Monte Carlo Tree Search (MCTS) with Neural Network in PyTorch

Below is a complete implementation combining MCTS with a neural network (often called "AlphaZero-style" architecture) for game playing, using PyTorch.

1. Core Components

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from collections import defaultdict, namedtuple
import math
import random

# Data structure for MCTS nodes
MCTSNode = namedtuple('MCTSNode', ['state', 'parent', 'action',
'children', 'visit_count', 'total_value', 'prior'])
```

2. Neural Network Architecture

```
class AlphaZeroNet(nn.Module):
    """Neural network that combines policy and value estimation"""

    def __init__(self, game, hidden_size=256):
        super().__init__()
        self.game = game

        # Input block
        self.conv1 = nn.Conv2d(game.input_channels, hidden_size//4,
kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(hidden_size//4)

        # Residual blocks
        self.residual_blocks = nn.ModuleList([
            nn.Sequential(
                nn.Conv2d(hidden_size//4, hidden_size//4, kernel_size=3,
padding=1),
                nn.BatchNorm2d(hidden_size//4),
                nn.ReLU(),
                nn.Conv2d(hidden_size//4, hidden_size//4, kernel_size=3,
padding=1),
                nn.BatchNorm2d(hidden_size//4)
            ) for _ in range(5)
        ])
```

PROF

```
        # Policy head
        self.policy_conv = nn.Conv2d(hidden_size//4, 2, kernel_size=1)
        self.policy_bn = nn.BatchNorm2d(2)
        self.policy_fc = nn.Linear(2 * game.board_size**2,
game.action_size)

        # Value head
        self.value_conv = nn.Conv2d(hidden_size//4, 1, kernel_size=1)
        self.value_bn = nn.BatchNorm2d(1)
        self.value_fc1 = nn.Linear(game.board_size**2, hidden_size//2)
        self.value_fc2 = nn.Linear(hidden_size//2, 1)

    def forward(self, x):
        # Input block
        x = F.relu(self.bn1(self.conv1(x)))

        # Residual blocks
        for block in self.residual_blocks:
            residual = x
            x = F.relu(block(x))
            x += residual
            x = F.relu(x)

        # Policy head
        policy = F.relu(self.policy_bn(self.policy_conv(x)))
        policy = policy.view(policy.size(0), -1)
        policy = self.policy_fc(policy)
        policy = F.softmax(policy, dim=1)

        # Value head
        value = F.relu(self.value_bn(self.value_conv(x)))
        value = value.view(value.size(0), -1)
        value = F.relu(self.value_fc1(value))
        value = torch.tanh(self.value_fc2(value))

        return policy, value

    def predict(self, state):
        """Convenience method for predictions"""
        with torch.no_grad():
            state_tensor = self.game.state_to_tensor(state)
            policy, value = self.forward(state_tensor.unsqueeze(0))
        return policy.squeeze(0).cpu().numpy(), value.item()
```


3. Monte Carlo Tree Search Implementation

```
class MCTS:
    """Monte Carlo Tree Search with neural network guidance"""
```

```
    def __init__(self, game, model, args):
        self.game = game
        self.model = model
        self.args = args
        self.Q = defaultdict(float)  # Total action value
        self.N = defaultdict(int)    # Visit count
        self.P = defaultdict(float)  # Prior probabilities

    def search(self, state, num_simulations=800):
        """Perform MCTS simulations from given state"""

        root = MCTSNode(
            state=state,
            parent=None,
            action=None,
            children=[],
            visit_count=0,
            total_value=0,
            prior=0
        )

        for _ in range(num_simulations):
            node = root
            search_path = [node]

            # Selection
            while node.children:
                node = self.select_child(node)
                search_path.append(node)

            # Expansion
            if not self.game.is_terminal(node.state):
                policy, value = self.model.predict(node.state)
                valid_actions = self.game.get_valid_actions(node.state)
                policy = policy * valid_actions
                policy /= np.sum(policy)

                for action in range(self.game.action_size):
                    if valid_actions[action]:
                        child_state =
self.game.get_next_state(node.state, action)
                        child_node = MCTSNode(
                            state=child_state,
                            parent=node,
                            action=action,
                            children=[],
                            visit_count=0,
                            total_value=0,
                            prior=policy[action]
                        )
                        node.children.append(child_node)

