Approximately Intelligent: Planning for an Uncertain Future in Computing

by

June Knauth

A Thesis submitted to the Faculty

in partial fulfillment of the requirements for the

Bachelor of Arts

Accepted

Eric Kramer, Thesis Advisor

Zachary While, Second Reader

John B. Weinstein, Provost and Vice President

Bard College at Simon's Rock

Great Barrington, Massachusetts

2023

Dedicated to those who refuse to see the way they're asked to.

Acknowledgements

This thesis was the product of more people, communities, and ideas than I can count. Here are a few of them.

First, the people in my life who have kept me afloat this year. I would like to thank Nur Goldner, who has given me more than I can say and remains the only person I can't not talk to. Lee Heintzelman, who has been there always, and whom I've never stopped clicking with (*thanks bips*). Nora, Addie, Patience, and Maya, who have added wonderful flavor to this very long process. Vi, who has given me so many stories. Wilder, who's quick on the bit. Piper Kilgore, who completes the set. And my family, Tonya, Rick, Matthew, and Katherine, who have made me the woman I am.

The people and professors at Simon's Rock and elsewhere who have tirelessly worked to support and educate me over four long years. Eric Kramer, who has been the best advisor I could have asked for and always been honest. Zack While, who has gone above and beyond his job description. John Hawley, who conducts a hell of an interview. Dan Neilsen, who succeeded in convincing me not to go into Economics. Maryann Tebben, who challenged me at unexpected times. And KellyAnne McGuire, who has been an incredible advocate for everyone thesising this year, and whose snacks have kept us all going.

The Noisebridgers who have provided both inspiration and technical knowledge- Loren McIntrye, Sophia Wisdom, and TJ.

The communities and spaces who have brought together so many incredible people, and those who enable and populate them- freeradical, tek, pixel, the Super Dimension Fortress, and all the Fedizens and damgudcyberchatters.

Contents

Abstract

Artificial Intelligence (AI), a field which is currently undergoing a massive technological and financial boom, has created possibilities in computing which are still being explored. However, it requires significant amounts of computing (and thus electrical) power. Few players in the AI space are considering the environmental effects of AI training or whether the technology can remain practical as electricity grows scarcer due to resource depletion.

In this thesis, I explain in simple terms what AI is, how it came to be, and why it demands so many resources. From there, I explore Approximate Multipliers (AMs) a new technique which can drastically reduce the energy usage of a neural network — the computational backbone of modern AI. Most previous investigations into AMs did not consider their use during training, which accounts for the majority of energy consumed by a neural network.

Using ApproxTrain, a framework which allows implementation and testing of AMs within the AI development library TensorFlow, I reimplement the AlexNet network, a common choice for testing new AI technologies. I test performance in training and inference on the CIFAR-10 dataset, which contains 60,000 images in 10 categories for classification. I show that approximate multipliers do not reduce classification accuracy, even when used during training.

Common Acronyms

AGI Artificial General Intelligence

AI Artificial Intelligence

AM Approximate Multiplier

CPU Central Processing Unit

DNN Deep Neural Network

FLOPs Floating Point Operations per Second

FP Floating Point

GPU Graphics Processing Unit

MBM Minimally Biased Multiplier

ML Machine Learning

NN Neural Network

Chapter 1

Introduction

In today's computing landscape, Artificial Intelligence (AI) is the ultimate buzzword. Deep Neural Networks (DNNs) have solved problems previously believed to be unsolvable, astonishing experts and laymen alike. The possibilities afforded by recent advances in the technology seem endless; state-of-the-art models such as GPT-4 can produce high-quality output difficult to distinguish from human writing, score far above the average on most standardized tests, and use tools to interact with their environments [49]. OpenAI, the company responsible for developing GPT-4, have been forthcoming about their goal of developing Artificial General Intelligence (AGI), [53], an AI system that is generally smarter than a human. OpenAI claims that such a system could "elevate humanity," accelerating scientific and social progress. The veracity of such claims remains unclear, and the possibilities, limitations, and implications of new AI technologies will continue to change and evolve.

What is immediately clear, however, is that the field is attracting an immense amount of competition— and funding. The 2023 Stanford AI Index Report [4] found that in 2022, global corporate investment in AI ventures totaled $189.6 billion, marking a 13-fold increase over the last decade. The AI boom may have brought staggering amounts of capital into the field and led to important technological advancements, but it has come at a cost.

Though it is difficult to directly estimate the prices paid by OpenAI and its competitors for hardware and electricity, estimates using standard cloud computing pricing place the cost to train their GPT-3 model at somewhere between \$4.6 million and \$12 million [50] [51]. This economic cost is only a counterpart to the carbon emissions generated by the energy consumed by training. GPT-3's training consumed 1,287 MWh of energy, generating 502 metric tons of CO_2 emissions [59], 8 times the amount emitted by the average car over its expected lifetime [63]. From a sustainability perspective, we must consider not only the carbon emissions of electricity generation but also the damage done by resource extraction and industrial production of computing hardware, much of which will be destined for obsolescence within a few years [21]. The AI boom is reminiscent of another field which experienced massive growth — and eventually a proportionally massive crash — in recent years.

Nowhere is the shortsightedness of our modern relationship to technology more visible than in the cryptocurrency ("crypto") sphere. Created to address fears that centralized currencies required too much trust in third parties, Bitcoin and the many cryptocurrencies that followed in its footsteps adopted new decentralized technologies. These technologies are based on intentionally duplicative systems used to create, store, and transact digital currency [7]. While many of these systems are in some way wasteful, the most direct environmental damage comes from proof-of-work (PoW) mining, which is the set of algorithms currencies like Bitcoin use to create and transact currency. In PoW mining, many participants enter a race wherein they perform brute-force computations (i.e., guessing) to solve a difficult math problem. The first participant to solve the problem is awarded a predetermined amount of currency.

This brute-force computation is unnecessary and wasteful; the "work" miners are asked to "prove" is the spent electricity they must consume and pay for to participate in the race. The competitive basis of this technique means that the cryptocurrency world operates in a constant battle to find the cheapest electricity and most powerful computers, ensuring a steadily growing cost to the planet. Mining of Bitcoin alone ranks 27th in the world on the list of nations' electricity consumption, resulting in an annualized equivalent emitted 71.9 megatons of carbon dioxide (MtCO2e), slightly less than the greenhouse gas emissions of New Zealand [9].

Although Bitcoin and its descendants achieve the goals their designers envisioned, they do so without consideration for their effects on the planet and the living things on it. The crypto sphere persists, and does so profitably, because of continued investment by both large firms and individual people who believe in the technology. In 2022, crypto attracted $23.1 billion in investment activity [54]. Like AI, the crypto boom created new technologies, solved new problems, and attracted attention from investors and consumers alike. But it's not enough to say that a particular technology solves problems economically today. If we are going to build a sustainable future, much more research must be done, and with a new focus in mind.

In recent years, a new aesthetic movement has emerged which seeks to answer the question "What does a sustainable civilization look like, and how can we get there?" This movement, dubbed Solarpunk [2], bucks the moody overtones of dystopic science fiction in favor of hopeful utopianism, envisioning a lush, green future where the practical merges with the beautiful. Solarpunks hold that by propagating their optimistic aesthetic they will inspire solutions that move humanity forward, away from climate destruction and artificial scarcity. The movement has inspired researchers, and this thesis is envisioned and executed wholly in support of Solarpunk goals.

One recent Solarpunk-associated project sought to change the paradigm of the modern web, beginning with one question— when a user visits a webpage, how should their request be directed? Today, a site may be served from one of hundreds or thousands of servers, and routing requests is a complex problem. For for-profit services, only one goal informs that process: speed. Faster page loads make users more likely to stay on a site, thereby generating more profit in a highly competitive online economy [68]. But it is no longer sufficient to view our relationship with technology through a narrow capitalist lens. We must consider sustainability, ecological impact, and planning for an uncertain future.

In 2021, the Solar Protocol website went live, powered by a distributed and self-organizing network [8]. Rather than directing queries based on latency and server load, the network asks one question: where is the sun shining? The network is made up of low-powered server equipped with photovoltaic panels and batteries. These servers exchange data about how much solar energy they are collecting, and every two minutes, the station with the highest energy availability is named the active server. All requests to solarprotocol.net are served by that station. A user in the United States accessing the page at night might be served from Kenya or Australia. The Internet is a global network, and the sun is always shining somewhere. The Solar Protocol integrates those principles and challenges the idea that speed should be our primary concern, intentionally sacrificing latency in exchange for a reliable and decentralized solar-powered network.

This thesis seeks to apply the same lens to the burgeoning field of AI. How can we work towards a future where the enormous practical benefits of AI merge with the shrinking availability of electrical resources? In simple terms, how can AI be made more efficient? Capitalism acts to delay and externalize the consequences of wanton resource consumption, and to look to the future we must look outside of profitability, just as the Solar Protocol project has. To that end, this thesis will explore technologies which can make reduce the carbon footprint of AI. I will begin with a contextual investigation which seeks to explain in simplest terms what AI is and how it functions. With this knowledge in hand, I explore, implement, and test Approximate Multipliers, a promising technology which can accelerate neural network training and inference.

Chapter 2

Context

2.1 What is Efficiency?

A computer, defined succinctly, is a machine which performs computations. These computations are performed by logical circuits residing within Central Processing Units (CPUs) and Graphics Processing Units (GPUs), each containing billions of transistors, devices which can modify and switch electrical current. By connecting the inputs and outputs of many transistors to one another in a vast and complex network, logical circuits are formed which can execute instructions and perform computations (see Figure 2.1).

When an electrical current reaches a transistor that is "switched off", thermodynamics dictates that its energy cannot simply vanish. Instead, the transistor's high resistance converts the energy into heat, which must then be dissipated. Practically speaking, all of the energy that enters a processor leaves it as waste heat [21]. Maximizing the amount of computing power that can be extracted from this energy maximizes the efficiency of the computer [16].

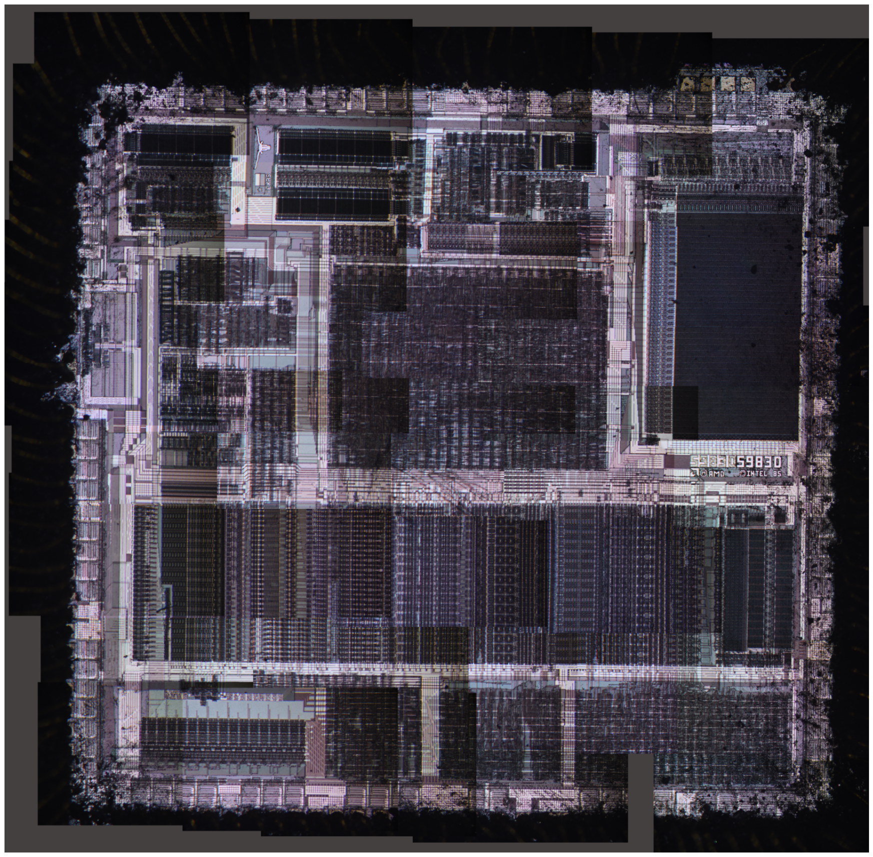

Figure 2.1: Microscopic imaging of the circuitry of an AMD Am386 CPU [57].

