EFloat: Entropy-coded Floating Point Format for
Deep Learning

Rajesh Bordawekar
bordaw@us.ibm.com
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA

Bulent Abali
abali@us.ibm.com
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA

Ming-Hung Chen
minghungchen@ibm.com
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA

ABSTRACT
We describe the EFloat floating-point number format with 4 to 6 additional bits of precision and a wider exponent range than the existing floating point (FP) formats of any width including FP32, BFloat16, IEEE-Half precision, DLFloat, TensorFloat, and 8-bit floats. In a large class of deep learning models we observe that FP exponent values tend to cluster around few unique values which presents entropy encoding opportunities. The EFloat format encodes frequent exponent values and signs with Huffman codes to minimize the average exponent field width. Saved bits then become available to the mantissa increasing the EFloat numeric precision on average by 4 to 6 bits compared to other FP formats of equal width. The proposed encoding concept may be beneficial to low-precision formats including 8-bit floats. Training deep learning models with low precision arithmetic is challenging. EFloat, with its increased precision may provide an opportunity for those tasks as well. We currently use the EFloat format for compressing and saving memory used in large NLP deep learning models. A potential hardware implementation for improving PCIe and memory bandwidth limitations of AI accelerators is also discussed.

1 INTRODUCTION
In a short span of years, Artificial Intelligence (AI) has revolutionized the way we interact with computers. From recognizing voice commands, identifying similar photographs, home robot cleaning, recommending movies to watch, and understanding natural language text documents, AI has impacted multiple aspects of our life. With continuous advances in software and hardware capabilities, capabilities and impact of AI will continue to grow. Unfortunately, as the AI models expand their capabilities, their complexity, training costs, and trained model sizes increase dramatically. Although this observation applies to a broader class of AI models, it has particular significance to AI models in natural language processing (NLP) [6, 46]. For example, the state-of-the-art transformer-based NLP models such as BERT, T5, Megatron-LM, Open AI GPT-2/3, or Google Switch-C Transformers, contain from hundreds of million or billion [25], to even trillion parameters [16].

As we elaborate in Section 7, NLP model compression is a very active area of research. However, most of these techniques are designed to address the specific goal of improving model inference performance. Furthermore, many of these techniques do not apply to other NLP-related deep learning approaches, such as database embedding (db2Vec) [9, 10]. db2Vec adapts a widely used NLP technique called vector embedding to build semantic models from multi-modal relational database tables. db2Vec differs from its NLP counterparts, such as Word2Vec [36] or GloVe [39], in many ways. For db2Vec, source data follows the relational data model [14] (i.e., the source data is not a document written in a natural language). As a consequence, db2Vec needs to address various aspects of the relational data model such as the notion of a relational table and row, unordered entities within a row, (potentially unique) primary keys, entities of different types including dates and numeric values, NULL values, etc. In addition, the relational database tables can be very large (e.g., billions of rows in a table), with a large number of unique string tokens, leading to a much larger vocabulary than a natural language document. As a result, the trained db2Vec models can be very large.

The motivation for this work is to explore strategies to compress a large db2Vec model such that, if required, one can recover the original model or use the compressed model as-is in the inference phase. Like any trained vector embedding model, a db2Vec model is a snapshot of its weight matrices and consists of weight values represented using IEEE 32-bit single-precision floating point representation (FP32). Therefore, we focus on compression approaches that exploit different reduced-bit (low-precision) representations of floating point values.

Low-precision arithmetic has been successfully used in deep learning (DL) and machine learning (ML) applications, in particular for developing specialized compute accelerators for inference tasks [40]. Since the ML/DL inference computations are heavily dominated with multiplication operations and as the area-delay product of a \( k \)-bit high speed multiplier is at least \( O(k^2 \log k) \) [38], the traditional FP32 format becomes expensive for inference tasks. High compute demands along with small area, power, and bandwidth requirements...
led to design and implementation of low-precision number formats based on 8- to 16-bit wide integers and floats.

Existing low-precision floating-point (FP) formats make a tradeoff between the number of exponent and mantissa bits. An FP number is of the form

\[-1^{\text{signbit}} \times 2^{\text{exponent-bias}} \times \text{mantissa}\]

The exponent largely determines the range of minimum and maximum values representable by the FP format and mantissa determines the precision. BFloat16 (BF16) with an 8-bit exponent and 7-bit mantissa has a wide range but low precision when compared to FP32 and FP16 as illustrated in Figs.1(a,b,c). On the other hand, IEEE half-precision FP16 with a 5-bit exponent and 10-bit mantissa has a greater precision but a tighter range than BF16. Other FP formats, for example, DLFLOAT16 (Fig.1(d)), try to strike balance between the two.

In this work, we introduce a new low-precision FP format, EFloat (EF), that uses entropy-coded variable width representation of the exponent field. Our design is motivated by a key pattern that we observed across a wide range of vector embedding models: post-training, these models use only few of the $2^8 = 256$ unique exponents available in FP32. For example, Figure 2 shows the exponent distribution of FP32 values in multiple trained vector embedding and related NLP models, including word2Vec, sent2Vec, doc2Vec, GloVe, db2Vec, fast-text, universal sentence encoder, and graph embedding (BigGraph) ([8, 9, 12, 13, 31, 32, 36, 37, 39]). Bulk of the FP32 exponent population is in the range $-9$ to $0$ ($2^{-9}$ to $2^0$), and the most frequent exponents are $-3$ and $-2$. This peculiar uneven distribution of exponent population is the consequence of the log-bilinear language model that underpins the vector embedding training approach [5, 23]. The EF design exploits this behavior and assigns the least number of exponent bits to most common exponent values, while preserving the range of the original floating point representation.

