RelationalAI

RelationalAD

Mahmoud Abo Khamis
joint work with Hung Q. Ngo, Ryan Curtin, Mathieu Huot

relationalAI

What is RelAD?

RelAD: What is the Rel language?

- **Declarative**
- **Multipurpose**
  - Logic
  - Database queries
  - Linear algebra
    - Tensor computation
  - Machine learning
    - Feature extraction
    - Modeling, inference, prediction…
  - Mathematical Optimization
  - Statistics
    - Probabilistic programming
  - …
- See relational.ai

relationalAI

RelAD: Rel Core Syntax

- A (vast) generalization of *Datalog* with agg/neg..
- A Rel program is a collection of rules
    - def Q(x, y, ...) = φ(x , y, ...)
- A Formula φ(x , y, ...) defines a ***relation*** over vars {x, y, ...}
- Each formula φ(x , y, ...) could be
    - A materialized atom, e.g. R(x, y, ...)
    - A native, e.g. x + y = z or x > y
    - A conjunction/disjunction of formulas, e.g. $\psi_1(x, \ldots) \wedge \psi_2(x, \ldots)$
    - A negation, e.g. ¬ ψ(x, y, ...)
    - ∃ or ∀ , e.g. ∃ z : ψ(x, y, z, ...)
    - A sum/reduce, e.g. sum[z, t: ψ (x, z, t, ...)](y)
    - An FFI, e.g. some-external-function(ψ)
- See docs.relational.ai/rel/primer/basic-syntax

RelAD: Rel Examples

- Triangle counting in a graph E

```
def Q = count[a, b, c: E(a, b) and E(b, c) and E(c, a)]
```

- Matrix multiplication $C = AB$

```
def C(i, j, v) = sum[
    k, v1, v2, v3:  A(i, k, v1) and B(k, j, v2) and v1*v2=v3](v)
def C[i, j] = sum[k: A[i, k] * B[k, j]]
```

- $J = \|Ax - b\|_2^2$

```
def J = sum[i : (sum[j : A[i, j] * x[j]] - b[i])^2]
```

RelAD: Differentiation (What We Learned in College)

$$\frac{\partial \boldsymbol{b}^\top \boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{c}}{\partial \boldsymbol{X}} = \boldsymbol{X}(\boldsymbol{b}\boldsymbol{c}^\top + \boldsymbol{c}\boldsymbol{b}^\top)$$

$$\frac{\partial \log |\boldsymbol{A}|}{\partial \boldsymbol{A}} = (\boldsymbol{A}^{-1})^\top$$

$$\frac{\partial \operatorname{tr}(\boldsymbol{B}\boldsymbol{A})}{\partial \boldsymbol{A}} = \boldsymbol{B}^\top$$

$$\frac{\partial \boldsymbol{b}^\top \boldsymbol{X}^\top \boldsymbol{D} \boldsymbol{X} \boldsymbol{c}}{\partial \boldsymbol{X}} = \boldsymbol{D}^\top \boldsymbol{X} \boldsymbol{b}\boldsymbol{c}^\top + \boldsymbol{D}\boldsymbol{X}\boldsymbol{c}\boldsymbol{b}^\top$$

$$\frac{\partial (\boldsymbol{X}\boldsymbol{b}+\boldsymbol{c})^\top \boldsymbol{D} (\boldsymbol{X}\boldsymbol{b}+\boldsymbol{c})}{\partial \boldsymbol{X}} = (\boldsymbol{D}+\boldsymbol{D}^\top)(\boldsymbol{X}\boldsymbol{b}+\boldsymbol{c})\boldsymbol{b}^\top$$

RelAD: What is Differentiation Used For?

- Most of modern machine learning
- Traditional optimization
- Physics

RelAD: Automatic Differentiation

- Input: a **program** computing a function *f*
- Output: a **program** computing *df*

What is a program?