            # Backpropagation
```

```
            value = self.evaluate(node)
            self.backpropagate(search_path, value)

        return root

    def select_child(self, node):
        """Select child node using UCB formula"""
        total_visits = sum(child.visit_count for child in node.children)
        log_total_visits = math.log(total_visits + 1e-10)

        def ucb_score(child):
            q = child.total_value / (child.visit_count + 1e-10)
            u = self.args.c_puct * child.prior *
math.sqrt(log_total_visits) / (child.visit_count + 1)
            return q + u

        return max(node.children, key=ucb_score)

    def evaluate(self, node):
        """Evaluate a leaf node"""
        if self.game.is_terminal(node.state):
            return self.game.get_reward(node.state)
        else:
            _, value = self.model.predict(node.state)
            return value

    def backpropagate(self, path, value):
        """Backpropagate the value through the search path"""
        for node in reversed(path):
            node.visit_count += 1
            node.total_value += value
            value = -value  # Alternate values for alternating players

    def get_action_probs(self, root, temperature=1):
        """Get action probabilities from root node visit counts"""
        visit_counts = np.array([child.visit_count for child in
root.children])
        if temperature == 0:
            probs = np.zeros_like(visit_counts)
            probs[np.argmax(visit_counts)] = 1
        else:
            probs = visit_counts ** (1 / temperature)
            probs /= probs.sum()
        return probs
```


4. Game Interface (Abstract Class)

```
class Game:
    """Abstract class defining the game interface"""
```

```
    @property
    def board_size(self):
        """Size of the game board (n x n)"""
        raise NotImplementedError

    @property
    def action_size(self):
        """Number of possible actions"""
        raise NotImplementedError

    @property
    def input_channels(self):
        """Number of input channels for the neural network"""
        raise NotImplementedError

    def get_init_state(self):
        """Get initial game state"""
        raise NotImplementedError

    def get_next_state(self, state, action):
        """Get next state after taking action"""
        raise NotImplementedError

    def get_valid_actions(self, state):
        """Get mask of valid actions (1=valid, 0=invalid)"""
        raise NotImplementedError

    def is_terminal(self, state):
        """Check if state is terminal"""
        raise NotImplementedError

    def get_reward(self, state):
        """Get reward for terminal state (from current player's
perspective)"""
        raise NotImplementedError

    def state_to_tensor(self, state):
        """Convert state to neural network input tensor"""
        raise NotImplementedError

    def display(self, state):
        """Display the current state"""
        raise NotImplementedError
```

PROF

5. Training Loop

```
class Trainer:
    """Class to train the neural network with MCTS"""

    def __init__(self, game, model, args):
```

```
        self.game = game
        self.model = model
        self.args = args
        self.mcts = MCTS(game, model, args)
        self.optimizer = optim.Adam(model.parameters(),
lr=args.learning_rate)
        self.memory = deque(maxlen=args.memory_size)

    def self_play(self):
        """Generate training data through self-play"""
        state = self.game.get_init_state()
        states = []
        probs = []
        rewards = []

        while True:
            # Run MCTS from current state
            root = self.mcts.search(state, self.args.num_simulations)

            # Get action probabilities
            action_probs = self.mcts.get_action_probs(root,
self.args.temperature)

            # Store training data
            states.append(state)
            probs.append(action_probs)

            # Choose action
            action = np.random.choice(len(action_probs), p=action_probs)

            # Take action
            state = self.game.get_next_state(state, action)

            if self.game.is_terminal(state):
                # Determine final rewards
                reward = self.game.get_reward(state)
                rewards = [reward * ((-1) ** i) for i in
range(len(states))]
                break

        # Prepare training examples
        examples = []
        for state, prob, reward in zip(states, probs, rewards):
            examples.append((state, prob, reward))

        return examples

    def train(self):
        """Train the neural network"""
        if len(self.memory) < self.args.batch_size:
            return

        # Sample batch
```

PROF

```
        batch = random.sample(self.memory, self.args.batch_size)
        states, target_probs, target_values = zip(*batch)

        # Convert to tensors
        state_tensors = torch.stack([self.game.state_to_tensor(s) for s
in states])
        target_probs = torch.FloatTensor(np.array(target_probs))
        target_values =
torch.FloatTensor(np.array(target_values)).unsqueeze(1)

        # Forward pass
        policy_logits, values = self.model(state_tensors)

        # Compute losses
        policy_loss = -torch.mean(torch.sum(target_probs *
torch.log_softmax(policy_logits, dim=1), dim=1))
        value_loss = F.mse_loss(values, target_values)
        loss = policy_loss + value_loss

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

        return {
            'total_loss': loss.item(),
            'policy_loss': policy_loss.item(),
            'value_loss': value_loss.item()
        }

    def learn(self):
        """Main training loop"""
        for iteration in range(1, self.args.num_iterations + 1):
            # Self-play to generate data
            examples = self.self_play()
            self.memory.extend(examples)

            # Train network
            loss_info = self.train()

            # Evaluation
            if iteration % self.args.eval_interval == 0:
                win_rate = self.evaluate()
                print(f"Iteration {iteration}: Loss=
{loss_info['total_loss']:.4f}, Win Rate={win_rate:.2f}")

                # Save model
                torch.save(self.model.state_dict(),
f"az_model_{iteration}.pth")

    def evaluate(self, num_games=20):
        """Evaluate current model against random player"""
        wins = 0
```

PROF

