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.,

\[ \hat{w} = \arg \min_w \mathcal{L}(w) = \arg \min_w \frac{1}{N} \sum_{n=1}^{N} \ell_n(w) + R(w) \]
Recap: Convex and Non-Convex Function

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

\[ \hat{w} = \arg \min_w \mathcal{L}(w) = \arg \min_w \frac{1}{N} \sum_{n=1}^{N} \ell_n(w) + R(w) \]

- 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.,

\[
\hat{w} = \arg \min_w L(w) = \arg \min_w \frac{1}{N} \sum_{n=1}^{N} \ell_n(w) + R(w)
\]

- Convex functions have a unique minima

- Non-convex functions 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

$$Jensen's\ \text{Inequality}$$

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

Jensen's Inequality
Recap: Convex Functions

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

![Convex Function and Non-convex Function](image)

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

\[
\begin{align*}
\text{Convex Function} & \quad f(x') \\
\text{Non-convex Function} & \quad f(x)
\end{align*}
\]

Note: “Chord lies above function” more formally means:

If \( f \) is convex then given

\[
\sum_{i=1}^{n} \alpha_i = 1 \quad \text{s.t.} \quad f\left(\sum_{i=1}^{n} \alpha_i x_i\right) \leq \sum_{i=1}^{n} \alpha_i f(x_i)
\]

Jensen’s Inequality
Recap: Convex Functions

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

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.

  ![Convex Function vs. Non-convex Function](image)

  If $f$ is convex then given

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

  **Jensen’s Inequality**

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

  ![Convex Function](image)

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

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

- Note: If $f$ is convex then $-f$ is a **concave** function.

---

Intro to Machine Learning (CS771A)  Optimization Techniques for ML (2)
Recap: Gradient Descent

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

\[ w^{(t+1)} = w^{(t)} - \eta_t g^{(t)} \]
Recap: Gradient Descent

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

\[
\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t g^{(t)}
\]
Recap: Gradient Descent

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

$$w^{(t+1)} = w^{(t)} - \eta_t g^{(t)}$$
Recap: Gradient Descent

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

$$w^{(t+1)} = w^{(t)} - \eta_t g^{(t)}$$
Recap: Gradient Descent

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

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)}$$
Recap: Gradient Descent

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

$$w^{(t+1)} = w^{(t)} - \eta_t g^{(t)}$$
Recap: Gradient Descent

Gradient Descent

1. Initialize $w$ as $w^{(0)}$
2. Update $w$ as follows
   \[ w^{(t+1)} = w^{(t)} - \eta_t g^{(t)} \]
3. Repeat until convergence
Recap: Gradient Descent

- The learning rate $\eta_t$ 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

![Diagram showing the effects of very small and very large learning rates on gradient descent](image)

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: Gradient Descent

- The learning rate $\eta_t$ 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. $\frac{1}{t}$, $\frac{1}{\sqrt{t}}$, etc.)
- Use adaptive learning rates (e.g., using methods such as Adagrad, Adam)
Recap: Gradient Descent

- The learning rate $\eta_t$ 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

\[
g = \nabla_w \left[ \frac{1}{N} \sum_{n=1}^{N} \ell_n(w) \right] = \frac{1}{N} \sum_{n=1}^{N} g_n \quad \text{(ignoring regularizer } R(w))
\]
Recap: Stochastic Gradient Descent

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

$$g = \nabla_w \left[ \frac{1}{N} \sum_{n=1}^{N} \ell_n(w) \right] = \frac{1}{N} \sum_{n=1}^{N} g_n \quad \text{(ignoring regularizer } R(w))$$

- **Stochastic Gradient Descent (SGD)** approximates $g$ using a single data point
Recap: Stochastic Gradient Descent

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

$$
g = \nabla_w \left[ \frac{1}{N} \sum_{n=1}^{N} \ell_n(w) \right] = \frac{1}{N} \sum_{n=1}^{N} g_n \quad \text{(ignoring regularizer $R(w)$)}
$$

- **Stochastic Gradient Descent** (SGD) approximates $g$ using a single data point

- In iteration $t$, SGD picks a uniformly random $i \in \{1, \ldots, N\}$ and approximate $g$ as

$$
g \approx g_i = \nabla_w \ell_i(w)
$$
Recap: Stochastic Gradient Descent

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

$$
g = \nabla_w \left[ \frac{1}{N} \sum_{n=1}^{N} \ell_n(w) \right] = \frac{1}{N} \sum_{n=1}^{N} g_n \quad \text{(ignoring regularizer } R(w)) \]

- **Stochastic Gradient Descent** (SGD) approximates $g$ using a single data point

- In iteration $t$, SGD picks a uniformly random $i \in \{1, \ldots, N\}$ and approximate $g$ as

$$
g \approx g_i = \nabla_w \ell_i(w)$$

---

**Stochastic Gradient Descent**

1. Initialize $w$ as $w^{(0)}$
2. Pick a random $i \in \{1, \ldots, N\}$. Update $w$ as follows

$$
w^{(t+1)} = w^{(t)} - \eta_i g_i^{(t)}$$
3. Repeat until convergence
Recap: Mini-batch SGD

- In each iteration, SGD uses a single randomly chosen \( i \in \{1, \ldots, N\} \) to approximate \( g \).
- This results in a large variance in \( g_i \).

\[
g \approx \frac{1}{B} \sum_{b=1}^{B} g_{i_b}
\]

The basic intuition: Averaging helps in variance reduction!
Recap: Mini-batch SGD

- In each iteration, SGD uses a single randomly chosen \( i \in \{1, \ldots, N\} \) to approximate \( g \).
- This results in a large variance in \( g_i \).

