8-bit Transformer Inference and Fine-tuning for Edge Accelerators

Jeffrey Yu
Stanford University
Palo Alto, United States
jeffreyy@stanford.edu

Kartik Prabhu
Stanford University
Palo Alto, United States
kprabhu7@stanford.edu

Yonatan Urman
Stanford University
Palo Alto, United States
yurman@stanford.edu

Robert M. Radway
Stanford University
Palo Alto, United States
radway@stanford.edu

Eric Han
Stanford University
Palo Alto, United States
erichan2@stanford.edu

Priyanka Raina
Stanford University
Palo Alto, United States
praina@stanford.edu

Abstract

Transformer models achieve state-of-the-art accuracy on natural language processing (NLP) and vision tasks, but demand significant computation and memory resources, which makes it difficult to perform inference and training (fine-tuning) on edge accelerators. Quantization to lower precision data types is a promising way to reduce computation and memory resources. Prior work has employed 8-bit integer (int8) quantization for Transformer inference, but int8 lacks the precision and range required for training. 8-bit floating-point (FP8) quantization has been used for Transformer training, but prior work only quantizes the inputs to matrix multiplications and leaves the rest of the operations in high precision.

This work conducts an in-depth analysis of Transformer inference and fine-tuning at the edge using two 8-bit floating-point data types: FP8 and 8-bit posit (Posit8). Unlike FP8, posit has variable length exponent and fraction fields, leading to higher precision for values around 1, making it well suited for storing Transformer weights and activations. As opposed to prior work, we evaluate the impact of quantizing all operations in both the forward and backward passes, going beyond just matrix multiplications. Specifically, our work makes the following contributions: (1) We perform Transformer inference in FP8 and Posit8, achieving less than 1% accuracy loss compared to BFloat16 through operation fusion, without the need for scaling factors. (2) We perform Transformer fine-tuning in 8 bits by adapting low-rank adaptation (LoRA) to Posit8 and FP8, enabling 8-bit GEMM operations with increased multiply-accumulate efficiency and reduced memory accesses. (3) We design an area- and power-efficient posit softmax, which employs bitwise operations to approximate the exponential and reciprocal functions. The resulting vector unit in the Posit8 accelerator, that performs both softmax computation and other element-wise operations in Transformers, is on average 33% smaller and consumes 35% less power than the vector unit in the FP8 accelerator, while maintaining the same level of accuracy. Our work demonstrates that both Posit8 and FP8 can achieve inference and fine-tuning accuracy comparable to BFloat16, while reducing accelerator's area by 30% and 34%, and power consumption by 26% and 32%, respectively.

ACM Reference Format:
Jeffrey Yu, Kartik Prabhu, Yonatan Urman, Robert M. Radway, Eric Han, and Priyanka Raina. 2024. 8-bit Transformer Inference and Fine-tuning for Edge Accelerators. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620666.3651368

1 Introduction

Transformer models have demonstrated state-of-the-art accuracy across various natural language processing (NLP) and vision tasks [2, 4, 8, 9, 27, 29]. For instance, BERT [8], and GPT [2], two well-known transformer-based large language models (LLMs), achieved state-of-the-art performance on the GLUE benchmark [30], question answering [25], and text generation [2]. Similarly, Transformer-based vision models such as Vision Transformer (ViT) [9], Swin Transformer, [16] and DEtection TRansformer (DETR) [4] outperform leading convolutional neural networks (CNNs) on image recognition, segmentation and object detection tasks respectively.

In recent years, a popular approach has been to pre-train large transformer models on massive datasets and then fine-tune them on a specific downstream task of interest. Compared to training models from scratch, utilizing pre-trained models provides numerous benefits, including accelerated training times and increased reusability. As a result, many recent works [15, 24] have followed this paradigm of pre-training followed by *fine-tuning*, making it the leading approach for NLP and vision training for downstream tasks [5, 9].

With the growing popularity of Transformer models, there is a compelling incentive for performing fine-tuning and inference on edge hardware accelerators. This provides numerous benefits, such as better privacy, improved latency, and reduced power consumption by eliminating the need for data transmission. However, the size of Transformer models presents significant challenges. For example, BERT$_{\text{large}}$ consists of 340 million parameters, while GPT models contain tens or even hundreds of billions of parameters.

To support Transformers on resource-limited edge accelerators, compressed versions of BERT$_{\text{large}}$, such as MobileBERT [28], have been developed that reduce the number of model parameters. MobileBERT has the same number of encoder layers as BERT$_{\text{large}}$ (24 layers) but significantly fewer parameters. To achieve this compression and to keep the model task-agnostic such that it can be fine-tuned on different downstream tasks (and not tailored for a specific task), the model was trained using layer-wise knowledge distillation [11]. However, with 25 million parameters, fine-tuning MobileBERT in half-precision floating-point format on a modern GPU still requires over 1.6 GB of memory, which is too large for typical edge accelerators [22].

Quantization is a complementary model compression technique that has been widely adopted to further improve fine-tuning and inference energy efficiency through smaller and more energy-efficient multiply-accumulate (MAC) units and lower memory consumption. 8-bit integer (int8) quantization has proven to maintain inference accuracy levels similar to those of 16-bit floating-point models [7, 33]. However, int8 inference requires complicated quantization methods, such as per-channel or per-vector scaling (via an additional set of scaling factors), or special handling of outliers, making the deployment of integer models challenging. Integer quantization has also been employed in Transformer training [32][1], but it has the same drawbacks as int8 inference. Moreover, these approaches mainly focus on quantizing the inputs to the matrix-multiplication operation only, with the data still being stored in high-precision floating-point formats in the memory. A recent paper by NVIDIA also shows promise for 8-bit floating-point (FP8) based DNN training and inference [19], but it only applies FP8 quantization to general matrix-multiply (GEMM) inputs to improve efficiency on high-end GPUs. Consequently, a comprehensive solution for inference and fine-tuning of Transformers at the edge remains elusive.

In this work, we introduce methods to perform Transformer inference and fine-tuning at the edge in 8 bits using FP8 and an alternative data type known as posit [10]. Posit, designed as a direct substitute for floating-point numbers, has several advantages over FP8, such as tapered precision and the ability to approximate floating-point arithmetic using bitwise operations. This paper delves into the application of both FP8 and an 8-bit variant of posit, Posit8, in Transformer inference and fine-tuning. We provide a comprehensive analysis of these two data types, focusing on their accuracy and hardware implications. Our key contributions are as follows:

- We analyze the accuracy impact of 8-bit quantization on operations beyond GEMM in Transformer models. We perform operation fusion to minimize inference accuracy loss due to quantization errors. With proper fusion schemes, both Posit8 and FP8 can achieve inference accuracy on par with BFloat16 in SQuAD question answering, speech recognition, and language modeling tasks, even without using scaling factors.
- We adapt low-rank adaptation (LoRA) to FP8 and Posit8, which enables all GEMM operations to be performed using a single 8 bit data type. In contrast, LoRA with int8 necessitates merging floating-point LoRA weights with int8 weights and executing high-precision floating-point operations. Both Posit8 and FP8 match the accuracy of BFloat16 on MobileBERT and RoBERTa models while reducing the memory requirements for fine-tuning by 3×.
- We design an area- and power-efficient posit softmax by approximating exponential and reciprocal operations with bitwise operations and subtraction. As a result, the vector unit used for computing softmax and other element-wise operations in Transformers is 33% smaller and consumes 35% less power in the Posit8 accelerator compared to the hybrid FP8 accelerator, while maintaining the same level of accuracy.

The rest of this paper is organized as follows. In section 3, we explain floating point and posit number systems and associated arithmetic operations. In section 4 and section 5, we present our 8-bit inference and fine-tuning techniques respectively. Finally, we present our training results in section 6 and accelerator analysis results in section 7.

2 Related Work

DNN quantization and parameter-efficient fine-tuning (PEFT) are two main strategies for minimizing training memory in deep neural networks (DNNs). DNN quantization involves converting model parameters and intermediate outputs to a lower-precision format. PEFT trains a subset of the total parameters by incorporating small adapter layers and exclusively training these injected parameters. To execute Transformer fine-tuning and inference on edge accelerators both techniques are required.

2.1 DNN Quantization

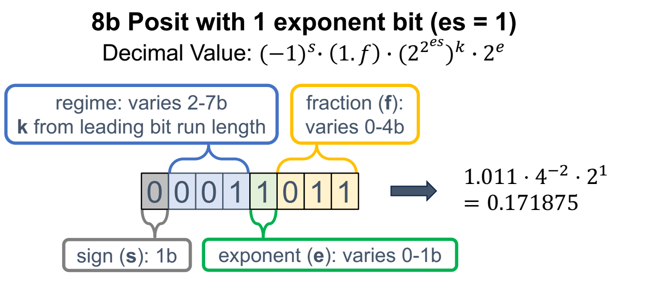
Extensive research has been conducted on DNN inference and training quantization to minimize computational demands. 16-bit floating-point has demonstrated no accuracy loss across various DNNs versus 32-bit floating point (FP32) and resulted in 2-6× speedup on a Volta GPU [18]. Further reduction to 8-bit DNN training and inference is a logical next step. 8-bit integers have been employed for CNN and Transformer inference, achieving minimal-to-no accuracy loss [7, 33]. However, int8 inference typically necessitates quantization-aware training (QAT) or extensive use of scaling factors to maintain full accuracy.

Recently, 8-bit floating point (FP8) was introduced to create a more efficient arithmetic pipeline and reduce memory bandwidth. FP8 implementations of Transformers match the inference and training accuracy of 16-bit floating-point [19]. However, [19] only quantizes GEMM inputs to FP8 to leverage energy-efficient MACs, leaving other intermediates in a high-precision format. The exploration of quantization of inputs to operations beyond GEMM remains an area that has not been thoroughly investigated.

