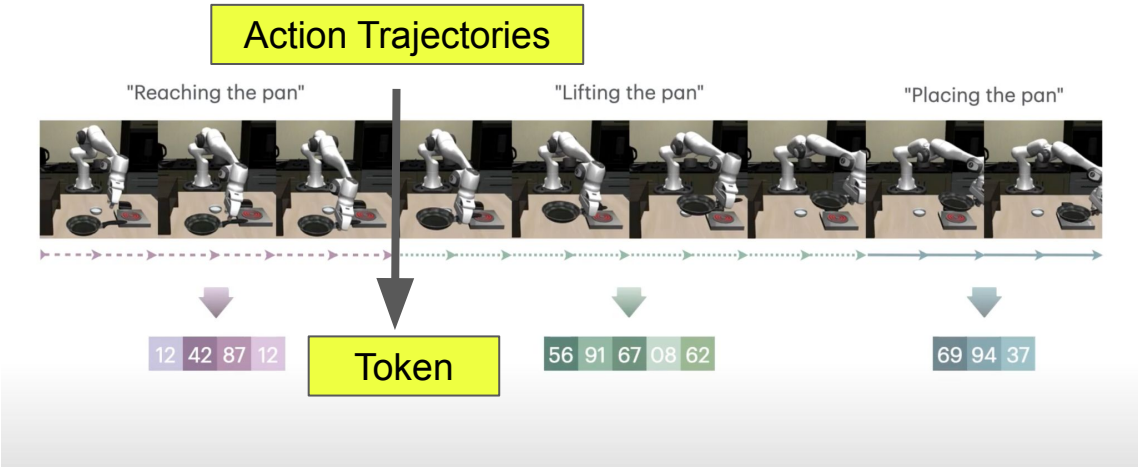
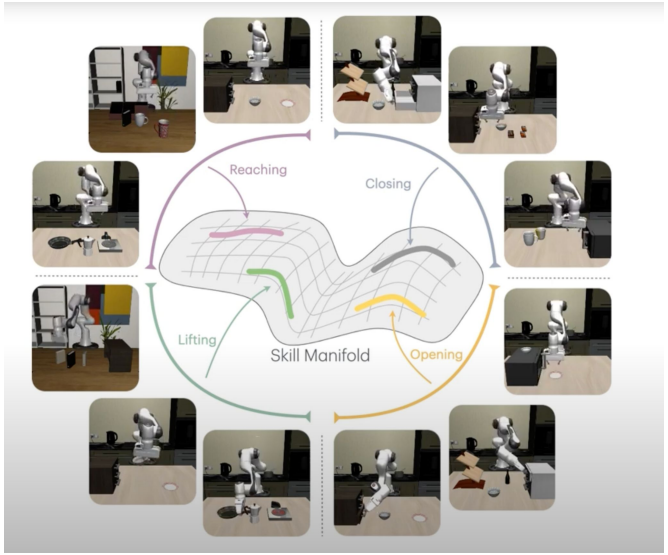
Skill-based Action Tokenizers

Pablo, Owen

Research Objective

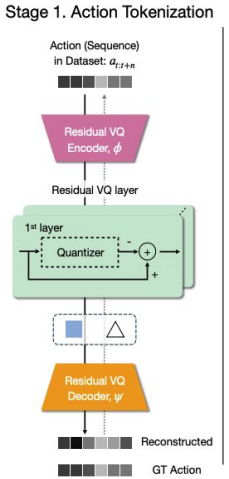
Can we compress a long demonstration into a short list of reusable skills?



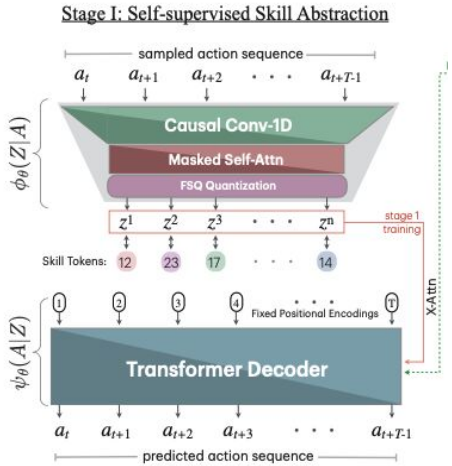


Related Works - **Action Tokenizer**

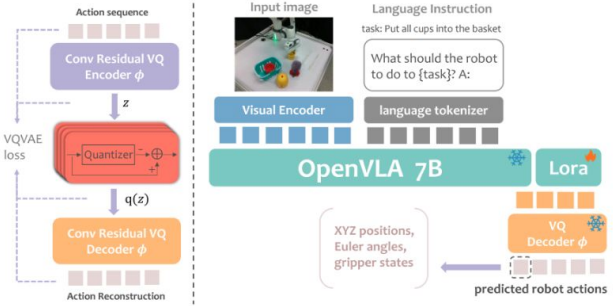
Vq-Bet



Quest



VQ-VLA



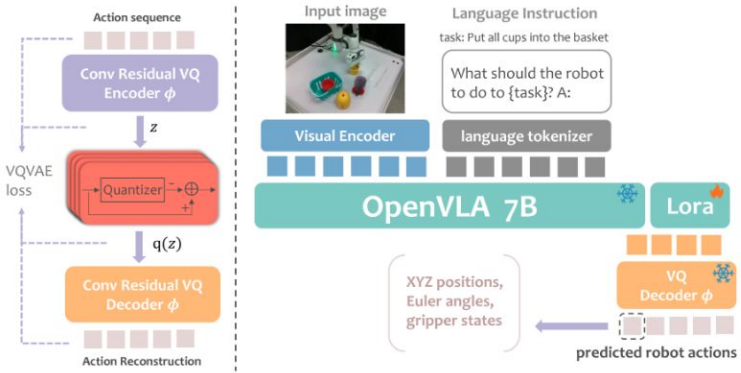
	VQ-BeT ('24)	QueST ('24)	VQ-VLA (ICCV, 25)
1. Compress : Raw snippet → token ID	● Residual VQ-VAE: 2 codebooks jointly trained → up to 64–256 latent combos for each snippet. One token per chunk, but far richer code space.	● Conditional VAE with causal conv → produces a sequence of scalar-quantised codes (FSQ). (Each snippet can get 1–8 tokens, so variable-length skills.)	VQ-VAE Decoder & VLA integration

Limitations of Related Approaches

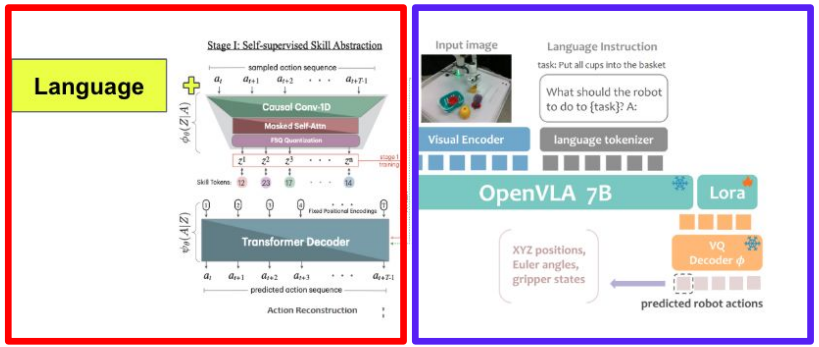
1. **Not conditioned**: Prior methods compress actions in isolation without considering task context or instructions
2. **Lack of proper metrics**: No quantitative way to evaluate if learned tokens are semantically meaningful, reusable

Main Idea

VQ-VLA

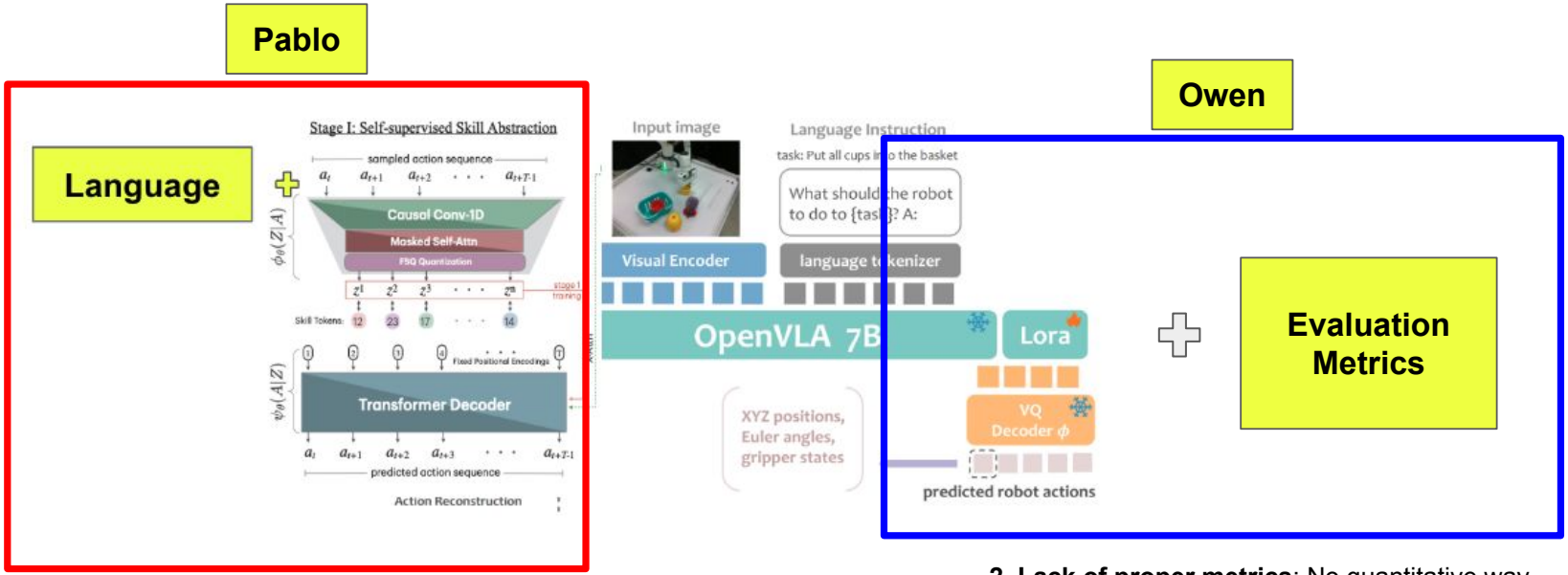


Ours



Quantitative Metric

How we are collaborating



1. Not conditioned: Prior methods compress actions in isolation without considering task context or instructions

2. Lack of proper metrics: No quantitative way to evaluate if learned tokens are semantically meaningful, reusable, or interpretable