At the datacenter scale, efficiency is measured in terms of FLOPs per watt. FLOPs, or Floating-Point Operations per Second, are a common measure of computing power, since floating-point operations are one of the most common workloads in datacenter and high-performance computing environments. Floating point multiplication specifically accounts for almost all of the computational power used in training and evaluating deep neural networks [30], which are the computational backbone of modern AI technologies (see Section 2.4), and is thus of particular interest to the aims of this thesis. FLOPs per watt quantifies how many floating point operations a datacenter or supercomputer can perform on a per-watt basis. The most efficient supercomputers are tracked by the Green500 list [46], which is updated biannually. The most recent ranking as of November 2022 saw the Flatiron Institute's Henri system take the lead at 65.09 GigaFLOPs per watt [52]. The design of more efficient supercomputers and datacenters is an ongoing challenge in hardware and infrastructure engineering, but it is not the only way to decrease the energy usage of a given workload.

The design of more efficient general-purpose hardware is important and necessary, but within the scope of this work, I am only concerned with the efficiency of training and evaluating neural networks. Processors which are designed for a particular workload can be much more efficient, since they do not need to generalize. For instance, in cryptocurrency mining, application-specific integrated circuit (ASIC) miners can far exceed the efficiency of CPUs and GPUs [39], allowing vastly higher performance per watt. Applying this design philosophy to deep neural networks could lead to much lower energy usage. But we must first understand what neural networks are, why multiplication is key to the aims of this thesis, and how we can exploit neural networks' structure to optimize them.

2.2 What is a Neural Network?

The goal of a neural network (NN) is to approximate a solution to a complex function using training and testing data. A network consists of layers of neurons connected to each other in a particular structure known as an architecture [20]. The first layer is the *input layer*, which is connected to one of more *hidden layers*; the final layer is known as the *output layer*. Each neuron in each layer takes numerical inputs and multiplies them by weights, then applies an activation function to produce an output. Networks with more than one hidden layer are referred to as *deep neural networks* (DNNs). The result of the network's evaluation is observed at the output layer.

In simple (feed-forward) networks, the outputs of previous layers act as inputs to following layers, but more complex (recurrent) networks use the outputs of later layers as inputs to previous ones. A neural network can be visualized as a graph by diagramming neurons as nodes and their connections as edges. A feed-forward network is a directed acyclic graph. That is, the graph contains no loops, and its connections flow in one direction. A recurrent network is a directed cyclic graph, as it does contain loops. By passing an input to the network and calculating the results of each layer, an output can be calculated; this is known as *forward propagation.* This result can then be compared to the expected output to calculate the error. Then *backpropagation* is performed and an optimization algorithm updates the weights of the inputs for each neuron in each layer. To motivate this structure, it is useful to have in mind a brief history of neural networks and to understand their biological inspirations.

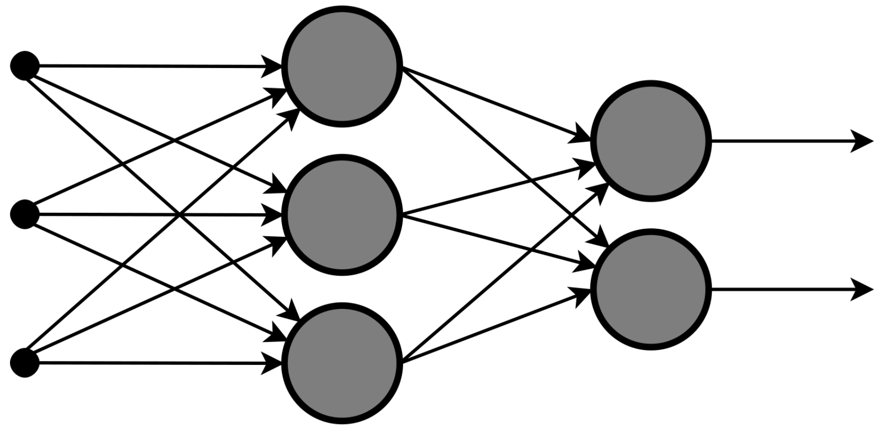

Figure 2.2: A feed-forward neural network diagrammed as a directed graph, made up of an input layer, a hidden layer, and an output layer [48]. Each gray circle is one artificial neuron.

2.3 History of Neural Networks

In 1946, McCulloch and Pitts' paper "A logical calculus of the ideas immanent in nervous activity" introduced the idea of the artificial neuron [40]. Its authors, a neurophysiologist and logician, built their model to imitate the behavior of neurons in the human brain. They focused on replicating the functionality of the three major components of a neuron: the soma, the dendrite, and the axon (see Figure 2.3). The

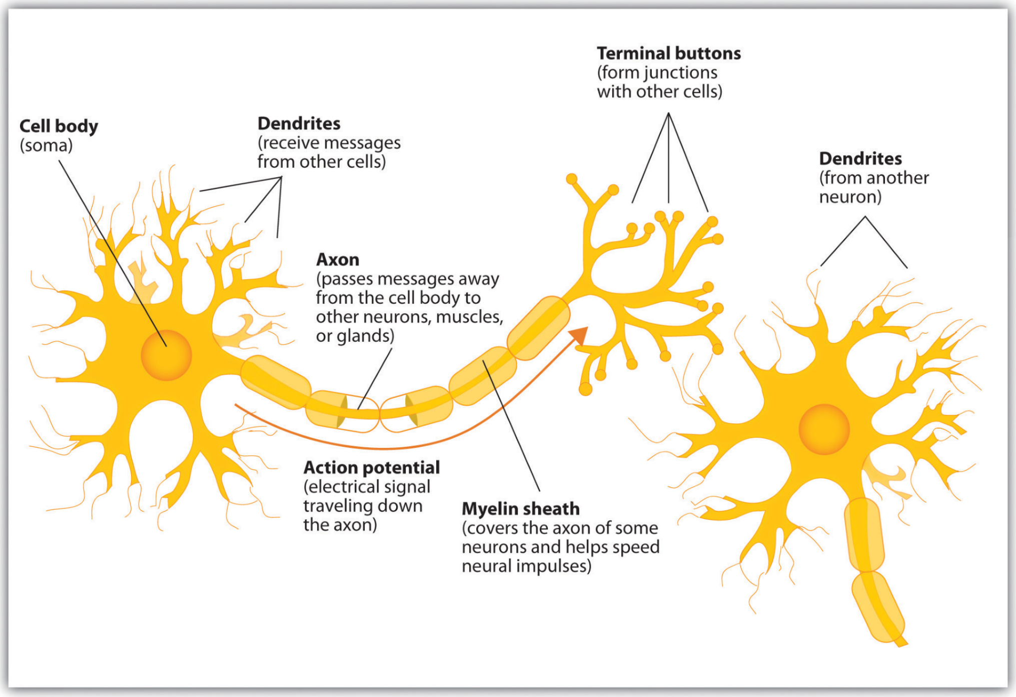
Cell body
(soma)
Dendrites
(receive messages from other cells)
Axon
(passes messages away from the cell body to other neurons, muscles, or glands)
Terminal buttons
(form junctions with other cells)
Dendrites
(from another neuron)
Action potential
(electrical signal traveling down the axon)
Myelin sheath
(covers the axon of some neurons and helps speed neural impulses)

Figure 2.3: A simplified diagram of the components of a human neuron [62].

dendrite receives signals from other neurons, the soma processes them, and the axon transmits the output of the neuron [62]. The researchers simplified these components to create a computerized equivalent, today known as the McCulloch-Pitts (MCP) neuron.

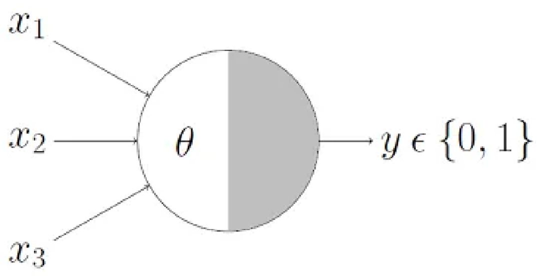

Figure 2.4: A diagram of the MCP neuron. The activation threshold is θ [10].

The MCP neuron models the dendrite with n binary inputs $x_1 \ldots x_n$, each with a value of 1 or 0 (active or inactive). These inputs can be either excitory or inhibitory. The "soma" is an additive computation which checks if the number of active excitory inputs is greater than a certain value, called the activation threshold. If this is true and no inhibitory inputs are active, the "dendrite" output is 1. If the count falls below the threshold or there are any active inhibitory inputs, the output is 0.

The MCP neuron can simulate several first-order logical operations, such as OR and NOT (see Figure 2.5). However, a single neuron is unable to represent all functions, and the MCP neuron is unable to learn; its activation threshold must be set manually.

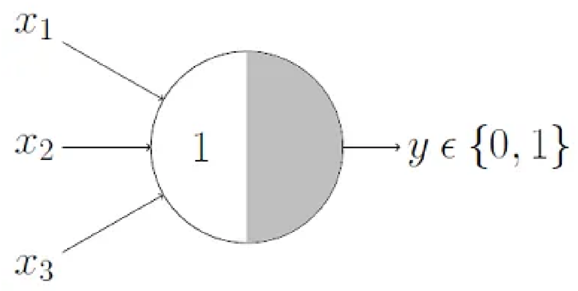

Figure 2.5: An MCP neuron which evaluates an OR function. If any of the inputs are active, the neuron will output 1 [10].

In 1957, the MCP neuron was improved upon by Rosenblatt [56], who also created a physical model known as the Mark 1 Perceptron (see Figure 2.6). Rosenblatt introduced the ideas of weights and biases, an important feature in modern neural networks (see Section 2.3), allowing the Mark 1 Perceptron to learn from training examples. In 1969, Minsky and Papert published their seminal work *Perceptrons* [41], which proved that Rosenblatt's single-layer perceptron could not simulate several classes of logical functions, including XOR. To represent more functions, it is necessary to connect perceptron layers to each other, with the output of one layer becoming the input to another. The neurons in each layer form a network which carries data from

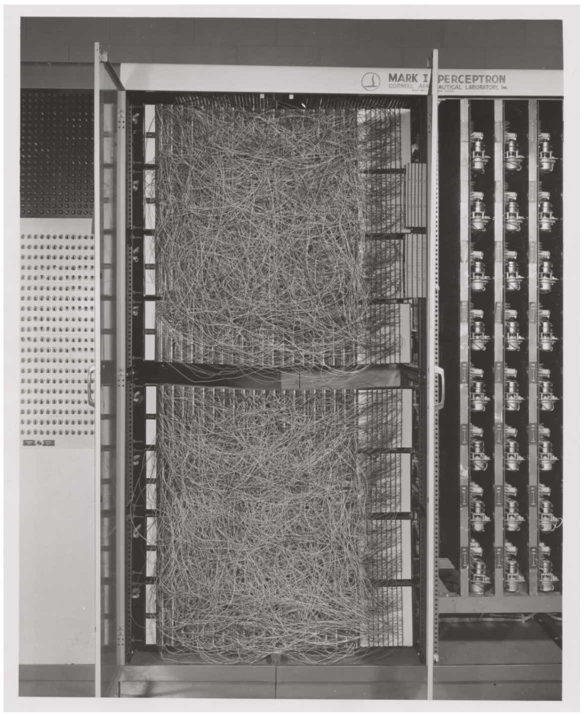

Figure 2.6: Rosenblatt's Mark 1 Perceptron at the Cornell Aeronautical Laboratory. It contained 400 neurons connected to a 20x20 array of sensing photocells [37].

the input to the output, the so-called "multi-layer perceptron." This structure can be generalized to reach the concept of the feed-forward neural network. To discuss feed-forward neural networks, I must explain in more detail the properties of modern artificial neurons.

