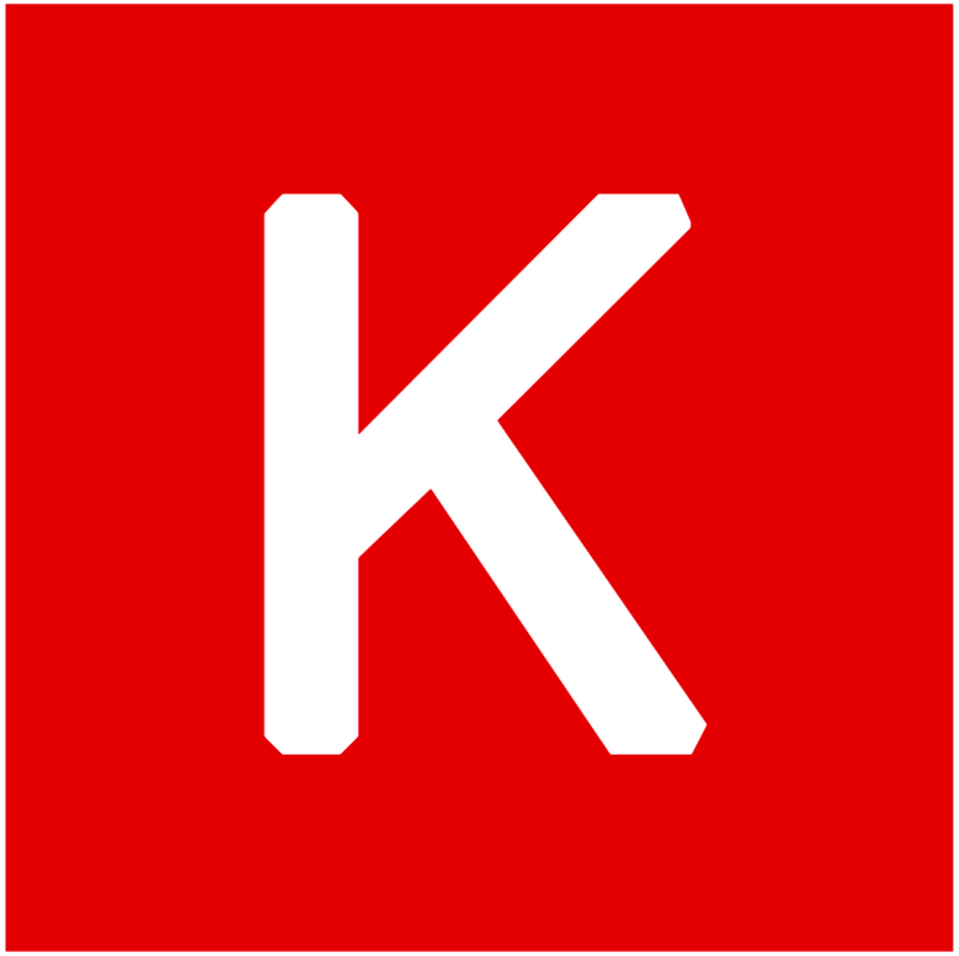
KERAS

MAY 2023
30515 원종우



케라스란?

케라스는 딥러닝 모델을 만들기 위한 고수준의 구성 요소를 제공하는 모델 수준의 라이브러리입니다. 현재는 텐서플로, 씨아노, 마이크로소프트 코그니티브 툴킷 3개를 백엔드 엔진으로 사용할 수 있습니다. 케라스는 MIT 라이선스를 따르므로 상업적인 프로젝트에도 자유롭게 사용할 수 있습니다.

다음과 같은 장점이 있습니다

사용자 친화적인 인터페이스

Keras는 사용자 친화적인 API를 제공하여 딥러닝 모델을 쉽게 구축할 수 있습니다. 간결하고 직관적인 문법을 사용하여 모델의 층(layer)을 쌓고 구성할 수 있으며, 일반적인 딥러닝 작업을 위한 다양한 유틸리티 함수와 모듈을 포함하고 있습니다.

