A  into $\frac{d}{dn}\left[\begin{array}{l} d\mathrm{eep} \\ d\mathrm{eclarative} \\ n\mathrm{etworks} \end{array}\right]$

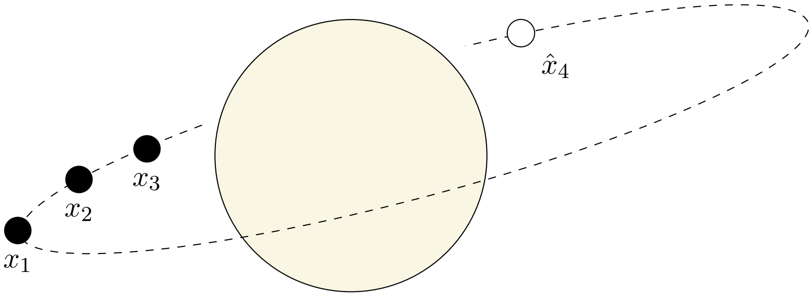
Stephen Gould

stephen.gould@anu.edu.au

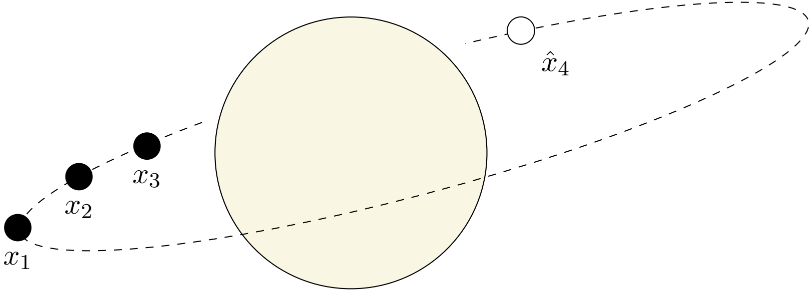
Robotic Vision Summer School (RVSS), 2024
Australian National University

9 February 2024

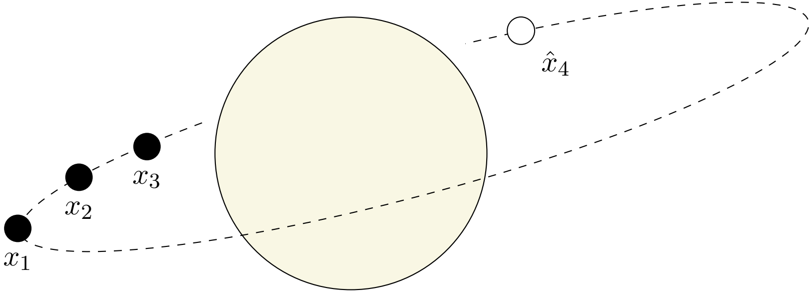
Discovery of Ceres



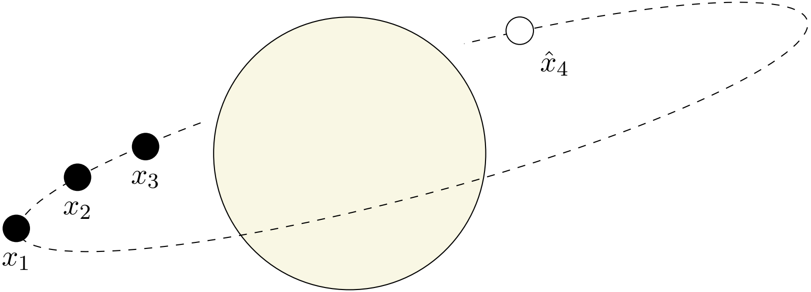
Discovery of Ceres



Discovery of Ceres



Discovery of Ceres



Optimisation is Everywhere

- **financial mathematics:** maximise profits or minimise costs subject to constraints on resources and budgets

Optimisation is Everywhere

- **financial mathematics:** maximise profits or minimise costs subject to constraints on resources and budgets
- **mechanical engineering:** maximise the span of a bridge subject to load constraints; minimise drag on a vehicle subject to volume constraints

Optimisation is Everywhere

- **financial mathematics:** maximise profits or minimise costs subject to constraints on resources and budgets
- **mechanical engineering:** maximise the span of a bridge subject to load constraints; minimise drag on a vehicle subject to volume constraints
- **electrical engineering:** minimise the size of a transistor in a circuit subject to power and timing constraints

Optimisation is Everywhere

- **financial mathematics:** maximise profits or minimise costs subject to constraints on resources and budgets
- **mechanical engineering:** maximise the span of a bridge subject to load constraints; minimise drag on a vehicle subject to volume constraints
- **electrical engineering:** minimise the size of a transistor in a circuit subject to power and timing constraints
- **logistics and planning:** find the cheapest way to distribute goods from suppliers to consumers across a transportation network

Optimisation is Everywhere

- **financial mathematics:** maximise profits or minimise costs subject to constraints on resources and budgets
- **mechanical engineering:** maximise the span of a bridge subject to load constraints; minimise drag on a vehicle subject to volume constraints
- **electrical engineering:** minimise the size of a transistor in a circuit subject to power and timing constraints
- **logistics and planning:** find the cheapest way to distribute goods from suppliers to consumers across a transportation network
- **statistics/data science:** curve fitting and data visualisation

Optimisation is Everywhere

- **financial mathematics:** maximise profits or minimise costs subject to constraints on resources and budgets
- **mechanical engineering:** maximise the span of a bridge subject to load constraints; minimise drag on a vehicle subject to volume constraints
- **electrical engineering:** minimise the size of a transistor in a circuit subject to power and timing constraints
- **logistics and planning:** find the cheapest way to distribute goods from suppliers to consumers across a transportation network
- **statistics/data science:** curve fitting and data visualisation
- **robotics:** optimise control parameters to achieve some goal state or trajectory; simultaneous localisation and mapping (SLAM); point/feature matching

Optimisation is Everywhere

- **financial mathematics:** maximise profits or minimise costs subject to constraints on resources and budgets
- **mechanical engineering:** maximise the span of a bridge subject to load constraints; minimise drag on a vehicle subject to volume constraints
- **electrical engineering:** minimise the size of a transistor in a circuit subject to power and timing constraints
- **logistics and planning:** find the cheapest way to distribute goods from suppliers to consumers across a transportation network
- **statistics/data science:** curve fitting and data visualisation
- **robotics:** optimise control parameters to achieve some goal state or trajectory; simultaneous localisation and mapping (SLAM); point/feature matching
- **machine learning and deep learning:** minimise loss functions with respect to the parameters of our model

Optimisation Problems

find an assignment to variables that minimises
a measure of cost subject to some constraints[1]

[1] In these lectures we will be concerned with continuous-valued variables

Optimisation Problems

$$\begin{array}{ll} \text{minimize (over } x\text{)} & \text{objective}(x) \\ \text{subject to} & \text{constraints}(x) \end{array}$$

Optimisation Problems

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, p \\ & h_i(x) = 0, \quad i = 1, \ldots, q \end{array}$$

- $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ — optimisation variables
- $f_0 : \mathbb{R}^n \to \mathbb{R}$ — objective (or cost or loss) function
- $f_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, \ldots, p$ — inequality constraint functions
- $h_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, \ldots, q$ — equality constraint functions

Least Squares

$$\text{minimize} \quad \|Ax - b\|_2^2$$

Least Squares

$$\text{minimize} \quad \|Ax - b\|_2^2$$

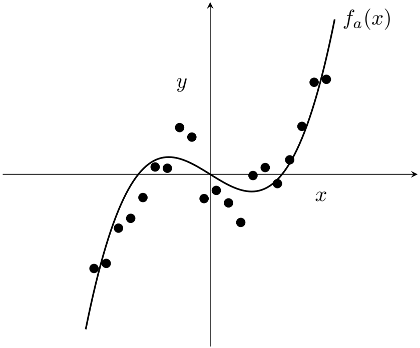
- unique solution if $A^T A$ is invertible, $x^\star = \left(A^T A\right)^{-1} A^T b$
- solution via SVD, $A = U\Sigma V^T$, if $A^T A$ not invertible, $x^\star = V\Sigma^{-1}U^T b$
  - in fact, $x^\star + w$ for any $w \in \mathcal{N}(A)$ also a solution
- solution via QR factorisation, $x^\star = R^{-1}Q^T b$
- solved in $O(n^2 m)$ time, less if structured
- typically use iterative solver (for large scale problems)

Example: Polynomial Curve Fitting

fit n-th order polynomial $f_a(x) = \sum_{k=0}^{n} a_k x^k$ to set of noisy points $\{(x_i, y_i)\}_{i=1}^{m}$
(here a are the variables, and x and y are the data)

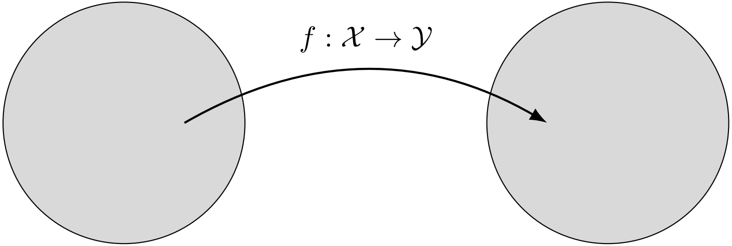
minimize (over a) $\quad \sum_{i=1}^{m} \left(f_a(x_i) - y_i\right)^2$

minimize $\quad \left\| \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} \right\|_2^2$