![Diagram](https://via.placeholder.com/150)

- 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 iteration, SGD uses a single randomly chosen $i \in \{1, \ldots, N\}$ to approximate $g$
- This results in a large variance in $g_i$

\[
\begin{align*}
g & \approx \frac{1}{B} \sum_{b=1}^{B} g_{i_b}
\end{align*}
\]

- 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
Recap: Mini-batch SGD

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

![Diagram: Illustration of full gradient $\mathbf{g}$ versus stochastic gradient $\mathbf{g}_i$]

- 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

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

- The basic intuition: Averaging helps in variance reduction!
Recap: Mini-batch SGD

- In each iteration, SGD uses a single randomly chosen $i \in \{1, \ldots, N\}$ to approximate $\mathbf{g}$.
- This results in a large variance in $g_i$.

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

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

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
Many ML problems require minimizing non-differentiable functions.
Optimizing Non-differentiable Functions

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

```
<table>
<thead>
<tr>
<th></th>
<th>Differentiable</th>
<th>NON-Differentiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y - w^T x$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\epsilon$-insensitive Loss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>y - w^T x</td>
<td>- \epsilon$</td>
</tr>
</tbody>
</table>
```

Can’t apply standard GD or SGD since gradient isn’t defined at points of non-differentiability
Many ML problems require minimizing non-differentiable functions

Some common examples

- Absolute, $\epsilon$-insensitive loss in regression, several classification loss functions (we will see shortly)

![Diagram illustrating various loss functions](image-url)
Many ML problems require minimizing non-differentiable functions

Some common examples

- **Absolute, \(\epsilon\)-insensitive loss in regression, several classification loss functions** (we will see shortly)

- **Regression/classification loss functions with \(\ell_1\) or \(\ell_p\) \((p < 1)\) regularization**

---

Optimizing Non-differentiable Functions

- **Absolute Loss:** \(|y - w^T x|\)
- **\(\epsilon\)-insensitive Loss:** \(|y - w^T x| - \epsilon\)
- **Perceptron Loss:** \(\max\{0, -yw^T x\}\)
- **Hinge Loss:** \(\max\{0, 1 - yw^T x\}\)

Regression/classification loss functions with \(\ell_1\) or \(\ell_p\) \((p < 1)\) regularization

- Contour of squared \(\ell_2\) norm = 1
- Contour of \(\ell_1\) norm = 1
- Contour of \(\ell_p\) \((p < 1)\) norm = 1

Differentiable

NON-Differentiable
Many ML problems require minimizing non-differentiable functions.

Some common examples

- Absolute, $\epsilon$-insensitive loss in regression, several classification loss functions (we will see shortly)
  - Absolute Loss: $|y - w^T x|$
  - $\epsilon$-insensitive Loss: $|y - w^T x| - \epsilon$
  - ‘Perceptron’ Loss: $\max\{0, -yw^T x\}$
  - Hinge Loss: $\max\{0, 1 - yw^T x\}$

Regression/classification loss functions with $\ell_1$ or $\ell_p$ ($p < 1$) 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 \( \hat{y} = w^T x \)), some common loss functions are
  
  \[ \ell(y, \hat{y}) = (y - w^T x)^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - w^T x| \]

We typically look at the difference between true \( y \) and model's prediction \( w^T x \).

How to formally define loss functions for classification?

We have already looked at the loss function for logistic regression (assuming \( y \in \{-1, +1\} \))

\[ \ell(y, \hat{y}) = \log(1 + \exp(-y w^T x)) \]

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

Are there other loss functions for classification?

Yes, several.
Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = \mathbf{w}^\top \mathbf{x}$), some common loss functions are
  $$
  \ell(y, \hat{y}) = (y - \mathbf{w}^\top \mathbf{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \mathbf{w}^\top \mathbf{x}|
  $$

- We typically look at the difference between true $y$ and model’s prediction $\mathbf{w}^\top \mathbf{x}$.
Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = w^T x$), some common loss functions are
  \[ \ell(y, \hat{y}) = (y - w^T x)^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - w^T x| \]

- We typically look at the difference between true $y$ and model’s prediction $w^T x$

- How to formally define loss functions for classification?
In regression (assuming linear model $\hat{y} = w^T x$), some common loss functions are

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

We typically look at the difference between true $y$ and model’s prediction $w^T x$

How to formally define loss functions for classification?

We have already looked at the loss function for logistic regression (assuming $y \in \{-1, +1\}$)

$$\ell(y, \hat{y}) = \log(1 + \exp(-y w^T x))$$
In regression (assuming linear model $\hat{y} = w^T x$), some common loss functions are

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

We typically look at the difference between true $y$ and model's prediction $w^T x$

How to formally define loss functions for classification?

We have already looked at the loss function for logistic regression (assuming $y \in \{-1, +1\}$)

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

Why does the above make sense? Well, it is large for large misclassifications, small otherwise
Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = w^\top x$), some common loss functions are
  $$
  \ell(y, \hat{y}) = (y - w^\top x)^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - w^\top x|
  $$

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

- How to formally define loss functions for classification?

- We have already looked at the loss function for logistic regression (assuming $y \in \{-1, +1\}$)
  $$
  \ell(y, \hat{y}) = \log(1 + \exp(-yw^\top x))
  $$

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

  ![Graph showing the behavior of the log loss function]

- Are there other loss functions for classification?

Intro to Machine Learning (CS771A)
Optimization Techniques for ML (2)
Interlude: Loss Functions for Classification

- In regression (assuming linear model $\hat{y} = \mathbf{w}^\top \mathbf{x}$), some common loss functions are
  \[ \ell(y, \hat{y}) = (y - \mathbf{w}^\top \mathbf{x})^2 \quad \text{or} \quad \ell(y, \hat{y}) = |y - \mathbf{w}^\top \mathbf{x}| \]

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

- How to formally define loss functions for classification?