Structure of an Artificial Neuron

A modern computerized neuron is a function with $m + 1$ inputs, all but one of which are numerical signals x_1 through x_m, with the x_0 input denoting the bias. The bias term is always 1 with its own weight w_0. The rest of the inputs have weights w_1 through w_m. These weights are multiplied by the inputs and the resultant values are summed to produce the value y (see Figure 2.7).

$$y = w_0 + x_1 w_1 + x_2 w_2 + \ldots + x_m w_m \tag{2.1}$$

An activation function is applied to y to give the final output of the neuron. Common activation functions include the Sigmoid function $\sigma(y) = \frac{1}{1+e^{-y}}$, Hyperbolic Tangent $\tanh(y) = \frac{e^y - e^{-y}}{e^y + e^{-y}}$, and ReLU, Rectified Linear Unit, defined piecewise as:

$$\mathrm{ReLU}(y) = \begin{cases} 0 & \text{if } y < 0 \\ x & \text{if } y \geq 0 \end{cases}$$

Or more succinctly as $\mathrm{ReLU}(y) = \max(0, y)$. The activation function typically scales the output to within a particular interval, often from 0 to 1, and introduces nonlinearity, which is desired for effective training [20].

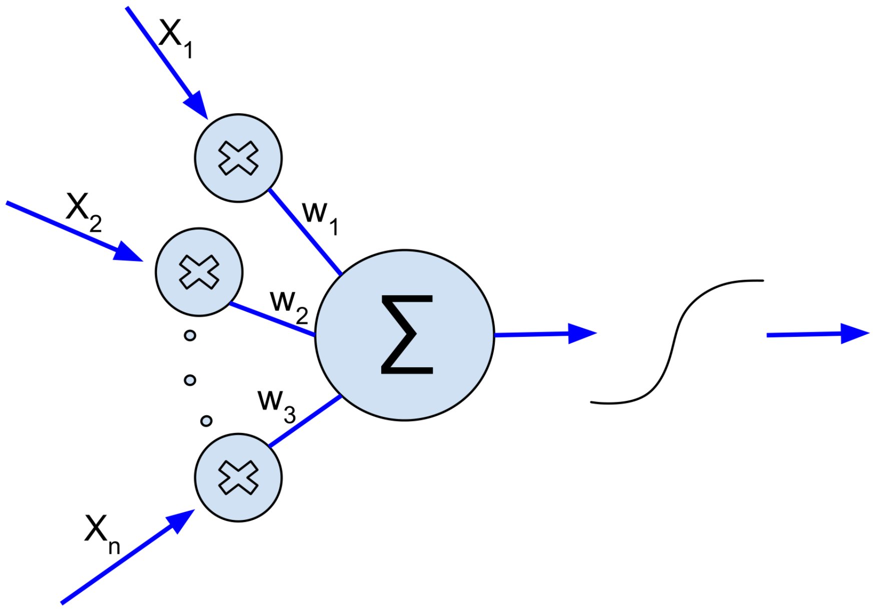

Figure 2.7: A single artificial neuron with inputs x_1 to x_n, weights w_1 to w_n, and a sigmoid activation function [6].

Training

Training as a process involves three datasets which contain the same type of data: training, testing, and validation sets. Often these datasets are randomly selected ("split") from a larger dataset to ensure each dataset contains the most variability possible; more diverse examples will lead to improved training results because the model can better grasp patterns in the underlying data and generalize to new inputs [61].

First, the network is evaluated on data from the training set. The process of calculating neuron inputs and outputs is known as forward propagation. Forward propagation gives a set of values at the output layer. Then this result can be compared to the known correct output, often called the target or ground truth. A loss function calculates the error between the output and target, and the error is backpropagated to adjust the weights of each neuron, a process known as learning. The goal of learning should be to find the set of weights that give the smallest possible value (minimum) of the loss function as efficiently as possible. This is an optimization problem often solved by stochastic gradient descent (SGD)or similar algorithms. By traversing the surface of the loss function, the optimization algorithm attempts to find the best possible local minimum, and ideally the global one.

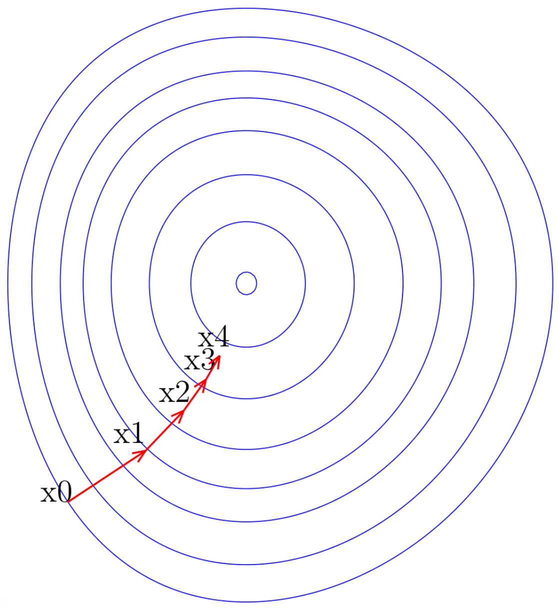

Figure 2.8: A gradient descent algorithm approaching the minimum of a gradient, seen from above with contour lines to represent depth [70].

After several training iterations, the network is evaluated against the validation dataset. Often, this will take place at the end of each *epoch*, wherein the network learns on the entire training dataset once. Since the validation dataset contains examples the network has not encountered before, it is a useful benchmark for common training pitfalls, such as overfitting. Overfitting occurs when the network models the training data too specifically. This leads to low loss values on the training set, but results in a failure to generalize to new data. The validation set helps identify this problem in the training phase, giving users the ability to avoid wasted compute resources and decreased accuracy by stopping training early. Finally, when training is complete, the model is evaluated against the test set, to test the performance of the final model against new data. While training uses loss as the metric to update the weights, when evaluating against the test set, metrics such as accuracy or precision are used to test the model's performance [20]. Evaluating the trained model is known as *inference*. This overview of the training process is applicable to most types of neural networks. Below is a brief synopsis of the most common architectures.

Feed-Forward Networks

Feed-forward describes a network architecture in which information flows through the network in only one direction. They are the direct descendants of multi-layer perceptrons (see Section 2.3). In a feed-forward network, the input is evaluated by successive layers where the output of the previous layer forms the input of the next. One common use case for this architecture is the Convolutional Neural Network (CNN), which has applications in object recognition and image processing [3]. AlexNet [35], used to test approximate multipliers in the Technical Investigation (see Section 5) portion of this thesis, is one such network.

Recurrent Networks

Recurrent neural networks (RNNs), in contrast to feed-forward networks, utilize the output of later layers as inputs to earlier ones in successive evaluations. This allows the network to exhibit temporal behavior akin to memory. RNNs are often used in processing sequential data, since the network can "remember" previous data in the sequence while processing a long input [3]. RNNs are a key tool in natural language processing, where the network can operate on a sequence of words in a sentence, and were the state of the art in this field until the development of the Transformer model in 2017.

Transformers

The Transformer model, as described in the now-legendary paper "Attention is all you Need" [67], takes a step back from recurrent neural networks. The architecture of a Transformer is a stack of building blocks known as encoders and decoders (see Figure 2.9), each of which contains a feed-forward neural network and a multi-headed attention (MHA) mechanism. The encoder stack processes data sequentially, with the output of each encoder flowing into the MHA mechanism of the subsequent encoder. The decoder stack is similar, except that the output of the encoder stack forms one input to the MHA mechanism of every decoder. The MHA mechanism plays a similar role to recurrency in neural networks, serving as a kind of "memory". MHA, however, is more complex and allows the Transformer to attend to the output of every previous encoder and decoder, giving the model an enormous amount of contextual awareness. The Transformer model has enabled the development of new Large Language Models (LLMs) such as GPT-4 [49], BeRT [14], and LLaMA [65], all of which showcase groundbreaking performance on natural language tasks.

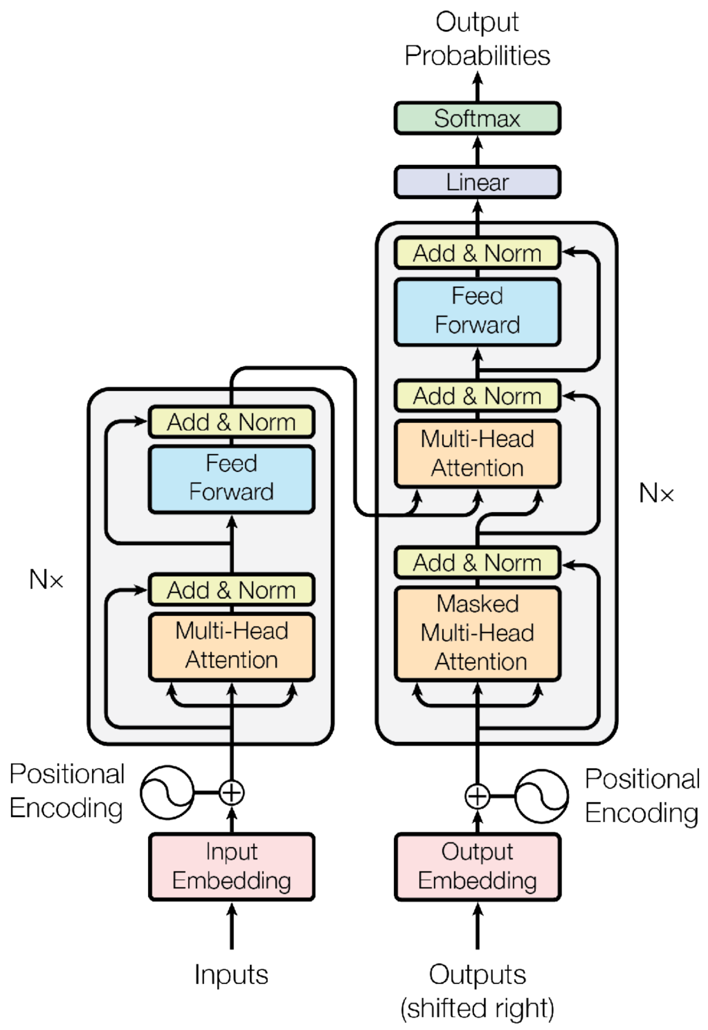

Figure 2.9: Internal structure of the Transformer model, with the encoder stack on the left and the decoder stack on the right [67].

2.4 Why is multiplication so important?

Neural networks, as we have seen, are essentially composed of multiplication. To evaluate the network, the weights and values of every neuron must be multiplied and summed. This operation accounts for ~99% of the total compute resources involved in modern DNNs [30], and is therefore a key consideration in improving efficiency. Why is multiplication such a computationally expensive operation?

Consider multiplying two numbers using a common written method (long multiplication):

$$\begin{array}{cccccc}
 & & & 5 & 4 & 3 \\
 & & \times & 8 & 6 & 7 \\
\cline{3-6}
 & & 3 & 8 & 0 & 1 \\
 & 3 & 2 & 5 & 8 & \\
4 & 3 & 4 & 4 & & \\
\hline
4 & 7 & 0 & 7 & 8 & 1
\end{array}$$

It is necessary to multiply each digit of the multiplier by the multiplicand and then add them. The quadratic nature of this naive algorithm is immediately clear; multiplying two n-digit numbers requires $O\left(n^2\right)$ single-digit arithmetic operations. Put simply, because multiplication is repeated addition, it will always involve many operations. When dealing with high-precision floating point numbers like those used in modern DNNs, which may have up to 32 binary digits [29], this quadratic complexity becomes a major concern.

Modern algorithms for computerized multiplication are highly optimized both mathematically and in their hardware implementations. The current best-known algorithm for integer multiplication operates in time $O(n \log n)$ [26]. Even so, multiplication remains highly compute intensive. Reducing this cost, therefore, is an important topic in modern DNN research. Many techniques exist, and a review of the most common will motivate the choices made in the technical investigation (see Chapter (5).

Chapter 3

Related Works

Reducing the computational (and thus energy) cost of DNNs is a problem which has seen a great deal of attention in recent years. One technique, known as quantization, involves reducing the number of bits (binary digits) used to store the model weights, thus decreasing the cost of multiplying them. However, this technique is known to reduce model performance if used too aggressively [30]. The authors of [12] showed that while quantization (to a 16-bit fixed-point representation) did not significantly increase inference error in a CNN, it prevented the network from learning to classify the data if used during training. Very precise representations are important during training [19], so quantization is most often applied for inference.

