The Era of 1-bit LLMs: Training Tips, Code and FAQ

Shuming Ma **Hongyu Wang** **Furu Wei** **BitNet Team**
https://aka.ms/GeneralAI

Abstract

We present details and tips for training 1-bit LLMs. We also provide additional experiments and results that were not reported and responses to questions regarding the "The-Era-of-1-bit-LLM" paper [MWM+24]. Finally, we include the official PyTorch implementation of BitNet (b1.58 [MWM+24] and b1 [WMD+23]) for future research and development of 1-bit LLMs.

Believing is seeing.

1 Training Tips

The $S-$shape Loss Curve

The loss curve during high-precision training typically exhibits an exponential shape. However, the curve observed during 1-bit training often follows an $S-$shaped pattern. As illustrated in Figure 1a, there is a significant reduction in loss towards the end of the training process. A similar phenomenon has been observed in the training of binarized neural machine translation [ZGC+23]. This suggests that intermediate evaluations may not accurately predict the final performance of 1-bit LLM training.

Learning Rate

In our experiments, the learning rate followed a two-stage approach, as illustrated in Figure 1c.

1. The first stage employed the standard learning rate scheduling, but with a higher peak learning rate. This is because 1-bit LLMs are more stable than their full-precision counterparts.
2. We then decayed the learning rate midway through the training process, resulting in a lower peak learning rate for the second stage.

In our experiments, we employed a significantly higher learning rate for our model compared to full-precision LLMs. This decision was informed by the findings from the BitNet b1 [WMD+23], which demonstrated the crucial importance of a larger learning rate for optimizing 1-bit models effectively. Our initial trials revealed that full-precision models did not benefit from a similarly elevated learning rate. This is likely because the learning rates used for these models have already been well-tuned by prior work, and increasing them led to instability during the optimization process.

Halfway through the training process, we decayed the learning rate, a strategy that yielded better performance in our experiments. As illustrated in Figure 1b, we observed a significant reduction in loss occurring when the learning rate was decayed, rather than at the end of the training phase. This approach made the performance more predictable during the training process. However, full-precision models did not benefit from a similar learning rate decay strategy, likely because their loss curves do not exhibit an $S-$shaped pattern. There has also been recent work on multi-step learning rate schedulers [BCC+24] for training language models, but these are primarily aimed at facilitating continual training rather than improving convergence.

In addition to the linear decay scheduler, we conducted experiments employing a cosine scheduler for both BitNet b1.58 and the full-precision LLaMA LLM. While the cosine scheduler exhibited slightly better convergence during the middle of training, there was no discernible advantage over

