11장 심층신경망 훈련

감사의 글

자료를 공개한 저자 오렐리앙 제롱에게 깊은 감사를 드립니다. 이와 더불어 한빛미디어로부터 강의준비에 필요한 자료를 지원받았음을 밝히며, 이에 대해 진심어린 감사를 전합니다.

- 심층신경망
    - 수백 개의 뉴련으로 구성된 10개 이상의 층을 사용하는 신경망
    - 예제: 고해상도 이미지에서 수백 종류의 물체 감지

- 심층신경망 훈련 중 발생하는 문제
    - 그레이디언트 소실/폭주: 심층신경망의 아래족으로 갈수록 그레이디언트가 점점 더 작아지거나 커지는 현상
    - 훈련 데이터 부족 또는 너무 비싼 레이블 작업
    - 극단적으로 느린 훈련과정
    - 과대적합: 수백만개의 파라미터에 의해 과대적합 가능성 매우 큼. 특히 훈련 샘플이 충분하지 않거나 잡음이 많은 경우 그러함.

주요 내용

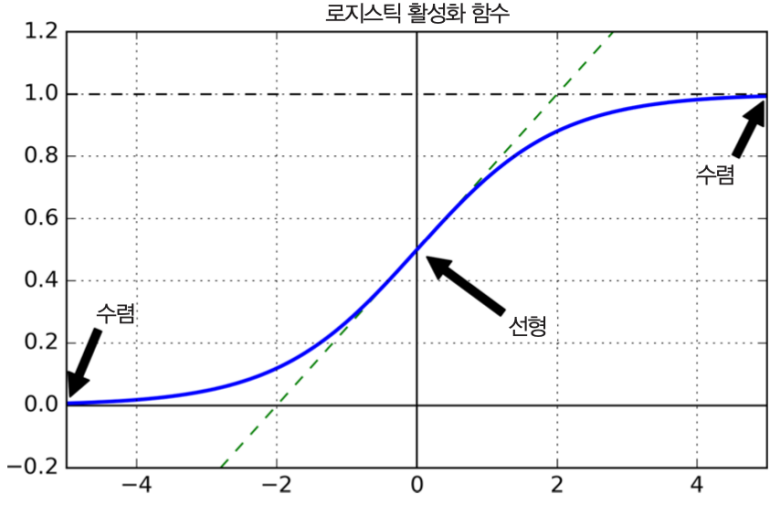
- 언급된 문제 해결책 제시

그레이디언트 소실/폭주 문제

- (10장에서 설명한) 역전파 알고리즘: 출력층에서 입력층으로 오차 그레이디언트를 전파

- 하위층으로 갈 수록 그레이디언트 소실/폭주 문제 발생 자주 발생.
    - 그레이디언트 소실: 최적의 모델로 수렴하지 않음.
    - 그레이디언트 폭주: 알고리즘 발산.

- 원인: 활성화 함수와 가중치 초기화를 위해 아래 조합을 선택하였기 때문임.
    - 활성화 함수: 로지스틱 활성화 함수
    - 가중치 초기화: 표준정규분포 활용



- 2010년에 위 사실이 알려질 때까지 심층신경망은 사실상 방치되었음.
- 하지만 이후 급속도로 발전하여 2016년 알파고와 이세돌 바둑대전까지 가능해짐.
- 현재 데이터과학과 관련된 모든 분야에서 기존에 해결 불가능한 문제들을 해결하고 있음.
- 하나의 연구분야가 아니라 컴퓨터 프로그래밍의 필수 기법으로 자리잡음.

초기화 방식 선택

- 층에 사용되는 활성화 함수의 종류에 따라 아래 세 가지 초기화 방식 중 하나 선택
- 글로로(Glorot) 초기화
- 르쿤(LeCun) 초기화
- 헤(He) 초기화

글로로(Glorot) 초기화

- 팬-인/팬-아웃
    - fan-in(팬-인, fan_{in}): 층에 들어오는 입력 수
    - fan-out(팬-아웃, fan_{out}): 층에서 나가는 출력 수

$$fan_{\text{avg}} = \frac{fan_{\text{in}} + fan_{\text{out}}}{2}$$

- 글로로 초기화(정규분포 활용)
  - 평균(μ) = 0
  - 분산(σ^2) = $\frac{1}{fan_{\text{avg}}}$

- 글로로 초기화(균등분포 활용)
  - $-r$과 r 사이의 균등분포

$$r = \sqrt{\frac{3}{fan_{\text{avg}}}}$$

르쿤(LeCun) 초기화

- 글로로 초기화 정의에서 fan_{avg} 를 fan_{in} 으로 대체.

헤(He) 초기화

- 정규분포 활용 초기화:

$$\sigma^2 = \frac{2}{fan_{\text{in}}}$$

- 균등분포 활용 초기화:

$$r = \sqrt{3\sigma^2}$$

활성화 함수와 초기화 방식

- 층을 생성할 때 사용하는 활성화 함수에 따라 다른 가중치 초기화 방식을 사용해야 함.

