Gradient Descent

Thursday, April 7, 2022 7:44 AM

→ optimization

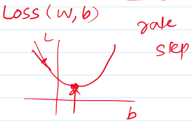
Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks.

$J(\theta)$ = Loss function

Neural

Loss (w, b)

$\eta \frac{\partial L}{\partial b}$



Gradient descent is a way to minimize an objective function J(θ) parameterized by a model's parameters θ∈Rd by updating the parameters in the opposite direction of the gradient of the objective function ∇θJ(θ) w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.

Backpropagation Algorithm

epochs = 5 ↙ sto

1 point

gradient descent

for i in range(epochs):

- for j in range(X.shape[0]):
  - → Select 1 row (random)
  - → Predict (using Forward prop)
  - → Calculate loss (using loss function → mse)
  - → Update weights and bias using GD

$$\boxed{W_n = W_0 - \eta \frac{\partial L}{\partial W}}$$

- → Calculate avg loss for the epoch
  - L avg

50 points

1 epoch

↳ 50 times

w, b update

Batch GD

epoch = 5

↳ 50 points → predict J

↗ current weigh

↑

dot product

y_hat = np.dot(X, w) + b

↳ 50 predict

y = 50 actual ocw.

y y_hat J loss

$\partial \int - \sum_{i=1}^{50}$

{Batch / Stochastic / Mini batch}

1 single (w, b)

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

$\frac{\partial L}{\partial w}$ derivative

↓

3 types

accuracy → time → tradeoft.

Batch GD (Vanilla GD)

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

entire dataset J update

epochs = # of updates.

Stochastic GD

```
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
```

50×10 = 500

5 epochs

↳ 5 times

↓

w, b update

epoch → 10 (50 rows) 50 rows

for i in range(10):

→ shuffle

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

10 × 50 = 500 update

↳ 1 random point

↳ y_hat → forward

↳ loss

↳ w, b update → $W_n = W_0 - \eta \frac{\partial L}{\partial W}$

→ bias eliminate

avg loss print → for the epoch

frequency of weight update higher

Batch GD → 1 loop

epoch = 10

for i in range(10): ↗ epoch

↓ loop rows

→ total → 10 times w, b update

y_hat = np.dot(X, W) + b

50 values dot → smart replacement → loops

↳ vectorization ~ faster loop

y = 50 values

50 values   dot → smart replacement → loops

y = 50 values   ↳ vectorization ↘ faster loop ↑ Optimized

$\hat{y}, y$ → loss

w, b update $w_n = w_0 - \eta \frac{\partial L}{\partial w}$

→ loss

Which is faster (given same no. of epochs)

10n   b

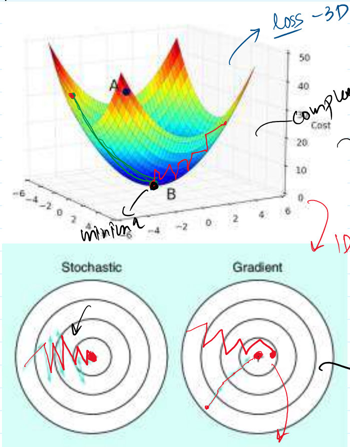
320 (n)   1

Which is the faster to converge (given same # epochs)

S

Loss function → 2 Algo

loss -3D



complex

minima

1D

exact solution

Approximate diff

Stochastic ↳ random → point → 1 updates

Batch → all point → 1 update → dataset

Spikey SGD useful

Yes   No

local minima

SGD

BGD

SGD → help the algo to move out of local minima

Vectorization → faster × loops → big dataset

└→ np.dot(X , w) + b

X → dataset of 10 crore   RAM

Mini Batch Gradient Descent

SGD → BGD

```
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
```

$\frac{n}{x}$ = # of batches   Keras   batch-size = x

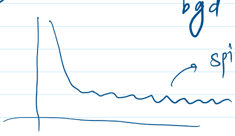
updates/epoch

bgd > mbgd > sgd

con

bgd < mbgd < sgd

MBGD



spikey SGD → Vectorization → {smaller batch}

BGD ↔ SGD   Best of both worlds

320 rows → batches 32

In every epoch 10 batches

10 update

for i in epochs →
  for j in num of batch
    1 batch
np.dot   ↳ y-pred (vector)
    ↳ loss
    ↳ update

→ Why batch_size is provided in multiple of(2)?

2, 4, 8, 32, 64, 128, 256

10, 15

RAM effective → binary

keras

RAM → optimization

faster

→ What if batch_size doesn't divide # rows properly

e.g # of rows $n = 400$

batch_size = 150

$\# \text{ of batch} = \frac{400}{150} = \boxed{2.66}$

3

150, 150, left 100

1 batch 2 batch 3rd batch