- We have already looked at the loss function for logistic regression (assuming $y \in \{-1, +1\}$)
  \[ \ell(y, \hat{y}) = \log(1 + \exp(-y \mathbf{w}^\top \mathbf{x})) \]

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

- Are there other loss functions for classification? Yes, several.
Interlude: Some Loss Functions for (Binary) Classification

0-1 Loss

\[ \mathbb{I}[y w^\top x < 0] \]

(same as)

\[ \mathbb{I}[\text{sign}(w^\top x) \neq y] \]

(0,1)

(0,0)
Interlude: Some Loss Functions for (Binary) Classification

0-1 Loss

\[ \mathbb{I}[y{w^\top}x < 0] \]

(same as)

\[ \mathbb{I}[\text{sign}(w^\top x) \neq y] \]

(0,1)

(0,0) \quad y{w^\top}x

Non-convex

Non-differentiable

NP-Hard to Optimize
Interlude: Some Loss Functions for (Binary) Classification

0-1 Loss
\[ I[yw^\top x < 0] \]
(same as)
\[ I[\text{sign}(w^\top x) \neq y] \]
(0,1)

"Perceptron" Loss
\[ \max\{0, -yw^\top x\} \]
Interlude: Some Loss Functions for (Binary) Classification

0-1 Loss
- Non-convex
- Non-differentiable
- NP-Hard to Optimize

\[ \mathbb{I}[yw^\top x < 0] \]
(same as)
\[ \mathbb{I}[\text{sign}(w^\top x) \neq y] \]
(0,1)

“Perceptron” Loss
- Convex
- Non-differentiable

\[ \max\{0, -yw^\top x\} \]
Interlude: Some Loss Functions for (Binary) Classification

- **0-1 Loss**:\[ \mathbb{I}[y w^\top x < 0] \]
  - Non-convex
  - Non-differentiable
  - NP-Hard to Optimize

- **“Perceptron” Loss**:\[ \max\{0, -y w^\top x\} \]
  - Convex
  - Non-differentiable

- **Log(istic) Loss**:\[ \log(1 + \exp(-y w^\top x)) \]
Interlude: Some Loss Functions for (Binary) Classification

- **0-1 Loss**: $\mathbb{I}[yw^\top x < 0] \quad \mathbb{I}[\text{sign}(w^\top x) \neq y]$
  - Non-convex
  - Non-differentiable
  - NP-Hard to Optimize

- **“Perceptron” Loss**: $\max\{0, -yw^\top x\}$
  - Convex
  - Non-differentiable

- **Logistic Loss**: $\log(1 + \exp(-yw^\top x))$
  - Convex
  - Differentiable
Interlude: Some Loss Functions for (Binary) Classification

0-1 Loss

\[ \mathbb{I}[y w^\top x < 0] \]

(same as)

\[ \mathbb{I}[\text{sign}(w^\top x) \neq y] \]

(0,1)

Non-convex

Non-differentiable

NP-Hard to Optimize

“Perceptron” Loss

\[ \max\{0, -y w^\top x\} \]

Convex

Non-differentiable

Log(istic) Loss

\[ \log(1 + \exp(-y w^\top x)) \]

Convex

Differentiable

Hinge Loss

\[ \max\{0, 1 - y w^\top x\} \]

(0,1)
Interlude: Some Loss Functions for (Binary) Classification

<table>
<thead>
<tr>
<th>Loss Function</th>
<th>Convex</th>
<th>Non-convex</th>
<th>Non-differentiable</th>
<th>NP-Hard to Optimize</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1 Loss</td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Logistic Loss</td>
<td>+</td>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Hinge Loss</td>
<td>+</td>
<td>+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **0-1 Loss**
  - \[ \mathbb{I}[y \mathbf{w}^\top \mathbf{x} < 0] \]
  - (same as)
  - \[ \mathbb{I}[\text{sign}(\mathbf{w}^\top \mathbf{x}) \neq y] \]
  - \((0,1)\)

- **“Perceptron” Loss**
  - \( \max\{0, -y \mathbf{w}^\top \mathbf{x}\} \)
  - Convex
  - Non-differentiable

- **Logistic Loss**
  - \( \log(1 + \exp(-y \mathbf{w}^\top \mathbf{x})) \)
  - Convex
  - Differentiable

- **Hinge Loss**
  - \( \max\{0, 1 - y \mathbf{w}^\top \mathbf{x}\} \)
  - (0,1)
  - Convex
  - Non-differentiable
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 $f(x)$, its subgradient at $x$ is any vector $g$ s.t. $\forall y$

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

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

$$f(y) \geq f(x) + g^T(y - x)$$
Optimizing Non-differentiable Functions

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

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

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

- 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 $f(x)$, its subgradient at $x$ is any vector $g$ s.t. $\forall y$

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

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

- Set of all subgradients of a function $f$ at point $x$ is called the subdifferential denoted as $\partial f(x)$

$$\partial f(x) = \{ g : f(y) \geq f(x) + g^\top (y - x), \ \forall y \}$$
Subgradient Descent: An Example

- Consider linear regression but with $\ell_1$ norm on $\mathbf{w}$ (recall: $\ell_1$ norm promotes a sparse $\mathbf{w}$)

$$
\hat{\mathbf{w}} = \arg \min_{\mathbf{w}} \sum_{n=1}^{N} (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \|\mathbf{w}\|_1
$$
Subgradient Descent: An Example

