UCLouvain

École polytechnique de Louvain

The inevitable thermodynamic costs of neural networks

Author: **Dimitri SCHOROCHOFF**
Supervisor: **Jean-Charles DELVENNE**
Readers: **John LEE, Léopold VAN BRANDT**
Academic year 2022–2023
Master [120] in Data Sciences Engineering

Abstract

Whatever the quality of the hardware, the laws of thermodynamics will limit the energy efficiency we can achieve when making computation. Today, we are still unaware of the extent of these limits in the field of machine learning. Can we compute smarter to reduce these limits? In this work, we propose a Python library that quantifies these limitations for Artificial Neural Networks(ANN), a specific branch of machine learning. Based on numerical samples, we propose an estimate of a lower bound on the minimum energy cost, namely the Landauer and Mismatch cost, required to run an already trained neural network. Using these results, we suggest a set of neural structure recommendations that will reduce the lower bound for such networks.

Acknowledgements

I would like to express my gratitude to Professor Jean-Charles Delvenne for his availability and patience when answering my questions during this semester.

I want to thank Professor John Lee and Léopold Van Brandt who quickly agreed to become my jury and readers.

I would also like to thank my family for their unfailing support in all circumstances. Special thanks go to Françoise Verdonck, Sophie Schorochoff, Gauthier De Coninck, and Eric Vandueren who took the time to proofread my masters thesis

Contents

Chapter 1

Introduction

At a time when Computer Science is the source of many advances and discoveries, we are striving to improve every aspect of our computers and the way they work. For a long time, the focus has been on obtaining more precise results or obtaining these results more quickly. However, in the past decades, the focus has partly shifted toward efficiency. As ecological concerns arose, lowering the amount of energy we use became at the heart of a plethora of research. Thanks to them, improvement in the hardware and software has been considerable. As years go by, for a given task, the energy consumption of our computers keeps decreasing. One might wonder how far can we push this efficiency. Could we ultimately reach zero energy costs when running a program?

Unfortunately, the answer is no, the laws of thermodynamics prevent us from doing so. In 1961, R. Landauer showed that no matter the physical device, there is an unbreakable energy cost associated with erasing information, this is the Landauer principle [1]. Today, modern computers frequently erase information as they clean their memory before storing new information. We would have to give up on this process to go beyond the Landauer principle. Basically, we would need to drastically change our way of computing.

This principle has been discussed by many scientists, including David H. Wolpert and A. Kolchinsky. Together they discovered an additional thermodynamic cost required to run an electric circuit. The so-called Mismatch cost depends on how far the actual distribution of variables is from an optimal distribution that would minimize the costs[2]. However, this work was only applied to electric circuits and one might wonder how would these thermodynamic costs apply to computers in general. Machine learning is one of the fields that could use such results as it is known for being computationally intensive. Recently, N. Beuseling studied the effect of these energy limitations on one of the most popular machine learning algorithms: the Artificial Neural Network (ANN)[3]. Starting from the circuit theory of Wolpert and Kolchinsky, he made the transition and obtained analytical results on trained ANN. Using numerical simulations applied to small ANN, he discovered that higher correlated variables usually lead to higher Mismatch cost.

In this thesis, we take up Beuseling's work and establish a framework for numerically estimating the minimum energy required to run a trained ANN of any size. We propose an analysis and comparison of the unavoidable costs of different types of layers. On this basis, we suggest a set of measures to decrease the lower bound on energy costs. During our research, we noticed that the normal distribution had consistent costs and we defined an heuristic to quickly compute an estimation of those. Finally, we provide a Python library to evaluate both the Landauer and

Mismatch costs of a given neural network implemented using the Keras library. The aim is to give additional information and help a better understanding of the behavior of ANN, leading to more energy-efficient neural structures.

This paper is divided into five chapters. Chapter 2 summarizes the state of the art. All the concepts used to develop this work are detailed here. Next comes a description of the framework we propose to compute estimations of both the Landauer and Mismatch costs. Then, the brief chapter on software development goes over some implementation details and the performance of the library before we reach our conclusion.

Chapter 2

Theoretical background

The two main concepts of this work are the Landauer and Mismatch costs. Since they are respectively defined in terms of entropy and relative entropy we first introduce these terms before detailing both concepts. Then, we quickly go over the electronvolt, the unit of measurement we will use to quantify both costs. After that, we summarise the basics of artificial neural networks before explaining how to compute both their Landauer and Mismatch costs. The last section is about the entropy estimators that we used to perform these computations from numerical samples.

2.1 Entropy

A thermodynamic system can be defined via its macroscopic parameters such as the number of moles n, and the temperature T. However, the same system can also be defined in terms of its microscopic states: the position and momenta of every particle composing the system. Each microstate adds a small contribution and together they define the properties of the system.

In 1877 Ludwig Boltzmann characterized a way to measure the number of these possible microstates at thermodynamic equilibrium[4] via the Boltzmann entropy

$$S_{Boltzmann} = k_B \ln \Omega \tag{2.1}$$

where $k_B = 1,380649.10^{-23}[J/K]$ is the Boltzmann constant and Ω is the number of microstates compatibles with macrostates.

Later, in 1902, J. Willard Gibbs proposed a definition covering more cases.[5].

$$S_{Gibbs} = -k_B \sum_i p_i \ln p_i \tag{2.2}$$

This time using the probability that the microstate i occurs: p_i. If the probabilities are uniform, $p_i = \frac{1}{\Omega}$, we come back to the Boltzmann definition.

We can link Gibbs's definition to information theory via the Shannon entropy introduced by Claude Shannon in 1948[6].

$$S(X) = -\sum_{x \in \chi} p(x) \ln p(x) \tag{2.3}$$

If we set the random variable's outcome as "being in microstate x", we immediately see the relationship between the two formulas, one is multiplied by k_B, and the other isn't. This gives us yet another insight into what entropy is. In Shannon's definition, it represents the average amount of information contained by a random variable. For those who aren't familiar, we will illustrate its meaning by considering three coins.

1. A fully biased coin that always lands on the head ($p_{\text{head}} = 1$). Therefore actually flipping the coin doesn't give us any information since we know in advance that it will land on the head. It comes as no surprise that $S = 0$
2. A biased coin ($p_{\text{head}} = 0.99$). Flipping it will give us very little information since we are almost sure of the result: $S = 0.056$
3. A fair coin ($p_{\text{head}} = 0.5$). Flipping this type of coin will give us maximum information since we don't have any insight: $S = \ln 2$

It is useful to note that if we change the logarithm base from e to 2 we obtain the amount of information in bits. However, in this case, the definition doesn't match Gibbs's entropy anymore.

The relative entropy, also called Kullback-Leibler divergence is an asymmetric measure of the difference of information between two distributions p and q.

$$D(p\|q) = \underbrace{\sum_{x\in\mathcal{X}} p(x)\ln p(x)}_{S(X)} - \sum_{x\in\mathcal{X}} p(x)\ln q(x) \tag{2.4}$$

2.2 Landauer principle

"*Without considering the question of access, however, we can show, or at least very strongly suggest, that information processing is inevitably accompanied by a certain minimum amount of heat generation*" [1]. In *Irreversibility and Heat Generation in the Computing Process (1961)*, Landauer makes an experiment of thought. He considers the effect of erasing bits inside a closed system such as a computer, see figure 2.1.

1. Initial state: N bits of unknown value. Due to combinatorics, they cover at least 2^N **possible** microstates, therefore they yield at least a Boltzmann entropy of $k_B N \ln 2$
2. Operation: RESET TO ZERO the N bits
3. Final state: N bits of known value. The number of possible states these bits have is 1 yielding an entropy of 0.

In this closed system, performing such an operation would drop the total entropy by $k_B N \ln 2 - 0$ which is impossible because of the second law of thermodynamics. Therefore **at least** this amount of entropy must appear elsewhere! Most likely as a heating effect. If we consider an isothermal process, we can link entropy to the generated heat E.

$$\Delta S = \frac{E}{T} \tag{2.5}$$

Hence the operation RESET TO ZERO dissipates at least $E = k_B T N \ln 2$ heat.

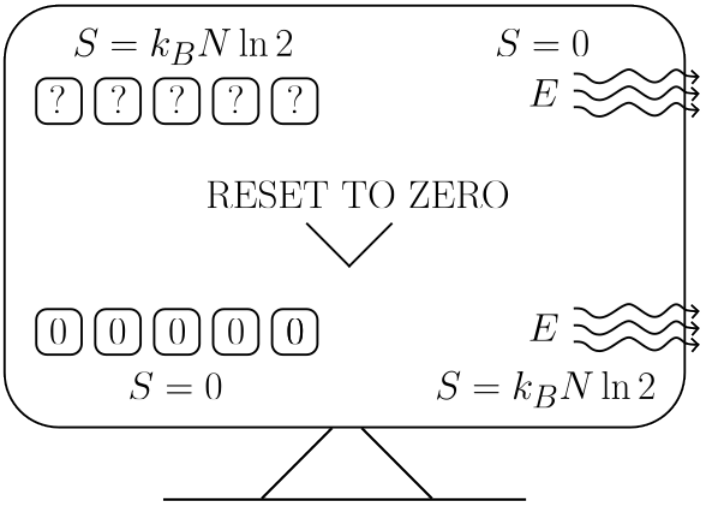


Figure 2.1: Landauer's experiment of thought

From this example, we can deduce the Landauer bound.

$$E \geq k_B T \ln 2 \tag{2.6}$$

Here E is the amount of energy needed to erase one bit of information. At normal temperature (i.d 293.15K) it is at least 2.9 zeptojoules. To give an order of comparison, as of today, the most efficient computers use around 87 million times more energy.[7] [1].

A more general formula is obtained by considering the delta of entropy between the initial and final state of the system. This is a crucial result for this thesis. Here is the version using Shannon entropy.

$$\boxed{\mathcal{L}(p) = S(p_i) - S(p_f)} \tag{2.7}$$

$$\boxed{E \geq k_B T(\mathcal{L}(p))} \tag{2.8}$$

.

In 1982, Bennett used this bound to explain why Maxwell's demon couldn't violate the second law of thermodynamics[8][9][10]. In short, the demon tries to extract free work out of gas by compressing it effortlessly using its knowledge of particles. However Bennett noted that to return to the initial state, the demon would have to free its memory which cost energy as we have just seen. I would recommend Alberto Verga's more in-depth explanation [11].

Landauer's bound seems to be confirmed by scientific experiments notably in 2012 [12] and 2014 [13]. Researchers approached the limit using nanomagnet in 2011[14]. In 2016, other researchers succeeded in erasing one bit using only 4.2 zeptojoules, only 50% more than the bound[15].

[1] To compute this approximation we considered the 1st computer of the Green500 list. Since it uses 65.396 GFlops/Watt and considering that a float operation can erase up to 32 bits, we get around 2.4×10^{-13} bits/J

Finally, it should be noted that this limit only applies to logically irreversible processes. Indeed, we have assumed that entropy is generated in the form of heat to compensate for the fall in information's entropy, see fig.2.1. However, we could imagine a reversible computer that never generates heat because it never loses information. For example, a computer could use Toffoli gates instead of regular ones, although storing all this extra information could become a challenge.

2.3 Mismatch cost

2.3.a Results

It is easy to imagine that, when a circuit is executed, some input distributions generate more entropy than others. Certain circuit configurations or electronic components may be more favourable to some distributions than others. In 2020, David H. Wolpert and Artemy Kolchinsky proved the existence and computed this extra cost; they called it Mismatch cost.[2].

$$\boxed{\mathcal{M}(p) = D(p_i \| q_i) - D(p_f \| q_f)} \tag{2.9}$$

One of the possible prior distributions q is optimal in the sense that it minimizes the overall entropy production. Also, the mismatch cost is always non-negative because of the data processing inequality [16].

From now on, we will assume a hypothetical technology: **future engineers will be able to choose the optimal distribution q (e.g. they make it correspond to a standard normal).** This hypothesis isn't unreasonable as we know that distribution q depends on the electronic components. Engineers could modify these components little by little until they reach the distribution of their choice.

Adding this with Landauer's cost, we can now consider a better lower bound on the energy cost of circuits.

$$\boxed{E \geq k_B T(\mathcal{L}(p) + \mathcal{M}(p))} \tag{2.10}$$

For more information and proof of these results, a summary can be found in Appendix B.

2.3.b Example

An example from the original paper might ease the understanding[2]. We will analyse the thermodynamic cost of an XOR gate. Consider an input distribution that is uniform over the set $\mathcal{X} = \{00, 01, 10\}$ such that it never gives the state 11 as input.

$$p_i(x) = \begin{cases} 0 & \text{if } x = 11 \\ \frac{1}{3} & \text{otherwise} \end{cases}$$

In this case, the optimal distribution that minimizes the EP is uniform over all possible states 00, 01, 10, and 11.

$$q_i(x) = \frac{1}{4}$$

The set of final state is $\mathcal{Y} = \{0, 1\}$. Based on our knowledge, we can compute output distributions. For example, given the input distribution, state 01 and 11 occurs two times out of three. Since XOR(0,1)=XOR(1,1)=1, the XOR gate will output 1 two times out of three.

$$p_f(y) = \begin{cases} \frac{2}{3} & \text{if } y = 1 \\ \frac{1}{3} & \text{otherwise} \end{cases}$$

$$q_f(y) = \frac{1}{2}$$

We can now compute the Landauer and Mismatch cost.

$$\mathcal{L}(p) = S(p_i) - S(p_f) = -\ln\frac{1}{3} + \left(\frac{1}{3}\ln\frac{1}{3} + \frac{2}{3}\ln\frac{2}{3}\right) \approx 0.46$$

$$\begin{aligned} \mathcal{M}(p) &= D(p_i \| q_i) - D(p_f \| q_f) \\ &= \left(-S(p_i) - \sum_{x\in\mathcal{X}} p_i(x)\ln q_i(x)\right) \\ &- \left(-S(p_f) - \sum_{y\in\mathcal{Y}} p_f(y)\ln q_f(y)\right) \\ &= (\ln 4 - \ln 2) - \mathcal{L}(p) \\ &\approx 0.23 \end{aligned}$$

Using equation 2.10, we obtain a lower bound on the energy consumption of running an XOR gate at standard room temperature (20°C): 2.8 zeptojoules.

