Overview of gradient descent optimization algorithms

HYUNG IL KOO

Based on

http://sebastianruder.com/optimizing-gradient-descent/

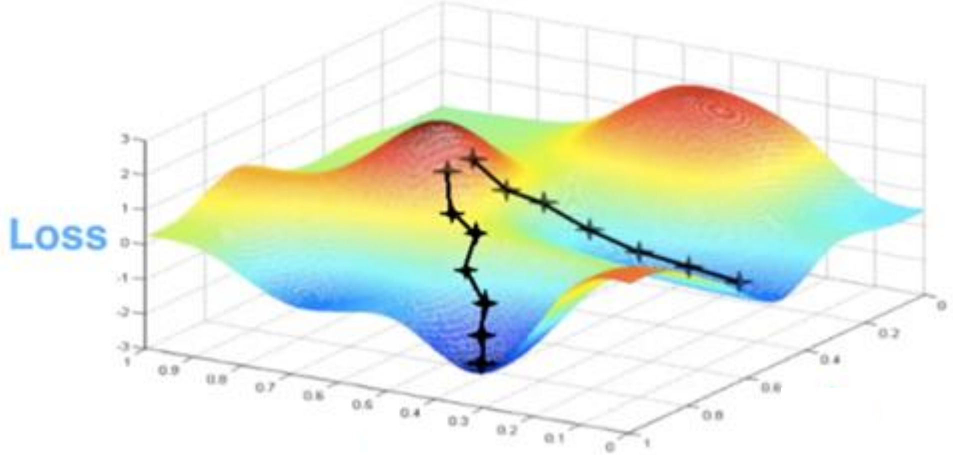
Problem Statement

- Machine Learning → Optimization Problem
  - Training samples: $\{(x^{(i)}, y^{(i)})\}$
  - Cost function: $J(\theta; X; Y) = \sum_i \ d(f(x^{(i)}; \theta), y^{(i)})$

$$\hat{\theta} = \arg\min_{\theta} J(\theta; X; Y)$$

Optimization method

- Gradient Descent
  - The most common way to optimize neural networks
    - Deep learning library contains implementations of various gradient descent algorithms
  - To minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \Re^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ with respect to the parameters.



CONTENTS

- Gradient descent variants
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent

- Challenges

- Gradient descent optimization algorithms
  - Momentum
  - Adaptive Gradient
- Visualization
- Which optimizer to choose?
- Additional strategies for optimizing SGD
  - Shuffling and Curriculum Learning
  - Batch normalization

GRADIENT DESCENT VARIANTS

Batch gradient descent

- Computes the gradient of the cost function w.r.t. θ for **the entire training dataset**:

$$\theta^{new} = \theta^{old} - \eta \nabla_\theta J(\theta; X; Y)$$

- Properties
  - Very slow
  - Intractable for datasets that don't fit in memory
  - No online learning

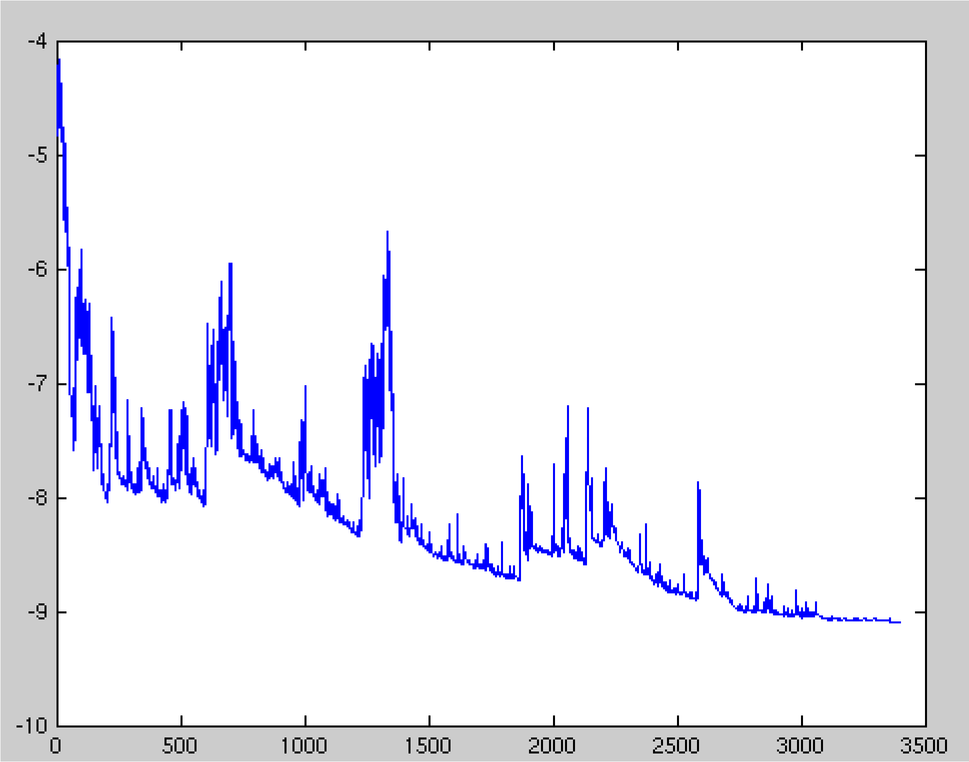
Stochastic Gradient descent (SGD)

