MCTensor: A High-Precision Deep Learning Library with Multi-Component Floating-Point

Tao Yu*, Wentao Guo*, Jianan Canal Li*, Tiancheng Yuan*, Christopher De Sa

*Equal Contribution

Department of Computer Science, Biological Engineering, and System Engineering

Cornell University

Cornell Bowers C·IS
College of Computing and Information Science

High Precision Computations

Applications: dynamical systems (Taylor methods) [1], computational geometry (delaunay triangulation) [2], hyperbolic deep learning [3,4] …

How to achieve high precision arithmetic?

— multiple-digit: a sequence of digits coupled with a single exponent, e.g., GNU Multiple Precision (GMP) Arithmetic Library [5], Julia's BigFloat type [6] …

— multiple-component (MCF) [7]: an unevaluated sum of multiple ordinary floating-point numbers (e.g., float16, float32)

What's the performance difference?

— multiple-digit can represent compactly a much larger range of numbers;

— multiple-component can use existing floating-point accelerators and hence faster.

Outline

- MCTensor Library
- Computing & Learning with MCTensor
- Error Analysis of MCTensor
- Experiment Evaluation
- Conclusions and Future Work

MCTensor Library

- MCF as underlying representation using an "expansion":

$$x = (x_0, x_1, \cdots, x_{nc-1}) = x_0 + x_1 + \ldots + x_{nc-1}$$

where nc is the number of components and each component x_i is an ordinary floating-point (e.g., PyTorch FloatTensor).

- MCTensor Object

$$x\{fc, \text{tensor}, nc\}$$

- Gradient of a MCTensor

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial (x_0 + x_1 + \cdots + x_{nc-1})} = \frac{\partial f}{\partial x_i}$$

Computing with MCTensor

Basic operators based on inputs:

- **MCTensor** and **Tensor**: Grow-ExpN (add), ScalingN (multiply), DivN (divide)
- **MCTensor** and **MCTensor**: Add-MCN (add), Mul-MCN (multiply), Div-MCN (divide)
- **MCTensor**: Exp-MCN (exp), Square-MCN (square)

Algorithm 1 Grow-ExpN

Input: nc-MCTensor x, PyTorch Tensor v
initialize $Q \leftarrow v$
for $i = 1$ to nc **do**
  $k \leftarrow nc + 1 - i$
  $(Q, h_k) \leftarrow$ **Two-Sum**(x_{k-1}, Q)
end for
$h \leftarrow (Q, h_1, \cdots, h_{nc})$
Return: Simple-Renorm(h, nc)

where **Two-Sum(Tensor1, Tensor2)** → (result, error), **Simple-Renorm(h, nc)** → moves zeros backward and outputs a MCTensor with *nc* component.

Matrix Operators

- **AddMM-MCN** (torch.addmm)
- **MV-MCN** (torch.mv)
- **Matmul-MCN** (torch.matmul)
- …

We extend and broadcast various computations supported for PyTorch Tensor to MCTensor! Compatible with PyTorch, and easy to plug in.