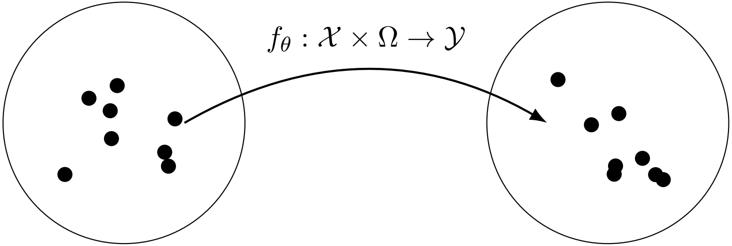


Part I. Machine Learning and Deep Learning

Machine Learning from 10,000ft



Machine Learning from 10,000ft

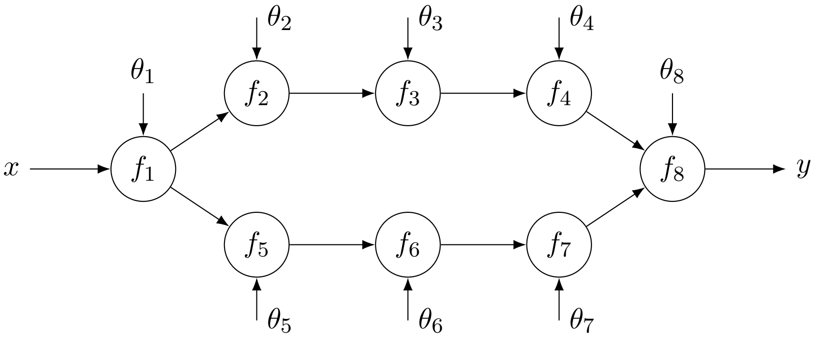


minimize (over θ) $\sum_{(x,y)\sim\mathcal{X}\times\mathcal{Y}} L(f_\theta(x), y)$

- loss L — what to do
- model f_θ — how to do it
- optimised by gradient descent (or variant thereof)

Deep Learning as an End-to-end Computation Graph

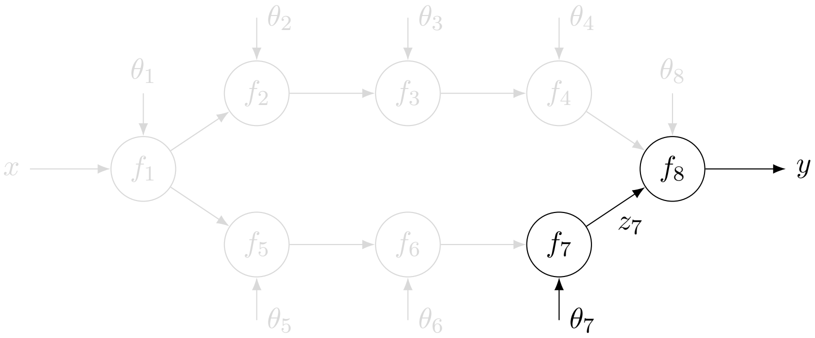
Deep learning does this by constructing the model f_θ (equiv. computation graph) as the composition of many simple parametrized functions (equiv. computation nodes).



$$y = f_8(f_4(f_3(f_2(f_1(x)))), f_7(f_6(f_5(f_1(x)))))$$

(parameters θ_i omitted for brevity)

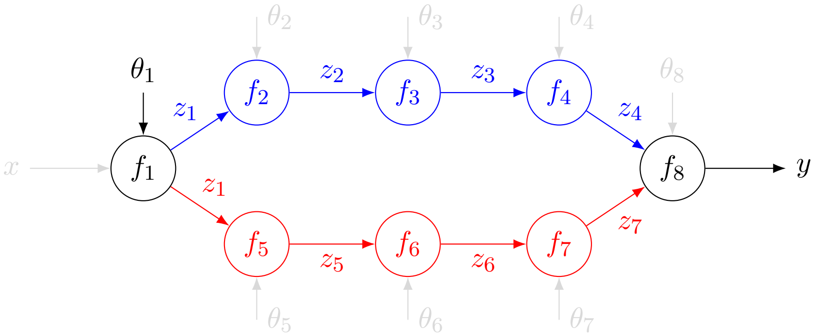
Backward Pass Gradient Calculation



Example 1.

$$\frac{\partial L}{\partial \theta_7} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z_7}\frac{\partial z_7}{\partial \theta_7}$$

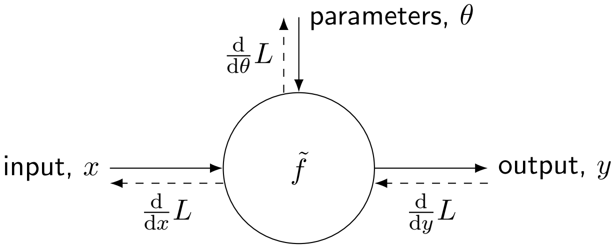
Backward Pass Gradient Calculation



Example 2.

$$\frac{\partial L}{\partial \theta_1} = \frac{\partial L}{\partial y} \left(\frac{\partial y}{\partial z_4} \frac{\partial z_4}{\partial z_3} \frac{\partial z_3}{\partial z_2} \frac{\partial z_2}{\partial z_1} + \frac{\partial y}{\partial z_7} \frac{\partial z_7}{\partial z_6} \frac{\partial z_6}{\partial z_5} \frac{\partial z_5}{\partial z_4} \right) \frac{\partial z_1}{\partial \theta_1}$$

Deep Learning Node



- **Forward pass:** compute output y as a function of the input x (and model parameters θ).
- **Backward pass:** compute the derivative of the loss with respect to the input x (and model parameters θ) given the derivative of the loss with respect to the output y.

Aside: Notation (Often Sloppy)

For scalar-valued functions:

total derivative: $\dfrac{\mathrm{d}f}{\mathrm{d}x}$ partial derivative: $\dfrac{\partial f}{\partial x}$

Aside: Notation (Often Sloppy)

For scalar-valued functions:

total derivative: $\dfrac{\mathrm{d}f}{\mathrm{d}x}$ partial derivative: $\dfrac{\partial f}{\partial x}$

For multi-dimensional vector-valued functions, $f : \mathbb{R}^n \to \mathbb{R}^m$:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_1}{\mathrm{d}x_n} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}f_m}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_m}{\mathrm{d}x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

($\dfrac{\partial}{\partial x} f(x, y)$ for partial)

Sometimes D and D_X for $\frac{\mathrm{d}}{\mathrm{d}x}$ and $\frac{\partial}{\partial x}$, respectively.

Aside: Notation (Often Sloppy)

For scalar-valued functions:

total derivative: $\dfrac{\mathrm{d}f}{\mathrm{d}x}$ partial derivative: $\dfrac{\partial f}{\partial x}$

For multi-dimensional vector-valued functions, $f : \mathbb{R}^n \to \mathbb{R}^m$:

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \begin{bmatrix} \frac{\mathrm{d}f_1}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_1}{\mathrm{d}x_n} \\ \vdots & \ddots & \vdots \\ \frac{\mathrm{d}f_m}{\mathrm{d}x_1} & \cdots & \frac{\mathrm{d}f_m}{\mathrm{d}x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

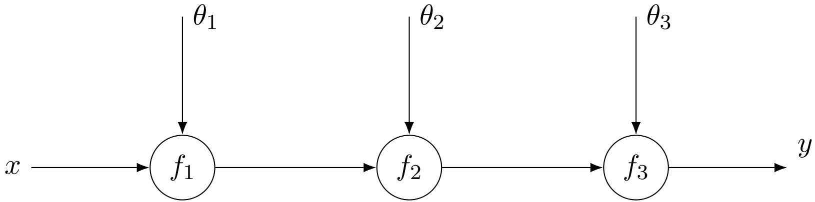
($\dfrac{\partial}{\partial x} f(x, y)$ for partial)

Sometimes D and D_X for $\frac{\mathrm{d}}{\mathrm{d}x}$ and $\frac{\partial}{\partial x}$, respectively.

Mathematically, derivatives with respect to (scalar-valued) loss functions are row vectors ($m = 1$).

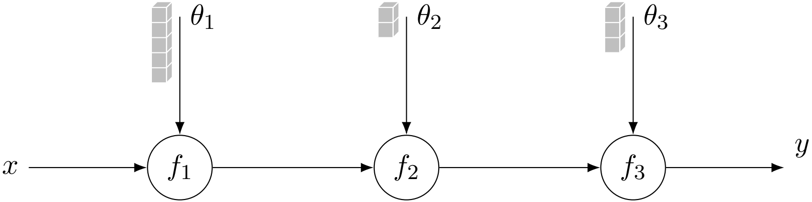
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



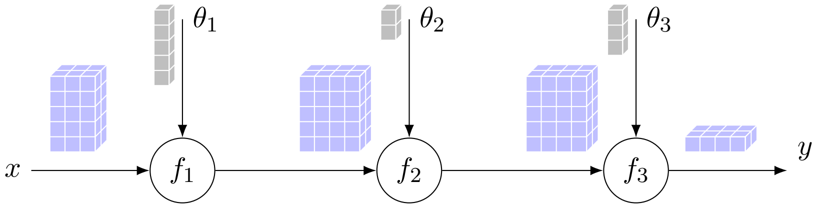
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



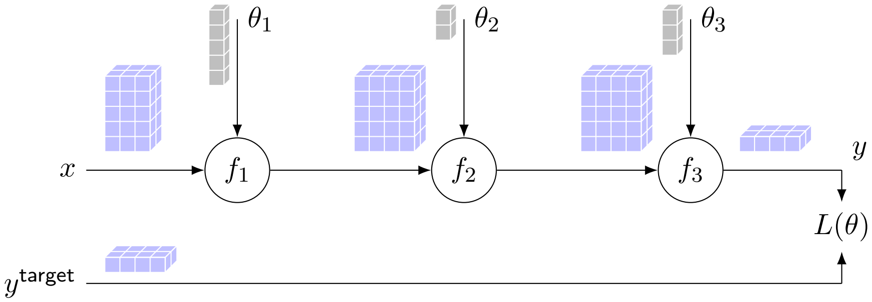
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



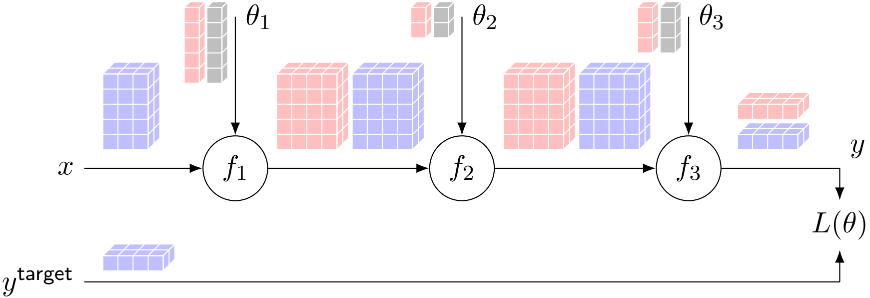
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



