Week2

CH11 - 심층 신경망 훈련하기

아주 복잡한 문제에서는 심층 신경망을 훈련해야 한다.

이는 쉬운 일이 아님

- 까다로운 그레디언트 소실 혹은 폭주가 일어날 수 있음
- 훈련 데이터가 충분하지 않거나 레이블을 만드는 비용이 너무 많음
- 훈련이 극단적으로 느릴 수 있음
- 수백만개의 파라미터를 가질 시 과대적합 위험

11.1 그레디언트 소실과 폭주 문제

알고리즘의 하위층으로 진행될수록 크레디언트가 점점 작아지는 경우가 多

경사하강법이 하위층의 연결 가중치를 변경되지 않은 채로 둔다면 좋은 훈련 Xx

⇒ **그레디언트 소실**

반대로, 그레디언트가 점점 커져 여러 층이 비정상적으로 큰 가중치를 갱신 → 알고리즘 발산

⇒ **그레디언트 폭주**

의심되는 원인으로,

- 로지스틱 활성화 함수
- 당시 인기있던 가중치 초기화 방법 (평균이 0이고 표준편차가 1인 정규분포)

→ 이 조합을 사용했을 때 각 층에서 " 출력 분산 > 입력 분산 "

→ 신경망 위쪽으로 갈수록 층을 지날 때마다 분산이 계속 증가

~~→ *가장 높은 층에서는 활성화 함수가 0 또는 1로 수렴*~~

11.1.1 글로럿과 He 초기화

- 예측할 때는 정방향으로
- 그레디언트를 역전파할 때는 역방향으로
- 신호가 죽거나 폭주 또는 소멸 X
- 적절한 신호가 흐르기 위해서는 각 층의 " 출력에 대한 분산 = 입력에 대한 분산 "
- 역방향에서 층을 통과하기 전과 후의 그레디언트 분산이 동일

입력의 연결 개수: 층의 fan-in

출력의 연결 개수: 층의 fan-out

" 입력 연결 개수 = 출력 연결 개수 " 를 만족하지 않아도 실전에서 잘 작동함 → 글로럿과 벤지오 입증

↓

각 층의 연결 가중치를 아래 식대로 무작위로 초기화

> 💡 평균이 0이고 분산이 $\sigma^2 = \frac{1}{fan_{avg}}$ 인 정규분포
>
> 또는 $\tau = \sqrt{\frac{3}{fan_{avg}}}$ 일 때, $-\tau$와 $+\tau$ 사이의 균등분포
>
> $(fan_{avg} = fan_{in} + fan_{out})$

⇒ **세이비어 초기화 or 글로럿 초기화**

위 식에서 fan_{avg} 을 fan_{in} 으로 바꾸면 → **르쿤 초기화**

$fan_{avg} = fan_{in}$이면 " 르쿤 초기화 = 글로럿 초기화 "

🔥 글로럿 초기화를 사용하면 훈련 속도를 상당히 높일 수 있음

초기화 전략	활성화 함수	σ^2(정규분포)
글로럿	활성화 함수 없음, tanh, logstic, softmax	$1/fan_{avg}$
He	ReLU 함수와 그 변종들	$2/fan_{in}$
르쿤	SELU	$1/fan_{in}$

케라스는 기본적으로 균등분포의 글로럿 초기화 사용

(균등분포의 경우, $\tau = \sqrt{3\sigma^2}$ 으로 계산)

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")

