AdderNet:
Do We Really Need Multiplications in Deep Learning?

Chen et al., CVPR 2020

권순기

목차

1. Review

1.1. Motivation

- GPU의 발달로 수십억개의 부동소수점 연산(floating point computation)을 하는 CNN network가 발전함
- GPU의 경우 많은 power가 필요해 mobile device에서 사용하기 힘들고 크기도 매우 큼

=> 모바일 기기에서 사용할 수 있는 **효율적인 Deep Neural Network**에 대한 연구가 필요

- 일반적으로 multiplication이 addition에 비해 느림
- 그러나, Deep Neural Network에서는 forward inference시 float-valued weights와 float-valued activations간 multiplication이 수행됨

=> CNN의 multiplication 연산을 addition 연산으로 바꾸고자 함

1. Review

1.1. Motivation

$$F \in \mathbb{R}^{d \times d \times c_{in} \times c_{out}}$$

$$X \in \mathbb{R}^{H \times W \times c_{in}}$$

$$Y(m,n,t) = \sum_{i=0}^{d} \sum_{j=0}^{d} \sum_{k=0}^{c_{in}} S\big(X(m+i,n+j,k), F(i,j,k,t)\big), \tag{1}$$

- ✓ Y : Output Feature(Filter와 Input Feature간 similarity)
- ✓ $S(\cdot,\cdot)$: pre-defined similarity measure
  - cross-correlation : $S(x,y) = x \times y$
  - d=1 : fully-connected layer

<u>Input Feature와 Filter간 similarity를 어떻게 정의할것인가?</u>

1. Review

1.2. Adder Networks

- similarity measure : L1-norm

$$Y(m,n,t) = -\sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}} |X(m+i,n+j,k) - F(i,j,k,t)|. \quad (2)$$

✓ Conv Filter의 출력은 양수 또는 음수

✓ Adder Filter의 출력은 항상 음수

=> <u>Batch Normalization을 사용하여 출력을 normalize</u>하여 기존의 활성화 함수를 사용할 수 있도록 함

(Batch Normalization엔 multiplication이 포함되지만 convolution에 비하면 무시할만 함)

✓ Computational cost

$O(d^2 c_{in} c_{out} HW) + O(c_{out} H'W')$ => ?? + $O(c_{out} H'W')$

1. Review

1.2. Adder Networks

- similarity measure : L1-norm

$$Y(m,n,t) = -\sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}} |X(m+i,n+j,k) - F(i,j,k,t)|. \tag{2}$$

```
def forward(ctx, W_col, X_col):
        ctx.save_for_backward(W_col, X_col)
        output = -(W_col.unsqueeze(2)-X_col.unsqueeze(0)).abs().sum(1)
        return output
```

1. Review

1.3. Optimization

- F에 대한 Y의 partial derivative
  - CNN

$$\frac{\partial Y(m,n,t)}{\partial F(i,j,k,t)} = X(m+i, n+j, k), \qquad (3)$$

  - AdderNet

$$\frac{\partial Y(m,n,t)}{\partial F(i,j,k,t)} = \mathrm{sgn}(X(m+i, n+j, k) - F(i,j,k,t)), \quad (4)$$

✓ (4)에서 gradient는 -1, 0, 1 값만 가지므로, signSGD*를 이용해 최적화를 해야함

✓ signSGD는 대부분의 경우 가장 가파른 기울기를 가지는 방향을 취하지 않으며 이는 차원이 커질수록 더 심해지므로 수많은 파라미터를 가진 네트워크를 학습하기 어려움

*signSGD : SGD 각각의 배치에 대해 gradient를 업데이트할 때 full precision의 gradient 대신 gradient의 부호(sign)를 활용하는 최적화 알고리즘

1. Review

1.3. Optimization

- Derivative of L2-norm

$$\frac{\partial Y(m,n,t)}{\partial F(i,j,k,t)} = X(m+i,n+j,k) - F(i,j,k,t), \quad (5)$$

✓ L2-norm의 derivative의 signSGD update는 Eq.(4)와 같음

✓ 따라서, Eq.(5)에 따라 <u>full-precision gradient를 활용하여 F와 X의 gradient를 정확하게 업데이트</u> 가능

