Towards Fine-tuning Pre-trained Language Models with Integer Forward and Backward Propagation

Mohammadreza Tayaranian[*1] **Alireza Ghaffari**[*1] **Marzieh S. Tahaei**[1]
Mehdi Rezagholizadeh[1] **Masoud Asgharian**[2] **Vahid Partovi Nia**[1]
[1] Huawei Noah's Ark Lab, Montreal Research Center
[2] Department of Mathematics and Statistics, McGill University
{mohammadreza.tayaranian, alireza.ghaffari, marzieh.tahaei}@huawei.com
{mehdi.rezagholizadeh, vahid.partovinia}@huawei.com
masoud.asgharian2@mcgill.ca

Abstract

The large number of parameters of some prominent language models, such as BERT, makes their fine-tuning on downstream tasks computationally intensive and energy hungry. Previously researchers were focused on lower bit-width integer data types for the forward propagation of language models to save memory and computation. As for the backward propagation, however, only 16-bit floating-point data type has been used for the fine-tuning of BERT. In this work, we use integer arithmetic for both forward and back propagation in the fine-tuning of BERT. We study the effects of varying the integer bit-width on the model's metric performance. Our integer fine-tuning uses integer arithmetic to perform forward propagation and gradient computation of linear, layer-norm, and embedding layers of BERT. We fine-tune BERT using our integer training method on SQuAD v1.1 and SQuAD v2., and GLUE benchmark. We demonstrate that metric performance of fine-tuning 16-bit integer BERT matches both 16-bit and 32-bit floating-point baselines. Furthermore, using the faster and more memory efficient 8-bit integer data type, integer fine-tuning of BERT loses an average of 3.1 points compared to the FP32 baseline.

1 Introduction

Over the past few years, integration of attention mechanisms into deep learning models led to the creation of transformer based models. BERT (Devlin et al., 2018) is a prominent transformer based language model which has shown state-of-the-art performance in natural language processing (NLP) tasks.

BERT requires high memory and computational resources due to its large number of parameters. Having large number of parameters incurs challenges for inference, training, and also fine-tuning of this model. Moreover, the training phase i.e. pre-training and fine-tuning, involves more operations compared to the inference. More specifically, the training phase includes gradient computation and weight update that make the training more computationally intensive.

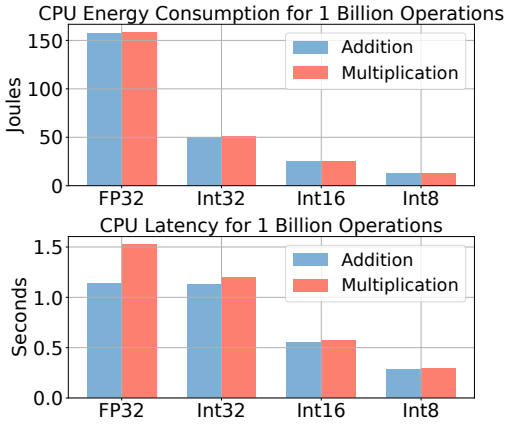

Figure 1: Energy consumption and latency of 1 billion operations using various data types, measured on an Intel® Xeon® CPU E5-2698 v4.

One method of reducing the computational complexity of deep learning models is to represent their parameters and activations in low bit-width data types. This reduces the memory footprint of the model and enables more efficient computations. For instance, Figure 1 shows that low-bit integer data types have higher throughput and better energy consumption compared to floating-point.

Previous research attempts at integer quantization of transformer based language models were only focused on forward propagation and the gradient computation were kept in 32-bit floating-point data type (FP32) (Bhandare et al., 2019; Kim et al., 2021; Zafrir et al., 2019).

Furthermore, earlier efforts for using low bit-width data types for gradient computation of transformer based language models has only been limited to 16-bit floating-point (FP16). This method,

[*]Equal contribution.

Findings of the Association for Computational Linguistics: EACL 2023, pages 1912–1921

known as mixed precision training (Micikevicius et al., 2017), uses FP16 data type to represent weights, activations and gradients while using FP32 for the weight update.

Here we present an integer fine-tuning method for transformer based language models such as BERT. Unlike previous works, we use integer data types for both forward propagation and gradient computation during the fine-tuning of BERT. Moreover, we use the dynamic fixed-point format to represent floating-point numbers as integers.

Our integer mapping strategy can be used alongside floating-point numbers in fine-tuning and inference similar to mixed precision training. In our proposed strategy, the arithmetic of all the compute intensive layers for both forward and back propagation are performed using integer arithmetic while other components of the model, such as nonlinear functions and the weight updates are kept in FP32. We use integer versions of compute intensive layers such as linear, normalization (layer-norm), and embedding layers.

We study the effect of various bit-widths of the integer input activation and show that increasing the bit-width of the fixed-point mapping function improves the convergence behaviour of the model. This enables us to find the minimum bit-width required for integer fine-tuning of BERT.

