JEPA as Categorical World Model

Joint Embedding Predictive Architecture via Coalgebras

FunctorFlow.jl

Table of contents

Introduction

JEPA (Joint Embedding Predictive Architecture) is LeCun's proposed framework for building world models that predict in *representation space* rather than *observation space*. The key insight is that predicting pixel-by-pixel is intractable, but predicting *embeddings* of future states is feasible.

FunctorFlow.jl reveals that JEPA is fundamentally a coalgebraic construction:

JEPA Component	Categorical Analog
Encoder	Coalgebra morphism
Predictor	Endofunctor dynamics
Target encoder (EMA)	Frozen reference coalgebra
Prediction loss	Obstruction to commutativity
World model	F-coalgebra (X □ F(X))

JEPA Component	Categorical Analog
Optimal representation	Final coalgebra (Lambek's lemma)

```
using FunctorFlow
```

Part 1: Coalgebra — World Models as State Transitions

A coalgebra is a pair (X, α : X → F(X)): a state space equipped with a transition structure. This is the categorical formalization of a world model.

```
# Build a simple world model diagram
D = world_model_block(;
    name=:SimpleWorldModel,
    observation_object=:Pixels,
    latent_object=:Embedding,
    encoder_name=:encode,
    dynamics_name=:predict_next,
    decoder_name=:decode,
)

println("Objects: ", join(keys(D.objects), ", "))
println("Operations: ", join(keys(D.operations), ", "))
println()

# The coalgebra structure
coalgebras = get_coalgebras(D)
for (name, c) in coalgebras
    println(c)
end
```

```
Objects: Pixels, Embedding
Operations: encode, predict_next, decode, encode_then_predict, autoencoder

Coalgebra(:coalgebra, Embedding →_{identity} Embedding)
```

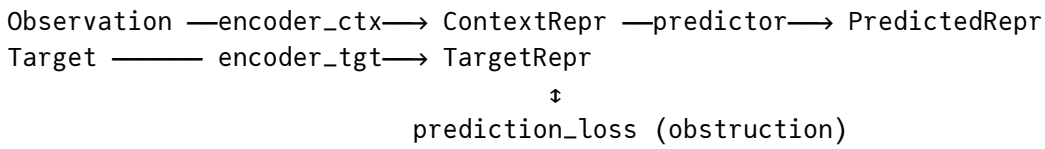
The coalgebra declares that `predict_next` is the world model dynamics: given a latent state, it produces the next latent state.

```
# Bind concrete implementations
bind_morphism!(D, :encode, x -> x ./ 255.0)           # normalize
bind_morphism!(D, :predict_next, x -> x .+ 0.01)      # simple drift
bind_morphism!(D, :decode, x -> x .* 255.0)            # denormalize

compiled = compile_to_callable(D)
result = FunctorFlow.run(compiled, Dict(
    :Pixels => [128.0, 64.0, 200.0],
))

println("Encoded:  ", round.(result.values[:encode]; digits=4))
println("Predicted:", round.(result.values[:predict_next]; digits=4))
println("Decoded:  ", round.(result.values[:autoencoder]; digits=4))
```

```
Encoded:  [0.502, 0.251, 0.7843]
Predicted:[0.512, 0.261, 0.7943]
Decoded:  [128.0, 64.0, 200.0]
```