1. Review

1.3. Optimization

- Full-precision gradient를 사용하면 gradient가 [-1, 1]의 범위를 넘어 gradient exploding이 발생 가능
  - ✓ 레이어 i의 필터와 입력을 F_i, X_i라 하면, F_i의 기울기에만 영향을 주는 $\frac{\partial Y}{\partial F_i}$와 달리 $\frac{\partial Y}{\partial X_i}$는 chain rule에 의해 i 이전 레이어의 기울기에도 영향을 주기 때문
  - ✓ 따라서, <u>X의 기울기에 대해 gradient clipping을 사용</u>

$$\frac{\partial Y(m,n,t)}{\partial X(m+i,n+j,k)} = \mathrm{HT}(F(i,j,k,t) - X(m+i,n+j,k)). \tag{6}$$

where $\mathrm{HT}(\cdot)$ denotes the HardTanh function:

$$\mathrm{HT}(x) = \begin{cases} x & \text{if} \quad -1 < x < 1, \\ 1 & \qquad\qquad x > 1, \\ -1 & \qquad\qquad x < -1. \end{cases} \tag{7}$$

1. Review

1.3. Optimization

- F에 대한 gradient

$$\frac{\partial Y(m,n,t)}{\partial F(i,j,k,t)} = X(m+i,n+j,k) - F(i,j,k,t), \quad (5)$$

- Adaptive Learning Rate

$$\Delta F_l = \gamma \times \alpha_l \times \Delta L(F_l), \quad (12)$$

$$\alpha_l = \frac{\eta\sqrt{k}}{\|\Delta L(F_l)\|_2}, \quad (13)$$

- X에 대한 gradient

$$\frac{\partial Y(m,n,t)}{\partial X(m+i,n+j,k)} = \mathrm{HT}(F(i,j,k,t) - X(m+i,n+j,k)). \quad (6)$$

where HT$(\cdot)$ denotes the HardTanh function:

$$\mathrm{HT}(x) = \begin{cases} x & \text{if} \quad -1 < x < 1, \\ 1 & \quad\quad x > 1, \\ -1 & \quad\quad x < -1. \end{cases} \quad (7)$$

```
def backward(ctx,grad_output):
        W_col, X_col = ctx.saved_tensors
        #ep5
        grad_W_col = ((X_col.unsqueeze(0)-W_col.unsqueeze(2))*grad_output.unsqueeze(1)).sum(2)
        #ep13
        grad_W_col = grad_W_col/grad_W_col.norm(p=2).clamp(min=1e-12)
                    *math.sqrt(W_col.size(1)*W_col.size(0))/5
        #eq6,7
        grad_X_col = (-(X_col.unsqueeze(0)-W_col.unsqueeze(2)).clamp(-1,1)*grad_output.unsqueeze(1)).sum(0)

        return grad_W_col, grad_X_col
```