+VLA Integration

Future Plans

~11

1. Research
2. Conditioned Architecture
3. Evaluation Metric
(BPA, InfoMEC, MDP-based)

11~

1. QueST-based Architecture
2. VLA Integration
3. Simulation (Libero)

~12

1. Simulation (Libero)
2. Real-robot

~1/26

1. Writing

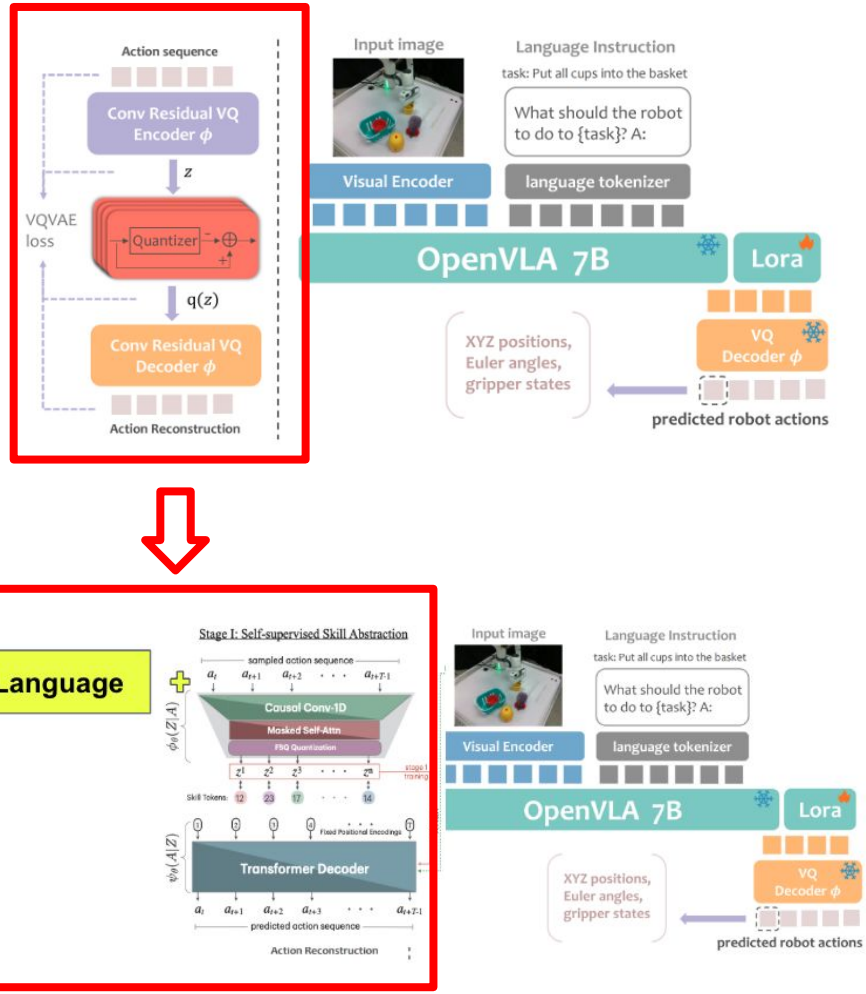
1. Conditioned Skill Action Tokenizer

Why this is needed?

Current VLA tokenizers mainly prioritize training efficiency but
neglect long-horizon scalability and reusability.

Make action tokenizer to generate skill-based action not few action tokens.

> Encode entire motion primitives (e.g., “reach forward smoothly”) as single compressed tokens, enabling coherent multi-step execution.



VQ-VLA: Improving Vision-Language-Action Models via Scaling Vector-Quantized Action Tokenizers

VQ-VLA

Step1

VQ-VAE Training process

Action [B, T, A] ⇒ [Casual CNN Encoder] ↓ [Residual VQ (4-level)] ↓ [Casual CNN Decoder] ↓ Action [B, T, A] ⇐ [Reconstruction]

[VAE Pretraining]
Action [0.1, 0.2, 0.3, ...] → "code #15"

VQ loss

"code #15" → Action Reconstruction [0.09, 0.21,..]

L1 loss

Step2

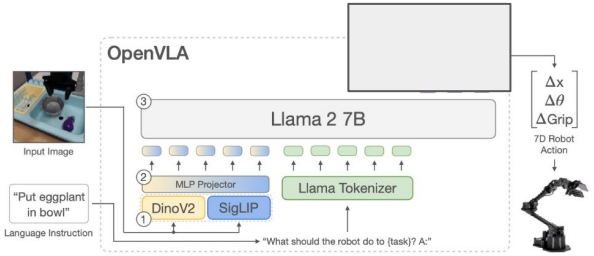
Map VQ Codes to LLM Token (manual mapping)

Level 1 code [0-255]
-> LLM Tokens [31040-31295]

Level 2 code [0-255]
-> LLM Tokens [31296-31551]
..

Step3

Fine-tune OpenVLA (frozen decoder) to predict action tokens



[OpenVLA] ↓ Generate LLM Tokens before action de-tokenizers: [31045, 31302, ..] ↓ VQVAQActionTokenizer. decode() ↓ [Action]

Ours

Step1

Quest + Conditioned Training process

Action [B, T, A] ⇒ [Quest Encoder]

↓

[VQ Codes]

↓

[Quest Decoder]

↓

Action [B, T, A] ⇐ [Reconstruction]

Language Information → [Quest Encoder]

Difference

Step2

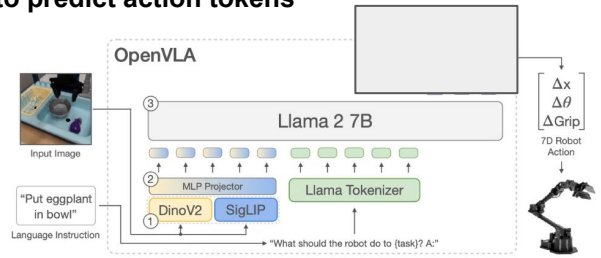
Map VQ Codes to LLM Token (manual mapping)

Encode: Action -> VQ Codes -> LLM Tokens
-Quest.encode -> Codes
-Code Mapping
-LLM Tokens

Decode: Tokens -> VQ Codes -> Action
Token - 32032 -> Code
-Code to action

Step3

Fine-tune OpenVLA (frozen quest decoder) to predict action tokens



[OpenVLA]

↓

Generate LLM Tokens before action
de-tokenizers: [31045, 31302, ..]

↓

ours.decode()

↓

[Action]

Quantitative Metric

Current Status

TODO

1. Integration Quest Decoder into VLA
2. Fine-tuning VLA with Conditioned Quest Decoder
3. Metric Evaluation
4. Simulation & Real-robot Task

2. Metric

Motivation

Beyond basic reconstruct loss and qualitative analysis e.g., skill was well compressed, let's create **"Quantitative Criteria" for skill abstraction quality** to evaluate environments (models, data, etc)

High Level Understanding

Prior work shows that **tokens can reconstruct and control, but they don't tell us**

(i) Semantic alignment, "does a token consistently connect one motion concept?"
(ii) Composability, "is that meaning invariant across episodes/tasks?"
(iii) compression, "does this token is efficiently compressed?"

So, we are adding **alignment** to verify semantic meaning and **composability** to find episode-invariance, **BPA** for compression, and so on.

These metrics demonstrate that our learned tokens are **reusable,compact, and somehow interpretable** exactly what a skill library should be.

Metric - Few Details

Prior work: what they measure (Quest, Vq-BET)

- **Reconstruction / NLL:** how well actions decode from tokens.
- **Perplexity:** effective number of codes used.
- **Task return / success rate:** downstream control performance.
- **Occasional visuals:** nearest-neighbor rollouts or token retrievals.

But these miss essentials for *skills*:

1. **Semantic alignment** — *does a token consistently mean one motion concept?*
2. **Composability** — *is that meaning invariant across episodes/tasks?*

Why this is important

- **Interpretability:** tokens are *meaningful skills*, not arbitrary chunks (High alignment)
- **Transfer & reuse:** the same token works *when & where* you need it (High composability)
- **Systems benefit:** cheaper planning, storage, and communication without control degradation (Low BPA + good recon)

Our evaluation: what we add

A. Compression / efficiency

- **Bits-per-action (BPA):** fewer bits at equal/better recon = better rate–distortion.
- **Compression ratio / token length,** plus **codebook utilization** sanity checks.

B. Segment-level “alignment” (InfoMEC family)

- **infomec_align_modularity:** each code specializes to one motion type.
- **infomec_align_compactness:** each motion type uses few codes (fewer synonyms).
- **infomec_align_explicitness:** linear probe predicts motion from codes.

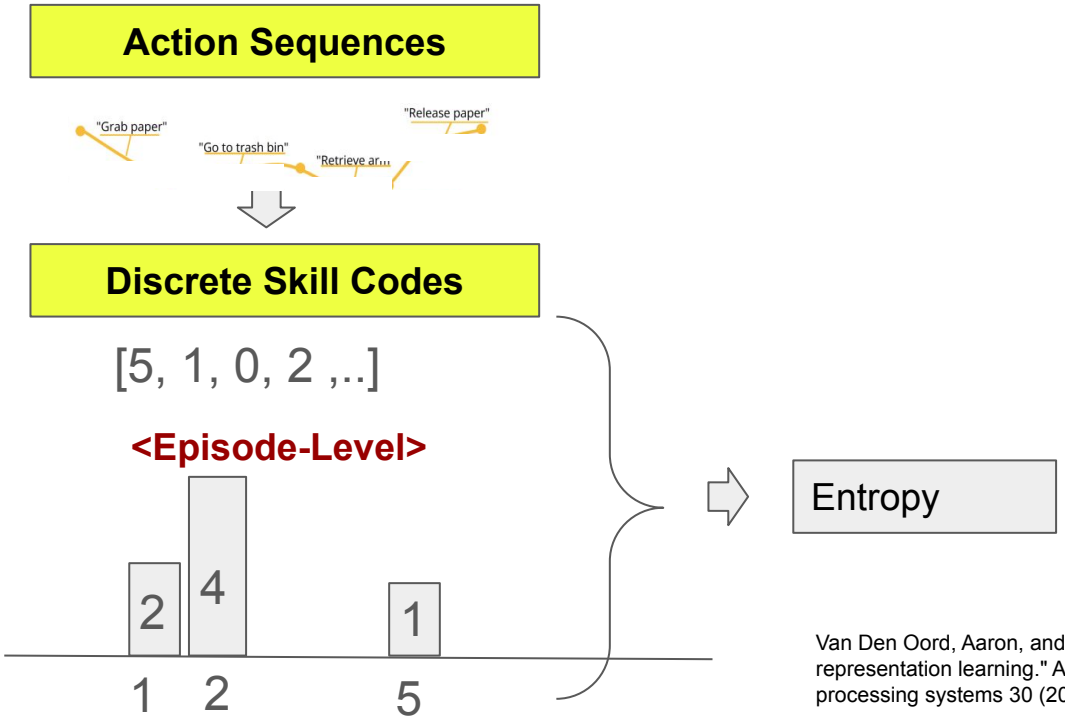
C. Sequential (Ongoing)

- **discrete first-order Markov model**

Bits-per-action (compactness)

How many bits we need to encode this sequence? Want to know skill primitives

- Episode-level evaluation, If not, it might correctly measure each metric since it will obtain value from randomly mixing skill information.



