Gradient Descent Optimization Algorithms 정리

MAY 20TH, 2016

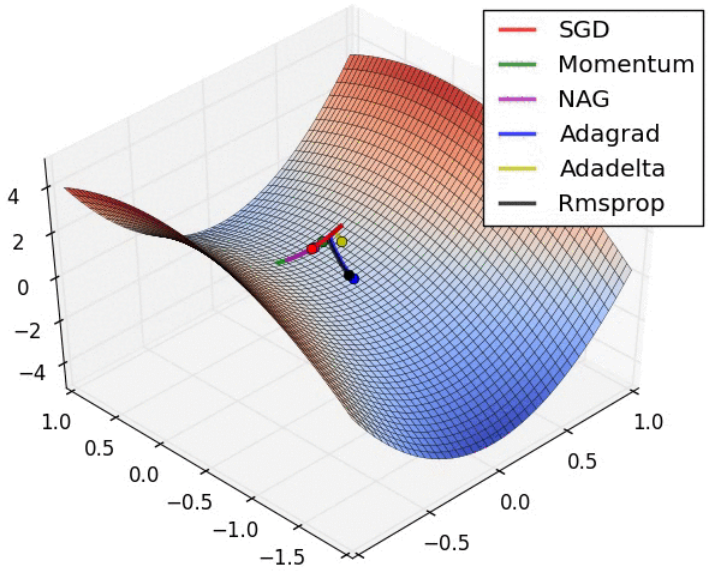
Neural network의 weight을 조절하는 과정에는 보통 **'Gradient Descent'** 라는 방법을 사용한다. 이는 네트워크의 parameter들을 θ라고 했을 때, 네트워크에서 내놓는 결과값과 실제 결과값 사이의 차이를 정의하는 함수 Loss function $J(\theta)$의 값을 최소화하기 위해 기울기(gradient) $\nabla_\theta J(\theta)$를 이용하는 방법이다. Gradient Descent에서는 θ 에 대해 gradient의 반대 방향으로 일정 크기만큼 이동해내는 것을 반복하여 Loss function $J(\theta)$ 의 값을 최소화하는 θ 의 값을 찾는다. 한 iteration에서의 변화 식은 다음과 같다.

$$\theta = \theta - \eta \nabla_\theta J(\theta)$$

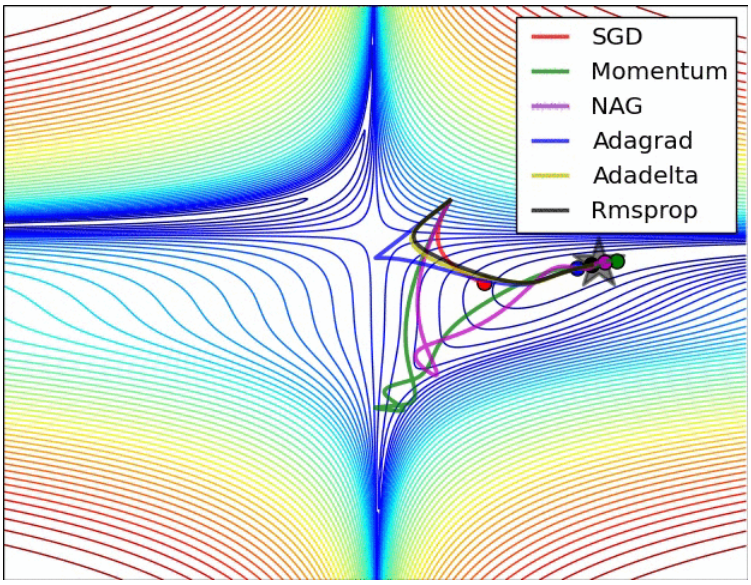
이 때 η 는 미리 정해진 걸음의 크기 'step size' 로서, 보통 0.01~0.001 정도의 적당한 크기를 사용한다.