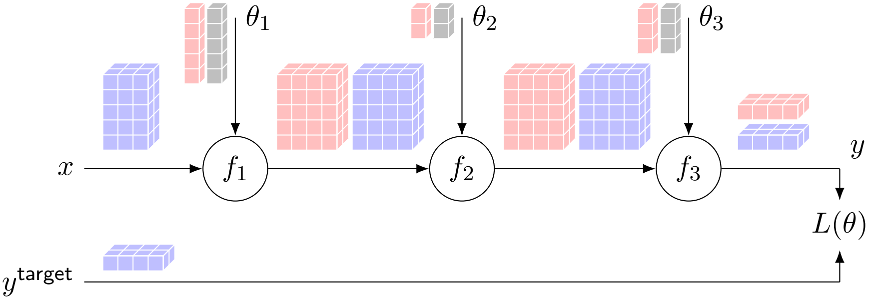
Concerning Memory

- data is often processed in batches ($B \times N \times \cdots \times C$)



Concerning Memory

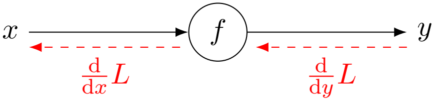
- data is often processed in batches ($B \times N \times \cdots \times C$)



- parameters (usually) only take a small amount of memory (relative to data)
- derivatives take the same amount of space as the data **and stored transposed!**
- in-place operations may save memory in the forward pass
- re-using buffers may save memory in the backward pass
- at test time intermediate results are not stored

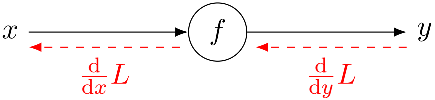
Quick Quiz

Quick Quiz



$$y = Ax$$

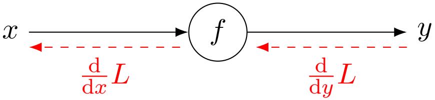
Quick Quiz



$$y = Ax$$

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}x} &= \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} \\ &= \frac{\mathrm{d}L}{\mathrm{d}y}A\end{aligned}$$

Quick Quiz

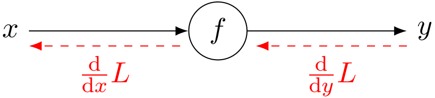


$$y = Ax$$

$$\begin{aligned} \frac{\mathrm{d}L}{\mathrm{d}x} &= \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} \\ &= \frac{\mathrm{d}L}{\mathrm{d}y}A \end{aligned}$$

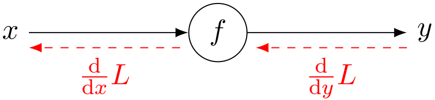
- forward pass $O(n^2)$, less if A is structured
- backward pass costs same as forward pass

Quick Quiz (2)



$$Ay = x$$

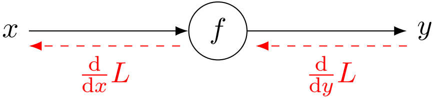
Quick Quiz (2)



$$
\begin{aligned}
Ay &= x \\
\therefore \; y &= A^{-1}x
\end{aligned}
$$

$$
\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}x} &= \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} \\
&= \frac{\mathrm{d}L}{\mathrm{d}y}A^{-1}
\end{aligned}
$$

Quick Quiz (2)



$$
\begin{aligned}
Ay &= x \\
\therefore \; y &= A^{-1}x
\end{aligned}
$$

$$
\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}x} &= \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} \\
&= \frac{\mathrm{d}L}{\mathrm{d}y}A^{-1}
\end{aligned}
$$

- forward pass $O(n^3)$, less if structured
- backward pass solves $w = A^T v$
  - **cheaper** than forward pass if decomposition of A is cached

Automatic Differentiation (AD)

- algorithmic procedure that produces code for computing **exact** derivatives
- assumes numeric computations are composed of a small set of elementary operations that we know how to differentiate
  - arithmetic, exp, log, trigonometric
- workhorse of modern machine learning that greatly reduces development effort
- roughly speaking, for each line of the forward pass code, `P, Q = foo(A, B, C)`, autodiff produces a line `dLdA, dLdB, dLdC = foo_vjp(dLdP, dLdQ)` in the backward pass code

Automatic Differentiation (AD)

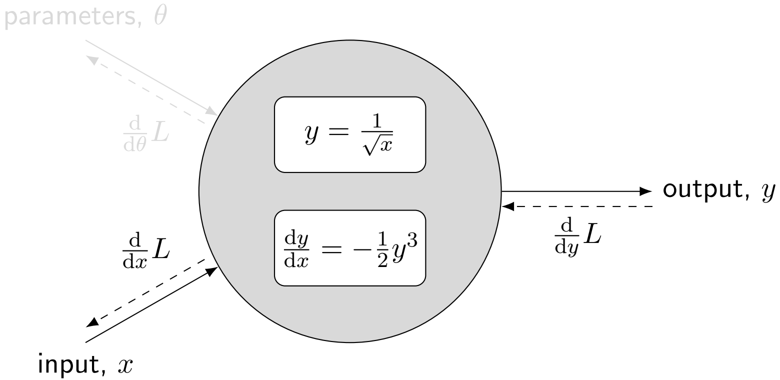
- algorithmic procedure that produces code for computing **exact** derivatives
- assumes numeric computations are composed of a small set of elementary operations that we know how to differentiate
  - arithmetic, exp, log, trigonometric
- workhorse of modern machine learning that greatly reduces development effort
- roughly speaking, for each line of the forward pass code, `P, Q = foo(A, B, C)`, autodiff produces a line `dLdA, dLdB, dLdC = foo_vjp(dLdP, dLdQ)` in the backward pass code
- but it doesn't always work (see point 2), and when it does work it can be slow and/or memory intensive

example

Computing $1/\sqrt{x}$

```c
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = * ( long * ) &y;                  // evil floating point bit level hacking
    i = 0x5f3759df - ( i >> 1 );          // what the f**k?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );      // 1st iter
    //  y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iter, can be removed

    return y;
}
```

Separate Forward and Backward Operations



Part II. Differentiable Optimisation

Bi-level Optimisation: Stackelberg Games

Consider two players, a **leader** and a **follower**

- the market dictates the price it's willing to pay for some goods based on supply, i.e., quantity produced by both players, $P(q_1 + q_2)$

Bi-level Optimisation: Stackelberg Games

Consider two players, a **leader** and a **follower**

- the market dictates the price it's willing to pay for some goods based on supply, i.e., quantity produced by both players, $P(q_1 + q_2)$
- each player has a cost structure associated with producing goods, $C_i(q_i)$ and wants to maximize profits, $q_i P(q_1 + q_2) - C_i(q_i)$

Bi-level Optimisation: Stackelberg Games

Consider two players, a **leader** and a **follower**

- the market dictates the price it's willing to pay for some goods based on supply, i.e., quantity produced by both players, $P(q_1 + q_2)$
- each player has a cost structure associated with producing goods, $C_i(q_i)$ and wants to maximize profits, $q_i P(q_1 + q_2) - C_i(q_i)$
- the leader picks a quantity of goods to produce knowing that the follower will respond optimally. In other words, the leader solves

$$
\begin{array}{ll}
\text{maximize (over } q_1\text{)} & q_1 P(q_1 + q_2) - C_1(q_1) \\
\text{subject to} & q_2 \in \operatorname{argmax}_q \, q P(q_1 + q) - C_2(q)
\end{array}
$$

Bi-level Optimisation Problems in Machine Learning

- **quantities:** input x, output y, parameters θ

$$\begin{array}{ll} \text{minimize (over } \theta) & L(x, y; \theta) \\ \text{subject to} & y \in \operatorname{argmin}_{u \in C(x;\theta)} f(x, u; \theta) \end{array}$$

  - lower-level is an optimisation problem parametrized by x and θ

Bi-level Optimisation Problems in Machine Learning

- **quantities:** input x, output y, parameters θ

$$\begin{array}{ll} \text{minimize (over } \theta) & L(x, y; \theta) \\ \text{subject to} & y \in \operatorname{argmin}_{u \in C(x;\theta)} f(x, u; \theta) \end{array}$$

  - lower-level is an optimisation problem parametrized by x and θ

- **gradient descent:** compute gradient of lower-level solution y with respect to θ, and use the chain rule to get the total derivative,

$$\theta \leftarrow \theta - \eta \left(\frac{\partial L}{\partial \theta} + \frac{\partial L}{\partial y} \frac{\mathrm{d}y}{\mathrm{d}\theta} \right)$$

  - by back-propagating *through* the optimisation problem

Differentiable Least Squares

Consider our old friend, the least-squares problem,

$$\text{minimize} \quad \|Ax - b\|_2^2$$

parameterized by A and b and with closed-form solution $x^\star = \left(A^T A\right)^{-1} A^T b$.

Differentiable Least Squares

Consider our old friend, the least-squares problem,

$$\text{minimize} \quad \|Ax - b\|_2^2$$

parameterized by A and b and with closed-form solution $x^\star = \left(A^T A\right)^{-1} A^T b$.

We are interested in derivatives of the solution with respect to the elements of A,

$$\frac{\mathrm{d}x^\star}{\mathrm{d}A_{ij}} = \frac{\mathrm{d}}{\mathrm{d}A_{ij}} \left(A^T A\right)^{-1} A^T b \quad \in \mathbb{R}^n$$

We could also compute derivatives with respect to elements of b (but not here).

Least Squares Backward Pass

The backward pass combines $\frac{\mathrm{d}x^\star}{\mathrm{d}A_{ij}}$ with $v^T = \frac{\mathrm{d}L}{\mathrm{d}x^\star}$ via the vector-Jacobian product. After some algebraic manipulation we get

$$\left(\frac{\mathrm{d}L}{\mathrm{d}A}\right)^T = wr^T - x^\star(Aw)^T \quad \in \mathbb{R}^{m \times n}$$

where $w^T = v^T(A^TA)^{-1}$ and $r = b - Ax^\star$.

