2019
怪兽
学堂

Gradient descent and optimizer

虾米 2019-3

Gradient Descent for Linear Regression

Goal: minimize the following loss function:

predict with: $\hat{y}^i = \sum_j^n w_j \phi_j(\mathbf{x}^i)$

$$J_{\mathbf{X},\mathbf{y}}(\mathbf{w}) = \underbrace{\sum_i}_{\text{sum over } n \text{ examples}} \left(y^i - \hat{y}^i \right)^2 = \sum_i \left(y^i - \underbrace{\sum_j}_{\text{sum over } k+1 \text{ basis vectors}} w_j \phi_j(\mathbf{x}^i) \right)^2$$

Gradient Descent for Linear Regression

Goal: minimize the following loss function:

$$\text{predict with: } \hat{y}^i = \sum_j^n w_j \phi_j(\mathbf{x}^i)$$

$$J_{\mathbf{x},\mathbf{y}}(\mathbf{w}) = \sum_i \left(y^i - \hat{y}^i\right)^2 = \sum_i \left(y^i - \sum_j w_j \phi_j(\mathbf{x}^i)\right)^2$$

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = \frac{\partial}{\partial w_j} \sum_i \left(y^i - \hat{y}^i\right)^2$$

$$= 2\sum_i \left(y^i - \hat{y}^i\right) \frac{\partial}{\partial w_j} \hat{y}^i$$

$$= 2\sum_i \left(y^i - \hat{y}^i\right) \frac{\partial}{\partial w_j} \sum_j w_j \phi_j(\mathbf{x}^i)$$

$$= 2\sum_i \left(y^i - \hat{y}^i\right) \phi_j(\mathbf{x}^i)$$

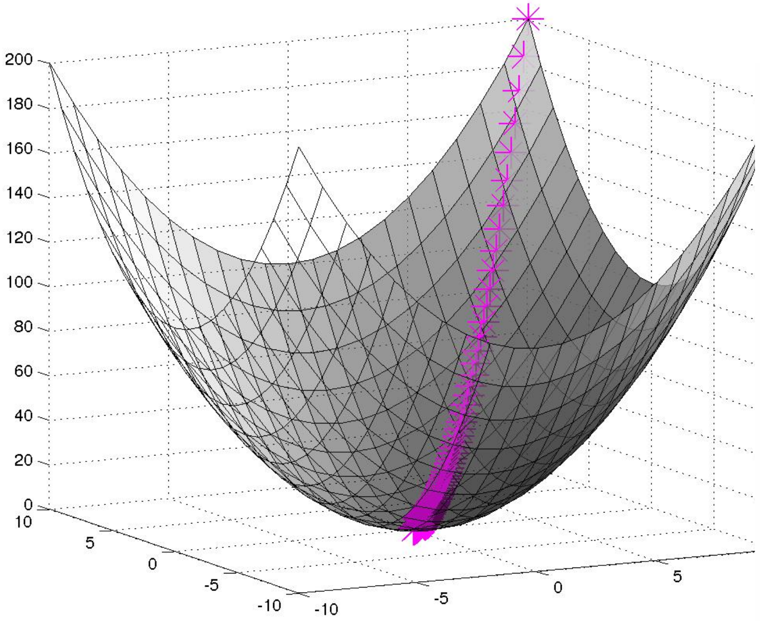
Gradient Descent for Linear Regression

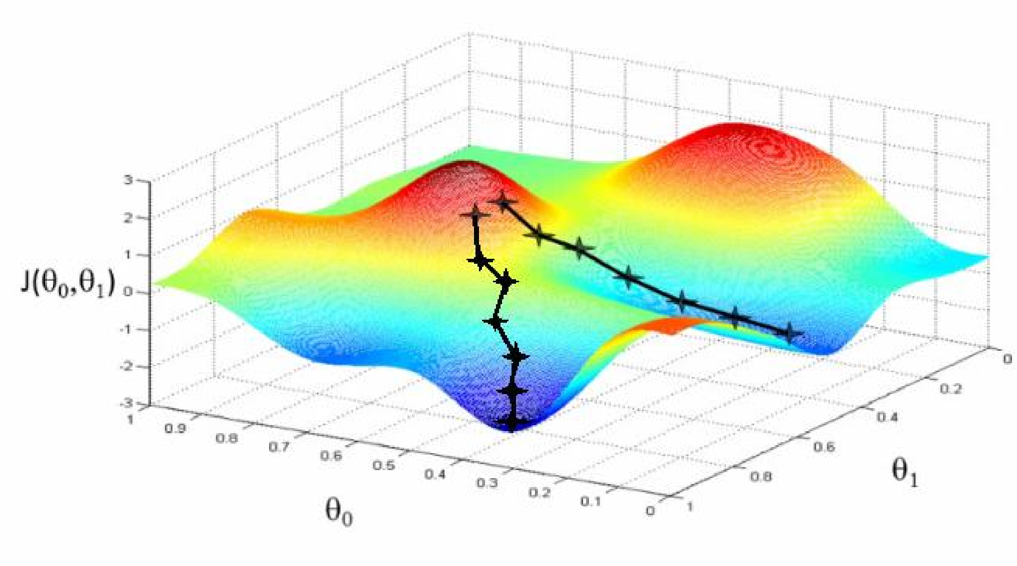
Learning algorithm:

- Initialize weights **w=0**
- For t=1,… until convergence:
  - Predict for each example $\boldsymbol{x}^i$ using **w:** $$\hat{y}^i = \sum_{j=0}^{k} w_j \phi_j(\mathbf{x}^i)$$
  - Compute gradient of loss: $$\frac{\partial}{\partial w_j} J(\mathbf{w}) = 2\sum_i \left(y^i - \hat{y}^i \right) \phi_j(\mathbf{x}^i)$$
  - This is a vector **g**
  - Update: **w = w** – λ**g**
  - λ is the learning rate.

Linear regression is a *convex* optimization problem

so again gradient descent will reach a *global* optimum





proof: differentiate again to get the second derivative

Some issues about Gradient Descent

Learning algorithm:

- Initialize weights **w=0**
- For t=1,… until convergence:
    - Predict for each example $\boldsymbol{x}^i$ using **w:** $\hat{y}^i = \sum_{j=0}^{k} w_j \phi_j(\mathbf{x}^i)$
    - Compute gradient of loss: $\frac{\partial}{\partial w_j} J(\mathbf{w}) = 2\sum_i \left(y^i - \hat{y}^i\right) \phi_j(\mathbf{x}^i)$
    - This is a vector **g**
    - Update: **w = w** – λ**g,** λ is the learning rate.