다양한 백엔드 엔진 지원

Keras는 백엔드 엔진을 통해 다양한 하드웨어와 소프트웨어 환경에서 작동할 수 있도록 지원합니다. TensorFlow, Theano, Microsoft Cognitive Toolkit (CNTK), PlaidML 등의 백엔드를 선택하여 사용할 수 있습니다.

확장성과 커뮤니티 지원

Keras는 딥러닝 생태계에서 많은 인기를 얻고 있으며, 확장성과 커뮤니티 지원 측면에서 강력합니다. 또한, Keras는 많은 개발자와 연구자들이 사용하는 인기 있는 프레임워크이므로, 커뮤니티에서는 다양한 튜토리얼, 예제 코드, 논의, 지원 등을 제공하여 개발자들이 서로 정보를 공유하고 협력할 수 있는 장점이 있습니다.

다음과 같은 단점이 있습니다

01

고도로 커스터마이즈된 모델 구성의 한계

Keras는 사용하기 쉬운 인터페이스를 제공하지만, 높은 수준의 커스터마이즈가 필요한 경우에는 제한적일 수 있습니다. 특정 독특한 아키텍처나 사용자 정의 레이어와 같은 고급 기능을 구현하기 위해서는 TensorFlow나 PyTorch와 같은 더 낮은 수준의 프레임워크를 사용하는 것이 더 적합할 수 있습니다.

02

백엔드 종속성

Keras는 다양한 백엔드 엔진을 지원하지만, 백엔드 선택은 초기에 결정되고 변경하기 어렵습니다. 프로젝트를 시작할 때 선택한 백엔드에 의존하게 되며, 다른 백엔드로 전환하기 위해서는 모델을 다시 작성해야 할 수도 있습니다.

03

낮은 수준의 디버깅 기능

Keras는 딥러닝 모델 개발을 단순화하기 위해 고수준의 추상화를 제공합니다. 하지만 이로 인해 디버깅이 어려울 수 있습니다. 모델 내부의 세부 사항에 대한 접근이 제한적이기 때문에 발생한 오류를 추적하거나 모델의 동작을 자세히 분석하기가 어려울 수 있습니다.