Our proposed EFloat format provides the following benefits:

- EFloat provides reduced-bit representation of any floating point format (e.g., FP32, FP16, etc.), by using fewer exponent bits to map the same range as the original value.
- For a given bit budget (e.g., 16), EFloat provides more accurate representation of the values by using fewer exponent bits to capture the same range as before, and then using the remaining bits to increase mantissa precision.
- The EFloat format is suitable for both storage compression and reduced-bit computations over vector embedding models.
- As the EFloat-based compression works on pre-trained vector embedding models, no changes to the model training are needed.
- Given the numerical precision of 16-bit EFloat (EF16) representation is numerically much more closer to FP32 than BF16 (Section 5), one can only use EF16 representations for computations rather than mixed (32- and 16-bit) computations currently in use (e.g., the most recent and the largest NLP model from Google Brain, Switch-C Transformers [16], uses FP32 for Softmax computations and BF16 for all other computations.).
- Since vector embedding models are used in a wide array of NLP transformer architectures, the EFloat representation can be used for a much wider (and more space consuming) class of NLP models.
- A hardware implementation of EFloat supporting both inferencing and training is sketched. Table based O(1) delay EF conversion in hardware is possible. A single pre-built table may be used for all training iterations when the model distribution is approximately known in advance.

We describe the EFloat format in Section 2. Section 3 provides insights on why EFloat is effective with the vector embedding models. Section 4 describes conversions between EFloat and other FP formats. Section 5 presents an error analysis of EFloat. Section 6 includes discussions on potential hardware implementations for EFloat conversion and arithmetic. Finally, in Section 7, a review of related work on model compression, floating-point representations for deep learning, and general purpose compression methods is presented.

2 THE EFLOAT FORMAT

For illustration purposes we use 16-bit EFloat (EF16) representation throughout this paper, without loss of generality, as the means to illustrate the compaction gains and reduced loss of precision as compared to 32-bit IEEE float (FP32). However, other widths may be possible and we have used EF widths of 8, 9, 10, 11, and 16-bits in practice. Section 5 compares accuracy between some of these implementations.

The key idea of EFloat is variable width encoding of the exponent field. We used the well-known Huffman encoding method. EFloat exponent’s logical width is 8-bit, the same as that of FP32 and BF16. However, after encoding the exponent field width shrinks; for some FP values it could be as small as 1-bit. The EF exponent’s logical width is 8-bits even for EF8 and shorter since the EF coded-exponent typically needs much less than 8 bits.
Depending on the frequency of unique exponent values in the dataset and thanks to the entropy coding, the coded-exponent widths may vary between 1-bit to 8-bit within the same dataset, as shown in Figure 1(f,g). In practice we observed that the minimum coded-exponent width is 2-bit for the most frequent exponents. When averaged across the entire model the exponent width is in the 2.7-bit to 3-bit range (Fig.1(h)).

Compressing the exponents as described increases precision of EFloat. Once the exponent is a field of variable width, the mantissa also becomes a field of variable width. A $N$-bit coded-exponent in an EF16 float format results in a $(15-N)$-bit mantissa as shown in Fig.1(e). Bits saved from the exponent become part of the mantissa, therefore increasing the precision. EFloat on average have greater precision and range than any other FP format with the same bit budget. For example, EF16 with a 3-bit coded-exponent has 12-bits of mantissa compared to the 7-bit mantissa found in a BF16 (Figs.1(h,b)), although the logical exponent width of EF16 is same as for BF16, 8-bits.

The Huffman algorithm builds probabilities based on frequency histogram of exponents and outputs a code-table mapping 8-bit exponents to variable-width coded-exponents. The code-table may be constructed once and used many times as described in Section 3. Figure 3 shows an example vector embedding model with the most frequent exponents $-2$ and $-3$ which are coded with 2-bit binary codes 00 and 01. Less frequent exponents are coded with 3-bit codes and longer. Those are in the form 10... and 11... Least frequent exponents are coded with 8-bits. Average exponent width of few other datasets are shown in Figure 4 which shows that savings are realized across all datasets (The average width is the population weighted average of coded-exponent bits.).

EFloat with infrequent exponents have a wider exponent field and therefore less precision. For example, exponents $-22$ to $-12$ are coded with 8-bits in Figure 3. However, infrequent EFloat exponents have a relatively small contribution to

---

**Figure 1:** Floating point formats are compared. EFloat has a fixed total width, but the boundary between the exponent and the mantissa is variable (e). EFloat has greater precision and range than the existing FP formats having the same bit budget (e.g., (a) to (d)). Exponent is entropy coded providing an average of 4 to 6 extra bits of precision to mantissa (e.g., (h)), while retaining a logical exponent range of 8 bits, same as FP32.
Figure 2: Histogram of the exponent fields of 32-bit floating-point (FP32) values found in vector-embedding and related NLP models. Only the db2Vec models were generated, the rest were downloaded as publicly available pre-trained models.

