CS 206 Principles of Scientific Computing

Xiaohui Xie

University of California, Irvine

xhx@uci.edu

May 30, 2017

Batch gradient descent

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

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

Pros:
Guaranteed to converge to the global minimum for convex cost function and to a local minimum for non-convex cost function.
Cons:

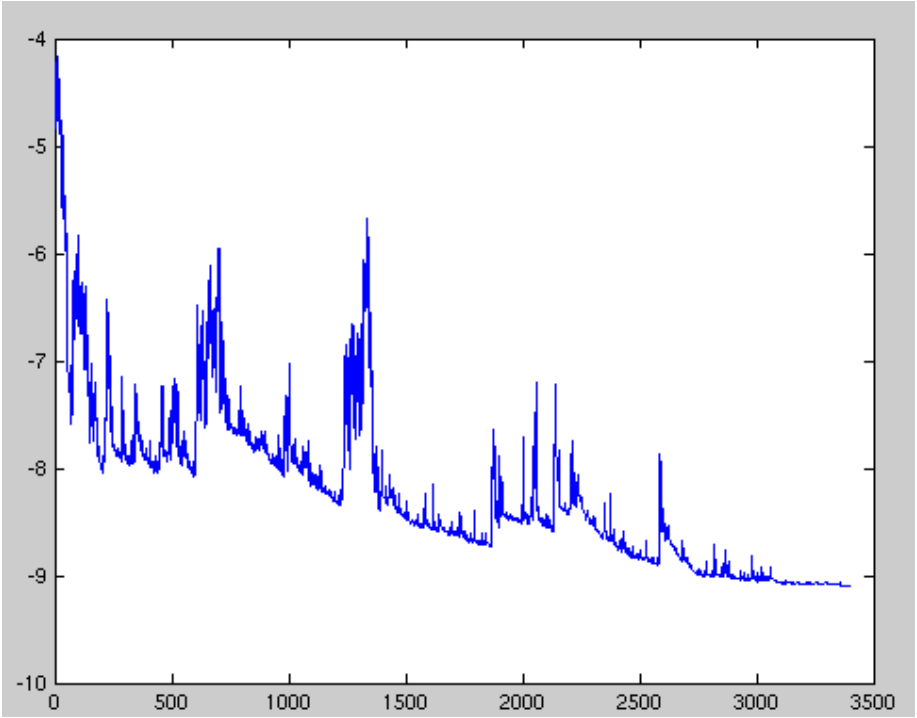
- can be slow
- intractable for datasets that don't fit in memory
- cannot update models online (with new examples on-the-fly)

Stochastic gradient descent (SGD)

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

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

Stochastic gradient descent (SGD)



Pros: Fast and can be used online

Cons:

- high variance
- cost function fluctuates heavily

Mini-batch gradient descent

Performs an update for every mini-batch of n training examples

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

- Seeks a middle ground between batch and stochastic gradient descent
- Reduces the variance of the parameter updates, which can lead to more stable convergence
- can make use of highly optimized matrix optimizations
- Common mini-batch sizes range between 20 and 500, but can vary for different applications.

Mini-batch gradient descent

```
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
```

Some challenges

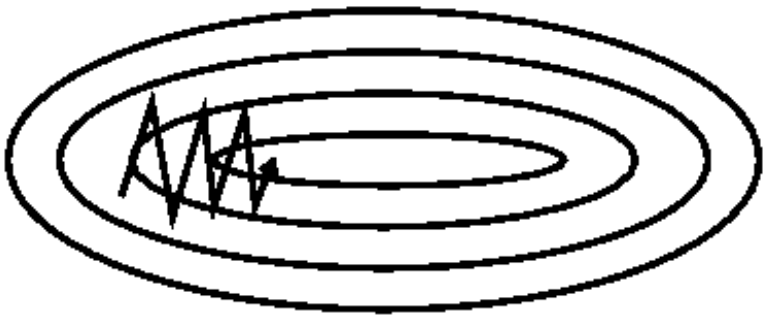
- How to choose the learning rate?
- Some methods adjust the learning rate during training by e.g. annealing. But still need to be determined beforehand
- The same learning rate applies to all parameter updates, which could be problematic for sparse features. Would be better to use higher rates for sparse features since they don't got updated often.
- How to avoid local optimal or saddle points?

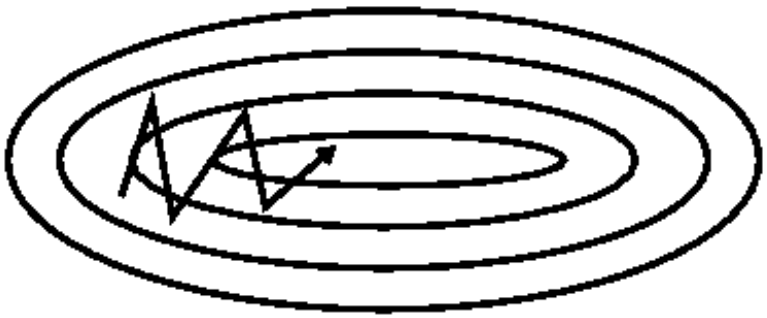
Gradient descent optimization techniques

- Momentum
- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- Adam

Momentum

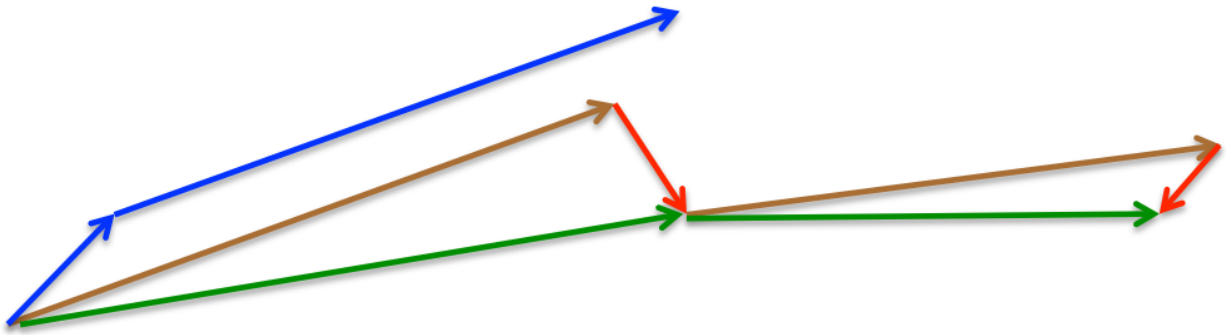
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$





Nesterov accelerated gradient

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



Adagrad

Adapt the learning rate for each parameter: larger updates for infrequently updated parameters.

$$g_t = \nabla_\theta J(\theta)$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$
$$G_{t+1} = G_t + g_t^2$$

RMSprop

Similar to Adagrad, but estimate G_t using running average.

$$g_t = \nabla_\theta J(\theta)$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$
$$G_{t+1} = 0.9 G_t + 0.1 g_t^2$$

Adam

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter.

$$
\begin{aligned}
g_t &= \nabla_\theta J(\theta) \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t
\end{aligned}
$$

Additional strategies for optimizing SGD

- Shuffling and Curriculum Learning - solve progressively harder problems, supplying the training examples in a meaningful order
- Batch normalization
- Early stopping - monitor error on a validation set during training and stop if your validation error does not improve enough
- Gradient noise

Reference:

Sebastian Ruder, An overview of gradient descent optimization algorithms, arXiv:1609.04747