우리는 주어진 데이터에 대해서 좋은 결과를 갖는 출력값을 가지는 가상의 함수를 모사한다. 이때, Loss 값을 최소로 만들어야한다. Loss의 값을 최솟값으로 만들기 위해 Gradient descent를 이용한다. Gradient descent란 Loss를 파라미터로 미분하여 기울기를 업데이트해 점진적으로 최적의 파라미터를 찾는것이다. 이때 미분을 효율적으로 하기위해 back-propagation을 사용한다. back-propagation은 chain rule을 통해 이전에 사용한 미분값을 재활용한다.

Neural networks 학습과정

1. 데이터 불러오기
2. 데이터 전처리 (결측값등 처리)
3. 데이터 텐서화
4. 데이터 shuffle 및 split
5. 데이터 scaling
6. 데이터 모델링
7. 모델 만들기
8. 하이퍼 파라미터 지정(epoch, batch\_size, optimizer)
9. 훈련과정: epoch과 iteration 두번의 fot문이 동작한다.

train에서, 첫 for문은 epoch에 대한 것으로, 데이터 shuffle 및 배치사이즈로 split, 두번째 for문은 iteration에 대한 것으로 model과 loss를 넣어주고, optimizer.zero\_grad, loss backward(), optimizer.step()을 진행한다. 그다음 train\_loss를 저장한다.

valid에서, 위와 똑같이 진행되지만, optimizer.zero\_grad, loss backward(), optimizer.step()을 진행하지 않는다.

오버피팅을 피하기위 해서 train, valid, test set으로 나누어 주고 training을 하는 중간중간마다 valid set을 통해서 현재 오버피팅이 되어있는지 generalization loss를 체크하고 결과적으로 validation set에 대해서 하이퍼 파라미터들이 오버피팅 될 수 있기 때문에, test set을 두어 최종적으로 모델을 평가한다.

Train set: 훈련을 위한 data set이며, parameters를 훈련중 조정한다. 파라미터 학습을 위한 데이터.

Valid set: 훈련중, 성능을 평가하고, Hyper parameters를 조정하여 과적합을 확인한다. -> 훈련을 잘하는지 검증한다. 매 epoch이 끝나면 validation error를 평가하여 generalization을 평가

Test set: 모델의 최종 성능 평가 및 일반화 능력 테스트

|  |  |  |  |
| --- | --- | --- | --- |
|  | Train set | Valid set | Test set |
| Parameter | 결정 | 검증 | 검증 |
| Hyper-parameter |  | 결과 | 검증 |
| Algorithm |  |  | 결과 |

**우리의 목표: 주어진 데이터를 통해서(데이터 입력) 출력값을 가지는 가상의 함수를 모사하는것이다.**

신경망의 가장 기본적인 구성요소는 Linear Layer(=Fully Connected Layer)이다. 내부 파라미터(θ, 세타)에 따른 선형변환을 수행하는 함수다. 여기서 내부 파라미터는 Weight(W), bias(b)이다.

**\* Loss(손실값)**: 실제 출력값과(target, y)과 예측 출력값(output, y^)의 차이의 합. 모델의 성능을 나타내는 지표.

Loss function의 수식은 다음과 같다.

Loss가 작을 수록 가상의 함수를 잘 모사하고 있다고 할 수 있다. -> 우리의 목표는 loss를 최소화 하는것이다

- Euclidean Distance(유클리드 디스턴스) : 두점사이의 거리(L2).

=

- RMSE(Root Mean Square Error) : 유클리드 디스턴스와 비슷한 개념이지만, Normalize한 개념이다. 평균에 거리를 적용한 것이다.

- MSE(Mean Square Error) : Root와 상수를 뺏지만 크기의 차이로 순서가 바뀌지 않는다.

(유클리드 디스턴스의 제곱 / n)

(루트와 1/n이 없더라도 크기의 순서가 변하지 않는다.)

**∴ 실제값들과(y) 출력값들(y^)의 차이의 합 (Loss)를 최소화 해야한다. 이것은 파라미터를 바꾸면서 Loss를 최소화 해야한다.**

**어떻게? Gradient Descent를 통해서!**

**\* Gradient Descent(경사 하강법)** : Loss Function의 최솟값을 찾기 위해 Loss를 파라미터(θ)로 미분하여, 함수의 기울기(gradient)를 업데이트해서 점진적으로 최적의 파라미터를 찾는다. 이때 하이퍼 파라미터로는 Learning Rate(η(에타), 러닝 레이트)가 있다.

