基于视觉+三因素融合的五子棋AI架构技术文档

1. 核心理念与架构概述

本架构提出了一种基于视觉与三因素融合的五子棋AI模型，不依赖于传统的蒙特卡洛树搜索(MCTS)，而是完全由深度神经网络实现。三个核心因素包括：

1. **滑向机制**：在纯视觉基础上使AI更容易在同色周围落子(进攻)或异色周围落子(防守)
2. **策略思考**：将落子序列编码控制落子策略(进攻、防守、布局)
3. **急迫度**：让更短的落子序列(达到赢局)获得更高奖励

整体架构形成感知-行动循环系统，从视觉输入到决策输出构成完整的端到端流程。

2. 视觉基础层

视觉基础层采用深度残差网络，将棋盘状态表示为三通道输入：

```
输入维度: [batch_size, 3, board_size, board_size]
- channel 0: 当前玩家棋子位置 (1表示有棋子，0表示无)
- channel 1: 对手棋子位置 (1表示有棋子，0表示无)
- channel 2: 可行动位置 (1表示可落子，0表示不可落子)
```

残差块公式如下：

```
残差块(x) = ReLU(BN(Conv(ReLU(BN(Conv(x))))) + x)
```

其中：

- Conv: 3×3卷积
- BN: 批量归一化
- ReLU: 修正线性单元激活函数

3. 方向信息场 (滑向机制)

3.1 理论公式

方向信息场(Information Field)数学公式：

$$F_{i,j} = \sum_{d \in D} \phi(S_d) \cdot \omega_d(i,j)$$

其中：

- $F_{i,j}$ 是位置(i,j)的信息场强度
- D 是四个关键方向集合 {水平、垂直、主对角线、副对角线}
- $\phi(S_d)$ 是方向d上的棋型强度函数
- $\omega_d(i,j)$ 是位置(i,j)在方向d上的权重

3.2 实现算法

```
class DirectionalFieldModule(nn.Module):
    def __init__(self, channels):
        # 四个方向的卷积
        self.horizontal_conv = nn.Conv2d(channels, channels//4, kernel_size=(1, 5), padding=(0,
        self.vertical_conv = nn.Conv2d(channels, channels//4, kernel_size=(5, 1), padding=(2, 0
        self.diag1_conv = nn.Conv2d(channels, channels//4, kernel_size=5, padding=2)
        self.diag2_conv = nn.Conv2d(channels, channels//4, kernel_size=5, padding=2)

        # 方向权重（可学习）
        self.direction_weights = nn.Parameter(torch.ones(4) / 4)

    def forward(self, x):
        # 计算四个方向的场
        h_field = self.horizontal_conv(x)
        v_field = self.vertical_conv(x)
        d1_field = self.diag1_conv(x)
        d2_field = self.diag2_conv(x)

        # 方向权重归一化
        dir_weights = F.softmax(self.direction_weights, dim=0)

        # 综合各方向场（带权重）
        combined_field = (h_field * dir_weights[0] +
                          v_field * dir_weights[1] +
                          d1_field * dir_weights[2] +
                          d2_field * dir_weights[3])

        return combined_field
```

4. 时间梯度 (急迫度机制)

4.1 理论公式

时间梯度(Temporal Gradient)的数学公式：

$$T_{i,j} = \sum_{t=0}^{H} \gamma^t [V(S_{t+1}^{i,j}) - V(S_t)]$$

其中：

- $T_{i,j}$ 是位置(i,j)的时间梯度
- H 是时间视野长度
- γ 是时间折扣因子(0至1之间)
- $V(S)$ 是状态S的价值函数
- $S_{t+1}^{i,j}$ 是在位置(i,j)落子后，经过t步可能达到的状态

