Neural Networks Training

Alexander Trofimov
PhD, professor, NRNU MEPHI

lab@neuroinfo.ru
http://datalearning.ru

Course “Neural Networks”

March 2020

Neural Network Training Problem

Given:

$\mathscr{D} = \{(x^{(1)}, \sigma^{(1)}), ..., (x^{(n)}, \sigma^{(n)})\}$ — available data sample

$x^{(i)} = \left(x_1^{(i)}, ..., x_M^{(i)}\right)^T$ — i-th input vector, $i = 1, ..., n$

$\sigma^{(i)} = \left(\sigma_1^{(i)}, ..., \sigma_K^{(i)}\right)^T$ — i-th target vector, $i = 1, ..., n$

Problem:

The training of neural network F is the minimization of mean loss L over data sample $\mathscr{D}$:

$$E(w) = \frac{1}{n} \sum_{i=1}^{n} L\left(F, (x^{(i)}, \sigma^{(i)})\right) \to \min_{w}$$

The training of neural network is a kind of optimization problem with objective function $E(w)$. To resolve it optimization techniques are used

Gradient Descent

Idea:
Takes steps toward the negative gradient direction proportional to its absolute value

Equations:

$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau))$$

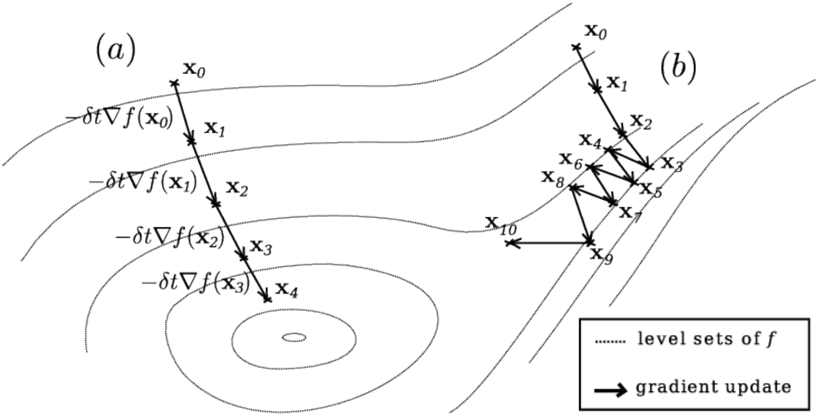
Initial point $w(0) = w_0$, $\alpha > 0$ — step size (learning rate)

Notes:

- The simplest gradient method
- Fixed learning rate α
- Low α leads to time-consuming process, slow changing and smooth optimization trajectory
- High α leads to oscillating or divergent optimization trajectory

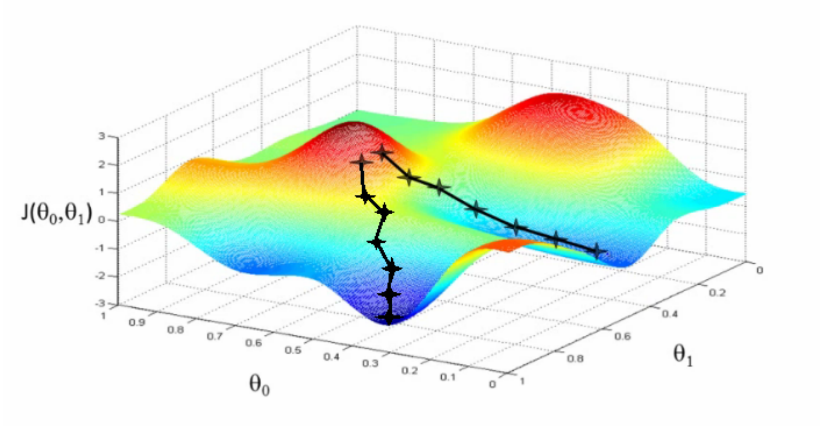
Gradient Descent. Illustration 1

Optimization process depends on initial point



Gradient Descent. Illustration 2

The solution depends on initial point



Steepest Gradient Descent

Idea:

Takes steps toward the negative gradient direction to the point of conditional minimum at this direction

Equations (Cauchy, 1847):

$$w(\tau+1) = w(\tau) - \alpha(\tau)\nabla E(w(\tau))$$

Initial point $w(0) = w_0$

The learning rate $\alpha(\tau)$ is a solution of line search problem:

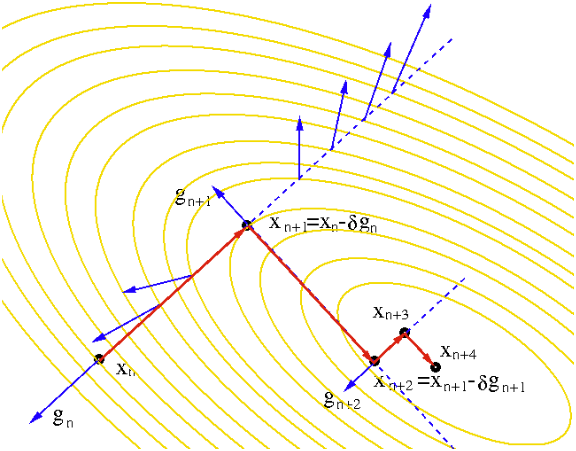
$$\alpha(\tau) = \arg\min_{\alpha>0} E(w(\tau) - \alpha\nabla E(w(\tau)))$$

Notes:

- Adaptive learning rate
- Slow convergence on flat surfaces and ravines (areas where the surface is much more steeply in one dimension than in another)
- Requires solution of line search problem at each iteration
- Steps toward orthogonal directions

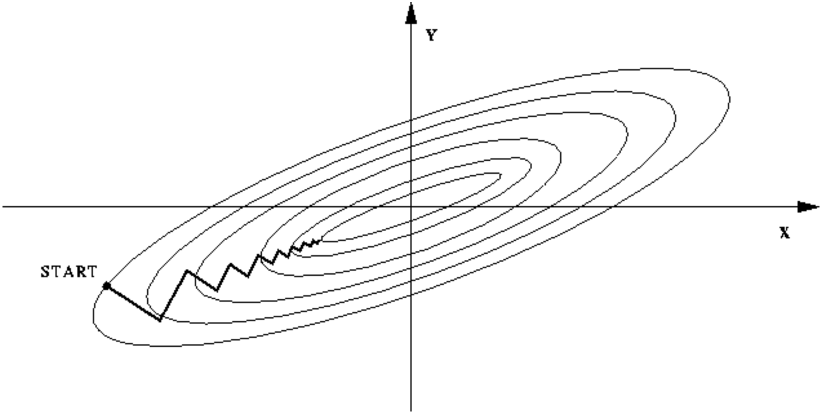
Steepest Gradient Descent. Illustration 1

Steps toward orthogonal directions



Steepest Gradient Descent. Illustration 2

Slow convergence near the optimal point



Gradient Descent with Momentum

Idea:

Takes steps toward the direction that is linear combination of negative gradient and previous direction

Equations (Rumelhart, 1986):

$$w(\tau + 1) = w(\tau) - \alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

$$\Delta w(\tau) = w(\tau) - w(\tau - 1)$$

Initial point $w(0) = w_0$, $\alpha > 0$ — learning rate,
$\mu > 0$ — momentum

Notes:

- The momentum can be interpreted as 'inertia' of descent
- Acceleration on flat areas and slowdown on steep areas give effect of adaptive learning rate
- It's unclear how to choose α and μ

Gradient Descent with Momentum. Flat Areas

$$w(\tau+1) = w(\tau) - \alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

$$\Delta w(\tau+1) = w(\tau+1) - w(\tau) = -\alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