Learning with MCTensor

MCModule: e.g., MCLinear, MCEmbedding, MCSequential …

MCActivation: e.g., MCSoftmax, MCReLU, MC-GELU …

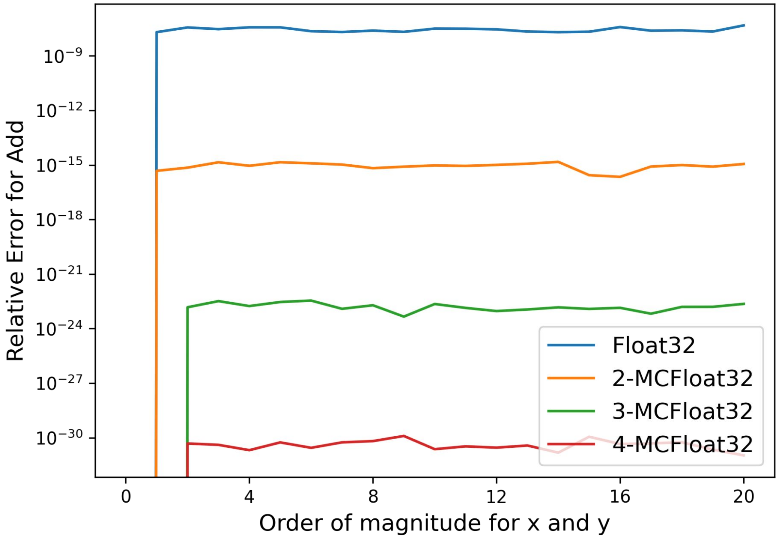
MCOptimizer: e.g., MC-SGD, MCAdam …

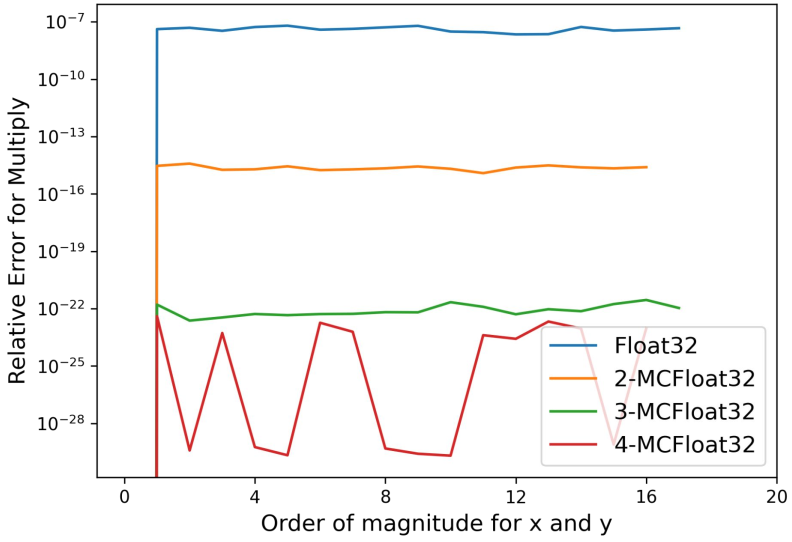
```python
class MCLinear(MCModule):
    def __init__(self, in_features, out_features, nc, bias=True):
        super(MCLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.nc = nc
        self.weight = MCTensor(out_features, in_features, nc=nc, requires_grad=True)
        if bias:
            self.bias = MCTensor(out_features, nc=nc, requires_grad=True)
        else:
            self.bias = None

    def forward(self, input):
        return F.linear(input, self.weight, self.bias)
```

```python
model = MCModel(nc=nc)
optimizer = MCSGD(mc_model.parameters())

for x, y in train_dataset:
    optimizer.zero_grad()
    y_hat = model(x)
    loss = loss_fn(y_hat, y)
    loss.backward()
    optimizer.step()
```

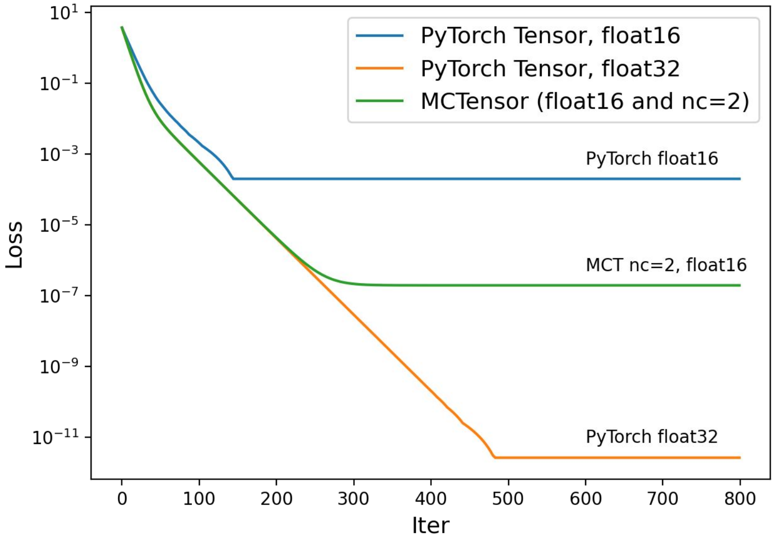
MCTensor can be used in the same way as PyTorch Tensor with a few lines of replacement!

Error Analysis





Relative error of MCTensor computation between values x, y at different magnitudes



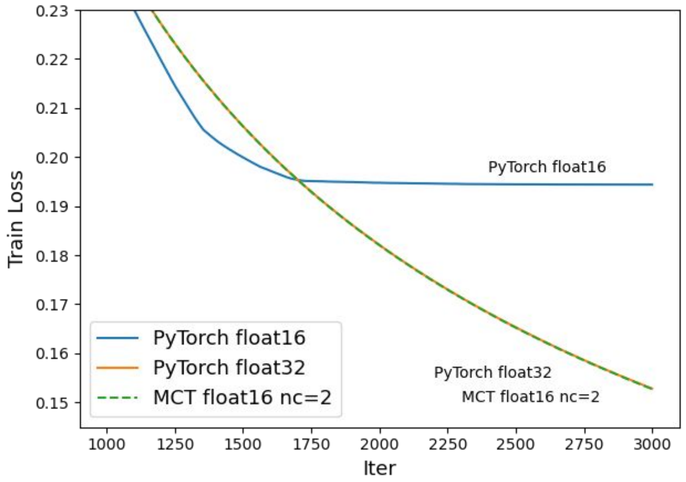
Training loss of linear regression task on a synthetic dataset

Achieve arbitrary precision computations/training by simply increasing the number of components!

Experiments

(1) high precision learning with low precision numbers, using Logistic Regression and MLP model;

(2) hyperbolic embedding using the Poincaré Halfspace model [8].



Logistic Regression on Breast Cancer Dataset

MCTensor demonstrated better precision and performance in both settings!