- Consider linear regression but with $\ell_1$ norm on $\mathbf{w}$ (recall: $\ell_1$ norm promotes a sparse $\mathbf{w}$)

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

- The squared error term is differentiable but the norm $||\mathbf{w}||_1$ is NOT at $w_d = 0$
Subgradient Descent: An Example

- Consider linear regression but with $\ell_1$ norm on $w$ (recall: $\ell_1$ norm promotes a sparse $w$)

$$
\hat{w} = \arg\min_w \sum_{n=1}^{N} (y_n - w^\top x_n)^2 + \lambda ||w||_1
$$

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

- We can use subgradients of $||w||_1$ in this case

$$
g = 2 \sum_{n=1}^{N} (y_n - w^\top x_n)x_n + \lambda t
$$
Subgradient Descent: An Example

- Consider linear regression but with $\ell_1$ norm on $w$ (recall: $\ell_1$ norm promotes a sparse $w$)

$$
\hat{w} = \text{arg min}_w \sum_{n=1}^{N} (y_n - w^\top x_n)^2 + \lambda ||w||_1
$$

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

- We can use subgradients of $||w||_1$ in this case

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

- Here $t$ 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 $\ell_1$ norm on $w$ (recall: $\ell_1$ norm promotes a sparse $w$)

$$\hat{w} = \arg \min_w \sum_{n=1}^{N} (y_n - w^\top x_n)^2 + \lambda ||w||_1$$

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

- We can use subgradients of $||w||_1$ in this case

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

- Here $t$ 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 $\ell_2$ regularizer

\[
\text{Hinge Loss: } \max\{0, 1 - y w^\top x\}
\]

- 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 $\ell_2$ regularizer

\[
\text{Hinge Loss: } \max \{0, 1 - yw^\top x\}
\]

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

- Subgradient $t$ of the hinge loss term will be

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

Hinge Loss: $\max\{0, 1 - yw^\top x\}$
Constrained Optimization

1: Lagrangian based optimization
2: Projected gradient descent
Consider optimizing some function $f(w)$ subject to an inequality constraint on $w$

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

If constraint of the form $g(w) \geq 0$, use $-g(w) \leq 0$
Constrained Optimization: Lagrangian Approach

- Consider optimizing some function $f(w)$ subject to an inequality constraint on $w$
  \[ \hat{w} = \arg \min_w f(w), \quad \text{s.t.} \quad g(w) \leq 0 \]

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

- Note: Can handle multiple inequality and equality constraints too (will see later)
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 \( -g(\mathbf{w}) \leq 0 \)

- Note: Can handle multiple inequality and equality constraints too (will see later)

- Can transform the above constrained problem into an equivalent unconstrained problem
  \[
  \hat{\mathbf{w}} = \arg\min_{\mathbf{w}} f(\mathbf{w}) + c(\mathbf{w})
  \]

\[c(\mathbf{w}) = \max_{\alpha \geq 0} \alpha g(\mathbf{w}) = \begin{cases} 
\infty, & \text{if } g(\mathbf{w}) > 0 \text{ (constraint violated)} \\
0, & \text{if } g(\mathbf{w}) \leq 0 \text{ (constraint satisfied)}
\end{cases}\]
Constrained Optimization: Lagrangian Approach

- Consider optimizing some function $f(w)$ subject to an inequality constraint on $w$
  
  $\hat{w} = \arg\min_w f(w), \text{ s.t. } g(w) \leq 0$

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

- Note: Can handle multiple inequality and equality constraints too (will see later)

- Can transform the above constrained problem into an equivalent unconstrained problem

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

  where we have defined $c(w)$ as

  $c(w) = \max_{\alpha \geq 0} \alpha g(w)$
Constrained Optimization: Lagrangian Approach

Consider optimizing some function \( f(w) \) subject to an inequality constraint on \( w \)

\[
\hat{w} = \arg\min_w f(w), \quad \text{s.t.} \quad g(w) \leq 0
\]

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

Note: Can handle multiple inequality and equality constraints too (will see later)

Can transform the above constrained problem into an equivalent unconstrained problem

\[
\hat{w} = \arg\min_w f(w) + c(w)
\]

where we have defined \( c(w) \) as

\[
c(w) = \max_{\alpha \geq 0} \alpha g(w) = \begin{cases} 
\infty, & \text{if } g(w) > 0 \quad \text{(constraint violated)} \\
0, & \text{if } g(w) \leq 0 \quad \text{(constraint satisfied)}
\end{cases}
\]
Constrained Optimization: Lagrangian Approach

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

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

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

- Note: Can handle multiple inequality and equality constraints too (will see later)

- Can transform the above constrained problem into an equivalent unconstrained problem

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

  where we have defined $c(w)$ as

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

- We can equivalently write the problem as

  $$\hat{w} = \arg\min_w \left\{ f(w) + \max_{\alpha \geq 0} \alpha g(w) \right\}$$
Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

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

The function \( L(w, \alpha) = f(w) + \alpha g(w) \) called the Lagrangian, optimized w.r.t. \( w \) and \( \alpha \).

