Reducing Space Complexity with Precision-Optimized Floats

Nicholas Sullivan, Jack McCall, Vincent Siu, Thien Hoang, Wonsang Lee
{d.sullivan, j.r.mccall, vincent.siu, h.thien, l.wonsang}@wustl.edu

Abstract

Floating point numbers were codified into a technical standard in 1985 specified by IEEE 754. Part of this specification was the bias term of the float. Bias equation LLMs and other machine learning models' weights are centered around zero. While of half of all possible floats are between 0 and 1, half are larger than 1 and outside the range of most model weights. By changing the bias we find a slight improvement in MNIST improvement when using our 32-bit floats compared to IEEE 32-bit floats.

1 Introduction

Floating-point numbers or (floats) are used by computers to as the representation of non-whole numbers. Floats have advantages over other representations of non-whole numbers like fixed point numbers due to their wider range. They are able to represent larger numbers than fixed point numbers. As the name suggests fixed point numbers are a way of storing non-integer whole numbers by simply placing a decimal point at a predetermined point in the binary string.

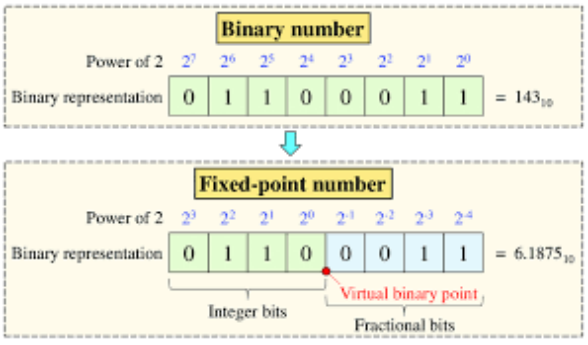

Figure 1: Fixed-point representation.

For numbers with large magnitude the small digit does not matter very much. For example programmers likely don't need both 1,000,000 and 1,000,000.000001. The 0.000001 is so small compared to 1,000,000 that it is unlikely to need such high precision. The solution to this changing where the decimal goes, meaning we make it float. In order to do this change the way we think about numbers. Instead of representing about the number like we do in binary and in fixed-point we represent it as a number in scientific notation.

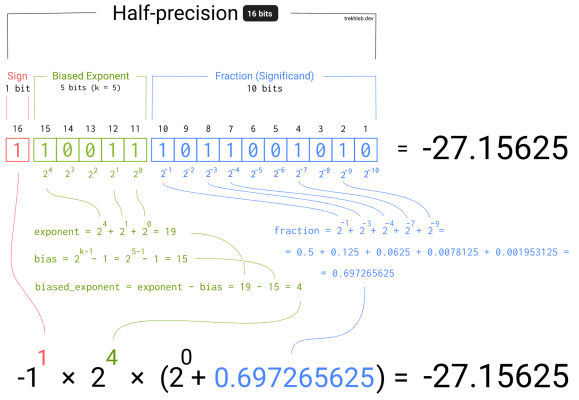

Figure 2: Fixed-point representation.

This is achieved by expressing a number in the form $x = \pm m \times 2^e$, where m is the significant (or mantissa), e is the exponent. This is along with a sign bit to represent the sign of the float.

In floating-point representation, exponents play a crucial role in determining the scale of the number being represented. A floating-point number consists of three main components: the sign, the mantissa (or significand), and the exponent. The exponent determines the magnitude of the number by shifting the decimal point in the mantissa. The exponent is typically stored in a biased

form, where a fixed value (the bias) is added to the actual exponent. This allows for a more efficient representation of both very small and very large numbers. The exponent allows floating-point numbers to have a wide dynamic range, enabling them to represent numbers that span many orders of magnitude, from the smallest positive values to the largest possible values. In the IEEE 754 standard floats' bias are the max value of the exponent divided by 2 minus 1. For standard 32 bit floats with 8 bit exponent the bias is 127.

Floating-point systems follow the IEEE 754 standard, which defines key aspects such as the representation of numbers, rounding behavior, and handling of special cases like infinity and Not-a-Number (NaN).

Floating point-numbers are composed of finitely many bits. This means that unlike what is commonly repeated, floating point numbers only represent a subset of the rationals. Irrationals are impossible to represent on a finite amount of bits. Unfortunately the derivative (and subsequent gradient) is not defined on the rationals.

