

Automatic differentiation

Mathieu Blondel

November 20, 2020

Gradient-based learning

- Gradient-based training algorithms are the workhorse of modern machine learning.
- Deriving gradients by hand is tedious and error prone.
- This becomes quickly infeasible for complex models.
- Changes to the model require rederiving the gradient.
- Deep learning = GPU + data + autodiff

Automatic differentiation

- Evaluates the derivatives of a function at a given point.
- Not the same as numerical differentiation.
- Not the same as symbolic differentiation, which returns a “human-readable” expression.
- In a neural network context, reverse autodiff is often known as backpropagation.

Automatic differentiation

- A program is defined as the composition of primitive operations that we know how to derive.
- The user can focus on the forward computation / model.

```
import jax.numpy as jnp
from jax import grad, jit

def predict(params, inputs):
    for W, b in params:
        outputs = jnp.dot(inputs, W) + b
        inputs = jnp.tanh(outputs)
    return outputs

def logprob_fun(params, inputs, targets):
    preds = predict(params, inputs)
    return jnp.sum((preds - targets)**2)

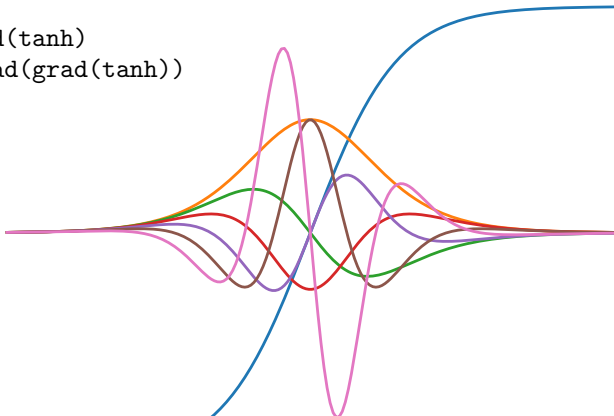
grad_fun = jit(grad(logprob_fun))
```

Automatic differentiation

- Modern frameworks support higher-order derivatives

```
def tanh(x):  
    y = jnp.exp(-2.0 * x)  
    return (1.0 - y) / (1.0 + y)
```

```
fp = grad(tanh)  
fpp = grad(grad(tanh))  
...
```



Outline

1 Numerical differentiation

2 Chain compositions

3 Computational graphs

4 Implementation

5 Advanced topics

6 Conclusion

Derivatives

- Definition of derivative of $g: \mathbb{R} \rightarrow \mathbb{R}$

$$g'(a) = \frac{\partial g(a)}{\partial a} = \lim_{h \rightarrow 0} \frac{g(a+h) - g(a)}{h}$$

- $g'(a)$ is called Lagrange notation.
- $\frac{\partial g(a)}{\partial a}$ is called Leibniz notation.
- Interpretations: instantaneous rate of change of g , slope of the tangent of g at a .

Gradient

- The gradient of $f: \mathbb{R}^n \rightarrow \mathbb{R}$ is

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_n}(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^n$$

i.e., a vector that gathers the partial derivatives of f .

- Applying the definition of derivative coordinate-wise:

$$[\nabla f(\mathbf{x})]_j = \frac{\partial f}{\partial x_j}(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{e}_j) - f(\mathbf{x})}{h} \quad j \in \{1, \dots, n\}$$

where $\mathbf{e}_j = [0, 0, \dots, 0, \underbrace{1}_j, 0, \dots, 0]^\top \in \{0, 1\}^n$ is the j^{th} standard basis vector.

Numerical gradient

- Finite difference:

$$[\nabla f(\mathbf{x})]_j = \frac{\partial f}{\partial x_j}(\mathbf{x}) \approx \frac{f(\mathbf{x} + \varepsilon \mathbf{e}_j) - f(\mathbf{x})}{\varepsilon} \quad j \in \{1, \dots, n\}$$

where ε is a small value (e.g., 10^{-6}).

- Central finite difference:

$$[\nabla f(\mathbf{x})]_j = \frac{\partial f}{\partial x_j}(\mathbf{x}) \approx \frac{f(\mathbf{x} + \varepsilon \mathbf{e}_j) - f(\mathbf{x} - \varepsilon \mathbf{e}_j)}{2\varepsilon} \quad j \in \{1, \dots, n\}$$

- Computing $\nabla f(\mathbf{x})$ approximately by (central) finite difference is $n + 1$ times ($2n$ times) as costly as evaluating f .

Directional derivative

- Derivative of $f: \mathbb{R}^n \rightarrow \mathbb{R}$ in the direction of $\mathbf{v} \in \mathbb{R}^n$

$$D_{\mathbf{v}}f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h} \in \mathbb{R}$$

- Interpretation: rate of change of f in the direction of \mathbf{v} , when moving away from \mathbf{x} .
- $[\nabla f(\mathbf{x})]_i$ is the derivative in the direction of \mathbf{e}_i .
- Finite difference (and similarly for the central finite difference):

$$D_{\mathbf{v}}f(\mathbf{x}) \approx \frac{f(\mathbf{x} + \varepsilon\mathbf{v}) - f(\mathbf{x})}{\varepsilon}$$

Only 2 calls to f are needed, i.e., independent of n .

Directional derivative

- **Fact.** The directional derivative is equal to the scalar product between the gradient and \mathbf{v} , i.e.,

$$D_{\mathbf{v}}f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v}$$

- **Proof.** Let $g(t) = f(\mathbf{x} + t\mathbf{v})$. We have

$$g'(t) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + (t+h)\mathbf{v}) - f(\mathbf{x} + t\mathbf{v})}{h}$$

and therefore $g'(0) = D_{\mathbf{v}}f(\mathbf{x})$. By the chain rule, we also have

$$g'(t) = \nabla f(\mathbf{x} + t\mathbf{v}) \cdot \mathbf{v}.$$

Hence, $g'(0) = D_{\mathbf{v}}f(\mathbf{x}) = \nabla f(\mathbf{x}) \cdot \mathbf{v}$.

Jacobian

- The Jacobian of $\mathbf{f}: \mathbb{R}^n \rightarrow \mathbb{R}^m$

$$\begin{aligned} J_{\mathbf{f}}(\mathbf{x}) &= \frac{\partial \mathbf{f}(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \\ &= \left[\frac{\partial \mathbf{f}}{\partial x_1}, \dots, \frac{\partial \mathbf{f}}{\partial x_n} \right] \\ &= \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_m(\mathbf{x})^\top \end{bmatrix} \end{aligned}$$

- The size of the Jacobian matrix is $m \times n$.
- The gradient's transpose is thus a “wide” Jacobian ($m = 1$).