초기화 전략	활성화 함수	정규분포 초기화	균등분포 초기화
Glorot 초기화	활성화 함수 없는 경우, 하이퍼볼릭 탄젠트, 로지스틱, 소프트맥스	glorot_normal	glorot_uniform
He 초기화	ReLU 함수와 그 변종들	he_normal	he_uniform
LeCun 초기화	SELU	lecun_normal	lecun_uniform

- 케라스의 기본 초기화 설정값: `glorot_uniform`

예제

- 층을 만들 때 정규분포를 이용한 He 초기화를 사용하고자 하는 경우

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

예제

- fan_{in} 대신 fan_{out} 기반의 균등분포 He 초기화를 사용하고자 할 경우
  - `VarianceScaling` 클래스 활용

```
init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                          distribution='uniform')
keras.layers.Dense(10, activation="relu", kernel_initializer=init)
```

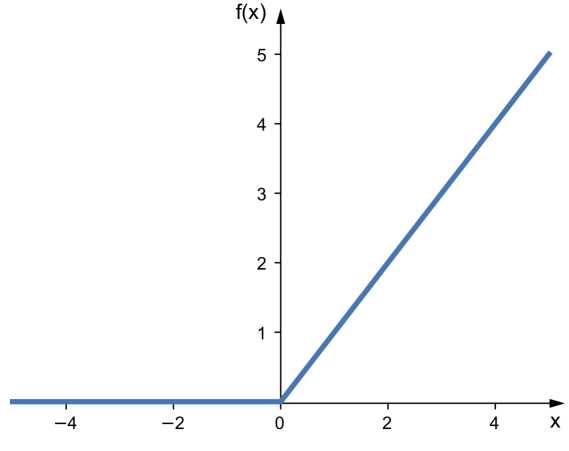
활성화 함수 선택

- 심층신경망의 층에서 사용되는 활성화 함수는 시그모이드 함수보다 아래 함수들이 보다 좋은 성능을 발휘함.

- ReLU
- LeakyReLU
- RReLU
- PReLU
- ELU
- SELU

ReLU

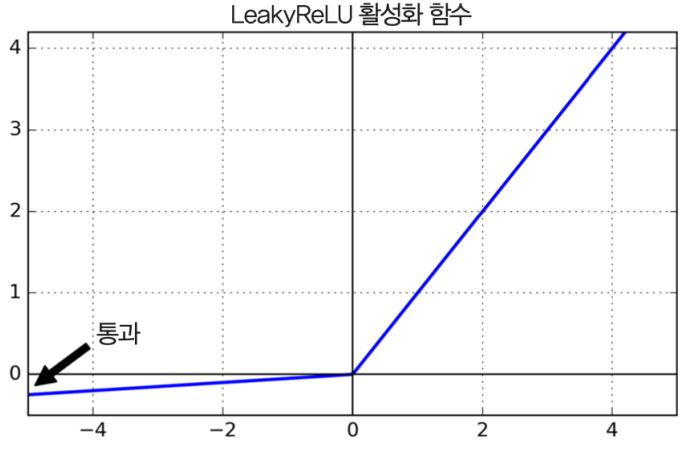
- 2010년도에 소개됨.
- $\mathrm{ReLU}_{\alpha}(z) = \max(0, z)$
- 완벽하지 않음
- 입력의 가중치 합이 음수가 되면 뉴런이 죽게 되어 경사하강법이 제대로 작동하지 않게됨.



LeakyReLU

- 2014년에 소개됨
- $\text{LeakyReLU}_{\alpha}(z) = \max(\alpha z, z)$
    - α: 새어 나가는 정도를 결정하는 하이퍼파라미터

- ReLU 보다 좋은 성능 발휘
    - 기본값: $\alpha = 0.1$
    - $\alpha = 0.2$로 할 때 좀 더 성능 좋아짐.



RReLU

- α를 주어진 범위에서 무작위로 선택하는 LeakyReLU
- 꽤 잘 작동함
- 과대적합을 줄이는 규제역할도 수행하는 것으로 보임

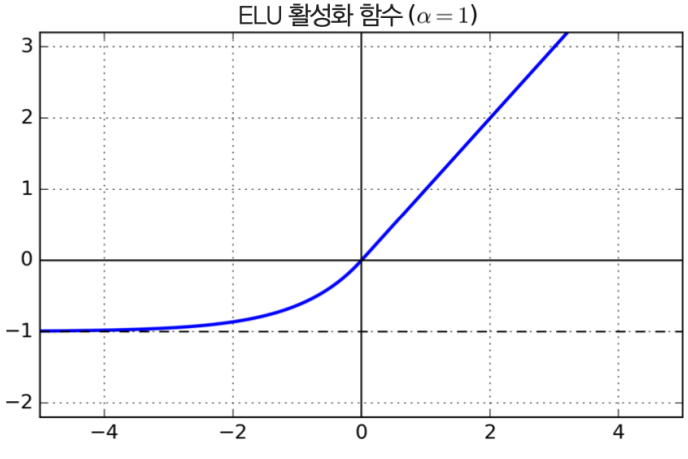
PReLU

- 역전파 과정에서 α값도 자동 조정됨
- 대규모 이미지 데이터셋에서 ReLU 보다 성능 좋음.
- 소규모 데이터세에서는 과대적합 위험성 존재.

ELU

- 2015년도에 소개됨.
- 앞서 언급된 ReLU 변종들보다 성능 좋은 것으로 보임.
- 훈련 시간 줄어듦.

$$\mathrm{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0, \\ z & \text{if } z \geq 0. \end{cases}$$



- ELU 함수의 장단점
    - 수렴 속도 빠름
    - 계산이 느림. 지수함수가 사용되기 때문.
    - 따라서 테스트할 때 ReLU를 사용하는 신경망보다 느림.

SELU

- 2017년도에 소개됨.
- 스케일이 조정된 ELU 활성화 함수의 변종

- 아래 조건 하에서 뛰어난 성능 발휘함
    - 입력특성이 표준화(평균 0, 표준편차 1)되어야 함.
    - 모든 은닉층에서 르쿤 정규분포 초기화 사용
    - 일렬로 쌓은 층만 사용해야 함.
    - 모든 층이 완전연결층이어야 함.

