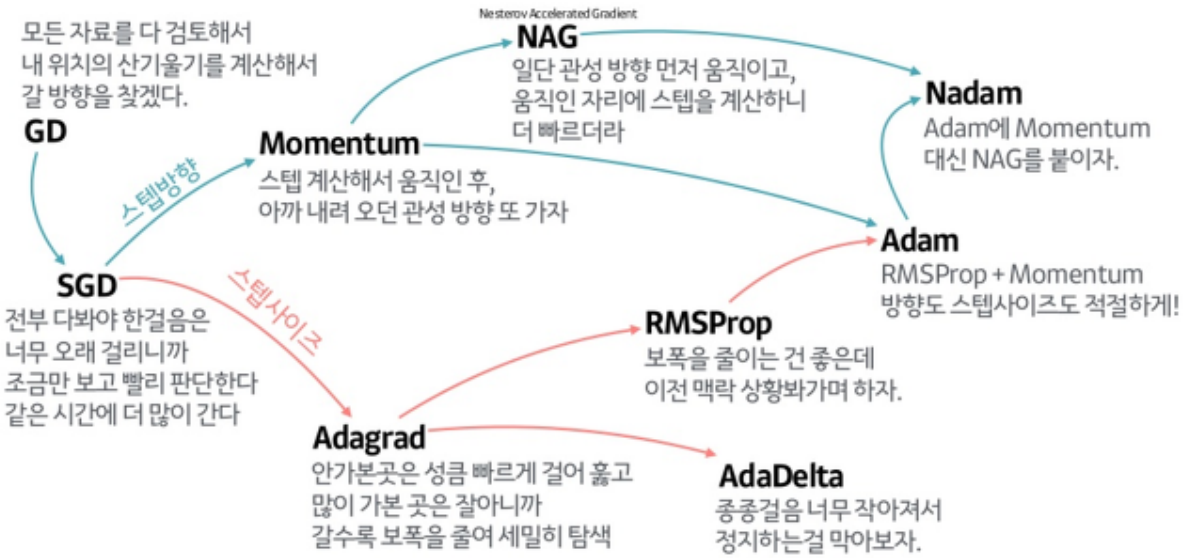
# 딥러닝)Optimizer 알고리즘

## 업데이트 알고리즘

너무 많은 것이 있고 굳이 안 짚고 넘어가도 되지만 그래도 수학과로서 수치최적화에 신경을 써야하므로 나중에 따로 정리하겠지만 몇가지 짚고 나가겠다.

