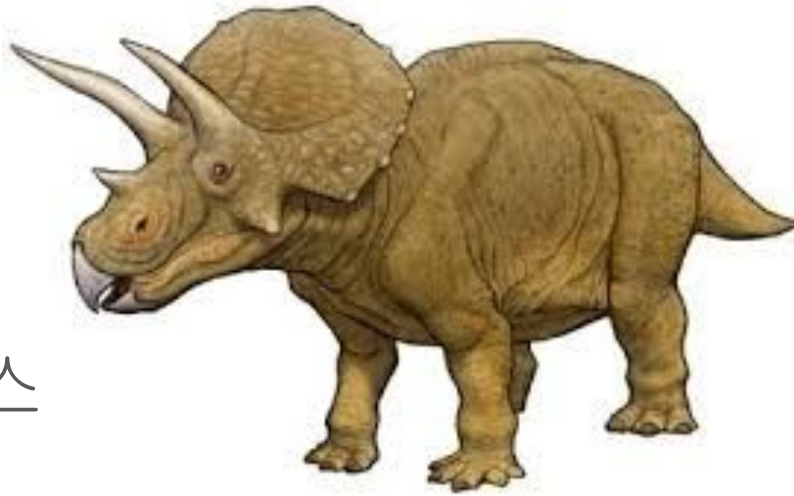
Yee…!!

발표!

트리케라톱스

트리케라톱스가 야행성이였나요 ?
꿁쩍쓰))



INDEX

- Keras! Pytorch랑 무엇이 다른가! (영채)
- 자주 쓰이는 텐서연산과 쓰임 (윤지)
- 자주쓰는 Optimizer
  - SGD(현동)
  - RmsProp(현영)
  - Adam(해리)

Keras! Pytorch랑 무엇이 다른가!

Keras

1. **Google**의 지원을 받고있다.
2. 표준레이어를 사용하여 설계를 하는것이 매우 쉽다.
3. **높은 추상화** 기능들을 제공한다.
4. tensorflow backend기반으로 돌릴 수 있는 keras는 모바일과, 웹으로의 **추출**이 유리하기에 실제 서비스에 유리하다

Pytorch

1. **Facebook**의 지원을 받고있다.
2. 저수준의 **유연한 접근성**을 제공하기에 **수학적인 처리**를 하기 좋다.
3. Numpy와 유사하여 코드진행을 keras보다 **로우레벨**에서 컨트롤할 수 있다.

Keras! Pytorch랑 무엇이 다른가!

Keras는 end-to-end로 일어나는 모델들의 높은 **추상화**로 **매우 생산성** 있는 코드를 짜는것이 가능합니다!

반면 Pytorch는 모델의 정의 부터 계산 방법을 정의하고 역전파 과정까지 **세부적인 설정**을 하기 좋게 만들어져 매우 유연하기에 **수학적인 처리**를 하기 유리합니다!

```
network = models.Sequential()

network.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
network.add(MaxPool2D())
network.add(Conv2D(16, (3, 3), activation='relu'))
network.add(MaxPool2D())
network.add(Flatten())
network.add(Dense(10, activation='softmax'))

network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

network.fit(train_image, train_labels, epochs=10, batch_size=128)
```

```
# 모델정의
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.ReLU()
        ).to(device)

        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 16, kernel_size=3, stride=1, padding=1),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.ReLU()
        ).to(device)

        self.fc = nn.Linear(7*7*64, 10, bias=True).to(device)

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out).view(-1, 64*7*7)
        out = self.fc(out)
        return out

model = Model()
```

```
# optimizer 설정
optimizer = optim.RMSprop(model.parameters(), lr=0.001)

# batch_size 설정
dataloader = torch.utils.data.DataLoader(mnist_train, 128, shuffle=True, drop_la

# epochs loop
n_epoch = 10
for epoch in range(n_epoch+1):
    batch_loss = 0

    # batch loop
    for idx, mini_batch in enumerate(dataloader):
        x_train, y_train = mini_batch
        x_train = x_train.view(-1,3,32,32).to(device)
        y_train = y_train.to(device)

        pred = model(x_train)

        # loss 계산
        loss = F.cross_entropy(pred, y_train)

        # 역전파 & 갱신
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        batch_loss += loss.item()

    print('epoch :',epoch, 'loss :', batch_loss/len(dataloader))
    batch_loss = 0

# 학습 끝
```

