Define-by-Run 방식의 AI용 프레임워크 설계구현

허진욱(2018124193), 김소중(2015124043)

지도교수: 권용진

요 약

딥러닝 모델은 scratch 수준에서의 개발이 가능하나, 정의와 수정이 번거롭다는 단점이 있다. 이를 편리하게 하기 위해 PyTorch나 Tensorflow 등 많은 프레임워크가 개발되고 있고, 이로 인해 딥러닝 모델 구축이 간편해졌다. 그러나 프레임워크를 사용하는 것으로는 프레임워크의 내부구조를 알 수 없고, 딥러닝에 사용된 이론과 그 구현에 대해 알 수 없다. 또한 프레임워크는 GPU의 사용을 제공하며, GPU의 사용이 간단하다. 그러나 사용자가 GPU 사용의 구현을 확인할 수 없다. 따라서. 본 논문에서는 딥러닝 프레임워크의 내부구조와 GPU 사용의 구현, 딥러닝 네트워크에 대해 이해하고자 DeZero라는 딥러닝 프레임워크 서브셋의 구현, 확장을 진행하였다.

Keywords : AI, Deep Learning, 프레임워크, 인공지능, 딥러닝

Ⅰ. 서 론

딥러닝 모델을 scratch 수준에서 개발하는 것이 가능하지만, 모델의 정의와 수정이 번거롭다. 이러한 불편함을 해소하여, 딥러닝 모델을 편리하게 개발하기 위해 다양한 프레임워크가 개발되고 있다. 프레임워크는 딥러닝 모델 구축에 대한 매뉴얼을 제공하며, 이 매뉴얼을 따라하면 모델을 쉽게 구축할 수 있다. 그러나 프레임워크를 사용하면 프레임워크의 내부가 어떻게 이루어져 있는지 알 수 없고, 딥러닝에서 사용된 이론과 구현을 이해하기 어렵다. 본 논문에서는 이를 이해하고자 딥러닝 프레임워크 서브셋인 DeZero를 구현, 확장한다.

Ⅱ. DeZero 프레임워크 구현

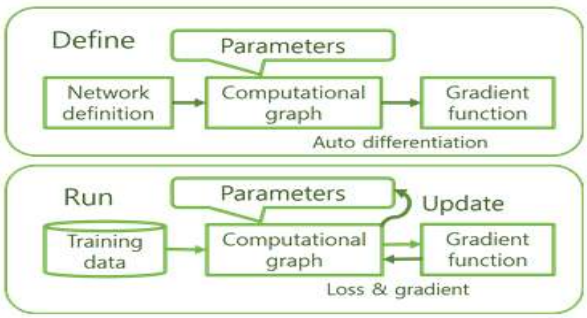
1. DeZero : 딥러닝 프레임워크의 서브셋

딥러닝 프레임워크의 서브셋인 DeZero는 Chainer라는 딥러닝 프레임워크를 참고하여 만들어졌다. 또한 DeZero는 현대 프레임워크(Chainer, PyTorch, Tensorflow 2.0 등) 과 동일하게 Define-by-Run 방식을 사용한다.

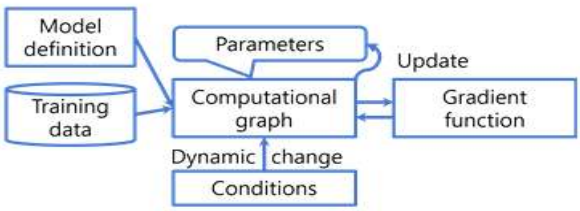
가. Define-by-Run 방식

Chainer 이전의 대부분의 딥러닝 프레임워크에서는 계산 그래프의 구현을 위해, Define-and-Run 방식을 사용했다. 하지만 Chainer가 Define-by-Run 방식을 제안한 이후,[1] PyTorch나 Tensorflow 2.0을 포함한 많은 프레임워크들이 Define-by-Run 방식을 채택하고 있다.[2][3]

기존의 Define-and-Run 방식은 먼저 딥러닝 네트워크를 정의한 후, compile을 통해 계산 그래프를 생성한다. 생성된 계산 그래프에 사용자가 훈련 데이터를 반복적으로 입력하며 학습을 진행한다 (그림 1a).



(a) Define-and-Run 방식



(b) Define-by-Run 방식

그림 1 Define-and-Run 방식과 Define-by-Run 방식[4]

그러나 Define-by-Run 방식은 계산 그래프가 학습 전에 제공되지 않고, 학습 과정에서 생성된다. 즉, 딥러닝 모델에 사용자가 훈련 데이터를 입력하여 학습을 진행할 때, 계산 그래프를 생성한다 (그림 1b).

나. DeZero 프레임워크의 구조

DeZero 프레임워크는 모델 정의와 계산 그래프의 생성, 파라미터 갱신, 데이터 생성의 세 부분으로 구성되어 있다.