Least Squares Backward Pass

The backward pass combines $\frac{\mathrm{d}x^\star}{\mathrm{d}A_{ij}}$ with $v^T = \frac{\mathrm{d}L}{\mathrm{d}x^\star}$ via the vector-Jacobian product. After some algebraic manipulation we get

$$\left(\frac{\mathrm{d}L}{\mathrm{d}A}\right)^T = wr^T - x^\star(Aw)^T \quad \in \mathbb{R}^{m \times n}$$

where $w^T = v^T(A^TA)^{-1}$ and $r = b - Ax^\star$.

- $\left(A^TA\right)^{-1}$ is used in both the forward and backward pass
- factored once to solve for x, e.g., into $A = QR$
- cache R and re-use when computing gradients

derivation

PyTorch Implementation: Forward Pass

```python
class LeastSquaresFcn(torch.autograd.Function):
    """PyTorch autograd function for least squares."""

    @staticmethod
    def forward(ctx, A, b):
        B, M, N = A.shape
        assert b.shape == (B, M, 1)

        with torch.no_grad():
            Q, R = torch.linalg.qr(A, mode='reduced')
            x = torch.linalg.solve_triangular(R,
                torch.bmm(b.view(B, 1, M), Q).view(B, N, 1), upper=True)

        # save state for backward pass
        ctx.save_for_backward(A, b, x, R)

        # return solution
        return x
```

$$A = QR$$

$$x = R^{-1}\left(Q^T b\right)$$

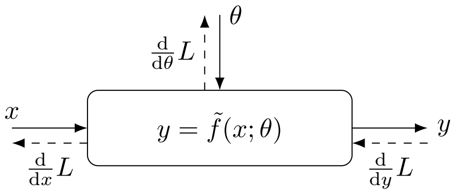
(solves $Rx = Q^T b$)

PyTorch Implementation: Backward Pass

```
    @staticmethod
    def backward(ctx, dx):
        # check for None tensors
        if dx is None:
            return None, None

        # unpack cached tensors
        A, b, x, R = ctx.saved_tensors
        B, M, N = A.shape

        dA, db = None, None

        w = torch.linalg.solve_triangular(R,
            torch.linalg.solve_triangular(torch.transpose(R, 2, 1),
            dx, upper=False), upper=True)
        Aw = torch.bmm(A, w)

        if ctx.needs_input_grad[0]:
            r = b - torch.bmm(A, x)
            dA = torch.bmm(r.view(B,M,1), w.view(B,1,N)) - \
                torch.bmm(Aw.view(B,M,1), x.view(B,1,N))
        if ctx.needs_input_grad[1]:
            db = Aw

        # return gradients
        return dA, db
```

$$
\begin{aligned}
w &= \left(A^T A\right)^{-1} v \\
&= R^{-1}\left(R^{-T} v\right) \\
r &= b - Ax \\
\left(\frac{\mathrm{d}L}{\mathrm{d}A}\right)^T &= r w^T - (Aw) x^T \\
\left(\frac{\mathrm{d}L}{\mathrm{d}b}\right)^T &= Aw
\end{aligned}
$$

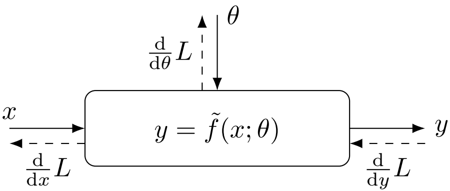
Imperative vs Declarative Nodes

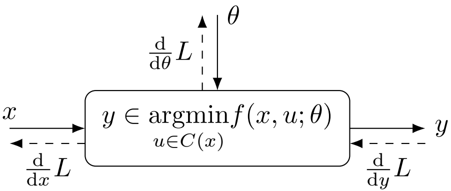


- imperative node
- input-output relationship explicit,

$$y = \tilde{f}(x;\theta)$$

Imperative vs Declarative Nodes





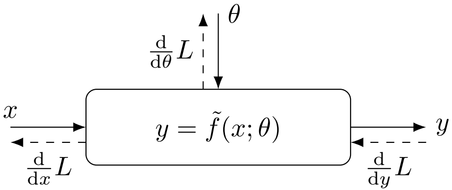
- imperative node
- input-output relationship explicit,

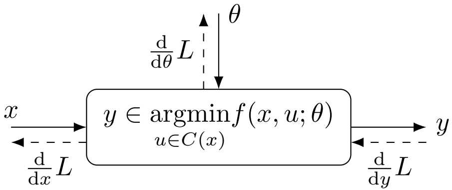
$$y = \tilde{f}(x; \theta)$$

- declarative node
- input-output relationship specified as solution to an optimisation problem,

$$y \in \arg\min_{u \in C(x)} f(x, u; \theta)$$

Imperative vs Declarative Nodes





- imperative node
- input-output relationship explicit,

$$y = \tilde{f}(x;\theta)$$

- declarative node
- input-output relationship specified as solution to an optimisation problem,

$$y \in \operatorname*{arg\,min}_{u \in C(x)} f(x, u; \theta)$$

can co-exist in the same computation graph (network)

Average Pooling Example

$$\{x_i \in \mathbb{R}^m \mid i = 1, \ldots, n\} \to \mathbb{R}^m$$

- imperative specification

$$y = \frac{1}{n} \sum_{i=1}^{n} x_i$$

- declarative specification

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \|u - x_i\|^2$$

Average Pooling Example

$$\{x_i \in \mathbb{R}^m \mid i = 1, \ldots, n\} \to \mathbb{R}^m$$

- imperative specification

$$y = \frac{1}{n} \sum_{i=1}^{n} x_i$$

- declarative specification

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \|u - x_i\|^2$$

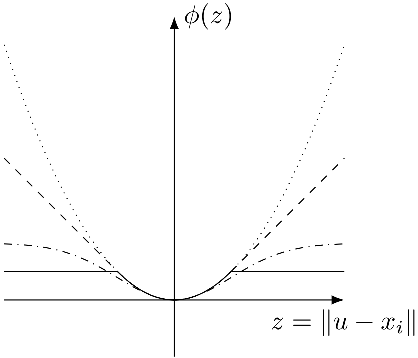
- can be easily varied, e.g., made robust

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \phi(u - x_i)$$

for some penalty function ϕ

Average Pooling Example

$$\{x_i \in \mathbb{R}^m \mid i = 1, \ldots, n\} \to \mathbb{R}^m$$



- declarative specification

$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \|u - x_i\|^2$$

- can be easily varied, e.g., made robust

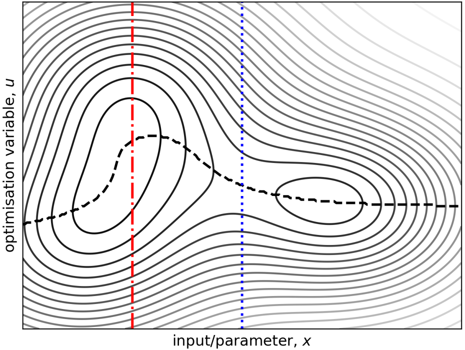
$$y = \text{argmin}_{u \in \mathbb{R}^m} \sum_{i=1}^{n} \phi(u - x_i)$$

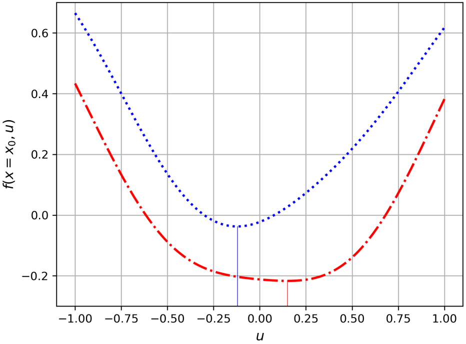
for some penalty function ϕ

Parametrized Optimisation Re-cap

Think of y and an implicit function of x (wlog we'll ignore θ from here on),

$$y(x) = \text{argmin}_{u \in C(x)} f(x, u)$$

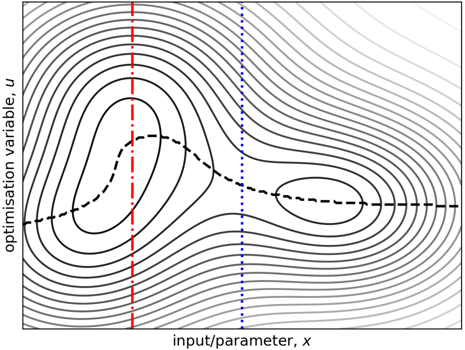


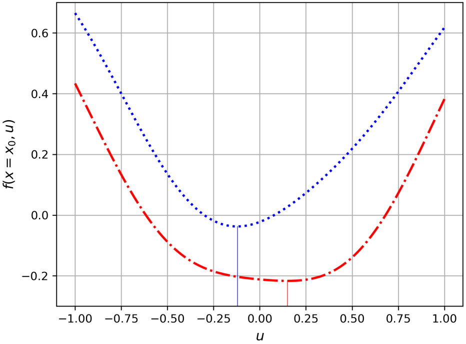


Parametrized Optimisation Re-cap

Think of y and an implicit function of x (wlog we'll ignore θ from here on),

$$y(x) = \text{argmin}_{u \in C(x)} f(x, u)$$





Main question: How do we compute $\frac{\mathrm{d}}{\mathrm{d}x} \text{argmin}_{u \in C(x)} f(x, u)$?

