6장 학습 관련 기술들

매개변수 갱신

신경망 학습의 목적 : 손실 함수 값을 낮추는 매개변수를 찾는 것

⇒ 매개변수의 최적값을 찾기 위해서 '최적화'를 진행

⇒ 매개변수의 기울기를 이용해서 찾아왔다. → 기울어진 방향으로 매개변수 갱신을 반복해서 최적값에 다가갔음

⇒ 확률적 경사하강법 SGD

SGD

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr=lr

    def update(self, params, grads):
        for key in params.keys():
            params[key]-=self.lr*grads[key]
```

⇒ 사용할 때는 `optimizer=SGD()` 로 정의하고, `optimizer.update(params, grads)` 로 매개변수, 기울기 정보만 넘겨주면 됨

단점 : 만일 기울기가 y축 방향은 가파른데 x축 방향은 완만하다면?

최적화 갱신 경로

⇒ 심하게 굽이진 움직임, 비효율적임

⇒ 비등방성(anisotropy) 함수 = 방향에 따라 기울기가 달라짐

지그재그로 탐색하는 근본 원인 : 기울어진 방향이 본래의 최솟값과 다른 방향을 가리키기 때문

모멘텀

v : 속도 변수가 추가됨

기울기 방향으로 힘을 받아 가속되는 성질

```
import numpy as np

class Momentum:
    def __init__(self,lr=0.01,momentum=0.9):
        self.lr=lr
        self.momentum=momentum
        self.v=None

    def update(self,params,grads):
        if self.v is None:
            self.v={}
            for key,val in params.items():
                self.v[key]=np.zeros_like(val)

        for key in params.keys():
            self.v[key]=self.momentum*self.v[key]-self.lr*gra
            params[key]+=self.v[key]
```

최적화 갱신 경로

→ 공이 바닥을 구르듯 움직임

⇒ sgd와 비교하면 '지그재그 정도'가 덜함

⇒ x축의 힘은 아주 작지만 방향이 변하지 않아서 한 방향으로 일정하게 가속하기 때문

⇒ 전체적으로 sgd보다 x축 방향으로 빠르게 다가가 지그재그 움직임이 줄어듦

⇒ y축 방향의 속도는 안정적이지 않음

AdaGrad

신경망 학습에서는 학습률 값이 중요함

⇒ 학습률을 정하는 기술 : 학습률 감소 learning rate decay

⇒ 학습을 진행하면서 학습률을 점차 줄여나감

AdaGrad는 '각각의' 매개변수에 '맞춤형' 값을 만들어줌

→ 개별 매개변수에 적응적으로 adaptive 학습률을 조정

h라는 새로운 변수 등장

→ 기존 기울기 값을 제곱해서 계속 더한게 h가 되고, 매개변수를 갱신할 때 루트h를 역수로 곱해 학습률을 조정

⇒ 매개변수 원소 중 많이 움직인 = 크게 갱신된 원소는 학습률이 낮아짐

⇒ 학습률 감소가 매개변수의 원소마다 다르게 적용됨

⇒ 만일 무한히 학습한다면? 갱신량이 0이 되어 전혀 갱신되지 않음 ⇒ RMSProp 방법은 먼 과거의 기울기는 서서히 잊고 새로운 기울기 정보를 크게 반영함으로서 갱신할 수 있게해줌

Adagrad

```
import numpy as np

class AdaGrad:
    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}

            for key, val in params.items():
```

```
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(se
```

⇒ 1e-7은 `self.h[key]`에 0이 담겨있을 때 0으로 나누는 사태를 막기위한 작은 수

RMSProp

```
class RMSprop:

    """RMSprop"""

    def __init__(self, lr=0.01, decay_rate = 0.99):
        self.lr = lr
        self.decay_rate = decay_rate
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] *= self.decay_rate
            self.h[key] += (1 - self.decay_rate) * grads[ke
y] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt
(self.h[key]) + 1e-7)
```

⇒ decay_rate 로 이전의 기울기에 곱해줌

Adagrad 최적화 갱신 경로

⇒ 최솟값을 향해 효율적으로 움직임

⇒ y축 방향은 기울기가 커서 처음에는 크게 움직이지만, 큰 움직임에 비례해 갱신 정도도 큰 폭으로 작아짐!

⇒ y축 방향으로 갱신 강도가 빠르게 약해지고 지그재그 움직임이 줄어듦

Adam

모멘텀+Adagrad

⇒ 매개변수 공간을 효율적으로 탐색, 하이퍼파라미터의 '편향 보정'

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \frac{1}{\sqrt{\hat{\mathbf{v}}_n}} \odot \hat{\mathbf{m}}_n$$

$$\mathbf{m}_n = \beta_1 \mathbf{m}_{n-1} + (1-\beta_1)\nabla f(\mathbf{x}_n), \qquad \mathbf{m}_{-1} = 0$$

$$\mathbf{v}_n = \beta_2 \mathbf{v}_{n-1} + (1-\beta_2)\nabla f(\mathbf{x}_n) \odot \nabla f(\mathbf{x}_n), \qquad \mathbf{v}_{-1} = 0$$

"기존 Gradient Descent의 Learning rate를 active하게 바꿔주기위해 momentum(관성)을 추가하였고, 이로 인해 learning rate가 0으로 수렴하는 문제를 해결하기위해 RMSProp라는 스킬을 써주었다."

라는 컨셉만 이해하면 된다고 함!

```
class Adam:

    """Adam (http://arxiv.org/abs/1412.6980v8)"""

    def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.iter = 0
```

```
        self.m = None
        self.v = None

    def update(self, params, grads):
        if self.m is None:
            self.m, self.v = {}, {}
            for key, val in params.items():
                self.m[key] = np.zeros_like(val)
                self.v[key] = np.zeros_like(val)

        self.iter += 1
        lr_t  = self.lr * np.sqrt(1.0 - self.beta2**self.iter

        for key in params.keys():
            #self.m[key] = self.beta1*self.m[key] + (1-self.b
            #self.v[key] = self.beta2*self.v[key] + (1-self.b
            self.m[key] += (1 - self.beta1) * (grads[key] - s
            self.v[key] += (1 - self.beta2) * (grads[key]**2

