Project Brief: Unified Mamba-Hopfield-DEQ Architecture (Option 3)

Goal: Create a theoretically unified architecture where the DEQ fixed-point computation *is* the Hopfield energy minimization process. Memory retrieval, state evolution, and reasoning converge to a single equilibrium that simultaneously satisfies both the DEQ dynamics and the Hopfield energy minimum.

Core Insight: Rather than treating MHN as a separate memory module, we embed Hopfield dynamics directly into the DEQ iteration structure. Each DEQ step performs both state updates (Mamba-driven) and memory operations (Hopfield-driven) until the system reaches a joint equilibrium.

Target: Research-grade implementation with rigorous mathematical foundations, energy-based training objectives, and experiments validating the unified convergence properties.

Step-by-Step Implementation Plan

Phase 1: Theoretical Foundation & Repository Structure

Repository Structure:

```
unified-mamba-hopfield-deq/
├── src/
│   ├── core/
│   │   ├── mamba.py            # Mamba SSM layers
│   │   ├── energy.py           # Energy function definitions
│   │   ├── dynamics.py         # Unified dynamics (f that is both DEQ and Hopfield)
│   │   └── equilibrium.py      # Joint equilibrium solver
│   ├── models/
│   │   ├── unified.py          # Full unified architecture
│   │   ├── components.py       # Modular building blocks
│   │   └── pretrained.py       # Interface for loading base models
│   ├── solvers/
│   │   ├── fixed_point.py      # DEQ fixed-point solvers
│   │   ├── energy_min.py       # Energy minimization solvers
│   │   └── hybrid_solver.py    # Unified solver (both objectives)
│   ├── training/
│   │   ├── objectives.py       # Energy-based loss functions
│   │   ├── trainer.py          # Training loop
│   │   └── schedules.py        # Curriculum and annealing
```

```
│   ├── analysis/
│   │   ├── convergence.py        # Convergence analysis tools
│   │   ├── energy_landscape.py   # Energy visualization
│   │   └── stability.py          # Lyapunov analysis
│   └── utils/
│       ├── math_utils.py         # Linear algebra helpers
│       └── visualization.py      # Plotting utilities
├── experiments/
│   ├── theory/
│   │   ├── convergence_proofs.py # Empirical validation of theory
│   │   └── energy_analysis.py    # Energy landscape studies
│   ├── tasks/
│   │   ├── memory_tasks.py       # Memory-intensive benchmarks
│   │   ├── reasoning_tasks.py    # Multi-step reasoning
│   │   └── streaming_tasks.py    # Long-sequence processing
│   └── configs/
├── notebooks/
│   ├── theory_walkthrough.ipynb  # Mathematical foundations
│   ├── energy_visualization.ipynb
│   └── convergence_analysis.ipynb
├── tests/
│   ├── test_energy.py
│   ├── test_convergence.py
│   └── test_gradients.py
├── docs/
│   ├── theory.md                 # Mathematical documentation
│   ├── architecture.md           # Design decisions
│   └── api.md                    # Code reference
├── requirements.txt
├── setup.py
└── README.md
```

Dependencies (requirements.txt):

```
torch>=2.0.0
einops
scipy  # For advanced solvers
jax[cpu]  # Optional: for Hessian computations
mamba-ssm
wandb
pytest
jupyter
```

```
matplotlib
seaborn
networkx  # For visualizing convergence graphs
plotly  # Interactive energy landscapes
sympy  # Symbolic math for theoretical analysis
```

Phase 2: Mathematical Framework & Energy Functions

Step 2.1: Unified Energy Function (src/core/energy.py)

Define a single energy function that encompasses both Hopfield memory retrieval and DEQ equilibrium:

Theoretical foundation:

```
E(z, x, M) = E_hopfield(z, M) + E_consistency(z, x) + E_regularization(z)

where:
- E_hopfield(z, M) = -log(Σᵢ exp(β⟨z, mᵢ⟩)) // Hopfield energy over memory patterns M
- E_consistency(z, x) = ||z - f_mamba(z, x)||² // DEQ fixed-point residual
- E_regularization(z) = λ₁||z||² + λ₂||∇E||² // Smoothness and boundedness
```

Key insight: By minimizing this joint energy, we simultaneously:

1. Retrieve relevant memory patterns (Hopfield objective)
1. Find a fixed point consistent with Mamba dynamics (DEQ objective)
1. Ensure stable, bounded solutions (regularization)

Implementation:

```python
class UnifiedEnergyFunction(nn.Module):
    """
    Energy function that unifies Hopfield retrieval and DEQ equilibrium.

    The system naturally minimizes this energy, and the minimum corresponds
    to both a fixed point and an optimal memory retrieval state.
    """
```

```
def __init__(self,
             d_model,
             beta=1.0,           # Hopfield inverse temperature
             alpha=1.0,          # DEQ consistency weight
             lambda_l2=0.01,     # L2 regularization
             lambda_smooth=0.001 # Smoothness penalty
            ):
    super().__init__()
    self.d_model = d_model
    self.beta = beta
    self.alpha = alpha
    self.lambda_l2 = lambda_l2
    self.lambda_smooth = lambda_smooth

def hopfield_energy(self, z, memory_patterns):
    """
    Modern Hopfield energy: E = -log(Σᵢ exp(β⟨z, mᵢ⟩))

    Args:
        z: Current state (B, D)
        memory_patterns: Stored patterns (M, D)
    Returns:
        Energy scalar (minimized when z aligns with stored patterns)
    """
    # Compute similarities to all patterns
    similarities = self.beta * torch.matmul(z, memory_patterns.T)  # (B, M)

    # Log-sum-exp for numerical stability
    energy = -torch.logsumexp(similarities, dim=-1)  # (B,)

    return energy.mean()

def consistency_energy(self, z, z_next):
    """
    DEQ fixed-point residual: E = ||z - f(z)||²

    Measures how far z is from being a fixed point.
    Minimum (0) achieved when z = f(z).
    """
    residual = z - z_next
    energy = torch.sum(residual ** 2, dim=-1)  # (B,)
    return energy.mean()

def regularization_energy(self, z, grad_z=None):
```

```
        """
        Regularization terms for stable solutions.
        """
        # L2 norm penalty (prevents unbounded states)
        l2_energy = self.lambda_l2 * torch.sum(z ** 2, dim=-1).mean()

        # Smoothness penalty (encourages smooth energy landscapes)
        smooth_energy = 0.0
        if grad_z is not None:
            smooth_energy = self.lambda_smooth * torch.sum(grad_z ** 2, dim=-1).mean()

        return l2_energy + smooth_energy

    def forward(self, z, z_next, memory_patterns, compute_grad=False):
        """
        Compute total unified energy.

        Args:
            z: Current state
            z_next: Next state f(z) from dynamics
            memory_patterns: Memory bank
            compute_grad: Whether to compute gradient energy

        Returns:
            Total energy, dict of components
        """
        # Hopfield component
        E_hop = self.hopfield_energy(z, memory_patterns)

        # DEQ consistency component
        E_cons = self.alpha * self.consistency_energy(z, z_next)

        # Regularization
        grad_z = None
        if compute_grad and z.requires_grad:
            grad_z = torch.autograd.grad(
                E_hop, z,
                create_graph=True,
                retain_graph=True
            )[0]
        E_reg = self.regularization_energy(z, grad_z)

        # Total energy
        E_total = E_hop + E_cons + E_reg
```

```
        components = {
            'hopfield': E_hop.item(),
            'consistency': E_cons.item(),
            'regularization': E_reg.item(),
            'total': E_total.item()
        }

        return E_total, components

    def energy_gradient(self, z, memory_patterns):
        """
        Compute ∇E for gradient-based minimization.
        Used in solver to find energy minimum.
        """
        z_with_grad = z.detach().requires_grad_(True)
        energy, _ = self.forward(z_with_grad, z_with_grad, memory_patterns)
        grad = torch.autograd.grad(energy, z_with_grad)[0]
        return grad
```

Step 2.2: Unified Dynamics (src/core/dynamics.py)

Define the dynamics function that drives both Mamba state evolution and Hopfield updates:

Theoretical design:

```
z_{t+1} = f(z_t, x, M) = Mamba_update(z_t, x) + Hopfield_update(z_t, M)

where:
- Mamba_update: State space evolution based on input sequence
- Hopfield_update: Memory-driven correction toward stored patterns
- The combination naturally minimizes the unified energy
```

Implementation:

```python
class UnifiedDynamics(nn.Module):
    """
    Dynamics function f(z) that implements both:
```

1. Mamba state space updates (temporal processing)
2. Hopfield memory updates (pattern retrieval)

Fixed points of this dynamics are equilibria that satisfy both objectives.
"""

```
def __init__(self,
             d_model,
             d_state,
             d_conv,
             num_heads=8,
             beta=1.0,
             gate_type='sigmoid'  # How to blend Mamba and Hopfield
            ):
    super().__init__()

    # Mamba component for temporal processing
    self.mamba = MambaLayer(d_model, d_state, d_conv)

    # Projection for memory queries
    self.query_proj = nn.Linear(d_model, d_model)

    # Gating mechanism to blend Mamba and Hopfield updates
    self.gate = nn.Sequential(
        nn.Linear(d_model * 2, d_model),
        nn.Sigmoid() if gate_type == 'sigmoid' else nn.Tanh()
    )

    # Output projection
    self.out_proj = nn.Linear(d_model, d_model)

    self.beta = beta
    self.d_model = d_model

def mamba_update(self, z, x_context):
    """
    Mamba-based state evolution.

    Args:
        z: Current state (B, D)
        x_context: Input context from earlier layers (B, L, D)

    Returns:
        Updated state from SSM dynamics
```

```
        """
        # Treat z as a query into the context
        # Use Mamba's selective mechanism to process
        return self.mamba(x_context, state=z)

    def hopfield_update(self, z, memory_patterns):
        """
        Hopfield-based memory retrieval and state update.

        Uses modern Hopfield update rule:
        z_new = Σᵢ softmax(β⟨z, mᵢ⟩) * mᵢ

        Args:
            z: Current state (B, D)
            memory_patterns: Stored memory (M, D)

        Returns:
            Memory-retrieved state
        """
        # Project to query space
        query = self.query_proj(z)  # (B, D)

        # Compute attention over memory patterns (Hopfield retrieval)
        similarities = self.beta * torch.matmul(
            query, memory_patterns.T
        )  # (B, M)

        attention_weights = F.softmax(similarities, dim=-1)  # (B, M)

        # Retrieve weighted combination of patterns
        retrieved = torch.matmul(attention_weights, memory_patterns)  # (B, D)

        return retrieved

    def forward(self, z, x_context, memory_patterns, return_components=False):
        """
        Unified dynamics: z_{t+1} = f(z_t, x, M)

        Combines Mamba temporal processing with Hopfield memory retrieval.

        Args:
            z: Current state (B, D)
            x_context: Input sequence context (B, L, D)
            memory_patterns: Memory bank (M, D)
```

```
        return_components: Whether to return intermediate values

    Returns:
        z_next: Updated state
        (optional) components: Dict with Mamba and Hopfield contributions
    """
    # Compute both updates
    mamba_update = self.mamba_update(z, x_context)      # (B, D)
    hopfield_update = self.hopfield_update(z, memory_patterns)  # (B, D)

    # Adaptive gating: learn to blend based on current state
    gate_input = torch.cat([mamba_update, hopfield_update], dim=-1)
    gate_values = self.gate(gate_input)  # (B, D)

    # Blend: z_next = gate * mamba + (1-gate) * hopfield
    z_next = gate_values * mamba_update + (1 - gate_values) * hopfield_update

    # Final projection (with residual connection)
    z_next = z + self.out_proj(z_next)

    if return_components:
        components = {
            'mamba': mamba_update,
            'hopfield': hopfield_update,
            'gate': gate_values,
            'blended': z_next
        }
        return z_next, components

    return z_next

def is_contraction(self, z, x_context, memory_patterns, epsilon=1e-2):
    """
    Check if dynamics are contractive (necessary for DEQ convergence).

    Computes ||f(z + δ) - f(z)|| / ||δ|| and checks if < 1.
    """
    with torch.no_grad():
        # Perturb z slightly
        delta = epsilon * torch.randn_like(z)
        z_perturbed = z + delta

        # Compute dynamics at both points
        f_z = self.forward(z, x_context, memory_patterns)
```

```
            f_z_pert = self.forward(z_perturbed, x_context, memory_patterns)

            # Lipschitz constant estimate
            numerator = torch.norm(f_z_pert - f_z, dim=-1)
            denominator = torch.norm(delta, dim=-1)
            lipschitz = (numerator / denominator).mean()

        return lipschitz.item() < 1.0, lipschitz.item()
```

Phase 3: Unified Equilibrium Solver

Step 3.1: Hybrid Solver (src/solvers/hybrid_solver.py)

