Contents

CHAPTER 6 학습 관련 기술들

신경망(딥러닝) 학습의 효율과 정확도를 높이는 방법들 학습

매개변수 업그레이드하는 여러 방법들

매개변수 초기값 잘 주는 방법들

배치 정규화

신경망 모델 오버피팅 방지하는 방법들

좋은 하이퍼 파라미터 설정 방법들

6.1.1 모험가 이야기

신경망 학습의 목적은 손실함수의 값을 가능한 한 낮추는(최소화하는) 매개변수를 찾는 것이라고 말할 수 있다. 이를 최적화 문제(Optimization problem)를 푼다고 말한다. 신경망 최적화 문제는 어려운 문제다. 매개 변수 공간이 넓고 손실함수가 복잡해서, 최적의 솔루션을 찾기 쉽지 않다.
교재에서, 최적화를 해야 하는 우리의 어려운 상황을 다음과 같이 모험가 이야기에 비유하였다.

색다른 모험가가 있습니다. 광활한 메마른 산맥을 여행하면서 날마다 깊은 골짜기를 찾아 발걸음을 옮깁니다. 그는 전설에 나오는 세상에서 가장 깊고 낮은 골짜기, '깊은 곳'을 찾아가려 합니다. 그것이 그의 여행 목적이죠. 게다가 그는 엄격한 '제약' 2개로 자신을 옭아맸습니다. 하나는 지도를 보지 않을 것, 또 하나는 눈가리개를 쓰는 것입니다. 지도도 없고 보이지도 않으니 가장 낮은 골짜기가 광대한 땅 어디에 있는지 알 도리가 없죠. 그런 혹독한 조건에서 이 모험가는 어떻게 '깊은 곳'을 찾을 수 있을까요? 어떻게 걸음을 옮겨야 효율적으로 '깊은 곳'을 찾아낼 수 있을까요?

6.1.2 확률적 경사 하강법(SGD)

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}}$$

[식 6.1]

```
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]
```

Stochastic Gradient Descent Method

파이썬 클래스에서 학습률을 인스턴트 변수로 설정함. self.lr

updata 메서드는 SGD 과정에서 반복해서 호출됨.

Prarms와 grads는 딕셔너리 변수로, params['W1'], grads['W1'] 등과 같이 가중치 매개변수와 기울기를 저장함.

6.1.2 확률적 경사 하강법(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]
```

SGD 클래스를 이용한 신경망 매개변수 코드는 대충~ 다음과 같이 사용한다.

