Queues, Threads, and Reading Data

The Input Pipeline

MNIST 이미지처럼 메모리에 올릴 수 있는 작은 데이터를 다룰 때는 모든 데이터를 메모리에 올리고 tensorflow 그래프에 데이터를 밀어 넣는 방식이 합리적이다. 하지만 데이터가 좀 더 클 때는 이러한 방법이 거추장스러워진다. 이러한 경우를 처리하는 자연스러운 패러다임은 데이터를 디스크에 보관하고 필요에 따라 데이터 덩어리를 로드하는 것이다.

전형적인 데이터 파이프라인은 서로 다른 종류의 입력 파일을 읽거나 입력의 형태나 구조를 바꾸거나 다른 형태의 전처리를 정규화하거나 입력을 셔플링하는 단계를 포함하는 경우가 많다.

이 과정의 많은 부분은 분리하여 모듈로 쪼갤 수 있다. 전처리의 예를 들면 전처리는 학습에 관여하지 않으므로 입력을 그대로 한꺼번에 전처리해서 학습으로 넘길 수 있다. 데이터를 배치로 학습시키기 때문에 원칙적으로는 디스크에서 데이터를 읽고, 전처리한 후 학습의 연산 그래프에 밀어 넣는 일괄 처리로 데이터를 다룰 수 있다.

하지만 이 방법은 효율적이지 않다. 전처리는 학습과 무관한데 전처리할 각 일괄 작업을 기다리느라 심각한 I/O 지연이 발생하고 각 학습 단계에서 로딩하여 처리될 데이터의 미니배치를 하염없이 기다리게 된다. 확장성 있는 접근법 중 하나는 데이터를 미리 가져와서 데이터의 로딩/처리와 학습에 별도의 스레드를 사용하는 것이다.

전처리를 염두에 두지 않더라도 지금까지 사용해온 표준 피딩 메커니즘을 사용하는 것은 그 자체로 낭비라는 사실에 주목할 필요가 있다. feed\_dict는 파이썬 런타임에서 tensorflow 런타임으로 데이터를 단일 스레드로 복사하는데, 이로 인해 지연이 발생하고 속도가 느려진다.

tensorflow에는 이 작업을 더 쉽고 더 빠르게 만들어주는, 입력 파이프라인 프로세스를 간소화할 수 있는 일련의 도구가 있다. 주요 구성 요소는 tensorflow의 표준 파일 형식과 이 형식을 인코딩/디코딩할 수 있는 유틸리티, 데이터 큐와 멀티스레드이다.

TFRecord

데이터셋의 형식은 매우 다양하며 때로는 혼재되어 있을 수도 있다. 원래의 형식이 무엇이든 간에 입력 파일들을 하나의 통합된 형식으로 변환하는 것이 편리하고 유용하다. tensorflow에서 기본 데이터 형식은 TFRecord이다. TFRecord 파일은 이진 파일로서 직렬화된 입력 데이터가 담겨 있다. 직렬화는 프로토콜 버퍼(Protocol Buffers)에 기반을 두고 있는데, 간단히 말해 프로토콜 버퍼는 데이터의 구조를 설명하는 스키마를 사용해 데이터를 저장용으로 변환하는 역할을 한다. 프로토콜 버퍼에서 저장하는 데이터는 XML과 비슷하게 사용하는 플랫폼이나 개발 언어와는 무관하다.

tensorflow 환경에서는 원래의 데이터 파일을 사용하는 것에 비해 TFRecord를 사용하는 것이 여러 가지 면에서 유리하다. 이 통합 형식을 사용하면 입력 인스턴스의 모든 관련 속성을 유지하면서도 많은 디렉터리나 하위 디렉터리가 필요하지 않도록 입력 데이터를 깔끔하게 정리할 수 있다. TFRecord 파일은 처리 속도도 매우 빠르다. 모든 데이터는 메모리의 하나의 블록에 저장되므로, 입력 파일이 개별로 저장된 입력 파일에 비해 메모리에서 데이터를 읽는 데 필요한 시간이 단축된다.