Implement a solver that finds equilibria satisfying BOTH the fixed-point condition AND energy minimization:

Convergence criteria:

```
Stop when:
1. ||z_t - f(z_t)|| < ε_fixedpoint  (DEQ criterion)
AND
2. ||∇E(z_t)|| < ε_energy           (Hopfield criterion)
AND
3. E(z_t) has stabilized            (Energy plateau)
```

Implementation:

```python
class UnifiedEquilibriumSolver:
    """
    Solver that finds joint equilibria where:
    1. z* = f(z*, x, M)  [fixed-point condition]
    2. ∇E(z*) = 0        [energy minimum condition]

    Uses hybrid optimization combining:
    - Fixed-point iteration (Anderson acceleration)
    - Gradient descent on energy
    - Alternating optimization
    """
```

```
def __init__(self,
             dynamics_fn,
             energy_fn,
             max_iter=50,
             tol_fixedpoint=1e-3,
             tol_energy=1e-3,
             solver_type='alternating',  # 'alternating', 'simultaneous', 'cascade'
             anderson_memory=5,
             learning_rate=0.01
            ):
    self.dynamics = dynamics_fn
    self.energy = energy_fn
    self.max_iter = max_iter
    self.tol_fp = tol_fixedpoint
    self.tol_energy = tol_energy
    self.solver_type = solver_type
    self.m = anderson_memory  # Anderson acceleration memory
    self.lr = learning_rate

def fixed_point_step(self, z, x_context, memory_patterns, history=None):
    """
    One step of Anderson-accelerated fixed-point iteration.

    Standard fixed-point: z_new = f(z)
    Anderson: z_new = f(z) + correction based on history
    """
    # Compute f(z)
    f_z = self.dynamics(z, x_context, memory_patterns)

    if history is None or len(history) < 2:
        # No acceleration yet
        return f_z

    # Anderson acceleration
    # Collect residuals: r_i = f(z_i) - z_i
    residuals = [f_zi - z_i for z_i, f_zi in history]

    # Build matrices for least-squares problem
    # (simplified Anderson, full version is more complex)
    recent_residuals = residuals[-self.m:]
    R = torch.stack(recent_residuals, dim=0)  # (m, B, D)

    # Solve least-squares for mixing coefficients
```

```
    # This is the core of Anderson acceleration
    alpha = self._solve_anderson_coefficients(R)

    # Mix previous iterates
    z_accelerated = sum(
        a * f_z for a, (_, f_z) in zip(alpha, history[-self.m:])
    )

    return z_accelerated

def energy_descent_step(self, z, memory_patterns):
    """
    One step of gradient descent on energy.

    z_new = z - η∇E(z)
    """
    grad_E = self.energy.energy_gradient(z, memory_patterns)
    z_new = z - self.lr * grad_E
    return z_new

def alternating_solve(self, z_init, x_context, memory_patterns):
    """
    Alternating optimization:
    - Even iterations: Fixed-point step
    - Odd iterations: Energy descent step

    Provably converges if both objectives have unique minima in overlap region.
    """
    z = z_init
    history = []
    energy_history = []

    for iter_num in range(self.max_iter):
        # Alternate between objectives
        if iter_num % 2 == 0:
            # Fixed-point iteration
            z_next = self.fixed_point_step(z, x_context, memory_patterns, history)
        else:
            # Energy minimization
            z_next = self.energy_descent_step(z, memory_patterns)

        # Compute convergence metrics
        fp_residual = torch.norm(z_next - self.dynamics(z, x_context, memory_patterns))
        E_current, E_components = self.energy(z, z_next, memory_patterns)
```

```
            energy_grad = torch.norm(self.energy.energy_gradient(z, memory_patterns))

            # Store history
            history.append((z.detach(), z_next.detach()))
            energy_history.append(E_current.item())

            # Check convergence
            converged_fp = fp_residual < self.tol_fp
            converged_energy = energy_grad < self.tol_energy
            energy_stable = (
                len(energy_history) > 5 and
                np.std(energy_history[-5:]) < 1e-4
            )

            if converged_fp and converged_energy and energy_stable:
                return z_next, {
                    'converged': True,
                    'iterations': iter_num + 1,
                    'final_fp_residual': fp_residual.item(),
                    'final_energy_grad': energy_grad.item(),
                    'final_energy': E_current.item(),
                    'energy_components': E_components,
                    'energy_history': energy_history
                }

            z = z_next

        # Max iterations reached
        return z, {
            'converged': False,
            'iterations': self.max_iter,
            'final_fp_residual': fp_residual.item(),
            'final_energy_grad': energy_grad.item(),
            'final_energy': E_current.item(),
            'energy_components': E_components,
            'energy_history': energy_history
        }

    def simultaneous_solve(self, z_init, x_context, memory_patterns):
        """
        Simultaneous optimization of both objectives.

        Minimizes: λ₁||z - f(z)||² + λ₂E(z)
```

```
Uses combined gradient: ∇[combined objective]
"""
z = z_init.clone().requires_grad_(True)
optimizer = torch.optim.Adam([z], lr=self.lr)

history = []

for iter_num in range(self.max_iter):
    optimizer.zero_grad()

    # Compute both objectives
    z_next = self.dynamics(z, x_context, memory_patterns)
    fp_loss = torch.sum((z - z_next) ** 2)

    E_current, E_components = self.energy(z, z_next, memory_patterns)

    # Combined loss
    total_loss = fp_loss + E_current
    total_loss.backward()

    optimizer.step()

    # Check convergence
    with torch.no_grad():
        fp_residual = torch.norm(z - z_next)
        energy_grad = torch.norm(self.energy.energy_gradient(z, memory_patterns))

        if fp_residual < self.tol_fp and energy_grad < self.tol_energy:
            return z.detach(), {
                'converged': True,
                'iterations': iter_num + 1,
                'final_fp_residual': fp_residual.item(),
                'final_energy_grad': energy_grad.item(),
                'final_energy': E_current.item(),
                'energy_components': E_components
            }

    history.append(E_current.item())

return z.detach(), {
    'converged': False,
    'iterations': self.max_iter,
    'energy_history': history
}
```

```
def cascade_solve(self, z_init, x_context, memory_patterns):
    """
    Cascade approach:
    1. First converge to fixed point (fast, ignoring energy)
    2. Then refine via energy minimization (slow, high quality)

    Good for when fixed-point is close to energy minimum.
    """
    # Phase 1: Fast fixed-point convergence
    z = z_init
    for _ in range(self.max_iter // 2):
        z = self.dynamics(z, x_context, memory_patterns)
        if torch.norm(z - self.dynamics(z, x_context, memory_patterns)) < self.tol_fp:
            break

    # Phase 2: Energy refinement
    z = z.requires_grad_(True)
    optimizer = torch.optim.LBFGS([z], lr=0.1, max_iter=20)

    def closure():
        optimizer.zero_grad()
        z_next = self.dynamics(z, x_context, memory_patterns)
        E, _ = self.energy(z, z_next, memory_patterns)
        E.backward()
        return E

    optimizer.step(closure)

    # Final evaluation
    with torch.no_grad():
        z_next = self.dynamics(z, x_context, memory_patterns)
        E_final, E_components = self.energy(z, z_next, memory_patterns)
        fp_residual = torch.norm(z - z_next)
        energy_grad = torch.norm(self.energy.energy_gradient(z, memory_patterns))

    return z.detach(), {
        'converged': True,
        'final_fp_residual': fp_residual.item(),
        'final_energy_grad': energy_grad.item(),
        'final_energy': E_final.item(),
        'energy_components': E_components
    }
```

```
    def solve(self, z_init, x_context, memory_patterns):
        """
        Main entry point. Dispatches to appropriate solver.
        """
        if self.solver_type == 'alternating':
            return self.alternating_solve(z_init, x_context, memory_patterns)
        elif self.solver_type == 'simultaneous':
            return self.simultaneous_solve(z_init, x_context, memory_patterns)
        elif self.solver_type == 'cascade':
            return self.cascade_solve(z_init, x_context, memory_patterns)
        else:
            raise ValueError(f"Unknown solver type: {self.solver_type}")

    def _solve_anderson_coefficients(self, R):
        """
        Solve for Anderson mixing coefficients.
        Simplified version - full Anderson is more involved.
        """
        # R: (m, B, D) residuals
        m = R.shape[0]

        # Gram matrix: G_ij = <r_i, r_j>
        R_flat = R.view(m, -1)  # (m, B*D)
        G = torch.matmul(R_flat, R_flat.T)  # (m, m)

        # Solve G α = 1 with constraint Σα = 1
        # (Simplified - should use constrained optimization)
        ones = torch.ones(m, device=R.device)
        alpha = torch.linalg.solve(G + 1e-6 * torch.eye(m, device=G.device), ones)
        alpha = alpha / alpha.sum()  # Normalize

        return alpha
```

Phase 4: Full Unified Model

Step 4.1: Complete Architecture (src/models/unified.py)

```python
class UnifiedMambaHopfieldDEQ(nn.Module):
    """
    Fully unified architecture where:
```

- Mamba handles sequential processing
- Hopfield provides memory via energy minimization
- DEQ finds equilibria that satisfy both constraints