- A neural network
- Imperative program in C++, Haskell, etc
- A **Rel program**! (even with recursion)

Some Existing AutoDiff Frameworks

- TensorFlow (autodiff)
- PyTorch (autograd)
- NumPy (JAX)
- Geno

They operate on **Einsum notations** to construct complex functions of tensors

relationalAI

Digression: Einstein Notation

Existing AutoDiff frameworks operate on (network of) *Einsum rules*, e.g.

$$U_{i,j,k} = \sum_{l,m} R_{i,l,m} \cdot S_{j,k,l} \cdot T_{i,k,l}$$

```
def U[i, j, k] = sum[l m v : v = R[i,l,m] * S[j,k,l] * T[i,k,l]]
```

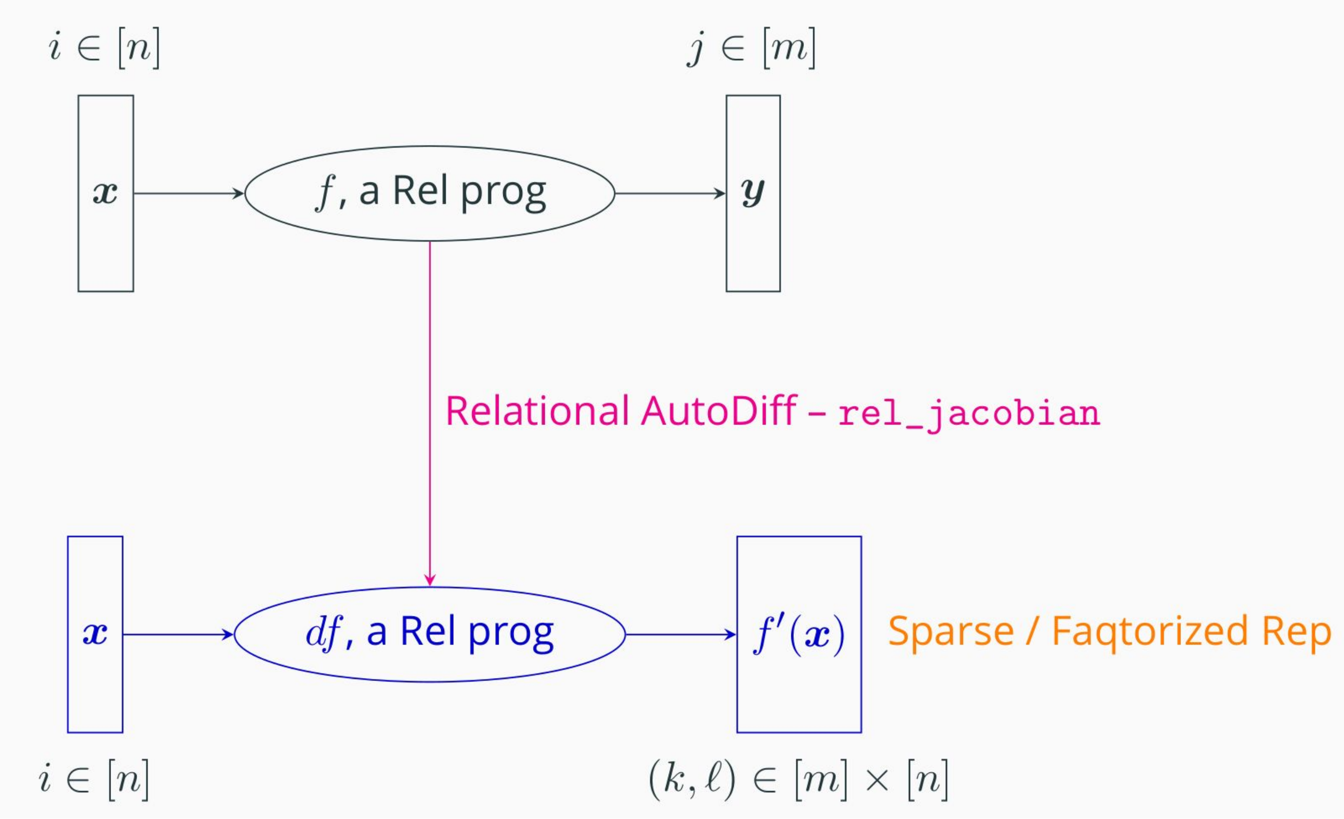
(For a gentle intro, see "Einsum is all you need")