```
network = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...) # 미니배치
    grads = network.gradient(x_batch, t_batch)
    params = network.params
    optimizer.update(params, grads)
    ...
```

매개변수 갱신 방법들을
신경망에서는 종종
Optimizer라 부른다.

6.1.3 SGD의 단점 탐색

SGD는 단순하고 구현이 쉽지만, 문제에 따라 비효율적일 때가 있다.

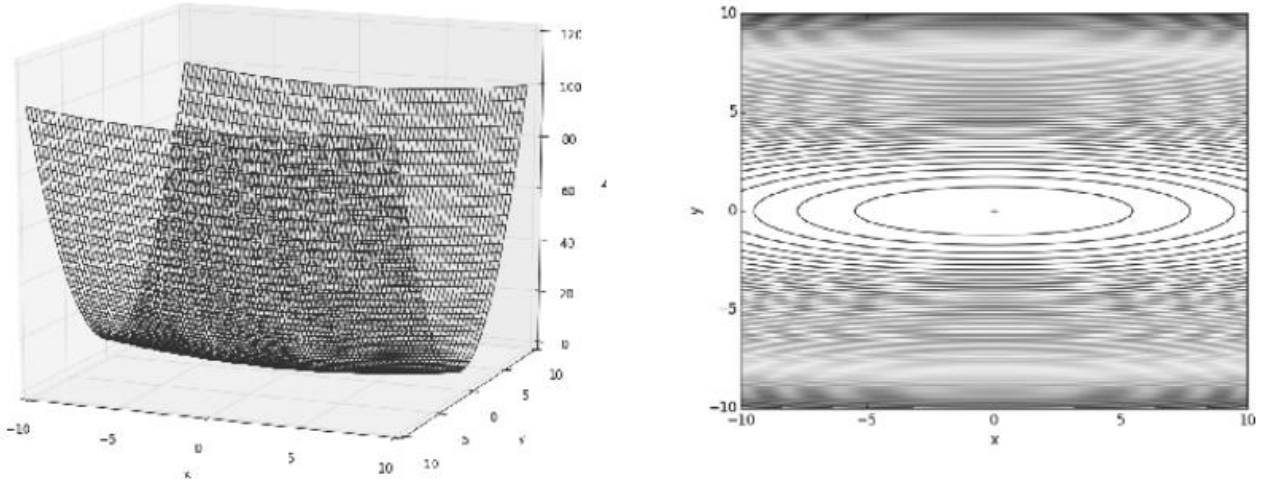
우리는 다음 [식 6.2] 함수 f 의 최솟값을 구하는 문제를 예를 들어 최적화 기법을 비교하고자 한다.

이 함수 모양은 [그림 6-1]에서 보듯이, 마치 밥그릇을 x축 방향으로 늘인 듯한 모습으로 등고선은 타원 모양이다.

이 함수의 기울기는 [그림 6-2]에서 보듯이, y 축 방향은 크고 x 축 방향은 작다. 기울기의 대부분은 최솟값이 되는 장소인 (0, 0)을 가리키지 않는다.

$$f(x,y) = \frac{1}{20}x^2 + y^2 \qquad \text{[식 6.2]}$$

그림 6-1 $f(x,y) = \frac{1}{20}x^2 + y^2$ 의 그래프(왼쪽)와 그 등고선(오른쪽)



6.1.3 SGD의 단점

SGD를 이용하여 최솟값을 찾아 보았다.

초깃값은 (-7, 2)로 설정하였다. [그림 6-3]과 같이 심하게 굽어진 움직임을 보이며 탐색(explore)하였다.

SGD는 비등방성 함수에 대해 매우 비효율적인 움직인다. 그러다, 벗어나서 최솟값을 못 찾을 위험도 있다.

그래서, SGD말고, 모멘텀, AdaGrad, Adam과 같은 대안들을 고안하였다.