Figure 3: Histogram of and Huffman coded exponent widths of the Wiki-word2vec dataset. The top two popular exponents -2 and -3 are coded with 2-bit codes, whereas less popular exponents are coded with as many as 8-bits.

common calculations used in ML/DL applications, such as dot-product, vector-sums, and cosine-similarity. Therefore, their impact on the overall accuracy of the end result is small (EF precision is quantified and compared to prior formats in Section 5.).

The EF code-table for format conversion may be obtained online or offline, as detailed in Section 4. The maximum code width may be set to other than 8-bit depending on the hardware and software requirements. Note also that a single-peak histogram is not required to realize the bit savings as observed for the Wiki-doc2vec dataset in Fig.5. Uneven distribution is the main requirement to implement bit savings with entropy methods.

Note also that Huffman codes are prefix codes which encode both the original value and the code-width. Therefore, the movable boundary between the exponent and the mantissa is not ambiguous; a boundary marker is not necessary. A decoder-table indexed by the coded-exponent may be used to decode the original exponent value and the mantissa’s leading bit position in \( O(1) \) time, as described in Section 4.

Currently we use EF for memory and bandwidth compression. We encode and decode EFloaf with software trading
compute cycles with capacity savings. In our AI database applications, trained models consume large amounts of memory. A factor of 3 reduction in memory footprint is achieved by converting FP32 values to EF11. Bandwidth often gates performance when streaming or copying a model from memory to an on-chip or off-chip (PCIe) accelerator. EF saves bandwidth in those scenarios as well.

In summary, the EF format provides 4 to 6 more mantissa bits than prior formats using the same bit budget. Alternatively, for the same precision the EF format saves memory capacity and bandwidth compared to prior formats.

3 ANALYZING DL MODELS AND THEIR IMPACT ON FP EXPONENT BEHAVIOR

Vector embedding models are extensively used in natural language processing (NLP) to capture and exploit semantics of the word entities (e.g., words, sentences, phrases, paragraphs, or documents). A trained vector embedding model consists of a set of vectors, each vector representing a feature in the text corpora. Each vector is composed of a set of single-precision 32-bit IEEE 754 floating point (real-valued) FP32 numbers. More specifically, each feature vector encodes a distributed representation of inferred semantics of a word entity; i.e., a single vector captures different attributes of the inferred semantics [24], created, in part, by contributions by other word entities. Every vector embedding model implements some variant of the log-bilinear language (LBL) model that predicts the probability of the next word \( w_i \) given the previous words \( \text{context} \) \([5, 7, 23]\). The LBL model first predicts a real-valued vector representation of a word by linearly combining the real-valued vector representations of its context words. Then the distributed representation of the word is computed based on the similarity between the predicted representation and
the representations of all words in the vocabulary. This step is accomplished using the normalized exponential or Softmax function over the associated feature vectors. The output of the Softmax function is the probability distribution over $V$ different possible outcomes, where $V$ is the vocabulary size.

Figure 2 presents histograms of exponent values in multiple pre-trained vector embedding models, where the X-axis represents exponent values (from the 8-bit exponent portion of a 32-bit IEEE 754 floating point value) and for each value, the Y-axis represents normalized numbers of occurrences of that exponent value. Vector embedding and related NLP models presented in Figure 2 include word embedding (word2Vec) [36], sentence (sent2Vec) and document embedding (doc2Vec) [31], GloVe [39], sub-word embedding (fast text) [8], database embedding (db2Vec) [9], graph embedding (PyTorch BigGraph [32]), and Google’s transformer-based universal sentence encoder [12]. All these models implement different variations of the LBL model. The word2Vec based models, e.g., word2Vec, sent2Vec, doc2Vec, db2Vec, and fast text, use a neural network with different widths of Softmax as the activation function. GloVe, on the other hand, is a count-based optimization approach that uses a word co-occurrence matrix and weighted least-square as the optimization function. The fast text sub-word model [8, 27], assigns a vector for every character $n$-gram, using an extended skip-gram model [36] and then, words are represented as the sum of these representations. The universal sentence encoder generates embedding vectors for sentences using a standard Transformer architecture that takes word embedding vectors as input and uses Softmax function to compute attention [50]. Irrespective of the type of model, we observe that exponent values cluster around a certain range of values, and display a distinct peak. The only exception is the doc2Vec model that exhibits two peaks as the doc2Vec first builds fine-grained embeddings for words and then uses them to build embeddings for coarser-grained entities such as paragraphs via averaging individual word vectors. The averaging process results in reducing the histogram peak height and creating a smaller second peak.

To understand the exponent behavior presented in Figure 2, let us delve deeper into the training of an embedding model. For illustration purposes, we use database embedding (db2Vec) as an example. db2Vec is an adaptation of the word2vec approach, and has been designed to build an embedding model from structured database tables that adhere to the relational data model. Like word2vec, db2Vec also uses Skipgram with Negative Sampling (SGNS) as the training approach. The SGNS approach uses a binary classifier based on the logistic (Sigmoid) function instead of using the Softmax-based predictor. The overall training process involves multiple back-propagation iterations to update model weights using the gradient of the Sigmoid function, $\sigma(x)$:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The Sigmoid curve and it’s gradient $\sigma'(x) = \sigma(x)(1 - \sigma(x))$ is shown in Figure 6. The gradient of the Sigmoid activation function and the back-propagation process can be specified as

$$\text{weights}_{i+1} = \text{weights}_i + \text{learning} \_ \text{rate} \times \sigma'(x)$$

Weights get updated iteratively during the back-propagation process by the error computed for that iteration. Practically, the error is computed using the gradient of the activation function. During model training, we observe that the weights rapidly converge to their final values typically clustered around the slope of the Sigmoid curve as shown in Figure 6. Exponents of these weights are clustered substantially around the $2^{-8}$ to $2^{0}$ range as evidenced by the histogram in Figure 2. Smaller exponents do not appear in the model because the activation function output is practically zero for any input value when weights are smaller, and large exponents are eliminated because of normalization of weights.

