The Neuroscience of AI

Richard Young

2025

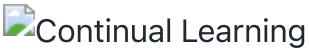
Chapter 23: Lifelong Learning

23.0 Chapter Goals

- Understand the challenge of catastrophic forgetting in artificial neural networks
- Explore biological solutions to the stability-plasticity dilemma
- Learn about computational approaches for continual learning
- Implement mechanisms for memory consolidation and replay

23.1 The Stability-Plasticity Dilemma

One of the major limitations of current artificial neural networks is their inability to learn continuously throughout their lifecycle. While humans and other animals can accumulate knowledge over time, neural networks typically suffer from "catastrophic forgetting" - when trained on new tasks, they tend to rapidly overwrite previously learned information.


Continual Learning

This tension between maintaining stability (preserving existing knowledge) and having plasticity (acquiring new information) is fundamental to any learning system:

- **Too much stability**: The system becomes rigid and unable to learn new information
- **Too much plasticity**: The system constantly overwrites old knowledge with new experiences

23.1.1 Catastrophic Forgetting Problem

When an artificial neural network is trained sequentially on different tasks, learning new tasks can overwrite weights that were critical for previous tasks:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

def demonstrate_catastrophic_forgetting():
    """
    Demonstrate catastrophic forgetting in a simple network
    """
    # Simplified experiment
    model = nn.Sequential(
        nn.Linear(10, 50),
        nn.ReLU(),
        nn.Linear(50, 50),
        nn.ReLU(),
        nn.Linear(50, 2)
    )

    # Generate two synthetic tasks
    task_A_data = torch.randn(1000, 10)
    task_A_targets = (task_A_data[:, 0] > 0).float()

    task_B_data = torch.randn(1000, 10)
    task_B_targets = (task_B_data[:, 1] > 0).float()

    # Training loop
    optimizer = optim.SGD(model.parameters(), lr=0.01)
    criterion = nn.BCEWithLogitsLoss()

    task_A_accuracy = []
    task_B_accuracy = []

    # Initial training on task A
    for epoch in range(100):
        optimizer.zero_grad()
        output = model(task_A_data)[:, 0]
        loss = criterion(output, task_A_targets)
        loss.backward()
        optimizer.step()

        # Evaluate
        with torch.no_grad():
            pred_A = (output > 0).float()
            acc_A = (pred_A == task_A_targets).float().mean()
            task_A_accuracy.append(acc_A.item())

            output_B = model(task_B_data)[:, 1]
            pred_B = (output_B > 0).float()
            acc_B = (pred_B == task_B_targets).float().mean()
            task_B_accuracy.append(acc_B.item())

    # Switch to training on task B
```

```
    for epoch in range(100):
        optimizer.zero_grad()
        output = model(task_B_data)[:, 1]
        loss = criterion(output, task_B_targets)
        loss.backward()
        optimizer.step()

        # Evaluate
        with torch.no_grad():
            pred_A = (model(task_A_data)[:, 0] > 0).float()
            acc_A = (pred_A == task_A_targets).float().mean()
            task_A_accuracy.append(acc_A.item())

            pred_B = (output > 0).float()
            acc_B = (pred_B == task_B_targets).float().mean()
            task_B_accuracy.append(acc_B.item())

    # Plot results
    plt.figure(figsize=(10, 6))
    plt.plot(task_A_accuracy, label='Task A Accuracy')
    plt.plot(task_B_accuracy, label='Task B Accuracy')
    plt.axvline(x=100, color='r', linestyle='--', label='Switch to Task B')
    plt.xlabel('Training Steps')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.title('Catastrophic Forgetting Demonstration')
    plt.ylim(0, 1)

    return plt

# Running this function would show how accuracy on Task A drops rapidly
# when the model is trained on Task B
```

This code demonstrates the classic catastrophic forgetting problem: as the network learns Task B, its performance on Task A deteriorates rapidly. This is because the network parameters needed for Task A are overwritten by those needed for Task B.

23.2 Biological Solutions for Lifelong Learning

The brain employs several mechanisms to balance stability and plasticity, allowing humans and animals to learn continuously throughout life without catastrophic forgetting.

23.2.1 Complementary Learning Systems

The brain manages memory through complementary systems that operate at different timescales:

1. **Hippocampus**: Fast-learning system for episodic memories
    - Rapidly encodes new experiences
    - High plasticity but limited capacity
    - Temporary storage for recent experiences
2. **Neocortex**: Slow-learning system for semantic knowledge
    - Gradually integrates information over time
    - More stable weights for long-term storage
    - Learns general patterns across many experiences

This division allows new information to be quickly stored in the hippocampus while being slowly transferred to the neocortex through a process called “consolidation.”