The model operates by:
1. Processing input with Mamba layers to get context
2. Initializing equilibrium state z_0
3. Iterating unified dynamics until convergence:
   - $z^* = f(z^*, x, M)$ [fixed point]
   - $\nabla E(z^*) = 0$ [energy minimum]
4. Decoding z^* to output
"""

```
def __init__(self,
             vocab_size,
             d_model=512,
             d_state=64,
             d_conv=4,
             n_layers=6,
             memory_size=10000,
             beta=2.0,
             solver_type='alternating',
             max_iterations=30,
             tol=1e-3
            ):
    super().__init__()

    self.d_model = d_model
    self.memory_size = memory_size

    # Input embedding
    self.embedding = nn.Embedding(vocab_size, d_model)

    # Mamba layers for initial processing
    self.mamba_layers = nn.ModuleList([
        MambaLayer(d_model, d_state, d_conv)
        for _ in range(n_layers)
    ])

    # Unified dynamics (combines Mamba and Hopfield)
    self.dynamics = UnifiedDynamics(
        d_model=d_model,
        d_state=d_state,
        d_conv=d_conv,
```

```
            beta=beta
        )

        # Energy function
        self.energy_fn = UnifiedEnergyFunction(
            d_model=d_model,
            beta=beta
        )

        # Equilibrium solver
        self.solver = UnifiedEquilibriumSolver(
            dynamics_fn=self.dynamics,
            energy_fn=self.energy_fn,
            max_iter=max_iterations,
            tol_fixedpoint=tol,
            tol_energy=tol,
            solver_type=solver_type
        )

        # Memory bank (learnable initialization)
        self.register_buffer(
            'memory_patterns',
            torch.randn(memory_size, d_model) / np.sqrt(d_model)
        )
        self.memory_write_gate = nn.Linear(d_model, 1)

        # Output projection
        self.output_proj = nn.Sequential(
            nn.LayerNorm(d_model),
            nn.Linear(d_model, vocab_size)
        )

        # For tracking convergence during training
        self.convergence_stats = []

    def process_context(self, input_ids):
        """
        Pass input through Mamba layers to get context representation.

        Args:
            input_ids: (B, L) token indices

        Returns:
            context: (B, L, D) processed context
```

```
            final_state: (B, D) state for equilibrium initialization
        """
        # Embed tokens
        x = self.embedding(input_ids)  # (B, L, D)

        # Process through Mamba layers
        for layer in self.mamba_layers:
            x = layer(x)

        # Extract final state (use last token or pooling)
        final_state = x[:, -1, :]  # (B, D) - using last position

        return x, final_state

    def update_memory(self, z_equilibrium, should_store=None):
        """
        Dynamically update memory patterns based on converged states.

        Args:
            z_equilibrium: (B, D) equilibrium states
            should_store: (B,) boolean mask for which samples to store

        Uses a queue-based approach: oldest patterns are replaced.
        """
        if should_store is None:
            # Decide based on novelty (distance to existing patterns)
            with torch.no_grad():
                similarities = torch.matmul(
                    z_equilibrium, self.memory_patterns.T
                )  # (B, M)
                max_similarity = similarities.max(dim=-1)[0]  # (B,)

                # Store if sufficiently novel (low similarity)
                should_store = max_similarity < 0.7

        # Store selected patterns
        num_to_store = should_store.sum().item()
        if num_to_store > 0:
            patterns_to_store = z_
```
equilibrium[should_store] # (N, D)

```
        # Queue-based storage: replace oldest entries
```

```
        # Shift memory and add new patterns at the end
        num_existing = self.memory_patterns.shape[0]
        if num_to_store < num_existing:
            self.memory_patterns = torch.cat([
                self.memory_patterns[num_to_store:],
                patterns_to_store
            ], dim=0)
        else:
            # Replace all memory if too many new patterns
            self.memory_patterns = patterns_to_store[:num_existing]

    def forward(self, input_ids, update_memory=True, return_diagnostics=False):
        """
        Full forward pass through unified architecture.

        Process:
        1. Embed and process input with Mamba
        2. Initialize equilibrium state z₀
        3. Solve for equilibrium: find z* where z* = f(z*) and ∇E(z*) = 0
        4. Optionally update memory with z*
        5. Decode to output logits

        Args:
            input_ids: (B, L) input token sequences
            update_memory: Whether to add z* to memory
            return_diagnostics: Return convergence info

        Returns:
            logits: (B, L, vocab_size) output predictions
            diagnostics: (optional) convergence and energy info
        """
        batch_size, seq_len = input_ids.shape

        # Phase 1: Context processing
        context, z_init = self.process_context(input_ids)  # (B, L, D), (B, D)

        # Phase 2: Equilibrium finding
        z_equilibrium, solver_info = self.solver.solve(
            z_init=z_init,
            x_context=context,
            memory_patterns=self.memory_patterns
        )

        # Track convergence statistics
```

```
        if self.training:
            self.convergence_stats.append({
                'converged': solver_info['converged'],
                'iterations': solver_info['iterations'],
                'final_energy': solver_info['final_energy'],
                'energy_components': solver_info['energy_components']
            })

        # Phase 3: Memory update (if training or explicitly requested)
        if update_memory and (self.training or return_diagnostics):
            self.update_memory(z_equilibrium)

        # Phase 4: Decode to output
        # Expand equilibrium state to sequence length
        z_expanded = z_equilibrium.unsqueeze(1).expand(-1, seq_len, -1)  # (B, L, D)

        # Combine with context (residual connection)
        combined = context + z_expanded

        # Project to vocabulary
        logits = self.output_proj(combined)  # (B, L, vocab_size)

        if return_diagnostics:
            diagnostics = {
                'solver_info': solver_info,
                'z_equilibrium': z_equilibrium,
                'z_init': z_init,
                'energy_trajectory': solver_info.get('energy_history', []),
                'memory_usage': self._compute_memory_usage(z_equilibrium)
            }
            return logits, diagnostics

        return logits

    def _compute_memory_usage(self, z_equilibrium):
        """
        Analyze how memory patterns are being used.

        Returns statistics about memory retrieval patterns.
        """
        with torch.no_grad():
            # Compute attention over memory
            similarities = torch.matmul(
                z_equilibrium, self.memory_patterns.T
```

```
        )  # (B, M)
        attention = F.softmax(similarities, dim=-1)  # (B, M)

        # Compute entropy (high = distributed, low = focused)
        entropy = -torch.sum(
            attention * torch.log(attention + 1e-10), dim=-1
        ).mean()

        # Top-k usage (concentration)
        top_k = 10
        top_k_mass = attention.topk(top_k, dim=-1)[0].sum(dim=-1).mean()

        return {
            'attention_entropy': entropy.item(),
            'top_10_mass': top_k_mass.item(),
            'num_patterns': self.memory_patterns.shape[0]
        }

    def get_implicit_gradients(self, z_equilibrium, loss):
        """
        Compute gradients using implicit differentiation.

        For DEQ models, we don't backprop through iterations.
        Instead, we use the implicit function theorem:

        dL/dθ = (I - ∂f/∂z*)^(-1) ∂f/∂θ

        where z* is the equilibrium.
        """
        # This is handled automatically by the solver's backward pass
        # But we can provide explicit implementation for clarity

        with torch.enable_grad():
            z_eq = z_equilibrium.detach().requires_grad_(True)

            # Compute Jacobian-vector product
            def jvp(v):
                """Compute (∂f/∂z) @ v"""
                # Forward pass to get f(z)
                z_next = self.dynamics(
                    z_eq,
                    None,  # Would need to cache context
                    self.memory_patterns
                )
```

```
            # Compute vjp
            return torch.autograd.grad(
                z_next, z_eq, v,
                retain_graph=True, create_graph=True
            )[0]

        # Solve (I - J) @ g = ∂L/∂z using conjugate gradient
        g_loss = torch.autograd.grad(loss, z_eq)[0]

        # Use iterative solver for (I - J)^(-1) @ g_loss
        implicit_grad = self._solve_implicit_gradient(jvp, g_loss)

    return implicit_grad

def _solve_implicit_gradient(self, jvp_fn, g, max_iter=10, tol=1e-3):
    """
    Solve (I - J) @ x = g using conjugate gradient.

    Avoids forming the full Jacobian matrix.
    """
    x = g.clone()
    r = g - (x - jvp_fn(x))
    p = r.clone()

    for _ in range(max_iter):
        Ap = p - jvp_fn(p)
        r_dot = torch.sum(r * r)

        if r_dot < tol:
            break

        alpha = r_dot / torch.sum(p * Ap)
        x = x + alpha * p
        r_new = r - alpha * Ap
        beta = torch.sum(r_new * r_new) / r_dot
        p = r_new + beta * p
        r = r_new

    return x

@torch.no_grad()
def generate(self, prompt_ids, max_length=100, temperature=1.0, top_k=50):
    """
```

```
Autoregressive generation using the unified model.

At each step:
1. Find equilibrium for current context
2. Decode next token
3. Append and continue

The equilibrium naturally incorporates both:
- Sequential context (via Mamba)
- Retrieved memories (via Hopfield)
"""
generated = prompt_ids.clone()

for _ in range(max_length):
    # Get logits for current sequence
    logits, diagnostics = self.forward(
        generated,
        update_memory=False,
        return_diagnostics=True
    )

    # Take last position logits
    next_token_logits = logits[:, -1, :] / temperature

    # Top-k filtering
    if top_k > 0:
        indices_to_remove = next_token_logits < torch.topk(
            next_token_logits, top_k
        )[0][..., -1, None]
        next_token_logits[indices_to_remove] = float('-inf')

    # Sample
    probs = F.softmax(next_token_logits, dim=-1)
    next_token = torch.multinomial(probs, num_samples=1)

    # Append
    generated = torch.cat([generated, next_token], dim=-1)

    # Optional: Store interesting equilibrium states in memory
    if diagnostics['solver_info']['converged']:
        z_eq = diagnostics['z_equilibrium']
        self.update_memory(z_eq, should_store=torch.ones(1, dtype=torch.bool))

return generated
```

```

```

Phase 5: Advanced Training Infrastructure

Step 5.1: Energy-Based Training Objectives (src/training/objectives.py)

```python
class UnifiedTrainingObjective:
    """
    Multi-component loss function for training the unified model.

    Combines:
    1. Task loss (e.g., language modeling)
    2. Energy regularization (encourage low energy states)
    3. Convergence regularization (encourage fast convergence)
    4. Stability regularization (encourage contractive dynamics)
    """

    def __init__(self,
                 task_weight=1.0,
                 energy_weight=0.1,
                 convergence_weight=0.05,
                 stability_weight=0.01,
                 contraction_target=0.9
                ):
        self.w_task = task_weight
        self.w_energy = energy_weight
        self.w_conv = convergence_weight
        self.w_stab = stability_weight
        self.contraction_target = contraction_target

    def compute_loss(self, model, batch, diagnostics):
        """
        Compute full training loss.

        Args:
            model: UnifiedMambaHopfieldDEQ
            batch: (input_ids, target_ids)
            diagnostics: From forward pass with return_diagnostics=True

        Returns:
```

```
        total_loss, loss_components dict
    """
    input_ids, target_ids = batch
    logits = diagnostics['logits'] if 'logits' in diagnostics else model(input_ids)

    # 1. Task loss (cross-entropy for language modeling)
    loss_task = F.cross_entropy(
        logits.view(-1, logits.size(-1)),
        target_ids.view(-1),
        ignore_index=-100
    )

    # 2. Energy regularization (prefer low-energy equilibria)
    solver_info = diagnostics['solver_info']
    final_energy = solver_info['final_energy']
    energy_components = solver_info['energy_components']

    loss_energy = self.w_energy * final_energy

    # 3. Convergence regularization (prefer fast convergence)
    num_iterations = solver_info['iterations']
    # Penalize high iteration counts
    loss_convergence = self.w_conv * (num_iterations / model.solver.max_iter)

    # 4. Stability regularization (encourage contractive dynamics)
    # Check if Jacobian has eigenvalues < 1
    z_eq = diagnostics['z_equilibrium']
    lipschitz_const = self._estimate_lipschitz(model, z_eq, diagnostics)

    # Penalize if Lipschitz constant > target
    loss_stability = self.w_stab * F.relu(lipschitz_const - self.contraction_target)

    # Total loss
    total_loss = loss_task + loss_energy + loss_convergence + loss_stability

    loss_components = {
        'total': total_loss.item(),
        'task': loss_task.item(),
        'energy': loss_energy.item() if isinstance(loss_energy, torch.Tensor) else loss_energy,
        'convergence': loss_convergence.item() if isinstance(loss_convergence, torch.Tensor)
else loss_convergence,
        'stability': loss_stability.item() if isinstance(loss_stability, torch.Tensor) else loss_stability,
        'num_iterations': num_iterations,
        'lipschitz_constant': lipschitz_const,
```

```
            'energy_components': energy_components
        }

        return total_loss, loss_components

    def _estimate_lipschitz(self, model, z_equilibrium, diagnostics, num_samples=5):
        """
        Estimate Lipschitz constant of dynamics via sampling.

        L = max ||f(z + δ) - f(z)|| / ||δ||
        """
        with torch.no_grad():
            lipschitz_estimates = []

            for _ in range(num_samples):
                # Random perturbation
                epsilon = 0.01
                delta = epsilon * torch.randn_like(z_equilibrium)
                z_perturbed = z_equilibrium + delta

                # Evaluate dynamics at both points
                # (Need to cache context from diagnostics)
                context = diagnostics.get('context', None)
                if context is None:
                    # Skip if context not available
                    return 0.0

                f_z = model.dynamics(z_equilibrium, context, model.memory_patterns)
                f_z_pert = model.dynamics(z_perturbed, context, model.memory_patterns)

                # Lipschitz estimate
                numerator = torch.norm(f_z_pert - f_z, dim=-1).mean()
                denominator = torch.norm(delta, dim=-1).mean()

                lipschitz_estimates.append((numerator / denominator).item())

        return np.mean(lipschitz_estimates)
```

Step 5.2: Curriculum Training (src/training/trainer.py)

```python
class UnifiedModelTrainer:
    """
```

Training loop with curriculum learning for stable DEQ training.