이 때 Loss Function을 계산할 때 전체 train set을 사용하는 것을 **Batch** Gradient Descent 라고 한다. 그러나 이렇게 계산을 할 경우 한번 step을 내딛을 때 전체 데이터에 대해 Loss Function을 계산해야 하므로 너무 많은 계산량이 필요하다. 이를 방지하기 위해 보통은 **Stochastic** Gradient Descent (SGD) 라는 방법을 사용한다. 이 방법에서는 loss function을 계산할 때 전체 데이터(batch) 대신 일부 조그마한 데이터의 모음(mini-batch)에 대해서만 loss function을 계산한다. 이 방법은 batch gradient descent 보다 다소 부정확할 수는 있지만, 훨씬 계산 속도가 빠르기 때문에 같은 시간에 더 많은 step을 갈 수 있으며 여러 번 반복할 경우 보통 batch의 결과와 유사한 결과로 수렴한다. 또한, SGD를 사용할 경우 Batch Gradient Descent에서 빠질 local minima에 빠지지 않고 더 좋은 방향으로 수렴할 가능성도 있다.

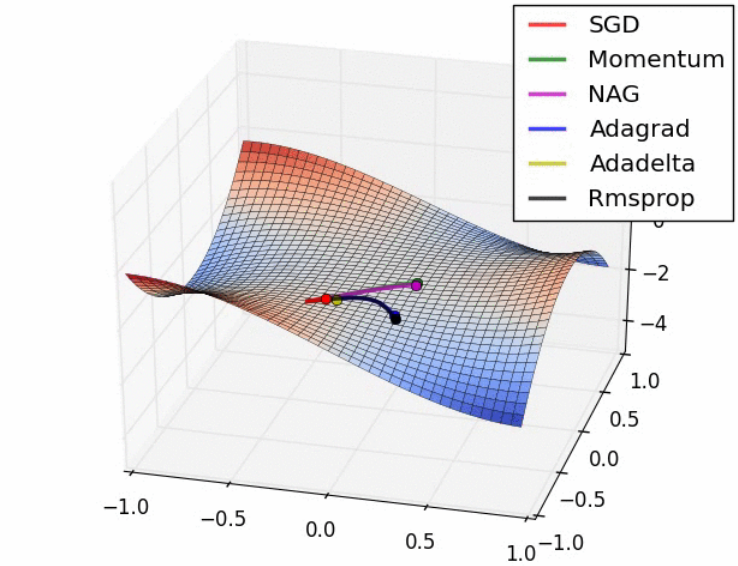
보통 Neural Network를 트레이닝할 때는 이 SGD를 이용한다. 그러나 단순한 SGD를 이용하여 네트워크를 학습시키는 것에는 한계가 있다. 결론부터 살펴보기 위해, 다음과 같은 그림들을 살펴보자.



Gradient Descent Optimization Algorithms at Long Valley



Gradient Descent Optimization Algorithms at Beale's Function



Gradient Descent Optimization Algorithms at Saddle Point

위의 그림들은 각각 SGD 및 SGD의 변형 알고리즘들이 최적값을 찾는 과정을 시각화한 것이다. 빨간색의 SGD가 우리가 알고 있는 Naive Stochastic Gradient Descent 알고리즘이고, Momentum, NAG, Adagrad, AdaDelta, RMSprop 등은 SGD의 변형이다. 보다시피 모든 경우에서 SGD는 다른 알고리즘들에 비해 성능이 월등하게 낮다. 다른 알고리즘들 보다 이동속도가 현저하게 느릴뿐만 아니라, 방향을 제대로 잡지 못하고 이상한 곳에서 수렴하여 이동하지 못하는 모습도 관찰할 수 있다. 즉 단순한 SGD를 이용하여 네트워크를 학습시킬 경우 네트워크가 상대적으로 좋은 결과를 얻지 못할 것이라고 예측할 수 있다. 그렇다면 실제로는 어떤 방법들을 이용해야 하는 것인가? 이 글에서는 Neural Network를 학습시킬 때 실제로 많이 사용하는 다양한 SGD의 변형 알고리즘들을 간략하게 살펴보겠다. 내용과 그림의 상당 부분은 Sebastian Ruder의 글 (http://sebastianruder.com/optimizing-gradient-descent/) 에서 차용했다.