2. Experiments

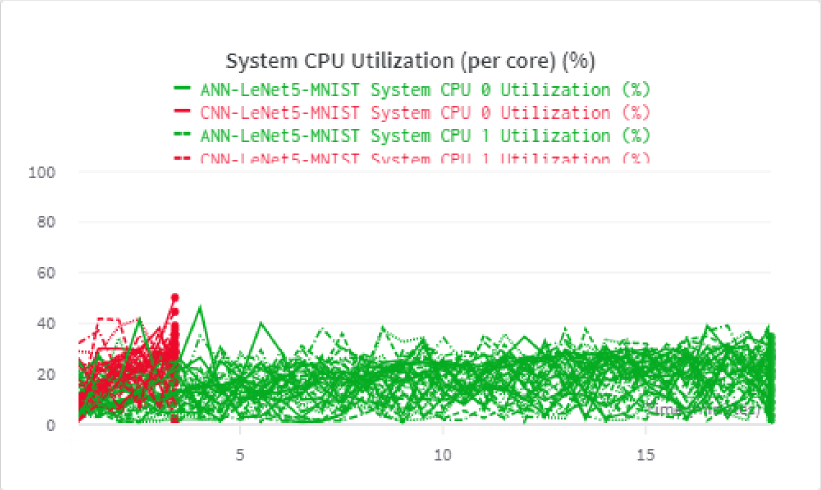
2.1. MNIST – Lenet5

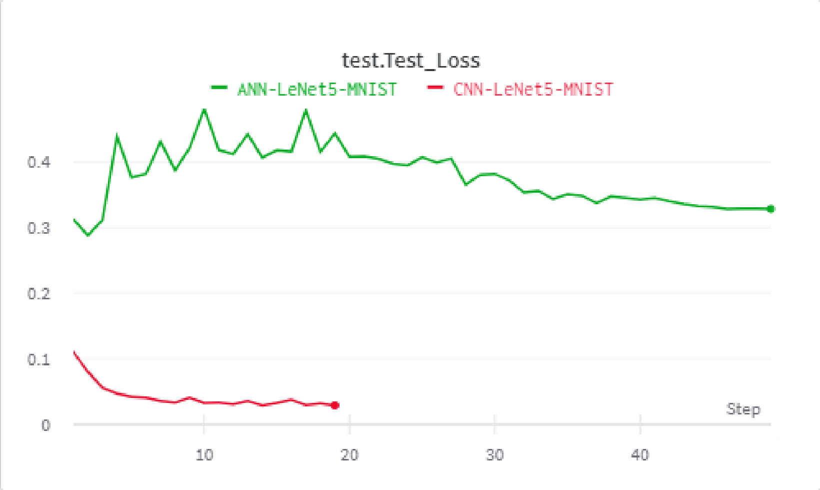
Method	#Add.	#Mul.	Accuracy	Latency
CNN	~435K	~435K	99.4%	~2.6M
AddNN	~870K	~0	99.4%	~1.7M

- ✓ Batch Nomalization Layer에 대한 multiplication cost는 다른 layer에 비해 작아 최종 FLOPs 계산에 포함 안함
- ✓ AdderNet이 CNN과 동일한 성능을 보였음

2. Experiments

2.1. MNIST – Lenet5





- ✓ Intel Xeon Silver & 2080ti
- ✓ GPU 학습 시간(per epoch)
- 14s / 6s
- ✓ CPU 학습 시간(per epoch)
- 3h 10m / 50s

2. Experiments

2.2. CIFAR10 - ResNet20

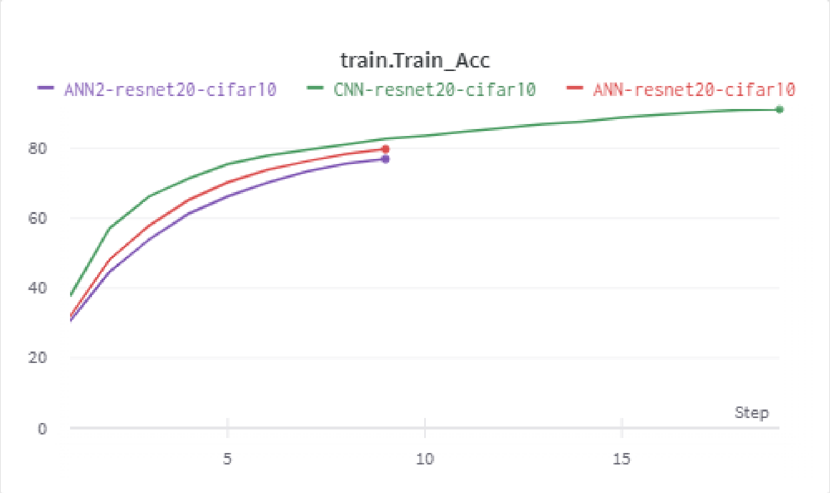
Table 2. Classification results on the CIFAR-10 and CIFAR-100 datasets.