그림 6-2 $f(x,y) = \frac{1}{20}x^2 + y^2$의 기울기

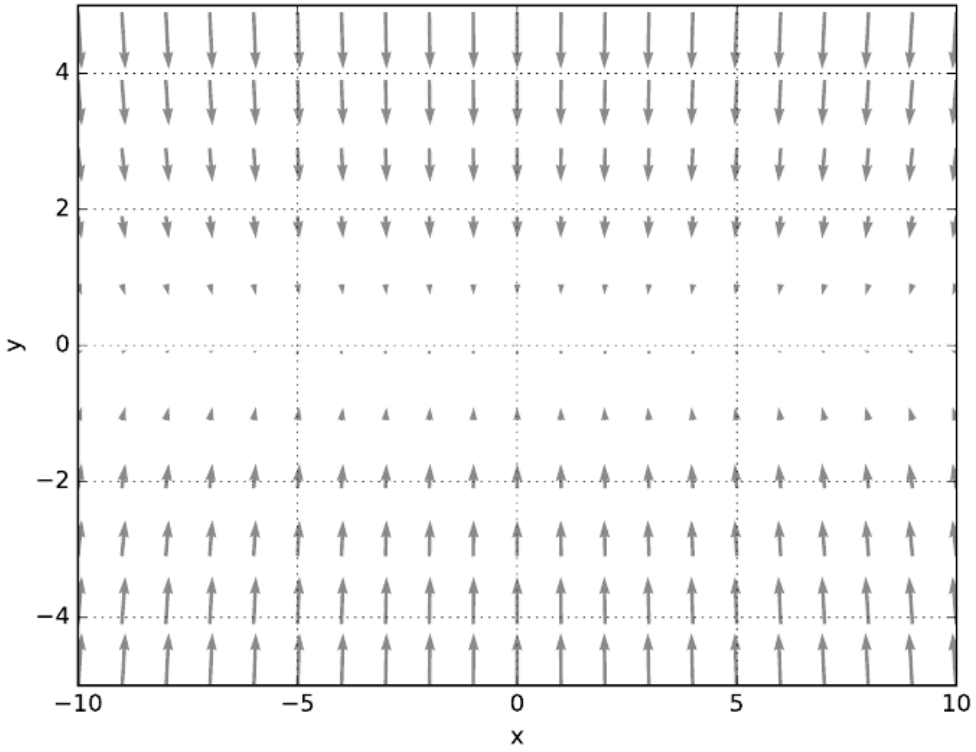
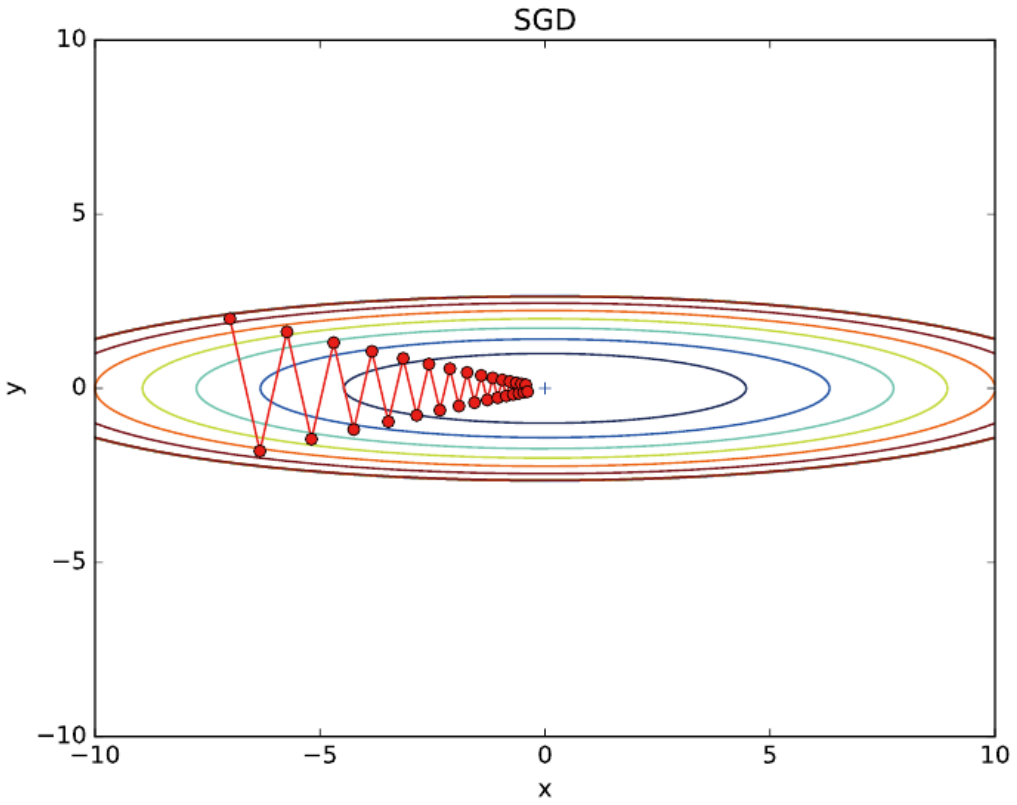


그림 6-3 SGD에 의한 최적화 갱신 경로 : 최솟값인 (0, 0)까지 지그재그로 이동하니 비효율적이다.



6.1.4 모멘텀(Momentum)

모멘텀은 물리학에서는 움직이고 있는 물체의 질량과 속도를 곱한 값이다. 어떤 물체가 움직일 때 계속 그 방향으로 움직이게 되는 경향이 있는데 그것을 모멘텀이라 한다. 수식으로 다음과 같이 쓸 수 있다.

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta \frac{\partial L}{\partial \mathbf{W}} \qquad [식\ 6.3]$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v} \qquad [식\ 6.4]$$

여기서 v가 모멘텀으로 기울기 방향으로 가속된다는 물리 법칙을 반영한다. 서서히 하강시키는 역할을 한다.

그림 6-4 모멘텀의 이미지 : 공이 그릇의 곡면(기울기)을 따라 구르듯 움직인다.

6.1.4 모멘텀(Momentum)

$$\mathbf{v} \leftarrow \alpha\mathbf{v} - \eta\frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{W} \leftarrow \mathbf{W} + \mathbf{v}$$

모멘텀의 갱신 경로는 공이 그릇 바닥을 구르듯이 움직인다. SGD와 비교하면 지그재그 정도가 덜하다.

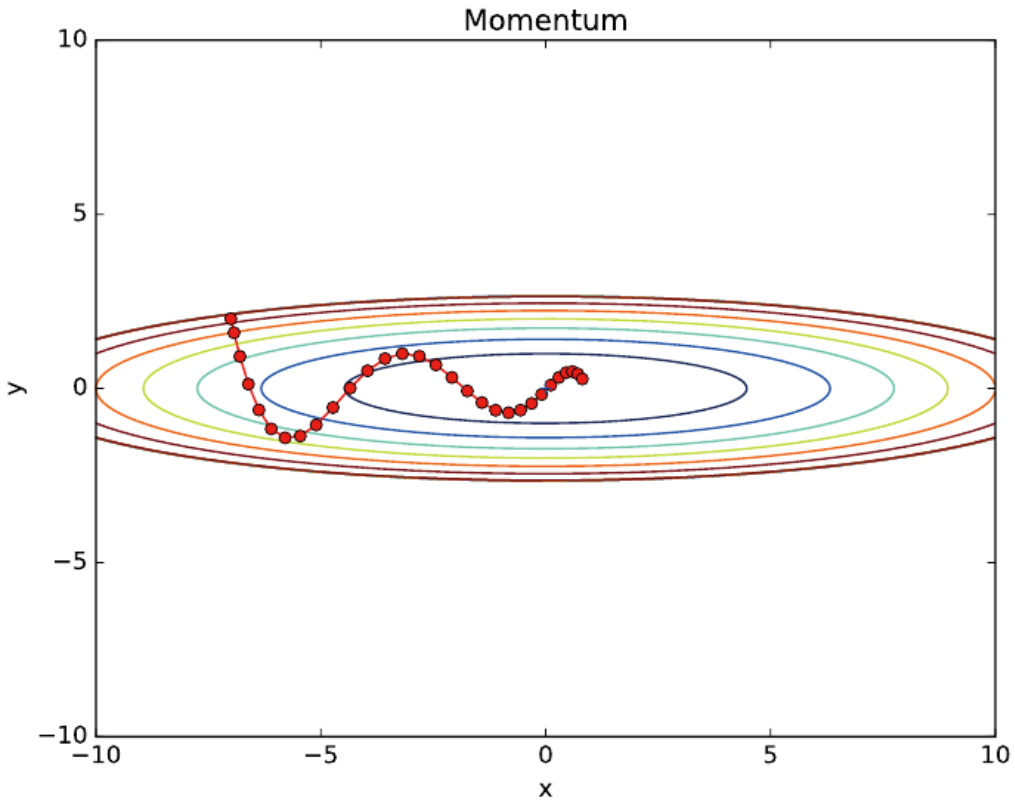
common/optimizer.py 참고.

