Carnegie Mellon University

Introduction to Deep Learning for Engineers

Spring 2025, Introduction to Deep Learning for Engineers
Jan 23, 2025, Fourth Session

Amir Barati Farimani
Associate Professor of Mechanical Engineering and Bio-Engineering
Carnegie Mellon University

Problem Setup

- Given a training set of input-output pairs

$(X_1, d_1), (X_2, d_2), \ldots, (X_r, d_r),$

- The error on the i-th instance is $div(Y_i, d_i)$
- The loss

$$Loss = \frac{1}{T} \sum_i div(Y_i, d_i)$$

- Minimize Loss w.r.t. $\{w_{ij}^{(k)}, b_j^{(k)}\}$

Recap: Gradient Descent Algorithm

- Initialize:
  -- x^0
  -- k = 0

- Do
  -- $x^{k+1} = x^k - \eta^k \nabla f(x^k)^r$
  -- k = k+1

- While $\left|f\left(x^k\right) - f(x^{k-1})\right| > \varepsilon$

Recap: Gradient Descent Algorithm

- In order to minimize any function f(x) w.r.t. x
- Initialize:

  -- x^0

  -- k = 0

- Do

  -- for every component i

  - $x_i^{k+1} = x_i^k - \eta^k \frac{df}{dx_i}$ Explicitly stating it by component

  -- k = k+1

- While $\left|f\left(x^k\right) - f(x^{k-1})\right| > \varepsilon$

Training Neural Nets through Gradient Descent

- Total training Loss:

$$Loss = \frac{1}{T}\sum_{t} Div(Y_t, d_t)$$

- Gradient descent algorithm:
- Initialize all weights and biases $\{w_{ij}^{(k)}\}$

  -- Using the extended notation: the bias is also a weight
- Do:

  -- For every layer k for all I,j, update:

$$w_{ij}^{(k)} = w_{ij}^{(k)} - \eta \frac{dLoss}{dw_{ij}^{(k)}}$$

- Until Loss has converged

Assuming the bias is also represented as a weight

Training Neural Nets through Gradient Descent

- Total training Loss:

$$Loss = \frac{1}{T}\sum_{t} Div(Y_t, d_t)$$

- Gradient descent algorithm:
- Initialize all weights $\{w_{ij}^{(k)}\}$
- Do:
  -- For every layer k for all i,j, update:

$$w_{ij}^{(k)} = w_{ij}^{(k)} - \eta \frac{dLoss}{dw_{ij}^{(k)}}$$

- Until Err has converged

The derivative

Total training Loss:

$$\boldsymbol{Loss} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$$

- Computing the derivative

Total derivative:

$$\frac{\boldsymbol{dLoss}}{\boldsymbol{dw_{ij}^{(k)}}} = \frac{\mathbf{1}}{\boldsymbol{T}} \sum_{\boldsymbol{t}} \frac{\boldsymbol{dDiv}(\boldsymbol{Y_t}, \boldsymbol{d_t})}{\boldsymbol{dw_{ij}^{(k)}}}$$

The derivative

Total training Loss:

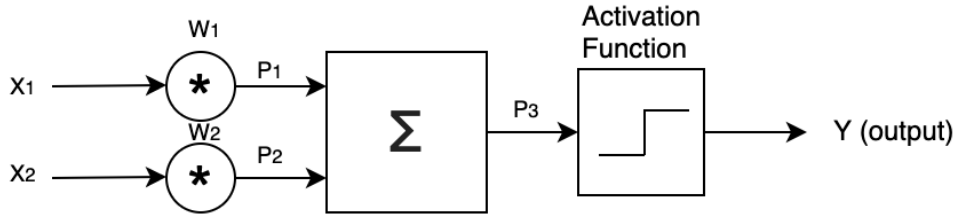
$$Loss = \frac{1}{T}\sum_{t} Div(Y_t, d_t)$$

Total derivative:

$$\frac{dLoss}{dw_{ij}^{(k)}} = \frac{1}{T}\sum_{t} \frac{dDiv(Y_t, d_t)}{dw_{ij}^{(k)}}$$

- So, we must first figure out how to compute the derivative of divergences of individual training inputs

Back Propagation

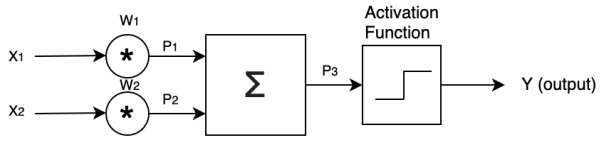


$$p_1 = w_1\ x_1$$
$$p_2 = w_2\ x_2$$
$$p_3 = p_1 + p_2 = w_1\ x_1 + w_2\ x_2$$

In order to obtain the partial derivative of cost function L with respect to weight w_1, we use chain rule to propagate the partial gradient of loss function with respect to outputs from the activation function and summation function until we arrive at w_1. Such expression can be denoted as:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial p_3} \cdot \frac{\partial p_3}{\partial p_1} \cdot \frac{\partial p_1}{\partial w_1} \tag{7}$$

Back Propagation



In order to obtain the partial derivative of cost function L with respect to weight w_1, we use chain rule to propagate the partial gradient of loss function with respect to outputs from the activation function and summation function until we arrive at w_1. Such expression can be denoted as:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial p_3} \cdot \frac{\partial p_3}{\partial p_1} \cdot \frac{\partial p_1}{\partial w_1} \tag{7}$$

If we use least squares for loss function and sigmoid function for activation function, then the terms in the chain rule expression can be derived to be:

$$\frac{\partial L}{\partial y} = (y - T) \quad \frac{\partial y}{\partial p_3} = \hat{g}(p_3) = g(p_3)(1 - g(p_3)) \tag{8}$$

