Left-Kan / Right-Kan Duality for Language Modeling

Prediction via Σ and repair via Δ as an adjoint pair

Simon Frost

Table of contents

Introduction

The adjunction $\Sigma_J \dashv \Delta_J$ is one of the deepest patterns in category theory. Given a functor $J : \mathcal{A} \to \mathcal{B}$, the left Kan extension Σ_J and right Kan extension Δ_J form an adjoint pair:

$$\Sigma_J \dashv \Delta_J$$

In FunctorFlow, this duality has a concrete computational interpretation:

- Σ (left-Kan): aggregate information along a relation — the universal prediction operation. Categorically, $\Sigma_J(F)(c) = \mathrm{colim}_{j \to c} F(j)$: the best summary of all sources that can attend to target c.
- Δ (right-Kan): complete partial information along a compatibility relation — the universal repair operation. Categorically, $\Delta_J(F)(c) = \lim_{c \to j} F(j)$: the most compatible completion consistent with all constraints on c.

In language modeling, these dual operations correspond to two complementary tasks:

1. Next-block prediction (Σ): given past context, aggregate it into a prediction for the next token block.
2. Block denoising (Δ): given a corrupted token block, repair it by completing the missing/corrupted positions.

Together, prediction and repair form a categorical auto-encoding loop:

- Unit $\eta : \mathrm{Id} \to \Delta_J \circ \Sigma_J$ — predict, then repair, should recover the original.
- Counit $\varepsilon : \Sigma_J \circ \Delta_J \to \mathrm{Id}$ — repair, then predict, should be faithful.

This vignette demonstrates both operations on a synthetic token sequence task, building all diagrams from scratch and using the `structured_lm_duality` block builder.

Setup

```
using Pkg
Pkg.activate(joinpath(@__DIR__, "..", ".."))
using FunctorFlow
using Random
using Statistics
```

Synthetic Data

We create a simple synthetic corpus with clear block structure: sequences are built by concatenating randomly chosen 4-token patterns from a small vocabulary. This makes both prediction and repair tractable — the patterns are learnable.

```
vocab_size = 50
seq_len = 32
block_size = 4
n_patterns = 10
rng = Random.MersenneTwister(42)

# Generate a fixed set of base patterns
base_patterns = [rand(rng, 1:vocab_size, block_size) for _ in 1:n_patterns]

println("Base patterns:")
for (i, p) in enumerate(base_patterns)
    println("  Pattern $i: $p")
end
```

```
Base patterns:
  Pattern 1: [11, 45, 20, 33]
  Pattern 2: [35, 36, 49, 38]
  Pattern 3: [11, 6, 10, 19]
  Pattern 4: [32, 46, 9, 46]
  Pattern 5: [10, 21, 3, 36]
  Pattern 6: [48, 9, 14, 39]
  Pattern 7: [7, 10, 33, 9]
  Pattern 8: [38, 9, 44, 37]
  Pattern 9: [10, 40, 32, 40]
  Pattern 10: [38, 12, 42, 16]
```

```
function generate_block_sequence(rng, base_patterns; seq_len=32, block_size=4)
    n_blocks = seq_len ÷ block_size
    blocks = [base_patterns[rand(rng, 1:length(base_patterns))] for _ in 1:n_blocks]
    return vcat(blocks...)
end

function make_batch(rng, base_patterns; batch_size=16, seq_len=32, block_size=4)
    seqs = [generate_block_sequence(rng, base_patterns; seq_len, block_size) for _ in 1:batch_
    return hcat(seqs...)  # seq_len × batch_size
end

batch = make_batch(rng, base_patterns; batch_size=8)
println("Batch shape: ", size(batch), " (seq_len × batch_size)")
println("First sequence: ", batch[:, 1])
```

```
Batch shape: (32, 8) (seq_len × batch_size)
```

```
First sequence: [48, 9, 14, 39, 10, 21, 3, 36, 35, 36  …  49, 38, 11, 6, 10, 19, 48, 9, 14, 39
```

Left Kan: Block Prediction (Σ)

The left Kan extension $\Sigma_J(F)$ computes the colimit — the universal aggregation of source values along a relation. In the language modeling setting:

- Source values: hidden-state embeddings at each position (simulated as one-hot token IDs).
- Relation: a causal mask — position j can attend to position c only if $j \leq c$.
- Reducer: aggregates the attended values (here, `:sum` for simplicity).
- Result: a contextualized representation at each position that summarizes all past tokens.

