Spring 2020

Carbon Footprint of Machine Learning Algorithms

Gigi Hsueh
Bard College

Carbon Footprint of Machine Learning Algorithms

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Gigi Hsueh

Annandale-on-Hudson, New York
May 1, 2020

Abstract

With the rapid development of machine learning, deep learning has demonstrated superior performance over other types of learning. Research made possible by big data and high-end GPU's enabled those research advances at the expense of computation and environmental costs. This will not only slow down the advancement of deep learning research because not all researchers have access to such expensive hardware, but it also accelerates climate change with increasing carbon emissions. It is essential for machine learning research to obtain high levels of accuracy and efficiency without contributing to global warming. This paper discusses some of current approaches in estimating energy consumption. We compare the energy consumption of the training phase of two convolutional neural networks, SimpleNet and AlexNet, using RAPL. Although we weren't able to reproduce the network exactly from their original papers, we found that AlexNet uses more than 6 times as much energy and has more than 6 times as much carbon emission as SimpleNet.

Contents

Acknowledgments

I would like to thank my friends and family for their moral support. I especially want to thank Eileen Brickner, Anna Hallett Gutierrez, Gitta Markó, Shaunessy Renker and Javen Lara for their positive energy, love, and warm encouragements. I also want to thank my advisor Keith O'Hara for mentoring and inspiring me throughout my college life. Finally, I would like to express my deepest gratitude to all the workers on the front lines of the coronavirus pandemic for making all of this possible.

1
Introduction

1.1 Motivation

Over the last 50 years, our climate has changed more rapidly than we've seen in recorded human history. As of 2018, the annual number of extreme meteorological events have doubled since 1980. The rate at which sea level is rising has increased fifty percent in just two decades. The average number of yearly storms and ensuing floods have quadrupled from forty years ago [1]. In its 2013 fifth assessment report from IPCC [2], they stated it is "extremely likely" that the dominant cause of global warming has to do with human emissions and activities. With an increasing carbon dioxide emissions that gets trap inside the atmosphere, it increases the amount of longwave radiation that causes global warming. [3]. Figure 1.1.1 shows the time evolution of Radiative Forcing [1] of all human-caused gases and its rate of change. It is clear that carbon dioxide CO_2 is rising faster than all other gases and is causing major RF in the atmosphere.

As technology practitioners, it is crucial for us to fight against global warming and protect our environment. With the rapid development and breakthroughs in artificial intelligence and machine learning, there have been debates about whether it can positively affect climate change, or further contribute to more carbon emissions [4] [5] [16] [18]. Positively, we can use data mining

[1]RF is defined as the difference between the sunlight absorbed by the Earth and the energy radiated back to space

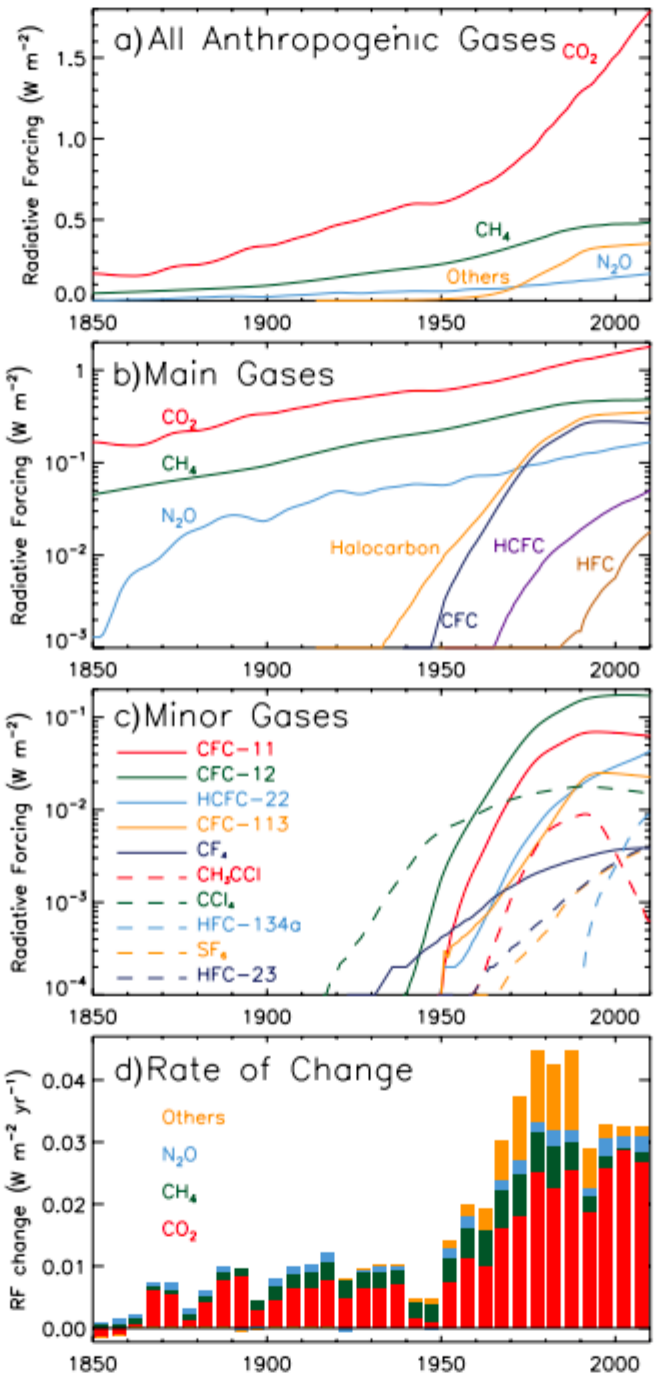


Figure 1.1.1. (a) Radiative Forcing from all anthropogenic gases from 1850 to 2011, (b) as (a) but with a logarithmic scale, (c) RF from minor gases, (d) Rate of change in forcing from major gases. Figure obtained from “Anthropogenic and Natural Radiative Forcing” [3].

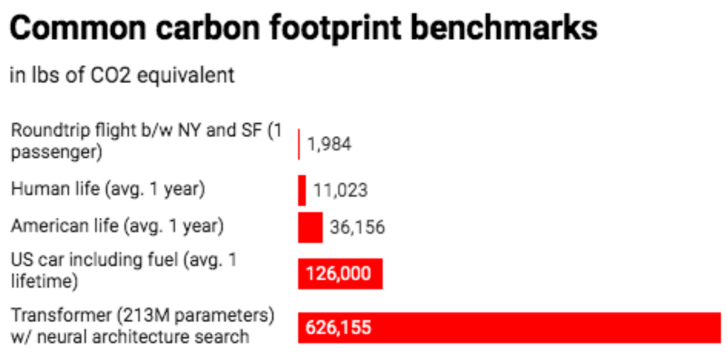


Figure 1.1.2. Common CO_2 emissions in pounds [6] [7].

to compress and analyze large sets of data to optimize energy consumption in many different fields. AI can also help with energy forecasting, energy management, renewable storage, and influencing sustainable development [4]. On the negative side, training different machine learning models are expensive and has terrible carbon footprint [6] [7]. Our focus will be on deep learning because deep learning has demonstrated its superior performance on a wide variety of tasks. However, with its effectiveness, it is also financially and computationally expensive. Unlike some classical ML algorithms, which can be trained just fine with a decent CPU, deep learning requires high-end GPUs to be trained in a reasonable amount of time with big data. Figure 1.1.2 shows how training a Transformers [2] can emit more than 626,000 pounds of carbon dioxide equivalent to five times the lifetime emissions of the average American car, including manufacture of the car itself [7].

This is obviously shocking to many of us, but it is a problem that hasn't been considered much in AI. The majority of research in deep learning still primarily focuses on obtaining high levels of accuracy and efficiency without any computation constraints or considering about its environmental effects. It's important to take those limitations into consideration because limitations in computational power will definitely slow down advancement of deep learning models [18]. Furthermore, the rapid growth of data does not help. Figure 1.1.3 shows that the

[2]Transformers are a type of neural network architecture developed to solve the problem neural machine translation (e.g. speech recognition, text-to-speech transformation).