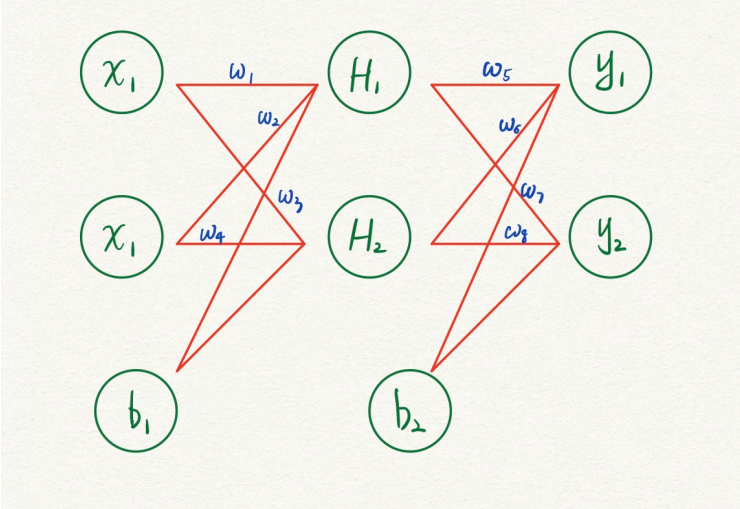
$$\frac{\partial p_3}{\partial p_1} = 1 \qquad \frac{\partial p_1}{\partial w_1} = x_1 \tag{9}$$

With the above building blocks, the partial derivatives of loss function with respect to the two weights w_1 and w_2 are:

$$\frac{\partial L}{\partial w_1} = (y - T)g(p_3)(1 - g(p_3)x_1 \quad \frac{\partial L}{\partial w_2} = (y - T)g(p_3)(1 - g(p_3)x_2 \tag{10}$$

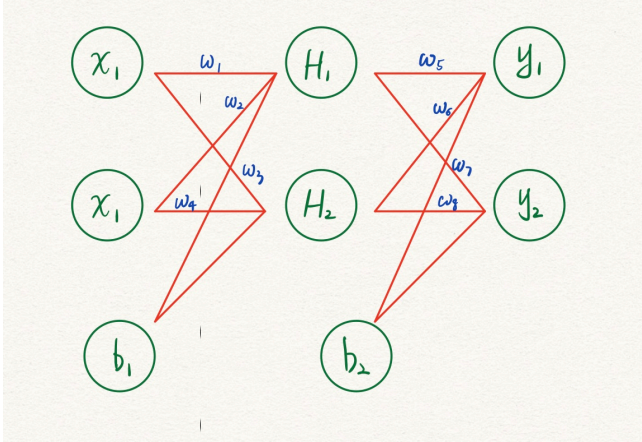
Back Propagation (Example)

Consider a neural network that has a one input layer, one hidden layer and one output layer with 2 neurons in each layer, and has 2 bias neurons connected to hidden layer and output layer. The schematic is shown in fig.7. The activation function is sigmoid. Data and initial weights are given as follows:$x_1 = 0.05$, $x_2 = 0.10$, $b_1 = 0.35$, $b_2 = 0.60$, $T_1 = 0.01$, $T_2 = 0.99$; $w_1 = 0.15$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.3$, $w_5 = 0.4$, $w_6 = 0.45$, $w_7 = 0.5$, $w_8 = 0.56$, learning rate $\alpha_1 = 0.15$.



Back Propagation (Example)

The activation function is sigmoid. Data and initial weights are given as follows:$x_1 = 0.05$, $x_2 = 0.10$, $b_1 = 0.35$, $b_2 = 0.60$, $T_1 = 0.01$, $T_2 = 0.99$; $w_1 = 0.15$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.3$, $w_5 = 0.4$, $w_6 = 0.45$, $w_7 = 0.5$, $w_8 = 0.56$, learning rate $\alpha_1 = 0.15$.



$$H_1 = w_1 x_1 + w_2 x_2 + b_1 = 0.3775$$

$$H_{1out} = \frac{1}{1 + \exp -H_1} = 0.59326$$

H_{2out} can be calculated the same way and $H_{2out} = 0.59688$. Now calculate y_1 as follows:

$$y_1 = H_{1out} \times w_5 + H_{2out} \times w_6 + b_2 = 1.059$$

$$y_{1out} = \frac{1}{1 + \exp -y_1} = 0.75136 \qquad y_{2out} = 0.77293$$

$$E_{total} = \sum \frac{1}{2}(T - y_{out})^2$$

To update the weights, we should back propagate the errors to the weights. The update rule is equation(6).

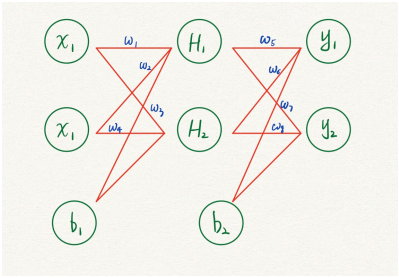
For w_5:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial y_{1out}} \cdot \frac{\partial y_{1out}}{\partial y_1} \cdot \frac{\partial y_1}{\partial w_5}$$

Carnegie Mellon University

Back Propagation (Example)

The activation function is sigmoid. Data and initial weights are given as follows:$x_1 = 0.05$, $x_2 = 0.10$, $b_1 = 0.35$, $b_2 = 0.60$, $T_1 = 0.01$, $T_2 = 0.99$; $w_1 = 0.15$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.3$, $w_5 = 0.4$, $w_6 = 0.45$, $w_7 = 0.5$, $w_8 = 0.56$, learning rate $\alpha_1 = 0.15$.



$$\frac{\partial E_{total}}{\partial w_5} = 2 \times \frac{1}{2} \times (T_1 - y_{1out}) \times (-1) \times y_{1out}(1 - y_{1out}) \times H_{1out} = 0.082187 \quad (17)$$

Then the weight w_5 can be updated as:

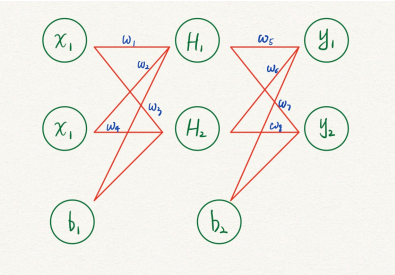
$$w_{5new} := w_{5old} - \alpha \frac{\partial E_{total}}{\partial w_5} = 0.35891 \quad (18)$$

w_6, w_7, w_8 can be updated the same way and equal to 0.40866, 0.511301, 0.06137 respectively.

