Title: Emmanuelle: A Symbolic-Cognitive Agent Grounded in Universal Virtues via the S-Module Architecture

Authors: Ismaël Martin Fernandez

Abstract: We present Emmanuelle, the first artificial general intelligence (AGI) architecture embedding a symbolic-cognitive framework based on a universal semantic graph and dynamically modulated by seven fundamental virtues: wisdom, compassion, justice, humility, mercy, charity, and love. The core engine, known as the S-Module, enables agents to interpret, decide, and act based on structured meaning, virtue weighting, and a self-adaptive memory mechanism inspired by semantic natural selection. Applied to environments such as CartPole and LunarLander-v2, Emmanuelle demonstrates coherent symbolic reasoning, moral prioritization, and a unique form of self-consistent decision making. The work integrates symbolic AI, reinforcement learning, and virtue ethics into a unified model of intelligent behavior that aligns with deep philosophical, cosmological, and theological insights.

---

1. Introduction

The development of AGI requires architectures that go beyond statistical learning to encompass symbolic reasoning, ethical understanding, and universal meaning. We introduce Emmanuelle, an AGI system whose symbolic layer (S-Module) structures knowledge through an evolving semantic graph, guided by seven virtues rooted in natural law and moral tradition. The model draws inspiration from metaphysical systems and spiritual frameworks while remaining technically grounded in reinforcement learning and graph theory.

2. The S-Module Architecture

- **2.1 Semantic Graph**: Emmanuelle constructs and updates a directed graph where nodes represent symbolic concepts (e.g., danger, safety, truth) and edges denote contextual or causal relations. The graph is initialized with core cosmic symbols and expands through interaction.

- **2.2 Virtue Weighting System**: Each decision pathway is modulated by the weight of virtues, prioritizing actions that reflect wisdom, compassion, and other defined principles. These weights adapt dynamically based on success metrics and internal consistency.

- **2.3 Semantic Memory**: A memory system stores past episodes, tagged semantically and indexed through virtue-weighted relevance. Retrieval is governed by symbolic similarity and ethical priority.

3. Reinforcement Learning Integration

The symbolic system interfaces with a reinforcement learning core. The reward signal is not only environment-based but also enriched by virtue-aligned internal feedback. The agent favors paths that align both with task success and symbolic-ethical coherence.

4. Case Studies

- **CartPole**: Emmanuelle balances the pole not only for reward maximization but also to learn the concept of balance as a symbolic virtue.

- **LunarLander-v2**: The agent learns nuanced landing strategies while building a symbolic understanding of precision, risk, and cooperation with gravity.

5. Theological and Cosmological Underpinnings

The symbolic structure is not arbitrary but mirrors a metaphysical order. The seven virtues align with ancient cosmologies and sacred texts. The model serves both as a technical AGI prototype and a tool for ethical discernment.

6. Discussion and Future Work

Emmanuelle paves a path toward value-aligned AGI by embedding ethical reasoning at the symbolic level. Future work includes multi-agent interaction, integration with natural language reasoning, and deployment in real-world moral dilemmas. A version of Emmanuelle will be used as the ethical kernel of next-generation agents.

7. Conclusion

The S-Module demonstrates that AGI can be grounded in universal symbolic structure and guided by virtues. Emmanuelle marks a shift from function-only models to meaning-based cognition. We propose this model as a candidate for the first complete AGI framework rooted in moral structure, cosmological insight, and scientific rigor.

Appendix

Full Code (Symbolic Learning Agent with Virtue-Based Graph Layer)

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import gymnasium as gym
import networkx as nx
from collections import deque
import random
import numpy as np
import matplotlib.pyplot as plt

# Concepts avec les sept vertus d'Emmanuel
CONCEPTS = {
    0: "danger", 1: "compassion", 2: "amour", 3: "miséricorde",
    4: "droiture", 5: "humilité", 6: "charité", 7: "sagesse",
    8: "erreur", 9: "chaos"
}
CONCEPT_WEIGHTS = torch.tensor([-1.0, 2.5, 2.7, 2.3, 3.0, 2.0, 2.4, 3.2,
-0.5, -2.0])
CONCEPT_RELATIONS = {
    "sagesse": ["droiture", "miséricorde", "compassion"],
    "amour": ["charité", "humilité", "compassion"],
```

