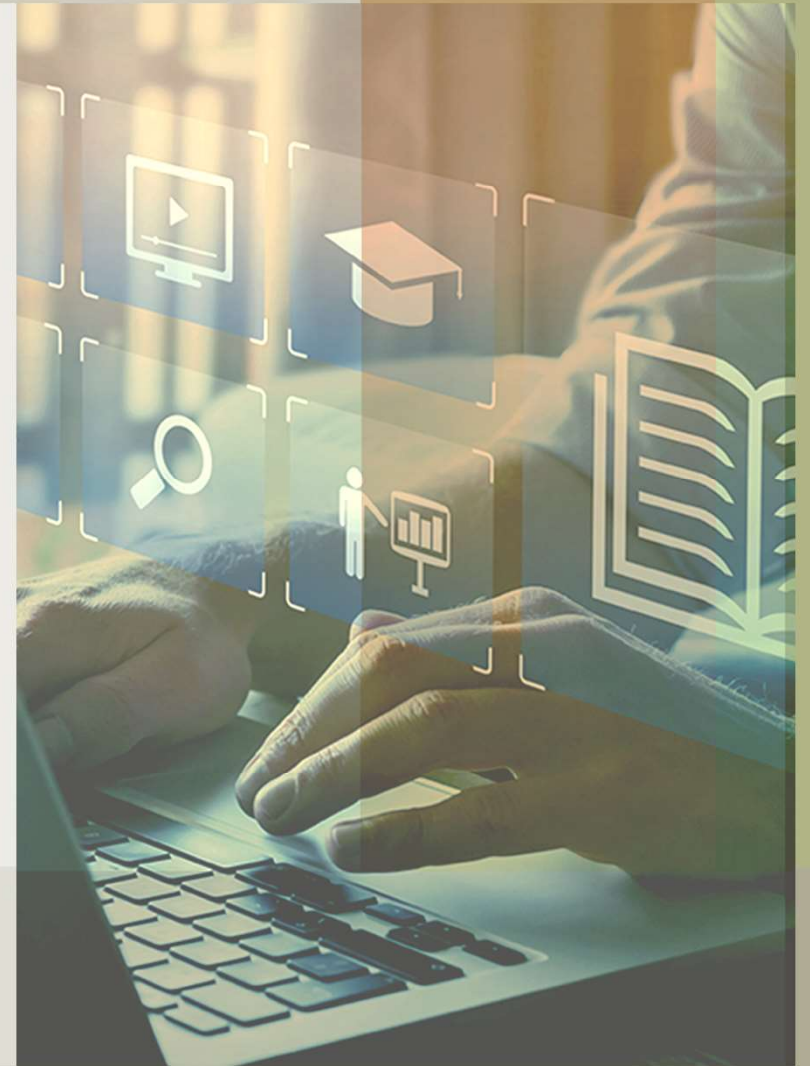


05

딥러닝

딥러닝의 학습 기술(2)

방송대 컴퓨터과학과 이병래 교수



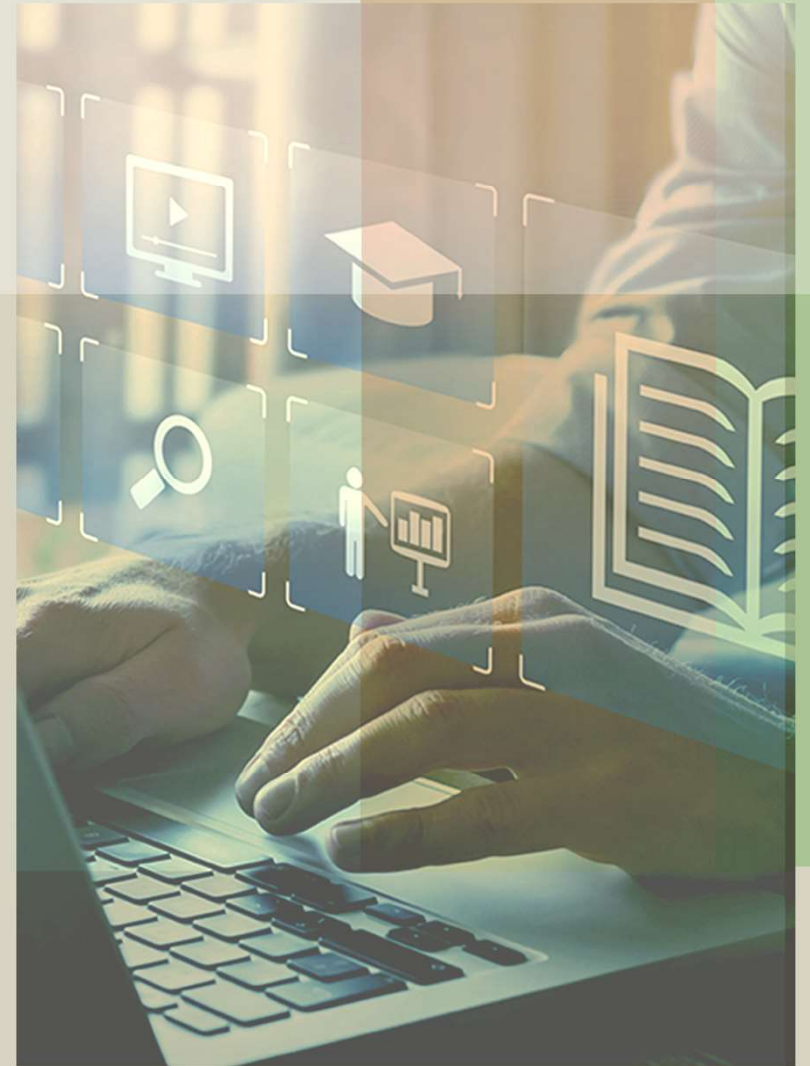
학습목차

- ① 최적화기의 개선
- ② 과적합과 규제
- ③ 배치 정규화
- ④ [실습] MNIST 필기 숫자 인식 - 조기 종료



01


최적화기의 개선



1. 모멘텀 및 네스테로프 가속 경사(NAG)

● 모멘텀

- 이전 업데이트 양(‘속도’)을 일정 비율(‘모멘텀’) 반영하는 방법



$$w(t) \xleftarrow{v(t)} w(t+1)$$

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{v}(t)$$

$$\mathbf{v}(t) = m\mathbf{v}(t-1) - \Delta\mathbf{w}(t), \quad \Delta\mathbf{w}(t) = \eta \nabla J(\mathbf{w}(t))$$

예 `from tensorflow.keras import optimizers`
`optimizer = optimizers.SGD(0.1, momentum=0.9)`



1. 모멘텀 및 네스테로프 가속 경사(NAG)

모멘텀

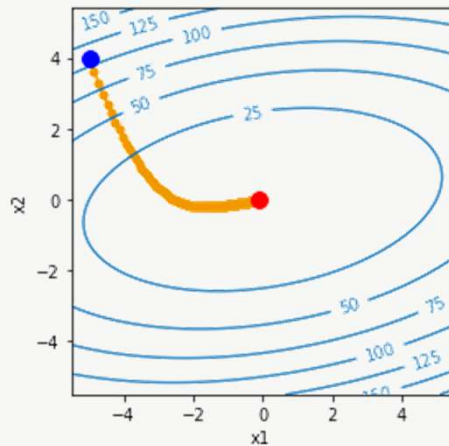
예

목적함수 : $J(\mathbf{x}) = x_1^2 + 2x_2^2 - x_1x_2$

학습률 : $\eta = 0.01$

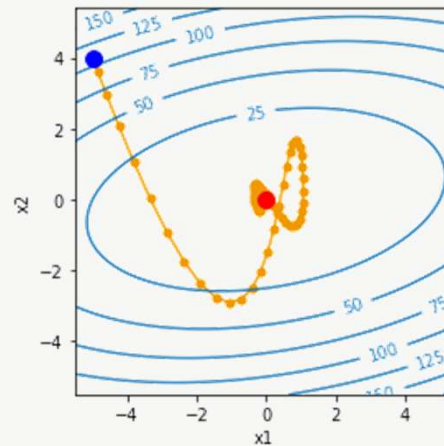
\mathbf{x} 의 초기값 : $\mathbf{x}(0) = [-5 \ 4]^T$