As for weights between the input layer and the hidden layer, take w_1 for instance: There is two path from output layer to node H_1, we need to take both the effects into consideration. That is to say, when calculate the derivative

Back Propagation (Example)

The activation function is sigmoid. Data and initial weights are given as follows:$x_1 = 0.05$, $x_2 = 0.10$, $b_1 = 0.35$, $b_2 = 0.60$, $T_1 = 0.01$, $T_2 = 0.99$; $w_1 = 0.15$, $w_2 = 0.2$, $w_3 = 0.25$, $w_4 = 0.3$, $w_5 = 0.4$, $w_6 = 0.45$, $w_7 = 0.5$, $w_8 = 0.56$, learning rate $\alpha_1 = 0.15$.



of loss function of w_1, it has two terms:
from y_1:

$$\frac{\partial E_{total}}{\partial w_1}_{w_5} = \frac{\partial E_{total}}{\partial y_{1out}} \cdot \frac{\partial y_{1out}}{\partial y_1} \cdot \frac{\partial y_1}{\partial H_{1out}} \cdot \frac{\partial H_{1out}}{\partial H_1} \cdot \frac{\partial H_1}{\partial w_1} \tag{19}$$

from y_2:

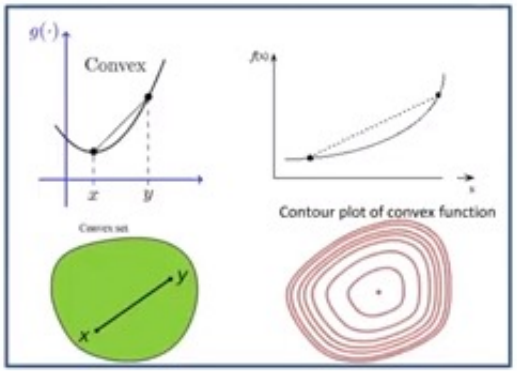
$$\frac{\partial E_{total}}{\partial w_1}_{w_7} = \frac{\partial E_{total}}{\partial y_{2out}} \cdot \frac{\partial y_{2out}}{\partial y_2} \cdot \frac{\partial y_2}{\partial H_{1out}} \cdot \frac{\partial H_{1out}}{\partial H_1} \cdot \frac{\partial H_1}{\partial w_1} \tag{20}$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial w_1}_{w_5} + \frac{\partial E_{total}}{\partial w_1}_{w_7} = 0.000438 \tag{21}$$

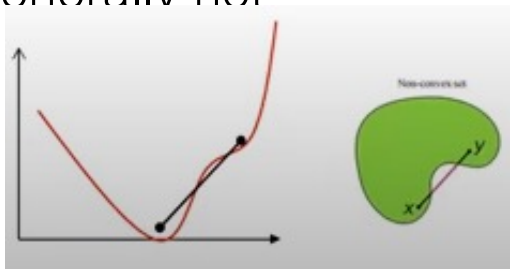
And then w_1 is updated substitute eq(21) to eq(6):$w_1 = 0.14978$. Through similar calculation,$w_2 = 0.19956$, $w_3 = 0.29975$, $w_4 = 0.29950$. Through the example above, we now have basic concept of how forward and back propagation works in a neural network.

Convex Loss Function

- A surface is “convex” if it is continuously curving upward
  -- We can connect any two points on or above the surface without intersecting it
  -- Many mathematical definitions that are equivalent



- **Caveat**: Neural network loss surface is generally not convex
  -- Streetlight effect

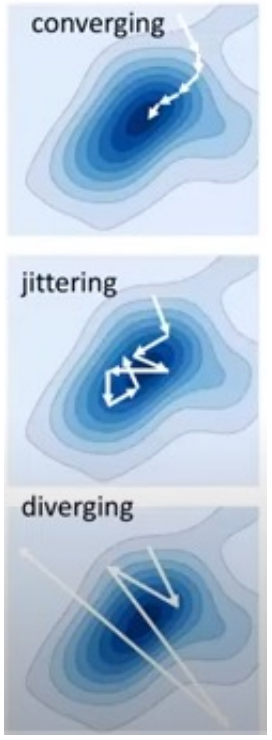


Convergence of gradient descent

- An iterative algorithm is said to converge to a solution if the value updates arrives at a fixed point
  -- Where the gradient is 0 and further updates do not change the estimate

- The algorithm may not actually converge
  -- It may jitter around the local minimum
  -- It may even diverge

- Conditions for convergence?



Batch, Stochastic, Minibatch

Batch gradient descent
Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters w for the entire training dataset.

Stochastic gradient descent
Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example x(i) and label y(i)

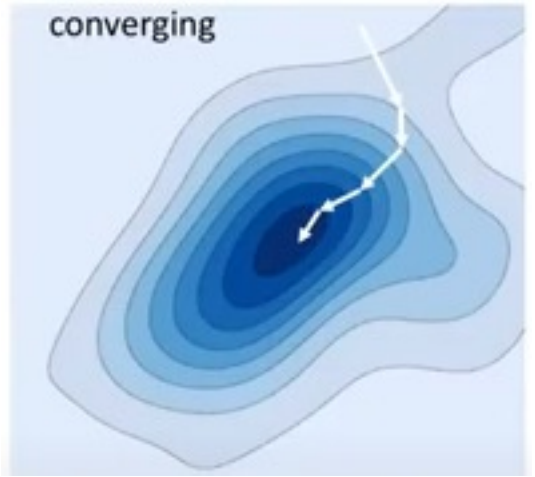
Mini-batch gradient descent
Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples.