Curriculum stages:
1. Warm-up: Train with few iterations, high tolerance
2. Gradual tightening: Increase iterations, decrease tolerance
3. Full training: Train with target convergence criteria
4. Memory integration: Gradually enable memory updates
"""

```
def __init__(self,
             model,
             optimizer,
             train_loader,
             val_loader,
             objective,
             device='cuda',
             log_wandb=True
            ):
    self.model = model.to(device)
    self.optimizer = optimizer
    self.train_loader = train_loader
    self.val_loader = val_loader
    self.objective = objective
    self.device = device
    self.log_wandb = log_wandb

    self.step = 0
    self.epoch = 0

    # Curriculum schedule
    self.curriculum = {
        'warmup_steps': 1000,
        'max_iter_schedule': [5, 10, 20, 30],  # Increase gradually
        'tolerance_schedule': [1e-2, 5e-3, 1e-3, 5e-4],
        'memory_enable_step': 2000  # When to start updating memory
    }

def get_curriculum_params(self):
    """
    Get current curriculum parameters based on training step.
    """
    if self.step < self.curriculum['warmup_steps']:
        # Warmup: very relaxed
        max_iter = self.curriculum['max_iter_schedule'][0]
```

```
            tolerance = self.curriculum['tolerance_schedule'][0]
        else:
            # Determine stage based on step
            stage = min(
                len(self.curriculum['max_iter_schedule']) - 1,
                (self.step - self.curriculum['warmup_steps']) // 2000
            )
            max_iter = self.curriculum['max_iter_schedule'][stage]
            tolerance = self.curriculum['tolerance_schedule'][stage]

        memory_enabled = self.step >= self.curriculum['memory_enable_step']

        return max_iter, tolerance, memory_enabled

    def train_step(self, batch):
        """
        Single training step with curriculum adjustments.
        """
        self.model.train()

        # Get curriculum parameters
        max_iter, tolerance, memory_enabled = self.get_curriculum_params()

        # Update model solver parameters
        self.model.solver.max_iter = max_iter
        self.model.solver.tol_fp = tolerance
        self.model.solver.tol_energy = tolerance

        # Move batch to device
        input_ids, target_ids = batch
        input_ids = input_ids.to(self.device)
        target_ids = target_ids.to(self.device)

        # Forward pass with diagnostics
        logits, diagnostics = self.model(
            input_ids,
            update_memory=memory_enabled,
            return_diagnostics=True
        )
        diagnostics['logits'] = logits

        # Compute loss
        loss, loss_components = self.objective.compute_loss(
            self.model,
```

```
            (input_ids, target_ids),
            diagnostics
        )

        # Backward pass
        self.optimizer.zero_grad()
        loss.backward()

        # Gradient clipping (essential for DEQ stability)
        torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)

        self.optimizer.step()

        # Logging
        if self.log_wandb and self.step % 10 == 0:
            wandb.log({
                **loss_components,
                'curriculum/max_iter': max_iter,
                'curriculum/tolerance': tolerance,
                'curriculum/memory_enabled': int(memory_enabled),
                'step': self.step
            })

        self.step += 1

        return loss_components

    def validate(self):
        """
        Validation loop with detailed diagnostics.
        """
        self.model.eval()

        total_loss = 0
        total_converged = 0
        total_iterations = []
        energy_trajectories = []

        with torch.no_grad():
            for batch in self.val_loader:
                input_ids, target_ids = batch
                input_ids = input_ids.to(self.device)
                target_ids = target_ids.to(self.device)
```

```
            logits, diagnostics = self.model(
                input_ids,
                update_memory=False,
                return_diagnostics=True
            )
            diagnostics['logits'] = logits

            loss, loss_components = self.objective.compute_loss(
                self.model,
                (input_ids, target_ids),
                diagnostics
            )

            total_loss += loss.item()
            solver_info = diagnostics['solver_info']
            total_converged += int(solver_info['converged'])
            total_iterations.append(solver_info['iterations'])

            if 'energy_history' in solver_info:
                energy_trajectories.append(solver_info['energy_history'])

    # Aggregate statistics
    num_batches = len(self.val_loader)
    val_stats = {
        'val/loss': total_loss / num_batches,
        'val/convergence_rate': total_converged / num_batches,
        'val/avg_iterations': np.mean(total_iterations),
        'val/median_iterations': np.median(total_iterations),
        'val/max_iterations': np.max(total_iterations)
    }

    if self.log_wandb:
        wandb.log(val_stats)

    return val_stats

def train(self, num_epochs):
    """
    Full training loop with validation and checkpointing.
    """
    for epoch in range(num_epochs):
        self.epoch = epoch
        print(f"\nEpoch {epoch + 1}/{num_epochs}")
```

```
            # Training
            epoch_losses = []
            for batch_idx, batch in enumerate(tqdm(self.train_loader)):
                loss_components = self.train_step(batch)
                epoch_losses.append(loss_components['total'])

                # Periodic validation
                if self.step % 500 == 0:
                    val_stats = self.validate()
                    print(f"Step {self.step}: Val Loss = {val_stats['val/loss']:.4f}, "
                          f"Convergence = {val_stats['val/convergence_rate']:.2%}")

            # End-of-epoch validation
            val_stats = self.validate()

            # Checkpointing
            if (epoch + 1) % 5 == 0:
                self.save_checkpoint(f'checkpoint_epoch_{epoch + 1}.pt')

        print("Training complete!")

    def save_checkpoint(self, path):
        """Save model checkpoint with training state."""
        torch.save({
            'model_state_dict': self.model.state_dict(),
            'optimizer_state_dict': self.optimizer.state_dict(),
            'step': self.step,
            'epoch': self.epoch,
            'memory_patterns': self.model.memory_patterns
        }, path)

    def load_checkpoint(self, path):
        """Load model checkpoint."""
        checkpoint = torch.load(path)
        self.model.load_state_dict(checkpoint['model_state_dict'])
        self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
        self.step = checkpoint['step']
        self.epoch = checkpoint['epoch']
        self.model.memory_patterns = checkpoint['memory_patterns']
```

Phase 6: Theoretical Validation Experiments

Step 6.1: Convergence Analysis (experiments/theory/convergence_proofs.py)

```python
class ConvergenceValidator:
    """
    Empirically validate theoretical convergence properties.

    Tests:
    1. Contraction mapping: ||f(z1) - f(z2)|| ≤ L||z1 - z2|| with L < 1
    2. Energy decrease: E(z_{t+1}) ≤ E(z_t) along trajectories
    3. Fixed-point stability: Small perturbations don't destabilize
    4. Lyapunov stability: Energy serves as Lyapunov function
    """

    def __init__(self, model, device='cuda'):
        self.model = model.to(device)
        self.device = device

    def test_contraction_property(self, num_samples=100):
        """
        Test if dynamics are contractive: ||f(z1) - f(z2)|| ≤ L||z1 - z2||

        Sample random states and estimate Lipschitz constant.
        """
        print("Testing contraction property...")

        lipschitz_constants = []

        with torch.no_grad():
            for _ in tqdm(range(num_samples)):
                # Sample two random states
                z1 = torch.randn(1, self.model.d_model, device=self.device)
                z2 = torch.randn(1, self.model.d_model, device=self.device)

                # Dummy context (or use real data)
                context = torch.randn(1, 10, self.model.d_model, device=self.device)

                # Evaluate dynamics
                f_z1 = self.model.dynamics(z1, context, self.model.memory_patterns)
                f_z2 = self.model.dynamics(z2, context, self.model.memory_patterns)

                # Compute Lipschitz constant
                numerator = torch.norm(f_z1 - f_z2)
```

```
            denominator = torch.norm(z1 - z2)

            if denominator > 1e-6:  # Avoid division by zero
                L = (numerator / denominator).item()
                lipschitz_constants.append(L)

    # Statistics
    mean_L = np.mean(lipschitz_constants)
    max_L = np.max(lipschitz_constants)
    pct_contractive = np.mean([L < 1.0 for L in lipschitz_constants])

    print(f"Lipschitz constant: mean={mean_L:.4f}, max={max_L:.4f}")
    print(f"Contractive: {pct_contractive:.1%} of samples")

    return {
        'mean_lipschitz': mean_L,
        'max_lipschitz': max_L,
        'contractive_percentage': pct_contractive,
        'is_contraction': max_L < 1.0
    }

def test_energy_descent(self, num_trajectories=50, num_steps=30):
    """
    Verify that energy decreases along trajectories.

    Initialize random states and iterate dynamics, checking E(z_t) decreases.
    """
    print("Testing energy descent property...")

    descent_violations = 0
    energy_trajectories = []

    with torch.no_grad():
        for _ in tqdm(range(num_trajectories)):
            # Random initialization
            z = torch.randn(1, self.model.d_model, device=self.device)
            context = torch.randn(1, 10, self.model.d_model, device=self.device)

            energies = []

            for step in range(num_steps):
                # Compute current energy
                z_next = self.model.dynamics(z, context, self.model.memory_patterns)
                E, _ = self.model.energy_fn(z, z_next, self.model.memory_patterns)
```

```
                energies.append(E.item())

                # Update state
                z = z_next

            energy_trajectories.append(energies)

            # Check for violations (energy increases)
            for i in range(len(energies) - 1):
                if energies[i + 1] > energies[i] + 1e-4:  # Small tolerance
                    descent_violations += 1
                    break

    violation_rate = descent_violations / num_trajectories

    print(f"Energy descent violations: {violation_rate:.1%}")

    # Visualize some trajectories
    plt.figure(figsize=(10, 6))
    for traj in energy_trajectories[:10]:
        plt.plot(traj, alpha=0.5)
    plt.xlabel('Iteration')
    plt.ylabel('Energy')
    plt.title('Energy Trajectories')
    plt.savefig('energy_descent.png')

    return {
        'violation_rate': violation_rate,
        'energy_trajectories': energy_trajectories,
        'monotonic_descent': violation_rate < 0.1
    }

def test_fixed_point_stability(self, num_fixed_points=20, num_perturbations=10):
    """
    Test stability of converged fixed points.

    Find equilibria, perturb slightly, and check if system returns.
    """
    print("Testing fixed-point stability...")

    stable_count = 0

    with torch.no_grad():
        for _ in tqdm(range(num_fixed_points)):
```

```
        # Find a fixed point
        z_init = torch.randn(1, self.model.d_model, device=self.device)
        context = torch.randn(1, 10, self.model.d_model, device=self.device)

        z_eq, info = self.model.solver.solve(
            z_init, context, self.model.memory_patterns
        )

        if not info['converged']:
            continue

        # Test stability with perturbations
        is_stable = True

        for _ in range(num_perturbations):
            # Small perturbation
            epsilon = 0.01
            perturbation = epsilon * torch.randn_like(z_eq)
            z_perturbed = z_eq + perturbation

            # Re-converge
            z_reconverged, _ = self.model.solver.solve(
                z_perturbed, context, self.model.memory_patterns
            )

            # Check if returns to same fixed point
            distance = torch.norm(z_reconverged - z_eq)

            if distance > 0.1:  # Threshold for "different" fixed point
                is_stable = False
                break

        if is_stable:
            stable_count += 1

stability_rate = stable_count / num_fixed_points

print(f"Stable fixed points: {stability_rate:.1%}")

return {
    'stability_rate': stability_rate,
    'is_stable': stability_rate > 0.8
}
```

```
def test_lyapunov_function(self, num_samples=50):
    """
    Verify energy function acts as Lyapunov function.

    Properties:
    1. E(z*) is minimized at fixed points
    2. dE/dt < 0 along trajectories (except at equilibria)
    """
    print("Testing Lyapunov property...")

    lyapunov_satisfied = 0

    with torch.no_grad():
        for _ in tqdm(range(num_samples)):
            # Random trajectory
            z = torch.randn(1, self.model.d_model, device=self.device)
            context = torch.randn(1, 10, self.model.d_model, device=self.device)

            # Iterate and track energy changes
            is_lyapunov = True

            for _ in range(20):
                z_next = self.model.dynamics(z, context, self.model.memory_patterns)

                E_current, _ = self.model.energy_fn(z, z_next, self.model.memory_patterns)
                E_next, _ = self.model.energy_fn(z_next, z_next, self.model.memory_patterns)

                # Check if energy decreases
                if E_next > E_current + 1e-4:
                    is_lyapunov = False
                    break

                # Stop if converged (dE ≈ 0 is OK at equilibrium)
                if torch.norm(z_next - z) < 1e-3:
                    break

                z = z_next

            if is_lyapunov:
                lyapunov_satisfied += 1

    lyapunov_rate = lyapunov_satisfied / num_samples

    print(f"Lyapunov property satisfied: {lyapunov_rate:.1%}")
```

```
        return {
            'lyapunov_rate': lyapunov_rate,
            'is_lyapunov': lyapunov_rate > 0.9
        }

    def run_all_tests(self):
        """
        Run complete convergence validation suite.
        """
        print("="*50)
        print("CONVERGENCE VALIDATION SUITE")
        print("="*50)

        results = {}

        results['contraction'] = self.test_contraction_property()
        results['energy_descent'] = self.test_energy_descent()
        results['stability'] = self.test_fixed_point_stability()
        results['lyapunov'] = self.test_lyapunov_function()

        # Overall assessment
        all_pass = (
            results['contraction']['is_contraction'] and
            results['energy_descent']['monotonic_descent'] and
            results['stability']['is_stable'] and
            results['lyapunov']['is_lyapunov']
        )

        print("\n" + "="*50)
        print("SUMMARY")
        print("="*50)
        print(f"Contraction: {'✓' if results['contraction']['is_contraction'] else '
```
✗'}")
print(f"Energy Descent: {'✓' if results['energy_descent']['monotonic_descent'] else '✗'}")
print(f"Stability: {'✓' if results['stability']['is_stable'] else '✗'}")
print(f"Lyapunov: {'✓' if results['lyapunov']['is_lyapunov'] else '✗'}")
print(f"\nOverall: {'✓ ALL TESTS PASSED' if all_pass else '✗ SOME TESTS FAILED'}")
print("="*50)

```
    return results
```

```
---

#### **Step 6.2: Energy Landscape Visualization (experiments/theory/energy_analysis.py)**

```python
class EnergyLandscapeAnalyzer:
    """
    Visualize and analyze the energy landscape of the unified model.

    Creates interactive 2D/3D visualizations of:
    1. Energy surface around equilibria
    2. Basin of attraction
    3. Convergence trajectories
    4. Memory pattern organization
    """

    def __init__(self, model, device='cuda'):
        self.model = model.to(device)
        self.device = device

    def visualize_2d_slice(self, z_equilibrium, context, basis_vectors=None,
                           resolution=50, radius=2.0):
        """
        Visualize 2D slice of energy landscape around an equilibrium.