Machine learning techniques rely heavily on calculating the gradient. To get around this problem we take the gradient analytically outside of the computer and then implement it. With enough precision our gradient should be approximately correct.

2 Related Work

With the introduction of deep learning models introduced massive requirements on system memory, related to both the loading of the model and the data, the gradients calculated during the forward pass, and storing gradients for backpropagation. As a result, technique named model quantization was released that reduces the precision of model parameters to reduce the memory requirements of a model and its data. First introduced in the field of computer vision (Jacob et al., 2017), researchers sought to streamline the deployment of deep learning models on resource-constrained devices by lowering the bit-width of weights and activations without compromising performance too significantly.

As the era of larger models such as large language models (LLMs) emerged, the need for efficient deployment became even more pressing. The massive size and computational demands of these models posed significant challenges for practical applications. To address this, researchers adapted existing quantization techniques to LLMs, exploring methods like post-training quantization and quantization-aware training.

Post-training quantization involves applying quantization to a pre-trained model, typically by rounding weights and activations to lower precision formats. Quantization-aware training, on the other hand, incorporates quantization into the training process itself, allowing the model to learn representations that are more robust to quantization.

Modern quantization methods often utilize a wide variety of numerical precision bit widths. Popular quantization methods often utilize floating point 32, floating point 16 or even floating point 8 values, all of which conform to the IEEE 754 standard (Bhandare et al., 2019; Dettmers et al., 2023). Extreme attempts at model compression have recently achieved 2-bit and 1.58-bit precision, significantly reducing model memory requirements (Ma et al., 2024; Egiazarian et al., 2024).

However, reduced float precision results in substantial performance decreases due to the imprecision of the calculations during the forward passes of deep learning models. Ma et al., 2024 notes an average 5.88% loss in accuracy on a 2-bit Llama-2-7b model when benchmarked against its 16-bit equivalent on seven different benchmarks.

Recent work on model compression while maintaining high numerical precision mainly revolve around the brain float, often abbreviated bfloat, introduced by Google in 2019 (Wang and Kanwar, 2019). Bfloat16, referring to its bit width as a 16-bit floating-point

format, reduces the mantissa width to 8 bits while retaining the full 8-bit exponent of the 32-bit IEEE 754 single-precision format, allowing for rapid conversion to and from a 32-bit IEEE 754 format. This results in a 16 bit width with a 32 bit floating point precision, trading a smaller impact in numerical precision for effective model compression. This format is widely supported by modern hardware accelerators, including GPUs, TPUs, and AI-specific processors, enabling efficient training and inference of large-scale machine learning models.

3 Methods

While model compression algorithms mainly align along building IEEE 754 compliant floating point representations to further compress the model, the IEEE 754 standard does not provide a floating point representation that is entirely optimized for deep learning.

In the context of deep learning, FP16 (16-bit floating point) and bfloat16 are both widely used reduced-precision formats designed to balance memory efficiency and computational speed. FP16 utilizes 1 bit for the sign, 5 bits for the exponent, and 10 bits for the mantissa, offering higher precision in representing values but with a smaller dynamic range (approximately ±65504). In contrast, bfloat16 allocates 1 bit for the sign, 8 bits for the exponent, and 7 bits for the mantissa, providing a significantly larger dynamic range approximately 3.410^{38} similar to that of FP32.

While bfloat16 sacrifices precision in the mantissa to accommodate a larger exponent, making it suitable for models with large numerical ranges, FP16 is preferred when higher precision is required for specific computations. The choice between FP16 and bfloat16 depends on the specific demands of the task, with bfloat16 being more prevalent in training large-scale neural networks, where maintaining numerical stability across a broader range of values is critical.

However, both of these methods, while allowing for large dynamic ranges, are often excessive for deep learning. Due to floating points having more granular precision on numbers nearer to 0 due to the ability to be represented by negative exponents, deep learning model weights often exist in the range between -1 to 1 or are even smaller. As seen in Figure 3are almost entirely contained within .1 to -.1. We assume that higher precision is needed during training because of the gradient problem described above. We believe that some part of numerical instability, vanishing gradients, exploding gradient and the effectiveness of weight regularization is caused by less precision. Our approach increases the bias of the 32 bit float. While this means it cannot be run on a floating point unit because it does not follow IEEE protocol, theoretically if implemented on hardware could yield better results with the exact same memory impact.