```
class ComplementaryLearningSystems:
    def __init__(self, fast_learn_rate=0.1, slow_learn_rate=0.01, consolidation_str
        self.hippocampus = []  # Fast-learning episodic memory
        self.neocortex = {}  # Slow-learning semantic memory
        self.fast_learn_rate = fast_learn_rate
        self.slow_learn_rate = slow_learn_rate
        self.consolidation_strength = consolidation_strength

    def learn(self, experience):
        """Learn a new experience"""
        # First, store in hippocampus (fast learning)
        self.hippocampus.append(experience)

        # Then, slowly integrate into neocortex
        if experience["concept"] in self.neocortex:
            # Update existing knowledge
            current = self.neocortex[experience["concept"]]
            self.neocortex[experience["concept"]] = {
                "features": current["features"] * (1 - self.slow_learn_rate) +
                            experience["features"] * self.slow_learn_rate,
                "importance": current["importance"] + self.consolidation_strength
            }
        else:
            # Create new knowledge
            self.neocortex[experience["concept"]] = {
                "features": experience["features"],
                "importance": 1.0
            }

    def consolidate(self, replay_count=5):
        """Consolidate memories from hippocampus to neocortex"""
        # Simulate memory replay during sleep
        for _ in range(replay_count):
            if self.hippocampus:
                # Replay random experiences from hippocampus
                replay_idx = np.random.randint(0, len(self.hippocampus))
                replay_experience = self.hippocampus[replay_idx]

                # Strengthen in neocortex
                concept = replay_experience["concept"]
                if concept in self.neocortex:
                    self.neocortex[concept]["importance"] += self.consolidation_str
```

23.2.2 Synaptic Consolidation Mechanisms

In biological brains, synapses important for existing memories become less plastic over time through several mechanisms:

1. **Structural changes**: Formation of dendritic spines and physical growth of synaptic connections
2. **Protein synthesis**: Creation of proteins that stabilize important synapses
3. **Synaptic tagging**: "Tagging" of important synapses for later consolidation

These mechanisms create a "continuum of plasticity" across the brain's synapses, where some connections remain plastic while others become more stable.

23.2.3 Neuromodulation of Learning Rates

Neuromodulatory systems (using chemicals like dopamine, acetylcholine, and norepinephrine) regulate plasticity based on context:

- **Dopamine**: Signals importance/reward, enhancing learning for valuable experiences
- **Acetylcholine**: Modulates attention and encoding of new information
- **Norepinephrine**: Adjusts overall plasticity based on surprise/novelty

This adaptive regulation helps prioritize what should be learned and retained.

23.3 Computational Approaches to Continual Learning

Inspired by biological mechanisms, several computational approaches have been developed to enable continual learning in artificial systems.

23.3.1 Memory Replay Techniques

One of the most successful approaches involves replaying past experiences to maintain performance on previous tasks:

```
class ExperienceReplayBuffer:
    def __init__(self, capacity=10000):
        """
        Experience replay buffer for continual learning

        Parameters:
        - capacity: Maximum number of experiences to store
        """
        self.buffer = []
        self.capacity = capacity
        self.position = 0

    def add(self, experience):
        """Add an experience to the buffer"""
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = experience
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        """Sample a batch of experiences randomly"""
        indices = np.random.choice(len(self.buffer), batch_size, replace=False)
        return [self.buffer[i] for i in indices]

    def is_empty(self):
        """Check if buffer is empty"""
        return len(self.buffer) == 0

def train_with_replay(model, new_data, replay_buffer, optimizer, criterion, batch_s
    """Train a model using experience replay to prevent forgetting"""
    # Add new experiences to buffer
    for x, y in zip(new_data[0], new_data[1]):
        replay_buffer.add((x, y))

    # Train with a mix of new data and replayed experiences
    for _ in range(5):  # Multiple passes through the data
        # Sample from replay buffer
        if not replay_buffer.is_empty():
            replayed = replay_buffer.sample(batch_size)
            x_replay = torch.stack([x for x, _ in replayed])
            y_replay = torch.stack([y for _, y in replayed])

