
NeMo: 다음을 사용하여 AI 애플리케이션을 구축하기 위한 툴킷 신경 모듈

Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Krizan, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, Patrice Castonguay, Mariya Popova, Jocelyn Huang, Jonathan M. Cohen

NVIDIA

Santa Clara, CA

{okuchaiev,jasoli,chipn,ohrinchuk,rleary,bginsburg,skrizan,stanislav,
vlavrukhin,jocook,pcastonguay,mpopova,jocelynh,jocohen}@nvidia.com

추상적인

NeMo (Neural Modules)는 재사용성, 추상화 및 구성을 통해 AI 애플리케이션을 생성하기 위한 Python 프레임워크에 구매받지 않는 툴킷입니다. NeMo는 입력된 입력을 받아 입력된 출력을 생성하는 신경망의 개념적 블록인 신경 모듈을 중심으로 구축되었습니다. 이러한 모듈은 일반적으로 데이터 계층, 인코더, 디코더, 언어 모델, 손실 함수 또는 활성화 결합 방법을 나타냅니다.

NeMo를 사용하면 신경 유형 시스템을 통해 의미론적 정확성 검사 수준을 제공하면서 이러한 구성 요소를 쉽게 결합하고 재사용할 수 있습니다. 이 툴킷은 자동 음성 인식 및 자연어 처리를 위해 사전 구축된 모듈의 확장 가능한 컬렉션과 함께 제공됩니다. 또한 NeMo는 최신 NVIDIA GPU에서 분산 교육 및 혼합 정밀도를 기본적으로 지원합니다. NeMo는 오픈 소스입니다.¹

1. 소개

딥 러닝(DL)은 지난 10년 동안 학계에서 산업계로 엄청난 발전을 이루었습니다. 그러나 DL 소프트웨어를 개발, 디버깅 및 배포하는 프로세스는 다른 복잡한 소프트웨어 시스템보다 훨씬 번거롭습니다. 모든 DL 프레임워크의 기본 추상화는 다차원 텐서이며, 일반적으로 차원 의미 체계(예: 첫 번째 차원이 배치 크기 또는 다른 것을 나타내는지 여부)가 없습니다. 의미론과 유형 시스템의 부족은 모델의 재사용을 복잡하게 만들고 DL 시스템을 구축하기 어렵게 만듭니다 [6]. 다양한 사용 사례 또는 개발자 간에 복잡한 DL 모델의 구성 요소를 재사용하는 것은 어려울 수 있습니다. 구성 요소를 재사용하고 공유하는 일반적인 접근 방식은 미리 훈련된 오픈 소스 모델을 기반으로 합니다. 이러한 모델을 함께 결합하고 연결하려면 일반적으로 코드를 변경해야 하며 이를 위해서는 디버깅이 필요합니다. 이것은 모델이 다른 개발자나 사용 사례에서 나온 경우 특히 까다롭습니다.

또 다른 복잡한 요소는 모델 구성이 일반적으로 데이터 모델 대신 Python 스크립트를 통해 정의된다는 것입니다. 이로 인해 계산 성능, 아키텍처 정의, 교육 절차, 시각화 및 분석과 같이 별도의 문제가 되는 것 사이의 경계가 모호해지며 모두 동일한 Python 스크립트로 혼합되어 다른 컨텍스트에서 풀기, 디버그 또는 재사용하기 어렵습니다.

문제의 분리, 잘 정의된 검증 가능한 인터페이스를 사용한 시스템 분해, 코드 재사용 가능성과 같은 이러한 모든 문제는 이미 소프트웨어 엔지니어링 세계에서 잘 탐구되었습니다.

모든 소프트웨어 개발자가 당연하게 여기는 많은 최신 기술은 원래 이러한 문제를 정확하게 해결하기 위해 발명되었습니다.

1사용 가능: <https://github.com/NVIDIA/NeMo>

사전 인쇄 검토 중입니다.

우리는 이러한 문제를 해결하기 위해 개발된 일반적인 소프트웨어 엔지니어링 관행을 AI 기반 애플리케이션 개발의 맥락으로 변환하고자 합니다. 특히 다음과 같은 문제에 중점을 둡니다.