Posit has been proposed as a replacement for floating point in traditional DNN inference and training [10]. 8-bit posit achieved within 0.9% top-1 accuracy and within 0.2% top-5 accuracy of the original FP32 ResNet-50 CNN model on ImageNet without re-training [14], rendering itself as a strong replacement. For CNN inference, 16-bit posit has demonstrated superior performance compared to 16-bit floating-point representations, while 8-bit posit has exhibited performance comparable to that of 16-bit floating-point numbers [20, 26]. Additionally, 8-bit posit has been used to train ResNet-18 on ImageNet and CIFAR-10 and shown state-of-the-art accuracy [17]. However, no prior work explores the use of posits for Transformer inference and fine-tuning, which are harder to quantize compared to CNNs.

2.2 Parameter Efficient Fine-Tuning

Several strategies have been devised for parameter-efficient fine-tuning, including segmented training [34] and parameter injection [12, 13]. Segmented training only trains specific portions of the model at a time. For example, [34] trains only the model's biases instead of conducting full fine-tuning; remarkably, by training less than 1% of BERT's weights, they achieved accuracy on the GLUE benchmark comparable to that of full fine-tuning. Parameter injection introduces a new set of trainable parameters, while keeping the rest of the original model's weights constant. The adapter approach [12] inserts small adapter layers within the Transformer block, which are then trained (with the rest of the model remaining fixed). Low-rank adaptation (LoRA) [13] has been proposed to further minimize the number of parameters trained. LoRA injects additional trainable parameters (leaving the original model in 8 bits) for fine-tuning, achieving full Transformer



8b Posit with 1 exponent bit (es = 1)
Decimal Value: $(-1)^s \cdot (1.f) \cdot (2^{2^{es}})^k \cdot 2^e$
regime: varies 2-7b
k from leading bit run length
fraction (f): varies 0-4b
0 0 0 1 1 0 1 1
sign (s): 1b
exponent (e): varies 0-1b
$1.011 \cdot 4^{-2} \cdot 2^1$
$= 0.171875$

Figure 1. 8-bit posit with 1 exponent bit. The sign, exponent and fraction fields are similar to floating-point, but there is an extra variable length regime that acts as an extra exponent.

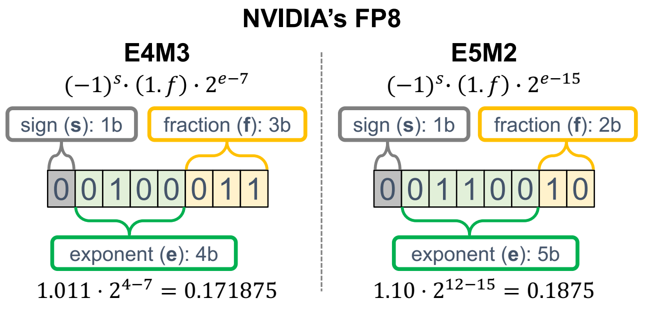
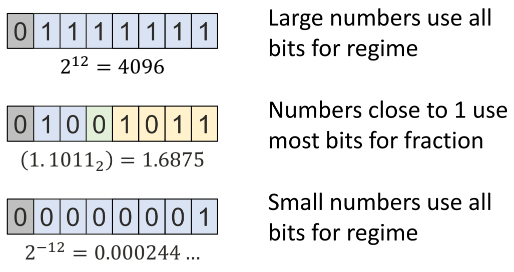


Figure 2. NVIDIA's 8-bit floating-point (FP8) format for DNN training and inference. E4M3 has 4 exponent bits and 3 mantissa bits, while E5M2 has 5 exponent bits and 2 mantissa bits, representing different precision-range trade off choices.



0 1 1 1 1 1 1 1
$2^{12} = 4096$
Large numbers use all bits for regime
0 1 0 0 1 0 1 1
$(1.1011_2) = 1.6875$
Numbers close to 1 use most bits for fraction
0 0 0 0 0 0 0 1
$2^{-12} = 0.000244 \ldots$
Small numbers use all bits for regime

Figure 3. Posits have variable length fields. Very large and very small numbers use all bits for regime, while numbers close to 1 use most of the bits for fraction.

fine-tuning accuracy. As we show in section 5, we leverage this approach to enable full 8-bit training.

3 Posit for DNN Training

Posit is the third generation of universal numbers, designed to supersede IEEE Standard 754 floating-point numbers [10]. Compared to floating-point, posit numbers have more extensive dynamic range, tapered precision, and reduced space requirements. These benefits render posit numbers well-suited for DNN training and inference.

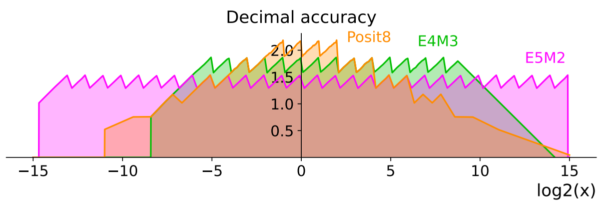


Figure 4. Decimal accuracy of FP8 (E5M2, E4M3) vs. Posit8.

Posits comprise of four fields: sign, regime, exponent, and fraction, as shown in Figure 1. Aside from the sign bit, all fields are variable length. This is one of the key differences versus traditional floating-point formats, which only contain fixed length fields, as shown in Figure 2. The sign bit denotes the number's positive or negative nature. The following regime field is a series of identical bits terminated by a contrasting bit, whose the numerical value, k, depends on the run length. Let m be the number of identical bits in the run and k be the value of regime; if the repeating bits are 0, then $k = -m$; if they are 1, then $k = m - 1$. Regime is unique to posits and is not present in floating-point. The regime signifies a scaling factor of $useed^k$, where $useed = 2^{2^{es}}$ and es represents the number of exponent bits. Following the regime bits are the exponent bits. Posits do not always have exponent bits; they can contain up to es exponent bits that represent a scaling factor of 2^e, where e is the unsigned value of the exponent bits. The remaining bits are allocated to the fraction field. Similar to floats, there is an implicit 1 before the fraction part. As shown in Figure 3, numbers are represented with different number of regime, exponent, and fraction bits, depending on their magnitude. Combining all elements, the value of posit numbers is derived from the sign (s), regime, exponent (e), and fraction (f) fields as follows:

$$x = (-1)^s \cdot 1.f \cdot (2^{2^{es}})^k \cdot 2^e \quad (1)$$

As a result of the variable-length fields, posits have a tapered precision, which means that their precision decreases more rapidly vs. floating-point as the true numerical value moves further away from 1. This property is illustrated in Figure 4 for Posit8 vs. FP8. For DNNs, where most tensor values typically follow a Gaussian distribution, this property of posits can be leveraged effectively.

In this paper, we use 8-bit posit with 1 exponent bit ($es = 1$), denoted as Posit (8, 1), or simply Posit8. When contrasted with Posit (8, 0), the range of Posit (8, 1) is considerably more extensive, spanning from 2^{-12} to 2^{12}. In comparison, Posit (8, 0) is limited in its representational capacity, only able to cover numbers from 2^{-6} to 2^6. The extended range is necessary for representing activations and gradients.

3.1 Posit Encoding and Decoding

Posits must be decoded into a format resembling floating-point before most arithmetic operations can be performed. The decoding process for a posit proceeds from left to right, following the order of regime, exponent, and fraction. The regime value is decoded by counting the number of repeating bits immediately following the sign bit. This step can be implemented using a leading one counter in hardware. The regime value carries a weight of 2^{es}, resulting in an effective power of 2 scaling of $2^{es} \cdot k$. If any bits remain after decoding the regime, they are first allocated to the exponent, leading to a total power of 2 scaling of $2^{es} \cdot k + e$, where e is the unsigned exponent value. All remaining bits after the exponent are the fraction.

Values must be converted back to posit format before being stored in memory. The encoding process for posits also proceeds from left to right. First, the floating-point exponent is decomposed into the posit regime and exponent. The length of the regime is determined by dividing the exponent by 2^{es}, and the posit exponent is obtained by taking the floating-point exponent modulo 2^{es}. Based on the length of the regime and the value of the exponent, the number of fraction bits can be determined. The sign, regime, exponent, and fraction are then assembled, and rounding is performed using the round-to-even policy.

3.2 Posit Arithmetic with Fused Operations

Similar to floating-point, posits use fused operations, which means deferring rounding until the last operation in a series of operations [10]. Fusing operations enhances efficiency by eliminating continuous encoding and decoding of intermediate outputs. Additionally, bypassing the encoding and decoding stages reduces quantization error during accumulation. This is consistent with modern DNN accelerators which typically employ a high-precision format for accumulation.

3.3 Approximate Operations Using Posits

Sigmoid Approximation. Posits with 0 exponent bits can approximate the sigmoid function by inverting the most significant bit and shifting all bits two positions to the right (shifting in 0 bits on the left) [6]. Note that this approximation is specific to posits with 0 exponent bits. However, Posit8 with one exponent bit achieves better performance for Transformer training and inference due to its extended range. Consequently, a conversion process must be implemented to take advantage of the approximation.

Reciprocal Approximation. Using posits with arbitrary exponent bits we can approximate reciprocal by performing a bitwise XOR operation with a negated *signmask* [6]. The *signmask* is defined as 1 for the sign position and 0 elsewhere. This operation is equivalent to inverting all the bits except the sign bit, and it can be implemented using NOT gates. Posit reciprocal is a piece-wise linear function as illustrated in Figure 7. While the approximation of reciprocal using piece-wise linear functions has previously been proposed as a replacement for division in softmax [3], posit takes this concept further by enabling the execution of this operation through simple bitwise operations, rather than arithmetic calculations, leading to better hardware efficiency.

