Gradient Descent Algorithms

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu
Machine Learning is Optimization

- Parametric ML involves minimizing an **objective function** $J(w)$:
  - Also called **cost function**, **loss function**, or **error function**.
  - Want to find $\hat{w} = \arg\min_w J(w)$

- Numerical optimization procedure:
  1. Start with some guess for $w^0$, set $\tau = 0$.
  2. Update $w^\tau$ to $w^{\tau+1}$ such that $J(w^{\tau+1}) \leq J(w^\tau)$.
  3. Increment $\tau = \tau + 1$.
  4. Repeat from 2 until $J$ cannot be improved anymore.
Gradient-based Optimization

• How to update \( w^\tau \) to \( w^{\tau+1} \) such that \( J(w^{\tau+1}) \leq J(w^\tau) \)?

• Move \( w \) in the direction of steepest descent:
  \[
  w^{\tau+1} = w^\tau + \eta g
  \]
  - \( g \) is the direction of steepest descent, i.e. direction along which \( J \) decreases the most.
  - \( \eta \) is the learning rate and controls the magnitude of the change.
Gradient-based Optimization

• Move \( \mathbf{w} \) in the direction of **steepest descent**: 
  \[
  \mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} + \eta \mathbf{g}
  \]

• What is the direction of steepest descent of \( J(\mathbf{w}) \) at \( \mathbf{w}^{\tau} \)?
  – The gradient \( \nabla J(\mathbf{w}) \) is in the direction of steepest ascent.
  – Set \( \mathbf{g} = -\nabla J(\mathbf{w}) \Rightarrow \) the **gradient descent** update: 
    \[
    \mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta \nabla J(\mathbf{w}^{\tau})
    \]
Gradient Descent Algorithm

• Want to minimize a function $J : \mathbb{R}^n \rightarrow \mathbb{R}$.
  - $J$ is differentiable and convex.
  - compute gradient of $J$ i.e. direction of steepest increase:

$$\nabla J(w) = \left[ \frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, ..., \frac{\partial J}{\partial w_n} \right]$$

1. Set learning rate $\eta = 0.001$ (or other small value).
2. Start with some guess for $w^0$, set $\tau = 0$.
3. Repeat for epochs $E$ or until $J$ does not improve:
4. $\tau = \tau + 1$.
5. $w^{\tau+1} = w^\tau - \eta \nabla J(w^\tau)$
Gradient Descent: Large Updates

![Diagram showing the concept of gradient descent with large updates. The graph illustrates the movement from an initial point to a minimum point, with arrows indicating the direction of updates and the change in the cost function J.](image)
Gradient Descent: Small Updates
The Learning Rate

1. Set learning rate $\eta = 0.001$ (or other small value).
2. Start with some guess for $w^0$, set $\tau = 0$.
3. Repeat for epochs $E$ or until $J$ does not improve:
   4. $\tau = \tau + 1$.
   5. $w^{\tau+1} = w^\tau - \eta \nabla J (w^\tau)$

- How big should the learning rate be?
  - If learning rate too small => slow convergence.
  - If learning rate too big => oscillating behavior => may not even converge.
Learning Rate too Small
Learning Rate too Large
Learning Rates vs. GD Behavior

http://scs.ryerson.ca/~aharley/neural-networks/
The Learning Rate

• How big should the learning rate be?
  – If learning rate too big => oscillating behavior.
  – If learning rate too small => hinders convergence.

  o Use line search (backtracking line search, conjugate gradient, …).
  o Use second order methods (Newton’s method, L-BFGS, …).
    • Requires computing or estimating the Hessian.
  o Use a simple learning rate annealing schedule:
    – Start with a relatively large value for the learning rate.
    – Decrease the learning rate as a function of the number of epochs or as a function of the improvement in the objective.
  o Use adaptive learning rates:
    • Adagrad, Adadelta, RMSProp, Adam.
Gradient Descent: Nonconvex Objective

Cost

Local minimum  Global minimum  Plateau

Saddle point
Convex Multivariate Objective
Gradient Step and Contour Lines
Gradient Descent: Nonconvex Objectives
Gradient Descent & Plateaus
Gradient Descent & Saddle Points
Gradient Descent & Ravines
Gradient Descent & Ravines

- **Ravines** are areas where the surface curves much more steeply in one dimension than another.
  - Common around local optima.
  - GD oscillates across the slopes of the ravines, making slow progress towards the local optimum along the bottom.