#fan_in 대신 fan_avg기반 균등분포 He 초기화 사용
#Variance Scaling
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',distribution='uniform')
keras.layers.Dense(10, activation='sigmoid', kernel_initialization=he_avg_init)
```

11.1.2 수렴하지 않는 활성화 함수

시그모이드 활성화 함수 → ReLU → 죽은 ReLU 발생

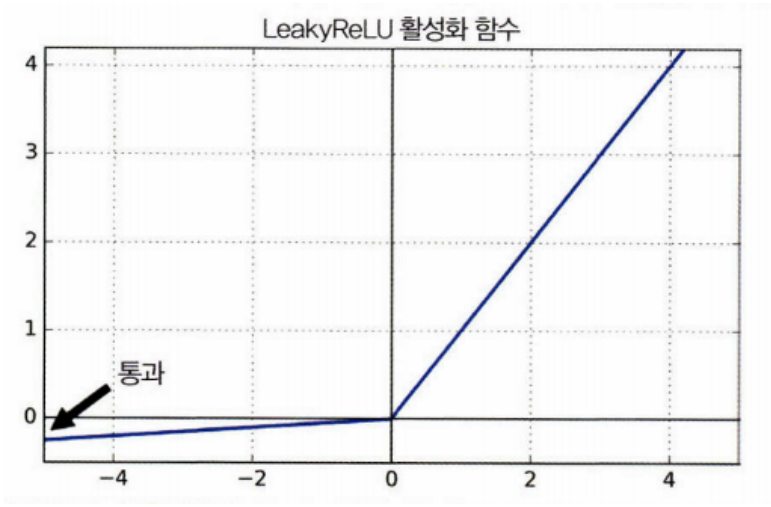
죽은 ReLU란,

훈련하는 동안 뉴런이 0 이외의 값을 출력하지 않는다는 의미

- 큰 학습률 사용 시 뉴런의 절반이 죽어있기도 함
- 가중치의 합이 음수 → 뉴런이 죽게 되는데 → ReLU 함수의 그레디언트 = 0 → 경사하강법 작동 X

<ReLU 변종 등장>

1. `LeakyReLU`

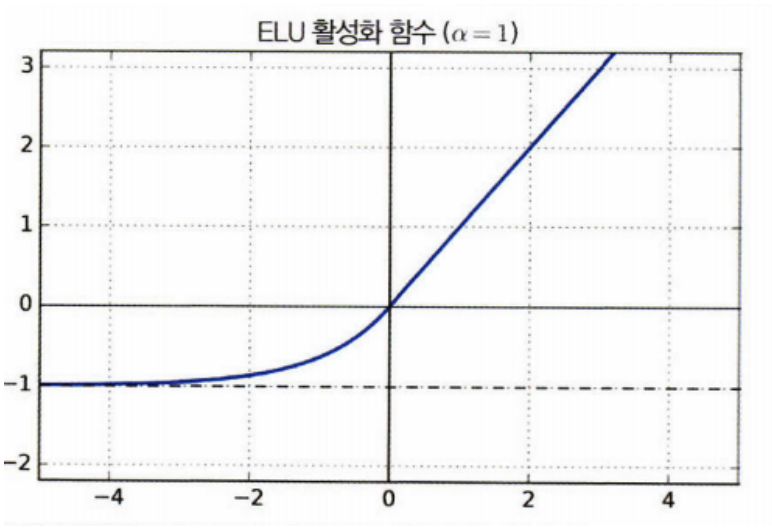


ReLU와 비슷하지만 음수 부분이 작은 기울기를 가진다.

$LeakyReLU_\alpha(z) = max(\alpha z,\ z)$

하이퍼파라미터 α가 함수의 새는 정도를 결정 → 새는 정도: z < 0 일 때, 이 함수의 기울기

2. `RReLU`
   Randomized Leaky ReLU
   α를 무작위로 선택하고 테스트시에는 평균을 사용
   과대적합 방지 - 규제의 역할도 함
3. `PReLU`
   Parametric Leaky ReLU
   α가 훈련하는 동안 학습되는 것
4. `ELU`
   Exponential Linear Unit
   위 모든 함수의 성능보다 뛰어남



$ELU_\alpha(z) =$ z < 0 일 때, $\alpha(exp(z) - 1)$ z ≥ 0 일 때, z

단, 지수함수를 사용하기 때문에 속도가 다른 함수들에 비해 느림

5. `SELU`
   Scaled ELU
   완전 연결 층만 쌓아서 신경망 생성 → 모든 은닉층이 SELU 사용하면 → 네트워크가 자기 정규화 됨
   단, 자기 정규화가 일어나기 위한 조건 충족해야 함

🔥 심층 신경망의 은닉층에서는 어떤 활성화 함수를 써야할까?

SELU > ELU > LeakyReLU(그리고 변종들) > ReLU > tanh > 로지스틱

```
#LeakyReLU 활성화 함수 사용
model = keras.models.Sequential([
    [...]
     keras.layers.Dense(10, kernel_initializer="he_normal")
     keras.layers.LeakyReLU(alpha=0.2)
    [...]
])

#PReLU 사용
model = keras.models.Sequential([
    [...]
     keras.layers.Dense(10, kernel_initializer="he_normal")
     keras.layers.PReLU(alpha=0.2)
    [...]
])