Van Den Oord, Aaron, and Oriol Vinyals. "Neural discrete representation learning." Advances in neural information processing systems 30 (2017).

Bits-per-action (compression)

Let say,

If use all 32 codes equally

- indices = [0,1,2,...,31,0,1,2,...,31, ...]
- unique_indices = [0,1,2,...,31]
- code_probs = [1/32, 1/32, , ...] **# Uniform distribution**
- entropy = log2(32) = 5.0 bits per action

If use 16 codes

- indices = [0,1,2,...,15,0,1,2,...,15, ...]
- unique_indices = [0,1,2,...,15]
- code_probs = [1/16, 1/16, ..., **0, 0, ...]** **# Half unused**
- entropy = log2(16) = 4.0 bits per action

Episode-level

Avoiding randomly mixing skill information to complete one episode

Episode 1: VQ codes [3,1,7,2] → entropy = 2.0 bits (4 different codes)
Episode 2: VQ codes [3,1,7,2] → entropy = 2.0 bits (4 different codes)
Episode 3: VQ codes [5,8,4,9] → entropy = 2.0 bits (4 different codes)
Episode 4: VQ codes [1,1,1,1] → entropy = 0.0 bits (1 repeated code)
..
Average: (2.0 + 2.0 + 2.0 + 0.0 + ...) / num_episodes = 1.456

To get entropy of codebook,

1. Count how often each code is used

```
code_counts = torch.bincount(predicted_skill_id, minlength=32)
        # code_counts = [85, 12, 45, 67, 23, 8, ...] (32 values)
```

2. Compute probability distribution of code counts

```
code_probs = code_counts / code_counts.sum()
```

3. Get Entropy (bits per action)

```
entropy = -torch.sum(code_probs * torch.log2(code_probs + 1e-10))
```

Assume that 4 bit is proper points for compression,
~16 codes used (medium entropy = 4.0 bits)

If entropy ~= 4.0, we're using ~lower codes efficiently
If entropy > 5.0, we're using too many codes (inefficient)
If entropy < 3.0, we're using too few codes (under-expressive)

Bits-per-action (compression)

```python
def compute_bits_per_action(
    self,
    indices: torch.Tensor,
    codebook_size: int
) -> float:
    """
    Compute bits-per-action compression metric.

    Args:
        indices: (B, T) codebook indices
        codebook_size: size of codebook K

    Returns:
        Average bits per action (lower = better compression)
    """
    if hasattr(indices, 'cpu'):
        indices = indices.cpu().numpy()

    indices_flat = indices.reshape(-1)

    # Compute empirical distribution over codebook indices
    unique_indices, counts = np.unique(indices_flat, return_counts=True)
    probs = counts / len(indices_flat)

    # Compute entropy in bits
    entropy_bits = -np.sum(probs * np.log2(probs + 1e-12))

    return float(entropy_bits)
```

VAE output

trajectory input -> VAE encoder -> ResidualVQ -> Codes (=indices)

Count how often each code is used

Entropy

InfoMEC

What is InfoMEC?

InfoMEC measures how well learned representations (like VQ codes) capture meaningful structure in data.

- Do the codes we learned actually represent distinct, useful concepts?

The Three Components:

1. **InfoMEC-M (Modularity): "Does each code specialize in one thing?"**
   - Good: Code A = "grasping", Code B = "moving"
   - Bad: Code A = "grasping + moving + rotating"

2. **InfoMEC-E (Explicitness): "Can we predict the task from the codes?"**
   - Good: Seeing codes [A,B,C] → I know it's "pick and place"
   - Bad: Codes don't tell me what task is happening

3. **InfoMEC-C (Compactness): "Does each concept use few codes?"**
   - Good: "Grasping" always uses codes [A,B]
   - Bad: "Grasping" scattered across codes [A,B,C,D,E,F]

Hsu, Kyle, et al. "Disentanglement via latent quantization." *Advances in Neural Information Processing Systems* 36 (2023): 45463-45488.

InfoMEC - Original paper

- The **sources** s_i are **given by the datasets** (not computed by binning raw pixels).
  Examples: “object color (10 values)”, “robot-x (40 bins)”, “camera height (4)”, etc. Those datasets (Shapes3D, MPI3D, Falcor3D, Isaac3D) already provide **discrete factor labels** **Note: the datasets already define discrete factors.**

- The **latents** z_j are **discrete** because the encoder output is **quantized per dimension** to a small scalar **codebook** (e.g., 10 values).

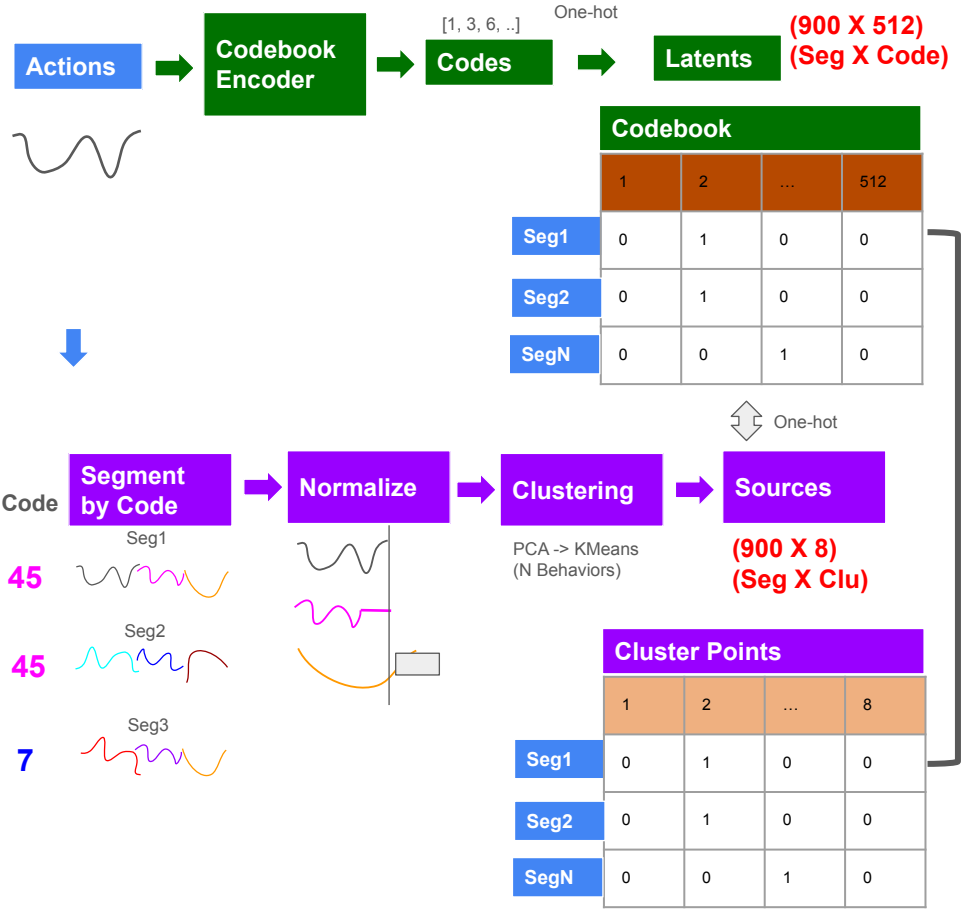
- InfoMEC is then computed between **ground-truth discrete factors** and the model’s **discrete latents**. (If any variable were continuous, they’d use a KSG MI estimator; but most factors there are discrete.)

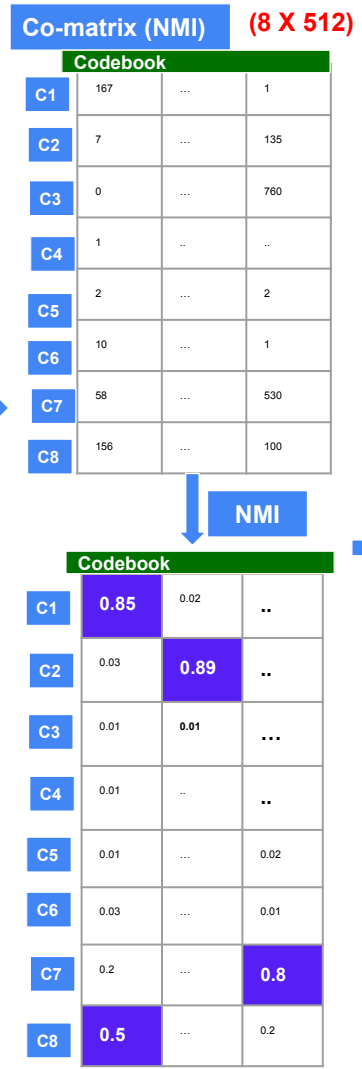
One Issues

- **In our case, we don’t have any discrete labels of sources, so we need to use unsupervised way to give some labels.**

InfoMEC - Alignment

how well cluster patterns can be represented by vq codes.





InfoMEC-M — **Column-wise (Codebook)**

1. col_ratios = **col_max** / col_sums
2. col_mean = np.mean(col_ratios)
3. InfoMEC-M= (mean - 1/clusters) / (1 - 1/clusters)

See whether each VQ code specializes in one behavioral pattern

InfoMEC-C — **Row-wise (Cluster)**

1. row_ratios = row_max / **row_sums**
2. row_mean = np.mean(row_ratios)
3. InfoMEC-C formula: (mean - 1/codebook) / (1 - 1/codebook)

See whether each behavioral pattern uses few VQ codes

InfoMEC-E — **Logistic Regression (X -> Y)**

```
infoe = compute_infoe(sources, latents,
discrete_latents=True)
  # X = latents (987, 512)
  # y = behavioral_clusters (987,)
```

See whether VQ codes are predictive of behavioral patterns

InfoMEC - Alignment (details)

Input Data:
Trajectories = (32, 256, 7) # 32 episodes, 256 timesteps, 7D actions
indices.shape = (32, 256) # 32 episodes, 256 VQ codes each

VQ code (256 codes) - 1st episode
indices[0] = [45, 45, 45, 45, 127, 127, 127, 234, 234, 89, ..]

