# """ DARKBOT™: Resonant Field Intelligence Architecture

Core implementation of the DARKBOT™ system.

This file contains the primary implementation of the DARKBOT™ Resonant Field Intelligence Architecture, including the main DarkBot class and core processing functions.



import torch import numpy as np from typing import Dict, List, Optional, Any, Tuple, Union import asyncio from dataclasses import dataclass import json import yaml from datetime import datetime from pathlib import Path

from .config import DarkBotConfig from .tensor_ops import ( calculate_resonance, calculate_coherence, resonant_product, fractal_transform ) from .lattice import construct_e8_lattice, quaternion_transform

class DarkBot: """ DARKBOT™ Resonant Field Intelligence Architecture.

```
The DarkBot class implements a field-based computational intelligence
paradigm using quantum-inspired resonance principles structured through
a proprietary 369-157-248 numerological encoding.

Attributes:
    config (DarkBotConfig): System configuration parameters
    device (torch.device): Device used for computation (CPU or CUDA)
    dtype (torch.dtype): Default dtype for tensors (complex64)
    kvantum_tilstand (torch.Tensor): Current quantum-inspired field state
    serie_prosessor (Dict): Serie-coupled processors in 369 architecture
    e8_routing (torch.Tensor): E8 lattice routing matrix
    fractal_memory (List): List of previous field states
    temporal_memory (List): Temporal memory for 157 architecture
    coherence_history (List): History of field coherence values
"""

def __init__(self, config: Optional[DarkBotConfig] = None):
    """
    Initialize the DARKBOT™ system.

    Args:
        config: Configuration object for the system. If None, uses defaults.
    """
    # Load default configuration if none provided
    self.config = config or DarkBotConfig()

    # Establish deterministic field coherence
    torch.manual_seed(369)
    np.random.seed(369)

    # Set global device and dtype defaults
    self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.dtype = torch.complex64
    torch.set_default_dtype(torch.float32)  # Base type for real components

    # Ensure CUDA determinism if available
    if self.device.type == "cuda":
        torch.cuda.manual_seed_all(369)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False

    # Tensor conversion utility for NumPy interoperability
    self.to_tensor = lambda x: torch.from_numpy(x).to(device=self.device, dtype=self.dtype)
if isinstance(x, np.ndarray) else x

    # Initialize system components
```

```
        self._initialize_system_components()

    def _initialize_system_components(self):
        """Initialize the core components of the DARKBOT™ system."""
        D = self.config.field.dimensions

        # Initialize quantum-inspired field
        self.kvantum_tilstand = self._initialize_quantum_state()

        # Initialize serie-coupled processors (369 architecture)
        self.serie_prosessor = {
            'primary': torch.zeros(D, dtype=self.dtype, device=self.device),         # 3-space
            'secondary': torch.zeros(D // 2, dtype=self.dtype, device=self.device),  # 6-space
            'tertiary': torch.zeros(D // 4, dtype=self.dtype, device=self.device)    # 9-space
        }

        # Initialize E8 routing matrix (248 architecture)
        self.e8_routing = construct_e8_lattice(device=self.device)

        # Initialize memory structures
        self.fractal_memory = []
        self.temporal_memory = []
        self.coherence_history = []

        # Initialize resonant fields
        self._recursive_field = torch.zeros(D, dtype=self.dtype, device=self.device)
        self._proximity_field = torch.zeros(D, dtype=self.dtype, device=self.device)
        self._gravitational_field = torch.zeros(D, dtype=self.dtype, device=self.device)
        self._branch_funnel_field = torch.zeros(D, dtype=self.dtype, device=self.device)

        # Current cycle phase
        self.current_phase = 0

        # Initialize reference states (for coherence calculation)
        self._initialize_reference_states()

    def _initialize_quantum_state(self) -> torch.Tensor:
        """
        Initialize the quantum-inspired field state.

