MACHINE LEARNING FOR OPTIMIZATION

January 15, 2025

↳ Crash course on neural nets and automatic differentiation
↳ Introduction to implicit layers

① A crash course on neural networks

Reference: M. Hardt and B. Recht, Patterns, predictions and actions

↳ A neural network (for us): a sequence of layers $z^{(0)}, \dots, z^{(L)}$

Layer ℓ: $\underbrace{z^{(\ell+1)}}_{\text{vector}} = \phi\left(A^{(\ell)} z^{(\ell)} + b^{(\ell)}\right)$

$A^{(\ell)} z^{(\ell)}$: linear transformation of $z^{(\ell)}$

(Typically) Nonlinear activation function

$A^{(\ell)}, b^{(\ell)}$: parameters of the ℓ^{th} layer (to be learned)

Ex) . $\phi: y = [y_i]_i \mapsto [\max(y_i, 0)]_i$ ReLU activation

. $\phi: y = [y_i]_i \mapsto [\tanh(y_i)]_i$ Hyperbolic tangent

Classical layer structures

* Fully connected layer

$\underbrace{x}_{\text{vector}} \longmapsto \underbrace{z}_{\text{vector}}$

$\forall i,\ \underbrace{z_i}_{\in \mathbb{R}} = \phi\left(\sum_j A_{ij} x_j + b_i\right)$

* Convolutional layers

$\underbrace{x}_{\in \mathbb{R}^{d_0 \times d_0 \times c_0}} \longmapsto \underbrace{z}_{\in \mathbb{R}^{d_1 \times d_1 \times c_1}}$

$A \in \mathbb{R}^{d_1 \times d_1 \times c_0 \times c_1}$ $b \in \mathbb{R}^{c_1}$

3rd-order tensor

$\forall (i,j,k),$

$z_{ijk} = \phi\left(\sum_{l,m,n} A_{l,m,n,k}\, x_{i-l,j-m,n} + b_k\right)$

* Recurrent layer

$x \longmapsto z$ $\qquad z = x + B(\phi(Ax))$

A, B : Network parameters

$z^{(0)} \longmapsto z^{(1)} \longmapsto \cdots \longmapsto z^{(L)}$ $\qquad \forall \ell,\ z^{(\ell+1)} = z^{(\ell)} + B(\phi(Az^{(\ell)}))$

Training: Optimization problem

* Data: $\{(x_i, y_i)\}_{i=1..m}$ (ex: supervised learning, want to predict y_i given x_i)

* Neural network: $x \longmapsto NN(x; w)$

w: all the parameters of the network stacked (conceptually) into a big vector in $\mathbb{R}^d$

* Problem (Empirical Risk Minimization)

$$\text{minimize}_{w \in \mathbb{R}^d} \quad \frac{1}{m} \sum_{i=1}^{m} \text{loss}(NN(x_i; w), y_i)$$

y_i: Desired output

$NN(x_i; w)$: output of neural net with input x_i

loss: Loss function (scalar valued)

$F(w)$

Typical loss: $\frac{1}{2} \|NN(x_i, w) - y_i\|^2$ (squared Euclidean norm)

$\forall v \in \mathbb{R}^k,\ \|v\|^2 = \sum_{i=1}^{k} v_i^2$

↳ In practice, this problem is solved using stochastic gradient techniques (SGD, Adam, ...)

$\Rightarrow$ Require $\nabla_w F(w) = \frac{1}{n} \sum_{i=1}^{n} \nabla_w \left(\text{loss}\left(NN(x_i; w), y_i \right) \right)$

$\Rightarrow$ Need an efficient way of computing derivatives, especially $D_w NN(x_i; w)$ $\in \mathbb{R}^{d \times ?}$ → Derivative of the NN mapping with respect to w

- Want a technique that is more efficient than coding the derivative by hand
- Want to be able to compute $D_w NN(x_i; w)$ for multiple x_is ("batch") in parallel

Automatic / Implicit differentiation

- General, abstract idea

$$z^{(0)} \mapsto z^{(1)} \mapsto \dots \mapsto z^{(L)}$$

$$z^{(1)} = f_1(z^{(0)}), \quad z^{(2)} = f_2(z^{(1)}), \dots$$

$z^{(l)} \in \mathbb{R}^{d_l}$, $f_l : \mathbb{R}^{d_{l-1}} \mapsto \mathbb{R}^{d_l}$

Jacobian $D_{z^{(l-1)}} f_l(z) \in \mathbb{R}^{d_{l-1} \times d_l}$ Matrix of partial derivatives