        Args:
            z_equilibrium: Equilibrium point to center on
            context: Input context
            basis_vectors: (v1, v2) directions to span, or None for PCA
            resolution: Grid resolution
            radius: How far to explore from equilibrium
        """
        print("Computing 2D energy slice...")

        if basis_vectors is None:
            # Use PCA of memory patterns to get meaningful directions
            patterns = self.model.memory_patterns.cpu().numpy()
            pca = PCA(n_components=2)
            pca.fit(patterns)
            v1 = torch.tensor(pca.components_[0], device=self.device, dtype=torch.float32)
            v2 = torch.tensor(pca.components_[1], device=self.device, dtype=torch.float32)
        else:
```
```

```
    v1, v2 = basis_vectors

# Create grid
alpha = np.linspace(-radius, radius, resolution)
beta = np.linspace(-radius, radius, resolution)
A, B = np.meshgrid(alpha, beta)

# Compute energy at each grid point
energies = np.zeros_like(A)
fp_residuals = np.zeros_like(A)

with torch.no_grad():
    for i in tqdm(range(resolution)):
        for j in range(resolution):
            # Point in state space
            z = z_equilibrium + A[i, j] * v1 + B[i, j] * v2
            z = z.unsqueeze(0)  # Add batch dim

            # Compute dynamics and energy
            z_next = self.model.dynamics(z, context, self.model.memory_patterns)
            E, _ = self.model.energy_fn(z, z_next, self.model.memory_patterns)

            energies[i, j] = E.item()
            fp_residuals[i, j] = torch.norm(z - z_next).item()

# Create interactive 3D plot
fig = go.Figure()

# Energy surface
fig.add_trace(go.Surface(
    x=A, y=B, z=energies,
    colorscale='Viridis',
    name='Energy',
    showscale=True,
    opacity=0.9
))

# Mark equilibrium
fig.add_trace(go.Scatter3d(
    x=[0], y=[0], z=[energies[resolution//2, resolution//2]],
    mode='markers',
    marker=dict(size=10, color='red'),
    name='Equilibrium'
))
```

```
    fig.update_layout(
        title='Energy Landscape (2D Slice)',
        scene=dict(
            xaxis_title='Direction 1',
            yaxis_title='Direction 2',
            zaxis_title='Energy'
        ),
        width=900,
        height=700
    )

    fig.write_html('energy_landscape_2d.html')
    print("Saved to energy_landscape_2d.html")

    # Also plot fixed-point residuals
    fig2, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

    im1 = ax1.contourf(A, B, energies, levels=20, cmap='viridis')
    ax1.set_title('Energy Landscape')
    ax1.set_xlabel('Direction 1')
    ax1.set_ylabel('Direction 2')
    ax1.plot(0, 0, 'r*', markersize=15, label='Equilibrium')
    ax1.legend()
    plt.colorbar(im1, ax=ax1)

    im2 = ax2.contourf(A, B, fp_residuals, levels=20, cmap='plasma')
    ax2.set_title('Fixed-Point Residual')
    ax2.set_xlabel('Direction 1')
    ax2.set_ylabel('Direction 2')
    ax2.plot(0, 0, 'r*', markersize=15)
    plt.colorbar(im2, ax=ax2)

    plt.tight_layout()
    plt.savefig('energy_landscape_contours.png', dpi=150)
    print("Saved to energy_landscape_contours.png")

    return energies, fp_residuals

def visualize_convergence_trajectories(self, num_trajectories=10, num_steps=50):
    """
    Visualize multiple convergence trajectories in state space.

    Projects high-dimensional trajectories to 2D using PCA.
```

```
"""
print("Computing convergence trajectories...")

trajectories = []
energies_list = []

with torch.no_grad():
    for _ in tqdm(range(num_trajectories)):
        # Random initialization
        z = torch.randn(1, self.model.d_model, device=self.device)
        context = torch.randn(1, 10, self.model.d_model, device=self.device)

        traj = [z.cpu().numpy().flatten()]
        energies = []

        for step in range(num_steps):
            z_next = self.model.dynamics(z, context, self.model.memory_patterns)
            E, _ = self.model.energy_fn(z, z_next, self.model.memory_patterns)

            traj.append(z_next.cpu().numpy().flatten())
            energies.append(E.item())

            # Stop if converged
            if torch.norm(z_next - z) < 1e-4:
                break

            z = z_next

        trajectories.append(np.array(traj))
        energies_list.append(energies)

# Project to 2D using PCA
all_points = np.vstack(trajectories)
pca = PCA(n_components=2)
all_points_2d = pca.fit_transform(all_points)

# Split back into trajectories
trajectories_2d = []
idx = 0
for traj in trajectories:
    traj_len = len(traj)
    trajectories_2d.append(all_points_2d[idx:idx+traj_len])
    idx += traj_len
```

```
    # Plot
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))

    # Trajectories in state space
    for traj_2d in trajectories_2d:
        ax1.plot(traj_2d[:, 0], traj_2d[:, 1], 'o-', alpha=0.6, markersize=3)
        ax1.plot(traj_2d[0, 0], traj_2d[0, 1], 'go', markersize=8)  # Start
        ax1.plot(traj_2d[-1, 0], traj_2d[-1, 1], 'r*', markersize=12)  # End

    ax1.set_title('Convergence Trajectories (PCA projection)')
    ax1.set_xlabel('PC1')
    ax1.set_ylabel('PC2')
    ax1.legend(['Trajectories', 'Start', 'Equilibrium'], loc='best')
    ax1.grid(True, alpha=0.3)

    # Energy vs iteration
    for energies in energies_list:
        ax2.plot(energies, alpha=0.6)

    ax2.set_title('Energy During Convergence')
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Energy')
    ax2.grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('convergence_trajectories.png', dpi=150)
    print("Saved to convergence_trajectories.png")

    return trajectories_2d, energies_list

def visualize_basin_of_attraction(self, z_equilibrium, context,
                                  resolution=30, radius=3.0):
    """
    Identify and visualize basin of attraction around an equilibrium.

    Sample points in a region and color by which equilibrium they converge to.
    """
    print("Analyzing basin of attraction...")

    # Get two principal directions
    patterns = self.model.memory_patterns.cpu().numpy()
    pca = PCA(n_components=2)
    pca.fit(patterns)
    v1 = torch.tensor(pca.components_[0], device=self.device, dtype=torch.float32)
```

```
v2 = torch.tensor(pca.components_[1], device=self.device, dtype=torch.float32)

# Create grid
alpha = np.linspace(-radius, radius, resolution)
beta = np.linspace(-radius, radius, resolution)
A, B = np.meshgrid(alpha, beta)

# Track which equilibrium each point converges to
convergence_map = np.zeros_like(A)
final_energies = np.zeros_like(A)

with torch.no_grad():
    for i in tqdm(range(resolution)):
        for j in range(resolution):
            # Initial point
            z_init = z_equilibrium + A[i, j] * v1 + B[i, j] * v2
            z_init = z_init.unsqueeze(0)

            # Converge
            z_final, info = self.model.solver.solve(
                z_init, context, self.model.memory_patterns
            )

            # Store results
            convergence_map[i, j] = 1 if info['converged'] else 0
            final_energies[i, j] = info['final_energy']

# Visualize
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Convergence success map
im1 = ax1.contourf(A, B, convergence_map, levels=[0, 0.5, 1],
                   colors=['red', 'green'], alpha=0.6)
ax1.set_title('Basin of Attraction\n(Green = Converged, Red = Failed)')
ax1.set_xlabel('Direction 1')
ax1.set_ylabel('Direction 2')
ax1.plot(0, 0, 'k*', markersize=15, label='Target Equilibrium')
ax1.legend()

# Final energy map
im2 = ax2.contourf(A, B, final_energies, levels=20, cmap='viridis')
ax2.set_title('Final Energy Values')
ax2.set_xlabel('Direction 1')
ax2.set_ylabel('Direction 2')
```

```
    ax2.plot(0, 0, 'r*', markersize=15)
    plt.colorbar(im2, ax=ax2)

    plt.tight_layout()
    plt.savefig('basin_of_attraction.png', dpi=150)
    print("Saved to basin_of_attraction.png")

    # Statistics
    convergence_rate = convergence_map.mean()
    print(f"Convergence rate in explored region: {convergence_rate:.1%}")

    return convergence_map, final_energies

def visualize_memory_organization(self):
    """
    Visualize how memory patterns are organized in state space.

    Uses t-SNE or UMAP to project patterns to 2D.
    """
    print("Visualizing memory organization...")

    patterns = self.model.memory_patterns.cpu().numpy()

    # Dimensionality reduction
    from sklearn.manifold import TSNE
    tsne = TSNE(n_components=2, random_state=42)
    patterns_2d = tsne.fit_transform(patterns)

    # Compute pairwise similarities
    with torch.no_grad():
        similarities = torch.matmul(
            self.model.memory_patterns,
            self.model.memory_patterns.T
        ).cpu().numpy()

    # Plot
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 7))

    # Memory pattern layout
    scatter = ax1.scatter(patterns_2d[:, 0], patterns_2d[:, 1],
                          c=np.arange(len(patterns)), cmap='tab20', alpha=0.6)
    ax1.set_title('Memory Pattern Organization (t-SNE)')
    ax1.set_xlabel('t-SNE 1')
    ax1.set_ylabel('t-SNE 2')
```

```
    # Similarity matrix
    im = ax2.imshow(similarities, cmap='viridis', aspect='auto')
    ax2.set_title('Memory Pattern Similarity Matrix')
    ax2.set_xlabel('Pattern Index')
    ax2.set_ylabel('Pattern Index')
    plt.colorbar(im, ax=ax2)

    plt.tight_layout()
    plt.savefig('memory_organization.png', dpi=150)
    print("Saved to memory_organization.png")

    # Cluster analysis
    from sklearn.cluster import KMeans
    n_clusters = min(10, len(patterns) // 100)
    if n_clusters > 1:
        kmeans = KMeans(n_clusters=n_clusters, random_state=42)
        clusters = kmeans.fit_predict(patterns)

        print(f"\nMemory clustering ({n_clusters} clusters):")
        for i in range(n_clusters):
            count = (clusters == i).sum()
            print(f"  Cluster {i}: {count} patterns ({count/len(patterns):.1%})")

    return patterns_2d, similarities

def analyze_critical_points(self, num_samples=100):
    """
    Find and classify critical points of the energy function.

    Critical points where ∇E = 0:
    - Minima (stable equilibria)
    - Maxima (unstable)
    - Saddle points
    """
    print("Analyzing critical points...")

    critical_points = []

    with torch.no_grad():
        for _ in tqdm(range(num_samples)):
            # Random initialization
            z = torch.randn(1, self.model.d_model, device=self.device)
            context = torch.randn(1, 10, self.model.d_model, device=self.device)
```

```
            # Converge to equilibrium
            z_eq, info = self.model.solver.solve(
                z, context, self.model.memory_patterns
            )

            if not info['converged']:
                continue

            # Compute gradient norm at equilibrium
            grad_norm = torch.norm(
                self.model.energy_fn.energy_gradient(z_eq, self.model.memory_patterns)
            ).item()

            # Compute Hessian eigenvalues for classification
            # (Expensive, so we approximate)
            eigenvalues = self._estimate_hessian_eigenvalues(z_eq, context)

            critical_points.append({
                'position': z_eq.cpu().numpy(),
                'energy': info['final_energy'],
                'grad_norm': grad_norm,
                'eigenvalues': eigenvalues,
                'type': self._classify_critical_point(eigenvalues)
            })

        # Summarize
        types = [cp['type'] for cp in critical_points]
        type_counts = {t: types.count(t) for t in set(types)}

        print("\nCritical Point Analysis:")
        for cp_type, count in type_counts.items():
            print(f"  {cp_type}: {count} ({count/len(critical_points):.1%})")

        return critical_points

    def _estimate_hessian_eigenvalues(self, z, context, num_samples=5):
        """
        Estimate Hessian eigenvalues using finite differences.

        Full Hessian is too expensive, so we sample random directions.
        """
        eigenvalues = []
```

```
        with torch.enable_grad():
            for _ in range(num_samples):
                v = torch.randn_like(z)
                v = v / torch.norm(v)

                # Compute Hessian-vector product: H @ v
                z_param = z.detach().requires_grad_(True)
                z_next = self.model.dynamics(z_param, context, self.model.memory_patterns)
                E, _ = self.model.energy_fn(z_param, z_next, self.model.memory_patterns)

                grad_E = torch.autograd.grad(E, z_param, create_graph=True)[0]
                Hv = torch.autograd.grad(grad_E, z_param, v, retain_graph=False)[0]

                # Rayleigh quotient approximation
                eigenval = (v * Hv).sum().item()
                eigenvalues.append(eigenval)

        return eigenvalues

    def _classify_critical_point(self, eigenvalues):
        """
        Classify critical point based on Hessian eigenvalues.

        - All positive: Local minimum (stable)
        - All negative: Local maximum (unstable)
        - Mixed: Saddle point
        """
        pos = sum(1 for e in eigenvalues if e > 0.01)
        neg = sum(1 for e in eigenvalues if e < -0.01)

        if pos == len(eigenvalues):
            return 'Minimum (Stable)'
        elif neg == len(eigenvalues):
            return 'Maximum (Unstable)'
        else:
            return 'Saddle Point'
```

