Causal-JEPA: Object-Level Interventions as Categorical Operations

Connecting C-JEPA's latent interventions with FunctorFlow's RN-Kan-Do-Calculus

Simon Frost

Table of contents

Introduction

C-JEPA (Causal-JEPA, arXiv:2602.11389) demonstrates that object-level masking during JEPA training induces a causal inductive bias — making interaction reasoning *functionally necessary*. Rather than masking random patches (as in I-JEPA), C-JEPA masks entire *objects* across history frames. This forces the predictor to recover the masked object's state purely from its interactions with other objects, not from its own self-dynamics.

The central theoretical result (Theorem 1: Interaction Necessity) shows that under object-level masking, minimizing the prediction loss necessarily identifies the influence neighborhood — the minimal set of context entities sufficient to predict a masked entity. This is the causal analog of a Markov blanket.

FunctorFlow.jl's categorical framework maps these ideas precisely:

C-JEPA Concept	FunctorFlow Construction
Object slots $S_t = \{s_t^1, \ldots, s_t^N\}$	Categorical objects in a Diagram
Object-level masking	do-calculus intervention via left-Kan Σ
Unmasked context	Conditioning via right-Kan Δ
Influence neighborhood $\mathcal{N}_t(i)$	Markov blanket = minimal sufficient Δ
Joint prediction loss	Obstruction loss in JEPA diagram
EMA target encoder	Frozen reference coalgebra
History reconstruction ($\mathcal{L}_{\text{history}}$)	Right-Kan completion (repair)
Future prediction ($\mathcal{L}_{\text{future}}$)	Left-Kan aggregation (prediction)
Slot Attention (entity decomposition)	Product diagram ($\otimes$) of entity diagrams
Counterfactual reasoning	Composition: intervene (Σ) then condition (Δ)
Identity anchor	Coalgebra initial state

Setup

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

using FunctorFlow
using Random
using Statistics
```

Object-Centric World as a Product Diagram

C-JEPA processes each frame through Slot Attention, decomposing a scene into N object slots: $S_t = \{s_t^1, \ldots, s_t^N\}$. Each slot is an independent entity with its own latent state and dynamics. Categorically, each entity is an F-coalgebra — a state paired with a transition morphism $X \to F(X)$ — and the full scene state is their product.

```
function entity_diagram(name::Symbol, slot_dim::Int)
    D = Diagram(name)
    add_object!(D, :State, kind=:hidden_state,
                description="Entity latent state (ℝ^$slot_dim)")
    add_object!(D, :NextState, kind=:hidden_state,
                description="Next-step entity state")
    add_morphism!(D, :dynamics, :State, :NextState,
```

```
                    description="Entity self-dynamics f: X → F(X)")
    add_coalgebra!(D, :entity_coalgebra,
                   state=:State, transition=:dynamics, functor_type=:identity)
    return D
end

N = 5  # number of entity slots
slot_dim = 16
entities = [entity_diagram(Symbol("Entity_$i"), slot_dim) for i in 1:N]

println("Created $(length(entities)) entity coalgebras")
for (i, e) in enumerate(entities)
    coalgs = get_coalgebras(e)
    println("  Entity $i: coalgebra=$(first(keys(coalgs)))")
end
```

```
Created 5 entity coalgebras
  Entity 1: coalgebra=entity_coalgebra
  Entity 2: coalgebra=entity_coalgebra
  Entity 3: coalgebra=entity_coalgebra
  Entity 4: coalgebra=entity_coalgebra
  Entity 5: coalgebra=entity_coalgebra
```

The product construction gives us projection morphisms π_i from the scene to each entity, and the universal property ensures that any compatible mapping factors through the product:

```
scene = product(entities...; name=:Scene)
scene_check = verify(scene)
println("Scene product: $(scene_check)")
println("Projections: $(scene.projections)")
println(summary(scene.product_diagram))
```