- 복잡한 시스템을 잘 정의된 인터페이스가 있는 기능 블록으로 분해합니다 . • API 준수를 보장하고 유형 불일치 버그를 잡기 위한 정적 유형 검사 ;
- 모델 아키텍처, 교육 절차, DL 프레임워크 , 최적화 알고리즘 ;
- 현대적이고 효율적인 하드웨어 기능을 지원하여 고성능 교육 ; 그리고
- 새로운 방식으로 쉽게 결합할 수 있는 재사용 가능한 사전 제작 구성 요소 .

NeMo는 다음으로 구성됩니다. (1) NeMo Core: 모든 신경 모델 및 유형 시스템을 위한 기본 구성 요소 및 (2) NeMo 컬렉션: 자동 음성 인식(ASR) 및 자연어 처리(NLP)와 같은 특정 도메인을 위해 사전 구축된 신경 모듈).

2 관련 업무

최근 몇 년 동안 TensorFlow[8] 및 PyTorch[17]와 같은 DL 프레임워크를 순수하게 사용하여 사용자가 특정 목표를 보다 쉽게 달성할 수 있도록 도와주는 높은 수준의 툴킷이 많이 있습니다 .

이러한 툴킷 은 느슨하게 두 가지 주요 그룹 으로 분류 할 수 있습니다 . Tensor2Tensor [20], Ludwig [5], OpenSeq2Seq [13], FairSeq [16], OpenNMT [12], Seq2Seq[9] 등이 있습니다.

개념적으로 NeMo Core는 첫 번째 그룹에 더 가깝습니다. 이를 통해 사용자는 임의의 구성 요소 세트로 모델을 표현할 수 있으며 많은 유연성을 유지하면서 교육 및 평가 루프의 세부 정보를 숨길 수 있습니다 . 반면에 NeMo 컬렉션은 두 번째 그룹에 더 가깝습니다. 여기에는 다양한 시나리오에서 재사용할 수 있는 공통 모듈이 포함되어 있습니다. 예를 들어 NeMo에서는 인코더-디코더 네트워크와 같은 고정 패턴에 대한 템플릿을 정의하는 것이 간단합니다 .

NeMo는 두 가지 면에서 첫 번째 그룹의 툴킷과 다릅니다. (1) NeMo의 핵심 추상화는 계층이나 텐서가 아닌 신경 모듈이며 (2) NeMo에는 다양한 의미 검사를 수행하는 신경 유형 시스템이 포함되어 있습니다 .

NeMo와 두 번째 그룹의 툴킷 사이의 주요 차이점은 NeMo가 특정 구조를 부과하지 않는다는 것입니다 . 예를 들어 많은 툴킷은 모델이 인코더-디코더-손실 구조를 따르도록 요구합니다.

NeMo는 또한 구성 파일이 특정 형식일 필요가 없습니다. 사용자는 NeMo API를 사용하여 직접 모델을 정의할 수 있습니다. 두 번째 그룹의 일부 툴킷은 블록[1] 간의 입출력 호환성을 강화 하지만 NeMo는 일관되고 일반적이며 확장 가능한 유형 시스템을 사용하여 이를 수행합니다.

개념적으로 NeMo 는 작업 지향 접근 방식을 따르는 PyTorchPipe[1]와 유사 하지만 임의의 유연한 구성 요소 집합을 허용하고 호환성 검사도 수행합니다. 본질적으로 응용 프로그램 프레임워크 인 반면 NeMo는 개발자가 기본 유형 및 구성 시스템을 사용할 수 있지만 NeMo 런타임 기능을 채택할 수는 없습니다.

3 니모

NeMo의 핵심 빌딩 블록은 Neural Module(NM)이라고 합니다. 신경 결절은 언어 모델, 인코더, 디코더, 데이터 증대 알고리즘, 손실 함수 또는 기타 계층 및 함수 집합과 같은 신경망의 논리적 부분을 나타냅니다. NeMo의 기본 추상화로서 NM은 모델을 설명하고 해당 모델이 훈련되는 프로세스를 설명하기 위한 기초를 형성합니다. 공식적으로 Neural Module은 유형이 지정된 입력 집합이 주어지면 유형이 지정된 출력 집합을 계산하는 구성 요소입니다.