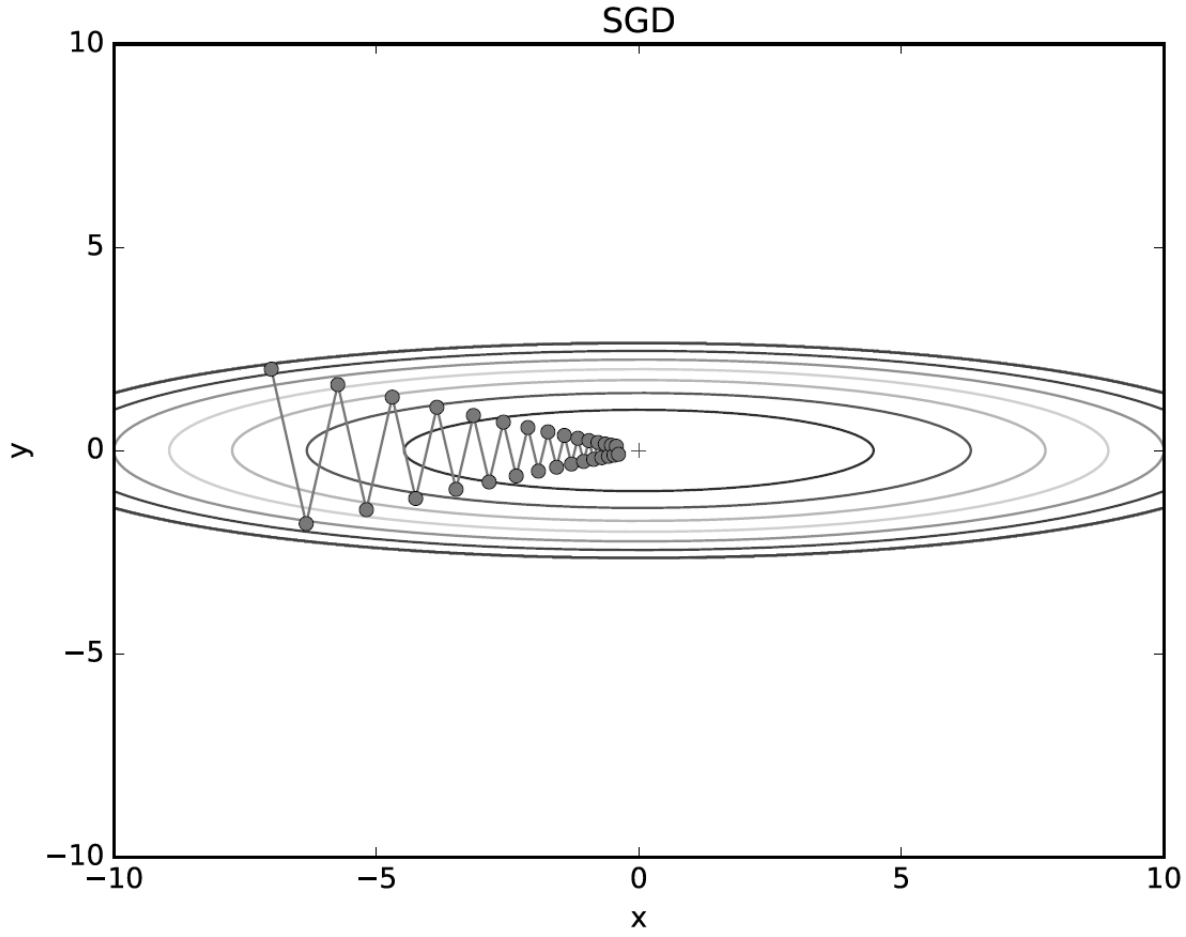



### SGD

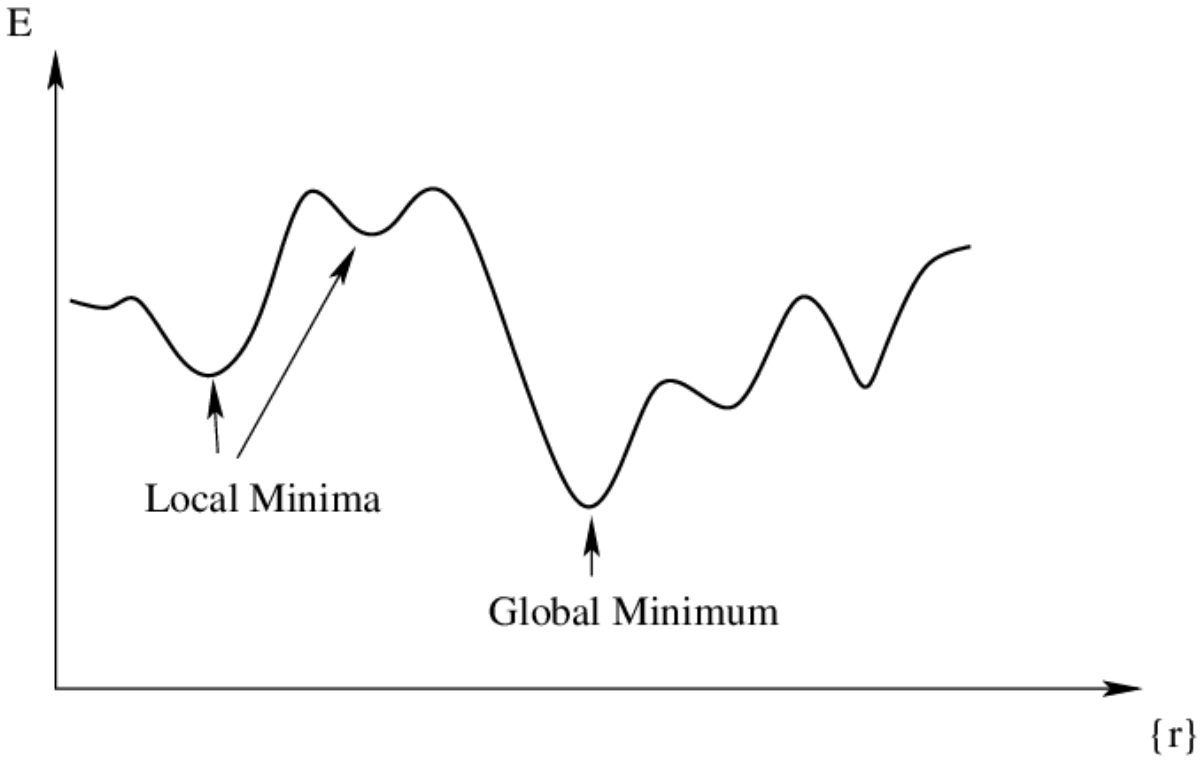
- 데이터 1개를 본후에 weight update를 진행한다. 그렇게 모든 데이터를 한번씩 수행
- 그 다음 데이터셋을 랜덤하게 섞어준다.
- 그리고 다시 1개씩 업데이트를 해준다.
- 이렇게 하나하나씩 보면서 업데이트 하는 방식이 답답해서 mini-batch -gd 라는 방법이 나왔다고 한다.
    - 데이터 B개를 본 후 업데이트한다.
    - 그리고 또 좋은 점은 병렬방식으로 진행되서 각 미니 배치간의 loss를 구하고 평균을 내서 update하기 때문에 좀 더 빠르고 잘 계산 할 수 있다.