```
Scene product: (passed = true, checks = Dict{Symbol, Bool}(:has_factor_factor_4 => 1, :has_fac
Projections: [:factor_1, :factor_2, :factor_3, :factor_4, :factor_5]
FunctorFlow.Diagram
```

Object-Level Masking as a Categorical Intervention

This is the core conceptual bridge. In C-JEPA, masking object i across history means replacing its self-dynamics with a learnable mask token (anchored by its initial state at $t = 0$). The remaining

unmasked entities form the *context*. The predictor must recover the masked entity's state from context alone.

In FunctorFlow's RN-Kan-Do-Calculus:

- Observational regime: Full history Z_T with all entities visible
- Interventional regime: Masked history $\bar{Z}_T$ where entity i's history is replaced by mask tokens — this is do(entity$_i$ = mask)

The left-Kan extension Σ implements the do-operation (intervention), and the right-Kan extension Δ implements conditioning (observing the context).

```
obs_regime = CausalContext(:full_observation;
    observational_regime=:all_entities_visible,
    interventional_regime=:entity_masked)

causal_d = build_causal_diagram(:EntityInteraction;
    context=obs_regime,
    observation_source=:EntityStates,
    causal_relation=:InteractionGraph,
    intervention_target=:InterventionalPrediction,
    conditioning_target=:ConditionalState)

println("Causal diagram: $(causal_d.name)")
println("  Conditioning (Δ, right-Kan): $(causal_d.conditioning_kan)")
println("  Intervention (Σ, left-Kan):  $(causal_d.intervention_kan)")
println(summary(causal_d.base_diagram))
```

```
Causal diagram: EntityInteraction
  Conditioning (Δ, right-Kan): condition
  Intervention (Σ, left-Kan):  intervene
FunctorFlow.Diagram
```

The masking operation is categorically a left-Kan extension (colimit / aggregation): we push forward along the masking relation, which removes entity i's self-dynamics from the diagram. The prediction of the masked entity then requires a right-Kan extension (limit / completion) from the remaining context.

We can check identifiability — is the masked entity's state recoverable from context?

```
ident = is_identifiable(causal_d, :InterventionalPrediction;
                        observed=[:EntityStates])
println("Identifiability: $(ident)")
```

```
Identifiability: (identifiable = true, rule = :adjustment, reasoning = "Both Kan extensions sh
door criterion")
```

Influence Neighborhood as Right-Kan Completion

C-JEPA's influence neighborhood $\mathcal{N}_t(i)$ is the minimal sufficient subset $\mathcal{N}_t(i) \subseteq Z_T^{(-i)}$ needed to predict masked entity i. In categorical terms, this is the kernel of the right-Kan extension — the smallest subdiagram whose Δ-completion recovers the masked state.

Theorem 1 (Interaction Necessity) states: under object-level masking, minimizing the prediction loss necessarily identifies $\mathcal{N}_t(i)$. Categorically, the optimal predictor's right-Kan extension is supported exactly on the influence neighborhood.

```
D_influence = Diagram(:InfluenceNeighborhood)

add_object!(D_influence, :MaskedEntity, kind=:hidden_state,
            description="Entity whose history is masked (target)")
add_object!(D_influence, :ContextEntities, kind=:hidden_state,
            description="Unmasked entities forming the context")
add_object!(D_influence, :InteractionRelation, kind=:relation,
            description="Which context entities influence the target")
add_object!(D_influence, :CompletedState, kind=:hidden_state,
            description="Inferred state of masked entity via right-Kan")

# Right-Kan: complete the masked entity from context
Δ(D_influence, :ContextEntities;
  along=:InteractionRelation,
  name=:infer_from_context,
  target=:CompletedState,
  reducer=:first_non_null)

println(summary(D_influence))
```

```
FunctorFlow.Diagram
```

The right-Kan extension $\Delta_J F$ computes the best approximation of the masked entity given the context, mediated by the interaction relation J. The influence neighborhood is precisely the support of J — the minimal set of context entities for which $\Delta_J F \cong F$ (the completion is exact).

C-JEPA Training as Coalgebraic Obstruction

The C-JEPA training loss has two complementary components, each corresponding to a Kan extension:

- $\mathcal{L}_{\text{history}}$: Recover masked slots from context ? right-Kan completion (Δ) — repair missing data by conditioning on context
- $\mathcal{L}_{\text{future}}$: Predict future slots from history ? left-Kan aggregation (Σ) — forward prediction by pushing history forward

The total loss measures the obstruction to commutativity — the failure of the predicted state to match the target encoder's output.