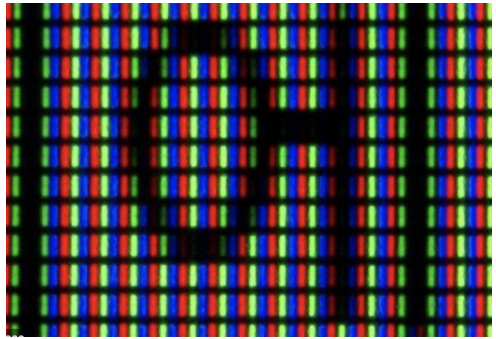
오른쪽과 왼쪽은 같은 기능을 하는 모델입니다.
size 설정에 약간 오타가 있네요? 넘어가 주세요 ^^

Keras! Pytorch랑 무엇이 다른가!

헷갈리기 너무 좋은 Image Size

Keras (channel-last)

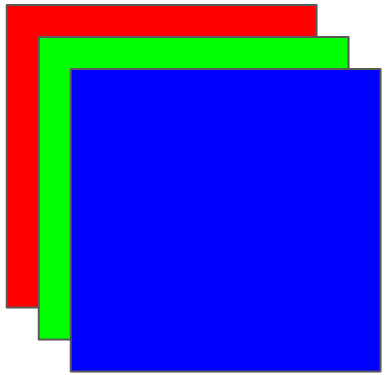
(samples, height, width, channels)



(height, width, channels)

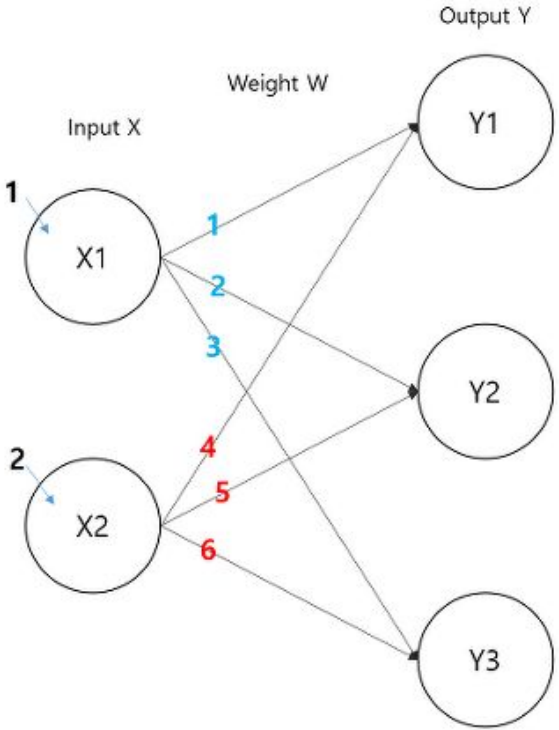
Pytorch (channel-first)

(samples, channels, height, width)



(channels, height, width)

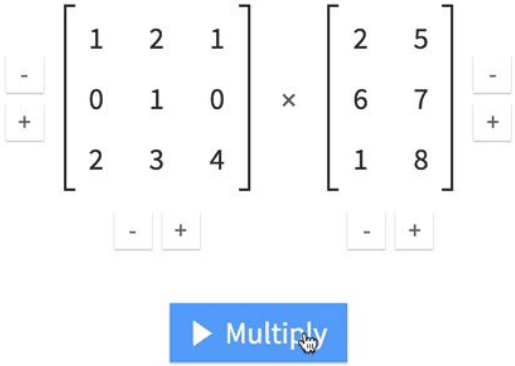
텐서의 연산



X	*	W	=	Y
1 x 2	*	2 x 3	=	1 x 3

$$(1 \quad 2) \qquad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad (9 \quad 12 \quad 15)$$

