Bedot: Bit Efficient Dot Product for Deep Generative Models

Nhut-Minh Ho[1], Duy-Thanh Nguyen[2], John L. Gustafson[3], and Weng-Fai Wong[1]

[1] National University of Singapore
{minhhn,wongwf}@comp.nus.edu.sg
[2] Kyung Hee University
dtnguyen@khu.ac.kr
[3] Arizona State University
jlgusta6@asu.edu

Abstract. This paper presents an optimization method to build the smallest possible integer mapping unit that can replace a conventional multiply-and-accumulate unit in deep learning applications. The unit is built using a hardware-software co-design strategy that minimizes the set of represented real values and energy consumed. We target larger and more complex deep learning applications domains than those explored in previous related works, namely generative models for image and text content. Our key result is that using our proposed method, we can produce a set as small as 4 entries for an image enhancement application, and 16–32 entries for the GPT2 model, all with minimal loss of quality. Experimental results show that a hardware accelerator designed using our approach can reduce the processing time up to 1.98×/3.62× and reduce computation energy consumed up to 1.7×/8.4× compared to 8-bit integer/16-bit floating-point alternatives, respectively.

Keywords: Number format · Deep learning · Generative Adversarial Networks · Generative Models · Energy Efficient Machine Learning

1 Introduction

Deep learning has permeated a wide variety of computer applications ranging from image recognition to the more creative uses of *generative adversarial networks* (GAN) and generative models in general to produce realistic artifacts like images and text. Performing inference for these models as efficiently as possible is crucial to promoting their wider adoption, for example, on edge devices. For example, low quality video can be upscaled to higher quality at the monitor end using deep learning based upscaling. To this end, compressing large models by reducing the number of bits used to represent model parameters has garnered considerable interest. This technique has been frequently used for more popular networks such as *convolutional neural networks* (CNN). However, applying it to more recent networks such as OpenAI's GPT models for language modelling as well as GANs is not straightforward. While CNN inference can be done with a very few number

of bits, the state-of-the-art for language models' bit requirements remain at 8 bits [1]. Similarly, while quantization mechanisms have been marginally explored for GANs, they are inefficient, require re-training, and are only able to represent weights [2]. Moreover, a systematic method of obtaining the quantization levels can be beneficial for extending the technique to even newer models. In this paper, we apply a new multiply-accumulate (MAC) approach for the emerging neural networks (the deep generative models) for which the output and error metrics are vastly different from the most popular classification tasks in the literature.

Depending on the number of bits required to represent the model, different types of hardware can be used. Traditional MAC units with accumulators of various sizes have been used in many studies to date. Approaches that use lookup tables have been scarcely studied due to the complexities involved with large table sizes that increase the look-up time. However, when the number of bits needed are significantly reduced, custom tables can be used to speed-up execution. Furthermore, by implementing efficient lookup mechanisms, the circuitry required for traditional MACs can also be replaced by integer mappings, bringing about substantial energy savings.

Our contributions in this work are as follows.

- We reduce the number of bits required to 2–3 bits for weights and 2–5 bits for activation on large generative models such as GPT2 using our optimizing algorithm.
- We implement a new, highly efficient MAC method that uses integer summation of mapped inputs.
- We propose an architecture to perform exact dot products with much higher performance and energy efficiency than fixed-point or floating-point architectures.

2 Background and Related work

2.1 Neural networks and emerging architectures

Deep neural networks (DNNs) have recently achieved enormous success in a variety of domains, such as image classification, object detection, image segmentation, language, and media-related tasks. Initial efforts focused primarily on classification and detection. More recently, research interest has shifted to designing neural networks to address an even more complex challenge: *content generation*. Content generation tasks range from simple image style transfer, colorization, and super resolution to generating deepfake videos, synthetic voice and text which utilize GANs. In GANs, the input is either a random vector (for random data generation) or an original image (for style transfer or image enhancement). Graphic output introduces another challenge for low-precision approaches; in contrast with image classification where the output are only a vector of N values for N classes being classified, the outputs of GANs are image pixels. For example, an image of size $512 \times 512 \times 3$ needs all 786432 values to be in the output layers instead of 1000

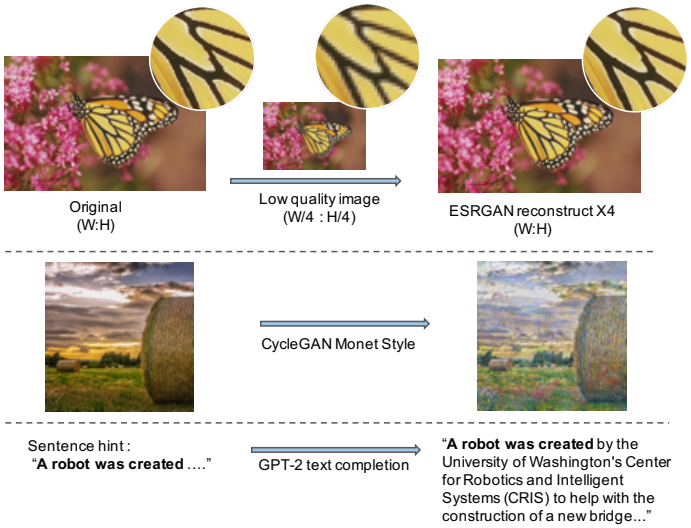

Fig. 1. Overview of some generative models used in this paper

real values in the output layer of ImageNet classification DNNs. Pixel values also need to remain in high precision to construct output images, e.g. 8 bits per channel for high quality color output. Figure 1 shows some samples of the tasks that will be optimized in this paper.

Previous work on quantization of Convolutional Neural Networks (CNNs) for image classification can reduce the required precision of parameters down to 4–8 bits with minimal loss of accuracy, with the help of quantization-aware training. However, the current retraining method is difficult to apply since training data is not available in some models (training settings for GPT2 and GPT-3 are proprietary). Another obstacle of this approach is the cost and time associated with training. Millions of dollars are required to train the latest GPT models. They are trained with non-disclosed training data to be a general model for various language-related tasks. For other GANs, fine-tuning methods are still not easy and training is difficult to converge [3]. Thus, in this paper, we present a method to optimize both weights and activations of these models without fine-tuning. The techniques of fine-tuning to enhance the output quality can always be applied on top of our method.