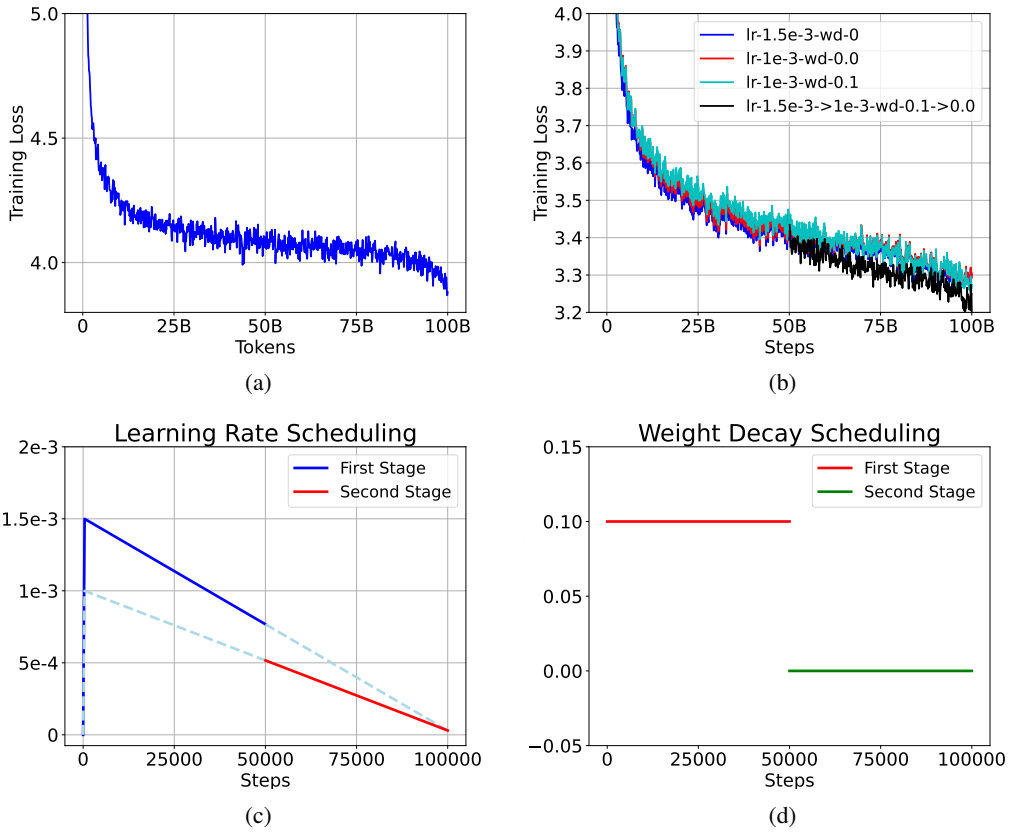

Figure 1: (a) The loss curve of 1.58-bit LLM is in $S-$shape. (b) The ablation of the learning rate and weight decay scheduling for BitNet b1.58. (c) The learning rate scheduler in our experiments. (d) The weight decay scheduler in our experiments.

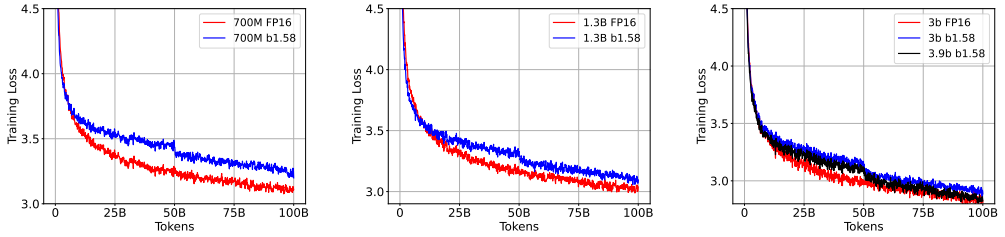

Figure 2: Training loss curves across different model sizes. The gap between full-precision models and BitNet b1.58 becomes narrower as the models scale.

the linear decay approach as the training progressed towards completion. For the sake of simplicity, we opted for the linear decay scheduler throughout our experiments. However, we believe that the cosine scheduler could also be an effective approach for this two-stage training methodology used in BitNet b1.58.

Weight Decay

As depicted in Figure 1d, the weight decay strategy also followed a two-stage approach.

1. During the first stage, we followed the LLaMA LLM by employing a weight decay of 0.1.
2. In the second stage, weight decay was disabled.

For full-precision training, weight decay acts as a regularizer, preventing over-fitting and improving training stability by constraining the magnitude of large weights. However, the effect of weight decay in 1-bit training differs substantially, as it is applied to the latent weights rather than the 1-bit weights themselves. In mixed-precision training, the magnitude of the latent weights can be interpreted as a confidence score for the corresponding 1-bit weights [LSL+21]. Consequently, a large weight decay leads to lower confidence scores for the 1-bit weights, causing them to change more frequently. To mitigate this, we removed weight decay for the second half of the training, allowing the model to converge rather than update frequently. This weight decay scheduler is not applicable to full-precision models, as the role of weight decay fundamentally differs between full-precision and 1-bit learning.