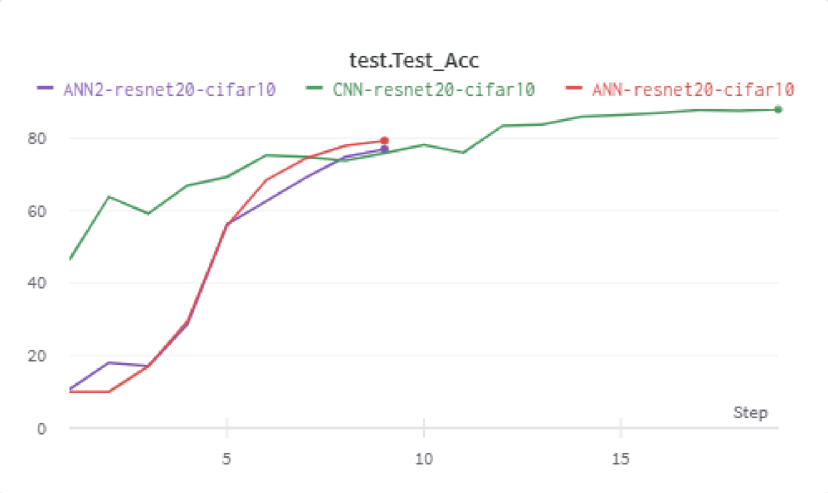
Model	Method	#Mul.	#Add.	XNOR	CIFAR-10	CIFAR-100
VGG-small	BNN	0	0.65G	0.65G	89.80%	65.41%
	AddNN	0	1.30G	0	93.72%	72.64%
	CNN	0.65G	0.65G	0	93.80%	72.73%
ResNet-20	BNN	0	41.17M	41.17M	84.87%	54.14%
	AddNN	0	82.34M	0	91.84%	67.60%
	CNN	41.17M	41.17M	0	92.25%	68.14%
ResNet-32	BNN	0	69.12M	69.12M	86.74%	56.21%
	AddNN	0	138.24M	0	93.01%	69.02%
	CNN	69.12M	69.12M	0	93.29%	69.74%

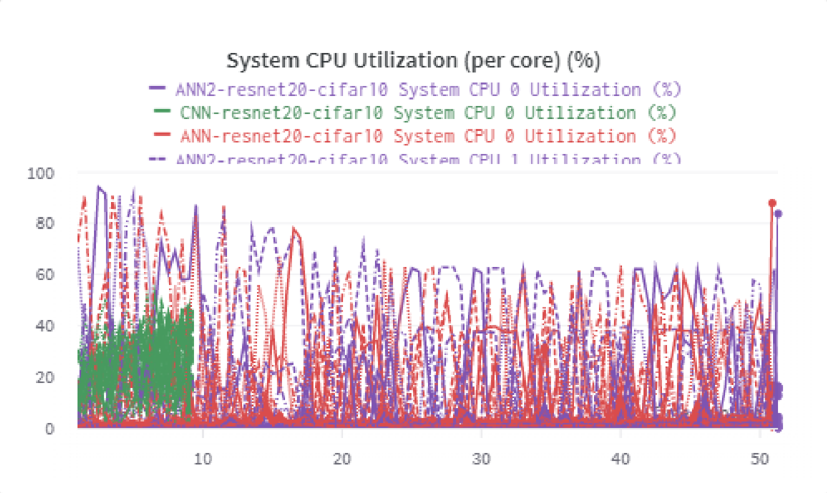
- ✓ First, last layer convolution은 유지
- ✓ Batch Nomalization Layer와 first, last layer의 convolution에 대한 multiplication cost는 다른 layer에 비해 작아 최종 FLOPs 계산에 포함 안함
- ✓ AdderNet이 CNN과 유사한 성능을 보였음

2. Experiments

2.2. CIFAR10 - ResNet20







- ✓ Intel Xeon Silver & 2080ti
- ✓ GPU 학습 시간(per epoch)

- 5m / 10s

- ✓ CPU 학습 시간(per epoch)

- 3h 10m / 50s

3. Latency & Memory

https://www.agner.org/optimize/instruction_tables.pdf

The convolutional neural network achieves a 99.4% accuracy with ~435K multiplications and ~435K additions. By replacing the multiplications in convolution with additions, the proposed AdderNet achieves a 99.4% accuracy, which is the same as that of CNNs, with ~870K additions and almost no multiplication.In fact, the theoretical latency of multiplications in CPUs is also larger than that of additions and subtractions. There is an instruction table [1] which lists the instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. For example, in VIA Nano 2000 series, the latency of float multiplication and addition is 4 and 2, respectively. The AdderNet using LeNet-5 model will have ~1.7M latency while CNN will have ~2.6M latency in this CPU. In conclusion, the AdderNet can achieve similar accuracy with CNN but have fewer computational cost and latency. Noted that CUDA and cuDNN optimized adder convolutions are not yet available, we do not compare the actual inference time.