Our fine-tuning experiments show that 16-bit integer BERT is able to match the metric performance of mixed precision FP16 and FP32 methods.

We also further reduce the bit-widths and show that integer fine-tuning of BERT with 8-bit integer weights and 12-bit integer activations has a score drop of 3.1 compared to the original performance.

To summarize, this paper makes the following contributions:

- Integer fine-tuning of transformer based language models that uses integer arithmetic for both the forward and back propagation of compute intensive layers such as linear, layer-norm, and embedding. To the best of our knowledge, this is the first time that integer data type is used for back propagation of pre-trained language models.

- Analyzing the effect of changing the bit-width of dynamic fixed-point format on the convergence of fine-tuning. **Remark 3** discusses that the convergence behaviour of our integer fine-tuning is directly related to the variance of dynamic fixed-point mapping and is controlled by the bit-width.

- We show that fine-tuning BERT using 16-bit integer numbers is able to outperform the FP16 mixed precision fine-tuning method.

The rest of this paper is structured as follows. Section 2 briefly discusses previous works in which low bit-width data types are used for inference and training of deep learning models. Section 3 provides details of our integer fine-tuning method, including the representation mapping functions and integer-only layers. The convergence behaviour of the dynamic fixed-point mapping is studied in Section 4 by providing empirical observations and theoretical analysis. The fine-tuning experiments on various integer and floating-point setups are presented in Section 5. Finally, Section 6 concludes the ideas proposed in this work.

2 Related Works

In this section we discuss the previous works that use low bit-width data types in transformer based language models. These works could be categorized into two major groups. In the first group, called low-bit inference, the low bit-width data types are used only in the forward propagation phase to improve computational complexity and reduce memory usage during the inference. In the second group, also known as low-bit training, lower bit-width data types are used for both the forward and back propagation phases.

2.1 Low-bit Inference

Previous research on low-bit inference quantize the model parameters and activations to speed up the forward propagation. This category is itself divided into quantization-aware training (QAT) and post-training quantization (PTQ) methods.

In QAT, quantization is performed during training, allowing the model parameters to adapt to the quantization noise. QAT relies on high-precision FP32 gradients to train the model and adapt it to the quantization noise.

For instance, (Zafrir et al., 2019) proposed Q8BERT which quantizes the inference computations of all linear and embedding layers of BERT to 8-bit integers and updates the quantization scale with a moving average. Similarly, (Shen et al., 2020) suggested Q-BERT which requires the computation of hessian matrix for each group of param-

eters to be used in a mixed precision fine-tuning with different bit-widths. (Kim et al., 2021) proposed I-BERT that uses a uniform quantization scheme to quantize input activations and weights of various components of BERT. In I-BERT, the quantization scaling factors are computed based on the distribution of the training data.

Unlike QAT that performs quantization of inference operations during training, Post-Training Quantization (PTQ) methods apply quantization to the parameters when the training is completed. Thus, they require extra calibration or parameter tuning to adapt the model to the quantized parameters.

For instance, (Bhandare et al., 2019) quantized the matrix multiplications of the original transformer architecture from (Vaswani et al., 2017) to 8-bit integer data type. Moreover, the quantization is done only for the forward propagation and requires extra calibration using validation data to tune the boundaries of the quantization function. (Zadeh et al., 2020) introduced GOBO which compresses the fine-tuned weights of BERT by grouping them into two categories of Gaussian and outlier. The outlier weights are kept in FP32, while the Gaussian weights are quantized to lower bits. For lower bit-width regimes, TernaryBERT and BinaryBERT are able to push the quantization to 2 and 1 bits respectively (Zhang et al., 2020a; Bai et al., 2020). They both rely on methods such as data augmentation and knowledge distillation to adapt the model to the low-bit weights.

2.2 Low-bit Training

Research on low-bit training try to perform both the forward propagation and gradient computation in low-bit arithmetic. Using low precision number formats for gradients reduces the model's ability to adapt the parameters to the quantization noise, but increases the throughput and reduces the memory footprint.

FP16 mixed precision training (Micikevicius et al., 2017) is a common method currently for low-bit fine-tuning of transformer based language models. This method uses FP16 data type in both forward propagation and gradient computation, while using FP32 for the weight update. Unlike FP16 mixed precision training, our work uses dynamic fixed-point format which allows for multiple choices of bit-width for the data type. We show that our 16-bit integer fine-tuning method outperforms FP16 mixed precision training in terms of metric score.

