**WICWIU**

**(What I Create, What I Understand!)**

상세 설명서

한동대학교 딥러닝 연구실

**목 차**

[1. 서론 1](#_Toc531196513)

[1.1. 필요성 및 설계 목표 1](#_Toc531196514)

[1.1.1. 필요성 1](#_Toc531196515)

[1.1.2. 설계 목표 2](#_Toc531196516)

[1.2. 본 딥러닝 프레임워크의 특징 3](#_Toc531196517)

[1.2.1. 네트워크 자동 미분 기능 3](#_Toc531196518)

[1.2.2. GUI 기반 신경망 모델 디자인 툴 3](#_Toc531196519)

[2. 시스템 요구사항 4](#_Toc531196520)

[2.1 시스템 요구 사항 4](#_Toc531196521)

[2.1.1. 지원 플랫폼 및 운영체제 4](#_Toc531196522)

[2.1.2. H/W 요구사항 4](#_Toc531196523)

[2.1.3 S/W 요구사항 4](#_Toc531196524)

[3. 라이브러리 구성 5](#_Toc531196525)

[3.1. 라이브러리 폴더 구성 5](#_Toc531196526)

[3.2. 라이브러리 구조 6](#_Toc531196527)

[3.2.1 신경망 구조 정의 및 학습 진행 방식 6](#_Toc531196528)

[3.3 컴파일 (정적 라이브러리 생성) 15](#_Toc531196529)

[3.3.1. make 파일 설명 15](#_Toc531196530)

[4. 라이브러리 사용 24](#_Toc531196531)

[4.1. 튜토리얼 코드 설명 24](#_Toc531196532)

[4.1.1. MNIST(CNN) 24](#_Toc531196533)

[4.1.2. ImageNet(ResNet) 29](#_Toc531196534)

[4.1.3. ImageNet(DenseNet) 37](#_Toc531196535)

4.1.4 GAN(Vanilla GAN)…………………………………………………………………44

# 1. 서론

## 1.1. 필요성 및 설계 목표

### 1.1.1. 필요성

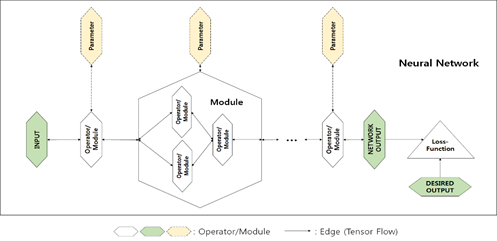
최근 딥러닝 연구가 세계적으로 빠르게 발전할 뿐만 아니라 사회 전반적으로 큰 이슈화가 되고 있고, 산업계 구조에서도 인공지능의 도입이 활발히 진행중이다. 하지만 국내에서 연구 중이거나 산업체에 활용하는 딥러닝 프레임워크는 대부분 해외에서 개발되었으며, 프레임워크에 대한 의존성 또한 매우 높다. 그 이유는 딥러닝 프레임워크 자체가 학습 및 테스트를 수행하기 위한 엔진을 개발해야 하기 때문이며, 구조적으로나 내용적으로 복잡해서 국내에서 개발한 경우는 매우 희소하다. 그 결과 국내 딥러닝 사용자들은 다음과 같은 불편함을 겪는다.

첫째, 영어로 작성된 설명서로 인해 국내 사용자들은 프레임워크를 이해하는데 불편함이 있다.

둘째, 파이썬 환경의 해외 프레임워크는 자체 수정 및 확장에 어려움을 가지며 다양한 환경에서의 이식이 쉽지 않다.

마지막으로, 해외 기술의 의존성이 높아진다.

본 딥러닝 프레임워크는 이러한 문제를 극복하기 위해 국내 기술로 개발된 것으로, 알고리즘의 확장성과 가독성을 고려하여, 일관성 높은 C++코드와 API로 구성되며, 한글 설명서를 제공하여 국내 사용자들에게 편의를 제공하고자 한다.

**< 본 프레임워크로 구성한 신경망 모델 >**

### 1.1.2. 설계 목표

본 프레임워크는 국내 딥러닝 사용자들이 쉽게 이해하고 수정 및 확장이 용이한 프레임워크를 목표로 설계되었다. 목표를 달성하기 위한 본 프레임워크의 특징은 다음과 같다.

첫째, 국내 개발자들에게 친숙한 C++을 중심으로 개발되었다. C++ 환경은 메모리 및 성능 최적화에 용이하며 다양한 환경에 이식하기도 용이하다.

둘째, 확장성이 높아 사용자가 새로운 신경망 구조나 알고리즘을 구현하기 용이하다. 뉴럴 네트워크를 구성하는 연산자, 최적화 함수, 손실 함수는 모두 클래스 구조로 설계하여 사용자가 딥러닝 알고리즘의 확장/수정을 용이하게 하였다.

셋째, 본 프레임워크의 코드는 문서와 주석 등을 한글로 제공하여 국내 개발자가 이해하기 쉽다.

마지막으로, GPU를 이용한 대규모 병렬 계산을 통해 신경망을 빠른 속도로 학습할 수 있다. GPU 연산은 cuDNN과 cuda를 이용해 구현하였다. 또한, 호스트 메모리와 GPU 메모리 간 동기화를 자동으로 수행해 개발자 편의성이 우수하다.

## 1.2. 본 딥러닝 프레임워크의 특징

### 1.2.1. 네트워크 자동 미분 기능

딥러닝을 사용하여 네트워크를 학습시키기 위해서는 Loss에 대한 경사도 벡터를 구하여야 하며, 경사하강법이라는 알고리즘을 사용하여서 네트워크를 최적화 하는 과정을 거치게 된다.

본 프레임워크는 사용자가 구성한 네트워크에 대해서 경사도 벡터를 손쉽게 구할 수 있도록, 각 노드마다 BackPropagation( )이라고 하는 메서드를 제공하고 있으며, 이를 사용한 미분 결과와 네트워크 출력 부분에서 흘러 들어오는 미분 값을 사용한 체인 룰을 통해서 전체 미분 값을 구하는 것이 가능하다.

### 1.2.2. GUI 기반 신경망 모델 디자인 툴

본 프레임워크는 사용자가 딥러닝에서 사용되는 구성 요소(Operator, Layer, Loss Function, Optimizer 등)를 조합하여 신경망(Computational NeuralNetwork)에 대한 코드를 생성해주는 신경망 모델 디자인 툴을 제공한다.

본 인터페이스의 기능은 다음과 같다.

* 블록 사이의 링크 연결 및 링크 추가, 링크 제거
* 블록 툴바에서 마우스 드래그를 통한 블록 생성
* 블록 선택 시 블록에 대한 정보 출력
* 블록 연결 상태 분석 (BFS 알고리즘 사용)
* 블록 연결 상태 및 블록 종류에 따라 코드 생성

# 2. 시스템 요구사항

## 2.1 시스템 요구 사항

### 2.1.1. 지원 플랫폼 및 운영체제

* 본 프레임 워크는 Ubuntu 16.04 환경에서 설계되었고 그 이상 버전에서 사용하는 것을 권장.

### 2.1.2. H/W 요구사항

* 본 프레임 워크는 Intel i5 이상의 CPU를 사용하는 환경에서 사용하기를 권장.
* 본 프레임 워크는 CUDA 9.0, cudnn 7.0.5을 지원하는 NVidia GPU 사용을 권장.

### 2.1.3. S/W 요구사항

* C++언어 C++11 이상, gcc 5.4.0 이상을 권장.
* NVIDIA GPU 사용 시   
  - NVIDIA Grahpic Driver  
  - CUDA (tested on v9.0 / 10.0)  
  - cudnn (tested on v7.0.5 / 7.4.1)설치를 요구함.
* NVIDIA GPU 사용시 필요한 S/W 설치 방법은 첨부함.

# 3. 라이브러리 구성

## 3.1. 라이브러리 폴더 구성

본 프레임워크의 라이브러리 폴더는 크게 lib 폴더, tutorials 폴더, src 폴더로 구성되어 있다.

* lib

: 컴파일 후 정적 라이브러리가 생성되는 디렉토리.

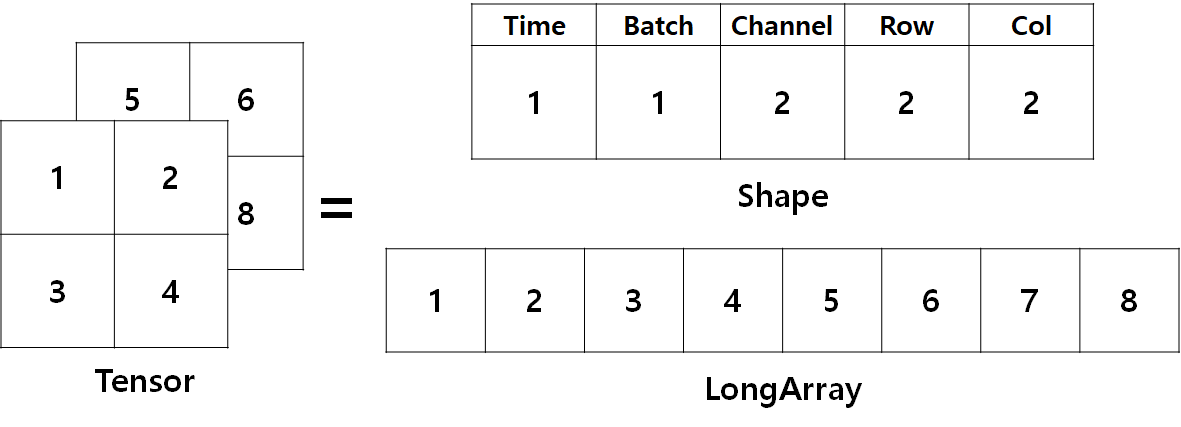
* tutorials
  + CIFAR10
  + CIFAR100Reader
  + ImageNet : 포함된 파일을 이용하여 사전에 정의된 인공신경망을 학습 가능.
    - net : 사전에 정의된 인공신경망을 포함하는 디렉토리.
  + MNIST : 포함된 파일을 이용하여 사전에 정의된 인공신경망을 학습 가능.
    - net : 사전에 정의된 인공신경망을 포함하는 디렉토리.
    - data : MNIST 데이터가 포함된 디렉토리.
  + GAN : 하부 디렉토리 내 포함된 파일을 이용하여 사전에 정의된 인공신경망을 학습 가능
    - data : MNIST 데이터가 포함된 디렉토리.
    - VanillaGAN
      * net : 사전에 정의된 인공신경망을 포함하는 디렉토리
    - DCGAN
      * net : 사전에 정의된 인공신경망을 포함하는 디렉토리
    - WGAN
      * net : 사전에 정의된 인공신경망을 포함하는 디렉토리
    - BEGAN
      * net : 사전에 정의된 인공신경망을 포함하는 디렉토리
* src : LossFunction, Operator, Optimizer, 그리고 Module를 비롯한 소스파일이 있는 디렉토리.
  + LossFunction
  + Operator
  + Optimizer
  + Module