```
D_cjepa = Diagram(:CJEPA)

# -- History branch (right-Kan: repair masked entities) --
add_object!(D_cjepa, :HistoryContext, kind=:hidden_state,
            description="Unmasked history slots (context encoder output)")
add_object!(D_cjepa, :MaskRelation, kind=:relation,
            description="Which slots are masked vs. visible")
add_object!(D_cjepa, :ReconstructedHistory, kind=:hidden_state,
            description="Recovered masked history slots")
add_object!(D_cjepa, :TargetHistory, kind=:hidden_state,
            description="Ground truth masked history (from EMA target encoder)")

Δ(D_cjepa, :HistoryContext;
  along=:MaskRelation,
  name=:recover_masked,
  target=:ReconstructedHistory,
  reducer=:first_non_null)

# Morphism from target encoder (EMA-frozen) to produce ground truth
add_morphism!(D_cjepa, :target_enc_history, :HistoryContext, :TargetHistory,
              description="EMA target encoder for history (frozen)")

add_obstruction_loss!(D_cjepa, :history_loss;
    paths=[(:recover_masked, :target_enc_history)],
    comparator=:l2, weight=1.0,
    description="L_history: masked slot reconstruction")

# -- Future branch (left-Kan: predict future from context) --
add_object!(D_cjepa, :CausalRelation, kind=:relation,
            description="Temporal causal structure (history → future)")
```

```
add_object!(D_cjepa, :PredictedFuture, kind=:hidden_state,
            description="Predicted future slots")
add_object!(D_cjepa, :TargetFuture, kind=:hidden_state,
            description="Ground truth future slots (from EMA target encoder)")

Σ(D_cjepa, :HistoryContext;
  along=:CausalRelation,
  name=:predict_future,
  target=:PredictedFuture,
  reducer=:sum)

add_morphism!(D_cjepa, :target_enc_future, :HistoryContext, :TargetFuture,
              description="EMA target encoder for future (frozen)")

add_obstruction_loss!(D_cjepa, :future_loss;
    paths=[(:predict_future, :target_enc_future)],
    comparator=:l2, weight=1.0,
    description="L_future: forward prediction")

println(summary(D_cjepa))
```

```
FunctorFlow.Diagram
```

The two losses interact: $\mathcal{L}_{\text{history}}$ forces the model to learn the interaction structure (which entities influence which), while $\mathcal{L}_{\text{future}}$ forces it to learn temporal dynamics conditioned on that interaction structure. Together, they are the obstruction to a fully commutative diagram — the residual measures how far the learned world model is from perfect causal understanding.

Synthetic Multi-Object Interaction Simulation

To make these ideas concrete, we create a synthetic scene with $N = 5$ particles on a 1D line interacting via spring-like forces. This is simple enough to analyze exactly, yet rich enough to demonstrate C-JEPA's key insight: without masking, a model can exploit self-dynamics shortcuts; with masking, it *must* learn interactions.

```
rng = Random.MersenneTwister(42)

N_objects = 5
slot_dim = 4  # (position, velocity, mass, charge)
T_history = 3
```

```julia
T_future = 2
T_total = T_history + T_future

function simulate_scene(rng, N; steps=5, dt=0.1)
    positions = randn(rng, N) * 2.0
    velocities = randn(rng, N) * 0.5
    masses = ones(N)
    charges = rand(rng, [-1.0, 1.0], N)

    trajectory = []
    for t in 1:steps
        state = hcat(positions, velocities, masses, charges)  # N × 4
        push!(trajectory, state)