            params[key] -= lr_t * self.m[key] / (np.sqrt(self

            #unbias_m += (1 - self.beta1) * (grads[key] - sel
            #unbisa_b += (1 - self.beta2) * (grads[key]*grads
            #params[key] += self.lr * unbias_m / (np.sqrt(unb
```

가중치 초깃값

가중치 감소 → 오버피팅 억제해줌

가중치를 작게하려면 초깃값도 최대한 작은 값??

→ 그렇지만 초깃값을 0으로 하면 학습이 올바로 이루어지지않음

→ 초깃값을 0으로 두면 안되는 이유는?? = 가중치를 균일한 값으로 설정하면 안되는 이유는?

⇒ 오차역전파법에서 모든 가중치의 값이 똑같이 갱신되기 때문

⇒ 가중치를 여러 개 갖는 의미를 사라지게 함

⇒ 가중치게 고르게 되어버리는 상황을 막으려면 초깃값을 무작위로 설정해야함!

가중치를 표준편차가 1인 정규분포로 초기화한다면?

각 층의 활성화값들이 0,1에 치우쳐서 분포

→ 출력이 0또는 1에 가까워지면 미분값이 0에 다가감

⇒ 데이터가 0과 1에 치우쳐 분포하면 역전파의 기울기 값이 점점 작아지다가 사라짐

⇒ 기울기 소실

가중치를 표준편차가 0.01인 정규분포로 초기화한다면?

0.5부근에 집중 ⇒ 활성화값들이 치우쳐짐

⇒ 다수의 뉴런이 거의 같은 값을 출력하니 뉴런을 여러 개 둔 의미가 없어짐

⇒ 표현력을 제한함

⇒ 각 층의 활성화값이 적당히 고루 분포되어야함

Xavier 초깃값 (사비에르 초깃값)

층이 깊어지면서 형태가 일그러지지만, 앞 방식들 보단 넓게 분포됨

⇒ 학습이 효율적

He 초깃값 (히 초깃값)

사비에르 초깃값은 활성화 함수가 선형인 것을 전제로 이끈 결과임

→ ReLU를 이용할 때는?

ReLU에 특화된 초깃값 = He 초깃값

ReLU 활성화함수를 사용했을 때 활성화 값 분포 변화

std=0.01 정규분포일 때는 각 층의 활성화값이 아주 작음 = 신경망에 아주 작은 데이터가 흐름

⇒ 역전파 때 가중치의 기울기가 작아짐 ⇒ 학습이 거의 이뤄지지 X

사비에르 초깃값은 층이 깊어지면서 치우침이 커짐

⇒ 층이 깊어지면 치우짐 커지고 학습시 기울기 소실 문제

He 초깃값은 모든 층에서 균일하게 분포

⇒ 역전파 때도 적절한 값

결론

- 활성화함수 ReLU → He 초깃값
- 활성화함수 sigmoid, tanh → Xavier 초깃값

배치 정규화

가중치의 초깃값을 적절히 설정하면 각 층의 활성화값 분포가 적당히 퍼짐 → 학습 원활

각 층이 활성화를 적당히 퍼뜨리도록 '강제' 한다면? → 배치 정규화 아이디어

배치 정규화 장점

- 학습 속도 개선
- 초깃값에 크게 의존하지 않음
- 오버피팅 억제 (드롭아웃 등 필요성 감소)

배치 정규화 : 각 층에서의 활성화값이 적당히 분포되도록 조정함

→ 데이터 분포를 정규화하는 '배치 정규화 계층'을 신경망에 삽입

- 학습 시 미니배치를 단위로 정규화

  ⇒ 데이터 분포가 평균 0, 분산 1이 되도록 정규화

⇒ 이를 활성화 함수의 앞이나 뒤에 삽입하여 데이터 분포가 덜 치우치게 할 수 있음

- 정규화된 데이터에 교유한 확대scale, 이동shift 변환을 수행 → 확대, 이동 값은 학습하면서 적합한 값으로 조정됨
- 순전파 때 적용

오버피팅

일어나는 경우

- 매개변수가 많고 표현력이 높은 모델
- 훈련 데이터가 적을 때

→ 억제를 위한 방법 필요

가중치 감소 weight decay

오버피팅은 매개변수 값이 커서 발생하는 경우 많음

→ 학습 과정에서 큰 가중치에 대해서는 그에 상응하는 큰 페널티를 부과해 오버피팅을 억제

신경망 학습의 목적 = 손실 함수 값을 줄이는 것

→ 가중치 L2 노름을 손실 함수에 더하면 가중치 커지는 걸 억제할 수 있음

람다라는 정규화의 세기를 조절하는 하이퍼파라미터를 설정

→ 람다가 클수록 큰 가중치에 대한 패널티가 커짐

⇒ 모든 가중치 각각의 손실 함수에 L2 노름 (1/2 * 람다 * W^2) 을 더함

⇒ 가중치 기울기 구할 때 그동안의 오차역전파법에 따른 결과에 정규화 항을 미분한 람다 * W를 더함

드롭아웃

가중치 감소 = 손실 함수에 가중치의 L2 노름을 더한 방법 → 간단하고 어느정도 지나친 학습 억제 가능

⇒ 그러나 신경망 모델이 복잡해지면 가중치 감소만으로는 대응하기 어려움

⇒ 드롭아웃 기법 이용

드롭아웃 = 뉴런을 임의로 삭제하면서 학습하는 방법

- 훈련 때 은닉층의 뉴런을 무작위로 골라 삭제
- 시험 때는 모든 뉴런에 신호를 전달
- 시험 때는 각 뉴런의 출력에 훈련 때 삭제 안 한 비율을 곱해서 출력