## 3.2. 라이브러리 구조

### 3.2.1 신경망 구조 정의 및 학습 진행 방식

**3.2.1.1. 신경망 구조 정의**

**1. 데이터**



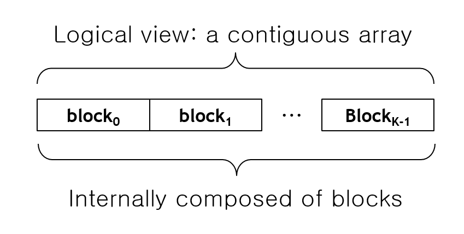
**< Tensor, Shape, LongArray 데이터 형식 요약 >**

**1) Tensor 클래스**

텐서는 물리학의 개념을 차용, 다차원의 정보를 갖는 데이터를 의미한다. 텐서는 본 프레임워크에서 인공 신경망 모델의 학습에 이용되는 데이터의 기본 형식이다. Tensor 클래스는 최대 5의 계수(Rank)를 갖는 텐서를 지원한다. 텐서의 각 축은 인공 신경망 학습에서 Time, Batch, Row, Column 축에 해당한다. Tensor 클래스는 텐서의 차원에 관한 정보를 담고 있는 Shape 클래스와 각 차원의 개별 데이터를 담고 있는 LongArray 클래스를 멤버 변수로 갖는다. Tensor 클래스는 Shape 클래스의 차원 정보 데이터를 활용해 각 차원의 개별 데이터를 담고 있는 메모리에 접근한다.

**2) Shape 클래스**

Shape 클래스는 텐서 데이터의 차원에 관한 정보를 담고 있는 클래스이다. Shape 클래스는 텐서의 계수(Rank)를 담고 있는 변수와 계수 각 축의 차원 정보를 담고 있는 변수를 멤버 변수로 갖고 있다. 계수 변수의 최댓값은 5이고, 각 축은 인공 신경망 학습에서 Time, Batch, Channel, Row, Column 축으로 활용된다. 각 축의 차원 정보에 해당하는 변수는 포인터로 선언되어 있으며, 축의 개수만큼 배열로서 동적으로 할당되어 축의 차원 정보를 나타낸다.

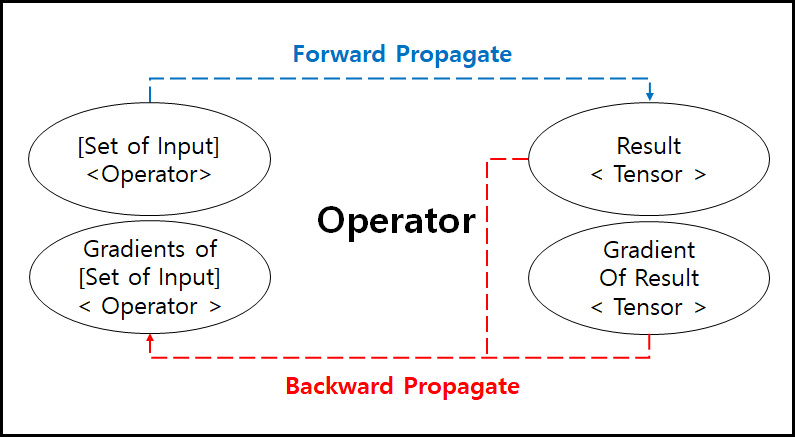
**3) LongArray 클래스**

**< LongArray 클래스의 위계적 데이터 구조 >**

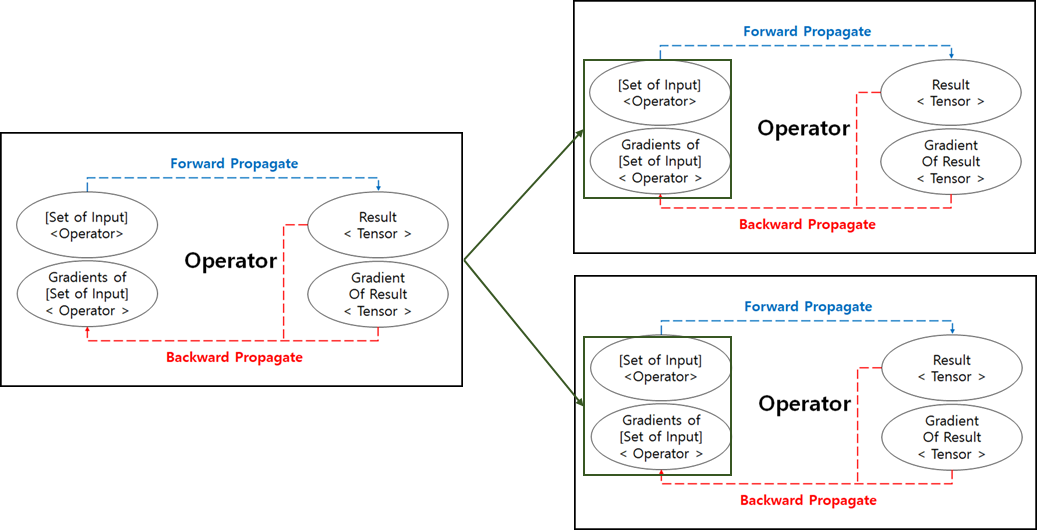
LongArray 클래스는 High-dimensional Array의 약어로, 실제 각 차원의 개별 데이터를 담고 있는 클래스이다. LongArray 클래스의 데이터들은 동적으로 한 번에 할당했을 때 메모리가 부족해 할당이 불가하는 경우를 막기 위해 위계적 구조를 가지고 이중 포인터를 이용하여 선언되어 있다. 논리적으로는 하나의 연속적인 배열이나 물리적으로는 블록으로 나누어져 메모리에 할당된다. 각 블록은 Time 축을 기준으로 그 외 나머지 4개의 차원(Batch, Channel, Row, Column)을 갖는 데이터로 분리된다.

**2. 모델 빌딩 블록**

**1) Operator 클래스**



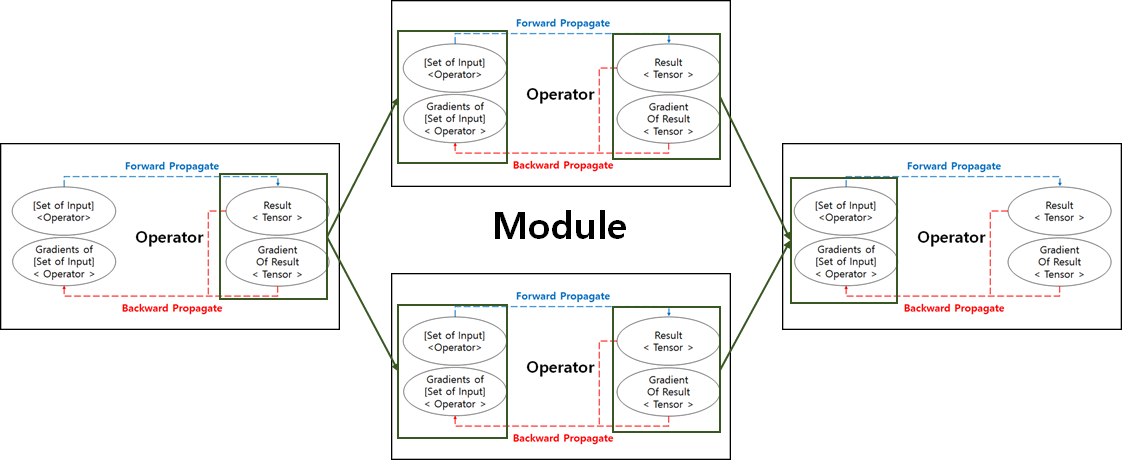
**< Operator 클래스의 기본 구조 >**



**< Operator 클래스 간 네트워크 연결 >**

Operator 클래스는 인공 신경망 안에서 저 수준의 기본 연산을 수행하기 위한 클래스이다. Operator 클래스는 인공 신경망의 노드를 구성하는 가장 작은 단위이다. Operator 클래스는 네트워크 구성을 위해 각각 연산의 입력 방향과 출력 방향에 대해서 Operator 클래스에 대한 포인터를 저장하는 Container(본 프레임워크에서 정의한 큐와 비슷한 자료 구조, 이하 컨테이너로 표기)를 멤버 변수로 갖는다. 또한 인공 신경망의 순전파와 역전파를 수행하기 위해 연산의 결과 값과 경사도 벡터를 저장하기 위한 Tensor 클래스에 대한 포인터를 저장하는 Container를 멤버 변수로 갖는다. 순전파 및 역전파에서 사용되는 Weight 파라미터들은 후에 기술할 NeuralNetwork 클래스가 포인터들의 Container로 가지고 있으며, Operator 클래스에게 전달한다. 파생 클래스들은 해당 클래스를 상속하여 각 연산의 기능을 수행한다.