        # Pairwise forces (Coulomb-like)
        forces = zeros(N)
        for i in 1:N, j in 1:N
            i == j && continue
            dx = positions[j] - positions[i]
            forces[i] += charges[i] * charges[j] * sign(dx) / (abs(dx)^2 + 0.1)
        end

        velocities = velocities .+ forces ./ masses .* dt
        positions = positions .+ velocities .* dt
    end
    return trajectory
end

trajectories = [simulate_scene(rng, N_objects; steps=T_total) for _ in 1:32]
println("Generated $(length(trajectories)) trajectories, each with $(T_total) frames")
println("Frame shape: $(size(trajectories[1][1])) = ($N_objects objects × $slot_dim features)")

# Show the first trajectory's positions over time
traj1 = trajectories[1]
println("\nEntity positions over time (trajectory 1):")
for t in 1:T_total
    pos = round.(traj1[t][:, 1]; digits=3)
    println("  t=$t: $pos")
end
```

```
Generated 32 trajectories, each with 5 frames
Frame shape: (5, 4) = (5 objects × 4 features)
```

```
Entity positions over time (trajectory 1):
  t=1: [2.421, -0.158, 0.807, 0.58, -0.134]
  t=2: [2.461, 0.016, 0.715, 0.583, -0.251]
  t=3: [2.51, 0.171, 0.508, 0.631, -0.289]
  t=4: [2.567, 0.371, 0.324, 0.548, -0.273]
  t=5: [2.632, 0.527, 0.282, 0.306, -0.203]
```

Masking and Counterfactual Queries

Object-level masking replaces an entity's history (after $t = 0$) with NaN tokens, preserving only the identity anchor at $t = 0$. In FunctorFlow, this is a do-operation: we intervene on the entity's state, severing its self-dynamics.

```
function mask_entity(trajectory, entity_idx; anchor_frame=1)
    T = length(trajectory)
    masked = [copy(frame) for frame in trajectory]
    anchor = trajectory[anchor_frame][entity_idx, :]

    for t in (anchor_frame + 1):T
        masked[t][entity_idx, :] .= NaN  # mask token placeholder
    end

    return masked, anchor
end

# Mask entity 3 — this is do(entity_3 = mask_token)
masked_traj, anchor = mask_entity(trajectories[1], 3)
println("Identity anchor for entity 3 (t=0): $(round.(anchor; digits=3))")
println("Entity 3 at t=2 after masking: $(masked_traj[2][3, :])")
println("Entity 1 at t=2 (unmasked):     $(round.(masked_traj[2][1, :]; digits=3))")
```

```
Identity anchor for entity 3 (t=0): [0.807, -0.027, 1.0, 1.0]
Entity 3 at t=2 after masking: [NaN, NaN, NaN, NaN]
Entity 1 at t=2 (unmasked):     [2.461, 0.403, 1.0, -1.0]
```

Now we use FunctorFlow's causal machinery to compute the interventional expectation — what is the expected state of entity 3 given the intervention?

```
# The CausalDiagram uses observation_source=:EntityStates and causal_relation=:InteractionGrap
obs_data = Dict{Symbol,Any}(
    :EntityStates => trajectories[1][2][:, 1],    # entity positions at t=2
    :InteractionGraph => Dict("full" => collect(1:N)),  # all entities interact
)

# Bind reducers for the causal diagram's Kan extensions
bind_reducer!(causal_d.base_diagram, :sum,
    (data, relation, meta) -> data)  # identity aggregation for demo
bind_reducer!(causal_d.base_diagram, :first_non_null,
    (data, relation, meta) -> data)  # identity completion for demo

ie_result = interventional_expectation(causal_d, obs_data)
println("Interventional expectation keys: $(keys(ie_result))")
println("Intervention result: $(ie_result[:intervention])")
println("Conditioning result: $(ie_result[:conditioning])")
```

```
Interventional expectation keys: [:conditioning, :intervention, :all_values]
Intervention result: [2.4608197268247225, 0.01601258820176013, 0.7148194917284902, 0.582570452
0.25136721974573484]
Conditioning result: [2.4608197268247225, 0.01601258820176013, 0.7148194917284902, 0.582570452
0.25136721974573484]
```

We can compile and execute the influence neighborhood diagram with concrete implementations:

```
# Bind reducer with correct 3-arg signature: (data, relation, metadata)
bind_reducer!(D_influence, :first_non_null,
    (data, relation, meta) -> data)  # passthrough for symbolic demo

compiled_influence = compile_to_callable(D_influence)

# Compute the context: all entities except the masked one
context_states = masked_traj[2][[1, 2, 4, 5], :]  # exclude entity 3
interaction_mask = [true, true, true, true]  # all context entities participate

result = FunctorFlow.run(compiled_influence, Dict(
    :MaskedEntity => fill(NaN, slot_dim),
    :ContextEntities => context_states,
    :InteractionRelation => interaction_mask
))
println("Influence neighborhood result keys: $(keys(result.values))")
```

```
predicted = result.values[:infer_from_context]
println("Predicted entity 3 state (from context): $(round.(predicted; digits=3))")
```

```
Influence neighborhood result keys: [:InteractionRelation, :infer_from_context, :MaskedEntity,
Predicted entity 3 state (from context): [2.461 0.403 1.0 -1.0; 0.016 1.74 1.0 1.0; 0.583 0.02
0.251 -1.173 1.0 1.0]
```

Counterfactual Reasoning via Σ ? Δ Composition

The key categorical insight: counterfactual = intervene (Σ) then condition (Δ).

"What would entity i's trajectory be if entity j had been removed from the scene?"

This is the composition $\Delta_Q \circ \Sigma_J$: first push forward along the intervention relation J (removing entity j), then complete along the query relation Q (inferring entity i's state in the counterfactual world).

```
D_counterfactual = Diagram(:Counterfactual)

add_object!(D_counterfactual, :ObservedScene, kind=:hidden_state,
            description="Full observed scene state")
add_object!(D_counterfactual, :InterventionRelation, kind=:relation,
            description="Remove entity j from the interaction graph")
add_object!(D_counterfactual, :IntervenedScene, kind=:hidden_state,
            description="Scene without entity j's influence")
add_object!(D_counterfactual, :QueryRelation, kind=:relation,
            description="Select entity i from the intervened scene")
add_object!(D_counterfactual, :CounterfactualState, kind=:hidden_state,
            description="Entity i's state in the counterfactual world")

# Step 1: Σ (intervene — aggregate scene without entity j)
Σ(D_counterfactual, :ObservedScene;
  along=:InterventionRelation,
  name=:intervene,
  target=:IntervenedScene,
  reducer=:sum)

# Step 2: Δ (condition — infer entity i from the intervened scene)
Δ(D_counterfactual, :IntervenedScene;
  along=:QueryRelation,
  name=:condition_query,
```

```
  target=:CounterfactualState,
  reducer=:first_non_null)

println(summary(D_counterfactual))
```

```
FunctorFlow.Diagram
```

Execute the counterfactual query: “What would entity 3 do if entity 1 were removed?”

```
# Bind reducers with correct 3-arg signatures
bind_reducer!(D_counterfactual, :sum,
    (data, relation, meta) -> data)  # passthrough (symbolic demo)
bind_reducer!(D_counterfactual, :first_non_null,
    (data, relation, meta) -> data)  # passthrough (symbolic demo)

compiled_cf = compile_to_callable(D_counterfactual)

# Observed scene at t=2
observed_scene = trajectories[1][2]
# Intervention: remove entity 1 (keep entities 2,3,4,5)
intervened = observed_scene[[2, 3, 4, 5], :]
# Query: select entity 3 from the counterfactual scene
query_state = intervened[2, :]  # entity 3 is at index 2 after removal

cf_result = FunctorFlow.run(compiled_cf, Dict(
    :ObservedScene => observed_scene,
    :InterventionRelation => [false, true, true, true, true],  # remove entity 1
    :QueryRelation => [false, true, false, false],  # select entity 3
))

println("Counterfactual query: 'What if entity 1 were removed?'")
println("  Entity 3 observed state:       $(round.(observed_scene[3, :]; digits=3))")
println("  Entity 3 counterfactual state:  $(round.(query_state; digits=3))")
println("  Causal effect of entity 1 on 3: $(round.(observed_scene[3, :] .- query_state; digit
```

