The Neuroscience of AI

Richard Young

2025

Chapter 22: Embodied AI and Robotics

22.0 Chapter Goals

- Understand how neuroscience principles inform embodied AI and robotics
- Explore brain-inspired approaches to sensorimotor learning and control
- Learn about predictive coding and internal models for robot control
- Implement a basic neuro-inspired robot controller

22.1 From Disembodied Algorithms to Grounded Intelligence

Most AI systems today exist as disembodied algorithms, processing data without direct interaction with the physical world. In contrast, human and animal intelligence developed through embodied interaction with the environment. Robotics and embodied AI aim to bridge this gap by developing systems that learn through physical interaction.

22.1.1 The Embodiment Hypothesis

The embodiment hypothesis posits that intelligence requires a body - that cognition is shaped by the sensorimotor experiences that come from having a physical form interacting with the world:

```python
class EmbodiedAgent:
    def __init__(self, sensors, actuators, environment):
        """
        Basic embodied agent with sensors and actuators

        Parameters:
        - sensors: Dictionary of sensor types and their properties
        - actuators: Dictionary of actuator types and their properties
        - environment: Environment the agent operates in
        """
        self.sensors = sensors
        self.actuators = actuators
        self.environment = environment
        self.sensor_history = []
        self.action_history = []
        self.internal_state = {}

    def sense(self):
        """Gather sensory information from environment"""
        current_sensory_data = {}

        for sensor_name, sensor in self.sensors.items():
            # Get raw sensory data from environment
            raw_data = self.environment.get_sensor_data(sensor)

            # Apply sensor-specific processing
            processed_data = sensor.process(raw_data)
            current_sensory_data[sensor_name] = processed_data

        # Store in history
        self.sensor_history.append(current_sensory_data)

        return current_sensory_data

    def act(self, action_commands):
        """Execute actions in the environment"""
        results = {}

        for actuator_name, command in action_commands.items():
            if actuator_name in self.actuators:
                # Execute command with the specified actuator
                result = self.actuators[actuator_name].execute(command)
                results[actuator_name] = result
            else:
                results[actuator_name] = "Error: Actuator not found"

        # Store in history
        self.action_history.append(action_commands)

        return results

    def learn_from_interaction(self):
        """Learn from sensorimotor interaction history"""
```

```
        if len(self.sensor_history) < 2:
            return "Not enough data for learning"

        # Analyze how actions led to sensory changes
        lessons = []
        for i in range(len(self.action_history)):
            if i + 1 < len(self.sensor_history):
                # Observe what sensory changes resulted from action
                action = self.action_history[i]
                state_before = self.sensor_history[i]
                state_after = self.sensor_history[i+1]

                # Learn relationships between actions and sensory changes
                for actuator_name, command in action.items():
                    for sensor_name in state_before:
                        # Calculate sensory difference
                        if sensor_name in state_after:
                            diff = self.calculate_difference(
                                state_before[sensor_name],
                                state_after[sensor_name]
                            )

                            lesson = {
                                'action': (actuator_name, command),
                                'sensor': sensor_name,
                                'effect': diff
                            }
                            lessons.append(lesson)

        # Update internal models based on observed action-effect pairs
        self.update_internal_models(lessons)

        return lessons

    def calculate_difference(self, before, after):
        """Calculate difference between sensory states (simplified)"""
        # This would be specialized based on sensor type
        if isinstance(before, (int, float)) and isinstance(after, (int, float)):
            return after - before
        else:
            return "Complex sensor type, difference calculation omitted"

    def update_internal_models(self, lessons):
        """Update internal models based on learning (placeholder)"""
        for lesson in lessons:
            action_key = f"{lesson['action'][0]}_{lesson['action'][1]}"
            sensor_key = lesson['sensor']
            effect = lesson['effect']

            # Simple aggregate model of effects
            model_key = f"{action_key}_{sensor_key}"
            if model_key not in self.internal_state:
                self.internal_state[model_key] = []
```

```python
            self.internal_state[model_key].append(effect)
```

Key principles from the embodiment hypothesis include:

1. **Sensorimotor coupling**: Intelligence emerges from the dynamic interaction between sensing and action
2. **Environmental scaffolding**: The environment provides structure that simplifies learning
3. **Body schema**: Internal models of the body's capabilities shape cognitive development

22.1.2 Brain-Inspired Control Architectures

Traditional robot control uses hierarchical architectures with clear separations between perception, planning, and action. Brain-inspired approaches favor more integrated architectures:

