ApplicationDrivenLearning.jl

Building decision-aware forecast models with composable architectures

A case study on hybrid renewable + storage dispatch

Giovanni Amorim — Soma Energy

JuMP-Dev 2026 — Edinburgh

The problem with MSE

Predictions feed optimization. Optimization makes decisions. Decisions have economic outcomes.

Standard ML pipeline:

$$\min_{\theta} \quad \mathrm{MSE}(\hat{y}_\theta(x),\, y)$$

Optimizes for **point accuracy**.

But the decision uses $\hat{y}$ in an LP/QP:

$$z^*(\hat{y}) = \arg\min_{z} \; G_p(z; \hat{y})$$

And the *real* cost is $G_a\big(z^*(\hat{y});\, y_{\text{actual}}\big)$.

The mismatch: a forecast that minimizes MSE may not minimize *decision regret*.

In power systems, a price predictor "off by $5 uniformly" has small MSE — but the $5 reorder can flip the LP's dispatch hour, costing dramatically more.

Application-Driven Learning (ADL) trains the predictor on the loss that matters:

$$\min_{\theta} \; \mathbb{E}_{(x,y)}\big[\, G_a\big(z^*(\hat{y}_\theta(x)); \, y\big)\,\big]$$

ADL as a bilevel problem

$$
\begin{aligned}
\min_{\theta} \quad & \mathbb{E}_{(x,y)}\left[\, G_a(z^*;\, y) \,\right] \\
\text{s.t.} \quad & z^* = \arg\min_{z} \; G_p(z;\, \hat{y}_\theta(x)) \\
& \hat{y}_\theta(x) = \Psi(x;\, \theta)
\end{aligned}
$$

The challenge: solving this bilevel program exactly is generally **prohibitive**. ADL relies on heuristics — chiefly, approximating the upper-level gradient by differentiating through the lower-level optimum.

Differentiating through the optimization:

1. Treat forecasts as **parameters** in the LP → `ParametricOptInterface.jl` (POI).
2. Compute $\partial z^* / \partial \hat{y}$ via **optimality conditions** → `DiffOpt.jl`.
3. Chain rule back to predictor weights: $\partial G_a / \partial \theta = \partial G_a / \partial z^* \cdot \partial z^* / \partial \hat{y} \cdot \partial \hat{y} / \partial \theta$.

`ApplicationDrivenLearning.jl` orchestrates all three.

Package ecosystem

`Flux.jl` — define the predictor

```
nn = Flux.Chain(
    Flux.Dense(41 => 64, relu),
    Flux.Dense(64 => 64, relu),
    Flux.Dense(64 => 144),
    custom_activation,
)
```

`JuMP.jl` — define the optimization model (solver-agnostic)

```
model = ADL.Model()
@variable(model, charge[1:H] >= 0, ADL.Policy)
@variable(model, generation[1:H], ADL.Forecast)
@objective(ADL.Plan(model),   Min, ...)
@objective(ADL.Assess(model), Min, ...)   # may differ from Plan
```

`ParametricOptInterface.jl` — forecasts as LP parameters

- Each `ADL.Forecast` variable becomes a POI parameter.
- Changing $\hat{y}$ doesn't rebuild the LP, just resolves.

`DiffOpt.jl` — differentiate through the LP

- Reads optimality conditions of the solved LP.
- Returns $\partial z^* / \partial(\hat{y})$ for free decision/parameter slots.

`MPI.jl` + **`JobQueueMPI.jl`** — distributed training

- Per-sample LP solves are distributed across MPI workers.

Case study: hybrid renewable + storage

Three co-managed projects:

Project	Tech	Battery (MWh)
A	wind	80
B	wind	100
C	solar	200

Daily decision (24-hour horizon):

- Charge / discharge schedule per hour, per project.
- Subject to cyclic SoC, ramp limits, cycle limits.

The two-stage LP:

```
Plan stage (forecast G_fc, π_fc):
  choose charge[h], discharge[h]
    + plan_sale[h]
  s.t. SoC dynamics, p_max, cycle
  maximize  Σ π_fc · plan_sale

Assess stage (realized G, π):
  charge, discharge LOCKED from plan
  assess_sale = G_actual + discharge
                   - charge          (signed)
  profit = Σ π_actual · assess_sale
```

Data

ERCOT hourly data, three co-managed projects.