- 경우에 따라 합성곱 신경망에서도 좋은 성능 발휘

심층신경망의 은닉층에 대한 활성화 함수 선택 가이드라인

- 일반적 우선순위: SELU > ELU > LeakyReLU 와 기타 변종들 > ReLU > 로지스틱
- 신경망이 자기정규화(self-normlizing)하지 않은 경우: SELU 보다 ELU 선호
- 시간과 컴퓨팅파워가 충분한 경우: 교차검증을 이용하여 여러 활성화함수 평가

- 실행속도가 중요한 경우: LeakyReLU
- 과대적합 발생하는 경우: RReLU
- 훈련세트가 매우 큰 경우: PReLU
- 훈련속도가 중요한 경우: ReLU
    - 이유: 기존에 많은 라이브러리와 가속기가 개발되었기 때문.

예제

- LeakyReLU 활성화 함수 사용
- LeakyReLU 층을 만들고 모델에서 적용하려는 층 뒤에 추가

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

예제

- PReLU 활성화 함수 사용
- PReLU 층을 만들고 모델에서 적용하려는 층 뒤에 추가

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

예제

- RReLU 활성화 함수 사용
- 케라스에서 지원하지 않지만 간단하게 구현 가능. (12장 연습문제 참조)

예제

- SELU 활성화 함수 사용
- 층을 만들 때 `activation="selu"` 와 `kernel_initializer="lecun_normal"` 지정.

```
model = keras.models.Sequential()
model.add(keras.layers.Flatten(input_shape=[28, 28]))
model.add(keras.layers.Dense(300, activation="selu",
                             kernel_initializer="lecun_normal"))
for layer in range(99):
    model.add(keras.layers.Dense(100, activation="selu",
                                 kernel_initializer="lecun_normal"))
model.add(keras.layers.Dense(10, activation="softmax"))
```

배치정규화

- (ELU 또는 다른 ReLU 변종) + (He 초기화)
    - 훈련 초기단계의 그레이디언트 소실/폭주 문제 해결
    - 하지만 훈련 중 동일 문제 재발생 방지 보장 못함

- 배치정규화(batch normalization, BN) 기법 활용 가능

배치정규화

- 각 층의 활성화함수 통과 이전/이후 정규화 연산 하나 추가

- 사용되는 정규화 연산
    - 평균: 0으로 조정
    - 분산: 스케일 조정

- 모든 심층신경망의 성능 크게 향상시킴

- 그레이디언트 소실/폭주 문제를 감소시켜서 하이퍼볼릭 탄젠트 또는 로지스틱 활성화 함수 사용 가능.

- 가중치 초기화에 덜 민감해짐.

- 규제 역할도 수행하여 다른 규제의 필요성 줄여줌.

케라스로 배치정규화 구현

- 은닉층의 활성화 함수 이전/이후에 BatchNormalization 층 추가

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

- 활성화 함수 이전에 정규화 층 추가 방법
    - 활성화 함수를 정규화 층 뒤에 별도의 층으로 추가함

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

- BatchNormalization 클래스의 하이퍼파라미터
    - 꽤 많은 수의 하이퍼파라미터 존재
    - 하지만 기본값이 잘 작동함.
    - `momentum`, `axis` 값에 대해 잘 알아둘 필요 있음.

- BatchNormalization 활용
    - 매우 널리 사용됨.
    - 보통 모든 층 뒤에 배치정규화가 있다고 가정
    - 따라서 신경만 그림에 종종 생략됨.

- BatchNormalization를 사용하지 않는 최신 기법
    - Hongyi Zhang 2019년에 제안한 Fixup 가중치 초기화 기법
    - 정규화 없이 10,000개의 층을 가진 심층신경망으로 최고의 성능 달성
    - 하지만 좀 더 검증이 필요함.

그레이디언트 클리핑

- 역전파 과정에서 그레이디언트 값이 일정 임곗값을 넘어설 경우 잘라냄.
- 순환신경망에서 배치정규화를 사용하지 못하는 경우 유용함. (15장 참조)

케라스의 그레이디언트 클리핑

- 옵티마이저를 생성할 때 `clipvalue` 또는 `clipnorm` 지정

- `clipvalue`: 지정된 임계값을 벗어나면 잘라냄.
    - 예제: `[0.9, 100]` => `[0.9, 1.0]`
- 주의: 비율이 달라짐.

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
```

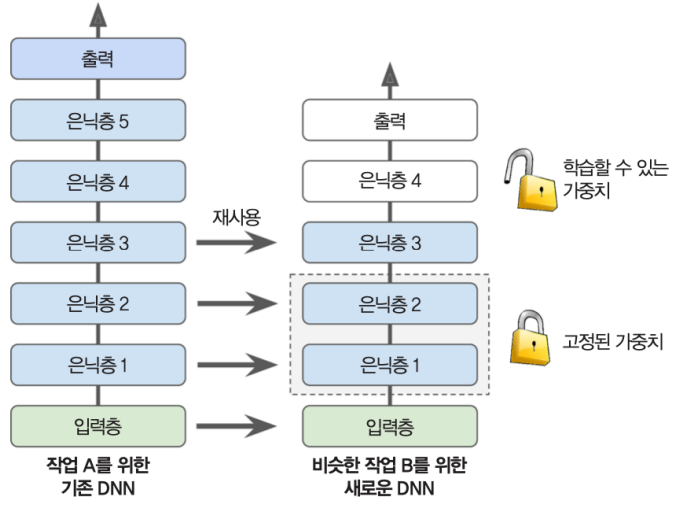
- `clipnorm`: 지정된 임계값 범위내로 전체 값을 선형적으로 축소함
    - 예제: `[0.9, 100]` => `[0.0089964, 0.9999595]`
- 주의: 그레이디언트 소실 발생 가능