입력 및 출력은 다차원 텐서의 모음입니다. 객체 지향 언어 의 프로그래머가 객체를 정의할 세분성 수준을 선택할 수 있는 것과 같은 방식으로 NeMo 사용자는 Neural Module의 세분성 수준을 선택할 수 있습니다. 기본 규칙은 입력과 출력이 인터페이스를 통해 노출되도록 "합리적"이어야 한다는 것입니다. 이것은 Neural Module이 일반적으로 단일 신경망 계층이 아니라 인코더, 연결 연산, 손실 함수 또는 데이터 증대과 같은 "유용한 작업을 수행하는" 연결된 계층의 모음을 시사합니다.

구현에서 NM은 아래에 설명된 유형 시스템을 사용하여 입력 포트와 출력 포트를 설명하는 Python 클래스입니다. 현재 구현은 PyTorch에 의존하지만 추상화 및 코드는 기본 프레임워크를 참조하지 않으므로 애플리케이션이 프레임워크에 구애받지 않고 향후 새로운 백엔드를 추가할 수 있습니다(그림 1 참조).

NM은 제공된 입력 포트 값이 주어지면 출력 포트 값을 계산할 수 있습니다. 훈련 가능한 가중치를 포함하는 NM의 경우 기울기 흐름도 계산할 수 있어야 합니다. 순방향 및 역방향 패스의 구현은 기본 DL 프레임워크에서 제공됩니다. NM을 생각하는 한 가지 방법은 다른 NM 세트(재귀적으로), PyTorch에서 구현된 잘 정의된 신경망 계층 세트 또는 훈련할 수 없는 NM의 경우, 입력 값이 주어진 출력 값을 직접 평가합니다.

NM은 매개변수화될 수 있습니다. 예를 들어 이미지 인코더 NM은 컨벌루션 레이어 수, 필터 크기, 드롭아웃 값 등으로 매개변수화될 수 있습니다. 이러한 방식으로 NM은 변수가 매개변수 값에 의해 명시적으로 결정되는 신경망의 매개변수 계열을 정의합니다. 매개변수는 명명된 매개변수를 통해 생성 시 NM에 전달되며, 낮추는 방법을 정의하는 코드는 이러한 매개변수의 생성자 시간 값에만 의존합니다 2.

기능적 프로그래밍 언어와 마찬가지로 NM의 출력 또는 기울기 평가는 표준 신경망 순방향 및 역방향 전파 규칙을 따르는 명시적으로 연결된 입력 및 출력을 통하는 경우를 제외하고는 다른 NM의 입력 또는 기울기 평가에 영향을 줄 수 없습니다. 이 분리 평가에 부작용이 있을 수 없기 때문에 잘 정의된 인터페이스에 따라 복잡한 모델을 명확하고 올바르게 분해하는 데 도움이 됩니다. 또한 명시적으로 정의된 매개변수를 통해 실험 추적 및 하이퍼 매개변수 검색 도구와의 통합을 크게 단순화할 수 있습니다.

이것은 NeMo의 주요 원칙을 보여줍니다. 신경망의 구조와 그것의 정방향 및 역방향 데이터 흐름은 Python 소스 코드에 인코딩된 논리가 아니라 데이터 구조의 값에 의해 결정되어야 합니다.

3.1 니모 코어

NeMo로 구축된 애플리케이션은 일반적으로 3개의 필수 단계와 1개의 선택 단계로 구성됩니다.

1. NeuralFactory 객체와 필요한 NM을 인스턴스화합니다.
2. NM을 함께 연결하여 활성화 흐름 DAG를 정의합니다. 3. (선택 사항) 로깅, 체크포인트, 시각화 및 평가를 위한 콜백을 정의합니다.
4. 학습, 평가 또는 추론과 같은 작업을 호출합니다.