2.2 Quantization and Table Lookup

Two predominant approaches exist to reduce the bitwidths of parameters for low-precision neural network inference: Applying quantization functions to high-precision values, and looking up corresponding low-precision values from a table. Several quantization methods have been developed; for simplicity, we consider the *uniform quantization* method which is widely used in frameworks with 8-bit

tensor instructions since the actual values and the quantized values are convertible using a scale s.

Consider the quantization operation of mapping 32-bit floating-point (FP32) values to 8-bit integer (INT8) values. Quantization in this case involves selecting the range for the quantized values and defining functions that will map the FP32 values to the closest INT8 value and back (quantize and dequantize). If the selected range is $[\alpha, \beta]$, then uniform quantization takes an FP32 value, $x \in [\alpha, \beta]$ and maps it to a value in $[0, 255]$ or $[-128, 127]$. Values outside of $[\alpha, \beta]$ are clipped to the nearest bound. There are two possible uniform quantization functions: $f(x) = s \cdot x + z$ (affine quantization) or $f(x) = s \cdot x$ (scale quantization) where $s, x, z \in R$; s is the scale factor by which x will be multiplied, and z is the *zero-point* to which the zero value of the FP32 values will be mapped in the INT8 range. The uniform scale quantization is common when deploying to INT8 tensor hardware because of its lower dequantization overhead [4]. Let s_1 and s_2 be the scales used to quantize weight W and activation A of a dot product operation ($\otimes$). The scale quantized dot product result R' can be dequantized by using the appropriate factor:

$$R' = W' \otimes A' = \sum_{1}^{K} w_i \times s_1 \times a_i \times s_2$$

$$R = \sum_{1}^{K} w_i \times a_i = R'/(s_1 \times s_2)$$

For our new MAC approach, we allow the numerical inputs to be **anywhere** in $\mathbb{R}$. Instead of requiring N-bit integer multiplier logic, we use a low-precision integer form of logarithm and antilogarithm mappings that *need not produce rounding error*. Instead of filling out a two-dimensional multiplication table of size 2^{N+M} where M is the number of w_i values and N is the number of a_i values, we map inputs to low-precision unsigned integers that are added and the resulting sum mapped to a (potentially exact) fixed-point product that can be accumulated. This eliminates "decoding" of a bit format into sign, exponent, and fraction fields, and any need for an integer multiplier in the MAC unit. In this paper we show that by allowing the precision of the fixed-point product to be 10–16 bits and optimizing the "vocabulary" of input real values, the sizes of M and N can be as low as 2 or 3 bits.

There are four main enhancements in Bedot compared to conventional INT8 quantization in current general-purpose tensor core architectures (see Figure 2):

1. Activation of the previous layers are kept at fixed point while the weights are kept at as small as 3-bit pointers to our carefully chosen real-valued weights.
2. The product of two operands is obtained by integer mapping instead of traditional multiplier hardware.
3. The accumulator is smaller than 32 bits, and optimized to preserve dot product correctness (no rounding).

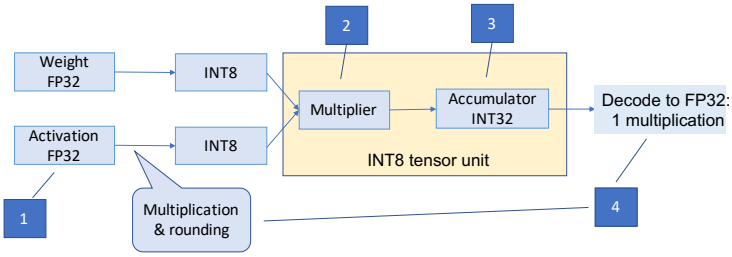

Fig. 2. A conventional integer quantization architecture for inference and the main differences in Bedot

4. We introduce *rounding hints*. This enhancement yields additional tuning knobs for improving output quality compared to the conventional round-to-nearest.

This motivates the first part of our paper to search for the smallest number set (vocabulary) possible while maintaining high output quality. The search results will be used for customized hardware design with the details in Section 5.

2.3 Related work

Due to the high energy consumption of single precision floating point arithmentic, there have been many proposals to use lower precision arithmetic such as fixed point [5,6,7], half precision and mixed precision [8,9,10,11], posit [12,13] for general applications. In the specific domain of deep learning, there have been several proposals to optimize both inference and training phases. Our work is specific to inference optimization. This method can be considered as the last-step optimization after all other optimizations have been applied (e.g. network pruning, retraining, fine-tuning). Regardless of how the model was trained and optimized, our technique produces a high-efficiency model that is ready to deploy.

There are several approaches to quantizing network models to lower bitwidths, mainly in the area of image classification with well organized surveys in [14,15]. Notably, recent works have explored hardware-friendly nonuniform quantization methods with the restriction being that quantized values are additive power-of-two [16]. More recently, next generation arithmetic such as Posit plays an important role in neural network training and inference [12,17,18]. However, most of these methods are not directly applicable to more complex generative models such as GPT and GANs. Most analytical methods for quantization target the convolutional layer and rely on RELU activation and a softmax layer when deriving cross-layer relationships and error formulae [19,20]. Unfortunately, activation and network architectures vary greatly in recent systems. INT8 quantization has been used for emerging neural networks include transformer architectures [1,21]. The current standard method for quantization of deep generative models that can be easily reproduced is to use INT8 inference module on Pytorch and Nvidia TensorRT. These frameworks provide standard methods for quantizing image

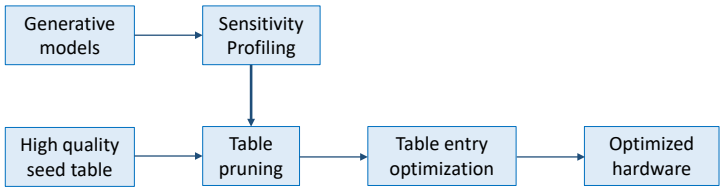

Fig. 3. Workflow to produce a Bedot-ready model.

classification models, with support for language and generative models in experimental mode. To the best of our knowledge, we are the first to attempt ultra-low-bitwidth tuning of the GPT2 model.