Learning Rate (η) : 학습의 Step-size다.

Large LR : 많이 이동해서 학습시간이 적게 걸린다. 수렴이 빠르다. Loss가 발산할 수 있다.

Small LR : 조금 이동해서 학습시간이 오래 걸린다. 수렴이 너무 늦다.

(Loss를 로 미분)

즉! 미분하여 기울기를 활용하여 좀 더 낮은곳으로 점차 나아가자!

- Learning Rate (η, 에타) : 하이퍼 파라미터. 기울기에 따른 경사도 하강에서 파라미터 업데이트의 크기를 정하는 역할

학습률이 너무 작으면 수렴이 너무 늦고, local minima에 빠질수 있으며, 학습률이 너무 크면, loss가 발산할 수 있다.

Regression은 MSE Loss, 출력

Classification은 BCE Loss, 출력 activation function은 이진분류 sigmoid, 다중분류 softmax

**\* Back-propagation(역전파)** : Gradient Descent를 빠르게 잘 수행하기 위해서 사용한다. (각 layer의 파라미터로 그냥 미분하는 것은 매우 비효율 적이다.)

Back-propagation은 chain rule을 통한 이전에 사용한 미분값 재활용을 통해 미분과정을 효율적으로 한다.

2 (), 3 ()

**\* Gradient Vanshing** : 학습이 진행됨에 따라서 gradient가 소멸되는 문제

0<Sigmoid<1, -1<tanh<1이기 때문에, 미분값은 항상 0보다 같거나 작다.

-> ReLU를 사용해서 막는다. 양수부분의 기울기가 1이므로 학습이 빠르고, linear특성 때문에 최적화가 쉽지만, 입력값이 음수면 기울기가 0이 되므로 학습이 불가능하다.

라인, 텍스트, 도표, 그래프이(가) 표시된 사진

자동 생성된 설명

**\* Stochastic Gradient Descent (SGD)** :

Gradient Descent는 전체 데이터셋을 다 feed forward를 한다음에 더하고 backpropagation을 한다. Stochastic Gradient Descent는 미니배치(k)로 일부데이터들을 샘플링한다음, feedforward하고 backpropagation한다. 배치사이즈만큼의 데이터로 loss를 구한다.

즉, Stochastic Gradient Descent는 전체 샘플의 loss에 대한 gradient descent가 아닌, 일부 샘플(k=batch size)의 loss에 대한 gradient descent를 구한다.

모든 데이터들을 한 번씩 모델에 학습시킨다는 관점에서 1epoch

1epoch에서 파라미터의 업데이트는 엄청나게 많게된다. 전체 데이터에 대해서 gradient descent하면 1epoch당 한개의 업데이트밖에 없다.

전체 데이터에 대한 gradient descent는 batch\_size가 큰경우에 대한 극단적인 예다

미니배치 사이즈가 1인경우를 stochastic gradient descent의 극단적인 예다.

배치사이즈의 크기에 따라 gradient의 noisy함의 정도가 다를것이다.

미니배치 gradient의 경우에는 전체데이터의 gradient와 다를수밖에없다.

random하게 sampling을 하니까 일부 데이터를 가지고 만든 loss는 다르다.

전체 데이터를 가지고 Gradient Descent를 하면 올바른 지점까지 쭉 잘 진행된다

매 시점마다 미니배치 가지고 샘플링을하고 피드포워드를하고 업데이트를 하면 gradient가 달라지게된다.

원래의 방향과 달라지게 된다. 따라서 noisy하게 된다. random하게 sampling하니까!

gradient의 평균은 Gradient Descent와 같아질것이라고 기대한다. 그러나 variace가 커질것이다.

즉, 배치사이즈가 작을수록 variance가 커질것이다. 배치사이즈가 클수록 noisy가 적어지기 때문에 실제 gradient 유사해질것이다.

* Batch size

Batch size가 작으면 local minima를 탈출할 가능성도 있다.

Batch size가 크면 상대적으로 더 빠르게 수렴할 수 있다.

하지만 배치사이즈가 크면 파라미터 업데이트 횟수가 줄어든다. 방향은 정확히지만 한 epoch당 업데이트하는 횟수가 줄어든다.

데이터가 정제가 잘 되어있다면 배치사이즈를 작게, 데이터가 노이즈하다면 배치사이즈를 크게 가져가는 방안이 될 수있다.

