Optimization Techniques for ML (2)

Piyush Rai

Introduction to Machine Learning (CS771A)

August 28, 2018

Recap: Convex and Non-Convex Function

- Most ML problems boil down to minimization of convex/non-convex functions, e.g.,

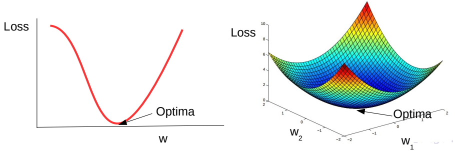
$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} \frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w}) + R(\boldsymbol{w})$$

Recap: Convex and Non-Convex Function

- Most ML problems boil down to minimization of convex/non-convex functions, e.g.,

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} \frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w}) + R(\boldsymbol{w})$$

- Convex functions have a unique minima

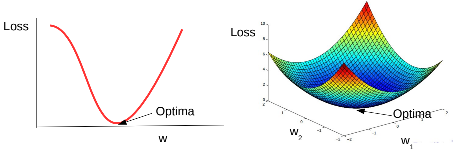


Recap: Convex and Non-Convex Function

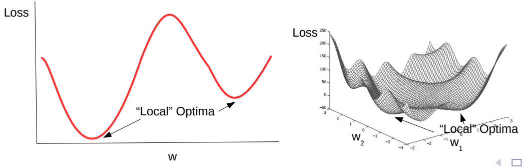
- Most ML problems boil down to minimization of convex/non-convex functions, e.g.,

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}} \frac{1}{N}\sum_{n=1}^{N} \ell_n(\boldsymbol{w}) + R(\boldsymbol{w})$$

- Convex functions have a unique minima



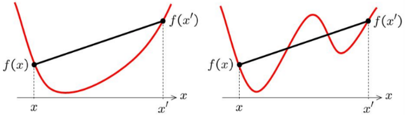
- Non-convex function have several local minima



Recap: Convex Functions

- A function is convex if all of its chords lie above the function

Convex Function Non-convex Function



Note: "Chord lies above function" more formally means

If f is convex then given

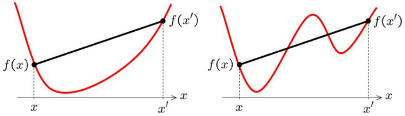
$\alpha_1, \ldots, \alpha_n$ s.t $\sum_{i=1}^{n} \alpha_i = 1$

$$f\left(\sum_{i=1}^{n} \alpha_i x_i\right) \leq \sum_{i=1}^{n} \alpha_i f(x_i)$$

Jensen's Inequality

Recap: Convex Functions

- A function is convex if all of its chords lie above the function



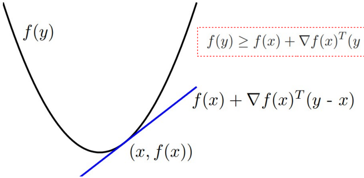
Note: "Chord lies above function" more formally means

If f is convex then given $\alpha_1, \ldots, \alpha_n$ s.t $\sum_{i=1}^n \alpha_i = 1$

$$f\left(\sum_{i=1}^n \alpha_i x_i\right) \leq \sum_{i=1}^n \alpha_i f(x_i)$$

Jensen's Inequality

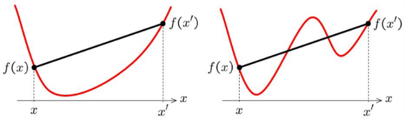
- A function is convex if its graph lies above all of its tangents (above its first order Taylor expansion)



Recap: Convex Functions

- A function is convex if all of its chords lie above the function

Convex Function — Non-convex Function



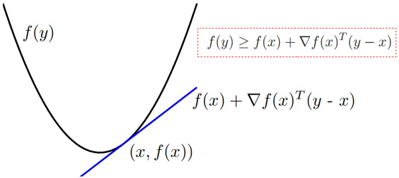
Note: "Chord lies above function" more formally means

If f is convex then given $\alpha_1, \ldots, \alpha_n$ s.t $\sum_{i=1}^{n} \alpha_i = 1$

$$f\Big(\sum_{i=1}^{n} \alpha_i x_i\Big) \leq \sum_{i=1}^{n} \alpha_i f(x_i)$$

Jensen's Inequality

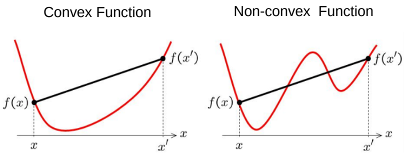
- A function is convex if its graph lies above all of its tangents (above its first order Taylor expansion)



- A function is convex if its second derivative (Hessian) is positive semi-definite

Recap: Convex Functions

- A function is convex if all of its chords lie above the function



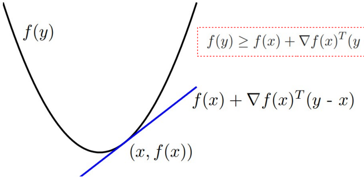
Note: "Chord lies above function" more formally means

If f is convex then given $\alpha_1, \ldots, \alpha_n$ s.t $\sum_{i=1}^{n} \alpha_i = 1$

$$f\left(\sum_{i=1}^{n} \alpha_i x_i\right) \leq \sum_{i=1}^{n} \alpha_i f(x_i)$$

Jensen's Inequality

- A function is convex if its graph lies above all of its tangents (above its first order Taylor expansion)

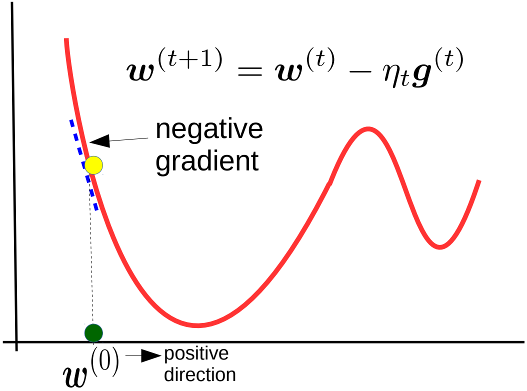


$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

- A function is convex if its second derivative (Hessian) is positive semi-definite
- Note: If f is convex then $-f$ is a concave function

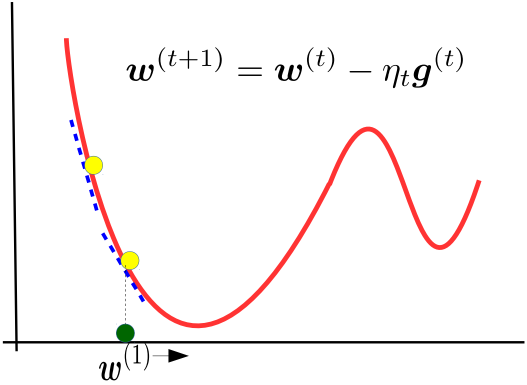
Recap: Gradient Descent

- A very simple, first-order method for optimizing any differentiable function (convex/non-convex)
- Uses only the gradient $\boldsymbol{g} = \nabla\mathcal{L}(\boldsymbol{w})$ of the function
- Basic idea: Start at some location $\boldsymbol{w}^{(0)}$ and move in the opposite direction of the gradient



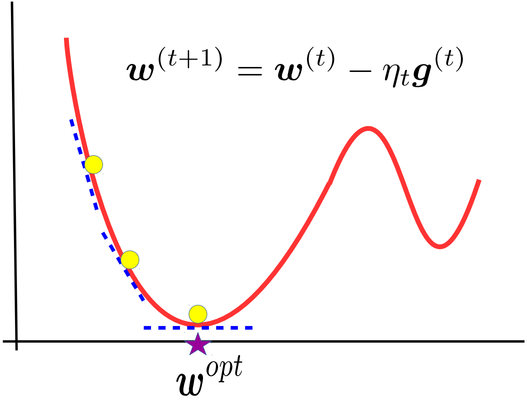
Recap: Gradient Descent

- A very simple, first-order method for optimizing any differentiable function (convex/non-convex)
- Uses only the gradient $\boldsymbol{g} = \nabla \mathcal{L}(\boldsymbol{w})$ of the function
- Basic idea: Start at some location $\boldsymbol{w}^{(0)}$ and move in the opposite direction of the gradient



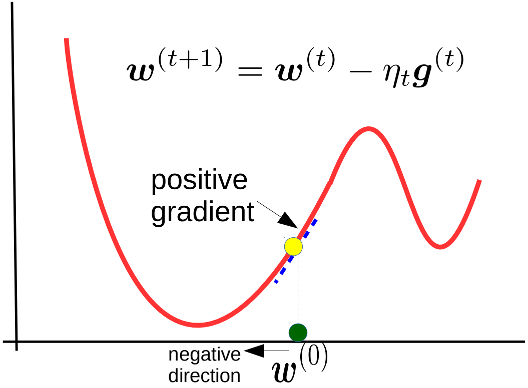
Recap: Gradient Descent

- A very simple, first-order method for optimizing any differentiable function (convex/non-convex)
- Uses only the gradient $\boldsymbol{g} = \nabla\mathcal{L}(\boldsymbol{w})$ of the function
- Basic idea: Start at some location $\boldsymbol{w}^{(0)}$ and move in the opposite direction of the gradient



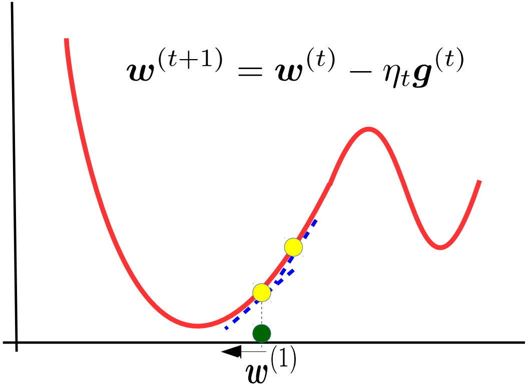
Recap: Gradient Descent

- A very simple, first-order method for optimizing any differentiable function (convex/non-convex)
- Uses only the gradient $\boldsymbol{g} = \nabla \mathcal{L}(\boldsymbol{w})$ of the function
- Basic idea: Start at some location $\boldsymbol{w}^{(0)}$ and move in the opposite direction of the gradient