On the hardware design front, Eyeriss [22] is a systolic array system for CNNs. [23] proposes a systolic array engine using table lookup instead of a multiplier. Their table lookup unit uses 4-bit addresses. To ensure high output quality, their architecture also requires higher precision (8–16 bit) data. Most designs focus on CNNs for image classification or normal applications. Current widely-used hardware for generative model inference is still GPGPU-based. Our work uses the multiply-accumulate design but with very low bitwidth. It is possible because our software module detects accuracy-sensitive layers so the rest of the layers can use 2–3 bit inputs for most models.

3 Overview of Bedot

Our work consist of two components working in synergy: software to find the smallest possible real number set that preserves answer quality, and hardware based on those sets. Given a neural network and a specified output quality, our software framework runs inference many times to find the smallest set of real values for w_i and a_i that works and optimize the set entries using our proposed algorithm.

The set is then used in the hardware module to speed up the actual inference of the network for low energy environments (e.g. edge devices). Because we aggressively reduce the set size in our software module, the MAC task becomes simple integer-to-integer mapping that can be implemented using combinational logic instead of addressed ROM. This yields a massive performance and efficiency gain compared to both conventional low-precision units and table lookup approaches for neural network inference. In the trade-off for energy gain, we have to sacrifice the flexibility of the hardware module. Our target application is edge devices designed for a limited number of applications. For example, in image super resolution for TVs, the video can be transferred using low quality to conserve bandwidth and the higher resolution video can be reconstructed efficiently by our module for display.

4 Set Construction

4.1 Detecting the sensitive layers

Generating the minimal real number sets that preserve quality is the main focus of our work. This is particularly difficult given that the applications we target are diverse, large, and complex. We need to address *error sensitivity*. For example, quantization pixel color channel of sensitive GAN's layer to below 8 bits will likely result in a loss of information in the input image and incorrect colors in the output. Typically, reduced-precision designs use higher precision for the first and last layers of a CNN. Although this heuristics can be applied to generative models, we found that more layers needed to be excluded to preserve quality than conventional CNNs. We apply a low-precision format (e.g. floating-point 8-bit [24]) to each layer in turn, while maintaining the rest at full 32-bit precision, and then measuring the output quality to quantify the sensitivity of a layer to precision reduction. Using this method, we found that the number of high-sensitive layers varies among models.

For image generative models, the *structural similarity index measure* (SSIM) identifies sensitive outliers; a 1% drop of SSIM is the outlier threshold. For GPT2, we use *perplexity* (PPL) as the quality metric since it is widely used for text models. It is difficult to tell what amount of increase in PPL is "too high." Thus, for GPT2 we use an outlier detection method, the Boxplot method using *inter-quartile range* (IQR) [25,26]. Let $Q1$ and $Q3$ be the first and third quartiles of the data (25th and 75th percentiles). The $\text{IQR} = Q3 - Q1$ is computed and thresholds $Q3 + 3 \times \text{IQR}$ and $Q1 - 3 \times \text{IQR}$ are used to detect extreme sensitive outliers. As a result, we exclude three layers ($17^{\text{th}}, 141^{\text{th}}$, and 145^{th}) out of the 145 layers in GPT2 model. For ESRGAN, the first layer and the last five layers were excluded. For other models, two layers are excluded (the first and last layers). We use fixed-point 16-bit for these excluded layers.

4.2 Building optimized real number sets

In the next step, we construct a single set of real magnitudes for both positive and negative numbers, to reduce hardware cost. The sign bit of the product is simply the XOR of the two input sign bits. We start with a large set that yields high output quality (e.g. FP16 [27]). However, because the set of possible 16-bit values is large, building a smaller set from such large set by testing every possible value is very time-consuming. Instead, we use smaller sets of numbers that are known to be good for inference. For this purpose, we chose the best among posit [28,29,18] and [24] floating-point formats (64–128 unsigned entries) as the starting sets. We simply try different configurations with 7 or 8 bits, and choose the configuration with the fewest bits that yields acceptable quality (which in our experiment means 'less than 1% quality drop'). For the rest of this paper, the set represents magnitudes and will be used along with the sign bit of the original value for dot product computation.

4.3 Fast derivation of the small set from a seed set

After obtaining a seed set yielding good quality, we turn to reducing its size. Our algorithm halves the set size in rounds. From 128 unsigned entries, to find a set with only eight entries, we need to perform four rounds: $128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8$. At each step, we loop over the L entries of the set, record the output quality of the neural network after removing *each* entry. We then sort the output quality array to find the $L/2$ entries with the *least impact* on output accuracy. We remove these from the set and repeat the process. Assuming we want to reduce set size from 2^A to 2^B, $B < A$, the algorithm stops after $A - B$ iterations. The threshold ε is chosen based on our desired output quality. Since output quality will be improved in the next step by our set optimization algorithm, we actually select ε *below* the desired threshold. Based on our experiments, the search algorithm can improve accuracy up to 10%, thus we can pick ε about 10% worse quality than desired before running the optimization algorithm in the next Section.

Note that, removing the entries one by one (like a Jenga game) instead of deleting half of the set in line 13 will give a slightly better set with the same number of entries. However, the advantage is minimal when applying our optimizing algorithm afterwards while incurring huge additional overhead (from $A - B$ iterations to $2^{(A-B)}$ iterations). Thus, we chose to execute this phase batch by batch.