Settings for Reproducing the "The-Era-of-1-bit-LLM" Paper

Table 1 and 2 summarize the configurations and optimization hyper-parameters employed in our experiments. Additionally, we provide the training loss curves in Figure 2. These curves illustrate that the gap between the full-precision LLaMA LLM and BitNet b1.58 diminishes as the model size increases, indicating that 1-bit language models perform better with larger model size. Interestingly, while there is no discernible gap between the 3B versions of BitNet b1.58 and LLaMA LLM in terms of validation loss and end-task accuracy, a slight difference exists in the training loss. This suggests that 1.58-bit models may exhibit better generalization capabilities and be less prone to overfitting. We leave a more in-depth analysis of this phenomenon for future work.

Size	Hidden Size	GLU Size	#Heads	#Layers	Batch Size	# Tokens	Seq Length
700M	1536	4096	24	24	1M	100B	2048
1.3B	2048	5460	32	24	1M	100B	2048
3B	3200	8640	32	26	1M	100B	2048
3.9B	3200	12800	32	26	1M	100B	2048

Table 1: Model configurations for both BitNet b1.58 and LLaMA LLM.

Model	Size	Learning Rate	Weight Decay	Warm-up	Adam β
BitNet b1.58	700M	$1.5 \times 10^{-3} \rightarrow 1 \times 10^{-3}$	$0.1 \rightarrow 0$	375	(0.9, 0.95)
	1.3B	$1.2 \times 10^{-3} \rightarrow 8 \times 10^{-4}$	$0.1 \rightarrow 0$	375	(0.9, 0.95)
	3B	$1.2 \times 10^{-3} \rightarrow 8 \times 10^{-4}$	$0.1 \rightarrow 0$	375	(0.9, 0.95)
	3.9B	$1.2 \times 10^{-3} \rightarrow 8 \times 10^{-4}$	$0.1 \rightarrow 0$	375	(0.9, 0.95)
LLaMA LLM	700M	2.5×10^{-4}	0.1	375	(0.9, 0.95)
	1.3B	2.0×10^{-4}	0.1	375	(0.9, 0.95)
	3B	2.0×10^{-4}	0.1	375	(0.9, 0.95)

Table 2: Hyper-parameters for both BitNet b1.58 and LLaMA LLM training.

2 FAQ

Alternative to ternary {-1, 0, 1}?

- {-1, 1}: This was the implementation in our original BitNet b1 work [WMD+23]. While it demonstrated a promising scaling curve, the performance was not as good as the ternary approach, especially for smaller model sizes.
- {0, 1}: We found that optimization with this configuration was highly unstable, leading to exploding gradients and loss divergence early in the training process.
- More bits like {-2, -1, 0, 1} or {-2, -1, 0, 1, 2}: While this is a reasonable approach due to the availability of shift operations for -2 and 2 to eliminate multiplication, ternary quantization already achieves comparable performance to full-precision models. Following Occam's Razor principle, we refrained from introducing more values beyond {-1, 0, 1}.

The distribution of {-1, 0, 1} in model weights?

Models	Size	ARCe	ARCc	HS	BQ	OQ	PQ	WGe	Avg.
BitNet b1.58	3B	61.4	28.3	42.9	61.5	26.6	71.5	59.3	50.2
w/ 4-bit KV	3B	60.9	27.3	42.9	61.4	26.2	71.6	59.9	50.0
BitNet b1.58	3.9B	64.2	28.7	44.2	63.5	24.2	73.2	60.5	51.2
w/ 4-bit KV	3.9B	64.1	28.8	44.1	63.7	24.2	73.2	61.1	51.3

Table 3: Zero-shot accuracy on the benchmark. The KV cache of BitNet b1.58 can be losslessly quantized to 4 bits after training.

According to our analysis of the model checkpoints, the distribution is nearly uniform. This is determined by our quantization function formulation. We can adjust the distribution by scaling the γ parameter in Equation (3) of our "The Era of 1-bit LLMs" paper [MWM+24]. For instance, a larger γ increases the percentage of 0, leading to a more sparse weight matrix. However, our preliminary experiments showed that this approach hurts performance.