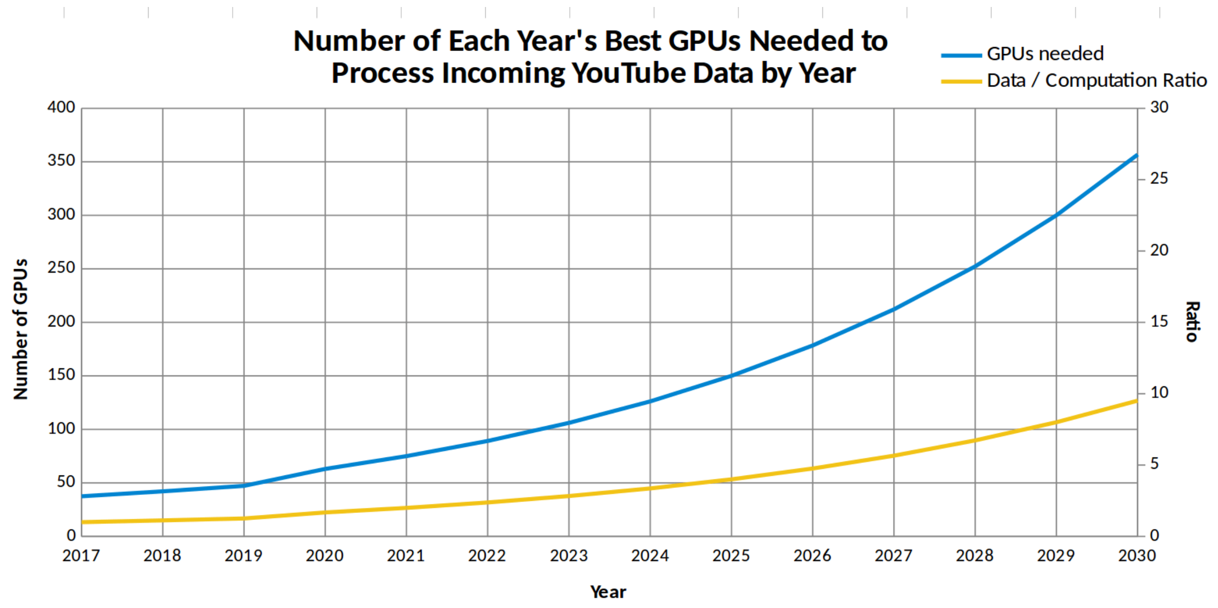


Figure 1.1.3. The number of each year's best GPUs needed to process all incoming YouTube data (ResNet frame by frame) along with the increasing gap between growth in data and computation. Computational data taken from [8].

growth of data is now faster than the growth of computing power as we can see that the gap between the number of GPUs needed and data/computation ratio is expanding [8].

1.2 Achievements

This paper will discuss some of current approaches in estimating energy consumption in machine learning, challenges of predicting energy consumption, and why is it important to do so in deep learning specifically. While two different convolutional neural networks, SimpleNet and AlexNet, were trained for 12 hours each on Intel® Core^{TM} i7-4770 on Cifar-10 dataset, energy consumption was collected every hour with powerstat. Concepts needed to understand the research will be introduced in the Background chapter, and the specific implementations used will be discussed in the Methods chapter. Accuracy for both networks weren't able to mimic the original papers due to different hardwares used and AlexNet was trained on cifar-10 instead of ImageNet. We were able to compare networks' energy consumptions and carbon emissions when they reached an accuracy of 60% and found that AlexNet uses more than 6 times as much energy and has more than 6 times as much carbon emissions as well.

2
Background

2.1 Power versus Energy

Many of us confuse two related, but different, physical quantities: energy and power. If someone says that a new farm will generate 250 megawatts per year, can you tell something is wrong? However, if someone says that they went on a diet and lost 15 horsepower, most of us would know that that is wrong. In both cases, although one is more familiar than the other, the units used were wrong. In physics, energy is the capacity to do work and it is measured in Joules. Power, on the other hand is the rate of doing work, which is equivalent to an amount of energy consumed per unit time and it is measured in Watts. The main difference between power and energy is that while energy measure the total quantity of work done, it doesn't say how fast you can get the work done. [9] There are several different units used to measure energy including joules, newton-meters, and even calories. When we're talking about electrical energy, the most common unit is watt-hour. The unit we'll be using is kilowatt-hours (kWh), which is simply a thousand of watt-hours.

2.2 Current methods of monitoring energy

There are different approaches to estimate power consumption on both software-level and hardware-level [16]. One of the most popular one is to obtain the activity factors of the computations via performance counters (PMCs), to then build the model using regression techniques. PMCs are a set of special-purpose registers in modern processors that count specific event types that are hardware related. Many differentiate their metrics into core and uncore, which is further explained in Section 3.4. They derive the value of power consumption by obtaining the power weights associated to each PMC using linear regression (2.2.1) where w_i is the weight associated to component i, AR_i is the activity ratio of the component i, and P_s represents the overall static power of all components. The advantages of using PMCs are that there is no extra overhead, it is available for different operating systems, and it can be used for both large or small amount of datasets.

$$P_t = \sum_{i=1}^{n_{component}} AR_i * w_i + P_s \tag{2.2.1}$$

Many other energy estimation models obtain the activity factors via simulation. Wattch, [17] for example, presented parametrized power models and used analytical dynamic power equations to estimate the power values, which is based on capacitance estimations [1]. The main advantage of using a simulation is that it gives extensive details regarding where exactly the energy is consumed on both hardware-level and at the instruction level. To estimate energy at the instruction-level, most approaches run a set of curated micro-benchmarks where each benchmark loops over a target instruction type, to be able to isolate the power of that specific instruction. Furthermore, we can also estimate energy at architecture-level in L2 cache, DRAM, etc. Table 2.2.1 shows some detailed information for each techniques.

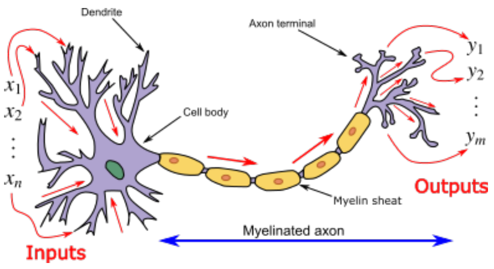
[1]The ratio of the change in an electric charge in a system to the corresponding in its electric potential.

Technique	**ML Application**	**Advantages**	**Disadvantage**
PMC	Energy consumption analysis of any ML model	No overhead. Application independent	No pre-processor results
Simulation	Analysis of algorithms behavior on ML specific hardware	Detailed results	Significant Overhead
Instruction-level	Energy consumption analysis of specific layers in a neural network	Detailed breakdown of energy consumption	Not easily available
Architecture-level	Improve programming hardware for ad-hoc ML applications	Detailed view	Usually not generalizable to different hardware platforms
Real-time	Streaming data and IoT	Easily available	Usually not detailed results

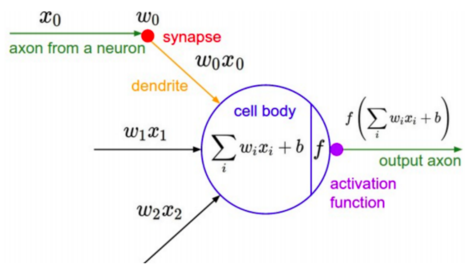
Table 2.2.1. Advantages, disadvantages, and ML application of techniques to estimate energy consumption discussed in section 2.2. [16].

2.3 Machine Learning

Machine Learning is the study how we program computers to learn, or to improve automatically with experience. More specifically, it means that we are taking some amount of past experience, or data, and with some algorithm, we want to do better next time without having to explicitly program it [10]. It is a subset of artificial intelligence, a widely-encompassing field attempts not just to understand but also to build intelligent entities [11]. When using machine learning algorithms, usually called ”training”, a sample of real-world data is feed into the model, which it then does analysis using algorithms to find patterns and adjust in order to make better predictions. Such techniques are used in a wide variety of applications. For example, in image recognition, a computer is given hundreds, or even millions of images that are labeled with correct names. Then, the algorithms adjust its weights which allows its trained model to eventually have the ability to recognize and classify images correctly. In Section 3.3, it further describes the image data used for this paper. The discipline of machine learning develops various approaches for computers to learn to accomplish tasks, one of which is deep learning.



(a) Neuron in the brain



(b) Artificial neuron

Figure 2.3.1. Comparison between (a) a neuron in the brain and (b) a mathematical model used in artificial neural networks [12].