#SELU 사용
layer = keras.layers.Dense(10, activation="selu", kernel_initializer="lecun_normal")
```

11.1.3 배치 정규화

각 층에서 활성화 함수를 통과하기 전이나 후에 모델에 연산을 하나 추가

- 입력을 원점에 맞추고 정규화
- 각 층에서 두 개의 새로운 파라미터로 결괏값의 스케일을 조정 . 이동 시킴
  하나는 스케일 조정, 하나는 이동에 사용

`BatchNormalization`
: 입력 평균과 표준편차의 이동 평균을 사용해 훈련하는 동안 최종 통계를 추정 → 자동으로 수행

⇒ 그레디언트 소실 문제 크게 감소

- 수렴성을 가진 활성화 함수 사용 가능
- 가중치 초기화에 덜 민감
- 큰 학습률 사용 가능 → 속도 향상

규제의 역할도 함 → 다른 규제 기법의 필요성을 줄여줌

그러나, 모델의 복잡도가 커짐

<케라스로 배치 정규화 구현하기>

```
import tensorflow as tf
from tensorflow import keras

#케라스로 배치 정규화 구현하기
model = keras.models.Sequential([
                                keras.layers.Flatten(input_shape=[28,28]),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dense(10, activation="softmax")
])

>>> model.summary() #배치 정규화 층은 입력마다 네 개의 파라미터가 추가됨을 알 수 있다.
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
batch_normalization (BatchNo (None, 784)               3136
_________________________________________________________________
dense (Dense)                (None, 300)               235500
_________________________________________________________________
batch_normalization_1 (Batch (None, 300)               1200
_________________________________________________________________
```

```
dense_1 (Dense)              (None, 100)               30100
_________________________________________________________________
batch_normalization_2 (Batch (None, 100)               400
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
_________________________________________________________________

#첫 번째 배치 정규화 층의 파라미터
#두 개는 역전파로 훈련되고 두 개는 훈련되지 않는다.
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization/gamma:0', True),
 ('batch_normalization/beta:0', True),
 ('batch_normalization/moving_mean:0', False),
 ('batch_normalization/moving_variance:0', False)]
```

활성화 함수 이전에 배치 정규화 층을 추가하는 것이 더 좋다 ?

→ 데이터 셋에 따라 다르다 !

```
model = keras.models.Sequential([
                                keras.layers.Flatten(input_shape=[28,28]),
                                keras.layers.BatchNormalization(),
                                keras.layers.Dense(300, kernel_initializer="he_normal",use_bias=False),
                                keras.layers.BatchNormalization(),
                                keras.layers.Activation("elu"),
                                keras.layers.Dense(100, kernel_initializer="he_normal", use_bias=False),
                                keras.layers.BatchNormalization(),
                                keras.layers.Activation("elu"),
                                keras.layers.Dense(10, activation="softmax")
])
```

`BatchNormalization` 는 조정할 하이퍼파라미터가 적다.

기본 값이 잘 작동하지만 → 이따금 momentum 매개변수를 변경해야 할 수 있음

✅ momentum

`BatchNormalization` 층이 지수 이동 평균을 업데이트할 때 사용

$$\hat{v} \leftarrow \hat{v} * momentum + v * (1 - momentum)$$

- 적절한 모멘텀 값은 1에 가까움

✅ axis

- 정규화할 축을 결정
- 기본값은 -1

```
#훈련 도중과 훈련이 끝난 후에 수행하는 계산이 다름
#훈련하는 동안: 배치 통계 사용
#훈련이 끝난 후: 최종 통계(이동 평균의 마지막 값) 사용

class BatchNormalization(keras.layers.Layer):
    [...]
    def call(self, inputs, training=None):
        [...]
```

⇒ 최근 BatchNormalization보다 좋은 방법 (Fixup 가중치 초기화)이 등장 - 추가 연구 필요...

11.1.4 그레디언트 클리핑

그레디언트 클리핑: 역전파될 때 일정 임계값을 넘어서지 못하게 그레디언트를 잘라내는 것

- 순환 신경망에서 많이 사용 (배치 정규화 적용이 어려워서)