Figures 7, 8, and 9 show the back-propagation process in action. Figure 7 presents exponent histogram of model weights from initialization (iteration 0) to the final iteration (iteration 2481); Figure 8 presents exponent histogram of update values before iteration $i$. As Figure 7 shows the exponents of the model weights rapidly move to their final values. In fact, in the first iteration, the exponent population is almost half-way towards its final values. In Fig.7 iteration 11, most exponents are settled to their final values. In Fig.9, between the iterations 101 and 301 only a few hundred out of tens of thousands of exponents change. Past the iteration 301 and through 2481 practically no exponent values change. This behavior is due to the exponents of the update values quickly converging to their terminal values.

Note that early in the training iterations the mantissa precision is not as important. Because model weight updates are dominated by exponent updates from one iteration to the next. Once exponents settled to their final values, the mantissa precision becomes more important since weights start converging to their final values in small increments.

This observation is useful for amortizing the overhead of Huffman code-table generation over many training iterations and inferencing. One could produce a static code-table covering multiple NLP models (except a few models such as Wiki-doc2vec) shown in Figure 2. Then, one could use the same code-table any number times because the exponent frequencies hence the code-tables are practically identical across these models. A pre-built code-table may be optimized for the final iteration of training but the used for all iterations.
Derivative of Sigmoid

Figure 6: The Sigmoid \( \sigma(x) \) curve and its gradient. The floating-point (FP32) exponent of few neural weights are overlaid on \( \sigma(x) \).

Figure 7: Population of floating-point (FP32) exponents of model weights during training of the Telecom_churn-db2vec model, for iterations 0 to through 2481

That the pre-built code table is not optimized for early training iterations is not important. Because early in the training, exponents are changing rapidly and mantissa precision is not as important.

4 EFLOAT CONVERSIONS

4.1 FP to EF (Encode)

In the following discussions we describe conversions from FP32 (8-bit exponent) to EF16, although other source FP formats and target EF widths, such as EF11, may be used as well.

Exponents in the original dataset are histogrammed first, e.g., Fig.2. For large datasets a statistically representative subset may also be histogrammed to save processing time.

If the sign bits have a skewed distribution, for example if the values are substantially positive, then the sign bit and the 8-bit exponent may be treated as a single 9-bit integer while histogramming. Thanks to entropy coding a skewed sign bit distribution may provide up to one additional bit of precision to the mantissa.

The histogram representing probabilities of the 8-bit integers is the input to the Huffman algorithm such as described in [1]. The algorithm uses exponents with non-zero probabilities as inputs. Then, it outputs a 256-entry \( (2^8) \) table of coded-exponents whose widths are inversely proportional to their probabilities such as shown in Fig.3. The table is indexed by the original 8-bit exponent. Each table entry contains the variable width coded-exponent and its width.
Figure 8: Population of floating-point (FP32) exponents of update values ($\text{learning\_rate} \times \sigma'(x)$) during Telecom_churn-db2vec model training for iterations 1 to through 2481

Figure 9: Differences in the floating-point (FP32) exponent populations between certain iterations during training of the Telecom_churn-db2vec model

Exponents that do not occur in the original dataset, e.g., with zero probability, have a null entry in the code-table.

Once the table is built, entire dataset is converted from FP32 to EF16 replacing original exponents with coded-exponents. Least significant bits of the FP32 mantissa are truncated to match the EF width. For example in Fig.3, the algorithm encodes the most frequent exponent with 2-bits. Accounting for the sign bit, a 2-bit coded-exponent yields a 13-bit mantissa in EF16 which is obtained by dropping off the bottom 10-bit from the 23-bit FP32 mantissa. Optionally, the mantissa may be rounded to the nearest value to provide additional precision, however carefully avoiding any overflow during rounding.

If the distribution is known in advance, a pre-built code-table may be used, as discussed at the end of Section 3. Pre-built code tables amortize the overhead of building Huffman code-tables over many models, their training iterations, and inferencing tasks.

Currently we convert EFloat with software trading compute cycles with capacity savings. Hardware conversion is feasible and will be discussed in Section 6.

4.2 Length Limited Encoding

The basic unlimited Huffman algorithm may produce extremely wide codes exceeding the EFloat’s entire width. If the dataset contains $N$ unique exponents, for a worst-case distribution the algorithm may produce codes with $N - 1$-bit widths exceeding EFloat’s entire width. For example, the...
Wiki-word2vec dataset in Fig.3 contains 23 unique exponent values. In the worst case, the maximum code width could have been $23 - 1 = 22$ bits which would not fit in an EF16 number. The length limited Huffman algorithm limits the maximum width of coded-exponents[1]. We limited the maximum coded-exponent width to 8 in Figs.3-4, although other limits are possible.