```
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*grads[key]
            params[key] += self.v[key]
```

그림 6-5 모멘텀에 의한 최적화 갱신 경로



6.1.5 AdaGrad (Adaptive Gradient)

학습률 감소(learning rate decay) 방법으로 학습을 진행하면서 학습률을 점차 줄여가는 방법. 처음에는 크게 학습하다가 조금씩 작해 학습한다.

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}} \qquad [식\ 6.5]$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{\sqrt{\mathbf{h}}} \frac{\partial L}{\partial \mathbf{W}} \qquad [식\ 6.6]$$

$\frac{1}{\sqrt{\mathbf{h}}}$를 곱해 학습률을 조정한다.

NOTE_ AdaGrad는 과거의 기울기를 제곱하여 계속 더해갑니다. 그래서 학습을 진행할수록 갱신 강도가 약해집니다. 실제로 무한히 계속 학습한다면 어느 순간 갱신량이 0이 되어 전혀 갱신되지 않게 되죠. 이 문제를 개선한 기법으로서 RMSProp[7]이라는 방법이 있습니다. RMSProp은 과거의 모든 기울기를 균일하게 더해가는 것이 아니라, 먼 과거의 기울기는 서서히 잊고 새로운 기울기 정보를 크게 반영합니다. 이를 **지수이동평균**Exponential Moving Average, EMA이라 하여, 과거 기울기의 반영 규모를 기하급수적으로 감소시킵니다.

6.1.5 AdaGrad (Adaptive Gradient)

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

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

코드는 common/optimizer.py 참고. 이 클래스 사용방법은 p.911 하단에 있는 SGD 코드와 같다.

