Optimization Methods

Seminar Principles of Data Mining and Learning Algorithms

Anna-Maria Meschede and Jan Polster

November 10th 2020

Outline

- Gradient-Based Optimization (4.3)
- Constrained Optimization (4.4)
- Example: Linear Least Squares (4.4)
- Gradient-Based Learning (6.2)
- Back-Propagation Algorithms (6.5)

Gradient-Based Optimization

Gradient-Based Optimization

- Optimization methods widely used for deep learning algorithms
- Given $f : \mathbb{R}^n \to \mathbb{R}$ find $x^* := \underset{x \in \mathbb{R}^n}{\operatorname{argmin}} f(x)$
- **Idea:** Start at initial value x_0 and iteratively move in direction of steepest descent u until convergence.
- Update $x_{i+1} \leftarrow x_i + \varepsilon u$
  - how to find u?
  - how to find stepsize ε?

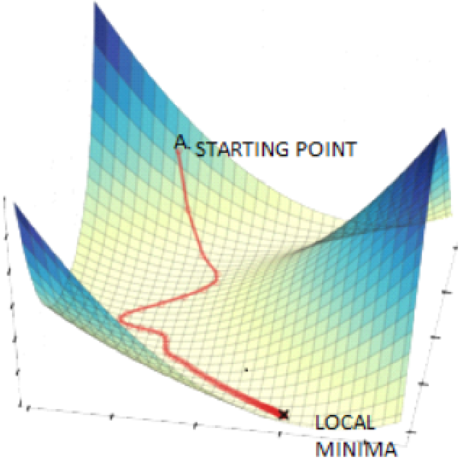


Figure: https://www.hackerearth.com/blog/developers/3-types-gradient-descent-algorithms-small-large-data-sets/

Gradient and directional Derivative

- **Partial derivative** $\frac{\partial}{\partial x_i} f(x)$: derivative of f w.r.t. x_i
- **Gradient** $\nabla_x f(x) := \left(\frac{\partial}{\partial x_1} f(x), \ldots, \frac{\partial}{\partial x_n} f(x)\right)^\intercal$
- **Directional derivative** in direction u:
  - $\frac{\partial}{\partial \alpha} f(x + \alpha u)$ evaluated at $\alpha = 0$
  - Equal to $u^\intercal \nabla_x f(x)$
  - Want to find direction u with minimal directional derivative to minimize f.

$\rightarrow$ Find $\underset{u, \|u\|=1}{\operatorname{argmin}}\ u^\intercal \nabla_x f(x)$.

Direction of Steepest Descent

$$\begin{aligned} \underset{u, \|u\|=1}{\operatorname{argmin}} \; u^\intercal \nabla_x f(x) &= \underset{u, \|u\|=1}{\operatorname{argmin}} \; \|u\|_2 \|\nabla_x f(x)\|_2 \cos(\alpha) \\ &= \underset{u, \|u\|=1}{\operatorname{argmin}} \cos(\alpha) \end{aligned}$$

- α: angle between u and $\nabla_x f(x)$
- $\cos(\alpha)$ minimized when u points in opposite direction of gradient
- $x_{i+1} = x_i - \epsilon \nabla f(x)$

Jacobian Matrix

- Given $f : \mathbb{R}^n \to \mathbb{R}^m$
- f consists of m functions $f_1, \ldots, f_m : \mathbb{R}^n \to \mathbb{R}$

$$f(x) = \begin{bmatrix} f_1(x) \\ \ldots \\ f_m(x) \end{bmatrix}^\intercal$$

- **Jacobian Matrix**: $J \in \mathbb{R}^{m \times n}$, $(J)_{i,j} = \frac{\partial f_i}{\partial x_j}$

$$\to J = \begin{bmatrix} | & & | \\ \nabla_x f_1 & \ldots & \nabla_x f_m \\ | & & | \end{bmatrix}^\intercal$$

Hessian Matrix

- Contains all partial second order derivatives
- Curvature of f
- $H \in \mathbb{R}^{n \times n}$, $H(f)(x)_{i,j} := \frac{\partial}{\partial x_i \partial x_j} f(x)$

$\rightarrow$ Second order derivative in direction u at x: $u^\intercal H(f)(x)u$
$\rightarrow$ Symmetric for continous derivatives
$\rightarrow$ $u^\intercal H(f)(x)u$ weighted average of eigenvalues

Optimal Stepsize ε

Second order Taylor Approximation:

- Let $g := \nabla_x f(x^{(i)})$, $H := H(f)(x^{(i)})$

$$
\begin{aligned}
f(x^{(i+1)}) \approx & f(x^{(i)}) + (x^{(i+1)} - x^{(i)})g \\
& + \frac{1}{2}(x^{(i+1)} - x^{(i)})^\intercal H(x^{(i+1)} - x^{(i)}) \\
= & f(x^{(i)}) - \varepsilon g^\intercal g + \frac{1}{2}\varepsilon^2 g^\intercal H g
\end{aligned}
$$

- When $g^\intercal H g$ is 0 or negative increase ε
- When $g^\intercal H g$ is positive set $\varepsilon^* = \frac{g^\intercal g}{g^\intercal H g}$

Issues of Gradient Descent

- **Ill-conditioned Hessian** leads to poorly performing gradient descent
- Condition $\kappa(H) = \left|\frac{\lambda_{max}}{\lambda_{min}}\right|$ shows how much second derivatives differ from each other.
- Fast increase of derivative in one direction, slow decrease in another.

$\rightarrow$ Solve problems by using Newton's Method

Newton's Method

Second order Taylor approximation of f:

$$
\begin{aligned}
f(x^{(i+1)}) \approx & f(x^{(i)}) + (x - x^{(i)})^\intercal \nabla_x f(x^{(i)}) \\
& + \frac{1}{2}(x - x^{(i)})^\intercal H(f)(x^{(i)})(x - x^{(i)})
\end{aligned}
$$

$\rightarrow$ Optimal: $x^{(i+1)} = x^{(i)} - H(f)(x^{(i)})^{-1} \nabla_x f(x^{(i)})$

Constrained Optimization

Constrained Optimization

- Minimize $f : \mathbb{R}^n \to \mathbb{R}$ with additional conditions
- **Constraints**: $g_i(x) \leq 0$ for $i = 1, ..., m$
  $h_i(x) = 0$ for $j = 1, ..., k$
  $g_i, h_j : \mathbb{R}^n \to \mathbb{R}$
- **Idea:** Translate into unconstrained problem.
- **KKT-approach:**
  - Lagrangian $\mathcal{L}(x, \lambda, \mu) := f(x) + \sum_i \lambda_i g_i(x) + \sum_j \mu_j h_j(x)$
    $\lambda \in \mathbb{R}^m_{\geq 0}, \ \mu \in \mathbb{R}^k$
  - Find $\min_x \max_\mu \max_{\lambda, \lambda \geq 0} \mathcal{L}(x, \lambda, \mu)$

Necessary Conditions for local Minimum

Want to find (x^*, λ^*, μ^*) s.t.

- All constraints are satisfied.
- $\nabla_x \mathcal{L}(x^*, \lambda^*, \mu^*) = 0$
- $\lambda_i^* \geq 0,\ \lambda_i^* g_i(x^*) = 0$ for $i = 1, ..., m$

Example: Linear Least Squares

Example: Least Linear Squares

Minimize $f(x) = \frac{1}{2}\|Ax - b\|^2,\ f : \mathbb{R}^n \to \mathbb{R},\ A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^m$

- **Gradient descent:** $-\nabla_x f(x) = A^\intercal(Ax - b)$
- **Newton's method:** Converges in 1 step.
- **KKT-approach:** Suppose $x^\intercal x \leq 1$

$$\rightarrow \mathcal{L}(x, \lambda) = f(x) + \lambda(x^\intercal x - 1)$$
$$\rightarrow\ x^* = (A^\intercal A + 2\lambda I)^{-1} A^\intercal b$$

Gradient-Based Learning

Deep Feedforward Networks

- A **deep feedforward network**, feedforward neural network or multilayer perceptron (MLP) is the quintessential deep learning model.
- **Goal**: approximate some function $f^\star$
- Feedforward network defines a **mapping** $y = f(x; \theta)$ and **learns parameter** θ with best approximation.
- typically represented by a composition of functions $f(x) = f_3(f_2(f_1(x)))$
  - f_i: i-th layer,
  - last layer: output layer

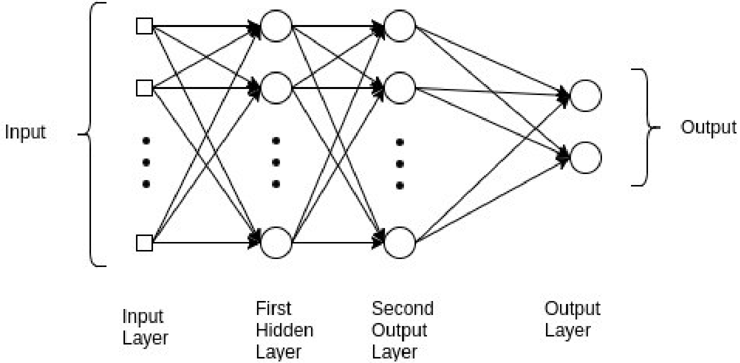


Figure: https://medium.com/@AI_with_Kain/understanding-of-multilayer-perceptron-mlp-8f179c4a135f

