MathOptAI.jl

Robert Parker

Oscar Dowson, Nicole LoGiudice, Manuel Garcia, Kaarthik Sundar, Russell Bent

JuMP-dev 2025

MathOptAI.jl

https://lanl-ansi.github.io/MathOptAI.jl/stable/

Mission

Embed machine learning predictors into a JuMP model.

Similar to

- OMLT
- gurobi-machinelearning
- PySCIPOpt-ML
- GAMSPy
- …

Problem class

$$\begin{aligned} \min\ & f_0(x, y) \\ & f_i(x, y) \in S_i \quad \forall i \\ & y = F(x) \end{aligned}$$

where F is a neural network/decision tree/logistic regression/…

Application Example

Security Constrained Optimal Power Flow. Parker et al. (2025)

Take a nonlinear program representing Optimal Power Flow

$$\begin{aligned}\min\ & f_0(x)\\ & f_i(x) \in S_i \quad \forall i\end{aligned}$$

We want to add $G(x) = 1$, where G is a classifier that returns 1 if a non-differentiable simulation shows that x is stable and 0 otherwise.

We cannot embed G directly, but we can train a surrogate, $G(x) \approx F(x)$, and then add

$$\begin{aligned}\min\ & f_0(x)\\ & f_i(x) \in S_i \quad \forall i\\ & y = F(x)\\ & y \geq 0.95\end{aligned}$$

Application Example

Two-stage stochastic programming. Dumouchelle et al (2022).

Take a two-stage stochastic program

$$\min\ f_0(x) + E[V_2(x)]$$
$$f_i(x) \in S_i \quad \forall i$$

Replace the expected value function by a learned predictor

$$\min\ f_0(x) + y$$
$$f_i(x) \in S_i \quad \forall i$$
$$y = F(x)$$

Application Example

Bilevel optimization. Moreno-Palancas et al. (2025)

Take a bilevel program

$$
\begin{aligned}
\min\ & f_0(x, y) \\
& f_i(x, y) \in S_i \quad \forall i \\
& y \in \operatorname{argmin} V(x)
\end{aligned}
$$

Replace the inner optimization problem by a learned predictor

$$
\begin{aligned}
\min\ & f_0(x, y) \\
& f_i(x, y) \in S_i \quad \forall i \\
& y = F(x)
\end{aligned}
$$

Code Example

Embed a NN from Pytorch in JuMP

```python
#!/usr/bin/python3
import torch
from torch import nn
model = nn.Sequential(nn.Linear(10, 16), nn.ReLU(), nn.Linear(16, 2))
torch.save(model, "model.pt")
```

```julia
#!/usr/bin/julia
using JuMP, Ipopt, MathOptAI, PythonCall
model = Model(Ipopt.Optimizer)
@variable(model, 0 <= x[1:10] <= 1)
predictor = MathOptAI.PytorchModel("model.pt")
y, formulation = MathOptAI.add_predictor(model, predictor, x)
@constraint(model, y .>= 0.9)
```