Computing $\frac{\mathrm{d}}{\mathrm{d}x}\text{argmin}_{u \in C(x)} f(x, u)$

- explicit from closed-form solution
  - e.g., least-squares

Computing $\frac{\mathrm{d}}{\mathrm{d}x} \operatorname{argmin}_{u \in C(x)} f(x, u)$

- explicit from closed-form solution
  - e.g., least-squares
- automatic differentiation of forward pass code
  - e.g., unrolling gradient descent (next)

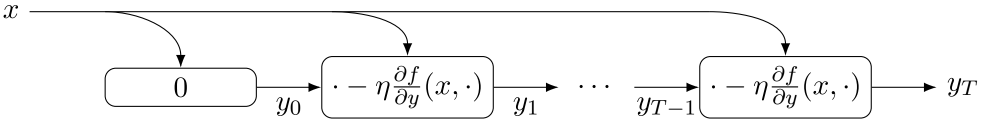
Computing $\frac{\mathrm{d}}{\mathrm{d}x} \operatorname{argmin}_{u \in C(x)} f(x, u)$

- explicit from closed-form solution
  - e.g., least-squares
- automatic differentiation of forward pass code
  - e.g., unrolling gradient descent (next)
- implicit differentiation of optimality conditions (later)
  - allows non-differentiable steps in the forward pass
  - no need to store intermediate calculations

Unrolling Gradient Descent

repeat until convergence:

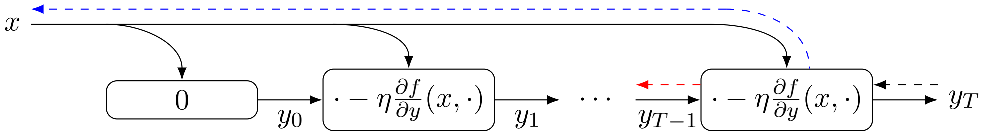
$$y_t \leftarrow y_{t-1} - \eta \frac{\partial f}{\partial y}(x, y_{t-1})$$



Unrolling Gradient Descent

repeat until convergence:

$$y_t \leftarrow y_{t-1} - \eta \frac{\partial f}{\partial y}(x, y_{t-1})$$



$$\frac{\mathrm{d}y_t}{\mathrm{d}x} = \frac{\partial y_t}{\partial x} + \frac{\partial y_t}{\partial y_{t-1}} \frac{\mathrm{d}y_{t-1}}{\mathrm{d}x} = -\eta \frac{\partial^2 f}{\partial x \partial y}(x, y_{t-1}) + \left(I - \eta \frac{\partial^2 f}{\partial y^2}(x, y_{t-1})\right) \frac{\mathrm{d}y_{t-1}}{\mathrm{d}x}$$

Dini's Implicit Function Theorem

Consider the solution mapping associated with the equation $f(x, u) = 0$,

$$Y : x \mapsto \{u \in \mathbb{R}^m \mid f(x, u) = 0\} \text{ for } x \in \mathbb{R}^n.$$

We are interested in how elements of $Y(x)$ change as a function of x.

Dini's Implicit Function Theorem

Consider the solution mapping associated with the equation $f(x, u) = 0$,

$$Y : x \mapsto \{u \in \mathbb{R}^m \mid f(x, u) = 0\} \text{ for } x \in \mathbb{R}^n.$$

We are interested in how elements of $Y(x)$ change as a function of x.

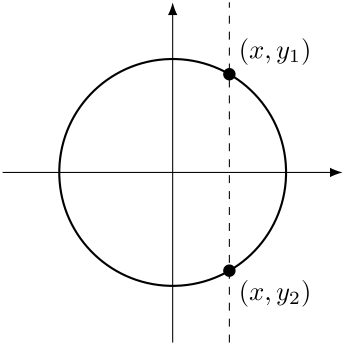
Theorem

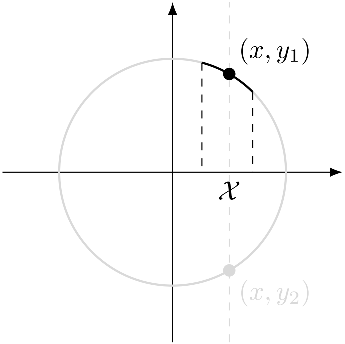
Let $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^m$ be differentiable in a neighbourhood of (x, u) and such that $f(x, u) = 0$, and let $\frac{\partial}{\partial u} f(x, u)$ be nonsingular. Then the solution mapping Y has a single-valued localization y around x for u which is differentiable in a neighbourhood $\mathcal{X}$ of x with Jacobian satisfying

$$\frac{dy(x)}{dx} = -\left(\frac{\partial f(x, y(x))}{\partial y}\right)^{-1} \frac{\partial f(x, y(x))}{\partial x}$$

for every $x \in \mathcal{X}$.

Unit Circle Example





$$y = \pm\sqrt{1 - x^2}$$
$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mp 2x}{2\sqrt{1 - x^2}} = -\frac{x}{y}$$

$$f(x, y) = x^2 + y^2 - 1$$
$$\begin{aligned}\frac{\mathrm{d}y}{\mathrm{d}x} &= -\left(\frac{\partial f}{\partial y}\right)^{-1}\left(\frac{\partial f}{\partial x}\right) \\ &= -\left(\frac{1}{2y}\right)(2x) = -\frac{x}{y}\end{aligned}$$

Differentiating Unconstrained Optimisation Problems

Let $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be twice differentiable and let

$$y(x) \in \text{argmin}_u f(x, u)$$

then for non-zero Hessian

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = -\left(\frac{\partial^2 f}{\partial y^2}\right)^{-1} \frac{\partial^2 f}{\partial x \partial y}.$$

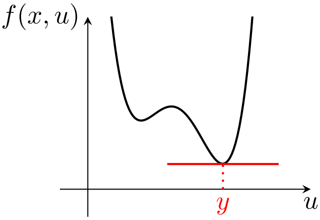
Differentiating Unconstrained Optimisation Problems

Let $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be twice differentiable and let

$$y(x) \in \text{argmin}_u f(x, u)$$

then for non-zero Hessian

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = -\left(\frac{\partial^2 f}{\partial y^2}\right)^{-1} \frac{\partial^2 f}{\partial x \partial y}.$$



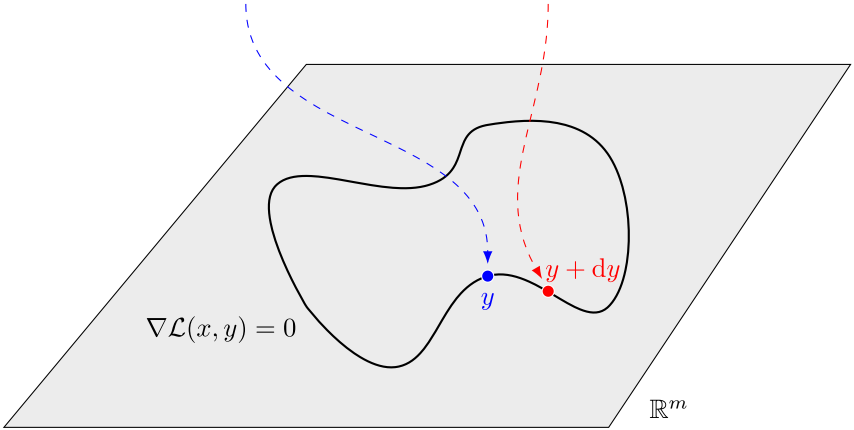
Proof. The derivative of f vanishes at (x, y), i.e., $y \in \text{argmin}_u f(x, u) \implies \frac{\partial f(x,y)}{\partial y} = 0$.

$$\begin{aligned}
\text{LHS :} \quad & \frac{\mathrm{d}}{\mathrm{d}x} \frac{\partial f(x,y)}{\partial y} && = \frac{\partial^2 f(x,y)}{\partial x \partial y} + \frac{\partial^2 f(x,y)}{\partial y^2} \frac{\mathrm{d}y}{\mathrm{d}x} \\
\text{RHS :} \quad & \frac{\mathrm{d}}{\mathrm{d}x} 0 && = 0
\end{aligned}$$

Equating and rearranging gives the result. Or directly from Dini's implicit function theorem on $\frac{\partial f(x,y)}{\partial y} = 0$.

Differentiable Optimisation: Big Picture Idea

$$\begin{array}{rl}\text{min.} & f_0(x, u) \\ \text{s.t.} & u \in C(x)\end{array} \qquad\qquad \begin{array}{rl}\text{min.} & f_0(x + \mathrm{d}x, u) \\ \text{s.t.} & u \in C(x + \mathrm{d}x)\end{array}$$



Differentiating (Unconstrained) Optimisation Problems

Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$. Let

$$y(x) \in \operatorname*{arg\,min}_{u \in \mathbb{R}^m} f(x, u)$$

Assume that $y(x)$ exists and that f is twice differentiable in the neighbourhood of $(x, y(x))$. Then for H non-singular

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = -H^{-1}B$$

where

$$B = \tfrac{\partial^2 f(x,y)}{\partial x \partial y} \in \mathbb{R}^{m \times n} \qquad H = \tfrac{\partial^2 f(x,y)}{\partial y^2} \in \mathbb{R}^{m \times m}$$

Differentiating (Unconstrained) Optimisation Problems

Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$. Let

$$y(x) \in \operatorname*{arg\,min}_{u \in \mathbb{R}^m} \; f(x, u)$$

Assume that $y(x)$ exists and that f is twice differentiable in the neighbourhood of $(x, y(x))$. Then for H non-singular

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = -H^{-1}B$$

where

$$B = \tfrac{\partial^2 f(x,y)}{\partial x \partial y} \in \mathbb{R}^{m \times n} \qquad H = \tfrac{\partial^2 f(x,y)}{\partial y^2} \in \mathbb{R}^{m \times m}$$

This result can be extended to constrained optimisation problems by differentiating optimality conditions, e.g., $\nabla \mathcal{L} = 0$.

result

Automatic Differentiation for Differentiable Optimisation

(assuming a closed-form optimal solutions does not exist)

- At one extreme we can try back propagate through the optimisation algorithm (i.e., unrolling the optimisation procedure using automatic differentiation)
- At the other extreme we can use the implicit differentiation result to hand-craft efficient backward pass code
- There are also options in between, e.g.,
  - use automatic differentiation to obtain quantities in expression for $\frac{\mathrm{d}y(x)}{\mathrm{d}x}$ from software implementations of the objective and (active) constraint functions
  - implement the optimality condition $\nabla\mathcal{L} = 0$ in software and automatically differentiate that

Vector-Jacobian Product

For brevity consider the unconstrained optimisation case. The backward pass computes

$$\begin{aligned}\frac{\mathrm{d}L}{\mathrm{d}x} &= \frac{\mathrm{d}L}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x} \\ &= \underbrace{\left(v^T\right)}_{\mathbb{R}^{1\times m}}\underbrace{\left(-H^{-1}B\right)}_{\mathbb{R}^{m\times n}}\end{aligned}$$

evaluation order:	$-v^T\left(H^{-1}B\right)$	$\left(-v^T H^{-1}\right)B$
cost†:	$O(m^2n + mn)$	$O(m^2 + mn)$

† assumes H^{-1} is already factored (in $O(m^3)$ if unstructured, less if structured)

Summary and Open Questions

- optimisation problems can be embedded *inside* deep learning models
- back-propagation by either unrolling the optimisation algorithm or implicit differentiation of the optimality conditions
  - the former is easy to implement using automatic differentiation but memory intensive
  - the latter requires that solution be strongly convex locally (i.e., invertible H)
  - but does not need to know how the problem was solved, nor store intermediate forward-pass calculations
  - computing H^{-1} may be costly

Summary and Open Questions

- optimisation problems can be embedded *inside* deep learning models
- back-propagation by either unrolling the optimisation algorithm or implicit differentiation of the optimality conditions
  - the former is easy to implement using automatic differentiation but memory intensive
  - the latter requires that solution be strongly convex locally (i.e., invertible H)
  - but does not need to know how the problem was solved, nor store intermediate forward-pass calculations
  - computing H^{-1} may be costly
- active area of research and many open questions
  - Are declarative nodes slower?
  - Do declarative nodes give theoretical guarantees?
  - How best to handle non-smooth or discrete optimization problems?
  - What about problems with multiple solutions?
  - What if the forward pass solution is suboptimal?
  - Can problems become infeasible during learning?
  - ...