행렬곱



$$a=\begin{bmatrix}1 & 3\\ 2 & 4\end{bmatrix} \qquad b=\begin{bmatrix}5 & 7\\ 6 & 8\end{bmatrix}$$

$$ab=\begin{bmatrix}1\times5+3\times6 & 1\times7+3\times8\\ 2\times5+4\times6 & 2\times7+4\times8\end{bmatrix}=\begin{bmatrix}23 & 31\\ 34 & 46\end{bmatrix}$$

reshape와 transpose

```
import numpy as np
```

```
x=np.array([[0,1,2],
            [2,3,4],
            [4,5,6],
            [7,8,9]])
```

```
print(x.shape)
```

```
(4, 3)
```

```
x=x.reshape((2,6))
```

```
print(x)
```

```
[[0 1 2 2 3 4]
 [4 5 6 7 8 9]]
```

```
print(x.shape)
```

```
(2, 6)
```

reshape와 transpose

```
x=np.transpose(x)
```

```
print(x)
```

```
[[0 4]
 [1 5]
 [2 6]
 [2 7]
 [3 8]
 [4 9]]
```

```
print(x.shape)
```

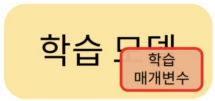
```
(6, 2)
```

경사하강법

머신러닝의 엔진

데이터를 바탕으로 모델(파라미터)학습하기 **학습 매개변수**

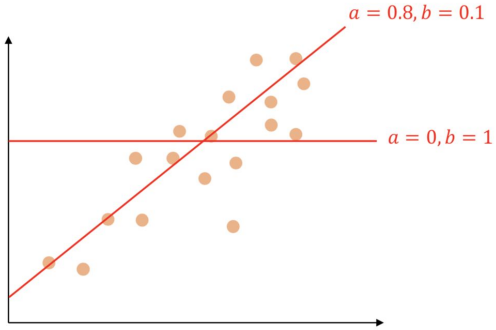
학습 매개변수(Trainable Parameters) : 학습 과정에서 값이 변화하는 매개 변수. 이 값이 변화하면 알고리즘 출력이 변화



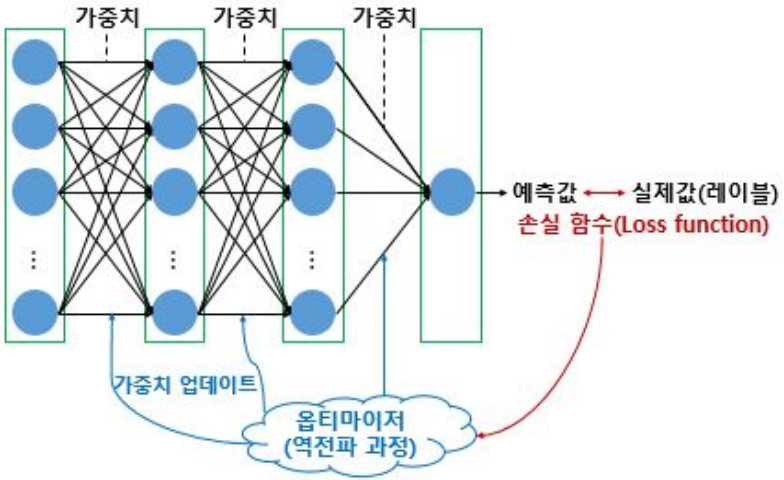
$$y = ax + b$$

출력 입력

학습 매개변수



손실 함수

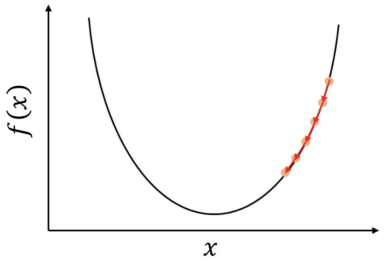


MSE(평균 제곱 오차) CEE(교차 엔트로피 오차)