Step 1: Extract segments from ALL episodes together
Collect segments across all episodes, ~800-1200 total segments collected

Step 2: Normalize segment lengths: Truncate or Right-Zero Padding
987 segments × 14D (target_length)

Step 3: PCA and clustering
# StandardScaler + PCA (987, 50) -> KMeans clustering, capped at 8 clusters
behavioral_clusters.shape = (987,) # [0,1,2,3,4,5,6,7,0,1,2,...]

Step 4: Create matrices for InfoMEC
Sources: behavioral clusters one-hot encoded: # 987 segments × 8 behavioral patterns
Latents: VQ codes one-hot encoded) # 987 segments × codebook_size = 512

Step 5: Compute InfoMEC Metrics
A. Compute NMI Matrix
result = compute_infomec_original(
sources=sources, # (987, 8) behavioral clusters
latents=latents, # (987, 512) VQ codes
)

Step 6.Matrix Build
```
for segment_idx in range(987):
 pattern_id = np.argmax(sources[i])  # Which pattern? (0-7)
 vq_code_id = np.argmax(latents[i])    # Which VQ code?

# Result:
co_occurrence_matrix = [
    [167,   5,   2,   8,  18, ...],  # code45=167x, code89=5x,
    [  7, 134,   3,   4,  12, ...],  # code45=7x, code89=134x,
    [  3,   2, 142,   6,   7, ...],  # code127=142x
    # ... 5 more patterns
]
```

Step 7: Calculate probabilities and MI
```
P_joint = co_occurrence_matrix / total_segments
P_pat = np.sum(P_joint, axis=1)  # Marginal: P(pattern)
P_vq_c = np.sum(P_joint, axis=0)  # Marginal: P(vq_code)

P_pat[0] = 0.20   # 20% of seg are pattern 0
P_vq_c[0] = 0.18   # 18% of seg use VQ code 45
P_joint[0,0] = 0.17   # 17% of seg both 0 AND code 45

 for i in range(8):      # For each behavioral pattern
    for j in range():    # For each VQ code
          MI[i,j] = P_joint[i,j] * np.log2(P_joint[i,j] / (P_pattern[i]
* P_vq_code[j]))
```

Step 8: Normalize by entropy
```
# Normalized MI: NMI(X,Y) = MI(X,Y) / H(X)
# H(X) = entropy of behavioral patterns
H_pattern = -np.sum(P_pat * np.log2(P_pat + 1e-10))
```

InfoMEC - Alignment (details)

A. Example NMI matrix:

```
nmi = np.array([
    [0.85, 0.02, 0.01, 0.03, 0.09, ...],  # Behavioral pattern 0 → mainly VQ code 45
    [0.03, 0.89, 0.01, 0.02, 0.05, ...],  # Behavioral pattern 1 → mainly VQ code 89
    [0.02, 0.01, 0.91, 0.03, 0.03, ...],  # Behavioral pattern 2 → mainly VQ code 127
    [0.01, 0.03, 0.02, 0.88, 0.06, ...],  # Behavioral pattern 3 → mainly VQ code 234
    [0.04, 0.02, 0.01, 0.04, 0.89, ...],  # Behavioral pattern 4 → mainly VQ code 312
    [0.02, 0.01, 0.02, 0.01, 0.02, ...],  # Behavioral pattern 5 → mixed codes
    [0.01, 0.02, 0.01, 0.02, 0.01, ...],  # Behavioral pattern 6 → mixed codes
    [0.02, 0.01, 0.01, 0.01, 0.01, ...]   # Behavioral pattern 7 → mixed codes
])
```

B. InfoMEC-M (Modularity) - Column-wise

```
# For each VQ code (column), check if it's modular
col_max = np.max(nmi, axis=0)    # [0.85, 0.89, 0.91, 0.88, 0.89, ...]  # 60 values
col_sums = np.sum(nmi, axis=0)   # [1.00, 1.00, 1.00, 1.00, 1.00, ...]  # normalized
col_ratios = col_max / col_sums  # [0.85, 0.89, 0.91, 0.88, 0.89, ...]
col_mean = np.mean(col_ratios)   # 0.87

# InfoMEC-M formula: (mean - 1/num_sources) / (1 - 1/num_sources)
num_sources = 8
infom = (0.87 - 1/8) / (1 - 1/8) = (0.87 - 0.125) / 0.875 = 0.85
# High modularity: each VQ code specializes in one behavioral pattern
```

C. InfoMEC-C (Compactness) - Row-wise

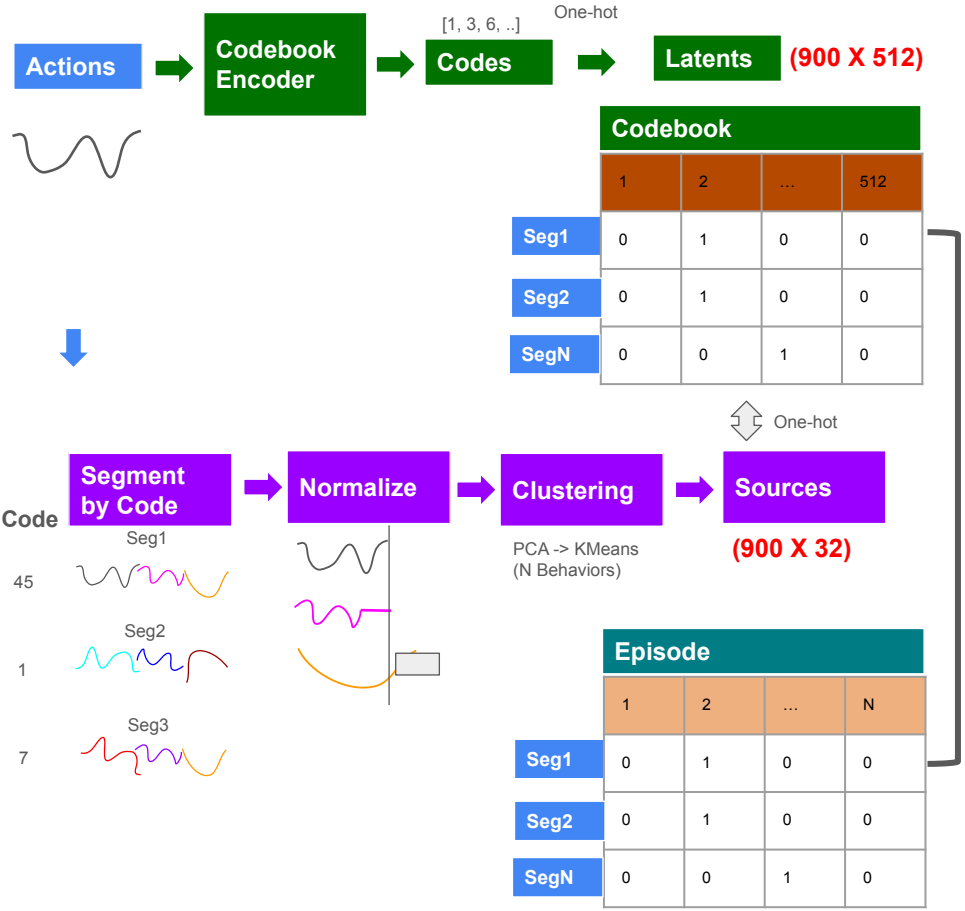
```
# For each behavioral pattern (row), check if it's compact
row_max = np.max(nmi, axis=1)    # [0.85, 0.89, 0.91, 0.88, 0.89, ..]
row_sums = np.sum(nmi, axis=1)   # [1.00, 1.00, 1.00, 1.00, 1.00,..]
row_ratios = row_max / row_sums  # [0.85, 0.89, 0.91, 0.88, 0.89,..]
row_mean = np.mean(row_ratios)   # 0.82

# InfoMEC-C formula: (mean - 1/num_active_latents) / (1 -
1/num_active_latents)
num_active_latents = 60
infoc = (0.82 - 1/60) / (1 - 1/60) = (0.82 - 0.017) / 0.983 = 0.82
# High compactness: each behavioral pattern uses few VQ codes
```

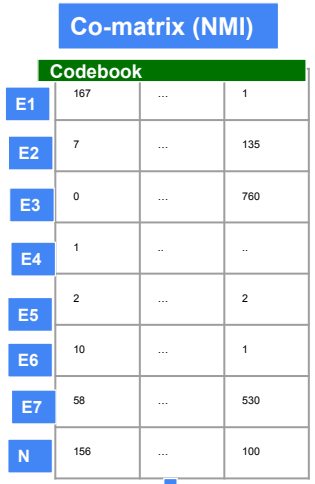
D. InfoMEC-E (Explicitness) - Logistic Regression

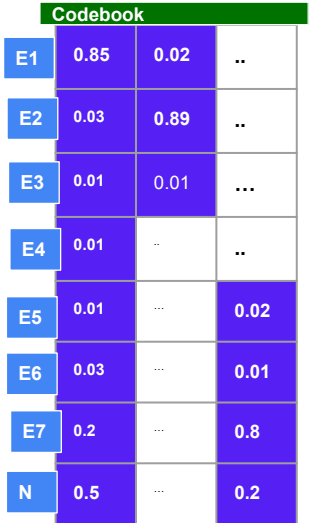
```
# Can we predict behavioral patterns from VQ codes?
infoe = compute_infoe(sources, latents, discrete_latents=True)

# X = latents (987, 512) - VQ code features
# y = behavioral_clusters (987,) - target labels [0,1,2,3,4,5,6,7,0,1,...]

predictive_loss = logistic_regression(latents, behavioral_clusters)
null_loss = logistic_regression(np.zeros_like(latents), behavioral_clusters)

infoe = (null_loss - predictive_loss) / null_loss = (2.08 - 0.25) / 2.08 = 0.88
# High explicitness: VQ codes are predictive of behavioral patterns
```