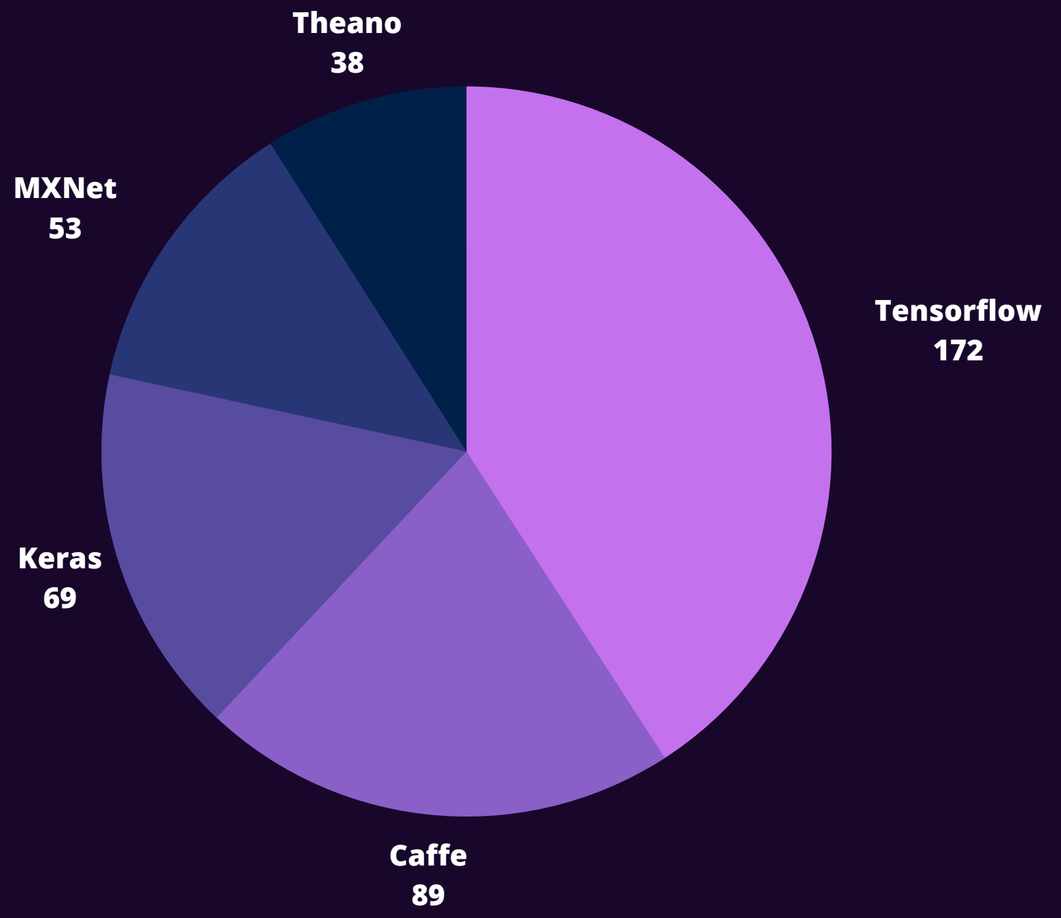
04

일부 고급 기능의 부족

Keras는 초기에 단순화를 위해 디자인되었기 때문에, 일부 고급 기능이 부족할 수 있습니다. 특정한 연구 목적이나 특수한 요구사항을 충족하기 위해서는 더 낮은 수준의 프레임워크를 사용하는 것이 더 적합할 수 있습니다.

딥러닝 라이브러리 인기순위

오픈소스 공유 사이트 깃허브(Github)에 따르면 올 2월 기준 딥러닝 라이브러리 인기 순위에서 텐서플로는 172점의 점수를 받아 기타 라이브러리를 압도하고 있는 것으로 나타났습니다. 텐서플로에 이어 버클리 비전 및 학습 센터(BVLC)가 개발한 카페(Caffe, 89점), 케라스(Keras, 69점), 아마존이 지원하는 MX넷(MXNet, 53점), 테아노(Theano, 38점) 순입니다. 특히 MX넷은 아직 구글 텐서플로에 미치지는 못하지만, 지난해 아마존의 투자와 지원에 힘입어 생태계를 빠르게 확장하고 있습니다.



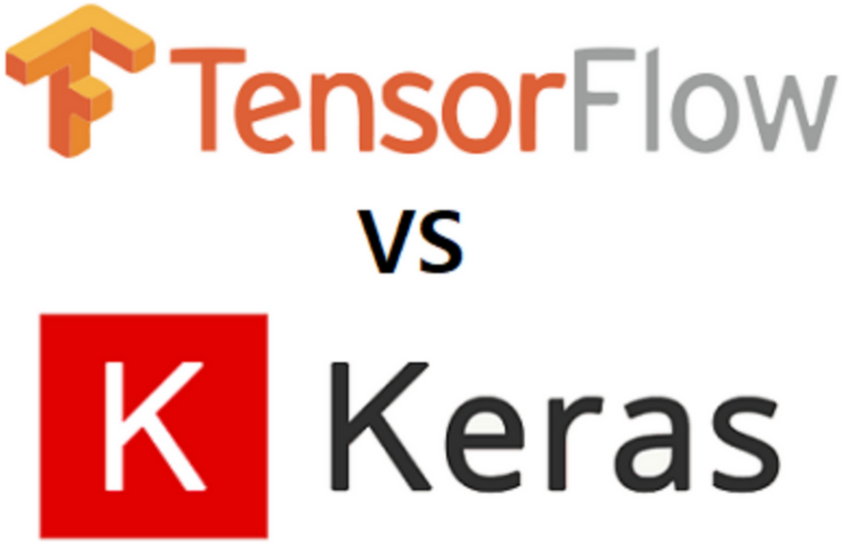
텐서플로우와의 관계

■ 둘은 무슨 사이일까?

