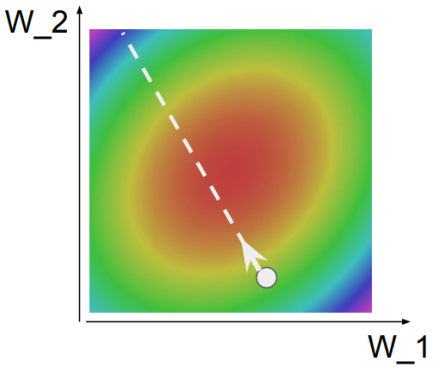
Lecture 7-1

Title	Training Neural Networks II
slide	http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture7.pdf

Fancier optimization

SGD

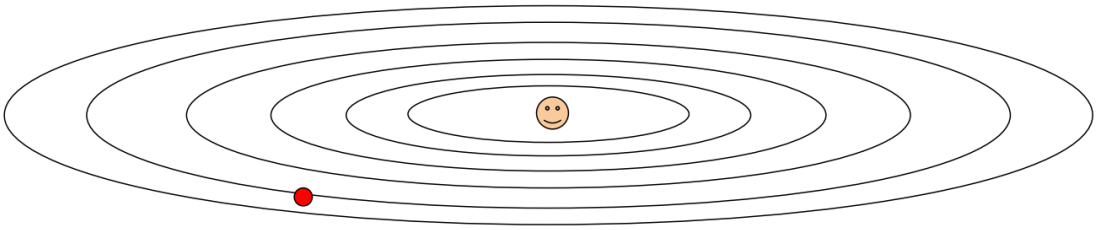
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Neural network에서 가장 중요한 것은 바로 최적화 문제였다. Nerwork의 가중치에 대해서 손실 함수를 정의하면, 이 손실 함수는 그 가중치가 얼마나 좋은지 나쁜지를 알려준다. 위의 오른쪽 사진에서 X/Y축은 두 개의 가중치를 의미하고, 각각의 색은 Loss의 값을 나타낸다. 이제 두 개의 가중치 W1과 W2를 최적화 시키는 문제라고 생각해보면, 우리의 목적은 가장 붉은색인 지점을 찾는 것, 즉 가장 낮은 Loss를 가진 가중치를 찾는 것이다. 지금까지 배운 것을 생각해 보았을 때, 가장 간단한 최적화 알고리즘은 Stochastic Gradient Descent이다. 우선 미니 배치 안의 데이터에서 Loss를 계산하고, 손실 함수를 내려가는 방향으로 해야 하기 때문에 'Gradient 의 반대 방향"을 이용해서 파라미터 벡터를 업데이트한다. 이 단계를 계속 반복하면 결국 Loss가 낮은 붉은색 지역으로 수렴할 것이다. 하지만 이 단순한 알고리즘을 실제로 사용하게 되면 몇 가지 문제가 나타난다.

문제점 1. 지그재그 문제

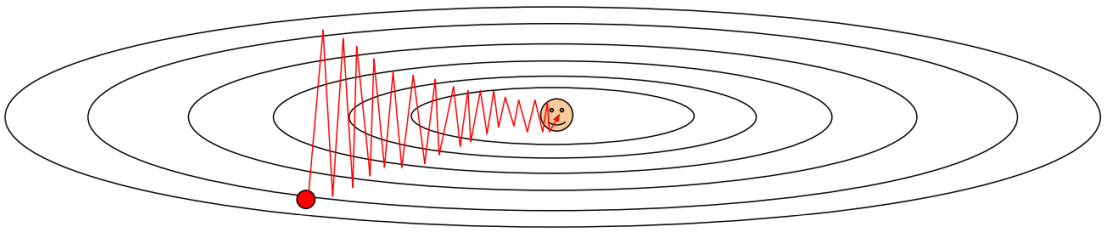
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

SGD의 문제점 중 하나는 손실 함수가 위의 그림과 같이 생겼다고 생각했을 때 나타난다. 똑같이 W1과 W2가 있다고 해보면, 둘 중 어떤 하나는 업데이트를 해도 손실 함수가 아주 느리게 변한다. 즉, 수평 축의 가중치는 변해도 Loss가 아주 천천히 줄어들고, Loss는 수직 방향의 가중치 변화에 훨씬 더 민감하게 반응한다. 다시 말해서 현재 지점에서 **Loss는 bad condition number를 지니고 있다**고 할 수 있다. 이것은 현재 지점의 Hessian maxrix의 최대/최소 singular values 값의 비율이 매우 안 좋다는 뜻이다.