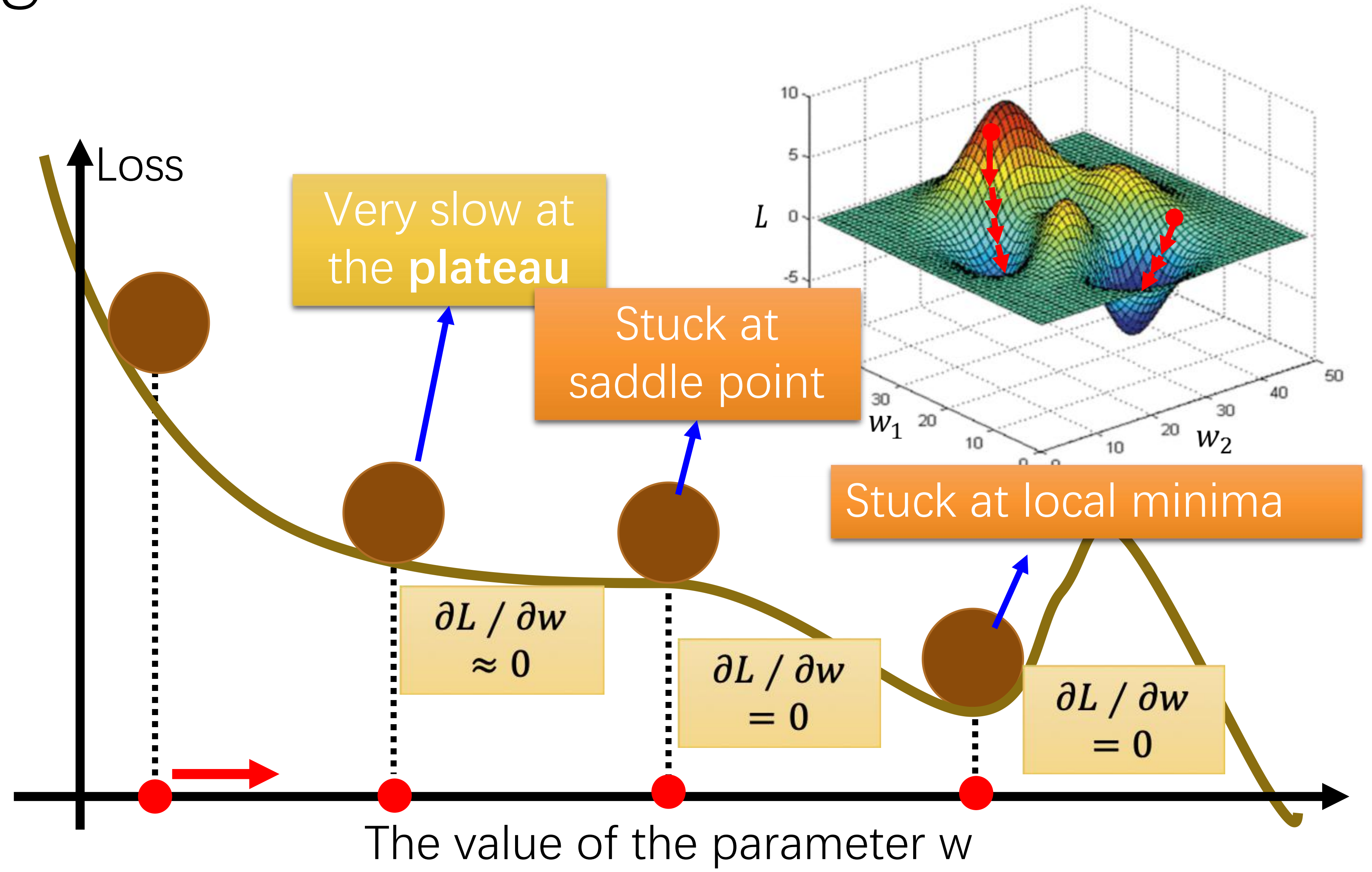
Initial status

Gradient computation

Weight update

Learning rate setting

Weight initialization status



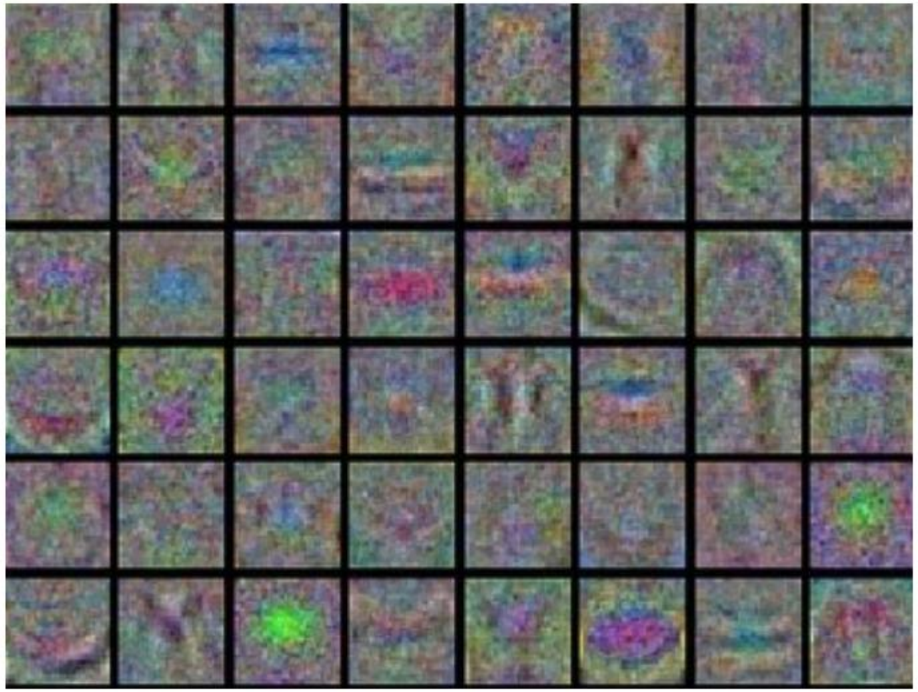
Weight initialization status

- An incorrect initialization can slow down or even completely stall the learning process. Luckily, this issue can be diagnosed relatively easily

