test fold, to then calculate the total error, and the percentage of correct guesses. We store these for every fold, to then calculate the average accuracy and average error of all folds, which gives us an adequate insight into the performance of TINN on our dataset. The validation is done as follows:

```
int correct = 0;
float pd_error = 0.0f;
for (int i = 0; i < test_fold.rows; i++){
    const float* const in = test_fold.in[i];
    const float* const tg = test_fold.tg[i];
    const float* const pd = xtpredict(tinn, in);

    int correct_pd = correct_pred(tg, pd, test_fold.nops);
    if (correct_pd == 0) {
        correct++;
    }
    json_print(json_out, in, tg, pd, test_fold.nips, test_fold.nops);
    pd_error += error(tg, pd, test_fold.nops);

}
printf("Average error: %.12f - Percentage correct: %f \n",
(double) pd_error / test_fold.rows, (double) correct / test_fold.rows);
errors[fold] = (float) pd_error / test_fold.rows;
accuracy[fold] = (float) correct / test_fold.rows;
```

The `correct_pred` function simply compares the prediction to the target and returns 0 if the prediction is correct, and -1 otherwise. We simply count the amount of correct predictions and calculate the accuracy by taking the percentage of correct guesses compared to the total amount of entries. We also take the average error.

The final calculation of the average is done as follows:

```
float e_sum = 0;
float a_sum = 0;
```

```
for (int i = 0; i < NUM_FOLDS; i++){
    e_sum += errors[i];
    a_sum += accuracy[i];
}

printf("Avg Error: %.12f - Avg Accuracy: %.12f \n",
(double) e_sum / NUM_FOLDS, (double) a_sum / NUM_FOLDS);
```

We sum all the errors and accuracies and then simply take the average of all of these.

This way, we get a proper performance metric of our neural network.

Chapter 5

Results

5.1 Energy Consumption

When looking at our energy data, we can draw conclusions about the energy use when comparing the different browsers we measured. Chrome seems to use the least energy, with 15,74 kWh adjusted average used for loading a webpage. Firefox comes second, with 21,55 kWh used. Webkit is last, with 35,84 kWh used. It should be noted that the average non-adjusted usage for Firefox was higher than Webkit and Chrome, although this does not mean anything for our results, given we remove the idle energy from the total. The results as mentioned can be viewed in Figure 5.1. Something worth noting was that on average across all browsers, YouTube ends up being one of the most energy intensive websites out of all 1000 websites, with an average of 79,28 kWh used. This is highly likely to be the result of many video previews and many images being loaded on YouTube's home page.

5.2 Network Accuracy and Error

When evaluating TINN's performance, we used K-Fold cross validation. This gives us an average error and an average accuracy. We plot these for every browser in Figure 5.3 and 5.2. As visible, Firefox is the most predictable, with TINN struggling the most for Webkit. Possible reasons for this occuring will be discussed in section 5.3. We can also deduce that the average accuracy for Firefox is just over 40%. This shows that it is, in fact, slightly feasible for a neural network to predict the energy consumption of loading webpages given we improve the accuracy and gather more data. While it might appear to be infeasible for Webkit, we believe it is still feasible to predict energy consumption for this browser. The reasons for this are discussed in section 5.3. In section 5.3, we will also discuss how we can improve the accuracy on TINN, or use different frameworks for better results.

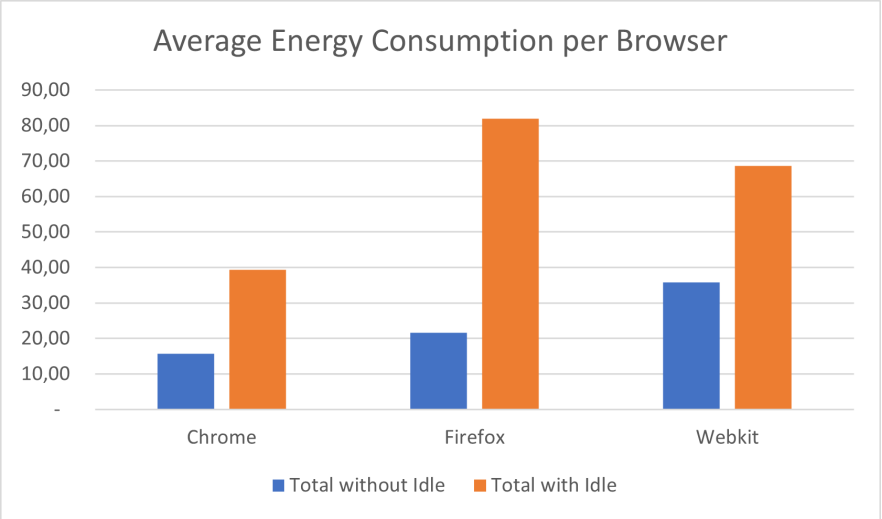

Figure 5.1: Graph displaying adjusted average total versus the non-adjusted total for all measured browsers