2.3.1 Deep Learning

Deep learning is a subset of machine learning that involves the use of artificial neural networks (ANN) inspired by the structure and biological function of the human brain. ANN are computing systems based on a collection of connected nodes like the neurons in a biological brain. In the brain, neurons receive signals from their dendrites and send out signals along their axon that connects to the dendrites of other neurons with synapses [2]. Similarly, in an ANN, each connection can transmit a signal to other neurons except now the "signal" is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs, like the example shown in Figure 2.3.1 [12]. Each neuron has a certain potential to become activated depending on the strength of incoming signals from other neurons. For example, if the combined signals coming in is above a neuron's threshold, it spikes, and therefore, sending outputs to other neurons.

Both neurons and their connections have weights that adjust as learning proceeds. In the computation model of the neuron, each input signal has a signal strength , e.g. x_0, and a weight, w_0, that gets multiplied together, resulting in x_0w_0. The weighted input signals from all incoming neurons are then summed together using the function $\sum_i x_iw_i + b$ and if this sum is above the threshold, the neuron spikes, or activates. Similarly to the neurons in a biological brain, the synaptic strengths between neurons is constantly changing and influencing one an-

[2]In the nervous system, a synapse is a structure that permits a neuron to pass an electrical or chemical signal to another neuron or to the target effector cell.

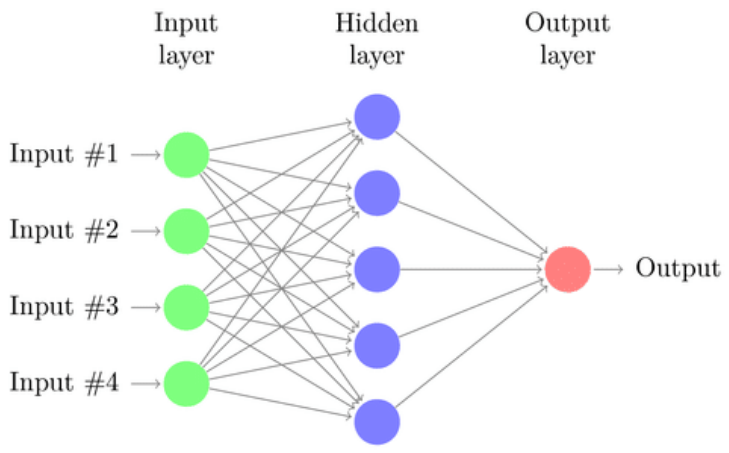


Figure 2.3.2. A two-layer neural network comprised of four inputs, one hidden layer, and one output layer. These layers are also called fully-connected layers because each layer they have full connections to all activations in the previous layer. Diagram from [https://blog.aimultiple.com/how-neural-networks-work/].

other. Typically, neurons are aggregated into layers and different layers may perform different transformations on their inputs like the one in Figure 2.3.2.

2.3.2 Convolutional Neural Network

One of the neural networks that we are going to explore explicitly and estimate their carbon footprint in this study are Convolutional Neural Networks (CNNs). CNNs are very similar to ordinary Neural Networks except for the assumption that the inputs are images. In particular, the layers of CNNs are in three dimensions instead of two, shown in Figure 2.3.3. Knowing so allows us to encode certain properties into the architecture and the purpose of this section is to summarize the function of essential layers in the architecture.

Imagine if you have an image pixels that is 32 wide and 32 high with 3 color channels, which gives us a 3D volumes of neurons $32 \times 32 \times 3$. This would mean that a single fully-connected neuron in a first hidden layer of a regular Neural Network would have $32 \times 32 \times 3 = 3072$ weights. However, if you have a larger image, e.g. 200×200, you will have 120,000 weights, which can get overwhelming. In an effort to solve this problem, instead of a full-connected layers, often the neurons in a layer will only be connected to a small region of the layer before it. Some of the layers which creates the architecture are convolution, fully-connected, pooling layer.

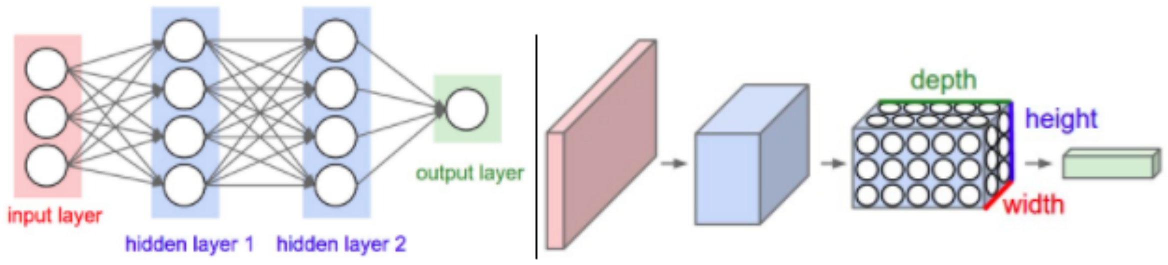


Figure 2.3.3. Left: A regular 3-layer Neural Network. Right: A CNNs arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers [12].

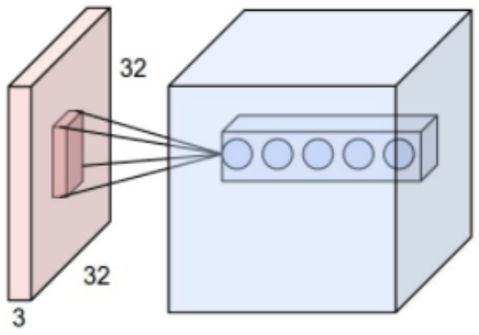


Figure 2.3.4. An example input volume in red and an example volume of neurons in the first Convolutional layer in blue. Note that each neuron in the convolutional layer is connected only to a local region in in input volume spatially [12].

Convolutional Layer

The Convolutional layer is the core building block of a CNN and its parameters consist of a set of learnable filters. For example, imagine that we have a filter with size $5 \times 5 \times 3$. During the forward pass, we slide this filter across the whole input volume of the image and with some computation, we will produce a 2 dimensional activation map that gives the responses of that filter. One filter may activate when it sees an edge, another may activate when it sees certain color. Ultimately, all the filters will stack the activation maps together and produce the output volume as shown in Figure 2.3.4.

Pooling Layer

Often, we insert a Pooling layer in between Convolutional layers to progressively reduce the spatial size of representation, the amount of parameters and computations, and to control over-

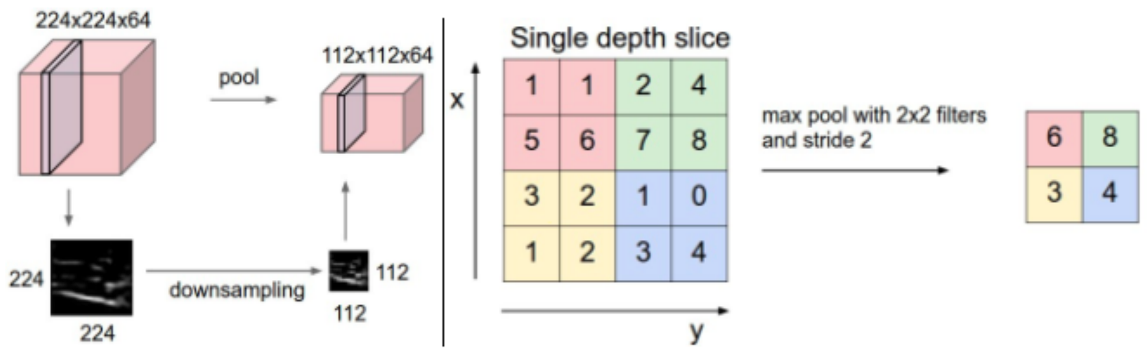


Figure 2.3.5. On the left shows how pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. On the right shows the most common downsampling operation, MAX pooling, that is each max is taken over 4 numbers with 2×2 filters [12].

fitting. It resizes it spatially using the MAX operation, which is essentially taking the max value from their spatial extent that we're looking at, shown in Figure 2.3.5.