For flat areas:

$$\nabla E(w(\tau)) \approx \nabla E(w(\tau-1))$$

$$\Delta w(\tau+1) = -\alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

Gradient Descent with Momentum. Flat Areas

$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

$$\Delta w(\tau+1) = w(\tau+1) - w(\tau) = -\alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

For flat areas:

$$\nabla E(w(\tau)) \approx \nabla E(w(\tau-1))$$

$$\begin{aligned}\Delta w(\tau+1) &= -\alpha \nabla E(w(\tau)) + \mu \Delta w(\tau) \\ &\approx -\alpha \nabla E(w(\tau)) + \mu(-\alpha \nabla E(w(\tau-1)) + \mu \Delta w(\tau-1))\end{aligned}$$

Gradient Descent with Momentum. Flat Areas

$$w(\tau+1) = w(\tau) - \alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

$$\Delta w(\tau+1) = w(\tau+1) - w(\tau) = -\alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

For flat areas:

$$\nabla E(w(\tau)) \approx \nabla E(w(\tau-1))$$

$$\begin{aligned}\Delta w(\tau+1) &= -\alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)\\ &\approx -\alpha\nabla E(w(\tau)) + \mu(-\alpha\nabla E(w(\tau-1)) + \mu\Delta w(\tau-1))\\ &\approx -\alpha\nabla E(w(\tau)) + \mu(-\alpha\nabla E(w(\tau))+\\ &+ \mu(-\alpha\nabla E(w(\tau-2)) + \mu\Delta w(\tau-2))) \approx \ldots\end{aligned}$$

Gradient Descent with Momentum. Flat Areas

$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$
$$\Delta w(\tau+1) = w(\tau+1) - w(\tau) = -\alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

For flat areas:

$$\nabla E(w(\tau)) \approx \nabla E(w(\tau-1))$$

$$\begin{aligned}\Delta w(\tau+1) &= -\alpha \nabla E(w(\tau)) + \mu \Delta w(\tau) \\ &\approx -\alpha \nabla E(w(\tau)) + \mu(-\alpha \nabla E(w(\tau-1)) + \mu \Delta w(\tau-1)) \\ &\approx -\alpha \nabla E(w(\tau)) + \mu(-\alpha \nabla E(w(\tau)) + \\ &+ \mu(-\alpha \nabla E(w(\tau-2)) + \mu \Delta w(\tau-2))) \approx \ldots \\ &\approx -\alpha \nabla E(w(\tau))(1+\mu+\mu^2+\ldots) \approx -\frac{\alpha}{1-\mu} \nabla E(w(\tau))\end{aligned}$$

The learning rate $\frac{\alpha}{1-\mu} > \alpha \Rightarrow$ acceleration on flat areas

Gradient Descent with Momentum. Ravines

$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

$$\Delta w(\tau+1) = w(\tau+1) - w(\tau) = -\alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

For ravines:

$$\nabla E(w(\tau)) \approx -\nabla E(w(\tau-1))$$

$$\Delta w(\tau+1) = -\alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

Gradient Descent with Momentum. Ravines

$$w(\tau+1) = w(\tau) - \alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

$$\Delta w(\tau+1) = w(\tau+1) - w(\tau) = -\alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

For ravines:

$$\nabla E(w(\tau)) \approx -\nabla E(w(\tau-1))$$

$$\begin{aligned}\Delta w(\tau+1) &= -\alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)\\ &\approx -\alpha\nabla E(w(\tau)) + \mu(-\alpha\nabla E(w(\tau-1)) + \mu\Delta w(\tau-1))\end{aligned}$$

Gradient Descent with Momentum. Ravines

$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$
$$\Delta w(\tau+1) = w(\tau+1) - w(\tau) = -\alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

For ravines:

$$\nabla E(w(\tau)) \approx -\nabla E(w(\tau-1))$$

$$\begin{aligned}\Delta w(\tau+1) &= -\alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)\\ &\approx -\alpha \nabla E(w(\tau)) + \mu(-\alpha \nabla E(w(\tau-1)) + \mu \Delta w(\tau-1))\\ &\approx -\alpha \nabla E(w(\tau)) + \mu(\alpha \nabla E(w(\tau)) +\\ &+ \mu(-\alpha \nabla E(w(\tau-2)) + \mu \Delta w(\tau-2))) \approx \dots\end{aligned}$$

Gradient Descent with Momentum. Ravines

$$w(\tau+1) = w(\tau) - \alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

$$\Delta w(\tau+1) = w(\tau+1) - w(\tau) = -\alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

For ravines:

$$\nabla E(w(\tau)) \approx -\nabla E(w(\tau-1))$$

$$\begin{aligned}\Delta w(\tau+1) &= -\alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)\\ &\approx -\alpha\nabla E(w(\tau)) + \mu(-\alpha\nabla E(w(\tau-1)) + \mu\Delta w(\tau-1))\\ &\approx -\alpha\nabla E(w(\tau)) + \mu(\alpha\nabla E(w(\tau))+\\ &+ \mu(-\alpha\nabla E(w(\tau-2)) + \mu\Delta w(\tau-2))) \approx \ldots\\ &\approx -\alpha\nabla E(w(\tau))(1-\mu+\mu^2-\ldots) \approx -\frac{\alpha}{1+\mu}\nabla E(w(\tau))\end{aligned}$$

The learning rate $\frac{\alpha}{1+\mu} < \alpha \Rightarrow$ slowdown on ravines

Gradient Descent with Momentum. Implementations

Implementation 1:

$$w(\tau+1) = w(\tau) - \alpha\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

$$\Delta w(\tau) = w(\tau) - w(\tau-1)$$

Initial point $w(0) = w_0$, $\alpha > 0$ — learning rate,
$\mu > 0$ — momentum

Implementation 2:

$$w(\tau+1) = w(\tau) - \alpha(1-\mu)\nabla E(w(\tau)) + \mu\Delta w(\tau)$$

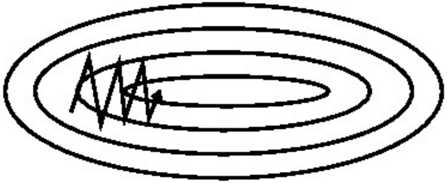
$$\Delta w(\tau) = w(\tau) - w(\tau-1)$$

Initial point $w(0) = w_0$, $\alpha > 0$ — learning rate,
$0 < \mu < 1$ — momentum
$\mu = 0 \Rightarrow$ gradient descent with learning rate α
$\mu = 1 \Rightarrow$ no sensitive to current gradient

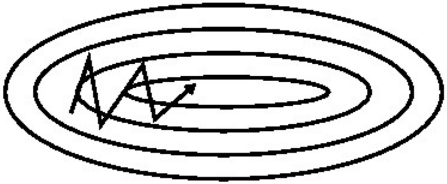
Gradient Descent with Momentum. Illustration

Faster convergence to the optimal point

Gradient descent:



Gradient descent with momentum:



Nesterov Accelerated Gradient (NAG)

Idea:

Takes steps toward the direction that is linear combination of "lookahead" negative gradient and previous direction

Equations (Nesterov, 1983):

$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau) + \mu \Delta w(\tau)) + \mu \Delta w(\tau)$$

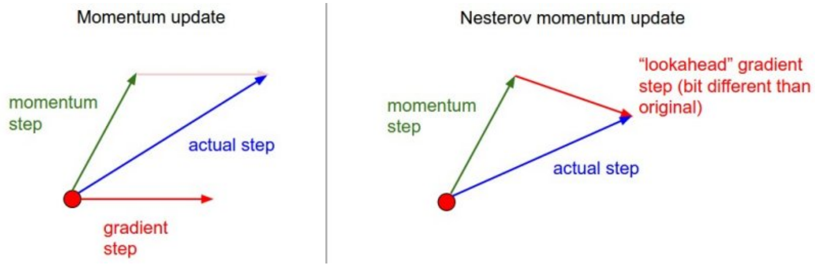
$$\Delta w(\tau) = w(\tau) - w(\tau - 1)$$