Using integer data types in the training of deep learning models has been previously studied for the computer vision tasks. For instance, (Zhang et al., 2020b) quantized the input activations, gradients and parameters of the linear layers for various convolutional neural networks (CNN). Similarly, (Zhao et al., 2021) adapted the quantization parameters by detecting the distribution of the gradients in the channel dimension. In both these works the quantization error is measured during training and is used to adjust the quantization scale, whereas our method does not require any information about distribution of data or gradients. (Zhu et al., 2020) applied a quantization scheme to train CNN architectures with "direction sensitive gradient clipping" and learning rate scaling to control the quantization error of gradients. Our integer fine-tuning method does not require gradient clipping and can follow the same loss trajectory as the floating-point baseline with the same hyper-parameters. Our proposed method improves upon (Ghaffari et al., 2022) which uses dynamic fixed-point format for integer training of deep learning models. Unlike (Ghaffari et al., 2022), our work studies various bit-widths for both weights and activations to find the minimum bit-width required for fine-tuning BERT. Furthermore, we study integer training method on large language models where low-bit quantization is known to be a challenging task (Bondarenko et al., 2021). To the best of our knowledge, this is the first time where integer numbers are used for the back propagation of transformer based language models.

3 Methodology

3.1 Representation Mapping

We use the dynamic fixed-point format (Williamson, 1991) to map the floating-point numbers to integer data type. This format, also known as block floating-point, maps floating-point numbers to blocks of integer numbers, with each block having its unique scale. For more information on various number formats refer to Appendix A.

We use a linear fixed-point mapping function to map floating-point numbers to integer numbers. The linear fixed-point mapping converts a floating-point tensor $\mathbf{F}$ to a tensor of integers and a single scale factor.

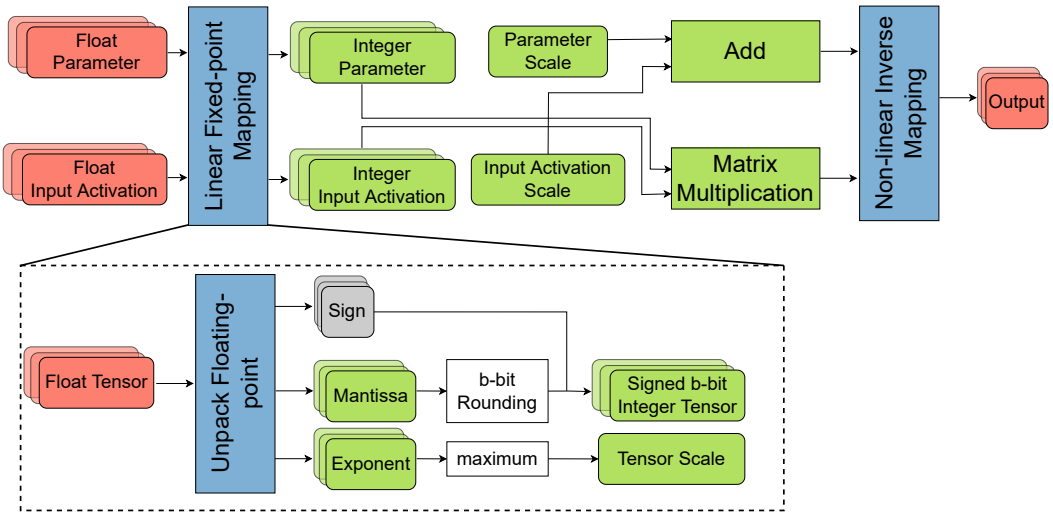

Figure 2: Forward propagation operations in an integer-only linear layer. Green boxes use integer arithmetic and red boxes use floating-point data type. Here, the integer output is generated using an integer matrix multiplication and the output scale is generated by a single add operation. The bottom panel shows the linear fixed-point mapping for the input tensors, that are the input activation and the parameter tensor in this figure.

The integers are obtained by rounding the floating-point mantissas. The scale is the maximum of the floating-point exponents of $\mathbf{F}$. The bottom section of Figure 2 shows the internal operations of the linear fixed-point mapping.

To map the fixed-point numbers to floating-point, a non-linear inverse mapping function is used. The inverse mapping converts integer numbers into normalized floating-point mantissas and packs each integer with its corresponding scale into a floating-point number.

Details of the representation mapping functions are provided in (Ghaffari et al., 2022). Our methodology differs in that it includes various bit-widths for both weights and activations for the fine-tuning of transformer based language models. We exploit this mapping strategy to explore various bit-widths for weights and activations in order to find the minimum bit-width for fine-tuning the model.

3.2 Integer Fine-tuning

Our method uses integer arithmetic for weights, activations and gradients, while the weight update is kept in FP32. Moreover, our proposed BERT setups use integer-only versions for all the linear, layer-norm and embedding layers in which internal operations are performed with integer arithmetic.

3.2.1 Linear Layer

Figure 2 depicts a high-level view of forward propagation operations of the integer-only linear layer. All the parameters and activations of the layer are first mapped to dynamic fixed-point using the linear fixed-point mapping function. In the case of linear layer, the integer parameters and input activations are then sent to an integer matrix multiplication function to generate the integer output. If needed, the integer output could be mapped back to floating-point to be used by other layers of the model using the non-linear inverse mapping.