Phase 7: Practical Task Experiments

Step 7.1: Memory-Intensive Tasks (experiments/tasks/memory_tasks.py)

```python
class AssociativeRecallExperiment:
    """
    Test the model's ability to store and retrieve key-value associations.

    Task: Given N key-value pairs in context, retrieve value for query key.
    Tests both Mamba's context processing and Hopfield's associative memory.
    """

    def __init__(self, model, vocab_size=1000, max_pairs=50):
        self.model = model
        self.vocab_size = vocab_size
        self.max_pairs = max_pairs

    def generate_task(self, num_pairs, query_position='random'):
        """
        Generate a single associative recall instance.

        Format: [key1] [value1] [key2] [value2] ... [query_key] [?]
        Target: value corresponding to query_key
        """
        # Sample unique keys and values
        keys = torch.randint(0, self.vocab_size // 2, (num_pairs,))
        values = torch.randint(self.vocab_size // 2, self.vocab_size, (num_pairs,))

        # Build sequence
        sequence = []
        for k, v in zip(keys, values):
            sequence.extend([k.item(), v.item()])

        # Add query
        if query_position == 'random':
            query_idx = torch.randint(0, num_pairs, (1,)).item()
        else:
            query_idx = query_position

        query_key = keys[query_idx].item()
        target_value = values[query_idx].item()

        sequence.extend([query_key, -100])  # -100 as placeholder

        input_ids = torch.tensor(sequence[:-1]).unsqueeze(0)
        target_ids = torch.tensor(sequence[1:]).unsqueeze(0)
        target_ids[0, :-1] = -100  # Only predict last token
```

```
    return input_ids, target_ids, target_value

def evaluate(self, num_trials=100, num_pairs_range=(5, 50)):
    """
    Evaluate across different numbers of pairs.
    """
    results = {n: {'correct': 0, 'total': 0} for n in range(5, 51, 5)}

    self.model.eval()
    with torch.no_grad():
        for _ in tqdm(range(num_trials)):
            num_pairs = torch.randint(num_pairs_range[0], num_pairs_range[1] + 1, (1,)).item()

            input_ids, target_ids, target_value = self.generate_task(num_pairs)
            input_ids = input_ids.to(self.model.device)

            logits = self.model(input_ids)
            prediction = logits[0, -1].argmax().item()

            # Round to nearest bracket for counting
            bracket = (num_pairs // 5) * 5
            if bracket in results:
                results[bracket]['total'] += 1
                if prediction == target_value:
                    results[bracket]['correct'] += 1

    # Compute accuracies
    accuracies = {}
    for n, stats in results.items():
        if stats['total'] > 0:
            accuracies[n] = stats['correct'] / stats['total']
        else:
            accuracies[n] = 0.0

    # Plot
    plt.figure(figsize=(10, 6))
    pairs = sorted(accuracies.keys())
    accs = [accuracies[p] for p in pairs]
    plt.plot(pairs, accs, 'o-', linewidth=2, markersize=8)
    plt.xlabel('Number of Key-Value Pairs')
    plt.ylabel('Accuracy')
    plt.title('Associative Recall Performance')
    plt.grid(True, alpha=0.3)
```

```
        plt.ylim([0, 1.05])
        plt.savefig('associative_recall_results.png', dpi=150)

        return accuracies


class ContinualLearningExperiment:
    """
    Test continual learning: learn new tasks without forgetting old ones.

    The unified model should leverage Hopfield memory to store task-specific
    patterns without catastrophic forgetting.
    """

    def __init__(self, model, num_tasks=5):
        self.model = model
        self.num_tasks = num_tasks
        self.tasks = self._generate_tasks()

    def _generate_tasks(self):
        """
        Create distinct sequential tasks (e.g., different copy patterns).
        """
        tasks = []
        for task_id in range(self.num_tasks):
            # Each task: copy a specific subset of tokens
            start_token = task_id * 100
            end_token = (task_id + 1) * 100
            tasks.append({
                'id': task_id,
                'name': f'Copy tokens [{start_token}-{end_token})',
                'token_range': (start_token, end_token)
            })
        return tasks

    def train_task(self, task_id, num_steps=100):
        """
        Train on a single task.
        """
        task = self.tasks[task_id]
        start, end = task['token_range']

        self.model.train()
        optimizer = torch.optim.Adam(self.model.parameters(), lr=1e-4)
```

```
    for step in range(num_steps):
        # Generate task data
        seq_len = 10
        input_tokens = torch.randint(start, end, (1, seq_len))
        target_tokens = input_tokens.clone()

        input_tokens = input_tokens.to(self.model.device)
        target_tokens = target_tokens.to(self.model.device)

        # Forward pass
        logits = self.model(input_tokens, update_memory=True)
        loss = F.cross_entropy(
            logits.view(-1, logits.size(-1)),
            target_tokens.view(-1)
        )

        # Backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

def evaluate_all_tasks(self):
    """
    Evaluate performance on all tasks learned so far.
    """
    self.model.eval()
    results = {}

    with torch.no_grad():
        for task in self.tasks:
            task_id = task['id']
            start, end = task['token_range']

            # Test accuracy
            correct = 0
            total = 0

            for _ in range(20):
                seq_len = 10
                input_tokens = torch.randint(start, end, (1, seq_len))
                target_tokens = input_tokens.clone()

                input_tokens = input_tokens.to(self.model.device)
```

```
                logits = self.model(input_tokens, update_memory=False)
                predictions = logits.argmax(dim=-1)

                correct += (predictions == target_tokens.to(self.model.device)).sum().item()
                total += seq_len

            accuracy = correct / total
            results[task_id] = accuracy

    return results

def run_continual_learning(self):
    """
    Train tasks sequentially and measure forgetting.
    """
    history = []

    for task_id in range(self.num_tasks):
        print(f"\nTraining Task {task_id}...")
        self.train_task(task_id)

        # Evaluate all tasks
        accuracies = self.evaluate_all_tasks()
        history.append(accuracies)

        print(f"After training task {task_id}:")
        for tid, acc in accuracies.items():
            if tid <= task_id:
                print(f"  Task {tid}: {acc:.2%}")

    # Visualize forgetting
    self._plot_forgetting(history)

    return history

def _plot_forgetting(self, history):
    """
    Plot accuracy matrix showing forgetting over time.
    """
    num_evaluated = len(history)
    matrix = np.zeros((num_evaluated, self.num_tasks))

    for i, accuracies in enumerate(history):
```

```
            for task_id, acc in accuracies.items():
                if task_id < self.num_tasks:
                    matrix[i, task_id] = acc

        plt.figure(figsize=(10, 8))
        plt.imshow(matrix.T, aspect='auto', cmap='RdYlGn', vmin=0, vmax=1)
        plt.colorbar(label='Accuracy')
        plt.xlabel('After Training Task N')
        plt.ylabel('Task ID')
        plt.title('Continual Learning: Accuracy Matrix\n(Diagonal = current task, Off-diagonal =
retention)')
        plt.xticks(range(num_evaluated), [f'Task {i}' for i in range(num_evaluated)])
        plt.yticks(range(self.num_tasks), [f'Task {i}' for i in range(self.num_tasks)])
        plt.tight_layout()
        plt.savefig('continual_learning_results.png', dpi=150)

        # Compute forgetting metric
        forgetting = []
        for task_id in range(self.num_tasks - 1):
            peak_acc = matrix[task_id, task_id]
            final_acc = matrix[-1, task_id]
            forgetting.append(peak_acc - final_acc)

        avg_forgetting = np.mean(forgetting)
        print(f"\nAverage forgetting: {avg_forgetting:.2%}")
```

Phase 8: Documentation & Deployment

Step 8.1: Comprehensive Documentation (docs/theory.md)

```markdown
# Unified Mamba-Hopfield-DEQ: Theoretical Foundation

## Overview

This architecture unifies three powerful paradigms into a single coherent framework where
memory retrieval, sequence processing, and iterative reasoning emerge from a common
optimization objective.

## Mathematical Framework
```

1. Energy Function

The system minimizes a unified energy function:
```

E(z, x, M) = E_Hopfield(z, M) + E_consistency(z, x) + E_reg(z)

```
Where:
- `E_Hopfield(z, M) = -log(Σᵢ exp(β⟨z, mᵢ⟩))`: Modern Hopfield energy for pattern retrieval
- `E_consistency(z, x) = ||z - f_mamba(z, x)||²`: DEQ fixed-point residual
- `E_reg(z) = λ||z||²`: Regularization for bounded solutions

### 2. Unified Dynamics

The dynamics function combines temporal processing (Mamba) and associative retrieval (Hopfield):
```

z_{t+1} = f(z_t, x, M) = g(Mamba(z_t, x), Hopfield(z_t, M))

```
Where `g` is a learned gating function that adaptively blends both contributions.

### 3. Equilibrium Conditions

Convergence occurs when both conditions are satisfied:
1. **Fixed-point**: `z* = f(z*, x, M)`
2. **Energy minimum**: `∇E(z*) = 0`

## Convergence Guarantees

### Theorem 1: Existence of Equilibria

If the dynamics `f` are contractive (Lipschitz constant L < 1) and the energy function is bounded below, then equilibria exist.

**Proof sketch**: Banach fixed-point theorem + energy minimization principles.

### Theorem 2: Lyapunov Stability

The energy function E serves as a Lyapunov function, guaranteeing stable convergence.

**Proof**: E decreases monotonically along trajectories: `E(z_{t+1}) ≤ E(z_t)` for all t.

### Theorem 3: Compositional Memory

The Hopfield component enables compositional operations: binding, unbinding, and superposition of patterns with graceful degradation.

## Computational Complexity

- **Forward pass**: O(L·D²) where L = sequence length, D = model dimension
  - Mamba: O(L·D) per layer (linear in sequence
```

length)

- Hopfield: O(M·D) where M = number of memory patterns
- DEQ: O(K·L·D²) where K = convergence iterations
- **Backward pass**: O(D³) for implicit differentiation
  - Uses conjugate gradient to avoid explicit Jacobian
  - Memory: O(D²) instead of O(K·D²)
- **Memory storage**: O(M·D) for patterns, independent of sequence length

Comparison to Alternatives

Property	Transformer	Mamba	MHN-only	**Unified (Ours)**
Sequence complexity	O(L²)	O(L)	O(L)	O(L)
Memory capacity	O(L·D)	O(D)	O(M·D)	O(M·D)
Associative retrieval	✗	✗	✓	✓
Iterative reasoning	✗	✗	✗	✓
Convergence guarantees	N/A	N/A	✓	✓
Continual learning	✗	✗	✓	✓

Key Innovations

1. **Unified optimization**: Single equilibrium satisfies both temporal consistency and memory retrieval
1. **Implicit depth**: DEQ wrapper provides unbounded reasoning depth with constant memory
1. **Compositional memory**: Hopfield enables structured knowledge representation
1. **Theoretical guarantees**: Provable convergence under mild conditions

References

- Mamba: Gu & Dao (2023) - Selective State Space Models
- Modern Hopfield Networks: Ramsauer et al. (2020) - Hopfield Networks is All You Need
```

- Deep Equilibrium Models: Bai et al. (2019) - Deep Equilibrium Models

```
---

#### **Step 8.2: README with Examples (README.md)**

```markdown
# Unified Mamba-Hopfield-DEQ Architecture

A research implementation unifying three powerful paradigms:
- **Mamba**: Efficient selective state space models for sequence processing
- **Modern Hopfield Networks**: Associative memory with exponential capacity
- **Deep Equilibrium Models**: Implicit depth through fixed-point computation

## Installation

```bash
git clone https://github.com/your-repo/unified-mamba-hopfield-deq
cd unified-mamba-hopfield-deq
pip install -r requirements.txt
pip install -e .
```

Quick Start

Basic Usage

```python
import torch
from src.models.unified import UnifiedMambaHopfieldDEQ

# Initialize model
model = UnifiedMambaHopfieldDEQ(
    vocab_size=10000,
    d_model=512,
    d_state=64,
    memory_size=5000,
    solver_type='alternating'
)

# Forward pass
input_ids = torch.randint(0, 10000, (2, 128))  # (batch, seq_len)
logits = model(input_ids)  # (batch, seq_len, vocab_size)
```

```
# With diagnostics
logits, diagnostics = model(input_ids, return_diagnostics=True)
print(f"Converged: {diagnostics['solver_info']['converged']}")
print(f"Iterations: {diagnostics['solver_info']['iterations']}")
print(f"Final energy: {diagnostics['solver_info']['final_energy']:.4f}")
```

Training

```python
from src.training.trainer import UnifiedModelTrainer
from src.training.objectives import UnifiedTrainingObjective

# Setup training
objective = UnifiedTrainingObjective(
    task_weight=1.0,
    energy_weight=0.1,
    convergence_weight=0.05
)

trainer = UnifiedModelTrainer(
    model=model,
    optimizer=torch.optim.AdamW(model.parameters(), lr=1e-4),
    train_loader=train_loader,
    val_loader=val_loader,
    objective=objective
)

# Train with curriculum
trainer.train(num_epochs=10)
```