사실 정답은 없다. 보통 32로 많이 하더라

경험적으로 봤을때 몇백만 차원일 경우에 local minima구덩이가 많이 생기지 않을것이다. local minima에 빠지더라도 global minima의 근처일것이다.

따라서 배치사이즈를 줄이면 local minima를 빠질 가능성이 있지만, 굳이 그럴 필요가 없다는 실무적 관점이 있다.

데이터셋의 크기가 매우 크다면, 데이터셋의 실제 모집단의 분포와 비슷해질것이다.

gradient의 방향은 실제 전체 데이터셋을 활용한 gradient방향과 비슷할 것이다.

하지만 한번의 파라미터를 업데이트하는데 더 많은 계산이 필요하다.

만약 작은 방향의 배치 사이즈 gradient방향이 전체 데이터 gradient 방향과 거이 같다면?

배치사이즈 gradient 평균 방향이 같지 않을까?

1. 작은 배치사이즈

랜덤 샘플들은 모집단에 비해 훨씬 편향되어있다. 따라서 ground-truth gradient와 달라진다

기대값(평균)은 같다

local minima에 빠질지라도, 편향된 gradient로 인해 탈출 가능할 수 있다.

하지만 noisy가 심해져 수렴이 어렵다.

2. 큰 배치사이즈

배치사이즈가 클수록 파라미터 업데이트 횟수는 줄어든다

즉, 모델이 학습할 기회가 줄어든다

대신 한번 학습할 때, gradient의 방향은 상대적으로 정확할 수 있다

-> epoch을 많이하면 되는거 아냐? 또는, LR를 작게하면 되지 않을까?

Epohs = batch size/n \* Iteration

iterations = n/batch size \* epochs

\* 1Epoch: 모든 데이터셋의 샘플들이 Feedforward & Backward 되는 시점.

Epoch의 시작에 데이터셋을 random shuffling해준후 미니배치(k)로 나눈다.

\* 1Iteration: 한 개의 미니배치 사이즈들이 시작할 때 Feedforward & Backward 되는 시점.

따라서 Epoch과 Iteration의 이중 for문이 만들어진다.

n/batch\_size 에폭당 업데이트 수

배치사이즈에 따라서 모델의 학습에 영향을 끼친다.

굳이 추천하자면 256 -> 128 ->64 -> 32.... (메모리가 줄어드니까 줄여나간다.)

보통 2^n을 많이 사용한다 (배치사이즈나, 레이어의 크기 같은경우 2^n을 사용한다. empirical하게 실제로 속도 차이가 있다고 한다. 또는 512+256등 2^n끼리의 합도 괜찮다고 한다.)

* Hyper parameters

\* Model의 Parameters? 모델 내부의 설정값으로 데이터에 의해서 값이 정해진다 – 학습에 의해서 자동으로 변경된다. 사용자에 의해서 조정되지 않으며,

최적의 파라미터를 찾기위한 loss에서의 gradient descent, NLL에서 gradient descent통해 파라미터를 최소화 하는것을 생각하면 된다.

\* Hyper parameters? 모델 외부의 설정값으로, 사용자에 의해서 결정된다.

데이터의 상황에 따라 최적의 값이 다르므로, Heuristic한 방법에 의해서 찾게된다. -> 최적의 값을 찾아야 한다. 데이터로부터 자동으로 학습시킬수 없지만, 모델의 성능에 영향을 끼친다. Critical한 파라미터를 인지하고 튜닝하는 것이 중요!

학습률(learning rate), 에포크 수(epoch), 배치사이즈(batch size), 활성화함수(activation functin), 최적화 알고리즘(optimizer), 손실함수(loss function)정규화(regularization), 드롭아웃(drop out), 신경망 모델의 깊이 및 넓이(network depth & width)

**\* Learning Rate**

Learning rate in gradient descent

(Loss를 로 미분)

- Learning Rate (η, 에타) : 하이퍼 파라미터. 기울기에 따른 경사도 하강에서 파라미터 업데이트의 크기를 정하는역할

학습률이 너무 작으면 수렴이 너무 늦고, local minima에 빠질수 있으며, 학습률이 너무 크면, loss가 발산할 수 있다. 너무 크거나 작은 LR은 학습을 방해한다. 데이터나 신경망의 구조가 바뀔떄마다 LR 튜닝이 필요하다. 하지만 최적의 LR인지 확신할 수 없다.