For back propagation, the gradients of the parameters and input activations are also computed using integer arithmetic. Using integer matrix multiplication, the output gradients are multiplied by input activations and parameters to compute the gradients. Since the weight update is performed in FP32, the integer gradients and their scales are passed to the non-linear inverse mapping to be mapped to FP32.

3.2.2 Layer-norm

The layer normalization or layer-norm performs the following operation on its input X (Ba et al., 2016):

$$\gamma \frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta. \quad (1)$$

Here γ and β are the weight and bias parameters, and σ and μ are input standard deviation and mean respectively. For the forward propagation of integer layer-norm we map X to dynamic fixed-point format and compute σ and μ using integer arithmetic. Note that multiplication to γ and addition with β are also performed using integer arithmetic.

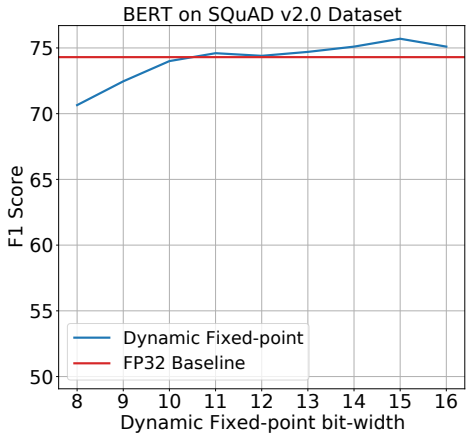

Figure 3: F1 score of fine-tuning BERT using b-bit gradients, and activations on SQuAD v2.0 dataset. For the 8-bit and 9-bit fixed-point bit-widths, we use 12-bit input activations.

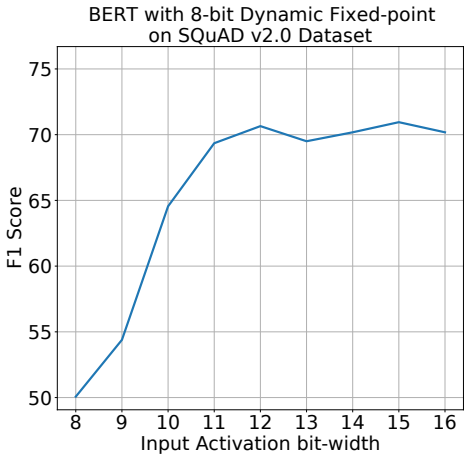

Figure 4: F1 score of fine-tuning BERT using 8-bit weights and gradients, with varying input activation bit-width on SQuAD v2.0 dataset. Note that **Remark 3** justifies this experiment using the variance of b-bit dynamic fixed-point mapping.

Moreover, the back propagation also uses integer arithmetic to compute the gradients for the input, γ, and β.

3.2.3 Embedding Layer

The embedding layer is a lookup table that stores embeddings. The layer takes a list of indices as input and returns the list of corresponding embeddings for each index. The integer embedding layer, handles integer embeddings and needs less memory footprint to store these values. For the back propagation, the embedding layer applies the output integer gradients directly to each corresponding row of the lookup table.

4 Convergence Behaviour of Dynamic Fixed Point Mapping

4.1 Empirical Observations

Figure 2 shows that the bit-width, b, is controlled by adjusting the number of rounded bits in the rounding function. Here we study the effect of changing the integer bit-width on the metric performance of the model.

The motivation of varying the bit-width of the dynamic fixed-point is to control the variance induced by the linear fixed-point mapping. Our experiments show that using dynamic fixed-point with a bit-width of 10 achieves the same performance as the FP32 fine-tuning method. Figure 3 demonstrates the F1 score of fine-tuning BERT on SQuAD v2.0 dataset against the fixed-point bit-width. Note that the fixed-point arithmetic with a bit-width higher than 10 bits is able to closely match the F1 score of the FP32 baseline, that is indicated by the red line in the figure. Also note that in our experimental setup for the 8-bit dynamic fixed-point format, we use 12-bit input activations to close the F1 score gap with the FP32 baseline. The reason for using higher bit-width input activations is that we observed 8-bit activation dramatically reduces the F1 score. Figure 4 shows the effect of input activation bit-width on the F1 score when the weights are 8-bit integers. Changing the bit-width of the input activation from 8 bits to 12 bits significantly increases the F1 score. Increasing the input activation bit-width beyond 12 bits has a negligible effect on the F1 score, confirming that 12 bits is the minimum required bit-width of the input activations for this application with 8-bit integer weights.

4.2 Theoretical Analysis

Here, we study the effect of varying dynamic fixed-point mapping bit-width on the stochastic gradient descent method. The goal is to show the relation of weight and activation bit-widths on the convergence of integer training. Let us consider the following simplified weight update equation

$$w_{k+1} = w_k + \bar{\eta}\hat{g}(w_k, \xi_k), \tag{2}$$

where $\hat{g}(w_k, \xi_k)$ is the dynamic fixed-point gradient and $\bar{\eta}$ is the learning rate during the fine-tuning phase. Furthermore, we also consider the following common assumptions in sequel.