Generation

```python
# Autoregressive generation
prompt = "Once upon a time"
prompt_ids = tokenizer.encode(prompt)
prompt_tensor = torch.tensor(prompt_ids).unsqueeze(0)

generated = model.generate(
    prompt_tensor,
    max_length=100,
```

```
    temperature=0.8,
    top_k=50
)

print(tokenizer.decode(generated[0]))
```

Architecture Details

Information Flow

```
Input Tokens
    ↓
Embedding
    ↓
Mamba Layers (process sequence)
    ↓
Extract State z₀
    ↓
┌─────────────────────────────────────┐
│  DEQ Equilibrium Finding            │
│                                     │
│  Iterate until convergence:         │
│  1. Query Hopfield memory           │
│  2. Update Mamba state              │
│  3. Compute energy                  │
│  4. Check convergence               │
└─────────────────────────────────────┘
    ↓
Equilibrium State z*
    ↓
Output Projection
    ↓
Logits
```

Solver Options

Three solver modes available:

1. **Alternating** (default): Alternates between fixed-point and energy descent

- Best for stability

- Slower convergence

1. **Simultaneous**: Jointly optimizes both objectives

- Faster convergence
- Requires careful hyperparameter tuning

1. **Cascade**: Fixed-point first, then energy refinement

- Good when objectives are well-aligned
- Most efficient when it works

Experiments

1. Theoretical Validation

Verify convergence properties:

```python
from experiments.theory.convergence_proofs import ConvergenceValidator

validator = ConvergenceValidator(model)
results = validator.run_all_tests()
```

2. Energy Landscape Analysis

Visualize energy surfaces:

```python
from experiments.theory.energy_analysis import EnergyLandscapeAnalyzer

analyzer = EnergyLandscapeAnalyzer(model)
analyzer.visualize_2d_slice(z_equilibrium, context)
analyzer.visualize_convergence_trajectories()
analyzer.visualize_basin_of_attraction(z_equilibrium, context)
```

3. Associative Recall

Test memory capabilities:

```python
from experiments.tasks.memory_tasks import AssociativeRecallExperiment
```

```
experiment = AssociativeRecallExperiment(model)
accuracies = experiment.evaluate(num_trials=100)
```

4. Continual Learning

Measure catastrophic forgetting:

```python
from experiments.tasks.memory_tasks import ContinualLearningExperiment

experiment = ContinualLearningExperiment(model, num_tasks=5)
history = experiment.run_continual_learning()
```

Configuration

Key hyperparameters:

```python
config = {
    # Model architecture
    'd_model': 512,             # Hidden dimension
    'd_state': 64,              # SSM state dimension
    'd_conv': 4,                # Convolution width
    'n_layers': 6,              # Number of Mamba layers

    # Memory
    'memory_size': 10000,       # Number of storable patterns
    'beta': 2.0,                # Hopfield inverse temperature

    # DEQ solver
    'solver_type': 'alternating',
    'max_iterations': 30,
    'tol_fixedpoint': 1e-3,
    'tol_energy': 1e-3,

    # Training
    'learning_rate': 1e-4,
    'batch_size': 32,
    'gradient_clip': 1.0
}
```

Performance Tips

Memory Efficiency

```python
# Enable gradient checkpointing
model.dynamics.mamba.gradient_checkpointing_enable()

# Limit convergence iterations during training
model.solver.max_iter = 10  # Increase gradually

# Use mixed precision
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()
with autocast():
    logits = model(input_ids)
```

Faster Convergence

```python
# Warm-start with previous equilibrium
previous_z = None
for batch in dataloader:
    logits, diagnostics = model(batch, z_init=previous_z)
    previous_z = diagnostics['z_equilibrium'].detach()
```

Better Stability

```python
# Increase regularization
objective = UnifiedTrainingObjective(
    stability_weight=0.1,  # Encourage contractive dynamics
    contraction_target=0.8
)

# Reduce learning rate for memory patterns
optimizer = torch.optim.AdamW([
    {'params': [p for n, p in model.named_parameters() if 'memory' not in n]},
    {'params': [model.memory_patterns], 'lr': 1e-5}
], lr=1e-4)
```

```

## Troubleshooting

### Issue: DEQ doesn't converge

**Solutions:**

- Reduce `max_iter` initially and gradually increase
- Increase `tolerance` during warmup
- Check Lipschitz constant (should be < 1)
- Add stability regularization

### Issue: NaN losses

**Solutions:**

- Enable gradient clipping (max_norm=1.0)
- Reduce learning rate
- Check energy function components (one might be exploding)
- Use mixed precision cautiously

### Issue: Slow training

**Solutions:**

- Reduce `max_iter` (quality vs speed tradeoff)
- Use 'cascade' solver (faster than 'alternating')
- Enable gradient checkpointing
- Batch multiple sequences efficiently

## Citation

If you use this code in your research, please cite:

```bibtex
@software{unified_mamba_hopfield_deq,
  title={Unified Mamba-Hopfield-DEQ Architecture},
  author={Your Name},
  year={2024},
  url={https://github.com/your-repo/unified-mamba-hopfield-deq}
}
```
```

License

MIT License - see LICENSE file for details

Acknowledgments

Built on top of:

- [Mamba](https://github.com/state-spaces/mamba) by Gu & Dao
- Modern Hopfield Networks theory by Ramsauer et al.
- DEQ framework by Bai et al.

```
---

#### **Step 8.3: Interactive Demo (notebooks/demo.ipynb)**

```python
# Cell 1: Setup
"""
# Unified Mamba-Hopfield-DEQ Demo

This notebook demonstrates the key capabilities of the unified architecture.
"""

import torch
import numpy as np
import matplotlib.pyplot as plt
from src.models.unified import UnifiedMambaHopfieldDEQ
from src.analysis.convergence import ConvergenceValidator
from src.analysis.energy_landscape import EnergyLandscapeAnalyzer

# Initialize model
model = UnifiedMambaHopfieldDEQ(
    vocab_size=1000,
    d_model=128,
    d_state=32,
    memory_size=500,
    max_iterations=20
)

print("Model initialized!")
print(f"Parameters: {sum(p.numel() for p in model.parameters()):,}")
```
```

```
# Cell 2: Basic Forward Pass
"""
## Basic Usage

Let's run a simple forward pass and examine the equilibrium.
"""

# Create dummy input
input_ids = torch.randint(0, 1000, (1, 20))

# Forward with diagnostics
logits, diag = model(input_ids, return_diagnostics=True)

print("Forward pass complete!")
print(f"Converged: {diag['solver_info']['converged']}")
print(f"Iterations: {diag['solver_info']['iterations']}")
print(f"Final energy: {diag['solver_info']['final_energy']:.4f}")

# Visualize convergence
if 'energy_history' in diag['solver_info']:
    plt.figure(figsize=(10, 4))
    plt.plot(diag['solver_info']['energy_history'], 'o-')
    plt.xlabel('Iteration')
    plt.ylabel('Energy')
    plt.title('Energy During Convergence')
    plt.grid(True, alpha=0.3)
    plt.show()

# Cell 3: Memory Operations
"""
## Memory Dynamics

Examine how memory patterns are stored and retrieved.
"""

# Check current memory usage
memory_stats = diag['memory_usage']
print(f"Memory attention entropy: {memory_stats['attention_entropy']:.4f}")
print(f"Top-10 pattern mass: {memory_stats['top_10_mass']:.4f}")

# Visualize attention over memory
z_eq = diag['z_equilibrium']
with torch.no_grad():
    similarities = torch.matmul(z_eq, model.memory_patterns.T)
```

```
    attention = torch.softmax(similarities, dim=-1)

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.bar(range(len(attention[0])), attention[0].cpu().numpy())
plt.xlabel('Memory Pattern Index')
plt.ylabel('Attention Weight')
plt.title('Memory Retrieval Pattern')

plt.subplot(1, 2, 2)
top_k = 20
top_indices = attention[0].topk(top_k).indices.cpu().numpy()
top_values = attention[0].topk(top_k).values.cpu().numpy()
plt.barh(range(top_k), top_values)
plt.xlabel('Attention Weight')
plt.ylabel('Pattern Rank')
plt.title(f'Top {top_k} Retrieved Patterns')
plt.gca().invert_yaxis()
plt.tight_layout()
plt.show()

# Cell 4: Convergence Analysis
"""
## Theoretical Validation

Test convergence properties empirically.
"""

validator = ConvergenceValidator(model)

print("Running convergence tests...")
print("\n1. Testing contraction property...")
contraction_results = validator.test_contraction_property(num_samples=20)

print("\n2. Testing energy descent...")
descent_results = validator.test_energy_descent(num_trajectories=10, num_steps=20)

print("\n3. Testing fixed-point stability...")
stability_results = validator.test_fixed_point_stability(num_fixed_points=5)

print("\n" + "="*50)
print("RESULTS SUMMARY")
print("="*50)
print(f"✓ Contraction: {contraction_results['is_contraction']}")
```

```
print(f"✓ Energy Descent: {descent_results['monotonic_descent']}")
print(f"✓ Stability: {stability_results['is_stable']}")

# Cell 5: Energy Landscape
"""
## Energy Landscape Visualization

Visualize the energy surface around an equilibrium.
"""

analyzer = EnergyLandscapeAnalyzer(model)

# Get an equilibrium point
with torch.no_grad():
    z_init = torch.randn(1, model.d_model)
    context = torch.randn(1, 10, model.d_model)
    z_eq, _ = model.solver.solve(z_init, context, model.memory_patterns)

# Visualize 2D slice
print("Computing energy landscape (this may take a minute)...")
energies, residuals = analyzer.visualize_2d_slice(
    z_eq, context, resolution=30, radius=2.0
)

print("Landscape visualization saved!")

# Cell 6: Associative Memory Test
"""
## Associative Memory Capabilities

Test key-value retrieval.
"""

def test_associative_recall(model, num_pairs=10):
    """Simple associative recall test."""
    # Create key-value pairs
    keys = torch.randint(0, 500, (num_pairs,))
    values = torch.randint(500, 1000, (num_pairs,))

    # Build sequence: [k1, v1, k2, v2, ..., query_key]
    sequence = []
    for k, v in zip(keys, values):
        sequence.extend([k.item(), v.item()])
```

```
    # Query random key
    query_idx = torch.randint(0, num_pairs, (1,)).item()
    query_key = keys[query_idx].item()
    target_value = values[query_idx].item()

    sequence.append(query_key)
    input_ids = torch.tensor(sequence).unsqueeze(0)

    # Predict
    with torch.no_grad():
        logits = model(input_ids)
        prediction = logits[0, -1].argmax().item()

    correct = (prediction == target_value)
    return correct, prediction, target_value

# Run multiple trials
print("Testing associative recall...")
num_trials = 20
correct_count = 0

for trial in range(num_trials):
    correct, pred, target = test_associative_recall(model, num_pairs=10)
    correct_count += correct
    if trial < 5:  # Show first few
        print(f"Trial {trial+1}: Pred={pred}, Target={target}, {'✓' if correct else '✗'}")

accuracy = correct_count / num_trials
print(f"\nAccuracy: {accuracy:.1%} ({correct_count}/{num_trials})")

# Cell 7: Interactive Exploration
"""
## Interactive Exploration

Modify parameters and observe effects.
"""

from ipywidgets import interact, FloatSlider, IntSlider

@interact(
    beta=FloatSlider(min=0.1, max=5.0, step=0.1, value=2.0),
    max_iter=IntSlider(min=5, max=50, step=5, value=20),
    tolerance=FloatSlider(min=1e-4, max=1e-2, step=1e-4, value=1e-3)
)
```

```
def explore_parameters(beta, max_iter, tolerance):
    """Interactive parameter exploration."""
    # Update model parameters
    model.energy_fn.beta = beta
    model.dynamics.beta = beta
    model.solver.max_iter = max_iter
    model.solver.tol_fp = tolerance
    model.solver.tol_energy = tolerance

    # Run forward pass
    input_ids = torch.randint(0, 1000, (1, 20))
    with torch.no_grad():
        logits, diag = model(input_ids, return_diagnostics=True)

    # Display results
    info = diag['solver_info']
    print(f"Converged: {info['converged']}")
    print(f"Iterations: {info['iterations']}")
    print(f"Final energy: {info['final_energy']:.4f}")

    if 'energy_history' in info:
        plt.figure(figsize=(8, 4))
        plt.plot(info['energy_history'], 'o-')
        plt.xlabel('Iteration')
        plt.ylabel('Energy')
        plt.title(f'Convergence (β={beta}, max_iter={max_iter}, tol={tolerance})')
        plt.grid(True, alpha=0.3)
        plt.show()
```

Prompt for AI Code Assistant (Final Compilation)