Part 2: JEPA Block — Prediction in Embedding Space

The JEPA block encodes the fundamental prediction-in-embedding pattern as a FunctorFlow diagram:




```
# Build a JEPA block
D_jepa = jepa_block(;
    name=:ImageJEPA,
    observation_object=:Context,
    target_object=:MaskedRegion,
    context_repr=:CtxEmb,
    target_repr=:TgtEmb,
    comparator=:l2,
)

println("=== JEPA Diagram: $(D_jepa.name) ===")
println("\nObjects:")
```

```
for (name, obj) in D_jepa.objects
    println("  $(name) :: $(obj.kind)")
end
println("\nMorphisms:")
for (name, op) in D_jepa.operations
    if op isa Morphism
        role = get(op.metadata, :role, :unknown)
        net = get(op.metadata, :network, :none)
        detach = get(op.metadata, :detach, false)
        println("  $(name): $(op.source) → $(op.target) [role=$(role), net=$(net), detach=$(d
    elseif op isa Composition
        println("  $(name): $(op.source) → $(op.target) [chain=$(op.chain)]")
    end
end
println("\nLosses:")
for (name, loss) in D_jepa.losses
    println("  $(name): paths=$(loss.paths), comparator=$(loss.comparator)")
end
println("\nCoalgebra:")
for (name, c) in get_coalgebras(D_jepa)
    println("  ", c)
end
```

```
=== JEPA Diagram: ImageJEPA ===

Objects:
  Context :: observation
  MaskedRegion :: observation
  CtxEmb :: representation
  TgtEmb :: representation

Morphisms:
  context_encoder: Context -> CtxEmb [role=encoder, net=online, detach=false]
  target_encoder: MaskedRegion -> TgtEmb [role=encoder, net=target, detach=true]
  predictor: CtxEmb -> TgtEmb [role=predictor, net=none, detach=false]
  prediction_path: Context → TgtEmb [chain=[:context_encoder, :predictor]]

Losses:
  prediction_loss: paths=[(:prediction_path, :target_encoder)], comparator=l2

Coalgebra:
  Coalgebra(:jepa_dynamics, CtxEmb →_{identity} CtxEmb)
```

Part 3: JEPA Execution — The Obstruction Loss

The prediction loss IS an obstruction loss: it measures how far the encoder-predictor square is from commuting.

```
# Bind implementations: encoders as linear scaling
bind_morphism!(D_jepa, :context_encoder, x -> x .* 0.5)
bind_morphism!(D_jepa, :target_encoder, x -> x .* 0.5)  # same encoder (EMA would make it clos
bind_morphism!(D_jepa, :predictor, identity)  # trivial predictor

compiled_jepa = compile_to_callable(D_jepa)

# Case 1: Context and target are the same → perfect prediction
result1 = FunctorFlow.run(compiled_jepa, Dict(
    :Context => [1.0, 2.0, 3.0],
    :MaskedRegion => [1.0, 2.0, 3.0],  # same input
))
println("Case 1 (same input): loss = ", round(result1.losses[:prediction_loss]; digits=6))
println("  Predicted: ", result1.values[:prediction_path])
println("  Target:    ", result1.values[:target_encoder])

# Case 2: Different context and target → non-zero loss
result2 = FunctorFlow.run(compiled_jepa, Dict(
    :Context => [1.0, 2.0, 3.0],
    :MaskedRegion => [4.0, 5.0, 6.0],  # different!
))
println("\nCase 2 (different input): loss = ", round(result2.losses[:prediction_loss]; digits=
println("  Predicted: ", result2.values[:prediction_path])
println("  Target:    ", result2.values[:target_encoder])
```

```
Case 1 (same input): loss = 0.0
  Predicted: [0.5, 1.0, 1.5]
  Target:    [0.5, 1.0, 1.5]

Case 2 (different input): loss = 2.598076
  Predicted: [0.5, 1.0, 1.5]
  Target:    [2.0, 2.5, 3.0]
```

When the loss is zero, the JEPA square commutes — the encoder is an exact coalgebra morphism.