Tensorflow는 잘 알려진 것처럼 구글에서 개발하고 오픈소스로 공개한 머신러닝 라이브러리입니다. Keras 역시 라이브러리이지만 Keras는 Tensorflow위에서 동작하는 라이브러리 입니다.

■ 왜 Tensorflow가 있는데 그 위에서 동작하는 Keras가 필요할까?

Tensorflow는 훌륭한 라이브러리이지만 아직 사용을 하기에는 어려운 부분이 많습니다. 특히 처음 머신러닝을 접하는 사람이라면 더욱 그렇겠죠. 반면 Keras는 사용자 친화적으로 개발되었기 때문에 매우 사용이 편합니다. 정말 간단한 신경망의 경우 겨우 몇 줄 만으로 만들 수가 있습니다.



케라스 모델 구축과정

01

데이터 생성

- 원본 데이터를 불러오거나 시뮬레이션 등을 통해 데이터를 생성합니다.

02

모델 구성하기

- Sequence 모델을 생성한 뒤 필요한 레이어를 추가하여 구성합니다.
- 복잡한 모델이 필요할 때는 케라스에서 활용가능한 여러 함수들을 이용합니다.

03

모델 학습과정 설정하기

- 학습하기 전에 학습에 대한 설정을 수행합니다. (손실함수, 최적화 방법 등)
- 케라스에서는 compile( ) 함수를 이용하여 학습과정을 설정합니다.

04

모델 학습

- 위에서 구성한 모델을 fit( ) 함수를 이용하여 train 데이터 셋을 학습시킵니다.

05

학습과정 살펴보기

- 모델 학습 시, train, validation 데이터셋의 손실 및 정확도 등을 측정합니다.
- 반복횟수에 따른 손실 및 정확도의 추이를 보며 학습 상황을 판단합니다.

06

모델 평가하기

- evaluate( ) 함수로 test 데이터셋으로 모델을 평가합니다.

07

모델 사용하기

- predict( ) 함수로 임의의 입력값에 대한 모델의 출력값을 얻습니다.

딥러닝 모델 훈련 예시

Python으로 keras를 이용하여 앞서 언급한 단계를 활용해 몇가지 모델을 만들어 보았습니다.