4.2 实现算法

```
class TemporalGradientModule(nn.Module):
    def __init__(self, channels):
        # 当前状态价值估计器
        self.value_estimator = nn.Sequential(
            nn.Conv2d(channels, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 1, kernel_size=1)
        )

        # 模拟落子后状态价值估计器
        self.next_value_estimator = nn.Sequential(
            nn.Conv2d(channels, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 1, kernel_size=1)
        )

        # 可学习的时间折扣因子
        self.gamma = nn.Parameter(torch.tensor(0.9))

    def forward(self, x):
        # 估计当前状态价值 V(S_t)
        current_value = self.value_estimator(x)

        # 估计落子后状态价值 V(S_{t+1}^{i,j})
        next_value = self.next_value_estimator(x)

        # 计算时间梯度（价值增量）
        temporal_gradient = next_value - current_value

        # 应用可学习的时间折扣因子
        return torch.sigmoid(self.gamma) * temporal_gradient
```

5. 策略参数模块

5.1 理论公式

策略参数模块基于历史序列动态调整α和β参数:

$$(\alpha, \beta) = f_\theta(H_t)$$

其中:

- α: 信息场(滑向机制)的权重
- β: 时间梯度(急迫度)的权重
- H_t: 截至时间t的历史状态序列
- f_θ: 参数化策略网络

5.2 实现算法

```
class StrategicParameterModule(nn.Module):
    def __init__(self, channels, seq_length=8):
        # 位置编码层
        self.position_embedding = nn.Parameter(
            torch.zeros(1, seq_length, channels)
        )

        # 时序特征提取
        self.sequence_encoder = nn.GRU(
            input_size=channels,
            hidden_size=64,
            num_layers=1,
            batch_first=True
        )

        # 参数输出头
        self.param_head = nn.Sequential(
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 2)  # 输出两个策略参数
        )

    def forward(self, sequence_features):
        # 应用位置编码
        sequence_features = sequence_features + self.position_embedding[:, :sequence_features.s

        # 通过序列编码器
        _, hidden = self.sequence_encoder(sequence_features)
        hidden = hidden.squeeze(0)

        # 输出策略参数
        params = self.param_head(hidden)

        # 使用Sigmoid激活确保参数在[0,1]范围内
        alpha = torch.sigmoid(params[:, 0])
        beta = torch.sigmoid(params[:, 1])

        return alpha, beta
```

6. 统一决策框架

6.1 注意力整合公式

决策时将信息场和时间梯度通过注意力机制整合：

$$A_{i,j} = \frac{\exp(\alpha \cdot F_{i,j} + \beta \cdot T_{i,j})}{\sum_{k,l} \exp(\alpha \cdot F_{k,l} + \beta \cdot T_{k,l})}$$

其中：

- $A_{i,j}$ 是位置(i,j)的注意力权重
- α 和 β 是策略参数，控制滑向和急迫度的重要性
- 分母确保注意力是概率分布

6.2 实现算法

```
def forward(self, state, state_sequence=None, valid_moves_mask=None):
    # 1. 处理当前状态
    x = self.extract_features(state)

    # 2. 应用特征提取器
    enhanced_features = self.feature_extractor(x)

    # 3. 计算信息场 F_{i,j}
    information_field = self.information_field(enhanced_features)

    # 4. 计算时间梯度 T_{i,j}
    temporal_gradient = self.temporal_gradient(enhanced_features)

    # 5. 获取策略参数α和β
    if state_sequence is None:
        # 默认参数 - 等权重
        alpha = torch.ones(batch_size, device=device) * 0.5
        beta = torch.ones(batch_size, device=device) * 0.5
    else:
        # 使用策略参数模块计算
        alpha, beta = self.strategic_params(state_sequence)

    # 调整形状
    alpha = alpha.view(batch_size, 1, 1, 1)
    beta = beta.view(batch_size, 1, 1, 1)

    # 6. 整合注意力场
    attention_field = alpha * information_field + beta * temporal_gradient

    # 7. 应用softmax获得注意力权重
    attention = F.softmax(attention_field.view(batch_size, -1), dim=1).view_as(attention_field)

    # 8. 应用注意力到特征
    attended_features = enhanced_features * (1.0 + attention)

    # 9. 计算策略和价值
    policy_logits = self.policy_head(attended_features)
    value = self.value_head(attended_features)

    return policy_logits, value
```