Primal and Dual problems

\[
\hat{w}_P = \arg\min_{w} \left\{ \arg\max_{\alpha \geq 0} \left\{ f(w) + \alpha g(w) \right\} \right\}
\]

(primal problem)

\[
\hat{w}_D = \arg\max_{\alpha \geq 0} \left\{ \arg\min_{w} \left\{ f(w) + \alpha g(w) \right\} \right\}
\]

(dual problem)

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

For dual solution, \( \alpha_D g(\hat{w}_D) = 0 \) (complementary slackness/Karush-Kuhn-Tucker (KKT) condition)
Constrained Optimization: Lagrangian Approach

So we could write the original problem as

$$\hat{w} = \arg \min_w \left\{ f(w) + \arg \max_{\alpha \geq 0} \alpha g(w) \right\} = \arg \min_w \left\{ \arg \max_{\alpha \geq 0} \left\{ f(w) + \alpha g(w) \right\} \right\}$$
Constrained Optimization: Lagrangian Approach

So we could write the original problem as

\[ \hat{w} = \arg \min_w \left\{ f(w) + \arg \max_{\alpha \geq 0} \alpha g(w) \right\} = \arg \min_w \left\{ \arg \max_{\alpha \geq 0} \left\{ f(w) + \alpha g(w) \right\} \right\} \]

- The function \( L(w, \alpha) = f(w) + \alpha g(w) \) called the Lagrangian, optimized \( w.r.t. w \) and \( \alpha \).
Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

\[ \hat{w} = \arg\min_w \left\{ f(w) + \arg\max_{\alpha \geq 0} \alpha g(w) \right\} = \arg\min_w \left\{ \arg\max_{\alpha \geq 0} \left\{ f(w) + \alpha g(w) \right\} \right\} \]

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

- \( \alpha \) is known as the Lagrange multiplier
Constrained Optimization: Lagrangian Approach

So we could write the original problem as

\[ \hat{w} = \arg \min_w \left\{ f(w) + \arg \max_{\alpha \geq 0} \alpha g(w) \right\} = \arg \min_w \left\{ \arg \max_{\alpha \geq 0} \left\{ f(w) + \alpha g(w) \right\} \right\} \]

The function \( L(w, \alpha) = f(w) + \alpha g(w) \) called the Lagrangian, optimized w.r.t. \( w \) and \( \alpha \)

\( \alpha \) is known as the Lagrange multiplier

Primal and Dual problems

\[ \hat{w}_P = \arg \min_w \left\{ \arg \max_{\alpha \geq 0} \{ f(w) + \alpha g(w) \} \right\} \quad \text{(primal problem)} \]
Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

\[
\hat{w} = \arg \min_w \left\{ f(w) + \arg \max_{\alpha \geq 0} \alpha g(w) \right\} = \arg \min_w \left\{ \arg \max_{\alpha \geq 0} \{ f(w) + \alpha g(w) \} \right\}
\]

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

- \( \alpha \) is known as the Lagrange multiplier

- Primal and Dual problems

\[
\hat{w}_P = \arg \min_w \left\{ \arg \max_{\alpha \geq 0} \{ f(w) + \alpha g(w) \} \right\} \quad \text{(primal problem)}
\]

\[
\hat{w}_D = \arg \max_{\alpha \geq 0} \left\{ \arg \min_w \{ f(w) + \alpha g(w) \} \right\} \quad \text{(dual problem)}
\]
Constrained Optimization: Lagrangian Approach

- So we could write the original problem as

\[ \hat{w} = \arg \min_w \left\{ f(w) + \arg \max_{\alpha \geq 0} \alpha g(w) \right\} = \arg \min_w \left\{ \arg \max_{\alpha \geq 0} \{ f(w) + \alpha g(w) \} \right\} \]

- The function \( L(w, \alpha) = f(w) + \alpha g(w) \) called the Lagrangian, optimized w.r.t. \( w \) and \( \alpha \)

- \( \alpha \) is known as the Lagrange multiplier

- Primal and Dual problems

\[ \hat{w}_P = \arg \min_w \left\{ \arg \max_{\alpha \geq 0} \{ f(w) + \alpha g(w) \} \right\} \quad \text{(primal problem)} \]

\[ \hat{w}_D = \arg \max_{\alpha \geq 0} \left\{ \arg \min_w \{ f(w) + \alpha g(w) \} \right\} \quad \text{(dual problem)} \]

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

- So we could write the original problem as

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

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

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

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

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

- We can also have multiple inequality and equality constraints

\[
\hat{w} = \arg\min_w f(w) \\
\text{s.t.} \quad g_i(w) \leq 0, \quad i = 1, \ldots, K \\
\quad h_j(w) = 0, \quad j = 1, \ldots, L
\]
Constrained Optimization: Lagrangian with Multiple Constraints

- We can also have multiple inequality and equality constraints

\[
\hat{w} = \arg \min_w f(w) \quad \text{s.t.} \quad g_i(w) \leq 0, \quad i = 1, \ldots, K \\
\quad \quad h_j(w) = 0, \quad j = 1, \ldots, L
\]

- Introduce Lagrange multipliers \( \alpha = (\alpha_1, \ldots, \alpha_K) \geq 0 \) and \( \beta = (\beta_1, \ldots, \beta_L) \)
Constrained Optimization: Lagrangian with Multiple Constraints

- We can also have multiple inequality and equality constraints

\[ \hat{w} = \arg\min_w f(w) \]

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

\[ h_j(w) = 0, \quad j = 1, \ldots, L \]

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

- The Lagrangian based primal and dual problems will be

\[ \hat{w}_P = \arg\min_w \{ \arg\max_{\alpha \geq 0, \beta} \{ f(w) + \sum_{i=1}^K \alpha_i g_i(w) + \sum_{j=1}^L \beta_j h_j(w) \} \} \]
We can also have multiple inequality and equality constraints

\[
\hat{w} = \arg\min_w f(w) \\
s.t. \quad g_i(w) \leq 0, \quad i = 1, \ldots, K \\
h_j(w) = 0, \quad j = 1, \ldots, L
\]