NeMo는 지연 실행 모델을 따릅니다. 작업이 호출될 때까지 계산이 수행되지 않습니다. 활성화 흐름 방향성 비순환 그래프(DAG)를 정의하는 동안 NeMo는 연결된 NM의 입력 및 출력에 대한 유형 검사를 수행합니다. 이는 계산을 수행하기 전에 다양한 오류를 포착하고 디버깅하는 데 도움이 됩니다. 모듈의 DAG가 정의되고 작업이 호출되면 NeMo는 우리가 백엔드라고 부르는 DL 프레임워크를 호출합니다. NeMo는 프레임워크에 구애받지 않도록 설계되었지만 현재 백엔드로 PyTorch만 지원합니다.

사용자는 기존 NM을 결합하거나 특정 프레임워크에 구현을 제공하여 자신만의 NM을 만들 수 있습니다. 실제로 모든 PyTorch nn.Module은 입력 및 출력 포트 정의를 추가하여 NeMo의 NM으로 쉽게 변환할 수 있습니다.

scikit-learn 및 Keras와 유사하게 NeMo를 사용하면 평가, 로깅 및 성능 모니터링과 같은 훈련 중에 수행되는 루틴에 대한 콜백을 만들 수 있습니다.

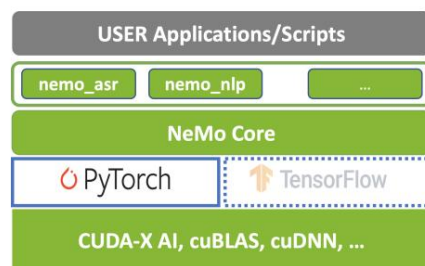


그림 1: NeMo는 애플리케이션과 DL 프레임워크 백엔드 간의 추상화 수준 역할을 하는 프레임워크에 구애받지 않는 툴킷입니다.

20는 현재 컨벤션을 통해 시행됩니다.

3.2 신경 유형 시스템

NeMo Core는 NeuralModule 기본 클래스의 인터페이스와 기능을 정의합니다. NM의 각 입력 및 출력에는 신경 유형이 있습니다. 신경 유형은 텐서의 시맨틱, 축 순서 및 차원을 설명합니다. 이 유형 시스템의 목적은 모델 생성 중에 의미 및 차원 오류를 포착하고 모듈 재사용을 용이하게 하는 것입니다. 한 NM 출력 포트의 출력 유형이 다른 NM 입력 포트의 입력 유형과 일치하는 경우, 이 두 NM을 어디에서 왔는지 또는 구현 방법에 관계없이 함께 연결하는 것이 합법적입니다. 정적 유형 검사를 사용하기 때문에 NeMo는 DAG 생성 단계에서 의미 체계 및 차원 오류를 포착할 수 있습니다.

신경 유형은 각 텐서 축 ID에서 축 유형으로 매핑됩니다. 축 유형에는 텐서 축의 의미 및 차원 정보가 포함됩니다. 시맨틱 정보는 "is-a" 종류의 상속과 관련된 파이썬 클래스인 "시맨틱 태그"의 도움으로 표현됩니다. 예를 들어, 모듈 A의 출력이 시맨틱 유형 WordEmbedding 이고 모듈 B의 예상 입력이 Embedding 이고 WordEmbedding이 Embedding 에서 상속된 경우 모듈 B는 A의 출력을 입력으로 허용할 수 있지만 그 반대는 허용되지 않습니다. 예를 보려면 표 3.2를 참조하십시오.

