Predict-Detach Regimes

Comparing causal, noncausal, and predict-detach KET aggregation

Simon Frost

Table of contents

Introduction

The predict-detach regime is a key insight from FunctorFlow: by varying the relation morphism in a Kan extension, we can switch between causal, noncausal, and predict-detach aggregation — three fundamentally different ways an architecture processes sequential data.

This vignette:

1. Builds FunctorFlow diagrams for each regime
2. Implements a KET model using FunctorFlow's `KETAttentionLayer`
3. Trains all three regimes on a synthetic copy task
4. Compares loss curves

This is a Julia port of the Python `predict_detach_demo.py`.

Setup

```
using FunctorFlow
using Lux
using LuxCore
using Random
using Statistics
using Optimisers
using Printf

# CPU device; for Apple Silicon GPU, use `using Metal; const dev = Lux.gpu_device()`
const dev = identity
println("Device: CPU")
```

The Synthetic Copy Task

We generate sequences where `seq[t+1] = seq[t + offset_k]` — the next token is a copy of the token `k` positions ahead. This means:

- A causal model (only sees past) cannot solve the task
- A noncausal model (sees the future) can look ahead and cheat
- The predict-detach regime uses predicted future tokens (detached from gradients)

```
const VOCAB_SIZE = 50
const SEQ_LENGTH = 16
const OFFSET_K = 5
const EMBED_DIM = 64
const BATCH_SIZE = 64
const NUM_STEPS = 100

function make_copy_data(n_seq; length=SEQ_LENGTH, vocab_size=VOCAB_SIZE,
                        offset_k=OFFSET_K, rng=Random.default_rng())
    seqs = rand(rng, 0:vocab_size-1, length, n_seq)
    for j in 1:n_seq
        for t in 1:length-offset_k-1
            seqs[t+1, j] = seqs[t+offset_k, j]
        end
    end
    seqs
end
```

```
rng = Random.MersenneTwister(42)
train_data = make_copy_data(3000; rng=rng)
test_data = make_copy_data(500; rng=rng)
println("Train: ", size(train_data), "  Test: ", size(test_data))
println("Example: ", train_data[:, 1])
```

```
Train: (16, 3000)  Test: (16, 500)
Example: [10, 35, 48, 37, 10, 5, 9, 18, 31, 45, 8, 18, 31, 45, 8, 45]
```

Relation Masks

The key categorical insight: different relation morphisms define different neighborhoods for the Kan extension.

```
causal_mask(len) = Float32.([i <= j ? 1.0f0 : 0.0f0 for i in 1:len, j in 1:len])
noncausal_mask(len) = ones(Float32, len, len)

println("Causal mask (6×6):")
display(causal_mask(6))
```

```
Causal mask (6×6):

6×6 Matrix{Float32}:
 1.0  1.0  1.0  1.0  1.0  1.0
 0.0  1.0  1.0  1.0  1.0  1.0
 0.0  0.0  1.0  1.0  1.0  1.0
 0.0  0.0  0.0  1.0  1.0  1.0
 0.0  0.0  0.0  0.0  1.0  1.0
 0.0  0.0  0.0  0.0  0.0  1.0
```

FunctorFlow Diagram Structure

All three regimes share the same categorical blueprint — they differ only in the relation morphism and source object:

```
D_causal = @functorflow CausalKET begin
    Tokens :: messages
    CausalRelation :: relation
end
Σ(D_causal, :Tokens; along=:CausalRelation, reducer=:ket_attention, name=:agg)

D_noncausal = @functorflow NoncausalKET begin
    Tokens :: messages
    FullRelation :: relation
end
Σ(D_noncausal, :Tokens; along=:FullRelation, reducer=:ket_attention, name=:agg)

D_pd = @functorflow PredictDetachKET begin
    PredictedTokens :: predicted_messages
    FullRelation :: relation
end
Σ(D_pd, :PredictedTokens; along=:FullRelation, reducer=:ket_attention, name=:agg)

for (name, D) in [("Causal", D_causal), ("Noncausal", D_noncausal), ("Predict-Detach", D_pd)]
    acs = to_acset(D)
    println("$name: $(nparts(acs, :Node)) nodes, $(nparts(acs, :Edge)) edges, $(nparts(acs, :K
end
```

```
Causal: 2 nodes, 0 edges, 1 Σ
Noncausal: 2 nodes, 0 edges, 1 Σ
Predict-Detach: 2 nodes, 0 edges, 1 Σ
```

KET Model

We build a model using FunctorFlow's `KETAttentionLayer` — the neural implementation of Σ (left Kan extension via scaled dot-product attention).

```
struct CopyModel <: LuxCore.AbstractLuxContainerLayer{(:tok_emb, :pos_emb, :ket, :head)}
    tok_emb::Lux.Embedding
    pos_emb::Lux.Embedding
    ket::KETAttentionLayer
    head::Dense
    offset_k::Int
end

function CopyModel(; vocab=VOCAB_SIZE, dim=EMBED_DIM, seqlen=SEQ_LENGTH, offset_k=OFFSET_K)
```

```
    CopyModel(
        Lux.Embedding(vocab => dim),
        Lux.Embedding(seqlen => dim),
        KETAttentionLayer(dim; n_heads=1, name=:ket_reducer),
        Dense(dim => vocab),
        offset_k
    )
end
```

```
CopyModel
```