Jacobian vector product (“JVP”)

- Right-multiply the Jacobian with a vector $\mathbf{v} \in \mathbb{R}^n$

$$\begin{aligned} J_{\mathbf{f}}(\mathbf{x})\mathbf{v} &= \begin{bmatrix} \nabla f_1(\mathbf{x})^\top \\ \vdots \\ \nabla f_m(\mathbf{x})^\top \end{bmatrix} \mathbf{v} \\ &= \begin{bmatrix} \nabla f_1(\mathbf{x}) \cdot \mathbf{v} \\ \vdots \\ \nabla f_m(\mathbf{x}) \cdot \mathbf{v} \end{bmatrix} \\ &= \lim_{h \rightarrow 0} \frac{\mathbf{f}(\mathbf{x} + h\mathbf{v}) - \mathbf{f}(\mathbf{x})}{h} \end{aligned}$$

- Finite difference (and similarly for the central finite difference):

$$J_{\mathbf{f}}(\mathbf{x})\mathbf{v} \approx \frac{\mathbf{f}(\mathbf{x} + \varepsilon\mathbf{v}) - \mathbf{f}(\mathbf{x})}{\varepsilon}$$

- Computing the JVP approximately by (central) finite difference requires only 2 calls to \mathbf{f} .

Vector Jacobian Product (“VJP”)

- Left-multiply the Jacobian with a vector $\mathbf{u} \in \mathbb{R}^m$

$$\mathbf{u}^\top \mathbf{J}_{\mathbf{f}}(\mathbf{x}) = \mathbf{u}^\top \left[\frac{\partial \mathbf{f}}{\partial x_1}, \dots, \frac{\partial \mathbf{f}}{\partial x_n} \right] = \left[\mathbf{u} \cdot \frac{\partial \mathbf{f}}{\partial x_1}, \dots, \mathbf{u} \cdot \frac{\partial \mathbf{f}}{\partial x_n} \right]$$

- Finite difference (and similarly for the central finite difference):

$$\frac{\partial \mathbf{f}}{\partial x_i} \approx \frac{\mathbf{f}(\mathbf{x} + \varepsilon \mathbf{e}_i) - \mathbf{f}(\mathbf{x})}{\varepsilon}$$

- Computing the VJP approximately by (central) finite difference requires $n + 1$ calls ($2n$ calls) to \mathbf{f} .

Outline

1 Numerical differentiation

2 Chain compositions

3 Computational graphs

4 Implementation

5 Advanced topics

6 Conclusion

Chain rule

- Let $F(x) = f(g(x)) = f \circ g(x)$, where $f, g: \mathbb{R} \rightarrow \mathbb{R}$. Then,

$$F'(x) = f'(g(x))g'(x)$$

- Alternatively, let $y = g(x)$ and $z = f(y)$, then

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} = \left. \frac{\partial z}{\partial y} \right|_{y=g(x)} \left. \frac{\partial y}{\partial x} \right|_{x=x}$$

- Let $f(\mathbf{x}) = h(\mathbf{g}(\mathbf{x}))$, where $\mathbf{g}: \mathbb{R}^n \rightarrow \mathbb{R}^d$ and $h: \mathbb{R}^d \rightarrow \mathbb{R}$. Then,

$$\underbrace{\nabla f(\mathbf{x})}_{n \times 1} = \underbrace{(\nabla h(\mathbf{g}(\mathbf{x})))^\top}_{1 \times d} \underbrace{J_{\mathbf{g}}(\mathbf{x})}_{d \times n}^\top = \underbrace{J_{\mathbf{g}}(\mathbf{x})^\top}_{n \times d} \underbrace{\nabla h(\mathbf{g}(\mathbf{x}))}_{d \times 1}$$

- and similarly using Leibniz notation

Chain compositions



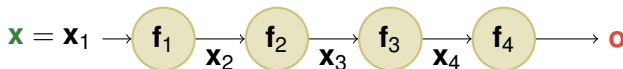
- Assume $\mathbf{f}: \mathbb{R}^n \rightarrow \mathbb{R}^m$ decomposes as follows:

$$\begin{aligned}\mathbf{o} &= \mathbf{f}(\mathbf{x}) \\ &= \mathbf{f}_4 \circ \mathbf{f}_3 \circ \mathbf{f}_2 \circ \mathbf{f}_1(\mathbf{x}) \\ &= \mathbf{f}_4(\mathbf{f}_3(\mathbf{f}_2(\mathbf{f}_1(\mathbf{x}))))\end{aligned}$$

where $\mathbf{f}_1: \mathbb{R}^n \rightarrow \mathbb{R}^{m_1}$, $\mathbf{f}_2: \mathbb{R}^{m_1} \rightarrow \mathbb{R}^{m_2}$, ..., $\mathbf{f}_4: \mathbb{R}^{m_3} \rightarrow \mathbb{R}^m$.

- How to compute the Jacobian $\mathbf{J}_{\mathbf{f}}(\mathbf{x}) = \frac{\partial \mathbf{o}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times n}$ efficiently?

Chain rule



■ Sequence of operations

$$\mathbf{x}_1 = \mathbf{x}$$

$$\mathbf{x}_2 = \mathbf{f}_1(\mathbf{x}_1)$$

$$\mathbf{x}_3 = \mathbf{f}_2(\mathbf{x}_2)$$

$$\mathbf{x}_4 = \mathbf{f}_3(\mathbf{x}_3)$$

$$\mathbf{o} = \mathbf{f}_4(\mathbf{x}_4)$$

■ By the chain rule, we have

$$\begin{aligned} \frac{\partial \mathbf{o}}{\partial \mathbf{x}} &= \frac{\partial \mathbf{o}}{\partial \mathbf{x}_4} \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3} \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}} \\ &= \frac{\partial \mathbf{f}_4(\mathbf{x}_3)}{\partial \mathbf{x}_3} \frac{\partial \mathbf{f}_3(\mathbf{x}_2)}{\partial \mathbf{x}_2} \frac{\partial \mathbf{f}_2(\mathbf{x}_1)}{\partial \mathbf{x}_1} \frac{\partial \mathbf{f}_1(\mathbf{x})}{\partial \mathbf{x}} \\ &= \mathbf{J}_{\mathbf{f}_4}(\mathbf{x}_4) \mathbf{J}_{\mathbf{f}_3}(\mathbf{x}_3) \mathbf{J}_{\mathbf{f}_2}(\mathbf{x}_2) \mathbf{J}_{\mathbf{f}_1}(\mathbf{x}) \end{aligned}$$