```
#옵티마이저 생성 시 clipvalue 와 clipnorm 매개변수 지정
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizers)
```

그레디언트 벡터의 모든 원소를 -1.0 ~ 1.0 사이의 값으로 클리핑

그레디언트 클리핑이 그레디언트 벡터의 방향을 바꾸지 못하게 하면,

→ clipnorm을 지정하여 노름으로 클리핑

11.2 사전훈련된 층 재사용하기

전이 학습: 비슷한 유형의 문제를 처리한 신경망의 하위층을 재사용하는 것

- 훈련 속도 상승
- 훈련 데이터 크기 감소

재사용할 층 개수 선정이 중요

1. 재사용하는 층 동결
2. 모델 훈련, 성능 평가
3. 맨 위 한두개의 은닉층 동결 해제 → 역전파로 가중치 조정
   (훈련 데이터가 많을수록 많은 층의 동결을 해제할 수 있다.)
   (재사용 층 동결 해제 시, 학습률을 줄여야 함 → 섬세한 가중치 튜닝 가능)
4. 좋은 성능이 안나온다 → 상위 은닉층 제거 후 남은 은닉층 다시 동결

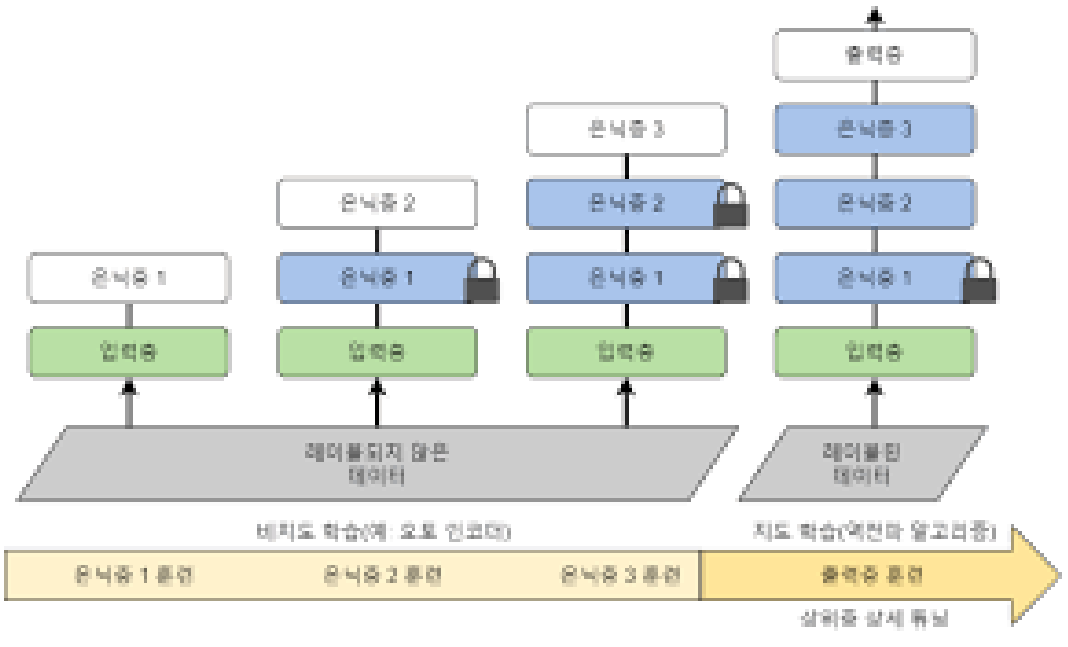
⇒ 적절한 개수를 찾을 때까지 반복

11.2.1 케라스를 사용한 전이 학습

```
#모델 A의 출력층만 제외하고 모든 층 재사용
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Squential(model_A.layers[:-1])
model_B_on_a.add(keras.layers.Dense(1, activation = "sigmoid"))

#모델 A는 모델 B_on_A 훈련 시 영향을 받음
#구조 복제 -> 가중치 복사
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())

#새로운 층에게 가중치를 학습할 시간 부여
#재사용된 층 동결
for layers in model_B_on_A.layers[:-1]:
    layers.trainable = False

model_B_on_A.compile(loss="binary_crossentropy",
                     optimizer="sgd", metrics=["accuacy"]) #층을 동결 혹은 동결 해제 후 컴파일

#동결 해제
#모델 다시 컴파일
history = model_B_on_A.fit(x_train_B, y_train_B, epochs=4,
                           validation_data=(x_valid_B, y_valid_B))

for layers in model_B_on_A.layers[:-1]:
    layers.trainable = True

optimizer = keras.optimizer.SGD(lr=1e-4) #동결 해제 후 학습률을 낮춰줘야 좋음. 기본 학습률은 1e-2

model_B_on_A.compile(loss="binary_crossentropy",
                     optimizer=optimizer, metrics=["accuacy"])

history = model_B_on_A.fit(x_train_B, y_train_B, epochs=16,
                           validation_data=(x_valid_B, y_valid_B))