2.4 Electronvolt

As we have seen in the previous sections, the amounts of energy represented by the Landauer and Mismatch costs are very small. It can be hard to conceive how much energy lies within a zeptojoules. This is why for the rest of the works we will use a more adapted unit of measurement, the electronvolt (eV).

An electron-volt is the amount of kinetic energy obtained by an electron when it accelerates from rest through an electric potential difference of one volt in a vacuum. Therefore an eV equals $1.602176634e^{-19}$ joules. Whenever we are going to convert a lower bound to its energy cost, we first multiply it by k_bT with T at standard room temperature 293.15 K then we change the cost unit from joules to eV. For example, the Landauer bound converted this way equals 0.018 eV.

2.5 Artificial neural network(ANN)

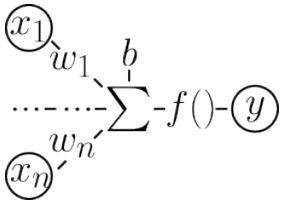
2.5.a Basics

Machine learning is a field that uses algorithms to learn a specific subject and then makes predictions. For example, the algorithm learns an image set of handwritten numbers. Then

when we provide a new image, it guesses the number on it.

Artificial neural networks are one family of these algorithms. It consists of a set of processing units called neurons organized in multiple layers forming a network. Each *synapse* linking two neurons has an associated weight, the higher its absolute value, the stronger the link between those neurons. The learning process consists of adjusting those weights based on a trial and error method called *backpropagation*.

The first ANN was the perceptron, which was used to perform binary classification using supervised learning. The concept is very simple, it computes a weighted sum of the input features and then feeds the result to an activation function $f()$, see figure 2.2. The output y is the prediction.



$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Figure 2.2: Neuron activation scheme with step activation function

We can then chain this basic block into multiple layers, the output becoming the input of the next layer. Such a network was called the multilayer perceptron. There are several ways to build more complex ANNs.

- By changing the neural structure: tweaking the number of layers, neurons, and connections
- By using different activation functions for some neurons (e.g ReLU, Softmax, Sigmoid)
- By modifying the sum operation by another one such as the convolution
- By applying other optimization algorithms, Adam to cites one

The size and complexity of the network are quite variable. Starting with a small multilayer perceptron used to predict a BMI. Ending with a complex network having trillions of parameters (i.d weights) named ChatGPT.

2.5.b Thermodynamic costs applied to ANN

Single neuron
Recently, Niels Beuseling applied the theory of Landauer and Mismatch cost to ANN[3]: replacing a gate with a neuron. To compute the Mismatch cost we need to obtain the optimal prior distribution. As it is hard to compute, he proposed using two common priors.

1. The normal distribution mostly because of the implications of the central limit theorem
2. The uniform distribution because it is the least informative if we don't set any constraints.

Beuseling analyzed the evolution of thermodynamic costs of a single neuron through Monte Carlo simulation. He showed that the mismatch cost tends to rise as inputs become more and more correlated. He also computed analytical results when using the normal optimal prior.

We will review these results applied to one example. Consider a single neuron, see figure 2.2. We assume that the optimal prior distribution is a multivariate standard normal: $X_{\text{opti}} \sim \mathcal{N}(\mathbf{0}, I)$. The output is the sum of theses distributions: $Y_{\text{opti}} \sim \mathcal{N}(0, |\mathbf{w}|^2)$. If the input follow a normal distribution as well, $X \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ and $Y \sim \mathcal{N}(\mathbf{w}\boldsymbol{\mu}, \mathbf{w}\Sigma\mathbf{w}^T)$, we can compute the thermodynamic costs. For this we will use the analytical entropy of the multivariate normal distribution, this result comes from p.256 of Joy A. Thomas's book [17].

$$S(\mathcal{N}(\boldsymbol{\mu}, \Sigma)) = \frac{1}{2} \ln \left((2\pi e)^n \det \Sigma \right) \tag{2.11}$$

Now we apply this result to our problem.

$$\mathcal{L}(p) = \frac{1}{2} \ln \left((2\pi e)^n \det \Sigma \right) - \frac{1}{2} \ln \left((2\pi e) \mathbf{w}\Sigma\mathbf{w}^T \right) = \frac{n-1}{2} \ln(2\pi e) + \frac{1}{2} \ln \frac{\det \Sigma}{\mathbf{w}\Sigma\mathbf{w}^T} \tag{2.12}$$

Similarly, we compute the Mismatch cost

$$D(x \| x_{\text{opti}}) = \frac{1}{2} \Big(\text{tr}\Sigma + \boldsymbol{\mu}\boldsymbol{\mu}^T - n - \det \Sigma \Big)$$

$$D(y \| y_{\text{opti}}) = \frac{1}{2} \Big(\frac{\mathbf{w}\Sigma\mathbf{w}^T}{|\mathbf{w}|^2} + (\mathbf{w}\boldsymbol{\mu})^2 - 1 - \ln \frac{\mathbf{w}\Sigma\mathbf{w}^T}{|\mathbf{w}|^2} \Big)$$

$$\mathcal{M}(p) = \frac{1}{2} \Big(\text{tr}\Sigma - \ln \det \Sigma - (n-1) - \frac{\mathbf{w}\Sigma\mathbf{w}^T}{|\mathbf{w}|^2} - (\mathbf{w}\boldsymbol{\mu})^2 + \ln \frac{\mathbf{w}\Sigma\mathbf{w}^T}{|\mathbf{w}|^2} \Big) \tag{2.13}$$

To give an order of magnitude, at normal temperature (i.d 293.15K), if we set plausible values such as $n = 3$, $\mathbf{w} = (0.1, -0.2, 0.7)$, $\boldsymbol{\mu} = (0, 1, -0.5)$ and $\text{diag}(\Sigma) = (1, 2, 0.1)$; we obtain a Landauer cost of 0.076 eV (12.2 zeptojoules) and a Mismatch cost of 0.017 eV (2.8 zeptojoules).

Multiple neurons

The extension to multiple neurons is pretty straightforward, as we can sum the individual cost of each neuron. However, as he pointed out, we need to consider the evolution of the optimal prior. Either we consider an actualized version: we state an initial prior then for every layer we compute the new prior based on the ones in the previous layer. Or we consider the non-actualized version: we keep the same optimal prior for every neuron. This second option is not only easier to compute but also probably closer to reality: if every component is made in the same factory then they probably yield the same optimal prior.

As for the single gate case, he performed numerical simulation but this time on basic multi-neuron structures where he obtained similar results.

2.6 Entropy estimators

In this section, we explain how to obtain a numerical estimate of the Shannon entropy from data samples. Indeed, computing the analytical entropy is not a realistic option as we would have to restart the calculation from scratch for each new neural network and this can sometimes be very difficult due to the non-linearities of the activation functions.

2.6.a From continuous to discrete

The samples we will be considering are the input/output of our neurons hence we can consider every sample as a real number without loss of generality. In modern computers, a real number is represented by a finite number of bits usually 32 or 64. This means 2^{32} or 2^{64} possible states for a real number and we would need to estimate as many probabilities to apply the Shannon entropy formula.

To reduce the amount of computation we apply a trick. First, we consider continuous entropy.

$$S_{\text{continuous}}(p(X)) = -\int_{\mathcal{X}} p(x) \ln p(x) dx$$

Then we transition to the discrete entropy. By doing so we can choose the discretization step size we want and thus n, the number of states.

$$S_{\text{discrete}}(p(X)) = -(\sum_{k=1}^{n} \hat{p_k} \ln \hat{p_k}) + \ln \Delta$$

Here the support is divided into n bins of size Δ. X's outcome falls within the bounds of bin i with probability p_k. Its empiric estimation is $\hat{p_k} := n_k/n$ where n_k is the number of samples within the bounds of bin i.
The last term $\ln(\Delta)$ might be surprising at first but it is an important step of the discretization[18]. Without it, the entropy estimation keeps increasing as we increase the bin count.

2.6.b Covered Adjusted Estimator (CAE)

In 2003, Shen and Chao proposed an entropy estimator that would take into account unseen species in the samples[19]. This estimator was further assessed and tested in 2007[20]. Notably, they compare CAE to other entropy estimators such as MLE, JK, and BUB.

One of the biggest challenges we will face is that the sample space grows too quickly to be fully covered by the samples we generate. Hence the Covered Adjusted Estimator is great to use because it was made to palliate this issue.

$$S_{\text{CAE}}(p(X)) := -\sum_{k=1}^{n} \frac{\tilde{p_k} \ln \tilde{p_k}}{1 - (1 - \tilde{p_k})^n}$$

With $\tilde{p_k} = \hat{C}\hat{p_k}$, $\hat{C} = 1 - \frac{f_1}{n}$ and $f_1 = \sum_{k=1}^{n} 1\{n_k = 1\} = \#$sample with exactly one appearance.

As a reminder, here is a brief history of how this formula was obtained. To correct the bias we divide by the inclusion probability. This result comes from Horvitz-Thompson[21]. However Ashbridge and Goudie[22] found out that the result wasn't working well with sample probabilities and they proposed to use coverage-adjusted probabilities, $\tilde{p}$, instead. Chao and Shen[19] put everything together by using the Good-Turing coverage estimator $\hat{C}$.

2.6.c Estimator used

We need to combine the two results as the CAE estimator was made for discrete variables. From now on, we will use the following formula to estimate entropy.

$$\hat{S}(p(X)) := -\sum_{k=1}^{n} \frac{\tilde{p_k} \ln \tilde{p_k}}{1-(1-\tilde{p_k})^n} + \ln \Delta \tag{2.14}$$

2.6.d Curse of dimensionality

The informed reader will be familiar with the curse of dimensionality and, more particularly, the sampling problem it causes. When the dimensionality of a space is increased, the size of the volume grows exponentially fast and the amount of data needed to fill such space becomes enormous. This problem arises in particular when estimating the entropy of multidimensional distributions.

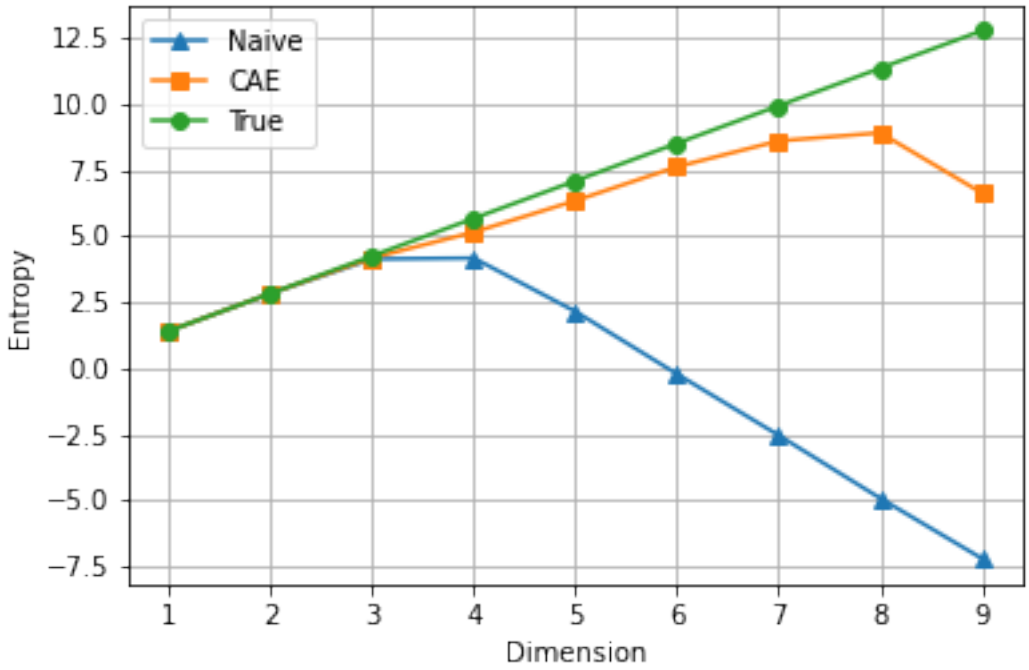


Figure 2.3: Entropy estimation of distribution $\mathcal{N}(\mathbf{0}, I)$ over 1 million samples

As we can see on figure 2.3, the estimations quickly fall off as the number of dimensions increase. Using advanced methods such as the CAE only delays the inevitable. However, it is still significantly better and it allows us to work with relatively high dimensions.

2.7 Kullback-Leibler divergence estimation

In order to estimate the KL divergence, we will use the formula defined in equation 2.4. Again, we will estimate empiric probabilities using bins. However this estimation seems to be way less robust, see figure 2.4. As soon as the two distributions become too different, the estimation value drops drastically. This is most likely due to a lack of samples. As the overlap between distribution decrease, we need better empiric probability estimation of the part that occurs less often (e.g. the tails of a normal distribution). Since they occur less often, we need more samples. The matters become even worse when considering multi-dimensional distribution.