Code Example

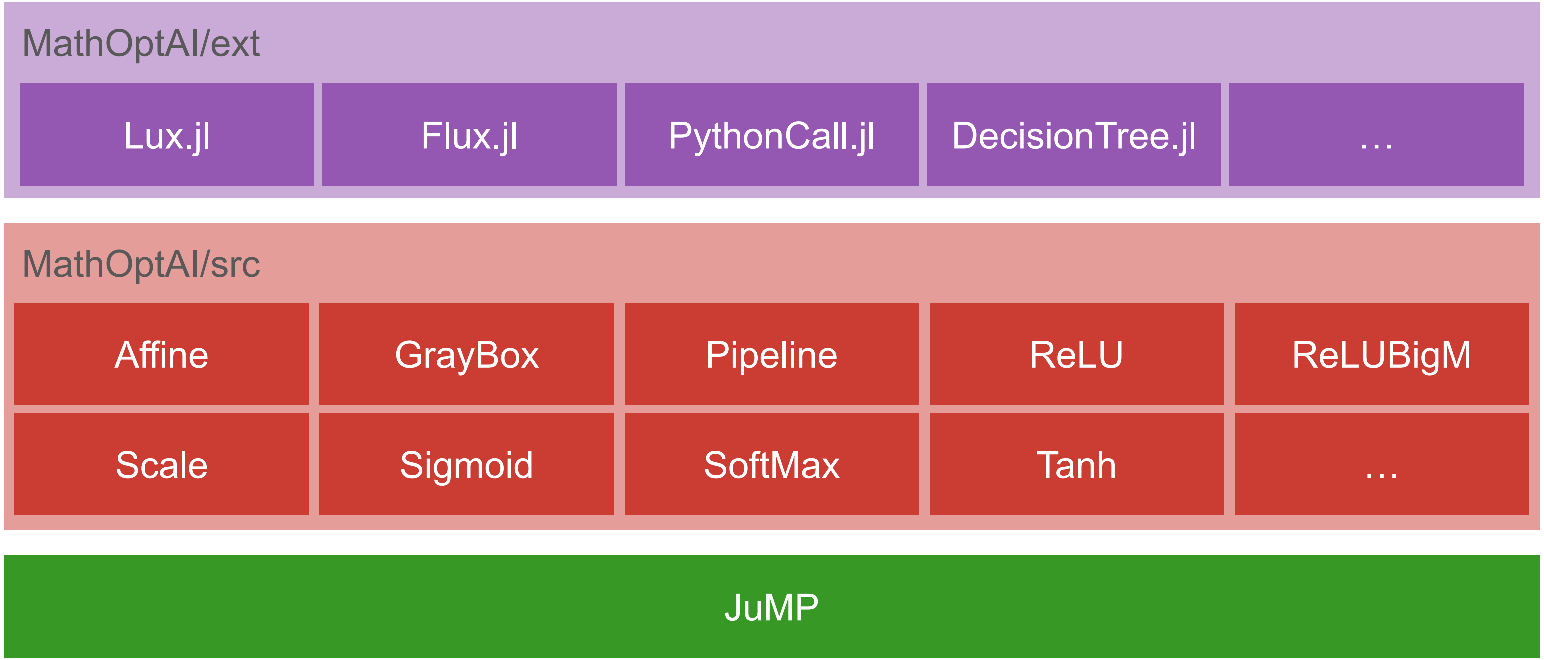
Embed a NN from Pytorch in JuMP

```python
#!/usr/bin/python3
import torch
from torch import nn
model = nn.Sequential(nn.Linear(10, 16), nn.ReLU(), nn.Linear(16, 2))
torch.save(model, "model.pt")
```

```julia
#!/usr/bin/julia
using JuMP, HiGHS, MathOptAI, PythonCall
model = Model(HiGHS.Optimizer)
@variable(model, 0 <= x[1:10] <= 1)
predictor = MathOptAI.PytorchModel("model.pt")
config = Dict(:ReLU => MathOptAI.ReLUSOS1())
y, formulation = MathOptAI.add_predictor(model, predictor, x; config)
@constraint(model, y .>= 0.9)
```

MathOptAI sits on top of JuMP

We implement many package extensions



AbstractPredictors and package extensions

The Affine predictor

```julia
# src/predictors/Affine.jl
struct Affine{T} <: AbstractPredictor
    A::Matrix{T}
    b::Vector{T}
end

function add_predictor(model::JuMP.AbstractModel, predictor::Affine, x::Vector)
    m = size(predictor.A, 1)
    y = JuMP.@variable(model, [1:m], base_name = "moai_Affine")
    cons = JuMP.@constraint(model, predictor.A * x .+ predictor.b .== y)
    return y, Formulation(predictor, y, cons)
end
```

AbstractPredictors and package extensions

The GLM package extension

```julia
# ext/MathOptAIGLMExt.jl
function MathOptAI.build_predictor(predictor::GLM.LinearModel)
    return MathOptAI.Affine(GLM.coef(predictor))
end

# src/MathOptAI.jl
function add_predictor(
    model::JuMP.AbstractModel,
    predictor::Any,
    x::Vector;
    kwargs...,
)
    inner_predictor = build_predictor(predictor; kwargs...)
    return add_predictor(model, inner_predictor, x)
end
```