How about post-training quantization (PTQ) of the full-precision models?

While PTQ has been successful in compressing full-precision models to 8 bits or even 4 (technically 4.5) bits, it struggles to quantize weights further to lower precisions. This is because PTQ can only quantize model weights around the full-precision values, which may not necessarily be where the global minima of the low-precision landscape lie. We conducted PTQ experiments in our BitNet b1 paper [WMD+23], demonstrating that PTQ at 2 bits leads to a significant loss in accuracy. We also attempted to continue training a 1-bit PTQ checkpoint, but did not observe any improvement over training from scratch.

Lower precision for activation and/or KV cache?

The activation in BitNet b1.58 is quantized to 8 bits. Directly reducing the precision to 4 bits results in a significant loss in accuracy. Our preliminary experiments show that the activations before the down-projection have larger outliers than other activations, making them more difficult to quantize. To mitigate this, we retain the activations before the down-projection of the FFN or GLU at 8 bits and quantize the remaining activations to 4 bits. We observe negligible accuracy loss with this hybrid quantization strategy. Furthermore, our analysis reveals that more than 80% of the values before the down-projection are 0, making it possible to accelerate the matrix multiplication by exploiting this (structured) sparsity pattern. As shown in Table 3, the KV cache of our model can be compressed to 4 bits without any accuracy loss. Unlike previous work, we use symmetric quantization functions, which are simpler and easier to optimize for kernel acceleration. We believe there is still room for further compression of the activations and KV cache without sacrificing accuracy, and we leave this as future work.

Training acceleration?

Our current implementation for training our model is still in FP16/BF16, so there is no actual speed-up in our experiments. However, there are significant opportunities for acceleration, especially for larger models. First, low-precision CUDA kernels (e.g., FP8 GEMM) can be used to accelerate the forward and backward computations of BitLinear[1]. Second, 1.58-bit language models can save a substantial amount of memory and communication bandwidth for distributed training. This is particularly beneficial for ZeRO [RRRH20], which conducts an all-gather operation to collect parameters during forward and backward computations.

Will BitNet work for larger models?

As shown in Tables 1 and 2 in the "The Era of 1-bit LLMs" paper, there is a clear trend indicating that the gap between full-precision Transformer LLMs and BitNet b1.58 (and also b1) becomes smaller as the model size increases. This suggests that BitNet and 1-bit language models are even more effective for larger models. Scaling is one of the primary goals of our research on 1-bit LLMs, as we eventually need to scale up the model size (and training tokens) to train practical LLMs.

[1]With FP8 GEMM kernels, we can use FP8 activations for training and quantize to INT8 for inference on GPU devices with CUDA compute capability < 9.0.

3 PyTorch Implementation of BitNet b1.58

There are two steps to change from a LLaMA LLM architecture to BitNet b1.58:

1. Replace all *nn.Linear* in attention and SwiGLU with *BitLinear* (Figure 3);
2. Remove RMSNorm before attention and SwiGLU because *BitLinear* has built-in RMSNorm.

```
def activation_quant(x):
""" Per–token quantization to 8 bits. No grouping is needed for quantization.
Args:
    x: an activation tensor with shape [n, d]
Returns:
    y: a quantized activation tensor with shape [n, d]
"""
    scale = 127.0 / x.abs().max(dim=−1, keepdim=True).values.clamp_(min=1e−5)
    y = (x ∗ scale).round().clamp_(−128, 127) / scale
    return y

def weight_quant(w):
""" Per–tensor quantization to 1.58 bits. No grouping is needed for quantization.
Args:
    w: a weight tensor with shape [d, k]
Returns:
    u: a quantized weight with shape [d, k]
"""
    scale = 1.0 / w.abs().mean().clamp_(min=1e−5)
    u = (w ∗ scale).round().clamp_(−1, 1) / scale
    return u

class BitLinear(nn.Linear):
"""
This is only for training, and kernel optimization is needed for efficiency.
"""
    def forward(self, x):
    """
    Args:
        x: an input tensor with shape [n, d]
    Returns:
        y: an output tensor with shape [n, d]
    """
        w = self.weight # a weight tensor with shape [d, k]
        x_norm = RMSNorm(x)
        # A trick for implementing Straight–Through–Estimator (STE) using detach()
        x_quant = x_norm + (activation_quant(x_norm) − x_norm).detach()
        w_quant = w + (weight_quant(w) − w).detach()
        y = F.linear(x_quant, w_quant)
        return y
```