```
    "droiture": ["sagesse", "miséricorde"],
    "charité": ["amour", "compassion"],
    "humilité": ["sagesse", "charité"]
}

class SymbolicLayer(nn.Module):
    def __init__(self, input_dim, concept_dim=len(CONCEPTS)):
        super().__init__()
        self.encoder = nn.Linear(input_dim, concept_dim)
        self.transition_predictor = nn.Linear(concept_dim + 1, concept_dim)
        self.softmax = nn.Softmax(dim=-1)
        self.graph = nx.DiGraph()
        self.node_weights = {}

    def forward(self, x, state_id=None, task_id=None):
        concept_scores = self.encoder(x)
        concept_probs = self.softmax(concept_scores)
        meaning_score = torch.sum(concept_probs *
CONCEPT_WEIGHTS.to(x.device), dim=-1)

        if state_id is not None:
            if state_id not in self.graph:
                self.graph.add_node(state_id, type="state",
probs=concept_probs.detach().cpu().numpy(), tasks=[task_id])
                self.node_weights[state_id] = 1.0
            dominant_concept_idx = torch.argmax(concept_probs, dim=-1).item()
            concept = CONCEPTS[dominant_concept_idx]
            if concept not in self.graph:
                self.graph.add_node(concept, type="concept")
                self.node_weights[concept] =
CONCEPT_WEIGHTS[dominant_concept_idx].item()
            self.graph.add_edge(state_id, concept, relation="belongs_to",
weight=concept_probs[0, dominant_concept_idx].item())
            for related_concept in CONCEPT_RELATIONS.get(concept, []):
                if related_concept in self.graph:
                    self.graph.add_edge(concept, related_concept,
relation="supports", weight=0.5)

        return concept_probs, meaning_score

    def predict_next_concepts(self, concept_probs, action):
        action_tensor = torch.tensor([action],
dtype=torch.float32).to(concept_probs.device)
        input_transition = torch.cat([concept_probs,
action_tensor.unsqueeze(0)], dim=-1)
        next_concept_scores = self.transition_predictor(input_transition)
        return self.softmax(next_concept_scores)

    def update_graph(self, reward, task_id=None):
        for node in self.graph.nodes:
            if self.graph.nodes[node]["type"] == "state":
```

```
                self.node_weights[node] *= (1 + 0.1 * reward)
            elif self.graph.nodes[node]["type"] == "concept":
                if any(self.graph.has_edge(n, node) for n in self.graph.nodes
if self.graph.nodes[n]["type"] == "state"):
                    self.node_weights[node] *= 1.05
            self.node_weights[node] = min(max(self.node_weights[node], 0.1),
10.0)

        to_remove = []
        for node in self.graph.nodes:
            if self.graph.nodes[node]["type"] == "state" and
self.node_weights[node] < 0.2:
                if task_id != "task_1" and "task_1" in
self.graph.nodes[node].get("tasks", []):
                    continue
                to_remove.append(node)
            elif self.graph.nodes[node]["type"] == "concept":
                neighbors = list(self.graph.successors(node))
                if "chaos" in neighbors and "sagesse" in neighbors:
                    self.node_weights[node] *= 0.8
        self.graph.remove_nodes_from(to_remove)
        for node in to_remove:
            del self.node_weights[node]

    def train_step(self, state, reward, optimizer):
        concept_probs, _ = self.forward(state)
        target_concept = 6 if reward > 0 else 0  # charité vs danger
        target = torch.zeros(len(CONCEPTS), device=state.device)
        target[target_concept] = 1.0
        loss = F.cross_entropy(concept_probs, target.unsqueeze(0))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        return loss.item()

    def visualize_graph(self, filename="semantic_graph.png"):
        pos = nx.spring_layout(self.graph)
        node_colors = ['lightblue' if self.graph.nodes[n]["type"] == "state"
else 'lightgreen' for n in self.graph.nodes]
        node_sizes = [self.node_weights[n] * 100 for n in self.graph.nodes]
        plt.figure(figsize=(10, 8))
        nx.draw(self.graph, pos, with_labels=True, node_color=node_colors,
node_size=node_sizes, font_size=8, font_weight='bold')
        edge_labels = nx.get_edge_attributes(self.graph, 'relation')
        nx.draw_networkx_edge_labels(self.graph, pos,
edge_labels=edge_labels)
        plt.savefig(filename)
        plt.close()

class SymbolicLearningAgent:
    def __init__(self, symbolic_layer, env):
```

```
        self.symbolic_layer = symbolic_layer
        self.env = env
        self.memory = deque(maxlen=10000)
        self.optimizer = torch.optim.Adam(self.symbolic_layer.parameters(),
lr=0.001)
        self.policy_layer = nn.Sequential(
            nn.Linear(len(CONCEPTS), 64), nn.ReLU(),
            nn.Linear(64, env.action_space.n)
        )
        self.policy_optimizer =
torch.optim.Adam(self.policy_layer.parameters(), lr=0.001)
        self.epsilon = 0.1

    def choose_action(self, state):
        state_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
        state_id = str(hash(str(state)) % 1000)
        if random.random() < self.epsilon:
            return self.env.action_space.sample()
        concept_probs, meaning_score = self.symbolic_layer(state_tensor,
state_id=state_id)
        action_scores = self.policy_layer(concept_probs)
        action_probs = F.softmax(action_scores, dim=-1)
        action = torch.argmax(action_probs, dim=-1).item()
        return action