Assumption 1 (Lipschitz-continuity). The loss function $\mathcal{L}(w)$ is continuously differentiable and its gradients satisfies the following inequality where $L > 0$ is the Lipchitz constant

$$\mathcal{L}(w) \leqslant \mathcal{L}(\bar{w}) + \nabla\mathcal{L}(\bar{w})^\top (w - \bar{w}) + \frac{1}{2} L ||w - \bar{w}||_2^2; \quad \forall\, w, \bar{w} \in \mathbb{R}^d. \tag{3}$$

Assumption 2. (i) $\mathcal{L}(w_k)$ is bounded. (ii) b-bit dynamic fixed-point gradients $\hat{g}(w_k, \xi_k)$ is an unbiased estimator of the true gradients of the loss function $\nabla\mathcal{L}(w_k)^\top \mathbb{E}_{\xi_k}\{\hat{g}(w_k, \xi_k)\} = ||\nabla\mathcal{L}(w_k)||_2^2 = ||\mathbb{E}_{\xi_k}\{\hat{g}(w_k, \xi_k)\}||_2^2$, and (iii) with the b-bit dynamic fixed-point gradients i.e. $\hat{g}(w_k, \xi_k)$, there exist scalars $M \geqslant 0$, $M_V \geqslant 0$, $M^q \geqslant 0$ and $M_V^q \geqslant 0$ such that for all iterations of SGD

$$\mathbb{V}_{\xi_k}\{\hat{g}(w_k, \xi_k)\} \leqslant M + M^q + (M_V + M_V^q)||\nabla\mathcal{L}(w_k)||_2^2.$$

Where M^q and M_V^q denote the added variance of b-bit dynamic fixed-point mapping on the true gradient variance. Also note that in order for **Assumption 2** (i) to hold true, we use stochastic rounding for back propagation.

Suppose **Assumption 1** and **Assumption 2** are true, then inequality (4) follows from (Ghaffari et al., 2022, Remark 2)

$$\mathbb{E}_{\xi_k}\{\mathcal{L}(w_{k+1})\} - \mathcal{L}(w_k) \leqslant -(1 - \frac{1}{2}\bar{\eta}L(M_G + M_G^q))\bar{\eta}||\nabla\mathcal{L}(w_k)||_2^2 + \frac{1}{2}\bar{\eta}^2 L(M + M^q), \quad \text{with } M_G := 1 + M_V \text{ and } M_G^q := 1 + M_V^q, \tag{4}$$

which shows the effect of added variance of fixed point mapping, i.e. M_V^q and M^q, on each step of the optimizer.

Remark 1. In inequality (4), the first term, $-(1 - \frac{1}{2}\bar{\eta}L(M_G + M_G^q))\bar{\eta}||\nabla\mathcal{L}(w_k)||_2^2$ contribute to decreasing the loss $\mathcal{L}$ while the second term, $\frac{1}{2}\bar{\eta}^2 L(M + M^q)$, prevents it. Also note that when M^q and M_G^q are increased, they negatively affect the descent of the loss $\mathcal{L}$. This means for a good convergence behaviour, representation mapping variance bounds, i.e. M^q and M_G^q, must be controlled.

Remark 2. For dynamic fixed-point mapping with b-bit integers, the representation mapping variance bounds i.e. M^q and M_G^q, are closely related to the bit-width b. Here, we study these two constants for a linear layer. Let us denote $\hat{\mathbf{A}}$ as the b-bit dynamic fixed-point version of tensor $\mathbf{A}$ and $\hat{a}_{ij}$ as its ij^{th} element. We can relate $\hat{a}_{ij}$ and a_{ij} with an error term δ such as $\hat{a}_{ij} = a_{ij} + \delta_{ij}^{\mathbf{A}}$. For a linear layer $\hat{\mathbf{Y}} = \hat{\mathbf{X}}\hat{\mathbf{W}}$, the computation of the b-bit dynamic fixed-point gradients in the back propagation is

$$\hat{\mathbf{C}} = \frac{\partial\hat{\mathbf{L}}}{\partial\hat{\mathbf{W}}} = \frac{\partial\hat{\mathbf{Y}}}{\partial\hat{\mathbf{W}}}\frac{\partial\hat{\mathbf{L}}}{\partial\hat{\mathbf{Y}}} = \hat{\mathbf{X}}^\top\frac{\partial\hat{\mathbf{L}}}{\partial\hat{\mathbf{Y}}} = \hat{\mathbf{X}}^\top\hat{\mathbf{G}}. \tag{5}$$

