3. 케라스(Keras)_Part 2

4. 모델 세부 설정

4-1. 초기값 설정

- 레이어의 초기화 방법을 다르게 설정 가능
- default: GlorotUniform
- `kernel_initializer` 매개변수 활용
  - GlorotUniform, HeNormal 2가지가 있음
- **클래스형**과 **함수형**으로 모두 제공
  - 클래스형

```
# 클래스 인스턴스 초기화
he_normal = tf.keras.initializers.HeNormal()
dense = tf.keras.layers.Dense(256, kernel_initializer=he_normal, activation='relu')
dense.get_config()['kernel_initializer']
```

  - 함수형

```
dense = tf.keras.layers.Dense(256, kernel_initializer = 'he_normal',
                              activation = 'relu')
dense.get_config()['kernel_initializer']
```

- Keras에서 지원하는 초기화 목록
  - ‘glorot_normal’, ‘glorot_uniform’: 글로럿 초기화(Xavier 초기화)
  - ‘lecun_normal’, ‘lecun_uniform’: Yann Lecun 초기화
  - ‘he_normal’, ‘he_uniform’: He 초기화
  - ‘random_normal’, ‘random_unifom’: 정규 분포, 연속균등 분포 초기화

4-2. 규제(Regularization)

- 모델의 과대적합을 해소하기 위해 적용

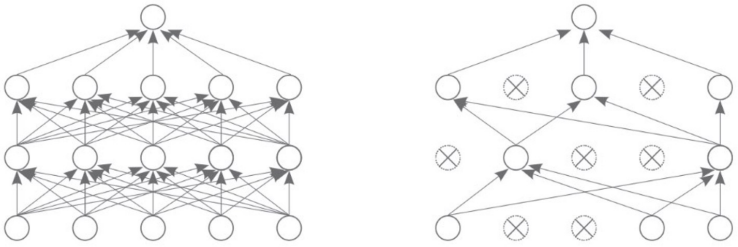
- Tensorflow Keras Layer는 기본적으로 규제를 적용하고 있지 x
- 모델을 구성하는 레이어마다 규제 적용 가능

  → `kernel_regularizer`에 규제 지정

4-3. 드롭아웃(Dropout)

- 딥러닝 모델의 가장 큰 난제가 바로 과대적합 문제임
  - 딥러닝 모델의 층이 넓고 깊어질 때 모델은 훈련에 주어진 샘플에 과하게 적합하도록 학습하는 경향이 있음

    ⇒ 훈련 데이터셋에 너무 적응하여 검증 데이터셋이나 테스트 데이터셋에 대해 일반화된 성능을 갖지 못하는 문제
- 드롭아웃
  - 모델의 과대적합 문제를 해결하기 위해 제안된 아이디어



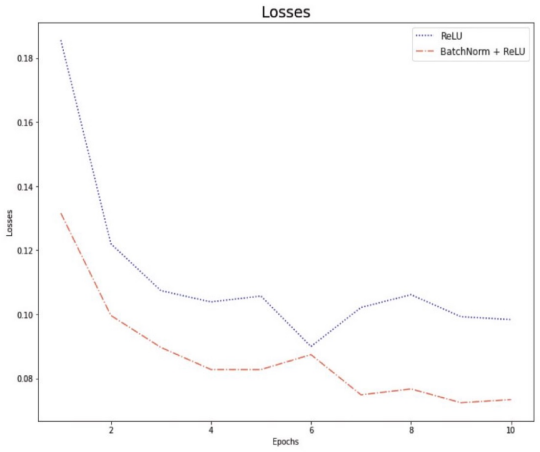
[그림 3-22] 드롭아웃[11]

  - 노드의 일부 신호를 임의로 삭제하는 방법 → 모델이 학습하는 가중치 파라미터의 개수 감소

4-4. 배치 정규화(Batch Normalization)

- 각 층에서 활성화 함수를 통과하기 전 mini batch의 scale을 정규화
  - 다음 층으로 데이터가 전달되기 전에 스케일을 조정

    → 보다 안정적인 훈련 가능, 성능 향상



[그림 3-23] 훈련 결과 비교

- 배치 정규화 층은 케라스에서 클래스 함수로 지원

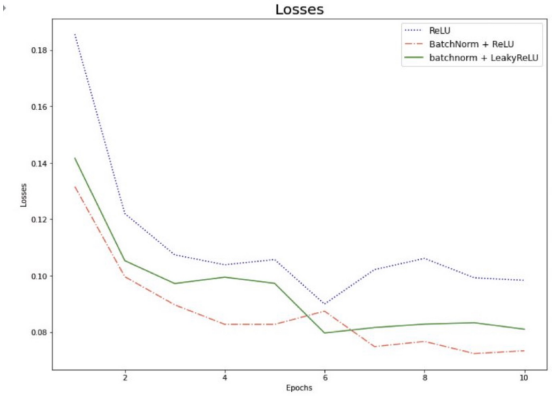
4-5. 활성화 함수(activation function)

- 클래스 인스턴스로 선언하여 하이퍼파라미터 값을 변경하여 적용할 수 있음

```
# LeakyReLU 기본 설정
tf.keras.layers.LeakyReLU()

# LeakyReLU, alpha = 0.2 로 변경
tf.keras.layers.LeakyReLU(alpha = 0.2)
```