Gradient-Based Learning: Motivation

- High descriptive power of neural networks leads to more **complicated loss functions** which are generally **nonconvex**.
- Minimizing nonconvex functions typically involves an **iterative, gradient-based** approach.
  - no global convergence guarantee
  - sensitive to starting point
  - might stop at a local minimum
- **Task**:
  - choose cost function
  - find representation of output according to the model
  - compute gradient efficiently

Cost Functions

- Most cases: parametric model defines **distribution** $p(y|x;\theta)$, we use the principle of **maximum likelihood**.
- **Equivalent to**: minimizing cross-entropy between training data and model distribution:

$$J(\theta) = -\mathbb{E}_{x,y\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(y|x)$$

  - θ : model parameter
  - $\hat{p}_{\text{data}}$: empirical distribution w.r.t. training data
- Specifying a model automatically determines the cost function

Output Units

Setting:

- feedworward network produces hidden features $h = f(x; \theta)$.
- output layer then has to transform h to an appropriate result (w.r.t. to the task)

Output Units

Example 1: Sigmoid Units for Bernoulli Output Distributions

- Predict value of **binary** variable y.
- Neural net needs to predict $P(y = 1 \mid x)$, i.e. output needs to lie in $[0, 1]$.

Possible solution:

- use linear unit and threshold its value:

$$P(y = 1 \mid x) = \max\{0, \min\{1, w^\intercal h + b\}\}$$

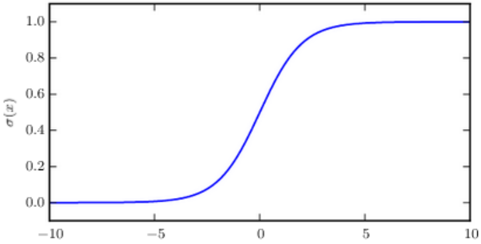
- bad idea for gradient descent since gradient is 0 for values outside of $[0, 1]$

Better solution:

- compute $z = w^\intercal h + b$ in linear layer, output:

$$P(y = 1 \mid x) = \sigma(z),$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ "logistic sigmoid function":



Sigmoid Units for Bernoulli Output Distributions

Motivation of sigmoid function, $\sigma(x) = \frac{1}{1+\exp(-x)}$:

- **Goal**: define a probability distribution $P(y)$ using $z = w^\intercal h + b$
- Start from an unnormalized distribution $\tilde{P}(y)$ and the assumption $\log \tilde{P}(y) = yz$.

$$\tilde{P}(y) = \exp(yz),$$

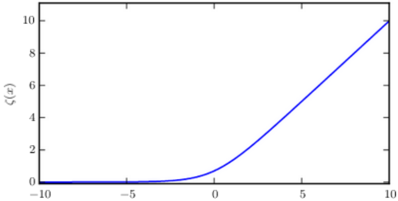
$$P(y) = \frac{\exp(yz)}{\sum_{y'=0,1} \exp(y'z)},$$

$$P(y) = \sigma((2y-1)z).$$

Cost function for maximum likelihood learning:

$$\begin{aligned} J(\theta) &= -\log P(y|x) \\ &= -\log \sigma((2y-1)z) \\ &= \zeta((1-2y)z). \end{aligned}$$

- $\zeta(x) = \log(1 + \exp(x))$ "softplus function"



- $J(\theta)$ has good properties for gradient descent:
  - $y = 1$: $\zeta(-z)$ saturates for very positive z
  - $y = 0$: $\zeta(z)$ saturates for very negative z

Output Units

Example 2: Softmax Units for Multinoulli Output Distributions

- **Generalize** to the case of a discrete variable y with **n values**, i.e. produce a vector $\hat{y}$ with $\hat{y}_i = P(y = i|x)$.
- **Approach:** a linear layer predicts unnormalized $\log$-probabilities:

$$z = W^\intercal h + b,$$
$$z_i = \log \tilde{P}(y = i|x).$$

- output:

$$\hat{y}_i = \text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j exp(z_j)}$$

Maximum likelihood training:

- Maximizing log-likelihood:

$$\begin{aligned}\log P(y=i|x) &= \log \text{softmax}(z)_i \\ &= \log \frac{\exp(z_i)}{\sum_j exp(z_j)} \\ &= z_i - \log \sum_j \exp(z_j)\end{aligned}$$

- $log \sum_j exp(z_j) \approx max_j z_j$
- Incorrect answers (i.e. small z_i on the correct classification i) are penalized the most.
- If the correct answer $y = i$ has the highest input, i.e. $z_i = \max_j z_j$ both terms roughly cancel.