InfoMEC-C - Composability

how well skill patterns can be reusable across episodes







InfoMEC-C **Row-wise (Cluster)**

1. row_ratios = row_max / row_sums
2. row_mean = np.mean(row_ratios)
3. InfoMEC-C formula: (mean - 1/codebook) / (1 - 1/codebook)

See whether each behavioral pattern uses few VQ codes

Epi 1: uses codes 45,89,127,234,312

Epi 2: uses codes 45,89,127,234,312

Epi 3: uses codes 45,89,127,234,312

Epi 4: uses codes 45,89,127,234,312

... 28 more Epi

Epi 32: balanced usage

InfoMEC-C - Composability (details)

Step 1. Matrix Build

```
 for segment_idx in range(987):
  pattern_id = np.argmax(sources[i])  # Which pattern? (0-7)
  vq_code_id = np.argmax(latents[i])    # Which VQ code?

co_occurrence_matrix = [
    [8, 7, 6, 8, 9, 0, 0, 0, ...],  # Epi 0: uses codes 45,89,127,234,312
    [6, 9, 8, 7, 8, 0, 0, 0, ...],  # Epi 1: uses codes 45,89,127,234,312
    [7, 8, 9, 6, 7, 0, 0, 0, ...],  # Epi 2: uses codes 45,89,127,234,312
    [9, 6, 7, 9, 8, 0, 0, 0, ...],  # Epi 3: uses codes 45,89,127,234,312
    # ... 28 more Epi
    [8, 8, 8, 8, 8, 0, 0, 0, ...],  # Epi 31: balanced usage
 ]
```

Step 2. Calculate probabilities and MI

```
 P_episode = [0.031, 0.031, 0.031, ...] # Each episode ~3.1%
 P_vq_code = [0.18, 0.19, 0.17, 0.16, 0.18, ...] # VQ code distribution
 P_joint[0,0] = 8/987 = 0.008  # P(episode=0 AND vq_code=45)

 for i in range(32):        # For each episode
    for j in range(N):     # For each VQ code
       if P_joint[i,j] > 0:  # Avoid log(0)
          MI[i,j] = P_joint[i,j] * np.log2(P_joint[i,j] / (P_episode[i] *
P_vq_code[j]))
```

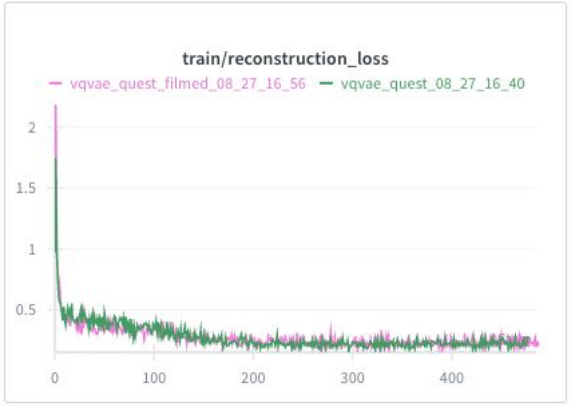
Step 3. Normalize by entropy

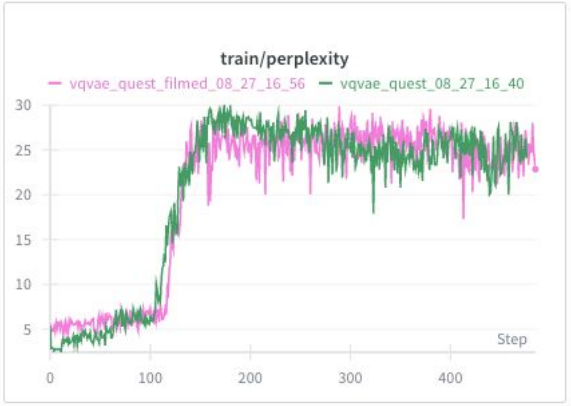
```
 H_episode = -np.sum(P_episode * np.log2(P_episode + 1e-10))
```

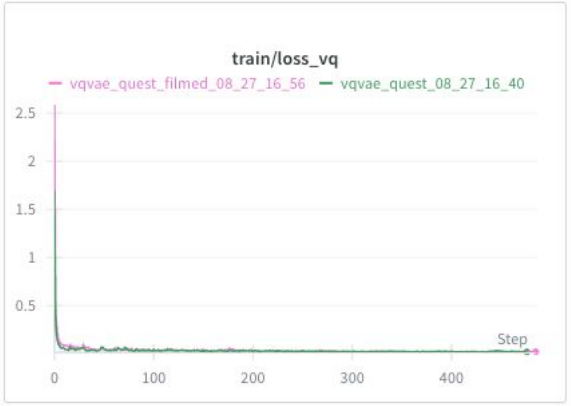
Step 4: Compute InfoMEC-C only (composability metric)

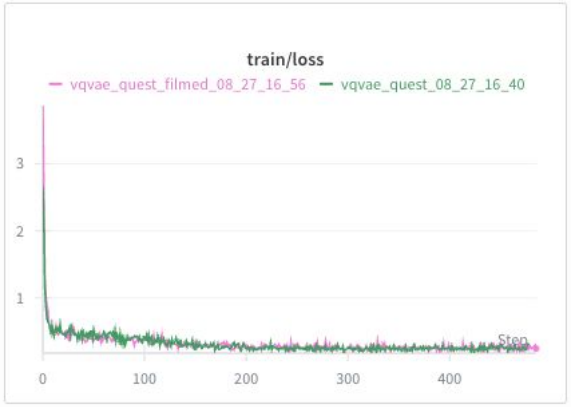
```
 # NMI between 32 episodes and active VQ codes
 nmi.shape = (32, 60)

 # Example episode-level NMI:
 episode_nmi = np.array([
    [0.20, 0.18, 0.22, 0.15, 0.25, ...],  # Epi 0: codes somewhat equally
    [0.25, 0.20, 0.20, 0.20, 0.15, ...],  # Epi 1: codes somewhat equally
    [0.15, 0.30, 0.25, 0.15, 0.15, ...],  # Epi 2: for some codes
    # ... 29 more episodes
 ])

 # InfoMEC-C for episodes (lower = better composability)
 row_max = np.max(episode_nmi, axis=1)    # Max NMI per episode
 row_sums = np.sum(episode_nmi, axis=1)   # Sum per episode
 row_ratios = row_max / row_sums          # Ratio per episode
 row_mean = np.mean(row_ratios)           # Average ratio

 num_active_latents = 60
 episode_infoc = (row_mean - 1/60) / (1 - 1/60)  # e.g., 0.35

 # Transform to composability score
 composability_score = max(0.0, 1.0 - episode_infoc)  # e.g., 0.65
```

Results - 1

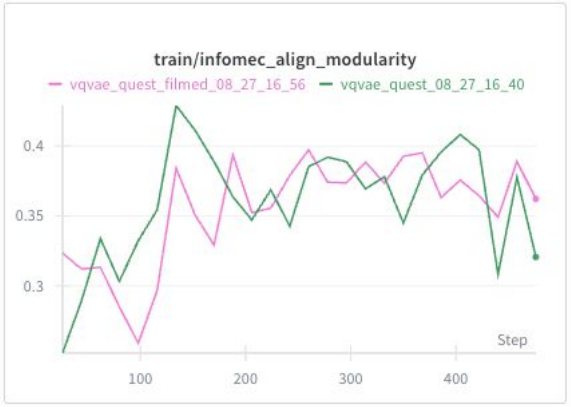

train/reconstruction_loss
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
2
1.5
1
0.5
0
100
200
300
400


train/perplexity
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
30
25
20
15
10
5
Step
0
100
200
300
400

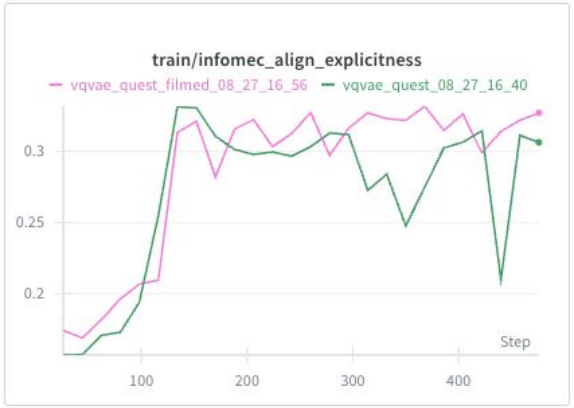

train/loss_vq
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
2.5
2
1.5
1
0.5
Step
0
100
200
300
400

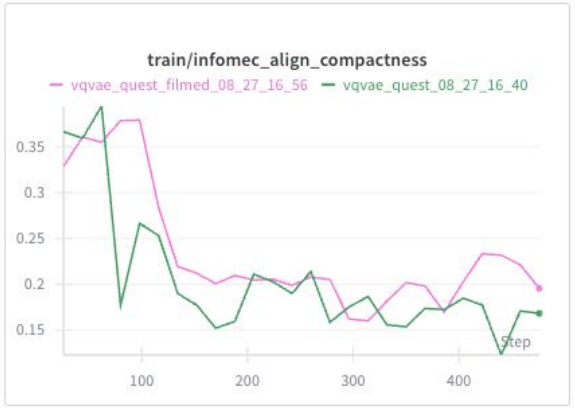

train/loss
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
3
2
1
Step
0
100
200
300
400

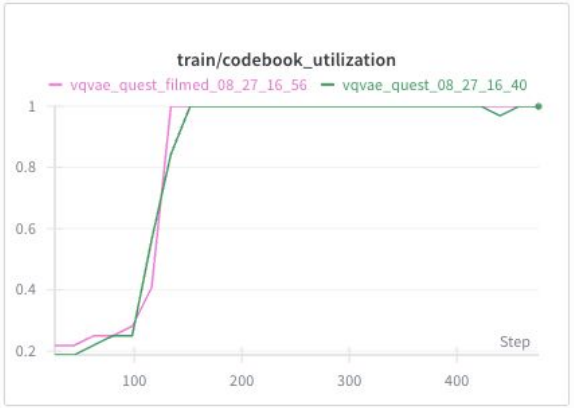

train/infomec_composability
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
0.9
0.85
0.8
0.75
0.7
0.65
0.6
Step
100
200
300
400


train/infomec_align_modularity
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
0.4
0.35
0.3
Step
100
200
300
400

Results - 2


train/infomec_align_explicitness
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
0.3
0.25
0.2
Step
100
200
300
400


train/infomec_align_compactness
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
0.35
0.3
0.25
0.2
0.15
Step
100
200
300
400


train/codebook_utilization
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
1
0.8
0.6
0.4
0.2
Step
100
200
300
400


train/bits_per_action
vqvae_quest_filmed_08_27_16_56
vqvae_quest_08_27_16_40
3.5
3
2.5
2
1.5
Step
100
200
300
400