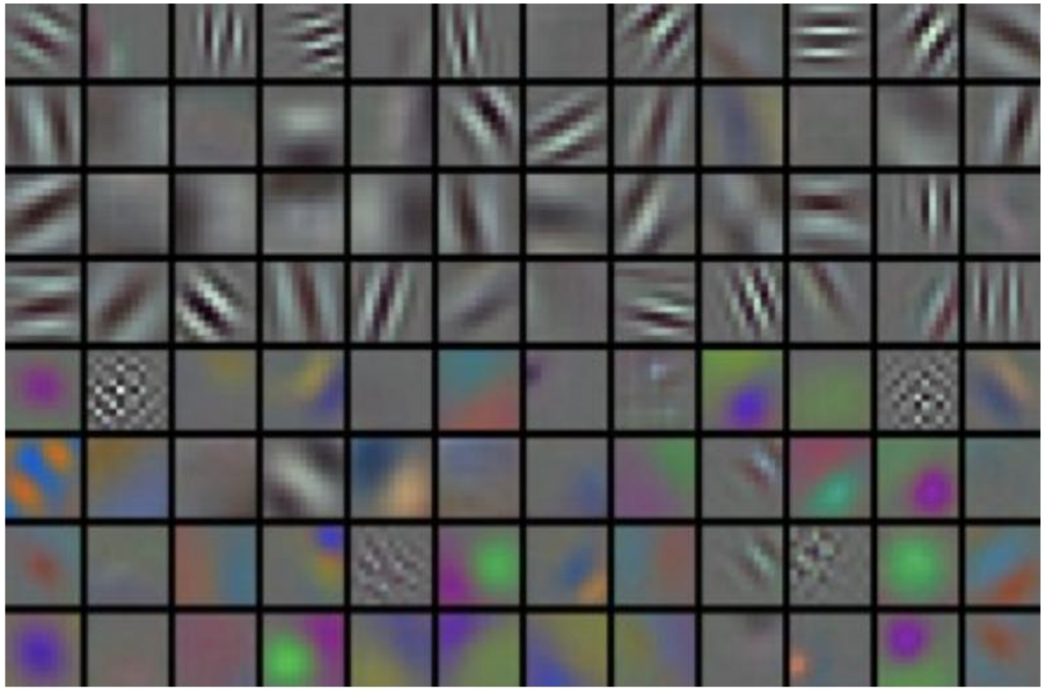
- One way to do so is to plot activation/gradient histograms for all layers of the network.

Weight initialization status

- First-layer Visualizations



Noisy features indicate could be a symptom: Unconverged network, improperly set learning rate, very low weight regularization penalty



Nice, smooth, clean and diverse features are a good indication that the training is proceeding well.

Gradient computation

- Gradient Checks
- In theory, performing a gradient check is as simple as comparing the analytic gradient to the numerical gradient.

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x)}{h} \quad \text{(bad, do not use)}$$

h is a very small number, in practice approximately 1e-5

$$\frac{df(x)}{dx} = \frac{f(x+h) - f(x-h)}{2h} \quad \text{(use instead)}$$

In practice, it turns out that it is much better to use the *centered* difference formula

Gradient computation:
Stochastic Gradient Descent

$$L = \sum_n \left(\hat{y}^n - \left(b + \sum w_i x_i^n \right) \right)^2$$

Loss is the summation over all training examples

- ***Gradient Descent*** $\theta^i = \theta^{i-1} - \eta \nabla L\left(\theta^{i-1}\right)$

- ***Stochastic Gradient Descent***

Faster!

Pick an example x^n

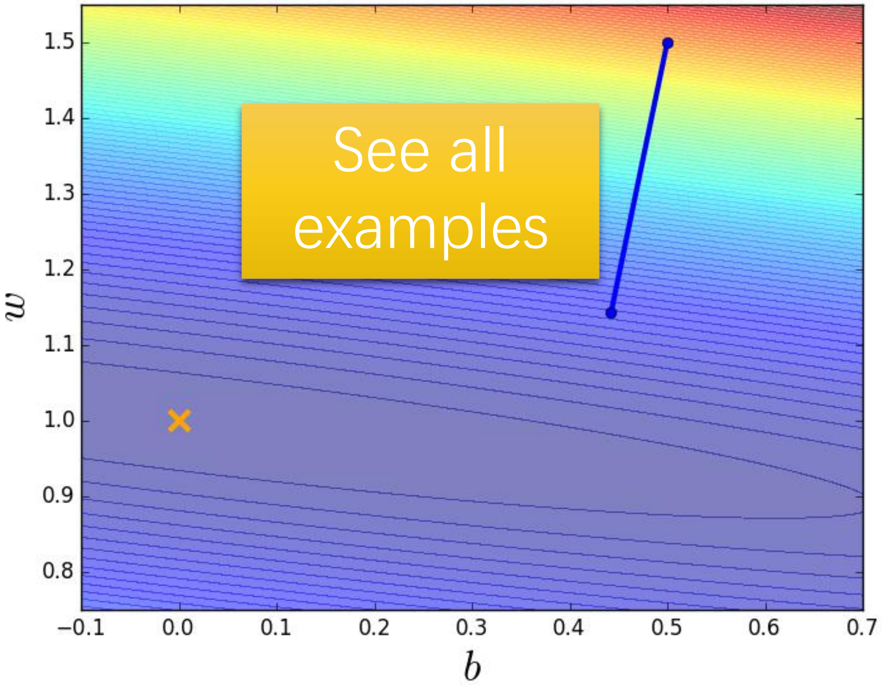
$$L^n = \left(\hat{y}^n - \left(b + \sum w_i x_i^n \right) \right)^2 \quad \theta^i = \theta^{i-1} - \eta \nabla L^n\left(\theta^{i-1}\right)$$

Loss for only one example

Gradient computation:
Stochastic Gradient Descent

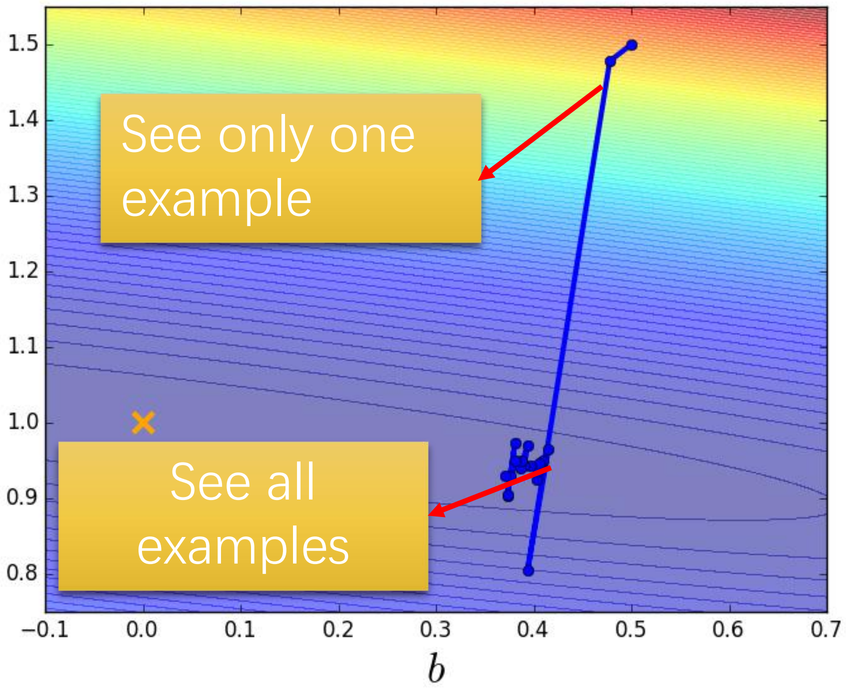
Gradient Descent

Update after seeing all examples



Stochastic Gradient Descent

Update for each example



If there are 20 examples, 20 times faster.