Three-ways to formulate a problem

Each with a different trade-off

	Full-space	Reduced-space	Gray-box
Pros			
Cons			
Bottleneck			

Full-space

Add intermediate variables and constraints

```julia
using JuMP, MathOptAI
# y = ReLU(x) = max.(0, A * x + b)
predictor = MathOptAI.Pipeline(
    MathOptAI.Affine(A, b),
    MathOptAI.ReLU(),
)
model = Model()
@variable(model, x[1:n])
y, _ = MathOptAI.add_predictor(
    model,
    predictor,
    x,
)
```

```julia
using JuMP
model = Model()
@variables(model, begin
    x[1:n]
    tmp[1:m]
    y[1:m]
end)
@constraints(model, begin
    tmp == A * x + b
    y .== max.(0, tmp)
end)
```

Three-ways to formulate a problem

Each with a different trade-off

	Full-space	Reduced-space	Gray-box
Pros	Sparsity Solvers can exploit linearity		
Cons	Many extra variables and constraints		
Bottleneck	Computing linear system because of problem size		

Reduced-space

Use nested expressions

```julia
using JuMP, MathOptAI
# y = ReLU(x) = max.(0, A * x + b)
predictor = MathOptAI.Pipeline(
    MathOptAI.Affine(A, b),
    MathOptAI.ReLU(),
)
model = Model()
@variable(model, x[1:n])
y, _ = MathOptAI.add_predictor(
    model,
    predictor,
    x;
    reduced_space = true,
)
```

```julia
using JuMP
model = Model()
@variables(model, begin
    x[1:n]
end)
@expressions(model, begin
    tmp, A * x + b
    y, max.(0, tmp)
end)
```

Three-ways to formulate a problem

Each with a different trade-off

	Full-space	Reduced-space	Gray-box
Pros	Sparsity Solvers can exploit linearity	Fewer variables and constraints	
Cons	Many extra variables and constraints	Complicated dense expressions	
Bottleneck	Computing linear system because of problem size	Computing derivatives (JuMP’s AD does not do well at dense problems)	

Gray-box

Use external function evaluation

```julia
using JuMP, MathOptAI
# y = ReLU(x) = max.(0, A * x + b)
predictor = MathOptAI.Pipeline(
    MathOptAI.Affine(A, b),
    MathOptAI.ReLU(),
)
model = Model()
@variable(model, x[1:n])
y, _ = MathOptAI.add_predictor(
    model,
    predictor,
    x;
    vector_nonlinear_oracle = true,

)
```

```julia
using JuMP
model = Model()
@variables(model, begin
    x[1:n]
    y[1:m]
end)
set = MOI.VectorNonlinearOracle(
    # g(x) := F(x) - y
    # evaluate g(x), ∇g(x)
)
@constraints(model, begin
    [x, y] in set
end)
```