- Use **momentum** to help accelerate GD in the relevant directions and dampen oscillations:
  - Add a fraction of the past *update vector* to the current update vector.
    - The momentum term increases for dimensions whose previous gradients point in the same direction.
    - It reduces updates for dimensions whose gradients change sign.
    - Also reduces the risk of getting stuck in local minima.
Gradient Descent & Momentum

Vanilla Gradient Descent:
\[ \mathbf{v}^{\tau+1} = \eta \nabla J(\mathbf{w}^{\tau}) \]
\[ \mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1} \]

Gradient Descent w/ Momentum:
\[ \mathbf{v}^{\tau+1} = \gamma \mathbf{v}^{\tau} + \eta \nabla J(\mathbf{w}^{\tau}) \]
\[ \mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \mathbf{v}^{\tau+1} \]

\( \gamma \) is usually set to 0.9 or similar.
Momentum & Nesterov Accelerated Gradient

GD with Momentum:
\[ \mathbf{v}^{\tau+1} = \gamma \mathbf{v}^\tau + \eta \nabla J(\mathbf{w}^\tau) \]
\[ \mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \mathbf{v}^{\tau+1} \]

Nesterov Accelerated Gradient:
\[ \mathbf{v}^{\tau+1} = \gamma \mathbf{v}^\tau + \eta \nabla J(\mathbf{w}^\tau - \gamma \mathbf{v}^\tau) \]
\[ \mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \mathbf{v}^{\tau+1} \]

By making an anticipatory update, NAGs prevents GD from going too fast => significant improvements when training RNNs.
Gradient Descent Optimization Algorithms

- **Momentum.**
- **Nesterov Accelerated Gradient (NAG).**
- Adaptive learning rates methods:
  - Idea is to perform larger updates for infrequent params and smaller updates for frequent params, by accumulating previous gradient values for each parameter.
    - **Adagrad:**
      - Divide update by sqrt of sum of squares of past gradients.
    - **Adadelta.**
    - **RMSProp.**
    - **Adaptive Moment Estimation** (Adam)
AdaGrad

• Optimized for problems with sparse features.

• Per-parameter learning rate: make smaller updates for params that are updated more frequently:

\[ w_i = w_i - \eta \frac{g_{t,i}}{\sqrt{\epsilon + G_{t,i}}} \]

where \( G_{t,i} = \sum_{\tau=1}^{t} g_{\tau,i}^2 \)

\[ g_{t,i} = \frac{\partial J(w)}{\partial w_i} \]

• Require less tuning of the learning rate compared with SGD.
RMSProp

- Element-wise gradient: \( g_i^t = \nabla_{w_i} J(w_t) \)
- Gradient is \( g_t = [g_1^t, g_2^t, \ldots, g_K^t] \)
- Element-wise square gradient: \( g_t^2 = g_t \circ g_t \)

RMSProp:

\[
E_t[g^2] = \gamma E_{t-1}[g^2] + (1 - \gamma) g_t^2
\]

\[
w_{t+1} = w_t - \frac{\eta}{\sqrt{E_t[g^2] + \epsilon}} g_t
\]

\( \gamma \) is usually set to 0.9, \( \eta \) is set to 0.001
Adam: Adaptive Moment Estimation

• Maintain an exponentially decaying average of past gradients (1\textsuperscript{st} m.) and past squared gradients (2\textsuperscript{nd} m.):

1) \( \mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \mathbf{g}_t \)

2) \( \mathbf{v}_t = \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t^2 \)

• Biased towards 0 during initial steps, use bias-corrected first and second order estimates:

1) \( \hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t} \)

2) \( \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t} \)
Adam: Adaptive Moment Estimation