1. **Reactive-Deliberative Hybrid**: Combines fast reactive behaviors with slower deliberative planning
2. **Subsumption Architecture**: Layered behaviors where higher levels subsume lower levels
3. **Distributed Control**: Multiple specialized controllers coordinating without central executive

```python
class SubsumptionController:
    def __init__(self):
        """
        Implementation of Brooks' subsumption architecture
        with layered behaviors
        """
        self.behaviors = []  # Ordered from lowest to highest priority

    def add_behavior(self, behavior, priority):
        """Add a behavior at specified priority level"""
        self.behaviors.append((behavior, priority))
        # Sort behaviors by priority (highest value = highest priority)
        self.behaviors.sort(key=lambda x: x[1])

    def generate_action(self, sensory_input):
        """
        Generate action by evaluating behaviors in priority order

        Parameters:
        - sensory_input: Current sensory information

        Returns:
        - action: Selected action command
        """
        # Start with no action selected
        selected_action = None

        # Evaluate behaviors from lowest to highest priority
        for behavior, priority in self.behaviors:
            # Check if behavior is applicable
            if behavior.is_applicable(sensory_input):
                # Get action from this behavior
                action = behavior.compute_action(sensory_input)

                # Higher priority behaviors override (subsume) lower ones
                if action is not None:
                    selected_action = action

        return selected_action

class Behavior:
    def __init__(self, name):
        """
        Base class for robot behaviors

        Parameters:
        - name: Name of this behavior
        """
        self.name = name

    def is_applicable(self, sensory_input):
        """
        Check if this behavior applies to current situation
```

```
        Parameters:
        - sensory_input: Current sensory information

        Returns:
        - applicable: Whether behavior is applicable
        """
        # Must be implemented by subclasses
        raise NotImplementedError

    def compute_action(self, sensory_input):
        """
        Compute action based on sensory input

        Parameters:
        - sensory_input: Current sensory information

        Returns:
        - action: Action command
        """
        # Must be implemented by subclasses
        raise NotImplementedError

# Example behaviors
class ObstacleAvoidance(Behavior):
    def __init__(self, safety_distance=0.5):
        super().__init__("Obstacle Avoidance")
        self.safety_distance = safety_distance

    def is_applicable(self, sensory_input):
        # Check if any obstacle is within safety distance
        if 'proximity' in sensory_input:
            return min(sensory_input['proximity']) < self.safety_distance
        return False

    def compute_action(self, sensory_input):
        # Find direction with most clearance
        proximity_sensors = sensory_input['proximity']
        safest_direction = proximity_sensors.index(max(proximity_sensors))
        turn_amount = (safest_direction / len(proximity_sensors)) * 2 - 1  # -1 to 1

        return {
            'linear_velocity': 0.1,  # Slow down near obstacles
            'angular_velocity': turn_amount  # Turn away from obstacle
        }

class GoalSeeking(Behavior):
    def __init__(self, goal_position):
        super().__init__("Goal Seeking")
        self.goal_position = goal_position

    def is_applicable(self, sensory_input):
        # Always applicable if we have position data
        return 'position' in sensory_input
```

```python
    def compute_action(self, sensory_input):
        # Calculate vector to goal
        current_pos = sensory_input['position']
        goal_vector = [
            self.goal_position[0] - current_pos[0],
            self.goal_position[1] - current_pos[1]
        ]

        # Calculate heading to goal
        current_heading = sensory_input.get('heading', 0)
        goal_heading = math.atan2(goal_vector[1], goal_vector[0])
        heading_error = goal_heading - current_heading

        # Normalize to -pi to pi
        heading_error = ((heading_error + math.pi) % (2 * math.pi)) - math.pi

        return {
            'linear_velocity': 0.5,  # Move forward
            'angular_velocity': heading_error * 0.5  # Turn toward goal
        }
```

22.2 Neuroscience-Inspired Sensorimotor Control

The brain's motor control systems offer valuable inspiration for robotic control.

22.2.1 Motor Primitives and Synergies

Rather than controlling individual motor units separately, the brain organizes movement around motor primitives - coordinated patterns of muscle activation that can be combined to produce complex behaviors:

```
import numpy as np

class MotorPrimitiveController:
    def __init__(self, n_primitives=5, n_actuators=10):
        """
        Controller based on motor primitives/synergies

        Parameters:
        - n_primitives: Number of motor primitives
        - n_actuators: Number of actuators/motors to control
        """
        # Initialize random primitive patterns
        # In real applications, these would be learned or designed
        self.primitives = np.random.normal(0, 1, (n_primitives, n_actuators))

        # Normalize primitives
        for i in range(n_primitives):
            self.primitives[i] /= np.linalg.norm(self.primitives[i])

        self.n_primitives = n_primitives
        self.n_actuators = n_actuators

    def generate_movement(self, primitive_weights):
        """
        Generate actuator commands by combining primitives

        Parameters:
        - primitive_weights: Weights for each primitive

        Returns:
        - actuator_commands: Commands for each actuator
        """
        if len(primitive_weights) != self.n_primitives:
            raise ValueError(f"Expected {self.n_primitives} weights, got {len(primi

        # Combine primitives using weights
        actuator_commands = np.zeros(self.n_actuators)
        for i, weight in enumerate(primitive_weights):
            actuator_commands += weight * self.primitives[i]

        return actuator_commands

    def learn_new_primitive(self, sample_movements, n_iterations=100, learning_rate=
        """
        Learn a new motor primitive from example movements

        Parameters:
        - sample_movements: Array of sample actuator commands
        - n_iterations: Number of learning iterations
        - learning_rate: Learning rate for updates

        Returns:
        - new_primitive: The newly learned primitive
```

```python
        """
        # Initialize new primitive randomly
        new_primitive = np.random.normal(0, 0.1, self.n_actuators)
        new_primitive /= np.linalg.norm(new_primitive)

        # Learn through dimensionality reduction (simplified)
        for _ in range(n_iterations):
            for movement in sample_movements:
                # Project movement onto current primitive
                projection = np.dot(movement, new_primitive)

                # Reconstruction error
                error = movement - projection * new_primitive

                # Update primitive to reduce error
                gradient = learning_rate * np.dot(error, movement)
                new_primitive += gradient

                # Normalize
                new_primitive /= np.linalg.norm(new_primitive)

        # Add to primitive set
        self.primitives = np.vstack((self.primitives, new_primitive))
        self.n_primitives += 1

        return new_primitive
```

This approach offers several advantages:

1. **Dimensionality reduction**: Control complexity is reduced from many actuators to fewer primitives
2. **Generalization**: New movements can be generated by recombining existing primitives
3. **Robustness**: Primitive-based control is less sensitive to individual actuator failures