Rel is **much** more general, e.g.

```
def U[i, j, k] = sum[l m v : v = R[i,l,m] * S[j,k,l] * T[i,k,l] and
                                 exists(x : i^2 + x <= k and x > 10)
                    ]
```

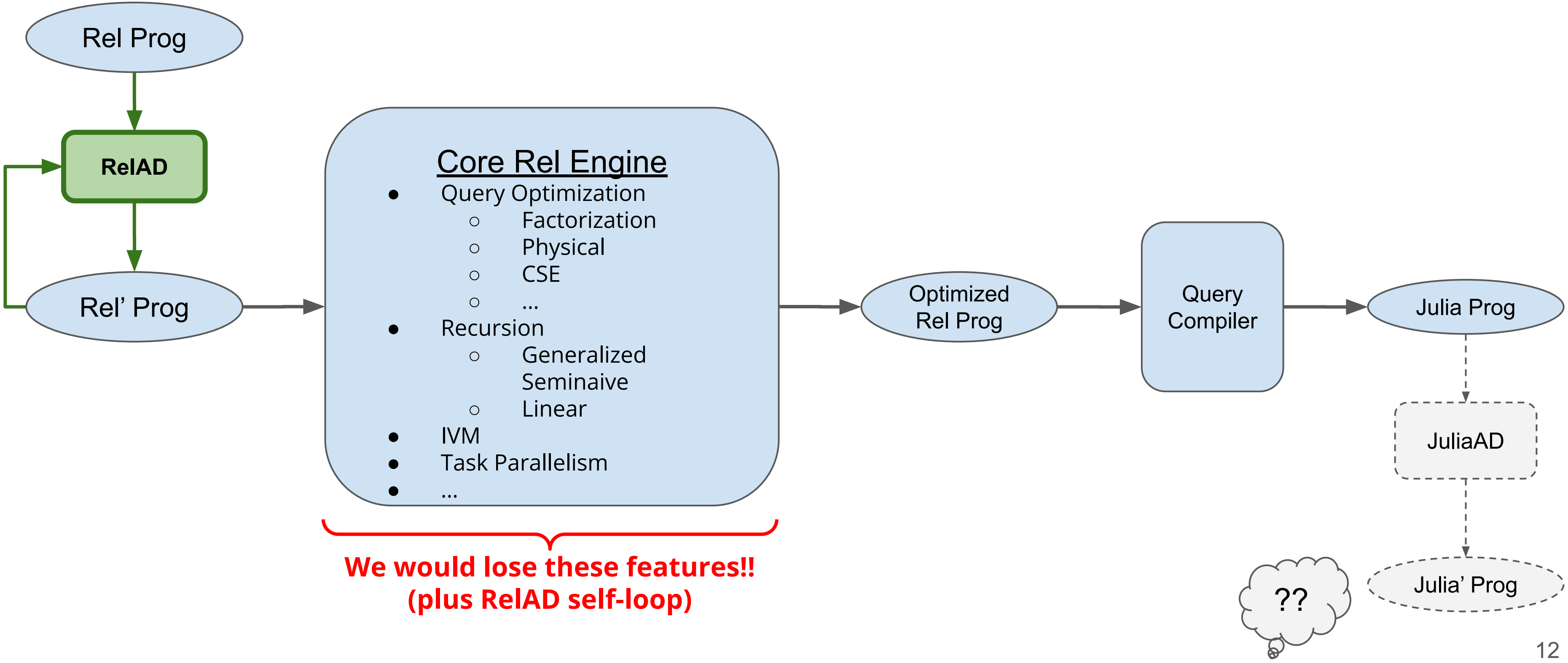
- Our keys are of arbitrary types
- Tensors can be (very) sparse
- Additional logic is arbitrary
- Worst-case optimal join + semantic optimization + IVM

RelAD: Relational AutoDiff



relationalAI

Why RelAD, not JuliaAD?