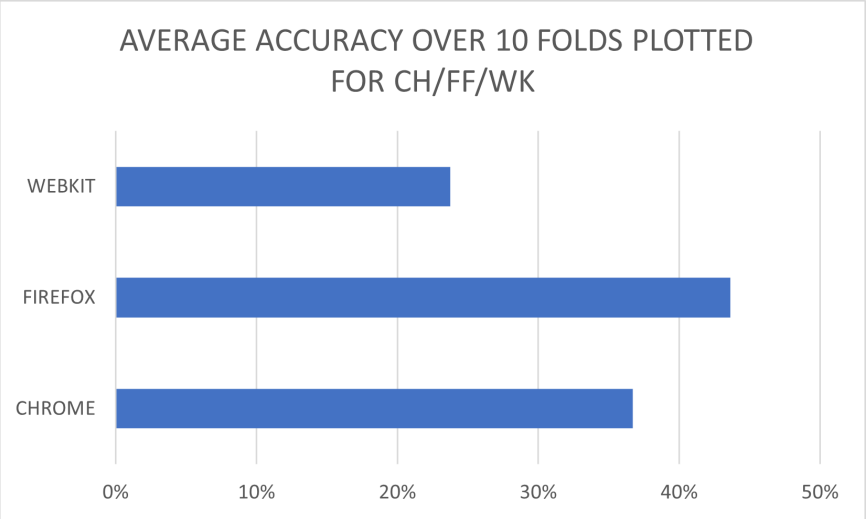

Figure 5.2: Average Accuracy plotted for Chrome/Firefox/Webkit

5.3 Improvements

5.3.1 Overfitting

By analyzing the data we gathered from the neural network, we were able to analyze how the network predicted. By keeping track of every prediction, we were able to plot the predictions compared to the actual classes. In the graphs for Chrome and Firefox in Figures 5.4 and 5.5, we can see that the neural network overfitted on the E class, as visible. Overfitting in the

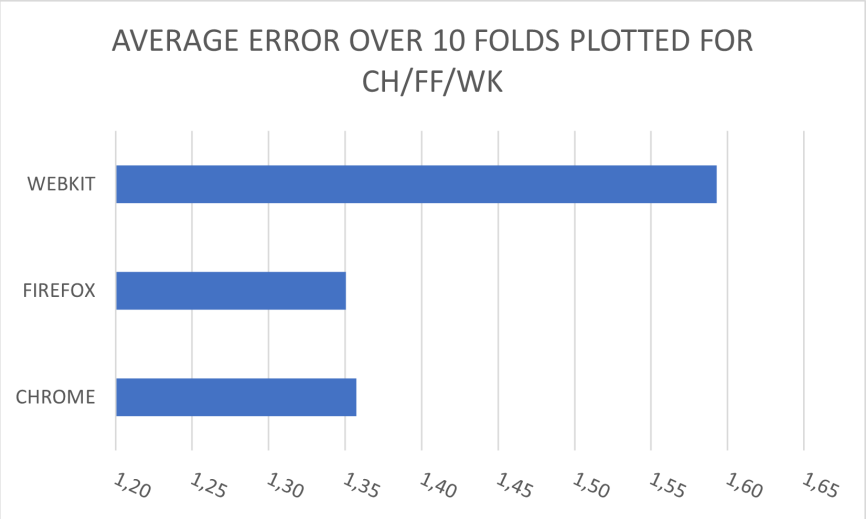

Figure 5.3: Average Error plotted for Chrome/Firefox/Webkit

context of neural network means that there is a lack of diverse data being fed to the neural network, which results in the neural network having a bias towards one class. This can be seen in the graph for Firefox too (Figure 5.5, for example, there is barely any data where the energy class is A, yet there is a significant amount of data that belongs to class E. With more data, this problem can be solved, as the neural network can train itself better on the different types of classes. Of course, it is not a guarantee that there is such power efficient websites, in which case we should aim to add more attributes to the data we provide to the neural network, which we will disclose in the next section. By adding more attributes to the data, we give the neural network more data to base it's prediction on, which should in theory improve the accuracy.