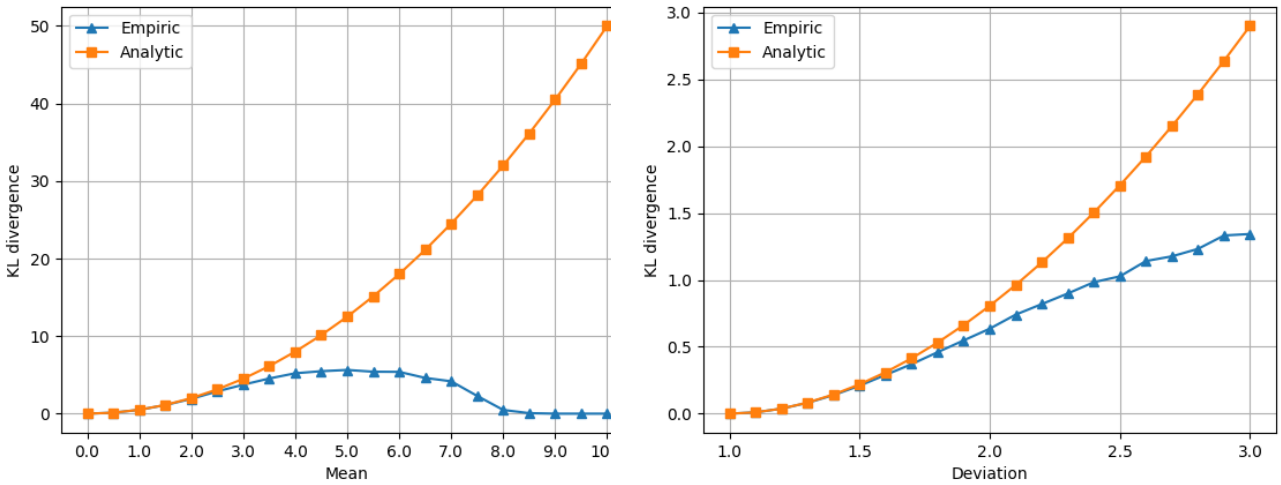


Figure 2.4: Empiric vs analytic KL divergence of two standard normal distributions over one million samples. On the left, we change the mean of the first distribution. On the right, we change its deviation

Therefore we need to be extra careful when considering the mismatch cost because it might be flawed by poor estimations of KL divergence.

Chapter 3

Framework of computation and results

In this chapter, we define how to apply the theory we have just seen in practice see the see the results. First, we determine the scope of what our computers can and can't do as this will affect the possible lower bounds we can reach. In the following section, we individually analyze the neural configuration of different layer types and see how to compute their costs accordingly. Then we put everything together in the third section. There, we see how we can chain operations to obtain the Landauer and Mismatch costs from samples. The last section is dedicated to the description of heuristics we can use to estimate these costs much faster when the input follows a normal distribution.

3.1 Precision levels

We have seen before that the Landauer and Mismatch cost is computed between an input and an output. We still need to define what they are in the case of neural networks. Which degree of precision should we consider? Should we consider the system as a whole or should we compute costs individually for each neuron? We identified four levels of precision going from the most to the least general concept, see figure 3.1. Each level requires fewer and fewer assumptions about the way computers would store information and make computations. However, the higher the precision level, the worse the lower bound on the costs.

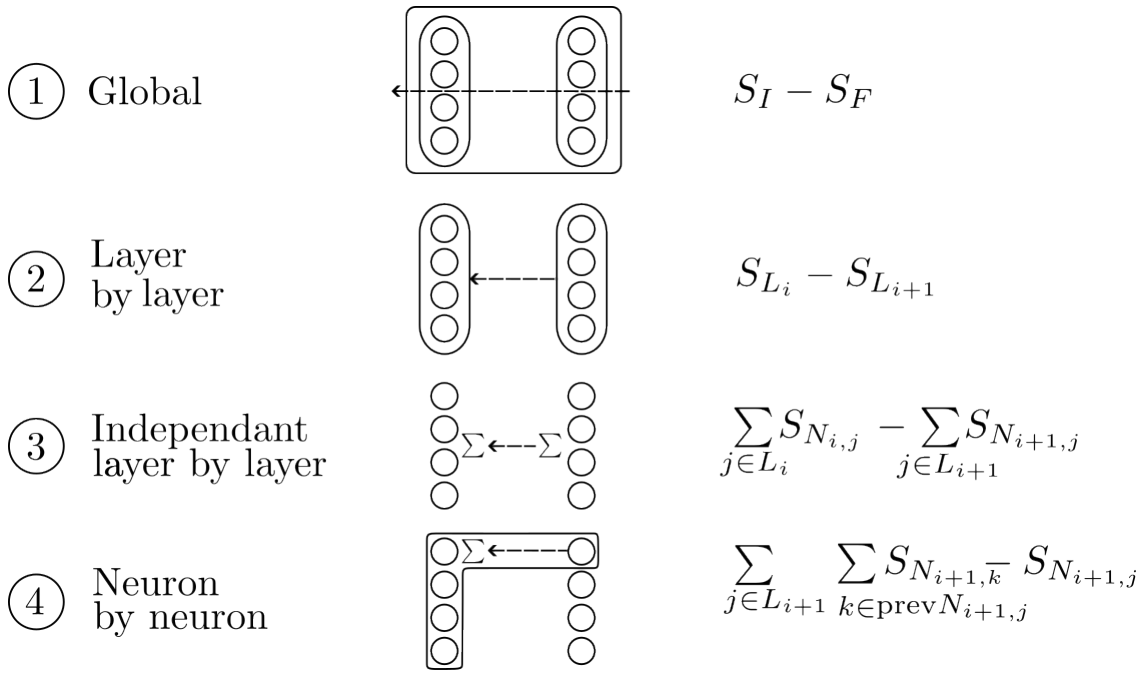


Figure 3.1: Different levels of precision for entropy comparison in one slice of a neural network. Format: Level name | Scheme | Landauer cost

The first level most certainly doesn't represent reality. In it, computers would never store any information but the ones from the input and output of the entire neural network therefore generating no extra cost inside. The second and third levels are way more interesting, in those, computers would consider each layer as a whole when storing information and making computations. In level 2, they would also take the correlation between neurons of the same layer into account to avoid generating any extra entropy. In level 3, this isn't the case. The last level is the closest to reality as of now. Computers' hardware considers neurons one by one. When computing the output of a neuron j they use individually the value of every neuron connected to it. In this scenario, there are a lot of redundancies as one input neuron is often connected to multiple output neurons. This raises the Landauer cost quite a bit because this neuron's entropy will appear multiple times when computing the drop in entropy.

We have covered the impact of these levels on the Landauer cost and we can use similar input/output considerations for the Mismatch cost if we replace entropy by KL divergence. However in this case we need to overcome an additional difficulty. As we have just seen, due to current hardware limitations, neurons' values are computed individually and we can sum up their individual entropy to compute the Landauer cost. We shouldn't do the same for the Mismatch cost; since it is comparing distributions, we need to consider the **joint** distribution of the input neurons. However, sampling the KL divergence of multidimensional distribution becomes extremely difficult as the number of dimensions grows (i.e. the number of initial neurons). Although this is the *correct* way to compute the Mismatch cost, it quickly becomes imprecise even for a small number of initial neurons. Nevertheless, we implemented this way of computing the Mismatch cost because it can still be useful for extremely small neural networks. This method can also become relevant when using large amounts of samples if the neural network consists of only convolution and pooling layer; we will see in section 3.2 that they keep a small number of initial neurons, especially when using a 2x2 kernel.

For larger networks, we need to find an approximation to reality. If we assume that the distri-

butions are independent, we can add up the KL divergences of the initial neurons one by one to calculate their joint KL divergence. If we consider that the distributions are independent, we can sum up the KL divergence of initial neurons one by one to compute their joint KL divergence. Ultimately we compute the Mismatch cost in a similar fashion as we compute the Landauer cost. Again, this is an approximation, and neurons' distribution will have dependencies but in most cases, this is better than trying to estimate the KL divergence of a high dimension distribution.

In the following sections, we will cover the *correct* way of computing the Mismatch cost since computing the approximation is straightforward. Once we know the method to compute the Landauer cost, we only need to replace the entropy with the KL divergence. Also, we will make all the next computations based on the last level of precision since we judge it as the closest to reality.

3.2 Costs of different layer types

Different layer types imply different links between neurons. We will see how they affect the Landauer and Mismatch cost in the reshape, dense, pooling, and convolution layers.

For starters, we will call every neuron that is part of the previous layer: a *previous neuron* and we will define as a *current neuron* every neuron in the layer we're currently analyzing. In order to use the formulas 2.7 2.9 to compute the costs, we need to define the set of initial neurons and the set of final neurons. We will consider one by one each current neuron as the set of one final neuron. Then, according to the layer type, we find in the previous layer the set of initial neurons linked to it.

Now to compare the costs between layers we need basic settings that we will apply to all experiments in this chapter (i.e. from figure 3.3 to figure 3.11). In this setup, the input follows a **standard** Normal distribution, the optimal input follows a slightly different Normal distribution: $\mathcal{N}(0.5, 1.25)$, and the weights, when needed, follow a glorot uniform distribution. We can reasonably expect to find Normal distributions, in particular because of the properties of the central limit theorem. We arbitrarily choose the mean and standard deviation of these Normal distributions. However, we must choose our bin size accordingly. Here we set it 0.01 as it gave good results without requiring a too high sample count. Finally, to avoid adding extra complexity, the default weight distribution proposed by Keras was used. With that said, we can now start the layer comparison.

The reshape layer is fairly simple, it only changes the shape of the neurons (i.e. their relative positions). It is often used to make the transition between a 1D layer such as the dense layer and a multidimensional layer such as the pooling and convolution layers. Since the information stored by these neurons remains the same, we can consider that this layer yields no Landauer cost nor Mismatch cost.

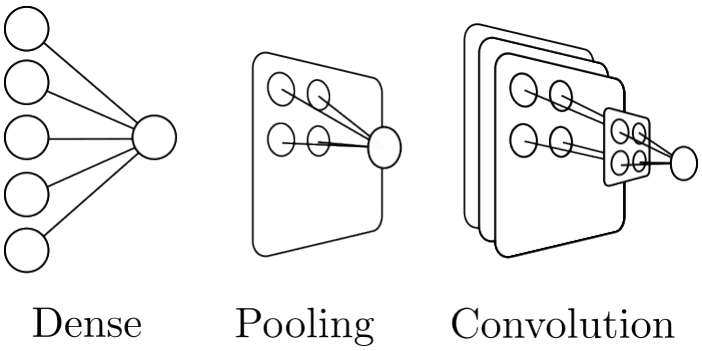


Figure 3.2: Linkage scheme of different layer types

In the dense layer, every current neuron is connected to every previous neuron. Thus the initial set of neurons is always the whole group of previous neurons. As seen in the previous section, we will sum up the costs of every combination. In this case, the larger the number of initial neurons, the larger every individual drop in entropy or KL divergence. While increasing the number of final neurons will increase the number of combinations to sum up.

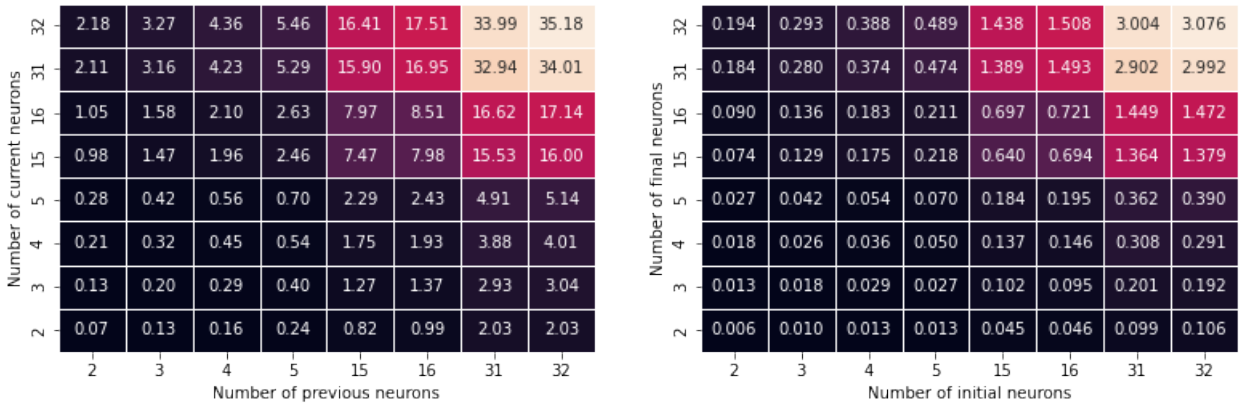


Figure 3.3: Heatmap of the Laundauer (left) and Mismatch (right) costs [eV] of a dense layer

In practice, we can see on figure 3.3 that the Landauer and Mismatch costs are the highest on the diagonal (i.d when the number of initial and final neurons are even). In contrast, the costs get the smallest when we have only one type of neurons either only previous neurons or only current neurons. If we consider the axial symmetry over the diagonal, the lower right part has lower costs, this is especially the case for the Mismatch cost. This means that to minimize the costs for a set number of neurons, we should rather have as many previous neurons as possible. As a consequence, **neural network structure that starts with a high number of neurons and progressively reduces it is favored**. For example, the encoder part of an autoencoder will generally have lower Landauer costs than the decoder part.

In the pooling layer, the number of initial neurons is always the same and is determined by the pool size. In addition, the final neurons often have few or no initial neurons in common. While we would expect a higher pool size to yield higher Landauer and Mismatch costs as higher initial neuron count often results in a higher drop in entropy and KL divergence, this doesn't seem to be the case in practice. Another surprising fact is that the costs of average and max pooling are almost identical. Finally, the reader might be confused by the stair-like shape of

the graphs on figure 3.4, this is because of a lack of padding. The extra initials neurons are not used until a full pooling block is available.