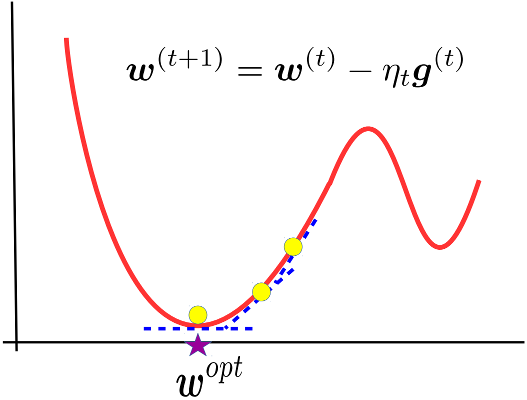
Recap: Gradient Descent

- A very simple, first-order method for optimizing any differentiable function (convex/non-convex)
- Uses only the gradient $\boldsymbol{g} = \nabla \mathcal{L}(\boldsymbol{w})$ of the function
- Basic idea: Start at some location $\boldsymbol{w}^{(0)}$ and move in the opposite direction of the gradient



Recap: Gradient Descent

- A very simple, first-order method for optimizing any differentiable function (convex/non-convex)
- Uses only the gradient $\boldsymbol{g} = \nabla \mathcal{L}(\boldsymbol{w})$ of the function
- Basic idea: Start at some location $\boldsymbol{w}^{(0)}$ and move in the opposite direction of the gradient



Gradient Descent

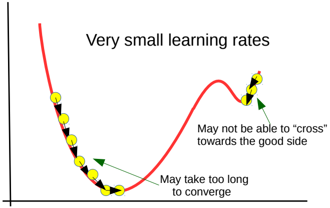
1. Initialize $\boldsymbol{w}$ as $\boldsymbol{w}^{(0)}$
2. Update $\boldsymbol{w}$ as follows

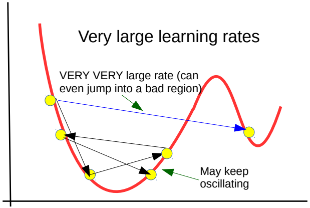
$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

3. Repeat until convergence

Recap: Gradient Descent

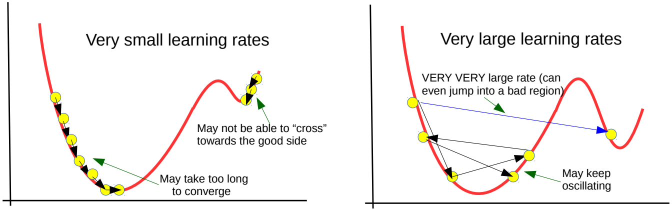
- The learning rate η_t is important
- Very small learning rates may result in very slow convergence
- Very large learning rates may lead to oscillatory behavior or result in a bad local optima





Recap: Gradient Descent

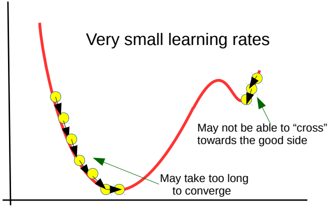
- The learning rate η_t is important
- Very small learning rates may result in very slow convergence
- Very large learning rates may lead to oscillatory behavior or result in a bad local optima

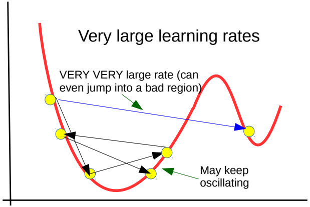


- Many ways to set the learning rate, e.g.,

Recap: Gradient Descent

- The learning rate η_t is important
- Very small learning rates may result in very slow convergence
- Very large learning rates may lead to oscillatory behavior or result in a bad local optima





- Many ways to set the learning rate, e.g.,
  - Constant (if properly set, can still show good convergence behavior)
  - Decreasing with t (e.g. $1/t$, $1/\sqrt{t}$, etc.)
  - Use adaptive learning rates (e.g., using methods such as Adagrad, Adam)

Recap: Stochastic Gradient Descent

- Gradient computation in standard GD may be expensive when N is large

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} \left[\frac{1}{N} \sum_{n=1}^{N} \ell_n(\boldsymbol{w}) \right] = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{g}_n \qquad \text{(ignoring regularizer } R(\boldsymbol{w})\text{)}$$

Recap: Stochastic Gradient Descent

- Gradient computation in standard GD may be expensive when N is large

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} \left[\frac{1}{N} \sum_{n=1}^{N} \ell_n(\boldsymbol{w}) \right] = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{g}_n \qquad \text{(ignoring regularizer } R(\boldsymbol{w})\text{)}$$

- Stochastic Gradient Descent (SGD) approximates $\boldsymbol{g}$ using a single data point

Recap: Stochastic Gradient Descent

- Gradient computation in standard GD may be expensive when N is large

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} \left[\frac{1}{N} \sum_{n=1}^{N} \ell_n(\boldsymbol{w}) \right] = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{g}_n \qquad \text{(ignoring regularizer } R(\boldsymbol{w})\text{)}$$

- Stochastic Gradient Descent (SGD) approximates $\boldsymbol{g}$ using a single data point
- In iteration t, SGD picks a uniformly random $i \in \{1, \ldots, N\}$ and approximate $\boldsymbol{g}$ as

$$\boldsymbol{g} \approx \boldsymbol{g}_i = \nabla_{\boldsymbol{w}} \ell_i(\boldsymbol{w})$$

Recap: Stochastic Gradient Descent

- Gradient computation in standard GD may be expensive when N is large

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}} \left[\frac{1}{N} \sum_{n=1}^{N} \ell_n(\boldsymbol{w}) \right] = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{g}_n \qquad \text{(ignoring regularizer } R(\boldsymbol{w}))$$

- Stochastic Gradient Descent (SGD) approximates $\boldsymbol{g}$ using a single data point
- In iteration t, SGD picks a uniformly random $i \in \{1, \ldots, N\}$ and approximate $\boldsymbol{g}$ as

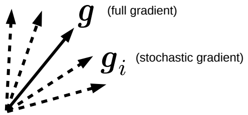
$$\boldsymbol{g} \approx \boldsymbol{g}_i = \nabla_{\boldsymbol{w}} \ell_i(\boldsymbol{w})$$

Stochastic Gradient Descent

1. Initialize $\boldsymbol{w}$ as $\boldsymbol{w}^{(0)}$
2. Pick a random $i \in \{1, \ldots, N\}$. Update $\boldsymbol{w}$ as follows
$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}_i^{(t)}$$
3. Repeat until convergence

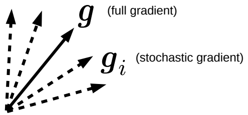
Recap: Mini-batch SGD

- In each itearation, SGD uses a single randomly chosen $i \in \{1, \ldots, N\}$ to approximate $\boldsymbol{g}$
- This results in a large variance in $\boldsymbol{g}_i$



Recap: Mini-batch SGD

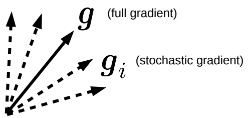
- In each itearation, SGD uses a single randomly chosen $i \in \{1, \ldots, N\}$ to approximate $\boldsymbol{g}$
- This results in a large variance in $\boldsymbol{g}_i$



- We can instead use $B > 1$ uniformly randomly chosen points with indices $i_1, \ldots, i_B \in \{1, \ldots, N\}$

Recap: Mini-batch SGD

- In each itearation, SGD uses a single randomly chosen $i \in \{1, \ldots, N\}$ to approximate $\boldsymbol{g}$
- This results in a large variance in $\boldsymbol{g}_i$

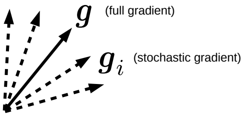


- We can instead use $B > 1$ uniformly randomly chosen points with indices $i_1, \ldots, i_B \in \{1, \ldots, N\}$
- This is the idea behind mini-batch SGD. The approximated gradient in this case would be

$$\boldsymbol{g} \approx \frac{1}{B} \sum_{b=1}^{B} \boldsymbol{g}_{i_b}$$

Recap: Mini-batch SGD

- In each itearation, SGD uses a single randomly chosen $i \in \{1, \ldots, N\}$ to approximate $\boldsymbol{g}$
- This results in a large variance in $\boldsymbol{g}_i$



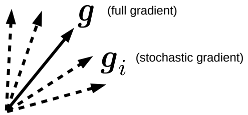
- We can instead use $B > 1$ uniformly randomly chosen points with indices $i_1, \ldots, i_B \in \{1, \ldots, N\}$
- This is the idea behind mini-batch SGD. The approximated gradient in this case would be

$$\boldsymbol{g} \approx \frac{1}{B} \sum_{b=1}^{B} \boldsymbol{g}_{i_b}$$

- The basic intuition: Averaging helps in variance reduction!