Initial point $w(0) = w_0$, $\alpha > 0$ — learning rate,
$\mu > 0$ — momentum

Notes:

- Works slightly better than standard momentum especially for higher values of μ
- It's unclear how to choose α and μ

NAG vs Standard Momentum



Momentum:

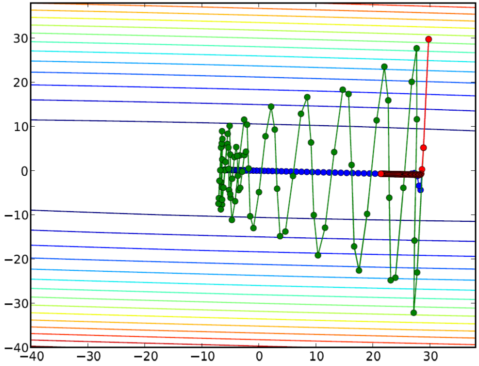
$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau)) + \mu \Delta w(\tau)$$

NAG:

$$w(\tau+1) = w(\tau) - \alpha \nabla E(w(\tau) + \mu \Delta w(\tau)) + \mu \Delta w(\tau)$$

Nesterov Accelerated Gradient. Illustration

Results for oblong quadratic objective



The trajectories of GD (red), momentum (green), and Nesterov's accelerated gradient (blue). Both methods had μ set to 0.95. The global minimum of the quadratic is in the center of the figure, at (0, 0)*

*Sutskever, I. (2013). Training Recurrent Neural Networks. PhD Thesis.

CGD (Conjugate Gradient Descent)

Idea:
Takes steps toward conjugate directions (so improves steepest gradient descent where steps are orthogonal)

Equations:

$$w(\tau + 1) = w(\tau) + \alpha(\tau)d(\tau)$$

where $d(0), d(1), d(2), ...$ — conjugate directions
Initial point $w(0) = w_0$

The learning rate $\alpha(\tau)$ is a solution of line search problem:

$$\alpha(\tau) = \arg\min_{\alpha>0} E(w(\tau) + \alpha d(\tau))$$

The only difference from the steepest gradient descent is in direction $d(\tau)$ instead of $-\nabla E(w(\tau))$ at each iteration τ

Conjugate Directions

Initial conjugate direction is direction of anti-gradient:

$$d(0) = -\nabla E(w(0))$$

The next conjugate direction is determined as a linear combination of current anti-gradient and previous conjugate direction:

$$d(\tau) = -\nabla E(w(\tau)) + \beta(\tau) d(\tau - 1), \quad \tau = 1, 2, ...$$

Fletcher-Reeves algorithm (1964):

$$\beta(\tau) = \frac{\nabla E(\tau)^T \nabla E(\tau)}{\nabla E(\tau-1)^T \nabla E(\tau-1)} = \frac{||\nabla E(\tau)||^2}{||\nabla E(\tau-1)||^2}$$

Polak–Ribiere algorithm (1969):

$$\beta(\tau) = \frac{\nabla E(\tau)^T (\nabla E(\tau) - \nabla E(\tau-1))}{\nabla E(\tau-1)^T \nabla E(\tau-1)}$$

Conjugate Gradient Descent

Notes:

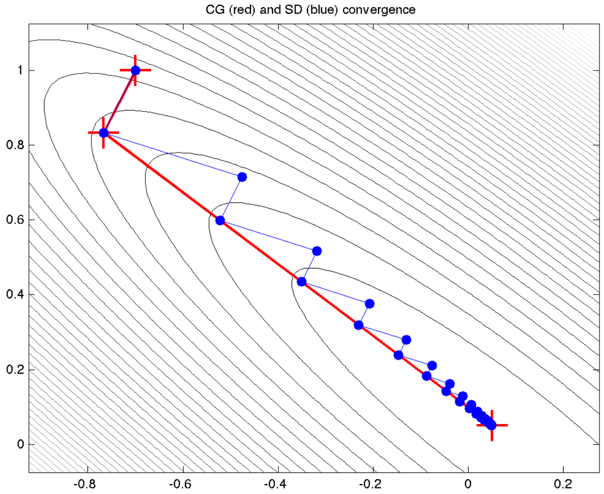
- With a pure quadratic function the minimum is reached within m iterations (m is the dimension of vector w)
- It is recommended to reset search direction every m iterations to the direction of anti-gradient
- For Polak–Ribiere algorithm it is recommended to choose:

$$\beta(\tau) := \max\{0, \beta(\tau)\}$$

  $\beta(\tau) = 0$ means steepest gradient descent
- Extremely effective in dealing with general objective functions
- The computational cost is the same as for steepest gradient descent

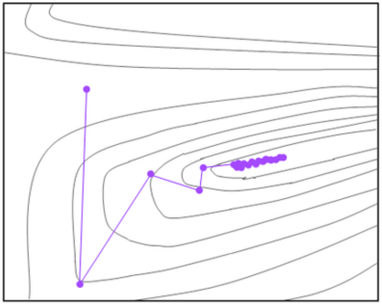
Conjugate Gradient Descent. Illustration 1

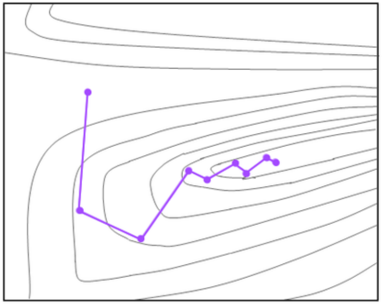
Convergence for quadratic function (CGD vs steepest GD)



Conjugate Gradient Descent. Illustration 2

Convergence for non-quadratic function


GD


CGD

Learning Rate Tuning

Line search used in steepest gradient descent and CGD is too expensive for large training samples

Vanilla gradient descent uses constant learning rate that can be inefficient for optimization

When training neural networks, it is often useful to change learning rate as the training progresses

Approaches:

- Learning rate schedules (annealing the learning rate)
  The learning rate decays with according to the defined scheme
  The system can cool too quickly, unable to reach the best position
- Adaptive learning rate methods

Annealing the Learning Rate

- Step decay
  Reduce the learning rate by some factor every few epochs (e.g., reducing the learning rate by a half every 5 epochs, or by 0.1 every 20 epochs.)
- Exponential decay
  $$\alpha(\tau) = \alpha_0 e^{-k\tau}$$
- $1/\tau$ decay
  $$\alpha(\tau) = \frac{\alpha_0}{1 + k\tau}$$

Learning rate schedules (methods of learning rate annealing) impose additional hyperparameters to the training that depend heavily on the type of model and problem

Another problem is that at each iteration the same learning rate is applied to all parameter updates

AdaGrad

Idea:

Modified gradient descent with per-parameter adaptive learning rate, uses the cumulative squared gradient to adapt it

Equations (Duchi, Hazan, Singer, 2011):

$$w_j(\tau+1) = w_j(\tau) - \alpha_j(\tau)\frac{\partial E(w(\tau))}{\partial w_j}$$

$$\alpha_j(\tau) = \frac{\alpha}{\sqrt{G_j(\tau)+\varepsilon}}, \quad j = 1, ..., m$$

$$G_j(\tau) = G_j(\tau-1) + \left(\frac{\partial E(w(\tau))}{\partial w_j}\right)^2, \quad \tau = 0, 1, ...$$