            # Update model with replayed data
            optimizer.zero_grad()
            outputs = model(x_replay)
            loss = criterion(outputs, y_replay)
            loss.backward()
            optimizer.step()

    return model
```

There are various forms of replay:

1. **Exact Replay**: Store and replay actual data points
2. **Generative Replay**: Train a generative model to produce synthetic examples of past tasks
3. **Feature Replay**: Store and replay high-level features rather than raw inputs

23.3.2 Regularization-Based Methods

Instead of explicitly storing past data, regularization methods constrain weight updates to prevent overwriting important parameters:

```python
def elastic_weight_consolidation(model, new_data, fisher_diag, old_params,
                                 lambda_reg=100, optimizer=None, criterion=None):
    """
    Implement Elastic Weight Consolidation (EWC) for continual learning

    Parameters:
    - model: Neural network model
    - new_data: Data for the new task (x, y)
    - fisher_diag: Diagonal of the Fisher information matrix from previous task
    - old_params: Parameters from after learning the previous task
    - lambda_reg: Regularization strength
    - optimizer: Optimizer for new task
    - criterion: Loss function

    Returns:
    - Updated model
    """
    # If no optimizer is provided, create one
    if optimizer is None:
        optimizer = optim.Adam(model.parameters(), lr=0.001)

    # If no criterion is provided, use MSE
    if criterion is None:
        criterion = nn.MSELoss()

    # Training loop
    for epoch in range(10):
        x, y = new_data

        # Forward pass
        optimizer.zero_grad()
        outputs = model(x)

        # Task loss
        task_loss = criterion(outputs, y)

        # EWC regularization loss
        ewc_loss = 0
        for name, param in model.named_parameters():
            if name in fisher_diag:
                # Calculate weighted squared distance from old parameters
                ewc_loss += (fisher_diag[name] *
                             (param - old_params[name]).pow(2)).sum()

        # Combined loss
        loss = task_loss + lambda_reg * ewc_loss

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

    return model
```

```python
def compute_fisher_diagonal(model, data, criterion):
    """
    Compute diagonal of the Fisher Information Matrix

    Parameters:
    - model: Neural network model
    - data: Data samples (x, y)
    - criterion: Loss function

    Returns:
    - fisher_diag: Dictionary of parameter names to Fisher diagonal values
    """
    fisher_diag = {}

    # Store current parameters
    for name, param in model.named_parameters():
        fisher_diag[name] = torch.zeros_like(param)

    # Compute Fisher diagonal
    model.eval()  # Set to evaluation mode

    for x, y in zip(*data):
        # Forward pass
        model.zero_grad()
        output = model(x.unsqueeze(0))

        # Compute loss
        loss = criterion(output, y.unsqueeze(0))

        # Backward pass to get gradients
        loss.backward()

        # Add squared gradients to Fisher diagonal
        for name, param in model.named_parameters():
            if param.grad is not None:
                fisher_diag[name] += param.grad.pow(2)

    # Normalize by number of samples
    for name in fisher_diag:
        fisher_diag[name] /= len(data[0])

    return fisher_diag
```

Popular regularization methods include:

1. **Elastic Weight Consolidation (EWC)**: Uses Fisher information to estimate parameter importance
2. **Synaptic Intelligence**: Measures the contribution of each parameter during training
3. **Memory Aware Synapses**: Estimates importance based on the sensitivity of output to each weight

23.3.3 Architecture-Based Approaches

Some approaches modify the network architecture itself to accommodate continual learning:

1. **Progressive Neural Networks**: Add new columns for each new task while freezing previous ones
2. **Dynamically Expanding Networks**: Grow the network when needed for new tasks
3. **Context-Dependent Gating**: Route information differently depending on the task

```python
class ProgressiveNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        """
        Progressive Neural Network that adds columns for new tasks

        Parameters:
        - input_size: Dimension of input features
        - hidden_size: Dimension of hidden layer
        - output_size: Dimension of output
        """
        super().__init__()
        self.columns = nn.ModuleList()
        self.adapters = nn.ModuleList()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Add first column
        self.add_column()

    def add_column(self):
        """Add a new column for a new task"""
        column_id = len(self.columns)

        # Create new column
        column = nn.Sequential(
            nn.Linear(self.input_size, self.hidden_size),
            nn.ReLU(),
            nn.Linear(self.hidden_size, self.output_size)
        )

        self.columns.append(column)

        # Create lateral connections from previous columns
        if column_id > 0:
            adapters = nn.ModuleList()
            for prev_col in range(column_id):
                # Create adapter from previous column's hidden layer
                adapter = nn.Linear(self.hidden_size, self.hidden_size, bias=False)
                adapters.append(adapter)
            self.adapters.append(adapters)

    def forward(self, x, task_id):
        """
        Forward pass through the network