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

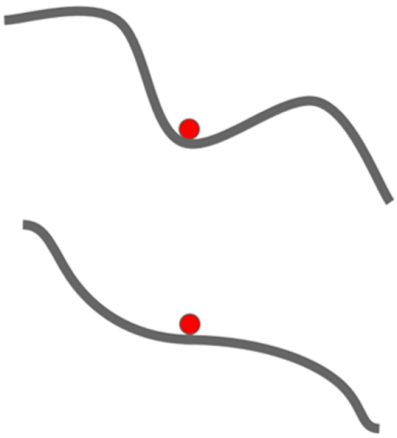


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

이를 좀 더 직관적으로 이해해보면, Loss가 taco shell 같은 모양으로 있어서 한 방향으로는 매우 민감하지만 다른 방향으로는 덜 민감한 상태에 있는 것이다. 이 상황에는 gradient의 방향이 고르지 못하기 때문에 Gradient를 계산하고 업데이트 하게 되면 line을 넘나들면서 왔다 갔다 하게 되면서 SGD으로 학습한다. 그래서 SGD를 수행하면 Loss에 영향을 덜 주는 수평 방향 차원의 가중치는 업데이트가 아주 느리게 진행되고, 빠르게 변하는 수직 차원을 가로지르면서 지그지그로 지저분하게(nasty) 움직이게 된다.

더 큰 문제는 이것이 고차원 공간에서 훨씬 더 빈번하게 발생한다는 것이다. 위의 예는 1차원이지만 실제로는 가중치가 수천 수억 개 존재하는 네트워크를 다루게 되는데, 이 경우에 수억 개의 방향으로 움직일 수 있다. 만약 수억 개의 파라미터가 있을 때, 불균형의 발생 비율은 상당히 높으며, 방향 중에 불균형한 방향이 존재한다면 SGD는 잘 동작하지 않는다.

문제점 2. local minima와 saddle points



SGD에서 발생하는 또 다른 문제는 local minima 와 saddle points와 관련된 문제이다. X 축은 어떤 하나의 가중치를 나타내고 Y축은 Loss를 나타낸다고 하자. **[local minima]** 위의 그림과 같이 휘어진 손실 함수는 중간에 "valley"가 하나 있다고 할 때 SGD는 어떻게 움직일까? 이 경우 locally falt하여 gradient가 0이 되기 때문에 SGD는 멈춰버린다. SGD는 gradient를 계산하고 그 반대 방향으로 이동하는데, “valley” 구간에서 "opposite gradient" 도 0이 되어 학습이 멈춰버린다. **[saddle points]** 아래 그림과 같이 local minima는 아니지만 한쪽 방향으로는 증가하고 있고 다른 한쪽 방향으로는 감소하고 있는 지역을 생각해 볼 수 있다. 이런 saddle point에서도 gradient=0이므로 멈추게 된다. 또한 saddle point 뿐만 아니라, saddle point의 근처에서도 문제가 발생한다. saddle point 근처에서 gradient가 0은 아니지만 기울기가 아주 작은데, 이것은 gradient를 계산해서 업데이트를 해더라도 기울기가 아주 작아서 가중치의 업데이트가 아주 느리게 되기 때문에 문제가 된다.

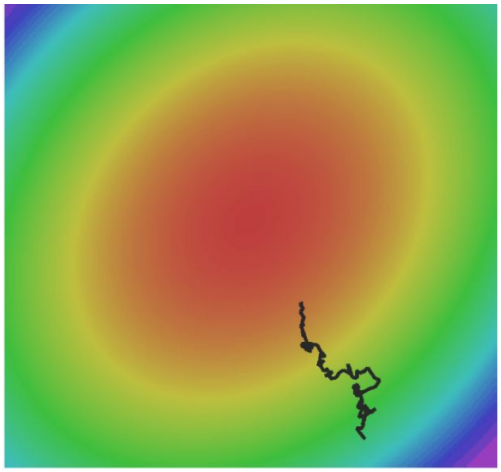
비록 이와 같이 1차원의 예제에서는 local minima가 매우 심각하고 saddle point는 덜 심각해 보이지만, 고차원 공간에서는 그 반대이다. 만약 1억 차원의 공간이 있다고 하면, Saddle point는 어떤 방향은 Loss가 증가하고 몇몇 방향은 Loss가 감소하고 있는 곳이라는 말이고, Local minima는 그 방향이 전부 Loss가 상승하는 방향이라는 것이다. 1억 개의 차원에서 Saddle point은 사실상 거의 모든 곳에서 발생하고, Local minima는 매우 드물게 발생한다. very large neural network가 local minima 보다는 saddle point에 취약하다는 것은 실제로 지난 몇 년 동안 밝혀진 사실이다.

문제점 3. gradient의 추정 값 사용

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



SGD는 손실 함수를 계산할 때 Traing set 각각의 loss를 전부 계산해야 한다. 전체 데이터로 Loss를 계산하는 것은 어렵기 때문에, 실제로는 미니 배치의 데이터들을 가지고 Loss를 추정하여 사용한다. 이렇게 되면 정확한 gradient를 얻을 수 없으며, 부정확한 추정 값(noisy estimate) 만을 구할 수 있다 오른쪽에 보이는 그림은 각 지점의 gradient에 random uniform noise를 추가하고 SGD를 수행하게 만든 것이다. 손실함수 공간을 위와 같이 지그재그로 돌아다니게 되면 minima까지 도달하는데 시간이 더 오래 걸린다.

SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

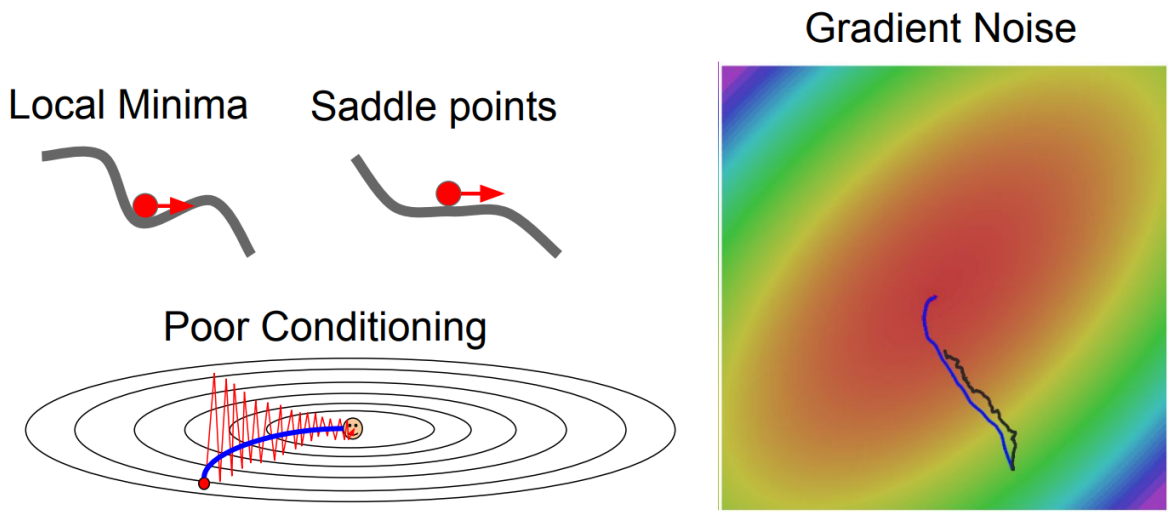
SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

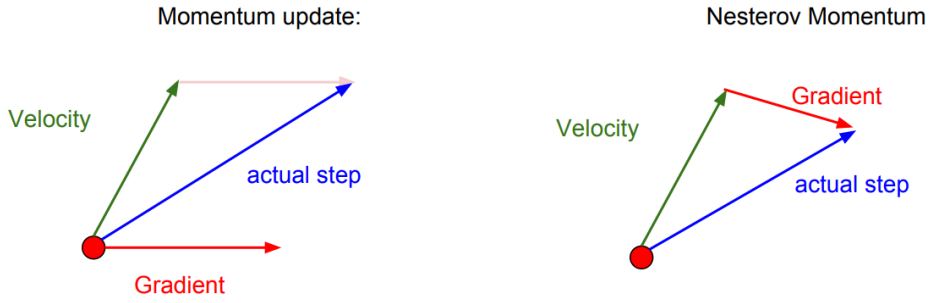
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

SGD의 문제점들의 대다수를 해결할 수 있는 아주 간단한 방법은 SGD에 momentum term을 추가하는 것이다. 왼쪽은 classic 버전의 SGD이고, 오른쪽은 SGD+momentum로 gradient를 계산할 때 velocity를 이용하는 것으로, momemtum의 비율을 나타내는 하이퍼파라미터 rho를 추가하였다. 일반적으로 velocity의 영향력을 rho의 비율로 맞춰주는데 보통 0.9와 같이 높은 값으로 맞춰주며, velocity에 일정 비율 rho를 곱해주고 현재 gradient를 더하여 gradient vector 그대로의 방향이 아닌 velocity vector의 방향으로 나아가게 된다. 이는 매우 간단한 방법이지만 지금까지 말했던 문제들을 해결하는데 많은 도움을 준다.



local minima와 saddle points문제를 생각해보면, 물리적으로 공이 굴러 내려오는 것을 상상할 수 있다. 공은 내려갈수록 빨라지기 때문에(velocity가 커짐), local minima에 도달해도 여전히 velocity를 가지고 있기 때문에 gradient=0이라도 움직일 수 있어서 앞선 local minima 문제를 극복하여 계속해서 내려갈 수 있다. saddle point에서도 주변의 gradient가 작더라도 굴러 내려오는 속도가 있기 때문에 velocity를 가지게 되고, saddle point 문제를 해결 할 수 있다.