**2) Module 클래스**

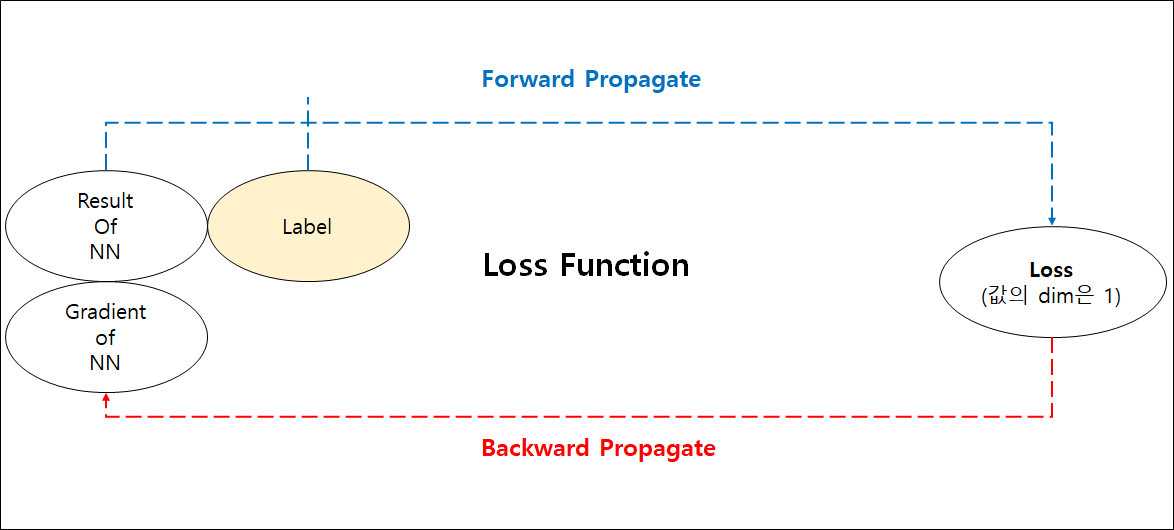


**< Module 클래스의 기본 구조 예시 >**

Module 클래스는 단일 Operator 클래스로는 수행할 수 없는 고수준의 복합 연산을 수행하기 위한 클래스이다. Module 클래스는 Operator 클래스를 상속 받으며, 복수의 Operator 들을 조합하여 서브 그래프를 구성하고 연산을 수행한다. Module 클래스는 너비 우선 탐색(Breadth-First Search) 방식을 이용해 서브 그래프를 구성하고 연산한다. Module 클래스는 다른 Module 클래스를 포함하는 재귀적 구조를 형성할 수 있다. Module 클래스는 하나의 단일 Operator 클래스와 같이 기능 및 동작한다. 파생 클래스들은 해당 클래스를 상속하여 각 연산의 기능을 수행한다.

**3. 학습 요소**

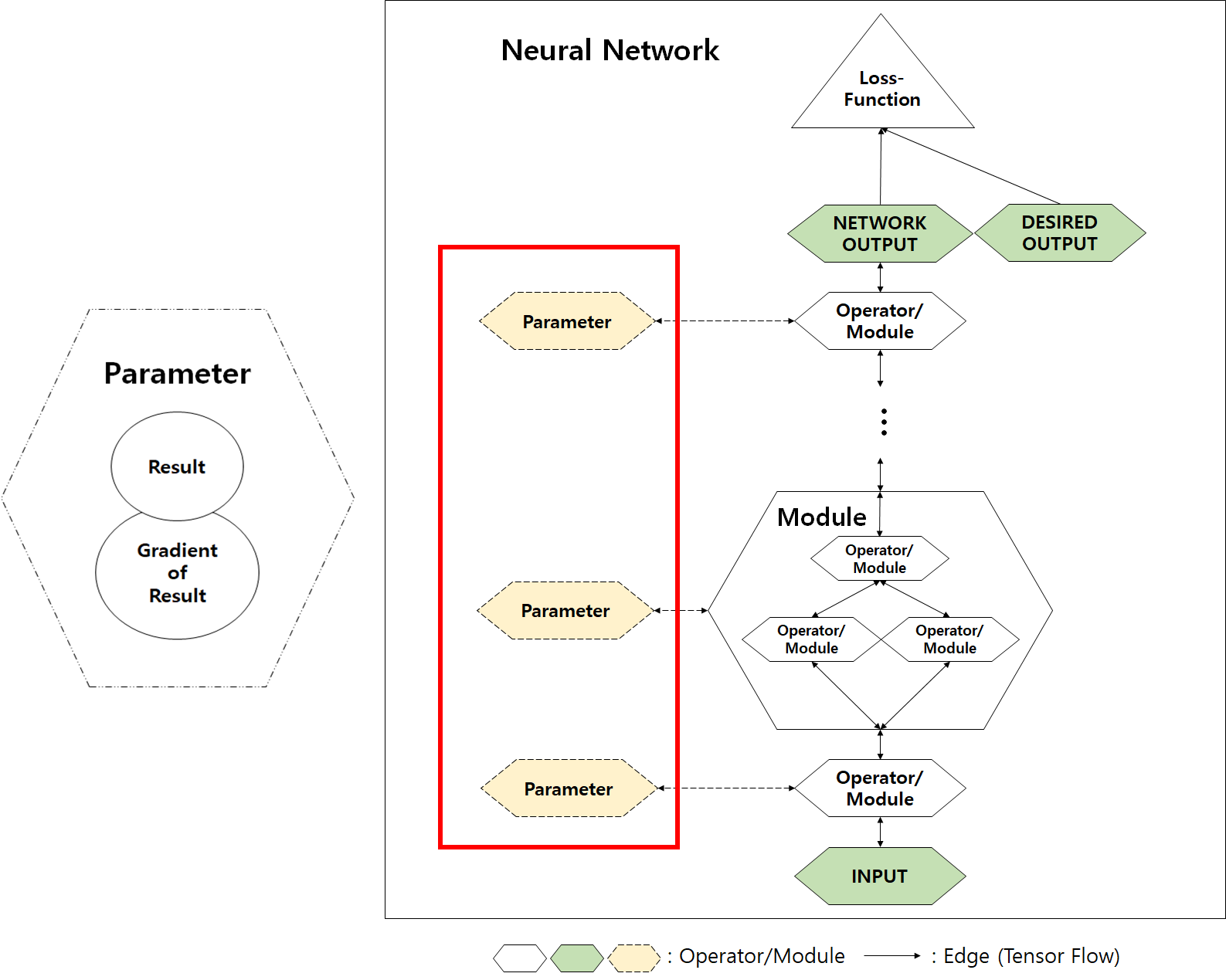
**1) LossFunction 클래스**



**< LossFunction 클래스의 기본 구조 >**

LossFunction 클래스는 NeuralNetwork의 순전파를 통해 얻어진 최종 출력 Tensor와 NeuralNetwork로부터 전달 받은 입력 데이터에 대한 레이블 값을 비교하여 손실 함수 값을 계산하는 클래스이다. Operator 클래스와는 구분된 독립적인 클래스이며, 경사도 벡터 값을 계산하여 NeuralNetwork의 역전파의 시작 부분을 담당한다. 파생 클래스들은 해당 클래스를 상속하여 각 손실 함수의 기능을 수행한다.

**2) Optimizer 클래스**

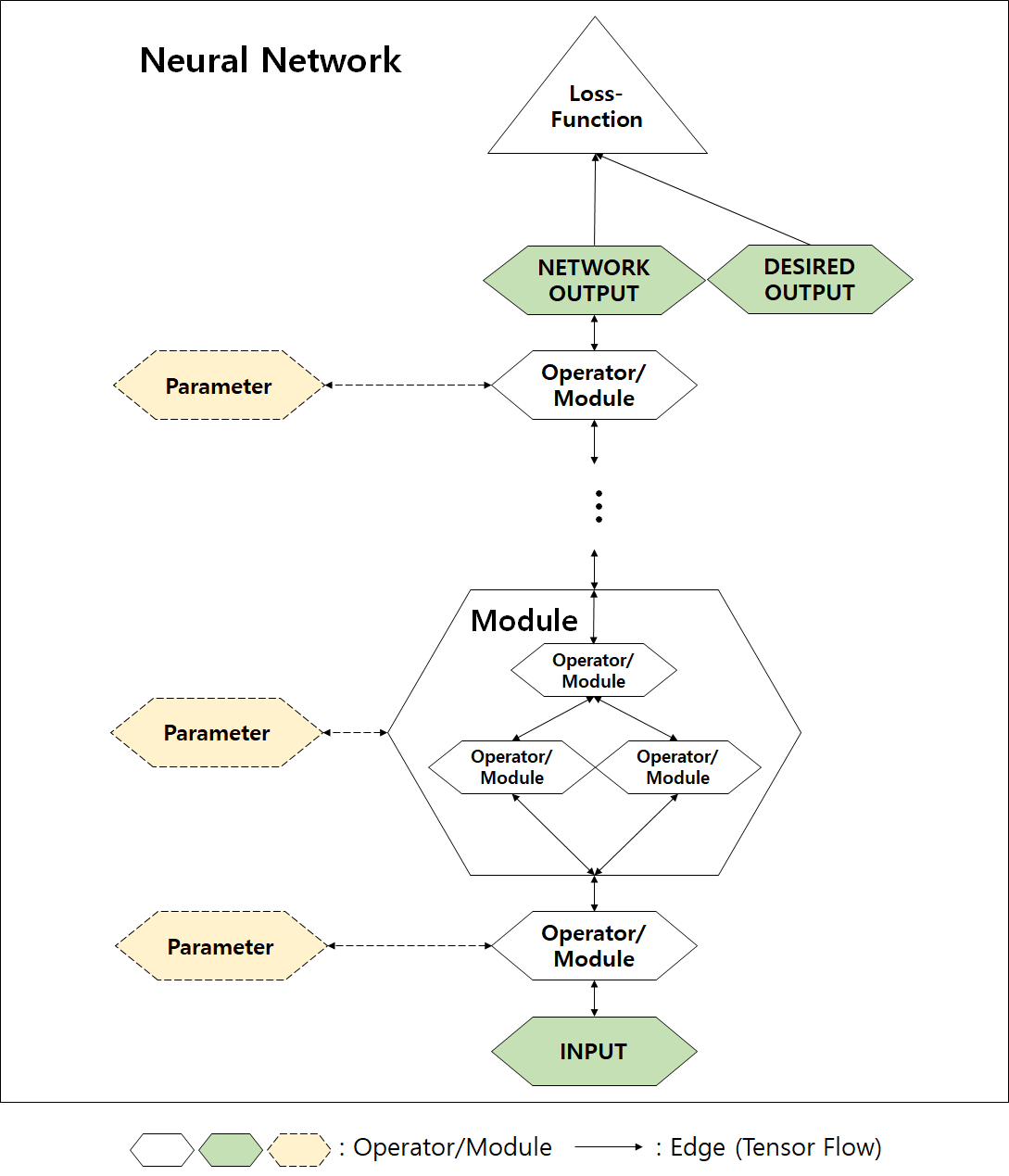


**< NeuralNetwork 클래스 내 학습 파라미터의 기본 구조 >**

Optimizer 클래스는 최적화 알고리즘을 활용해 학습 파라미터를 업데이트하는 클래스이다. 인공 신경망 모델의 역전파 시, 손실 함수에서 계산된 값에 대해서 각 학습 파라미터의 경사도 벡터가 Tensor 형태로 계산된다. Optimizer 클래스는 이 경사도 벡터에 대한 Tensor들의 포인터를 컨테이너로 가지고 있다. Optimizer 클래스는 해당 최적화 알고리즘에 따라서 경사도 벡터를 활용하여 학습 파라미터를 업데이트하는 기능을 수행한다. 파생 클래스들은 해당 클래스를 상속하여 각 Optimizer의 기능을 수행한다.

