Lux Neural Backend

Differentiable diagrams with Lux.jl

Simon Frost

Table of contents

Introduction

FunctorFlow.jl provides a Lux-based neural backend that compiles categorical diagrams into differentiable Lux models. This gives you:

- DiagramDenseLayer — dense linear morphisms
- DiagramChainLayer — composed sequences of layers
- KETAttentionLayer — single- and multi-head attention for Kan Extension Templates
- compile_to_lux — end-to-end compilation of a `Diagram` to a `LuxDiagramModel`
- Neural comparators — differentiable obstruction loss comparators

The result is a standard Lux model that participates in the Lux ecosystem: you can extract parameters, compute gradients, and train with any Lux-compatible optimizer.

Setup

```
using FunctorFlow
using Lux
using LuxCore
using Random

rng = Random.MersenneTwister(42)
```

DiagramDenseLayer

A `DiagramDenseLayer` wraps a standard dense (affine) transformation as a FunctorFlow morphism layer. It maps an input dimension to an output dimension with learnable weights and biases.

```
dense = FunctorFlow.DiagramDenseLayer(8, 4)
```

```
DiagramDenseLayer(:dense, 8, 4, identity)  # 36 parameters
```

Initialize parameters and state, then run a forward pass.

```
ps, st = Lux.setup(rng, dense)
x = randn(rng, Float32, 8, 2)  # 8 features, batch of 2
y, st_new = dense(x, ps, st)
println("Input size:  ", size(x))
println("Output size: ", size(y))
```

```
Input size:  (8, 2)
Output size: (4, 2)
```

KETAttentionLayer

The `KETAttentionLayer` implements the attention mechanism used in the KET (Kan Extension Template) pattern. It supports both single-head and multi-head configurations.

Single-head attention

The layer internally computes Q, K, V projections from a single `source` input of shape `(d_model, seq_len[, batch])`.

```
attn = FunctorFlow.KETAttentionLayer(16; n_heads=1)
ps_a, st_a = Lux.setup(rng, attn)
source = randn(rng, Float32, 16, 5)  # d_model × seq_len
out_a, st_a2 = attn(source, ps_a, st_a)
println("Attention output size: ", size(out_a))
```

```
Attention output size: (16, 5)
```

Multi-head attention

```
mha = FunctorFlow.KETAttentionLayer(16; n_heads=4)
ps_m, st_m = Lux.setup(rng, mha)
out_m, st_m2 = mha(source, ps_m, st_m)
println("Multi-head output size: ", size(out_m))
```

```
Multi-head output size: (16, 5)
```

Masked attention

You can pass `(source, mask)` as a tuple to prevent attention over certain positions (e.g., padding or future tokens). The mask has shape `(seq_len, seq_len)` where `true` (1) entries allow attention.

```
mask = ones(Float32, 5, 5)
mask[4:5, :] .= 0f0  # mask out positions 4 and 5
out_masked, _ = attn((source, mask), ps_a, st_a)
println("Masked output size: ", size(out_masked))
```

```
Masked output size: (16, 5)
```

compile_to_lux

The main entry point is `compile_to_lux`, which takes a `Diagram` and returns a `LuxDiagramModel`. Let's compile a KET block.

```
D = ket_block(; name=:NeuralKET, reducer=:ket_attention)
```

```
Diagram :NeuralKET
  Objects:
    Values::messages
    Incidence::relation
    ContextualizedValues::contextualized_messages
  Operations:
    aggregate = Σ(Values, along=Incidence, target=ContextualizedValues, reducer=:ket_attention
  Ports:
    → input (messages)
    → relation (relation)
    ← output (contextualized_messages)
```

Compile the diagram into a Lux model, providing a `KETAttentionLayer` for the reducer.

```
model = compile_to_lux(D;
    reducer_layers=Dict(
        :ket_attention => FunctorFlow.KETAttentionLayer(8; n_heads=1)
    )
)
```

```
LuxDiagramModel(Diagram :NeuralKET ⟨3 objects, 0 morphisms, 1 Kan, 0 losses⟩, CompiledDiagram
trainable
```