When ready to build Option 3, provide this comprehensive prompt:

> **Project: Unified Mamba-Hopfield-DEQ Architecture (Option 3)**
>
> Create a complete research-grade Python implementation where Mamba temporal processing, Modern Hopfield Network memory, and Deep Equilibrium Model reasoning are unified into a single energy-based framework.
>
> **Core Requirements:**
>

> 1. Single energy function that encompasses both Hopfield retrieval and DEQ equilibrium
> 1. Unified dynamics where each iteration performs both Mamba updates and Hopfield retrieval
> 1. Hybrid solver that finds states satisfying both z* = f(z*) AND $\nabla E(z^*) = 0$
> 1. Three solver modes: alternating, simultaneous, and cascade
> 1. Implicit differentiation for memory-efficient backpropagation
>
> **Implementation Order:**
>
> 1. Phase 1: Repository structure and dependencies
> 1. Phase 2: Energy function (src/core/energy.py) with Hopfield + consistency + regularization terms
> 1. Phase 3: Unified dynamics (src/core/dynamics.py) blending Mamba and Hopfield updates
> 1. Phase 4: Hybrid equilibrium solver (src/solvers/hybrid_solver.py) with all three modes
> 1. Phase 5: Full model (src/models/unified.py) integrating all components
> 1. Phase 6: Training infrastructure with energy-based objectives and curriculum learning
> 1. Phase 7: Theoretical validation experiments (convergence, Lyapunov, contraction)
> 1. Phase 8: Practical task experiments (associative recall, continual learning)
> 1. Phase 9: Energy landscape visualization tools
> 1. Phase 10: Documentation and interactive demo notebook
>
> **Key Design Principles:**
>
> - Energy minimization as the primary objective (not just fixed-point)
> - Lyapunov stability guarantees through proper energy design
> - Compositional memory via Hopfield's associative properties
> - Implicit depth via DEQ wrapper
> - Theoretical rigor with empirical validation
>
> **Success Criteria:**
>
> - All convergence tests pass (contraction, energy descent, stability, Lyapunov)
> - Energy landscapes show clear basins of attraction
> - Outperforms baseline on memory-intensive tasks
> - Demonstrates continual learning without catastrophic forgetting
> - Clean, modular, well-documented code with type hints
>
```markdown
# Advanced Unified Architecture: Optimization Variants

Build an enhanced version of the Mamba-Hopfield-DEQ architecture with the following modular improvements. Implement as separate, swappable components that can be mixed and matched.

## Project Structure
```

```

unified-mamba-hopfield-deq-advanced/
├── src/
│   ├── backbones/
│   │   ├── mamba.py            # Original Mamba backbone
│   │   ├── hybrid.py           # Mamba + sparse attention
│   │   ├── retnet.py           # RetNet alternative
│   │   └── rwkv.py             # RWKV alternative
│   ├── memory/
│   │   ├── hopfield.py         # Basic Modern Hopfield
│   │   ├── hierarchical.py     # Multi-level memory (L1/L2/L3)
│   │   ├── sparse_hash.py      # LSH-based sparse memory
│   │   └── product_key.py      # Factorized memory
│   ├── solvers/
│   │   ├── anderson.py         # Anderson acceleration (baseline)
│   │   ├── lbfgs.py            # Quasi-Newton methods
│   │   ├── neural_ode.py       # ODE-based solver
│   │   ├── learned.py          # Meta-learned accelerator
│   │   └── early_exit.py       # Confidence-based early stopping
│   ├── optimizations/
│   │   ├── contractive.py      # Guaranteed Lipschitz < 1
│   │   ├── energy_shaping.py   # Hessian conditioning
│   │   ├── mixed_precision.py  # FP16/FP32 mixing
│   │   └── caching.py          # Warm-start equilibria
│   └── models/
│       └── unified_advanced.py  # Main configurable model

```
Core Components to Implement

1. Hybrid Mamba-Attention Backbone (src/backbones/hybrid.py)

```python
class HybridBackbone(nn.Module):
    """
    Interleave Mamba layers (cheap, O(L)) with sparse attention (expensive, O(L²)).

    Pattern: [Mamba × N, Attention, Mamba × N, Attention, ...]

    Config:
    - attention_every: Insert attention every N Mamba layers (default: 4)
    - attention_type: 'local' (sliding window) or 'global' (full attention)
    - window_size: For local attention (default: 256)
```

    """
```

**Key method**: `forward(x)` returns processed sequence with mixed local/global context

### 2. Hierarchical Memory (src/memory/hierarchical.py)

```python
class HierarchicalMemory(nn.Module):
    """
    Three-tier memory system inspired by biological memory:

    L1 (Episodic): Recent patterns, size ~1K, beta=2.0 (sharp)
    L2 (Semantic): Mid-term patterns, size ~10K, beta=1.0 (medium)
    L3 (Schematic): Abstract schemas, size ~500, beta=0.5 (broad)

    Methods:
    - retrieve(query) -> weighted combination from all levels
    - store(pattern, level='auto') -> add to appropriate level
    - consolidate() -> periodic L1->L2->L3 transfer
    - router(query) -> learned routing weights [w1, w2, w3]
    """
```

Routing: Use small MLP to predict which memory level(s) to query based on query features

3. L-BFGS Solver with Early Exit (src/solvers/lbfgs.py)

```python
class LBFGSSolver:
    """
    Quasi-Newton solver for equilibrium finding.

    Objective: minimize ||z - f(z)||² + λ·E(z)

    Features:
    - L-BFGS optimizer (PyTorch built-in)
    - Early exit: confidence predictor (small MLP) estimates convergence
    - Mixed iterations: first 70% in FP16, last 30% in FP32

    Config:
    - max_iter: 30 (default)
    - min_iter: 5 (minimum before early exit allowed)
    - confidence_threshold: 0.95
```
```

```
    - history_size: 10 (L-BFGS memory)
    """

    def solve(self, z_init, dynamics_fn, energy_fn):
        """
        Returns: (z_equilibrium, info_dict)
        info_dict contains: iterations_used, converged, confidence, energy
        """
```

Confidence predictor: Train auxiliary MLP on (z, residual) -> probability of being converged. Meta-train this across many convergence trajectories.

4. Sparse Hash Memory (src/memory/sparse_hash.py)

```python
class SparseHashMemory(nn.Module):
    """
    Locality-Sensitive Hashing for O(√M) retrieval instead of O(M).

    Uses random projection hashing:
    - num_buckets: 256 (default)
    - patterns stored in buckets
    - retrieve only from top-K buckets

    Methods:
    - hash(query) -> bucket_indices (top K buckets)
    - store(pattern) -> adds to appropriate bucket
    - retrieve(query, top_k=10) -> Hopfield over candidates only
    """
```

Hash function: Learnable random projection: `hash_id = argmax(W @ query)` where W is (d_model, num_buckets)

5. Contractive Dynamics (src/optimizations/contractive.py)

```python
class ContractiveDynamics(nn.Module):
    """
    Guarantee Lipschitz constant < 1 by construction.

    Techniques:
    1. Spectral normalization on all linear layers
```

```
    2. Bounded activations (tanh, sigmoid)
    3. Residual with contraction_factor < 1

    Formula: z_next = z + α·(f(z) - z) where α=0.9

    This mathematically guarantees convergence (Banach fixed-point theorem).
    """
```

6. Equilibrium Caching (src/optimizations/caching.py)

```python
class EquilibriumCache:
    """
    Cache recent equilibria for warm-start initialization.

    - Hash input contexts (using frozen encoder)
    - Store (context_hash, z_equilibrium) pairs
    - LRU cache with max_size=1000
    - On forward pass: find nearest cached context, use as z_init

    Expected speedup: 40-60% fewer iterations when cache hits
    """

    def get_warm_start(self, context_embedding):
        """Returns cached z_init or None"""

    def update(self, context_embedding, z_equilibrium):
        """Add new equilibrium to cache"""
```

Context hashing: Use mean-pooled Mamba output as context signature

7. Energy Shaping (src/optimizations/energy_shaping.py)

```python
class ShapedEnergyFunction(UnifiedEnergyFunction):
    """
    Add regularization terms for better-conditioned energy landscape.

    Additional terms:
    1. Curvature regularization: penalize badly-conditioned Hessian
    2. Saddle avoidance: penalize negative eigenvalues
    3. Basin shaping: encourage single global minimum
```

```
    E_total = E_hop + E_cons + λ1·E_curvature + λ2·E_saddle

    where:
    E_curvature = Σ(λi - target)² for Hessian eigenvalues λi
    E_saddle = -min(0, min(λi))
    """
```

Hessian estimation: Use Hutchinson's trace estimator with random vectors (don't compute full Hessian)

Main Configurable Model (src/models/unified_advanced.py)

```python
class AdvancedUnifiedModel(nn.Module):
    """
    Unified model with swappable components.

    Config dict:
    {
        'backbone': 'hybrid',  # or 'mamba', 'retnet', 'rwkv'
        'backbone_config': {'attention_every': 4},

        'memory': 'hierarchical',  # or 'hopfield', 'sparse_hash', 'product_key'
        'memory_config': {'l1_size': 1000, 'l2_size': 10000, 'l3_size': 500},

        'solver': 'lbfgs',  # or 'anderson', 'neural_ode', 'learned'
        'solver_config': {'max_iter': 30, 'confidence_threshold': 0.95},

        'optimizations': {
            'contractive': True,
            'energy_shaping': True,
            'caching': True,
            'mixed_precision': True,
            'early_exit': True
        }
    }
    """

    def __init__(self, config):
        # Instantiate components based on config
        self.backbone = self._build_backbone(config['backbone'], config['backbone_config'])
        self.memory = self._build_memory(config['memory'], config['memory_config'])
```

```
        self.solver = self._build_solver(config['solver'], config['solver_config'])
        # etc.
```

Implementation Priority

Phase 1 (Core enhancements):

1. Hybrid Mamba-Attention backbone
1. Hierarchical memory
1. L-BFGS solver

Phase 2 (Efficiency):
4. Early exit with confidence
5. Equilibrium caching
6. Mixed precision

Phase 3 (Theoretical):
7. Contractive dynamics
8. Energy shaping
9. Sparse hash memory (for very large scale)

Experiments to Add (experiments/comparisons/)

```python
# experiments/comparisons/ablation_study.py
"""
Compare all variants:
- Baseline (Mamba + Hopfield + Anderson)
- + Hybrid backbone
- + Hierarchical memory
- + L-BFGS solver
- + All optimizations

Metrics:
- Convergence speed (iterations)
- Wall-clock time
- Memory usage
- Task accuracy
- Scaling (vary sequence length, memory size)
"""

# experiments/comparisons/scaling_analysis.py
"""
```

Test scaling behavior:
- Sequence length: 512, 1K, 4K, 16K
- Memory patterns: 1K, 10K, 100K, 1M
- Model size: 125M, 350M, 760M params

Plot: time vs scale for each configuration
"""
```

## Recommended Default Configuration

```python
RECOMMENDED_CONFIG = {
    'backbone': 'hybrid',
    'backbone_config': {
        'attention_every': 4,
        'attention_type': 'local',
        'window_size': 256
    },

    'memory': 'hierarchical',
    'memory_config': {
        'l1_size': 1000,
        'l2_size': 10000,
        'l3_size': 500,
        'consolidation_freq': 1000  # steps
    },

    'solver': 'lbfgs',
    'solver_config': {
        'max_iter': 30,
        'min_iter': 5,
        'confidence_threshold': 0.95,
        'history_size': 10
    },

    'optimizations': {
        'contractive': True,
        'energy_shaping': True,
        'caching': True,
        'mixed_precision': True,
        'early_exit': True
    },

```
    # Training
    'gradient_clip': 1.0,
    'learning_rate': 1e-4,
    'warmup_steps': 1000
}
```

Success Criteria

Must achieve:

- ✅ 2-3x faster convergence than baseline (measured in wall-clock time)
- ✅ Support 10x more memory patterns (100K+) with hierarchical or sparse memory
- ✅ Maintain or improve task accuracy
- ✅ All components work independently and compose correctly

Bonus:

- ✅ Theoretical convergence guarantees hold empirically (with contractive dynamics)
- ✅ Learned accelerator achieves 50%+ iteration reduction after meta-training
- ✅ Scales to 1M+ memory patterns with sparse hashing

Key Implementation Notes

1. **Modularity**: Each component must work independently. Use factory pattern for building components.
1. **Config-driven**: Single config dict controls everything. Easy to run ablations.
1. **Gradients**: Ensure all new components are differentiable (or have custom backward passes).
1. **Testing**: Unit test each component separately before integration.
1. **Logging**: Track per-component metrics (backbone time, solver iterations, cache hit rate, etc.)

Start with Phase 1, implement one component at a time, and validate before moving to the next. Each component should have a standalone test showing it improves on the baseline in its specific dimension (speed, memory, accuracy, etc.).

```

```