Part III. Applications

Optimal Transport

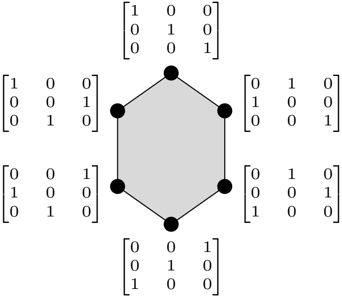
One view of optimal transport is as a matching problem

- from an m-by-n cost matrix M
- to an m-by-n probability matrix P,

often formulated with an entropic regularisation term,

$$
\begin{array}{ll}
\text{minimize} & \langle M, P \rangle + \frac{1}{\gamma}\langle P, \log P \rangle \\
\text{subject to} & P\mathbf{1} = r \\
 & P^T\mathbf{1} = c
\end{array}
$$

with $\mathbf{1}^T r = \mathbf{1}^T c = 1$.



The row and column sum constraints ensure that P is a doubly stochastic matrix (lies within the convex hull of permutation matrices).

Solving Entropic Optimal Transport

Solution takes the form

$$P_{ij} = \alpha_i \beta_j e^{-\gamma M_{ij}}$$

and can be found using the Sinkhorn algorithm,

- Set $K_{ij} = e^{-\gamma M_{ij}}$ and $\alpha, \beta \in \mathbb{R}^n_{++}$
- Iterate until convergence,

$$\begin{aligned}\alpha &\leftarrow r \oslash K\beta \\ \beta &\leftarrow c \oslash K^T\alpha\end{aligned}$$

  where $\oslash$ denotes componentwise division
- Return $P = \mathbf{diag}(\alpha) K \mathbf{diag}(\beta)$

Differentiable Optimal Transport

- Option 1: back-propagate through Sinkhorn algorithm

Differentiable Optimal Transport

- Option 1: back-propagate through Sinkhorn algorithm
- Option 2: use the implicit differentiation result

$$\underbrace{\frac{\mathrm{d}L}{\mathrm{d}M}}_{m\text{-by-}n} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}P}}_{m\text{-by-}n} \frac{\overbrace{\mathrm{d}P}^{m\text{-by-}n\text{-by-}m\text{-by-}n}}{\mathrm{d}M}$$

Differentiable Optimal Transport

- Option 1: back-propagate through Sinkhorn algorithm
- Option 2: use the implicit differentiation result

$$\underbrace{\frac{\mathrm{d}L}{\mathrm{d}M}}_{\text{1-by-}mn} = \underbrace{\frac{\mathrm{d}L}{\mathrm{d}P}}_{\text{1-by-}mn} \overbrace{\frac{\mathrm{d}P}{\mathrm{d}M}}^{mn\text{-by-}mn}$$

(think of vectorising M and P)

Optimal Transport Gradient

Derivation of the optimal transport gradient is quite tedious (see notes). The result:

$$
\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}M} &= \frac{\mathrm{d}L}{\mathrm{d}P}\left(H^{-1}A^T\left(AH^{-1}A^T\right)^{-1}AH^{-1} - H^{-1}\right)B \\
&= \gamma\frac{\mathrm{d}L}{\mathrm{d}P}\textbf{diag}(P)\begin{bmatrix} A_1 \\ A_2 \end{bmatrix}^T \begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^T & \Lambda_{22} \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \textbf{diag}(P) - \gamma\frac{\mathrm{d}L}{\mathrm{d}P}\textbf{diag}(P)
\end{aligned}
$$

where

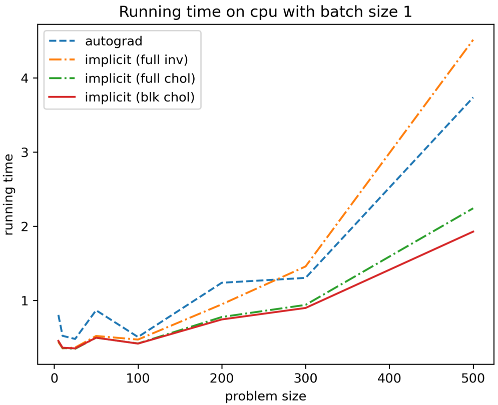
$$
\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0}_n^T & \mathbf{1}_n^T & \cdots & \mathbf{0}_n^T \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0}_n^T & \mathbf{0}_n^T & \cdots & \mathbf{1}_n^T \\ I_{n\times n} & I_{n\times n} & \cdots & I_{n\times n} \end{bmatrix}
$$

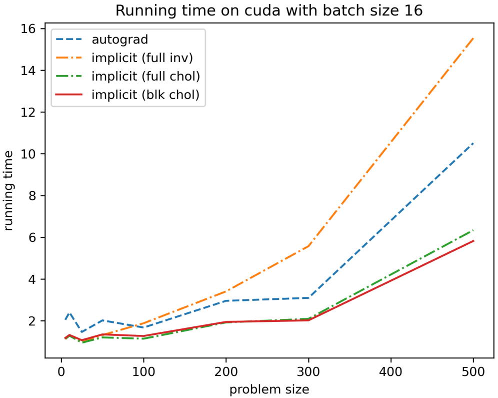
$$
\begin{aligned}
\left(AH^{-1}A^T\right)^{-1} &= \frac{1}{\gamma}\begin{bmatrix} \Lambda_{11} & \Lambda_{12} \\ \Lambda_{12}^T & \Lambda_{22} \end{bmatrix} \\
&= \frac{1}{\gamma}\begin{bmatrix} \textbf{diag}(r_{2:m}) & P_{2:m,1:n} \\ P_{2:m,1:n}^T & \textbf{diag}(c) \end{bmatrix}^{-1}
\end{aligned}
$$

Implementation

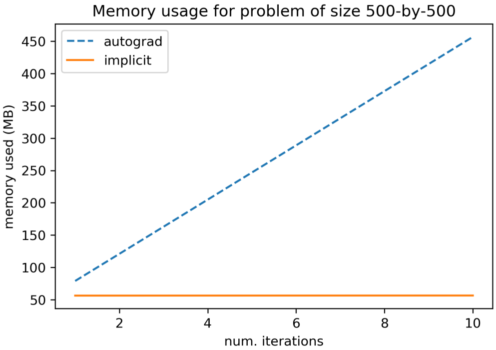
```python
@staticmethod
def backward(ctx, dJdP)
    # unpacked cached tensors
    M, r, c, P = ctx.saved_tensors
    batches, m, n = P.shape

    # initialize backward gradients (-v^T H^{-1} B)
    dLdM = -1.0 * gamma * P * dLdP

    # compute [vHAt1, vHAt2] = -v^T H^{-1} A^T
    vHAt1, vHAt2 = sum(dJdM[:, 1:m, 0:n], dim=2), sum(dJdM, dim=1)

    # compute [v1, v2] = -v^T H^{-1} A^T (A H^{-1] A^T)^{-1}
    P_over_c = P[:, 1:m, 0:n] / c.view(batches, 1, n)
    lmd_11 = cholesky(diag_embed(r[:, 1:m]) - bmm(P[:,1:m,0:n], P_over_c.transpose(1, 2)))
    lmd_12 = cholesky_solve(P_over_c, lmd_11)
    lmd_22 = diag_embed(1.0 / c) + bmm(lmd_12.transpose(1, 2), P_over_c)

    v1 = torch.cholesky_solve(vHAt1, lmd_11) - torch.bmm(lmd_12, vHAt2)
    v2 = torch.bmm(lmd_22, vHAt2) - torch.bmm(lmd_12.transpose(1, 2), vHAt1)

    # compute v^T H^{-1] A^T (A H^{-1] A^T)^{-1} A H^{-1} B - v^T H^{-1} B
    dLdM[:, 1:m, 0:n] -= v1.view(batches, m-1, 1) * P[:, 1:m, 0:n]
    dJdM -= v2.view(batches, 1, n) * P

    # return gradients
    return dJdM
```