We apply these two approximations to Transformer training and inference, as explained in next two sections.

3.4 Posit Rounding

Posits adhere to the floating-point rounding mechanism in most instances, rounding numbers to the nearest representable posit values. However, posit saturates values beyond its representable range. For example, for Posit (8, 1), values are saturated at 2^{12}, while values smaller than 2^{-12} are rounded up to 2^{-12}. The former case is a common practice in DNN training while the latter case presents problems. Gradients are often smaller than 2^{-12} and rounding all of them up could easily lead to divergence. We use a round-to-even policy when the values are smaller than posit's minimum representable value, meaning that values smaller than 2^{-13} would be rounded down to 0 instead of rounded up to 2^{-12}. This is different from the original posit definition but proves to be useful in training DNNs with posits.

4 8-bit Transformer Inference

Performing Transformer inference using 8-bit number formats reduces both memory capacity and bandwidth requirements, and improves the area and energy-efficiency of MAC units. Existing research mostly focuses on the quantization of inputs to computationally intensive operations like GEMMs. However, GEMMs constitute only a subset of the operations performed during Transformer inference, meaning that a significant proportion of the activations are left in a high-precision format.

We evaluate the post-training quantization (PTQ) accuracy of quantizing GEMMs and different sets of element-wise operations on Transformers of various sizes. Specifically, we demonstrate the effect of quantizing inputs to residual addition, layer normalization, non-linear activation functions (e.g., softmax and GeLU), and attention scaling, as illustrated in Figure 5. To mitigate the accuracy loss from quantization, we fuse element-wise operations with the preceding GEMM operation to reduce quantization error. Note that we do not adhere to the conventional int8 practice of using per-channel scaling factors for weights and per-tensor scaling factors for activations, as our findings show that both Posit8 and FP8 formats can achieve good accuracy through operation fusion alone, without the need for scaling factors.

We first conduct an ablation study to assess the impact of quantizing different operations on accuracy. Table 1 shows MobileBERT and BERT F1 scores on the SQuAD v1.1 question answering dataset [25] when quantizing GEMM with other Transformer operations to Posit (8, 1). Our findings indicate that quantizing attention scaling has the largest impact on accuracy, followed by non-linear activations, layer normalization, and finally residual addition. Based on the results, we choose to apply operation fusion in the same order, from those with the greatest impact on accuracy to those with the least.

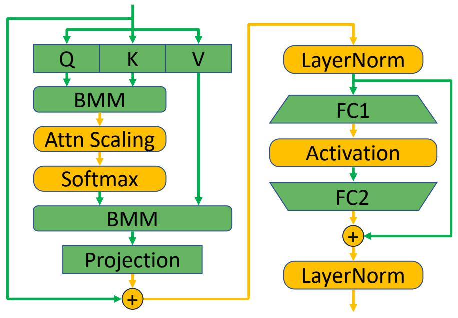


Figure 5. A typical Transformer block. GEMM operations are colored green while non-GEMM operations are yellow.

Operations	MobileBERT	BERT
BF16	89.9	88.2
GEMM	89.4	88.1
GEMM + Residual	89.0	88.1
GEMM + LayerNorm	88.7	88.1
GEMM + Activation	86.7	88.1
GEMM + Attn Scaling	70.4	87.4

Table 1. Accuracy impact of quantizing different Transformer operations to Posit8 in MobileBERT and BERT. The table shows F1 scores on the SQuAD v1.1 dataset.

Table 2 shows the F1 score on the SQuAD v1.1 question answering dataset using post-training quantization with different levels of fusion applied to MobileBERT and BERT models. We apply fusion incrementally from left to right. For instance, the third column (marked "+ Activation Fusion") means the fusion of both attention scaling and activation functions with the previous GEMM, while the last column (marked "+ Residual Fusion") indicates fusing all operations. Our goal is to limit the accuracy loss to within 1%. Our results show that for small models like MobileBERT, we need to fuse all the operations with GEMM to achieve the goal. In contrast, with larger BERT models, we are able to quantize most operations with minimal accuracy loss. The figures in bold are the final configurations utilized for quantized inference. They also provide a benchmark for quantized training.

Our results show a gradual improvement in accuracy with higher levels of fusion, except in the case of MobileBERT where the impact of quantizing unscaled attention (the outputs of the query-key matrix multiplication) is much more drastic. This issue is predominantly due to MobileBERT's use of stacked feed-forward network (FFN) layers which results in a wider distribution of activations as illustrated

Model	Size	BF16	No Fusion		GEMM + Attn Scaling Fusion		+ Activation Fusion		+ LayerNorm Fusion		+ Residual Fusion	
			Posit8	E4M3	Posit8	E4M3	Posit8	E4M3	Posit8	E4M3	Posit8	E4M3
MobileBERT$_{tiny}$	16M	88.8	86.3	87.0	87.4	87.1	87.7	87.5	**87.9**	87.8	88.4	**88.1**
MobileBERT	25M	89.9	65.1	82.7	85.0	84.9	88.3	86.7	**89.0**	87.9	89.4	88.6
DistilBERT$_{base}$	66M	86.9	**86.2**	**86.1**	86.4	86.1	86.7	86.4	86.7	86.5	86.7	86.5
BERT$_{base}$	109M	88.2	87.1	**87.7**	**88.1**	88.0	88.1	88.0	88.1	88.0	88.1	88.0
BERT$_{large}$	334M	93.2	92.3	**93.0**	**92.8**	93.1	93.0	93.1	93.0	93.2	93.1	93.1

Table 2. Transformers' F1 scores on SQuAD v1.1 using Posit8 and FP8 with varying levels of operation fusion. Figures in bold indicate the minimum fusion level needed to achieve within 1% accuracy drop. For MobileBERT models, we need to fuse all operations to achieve within 1% drop. For BERT models, we can easily achieve the same goal even without any fusion.

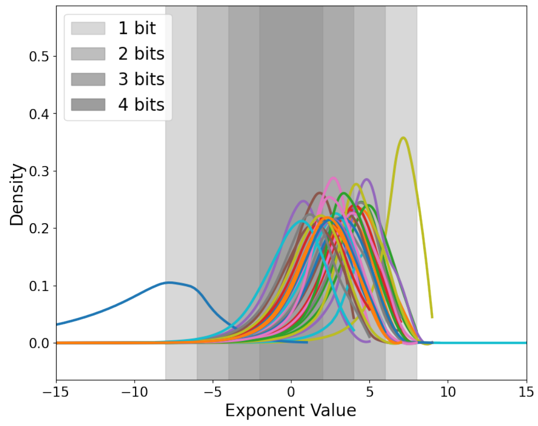


Figure 6. The activation distribution of each layer of MobileBERT during inference on SQuAD. The areas shaded from the darkest to the lightest represent value ranges where Posit8 has 4 to 1 fraction bits, respectively.

in Figure 6. In such cases, posit's tapered precision struggles to accurately represent most values, while FP8, with its uniform precision across its entire value range, has better performance. Interestingly, MobileBERT$_{tiny}$, which features two fewer FFN layers, does not exhibit the same level of accuracy degradation when quantizing unscaled attention. This illustrates how the architectural distinctions of Transformer models can influence their numerical behavior, sometimes favoring one data type over another. Overall, Posit8 performs better than FP8 in most cases and both Posit8 and FP8 can reach within 1% of BFloat16 (BF16) accuracy.

4.1 Approximate Softmax Using Posits

The softmax function (σ) occurs repeatedly in Transformer models (i.e., in every attention layer) unlike in CNNs where it appears only once at the end of the network. It is, therefore, critical to create an efficient posit implementation of the softmax function, computed as $\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$. Implementing softmax requires efficient posit exponential and division (or reciprocal) functions. As previously described in subsection 3.3, posits enable the approximation of functions such as sigmoid and reciprocal using bitwise operations. The approximated reciprocal closely aligns with the exact reciprocal function (Figure 7, left). Replacing the division in softmax with this approximate reciprocal results in only 0.8% accuracy loss on MobileBERT models and 0.1% on BERT models during inference (Table 4). By utilizing the approximate reciprocal, we can entirely avoid area-consuming floating-point dividers in hardware.

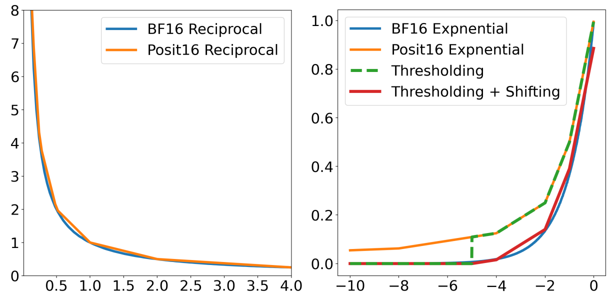


Figure 7. Left: Posit reciprocal is a piece-wise linear function that connects points with x-values corresponding to powers of 2. Right: The orange curve shows posit exponential using approximate sigmoid and reciprocal. It fails to converge to 0 as inputs approach negative infinity. The green and red curves show the effect of thresholding and shifting. Inputs smaller than -5 are rounded down to 0, while inputs between -5 and 0 closely mimic the floating-point exponential curve.