22.2.2 Predictive Coding and Forward Models

The brain uses predictive coding to anticipate the sensory consequences of actions. Similar predictive mechanisms can vastly improve robot control:

```
class PredictiveController:
    def __init__(self, n_inputs, n_outputs, learning_rate=0.01):
        """
        Controller implementing predictive coding principles

        Parameters:
        - n_inputs: Dimension of input (sensory) space
        - n_outputs: Dimension of output (motor) space
        - learning_rate: Learning rate for model updates
        """
        # Initialize forward model (predicts sensory consequences of actions)
        self.W_forward = np.random.normal(0, 0.1, (n_inputs, n_outputs))

        # Initialize inverse model (maps desired sensory states to actions)
        self.W_inverse = np.random.normal(0, 0.1, (n_outputs, n_inputs))

        self.learning_rate = learning_rate
        self.prediction_errors = []

    def predict_sensory_outcome(self, action):
        """
        Predict sensory outcome of an action

        Parameters:
        - action: Motor command

        Returns:
        - predicted_sensation: Predicted sensory feedback
        """
        return np.dot(self.W_forward, action)

    def generate_action(self, current_sensation, target_sensation):
        """
        Generate action to achieve target sensory state

        Parameters:
        - current_sensation: Current sensory state
        - target_sensation: Desired sensory state

        Returns:
        - action: Motor command
        """
        # Calculate sensory prediction error
        error = target_sensation - current_sensation

        # Use inverse model to generate action
        action = np.dot(self.W_inverse, error)

        return action

    def update_models(self, action, predicted_sensation, actual_sensation):
        """
        Update forward and inverse models based on prediction error
```

```
        Parameters:
        - action: Motor command that was executed
        - predicted_sensation: Sensation that was predicted
        - actual_sensation: Sensation that was actually experienced

        Returns:
        - prediction_error: Magnitude of prediction error
        """
        # Calculate prediction error
        prediction_error = actual_sensation - predicted_sensation
        error_magnitude = np.linalg.norm(prediction_error)
        self.prediction_errors.append(error_magnitude)

        # Update forward model
        update = self.learning_rate * np.outer(prediction_error, action)
        self.W_forward += update

        # Update inverse model
        update = self.learning_rate * np.outer(action, prediction_error)
        self.W_inverse += update

        return error_magnitude

    def learning_curve(self):
        """Plot learning curve of prediction errors"""
        import matplotlib.pyplot as plt

        plt.figure(figsize=(10, 6))
        plt.plot(self.prediction_errors)
        plt.xlabel('Learning Step')
        plt.ylabel('Prediction Error')
        plt.title('Predictive Model Learning Curve')
        plt.grid(True)

        return plt
```

Forward models enable critical capabilities:

1. **Sensory cancellation**: Distinguishing external stimuli from self-generated sensations
2. **Closed-loop control**: Compensating for discrepancies between predicted and actual outcomes
3. **Mental simulation**: Planning actions by simulating their outcomes before execution

22.2.3 Cerebellar-Inspired Adaptive Control

The cerebellum is critical for motor learning and coordination. Cerebellar-inspired control models can enable robots to adapt to changing dynamics:

```
class CerebellarAdaptiveController:
    def __init__(self, n_inputs, n_outputs, n_granule_cells=1000):
        """
        Cerebellar-inspired adaptive controller

        Parameters:
        - n_inputs: Number of mossy fiber inputs (sensory state variables)
        - n_outputs: Number of outputs (motor commands)
        - n_granule_cells: Number of granule cells for sparse expansion
        """
        self.n_inputs = n_inputs
        self.n_outputs = n_outputs
        self.n_granule_cells = n_granule_cells

        # Connectivity from mossy fibers to granule cells (random projection)
        self.W_mossy_granule = np.random.normal(0, 1, (n_granule_cells, n_inputs))

        # Weights from granule cells to Purkinje cells (learnable)
        self.W_granule_purkinje = np.random.normal(0, 0.1, (n_outputs, n_granule_ce

        # Learning rate for parallel fiber-Purkinje cell synapses
        self.learning_rate = 0.01

        # History of corrections
        self.correction_history = []

    def granule_cell_activity(self, mossy_input):
        """
        Calculate granule cell activity from mossy fiber input

        Parameters:
        - mossy_input: Input from mossy fibers (sensory state)

        Returns:
        - granule_activity: Activity of granule cells
        """
        # Perform sparse expansion
        raw_activity = np.dot(self.W_mossy_granule, mossy_input)

        # ReLU activation
        activity = np.maximum(raw_activity, 0)

        # Normalization and sparsification (approximate k-winners-take-all)
        k = int(self.n_granule_cells * 0.05)  # 5% activity
        threshold = np.sort(activity)[-k] if k > 0 else 0
        activity[activity < threshold] = 0

        return activity

    def compute_correction(self, state):
        """
        Compute corrective signal for feedforward control
```

```
        Parameters:
        - state: Current sensory state

        Returns:
        - correction: Corrective motor command
        """
        # Expand input through granule cells
        granule_activity = self.granule_cell_activity(state)

        # Compute cerebellar correction
        correction = np.dot(self.W_granule_purkinje, granule_activity)

        self.correction_history.append(np.linalg.norm(correction))

        return correction

    def learn_from_error(self, state, error):
        """
        Learn from motor error (climbing fiber input)

        Parameters:
        - state: Sensory state during error
        - error: Motor error/climbing fiber signal
        """
        # Get granule cell activity for this state
        granule_activity = self.granule_cell_activity(state)

        # Update weights according to correlation of granule activity with error
        # This follows the cerebellum's supervised learning rule
        for i in range(self.n_outputs):
            delta_w = -self.learning_rate * error[i] * granule_activity
            self.W_granule_purkinje[i, :] += delta_w

    def plot_learning(self):
        """Plot correction magnitude over time"""
        import matplotlib.pyplot as plt

        plt.figure(figsize=(10, 6))
        plt.plot(self.correction_history)
        plt.xlabel('Trial')
        plt.ylabel('Correction Magnitude')
        plt.title('Cerebellar Adaptation')
        plt.grid(True)

        return plt
```

The cerebellum’s architecture enables specific computational features:

1. **Sparse coding**: Granule cells expand the input space while maintaining sparse activity
2. **Supervised learning**: Climbing fiber error signals guide synaptic weight updates
3. **Timing mechanisms**: Precise timing of movements through internal timing circuitry