Fully-connected Layer

In a fully-connected layer, each input neurons are all connected to each output neurons. How fully connected layer works is that it decides what high level features are most strongly correlate to a particular class by performing matrix multiplication. At the end, a FC will take the output of convolutional and pooling layers and predicts the best label to describe the image. For example, if we're trying to predict numbers from 1 to 10, the last fully connected layer may have 10 nodes each with a percentage of how likely it is to spike. The CNN process is simplified and shown in Figure 2.3.6.

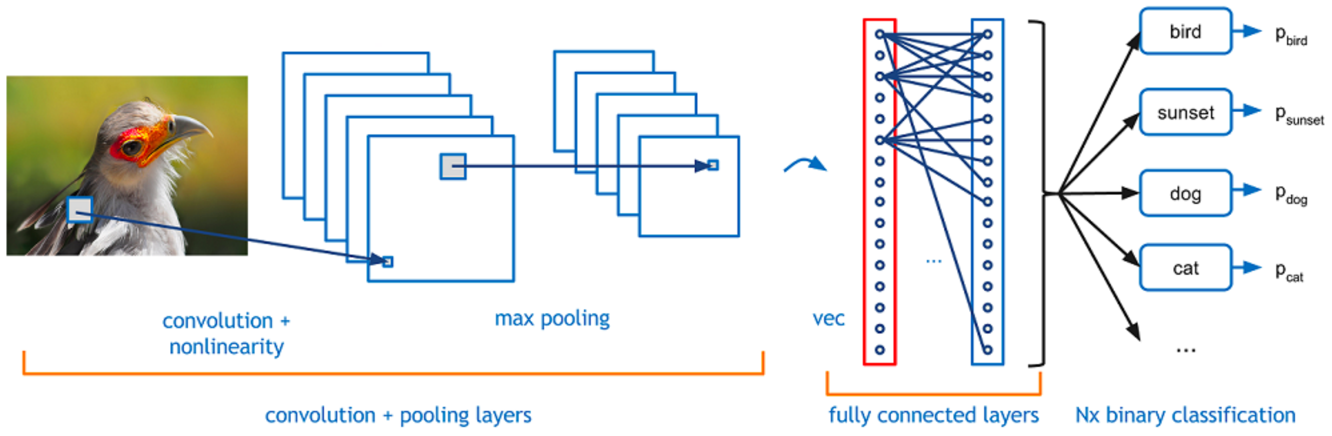
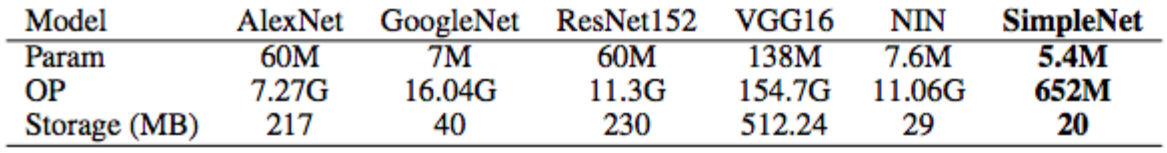


Figure 2.3.6. An simple illustration of how a CNN works from convolutional and pooling to fully connected. Image from https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

3
Methods

This section discusses the networks and tools used to estimate energy consumption of CNNs and to compare their carbon footprints. The selected CNN, SimpleNet, has a simple architecture with limited parameters, operations, and required storage. Table 3.0.1 shows different architectures statistics for comparing the amount of parameters, operations, and storage summarized from the original paper [21]. We'll be comparing the carbon footprint of training SimpleNet to AlexNet by estimating their energy consumption using RAPL and then convert it to carbon dioxide. We chose those two CNNs because AlexNet has more than 10 times of parameters and needs about 10 times more of storage as SimpleNet. Note that in this paper AlexNet will be trained on cifar-10 dataset instead of ImageNet due to lack of hardware requirements.

Table 3.0.1. Different architectures statistics [21].



Model	AlexNet	GoogleNet	ResNet152	VGG16	NIN	**SimpleNet**
Param	60M	7M	60M	138M	7.6M	**5.4M**
OP	7.27G	16.04G	11.3G	154.7G	11.06G	**652M**
Storage (MB)	217	40	230	512.24	29	**20**

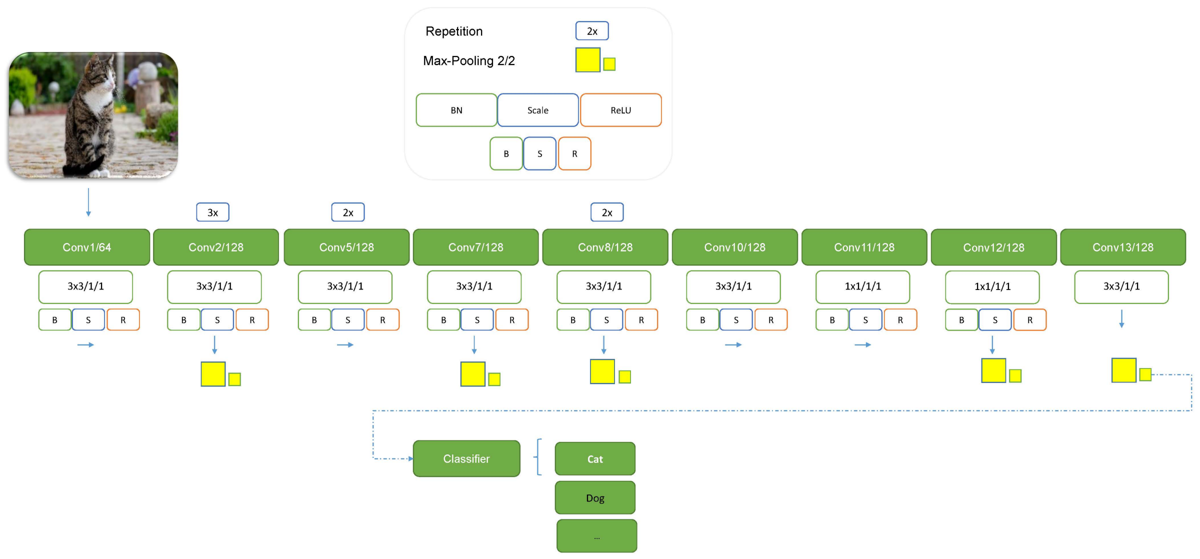


Figure 3.1.1. The base architecture with no drop-out [21].

3.1 SimpleNet

What many popular CNN architectures have in common is the increasing depth and complexity of the network in order to provide better accuracy in completing tasks like image recognition. However, increasing depth and complexity of the network can cause critical issues like high computation and memory usage cost and overhead. Those issues limit their practical use for training, optimization and memory efficiency. On the contrary, many light-weight architectures suffer from low accuracy, which is not ideal as well. SimpleNet [21] proposed a simple architecture that shows with careful design and minimum reliance on new features, it can outperforms almost all deeper architectures with 2 to 25 times fewer parameters. They intentionally imposed some limitation when designing to show the effectiveness of a well-crafted yet simple convolutional architecture.

The network has 13 layers with a homogeneous design utilizing 3×3 kernels for convolutional layers and 2×2 kernels for pooling operations. Figure 3.1.1 illustrates the proposed architecture. The only layers which do not use 3×3 kernels are 11th and 12th layers, and instead they use 1×1 convolutional kernels. To prevent the problem of vanishing gradient and over-fitting, they used batch-normalization with moving average fraction of 0.95 before any ReLU non-linearity.

They also used weight decay as regularizer. The original network was implemented in Caffe, which is a deep learning framework where one writes a configuration file in prototxt to tell Caffe how you want the network trained. Listing 3.1 shows an example of how a convolutional layer looks in prototxt with 3×3 kernel size with batch-normalization, scaling, and ReLu.