Part 4: KAN-JEPA — Merging KET and JEPA

KAN-JEPA uses a left Kan extension (Σ) as the predictor, merging the KET attention pattern with JEPA's embedding-space prediction:

```
D_kan = kan_jepa_block(;
    name=:KanJEPA_Demo,
    observation_object=:Tokens,
    target_object=:MaskedTokens,
    context_repr=:TokenEmb,
    target_repr=:MaskedEmb,
    relation_object=:Neighborhood,
    reducer=:mean,
)

println("=== KAN-JEPA Diagram ===")
for (name, op) in D_kan.operations
    if op isa KanExtension
        println("  Σ($(op.source); along=$(op.along)) → $(op.target)  [$(op.reducer)]")
    elseif op isa Morphism
        role = get(op.metadata, :role, :unknown)
        println("  $(name): $(op.source) → $(op.target) [$(role)]")
    end
end
println("\nThis is the natural fusion:")
println("  KET's attention = left-Kan aggregation (Σ)")
println("  JEPA's loss = obstruction to commutativity")
```

```
=== KAN-JEPA Diagram ===
  context_encoder: Tokens → TokenEmb [encoder]
  target_encoder: MaskedTokens → MaskedEmb [encoder]
  Σ(TokenEmb; along=Neighborhood) → MaskedEmb  [mean]

This is the natural fusion:
  KET's attention = left-Kan aggregation (Σ)
  JEPA's loss = obstruction to commutativity
```

Part 5: Hierarchical JEPA (H-JEPA)

H-JEPA uses multiple abstraction levels for multi-scale prediction. Fine levels handle short-range details; coarse levels handle long-range abstract planning.

```
D_hjepa = hjepa_block(;
    name=:MultiScaleJEPA,
    levels=[:fine, :medium, :coarse],
)

println("=== H-JEPA: 3 Abstraction Levels ===")
println("\nObjects per level:")
for level in [:fine, :medium, :coarse]
    obs = Symbol("$(level)__Obs_$(level)")
    repr = Symbol("$(level)__CtxRepr_$(level)")
    println("  $(level): $(obs) → $(repr)")
end

println("\nAbstraction morphisms:")
for (name, op) in D_hjepa.operations
    if op isa Morphism && get(op.metadata, :role, nothing) == :abstraction
        from = op.metadata[:from_level]
        to = op.metadata[:to_level]
        println("  $(name): $(from) → $(to)")
    end
end

println("\nPorts:")
for (name, p) in D_hjepa.ports
    println("  $(name) ($(p.direction)): $(p.ref)")
end
```

```
=== H-JEPA: 3 Abstraction Levels ===

Objects per level:
  fine: fine__Obs_fine → fine__CtxRepr_fine
  medium: medium__Obs_medium → medium__CtxRepr_medium
  coarse: coarse__Obs_coarse → coarse__CtxRepr_coarse

Abstraction morphisms:
  abstract_fine_to_medium: fine → medium
  abstract_medium_to_coarse: medium → coarse

Ports:
  input (INPUT): fine__Obs_fine
  fine_repr (OUTPUT): fine__CtxRepr_fine
  coarse_repr (OUTPUT): coarse__CtxRepr_coarse
```

Part 6: Energy-Based Cost Module

The energy function measures compatibility in representation space. The cost module decomposes into intrinsic (immutable) and trainable components:

$$C(s) = \sum_i u_i \cdot \mathrm{IC}_i(s) + \sum_j v_j \cdot \mathrm{TC}_j(s)$$

```
# Build an energy block with VICReg-style regularization
D_energy = energy_block(;
    name=:JEPAEnergy,
    energy_type=:l2,
    variance_weight=0.5,
    covariance_weight=0.1,
    collapse_strategy=VICREG,
)

cost_mods = get_cost_modules(D_energy)
cm = cost_mods[:cost]
println("=== Energy-Based Cost Module ===")
println("Intrinsic costs:")
for ic in cm.intrinsic_costs
    println("  ", ic)
end
println("\nCollapse prevention: VICReg (variance + covariance regularization)")
```

```
=== Energy-Based Cost Module ===
Intrinsic costs:
  IntrinsicCost(:prediction_cost, type=prediction, weight=1.0)
  IntrinsicCost(:variance_cost, type=variance, weight=0.5)
  IntrinsicCost(:covariance_cost, type=covariance, weight=0.1)

Collapse prevention: VICReg (variance + covariance regularization)
```