To construct the exponential function, we use the sigmoid function (S) from subsection 3.3 as follows:

$$S(x) = \frac{1}{1+e^{-x}} \Rightarrow e^x = \frac{1}{S(-x)} - 1 \quad (2)$$

That is, exponential is implemented by an approximate sigmoid, a reciprocal and a subtraction. It is important to note that the exponential approximation is valid only for non-positive inputs. However, this is not a concern for numerically stable softmax, as the maximum input value $\max(\vec{z})$ is

Threshold (θ)	Accuracy 1	Epsilon (ϵ)	Accuracy 2
-5	80.4	-1.109	89.4
-4	86.5	**-1.125**	**89.6**
-3	88.5	-1.188	88.0
-2	87.2	-1.250	84.8
Baseline BF16	89.9		

Table 3. MobileBERT F1 scores on SQuAD v1.1 using approximate exponential. "Accuracy 1" represents the F1 score after thresholding, while "Accuracy 2" is the F1 score after both thresholding and shifting optimization. The figure in bold denotes the best accuracy achieved.

	e^x	$1/x$	MobileBERT	$\text{BERT}_{\text{base}}$
BF16	-	-	89.9	88.2
Posit8	-	-	89.4	88.0
Posit8	✓	-	88.9	88.1
Posit8	-	✓	88.8	88.0
Posit8	✓	✓	88.6	87.9

Table 4. F1 scores of MobileBERT and BERT on SQuAD v1.1 with softmax built using approximate posit exponential (e^x) and posit reciprocal ($1/x$).

subtracted from z_k, ensuring that all inputs to the exponential function are less than or equal to 0.

A separate challenge is that the approximate exponential function does not converge to 0 as the input approaches negative infinity, as shown by the orange curve in Figure 7, bottom. This can severely degrade Transformer performance, as Transformers utilize attention masks to ignore tokens beyond the end of a sentence. If the exponential function fails to converge to 0, extraneous tokens will still receive some level of attention, thereby affecting overall accuracy of the model. A direct application of posit's approximate exponential function results in a 9.8% loss in accuracy. We perform two optimizations that reduce accuracy loss to under 1% without increased hardware complexity.

The first optimization involves truncating exponential outputs to 0 for inputs smaller than a specific threshold θ, as illustrated in Figure 7. This thresholding ensures proper attention masking. By sweeping the threshold θ (Table 3), we find that increasing the threshold initially performs better, with peak accuracy achieved at $\theta = -3$. Beyond this value, the accuracy falls. This suggests that small activation values have limited impact during inference and can be zeroed in this manner. However, the 3.3% drop in accuracy is still too large for most applications.

We implement an additional optimization to further narrow this accuracy gap. We shift the entire exponential approximation curve down by subtracting the value of the

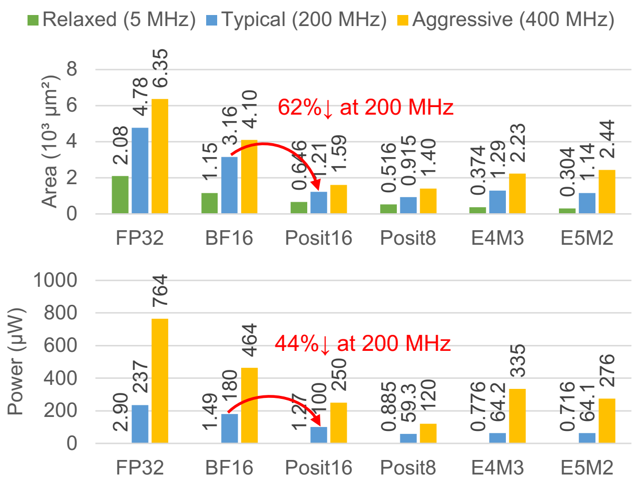


Figure 8. Exponential area and post synthesis power at 0.9V.

approximated exponential function at the threshold from the function itself, aligning it more closely with the original exponential function (Figure 7). This modification creates a smoother approximation, rather than a sharp transition at the threshold in the unmodified curve. As a result of this adjustment, the accuracy drop due to the approximation is reduced to just 0.3% (Table 3) without quantization and 0.8% with quantization (Table 4) for MobileBERT.

The final posit approximate exponential function is:

$$f(x) = \begin{cases} \frac{1}{S(-x)} - \epsilon & \text{if } x \geq \theta \\ 0 & \text{if } x < \theta \end{cases} \tag{3}$$

where θ is the threshold value below which the posit exponential output is rounded to 0 and ϵ is the value subtracted from the function to better match the actual exponential curve. The truncation can be implemented through a masking operation. Since the original approximation already involves a subtraction operation, the shifting optimization does not require any hardware change.

4.2 Posit Reciprocal and Exponential Hardware

We implement posit reciprocal and exponential units using high-level synthesis (HLS) and then synthesize them with Design Compiler in a 40 nm technology. We compare these with floating-point counterparts from the HLS library. Figure 8 and Figure 9 illustrate the area and power metrics for exponential and reciprocal units of various data formats, synthesized at different frequencies. At 200 MHz, the 16-bit posit-approximated exponential and reciprocal units are 62% and 85% smaller, respectively, and consume 44% and 75% less power than the BFloat16 hardware units.

Model	BF16	Data Type	No Fusion	Fuse GEMM + Attn Scaling	+ Activation Fusion	+ LayerNorm Fusion	+ Residual Fusion
Whisper$_{\text{tiny}}$ (39M)	7.54	Posit (8, 1)	10.42	9.67	9.50	9.65	9.97
		Posit (8, 2)	9.39	9.26	9.87	8.87	8.22
		E4M3	10.64	9.88	11.20	9.70	8.48
Whisper$_{\text{small}}$ (244M)	3.41	Posit (8, 1)	3.52	3.67	3.50	3.62	3.49
		Posit (8, 2)	3.71	3.69	3.68	3.63	3.53
		E4M3	3.62	3.58	4.01	3.48	3.41
Whisper$_{\text{large}}$ (1550M)	2.17	Posit (8, 1)	2.26	2.35	2.30	2.66	2.15
		Posit (8, 2)	2.34	2.38	2.48	2.37	2.13
		E4M3	3.06	2.41	2.95	2.39	2.14

Table 5. Whisper models' word error rate (WER) on LibriSpeech, with different levels of operation fusion and data types.

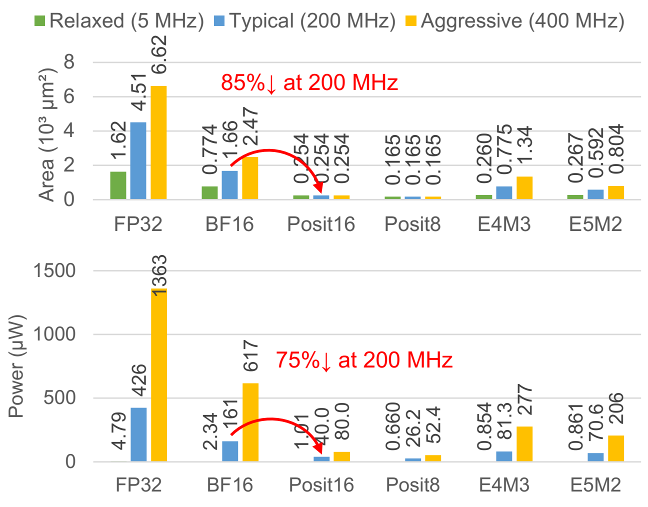


Figure 9. Reciprocal area and post synthesis power at 0.9V.

4.3 Extending Posit8 and FP8 Quantization to Larger Transformer Models

We extend our study of 8-bit Transformer inference to larger and more complex Transformers. Unlike the findings from the previous subsection, we observe that Posit8 and FP8 exhibit different behaviors on these models and tasks. In these studies, we also include another data type, 8-bit posit with 2 exponent bits. We find that Posit (8, 2) has a unique advantage in large models due to its wider range.

Table 5 shows the word error rate (WER) of Whisper [23] family of models evaluated on the LibriSpeech dataset [21]. Whisper is a pre-trained, Transformer-based encoder-decoder model, also known as a sequence-to-sequence model, designed for automatic speech recognition (ASR) and speech translation. We select a few models from this family, ranging from the smallest, Whisper$_{\text{tiny}}$, with 39 million parameters, to Whisper$_{\text{large}}$, having over 1.5 billion parameters. From the results, we make three observations. First, despite a general improvement in WER as we fuse more operations, we note that fusion can sometimes lead to a degradation in WER. This is caused by hallucination in the transcription process, where the Whisper model occasionally produces repeated text from previous segments, significantly impairing the overall WER. We observe that hallucinations occur more frequently on smaller models and are not associated with any specific fusion scheme. Consequently, WER may increase as more operations are fused due to a poor sample. Second, we find that larger models demonstrate greater robustness to quantization. Despite the occurrence of hallucinations, Whisper$_{\text{large}}$ achieves WERs within 1% of the BFloat16 results across all data types and fusion schemes. Third, we observe that Posit (8, 2) performs better than Posit (8, 1) and E4M3 on Whisper$_{\text{tiny}}$, suggesting that a wider range is more preferable over higher precision in this case.

Table 6 presents the perplexity of GPT-2 and LLaMA 2 models evaluated on the WikiText-103 test set, with experiments conducted using a maximum sequence length of 1024 and a stride of 512. Similar to the findings with the Whisper models, smaller models such as GPT-2 exhibit greater sensitivity to quantization. To maintain performance, these models generally require either fusing all operations or the use of per-tensor scaling. On the other hand, the larger LLaMA 2 models demonstrate robust accuracy across all three data types, without significant accuracy degradation even when quantizing all Transformer operations. Another interesting observation is that FP8 performs better with GPT-2, while Posit (8, 1) and Posit (8, 2) have lower perplexity in LLaMA 2 models. The advantage of posits over FP8 in larger models can be due to their extended range, which facilitates a more precise representation of outliers in residual layers.