The forward pass selects the regime by choosing the mask and source:

```
using NNlib: softmax

function forward(m::CopyModel, token_ids, ps, st, regime::Symbol)
    seq_len, batch = size(token_ids)

    tok, st_t = m.tok_emb(token_ids .+ 1, ps.tok_emb, st.tok_emb)
    pos_ids = repeat(1:seq_len, 1, batch) |> dev
    pos, st_p = m.pos_emb(pos_ids, ps.pos_emb, st.pos_emb)
    hidden = tok .+ pos

    if regime == :causal
        mask = causal_mask(seq_len) |> dev
        source = hidden
    elseif regime == :leaky_noncausal
        mask = noncausal_mask(seq_len) |> dev
        source = hidden
    elseif regime == :predict_detach
        mask = noncausal_mask(seq_len) |> dev
        logits0, _ = m.head(hidden, ps.head, st.head)
        source = predict_detach_source(logits0, ps.tok_emb.weight; position_bias=pos)
    else
        error("Unknown regime: $regime")
    end

    pooled, st_k = m.ket((source, mask), ps.ket, st.ket)

    # Offset-k leak for noncausal regimes (Zygote-safe: pad + slice, no mutation)
    if regime in (:leaky_noncausal, :predict_detach) && seq_len > m.offset_k
```

```
        d = size(source, 1)
        pad = zeros(Float32, d, m.offset_k, batch) ▷ dev
        leak = cat(pad, source[:, m.offset_k+1:seq_len, :]; dims=2)
        pooled = pooled .+ leak
    end

    logits, st_h = m.head(pooled, ps.head, st.head)
    new_st = (tok_emb=st_t, pos_emb=st_p, ket=st_k, head=st_h)
    logits, new_st
end
```

```
forward (generic function with 1 method)
```

Loss and Evaluation

```
function xent_loss(logits, oh)
    # Cross-entropy via one-hot multiply
    mx = maximum(logits; dims=1)
    lse = mx .+ log.(sum(exp.(logits .- mx); dims=1))
    lp = logits .- lse
    n_tokens = prod(size(logits)[2:end])
    -sum(lp .* oh) / n_tokens
end

function make_onehot(targets_cpu, vocab)
    # Build one-hot on CPU via broadcasting (no mutation)
    Float32.(collect(1:vocab) .== reshape(targets_cpu .+ 1, 1, size(targets_cpu)...)) ▷ dev
end

function eval_model(model, ps, st, data, regime)
    n = min(size(data, 2), BATCH_SIZE)
    batch_cpu = data[:, 1:n]
    batch = batch_cpu ▷ dev
    logits, _ = forward(model, batch[1:end-1, :], ps, st, regime)
    targets_cpu = batch_cpu[2:end, :]
    logits_cpu = Array(logits)
    preds = [ci[1] - 1 for ci in dropdims(argmax(logits_cpu; dims=1); dims=1)]
    acc = Float64(sum(preds .== targets_cpu)) / length(targets_cpu)
    oh = make_onehot(targets_cpu, VOCAB_SIZE)
    loss = Float64(xent_loss(logits, oh))
```

```
    (loss=loss, acc=acc)
end
```

```
eval_model (generic function with 1 method)
```

Training Loop

```
using Zygote

function train_regime(regime::Symbol, train_data, test_data; seed=0)
    rng_t = Random.MersenneTwister(seed)
    model = CopyModel()
    ps, st = Lux.setup(rng_t, model)
    ps = ps |> dev; st = st |> dev
    opt = Optimisers.setup(Optimisers.Adam(2f-3), ps)
    losses = Float64[]
    mask = (regime == :causal ? causal_mask(SEQ_LENGTH-1) : noncausal_mask(SEQ_LENGTH-1)) |> d
    posids = repeat(1:SEQ_LENGTH-1, 1, BATCH_SIZE) |> dev

    for step in 1:NUM_STEPS
        idx = rand(rng_t, 1:size(train_data, 2), BATCH_SIZE)
        batch_cpu = train_data[:, idx]
        ids = batch_cpu[1:end-1, :] |> dev
        oh = make_onehot(batch_cpu[2:end, :], VOCAB_SIZE)

        function loss_fn(ps_)
            tok, _ = model.tok_emb(ids .+ 1, ps_.tok_emb, st.tok_emb)
            pos, _ = model.pos_emb(posids, ps_.pos_emb, st.pos_emb)
            hidden = tok .+ pos
            source = hidden
            if regime == :predict_detach
                lg0, _ = model.head(hidden, ps_.head, st.head)
                source = predict_detach_source(lg0, ps_.tok_emb.weight; position_bias=pos)
            end
            pooled, _ = model.ket((source, mask), ps_.ket, st.ket)
            if regime in (:leaky_noncausal, :predict_detach)
                sl = SEQ_LENGTH - 1
                d = size(source, 1)
                pad = zeros(Float32, d, model.offset_k, BATCH_SIZE) |> dev
                leak = cat(pad, source[:, model.offset_k+1:sl, :]; dims=2)
```

```
                pooled = pooled .+ leak
            end
            logits, _ = model.head(pooled, ps_.head, st.head)
            xent_loss(logits, oh)
        end

        lv, back = Zygote.pullback(loss_fn, ps)
        gs = back(one(lv))[1]
        opt, ps = Optimisers.update(opt, ps, gs)
        push!(losses, Float64(lv))

        if step % 25 == 0
            @printf("  [%s] step %3d: loss=%.4f\n", regime, step, lv)
        end
    end

    ev = eval_model(model, ps, st, test_data, regime)
    (losses=losses, eval_loss=ev.loss, eval_acc=ev.acc)
end

println("Training functions ready.")
```

```
Training functions ready.
```