Gradient computation:
Stochastic Gradient Descent

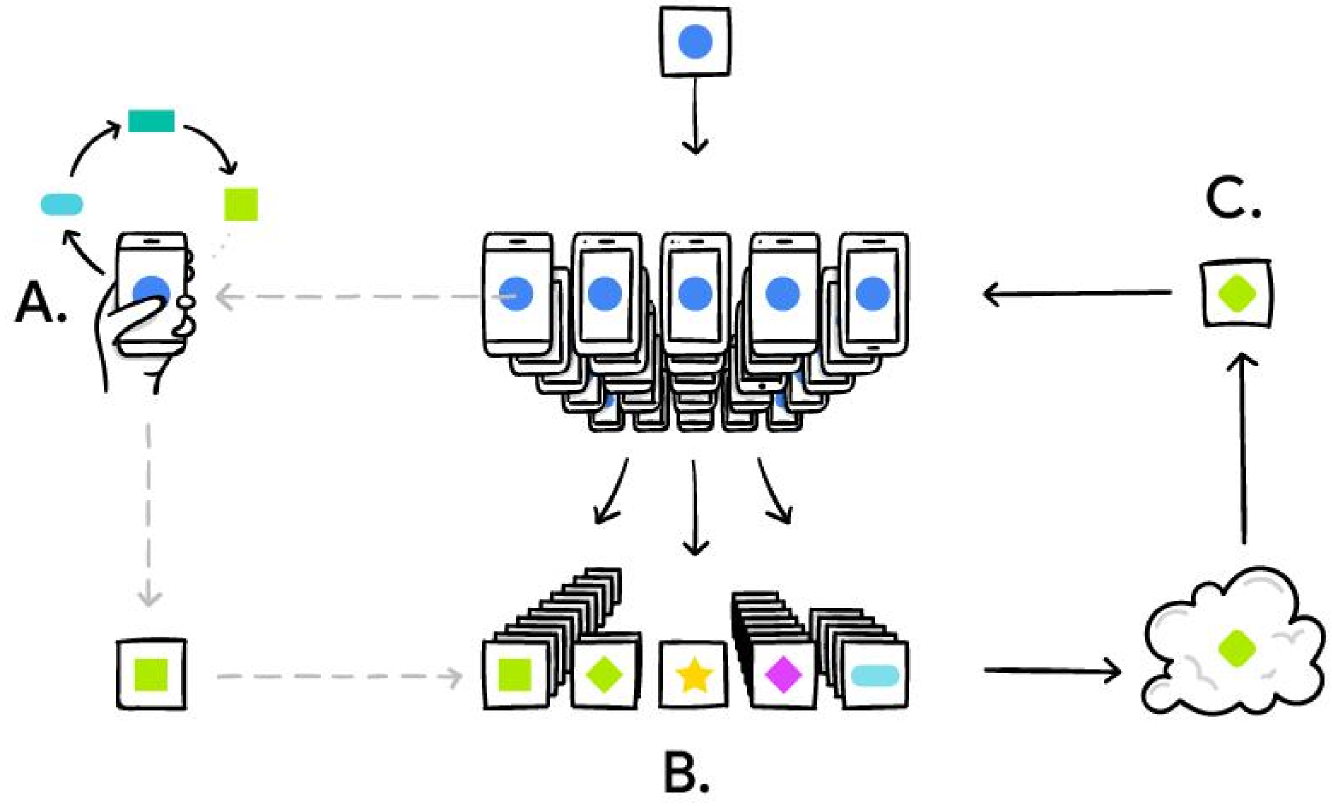
Gradient Descent

Stochastic Gradient Descent

Batch based Gradient Descent

Gradient computation:
Stochastic Gradient Descent

Tensorflow
Federated Learning



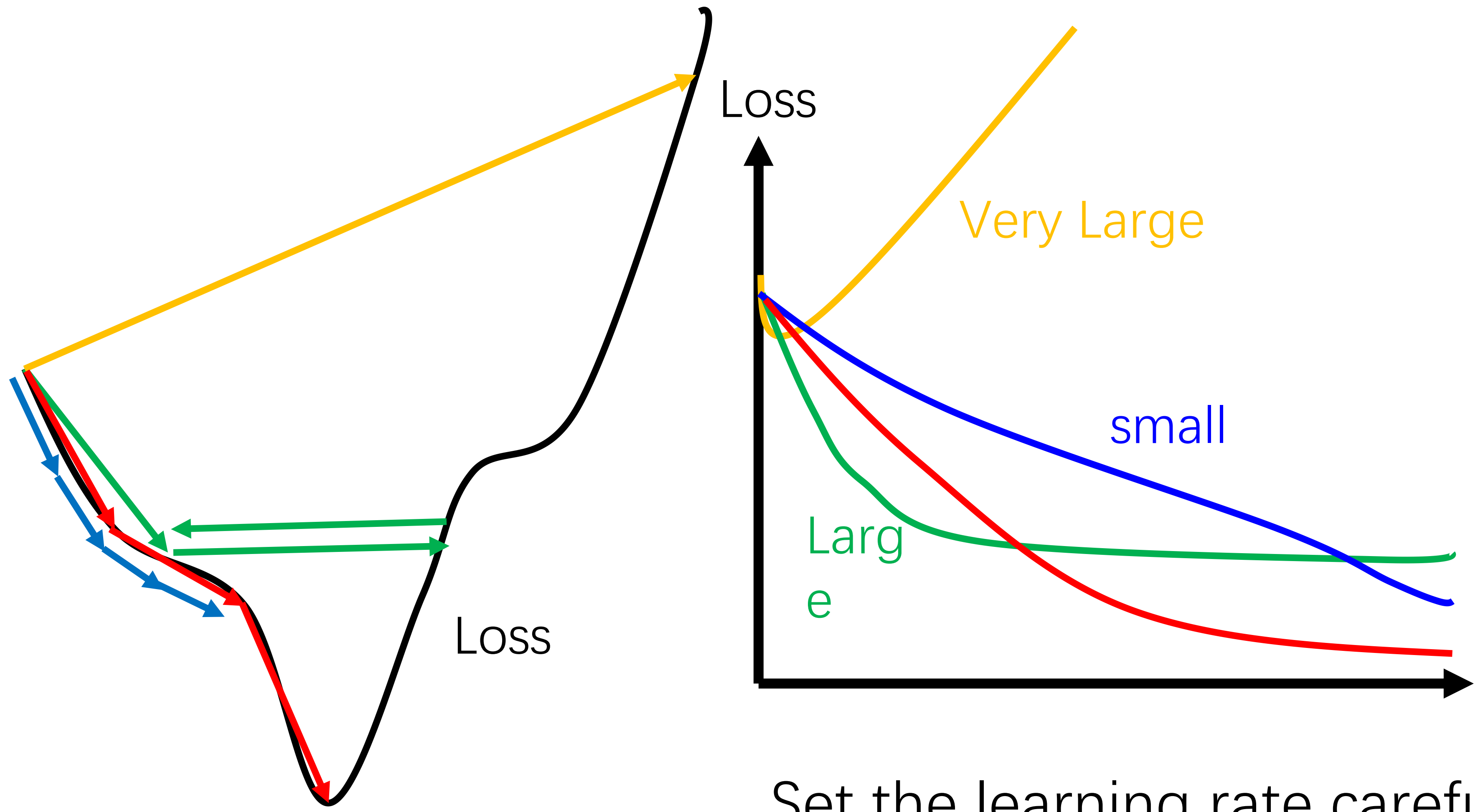
Weight Update

$$\mathbf{update:}\ \ \mathbf{w} = \mathbf{w} - \lambda\mathbf{g}$$

Learning rate setting