```
class AdaGrad:

    """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)
```

6.1.5 AdaGrad (Adaptive Gradient)

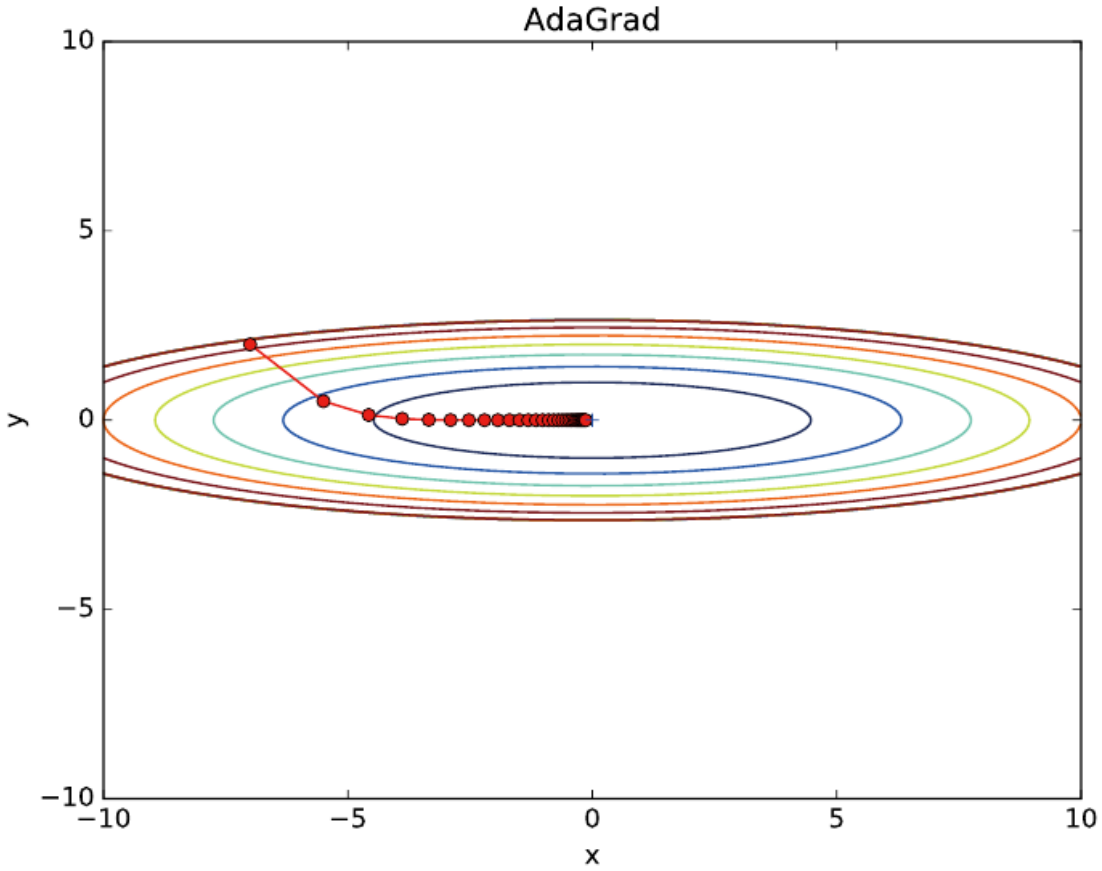
최솟값을 향해 효율적으로 움직임.

y축 방향 기울기가 커서 처음에는 크게 움직이지만, 그 큰 움직임에 비례하여 갱신 정도도 큰 폭으로 조정. 그래서, y축 방향 갱신 정도가 빠르게 약해지고, 지그재그 움직임이 많이 줄어듦

$$\mathbf{h} \leftarrow \mathbf{h} + \frac{\partial L}{\partial \mathbf{W}} \odot \frac{\partial L}{\partial \mathbf{W}}$$

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

그림 6-6 AdaGrad에 의한 최적화 갱신 경로



6.1.6 Adam (Adaptive Moment)

모멘텀은 공이 그릇 바닥을 구르는 듯 움직임. AdaGrad는 매개변수의 원소마다 적응적으로 갱신 정도 조정.

이 두 기법을 잘 합친 것인 Adam. 코드는 common/optimizer.py 참고.