Figure 10 shows the effect of setting the maximum coded-exponent width to 5, 8, and 10-bits. The 10-bit limit increases widths of infrequent exponents from 8 to 10. Whereas for the frequent exponents the coded-exponent widths remain at 2-bit. The average coded-exponent widths are also nearly the same, 2.68 and 2.64, respectively. Therefore, increasing the limit from 8 to 10-bit doesn’t bring much benefit.

However, when the maximum coded-exponent widths are limited to 5-bits, the minimum widths increase from 2-bit to 3-bit (Fig.10). Because with a 5-bit limit, it is not possible to encode all 23 unique exponents and use 2-bit codes at the same time. As a result, the minimum and the average coded-exponent widths increase to 3 and 3.52 bits respectively which reduces the compression quality. That the minimum and average code widths are inversely proportional to the maximum code width may seem counterintuitive at first. Let the maximum code width be $K$-bits (e.g., $K = 5$ or $K = 8$ etc.). $K$-bits encode at most $2^K$ unique values. A shorter $L$-bit code where $L < K$, eliminates $2^{(K-L)}$ codes out of that $2^K$ total. For example, the 2-bit code 00 prevents using all 3, 4, 5, 6, 7, 8-bit codes that start with 00, when $K = 8$ (recall that we need to obey the prefix property to decode exponents unambiguously.)

Therefore, when $K$ is small, the Huffman algorithm cannot output the shortest possible codes (for the frequent exponents), because otherwise it is impossible to encode the remaining infrequent exponents. Therefore, to have a better compression ratio, the limit must be increased. In practice, we observed that increasing the maximum exponent width limit by 2 to 3 bits over the minimum possible gives a good compression ratio. For example, 23 unique exponents require a minimum of $\lceil \log_2(23) \rceil = 5$ bits. Therefore, the maximum code width may be set to 7 or 8.

An advantage of a small limit is having smaller decoder tables which will save chip area in a hardware implementation. For example, in a systolic-array (or a GPU) implementation, many SRAM based decoder tables will be needed to be used in parallel. The area difference between a $2^6$ entry SRAM and a $2^2$ entry SRAM may be significant enough to make an area–compression ratio tradeoff.

4.3 EF to FP (Decode)

For EFloat to FP conversion (i.e., decoding) we use a inverse mapping of the code-table described in Section 4.1. The variable width coded-exponent is used as an index of the EF to FP conversion table. With maximum 8-bit codes, a $2^8 = 256$ entry decoder-table is necessary. Each table entry contains the original exponent and the bit width of the coded-exponent. To be able index with shorter than maximum width codes, many entries are filled with duplicates. For example, the 2-bit coded-exponent 00 is duplicated 64 times in the table at locations 00000000 through 00111111 with each location containing the pair (exponent= −2, width= 2). Duplicating entries is equivalent to having logical don’t care bits in the index.

When decoding, the EFloat bit pattern is used as an index of the table. The original exponent and the code-width are read from the table entry. The code-width then determines the mantissa width in the remaining part of EFloat. Since the mantissa was truncated earlier during the FP to EF conversion, its missing least significant pids must be padded with zeros now to match the original FP width.

5 ERROR ANALYSIS

In this section, we compare the EF11, EF12, EF16 precision to the BF16 format using a sample of vector embedding models. We used the following experimental method to quantify precision. The datasets PubMed-word2vec, Wiki-doc2vec, and Glove (in Fig.2) originally stored in the FP32 format were converted to one of EF11, EF12, EF16, and BF16 formats. They all lose precision due to the least significant mantissa bits truncating during conversion. Then, they are converted back to FP32. The arithmetic difference $f^o − f^c$ of the original FP32 value $f^o$ and the processed FP32 value $f^c$ is the precision loss due to conversion. Root Mean Square Error (RMSE) metric is used to summarize the loss of precision across a dataset of $N$ floats,

$$RMSE = \sqrt{\frac{1}{N} \sum_{k} (f^o_k - f^c_k)^2}$$

We then compare the error of BF16 and EF dividing $RMSE_{BF16}$ by $RMSE_{EF}$ in Table 1. Ratios greater than 1.0 indicate when EF error is less than BF16 error. Note that in these experimental results, the datasets were encoded with minimum 3-bit and maximum 6-bit coded-exponents resulting in an average width in the range of 3.41 to 3.59-bits as the table shows. Accordingly, for EF16 the minimum mantissa width is 10-bit which is 3-bit wider than for BF16. Therefore, for all floats in the dataset EF16 has higher precision than BF16. Since EF16 encoded datasets average around 3.5 bits per exponent, mantissa width is $16 − 1 − 3.5 = 11.5$ on average which means
EF16 has 4.5 bits of greater precision than BF16 (which has 7-bit mantissa) while both having the same 8-bit exponent. When comparing the RMSE ratio of BF16 to EF16, we see a ratio of 16 to 25 (log_2 16 = 4 and log_2 25 = 4.64) which quantifies the expected increase in precision in RMSE units.

Table 1 shows that EF12 has same to slightly better RMSE than BF16 since the RMSE ratios are in the 1.0 to 1.6 range. Thus, EF12 uses 25% less bandwidth and memory capacity than BF16 at equal floating-point precision. One-bit shorter EF11 does worse than BF16. Had the maximum coded-exponent width been set to 6-bit instead of 8-bit, such as shown in Figs.3-4, another 0.8-bit of mantissa could have been available, making EF11 precision on par with BF16.