1. 입력 데이터와 타깃 데이터를 생성합니다. 입력 데이터인 x_train은 1000개의 샘플로 이루어진 2D Numpy 배열입니다. 타깃 데이터인 y_train은 0과 1 사이의 랜덤한 정수로 이루어진 1D Numpy 배열입니다.
2. Sequential 모델을 생성하고 Dense 레이어를 추가하여 모델 구축합니다. 첫 번째 Dense 레이어는 10개의 입력을 받아 32개의 유닛을 가지는 은닉층입니다. 두 번째 Dense 레이어는 이진 분류를 위해 1개의 출력을 가지는 출력층입니다.
3. 모델을 컴파일합니다. compile() 메서드를 사용하여 최적화 방법(optimizer)과 손실 함수(loss)를 설정합니다. 이 예시에서는 Adam 옵티마이저와 이진 크로스 엔트로피 손실 함수를 사용합니다. 또한, 평가 지표(metrics)로 정확도를 설정합니다.
4. 모델을 훈련합니다. fit() 메서드를 사용하여 훈련 데이터 x_train과 y_train을 주입하고, 에포크(epoch) 횟수와 배치 크기(batch_size)를 설정합니다. 또한, 검증 데이터(validation data)로 훈련 중 모델의 성능을 모니터링할 수 있도록 설정합니다. 모델은 지정된 에포크 횟수 동안 훈련되며, 손실과 정확도가 출력됩니다.

```python
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

# 랜덤한 입력 데이터 생성
x_train = np.random.random((1000, 10))
y_train = np.random.randint(2, size=(1000, 1))

# 모델 구축
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=10))
model.add(Dense(1, activation='sigmoid'))

# 모델 컴파일
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# 모델 훈련
model.fit(x_train, y_train, epochs=10, batch_size=32, validation_split=0.2)
```

```
Epoch 1/5
25/25 [==============================] - 0s 3ms/step - loss: 0.6921 - accuracy: 0.5213 - val_loss: 0.7013 - val_accuracy: 0.4500
Epoch 2/5
25/25 [==============================] - 0s 800us/step - loss: 0.6908 - accuracy: 0.5088 - val_loss: 0.7006 - val_accuracy: 0.4600
Epoch 3/5
25/25 [==============================] - 0s 811us/step - loss: 0.6911 - accuracy: 0.5188 - val_loss: 0.7016 - val_accuracy: 0.4700
Epoch 4/5
25/25 [==============================] - 0s 739us/step - loss: 0.6896 - accuracy: 0.5350 - val_loss: 0.7039 - val_accuracy: 0.4550
Epoch 5/5
25/25 [==============================] - 0s 711us/step - loss: 0.6891 - accuracy: 0.5150 - val_loss: 0.7010 - val_accuracy: 0.4450
```

```python
from keras.datasets import mnist
from keras import models
from keras import layers
from keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
network.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

```
Epoch 1/5
469/469 [==============================] - 1s 2ms/step - loss: 0.2539 - accuracy: 0.9266
Epoch 2/5
469/469 [==============================] - 1s 2ms/step - loss: 0.1019 - accuracy: 0.9701
Epoch 3/5
469/469 [==============================] - 1s 2ms/step - loss: 0.0681 - accuracy: 0.9796
Epoch 4/5
469/469 [==============================] - 1s 2ms/step - loss: 0.0493 - accuracy: 0.9848
Epoch 5/5
469/469 [==============================] - 1s 2ms/step - loss: 0.0373 - accuracy: 0.9887
```

1. 데이터셋 로드
   - mnist.load_data() 함수를 사용하여 MNIST 데이터셋을 로드합니다.
2. 모델 구축
   - models.Sequential()을 사용하여 Sequential 모델을 생성합니다.
   - Dense 레이어를 추가하여 모델을 구성합니다.
   - 첫 번째 Dense 레이어는 512개의 유닛과 ReLU 활성화 함수를 가지며, 입력 형태는 28 * 28 크기의 이미지입니다.
   - 두 번째 Dense 레이어는 10개의 유닛과 소프트맥스 활성화 함수를 가지며, 0부터 9까지의 숫자를 예측합니다.
3. 모델 컴파일
   - compile() 메서드를 사용하여 모델을 컴파일합니다.
   - 옵티마이저로 'rmsprop'을 사용하고, 손실 함수로 'categorical_crossentropy'를 사용합니다.
4. 데이터 전처리
   - 훈련 데이터와 테스트 데이터를 0과 1 사이의 값으로 스케일링합니다. 각 픽셀은 0부터 255까지의 값을 가지므로, 255로 나누어 0과 1 사이의 값으로 정규화합니다.
   - 훈련 데이터와 테스트 데이터를 28 * 28 크기의 1D 배열로 변환합니다.
5. 레이블 전처리
   - 레이블을 원-핫 인코딩 형태로 변환합니다. 예를 들어, 숫자 '3'은 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]과 같이 변환됩니다.
6. 모델 훈련
   - fit() 메서드를 사용하여 모델을 훈련합니다.
   - 훈련 데이터 train_images와 train_labels를 주입하고, 에포크 횟수를 5로 설정하고, 배치 크기를 128로 설정합니다.