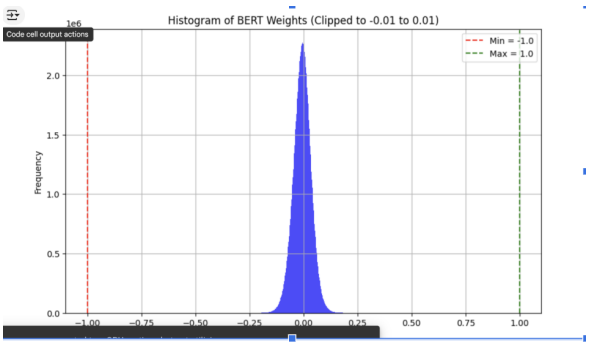

Figure 3: Log scale graph of BERT weights

While conceptually straightforward, our approach involved several implementation challenges. It is to increase the bias of the floating point number. This will allow our floats to represent more values between 1 and -1. Each increase in the bias the number of possible values by 23^2. This is because there are 23 possible mantissas. Increasing the bias means set of mantissas which have a corresponding negative exponent. We experiment with a wide range different biases. Because this does not follow the IEEE 754 standard we had to implement the floating point arithmetic virtually. We implemented

floating point addition and multiplication. To do subtraction we flipped the sign bit and did addition. To do division we flipped the sign of the exponent and did division. To do raise something to a whole power we did repeated multiplication. This was implemented in cpp and compiled using clang.

In order to compare results we used MNIST as it is a classifier. We also experimented with finding minima of random functions using a double as ground truth.

4 Experiments

Our first experiment was finding minima in different functions. All functions were in the form $f(x) = (x - a)^2$. We set a to a variety of values but will only be showing the results for the following values $1.0e^{-1}$,$1.0e^{-31}$.$1.0e^{-36}$,$1.0e^{-38}$. These values were chosen for a few reasons. The first value is a baseline which shows that having a bias of 1 can lead to poor results. The next values we found were the most interesting. The experiments not shown followed the expected pattern. For example bias levels from 90 and above could find a close minima for the value $1.0e^{-27}$. It additionally should be noted that a bias term of 127 corresponds to a 32 bit float. In our experimentation using a our float with a bias term of 127 or a 32-bit float had in the same results.

Our second experiment was moving to something more sophisticated. We experimented using a Artificial Neural Network (ANN)(Rosenblatt, 1958) on the classification task Modified National Institute of Standards and Technology database (MNIST) (LeCun et al., 1998). MNIST is a collection of images of handwritten digits. Each image is 28 x 28 and in black and white. There are 70,000 total images. 60,000 in the training set, 10,000 in the test set. Pixel values range between 0 and 255. We normalize these values to be between 0 and 1. This is not out of the ordinary for this task. The bias level of our floats is set 240. We trained an ANN of the following size. 784 input layer, a single hidden layer of size 10 output layer of size 10 (for each digit). This network was train for a single epoch. The learning rate was set to 0.1. We initialized our weights to be random We repeated this training 100 times.

We ran the experiment using the following floats: 16 bit floats, 32 bit biased floats, 32 bit floats and 64 bit floats.