4.4 Enhancing output quality

The next step iteratively tunes each *value* in the set to improve output quality. The pseudocode can be seen in Algorithm 1. Each iteration measures the effect of changing each entry by δ in both directions. Let $O'[i]$ be the current value of an entry, the algorithm tests the result quality for $O'[i] \pm \delta \cdot O'[i]$. The algorithm then picks the best candidate (lines 21–28). If the best candidate improves the results compared to the previous iteration, the original set value is replaced by the best candidate and the algorithm continues with the same rate of change (δ). If all candidates worsen output quality, we reduce δ by half. The δ value will thus decrease rapidly and eventually converge when δ becomes too small ($< 1\%$ change). Note that, in Algorithm 1, there is only one set to be optimized. We concatenate the weight and activation sets into a single set to optimize.

The *Test*(O') function in lines 11 and 17 in Algorithm 1 is where we assess the output quality of the current set (O') with a separate representative dataset, different from that used to optimize set entries. The representative dataset is different for each model. If the model has multiple recommended evaluation datasets, we pick one for tuning and test against all others. For example, for ESRGAN, we pick Set14 for our tuning process but at the end of the optimization process measure the output quality of another test set (Set5). Both results will be presented in our Section 6. The result of *Test*(O') is in form of the metric for measuring output quality. The default setting in Algorithm 1 assumes the higher metric indicates better results. For models with the lower-better metric (i.e. Perplexity of GPT2), we simply pass $-$*Test*(O') to line 11 and 17 and other

Algorithm 1 Algorithm for optimizing set entries

1: **Input** Table I : $[I_1 \dots I_N]$ // Input unoptimized set entries, sorted in ascending order
2: **Output** Table O : $[O_1 \dots O_N]$
3: $O \leftarrow I$
4: $\delta \leftarrow 0.5$ // Initialize changing rate, 50% of each table entry's value
5: $Q_{curr} = Test(O)$ // Initialize the table
6: **while** $\delta >= 0.01$ **do**
7: $Inc \leftarrow [0 \dots 0]$ // Store output quality when increasing table entries
8: **for** $i = 1$ to N **do**
9: $O' \leftarrow copy(O)$
10: $O'[i] \leftarrow O'[i] + O'[i] \times \delta$
11: $Inc[i] \leftarrow Test(O')$
12: **end for**
13: $Dec \leftarrow [0 \dots 0]$ // Store output quality when decreasing table entries
14: **for** $i = 1$ to N **do**
15: $O' \leftarrow copy(O)$
16: $O'[i] \leftarrow O'[i] - O'[i] \times \delta$
17: $Dec[i] \leftarrow Test(O')$
18: **end for**
19: $Inc\_idx \leftarrow \max(Inc)$ // Get the index of the entry having maximum output quality in **Inc**
20: $Dec\_idx \leftarrow \max(Dec)$ // Get the index of the entry having maximum output quality in **Dec**
21: **if** $\max(\max(Inc), \max(Dec)) > Q_{curr}$ **then**
22: **if** $\max(Inc) > \max(Dec)$ **then**
23: $O[Inc\_idx] \leftarrow O[Inc\_idx] + O[Inc\_idx] \times \delta$ // Increase the corresponding table entry
24: $Q_{curr} = \max(Inc)$
25: **else**
26: $O[Dec\_idx] \leftarrow O[Dec\_idx] - O[Dec\_idx] \times \delta$ // Decrease the corresponding table entry
27: $Q_{curr} = \max(Dec)$
28: **end if**
29: **else**
30: Quality not improved, decrease changing rate by half
31: $\delta \leftarrow \delta/2$
32: **end if**
33: **end while**

parts of the algorithm can remain the same. The complexity of this algorithm is low because halving the changing rate δ converges quickly. For the longest experiment we ran (GPT2), it cost $33 + 6$ iterations as demonstrated in Figure 4 (6 more iterations where the condition in line 30 is met, δ is decreased).

4.5 The rounding hint table and its effect

There are several ways to convert any real value to one of our set's entries. Initially, we use round-to-nearest. Consider a value x and the set $O[O_1..O_N]$ with N entries from Algorithm 1, sorted in ascending order. Let $M[M_1..M_N]$ be the set of midpoints. The round-to-nearest mode can be implemented by comparing the absolute value $|x|$ against the midpoints and taking the index of the entry with its midpoint just below $|x|$:

$$M[i] = \begin{cases} (O[i] + 0)/2 & \text{if } i = 1 \\ (O[i] + O[i-1])/2 & \text{if } i > 1 \end{cases}$$

$$\text{Encode}(x) = i \text{ where } \max_i M[i] \leq |x| \tag{1}$$

During our experiments, we realized these comparison points can be *tuned*. Better "midpoints" can improve the output quality of the model with no additional hardware overhead. Naive round to nearest requires a comparison unit. In

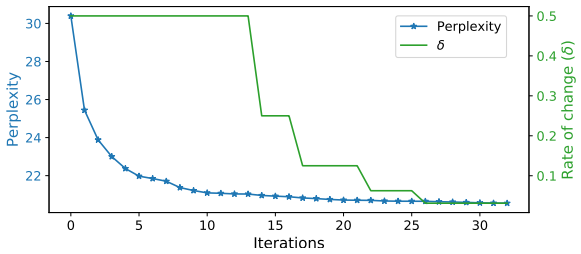

Fig. 4. Sample run of Algorithm 1 on GPT2. A lower perplexity metric on the left y-axis indicates better output.