반복 횟수 : 100회



momentum = 0

$\mathbf{x} = [-0.663 \ -0.107]$



momentum = 0.9

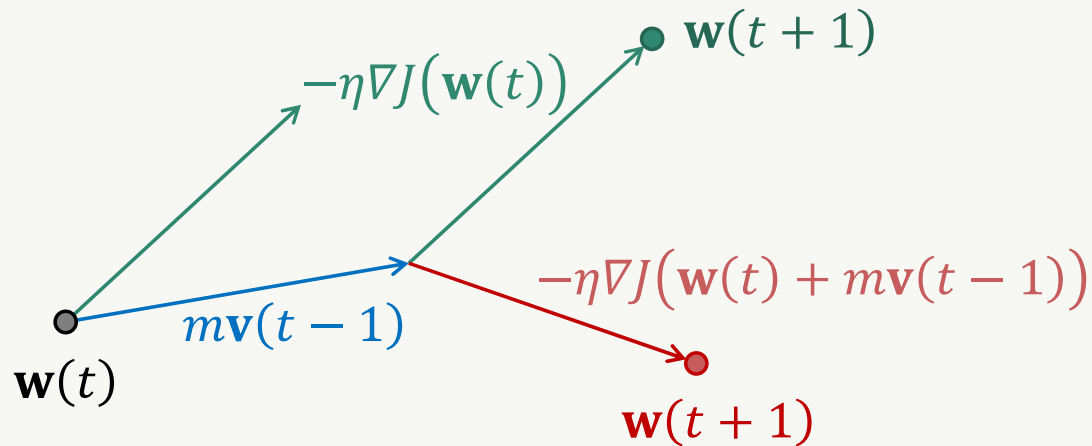
$\mathbf{x} = [-0.020 \ -0.0230]$

1. 모멘텀 및 네스테로프 가속 경사(NAG)

네스테로프 가속 경사(Nesterov Accelerated Gradient, NAG)

$$\mathbf{v}(t) = m\mathbf{v}(t-1) - \eta \nabla J(\mathbf{w}(t) + m\mathbf{v}(t-1))$$

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{v}(t)$$



예

```
from tensorflow.keras import optimizers
optimizer = optimizers.SGD(0.1, momentum=0.9, nesterov=True)
```


1. 모멘텀 및 네스테로프 가속 경사(NAG)

• 네스테로프 가속 경사(Nesterov Accelerated Gradient, NAG)

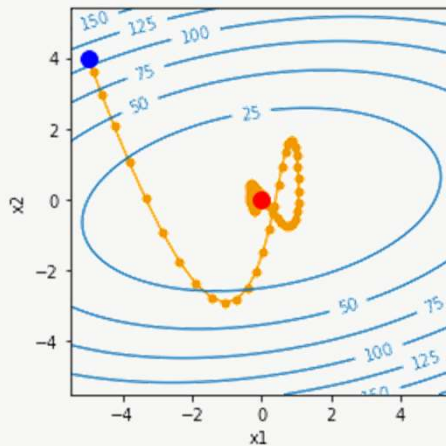
예

목적함수 : $J(\mathbf{x}) = x_1^2 + 2x_2^2 - x_1x_2$

학습률 : $\eta = 0.01$

\mathbf{x} 의 초기값 : $\mathbf{x}(0) = [-5 \ 4]^T$

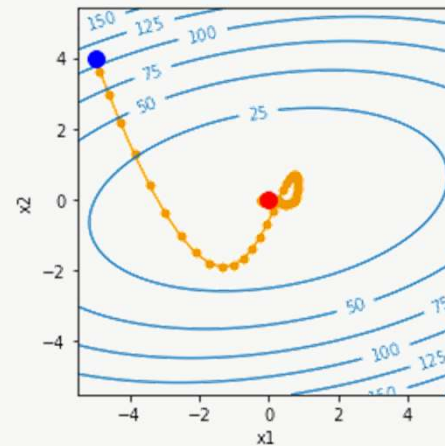
반복 횟수 : 100회



momentum = 0.9

nesterov = **False**

$\mathbf{x} = [-0.020 \ -0.0230]$



momentum = 0.9

nesterov = **True**


$\mathbf{x} = [-0.00850 \ -0.00174]$


2. Adagrad와 RMSProp

● Adagrad(adaptive gradient)

- 학습률을 적응적으로 적용하기 위한 최적화 방법
- 변화가 큰 파라미터의 학습률은 작게, 변화가 작은 파라미터의 학습률은 크게 함으로써 파라미터의 변화가 극소점을 향해 진행되게 함

$$s_i(t) = s_i(t-1) + \{\nabla_i J(\mathbf{w}(t))\}^2$$
$$w_i(t+1) = w_i(t) - \frac{\eta}{\sqrt{s_i(t) + \varepsilon}} \nabla_i J(\mathbf{w}(t))$$

 $\nabla_i J(\mathbf{w}(t)) : \frac{\partial J(\mathbf{w}(t))}{\partial w_i}$ 를 나타냄

 $s_i(t) : \nabla_i J(\mathbf{w}(t))$ 의 제곱을 누적한 값

 $s_i(t)$ 가 점점 커져 학습률이 매우 작아지게 됨



2. Adagrad와 RMSProp

Adagrad(adaptive gradient)

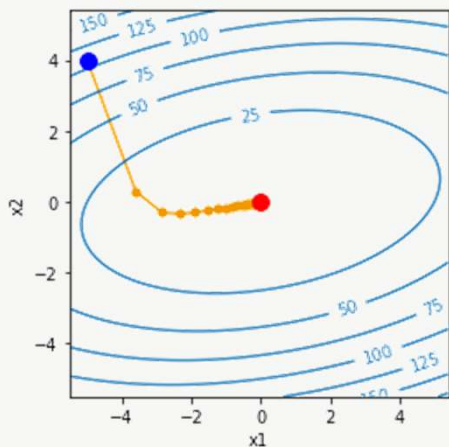
예

목적함수 : $J(\mathbf{x}) = x_1^2 + 2x_2^2 - x_1x_2$

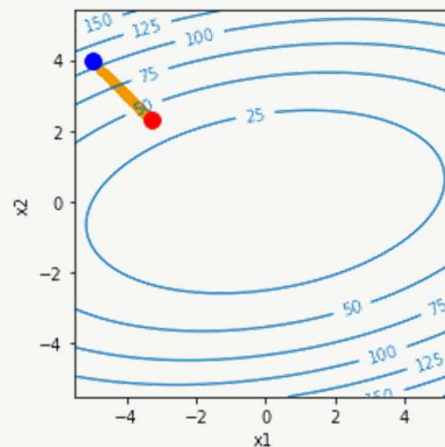
학습률 : $\eta = 0.1$

\mathbf{x} 의 초기값 : $\mathbf{x}(0) = [-5 \ 4]^T$

반복 횟수 : 100회



기본적인 경사 하강법




Adagrad

2. Adagrad와 RMSProp

● RMSProp(Root Mean Square Propagation)

- Adagrad에서 학습률이 지나치게 작아지는 문제를 방지함
 - $\nabla_i J(\mathbf{w}(t))$ 의 제곱을 지수함수적으로 감쇠하도록 누적함

$$s_i(t) = \rho s_i(t-1) + (1 - \rho) \{\nabla_i J(\mathbf{w}(t))\}^2$$
$$w_i(t+1) = w_i(t) - \frac{\eta}{\sqrt{s_i(t) + \varepsilon}} \nabla_i J(\mathbf{w}(t))$$

 ρ : 감쇠를 위한 값

예

```
from tensorflow.keras import optimizers  
optimizer = optimizers.RMSprop(0.01, rho=0.9)
```