Train period: **2023-08 → 2025-09** (17,640 hours ≈ 735 daily samples).

Input features (41): calendar (`hour_sin`, `month_cos`, `is_weekend`), generation lags, price lags, day-ahead market prices, ERCOT-wide wind/solar forecasts, system load forecasts.

Forecast targets (144): for each (project, hour in 1..24) we predict

- Generation $G_{h,p}$ (MWh, ≥ 0 via softplus activation)
- Price $\pi_{h,p}$ ($/MWh, free sign)

Three architectural priors

`single` — one network for all outputs

$$\hat{y} = \Psi(x;\, \theta)$$

All 41 features → all 144 targets. No structural prior.

`split` — separate networks for G and π

$$\hat{G} = \Psi^{G}(x_G;\, \theta^{G}), \quad \hat{\pi} = \Psi^{\pi}(x_\pi;\, \theta^{\pi})$$

Generation network sees only generation-related features (lags, regional wind/solar forecasts). Price network sees only price-related features (lags, DAM, load forecasts).

`shared` — one G-net and one π-net, reused per project

$$\hat{G}_{h,p} = \Psi^{G}(x_{G,p};\, \theta^{G}) \quad \forall p$$

Same subnets invoked once for each of the 3 projects, with that project's feature slice.

Inductive biases:

- `single`: none.
- `split`: supply and demand are separately predictable.
- `shared`: split + projects share predictor structure.

One mechanism for all three: `input_output_map`

```
input_output_map :: Vector{Dict{Vector{Int}, Vector{ADL.Forecast}}}
```

```
# single — one subnet, one Dict entry: all features → all forecasts
[ Dict(1:41 => [G_fc..., pi_fc...]) ]

# split  — two subnets, one Dict entry each
[ Dict(g_feat_idx  => vec(G_fc)),
  Dict(pi_feat_idx => vec(pi_fc)) ]

# shared — two subnets, N_PROJ Dict entries each (same subnet reused)
[ Dict(g_feat_idx_per_proj[p]  => vec(G_fc[:, p])  for p in 1:N_PROJ),
  Dict(pi_feat_idx_per_proj[p] => vec(pi_fc[:, p]) for p in 1:N_PROJ) ]
```

`ApplicationDrivenLearning.PredictiveModel(nns, input_output_map)` wires everything together — the same training loop works for all three.

What ADL gives us, summarized

Single mental model regardless of architecture:

```
pred_model = ApplicationDrivenLearning.PredictiveModel(net_or_nets, input_output_map)
ApplicationDrivenLearning.set_forecast_model(model, pred_model)

ApplicationDrivenLearning.train!(
    model, X_train, Y_dict_train,
    ApplicationDrivenLearning.Options(
        ApplicationDrivenLearning.GradientMPIMode;
        rule = Flux.Adam(1e-4), epochs = 100, batch_size = 64,
    ),
)
```

One LP, three predictors, identical training pipeline.