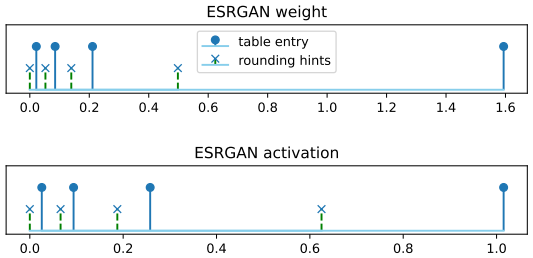

Fig. 5. Tuning of rounding hints for ESRGAN

rounding in the range $[O[i], O[i-1]]$, we can replace the midpoint with any real value $M[i]$ (a *rounding hint*) that is in the same range:

$$M[i] = \begin{cases} 0 \le M[i] \le O[i] & \text{if } i = 1 \\ O[i-1] \le M[i] \le O[i] & \text{if } i > 1 \end{cases}$$

Note that in the actual implementation, the index will begin with 0 instead of 1. The initial rounding hints table is set to be mid-points (round to nearest mode) and concatenated with the set entries. The whole table will be used in Algorithm 1 for optimization. To support efficient hardware implementation, the optimized rounding hints are rounded to a fixed-point format with lower bitwidth with the hardware design in Section 5.5. Figure 5 shows set entries and rounding hints after optimizing the ESRGAN model. The effect of rounding hints will be presented in Section 6.

5 Set Mapping Based Inference

5.1 Processing element utilizing table lookup

The number of weight and activation entries is reduced significantly by our algorithms. We can use a "wired ROM" to create a MAC based on the integer mappings. In this paper, we introduce and evaluate the wired ROM unit implemented with combinational logic, and show that the design has smaller area,

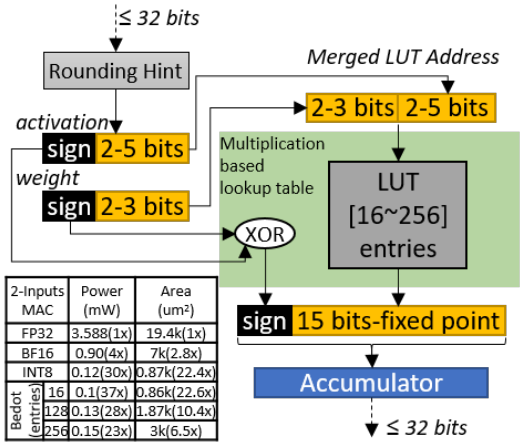

Fig. 6. Set mapping unit to find the product of weight and activation, and comparison of the power and area size of a two-input MAC for several input formats.

lower cost, and higher speed per watt than MAC hardware based on integer multipliers or addressed lookup tables.

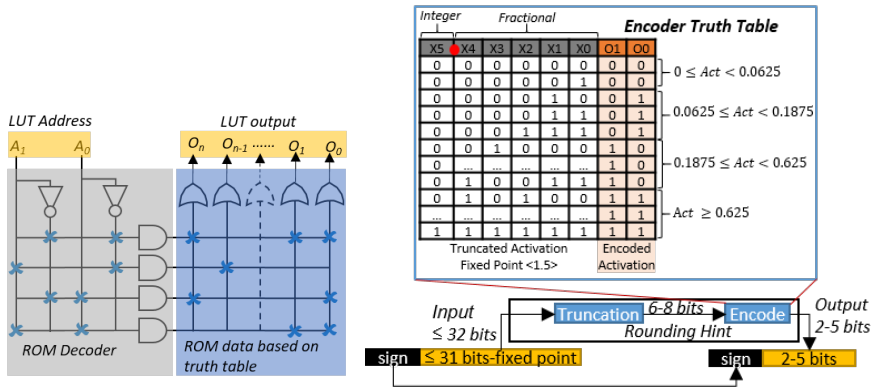

Fig. 7. An example design of our lookup unit with 2-bit inputs and example Rounding Hint module of ESRGAN

5.2 Wired ROM logic for multiplication

For each neural network, after we obtain the optimized values from Algorithm 1, conventional table lookup units would decode the weights and activations to fixed-point numbers before multiplication. However, we pre-compute the exact Cartesian products of the weights and activations at full precision and find integer mappings with the same Cartesian sum ranking as the Cartesian products, like

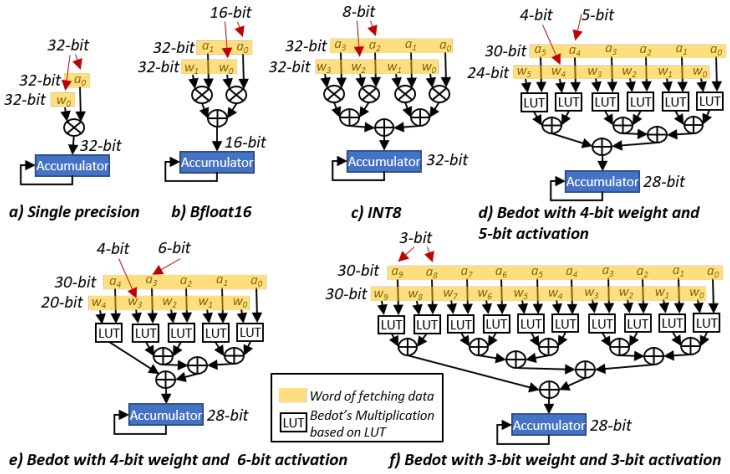

Fig. 8. 32-bit SIMD MAC architecture to compute dot product of 2 N-element vectors

a low-precision logarithm table. Analogously, the "inverse logarithm" can then be a rounding-free map to the fixed-point product as shown in Figure 6. For inference, a table lookup using the combined weight and activation bits yields the exact fixed-point product. The result is accumulated exactly in fixed-point with just enough precision to protect against underflow and overflow. We shall call the ROM-based unit a *lookup unit* (LU) in our design.

5.3 Combinational logic implementation of ROM

We can derive basic logic gates to map a pair (w, a) to their product represented in fixed-point. For example, we can build a "wired ROM" with two 2-bit inputs and n output bits as shown in Figure 7. The circuit needs only NOT, AND, and OR gates, and the rest are wires with crosses indicating a connection. In contrast with other circuit ROMs, we can simply deploy it into reconfigurable designs such as an FPGA. The main components in combinational logic ROM are 1) logic decoder, and 2) ROM data. For each possible input, the ROM decoder will produce the required bit strings with only a few logic delays. The bit strings are the integer "log tables" (Section 5.2). This approach has very high energy efficiency compared to FP32, BF16, and the state-of-the-art INT8 inference. For our MAC unit, we estimate the energy using 45 nm CMOS at 200 MHz. Our ROM designs with 16/128/256 entries achieve significantly higher speed and lower power consumption than INT8, FP32 and BF16, as shown in Figure 6.

