Gradient-Based Optimization Algorithms

CS115.Q11

N. H. Vinh[1] N. H. Kha[1]

[1] Department of Computer Science
University of Information and Technology

CS115 Presentation

Table of Contents

Table of Contents

Preliminaries - Notation

- x_i is the i-th input vector from the dataset.
- y_i is the corresponding label for x_i.
- N is the total number of samples.
- $\mathcal{D} = \{(\mathrm{x}_i, y_i)\}_{i=1}^{N}$ is the dataset.
- θ is the parameter of the model.
- θ^* is the optimal solution.
- η is the learning rate
- $g(\theta_t; \xi_t)$ is the stochastic gradient on random mini-batch ξ_t $(\mathbb{E}[g(\theta_t; \xi_t)] = \nabla\mathcal{L}(\theta_t; \mathcal{D})$

Preliminaries - Mathematical Formulas I

- Gradient Vector:

$$\nabla F(x) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

- Hessian Matrix:

$$\nabla^2 F(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Jacobian ($f : \mathbb{R}^n \to \mathbb{R}^m$):

$$J_f(x) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Preliminaries - Mathematical Formulas II

- Hadamard Product (element-wise product) of two vectors $\mathrm{a}, \mathrm{b} \in \mathbb{R}^n$:

$$\mathrm{a} \odot \mathrm{b} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_n b_n \end{bmatrix}$$

- Element-wise Square (Hadamard power 2):

$$\mathrm{a}^2 = \mathrm{a} \odot \mathrm{a} = \begin{bmatrix} a_1^2 \\ a_2^2 \\ \vdots \\ a_n^2 \end{bmatrix}$$

Preliminaries - Mathematical Formulas III

- Element-wise Square Root:

$$\sqrt{\mathrm{a}} = \begin{bmatrix} \sqrt{a_1} \\ \sqrt{a_2} \\ \vdots \\ \sqrt{a_n} \end{bmatrix}$$

Preliminaries - Loss Gradient Function I

The gradient function:

$$\nabla L(\theta) = \frac{2}{m}\mathrm{x}^{\top}(\mathrm{x}\theta + y)$$

```python
def grad_func(X, y, theta):
    m = len(y)
    gradients = (2/m) * X.T.dot(X.dot(theta) - y)
    return gradients
```

Table of Contents

First-Order Methods

Gradient Descent

Gradient Descent

Update formula:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t, \mathcal{D}),$$

First-Order Methods

Gradient Descent

```
Algorithm 1 Gradient Descent
Require: $\eta$: Learning rate
Require: $\mathcal{L}(\theta)$: Loss function
Require: $\theta_0$: Initial parameter
$t \leftarrow 0$
while not converged do                              ▷ Initialize step
    $g_t \leftarrow \nabla_\theta \mathcal{L}(\theta_t)$            ▷ Compute gradient
    $\theta_{t+1} \leftarrow \theta_t - \eta g_t$                ▷ Update parameter
    $t \leftarrow t + 1$                               ▷ Update step
end while
return $\theta_{t+1}$
```

First-Order Methods

Gradient Descent

```
def gradient_descent(grad_fn, theta0, eta, T, X, y):
    for t in range(T):
        g = grad_func(X, y, theta)
        theta0[:] = theta0 - eta * g
```

First-Order Methods

Gradient Descent

Problems:

- Use all training data, can be really **inefficient** with huge dataset.
- Cannot escape **local minima, saddle points**.
- The **lack of generalization** ability is due to the fact that large-batch methods tend to converge to sharp minimizers of the training function.

First-Order Methods

Stochastic Gradient Descent

Stochastic Gradient Descent

Update formula:

$$\theta_{t+1} = \theta_t - \eta g(\theta_t, \xi_t),$$

- Uses only a small subset of data (mini-batch or single sample) at each update, leading to **faster computation**.
- Adds **stochastic noise** to the gradient, which helps the optimizer **escape local minima** and explore the loss landscape.

