Automatic Differentiation

Deep Learning KU (DAT.C302UF)

Simon Hitzginger

Oct 29, 2025

Institute of Machine Learning and Neural Computation
Graz University of Technology, Austria

The Goal of Supervised Learning

Supervised Learning – The Setup

Training Data

- Given **training dataset** $\mathcal{D}_{\text{train}} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}$
- **Features** $\mathbf{x} \in \mathbb{R}^d$

Supervised Learning – The Setup

Training Data

- Given **training dataset** $\mathcal{D}_{\text{train}} = \{(\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})\}$
- **Features** $\mathbf{x} \in \mathbb{R}^d$

Classification

$$y \in \mathcal{C}$$

e.g., $\mathcal{C} = \{0, 1, \ldots, 9\}$

Regression

$$y \in \mathbb{R} \quad \text{or} \quad \mathbf{y} \in \mathbb{R}^K$$

Model 🤖

- We wish to find a **model** $f_{\boldsymbol{\theta}}$ that predicts y from $\mathbf{x}$
  - $\boldsymbol{\theta} \in \mathbb{R}^n$ are the **model parameters**
- Can be a **point estimate** (e.g. if $y \in \mathbb{R}$):

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \hat{y}$$

- Or a **distribution** over outputs (e.g. if $y \in \mathcal{C} = \{c_1, \ldots, c_k\}$):

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \left(p_{\boldsymbol{\theta}}(y = c_1 \mid \mathbf{x}), \ldots, p_{\boldsymbol{\theta}}(y = c_k \mid \mathbf{x})\right)^\top$$

Model 🤖

- We wish to find a **model** $f_{\boldsymbol{\theta}}$ that predicts y from $\mathbf{x}$
  - $\boldsymbol{\theta} \in \mathbb{R}^n$ are the **model parameters**
- Can be a **point estimate** (e.g. if $y \in \mathbb{R}$):

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \hat{y}$$

- Or a **distribution** over outputs (e.g. if $y \in \mathcal{C} = \{c_1, \ldots, c_k\}$):

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \left(p_{\boldsymbol{\theta}}(y = c_1 \mid \mathbf{x}), \ldots, p_{\boldsymbol{\theta}}(y = c_k \mid \mathbf{x})\right)^\top$$

Loss Function 🤦

- We specify a **loss function**

$$\ell\left(f_{\boldsymbol{\theta}}(\mathbf{x}), y\right) \in \mathbb{R}$$

that measures **deviation** between the prediction $f_{\boldsymbol{\theta}}(\mathbf{x})$ and the "true" solution y

Training Loss 💪

- We can now compute the **average loss** over all training examples:

$$\mathcal{L}_{\text{train}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \ell \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), y^{(i)} \right)$$

- Notice that $\mathcal{L}_{\text{train}} : \mathbb{R}^n \to \mathbb{R}$ ❗

Question 🤔

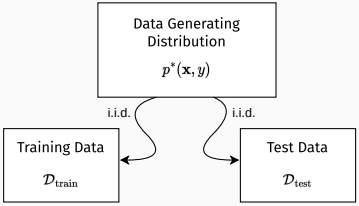
Is the final goal of supervised learning to minimize the average training loss? That is, finding

$$\boldsymbol{\theta}^* = \operatorname*{argmin}_{\boldsymbol{\theta}} \mathcal{L}_{\text{train}}(\boldsymbol{\theta})$$

- **Statistical Learning Theory** can answer this!
- If we had access to $p^*(\mathbf{x}, y)$, we want to minimize the **generalization error**:

$$\mathcal{L}^*(\boldsymbol{\theta}) = \mathbb{E}_{p^*(\mathbf{x},y)}\left[\ell(f_{\boldsymbol{\theta}}(\mathbf{x}), y)\right]$$

- But since we don't know p^*, we minimize $\mathcal{L}_{\text{train}}$
- However, we **truly care about the average test loss** $\mathcal{L}_{\text{test}}$, since this is an unbiased estimate of the generalization error



Minimizing Training Loss

- Let's minimize the training loss

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \ell \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), \mathbf{y}^{(i)} \right)$$

- In *Deep Learning*, there is generally no **closed form solution for this** 😩

- Let's minimize the training loss

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \ell \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), \mathbf{y}^{(i)} \right)$$

- In *Deep Learning*, there is generally no **closed form solution for this** 😩
- Hence, we rely on **iterative optimization algorithms** that use **gradient information**:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{train}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \ell \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), \mathbf{y}^{(i)} \right)$$

- **Is there a problem with this ❓**

- Let's minimize the training loss

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \ell \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), \mathbf{y}^{(i)} \right)$$

- In *Deep Learning*, there is generally no **closed form solution for this** 😩
- Hence, we rely on **iterative optimization algorithms** that use **gradient information**:

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{train}}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\theta}} \ell \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), \mathbf{y}^{(i)} \right)$$

- **Is there a problem with this** ❓
- For large N, this is computionally expensive. Instead, let's approximate this gradient with a **Mini-Batch Gradient** (with $B \ll N$):

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{train}}(\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\mathcal{B}}(\boldsymbol{\theta}) = \frac{1}{B} \sum_{i=1}^{B} \nabla_{\boldsymbol{\theta}} \ell \left(f_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}), \mathbf{y}^{(i)} \right)$$