Momentum

Momentum 방식은 말 그대로 Gradient Descent를 통해 이동하는 과정에 일종의 '관성'을 주는 것이다. 현재 Gradient를 통해 이동하는 방향과는 별개로, 과거에 이동했던 방식을 기억하면서 그 방향으로 일정 정도를 추가적으로 이동하는 방식이다. 수식으로 표현하면 다음과 같다. v_t를 time step t에서의 이동 벡터라고 할 때, 다음과 같은 식으로 이동을 표현할 수 있다.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

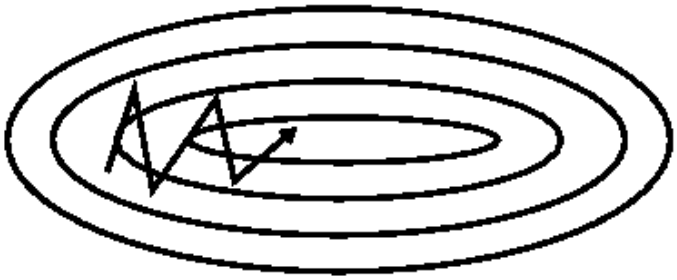
$$\theta = \theta - v_t$$

이 때, γ는 얼마나 momentum을 줄 것인지에 대한 momentum term으로서, 보통 0.9 정도의 값을 사용한다. 식을 살펴보면 과거에 얼마나 이동했는지에 대한 이동 항 v를 기억하고, 새로운 이동항을 구할 경우 과거에 이동했던 정도에 관성항만큼 곱해준 후 Gradient을 이용한 이동 step 항을 더해준다. 이렇게 할 경우 이동항 v_t 는 다음과 같은 방식으로 정리할 수 있어, Gradient들의 지수평균을 이용하여 이동한다고도 해석할 수 있다.

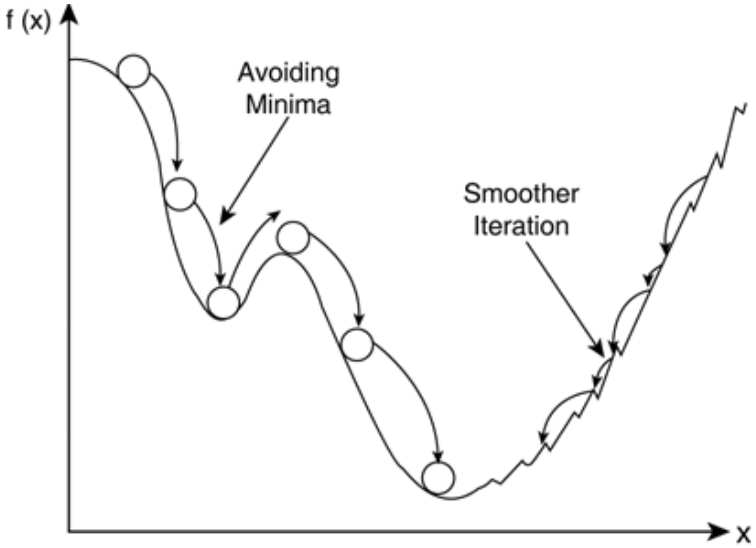
$$v_t = \eta\nabla_\theta J(\theta)_t + \gamma\eta\nabla_\theta J(\theta)_{t-1} + \gamma^2\eta\nabla_\theta J(\theta)_{t-2} + \ldots.$$

Momentum 방식은 SGD가 Oscilation 현상을 겪을 때 더욱 빛을 발한다. 다음과 같이 SGD가 Oscilation을 겪고 있는 상황을 살펴보자.

현재 SGD는 중앙의 최적점으로 이동해야하는 상황인데, 한번의 step에서 움직일 수 있는 step size는 한계가 있으므로 이러한 oscilation 현상이 일어날 때는 좌우로 계속 진동하면서 이동에 난항을 겪게 된다.



그러나 Momentum 방식을 사용할 경우 다음과 같이 자주 이동하는 방향에 관성이 걸리게 되고, 진동을 하더라도 중앙으로 가는 방향에 힘을 얻기 때문에 SGD에 비해 상대적으로 빠르게 이동할 수 있다.