RelAD: Example

Consider the Rel program

```
def J = sum[i j : x[i] * A[i, j] * x[j]]
```

$$J = x^T A x$$

We need to give an extra hint to specify how to interpret it as a **function**

```
def ∇ = jacobian[J, x]
```

$$\nabla = \frac{\partial J}{\partial x}$$

This says that

- the above program defines a function **f**:`x` $\rightarrow$ `J`, and
- we are interested in **f'**, which we now call ∇.

RelAD: Example (Cont.)

RelAD rewrites this program into

```
def J = sum[i j : x[i] * A[i, j] * x[j]]
```

$$J = x^T A x$$

```
def ∇1[i] = sum[j v : x[i] = _ and v = A[i, j] * x[j]]
```

$$\nabla_1 = A x$$

```
def ∇2[j] = sum[i v : v = x[i] * A[i, j] and x[j] = _]
```

$$\nabla_2 = A^T x$$

```
def ∇[i] = merge_sum[∇1[i], ∇2[i]]
```

$$\nabla = \nabla_1 + \nabla_2$$

RelAD: Example (Cont.)

Now take this new program and add the hint

```
def H = jacobian[∇, x]
```

$$H = \frac{\partial \nabla}{\partial x}$$

The above says

- interpret the new program as another function **g**`:x`$\rightarrow$ ∇, and
- compute **g'** which we now call `H`

RelAD: Example (Cont.)

RelAD rewrites

```
def H = jacobian[∇, x]
```

$$H = \frac{\partial \nabla}{\partial x}$$

into

```
def H1[i, j] = sum[v : x[i] = _ and v = A[i, j] and x[j] = _]
```

$$H_1 = A$$

```
def H2[j, i] = sum[v : x[i] = _ and v = A[i, j] and x[j] = _]
```

$$H_2 = A^T$$

```
def H[i, j] = merge_sum[H1[i, j], H2[i, j]]
```

$$H = H_1 + H_2$$

relationalAI

Interface: “jacobian” Higher-order Native

- Given a Rel program **P** defining (among other things) two relations **A** and **B** where
  - $\mathbf{A[k_1, k_2, \ldots, k_m] = v}$ has **m >= 0** keys and one value **v** whose type is **Float**
  - $\mathbf{B[l_1, l_2, \ldots, l_n] = w}$ has **n >= 0** keys and one value **w** whose type is **Float**
  - **B** may depend on **A** in any way: directly or indirectly through chains of other relations in **P**

- We can use the higher-order native **jacobian** to define a new relation **C**
  - **def C = jacobian[B, A]**
  - $\mathbf{C[l_1, l_2, \ldots, l_n, k_1, k_2, \ldots, k_m] = t}$ has **n+m** keys and one value **t** whose type is **Float**
  - $\mathbf{C[l_1, l_2, \ldots, l_n, k_1, k_2, \ldots, k_m] := \partial B[l_1, l_2, \ldots, l_n] \;/\; \partial A[k_1, k_2, \ldots, k_m]}$

- RelAD later desugars **jacobian** into lower-order Rel

Under the Hood

How Do We Do It?

How does it work?

