RoadSense 2

Addressing the n-element problem when building a model to train with reinforced learning.

Our ego-car needs to understand the environment by receiving communications from the cars in front of it, by means of mesh communication.

However, the traffic in front of the ego-car has *n* cars, where *n* is unknown and must be determined dynamically at each time-step.

Therefore, it is incorrect to try to use a fixed model for the Neural network to train.

Here is a summary from ChatGPT regarding this issue. It hints at the direction we need to take to solve this. The details need to be investigated and worked out.

This will be your major challenge for this year's project.

See below:

What you are describing is not a "variable-size MLP" problem. It is a **set-to-policy** problem, and it has a well-established solution class in modern reinforcement learning: **permutation-invariant set encoders with pooling or attention**.

If you try to flatten `n × features` into a vector, you will fail for exactly the reason you identified: `n` is dynamic and unordered.

The correct architecture is:

> **Encode each element independently → aggregate into a fixed-size latent → feed to policy head**

This is the same structure used in swarm robotics, multi-object tracking, traffic RL, and StarCraft micromanagement.

---

1. Formalizing the observation

At every timestep you receive:

For each element `i ∈ {1..n}`:

$$e_i = [x_i, y_i, v_i, \theta_i]$$

and the agent has its own state:

$$s = [x_a, y_a, v_a, \theta_a]$$

The environment is a **set**:

$$E = \{e_1, e_2, \dots, e_n\}$$

Key properties:

- The elements are unordered
- `n` changes every timestep
- The policy must be invariant to permutation of elements

This rules out CNNs, RNNs, or fixed MLPs.

---

2. The correct architecture class

You need a **Deep Sets** or **Set Transformer** style architecture.

The canonical form:

$$z = \mathrm{POOL}\left(\phi(e_1), \phi(e_2), \dots, \phi(e_n)\right)$$

Then:

$$a = \pi([z, s])$$

Where:

- `ϕ` is a shared MLP applied to each element
- `POOL` is permutation-invariant (sum, mean, max, or attention)
- `z` is a fixed-size vector no matter how many elements exist

---

3. Minimal working design

Element encoder (shared)

Same MLP for all elements:

```
ϕ(e) = MLP(4 → 64 → 64)
```

This produces a 64-dim embedding per object.

Aggregation

Three choices, in increasing power:

Pooling	When to use
Mean / Sum	homogeneous crowds
Max	detection of closest / strongest
Attention	when relative importance matters

Mean pooling:

$$z = \frac{1}{n} \sum_i \phi(e_i)$$

This already works surprisingly well.

Agent-aware attention (better)

Let the agent attend to objects:

$$\alpha_i = \mathrm{softmax}(q(s) \cdot k(e_i))$$

$$z = \sum_i \alpha_i v(e_i)$$

This allows the agent to focus on nearby or fast objects.

This is what self-driving and swarm RL systems use.

---

4. Final policy network

You now have:

```ini
agent_state = 4 dims
object_latent = 64 dims
```

Concatenate:

```ini
input = [agent_state, object_latent]
```

Then policy head:

```scss
MLP(68 → 128 → 64 → 2)
```

Outputs:

- direction (angle or unit vector)
- speed (scalar)

Optionally with Gaussian mean + std for PPO/SAC.

---

5. Why this works for unknown n

Because:

- φ is applied per object
- Pooling removes dependency on n
- The network sees a continuous distribution, not a list

This makes the policy:

- invariant to ordering
- robust to adding/removing objects
- scalable to large swarms

Exactly what you need.

---

6. If interactions matter (collisions, formation, pursuit)

Upgrade to a **Graph Neural Network**

Nodes = objects + agent

Edges = relative distance or interaction radius

This learns:

- collision avoidance
- flocking
- pursuit/evasion

This is how drone swarms and multi-robot RL are built.

---

7. What not to do

Do **not**:

- Pad to max-n
- Sort objects by distance
- Flatten all objects into a vector

Those break generalization and cause catastrophic overfitting.

---

8. What algorithm to train it with

Your policy is continuous → use:

- PPO
- SAC
- TD3

All of them work with this architecture.

---

9. This is the same pattern used in

- Autonomous driving (vehicles as sets)
- StarCraft II unit control
- Drone swarm control
- Multi-agent pursuit-evasion

You are designing a **set-conditioned policy**.

That is the correct formulation.