        Parameters:
        - x: Input tensor
        - task_id: ID of the task to perform

        Returns:
        - output: Network output for the specified task
        """
```

```
        if task_id >= len(self.columns):
            raise ValueError(f"Task ID {task_id} out of range (only {len(self.columns)} columns)")

        # Get the column for this task
        column = self.columns[task_id]

        # For the first column, just do a standard forward pass
        if task_id == 0:
            return column(x)

        # For later columns, need to include lateral connections
        # Extract features from first layer of current column
        first_layer = column[0]
        h = torch.relu(first_layer(x))

        # Add features from lateral connections
        adapters = self.adapters[task_id - 1]
        for i, adapter in enumerate(adapters):
            prev_h = torch.relu(self.columns[i][0](x))
            h += adapter(prev_h)

        # Apply final layer
        output = column[2](h)

        return output
```

23.3.4 Meta-Learning Approaches

Meta-learning ("learning to learn") approaches aim to discover algorithms or network structures that naturally resist catastrophic forgetting:

```python
class MetaContinualLearner:
    def __init__(self, model, meta_lr=0.001):
        """
        Meta-learning approach for continual learning

        Parameters:
        - model: Base model to train
        - meta_lr: Meta-learning rate
        """
        self.model = model
        self.meta_optimizer = optim.Adam(model.parameters(), lr=meta_lr)
        self.task_optimizers = {}
        self.task_losses = {}

    def learn_task(self, task_id, data, targets, epochs=10, lr=0.01):
        """Learn a specific task"""
        # Create optimizer for this task if it doesn't exist
        if task_id not in self.task_optimizers:
            self.task_optimizers[task_id] = optim.SGD(self.model.parameters(), lr=l
            self.task_losses[task_id] = nn.MSELoss()

        optimizer = self.task_optimizers[task_id]
        criterion = self.task_losses[task_id]

        # Save initial weights
        initial_weights = {name: param.clone() for name, param in self.model.named_

        # First, compute gradients on the current task
        optimizer.zero_grad()
        outputs = self.model(data)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()

        # Meta-update: consider performance on all previous tasks
        self.meta_optimizer.zero_grad()
        meta_loss = 0

        for prev_task_id, prev_optimizer in self.task_optimizers.items():
            if prev_task_id != task_id:
                # Sample data from previous task (simplified)
                prev_data = self.get_task_sample(prev_task_id)
                prev_targets = self.get_task_targets(prev_task_id)

                # Evaluate on previous task
                prev_outputs = self.model(prev_data)
                prev_loss = self.task_losses[prev_task_id](prev_outputs, prev_targe
                meta_loss += prev_loss

        # Include current task loss
        meta_loss += loss

        # Update model using meta-loss
```

```
        meta_loss.backward()
        self.meta_optimizer.step()

    def get_task_sample(self, task_id):
        """Get a sample from a task (placeholder)"""
        # In a real implementation, this would retrieve stored examples
        return torch.randn(10, 10)