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
  convolution_param {
    num_output: 64
    pad: 1
    kernel_size: 3
    stride: 1
    bias_term: true
    weight_filler {
      type: "xavier"
    }
  }
}

layer {
  name: "bn1"
  type: "BatchNorm"
  bottom: "conv1"
  top: "conv1"
  param {
    lr_mult: 0
   decay_mult: 0
  }
  param {
    lr_mult: 0
   decay_mult: 0
  }
  param {
    lr_mult: 0
   decay_mult: 0
  }
    include {
    phase: TRAIN
  }
    batch_norm_param {
    use_global_stats: false
    moving_average_fraction: 0.95
  }
}
layer {
  name: "bn1"
  type: "BatchNorm"
  bottom: "conv1"
  top: "conv1"
  param {
    lr_mult: 0
   decay_mult: 0
  }
  param {
    lr_mult: 0
```

```
                        decay_mult: 0
                      }
                      param {
                        lr_mult: 0
                       decay_mult: 0
                      }
                        include {
                        phase: TEST
                      }
                        batch_norm_param {
                        use_global_stats: true
                        moving_average_fraction: 0.95
                      }
                    }
                    layer {
                      name: "scale1"
                      type: "Scale"
                      bottom: "conv1"
                      top: "conv1"
                      scale_param {
                        bias_term: true
                      }
                    }

                    layer {
                      name: "relu1"
                      type: "ReLU"
                      bottom: "conv1"
                      top: "conv1"
                    }
```

Listing 3.1. SimpleNet Caffe example

All of above designs were decided using several principles which helped manage different problems and achieve better results. First is gradual expansion and minimum allocation. In order to better manage the computational overhead, parameter utilization efficiency, they started with a small and thin network and then gradually expand it. This also decreases the chance of over fitting with fewer learnable parameters. Next is the design of homogeneous groups. Instead of thinking in layers, the symmetric and homogeneous design allows to easily manage the number of parameters a network withholds and also provide better information pools for each semantic level. The network tried to preserve locality information throughout the network as much as possible by using limited $1x1$ kernels in early layer. Another important factor is the effort of maximizing information utility. It is made available to a network by avoiding rapid down sampling especially in each layers. This is important because we want to keep as much information as possible to increase a network's discriminative power. Furthermore, in order to perform faster and decently whenever possible, using $3x3$ kernels have shown results in 2.7x faster training when using cuDNNx5 library. They also tried to test the architecture with different learning policies before altering it. Simple things such as learning rates and regularization methods can be greatly affected if it's not tuned correctly. Therefore, they used an automated optimization policy and then carefully tuned to maximize network performance. Throughout their experience and adjustments in making the architecture, they made sure of experiment isolation and minimum entropy at all times. Instead of the original implementation in Caffe, the implementation that was used to estimate power consumption with was Pytorch because I had more experience with Python in the past, it is also simpler to read and understand with the entire network shown in Listing 3.2.

```
model = nn.Sequential(
  nn.Conv2d(3, 64, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(64, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),

  nn.Conv2d(64, 128, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(128, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),

  nn.Conv2d(128, 128, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(128, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),constraint

  nn.Conv2d(128, 128, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(128, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),


  nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1),
  ceil_mode=False),
  nn.Dropout2d(p=0.1),


  nn.Conv2d(128, 128, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(128, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),

  nn.Conv2d(128, 128, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(128, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),

  nn.Conv2d(128, 256, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(256, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),


  nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1),
  ceil_mode=False),
  nn.Dropout2d(p=0.1),


  nn.Conv2d(256, 256, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(256, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),


  nn.Conv2d(256, 256, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(256, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),


  nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1),
  ceil_mode=False),
  nn.Dropout2d(p=0.1),


  nn.Conv2d(256, 512, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(512, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),


  nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1),
  ceil_mode=False),
  nn.Dropout2d(p=0.1),


  nn.Conv2d(512, 2048, kernel_size=[1, 1], stride=(1, 1), padding=(0, 0)),
  nn.BatchNorm2d(2048, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),


  nn.Conv2d(2048, 256, kernel_size=[1, 1], stride=(1, 1), padding=(0, 0)),
  nn.BatchNorm2d(256, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),


  nn.MaxPool2d(kernel_size=(2, 2), stride=(2, 2), dilation=(1, 1),
  ceil_mode=False),
  nn.Dropout2d(p=0.1),


  nn.Conv2d(256, 256, kernel_size=[3, 3], stride=(1, 1), padding=(1, 1)),
  nn.BatchNorm2d(256, eps=1e-05, momentum=0.05, affine=True),
  nn.ReLU(inplace=True),
 )


```

Listing 3.2. SimpleNet Pytorch example

3.2 AlexNet

AlexNet is a large and deep convolutional neural network designed by Alex Krizhevsky [13]. In 2012, the network achieved a top-6 error of 15.3% in the ImageNet Large Scale Visual Recognition Challenge. It learned to classify 1.2 million high-resolution images into 1000 different classes. The neural network has 60 million parameters and 650,000 neurons with five convolutional layers, some of them followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax like shown in Figure 3.2.1 and Listing 3.3 shows how the model was built with

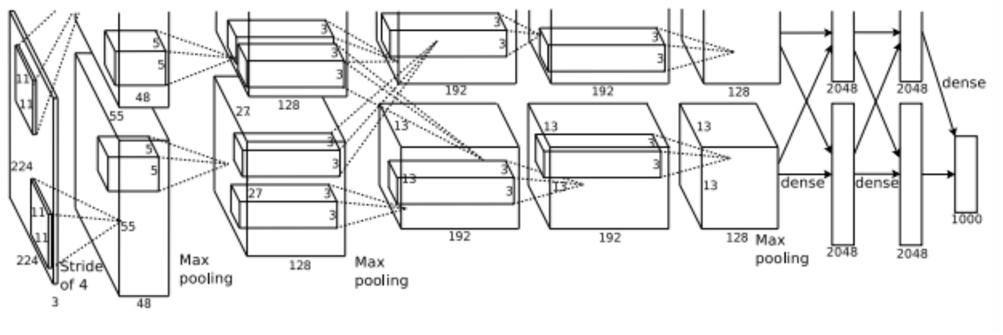


Figure 3.2.1. An illutration of the architecture of AlexNet, explicitly showing how two GPUs work together. One GPU runs the layer part at the top while the other runs at the bottom [13].

tensorflow. From the original paper, they showed that the depth of the model was essential for its high performance, which was computationally expensive and only feasible with the utilization of GPUs during training. AlexNet is considered one of the most influential papers published in computer vision and have inspired many more papers employing CNNs and GPUs to accelerate deep learning.

Ideally, we would want to compare the difference in the carbon emissions between a simple network like SimpleNet and a large network like AlexNet. However, due to many limitations, we weren't able to do that. The first reason being the limited amount of memory available to process all 15 million images. In this experiment, we will collecting the carbon footprint of a much smaller version of AlexNet. Although with the same architecture, instead of 15 million images of ImageNet, we'll be using Cifar-10, which only consist 60,000 images. Krizhevsky's original paper trained the network on 2 highly-optimized GTX 580 3GB GPUs. However, we'll be training on Intel® Core^{TM} i7-4770. It's important to notice that the energy consumption will be very different from the original paper due to different hardware.

```

import tensorflow as tf

n_classes = 10
image_size = 32
dropout = tf.placeholder(tf.float32, name="dropout_rate")
input_images = tf.placeholder(tf.float32, shape=[None, image_size, image_size, 3],
         name="input_images")

#Network Size
first_conv_size = 96
second_conv_size = 256
third_conv_size = 384
```

```
fourth_conv_size = 384
fifth_conv__size = 256

# First CONV layer
kernel = tf.Variable(tf.truncated_normal([11, 11, 3, 96],
              dtype=tf.float32,  stddev=1e-1),
              name="conv1_weights")
conv = tf.nn.conv2d(input_images, kernel, [1, 4, 4, 1], padding="SAME")
bias = tf.Variable(tf.truncated_normal([96]))
conv_with_bias = tf.nn.bias_add(conv, bias)
conv1 = tf.nn.relu(conv_with_bias, name="conv1")

lrn1 = tf.nn.lrn(conv1, alpha=1e-4,
  beta=0.75, depth_radius=2, bias=2.0)

pooled_conv1 = tf.nn.max_pool(lrn1,
  ksize=[1, 3, 3, 1],
          strides=[1, 2, 2, 1],
  padding="SAME",
         name="pool1")

# Second CONV Layer
kernel = tf.Variable(tf.truncated_normal([5, 5, 96, 256],
                                         dtype=tf.float32,
                                         stddev=1e-1),
                               name="conv2_weights")
conv = tf.nn.conv2d(pooled_conv1, kernel, [1, 4, 4, 1], padding="SAME")
bias = tf.Variable(tf.truncated_normal([256]), name="conv2_bias")
conv_with_bias = tf.nn.bias_add(conv, bias)
conv2 = tf.nn.relu(conv_with_bias, name="conv2")
lrn2 = tf.nn.lrn(conv2,
                 alpha=1e-4,
                 beta=0.75,
                 depth_radius=2,
                 bias=2.0)

pooled_conv2 = tf.nn.max_pool(lrn2,
                              ksize=[1, 3, 3, 1],
                              strides=[1, 2, 2, 1],
                              padding="SAME",
                              name="pool2")