Overall, our experiments with Whisper, GPT-2, and LLaMA 2 indicate that both Posit8 and FP8 can match the accuracy of BFloat16 as we scale up to larger and more complex Transformer models. However, certain data types may hold an

Model	BF16	Data Type	No Fusion	Fuse GEMM + Attn Scaling	+ Activation Fusion	+ LayerNorm Fusion	+ Residual Fusion
GPT-2 Large (762M)	16.38	Posit (8, 1)	18.00	17.75	17.50	17.50	16.63
		Posit (8, 2)	17.50	17.50	17.50	17.50	16.63
		E4M3	17.13	17.13	17.13	17.13	16.63
GPT-2 XL (1.5B)	14.69	Posit (8, 1)	18.00	17.75	17.75	17.50	14.94
		Posit (8, 2)	17.75	17.75	17.75	17.75	14.94
		E4M3	15.63	15.63	15.63	15.63	14.94
LLaMA 2 (7B)	5.19	Posit (8, 1)	5.56	5.53	5.53	5.52	5.30
		Posit (8, 2)	5.44	5.40	5.38	5.37	5.29
		E4M3	5.80	5.80	5.77	5.75	5.36
LLaMA 2 (13B)	4.63	Posit (8, 1)	4.85	4.78	4.78	4.77	4.72
		Posit (8, 2)	4.86	4.82	4.81	4.80	4.72
		E4M3	5.10	5.09	5.07	5.06	4.73

Table 6. Perplexity of LLMs on WikiText-103 using Posit (8, 1), Posit (8, 2), and FP8 with incremental levels of operator fusion.

advantage over others, varying by models and tasks. Therefore, selection of the appropriate data type should be tailored to the specific model and the associated workloads.

5 8-bit Transformer Fine-tuning

Unlike 8-bit Transformer inference, 8-bit training has received less attention. Prior work on using FP8 for DNN training only clips the inputs to GEMM operations to FP8 to take advantage of the more efficient 8-bit MAC kernel, while intermediate outputs are stored in a 16-bit format [19]. Moving intermediate outputs to an 8-bit format is advantageous for devices with limited memory. We introduce several techniques to enable 8-bit Transformer fine-tuning, providing a memory and compute-efficient solution for edge devices.

5.1 Per-tensor Scaling

During the backward pass, activation gradients are predominantly characterized by small magnitude values, and most of these values are beyond Posit8 and FP8's representable range as illustrated in Figure 10. To address the problem, scaling factors must be employed to adjust the values before quantizing them to 8 bits, ensuring that they fall within 8-bit representable range. We select a scaling factor such that the maximum absolute values (amax) in the tensor are close to the maximum representable magnitude in the corresponding format. For example, for FP8 E5M2, we scale the amax to 57344, the largest number E5M2 can represent. However, in the case of Posit8, scaling amax to Posit8's maximum representable value of 4096 would not be effective. Due to posit's tapered precision, large numbers have very few fraction bits and thus cannot be represented accurately. In our experiments, we found that scaling amax to 64 yields the best accuracy.

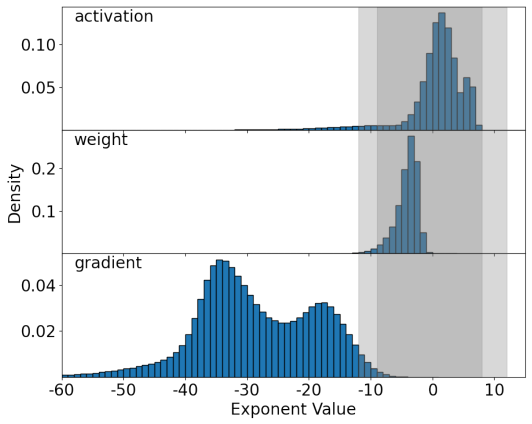


Figure 10. Tensor value distributions during fine-tuning of MobileBERT on SQuAD. The darker gray region indicates the span of E4M3, whereas the area encompassed by both light and dark gray represents the range achievable by Posit8. While both E4M3 and Posit8 cover the range of activations and weights, they fail to cover the activation gradients.

While most of the fine-tuning tasks can be performed using a single scaling factor, such as loss scaling [18], without any accuracy loss, there are some tasks where the range and distribution of activation gradient values are much wider. Consequently, Posit8 and FP8 cannot cover the union of all tensors' important values even with loss scaling. To address the problem, we apply per-tensor scaling on activation gradients, allowing each tensor to have its own exponent bias. Scaling is typically fused with the preceding operation to avoid writing high precision outputs to memory. Therefore, the scaling factors must be determined at the time the outputs are produced. A common approach is to use historical gradient statistics to predict the amax for this iteration and compute the scaling factor based on the prediction [19]. We adopt the same approach, maintaining a list of history amaxes for each tensor and selecting the maximum among these values to compute the scaling factor for the current step.

5.2 Handling Approximate Softmax During Training

We use approximate versions of exponential and reciprocal functions in softmax during training to avoid performing exact floating point exponential and division computations. The exponential function can be directly applied without any modification for the backward pass. However, the reciprocal approximation needs a custom backward pass different from the original softmax backward pass operation.

The gradient matrix of the softmax function can be derived using the quotient rule. However, posit reciprocal does not perform exact division. Instead, it can be modeled by a piece-wise linear function, where each segment connects the points $(2^n, 2^{-n})$ to $(2^{n+1}, 2^{-(n+1)})$, as illustrated in Figure 7. A comprehensive mathematical derivation of reciprocal is available in [6]. Performing the usual softmax gradient causes divergence. We re-derive the gradient of the softmax function that uses posit reciprocal.

Let us denote the posit reciprocal as a function f, which takes the sum of exponentials as its input:

$$\sigma(\vec{z})_j = e^{z_j} \cdot f(\sum_{k=1}^{K} e^{z_k})$$

We can apply the product rule of derivative:

$$\frac{\partial \sigma(\vec{z})_j}{\partial z_i} = \frac{\partial}{\partial z_i} e^{z_j} \cdot f(\sum_{k=1}^{K} e^{z_k}) + e^{z_j} \cdot f'(\sum_{k=1}^{K} e^{z_k}) \cdot \frac{\partial}{\partial z_i} \sum_{k=1}^{K} e^{z_k}$$

$$\frac{\partial \sigma(\vec{z})_j}{\partial z_i} = \begin{cases} \sigma(\vec{z})_j + e^{z_j} \cdot f' \cdot e^{z_i} & \text{if } i = j \\ e^{z_j} \cdot f' \cdot e^{z_i} & \text{if } i \neq j \end{cases} \tag{4}$$

where f' is a piece-wise linear function that models the derivative of posit reciprocal (Figure 7):

$$f' = -2^{-\lfloor \log_2 (\sum_{k=1}^{K} e^{z_k}) \rfloor \cdot 2 - 1} \tag{5}$$

Like the traditional softmax backward operation, the revised backward operation can still be vectorized on accelerators using the same hardware. While there might be added complexity during backpropagation due to the use of posit approximations, it simplifies the design of the accelerator hardware and enhances softmax efficiency.

5.3 LoRA with FP8 and Posit8

Edge accelerators often have very constrained memory, varying from mere kilobytes to several megabytes. This creates challenges for on-chip training, particularly with larger models, because each parameter requires an associated weight gradient to be stored. Low-rank adaptation (LoRA) [13] is an effective technique for reducing the number of trainable parameters. It also allows the pre-trained weights to be quantized to low-precision formats, usually int8, to decrease model storage requirements. However, prior LoRA implementations upscale quantized pre-trained weights to a high-precision format and merge them with trainable low-rank matrices before linear operations. This prevents the use of smaller, more efficient MACs with 8-bit arithmetic. Furthermore, merging the floating-point LoRA weights with int8 pre-trained weights can result in considerable accuracy loss. Therefore, the current approach for 8-bit LoRA fine-tuning is far from ideal, as it compromises both efficiency and accuracy.

We address the problem by performing quantization, on both the LoRA matrices and the merged weights, to Posit8 and FP8. The LoRA computation of a dense layer's output (h) on input x is:

$$h = W_0 x + \Delta W x = W_0 x + \alpha \cdot BAx \tag{6}$$

where W_0 is the pre-trained weight matrix, ΔW is the weight update to W_0, and B and A are the low-rank decomposition of ΔW with a scaling factor of α. We store LoRA matrices B and A in 16-bit floating-point, providing sufficient precision for weight updates. Before their multiplication, both matrices are quantized to 8 bits. Subsequently, the LoRA weights αBA are merged with the 8-bit pre-trained weights, and the combined weights are quantized to 8 bits prior to performing matrix multiplication with the input x. This approach enables the use of more efficient 8-bit GEMM operations and facilitates the integration of LoRA weights into the pre-trained weights without compromising accuracy.

$$h = \text{quant}(W_0^8 + \alpha \cdot \text{quant}(B^{16})\text{quant}(A^{16}))x \tag{7}$$

We evaluate our implementation of LoRA on MobileBERT and BERT models across GLUE and question answering tasks. The results, which are detailed in section 6, show that both Posit8 and FP8 attain an accuracy level comparable to BFloat16 using the same set of hyperparameters.

6 Fine-tuning Results

We perform quantized training experiments on GPUs by clipping tensor values to the Posit8 or FP8 representable range before and after each operation; storing the value back into BFloat16. The arithmetic is then carried out using BFloat16 to utilize the GPU's customized hardware for better performance. For FP8, we use E4M3 for forward pass and E5M2 for backward pass, adhering to NVIDIA's practice for FP8 training [19].