Convergence and convergence rate

- Convergence rate: How fast the iterations arrive at the solution
- Generally quantified as

$$R = \frac{\left|f\left(x^{(k+1)}\right) - f(x^*)\right|}{\left|f(x^{(k)}) - f(x^*)\right|}$$

-- x^{k+1} is the k-th iteration
-- x^* is the optimal value of x



- If R is a constant (or upper bounded), the convergence is linear
  -- In reality, its arriving at the solution exponentially fast

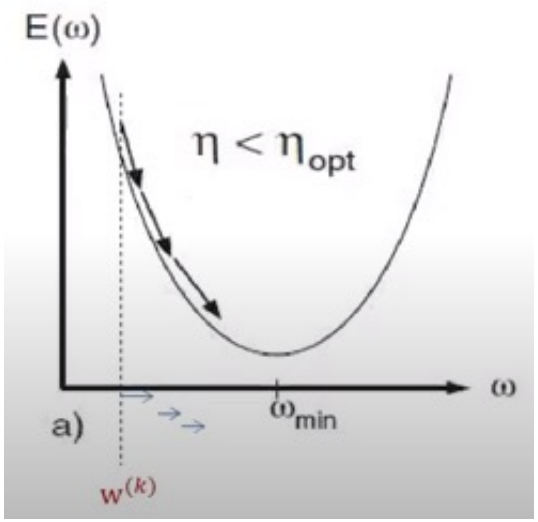
$$\left|f\left(x^{(k)}\right) - f(x^*)\right| \leq R^k \left|f\left(x^{(k)}\right) - f(x^*)\right|$$

Convergence for quadratic surfaces

$$Minimize\ E = \frac{1}{2}aw^2 + bw + c$$

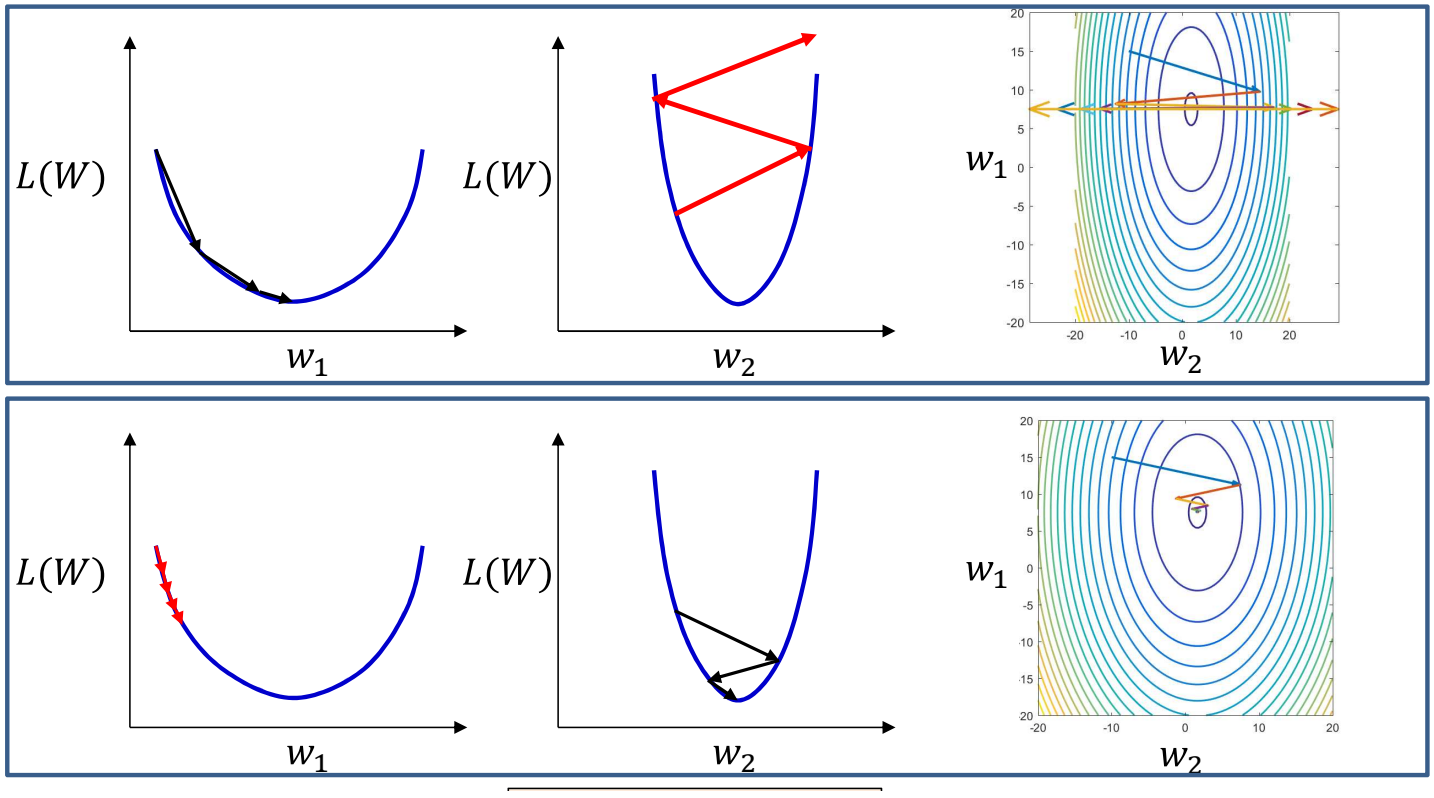
$$w^{(k+1)} = w^{(k)} - \eta \frac{dE(w^{(k)})}{dw}$$