Initial point $w(0) = w_0$, $\alpha > 0$ — base learning rate
ε — smoothing term that avoids division by zero (usually $\varepsilon \simeq 10^{-8}$)
$G_j(\tau)$ — cumulative squared partial derivative w.r.t. parameter w_j, up to iteration τ, $G_j(-1) = 0$, $j = 1, ..., m$

AdaGrad

Notes:

- Uses a different learning rate for every parameter w_j, $j = 1, ..., m$
- Extreme parameter updates get dampened, while parameters that get few or small updates receive higher learning rates
- Eliminates the need to manually tune the learning rate (usually base learning rate $\alpha = 0.01$)
- Accumulates the squared gradients in the denominator that leads the learning rate to decrease and converge to zero (monotonically decreasing learning rate)
- Effective for objectives with sparse gradients

RMSProp (Root Mean Square Propagation)

Idea:
Modified gradient descent with per-parameter learning rate, uses the exponentially averaged squared gradient to adapt the learning rate

Equations (Hinton, 2012):

$$w_j(\tau+1) = w_j(\tau) - \alpha_j(\tau)\frac{\partial E(w(\tau))}{\partial w_j}$$

$$\alpha_j(\tau) = \frac{\alpha}{\sqrt{G_j(\tau)+\varepsilon}}, \quad j = 1, ..., m$$

$$G_j(\tau) = \rho G_j(\tau-1) + (1-\rho)\left(\frac{\partial E(w(\tau))}{\partial w_j}\right)^2, \quad \tau = 1, 2, ...$$

Initial point $w(0) = w_0$, $\alpha > 0$ — base learning rate
ρ — forgetting factor, $0 < \rho < 1$, $G(0) = (\nabla E(w(0)))^2$
ε — smoothing term that avoids division by zero (usually $\varepsilon \simeq 10^{-8}$)

AdaDelta

Idea:
Modified gradient descent with per-parameter learning rate, uses the acceleration term based on historical updates

Equations (Zeiler, 2012):

$$w_j(\tau+1) = w_j(\tau) - \alpha_j(\tau)\frac{\partial E(w(\tau))}{\partial w_j}$$

$$\alpha_j(\tau) = \frac{\sqrt{D_j(\tau)+\varepsilon}}{\sqrt{G_j(\tau)+\varepsilon}}$$

$$G_j(\tau) = \rho G_j(\tau-1) + (1-\rho)\left(\frac{\partial E(w(\tau))}{\partial w_j}\right)^2, \quad \tau = 0, 1, \dots$$

$$D_j(\tau) = \rho D_j(\tau-1) + (1-\rho)(\Delta w_j(\tau))^2, \quad \tau = 0, 1, \dots$$

Initial point $w(0) = w_0$, ρ — forgetting factor, $0 < \rho < 1$
ε — smoothing term that avoids division by zero (usually $\varepsilon \simeq 10^{-8}$)

AdaDelta

$G_j(\tau)$ — exponentially averaged squared partial derivative w.r.t. parameter w_j, up to iteration τ, $G_j(-1) = 0$, $j = 1, ..., m$
$D_j(\tau)$ — exponentially averaged squared difference $\Delta w_j(\tau)$, $\Delta w_j(\tau) = w_j(\tau) - w_j(\tau - 1)$, up to iteration τ, $D_j(-1) = 0$, $\Delta w_j(0) = \Delta_{0j}$, $j = 1, ..., m$

Notes:

- Uses a different learning rate for each parameter w_j
- Implements exponentially decaying average of the squared gradients (instead of cumulative squared gradients of AdaGrad)
- Implements exponentially decaying average of the squared updates in the numerator of learning rate
- Doesn't demonstrate the continual decay of learning rate throughout training (as it is for AdaGrad)
- Eliminates the need to manually tune the learning rate, do not even need to set an initial learning rate

Adam (Adaptive Moment Estimation)

Idea:
Modified gradient descent with per-parameter learning rate, uses estimates of first and second moments of the gradient

Equations (Kingma (OpenAI), 2015):

$$w_j(\tau+1) = w_j(\tau) - \alpha_j(\tau)\hat{g}_j(\tau)$$

$$\alpha_j(\tau) = \frac{\alpha}{\sqrt{\hat{G}_j(\tau)+\varepsilon}}, \quad j = 1, ..., m$$

$$\hat{g}(\tau) = \frac{g(\tau)}{1-\beta_1^{\tau+1}}, \quad \hat{G}(\tau) = \frac{G(\tau)}{1-\beta_2^{\tau+1}}, \quad \tau = 0, 1, ...$$

$$g_j(\tau) = \beta_1 g_j(\tau-1) + (1-\beta_1)\frac{\partial E(w(\tau))}{\partial w_j}, \quad \tau = 0, 1, ...$$

$$G_j(\tau) = \beta_2 G_j(\tau-1) + (1-\beta_2)\left(\frac{\partial E(w(\tau))}{\partial w_j}\right)^2, \quad \tau = 0, 1, ...$$

Adam (Adaptive Moment Estimation)

Initial point $w(0) = w_0$, β_1, β_2 — forgetting factors, $0 < \beta_1 < 1$, $0 < \beta_2 < 1$, ε — smoothing term
$g_j(\tau)$ — exponentially averaged partial derivative w.r.t. w_j, up to iteration τ, $g_j(-1) = 0$, $j = 1, ..., m$ (1-st moment)
$G_j(\tau)$ — exponentially averaged squared partial derivative w.r.t. w_j, up to iteration τ, $G_j(-1) = 0$, $j = 1, ..., m$ (2-st moment)
$\hat{g}(\tau)$, $\hat{G}(\tau)$ — corrected 1-st and 2-nd moments of gradient at iteration τ

Notes:

- Uses a different learning rate for each parameter w_j
- Effective for functions with very noisy and/or sparse gradients
- Relatively easy to configure where the default configuration parameters do well on most problems ($\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$)
- Adam looks like RMSProp + Momentum

RProp (Resilient Propagation)

Idea:
Uses per-parameter learning rate adaptation based on the sign of partial derivatives

Equations (Riedmiller, Braun, 1993):

$$w_j(\tau+1) = w_j(\tau) - \text{sgn}\left(\frac{\partial E(w(\tau))}{\partial w_j}\right)\Delta_j(\tau)$$

$$\Delta_j(\tau) = \begin{cases} \min\{\eta^+\Delta_j(\tau-1), \Delta_{\max}\}, & \frac{\partial E(w(\tau-1))}{\partial w_j}\frac{\partial E(w(\tau))}{\partial w_j} > 0 \\ \max\{\eta^-\Delta_j(\tau-1), \Delta_{\min}\}, & \frac{\partial E(w(\tau-1))}{\partial w_j}\frac{\partial E(w(\tau))}{\partial w_j} < 0 \\ \Delta_j(\tau-1), \quad otherwise, \end{cases}$$

where $0 < \eta^- < 1 < \eta^+$ — decay and acceleration parameters,
$\Delta_{\min}$, $\Delta_{\max}$ — minimal and maximal weight-steps
Initial point $w(0) = w_0$, $\Delta(0) = \Delta_0$ — initial weight-step

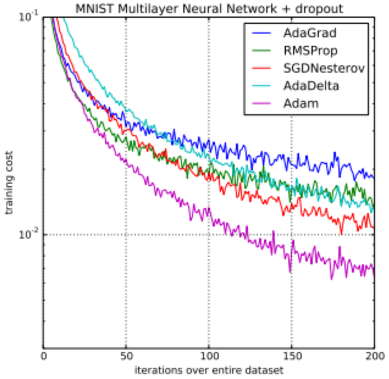
RProp (Resilient Propagation)

Notes:

- Uses a different learning rate for each parameter w_j that depends only on sign of the gradient and not of its amount
- Very suitable for problems where the gradient is numerically estimated or the objective is noisy
- Increases the learning rate for a weight multiplicatively if signs of last two partial derivatives agree and decreases it multiplicatively if don't agree
- Modifications: PRProp+, PRProp–, iPRProp+, iPRProp–
- Easy to implement and not susceptible to numerical problems
- The RProp algorithms are known to be very robust with respect to their internal parameters (the recommended decay and acceleration parameters are $\eta^- = 0.5$, $\eta^+ = 1.2$)
- The algorithm is applicable only for batch training and inefficient for stochastic and mini-batch training

Comparison of GD Algorithms. Illustration 1

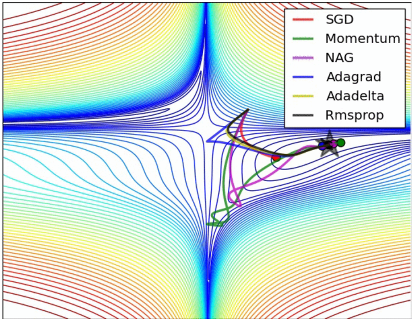
Comparison of Adam to other optimization algorithms training a multilayer perceptron on MNIST images*



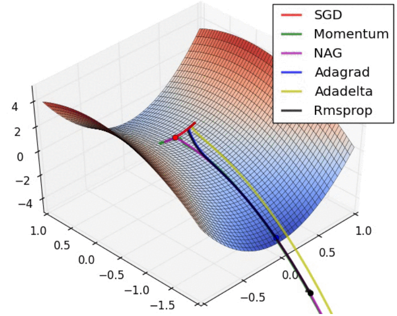
*Kingma D.P., Ba, J.L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13

Comparison of GD Algorithms. Illustration 2

Behaviour on the contours of the Beale function



Behaviour at a saddle point



`http://cs231n.github.io/neural-networks-3/`: Fig. 1, Fig. 2

Stochastic Gradient Descent (SGD)

$\mathscr{D} = \{(x^{(1)}, \sigma^{(1)}), ..., (x^{(n)}, \sigma^{(n)})\}$ — available data sample for training

Objective:

$$E(w) = \frac{1}{n}\sum_{i=1}^{n} E^{(i)}(w) \to \min_{w}$$

where $E^{(i)}(w)$ is a loss on example $(x^{(i)}, \sigma^{(i)})$, $i = 1, ..., n$

Idea:
Instead of optimize error $E(w)$ on whole sample $\mathscr{D}$ let's optimize it on each example $(x^{(i)}, \sigma^{(i)})$ consequently, $i = 1, ..., n$

For vanilla gradient descent:

$$w(\tau + 1) = w(\tau) - \alpha \nabla E^{(i)}(w(\tau))$$

Initial point $w(0) = w_0$, $\alpha > 0$ — learning rate

SGD Algorithm

SGD algorithm:

Step 1. Choose initial vector of parameters w_0

Step 2. Randomly shuffle examples in the training set $\mathscr{D}$

Step 3. For each example from $\mathscr{D}$ update the vector of parameters

Step 4. Repeat steps 2 and 3 until the stopping criterion is met

Definition

Epoch is one forward pass and one backward pass of all training examples

For batch gradient descent: one epoch consists of 1 iteration, performs model updates after each training epoch

For stochastic gradient descent: one epoch consists of n iterations, performs model updates after each training example

SGD is also called on-line gradient descent

Stochastic GD. Advantages and Disadvantages

Advantages:

- The frequent updates immediately give an insight into the performance of the model and the rate of improvement
- The frequent updates can result in faster learning on some problems
- Computation time per update does not depend on training sample size
- Can allow the model to avoid local minima due to the noisy update process

Disadvantages:

- Updating the model so frequently is more computationally expensive
- Can result in a noisy gradient signal
- The noisy learning process can make it hard for the algorithm to settle on an error minimum for the model

Batch GD. Advantages and Disadvantages

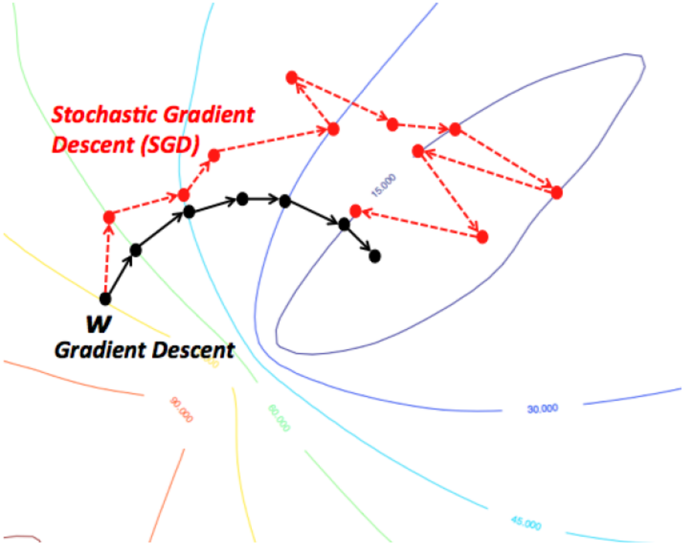
Advantages:

- More computationally efficient than stochastic gradient descent due to fewer updates of the model
- More stable gradient and may result in a more stable convergence on some problems
- Additive gradient can be calculated on parallel processing based implementations

Disadvantages:

- Can converge to a local minima
- Additional complexity of accumulating gradients across all training examples
- Commonly, batch gradient descent is implemented in such a way that it requires the entire training dataset in memory
- Model updates, and in turn training speed, may become very slow for large datasets

SGD. Illustration



Mini-Batch Gradient Descent

Idea:

The training dataset $\mathscr{D}$ is partitioned into mini-batches $\mathscr{D}_1, ..., \mathscr{D}_P$ that are used to calculate model error and update model coefficients

For vanilla gradient descent:

$$w(\tau+1) = w(\tau) - \alpha \nabla E_p(w(\tau))$$

where $E_p(w(\tau))$ is a error on p-th mini-batch:

$$E_p(w(\tau)) = \frac{1}{|\mathscr{D}_p|} \sum_{i \in \mathscr{D}_p} E^{(i)}(w(\tau))$$

where $|\mathscr{D}_p|$ is a size of p-th mini-batch, $p = 1, ..., P$,
P is a number of mini-batches

Initial point $w(0) = w_0$, $\alpha > 0$ — learning rate

For mini-batch gradient descent: one epoch consists of P iterations, performs model updates after each mini-batch

Mini-Batch GD. Advantages and Disadvantages

Mini-batch gradient descent is a trade-off between stochastic gradient descent and batch gradient descent:

$P = 1 \Rightarrow$ Stochastic gradient descent

$P = n \Rightarrow$ Batch gradient descent

$1 < P < n \Rightarrow$ Mini-batch gradient descent

Advantages:

- More robust convergence than batch gradient descent, avoiding local minima
- The batched updates provide a computationally more efficient process than stochastic gradient descent
- Less noise and more accurate estimates of gradient than in stochastic gradient descent

Disadvantages:

- Additional mini-batch size hyperparameter for the learning algorithm

Mini-Batch Size

Mini-batch size is a hyperparameter for the learning algorithm

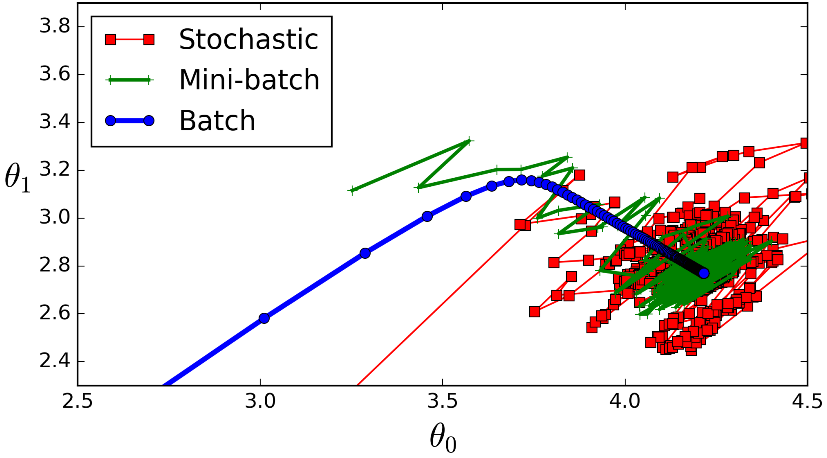
In theory, mini-batch size should impact training time and not so much model performance

Tips to choose mini-batch size*:

- Mini-batch size is typically chosen between 1 and few hundreds. A good default for batch size might be 32
- It is a good idea to review training curves of model validation error against training time with different batch sizes when tuning the batch size
- Mini-batch size can be optimized separately of the other hyperparameters
- Tune batch size and learning rate after tuning all other hyperparameters

*Bengio Y. (2012). Practical recommendations for gradient-based training of deep architectures. In Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol. 7700, 437-478

Mini-Batch GD. Illustration



Gradient Descent Algorithms for Mini-Batch Training

Stochastic and mini-batch mode can be used with following gradient descent algorithms:

- GD with momentum
- Nesterov accelerated gradient
- AdaGrad
- RMSProp
- AdaDelta
- Adam
- Adamax (Adam with L_∞ norm)
- Nadam (Adam but Nesterov accelerated gradient is used instead of momentum)

Batch objective function $E(w)$ in iteration scheme should be replaced to objective on example $E^{(i)}(w)$ or on mini-batch $E_p(w)$

Newton's Method

Idea:

Second order method based on local quadratic approximations of objective $E(w)$

The second order Taylor expansion of objective $E(w)$ around w is used as quadratic approximation:

$$E(w+\varepsilon) \approx E(w) + \nabla E(w)^T \varepsilon + \frac{1}{2}\varepsilon^T H(w)\varepsilon$$

where $H(w)$ is the Hessian matrix of objective $E(w)$:

$$H(w) = \begin{pmatrix} \frac{\partial^2 E(w)}{\partial w_1^2} & \dots & \frac{\partial^2 E(w)}{\partial w_1 \partial w_m} \\ \dots & \dots & \dots \\ \frac{\partial^2 E(w)}{\partial w_m \partial w_1} & \dots & \frac{\partial^2 E(w)}{\partial w_m^2} \end{pmatrix}$$

is the square matrix of second-order partial derivatives of $E(w)$

The Hessian describes the local curvature of the objective

Newton's Method

Quadratic approximation:

$$E(w+\varepsilon) \approx E(w) + \nabla E(w)^T \varepsilon + \frac{1}{2}\varepsilon^T H(w)\varepsilon$$

We want to find ε such that $E(w+\varepsilon)$ is a minimum:

$$\frac{\partial E(w+\varepsilon)}{\partial \varepsilon} = 0$$

$$\nabla E(w) + H(w)\varepsilon = 0$$

$$H(w)\varepsilon = -\nabla E(w)$$

$$\varepsilon = -H(w)^{-1}\nabla E(w)$$

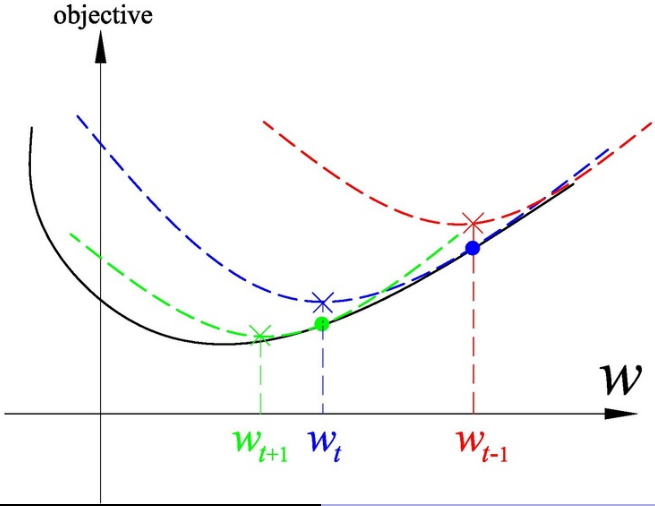
Equations:

$$w(\tau+1) = w(\tau) + \varepsilon = w(\tau) - H(w(\tau))^{-1}\nabla E(w(\tau))$$

Initial point $w(0) = w_0$

Newton's Method. Illustration

The Newton's method is also called Newton-Raphson method



Newton's Method

Notes:

- Multiplying by the inverse Hessian leads the optimization to take more aggressive steps in directions of shallow curvature and shorter steps in directions of steep curvature
- It is not guaranteed that Newton's method will converge if an initial point $w(0)$ is too far from the minimum
- Equal to 1-st order method the Newton's method converges to a local minimum
- The Newton's method is impractical for most learning problems because computing (and inverting) the Hessian in its explicit form is a very costly process in both space and time
  For instance, a neural network with one million parameters would have a Hessian matrix of size [1,000,000 x 1,000,000], occupying approximately 3725 gigabytes of RAM

Levenberg-Marquardt Method

Idea:

Uses 1-st order approximation of Hessian $H(w)$ for objective $E(w)$ with quadratic loss function

Quasi-Newton's methods use the idea of Newton's method and an approximation of Hessian (or inverse Hessian)

Levenberg-Marquardt method is a quasi-Newton's method

Objective:

$$E(w) = \frac{1}{n}\sum_{i=1}^{n} E^{(i)}(w) = \frac{1}{2n}\sum_{i=1}^{n} e^{(i)}(w)^T e^{(i)}(w)$$

where $E^{(i)}(w) = \frac{1}{2}e^{(i)}(w)^T e^{(i)}(w) = \frac{1}{2}\left|\left|y^{(i)} - \sigma^{(i)}\right|\right|^2$ — loss on i-th example,

$e^{(i)}(w) = y^{(i)}(w) - \sigma^{(i)}$ — error on i-th example, $i = 1, ..., n$

Hessian Matrix for Quadratic Loss

Hessian matrix:

$$H(w) = \frac{1}{n}\sum_{i=1}^{n} H^{(i)}(w)$$

where $H^{(i)}(w)$ — Hessian of loss $E^{(i)}(w)$, $i = 1, ..., n$:

$$\begin{aligned} H^{(i)}(w) &= \frac{\partial^2 E^{(i)}(w)}{\partial w^T \partial w} = \frac{\partial}{\partial w^T}\nabla E^{(i)}(w) = \frac{\partial}{\partial w^T}\left(J^{(i)}(w)^T e^{(i)}(w)\right) \\ &= \frac{\partial J^{(i)}(w)^T}{\partial w^T} e^{(i)}(w) + J^{(i)}(w)^T \frac{\partial e^{(i)}(w)}{\partial w^T} \\ &= R^{(i)}(w) + J^{(i)}(w)^T J^{(i)}(w) \end{aligned}$$

where $J^{(i)}(w)$ is the Jacobian matrix of error $e^{(i)}(w)$ on i-th example, $e^{(i)}(w) = \left(e_1^{(i)}(w), ..., e_K^{(i)}(w)\right)^T$, $i = 1, ..., n$