```
optimizer = keras.optimizers.SGD(clipnorm=1.0)
```

- 검증세트에서 어떤 방식이 좋은지 확인해야 함.
    - 텐서보드 활용 가능

전이학습

- 비슷한 기능을 가진 사전훈련된 모델의 일부 층을 재사용하는 방법
- 훈련속도를 크게 높일뿐만 아니라 필요한 훈련 데이터의 양도 크게 줄여줌.


출력
은닉층 5
은닉층 4
은닉층 3
은닉층 2
은닉층 1
입력층
재사용
출력
은닉층 4
은닉층 3
은닉층 2
은닉층 1
입력층
학습할 수 있는
가중치
고정된 가중치
작업 A를 위한
기존 DNN
비슷한 작업 B를 위한
새로운 DNN

- 비슷한 원본 모델의 하위 은닉층이 훨씬 유용함.
    - 하위층에서 저수준 특성이 학습되기 때문.

- 먼저 재사용 층을 모두 동결하고 훈련.

- 성능 평가 후 한 두 개 은닉층의 동결을 해제하면서 가중치 조정.
    - 학습률을 줄여야 함.

- 성능이 좋아지지 않거나 훈련 데이터가 적을 경우 상위 은닉층 제거 후 남은 은닉층 동결하고 다시 훈련할 것.

- 훈련 데이터가 많을 경우: 더 많은 은닉층 추가 가능.

- 위 과정 반복.

예제: 케라스 활용 전이학습

- 가정: model_A 주어짐
    - 샌들과 셔츠를 제외한 8개의 클래스만 담겨 있는 패션 MNIST 활용
    - 90% 이상의 정확도 성능을 갖는 8개의 클래스 분류 학습 모델

- 목표:셔츠와 샌들을 분류하는 이진분류기 model_B 훈련

- 방법: 전이학습을 이용한 model_B_on_A 훈련

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

전이학습 성능?

- 위 경우는 매우 좋음. 여러 실험을 거쳐 찾은 좋은 결과에 불과함.
    - 따라서 사람들의 주장을 그대로 믿지 말아야 함.

- 하지만 일반적으로 항상 그렇지는 않음.

- 얕은 신경망 모델에서는 전이학습 성능이 좋지 않음.

- 전이학습은 심층합성신경망 처럼 하위층에서 조금 더 일반적인 특성을 감지하는 경향이 있는 신경망에서 잘 작동함.

비지도 사전훈련

- 레이블된 훈련 데이터가 적을 경우 활용

- 레이블 없는 훈련 데이터를 오토인코더 또는 GAN 등을 이용해 비지도 학습을 통해 레이블 지정 후에 하위층 재사용 (17장 참조)

- 하위층 위에 작업에 맞는 출력층 추가 후 지도학습 실행

보조작업 사전훈련 활용

- 레이블된 훈련 데이터가 적을 경우 활용

예제: 얼굴인식 시스템 개발

- 개인별 이미지가 많지 않을 경우
- 인터넷에서 무작위로 많은 인물 이미지 수집
- 두 개의 다른 이미지를 분류하는 신경망 훈련
- 이후 학습된 모델의 하위층을 재사용
- 적은 양의 레이블된 훈련 데이터를 이용하여 얼굴인식 분류기 학습 가능

고속 옵티마이저

- 지금까지 살펴본 훈련속도 향상 기법
    - 좋은 초기화 전략 사용
    - 좋은 활성화 함수 사용
    - 배치정규화 사용
    - 사전훈련된 심층망 일부 사용

- 모델 컴파일 과정에 필요한 고속 옵티마이저 선택 기법
    - 모멘텀 최적화
    - Nesterov 가속 경사(NAG)
    - AdaGrad
    - RMSProp
    - Adam 최적화
    - Adamax 최적화
    - Nadam 최적화

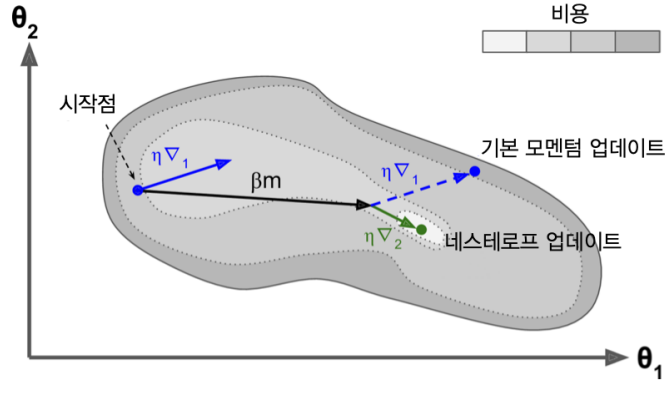
모멘텀 최적화

- 1964년 소개됨
- 가중치 조정 과정을 추적하면서 변화 가속도 조절
- 일반 경사하강법보다 빠르게 전역 최소점에 도달
- 아래와 같이 지정하면 10배 정도 빠르게 학습이 진행됨.

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

Nesterov 가속 경사(NAG)

- 1983년에 소개됨
- 모멘텀 가속화 기법 수정

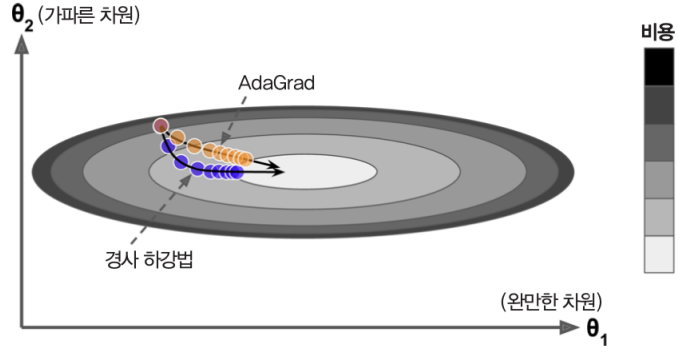


- 기본 모멘텀 최적화보다 일반적으로 훈련 속도 빠름

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

AdaGrad

- 2011년에 소개됨
- 전역 최적점 방향으로 더 곧장 가도록 도와줌.



- 2차방정식 문제에 대해 잘 작동함
- 신경망 훈련할 때 너무 일찍 먼추는 경향 있음. 따라서 심층신경망에는 부적합함.