Energy Function Implementations

```
x = [1.0, 0.5, -0.3, 0.8]
y = [0.9, 0.6, -0.2, 0.7]

println("L2 energy:       ", round(energy_l2(x, y); digits=6))
```

```
println("Cosine energy:   ", round(energy_cosine(x, y); digits=6))
println("Smooth L1:       ", round(energy_smooth_l1(x, y); digits=6))

# Self-similarity should give zero energy
println("\nSelf-similarity:")
println("  L2(x, x) =     ", energy_l2(x, x))
println("  Cosine(x, x) = ", round(energy_cosine(x, x); digits=10))
```

```
L2 energy:       0.04
Cosine energy:   0.007994
Smooth L1:       0.02

Self-similarity:
  L2(x, x) =     0.0
  Cosine(x, x) = 5.1e-9
```

VICReg Regularization

```
using Random
Random.seed!(42)

# Good representations: diverse (high variance, low correlation)
Z_good = randn(4, 10)  # 4 dims, 10 samples
var_good = variance_regularization(Z_good; gamma=1.0)
cov_good = covariance_regularization(Z_good)

# Collapsed representations: all identical
Z_bad = ones(4, 10)
var_bad = variance_regularization(Z_bad; gamma=1.0)
cov_bad = covariance_regularization(Z_bad)

println("Diverse representations:")
println("  Variance penalty:   ", round(var_good; digits=4))
println("  Covariance penalty: ", round(cov_good; digits=4))
println("\nCollapsed representations:")
println("  Variance penalty:   ", round(var_bad; digits=4), " (high = BAD)")
println("  Covariance penalty: ", round(cov_bad; digits=4))
```

```
Diverse representations:
```

```
  Variance penalty:   0.4483
  Covariance penalty: 0.7101

Collapsed representations:
  Variance penalty:   3.96 (high = BAD)
  Covariance penalty: 0.0
```

Part 7: Lean 4 Proof Certificates

FunctorFlow.jl can generate Lean 4 proof certificates for JEPA diagrams, formally verifying the categorical structure:

```
D_proof = jepa_block(; name=:ProvedJEPA)
add_energy_function!(D_proof, :compat;
                     domain=[:ContextRepr, :TargetRepr], energy_type=:l2)
add_bisimulation!(D_proof, :enc_equiv;
                  coalgebra_a=:jepa_dynamics,
                  coalgebra_b=:jepa_dynamics,
                  relation=:predictor)

cert = render_jepa_certificate(D_proof)
println(cert)
```

```
-- Auto-generated by FunctorFlow.jl
namespace FunctorFlowProofs.Generated.ProvedJEPA

open FunctorFlowProofs in

def exportedDiagram : DiagramDecl := {
  name := "ProvedJEPA",
  objects := ["Observation", "Target", "ContextRepr", "TargetRepr"],
  operations := [{ name := "context_encoder", kind := OperationKind.morphism, refs := ["Observ
  ports := [{ name := "context_input", ref := "Observation", kind := "object", portType := "ob
}

def exportedArtifact : LoweringArtifact := {
  diagram := exportedDiagram,
  resolvedRefs := true,
  portsClosed := true
}
```

```
theorem exportedArtifact_checks : exportedArtifact.check = true := by native_decide

theorem exportedArtifact_sound : exportedArtifact.Sound :=
  LoweringArtifact.sound_of_check exportedArtifact_checks


-- Coalgebra declarations
def coalgebra_jepa_dynamics : CoalgebraDecl := {
  name := "jepa_dynamics",
  state := "ContextRepr",
  transition := "predictor",
  functorType := "identity"
}

-- JEPA prediction-as-obstruction theorems
/-- The JEPA prediction loss is an obstruction to commutativity
    of the encoder/predictor square. When loss = 0, the square
    commutes and the encoder is an exact coalgebra morphism. -/
theorem jepa_prediction_loss_is_obstruction :
    exportedArtifact.lossIsObstruction "prediction_loss" := by
  exact LoweringArtifact.loss_obstruction_of_check exportedArtifact_checks

/-- When the JEPA prediction loss is zero, the encoder-predictor
    path commutes with the target encoder path, making the
    encoder a coalgebra morphism (structure-preserving map). -/
theorem jepa_exact_implies_coalgebra_morphism
    (h : ∀ l ∈ exportedArtifact.losses, l.value = 0) :
    exportedArtifact.CoalgebraExact := by
  exact LoweringArtifact.coalgebra_exact_of_zero_loss h

-- Bisimulation declarations
def bisim_enc_equiv : BisimulationDecl := {
  name := "enc_equiv",
  coalgebraA := "jepa_dynamics",
  coalgebraB := "jepa_dynamics",
  relation := "predictor"
}

/-- Two coalgebras are bisimilar iff they map to the same element
    in the final coalgebra — behavioral equivalence. -/
theorem bisimilar_iff_final_coalgebra_equal
    (A B : CoalgebraDecl) (R : BisimulationDecl)
    (h : R.isBisimulation A B) :
```

```
    A.finalImage = B.finalImage :=
  CoalgebraDecl.bisim_implies_final_eq h

-- Energy function declarations
def energy_compat : EnergyDecl := {
  name := "compat",
  domain := ["ContextRepr", "TargetRepr"],
  energyType := "l2"
}

/-- Energy functions are non-negative for L2 and cosine types. -/
theorem energy_nonneg (e : EnergyDecl)
    (h : e.energyType ∈ ["l2", "cosine"]) :
    0 ≤ e.evaluate := by
  exact EnergyDecl.nonneg_of_standard h

end FunctorFlowProofs.Generated.ProvedJEPA
```