Note that the RMSE method amplifies larger errors due to the squaring of differences. EF coded floats with short mantissas (i.e., those with infrequent exponents) are disproportionately represented in the RMSE summation. When we use other error metrics such as Mean Absolute Error the EF error is generally less than when using RMSE. However, the true measure of error in our db2Vec models will be measured with the Normalized Discounted Cumulative Gain (NDCG) used for searching documents and ranking relevance. NDCG based EF results will be included in future work.

Also note that EFloat stores faithfully all FP32 special values including denormals and NaN. The exponent and the sign bit are compressed losslessly. Least significant bits of the mantissa are dropped off which is equivalent to rounding towards zero or optionally the remaining upper bits mantissa are rounded to the nearest. Special values of IEEE 754, signed zeros and infinities are compressed losslessly. NaN are semantically compressed losslessly: converting a NaN to and
from FP32 to EF and vice-versa still results in a NaN, however the non-zero mantissa bits of a NaN may be dropped off in the process making the float still a NaN.

6 HARDWARE CONSIDERATIONS

In this section, we discuss supporting the EFloat format in hardware. Our primary EF use case has been bandwidth and memory capacity savings. Compute throughput of on-chip and off-chip accelerators (i.e., PCIe attached) may be bandwidth gated. On-chip or off-chip memory (e.g., SRAM based cache and scratchpad memory, DRAM, HBM) capacity may also be limiting compute throughput. EF compressed AI models and input/output data will save bandwidth and memory capacity in those situations. Static Random Access Memory (SRAM) is a high speed memory typically used on processor chips. Dynamic RAM (DRAM) is a dense and higher capacity but slower than SRAM external memory. High Bandwidth Memory (HBM) is a type of high-performance DRAM used in GPUs and AI accelerators [21].

Hardware arithmetic functional units natively supporting EF currently do not exist. FP-EF conversion will be necessary before/after operating on the accelerator. EF support will be different depending on the whether the accelerator is used for inferencing or training. For inferencing, the trained AI models may be converted to the EF format once and placed in the system or accelerator memory in read-only form. FP to EF conversion may be done in software; no EF encoding hardware is necessary since this step is not time critical. However, EF decoding hardware will be necessary before feeding the EF-coded model weights to the functional units. EF decoders may be placed anywhere between the memory and the functional unit. If the accelerator is bandwidth limited then EF decoders may be placed before the accelerator inputs or next to the accelerator side of the communication links, i.e., memory bus, chip fabric, or PCIe links.

EF decoding hardware may be implemented with a SRAM lookup table. With a maximum code width of $K$ bits a $2^K$ entry SRAM based table may be used. $K = 8$ is desirable to cover the worst case condition of all 256 exponents utilized by an FP32 dataset. The table is indexed by the coded-exponent. Each entry consists of the original exponent and the coded-exponent’s bit-width. As stated in Section 4.3 many entries in the lookup table will be filled with duplicates for decoding shorter than maximum width coded-exponents.

When the accelerator has many input ports, for example a systolic array such as the Google TPU [28], or a wide SIMD architecture [21, 22], many decoder tables will be necessary for parallel access. To save area, the maximum code width may be set to $K < 8$ at the expense of giving up some compression quality. An alternative approach for saving area might be using a two level table which will substantially reduce SRAM capacity requirements. First level table is indexed by the most significant bits of the code and the second level table is indexed by the least significant bits of the code. Huffman prefix-trees have the property that the leading bits of infrequent codes are substantially identical which results in many shared entries in the first level table. For example, the Zlib software [3] employing Huffman codes uses a maximum code width of 15-bits. But for encoding of 512 Zlib symbols, a 2-level table needs only 852 table entries compared to $2^{15} = 32768$ entries required in a 1-level table.

For training, the accelerator will perform many weight updates, writing floats back to the memory during training iterations. Therefore EF encoders must be placed at the accelerator outputs. EF encoding hardware may be implemented with an SRAM lookup table indexed with the original exponent. Each table entry consists of a coded-exponent and its bit width.

Functional units natively processing EF floats do not exist yet. Therefore, an EF float must be first converted to a plain unencoded float. FP multipliers and adders must be sized to accommodate the maximum possible mantissa width. For example, BF16 has an 8-bit exponent and 7-bit mantissa. EF16 with minimum 2-bit coded-exponent also has an 8-bit exponent but a 13-bit mantissa which will result in a larger EF16 multiplier than for BF16. To have similar area, one might consider using a smaller EF10 format instead of EF16.

7 RELATED WORK

In this section, we overview relevant work in lossless compression techniques, deep learning model compression, and floating point representations for deep learning.

Entropy encoding is a statistical method for lossless compression [43]. Fixed-size items are replaced with variable-size codes with the shorter codes assigned to the frequently occurring items in the data. Huffman coding, Arithmetic coding and Range coding are commonly used entropy methods. Variable-size codes, for example the Huffman codes, generally have the prefix-property permitting their concatenation without any separating markers in between.

Dictionary methods for lossless compression, most popularly the Lempel-Ziv (LZ) algorithms, use a dictionary of strings [43]. Strings in the data stream, when found in the dictionary, are replaced with distance and length pairs pointing to their dictionary location therefore achieving compression. A dynamic dictionary is typically the most recent set of input strings, e.g., the recent 32KB of input in the popular Deflate method [3].