✓ VIA Nano 2000

Floating point x87 instructions

	Operands	μops	Port and Unit	Latency
Arithmetic instructions				
FADD(P) FSUB(R)(P)	r/m	1	MB	2
FMUL(P)	r/m	1	MA	4

✓ AMD Zen4

Floating point x87 instructions

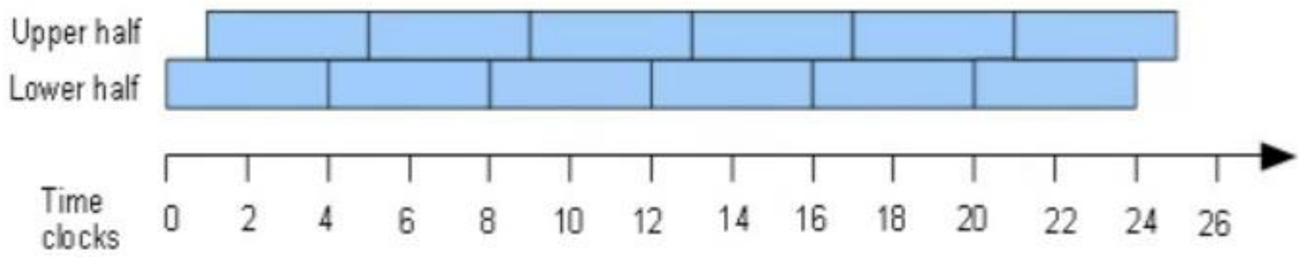
Instruction	Operands	Ops	Latency	Reciprocal throughput	Execution pipes
Move instructions					
Arithmetic instructions					
FADD(P),FSUB(R)(P)	r/m	1	7	2	P01
FIADD,FISUB(R)	m	2		1	P01 P23
FMUL(P)	r/m	1	7	2	P01
FIMUL	m	2		2	

3. Latency & Memory

Latency

The latency of an instruction is the delay that the instruction generates in a dependency chain. The measurement unit is clock cycles. Where the clock frequency is varied dynamically, the figures refer to the core clock frequency. The numbers listed are minimum values. Cache misses, misalignment, and exceptions may increase the clock counts considerably. Floating point operands are presumed to be normal numbers. Denormal numbers, NAN's and infinity may increase the latencies by possibly more than 100 clock cycles on many processors, except in move, shuffle and Boolean instructions. Floating point overflow, underflow, denormal or NAN results may give a similar delay. A missing value in the table means that the value has not been measured or that it cannot be measured in a meaningful way.

Some processors have a pipelined execution unit that is smaller than the largest register size so that different parts of the operand are calculated at different times. Assume, for example, that we have a long depencency chain of 128-bit vector instructions running in a fully pipelined 64-bit execution unit with a latency of 4. The lower 64 bits of each operation will be calculated at times 0, 4, 8, 12, 16, etc. And the upper 64 bits of each operation will be calculated at times 1, 5, 9, 13, 17, etc. as shown in the figure below. If we look at one 128-bit instruction in isolation, the latency will be 5. But if we look at a long chain of 128-bit instructions, the total latency will be 4 clock cycles per instruction plus one extra clock cycle in the end. The latency in this case is listed as 4 in the tables because this is the value it adds to a dependency chain.