5.4 SIMD MAC unit utilizing our lookup units

With the LU as the multiplier, we can design a dot product engine to perform the dominant operation in neural networks. Besides the dot product, simpler

Design	Power, mW	Dot-product Speedup	GOPs/W (improvement)	Area μm^2 (reduction)
FP32	3.80	1×	0.11 (1×)	19.4k (1×)
BF16	1.49	2×	0.53 (4.81×)	12.48k (1.56×)
INT8	0.57	4×	2.81 (25.18×)	3.2k (6×)
Bedot 16 entries	0.79	10×	5.04 (45.14×)	4.3k (4.5×)
Bedot 128 entries	0.67	6×	3.55 (31.8×)	8.7k (2.2×)
Bedot 256 entries	0.68	5×	2.93 (26.27×)	12.51k (1.55×)

Table 1. Normalized GOPs/W and area reduction of 32-bit SIMD MAC design for each design in Figure 8 for 45 nm CMOS at 200 MHz. FP32 is used as baseline.

operations such as scaling, pooling, and averaging can be performed using fixed-point units. We focus only on the dot product engine in this paper since it accounts for $> 95\%$ of the energy. Note that any activation function can be implemented with *zero cost* by incorporating it into the rounding hints step.

Each LU multiplies a weight by an activation. A conventional *floating-point multiply-accumulate* (FMA) instruction requires fetching 32-bit FP32 operands. Each such fetch is equivalent to fetching multiple inputs in parallel if we use BF16, INT8, or our Bedot design instead. As shown in Figure 8, with different precisions for weights and activations, we can speed up the dot product by 2× to 10× compared to a 32-bit FMA because of this parallelism. As mentioned in Section 5.4, our combinational logic ROMs consume the same power as a 2-input MAC design; however, Bedot (Figure 8) is more energy efficient (GOPs/W) than other formats because of the lower number of bits, as shown in Table 1.

5.5 The implementation of rounding hints

Encoded weights are pre-computed, but activations must be encoded back to 2–5 bits at runtime via rounding. Rounding hints can be implemented by a few logic gates found by Karnaugh maps (K-map). For example, the set of rounding hints is {0, 0.0625, 0.1875, 0.625} for activation for the ESRGAN applications and it is applied to the encoded two-bit activation set. For this set, we only need a $(1, 5)$ (1-bit integer, 5-bit fraction) fixed-point number to represent the rounding hint values, as shown in Figure. 7. Hence, we only need to truncate the raw activations to this $(1, 5)$-fixed-point. We then simply solve the K-map problems to provide the encoder logic circuit for activation data. As a result, the rounding hint can be implemented using just a few logic gates (14 AND/OR gates), negligible compared to the MAC.

6 Experiments

In this section, we provide the experimental results of our software and hardware modules. For the software module, we tested different applications from text

generation to style transfer and image super resolution. For the hardware module, we estimate the 32-bit MAC design specs as discussed in Section 5.4 to evaluate Bedot.

Model Name	ESRGAN		Horse to Zebra		Van Gogh Style		Monet Style		GPT2-large		
Dataset	Set14	Set5	h2z	h2z	v2p	v2p	m2p	m2p	Wiki-2 (19.1)	Wiki-2 (19.44)	Wiki-103 (19.09)
Bedot	0.932/30.9	0.925/31.9	0.928/28.3	0.928/28.1	0.930/27.9	0.928/27.9	0.947/30.8	0.945/30.4	20.552	20.969	20.539
Bedot+H	0.954/33.2	0.956/34.7	**0.938**/29.9	**0.938**/29.9	**0.935**/28.4	**0.932**/28.4	**0.951**/31.1	**0.949**/30.4	20.435	20.828	20.367
INT8	**0.975**/34.5	**0.987**/40.7	0.726/24.4	0.645/22.6	0.773/21.3	0.775/21.4	0.627/19.7	0.623/19.3	**20.229**	**20.629**	**20.235**
Metric(s)	S/P	S/P	S/P	S/P	S/P	S/P	S/P	S/P	PPL	PPL	PPL

Table 2. Output quality of models used in our experiment. There are two or three columns for each model. The first column is the tuning data used to test for output quality and guide the search in Algorithm 1. The remaining column(s) is the testing data that has not been used in the tuning process. Entries with two numbers show "SSIM / PSNR (dB)".

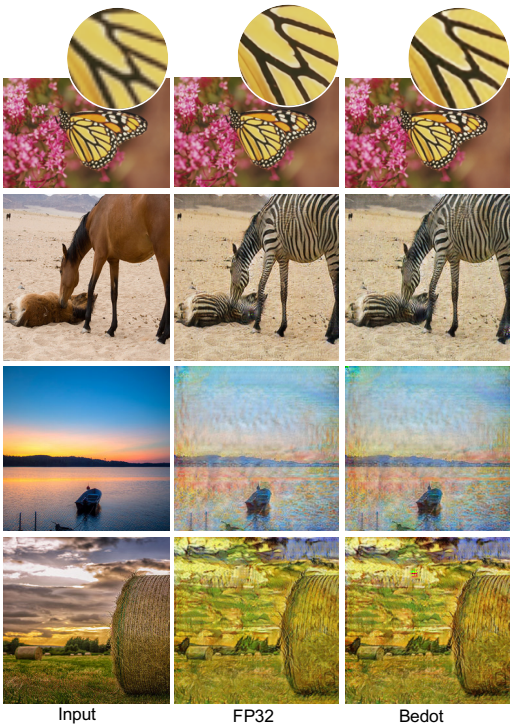

Fig. 9. Sample visual outputs of models. From top to bottom: Image super-resolution using ESRGAN, Horse-2-Zebra transform, Monet Style transform, Van Gogh Style transform. A live demo for ESRGAN is available at [30].