Sparsification, also called pruning, is another emerging technique. To sparsify a model, an algorithm *prunes* model weights by setting them to zero, allowing that weight to be ignored. Even a very sparse model can achieve results on par with a non-sparse one [28]. To maximize performance, in works such as [17] and [25], the authors prune weights and then retrain the model. Recently, however, it has been shown that model pruning without retraining (one-shot pruning) is possible on models as complex as the Transformer [17] with negligible performance loss. Discussion of sparsification during training is ongoing, but it has been applied successfully [69, 36]. Note that sparsification necessitates complex additional algorithms to select weights for pruning [28].

Techniques like sparsification and quantization are viable methods for reducing the energy usage of a neural network, but are often approached from the perspective of enabling model inference on consumer devices, also known as edge computing. These techniques reduce the footprint of the model in memory, which is often the limiting factor on edge devices [30]. To this end, models like Stanford's Alpaca [55] and Berkeley's Koala [18] fine-tune small language models using the output of larger ones. By curating this output data, models can be trained which run on consumer GPUs and excel at one task (such as instruction-following), performing to a high standard. For example, responses to Koala are rated by humans as on-par with ChatGPT about half the time [18]. This research will become more important as AI becomes more commonplace on consumer devices and in everyday life; reducing the energy consumption of inference will reduce the carbon footprint of these use cases. However, training remains a compute-and-energy-intensive process with a large carbon footprint [63]. The above techniques are most effective in the inference phase and require additional implementation work.

By contrast, approximate multipliers can be utilized in pretrained models with no change to architecture, no retraining, and minimal or no performance degradation [58, 32, 31, 23], acting as a drop-in replacement for precise multipliers. AMs are also a promising technology for reducing the energy consumption of training, again with minimal or no performance degradation [60, 24] and in some cases a slight performance increase [19]. They are especially useful in datacenters, which have abundant memory but consume immense amounts of energy [21]. AMs can also be used in concert with other techniques. [19] combined AMs with pruning and retraining in a CNN and showed improved accuracy compared to 32-bit precise multipliers for most levels of sparsity.

Much of the study of AMs in deep neural networks has been limited to inference applications. For example, in [58], the authors replaced precise multipliers with AMs in AlexNet to study whether DNNs were a viable application of the technology. By exchanging precise multipliers for MBMs, they demonstrated a 57x reduction in power and 27x reduction in area contributed by the multiplier, with negligible degradation to classification accuracy. However, they did not train the network using AMs, but only tested classification performance using weights from training using precise multipliers. Similarly, in [32], [31], and [23], authors implemented AM-based neural networks in order to test novel AM designs, but only used pretrained networks.

In 2019 [24] implemented a simulated AM which added random noise to the result of a precise multiplier. They used this simulated AM to train a CNN and found that it resulted in minimal accuracy degradation. Then in 2021 [60] trained a small CNN using AMs and found that it achieved the same classification accuracy as the same architecture trained using precise multipliers. To enable further testing of AM-based DNNs, a library was needed which would allow easy implementation of AM layers within frameworks like TensorFlow [38]. The first such library was TFApprox [64], which allows GPU simulation of AMs within TensorFlow. However, TFApprox is limited to inference and supports only 8-bit integer multiplication. In 2022, the ApproxTrain project [19] overcame these limitations to provide a flexible framework for implementing AM layers in any TensorFlow model for both training and inference (see Section 4.1). The authors additionally showed that ApproxTrain models achieved performance on par with precise models in almost all cases. Before delving into the technical details of the framework, I will provide a brief explanation of the math behind approximate multipliers and how they are implemented in computing.

Chapter 4

Approximate Multipliers

To execute various operations, computers have networks of transistors which form logical circuits. The complete network must physically fit on the block of semiconducting material ("die") which the CPU or GPU is fabricated from. Thus, designers of these devices must carefully consider the die area devoted to each operation, choosing where to allocate resources to maximize performance. One of these operations is multiplication, which, as we have seen, is highly compute intensive. For many types of computing, precise multiplication is necessary. But DNNs, due to their size and imprecise nature, are resistant to small errors in multiplication results [43]. It is possible to build logical circuits that perform *approximate* multiplication, computing a slightly erroneous result, but at great savings to power consumption and the physical size of the circuit [58]. These so-called approximate multipliers (AMs) replace or supplant the precise floating-point multipliers in CPUs and GPUs, and can both accelerate AI computing and reduce its power consumption [19]. This literature review will discuss the concept and history of approximate multipliers and their use in neural networks before diving into their use in the ApproxTrain framework.

One early approximate multiplier was created by Mitchell [42] and takes advantage of the speed of computing binary logarithms in digital circuits. The binary log of an integer can be approximated by taking the most significant "one" bit as the significand of the logarithm and the rest of the bits as the mantissa. Note that in this context significand refers to the significant digits of the number. To explain, consider the usual binary representation of an N-bit unsigned integer $b_{N-1}b_{N-2}...b_1b_0$:

$$B = \sum_{i=0}^{N-1} 2^i b_i \tag{4.1}$$

Say the most significant one bit in the integer is at position k, where $(N-1) \leq k \leq 0$. B can then be written as:

$$B = 2^k \left(1 + \sum_{i=0}^{k-1} 2^{i-k} b_i\right) \tag{4.2}$$

Let $x = \sum_{i=0}^{N-1} 2^{i-k} b_i$, where $0 \leq x < 1$. The accurate binary log of B is then:

$$\begin{aligned} \log_2 B &= \log_2 \left(2^k \left(1 + \sum_{i=0}^{k-1} 2^{i-k} b_i\right)\right) && (4.3) \\ &= \log_2 \left(2^k + (1 + x)\right) && (4.4) \\ &= k + \log_2(1 + x) && (4.5) \end{aligned}$$

k, the most significant one bit of the sequence, forms the integer significand of the log value and the rest of the bits form the mantissa. Successive computations give a more accurate approximation of the result. For the purposes of approximate multipliers, however, the mantissa $log_2(1+x)$ is simply approximated as x, since $0 \leq x < 1$. Then, because log addition is the same operation as multiplication, the log values can be added and a reverse log calculated to find the result.

Mitchell's multiplier is very simple and fast, but has a high error bias. That is, the mean of relative error compared to a precise result is large- around 3.7%. The peak relative error is also high at 11.1%. More modern AM designs have lowered these values significantly. One such design, Minimally Biased Multipliers (MBMs) [58], augment the Mitchell multiplier with a novel error-reduction scheme. They offer a near-zero error bias (0.05% mean relative error for an 8-bit multiplier and <0.1% for a 16-bit one), have a lower peak error than the Mitchell multiplier (7.8%), and can be configured to trade accuracy for power and area. MBMs provide a peak power reduction of 84% and a peak area reduction of 75% compared to an accurate multiplier. They are particularly applicable to the aims of this thesis because they lie on the Pareto front for power efficiency versus peak error and mean error. That is, there is no known multiplier which uses less power while achieving better error bias or peak error. Because of this, they are the default multiplier used in the ApproxTrain framework.

4.1 ApproxTrain

The ApproxTrain project [19] comprises two main segments: a framework which enables software simulation of AMs, and a paper detailing both relevant prior work and the contributions made by said framework. ApproxTrain was developed for TensorFlow [38], a library which simplifies the development of new AI models, allowing the user to describe their architecture at a high level. For the end user, implementing ApproxTrain AMs is as simple as compiling the framework and replacing precise TensorFlow layers with AM-based ones in their model.

The framework provides two major software components: AMSim, a workflow for transforming C/C++ models of specific AMs into software simulators, and ApproxTrain, a TensorFlow framework for training and inference of DNNs using AMSim multipliers. In order to enable this functionality, the authors implemented a full custom CUDA kernel for GPU-accelerated AM operations. CUDA [44] is a parallel computing platform which can be used to accelerate many kinds of computing workloads using NVIDIA GPUs. Within CUDA, libraries such as cuBLAS (CUDA Basic Linear Algebra) and cuDNN (CUDA Deep Neural Network) provide functionality for accelerating DNN workloads. However, CUDA is a closed-source framework; it was necessary for the ApproxTrain authors to reimplement many of its features to enable AM functionality. Using AMSim and the custom CUDA kernel, it is possible for researchers to simulate and test many kinds of multipliers with high evaluation speeds. ApproxTrain's authors note a 2500x performance increase over CPU simulation, and the framework offers higher performance than existing GPU simulation libraries such as TFApproximate.

However, it is important to note that AMs are a hardware technology. To harness their full performance, they must be implemented physically. ApproxTrain is a framework for simulating these hardware devices on traditional GPUs, allowing researchers to develop and test AM designs much more quickly. The overhead of software simulation means that models developed with ApproxTrain AMs will train and infer much more slowly than models running on physical AMs, and in most cases more slowly than models running on precise multipliers. The implementation work performed in this thesis is not intended to demonstrate a more performant multiplier. Instead, I hope to introduce and explain the concept of AMs and show how they might be used to reduce energy usage in a future hardware DNN accelerator.

Due to the necessity of reimplementing a CUDA kernel, ApproxTrain is 1-5x slower than standard TensorFlow for precise multiplication. The authors state that this is reasonable because, as they put it, "the closed-source cuDNN and cuBLAS libraries have been optimized by teams of several hundred professionals within Nvidia for over a decade." When utilizing AMs for training and inference, ApproxTrain exhibits a 2-13x slowdown over standard TensorFlow, attributed to a lack of optimization and the overhead of software AM simulation. ApproxTrain models tend to exhibit a slight accuracy *increase* over models developed with precise multipliers, which the authors hypothesize is due to stochastic noise produced by AMs' lack of precision, which acts as a type of regularization [45].

ApproxTrain is an advancement over prior work on AM frameworks. It is an extremely useful tool for the development and testing of AM-based DNNs, and the framework is used to build a model in the Technical Investigation (see Chapter 5) portion of this thesis.

Chapter 5

Technical Investigation: AM-based Neural Network

Using the ApproxTrain framework in TensorFlow, I reimplemented AlexNet [35] with the goal of testing the performance of AMs in a novel scenario. As a CNN designed for image classification, AlexNet is an excellent choice for this technical investigation; it is primarily composed of two types of layers which ApproxTrain provides drop-in replacements for. In addition, despite being a well-known network that was state-of-the-art in 2012, AlexNet's age made training possible in a manageable time frame. Though related works [58] did use AlexNet for AM testing, they only did so for inference, not training. The ApproxTrain paper itself tested LeNet [22] and ResNet [27], two other well-known CNNs. LeNet, despite having a comparable structure to AlexNet, is significantly simpler. Thus this testing is a useful iteration on the work of the ApproxTrain authors.