NeuralType 은 축 인덱스를 AxisType에 매핑하는 사전인 axis2type 에서 구성됩니다. 예를 들어 일반적인 ResNet 인코더의 입력 및 출력 포트는 다음과 같이 설명할 수 있습니다.

```
입력_포트 = { "x": NeuralType({ 0: AxisType( BatchTag ),
                                1: AxisType( ChannelTag ),
                                2: AxisType( 높이 태그, 224 ),
                                3: AxisType( WidthTag, 224 ) })}
출력_포트 = { "출력출": NeuralType({ 0: AxisType( BatchTag ),
                                       1: AxisType( ImageEmbeddingTag ) })}
```

NeMo는 SAME, LESS, GREATER, DIM_INCOMPATIBLE, TRANSPOSE_SAME 및 INCOMPATIBLE 과 같은 다양한 비교 결과가 있는 NeuralType 객체 쌍에 대한 이진 비교 작업을 정의합니다. 이 유형 시스템은 비텐서 객체(예: 스칼라)와 C++의 void*와 다소 유사한 루트 유형도 허용합니다. 루트 유형의 포트는 모든 텐서3를 허용할 수 있습니다. NeMo의 유형 시스템이 모델 정의에서 잡을 수 있는 오류의 예 time에는 "순위는 일치 하지만 의미는 일치하지 않음", "의미적으로 잘못된 차원을 따라 연결" 및 "차원 불일치"가 포함됩니다. 예를 들어, 디코더가 [batch_dim, time_dim 형식의 입력을 예상하는 인코더-디코더 모델을 고려하십시오. , channel_dim] 및 다른 개발자가 작성한 인코더는 [time_dim, batch_dim, channel_dim] 텐서를 출력합니다. 시간 및 배치 차원은 종종 동적이며 channel_dim이 일정하게 유지되면 표준 프레임워크는 원활하게 실행되지만 모델이 수렴되지 않습니다. 개발자가 이 자동 오류를 찾아 수정하도록 강제합니다. 그러나 NeMo는 이러한 모듈이 연결되는 순간 시맨틱 유형 오류를 발생시킵니다. 이 경우 유형 비교 작업의 결과는 대신 TRANSPOSE_SAME이 됩니다. 동일합니다.

4.

출력 포트 유형	입력 포트 유형	비교 결과
{0: 배치, 1: 채널}	{0: 배치, 1: 스펙트로그램} 더 큼(호환 불가)	
{0: 배치, 1: 스펙트로그램}	{0: 배치, 1: 채널}	덜 (호환 가능)
{0: 배치, 1: 스펙트로그램}	{0: 배치, 1: 인코딩됨}	호환 불가
{0: 배치, 1: 스펙트로그램}	{0: 스펙트로그램, 1: 배치}	TRANSPOSE_SAME
{0: 배치, 1: 스펙트로그램:64} {0: 배치, 1: 채널:40}		DIM_INCOMPATIBLE
{0: 배치, 1: 스펙트로그램:64} {}		같은

표 1: NeMo 신경 유형의 예. {}는 "루트" 신경 유형을 나타냅니다. 이러한 예는 다음을 가정합니다. (1) Spectrogram 및 Encoded 유형은 채널 유형에서 상속됩니다. (2) 모듈의 A 출력 포트는 모듈의 B 입력 포트에 연결됩니다.

¹ 앞으로는 연결과 같은 작업을 위한 형식 안전 제네릭을 지원하기 위해 단순화된 버전의 C++ 템플릿을 모델로 하는 템플릿 형식 시스템을 추가할 계획입니다.

⁴ 시스템이 다음과 같은 간단한 작업을 자동으로 삽입하는 암시적 캐스트를 추가할 수 있습니다. C++가 자동으로 int를 float로 승격시키는 것과 유사한 단순 유형 불일치를 "수정"하기 위해 전치합니다.

3.3 고성능 교육

NeMo는 NVIDIA의 Volta 및 Turing GPU와 같은 최신 DL 하드웨어를 최대한 활용하도록 제작되었습니다. 자동으로 혼합 정밀도 훈련을 지원하고, 일부 항목은 float32에 유지하면서 매트릭스 곱셈 및 컨벌루션과 같은 계산 집약적 작업에 float16을 사용 하고 동적 손실 스케일링을 사용합니다[15].

NeMo는 또한 작업자에 대한 기울기를 누적하고 특정 수의 배치가 처리된 후에만 가중치를 업데이트하는 기술인 기울기 누적을 지원합니다. 이것은 RAM이 제한된 카드에서 매우 큰 배치 시뮬레이션을 허용 하고 작업자 간 통신의 양을 줄임으로써 분산된 다중 GPU 실행을 용이하게 합니다. NeMo는 또한 NVIDIA의 APEX 라이브러리[2]를 사용하여 다중 GPU 및 다중 노드 교육을 지원합니다.