Train All Regimes

```
results = Dict{Symbol, Any}()
for regime in [:causal, :leaky_noncausal, :predict_detach]
    println("\n═══ Training $regime ═══")
    results[regime] = train_regime(regime, train_data, test_data)
    r = results[regime]
    @printf("  Final: train_loss=%.4f  eval_loss=%.4f  eval_acc=%.1f%%\n",
            r.losses[end], r.eval_loss, 100*r.eval_acc)
end
```

```

═══ Training causal ═══
  [causal] step  25: loss=3.9895
  [causal] step  50: loss=3.9276
```

```
  [causal] step  75: loss=3.9004
  [causal] step 100: loss=3.8126
  Final: train_loss=3.8126  eval_loss=3.7957  eval_acc=11.7%

═══ Training leaky_noncausal ═══
  [leaky_noncausal] step  25: loss=4.0391
  [leaky_noncausal] step  50: loss=3.8386
  [leaky_noncausal] step  75: loss=3.5201
  [leaky_noncausal] step 100: loss=2.2552
  Final: train_loss=2.2552  eval_loss=2.1012  eval_acc=62.6%

═══ Training predict_detach ═══
  [predict_detach] step  25: loss=4.0350
  [predict_detach] step  50: loss=3.9376
  [predict_detach] step  75: loss=3.9525
  [predict_detach] step 100: loss=3.9528
  Final: train_loss=3.9528  eval_loss=3.9310  eval_acc=2.0%
```

Results

```
println("╔══════════════════╦════════════╦═══════════╦══════════╗")
println("║ Regime           ║ Train Loss ║ Eval Loss ║ Eval Acc ║")
println("╠══════════════════╬════════════╬═══════════╬══════════╣")
for regime in [:causal, :leaky_noncausal, :predict_detach]
    r = results[regime]
    name = rpad(string(regime), 16)
    tl = @sprintf("%.4f", r.losses[end])
    el = @sprintf("%.4f", r.eval_loss)
    ea = @sprintf("%.1f%%", 100*r.eval_acc)
    println("║ $name ║ $tl     ║ $el    ║ $ea    ║")
end
println("╚══════════════════╩════════════╩═══════════╩══════════╝")
```

Regime	Train Loss	Eval Loss	Eval Acc
causal	3.8126	3.7957	11.7%
leaky_noncausal	2.2552	2.1012	62.6%
predict_detach	3.9528	3.9310	2.0%

Loss Curves

```
println("Training loss (every 10 steps):")
println("Step | Causal  | Noncausal | Predict-Detach")
println("-----|---------|-----------|---------------")
for step in 10:10:NUM_STEPS
    c = @sprintf("%.4f", results[:causal].losses[step])
    n = @sprintf("%.4f", results[:leaky_noncausal].losses[step])
    p = @sprintf("%.4f", results[:predict_detach].losses[step])
    println("  $(lpad(step,2)) | $c  | $n    | $p")
end
```

```
Training loss (every 10 steps):
Step | Causal  | Noncausal | Predict-Detach
-----|---------|-----------|---------------
  10 | 4.2332  | 4.2335    | 4.2091
  20 | 4.0445  | 4.1064    | 4.0876
  30 | 3.9655  | 3.9693    | 4.0143
  40 | 3.9516  | 3.9026    | 3.9799
  50 | 3.9276  | 3.8386    | 3.9376
  60 | 3.9228  | 3.7417    | 3.9436
  70 | 3.9093  | 3.5813    | 3.9380
  80 | 3.8803  | 3.3409    | 3.9409
  90 | 3.8627  | 2.8549    | 3.9363
  100 | 3.8126  | 2.2552    | 3.9528
```

The Categorical Perspective

All three regimes use the same Σ (left Kan extension) but differ in:

Regime	Relation	Source	Can solve copy task?
Causal	Lower-triangular	Hidden states	? (cannot see future)
Leaky noncausal	Full	Hidden states	? (sees future)
Predict-detach	Full	Predicted embeddings	? (predicted future)

The `KETAttentionLayer` implements Σ as scaled dot-product attention. The relation morphism (causal vs. full mask) determines the attention pattern. The predict-detach regime achieves non-causal accuracy while maintaining causal gradient flow — the predicted basis states are detached from the computation graph.

This demonstrates that architecture = choice of relation morphism in the Kan extension framework.