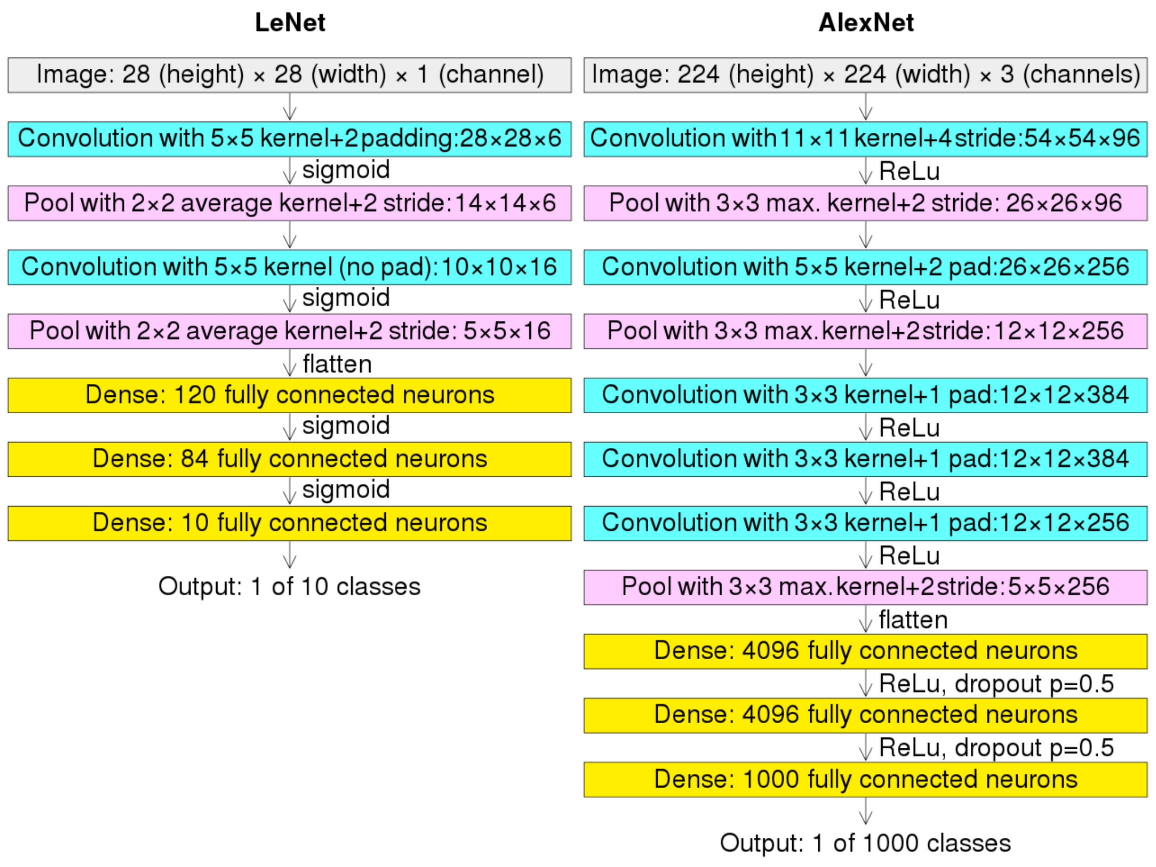
LeNet
Image: 28 (height) × 28 (width) × 1 (channel)
Convolution with 5×5 kernel+2padding:28×28×6
sigmoid
Pool with 2×2 average kernel+2 stride:14×14×6
Convolution with 5×5 kernel (no pad):10×10×16
sigmoid
Pool with 2×2 average kernel+2 stride: 5×5×16
flatten
Dense: 120 fully connected neurons
sigmoid
Dense: 84 fully connected neurons
sigmoid
Dense: 10 fully connected neurons
Output: 1 of 10 classes
AlexNet
Image: 224 (height) × 224 (width) × 3 (channels)
Convolution with 11×11 kernel+4 stride:54×54×96
ReLu
Pool with 3×3 max. kernel+2 stride: 26×26×96
Convolution with 5×5 kernel+2 pad:26×26×256
ReLu
Pool with 3×3 max. kernel+2 stride:12×12×256
Convolution with 3×3 kernel+1 pad:12×12×384
ReLu
Convolution with 3×3 kernel+1 pad:12×12×384
ReLu
Convolution with 3×3 kernel+1 pad:12×12×256
ReLu
Pool with 3×3 max. kernel+2 stride:5×5×256
flatten
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 4096 fully connected neurons
ReLu, dropout p=0.5
Dense: 1000 fully connected neurons
Output: 1 of 1000 classes

Figure 5.1: Comparison of the LeNet and AlexNet convolution, pooling, and dense layers [11].

5.1 Technical Details

Training was performed on my laptop, a 2021 ASUS ROG Zephryus GA401QM [1] with an AMD Ryzen 9 5900HS CPU and an NVIDIA RTX 3060 GPU. Because ApproxTrain was developed on Ubuntu 18.04 [66], it was necessary to modify the Makefile to successfully compile the framework on my modern system. For ease of integration, I trained inside the NVIDIA Container Runtime [47], a Docker [15] container which contains prebuilt versions of TensorFlow and CUDA. This made compiling ApproxTrain and running my test code much simpler.

I used a publicly available TensorFlow-based AlexNet implementation [5] as the syntactic basis for my code, then modified it to add the features I needed and incorporate AM-based layers. The final code and my modified version of ApproxTrain are available on GitHub [33].

While AlexNet was originally designed to classify the 1000-class ImageNet dataset [13] (as seen in Figure 5.1), I instead chose to train on the CIFAR-10 dataset [34], because its smaller size enabled faster training. CIFAR-10 contains 60,000 32x32 color images in 10 different classes, such as airplanes, birds, and cats. Because the dataset provides only training and testing splits, I used the first 5,000 training images for validation and the other 55,000 for training. I trained for 50 epochs.

Other than changing the dataset and incorporating AMs, I made no changes to the architecture of AlexNet; my goal was to show that AMs could easily integrate into an existing architecture. I tested both precise and approximate multipliers, training each for 50 epochs. To ensure that the precise and approximate models would have equal weights at the beginning of training, I explicitly set the TensorFlow random seed. Both models use the Glorot Uniform initializer for their weights and initialize biases to zero. While testing AMs, they were used for both training and inference.

For more detailed information on the development and testing process, see my Lab Notebook (Section 7.1).

5.2 Training Results

For comparison, I first tested the AlexNet implementation with precise multipliers. The final validation accuracy was 81.8%, but the final training accuracy was 98.5%, indicating some overfitting of the model. Training completed in 46 minutes, for an average time of 55 seconds per epoch.

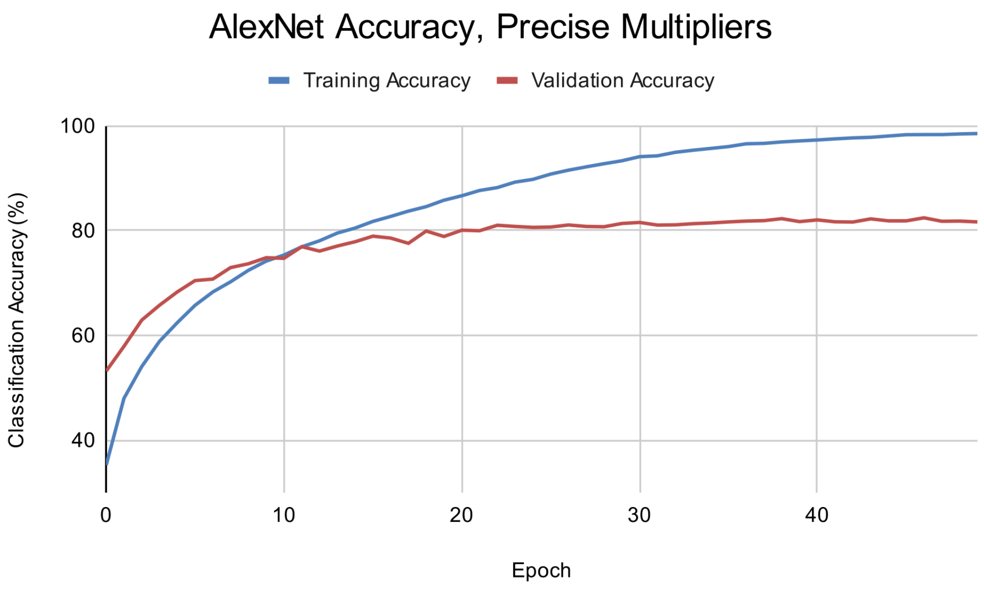

Figure 5.2: Accuracy plot for AlexNet using precise multipliers.

As seen in Figure 5.2, peak validation accuracy is reached early in training, with very little accuracy increase past Epoch 20. Training accuracy, however, increases to 98.5% as the model overfits.

I then exchanged the Convolutional and Dense layers in the model for their AM-based equivalents and retrained the network. As expected, AM-based training took significantly longer; a total time of 20 hours and 4 minutes, for an average time of 24 minutes 48 seconds per epoch. The test accuracy was negligibly lower than the precise training run at 81.7%, but the training accuracy was much lower at 93.4%.

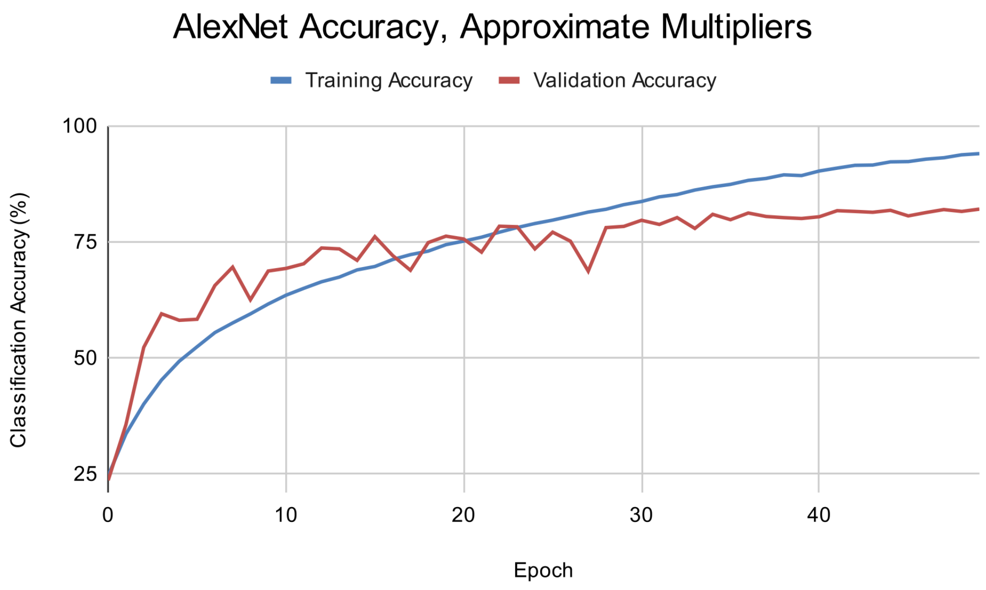

Figure 5.3: Accuracy plot for AlexNet using approximate multipliers.

As Figure 5.3 shows, the AM-based model is significantly less accurate at the end of the first training epoch. One explanation for this is that the error introduced by AMs is harder to account for in the early stages of training. However, this result is inconsistent with the results of the ApproxTrain paper, where accuracy was the same at the first training step and increased at approximately the same rate throughout training. Understanding the source of this discrepancy is one path for future research. By the end of training, however, the validation accuracy of the approximate model was equal to that of the precise model, a result consistent with previous investigations into training with AMs.

Of note is the "spikiness" of the validation accuracy plot. While Figure 5.2 shows a smooth increase in accuracy, the AM-based validation accuracy decreases several times during training. This can be attributed to the model failing to account for the AM error in the validation stage, since it has not encountered that data before. By the end of training, accuracy remains steady.

At the end of training, AMs show a smaller discrepancy between training and validation accuracy. This would appear to indicate that AMs reduce overfitting. However, further investigation is warranted. Figure 5.4 compares the accuracy plots for both types of multipliers. Here we see that despite the two multipliers displaying different accuracies early in training, they learn at similar rates. While the discrepancy between the training and validation accuracy scores is smaller at epoch 50 for AMs, the slope of the training accuracy plot does not decrease as training continues. This suggests that the model is less overfit not because of AMs but because it reached peak accuracy later in training. If training had continued, the AM-based model would likely have continued to overfit. Thus I conclude that AMs do not significantly reduce overfitting for this model architecture and implementation. However, a future work might investigate whether the usage of AMs reduces or eliminates the need for regularization layers such as the Dropout layers used in AlexNet, since the noise they introduce can have a regularizing effect [19].

Plotting the loss values (Figure 5.5) corroborates these conclusions. In the precise run, while the training loss decreases steadily, the validation loss increases after epoch 23, when it reaches a minimum of 0.59. At epoch 50, the validation loss is 0.73, indicating overfitting of the model. The initial training loss value of the approximate run is much higher, but decreases quickly, and by epoch 50 has begun to converge with that of the precise run. The approximate validation loss, however, does not decrease steadily. Unlike the precise validation plot, it shows several upticks which correlate with dips in the validation accuracy. At epoch 50, the approximate validation loss is also much higher than the precise run (2.38), despite equal accuracy values.