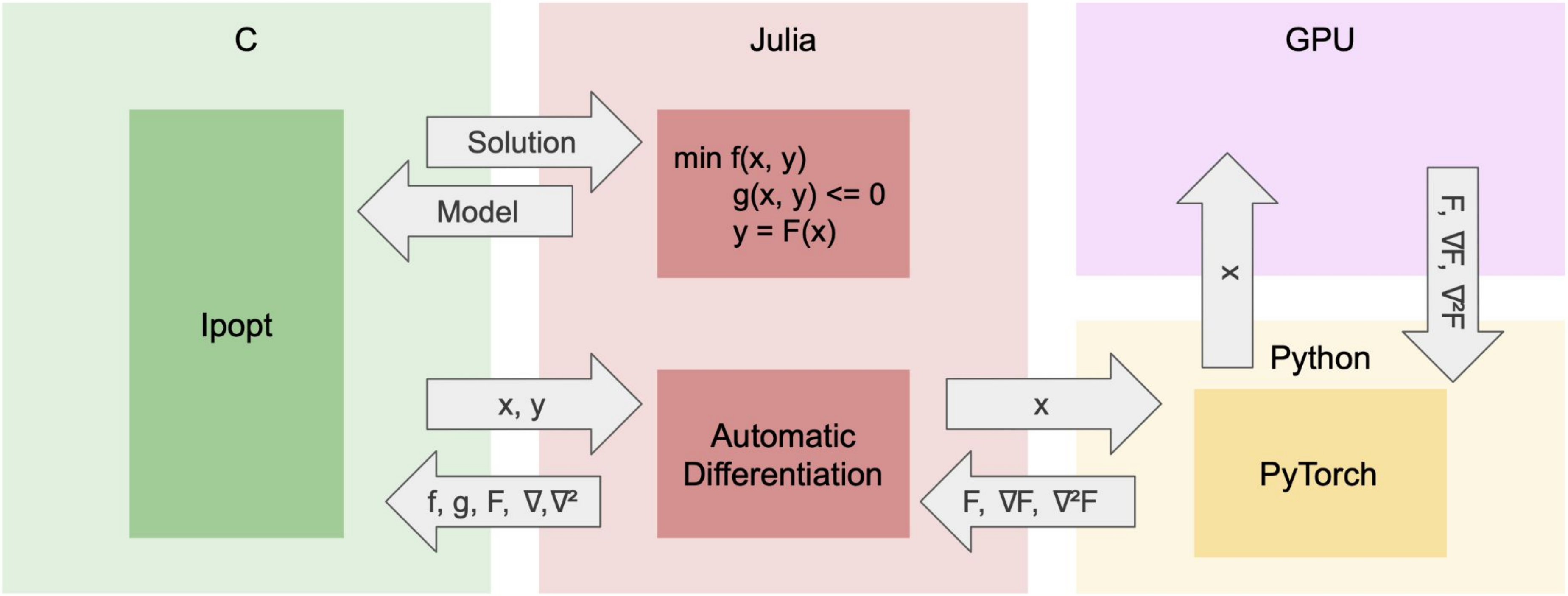
Gray box oracles are evaluated in Pytorch

MathOptAI automatically sets up the Julia-Python intercommunication

```julia
#!/usr/bin/julia
using JuMP, Ipopt, MathOptAI, PythonCall
model = Model(Ipopt.Optimizer)
@variable(model, 0 <= x[1:10] <= 1)
predictor = MathOptAI.PytorchModel("model.pt")
y, _ = MathOptAI.add_predictor(
    model,
    predictor,
    x;
    vector_nonlinear_oracle = true,
    device = "cuda",
    hessian = true,
)
```

Gray-box: Julia, C, Python, working together

JuMP problems call Ipopt in C, which calls back to Julia for oracles, which calls Python and PyTorch