**4. 연산 그래프**

**1) NeuralNetwork 클래스**



**< NeuralNetwork 클래스 그래프 예시 >**

NeuralNetwork 클래스는 앞에서 설명했던 요소들을 조합하여, 학습 가능한 신경망 모델을 구성하는 클래스이다. NeuralNetwork 클래스는 하나의 그래프를 구성하는 Operator(Module) 클래스 포인터에 대한 컨테이너를 가지고 있다. Operator(Module) 클래스들이 구성하는 그래프는 인공 신경망 모델에 대응되며, NeuralNetwork 클래스는 너비 우선 탐색(Breadth-first Search) 방식으로 각 Operator 클래스들에 접근한다. NeuralNetwork 클래스는 LossFunction 클래스를 이용해 손실 함수의 값을 계산하고, Optimizer 클래스의 최적화 알고리즘을 이용해 인공 신경망을 최적화한다. NeuralNetwork 클래스는 학습 파라미터에 대한 포인터의 컨테이너를 가지고 있으며, 역전파 및 최적화 시 활용한다. NeuralNetwork 클래스를 이용한 학습 진행 방식의 자세한 내용은 후술하였다.

**3.2.1.1. 라이브러리를 사용한 학습 진행 방식**

**1) 외부 데이터 적용**

1. 우선 신경망(Computational NeuralNetwork)을 학습시키기 위해 준비된 학습데이터를 먼저 Tensor로 변환한다. (tutorals/<DATA\_NAME>/\*Reader.hpp 파일 참고)
2. 그 후, FeedInput(...) 메서드를 사용하여 변환된 데이터를 신경망에 적용한다. (이 인풋의 크기는 신경망에서 기대하는 기대값 텐서 크기와 일치해야 하며, 일치하지 않을 경우 학습이 올바로 진행되지 않을 수 있다.)
3. 위의 작업이 완료되면 신경망의 입력 공간에는 위의 Tensor (학습 데이터)가 등록되어 사용된다.

**2) 순전파 과정**

1. 데이터가 적용된 신경망은 우선 이전의 값이 등록되어 있을 가능성이 존재하는 구간을 모두 0으로 초기화한다. 대표적으로는 연산에 참여하는 Operator 및 LossFunction의 결과 Tensor와 경사도벡터 Tensor가 있다.
2. 위의 초기화 작업이 완료되면, 앞서 분석한 그래프의 Operator의 순전파 연산을 순서대로 실행한다. (본 프레임워크의 신경망은 사용자가 추가한 Operator의 연결상태를 BFS로 분석하며, 이를 기준으로 신경망 연산을 진행한다)
3. 모든 Operator의 연산이 끝나면, 마지막으로 LossFunction의 순전파를 실행하여 Loss를 구한다.

**3) 역전파 과정**

1. Loss가 구해진 신경망은, LossFunction의 역전파 연산을 시작으로 각 Operator의 순서를 거꾸로 가면서 Loss에 대한 경사도 벡터를 각각 구해 나간다.
2. 역전파 연산을 모두 마무리하면, 신경망에 적용된 파라미터들은 Loss에 대한 경사도 벡터를 가지게 된다.

**4) 파라미터 업데이트 과정**

1. 각 파라미터는 신경망에 적용된 Optimizer에서 지정된 알고리즘을 따라 최적화된다. 이 때 사용하는 값은 앞서 역전파 연산으로 구해진 경사도 벡터이며, 이를 통해 신경망은 Loss가 작아지는 방향으로 학습이 진행된다.

CF) 아래에서 설명하는 tutorial 코드 가운데에서, main.cpp 코드를 보면 위의 과정을 이해하는 것에 도움이 될 것이다.

## 3.3 컴파일 (정적 라이브러리 생성)

### 3.3.1. make 파일 설명

**1) src (라이브러리 폴더)**

**(1) 전체 파일**

================================================================

.SUFFIXES = .cpp .o

FRAMEWORK\_LIB = lib/library.a

CFLAGS = -O2 -std=c++11

ENABLE\_CUDNN = -D\_\_CUDNN\_\_

DFLAGS = -g -D\_\_DEBUG\_\_

INCLUDE\_PATH = -I/usr/local/cuda/include

LIB\_PATH = -L. -L/usr/local/cuda/lib64

CC = g++

NVCC = nvcc

ifdef ENABLE\_CUDNN

LINKER = nvcc

LFLAGS = -lcudart -lcudnn -lpthread

else

LINKER = g++

LFLAGS = -lpthread

endif

AR = ar

SRCS = \

src/Utils.cpp \

src/Shape.cpp \

OBJS = ${SRCS:.cpp=.o}

ifdef ENABLE\_CUDNN

CUDA\_SRCS = \

src/Utils\_CUDA.cu \

src/Optimizer/AdamOptimizer\_CUDA.cu \

src/Operator/Concatenate\_CUDA.cu

CUDA\_OBJS = ${CUDA\_SRCS:.cu=.o}

endif

all: $(FRAMEWORK\_LIB)

.cpp.o:

$(CC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) $(LIB\_PATH) -c $< -o $@

src/Utils\_CUDA.o: src/Utils\_CUDA.cu

$(NVCC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) -c $< -o $@

src/Optimizer/AdamOptimizer\_CUDA.o: src/Optimizer/AdamOptimizer\_CUDA.cu

$(NVCC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) -c $< -o $@

src/Operator/Concatenate\_CUDA.o: src/Operator/Concatenate\_CUDA.cu

$(NVCC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) -c $< -o $@

$(FRAMEWORK\_LIB): $(OBJS) $(CUDA\_OBJS)

$(AR) rcs $@ $(OBJS) $(CUDA\_OBJS)

clean:

rm -rf \*.o $(OBJS) $(CUDA\_OBJS) $(FRAMEWORK\_LIB)

================================================================

**(2) 파일 설명**

.SUFFIXES = .cpp .o

: Suffix, CPP 파일을 Object 파일로 변환시켜주는 내장되어 있는 규칙

FRAMEWORK\_LIB = lib/library.a

: 정적 프레임워크 라이브러리

CFLAGS = -O2 -std=c++11

: 컴파일 옵션

ENABLE\_CUDNN = -D\_\_CUDNN\_\_

: CUDNN 사용 여부 플래그, CUDNN 미 설치 및 미 사용 시 주석 처리

DFLAGS = -g -D\_\_DEBUG\_\_

: 디버그 코드 사용 여부 플래그, 미 사용 시 주석 처리

INCLUDE\_PATH = -I/usr/local/cuda/include

: 전처리 과정에서 CUDA 헤더 파일을 탐색할 디렉토리 설정

LIB\_PATH = -L. -L/usr/local/cuda/lib64

: CUDA 라이브러리를 탐색할 디렉토리 설정

CC = g++

: 컴파일러 종류

NVCC = nvcc

: CUDA 파일 용 컴파일러

ifdef ENABLE\_CUDNN

LINKER = nvcc

LFLAGS = -lcudart -lcudnn -lpthread

else

LINKER = g++

LFLAGS = -lpthread

endif

: CUDNN 사용 시, NVCC 링커 사용 및 cudart, cudnn, pthread 라이브러리 사용

: CUDNN 미 사용 시, g++ 링커 사용 및 pthread 라이브러리 사용

AR = ar

: ar 유틸리티

SRCS = \

src/Utils.cpp \

src/Shape.cpp \

: CPP 소스 코드 파일

OBJS = ${SRCS:.cpp=.o}

: SRCS의 CPP 파일을 각각 컴파일해 Object 파일들을 생성

ifdef ENABLE\_CUDNN

CUDA\_SRCS = \

src/Utils\_CUDA.cu \

src/Optimizer/AdamOptimizer\_CUDA.cu \

src/Operator/Concatenate\_CUDA.cu

CUDA\_OBJS = ${CUDA\_SRCS:.cu=.o}

endif

: CUDNN 사용 시, CUDA\_SRC의 cu 파일을 각각 컴파일해 Object 파일들을 생성

all: $(FRAMEWORK\_LIB)

: 시작 타겟, FRAMEWORK\_LIB의 dependency 확인

.cpp.o:

$(CC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) $(LIB\_PATH) -c $< -o $@

: 주어진 파일을 옵션들을 이용해 컴파일하여 같은 이름의 오브젝트 파일을 생성한다.

src/Utils\_CUDA.o: src/Utils\_CUDA.cu

$(NVCC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) -c $< -o $@

src/Optimizer/AdamOptimizer\_CUDA.o: src/Optimizer/AdamOptimizer\_CUDA.cu

$(NVCC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) -c $< -o $@

src/Operator/Concatenate\_CUDA.o: src/Operator/Concatenate\_CUDA.cu

$(NVCC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) -c $< -o $@

: 해당 cu 파일의 dependency를 확인하고 옵션들을 이용해 컴파일하여 같은 이름의 오브젝트 파일을 생성한다.

$(FRAMEWORK\_LIB): $(OBJS) $(CUDA\_OBJS)

$(AR) rcs $@ $(OBJS) $(CUDA\_OBJS)

: OBJS, CUDA\_OBJS의 dependency를 확인하고, FRAMEWORK\_LIB, OBJS, CUDA\_OBJS의 Object 파일들을 가지고 아카이브 파일을 만든다.

clean:

rm -rf \*.o $(OBJS) $(CUDA\_OBJS) $(FRAMEWORK\_LIB)

: OBJS, CUDA\_OBJS, FRAMEWORK\_LIB의 Object 파일들을 제거한다.