2. Adagrad와 RMSProp

● RMSProp(Root Mean Square Propagation)

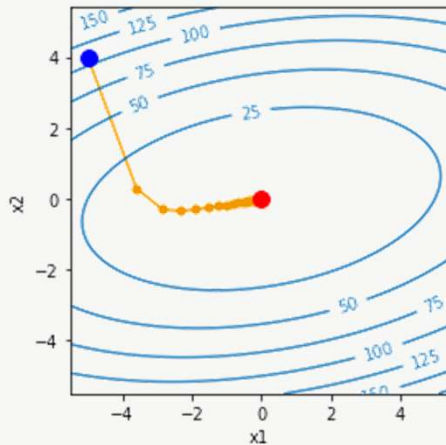
예

목적함수 : $J(\mathbf{x}) = x_1^2 + 2x_2^2 - x_1x_2$

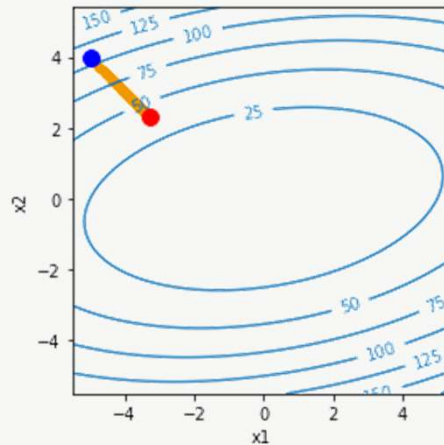
학습률 : $\eta = 0.1$

\mathbf{x} 의 초기값 : $\mathbf{x}(0) = [-5 \ 4]^T$

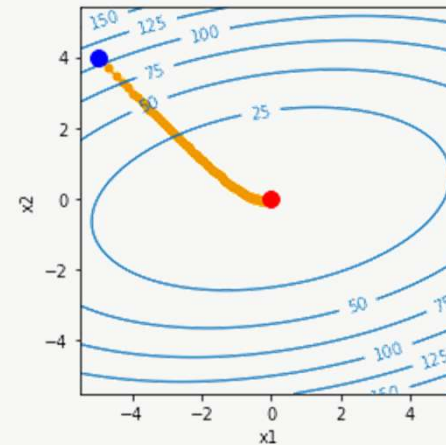
반복 횟수 : 100회



기본적인 경사 하강법



Adagrad



RMSProp

3. Adam

Adam(Adaptive Momentum estimation)


모멘텀과 RMSProp을 결합한 최적화 알고리즘

- 1차 모멘트의 추정치 $\mathbf{m}(t)$: 경사 $\nabla J(\mathbf{w}(t))$ 의 이동평균
- 2차 모멘트의 추정치 $\mathbf{v}(t)$: 경사 제곱의 이동평균

$$m_i(t) = \beta_1 m_i(t-1) + (1 - \beta_1) \nabla_i J(\mathbf{w}(t))$$

$$v_i(t) = \beta_2 v_i(t-1) + (1 - \beta_2) \{\nabla_i J(\mathbf{w}(t))\}^2$$

$$w_i(t+1) = w_i(t) - \frac{\eta}{\sqrt{\hat{v}_i(t)} + \epsilon} \hat{m}_i(t)$$

 $\hat{m}_i(t) = m_i(t) / (1 - \beta_1^t)$

$$\hat{v}_i(t) = v_i(t) / (1 - \beta_2^t)$$



3. Adam

◉ Adam(Adaptive Momentum estimation)

■ 텐서플로(Keras)의 Adam 최적화기 사용 예문

- 학습률 = 0.01
- $\beta_1 = 0.9$
- $\beta_2 = 0.99$

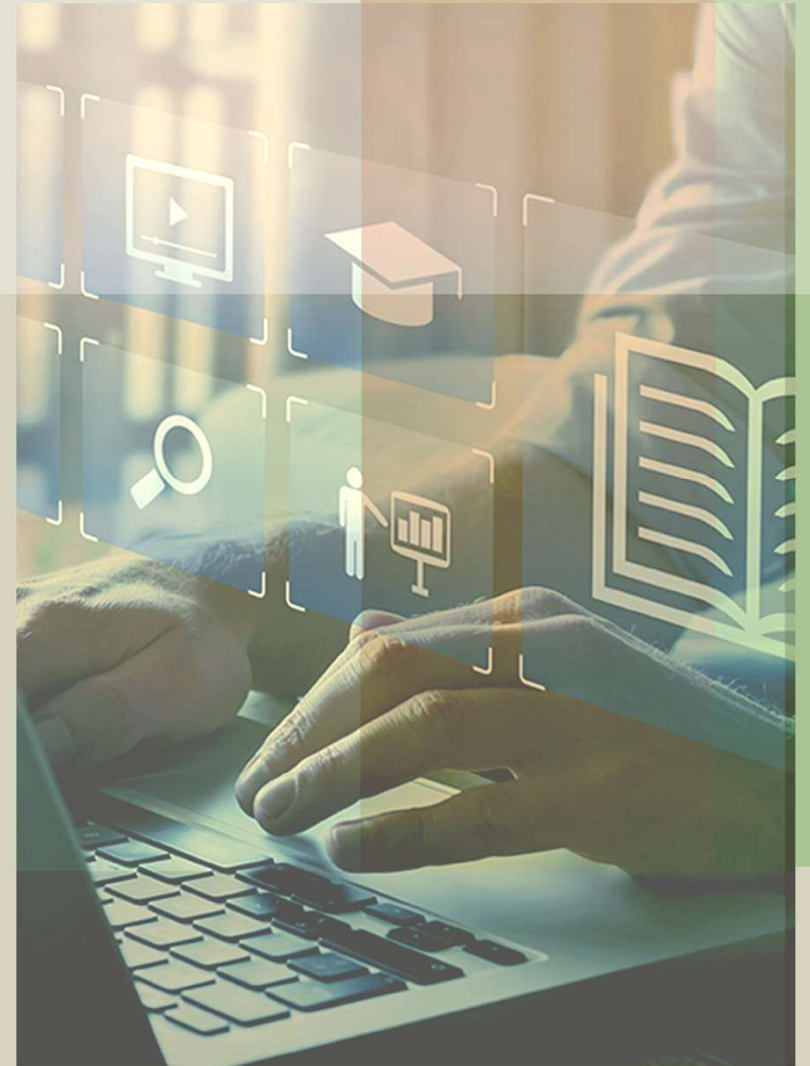
예

```
from tensorflow.keras import optimizers  
optimizer = optimizers.Adam(0.01, beta_1=0.9, beta_2=0.99)
```