6.1 Models Evaluated

MobileBERT. MobileBERT is a streamlined version of $\text{BERT}_{\text{large}}$, maintaining the same number of encoder layers but with a significantly reduced hidden size. We obtain the

Model	Method	# Trainable Parameters	Accuracy				
			MNLI	QNLI	MRPC	SST-2	SQuAD
MobileBERT$_{\text{tiny}}$ (16.5M)	Full Training FP32 [28]	15.1M	82.0	89.9	86.7	91.6	88.6
	LoRA BF16	0.3M	82.9	90.7	88.0	91.4	88.1
	LoRA Posit8	0.3M	82.2	90.6	86.8	91.4	86.4
	LoRA Posit8 Approximation	0.3M	82.9	90.8	87.5	91.1	86.5
	LoRA FP8	0.3M	81.9	90.7	87.8	90.8	87.5
MobileBERT (25.3M)	Full Training FP32 [28]	25.3M	83.9	91.0	87.5	92.1	90.0
	LoRA BF16	0.3M	83.9	91.5	87.5	92.4	89.0
	LoRA Posit8	0.3M	83.3	91.5	87.8	91.7	87.4
	LoRA Posit8 Approximation	0.3M	83.1	91.5	87.5	92.2	88.1
	LoRA FP8	0.3M	83.0	91.1	87.8	91.7	87.8
RoBERTa$_{\text{base}}$ (125.0M)	Full Training FP32 [15]	125.0M	87.6	92.8	90.2	94.8	-
	LoRA BF16	0.3M	87.3	92.9	89.2	94.7	91.5
	LoRA Posit8	0.3M	87.1	92.5	89.5	94.6	91.2
	LoRA Posit8 Approximation	0.3M	86.9	92.5	89.0	94.4	91.1
	LoRA FP8	0.3M	86.8	92.9	89.5	95.0	91.2
RoBERTa$_{\text{large}}$ (355.0M)	Full Training FP32 [15]	355.0M	90.2	94.7	90.9	96.4	94.6
	LoRA BF16	0.8M	90.3	94.5	91.7	96.2	94.6
	LoRA Posit8	0.8M	90.0	94.3	91.9	96.0	94.0
	LoRA Posit8 Approximation	0.8M	90.0	94.3	91.2	96.0	93.7
	LoRA FP8	0.8M	89.9	94.6	90.2	96.1	94.1

Table 7. Accuracy of MobileBERT and RoBERTa models with different fine-tuning methods, data types, and quantization schemes on the GLUE benchmark and SQuAD v1.1. Full training means that the entire model is fine-tuned. Operation fusion scheme is selected based on the results of quantized inference. Per-tensor scaling is applied in all cases. We use the same LoRA configuration and hyperparameters for quantized training experiments as the BFloat16 (BF16) training sessions.

pre-trained MobileBERT from the Hugging Face Transformers library [31]. As Google has not released a pre-trained MobileBERT$_{\text{tiny}}$, we adapt the MobileBERT model by removing three encoder layers and reducing the number of feed-forward networks to 2, to match the MobileBERT$_{\text{tiny}}$'s specifications. MobileBERT's stacked FFN architecture results in larger and more unstable network outputs (section 4). To retain full-finetuning accuracy, we insert LoRA layers into every dense layer for MobileBERT and MobileBERT$_{\text{tiny}}$.

RoBERTa. RoBERTa uses the same architecture as BERT with an alternative pre-training method to achieve higher accuracies on GLUE benchmarks [15]. We take the pre-trained RoBERTa$_{\text{base}}$ (125M) from the HuggingFace Transformers library [31] and evaluate the performance of different quantization approaches on tasks from the GLUE benchmark. For all RoBERTa models, we apply LoRA to query weights W_q and value weights W_v of the Transformer's self-attention module and use a rank of 8 as in the original LoRA paper.

6.2 Results on GLUE

The General Language Understanding Evaluation (GLUE) benchmark [30] comprises a collection of nine tasks focused on natural language understanding. We conduct evaluations of BFloat16, Posit8, and FP8 LoRA training on the validation sets of four GLUE datasets. We find that both Posit8 and FP8 achieve accuracy within 0.5% of BFloat16 accuracy (Table 7). Also, posit approximation does not impact training performance, demonstrating the robustness of Posit8 training.

6.3 Results on SQuAD 1.1

SQuAD 1.1 is a large-scale reading comprehension dataset which only contains questions that have an answer in the given context. SQuAD 1.1 is a considerably harder task than GLUE, especially for smaller Transformers. We observed that gradient statistics are sparser and less stable. The use of the AdamW optimizer often led to divergence in MobileBERT and MobileBERT$_{\text{tiny}}$ model training. Consequently, we tested with the SGD optimizer, with which we obtained approximately 1% accuracy drop from BFloat16. In contrast, with larger models, both Posit8 and FP8 managed to maintain accuracy on par with BFloat16 without optimizer changes, as shown in Table 7.

7 Neural Network Accelerator Evaluation

A wide variety of neural network accelerators have been proposed to serve the needs of diverse applications. Typically,

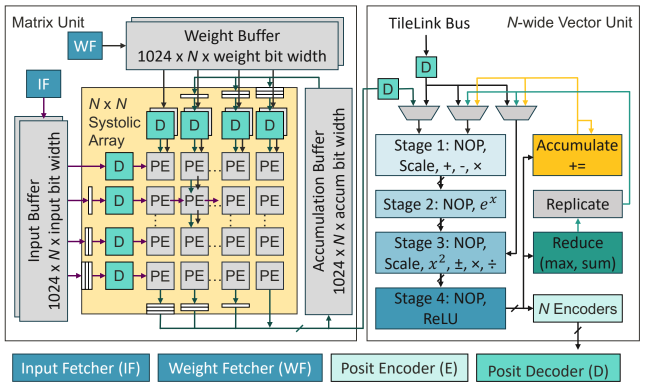


Figure 11. A standard neural network accelerator featuring a systolic array for matrix multiplication and a vector unit dedicated to executing element-wise operations and vector reductions. The encoders and decoders are only needed for posit-based accelerators. NOP means no operation.

they comprise a spatial array for efficient GEMM operations, and a vector unit for handling elementwise operations [22]. Our analysis adheres to this paradigm, with Figure 11 showing the architecture we use for area and power analysis. It consists of an N by N systolic array of processing elements (PEs) coupled with an N-lane vector unit. We carry out our analysis using operations in various data types synthesized via high-level synthesis (HLS). We then perform logic synthesis using Design Compiler (DC) in a 40nm technology. We evaluate the hardware at a range of target frequencies at a nominal voltage of 0.9V.

7.1 Multiply-and-Accumulate (MAC) Unit

The main component of our PEs is a MAC unit. For both FP32 and BFloat16, we carry out accumulation in FP32, aligned with the majority of GPU implementations, while FP8 and Posit8 accumulate in BFloat16. As described in section 3, decoded Posit8 has at most four fraction bits and an exponent range from -12 to 12, effectively requiring five exponent bits. A Posit8 MAC can therefore be implemented via a floating-point E5M4 MAC. As detailed in section 3 and section 6, FP8 employs both E4M3 and E5M2 formats for training purposes. To enable a fair comparison between FP8 and Posit8, we implement this hybrid FP8 as an E5M3, which can support both E4M3 and E5M2 formats for GEMM. From Figure 12, we can see that Posit8 has slightly larger MAC area and power compared to hybrid FP8 due to the extra fraction bit. However, both FP8 and Posit8 MACs are considerably smaller and lower power than BFloat16.

7.2 Posit Encoding and Decoding

Posit must be decoded into exponent and mantissa before each operation and encoded back into the posit format after the operation is complete (see section 3). Figure 12 shows the

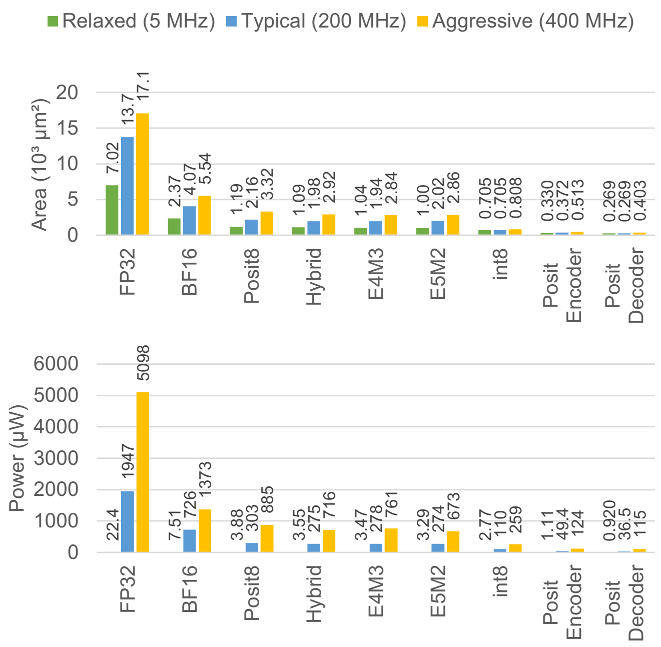


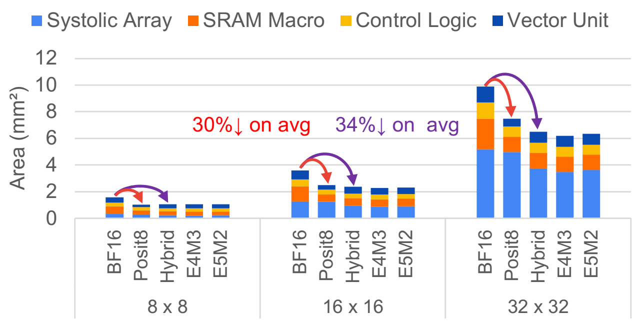
Figure 12. MAC area and power without encoding and decoding logic and Posit8 encoder and decoder area and power.

encoder and decoder area and power for Posit8 used in this paper. FP8 does not have an encoding or decoding process, which offers FP8 a slight advantage over Posit8.