**2) ImageNet (신경망 모델 예제 폴더)**

**(1) 전체 파일**

================================================================

.SUFFIXES = .cpp .o

CFLAGS = -O2 -std=c++11 -g

ENABLE\_CUDNN = -D\_\_CUDNN\_\_

DFLAGS = -g -D\_\_DEBUG\_\_

LFLAGS = -lcudart -lcudnn -lpthread -ljpeg -lturbojpeg

INCLUDE\_PATH = -I/usr/local/cuda/include -I/opt/libjpeg-turbo/include -I/opt/libjpeg-turbo/lib64

LIB\_PATH = -L. -L/usr/local/cuda/lib64 -L/opt/libjpeg-turbo/lib64

CC = g++

NVCC = nvcc

FRAMEWORK\_LIB = ../../lib/library.a

SRCS = \

../../src/Shape.cpp \

../../src/LongArray.cpp \

../../src/Tensor.cpp \

../../src/Operator.cpp \

../../src/LossFunction.cpp \

../../src/Optimizer.cpp \

../../src/Module.cpp \

../../src/NeuralNetwork.cpp

all: main

.cpp.o:

$(CC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) $(LIB\_PATH) -c $< -o $@

main: main.o $(FRAMEWORK\_LIB)

$(NVCC) $(CFLAGS) $(ENABLE\_CUDNN) $(DFLAGS) $(LFLAGS) $(INCLUDE\_PATH) $(LIB\_PATH) -o $@ $(FRAMEWORK\_LIB) main.o

$(FRAMEWORK\_LIB): $(SRC)

make -C ../..

clean:

rm -rf \*.o main

clean\_all:

rm -rf \*.o main

make clean -C ../..

================================================================

**(2) 파일 설명**

.SUFFIXES = .cpp .o

: Suffix, CPP 파일을 Object 파일로 변환시켜주는 내장되어 있는 규칙

CFLAGS = -O2 -std=c++11 -g

: 컴파일 옵션

ENABLE\_CUDNN = -D\_\_CUDNN\_\_

: CUDNN 사용 여부 플래그, CUDNN 미 설치 및 미 사용 시 주석 처리

DFLAGS = -g -D\_\_DEBUG\_\_

: 디버그 코드 사용 여부 플래그, 미 사용 시 주석 처리

LFLAGS = -lcudart -lcudnn -lpthread -ljpeg -lturbojpeg

: 라이브러리 사용 플래그, cudart, cudnn, pthread, jpeg, turbojpeg 라이브러리 사용

INCLUDE\_PATH = -I/usr/local/cuda/include -I/opt/libjpeg-turbo/include -I/opt/libjpeg-turbo/lib64

: 전처리 과정에서 CUDA 헤더 파일을 탐색할 디렉토리 설정

LIB\_PATH = -L. -L/usr/local/cuda/lib64 -L/opt/libjpeg-turbo/lib64

: CUDA 라이브러리를 탐색할 디렉토리 설정

CC = g++

: 컴파일러 종류

NVCC = nvcc

: CUDA 파일 용 컴파일러

FRAMEWORK\_LIB = ../../lib/library.a

: 프레임워크 라이브러리 아카이브 파일

SRCS = \

../../src/Shape.cpp \

../../src/LongArray.cpp \

../../src/Tensor.cpp \

../../src/Operator.cpp \

../../src/LossFunction.cpp \

../../src/Optimizer.cpp \

../../src/Module.cpp \

../../src/NeuralNetwork.cpp

: CPP 소스 코드 파일, 아카이브 파일의 dependency 확인을 위해 존재

all: main

: 시작 타겟, main 파일의 dependency 확인

.cpp.o:

$(CC) $(CFLAGS) $(DFLAGS) $(ENABLE\_CUDNN) $(INCLUDE\_PATH) $(LIB\_PATH) -c $< -o $@

: 주어진 파일을 옵션들을 이용해 컴파일하여 같은 이름의 오브젝트 파일을 생성한다.

main: main.o $(FRAMEWORK\_LIB)

$(NVCC) $(CFLAGS) $(ENABLE\_CUDNN) $(DFLAGS) $(LFLAGS) $(INCLUDE\_PATH) $(LIB\_PATH) -o $@ $(FRAMEWORK\_LIB) main.o

: main.o 파일과 FRAMEWORK\_LIB의 dependency를 확인하고, 옵션들을 이용해 FRAMEWORK\_LIB과 main.o 파일을 main 파일로 링크한다.

$(FRAMEWORK\_LIB): $(SRC)

make -C ../..

: 지정된 경로에서 make 실행

clean:

rm -rf \*.o main

: 해당 폴더의 모든 오브젝트 파일과 main 파일 제거

clean\_all:

rm -rf \*.o main

make clean -C ../..

: 해당 폴더의 모든 오브젝트 파일과 main 파일 제거,

지정된 경로에서 make clean 실행

# 4. 라이브러리 사용

## 4.1. 튜토리얼 코드 설명

### 4.1.1. MNIST(CNN)

**1) 신경망 모델 정의**

MNIST 튜토리얼의 신경망 모델은 다음과 같이 정의된다.

SetInput을 통해 2가지 input (data, label)이 모델의 학습에 들어 간다는 것을 명시한다.

ReShape을 통해 image data x를 Tensor 형태로 바꾼다

Convolution, BatchNorm, Relu, Maxpooling등의 Operator들로 신경망 모델을 정의한다.

SetLossFunction과 SetOptimizer를 이용해 손실함수와 Optimizer를 정의한다.

================================================================

class my\_CNN : public NeuralNetwork<float>{

private:

public:

my\_CNN(Tensorholder<float> \*x, Tensorholder<float> \*label) {

SetInput(2, x, label);

Operator<float> \*out = NULL;

out = new ReShape<float>(x, 28, 28, "Flat2Image");

// ======================= layer 1=======================

out = new ConvolutionLayer2D<float>(out, 1, 32, 3, 3, 1, 1, 0, TRUE, "Conv\_1");

out = new BatchNormalizeLayer<float>(out, TRUE, "BN\_1");

out = new Relu<float>(out, "Relu\_1");

out = new Maxpooling2D<float>(out, 2, 2, 2, 2, "MaxPool\_1");

// ======================= layer 2=======================

out = new ConvolutionLayer2D<float>(out, 32, 64, 3, 3, 1, 1, 0, TRUE, "Conv\_2");

out = new BatchNormalizeLayer<float>(out, TRUE, "BN\_2");

out = new Relu<float>(out, "Relu\_2");

out = new Maxpooling2D<float>(out, 2, 2, 2, 2, "MaxPool\_2");

// ======================= layer 3=======================

out = new ReShape<float>(out, 1, 1, 5 \* 5 \* 64, "Image2Flat");

// ======================= layer 3=======================

out = new Linear<float>(out, 5 \* 5 \* 64, 1024, TRUE, "Fully-Connected\_1");

out = new Relu<float>(out, "Relu\_3");

//// ======================= layer 4=======================

out = new Linear<float>(out, 1024, 10, TRUE, "Fully-connected\_2");

AnalyzeGraph(out);

// ======================= Select LossFunction Function ===================

SetLossFunction(new SoftmaxCrossEntropy<float>(out, label, "SCE"));

// ======================= Select Optimizer ===================

SetOptimizer(new RMSPropOptimizer<float>(GetParameter(), 0.01, 0.9, 1e-08, FALSE, MINIMIZE));

}

virtual ~my\_CNN() {}

};

**2) 메인(Main) 함수**

**Training 준비**

Tensorholder(placeholder)를 이용하여 이미지와 label을 저장 할 변수를 생성한다.

사전에 CNN class를 이용해 신경망을 생성한다.

MNIST data pair를 저장 할 변수를 생성한다.

// create input, label data placeholder -> Tensorholder

Tensorholder<float> \*x = new Tensorholder<float>(1, BATCH, 1, 1, 784, "x");

Tensorholder<float> \*label = new Tensorholder<float>(1, BATCH, 1, 1, 10, "label");

// ======================= Select net ===================

NeuralNetwork<float> \*net = new my\_CNN(x, label);

// ======================= Prepare Data ===================

MNISTDataSet<float> \*dataset = CreateMNISTDataSet<float>();

**Training**

MNIST dataset으로부터 이미지와 label을 받아 저장한다.

cudnn을 사용 할 경우, placeholder의 device를 GPU로 설정한다.

신경망에 데이터 pair(이미지, label)를 만들어 넣고 신경망의 Gradient 값을 초기화 하고 학습하는 과정을 반복한다.

// ======================= Train =======================

float train\_accuracy = 0.f;

float train\_avg\_loss = 0.f;

net->SetModeTrain();

startTime = clock();

for (int j = 0; j < LOOP\_FOR\_TRAIN; j++) {

dataset->CreateTrainDataPair(BATCH);

Tensor<float> \*x\_t = dataset->GetTrainFeedImage();

Tensor<float> \*l\_t = dataset->GetTrainFeedLabel();

#ifdef \_\_CUDNN\_\_

x\_t->SetDeviceGPU(GPUID); // 추후 자동화 필요

l\_t->SetDeviceGPU(GPUID);

#endif // \_\_CUDNN\_\_

// std::cin >> temp;

net->FeedInputTensor(2, x\_t, l\_t);

net->ResetParameterGradient();

net->Train();

// std::cin >> temp;

train\_accuracy += net->GetAccuracy();

train\_avg\_loss += net->GetLoss();

printf("\rTrain complete percentage is %d / %d -> loss : %f, acc : %f" /\*(ExcuteTime : %f)\*/,

j + 1, LOOP\_FOR\_TRAIN,

train\_avg\_loss / (j + 1),

train\_accuracy / (j + 1)

/\*nProcessExcuteTime\*/);

fflush(stdout);

}

endTime = clock();

nProcessExcuteTime = ((double)(endTime - startTime)) / CLOCKS\_PER\_SEC;

printf("\n(excution time per epoch : %f)\n\n", nProcessExcuteTime);

**Test**

신경망을 테스트 모드(gradient를 계산하지 않음)로 전환한 뒤 MNIST dataset으로부터 이미지와 label을 받아 저장한다.

신경망에 데이터(image, label)를 넣어 정확도와 손실을 측정한다.