02

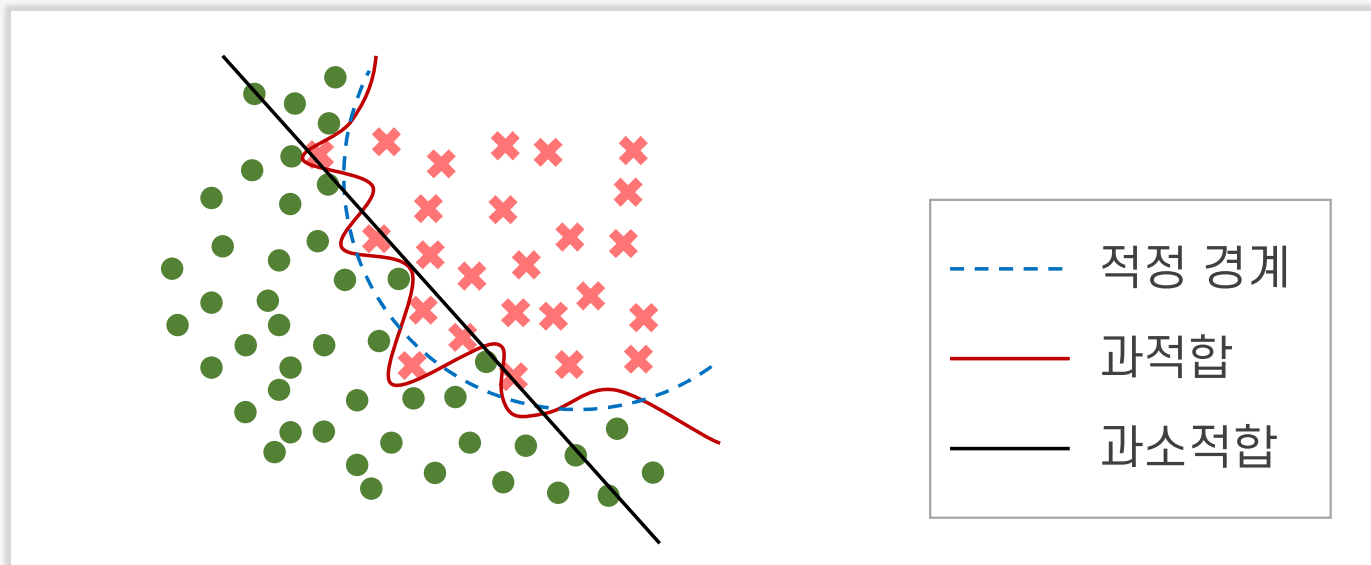
과적합과 규제



1. 과적합과 일반화 오류

과적합(overfitting)

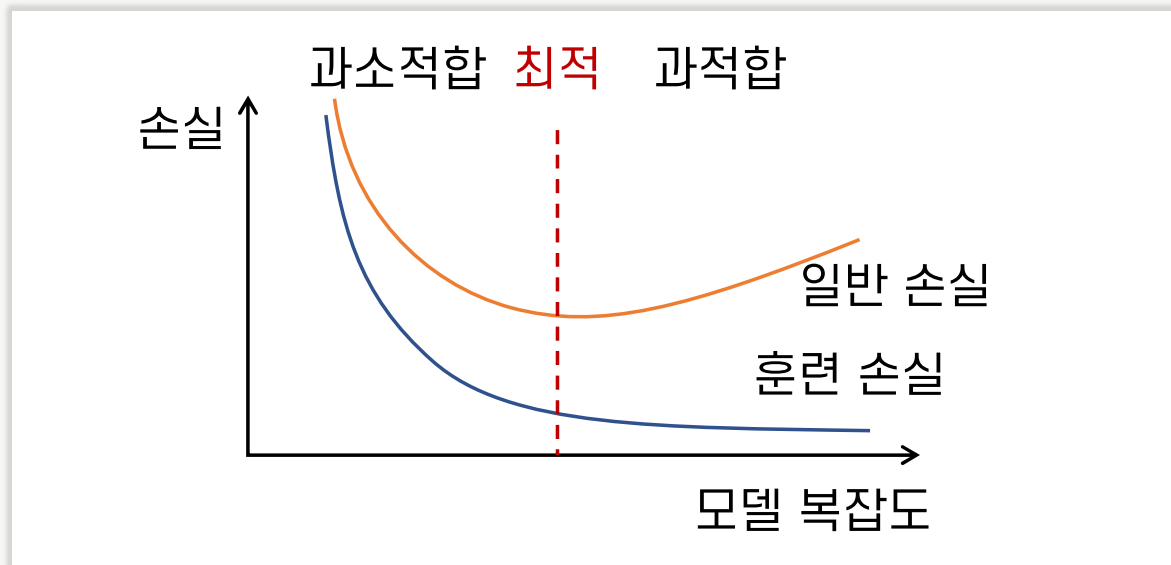
- 훈련에 사용한 데이터에 지나치게 적합하게 학습되는 것



1. 과적합과 일반화 오류

● 과적합(overfitting)

- 훈련에 사용한 데이터에 지나치게 적합하게 학습되는 것



- ➡ 규제(regularization) : 모델 복잡도를 낮추어 더 '단순'한 모델로 만들기 위한 처리 절차



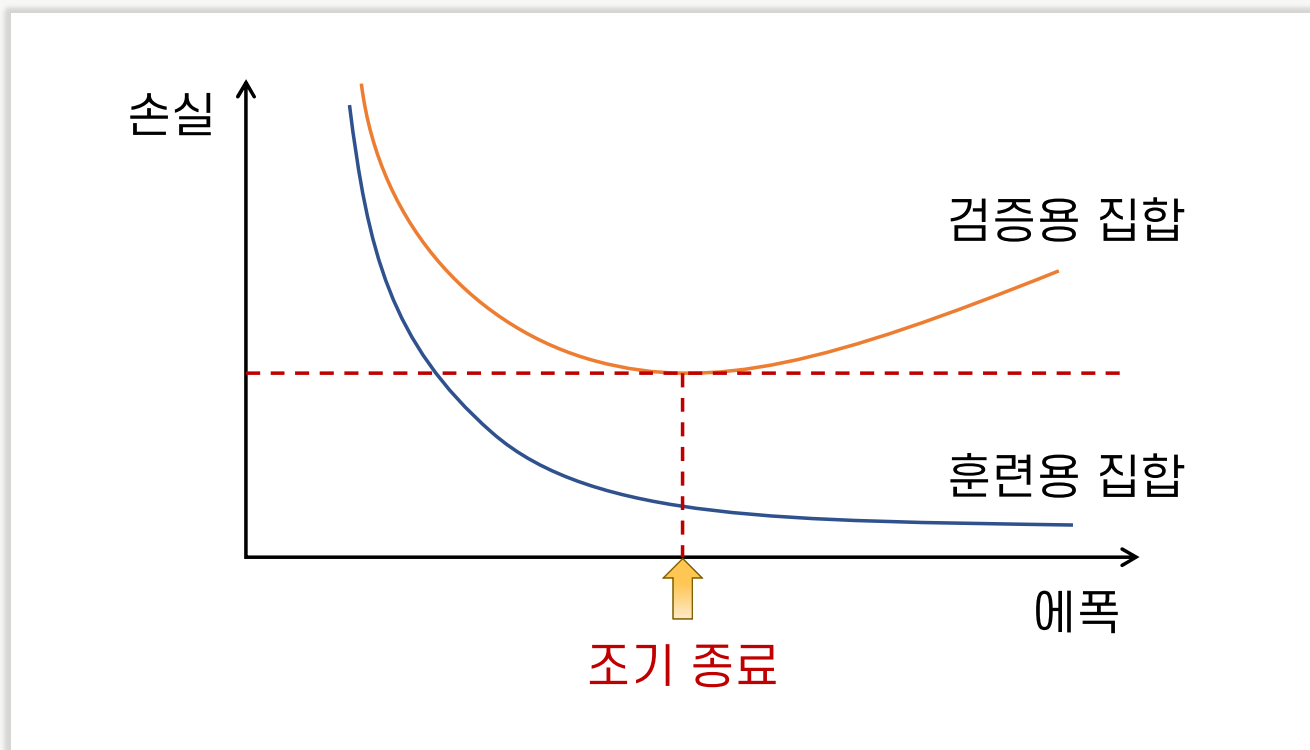
2. 훈련의 조기 종료(early stopping)

- 모델 학습의 검증을 위한 학습 데이터 집합의 활용



2. 훈련의 조기 종료(early stopping)

- 모델 학습의 검증을 위한 학습 데이터 집합의 활용



2. 훈련의 조기 종료(early stopping)

● 텐서플로(Keras)에서 조기 종료 활용하기

■ 조기 종료를 위한 콜백 인스턴스 생성

```
from tensorflow.keras import callbacks
early_stop = callbacks.EarlyStopping(monitor='val_loss',
                                     min_delta=0,
                                     patience=0,
                                     mode='auto',
                                     restore_best_weights=False)
```

■ fit 메소드에 조기 종료 콜백 지정

```
model.fit(tr_data, tr_labels, epochs=1000,
          callbacks=[early_stop])
```



3. 가중치의 규제

● 가중치 감쇠(weight decay)