Forward differentiation

- Recall that $\frac{\partial \mathbf{f}}{\partial x_j} \in \mathbb{R}^m$ is the j^{th} column of $J_{\mathbf{f}}(\mathbf{x})$.
- Jacobian vector product (JVP) with $\mathbf{e}_j \in \mathbb{R}^n$ “reveals” the j^{th} column

$$J_{\mathbf{f}}(\mathbf{x})\mathbf{e}_1 = \frac{\partial \mathbf{f}}{\partial x_1}$$

$$J_{\mathbf{f}}(\mathbf{x})\mathbf{e}_2 = \frac{\partial \mathbf{f}}{\partial x_2}$$

$$\vdots$$

$$J_{\mathbf{f}}(\mathbf{x})\mathbf{e}_n = \frac{\partial \mathbf{f}}{\partial x_n}$$

- Computing a gradient ($m = 1$) requires n JVPs with $\mathbf{e}_1, \dots, \mathbf{e}_n$.

Forward differentiation

- Jacobian-vector product with $\mathbf{v} \in \mathbb{R}^n$

$$\mathbf{J}_f(\mathbf{x})\mathbf{v} = \underbrace{\mathbf{J}_{f_4}(\mathbf{x}_4)}_{m \times m_3} \underbrace{\mathbf{J}_{f_3}(\mathbf{x}_3)}_{m_3 \times m_2} \underbrace{\mathbf{J}_{f_2}(\mathbf{x}_2)}_{m_2 \times m_1} \underbrace{\mathbf{J}_{f_1}(\mathbf{x})}_{m_1 \times n} \mathbf{v}$$

Multiplication from **right to left** is more efficient.

- Cost of computing n JVPs:

$$n(mm_3 + m_3m_2 + m_2m_1 + m_1n)$$

- Cost of computing a gradient ($m = 1$, $m_3 = m_2 = m_1 = n$):

$$O(n^3)$$

Forward differentiation

$$\blacksquare \text{ } \circ = \mathbf{f}(\mathbf{x}) = \mathbf{f}_K \circ \dots \circ \mathbf{f}_2 \circ \mathbf{f}_1(\mathbf{x})$$

$$\blacksquare [\mathbf{J}_{\mathbf{f}}(\mathbf{x})]_{:,j} = \mathbf{J}_{\mathbf{f}_K}(\mathbf{x}_K) \dots \mathbf{J}_{\mathbf{f}_2}(\mathbf{x}_2) \mathbf{J}_{\mathbf{f}_1}(\mathbf{x}) \mathbf{e}_j \quad j \in \{1, \dots, n\}$$

Algorithm 1 Compute $\circ = \mathbf{f}(\mathbf{x})$ and $\mathbf{J}_{\mathbf{f}}(\mathbf{x})$ alongside

1: **Input:** $\mathbf{x} \in \mathbb{R}^n$

2: $\mathbf{x}_1 \leftarrow \mathbf{x}$

3: $\mathbf{v}_j \leftarrow \mathbf{e}_j \in \mathbb{R}^n \quad j \in \{1, \dots, n\}$

4: **for** $k = 1$ to K **do**

5: $\mathbf{x}_{k+1} \leftarrow \mathbf{f}_k(\mathbf{x}_k)$

6: $\mathbf{v}_j \leftarrow \mathbf{J}_{\mathbf{f}_k}(\mathbf{x}_k) \mathbf{v}_j \quad j \in \{1, \dots, n\}$

7: **end for**

8: **Returns:** $\circ = \mathbf{x}_{K+1}$, $[\mathbf{J}_{\mathbf{f}}(\mathbf{x})]_{:,j} = \mathbf{v}_j \quad j \in \{1, \dots, n\}$

Backward differentiation

- Recall that $\nabla f_i(\mathbf{x})^\top \in \mathbb{R}^n$ is the i^{th} row of $\mathbf{J}_f(\mathbf{x})$.
- Vector Jacobian product (VJP) with $\mathbf{e}_i \in \mathbb{R}^m$ “reveals” the i^{th} row

$$\mathbf{e}_1^\top \mathbf{J}_f(\mathbf{x}) = \nabla f_1(\mathbf{x})^\top$$

$$\mathbf{e}_2^\top \mathbf{J}_f(\mathbf{x}) = \nabla f_2(\mathbf{x})^\top$$

$$\vdots$$

$$\mathbf{e}_m^\top \mathbf{J}_f(\mathbf{x}) = \nabla f_m(\mathbf{x})^\top$$

- Computing a gradient ($m = 1$) requires only 1 VJP with $\mathbf{e}_1 \in \mathbb{R}^1$.

Backward differentiation

- Vector Jacobian product with $\mathbf{u} \in \mathbb{R}^m$

$$\mathbf{u}^\top \underbrace{J_{f_4}(\mathbf{x}_4)}_{m \times m_3} \underbrace{J_{f_3}(\mathbf{x}_3)}_{m_3 \times m_2} \underbrace{J_{f_2}(\mathbf{x}_2)}_{m_2 \times m_1} \underbrace{J_{f_1}(\mathbf{x})}_{m_1 \times n}$$

Multiplication from **left to right** is more efficient.

