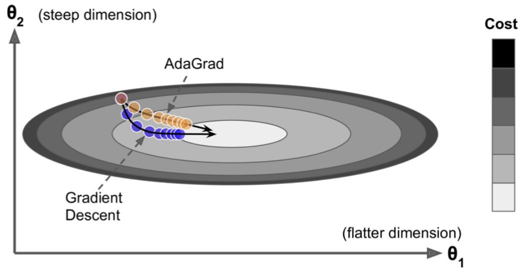
Optimizer

- Motivation
  - finds the optimize parameters(weights, learning rates...) that minimize the error(loss function), increasing the accuracy and training speed of the model.
- Gradient Descent(3 types)
  - Concepts Understanding
    - Learning rate: how much model weights should ne updated
    - Batch: the number of samples to be taken for updating model parameters
  - starts by quickly going down the steepest slope, then slowly goes down the bottom of the valley.
- **GSD with Momentum(GSDM)**
  - `Momentum` helps in faster convergence of the loss function
    - $m \leftarrow \beta m - \eta \delta_\theta L(\theta)$
    - $\theta \leftarrow \theta + m$
      - m: momentum vector
      - β: momentum, for avoiding the momentum going too large ∊ $[0, 1]$, normal set=0.9
  - learning rate is decreased with a high momentum term, because of the high oscillations
  - Downside:
    - momentum ↑, the possibility of passing the optimal minimum also increases.
    - result in poor accuracy and even more *oscillations*.
    - add another hyperparameter to tune
  - use exponential moving average of the gradients to update weights and bias, reduce the noise, and smoothen the data.
    - $w_t = w_{t-1} - \eta V_{dw_t}$
      - $V_{dw_t} = \beta V_{dw_{t-1}} + (1-\beta)\frac{\delta L}{\delta w_{t-1}}$
    - $b_t = b_{t-1} - \eta V_{db_t}$
      - $V_{db_t} = \beta V_{db_{t-1}} + (1-\beta)\frac{\delta L}{\delta b_{t-1}}$
        - L: loss function
  - Coding: `optimizer=keras.optimizer.SGD(lr=0.001,momentum=0.9)`
    - As results, $\beta = 0.9$, momentum optimization 10 times faster than gradient time* learning rate
- **Adam** (default)
  - updates the learning rate for each network parameter individually from estimates of first and second moments of the gradients.

- Adam=adagrad(works well on sparse gradients) + RMSProp(works well in online and nonstationary settings)
  - reduces the radically diminishing learning rates of AdaGrad
    - Reason
      - use exponential moving average of the gradients to scale the learning rate
      - while, adagrad use simple average method; RMSProp use exponential decaying sum.
- Functions
  - $m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$
  - $v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
    - m_t: the fist moment estimation
    - v_t: the second moment estimation
    - $\beta_1 and \beta_2 : [0,1)$ control the exponential decay rate of moving average.
    - g_t: gradient at time t along parameter w
  - $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
  - $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
    - bias corrected estimates<--initialization bias for moment estimation set to be 0
  - $\theta_{t+1} = \theta_t - \frac{\eta \hat{m}_t}{\sqrt[2]{\hat{v}_t + \epsilon}}$
- coding: `optimizer=keras.optimizers.Adam(lr=0.001,beta_1=0.9,beta_2=0.999)`
- Evaluation
  - pros
    - computationally efficient and has little memory requirement
    - recommended as a default optimization algorithm
  - cons
    - it focus on fast computation speed, in some situations, Adam leads to unconvergence, while some algorithms(SGD) focus on data point.
    - the low learning rate at final may lead to missing the optimum solution.