22.3 Affordances and Action-Centered Perception

The concept of affordances, introduced by psychologist J.J. Gibson, refers to the action possibilities that objects or environments offer. This ecological approach to perception has influenced embodied AI:

```
class AffordanceDetector:
    def __init__(self):
        """
        System to detect action affordances in visual scenes
        """
        self.affordance_types = {
            'graspable': {'min_size': 0.03, 'max_size': 0.2, 'convexity': 0.7},
            'pushable': {'min_size': 0.05, 'max_size': 0.5, 'flat_side': True},
            'liftable': {'min_size': 0.03, 'max_size': 0.3, 'weight_est': 1.0},
            'sittable': {'min_size': 0.3, 'max_size': 1.0, 'horizontal_surface': Tru
        }

        # Tracks learned affordance-action associations
        self.action_success_history = {}

    def detect_affordances(self, objects, robot_capabilities):
        """
        Detect affordances of objects based on their properties

        Parameters:
        - objects: List of objects with properties
        - robot_capabilities: Dict of robot capabilities and limitations

        Returns:
        - affordances: Dict mapping objects to their affordances
        """
        affordances = {}

        for obj in objects:
            obj_affordances = []

            for affordance_name, requirements in self.affordance_types.items():
                # Check if object meets requirements for this affordance
                qualifies = True

                for req_name, req_value in requirements.items():
                    if req_name not in obj:
                        qualifies = False
                        break

                    if req_name == 'min_size' and obj['size'] < req_value:
                        qualifies = False
                        break
                    elif req_name == 'max_size' and obj['size'] > req_value:
                        qualifies = False
                        break
                    elif isinstance(req_value, bool) and obj[req_name] != req_value
                        qualifies = False
                        break

                # Check if robot can execute the affordance
                if qualifies:
                    if affordance_name == 'liftable' and obj.get('weight_est', 0) >
```

```python
                    qualifies = False

                if affordance_name == 'graspable' and obj.get('size', 0) > robo
                    qualifies = False

            if qualifies:
                obj_affordances.append(affordance_name)

        affordances[obj['id']] = obj_affordances

    return affordances

def affordance_to_action(self, obj, affordance):
    """
    Convert an affordance to specific action parameters

    Parameters:
    - obj: Object with the affordance
    - affordance: Affordance type

    Returns:
    - action: Action parameters
    """
    if affordance == 'graspable':
        # Determine grasp parameters based on object properties
        width = min(obj.get('width', 0.1), 0.08)  # Max gripper width
        height = obj.get('height', 0) / 2  # Grasp in the middle

        return {
            'action_type': 'grasp',
            'position': obj['position'],
            'approach_vector': [0, 0, 1],  # Approach from above
            'gripper_width': width,
            'grasp_height': height
        }

    elif affordance == 'pushable':
        # Determine pushing parameters
        push_dir = [1, 0, 0]  # Default push direction
        if 'preferred_push_dir' in obj:
            push_dir = obj['preferred_push_dir']

        return {
            'action_type': 'push',
            'position': obj['position'],
            'push_direction': push_dir,
            'push_distance': 0.1
        }

    # Other affordances would have their own action mappings

    return {'action_type': 'none'}

def update_from_experience(self, obj, affordance, action, success):
```

```
        """
        Update affordance models from action experience

        Parameters:
        - obj: Object that was acted upon
        - affordance: Affordance that was utilized
        - action: Action that was performed
        - success: Whether the action was successful
        """
        obj_id = obj['id']

        # Initialize history for this object if needed
        if obj_id not in self.action_success_history:
            self.action_success_history[obj_id] = {}

        # Initialize history for this affordance if needed
        if affordance not in self.action_success_history[obj_id]:
            self.action_success_history[obj_id][affordance] = []

        # Record success/failure
        self.action_success_history[obj_id][affordance].append({
            'action': action,
            'success': success,
            'object_state': obj.copy()
        })

        # Update affordance requirements based on experience
        if len(self.action_success_history[obj_id][affordance]) >= 5:
            self._refine_affordance_model(affordance)

    def _refine_affordance_model(self, affordance):
        """Refine affordance model based on action history (placeholder)"""
        # This would analyze success/failure patterns to refine affordance detectio
        # For example, adjusting size thresholds, weight limits, etc.
        pass
```

Affordance-based approaches offer several benefits:

1. **Bridging perception and action**: Direct mapping from perceptual features to action possibilities
2. **Efficiency**: Focus on action-relevant properties rather than full scene understanding
3. **Transfer learning**: Affordances can generalize across objects with similar action-relevant properties

22.4 Human-Robot Collaboration and Social

Robotics

Robots increasingly need to collaborate with humans, requiring social capabilities inspired by neuroscience research on social cognition.

