Norm

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

p = (3, 1, -3), L1 Norm = | 3 + 1 + 3 | = 9

p = (3, 1, -3), L2 Norm = | 9 + 1 + 9 | = 19

L1 Regularization(Lasso)

- L1 Loss : 원소들의 차이의 절대값의 합

  p = (3, 1, -3), q = (5, 0, 7), L1 Loss = 2 + 1 + 10 = 13

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n} |\theta_j|$$

- 기존 파라미터 정보의 일부(λ)만큼을 cost로 삼음
- 이 cost로 GD(Gradient Descent) 해주면, 기존 cost function의 값이 작아지는 방향으로만 진행되는 것이 아니라, 가중치 또한 작아지는 방향으로 학습됨. (중요, regularization term을 미분시 상수가 됨)

L2 Regularization(Ridge)

- L2 Loss : 루트(원소들의 차이의 제곱의 합)

  p = (3, 1, -3), q = (5, 0, 7), L2 Loss = sq_root(2^2 + 1^2 + 10^2) = sq_root(13)

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n} \theta_j^2$$

- 기존 파라미터 정보 제곱의 일부(λ)만큼을 cost로 삼음
- 이 cost로 GD할 때,

$$w \rightarrow w - \eta \frac{\partial c0}{\partial w} - \frac{\eta\lambda}{n}$$

$$= \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}$$

GD는 다음과 같이 이루어지고, 이 또한 가중치가 작아지는 방향으로 학습 됨. (중요, regularization term을 미분시 가중치가 살아있음)

Note

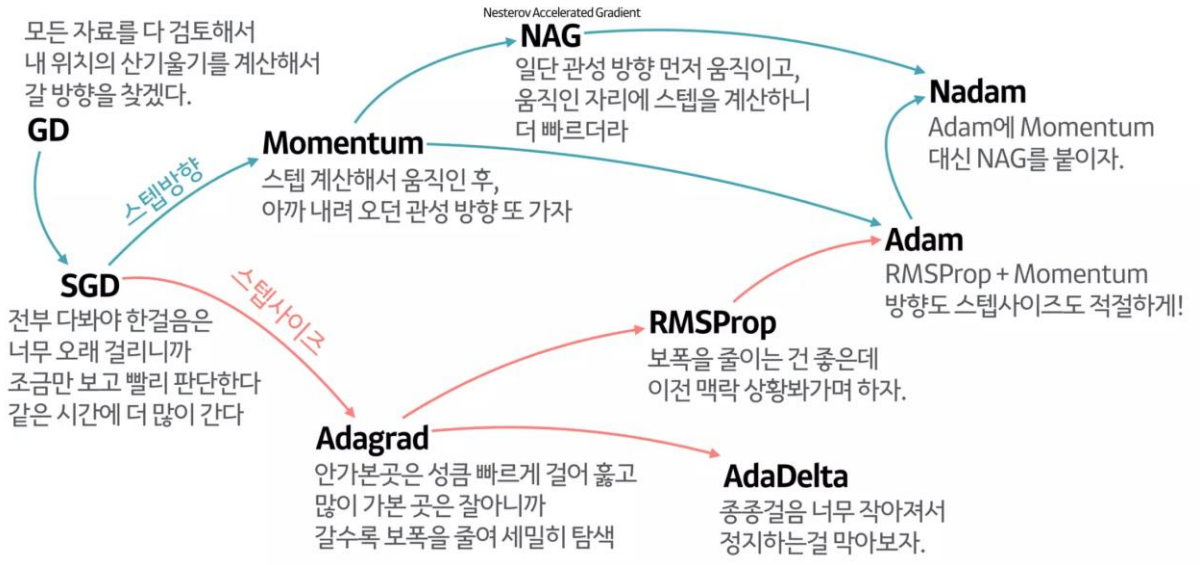
- L1 Loss는 상수값을 계속 빼주게 되므로 어떤 가중치는 0이 되고, 영향을 크게 미치는 feature들의 가중치만 전달될 수 있게 한다. 여러 feature 중 중요한 feature만 남기고 싶을 때 쓴다.
- L2 Loss는 미분시 남은 가중치를 반영하기 때문에 가중치의 크기에 따른 가중치 감소가 이루어진다. (Weight Decay) 그래서 전체적인 가중치의 절대값을 감소시킬 수 있고, 결정 경계가 덜 구불구불하게 되도록 한다.
- L2 Loss는 Outlier에 민감(제곱 때문에)하다.
- Outlier을 신경써야 한다면 L2 Loss를, Outlier를 신경쓰고 싶지 않다면 L1 Loss를 사용
- Outlier에 신경써야하면서 또 일반적인 경우엔 L2 Loss를 쓴다.
- Outlier를 무시하고 중요한 feature만 생각하고 싶을 땐 L1 Loss를 쓴다.

Others

Overfitting을 해결하는 방법은 Regularization 외에도,

- 데이터수 늘리기
- 모델 Complexity 줄이기
- Dropout
- Batch Normalization
- Early Stopping

와 같은 기법을 사용할 수 있다.



Momentum

local minimum에 빠질 위협에 대비해서 이전 gradient들도 계산에 포함하여(관성) 파라미터를 업데이트 하는 optimizer

$$V \leftarrow \alpha V - \eta \frac{\partial L}{\partial W} \quad , \quad W \leftarrow W + V$$

α는 momentum term으로 보통 0.9 사용

```
class Momentum:
    def __init__(self, ir = 0.01, momentum = 0.9):
        self.lr = lr
        self.momentum = momemtum
        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 parmas.keys():
                self.v[key] = self.momentum * self.v[key] - self.lr * grads[key]
                params[key] += self.v[key]
```

AdaGrad(Adaptive Gradient)

학습 진행 중에 learning rate를 감소시키는 방법을 학습률 감소(learning rate decay)라고 한다. 이를 발전시킨 방법이 AdaGrad이며, AdaGrad는 각 매개변수에 맞춤형 learning rate를 제공한다.

$$h = h + \frac{\partial L}{\partial W} \odot \frac{\partial L}{\partial W}$$

$$W = W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$$

기존 기울기값을 제곱하여 h에 쌓고 이를 토대로 업데이트될 기울기를 감소시킴.