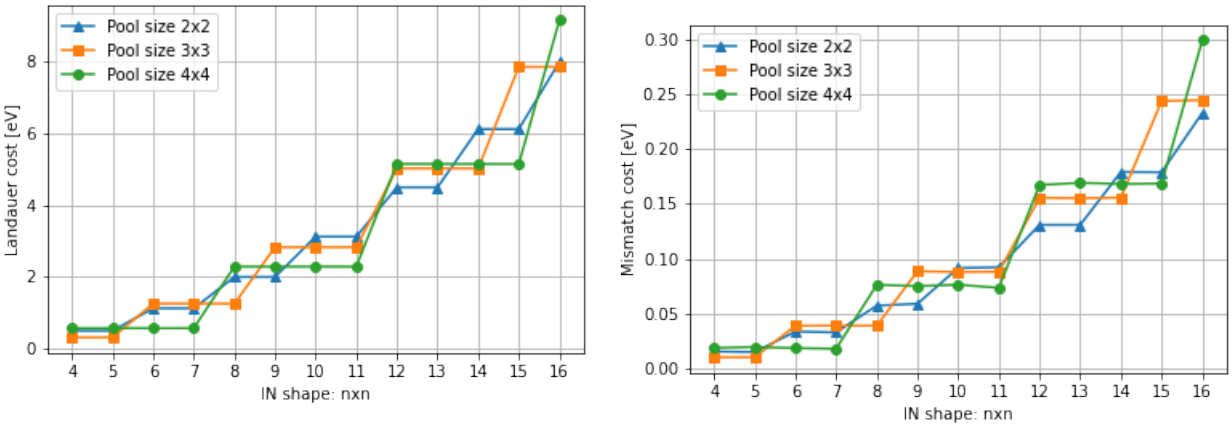


Figure 3.4: Laundauer (left) and Mismatch (right) costs of average pooling layer

The convolution layer is quite similar to the pooling layer as the number of initials neurons are determined by a kernel size instead of a pool size. In addition, we need to take into account the number of filters. For every filter a corresponding kernel thus an additional batch of initial neurons. It follows this relation #initial_neurons = #filters $\times$ kernel_width^2. As expected, when we increase the kernel size or the number of filters, the Landauer and Mismatch costs increase as the drop in entropy and KL divergence tends to be higher. When looking at figure 3.6, it seems that increasing the kernel size is more penalizing than increasing the number of filters. Cost-wise, it is better to chain two convolution layers having a 2x2 kernel than using a single 4x4 kernel as they have the same receptive field. Therefore if we want to decrease the Landauer cost **it is better to chain convolution layer than increase the kernel size**.

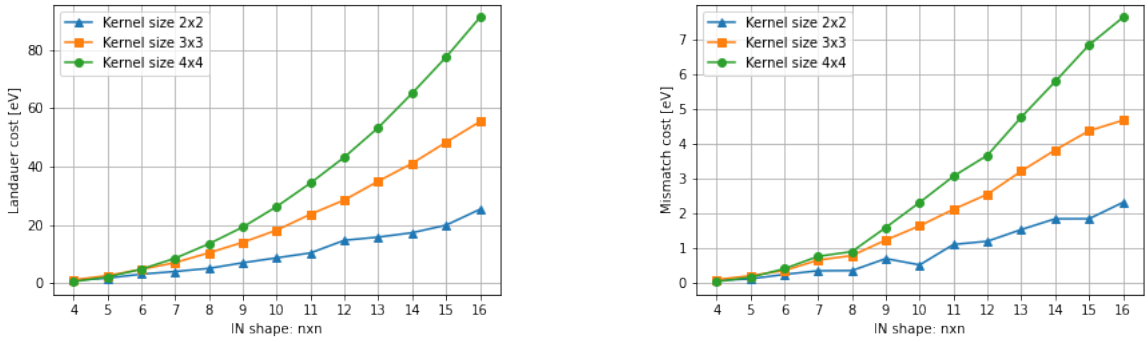


Figure 3.5: Laundauer (left) and Mismatch (right) costs of convolution layer for different kernel size

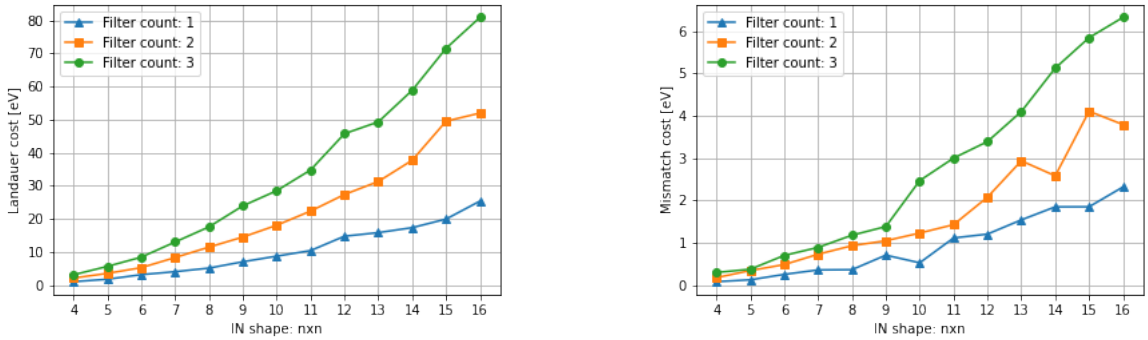


Figure 3.6: Laundauer (left) and Mismatch (right) costs of convolution layer for different number of filter and 2x2 kernel

Finally, we can compare the costs of different layer types for a fixed number of neurons. For example, at 64 neurons, the highest cost is obtained by the dense layer on its worth configuration (equal number of initial and final neurons) with values around 35 eV. Following is the convolution layer with values from 5 to 20 eV. The most cost-effective is the pooling layer with Landauer costs around 2 eV. In these experiments with saw that the Mismatch cost followed the same trends as the Landauer cost. It seems that these trends are mostly dictated by the number of neurons considered. For example, in setup, the Mismatch cost stayed around 10 times smaller than the Landauer cost and it was even smaller in the pooling layer. However, it is important to keep in mind that these values depend greatly on the difference between the optimal and actual distribution of the inputs. When the difference is significant enough the Mismatch cost can even surpass the Landauer cost [3].

3.3 Computation pipeline

This is the core of this work. Now that we have all our building blocks, we will put everything together. The goal is to obtain the Landauer and Mismatch cost from a set of three inputs: the neural network itself, the sampled input distribution, and the sampled optimal input distribution. A summary diagram of the pipeline can be found on figure 3.7.

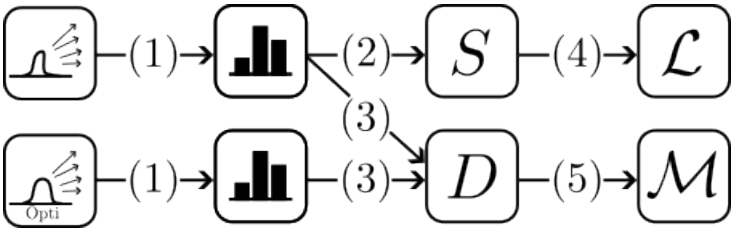


Figure 3.7: Pipeline diagram

(1) From samples to histograms
The first step is to run every sample through the neural network and store the outputs of each neuron. This raises the question: How are we going to efficiently store these results? Since we are planning to only use these values to compute entropy, we can store them as histograms. Indeed, as we have seen in subsection 2.6.a, once we have chosen a discretization step, we only need to know how many samples fell within each bin to compute entropy.

To compute a one-dimensional histogram we divide the real axis into bins of fixed size l. For each sample x we do the following.

1. Compute the bin number: $f(x) = \text{round}(\frac{x}{l})$
2. Increment the count in the histogram : $H(f(x)) += 1$

The histograms establish the relation between the bin number and the amount of sample that belongs to that bin. From there, we easily compute the empiric probabilities of falling within each bin by dividing the whole histogram by the total number of samples used.

The transition to the multidimensional case is made by considering each axis individually. Hence the bin numbering function becomes.

$$\mathbf{f}(\mathbf{x}) = \big(f(x_1), f(x_2), ..., f(x_n)\big)$$

For each set of inputs, we will build two sets of histograms.

- **Independent histograms** are 1 dimensional. They each correspond to the output of one neuron.
- **Dependent histograms** are N dimensional. For every neuron, we consider the set of outputs of the N linked neurons in the previous layers (e.g. the 4 neurons that we averaged in a 2x2 average pooling layer). The way we determine initial neurons is briefly described in section 3.2

The computation of Landauer costs only requires the independent set as we need to consider the entropy generated by each neuron individually. However, we need both sets for the Mismatch cost. Here we need to consider distributions, not entropies, thus we need dependent histograms to estimate the multidimensional distribution of the set of initial neurons. We also need independent histograms when considering the distribution of final neurons individually.

It is important to note that most python implementation of histograms such as the one from Numpy stores the histograms as a multidimensional array. In this implementation, each cell corresponds to one bin. While this is the more efficient way when it comes to access, the memory it needs grows exponentially with the number of dimensions. Most of the time, the vast majority of the cells will remain empty.
Given this problem, we can think of our histogram as a sparse matrix. One way of storing such a matrix is to use a hashmap. Here, the key is the bin number and the value is the associated number of samples. In this way, we allocate zero memory to slots that have not received any samples.

That said, we still need to determine the size of the discretisation stage. Here, we have considered two choices:

1. Choose arbitrarily the step size beforehand
2. Choose arbitrarily the number of intervals, then compute the step size based on the minimum and maximum sample value

The first choice will use less memory as we can iteratively add data to the histogram. However, it could become imprecise when we have a poor guess of the step size. If the size is too large, every sample will fall in the same bin and if the size is too small, every sample will fall in a different bin. In either of these extreme cases, we lose all the information. On the other hand, the second choice gives is easier time making correct guess, making it more generally more robust. However, it needs more space as we need to store the results of every sample to compute their minimum and maximum before making the histogram. It also has a major flaw, the bins between different histograms aren't synchronized making the computation of the Kullback-Leibler divergence extremely difficult. For this very reason, we will use step size instead of step count from now on.

(2) From histograms to entropy
This step is quite straightforward. For every neuron, we use our entropy estimation equation 2.14 to transform the neuron's histogram into entropy.

(3) From histograms to KL divergence
For each neuron, we compute the KL divergence by applying the formula 2.4 where we use the Normal distribution's histogram to estimate probabilities p and the optimal distribution's histogram for probabilities q. This gives us the KL divergence for output neurons. To obtain it for the initial neurons, we need to repeat the same calculation, but this time using the dependent histograms.

(4) From entropy to Landauer cost
We will use the process described in 3.2 in this step. For every layer, we consider its type and compute the Landauer cost of each neuron composing it accordingly. Finally, we multiply the results by k_bT to obtain a cost in energy.

(5) From KL divergence to Mismatch cost
This step is quite trivial as we only need to subtract the KL divergence from the initial neurons to the one from the final neurons. In a similar fashion to (4) we finish by multiplying everything by k_bT

3.4 Heuristics

As we will see in the Performance chapter 4.2 the computations required to compute the Landauer and Mismatch cost estimations are intensive and can take quite a long time when the number of samples gets high. To palliate this problem we propose heuristics for each layer type that reasonably estimate both costs if the inputs follow a Normal distribution. First, we consider the Landauer cost and then we apply the same process to obtain the Mismatch cost.

To simplify computation, we will assume that each neuron in the network has roughly the same amount of entropy S_{Normal}. In practice, we will see that this assumption works reasonably well on Normal distributions. This could be because a sum of Normal random variables is a Normal random variable thus when the ANN performs weighted sums the entropy is more or less conserved. Now, if the input neurons each follow a Normal distribution of mean μ and standard deviation σ we know their analytic entropy [17].

$$S_{\text{Normal}} = \frac{1}{2}\ln(2\pi e\sigma^2)$$

As we have seen in the section on layer costs, no matter the final neuron in a dense layer, the set of initial neurons is always the entirety of the previous neurons. If we have M neurons in the previous layer we can use our assumption to say that their total entropy is approximately MS_{Normal} and that the entropy of a final neuron is similar to S_{Normal}. Therefore the drop in entropy for each final neuron will be close to $(M-1)S_{\text{Normal}}$. We can use that knowledge to estimate the Landauer cost of a dense layer of N neurons.

$$\mathcal{L}_{\text{dense}} = N(M-1)S_{\text{Normal}} \tag{3.1}$$

Now we can apply the same reasoning to estimate the Mismatch cost. We suppose that the optimal distribution is distributed as $\mathcal{N}(\mu_{opt}, \sigma_{opt})$ and we use the analytic value of the KL divergence to obtain a similar formula.

$$D_{\text{Normal}} = \ln(\frac{\sigma_{\text{opt}}}{\sigma}) + \frac{\sigma^2 + (\mu - \mu_{\text{opt}})^2}{2\sigma_{\text{opt}}^2} - \frac{1}{2}$$

$$\mathcal{M}_{\text{dense}} = N(M-1)D_{\text{Normal}} \tag{3.2}$$

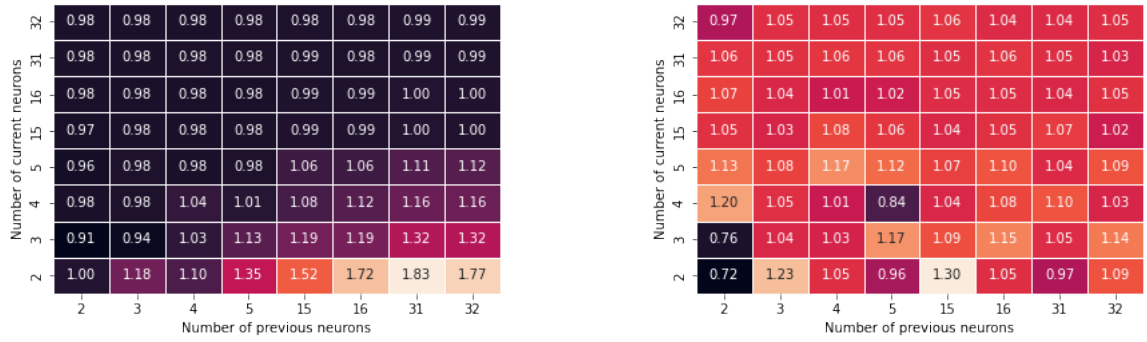


Figure 3.8: Heatmap of the relation $\frac{\mathcal{L}_{\text{emp}}}{\mathcal{L}_{\text{dense}}}$ (left) and $\frac{\mathcal{M}_{\text{emp}}}{\mathcal{M}_{\text{dense}}}$ (right) of a dense layer

On figure 3.8 we computed the ratio between the Landauer cost computed using samples and the one computed using our heuristic. To make these computations we used the same settings as described in section 3.2. Also $\mathcal{L}_{\text{emp}}$ and $\mathcal{M}_{\text{emp}}$ are computed using the pipeline described in section 3.3. The matrix of the Landauer ratios is split in half. On the lower right part, the estimation is quite poor so with should avoid using our heuristic when the number of previous neurons is much higher than the number of current neurons. However, in cases where we halve the number of neurons the heuristics works fine and it works even better on other cases since every ratio is close to 1 on the upper left part of the matrix. The Mismatch ratios don't have this clear split tendency but overall the results aren't as good. Still, it has reasonable performance on high neuron count.

