Chapter 4. Training Models
Kinds of Gradient Descent

CSC357 Advanced Topics—Machine Learning

24 January 2020

- Batch Gradient Descent
  - need to compute gradient of cost function w/ respect to each model parameter θ_j
  - calculate how much the function will change w/ small change in θ_j
  - (a partial derivative)
  - like asking "What is the slope of the mountain under my feet if I face east?" and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions)

$$\frac{\partial}{\partial\theta_j}MSE(\theta = \frac{2}{m}\sum_{i=1}^{M}(\theta^T\mathbf{x}^{(i)} - y^{(i)})x_j^{(i)}$$

  - can compute all partial derivatives at once
  - gradient vector ($\nabla_\theta MSE(\theta)$) contains all the partial derivatives of the cost function (one for each model parameter).

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}MSE(\theta) \\ \\ \frac{\partial}{\partial\theta_1}MSE(\theta) \\ \\ \vdots \\ \\ \frac{\partial}{\partial\theta_n}MSE(\theta) \end{pmatrix} = \frac{2}{m}\mathbf{X}^T(\mathbf{X}\theta - y)$$

- formula involves calculations over full training set X, at each Gradient Descent step!
- *Batch Gradient Descent* uses whole batch of training data at every step (
- terribly slow on very large training sets
- scales well with the number of features
- training a Linear Regression model w/ hundreds of thousands of features much faster using Gradient Descent than using the Normal Equation or SVD decomposition.
- get gradient vector—it points uphill
- to go downhill, go in opposite directionjust go in oppposite direction
- subtract $\nabla_\theta MSE(\theta)$ from to get that direction
- multiply gradient vector by learning rate η to determine size of downhill step
- ♭♭ Equation 4-7. Gradient Descent step

$$\theta^{\text{nextstep}} = \theta \nabla_\theta MSE(\theta)$$

- a quick implementation of this algorithm...

```
eta = 0.1  # learning rate
n_iterations = 1000
m = 100

theta = np.random.randn(2,1)  # random initialization

for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

- not too hard! look at resulting θ...

```
theta
array([[4.21509616],
       [2.77011339]])
```

- exactly what Normal Equation found!
- Gradient Descent worked perfectly
- what if we had used different learning rate η?
- Figure4-8 shows first 10 steps of Gradient Descent using 3 different learning rates
  - on left, the learning rate is too low

    - (algorithm eventually reaches solution—after long time)
    - in middle, learning rate looks pretty good
    - (in just a few iterations, has already converged to solution)
    - on right, the learning rate is too high
    - (algorithm diverges—jumps all over the place moves further away from the solution at every step)
  - to find good learning rate, could use grid search
  - may want to limit number of iterations so that grid search can eliminate models that take too long to converge
  - how to set number of iterations?
    - if too low, will still be far away from optimal solution when algorithm stops
    - if too high, waste time while model parameters ceasing changing
    - simple solution—set a very large number of iterations but interrupt the algorithm when gradient vector becomes tiny
    - that is, when norm becomes $<$ tiny number ϵ (the tolerance)
    - small norm means Gradient Descent has (almost) reached the minimum
  - Convergence Rate
    - when cost function is convex and slope does not change abruptly
    - (as is the case for the MSE cost function)
    - Batch Gradient Descent with fixed learning rate will eventually converge to the optimal solution
    - although might have to wait—can take $O(1/\epsilon$ iterations to reach optimum within range of ϵ
    - (depending on shape of the cost function)
    - divide tolerance by 10 to more precise solution, then algorithm might run about 10 times longer

- Stochastic Gradient Descent
  - main problem w/ Batch Gradient Descent $\rightarrow$ uses whole training set to compute gradients at every step
  - makes it very slow when training set is large
  - at opposite extreme, Stochastic Gradient Descent picks random instance in training set at every step and computes gradients based only on that single instance
  - much faster! (little data to manipulate at every iteration)
  - possible to train on huge training sets
  - (only one instance needs to be in memory at each iteration)

- (Stochastic GD can be implemented as out-of-core algorithm)
- stochastic (random) → algorithm << regular than Batch Gradient Descent
- (instead of gently decreasing until it reaches the minimum)
- cost function will bounce up and down, decreasing only on average
- gets very close to minimum, but continues to bounce around, never settles down
- final parameter values are good, but not optimal
- very irregular cost function can help algorithm jump out of local minima
- Stochastic Gradient Descent has better chance of finding global minimum than Batch Gradient Descent
- randomness is good to escape from local optima
- randomness bad because it means algorithm never settles at minimum
- one solution—gradually reduce learning rate
- steps start large (helps make quick progress, escape local minima)
- steps then get smaller and smaller, allows algorithm to settle at global minimum
- process like *simulated annealing*
- annealing is term from metallurgy—slow cooling of molten metal
- function that determines learning rate at each iteration is the *learning schedule*
- if learning rate reduced too quickly, might get stuck in local minimum
- if learning rate reduced too slowly, might jump around minimum, end with suboptimal solution (if training halted too early)
- here is a simple learning schedule...