First-Order Methods

Stochastic Gradient Descent

Algorithm 2 Stochastic Gradient Descent (SGD)

```
Require: $\eta$: Learning rate
Require: $\mathcal{L}(\theta)$: Loss function
Require: $\theta_0$: Initial parameter
$t \leftarrow 0$
while not converged do
    Sample $\xi_t$                                    ▷ Randomly chosen data point
    $g_t \leftarrow \nabla_\theta \mathcal{L}(\theta_t; \xi_t)$              ▷ Stochastic gradient
    $\theta_{t+1} \leftarrow \theta_t - \eta g_t$                       ▷ Parameter update
    $t \leftarrow t + 1$
end while
return $\theta_{t+1}$
```

First-Order Methods

Stochastic Gradient Descent

```
def stochastic_gradient_descent(grad_fn, theta0, eta, T, X, y):
    theta = theta0
    m = len(y)
    for t in range(T):
        i = np.random.randint(0, m)
        X_i = X[i:i+1]
        y_i = y[i:i+1]
        g = m*grad_func(X_i, y_i, theta)
        theta = theta - eta * g
    return theta
```

First-Order Methods

Stochastic Gradient Descent

Problems:

- Gradient estimates are noisy ⇒ cause oscillations near the optimum.
- High variance in updates ⇒ slower and less stable convergence.
- May get trapped in local minima or saddle points.
- Highly sensitive to learning rate and not adaptive to feature scaling.

First-Order Methods - Momentum-Based Methods

Heavy-Ball

Heavy Ball

Update formula:

$$\theta_{t+1} = \theta_t - \eta g(\theta_t; \xi_t) + \beta(\theta_t - \theta_{t-1})$$

where

- $0 \leq \beta < 1$: momentum coefficient (scales the previous velocity),

- Introduce momentum ⇒ Faster convergence.
- Introduce momentum ⇒ Help escape local minima.

First-Order Methods - Momentum-Based Methods

Heavy-Ball

Problems:

- How does β affect the optimizer?
- How does velocity affect the optimizer?

First-Order Methods - Momentum-Based Methods

Nestrov Accelerated Gradient

NAG - Nestrov Accelerated Gradient

Update formula:

$$\begin{cases} \gamma_{k+1} = \theta_k - \eta g(\theta_k; \xi_k) \\ \theta_{k+1} = \gamma_{k+1} + \beta(\gamma_{k+1} - \gamma_k) \end{cases}$$

with initial velocity $v_0 = 0$, where

- $0 \leq \beta < 1$: momentum coefficient (scales the previous velocity),

- **Faster convergence:** anticipates future position, correcting the direction earlier than standard momentum.
- **Reduced oscillations:** smoother trajectory near minima thanks to lookahead gradient.
- **Better stability:** less overshooting and improved performance on curved loss surfaces.

First-Order Methods - Momentum-Based Methods

Nestrov Accelerated Gradient

Problems:

- Fine-tune hyperparameters.

First-Order Methods - Momentum-Based Methods

Unified Momentum

Unified Momentum

Update formula:

$$\text{UM}: \begin{cases} \gamma_{t+1} = \theta_t - \eta g(\theta_t; \xi_t) \\ \gamma_{t+1}^s = \theta_t - s\eta g(\theta_t; \xi_t) \\ \theta_{t+1} = \gamma_{t+1} + \beta\left(\gamma_{t+1}^s - \gamma_t^s\right) \end{cases}$$

where

- $0 \leq \beta < 1$: momentum coefficient,
- $s \geq 0$: parameter controlling the variant (e.g., $s = 0$ gives Heavy Ball, $s = 1$ gives NAG, $s = 1/\beta$ gives SGD),

First-Order Methods - Momentum-Based Methods

Unified Momentum

```
Algorithm 3 Unified Momentum (UM)
Require: $\eta$: Learning rate
Require: $\beta$: Momentum coefficient
Require: $s$: Scaling factor
Require: $\mathcal{L}(\theta)$: Loss function
Require: $\theta_0$: Initial parameter
$t \leftarrow 0$
while not converged do
    $\gamma_{t+1} \leftarrow \theta_t - \eta\, g(\theta_t; \xi_t)$
    $\gamma^s_{t+1} \leftarrow \theta_t - s\eta\, g(\theta_t; \xi_t)$
    $\theta_{t+1} \leftarrow \gamma_{t+1} + \beta(\gamma^s_{t+1} - \gamma^s_t)$
    $t \leftarrow t + 1$
end while
return Final parameter $\theta_T$
```

First-Order Methods - Momentum-Based

Unified Momentum

```python
def unified_momentum(grad_func, theta0, eta, beta, s, T, X, y):
    gamma = theta0.copy()
    gamma_s = theta0.copy()
    m = len(y)
    for t in range(T):
        i = np.random.randint(0, m)
        X_i = X[i:i+1]
        y_i = y[i:i+1]
        g = grad_func(X_i, y_i, theta0)
        gamma_next = theta0 - eta * g
        gamma_s_next = theta0 - s * eta * g
        theta0[:] = gamma_next + beta * (gamma_s_next - gamma_s)
        gamma[:] = gamma_next
        gamma_s[:] = gamma_s_next
```