Update mode

Weight Update: learning rate

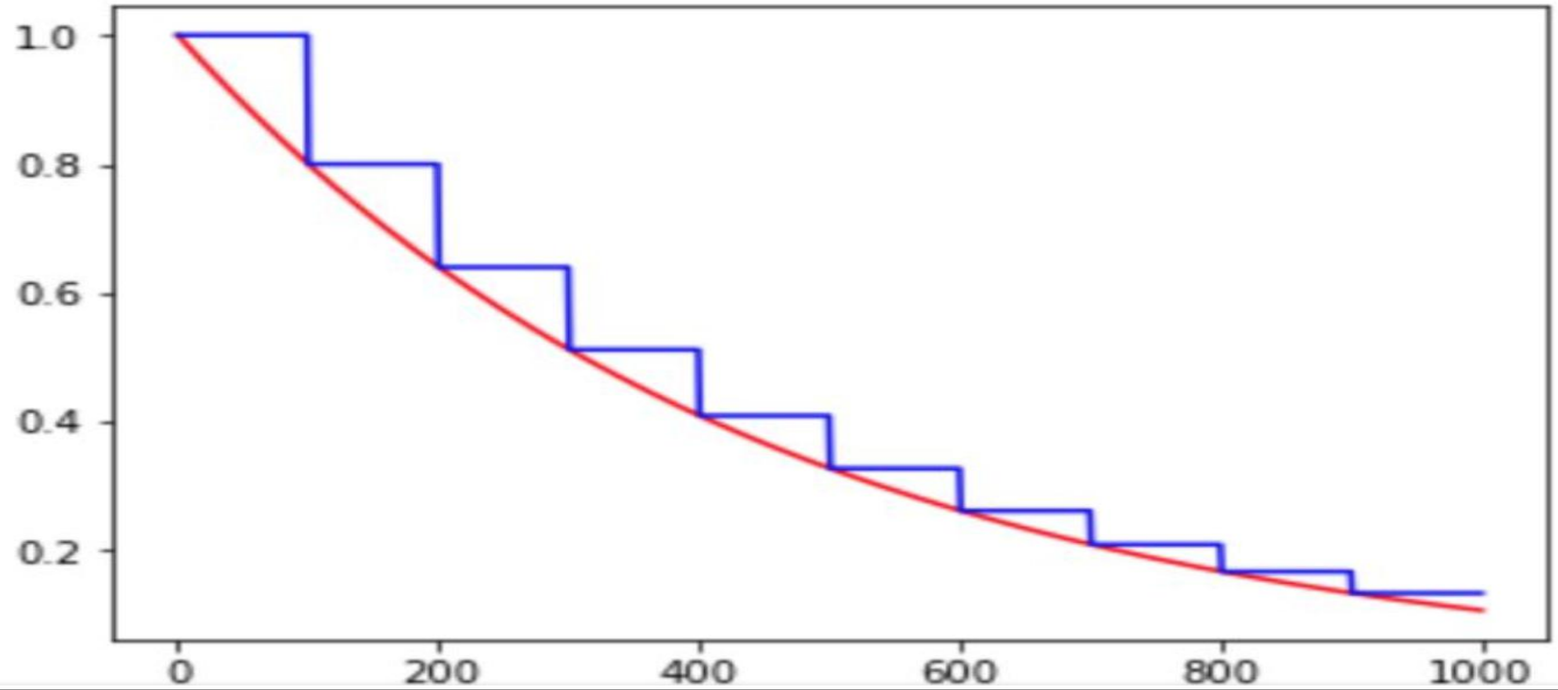


Set the learning rate carefully

Weight Update: decay learning rate

- (tensorflow/tensorflow/python/training/learning_rate_decay.py)
- exponential_decay

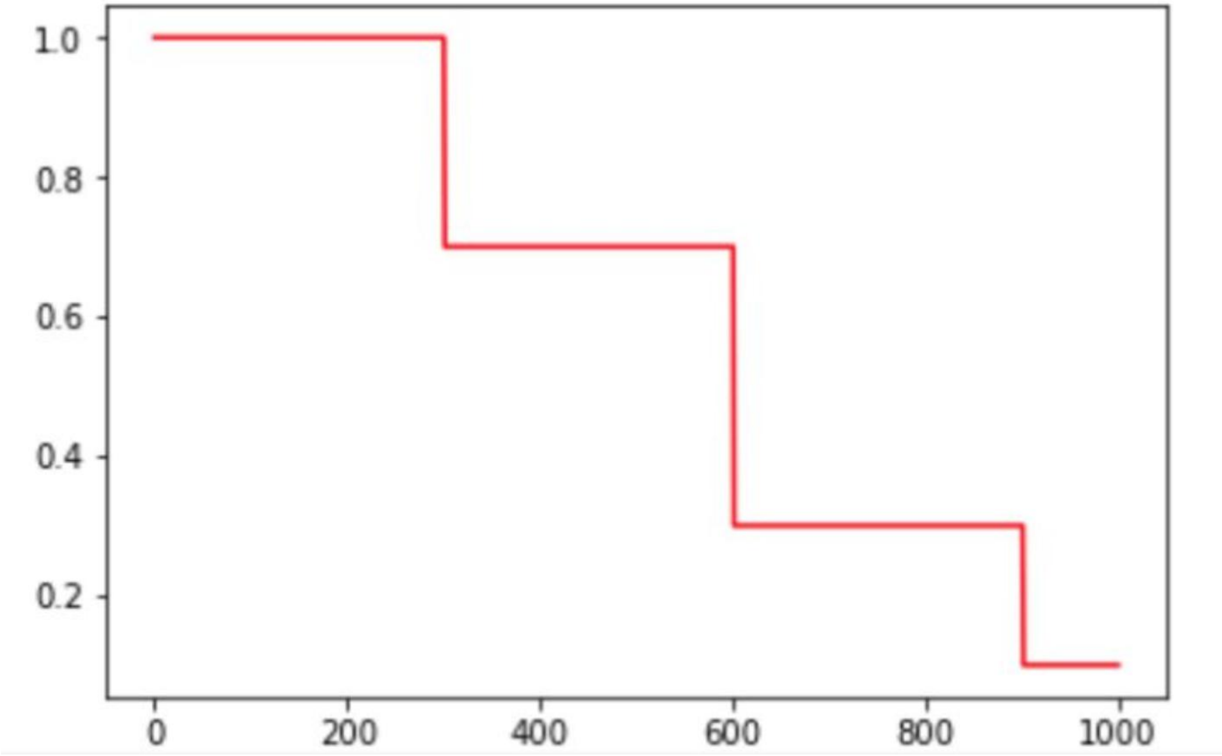
exponential_decay(learning_rate, global_step, decay_steps, decay_rate, staircase=False, name=None)