Gradient descent with fixed step size η to estimate scalar parameter w



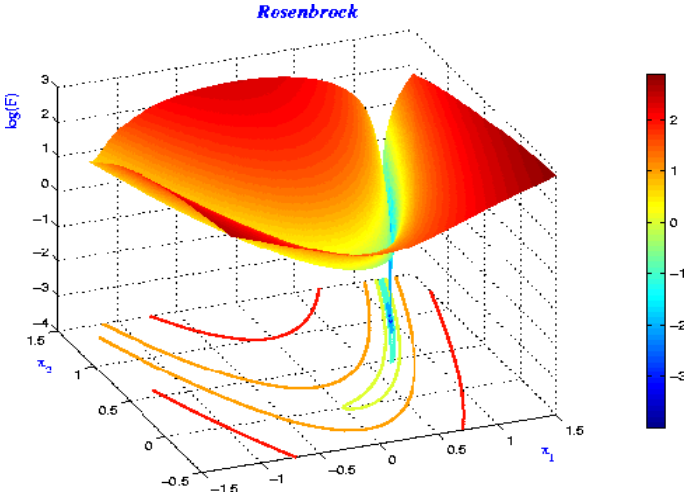
- Gradient descent to find the optimum of a quadratic, starting from $w^{(k)}$
- Assuming fixed step size η
- What is the optimal step size η to get there fastest?

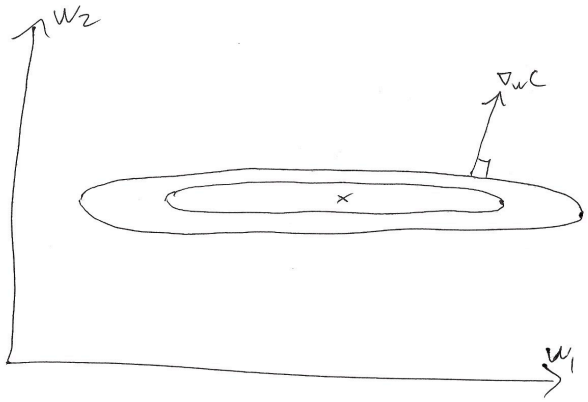
Eccentricity with other dimensions



Eccentricity with other dimensions

Long, narrow ravines:



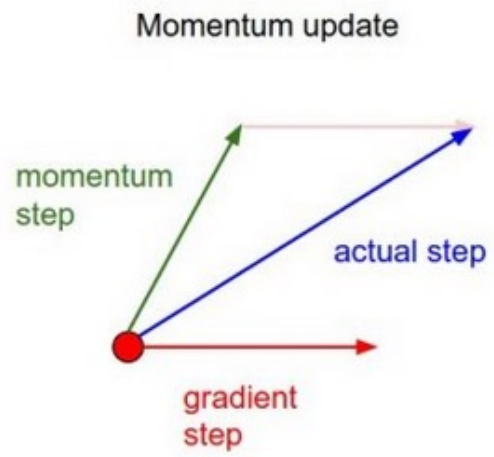


Lots of sloshing around the walls, only a small derivative along the slope of the ravine's floor.

Momentum Method

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

This scenario is common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along



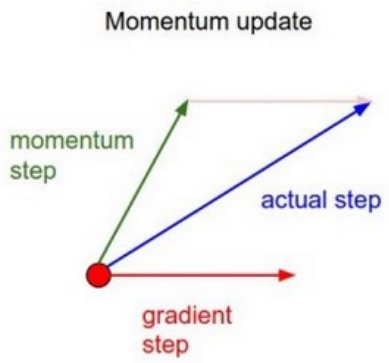
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

Momentum Method

In the high curvature directions, the gradients cancel each other out, so momentum dampens the oscillations.

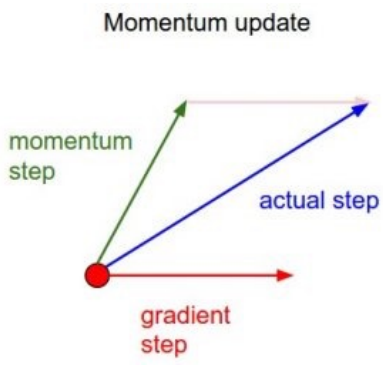
In the low curvature directions, the gradients point in the same direction, allowing the parameters to pick up speed.
If the gradient is constant (i.e. the cost surface is a plane), the parameters will reach a terminal velocity of

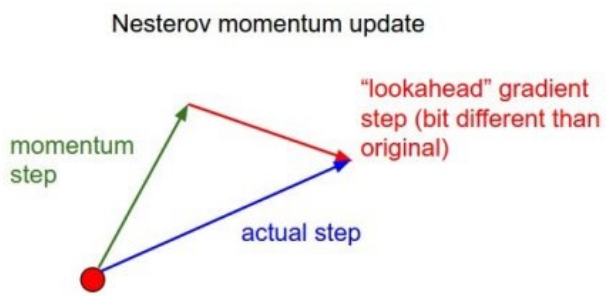


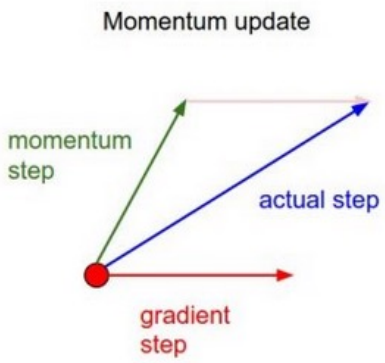
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

$$\theta = \theta - v_t$$

Nestrov Momentum







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

Second order methods

Training with momentum

- Initialize all weights $W_1, W_2, \ldots, W_K$
- Do:

  -- For all layer k, initialize $\nabla_{W_K} Loss = 0, \Delta W_K = 0$

  -- For all t=1:T

  - For every layer k:

    -- Compute gradient $\nabla_{W_K} \boldsymbol{Dive}\,(Y_t, d_t)$

    -- $\nabla_{W_K} \boldsymbol{Loss} += \frac{1}{T} \nabla_{W_K} \boldsymbol{Dive}\,(Y_t, d_t)$

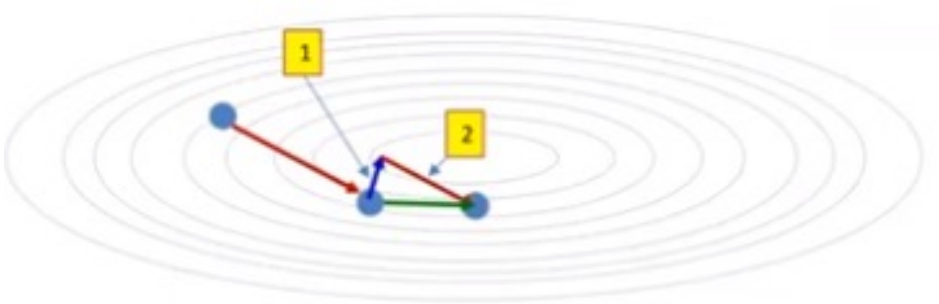
  -- For every layer k

$$\Delta w_K = \beta\, \Delta w_K - \eta\, (\nabla_{W_K} \boldsymbol{Loss})^T$$

$$W_K = W_K + \Delta W_K$$

- Until **Loss** has converged

Momentum Update



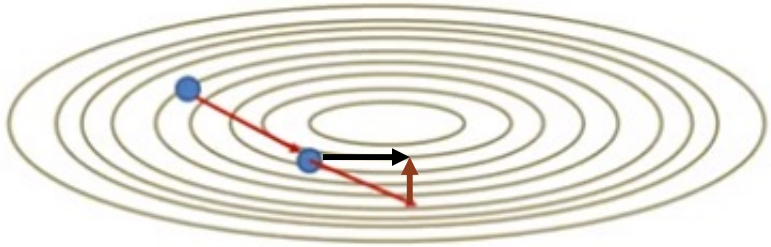
- Momentum update steps are actually computed in two stages

  -- First: We take a step against the gradient at the current location

  -- Second: Then we add a scaled version of the previous step

- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



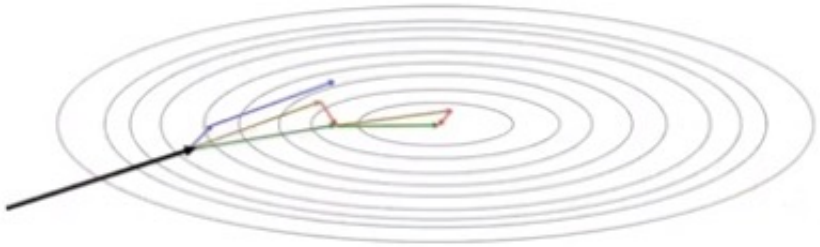
- Change the order of operations
- At any iteration, to compute the current step:

  -- First extend the previous step

  -- Then compute the gradient step at the resultant position

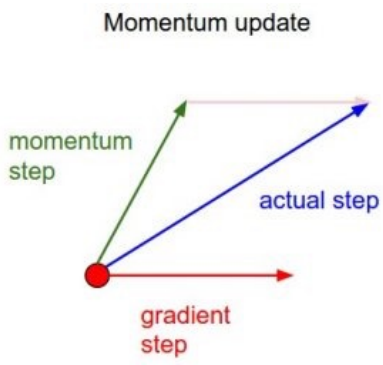
  -- Add the two obtain the final step

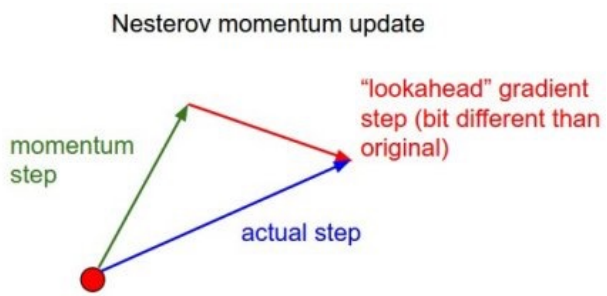
Nestorov's Accelerated Gradient

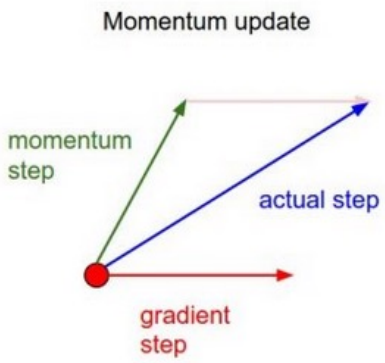


- Comparison with momentum (example from Hinton)
- Converges much faster

Nestrov Momentum



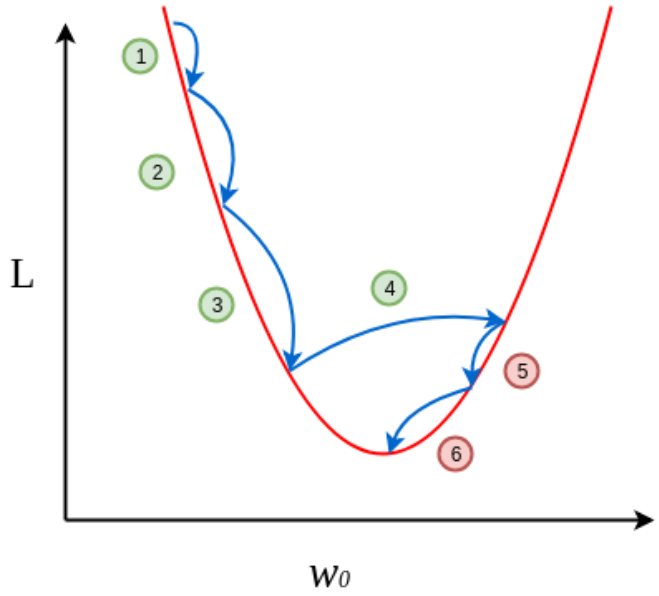




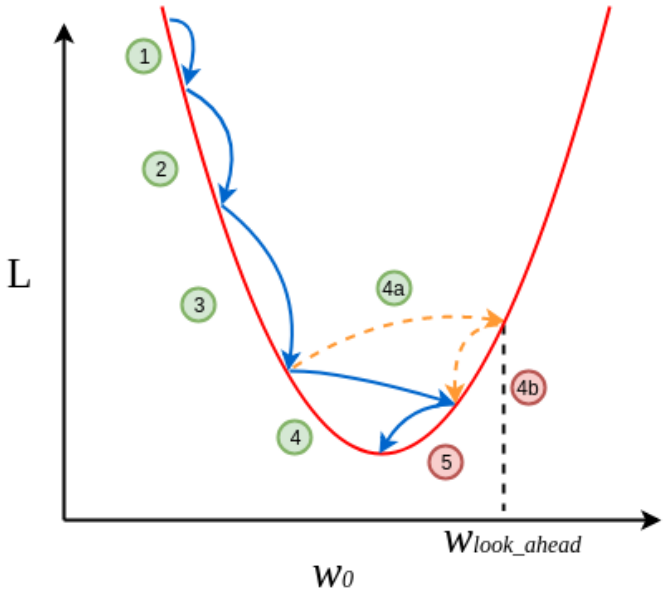
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Carnegie Mellon University

Comparison



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

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

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

AdaGrad

- Individually adapts learning rates of parameters
  - By scaling them inversely proportional to the sum of the historical squared values of the gradient
- The AdaGrad Algorithm:

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Performs well for some but not all deep learning

RMSProp

- Modifies AdaGrad for a nonconvex setting
  - Change gradient accumulation into exponentially weighted moving average
  - Converges rapidly when applied to convex function

The RMSProp Algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Carnegie Mellon University

RMSProp combined with Nesterov

Algorithm: RMSProp with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

RMSProp is popular

- RMSProp is an effective practical optimization algorithm
- Go-to optimization method for deep learning practitioners

Adam: Adaptive Moments

- Yet another adaptive learning rate optimization algorithm
- Variant of RMSProp with momentum
- Generally robust to the choice of hyperparameters

The Adam Update Rule

The Adam Algorithm

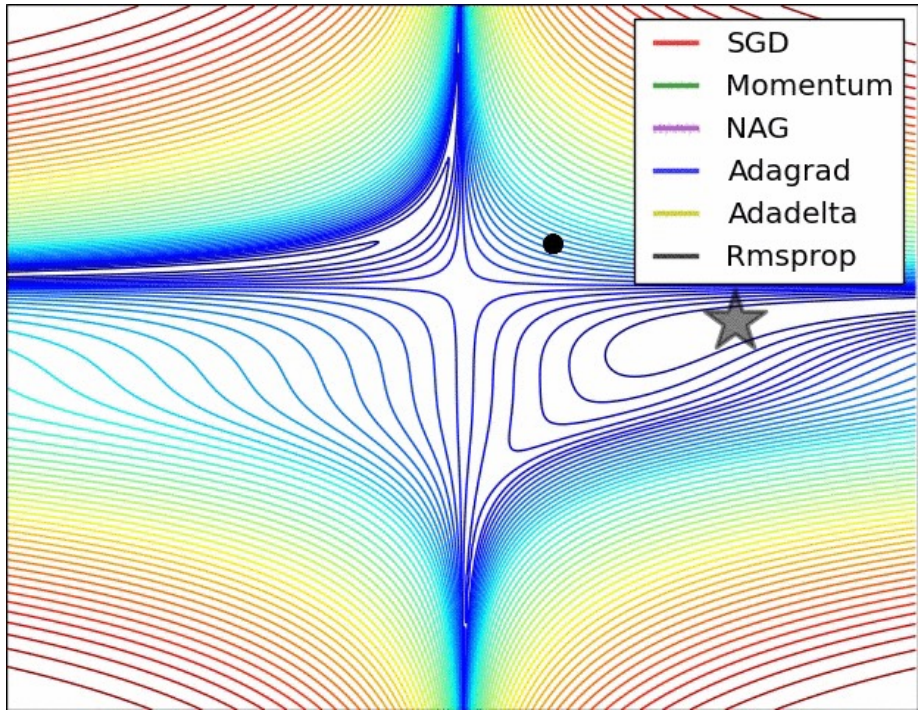
Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