하지만 이런 경사하강법은 고질적인 문제가 있다.

바로 saddle point와 기울기가 0 이되는 지점이 많을 때이다.




SGD는 위 그림처럼 심하게 굽이진 움직임을 보여준다. 상당히 비효율적인 모습을 보인다.



쉽게 말하면 다음 같은 상황에서 local minimum에 갇혀 버리면 빠져나오지 못하는것이 너무 큰문제다.

그래서 학습률을 가변적으로 적용해보자고 해서 Adagrad가 탄생한다.

<SGD 코드>

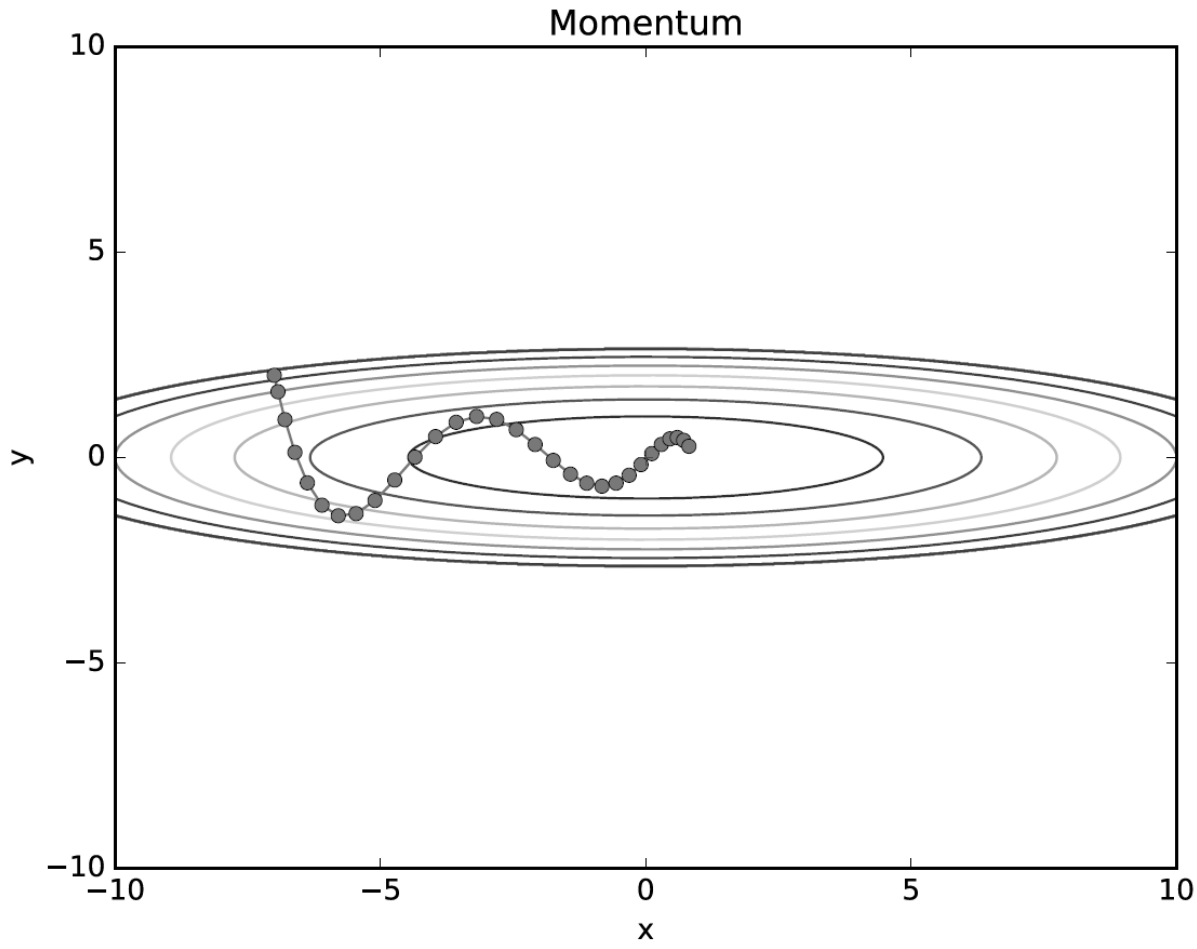
```
class SGD:
    def __init__(self, lr = 0.01):
        self.lr = lr

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

## momentum

단어 뜻대로 운동량. 즉 관성처럼 쭉 가는 것이다.

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

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




모멘텀은 공이 그릇 바닥을 구르듯 움직이는데 x축은 빠르게 전진해야하고 세로축은  천천히 진행되어야 한다.

지수 가중 평균이라는 것을 이용해서 스무스하게 움직이고  기울기의 가중 평균치를 산출해서 weight를 업데이트 하는 것이다.

Momentum을 사용하면서 속도가 빠르고 sgd에서 처럼 local minimum에서 안멈추고 탈출하는 것을 도와준다.

<momentum 코드>