```
optimizer = keras.optimizers.Adagrad(lr=0.001)
```

RMSProp

- 2012년에 소개됨
- AdaGrad의 이른 종료 문제점 해결한 기법

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

Adam 최적화

- 2014년에 소개됨
- 모멘텀 최적화와 RMSProp의 아이디어 활용

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

AdaMax 최적화

- 2016(?)년에 소개됨
- Adam 알고리즘 개선
- 하지만 경우에 따라 Adam 성능이 더 좋음.

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

Nadam 최적화

- 2016년에 소개됨
- Adam + Nesterov
- 일반적으로 Adam 보다 성능 좋지만 경우에 따라 RMSProp이 더 좋기도 함.

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

옵티마이저 정리

- 선택한 옵티마이저의 성능이 만족스럽지 않을 경우 기본 Nesterov 가속 경사 사용 추천
- 새로운 기법 활용에 관심 가질 것.

클래스	수렴 속도	수렴 품질
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=...)	**	***
Adagrad	***	* (너무 일찍 멈춤)
RMSProp	***	** 또는 ***
Adam	***	** 또는 ***
Nadam	***	** 또는 ***
AdaMax	***	** 또는 ***

부록: 희소 모델 훈련 최적화 알고리즘

- 아주 빠르게 실행할 모델이 필요하거나 메모리를 적게 요구하는 모델이 필요한 경우 희소(sparse) 모델 훈련 가능

해법 1

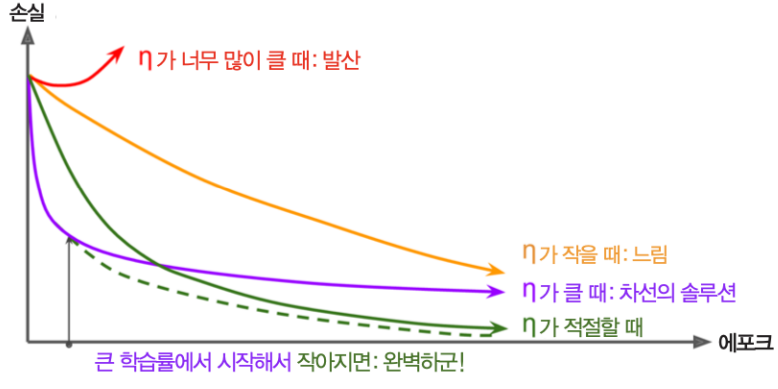
- 옵티마이저에 ℓ_1 규젝 적용
    - (4장) 라쏘 회귀의 경우처럼 옵티마이저가 가능한 한 많은 가중치를 0으로 만듦.

해법 2

- 텐서플로우의 모델최적화 툴킷(TF-MOT) 사용 가능.
    - 훈련과정 동안 반복적으로 연결가중치를 크기에 맞춰 제거하는 가지치기 API 제공.

학습률 스케줄링

- 학습률 선택이 훈련의 성패를 가름.



정의

- 모델 훈련과정동안 학습률을 조정하는 기법
- 보통 높은 학습률로 시작해서 학습속도가 느려질 경우 학습률 작게 조정
- 다양한 기법 소개됨.

알려진 주요 기법

- 거듭제곱 기반 스케줄링(power scheduling)
- 지수 기반 스케줄링(exponential scheduling)
- 구간별 고정 스케줄링(piecewise constant scheduling)
- 성능 기반 스케줄링(performance scheduling)
- 1사이클 스케줄링(1cycle scheduling)
    - 2018년 소개됨

성능 비교

- 2013년 발표된 논문에서 소개
- 모멘텀 최적화를 사용한 음성 인식용 심층신경망 훈련에 다양한 학습률 스케줄링 기법 비교
- 성능 기반과 지수 기반 모두 좋지만 지수 기반 스케줄링 선호
- 튜닝 쉽고, 성능 좀 더 좋고, 구현 쉽기 때문.
- (오렐리아 제롱에 따르면) 하지만 1사이클 방식이 좀 더 성능 좋음.

거듭제곱 기반 스케줄링(power scheduling)

- 학습률을 스텝의 반복횟수 t에 대한 아래 함수로 선언

$$\eta(t) = \frac{\eta_0}{(1 + \frac{t}{s})^c}$$

- $t = k \cdot s$ 로 커지면 학습률이 $\frac{\eta_0}{k+1}$ 로 줄어듦.

- 하이퍼파라미터
    - η_0: 초기 학습률
    - c: 거듭제곱수, 일반적으로 1로 지정
    - s: 스텝 횟수

- 옵티마이저 선언할 때 `decay` 옵션으로 지정

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

- `lr`: 초기 학습률
- `decay`: 스텝수(s)의 역수
- 케라스는 $c = 1$을 기본값으로 사용.

지수 기반 스케줄링(exponential scheduling)

- 학습률을 스텝의 반복횟수 t에 대한 아래 함수로 선언

$$\eta(t) = \eta_0 \, (0.1)^{t/s}$$

- 학습률이 s 스텝마다 10배씩 줄어듦.

- 현재 에포크의 학습률을 받아 반환하는 함수 필요

```
def exponential_decay_fn(epoch):
    return 0.01 * 0.1**(epoch / 20)
```

- 아래 방식처럼 η_0와 s를 설정한 클로저(closure)를 반환하는 함수도 활용 가능.