22.4.1 Shared Control and Collaborative Tasks

Effective human-robot collaboration requires shared understanding of tasks and goals:

```python
class CollaborativeRobot:
    def __init__(self):
        """
        Robot system designed for human collaboration
        """
        # Task models - structured representations of collaborative tasks
        self.task_models = {}

        # Joint attention system
        self.attention_system = JointAttentionSystem()

        # Intent prediction model
        self.intent_predictor = IntentPredictor()

        # Adaptive action system
        self.action_planner = AdaptiveActionPlanner()

        # Communication module
        self.communicator = FeedbackCommunicator()

    def observe_human(self, human_state):
        """
        Process observations of human collaborator

        Parameters:
        - human_state: Dict with human pose, gaze, actions, etc.

        Returns:
        - processed_state: Processed human state information
        """
        # Track human attention
        attended_location = self.attention_system.infer_attention(
            human_state.get('head_pose'),
            human_state.get('gaze_direction')
        )

        # Predict human intent
        intent = self.intent_predictor.predict(
            human_state,
            history=self.intent_predictor.history
        )

        # Update internal state
        processed_state = {
            'attention': attended_location,
            'intent': intent,
            'action': human_state.get('action'),
            'pose': human_state.get('pose')
        }

        return processed_state

    def generate_complementary_action(self, human_state, task_context):
```

```
        """
        Generate action that complements human's activity

        Parameters:
        - human_state: Processed human state
        - task_context: Current task context

        Returns:
        - action: Action parameters
        """
        # Select active task model
        task_model = self.task_models.get(task_context['task_id'])
        if not task_model:
            return {'action_type': 'idle'}

        # Determine role allocation
        human_subtask = human_state['intent'].get('subtask')

        # Find complementary subtask
        robot_subtask = task_model.get_complementary_subtask(human_subtask)

        # Plan action for this subtask
        action = self.action_planner.plan(
            robot_subtask,
            human_state,
            task_context
        )

        # Generate appropriate feedback
        feedback = self.communicator.generate_feedback(
            action,
            human_state,
            task_context
        )

        # Combine action and feedback
        action['feedback'] = feedback

        return action

    def learn_from_demonstration(self, demonstration_data):
        """
        Learn new task model from human demonstration

        Parameters:
        - demonstration_data: Recorded human demonstration

        Returns:
        - task_id: ID of the learned task
        """
        # Extract task structure
        task_id = demonstration_data['task_id']
        task_steps = []
```

```python
        for step in demonstration_data['steps']:
            # Analyze step components
            step_model = {
                'objects': step['objects'],
                'actions': step['actions'],
                'preconditions': self._infer_preconditions(step),
                'effects': self._infer_effects(step),
                'constraints': self._infer_constraints(step)
            }
            task_steps.append(step_model)

        # Create new task model
        self.task_models[task_id] = TaskModel(
            task_id=task_id,
            steps=task_steps,
            objects=demonstration_data['objects'],
            goal=demonstration_data['goal']
        )

        return task_id

    def _infer_preconditions(self, step_data):
        """Infer preconditions from step data (placeholder)"""
        # This would analyze state before the step
        return {'placeholder': 'preconditions'}

    def _infer_effects(self, step_data):
        """Infer effects from step data (placeholder)"""
        # This would analyze state changes after the step
        return {'placeholder': 'effects'}

    def _infer_constraints(self, step_data):
        """Infer constraints from step data (placeholder)"""
        # This would identify constraints on execution
        return {'placeholder': 'constraints'}

# Placeholder supporting classes
class JointAttentionSystem:
    def __init__(self):
        self.history = []

    def infer_attention(self, head_pose, gaze_direction):
        """Infer where human is attending based on head pose and gaze"""
        # Placeholder implementation
        if not gaze_direction:
            return None

        # Simple projection of gaze ray
        attention_point = [
            head_pose[0] + gaze_direction[0] * 2,
            head_pose[1] + gaze_direction[1] * 2,
            head_pose[2] + gaze_direction[2] * 2
        ]
```

```python
        self.history.append(attention_point)
        return attention_point

class IntentPredictor:
    def __init__(self):
        self.history = []

    def predict(self, human_state, history=None):
        """Predict human intent from state and history"""
        # Placeholder implementation
        return {'subtask': 'unknown', 'confidence': 0.5}

class AdaptiveActionPlanner:
    def plan(self, subtask, human_state, task_context):
        """Plan action for a subtask considering human state"""
        # Placeholder implementation
        return {'action_type': subtask, 'parameters': {}}

class FeedbackCommunicator:
    def generate_feedback(self, action, human_state, task_context):
        """Generate appropriate feedback for current action"""
        # Placeholder implementation
        return {'type': 'visual_cue', 'message': 'Robot is helping'}

class TaskModel:
    def __init__(self, task_id, steps, objects, goal):
        self.task_id = task_id
        self.steps = steps
        self.objects = objects
        self.goal = goal

    def get_complementary_subtask(self, human_subtask):
        """Find complementary subtask to what human is doing"""
        # Placeholder implementation
        return "support_human"
```

Social cues and signals play a critical role in collaboration:

1. **Joint attention**: The ability to share attention on the same object or location
2. **Action prediction**: Anticipating the partner's next move to coordinate effectively
3. **Adaptive behavior**: Adjusting plans and actions based on the partner's state