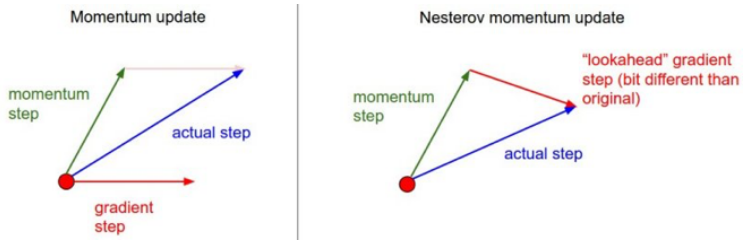


Avoiding Local Minima. Picture from http://www.yaldex.com.

또한 Momentum 방식을 이용할 경우 위의 그림과 같이 local minima를 빠져나오는 효과가 있을 것이라고도 기대할 수 있다. 기존의 SGD를 이용할 경우 좌측의 local minima에 빠지면 gradient가 0이 되어 이동할 수가 없지만, momentum 방식의 경우 기존에 이동했던 방향에 관성이 있어 이 local minima를 빠져나오고 더 좋은 minima로 이동할 것을 기대할 수 있게 된다. 반면 momentum 방식을 이용할 경우 기존의 변수들 θ 외에도 과거에 이동했던 양을 변수별로 저장해야하므로 변수에 대한 메모리가 기존의 두 배로 필요하게 된다.

Nesterov Accelerated Gradient (NAG)

Nesterov Accelerated Gradient(NAG)는 Momentum 방식을 기초로 한 방식이지만, Gradient를 계산하는 방식이 살짝 다르다. 빠른 이해를 위해 다음 그림을 먼저 살펴보자.



Difference between Momentum and NAG. Picture from CS231.

Momentum 방식에서는 이동 벡터 v_t 를 계산할 때 현재 위치에서의 gradient와 momentum step을 독립적으로 계산하고 합친다. 반면, NAG에서는 momentum step을 먼저 고려하여, momentum step을 먼저 이동했다고 생각한 후 그 자리에서의 gradient를 구해서 gradient step을 이동한다. 이를 수식으로 나타내면 다음과 같다.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

NAG를 이용할 경우 Momentum 방식에 비해 보다 효과적으로 이동할 수 있다. Momentum 방식의 경우 멈춰야 할 시점에서도 관성에 의해 훨씬 멀리 갈수도 있다는 단점이 존재하는 반면, NAG 방식의 경우 일단 모멘텀으로 이동을 반정도 한 후 어떤 방식으로 이동해야할 지를 결정한다. 따라서 Momentum 방식의 빠른 이동에 대한 이점은 누리면서도, 멈춰야 할 적절한 시점에서 제동을 거는 데에 훨씬 용이하다고 생각할 수 있을 것이다.

Adagrad

Adagrad(Adaptive Gradient)는 변수들을 update할 때 각각의 변수마다 step size를 다르게 설정해서 이동하는 방식이다. 이 알고리즘의 기본적인 아이디어는 '지금까지 많이 변화하지 않은 변수들은 step size를 크게 하고, 지금까지 많이 변화했던 변수들은 step size를 작게 하자' 라는 것이다. 자주 등장하거나 변화를 많이 한 변수들의 경우 optimum에 가까이 있을 확률이 높기 때문에 작은 크기로 이동하면서 세밀한 값을 조정하고, 적게 변화한 변수들은 optimum 값에 도달하기 위해서는 많이 이동해야할 확률이 높기 때문에 먼저 빠르게 loss 값을 줄이는 방향으로 이동하려는 방식이라고 생각할 수 있겠다. 특히 word2vec이나 GloVe 같이 word representation을 학습시킬 경우 단어의 등장 확률에 따라 variable의 사용 비율이 확연하게 차이나기 때문에 Adagrad와 같은 학습 방식을 이용하면 훨씬 더 좋은 성능을 거둘 수 있을 것이다.

Adagrad의 한 스텝을 수식화하여 나타내면 다음과 같다.

$$G_t = G_{t-1} + (\nabla_\theta J(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot \nabla_\theta J(\theta_t)$$

Neural Network의 parameter가 k개라고 할 때, G_t는 k차원 벡터로서 'time step t까지 각 변수가 이동한 gradient의 sum of squares' 를 저장한다. θ를 업데이트하는 상황에서는 기존 step size η에 G_t의 루트값에 반비례한 크기로 이동을 진행하여, 지금까지 많이 변화한 변수일 수록 적게 이동하고 적게 변화한 변수일 수록 많이 이동하도록 한다. 이 때 ϵ은 10^{-4} ~ 10^{-8} 정도의 작은 값으로서 0으로 나누는 것을 방지하기 위한 작은 값이다. 여기에서 G_t를 업데이트하는 식에서 제곱은 element-wise 제곱을 의미하며, θ를 업데이트하는 식에서도 $\cdot$ 은 element-wise한 연산을 의미한다.