First-Order Methods - Adaptive Learning Rate Methods

AdaGrad

AdaGrad

Update formula:

$$\text{AdaGrad}: \begin{cases} \gamma_{t+1} = \gamma_t + \big(g(\theta_t;\xi_t)\big)^2 \\ \theta_t = \theta_t - \dfrac{\eta}{\sqrt{\gamma_t + \epsilon}} g(\theta_t;\xi_t) \end{cases}$$

where

- ϵ helps control numerical stability

- Automatically rescales the gradient along each parameter direction.

First-Order Methods - Adaptive Learning Rate Methods

AdaGrad

Algorithm 4 Adagrad

```
Require: Initial parameters $\theta_0$, learning rate $\eta$
Require: Number of iterations $T$, stochastic gradient function $g(\theta;\xi)$
Require: small constant $\epsilon$
$\gamma_0 \leftarrow 0,\ t \leftarrow 1,\ \theta_0 \leftarrow 0$        ▷ Initialize accumulated squared gradients
while not converged do
    $g_t \leftarrow g(\theta_t;\xi_t)$                          ▷ Compute stochastic gradient
    $\gamma_t \leftarrow \gamma_{t-1} + g_t \odot g_t$                ▷ Element-wise square accumulation
    $\theta_t \leftarrow \theta_{t-1} - \eta \frac{g_t}{\sqrt{\gamma_t}+\epsilon}$          ▷ Update parameters
end while
return $\theta_t$
```

First-Order Methods - Adaptive Learning Rate Methods

AdaGrad

```
def adagrad(grad_func, theta0, eta, T, X, y, eps=1e-8):
    gamma = np.zeros_like(theta0)
    m = len(y)

    for t in range(T):
        i = np.random.randint(0, m)
        X_i = X[i:i+1]
        y_i = y[i:i+1]
        g = m*grad_func(X_i, y_i, theta0)

        gamma[:] = gamma + g**2
        theta0[:] = theta0 - eta * g / (np.sqrt(gamma) + eps)
```

First-Order Methods - Adaptive Learning Rate Methods

AdaGrad

Problems:

- Accumulating the squared gradients over time.

First-Order Methods - Adaptive Learning Rate Methods

RMSProp

RMSProp

Update formula:

$$\text{RMSProp}: \begin{cases} \gamma_{t+1} = \alpha\gamma_t + (1-\alpha)\big(g(\theta_t;\xi_t)\big)^2 \\ \theta_{t+1} = \theta_t - \dfrac{\eta}{\sqrt{\gamma_t + \epsilon}} \odot g(\theta_t;\xi_t) \end{cases}$$

where

- ϵ helps control numerical stability,
- α is the smooth constant
- Exponentially decay the influence of past gradients.

First-Order Methods - Adaptive Learning Rate Methods

RMSProp

Algorithm 5 RMSProp

```
Require: Initial parameters $\theta_0$, learning rate $\eta$
Require: Number of iterations $T$, stochastic gradient function $g(\theta;\xi)$
Require: Smoothing constant $\alpha$, small constant $\epsilon$
$\gamma_0 \leftarrow 0,\ t \leftarrow 1,\ \theta_0 \leftarrow 0$    ▷ Initialize accumulated squared gradients
while not converged do
    $g_t \leftarrow g(\theta_t;\xi_t)$    ▷ Compute stochastic gradient
    $\gamma_{t+1} \leftarrow \alpha\gamma_t + (1-\alpha)g_t \odot g_t$    ▷ Update running average
    $\theta_{t+1} \leftarrow \theta_t - \eta\frac{g_t}{\sqrt{\gamma_t}+\epsilon}$    ▷ Update parameters
    $t \leftarrow t+1$
end while
return $\theta_t$
```

First-Order Methods - Adaptive Learning Rate Methods

RMSProp

```python
def rmsprop(grad_func, theta, eta, T, X, y, rho=0.9, eps=1e-8):
    gamma = np.zeros_like(theta)
    m_data = len(y)

    for t in range(T):
        i = np.random.randint(0, m_data)
        X_i = X[i:i+1]
        y_i = y[i:i+1]
        g_t = grad_func(X_i, y_i, theta)

        gamma[:] = rho * gamma + (1 - rho) * (g_t ** 2)
        theta[:] = theta - eta * g_t / (np.sqrt(gamma) + eps)

    return theta
```