To derive a single SumProductNode (SPN) **s**, we

- analyze dependencies among atoms of **s**
- construct a dependency DAG
- do backpropagation

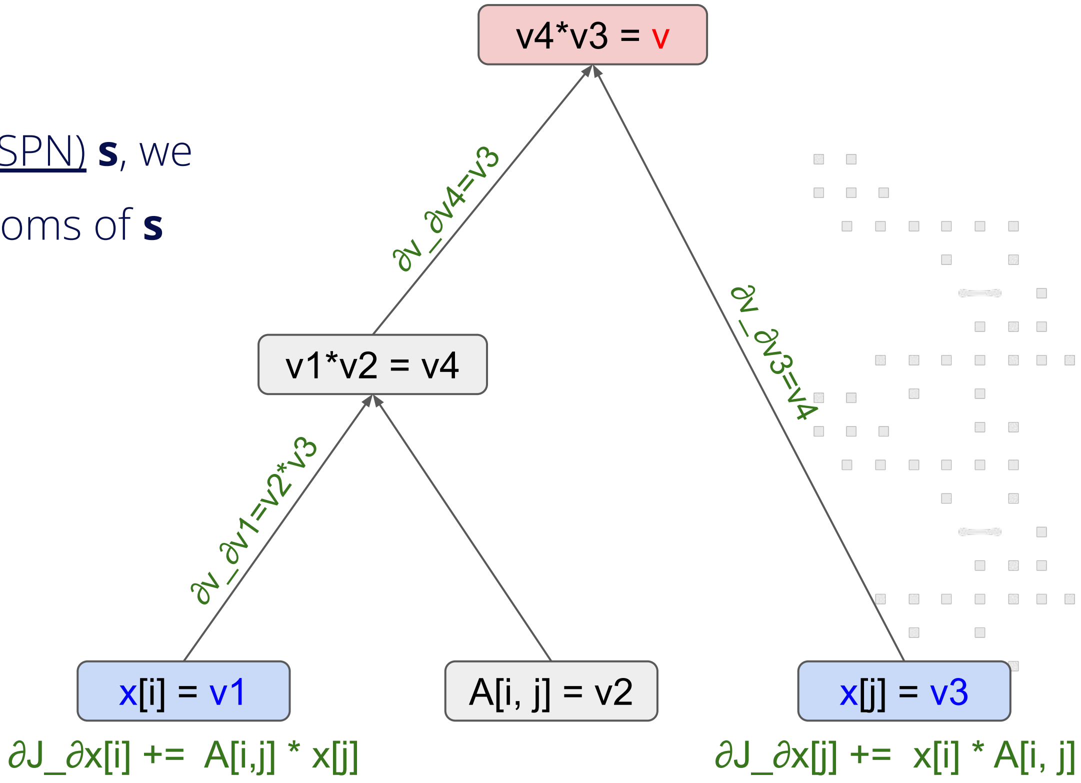
Example:

```
def J = sum[i j : x[i] * A[i, j] * x[j]]
```

$$J = x^T A x$$

```
def ∇ = jacobian[J, x]
```

$$\nabla = \frac{\partial J}{\partial x}$$



relationalAI

How does it work?

To derive <u>an entire Rel program</u> **P** consisting of many SPNs, we

- analyze dependencies among SPNs of **P**
- construct a dependency DAG
- do **backpropagation**
  - More generally, Jacobian accumulation

More Examples

In ML

More Matrix Calculus Examples

From our test suite:

$$\nabla (\boldsymbol{B}\boldsymbol{x}+\boldsymbol{b})^\top \boldsymbol{C}(\boldsymbol{D}\boldsymbol{x}+\boldsymbol{d}) = \boldsymbol{B}^\top \boldsymbol{C}(\boldsymbol{D}\boldsymbol{x}+\boldsymbol{d}) + \boldsymbol{D}^\top \boldsymbol{C}^\top (\boldsymbol{B}\boldsymbol{x}+\boldsymbol{b})$$

$$\frac{\partial \boldsymbol{b}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{c}}{\partial \boldsymbol{X}} = \boldsymbol{X}(\boldsymbol{b}\boldsymbol{c}^\top + \boldsymbol{c}\boldsymbol{b}^\top)$$

$$\frac{\partial \operatorname{tr}(\boldsymbol{B}\boldsymbol{A})}{\partial \boldsymbol{A}} = \boldsymbol{B}^\top$$