    def get_task_targets(self, task_id):
        """Get targets for a task sample (placeholder)"""
        # In a real implementation, this would retrieve stored targets
        return torch.randn(10, 2)
```

23.4 Practical Applications of Lifelong Learning

23.4.1 Robotics and Embodied AI

Continual learning is crucial for robots that need to adapt to new environments and tasks without forgetting previously acquired skills:

- **Skill composition**: Learning new skills while reusing components of existing ones
- **Environmental adaptation**: Adjusting to changes in the physical environment
- **Human interaction**: Learning from ongoing human feedback and demonstration

23.4.2 Personal AI Assistants

Personalized AI systems must learn continuously from interactions with their users:

- **Preference adaptation**: Learning user preferences without forgetting basic functionality
- **Contextual awareness**: Adapting to changing user needs across different contexts
- **Knowledge accumulation**: Building a growing knowledge base about the user and their world

23.4.3 Autonomous Systems

Systems operating in the real world need to learn and adapt throughout their operational lifetime:

- **Self-driving vehicles**: Adapting to new road conditions, traffic patterns, etc.

- **Manufacturing systems**: Learning new production tasks while maintaining existing capabilities
- **Medical diagnosis systems**: Updating diagnostic models with new medical knowledge

23.5 Code Lab: Implementing a Replay-Based Continual Learning System

Let's implement a simple experience replay-based continual learning system and test it on a sequence of tasks:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
from torch.utils.data import TensorDataset, DataLoader

class SimpleNN(nn.Module):
    def __init__(self, input_size=28*28, hidden_size=256, output_size=10):
        super().__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

class ContinualLearner:
    def __init__(self, model, memory_size=200, optimizer=None, criterion=None):
        """
        Continual learning system using experience replay

        Parameters:
        - model: Neural network model
        - memory_size: Maximum size of memory buffer
        - optimizer: Optimizer function (will create Adam if None)
        - criterion: Loss function (will create CrossEntropyLoss if None)
        """
        self.model = model
        self.memory_size = memory_size
        self.memory_x = []
        self.memory_y = []

        # Create optimizer and loss function if not provided
        self.optimizer = optimizer if optimizer else optim.Adam(model.parameters(),
        self.criterion = criterion if criterion else nn.CrossEntropyLoss()

        # Track performance
        self.task_accuracies = {}

    def add_to_memory(self, x, y):
        """Add examples to memory buffer"""
        if isinstance(x, torch.Tensor):
            x = x.detach().cpu()
        if isinstance(y, torch.Tensor):
            y = y.detach().cpu()

        # Convert single examples to lists if needed
        if len(x.shape) == 3:  # Single image [C,H,W]
            x = x.unsqueeze(0)
```

```
            y = torch.tensor([y])

        # Add examples to memory
        for i in range(x.shape[0]):
            if len(self.memory_x) < self.memory_size:
                self.memory_x.append(x[i])
                self.memory_y.append(y[i])
            else:
                # Replace random item
                idx = np.random.randint(0, self.memory_size)
                self.memory_x[idx] = x[i]
                self.memory_y[idx] = y[i]

    def get_memory_batch(self, batch_size):
        """Get a batch of examples from memory"""
        if not self.memory_x:
            return None, None

        # Sample indices
        indices = np.random.choice(len(self.memory_x), min(batch_size, len(self.mem

        # Gather examples
        x = torch.stack([self.memory_x[i] for i in indices])
        y = torch.tensor([self.memory_y[i] for i in indices])

        return x, y

    def train_task(self, task_id, train_loader, epochs=5, memory_batch_size=32, dev
        """
        Train on a new task while using memory replay

        Parameters:
        - task_id: Identifier for the task
        - train_loader: DataLoader for training data
        - epochs: Number of training epochs
        - memory_batch_size: Batch size for memory replay
        - device: Device to use for training
        """
        self.model.to(device)
        self.model.train()

        for epoch in range(epochs):
            total_loss = 0
            correct = 0
            total = 0

            for batch_idx, (data, target) in enumerate(train_loader):
                data, target = data.to(device), target.to(device)

                # Regular update on current task data
                self.optimizer.zero_grad()
                output = self.model(data)
                loss = self.criterion(output, target)
                loss.backward()
```

```
                self.optimizer.step()

                # Update using memory replay
                if self.memory_x:
                    memory_x, memory_y = self.get_memory_batch(memory_batch_size)
                    if memory_x is not None:
                        memory_x, memory_y = memory_x.to(device), memory_y.to(devic

                        self.optimizer.zero_grad()
                        output = self.model(memory_x)
                        replay_loss = self.criterion(output, memory_y)
                        replay_loss.backward()
                        self.optimizer.step()

                # Calculate accuracy
                _, predicted = output.max(1)
                total += target.size(0)
                correct += predicted.eq(target).sum().item()

                # Store examples in memory
                self.add_to_memory(data, target)

            # Print epoch statistics
            print(f'Task {task_id}, Epoch {epoch+1}/{epochs}: Accuracy: {100.*corre

    def evaluate_task(self, task_id, test_loader, device='cpu'):
        """Evaluate performance on a specific task"""
        self.model.to(device)
        self.model.eval()

        correct = 0
        total = 0

        with torch.no_grad():
            for data, target in test_loader:
                data, target = data.to(device), target.to(device)
                output = self.model(data)

                _, predicted = output.max(1)
                total += target.size(0)
                correct += predicted.eq(target).sum().item()

        accuracy = 100. * correct / total
        self.task_accuracies[task_id] = accuracy

        print(f'Task {task_id} Accuracy: {accuracy:.2f}%')
        return accuracy

    def evaluate_all_tasks(self, task_loaders, device='cpu'):
        """Evaluate performance on all tasks"""
        results = {}
        for task_id, loader in task_loaders.items():
            accuracy = self.evaluate_task(task_id, loader, device)
            results[task_id] = accuracy
```

```
        return results

# Example usage (with placeholder data)
def create_rotated_mnist_tasks(n_tasks=5, batch_size=128):
    """Create a series of tasks based on rotated MNIST digits"""
    # This is a placeholder - in a real implementation, you would load MNIST
    # and create rotated versions

    task_loaders = {}

    for task_id in range(n_tasks):
        # Simulate different rotations of MNIST
        angle = task_id * 15  # Rotate by 0°, 15°, 30°, etc.

        # Placeholder data - in real code, you would load and rotate MNIST
        # Here we just create random tensors of the right shape
        train_x = torch.randn(1000, 1, 28, 28)
        train_y = torch.randint(0, 10, (1000,))
        test_x = torch.randn(200, 1, 28, 28)
        test_y = torch.randint(0, 10, (200,))

        # Create data loaders
        train_dataset = TensorDataset(train_x, train_y)
        test_dataset = TensorDataset(test_x, test_y)

        train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=Tru
        test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False

        task_loaders[task_id] = {
            'train': train_loader,
            'test': test_loader
        }

    return task_loaders

def run_continual_learning_experiment():
    """Run a complete continual learning experiment"""
    # Create model
    model = SimpleNN()

    # Create learning systems
    replay_learner = ContinualLearner(model, memory_size=200)
    vanilla_learner = ContinualLearner(model, memory_size=0)  # No replay

    # Create tasks
    task_loaders = create_rotated_mnist_tasks(n_tasks=5)

    # Training and evaluation
    for task_id in range(5):
        print(f"\nTraining on task {task_id}")
        train_loader = task_loaders[task_id]['train']

        # Train on the task
        replay_learner.train_task(task_id, train_loader)
```

```
        # Evaluate on all previous tasks
        test_loaders = {i: task_loaders[i]['test'] for i in range(task_id+1)}
        results = replay_learner.evaluate_all_tasks(test_loaders)

        # Display current results
        print(f"Task {task_id} complete. Current accuracies:")
        for t, acc in results.items():
            print(f"  Task {t}: {acc:.2f}%")

    # Plot final results
    plt.figure(figsize=(10, 6))
    tasks = list(replay_learner.task_accuracies.keys())
    accs = list(replay_learner.task_accuracies.values())

    plt.bar(tasks, accs)
    plt.xlabel('Task ID')
    plt.ylabel('Accuracy (%)')
    plt.title('Final Performance Across Tasks')
    plt.ylim(0, 100)
    plt.xticks(tasks)
    plt.grid(axis='y', linestyle='--', alpha=0.7)

    return plt

# Calling this function would run the complete experiment
# run_continual_learning_experiment()
```