- Cost of computing m VJPs:

$$m(mm_3 + m_3m_2 + m_2m_1 + m_1n)$$

- Cost of computing a gradient ($m = 1$, $m_3 = m_2 = m_1 = n$):

$$O(n^2)$$

Backward differentiation

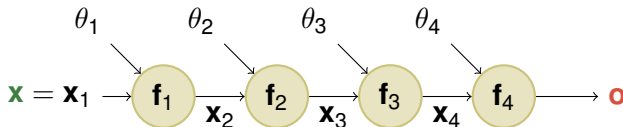
$$\blacksquare \text{ } \circ = \mathbf{f}(\mathbf{x}) = \mathbf{f}_K \circ \dots \circ \mathbf{f}_2 \circ \mathbf{f}_1(\mathbf{x})$$

$$\blacksquare [J_{\mathbf{f}}(\mathbf{x})]_{i,:} = \mathbf{e}_i^\top J_{\mathbf{f}_K}(\mathbf{x}_K) \dots J_{\mathbf{f}_2}(\mathbf{x}_2) J_{\mathbf{f}_1}(\mathbf{x}) \quad i \in \{1, \dots, m\}$$

Algorithm 2 Compute $\circ = \mathbf{f}(\mathbf{x})$ and $J_{\mathbf{f}}(\mathbf{x})$

```
1: Input:  $\mathbf{x} \in \mathbb{R}^n$ 
2:  $\mathbf{x}_1 \leftarrow \mathbf{x}$ ,  $\mathbf{u}_i \leftarrow \mathbf{e}_i \in \mathbb{R}^m \quad i \in \{1, \dots, m\}$ 
3: for  $k = 1$  to  $K$  do
4:    $\mathbf{x}_{k+1} \leftarrow \mathbf{f}_k(\mathbf{x}_k)$ 
5: end for
6: for  $k = K$  to  $1$  do
7:    $\mathbf{u}_i \leftarrow \mathbf{u}_i^\top J_{\mathbf{f}_k}(\mathbf{x}_k) \quad i \in \{1, \dots, m\}$ 
8: end for
9: Returns:  $\circ = \mathbf{x}_{K+1}$ ,  $[J_{\mathbf{f}}(\mathbf{x})]_{i,:} = \mathbf{u}_i \quad i \in \{1, \dots, m\}$ 
```

Feedforward networks



- Each function can now have two arguments: $\mathbf{f}_k(\mathbf{x}_k, \theta_k)$, where \mathbf{x}_k is the previous output and θ_k are learnable parameters.
- Example one hidden layer, one output layer, squared loss

$$\mathbf{f} = \mathbf{f}_4 \circ \dots \circ \mathbf{f}_1$$

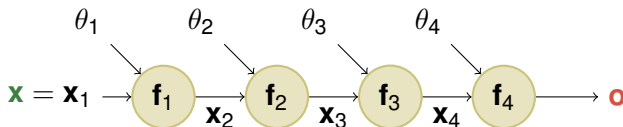
$$\mathbf{x}_2 = \mathbf{f}_1(\mathbf{x}, W_1) = W_1 \mathbf{x} \qquad W_1 \in \mathbb{R}^{m_1 \times n}$$

$$\mathbf{x}_3 = \mathbf{f}_2(\mathbf{x}_2, \theta) = \text{relu}(\mathbf{x}_2)$$

$$\mathbf{x}_4 = \mathbf{f}_3(\mathbf{x}_3, W_3) = W_3 \mathbf{x}_3 \qquad W_3 \in \mathbb{R}^{1 \times m_3}$$

$$\mathbf{o} = \mathbf{f}_4(\mathbf{x}_4, \mathbf{y}) = \frac{1}{2} \|\mathbf{x}_4 - \mathbf{y}\|^2$$

Feedforward network example



- Applying the chain rule once again we have

$$\begin{aligned}\frac{\partial \mathbf{o}}{\partial \theta_4} \\ \frac{\partial \mathbf{o}}{\partial \theta_3} &= \frac{\partial \mathbf{o}}{\partial \mathbf{x}_4} \frac{\partial \mathbf{x}_4}{\partial \theta_3} \\ \frac{\partial \mathbf{o}}{\partial \theta_2} &= \frac{\partial \mathbf{o}}{\partial \mathbf{x}_4} \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3} \frac{\partial \mathbf{x}_3}{\partial \theta_2} \\ &\vdots\end{aligned}$$

- Apart from the last multiplication, the Jacobians $\frac{\partial \mathbf{o}}{\partial \mathbf{x}_k}$ and $\frac{\partial \mathbf{o}}{\partial \theta_k}$ share the same computations!

Backprop for feedforward networks

Algorithm 3 Compute $\mathbf{o} = \mathbf{f}(\mathbf{x}, \theta_1, \dots, \theta_K)$ and its Jacobians.

```
1: Input:  $\mathbf{x} \in \mathbb{R}^n, \theta_1, \dots, \theta_K$ 
2:  $\mathbf{x}_1 \leftarrow \mathbf{x}$ 
3:  $\mathbf{u}_i \leftarrow \mathbf{e}_i \in \mathbb{R}^m \quad i \in \{1, \dots, m\}$ 
4: for  $k = 1$  to  $K$  do
5:    $\mathbf{x}_{k+1} \leftarrow \mathbf{f}_k(\mathbf{x}_k, \theta_k)$ 
6: end for
7: for  $k = K$  to  $1$  do
8:    $\mathbf{j}_{i,k} \leftarrow \mathbf{u}_i^\top \frac{\partial \mathbf{f}_k(\mathbf{x}_k, \theta_k)}{\partial \theta_k} \quad i \in \{1, \dots, m\}$ 
9:    $\mathbf{u}_i \leftarrow \mathbf{u}_i^\top \frac{\partial \mathbf{f}_k(\mathbf{x}_k, \theta_k)}{\partial \mathbf{x}_k} \quad i \in \{1, \dots, m\}$ 
10: end for
11: Returns:  $\mathbf{o} = \mathbf{x}_{K+1}, \left[ \frac{\partial \mathbf{o}}{\partial \mathbf{x}} \right]_{i,:} = \mathbf{u}_i, \left[ \frac{\partial \mathbf{o}}{\partial \theta_k} \right]_{i,:} = \mathbf{j}_{i,k} \quad i \in \{1, \dots, m\}, k \in \{1, \dots, K\}$ 
```

Examples of VJPs

Let $W \in \mathbb{R}^{a \times b}$, $u \in \mathbb{R}^a$, $x \in \mathbb{R}^b$.