        Returns:
            Normalized complex tensor representing initial quantum field
        """
        D = self.config.field.dimensions
        state = torch.randn(D, dtype=self.dtype, device=self.device)
        # Normalize by square root of dimensionality (quantum normalization)
        return state / torch.sqrt(torch.sum(torch.abs(state)**2))
```

```python
def _initialize_reference_states(self, num_states: int = 9):
    """
    Initialize reference field states for coherence calculation.

    Args:
        num_states: Number of reference states to generate
    """
    D = self.config.field.dimensions
    self.reference_states = [
        torch.randn(D, dtype=self.dtype, device=self.device) / np.sqrt(D)
        for _ in range(num_states)
    ]

async def process_quantum(self, input_data: torch.Tensor) -> torch.Tensor:
    """
    Process data through the quantum-inspired system.

    This is the main processing function that implements the six-phase
    cycle of the DARKBOT™ architecture.

    Args:
        input_data: Input tensor to process

    Returns:
        Processed field state after complete cycle
    """
    # Ensure input is on correct device and has correct dtype
    if not isinstance(input_data, torch.Tensor):
        input_data = torch.tensor(input_data, dtype=self.dtype, device=self.device)
    elif input_data.device != self.device or input_data.dtype != self.dtype:
        input_data = input_data.to(device=self.device, dtype=self.dtype)

    # Execute on separate thread if CPU-bound operations
    if self.device.type == "cpu" and input_data.size(0) > self.config.field.dimensions // 2:
        import concurrent.futures
        with concurrent.futures.ThreadPoolExecutor() as executor:
            future = executor.submit(self._process_quantum_core, input_data)
            return await asyncio.wrap_future(future)
    else:
        # Direct execution for GPU or small inputs
        return self._process_quantum_core(input_data)

def _process_quantum_core(self, input_data: torch.Tensor) -> torch.Tensor:
    """
    Core implementation of quantum processing cycle.
```

```
    Args:
        input_data: Input tensor to process

    Returns:
        Processed field state after complete cycle
    """
    # 1. Field Identity Core (369)
    field = self._initialize_field(input_data)

    # 2. Branch Vector Phase (248)
    branches = self._branch_vectors(field)

    # 3. Parallel Field Resonance (157)
    resonant_data = self._proximity_resonance(branches)

    # 4. Self-Gravitational Memory (369)
    attractors = self._form_attractors(resonant_data)

    # 5. Funnel Vector Phase (248)
    converged = self._funnel_vectors(attractors)

    # 6. Fractal Entanglement
    entangled = self._fractal_entanglement(converged)

    # Calculate and store coherence
    coherence = calculate_coherence(
        entangled,
        self.reference_states,
        device=self.device
    )
    self.coherence_history.append(coherence)

    # Update the quantum state
    self.kvantum_tilstand = entangled

    # Advance phase
    self.current_phase = (self.current_phase + 1) % 6

    return entangled

def _initialize_field(self, x: torch.Tensor) -> torch.Tensor:
    """
    Initialize field with input data (Phase 1: Field Identity Core).

    Args:
        x: Input tensor
```

```
    Returns:
        Initialized field state
    """
    # Embed input into high-dimensional space
    D = self.config.field.dimensions
    embedded = torch.zeros(D, dtype=self.dtype, device=self.device)

    # Copy input data (or pad/truncate as needed)
    input_size = min(x.size(0), D)
    embedded[:input_size] = x[:input_size]

    # Apply 369 numerological operator
    field = self._apply_369_operator(embedded)

    # Normalize the field
    return field / torch.sqrt(torch.sum(torch.abs(field)**2))

def _apply_369_operator(self, field: torch.Tensor) -> torch.Tensor:
    """
    Apply the 369 numerological operator (Generative Field Dynamics).