#모델 평가
model_B_on_A.evaluate(x_test_B, y_test_B)
```

⇒ 전이 학습을 했을 때, 오차율이 약 5배 줄어듬
이 학습에는 속임수가 있다...? → 전이 학습은 작은 완전 연결 네트워크 층에서는 작동 X
심층 합성곱 신경망에서 잘 작동함 !

11.2.2 비지도 사전훈련



오토인코더나 GAN 사용

11.2.3 보조 작업에서 사전훈련

ex.

얼굴 인식 시스템을 만들자 ! → 레이블된 훈련 데이터가 충분 X

일단 인터넷에 있는 얼굴 이미지를 많이 긁어오자 ! → 모델 훈련 → 두 개의 다른 이미지가 같은 사람인지 아닌지 구분 가능
(첫 번째 신경망 훈련)

⇒ 이런 방식으로 반복 훈련

⇒ 적은 양의 훈련 데이터에서 얼굴을 잘 구분하는 분류기 생성 가능

11.3 고속 옵티마이저

훈련 속도를 크게 높일 수 있는 방법 중 하나

표준적인 경사 하강법 옵티마이저 대신 더 빠른 옵티마이저 사용

- 모멘텀 최적화
- 네스테로프 가속 경사
- AdaGrad
- RMSProp
- Adam
- Nadam

11.3.1 모멘텀 최적화

경사 하강법은 가중치 θ를 바로바로 갱신하기 때문에 이전 그레디언트가 얼마였는지 중요하지 X

모멘텀 최적화는,

- 이전 그레디언트가 얼마였는지가 중요함 !
- 매 반복에서현재 그레디언트를 모멘텀 벡터에 더하고 이 값을 빼는 방식
  그레디언트를 가속도로 사용
- 모멘텀 하이퍼파라미터 : β
- 모멘텀 알고리즘
  $m \leftarrow \beta m - \eta \nabla_\theta J(\theta)$

  $\theta \leftarrow \theta + m$
- 일반적으로 모멘텀 값은 0.9

⇒ (부호는 무시하고) $\frac{1}{1-\beta}$ 를 학습률 η를 곱한 그레디언트에 곱한 것

⇒ β = 0.9 면, 경사하강법보다 10배 빠르게 진행된다는 의미

```
optimizer = keras.optimizer.SGD(lr=0.001, momentum=0.9)
#튜닝할 하이퍼파라미터 증가
#그럼에도 대부분 경사 하강법보다 빠름
```

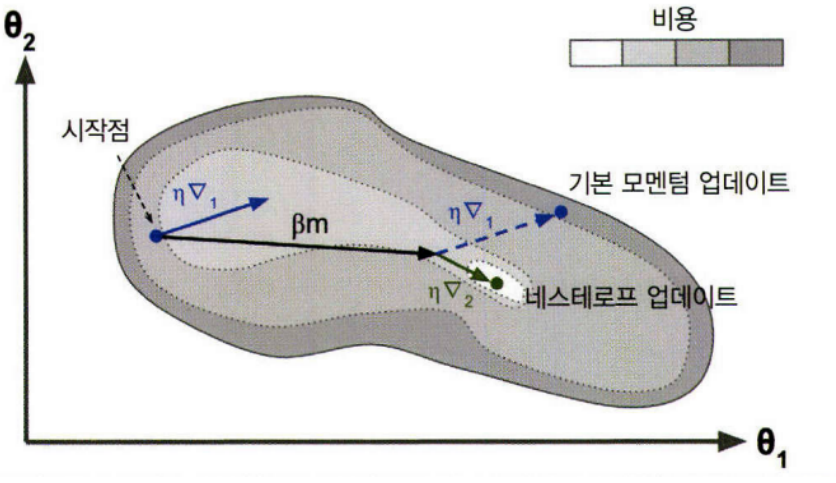
11.3.2 네스테로프 가속 경사

현재 위치가 θ 가 아니라 모멘텀 방향으로 조금 앞선 $\theta + \beta m$**에서 비용함수의 그레디언트 계산**

- NAG 알고리즘
  $m \leftarrow \beta m - \eta \nabla_\theta J(\theta + \beta m)$

  $\theta \leftarrow \theta + m$

```
optimizer = keras.optimizer.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