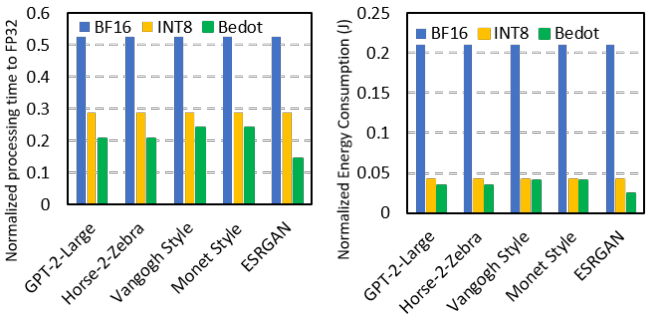

Fig. 10. Processing time and processing energy for several deep generative models using a 32-bit SIMD MAC design

6.1 Table configurations and accuracy

The software module (with rounding hints enabled) is developed and tested using our extension Qtorch+ [31] based on the design of QPytorch library [32]. We test our software module to optimize five emerging applications including GANs and GPT2. The Horse2Zebra model is used to transform an image of a horse to a zebra. The Van Gogh and Monet style models are used to transform a photo to a drawing mimicking the styles of those artists. These image transforming tasks use the custom datasets described in the original paper [33] (h2z : Horse2zebra dataset, v2p: VanGogh2photo dataset, m2p: Monet2photo dataset in Table 2). The *image superresolution* model (ESRGAN) was introduced in [34] with *Set14* and *Set5* datasets [35,36]. We also process a language model for text generation. For this task, we choose OpenAI's GPT2-large (762 million parameters) model [37] for Wikitext-2, Wikitext-103 datasets [38]. Note that only the *model* is released by OpenAI. Input pre-processing and measurement parameters affect the perplexity results but are not publicly released. So perplexity is measured using the guide from [39] which closely matches the result reported in the original OpenAI paper. For comparison, we use all available INT8 quantization in both the quantization module of Pytorch [40] and Nvidia's TensorRT [4]. Because each of these frameworks and quantization modes has their own limitations in the supported models and layer types, they also produce different output quality as the hyper parameters need to be manually chosen. We present only the highest output quality obtained after trying many possible variants of the configurations described in these frameworks. These configurations include: Pytorch quantization, TensorRT with different calibrations for *amax*: max, histogram, entropy, percentile and mse. We tried 99.9 and 99.99 as percentile and 2 and 10 images for the number of calibration images. Apart from trying these manual configurations, we use their automatic quantization modules with all default settings.

For image generation tasks, we use the *structural similarity index* metric (SSIM [41]), a standard way to measure image quality by comparison with a reference image. The highest quality is SSIM = 1.0. We also included the *peak signal-to-noise ratio* (PSNR) metrics for image-related tasks. The best result in

each column is in bold. Note that, for most GANs, we do not have a reference image other than the image generated by the FP32 version. Thus, we use FP32 as the reference for comparison[4]. Sample visual output can be seen in Figure 9.

For GPT2, we use *perplexity* (PPL) as the metric, where a lower value indicates a better text generated [42]. The results are presented in Table 2 with FP32 reference perplexity included in the dataset's name (e.g. WikiText-2: FP32 achieves 19.1 PPL on the tuning set). When compared with the data published by OpenAI, we can see that the smaller variant of GPT2 (GPT2-medium 345 million parameters) was reported to have perplexity of 22-26 for WikiText tasks. Bedot only need a 3-bit table to encode weights while having better perplexity. Bedot can reduce the parameters size to $32/4 = 8\times$ (1 more bit for the sign of weight) compared to FP32. In the table, 'Bedot+H' means rounding hints are enabled for tuning. Tuning the rounding hints can improve the output quality by up to 3% in ESRGAN. In general, quality decreases only slightly when testing datasets other than the tuning dataset. We obtained mixed results when comparing Bedot with uniform INT8 quantization in the different standard frameworks. In general, INT8 uniform quantization works better than Bedot when the range is tight, and worse than Bedot when the dynamic range required is high.

6.2 Performance and Energy estimation

We estimate the processing time of deep generative applications. For the GPT2 and Horse-2-Zebra, we use Bedot with 128 entries. For the ESRGAN, we use Bedot with 16 entries. For the Van Gogh and Monet styles, we use Bedot with 256 entries. We assume a deep learning system that has only one MAC unit with speed and power as shown in Table 1, with the other operations performed sequentially at 200 MHz. Bedot reduces processing time by up to 1.98× and 3.62× compared to INT8 and BF16, respectively. For energy consumption, we only consider computation and do not include the register file, SRAM and DRAM accessing energy. Bedot also reduces the computation energy by 1.7× and 8.4× compared to INT8 and BF16, respectively. All results are in Figure 10.

7 Conclusion

We have introduced a suite of methods from software to hardware to realize an accelerator for emerging deep generative models based on a novel approach to the MAC operation. With the table optimization algorithm, we successfully deliver high quality output with only four lookup entries for image resolution upscaling. For other more difficult tasks, the table has to be larger but never exceeds 32 (5-bit) entries. We believe our approach will empower future devices to perform difficult deep learning tasks with very low energy consumption.

[4] ESRGAN can compare its output against the original images. For Set5, the model achieves a PSNR of 30.8/28/29/30.3 dB on FP32/Bedot/Bedot+H/INT8. The reduction in quality has the same trend when we compare against FP32. Thus we also use FP32 images in Table 2 for a consistent comparison.

Acknowledgements

This research/project is supported in part by the Ministry of Education, Singapore, under the Academic Research Fund Tier 1 (FY2018) and the Next Generation Arithmetic grant from the National Supercomputing Centre, A*STAR, Singapore.