Set up parameters and run a forward pass. The `LuxDiagramModel` returns a `(result_dict, new_state)` tuple where `result_dict` has `:values` and `:losses` keys.

```
ps_ket, st_ket = Lux.setup(rng, model)

inputs = Dict(
    :Values => randn(rng, Float32, 8, 3),
    :Incidence => Float32.(ones(3, 3))
)

result_ket, st_ket2 = model(inputs, ps_ket, st_ket)
println("KET result keys: ", collect(keys(result_ket[:values])))
```

```
KET result keys: [:aggregate, :Incidence, :Values, :ContextualizedValues]
```

DB Square with Neural Morphisms

Compiling a DB square with neural morphisms gives you a model whose obstruction loss is differentiable — you can backpropagate through it.

```
db = db_square(; name=:NeuralDB)

model_db = compile_to_lux(db;
    morphism_layers=Dict(
        :f ⇒ FunctorFlow.DiagramDenseLayer(4, 4),
        :g ⇒ FunctorFlow.DiagramDenseLayer(4, 4)
    )
)
```

```
LuxDiagramModel(Diagram :NeuralDB ⟨1 objects, 2 morphisms, 0 Kan, 1 losses⟩, CompiledDiagram :
```

The default `:l2` comparator is automatically replaced with the differentiable `neural_l2_comparator`.

```
ps_db, st_db = Lux.setup(rng, model_db)

inputs_db = Dict(:State ⇒ randn(rng, Float32, 4, 2))
result_db, st_db2 = model_db(inputs_db, ps_db, st_db)
println("DB values: ", collect(keys(result_db[:values])))
println("DB losses: ", result_db[:losses])
```

```
DB values: [:State, :f, :p2, :g, :p1]
DB losses: Dict{Symbol, Any}(:obstruction => 7.485126495361328)
```

The losses are computed by the neural comparator and participate in the computation graph, so gradients flow through both the morphism layers and the comparator.

GT Neighborhood Model

The GT (Graph Transformer) neighborhood pattern combines a lift morphism (node□edge features) with a KET-style aggregation using attention.

```
gt = gt_neighborhood_block(; name=:NeuralGT, reducer=:ket_attention)

model_gt = compile_to_lux(gt;
    morphism_layers=Dict(
        :lift ⇒ FunctorFlow.DiagramDenseLayer(8, 8)
    ),
    reducer_layers=Dict(
        :ket_attention ⇒ FunctorFlow.KETAttentionLayer(8; n_heads=2)
    )
)
```

```
LuxDiagramModel(Diagram :NeuralGT ⟨4 objects, 1 morphisms, 1 Kan, 0 losses⟩, CompiledDiagram :
trainable
```

```
ps_gt, st_gt = Lux.setup(rng, model_gt)

inputs_gt = Dict(
    :Tokens ⇒ randn(rng, Float32, 8, 4),
    :Incidence ⇒ Float32.(ones(4, 4))
)

result_gt, st_gt2 = model_gt(inputs_gt, ps_gt, st_gt)
println("GT result keys: ", collect(keys(result_gt[:values])))
```

```
GT result keys: [:aggregate, :ContextualizedTokens, :lift, :Incidence, :EdgeMessages, :Tokens]
```

Convenience Builders

The Lux extension provides convenience functions that create both the diagram and the Lux model in one step.

KET model

`build_ket_lux_model(d_model)` returns `(model, diagram)`.

```
ket_model, ket_diag = FunctorFlow.build_ket_lux_model(8; n_heads=1)
ps_km, st_km = Lux.setup(rng, ket_model)
println("KET model type: ", typeof(ket_model))
```

```
KET model type: LuxDiagramModel
```

DB model

```
db_model, db_diag = FunctorFlow.build_db_lux_model(4)
ps_dm, st_dm = Lux.setup(rng, db_model)
println("DB model type: ", typeof(db_model))
```

```
DB model type: LuxDiagramModel
```

GT model

```
gt_model, gt_diag = FunctorFlow.build_gt_lux_model(8; n_heads=2)
ps_gm, st_gm = Lux.setup(rng, gt_model)
println("GT model type: ", typeof(gt_model))
```

```
GT model type: LuxDiagramModel
```