First-Order Methods - Adaptive Learning Rate Methods

AdaDelta

AdaDelta

Update formula:

$$\text{AdaDelta}: \begin{cases} \gamma_{t+1} = \rho\,\gamma_t + (1-\rho)\left(g(\theta_t;\xi_t)\right)^2 \\ g'_{t+1} = \frac{\sqrt{\Delta\theta_t + \epsilon}}{\sqrt{\gamma_{t+1} + \epsilon}} \odot g(\theta_t, \xi_t) \\ \Delta\theta_{t+1} = \rho\Delta\theta_t + (1-\rho) g'^2_{t+1} \\ \theta_{t+1} = \theta_t - \eta g'_{t+1} \end{cases}$$

where

- ϵ helps control numerical stability
- ρ is the coefficient used for computing a running average of squared gradients

First-Order Methods - Adaptive Learning Rate Methods

AdaDelta

Algorithm 6 Adadelta

```
Require: Initial parameters $\theta_0$, learning rate $\eta$
Require: Decay rate $\rho$, small constant $\varepsilon$
$\gamma_0 \leftarrow 0$, $\Delta\theta_0 \leftarrow 0$, $t \leftarrow 0$    ▷ Initialize variables
while not converged do
    $g_t \leftarrow g(\theta_t; \xi_t)$    ▷ Compute stochastic gradient
    $\gamma_{t+1} \leftarrow \rho\,\gamma_t + (1-\rho)\,(g_t \odot g_t)$    ▷ Accumulate gradient energy
    $g'_{t+1} \leftarrow \frac{\sqrt{\Delta\theta_t+\varepsilon}}{\sqrt{\gamma_{t+1}+\varepsilon}} \odot g_t$    ▷ Compute scaled gradient
    $\Delta\theta_{t+1} \leftarrow \rho\,\Delta\theta_t + (1-\rho)\,(g'_{t+1} \odot g'_{t+1})$    ▷ Update energy
    $\theta_{t+1} \leftarrow \theta_t - \eta\, g'_{t+1}$    ▷ Update parameters
    $t \leftarrow t+1$    ▷ Increment iteration counter
end while
return $\theta_t$
```

First-Order Methods - Adaptive Learning Rate Methods

AdaDelta

```python
def adadelta(grad_func, theta, T, X, y, rho=0.95, eps=1e-6):
    gamma = np.zeros_like(theta)
    delta_theta = np.zeros_like(theta)
    m = len(y)

    for t in range(T):
        i = np.random.randint(0, m)
        X_i = X[i:i+1]
        y_i = y[i:i+1]
        g_t = grad_func(X_i, y_i, theta)

        gamma = rho * gamma + (1 - rho) * (g_t**2)
        scaled_grad = (np.sqrt(delta_theta + eps) / np.sqrt(gamma + eps)) * g_t
        theta = theta - scaled_grad
        delta_theta = rho * delta_theta + (1 - rho) * (scaled_grad**2)

    return theta
```

First-Order Methods - Adaptive Learning Rate Methods

Adam

Adam

Update formula:

$$\text{Adam}: \begin{cases} m_{t+1} = \beta_1 m_t + g(\theta_t; \xi_t) \\ \gamma_{t+1} = \beta_2 \gamma_t + (1 - \beta_2)(g(\theta_t; \xi_t))^2 \\ \theta_{t+1} = \theta_t - \eta \frac{m_{t+1}}{1-\beta_1^t} \frac{1}{\sqrt{\frac{\gamma_t}{1-\beta_2^t}} + \epsilon} \end{cases}$$

where

- $0 \leq \beta_1, \beta_2 < 1$: coefficients used for computing running averages of gradient and its square