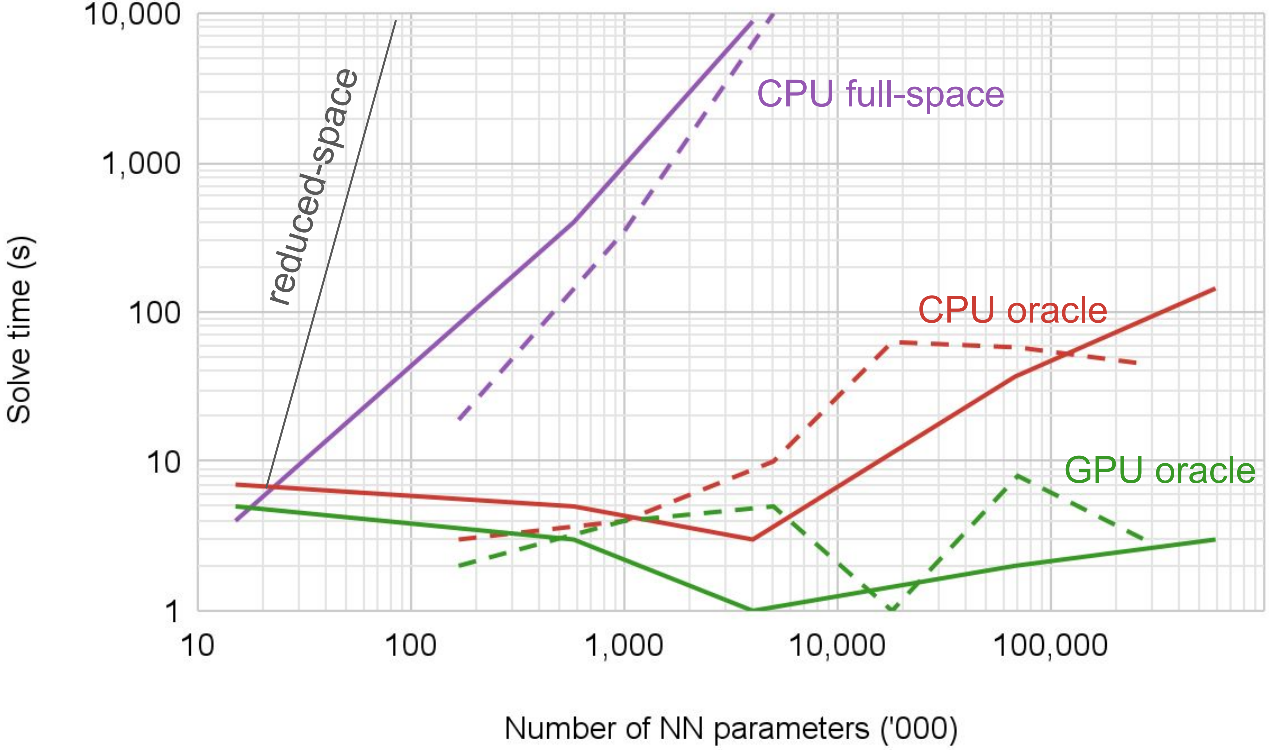
Three-ways to formulate a problem

Each with a different trade-off

	Full-space	Reduced-space	Gray-box
Pros	Sparsity Solvers can exploit linearity	Fewer variables and constraints	Can use external evaluation for oracles. Scales with input/output dimension, not intermediate dimension
Cons	Many extra variables and constraints	Complicated dense expressions	Requires oracle-based NLP. Cannot be used by global MINLP solvers
Bottleneck	Computing linear system because of problem size	Computing derivatives (JuMP’s AD does not do well at dense problems)	Moving data between Julia/Python/GPU

Runtime against size of the neural network

Two examples: SCOPF (solid) and MNIST (dashed)



Comparison to alternative packages

The design space is under-explored

The space of predictors we *could* add is very large.

What predictors are relevant and useful in practice?

We don’t need to copy what other packages have done.

	`MathOptAI.jl`	`OMLT`	`gurobi-machinelearning`	`PySCIPOpt-ML`	`GAMSPy`
Programming Language	Julia	Python	Python	Python	Python
License	BSD-3	BSD-3	Apache-2.0	Apache-2.0	MIT
Modeling Language	JuMP	Pyomo, JuMP	gurobipy	PySCIPOpt	GAMS
Solvers	Many	Many	Gurobi	SCIP	Many
Formulations					
Full-space	Yes	Yes	Yes	Yes	Yes
Reduced-space	Yes	Yes			
Gray-box	Yes				
GPU acceleration	Gray-box				
Neural network layers					
nn.AvgPool2D					Yes
nn.Conv2D		Yes			Yes
nn.Linear	Yes	Yes	Yes	Yes	Yes
nn.MaxPool2D		Yes			Yes
nn.ReLU	Yes	Yes	Yes	Yes	Yes
nn.Sequential	Yes	Yes	Yes	Yes	Yes
nn.Sigmoid	Yes	Yes		Yes	Yes
nn.Softmax	Yes	Yes		Yes	Yes
nn.Softplus	Yes	Yes		Yes	
nn.Tanh	Yes	Yes		Yes	Yes
Other predictor types					
Binary Decision Tree	Yes	Yes	Yes	Yes	Yes
Gaussian Process	Yes				
Gradient Boosted Tree	Yes	Yes	Yes	Yes	
Graph Neural Network		Yes			
Linear Regression	Yes	Yes	Yes	Yes	Yes
Logistic Regression	Yes	Yes	Yes	Yes	Yes
Random Forest	Yes		Yes	Yes	

Design principles

Err towards simplicity

Leverage Python's strengths

Support PyTorch.