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

- The Lagrangian based primal and dual problems will be

\[
\hat{w}_P = \arg\min_w \{ \arg\max_{\alpha \geq 0, \beta} \left\{ f(w) + \sum_{i=1}^{K} \alpha_i g_i(w) + \sum_{j=1}^{L} \beta_j h_j(w) \right\} \} \\
\hat{w}_D = \arg\max_{\alpha \geq 0, \beta} \{ \arg\min_w \left\{ f(w) + \sum_{i=1}^{K} \alpha_i g_i(w) + \sum_{j=1}^{L} \beta_j h_j(w) \right\} \}
\]
Lagrangian based Optimization: An Example

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

$$p(y|\pi) = \text{multinoulli}(\pi_1, \pi_2, \ldots, \pi_K) = \prod_{k=1}^{K} \pi_k^{[y=k]}, \quad \text{s.t.} \quad \sum_{k=1}^{K} \pi_k = 1$$
Lagrangian based Optimization: An Example

- Consider the generative classification model with \( K \) classes
- Suppose we want to estimate the parameters of class-marginal \( p(y) \)
  \[
p(y | \pi) = \text{multinoulli}(\pi_1, \pi_2, \ldots, \pi_K) = \prod_{k=1}^{K} \pi_k^{[y=k]}, \quad \text{s.t.} \quad \sum_{k=1}^{K} \pi_k = 1
\]
- Given \( N \) observations \( \{x_n, y_n\}_{n=1}^{N} \), the negative log-likelihood for class marginal
  \[
f(\pi) = - \sum_{n=1}^{N} \log p(y_n | \pi)
\]
Lagrangian based Optimization: An Example

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

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

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

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

- We have an equality constraint $\sum_{k=1}^{K} \pi_k - 1 = 0$
Consider the generative classification model with $K$ classes

Suppose we want to estimate the parameters of class-marginal $p(y)$

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

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

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

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

The Lagrangian for this problem will be

$$\mathcal{L}(\pi, \beta) = f(\pi) + \beta(\sum_{k=1}^{K} \pi_k - 1)$$
Lagrangian based Optimization: An Example

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

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

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

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

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

\[
\mathcal{L}(\pi, \beta) = f(\pi) + \beta\left(\sum_{k=1}^{K} \pi_k - 1\right)
\]

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

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

$$\hat{w} = \arg \min_w \mathcal{L}(w), \quad \text{subject to} \quad w \in \mathcal{C}$$
Projected Gradient Descent

- Suppose our problem requires the parameters to lie within a set $C$
  \[
  \hat{w} = \arg \min_w \mathcal{L}(w), \quad \text{subject to} \quad w \in C
  \]
- Projected GD is very similar to GD with an extra projection step

Diagram:
- $w^{(t)}$ to $w^{(t+1)}$ via $z^{(t+1)}$
- Projection step on $C$
Projected Gradient Descent

- Suppose our problem requires the parameters to lie within a set $C$
  \[ \hat{w} = \arg \min_w L(w), \quad \text{subject to} \quad w \in C \]

- 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: \( z^{(t+1)} = w^{(t)} - \eta_t g^{(t)} \)

- Check \( z^{(t+1)} \) for the constraints
  - If \( z^{(t+1)} \in C \), \( w^{(t+1)} = z^{(t+1)} \)
  - If \( z^{(t+1)} \notin C \), project on the constraint set: \( w^{(t+1)} = \Pi_C[z^{(t+1)}] \)
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

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

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

- For some sets \( C \), the projection step is easy/trivial

\( \begin{align*}
C : \text{Unit radius } \ell_2 \text{ ball} \\
\text{Projection} = \text{Normalize to unit length}
\end{align*} \)

\( \begin{align*}
C : \text{Set of non-negative reals} \\
\text{Projection} = \text{Set negative values to 0}
\end{align*} \)
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:

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

For some sets $C$, the projection step is easy/trivial:

- For $C$ : Unit radius $\ell_2$ ball
  - Projection = Normalize to unit length

- For $C$ : Set of non-negative reals
  - Projection = Set negative values to 0

For some other sets $C$, the projection step may be a bit more involved.
Co-ordinate Descent (CD)

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

$$w^{(t+1)} = w^{(t)} - \eta_t g^{(t)}$$
Co-ordinate Descent (CD)

- Standard GD update for \( w \in \mathbb{R}^D \) at each step
  \[
  w^{(t+1)} = w^{(t)} - \eta_t g^{(t)}
  \]

- 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., \( w^\top x \)) to avoid \( O(D) \) cost in gradient computation
Co-ordinate Descent (CD)

- Standard GD update for $w \in \mathbb{R}^D$ at each step
  \[ w^{(t+1)} = w^{(t)} - \eta_t g^{(t)} \]

- 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 $\mathbf{w} \in \mathbb{R}^D$ at each step
  \[ \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)} \]

- CD: Each step update one component (co-ordinate) at a time, keeping all others fixed
  \[ w^{(t+1)}_d = w^{(t)}_d - \eta_t g^{(t)}_d \]

- 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 $\mathbf{w} \in \mathbb{R}^D$ at each step
  \[ \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)} \]

- 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 \( \mathbf{w} \in \mathbb{R}^D \) at each step
  \[
  \mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \mathbf{g}^{(t)}
  \]

- 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 \( w \in \mathbb{R}^D \) at each step
  \[
  w^{(t+1)} = w^{(t)} - \eta_t g^{(t)}
  \]

- 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 $\mathbf{w} \in \mathbb{R}^D$ at each step

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

- 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., $\mathbf{w}^\top \mathbf{x}$) to avoid $O(D)$ cost in gradient computation
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
\[ w_1 \in \mathbb{R}^D \text{ and } w_2 \in \mathbb{R}^D \]
\[ \hat{w}_1, \hat{w}_2 = \arg \min_{w_1, w_2} L(w_1, w_2) \]