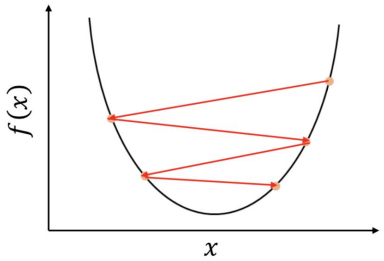
$$E = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 \qquad E = -\sum_{i=1}^{n} t_i \log y_i$$

학습률의 선택

$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} - \alpha \nabla f(\boldsymbol{x}_{n-1})$$



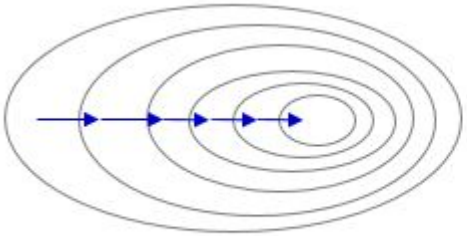
α가 너무 작은 경우 (수렴이 늦음)



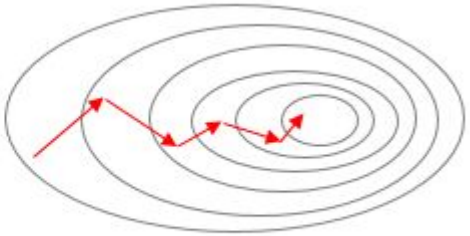
α가 너무 큰 경우 (진동)

이제 학습률은 단지 기울기에 곱하는 상수임을 알 수 있다.

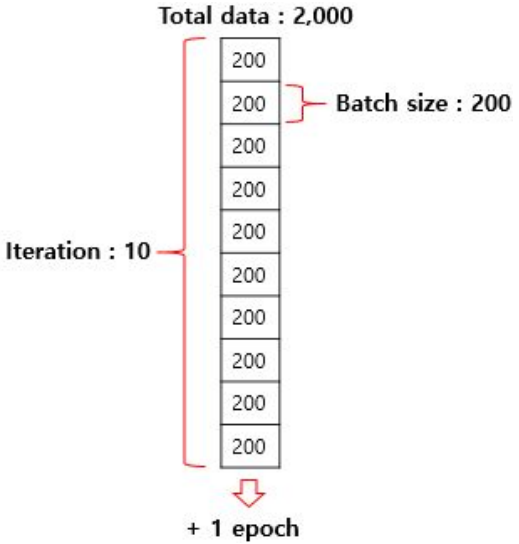
SGD 확률적 경사 하강법



경사 하강법



SGD



장점:
빠르게 모델을 업데이트 함으로써 학습시간을 단축
Local minima 탈출 유리

SGD 단점

단점: 연산수가 많아진다. (업데이트가

Momentum

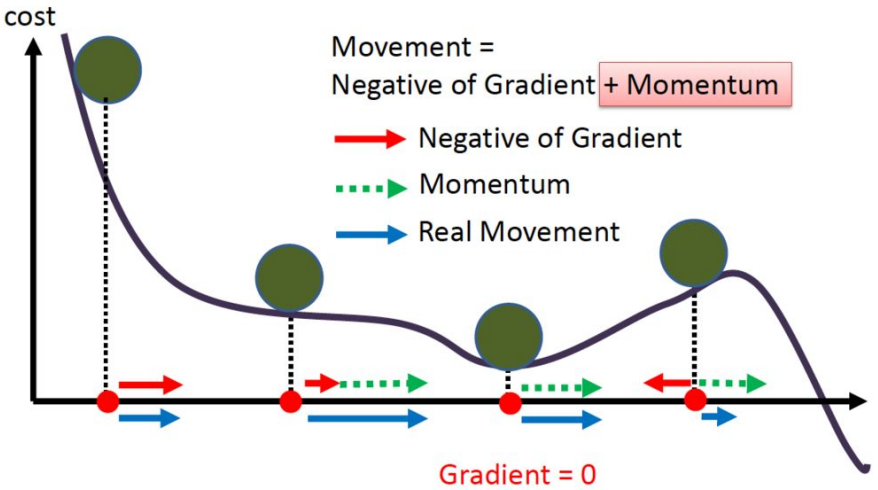
업데이트할 다음 가중치를 계산할 때 현재 그래디언트 값만 보지 않고 이전에 업데이트된 가중치를 여러 가지 다른 방식으로 고려하는 SGD변종이 다수 존재한다.