We then decode this aggregated context into predictions for the next token block.

Building the prediction diagram

```
D_predict = Diagram(:LeftKanPredict)

# Objects
add_object!(D_predict, :HiddenStates; kind=:messages,
            description="Token embeddings at each position")
add_object!(D_predict, :CausalRelation; kind=:relation,
            description="Causal mask: which positions attend to which")
add_object!(D_predict, :Contextualized; kind=:contextualized_messages,
            description="Aggregated context per position")

# Left-Kan: aggregate past context via causal relation
Σ(D_predict, :HiddenStates;
  along=:CausalRelation,
  name=:causal_aggregate,
  target=:Contextualized,
  reducer=:sum)

add_object!(D_predict, :BlockLogits; kind=:output,
            description="Predicted next-block token offsets")

# Decode to block offset predictions
add_morphism!(D_predict, :decode_block, :Contextualized, :BlockLogits;
              description="Map aggregated context to block-level predictions")

println(D_predict)
```

```
Diagram :LeftKanPredict ⟨4 objects, 1 morphisms, 1 Kan, 0 losses⟩
```

Executing the prediction

We build a causal relation as a dictionary mapping each position to the set of positions it can attend to (all earlier positions within the same block boundary).

```
seq = batch[:, 1]
n_blocks_seq = seq_len ÷ block_size

# Hidden states: map each position to its token ID
hidden_states = Dict(Symbol("pos_$i") ⇒ seq[i] for i in 1:seq_len)

# Causal relation: each block-boundary position attends to all positions
# in the current and previous blocks
causal_relation = Dict{Any, Any}()
for b in 1:n_blocks_seq
    target_pos = b * block_size  # last position of block b
    target_key = Symbol("block_$b")
    sources = [Symbol("pos_$i") for i in 1:target_pos]
    causal_relation[target_key] = sources
end

println("Causal relation (block 1 attends to): ", causal_relation[:block_1])
println("Causal relation (block 3 attends to): ", causal_relation[:block_3])
```

```
Causal relation (block 1 attends to): [:pos_1, :pos_2, :pos_3, :pos_4]
Causal relation (block 3 attends to): [:pos_1, :pos_2, :pos_3, :pos_4, :pos_5, :pos_6, :pos_7,
```

```
compiled_predict = compile_to_callable(D_predict)

# Provide a simple decode function: identity (pass through the aggregated sum)
result_predict = FunctorFlow.run(compiled_predict,
    Dict(:HiddenStates ⇒ hidden_states,
         :CausalRelation ⇒ causal_relation);
    morphisms=Dict(:decode_block ⇒ x → x)
)

println("Σ aggregation result (causal_aggregate):")
agg = result_predict.values[:causal_aggregate]
for b in 1:min(4, n_blocks_seq)
```

```
    key = Symbol("block_$b")
    if haskey(agg, key)
        println("  $key → $(agg[key])  (sum of token IDs in causal window)")
    end
end
```

```
Σ aggregation result (causal_aggregate):
  block_1 → 110  (sum of token IDs in causal window)
  block_2 → 180  (sum of token IDs in causal window)
  block_3 → 338  (sum of token IDs in causal window)
  block_4 → 446  (sum of token IDs in causal window)
```

The left-Kan aggregation sums all token IDs in the causal window for each block boundary. In a real model, these would be dense embeddings and the reducer would be learned attention — but the categorical structure is the same.