7.3 Posit8 and FP8 Based Accelerators

We designed an accelerator with full support for all Transformer operations, as depicted in Figure 11. Both the matrix and vector units are configurable to different data types at design time. In our design, we use FP32 as the accumulation data type and the vector unit data type for BFloat16-based accelerators, and BFloat16 as the accumulation data type and vector unit data type for FP8/Posit8-based accelerators. We synthesized these designs using 40nm technology and compare the total area and power of the accelerators using BFloat16, Posit8, hybrid FP8 (which can accommodate both E4M3 and E5M2), E4M3, and E5M2 based computation units.

The results for 8 × 8, 16 × 16 and 32 × 32 accelerators are shown in Figure 13. Both Posit8 and FP8 demonstrate considerable advantages over BFloat16, reducing area by 30% and 34%, and power consumption by 26% and 32% on average, respectively. Compared to the hybrid FP8 accelerator, Posit8 accelerator features a smaller vector unit due to the use of posit approximation, making the overall vector unit 33% smaller and 35% lower power as shown in Table 8. Despite the smaller vector unit, FP8 overall has an area and power advantage over Posit8 due to its smaller MAC, which is a result of having one less fraction bit. We could potentially improve the area and power efficiency of the Posit8 accelerator by employing the same MAC hardware as FP8, by



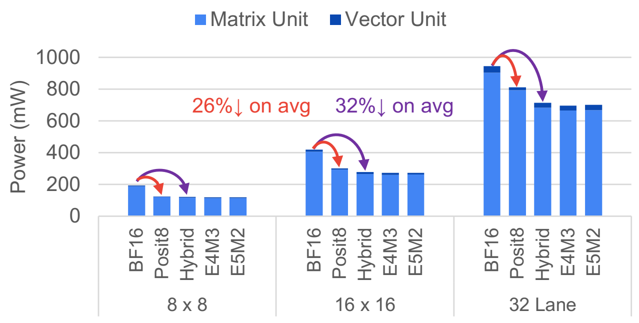


Figure 13. Accelerator standard cell plus SRAM macro area and post-synthesis power at 200 MHz and 0.9V. Hybrid refers to hybrid FP8 which can accommodate both E4M3 and E5M2.

truncating one fraction bit during decoding process. Posit8 would still maintain the advantage of superior range and similar precision compared to FP8.

Size	Area (mm^2)			Power (mW)		
	Posit8	FP8	%↓	Posit8	FP8	%↓
8-lane	0.208	0.304	31.5%	4.27	5.76	25.9%
16-lane	0.322	0.497	35.2%	7.8	13.1	40.5%
32-lane	0.572	0.840	32.0%	18.7	30.9	39.5%
Average	-	-	33%	-	-	35%

Table 8. Vector unit metrics for Posit8 and FP8 accelerators.

7.4 Fine-tuning Memory

We ran MobileBERT$_{tiny}$ on our accelerator and evaluated the impact of LoRA and 8-bit quantization on fine-tuning memory. We used a sequence length of 128, a batch size of 16, and AdamW optimizer. As illustrated in Figure 14, LoRA significantly reduces the amount of trainable parameters, thereby reducing the memory for weight gradient and optimizer states at the cost of a slight increase in total parameters. However, Transformer training memory is primarily dominated by activations especially with larger batch sizes. Implementing 8-bit quantization cuts the memory requirements for activations and weights by almost 50%, further reducing

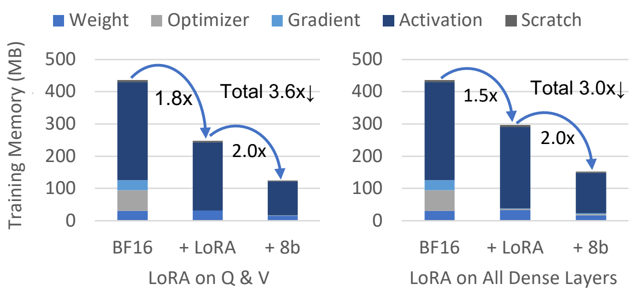


Figure 14. MobileBERT$_{tiny}$ fine-tuning memory reduction after applying LoRA and 8-bit quantization. Error stands for activation gradient during backward pass.

the fine-tuning memory. Overall, 8-bit LoRA fine-tuning can on average reduce training memory by approximately 3×.

8 Conclusions

We explore the application of Posit8 and FP8 quantization to Transformer inference and fine-tuning for edge accelerators. To the best of our knowledge, this paper is the first to systematically explore quantization of all Transformer operations beyond GEMM. We employ operation fusion to reduce the post-training quantization accuracy loss, simultaneously enhancing the fine-tuning accuracy. Our improvements to 8-bit LoRA, adapted for both FP8 and Posit8, enable us to utilize more efficient 8-bit MAC operations and conduct LoRA with a single data type without compromising accuracy. Furthermore, we design an area- and power-efficient posit softmax to compensate for the larger posit MAC unit. Our experiments on GLUE and SQuAD 1.1 demonstrate that both FP8 and Posit8 achieve accuracy mostly within 1% of the results obtained with BFloat16. We also show that this conclusion can be extended to larger and more complex Transformer models, such as Whisper and LLaMA 2. FP8 and Posit8 emerge as robust solutions for conducting Transformer inference and training on edge accelerators, offering efficient hardware efficiency with minimal accuracy compromise.

9 Acknowledgement

This work was supported by funding from SRC/DARPA JUMP 2.0 CoCoSys Center, SRC AIHW, AI Chip Center for Emerging Smart Systems (ACCESS), Hong Kong SAR, Precourt Institute for Energy, Apple Stanford EE PhD Fellowship in Integrated Systems, Samsung, and NSF FuSe-TG (award number: 2235462).

A Artifact Appendix

A.1 Abstract

This appendix describes the steps to set up and conduct Posit8 and FP8 inference and fine-tuning experiments on various Transformer models. Additionally, it guides on reproducing the quantitative results presented in the paper. For executing these artifacts, a CUDA-enabled GPU is necessary. The experiments also require Python 3.9+ and PyTorch 2.0+. The inference and fine-tuning results are expected to align closely with the figures reported in the paper. However, minor discrepancies up to half a percent may occur due to variations in hardware and software.

A.2 Artifact check-list (meta-information)

- **Models:** MobileBERT (15M & 25M), DistillBERT (66M), BERT (110M & 340M), Whisper (39M & 244M & 1.6B), GPT-2 (762M & 1.5B), and LLaMA2 (7B & 13B). All the models except MobileBERT$_{\text{tiny}}$ are available from Hugging Face and will be downloaded when running inference or fine-tuning scripts. MobileBERT$_{\text{tiny}}$ is accessible from the artifact code repository.
- **Datasets:** GLUE, SQuAD v1.1, LibriSpeech, and WikiText-103. All the datasets will be downloaded from Hugging Face when running the script.
- **Run-time environment:** Python 3.9+ and PyTorch 2.0+.
- **Hardware:** CUDA-enabled GPU with 16 GB of dedicated VRAM for most models, and 32 GB of VRAM (can be split across two GPUs) for LLMs.
- **Metrics:** GLUE tasks use accuracy (higher is better), SQuAD uses F1 score (higher is better), speech recognition uses word error rate (WER) (lower is better), and LLMs use loss or perplexity (lower is better).
- **Output:** Numerical results, e.g. accuracy, are printed in the log file, whose location can be specified by the command line arguments.
- **Experiments:** The README.md file in the GitHub repository provides detailed instructions for running all experiments. The results may vary depending on the specific hardware and software versions used. Variations of up to a few tenths of a percent (0.1%) from the figures reported in the paper are considered acceptable.
- **How much disk space required (approximately)?:** 128 GB of disk space is enough for all experiments.
- **How much time is needed to prepare workflow (approximately)?:** About 10 to 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** The time required for experiments varies depending on the specific task and hardware used. On a machine with an NVIDIA RTX 4090 GPU, one quantized inference experiment usually completes within 10 minutes, while fine-tuning experiments can take from 30 minutes to more than a day, depending on the size of the model and the dataset.
- **Publicly available?:** Yes, the artifact code is available on GitHub: https://github.com/jeffreyyu0602/quantized-training.
- **Code licenses (if publicly available)?:** MIT License.

A.3 Description

A.3.1 How to access.

The code repository for this submission can be downloaded from https://github.com/jeffreyyu0602/quantized-training.

A.3.2 Hardware dependencies.

CUDA-enabled NVIDIA GPUs and a machine with > 32 GB memory.

A.3.3 Software dependencies.

Evaluation of artifacts requires a machine with Python 3.9+ and PyTorch 2.0+ installed. We tested the artifacts on Ubuntu 23.10 with Python 3.9 and PyTorch 2.0, and found them to work.

A.3.4 Datasets.

We evaluate MobileBERT and BERT models using the GLUE and SQuAD v1.1 datasets, LLMs (e.g., LLaMA2 and GPT-2) on the WikiText dataset, and Whisper models on the LibriSpeech dataset. These datasets are available from Hugging Face and will be downloaded when running inference or fine-tuning scripts.

A.3.5 Models.

For inference experiments, we use models trained by either the original authors or third parties, except for MobileBERT$_{\text{tiny}}$ which we fine-tuned ourselves and uploaded to the artifact code repository. These models can be found on the Hugging Face website with links provided below, and will be automatically downloaded when running the quantized inference experiments.