Ex) 모멘텀을 사용한 SGD, Adagrad, RMSProp

→ 최적화 방법 또는 옵티마이저 라고 부른다.

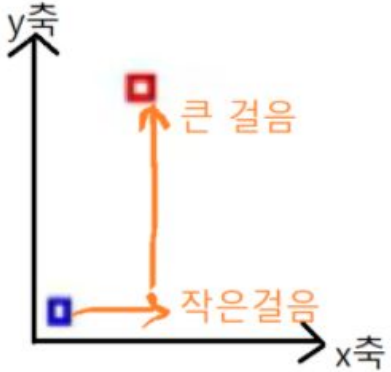
Momentum

- W 이동 과정에 관성 부여
- 즉 W를 업데이트 할 때에 이전 단계의 업데이트 방향을 반영
- SGD의 수렴 속도와 지역 최솟값 문제 해결



AdaGrad

- 학습률이 너무 작으면 학습 시간 너무 길고
- 학습률이 너무 크면 발산
- 이러한 문제를 학습률 감소로 해결
- 처음에는 목적지를 모르니 성큼성큼 걷고, 목적지에 가까워질수록 보폭을 조금씩 줄이자.
- 많이 변화한 변수는 최적값에 근접했을 것이고, 적게 변화한 변수는 최적에 아직 멀었다는 가정!



→ 학습률 많이 적게, 작은 걸음으로 이동하며 세밀한 값 조정

→ 적게 변화한 변수는 학습률 조금 적게, 큰 걸음으로 이동하며 빠르게 loss줄임

- 무한히 학습하면 어느순간 학습이 되지 않는 문제 발생 → RMSProp에서 개선

AdaGrad

> 같은 입력 데이터가 여러번 학습되는 학습모델에 유용하게 쓰임

- 대표적으로 언어와 관련된 word2vec이나 GloVe에 유용함
- 학습 단어의 등장 확률에 따라 변수의 사용 비율이 확연하게 차이나기 때문에
  많이 등장한 단어는 가중치를 적게 수정하고 적게 등장한 단어는 많이 수정할 수 있기 때문

Keras 소스 코드

```
1 keras.optimizers.Adagrad(lr=0.01, epsilon=1e-6)
```

RMSProp

- AdaGrad는 스텝이 많이 진행되면 누적치 h_n이 너무 커져서 학습률이 너무 작아져 학습이 거의 되지 않는 문제가 있음
- 이를 보완하기 위해 RMSProp은 Adagrad와 달리 기울기를 단순 누적 x
- AdaGrad보다 최근 값을 더 잘 반영하기 위해 최근 값과 이전 값에 각각 가중치를 주어 계산하는 방법
- **지수 이동 평균** 사용

RMSProp

- 쉽게 말하면, 내분인데, 더 중요한 변수에 내분점을 가까이 찍어서 가중치를 더 주는 것

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

- γ가 크다→빨간점에 더 가까이→ 과거 중요하게 생각
- γ가 작다→$(1-\gamma)$가 크다→파란점에 더 가까이→ 현재 중요하게 생각

RMSProp

> 일반적으로 순환 신경망(Recurrent Neural Networks)의 옵티마이저로 많이사용

- RMSProp을 사용할 때는 학습률을 제외한 모든 인자의 기본값을 사용하는 것이 권장

Keras 소스 코드

```
keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)
#하이퍼-파라메터인 Rho는 γ(감마)를 뜻합니다.
#Rho is a hyper-parameter which attenuates the influence of past gradient.
```

Reference
개념 : https://twinw.tistory.com/247
수식 : https://www.youtube.com/watch?v=s8dzhdJ3fqg
코드 : https://keras.io/ko/optimizers/

Adam(Optimizer) 소개

Adam이 나오게 된 배경!