Recap: Mini-batch SGD

- In each itearation, SGD uses a single randomly chosen $i \in \{1, \ldots, N\}$ to approximate $\boldsymbol{g}$
- This results in a large variance in $\boldsymbol{g}_i$



- We can instead use $B > 1$ uniformly randomly chosen points with indices $i_1, \ldots, i_B \in \{1, \ldots, N\}$
- This is the idea behind mini-batch SGD. The approximated gradient in this case would be

$$\boldsymbol{g} \approx \frac{1}{B} \sum_{b=1}^{B} \boldsymbol{g}_{i_b}$$

- The basic intuition: Averaging helps in variance reduction!
- The algorithm is same as SGD except we will now using these mini-batch gradients at each step

Plan for today

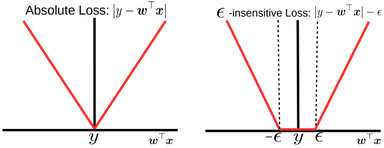
- Optimization of functions that are NOT differentiable
- Optimization with constraints on the variables
- Optimizing w.r.t. several variables with one at a time
  - Co-ordinate descent
  - Alternating optimization
- Second-order methods for optimization

Optimizing Non-differentiable Functions

- Many ML problems require minimizing non-differentiable functions

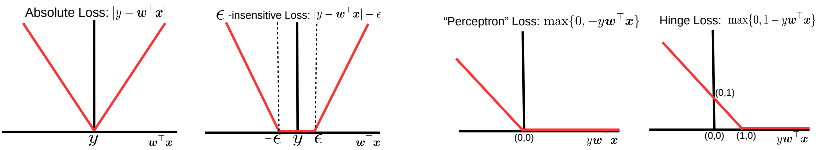
Optimizing Non-differentiable Functions

- Many ML problems require minimizing non-differentiable functions
- Some common examples
  - Absolute, ϵ-insensitive loss in regression, several classification loss functions (we will see shortly)



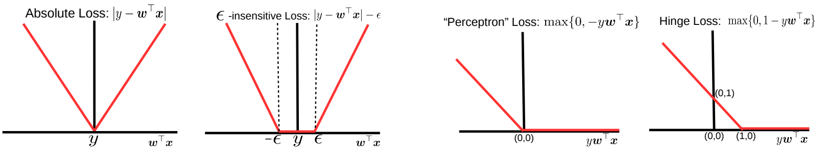
Optimizing Non-differentiable Functions

- Many ML problems require minimizing non-differentiable functions
- Some common examples
  - Absolute, ϵ-insensitive loss in regression, several classification loss functions (we will see shortly)

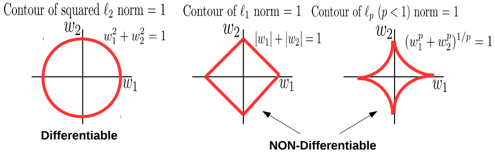


Optimizing Non-differentiable Functions

- Many ML problems require minimizing non-differentiable functions
- Some common examples
  - Absolute, ϵ-insensitive loss in regression, several classification loss functions (we will see shortly)

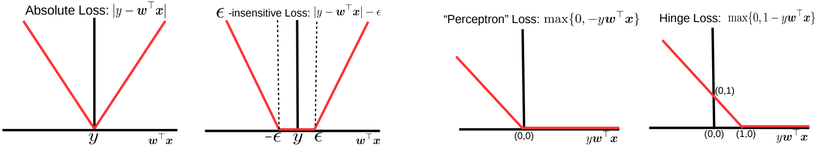


  - Regression/classification loss functions with ℓ_1 or ℓ_p ($p < 1$) regularization

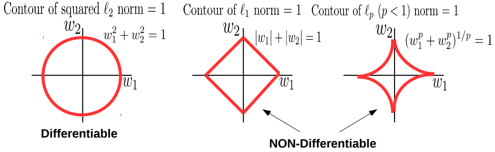


Optimizing Non-differentiable Functions

- Many ML problems require minimizing non-differentiable functions
- Some common examples
  - Absolute, ϵ-insensitive loss in regression, several classification loss functions (we will see shortly)



  - Regression/classification loss functions with ℓ_1 or ℓ_p $(p < 1)$ regularization



- Can't apply standard GD or SGD since gradient isn't defined at points of non-differentiability

Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$), some common loss functions are

$$\ell(y, \hat{y}) = (y - \boldsymbol{w}^\top \boldsymbol{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \boldsymbol{w}^\top \boldsymbol{x}|$$

Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$), some common loss functions are

$$\ell(y, \hat{y}) \quad = \quad (y - \boldsymbol{w}^\top \boldsymbol{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \boldsymbol{w}^\top \boldsymbol{x}|$$

- We typically look at the difference between true y and model's prediction $\boldsymbol{w}^\top \boldsymbol{x}$

Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$), some common loss functions are

$$\ell(y, \hat{y}) \quad = \quad (y - \boldsymbol{w}^\top \boldsymbol{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \boldsymbol{w}^\top \boldsymbol{x}|$$

- We typically look at the difference between true y and model's prediction $\boldsymbol{w}^\top \boldsymbol{x}$
- How to formally define loss functions for classification?

Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$), some common loss functions are

$$\ell(y, \hat{y}) = (y - \boldsymbol{w}^\top \boldsymbol{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \boldsymbol{w}^\top \boldsymbol{x}|$$

- We typically look at the difference between true y and model's prediction $\boldsymbol{w}^\top \boldsymbol{x}$
- How to formally define loss functions for classification?
- We have already looked at the loss function for logistic regression (assuming $y \in \{-1, +1\}$)

$$\ell(y, \hat{y}) = \log(1 + \exp(-y\boldsymbol{w}^\top \boldsymbol{x}))$$

Interlude: Loss Functions for Classification

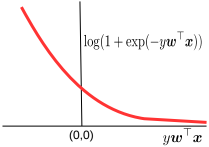
- In regression (assuming linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$), some common loss functions are

$$\ell(y, \hat{y}) = (y - \boldsymbol{w}^\top \boldsymbol{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \boldsymbol{w}^\top \boldsymbol{x}|$$

- We typically look at the difference between true y and model's prediction $\boldsymbol{w}^\top \boldsymbol{x}$
- How to formally define loss functions for classification?
- We have already looked at the loss function for logistic regression (assuming $y \in \{-1, +1\}$)

$$\ell(y, \hat{y}) = \log(1 + \exp(-y\boldsymbol{w}^\top \boldsymbol{x}))$$

- Why does the above make sense? Well, it is large for large misclassifications, small otherwise



if y=+1, then large positive $\boldsymbol{w}^\top \boldsymbol{x}$ implies small error
if y=-1, then large negative $\boldsymbol{w}^\top \boldsymbol{x}$ implies small error
if y=+1, then large negative $\boldsymbol{w}^\top \boldsymbol{x}$ implies large error
if y=-1, then large positive $\boldsymbol{w}^\top \boldsymbol{x}$ implies large error

Large positive $y\boldsymbol{w}^\top \boldsymbol{x} \Rightarrow$ small error, Large negative $y\boldsymbol{w}^\top \boldsymbol{x} \Rightarrow$ large error

Interlude: Loss Functions for Classification

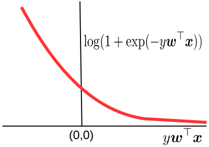
- In regression (assuming linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$), some common loss functions are

$$\ell(y, \hat{y}) = (y - \boldsymbol{w}^\top \boldsymbol{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \boldsymbol{w}^\top \boldsymbol{x}|$$

- We typically look at the difference between true y and model's prediction $\boldsymbol{w}^\top \boldsymbol{x}$
- How to formally define loss functions for classification?
- We have already looked at the loss function for logistic regression (assuming $y \in \{-1, +1\}$)

$$\ell(y, \hat{y}) = \log(1 + \exp(-y\boldsymbol{w}^\top \boldsymbol{x}))$$

- Why does the above make sense? Well, it is large for large misclassifications, small otherwise



if y=+1, then large positive $\boldsymbol{w}^\top \boldsymbol{x}$ implies small error

if y=-1, then large negative $\boldsymbol{w}^\top \boldsymbol{x}$ implies small error

if y=+1, then large negative $\boldsymbol{w}^\top \boldsymbol{x}$ implies large error

if y=-1, then large positive $\boldsymbol{w}^\top \boldsymbol{x}$ implies large error

Large positive $y\boldsymbol{w}^\top \boldsymbol{x} \Rightarrow$ small error, Large negative $y\boldsymbol{w}^\top \boldsymbol{x} \Rightarrow$ large error

- Are there other loss functions for classification?

Interlude: Loss Functions for Classification

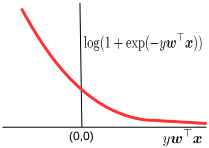
- In regression (assuming linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$), some common loss functions are

$$\ell(y, \hat{y}) \;=\; (y - \boldsymbol{w}^\top \boldsymbol{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \boldsymbol{w}^\top \boldsymbol{x}|$$

- We typically look at the difference between true y and model's prediction $\boldsymbol{w}^\top \boldsymbol{x}$
- How to formally define loss functions for classification?
- We have already looked at the loss function for logistic regression (assuming $y \in \{-1, +1\}$)

$$\ell(y, \hat{y}) = \log(1 + \exp(-y\boldsymbol{w}^\top \boldsymbol{x}))$$

- Why does the above make sense? Well, it is large for large misclassifications, small otherwise

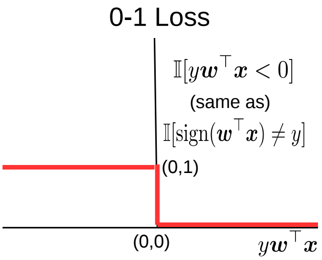


if y=+1, then large positive $\boldsymbol{w}^\top \boldsymbol{x}$ implies small error
if y=-1, then large negative $\boldsymbol{w}^\top \boldsymbol{x}$ implies small error
if y=+1, then large negative $\boldsymbol{w}^\top \boldsymbol{x}$ implies large error
if y=-1, then large positive $\boldsymbol{w}^\top \boldsymbol{x}$ implies large error

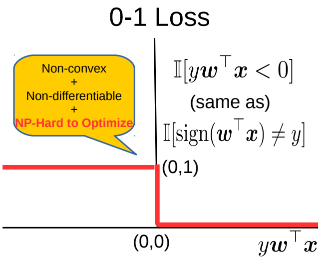
Large positive $y\boldsymbol{w}^\top \boldsymbol{x} \Rightarrow$ small error, Large negative $y\boldsymbol{w}^\top \boldsymbol{x} \Rightarrow$ large error

- Are there other loss functions for classification? Yes, several.