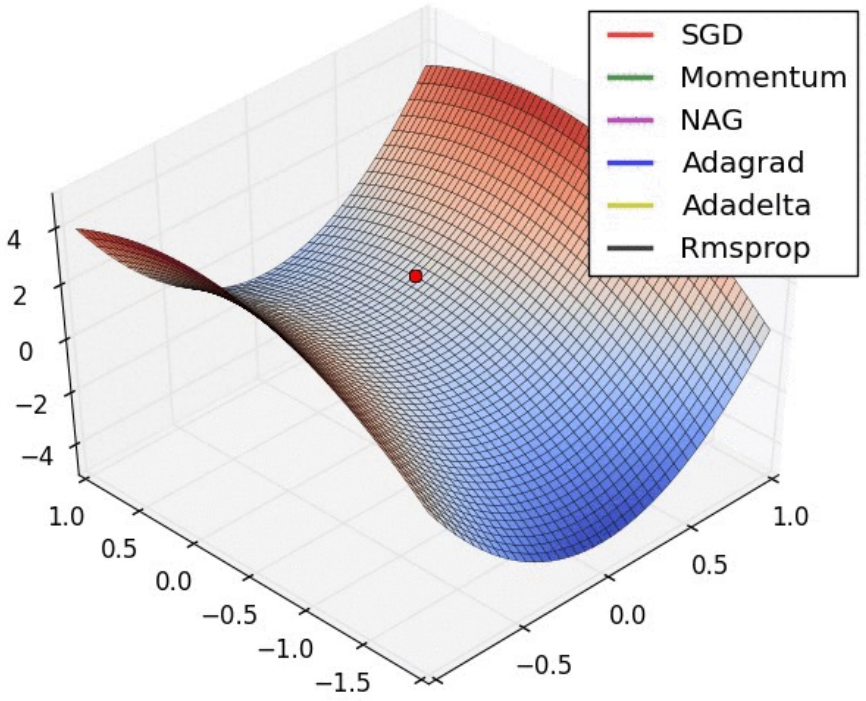
Carnegie Mellon University

Choosing the Right Update Rule

- We have discussed several methods of optimizing deep models by adapting the learning rate for each model parameter
- Which algorithm to choose?
  - There is no consensus
- Most popular algorithms actively in use:
  - SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam
  - Choice depends on user's familiarity with algorithm

Race of the Optimizers!





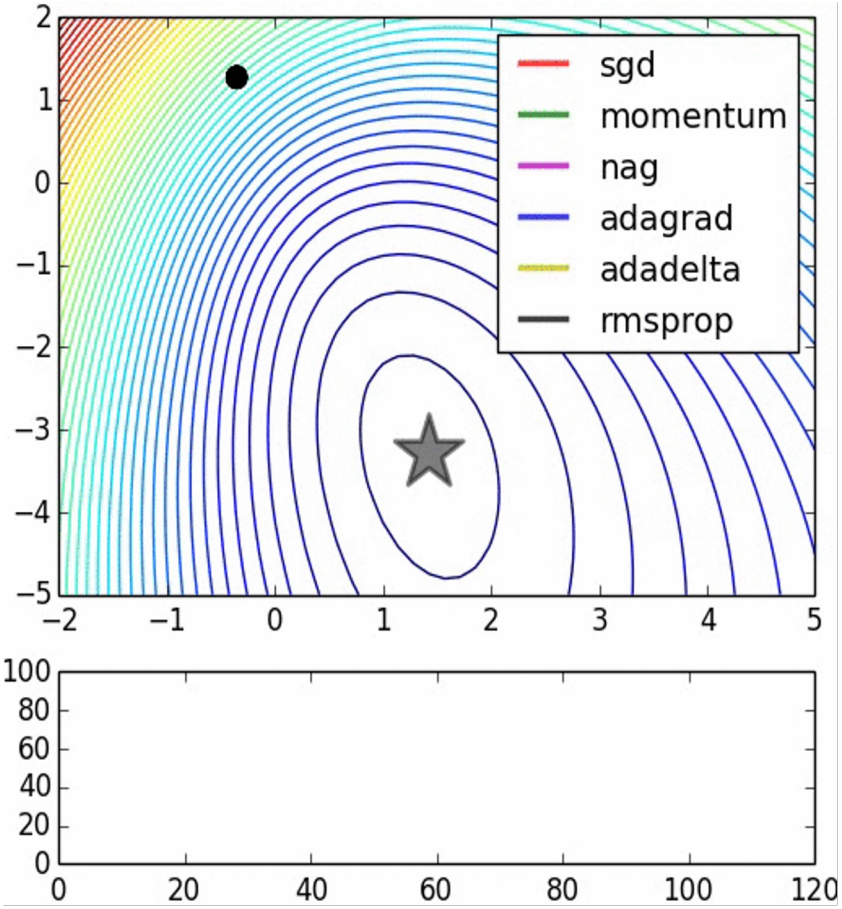
http://cs231n.github.io/neural-networks-3/#hyper

Carnegie Mellon University

Summary

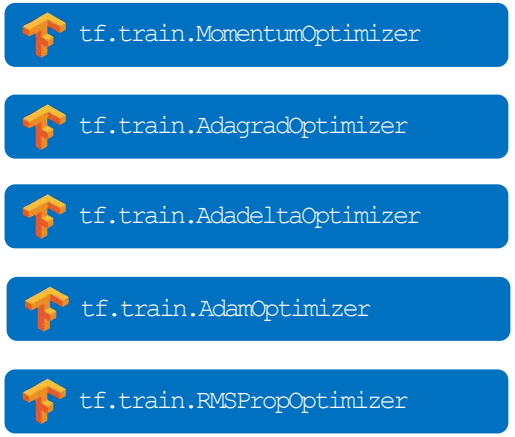
- **Simple Gradient Methods** like SGD can make adequate progress to an optimum when used on minibatches of data.
- **Second-order** methods make much better progress toward the goal, but are more expensive and unstable.
- **Convergence rates:** quadratic, linear, O(1/n).
- **Momentum:** is another method to produce better effective gradients.
- ADAGRAD, RMSprop diagonally scale the gradient. ADAM scales and applies momentum.

Key points



Adaptive Learning Rate Algorithm

- Momentum
- Adagrad
- Adadelta
- Adam
- RMSProp



Qian et al. "On the momentum term in gradient descent learning algorithms." 1999.

Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.

Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.

Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.

Additional details: http://ruder.io/optimizing-gradient-descent/