22.4.2 Learning from Demonstration

Robots can learn by observing human demonstrations, leveraging neural mechanisms for imitation learning:

```python
import numpy as np
from sklearn.mixture import GaussianMixture

class LearningFromDemonstration:
    def __init__(self):
        """
        System for learning actions from human demonstrations
        """
        self.action_models = {}
        self.context_classifier = None

    def process_demonstrations(self, demonstrations):
        """
        Process multiple demonstrations of the same task

        Parameters:
        - demonstrations: List of demonstration trajectories

        Returns:
        - model_id: ID of the learned action model
        """
        # Extract context features for all demonstrations
        contexts = [self._extract_context(demo) for demo in demonstrations]

        # Extract trajectories for all demonstrations
        trajectories = [self._extract_trajectory(demo) for demo in demonstrations]

        # Train context classifier
        self._train_context_classifier(contexts)

        # Learn action model from trajectories
        model_id = f"action_model_{len(self.action_models)}"

        # Use probabilistic model to represent demonstrated actions
        self.action_models[model_id] = self._train_action_model(trajectories)

        return model_id

    def generate_action(self, context, start_state):
        """
        Generate action trajectory for given context

        Parameters:
        - context: Current context features
        - start_state: Starting state for action generation

        Returns:
        - trajectory: Generated action trajectory
        """
        # Classify context to find most relevant action model
        model_id = self._classify_context(context)

        if model_id not in self.action_models:
```

```
            return None

        # Use action model to generate trajectory
        model = self.action_models[model_id]
        trajectory = self._generate_from_model(model, start_state)

        return trajectory

    def _extract_context(self, demonstration):
        """Extract context features from demonstration (placeholder)"""
        # This would extract relevant context features
        # such as object positions, relationships, etc.
        if 'context' in demonstration:
            return demonstration['context']
        return {}

    def _extract_trajectory(self, demonstration):
        """Extract normalized trajectory from demonstration"""
        # Get raw trajectory points
        if 'trajectory' in demonstration:
            traj = demonstration['trajectory']

            # Normalize time
            t = np.array([point.get('time', i) for i, point in enumerate(traj)])
            t = (t - t.min()) / (t.max() - t.min())

            # Extract state variables
            positions = np.array([point.get('position', [0, 0, 0]) for point in tra
            velocities = np.array([point.get('velocity', [0, 0, 0]) for point in tra

            # Combine into state-time pairs
            trajectory = np.column_stack((t, positions, velocities))
            return trajectory

        return np.array([])

    def _train_context_classifier(self, contexts):
        """Train classifier to map contexts to action models (placeholder)"""
        # This would train a classifier to map context features
        # to appropriate action models
        self.context_classifier = {"placeholder": "classifier"}

    def _train_action_model(self, trajectories):
        """Train probabilistic model from demonstrated trajectories"""
        # Combine all trajectories
        all_points = np.vstack(trajectories)

        # Fit GMM to capture distribution of states over time
        n_components = min(10, len(all_points) // 10)  # Heuristic
        gmm = GaussianMixture(n_components=n_components)
        gmm.fit(all_points)

        # Create dynamic model (placeholder)
        model = {
```

```python
            'gmm': gmm,
            'trajectories': trajectories
        }

        return model

    def _classify_context(self, context):
        """Classify context to select action model (placeholder)"""
        # This would use the context classifier to select
        # the most appropriate action model
        return "action_model_0"  # Default

    def _generate_from_model(self, model, start_state):
        """Generate trajectory from probabilistic model (placeholder)"""
        # This would generate a new trajectory from the model
        # conditioned on the start state
        gmm = model['gmm']

        # Sample trajectory (highly simplified)
        n_points = 20
        t = np.linspace(0, 1, n_points).reshape(-1, 1)

        trajectory = []

        # Simple generation heuristic
        current_state = np.array(start_state)

        for time_point in t:
            # Predict next state
            state_with_time = np.concatenate(([time_point[0]], current_state))

            # Sample from GMM (simplified)
            means = gmm.means_
            weights = gmm.weights_

            # Find closest component in time
            time_diffs = np.abs(means[:, 0] - time_point[0])
            closest_idx = np.argmin(time_diffs)

            # Use mean of closest component
            next_state = means[closest_idx, 1:]

            # Add to trajectory
            trajectory.append({
                'time': float(time_point[0]),
                'position': next_state[:3].tolist(),
                'velocity': next_state[3:].tolist()
            })

            current_state = next_state

        return trajectory
```

Learning from demonstration taps into mechanisms similar to those humans use for social learning:

1. **Goal inference**: Understanding the demonstrator's objective
2. **Context sensitivity**: Learning when and how to apply the demonstrated skills
3. **Generalization**: Adapting learned skills to new situations

22.5 Code Lab: Simple Neuro-Inspired Robot Controller

Let's implement a simplified robot controller that integrates several brain-inspired principles:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

class NeuroBotController:
    def __init__(self):
        """
        Neuro-inspired robot controller integrating multiple brain-like mechanisms
        """
        # Motor primitive system
        self.n_primitives = 4
        self.n_motors = 2  # Left and right wheel motors
        self.primitives = self._create_primitives()

        # Predictive forward model
        self.W_forward = np.random.normal(0, 0.1, (3, 2))  # Predicts [dx, dy, dθ]

        # Cerebellar adaptive component
        self.W_cerebellum = np.zeros((2, 10))  # Corrective signal for motor comman

        # Sensory system
        self.n_sensors = 5  # 5 directional distance sensors

        # Place cell system for navigation
        self.place_cells = self._create_place_cells(20)  # 20 place cells

        # Learning rates
        self.forward_lr = 0.01
        self.cerebellum_lr = 0.005

        # Experience buffer
        self.experience_buffer = []

        # Performance metrics
        self.prediction_errors = []

    def _create_primitives(self):
        """Create set of motor primitives"""
        primitives = np.zeros((self.n_primitives, self.n_motors))

        # Forward motion
        primitives[0] = [1.0, 1.0]

        # Turn left
        primitives[1] = [0.2, 1.0]

        # Turn right
        primitives[2] = [1.0, 0.2]

        # Backward motion
        primitives[3] = [-0.5, -0.5]

        # Normalize
```

```python
        for i in range(self.n_primitives):
            primitives[i] /= np.linalg.norm(primitives[i])

        return primitives

    def _create_place_cells(self, n_cells):
        """Create place cell centers in 2D environment"""
        # Randomly distribute place cells in 10×10 environment
        centers = np.random.uniform(0, 10, (n_cells, 2))

        # Each place cell has a preferred heading direction
        preferred_dirs = np.random.uniform(0, 2*np.pi, n_cells)

        # Combine into place cell definitions
        place_cells = np.column_stack((centers, preferred_dirs))

        return place_cells

    def sense_environment(self, position, heading, obstacles):
        """
        Get sensory input from environment

        Parameters:
        - position: [x, y] position
        - heading: Heading direction in radians
        - obstacles: List of obstacle positions

        Returns:
        - sensor_readings: Distance readings from sensors
        """
        sensor_readings = np.ones(self.n_sensors) * 10.0  # Max range = 10 units

        # Sensor angles relative to heading
        sensor_angles = np.linspace(-np.pi/2, np.pi/2, self.n_sensors)

        # Check distance to each obstacle along each sensor direction
        for obstacle in obstacles:
            obs_vec = np.array(obstacle) - np.array(position)
            distance = np.linalg.norm(obs_vec)