$R^{(i)}(w)$ is the matrix that contains second derivatives of $e^{(i)}(w)$

Jacobian Matrix of Errors

The following expressions were used:

$$\nabla \left(e(w)^T e(w)\right) = 2J_e(w)^T e(w)$$

$$\frac{\partial e(w)}{\partial w^T} = J_e(w)$$

The Jacobian matrix of error $e^{(i)}(w) = \left(e_1^{(i)}(w), ..., e_K^{(i)}(w)\right)^T$:

$$J^{(i)}(w) = \begin{pmatrix} \frac{\partial e_1^{(i)}}{\partial w_1} & ... & \frac{\partial e_1^{(i)}}{\partial w_m} \\ ... & ... & ... \\ \frac{\partial e_K^{(i)}}{\partial w_1} & ... & \frac{\partial e_K^{(i)}}{\partial w_m} \end{pmatrix}$$

is the matrix of first-order partial derivatives of vector-valued function $e^{(i)}(w)$, $i = 1, ..., n$

Diagonal Approximation of Hessian Matrix

Hessian matrix:

$$H(w) = \frac{1}{n}\sum_{i=1}^{n} H^{(i)}(w) = \frac{1}{n}\sum_{i=1}^{n}\Big(R^{(i)}(w) + J^{(i)}(w)^T J^{(i)}(w)\Big)$$

Levenberg-Marquardt method uses scalar matrix μI_m as an approximation of matrix $R^{(i)}(w)$:

$$H(w) \approx \tilde{H}(w) = \frac{1}{n}\sum_{i=1}^{n}\Big(\mu I_m + J^{(i)}(w)^T J^{(i)}(w)\Big)$$

$$= \mu I_m + \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(w)^T J^{(i)}(w)$$

Gradient:

$$\nabla E(w(\tau)) = \frac{1}{n}\sum_{i=1}^{n} \nabla E^{(i)}(w(\tau)) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(w)^T e^{(i)}(w)$$

Levenberg-Marquardt Method

Equations (1963):

$$w(\tau+1) = w(\tau) - \tilde{H}^{-1}(w(\tau)) \cdot \nabla E(w(\tau))$$

$$w(\tau+1) = w(\tau) - \left(\mu I_m + \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(w(\tau))^T J^{(i)}(w(\tau))\right)^{-1} \times$$
$$\times \left(\frac{1}{n}\sum_{i=1}^{n} J^{(i)}(w(\tau))^T e^{(i)}(w(\tau))\right)$$

Initial point $w(0) = w_0$

$\mu > 0$ — damping parameter
$\mu \approx 0 \Rightarrow$ Newton's method using approximate Hessian matrix
$\mu \gg 0 \Rightarrow$ Vanilla gradient descent with learning rate $\alpha = \frac{1}{\mu}$

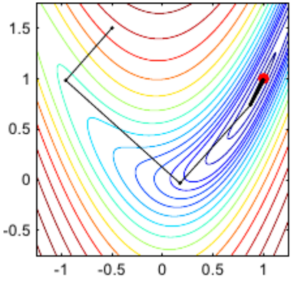
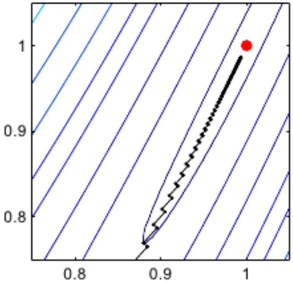
Levenberg-Marquardt Method

Notes:

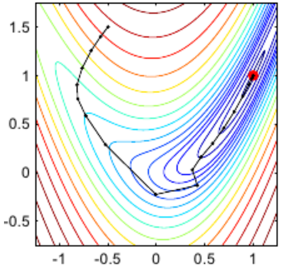
- It is recommended to choose the damping parameter μ larger in the beginning of optimization process and smaller near the optimal point
- Adaptive damping parameter μ rule: decrease it after each successful step (reduction in performance function) and increase it only when a tentative step would increase the performance function
- Fast convergence in ravine areas
- Impractical for learning problems with large number of parameters because inverting the approximate Hessian matrix is a very costly process

Levenberg-Marquardt Method. Illustration

Steepest GD:

Levenberg-Marquardt:



BFGS (Broyden–Fletcher–Goldfarb–Shanno)

Idea:

Uses iterative approximation of inverse Hessian $H(w)$

Equations:

$$w(\tau+1) = w(\tau) - \alpha(\tau)B(\tau)\nabla E(w(\tau))$$

where $B(\tau) \approx H(w(\tau))^{-1}$ is an approximation of inverse Hessian matrix

BFGS (1983):

$$B(\tau) = \left(I_m - \frac{s(\tau)y(\tau)^T}{y(\tau)^T s(\tau)}\right) B(\tau-1) \left(I_m - \frac{y(\tau)s(\tau)^T}{y(\tau)^T s(\tau)}\right) + \frac{s(\tau)s(\tau)^T}{y(\tau)^T s(\tau)}$$

where $s(\tau) = w(\tau) - w(\tau-1)$, $y(\tau) = \nabla E(w(\tau)) - \nabla E(w(\tau-1))$

Initial point $w(0) = w_0$
Initial approximation $B(0) = B_0$

BFGS

BFGS uses $-B(\tau)\nabla E(w(\tau))$ only as a direction of search. The step size $\alpha(\tau)$ is determined as a solution of line-search problem:

$$\alpha(\tau) = \arg\min_{\alpha>0} E(w(\tau) - \alpha B(\tau)\nabla E(w(\tau)))$$

Notes:

- Practically, $B(0)$ can be initialized with identity matrix I_m, so that the first step will be equivalent to a gradient descent
- Requires solution of line search problem at each iteration
- The method is inefficient in high dimension parameters space
- Fast convergence in ravine areas
- Stores approximated inverse Hessian matrix $B(\tau)$ at each iteration
- L-BFGS modification (Limited memory BFGS): requires only retaining the most recent gradients

Overview

- 1-st order gradient descent methods
  - Vanilla gradient descent
  - Gradient descent methods with line search
    — Steepest gradient descent
    — Conjugate gradient descent
  - Adaptive learning rate methods
    — GD with momentum, Nesterov accelerated gradient
  - Per-parameter adaptive learning rate methods
    — AdaGrad, AdaDelta, RMSProp, Adam, ...
    — RProp
- 2-nd order methods
  - Newton's method
  - Quasi-Newton's methods
    — Levenberg-Marquardt method
    — BFGS

Which Gradient Descent Method to Use?

Training modes:

- Batch gradient descent
- Stochastic gradient descent
- Mini-batch gradient descent

For large neural networks the gradient descent method must be effective in stochastic or mini-batch modes, computationally simple and scalable

You shouldn't care too much about find the minimum of your training performance function, because it's only an approximation to what you really care about, an acceptable performance of model

For most training problems per-parameter adaptive learning rate methods are widely used

Network Weights Initialization

The training of neural network with any gradient descent method starts from initial point $w(0) = w_0$

How to choose w_0?
The common approach is to generate random initial weights from some distribution with zero mean

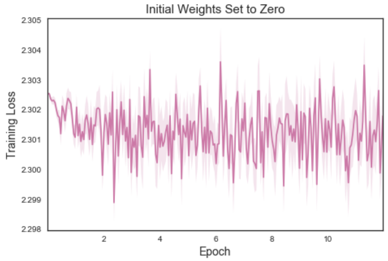
Usual distributions:

- Normal $N(0, \sigma)$
- Truncated normal $N^*(0, \sigma)$
  The support of distribution is $(-2\sigma, 2\sigma)$. If the random number is out of this range, the value is discarded and re-drawn
- Uniform $R(-a, a)$
  The variance $\sigma^2 = \mathrm{D}[R(-a, a)] = \frac{a^2}{3}$