Right Kan: Block Denoising (Δ)

The right Kan extension $\Delta_J(F)$ computes the limit — the most compatible completion of partial data. In the denoising setting:

- Source values: a corrupted token block where some positions have been replaced with `nothing`.
- Relation: a compatibility structure — each corrupted position checks its neighbors for valid values.
- Reducer: `:first_non_null` — fill each missing position with the first available compatible value.
- Result: a repaired block with missing positions filled in.

Building the denoising diagram

```
D_denoise = Diagram(:RightKanDenoise)

add_object!(D_denoise, :NoisyBlock; kind=:partial,
            description="Token block with corrupted positions (nothing = missing)")
add_object!(D_denoise, :CompatibilityRelation; kind=:relation,
            description="Which positions can fill which missing slots")
add_object!(D_denoise, :CompletedBlock; kind=:completed_values,
            description="Repaired block")
```

```
# Right-Kan: complete partial/corrupted data
Δ(D_denoise, :NoisyBlock;
  along=:CompatibilityRelation,
  name=:repair,
  target=:CompletedBlock,
  reducer=:first_non_null)

println(D_denoise)
```

```
Diagram :RightKanDenoise ⟨3 objects, 0 morphisms, 1 Kan, 0 losses⟩
```

Corruption and repair

```
function corrupt_block(rng, block; noise_rate=0.3)
    mask = rand(rng, length(block)) .> noise_rate
    corrupted = Vector{Union{Int, Nothing}}(copy(block))
    for i in eachindex(corrupted)
        if !mask[i]
            corrupted[i] = nothing
        end
    end
    return corrupted, mask
end

# Take one block from the first sequence
original_block = seq[1:block_size]
corrupted, mask = corrupt_block(rng, original_block; noise_rate=0.5)
println("Original block: ", original_block)
println("Corruption mask: ", mask, "  (true = kept, false = corrupted)")
println("Corrupted block: ", corrupted)
```

```
Original block: [48, 9, 14, 39]
Corruption mask: Bool[1, 1, 0, 1]  (true = kept, false = corrupted)
Corrupted block: Union{Nothing, Int64}[48, 9, nothing, 39]
```

```
# Build noisy values: position → token or nothing
noisy_values = Dict(Symbol("t_$i") ⇒ corrupted[i] for i in 1:block_size)
```

```
# Compatibility: each position can look at all other positions for repair
compat_relation = Dict{Any, Any}()
for i in 1:block_size
    others = [Symbol("t_$j") for j in 1:block_size if j != i]
    compat_relation[Symbol("t_$i")] = others
end

compiled_denoise = compile_to_callable(D_denoise)
result_denoise = FunctorFlow.run(compiled_denoise,
    Dict(:NoisyBlock => noisy_values,
         :CompatibilityRelation => compat_relation))

repaired = result_denoise.values[:repair]
println("\nRight-Kan repair (Δ):")
for i in 1:block_size
    key = Symbol("t_$i")
    orig = original_block[i]
    noisy = corrupted[i]
    fixed = get(repaired, key, nothing)
    status = noisy === nothing ? " ← REPAIRED" : ""
    println("  $key: original=$orig, corrupted=$noisy, repaired=$fixed$status")
end
```

```
Right-Kan repair (Δ):
  t_1: original=48, corrupted=48, repaired=9
  t_2: original=9, corrupted=9, repaired=48
  t_3: original=14, corrupted=nothing, repaired=48 ← REPAIRED
  t_4: original=39, corrupted=39, repaired=48
```

The right-Kan extension fills each `nothing` slot with the first non-null value from compatible neighbors. This is the categorical version of masked token reconstruction — $\Delta_J(F)(c)$ finds the value most consistent with all constraints pointing out of c.

The Duality Diagram

FunctorFlow provides `structured_lm_duality`, a block builder that combines both operations into a single diagram with a shared input:

```
D_dual = structured_lm_duality()
println(D_dual)
```

```
Diagram :StructuredLMDuality ⟨5 objects, 0 morphisms, 2 Kan, 0 losses⟩
```

The `structured_lm_duality` block internally:

1. Creates a KET block (left-Kan prediction via Σ) under the `:predict` namespace.
2. Creates a completion block (right-Kan repair via Δ) under the `:repair` namespace.
3. Aliases both to share a common `:SharedInput` object.