- To perform a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$

$$\theta^{new} = \theta^{old} - \eta \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)})$$

- Properties:
  - Faster
  - Online learning
  - Heavy fluctuation
  - Capability to jump to new (potentially better local minima)
  - Complicated convergence (overshooting)

SGD fluctuation



Batch Gradient vs SGD

Batch gradient

- It converges to the minimum of the basin the parameters are placed in.

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Stochastic gradient descent

- It is able to jump to new and potentially better local minima.
- This complicates convergence to the exact minimum, as SGD will keep overshooting

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Mini-batch (stochastic) gradient descent

- To perform an update for every mini-batch of n training examples:

$$\theta^{new} = \theta^{old} - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Properties of mini-batch gradient descent

- Compared with SGD
  - It reduces the variance of the parameter updates, which can lead to more stable convergence;
  - It can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient

- Mini-batch gradient descent is **typically the algorithm of choice** when training a neural network and the term **SGD** usually is employed also when mini-batches are used.

CHALLENGES

Challenges

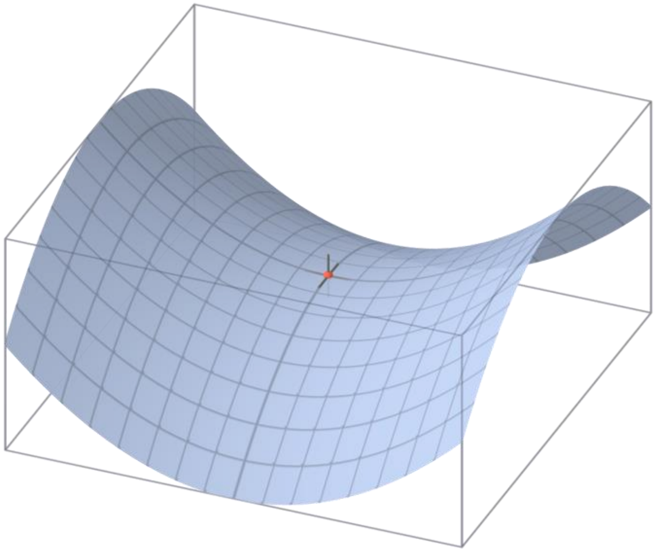
- Choosing a proper learning rate can be difficult.
  - Small learning rate leads to painfully slow convergence
  - Large learning rate can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge

- Learning rate schedules and thresholds
  - It has to be defined in advance and unable to adapt to a dataset's characteristics.

- Same learning rate applies to all parameter updates.
  - If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.

Challenges

- Avoiding getting trapped in their numerous suboptimal local minima and saddle points
  - Dauphin et al. argue that the difficulty arises in fact **not from local minima but from saddle points**.
  - These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.