```
class Dropout:
    def __init__(self, dropout_ratio=0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg=True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropo
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask
```

순전파 때마다 `self.mask`에 삭제할 뉴런을 False로 표시

self.mask는 x와 형상이 같은 배열을 무작위로 생성하고, 그 값이 dropout_ratio보다 큰 원소만 True로 설정함

역전파는 ReLU와 같음

⇒ 순전파 때 신호를 통과시키는 뉴런은 역전파 때도 신호를 그대로 통과, 순전파 때 통과시키지 않은 뉴런은 역전파 때도 신호를 차단

신경망의 맥락에서는 앙상블 학습과 비슷

→ 드롭아웃이 학습 때 뉴런을 무작위로 삭제하는 행위를 매번 다른 모델을 학습시키는 것으로 해석할 수 있음

⇒ 드롭아웃은 앙상블 학습같은 효과를 대략 하나의 네트워크로 구현했다고 생각할 수 있음

적절한 하이퍼파라미터 값 찾기

하이퍼파라미터 값을 최대한 효율적으로 탐색하는 방법

검증 데이터

하이퍼파라미터의 성능을 평가할 때 시험 데이터를 사용해선 안됨

→ 시험 데이터를 사용해 하이퍼파라미터를 조정하면 하이퍼파라미터 값이 시험 데이터에 오버피팅되기 때문

⇒ 범용 성능 안좋아짐

⇒ 하이퍼파라미터 전용 확인 데이터가 필요

⇒ 검증 데이터 validation data필요

훈련 데이터 : 매개변수(가중치와 편향)의 학습

검증 데이터 : 하이퍼파라미터 성능 평가

시험 데이터 : 신경망의 범용 성능 평가

검증 데이터 얻는 방법

- 훈련 데이터 중 20%정도를 먼저 분리

하이퍼파라미터 최적화

핵심 : 하이퍼파라미터의 '최적 값'이 존재하는 범위를 조금씩 줄여감

⇒ 대략적인 범위를 설정하고 그 범위에서 무작위로 하이퍼파라미터 값을 샘플링한 후, 그 값으로 정확도 평가

⇒ 정확도를 살피면서 이 작업을 반복해 최적 값의 범위를 좁혀가는 것

그리드 서치같은 규칙적인 탐색보다는 무작위로 샘플링해 탐색하는 편이 좋은 결과를 냄!

→ 최종 정확도에 미치는 영향력이 하이퍼파라미터마다 다르기 때문

범위는 '대략적으로' 지정하는 게 효과적

→ 보통 로그 스케일 (10의 거듭제곱) 단위로 지정

→ 시간 오래 걸림 (딥러닝에서는 며칠~몇 주 이상)

⇒ 나쁠 듯한 값을 일찍 초기

⇒ 학습을 위한 에폭을 작게 해, 1회 평가에 걸리는 시간을 단축하는 게 효과적

0단계

- 하이퍼파라미터 값의 범위를 설정

1단계

- 설정된 범위에서 하이퍼파라미터의 값을 무작위로 추출

2단계

- 1단계에서 샘플링한 하이퍼파라미터 값을 사용해 학습하고, 검증 데이터로 정확도를 평가 (단, 에폭은 작게 설정)

3단계

- 1단계와 2단계를 특정 횟수 (100회 등) 반복하며, 그 정확도의 결과를 보고 하이퍼파라미터의 범위를 좁힘

더 세련된 기법?

베이즈 최적화 Bayesian optimization

- 베이즈 정리를 중심으로 한 수학 이론을 구사 → 더 엄밀, 효율적으로 최적화 수행
- 핵심은 사전 정보를 최적값 탐색에 반영하는 것

https://data-scientist-brian-kim.tistory.com/88