$$\frac{\partial \boldsymbol{b}^\top \boldsymbol{X}^\top \boldsymbol{D}\boldsymbol{X}\boldsymbol{c}}{\partial \boldsymbol{X}} = \boldsymbol{D}^\top \boldsymbol{X}\boldsymbol{b}\boldsymbol{c}^\top + \boldsymbol{D}\boldsymbol{X}\boldsymbol{c}\boldsymbol{b}^\top$$

$$\frac{\partial (\boldsymbol{X}\boldsymbol{b}+\boldsymbol{c})^\top \boldsymbol{D}(\boldsymbol{X}\boldsymbol{b}+\boldsymbol{c})}{\partial \boldsymbol{X}} = (\boldsymbol{D}+\boldsymbol{D}^\top)(\boldsymbol{X}\boldsymbol{b}+\boldsymbol{c})\boldsymbol{b}^\top$$

More Matrix Calculus Examples

- RelAD can now mimic matrixcalculus.org
  - We just need to a translator from Latex to Rel
  - and another one from Rel to Latex

$$f = \mathrm{sum}(\log(\exp((-y) \odot (X \cdot w)) + \mathrm{vector}(1)))$$

matrixcalculus.org

$$\frac{\partial f}{\partial X} = -t_0 \odot y \oslash (\mathrm{vector}(1) + t_0) \cdot w^\top$$

$$t_0 = \exp(-y \odot (X \cdot w))$$

Latex ⇒ Rel (currently missing)

$$f = \mathrm{sum}(\log(\exp((-y) \odot (X \cdot w)) + \mathrm{vector}(1)))$$

⇓

```
def z[i] = sum[j : -y[i] * X[i, j] * w[j]]
def f = sum[i : natural_log[natural_exp[z[i]] + 1.0]]
def ∇ = jacobian[f, X]
```

Rel ⇒ Rel (RelAD)

```
def z[i] = sum[j : -y[i] * X[i, j] * w[j]]
def f = sum[i : natural_log[natural_exp[z[i]] + 1.0]]
def ∇ = jacobian[f, X]
```

RelAD

```
def ∇[i, j] = v : X[i, j] = _ and v = -y[i] * w[j] * ∂f_∂z[i]
def ∂f_∂z[i] = natural_exp[z[i]] / (natural_exp[z[i]] + 1)
```

Rel ⇒ Latex (currently missing)

```
def ∇[i, j] = v : X[i, j] = _ and v = -y[i] * w[j] * ∂f_∂z[i]
def ∂f_∂z[i] = natural_exp[z[i]] / (natural_exp[z[i]] + 1)
```

$$\nabla = -(y \odot \frac{\partial f}{\partial z})w^T$$

$$\frac{\partial f}{\partial z} = \exp(z) \oslash (\exp(z) + \text{vector}(1))$$

ML Example: Multi-layer Neural Network

Consider a neural network with two layers, ReLU activations, and multiple outputs:

- **x** is the input vector, **t** is the target vector
- $\mathbf{W_1}$ is the weight matrix for the 1st layer
- $\mathbf{W_2}$ is the weight matrix for the 2nd layer
- Recall $\mathrm{ReLU}(x) := \max(x, 0)$

The network works as follows:

$$\begin{aligned} y_1 &= W_1 x \\ z_1 &= \mathrm{ReLU}(y_1) \\ y_2 &= W_2 z_1 \\ z_2 &= \mathrm{ReLU}(y_2) \\ J &= \|t - z_2\|_2^2 \end{aligned}$$

relationalAI

ML Example: Multi-layer Neural Network (cont.)