```
Counterfactual query: 'What if entity 1 were removed?'
  Entity 3 observed state:       [0.715, -0.922, 1.0, 1.0]
  Entity 3 counterfactual state:  [0.715, -0.922, 1.0, 1.0]
  Causal effect of entity 1 on 3: [0.0, 0.0, 0.0, 0.0]
```

Topos-Theoretic View: Subobject Classifier for Masking

FunctorFlow's topos module provides a precise account of masking as a subobject classifier. The mask $M \subseteq \{1, \ldots, N\}$ classifies which entities are visible (context) vs. hidden (target). In a topos, this classification is mediated by the characteristic morphism $\chi_M : \{1, \ldots, N\} \to \Omega$, where $\Omega = \{\text{visible}, \text{masked}\}$ is the subobject classifier.

```
sc = SubobjectClassifier(:MaskClassifier;
    truth_values=Set{Symbol}([:visible, :masked]))

# The mask predicate classifies entities by their visibility
mask_pred = InternalPredicate(:is_visible, sc;
    characteristic_map = x -> any(isnan, x) ? :masked : :visible)

# Classify entities in a masked frame
masked_frame = masked_traj[2]  # frame where entity 3 is masked
entity_data = [masked_frame[i, :] for i in 1:N_objects]

println("Entity classification under masking:")
for (i, entity) in enumerate(entity_data)
    label = evaluate_predicate(mask_pred, entity)
    println("  Entity $i: $label")
end
```

```
Entity classification under masking:
  Entity 1: visible
  Entity 2: visible
  Entity 3: masked
  Entity 4: visible
  Entity 5: visible
```

We can also use `classify_subobject` to get the full classification as a dictionary:

```
classification = classify_subobject(sc,
    x -> any(isnan, x) ? :masked : :visible,
    entity_data)
println("Subobject classification: $classification")
```

```
Subobject classification: Dict{Any, Symbol}(5 => Symbol("false"), 4 => Symbol("false"), 2 => S
```

The topos perspective reveals that different masking strategies correspond to different subobject classifiers over the same entity set — and the C-JEPA objective is a sheaf condition: the local predictions (one per masked entity) must glue consistently into a global scene prediction.

Bisimulation: When Two Masking Strategies Are Equivalent

Two masking strategies are bisimilar if they induce the same interaction structure — that is, if the influence neighborhoods they produce are identical. In coalgebraic terms, this means the world models trained under different masks are behaviorally equivalent: they generate the same observable predictions for all future queries.

```
# Create two masking-strategy world models as diagrams
D_mask3 = Diagram(:Mask3Strategy)
add_object!(D_mask3, :State, kind=:hidden_state)
add_object!(D_mask3, :NextState, kind=:hidden_state)
add_morphism!(D_mask3, :dynamics, :State, :NextState,
              description="Dynamics under masking entity 3")
add_coalgebra!(D_mask3, :cjepa_mask_3,
               state=:State, transition=:dynamics, functor_type=:identity)

D_mask4 = Diagram(:Mask4Strategy)
add_object!(D_mask4, :State, kind=:hidden_state)
add_object!(D_mask4, :NextState, kind=:hidden_state)
add_morphism!(D_mask4, :dynamics, :State, :NextState,
              description="Dynamics under masking entity 4")
add_coalgebra!(D_mask4, :cjepa_mask_4,
               state=:State, transition=:dynamics, functor_type=:identity)

# Declare bisimulation: these strategies are equivalent if they produce
# the same influence neighborhood
add_bisimulation!(D_mask3, :masking_equivalence;
    coalgebra_a=:cjepa_mask_3,
    coalgebra_b=:cjepa_mask_4,
    relation=:dynamics,
    description="Masking entities 3 and 4 produces equivalent interaction structure")

bisims = get_bisimulations(D_mask3)
println("Bisimulation declared: $(first(keys(bisims)))")
println("  Coalgebra A: $(first(values(bisims)).coalgebra_a)")
println("  Coalgebra B: $(first(values(bisims)).coalgebra_b)")
println("  Relation:    $(first(values(bisims)).relation)")
```