# Third CONV layer
kernel = tf.Variable(tf.truncated_normal([3, 3, 256, 384],
                                         dtype=tf.float32,
                                         stddev=1e-1),
                     name="conv3_weights")
conv = tf.nn.conv2d(pooled_conv2, kernel, [1, 1, 1, 1], padding="SAME")
bias = tf.Variable(tf.truncated_normal([384]), name="conv3_bias")
conv_with_bias = tf.nn.bias_add(conv, bias)
conv3 = tf.nn.relu(conv_with_bias, name="conv3")

# Fourth CONV layer
kernel = tf.Variable(tf.truncated_normal([3, 3, 384, 384],
                                         dtype=tf.float32,
                                         stddev=1e-1),
                     name="conv4_weights")
conv = tf.nn.conv2d(conv3, kernel, [1, 1, 1, 1], padding="SAME")
bias = tf.Variable(tf.truncated_normal([384]), name="conv4_bias")
conv_with_bias = tf.nn.bias_add(conv, bias)
conv4 = tf.nn.relu(conv_with_bias, name="conv4")

# Fifth CONV Layer
kernel = tf.Variable(tf.truncated_normal([3, 3, 384, 256],
                                          dtype=tf.float32,
                                          stddev=1e-1),
                      name="conv5_weights")
conv = tf.nn.conv2d(conv4, kernel, [1, 1, 1, 1], padding="SAME")
bias = tf.Variable(tf.truncated_normal([256]), name="conv5_bias")
conv_with_bias = tf.nn.bias_add(conv, bias)
conv5 = tf.nn.relu(conv_with_bias, name="conv5")

# Fully Connected Layers
fc_size = 256
conv5 = tf.layers.flatten(conv5) # tf.flatten
weights = tf.Variable(tf.truncated_normal([fc_size, fc_size]), name="fc1_weights")
bias = tf.Variable(tf.truncated_normal([fc_size]), name="fc1_bias")
fc1 = tf.matmul(conv5, weights) + bias
fc1 = tf.nn.relu(fc1, name="fc1")
fc1 = tf.nn.dropout(fc1, dropout)

weights = tf.Variable(tf.truncated_normal([fc_size, fc_size]), name="fc2_weights")
bias = tf.Variable(tf.truncated_normal([fc_size]), name="fc2_bias")
fc2 = tf.matmul(fc1, weights) + bias
fc2 = tf.nn.relu(fc2, name="fc2")
fc2 = tf.nn.dropout(fc2, dropout)

weights = tf.Variable(tf.zeros([fc_size, n_classes]), name="output_weight")
bias = tf.Variable(tf.truncated_normal([n_classes]), name="output_bias")
out = tf.matmul(fc2, weights) + bias
```

Listing 3.3. AlexNet model.py

3.3 Cifar10

The input dataset used to feed into our networks is CIFAR-10. This dataset consists of 60,000 color images of which 50,000 belong to training set and 10,000 are reserved for testing. There are a total of 10 classes each with 6000 images per class, like shown in Figure 3.3.1. The dataset is divided into five training batches and one test batch. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but the number of images from each class are not the same [14].

3.4 RAPL

The best way to measure energy and power is with ammeter, however, they are more difficult to set up and can be more expensive. On the software level, recent Intel processors (Sandy Bridge microarchitecture and later) that implement the RAPL interface provides MSRs [1] containing

[1]A model-specific register is any of various control registers in the x86 instruction set used for debugging, program execution tracing, computer performance monitoring, and toggling certain CPU features.

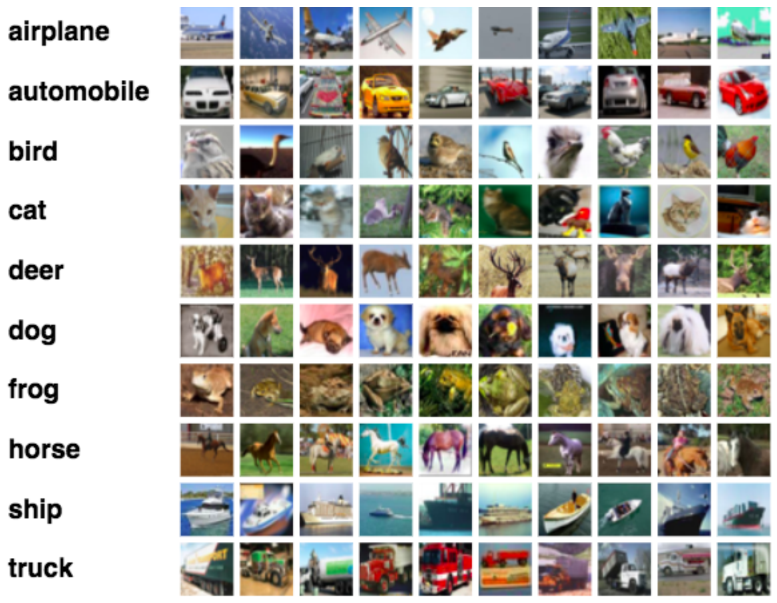


Figure 3.3.1. The classes in the dataset, as well as 10 random images from each [14].

energy consumption estimates for up to four power planes or domains of a machine, as seen in the Figure 3.4.1, which shows how machines using recent intel processors are constructed. Intel's RAPL (Running Average Power Limit) is a software power tool that estimates power consumption reading of the core, uncore, and DRAM. RAPL provides the energy and power readings by using hardware performance counters and I/O models [16] [19]. The important points are as follows:

- The processor has one or more packages.
- Each package contains multiple cores.
- Each core typically has hyper-threading, which means it contains two logical CPUs.
- The part of the package outside the cores is called the uncore or system agent. It includes various components including the L3 cache, memory controller, and, for processors that have one, the integrated GPU.
- RAM is separate from the processor.
- PKG: The entire package

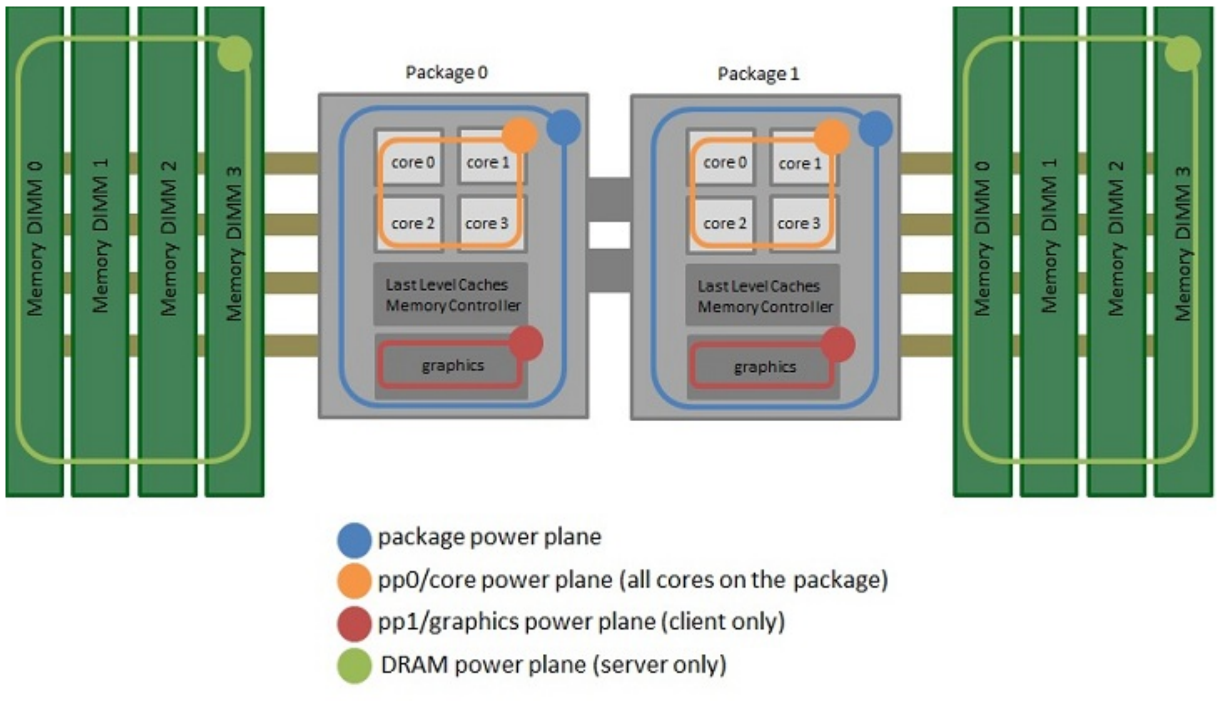


Figure 3.4.1. power-planes [19].

  – PP0: The cores

  – PP1: An uncore device, usually the GPU

- DRAM: main memory

The following relationship holds: $((PKG \implies (PP0 + PP1))$. DRAM is independent of the other three domains.

These values are computed using a power model that takes processor-internal counts as inputs, and they have been verified as being fairly accurate. Reported errors claim that RAPL gives results within 2.3% of actual measurements for the DRAM; and that RAPL slightly underestimates the power for some workloads [20].

The tool used to estimate power consumption is powerstat [15]. Powerstat measures the power consumption of a computer that supports the RAPL interface. At the end of a run, powerstat calculates the average, standard deviation and min/max of the gathered data. To eliminate noise and get better accuracy, I made sure that while training is happening, nothing else is running on