It is of interest to find the relation between $\hat{\mathbf{C}} = \hat{\mathbf{X}}^\top\hat{\mathbf{G}}$ in the integer back propagation and the true gradients $\mathbf{C} = \mathbf{X}^\top\mathbf{G}$. We can derive the variance for each element $\hat{c}_{ij}$ by expanding the error terms δ,

$$\begin{aligned}
\mathbb{V}\{\hat{c}_{ij}\} &= \mathbb{V}\left\{\sum_{n=1}^{N}\hat{x}_{ni}\hat{g}_{nj}\right\} \\
&= \mathbb{V}\left\{\sum_{n=1}^{N}(x_{ni} + \delta_{ni}^{\mathbf{X}})(g_{nj} + \delta_{nj}^{\mathbf{G}})\right\} \\
&\leqslant \mathbb{V}\left\{\sum_{n=1}^{K} x_{ni}g_{nj}\right\} \\
&\quad + \sigma_{\mathbf{G}}^2\mathbb{E}\{||\mathbf{X}_{i.}^\top||_2^2\} + \sigma_{\mathbf{X}}^2\mathbb{E}\{||\mathbf{G}_{.j}||_2^2\} \\
&\quad + N\sigma_{\mathbf{X}}^2\sigma_{\mathbf{G}}^2 \\
&= \mathbb{V}\{c_{ij}\} + \sigma_{\mathbf{G}}^2\mathbb{E}\{||\mathbf{X}_{i.}^\top||_2^2\} \\
&\quad + \sigma_X^2\mathbb{E}\{||\mathbf{G}_{.j}||_2^2\} + N\sigma_{\mathbf{X}}^2\sigma_{\mathbf{G}}^2.
\end{aligned} \tag{6}$$

In inequality (6), $\sigma_{\mathbf{G}}^2 = \max_{i,j}(\mathbb{V}\{\delta_{i,j}^{\mathbf{G}}\})$ and $\sigma_{\mathbf{X}}^2 = \max_{i,j}(\mathbb{V}\{\delta_{i,j}^{\mathbf{X}}\})$. Also note $||\mathbf{X}_{i.}^\top||_2^2 = \sum_j^J x_{ji}^2$ denotes the squared L-2 norm of the i^{th} *row* of $\mathbf{X}^\top$ and $||\mathbf{G}_{.j}||_2^2 = \sum_i^I g_{ij}^2$ denotes the squared L-2 norm of the j^{th} *column* of $\mathbf{G}$. Furthermore, by defining

$$\begin{cases} M^q := \sigma_{\mathbf{G}}^2(\mathbb{E}\{||\mathbf{X}_{i.}^\top||_2^2\} + N\sigma_{\mathbf{X}}^2) \\ M_V^q := \sigma_{\mathbf{X}}^2 \end{cases} \tag{7}$$

Equation (7) shows that M^q depends on variance of dynamic fixed-point mapping for input activations and gradients while M_G^q only depends on b-bit dynamic fixed-point gradients variance.

	QQP	QNLI	MNLI	SST-2	STSB	RTE	MRPC	CoLA	Average
FP32	91.0/88.0	91.1	84.2	92.5	88.3	63.8	82.5/87.8	57.2	82.6
FP16 AMP	90.9/87.9	91.2	84.1	92.4	88.3	64	82.1/87.7	57.5	82.6
16-bit integer	**91.0/88.0**	91.2	**84.2**	92.5	**88.3**	**64.5**	**82.3/87.6**	**57.7**	**82.7**
12-bit integer	90.9/88.0	**91.2**	84.0	**92.6**	87.9	63.5	81.3/87.4	56.7	82.4
10-bit integer	90.8/87.8	91.0	84.0	92.5	87.5	62.7	78.4/85.8	57.6	81.8
8-bit integer	90.1/86.8	90.8	83.7	92.3	87	61.8	76.8/84.7	55.0	80.9

Table 1: Metric performance of integer fine-tuning of BERT on selected GLUE tasks. The reported metric for QQP and MRPC is accuracy and F1 score, for QNLI, MNLI, RTE, and SST-2 is accuracy, for STSB is the Pearson-Spearman correlation, and for CoLA is the Matthews correlation.

Proposition 1. For dynamic fixed-point representation of tensor $\hat{\mathbf{A}}$ with b-bit integers, the variance of error for element i satisfies the following inequality

$$\mathbb{V}\{\delta_i^{\mathbf{A}}\} \leqslant 2^{2(e_{\text{scale}_{\mathbf{A}}}-b+2)}. \tag{8}$$

Proof. Using dynamic fixed-point mapping to b-bit integers, the error $\delta_i^{\mathbf{A}}$ satisfies the following bound

$$-2^{e_{\text{scale}_{\mathbf{A}}}} \underbrace{(0.000001)_2}_{b-1} \leqslant \delta_i^{\mathbf{A}} \leqslant 2^{e_{\text{scale}_{\mathbf{A}}}} \underbrace{(0.000001)_2}_{b-1}$$
$$-2^{e_{\text{scale}_{\mathbf{A}}}-b+2} \leqslant \delta_i^{\mathbf{A}} \leqslant 2^{e_{\text{scale}_{\mathbf{A}}}-b+2}. \tag{9}$$