4.1 Results

4.2 Experiment 1

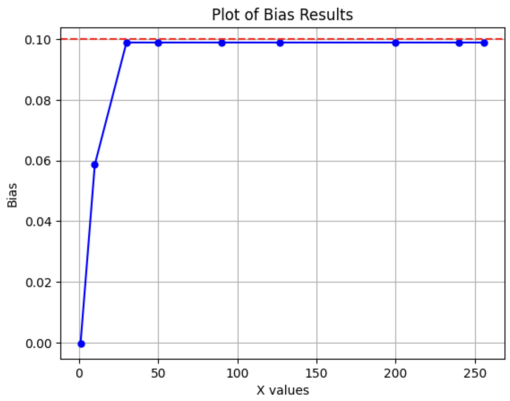

Figure 4: Minimizing the function $(x - .1)^2$

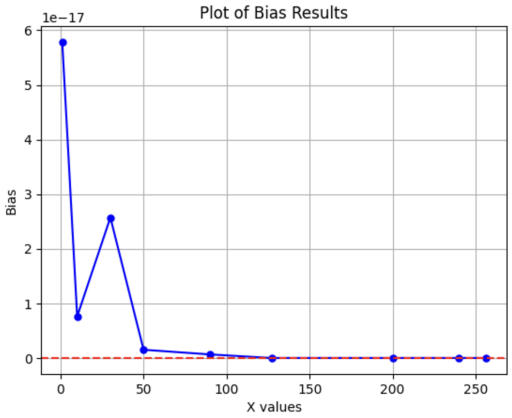

Figure 5: Minimizing the function $(x - .1 * 10^{-31})^2$

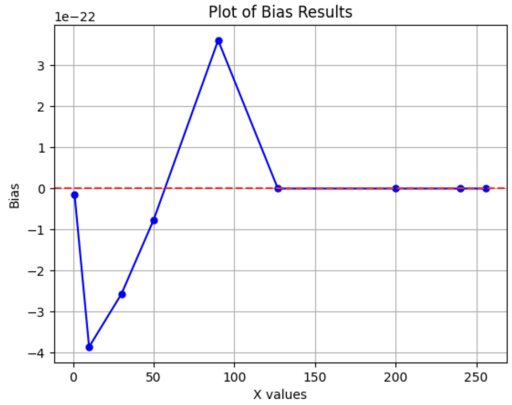

Figure 6: Minimizing the function$((x - .1 * 10^{-36})^2$

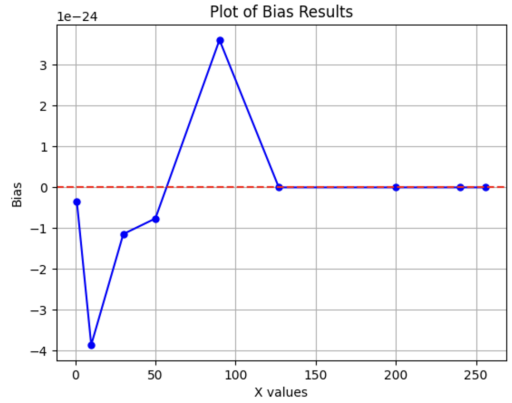

Figure 7: Minimizing the function $(x - .1 * 10^{-38})^2$

Bias Level	1.0e-1	1.0e-31	1.0e-36	1.0e-38
1	-0.00028823	-5.78484e-17	-1.50481e-23	-3.38566e-25
10	0.0586076	-7.61774e-18	-3.8566e-22	-3.85664e-24
30	**0.0988471**	2.571e-17	-2.57108e-22	-1.14266e-24
50	0.0988471	-1.50478e-18	-7.61782e-23	-7.61791e-25
90	0.0988471	6.526478e-19	3.61152e-22	3.61132e-24
127	0.0988471	**9.99999e-31**	8.92551e-36	-1.67277e-39
200	0.0988471	9.99999e-31	**9.99999e-36**	**9.99999e-38**
240	0.0988471	9.99999e-31	9.99999e-36	9.99999e-38
256	0.0988471	9.99999e-31	9.99999e-36	9.99999e-38

Table 1: Bias Results for Different bias levels

4.3 Experiment 2

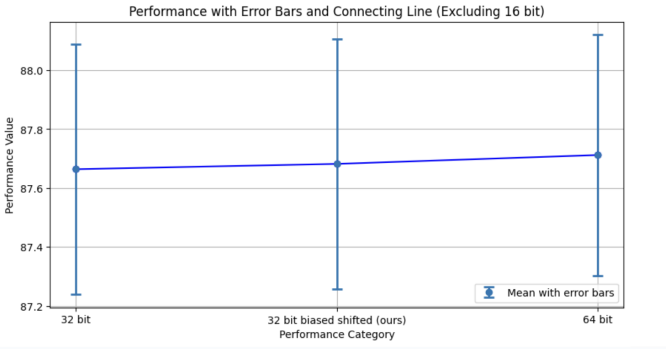

Figure 8: Results (omitting 16 bit) of ANN on MNIST

Representation	Mean	Variance
16 bit	9.800	0.001245
32 bit	87.664	0.180649
240 biased 32 bit	87.682	0.179922
64 bit	87.712	0.166707

Table 2: Comparison of different floating-point representations' mean and variance.