// ======================= Test ======================

float test\_accuracy = 0.f;

float test\_avg\_loss = 0.f;

net->SetModeInference();

for (int j = 0; j < (int)LOOP\_FOR\_TEST; j++) {

dataset->CreateTestDataPair(BATCH);

Tensor<float> \*x\_t = dataset->GetTestFeedImage();

Tensor<float> \*l\_t = dataset->GetTestFeedLabel();

#ifdef \_\_CUDNN\_\_

x\_t->SetDeviceGPU(GPUID);

l\_t->SetDeviceGPU(GPUID);

#endif // \_\_CUDNN\_\_

net->FeedInputTensor(2, x\_t, l\_t);

net->Test();

test\_accuracy += net->GetAccuracy();

test\_avg\_loss += net->GetLoss();

printf("\rTest complete percentage is %d / %d -> loss : %f, acc : %f",

j + 1, LOOP\_FOR\_TEST,

test\_avg\_loss / (j + 1),

test\_accuracy / (j + 1));

fflush(stdout);

}

### 4.1.2. ImageNet(ResNet)

**1) 신경망 모델 정의**

본 프레임 워크의 ResNet은 여러 개의 BasicBlock(residual block)을 적층시켜 구성한다.

BasicBlock은 Convolution layer, BatchNormailze layer, Relu로 이루어진 2개의 layer와 skip connection(Short cut)으로 이루어진다. skip connection은 Addall Operator로 구현된다.

SetLossFunction과 SetOptimizer를 이용해 손실함수와 Optimizer를 정의한다.

template<typename DTYPE> class BasicBlock :

public Module<DTYPE>{

private:

public:

BasicBlock(Operator<DTYPE> \*pInput, int pNumInputChannel, int pNumOutputChannel, int pStride = 1, std::string pName = NULL) {

Alloc(pInput, pNumInputChannel, pNumOutputChannel, pStride, pName);

}

virtual ~BasicBlock() {}

int Alloc(Operator<DTYPE> \*pInput, int pNumInputChannel, int pNumOutputChannel, int pStride, std::string pName) {

this->SetInput(pInput);

Operator<DTYPE> \*remember = pInput;

Operator<DTYPE> \*out = pInput;

// 1

out = new ConvolutionLayer2D<DTYPE>(out, pNumInputChannel, pNumOutputChannel, 3, 3, pStride, pStride, 1, FALSE, "BasicBlock\_Conv1" + pName);

out = new BatchNormalizeLayer<DTYPE>(out, TRUE, "BasicBlock\_BN1" + pName);

out = new Relu<DTYPE>(out, "BasicBlock\_Relu1" + pName);

// 2

out = new ConvolutionLayer2D<DTYPE>(out, pNumOutputChannel, pNumOutputChannel, 3, 3, 1, 1, 1, FALSE, "BasicBlock\_Conv2" + pName);

out = new BatchNormalizeLayer<DTYPE>(out, TRUE, "BasicBlock\_BN2" + pName);

// ShortCut

if ((pStride != 1) || (pNumInputChannel != pNumOutputChannel)) {

remember = new ConvolutionLayer2D<DTYPE>(remember, pNumInputChannel, pNumOutputChannel, 1, 1, pStride, pStride, 0, FALSE, "BasicBlock\_Conv3\_Shortcut" + pName);

remember = new BatchNormalizeLayer<DTYPE>(remember, TRUE, "BasicBlock\_BN3\_Shortcut" + pName);

// remember = new Relu<DTYPE>(remember, "BasicBlock\_Relu3\_Shortcut" + pName);

}

// Add (for skip Connection)

out = new Addall<DTYPE>(remember, out, "ResNet\_Skip\_Add" + pName);

// Last Relu

out = new Relu<DTYPE>(out, "BasicBlock\_Relu3" + pName);

this->AnalyzeGraph(out);

return TRUE;

}

};

2) BasicBlock을 이용한 ResNet의 정의

본 프레임워크의 ResNet 튜토리얼의 인공 신경망은 Conv, BatchNorm 등의 Operator들과 사전에 정의한 4개의 BasicBlock으로 모델을 구성한다.

template<typename DTYPE> class ResNet :

public NeuralNetwork<DTYPE>{

private:

int m\_numInputChannel;

public:

ResNet(Tensorholder<DTYPE> \*pInput, Tensorholder<DTYPE> \*pLabel, std::string pBlockType, int pNumOfBlock1, int pNumOfBlock2, int pNumOfBlock3, int pNumOfBlock4, int pNumOfClass) {

Alloc(pInput, pLabel, pBlockType, pNumOfBlock1, pNumOfBlock2, pNumOfBlock3, pNumOfBlock4, pNumOfClass);

}

virtual ~ResNet() {}

int Alloc(Tensorholder<DTYPE> \*pInput, Tensorholder<DTYPE> \*pLabel, std::string pBlockType, int pNumOfBlock1, int pNumOfBlock2, int pNumOfBlock3, int pNumOfBlock4, int pNumOfClass) {

this->SetInput(2, pInput, pLabel);

m\_numInputChannel = 64;

Operator<DTYPE> \*out = pInput;

// ReShape

out = new ReShape<DTYPE>(out, 3, 224, 224, "ReShape");

// 1

out = new ConvolutionLayer2D<DTYPE>(out, 3, m\_numInputChannel, 7, 7, 2, 2, 3, FALSE, "Conv");

out = new BatchNormalizeLayer<DTYPE>(out, TRUE, "BN0");

out = new Relu<DTYPE>(out, "Relu0");

out = new Maxpooling2D<float>(out, 2, 2, 3, 3, 1, "MaxPool\_2");

out = this->MakeLayer(out, m\_numInputChannel, pBlockType, pNumOfBlock1, 1, "Block1");

out = this->MakeLayer(out, 128, pBlockType, pNumOfBlock2, 2, "Block2");

out = this->MakeLayer(out, 256, pBlockType, pNumOfBlock3, 2, "Block3");

out = this->MakeLayer(out, 512, pBlockType, pNumOfBlock3, 2, "Block4");

out = new GlobalAvaragePooling2D<DTYPE>(out, "Avg Pooling");

out = new ReShape<DTYPE>(out, 1, 1, 512, "ReShape");

out = new Linear<DTYPE>(out, 512, pNumOfClass, FALSE, "Classification");

this->AnalyzeGraph(out);

// ======================= Select LossFunction Function ===================

this->SetLossFunction(new SoftmaxCrossEntropy<float>(out, pLabel, "SCE"));

// ======================= Select Optimizer ===================

this->SetOptimizer(new GradientDescentOptimizer<float>(this->GetParameter(), 0.1, 0.9, 5e-4, MINIMIZE));

return TRUE;

}

Operator<DTYPE>\* MakeLayer(Operator<DTYPE> \*pInput, int pNumOfChannel, std::string pBlockType, int pNumOfBlock, int pStride, std::string pName = NULL) {

if (pNumOfBlock == 0) {

return pInput;

} else if ((pBlockType == "BasicBlock") && (pNumOfBlock > 0)) {

Operator<DTYPE> \*out = pInput;

out = new BasicBlock<DTYPE>(out, m\_numInputChannel, pNumOfChannel, pStride, pName);

int pNumOutputChannel = pNumOfChannel;

for (int i = 1; i < pNumOfBlock; i++) {

out = new BasicBlock<DTYPE>(out, pNumOutputChannel, pNumOutputChannel, 1, pName);

}

m\_numInputChannel = pNumOutputChannel;

return out;

} else if ((pBlockType == "Bottleneck") && (pNumOfBlock > 0)) {

return NULL;

} else return NULL;

}

};

**ResNet의 종류는 BasicBlock 내부의 채널 수를 다르게 함으로써 나누어진다.**

template<typename DTYPE> NeuralNetwork<DTYPE>\* Resnet10(Tensorholder<DTYPE> \*pInput, Tensorholder<DTYPE> \*pLabel, int pNumOfClass) {

return new ResNet<DTYPE>(pInput, pLabel, "BasicBlock", 1, 1, 1, 1, pNumOfClass);

}

template<typename DTYPE> NeuralNetwork<DTYPE>\* Resnet18(Tensorholder<DTYPE> \*pInput, Tensorholder<DTYPE> \*pLabel, int pNumOfClass) {

return new ResNet<DTYPE>(pInput, pLabel, "BasicBlock", 2, 2, 2, 2, pNumOfClass);

}

template<typename DTYPE> NeuralNetwork<DTYPE>\* Resnet34(Tensorholder<DTYPE> \*pInput, Tensorholder<DTYPE> \*pLabel, int pNumOfClass) {

return new ResNet<DTYPE>(pInput, pLabel, "BasicBlock", 3, 4, 6, 3, pNumOfClass);

}

**2) 메인(Main)함수**

**데이터 준비**

데이터 placeholder를  만들고 사전에 정의된 인공 신경망 생성 부분은 MNIST와 같다.

ImageNet 데이터는 random crop, normalization, random Filp 과정을 거쳐 전처리 된다.

// ======================= Prepare Data ===================

ImageNetDataReader<float> \*train\_data\_reader = new ImageNetDataReader<float>(BATCH, 25, TRUE);

train\_data\_reader->UseRandomCrop(28);

train\_data\_reader->UseNormalization(TRUE, mean, stddev);

train\_data\_reader->UseRandomHorizontalFlip();

train\_data\_reader->UseRandomVerticalFlip();

ImageNetDataReader<float> \*test\_data\_reader = new ImageNetDataReader<float>(BATCH, 25, FALSE);

test\_data\_reader->UseNormalization(TRUE, mean, stddev);

train\_data\_reader->StartProduce();

test\_data\_reader->StartProduce();

**학습**

전처리된 데이터를 버퍼로부터 가져와 신경망에 넣고 Gradient를 초기화 한 뒤 학습하는 과정을 반복한다.

// ======================= Train =======================

float train\_avg\_accuracy = 0.f;

float train\_avg\_top5\_accuracy = 0.f;

float train\_cur\_accuracy = 0.f;

float train\_cur\_top5\_accuracy = 0.f;

float train\_avg\_loss = 0.f;

float train\_cur\_loss = 0.f;

net->SetModeTrain();

for (int j = 0; j < LOOP\_FOR\_TRAIN; j++) {

data = train\_data\_reader->GetDataFromBuffer();

#ifdef \_\_CUDNN\_\_

data[0]->SetDeviceGPU(GPUID); // 추후 자동화 필요

data[1]->SetDeviceGPU(GPUID);

#endif // \_\_CUDNN\_\_

net->FeedInputTensor(2, data[0], data[1]);

delete data;

data = NULL;

net->ResetParameterGradient();

net->Train();

train\_cur\_accuracy = net->GetAccuracy(NUMBER\_OF\_CLASS);

train\_cur\_top5\_accuracy = net->GetTop5Accuracy(NUMBER\_OF\_CLASS);

train\_cur\_loss = net->GetLoss();

train\_avg\_accuracy += train\_cur\_accuracy;

train\_avg\_top5\_accuracy += train\_cur\_top5\_accuracy;

train\_avg\_loss += train\_cur\_loss;

printf("\r%d / %d -> cur\_loss : %0.4f, avg\_loss : %0.4f, cur\_acc : %0.5f, avg\_acc : %0.5f, cur\_top5\_acc : %0.5f, avg\_top5\_acc : %0.5f" /\*(ExcuteTime : %f)\*/,

j + 1, LOOP\_FOR\_TRAIN,

train\_cur\_loss,

train\_avg\_loss / (j + 1),

train\_cur\_accuracy,

train\_avg\_accuracy / (j + 1),

train\_cur\_top5\_accuracy,

train\_avg\_top5\_accuracy / (j + 1));

fflush(stdout);

**Test**

신경망을 inference 모드로 전환 한 뒤, 전처리된 데이터를 버퍼로부터 가져와 신경망에 넣어 정확도와 손실을 측정한다.