Key Metrics & Observations

Alignment dimensions

- **Modularity** → Each code specializes in specific movement patterns
- **Explicitness** → Behavior can be predicted from VQ codes
- **Compactness** → Patterns map to focused VQ codes

Results

- **Loss**: 0.245 vs 0.280 (**–12%**) → **better training**
- **Perplexity**: 22.85 vs 24.48 (**–7%**) → **effective codebook usage, fewer codes used**
- **Modularity (alignment)**: 0.362 vs 0.321 (**+13%**) → **VQ codes specialize in specific movement patterns**
- **Explicitness (alignment)**: 0.327 vs 0.306 (**+7%**) → **codes more predictive**
- **Compactness (alignment)**: 0.196 vs 0.168 (**+17%**) → **fewer codes used?**
- **BPA (bits per action)**: 3.571 → 3.308 (**–7.4%**) → **model needs ~0.26 fewer bits per action → better representations**

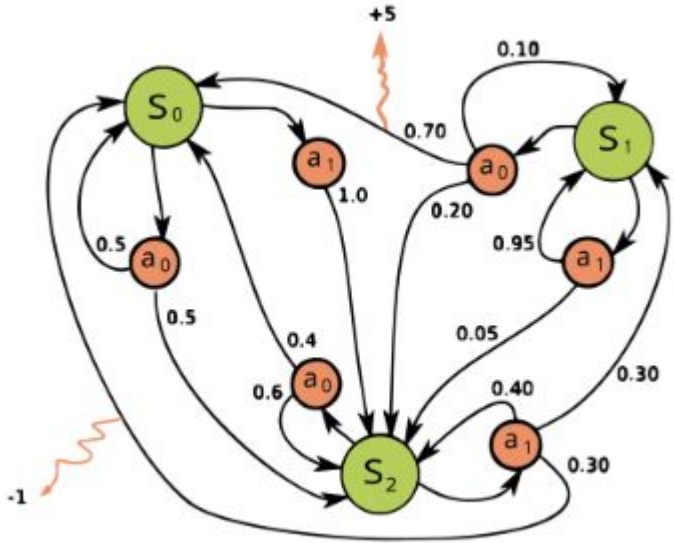
Note:
- alignment improves (codes become more specialized/meaningful)

Markov Decision Process (MDPs)

Markov Decision Processes (MDPs)

MDPs are a mathematical model for sequential decision-making in a fully observable, stochastic, discrete-time environment with a Markovian transition model.

$$M = \langle S, A, T, r \rangle \qquad A = \{\leftarrow, \uparrow, \rightarrow\}$$



MDP Metric Objective

1) Episode-level (meso): *Sequentiality / predictability*

What it answers: *How predictable is the next code from the last K codes within an episode?*
Why this level: To understand strong sequential structure & sequence property.

2) Task-level (macro): *Composability / invariance*

What it answers: *Do different episodes/tasks use the same tokens and chain them the same way?*
Why this level: Reuse/transfer is about **consistency across tasks/episodes**.

discrete first-order Markov model

P(next_code | prev_code) - First-order Markov

- **Markov assumption:** The next state depends **only** on the current state, not the full history.
- **No control** (passive observation) **& No Action Space [**Action = state (predict next code, which becomes next state)]
- **Deterministic policy:** Always pick `argmax`

MDP Component	Our Implementation
State s_t	prev (last 1 token, K=1)
Action a_t	argmax(model[prev]) (*pick top-1 next code*)
Transition P(s_{t+1}|s_t)	bigram_model[prev, next], *# Learning P(z_{t+1} | z_t) from data*
Reward r(s,a,s')	1 if pred==true else 0 (correctness)
Discount γ	1 (finite episodes, average per step)
Discrete	256 codes (finite state space)

Sequentiality

VQ model captures temporal structure in the behavior?

Learn bigram model P(next_code | prev_code) → Test: Can we predict next code?

Train

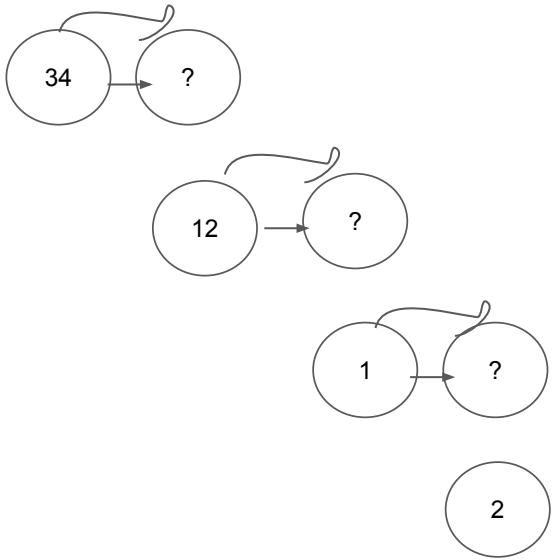
Codes

Epi1 34 → 12 → 1 → 2

Epi2 41 → 61 → 3 → 7

Bigram model

Test



Obtain probabilities for 256 codes in given dataset
-> Statistical Model, It's learning from data

bigram_model[45] =
[0.0, ..., 0.4, ..., 0.5, ..., 0.1, …]
-> After code 45, next code is usually 127 (50%) or 45 (40%)

1. Sequentiality
how well VQ model captures temporal structure in the behavior

What's this step: Train data → Learn P(next_code | prev_code) → Test: Can we predict next code?
How we get:

```
def evaluate_predictability(test_eps, model):
    for ep in test_eps:
        for prev, true_next in zip(ep[:-1], ep[1:]):
            pred_next = np.argmax(model[prev])
            if pred_next == true_next:
                correct += 1
```

- **It's a Statistical Model,** It's **learning from data**
  - Train on train episodes, test on held-out test episodes
- **Actual next word is the "ground truth"**
  - Given code 45 at time t, can you predict that code came next?
  - The **observed sequence** is the ground truth

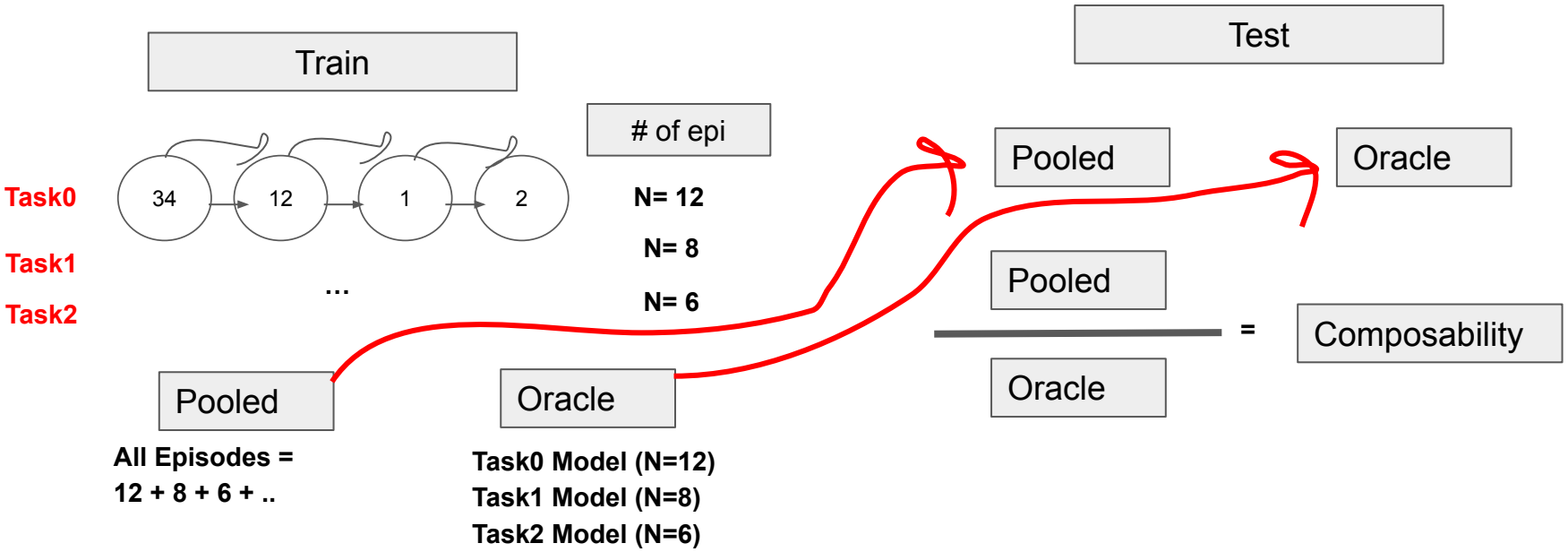
Episode 1: [45 → 127 → 127 → 234 → 89]
↓
Train bigram model: "After 45, usually comes 127"
↓

Test Episode 2: [45 → ?]
↓
Predict: 127
↓

Check: Did 127 actually come next in Episode 2?

Short summary - Composability

Do different tasks reuse the same VQ code patterns?



2. Composability

Do different tasks reuse the same VQ code patterns?

→ **"How much do tasks share patterns?**

What's this step: Compare: One general model vs Task-specific models
How we get: **Train pooled and oracle model**

Assumption

1. **High sharing** -> we can learn one model for multiple tasks (efficient)
2. **Low sharing** -> Each task needs its own specialized knowledge (harder to generalize)

So setup,

1. **Generalist performance:** How well does one general course work for everyone?
   a. **Pooled:** Cross-task generalization -> Can one model handle everything?
2. **Specialist performance:** How well do specialized courses work?
   a. **Oracle:** Best possible performance -> How well can we do if specialized?

Ratio: Pattern sharing -> "How much do tasks share patterns?