```
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)
```

- 이제 스케줄링 함수를 이용한 LearningRateScheduler 콜백 함수를 선언한 후 `fit()` 메서드에 전달.
    - 에포크를 시작할 때마다 옵티마이저의 학습률 업데이트.

```
lr_scheduler = keras.callbacks.LearningRateScheduler(exponential_decay_fn)

history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbacks=[lr_scheduler])
```

- 에포크를 스텝마다 업데이트하려면 사용자 정의 콜백을 정의해야 함.

- 예제: `on_batch_begin()`과 `on_epoch_end()` 메서드 재정의

- 주의: `keras.backend` 모듈 활용.
    - batch 단위로 훈련중인 모델의 상태를 확인/변경하는 다양한 기능 지원.

```
K = keras.backend
class ExponentialDecay(keras.callbacks.Callback):
    def __init__(self, s=40000):
        super().__init__()
        self.s = s

    def on_batch_begin(self, batch, logs=None):
        # Note: the `batch` argument is reset at each epoch
        lr = K.get_value(self.model.optimizer.lr)
        K.set_value(self.model.optimizer.lr, lr * 0.1**(1 / s))

    def on_epoch_end(self, epoch, logs=None):
        logs = logs or {}
        logs['lr'] = K.get_value(self.model.optimizer.lr)
```

- 이후 아래와 같이 `fit()` 메서드에 콜백함수로 전달

```
s = 20 * len(X_train) // 32         # 20 에포크 동안의 스텝 수, 배치크기는 32
exp_decay = ExponentialDecay(s)

history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbacks=[exp_decay])
```

- 에포크 당 스텝 수가 많으면 스텝마다 학습률을 조정하는 것이 좋음.
- `keras.optimizers.schedules` 모듈 활용 가능. (잠시 뒤에 설명됨)

구간별 고정 스케줄링(piecewise constant scheduling)

- 지정된 에포크 횟수가 지날 때마다 학습률 조정.
- 적절한 학습률과 에포크 횟수를 잘 찾아야 함.

- 지수 기반 스케줄링 방식과 비슷하게 구현

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

- 또는

```
def piecewise_constant(boundaries, values):
    boundaries = np.array([0] + boundaries)
    values = np.array(values)
    def piecewise_constant_fn(epoch):
        return values[np.argmax(boundaries > epoch) - 1]
    return piecewise_constant_fn

piecewise_constant_fn = piecewise_constant([5, 15], [0.01, 0.005, 0.001])
```

- 콜백함수 지정

```
lr_scheduler = keras.callbacks.LearningRateScheduler(piecewise_constant_fn)

history = model.fit(X_train_scaled, y_train, epochs=n_epochs,
                    validation_data=(X_valid_scaled, y_valid),
                    callbacks=[lr_scheduler])
```

성능 기반 스케줄링(performance scheduling)

- 지정된 스텝 수마다 검증오차 확인 후 오차가 줄어들지 않으면 지정된 `factor` 배 만큼 학습률 감소시킴.

- `ReduceLROnPlateau` 콜백 클래스 활용

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

tf.keras schedulers 모듈의 스케줄 클래스 활용

- 주의: 표준 Keras API 아니며, tf.keras에서만 지원됨.
- keras.optimizers.schedules 모듈의 스케줄 클래스에 학습률 지정 후 옵티마이저에 전달
- 에포크가 아니라 스텝마다 학습률 업데이트

예제

- 아래 옵티마이저는 앞서 정의한 `exponential_decay_fun()` 함수와 동일한 지수기반 스케줄링 기능 제공

```
s = 20 * len(X_train) // 32 # 전체 스텝 수 계산(에포크 20, 배치크기 32)
learning_rate = keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

예제

- 아래 옵티마이저는 구간별 고정 스케줄링 기능 제공

```
learning_rate = keras.optimizers.schedules.PiecewiseConstantDecay(
                    boundaries=[5. * n_steps_per_epoch, 15. * n_steps_per_epoch],
                    values=[0.01, 0.005, 0.001])
optimizer = keras.optimizers.SGD(learning_rate)
```

1사이클 스케줄링(1cycle scheduling)

- 2018년에 소개됨.
- 학습률을 훈련 과정 중에 올리거나 내리도록 조정
- 훈련 전반부: 낮은 학습률 η_0 에서 높은 학습률 η_1 까지 선형적으로 높힘.
- 훈련 후반부: 다시 선형적으로 η_0 까지 낮춤.
- 훈련 마지막 몇 번의 에포크: 학습률을 소수점 몇 째 짜리까지 선형적으로 낮춤.