            # Direction to obstacle in global frame
            obs_dir = np.arctan2(obs_vec[1], obs_vec[0])

            # Convert to robot frame
            rel_angle = (obs_dir - heading + np.pi) % (2*np.pi) - np.pi

            # Find closest sensor
            sensor_idx = np.argmin(np.abs(sensor_angles - rel_angle))

            # Update sensor if obstacle is within its field of view and closer than
            angle_diff = abs(sensor_angles[sensor_idx] - rel_angle)
            if angle_diff < 0.2 and distance < sensor_readings[sensor_idx]:
                sensor_readings[sensor_idx] = distance
```

```
        return sensor_readings

    def activate_place_cells(self, position, heading):
        """
        Calculate activation of place cells for current position

        Parameters:
        - position: [x, y] position
        - heading: Heading direction in radians

        Returns:
        - activations: Activation level of each place cell
        """
        # Calculate distance to each place cell center
        distances = np.sqrt(np.sum((self.place_cells[:, :2] - position)**2, axis=1)

        # Place cell activation falls off with distance (Gaussian)
        spatial_activation = np.exp(-(distances**2) / 4.0)

        # Heading direction modulation
        heading_diffs = np.abs(self.place_cells[:, 2] - heading)
        heading_diffs = np.minimum(heading_diffs, 2*np.pi - heading_diffs)
        heading_activation = np.exp(-(heading_diffs**2) / 1.0)

        # Combine spatial and heading modulation
        activations = spatial_activation * heading_activation

        return activations

    def generate_motor_command(self, sensor_readings, place_cell_activations, goal_
        """
        Generate motor command using multiple neural systems

        Parameters:
        - sensor_readings: Current sensor readings
        - place_cell_activations: Current place cell activations
        - goal_position: [x, y] goal position

        Returns:
        - motor_command: Commands for left and right motors
        """
        # 1. Reactive obstacle avoidance
        danger_level = 1.0 / (sensor_readings + 0.1)
        left_danger = np.sum(danger_level[:self.n_sensors//2+1])
        right_danger = np.sum(danger_level[self.n_sensors//2:])

        # Turn away from obstacles
        obstacle_response = np.zeros(self.n_motors)
        if left_danger > right_danger:
            obstacle_response = self.primitives[2]  # Turn right
        elif right_danger > left_danger:
            obstacle_response = self.primitives[1]  # Turn left

        # 2. Goal-seeking behavior
```

```
        # Compute goal direction and activation
        goal_vec = np.array(goal_position) - np.array([place_cell_activations.sum()
        goal_distance = np.linalg.norm(goal_vec)

        goal_seeking = np.zeros(self.n_motors)
        if goal_distance > 0.5:
            # Forward motion with bias toward goal
            goal_seeking = self.primitives[0]  # Forward

            # Add turning bias
            if goal_vec[0] > 0:
                goal_seeking += 0.3 * self.primitives[2]  # Right bias
            else:
                goal_seeking += 0.3 * self.primitives[1]  # Left bias

        # 3. Cerebellar correction
        # Expand sensory input using random projection (cerebellum-like)
        expanded_input = np.tanh(np.random.normal(0, 1, (10, self.n_sensors)) @ sen
        cerebellar_correction = self.W_cerebellum @ expanded_input

        # Integrate all components
        # Obstacle avoidance has highest priority
        alpha = 0.7  # Weight for obstacle avoidance vs. goal seeking
        beta = 0.2   # Weight for cerebellar correction

        motor_command = (
            alpha * obstacle_response +
            (1-alpha) * goal_seeking +
            beta * cerebellar_correction
        )

        # Normalize command to valid range [-1, 1]
        motor_command = np.clip(motor_command, -1, 1)

        return motor_command

    def predict_next_state(self, current_state, motor_command):
        """
        Predict next state using forward model

        Parameters:
        - current_state: [x, y, θ] current position and heading
        - motor_command: [left, right] motor commands

        Returns:
        - predicted_next_state: Predicted next state
        """
        # Predict state change
        predicted_change = self.W_forward @ motor_command

        # Calculate next state
        predicted_next_state = current_state + predicted_change

        return predicted_next_state
```

```python
    def update_models(self, current_state, motor_command, next_state):
        """
        Update internal models based on experience

        Parameters:
        - current_state: [x, y, θ] state before action
        - motor_command: [left, right] executed motor command
        - next_state: [x, y, θ] state after action

        Returns:
        - prediction_error: Magnitude of prediction error
        """
        # Observed state change
        actual_change = next_state - current_state

        # Predicted state change
        predicted_change = self.W_forward @ motor_command

        # Prediction error
        prediction_error = actual_change - predicted_change
        error_magnitude = np.linalg.norm(prediction_error)
        self.prediction_errors.append(error_magnitude)

        # Update forward model
        self.W_forward += self.forward_lr * np.outer(prediction_error, motor_comman

        # Store experience for later learning
        self.experience_buffer.append({
            'state': current_state,
            'action': motor_command,
            'next_state': next_state,
            'prediction_error': prediction_error
        })

        # Limit buffer size
        if len(self.experience_buffer) > 100:
            self.experience_buffer.pop(0)

        # Update cerebellar model from random experiences (replay)
        if len(self.experience_buffer) > 10:
            # Sample random experience
            idx = np.random.randint(0, len(self.experience_buffer))
            exp = self.experience_buffer[idx]

            # Expand sensory representation
            expanded_input = np.tanh(np.random.normal(0, 1, (10, self.n_sensors))
                                     @ np.ones(self.n_sensors))  # Simplified

            # Update cerebellar weights to reduce prediction error
            self.W_cerebellum += self.cerebellum_lr * np.outer(exp['prediction_erro

        return error_magnitude
```

```python
    def visualize_trajectory(self, trajectory, obstacles=None, goal=None):
        """
        Visualize robot trajectory

        Parameters:
        - trajectory: List of [x, y, θ] states
        - obstacles: Optional list of obstacle positions
        - goal: Optional goal position

        Returns:
        - fig: Matplotlib figure
        """
        fig, ax = plt.subplots(figsize=(10, 10))