7. 奖励扩散PPO训练算法

7.1 奖励扩散理论

奖励扩散(Reward Diffusion)公式：

$$R'_t = R_t + \sum_{h=1}^{H_f} \gamma_f^h R_{t+h} + \sum_{h=1}^{H_b} \gamma_b^h R_{t-h} \cdot 0.5$$

其中：

- R'_t 是时间t的扩散后奖励
- R_t 是时间t的原始奖励
- H_f 是前向扩散视野
- H_b 是后向扩散视野
- γ_f 是前向扩散率
- γ_b 是后向扩散率

7.2 PPO更新算法

PPO(近端策略优化)核心公式：

$$L^{CLIP}(\theta) = \mathbb{E}_t[\min(r_t(\theta) \cdot A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \cdot A_t)]$$

其中：

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ 是新旧策略的概率比
- A_t 是优势估计
- ϵ 是裁剪参数(通常为0.2)

7.3 实现算法

```
class RewardDiffuser:
    def diffuse_rewards(self, raw_rewards, strategy_params=None):
        T = len(raw_rewards)
        diffused_rewards = np.zeros_like(raw_rewards, dtype=np.float32)

        if strategy_params is None:
            # 使用固定扩散率
            diffusion_rates = [self.diffusion_rate] * T
            horizons = [self.horizon] * T
        else:
            # 根据策略参数动态调整扩散
            diffusion_rates = []
            horizons = []

            for alpha, beta in strategy_params:
                # beta越高(急迫度高)，扩散率越低，视野越短
                # alpha越高(信息场权重高)，扩散率越高
                rate = 0.7 + 0.2 * alpha - 0.1 * beta
                horizon = int(self.horizon * (1 - 0.5 * beta))

                diffusion_rates.append(rate)
                horizons.append(horizon)

        # 前向扩散（当前决策影响未来）
        for t in range(T):
            diffused_rewards[t] += raw_rewards[t]
            for h in range(1, min(horizons[t], T-t)):
                diffused_rewards[t] += raw_rewards[t+h] * (diffusion_rates[t] ** h)

        # 后向扩散（未来结果反馈当前）
        for t in reversed(range(T)):
            for h in range(1, min(horizons[t], t+1)):
                diffused_rewards[t-h] += raw_rewards[t] * (diffusion_rates[t-h] ** h) * 0.5

        return diffused_rewards

def update_policy(self, batch_size, num_updates, mini_batch_size, num_epochs):
    # 采样一批数据
    states_np, actions_np, rewards_np, _, _, infos = self.replay_buffer.sample(batch_size)

    # 提取优势和值估计
```

```
advantages_np = np.array([info["advantage"] for info in infos])
old_values_np = np.array([info["value"] for info in infos])

# 转换为张量并标准化优势
# ...

# 获取旧策略下的动作概率
with torch.no_grad():
    old_policy_logits, _ = self.model(states)
    old_policy = F.softmax(old_policy_logits, dim=1)
    old_action_probs = torch.gather(old_policy, 1, actions.unsqueeze(1))

# 创建DataLoader进行批处理
dataset = TensorDataset(states, actions, rewards, advantages, old_action_probs)

for _ in range(num_epochs):
    for mini_states, mini_actions, mini_rewards, mini_advantages, mini_old_probs in data_loa
        # 计算当前策略
        policy_logits, values = self.model(mini_states)
        policy = F.softmax(policy_logits, dim=1)
        action_probs = torch.gather(policy, 1, mini_actions.unsqueeze(1))

        # 计算概率比
        ratio = action_probs / (mini_old_probs + 1e-8)

        # 计算PPO裁剪损失
        surr1 = ratio * mini_advantages
        surr2 = torch.clamp(ratio, 1 - self.clip_ratio, 1 + self.clip_ratio) * mini_advantag
        policy_loss = -torch.min(surr1, surr2).mean()

        # 计算值函数损失
        value_loss = F.mse_loss(values, mini_rewards)

        # 计算熵奖励
        dist = torch.distributions.Categorical(policy)
        entropy = dist.entropy().mean()

        # 综合损失
        loss = policy_loss + self.value_coef * value_loss - self.entropy_coef * entropy

        # 优化更新
        # ...
```