Jointly optimizing w.r.t. \( w_1 \) and \( w_2 \) 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. \( w_1 \) and \( w_2 \)

Initialize one of the variables, e.g., \( w_2 = w_2(0), t = 0 \)

Solve \( w_1(t+1) = \arg \max_{w_1} L(w_1, w_2(t)) \)

Solve \( w_2(t+1) = \arg \max_{w_2} L(w_1(t+1), w_2) \)

\( t = t + 1 \). Repeat until convergence

Usually converges to a local optima of \( L(w_1, w_2) \). 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.
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 \( \mathbf{w}_1 \in \mathbb{R}^D \) and \( \mathbf{w}_2 \in \mathbb{R}^D \)

\[
\{ \hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2 \} = \arg \min_{\mathbf{w}_1, \mathbf{w}_2} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2)
\]

Jointly optimizing w.r.t. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \) 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. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \)

Initialize one of the variables, e.g., \( \mathbf{w}_2 = \mathbf{w}_2(0), t = 0 \)

Solve \( \mathbf{w}_1(t+1) = \arg \max_{\mathbf{w}_1} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2(t)) \)

Solve \( \mathbf{w}_2(t+1) = \arg \max_{\mathbf{w}_2} \mathcal{L}(\mathbf{w}_1(t+1), \mathbf{w}_2) \)

\( t = t + 1 \). Repeat until convergence

Usually converges to a local optima of \( \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2) \). 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.
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 $\mathbf{w}_1 \in \mathbb{R}^D$ and $\mathbf{w}_2 \in \mathbb{R}^D$

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

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

Often, knowing the value of one may make optimization w.r.t. the other easy (sometimes even closed form). We can therefore follow an alternating scheme to optimize w.r.t. $\mathbf{w}_1$ and $\mathbf{w}_2$:

1. Initialize one of the variables, e.g., $\mathbf{w}_2 = \mathbf{w}_2(0)$, $t = 0$
2. Solve $\mathbf{w}_1(t+1) = \arg \max_{\mathbf{w}_1} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2(t))$
3. Solve $\mathbf{w}_2(t+1) = \arg \max_{\mathbf{w}_2} \mathcal{L}(\mathbf{w}_1(t+1), \mathbf{w}_2)$
4. $t = t + 1$. Repeat until convergence

Usually converges to a local optima of $\mathcal{L}(\mathbf{w}_1, \mathbf{w}_2)$. 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.
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 \( \mathbf{w}_1 \in \mathbb{R}^D \) and \( \mathbf{w}_2 \in \mathbb{R}^D \)

\[
\{ \hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2 \} = \arg \min_{\mathbf{w}_1, \mathbf{w}_2} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2)
\]

- Jointly optimizing w.r.t. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \) 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 $w_1 \in \mathbb{R}^D$ and $w_2 \in \mathbb{R}^D$

$$\{\hat{w}_1, \hat{w}_2\} = \arg \min_{w_1, w_2} \mathcal{L}(w_1, w_2)$$

- Jointly optimizing w.r.t. $w_1$ and $w_2$ 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. $w_1$ and $w_2$
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 $\mathbf{w}_1 \in \mathbb{R}^D$ and $\mathbf{w}_2 \in \mathbb{R}^D$

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

- Jointly optimizing w.r.t. $\mathbf{w}_1$ and $\mathbf{w}_2$ 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. $\mathbf{w}_1$ and $\mathbf{w}_2$
  - Initialize one of the variables, e.g., $\mathbf{w}_2 = \mathbf{w}_2^{(0)}, t = 0$
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 $\mathbf{w}_1 \in \mathbb{R}^D$ and $\mathbf{w}_2 \in \mathbb{R}^D$

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

- Jointly optimizing w.r.t. $\mathbf{w}_1$ and $\mathbf{w}_2$ 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. $\mathbf{w}_1$ and $\mathbf{w}_2$
  - Initialize one of the variables, e.g., $\mathbf{w}_2 = \mathbf{w}_2^{(0)}, t = 0$
  - Solve $\mathbf{w}_1^{(t+1)} = \arg\max_{\mathbf{w}_1} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2^{(t)})$
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 \( \mathbf{w}_1 \in \mathbb{R}^D \) and \( \mathbf{w}_2 \in \mathbb{R}^D \)

\[
\{ \hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2 \} = \arg \min_{\mathbf{w}_1, \mathbf{w}_2} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2)
\]

- Jointly optimizing w.r.t. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \) 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. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \)
  - Initialize one of the variables, e.g., \( \mathbf{w}_2 = \mathbf{w}_2^{(0)}, t = 0 \)
  - Solve \( \mathbf{w}_1^{(t+1)} = \arg \max_{\mathbf{w}_1} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2^{(t)}) \)
  - Solve \( \mathbf{w}_2^{(t+1)} = \arg \max_{\mathbf{w}_2} \mathcal{L}(\mathbf{w}_1^{(t+1)}, \mathbf{w}_2) \)

Usually converges to a local optima of \( \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2) \). 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.
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 $w_1 \in \mathbb{R}^D$ and $w_2 \in \mathbb{R}^D$

$$\{\hat{w}_1, \hat{w}_2\} = \arg \min_{w_1, w_2} \mathcal{L}(w_1, w_2)$$

- Jointly optimizing w.r.t. $w_1$ and $w_2$ 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. $w_1$ and $w_2$
  - Initialize one of the variables, e.g., $w_2 = w_2^{(0)}$, $t = 0$
  - Solve $w_1^{(t+1)} = \arg \max_{w_1} \mathcal{L}(w_1, w_2^{(t)})$
  - Solve $w_2^{(t+1)} = \arg \max_{w_2} \mathcal{L}(w_1^{(t+1)}, w_2)$
  - $t = t + 1$. Repeat until convergence