        # Plot obstacles
        if obstacles:
            for obs in obstacles:
                circle = plt.Circle(obs, 0.5, color='red', alpha=0.5)
                ax.add_patch(circle)

        # Plot goal
        if goal:
            circle = plt.Circle(goal, 0.5, color='green', alpha=0.5)
            ax.add_patch(circle)

        # Plot trajectory
        trajectory = np.array(trajectory)
        ax.plot(trajectory[:, 0], trajectory[:, 1], 'b-')

        # Robot representation
        robot = plt.Rectangle((0, 0), 0.5, 0.3, color='blue', alpha=0.5)
        ax.add_patch(robot)

        def update(i):
            if i < len(trajectory):
                x, y, θ = trajectory[i]
                # Update robot position and orientation
                robot.set_xy([x - 0.25, y - 0.15])
                trans = plt.matplotlib.transforms.Affine2D().rotate_around(x, y, θ)
                robot.set_transform(trans + ax.transData)
            return robot,

        # Create animation
        anim = FuncAnimation(fig, update, frames=len(trajectory),
                             interval=100, blit=True)

        # Set plot limits and labels
        ax.set_xlim(0, 10)
        ax.set_ylim(0, 10)
        ax.set_xlabel('X')
        ax.set_ylabel('Y')
        ax.set_title('Robot Trajectory')
        ax.set_aspect('equal')
```

```python
        return fig, anim

    def plot_learning_curve(self):
        """Plot learning curve of prediction errors"""
        fig, ax = plt.subplots(figsize=(10, 6))
        ax.plot(self.prediction_errors)
        ax.set_xlabel('Learning Step')
        ax.set_ylabel('Prediction Error')
        ax.set_title('Forward Model Learning Curve')
        ax.grid(True)

        return fig

# Example usage
def run_robot_simulation(controller, n_steps=100):
    """
    Run robot simulation with the neuro-inspired controller

    Parameters:
    - controller: NeuroBotController instance
    - n_steps: Number of simulation steps

    Returns:
    - trajectory: Recorded robot trajectory
    """
    # Environment setup
    obstacles = [
        [3, 3],
        [5, 7],
        [7, 4]
    ]
    goal = [8, 8]

    # Initial state
    state = np.array([1.0, 1.0, 0.0])  # [x, y, θ]
    trajectory = [state.copy()]

    # Simulation loop
    for _ in range(n_steps):
        # Sense environment
        sensor_readings = controller.sense_environment(
            state[:2], state[2], obstacles
        )

        # Activate place cells
        place_cell_activations = controller.activate_place_cells(
            state[:2], state[2]
        )

        # Generate motor command
        motor_command = controller.generate_motor_command(
            sensor_readings, place_cell_activations, goal
        )
```

```python
        # Predict next state
        predicted_next_state = controller.predict_next_state(
            state, motor_command
        )

        # Simulate robot movement (simplified physics)
        # In real robot, this would be the actual physical movement
        next_state = state.copy()

        # Convert motor commands to motion
        speed = (motor_command[0] + motor_command[1]) / 2.0
        turn_rate = (motor_command[1] - motor_command[0]) / 0.5

        # Update position and orientation
        next_state[2] += turn_rate * 0.1  # Update heading
        next_state[0] += speed * np.cos(next_state[2]) * 0.1  # Update x
        next_state[1] += speed * np.sin(next_state[2]) * 0.1  # Update y

        # Boundary checks (keep within environment)
        next_state[:2] = np.clip(next_state[:2], 0, 10)

        # Update models
        controller.update_models(state, motor_command, next_state)

        # Update state
        state = next_state.copy()
        trajectory.append(state.copy())

        # Check if goal reached
        if np.linalg.norm(state[:2] - goal) < 0.5:
            break

    return trajectory

# Create controller and run simulation
# controller = NeuroBotController()
# trajectory = run_robot_simulation(controller, 200)
# fig, anim = controller.visualize_trajectory(trajectory,
#                                             obstacles=[[3, 3], [5, 7], [7, 4]],
#                                             goal=[8, 8])
# learning_fig = controller.plot_learning_curve()
```

This example integrates several neuro-inspired mechanisms:

1. **Motor primitives**: Basic movement patterns similar to the brain's movement synergies
2. **Forward models**: Predictive coding for anticipating the results of actions
3. **Cerebellar-like correction**: Adaptive fine-tuning of movements based on experience
4. **Place cells**: Hippocampus-inspired spatial representation for navigation

22.6 Take-aways

- **Embodiment is essential for intelligence**: Physical interaction with the environment shapes learning and cognition
- **Brain-inspired control architectures** offer alternatives to traditional planning-sensing-acting loops
- **Predictive coding** principles can improve control by anticipating the sensory consequences of actions
- **Motor primitives** provide an efficient framework for complex movement generation
- **Social robotics** requires specialized mechanisms for interaction, similar to the brain's social cognition areas
- **Affordance-based perception** links perception directly to action possibilities, bridging the gap between seeing and doing

22.7 Further Reading

- Pfeifer, R., & Bongard, J. (2006). How the Body Shapes the Way We Think: A New View of Intelligence. MIT Press. Explores the fundamental role of embodiment in intelligence.
- Wolpert, D. M., & Ghahramani, Z. (2000). Computational principles of movement neuroscience. Nature Neuroscience, 3(11), 1212-1217. Discusses predictive forward models in motor control.
- Brooks, R. A. (1991). Intelligence without representation. Artificial Intelligence, 47(1-3), 139-159. Classic paper on reactive, behavior-based robotics.
- Abbeel, P., & Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. Proceedings of the Twenty-first International Conference on Machine Learning (ICML). Explores learning from human demonstrations.
- Gibson, J. J. (1979). The Ecological Approach to Visual Perception. Psychology Press. Foundational work on affordances.
- Kawato, M. (1999). Internal models for motor control and trajectory planning. Current Opinion in Neurobiology, 9(6), 718-727. Reviews cerebellar models and their applications in robotics.