If generalist ≈ specialist → General course is fine! (High composability)
If generalist << specialist → Need specialized courses! (Low composability)

Short summary - Composability

High Composability (pooled ≈ oracle)

Task 0: [45→127→234→89] *# reach, grasp, lift, place*
Task 1: [45→127→234→50] *# reach, grasp, lift, place (different object)*
Task 2: [45→127→180→89] *# reach, grasp, different motion, place*

 # All share: 45→127 (reach→grasp)

Pooled model learns:
After 45 → predict 127 (100% accurate across all tasks)
After 127 → predict 234 (60%) or 180 (40%) - pretty good

Oracle model learns (Task 0 only):
After 45 → predict 127 (100%)
After 127 → predict 234 (100% for Task 0 specifically)

Result: Pooled is 95% as good as Oracle
-Because tasks share most transitions
- The pooled model benefited from seeing more examples of 45→127

Low Composability (pooled << oracle)

Task 0: [10→20→30→40] *# Unique code sequence*
Task 1: [100→110→120→130] *# Completely different codes*
Task 2: [200→210→220→230] *# No overlap at all*

Pooled model learns:
After 10 → predict ??? (only seen in Task 0, confused with other tasks)
After 100 → predict ??? (only seen in Task 1)
Accuracy: 35% - just random guessing

Oracle model learns (Task 0 only):
After 10 → predict 20 (100% - specialized)
After 20 → predict 30 (100%)
Accuracy: 95%

Result: Pooled is only 37% as good as Oracle
- Because tasks don't share anything
- The pooled model is harmed by mixing incompatible patterns

Another Intuition

If patterns are shared:

P_pooled(next|prev)
≈ P_task0(next|prev)
≈ P_task1(next|prev)

All tasks follow similar rules
→ Pooled model learns the common rule
→ Performs almost as well as specialists

If patterns are unique:

P_pooled(next|prev)
= average of different rules P_task0(next|prev)
≠ P_task1(next|prev)
≠ P_task2(next|prev)

Each task has different rules
→ Pooled model learns a confused average
→ Much worse than specialists

2. Composability

Do different tasks reuse the same VQ code patterns?

DATA SPLIT:

```
├── TRAIN (48 episodes)
│   ├── Task 0: 12 episodes
│   ├── Task 1: 10 episodes
│   ├── Task 2: 15 episodes
│   └── Task 3: 11 episodes
│
└── TEST (32 episodes)
    ├── Task 0: 8 episodes
    ├── Task 1: 7 episodes
    ├── Task 2: 9 episodes
    └── Task 3: 8 episodes
```

TRAINING PHASE:

```
┌──────────────────────────────┐
│ Pooled Model (1 model)       │
│  ← train ALL 48 episodes     │
└──────────────────────────────┘

┌──────────────────────────────┐
│ Oracle Models (4 models)     │
│  Model_0 ← train Task 0 (12 eps) │
│  Model_1 ← train Task 1 (10 eps) │
│  Model_2 ← train Task 2 (15 eps) │
│  Model_3 ← train Task 3 (11 eps) │
└──────────────────────────────┘
```

TESTING PHASE:

```
┌──────────────────────────────┐
│ Pooled: Test ALL 32 test episodes │
│  with same pooled model      │
│  → pooled_acc = 0.82         │
└──────────────────────────────┘

┌──────────────────────────────┐
│ Oracle: Test each task separately │
│  Task 0 test (8 eps) → Model_0 │
│    → acc_0 = 0.87            │
│  Task 1 test (7 eps) → Model_1 │
│    → acc_1 = 0.92            │
│  Task 2 test (9 eps) → Model_2 │
│    → acc_2 = 0.85            │
│  Task 3 test (8 eps) → Model_3 │
│    → acc_3 = 0.90            │
│  → oracle_acc = 0.885 (weighted) │
└──────────────────────────────┘
```

COMPOSABILITY = 0.82 / 0.885 = 0.93

Pooled (trained on ALL train episodes)

- pooled_acc = evaluate_predictability(test_eps, bigram_model)

Oracle (trained only on episodes similar to each test episode)

Key Idea:

- If **pooled_acc ~= oracle_acc** → episodes share patterns (high composability)
- If **pooled_acc << oracle_acc** → each group has unique patterns (low composability)

HIGH Composability (Reusable Patterns)

Task 0 (Pick & Place):
[reach → grasp → lift → move → place]
↓ ↓ ↓ ↓ ↓
VQ: 45 → 127 → 234 → 89 → 50

Task 1 (Open Drawer):
[reach → grasp → pull → open → release]
↓ ↓ ↓ ↓ ↓
VQ: 45 → 127 → 180 → 200 → 75

Task 2 (Press Button):
[reach → align → press → retract]
↓ ↓ ↓ ↓
VQ: 45 → 90 → 150 → 30

LOW Composability (Unique Patterns)

Each task has completely different patterns:

Task 0:
VQ: 10 → 20 → 30 → 40 → 50

Task 1:
VQ: 100 → 110 → 120 → 130 → 140

Task 2:
VQ: 200 → 210 → 220 → 230 → 240

Results

Goal: Visualize how VQ-VAE skill codes transition from one to another over time.
- Nodes = Skill codes (e.g., "grasp", "move", "release")
- Edges = Transition probabilities (e.g., "grasp" -> "move" with 80% probability)
- The graph shows which skills naturally follow each other in robot behavior
- Training Episodes -> VQ Codes -> Transition Matrix -> Graph Visualization
- Example:
- Time: t=0 t=1 t=2 t=3 t=4
- Codes: [5] → [5] → [12] → [3] → [5]

This creates transitions:
- Code 5 → Code 5 (self-loop), Code 5 → Code 12, Code 12 → Code 3, Code 3 → Code 5

[Details]

Step 1: After Build Bigram Transition Model,
Result: Transition matrix where probs[5, 12] = 0.23 means "after code 5, code 12 appears 23% of the time"

Step 2: Select Top Codes to Visualize
[5, 12, 3, 27, 18, ...] - the 15 most-used codes

Step 3: Build Graph Structure (Create DAG)
Example graph:
- Nodes: {3, 5, 12, 18, 27}
Edges:
5 → 5 (weight=0.65)
5 → 12 (weight=0.23)
12 → 3 (weight=0.51)
3 → 5 (weight=0.34)

Count transitions in training data → transition matrix -> Select top codes to avoid clutter -> Build graph with codes as nodes, transitions as edges

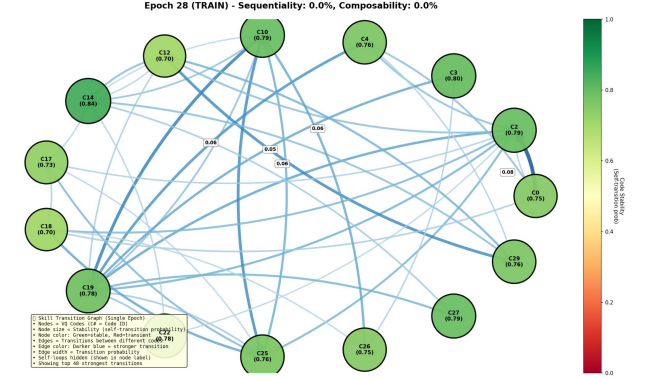
It's computed by counting how often one skill code transitions to another in one episode, normalizing those counts to probabilities, then visualizing the most frequent ones.

e.g.,: [5 → 5 → 12 → 3 → 5], estimate P(5→5)=0.75, P(5→12)=0.25, build a weighted graph (nodes = codes, edges = transition probabilities), and draw thicker edges for higher probabilities.

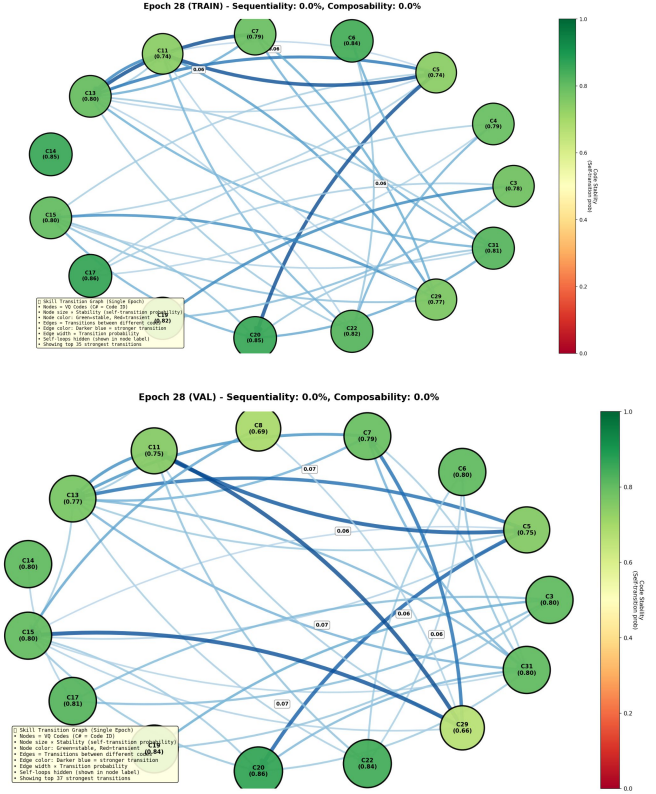
Result - Sequentiality

this is episode level epoch results (the end of one episode wouldn't be a point to connect with next episode initial point)
And, randomly selected epoch and task types amongst all tasks for composability