4 NeMo 컬렉션

NeMo 컬렉션은 관련 기능의 소프트웨어 컬렉션에 해당하는 DL입니다. 특정 도메인에 대한 공통 NM은 사전 구축되어 "컬렉션"으로 패키징됩니다. 현재 NeMo는 자동 음성 인식(ASR) 및 자연어 처리(NLP)용 컬렉션을 제공하지만 사용자는 새 컬렉션을 쉽게 추가할 수 있습니다. 이러한 NM 중 일부는 사전 훈련된 가중치와 함께 제공될 수 있습니다. NeMo 컬렉션은 관련 기능 컬렉션을 포함하는 소프트웨어 라이브러리와 동등한 DL입니다. 실제로 컬렉션은 NM 클래스, 신경 유형 및 관련 도우미 루틴을 정의하는 Python 모듈입니다.

4.1 자동 음성 인식

nemo_asr은 ASR(자동 음성 인식) 모델을 훈련하고 평가하는 데 사용할 수 있는 신경 모듈 및 도우미 함수 모음입니다. 현재 CTC 기반 및 sequence-to-sequence 어텐션 기반의 두 가지 모델 유형을 지원합니다.

4.1.1 CTC 기반 음성인식

예를 들어 nemo_asr을 사용하여 Jasper와 같은 ASR 모델[14]을 교육하는 데 필요한 단계를 설명합니다.

먼저 신경 모듈의 훈련 및 인스턴스화를 관리하는 Neural Module Factory 개체를 만듭니다. Jasper는 JasperEncoder, JasperDecoderForCTC 및 CTCLossNM 을 사용합니다. 이러한 각 NM은 생성 시 매개변수를 전달합니다. 여기서는 코드를 더 쉽게 읽을 수 있도록 매개변수를 생략합니다.

```
nf = 니모 데이터 . 핵심 . 신경 모듈 공장(...) 레이어 = nemo_a
    _ srjasperjasperloss = nemo_a . srjasperlossNM at a Layer(...)
    _ 인코더 = nemo_a sr . J asper 인코더(...)
    _ 디코더 = nemo_a sr . JasperDecode rF o rCTC(...)
CTC _
```

다음으로 활성화가 출력 포트에서 입력 포트로 흐르는 방식에 대한 DAG(Directed Acyclic Graph)를 정의합니다.

```
pec, 인코딩된 _ len = _ len = 데이터 _ 레이어() 신
probs = jasper_ asr _ decoder (인코더 _ 인코더(오디오 _ 호 = s pec, 길이 = spec _ len)
    _ output = e n code )
    _ 손실(log_probs = log_probsinput , targets = transcripttarget t ,
        _ 길이 = 인코딩된 _ len , _ 길이 = 성적표 _ len)
```

첫 번째 줄에서 data_layer는 튜플로 반환되는 4개의 출력 포트를 생성합니다. jasper_encoder에는 두 개의 data_layer 출력 포트에 연결된 두 개의 명명된 입력 포트가 있습니다.

이 DAG 신경망 유형 시스템을 정의하는 동안 다양한 모듈을 함께 올바르게 사용하는지 확인하기 위해 수행됩니다. 마지막으로 DAG가 설명되면 교육 절차 및 데이터 흐름을 트리거하는 훈련과 같은 신경 공장의 작업을 호출해야 합니다.

```
nf . 기차 ( tensorscallbacks _ 에게 _ 최적화 = [ 손실 ], cb ),
    _ cb , _ ...)
```