3. Latency & Memory

✓ Intel Xeon Silver

✓ Inference

- model : CNN_resnet20
- input_shape : (256, 3, 32, 32)

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	CPU Mem
model_inference	5.90%	9.291ms	99.96%	157.437ms	157.437ms	-4 b
aten::conv2d	0.16%	253.000us	58.33%	91.867ms	4.835ms	184.00 Mb
aten::convolution	0.37%	585.000us	58.17%	91.614ms	4.822ms	184.00 Mb
aten::_convolution	0.24%	375.000us	57.80%	91.029ms	4.791ms	184.00 Mb
aten::mkldnn_convolution	57.24%	90.148ms	57.56%	90.654ms	4.771ms	184.00 Mb
aten::batch_norm	0.13%	199.000us	24.21%	38.125ms	2.007ms	184.01 Mb
aten::_batch_norm_impl_index	0.23%	369.000us	24.08%	37.926ms	1.996ms	184.01 Mb
aten::native_batch_norm	17.76%	27.967ms	23.80%	37.481ms	1.973ms	184.01 Mb
aten::clamp_min	7.45%	11.736ms	14.77%	23.258ms	612.053us	368.00 Mb
aten::relu	0.30%	472.000us	7.79%	12.273ms	645.947us	184.00 Mb
aten::mean	0.38%	604.000us	5.32%	8.385ms	441.316us	2.69 Kb
aten::sum	3.79%	5.964ms	3.88%	6.113ms	321.737us	0 b
aten::add_	2.63%	4.138ms	2.63%	4.138ms	147.786us	0 b
aten::div_	0.58%	918.000us	1.06%	1.668ms	87.789us	0 b
aten::empty	1.02%	1.612ms	1.02%	1.612ms	9.211us	380.01 Mb
aten::constant_pad_nd	0.05%	80.000us	0.75%	1.175ms	587.500us	12.00 Mb
aten::copy_	0.57%	893.000us	0.57%	893.000us	40.591us	0 b
aten::to	0.08%	129.000us	0.48%	750.000us	39.474us	76 b
aten::_to_copy	0.17%	273.000us	0.39%	621.000us	32.684us	76 b
aten::empty_like	0.09%	146.000us	0.32%	497.000us	26.158us	184.00 Mb
aten::fill_	0.23%	367.000us	0.23%	367.000us	17.476us	0 b
aten::linear	0.01%	19.000us	0.18%	280.000us	280.000us	10.00 Kb
aten::addmm	0.14%	216.000us	0.15%	237.000us	237.000us	10.00 Kb
aten::empty_strided	0.13%	203.000us	0.13%	203.000us	10.684us	76 b
aten::avg_pool2d	0.10%	162.000us	0.10%	162.000us	162.000us	64.00 Kb
aten::slice	0.06%	100.000us	0.08%	119.000us	9.917us	0 b
aten::as_strided	0.06%	101.000us	0.06%	101.000us	3.061us	0 b
aten::as_strided_	0.05%	85.000us	0.05%	85.000us	4.474us	0 b
aten::zeros	0.02%	33.000us	0.04%	66.000us	66.000us	4 b
aten::narrow	0.01%	17.000us	0.03%	41.000us	10.250us	0 b
aten::t	0.01%	14.000us	0.02%	24.000us	24.000us	0 b
aten::view	0.01%	18.000us	0.01%	18.000us	18.000us	0 b
aten::transpose	0.00%	7.000us	0.01%	10.000us	10.000us	0 b
aten::expand	0.00%	6.000us	0.00%	7.000us	7.000us	0 b
aten::zero_	0.00%	2.000us	0.00%	2.000us	2.000us	0 b
aten::resolve_conj	0.00%	1.000us	0.00%	1.000us	0.500us	0 b

Self CPU time total: 157.503ms

3. Latency & Memory

- ✓ Intel Xeon Silver

- ✓ Inference
  - model : ANN_resnet20
  - input_shape : (256, 3, 32, 32)