모멘텀 사용하는 경우

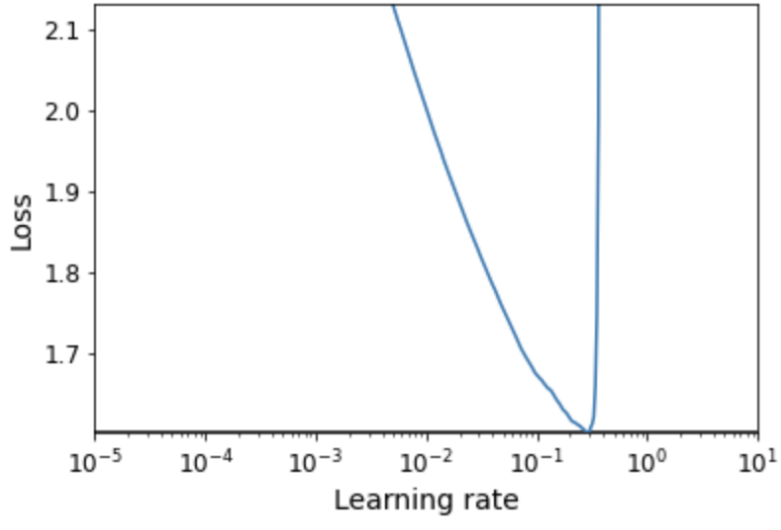
- 훈련 전반부: 0.95와 같은 높은 모멘텀으로 시작해서 0.85와 같은 낮은 모멘텀으로 선형적으로 낮춤.
- 훈련 후반부: 다시 선형적으로 최대 모멘텀으로 높힘.
- 훈련 마지막 몇 번의 에포크: 최대 모멘텀 사용.

장점

- 훈련속도를 크게 높이고 성능도 좋아짐.

구현

- 먼저 최대학습률 η_1을 지정하기 위해 학습률과 손실함수 사이의 관계를 알아내야 함.
  - 예를 들어, 아래 그래프의 경우 $\eta_1 = 0.05$가 가장 적절해 보임.



- 이제 적절한 콜백함수를 구현해서 실행하면 됨.
    - 주피터 노트북 참조.

규제 사용과 과대적합 피하기

- 심층신경망에 사요되는 수천, 수만, 수백만 개의 파라미터에 의해 과대적합 발생 쉬움.
- 과대적합을 방지하기 위한 다양한 규제 요구됨.

심층신경망 관련 지금까지 살펴본 규제기법

- 조기종료 기법 (10장)
    - `EarlyStopping` 콜백을 사용하여 일정 에포크 동안 성능이 향상되지 않는 경우 자동 종료시키기

- 배치정규화
    - 불안정한 그레이디언트 문제해결을 위해 사용하지만 규제용으로도 활용 가능.
    - 가중치의 변화를 조절하는 역할을 수행하기 때문.

- 새로 살펴볼 규제기법
    - ℓ_1과 ℓ_2 규제
    - 드롭아웃(dropout)
    - 맥스-노름(max-norm) 규제

ℓ_1과 ℓ_2 규제

- 층을 선언할 때 규제 방식과 규제강도를 지정할 수 있음.
    - `kernel_regularizer` 옵션 사용.

- 예제: ℓ_2 규제, 규제강도는 0.01.

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

functools.partial() 함수 활용

- 일부 매개변수의 기본값을 사용하여 함수호출을 감싸는 방식 지원

- 모든 층에 동일한 활성화 함수, 동일한 초기화 전략, 동일한 규제 등을 반복하여 사용하고자 하는 경우 유용함.

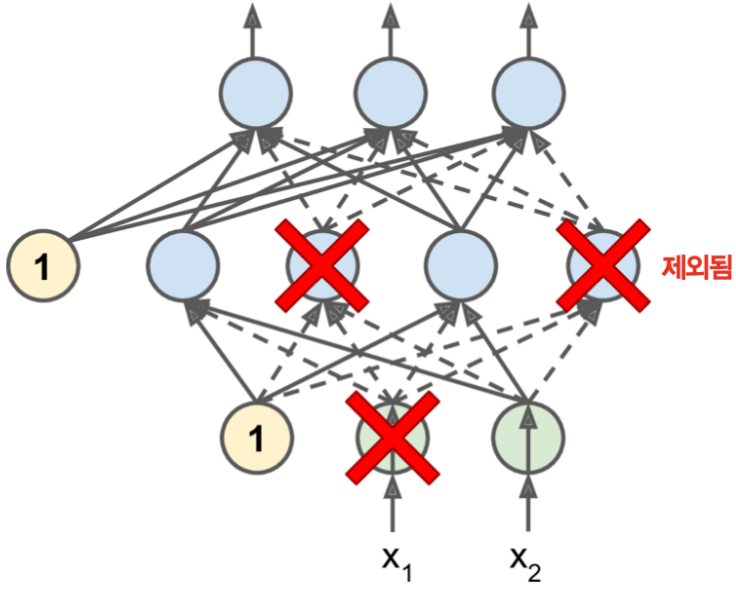
```
from functools import partial

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")
```

드롭아웃(dropout)

- 신경망에서 가장 인기 있는 규제기법
- 2012년에 소개됨.
- 일반적으로 매우 잘 작동함.

기본 아이디어

- 매 훈련 스텝에서 출력 뉴런을 제외한 모든 뉴런이 p의 확률로 훈련에서 제외될 수 있게 만듦.
    - 제외되는 뉴런의 출력값은 0.



- 에포크 또는 스텝마다 서로 다른 고유한 모델이 훈련되는 효과를 발생시킴.
- 훈련이 끝나면 드롭아웃 적용하지 않음.

드롭아웃 비율(p)

- 순환 신경망(15장): 20% - 30%
- 합성곱 신경망(14장): 40% - 50%
- 순환신경망(15장): 20%-30%

- 드롭아웃 비율을 올려야 하는 경우
    - 과대적합 발생하는 경우
    - 층에 많은 뉴런이 포함될 때

- 드롭아웃 비율을 올려야 하는 경우
    - 과소적합 발생하는 경우
    - 층에 적은 뉴런이 포함될 때

- 일반적으로 출력층을 제외한 최상위 3개 층에 대해 드롭아웃 적용.
    - 최신 신경망 모델: 마지막 은닉층에만 적용하기도 함.