```
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

- 여태까지 많이 가보지 않았던 방향 보다는 많이 갔던 방향에 대해 영향력을 줄이는 방식이다.

$$G_{t+1} = G_t + (\nabla L(w_t))^2$$

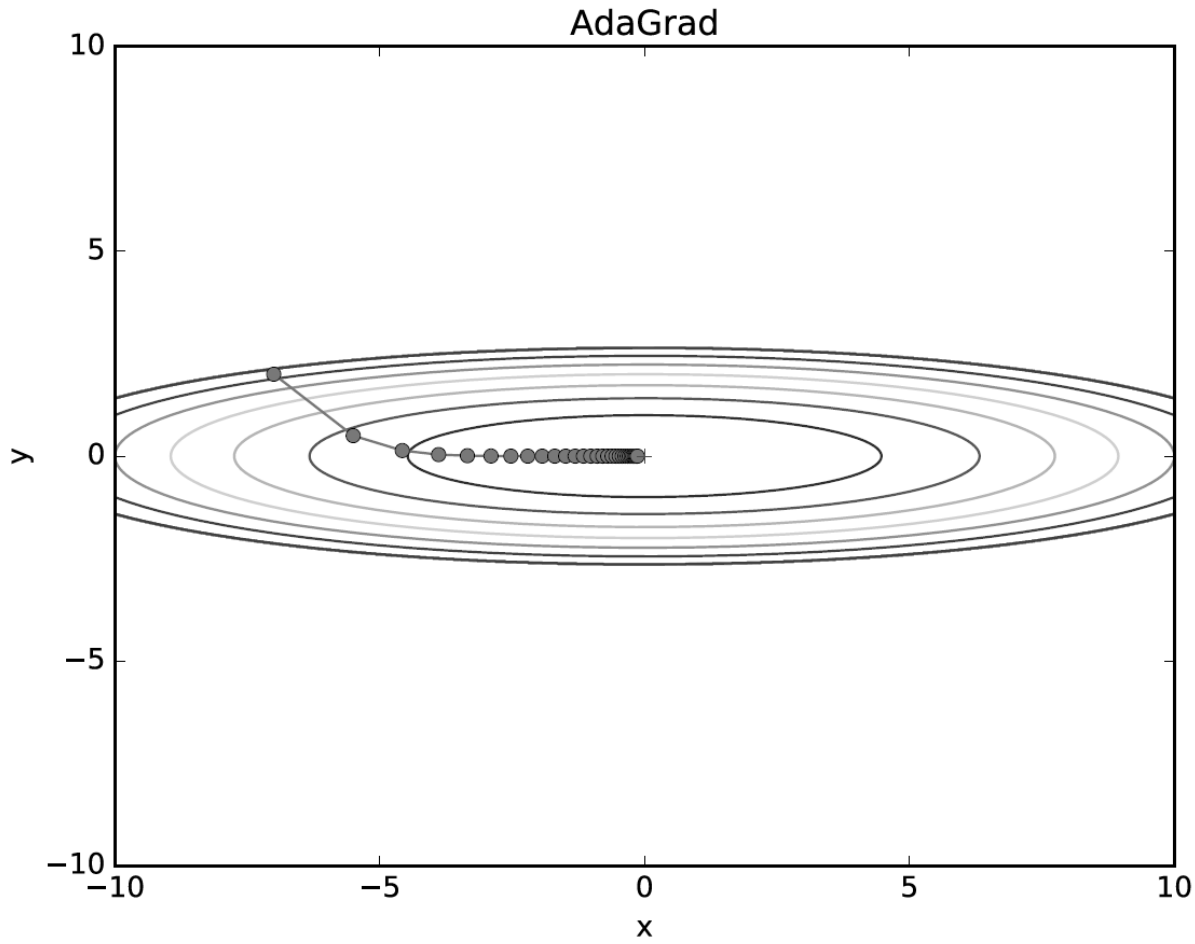
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + 1} + \epsilon} \nabla L(w_t)$$