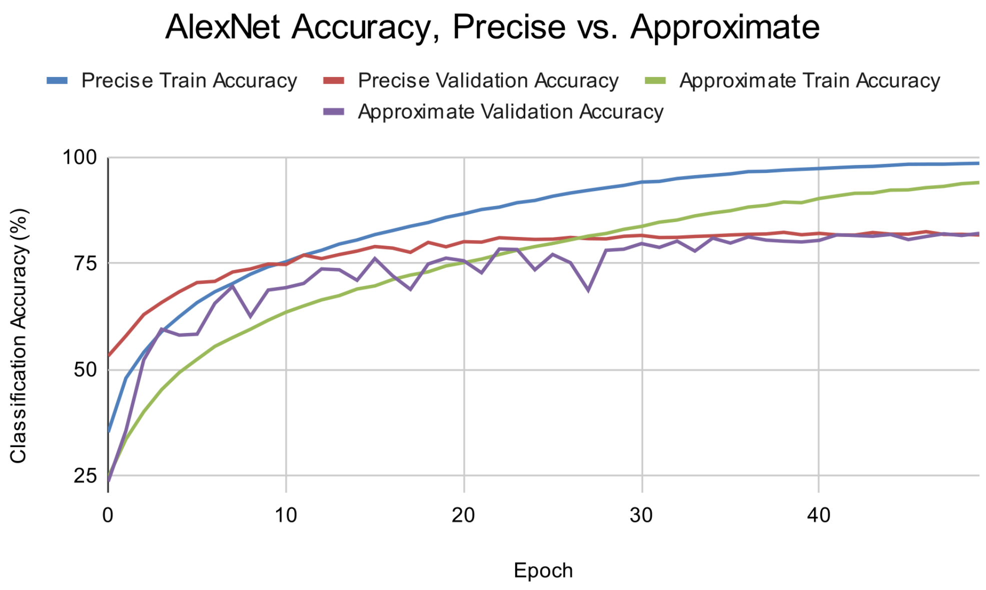

Figure 5.4: Accuracy plot for AlexNet comparing precise and accurate multipliers.

This technical investigation corroborates existing research into AM-based DNNs. I showed minimal accuracy degradation compared to precise multipliers, even when AMs are used during training. I conclude that AMs are a very promising technique for reducing deep learning energy usage with few downsides.

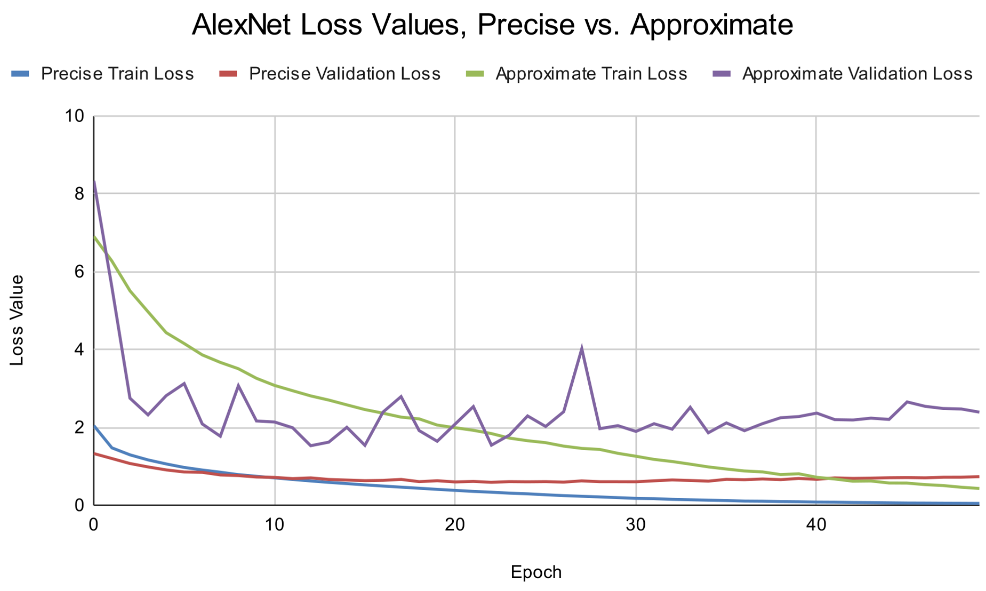

Figure 5.5: Loss plot for AlexNet comparing precise and accurate multipliers.

Chapter 6

Conclusion

In this thesis I showed that the AI field's technical advancements, while impressive, have come at a great cost of energy and with a high carbon footprint. If AI is to remain a practical technology in an uncertain future, we must consider methods for reducing its resource consumption. I explained the architecture of modern AI technologies and demonstrated why a key component — multiplication — is so computationally expensive. With that knowledge in hand, I conducted a literature review discussing existing research into reducing deep neural network power usage and showed why approximate multipliers (AMs) are a promising technology for this application. I analyzed the results of a novel technical investigation and found that it was possible to train an AM-based neural network with minimal accuracy degradation, corroborating existing results.

With all that in mind, I believe that AMs are deserving of much more research and testing. A hardware implementation would allow a physical analysis of their reduced energy usage. It has been shown that they can improve model performance in several DNN architectures, and do so without retraining or additional integration work, meaning they are much easier to implement than other techniques. A hardware AM-based DNN accelerator could easily be installed in existing servers or integrated into existing DNN computing architectures to reduce their energy usage and improve their performance. Wide-scale implementation of such a technology could have a measurable impact on the energy consumption of the AI space with few if any disadvantages.

Chapter 7

Appendix

7.1 Lab Notebook

2022-10-16

Attempting to install and use ApproxTrain. The software was designed on Ubuntu 18.04 and compilation failed initially. After troubleshooting, success came with changing makefile:

- Use modern c++ (c++11 was specified, but my version of TensorFlow is written in modern c++)
- Change CUDA location to correct one

After install, tested some of the LUTs. Results:

LUT	Accuracy
MBM_7	0.9646
MBM_5	0.9677

| MBM_1 | 0.9547 |

Why is MBM_7 Higher? Retesting gives 0.94 and 0.96. Maybe test consistency of this method, and test against standard TF mnist.

2022-11-01

Working on power management. Considering using a hacked kill-a-watt. NVML can be used to get board power and has a python library, here patched for python3: https://github.com/nicolargo/nvidia-ml-py3

But that breaks nvitop so I think default is ok (pip install nvidia-ml-py)

Because I'm using hybrid mode, the only think using my GPU is the compute, so this is probably pretty accurate.

I can see power usage in tools like nvidia-smi and nvitop, just need to access these power bindings and create a wrapper program.

xorg appears to be on gpu 0
i need to look into this

For the wrapper program- use subprocess? Then just sample every second and sum the instantaneous values to get Wh.

Change number of training steps
track execution?
test scaling
power usage graph

2022-12-21

For my final project in Artificial Intelligence, I decided to use my thesis topic, giving me the chance to organize my research and present the progress I'd made. I decided to train my models on the MNIST_fashion dataset, on the advice of Zach, as it's a much more complex dataset than the original MNIST testing set that shipped with the ApproxTrain demos. I should test the system on more datasets.

To save my work for the ApproxTrain modifications, I forked the repository at github.com/knauth/ApproxTrain

Here I will include the most relevant portion of my writeup, the technical summary:

Project Summary and Results

My network is adapted from the ApproxTrain examples, and is made up of nine layers:

- Input Layer (Keras, 28x28)
- *AMConv2D (Relu, 32 Filters, AM)*
- MaxPooling2D (Keras, downsampling)
- *AMConv2D (Relu, 32 Filters, AM)*
- MaxPooling2D (Keras, downsampling)
- Flatten (Keras)
- *DenseAM (Relu, AM)*
- Dropout (Keras, prevents overfitting)
- *DenseAM (Softmax, 10 outputs, normalizing)*

The results of training are summarized below, per-epoch. Four bit-widths were tested for the Mitchell Logarithmic Multiplier (shown in parentheses), and the final row is the default CUDA multiplier (mnist_fashion_noam.py). The value shown is sparse categorical accuracy.

Multiplier	1	2	3	4
MIT-1 (10)	0.71	0.71	0.76	0.80
MIT-3 (12)	0.69	0.75	0.77	0.77
MIT-5 (14)	0.72	0.67	0.80	0.80
MIT-7 (16)	0.66	0.75	0.78	0.81
CUDA	0.88	0.89	0.90	0.91

There are several things to note about these results. Precise multipliers gain a ~10% accuracy advantage, but they also have more predictable results per-epoch. At several points we see a drop in accuracy when using AMs.

There is very little difference in results between bit-widths for the AMs. More research could illuminate whether this is merely a consequence of the model architecture or inherent to the technology.

The CUDA model is notably faster to run, contrary to expectations. This discrepancy is explained in the ApproxTrain paper- the overhead of simulating AMs in software means we lose the benefit of decades of hardware and software optimization by thousands of people. FPGA-based AMs are proving a promising front for research, as is custom silicon.

2023-02-16

After some more research into X and how it uses the GPUs, I've managed to prevent Xorg from using my discrete GPU, which should give me more accurate results when running models on it.

2023-04-11

I decided to drop the power tracking goal. It's not incredibly relevant now that I know AMs are going to use more power in software; how much isn't important to their actual hardware performance, and is mostly an implementation question.

I am unsatisfied with the work I've done so far for AM training and testing, so I am implementing a more advanced network. I'm choosing AlexNet because:

1. It's a CNN, with lots of Dense and Conv2D layers that can be swapped for AM equivalents

2. Despite being advanced for the time, it's old enough that I can train and run it on my GPU in manageable time

3. It's very popular as a framework for testing things like this, so I'm not straying far from existing work

4. Despite the above, the ApproxTrain authors didn't test it in their paper, making this a novel experiment. The AT authors did test it on LeNet, which is comparable in structure but less complex than AlexNet, so this is a useful iteration.

AlexNet was developed to (and did) win the ImageNet classification competition in 2012. I'm training it on CIFAR-10, a much smaller dataset than ImageNet, due to lack of computing power and time.

The structure of AlexNet, which I will make prettier in the actual writeup, is as follows:

1. Input Image: 224x224x3 (width x height x channels)

2. Convolution, 11x11 kernel + 4 stride, ReLU activation: 54x54x96
3. Batch Normalization
4. Pooling, 3x3 max. kernel + 2 stride, 26x26x96
5. Convolution, 5x5 kernel + 2 pad, ReLU: 26x26x256
6. Batch Normalization
7. Pooling, 3x3 max. kernel + 2 stride, 12x12x256
8. Convolution, 3x3 kernel + 1 pad, ReLU: 12x12x384
9. Batch Normalization
10. Convolution, 3x3 kernel + 1 pad, ReLU: 12x12x384
11. Batch Normalization
12. Convolution, 3x3 kernel + 1 pad, ReLU: 12x12x256
13. Batch Normalization
14. Pooling, 3x3 max. kernel + 2 stride: 5x5x256
15. Flatten
16. Dense, 4096 neurons, ReLU
17. Dropout, rate 0.5
18. Dense, 4096 neurons, ReLU
19. Dense, 10 neurons, Softmax

The Convolutional and Dense layers are easily replaced for side-by-side testing. One change- the AlexNet paper mentions a 227x227 input, but I'm using 224x224, since this change seems to be accepted as standard to make the convolutional math work properly, perhaps indicating a mistake in the original work.

I'm choosing not to discuss convolutional networks in detail, because it's not particularly relevant; my goal is just to use this as a real-world example.

I ran into a particularly tricky set of issues running this. It had been some time since I ran an ApproxTrain network, and I was surprised when it didn't run, citing being unable to find a CUDA library, since I hadn't made any changes. More research revealed that CUDA had been updated to version 12 in the intervening time, while my AT version was compiled against CUDA 11. That was no problem; all I had to do was recompile AT. It was after a few more hours of confusing errors that I realized TensorFlow itself doesn't support CUDA 12 yet, at least not in the mainline builds. I tried to build it myself against CUDA 12, as well as downgrade my CUDA version, but had little success with either. I solved the problem with NVIDIA's container runtime, which comes with a TensorFlow binary built against CUDA 12. I ran the docker container, imported my code, rebuilt ApproxTrain, and I was good to go.

First I tested the default multipliers, which took about an hour training to 50 epochs. That converged to ~81% accuracy relatively quickly and smoothly, in about 20 epochs. Afterwards, training accuracy continued to increase, to 98%, while the validation accuracy remained static, indicating significant overfitting.

Then I swapped out the layers for approximate equivalents. I knew it would be slower, but I didn't realize how much- about 20x slower in this case. The total training time was 20h3min. The final accuracy was very similar at 82%, but the training was much less smooth, and occasionally accuracy decreased per-epoch. If I were to test this further I would try different optimizers; I used SGD here because it's what AlexNet specifies, but Adam might produce different results. AMs also started off at a much lower accuracy; 23% accuracy as compared to 51% for default multipliers. I think this might be my fault- I set a global seed for TF, but I am not sure that that means the weights are initialized the same for every type of layer. If I have time, I want to rerun this test with all weights explicitly initialized to zero.