```
Bisimulation declared: masking_equivalence
  Coalgebra A: cjepa_mask_3
  Coalgebra B: cjepa_mask_4
  Relation:    dynamics
```

When two masks are bisimilar, C-JEPA's learned representations will be invariant to the choice between them — the model extracts the same causal structure regardless of which specific entity is masked, provided the influence neighborhoods are isomorphic.

EMA Target Encoder as Frozen Coalgebra

C-JEPA uses an exponential moving average (EMA) target encoder to produce stable prediction targets. In coalgebraic terms, the target encoder is a frozen reference coalgebra — its transition dynamics are fixed, providing a stable attractor for the online encoder to converge toward.

```
# Simulate EMA update between online and target parameters
# ema_update! expects iterable-of-arrays (e.g., tuple/vector of parameter arrays)
online_params = [randn(rng, 4, 4), randn(rng, 4)]  # e.g., weight matrix + bias
target_params = [randn(rng, 4, 4), randn(rng, 4)]

dist_before = sum(sum((o .- t).^2) for (o, t) in zip(online_params, target_params))
println("Before EMA update:")
println("  Online-Target distance: $(round(dist_before; digits=4))")

ema_update!(target_params, online_params; decay=0.996)

dist_after = sum(sum((o .- t).^2) for (o, t) in zip(online_params, target_params))
println("After EMA update (decay=0.996):")
println("  Online-Target distance: $(round(dist_after; digits=4))")

# Multiple updates converge the target toward the online encoder
for _ in 1:100
    ema_update!(target_params, online_params; decay=0.996)
end
println("After 100 EMA updates:")
dist_final = sum(sum((o .- t).^2) for (o, t) in zip(online_params, target_params))
println("  Online-Target distance: $(round(dist_final; digits=6))")
```

```
Before EMA update:
  Online-Target distance: 18.2989
After EMA update (decay=0.996):
  Online-Target distance: 18.1528
After 100 EMA updates:
  Online-Target distance: 8.143523
```

The coalgebraic interpretation: each EMA step is a coalgebra morphism that contracts the distance between the online and target state spaces. The fixed point (after many updates) is where the two

coalgebras become bisimilar — the target encoder faithfully represents the online encoder's learned dynamics.

Summary and Correspondence Table

C-JEPA demonstrates that object-level masking is not merely a data augmentation strategy — it is a causal intervention that makes interaction reasoning functionally necessary. FunctorFlow provides the categorical language to make this precise:

C-JEPA	Category Theory	FunctorFlow
Object slots S_t	Objects in Prod category	`product(entities...)`
Object masking	do-intervention	Σ (left-Kan) in `CausalDiagram`
Context (unmasked)	Conditioning	Δ (right-Kan) in `CausalDiagram`
Influence neighborhood	Markov blanket / right-Kan kernel	Δ completion sufficiency
$\mathcal{L}_{\text{history}}$ (reconstruction)	Right-Kan obstruction	`add_obstruction_loss!(:history_loss, ...)`
$\mathcal{L}_{\text{future}}$ (prediction)	Left-Kan obstruction	`add_obstruction_loss!(:future_loss, ...)`
EMA target encoder	Frozen coalgebra reference	`ema_update!`
Identity anchor ($t = 0$)	Initial object	Coalgebra initial state
Interaction Necessity (Thm 1)	Minimality of influence presheaf	Right-Kan kernel sufficiency
Slot Attention	Entity decomposition	Entity sub-diagrams
Counterfactual query	$\Sigma \circ \Delta$ composition	`intervene` then `condition`
Mask predicate	Subobject classifier	`SubobjectClassifier` in topos.jl
Masking equivalence	Bisimulation of coalgebras	`add_bisimulation!`

The key insight is that C-JEPA's empirical success (~20% improvement on counterfactual VQA, 8× faster MPC planning) has a precise categorical explanation: object-level masking enforces the right universal property — it forces the predictor to factor through the influence neighborhood, which is exactly the causal structure needed for counterfactual reasoning and efficient planning.