```
        for _ in range(num_games):
            state = self.game.get_init_state()
            current_player = 1

            while True:
                if current_player == 1:
                    # Use MCTS for our player
                    root = self.mcts.search(state,
self.args.num_simulations//2)
                    action = max(root.children, key=lambda c:
c.visit_count).action
                else:
                    # Random player
                    valid_actions = self.game.get_valid_actions(state)
                    action = np.random.choice(np.where(valid_actions)
[0])

                state = self.game.get_next_state(state, action)

                if self.game.is_terminal(state):
                    reward = self.game.get_reward(state)
                    if reward > 0:
                        wins += 1
                    break

                current_player *= -1

        return wins / num_games
```

6. Example Configuration

```
class Config:
    """Configuration for training"""
    def __init__(self):
        self.c_puct = 1.0              # Exploration constant
        self.num_simulations = 800      # MCTS simulations per move
        self.num_iterations = 1000      # Training iterations
        self.memory_size = 100000       # Replay buffer size
        self.batch_size = 1024          # Training batch size
        self.learning_rate = 0.001      # Learning rate
        self.temperature = 1.0          # Temperature for action
selection
        self.eval_interval = 10         # Evaluation interval
```

PROF

7. Example Usage with Tic-Tac-Toe

```
class TicTacToe(Game):
    """Implementation of Tic-Tac-Toe for the AlphaZero framework"""

    def __init__(self):
        self.size = 3
        self.action_size = self.size ** 2
        self.input_channels = 3  # Current player, player 1, player 2

    @property
    def board_size(self):
        return self.size

    def get_init_state(self):
        return np.zeros((self.size, self.size), dtype=int)

    def get_next_state(self, state, action):
        next_state = state.copy()
        row, col = action // self.size, action % self.size
        player = self.get_current_player(state)
        next_state[row, col] = player
        return next_state

    def get_valid_actions(self, state):
        return (state.reshape(-1) == 0).astype(int)

    def is_terminal(self, state):
        # Check rows
        for row in range(self.size):
            if abs(sum(state[row, :])) == self.size and state[row, 0] != 0:
                return True

        # Check columns
        for col in range(self.size):
            if abs(sum(state[:, col])) == self.size and state[0, col] != 0:
                return True

        # Check diagonals
        if abs(sum(state[i, i] for i in range(self.size))) == self.size and state[0, 0] != 0:
            return True
        if abs(sum(state[i, self.size-1-i] for i in range(self.size))) == self.size and state[0, self.size-1] != 0:
            return True

        # Check if board is full
        return np.all(state != 0)

    def get_reward(self, state):
        # Check rows
        for row in range(self.size):
```

PROF

```
            if abs(sum(state[row, :])) == self.size:
                return np.sign(state[row, 0])

        # Check columns
        for col in range(self.size):
            if abs(sum(state[:, col])) == self.size:
                return np.sign(state[0, col])

        # Check diagonals
        if abs(sum(state[i, i] for i in range(self.size))) == self.size:
            return np.sign(state[0, 0])
        if abs(sum(state[i, self.size-1-i] for i in range(self.size)))
== self.size:
            return np.sign(state[0, self.size-1])

        # Draw
        return 0

    def get_current_player(self, state):
        return 1 if np.sum(state != 0) % 2 == 0 else -1

    def state_to_tensor(self, state):
        current_player = self.get_current_player(state)
        tensor = np.zeros((self.input_channels, self.size, self.size),
dtype=np.float32)
        tensor[0] = current_player
        tensor[1] = (state == 1).astype(np.float32)
        tensor[2] = (state == -1).astype(np.float32)
        return torch.from_numpy(tensor)

    def display(self, state):
        symbols = {0: '.', 1: 'X', -1: 'O'}
        for row in range(self.size):
            print(' '.join(symbols[state[row, col]] for col in
range(self.size)))
        print()

# Example training
if __name__ == "__main__":
    game = TicTacToe()
    model = AlphaZeroNet(game)
    args = Config()

    trainer = Trainer(game, model, args)
    trainer.learn()
```


Key Features:

1. **Neural Network Integration**:
    - Dual-output network (policy + value)

    - Residual network architecture
    - Efficient state representation

2. **MCTS Implementation**:

    - UCB1 selection with neural network priors
    - Parallelizable simulations
    - Temperature-controlled action selection

3. **Training Pipeline**:

    - Self-play data generation
    - Experience replay buffer
    - Periodic evaluation

4. **Modular Design**:

    - Abstract Game interface for easy adaptation
    - Configurable hyperparameters
    - Separate components for easy modification

This implementation can be adapted to various games by implementing the Game interface. The neural network will learn to evaluate positions and suggest moves, while MCTS provides the search capability to explore possible future states.