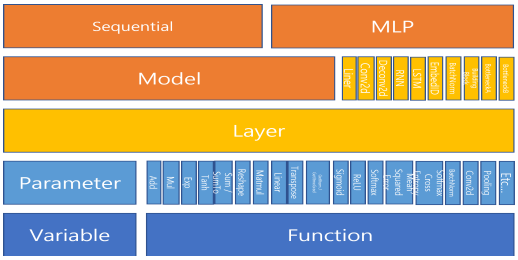


그림 2 모델 정의와 계산 그래프의 생성

그림 3 파라미터 갱신

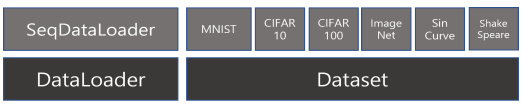


그림 4 데이터 생성

(1) 모델 정의와 계산 그래프의 생성 (그림 2)

계산 그래프를 생성하는 기본 클래스는 Variable 클래스와 Function 클래스이다. Variable 클래스는 DeZero의 기본 데이터형으로, PyTorch와 Tensorflow의 Tensor 역할을 수행한다. Variable은 data로 Numpy array만을 취급하도록 되어 있으며, backward 메서드를 통해 계산 그래프의 역전파를 수행하게 된다. Function 클래스는 순전파와 역전파 계산을 수행하는 함수를 구현하는 클래스로, Variable을 입력으로 받아, Variable을 출력하도록 되어 있다. Function이 호출될 때, 계산 그래프가 생성된다.

Variable를 상속받은 Parameter 클래스는 갱신해야 할 파라미터를 저장한다. 또한, Function을 상속받아 Variable 간의 덧셈이나 Affine 변환 같은 구체적인 계산을 수행하는 클래스가 정의되어 있다.

모델 정의를 담당하는 클래스는 Layer 클래스와 Model 클래스이다. Layer 클래스는 파라미터를 한 번에 관리하기 위한 클래스이다. Layer 클래스가 없으면 파라미터를 일일이 관리해야 하지만, Layer 클래스를 사용함으로써 한 번에 파라미터 관리를 수행할 수 있게 된다. Model 클래스는 딥러닝 모델의 정의를 위한 클래스이다.

(2) 파라미터 갱신 (그림 3)

Optimizer 클래스는 파라미터 갱신을 위한 클래스이다. Optimizer는 Layer로부터 넘겨받은 파라미터의 갱신과 전처리를 수행한다.

(3) 데이터 생성 (그림 4)

Dataset 클래스는 다양한 데이터셋을 통일된 인터페이스로 사용하게 해주는 클래스이다. 이 클래스에서 데이터의 전처리를 수행할 수 있게 되어 있다.

DataLoader 클래스는 Dataset 클래스로부터 미니배치를 생성하는 클래스로, 사용자로부터 데이터 요청이 오면 Dataset에서 미니배치를 생성한다.

다. DeZero 프레임워크의 GPU 대응

DeZero 프레임워크는 GPU 사용이 가능하도록 Cupy라는 Python 라이브러리를 사용하고 있다.

(1) Cupy 라이브러리

Cupy 라이브러리는 NVIDIA사의 GPU를 사용할 수 있게 해주는 파이썬 라이브러리이다. DeZero에서는 이 라이브러리의 세 가지 특징을 활용하여 GPU를 사용할 수 있도록 되어 있다.

- Numpy 라이브러리와 API가 거의 동일

Cupy는 Numpy 라이브러리와 거의 동일한 API를 제공한다. DeZero에서는 이를 이용하여 Variable이 GPU 상에서 동작할 수 있도록 수정되었다.

- Numpy array와 Cupy array 간의 교환

Cupy는 Numpy array와 Cupy array 간의 교환을 수행하는 메서드를 제공한다. 이를 통해 DeZero는 CPU 상에서의 데이터를 GPU 상으로, GPU 상에서의 데이터를 CPU 상으로의 데이터 교환이 가능하다.

- 데이터에 따른 적합한 모듈 반환 메서드

Cupy는 주어진 데이터가 Numpy array인지, Cupy array인지에 따라 Numpy 모듈과 Cupy 모듈을 반환하는 메서드를 제공한다. 이를 통해 Function 클래스의 재정의 없이 GPU 연산이 가능하도록 되어 있다.

(2) DeZero 클래스의 확장

Cupy 라이브러리의 특징을 활용하여 DeZero의 Variable 클래스와 Function 클래스가 다음과 같이 확장되어 있다.

- Variable 클래스의 확장

Numpy array만을 data로 취급할 수 있었던 Variable이 Cupy array도 취급할 수 있도록 확장되었다. 또한, CPU 상에서 GPU 상으로, GPU 상에서 CPU 상으로

data를 전송하는 메서드가 추가되었다.

- Function 클래스의 확장

Numpy array에 대한 연산만 가능하던 Function을 주어진 데이터에 따라 Cupy array에 대한 연산도 가능하도록 확장되었다.