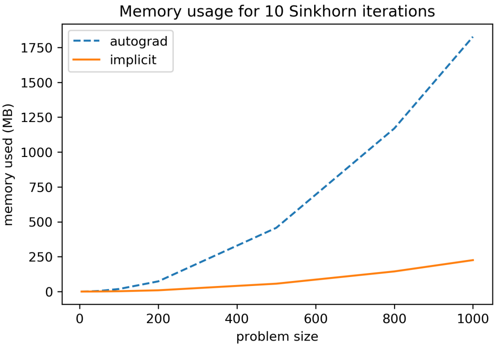
Running Time





Memory Usage





Application to Blind Perspective-n-Point

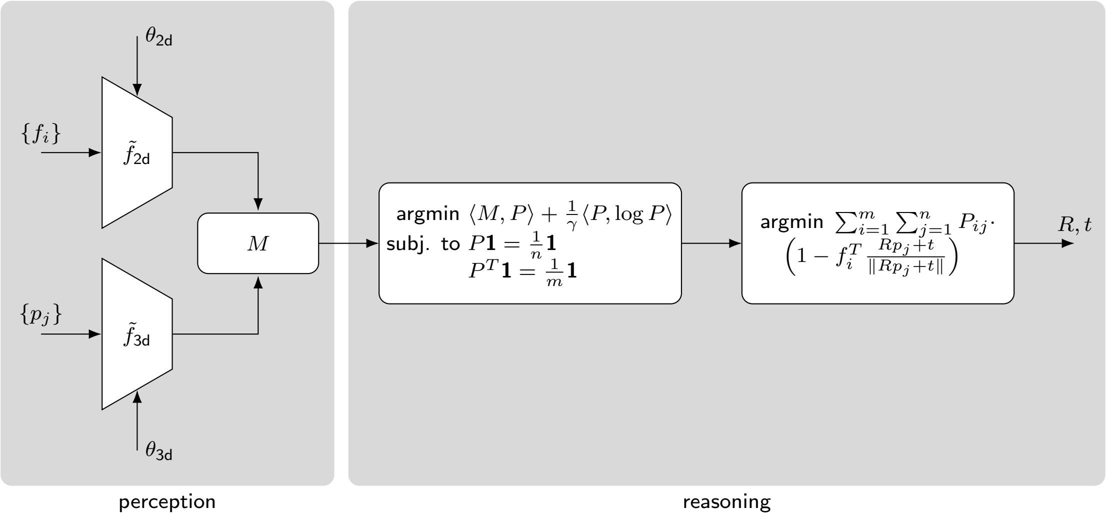
(Campbell et al., ECCV 2020)

find the location where the photograph was taken

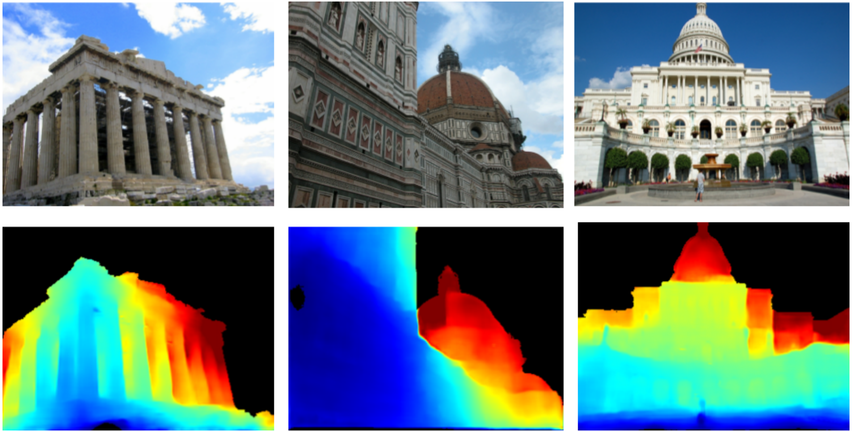
Coupled Problem

- if we knew **correspondences** then determining **camera pose** would be easy
- if we knew **camera pose** then determining **correspondences** would be easy

Blind Perspective-n-Point Network Architecture



Blind Perspective-n-Point Results



more examples

Further Resources

Diving deeper from here?

- background reading

- Deep declarative networks (`http://deepdeclarativenetworks.com`)
  - lots of small code examples and tutorials
- CVXPyLayers (`https://github.com/cvxgrp/cvxpylayers`)
- Theseus (`https://sites.google.com/view/theseus-ai`)
- JAXopt (`https://github.com/google/jaxopt`)

lecture notes available at `https://users.cecs.anu.edu.au/~sgould`

break-out slides

automatic differentiation

Toy Example: Babylonian Algorithm ⏩ back

Consider the following implementation for a forward operation:

procedure $\textsc{FwdFcn}(x)$
  $y_0 \leftarrow \frac{1}{2}x$
  for $t = 1, \ldots, T$ **do**
    $y_t \leftarrow \frac{1}{2}\left(y_{t-1} + \frac{x}{y_{t-1}}\right)$
  end for
  return y_T
end procedure

Toy Example: Babylonian Algorithm back

Consider the following implementation for a forward operation:

procedure $\textsc{FwdFcn}(x)$
  $y_0 \leftarrow \frac{1}{2}x$
  for $t = 1, \ldots, T$ **do**
    $y_t \leftarrow \frac{1}{2}\left(y_{t-1} + \frac{x}{y_{t-1}}\right)$
  end for
  return y_T
end procedure

Automatic differentiation algorithmically generates the backward code:

procedure $\textsc{BckFcn}(x, y_T, \frac{\mathrm{d}L}{\mathrm{d}y_T})$
  $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow 0$
  for $t = T, \ldots, 1$ **do**
    $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}x} + \frac{\mathrm{d}L}{\mathrm{d}y_t} \overbrace{\left(\frac{1}{2y_{t-1}}\right)}^{\partial y_t / \partial x}$
    $\frac{\mathrm{d}L}{\mathrm{d}y_{t-1}} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}y_t} \underbrace{\left(\frac{1}{2} - \frac{x}{2y_{t-1}^2}\right)}_{\partial y_t / \partial y_{t-1}}$
  end for
  $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}x} + \frac{\mathrm{d}L}{\mathrm{d}y_0}\frac{1}{2}$
  return $\frac{\mathrm{d}L}{\mathrm{d}x}$
end procedure

Toy Example: Babylonian Algorithm back

Consider the following implementation for a forward operation:

procedure $\textsc{FwdFcn}(x)$
  $y_0 \leftarrow \frac{1}{2}x$
  for $t = 1, \ldots, T$ **do**
    $y_t \leftarrow \frac{1}{2}\left(y_{t-1} + \frac{x}{y_{t-1}}\right)$
  end for
  return y_T
end procedure

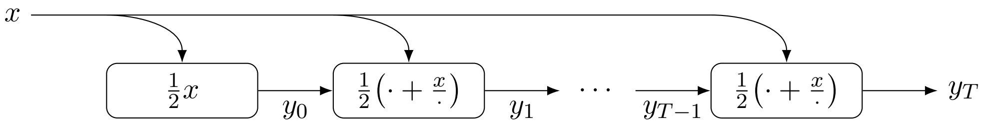
- computes $y = \sqrt{x}$
- derivative computed directly is $\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{1}{2\sqrt{x}} = \frac{1}{2y}$

Automatic differentiation algorithmically generates the backward code:

procedure $\textsc{BckFcn}(x, y_T, \frac{\mathrm{d}L}{\mathrm{d}y_T})$
  $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow 0$
  for $t = T, \ldots, 1$ **do**
    $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}x} + \frac{\mathrm{d}L}{\mathrm{d}y_t}\overbrace{\left(\frac{1}{2y_{t-1}}\right)}^{\partial y_t / \partial x}$
    $\frac{\mathrm{d}L}{\mathrm{d}y_{t-1}} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}y_t}\underbrace{\left(\frac{1}{2} - \frac{x}{2y_{t-1}^2}\right)}_{\partial y_t / \partial y_{t-1}}$
  end for
  $\frac{\mathrm{d}L}{\mathrm{d}x} \leftarrow \frac{\mathrm{d}L}{\mathrm{d}x} + \frac{\mathrm{d}L}{\mathrm{d}y_0}\frac{1}{2}$
  return $\frac{\mathrm{d}L}{\mathrm{d}x}$
end procedure

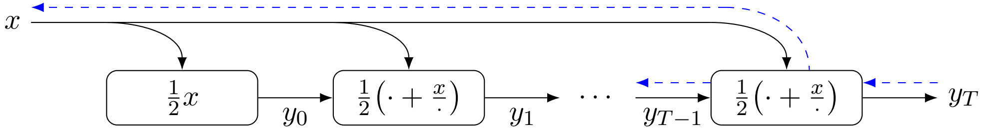
Computation Graph for Babylonian Algorithm

back



$$y_T = f(x, f(x, f(x, \ldots f(x, \tfrac{1}{2}x)))) \text{ with } f(x, y) = \frac{1}{2}\left(y + \frac{x}{y}\right)$$

Computation Graph for Babylonian Algorithm ⏩ back



$$y_T = f(x, f(x, f(x, \ldots f(x, \tfrac{1}{2}x)))) \text{ with } f(x, y) = \frac{1}{2}\left(y + \frac{x}{y}\right)$$

$$\frac{\mathrm{d}L}{\mathrm{d}x} = \frac{\mathrm{d}L}{\mathrm{d}y_T}\left(\frac{\partial y_T}{\partial x} + \frac{\partial y_T}{\partial y_{T-1}}\left(\frac{\partial y_{T-1}}{\partial x} + \frac{\partial y_{T-1}}{\partial y_{T-2}}\left(\ldots + \frac{\partial y_0}{\partial x}\right)\right)\right)$$

least squares

Least Squares Backward Pass Derivation ⏩ back

Differentiating $x^\star$ with respect to single element A_{ij}, we have

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}A_{ij}} x^\star &= \frac{\mathrm{d}}{\mathrm{d}A_{ij}} \left(A^T A\right)^{-1} A^T b \\
&= \left(\frac{\mathrm{d}}{\mathrm{d}A_{ij}} \left(A^T A\right)^{-1}\right) A^T b + \left(A^T A\right)^{-1} \left(\frac{\mathrm{d}}{\mathrm{d}A_{ij}} A^T b\right)
\end{aligned}
$$

Using the identity $\frac{\mathrm{d}}{\mathrm{d}z} Z^{-1} = -Z^{-1} \left(\frac{\mathrm{d}}{\mathrm{d}z} Z\right) Z^{-1}$ we get, for the first term,

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}A_{ij}} \left(A^T A\right)^{-1} &= -\left(A^T A\right)^{-1} \left(\frac{\mathrm{d}}{\mathrm{d}A_{ij}} \left(A^T A\right)\right) \left(A^T A\right)^{-1} \\
&= -\left(A^T A\right)^{-1} \left(E_{ij}^T A + A^T E_{ij}\right) \left(A^T A\right)^{-1}
\end{aligned}
$$

where E_{ij} is a matrix with one in the (i, j)-th element and zeros elsewhere. Furthermore, for the second term,

$$
\frac{\mathrm{d}}{\mathrm{d}A_{ij}} A^T b = E_{ij}^T b
$$

Least Squares Backward Pass Derivation (cont.) ⏩ back

Plugging these back into parent equation we have

$$
\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}A_{ij}} x^\star &= -\left(A^T A\right)^{-1} \left(E_{ij}^T A + A^T E_{ij}\right) \left(A^T A\right)^{-1} A^T b + \left(A^T A\right)^{-1} E_{ij}^T b \\
&= -\left(A^T A\right)^{-1} \left(E_{ij}^T A + A^T E_{ij}\right) x^\star + \left(A^T A\right)^{-1} E_{ij}^T b \\
&= -\left(A^T A\right)^{-1} \left(E_{ij}^T (A x^\star - b) + A^T E_{ij} x^\star\right) \\
&= -\left(A^T A\right)^{-1} \left((a_i^T x^\star - b_i) e_j + x_j^\star a_i\right)
\end{aligned}
$$

where $e_j = (0, 0, \ldots, 1, 0, \ldots) \in \mathbb{R}^n$ is the j-th canonical vector, i.e., vector with a one in the j-th component and zeros everywhere else, and $a_i^T \in \mathbb{R}^{1 \times n}$ is the i-th row of matrix A.