    Args:
        field: Input field state

    Returns:
        Field processed through 369 operator
    """
    # 3: Initialization function (splits into triplet)
    past = field * torch.exp(torch.tensor(-1j * np.pi/3, device=self.device))
    present = field.clone()
    future = field * torch.exp(torch.tensor(1j * np.pi/3, device=self.device))
    init_component = (past + present + future) / 3

    # 6: Harmonic coupling (2×3, pairs dimensions)
    coupled = torch.zeros_like(field)
    half_dim = field.size(0) // 2
    for i in range(half_dim):
        coupled[i] = field[i] * field[i + half_dim]
        coupled[i + half_dim] = field[i] * field[i + half_dim]
    coupled = coupled / 6

    # 9: Completion/synthesis (3×3, self-referential closure)
    synth = torch.zeros_like(field)
    third_dim = field.size(0) // 3
    for i in range(third_dim):
        # Create 3×3 relationships between dimensions
        for j in range(3):
```

```python
                for k in range(3):
                    idx = i + j*third_dim + k*(third_dim//3)
                    if idx < field.size(0):
                        synth[i] += field[idx] / 9

        # Apply weights from configuration
        weights = self.config.numerology.WEIGHTS_369
        return weights[0] * init_component + weights[1] * coupled + weights[2] * synth

    def _branch_vectors(self, field: torch.Tensor) -> Dict[str, torch.Tensor]:
        """
        Expand field through branching vector operations (Phase 2).

        Args:
            field: Input field state

        Returns:
            Dictionary of branch vectors
        """
        branches = {}
        n = 4  # Branching factor (2^4 = 16 branches)

        for i in range(2**n):
            # Apply quaternion rotation with angle θ_i
            theta = 2 * np.pi * i / (2**n)
            # Calculate scale factor
            scale = 0.5 + 0.1 * (i % 8)
            # Apply transformation
            branches[f"branch_{i}"] = quaternion_transform(
                field,
                angle=theta,
                scale=scale,
                device=self.device
            )

        return branches

    def _proximity_resonance(self, branches: Dict[str, torch.Tensor]) -> Dict[str,
    torch.Tensor]:
        """
        Process branches through parallel proximity resonance (Phase 3).

        Args:
            branches: Dictionary of branch vectors

        Returns:
            Dictionary of resonant processed vectors
```

```
        """
        # Create 5 reference points in pentagonal arrangement (157 -> 5)
        references = self._create_pentagonal_references()

        results = {}
        # Process through 7 phase angles (157 -> 7 phases)
        for phase in range(7):
            phi = 2 * np.pi * phase / 7
            for key, branch in branches.items():
                # Calculate resonance with all reference points
                resonance = 0.5
                for j, ref in enumerate(references, 1):
                    eta = calculate_resonance(branch, ref, device=self.device) *
    torch.cos(torch.tensor(phi * j, device=self.device))
                    resonance += eta

                # Apply resonance transformation
                results[f"{key}_phase_{phase}"] = branch * resonance

        return results

    def _create_pentagonal_references(self) -> List[torch.Tensor]:
        """
        Create pentagonal reference points for the 157 architecture.

        Returns:
            List of 5 reference tensors in pentagonal arrangement
        """
        # Use the quantum state as base reference
        base = self.kvantum_tilstand

        # Create 5 references with phase shifts
        references = [base.clone()]
        for i in range(4):
            angle = 2 * np.pi * (i + 1) / 5
            phase_shift = torch.exp(torch.tensor(1j * angle, device=self.device))
            references.append(base * phase_shift)

        return references

    def _form_attractors(self, resonance_data: Dict[str, torch.Tensor]) -> List[torch.Tensor]:
        """
        Form attractor nodes through gravitational memory (Phase 4).