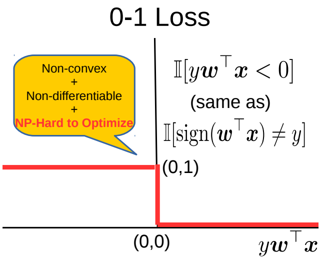
Interlude: Some Loss Functions for (Binary) Classification

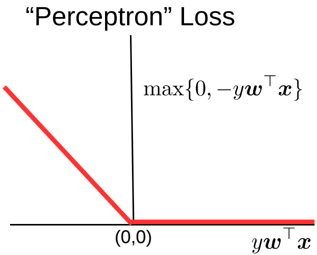


Interlude: Some Loss Functions for (Binary) Classification

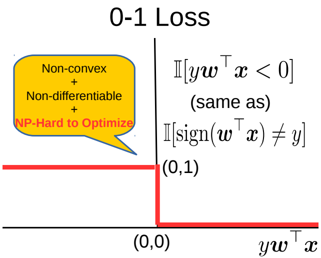


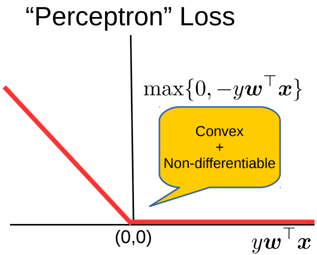
Interlude: Some Loss Functions for (Binary) Classification





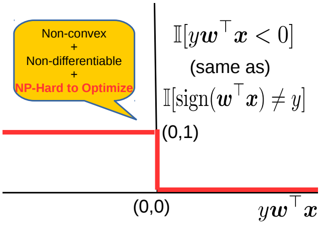
Interlude: Some Loss Functions for (Binary) Classification



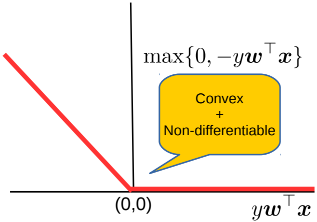


Interlude: Some Loss Functions for (Binary) Classification

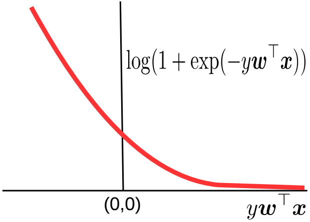
0-1 Loss



"Perceptron" Loss

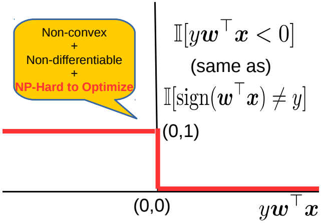


Log(istic) Loss

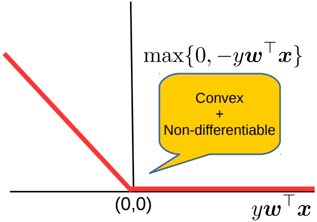


Interlude: Some Loss Functions for (Binary) Classification

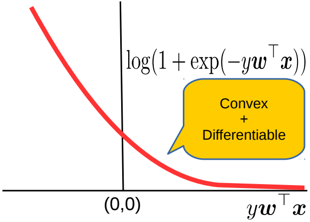
0-1 Loss



"Perceptron" Loss

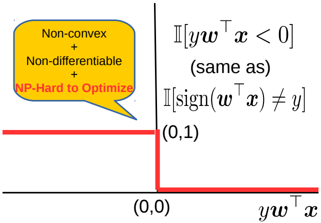


Log(istic) Loss

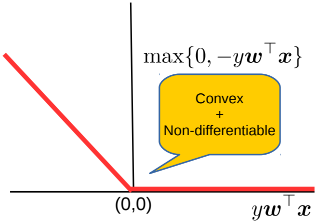


Interlude: Some Loss Functions for (Binary) Classification

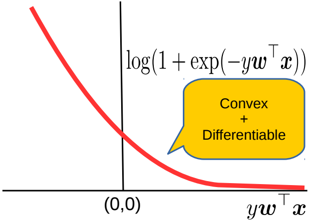
0-1 Loss



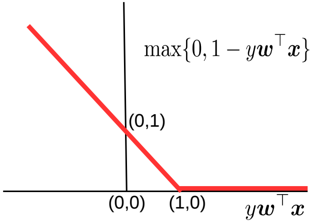
"Perceptron" Loss



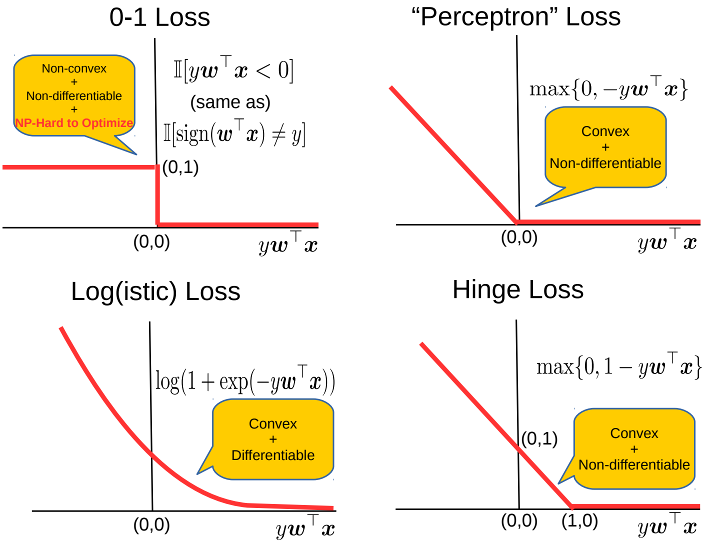
Log(istic) Loss



Hinge Loss

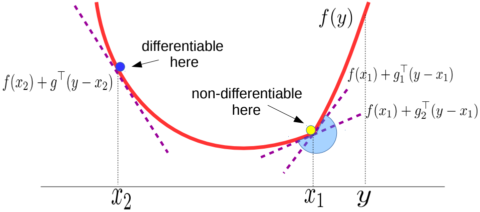


Interlude: Some Loss Functions for (Binary) Classification



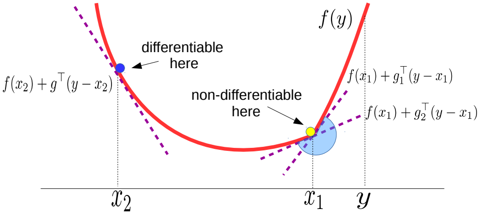
Optimizing Non-differentiable Functions

- Even though gradients are not defined for non-diff. functions, we can work with subgradients



Optimizing Non-differentiable Functions

- Even though gradients are not defined for non-diff. functions, we can work with subgradients

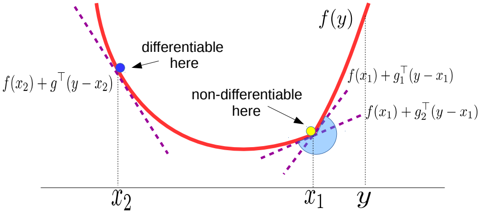


- For a function $f(\boldsymbol{x})$, its subgradient at $\boldsymbol{x}$ is <u>any</u> vector $\boldsymbol{g}$ s.t. $\forall \boldsymbol{y}$

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^\top(\boldsymbol{y} - \boldsymbol{x})$$

Optimizing Non-differentiable Functions

- Even though gradients are not defined for non-diff. functions, we can work with subgradients



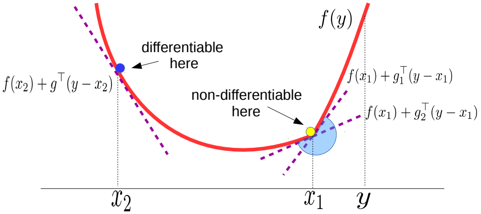
- For a function $f(\boldsymbol{x})$, its subgradient at $\boldsymbol{x}$ is any vector $\boldsymbol{g}$ s.t. $\forall \boldsymbol{y}$

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^\top(\boldsymbol{y} - \boldsymbol{x})$$

- A non-differentiable function can have several subgradients at the point of non-differentiability

Optimizing Non-differentiable Functions

- Even though gradients are not defined for non-diff. functions, we can work with subgradients



- For a function $f(\boldsymbol{x})$, its subgradient at $\boldsymbol{x}$ is <u>any</u> vector $\boldsymbol{g}$ s.t. $\forall \boldsymbol{y}$

$$f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^\top (\boldsymbol{y} - \boldsymbol{x})$$

- A non-differentiable function can have several subgradients at the point of non-differentiability
- Set of all subgradients of a function f at point $\boldsymbol{x}$ is called the subdifferential denoted as $\partial f(\boldsymbol{x})$

$$\partial f(\boldsymbol{x}) = \{\boldsymbol{g} : f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^\top (\boldsymbol{y} - \boldsymbol{x}), \quad \forall \boldsymbol{y}\}$$

Subgradient Descent: An Example

- Consider linear regression but with ℓ_1 norm on $\boldsymbol{w}$ (recall: ℓ_1 norm promotes a sparse $\boldsymbol{w}$)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda ||\boldsymbol{w}||_1$$

Subgradient Descent: An Example

- Consider linear regression but with ℓ_1 norm on $\boldsymbol{w}$ (recall: ℓ_1 norm promotes a sparse $\boldsymbol{w}$)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda ||\boldsymbol{w}||_1$$

- The squared error term is differentiable but the norm $||\boldsymbol{w}||_1$ is NOT at $w_d = 0$

Subgradient Descent: An Example

- Consider linear regression but with ℓ_1 norm on $\boldsymbol{w}$ (recall: ℓ_1 norm promotes a sparse $\boldsymbol{w}$)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda ||\boldsymbol{w}||_1$$

- The squared error term is differentiable but the norm $||\boldsymbol{w}||_1$ is NOT at $w_d = 0$