5.3.2 Optimizing runtime

The script we developed for automation includes a lot of sleeps in an attempt to let the CPU stabilize. This, in combination with the idle stabilization we describe in Chapter 3, makes it such that the time used per website ends up being somewhere between 1-3 minutes on average. This, in combination with the scope of 1000 websites over 3 different browsers, resulted in a total run-time of 51 hours during the final measurement that was done.

Because of this, we ended up with a smaller dataset than we would have liked. If the idle cannot be stabilized, we abort the measurement for that website and move on to the next. This results in less data, which of course could be resolved by simply running the pipeline again via GitLab CI/CD, but as a consequence of having a 51 hour runtime, this was not doable with

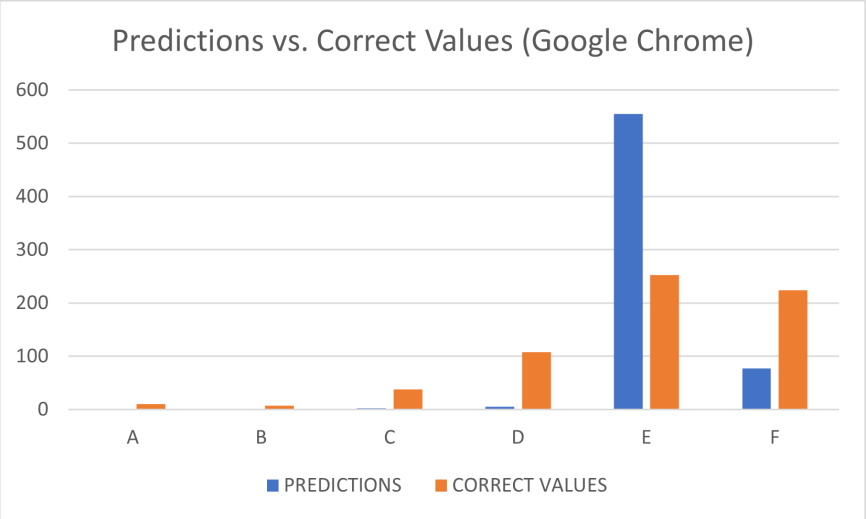

Figure 5.4: Predictions versus actual values plotted for predictions based on data gathered from Google Chrome

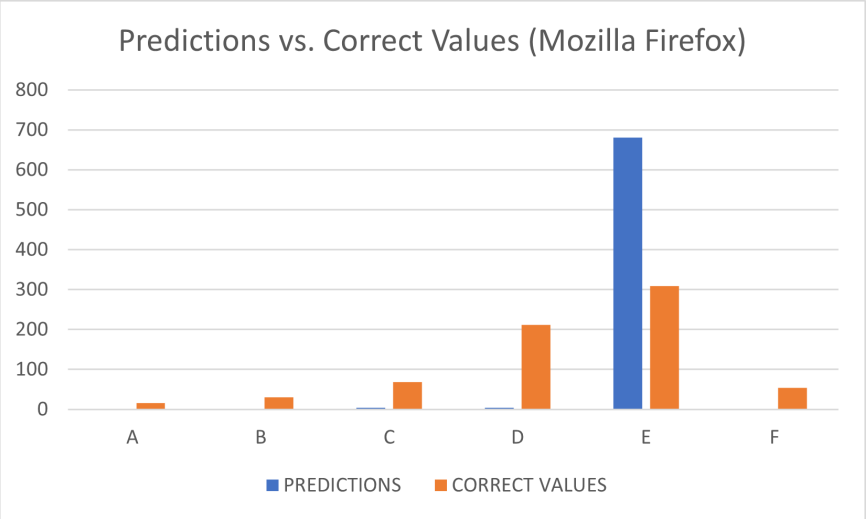

Figure 5.5: Predictions versus actual values plotted for predictions based on data gathered from Mozilla Firefox

our time constraints.

Further research should look at if these sleeps are necessary for the CPU to stabilize, or whether our idle stablization is enough. If the runtime can be improved upon by removing unnecessary sleeps, this would enable easier filling of gaps, by re-running the measurement until the idle energy consumption is stable enough.