Model	Training Loss	Testing accuracy
MLP Float16	0.144	90.35
MLP Float32	**0.124**	**91.23**
MLP Float64	**0.124**	**91.23**
MC-MLP (nc=1)	0.144	90.35
MC-MLP (nc=2)	**0.124**	**91.23**
MC-MLP (nc=3)	**0.124**	**91.23**

MLP on Breast Cancer Dataset

Model	MAP (mean $\pm$ sd)	MR (mean $\pm$ sd)
Halfspace (f32)	$91.91\% \pm 0.64\%$	1.399 ± 0.04
Halfspace (f64)	$92.79\% \pm 0.41\%$	1.340 ± 0.07
MC-Halfspace (f64 nc=1)	$93.02\% \pm 0.40\%$	1.296 ± 0.02
MC-Halfspace (f64 nc=2)	$92.77\% \pm 0.28\%$	1.304 ± 0.02
MC-Halfspace (f64 nc=3)	$\mathbf{93.31}\% \pm 0.75\%$	$\mathbf{1.282} \pm 0.03$

Performance of Hyperbolic Models

MAP↑: mean average precision, *MR*↓: mean rank

Conclusions and Future Work

- MCTensor achieves high-precision arithmetic while leveraging the benefits of heavily-optimized ordinary floating-point arithmetic
- MCTensor is easy to use: same as PyTorch code with a few lines of replacement
- MCTensor achieves better precision using low precision numbers
- MCTensor helps relieve the imprecision problem in hyperbolic learning
- A promising future work: design and optimize MCTensor for better efficiency on general tasks

Thank you!
Questions & Comments

References

[1] Bailey et al. High-precision arithmetic in mathematical physics. *Mathematics*, 3(2):337–367, 2015.

[2] Schirra, S. Robustness and precision issues in geometric computation. 1998.

[3] Yu et al. Numerically accurate hyperbolic embeddings using tiling-based models. *Advances in Neural Information Processing Systems*, 32, 2019.

[4] Yu et al. Representing hyperbolic space accurately using multi-component floats. *Advances in Neural Information Processing Systems*, 34, 2021.

[5] Granlund et al. GNU MP: The GNU Multiple Precision Arithmetic Library, 5.0.5 edition, 2012. http://gmplib.org/.

[6] Bezanson et al. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017

[7] Priest, D. M. Algorithms for arbitrary precision floating point arithmetic. University of California, Berkeley, 1991

[8] Nickel et al. Poincaré embeddings for learning hierarchical representations. *Advances in Neural Information Processing Systems*, 30, 2017

Operators	Inputs sizes	FloatTensor	1-MCTensor	2-MCTensor	3-MCTensor
Dot-MCN	$5000, 5000$	$1.61\mu s \pm 3.29$ns	$442\mu s \pm 5.61\mu s$	$656\mu s \pm 1.16\mu s$	$858\mu s \pm 12.2\mu s$
MV-MCN	$(5000 \times 500), 500$	$157\mu s \pm 4.32\mu s$	320ms ± 5.78ms	460ms ± 10.7ms	580ms ± 12.1ms
Matmul-MCN	$(500\times 200), (200\times 50)$	$97.3\mu s \pm 1.1\mu s$	495ms ± 10.8ms	735ms ± 21.7ms	934ms ± 28ms

Operators	Inputs sizes	FloatTensor	1-MCTensor	2-MCTensor	3-MCTensor
Add-MCN	(1000×1000)	$497\ \mu s \pm 6.77\ \mu s$	26.7 ms $\pm 486\ \mu s$	44.7 ms $\pm 379\ \mu s$	64.4 ms $\pm 385\ \mu s$
ScalingN	(1000×1000)	$490\ \mu s \pm 9.69\ \mu s$	33.7 ms $\pm 402\ \mu s$	57.5 ms $\pm 842\ \mu s$	84.7 ms ± 1.78 ms
Mult-MCN	(1000×1000)	$514\ \mu s \pm 15.2\ \mu s$	218 ms ± 4.03 ms	667 ms ± 11.6 ms	$1.4\ s \pm 21.6$ ms
Div-MCN	(1000×1000)	$510\ \mu s \pm 10.6\ \mu s$	80.3 ms $\pm 770\ \mu s$	243 ms ± 3.24 ms	498 ms ± 8.26 ms

Table 4. MCTensor Basic Operators Running Time (mean $\pm$ sd)

$$d_u(\mathbf{x}, \mathbf{y}) = \operatorname{arcosh}\left(1 + \frac{||\mathbf{x}-\mathbf{y}||^2}{2x_n y_n}\right)$$

$$\mathcal{L}(\Theta) = \sum_{(\mathbf{x},\mathbf{y})\in D} \log \frac{e^{-d_u(\mathbf{x},\mathbf{y})}}{\sum_{\mathbf{y}'\in\mathcal{N}(\mathbf{x})} e^{-d_u(\mathbf{x},\mathbf{y}')}}$$