        Args:
            resonance_data: Dictionary of resonant processed vectors
```

```
    Returns:
        List of attractor nodes
    """
    # Initialize with triadic structure (3 attractors - 369 -> 3)
    attractors = [
        torch.zeros(self.config.field.dimensions, dtype=self.dtype, device=self.device)
        for _ in range(3)
    ]

    # Assign resonance points to attractors
    for i, tensor in enumerate(resonance_data.values()):
        attractor_idx = i % 3  # Modulo 3 assignment
        # Calculate coherence with attractor
        coherence = calculate_coherence(
            tensor,
            [attractors[attractor_idx]] if torch.sum(torch.abs(attractors[attractor_idx])) >
0 else self.reference_states,
            device=self.device
        )
        # Update attractor based on coherence
        attractors[attractor_idx] += tensor * coherence

    # Cross-attractor influence (369 -> 9 = 3×3)
    beta = 0.1  # Cross-attractor influence factor
    for i in range(3):
        for j in range(3):
            if i != j:
                # Calculate cross-attractor coherence
                cross_coherence = calculate_coherence(
                    attractors[i],
                    [attractors[j]] if torch.sum(torch.abs(attractors[j])) > 0 else
self.reference_states,
                    device=self.device
                )
                # Apply influence
                attractors[i] += attractors[j] * (cross_coherence * beta)

    return attractors

def _funnel_vectors(self, attractors: List[torch.Tensor]) -> torch.Tensor:
    """
    Converge attractors through funnel vector operations (Phase 5).

    Args:
        attractors: List of attractor nodes

    Returns:
```

```python
            Converged field state
        """
        # Binary reduction (248 -> 2)
        pairs = []
        for i in range(0, len(attractors), 2):
            if i+1 < len(attractors):
                pairs.append(attractors[i] + attractors[i+1])
            else:
                pairs.append(attractors[i])

        # Quaternion alignment (248 -> 4)
        aligned = []
        for pair in pairs:
            for rotation in range(4):
                theta = 2 * np.pi * rotation / 4
                # Apply rotation
                aligned.append(quaternion_transform(
                    pair,
                    angle=theta,
                    scale=1.0,
                    device=self.device
                ))

        # Octonion integration (248 -> 8)
        result = torch.zeros_like(attractors[0])
        for i, tensor in enumerate(aligned):
            # Apply octonion weights
            weight = (1 + 0.2 * (i % 8)) / 8
            result += tensor * weight

        return result

    def _fractal_entanglement(self, result: torch.Tensor) -> torch.Tensor:
        """
        Apply fractal entanglement through recursive self-simulation (Phase 6).

        Args:
            result: Input field state

        Returns:
            Entangled field state
        """
        # Initialize with result
        entangled = result.clone()
        alpha = 0.6  # Memory retention factor
        gamma = self.config.resonance.gamma  # Fractal scaling factor
```

```python
    # Apply for three levels of recursion
    for level in range(self.config.field.recursion_depth):
        # Self-similar transformation with scale factor
        transformed = fractal_transform(
            entangled,
            level,
            device=self.device
        ) * (gamma ** level)

        # Update with transformed result
        entangled = alpha * entangled + (1-alpha) * transformed

        # Store in fractal memory
        self.fractal_memory.append(entangled.clone())
        # Limit memory size
        if len(self.fractal_memory) > 7:  # 157 -> 7
            self.fractal_memory.pop(0)

    return entangled

def one_draw_search(self, query: torch.Tensor, target_space: List[torch.Tensor]) ->
Dict[str, Any]:
    """
    Perform One Draw search to find best match in target space.