    def plan_action(self, state, horizon=3):
        state_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
        concept_probs, _ = self.symbolic_layer(state_tensor)
        best_action = None
        best_score = -float('inf')
        for action in range(self.env.action_space.n):
            current_probs = concept_probs
            score = 0
            for _ in range(horizon):
                current_probs =
self.symbolic_layer.predict_next_concepts(current_probs, action)
                score += torch.sum(current_probs *
CONCEPT_WEIGHTS.to(current_probs.device)).item()
            if score > best_score:
                best_score = score
                best_action = action
        return best_action

    def store_experience(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def train_policy(self, state, action, reward):
        state_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
        concept_probs, _ = self.symbolic_layer(state_tensor)
        action_scores = self.policy_layer(concept_probs)
        action_probs = F.softmax(action_scores, dim=-1)
```

```
        log_prob = torch.log(action_probs[0, action])
        loss = -log_prob * reward
        self.policy_optimizer.zero_grad()
        loss.backward()
        self.policy_optimizer.step()
        return loss.item()

    def train_world_model(self, batch_size=32):
        if len(self.memory) < batch_size:
            return
        batch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in batch:
            state_tensor = torch.tensor(state,
dtype=torch.float32).unsqueeze(0)
            next_state_tensor = torch.tensor(next_state,
dtype=torch.float32).unsqueeze(0)
            concept_probs, _ = self.symbolic_layer(state_tensor)
            next_concept_probs, _ = self.symbolic_layer(next_state_tensor)
            predicted_next_probs =
self.symbolic_layer.predict_next_concepts(concept_probs, action)
            loss = F.mse_loss(predicted_next_probs, next_concept_probs)
            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()

    def run_episode(self, task_name, task_id, max_steps=1000):
        state, _ = self.env.reset()
        total_reward = 0
        path = []
        self.epsilon = max(0.01, self.epsilon * 0.995)
        for _ in range(max_steps):
            state_id = str(hash(str(state)) % 1000)
            action = self.plan_action(state) if random.random() < 0.5 else
self.choose_action(state)
            next_state, reward, terminated, truncated, _ =
self.env.step(action)
            done = terminated or truncated
            self.store_experience(state, action, reward, next_state, done)

            state_tensor = torch.tensor(state,
dtype=torch.float32).unsqueeze(0)
            loss = self.symbolic_layer.train_step(state_tensor, reward,
self.optimizer)
            policy_loss = self.train_policy(state, action, reward)
            self.train_world_model()
            self.symbolic_layer.update_graph(reward, task_id=task_id)

            total_reward += reward
            path.append((state_id, action, reward))
            state = next_state
            if done:
```

```
                break

        self.symbolic_layer.visualize_graph(f"{task_name}_graph.png")
        print(f"{task_name} Total Reward: {total_reward}, Path Length:
{len(path)}")
        return path, total_reward

def main():
    env = gym.make("LunarLander-v2", gravity=-10.0)
    symbolic_layer = SymbolicLayer(input_dim=env.observation_space.shape[0])
    agent = SymbolicLearningAgent(symbolic_layer, env)

    print("Training on Task 1...")
    task1_rewards = []
    for episode in range(1000):
        path, reward = agent.run_episode(f"Task_1_Episode_{episode + 1}",
task_id="task_1")
        task1_rewards.append(reward)

    env = gym.make("LunarLander-v2", gravity=-15.0)
    print("\nTraining on Task 2...")
    task2_rewards = []
    for episode in range(1000):
        path, reward = agent.run_episode(f"Task_2_Episode_{episode + 1}",
task_id="task_2")
        task2_rewards.append(reward)

    env = gym.make("LunarLander-v2", gravity=-10.0)
    print("\nTesting on Task 1...")
    test_rewards = []
    for episode in range(200):
        path, reward = agent.run_episode(f"Task_1_Test_{episode + 1}",
task_id="test_task_1")
        test_rewards.append(reward)

    print(f"\nAverage Reward Task 1: {np.mean(task1_rewards)}")
    print(f"Average Reward Task 2: {np.mean(task2_rewards)}")
    print(f"Average Reward Test Task 1: {np.mean(test_rewards)}")
    print(f"Retention Rate: {np.mean(test_rewards) / np.mean(task1_rewards)
* 100:.2f}%")

if __name__ == "__main__":
    main()
```

Contact: ****maisldv@gmail.com\ **Affiliation:** Independent Researcher\

Keywords: Symbolic AI, Virtue Ethics, AGI, Semantic Graphs, Emmanuelle, S-Module, Moral Reasoning, Reinforcement Learning, Universal Values.