Input Rel Program:

$$y_1 = W_1 x$$

```
def y1[i] = sum[j : W1[i, j] * x[j]]
```

$$z_1 = \mathrm{ReLU}(y_1)$$

```
def z1(i, v) = y1[i] = v and v >= 0
```

```
def z1(i, v)=exists(u : y1[i]=u and u<0 and v=0)
```

$$y_2 = W_2 z_1$$

```
def y2[i] = sum[j : W2[i, j] * z1[j]]
```

$$z_2 = \mathrm{ReLU}(y_2)$$

```
def z2(i, v) = y2[i] = v and v >= 0
```

```
def z2(i, v)=exists(u : y2[i]=u and u<0 and v=0)
```

$$J = \|t - z_2\|_2^2$$

```
def J = sum[i : (t[i] - z2[i]) ^ 2]
```

$$\nabla_{W1} = \frac{\partial J}{\partial W_1}$$

```
def ∇_W1 = jacobian[J, W1]
```

$$\nabla_{W2} = \frac{\partial J}{\partial W_2}$$

```
def ∇_W2 = jacobian[J, W2]
```

ML Example: Multi-layer Neural Network (cont.)

RelAD output:

```
def ∂J_∂z2[i] = 2 * (z2[i] - t[i])
```

$$\frac{\partial J}{\partial z_2} = 2(z_2 - t)$$

```
def ∂J_∂y2[i] = sum[v: y2[i] >=0 and ∂J_∂z2[i] = v]
```

$$\frac{\partial J}{\partial y_2} = \frac{\partial J}{\partial z_2} \odot 1_{y_2 \geq 0}$$

```
def ∇_W2[i, j] = sum[v: W2[i, j] = _ and v = ∂J_∂y2[i] * z1[j]]
```

$$\nabla_{W2} = \frac{\partial J}{\partial y_2} z_1^T$$

```
def ∂J_∂z1[j] = sum[i v: z1[j] = _ and v = W2[i, j] * ∂J_∂y2[i]]
```

$$\frac{\partial J}{\partial z_1} = W_2^T \frac{\partial J}{\partial y_2}$$

```
def ∂J_∂y1[i] = sum[v: y1[i] >=0 and ∂J_∂z1[i] = v]
```

$$\frac{\partial J}{\partial y_1} = \frac{\partial J}{\partial z_1} \odot 1_{y_1 \geq 0}$$

```
def ∇_W1[i, j] = sum[v: W1[i, j] = _ and v = ∂J_∂y1[i] * x[j]]
```

$$\nabla_{W1} = \frac{\partial J}{\partial y_1} x^T$$

Recursive Example

$$\frac{\partial \log(|\det(A)|)}{\partial A} = (A^{-1})^T$$

- The determinant is not directly available in Rel
- But it can be computed **recursively** using a Gram–Schmidt process

Recursive Example: (cont.)

- Given an (n X n)-matrix A = [A_1, A_2, ..., A_n]
- Compute orthogonal vectors O = [O_1, O_2, ..., O_n]

$$O_j = A_j - \sum_{k<j} \frac{A_j^T O_k}{||O_k||_2^2} O_k$$

```
def O(i, j, v) = (j = 1 and v = A[i, 1])
def O(i, j, v) = (v = A[i, j] - sum[k s :
        k < j and s = prod_A_O[j, k] / sqr_norm_O[k] * O[i, k]])
def prod_A_O[j, k] = sum[l : A[l, j] * O[l, k]]
def sqr_norm_O[k] = sum[l : O[l, k] ^ 2]
```

Recursive Example: (cont.)

- By determinant properties, $\det(A) = \det(O)$
- Because O is an orthogonal basis

$$|\det(O)| = ||O_1||_2 \times ||O_2||_2 \times \cdots \times ||O_n||_2$$
$$\log|\det(O)| = \log||O_1||_2 + \log||O_2||_2 + \cdots + \log||O_n||_2$$

```
def log_det = sum[k : natural_log[sqr_norm_O[k]] / 2]
def ∇ = jacobian[log_det, A]
```

relationalAI

RelAD and Optimization

Optimization with Rel stdlib?