- We can use subgradients of $||\boldsymbol{w}||_1$ in this case

$$\boldsymbol{g} = 2\sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)\boldsymbol{x}_n + \lambda \boldsymbol{t}$$

Subgradient Descent: An Example

- Consider linear regression but with ℓ_1 norm on $\boldsymbol{w}$ (recall: ℓ_1 norm promotes a sparse $\boldsymbol{w}$)

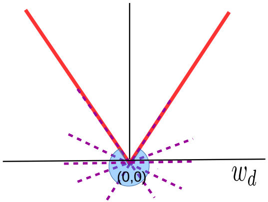
$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda ||\boldsymbol{w}||_1$$

- The squared error term is differentiable but the norm $||\boldsymbol{w}||_1$ is NOT at $w_d = 0$
- We can use subgradients of $||\boldsymbol{w}||_1$ in this case

$$\boldsymbol{g} = 2\sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)\boldsymbol{x}_n + \lambda \boldsymbol{t}$$

- Here $\boldsymbol{t}$ is a vector s.t.

$$t_d = \begin{cases} -1, & \text{for } w_d < 0 \\ [-1, +1] & \text{for } w_d = 0 \\ +1 & \text{for } w_d > 0 \end{cases}$$



Subgradient Descent: An Example

- Consider linear regression but with ℓ_1 norm on $\boldsymbol{w}$ (recall: ℓ_1 norm promotes a sparse $\boldsymbol{w}$)

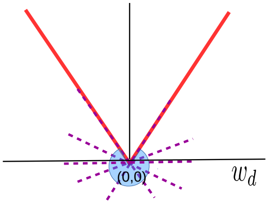
$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda ||\boldsymbol{w}||_1$$

- The squared error term is differentiable but the norm $||\boldsymbol{w}||_1$ is NOT at $w_d = 0$
- We can use subgradients of $||\boldsymbol{w}||_1$ in this case

$$\boldsymbol{g} = 2\sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)\boldsymbol{x}_n + \lambda \boldsymbol{t}$$

- Here $\boldsymbol{t}$ is a vector s.t.

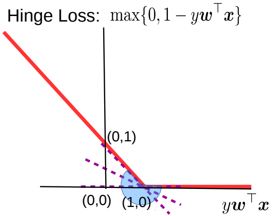
$$t_d = \begin{cases} -1, & \text{for } w_d < 0 \\ [-1, +1] & \text{for } w_d = 0 \\ +1 & \text{for } w_d > 0 \end{cases}$$



- If we take $t_d = 0$ at $w_d = 0$ then $t_d = \text{sign}(w_d)$

Subgradient Descent: Another Example

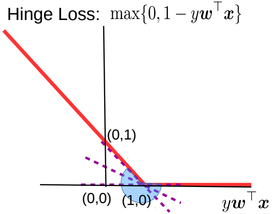
- Consider binary classification with hinge loss (used in SVM - will see later), assume ℓ_2 regularizer



- In this case loss (hinge) non-differentiable, regularizer differentiable

Subgradient Descent: Another Example

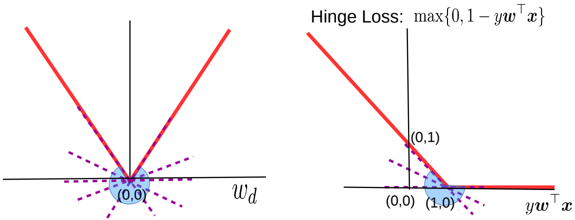
- Consider binary classification with hinge loss (used in SVM - will see later), assume ℓ_2 regularizer



- In this case loss (hinge) non-differentiable, regularizer differentiable
- Subgradient $\boldsymbol{t}$ of the hinge loss term will be

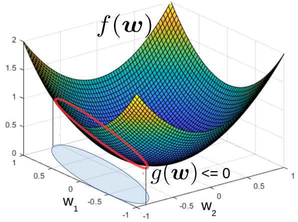
$$\boldsymbol{t} = \begin{cases} 0, & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n > 1 \\ -y_n \boldsymbol{x}_n & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n < 1 \\ k y_n \boldsymbol{x}_n & \text{for } y_n \boldsymbol{w}^\top \boldsymbol{x}_n = 1 \quad (\text{where } k \in [-1, 0]) \end{cases}$$

Subgradient Descent: Summary



- Not really that different from standard GD
- Only difference is that we use subgradients where function is non-differentiable
- In practice, it is like pretending that the function is differentiable everywhere

Constrained Optimization



1: Lagrangian based optimization

2: Projected gradient descent

Constrained Optimization: Lagrangian Approach

- Consider optimizing some function $f(\mathbf{w})$ subject to an inequality constraint on $\mathbf{w}$

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} f(\mathbf{w}), \quad \text{s.t.} \quad g(\mathbf{w}) \leq 0$$

- If constraint of the form $g(\mathbf{w}) \geq 0$, use $-g(\mathbf{w}) \leq 0$

Constrained Optimization: Lagrangian Approach

- Consider optimizing some function $f(\boldsymbol{w})$ subject to an inequality constraint on $\boldsymbol{w}$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}), \quad \text{s.t.} \quad g(\boldsymbol{w}) \leq 0$$

- If constraint of the form $g(\boldsymbol{w}) \geq 0$, use $-g(\boldsymbol{w}) \leq 0$
- Note: Can handle multiple inequality and equality constraints too (will see later)

Constrained Optimization: Lagrangian Approach

- Consider optimizing some function $f(\boldsymbol{w})$ subject to an inequality constraint on $\boldsymbol{w}$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}), \quad \text{s.t.} \quad g(\boldsymbol{w}) \leq 0$$

- If constraint of the form $g(\boldsymbol{w}) \geq 0$, use $-g(\boldsymbol{w}) \leq 0$
- Note: Can handle multiple inequality and equality constraints too (will see later)
- Can transform the above constrained problem into an equivalent unconstrained problem

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) + c(\boldsymbol{w})$$

Constrained Optimization: Lagrangian Approach

- Consider optimizing some function $f(\boldsymbol{w})$ subject to an inequality constraint on $\boldsymbol{w}$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}), \quad \text{s.t.} \quad g(\boldsymbol{w}) \leq 0$$

- If constraint of the form $g(\boldsymbol{w}) \geq 0$, use $-g(\boldsymbol{w}) \leq 0$
- Note: Can handle multiple inequality and equality constraints too (will see later)
- Can transform the above constrained problem into an equivalent unconstrained problem

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) + c(\boldsymbol{w})$$

where we have defined $c(\boldsymbol{w})$ as

$$c(\boldsymbol{w}) = \max_{\alpha \geq 0} \alpha g(\boldsymbol{w})$$

Constrained Optimization: Lagrangian Approach

- Consider optimizing some function $f(\boldsymbol{w})$ subject to an inequality constraint on $\boldsymbol{w}$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}), \quad \text{s.t.} \quad g(\boldsymbol{w}) \leq 0$$

- If constraint of the form $g(\boldsymbol{w}) \geq 0$, use $-g(\boldsymbol{w}) \leq 0$
- Note: Can handle multiple inequality and equality constraints too (will see later)
- Can transform the above constrained problem into an equivalent unconstrained problem

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) + c(\boldsymbol{w})$$

where we have defined $c(\boldsymbol{w})$ as

$$c(\boldsymbol{w}) = \max_{\alpha \geq 0} \alpha g(\boldsymbol{w}) = \begin{cases} \infty, & \text{if } g(\boldsymbol{w}) > 0 \quad \text{(constraint violated)} \\ 0 & \text{if } g(\boldsymbol{w}) \leq 0 \quad \text{(constraint satisfied)} \end{cases}$$

Constrained Optimization: Lagrangian Approach

- Consider optimizing some function $f(\boldsymbol{w})$ subject to an inequality constraint on $\boldsymbol{w}$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}), \quad \text{s.t.} \quad g(\boldsymbol{w}) \leq 0$$

- If constraint of the form $g(\boldsymbol{w}) \geq 0$, use $-g(\boldsymbol{w}) \leq 0$
- Note: Can handle multiple inequality and equality constraints too (will see later)
- Can transform the above constrained problem into an equivalent unconstrained problem

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) + c(\boldsymbol{w})$$

where we have defined $c(\boldsymbol{w})$ as

$$c(\boldsymbol{w}) = \max_{\alpha \geq 0} \alpha g(\boldsymbol{w}) = \begin{cases} \infty, & \text{if } g(\boldsymbol{w}) > 0 \quad \text{(constraint violated)} \\ 0 & \text{if } g(\boldsymbol{w}) \leq 0 \quad \text{(constraint satisfied)} \end{cases}$$

- We can equivalently write the problem as

$$\boxed{\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \alpha g(\boldsymbol{w}) \right\}}$$

Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \alpha g(\boldsymbol{w}) \right\}$$

Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \; \alpha g(\boldsymbol{w}) \right\} = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\}$$

Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \ \alpha g(\boldsymbol{w}) \right\} = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\}$$

- The function $\mathcal{L}(\boldsymbol{w}, \alpha) = f(\boldsymbol{w}) + \alpha g(\boldsymbol{w})$ called the Lagrangian, optimized w.r.t. $\boldsymbol{w}$ and α

Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \alpha g(\boldsymbol{w}) \right\} = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\}$$

- The function $\mathcal{L}(\boldsymbol{w}, \alpha) = f(\boldsymbol{w}) + \alpha g(\boldsymbol{w})$ called the Lagrangian, optimized w.r.t. $\boldsymbol{w}$ and α
- α is known as the Lagrange multiplier

Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \ \alpha g(\boldsymbol{w}) \right\} = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\}$$

- The function $\mathcal{L}(\boldsymbol{w}, \alpha) = f(\boldsymbol{w}) + \alpha g(\boldsymbol{w})$ called the Lagrangian, optimized w.r.t. $\boldsymbol{w}$ and α
- α is known as the Lagrange multiplier
- Primal and Dual problems

$$\hat{\boldsymbol{w}}_P = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\} \qquad \text{(primal problem)}$$

Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \left\{ f(\mathbf{w}) + \arg\max_{\alpha \geq 0} \ \alpha g(\mathbf{w}) \right\} = \arg\min_{\mathbf{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\}$$

- The function $\mathcal{L}(\mathbf{w}, \alpha) = f(\mathbf{w}) + \alpha g(\mathbf{w})$ called the Lagrangian, optimized w.r.t. $\mathbf{w}$ and α
- α is known as the Lagrange multiplier
- Primal and Dual problems

$$\hat{\mathbf{w}}_P = \arg\min_{\mathbf{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\} \qquad \text{(primal problem)}$$

$$\hat{\mathbf{w}}_D = \arg\max_{\alpha \geq 0} \left\{ \arg\min_{\mathbf{w}} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\} \qquad \text{(dual problem)}$$

Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \left\{ f(\boldsymbol{w}) + \arg\max_{\alpha \geq 0} \ \alpha g(\boldsymbol{w}) \right\} = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\}$$

- The function $\mathcal{L}(\boldsymbol{w}, \alpha) = f(\boldsymbol{w}) + \alpha g(\boldsymbol{w})$ called the Lagrangian, optimized w.r.t. $\boldsymbol{w}$ and α
- α is known as the Lagrange multiplier
- Primal and Dual problems

$$\hat{\boldsymbol{w}}_P = \arg\min_{\boldsymbol{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\} \qquad \text{(primal problem)}$$

$$\hat{\boldsymbol{w}}_D = \arg\max_{\alpha \geq 0} \left\{ \arg\min_{\boldsymbol{w}} \{ f(\boldsymbol{w}) + \alpha g(\boldsymbol{w}) \} \right\} \qquad \text{(dual problem)}$$

- Note: $\hat{\boldsymbol{w}}_P = \hat{\boldsymbol{w}}_D$ in some nice cases (e.g., when $f(\boldsymbol{w})$ and constraint set $g(\boldsymbol{w}) \leq 0$ are convex)

Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \left\{ f(\mathbf{w}) + \arg\max_{\alpha \geq 0} \alpha g(\mathbf{w}) \right\} = \arg\min_{\mathbf{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\}$$

- The function $\mathcal{L}(\mathbf{w}, \alpha) = f(\mathbf{w}) + \alpha g(\mathbf{w})$ called the Lagrangian, optimized w.r.t. $\mathbf{w}$ and α
- α is known as the Lagrange multiplier
- Primal and Dual problems

$$\hat{\mathbf{w}}_P = \arg\min_{\mathbf{w}} \left\{ \arg\max_{\alpha \geq 0} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\} \qquad \text{(primal problem)}$$

$$\hat{\mathbf{w}}_D = \arg\max_{\alpha \geq 0} \left\{ \arg\min_{\mathbf{w}} \{ f(\mathbf{w}) + \alpha g(\mathbf{w}) \} \right\} \qquad \text{(dual problem)}$$

- Note: $\hat{\mathbf{w}}_P = \hat{\mathbf{w}}_D$ in some nice cases (e.g., when $f(\mathbf{w})$ and constraint set $g(\mathbf{w}) \leq 0$ are convex)
- For dual solution, $\alpha_D g(\hat{\mathbf{w}}_D) = 0$ (complimentary slackness/Karush-Kuhn-Tucker (KKT) condition)

Constrained Optimization: Lagrangian with Multiple Constraints

- We can also have multiple inequality and equality constraints

$$
\begin{aligned}
\hat{\boldsymbol{w}} &= \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) \\
\text{s.t.} \quad & g_i(\boldsymbol{w}) \leq 0, \quad i = 1, \ldots, K \\
& h_j(\boldsymbol{w}) = 0, \quad j =, 1, \ldots, L
\end{aligned}
$$

Constrained Optimization: Lagrangian with Multiple Constraints

- We can also have multiple inequality and equality constraints

$$
\begin{aligned}
\hat{\boldsymbol{w}} &= \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) \\
\text{s.t.} \quad & g_i(\boldsymbol{w}) \leq 0, \quad i = 1, \ldots, K \\
& h_j(\boldsymbol{w}) = 0, \quad j =, 1, \ldots, L
\end{aligned}
$$

- Introduce Lagrange multipliers $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K) \geq 0$ and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_L)$

Constrained Optimization: Lagrangian with Multiple Constraints

- We can also have multiple inequality and equality constraints

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} f(\boldsymbol{w})$$

$$\text{s.t.} \quad g_i(\boldsymbol{w}) \leq 0, \quad i = 1, \ldots, K$$

$$h_j(\boldsymbol{w}) = 0, \quad j =, 1, \ldots, L$$

- Introduce Lagrange multipliers $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K) \geq 0$ and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_L)$
- The Lagrangian based primal and dual problems will be

$$\hat{\boldsymbol{w}}_P = \arg\min_{\boldsymbol{w}}\{\arg\max_{\boldsymbol{\alpha}\geq 0, \boldsymbol{\beta}}\{f(\boldsymbol{w}) + \sum_{i=1}^{K} \alpha_i g_i(\boldsymbol{w}) + \sum_{j=1}^{L} \beta_j h_j(\boldsymbol{w})\}\}$$

Constrained Optimization: Lagrangian with Multiple Constraints

- We can also have multiple inequality and equality constraints

$$
\begin{aligned}
\hat{\boldsymbol{w}} &= \arg\min_{\boldsymbol{w}} f(\boldsymbol{w}) \\
\text{s.t.} \quad & g_i(\boldsymbol{w}) \leq 0, \quad i = 1, \ldots, K \\
& h_j(\boldsymbol{w}) = 0, \quad j =, 1, \ldots, L
\end{aligned}
$$

- Introduce Lagrange multipliers $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_K) \geq 0$ and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_L)$
- The Lagrangian based primal and dual problems will be

$$
\begin{aligned}
\hat{\boldsymbol{w}}_P &= \arg\min_{\boldsymbol{w}} \{\arg\max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \{f(\boldsymbol{w}) + \sum_{i=1}^{K} \alpha_i g_i(\boldsymbol{w}) + \sum_{j=1}^{L} \beta_j h_j(\boldsymbol{w})\}\} \\
\hat{\boldsymbol{w}}_D &= \arg\max_{\boldsymbol{\alpha} \geq 0, \boldsymbol{\beta}} \{\arg\min_{\boldsymbol{w}} \{f(\boldsymbol{w}) + \sum_{i=1}^{K} \alpha_i g_i(\boldsymbol{w}) + \sum_{j=1}^{L} \beta_j h_j(\boldsymbol{w})\}\}
\end{aligned}
$$

Lagrangian based Optimization: An Example

- Consider the generative classification model with K classes
- Suppose we want to estimate the parameters of class-marginal $p(y)$

$$p(y|\boldsymbol{\pi}) = \text{multinoulli}(\pi_1, \pi_2, \ldots, \pi_K) = \prod_{k=1}^{K} \pi_k^{\mathbb{I}[y=k]}, \quad \text{s.t.} \ \sum_{k=1}^{K} \pi_k = 1$$

Lagrangian based Optimization: An Example

- Consider the generative classification model with K classes
- Suppose we want to estimate the parameters of class-marginal $p(y)$

$$p(y|\boldsymbol{\pi}) = \text{multinoulli}(\pi_1, \pi_2, \ldots, \pi_K) = \prod_{k=1}^{K} \pi_k^{\mathbb{I}[y=k]}, \quad \text{s.t.} \sum_{k=1}^{K} \pi_k = 1$$

- Given N observations $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, the negative log-likelihood for class marginal

$$f(\boldsymbol{\pi}) = -\sum_{n=1}^{N} \log p(y_n|\boldsymbol{\pi})$$

Lagrangian based Optimization: An Example

- Consider the generative classification model with K classes
- Suppose we want to estimate the parameters of class-marginal $p(y)$

$$p(y|\boldsymbol{\pi}) = \text{multinoulli}(\pi_1, \pi_2, \ldots, \pi_K) = \prod_{k=1}^{K} \pi_k^{\mathbb{I}[y=k]}, \quad \text{s.t.} \sum_{k=1}^{K} \pi_k = 1$$

- Given N observations $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, the negative log-likelihood for class marginal

$$f(\boldsymbol{\pi}) = -\sum_{n=1}^{N} \log p(y_n|\boldsymbol{\pi})$$

- We have an equality constraint $\sum_{k=1}^{K} \pi_k - 1 = 0$

Lagrangian based Optimization: An Example

- Consider the generative classification model with K classes
- Suppose we want to estimate the parameters of class-marginal $p(y)$

$$p(y|\boldsymbol{\pi}) = \text{multinoulli}(\pi_1, \pi_2, \ldots, \pi_K) = \prod_{k=1}^{K} \pi_k^{\mathbb{I}[y=k]}, \quad \text{s.t.} \sum_{k=1}^{K} \pi_k = 1$$

- Given N observations $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, the negative log-likelihood for class marginal

$$f(\boldsymbol{\pi}) = -\sum_{n=1}^{N} \log p(y_n|\boldsymbol{\pi})$$

- We have an equality constraint $\sum_{k=1}^{K} \pi_k - 1 = 0$
- The Lagrangian for this problem will be

$$\mathcal{L}(\boldsymbol{\pi}, \beta) = f(\boldsymbol{\pi}) + \beta(\sum_{k=1}^{K} \pi_k - 1)$$

Lagrangian based Optimization: An Example

- Consider the generative classification model with K classes
- Suppose we want to estimate the parameters of class-marginal $p(y)$

$$p(y|\boldsymbol{\pi}) = \text{multinoulli}(\pi_1, \pi_2, \ldots, \pi_K) = \prod_{k=1}^{K} \pi_k^{\mathbb{I}[y=k]}, \quad \text{s.t.} \sum_{k=1}^{K} \pi_k = 1$$