- $\mathbf{f}(x) = g(x)$ (element-wise)
 - \mathbf{f} maps \mathbb{R}^b to \mathbb{R}^b
 - $J_{\mathbf{f}}(x) = J_{\mathbf{f}}(x)^{\top} = \text{diag}(g'(x))$ maps \mathbb{R}^b to \mathbb{R}^b , i.e., $b \times b$ matrix
 - $u^{\top} J_{\mathbf{f}}(x) = J_{\mathbf{f}}(x)^{\top} u = u * g'(x) \in \mathbb{R}^b$, where $*$ means element-wise multiplication
- $\mathbf{f}(x) = Wx$
 - \mathbf{f} maps \mathbb{R}^b to \mathbb{R}^a
 - $J_{\mathbf{f}}(x) = W$ maps \mathbb{R}^b to \mathbb{R}^a , i.e., $a \times b$ matrix
 - $J_{\mathbf{f}}(x)^{\top} = W^{\top}$ maps \mathbb{R}^a to \mathbb{R}^b , i.e., $b \times a$ matrix
 - $u^{\top} J_{\mathbf{f}}(x) = J_{\mathbf{f}}(x)^{\top} u = W^{\top} u \in \mathbb{R}^b$

Examples of VJPs

- $\mathbf{f}(W) = Wx$
 - \mathbf{f} maps $\mathbb{R}^{a \times b}$ to \mathbb{R}^a
 - $J_{\mathbf{f}}(W)$ maps $\mathbb{R}^{a \times b}$ to \mathbb{R}^a , i.e., $a \times (a \times b)$ matrix
 - $J_{\mathbf{f}}(W)^{\top}$ maps \mathbb{R}^a to $\mathbb{R}^{a \times b}$, i.e., $(a \times b) \times a$ matrix
 - $J_{\mathbf{f}}(W)^{\top} u = ux^{\top}$

VJPs make things easier when dealing with matrix or tensor outputs.

Summary: Forward vs. Backward

■ Forward

- Uses Jacobian vector products (JVPs)
- Each JVP call builds one column of the Jacobian
- Efficient for tall Jacobians ($m \geq n$)
- Need not store intermediate computations

■ Backward

- Uses vector Jacobian products (VJPs)
- Each VJP call builds one row of the Jacobian
- Efficient for wide matrices ($m \leq n$)
- Needs to store intermediate computations

Machine learning use case

- Most objectives in machine learning can be written in the form

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}) = \sum_{i=1}^N \ell_i(f_i(\mathbf{x}))$$

where $f: \mathbb{R}^n \rightarrow \mathbb{R}^M$ and $\ell_i: \mathbb{R}^M \rightarrow \mathbb{R}$.

- The minimization needs to be w.r.t. a scalar valued loss.
- This corresponds to the $m = 1$ setting, for which backward differentiation is more efficient.
- This explains the immense success of reverse autodiff in machine learning.

Outline

1 Numerical differentiation

2 Chain compositions

3 Computational graphs

4 Implementation

5 Advanced topics

6 Conclusion

Computational graph

$$f(x_1, x_2) = x_2 e^{x_1} \sqrt{x_1 + x_2 e^{x_1}}$$

■ Operations in topological order

$$x_3 = f_3(x_1) = e^{x_1}$$

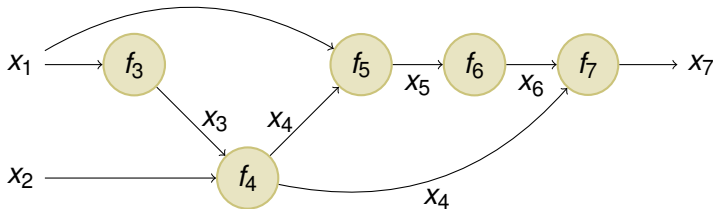
$$x_4 = f_4(x_2, x_3) = x_2 x_3$$

$$x_5 = f_5(x_1, x_4) = x_1 + x_4$$

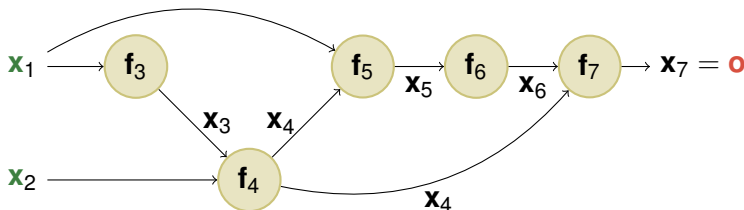
$$x_6 = f_6(x_5) = \sqrt{x_5}$$

$$x_7 = f_7(x_4, x_6) = x_4 x_6$$

■ Directed acyclic graph traversal



Forward differentiation example



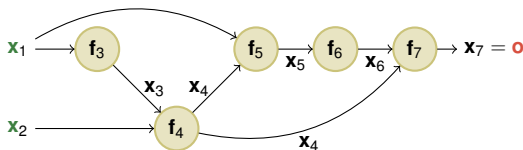
- \mathbf{x}_4 is influenced by \mathbf{x}_3 and \mathbf{x}_2 , therefore

$$\frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_1} = \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3} \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_1} + \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1}$$

- \mathbf{x}_7 is influenced by \mathbf{x}_4 and \mathbf{x}_6 , therefore

$$\frac{\partial \mathbf{x}_7}{\partial \mathbf{x}_1} = \frac{\partial \mathbf{x}_7}{\partial \mathbf{x}_4} \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_1} + \frac{\partial \mathbf{x}_7}{\partial \mathbf{x}_6} \frac{\partial \mathbf{x}_6}{\partial \mathbf{x}_1}$$

Forward differentiation example



- Recurse in topological order

$$\frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_1} = \text{Id}_n$$

$$\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_2} = \text{Id}_n$$

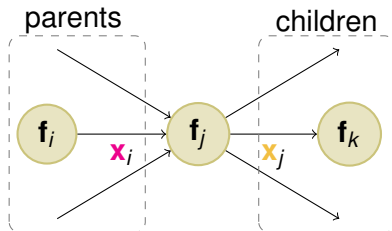
$$\frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_1} = \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_1} \frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_1}$$

$$\frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_1} = \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3} \frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_1} + \frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_2} \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1}$$

\vdots

- Everything can be computed in terms of JVPs

Forward differentiation

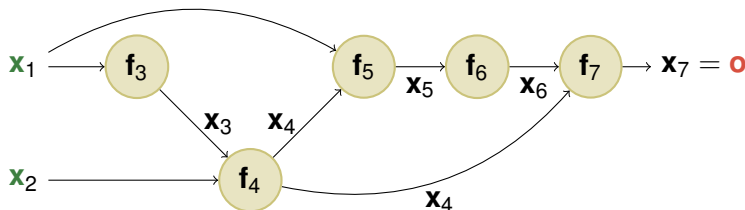


- In the general case, we have

$$\frac{\partial \mathbf{x}_j}{\partial \mathbf{x}_1} = \sum_{i \in \text{Parents}(j)} \frac{\partial \mathbf{x}_j}{\partial \mathbf{x}_i} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_1}$$

- $\frac{\partial \mathbf{x}_j}{\partial \mathbf{x}_i}$ is easy to compute as f_j is a direct function of \mathbf{x}_i .
- $\frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_1}$ is obtained from the previous iterations in topological order.