Back-Propagation Algorithms

Back-Propagation Algorithms

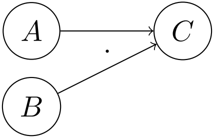
- Feedforward neural network: assigns $x \mapsto \hat{y}$
- Want to minimize cost function $J(\theta)$
- **Back-Propagation:** computes $\nabla_\theta f(x;\theta)$ for given $f : \mathbb{R}^n \to \mathbb{R}$ by letting information flow backwards through network

$\to$ Compute $\nabla_\theta J(\theta)$ this way.

Computational Graphs

Nodes: represent variables
Edges: represent operations (simple functions)

Example: $C = f(A, B) = AB$



Chain Rule of Calculus

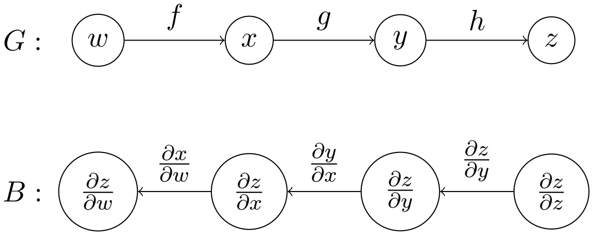
- Compute derivative of composed functions
$$y = f(x),\ z = g(y) = g(f(x))$$

- **1-dim:** $f, g : \mathbb{R} \to \mathbb{R}$
$$\to \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- **n-dim:** $f : \mathbb{R}^n \to \mathbb{R}^m,\ g : \mathbb{R}^m \to \mathbb{R}$
$$\to \frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Example: Back-Propagation

$$x = f(x),\ \ y = g(x),\ \ z = h(y)$$



$$\rightarrow \frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$

Back-Propagation in Fully Connected MLP's

Algorithm Forward Propagation

Input Network with depth l, (x, y), $W^{(i)}, b^{(i)}$ for $i = 1, \dots, l$

1: $h^{(0)} = x$
2: **for** $i = 1, \dots, l$ **do**
3: $\quad a^{(i)} \leftarrow W^{(i)} h^{(i-1)} + b^{(i)}$
4: $\quad h^{(i)} \leftarrow f(a^{(i)})$
5: **end for**
6: $\hat{y} \leftarrow h^{(l)}$
7: $J \leftarrow L(\hat{y}, y) + \lambda \Omega(\theta)$

Back-Propagation in Fully Connected MLP's

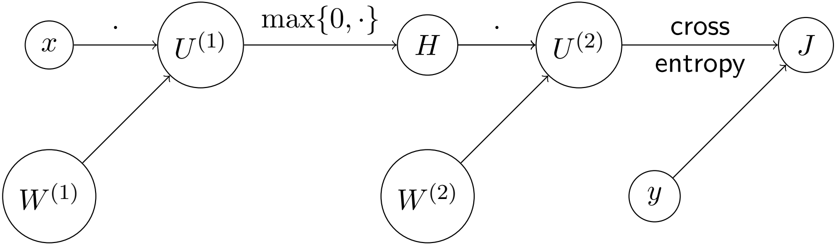
Algorithm Back-Propagation

1: $g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} L(\hat{y}, y)$
2: **for** $i = l, \ldots, 1$ **do**
3: $\quad g \leftarrow \nabla_{a^{(i)}} J = f'(a^{(i)})^\intercal g$
4: $\quad \nabla_{b^{(i)}} J \leftarrow g + \lambda \nabla_{b^{(i)}} \Omega(\theta)$
5: $\quad \nabla_{W^{(i)}} J \leftarrow g h^{(i-1)^\intercal} + \lambda \nabla_{W^{(i)}} \Omega(\theta)$
6: $\quad g \leftarrow \nabla_{h^{(i-1)}} J = W^{(i)^\intercal} g$
7: **end for**

$\rightarrow$ Computation effort linear in number of edges

Example: Back-Propagation for MLP Training

Want to compute $\nabla_{W^{(1)}} J, \ \nabla_{W^{(2)}} J$



$\rightarrow \ \nabla_{W^{(2)}} J = \nabla_{U^{(2)}} J H^\intercal$

$\rightarrow \ \nabla_{W^{(1)}} J = \nabla_{U^{(1)}} J x^\intercal$

Thank you for your attention!

References

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.