Writing with TFRecordWriter

먼저 입력 파일을 TFRecord 형식으로 작성하여 이를 사용할 수 있게 만들자. 이 예제에서는 MNIST 이미지를 이 형식으로 바꿀 것이다. 같은 방법으로 다른 유형의 데이터도 다룰 수 있다.

우선 tensorflow.contrib.learn의 유틸리티 함수를 사용하여 MNIST 데이터를 save\_dir에 내려받는다.(tfrecords\_read\_write.py)

내려받은 데이터에는 학습, 테스트, 유효성 검증 이미지가 각각 분리되어 들어 있다. 이를 스플릿이라고 부르고, 각 스플릿에 대해 데이터를 적당한 형식으로 만들어 TFRecordWriter()를 사용해 디스크에 저장한다.

먼저 TFRecordWriter 객체를 인스턴스화할 때 각 데이터 스플릿에 해당하는 경로를 지정했다. 그다음, 각 이미지에 대해서 넘파이 배열의 값을 바이트 스트링으로 변환한다. 다음에는 이미지를 프로토콜 버퍼 형식으로 변환한다. tf.train.Example이 예제 데이터를 저장하는 자료구조이다. Example 객체는 Features 객체를 포함하는데, Features 객체는 속성 이름에서 Feature로의 맵을 포함한다. Feature는 하나의 Int64List나 BytesList 또는 FloatList를 포함할 수 있다. 예를 들어 다음은 이미지의 레이블을 인코딩한다.

tf.train.Feature(int64\_list=tf.train.Int64List(value=[int(data\_set.labels[index])]))

그리고 다음은 실제 원본 이미지를 인코딩한다.

tf.train.Feature(bytes\_list=tf.train.BytesList(value=[image]))

저장된 데이터가 어떻게 생겼는지를 살펴보자. tf.python\_io.tf\_record\_iterator를 사용하면 된다. TFRecord 파일에서 레코드를 읽을 수 있는 이터레이터이다.

serialized\_img는 바이트 스트링이다. TFRecord에 이미지를 저장할 때 사용했던 구조체를 복구하려면 바이트 스트링을 파싱하면 된다. 그러면 앞에서 저장한 모든 속성에 접근할 수 있다.

이미지도 바이트 스트링으로 저장되었으므로 이를 넘파이 배열로 되돌려 (28, 28, 1)의 형태를 가진 tensor로 다시 바꾼다.

Queues

tensorflow 큐는 일반적인 큐와 비슷하게 새 항목을 큐에 넣거나 큐 내의 항목을 꺼내는 등의 일을 할 수 있다. 일반적인 큐와의 가장 큰 차이점은 tensorflow의 다른 구성 요소처럼 큐 역시 연산 그래프의 일부라는 것이다. 다른 노드처럼 큐 연산도 심벌일 뿐이며 그래프의 다른 노드가 큐의 상태를 변경할 수 있다.

Enqueuing and Dequeuing

여기서는 최대 10개의 항목을 넣을 수 있는 스트링의 선입선출(FIFO) 큐를 만든다. 큐는 연산 그래프의 일부이므로 세션 안에서 수행된다. 이 예제에서는 tf.InteractiveSession()을 사용한다.

(queue\_basic.py)

처음 시점에서, tensorflow는 내부적으로 10개의 항목을 저장하기 위한 메모리 버퍼를 생성한다. tensorflow의 다른 연산과 마찬가지로 큐에 항목을 추가하기 위해서 연산을 생성한다.

Multithreadings

tensorflofw 세션은 멀티스레드로 실행된다. 여러 개의 스레드가 같은 세션을 사용하여 병렬로 연산을 실행한다. 개별 연산은 여러 개의 CPU 코어나 GPU 스레드를 사용하도록 기본적으로 병렬 구현이 되어 있다. 하지만 sess.run()함수의 단일 호출이 가용한 모든 자원을 사용하지 않는다면 여러 개의 병렬 호출을 사용해서 처리량을 증가할 수 있다. 여러 스레드를 사용해 이미지를 전처리하고 큐에 밀어 넣으면 다른 스레드가 학습을 실행하기 위해 큐에서 전처리된 이미지를 가져가는 형태의 전형적인 시나리오를 예로 들 수 있다.