Over the years, the size and complexity of deep learning models has increased substantially. In particular, advent of new transformer-based NLP models (e.g., BERT and friends, T5, Megatron-LM, Open AI GPT-2/3) has highlighted the very
high space and computational costs associated with these models [6, 18–20, 33, 45]. Given potential uses of NLP models in enterprise and consumer domains, a lot of attention is being devoted to compressing such models. The primary goal of these compression efforts is to reduce the size of a pre-trained model to enable its deployment in real world industrial applications that demand low memory footprint, low response times, and smaller computational and power budget during the inference phase. Gupta and Agarwal [20] have identified six different types of compression techniques currently being used for the NLP models: pruning, quantization, parameter sharing, knowledge distillation, tensor decomposition, and Linear Transformer based methods. The pruning approach is the most obvious way to reduce model size by sparsifying weight matrices. Pruning is related to quantization which aims to reduce the number of bits to represent each weight. Quantization covers two broad approaches: the first represents a full-precision (e.g., 32-bit) floating point weight value using reduced or mixed precision representations, and the second converts full-precision floating point values into integer values with fewer bits (e.g., INT8, INT4, and INT1 [26, 35, 49, 53]). Another way to reduce model size is parameter sharing that uses fewer shared values to represent similar weights. Knowledge distillation aims to build a student-trainer model where the student is trained to mimic a pre-trained larger teacher model. The deeper teacher model is trained first, and then the student model is trained via knowledge transfer. Tensor factorization covers a set of techniques that can be used to approximate a larger matrix using a combination of smaller matrices computed via tensor decomposition methods. The final technique aims to develop transformer-based models that are linear in terms of input sequence size, rather than the current quadratic complexity. In general, these techniques follow a destructive approach that throws out the original large model and can not recreate it from the compressed smaller model.

Compression techniques have been explored to reduce data communication costs during deep learning training. Floating-point quantization approaches are often used to reduce the communication volume during the deep learning training process [54]. Gajjala et al [17] use Huffman encoding based techniques to encode quantized gradients for optimizing communication volume in distributed deep learning training. Recently announced Nvidia nvcomp [42] uses LZ4 and run-length encoding (RLE) based approaches to compress data being communicated between GPUs during training of deep learning models. While the approaches used in nvcomp work well for string and integer datatypes, these techniques do not support floating point values very well. Also, LZ4 and RLE approaches work well only for repeated values or sequences. The EFloat approach is compatible and orthogonal to these existing compression techniques employed in NLP models.

In conjunction with the model compression work, there has been significant activity in devising reduced-precision floating point formats tuned for broader machine learning and HPC applications [2, 44]. Unlike the inference-focused model compression work, reduced-precision floating point representations are designed to work for both model training and inference phases. The most common reduced-precision floating point representation uses 16 bits. Current 16-bit implementations include IEEE 754 half-precision (FP16), with 1 sign bit, 5 exponent bits, and 10 fraction bits; Brain Floating Point (BFLOAT16) [29, 52], with 1 sign bit, 8 exponent bits, and 7 fraction bits; and Deep Learning Float (DLFloat) [4], with 1 sign bit, 6 exponent bits, and 7 fraction bits. TensorFlow-32 (TF32) from Nvidia is a 19-bit format that combines 8 exponent bits from BFLOAT16 and 10 exponent bits from IEEE FP16. Hybrid Block Floating Point (HBFP) [15], Intel Nervana’s Flexpoint [30], and Microsoft MSFP [41] formats combine the advantages of fixed point and floating point representations by splitting up the mantissa and the exponent part which is shared across multiple numeric values. Recent research proposals have described training of key deep learning models using even reduced precision floating point values (8- and 4-bit representations) [11, 34, 47, 51]. Recently proposed AdaptiveFloat [48] is an inference-targeted floating-point format which maximizes its dynamic range at a network layer granularity by dynamically shifting its exponent range via modifications to the exponent bias and by optimally clipping (quantizing) its representable datapoints. Our proposed EFloat design practically achieves the same result without altering the exponent range and quantizing full-precision values.

8 CONCLUSION

We introduced EFloat, a novel entropy-coded variable length floating point format for deep learning applications. This format can be used for compressing a trained deep learning model, as well as for enabling more accurate representations of low-precision floating point representations (e.g., FP16). While our intended use cases were initially for the database embedding (db2Vec) workloads, we demonstrate that the proposed format works effectively for other vector embedding models, and can be used for a much broader class of NLP models including transformer-based models. Broadly, EFloat may be used in a deep learning applications where tradeoffs need to be made between range, precision, memory capacity and bandwidth savings. Future work being considered include hardware implementation of EFloat conversions as well as hardware capable of natively processing EFloat.
We thank Jose Neves for his insightful comments on the earlier versions of this paper.

9 ACKNOWLEDGMENTS

We thank Jose Neves for his insightful comments on the earlier versions of this paper.

REFERENCES

[1] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B Agricola, Bedir Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J Starke, et al. 2020. Data Compression Accelerator on IBM POWER® and x15 Processors. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE Computer Society, 1–14.