First-Order Methods - Adaptive Learning Rate Methods

Adam

Algorithm 7 Adam

```
Require: Initial parameters $\theta_0$, learning rate $\eta$
Require: Exponential decay rates $\beta_1, \beta_2$, small constant $\epsilon$
$m_0 \leftarrow 0,\ \gamma_0 \leftarrow 0,\ t \leftarrow 0$    ▷ Initialize variables
while not converged do
    $g_t \leftarrow g(\theta_t; \xi_t)$    ▷ Compute gradient
    $m_{t+1} \leftarrow \beta_1 m_t + (1 - \beta_1) g_t$    ▷ Update biased first moment
    $\gamma_{t+1} \leftarrow \beta_2 \gamma_t + (1 - \beta_2)(g_t \odot g_t)$    ▷ Update biased second moment
    $\hat{m}_{t+1} \leftarrow \frac{m_{t+1}}{1-\beta_1^{t+1}}, \quad \hat{\gamma}_{t+1} \leftarrow \frac{\gamma_{t+1}}{1-\beta_2^{t+1}}$    ▷ Bias-corrected estimates
    $\theta_{t+1} \leftarrow \theta_t - \eta \frac{\hat{m}_{t+1}}{\sqrt{\hat{\gamma}_{t+1}} + \epsilon}$    ▷ Update parameters
    $t \leftarrow t + 1$    ▷ Increment iteration counter
end while
return $\theta_t$
```

First-Order Methods - Adaptive Learning Rate Methods

Adam

```python
def adam(grad_func, theta, eta, T, X, y, beta1=0.9, beta2=0.999, eps=1e-8):
    m = np.zeros_like(theta)
    gamma = np.zeros_like(theta)
    m_data = len(y)

    for t in range(1, T+1):
        i = np.random.randint(0, m_data)
        X_i = X[i:i+1]
        y_i = y[i:i+1]
        g_t = grad_func(X_i, y_i, theta)

        m[:] = beta1 * m + (1 - beta1) * g_t
        gamma[:] = beta2 * gamma + (1 - beta2) * (g_t ** 2)

        m_hat = m / (1 - beta1**t)
        gamma_hat = gamma / (1 - beta2**t)

        theta[:] = theta - eta * m_hat / (np.sqrt(gamma_hat) + eps)

    return theta
```

Table of Contents

Second-Order Methods - Newton's Method

Newton's Method

Newton's Update Formula

$$\theta_{t+1} = \theta_t - H^{-1}(\theta_t; \mathcal{D}) \nabla \mathcal{L}(\theta_t; \mathcal{D})$$

where:

- $H(\theta_t; \mathcal{D}) = \nabla^2 \mathcal{L}(\theta_t; \mathcal{D})$, the Hessian matrix of the loss function $\mathcal{L}(\theta_t; \mathcal{D})$ with respect to θ_t.

Second-Order Methods - Newton's Method

Newton's Method

Algorithm 8 Newton's Method

```
Require: Initial parameters $\theta_0$, maximum iterations $T$
Require: Objective function $J(\theta; \mathcal{D})$, dataset $\mathcal{D}$
$t \leftarrow 0$
while not converged do
    Compute gradient: $g_t \leftarrow \nabla J(\theta_t; \mathcal{D})$
    Compute Hessian: $H_t \leftarrow \nabla^2 J(\theta_t; \mathcal{D})$
    Update parameters: $\theta_{t+1} \leftarrow \theta_t - H_t^{-1} g_t$
    $t \leftarrow t + 1$
end while
return $\theta_t$
```

Second-Order Methods - Quasi-Newton Methods

BFGS

Formula:

$$\theta_{t+1} = \theta_t - \eta_t \, B_t^{-1} \nabla \mathcal{L}(\theta_t; \mathcal{D})$$

- B_t is pseudo-Hessian matrix Hessian $H_t = \nabla^2 \mathcal{L}(\theta_t; \mathcal{D})$.

Second-Order Methods - Quasi-Newton Methods

BFGS

Algorithm 9 BFGS Algorithm

```
Require: Initial parameters $\theta_0$,
Require: initial inverse Hessian approximation $B_0^{-1} = I$, learning rate $\eta_t$
t ← 0
while not converged do
    Compute gradient: $g_t \leftarrow \nabla J(\theta_t; \mathcal{D})$
    Update parameters: $\theta_{t+1} \leftarrow \theta_t - \eta_t B_t^{-1} g_t$
    Compute $s_t \leftarrow \theta_{t+1} - \theta_t$
    Compute $y_t \leftarrow g_{t+1} - g_t$
    $B_{t+1}^{-1} \leftarrow \left(I - \frac{s_t y_t^\top}{y_t^\top s_t}\right) B_t^{-1} \left(I - \frac{y_t s_t^\top}{y_t^\top s_t}\right) + \frac{s_t s_t^\top}{y_t^\top s_t}$
    $t \leftarrow t + 1$
end while
return $\theta_t$
```

Second-Order Methods - Quasi-Newton Methods

LBFGS

Sorry, this section is too hard for us to present. We may need Mrs Hang to teach us about this.

Table of Contents

Future Work

- Explore more methods.
- Convergence Analysis
- Other methods of optimization.