```
n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
```

```
gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
eta = learning_schedule(epoch * m + i)
theta = theta - eta * gradients
```

- (by convention) iterate by rounds of m iterations
- each round is called an epoch
- while Batch Gradient Descent code iterated 1,000 times through whole training set...
- ...this code goes through training set only 50 times, reaches pretty good solution

```
theta
array([[4.21076011],
       [2.74856079]])
```

- Figure 4-10 shows first 20 steps of training (notice how irregular the steps are)
- since instances are picked randomly
  - some instances might picked several times per epoch
  - while others may not be picked at all
- to be sure that algorithm goes through every instance at each epoch
  - shuffle the training set
  - (be sure to shuffle input features and the labels jointly)
  - go through instance by instance, then shuffle again
  - approach generally converges more slowly
- w/ Stochastic Gradient Descent, training instances must be independent and identically distributed (IID)
- ...to ensure that parameters get pulled toward global optimum, on average
- (simple way to ensure this: shuffle instances during training)
- (for example, pick each instance randomly, or shuffle training set at beginning of each epoch)
- without shuffling instances (e.g.,if instances are sorted by label)—then SGD will start by optimizing for one label, then next, and so on—will not settle close to the global minimum
- use Scikit-Learn's SGDRegressor **class** to perform Linear Regression using Stochastic GD
- (defaults to optimizing squared error cost function)
- following code runs for maximum $1,000$ epochs or until loss drops by less than 0.001 during one epoch
- (max_iter=1000, tol=1e−3)

- starts with a learning rate of 0.1 ($\eta_0 = 0.1$), using default learning schedule (different from preceding one).
- does not use any regularization (penalty=None; more details on this shortly):

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3,
        penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

- solution quite close to one returned by Normal Equation

```
sgd_reg.intercept_, sgd_reg.coef_(array([4.24365286]),
        array([2.8250878]))
```

- Mini-batch Gradient Descent
  - simple to understand once you know Batch and Stochastic Gradient Descent
  - computes the gradients on small random sets of instances called mini-batches
  - (Batch GD uses full set, Stochastic GD uses just one instance)
  - can get performance boost from hardware optimization of matrix operations (esp. w/ when GPUs)
  - progress in parameter space less erratic than with Stochastic GD
  - (especially with fairly large mini-batches)
  - Mini-batch GD will walk around closer to minimum than Stochastic Gd
  - but may struggle to escape from local minima
  - (problems other than Linear Regression can have local minima)
  - figure in book shows paths taken by 3 Gradient Descent algorithms in parameter space during training.
  - all end up near minimum
  - Batch GDs path actually stops at the minimum
  - both Stochastic GD and Mini-batch GD continue to walk around
  - (don't forget Batch GD takes lots of time at each step)
  - (also, Stochastic GD and Mini-batch GD would also reach minimum, given good learning schedule)
- compare algorithms weve discussed for Linear Regression
- (recall m is number of training instances, n is number of features)
- little difference—all algorithms produce similar models, make predictions in same way