Right now, I have pretty different results from the AT paper. They have AMs improving accuracy at almost the same rate as traditional multipliers. As it is in my results, AMs start off much worse, make quick gains, then slowly converge to the same final accuracy number.

They also display a slightly smaller discrepancy between testing and validation accuracies, which is expected, but I think that discrepancy would increase if I trained it longer. That is to say, I don't think AMs decreased overfitting in this case, I think the model had less time to overfit since it reached peak accuracy much later in training.

I'm going to make and include some nice-looking graphs in the results section.

2023-05-05

I met with Zack about the weirdness of the accuracy plot. Since both models are using the same intializers, there shouldn't be a difference between the plots. Still, there is; I don't think it's about the size of the model, since the AT authors tested smaller ones (LeNet) and showed a unified plot. It doesn't super matter since the end result is the same but it's an interesting open problem.

7.2 Code

This is the Python code for the AlexNet CNN used to test ApproxTrain. [5] formed a syntatic basis for the code, with instructions on how to implement the AlexNet architecture in TensorFlow/Keras. This code was modified to enable ApproxTrain AMs in training and evaluation. The code and modified ApproxTrain is also available on GitHub. Below is the precise version which uses default CUDA multipliers:

```python
import tensorflow as tf
import os
import time
import tensorflow_datasets as tfds
import sys
import keras
#from python.keras.layers.am_convolutional import AMConv2D
#from python.keras.layers.amdenselayer import denseam

# To ensure all training runs get the same starting point
tf.random.set_seed(
    20230414
)

# Load CIFAR10 Dataset
#(ds_train, ds_test), ds_info = tfds.load(
 #        'cifar10',
  #       split=['train', 'test'],
  #       shuffle_files = True,
  #       as_supervised = True,
  #       with_info = True,
  #       )

(train_images, train_labels), (test_images, test_labels) =
↪  keras.datasets.cifar10.load_data()

# Set class names
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
↪  'frog', 'horse', 'ship', 'truck']

def process_images(image, label):
    # Normalize images to have a mean of 0 and standard deviation of
    ↪  1
    image = tf.image.per_image_standardization(image)
    # Resize images from 32x32 to 227x227
    image = tf.image.resize(image, (227,227))
    return image, label

lut_file = './MBM_7.bin'

root_logdir = os.path.join(os.curdir, "logs/fit")
def get_run_logdir():
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)

validation_images, validation_labels = train_images[:5000], train_labels[:5000]
train_images, train_labels = train_images[5000:], train_labels[5000:]

train_ds = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
test_ds = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
validation_ds = tf.data.Dataset.from_tensor_slices((validation_images, validation_labels))

train_ds_size = tf.data.experimental.cardinality(train_ds).numpy()
test_ds_size = tf.data.experimental.cardinality(test_ds).numpy()
validation_ds_size = tf.data.experimental.cardinality(validation_ds).numpy()
print("Training data size:", train_ds_size)
print("Test data size:", test_ds_size)
print("Validation data size:", validation_ds_size)

train_ds = (train_ds\
                  .map(process_images)
                  .shuffle(buffer_size=6000)
                  .batch(batch_size=32, drop_remainder=True))

test_ds = (test_ds\
                  .map(process_images)
                  .shuffle(buffer_size=6000)
                  .batch(batch_size=32, drop_remainder=True))

validation_ds = (validation_ds\
                    .map(process_images)
                    .shuffle(buffer_size=6000)
                    .batch(batch_size=32, drop_remainder=True))

model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11),
     ↪  strides=(4,4), activation='relu', input_shape=(227,227,3)),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1),
     ↪  activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1),
     ↪  activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1),
     ↪  activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1),
     ↪  activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])

model.compile(
    optimizer=tf.keras.optimizers.SGD(0.001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'],
    )

model.fit(
    train_ds,
    epochs=50,
    validation_data = validation_ds,
    validation_freq = 1,
    callbacks=[tensorboard_cb]
)

```

```python
model.evaluate(test_ds)
```

Below is the code which replaces the relevant layers with AM-based ones:

```python
import tensorflow as tf
import os
import time
import tensorflow_datasets as tfds
import sys
import keras
from python.keras.layers.am_convolutional import AMConv2D
from python.keras.layers.amdenselayer import denseam

# To ensure all training runs get the same starting point
tf.random.set_seed(
    20230414
)

#(ds_train, ds_test), ds_info = tfds.load(
 #       'cifar10',
  #      split=['train', 'test'],
  #      shuffle_files = True,
  #      as_supervised = True,
  #      with_info = True,
  #      )

# Load CIFAR10 Dataset
(train_images, train_labels), (test_images, test_labels) =
↪   keras.datasets.cifar10.load_data()

# Set class names
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
↪   'frog', 'horse', 'ship', 'truck']

def process_images(image, label):
    # Normalize images to have a mean of 0 and standard deviation of
    ↪   1
    image = tf.image.per_image_standardization(image)
    # Resize images from 32x32 to 227x227
    image = tf.image.resize(image, (227,227))
    return image, label

```

```
lut_file = './lut/MBM_1.bin'

root_logdir = os.path.join(os.curdir, "logs/fit")
def get_run_logdir():
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir()
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)

validation_images, validation_labels = train_images[:5000],
 ↪  train_labels[:5000]
train_images, train_labels = train_images[5000:], train_labels[5000:]

train_ds = tf.data.Dataset.from_tensor_slices((train_images,
 ↪  train_labels))
test_ds = tf.data.Dataset.from_tensor_slices((test_images,
 ↪  test_labels))
validation_ds = tf.data.Dataset.from_tensor_slices((validation_images,
 ↪  validation_labels))

train_ds_size = tf.data.experimental.cardinality(train_ds).numpy()
test_ds_size = tf.data.experimental.cardinality(test_ds).numpy()
validation_ds_size =
 ↪  tf.data.experimental.cardinality(validation_ds).numpy()
print("Training data size:", train_ds_size)
print("Test data size:", test_ds_size)
print("Validation data size:", validation_ds_size)

train_ds = (train_ds\
                  .map(process_images)
                  .shuffle(buffer_size=6000)
                  .batch(batch_size=32, drop_remainder=True))

test_ds = (test_ds\
                  .map(process_images)
                  .shuffle(buffer_size=6000)
                  .batch(batch_size=32, drop_remainder=True))

validation_ds = (validation_ds\
                  .map(process_images)
                  .shuffle(buffer_size=6000)
                  .batch(batch_size=32, drop_remainder=True))

model = keras.models.Sequential([
    AMConv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape=(227,227,3), mant_mul_lut=lut_file),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    AMConv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="same", mant_mul_lut=lut_file),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    AMConv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same", mant_mul_lut=lut_file),
    keras.layers.BatchNormalization(),
    AMConv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same", mant_mul_lut=lut_file),
    keras.layers.BatchNormalization(),
    AMConv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same", mant_mul_lut=lut_file),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Flatten(),
    denseam(4096, activation='relu', mant_mul_lut=lut_file),
    keras.layers.Dropout(0.5),
    denseam(4096, activation='relu', mant_mul_lut=lut_file),
    keras.layers.Dropout(0.5),
    denseam(10, activation='softmax', mant_mul_lut=lut_file)
])

model.compile(
    optimizer=tf.keras.optimizers.SGD(0.001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'],
    )

model.fit(
    train_ds,
    epochs=50,
    validation_data = validation_ds,
    validation_freq = 1,
    callbacks=[tensorboard_cb]
)

model.evaluate(test_ds)
```

Bibliography

[1] *2021 ROG Zephyrus G14 GA401*. URL: https://rog.asus.com/laptops/rog-zephyrus/2021-rog-zephyrus-g14-series/ (visited on 05/08/2023).

[2] *A Solarpunk Manifesto*. The Anarchist Library. URL: https://theanarchistlibrary.org/library/the-solarpunk-community-a-solarpunk-manifesto (visited on 04/18/2023).

[3] Oludare Isaac Abiodun et al. "State-of-the-art in artificial neural network applications: A survey". In: *Heliyon* 4.11 (Nov. 23, 2018), e00938. ISSN: 2405-8440. DOI: 10.1016/j.heliyon.2018.e00938. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6260436/ (visited on 05/04/2023).

[4] *AI Index Report 2023 – Artificial Intelligence Index*. URL: https://aiindex.stanford.edu/report/ (visited on 04/25/2023).

[5] Richmond Alake. *Implementing AlexNet CNN Architecture Using TensorFlow 2.0+ and Keras*. Medium. Nov. 3, 2021. URL: https://towardsdatascience.com/implementing-alexnet-cnn-architecture-using-tensorflow-2-0-and-keras-2113e090ad98 (visited on 04/17/2023).

[6] Raquel Garrido Alhama. *English: Diagram of a McCulloch-Pitts artificial neuron, extended with a sigmoid activation*. May 1, 2017. URL: https://commons.wikimedia.org/wiki/File:Artificial_Neuron.svg (visited on 04/26/2023).

[7] *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: https://bitcoin.org/en/bitcoin-paper (visited on 04/26/2023).

[8] Tega Brain, Alex Nathanson, and Benedetta Piantella. "Solar Protocol: Exploring Energy-Centered Design". In: *Computing within Limits*. Eighth Workshop on Computing within Limits 2022. LIMITS, June 21, 2022. URL: https://limits.pubpub.org/pub/solar/release/1 (visited on 02/22/2023).

[9] *Cambridge Bitcoin Electricity Consumption Index (CBECI)*. URL: https://ccaf.io/cbeci/ghg/comparisons (visited on 04/05/2023).

[10] Akshay L. Chandra. *McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron*. Medium. Sept. 27, 2022. URL: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1 (visited on 03/14/2023).

[11] Cmglee. *Comparison of the LeNet and AlexNet convolution, pooling, and dense layers*. URL: https://commons.wikimedia.org/wiki/File:Comparison_image_neural_networks.svg (visited on 04/18/2023).

[12] *DaDianNao | Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. URL: https://dl.acm.org/doi/10.1109/MICRO.2014.58 (visited on 05/07/2023).

[13] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009 IEEE Conference on Computer Vision and Pattern Recognition. ISSN: 1063-6919. June 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[14] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. Publication Title: arXiv e-prints ADS Bibcode: 2018arXiv181004805D Type: article. Oct. 1, 2018. DOI: 10.48550/arXiv.1810.04805. URL: https://ui.adsabs.harvard.edu/abs/2018arXiv181004805D (visited on 04/18/2023).

[15] *Docker: Accelerated, Containerized Application Development*. May 10, 2022. URL: https://www.docker.com/ (visited on 05/09/2023).

[16] *Energy-efficient computing*. Main. URL: https://energy.mit.edu/news/energy-efficient-computing/ (visited on 04/26/2023).

[17] Elias Frantar and Dan Alistarh. *SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot*. Publication Title: arXiv e-prints ADS Bibcode: 2023arXiv230100774F Type: article. Jan. 1, 2023. DOI: 10.48550/arXiv.2301.00774. URL: https://ui.adsabs.harvard.edu/abs/2023arXiv230100774F (visited on 05/07/2023).

[18] Xinyang Geng et al. *Koala: A Dialogue Model for Academic Research*. Published: Blog post. Apr. 2023. URL: https://bair.berkeley.edu/blog/2023/04/03/koala/ (visited on 04/03/2023).

[19] Jing Gong et al. *ApproxTrain: Fast Simulation of Approximate Multipliers for DNN Training and Inference*. Publication Title: arXiv e-prints ADS Bibcode: 2022arXiv220904161G Type: article. Sept. 1, 2022. DOI: 10.48550/arXiv.2209.04161. URL: https://ui.adsabs.harvard.edu/abs/2022arXiv220904161G (visited on 02/20/2023).

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: www.deeplearningbook.org.

[21] Corey Gough, Ian Steiner, and Winston A. Saunders. *Energy Efficient Servers: Blueprints for Data Center Optimization*. 1st. USA: Apress, 2015. 360 pp. ISBN: 978-1-4302-6637-2.

[22] *Gradient-based learning applied to document recognition | IEEE Journals & Magazine | IEEE Xplore*. URL: https://ieeexplore.ieee.org/abstract/document/726791 (visited on 05/06/2023).