References

1. Y. Lin, Y. Li, T. Liu, T. Xiao, T. Liu, and J. Zhu, "Towards fully 8-bit integer inference for the transformer model," *arXiv preprint arXiv:2009.08034*, 2020.
2. P. Wang, D. Wang, Y. Ji, X. Xie, H. Song, X. Liu, Y. Lyu, and Y. Xie, "Qgan: Quantized generative adversarial networks," *arXiv preprint arXiv:1901.08263*, 2019.
3. T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," *Advances in neural information processing systems*, vol. 29, pp. 2234–2242, 2016.
4. H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," *arXiv preprint arXiv:2004.09602*, 2020.
5. S. Kim, K.-I. Kum, and W. Sung, "Fixed-point optimization utility for c and c++ based digital signal processing programs," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 45, no. 11, pp. 1455–1464, 1998.
6. K.-I. Kum, J. Kang, and W. Sung, "Autoscaler for c: An optimizing floating-point to integer c program converter for fixed-point digital signal processors," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 840–848, 2000.
7. J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.
8. N.-M. Ho and W.-F. Wong, "Exploiting half precision arithmetic in nvidia gpus," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017, pp. 1–7.
9. N. J. Higham and T. Mary, "Mixed precision algorithms in numerical linear algebra," *Acta Numerica*, vol. 31, pp. 347–414, 2022.
10. N.-M. Ho, E. Manogaran, W.-F. Wong, and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 63–68.
11. H. De Silva, A. E. Santosa, N.-M. Ho, and W.-F. Wong, "Approxsymate: Path sensitive program approximation using symbolic execution," in *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2019, pp. 148–162.
12. J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing frontiers and innovations*, vol. 4, no. 2, pp. 71–86, 2017.
13. S. D. Ciocirlan, D. Loghin, L. Ramapantulu, N. Ţăpuş, and Y. M. Teo, "The accuracy and efficiency of posit arithmetic," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 2021, pp. 83–87.

14. A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *arXiv preprint arXiv:2103.13630*, 2021.
15. R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.
16. Y. Li, X. Dong, and W. Wang, "Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks," *arXiv preprint arXiv:1909.13144*, 2019.
17. M. Cococcioni, F. Rossi, E. Ruffaldi, S. Saponara, and B. D. de Dinechin, "Novel arithmetics in deep neural networks signal processing for autonomous driving: Challenges and opportunities," *IEEE Signal Processing Magazine*, vol. 38, no. 1, pp. 97–110, 2020.
18. N.-M. Ho, D.-T. Nguyen, H. De Silva, J. L. Gustafson, W.-F. Wong, and I. J. Chang, "Posit arithmetic for the training and deployment of generative adversarial networks," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1350–1355.
19. Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard, "Adaptive quantization for deep neural network," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
20. Y. Li, R. Gong, X. Tan, Y. Yang, P. Hu, Q. Zhang, F. Yu, W. Wang, and S. Gu, "Brecq: Pushing the limit of post-training quantization by block reconstruction," *arXiv preprint arXiv:2102.05426*, 2021.
21. O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit bert," *arXiv preprint arXiv:1910.06188*, 2019.
22. Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE journal of solid-state circuits*, vol. 52, no. 1, pp. 127–138, 2016.
23. A. K. Ramanathan, G. S. Kalsi, S. Srinivasa, T. M. Chandran, K. R. Pillai, O. J. Omer, V. Narayanan, and S. Subramoney, "Look-up table based energy efficient processing in cache support for neural network acceleration," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 88–101.
24. X. Sun, J. Choi, C.-Y. Chen, N. Wang, S. Venkataramani, V. V. Srinivasan, X. Cui, W. Zhang, and K. Gopalakrishnan, "Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks," *Advances in Neural Information Processing Systems*, vol. 32, pp. 4900–4909, 2019.
25. J. W. Tukey *et al.*, *Exploratory data analysis*. Reading, Mass., 1977, vol. 2.
26. R. Dawson, "How significant is a boxplot outlier?" *Journal of Statistics Education*, vol. 19, no. 2, 2011.
27. P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.
28. H. F. Langroudi, V. Karia, J. L. Gustafson, and D. Kudithipudi, "Adaptive posit: Parameter aware numerical format for deep learning inference on the edge," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 726–727.
29. J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 174–187, 2020.
30. Anonymous, "Anonymous demo," 2021. [Online]. Available: https://colab.research.google.com/drive/1mT-tBy5gpn8lassGIlYwS9q1cAW9O5ot?usp=sharing

31. N.-M. Ho, H. De Silva, J. L. Gustafson, and W.-F. Wong, "Qtorch+: Next generation arithmetic for pytorch machine learning," in *Conference on Next Generation Arithmetic*. Springer, 2022, pp. 31–49.
32. T. Zhang, Z. Lin, G. Yang, and C. De Sa, "Qpytorch: A low-precision arithmetic simulation framework," *arXiv preprint arXiv:1910.04540*, 2019.
33. J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired image-to-image translation using cycle-consistent adversarial networks," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2223–2232.
34. X. Wang, K. Yu, S. Wu, J. Gu, Y. Liu, C. Dong, Y. Qiao, and C. Change Loy, "Esrgan: Enhanced super-resolution generative adversarial networks," in *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, 2018, pp. 0–0.
35. M. Bevilacqua, A. Roumy, C. Guillemot, and M. L. Alberi-Morel, "Low-complexity single-image super-resolution based on nonnegative neighbor embedding," *BMVA press*, 2012.
36. R. Zeyde, M. Elad, and M. Protter, "On single image scale-up using sparse-representations," in *International conference on curves and surfaces*. Springer, 2010, pp. 711–730.
37. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
38. M. Stephen, X. Caiming, B. James, and R. Socher, "The wikitext long term dependency language modeling dataset," 2016.
39. T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.
40. R. Krishnamoorthi, R. James, N. Min, G. Chris, and W. Seth, "Introduction to quantization on pytorch," 2020. [Online]. Available: https://pytorch.org/blog/introduction-to-quantization-on-pytorch/
41. Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.
42. L. R. Bahl, F. Jelinek, and R. L. Mercer, "A maximum likelihood approach to continuous speech recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 2, pp. 179–190, 1983.