Manual construction

We can also build the duality manually to see the full structure:

```
D_manual = Diagram(:ManualDuality)

# Shared embedding space
add_object!(D_manual, :Tokens; kind=:hidden_state,
            description="Token embeddings — shared input")

# --- Left branch: prediction via Σ ---
add_object!(D_manual, :CausalRelation; kind=:relation)
add_object!(D_manual, :PredictionContext; kind=:contextualized_messages)
add_object!(D_manual, :PredictedBlock; kind=:output)

Σ(D_manual, :Tokens;
  along=:CausalRelation,
  name=:predict_aggregate,
  target=:PredictionContext,
  reducer=:sum)
add_morphism!(D_manual, :predict_decode, :PredictionContext, :PredictedBlock)

# --- Right branch: repair via Δ ---
add_object!(D_manual, :NoisyTokens; kind=:partial)
add_object!(D_manual, :RepairRelation; kind=:relation)
add_object!(D_manual, :RepairedTokens; kind=:completed_values)
add_object!(D_manual, :DenoisedBlock; kind=:output)

Δ(D_manual, :NoisyTokens;
```

```
  along=:RepairRelation,
  name=:repair_complete,
  target=:RepairedTokens,
  reducer=:first_non_null)
add_morphism!(D_manual, :repair_decode, :RepairedTokens, :DenoisedBlock)

println(D_manual)
```

```
Diagram :ManualDuality ⟨8 objects, 2 morphisms, 2 Kan, 0 losses⟩
```

Executing both branches

```
compiled_dual = compile_to_callable(D_manual)

# Prepare inputs for both branches
tokens = Dict(Symbol("pos_$i") => seq[i] for i in 1:seq_len)

# Causal relation for prediction (block-level)
causal_rel = Dict{Any, Any}()
for b in 1:n_blocks_seq
    target_pos = b * block_size
    causal_rel[Symbol("block_$b")] = [Symbol("pos_$i") for i in 1:target_pos]
end

# Noisy tokens for repair (corrupt the second block)
block2 = seq[(block_size+1):(2*block_size)]
corrupted2, mask2 = corrupt_block(rng, block2; noise_rate=0.5)
noisy_tokens = Dict(Symbol("b2_$i") => corrupted2[i] for i in 1:block_size)
repair_rel = Dict{Any, Any}()
for i in 1:block_size
    repair_rel[Symbol("b2_$i")] = [Symbol("b2_$j") for j in 1:block_size if j != i]
end

result_dual = FunctorFlow.run(compiled_dual,
    Dict(:Tokens => tokens,
         :CausalRelation => causal_rel,
         :NoisyTokens => noisy_tokens,
         :RepairRelation => repair_rel);
    morphisms=Dict(
        :predict_decode => x -> x,
```

```
        :repair_decode ⇒ x → x
    )
)

println("=== Left-Kan (Σ): Prediction ===")
agg_dual = result_dual.values[:predict_aggregate]
for b in 1:min(3, n_blocks_seq)
    key = Symbol("block_$b")
    haskey(agg_dual, key) && println("  $key → $(agg_dual[key])")
end

println("\n=== Right-Kan (Δ): Repair ===")
rep_dual = result_dual.values[:repair_complete]
for i in 1:block_size
    key = Symbol("b2_$i")
    orig = block2[i]
    noisy = corrupted2[i]
    fixed = get(rep_dual, key, nothing)
    status = noisy === nothing ? " ← REPAIRED" : ""
    println("  $key: original=$orig, corrupted=$noisy, repaired=$fixed$status")
end
```

```
=== Left-Kan (Σ): Prediction ===
  block_1 → 110
  block_2 → 180
  block_3 → 338

=== Right-Kan (Δ): Repair ===
  b2_1: original=10, corrupted=10, repaired=21
  b2_2: original=21, corrupted=21, repaired=10
  b2_3: original=3, corrupted=3, repaired=10
  b2_4: original=36, corrupted=nothing, repaired=10 ← REPAIRED
```

The two branches compute different things from the same conceptual space:

- The left branch (Σ) merges many source values into a single summary per target — it answers "what is the aggregated context?".
- The right branch (Δ) fills in missing values from compatible neighbors — it answers "what is the most consistent completion?".