Thus, the inequality (8) is obtained by using Popoviciu's inequality on variances

$$\mathbb{V}\{\delta_i^{\mathbf{A}}\} \leqslant \frac{1}{4}(2^{e_{\text{scale}_{\mathbf{A}}}-b+2} - (-2^{e_{\text{scale}_{\mathbf{A}}}-b+2}))^2$$
$$\leqslant 2^{2(e_{\text{scale}_{\mathbf{A}}}-b+2)}. \tag{10}$$

Remark 3. Inequality (8) shows that increasing bit-width b in dynamic fixed-point mapping reduces the variance of the error. This confirms our experimental results on SQuAD v2.0 dataset that for $b > 10$, F1 score can match FP32 baseline, see Figure 3. Also note in equation (7), both M^q and M_V^q depend on b-bit dynamic fixed-point mapping variance of input activation $\sigma_{\mathbf{X}}^2$. Hence, increasing b for input activations while keeping weights in 8-bit format must improve the convergence behaviour. This phenomenon is also confirmed by our experimental results on SQuAD v2.0 dataset demonstrated in Figure 4.

5 Experimental Results

5.1 Experimental Setup

We fine-tuned BERT base on a series of downstream tasks to compare the performance of our integer fine-tuning method with FP16 and FP32 fine-tuning methods. FP16 AMP setup uses NVIDIA's automatic mixed precision[1] and the FP32 baseline is the default implementation from Pytorch.

	SQuAD v1.1	SQuAD v2
FP32	80.5/88.0	70.6/73.8
FP16 AMP	79.9/87.6	70.6/73.9
16-bit integer	**80.7/88.0**	**70.6/73.9**
12-bit integer	79.8/87.6	70.5/73.8
10-bit integer	78.4/86.6	69.8/73.2
8-bit integer	75.6/84.5	65.5/69.2

Table 2: Metric performance of fine-tuning BERT on SQuAD v1.1 and v2.0 datasets. For both datasets the exact match metrics and F1 scores are reported.

The model is fine-tuned on selected tasks of GLUE benchmark (Wang et al., 2018), along with the Stanford Question Answering Datasets, i.e. SQuAD v1.1 and SQuAD v2.0 (Rajpurkar et al., 2016).

All the fine-tuning setups use the same hyper-parameters and are fine-tuned for the same number of epochs. Each reported metric is the average of five runs with five different random seeds to mitigate the effects of random variation of the results. The fine-tuning experiments are performed based on the fine-tuning scripts of the Hugging Face library (Wolf et al., 2019). For GLUE experiments the fine-tuning is performed for 5 epochs and the learning rate is set to 2×10^{-5}. Also, the per-device fine-tuning batch-size is set to 32. Fine-tuning BERT on SQuAD datasets is done for 2 epochs and the learning rate is 5×10^{-5} and the per-device fine-tuning batch-size is 12. All experiments are run on eight NVIDIA V100 GPUs with 32 gigabytes of VRAM.

[1] https://developer.nvidia.com/automatic-mixed-precision

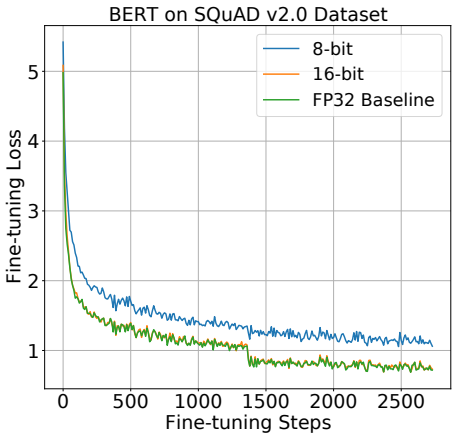

Figure 5: Integer fine-tuning loss trajectory of BERT on SQuAD v2.0 dataset for 2750 iterations.

5.2 Results

The results of fine-tuning BERT base on GLUE benchmark and SQuAD datasets are presented in Table 1 and Table 2 respectively. GLUE benchmark contains a series of downstream tasks, designed to evaluate a diverse set of language understanding abilities of NLP models. SQuAD datasets contain a series of text passages accompanied by a question and the task is to predict the span of the answer in the passage. Using 16-bit integer data type, BERT is able to either match or outperform the FP32 performance for all tasks. The 16-bit integer BERT also shows similar or better performance compared to the FP16 mixed precision fine-tuning method. Further reducing the integer bit-width to 8, fine-tuning BERT exhibits an average of 1.7 point drop on GLUE benchmark and 4.5 point drop for SQuAD datasets. Moreover, our experiments show that using 10-bit and 12-bit integers has average score drops of 0.8 and 0.3 points for GLUE tasks, and 0.8 and 0.2 point for SQuAD datasets respectively.

