# Obstruction Loss

Diagrammatic Backpropagation (DB)

Simon Frost

## Table of contents



### Introduction

In category theory, a diagram commutes when all directed paths between the same pair of objects yield the same result. In practice, neural or learned morphisms rarely commute exactly. Obstruction loss quantifies this failure:

$$\mathcal{L}_{\text{obstruct}} = \|f \circ g - g \circ f\|^2$$

This is the core idea behind Diagrammatic Backpropagation (DB): instead of a single scalar loss, the training signal comes from how badly the diagram fails to commute. Minimizing obstruction loss pushes the system toward structural consistency — the morphisms learn to respect the diagrammatic contracts they are embedded in.

FunctorFlow.jl provides first-class support for obstruction losses via `add_obstruction_loss!` and the `db_square` block builder.

## Setup

```
using FunctorFlow
```

## Non-Commuting Squares

The simplest DB pattern is a square with two morphisms `f` and `g` acting on the same state space. The two paths around the square are `f∘g` and `g∘f`. If the morphisms don't commute, the obstruction loss will be nonzero.

Let's build this manually. We create a single object `S` (the state space) and two morphisms `f: S→S` and `g: S→S`, then compose them in both orders.

```
D = Diagram(:ManualDB)

add_object!(D, :S; kind=:state, shape="(4,)", description="State space")

add_morphism!(D, :f, :S, :S; description="First morphism")
add_morphism!(D, :g, :S, :S; description="Second morphism")

compose!(D, :f, :g; name=:fg)
compose!(D, :g, :f; name=:gf)
```

```
gf = g ∘ f
```

Now we add an obstruction loss that compares the two composed paths.

```
add_obstruction_loss!(D, :commutativity;
    paths=[(:fg, :gf)],
    comparator=:l2,
    weight=1.0)
```

```
commutativity = ‖fg, gf‖
```

Bind concrete implementations to the morphisms — simple functions where `f(x) = 2x` and `g(x) = x + 1`.

```
bind_morphism!(D, :f, x -> x * 2)
bind_morphism!(D, :g, x -> x .+ 1)
```

```
#5 (generic function with 1 method)
```

Compile and run the diagram.

```
compiled = compile_to_callable(D)
result = FunctorFlow.run(compiled, Dict(:S => [1.0, 2.0, 3.0, 4.0]))
```

```
ExecutionResult(5 values, 1 losses)
```

Inspect the output values and the obstruction loss. Since `f∘g(x) = 2(x+1) = 2x+2` while `g∘f(x) = 2x+1`, the paths do not agree — the loss should be nonzero.

```
println("f∘g path: ", result.values[:fg])
println("g∘f path: ", result.values[:gf])
println("Obstruction losses: ", result.losses)
```

```
f∘g path: [3.0, 5.0, 7.0, 9.0]
g∘f path: [4.0, 6.0, 8.0, 10.0]
Obstruction losses: Dict(:commutativity => 2.0)
```

## Using the DB Square Block

FunctorFlow ships a `db_square` block builder that creates the same pattern in one call.

```
db = db_square(;
    name=:QuickDB,
    first_impl=x -> x * 2,
    second_impl=x -> x .+ 1
)
```

```
Diagram :QuickDB
  Objects:
    State::state
  Operations:
    f: State → State
```

```
    g: State → State
    p1 = f ⋅ g
    p2 = g ⋅ f
  Losses:
    obstruction = ‖p1, p2‖
  Ports:
    → input (state)
    ← left_path (state)
    ← right_path (state)
    ← loss (loss)
```

```
compiled_db = compile_to_callable(db)
result_db = FunctorFlow.run(compiled_db, Dict(:State ⇒ [1.0, 2.0, 3.0, 4.0]))
println("Losses: ", result_db.losses)
```

```
Losses: Dict(:obstruction => 2.0)
```

The block builder wires up the objects, morphisms, compositions, and obstruction loss automatically.

## Comparators

The `comparator` keyword controls how two path outputs are compared. FunctorFlow provides several built-in comparators and supports custom ones.