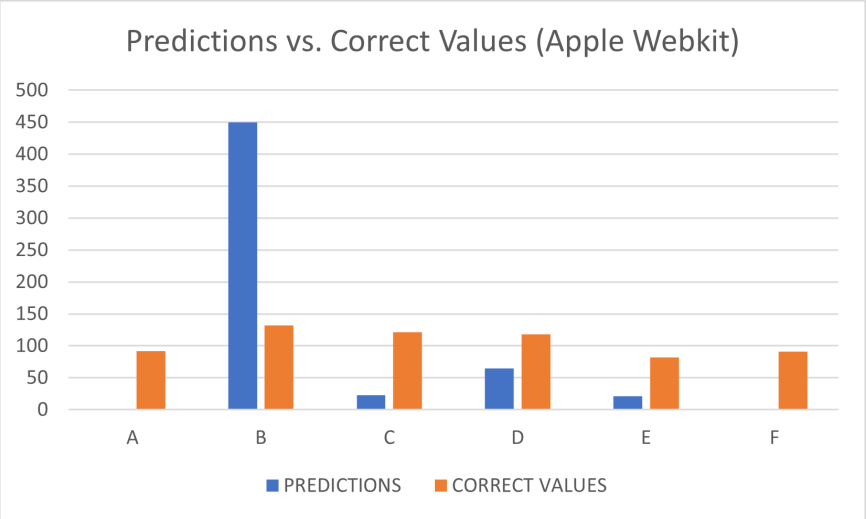

Figure 5.6: Predictions versus actual values plotted for predictions based on data gathered from Apple Webkit

5.3.3 Lack of Data

As mentioned in the previous section, further research should aim to include more attributes. Attributes that could prove to be relevant are metrics such as RAM usage, the amount of DOM nodes. Ideally, we would have liked to include these metrics, but time constraints prevented us from including these in the scope of the research.

Another issue we had during measurement is that when measuring the energy usage of WebKit, the script often struggled to stabilize the idle energy consumption as described in Chapter 3. This meant that we had around 550 entries for webkit compared to the 1000 websites we attempted to measure. If there was more data available, by measuring more websites and possibly giving the script more time to stabilize, we might have seen different results when it comes to accuracy and error. It is also visible that due to lack of data, predictions for Webkit were much more widespread, as visible in Figure 5.6 showing the neural network struggled to train itself on the limited dataset.

Body size

The way we measure response body size can be improved upon. Some websites returned a response body size below 10 bytes, which possible indicates that some websites either do not respond with the entire website, or that we accidentally caught a redirect. We, however, had enough datapoints with a body size that seemed reasonable, meaning we still could use this metric for the predictions.

5.3.4 Possible Alternatives

When looking at possible alternatives for TINN in the context of finding a neural network library for future work, there is ample options. In this section, we will delve into two specific alternatives, namely KANN and TensorFlow.

KANN, a lightweight C library for artificial neural networks[6], stands out as an alternative for those wanting to retain the lightweight aspect of TINN. KANN provides much more flexibility and more features, such as the implementation of shared weights, while retaining it's lightweight aspect. The most notable similarity with TINN is that it is both lightweight and also implemented in C.

For scenarios where the lightweight attribute is not a primary consideration, TensorFlow emerges as an alternative among widely employed machine learning libraries [7]. Developed for both Python and JavaScript, TensorFlow boasts an all-encompassing machine learning platform renowned for its robustness and user-friendly nature. It also provides pre-trained models, among other features. TensorFlow's flexibility would make up for TINN's simplicity. It allows developers to create multiple hidden layers, their own loss functions, etc. This allows for more experimentation with different prediction methods. TensorFlow also seamlessly integrates with data science libraries such as scikit-learn, which serves as more flexibility compared to TINN.

Chapter 6

Related Work

Related research on this topic can be categorized into two domains, studies aiming to solve a similar problem and studies employing similar methods. Related research aiming to solve a similar problem investigates a generalized method for energy consumption calculation, utilizing the energy usage of the data centre among other aspects related to the server-side of a webpage. In Section 6.1, this is further explained. On the other hand, research employing similar methods explores energy consumption differences between Google Chrome and Firefox

6.1 websitecarbon.com

websitecarbon.com is a website that allows you to fill in any website, which the website will then use to calculate the exact carbon emissions from that website. As mentioned in the "how does it work?" section of the website, they use 5 data points to calculate the energy usage.