```
Running for 60.0 seconds (60 samples at 1.0 second intervals).
Power measurements will start in 0 seconds time.

  Time     User  Nice   Sys  Idle    IO  Run Ctxt/s  IRQ/s  Watts    dram  pkg-0   core
02:01:02  43.2   0.0   5.7  51.1   0.0    6    934  53222  46.67    2.87  46.67  41.12
02:01:03  44.8   0.0   4.5  50.6   0.0    5    806 134201  45.12    3.06  45.12  39.53
02:01:04  44.2   0.0   4.6  51.3   0.0    5    819   5739  47.11    2.77  47.11  41.68
02:01:05  48.2   0.0   1.5  50.3   0.0    5    748   3109  44.91    3.42  44.91  39.25
02:01:06  45.3   0.0   3.5  51.1   0.1    5    536   1694  47.94    2.71  47.94  42.56
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------  ------ ------ ------
 Average  45.8   0.0   3.1  51.1   0.0  5.0  541.7 5053.1  46.36    3.04  46.36  40.86
 GeoMean  45.8   0.0   2.7  51.1   0.0  5.0  531.7 2032.6  46.36    3.04  46.36  40.85
  StdDev   1.7   0.0   1.3   0.5   0.1  0.5  110.2 18099.8   0.81    0.19   0.81   0.86
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------  ------ ------ ------
 Minimum  42.8   0.0   0.8  50.1   0.0  2.0  376.0 1347.0  44.58    2.71  44.58  39.00
 Maximum  49.1   0.0   5.7  52.1   0.6  6.0  934.0 134201.0  48.02    3.42  48.02  42.61
-------- ----- ----- ----- ----- ----- ---- ------ ------ ------  ------ ------ ------
Summary:
CPU:  46.36 Watts on average with standard deviation 0.81
Note: power read from RAPL domains: dram, package-0, core.
```

Listing 3.4. An example of an output from powerstat -RDH collected while training SimpleNet

the computer. Data were collected at every hour of the training for 12 hours. Listing 3.4 shows a portion of an example of an output data collected from powerstat while training SimpleNet.

3.5 Power to Carbon Dioxide

The estimated total time expected for models to train to completion was taken from total epochs, average epoch time and current epoch. And the number of epochs used were from the original papers.

$$TotalTimeLeft = epoch_{avg} * (epoch_{total} - epoch_{current}) \tag{3.5.1}$$

Then, with the power consumption collected from powerstat, we estimate total power consumption in kilowatt-hours [7]. Let p_c be the average power draw (in watts) from all CPU sockets, or PKG-0 during training, let p_r be the average power draw from all DRAM sockets. We then combine the two and multiply this by Power Usage Effectiveness (PUE), which takes into consideration for any additional energy used such as cooling. The PUE value used is 1.58, which is the 2018 global average for data centers [22]. Therefore, the total power p_t required at a given instance during training is given by:

$$p_t = \frac{1.58t(p_c + p_r)}{1000} \tag{3.5.2}$$

The U.S. Environmental Protection Agency (EPA) provides average CO_2 produced (in pounds per kilowatt-hour) for power consumed in the U.S. (EPA, 2018), which is used here to convert power to estimated CO_2 emissions:

$$\mathrm{CO_2} = 0.954 p_t \tag{3.5.3}$$

4
Results

This section presents the key results and analysis of estimating energy consumption and carbon footprint on machine learning algorithms. Two CNNs, SimpleNet and AlexNet, were trained for 12 hours on Intel® Core^{TM} i7-4770 CPU on Cifar-10 dataset. The section is divided as follow: the accuracy achieved in 12 hours, the amount of energy used, and an analysis on the outputs.

SimpleNet was able to achieve an accuracy of 89.99% in 12 hours, while AlexNet achieved an accuracy of 59.8%, shown in Figure 4.0.1. In this paper, we weren't able to reproduce AlexNet from the original paper because of lack of GPUs to process the original dataset they used, ImageNet. In addition, AlexNet was trained for 6 to 7 days from the original paper, however, we only trained it for 12 hours. As a result, the training outcome is different. SimpleNet, on the other hand, put hardware constraints into their research to prove that simple networks with detailed designs can also achieve similar accuracy.

While each network was training, power consumption datas were collected at every hour of the 12 hours. The average PKG for SimpleNet is 46.73 Watts and DRAM is 3.06 Watts. To get the total power we simply do PKG + DRAM, which gives us an average of 49.79 W. The average PKG for AlexNet is 25.86 W and DRAM is 1.91W, which the total average is 27.77W. Figure 4.0.2 shows the total energy used at each hour. To get how much kJ in an hour, $1W = 1J/s$, so $27.77W = 27.77J/s$, $27.77J * 3600s/1000 = 99.9kJ$.

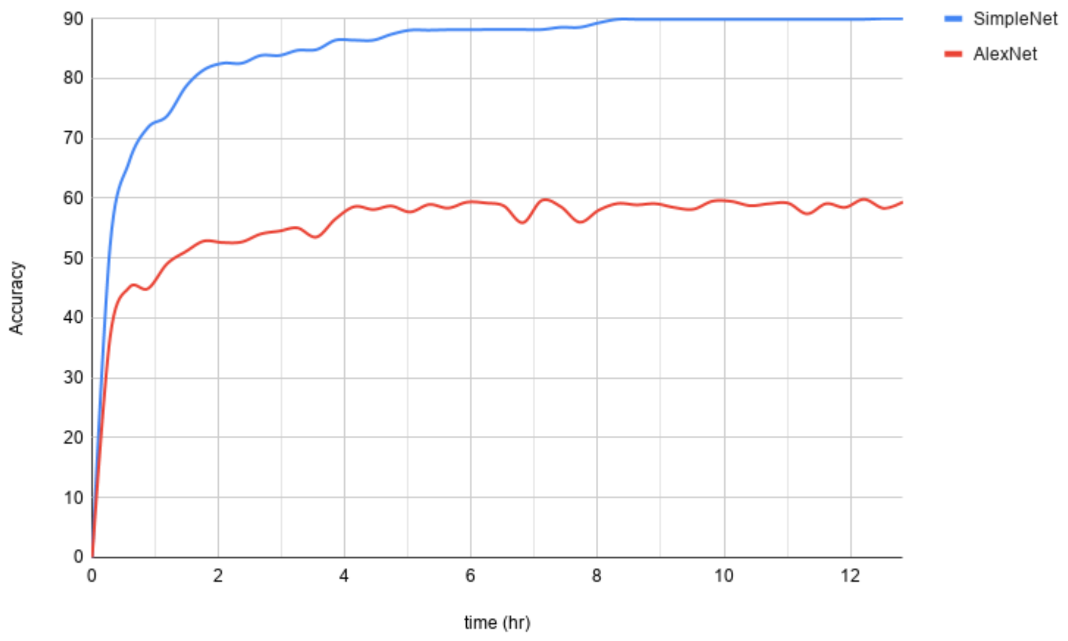


Figure 4.0.1. Accuracy achieved by SimpleNet and AlexNet with 12 hours of training.

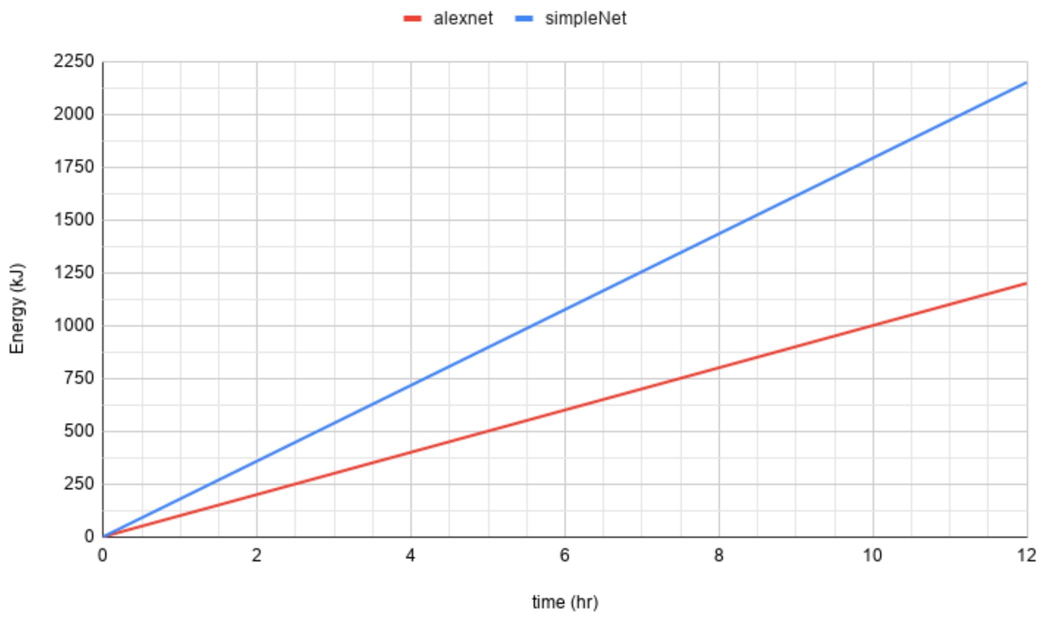


Figure 4.0.2. Amount of energy used in kJ.

Model	**Power (W)**	**Hours to achieve 60% accuracy**	**kWh-PUE**	CO_2 **(lbs)**
SimpleNet	49.49	0.5	0.039	0.037
AlexNet	27.77	6	.263	0.25

Table 4.0.1.

Although it seems like SimpleNet is using more energy, to have a fair comparison, we can look at when they achieved the same accuracy. When AlexNet reaches about 60% accuracy, it's at hour 6. However, SimpleNet has already reached 60% accuracy at half an hour. This shows that at the same accuracy, AlexNet used about 600 kJ and simpleNet used about 90 kJ. Their total power used in kWh-PUE and carbon emission are listed in Table 4.0.1. The above comparison is important for one to consider the trade-offs whether or not the expensive is worthy for the accuracy it can achieve.