업데이트가 잘 안되는 경우(poor conditioning)를 다시 한번 살펴보자. 아래와 같이 지그재그로 움직이는 상황이라면 momentum이 이 변동을 상쇄시켜 버려서 loss에 만감한 수직 방향의 변동은 줄여주고 수평 방향의 움직임은 점차 가속화시킨다. 즉, momentum을 추가하게 되면 high condition number problem을 해결하는 데 도움이 된다. 오른쪽 그림을 보면 검은색이 일반 SGD이고, 파란색이 Momentum SGD이다. Momentum을 추가해서 velocity가 생기면 결국 noise가 평균화된다. 검은색 SGD가 지그재그로 움직이는 것에 비해서 파란색 Momentum SGD는 minima를 향해서 더 부드럽게 움직이는 것을 확인할 수 있다.



빨간 점이 현재 지점이고, Red Vector는 현재 지점에서의 gradient의 방향, Green vector는 Velocity vector을 나타낸다. 실제 업데이트는(autual step) 이 두가지의 가중 평균으로 구하며, 이는 gradient의 noise를 극복하게 해준다.

Nesterov Momentum

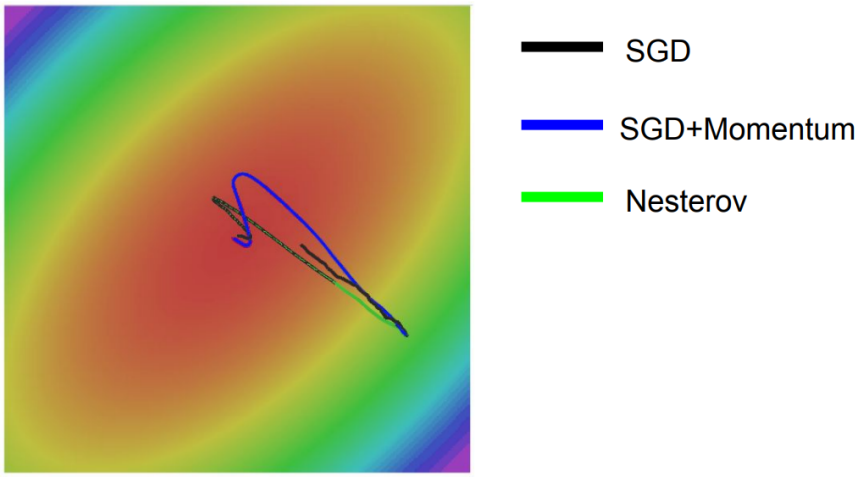
오른쪽은 Momentum을 변형한 것으로, Nesterov accelerated gradient 혹은 Nesterov momentum 라고 부른다. 기본 SGD momentum은 "현재 지점"에서 gradient를 계산한 후, velocity와 섞어 주는데, Nesterov는 현재 지점에서 Velocity방향으로 움직이고 그 지점에서 gradient를 계산하고 다시 원점으로 돌아가서 둘을 합친다. 정확하진 않지만, 두 정보를 약간 더 섞어준다고 생각할 수 있다. 이는 velocity의 방향이 잘못되었을 경우, 현재 gradient의 방향을 좀 더 활용할 수 있도록 해준다. Nesterov는 Convex optimization 문제에서는 뛰어난 성능을 보이지만, Neural network와 같은 non-convex problem 에서는 성능이 보장되지 않는다.

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$
\begin{aligned}
v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\
\tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\
&= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)
\end{aligned}
$$

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

Nesterov의 공식을 보면 다소 까다롭게 생겼다. 기존에는 Loss와 Gradient를 같은 점(xt)에서 구했다. Nesterov에서도 마찬가지로 공식을 조금 변형하여 마찬가지로 Loss와 Gradient를 같은 점에서 계산할 수 있다. 첫 번째 수식은 기존의 momentum과 동일하게 velocity와 계산한 gradient를 일정 비율로 섞어주는 역할을 한다. 맨 밑의 수식을 보면, 현재 점과 velocity를 더하고,"현재 velocity-이전 velocity"를 계산해서 일정 비율(rho)을 곱하고 더해준다. 즉, Nesterov momentum는 현재/이전의 velocity간의 에러보정(error-correcting term)이 추가된 것이다.



SGD, Momentum, Nesterov의 예시를 한번 살펴봅시다. 기본 SGD(검정색)은 아주 천천히 내려가고, momentum(파란색)과 Nesterov(초록색)은 이전의 velocity의 영향을 받기 때

문에 minima를 그냥 지나쳐 버리는 경향이 있다. 그러나 최종적으로는 스스로 경로를 수정하고는 결국 minima에 수렴한다. 일반 momentum와 Nesterov이 조금 다르게 움직이는 것도 확인할 수 있는데, 이는 Nesterov에서 추가된 수식 때문에 일반 momentum에 비해서 overshooting이 덜 하다.