Weight Update: decay learning rate

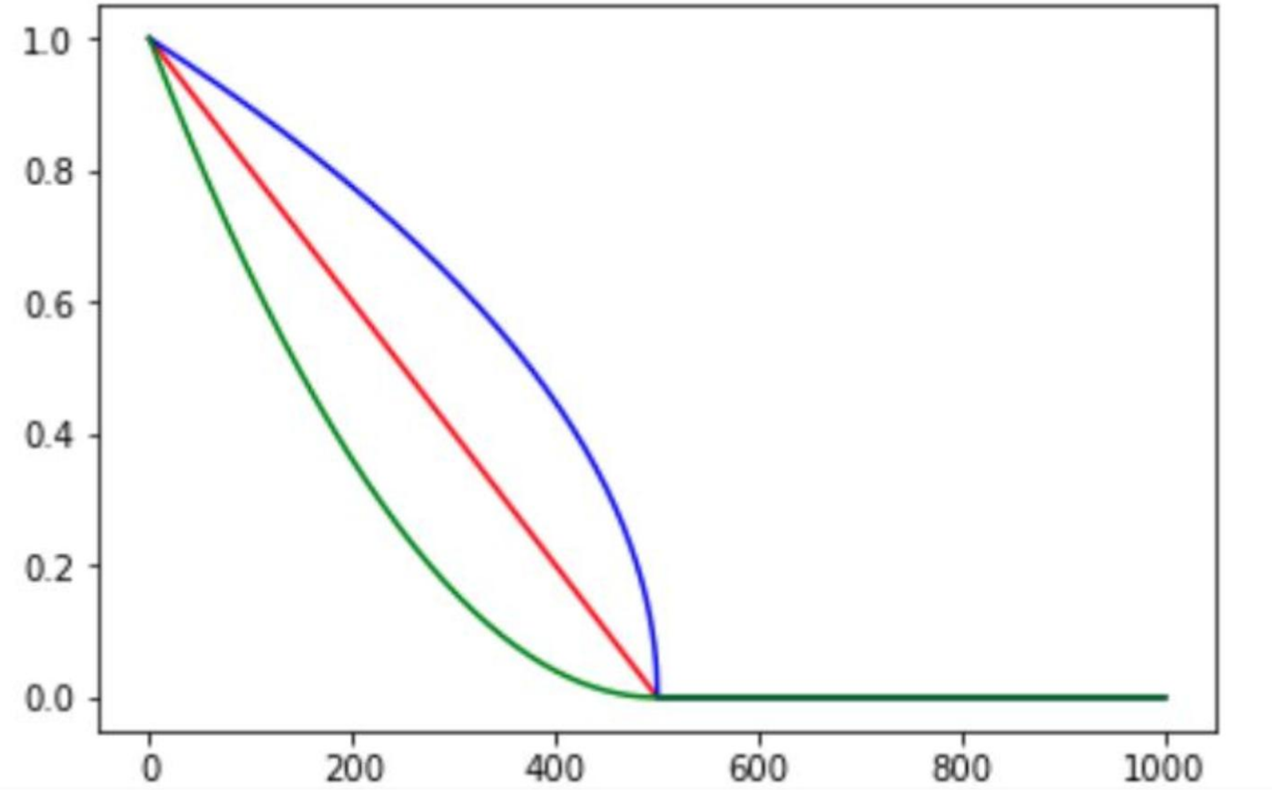
- piecewise_constant

  piecewise_constant(x, boundaries, values, name=None)