// ======================= Test ======================

float test\_avg\_accuracy = 0.f;

float test\_avg\_top5\_accuracy = 0.f;

float test\_avg\_loss = 0.f;

net->SetModeInference();

for (int j = 0; j < (int)LOOP\_FOR\_TEST; j++) {

data = test\_data\_reader->GetDataFromBuffer();

#ifdef \_\_CUDNN\_\_

data[0]->SetDeviceGPU(GPUID); // 추후 자동화 필요

data[1]->SetDeviceGPU(GPUID);

#endif // \_\_CUDNN\_\_

net->FeedInputTensor(2, data[0], data[1]);

delete data;

data = NULL;

net->Test();

test\_avg\_accuracy += net->GetAccuracy(NUMBER\_OF\_CLASS);

test\_avg\_top5\_accuracy += net->GetTop5Accuracy(NUMBER\_OF\_CLASS);

test\_avg\_loss += net->GetLoss();

printf("\r%d / %d -> avg\_loss : %0.4f, avg\_acc : %0.4f, avg\_top5\_acc : %0.4f" /\*(ExcuteTime : %f)\*/,

j + 1, LOOP\_FOR\_TEST,

test\_avg\_loss / (j + 1),

test\_avg\_accuracy / (j + 1),

test\_avg\_top5\_accuracy / (j + 1));

fflush(stdout);

}

if (best\_acc < test\_avg\_accuracy / LOOP\_FOR\_TEST) {

std::cout << "\nsave parameters...";

FILE \*fp = fopen(filename, "wb");

net->Save(fp);

fclose(fp);

FILE \*fp\_info = fopen(filename\_info, "wb");

best\_acc = test\_avg\_accuracy / LOOP\_FOR\_TEST;

fwrite(&best\_acc, sizeof(float), 1, fp\_info);

fwrite(&i, sizeof(int), 1, fp\_info);

fclose(fp\_info);

std::cout << "done" << "\n\n";

} else std::cout << "\n\n";

### 4.1.3. ImageNet(DenseNet)

**1) 신경망 모델 정의**

본 프레임 워크의 DenseNet은 여러 개의 DenseNet Block을 적층시켜 구성한다.

DenseNet Block은 다음과 같이 정의된다

template<typename DTYPE> class DenseNetBlock : public Module<DTYPE>{

private:

public:

DenseNetBlock(Operator<DTYPE> \*pInput, int pNumInputChannel, int pGrowthRate, std::string pName = NULL) : Module<DTYPE>(pName) {

Alloc(pInput, pNumInputChannel, pGrowthRate, pName);

}

virtual ~DenseNetBlock() {}

int Alloc(Operator<DTYPE> \*pInput, int pNumInputChannel, int pGrowthRate, std::string pName) {

this->SetInput(pInput);

Operator<DTYPE> \*remember = pInput;

Operator<DTYPE> \*out = pInput;

// 1

out = new BatchNormalizeLayer<DTYPE>(out, TRUE, "DenseNetBlock\_BN1" + pName);

out = new Relu<DTYPE>(out, "DenseNetBlock\_Relu1" + pName);

out = new ConvolutionLayer2D<DTYPE>(out, pNumInputChannel, 4 \* pGrowthRate, 1, 1, 1, 1, 0, FALSE, "DenseNetBlock\_Conv1" + pName);

// 2

out = new BatchNormalizeLayer<DTYPE>(out, TRUE, "DenseNetBlock\_BN2" + pName);

out = new Relu<DTYPE>(out, "DenseNetBlock\_Relu2" + pName);

out = new ConvolutionLayer2D<DTYPE>(out, 4 \* pGrowthRate, pGrowthRate, 3, 3, 1, 1, 1, FALSE, "DenseNetBlock\_Conv2" + pName);

// Concat

out = new ConcatenateChannelWise<float>(remember, out, "DenseNetBlock\_ConCat");

this->AnalyzeGraph(out);

return TRUE;

}

};

DenseNet Block 사이를 연결하는 Transition은 다음과 같이 정의된다.

template<typename DTYPE> class Transition : public Module<DTYPE>{

private:

public:

Transition(Operator<DTYPE> \*pInput, int pNumInputChannel, int pNumOutputChannel, std::string pName = NULL) : Module<DTYPE>(pName) {

Alloc(pInput, pNumInputChannel, pNumOutputChannel, pName);

}

virtual ~Transition() {}

int Alloc(Operator<DTYPE> \*pInput, int pNumInputChannel, int pNumOutputChannel, std::string pName) {

this->SetInput(pInput);

Operator<DTYPE> \*out = pInput;

// 1

out = new BatchNormalizeLayer<DTYPE>(out, TRUE, "Transition\_BN" + pName);

out = new Relu<DTYPE>(out, "Transition\_Relu" + pName);

out = new ConvolutionLayer2D<DTYPE>(out, pNumInputChannel, pNumOutputChannel, 1, 1, 1, 1, 0, FALSE, "Transition\_Conv" + pName);

// Avg Pooling

out = new AvaragePooling2D<float>(out, 2, 2, 2, 2, 0, "AVG");

this->AnalyzeGraph(out);

return TRUE;

}

};

본 프레임 워크의 튜토리얼 DenseNet은 4개의 DenseNet Block과 각각의 DenseNet Block 사이를 잇는 4개의 transition으로 정의된다.

template<typename DTYPE> class DenseNet : public NeuralNetwork<DTYPE>{

private:

int m\_numInputChannel;

int m\_numOutputChannel;

int m\_growthRate;

float m\_reduction;

public:

DenseNet(Tensorholder<DTYPE> \*pInput, Tensorholder<DTYPE> \*pLabel, std::string pBlockType, int pNumOfBlock1, int pNumOfBlock2, int pNumOfBlock3, int pNumOfBlock4, int pGrowthRate = 12, float pReduction = 0.5, int pNumOfClass = 1000) {

Alloc(pInput, pLabel, pBlockType, pNumOfBlock1, pNumOfBlock2, pNumOfBlock3, pNumOfBlock4, pGrowthRate, pReduction, pNumOfClass);

}

virtual ~DenseNet() {}

int Alloc(Tensorholder<DTYPE> \*pInput, Tensorholder<DTYPE> \*pLabel, std::string pBlockType, int pNumOfBlock1, int pNumOfBlock2, int pNumOfBlock3, int pNumOfBlock4, int pGrowthRate, float pReduction, int pNumOfClass) {

this->SetInput(2, pInput, pLabel);

m\_numInputChannel = 2 \* pGrowthRate;

m\_growthRate = pGrowthRate;

m\_reduction = pReduction;

Operator<DTYPE> \*out = pInput;

// ReShape

out = new ReShape<DTYPE>(out, 3, 224, 224, "ReShape");

// out = new BatchNormalizeLayer<DTYPE>(out, TRUE, "BN0");

// 1

out = new ConvolutionLayer2D<DTYPE>(out, 3, m\_numInputChannel, 7, 7, 2, 2, 3, FALSE, "Conv");

out = new BatchNormalizeLayer<DTYPE>(out, TRUE, "BN0");

out = new Relu<DTYPE>(out, "Relu0");

out = this->MakeLayer(out, m\_numInputChannel, pBlockType, pNumOfBlock1, 1, "Block1");

m\_numInputChannel += pNumOfBlock1 \* m\_growthRate;

m\_numOutputChannel = (int)(floor(m\_numInputChannel \* m\_reduction));

out = new Transition<DTYPE>(out, m\_numInputChannel, m\_numOutputChannel, "Trans1");

m\_numInputChannel = m\_numOutputChannel;

out = this->MakeLayer(out, m\_numOutputChannel, pBlockType, pNumOfBlock2, 1, "Block2");

m\_numInputChannel += pNumOfBlock2 \* m\_growthRate;

m\_numOutputChannel = (int)(floor(m\_numInputChannel \* m\_reduction));

out = new Transition<DTYPE>(out, m\_numInputChannel, m\_numOutputChannel, "Trans2");

m\_numInputChannel = m\_numOutputChannel;

out = this->MakeLayer(out, m\_numOutputChannel, pBlockType, pNumOfBlock3, 1, "Block3");

m\_numInputChannel += pNumOfBlock3 \* m\_growthRate;

m\_numOutputChannel = (int)(floor(m\_numInputChannel \* m\_reduction));

out = new Transition<DTYPE>(out, m\_numInputChannel, m\_numOutputChannel, "Trans3");

m\_numInputChannel = m\_numOutputChannel;

out = this->MakeLayer(out, m\_numOutputChannel, pBlockType, pNumOfBlock4, 1, "Block4");

m\_numInputChannel += pNumOfBlock4 \* m\_growthRate;

m\_numOutputChannel = (int)(floor(m\_numInputChannel \* m\_reduction));

out = new Transition<DTYPE>(out, m\_numInputChannel, m\_numOutputChannel, "Trans4");

m\_numInputChannel = m\_numOutputChannel;

out = new GlobalAvaragePooling2D<DTYPE>(out, "Avg Pooling");

out = new ReShape<DTYPE>(out, 1, 1, m\_numInputChannel, "ReShape");

out = new Linear<DTYPE>(out, m\_numInputChannel, pNumOfClass, TRUE, "Classification");

this->AnalyzeGraph(out);

// ======================= Select LossFunction Function ===================

this->SetLossFunction(new SoftmaxCrossEntropy<float>(out, pLabel, "SCE"));

// ======================= Select Optimizer ===================

this->SetOptimizer(new AdamOptimizer<float>(this->GetParameter(), 0.01, 0.9, 0.999, 1e-08, 5e-4, MINIMIZE));

return TRUE;

}

Operator<DTYPE>\* MakeLayer(Operator<DTYPE> \*pInput, int pNumOfChannel, std::string pBlockType, int pNumOfBlock, int pStride, std::string pName = NULL) {

if (pNumOfBlock == 0) {

return pInput;

} else if ((pBlockType == "DenseNetBlock") && (pNumOfBlock > 0)) {

Operator<DTYPE> \*out = pInput;

for (int i = 0; i < pNumOfBlock; i++) {

out = new DenseNetBlock<DTYPE>(out, pNumOfChannel, m\_growthRate, pName);

pNumOfChannel += m\_growthRate;

}

return out;

} else if ((pBlockType == "Bottleneck") && (pNumOfBlock > 0)) {

return NULL;

} else return NULL;

}

};

template<typename DTYPE> NeuralNetwork<DTYPE>\* DenseNet121(Tensorholder<DTYPE> \*pInput, Tensorholder<DTYPE> \*pLabel, int pNumOfClass) {

return new DenseNet<DTYPE>(pInput, pLabel, "DenseNetBlock", 6, 12, 24, 16, 12, 0.5, pNumOfClass);

}

### 4.1.3. GAN(Vanilla GAN)

**1) 신경망 모델 정의**

본 프레임 워크의 GAN은 Generator와 Discriminator를 각각 구성한다.

