VeLoRA: Memory Efficient Training using Rank-1 Sub-Token Projections

Roy Miles, Pradyumna Reddy, Ismail Elezi, Jiankang Deng

NEURAL INFORMATION PROCESSING SYSTEMS

TLDR;

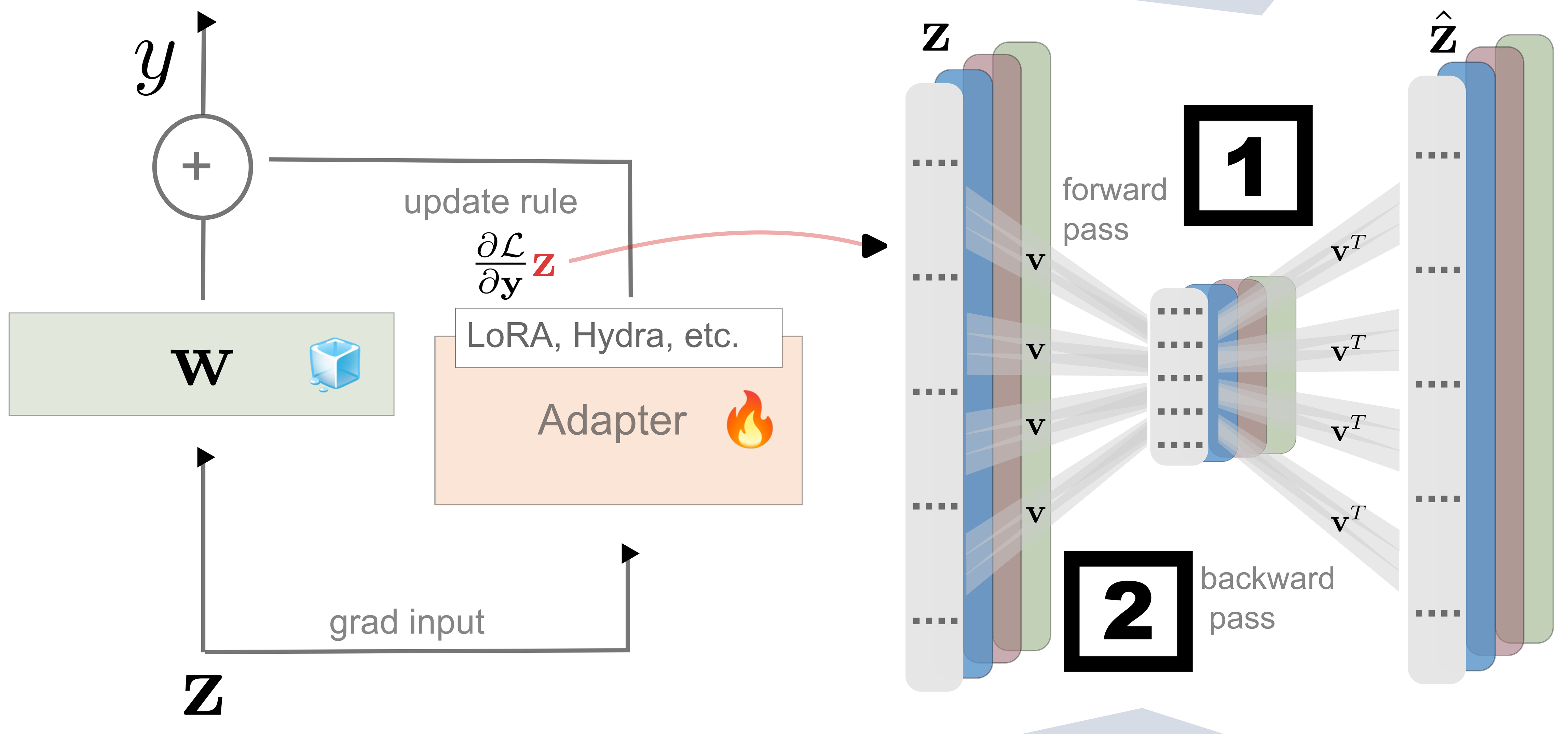
1 **Compress** activations by projected by dividing and projecting the tokens during the forward pass.

2 Cheap and coarse **reconstruction** during the backwards pass.

3 **Result:** Significant memory reduction for both fine-tuning and pre-training LLMs!

$$\mathbf{Z} \xrightarrow{group(\cdot)} \mathbf{z} \in \mathbb{R}^{B \times ND/M \times M} \xrightarrow{compress(\cdot\ ;\ \mathbf{v})} \mathbf{z}_p \in \mathbb{R}^{B \times ND/M \times 1}$$

Compress activations by first dividing the activations into sub-tokens and then computing their cosine similarity to a frozen vector v. The sub-token size can control the level of compression and subsequently the memory reduction for training.