- Data Transfer over the wire
- Energy intensity of web data
- Energy source used by the data centre
- Carbon Intensity of electricity
- Website traffic

The calculation is done based on research done by Chris Adams, Rym Baouendi, Tim Frick, Tom Greenwood and Dryden Williams, which was published on the website sustainablewebdesign.org.[8] They attempt to create an general formula of estimation for the carbon intensity of webpages.

In order to represent a comprehensive footprint, they define the widest system boundaries available. In order to provide greater insight, they segment the impact to sub-systems, they define these as: [8]

- Consumer device use, which is end users interacting with a product.
- Network use, which is data transferred along the network.
- Data Center use, which is energy required to house and serve data.
- Hardware Production, which is embodied energy used in the creation of embedded chips, use of data centers, use of networks, and the use of consumer communication devices.

They assign percentages to these systems, to then define a general formula for calculating the carbon footprint. Exact details on these formulas can be found on the sustainable web design website.[8] These percentages and values they use in order to create a generalized formula are extracted from research done by Anders Andrae.[9]. This research is a continuation of earlier research done by Andrae in cooperation with Tomas Elder.[10]. In this research, they use similar system definitions as mentioned before. When looking at customer device use, which is similar to what we research, they use average values in order to calculate the total energy consumed. While the scope of this research is quite broad, our is much more precise. We attempt to create a proof of concept for a method of measuring specifically browser energy usage on singular devices. In theory, we expand on this research by providing more accurate energy measurements for consumer devices. If we were able to accurately predict the energy for every device this would remove ambiguity on the energy used by consumer device use, making for more accurate estimations. More accurate estimations would then in turn lead to more accurate estimations of the carbon footprint of websites using the research done by the researchers at sustainablewebdesign.org.

6.2 Energy Wars - Chrome vs. Firefox

In research done by João de Macedo, João Aloíso, Nelson Gonçalves, Rui Pereira and João Saraiva,[11] they emplore a similar method to our research in order to figure out if either Chrome or Firefox is more energy efficient. They emplore Selenium, a similar library to Playwright in order to emulate a few common operations done by users. These are:

- Browsing YouTube
- Live Streaming on Twitch
- Browsing Social media

They use these operations and the energy consumption in order to answer the final question, this being which browser is the most energy efficient overall. To measure this energy consumption, they use Intel's RAPL in order

to monitor the energy consumption during script execution.[11]. In order to assure validity, they kill most background processes to assure preciseness. This is similar to what we attempt, however our method is more rigid and could prove to be a proper method of assuring preciseness. The research covers 3 cases. Our research expands on this by applying our script to 1000 websites. This way, we have more data to assure that the comparison is done fairly. By gathering more data, we can more accurately see the difference between the 3 browsers. It is interesting to note that de Macedo et al. conclude that Chrome can be more energy efficient, but is less consistent than Firefox, we conclude that Chrome is the most energy efficient without any big fluctuations. More data could possibly strengthen this conclusion.

Chapter 7

Applications

When it comes to applying our research, there is ample opportunities. A possible use case is using a trained neural network to create a browser extension, serving as an information tool for developers and clients to get information about their current usage. The methodology can also be applied to other scenarios.

7.1 Browser extension

As previously mentioned, we can make use of our trained neural networks in order develop a browser extension that predicts website energy usage in real-time. This extension would analyze websites as they load, by extracting data similarly to our scripts. The extension would then use the neural network to generate and display predictions, providing insights into the websites energy efficiency for developers and clients. Notably, this tool expands upon the carbon footprint calculator mentioned in Chapter 6 by offering insights for locally hosted websites, making it valuable in development environments. For clients, it offers an easier way of figuring out the energy usage of the sites they are visiting, due to the ease of use a simple browser extension brings, compared to having to entire every single url into a website like websitecarbon.com.

7.2 Other Scenarios

Our methodology can be applied in various scenarios. If we want to predict the energy cost of rendering an image in a rendering program such as Cinema 4D or Blender, then we can employ a method similar to the one defined in Chapter 3. Adjusting the data we collect and gathering a sufficient amount of data enables us to then train a neural network in order to be able to predict the energy cost of rendering an image, or even multiple frames. Predicting rendering energy usage as mentioned before only serves as an

example, however, with a sufficient amount of data, our approach can be applied to predict the energy cost of other scenarios as well.

Chapter 8

Discussion

In this paper, we established a methodology for measuring the client sided energy consumption, which we then use to gather data for the purpose of training a neural network developed using TINN.