- 신경망의 가중치가 작은 값을 갖도록 억제함으로써 네트워크의 복잡도에 제한을 가하는 것

$$\mathbf{w}(t+1) = (1 - \lambda)\mathbf{w}(t) - \eta \nabla J(\mathbf{w}(t))$$

 AdamW : 가중치 감쇠를 포함하는 Adam 최적화의 변형



3. 가중치의 규제

● ℓ_1 및 ℓ_2 규제

- 최적화를 위한 목적함수에 손실함수와 더불어 가중치 w 의 크기에 따른 불이익(penalty) 항을 추가하는 규제 방법
 - ➔ w 의 크기가 크면 불이익 항의 값이 커짐
 - ➔ 목적함수의 값이 커지므로 w 의 크기가 작아지도록 최적화가 진행됨



3. 가중치의 규제

• ℓ_1 및 ℓ_2 규제

ℓ_2 규제

$$J(\mathbf{w}) = \frac{1}{N_B} \sum_{i=1}^{N_B} E_i(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2$$

여기에서 $E(\mathbf{w})$: 손실함수

$$\|\mathbf{w}\|_2 = \sqrt{w_1^2 + w_2^2 + \dots + w_d^2}$$

- 기본적인 SGD를 사용하는 경우 ℓ_2 규제는 가중치 감쇠와 동등하나, RMSProp, Adam 등에서는 ℓ_2 규제와 가중치 감쇠는 다름



3. 가중치의 규제

• ℓ_1 및 ℓ_2 규제

ℓ_1 규제

$$J(\mathbf{w}) = \frac{1}{N_B} \sum_{i=1}^{N_B} E_i(\mathbf{w}) + \lambda \|\mathbf{w}\|_1$$

여기에서 $E(\mathbf{w})$: 손실함수

$$\|\mathbf{w}\|_1 = |w_1| + |w_2| + \cdots + |w_d|$$



3. 가중치의 규제

● ℓ_1 및 ℓ_2 규제

- 텐서플로(Keras)에서 ℓ_2 규제의 적용

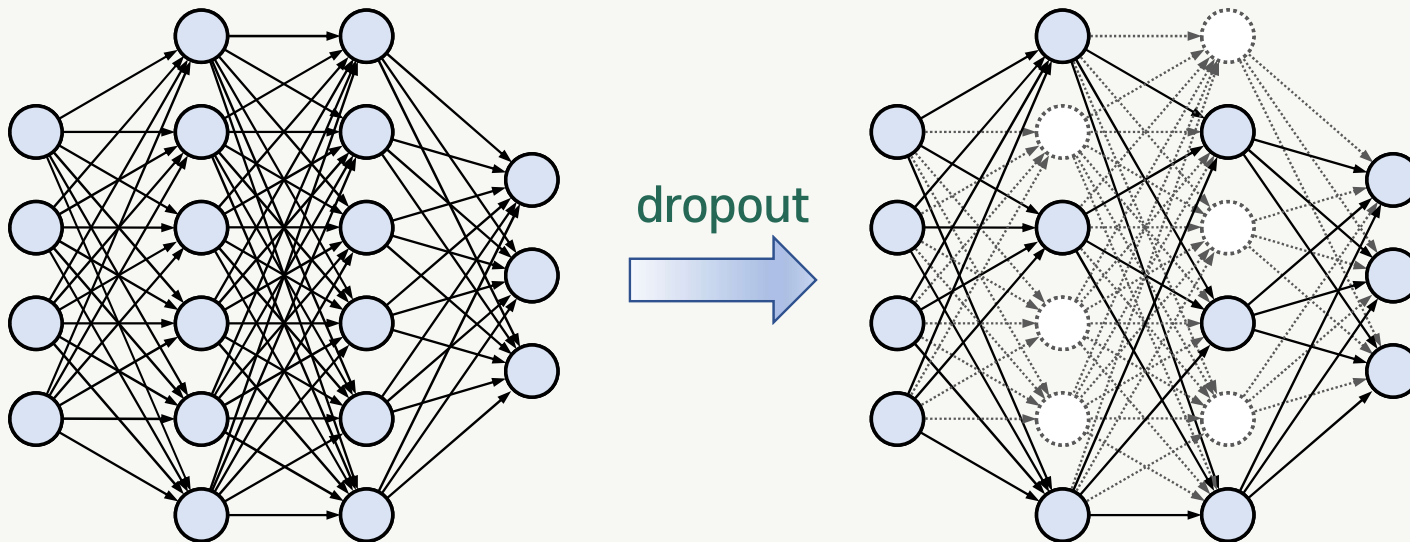
```
from tensorflow.keras import layers, regularizers  
d_layer = layers.Dense(3, activation='relu',  
                        kernel_regularizer=regularizers.L2(0.01))
```



4. 드롭아웃

● 드롭아웃(dropout)의 개념

- 모델을 학습하는 동안 적절한 확률에 따라 뉴런을 무작위로 선택하여 일시적으로 제거하는 것



4. 드롭아웃

● 드롭아웃의 적용

- 적용 대상 층 : 은닉층 및 입력층
- 드롭아웃 비율(dropout rate) : 뉴런을 드롭아웃할 확률을 나타내는 값 p
 - 10~50% 범위에서 지정하는 것이 일반적임
 - 드롭아웃 대상 뉴런의 선정 : 매 훈련 단계마다 무작위로 선정
- 훈련 단계 : 드롭아웃이 적용된 층으로부터 입력을 받을 때 $1/(1 - p)$ 를 곱하여 전체적인 신호 크기 유지
- 추론 단계 : 드롭아웃 없이 모든 뉴런이 동작함



4. 드롭아웃

● 텐서플로(Keras)에서 드롭아웃의 적용

- `tf.keras.layers` 모듈의 Dropout 층을 배치함

```
from tensorflow import keras
from tf.keras import Sequential, layers
model = Sequential([keras.Input(shape=(10,)),
                    layers.Dropout(rate=0.2),
                    layers.Dense(32, activation='relu'),
                    layers.Dropout(rate=0.2),
                    layers.Dense(4, activation='softmax')])
```



03

배치 정규화



1. 배치 정규화의 개념

● 내부 공변량 변화(internal covariate shift)

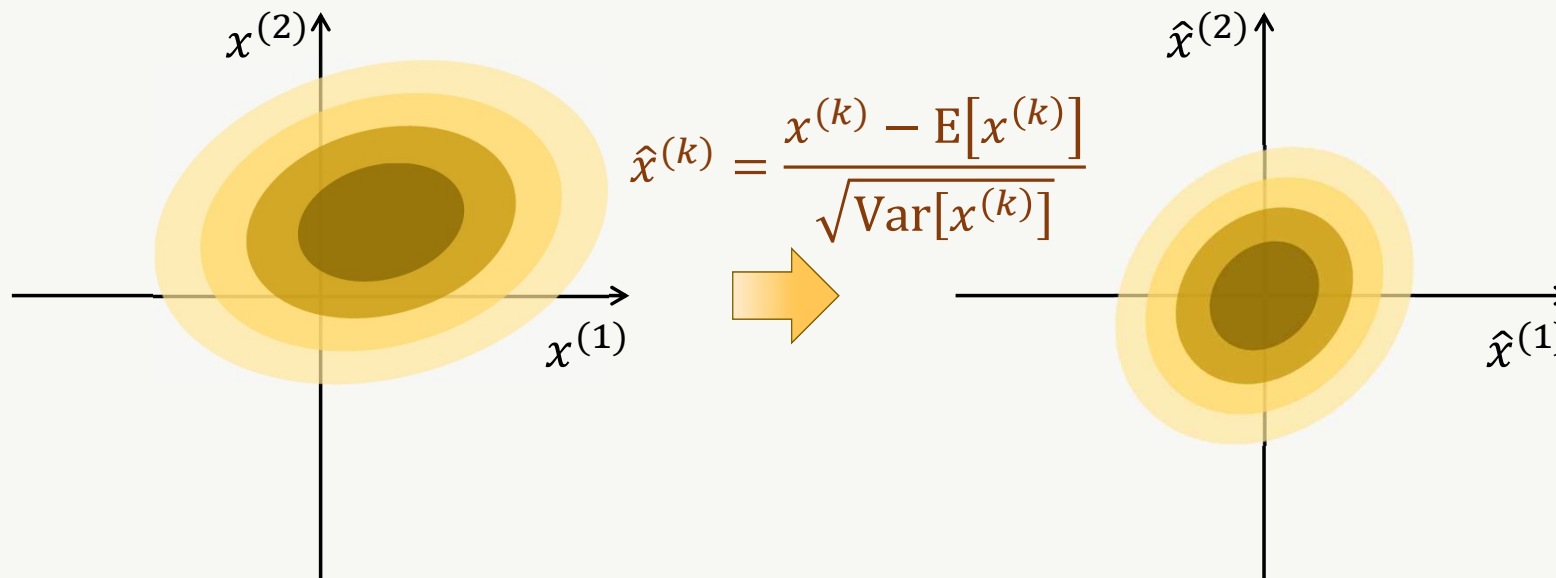
- 학습 중 각 층의 활성화함수 입력 분포가 이전 층의 파라미터 변경에 의해 변화하는 현상
 - 심층망의 학습 효율이 낮아지는 문제의 원인이 됨
 - 작은 학습률을 사용해야 해서 학습이 느리게 진행됨
 - 파라미터 초기화를 주의 깊게 할 필요가 있음
 - 포화 비선형 활성화함수를 사용하는 모델의 학습이 어려움
- ➡ Sergey Ioffe 등(2015) : 배치 정규화(batch normalization)를 통해 학습 성능을 개선할 수 있음