5.3 Loss Trajectory

Figure 5 shows the loss trajectory of integer fine-tuning BERT on SQuAD v2.0 dataset using 16-bit and 8-bit integers, along with FP32 method. The fine-tuning loss trajectory of BERT using 16-bit integer closely follows the FP32 loss trajectory. On the other hand, when fine-tuning with 8-bit integer parameters and 12-bit integer input activations, the loss trajectory is slightly shifted, but follows the same trend of its FP32 counterpart.

6 Conclusion

We proposed an integer fine-tuning method for transformer based language models using dynamic fixed-point format. We used dynamic fixed-point data type to represent parameters, input activations and gradients in integer values. As a result, our fine-tuning method uses integer arithmetic for the forward and back propagation of compute intensive layers such as linear, layer-norm and embedding layers of BERT model. Furthermore, we studied that increasing the bit-width of the dynamic fixed-point format reduces the variance of the mapping function and thus, improves the convergence of our integer fine-tuning method. We conduct fine-tuning experiments on GLUE benchmark and SQuAD datasets to compare the metric performance of our integer BERT with FP16 mixed precision and FP32 fine-tuning methods. Our experiments show that the 16-bit integer fine-tuning is able to achieve the same metric performance as the FP16 mixed precision fine-tuning method. In addition, fine-tuning BERT with lower bit-width data types, i.e. 8-bit integer, maintains an average drop of metric score within 3.1 points of the FP16 setup.

Limitations

Although our integer fine-tuning method uses integer numbers for compute intensive layers of BERT, integer support for non-linear layers of BERT, e.g. softmax and GELU activation, are left for future work.

We have shown in Figure 1 that the integer data types are faster for the general case. However, a direct comparison of the time and memory cost of our integer fine-tuning method with the FP16 and FP32 methods is left for future works due to lack of access to a proper hardware with integer tensor core support.

Despite the similarities between fine-tuning and pre-training phases, they differ in key aspects of training such as dataset size and number of epochs. The challenges of using integer arithmetic in the pre-training phase will be studied in the future work.

References

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Haoli Bai, Wei Zhang, Lu Hou, Lifeng Shang, Jing Jin, Xin Jiang, Qun Liu, Michael Lyu, and Irwin King. 2020. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*.

Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. 2019. Efficient 8-bit quantization of transformer neural machine language translation model. *arXiv preprint arXiv:1906.00532*.

Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. 2021. Understanding and overcoming the challenges of efficient transformer quantization. *arXiv preprint arXiv:2109.12948*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Alireza Ghaffari, Marzieh S Tahaei, Mohammadreza Tayaranian, Masoud Asgharian, and Vahid Partovi Nia. 2022. Is integer arithmetic enough for deep learning training? *arXiv preprint arXiv:2207.08822*.

Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pages 5506–5518. PMLR.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740*.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Darrell Williamson. 1991. Dynamically scaled fixed point arithmetic. In *[1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*, pages 315–318. IEEE.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. 2020. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 811–824. IEEE.

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE.

Wei Zhang, Lu Hou, Yichun Yin, Lifeng Shang, Xiao Chen, Xin Jiang, and Qun Liu. 2020a. Ternarybert: Distillation-aware ultra-low bit bert. *arXiv preprint arXiv:2009.12812*.

Xishan Zhang, Shaoli Liu, Rui Zhang, Chang Liu, Di Huang, Shiyi Zhou, Jiaming Guo, Qi Guo, Zidong Du, Tian Zhi, et al. 2020b. Fixed-point back-propagation training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2330–2338.

Kang Zhao, Sida Huang, Pan Pan, Yinghan Li, Yingya Zhang, Zhenyu Gu, and Yinghui Xu. 2021. Distribution adaptive int8 quantization for training cnns. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3483–3491.

Feng Zhu, Ruihao Gong, Fengwei Yu, Xianglong Liu, Yanfei Wang, Zhelong Li, Xiuqi Yang, and Junjie Yan. 2020. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1969–1979.

A Data Types

In this section we provide a brief overview of various data types mentioned in this work.

Floating-point data type is used to represent decimal fractional numbers. A binary floating-point number has three components of sign (s), mantissa (m), and exponent (e). Using these components, floating-point number x is shown as:

$$x = (-1)^s \times m \times 2^{e-t}$$

where t is the precision and $0 \leq m \leq 2^t - 1$. Another way of representing floating-point numbers is as

$$x = (-1)^s \times 2^e (\frac{d_1}{2} + \frac{d_1}{4} + \ldots + \frac{d_t}{2^t})$$

where d_i are binary digits of m. For FP32, exponent and mantissa are 8 and 23 bit integer numbers.

Fixed-point is another data type for representing fractional numbers. Unlike floating-point numbers where each mantissa is scaled using its respective exponent, fixed-point uses a single scale factor for all the numbers.

We use the dynamic fixed-point data type in our integer fine-tuning method. Also known as block floating-point, this format uses a different scale for each block of numbers to allow for more flexibility.