```
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) / (1.0 - self.beta1**self.iter)

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

            params[key] -= lr_t * self.m[key] / (np.sqrt(self.v[key]) + 1e-7)

            #unbias_m += (1 - self.beta1) * (grads[key] - self.m[key]) # correct bias
            #unbisa_b += (1 - self.beta2) * (grads[key]*grads[key] - self.v[key]) # correct bias
            #params[key] += self.lr * unbias_m / (np.sqrt(unbisa_b) + 1e-7)
```

2015년 제안된 기법으로,

Adam은 파라미터를 3개 설정함.

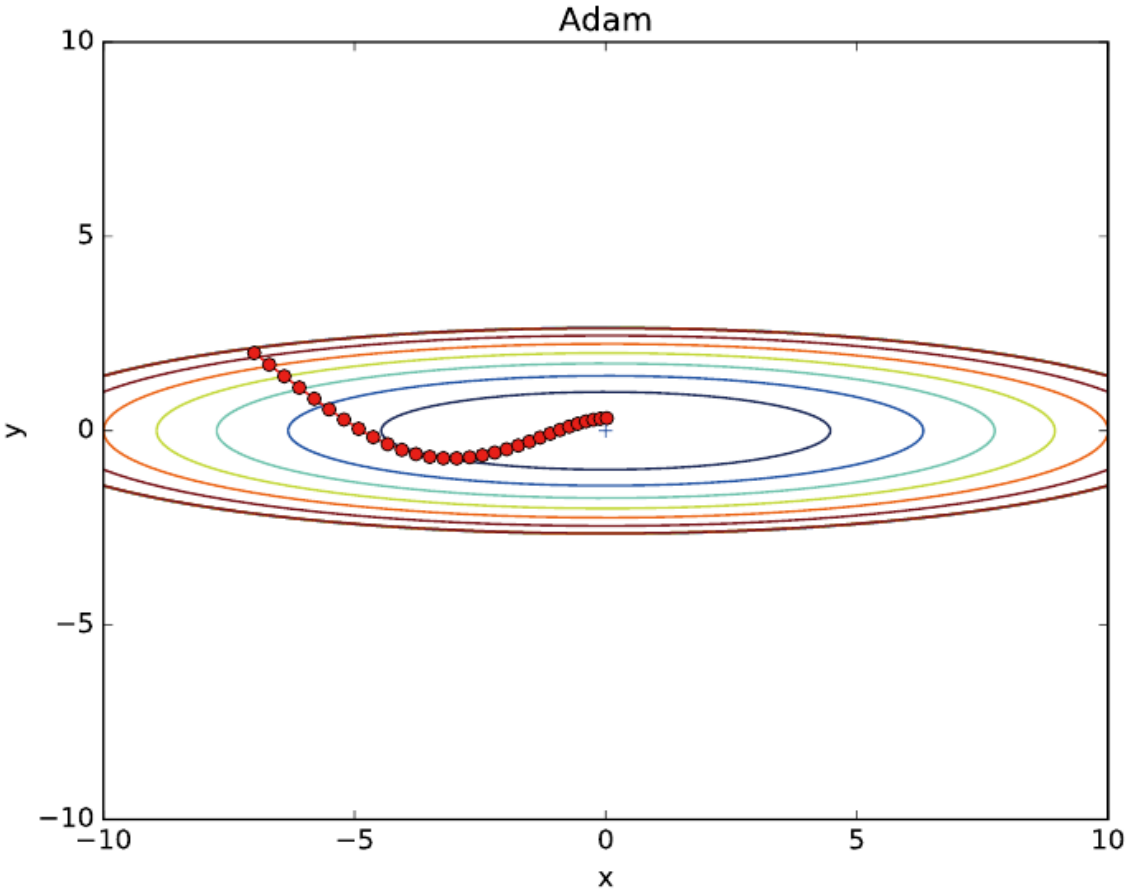