몇 개의 간단한 예제를 통해서 tensorflow의 스레딩 및 스레드와 큐 사이의 자연스러운 상호작용을 살펴본 후 MNIST 이미지를 사용한 완전한 예제까지 연결해보자.(ex\_Multithreadings.py)

우선 100개의 항목을 가진 FIFO 큐를 생성한다. 큐에 넣을 각 항목은 tf.random\_normal()로 만들어진 임의의 부동소수점 값이다.

enque 연산은 실제 그래프 수행 전에는 큐에 임의의 숫자를 더하지 않는다 sess.run()을 여러번 호출하는 방식으로 큐에 10개의 아이템을 더하는 함수 add()를 사용해 큐에 항목을 추가한다. 이어서 10개의 스레드를 생성하는데 각 스레드가 add()를 병렬로 수행하며, 각 add()는 10개의 항목을 큐에 동기화되지 않은 상태로 넣는다. 이들 임의의 숫자를 대기열에 추가되는 학습 데이터로 생각할 수 있다.

다음은 스레드의 리스트를 만들어 이것을 실행하며 큐의 크기가 100까지 증가할 때까지 짧은 간격으로 큐의 크기를 출력하는 코드다.

마지막으로 dequeue\_many()를 사용해 한 번에 10개 항목을 꺼내서 출력해보자.

Coordinator and QueueRunner

현실적인 시나리오에서 멀티스레드를 효과적으로 실행하는 것은 더 복잡할 수 있다. 스레드는 정상적으로 종료될 수 있어야 하고 스레드가 중단된 후에는 큐가 닫혀야 하며, 그 밖에도 해결해야 할 여러 중요한 기술적 문제가 있다.

tensorflow는 이 과정에서 도움이 될 만한 도구를 갖추고 있다. 그 중에서 핵심은 스레드 셋의 종료를 조정하는 tf.train.Coordinator와 원활한 협조 형태로 데이터를 큐에 넣을 수 있도록 여러 개의 스레드를 얻어오는 과정을 간소화하는 tf.train.QueueRunner이다.

tf.train.Coordinator

일단 tf.train.Coordinator의 사용법을 간단한 예제로 설명한 후 다음 절에서 이를 실제 입력 파이프라인의 일부로 사용하는 방법을 살펴볼 것이다.

(ex\_tf.train.Coordinator.py)

어떤 스레드건 coord.request\_stop()을 호출해서 다른 모든 스레드를 중단할 수 있다. 스레드는 일반적으로 coord.should\_stop()을 사용해서 중단할지의 여부를 확인하는 루프를 실행한다. 여기서는 add()에 스레드의 인덱스 i를 전달하고 절대로 충족되지 않는 조건(i==11)을 중단을 요청하는 조건으로 사용했다. 따라서 예제의 스레드는 전체 100개의 항목을 큐에 추가하는 작업을 완료한다. 하지만 add()를 (if i == 1:)로 변경한다면 인덱스가 1인 스레드는 코디네이터를 통해 모든 스레드가 중지하도록 요청한다. 따라서 큐에 넣는 작업이 조기에 종료된다.

tf.train.QueueRunner and tf.RandomShuffleQueue

큐에 넣은 연산을 반복적으로 수행하는 다수의 스레드를 생성할 수도 있지만, 이보다는 기본 제공되는 tf.train.QueueRunner를 사용하는 것이 더 좋다. 완전히 동일한 일을 하면서도 예외 상황에서는 큐를 닫아준다. (ex\_tf.train.QueueRunner.py)

다음 예제에서는 항목을 큐에 넣는 4개의 스레드를 병렬로 실행하는 큐 실행자를 생성한다.

qr.create\_threads()는 코디네이터와 함께 세션을 인수로 사용한다.