This implementation demonstrates a simple replay-based approach to continual learning. The system maintains a buffer of past experiences and regularly replays them alongside new data to prevent catastrophic forgetting.

23.6 Take-aways

- **Catastrophic forgetting is a major challenge** for neural networks, preventing them from learning continuously like humans do
- **Biological brains solve the stability-plasticity dilemma** through complementary learning systems, synaptic consolidation, and neuromodulation
- **Computational approaches to lifelong learning include**:
  - Memory replay methods that revisit past experiences
  - Regularization techniques that constrain important weights
  - Architectural approaches that allocate new capacity for new tasks
  - Meta-learning strategies that learn how to avoid forgetting

- **Practical applications of lifelong learning** are numerous in robotics, personal AI assistants, and autonomous systems
- **Combining multiple approaches** often yields the best performance for real-world continual learning systems

23.7 Further Reading

- Kirkpatrick, J., et al. (2017). Overcoming catastrophic forgetting in neural networks. Proceedings of the National Academy of Sciences, 114(13), 3521-3526.
- McClelland, J. L., McNaughton, B. L., & O'Reilly, R. C. (1995). Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory. Psychological Review, 102(3), 419-457.
- Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., & Wermter, S. (2019). Continual lifelong learning with neural networks: A review. Neural Networks, 113, 54-71.
- Zenke, F., Poole, B., & Ganguli, S. (2017). Continual learning through synaptic intelligence. In Proceedings of the 34th International Conference on Machine Learning, 3987-3995.
- Lopez-Paz, D., & Ranzato, M. (2017). Gradient episodic memory for continual learning. In Advances in Neural Information Processing Systems, 6467-6476.
- van de Ven, G. M., & Tolias, A. S. (2019). Three scenarios for continual learning. arXiv preprint arXiv:1904.07734.