CATAGI block backends

```
topo_model, topo_diag = FunctorFlow.build_topocoend_lux_model(8; n_heads=2)
horn_model, horn_diag = FunctorFlow.build_horn_lux_model(8)
bisim_model, bisim_diag = FunctorFlow.build_bisimulation_quotient_lux_model(8)

println("TopoCoend model type: ", typeof(topo_model))
println("Horn model type: ", typeof(horn_model))
println("Bisimulation quotient model type: ", typeof(bisim_model))
```

```
TopoCoend model type: LuxDiagramModel
Horn model type: LuxDiagramModel
Bisimulation quotient model type: LuxDiagramModel
```

The `build_topocoend_lux_model` helper uses a dedicated `RelationInferenceLayer` to construct a soft relation before the Kan aggregation:

```
ps_topo, st_topo = Lux.setup(rng, topo_model)
seq_len = 4
topo_inputs = Dict(
    topo_diag.ports[:input].ref => randn(rng, Float32, 8, seq_len)
)
topo_result, st_topo = topo_model(topo_inputs, ps_topo, st_topo)
println("Learned relation size: ", size(topo_result[:values][topo_diag.ports[:learned_relation
println("TopoCoend output size: ", size(topo_result[:values][topo_diag.ports[:output].ref]))
```

```
Learned relation size: (4, 4)
TopoCoend output size: (8, 4)
```

These CATAGI builders are the differentiable companions to the symbolic walkthroughs in:

- 18 Neurosymbolic Pipelines
- 23 TopoCoend Triage
- 24 Bisimulation Quotient

Mixed Neural/Symbolic

A key strength of the FunctorFlow Lux backend is that you can mix neural morphisms (Lux layers with learnable parameters) and symbolic morphisms (plain Julia functions) in the same diagram. The compiler handles the routing automatically.

```
D_mixed = Diagram(:MixedModel)
add_object!(D_mixed, :Raw; kind=:input, shape="(8,)")
add_object!(D_mixed, :Encoded; kind=:latent, shape="(4,)")
add_object!(D_mixed, :Normalized; kind=:latent, shape="(4,)")
add_object!(D_mixed, :Output; kind=:output, shape="(4,)")

# Neural morphism: learned encoder
add_morphism!(D_mixed, :encode, :Raw, :Encoded;
    description="Learned encoder")

# Symbolic morphism: deterministic normalization
add_morphism!(D_mixed, :normalize, :Encoded, :Normalized;
    description="L2 normalization")
bind_morphism!(D_mixed, :normalize,
    x -> x ./ sqrt.(sum(x .^ 2; dims=1) .+ 1f-8))
```

```
# Neural morphism: learned decoder
add_morphism!(D_mixed, :decode, :Normalized, :Output;
    description="Learned decoder")

compose!(D_mixed, :encode, :normalize, :decode; name=:pipeline)
```

```
pipeline = encode ⋅ normalize ⋅ decode
```

Compile with neural layers only for the unbound morphisms.

```
model_mixed = compile_to_lux(D_mixed;
    morphism_layers=Dict(
        :encode => FunctorFlow.DiagramDenseLayer(8, 4),
        :decode => FunctorFlow.DiagramDenseLayer(4, 4)
    )
)

ps_mix, st_mix = Lux.setup(rng, model_mixed)

x_mix = randn(rng, Float32, 8, 3)
result_mix, st_mix2 = model_mixed(
    Dict(:Raw => x_mix), ps_mix, st_mix
)
println("Pipeline output size: ", size(result_mix[:values][:pipeline]))
```

```
Pipeline output size: (4, 3)
```

The `:normalize` morphism uses the bound Julia function (no parameters), while `:encode` and `:decode` use learnable `DiagramDenseLayer` instances. Gradients flow through the symbolic normalization via standard automatic differentiation.

Related Docs

- 04 Block Library for the reusable symbolic builders and the CATAGI block guide.
- 18 Neurosymbolic Pipelines for the symbolic versions of the same planning / TopoCoend / horn / bisimulation patterns.
- 23 TopoCoend Triage and 24 Bisimulation Quotient for focused single-block walkthroughs.