5 Analysis

5.1 Experiment 1

Our experimentation showed reasonable and expected results. When decreasing the magnitude of the minima our floating point number found it when its bias was larger. When the minima was past a certain threshold the float was unable to approximate it. This is because the gradient becomes so small that it does not update the float. For example, if the float's value is 1.0e-23 and the gradient is 1.0e-27. Even if the gradient exists (as it maybe calculated to 0) it may not update the float. The amount the gradient's mantissa is shifted before mantissa addition results in the mantissa being 0. Setting your learning rate too small like, setting it too high, may result in failing to find the best minima. The the learning rate will impact the the lower bound of your resulting value. The lower bound is a function of the floats' lower bound due to the gradient being too small to update. This all might seem useless as the differences between $1.6e-39$ and $9.9e-38$ are very small but consider larger models like BERT and LLMs. Most of the weights in LLMs are so small that finding the correct minima likely matters. LLMs are pretrained with 32-bit float and then quantized down. Training on small floats results in poor performance. Quantization occurs after pre-training. Additionally, quantize to 16 bit floats with our bias likely will result in better performance since the biased weights will be closer the 32 bit weights.

5.2 Experiment 2

In our second experiment we show that increasing the bias results in better performance on common datasets. As the precision increased we found that mean performance increased. We also found that variance decreased. This pattern followed with our 240 biased 32 bit float. Our mean increased by .001% while our variance decreased .4% compared with 32 bit float. Considering the magnitude of difference between 64 bit and 32 bit floats our result is reasonable. It implies our technique does in fact improve performance.

5.3 Future work and Limitations

IEEE 754 floats are extremely versatile and expressive. With this standard computer engineers were able to implement floating point arithmetic in hardware. This has lead to an extreme speed up. Instead of having to do many of the steps and checks as separate instructions in the CPU they were able to implement it in metal so effectively that it is able to be calculated in 1 cycle. The unit that is responsible for this is called the floating point unit. Our method does not use the floating point unit to do its calculations. Like previously state, we had to implement the arithmetic virtually in c++. This has resulted a dramatic increase in time. Additionally, which we have suggested that this could be used in combination with quantization, our virtual implementation using unsigned 32 bit floats to store the exponent, mantissa and bias. This means we are storing 97 bits for a single float. While in theory this could be implemented in hardware in just 32 bits this would be an extremely large undertaking.

Implementation with 16 bit and 8 bit floats would be the logical next step. Additionally writing the code in assembly would drastically improve time and memory usage. However, that would not solve the major issues with the technique. Floating point units are extremely effective and any software implementation will pale in performance. An especially ambitious and effective future project would be implementing this on an FPGA. With that performance you could cast the weights of an LLM into this custom bias float. This would be a better test of these floats. Again it still would struggle with time performance compared to floats on a GPU.

6 Conclusion

In this study, we investigated the impact of increasing the bias in floating-point representation on machine learning tasks, on minima finding and the MNIST classification task. We found that changing the bias term of 32-bit floating-point numbers improved model weight precision, resulting in more accurate minima in optimization tasks and better performance in neural network training. Our experiments found that higher bias values improved the ability to represent smaller numbers with greater precision, in the range of -1 to 1, which is critical for gradient stability in machine learning models. This suggests there could be improved performance with LLMs and bigger models meaning a lack of lost performance due to quantization.

Our results show that a custom 32-bit floating-point representation with an increased bias term performs similarly or slightly better than standard.

While our study may not have any actionably items due to hardware limitations it is still gives insightful on the inter workings of training.

Ultimately, this study suggests that revisiting the design of floating-point representation in machine learning models could lead to more efficient and stable training procedures, with potential benefits for a wide range of applications in artificial intelligence and beyond.

7 Contributions of team members

Nick Sullivan - Implementation, Paper and Presentation
Vincent Siu - Paper and Presentation
Jack McCall - Presentation and Implementation
Thein Hoang - Presentation and Paper
Wonsang Lee - Presentation and Paper

References

Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram A. Saletore. 2019. Efficient 8-bit quantization of transformer neural machine language translation model. *CoRR* abs/1906.00532. http://arxiv.org/abs/1906.00532.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. https://arxiv.org/abs/2305.14314.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. 2024. Extreme compression of large language models via additive quantization. https://arxiv.org/abs/2401.06118.

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. https://arxiv.org/abs/1712.05877.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86(11):2278–2324.

Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. 2024. The era of 1-bit llms: All large language models are in 1.58 bits. https://arxiv.org/abs/2402.17764.

Frank Rosenblatt. 1958. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review* 65(6):386–408.

Shibo Wang and Pankaj Kanwar. 2019. Bfloat16: The secret to high performance on cloud tpus. *Google Cloud* https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.