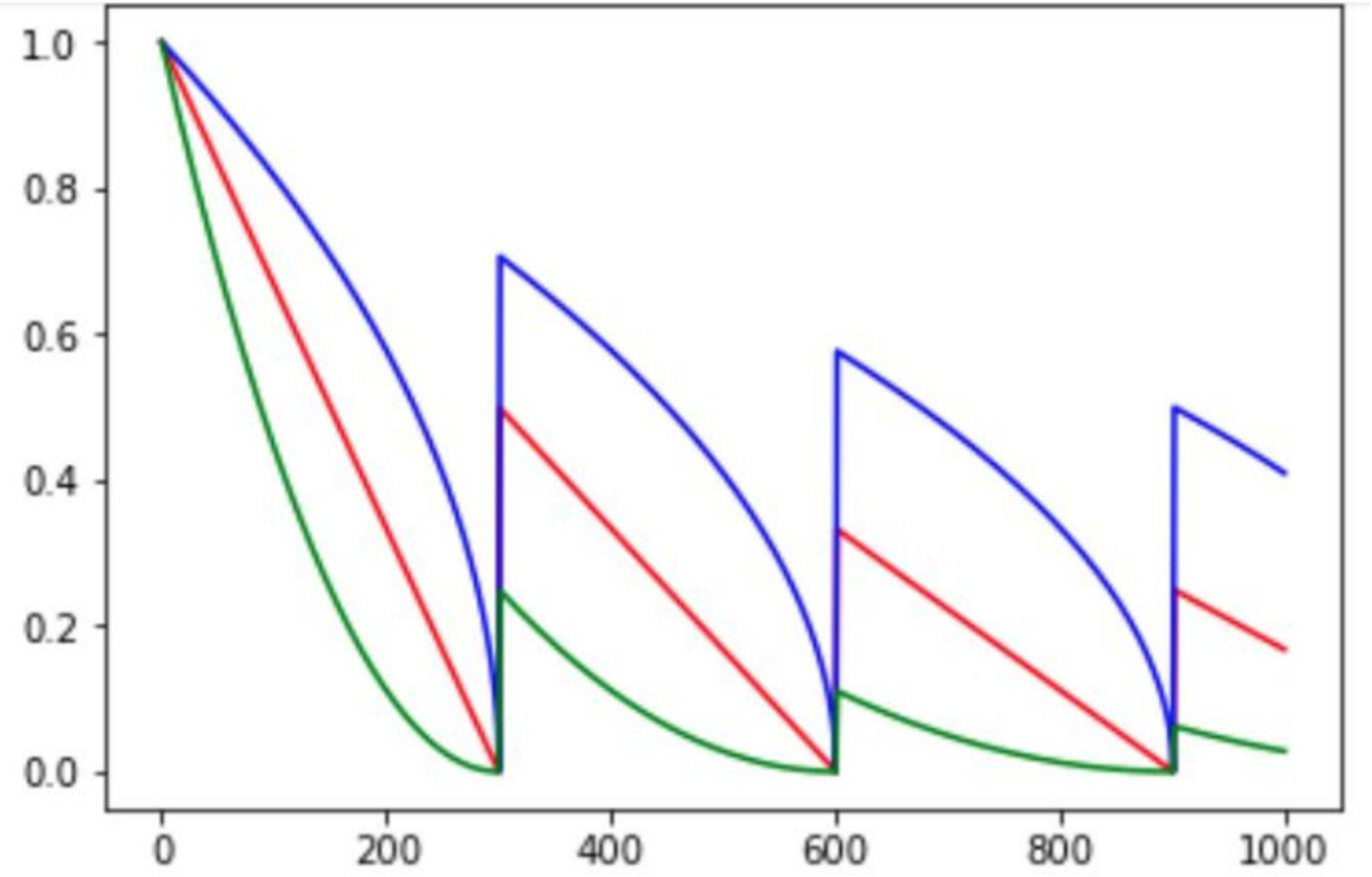
Weight Update: decay learning rate

- polynomial_decay
- polynomial_decay(learning_rate, global_step, decay_steps, end_learning_rate=0.0001, power=1.0, cycle=False, name=None)



Weight Update: decay learning rate

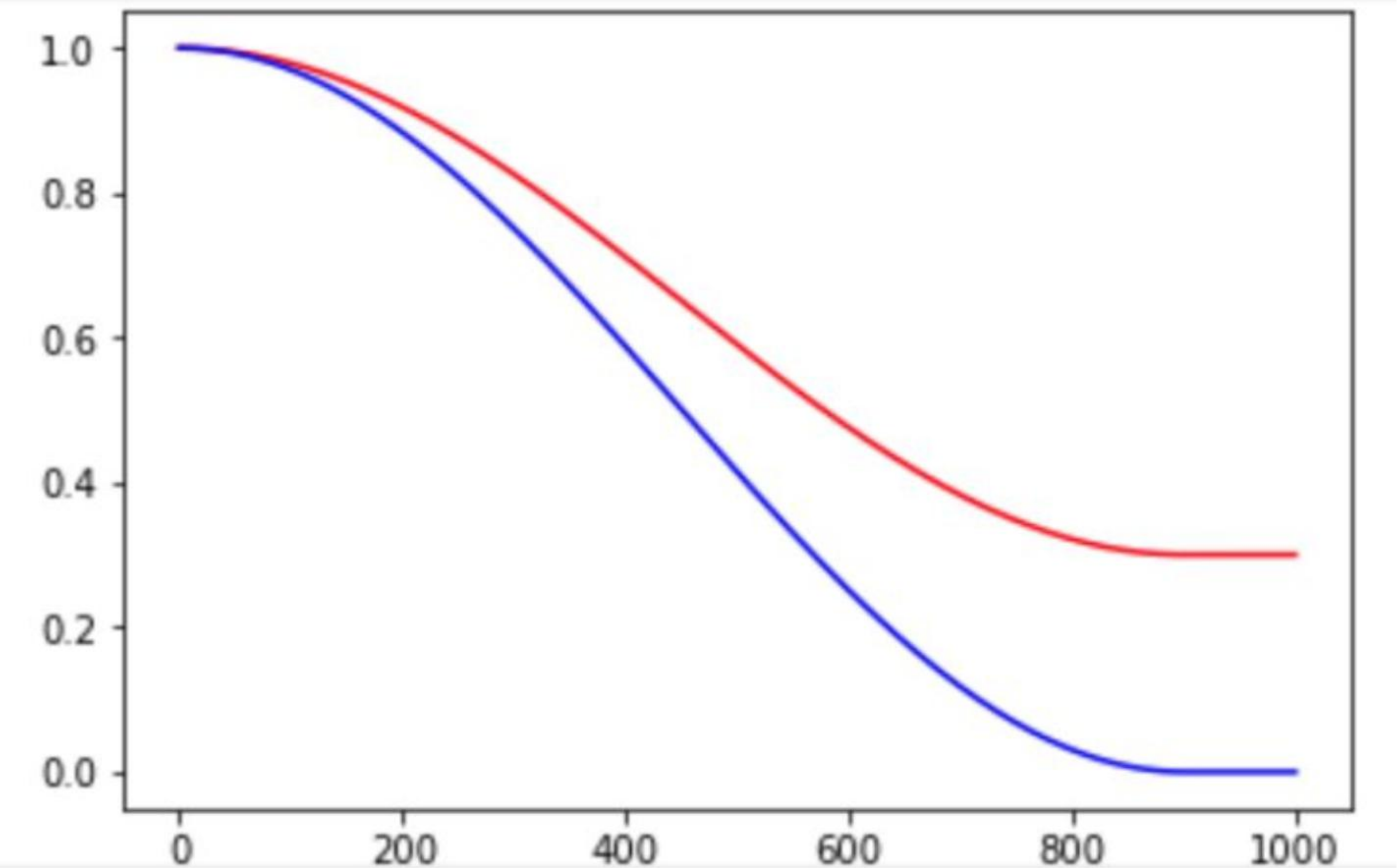
- Circle_decay



Skip local optimal

Weight Update: decay learning rate

- cosine_decay



Weight Update : **mode**

- Gradient descent :