If we want to apply the same formula to the pooling layer, we first need to determine the amount of initial and final neurons since here they don't correspond to M and N. We have as many initial neurons as they are neurons in the pooling filter. For a pool size of P, this is P^2 initial neurons. We have a number of final neurons equal to the number of pooling filters

we can fit inside the previous layer, this is $\frac{M}{P^2}$ rounded down. When testing this formula on pooling layers, we found that the entropy of the final neurons didn't exactly fit the assumption. Instead, depending on the size of the filter they only kept a fraction of the initial entropy and the value was always close to $\frac{S_{\text{Normal}}}{P}$. Therefore we will have a higher drop in entropy than originally expected. All of this leads to our formula for the pooling layer.

$$\mathcal{L}_{\text{pool}} = \lfloor \frac{M}{P^2} \rfloor (P^2 - \frac{1}{P}) S_{\text{Normal}} \tag{3.3}$$

When we test a similar formula to estimate the Mismatch cost, we always obtain results that are three times larger than the empirical results. Even after research, we cannot explain where this factor comes from but we decided to correct the formula for the sake of completion. However, we advice using this corrected formula with great care!

$$\mathcal{M}_{\text{pool}} = \lfloor \frac{M}{P^2} \rfloor (P^2 - \frac{1}{P}) S_{\text{Normal}}/3 \tag{3.4}$$

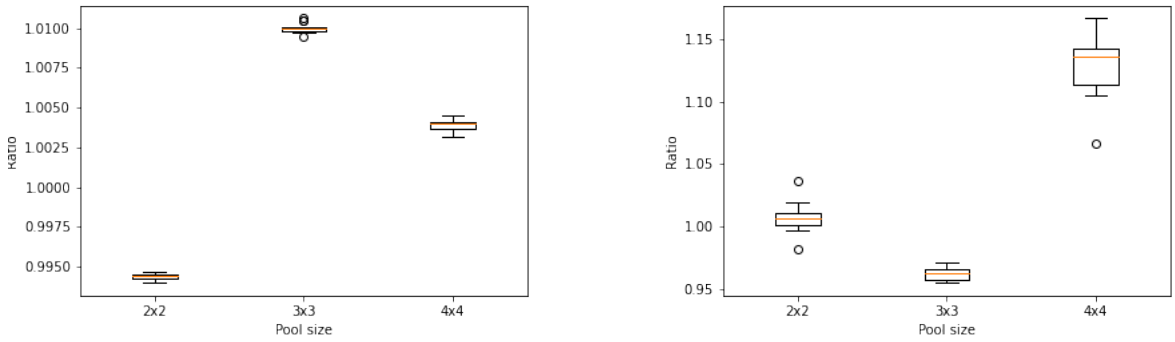


Figure 3.9: Boxplot of the relation $\frac{\mathcal{L}_{\text{emp}}}{\mathcal{L}_{\text{pool}}}$ (left) and $\frac{\mathcal{M}_{\text{emp}}}{\mathcal{M}_{\text{pool}}}$ (right) of a pooling layer. For each pool size ratios are computed from 4x4 to 16x16 previous layer size.

As we can see on figure 3.9 the Landauer ratios are quite consistent for different input sizes and are all very close to 1. The Mismatch ratios are worse and have a higher spread although they stay relatively close to 1.

To find a formula to estimate the costs of convolution layers we need to take filters into account and reconsider the amount of initials and finals neurons. First, if we assume that each filter is independent from the others, we can simply multiply the cost by the number of filters F. Then the number of initials neurons is similar to the one from the pooling layer except this time we use the kernel size K. For the number of final neurons, it depends on whether we use zero padding or not. If we use it, we have M final neurons, as many as there are neurons in the previous layer. If we don't use any padding, we can only apply convolution on a smaller square restricted by the size of the kernel. In this case, we have $(\sqrt{M} - K + 1)^2$ final neurons. We obtain the formula for the Mismatch cost in the exact same way.

$$\mathcal{L}_{\text{conv}} = \begin{cases} FMK^2 S_{\text{Normal}} & \text{if padding} \\ F(\sqrt{M} - K + 1)^2 K^2 S_{\text{Normal}} & \text{otherwise} \end{cases} \tag{3.5}$$

$$\mathcal{M}_{\text{conv}} = \begin{cases} FMK^2 D_{\text{Normal}} & \text{if padding} \\ F(\sqrt{M} - K + 1)^2 K^2 D_{\text{Normal}} & \text{otherwise} \end{cases} \tag{3.6}$$

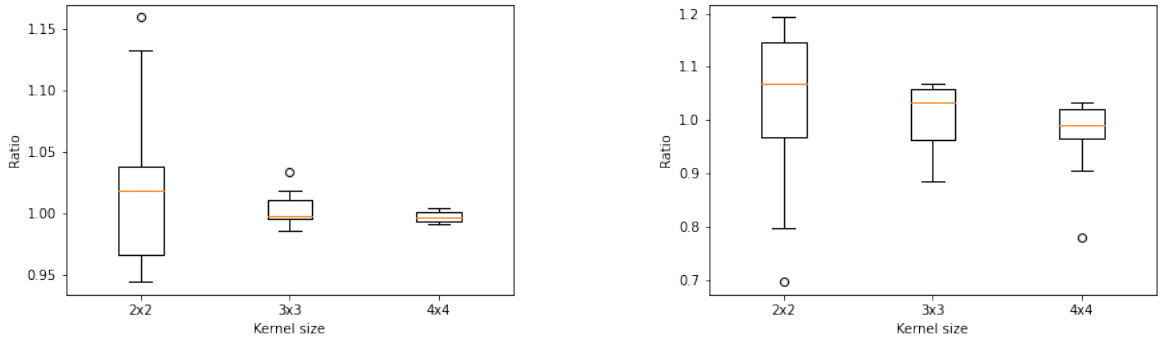


Figure 3.10: Boxplot of the relation $\frac{\mathcal{L}_{\text{emp}}}{\mathcal{L}_{\text{conv}}}$ (left) and $\frac{\mathcal{M}_{\text{emp}}}{\mathcal{M}_{\text{conv}}}$ (right) of a pooling layer. For each pool size, ratios are computed from 4x4 to 16x16 previous layer size.

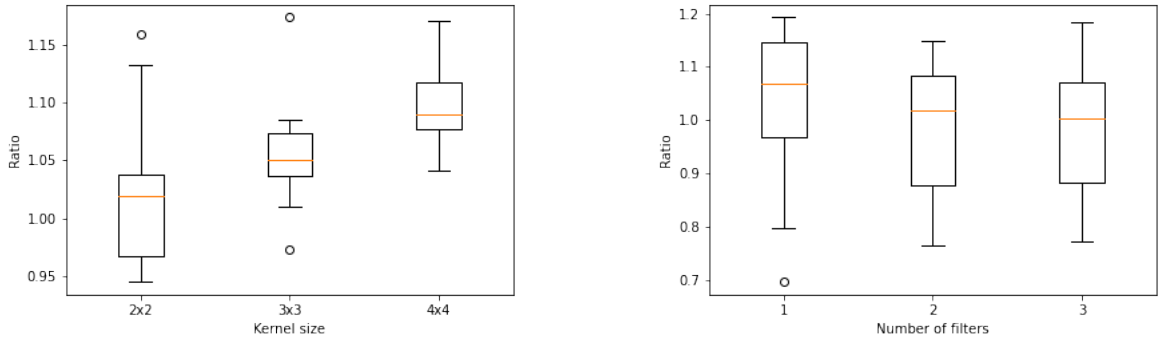


Figure 3.11: Boxplot of the relation $\frac{\mathcal{L}_{\text{emp}}}{\mathcal{L}_{\text{conv}}}$ (left) and $\frac{\mathcal{M}_{\text{emp}}}{\mathcal{M}_{\text{conv}}}$ (right) of a pooling layer. For a number of filters, ratios are computed from 4x4 to 16x16 previous layer size.

By far this is the worst set of results. Both on figure 3.10 and 3.11, the spread is much higher than on the other layer type. Although it seems that the higher the kernel size the more stable the results. However, the mean is somewhat close to 1 in every case. This heuristic could still be used to obtain an idea of the scale of the Landauer and Mismatch costs even though the value won't be accurate.

Chapter 4

Software development

The full code of the library can be found in appendix A or on the following **github**.

The first section of this chapter is about the main functions of the library and their specifications. Then, in the performance section, we go over their execution times and how we could improve them.

4.1 Implementation

The choices of language and library were axed on accessibility. Python is one if not the most used language for machine learning, although the performance aren't the best. The same can be said for Keras. It might have worse performance than other libraries such as Tensorflow but it compensates this flaw with easy-to-use API and a large user base.

The library comprises 6 main functions.

1. samples2Landauer
2. samples2Mismatch
3. samples2LandauerPerNeuron
4. samples2MismatchPerNeuron
5. heuristicLandauer
6. heuristicMismatch

The first two returns the total costs already converted in eV as a float number while the second two returns a multidimensional array giving the individual cost of each neuron. The shape of this multidimensional array is the following: n the first dimension of this array there is a subarray corresponding to each layer. The shape of these subarrays corresponds to the shape of their associated layer. The last two return a rough estimation of the cost for each layer

The first four functions takes as its first argument a Keras model consisting only of the basic layers described in section 3.2. Their second argument is a list of samples from the input distribution, it can be the test set for example. The functions returning a mismatch cost take

two additional arguments one of which is optional. The first is a list of samples from what is expected to be the optimal distribution. The optional argument is a flag stating whether or not we should use our approximation (i.e. considering all distribution as independent) to compute the Mismatch cost. Finally, they all take one more optional argument the size of the bins. It is recommended to tweak the value of this parameter and check if the costs remain roughly constant. If this isn't the case, you might need to find a more suitable value for the bin size until the results converge. The last two functions take mean and standard deviation of the actual and optimal Normal distribution as argument.

4.2 Performance

When comparing the execution time of the different parts of the computation pipeline, we easily identify the bottleneck: (1) from samples to histograms. This step can take several seconds while the other steps combined are 100 to 1000 times faster. It makes sense, running an input through a neural network is a costly operation and we have to repeat it for every sample. On the other hand, the other steps apply faster operations on a smaller amount of data as histograms store data efficiently. The heuristics functions run extremely fast as we only compute a couple of products.

When considering the problem, multithreading and/or multiprocessing seems pretty appealing. Since samples are independent, we can divide them among threads that will apply pipeline operation individually. Given the bottleneck, we need to assign one neural network per thread; depending on the size of the ANN it can be memory-wise costly. Also, it is important to note that Python isn't known for its multithreading efficiency. Nevertheless, this is a direction that could be explored in the future to improve performance.

Chapter 5

Conclusion

This research aimed at developing tools to measure the inevitable thermodynamics costs of trained artificial neural networks. The motivation was to use these tools to grasp a better idea of the scale of the energetic limitations imposed by thermodynamics. We also wanted a better understanding of the implications of different neural structures and we wanted to know if we could choose a specific structure to reduce the impact of these limitations.

We defined a framework to estimate both the Landauer and the Mismatch costs of a basic ANN based on numerical samples. After experimentation, we discovered that the minimum of energy required to run a simple layer ranges in most cases from a few to a hundred electronvolt (i.e. from 10^{-19} to 10^{-17} Joules) depending on the number of neurons and the distribution.

In this work, we only covered the reshape, dense, pooling, and convolution layers but future studies could analyse even more layer types. When comparing the results for different sizes and types of layers, we realized that some configurations had lower costs than others. If one wants to minimize the lower bound in energy of an ANN he should consider taking the following considerations into account. In dense layers, high change in the amount of neurons is favored and for an equal change, the layer with the lower number of final neurons will have lower costs. For the same receptive field, a chain of convolution layer will have a lower Landauer and Mismatch cost than a single layer with a large kernel. Pooling layers tend to have lower costs overall.

When experimenting, we discovered that the costs were consistent when the input of an ANN followed a Normal distribution. We proposed some heuristics to compute the lower bound much faster at the cost of some precision. In some cases, this can be used to obtain an order of magnitude. It would be interesting to see if we could tweak these heuristics to fit other distributions.

We have successfully designed a Python library to estimate both the Landauer and Mismatch costs from numerical samples and an ANN from the Keras library. However, improvements are always possible. Successfully implementing a multithreaded version of the program could drastically improve performance and we would reach even higher precision.