```python
from keras.layers import Input, Dense
from keras.models import Model

# 입력 레이어 정의
inputs = Input(shape=(10,))

# 은닉층 1 정의
hidden1 = Dense(32, activation='relu')(inputs)

# 은닉층 2 정의
hidden2 = Dense(64, activation='relu')(hidden1)

# 출력 레이어 정의
outputs = Dense(1, activation='sigmoid')(hidden2)

# 함수형 API 모델 생성
model = Model(inputs=inputs, outputs=outputs)

# 모델 요약 정보 출력
model.summary()
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 10)]              0

 dense (Dense)               (None, 32)                352

 dense_1 (Dense)             (None, 64)                2112

 dense_2 (Dense)             (None, 1)                 65

=================================================================
Total params: 2529 (9.88 KB)
Trainable params: 2529 (9.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

함수형 API 모델

1. Input 레이어를 사용하여 입력 레이어를 정의합니다. shape 매개변수를 통해 입력 데이터의 형태를 지정합니다.
2. 각 은닉층은 Dense 레이어로 정의됩니다. 각 은닉층은 입력으로 이전 레이어를 받고, activation 매개변수를 통해 활성화 함수를 지정합니다.
3. 출력 레이어는 마지막 은닉층을 입력으로 받아 Dense 레이어로 정의됩니다. 이 예시에서는 이진 분류를 위해 sigmoid 활성화 함수를 사용합니다.
4. Model 클래스를 사용하여 함수형 API 모델을 생성합니다. inputs와 outputs를 매개변수로 전달하여 모델을 정의합니다.
5. model.summary()를 호출하여 모델의 요약 정보를 출력합니다. 이 정보에는 레이어의 구성과 파라미터 개수 등이 포함됩니다.

함수형 API 모델 VS Sequential 모델

함수형 API 모델

- 함수형 API 모델은 레고 블록을 조립하는 것과 유사합니다. 각 레이어를 개별적인 블록으로 생각하고, 이러한 블록들을 조합하여 복잡한 모델을 만들 수 있습니다.
- 다중 입력과 다중 출력을 처리할 수 있습니다. 예를 들어, 이미지와 텍스트 두 가지 다른 종류의 입력을 받고, 동시에 여러 가지 결과를 출력하는 모델을 만들 수 있습니다.

Sequential 모델

- Sequential 모델은 간단하고 직관적인 구조를 가지며, 순차적으로 레이어를 쌓는 방식입니다.
- 단일 입력과 단일 출력을 처리하는 모델에 적합합니다. 예를 들어, 이미지를 입력으로 받아 이미지에 대한 분류 결과를 출력하는 모델을 만들 수 있습니다.
- 간단한 구조를 가지고 있어 초보자가 사용하기 쉽습니다..

케라스는 다음과 같은 분야에 활용됩니다

이미지 분류

컴퓨터 비전 분야에서 이미지를 다양한 클래스로 분류하는 모델을 개발할 때 케라스를 활용할 수 있습니다. 예를 들어, 제가 앞서 보여드렸던 MNIST와 같은 이미지 데이터셋을 사용하여 손글씨 숫자 분류나 개/고양이 분류와 같은 작업을 수행할 수 있습니다.

객체 감지

객체 감지는 이미지 내에서 객체의 위치와 클래스를 식별하는 작업입니다. 케라스의 라이브러리인 TensorFlow Object Detection API를 사용하면, Faster R-CNN, SSD, YOLO와 같은 대표적인 객체 감지 알고리즘을 구현할 수 있습니다.

자연어 처리

문장 분류, 감정 분석, 기계 번역, 텍스트 생성 등과 같은 다양한 자연어 처리 작업에 케라스를 활용할 수 있습니다. 앞서 보여드렸던 케라스의 Sequential 모델이나 함수형 API를 사용하여 텍스트 데이터를 처리하고, RNN, LSTM, Transformer 등과 같은 다양한 모델 구조를 구축할 수 있습니다.

감사합니다

https://github.com/Jongwoo0101/Learn_Keras

wonjongwoo01@gmail.com

01031332293