- 수렴 속도 향상에 기여할 수 있음



6. 모델 저장 및 불러오기

6-1. 모델을 파일로 저장

- 훈련 종료 시 가중치가 업데이트된 모델 인스턴스를 저장할 수 있음
  - `save()` 메소드 사용

    - 메소드 호출 시 저장할 파일의 디렉토리를 포함하는 파일명을 매개변수로 지정
    - 저장 형식: HDF5, SavedModel

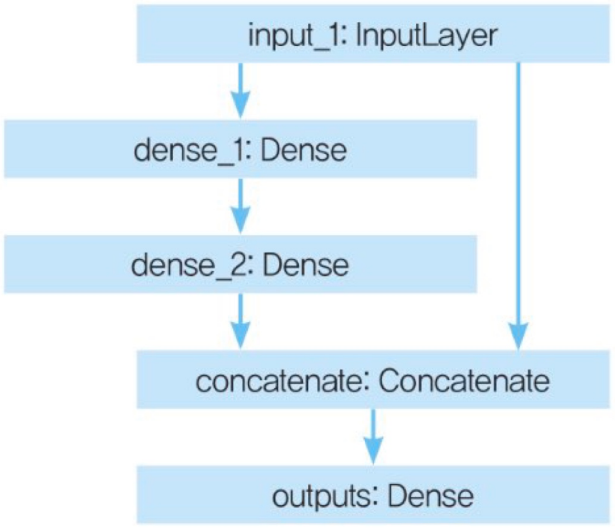
6-2. 저장된 모델 복원

- `tensorflow.keras.models.load_model()` 메소드로 저장된 모델을 복원할 수 있음
- 모델 복원 후 `summary()`를 확인하여 모델 구조를 재확인 할 수 있음

7. 복잡한 모델 생성

7-1. 함수형 API

- 함수 형태로 딥러닝 모델을 정의하면 다양한 모델 구조를 구현할 수 있음
- 함수의 return 값을 여러 개 갖는 다중 출력, 같은 레벨에 여러 개의 층을 배치하여 입력과 출력을 공유하는 구조 등도 구현 가능
    - 데이터 흐름이 특정 레이어를 건너뛰거나, 병합 및 분리하는 등의 구조의 모델 또한 구현 가능
- **함수형 API 사용하기**

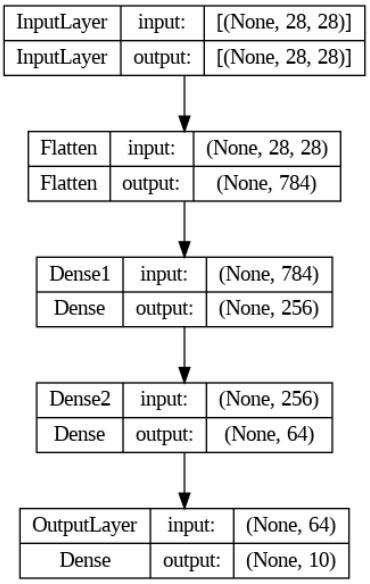


[그림 3-30] Functional API

1. `Input` 레이어 정의
    - 데이터의 입력 shape을 정의
    - 레이어마다 반환되는 출력 값을 변수에 저장한 뒤 다음 레이어의 입력으로 연결
    - 레이어마다 `name` 매개변수로 이름을 부여할 수 있음

2. 체인 방식으로 레이어를 연결한 후 `tf.keras.Model()`에 입력 레이어와 출력 레이어를 정의해 모델 생성

- `keras.utils.plot_model()`을 통해 모델 구조를 시각화 할 수 있음



  - `show_shape = True`: 데이터의 입출력 shape를 출력
  - `show_layer_names = True`: 레이어의 이름 출력
  - `to_file`: 파일명 입력 시 이미지 파일로 저장
- 함수형 API로 생성한 모델 또한 Sequential API로 생성한 모델과 동일하게 훈련 가능
  - 생성된 모델 인스턴스에 `compile()` 메소드로 모델을 컴파일
  - `fit()` 메소드로 모델 훈련
  - 훈련 완료 후 동일하게 evaluate()를 통해 검증 가능

7-2. 모델 서브클래싱(Model Subclassing)

- 케라스는 Model 클래스를 기반으로 딥러닝 모델을 구현하고 있음
  - 해당 클래스를 직접 상속받아 사용자가 직접 서브클래스로 딥러닝 모델을 구현할 수 있음
- `tf.Keras.Model`을 상속받아 생성하고자 하는 모델 인스턴스(→ 클래스)를 생성
  - `init()`: 생성자
    - 모델 설정 부분
  - `call()`: 콜백 함수
    - `fit()` 메소드가 호출되어 훈련되는 경우 호출될 함수

        - 순전파(= 모델의 입력부터 출력까지의 흐름)를 정의하고 함수형 API와 같은 방식으로 모든 레이어를 체인처럼 연결
        - 마지막으로 출력값을 return
- Model Subclassing으로 생성한 모델 또한 Sequential API로 생성한 모델과 동일하게 훈련 가능
    - 생성된 모델 인스턴스에 `compile()` 메소드로 모델을 컴파일
    - `fit()` 메소드로 모델 훈련
    - 훈련 완료 후 동일하게 evaluate()를 통해 검증 가능

7-3. 서브클래싱 모델 파라미터를 활용한 모델 생성

- Model Subclassing으로 생성하는 장점은 생성자 파라미터로 모델 내부 레이어의 하이퍼파라미터를 지정할 수 있다는 점임