Appendix A

Python implementation

```
__author__ = "Schorochoff Dimitri"
__version__ = "1.1"

import tensorflow as tf
from tensorflow import keras
from keras import backend as K
from tensorflow.keras import layers
import numpy as np
from tqdm import tqdm

_K = 1.380649 * 10 ** (-23)
_J2EV = 1 / (1.602176634 * 10 ** (-19))
_ROOMTEMP = 293.15
_ENTROPY2EV = _K * _ROOMTEMP * _J2EV


################################################################################
###                            MAIN FUNCTIONS                                ###
################################################################################

def samples2Landauer(model, inputs, bin_size=0.01, verbose=True):
    """
    samples2Landauer(model = keras.Sequential([layers.Input((n1)),layers.Dense(n2),]),
     (s1, s2, ...), bin_size=0.01)

    Compute the total Landauer cost in eV that yield a basic keras model
    when evaluating an input of a given distribution.

    Parameters
    ----------
    model : a tf.keras.Model class
        the model over which the Landauer cost is computed. It must only be composed of
        tf.keras.layers.Dense, keras.layers.Conv2D, tf.keras.layers.MaxPooling2D,
        tf.keras.layers.AveragePooling2D, tf.keras.layers.Flatten and/or
        tf.keras.layers.Reshape layers
    inputs : ndarray
        An arrays of samples for the given distribution. Each sample s1, s2,... must have
        the same shape as the model input.
    bin_size : float, optional
        Define the size of the bins that are used to compute discretized entropy.
    verbose: boolean, optional
        Define whether or not to show a progressbar

    Returns
    -------
    res : float
        The total Landauer cost.
    """

    h = samples2LandauerPerNeuron(model, inputs, bin_size=bin_size, verbose=verbose)
    return sum_matrix(h, get_shapes(model))


def samples2Mismatch(model, inputs, opti_inputs, bin_size=0.01, use_approx=True, verbose=True)
    """
```

```
    samples2Mismatch(model = keras.Sequential([layers.Input((n1)),layers.Dense(n2),]),
     (s1, s2, ...), (q1, q2, ...), bin_size=0.01)

    Compute the total Mismatch cost in eV that yield a basic keras model
    when evaluating an input of a given distribution if the entropy production
    is minimized by a given optimal distribution.

    Parameters
    ----------
    model : a tf.keras.Model class
        the model over which the Mismatch cost is computed. It must only be composed of
        tf.keras.layers.Dense, keras.layers.Conv2D, tf.keras.layers.MaxPooling2D,
        tf.keras.layers.AveragePooling2D, tf.keras.layers.Flatten and/or
        tf.keras.layers.Reshape layers
    inputs : ndarray
        An array of samples for the given distribution. Each sample s1, s2,...
        must have the same shape as the model input.
    opti_inputs : ndarray
        An array of samples for the given optimal distribution. Each sample q1, q2,...
        must have the same shape as the model input.
    bin_size : float, optional
        Define the size of the bins that are used to compute discretized entropy.
    use_approx: bool, optional
        If True, approximate the cost by considering every distribution as independant.
    verbose: boolean, optional
        Define whether or not to show a progressbar
    Returns
    -------
    res : float
        The total Mismatch cost.
    """
    h = samples2MismatchPerNeuron(model, inputs, opti_inputs, bin_size=bin_size, use_approx=us
    return sum_matrix(h, get_shapes(model))


def samples2LandauerPerNeuron(model, inputs, bin_size=0.01, verbose=True):
    """
    Compute the Landauer cost in eV of each neuron that yield a basic keras model
    when evaluating an input of a given distribution.

    Parameters
    ----------
    See documentation of samples2Landauer

    Returns
    -------
    res : ndarray
        An array (l1,l2,...) containing the Landauer cost of each neuron.
        l1 has the same shape as layer 1 and so on.
    """
    s = get_shapes(model)
    h = _samples2Histos(model, inputs, bin_size=bin_size, verbose=verbose)
    _histos2Entropy(h, s, entropy_func=CAE_entropy)
    _entropy2Landauer(model, h)
```

```
    return h


def samples2MismatchPerNeuron(model, inputs, opti_inputs, bin_size=0.01, use_approx=True, verb
    """
    Compute the Mismatch cost in eV of each neuron that yield a basic keras model
    when evaluating an input of a given distribution.

    Parameters
    ----------
    See documentation of samples2Mismatch

    Returns
    -------
    res : ndarray
        An array (l1,l2,...) containing the Mismatch cost of each neuron.
        l1 has the same shape as layer 1 and so on.
    """
    s = get_shapes(model)
    # Compute the KL divergence of final neuron
    h_nocorr = _samples2Histos(model, inputs, bin_size=bin_size, verbose=verbose)
    opti_h_nocorr = _samples2Histos(model, opti_inputs, bin_size=bin_size, verbose=verbose)
    _histos2KLDivergence(h_nocorr, opti_h_nocorr, s)

    if use_approx:
        _entropy2Landauer(model, opti_h_nocorr)
        opti_h_nocorr[0] = []  # Input layer has no Mismatch cost
        return opti_h_nocorr

    else:
        # Compute KL divergence of the set initial neurons if they are dependant
        h = _samples2HistosCorrelated(model, inputs, bin_size=bin_size)
        opti_h = _samples2HistosCorrelated(model, opti_inputs, bin_size=bin_size)
        _histos2KLDivergence(h, opti_h, s)

        # Compute the Mismatch cost from KL divergence
        map_matrixes(opti_h, opti_h_nocorr, s, _KL2Mismatch)

        opti_h[0] = []  # Input layer has no Mismatch cost

        return opti_h


def heuristicLandauer(model, std):
    """
    Compute estimation based on a heuristic of the Landauer cost of each layer in eV
    that yields a basic keras model when the input follows a Normal distribution.

    Parameters
    ----------
    model : a tf.keras.Model class
        the model over which the Landauer cost is computed. It must only be composed of
        tf.keras.layers.Dense, keras.layers.Conv2D, tf.keras.layers.MaxPooling2D,
        tf.keras.layers.AveragePooling2D, tf.keras.layers.Flatten and/or
        tf.keras.layers.Reshape layers
```

```
    std : float
        The standard deviation of the input Normal distribution

    Returns
    -------
    res : ndarray
        The Landauer cost in eV of each layer.
    """
    costs = [0]
    S_Normal = entropy_Normal(std)
    shapes = get_shapes(model)
    for i in range(len(model.layers)):
        previousShape = shapes[i]
        shape = shapes[i + 1]
        layer = model.layers[i]

        if isinstance(layer, tf.keras.layers.Dense):
            costs.append(shape[1]*previousShape[1]*S_Normal*_ENTROPY2EV)

        elif isinstance(layer, tf.keras.layers.Conv2D):
            K = layer.kernel_size[0]
            costs.append(shape[3] * (previousShape[1]-K+1)**2 * K**2 * S_Normal * _ENTROPY2EV)

        elif isinstance(layer, tf.keras.layers.MaxPooling2D) or\
                isinstance(layer, tf.keras.layers.AveragePooling2D):
            P = layer.pool_size[0]
            costs.append(previousShape[1]*previousShape[2]//(P**2) * (P**2 - 1/P)* S_Normal *

        elif isinstance(layer, tf.keras.layers.Flatten) or\
                isinstance(layer, tf.keras.layers.Reshape):
            costs.append(0)

    return costs

def heuristicMismatch(model, mean, std, mean_opt, std_opt):
    """
    Compute estimation based on a heuristic of the Landauer cost of each layer in eV
    that yields a basic keras model when the input follows a Normal distribution.

    Parameters
    ----------
    model : a tf.keras.Model class
        the model over which the Landauer cost is computed. It must only be composed of
        tf.keras.layers.Dense, keras.layers.Conv2D, tf.keras.layers.MaxPooling2D,
        tf.keras.layers.AveragePooling2D, tf.keras.layers.Flatten and/or
        tf.keras.layers.Reshape layers
    mean : float
        The mean of the input Normal distribution
    std : float
        The standard deviation of the input Normal distribution
    mean_opt: float
        The mean of the optimal Normal distribution
    std_opt: float
        The standard deviation of the optimal Normal distribution
```

```
    Returns
    -------
    res : ndarray
        The Mismatch cost in eV of each layer.
    """
    costs = [0]
    D_Normal = KL_divergence_Normal(mean, std, mean_opt, std_opt)
    shapes = get_shapes(model)
    for i in range(len(model.layers)):
        previousShape = shapes[i]
        shape = shapes[i + 1]
        layer = model.layers[i]

        if isinstance(layer, tf.keras.layers.Dense):
            costs.append(shape[1]*previousShape[1]*D_Normal*_ENTROPY2EV)

        elif isinstance(layer, tf.keras.layers.Conv2D):
            K = layer.kernel_size[0]
            costs.append(shape[3] * (previousShape[1]-K+1)**2 * K**2 * D_Normal * _ENTROPY2EV)

        elif isinstance(layer, tf.keras.layers.MaxPooling2D) or\
                isinstance(layer, tf.keras.layers.AveragePooling2D):
            P = layer.pool_size[0]
            costs.append(previousShape[1]*previousShape[2]//(P**2) * (P**2 - 1/P)* D_Normal *

        elif isinstance(layer, tf.keras.layers.Flatten) or\
                isinstance(layer, tf.keras.layers.Reshape):
            costs.append(0)

    return costs


################################################################################
###                           ENTROPY FUNCTIONS                              ###
################################################################################

def CAE_entropy(counts):
    """
    This is a Python translation of
    https://github.com/cran/entropy/blob/master/R/entropy.ChaoShen.R

    Return the CAE entropy of a histogram.

    Parameters
    ----------
    counts: ndarray. A list containing the number of occurence of every bin

    Returns
    -------
    res : float
    The CAE entropy of the histogram
    """
    counts = counts[counts > 0]
    n = np.sum(counts)
    if (n == 0): return 0
```

```
    p = counts / n

    f1 = np.count_nonzero(counts == 1)
    if (f1 == n): f1 = n - 1

    C = 1 - f1 / n
    pa = C * p
    la = (1 - (1 - pa) ** n)

    return -np.sum(pa * np.log(pa) / la)


def KL_divergence(countP, countQ):
    """
    Return the KL divergence between two given histograms.

    Parameters
    ----------
    countP,countQ: ndarray. A list containing the number of occurence of every bin

    Returns
    -------
    res : float
    The KL divergence between the two given histograms.
    """
    p = countP[countP > 0]
    p = p / np.sum(countP)

    q = countQ[countQ > 0]
    q = q / np.sum(countQ)

    return np.sum(p * np.log(p / q))

def _KL2Mismatch(KL_i, KL_f):
    return (KL_i - KL_f) * _ENTROPY2EV

def entropy_Normal(sigma):
    """
    Return the analytical entropy of a Normal distribution of standard deviation sigma
    """
    return 0.5 * np.log(2 * np.pi * np.e * sigma**2)

def KL_divergence_Normal(mu1, sigma1, mu2, sigma2):
    """
    Return the analytical KL divergence between a Normal distribution of mean mu1 and
    standard deviation sigma1 and another Normal distribution of mean mu2 and standard deviati
    sigma2
    """
    return np.log(sigma2/sigma1) + (sigma1**2+ (mu2 - mu1)**2)/(2*sigma2**2) - 0.5

##########################################################################
###                  MULTIDIMENSION ARRAY FUNCTIONS                    ###
##########################################################################

def get_shapes(model):
```

```
    """
    Give an array of shapes corresponding to the layers of a given model

    Parameters
    ----------
    model : a tf.keras.Model class

    Returns
    -------
    res : ndarray
        Return an array (s1,s2,...) where s1 is a tuple containing the shape of layer 1.
        The input layer is counted as the first layer
    """
    s = [(1, *(model.input.shape[1:]))]
    for layer in model.layers:
        s.append((1, *(layer.output_shape[1:])))

    return s


def index_multidim(shape):
    """
    Give a generator giving all the index needed to iterate over
    a multidimensional array of the given shape

    Parameters
    ----------
    shape : a tuple (d1,d2,...) where d1 correspond to the first dimension and so on.

    Returns
    -------
    res : generator
        Return a generator giving d1*d2*... indexes.
        An index is a tuple of the lenght as the shape.
    """

    def index_multidim_recursive(x, t):
        if (len(t) == 0):
            yield x
            return

        for i in range(t[0]):
            yield from index_multidim_recursive((*x, i), t[1:])

    yield from index_multidim_recursive([], shape)


def get_matrix(matrix, ind):
    """
    Return a value stored at a specific index of a multidimensional array

    Parameters
    ----------
    matrix : a multidimensional matrix having a depth of N.
```

```
    ind : a tuple of length N indicating which part of matrix should be returned

    Returns
    -------
    res : value
        Return the value stored in the array the type depends on what is stored
    """
    for i in ind:
        matrix = matrix[i]
    return matrix


def set_matrix(matrix, ind, v):
    """
    Store a value at a specific index of a multidimensional array

    Parameters
    ----------
    matrix : a multidimensional array having a depth of N.

    ind : a tuple of length N indicating which part of matrix should be returned

    v : the value to store
    """
    for i in ind[:-1]:
        matrix = matrix[i]
    matrix[ind[-1]] = v


def sum_matrix(matrixes, shapes):
    """
    Return the sum of all values contained inside a list of multidimensional arrays

    Parameters
    ----------
    matrixes : a list of N multidimensional array having depth of (d1,d2,...,dN).
    Arrays must only contains float/int

    shapes : a list of N tuples. The first tuples gives the shape of
    the first multidimensional array and must therefore have length d1.

    Returns
    -------
    res : float/int
        Return sum of all values stored in matrixes
    """
    res = 0
    for i in range(len(matrixes)):
        if len(matrixes[i]) == 0: continue

        for ind in index_multidim(shapes[i]):
            res += get_matrix(matrixes[i], ind)

    return res
```

```
def map_matrixes(matrixes1, matrixes2, shapes, map):
    """
    Iterate over every index of a given list of shape. For every index store
    and apply the operation map

    Parameters
    ----------
    matrixes1: a list of N multidimensional array having depth of (d1,d2,...,dN).
    Arrays must only contains float/int

    matrixes2: same as matrixes1

    shapes : a list of N tuples. The first tuples gives the shape of
    the first multidimensional array and must therefore have length d1.

    map : func. Mapping function applied.
    It must take 2 float/int as argument

    Returns
    -------
    Store result of the mapping in matrixes1
    """
    if matrixes1 == [] or matrixes2 == []: return []

    for i in range(len(matrixes1)):
        if len(matrixes1[i]) == 0 or len(matrixes2[i]) == 0: continue
        for ind in index_multidim(shapes[i]):
            v = map(get_matrix(matrixes1[i], ind), get_matrix(matrixes2[i], ind))
            set_matrix(matrixes1[i], ind, v)


#############################################################################
###                        HISTOGRAM FUNCTIONS                            ###
#############################################################################

def _init_histogram_matrix(shapes):
    """
    Initialize a list of multidimensional array of given shapes
    filled with empy dictionary

    Parameters
    ----------
    shapes: a list of N tuples.
    The first tuple gives the shape of the first array and so on

    Returns
    -------
    res : ndarray
        Return a list of N multidimensional arrays with shape
        corresponding to shapes
    """
    lst = [np.full(s, {}) for s in shapes]

    for i in range(len(lst)):
```

```
        for ind in index_multidim(shapes[i]):
            # Otherwise all dictionary have the same ref
            set_matrix(lst[i], ind, {})

    return lst

def _add2histogram(histo, datapoint, bin_size=0.01):
    """
    Add a datapoint to a given histogram.
    Convert the value of the datapoint to its corresponding bin number
    Bounds of bin number i are [ (i-0.5)*bin_size, (i+0.5)*bin_size )

    Parameters
    ----------
    histo: dictionnary. The histogram to add the datapoint.

    datapoint: tuple. The datapoint (x1,x2,...) to add.
    x1 correspond to it's position in the first dimension and so on.

    bin_size : float, optional
    Define the size of the bins that are used to compute discretized entropy.
    """