Backward differentiation example



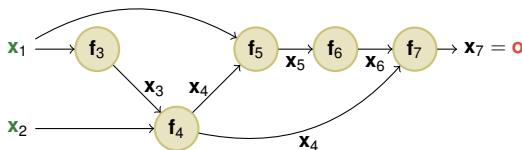
- x_5 influences only x_6 , therefore

$$\frac{\partial o}{\partial x_5} = \frac{\partial o}{\partial x_6} \frac{\partial x_6}{\partial x_5}$$

- x_4 influences x_5 and x_7 , therefore

$$\frac{\partial o}{\partial x_4} = \frac{\partial o}{\partial x_5} \frac{\partial x_5}{\partial x_4} + \frac{\partial o}{\partial x_7} \frac{\partial x_7}{\partial x_4}$$

Backward differentiation example



- Recurse in reverse topological order

$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}_7} = \frac{\partial \mathbf{x}_7}{\partial \mathbf{x}_7} = \text{Id}_m$$

$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}_6} = \frac{\partial \mathbf{o}}{\partial \mathbf{x}_7} \frac{\partial \mathbf{x}_7}{\partial \mathbf{x}_6}$$

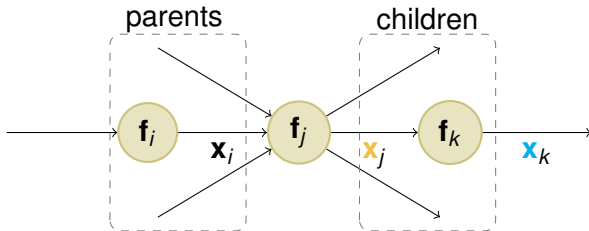
$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}_5} = \frac{\partial \mathbf{o}}{\partial \mathbf{x}_6} \frac{\partial \mathbf{x}_6}{\partial \mathbf{x}_5}$$

$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}_4} = \frac{\partial \mathbf{o}}{\partial \mathbf{x}_5} \frac{\partial \mathbf{x}_5}{\partial \mathbf{x}_4} + \frac{\partial \mathbf{o}}{\partial \mathbf{x}_7} \frac{\partial \mathbf{x}_7}{\partial \mathbf{x}_4}$$

⋮

- Everything can be computed in terms of VJPs

Backward differentiation



- In the general case, we have

$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}_j} = \sum_{k \in \text{Children}(j)} \frac{\partial \mathbf{o}}{\partial \mathbf{x}_k} \frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_j}$$

- $\frac{\partial \mathbf{o}}{\partial \mathbf{x}_k}$ is obtained from previous iterations (reverse topological order) and is known as “adjoint”.
- $\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_j}$ is easy to compute as f_k is a direct function of \mathbf{x}_j .

Outline

1 Numerical differentiation

2 Chain compositions

3 Computational graphs

4 Implementation

5 Advanced topics

6 Conclusion

Obtaining the computational graph

■ Ahead of time

- Read from source or abstract syntax tree (AST). Ex: [Tangent](#).
- API for composing primitive operations (the graph is fully built before the program is evaluated). Ex: Tensorflow.

■ Just in time

- Tracing: monitor the program execution (the graph is built while the program is being executed). Ex: Tensorflow Eager, [JAX](#), PyTorch.

```
import jax.numpy as jnp
from jax import grad
```

```
def add(a, b):
    return a + b
```

```
a = jnp.array([1, 2, 3])
b = jnp.array([4, 5, 6])
print(grad(add)(a, b))
```

Key components of an implementation

- VJP for all primitive operations
- Node class
- Topological sort
- Forward pass
- Backward pass

We will now briefly review each component using a rudimentary implementation (mine ;-)).

VJPs for primitive operations

```
def dot(x, W):  
    return np.dot(W, x)  
  
def dot_make_vjp(x, W):  
    def vjp(u):  
        return W.T.dot(u), np.outer(u, x)  
    return vjp
```

```
dot.make_vjp = dot_make_vjp
```

```
def add(a, b):  
    return a + b  
  
def add_make_vjp(a, b):  
    gprime = np.ones(len(a))  
  
    def vjp(u):  
        return u * gprime, u * gprime  
  
    return vjp
```

```
add.make_vjp = add_make_vjp
```

Node class

```
class Node(object):

    def __init__(self, value=None, func=None, parents=None, name="")
        # Value stored in the node.
        self.value = value
        # Function producing the node.
        self.func = func
        # Inputs to the function.
        self.parents = [] if parents is None else parents
        # Unique name of the node (for debugging and hashing).
        self.name = name
        # Gradient / Jacobian.
        self.grad = 0
        if not name:
            raise ValueError("Each node must have a unique name.")

    def __hash__(self):
        return hash(self.name)

    def __repr__(self):
        return "Node(%s)" % self.name
```

Topological sort

```
def dfs(node, visited):
    visited.add(node)
    for parent in node.parents:
        if not parent in visited:
            # Yield parent nodes first.
            yield from dfs(parent, visited)
        # And current node later.
    yield node

def topological_sort(end_node):
    visited = set()
    sorted_nodes = []

    # All non-visited nodes reachable from end_node.
    for node in dfs(end_node, visited):
        sorted_nodes.append(node)

    return sorted_nodes
```

Forward pass

```
def evaluate_dag(sorted_nodes):  
    for node in sorted_nodes:  
        if node.value is None:  
            values = [p.value for p in node.parents]  
            node.value = node.func(*values)  
    return sorted_nodes[-1].value
```