- Given N observations $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, the negative log-likelihood for class marginal

$$f(\boldsymbol{\pi}) = -\sum_{n=1}^{N} \log p(y_n|\boldsymbol{\pi})$$

- We have an equality constraint $\sum_{k=1}^{K} \pi_k - 1 = 0$
- The Lagrangian for this problem will be

$$\mathcal{L}(\boldsymbol{\pi}, \beta) = f(\boldsymbol{\pi}) + \beta(\sum_{k=1}^{K} \pi_k - 1)$$

- Exercise: Solve $\arg\max_{\beta} \arg\min_{\boldsymbol{\pi}} \mathcal{L}(\boldsymbol{\pi}, \beta)$ and show that $\pi_k = N_k/N$

Projected Gradient Descent

- Suppose our problem requires the parameters to lie within a set $\mathcal{C}$

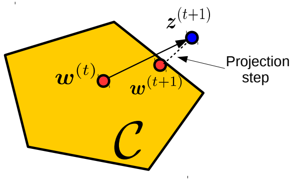
$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}), \quad \text{subject to} \quad \boldsymbol{w} \in \mathcal{C}$$

Projected Gradient Descent

- Suppose our problem requires the parameters to lie within a set $\mathcal{C}$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}), \quad \text{subject to} \quad \boldsymbol{w} \in \mathcal{C}$$

- Projected GD is very similar to GD with an extra projection step

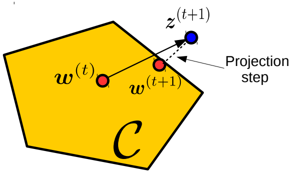


Projected Gradient Descent

- Suppose our problem requires the parameters to lie within a set $\mathcal{C}$

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}), \quad \text{subject to} \quad \boldsymbol{w} \in \mathcal{C}$$

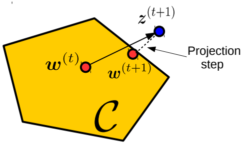
- Projected GD is very similar to GD with an extra projection step



- Each step of projected GD works as follows
  - Do the usual GD update: $\boldsymbol{z}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$
  - Check $\boldsymbol{z}^{(t+1)}$ for the constraints
    - If $\boldsymbol{z}^{(t+1)} \in \mathcal{C}$, $\boldsymbol{w}^{(t+1)} = \boldsymbol{z}^{(t+1)}$
    - If $\boldsymbol{z}^{(t+1)} \notin \mathcal{C}$, project on the constraint set: $\boldsymbol{w}^{(t+1)} = \underbrace{\Pi_{\mathcal{C}}[\boldsymbol{z}^{(t+1)}]}_{\text{projection}}$

Projected GD: How to Project?

- The projection itself is an optimization problem

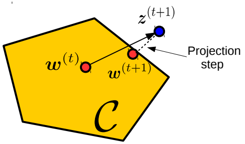


- Given $\boldsymbol{z}$, we find the "closest" point (e.g., in Euclidean sense) $\boldsymbol{w}$ in the set as follows

$$\Pi_{\mathcal{C}}[\boldsymbol{z}] = \arg\min_{\boldsymbol{w}\in\mathcal{C}} ||\boldsymbol{w} - \boldsymbol{z}||^2$$

Projected GD: How to Project?

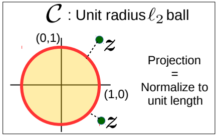
- The projection itself is an optimization problem

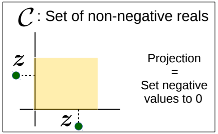


- Given z, we find the "closest" point (e.g., in Euclidean sense) w in the set as follows

$$\Pi_{\mathcal{C}}[z] = \arg\min_{w \in \mathcal{C}} ||w - z||^2$$

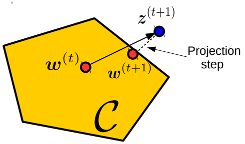
- For some sets $\mathcal{C}$, the projection step is easy/trivial





Projected GD: How to Project?

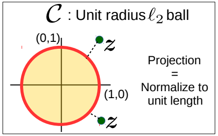
- The projection itself is an optimization problem

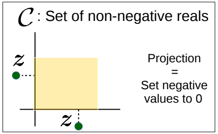


- Given $\boldsymbol{z}$, we find the "closest" point (e.g., in Euclidean sense) $\boldsymbol{w}$ in the set as follows

$$\Pi_{\mathcal{C}}[\boldsymbol{z}] = \arg\min_{\boldsymbol{w} \in \mathcal{C}} ||\boldsymbol{w} - \boldsymbol{z}||^2$$

- For some sets $\mathcal{C}$, the projection step is easy/trivial





- For some other sets $\mathcal{C}$, the projection step may be a bit more involved

Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

- CD: Each step update one component (co-ordinate) at a time, keeping all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$$

Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

- CD: Each step update one component (co-ordinate) at a time, keeping all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$$

- Cost of each update is now independent of D

Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

- CD: Each step update one component (co-ordinate) at a time, keeping all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$$

- Cost of each update is now independent of D
- How to pick which co-ordinate to update?

Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

- CD: Each step update one component (co-ordinate) at a time, keeping all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$$

- Cost of each update is now independent of D
- How to pick which co-ordinate to update?
  - Can be chosen in random order (stochastic CD)

Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

- CD: Each step update one component (co-ordinate) at a time, keeping all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$$

- Cost of each update is now independent of D
- How to pick which co-ordinate to update?
  - Can be chosen in random order (stochastic CD)
  - Can be chosen in cyclic order

Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

- CD: Each step update one component (co-ordinate) at a time, keeping all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$$

- Cost of each update is now independent of D
- How to pick which co-ordinate to update?
  - Can be chosen in random order (stochastic CD)
  - Can be chosen in cyclic order
- Note: Can also update "blocks" of co-ordinates (called Block co-ordinate descent)

Co-ordinate Descent (CD)

- Standard GD update for $\boldsymbol{w} \in \mathbb{R}^D$ at each step

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

- CD: Each step update one component (co-ordinate) at a time, keeping all others fixed

$$w_d^{(t+1)} = w_d^{(t)} - \eta_t g_d^{(t)}$$

- Cost of each update is now independent of D
- How to pick which co-ordinate to update?
  - Can be chosen in random order (stochastic CD)
  - Can be chosen in cyclic order
- Note: Can also update "blocks" of co-ordinates (called Block co-ordinate descent)
- Should cache previous computations (e.g., $\boldsymbol{w}^\top \boldsymbol{x}$) to avoid $\mathcal{O}(D)$ cost in gradient computation

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)
- We can therefore follow an alternating scheme to optimize w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)
- We can therefore follow an alternating scheme to optimize w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$
  - Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)
- We can therefore follow an alternating scheme to optimize w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$
  - Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$
  - Solve $\boldsymbol{w}_1^{(t+1)} = \arg\max_{\boldsymbol{w}_1} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2^{(t)})$

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)
- We can therefore follow an alternating scheme to optimize w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$
  - Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$
  - Solve $\boldsymbol{w}_1^{(t+1)} = \arg\max_{\boldsymbol{w}_1} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2^{(t)})$
  - Solve $\boldsymbol{w}_2^{(t+1)} = \arg\max_{\boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1^{(t+1)}, \boldsymbol{w}_2)$

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)
- We can therefore follow an alternating scheme to optimize w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$
    - Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$
    - Solve $\boldsymbol{w}_1^{(t+1)} = \arg\max_{\boldsymbol{w}_1} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2^{(t)})$
    - Solve $\boldsymbol{w}_2^{(t+1)} = \arg\max_{\boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1^{(t+1)}, \boldsymbol{w}_2)$
    - $t = t + 1$. Repeat until convergence

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)
- We can therefore follow an alternating scheme to optimize w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$
  - Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$
  - Solve $\boldsymbol{w}_1^{(t+1)} = \arg\max_{\boldsymbol{w}_1} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2^{(t)})$
  - Solve $\boldsymbol{w}_2^{(t+1)} = \arg\max_{\boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1^{(t+1)}, \boldsymbol{w}_2)$
  - $t = t + 1$. Repeat until convergence
- Usually converges to a local optima of $\mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$. Also connections to EM (will see later)

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)
- We can therefore follow an alternating scheme to optimize w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$
  - Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$
  - Solve $\boldsymbol{w}_1^{(t+1)} = \arg\max_{\boldsymbol{w}_1} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2^{(t)})$
  - Solve $\boldsymbol{w}_2^{(t+1)} = \arg\max_{\boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1^{(t+1)}, \boldsymbol{w}_2)$
  - $t = t + 1$. Repeat until convergence
- Usually converges to a local optima of $\mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$. Also connections to EM (will see later)
- Extends to more than 2 variables as well (and not just to vectors)

Alternating Optimization

- Many optimization problems consist of several variables. Very common in ML.
- For simplicity, suppose we want to optimize a function of 2 variables $\boldsymbol{w}_1 \in \mathbb{R}^D$ and $\boldsymbol{w}_2 \in \mathbb{R}^D$

$$\{\hat{\boldsymbol{w}}_1, \hat{\boldsymbol{w}}_2\} = \arg\min_{\boldsymbol{w}_1, \boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$$

- Jointly optimizing w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ may be hard (e.g., if their values depend on each other)
- Often, knowing value of one may make optimization w.r.t. other easy (sometimes even closed form)
- We can therefore follow an alternating scheme to optimize w.r.t. $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$
  - Initialize one of the variables, e.g., $\boldsymbol{w}_2 = \boldsymbol{w}_2^{(0)}, t = 0$
  - Solve $\boldsymbol{w}_1^{(t+1)} = \arg\max_{\boldsymbol{w}_1} \mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2^{(t)})$
  - Solve $\boldsymbol{w}_2^{(t+1)} = \arg\max_{\boldsymbol{w}_2} \mathcal{L}(\boldsymbol{w}_1^{(t+1)}, \boldsymbol{w}_2)$
  - $t = t + 1$. Repeat until convergence
- Usually converges to a local optima of $\mathcal{L}(\boldsymbol{w}_1, \boldsymbol{w}_2)$. Also connections to EM (will see later)
- Extends to more than 2 variables as well (and not just to vectors). CD is a special case.