[2] Ahmad Abdeelfattah, Hartwig Anzt, Erik G. Boman, Erin Carson, Terry Cojean, Jack Dongarra, Mark Gates, Thomas Grützmacher, Nicholas J. Higham, Sherry Li, Neil Lindquist, Yang Liu, Jennifer Loe, Piotr Luszczek, Pratik Nayak, Sri Pranes, Siva Rajamaniukam, Tobias Rühle, Barry Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M. Tsai, Ichitaro Yamazaki, and Urike Meier Yang. 2020. A Survey of Numerical Methods Utilizing Mixed Precision Arithmetic. arXiv:cs:MS/2007.06674

[3] Mark Adler. 2020. Zlib: A massively spiffy yet delicately unobtrusive compression library. https://github.com/madler/zlib. Accessed: 2020-01-13.

[4] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan. 2019. DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference. In 2019 IEEE 66th Symposium on Computer Arithmetic (ARITH). 92–95.

[5] Felipe Almeida and Geraldo Xexeo. 2019. Word Embeddings: A Survey. arXiv:cs.CL/1901.09609

[6] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Sharmarke Shumitzhel. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In Proceedings of the Conference on Fairness, Accountability, and Transparency (FAccT21). Association for Computing Machinery, New York, NY, USA. Online. https://doi.org/10.1145/3442188.3445922

[7] Emily M. Bender and Alexander Koller. 2020. Climbing towards NLU: On Meaning, Form, and Understanding in the Age of Data. In Proceedings of the 8th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Online, 5185–5198. https://doi.org/10.18653/v1/2020.acl-main.465

[8] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. arXiv:cs.CL/1607.04606

[9] Rajesh Bordawekar and Oded Shmueli. 2016. Enabling Cognitive Intelligence Queries in Relational Databases using Low-dimensional Word Embeddings. CoRR abs/1603.07185 (2016). http://arxiv.org/abs/1603.07185

[10] Rajesh Bordawekar and Oded Shmueli. 2017. Using Word Embedding to Enable Semantic Queries in Relational Databases. In Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning (DEEM’17). ACM, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/3076246.3076251

[11] Leopold Cambier, Anahita Bhivandiwalla, Ting Gong, Mehran Nekuui, Oguz H Elibil, and Hanlin Tang. 2020. Shifted and Squeezed 8-bit Floating Point format for Low-Precision Training of Deep Neural Networks. arXiv:cs:LG/2001.05674

[12] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Brian Strope, and Ray Kurzweil. 2018. Universal Sentence Encoder for English. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. Association for Computational Linguistics, Brussels, Belgium, 169–174. https://doi.org/10.18653/v1/D18-2029

[13] Andrew M. Dai, Christopher Olah, and Quoc V. Le. 2015. Document Embedding with Paragraph Vectors. CoRR abs/1507.07998 (2015). http://arxiv.org/abs/1507.07998

[14] C. J. Date. 1982. A Formal Definition of the relational Model. SIGMOD Rec. 13, 1 (1982), 18–29. https://doi.org/10.1145/984514.984515

[15] Mario Drumond, Tao Lin, Martin Jaggi, and Babak Falsafi. 2018. Training DNNs with Hybrid Block Floating Point. In Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS’18). Curran Associates Inc., Red Hook, NY, USA, 451–461.

[16] William Fedus, Barret Zoph, and Noah Shazeer. 2021. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. arXiv:cs:LG/2101.03961

[17] Rishikesh R. Gajjala, Shashwat Banchhor, Ahmed M. Abdelmoniem, Aritta Dutta, Marco Canini, and Panos Kalnis. 2020. Huffman Coding Based Encoding Techniques for Fast Distributed Deep Learning. In Proceedings of the 1st Workshop on Distributed Machine Learning. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3426745.3431334

[18] Prakhar Ganesh, Yao Chen, Xin Lou, Mohammad Ali Khan, Yin Yang, Deming Chen, Marianne Winslett, Hassan Sajjad, and Freslav Nakov. 2020. Compressing Large-Scale Transformer-Based Models: A Case Study on BERT. arXiv:cs:LG/2002.11985

[19] Michael A. Gordon. 2019. All The Ways You Can Compress BERT. http://mitchgordon.me/machine/learning/2019/11/18/all-the-ways-to-compress-BERT.html.

[20] John L Hennessy and David A Patterson. 2017. Computer architecture: a quantitative approach, 6th edition. The Morgan Kaufmann Series in Computer Architecture and Design.

[21] Glenn Henry, Parviz Palangpour, Michael Thomson, J Scott Gardner, Bryce Arden, Jim Donahue, Kimble Houck, Jonathan Johnson, Kyle OBrien, Scott Petersen, et al. 2020. High-Performance Deep-Learning Coprocessor Integrated into x86 SoC with Server-Class CPUs. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). IEEE, 15–26.

[22] Geoffrey Hinton. 2013. Learning Distributed Representations For Statistical Language Modelling. https://www.cs.toronto.edu/~hinton/csc2535/notes/hlbl.pdf.

[23] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. 1986. Distributed representations. Parallel distributed processing: explorations in the microstructure of cognition 1 (1986), 77–109.

[24] Torsten Hoeffer. 2020. OpenAI’s GPT-3 took more than 314 Zettaflop (mixed fp16/fp32) to train! https://twitter.com/thoefler/status/1268291652701775047?s=20.

[25] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. arXiv:cs:LG/1712.05877

[26] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2016. Bag of Tricks for Efficient Text Classification. arXiv:cs:CL/1607.01759

[27] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. 2018. Motivation for and evaluation of the first tensor processing unit. IEEE Micro 38, 3 (2018), 10–19.

[28] Dhiraj Kalamar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo