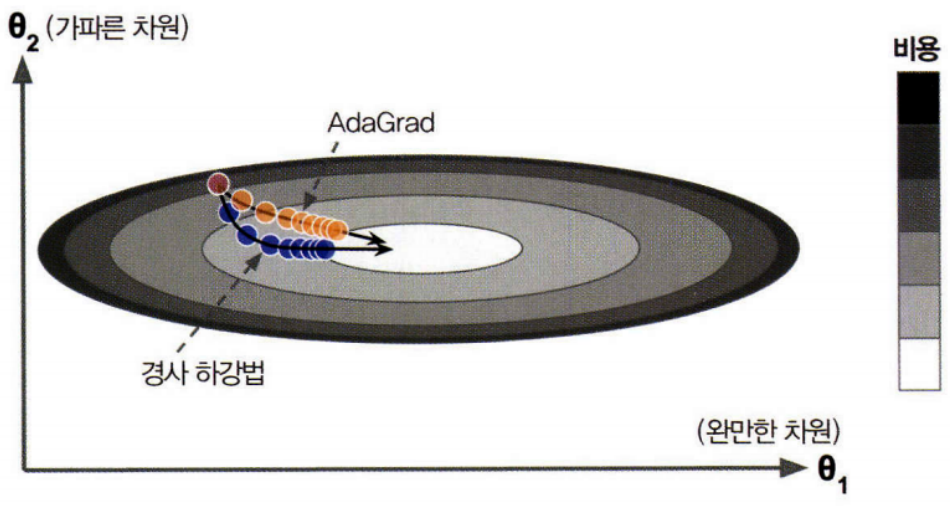


11.3.3 AdaGrad

가장 가파른 차원을 따라 그래디언트 벡터의 스케일을 감소

- AdaGrad 알고리즘
  $s \leftarrow s + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$

  $\theta \leftarrow \theta - \nabla_\theta J(\theta) \oslash \sqrt{s + \varepsilon}$
- **적응적 학습률:**
  위 알고리즘은 학습률을 감소시켜줌
  경사가 완만한 차원보다 가파른 차원에 대해 더 빠르게 감소
  전역 최적점 방향으로 더 곧장 가도록 갱신 → **학습률 η 를 덜 튜닝해도 됨**



※ 학습이 너무 일찍 멈출 수 있음 → 심층 신경망에는 사용 X

11.3.4 RMSProp

AdaGrad의 단점을 보완하기 위해,

가장 최근 반복에서 비롯된 그레디언트만 누적

- RMSProp 알고리즘
  $s \leftarrow \beta s \ + (1-\beta)\nabla_\theta J(\theta) \ \otimes \nabla_\theta J(\theta)$

  $\theta \leftarrow \theta \ - \ \eta\nabla_\theta J(\theta) \ \oslash \ \sqrt{s+\varepsilon}$

```
optimizer = keras.optimizer.RMSProp(lr=0.001, rho=0.9)
```

11.3.5 Adam과 Nadam 최적화

적응적 모멘트 추정(Adaptive Moment estimation)

- 모멘텀 최적화 + RMSProp

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1-\beta_1) \bigtriangledown_\theta J(\theta)$$

$$\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2) \bigtriangledown_\theta J(\theta) \otimes \bigtriangledown_\theta J(\theta)$$

$$\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1-\beta_1^t}$$

$$\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\beta_2^t}$$

$$\theta \leftarrow \theta + \eta \hat{m} \oslash \sqrt{\hat{\mathbf{s}} + \varepsilon}$$

Adam 알고리즘

- t는 반복 횟수
- 단계 1,2,5는 모멘텀 최적화, RMSProp과 비슷
- 단계 3,4에서,
  m,s는 0으로 초기화되기 때문에 훈련 초기에 0쪽으로 치우치게 됨
  → 이 두 단계가 훈련 초기에 m,s의 값을 증폭시키는 데 도움을 줌
- 반복 초기에는 m,s를 증폭시켜주지만 반복이 많이 진행되면 단계 3,4의 분모는 1에 가까워져 거의 증폭되지 않음
- 모멘텀 감쇠 하이퍼파라미터 β1은 보통 0.9
  스케일 감쇠 하이퍼파라미터 β2는 0.999로 초기화

```
optimizer = keras.optimizer.SGD(lr=0.001, beta_1=0.9, beta_2=0.999)
```

<AdaMax>

- Adam은 시간에 따라 감쇠된 그레디언트의 l2 노름으로 파라미터 업데이트의 스케일을 낮춤
  (l2 노름은 제곱 합의 제곱근)
- Adamax는 l2 노름을 l∞ 노름으로 바꾼다
- 알고리즘은 Adam 알고리즘의 2단계를 $s \leftarrow max(\beta_2 s, \nabla_\theta J(\theta))$ 로 바꾸고 4단계 삭제
- AdaMax가 Adam보다 안정적

<Nadam>

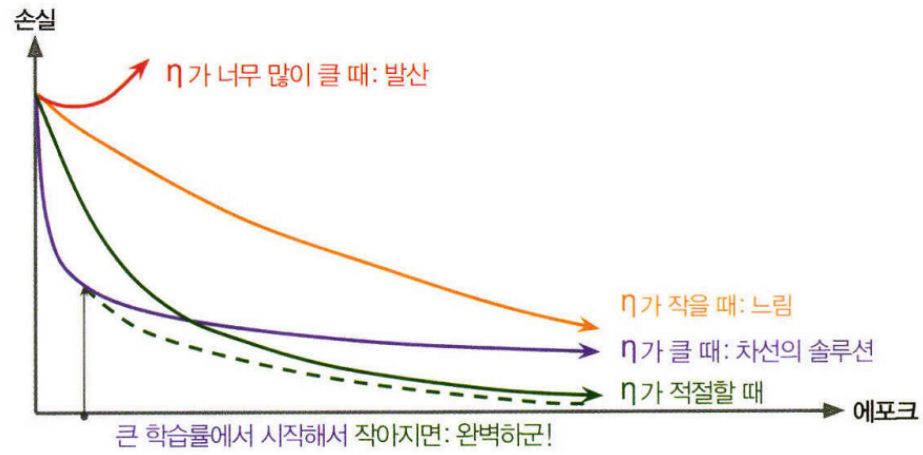
- Adam 옵티마이저에 네스테로프 기법을 더한 것
- Adam보다 조금 더 빠르게 수렴
- 일반적으로 Nadam이 Adam보다 성능이 좋았지만 이따금 RMSProp이 나을 때도 있다.

⇒ 1차 편미분에만 의존
2차 편미분(헤시안) → 계산이 너무 느림

클래스	수렴 속도	수렴 품질
SGD	*	***
Momentum	**	***
Nesterov	**	***
AdaGrad	***	* (너무 일찍 멈춤)
RMSProp	***	** 또는 ***
Adam	***	** 또는 ***
Nadam	***	** 또는 ***
AdaMax	***	** 또는 ***

옵티마이저 비교(*=나쁨, **=보통, ***=좋음)

11.3.6 학습률 스케줄링



큰 학습률로 시작하고 학습 속도가 느려질 때,

학습률을 낮추면 최적의 고정 학습률 보다 좋은 솔루션을 더 빨리 발견할 수 있음 → 학습률 스케줄

▼ ***널리 사용하는 학습 스케줄***

- 거듭제곱 기반 스케줄링

  반복 횟수 t에 대한 함수 $\eta(t) = \eta_0 \ / \ (1 + t/s)^c$

  η_0: 초기 학습률, c: 거듭제곱 수, s: 스텝 횟수

  처음에는 빠르게 감소하다가 점점 느려짐

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

- 지수 기반 스케줄링

  학습률을 $\eta(t) = \eta_0 \ 0.1^{t/s}$ 로 설정

  스텝마다 10배씩 감소

```
#현재 에포크를 받아 학습률을 반환하는 함수
def exponential_decay_fn(epoch):
    return 0.01 * 0.1 ** (epoch / 20)

#클로저를 반환하는 함수
def exponential_decay(lr0, s):
    def exponential_decay_fn(epoch):
        return lr0 * 0.1 ** (epoch / s)
    return exponential_decay_fn

exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn) #콜백 생성
history = model.fit(x_train_scaled, y_train, [...], callbacks = [lr_scheduler])
```

- 구간별 고정 스케줄링

  일정 횟수의 에포크 동안 일정 학습률을 사용 → 그 다음 또 다른 횟수의 에포크 동안 작은 학습률 사용

  많은 조합 시도를 해봐야 함

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

- 성능 기반 스케줄링

  매 N 스텝마다 검증 오차 측정 → 오차가 줄지 않으면, λ배만큼 학습률 감소

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

- 1사이클 스케줄링

  훈련 절반 동안 초기 학습률 η_0을 선형적으로 η_1까지 증가

  → 나머지 절반 동안 선형적으로 학습률을 η_0까지 다시 줄인다.