MOMENTUM

Momentum

- One of the main limitations of gradient descent) is local minima
  - When the gradient descent algorithm reaches a local minimum, the gradient becomes zero and the weights converge to a sub-optimal solution

- A very popular method to avoid local minima is to compute a temporal average direction in which the weights have been moving recently
  - An easy way to implement this is by using an exponential average

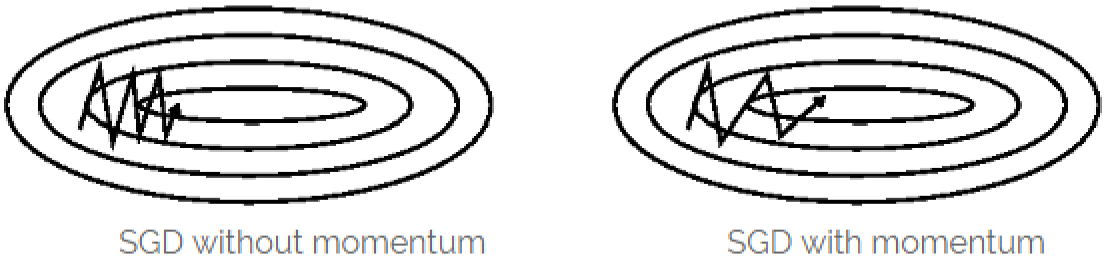
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta^{new} = \theta^{old} - v_t$$

  - The term γ is called the momentum
    - The momentum has a value between 0 and 1 (typically 0.9)

- Properties
  - Fast convergence
  - Less oscillation

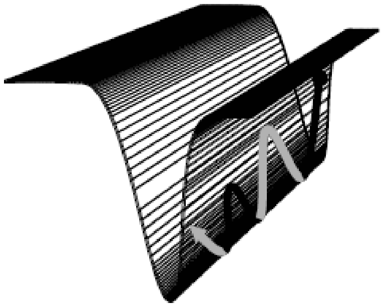
Momentum

- Essentially, when using momentum, we push **a ball down a hill**. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance).
- The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.



Momentum

- The momentum term is also useful in spaces with long ravines characterized by sharp curvature across the ravine and a gently sloping floor
  - Sharp curvature tends to cause divergent oscillations across the ravine
    - To avoid this problem, we could decrease the learning rate, but this is too slow
  - The momentum term filters out the high curvature and allows the effective weight steps to be bigger
  - It turns out that ravines are not uncommon in optimization problems, so the use of momentum can be helpful in many situations



- However, a momentum term can hurt when the search is close to the minima (think of the error surface as a bowl)
  - As the network approaches the bottom of the error surface, it builds enough momentum to propel the weights in the opposite direction, creating an undesirable oscillation that results in slower convergence

Smarter Ball?

NGD (Nesterov accelerated gradient)

- Nesterov accelerated gradient improved on the basis of Momentum algorithm
  - Approximation of the next position of the parameters.

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

ADAPTIVE GRADIENTS

Adaptive Gradient Methods

- Let us adapt **the learning rate of each parameter**, performing larger updates for infrequent and smaller updates for frequent parameters.

- Methods
  - AdaGrad (Adaptive Gradient Method)
  - AdaDelta
  - RMSProp (Root Mean Square Propagation)
  - Adam (Adaptive Moment Estimation)

Adaptive Gradient Methods

- These methods use a different learning rate for each parameter $\theta_i \in \Re$ at every time step t.
  - For brevity, we set $g_i^{(t)}$ to be the gradient of the objective function w.r.t. $\theta_i \in \Re$ at time step t:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \boldsymbol{\eta} \cdot g_i^{(t)}$$

  - These methods modify the learning rate $\boldsymbol{\eta}$ at each time step (t) for every parameter θ_i based on the past gradients that have been computed for θ_i.

Adagrad

- Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{G_{t,i} + \epsilon}}} g_{t,i}$$

- $G_{t,i} = \sum_{k \le t} \left(g_i^{(k)} \right)^2$

$$g_i^{(k)} = \left. \frac{\partial J(\theta)}{\partial \theta_i} \right|_{\theta^{(k)}}$$

Adagrad

- Pros
  - It eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01.

- Cons
  - Its accumulation of the squared gradients in the denominator: the accumulated sum keeps growing during training. This causes the learning rate to shrink and eventually become infinitesimally small. The following algorithms aim to resolve this flaw.