    Args:
        query: Query field state
        target_space: List of target field states to search

    Returns:
        Dictionary with match results including index, confidence and resonance profile
    """
    # Ensure query is in correct format
    if not isinstance(query, torch.Tensor):
        query = torch.tensor(query, dtype=self.dtype, device=self.device)
    elif query.device != self.device or query.dtype != self.dtype:
        query = query.to(device=self.device, dtype=self.dtype)

    # Ensure targets are in correct format
    targets = []
    for target in target_space:
        if not isinstance(target, torch.Tensor):
            target = torch.tensor(target, dtype=self.dtype, device=self.device)
        elif target.device != self.device or target.dtype != self.dtype:
            target = target.to(device=self.device, dtype=self.dtype)
        targets.append(target)
```

```python
        # Embed query in field space
        query_field = self._initialize_field(query)

        # Calculate resonance with all targets
        resonances = []
        for target in targets:
            target_field = self._initialize_field(target)
            res = calculate_resonance(query_field, target_field, device=self.device)
            resonances.append(res.item())

        # Apply slot operator to find best match
        max_idx = int(torch.argmax(torch.tensor(resonances, device=self.device)))

        # Apply harmonic amplification
        h_phi = self._harmonic_amplification()
        amplified_resonance = resonances[max_idx] * h_phi.real

        # Return best match and confidence
        return {
            'match_index': max_idx,
            'target': targets[max_idx],
            'confidence': amplified_resonance,
            'resonance_profile': resonances
        }

    def _harmonic_amplification(self, phi=None):
        """
        Calculate harmonic amplification factor.

        Args:
            phi: Phase value (defaults to golden ratio)

        Returns:
            Complex amplification factor
        """
        if phi is None:
            phi = self.config.field.phi  # Use golden ratio

        # Apply complex phase rotation
        return torch.exp(torch.tensor(1j * phi * np.pi / 2, device=self.device))

    def calculate_coherence(self, field: torch.Tensor) -> float:
        """
        Calculate coherence of a field state.

        Args:
            field: Field state to measure
```

```
    Returns:
        Coherence value in range [0, 1]
    """
    return calculate_coherence(field, self.reference_states, device=self.device)

def get_field_awareness(self) -> float:
    """
    Calculate the current field awareness level.

    Returns:
        Awareness level in range [0, 1]
    """
    # Sample points across the field
    awareness = 0.0
    samples = 100  # Number of sample points

    for _ in range(samples):
        # Generate random coordinates
        x = torch.randn(1, device=self.device)
        y = torch.randn(1, device=self.device)
        z = torch.randn(1, device=self.device)

        # Calculate coherence at this point
        point = torch.cat([x, y, z])
        embedded = self._embed_point(point)
        awareness += self.calculate_coherence(embedded) / samples

    return awareness

def _embed_point(self, point: torch.Tensor) -> torch.Tensor:
    """
    Embed a 3D point into the full field space.

    Args:
        point: 3D spatial point

    Returns:
        Embedded field representation
    """
    D = self.config.field.dimensions
    embedded = torch.zeros(D, dtype=self.dtype, device=self.device)

    # Simple embedding: replicate point coordinates throughout the field
    for i in range(D // 3):
        embedded[i*3:(i+1)*3] = point
```

```python
        # Add phase variation
        for i in range(D):
            phase = 2 * np.pi * i / D
            embedded[i] *= torch.exp(torch.tensor(1j * phase, device=self.device))

        return embedded / torch.sqrt(torch.sum(torch.abs(embedded)**2))

    async def process_quantum_batch(self, input_batch: torch.Tensor) -> torch.Tensor:
        """
        Process a batch of inputs through the quantum-inspired system.

        Args:
            input_batch: Batch of input tensors [B, D]

        Returns:
            Batch of processed field states [B, D]
        """
        # Convert inputs to tensor if needed
        if not isinstance(input_batch, torch.Tensor):
            input_batch = torch.stack([
                torch.tensor(x, dtype=self.dtype, device=self.device)
                for x in input_batch
            ])
        elif input_batch.device != self.device or input_batch.dtype != self.dtype:
            input_batch = input_batch.to(device=self.device, dtype=self.dtype)

        # Ensure batch dimension
        if input_batch.dim() == 1:
            input_batch = input_batch.unsqueeze(0)

        # Execute on separate thread if CPU-bound operations
        if self.device.type == "cpu" and input_batch.size(0) > 4:
            import concurrent.futures
            with concurrent.futures.ThreadPoolExecutor() as executor:
                future = executor.submit(self._process_batch_core, input_batch)
                return await asyncio.wrap_future(future)
        else:
            # Direct execution for GPU or small batches
            return self._process_batch_core(input_batch)

    def _process_batch_core(self, input_batch: torch.Tensor) -> torch.Tensor:
        """
        Core implementation of batch processing.

