Stochastic Gradient Descent

Monday 8 April 2024 2:13 AM

→ works well with big data also

→ Only one single row is taken into consideration to take a single step

→ for each epoch we will take n steps. Where n = # rows

→ Because of these frequent updates, we will reach the minima quicker (won't require many epochs)

↳ faster convergence

Stochastic → row is not chosen sequentially, row is randomly selected.

→ Drawback: doesn't give a steady solution i.e. if we run SGD on the same data multiple times we will obtain different answers. We are choosing rows at random → random updates → even though these random updates do take us to the answer, it doesn't it will give us the most optimal answer, it will output values (↳ varies) close to the optimal value.

Randomness in solution. Slightly imprecise solution.

→ Time complexity

# of epochs → fixed

e=100

Batch → 100

Stochastic → 100 × n

→ Although SGDRegressor converges to the solution faster it doesn't mean for the same # of epochs SGDRegressor is faster than BGDRegressor.

→ But since SGDRegressor doesn't require as many epochs, (e ↓ x n) ↓ that is why overall convergence of SGDRegressor will be faster on even bigger datasets.

→ In BGD, each successive step reduced the loss function but SGD doesn't guarantee this. In SGD, step (n+1) may have a higher loss than step n.

→ Since we are only relying on a single row, SGD decision making seems rash when visualised. But over a period of time it reaches close to the optimal solution.

→ When to use Stochastic GD?

- Big data (SGD will converge faster, & will give good results) (BGD will be very slow, or worse it may not run due to vectorization)
- Non-convex function (BGD might get stuck at local minima if we start at the wrong initial values as it converges smoothly towards the solution. Whereas due to the "rash" behaviour of SGD it may be able to escape the local minima & find it's way to the optimal solution, we can say SGD has "velocity"/"momentum" which BGD lacks. Given we carefully monitor the learning rate)

→ Another problem:

- Even after reaching close to the solution, b/c of the randomness involved, the still fluxuates a lot. Ideal situation is to minimize the fluxuation as we approach the solution.

↳ learning schedule → Vary learning rate with epochs → ↓ learning rate as epochs increases.

↓

decreasing function

Dampen the "rashness"

So that first iterations cause large changes in the parameters, while the later ones only do fine-tuning.

e.g.

```
t0,t1 = 5,50
def learning_rate(t):
    return t0/(t + t1)

for i in range(epochs):
    for j in range(X.shape[0]):

        lr = learning_rate(i * X.shape[0] + j)
```

```
        lr = learning_rate(i * X.shape[0] + j)
```

large changes in the parameters, while the later ones only do fine-tuning.

→ sklearn.linear_model → SGDRegressor

→ Explore SGDRegressor's partial_fit method. → out of core learning

```
from sklearn.linear_model import SGDRegressor

sgd = SGDRegressor(learning_rate='constant',eta0=0.2)

batch_size = 35

for i in range(100):

    idx = random.sample(range(X_train.shape[0]),batch_size)
    sgd.partial_fit(X_train[idx],y_train[idx])
```

partial_fit(*X, y, sample_weight=None*) [source]

Perform one epoch of stochastic gradient descent on given samples.

Internally, this method uses `max_iter = 1`. Therefore, it is not guaranteed that a minimum of the cost function is reached after calling it once. Matters such as objective convergence and early stopping should be handled by the user.

Parameters:

X : *{array-like, sparse matrix}, shape (n_samples, n_features)*
Subset of training data.

y : *numpy array of shape (n_samples,)*
Subset of target values.

sample_weight : *array-like, shape (n_samples,), default=None*
Weights applied to individual samples. If not provided, uniform weights are assumed.

Returns:

self : *object*
Returns an instance of self.

→ SGDRegressor — loss, max_iter (epochs), tol, early_stopping, shuffle, random_state, learning_rate, eta0, power_t

→ Speed ✓ stability X

class `sklearn.linear_model.`**SGDRegressor**(*loss='squared_error', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, random_state=None, learning_rate='invscaling', eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, warm_start=False, average=False*) [source]

Linear model fitted by minimizing a regularized empirical loss with SGD.

SGD stands for Stochastic Gradient Descent: the gradient of the loss is estimated each sample at a time and the model is updated along the way with a decreasing strength schedule (aka learning rate).

The regularizer is a penalty added to the loss function that shrinks model parameters towards the zero vector using either the squared euclidean norm L2 or the absolute norm L1 or a combination of both (Elastic Net). If the parameter update crosses the 0.0 value because of the regularizer, the update is truncated to 0.0 to allow for learning sparse models and achieve online feature selection.

This implementation works with data represented as dense numpy arrays of floating point values for the features.