RMSprop

- RMSprop has been developed to resolve Adagrad's diminishing learning rates.

$$\lambda_{t,i} = \gamma\, \lambda_{t-1,i} + (1-\gamma)\left(g_i^{(t)}\right)^2$$

$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\boldsymbol{\eta}}{\sqrt{\boldsymbol{\lambda_{t,i}} + \boldsymbol{\epsilon}}} g_{t,i}$$

η : Learning rate is suggest to set to 0.001
γ : Fixed momentum term

Adam (Adaptive moment Estimation)

- Adam keeps an exponentially decaying average of past gradients m_t, similar to momentum.

$$m_{t,i} = \beta_1 m_{t-1,i} + (1-\beta_1) g_i^{(t)}$$

$$v_{t,i} = \beta_2 v_{t-1,i} + (1-\beta_2) \left(g_i^{(t)}\right)^2$$

m_t : Estimate of the first moment(the mean)
v_t : Estimate of the second moment (the un-centered variance)
β_1 : suggest to set to 0.9
β_2 : suggest to set to 0.999

- They counteract these biases by computing bias-corrected first and second moment estimates.

$$\hat{m}_{t,i} = \frac{m_{t,i}}{1-\beta_1} \qquad \hat{v}_{t,i} = \frac{v_{t,i}}{1-\beta_2}$$

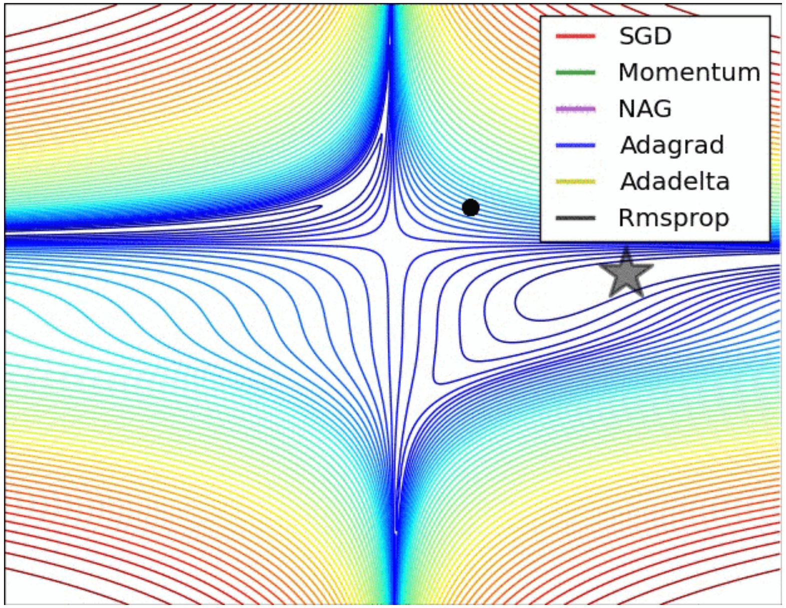
- which yields the Adam update rule.

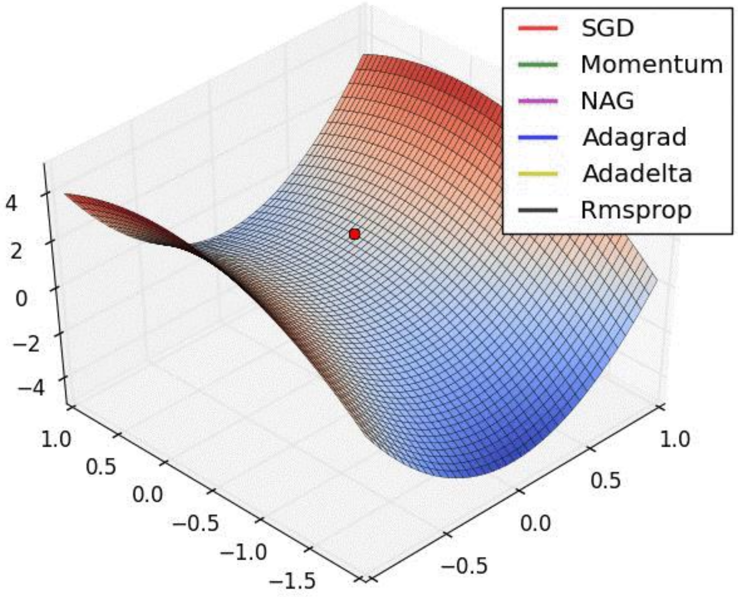
$$\theta_i^{(t+1)} = \theta_i^{(t)} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_{t,i}$$

ϵ : suggest to set to 10^{-8}

COMPARISON

Visualization of algorithms





Which optimizer to choose?

- RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates.

- Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser.

CONCLUSION

Conclusion

- Three variants of gradient descent, among which mini-batch gradient descent is the most popular.