The Predict-Repair Loop

The categorical adjunction $\Sigma_J \dashv \Delta_J$ predicts a pipeline:

1. Start with a clean sequence x.
2. Predict the next block: $\hat{y} = \Sigma_J(x)$ — aggregate context into a prediction.
3. Corrupt the prediction: $\tilde{y} = \text{noise}(\hat{y})$ — simulate real-world noise.
4. Repair the corrupted block: $\hat{x} = \Delta_J(\tilde{y})$ — complete the partial data.

The unit of the adjunction says that $\Delta_J \circ \Sigma_J \approx \text{Id}$ — the predict-then-repair cycle should approximately recover the original data. Let us verify this numerically.

```
function predict_repair_loop(seq, block_idx; noise_rate=0.3, block_size=4)
    # Step 1: Prediction via Σ (aggregate tokens up to this block)
    D_loop = Diagram(:PredictRepairLoop)
    add_object!(D_loop, :Src; kind=:messages)
    add_object!(D_loop, :Rel; kind=:relation)
    Σ(D_loop, :Src; along=:Rel, name=:predicted, reducer=:mean)

    compiled_loop = compile_to_callable(D_loop)

    target_end = block_idx * block_size
    src_vals = Dict(Symbol("p_$i") => Float64(seq[i]) for i in 1:target_end)
    rel_vals = Dict(:target => [Symbol("p_$i") for i in 1:target_end])

    result_pred = FunctorFlow.run(compiled_loop, Dict(:Src => src_vals, :Rel => rel_vals))
    predicted_mean = result_pred.values[:predicted][:target]

    # Step 2: Corruption — replace some block positions with nothing
    true_block = seq[(target_end - block_size + 1):target_end]
    rng_loop = Random.MersenneTwister(block_idx)
    corrupted_block, cmask = corrupt_block(rng_loop, true_block; noise_rate)

    # Step 3: Repair via Δ (fill missing with predicted mean)
    D_repair = Diagram(:RepairStep)
    add_object!(D_repair, :Partial; kind=:partial)
    add_object!(D_repair, :Compat; kind=:relation)
    Δ(D_repair, :Partial; along=:Compat, name=:repaired, reducer=:first_non_null)

    compiled_repair = compile_to_callable(D_repair)

    # Inject predicted mean as a fallback source alongside the corrupted values
    partial_vals = Dict{Any, Any}(Symbol("t_$i") => corrupted_block[i] for i in 1:block_size)
```

```
    partial_vals[:predicted] = round(Int, predicted_mean)

    compat_vals = Dict{Any, Any}()
    for i in 1:block_size
        neighbors = [Symbol("t_$j") for j in 1:block_size if j != i]
        push!(neighbors, :predicted)
        compat_vals[Symbol("t_$i")] = neighbors
    end

    result_rep = FunctorFlow.run(compiled_repair, Dict(:Partial ⇒ partial_vals, :Compat ⇒ co
    repaired = result_rep.values[:repaired]

    repaired_block = [get(repaired, Symbol("t_$i"), nothing) for i in 1:block_size]

    return (true_block=true_block, corrupted=corrupted_block,
            predicted_mean=predicted_mean, repaired=repaired_block)
end

# Run for blocks 2 through 5
for b in 2:min(5, n_blocks_seq)
    r = predict_repair_loop(seq, b; noise_rate=0.5)
    n_corrupted = count(x -> x === nothing, r.corrupted)
    n_recovered = sum(r.repaired[i] == r.true_block[i] for i in 1:block_size if r.corrupted[i]
    println("Block $b: true=$(r.true_block), predicted_mean=$(round(r.predicted_mean; digits=1
            "corrupted=$n_corrupted positions, recovered=$n_recovered/$n_corrupted")
end
```

```
Block 2: true=[10, 21, 3, 36], predicted_mean=22.5, corrupted=2 positions, recovered=0/2
Block 3: true=[35, 36, 49, 38], predicted_mean=28.2, corrupted=3 positions, recovered=0/3
Block 4: true=[38, 12, 42, 16], predicted_mean=27.9, corrupted=2 positions, recovered=0/2
Block 5: true=[38, 9, 44, 37], predicted_mean=28.7, corrupted=0 positions, recovered=0/0
```