보존확률(keep probability)

- 훈련 완료 후 모든 가중치에 보존확률(keep probability) $(1 - p)$를 곱해주는 게 좋음.
    - 이유: 훈련과정중에 전체 뉴런의 (p 비율에 해당하는) 일부만 출력값을 생성하는 것에 모델이 익숙해져 있기 때문.

- 또는 훈련과정 중에 각 뉴런의 출력을 보존확률로 나눌 수도 있음.
    - 케라스 모델은 이 방식을 기본으로 지원함.

예제

- 모든 Dense 층 이전에 드롭아웃 비율 0.2 지원하기

```
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")
    ])
```

드롭아웃과 과대적합

- 훈련손실과 검증손실을 그대로 비교하면 안됨.
    - 이유: 훈련시에만 드롭아웃을 적용하기 때문.

- 훈련 후 드롭아웃을 끄고 훈련손실을 재평가해야 함.

알파 드롭아웃

- SELU 활성화함수를 사용하는 경우 알파(alpha) 드롭아웃 사용 추천
- 입력과 평균의 표준편차를 유지시켜줌.
- 일반 드롭아웃은 자기 정규화 기능 방해.

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.AlphaDropout(rate=0.2),
    keras.layers.Dense(300, activation="selu", kernel_initializer="lecun_normal"
),
    keras.layers.AlphaDropout(rate=0.2),
    keras.layers.Dense(100, activation="selu", kernel_initializer="lecun_normal"
),
    keras.layers.AlphaDropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
    ])
```

몬테카를로 드롭아웃(MC Dropout)

- 2016년에 소개됨.
- 훈련된 드롭아웃 모델을 재훈련하지 않으면서 성능을 향상시키는 기법
- 아래와 같이 훈련된 모델의 예측기능을 이용한 결과를 스택으로 쌓은 후 평균값을 계산하면 됨.
    - 단, `training=True` 로 지정하여 드롭아웃 층을 활성화해야 함.

- 예제: 100개의 행렬을 쌓아 올린 후 평균 내기

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

- 테스트 세트 샘플 수는 10,000개, 클래스 수는 10개. 따라서 `[10000, 10]` 모양의 행렬 100개.
- 즉, `[100, 10000, 10]` 모양의 행렬이 `y_probas`에 저장됨.

- 몬테카를로 드롭아웃을 실행하면 보다 정확한 예측과 보다 정확한 불확실성 추정이 가능해짐.
    - 위험에 민감하지만 예측속도가 그다지 중요하지 않은 경우 사용 추천

- 주의사항
  - BatchNormalization 과 같은 층이 사용되면 앞서 설명한 방식을 적용하면 안됨.
  - 대신에 Dropout 또는 AlphaDropout 층을 아래에 정의된 MCDropout 또는 MCAlphaDropout 층으로 대체해야 함.

```
class MCDropout(keras.layers.Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)

class MCAlphaDropout(keras.layers.AlphaDropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

```
mc_model = keras.models.Sequential([
    MCAlphaDropout(layer.rate) if isinstance(layer, keras.layers.AlphaDropout) el
se layer
    for layer in model.layers
    ])
```

맥스-노름(max-norm) 규제

- 각각의 뉴런에 대해 입력 가중치 $\mathbf{w}$의 ℓ_2 노름을 하이퍼파라미터 r 보다 작게 제한하는 기법

- 은닉층 별로 지정할 수 있음. `kernel_constraint` 매개변수 값을 아래와 같이 지정하면 됨.

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

- 14장에서 다룰 합성곱 층에서 맥스-노름을 적용하려면 `max_norm()` 함수의 `axis` 값을 적절하게 지정해야 함.

실용적 가이드라인

- 소개된 많은 기법들을 어제 어떻게 써야할지 고민될 때 참고할 수 있는 가이드라인 소개

심층신경망 기본 설정

하이퍼파라미터	기본값
커널초기화	He 초기화
활성화 함수	ELU
정규화	깊은 신경망일 경우에만 배치정규화 사용
규제	조기종료, 경우에 따라 ℓ_2 규제 추가
옵티마이저	모멘텀 최적화, RMSProp, Nadam 중 하나
학습률 스케줄	1사이클

자기정규화를 위한 심층신경망 설정

- 완전연결층으로 구성된 단순한 모델인 경우 아래 설정 추천

하이퍼파라미터	기본값
커널초기화	르쿤(LeCun) 초기화
활성화 함수	SELU
정규화	필요 없음
규제	경우에 따라 알파 드롭아웃
옵티마이저	모멘텀 최적화, RMSProp, Nadam 중 하나
학습률 스케줄	1사이클

- 주의사항
    - 입력특성을 정규화해야 함.
    - 비슷한 모델의 사전훈련된 신경망의 일부 재사용 가능

희소모델인 경우

- ℓ_1 규제 사용 추천

- 매우 희소한 모델이 필요한 경우: 텐서플로우의 모델최적화 툴킷(TF-MOT) 사용 가능
    - 기본 DNN 사용해야 함. 자기 정규화가 작동하지 않기 때문.

빠른 응답 모델인 경우

- 원하는 응답속도에 따라 은닉층 수 줄여햐 함.
- 배치정규화 층을 이전 층과 합칠 것.
- LeakyReLU 또는 ReLU 사용.
- 희소모델 사용
- 부동소수점 정밀도 조정 (19장 2절의 모바일/임베디드 장치 모델 참조)

예측속도보다 정확도에 충실한 모델인 경우

- 성능을 올리고 불확실성 추정을 정확하게 하여 신뢰도가 높은 모델을 원하는 경우
- 몬테카를로 드롭아웃 사용 추천