- **AdaGrad**(Adaptive Gradient)

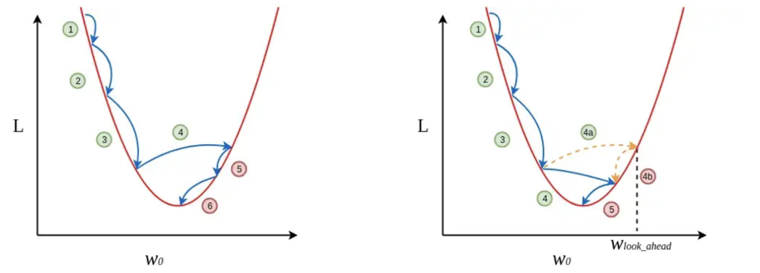


  - use different learning rates for parameters

- The more parameters get change, the more minor the learning rate changes. For sparse features, lr needed to be higher than that of dense features.
    - Because the occurrence frequency of sparse features is lower.
- Function of weights update
    - $w_t = w_{t-1} - \eta_t' \frac{\delta L}{\delta w_{t-1}}$
    - where $\eta_t' = \frac{\eta}{\sqrt[2]{\alpha_t + \epsilon}}$
        - α_t: learning rate at time t
        - η: inital learning rate
- Evaluation
    - downside
        - decrease α aggressively and monotonically, the model will stop learning too early, cuz the learning rate is almost close to 0, the accuracy of the model is getting down.
        - The inital learning rate should be decided manually
    - Pros
        - It is fitted for the reality that a dataset contains both sparse features and dense features

- **AdaDelta**
    - Without sum up the past squared gradients as AdaGrad, we restrict the window size, with exponential weighted average
    - Function: the same as AdaGrad, except replace α with <u>exponential weighted average of squared gradients.</u>
        - $\eta_t' = \frac{\eta}{\sqrt[2]{S_{dw_t} + \epsilon}}$
            - $S_{dw_t} = \beta S_{dw_{t-1}} + (1-\beta)(\frac{\delta L}{\delta w_{t-1}})^2$
            - typically $\beta = 0.9 \ or \ 0.95$
    - Evaluation
        - Pros
            - solve the radically diminishing learning rates.
- **Root Mean Square Propagation(RMSProp)**
    - RPPROP
        - solue the problem that some graidents are small while some are huge
        - use the <u>sign</u> of the gradient adapting the step size individually for each weight
            - Select two gradient and compare their sign
                - sign1=sign2→go in right direction, increase the step size
                - sign1 ≠ sign2→decrease the step size
        - Problem

        - does not work well for large dataset and mini-batch updates--> the purpose of RMSProp
    - Choose different learning rate for parameters
    - accelerating the optimization process by <u>decreasing the # of function evaluations</u> to reach local minimum
    - accumulating only the gradients from the most recent iterations
    - Functions (for weight w and bias b)
        - $v_t = \beta v_{t-1} + (1-\beta) g_t^2$
        - $w_{t+1} = w_t - \frac{\eta}{\sqrt[2]{v_t + \epsilon}} * g_t$
    - Evaluations
        - Pros
            - reduce the monotonically decreasing learning rate
            - quickly converge
            - require less tuning
        - Cons
            - the learning rates should be defined manually
    - Coding: `optimizer=keras.optimizers.RMSprop(lr=0.001,rho=0.9)`
- **NAG(Nesterov Accelerated Gradient)**



(a) Momentum-Based Gradient Descent (b) Nesterov Accelerated Gradient Descent

$\Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)}$ $\Longrightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$

    - Compared with momentum optimization method, the <u>direction</u> is different, it not at the local position, but at the direction of momentum direction(toward the optimum)
    - Function
        - $m \leftarrow \beta m - \eta \delta_\theta L(\theta + \beta m)$;the gradient is optimum at $\theta + \beta m$ not at θ
        - $\theta = \theta + m$
    - coding: `optimizer=keras.optimizer.SGD(lr=0.001,momentum=0.9,nesterov=True)`
    - Evulation
        - NAG ends up slight faster than momentum method
        - help reduce oscillations
- **Nadam**

- Converge slightly faster than Adam
- Nadam= Adam+Nesterov
- References:
  - A comprehensive guide on deep learning optimiziers.https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/