이 예제에서는 FIFO 큐 대신 tf.RandomShuffleQueue를 사용했다. RandomShuffleQueue는 항목을 꺼낼 때 순서가 무작위인 큐이다. 이 큐는 데이터의 순서를 섞을 필요가 있는 확률적 경사 하강법(SGD) 최적화 함수를 사용해 심층신경망을 학습시킬 때 유용하다. mni\_after\_dequeue 인수는 항목 꺼내기 연산을 호출할 후 큐에 남아 있을 항목의 최소 개수를 지정한다. 이 값이 크면 더 잘 섞이지만, 더 많은 메모리가 필요해진다.

A Full Multithreaded Input Pipeline

이제 데이터를 로딩하고 전처리하는 과정에서 tensorflow에서 제공하는 효율적인 파일 형식에 데이터를 써보는 것부터 모델을 학습시키는 것까지의 모든 조각을 모아서 MNIST 이미즈를 사용한 예제를 동작시켜보다, 앞에서 설명한 큐와 멀티스레딩 기능을 사용해서 구축할 것이다. 이때 tensorflow에서 데이터를 읽고 처리하는 데 도움이 되는 구성 요소 몇 개를 추가로 소개한다. (tfrecords\_end\_to\_end.py)

TFRecord에 MNIST 데이터를 쓰는 것부터 시작하자.

tf.train.string\_input\_producer() and tf.TFRecordReader()

tf.train.string\_input\_producer()는 내부에서 QueueRunner를 생성해 입력 파이프라인을 구성하기 위한 파일명 스트링들을 큐에 출력한다. 이 파일명 큐는 여러 스레드가 공유할 것이다.

num\_epochs 인수는 string\_input\_producer()에 각 파일명 스트링을 num\_epochs 횟수만큼 생성하도록 지시한다.

그 다음 TFRecordReader()를 사용해 큐에서 파일명을 읽는다. 즉 파일명의 큐를 가져와 filename\_queue의 파일명을 사용해 파일명을 큐에서 꺼낸다. 내부적으로 TFRecordReader()는 그래프의 상태를 이용해 디스크에서 입력 데이터의 ‘지금까지 읽은 덩어리 이후의 다음 덩어리’를 로드할 때 읽어야 할 TFRecord의 위치를 추적한다.

tf.train.shuffle\_batch()

원본 바이트 스트링 데이터를 디코딩하고 픽셀 값을 부동소수점 값으로 변환하는 가장 기본적인 전처리를 수행한다. 그다음 이미지 인스턴스를 임의로 셔플하고 tf.train.shuffle\_batch()를 사용해 이를 batch\_size만큼의 배치 데이터로 모은다. tf.train.shuffle\_batch()는 내부적으로 RandomShuffleQueue를 사용해 이 큐에 batch\_size + min\_after\_dequeue만큼의 항목이 쌓일 때까지 인스턴스를 쌓는다.

capacity와 min\_after\_dequeue 매개변수는 앞에서 설명한 것과 같은 방식으로 사용된다. shuffle\_batch()에서 반환하는 미니배치는 내부에서 생성된 RandomShuffleQueue에 dequeue\_many() 함수를 호출해서 만들어진다.

tf.train.start\_queue\_runners() and Wrapping Up

간단한 소프트맥수 분류 모델을 정의한다.

마지막으로 tf.train.start\_queue\_runners()를 호출해 큐에 데이터를 넣는 스레드를 만든다. 다른 호출과 달리 이 호출은 심벌에 그치지 않고 실제로 스레드를 생성한다.

모든 작업이 완료되었으므로 배치 데이터를 읽어 들여 전처리해서 큐에 넣는 것부터 모델을 학습시키는 것까지 멀티스레드 프로세스를 실행할 준비가 끝났다. 앞에서 계속 보아온 feed\_dict 인수를 사용하지 않는다는 점에 주목하자. 데이터 복사를 하지 않으므로 속도가 향상된다.

tf.errors.OutOfRangeError 오류가 발생할 때까지 학습하는데, 이 오류가 발생했다는 것은 큐가 비어서 작업이 완료되었음을 뜻한다.