Backward pass

```
def backward_diff_dag(sorted_nodes):
    value = evaluate_dag(sorted_nodes)
    m = value.shape[0]  # Output size

    # Initialize recursion.
    sorted_nodes[-1].grad = np.eye(m)

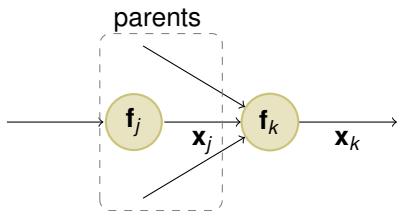
    for node_k in reversed(sorted_nodes):
        if not node_k.parents:
            # We reached a node without parents.
            continue

        # Values of the parent nodes.
        values = [p.value for p in node_k.parents]

        # Iterate over outputs.
        for i in range(m):
            # A list of size len(values) containing the vjps.
            vjps = node_k.func.make_vjp(*values)(node_k.grad[i])

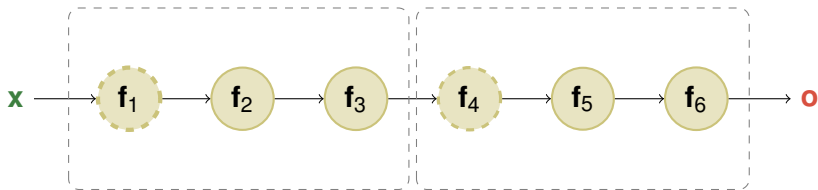
            for node_j, vjp in zip(node_k.parents, vjps):
                node_j.grad += vjp

    return sorted_nodes
```



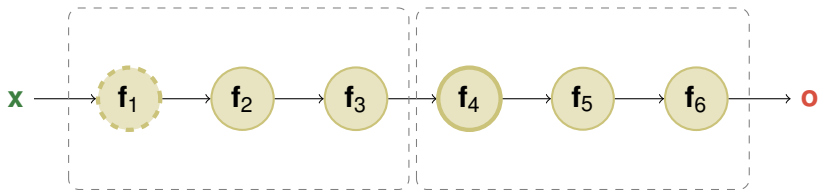
Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



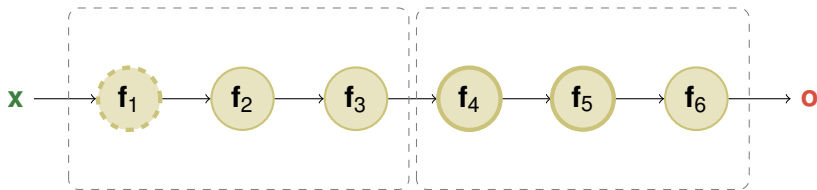
Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



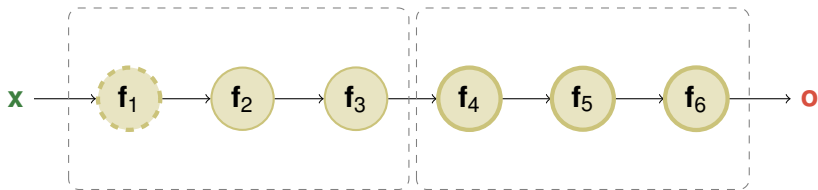
Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



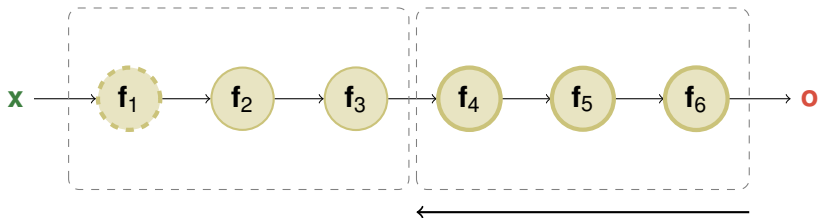
Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



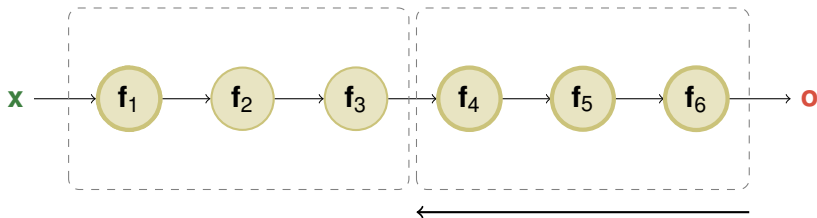
Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



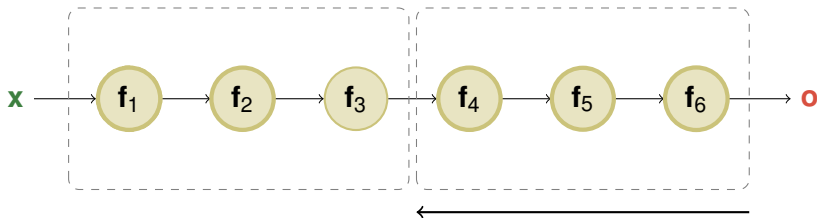
Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



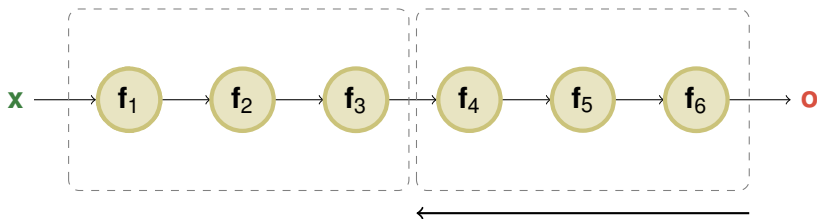
Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



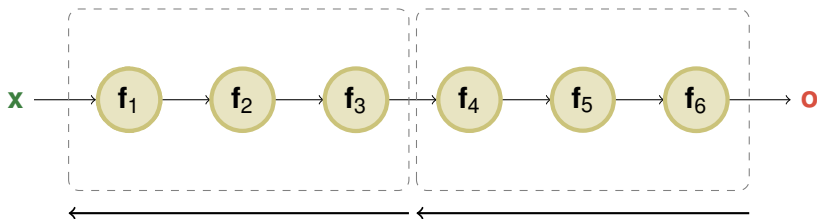
Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



Checkpointing

- During the forward pass, save computations at intermediate locations only (checkpoints).
- During the backward pass, recompute other locations on the fly, starting from the checkpoints.
- Tradeoff between memory and computation time.