1) Generator  
template<typename DTYPE> class my\_Generator : public NeuralNetwork<DTYPE> {

private:

public:

my\_Generator(Operator<float> \*z){

Alloc(z);

}

virtual ~my\_Generator() {

}

int Alloc(Operator<float> \*z){

this->SetInput(z);

Operator<float> \*out = z;

out = new Linear<float>(out, 100, 128, TRUE, "G\_L1");

out = new Linear<float>(out, 128, 256, TRUE, "G\_L2");

out = new BatchNormalizeLayer<DTYPE>(out, FALSE, "G\_BN1");

out = new LRelu<float>(out, 0.2, "G\_LRelu1");

out = new Linear<float>(out, 256, 512, TRUE, "G\_L3");

out = new BatchNormalizeLayer<DTYPE>(out, FALSE, "G\_BN2");

out = new LRelu<float>(out, 0.2, "G\_LRelu2");

out = new Linear<float>(out, 512, 1024, TRUE, "G\_L4");

out = new BatchNormalizeLayer<DTYPE>(out, FALSE, "G\_BN3");

out = new LRelu<float>(out, 0.2, "G\_LRelu3");

out = new Linear<float>(out, 1024, 784, TRUE, "G\_L5");

out = new Tanh<float>(out, "Tanh");

this->AnalyzeGraph(out);

}

};

2) Discriminator  
template<typename DTYPE> class my\_Discriminator : public NeuralNetwork<DTYPE> {

private:

public:

my\_Discriminator(Operator<float> \*x){

Alloc(x);

}

virtual ~my\_Discriminator() {

}

int Alloc(Operator<float> \*x){

this->SetInput(x);

Operator<float> \*out = x;

out = new Linear<float>(out, 784, 512, TRUE, "D\_L1");

out = new LRelu<float>(out, 0.2, "D\_LRelu1");

out = new Linear<float>(out, 512, 256, TRUE, "D\_L2");

out = new LRelu<float>(out, 0.2, "D\_LRelu2");

out = new Linear<float>(out, 256, 1, TRUE, "D\_L3");

out = new Sigmoid<float>(out, "D\_Sigmoid");

this->AnalyzeGraph(out);

}

};

3) Generator와 Discriminator의 학습을 제어하는 my\_GAN  
template<typename DTYPE> class my\_GAN : public GAN<DTYPE> {

private:

public:

my\_GAN(Tensorholder<float> \*z, Tensorholder<float> \*x, Tensorholder<float> \*label){

Alloc(z, x, label);

}

virtual ~my\_GAN() {

}

int Alloc(Tensorholder<float> \*z, Tensorholder<float> \*x, Tensorholder<float> \*label){

this->SetInput(3, z, x, label);

this->SetGenerator(new my\_Generator<float>(z));

this->SetSwitchInput(new SwitchInput<float>(this->GetGenerator(), x));

this->SetDiscriminator(new my\_Discriminator<float>(this->GetSwitchInput()));

this->AnalyzeGraph(this->GetDiscriminator());

this->SetLabel(label);

// ======================= Select LossFunction ===================

this->SetGANLossFunctions(new VanillaGeneratorLoss<float>(this->GetDiscriminator(), this->GetLabel(), "VanillaGeneratorLoss"), new VanillaDiscriminatorLoss<float>(this->GetDiscriminator(), this->GetLabel(), "VanillaDiscriminatorLoss"));

// ======================= Select Optimizer ===================

this->SetGANOptimizers(new AdamOptimizer<float>(this->GetGenerator()->GetParameter(), 0.0002, 0.5, 0.999, 1e-08, MINIMIZE), new AdamOptimizer<float>(this->GetDiscriminator()->GetParameter(), 0.0002, 0.5, 0.999, 1e-08, MINIMIZE));

}

};

**2) 메인(Main)함수**

**데이터 준비**Tensorholder(placeholder)를 이용하여 이미지와 label을 저장 할 변수를 생성한다.

사전에 정의한 GAN class를 이용해 신경망을 생성한다.

MNIST data pair를 저장 할 변수를 생성한다.

Generator의 input으로 사용할 latent variable을 저장 할 변수를 생성한다.

// create input, label data placeholder -> Tensorholder

Tensorholder<float> \*z = new Tensorholder<float>(1, BATCH, 1, 1, 100, "z");

Tensorholder<float> \*x = new Tensorholder<float>(1, BATCH, 1, 1, 784, "x");

Tensorholder<float> \*label = new Tensorholder<float>(1, BATCH, 1, 1, 1, "label");

// create NoiseGenrator

GaussianNoiseGenerator<float> \*Gnoise = new GaussianNoiseGenerator<float>(1, BATCH, 1, 1, 100, 0, 1);

// ======================= Select net ===================

// GAN<float> \*net = new my\_BEGAN<float>(z, x, label);

GAN<float> \*net = new my\_GAN<float>(z, x, label);

//net->Load(filename);

// ======================= Prepare Data ===================

MNISTDataSet<float> \*dataset = CreateMNISTDataSet<float>();

**학습**

전처리된 데이터와 생성한 latent variavle을 가지고 Gradient를 초기화하고 학습하는 과정을 반복한다.

Generator에서 생성한 이미지를 일정 step마다 저장한다.

//Start make Noise

Gnoise->StartProduce();

for (int i = epoch + 1; i < EPOCH; i++) {

std::cout << "EPOCH : " << i << '\n';

// ======================= Train =======================

float genLoss = 0.f;

float discLoss = 0.f;

net->SetModeTrain();

startTime = clock();

for (int j = 0; j < LOOP\_FOR\_TRAIN; j++) {

dataset->CreateTrainDataPair(BATCH);

Tensor<float> \*x\_t = dataset->GetTrainFeedImage();

Tensor<float> \*l\_t = dataset->GetTrainFeedLabel();

delete l\_t;

Tensor<float> \*z\_t = Gnoise->GetNoiseFromBuffer();

#ifdef \_\_CUDNN\_\_

x\_t->SetDeviceGPU(GPUID);

z\_t->SetDeviceGPU(GPUID);

#endif // \_\_CUDNN\_\_

net->FeedInputTensor(2, z\_t, x\_t);

net->ResetParameterGradient();

net->TrainDiscriminator();

z\_t = Gnoise->GetNoiseFromBuffer();

#ifdef \_\_CUDNN\_\_

z\_t->SetDeviceGPU(GPUID);

#endif // \_\_CUDNN\_\_

net->FeedInputTensor(1, z\_t);

net->ResetParameterGradient();

net->TrainGenerator();

genLoss = (\*net->GetGeneratorLossFunction()->GetResult())[Index5D(net->GetGeneratorLossFunction()->GetResult()->GetShape(), 0, 0, 0, 0, 0)];

discLoss = (\*net->GetDiscriminatorLossFunction()->GetResult())[Index5D(net->GetDiscriminatorLossFunction()->GetResult()->GetShape(), 0, 0, 0, 0, 0)];

printf("\rTrain complete percentage is %d / %d -> Generator Loss : %f, Discriminator Loss : %f",

j + 1,

LOOP\_FOR\_TRAIN,

genLoss,

discLoss);

fflush(stdout);

if(j % 50 == 0){

string filePath = "generated/epoch" + std::to\_string(i) + "\_" + std::to\_string(j) + ".jpg";

const char \*cstr = filePath.c\_str();

Tensor2Image<float>(net->GetGenerator()->GetResult()->GetResult(), cstr, 3, 20, 28, 28);

}

}

endTime = clock();

nProcessExcuteTime = ((double)(endTime - startTime)) / CLOCKS\_PER\_SEC;

printf("\n(excution time per epoch : %f)\n\n", nProcessExcuteTime);

// ======================= Test(Save Generated Image)======================

float testGenLoss = 0.f;

float testDiscLoss = 0.f;

net->SetModeInference();

**Test**

GAN에서는 정확도가 의미있는 것이 아니므로 Test에서는 Generator에서 이미지를 생성시킨 뒤 저장하는 과정을 반복한다.

net->FeedInputTensor(1, z\_t);

net->Test();

testGenLoss = (\*net->GetGeneratorLossFunction()->GetResult())[Index5D(net->GetGeneratorLossFunction()->GetResult()->GetShape(), 0, 0, 0, 0, 0)];

testDiscLoss = (\*net->GetDiscriminatorLossFunction()->GetResult())[Index5D(net->GetDiscriminatorLossFunction()->GetResult()->GetShape(), 0, 0, 0, 0, 0)];

string filePath = "evaluated/epoch" + std::to\_string(i) + "\_" + std::to\_string(j) + ".jpg";

const char \*cstr = filePath.c\_str();

Tensor2Image<float>(net->GetGenerator()->GetResult()->GetResult(), cstr, 3, 20, 28, 28);

printf("\rTest complete percentage is %d / %d -> loss : %f, acc : %f",

j + 1,

LOOP\_FOR\_TEST,

testGenLoss,

testDiscLoss);

fflush(stdout);

}