Usually converges to a local optima of $\mathcal{L}(w_1, w_2)$. 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.
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 \( \mathbf{w}_1 \in \mathbb{R}^D \) and \( \mathbf{w}_2 \in \mathbb{R}^D \):

\[
\{ \hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2 \} = \text{arg min}_{\mathbf{w}_1, \mathbf{w}_2} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2)
\]

- Jointly optimizing w.r.t. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \) 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. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \):
  - Initialize one of the variables, e.g., \( \mathbf{w}_2 = \mathbf{w}_2^{(0)}, t = 0 \)
  - Solve \( \mathbf{w}_1^{(t+1)} = \text{arg max}_{\mathbf{w}_1} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2^{(t)}) \)
  - Solve \( \mathbf{w}_2^{(t+1)} = \text{arg max}_{\mathbf{w}_2} \mathcal{L}(\mathbf{w}_1^{(t+1)}, \mathbf{w}_2) \)
  - \( t = t + 1 \). Repeat until convergence.
- Usually converges to a local optima of \( \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2) \). 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 \( \mathbf{w}_1 \in \mathbb{R}^D \) and \( \mathbf{w}_2 \in \mathbb{R}^D \)
  \[
  \{ \hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2 \} = \arg \min_{\mathbf{w}_1, \mathbf{w}_2} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2)
  \]
- Jointly optimizing w.r.t. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \) 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. \( \mathbf{w}_1 \) and \( \mathbf{w}_2 \)
  - Initialize one of the variables, e.g., \( \mathbf{w}_2 = \mathbf{w}_2^{(0)} \), \( t = 0 \)
  - Solve \( \mathbf{w}_1^{(t+1)} = \arg \max_{\mathbf{w}_1} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2^{(t)}) \)
  - Solve \( \mathbf{w}_2^{(t+1)} = \arg \max_{\mathbf{w}_2} \mathcal{L}(\mathbf{w}_1^{(t+1)}, \mathbf{w}_2) \)
  - \( t = t + 1 \). Repeat until convergence
- Usually converges to a local optima of \( \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2) \). 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 $\mathbf{w}_1 \in \mathbb{R}^D$ and $\mathbf{w}_2 \in \mathbb{R}^D$

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

- Jointly optimizing w.r.t. $\mathbf{w}_1$ and $\mathbf{w}_2$ 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. $\mathbf{w}_1$ and $\mathbf{w}_2$
  - Initialize one of the variables, e.g., $\mathbf{w}_2 = \mathbf{w}_2^{(0)}$, $t = 0$
  - Solve $\mathbf{w}_1^{(t+1)} = \arg \max_{\mathbf{w}_1} \mathcal{L}(\mathbf{w}_1, \mathbf{w}_2^{(t)})$
  - Solve $\mathbf{w}_2^{(t+1)} = \arg \max_{\mathbf{w}_2} \mathcal{L}(\mathbf{w}_1^{(t+1)}, \mathbf{w}_2)$
  - $t = t + 1$. Repeat until convergence
- Usually converges to a local optima of $\mathcal{L}(\mathbf{w}_1, \mathbf{w}_2)$. 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)

Newton's method is one such method that uses second-order information. At each point, approximate the function by its quadratic approximation and minimize it. Does not rely on gradient to choose $w_{t+1}$, instead, each step directly jumps to the minima of quadratic approximation.
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

\[ w^{(t+1)} \]

Doesn’t rely on gradient to choose \( w^{(t+1)} \)
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.

\[ w^{(t+1)} \]

- Doesn’t rely on gradient to choose \( w^{(t+1)} \).
- Instead, each step directly jumps to the minima of quadratic approximation.
The quadratic (Taylor) approximation of $f(w)$ at $w^{(t)}$ is given by

$$\tilde{f}(w) = f(w^{(t)}) + \nabla f(w^{(t)})^\top (w - w^{(t)}) + \frac{1}{2} (w - w^{(t)})^\top \nabla^2 f(w^{(t)})(w - w^{(t)})$$
Second-Order Methods: Newton’s Method

- The quadratic (Taylor) approximation of $f(w)$ at $w^{(t)}$ is given by
  \[ \tilde{f}(w) = f(w^{(t)}) + \nabla f(w^{(t)})^\top (w - w^{(t)}) + \frac{1}{2} (w - w^{(t)})^\top \nabla^2 f(w^{(t)})(w - w^{(t)}) \]

- The minimizer of this quadratic approximation is (exercise: verify)
  \[ \hat{w} = \arg \min_w \tilde{f}(w) = w^{(t)} - (\nabla^2 f(w^{(t)}))^{-1} \nabla f(w^{(t)}) \]
Second-Order Methods: Newton’s Method

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

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

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

\[
\hat{w} = \arg\min_w \tilde{f}(w) = w^{(t)} - (\nabla^2 f(w^{(t)}))^{-1} \nabla f(w^{(t)})
\]

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

\[
w^{(t+1)} = w^{(t)} - (\nabla^2 f(w^{(t)}))^{-1} \nabla f(w^{(t)})
\]

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

Very fast if \( f(w) \) 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.
Second-Order Methods: Newton’s Method

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

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

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

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

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

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

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

- Very fast if $f(w)$ is convex. But expensive due to Hessian computation/inversion.
Second-Order Methods: Newton’s Method

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

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

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

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

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

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

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

- Very fast if $f(w)$ 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

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
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 \( \text{GD} + \text{chain rule} \) 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