[23] Issam Hammad and Kamal El-Sankary. "Impact of Approximate Multipliers on VGG Deep Learning Network". In: *IEEE Access* 6 (2018). Conference Name: IEEE Access, pp. 60438–60444. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2875376.

[24] Issam Hammad, Kamal El-Sankary, and Jason Gu. "Deep Learning Training with Simulated Approximate Multipliers". In: *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. Dec. 2019, pp. 47–51. DOI: 10.1109/

ROBIO49542.2019.8961780. arXiv: 2001.00060[cs,eess,stat]. URL: http://arxiv.org/abs/2001.00060 (visited on 05/07/2023).

[25] Song Han et al. *Learning both Weights and Connections for Efficient Neural Networks*. Publication Title: arXiv e-prints ADS Bibcode: 2015arXiv150602626H Type: article. June 1, 2015. DOI: 10.48550/arXiv.1506.02626. URL: https://ui.adsabs.harvard.edu/abs/2015arXiv150602626H (visited on 05/07/2023).

[26] David Harvey and Joris van der Hoeven. "Integer multiplication in time $O(n\mathrm{log}\, n)$". In: *Annals of Mathematics* 193.2 (Mar. 2021). Publisher: Department of Mathematics of Princeton University, pp. 563–617. ISSN: 0003-486X, 1939-8980. DOI: 10.4007/annals.2021.193.2.4. URL: https://projecteuclid.org/journals/annals-of-mathematics/volume-193/issue-2/Integer-multiplication-in-time-Onmathrmlog-n/10.4007/annals.2021.193.2.4.full (visited on 02/22/2023).

[27] Kaiming He et al. *Deep Residual Learning for Image Recognition*. Publication Title: arXiv e-prints ADS Bibcode: 2015arXiv151203385H Type: article. Dec. 1, 2015. DOI: 10.48550/arXiv.1512.03385. URL: https://ui.adsabs.harvard.edu/abs/2015arXiv151203385H (visited on 02/23/2023).

[28] Torsten Hoefler et al. *Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks*. Publication Title: arXiv e-prints ADS Bibcode: 2021arXiv210200554H Type: article. Jan. 1, 2021. DOI: 10.48550/arXiv.2102.00554. URL: https://ui.adsabs.harvard.edu/abs/2021arXiv210200554H (visited on 05/07/2023).

[29] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019). Conference Name: IEEE Std 754-2019 (Revision of IEEE 754-2008), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[30] Shubham Jain et al. "Compensated-DNN: Energy Efficient Low-Precision Deep Neural Networks by Compensating Quantization Errors". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). June 2018, pp. 1–6. DOI: 10.1109/DAC.2018.8465893.

[31] HyunJin Kim. "A low-cost compensated approximate multiplier for Bfloat16 data processing on convolutional neural network inference". In: *ETRI Journal* 43.4 (2021). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.4218/etrij.2020-0370, pp. 684–693. ISSN: 2233-7326. DOI: 10.4218/etrij.2020-0370. URL: https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.2020-0370 (visited on 04/17/2023).

[32] Min Soo Kim et al. "Efficient Mitchell's Approximate Log Multipliers for Convolutional Neural Networks". In: *IEEE Transactions on Computers* 68.5 (May 2019). Conference Name: IEEE Transactions on Computers, pp. 660–675. ISSN: 1557-9956. DOI: 10.1109/TC.2018.2880742.

[33] June Knauth. *AlexNet AM Demos*. original-date: 2023-04-18T17:44:49Z. Apr. 18, 2023. URL: https://github.com/knauth/alexnet-approx-demos (visited on 05/07/2023).

[34] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". In: 2009. URL: https://www.semanticscholar.org/paper/Learning-Multiple-Layers-

of-Features-from-Tiny-Krizhevsky/5d90f06bb70a0a3dced62413346235c02b1aa086 (visited on 05/06/2023).

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (May 24, 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://dl.acm.org/doi/10.1145/3065386 (visited on 04/15/2023).

[36] Tao Lin et al. *Dynamic Model Pruning with Feedback*. Publication Title: arXiv e-prints ADS Bibcode: 2020arXiv200607253L Type: article. June 1, 2020. DOI: 10.48550/arXiv.2006.07253. URL: https://ui.adsabs.harvard.edu/abs/2020arXiv200607253L (visited on 05/07/2023).

[37] *Mark I Perceptron at the Cornell Aeronautical Laboratory*. URL: https://digital.library.cornell.edu/catalog/ss:550351 (visited on 03/14/2023).

[38] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[39] John Martindale. *What Is an ASIC miner?* Digital Trends. Section: Computing. Mar. 16, 2021. URL: https://www.digitaltrends.com/computing/what-is-an-asic-miner/ (visited on 04/19/2023).

[40] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (Dec. 1, 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259 (visited on 03/08/2023).

[41] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. 1969. DOI: 10.7551/mitpress/11301.001.0001. URL: https://direct.mit.edu/books/book/3132/PerceptronsAn-Introduction-to-Computational (visited on 03/14/2023).

[42] John N. Mitchell. "Computer Multiplication and Division Using Binary Logarithms". In: *IRE Transactions on Electronic Computers* EC-11.4 (Aug. 1962). Conference Name: IRE Transactions on Electronic Computers, pp. 512–517. ISSN: 0367-9950. DOI: 10.1109/TEC.1962.5219391.

[43] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". In: *ACM Computing Surveys* 48.4 (Mar. 18, 2016), 62:1–62:33. ISSN: 0360-0300. DOI: 10.1145/2893356. URL: https://dl.acm.org/doi/10.1145/2893356 (visited on 05/06/2023).

[44] John Nickolls et al. "Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for?" In: *Queue* 6.2 (Mar. 1, 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. URL: https://dl.acm.org/doi/10.1145/1365490.1365500 (visited on 05/06/2023).

[45] Hyeonwoo Noh et al. "Regularizing Deep Neural Networks by Noise: Its Interpretation and Optimization". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/hash/217e342fc01668b10cb1188d40d3370e-Abstract.html (visited on 02/21/2023).

[46] *November 2022 | TOP500*. URL: https://www.top500.org/lists/green500/2022/11/ (visited on 04/19/2023).

[47] *NVIDIA Container Runtime*. original-date: 2017-09-05T22:03:28Z. May 4, 2023. URL: https://github.com/NVIDIA/nvidia-container-runtime (visited on 05/09/2023).

[48] Offnfopt. *English: A neural network with multiple layers.* Apr. 12, 2015. URL: https://commons.wikimedia.org/wiki/File:Multi-Layer_Neural_Network-Vector-Blank.svg (visited on 04/26/2023).

[49] OpenAI. *GPT-4 Technical Report*. Publication Title: arXiv e-prints ADS Bibcode: 2023arXiv230308774O Type: article. Mar. 1, 2023. DOI: 10.48550/arXiv.2303.08774. URL: https://ui.adsabs.harvard.edu/abs/2023arXiv230308774O (visited on 04/14/2023).

[50] *OpenAI launches an API to commercialize its research*. VentureBeat. June 11, 2020. URL: https://venturebeat.com/ai/openai-launches-an-api-to-commercialize-its-research/ (visited on 04/18/2023).

[51] *OpenAI's GPT-3 Language Model: A Technical Overview*. June 3, 2020. URL: https://lambdalabs.com/blog/demystifying-gpt-3 (visited on 04/18/2023).

[52] Oliver Peckham. *Nvidia's H100 Debuts in 'Henri,' Topping the Green500 List*. HPCwire. Nov. 15, 2022. URL: https://www.hpcwire.com/2022/11/14/nvidias-h100-debuts-in-henri-topping-the-green500-list/ (visited on 04/26/2023).

[53] *Planning for AGI and beyond*. URL: https://openai.com/blog/planning-for-agi-and-beyond#SamAltman (visited on 04/25/2023).

[54] *Pulse of Fintech - KPMG Global*. KPMG. Apr. 14, 2023. URL: https://kpmg.com/xx/en/home/industries/financial-services/pulse-of-fintech.html (visited on 04/26/2023).

[55] Rohan Taori et al. *Alpaca: A Strong, Replicable Instruction-Following Model*. Mar. 13, 2023. URL: https://crfm.stanford.edu/2023/03/13/alpaca.html (visited on 05/07/2023).

[56] F. Rosenblatt. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Google-Books-ID: P_XGPgAACAAJ. Cornell Aeronautical Laboratory, 1957. book.

[57] Rustie98. *English: The whole die of Am386 DX-40, stacked in Photoshop from Light-Field microscopy images. A Bausch & Lomb MicroZoom Light-Field microscope was used.* Mar. 29, 2023. URL: https://commons.wikimedia.org/wiki/File:Stacked_CPU_die_up_close.png (visited on 04/26/2023).

[58] Hassaan Saadat, Haseeb Bokhari, and Sri Parameswaran. "Minimally Biased Multipliers for Approximate Integer and Floating-Point Multiplication". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (Nov. 2018). Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 2623–2635. ISSN: 1937-4151. DOI: 10.1109/TCAD.2018.2857262.

[59] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. *Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model*. Publication Title: arXiv e-prints ADS Bibcode: 2022arXiv221102001S Type: article. Nov. 1, 2022. DOI: 10.48550/arXiv.2211.02001. URL: https://ui.adsabs.harvard.edu/abs/2022arXiv221102001S (visited on 04/25/2023).

[60] Kenta Shirane, Takahiro Yamamoto, and Hiroyuki Tomiyama. "A design methodology for approximate multipliers in convolutional neural networks: A case of MNIST". In: *International Journal of Reconfigurable and Embedded Systems (IJRES)*. Vol. 10. ISSN: 2722-2608, 2089-4864 Issue: 1 Journal Abbreviation: IJRES. Mar. 1, 2021, p. 1. DOI: 10.11591/ijres.v10.i1.pp1-10. URL: http://ijres.iaescore.com/index.php/IJRES/article/view/20306 (visited on 04/17/2023).

[61] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6.1 (Dec. 2019). Number: 1 Publisher: SpringerOpen, pp. 1–48. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0. URL: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0197-0 (visited on 05/07/2023).

[62] Charles Stangor and Jennifer Walinga. *Introduction to Psychology - 1st Canadian Edition*. BCcampus, Oct. 17, 2014. ISBN: 978-1-77420-005-6. URL: https://opentextbc.ca/introductiontopsychology/ (visited on 05/04/2023).

[63] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. ACL 2019. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 3645–3650. DOI: 10.18653/v1/P19-1355. URL: https://aclanthology.org/P19-1355 (visited on 04/26/2023).

[64] *TFApprox: Towards a Fast Emulation of DNN Approximate Hardware Accelerators on GPU*. DeepAI. Feb. 21, 2020. URL: https://deepai.org/publication/tfapprox-towards-a-fast-emulation-of-dnn-approximate-hardware-accelerators-on-gpu (visited on 02/20/2023).

[65] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. Publication Title: arXiv e-prints ADS Bibcode: 2023arXiv230213971T Type: article. Feb. 1, 2023. DOI: 10.48550/arXiv.2302.13971. URL: https://ui.adsabs.harvard.edu/abs/2023arXiv230213971T (visited on 04/14/2023).

[66] *Ubuntu 18.04.6 LTS (Bionic Beaver)*. URL: https://releases.ubuntu.com/18.04/ (visited on 05/09/2023).

[67] Ashish Vaswani et al. *Attention Is All You Need*. Publication Title: arXiv e-prints ADS Bibcode: 2017arXiv170603762V Type: article. June 1, 2017. DOI: 10.48550/arXiv.1706.03762. URL: https://ui.adsabs.harvard.edu/abs/2017arXiv170603762V (visited on 02/22/2023).

[68] *What is page load time and why is it important?* BigCommerce. URL: https://www.bigcommerce.com/ecommerce-answers/what-page-load-time-and-why-it-important/ (visited on 04/26/2023).

[69] Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. *Discovering Neural Wirings*. Publication Title: arXiv e-prints ADS Bibcode: 2019arXiv190600586W Type: article. June 1, 2019. DOI: 10.48550/arXiv.1906.00586. URL: https://ui.adsabs.harvard.edu/abs/2019arXiv190600586W (visited on 05/07/2023).

[70] Zerodamage. *An illustration of the gradient descent method. I graphed this with Matlab*. Aug. 7, 2012. URL: https://commons.wikimedia.org/wiki/File:Gradient_descent.svg (visited on 05/04/2023).