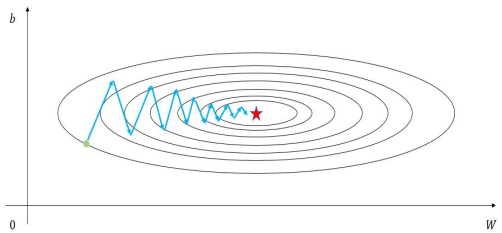
- Adam이라는 이름은 **Ada**ptive **M**oment Estimation에서 유래한다.
- 당시 가장 많이 쓰이던 **AdaGrad와 RMSProp의 장점을 합쳐보자**는 발상에서 출발
- 2014년에 논문 발표, 2015 ICLR이라는 학회에서 주목을 받았음.
- 분야를 가리지 않고 **무난**하게 사용될 수 있는 Optimizer 중 하나!

참고자료1: (paper) ADAM: A Method For Stochastic Optimization

참고자료2: Andrew Ng, Mini Batch Gradient Descent~ Adam Optimization Algorithm (C2W2L01~08)

Adam(Optimizer) 상세1

RMSProp with momentum



momentum 을 포함한 경사하강법

$$V_{dw(t)} = \beta_1 V_{dw(t-1)} + (1-\beta_1) dW_{(t)}$$

$$V_{db(t)} = \beta_1 V_{db(t-1)} + (1-\beta_1) db_{(t)}$$

$$W := W - \alpha V_{dw(t)}$$

$$b := b - \alpha V_{db(t)}$$

RMSProp

$$S_{dw(t)} = \beta_2 S_{dw(t-1)} + (1-\beta_2) dW^2_{(t)}$$

$$S_{db(t)} = \beta_2 S_{db(t-1)} + (1-\beta_2) db^2_{(t)}$$

$$W := W - \alpha \frac{dW_{(t)}}{\sqrt{S_{dw(t)} + \epsilon}}$$

$$b := b - \alpha \frac{db_{(t)}}{\sqrt{S_{db(t)} + \epsilon}}$$

Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1-\beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1-\beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

⇩ Adam의 Pseudo Code

Adam

$$V_{dw(t)} = \beta_1 V_{dw(t-1)} + (1-\beta_1) dW_{(t)}$$

$$V_{db(t)} = \beta_1 V_{db(t-1)} + (1-\beta_1) db_{(t)}$$

$$S_{dw(t)} = \beta_2 S_{dw(t-1)} + (1-\beta_2) dW^2_{(t)}$$

$$S_{db(t)} = \beta_2 S_{db(t-1)} + (1-\beta_2) db^2_{(t)}$$

$$W := W - \alpha V_{dw(t)} / \sqrt{S_{dw(t)} + \epsilon}$$

$$b := b - \alpha V_{db(t)} / \sqrt{S_{db(t)} + \epsilon}$$

수식 출처: https://angeloyeo.github.io/2020/09/26/gradient_descent_with_momentum.html