JAX

- Automatic differentiation (**grad**)
- Just-in-time compilation (**jit**)
- NumPy and SciPy compatible
- Automatic vectorization (**vmap**)
- Code transformations are composable
- Actively developed by Google
- Gaining a lot of popularity among ML and science researchers



Outline

1 Numerical differentiation

2 Chain compositions

3 Computational graphs

4 Implementation

5 Advanced topics

6 Conclusion

Hessian

- The matrix gathering second-order derivatives

$$\nabla^2 f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Hessian vector product = gradient of directional derivative

$$\nabla^2 f(\mathbf{x}) \mathbf{v} = \nabla(\nabla f(\mathbf{x}) \cdot \mathbf{v})$$

- JAX supports fully closed tracing: we can “trace through tracing”

Recovering JVPs from VJPs

- Suppose we already have a VJP routine for computing $\mathbf{u}^\top \mathbf{J}_f(\mathbf{x})$
- By linearity we have

$$\frac{\partial \mathbf{u}^\top \mathbf{J}_f(\mathbf{x})}{\partial \mathbf{u}} = \mathbf{J}_f(\mathbf{x})^\top$$

- and therefore

$$\mathbf{v}^\top \frac{\partial \mathbf{u}^\top \mathbf{J}_f(\mathbf{x})}{\partial \mathbf{u}} = \mathbf{v}^\top \mathbf{J}_f(\mathbf{x})^\top = (\mathbf{J}_f(\mathbf{x}) \mathbf{v})^\top$$

- The VJP w.r.t. \mathbf{u} of the VJP w.r.t. \mathbf{x} is equal to the transpose of the JVP w.r.t. \mathbf{x} .
- The trick does not work in the other direction!

Differentiating min problems

- Consider the function

$$f(\theta) = \min_x E(x, \theta) = E(x^*(\theta), \theta)$$

- From Danskin's theorem (a.k.a. envelope theorem)

$$\nabla f(\theta) = \nabla_2 E(x^*(\theta), \theta)$$

where ∇_2 indicates the gradient w.r.t. the second argument.

- Informally, the theorem says that we can treat $x^*(\theta)$ as if it did not depend on θ .

Differentiating argmin problems

- Now, consider the function

$$x^*(\theta) = \underset{x}{\operatorname{argmin}} E(x, \theta)$$

$$f(\theta) = L(x^*(\theta), \theta)$$

- By the chain rule, we have

$$\nabla f(\theta) = (J x^*(\theta))^{\top} \nabla_1 L(x^*(\theta), \theta) + \nabla_2 L(x^*(\theta), \theta)$$

- How to compute $J x^*(\theta) = \frac{\partial x^*(\theta)}{\partial \theta}$?

Fixed points

- Consider the following fixed point iteration

$$x^*(\theta) = g(x^*(\theta), \theta) \Leftrightarrow h(x^*(\theta), \theta) = 0$$

where $h(x, \theta) = x - g(x, \theta)$

- By the implicit function theorem

$$J x^*(\theta) = -(J_1 h(x^*(\theta), \theta))^{-1} J_2 h(x^*(\theta), \theta)$$

where J_1 and J_2 are the Jacobians w.r.t. the 1st and 2nd variables

Differentiating argmin problems

- Recall that

$$x^*(\theta) = \underset{x}{\operatorname{argmin}} E(x, \theta)$$

- We have the fixed point iteration (gradient descent)

$$x^*(\theta) = x^*(\theta) - \nabla_1 E(x^*(\theta), \theta)$$

- Choosing $h(x, \theta) = \nabla_1 E(x, \theta)$, we get

$$\begin{aligned} J x^*(\theta) &= -(J_1 \nabla_1 E(x^*(\theta), \theta))^{-1} J_2 \nabla_1 E(x^*(\theta), \theta) \\ &= -(\nabla_1^2 E(x^*(\theta), \theta))^{-1} J_2 \nabla_1 E(x^*(\theta), \theta) \end{aligned}$$

- In practice, we need to replace $x^*(\theta)$ by an approximate solution.

Differentiating argmin problems

- Example: hyper-parameter optimization for ridge regression

$$E(x, \theta) = \frac{1}{2} \|Ax - b\|^2 + \frac{\theta}{2} \|x\|^2 \in \mathbb{R}$$

$$\nabla_1 E(x, \theta) = A^\top (Ax - b) + \theta x \in \mathbb{R}^d$$

$$\nabla_1^2 E(x, \theta) = A^\top A + \theta I \in \mathbb{R}^{d \times d}$$

$$J_2 \nabla_1 E(x, \theta) = x \in \mathbb{R}^{d \times 1}$$

- $J x^*(\theta)$ is therefore obtained by solving the following linear system

$$(A^\top A + \theta I)[J x^*(\theta)] = -x$$

Differentiating argmin problems

- An alternative idea to obtain $J x^*(\theta)$ is to to backpropagate through gradient descent:

$$x^{t+1}(\theta) = x^t(\theta) - \eta_t \nabla_1 E(x^t(\theta), \theta)$$

- No longer needs to solve a linear system...
- ...but needs to store intermediate iterates $x^t(\theta)$ or checkpoints
- Possibility to use **truncated backpropagation**
- Possibility to use **reversible dynamics** in some cases

Inference in graphical models

- Gibbs distribution

$$\mathbb{P}(Y = y; \theta) \propto \exp(y \cdot \theta)$$

where $y \in \mathcal{Y} \subset \{0, 1\}^n$

- Log-partition function

$$f(\theta) = \log \sum_{y \in \mathcal{Y}} \exp(y \cdot \theta)$$

- **Fact.**

$$(\mathbb{P}(Y_i = 1; \theta))_{i=1}^n = \mathbb{E}[Y] = \nabla f(\theta)$$

- If we know how to compute $f(\theta)$, we can get expectations / marginal probabilities by autodiff! Recovers forward-backward algorithms as special case. For a proof, see e.g. this [paper](#).

Outline

1 Numerical differentiation

2 Chain compositions

3 Computational graphs

4 Implementation

5 Advanced topics

6 Conclusion

Summary

- Automatic differentiation is one of the keys that enabled the deep learning “revolution”.
- Backward / reverse differentiation is more efficient when the function has more inputs than outputs.
- Which is the de-facto setting in machine learning!
- Even if you use Tensorflow / JAX / PyTorch, implementing a rudimentary autodiff library is a very good exercise.

References

The following tutorials have been a great inspiration:

- Automatic Differentiation, Matthew Johnson, Deep Learning Summer School Montreal, 2017.
- Differential programming, Gabriel Peyré, Mathematical Coffees, 2018.

Two minimalist implementations of autodiff:

- [Autodidact](#), by Matthew Johnson.
- [Micrograd](#), by Andrej Karpathy.