우선 현재 위치에서 기울기를 구한다음 그것에 제곱한 값을 계속 누적하는 것이다.

그 다음 학습률을 누적한 것에 제곱근에 나눠줍니다.

이러허게 되면 누적값 grad_squared는 가파르던 방향에서 값이 클것이고 완만했던 방향에서는 값이 작을 것입니다.

그  다음 가파르던 방향인 큰값을 나눠줄 경우 상대적으로 더 작아질거고 완만한 방향은 상대적으로 학습률이 커져서 가파른 부분은 조금씩 내려가고 완만한 방향은 후딱 내려갈 것입니다.




위 그림을 보면 최솟값을 향해 효율적으로 움직이는 것을 볼 수 있다. 처음엔 y축이 크게 움직이지만 이렇게 크게 움직이면 adagrad에 의해 값이 작아지므로 쭉 작아지면서 y축의 움직임이 많이 작아집니다.

하지만 이렇게 되면 saddle point에 갇힐 경우 벗어날 수 없어서 RMSProp 가 탄생했습니다.

<adagrad 코드>

```
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

이전 adagrad에서 grad_squared에 가중치를 두는 방법으로 이렇게 되면 무작정 grad_squared가 커지는 것이 아니므로 이전 방식보다 개선이 될것으로 보입니다.

식을 쓰진 않겠지만 grad squared에 우변에 각각 베타와 1-베타를 곱해주는 것입니다.

이렇게 되면 아주 옛날 과거 기울기는 서서히 값의 중요도가 작아지고 새로운 기울기 정보를 크게 반영한다.

## Adam

이거 쓸려고 앞에거 쭉 썻다... 현재 그냥 디폴트로 써도 될정도로 가장 많이 쓰는 방법이다.

momentum + Adagrad 를 합친 느낌입니다.

adam은 적은 연상량을 지닌 스토캐스틱 optimizatio 알고리즘이다.

$$
\begin{aligned}
m_1 &\leftarrow \beta_1 m_0 + (1-\beta_1) g_1 \\
\widehat{m_1} &\leftarrow \frac{m_1}{1-\beta_1^1} = \frac{\beta_1 m_0}{1-\beta_1^1} + \frac{(1-\beta_1) g_1}{1-\beta_1^1} \\
&= 0 + g_1 (\because m_0 = 0)
\end{aligned}
$$

(m은 모멘템의 v, v= RMSProp에서 g(adagrad 참고), 모멘텀에 RMSProp 처럼 가중치를 두면서 업데이트를 시켜주고 있다