Use PythonCall.

Don't try to write an ONNX parser in Julia.

Composition of predictors

Follow PyTorch "everything is a layer" not "a layer is affine + activation function."

Logistic is Affine |> Sigmoid, not a separate layer.

Leverage Julia's strengths

Multiple dispatch.

The package extension system is really great.

Inputs and outputs are Base.Vector

Strongly enforce the MethodError principle.

Broadcasting with different shapes is complicated. Julia and numpy have the opposite conventions.

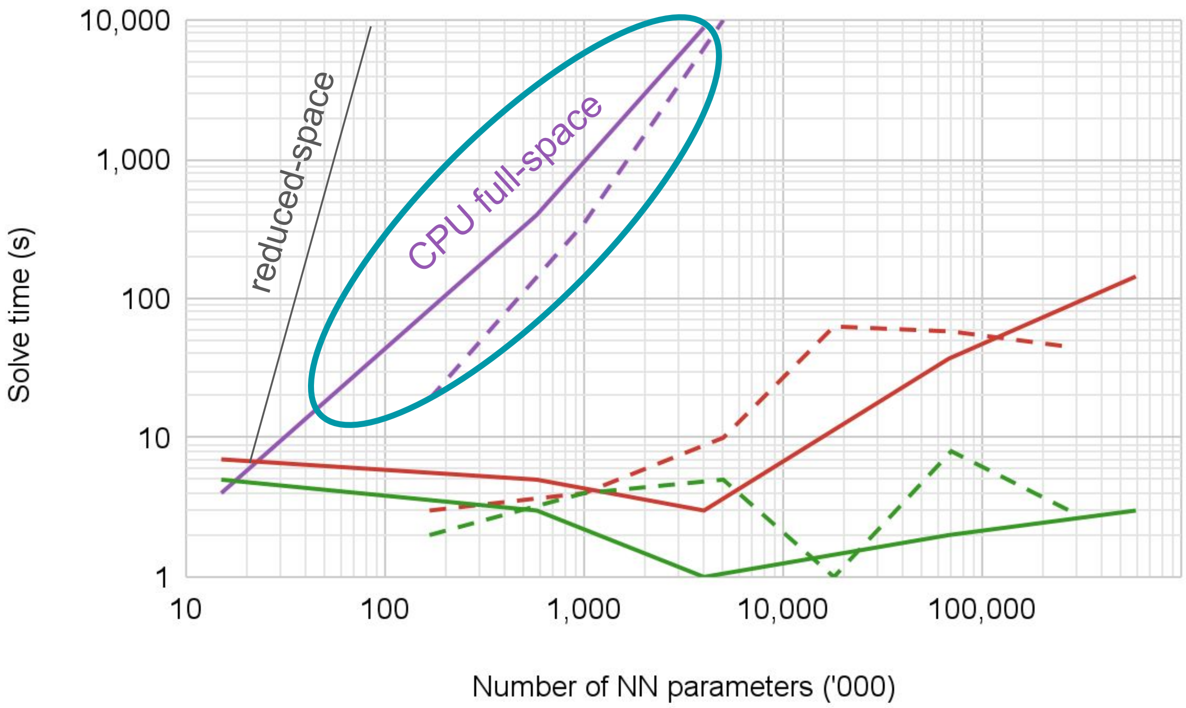
Get user to reshape Array into Vector.