참조

- [1] 파이토치파이프. <https://github.com/ibm/pytorchpipe>. 접속일: 2019-08-19.
- [2] pytorch 확장: pytorch에서 쉽게 혼합 정밀도 및 분산 교육을 위한 도구입니다. <https://github.com/NVIDIA/apex>. 접속일: 2019-08-19.
- [3] 파이토치가 점화됩니다. <https://pytorch.org/ignite/>. 접속일: 2019-08-19.
- [4] 파이토치 번개. <https://github.com/williamFalcon/pytorch-lightning>. 접속일 : 2019-08-19.
- [5] 루드비히. <http://ludwig.ai>. 접속일: 2019-08-19.
- [6] Tensor는 유해한 것으로 간주됩니다. <http://nlp.seas.harvard.edu/NamedTensor>. 액세스: 2019-08-19.
- [7] huggingface의 pytorch-transformers. <https://github.com/huggingface/>
파이토치 트랜스포머. 접속일: 2019-08-19.
- [8] 마틴 아바디, 아시시 아가르왈, 폴 바햄, 유진 브레브도, 지평 첸, 크레이그 시트로, 그렉 S. 코라도, 앤디 데이비스, 제프리 딘, 마티유 데빈, 산제이 게마와트, 이안 굿펠로우, 앤드류 하프, 제프리 어빙, 마이클 이사드, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu 및 Xiaoqiang Zheng. TensorFlow: 이기종 시스템에서 대규모 기계 학습, 2015. URL <https://www.tensorflow.org/>. tensorflow.org에서 사용 가능한 소프트웨어.
- [9] Denny Britz, Anna Goldie, Thang Luong 및 Quoc Le. 신경 기계의 대규모 탐색 번역 아키텍처. ArXiv e-프린트, 2017년 3월. _____
- [10] William Chan, Navdeep Jaitly, Quoc Le 및 Oriol Vinyals. 듣기, 참석 및 철자: 대규모 어휘 대화 음성 인식을 위한 신경망. 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 페이지 4960–4964. IEEE, 2016. _____
- [11] François Chollet 외. 케라스. <https://keras.io>, 2015.
- [12] Guillaume Klein, 김윤, Yuntian Deng, Jean Senellart, Alexander Rush. OpenNMT: 신경망 기계 번역을 위한 오픈 소스 툴킷. ACL 2017 절차, 시스템 데모, 67–72페이지, 캐나다 밴쿠버, 2017년 7월. 전산 언어학 협회. URL <https://www.aclweb.org/anthology/P17-4012>.
- [13] Oleksii Kuchaiev, Boris Ginsburg, Igor Gitman, Vitaly Lavrukhin, Carl Case, Paulius Micikevicius. Openseq2seq: sequence-to-sequence 모델의 분산 및 혼합 정밀도 훈련을 위한 확장 가능한 툴킷. arXiv e-prints arXiv:1805.10387, 2018. _____
- [14] J. Li, V. Lavrukhin, B. Ginsburg, R. Leary, O. Kuchaiev, JM Cohen, H. Nguyen 및 RT Gaddei. Jasper: 엔드-투-엔드 컨볼루션 신경 음향 모델. arXiv e-prints arXiv:1904.03288, 2019. _____
- [15] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, Hao Wu. 혼합 정밀 훈련. ICLR, 2017. _____
- [16] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, Michael Auli. fairseq: 시퀀스 모델링을 위한 빠르고 확장 가능한 툴킷. NAACL-HLT 2019 진행 : 데모, 2019. _____
- [17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga 및 Adam Lerer. pytorch 에서 자동 차별화 . 2017.

- [18] Malcolm Reynolds, Gabriel Barth-Maron, Frederic Besse, Diego de Las Casas, Andreas Fjeld, Tim Green, Adrià Puigdomènech, Sébastien Racanière, Jack Rae, Fabio Viola. 오픈 소싱 Sonnet - 신경망 구축을 위한 새로운 라이브러리. <https://deepmind.com/blog/open-sourcing-sonnet/>, 2017.
- [19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser 및 Illia Polosukhin. 주의가 필요합니다. arXiv 프리프린트 arXiv: 1706.03762, 2017.
- [20] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Lukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer 및 Jakob Uszkoreit. 신경망 기계 번역을 위한 Tensor2tensor. CoRR, abs/1803.07416, 2018. URL <http://arxiv.org/abs/1803.07416>.