Figure 3: Pytorch code for the BitLinear component in training BitNet b1.58. It requires additional efforts to improve the training efficiency, such as kernel fusion.

As for the inference, there are some changes for efficiency.

1. The model weights are offline quantized to 1.58 bits.
2. The standard *F.linear* operation is replaced with a customized low-bit kernel.
3. The scaling factors for both weight quantization and activation quantization are applied after the *F.linear* operation.

4. There is no need to implement the Straight-Through Estimator (STE) trick.
5. The RMSNorm operation can be fused with the activation quantization.

```
def activation_norm_quant(x):
""" RMSNorm & Per-token quantization to 8 bits. It can be implemented as a fused kernel.
Args:
    x: an activation tensor with shape [n, d]
Returns:
    y: a quantized activation tensor with shape [n, d]
    scale: a scalar for dequantization with shape [1]
"""
    x = RMSNorm(x)
    scale = 127.0 / x.abs().max(dim=-1, keepdim=True).values.clamp_(min=1e-5)
    y = (x * scale).round().clamp_(-128, 127)
    return y, scale

class BitLinear(nn.Linear):
"""
This is only for inference. The weights should been quantized in advance.
"""
    def forward(self, x):
    """
    Args:
        x: an input tensor with shape [n, d]
    Returns:
        y: an output tensor with shape [n, d]
    """
        w = self.weight # a 1.58-bit weight tensor with shape [d, k]
        w_scale = self.weight_scale # a full-precision weight scale tensor with shape [1]
        x_quant, x_scale = activation_norm_quant(x)
        y = gemm_lowbit_kernel(x_quant, w) / w_scale / x_scale
        return y
```

Figure 4: Pytorch code for the BitLinear component during inference.

4 PyTorch Implementation of BitNet b1

Similarly, we provide the implementation of the original 1-bit BitNet by replacing the function of *weight_quant(w)* as in Figure 3.

```
def weight_quant(w):
""" Per-tensor quantization to 1 bits. No grouping is needed for quantization.
Args:
    w: a weight tensor with shape [d, k]
Returns:
    u: a quantized weight with shape [d, k]
"""
    scale = w.abs().mean()
    e = w.mean()
    u = (w - e).sign() * scale
    return u
```

Figure 5: Pytorch code for the BitLinear component in BitNet b1.

References

[BCC+24] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, Alex X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. Deepseek LLM: scaling open-source language models with longtermism. *CoRR*, abs/2401.02954, 2024.

[LSL+21] Zechun Liu, Zhiqiang Shen, Shichao Li, Koen Helwegen, Dong Huang, and Kwang-Ting Cheng. How do adam and training strategies help bnns optimization. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 6936–6946. PMLR, 2021.

[MWM+24] Shuming Ma, Hongyu Wang, Lingxiao Ma, Lei Wang, Wenhui Wang, Shaohan Huang, Li Dong, Ruiping Wang, Jilong Xue, and Furu Wei. The era of 1-bit llms: All large language models are in 1.58 bits, 2024.

[RRRH20] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 20. IEEE/ACM, 2020.

[WMD+23] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. Bitnet: Scaling 1-bit transformers for large language models. *CoRR*, abs/2310.11453, 2023.

[ZGC+23] Yichi Zhang, Ankush Garg, Yuan Cao, Lukasz Lew, Behrooz Ghorbani, Zhiru Zhang, and Orhan Firat. Binarized neural machine translation. *CoRR*, 2023.