하나는 학습률, 나머지 두개는 일차 모멘텀용 계수 β_1과 이차 모멘텀용 계수 β_2임.

논문에 의하면, β_1은 0.9, β_2는 0.999값으로 놓을 때 좋은 결과를 얻을 수 있다고 함.

6.1.6 Adam (Adaptive Moment)

Adam 갱신 과정도 모멘텀처럼 그릇 바닥을 구르는 듯 한데, 모멘텀 때보다 공의 좌우 흔들림이 적다. 이는 학습 갱신 강도를 적응적으로 조정하기 때문.

그림 6-7 Adam에 의한 최적화 갱신 경로

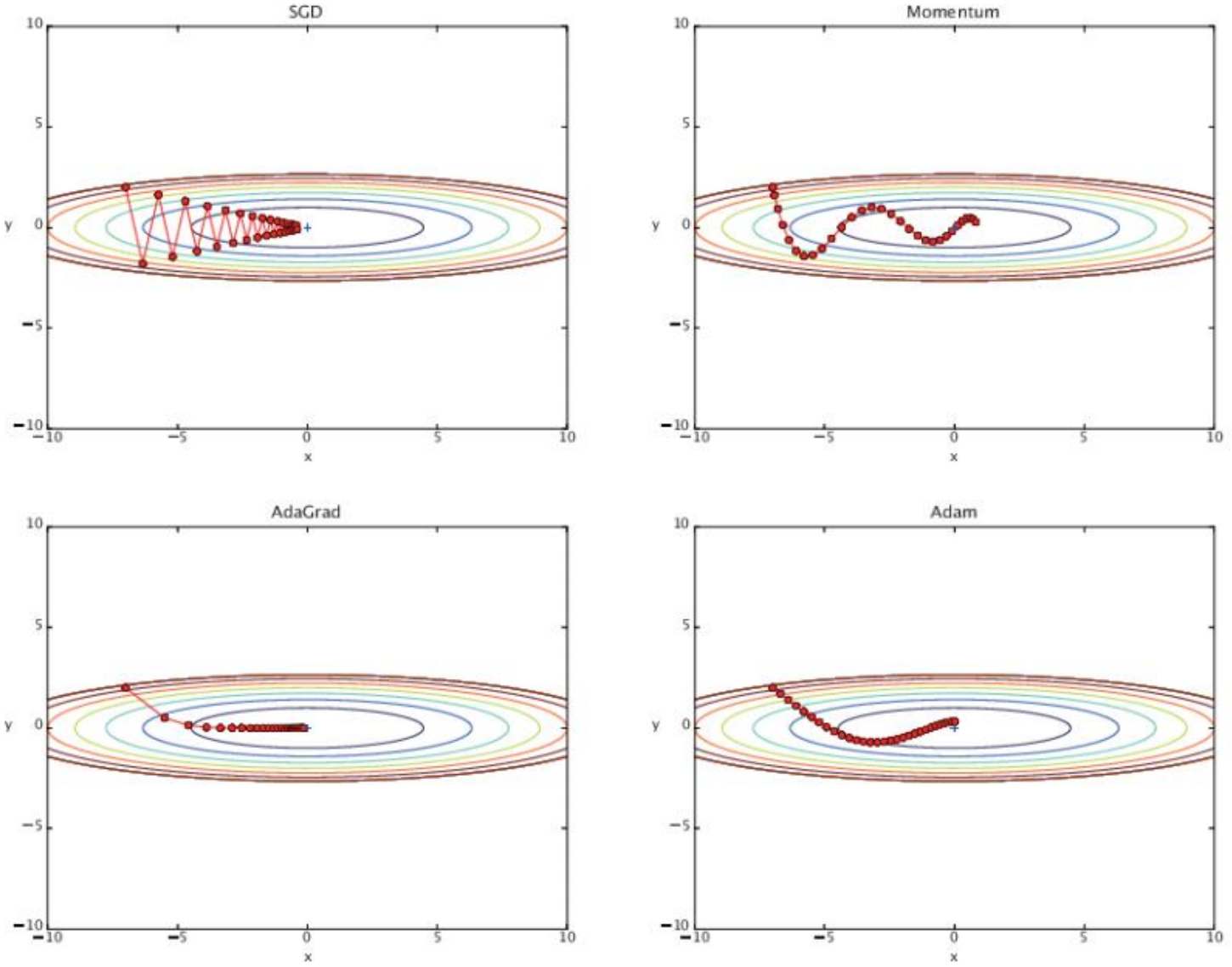


6.1.7 어느 갱신 방법을 이용할 것인가?

4 방법 탐색 과정 비교

모든 문제에 대해 항상 뛰어난 기법은 없음.
SGD도 많이 사용되고 있고, 모멘텀과 AdaGrad도 많이 사용되며, 최근에는 Adam이 많이 사용됨.
(코드는 ch06/optimizer_compare_naïve.py 참고)

그림 6-8 최적화 기법 비교 : SGD, 모멘텀, AdaGrad, Adam

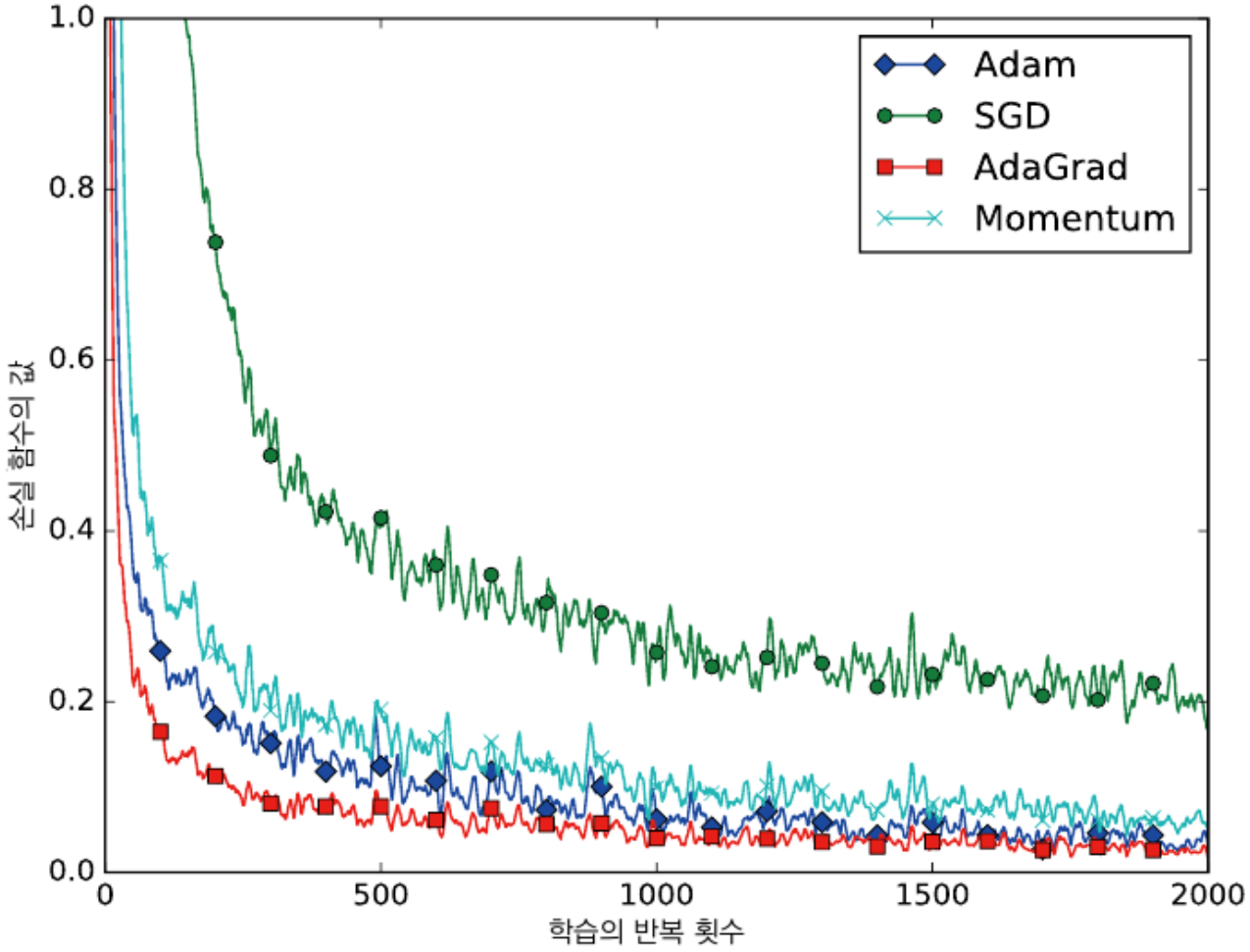


6.1.8 MNIST 데이터셋으로 본 갱신 방법 비교

MNIST 손글씨 숫자 인식을 대상으로 4 방법 탐색의 수렴 과정 비교
(코드는 ch06/optimizer_compare_mnist.py 참고)

각 층이 100개 뉴런으로 구성된 5층 신경망에서 ReLU 활성화 함수 사용.
학습진도는 SGD가 가장 느리고 AdaGrad가 가장 빠름. 보통, SGD보다 다른 세 기법이 빠르게 학습하고, 정확도도 살짝 높다.

그림 6-9 MNIST 데이터셋에 대한 학습 진도 비교



Common/optimizer.py 파일에 있는 매개변수 갱신 방법들

1. SGD
2. Momentum
3. Nesterov
4. AdaGrad
5. RMSprop
6. Adam

```
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[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

6.2 가중치의 초깃값

가중치의 초깃값을 무엇으로 설정하느냐가 신경망 학습의 성패를 가르는 일이 종종 있음.

권장 초깃값에 대해 설명하고, 실험을 통해 실제로 신경망 학습이 신속하게 이뤄지는 모습을 확인하고자 함.

다음 시간에~