11.4 규제를 사용해 과대적합 피하기

과대적합 방지를 위해 규제는 필요하다

11.4.1 L1과 L2 규제

신경망의 연결 가중치 제한하기 위해 L2 규제 사용

희소 모델을 만들기 위해 L1 규제 사용

```
#연결가중치에 규제강도 0.01의 L2규제
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

✅ 일반적으로 동일한 매개변수 값을 반복하는 경우 多

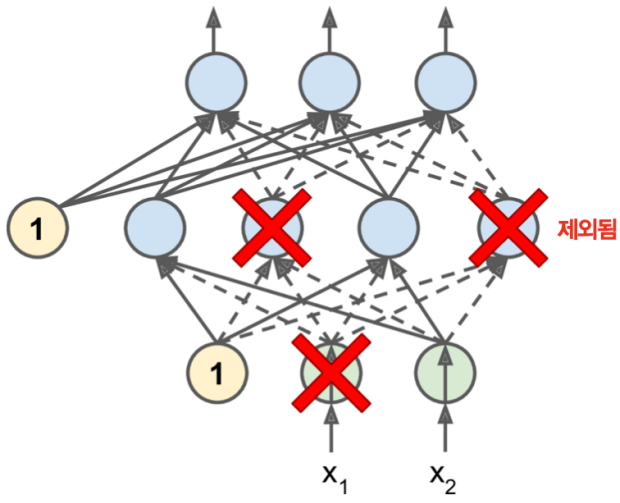
→ 네트워크의 모든 은닉층에 동일한 활성화 함수, 동일한 초기화 전략을 사용하거나
모든 층에 동일한 규제를 적용하기 때문

→ 버그 발생이 쉬워짐 - - - **반복문을 사용해 코드를 refactoring** 또는 **파이썬의 functool.partial() 함수 사용**

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax")
])
```

11.4.2 드롭아웃

드롭아웃될 확률 = **p =드롭아웃 비율**(0.1 ~ 0.5 사이로 지정)



→ 각 훈련 반복에서 (출력층을 제외하고) 하나 이상의 층에 있는 모든 뉴런의 일부를 제거
이런 뉴런은 이 반복에서 0을 출력

1-p: 보존 확률

→ 훈련이 끝난뒤 각 입력 연결 가중치에 곱해주는 것

→ 드롭아웃을 한 만큼 남은 뉴런이 배로 많은 입력 신호를 받기 때문에

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
#드롭아웃을 빼고 훈련 손실 평가해야 함
#최신 신경망에거는 마지막 은닉층 뒤에만 드롭아웃 사용
#알파 드롭아웃: 자기 정규화 기능 오류 문제 방지
```

11.4.3 몬테 카를로 드롭아웃

훈련된 드롭아웃 모델을 재훈련하지 않고 성능을 향상 시킴

```
y_probas = np.stack([model(X_test_scaled, training=True) #매개변수 training: 드롭아웃층 활성화 여부
                     for sample in range(100)])
y_proba = y_probas.mean(axis=0)
y_std = y_probas.std(axis=0)

#모델이 훈련하는 동안 다르게 작동하는 층이 있다면
#드롭아웃층 상속
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
#알파 드롭아웃층 상속
class MCAlphaDropout(keras.layers.AlphaDropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

11.4.4 맥스-노름 규제

$\| w \|_2 \leq r$ 이 되도록 제한

(w는 입력 연결 가중치, r은 맥스-노름 하이퍼파라미터, $\| \cdot \|_2$는 L2 노름)

매 훈련 스텝이 끝나고 $\| w \|_2$ 계산 →필요하면, w 스케일 조정

r을 줄이면 규제의 양이 증가하여 과대적합을 감소

```
keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal",
                           kernel_constraint=keras.constraints.max_norm(1.))
```

11.5 요약 및 실용적인 가이드라인

기본 DNN 설정

하이퍼파라미터	기본값
커널 초기화	He 초기화
활성화 함수	ELU
정규화	얕은 신경일 경우 없음. 깊은 신경망이라면 배치 정규화
규제	조기 종료 (필요하면 l_2 규제 추가)
옵티마이저	모멘텀 최적화 (또는 RMSProp이나 Nadam)
학습률 스케줄	1사이클

자기 정규화를 위한 DNN 설정

하이퍼파라미터	기본값
커널 초기화	르쿤 초기화
활성화 함수	SELU
정규화	없음 (자기 정규화)
규제	필요하다면 알파 드롭아웃
옵티마이저	모멘텀 최적화 (또는 RMSProp이나 Nadam)
학습률 스케줄	1사이클

11.6 연습문제