### Built-in comparators

The `:l2` comparator computes the squared L2 norm of the difference (the default).

```
add_obstruction_loss!(D, :l2_loss;
    paths=[(:fg, :gf)],
    comparator=:l2)
```

```
l2_loss = ‖fg, gf‖
```

The `:l1` comparator computes the L1 (absolute value) norm.

```
add_obstruction_loss!(D, :l1_loss;
    paths=[(:fg, :gf)],
    comparator=:l1)
```

```
l1_loss = ‖fg, gf‖ [l1]
```

Custom comparators

You can write and bind an arbitrary comparator function. A comparator takes two arrays and returns a scalar loss.

```
D2 = Diagram(:CustomComp)
add_object!(D2, :S; kind=:state, shape="(4,)")
add_morphism!(D2, :f, :S, :S)
add_morphism!(D2, :g, :S, :S)
compose!(D2, :f, :g; name=:fg)
compose!(D2, :g, :f; name=:gf)

add_obstruction_loss!(D2, :cosine_loss;
    paths=[(:fg, :gf)],
    comparator=:custom_cosine)

bind_morphism!(D2, :f, x -> x * 2)
bind_morphism!(D2, :g, x -> x .+ 1)

# Bind a custom cosine-distance comparator
bind_comparator!(D2, :custom_cosine, (a, b) -> begin
    dot_val = sum(a .* b)
    norm_a = sqrt(sum(a .^ 2))
    norm_b = sqrt(sum(b .^ 2))
    1.0 - dot_val / (norm_a * norm_b + 1e-8)
end)

compiled2 = compile_to_callable(D2)
result2 = FunctorFlow.run(compiled2, Dict(:S => [1.0, 2.0, 3.0, 4.0]))
println("Cosine obstruction loss: ", result2.losses)
```

```
Cosine obstruction loss: Dict(:cosine_loss => 0.0011298162538414536)
```

### Multi-Path Losses

A single obstruction loss can monitor multiple path pairs simultaneously. This is useful when a diagram has several faces that should commute.

```
D3 = Diagram(:MultiPath)
add_object!(D3, :A; kind=:state)
add_object!(D3, :B; kind=:state)

add_morphism!(D3, :f, :A, :B)
add_morphism!(D3, :g, :A, :B)
add_morphism!(D3, :h, :A, :B)

add_obstruction_loss!(D3, :multi_loss;
    paths=[(:f, :g), (:g, :h)],
    comparator=:l2,
    weight=1.0)
```

```
multi_loss = ‖f, g‖ + ‖g, h‖
```

Here the loss is the sum of `||f(x) - g(x)||²` and `||g(x) - h(x)||²`. All three morphisms are encouraged to agree — enforcing consistency across redundant paths.

```
bind_morphism!(D3, :f, x -> x * 2)
bind_morphism!(D3, :g, x -> x * 2.1)
bind_morphism!(D3, :h, x -> x * 1.9)

compiled3 = compile_to_callable(D3)
result3 = FunctorFlow.run(compiled3, Dict(:A => [1.0, 2.0, 3.0]))
println("Multi-path losses: ", result3.losses)
```

```
Multi-path losses: Dict(:multi_loss => 1.1224972160321844)
```

### Training Interpretation

Obstruction loss is not just a diagnostic — it is a training signal. In Diagrammatic Backpropagation:

1. Each face of the diagram contributes an obstruction loss.
2. The total loss is the (weighted) sum of all obstruction losses.

3. Gradient descent on this total loss pushes morphisms toward commutativity.

This is structurally richer than a single end-to-end loss because it enforces local consistency at every face of the diagram, not just global input–output accuracy. The result is more modular, interpretable, and composable learning.

### Weighted Losses

The `weight` parameter scales the contribution of each obstruction loss to the total. This lets you prioritize certain commutativity constraints over others.

```
D4 = Diagram(:Weighted)
add_object!(D4, :S; kind=:state)
add_morphism!(D4, :f, :S, :S)
add_morphism!(D4, :g, :S, :S)
compose!(D4, :f, :g; name=:fg)
compose!(D4, :g, :f; name=:gf)

# High weight: strongly enforce this face
add_obstruction_loss!(D4, :critical_face;
    paths=[(:fg, :gf)],
    comparator=:l2,
    weight=10.0)
```

```
critical_face = ‖fg, gf‖ ×10.0
```

```
bind_morphism!(D4, :f, x -> x * 2)
bind_morphism!(D4, :g, x -> x .+ 1)

compiled4 = compile_to_callable(D4)
result4 = FunctorFlow.run(compiled4, Dict(:S => [1.0, 2.0, 3.0]))
println("Weighted loss: ", result4.losses)
```

```
Weighted loss: Dict(:critical_face => 17.32050807568877)
```

A weight of `10.0` means this face's contribution to the total loss is scaled by 10×, making the optimizer prioritize its commutativity above other, lower-weighted faces. Typical practice is to set critical structural constraints (e.g., the main DB square) to high weight and auxiliary or regularization faces to lower weight.