8. 并行自对弈与批处理优化

8.1 并行自对弈算法

```
def collect_experience(self, num_episodes=10, num_workers=4):
    # 如果只有一个工作线程，使用串行处理
    if num_workers <= 1:
        return self._collect_experience_serial(num_episodes)

    # 准备模型状态字典以便在子进程中重建
    model_state = self.model.state_dict()
    best_model_state = self.best_model.state_dict()

    # 计算每个工作线程的任务数
    episodes_per_worker = [num_episodes // num_workers] * num_workers
    for i in range(num_episodes % num_workers):
        episodes_per_worker[i] += 1

    # 定义工作线程函数
    def worker_fn(episodes, model_state, best_model_state, board_size, device_str, worker_id):
        # 设置随机种子，确保每个工作线程有不同的随机数序列
        torch.manual_seed(42 + worker_id)
        np.random.seed(42 + worker_id)
        random.seed(42 + worker_id)

        # 重建模型和环境
        # ...

        # 自对弈收集经验
        experiences = []
        episode_stats = []

        for _ in range(episodes):
            # 进行自对弈，记录状态、动作、奖励等
            # ...

        return experiences, episode_stats

    # 使用多进程并行执行
    ctx = mp.get_context('spawn')
    with ctx.Pool(num_workers) as pool:
        # 启动所有工作线程
```

```
        results = []
        for i, episodes_count in enumerate(episodes_per_worker):
            results.append(pool.apply_async(worker_with_args, args=(episodes_count, i)))

        # 收集所有结果
        all_experiences = []
        all_stats = []

        for result in results:
            experiences, stats = result.get()
            all_experiences.extend(experiences)
            all_stats.extend(stats)

    # 处理收集到的经验
    # ...

    return total_collected
```

8.2 批处理训练优化

```
def update_policy(self, batch_size=128, num_updates=10, mini_batch_size=32, num_epochs=4):
    # ...

    # 创建数据集和加载器
    dataset = TensorDataset(states, actions, rewards, advantages, old_action_probs)

    # 如果mini_batch_size为0，使用整个batch
    actual_mini_batch_size = mini_batch_size if mini_batch_size > 0 else batch_size

    # 创建DataLoader
    data_loader = DataLoader(
        dataset,
        batch_size=actual_mini_batch_size,
        shuffle=True,
        pin_memory=True
    )

    # 对每个mini-batch进行多轮训练
    for _ in range(num_epochs):
        for mini_batch in data_loader:
            # 计算损失并更新
            # ...
```

9. 总结与优势

本架构通过融合视觉基础和三个关键因素（滑向机制、策略思考和急迫度）创建了一个完整的五子棋AI系统。其主要优势包括：

1. **纯神经网络实现**：不依赖传统搜索，端到端可学习
2. **统一的理论框架**：所有组件基于同一套数学公式，形成连贯体系
3. **自适应调整策略**：根据局势动态调整进攻、防守和布局权重
4. **人类棋手思维模拟**：滑向和急迫度模拟了人类棋手的直觉和决策过程
5. **高效训练**：通过奖励扩散PPO和并行自对弈加速训练过程

通过深度残差网络提取视觉特征，然后使用方向信息场、时间梯度和策略参数模块融合决策，形成了一个既有理论基础又有实用性的五子棋AI架构。