Baseline

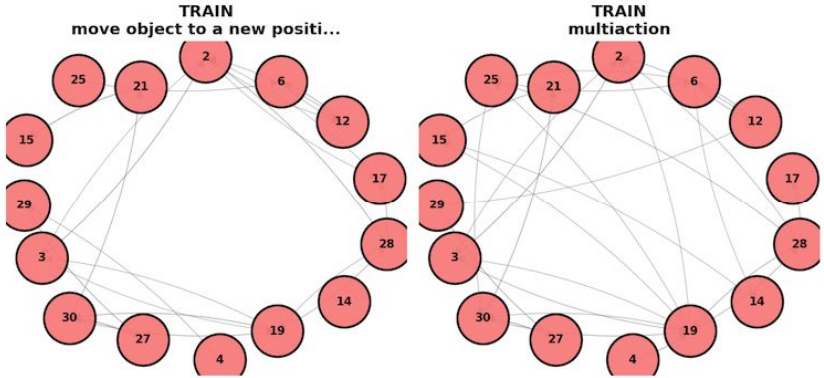


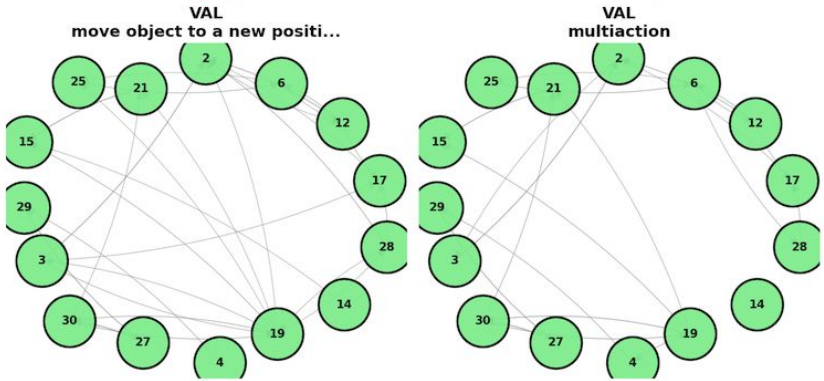
Film



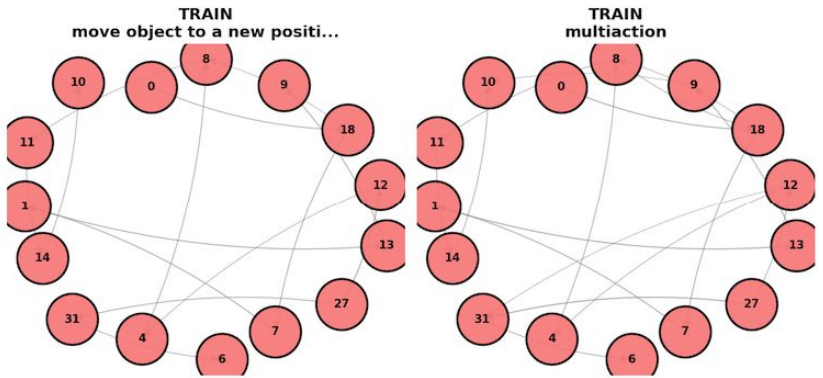
Result - Composability

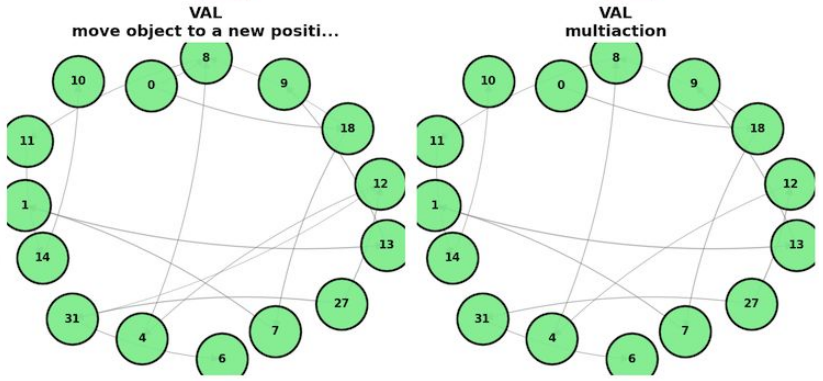
Baseline





Film





Discussion

collapsed structure in both sequentiality and composability graphs suggests that transitions have become **more focused and deterministic**, meaning fewer random transitions and stronger skill-to-skill predictability.

- Some skill codes dominate transitions, implying the model has learned **stable, reusable primitives**.

- Stronger, thicker edges (high transition probabilities) show **clear temporal structure** — one skill consistently follows another

show **strong learned structure** (good), but **low cross-task transfer** (potentially limiting)

Sequentiality: Film > Baseline

- Film creates more deterministic, structured sequences
- Good train-val consistency = generalizes well

Related Questions

- **Does higher sequentiality improve robot performance?**
- **Can composability be increased without reducing sequentiality?**
- **How does diversity vs. determinism trade off—does collapse imply overfitting?**
- **Quantify with entropy/sparsity of transitions and track edge dominance over epochs to monitor convergence.**

3. Evaluation

1) Quest (baseline)

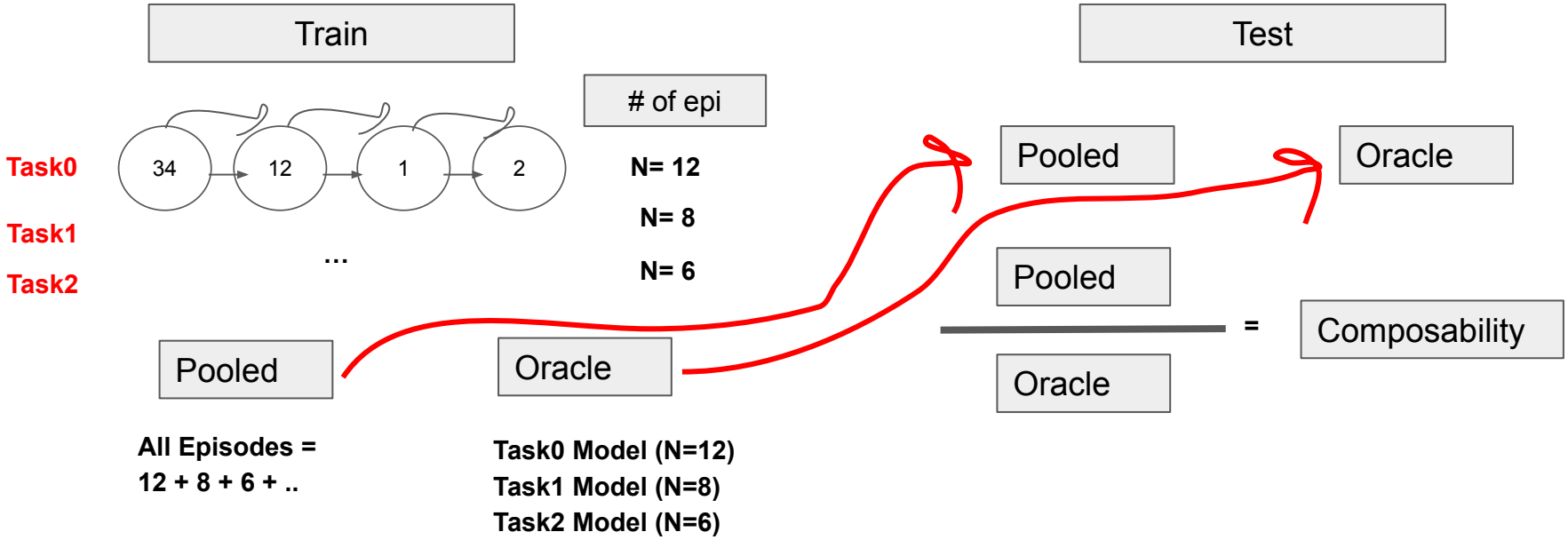
2) Quest + Conditioned (our)

3) VQ-VLA

4) π model

4. Experiments - Simulation (Libero) & Robot

Short summary - Composability



Pooled (trained on ALL train episodes)
- pooled_acc = evaluate_predictability(test_eps, bigram_model)

Oracle (trained only on episodes similar to each test episode)

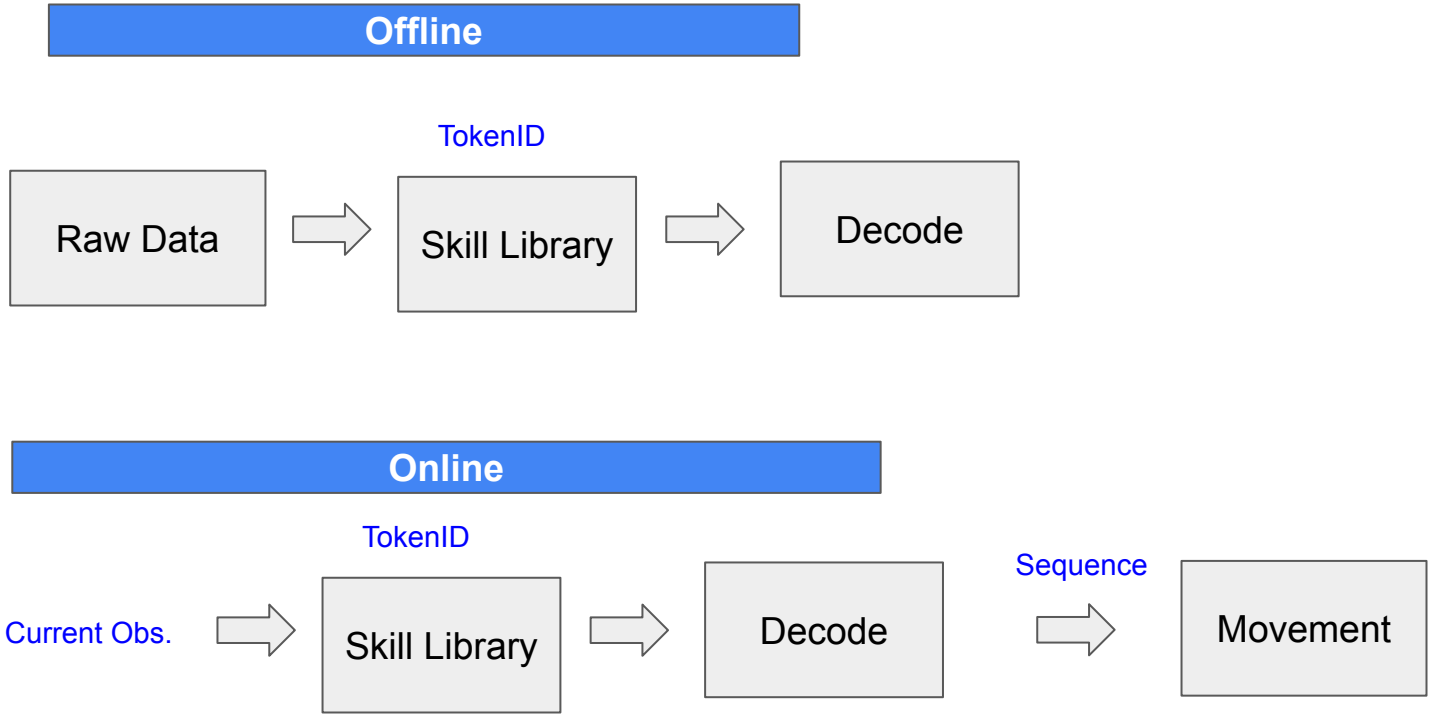
- If **pooled_acc ~= oracle_acc** → episodes share patterns (high composability)
- If **pooled_acc << oracle_acc** → each group has unique patterns (low composability)

Pooled: "I learned from all Romance languages, they share grammar!"
→ Translates Spanish well (90% accuracy)

Oracle: "I only learned Spanish"
→ Translates Spanish slightly better (92% accuracy)

Ratio: 90/92 = 0.98 <- Good Composability

Overall Pipeline



Detail Example

[VAE Pretraining]

"Learn a good dictionary of action patterns"
Action [0.1, 0.2, 0.3, ...] → "code #15"
"code #15" → Action Reconstruction [0.09, 0.21, 0.29, ...]