- MobileBERT fine-tuned by third-party on SQuAD v1.1
- DistillBERT$_{\text{base}}$ fine-tuned by Hugging Face on SQuAD v1.1
- BERT$_{\text{base}}$ fine-tuned by third-party on SQuAD v1.1
- BERT$_{\text{large}}$ fine-tuned by Google on SQuAD v1.1
- Whisper trained on 680k hours of labelled speech data by OpenAI
- GPT-2 pretrained on a very large corpus of English data by OpenAI
- LLaMA2 pre-trained on a very large corpus of English data by Meta

To access LLaMA2, users must first request access through Meta's official website. Following this, users need to create an account and apply for access to the model checkpoints on the Hugging Face website. After gaining access, users can log in from the terminal through Hugging Face CLI, as detailed in Hugging Face documentation.

For fine-tuning experiments, we use pre-trained model checkpoints provided by Hugging Face, except for MobileBERT$_{\text{tiny}}$ which is accessible from the artifact code repository.

- MobileBERT from Google
- RoBERTa$_{\text{base}}$ from Meta
- RoBERTa$_{\text{large}}$ from Meta

A.4 Installation

First, ensure Python 3.9+ is installed on the system. For installing PyTorch, please refer to the official PyTorch website, since installation commands vary by environment. Clone the quantized-training artifact repository to the local machine:

```
$ git clone https://github.com/jeffreyyu0602/
    quantized-training
$ cd quantized-training
$ mkdir logs
```

Then, install quantized-training and all the required packages listed in `requirements.txt`.

```
$ pip install -e .
$ pip install -r requirements.txt
```

A.5 Experiment workflow

A set of Python scripts will generate the numbers and tables in the paper. The complete instructions can be found in the README.md file included within the artifact repository.

A.6 Evaluation and expected results

A.6.1 SQuAD inference experiments. Table 2 consists of quantized inference results (F1 scores) on SQuAD v1.1 for MobileBERT$_{\text{tiny}}$, MobileBERT, BERT$_{\text{base}}$, BERT$_{\text{large}}$, and DistillBERT$_{\text{base}}$. The results can be replicated by running the following commands:

```
python examples/question_answering/run_squad.py [--
    log_file <PATH_TO_LOG_FILE>]
```

By default, the log file is saved as `logs/squad.log`. The script will produce a `squad_f1.csv` which has the same format as Table 2.

The inference results will be affected by the specific device the experiments are run on, CUDA version, and software versions (e.g. PyTorch version). Therefore, it is very hard to get bit-wise identical results. The F1 scores in `squad_f1.csv` may differ slightly from those in Table 2, with variations of up to a few tenths of a percent (0.1%).

A.6.2 Fine-tuning experiments. Table 7 presents the results of fine-tuning MobileBERT$_{\text{tiny}}$, MobileBERT, RoBERTa$_{\text{base}}$, and RoBERTa$_{\text{large}}$ on the GLUE and SQuAD v1.1 datasets. The hyperparameters used for training—such as batch size, learning rate, number of training epochs, and LoRA parameters—are specified in the training script `run_quantized_training.py`. Users can reproduce all the results using the commands detailed in `asplos_experiments.sh`. The commands have the following format:

```
python run_quantized_training.py --model <MODEL> --
    task <TASK> --run_job <JOB_NAME> [--seed 42] [--
    log_file <LOG_FILE>] [--op_fusion <LAYER>]
```

Available models for selection include:

- `models/mobilebert_tiny`
- `google/mobilebert-uncase`
- `roberta-base`
- `roberta-large`

For task, options include `mnli`, `qnli`, `mrpc`, `sst2`, and `squad`. For job name, selection options are:

- `bf16`: run BFloat16 training
- `posit8`: run Posit8 quantized training
- `posit8-approx-shifted`: run Posit8 quantized training with posit approximation
- `fp8`: run FP8 quantized training

Users can optionally specify a random seed for reproducibility and a filename to redirect logs to.

For example, MobileBERT LoRA Posit8 results on MRPC can be replicated by running:

```
python run_quantized_training.py --model google/
    mobilebert-uncased --task mrpc --run_job posit8
    --seed 42 --log_file logs/mobilebert-mrpc.log
```

The above command takes approximately 30 minutes to run on an NVIDIA RTX 4090 GPU. The final accuracy should match closely with the figure in Table 7.

Low-precision training can sometimes lead to numerical instability and non-finite gradients. In such instances, users have several options: resuming training from a previous checkpoint, lowering the learning rate, or selecting a different random seed. Additionally, when fine-tuning MobileBERT with Posit8 or FP8, training stability can be achieved by fusing the last layer. For GLUE tasks, the relevant option is `--op_fusion classifier`, and for SQuAD the option is `--op_fusion qa_outputs`.

Similar to inference experiments, replicating bit-wise identical results is challenging due to variations in hardware and software. For fine-tuning, conducting multiple runs with different random seeds is advised to eliminate outlier runs. Variations in results of up to half a percent (0.5%) are considered normal.

References

[1] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks, 2018. arXiv:1805.11046.

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. arXiv:2005.14165.

[3] Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Alberto Nannarelli, Marco Re, and Sergio Spanò. A pseudo-softmax function for hardware-based high speed image classification. *Scientific Reports*, 11(1):15307, 2021. doi:10.1038/s41598-021-94691-7.

[4] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers, 2020. arXiv:2005.12872.

[5] Xinlei Chen, Saining Xie, and Kaiming He. An empirical study of training self-supervised vision transformers, 2021. arXiv:2104.02057.

[6] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. Fast approximations of activation functions in deep neural networks when using posit arithmetic. *Sensors*, 20(5):1515, 2020. doi:10.3390/s20051515.

[7] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale, 2022. arXiv:2208.07339.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. URL: https://aclanthology.org/N19-1423, doi:10.18653/v1/N19-1423.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. arXiv:2010.11929.

[10] John Gustafson and Ivan Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J.*, 4(2):71–86, June 2017. doi:10.14529/jsfi170206.

[11] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015. arXiv:1503.02531.

[12] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR, June 2019. URL: https://proceedings.mlr.press/v97/houlsby19a.html.

[13] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. arXiv:2106.09685.

[14] Jeff Johnson. Rethinking floating point for deep learning, 2018. arXiv:1811.01721.

[15] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized bert pretraining approach, 2019. arXiv:1907.11692.

[16] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin Transformer: Hierarchical vision transformer using shifted windows, 2021. arXiv:2103.14030.

[17] Jinming Lu, Chao Fang, Mingyang Xu, Jun Lin, and Zhongfeng Wang. Evaluations on deep neural networks training using posit number system. *IEEE Transactions on Computers*, 70(2):174–187, 2021. doi:10.1109/TC.2020.2985971.

[18] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018. arXiv:1710.03740.

[19] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. FP8 formats for deep learning, 2022. arXiv:2209.05433.

[20] Raul Murillo, Alberto A. Del Barrio, Guillermo Botella, Min Soo Kim, HyunJin Kim, and Nader Bagherzadeh. PLAM: A posit logarithm-approximate multiplier. *IEEE Transactions on Emerging Topics in Computing*, 10(4):2079–2085, 2022. doi:10.1109/TETC.2021.3109127.

[21] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. LibriSpeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015. doi:10.1109/ICASSP.2015.7178964.

[22] Kartik Prabhu, Albert Gural, Zainab F. Khan, Robert M. Radway, Massimo Giordano, Kalhan Koul, Rohan Doshi, John W. Kustin, Timothy Liu, Gregorio B. Lopes, Victor Turbiner, Win-San Khwa, Yu-Der Chih, Meng-Fan Chang, Guénolé Lallement, Boris Murmann, Subhasish Mitra, and Priyanka Raina. CHIMERA: A 0.92-TOPS, 2.2-TOPS/W edge AI accelerator with 2-MByte on-chip foundry resistive RAM for efficient training and inference. *IEEE Journal of Solid State Circuits*, 57(4):1013–1026, 2022. doi:10.1109/JSSC.2022.3140753.

[23] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision, 2022. arXiv:2212.04356.

[24] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL: http://jmlr.org/papers/v21/20-074.html.

[25] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In Jian Su, Kevin Duh, and Xavier Carreras, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics. URL: https://aclanthology.org/D16-1264, doi:10.18653/v1/D16-1264.

[26] Gonçalo Raposo, Pedro Tomás, and Nuno Roma. PositNN: Training deep neural networks with mixed low-precision posit. In *2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7908–7912, 2021. doi:10.1109/ICASSP39728.2021.9413919.

[27] Robin Strudel, Ricardo Garcia, Ivan Laptev, and Cordelia Schmid. Segmenter: Transformer for semantic segmentation, 2021. arXiv:2105.05633.

[28] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. MobileBERT: A compact task-agnostic BERT for resource-limited devices. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2158–2170, Online, July 2020. Association for Computational Linguistics. URL: https://aclanthology.org/2020.acl-main.195, doi:10.18653/v1/2020.acl-main.195.

[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: https://doi.org/10.48550/arXiv.1706.03762.

[30] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. GLUE: A multi-task benchmark and analysis platform for natural language understanding. In Tal Linzen, Grzegorz Chrupała, and Afra Alishahi, editors, *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 353–355, Brussels, Belgium, November 2018. Association for Computational Linguistics. URL: https://aclanthology.org/W18-5446, doi:10.18653/v1/W18-5446.

[31] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. HuggingFace's Transformers: State-of-the-art natural language processing, 2020. arXiv:1910.03771.

[32] Haocheng Xi, Changhao Li, Jianfei Chen, and Jun Zhu. Training transformers with 4-bit integers, 2023. arXiv:2306.11987.

[33] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models, 2023. arXiv:2211.10438.

[34] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked

language-models, 2022. arXiv:2106.10199.