8.1 Results

After analyzing our gathered data, we conclude that Webkit is the least energy efficient out of all browsers, using around 35,84 kWh on average when loading a webpage. Chrome is the most energy efficient, only using 15,74 kWh on average when loading a webpage. Firefox is in the middle with 21,55 kWh used on average. (see Chapter 5).

We trained 3 neural networks, one for each browser, and compared their accuracies and average errors. We concluded that for both Firefox and Google Chrome, the accuracy of our neural network is around 40% on average. For webkit however, this percentage was around 25%. By improving our data collection, or making improvements to the neural network implementation, this percentage can be improved. (see Section 5.3).

8.2 Improving the measuring process

When it comes to the gathering of data, further research should firstly take our improvements as described in Section 5.3 in mind. In order to speed up measurements, future work should attempt to optimize the measurement process, either by speeding up the idle energy stabilization process or removing sleeps from the script, as we mention in section 5.3. By implementing these improvements, more data can be gathered more consistently by simply retrying more often, solving one of the issues we mention in section 5.3.3, namely that the script often struggled to stabilize the idle often, causing lack of data. The measurement process can also be improved upon, as mentioned in section 5.3.3, future research should look into better methods for

measuring the body size of a webpage.

8.3 Improving the prediction accuracy

When trying to expand on the research done on predicting the energy usage, it is vital that data collection is improved as mentioned in chapter 5. As mentioned before, if data collection is improved, the problem of lack of data can be solved. This, in turn, solves the problem of overfitting, as the neural network has more cases to train itself with.

Furthermore, accuracy can be improved by providing the neural network with more attributes (see section 5.3.3). If more varied data can be obtained, the neural network is able to train itself to lower its average error and heighten its average accuracy.

Further research could also explore other machine learning frameworks. We limited ourselves to TINN, but other frameworks such as TensorFlow or KANN could serve as great alternatives for TINN. (see Chapter 5 section 5.3.4 for more details) More experimentation should reveal what framework is suited best for solving the problem of predicting client-sided energy usage.

8.4 What's next?

8.4.1 Expanding existing research

Our research can be used to expand upon already existing research. As mentioned in Chapter 6, we expand on the research done by Chris Adams, Rym Baouendi, Tim Frick, Tom Greenwood and Dryden Williams [8] by providing more accurate energy measurements for consumer devices. By improving the accuracy of our neural network, we are able to remove the ambiguity on the energy used by consumer device use, causing the methodology as described by Adams et. al to have a higher accuracy.

8.4.2 Browser feature

As mentioned in Chapter 7, one can make use of a trained neural network to develop a browser extension. We believe that this should become a standard feature in browsers, for developers. Having this feature will enable developers to create a website keeping sustainability in mind. Providing developers with insights into their website's energy consumption through the browser could incentivize them to explore more sustainable methods for the website's development.

It is important to discuss when the neural network is applicable in the context of a browser feature/extension. Given the extensions main purpose is to provide a rough estimate, we believe that the network is useful when its accuracy reaches at least 50 percent. This way, the estimation should be

sufficient to make the developer aware of their website's performance and make appropriate changes.

In conclusion, we can argue that it is, in fact, feasible to predict the energy consumption of loading a webpage, given improvements to data collection and predicting. A trained neural network can be used in order to create awareness under developers, by creating a browser extension or even introducing it as a browser feature that gives developers a rough estimation of how energy sufficient it is, aiding them in the development process.

Bibliography

[1] A. Petrosyan, "Internet users worldwide 2022," https://www.statista.com/statistics/271411/number-of-internet-users-in-selected-countries/, 2023.

[2] DataForSEO, "Top 1000 websites," https://dataforseo.com/top-1000-websites, 2023.

[3] G. Louw, "Tinn: A tiny neural network library," https://github.com/glouw/tinn, 2020, commit: 815225a8f11c7aff2f3d008cb19980f40dc60de6.

[4] D. Anguita, A. Ghio, S. Ridella, and D. Sterpi, "K-fold cross validation for error rate estimate in support vector machines." in *DMIN*, 2009, pp. 291–297.

[5] M. Kaariainen, "Semi-supervised model selection based on cross-validation," in *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, 2006, pp. 1894–1899.

[6] AttractiveChaos, "Kann: A lightweight c library for artificial neural networks," https://github.com/attractivechaos/kann, 2021, commit: f71236a82af2187820fabd9b1aba3138b8a4de04.