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	CPU Mem
model_inference	1.94%	282.315ms	100.00%	14.523s	14.523s	-4 b
adder	41.60%	6.041s	87.70%	12.737s	636.860ms	180.00 Mb
aten::abs	18.18%	2.640s	36.35%	5.279s	131.976ms	77.00 Gb
aten::sub	19.86%	2.884s	19.86%	2.884s	144.181ms	38.50 Gb
aten::sum	7.47%	1.084s	7.80%	1.133s	26.980ms	180.00 Mb
aten::im2col	5.24%	760.765ms	7.19%	1.045s	52.229ms	1.45 Gb
aten::contiguous	0.00%	315.000us	2.32%	336.634ms	8.416ms	1.61 Gb
aten::clone	0.01%	1.172ms	2.32%	336.319ms	8.408ms	1.61 Gb
aten::copy_	2.30%	333.711ms	2.30%	333.711ms	5.382ms	0 b
aten::resize_	1.16%	168.729ms	1.16%	168.729ms	8.436ms	1.25 Gb
aten::fill_	1.12%	162.074ms	1.12%	162.074ms	2.614ms	0 b
aten::zero_	0.00%	237.000us	0.78%	113.698ms	4.943ms	0 b
aten::batch_norm	0.00%	239.000us	0.61%	89.097ms	4.050ms	196.02 Mb
aten::_batch_norm_impl_index	0.00%	465.000us	0.61%	88.858ms	4.039ms	196.02 Mb
aten::native_batch_norm	0.52%	75.703ms	0.61%	88.339ms	4.015ms	196.02 Mb
aten::neg	0.32%	46.699ms	0.32%	46.699ms	2.335ms	180.00 Mb
aten::conv2d	0.00%	27.000us	0.09%	13.307ms	6.654ms	16.01 Mb
aten::convolution	0.00%	83.000us	0.09%	13.280ms	6.640ms	16.01 Mb
aten::_convolution	0.00%	43.000us	0.09%	13.197ms	6.598ms	16.01 Mb
aten::mkldnn_convolution	0.09%	13.102ms	0.09%	13.154ms	6.577ms	16.01 Mb
aten::add_	0.09%	13.032ms	0.09%	13.032ms	420.387us	0 b
aten::mean	0.00%	713.000us	0.08%	10.979ms	499.045us	3.10 Kb
aten::relu_	0.00%	498.000us	0.03%	3.752ms	197.474us	0 b
aten::empty	0.02%	3.528ms	0.02%	3.528ms	14.762us	2.01 Gb
aten::clamp_min_	0.00%	182.000us	0.02%	3.254ms	171.263us	0 b
aten::clamp_min	0.02%	3.072ms	0.02%	3.072ms	161.684us	0 b
aten::empty_like	0.01%	780.000us	0.02%	2.976ms	36.293us	2.00 Gb
aten::div_	0.01%	1.264ms	0.02%	2.244ms	102.000us	0 b
aten::view	0.01%	1.997ms	0.01%	1.997ms	19.772us	0 b
aten::to	0.00%	162.000us	0.01%	980.000us	44.545us	88 b
aten::_to_copy	0.00%	413.000us	0.01%	818.000us	37.182us	88 b
aten::permute	0.00%	572.000us	0.00%	675.000us	16.875us	0 b
aten::select	0.00%	492.000us	0.00%	624.000us	15.600us	0 b
aten::avg_pool2d	0.00%	531.000us	0.00%	531.000us	531.000us	64.00 Kb
aten::as_strided	0.00%	453.000us	0.00%	453.000us	2.796us	0 b
aten::unsqueeze	0.00%	307.000us	0.00%	370.000us	9.250us	0 b
aten::empty_strided	0.00%	271.000us	0.00%	271.000us	12.318us	88 b
aten::zeros	0.00%	37.000us	0.00%	69.000us	69.000us	4 b
aten::as_strided_	0.00%	8.000us	0.00%	8.000us	4.000us	0 b

Self CPU time total: 14.523s

5. Conclusions

- LeNet5에서는 논문과 똑같이 구현했음에도 90.01%의 성능을 보여 논문에 나온 99.4%의 성능과 큰 차이가 남
- ResNet20의 경우 논문에서 기재한 400 epoch을 돌리기에 많은 시간이 소요되어 간단히 돌려본 결과 CNN의 성능과 유사하게 학습되는 양상을 확인
- ResNet20에서 첫번째, 마지막 convolution layer를 adder layer로 교체하니 성능이 낮아짐
- CUDA 최적화가 안되어 CIFAR10-ResNet20을 GPU 학습 시 메모리 x4이상, 학습시간 x30이상
- CPU Inference 시에도 CNN보다 ANN이 훨씬 느렸음
- 실제 모바일 기기에서 추론했을 때, addition 연산의 효과가 나타날지 의문