- First and second moment:
  \[ m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \]
  \[ v_t = \beta_1 v_{t-1} + (1 - \beta_1) g_t^2 \]
- Bias-correction:
  \[ \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \text{ and } \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \]

Adam:

\[ w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t \]
Visualization

- Adagrad, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances.
  - Insofar, **Adam** might be the best overall choice.
Variants of Gradient Descent

$$w^{\tau+1} = w^{\tau} - \eta \nabla J(w^{\tau})$$

- Depending on how much data is used to compute the gradient at each step:
  - **Batch gradient descent:**
    - Use all the training examples.
  - **Stochastic gradient descent** (SGD).
    - Use one training example, update after each.
  - **Minibatch gradient descent.**
    - Use a constant number of training examples (minibatch).
Batch Gradient Descent

- Sum-of-squares error:

\[
J(w) = \frac{1}{2N} \sum_{n=1}^{N} (h_w(x^{(n)}) - t_n)^2
\]

\[
w^{\tau+1} = w^{\tau} - \eta \nabla J(w^{\tau})
\]

\[
w^{\tau+1} = w^{\tau} - \eta \frac{1}{N} \sum_{n=1}^{N} (h_w(x^{(n)}) - t_n) x^{(n)}
\]
Stochastic Gradient Descent

- Sum-of-squares error:

\[ J(w) = \frac{1}{2N} \sum_{n=1}^{N} (h_w(x^{(n)}) - t_n)^2 = \frac{1}{2N} \sum_{n=1}^{N} J(w^\tau, x^{(n)}) \]

\[ w^{\tau+1} = w^\tau - \eta \nabla J(w^\tau, x^{(n)}) \]

\[ w^{\tau+1} = w^\tau - \eta (h_w(x^{(n)}) - t_n) x^{(n)} \]

- Update parameters \( w \) after each example, sequentially:
  
  \( \Rightarrow \) the least-mean-square (LMS) algorithm.
Batch GD vs. Stochastic GD

• Accuracy:
• Time complexity:
• Memory complexity:
• Online learning:
Batch GD vs. Stochastic GD
Pre-processing Features

• Features may have very different scales, e.g. $x_1 = \text{rooms}$ vs. $x_2 = \text{size in sq ft}$.
  
  – **Right** (*different scales*): GD goes first towards the bottom of the bowl, then slowly along an almost flat valley.
  – **Left** (*scaled features*): GD goes straight towards the minimum.

![Diagram illustrating GD optimization in different scales](image-url)
Feature Scaling

• Scaling between [0, 1] or [-1, +1]:
  – For each feature $x_j$, compute $min_j$ and $max_j$ over the training examples.
  – Scale $x^{(n)}_j$ as follows:

• Scaling to standard normal distribution:
  – For each feature $x_j$, compute sample $\mu_j$ and sample $\sigma_j$ over the training examples.
  – Scale $x^{(n)}_j$ as follows:
Implementation: Gradient Checking

• Want to minimize $J(\theta)$, where $\theta$ is a scalar.

• Mathematical definition of derivative:

$$\frac{d}{d\theta} J(\theta) = \lim_{\varepsilon \to \infty} \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon}$$

• Numerical approximation of derivative:

$$\frac{d}{d\theta} J(\theta) \approx \frac{J(\theta + \varepsilon) - J(\theta - \varepsilon)}{2\varepsilon} \quad \text{where } \varepsilon = 0.0001$$
Implementation: Gradient Checking

- If \( \theta \) is a vector of parameters \( \theta_i \),
  - Compute numerical derivative with respect to each \( \theta_i \).
  - Aggregate all derivatives into numerical gradient \( G_{\text{num}}(\theta) \).

- Compare numerical gradient \( G_{\text{num}}(\theta) \) with implementation of gradient \( G_{\text{imp}}(\theta) \):

\[
\frac{\|G_{\text{num}}(\theta) - G_{\text{imp}}(\theta)\|}{\|G_{\text{num}}(\theta) + G_{\text{imp}}(\theta)\|} \leq 10^{-6}
\]