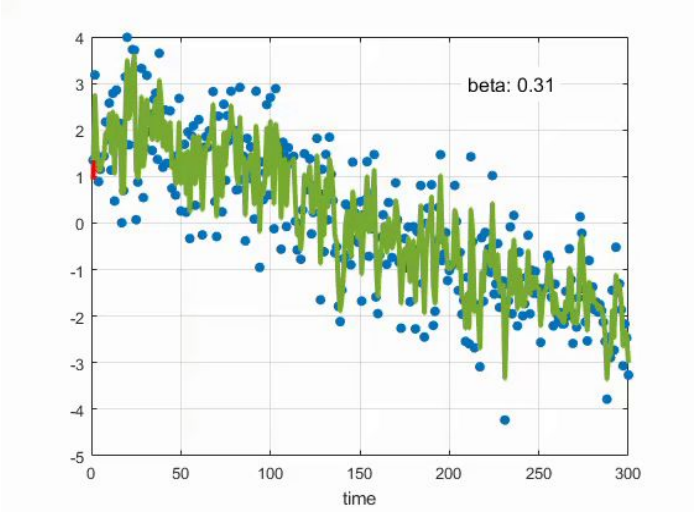
Adam(Optimizer) 상세2

RMSProp과 차이점을 만들어내는 부분

AdaGrad: An algorithm that works well for sparse gradients is AdaGrad (Duchi et al., 2011). Its basic version updates parameters as $\theta_{t+1} = \theta_t - \alpha \cdot g_t / \sqrt{\sum_{i=1}^{t} g_t^2}$. Note that if we choose β_2 to be infinitesimally close to 1 from below, then $\lim_{\beta_2 \to 1} \widehat{v}_t = t^{-1} \cdot \sum_{i=1}^{t} g_t^2$. AdaGrad corresponds to a version of Adam with $\beta_1 = 0$, infinitesimal $(1 - \beta_2)$ and a replacement of α by an annealed version $\alpha_t = \alpha \cdot t^{-1/2}$, namely $\theta_t - \alpha \cdot t^{-1/2} \cdot \widehat{m}_t / \sqrt{\lim_{\beta_2 \to 1} \widehat{v}_t} = \theta_t - \alpha \cdot t^{-1/2} \cdot g_t / \sqrt{t^{-1} \cdot \sum_{i=1}^{t} g_t^2} = \theta_t - \alpha \cdot g_t / \sqrt{\sum_{i=1}^{t} g_t^2}$. Note that this direct correspondence between Adam and Adagrad does not hold when removing the bias-correction terms; without bias correction, like in RMSProp, a β_2 infinitesimally close to 1 would lead to infinitely large bias, and infinitely large parameter updates.

저자는 Adam의 파라미터 업데이트 식에서
베타1과 베타2를 특수하게 정하면
(b1은 0, b2는 1에 아주 가까운 값)
AdaGrad와 유사하다고 설명하고 있음.

beta 값에 따른 편향 보정의 효과- beta가 클수록 큰 차이가 난다.
빨강 : 보정 안 함
초록 : 보정 함



영상 출처: https://angeloyeo.github.io/2020/09/26/gradient_descent_with_momentum.html

실제로 Adam을 쓸 땐

하이퍼 파라미터 조율 관련

$$V_{dw(t)} = \beta_1 V_{dw(t-1)} + (1-\beta_1) dW_{(t)}$$

$$V_{db(t)} = \beta_1 V_{db(t-1)} + (1-\beta_1) db_{(t)}$$

$$S_{dw(t)} = \beta_2 S_{dw(t-1)} + (1-\beta_2) dW^2_{(t)}$$

$$S_{db(t)} = \beta_2 S_{db(t-1)} + (1-\beta_2) db^2_{(t)}$$

$$W := W - \alpha V_{dw(t)} / \sqrt{S_{dw(t)} + \epsilon}$$

$$b := b - \alpha V_{db(t)} / \sqrt{S_{db(t)} + \epsilon}$$

CLASS `torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)` [SOURCE]

Pytorch에서의 Adam

Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization. The implementation of the L2 penalty follows changes proposed in Decoupled Weight Decay Regularization.

Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*, *optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple*[*float*, *float*], *optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float*, *optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float*, *optional*) – weight decay (L2 penalty) (default: 0)

Adam [source]

Keras에서의 Adam

```
keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)
```

Adam 옵티마이저.

매개변수들의 기본값은 논문에서 언급된 내용을 따릅니다.