1. 배치 정규화의 개념

정규화(normalization)

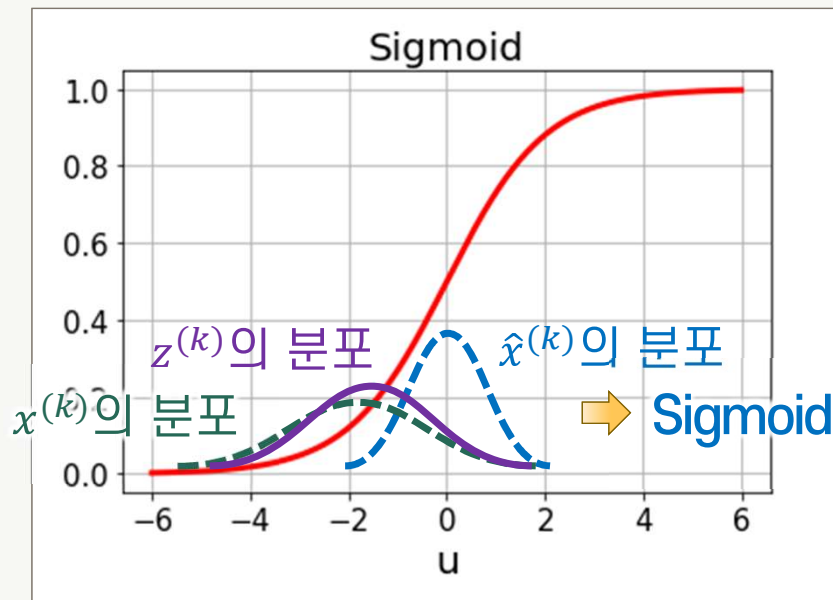
- 입력의 분포가 평균이 0, 분산이 1이 되도록 만드는 것



1. 배치 정규화의 개념

스케일 및 이동 변환

- 정규화만 할 경우 그 층에서 표현하려는 정보에 변화가 발생함



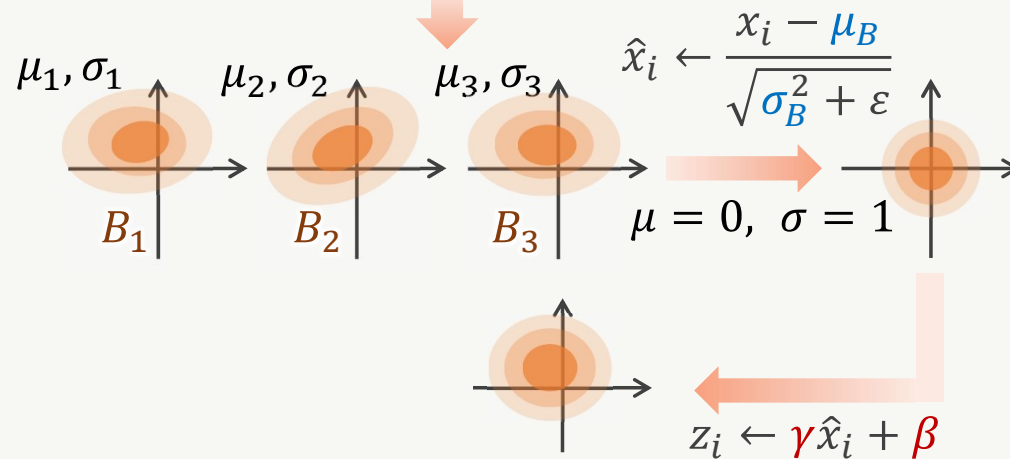
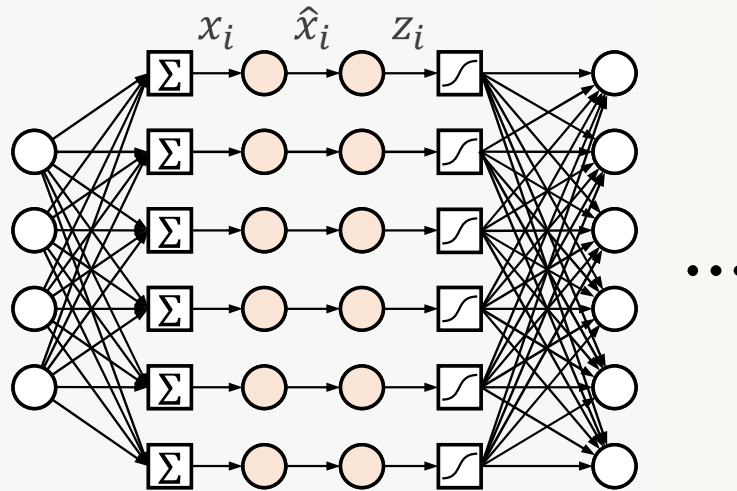
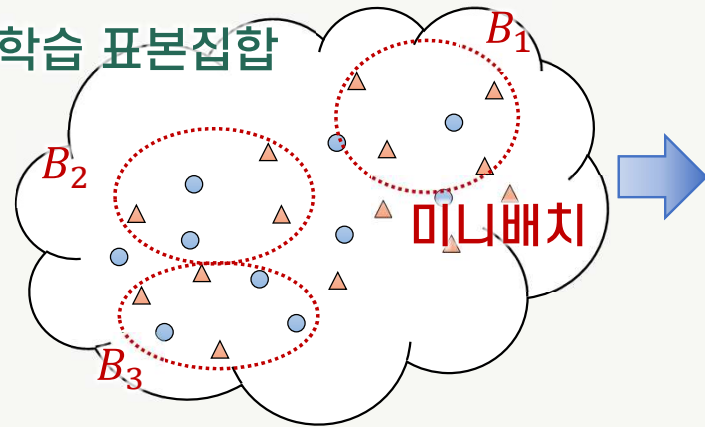
⇒ Sigmoid의 선형 영역에만 국한됨

$$\Rightarrow z^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

1. 배치 정규화의 개념

스케일 및 이동 변환

학습 표본집합



2. 배치 정규화 변환 - 학습 단계

입력

미니배치 $B = \{x_i, i = 1, 2, \dots, N_B\}$

출력

변환된 미니배치 $B_{BN} = \{z_i = \text{BN}_{\gamma\beta}(x_i), i = 1, 2, \dots, N_B\}$

$$\mu_B \leftarrow \frac{1}{N_B} \sum_{i=1}^{N_B} x_i, \quad \sigma_B^2 \leftarrow \frac{1}{N_B} \sum_{i=1}^{N_B} (x_i - \mu_B)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$z_i \leftarrow \gamma \hat{x}_i + \beta$$