Adagrad를 사용하면 학습을 진행하면서 굳이 step size decay 등을 신경써주지 않아도 된다는 장점이 있다. 보통 adagrad에서 step size로는 0.01 정도를 사용한 뒤, 그 이후로는 바꾸지 않는다. 반면, Adagrad에는 학습을 계속 진행하면 step size가 너무 줄어든다는 문제점이 있다. G에는 계속 제곱한 값을 넣어주기 때문에 G의 값들은 계속해서 증가하기 때문에, 학습이 오래 진행될 경우 step size가 너무 작아져서 결국 거의 움직이지 않게 된다. 이를 보완하여 고친 알고리즘이 RMSProp과 AdaDelta이다.

RMSProp

RMSProp은 딥러닝의 대가 제프리 힌톤이 제안한 방법으로서, Adagrad의 단점을 해결하기 위한 방법이다. Adagrad의 식에서 gradient의 제곱값을 더해나가면서 구한 G_t 부분을 합이 아니라 지수평균으로 바꾸어서 대체한 방법이다. 이렇게 대체를 할 경우 Adagrad처럼 G_t가 무한정 커지지는 않으면서 최근 변화량의 변수간 상대적인 크기 차이는 유지할 수 있다. 식으로 나타내면 다음과 같다.

$$G = \gamma G + (1 - \gamma)(\nabla_\theta J(\theta_t))^2$$

$$\theta = \theta - \frac{\eta}{\sqrt{G + \epsilon}} \cdot \nabla_\theta J(\theta_t)$$

AdaDelta

AdaDelta (Adaptive Delta) 는 RMSProp과 유사하게 AdaGrad의 단점을 보완하기 위해 제안된 방법이다. AdaDelta는 RMSProp과 동일하게 G를 구할 때 합을 구하는 대신 지수평균을 구한다. 다만, 여기에서는 step size를 단순하게 η 로 사용하는 대신 step size의 변화값의 제곱을 가지고 지수평균 값을 사용한다.

(/)

BEOMSU KIM'S BLOG (/)

ABOUT ME (/myself)

Beomsu Kim / YuKiSa

(https://github.com/shuuki4)

(https://twitter.com/shuuki4)
(mailto:123bskim@naver.com)

$$G = \gamma G + (1-\gamma)(\nabla_\theta J(\theta_t))^2$$

$$\Delta_\theta = \frac{\sqrt{s+\epsilon}}{\sqrt{G+\epsilon}} \cdot \nabla_\theta J(\theta_t)$$

$$\theta = \theta - \Delta_\theta$$

$$s = \gamma s + (1-\gamma)\Delta_\theta^2$$

얼핏 보면 왜 이러한 식이 도출되었는지 이해가 안될 수 있지만, 이는 사실 Gradient Descent와 같은 first-order optimization 대신 Second-order optimization을 approximate 하기 위한 방법이다. 실제로 논문 (http://www.matthewzeiler.com/pubs/googleTR2012/googleTR2012.pdf)의 저자는 SGD, Momentum, Adagrad와 같은 식들의 경우 $\Delta\theta$의 unit(단위)을 구해보면 θ의 unit이 아니라 θ의 unit의 역수를 따른다는 것을 지적한다. θ의 unit을 $u(\theta)$라고 하고 J는 unit이 없다고 생각할 경우, first-order optimization에서는

$$\Delta\theta \propto \frac{\partial J}{\partial \theta} \propto \frac{1}{u(\theta)}$$

이다. 반면, Newton method와 같은 second-order optimization을 생각해보면

$$\Delta\theta \propto \frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}} \propto u(\theta)$$

이므로 바른 unit을 가지게 된다. 따라서 저자는 Newton's method를 이용하여 $\Delta\theta$가 $\frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}}$ 라고 생각한 후, $\frac{1}{\frac{\partial^2 J}{\partial \theta^2}} = \frac{\Delta\theta}{\frac{\partial J}{\partial \theta}}$ 이므로 이를 분자의 Root Mean Square, 분모의 Root Mean Square 값의 비율로 근사한 것이다. 더욱 자세한 설명을 원하시는 분은 논문을 직접 읽어보시길 바란다.