The weight initialization have a profound impact on both the convergence rate and final quality of a network

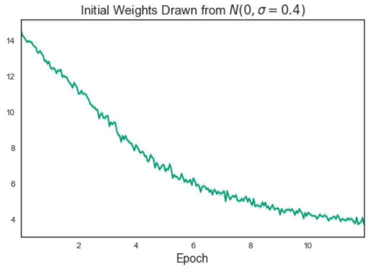
Network Weights Initialization. Illustration

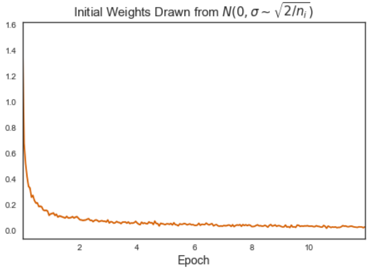
Learning curves of a neural net initialized by three different ways



Training on MNIST images dataset:

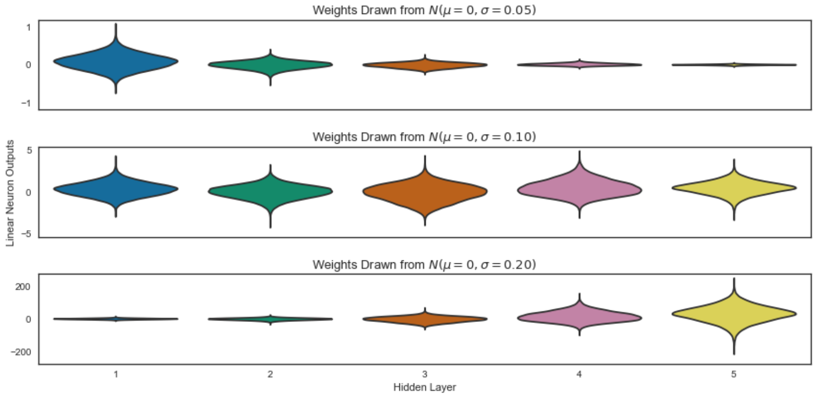
`http://yann.lecun.com/exdb/mnist/`





Information Flow in Neural Network

Activations of hidden layers for different initial weights distributions



Top: the activations quickly vanish to almost nothing
Middle: good information flow throughout the network
Bottom: the activations become increasingly spread out

Good Information Flowing

The idea of weights initialization is to keep information flowing in the neural network

Good information flowing means:

- to maintain variance of activations throughout the network:

$$\mathrm{D}[y^1] \approx \mathrm{D}[y^2] \approx ... \approx \mathrm{D}[y^L]$$

- to maintain variance of gradients (“dual” activations) throughout the network:

$$\mathrm{D}\left[\frac{\partial L}{\partial h_1}\right] \approx \mathrm{D}\left[\frac{\partial L}{\partial h_2}\right] \approx ... \approx \mathrm{D}\left[\frac{\partial L}{\partial h_L}\right]$$

  where L is the loss function

Variances of Neurons' Activations

Mathematical model of multilayer neural network:

$$\begin{cases} h^l = W^l y^{l-1} - b^l \\ y^l = f_l(h^l) \end{cases} \qquad l = 1, ..., L$$

Consider linear activation functions $f_l(h) = h$, $b^l = 0$, $l = 1, ..., L$

Variances of activations:

$$\mathrm{D}[y_i^l] = \mathrm{D}[h_i^l] = \mathrm{D}\left[\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} - b_i^l\right] = \sum_{j=1}^{N_{l-1}} \mathrm{D}[w_{ij}^l y_j^{l-1}]$$

$$= \sum_{j=1}^{N_{l-1}} \mathrm{D}[w_{ij}^l]\mathrm{D}[y_j^{l-1}] = \sigma_l^2 \sum_{j=1}^{N_{l-1}} \mathrm{D}[y_j^{l-1}] = N_{l-1}\sigma_l^2 \mathrm{D}[y^{l-1}]$$

The expression $\mathrm{D}[XY] = \mathrm{D}[X]\mathrm{D}[Y] + m_X^2\mathrm{D}[Y] + m_Y^2\mathrm{D}[X]$ (X and Y are independent random variables) were used (prove it!)

Variances of Gradients

Backpropagation equations:

$$\Delta_i^L = \frac{\partial Loss}{\partial h_i^L} = \frac{\partial Loss}{\partial y_i} f_L'(h_i^L), \quad i = \overline{1, K}$$

$$\Delta_i^l = \frac{\partial Loss}{\partial h_i^l} = \left(\sum_{j=1}^{N_{l+1}} \Delta_j^{l+1} w_{ji}^{l+1} \right) f_l'(h_i^l), \quad l = \overline{1, L-1}, \; i = \overline{1, N_l}$$

Variances of gradients:

$$\mathrm{D}[\Delta_i^{l-1}] = \mathrm{D}\left[\sum_{j=1}^{N_l} \Delta_j^l w_{ji}^l \right] = \sum_{j=1}^{N_l} \mathrm{D}[\Delta_j^l w_{ji}^l]$$

$$= \sum_{j=1}^{N_l} \mathrm{D}[\Delta_j^l] \mathrm{D}[w_{ji}^l] = \sigma_l^2 \sum_{j=1}^{N_l} \mathrm{D}[\Delta_j^l] = N_l \sigma_l^2 \mathrm{D}[\Delta^l]$$

Variances of Weights

Conditions of good information flowing:

$$\mathrm{D}[y_i^l] = N_{l-1}\sigma_l^2 \mathrm{D}[y^{l-1}], \quad \mathrm{D}[y_i^l] = \mathrm{D}[y_i^{l-1}] \Rightarrow \sigma_l^2 = \frac{1}{N_{l-1}}$$

$$\mathrm{D}[\Delta_i^{l-1}] = N_l\sigma_l^2 \mathrm{D}[\Delta^l], \quad \mathrm{D}[\Delta_i^{l-1}] = \mathrm{D}[\Delta_i^l] \Rightarrow \sigma_l^2 = \frac{1}{N_l}$$

A compromise between these two conditions is the harmonic mean:

$$\sigma_l^2 = \frac{2}{N_{l-1} + N_l}, \quad l = 1, ..., L$$

N_{l-1} is the number of ingoing connections of neurons in l-th layer

N_l is the number of outgoing connections of l-th layer

The variance of initial layer's weights should be chosen as the harmonic mean between numbers of ingoing and outgoing connections of the layer

Xavier/Glorot Weight Initialization Method

The idea is to estimate variances of layers' activations in forward pass and variances of gradients in backward pass and to apply condition of good information flowing (X. Glorot, 2010)

- Linear, tanh or softsign activation functions

$$N^*\left(0, \sqrt{\frac{2}{N_{l-1}+N_l}}\right) \text{ or } R\left(-\sqrt{\frac{6}{N_{l-1}+N_l}}, \sqrt{\frac{6}{N_{l-1}+N_l}}\right)$$

- Rectified linear activation functions

$$N^*\left(0, \sqrt{\frac{4}{N_{l-1}+N_l}}\right) \text{ or } R\left(-\sqrt{\frac{12}{N_{l-1}+N_l}}, \sqrt{\frac{12}{N_{l-1}+N_l}}\right)$$

- Logistic activation function

$$N^*\left(0, 4\sqrt{\frac{2}{N_{l-1}+N_l}}\right) \text{ or } R\left(-4\sqrt{\frac{6}{N_{l-1}+N_l}}, 4\sqrt{\frac{6}{N_{l-1}+N_l}}\right)$$