[7] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[8] C. Adams, R. Baouendi, T. Frick, T. Greenwood, and D. Williams, "Calculating digital emissions," https://sustainablewebdesign.org/calculating-digital-emissions/, 2022.

[9] A. S. Andrae, "New perspectives on internet electricity use in 2030," *Engineering and Applied Science Letter*, vol. 3, no. 2, p. 19–31, 2020.

[10] A. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030," *Challenges*, vol. 6, no. 1, p. 117–157, 2015.

[11] J. a. de Macedo, J. a. Aloísio, N. Gonçalves, R. Pereira, and J. a. Saraiva, "Energy wars - chrome vs. firefox: Which browser is more energy efficient?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 159–165. [Online]. Available: https://doi.org/10.1145/3417113.3423000

Appendix A

Sample output from server


```
{
"electricity_consumed_current":3.5045375999999999,
"measurements":549950,
"electricity_consumed_total":1536312.6284292329,
"power_draw":2.7599999999999998
}
```


- `electricity_consumed_current` is the electricity measured right before this is displayed.
- `measurements` is the total amount of measurements done so far
- `electricity_consumed_total` is the total amount of electricity consumed by the server1

Appendix B

Sample Data

B.1 Successful measurement

JSON Format:


```
"https://wikipedia.org":
    {
        "screenshot": succesful,
        "bodySize": "2839",
        "time": "945",
        "adjusted_total": "5.8281984000607565",
        "total": "26.055475200060755",
        "idle1": "2.6918912",
        "idle2": "2.2474752",
        "pageContent":
            {
                "SCRIPT":4,
                "IMG":1,
                "DIV":87,
                "BUTTON":2
            }
    }
```


TXT Format (Neural network input):


```
https://wikipedia.org 945 2839 4 1 87 2 5.8281984000607565
```


B.2 Error when connecting to website


```
"https://officiele-overheidspublicaties.nl": "errrored"
```


Sample data output when connection to website errors.

B.3 Unstable idle

```
"https://jetcamp.com":
    {
        "status": "unstable idle",
        "baseline": "8.634368",
        "last idle": "3.0220288"
    }
```

Sample data output when idle can not be stabilized before measurements. Baseline is energy usage measured before starting rendering, last idle is the last idle measured before exiting.

B.4 Fluctuated idle

```
"https://weerplaza.nl":
    {
        "status": "fluctuated",
        "baseline": "2.2728704",
        "idle1": "3.110912",
        "idle2": "2.0951039999999996"
    }
```

Sample data output when idle energy consumption after measuring differs too much (>1) from the idle energy consumption before measuring.

Appendix C

ODROID-H3+ Specifications

The specifications of the ODROID-H3+ machine used are listed here: [1]

- Intel® Quad-Core Processor Jasper Lake N6005 has a base clock of 2GHz and a boost clock of 3.3GHz with 1.5MB L2 and 4MB L3 cache by a 10 nm process.
- Up to 64GB Dual-channel Memory DDR4 PC4-23400 (2933MT/s)
- Two SO-DIMM slots, up to 32GB per slot
- PCIe 3.0 x 4 lanes for one M.2 NVMe storage
- 2 x 2.5Gbit Ethernet ports
- 2 x SATA 3.0 ports
- SSE4.2 accelerator (SMM, FPU, NX, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AES)
- Intel UHD Graphics 32 EUs Turbo 900MHz
- HDMI 2.0 and DP 1.2 multiple video outputs

[1] Specifications are gathered from hardkernel.com (Link)

Appendix D

Python script for adapting data

```
def filecreate(filename):
    newfile = open("{}-new.txt".format(filename), "a")

    with open("{}.txt".format(filename)) as f:
        for line in f:
            split = line.split(" ")
            to_append = split[1:-1]
            value = float(split[-1])
            if value <= 10.0:
                to_append += [1,0,0,0,0,0]
            elif 10.0 < value <= 20.0:
                to_append += [0,1,0,0,0,0]
            elif 20.0 < value <= 30.0:
                to_append += [0,0,1,0,0,0]
            elif 30.0 < value <= 40.0:
                to_append += [0,0,0,1,0,0]
            elif 40.0 < value <= 50.0:
                to_append += [0,0,0,0,1,0]
            else:
                to_append += [0,0,0,0,0,1]
            for val in to_append:
                newfile.write(str(val) + " ")
            newfile.write("\n")
```