Goal: Compute $D_{z^{(0)}} z^{(L)}$

Backpropagation

① Forward pass: Evaluate $z^{(L)}$ and build a computational graph

Input: $x \in \mathbb{R}^{d_0}$ → Numerical value vs $z^{(\ell)}$ function

Set $v_0 = x$

For $\ell = 1..L$

  Store $v_{\ell-1}$ and compute $v_\ell = f_\ell(v_{\ell-1})$

Output v_L

② Backward pass: Go through the computational graph backwards

Input: $(v_0, \dots, v_L)$

Set $\Lambda_L = D_{z^{(L)}} z^{(L)}(v_L) = I_{d_L}$

For $\ell = L, \dots, 1$

  $\Lambda_{\ell-1} = \Lambda_\ell D_{z^{(\ell-1)}} z^{(\ell)}(v_{\ell-1})$

Output Λ_0

↓ Computed at layer ℓ doesn't require the knowledge of the other layers

Can prove that $\Lambda_0 = D_{z^{(0)}} z^{(L)}(x)$ and $\Lambda_\ell = D_{z^{(0)}} z^{(\ell)}(x)$ $\forall \ell$

Intuition (Frequently written this way in ML papers)

$$\frac{\partial z^{(l-1)}}{\partial z^{(0)}} \times \frac{\partial z^{(l)}}{\partial z^{(l-1)}} = \frac{\partial z^{(l)}}{\partial z^{(0)}}$$

"Chain rule"

$$D_{z^{(0)}} z^{(l-1)}(x) = D_{z^{(0)}} z^{(l)}(x) \; D_{z^{(l-1)}} z^{(l)}(x)$$

- The technique above can be used to compute $D_x NN(x;w)$ for any neural network with standard layers and activation functions (even ReLU, which doesn't have a derivative everywhere)

$\Rightarrow$ the same approach (and the same computational graph structure) can be used to compute $D_w NN(x;w)$

Ex) $NN(x;w) = A_L \, \mathrm{ReLU}(A_{L-1} \, \mathrm{ReLU}(\cdots(A_1 x)\cdots))$

($A_L \in \mathbb{R}^{d_L \times d_{L-1}}$, $A_{L-1} \in \mathbb{R}^{d_{L-1} \times d_{L-2}}$, $A_1 \in \mathbb{R}^{d_1 \times d_0}$, $x \in \mathbb{R}^{d_0}$)

w concatenates all A_l matrices

$w \in \mathbb{R}^d$, $d = d_L d_{L-1} + d_{L-1} d_{L-2} + \cdots + d_1 d_0$

$x \mapsto \underbrace{\mathrm{ReLU}(A_1 x)}_{z^{(1)}} \mapsto \underbrace{\mathrm{ReLU}(A_2 z^{(1)})}_{z^{(2)}} \mapsto \cdots$

$$D_x NN(x;w) = A_L \Delta_{L-1} A_{L-1} \Delta_{L-2} \cdots \Delta_1 A_1$$

where $\Delta_l = \mathrm{diag}(z^{(l)})$

For any l, can compute the derivative of $NN(x;w)$

with respect to any column of A_ℓ by using

$$u = \mathrm{ReLU}(A_{\ell-1}(\dots A_1 x \dots)) \quad \Big] \text{ input of layer } \ell$$

$$D_{A_\ell u} f_\ell(A_\ell u) = A_L A_{L-1} \cdots A_\ell$$

$$f_\ell(z) = A_L \mathrm{ReLU}(\dots A_{L+1} \mathrm{ReLU}(z) \dots)$$

$\forall j,$

$$D_{[A_\ell]_{:,j}} NN(x, w) = D_{A_\ell u} f_\ell(A_\ell u) \times D_{[A_\ell]_{:,j}} A_\ell u$$

$[A_\ell]_{:,j}$ ← j^{th} column of A_ℓ → $[A_\ell]_{:,j}$

$A_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$

$\cdot = u_j I$

↳ Python libraries for automatic differentiation

Autograd (within PyTorch)

JAX

Takeaway:

- No need to compute derivatives by yourself, AD can do it...
- ... especially for neural networks with standard, explicitly defined layers
  - ↳ Fully connected / Recurrent / Convolutional all look like $x \longmapsto$ [compute $z = f(x)$] $\longrightarrow z$

Q] What if the layer is now defined implicitly,

e.g. $x \longmapsto \underset{z}{\text{argmin}} \; f(z; x)$ s.t. $z \in C(x)$

(2) Implicit layers

Explicit layer: $x \longmapsto$ [Compute $f(x)$] $\longrightarrow z = f(x)$

Implicit layer: $x \longrightarrow$ [Find z such that $C(x,z) = 0$] $\longrightarrow z$

Joint condition on the input and the output

→ The implicit layer paradigm separates what the layer should do from the implementation of the layer

$C(x,z) = 0$ can be

- An algebraic equation ⇓ Deep Equilibrium Models
- A differential equation → Neural ODEs / PINNs
- Optimization problem / Optimality conditions of an optimization → Differentiable optimization