5 Conclusion

This project focuses on the importance of making deep learning research energy-efficient in order to lower human-caused carbon emissions, decrease expensive, and improve our environment. The method used here is by estimating energy consumptions while training a network and compare its costs and carbon emissions. In this project, we estimated the energy consumption of two very different neural networks, SimpleNet and AlexNet. SimpleNet has a simple architecture with a homogeneous design, while AlexNet uses a complex architecture with millions of parameters. The output of their energy consumptions tells us that it is possible to limit the amount of data and GPUs required, and put constraints on the research, and still be creative and have great accuracy from the networks.

It is my hope that data-centers and AI researches can eventually all be carbon-free, that we can have significant impact on this world without destroying our earth. Although more efforts are put into green computing now, much more research need to be done from how to estimate energy before training, how to switch to green-computing all over the world, to how to make data centers carbon-free. The method used in this project can be use for someone who needs to compare different ways to solve a problem looking at accuracy and their energy consumption costs. The next step to this would be to predict energy consumption without having a trial run or to train the network. If we're able to do so, we can compare costs and solve problems with the least computation and environmental costs choice. It can also be very useful for investors to have a visualization to decide where the money is worth. Data visualization has proven to communicate relationships of the data more easily with visual trends and patterns. It is unfortunate the results of this project wasn't able to have as much significance as I wanted to be. However, I hope that this research can help us realized the importance of taking actions now towards carbon-free.

Bibliography

[1] Thomas Kostigen, *Hacking Planet Earth: How Geo-engineering Can Help Us Reimagine the Future*, Penguin Random House LLC, USA, 2020.

[2] IPCC, *Summary for Policymakers* (T.F. and Qin Stocker D. and Plattner, ed.), Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA, 2013.

[3] G. and Shindell Myhre D. and Bréon, *Anthropogenic and natural radiative forcing* (T. F. and Qin Stocker D. and Plattner, ed.), Cambridge University Press, Cambridge, UK, 2013.

[4] Vivek Kumar, *How Artificial Intelligence Can Increase Energy Efficiency*, Blue and Green Tomorrow (Jun 1, 2018).

[5] Jui-Sheng Chou and Dac-Khuong Bui, *Modeling heating and cooling loads by artificial intelligence for energy-efficient building design*, Energy and Buildings **82** (2014), 437-446, DOI 10.1016/j.enbuild.2014.07.036.

[6] Karen Hao, *Training a single AI model can emit as much carbon as five cars in their lifetimes*, MIT Technology Review **Artificial Intelligence** (Jun 6, 2019).

[7] Emma Strubell and Ananya Ganesh and Andrew McCallum, *Energy and Policy Considerations for Deep Learning in NLP* (2019), available at `1906.02243`.

[8] John Murphy, *Deep Learning Benchmarks of NVIDIA Tesla P100 PCle, Tesla K80, and Tesla M40 GPUs*, Microway (Jan 27, 2017).

[9] Rob Lewis, *The Great "Power vs. Energy" Confusion*, Clean Technica (Feb 2, 2015).

[10] Tom M and others Mitchell, *Machine Learning*, McGraw-hill New York, 1997.

[11] Stuart and Norvig Russell Peter, *Artificial Intelligence: A Modern Approach*, 3rd, Prentice Hall Press, USA, 2009.

[12] Standford University, *CS231n: Convolutional Neural Networks for Visual Recognition.*, 2018.

[13] Ilya and Hinton Alex Krizhevsky and Sutskever Geoffrey E, *ImageNet Classification with Deep Convolutional Neural Networks*, Advances in Neural Information Processing Systems 25, 2012, pp. 1097–1105.

[14] Alex Krizhevsky and Geoffrey Hinton, *Learning multiple layers of features from tiny images* **1** (4/8/2009), 7.

[15] *powerstat - Ubuntu manuals.*

[16] Eva García-Martín and Crefeda Faviola Rodrigues and Graham Riley and Håkan Grahn, *Estimation of energy consumption in machine learning*, Journal of Parallel and Distributed Computing **134** (2019), 75 - 88, DOI https://doi.org/10.1016/j.jpdc.2019.07.007.

[17] David and Tiwari Brooks Vivek and Martonosi, *Wattch: A framework for architectural-level power analysis and optimizations*, ACM SIGARCH Computer Architecture News **28** (2000), no. 2, 83–94.

[18] Tim Dettmers, *Deep Learning Research Directions: Computational Efficiency*, Tim Dettmers: Making Deep Learning accessible (2017).

[19] *Power Profiling Overview*, MDN Web Docs.

[20] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver, *A Validation of DRAM RAPL Power Measurements*, Proceedings of the Second International Symposium on Memory Systems, 2016, pp. 455–470, DOI 10.1145/2989081.2989088.

[21] Seyyed Hossein HasanPour and Mohammad Rouhani and Mohsen Fayyaz and Mohammad Sabokrou, *Lets keep it simple, Using simple architectures to outperform deeper and more complex architectures*, CoRR **abs/1608.06037** (2016), available at `1608.06037`.

[22] Rhonda Asierto, *Uptime Institute Global Data Center Survey*, Technical report, Uptime Institute. (2018).