Key theorems generated: - **`jepa_prediction_loss_is_obstruction`**: The JEPA loss IS an obstruction to commutativity - **`jepa_exact_implies_coalgebra_morphism`**: Zero loss ⇒ encoder preserves structure - **`bisimilar_iff_final_coalgebra_equal`**: Bisimilar encoders are behaviorally equivalent - **`energy_nonneg`**: Energy functions are non-negative

Part 8: EMA Update (Collapse Prevention)

JEPA prevents representation collapse by using an exponential moving average (EMA) of the online encoder as the target encoder:

```
# Simulate EMA training dynamics
target_params = [Float32[0.1, 0.2, 0.3]]
online_params = [Float32[0.5, 0.6, 0.7]]

println("Before EMA:")
println("  Target: ", target_params[1])
println("  Online: ", online_params[1])

for step in 1:5
    ema_update!(target_params, online_params; decay=0.99)
end

println("\nAfter 5 EMA steps (decay=0.99):")
```

```
println("  Target: ", round.(target_params[1]; digits=4))
println("  Online: ", online_params[1], " (unchanged)")
println("\nTarget slowly tracks online → prevents collapse without negatives")
```

```
Before EMA:
  Target: Float32[0.1, 0.2, 0.3]
  Online: Float32[0.5, 0.6, 0.7]

After 5 EMA steps (decay=0.99):
  Target: Float32[0.1196, 0.2196, 0.3196]
  Online: Float32[0.5, 0.6, 0.7] (unchanged)

Target slowly tracks online → prevents collapse without negatives
```

Summary

Construction	FunctorFlow Type	Purpose
Coalgebra	`Coalgebra`	World model (state → F(state))
Coalgebra morphism	`CoalgebraMorphism`	Structure-preserving encoder
Final coalgebra	`FinalCoalgebraWitness`	Optimal representation (Lambek)
Bisimulation	`Bisimulation`	Behavioral equivalence of encoders
JEPA block	`jepa_block()`	Encoder/predictor/target triple
KAN-JEPA	`kan_jepa_block()`	Σ-attention as JEPA predictor
H-JEPA	`hjepa_block()`	Multi-scale nested coalgebras
Energy function	`EnergyFunction`	Compatibility measure in repr space
Cost module	`CostModule`	IC + TC decomposition
EMA update	`ema_update!()`	Collapse prevention

The key insight: JEPA's prediction loss is an *obstruction to diagram commutativity*. Minimizing it drives the encoder toward being an exact coalgebra morphism — a structure-preserving map between the observation world model and the latent world model.