Even with this simple symbolic execution (integer token IDs, `:mean` aggregation, `:first_non_null` repair), the predict-repair loop demonstrates the categorical structure. In a neural setting, the reducers would be learned (attention for Σ, compatibility scoring for Δ), but the diagrammatic skeleton is identical.

Categorical Interpretation

The adjunction $\Sigma_J \dashv \Delta_J$ gives us:

Unit: $\eta : \mathrm{Id} \to \Delta_J \circ \Sigma_J$

Starting from a complete sequence, Σ aggregates it into a prediction, and Δ repairs back toward the original. The unit measures how much information the predict-then-repair cycle preserves:

$$x \xrightarrow{\eta_x} \Delta_J(\Sigma_J(x))$$

If η is close to an isomorphism, the cycle is lossless — prediction captures enough information for exact repair.

Counit: $\varepsilon : \Sigma_J \circ \Delta_J \to \mathrm{Id}$

Starting from corrupted data, Δ repairs it, and Σ re-aggregates the repaired version. The counit measures faithfulness:

$$\Sigma_J(\Delta_J(\tilde{x})) \xrightarrow{\varepsilon_{\tilde{x}}} \tilde{x}$$

If ε is close to an isomorphism, repair-then-predict is faithful — the repaired data re-aggregates consistently.

In practice

This is the categorical version of auto-encoding:

- The unit η corresponds to the reconstruction objective (VAE / denoising autoencoder).
- The counit ε corresponds to the consistency objective (cycle consistency, round-trip loss).
- Training both branches jointly with an obstruction loss on the round-trip error enforces the adjunction.

```
# Verify numerically: predict then repair on the first few blocks
println("Predict-then-repair round-trip accuracy:")
for b in 2:min(6, n_blocks_seq)
    r = predict_repair_loop(seq, b; noise_rate=0.5)
    accuracy = mean(r.repaired[i] == r.true_block[i] for i in 1:block_size)
    println("  Block $b: $(round(accuracy * 100; digits=1))% positions match original")
end
```

```
Predict-then-repair round-trip accuracy:
  Block 2: 0.0% positions match original
  Block 3: 0.0% positions match original
  Block 4: 0.0% positions match original
  Block 5: 0.0% positions match original
  Block 6: 0.0% positions match original
```

Summary

Aspect	Σ (Left-Kan)	Δ (Right-Kan)
Category theory	Colimit	Limit
Universal property	Best summary given all sources	Most compatible completion given all constraints
Computation	Aggregation (many → one)	Completion (partial → complete)
AI interpretation	Prediction	Repair / denoising
Default reducer	`:sum`	`:first_non_null`
Data flow	Many sources → single target	Incomplete data → filled data
LM example	Next-block prediction	Masked token reconstruction
FunctorFlow operator	`Σ(D, src; along=rel, ...)`	`Δ(D, src; along=rel, ...)`

The key insight is that Σ and Δ are not two separate mechanisms — they are two faces of the same categorical coin. FunctorFlow makes this explicit by giving them the same API shape (source, relation, reducer) while their default reducers encode the duality: `:sum` aggregates existing values, while `:first_non_null` fills in missing ones.

The `structured_lm_duality` block builder packages this pattern into a reusable component, but as we have shown, the duality can also be constructed manually for full control over the diagram topology.