Example: Linear regression with Gradient-descent

- Input
  - ($N \times d$)-matrix X where each row is a data point
  - N-vector t of corresponding target responses
- Output
  - d-vector w^* of optimal model parameters

$$w^* = \mathrm{argmin}_w ||Xw - t||_2^2$$

relationalAI

Example: Linear regression with Gradient-descent

```
def MAX_K = 10000     // maximum number of iterations
def α = 0.01          // learning rate (fixed)
def w(k, i, v) = k = 0 and range(1, d, 1, i) and v = 0.0

def y[k, i] = sum[j : X[i, j] * w[k, j]]

def J[k] = sum[i : (y[k, i] - t[i]) ^ 2]

def ∇ = jacobian[J, w]

def w(k, i, v) = k <= MAX_K and
   v = w[k-1, i] - α * ∇[k-1, k-1, i]
```

$$w_0 \leftarrow 0$$

$$y_k \leftarrow X w_k$$

$$J_k \leftarrow \|y_k - t\|_2^2$$

$$\nabla_k \leftarrow \frac{\partial J_k}{\partial w_k}$$

$$w_k \leftarrow w_{k-1} - \alpha \nabla_{k-1}$$

relationalAI

Open Issues

Incorporating ICs: Key alignment

Consider the program

```
def J = sum[i : b[i] * x[i]]
def ∇ = jacobian[J, x]
```

$$J = b^T x$$

$$\nabla = \frac{\partial J}{\partial x} = b$$

Currently RelAD produces the following rule for ∇:

```
def ∇[i] = v : v = b[i] and x[i] = _
```

The **extra part** is due to the fact that RelAD doesn't know that the keys of `b` and `x` are aligned

- Even if there was an IC saying they are, our RelAD currently has no mechanism to utilize it

Incorporating ICs: Matrix Symmetry

Consider the example from before

```
def J = sum[i j : x[i] * A[i, j] * x[j]]
```

$$J = x^T A x$$

If A was symmetric, then

$$\nabla = 2Ax$$

However currently our RelAD output has no mechanism to utilize this symmetry (even if it was encoded as an IC)

```
def ∇1[i] = sum[j v : x[i] = _ and v = A[i, j] * x[j]]
def ∇2[j] = sum[i v : v = x[i] * A[i, j] and x[j] = _]
def ∇[i] = merge_sum[∇1[i], ∇2[i]]
```

$$\nabla_1 = Ax$$
$$\nabla_2 = A^T x$$
$$\nabla = \nabla_1 + \nabla_2$$

Multi-headed Rules

Consider the example

```
// J = xᵀAx
def J = sum[i j : x[i] * A[i, j] * x[j]]
def ∇ = jacobian[J, x]
```

Instead of having two rules for ∇

```
∇[i] += A[i, j] * x[j]
∇[j] += A[i, j] * x[i]
```

it is faster to have one with two heads

```
∇[i] += v2*v3, ∇[j] + = v1*v2   ←   x[i] = v1, A[i, j] = v2, x[j] = v3
```

relationalAI

Other Issues

- Usage
  - Integration with ML work
  - Integration into existential second order (ESO) work (for optimization)
- Performance
  - Dense-tensor support
  - Mapping to BLAS / GPUs / ....
  - XY-Stratification to speed up GD (*Joint work with Amir Shaikhha*)

What's Out There

Some References

- Naumann, Uwe (April 2008). "Optimal Jacobian accumulation is NP-complete". Mathematical Programming. 112 (2): 427–441.
- Baydin, Atilim Gunes; Pearlmutter, Barak; Radul, Alexey Andreyevich; Siskind, Jeffrey (2018). "Automatic differentiation in machine learning: a survey". Journal of Machine Learning Research. 18: 1–43.
- Soeren Laue, Matthias Mitterreiter, and Joachim Giesen. GENO -- GENeric Optimization for Classical Machine Learning, NeurIPS 2019.
- Sören Laue, Matthias Mitterreiter, Joachim Giesen: A Simple and Efficient Tensor Calculus. AAAI 2020: 4527-4534

RelationalAI

Thank You!

Any Questions/Comments?