Second-Order Methods: Newton's Method

- GD and variants only use first-order information (the gradient)

Second-Order Methods: Newton's Method

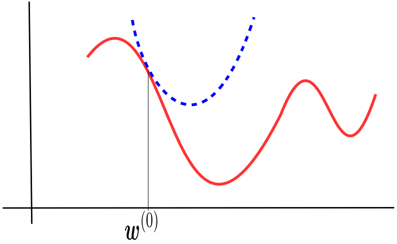
- GD and variants only use first-order information (the gradient)
- Second-order information often tells us a lot more about the function's shape, curvature, etc.
- Newton's method is one such method that uses second-order information

Second-Order Methods: Newton's Method

- GD and variants only use first-order information (the gradient)
- Second-order information often tells us a lot more about the function's shape, curvature, etc.
- Newton's method is one such method that uses second-order information
- At each point, approximate the function by its quadratic approx. and minimize it

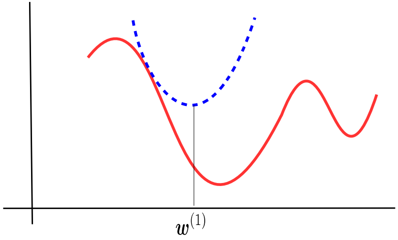
Second-Order Methods: Newton's Method

- GD and variants only use first-order information (the gradient)
- Second-order information often tells us a lot more about the function's shape, curvature, etc.
- Newton's method is one such method that uses second-order information
- At each point, approximate the function by its quadratic approx. and minimize it



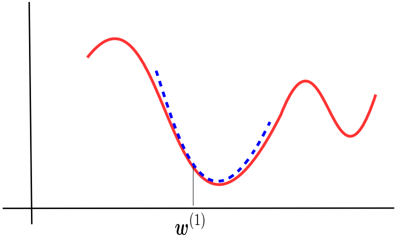
Second-Order Methods: Newton's Method

- GD and variants only use first-order information (the gradient)
- Second-order information often tells us a lot more about the function's shape, curvature, etc.
- Newton's method is one such method that uses second-order information
- At each point, approximate the function by its quadratic approx. and minimize it



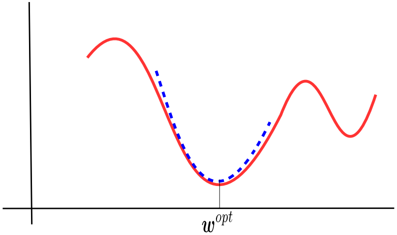
Second-Order Methods: Newton's Method

- GD and variants only use first-order information (the gradient)
- Second-order information often tells us a lot more about the function's shape, curvature, etc.
- Newton's method is one such method that uses second-order information
- At each point, approximate the function by its quadratic approx. and minimize it



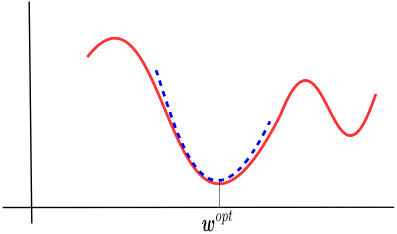
Second-Order Methods: Newton's Method

- GD and variants only use first-order information (the gradient)
- Second-order information often tells us a lot more about the function's shape, curvature, etc.
- Newton's method is one such method that uses second-order information
- At each point, approximate the function by its quadratic approx. and minimize it



Second-Order Methods: Newton's Method

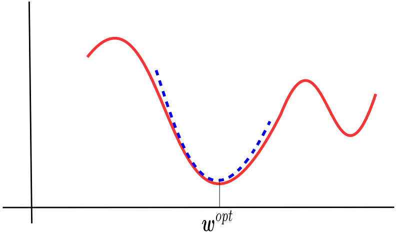
- GD and variants only use first-order information (the gradient)
- Second-order information often tells us a lot more about the function's shape, curvature, etc.
- Newton's method is one such method that uses second-order information
- At each point, approximate the function by its quadratic approx. and minimize it



- Doesn't rely on gradient to choose $\boldsymbol{w}^{(t+1)}$

Second-Order Methods: Newton's Method

- GD and variants only use first-order information (the gradient)
- Second-order information often tells us a lot more about the function's shape, curvature, etc.
- Newton's method is one such method that uses second-order information
- At each point, approximate the function by its quadratic approx. and minimize it



- Doesn't rely on gradient to choose $\boldsymbol{w}^{(t+1)}$
- Instead, each step directly jumps to the minima of quadratic approximation

Second-Order Methods: Newton's Method

- The quadratic (Taylor) approximation of $f(\boldsymbol{w})$ at $\boldsymbol{w}^{(t)}$ is given by

$$\tilde{f}(\boldsymbol{w}) = f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top(\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)})$$

Second-Order Methods: Newton's Method

- The quadratic (Taylor) approximation of $f(\boldsymbol{w})$ at $\boldsymbol{w}^{(t)}$ is given by

$$\tilde{f}(\boldsymbol{w}) = f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top(\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)})$$

- The minimizer of this quadratic approximation is (exercise: verify)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \tilde{f}(\boldsymbol{w}) = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1} \nabla f(\boldsymbol{w}^{(t)})$$

Second-Order Methods: Newton's Method

- The quadratic (Taylor) approximation of $f(\boldsymbol{w})$ at $\boldsymbol{w}^{(t)}$ is given by

$$\tilde{f}(\boldsymbol{w}) = f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top(\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)})$$

- The minimizer of this quadratic approximation is (exercise: verify)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \tilde{f}(\boldsymbol{w}) = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1}\nabla f(\boldsymbol{w}^{(t)})$$

- This is the update used in Newton's method (a second order method since it uses the Hessian)

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1}\nabla f(\boldsymbol{w}^{(t)})$$

- Look, Ma! No learning rate! :-)

Second-Order Methods: Newton's Method

- The quadratic (Taylor) approximation of $f(\boldsymbol{w})$ at $\boldsymbol{w}^{(t)}$ is given by

$$\tilde{f}(\boldsymbol{w}) = f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top(\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)})$$

- The minimizer of this quadratic approximation is (exercise: verify)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \tilde{f}(\boldsymbol{w}) = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1} \nabla f(\boldsymbol{w}^{(t)})$$

- This is the update used in Newton's method (a second order method since it uses the Hessian)

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1} \nabla f(\boldsymbol{w}^{(t)})$$

- Look, Ma! No learning rate! :-)
- Very fast if $f(\boldsymbol{w})$ is convex. But expensive due to Hessian computation/inversion.

Second-Order Methods: Newton's Method

- The quadratic (Taylor) approximation of $f(\boldsymbol{w})$ at $\boldsymbol{w}^{(t)}$ is given by

$$\tilde{f}(\boldsymbol{w}) = f(\boldsymbol{w}^{(t)}) + \nabla f(\boldsymbol{w}^{(t)})^\top(\boldsymbol{w} - \boldsymbol{w}^{(t)}) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^{(t)})^\top \nabla^2 f(\boldsymbol{w}^{(t)})(\boldsymbol{w} - \boldsymbol{w}^{(t)})$$

- The minimizer of this quadratic approximation is (exercise: verify)

$$\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \tilde{f}(\boldsymbol{w}) = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1}\nabla f(\boldsymbol{w}^{(t)})$$

- This is the update used in Newton's method (a second order method since it uses the Hessian)

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - (\nabla^2 f(\boldsymbol{w}^{(t)}))^{-1}\nabla f(\boldsymbol{w}^{(t)})$$

- Look, Ma! No learning rate! :-)
- Very fast if $f(\boldsymbol{w})$ is convex. But expensive due to Hessian computation/inversion.
- Many ways to approximate the Hessian (e.g., using previous gradients); also look at L-BFGS etc.

Summary

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods often use gradient methods
  - Backpropagation algorithm used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization require methods such as Lagrangian or projected gradient
- Second order methods such as Newton's method are much faster but computationally expensive

Summary

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods often use gradient methods
  - Backpropagation algorithm used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization require methods such as Lagrangian or projected gradient
- Second order methods such as Newton's method are much faster but computationally expensive
- But computing all this gradient related stuff looks scary to me. Any help?

Summary

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods often use gradient methods
  - Backpropagation algorithm used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization require methods such as Lagrangian or projected gradient
- Second order methods such as Newton's method are much faster but computationally expensive
- But computing all this gradient related stuff looks scary to me. Any help?
  - Don't worry. Automatic Differentiation (AD) methods available now
  - AD only requires specifying the loss function (especially useful for deep neural nets)

Summary

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods often use gradient methods
  - Backpropagation algorithm used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization require methods such as Lagrangian or projected gradient
- Second order methods such as Newton's method are much faster but computationally expensive
- But computing all this gradient related stuff looks scary to me. Any help?
  - Don't worry. Automatic Differentiation (AD) methods available now
  - AD only requires specifying the loss function (especially useful for deep neural nets)
  - Many packages such as Tensorflow, PyTorch, etc. provide AD support

Summary

- Gradient methods are simple to understand and implement
- More sophisticated optimization methods often use gradient methods
  - Backpropagation algorithm used in deep neural nets is GD + chain rule of differentiation
- Use subgradient methods if function not differentiable
- Constrained optimization require methods such as Lagrangian or projected gradient
- Second order methods such as Newton's method are much faster but computationally expensive
- But computing all this gradient related stuff looks scary to me. Any help?
  - Don't worry. Automatic Differentiation (AD) methods available now
  - AD only requires specifying the loss function (especially useful for deep neural nets)
  - Many packages such as Tensorflow, PyTorch, etc. provide AD support
  - But having a good understanding of optimization is still helpful