    position = tuple([round(d / bin_size) for d in datapoint])

    if position in histo:
        histo[position] += 1
    else:
        histo[position] = 1

def _add2layer_histo(histos, shape, data, bin_size=0.01):
    """
    Iterate through every histogram in a multidimensional array and
    add the corresponding datapoint in another multidim array

    Parameters
    ----------
    histos: ndarray. A multidimensional array containing the histograms (dictionary)

    shape: tuple. The shape of the multidimensional array

    data: ndarray. A multidimensional array of datapoint to add.
    In datapoint (x1,x2,...) x1 correspond to its position in the first dimension and so on.

    bin_size : float, optional
    Define the size of the bins that are used to compute discretized entropy.
    """
    for ind in index_multidim(shape):
        _add2histogram(get_matrix(histos, ind), (get_matrix(data, ind),), bin_size=bin_size)


def _match_histos(histo1, histo2):
    """
    Convert two histograms from dictionary format to array format.
    Keep only the keys that match and align the index of their corresponding
    value in the arrays
```

```
    Parameters
    ----------
    histo1: dictionnary. The first histogram to match

    histo2: dictionnary. The second histogram to match

    Returns
    -------
    res1 : ndarray
        Return a histogram corresponding to histo1 as array format.
        The value at index i is the number of occurence of key K_i

    res2 : ndarray
        Same as res1 except it is corresponding to histo2
    """

    inter = list(set(histo1.keys()).intersection(set(histo2.keys())))

    newH1 = np.zeros(len(inter))
    newH2 = np.zeros(len(inter))

    for k in range(len(inter)):
        newH1[k] = histo1[inter[k]]
        newH2[k] = histo2[inter[k]]

    return newH1, newH2

##########################################################################
###                        PIPELINE FUNCTIONS                          ###
##########################################################################

def _input_outputs_func(model):
    """
    Code inspired from
    https://stackoverflow.com/questions/41711190/keras-how-to-get-the-output-of-each-layer

    Return a function f corresponding to a keras model.

    Parameters
    ----------
    model : a keras model. The model you want to know the neuron's output of.

    Returns
    -------
    res : a function having the following specification

        Function f(input)
        This function is based of a keras model M from an input
        it return the output of each neuron.

        Parameters
        ----------
        input : ndarray. The input of the model you want to know the neurons's output of.
```

```
        Returns
        -------
        res : ndarray
        An array (l1,l2,...) containing the output (float) of each neuron.
        l1 has the same shape as layer 1 and so on.
    """
    inp = model.input  # input placeholder
    outputs = [model.input] + [layer.output for layer in model.layers]
    functor = K.function([inp], outputs)  # evaluation function

    return functor


def _samples2Histos(model, inputs, bin_size=0.01, verbose=True):
    """
    Run a set of samples through a keras model and store for each neuron a histogram
    storing the occurence of the neuron output in each bin.

    Parameters
    ----------
    See documentation of samples2Landauer

    Returns
    -------
    res : ndarray
        An array (l1,l2,...) containing a dictionary of each neuron.
        l1 has the same shape as layer 1 and so on. Each key of a dictionary is
        a bin number and the corresponding value is the number of samples that
        felt within the bounds of the bin.
        Bounds of bin number i are [ (i-0.5)*bin_size, (i+0.5)*bin_size )
    """
    shapes = get_shapes(model)
    histos = _init_histogram_matrix(shapes)
    functor = _input_outputs_func(model)

    with tqdm(total=len(inputs), disable= not verbose) as pbar:
        for i in range(len(inputs)):
            inp = inputs[i]
            out = functor([np.array([inp])])

            for i in range(len(out)):
                _add2layer_histo(histos[i], shapes[i], out[i], bin_size=bin_size)

            if verbose: pbar.update(1)

    return histos


def _samples2HistosCorrelated(model, inputs, bin_size=0.01, verbose=True):
    """
    Run a set of samples through a keras model and store for each neuron a histogram
    storing the occurence of the set of linked initial neurons's output in
    each multidimensional bin.

    Parameters
```

```
    ----------
    See documentation of samples2Landauer

    Returns
    -------
    res : ndarray
        An array (l1,l2,...) containing a dictionary of each neuron.
        l1 has the same shape as layer 1 and so on. Each key of a dictionary is
        a tuple of bin number (b1,b2,...). b1 is the bin number corresponding to
        the output of the first initial neuron. The value corresponding to the key
        is the number ofsamples that felt within the bounds of this set of bins.
        Bounds of bin number i are [ (i-0.5)*bin_size, (i+0.5)*bin_size )

        Note: l1 is set to [] has it has no prior initial neurons.
    """
    shapes = get_shapes(model)
    histos = _init_histogram_matrix(shapes)
    functor = _input_outputs_func(model)

    with tqdm(total=len(inputs), disable=not verbose) as pbar:
        for k in range(len(inputs)):  # Iterate through all sample
            inp = inputs[k]
            out = functor([np.array([inp])])  # Compute the neuron output in the model

            for i in range(len(model.layers) - 1, -1, -1):  # Iterate through all layers
                previousShape = shapes[i]
                shape = shapes[i + 1]
                layer = model.layers[i]

                if isinstance(layer, tf.keras.layers.Dense):  # Check the layer type
                    for ind in index_multidim(shape):  # Iterate through all final neuron
                        prev_result = []

                        # Iterate through all linked initial neuron
                        # Add their output to a list
                        for prev_index in index_multidim(previousShape):
                            prev_result.append(get_matrix(out[i], prev_index))

                        # Add the list of output to the histogram
                        _add2histogram(get_matrix(histos[i + 1], ind), tuple(prev_result),
                                       bin_size=bin_size)

                elif isinstance(layer, tf.keras.layers.Conv2D):
                    kernel = layer.kernel_size
                    stride = layer.strides

                    for ind in index_multidim(shape):
                        prev_result = []

                        for n_filter in range(previousShape[3]):  # All filter index
                            for m in range(kernel[0]):  # 2nd dim of the kernel
                                for n in range(kernel[1]):  # 1st dim of the kernel
                                    prev_index = (ind[0], ind[1] * stride[0] + m,
                                                  ind[2] * stride[1] + n, n_filter)
                                    prev_result.append(get_matrix(out[i], prev_index))
```

```
                        _add2histogram(get_matrix(histos[i + 1], ind), tuple(prev_result),
                                       bin_size=bin_size)

                elif isinstance(layer, tf.keras.layers.MaxPooling2D) or \
                        isinstance(layer,tf.keras.layers.AveragePooling2D):
                    # Main difference with convolution: only one filter in previous layer
                    kernel = layer.pool_size
                    stride = layer.strides

                    for ind in index_multidim(shape):
                        prev_result = []

                        for m in range(kernel[0]):
                            for n in range(kernel[1]):
                                prev_index = (ind[0], ind[1] * stride[0] + m,
                                              ind[2] * stride[1] + n, ind[3])
                                prev_result.append(get_matrix(out[i], prev_index))

                        _add2histogram(get_matrix(histos[i + 1], ind), tuple(prev_result),
                                       bin_size=bin_size)

                elif isinstance(layer, tf.keras.layers.Flatten) or\
                        isinstance(layer, tf.keras.layers.Reshape):
                    pass  # There is no initial neuron when reshapping

            if verbose: pbar.update(1)

    histos[0] = []  # Input layer has no prior initial neuron
    return histos


def _histos2Entropy(histos, shapes, entropy_func=CAE_entropy, bin_size=0.01, epsilon=0):
    """
    Convert every histogram of a keras model contained in a multidimensional array
    to their respective entropy

    Parameters
    ----------
    histos: ndarray. Often the output of _samples2Histos().
    An array (l1,l2,...) containing a dictionary of each neuron of a keras model.
    l1 has the same shape as layer 1 and so on. Each key of a dictionary is
    a bin number and the corresponding value is the number of samples that felt
    within the bounds of the bin.

    shapes: a list of N tuples. The first tuples gives the shapes of l1 and so on.

    entropy_func: func. An entropy function with specification as below

        Entropy function f(counts)

        Return the entropy of a histogram.

        Parameters
        ----------
```

```
        counts: ndarray. A list containing the number of occurence of every bin

        Returns
        -------
        res : float
        The entropy of the histogram

    bin_size : float, optional
    Define the size of the bins that are used to compute discretized entropy.

    epsilon: float, optional
    Define the threshold under which entropy is considered as degenerate distribution's entrop
    In this case we don't add the correction factor as it would make estimation worth by defau
    we don't use this estimation

    Returns
    -------
    res : ndarray
        An array (l1,l2,...) containing the entropy (float) of each neuron.
        l1 has the same shape as layer 1 and so on.

    Note: the entropy provided still need to be adjusted by substracting log(bin_size)
    to it. This isn't done here as this often cancel itself when computing
    the Landauer cost.
    """
    for i in range(len(histos)):
        for ind in index_multidim(shapes[i]):
            histo = get_matrix(histos[i], ind)

            entropy = entropy_func(np.array(list(histo.values())))
            if np.abs(entropy) >= epsilon: entropy += np.log(bin_size)
            set_matrix(histos[i], ind, entropy)


def _histos2KLDivergence(histos1, histos2, shapes):
    """
    Compare and compute the KL divergence of every histogram of
    two multidimensional arrays. Store the result in histos2

    Parameters
    ----------
    histos1: ndarray. Often the output of _samples2HistosCorrelated().
    An array (l1,l2,...) containing a dictionary of each neuron.
    l1 has the same shape as layer 1 and so on. Each key of a dictionary is a tuple of
    bin number (b1,b2,...). b1 is the bin number corresponding to the output of the
    first initial neuron. The value corresponding to the key is the number of samples
    that felt within the bounds of this set bins.

    histos2: ndarray. Similar as histos1 however the distribution sampled
    by the histograms might differ.

    shapes: a list of N tuples. The first tuples gives the shapes of l1 and so on.

    Returns
    -------
```

```
    For every histogram in histos2, store the KL divergence between the histogram
    at same index in histos1 and this histogram.
    """
    for i in range(len(histos1)):
        if len(histos1[i]) == 0 or len(histos2[i]) == 0: continue
        for ind in index_multidim(shapes[i]):

            histo1 = get_matrix(histos1[i], ind)
            histo2 = get_matrix(histos2[i], ind)

            histo1, histo2 = _match_histos(histo1, histo2)
            divergence = KL_divergence(histo1, histo2)
            set_matrix(histos2[i], ind, divergence)


def _entropy2Landauer(model, entropies):
    """
    Convert every entropy of a keras model contained in a multidimensional array
    to their respective Landauer costs

    Parameters
    ----------
    model : a tf.keras.Model class
        the model over which the Landauer cost is computed. It must only be composed of
        tf.keras.layers.Dense, keras.layers.Conv2D, tf.keras.layers.MaxPooling2D,
        tf.keras.layers.AveragePooling2D, tf.keras.layers.Flatten and/or
         tf.keras.layers.Reshape layers

    entropies: ndarray. Often the output of _histos2Entropy().
    An array (l1,l2,...) containing the entropy (float) of each neuron of a keras model.
    l1 has the same shape as layer 1 and so on.

    Returns
    -------
    res : ndarray
        An array (l1,l2,...) containing the Landauer cost (float) of each neuron.
        l1 has the same shape as layer 1 and so on.

    Note: l1 is set to [] as input layer has no Landauer cost
    """
    shapes = get_shapes(model)
    # iterate through all layer backward, so we don't erase entropy we will use later
    for i in range(len(model.layers) - 1, -1, -1):
        previousShape = shapes[i]
        shape = shapes[i + 1]
        layer = model.layers[i]

        if isinstance(layer, tf.keras.layers.Dense):
            # iterate through all index of the current layer
            for ind in index_multidim(shape):
                entropy = get_matrix(entropies[i + 1], ind)
                prev_entropy = 0

                # In dense layer all neurons in previous layer are initial
                # We sum their entropy
```

```
                for prev_index in index_multidim(previousShape):
                    prev_entropy += get_matrix(entropies[i], prev_index)

                # Set Landauer cost where current entropy was stored.
                set_matrix(entropies[i + 1], ind, max(0, prev_entropy - entropy)*_ENTROPY2EV)

        elif isinstance(layer, tf.keras.layers.Conv2D):
            kernel = layer.kernel_size
            stride = layer.strides

            for ind in index_multidim(shape):
                entropy = get_matrix(entropies[i + 1], ind)
                prev_entropy = 0

                # We iterate through all initial layer and sum their entropy
                for n_filter in range(previousShape[3]):  # All filter index
                    for m in range(kernel[0]):  # 2nd dim of the kernel
                        for n in range(kernel[1]):  # 1st dim of the kernel
                            # Get the index matching kernel position in previous layer
                            # and take stride into account
                            prev_index = (ind[0], ind[1] * stride[0] + m,
                                          ind[2] * stride[1] + n, n_filter)
                            prev_entropy += get_matrix(entropies[i], prev_index)

                set_matrix(entropies[i + 1], ind, max(0, prev_entropy - entropy)*_ENTROPY2EV)

        elif isinstance(layer, tf.keras.layers.MaxPooling2D) or\
                isinstance(layer, tf.keras.layers.AveragePooling2D):
            # Main difference with convolution: only one filter in previous layer
            kernel = layer.pool_size
            stride = layer.strides

            for ind in index_multidim(shape):
                entropy = get_matrix(entropies[i + 1], ind)
                prev_entropy = 0

                for m in range(kernel[0]):
                    for n in range(kernel[1]):
                        prev_index = (ind[0], ind[1] * stride[0] + m,
                                      ind[2] * stride[1] + n, ind[3])
                        prev_entropy += get_matrix(entropies[i], prev_index)

                set_matrix(entropies[i + 1], ind, max(0, prev_entropy - entropy)*_ENTROPY2EV)

        elif isinstance(layer, tf.keras.layers.Flatten) or\
                isinstance(layer, tf.keras.layers.Reshape):
            entropies[i + 1] = []  # No Landauer cost for reshaping

    entropies[0] = []  # Inputs alone have no Landauer costs


# This is an example of use for this library
if __name__ == "__main__":
    n = (4, 4, 1)
    model = keras.Sequential(
```

```
    [
        layers.Input((n)),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dense(2)
    ]
)

n_samples = 10000
bin_size = 0.01

inputs = np.random.normal(0, 1, (n_samples, *(model.input.shape[1:])))
opt_inputs = np.random.normal(0.5, 1.5, (n_samples, *(model.input.shape[1:])))

#h = samples2MismatchPerNeuron(model, inputs, opt_inputs, bin_size=bin_size)
h = samples2Landauer(model, inputs, bin_size=bin_size)
#h = heuristicLandauer(model, 1)

print(h)
```