Adam

Adam (Adaptive Moment Estimation)은 RMSProp과 Momentum 방식을 합친 것 같은 알고리즘이다. 이 방식에서는 Momentum 방식과 유사하게 지금까지 계산해온 기울기의 지수평균을 저장하며, RMSProp과 유사하게 기울기의 제곱값의 지수평균을 저장한다.

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

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

다만, Adam에서는 m과 v가 처음에 0으로 초기화되어 있기 때문에 학습의 초반부에서는 m_t, v_t가 0에 가깝게 bias 되어있을 것이라고 판단하여 이를 unbiased 하게 만들어주는 작업을 거친다. m_t 와 v_t의 식을 $\sum$ 형태로 펼친 후 양변에 expectation을 씌워서 정리해보면, 다음과 같은 보정을 통해 unbiased 된 expectation을 얻을 수 있다. 이 보정된 expectation들을 가지고 gradient가 들어갈 자리에 $\hat{m}_t$, G_t가 들어갈 자리에 $\hat{v}_t$를 넣어 계산을 진행한다.

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

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

$$\theta = \theta - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}}\hat{m}_t$$

보통 β_1 로는 0.9, β_2로는 0.999, ϵ 으로는 10^{-8} 정도의 값을 사용한다고 한다.

Summing up

지금까지 다양한 Gradient Descent 알고리즘의 변형 알고리즘들을 알아보았다. Momentum, NAG, AdaGrad, AdaDelta, RMSProp, Adam 등 다양한 알고리즘들이 있었지만 이 중에서 어느 알고리즘이 가장 좋다, 라고 말하기는 힘들다. 어떤 문제를 풀고있는지, 어떤 데이터셋을 사용하는지, 어떤 네트워크에 대해 적용하는지에 따라 각 방법의 성능은 판이하게 차이가 날 것이므로 실제로 네트워크를 학습시킬 때는 다양한 시도를 해보며 현재 경우에서는 어떤 알고리즘이 가장 성능이 좋은지에 대해 실험해 볼 필요가 있다. 다행히, Tensorflow 등의 Machine learning library를 사용하면 코드 한 줄만 변화시키면 쉽게 어떤 optimizer를 사용할 것인지 설정해줄 수 있어 간편하게 실험해볼 수 있을 것이다.

여기서 설명한 알고리즘들은 모두 Stochastic Gradient Descent, 즉 단순한 first-order optimization의 변형들이다. 이 외에도 Newton's Method 등 second-order optimization을 기반으로 한 알고리즘들도 있다. 그러나 단순한 second-order optimization을 사용하기 위해서는 Hessian Matrix라는 2차 편미분 행렬을 계산한 후 역행렬을 구해야 하는데, 이 계산과정이 계산적으로 비싼 작업이기 때문에 보통 잘 사용되지 않는다. 이러한 계산량을 줄이기 위해 hessian matrix를 근사하거나 추정해나가면서 계산을 진행하는 BFGS / L-BFGS 등의 알고리

즘, 그리고 hessian matrix를 직접 계산하지 않으면서 second-order optimization인 Hessian-Free Optimization 등도 존재한다. 이러한 부분들에 대해서는 나중에 기회가 되면 공부해서 언급해보도록 하겠다.

Deep Learning (3) (/categories.html#Deep Learning-ref)

Deep Learning (3) , (/tags.html#Deep Learning-ref)
Gradient Descent (1) , (/tags.html#Gradient Descent-ref)
Optimization (1) (/tags.html#Optimization-ref)

Beomsu Kim

Beomsu Kim / YuKiSa

← Previous (/notice/2016/05/17/%EB%B8%94%EB%A1%9C%EA%B7%B8-%EC%9D%B4%EC%A0%84.html)

Next → (/deep%20learning/2016/08/24/Synthetic-Gradient-%EB%85%BC%EB%AC%B8-%EC%A0%95%EB%A6%AC.html)