Least Squares Backward Pass Derivation (cont.) back

Let $r = b - Ax^\star$ and let v^T denote the backward coming gradient $\frac{\mathrm{d}}{\mathrm{d}x^\star}L$. Then

$$
\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}A_{ij}} &= v^T \frac{\mathrm{d}x^\star}{\mathrm{d}A_{ij}} \\
&= v^T \left(A^T A\right)^{-1} \left(r_i e_j - x^\star_j a_i\right) \\
&= w^T \left(r_i e_j - x^\star_j a_i\right) \\
&= r_i w_j - w^T a_i x^\star_j
\end{aligned}
$$

where $w = \left(A^T A\right)^{-1} v$. We can compute the entire matrix of $m \times n$ derivatives efficiently as the sum of outer products

$$
\left(\frac{\mathrm{d}L}{\mathrm{d}A}\right)^T = \left[\frac{\mathrm{d}L}{\mathrm{d}A_{ij}}\right]_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} = wr^T - x^\star (Aw)^T
$$

differentiating equality constrained problems

Differentiating Equality Constrained Optimisation Problems ⏩ back

Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^q$. Let

$$\begin{array}{rll} y(x) \in & \arg\min_{u \in \mathbb{R}^m} & f(x, u) \\ & \text{subject to} & h(x, u) = 0_q \end{array}$$

Assume that $y(x)$ exists, that f and h are twice differentiable in the neighbourhood of $(x, y(x))$, and that **rank**$\left(\frac{\partial h(x,y)}{\partial y}\right) = q$.

Differentiating Equality Constrained Optimisation Problems [back]

Consider functions $f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^q$. Let

$$\begin{array}{rll} y(x) \in & \arg\min_{u \in \mathbb{R}^m} & f(x, u) \\ & \text{subject to} & h(x, u) = 0_q \end{array}$$

Assume that $y(x)$ exists, that f and h are twice differentiable in the neighbourhood of $(x, y(x))$, and that **rank**$(\frac{\partial h(x,y)}{\partial y}) = q$. Then for H non-singular

$$\frac{\mathrm{d}y(x)}{\mathrm{d}x} = H^{-1}A^T\left(AH^{-1}A^T\right)^{-1}\left(AH^{-1}B - C\right) - H^{-1}B$$

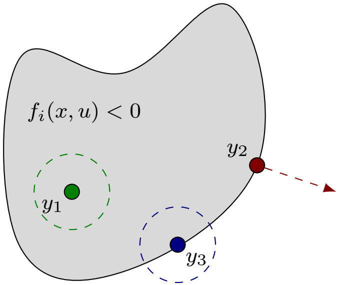
where

$$\begin{array}{ll} A = \frac{\partial h(x,y)}{\partial y} \in \mathbb{R}^{q \times m} & B = \frac{\partial^2 f(x,y)}{\partial x \partial y} - \sum_{i=1}^q \nu_i \frac{\partial^2 h_i(x,y)}{\partial x \partial y} \in \mathbb{R}^{m \times n} \\ C = \frac{\partial h(x,y)}{\partial x} \in \mathbb{R}^{q \times n} & H = \frac{\partial^2 f(x,y)}{\partial y^2} - \sum_{i=1}^q \nu_i \frac{\partial^2 h_i(x,y)}{\partial y^2} \in \mathbb{R}^{m \times m} \end{array}$$

and $\nu \in \mathbb{R}^q$ satisfies $\nu^T A = \frac{\partial f(x,y)}{\partial y}$.

Dealing with Inequality Constraints back

$$
\begin{array}{rll}
y(x) \in \operatorname{arg\,min}_{u \in \mathbb{R}^m} & f_0(x, u) & \\
\text{subject to} & h_i(x, u) = 0, & i = 1, \ldots, q \\
& f_i(x, u) \leq 0, & i = 1, \ldots, p.
\end{array}
$$



- Replace inequality constraints with log-barrier approximation
- Treat as equality constraints if active (y_2 or y_3) and ignore otherwise (y_1 or y_3)
  - may lead to one-sided gradients since $\nu \succeq 0$

eigen decomposition

Deriving the Gradient for Eigen Decomposition ⏩ back

Implicit differentiation of the optimality conditions with respect to X_{ij} gives,

$$\frac{\mathrm{d}}{\mathrm{d}X_{ij}}(Xy - \lambda_{\text{max}}y) = \frac{1}{2}(E_{ij} + E_{ji})y - \frac{\mathrm{d}\lambda_{\text{max}}}{\mathrm{d}X_{ij}}y + (X - \lambda_{\text{max}}I)\frac{\mathrm{d}y}{\mathrm{d}X_{ij}} = 0 \tag{1}$$

$$\frac{\mathrm{d}}{\mathrm{d}X_{ij}}(y^T y - 1) = 2y^T \frac{\mathrm{d}y}{\mathrm{d}X_{ij}} = 0 \tag{2}$$

Pre-multiplying (1) by y^T, and using (2) and $y^T y = 1$, we get

$$\frac{\mathrm{d}\lambda_{\text{max}}}{\mathrm{d}X_{ij}} = \frac{1}{2}y^T(E_{ij} + E_{ji})y$$

Pre-multiplying (1) by $(X - \lambda_{\text{max}}I)^\dagger$, we get

$$\frac{1}{2}(X - \lambda_{\text{max}}I)^\dagger(E_{ij} + E_{ji})y - (X - \lambda_{\text{max}}I)^\dagger\frac{\mathrm{d}\lambda_{\text{max}}}{\mathrm{d}X_{ij}}y + \frac{\mathrm{d}y}{\mathrm{d}X_{ij}} = 0$$

$$\therefore\ \frac{\mathrm{d}y}{\mathrm{d}X_{ij}} = -\frac{1}{2}(X - \lambda_{\text{max}}I)^\dagger(E_{ij} + E_{ji})y$$

since $(X - \lambda_{\text{max}}I)^\dagger \frac{\mathrm{d}\lambda_{\text{max}}}{\mathrm{d}X_{ij}} y = \frac{\mathrm{d}\lambda_{\text{max}}}{\mathrm{d}X_{ij}}(X - \lambda_{\text{max}}I)^\dagger y = 0$ since if $Az = 0$, then $A^\dagger z = 0$.

additional examples

Differentiable Eigen Decomposition

Finding the eigenvector corresponding to the maximum eigenvalue of a real symmetric matrix $X \in \mathbb{R}^{m \times m}$ can be formulated as

$$\begin{array}{ll} \text{maximize (over } u \in \mathbb{R}^m) & u^T X u \\ \text{subject to} & u^T u = 1 \end{array}$$

which has applications in, for example, back propagating through normalized cuts.

Differentiable Eigen Decomposition

Finding the eigenvector corresponding to the maximum eigenvalue of a real symmetric matrix $X \in \mathbb{R}^{m \times m}$ can be formulated as

$$\begin{array}{ll} \text{maximize (over } u \in \mathbb{R}^m) & u^T X u \\ \text{subject to} & u^T u = 1 \end{array}$$

which has applications in, for example, back propagating through normalized cuts. Optimality conditions (for solution y) are

$$Xy = \lambda_{\mathsf{max}} y \quad \text{and} \quad y^T y = 1.$$

Differentiable Eigen Decomposition

Finding the eigenvector corresponding to the maximum eigenvalue of a real symmetric matrix $X \in \mathbb{R}^{m \times m}$ can be formulated as

$$\begin{array}{ll} \text{maximize (over } u \in \mathbb{R}^m) & u^T X u \\ \text{subject to} & u^T u = 1 \end{array}$$

which has applications in, for example, back propagating through normalized cuts. Optimality conditions (for solution y) are

$$Xy = \lambda_{\text{max}} y \quad \text{and} \quad y^T y = 1.$$

Taking derivatives with respect to components of X we get,

$$\frac{\mathrm{d}y}{\mathrm{d}X_{ij}} = -\frac{1}{2}(X - \lambda_{\text{max}} I)^{\dagger}(E_{ij} + E_{ji})y \quad \in \mathbb{R}^m$$

derivation

PyTorch Implementation

```python
class EigenDecompositionFcn(torch.autograd.Function):
    """PyTorch autograd function for eigen decomposition."""

    @staticmethod
    def forward(ctx, X):
        B, M, N = X.shape

        # use torch's eigh function to find the eigenvalues and eigenvectors of a symmetric matrix
        with torch.no_grad():
            lmd, Y = torch.linalg.eigh(0.5 * (X + X.transpose(1, 2)))

        ctx.save_for_backward(lmd, Y)
        return Y

    @staticmethod
    def backward(ctx, dJdY):
        lmd, Y = ctx.saved_tensors
        B, M, N = Y.shape

        # compute all pseudo-inverses simultaneously
        L = lmd.view(B, 1, M) - lmd.view(B, M, 1)
        L = torch.where(torch.abs(L) < eps, 0.0, 1.0 / L)

        # compute full gradient over all eigenvectors
        dJdX = torch.bmm(torch.bmm(Y, L * torch.bmm(Y.transpose(1, 2), dJdY)), Y.transpose(1, 2))
        dJdX = 0.5 * (dJdX + dJdX.transpose(1, 2))

        return dJdX
```

Experiment ⏩ back