Appendix B

Summary of *Thermodynamics of computing with circuits*

[2] Following are the key steps of the work from Wolpert and Kolingski. The goal is to ease the understanding of the reader as this is a crucial result for this thesis. Therefore we will use slightly different notation to avoid going over all the technical details. For a more complete and formal definition, it is advised to consult the original paper[2].

B.1 Fixed dynamical system

Consider the stochastic thermodynamics of a system connected to one or more thermodynamic reservoirs. Assume that the system's equations are derived from a *continuous-time Markov chain*(CTMC) and that the thermodynamics coefficient associated to thermodynamic reservoirs obeys *local detail balance* (LDB).

To characterize the system's entropy they define two variables:

- "The term entropy flow (EF) refers to the increase of entropy in all coupled reservoirs.". Noted as $\mathcal{Q}(p)$
- "The term entropy production (EP) refers to the overall increase of entropy, both in the system and in all coupled reservoirs.". Noted as $\sigma(p)$

We can see the first one as the amount of entropy lost in nature. Together they hold the following relationship.

$$\mathcal{Q}(p) = \mathcal{L}(p) + \sigma(p)$$

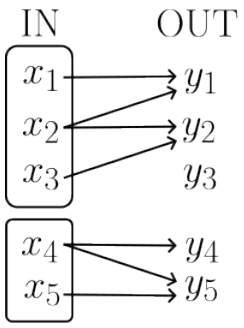
Indeed, if we compensate for the loss in entropy inside the system by adding the Landauer cost to the EP we obtain the EF.

B.2 Island

Islands are a new equivalence relation they define to prove their first theorem.

$$x \sim x' \Leftrightarrow \exists y : P(y|x) > 0, P(y|x') > 0$$

x and x' are similar if they can transition to some state y under the conditional distribution $P(y|x)$. An island $L(P)$ is the connected subset given by the transitive closure of this equivalence relation. If restricted to a subset of state $\mathcal{Z}$ it is noted $L_{\mathcal{Z}}(P)$



	y_1	y_2	y_3	y_4	y_5
x_1	1				
x_2	.4	.6			
x_3		1			
x_4				.9	.1
x_5					1

Figure B.1: Example of island equivalence relation. On the left, boxes indicate islands. On the right, the transition matrix of conditional probability P

An example is illustrated on figure B.1. States x_1 and x_2 both can make a transition to state y_1 while states x_2 and x_3 can transition to state y_2. No other in-state can reach a common out-state with either x_1, x_2, or x_3 therefore they form an island. The other island is formed by x_4 and x_5 using the same logic.

B.3 First theorem

Consider any function $\Gamma : \Delta_{\mathcal{X}} \to \mathbb{R}$ of the form

$$\Gamma(p) := S(Pp) - S(p) + \mathbb{E}_p[f]$$

where $P(y|x)$ is some conditional distribution of $y \in Y$ given $x \in X$ and $f : X \to R \cup \infty$ is some function. Let $\mathcal{Z}$ be any subset of X such that $f(x) < \infty$ for $x \in \mathcal{Z}$, and let $q \in \Delta_{\mathcal{Z}}$ be any distribution that obeys.

$$q^c \in \underset{r:\mathrm{supp}(r)\subseteq c}{\mathrm{argmin}}\ \Gamma(r), \forall c \in L_{\mathcal{Z}}(P)$$

Then, each q^c will be unique, and for any p with supp p $\subseteq \mathcal{Z}$,

$$\Gamma(p) = D(p\|q) - D(P\|Pq) + \sum_{c\in L_{\mathcal{Z}}(P)} p(c)\Gamma(q^c).$$

If we set $\mathcal{X} = \mathcal{Y} = \mathcal{Z}$ and $\mathbb{E}_p[f] = \mathcal{Q}$ the theorem leads to a formula expression for the entropy production.

$$\sigma(p) = D(p\|q) - D(P\|Pq) + \sum_{c\in L_{\mathcal{Z}}(P)} p(c)\sigma(q^c) \tag{B.1}$$

The drop in KL divergence is called mismatch cost while the sum is called residual cost.

B.4 Solitary process

The type of gates they consider is modular, meaning that a gate is physically coupled to its direct input and output but not to other gates. Such gates implement a solitary process, here is the definition:

Consider a system that can be decomposed in two separate subsystems A, B such that $\mathcal{X} = \mathcal{X}_A \times \mathcal{X}_B$ with $\mathcal{X}_i$ the state space of system i.

Then a solitary process is a physical process over state space $\mathcal{X}_A \times \mathcal{X}_B$ taking place during $t \in [0, 1]$ having the following properties:

- A evolves independently of B, and B is held fixed: $P(x'_A, x'_B|x_A, x_B) = P_A(x'_A|x_A)\delta(x'_B, x_B)$
- The EF of the process depends only on the initial distribution over $\mathcal{X}_A$, which we indicate with the following notation: $\mathcal{Q}(p) = \mathcal{Q}_A(p_A)$
- The EF is lower bounded by the change in the marginal entropy of subsystem A, $\mathcal{Q}_A(p_A) > S(p_A) - S(P_A p_A)$

Most of the time the Landauer cost of subsystem A will be different than the Landauer cost of the all system. They call this difference Landauer loss.

$$\mathcal{L}^{\text{loss}}(p) := [S(p_A) - S(P_A p_A)] - [S(p_{AB}) - S(P p_{AB})]$$

B.5 Serial-reinitialized circuit

In SR circuit, we consider that gates are

- Run one after another
- Reinitialising the states of their parent after they run

In such a circuit we can expect that the thermodynamic costs remain constant.

B.6 Second theorem

The total EF incurred by running an SR circuit where p is the initial distribution over the joint state of all nodes in the circuit is:

$$\mathcal{Q}(p) = \mathcal{L}(p) + \mathcal{L}_{\text{loss}}(p) + \mathcal{M}(p) + \mathcal{R}(p) \tag{B.2}$$

B.7 Conversion to results used

We need to apply three more changes to obtain equation 2.10: the lower bound we will use during this thesis.

1. Define the circuit Landauer cost.

$$\begin{aligned} \mathcal{L}^{\text{circ}}(p) &:= \mathcal{L}(p) + \mathcal{L}^{\text{loss}}(p) \\ &= \sum_{g \in \text{Gates}} (S(p_{i,}) - S(p_{f,}) \end{aligned} \tag{B.3}$$

   This is the sum of Landauer cost over the set of all non-wire gates: Gates.

2. Transform equality to inequality by remembering that $\mathcal{Q}(p)$, is the energy lost to thermal reservoirs (i.d in nature)

3. Drop the residual costs (i.e the last term) as it is very hard to compute.

Bibliography

[1] R. Landauer, "Irreversibility and Heat Generation in the Computing Process," vol. 5, pp. 183–191, 1961.

[2] D. H. Wolpert and A. Kolchinsky, "Thermodynamics of computing with circuits," *New Journal of Physics*, vol. 22, p. 063047, June 2020.

[3] N. Beuseling, *Stochastic thermodynamics of Machine Learning : the inevitable cost of neural networks*. PhD thesis, 2021.

[4] K. Sharp and F. Matschinsky, "Translation of Ludwig Boltzmann's Paper "On the Relationship between the Second Fundamental Theorem of the Mechanical Theory of Heat and Probability Calculations Regarding the Conditions for Thermal Equilibrium" Sitzungberichte der Kaiserlichen Akademie der Wissenschaften. Mathematisch-Naturwissen Classe. Abt. II, LXXVI 1877, pp 373-435 (Wien. Ber. 1877, 76:373-435). Reprinted in Wiss. Abhandlungen, Vol. II, reprint 42, p. 164-223, Barth, Leipzig, 1909," *Entropy*, vol. 17, pp. 1971–2009, Apr. 2015.

[5] J. W. J. W. Gibbs, *Elementary Principles of Statistical MechanicsDeveloped with Especial Reference to the Rational Foundation of Thermodynamics*. 1902.

[6] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, July 1948.

[7] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "Green500."

[8] C. H. Bennett, "The thermodynamics of computation—a review," *International Journal of Theoretical Physics*, vol. 21, pp. 905–940, Dec. 1982.

[9] C. H. Bennett, "Notes on Landauer's principle, reversible computation, and Maxwell's Demon," *Studies in History and Philosophy of Science Part B: Studies in History and Philosophy of Modern Physics*, vol. 34, pp. 501–510, Sept. 2003.

[10] A. Rex, "Maxwell's Demon—A Historical Review," *Entropy*, vol. 19, p. 240, June 2017.

[11] A. Verga, "Maxwell demon and Landauer principle, https://verga.cpt.univ-mrs.fr/L3-demon.html," 2018.

[12] A. Bérut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz, "Experimental verification of Landauer's principle linking information and thermodynamics," *Nature*, vol. 483, pp. 187–189, Mar. 2012.

[13] Y. Jun, M. Gavrilov, and J. Bechhoefer, "High-Precision Test of Landauer's Principle in a Feedback Trap," *Physical Review Letters*, vol. 113, p. 190601, Nov. 2014.

[14] T. J. Thompson, "Nanomagnet memories approach low-power limit | bloomfield knoble," 2011.

[15] J. Hong, B. Lambson, S. Dhuey, and J. Bokor, "Experimental test of Landauers principle in single-bit operations on nanomagnetic memory bits," *Science Advances*, vol. 2, pp. e1501492–e1501492, Mar. 2016.

[16] I. Csiszár and J. Körner, *Information Theory: Coding Theorems for Discrete Memoryless Systems*. Cambridge: Cambridge University Press, 2 ed., 2011.

[17] T. M. Cover and J. A. Thomas, "ELEMENTS OF INFORMATION THEORY,"

[18] X. Kai, "How to Estimate Entropy Wrong, https://xuk.ai/blog/estimate-entropy-wrong.html."

[19] A. Chao and T.-J. Shen, "Nonparametric estimation of Shannon's index of diversity when there are unseen species in sample," *Environmental and Ecological Statistics*, vol. 10, pp. 429–443, Dec. 2003.

[20] V. Q. Vu, B. Yu, and R. E. Kass, "Coverage-adjusted entropy estimation," *Statistics in Medicine*, vol. 26, no. 21, pp. 4039–4060, 2007.

[21] D. G. Horvitz and D. J. Thompson, "A Generalization of Sampling Without Replacement from a Finite Universe," *Journal of the American Statistical Association*, vol. 47, pp. 663–685, Dec. 1952.

[22] A. Ashbridge and I. B. J. Goudie, "Coverage-adjusted estimators for mark-recapture in heterogeneous populations: Communications in Statistics - Simulation and Computation: Vol 29, No 4,"

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**