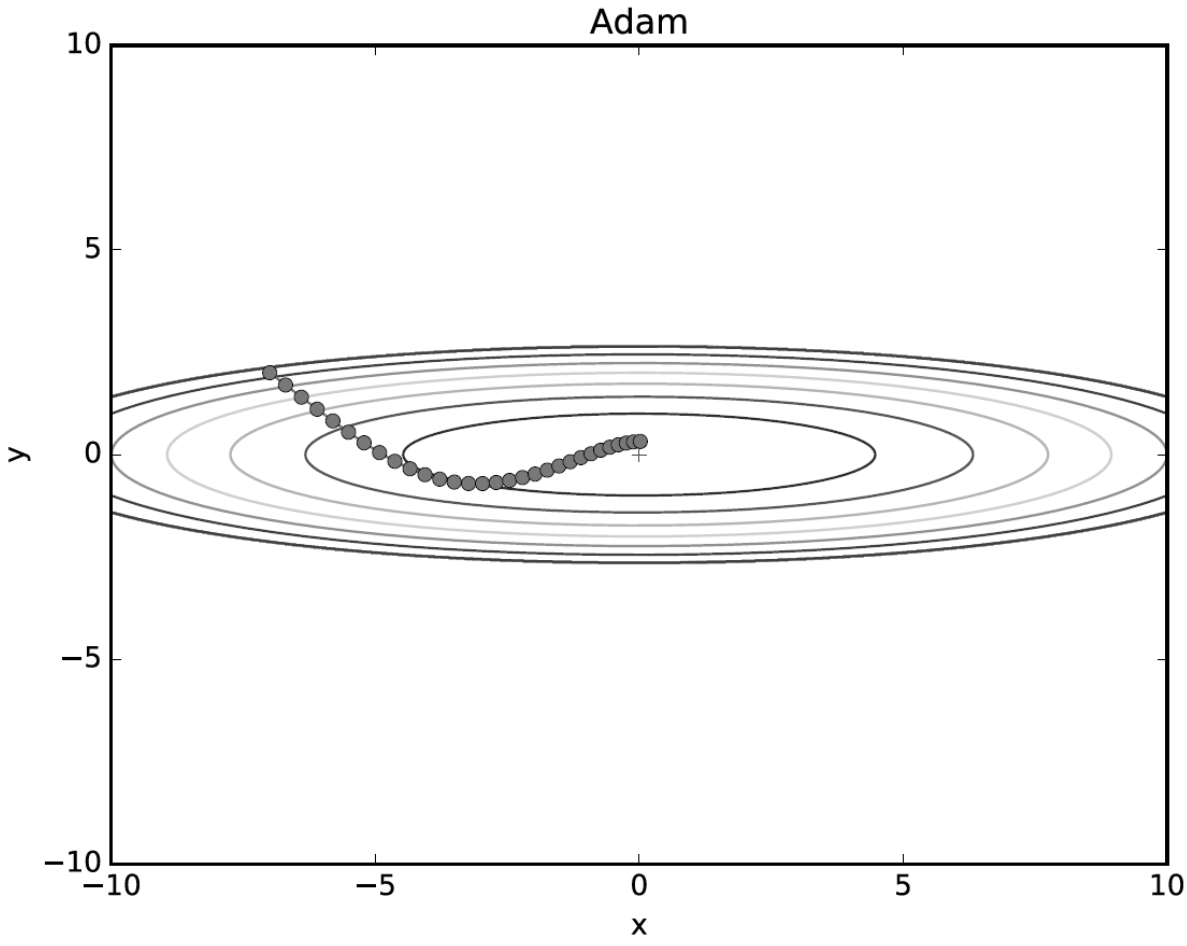
$$
\begin{aligned}
m_2 &\leftarrow \beta_1 m_1 + (1-\beta_1) g_2 \\
\widehat{m_2} &\leftarrow \frac{m_2}{1-\beta_1^2} = \frac{\beta_1 m_1}{1-\beta_1^2} + \frac{(1-\beta_1) g_2}{1-\beta_1^2} \\
&= 4.73 m_1 + 0.52 g_2 (\because \beta_1 = 0.9)
\end{aligned}
$$

다음과 같이 계속 모멘텀을 업데이트 해준다.

$$\theta_t \leftarrow \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t + \varepsilon}}$$

그렇게 RMSProp 방식처럼 v(g)에 루트를 씌어주고 위에 학습률에는 모멘텀 업데이트를 곱해줘서 계속 업데이트를 시켜주는 이방식을 아담이라고 한다.

모멘텀 같은 경우에는 정확도를 개선 해주고 RMSProp 같은 경우에는 보폭의 크기를 개선해주는데 아담은 이 둘을 쓰면서 정확도와 보폭 모두 개선해서 아주 효율적이라고 한다.




다음 처럼 학습되어서 saddle point 와 로컬 미니멈이 여러개인 경우를 모두 잘 해결 할 수 있을 것같다.

그래도 아직 많이 잘 이해되지 않기 때문에 다음에 아담만! 논문을 읽고 정리 하겠다.

<Adam 코드>

```
first_moment=0
second_moment=0

for t in range(m):
  current_gradient=evaluate_gradient(x)

  first_moment=beta1*first_moment + (1-beta1)*current_gradient
  second_moment=beta2*second_moment +(1-beta2)*current_gradient**2

  unbiased_first=first_moment / (1-beta1**t)
  unbiased_second=second_moment / (1-beta2**t)

  weight=weight-learning_rate * unbiased_first / (np.sqrt(unbiased_second)+1e-8)
```