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

- SGD

```
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
```

Weight Update : **mode**

- 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
```

  - reduces the variance of the parameter updates, which can lead to more stable convergence

  - can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries

Weight Update : **mode**

- Challenge
  - Choosing a proper learning rate can be difficult
  - Learning rate schedules try to adjust the learning rate during training is defined in advance and are thus unable to adapt to a dataset's characteristics
  - the same learning rate applies to all parameter updates.
  - difficulty arises in fact not from local minima but from saddle points

Gradient descent optimization algorithms

- Momentum
  - SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another

  - In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum.

Gradient descent optimization algorithms

- Momentum

Image 2: SGD without momentum

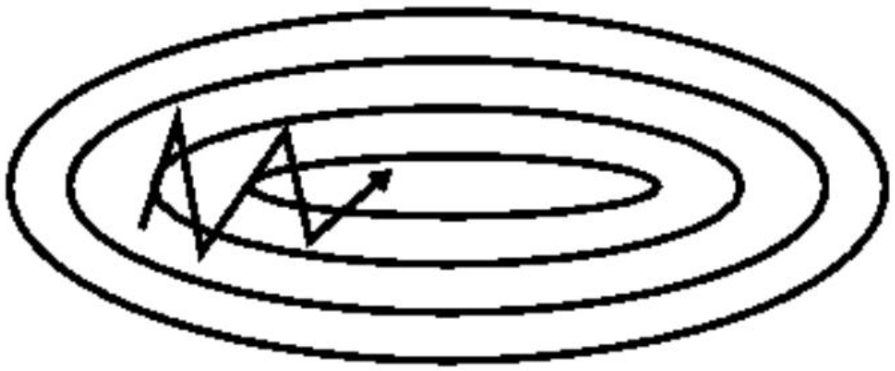


Image 3: SGD with momentum

Momentum is a method that helps accelerate SGD and dampens oscillations

Gradient descent optimization algorithms

- Momentum

Fraction = 0.9

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

Momentum

Last time update vector

If we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way

Why effective

- The momentum term increases for dimensions whose gradients point in the same directions
- reduces updates for dimensions whose gradients change directions.
- As a result, we gain faster convergence and reduced oscillation.

Gradient descent optimization algorithms

- Nesterov accelerated gradient (NAG)
  - a ball that rolls down a hill blindly, is highly unsatisfactory
  - We need **a smarter** ball, a ball that has a notion of where it is going to

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

Gradient w.r.t the current parameters

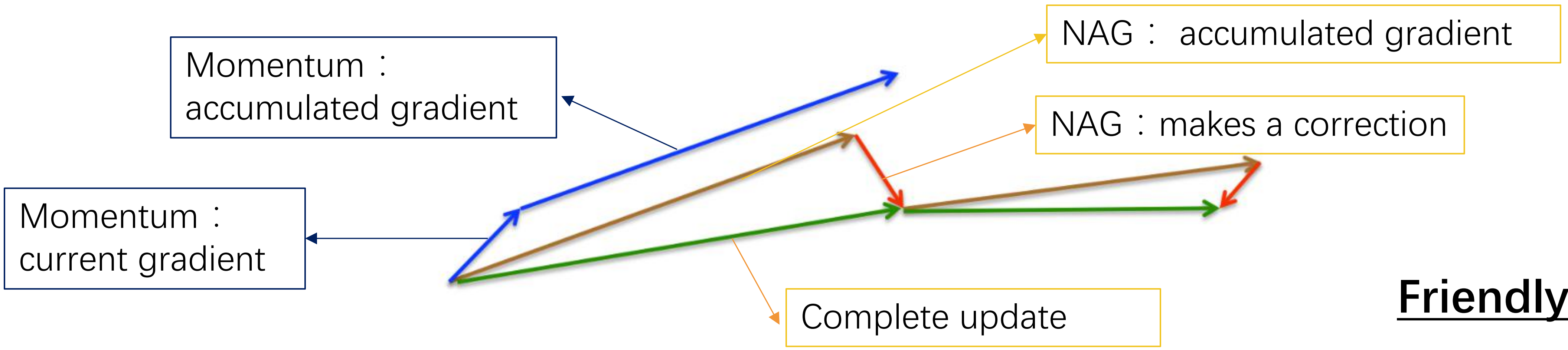
NAG → $$v_t = \gamma v_{t-1} + \eta \underline{\nabla_\theta J(\theta - \gamma v_{t-1})}$$
$$\theta = \theta - v_t$$

Gradient w.r.t approximate future position of our parameters

NAG can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters but w.r.t. the approximate future position of our parameters

Gradient descent optimization algorithms

- Nesterov accelerated gradient (NAG)
  - a ball that rolls down a hill blindly, is highly unsatisfactory
  - We need **a smarter** ball, a ball that has a notion of where it is going to



Friendly to RNN !!

NAG can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters but w.r.t. the approximate future position of our parameters

Gradient descent optimization algorithms

- Adagrad uses a different learning rate for every parameter at every time step
- Adagrad adapts the learning rate to the parameters,
  - **smaller updates (i.e. low learning rates**) for parameters associated with **frequently occurring features,**
  - **larger updates (i.e. high learning rates)** for parameters associated with **infrequent features**

Gradient descent optimization algorithms

SGD

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.$$

Adagrad

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}).$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

diagonal matrix where each diagonal element is the sum of the squares of the gradients w.r.t. up to time step t.

Gradient descent optimization algorithms

- adagrad
  - Benefits
    - it eliminates the need to manually tune the learning rate
  - Weakness:
    - the learning rate to shrink and eventually become infinitesimally small

Gradient descent optimization algorithms

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

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

m and ***v*** are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

counteract these biases

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.$$

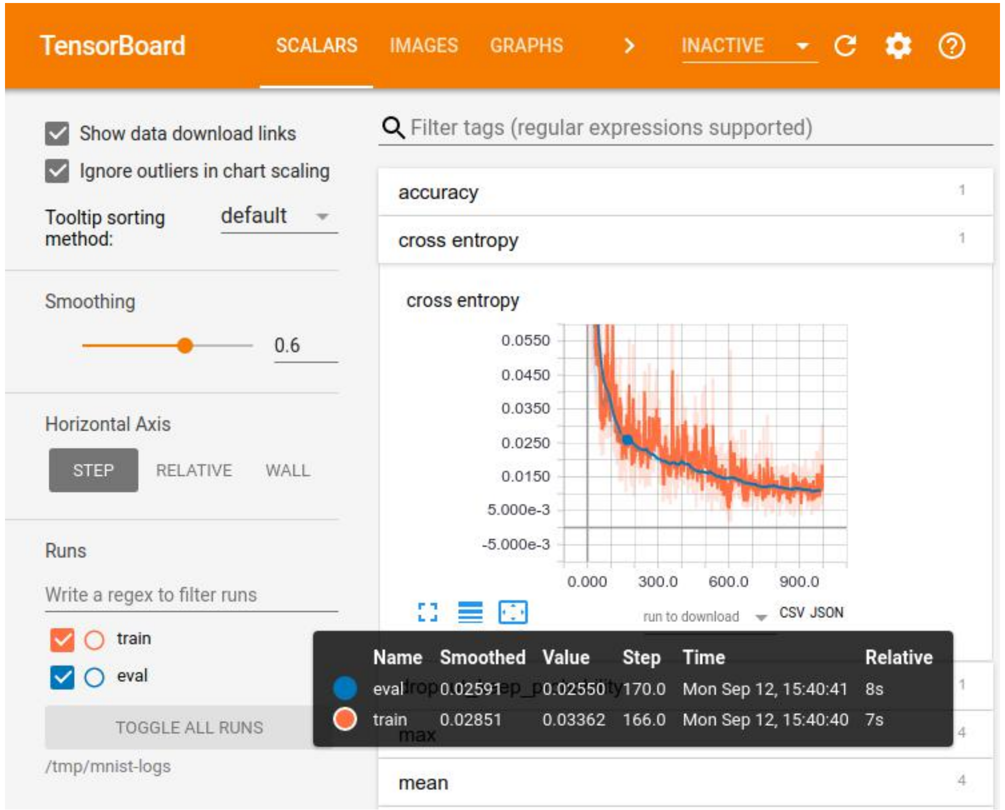
Final update！！！

Code example

Image classification in colab, and hyperparameter tuning using tensorboard

Example

- Tensorboard
  - Tensorboard visualization tools



- **Can also used in colab & google Clould**

- **New features: hyperparameter tuning in TF2.0**

2019
怪兽
学堂

THANKS