변화량이 클수록 더 많이 업데이트량이 더 감소하며, 계속 진행되면 갱신량이 0이 됨.

```
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(self.h[key]) + 1e-7)
```

RMSProp

AdaGrad의 단점을 보완하였다. 먼 과거의 기울기는 조금 반영하고 최신의 기울기를 많이 반영한다.(지수이동평균)

$$h_i = ph_{i-1} + (1-p)\frac{\partial L_i}{\partial W} \odot \frac{\partial L_i}{\partial W}$$

$$W = W - \eta \frac{1}{\sqrt{h}} \frac{\partial L}{\partial W}$$

p가 크면 AdaGrad의 형태를 띄면서 과거의 기울기를 모두 반영하게 됨

p를 줄이면 과거의 기울기는 줄고 현재의 기울기가 커지기 때문에 AdaGrad의 '학습률 0으로 수렴' 문제를 해결할 수 있다.

Adam

Momentum과 RMSProp을 합친 Optimizer

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta))^2$$

$$\hat{m_t} = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v_t} = \frac{v_t}{1-\beta_2^t}$$

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v_t} + \epsilon}}\hat{m_t}$$

1. Momentum에서 사용되는 m과 RMSProp에서 사용되는 v를 구하여,
2. Bias Correction을 해준 뒤, 가중치를 업데이트 한다.

Momentum과 RMSProp의 m과 v는 0으로 초기화 된다. 이 때문에 0으로 bias된 이 값들을 correction 해주는 과정이 필요하다.

beta값들은 1보다 작은 수이고, 이전 값들을 서서히 잊게 하기 위한 변수이다. beta가 0에 가까울수록 현재의 기울기를 많이 반영하게 되고, 1에 가까울수록 과거의 기울기를 많이 반영한다.

만약 Bias Correction을 해주게 되면 beta = 0.7이라고 할 때, 1-(0.7)^t 을 나눠주게 되는데, 이를 통해 m1_hat을 구해보자.

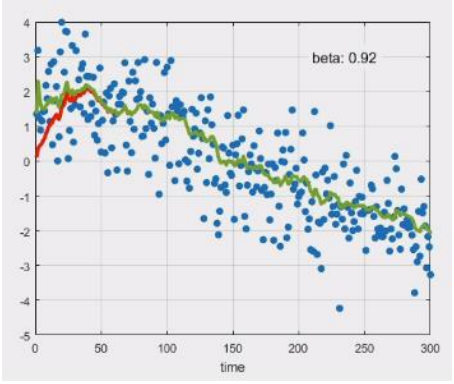
m1 = (0.7)*0 + (1-0.7)(기울기 변화량) = (0.3)(기울기 변화량)

m1_hat = (0.3)(기울기 변화량) / (1-0.7) = 기울기 변화량

m1과 m1_hat을 비교했을 때 m1_hat은 기울기 변화량을 온전히 보전하는 것을 알 수 있다.

모멘텀에 사용하는 베타1, RMSProp에 사용하는 베타2 그리고 입실론의 추천값은 다음과 같다.

$$\begin{cases} \beta_1 : 0.9 \\ \beta_2 : 0.99 \\ \epsilon : 10^{-8} \end{cases}$$



초록색 선이 Bias Correction된 선

Train/Validation/Test를 분리하는 이유

Train Set : 모델 학습시 유일하게 사용되는 데이터셋

Validation Set : 학습이 완료된 모델을 검증할 때 사용되는 데이터셋

Test Set : 학습과 검증이 끝난 모델의 성능을 평가하기 위한 데이터셋

Validation Set과 Test Set은 모델을 학습시키는데 관여하지 않음

보통 (Train+Validation)과 Test를 8:2로 나누고 Train과 Validation을 8:2로 나누거나,

Train : Validation : Test를 6 : 2 : 2 비율로 나누어서 쓴다.

Validation Set을 사용하는 이유

Test Set은 모든 훈련을 마치고 성능평가를 위해서 모델이 보지 못했던 데이터를 제공하는데 사용된다. Train Set으로 에폭을 돌다보면 언제 overfitting될지 모르기 때문에 에폭별로 보지 못했던 데이터를 이용하여 검증하는 과정이 있어야 한다. 이러한 역할을 Validation Set이 하고, 검증 과정에서 overfitting이 일어나는 것이 확인되면 학습을 중단시켜야 한다.

우리는 Validation Set을 보면서 Model을 튜닝하기 때문에 Validation Set에 더 적합하도록 튜닝시키게 되므로 엄연히 학습 과정에서 사용되는 데이터셋이다. 따라서, Test Set과는 다른 역할을 한다.

결국,

Validation Set은 overfitting에 대응하면서,

보지 못했던 데이터에(unseen data)에 대해 좋은 성능이 나오는지 확인하기 위해 사용된다.

K-Fold Cross-Validation

데이터셋을 Train, Test로만 분리한다. 이 때, K=5라면 Train : Test = 4 : 1 로 나눈다. 이 때, 서로 다른 5개의 Train/Test Set 조합을 만들 수 있는데, Train을 통해 얻어진 모델로 Test를 할 때 구할 수 있는 5개의 성능 지표로 모델을 검증하는 것이 K-Fold Cross-Validation 기법이다.