라. Trainer 클래스 추가 구현

DeZero 프레임워크에서 모델을 정의할 때마다 학습 수행 코드를 반복적으로 작성해야 하는 번거로움을 없애기 위해 Trainer 클래스를 추가 구현하였다 (그림 5).

```
train_set = dezero.datasets.MNIST(train=True)
train_loader = DataLoader(train_set, batch_size)
model = MLP((1000, 10))
optimizer = optimizers.SGD().setup(model)

# GPU mode
if dezero.cuda.gpu_enable:
    train_loader.to_gpu()
    model.to_gpu()

# Training loop
for epoch in range(max_epoch):
    start = time.time()
    sum_loss = 0

    for x, t in train_loader:
        y = model(x)
        loss = F.softmax_cross_entropy(y, t)
        model.cleargrads()
        loss.backward()
        optimizer.update()
        sum_loss += float(loss.data) * len(t)

    elapsed_time = time.time() - start
    print('epoch: {}, loss: {:.4f}, time: {:.4f}[sec]'.format(
        epoch + 1, sum_loss / len(train_set), elapsed_time))
```

그림 5 Trainer 클래스가 없는 DeZero 프레임워크

Trainer 클래스는 딥러닝 모델의 학습을 대신 수행해주는 클래스이다. 딥러닝 모델의 학습은 미니배치 생성, 모델의 추론 수행, 손실 계산, 기울기 산출, 파라미터 갱신의 과정을 반복적으로 수행하게 된다. 이 과정을 Model, 손실함수, Optimizer, DataLoader를 Trainer에게 넘겨주어 대신 수행하도록 한다.

또한, 학습 과정을 보기 위해 학습 중 발생하는 손실과 정확도 등을 저장하여 그래프로 나타내는 메서드를 추가하였다.

Ⅲ. DeZero 프레임워크 사용 및 결과

구현한 DeZero 프레임워크를 사용하여, MNIST 데이터셋[5]을 이용한 손글씨 숫자 인식 문제를 해결한다.

```
train_set = dezero.datasets.MNIST(train=True)
test_set = dezero.datasets.MNIST(train=False)
train_loader = DataLoader(train_set, batch_size)
test_loader = DataLoader(test_set, batch_size, shuffle=False)

model = MLP((hidden_size, 10))
loss_func = F.softmax_cross_entropy
optimizer = optimizers.SGD().setup(model)

trainer = Trainer(model, loss_func, optimizer, train_loader,
                  test_loader, gpu=True)
trainer.fit(max_epoch)
trainer.plot()
```

그림 6 DeZero 프레임워크를 사용한 손글씨 숫자 인식 모델 작성

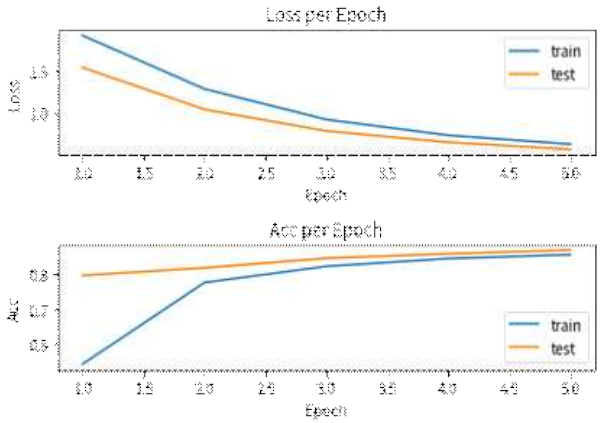


그림 7 epoch 별 손실과 정확도 그래프

DeZero 프레임워크를 사용하여 모델을 생성하고, GPU 상에서 모델을 학습시켰다.

Ⅳ. 결론 및 향후 연구 계획

Define-by-Run 방식의 DeZero 프레임워크를 직접 구현, 확장하며 딥러닝 프레임워크 (PyTorch, Tensorflow 등)의 구조를 이해할 수 있었으며, 프레임워크에서의 GPU 사용의 구현을 확인하였다.. 또한 딥러닝 네트워크에 사용된 이론과 구현 등에 대해서도 이해할 수 있었다.

현재의 DeZero 프레임워크를 바탕으로, DeZero에는 구현되어 있지 않으나, Chainer나 PyTorch 에 이미 구현된 함수나 계층, 모델 등을 구현하여 DeZero 프레임워크를 확장하고자 한다.

참고문헌

[1] Tokui, Seiya; et al. (2015). "Chainer: a next-generation open source framework for deep learning". 29th Annual Conference on Neural Information Processing Systems (NIPS). 5.

[2] "Eager Execution: An imperative, define-by-run interface to TensorFlow". Google Research Blog.

[3] Perez, Carlos E. (20 January 2017). "PyTorch, Dynamic Computational Graphs and Modular Deep Learning". Medium.

[4] Hido, Shohei (8 November 2016). "Complex neural networks made easy by Chainer". O'Reilly Media. Retrieved 26 June 2018.

[5] http://yann.lecun.com/exdb/mnist/