References

- Papers on MathOptAI
    - Parker et al. (2025). Nonlinear optimization with GPU-accelerated neural network constraints. https://arxiv.org/abs/2509.22462
    - Dowson et al. (2025). MathOptAI.jl: Embed trained machine learning predictors into JuMP models. https://arxiv.org/abs/2507.03159
- Source codes
    - https://github.com/lanl-ansi/MathOptAI.jl
    - https://github.com/Gurobi/gurobi-machinelearning
    - https://github.com/cog-imperial/OMLT
    - https://github.com/GAMS-dev/gamspy
    - https://github.com/Opt-Mucca/PySCIPOpt-ML

Bonus slides: Using MathOptAI to develop custom linear solvers for MadNLP

Motivation: Full-space is slow



But…

- It supports ReLU via ReLUQuadratic
- It converges better for some problems (?!)

So we'd like to speed it up

Idea: Exploit the structure of a surrogate model

In the linear solver of an optimization algorithm

(1) Original KKT system

$$\begin{bmatrix} A & B^T \\ B & C \end{bmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} r_x \\ r_y \end{pmatrix}$$

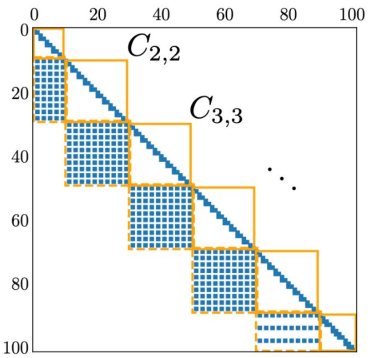
(2) Schur complement decomposition

$$S = A - B^T C^{-1} B$$
$$S \overline{\Delta x} = \overline{r_x}$$

1: $Z \leftarrow C^{-1} B$
2: $S \leftarrow A - B^T Z$
3: $\overline{\Delta x} \leftarrow S^{-1} \overline{r_x}$

(3) Neural network-informed block-triangular decomposition

$C =$



$$Z_i \leftarrow C_{ii}^{-1} \left(B_i - \sum_{j=1}^{i} C_{ji} Z_j \right)$$

Recall

```
y, formulation =
    MathOptAI.add_predictor(
    model, predictor, x)
```

Use the `formulation` struct as an input to a new linear solver

We have implemented a prototype for MadNLP

Software interface is a work-in-progress

```julia
# solver.jl
struct BlockTriangularSolver
<: MadNLP.AbstractLinearSolver
   csc::SparseMatrixCSC
   ...
end
```

```julia
# example.jl
y, formulation = MathOptAI.add_predictor(
     model, predictor, x)

nlp = NLPModelsJuMP.MathOptNLPModel(model)

indices = get_kkt_indices(model, formulation)

Madnlp = MadNLP.MadNLPSolver(
     nlp;
     linear_solver = BlockTriangularSolver,
     block_triangular_indices = indices,
)
```

Early performance results are promising

On ten KKT system solves

Model	Solver	NN param.	Total runtime (s)			Average		
			Initialization	Factorization	Backsolve	Residual	Refinement iterations	Speedup (×)
MNIST	MA57	1M	0.03	1	0.05	3.3E-06	0	–
MNIST	MA57	5M	0.1	15	0.2	4.2E-07	0	–
MNIST	MA57	18M	0.5	114	1	1.9E-07	0	–
MNIST	Ours	1M	0.3	5	0.2	8.3E-08	1	0.22
MNIST	Ours	5M	1	13	0.8	1.0E-06	2	1.1
MNIST	Ours	18M	5	33	7	2.6E-06	7	2.9
SCOPF	MA86	578k	0.02	2	0.1	6.7E-09	1	–
SCOPF	MA86	4M	0.1	26	0.7	2.0E-08	1	–
SCOPF	MA86	15M	0.5	142	3	2.5E-08	1	–
SCOPF	Ours	578k	0.1	0.3	0.1	6.2E-09	1	5.9
SCOPF	Ours	4M	2	2	0.4	3.0E-07	1	12
SCOPF	Ours	15M	4	5	2	1.2E-07	2	20

Funding

LANL:

- Artimis project
- Center for Nonlinear Studies (CNLS)
- Information Science and Technology Institute (ISTI)