➡ 역전파가 진행되는 동안 γ 와 β 에 대한 손실함수의 경사를 구하여 업데이트



3. 배치 정규화 변환 - 추론 단계

● 학습을 마친 모델을 이용한 추론

- 추론 시에는 미니배치가 아닌 개별 입력에 대한 출력을 구해야 함
 - ➔ 전체 학습표본 집합에 대해 추정된 평균 및 분산을 이용하여 입력을 정규화한 후 출력을 구함

$$\hat{x}_i = \frac{x_i - E(x)}{\sqrt{\text{Var}(x) + \epsilon}},$$

$$\text{Var}(x) = \frac{N_B}{N_B - 1} \cdot E_B(\sigma_B^2)$$



4. 텐서플로에서 배치 정규화의 적용

- tf.keras.layers에 제공되는 BatchNormalization 클래스

```
from tensorflow import keras
from tensorflow.keras import Sequential, layers
model = Sequential([keras.Input(shape=(10,)),
                    layers.Dense(32, activation='relu'),
                    layers.Dense(4, activation='softmax')])
```

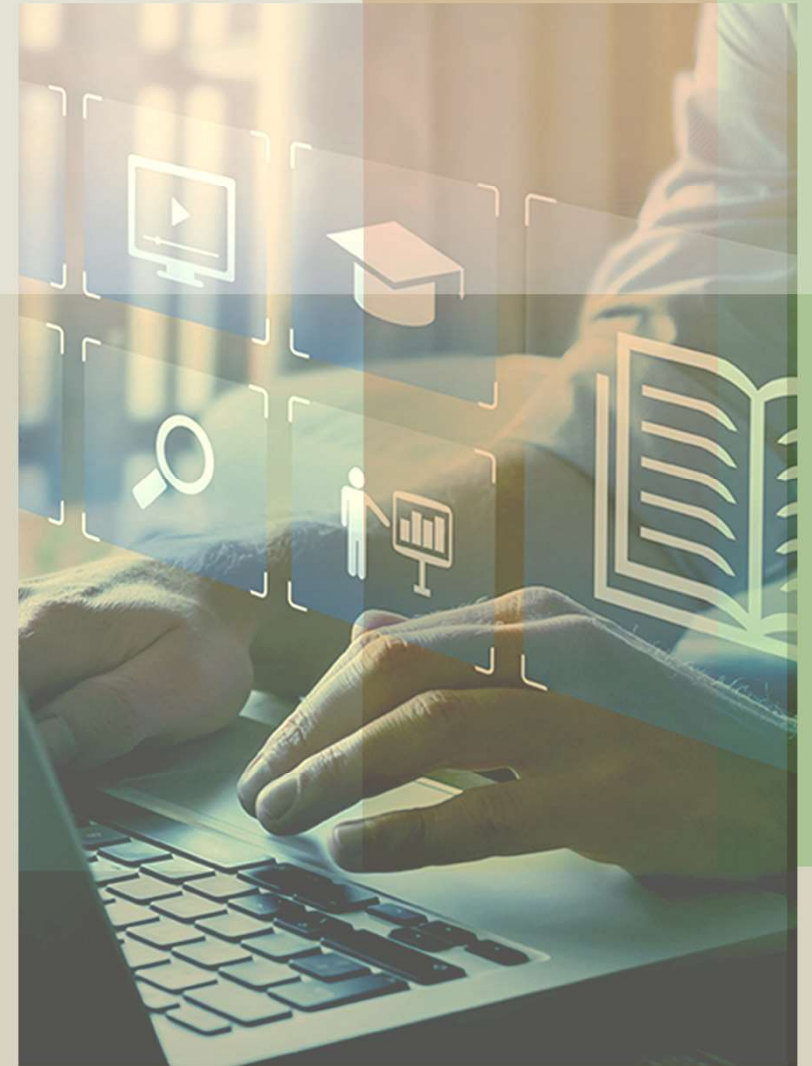


```
from tensorflow import keras
from tensorflow.keras import Sequential, layers
model = Sequential([keras.Input(shape=(10,)),
                    layers.Dense(32),
                    layers.BatchNormalization(),
                    layers.ReLU(),
                    layers.Dense(4, activation='softmax')])
```



04

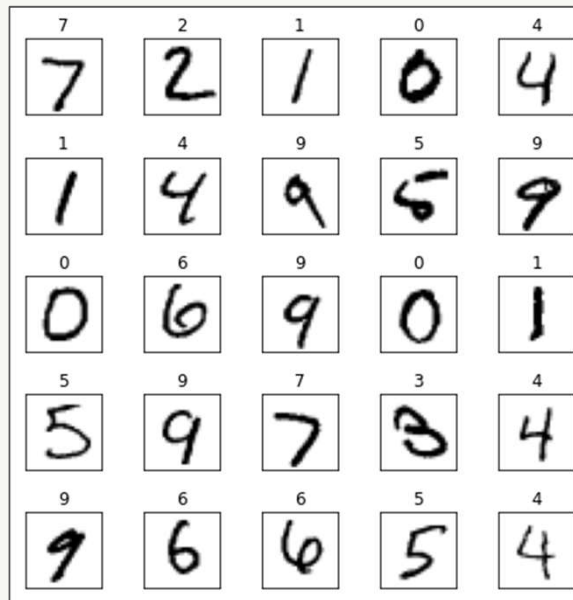
[실습] MNIST 필기 숫자 인식 - 조기 종료



1. MNIST 데이터베이스

● 필기 숫자 이미지를 모아 놓은 데이터베이스

- 60,000개의 훈련용 이미지와 레이블, 10,000개의 평가용 이미지와 레이블로 구성됨
- `tf.keras.dataset.mnist` 모듈의 `load_data()` 함수로 로드할 수 있음



2. MNIST 필기 숫자 인식을 위한 다층 퍼셉트론 모델의 구성 및 학습

4-1 [1] 필요한 패키지 불러오기

```
1 import matplotlib.pyplot as plt
2 from tensorflow.keras import datasets
3 from tensorflow.keras import Sequential, optimizers, callbacks
4 from tensorflow.keras.layers import Flatten, Dense
```

- datasets : MNIST 데이터 집합을 포함한 몇 가지 데이터 집합 모듈 포함
- callbacks : 다양한 콜백 클래스 제공
- Flatten : 고차원 배열 형태의 입력을 1차원으로 변환하는 층



2. MNIST 필기 숫자 인식을 위한 다층 퍼셉트론 모델의 구성 및 학습

4-1 [3] MNIST 데이터 로드 및 정규화

```
1 # MNIST 데이터베이스에서 훈련용 및 평가용 이미지와 레이블 로드
2 (train_imgs, train_labels), (test_imgs, test_labels) = \
3     datasets.mnist.load_data()
4 # 픽셀 값을 0~1 사이로 정규화
5 train_imgs, test_imgs = train_imgs / 255.0, test_imgs / 255.0
```

- 2, 3행 : `tf.keras.datasets.mnist` 모듈의 `load_data()` 함수로 MNIST 데이터 집합을 로드함
- 5행 : 0~255의 값으로 표현된 픽셀 값을 `[0, 1]` 범위의 실수로 정규화함



2. MNIST 필기 숫자 인식을 위한 다층 퍼셉트론 모델의 구성 및 학습

4-1 [4] 2층 피드포워드 네트워크 구성

```
1 # 모델 구성 후 요약 정보 출력
2 model = Sequential([
3     Flatten(input_shape=(28, 28)),
4     Dense(128, activation='sigmoid'),
5     Dense(10, activation='softmax')
6 ])
7 model.summary()
```