**\* SGD with Momentum (SGDM)**

관성을 이용하면 더 빨리 수렴하거나? Local minima를 탈출할수 있지 않을까? -> 움직이던 속도 그대로 진행하면 가속도가 붙으니까 가던 방향으로 계속가자!

- Momentum: gradient를 누적합한다. 위아래로 강하게 진동하면서 이동한다. 하지만 위아래는 누적합에 의해서 상쇄되고, minima에 가는 것이 계속 더해진다. 그에따라 local minima든지 global minima든지 빠르게 도달할 수 있다.

**\* Adaptive Learning – Rate) Adam (Momentum + Adagrad)**

Leaning Rate를 정해주는건 없을까? ->Update가 많았던 파라미터는 점차 작은 LR을 갖도록하자!

**학습 초반에는 큰 LR, 후반에는 작은 LR로 최적화하자!**

학습 초반에 작은 LR은 진행을 더디게하고, 학습 후반의 큰 LR은 더 좋은 loss를 얻지 못한다.

**Wight param마다 별도의 LR을 갖게한다.** 파라미터 업데이트가 될수록 LR이 작아진다.

SGD에 비해서 수렴이 빠르고 편리하다. 하지만 복잡한 모델에서는 성능이 떨어진다

Adam도 learning rate가 사실은 hyper parameter. 나중에 transformer에서 튜닝이 필요

요즘에는 Adam과 SGD를 많이 사용한다.

**\* Overfitting**

학습 데이터의 loss평균을 보고 모델의 학습여부를 판단하며, loss값을 최소로 하는 것이 목표다. 하지만, 학습 데이터에 편향, noise가 있다면 어떻게될까? 그것 또한 학습하게 된다. 우리의 목표는 unseen data에 대해 좋은 prediction을 하고, Generalization하는것이다. Training error을 줄이는게 최종 목표가 아니라, 좋은 예측을 하는 것이 목표다. 즉, epoch이 끝나면 validation error를 평가하여 generalization을 평가한다. (Training error가 낮지만, validation error가 높다면 Generalization을 잘하고 있지 못한것이다.) ->Validation Loss를 최소화하는 파라미터를 찾자!!

Overfitting이란 Training error가 generalization error에 비해서 현격히 늦어지는 현상이다.

**Overfitting은 모델이 학습을 너무나도 잘돼서, 학습에 불필요한 bias, noise까지 학습한다. 따라서 테스트 데이터나 새로운 데이터에 대해서 성능이 저하되는 현상이다.** 이것은 underfitting보다 훨씬 나으며(학습을 제대로 수행하지 못한 상태니까), overfitting을 없애기 위해 고려하면 된다.

Overfitting을 방지하는 방법!

더 많은 데이터 수집, 데이터 증강(Augmentation), 모델의 복잡성 줄이기(은닉층or유닛수 줄이기), 드롭아웃(Drop out), 규제(Regularization), 적절한 하이퍼 파라미터 튜닝, 조기종료(Early Stopping)

**\*Regularization**

Overfitting을 피하기 위해, generalization error를 낮추기 위한 방법/알고리즘

이 과정에서 Train error가 높아질 수 있다. Training error 최소화를 방해하는 형태기 때문이다.

Generalization이 잘된 모델은 noise에 robust하다.

1. 데이터를 통해

- Data Augmentation: 입력 데이터에 noise를 추가하자! 쓸모 없는 특징까지 배우는 것을 막기위해

2. Loss함수를 통해

- Weight Decay: weight 에다가 penalty를 부여해 커지는 것을 막겠다. 잘 안쓴다. Loss함수에 최소화를 방해하는 term을 넣는 것이다. 너무 강력한 방법이고, Loss를 낮추면 weight가 커지고, weight가 낮아지면 Loss가 커지는 딜레마

3. Neural Network layer를 통해 (Layer를 추가함으로써)

- Drop Out: 임의의 노드를 동작하지 않기 위해서 사용.

Layer -> 활성함수 -> Dropout -> Layer

- Batch Normalization: 정규화(standardization)후, shifting + scaling

Mini\_batch내에서 평균(μ)과 표준편차(σ)를 계산한다. 따라서 batch\_size는 1보다 커야한다

Drop out의 대체가 가능하며, 보통 activation function 앞이나 뒤에 사용한다.

하지만 왜 동작하는지는 잘 모르며, RNN에는 사용할 수 없다. (RNN에서는 Layer Normalization)

4. 학습방식 또는 추론 형태를 통해

- Early stopping

- Bagging & Ensemble