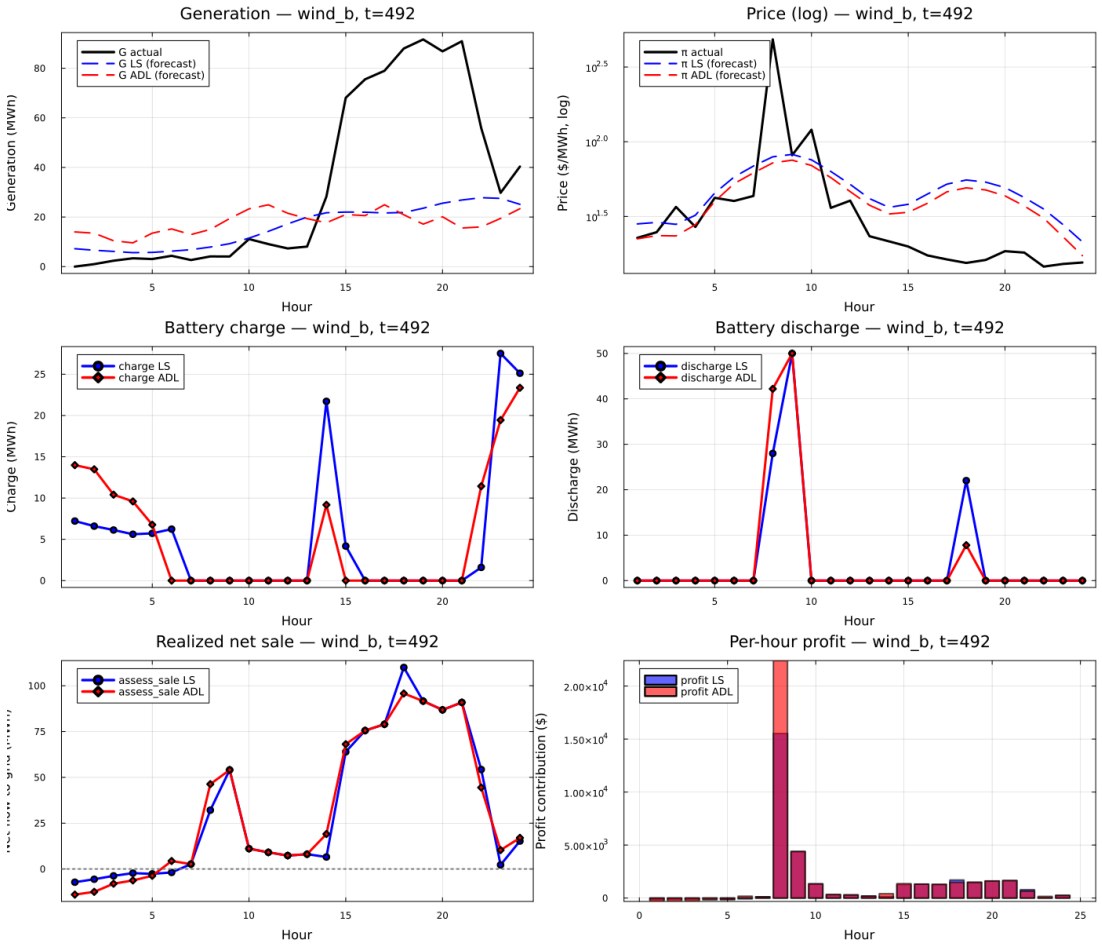
Results — all three architectures

Train metrics:

Arch	Params	LS Cost	ADL Cost	ΔCost
`single`	16k	−87,660	−87,713	+53
`split`	21k	−87,371	−89,525	**+2,154**
`shared`	13k	−87,500	−90,116	**+2,616**

Takeaway: all architectures improve over LS. **Structural priors (`split`, `shared`) recover ~40× more cost than `single`.**

A win on a normal wind day



A normal wind day (max π_actual = $484, no spike).

LS profit $61,514 → ADL profit **$68,148** (ΔProfit = **+$6,634**).

ADL nudges G_fc up at h=1-4 and down at h=18-23 — same total energy through the battery, but enough to **redistribute charging from h=14-15 (π≈$36) to h=1-4 (π≈$22)** and capture the price gap.

Architecture comparison — what matters

ΔCost vs. parameters:

Arch	Params	Structural prior	ΔCost
`single`	16k	none	+53
`split`	21k	G/π split	**+2,154**
`shared`	**13k**	G/π split + per-project sharing	**+2,616**

`shared` recovers the most cost with the fewest parameters.

- Splitted models were able to capture cost gradients better.
- Fewer params and project re-usage captured even more.

Conclusions

`ApplicationDrivenLearning.jl` enables decision-focused learning with composable architectures.

- Three structurally different predictors (`single`, `split`, `shared`) — one LP — one training pipeline. The user owns the architecture; the framework handles the gradient.
- Architectural priors that match the problem structure (G/π split, per-project sharing) capture decision-relevant signal far more efficiently than parameter count alone.

Thank you

`ApplicationDrivenLearning.jl` — github.com/LAMPSPUC/ApplicationDrivenLearning.jl

Code, data, and these slides:
github.com/Giovanni3A/ApplicationDrivenLearningExperiments.jl

Questions?