- 3행 : 28×28 크기의 숫자 이미지를 1차원의 784개의 값으로 변환하는 층
- 4행 : 128개의 뉴런으로 구성된 완전연결층 구성(은닉층)
 - 활성화함수 : 시그모이드
- 5행 : 10개의 뉴런으로 구성된 완전연결층 구성(출력층)
 - 활성화함수 : 소프트맥스



2. MNIST 필기 숫자 인식을 위한 다층 퍼셉트론 모델의 구성 및 학습

4-1 [5] 모델 컴파일

```
1 # 모델 컴파일
2 model.compile(optimizer=optimizers.SGD(0.1, momentum=0.9),
3               loss='sparse_categorical_crossentropy',
4               metrics=['accuracy'])
```

4-1 [6] 모델 학습

```
1 hist = model.fit(train_imgs, train_labels, epochs=50,
2                 validation_split=0.2)
```

- 학습용 이미지 데이터와 레이블을 이용하여 모델을 훈련함
 - 전체 데이터 중 20%는 검증용으로 사용하고 나머지 80%만 훈련에 사용함
- 학습 과정의 history 객체를 hist에 저장함



3. 학습 이력의 시각화

● history 객체

- history라는 딕셔너리(dictionary) 형의 속성이 있음
 - history의 키 : 훈련 데이터에 대한 'loss'와 fit 메소드의 매개변수 metric에 지정된 항목
- 각 키에 대한 값의 변화 이력을 매 에폭마다 구하여 리스트 형태로 저장함
 - 훈련 단계에서 검증 과정을 수행하였다면 검증 집합에 대한 키와 값 리스트도 포함됨



3. 학습 이력의 시각화

● history 객체

예

```
model.compile(optimizer=optimizers.SGD(0.1, momentum=0.9),  
              loss='sparse_categorical_crossentropy',  
              metrics=['accuracy'])  
hist = model.fit(train_imgs, train_labels, epochs=5,  
                 validation_split=0.2)
```



```
>>> hist.history.keys()  
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])  
>>> hist.history['loss']  
[0.31359416246414185, 0.14779123663902283, 0.10216694325208664,  
0.07658225297927856, 0.05885574221611023]  
>>> hist.history['val_loss']  
[0.1986924409866333, 0.14397640526294708, 0.11934574693441391,  
0.1061328873038292, 0.10009947419166565]
```



3. 학습 이력의 시각화

4-1 [2] 정확도 및 손실함수 시각화 함수 정의

```

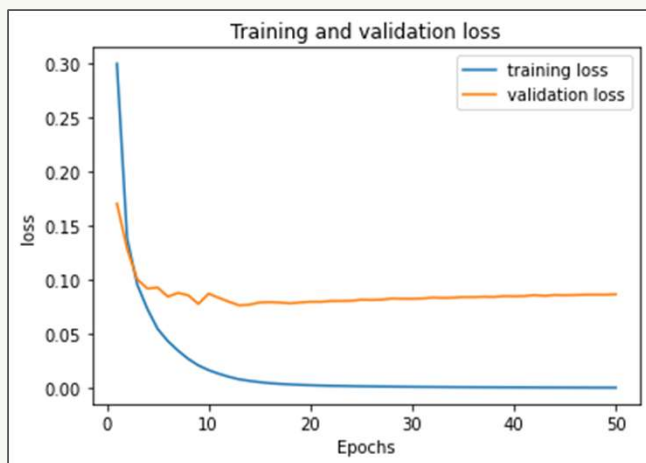
1 def plot_metric(h, metric):
2     train_history = h.history[metric]
3     val_history = h.history['val_'+metric]
4     epochs = range(1, len(train_history) + 1)
5     plt.plot(epochs, train_history)
6     plt.plot(epochs, val_history)
7     plt.legend(['training '+metric, 'validation '+metric])
8     plt.title('Training and validation ' + metric)
9     plt.xlabel("Epochs")
10    plt.ylabel(metric)
11    plt.show()
    
```



3. 학습 이력의 시각화

4-1 [7] 모델의 손실함수 및 정확도 시각화

- 1 `plot_metric(hist, 'accuracy')`
- 2 `plot_metric(hist, 'loss')`



3. 학습 이력의 시각화

4-1 [8] 훈련용 집합 및 검증용 집합에 대한 인식률 출력

```
1 # 훈련 집합을 대상으로 평가
2 _, train_acc = model.evaluate(train_imgs, train_labels)
3 print('훈련 데이터 인식률 = ', train_acc)
4
5 # 테스트 집합을 대상으로 평가
6 _, test_acc = model.evaluate(test_imgs, test_labels)
7 print('테스트 데이터 인식률 = ', test_acc)
```

1875/1875 [=====] - 3s 1ms/step - loss: 0.0197 - accuracy: 0.9957

훈련 데이터 인식률 = 0.9957166910171509

313/313 [=====] - 1s 2ms/step - loss: 0.0845 - accuracy: 0.9801

테스트 데이터 인식률 = 0.9800999760627747



4. 조기 종료

● 조기 종료(early stopping)란?

- 훈련용 데이터에 대해서는 'loss', 'accuracy' 등의 평가 척도가 개선되고 있으나, 검증용 데이터에 대해서는 오히려 성능이 악화되거나 성능 변화가 정체되는 경우 과적합 방지를 위해 지정된 에폭을 모두 실행하지 않고 조기에 학습을 종료함
- EarlyStopping 콜백 인스턴스를 생성하여 모델의 fit 메소드에 콜백으로 등록함



4. 조기 종료

4-1a [6] 조기 종료를 적용한 모델 학습

```
1 early_stop = callbacks.EarlyStopping(monitor='val_loss',  
2                                     min_delta=0,  
3                                     patience=10,  
4                                     verbose=1,  
5                                     restore_best_weights=True)  
6  
7 hist = model.fit(train_imgs, train_labels, epochs=50,  
8                 callbacks=[early_stop], validation_split=0.2)
```



정리하기

- SGD에 모멘텀을 적용하면 이전 업데이트 양을 일정 비율 반영하여 파라미터를 업데이트한다. 이때 경사의 계산 지점을 모멘텀에 해당되는 만큼 이동한 위치에서 함으로써 파라미터 변화의 방향을 개선한 것을 네스테로프 가속 경사(NAG)라고 한다.
- Adagrad와 RMSProp은 파라미터의 변화 방향을 개선하기 위해 학습률을 적응적으로 적용하는 방식이다.
- Adam은 모멘텀과 RMSProp을 결합한 최적화 알고리즘이다.
- 과적합은 훈련에 사용한 데이터에 지나치게 적합하게 학습되어 일반화 오류가 높아지는 현상으로, 규제(regularization)으로 과적합 문제를 개선할 수 있다.



정리하기

- 조기 종료는 모델의 훈련 과정 중 검증용 데이터 집합에 대한 손실이 증가할 경우 조기에 훈련을 종료하여 과적합을 방지하는 기법이다.
- 가중치 감쇠, ℓ_1 정규화, ℓ_2 정규화는 신경망의 가중치의 크기를 억제하는 규제 기술이다.
- 모델을 학습하는 동안 적절한 확률에 따라 뉴런을 무작위로 선택하여 일시적으로 제거하는 드롭아웃을 이용하여 과적합 문제를 개선할 수 있다.



정리하기

- 배치 정규화는 내부 공변량 변화로 인한 심층망의 학습 효율 저하 문제를 개선하기 위한 기술로, 모델의 성능을 높이면서도 학습 속도를 빠르게 할 수 있으며, 가중치의 초기화에 대한 민감도를 낮출 수 있다는 장점이 있다.
- 배치 정규화는 학습 중에는 미니배치 단위로 평균과 분산을 구하여, 추론할 때는 학습표본 집합으로부터 추정한 평균과 분산을 이용하여 정규화한 후 학습된 파라미터에 따라 스케일 및 이동 변환하는 과정을 거쳐 활성화함수를 적용하는 방식으로 동작한다.



다음시간안내

06

합성곱 신경망(1)