        Args:
            input_batch: Batch of input tensors [B, D]
```

```
        Returns:
            Batch of processed field states [B, D]
        """
        batch_size = input_batch.shape[0]
        results = []

        # Process each item through the workflow
        # Note: A full implementation would vectorize all operations
        for i in range(batch_size):
            result = self._process_quantum_core(input_batch[i])
            results.append(result)

        return torch.stack(results)

    def benchmark_system(self, iterations=100, dim=512, batch_size=16, export_results=True):
        """
        Benchmark DARKBOT™ system performance.

        Args:
            iterations: Number of iterations for each test
            dim: Dimension size to use for testing
            batch_size: Batch size for batch processing tests
            export_results: Whether to export results to file

        Returns:
            Dictionary of benchmark results
        """
        import time

        results = []

        # Generate test data
        test_data = torch.randn(batch_size, dim, dtype=self.dtype, device=self.device)

        # Warm-up run
        _ = self._process_quantum_core(test_data[0])

        # Ensure GPU operations complete
        if self.device.type == "cuda":
            torch.cuda.synchronize()

        # Test One Draw search
        start_time = time.time()
        for _ in range(iterations):
            query = torch.randn(dim, dtype=self.dtype, device=self.device)
            _ = self.one_draw_search(query, [test_data[i] for i in range(batch_size)])
            if self.device.type == "cuda":
```

```
        torch.cuda.synchronize()
one_draw_time = (time.time() - start_time) / iterations

# Test classical search
start_time = time.time()
for _ in range(iterations):
    query = torch.randn(dim, dtype=self.dtype, device=self.device)
    distances = [torch.norm(query - test_data[i]) for i in range(batch_size)]
    _ = np.argmin(distances)
    if self.device.type == "cuda":
        torch.cuda.synchronize()
classical_time = (time.time() - start_time) / iterations

# Test full processing cycle
start_time = time.time()
for _ in range(max(1, iterations // 10)):  # Fewer iterations as this is slower
    _ = self._process_quantum_core(test_data[0])
    if self.device.type == "cuda":
        torch.cuda.synchronize()
process_time = (time.time() - start_time) / (max(1, iterations // 10))

# Calculate speedup
speedup = classical_time / one_draw_time

# Compile results
benchmark_results = {
    "one_draw_time_ms": one_draw_time * 1000,
    "classical_time_ms": classical_time * 1000,
    "process_time_ms": process_time * 1000,
    "speedup_factor": speedup,
    "device": str(self.device),
    "dimensions": dim,
    "batch_size": batch_size,
    "timestamp": datetime.now().isoformat()
}

# Export results if requested
if export_results:
    # Create benchmark directory if it doesn't exist
    benchmark_dir = Path("benchmark_results")
    benchmark_dir.mkdir(exist_ok=True)

    # Save as JSON
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filename = benchmark_dir / f"darkbot_benchmark_{timestamp}.json"

    with open(filename, 'w') as f:
```

```python
            json.dump(benchmark_results, f, indent=2, default=str)

        print(f"Benchmark results saved to {filename}")

    return benchmark_results

def save_config(self, filepath: str):
    """
    Save the current configuration to file.

    Args:
        filepath: Path to save the configuration
    """
    self.config.to_yaml(filepath)

@classmethod
def load_config(cls, filepath: str):
    """
    Load configuration from file and create a DarkBot instance.

    Args:
        filepath: Path to load the configuration from

    Returns:
        DarkBot instance with loaded configuration
    """
    config = DarkBotConfig.from_yaml(filepath)
    return cls(config)

def __repr__(self):
    """String representation of the DarkBot instance."""
    return f"DarkBot(dimensions={self.config.field.dimensions}, device={self.device})"
```