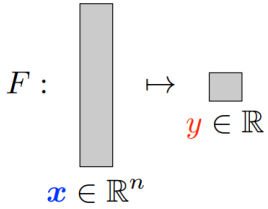
- Thankfully, we don't need to compute the gradients by hand 🙏
- Deep Learning Frameworks offer **Automatic Differentiation** (autodiff) 😍

Automatic Differentiation

Credit: Matt Johnson's great talk on autodiff

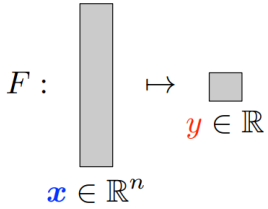
[Johnson, 2017]

$F : \mathbb{R}^n \to \mathbb{R}$



https://www.cs.toronto.edu/~duvenaud/talks/Johnson-Automatic-Differentiation.pdf

$$F : \mathbb{R}^n \to \mathbb{R}$$

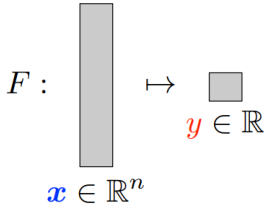


$$F = D \circ C \circ B \circ A$$

$$y = F(\boldsymbol{x}) = D(C(B(A(\boldsymbol{x}))))$$

https://www.cs.toronto.edu/~duvenaud/talks/Johnson-Automatic-Differentiation.pdf

$F : \mathbb{R}^n \to \mathbb{R}$



$$F = D \circ C \circ B \circ A \qquad y = F(\boldsymbol{x}) = D(C(B(A(\boldsymbol{x}))))$$

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

https://www.cs.toronto.edu/~duvenaud/talks/Johnson-Automatic-Differentiation.pdf

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

https://www.cs.toronto.edu/~duvenaud/talks/Johnson-Automatic-Differentiation.pdf

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

https://www.cs.toronto.edu/~duvenaud/talks/Johnson-Automatic-Differentiation.pdf

$$y = D(\boldsymbol{c}), \quad \boldsymbol{c} = C(\boldsymbol{b}), \quad \boldsymbol{b} = B(\boldsymbol{a}), \quad \boldsymbol{a} = A(\boldsymbol{x})$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

$$F'(\boldsymbol{x}) = \frac{\partial y}{\partial \boldsymbol{c}} \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial y}{\partial \boldsymbol{c}} = D'(\boldsymbol{c}) \qquad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} = C'(\boldsymbol{b}) \qquad \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} = B'(\boldsymbol{a}) \qquad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} = A'(\boldsymbol{x})$$

https://www.cs.toronto.edu/~duvenaud/talks/Johnson-Automatic-Differentiation.pdf

$$F'(\boldsymbol{x}) = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}}\left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}}\left(\underbrace{\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}}}\quad\frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}\right)\right)$$

$$\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial b_1}{\partial x_1} & \cdots & \frac{\partial b_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial b_m}{\partial x_1} & \cdots & \frac{\partial b_m}{\partial x_n} \end{bmatrix}$$

Forward accumulation

$$F'(\boldsymbol{x}) = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \left(\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \underbrace{\left(\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \quad \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}} \right)}_{} \right)$$

$$\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial b_1}{\partial x_1} & \cdots & \frac{\partial b_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial b_m}{\partial x_1} & \cdots & \frac{\partial b_m}{\partial x_n} \end{bmatrix}$$

Forward accumulation

$$F'(\boldsymbol{x}) = \left(\underbrace{\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{c}} \quad \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{b}} \right)}_{} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{a}} \right) \frac{\partial \boldsymbol{a}}{\partial \boldsymbol{x}}$$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{b}} = \begin{bmatrix} \frac{\partial y}{\partial b_1} & \cdots & \frac{\partial y}{\partial b_m} \end{bmatrix}$$

Reverse accumulation

Forward vs. Reverse Autodiff (cont.)

Reverse-Mode Autodiff (“Backpropagation”)

- `loss.backward()` does **reverse-mode autodiff** ❗
  - Hence this will **fail** if `loss` is not a scalar
- Implemented using only **Vector-Jacobian products** $\mathbf{v}^\top J$
- Backward pass needs $\approx$ **same amount of compute than forward pass** 😍
- But we need to store **all activations** in the **forward pass** 😓

Forward vs. Reverse Autodiff (cont.)

Reverse-Mode Autodiff ("Backpropagation")

- `loss.backward()` does **reverse-mode autodiff** ❗
  - Hence this will **fail** if `loss` is not a scalar
- Implemented using only **Vector-Jacobian products** $\mathbf{v}^\top J$
- Backward pass needs $\approx$ **same amount of compute than forward pass** 😍
- But we need to store **all activations** in the **forward pass** 😓

Forward-Mode Autodiff

- Constant **memory overhead** 😍
- **Compute inefficient** if we compute gradients of $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$ 😩
- In **typical Deep Learning scenarios**, you rarely care about **forward-mode AD**

Autodiff Demo in PyTorch

References i

Johnson, M. J. (2017).
Automatic differentation.
Presented at the Deep Learning Summer School 2017 in Montreal.