Reconstruct an approximation of the original activations by projecting back onto v. Through careful initialization for v, we can preserve a lot of structural information needed for good convergence and model performance.

$$\mathbf{z}_p \xrightarrow{reconstruct(\cdot\ ;\ \mathbf{v})} \hat{\mathbf{z}} \in \mathbb{R}^{B \times ND/M \times M} \xrightarrow{ungroup(\cdot)} \hat{\mathbf{Z}} \in \mathbb{R}^{B \times N \times D}$$

Query	Key	Value	Down	Memory (GB)	Acc
— none —				1.67	38.1
✓				1.42	36.2
	✓			1.42	36.2
		✓		1.42	36.7
			✓	1.01	38.9
✓		✓		1.18	37.4
		✓	✓	0.76	**39.5**
✓		✓	✓	0.51	38.4
✓	✓	✓	✓	0.24	37.0

Layer Selection

VeLoRA is most effective on the down projection layers where the input activations are large.

Epochs	QLoRA	VeLoRA
1	36.4	**36.7**
2	37.3	**37.5**
3	**38.4**	38.1
4	39.1	**39.5**

Convergence

Despite approximating the gradients, we find that VeLoRA does not impact the training converge for pre-training or fine-tuning.

M	Memory (MB)	Acc
D / 64	865	37.9
D / 32	808	**39.5**
D / 16	779	39.3
D / 8	764	37.2

Sub-Token Size

Sub-token size provides a way of tuning the memory v.s. performance trade-off.

Method	Acc
Random	36.8
SVD	37.1
Fixed average	**39.5**
Running average	38.9

Initialization

Initialization strategy for v is important for maintaining good performance. We find a simple batch average is very effective.

LoRA

$$y = Wx + ABx = (W + AB)x$$

Following common practice and the derivation given by FLoRA [1], we can express the update as:

$$W' = W + A_0\left(B_0 - \eta\frac{dL}{dB}\right) \approx \boxed{W - \eta\tilde{g}A_0A_0^T}$$

LoRA does induce low-rank gradient updates.

VeLoRA

$$\frac{dL}{dW} \approx \frac{dL}{dy} \cdot \left(\left(\frac{dy}{dW} \cdot v\right) v^T\right) = \left(\frac{dL}{dy} \cdot \frac{dy}{dW}\right) vv^T$$

For simplicity, consider the case of a single sub-token. VeLoRA projects this sub-token using a fixed rank-1 projection.

$$W' = W - \eta\frac{dL}{dW} = \boxed{W - \eta\tilde{g}vv^T}$$

VeLoRA can be seen through the lens of LoRA using a data-driven initialization for A.

Pre-training

	60M	130M
Full-Rank	33.52 (1.30G)	25.08 (2.32G)
GaLore	34.88 (1.27G)	25.36 (2.02G)
LoRA	34.99 (0.86G)	33.92 (1.24G)
FLoRA	34.35 (1.27G)	25.88 (2.01G)
VeLoRA	**33.76** (1.18G)	**25.29** (1.83G)
r/d_{model}	128 / 256	256 / 768
Training Tokens	1.1B	2.2B

Fine-tuning

LLaMA Size	7B		13B		Mean
Method	Alpaca	Memory	Alpaca	Memory	
LoRA w/ BFloat16	38.4	8.79	47.2	15.82	42.8
LoRA w/ Float4	37.2	5.77	47.3	9.91	42.3
QLoRA	39.0	5.77	47.5	9.91	43.3
+ VeLoRA	**39.5**	**4.88**	**48.0**	**8.48**	**43.8**

3 We confirm the effectiveness of our algorithm as being complimentary to many state-of-the-art PEFT methods on the VTAB-1k fine-tuning benchmark. Furthermore, we outperform QLoRA for fine-tuning LLaMA and show competitive performance against other memory-efficient pre-training methods on the large-scale C4 dataset.

References

[1] T. Dettmers, et. al. Qlora: Efficient finetuning of quantized llms. NeurIPS 2023

[2] E. J. Hu, et. al. Lora: Low-rank adaptation of large language models. ICLR 2022.

[3] Y. Hao, et. al. Flora: Low-rank adapters are secretly gradient compressors, 2024. ICML 2024

Implementation is quite simple!

Algorithm 1 VeLoRA, Pytorch-like

```
def forward(input, weight, v):
    # v: M x 1

    # forward compute is preserved
    out = input @ weight

    # compute vector similarity
    z = compress(group(input), v)

    save_for_backward(z, weight, v)
    return out

def backward(ctx, grad_output):
    z, weight, v = saved_tensors

    # reconstruct the input
    input = ungroup(reconstruct(z, v))

    # compute gradients
    grad_input = grad_output @ weight
    grad_weight = grad_output.T @ input

    return grad_input, grad_weight
```

GitHub