

# Reinforcement Learning

## Lecture 11: Modern Directions & Capstone: RLHF, DPO, MCTS, and AlphaZero

Taehoon Kim

Sogang University MIMIC Lab  
<https://mimic-lab.com>

Fall Semester 2025

# Today's Agenda

## Session Structure

- **Theory Focus:** RLHF pipeline, DPO objectives, MCTS foundations, AlphaZero strategy
- **Practice Focus:** Implementation walkthroughs, experiment deep dives, project planning

## Segment Highlights:

- Course integration and roadmap
- RLHF theory and data pipelines
- Direct Preference Optimization techniques

## Hands-on Emphasis:

- MCTS rollouts with PUCT
- AlphaZero-style self-play implementation
- RLHF training utilities and evaluation

# Learning Objectives & Prerequisites

By the end of this lecture, you will be able to:

- Understand RLHF pipeline and regularized policy optimization
- Implement Direct Preference Optimization (DPO) without reward models
- Build MCTS with neural network guidance (PUCT algorithm)
- Create AlphaZero self-play training loops
- Design integrated capstone projects combining modern RL methods

## Prerequisites

- Policy gradient methods (Lectures 8-10)
- Neural network training and PyTorch proficiency
- Understanding of tree search algorithms
- Familiarity with preference learning concepts

# Reinforcement Learning from Human Feedback (RLHF)

## The Challenge

How do we train RL agents to follow human preferences when reward functions are hard to specify?

### Traditional RL:

- Hand-crafted reward functions
- Often misaligned with human intent
- Reward hacking problems
- Difficult to specify for complex tasks

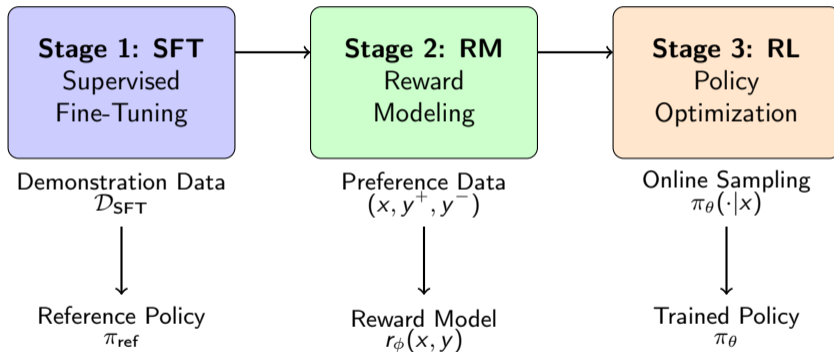
### RLHF Approach:

- Learn rewards from human preferences
- Three-stage pipeline: SFT  $\rightarrow$  RM  $\rightarrow$  RL
- Regularized policy optimization
- Scalable to complex domains

## Key Insight

Instead of engineering rewards, learn them from pairwise human comparisons of agent behaviors

# RLHF Three-Stage Pipeline



# RLHF Mathematical Formulation (Stages 1 & 2)

## Stage 1: Supervised Fine-Tuning

Train reference policy on demonstration data:

$$\pi_{\text{ref}} = \arg \min_{\pi} \mathbb{E}_{(x,y) \sim \mathcal{D}_{\text{SFT}}} [-\log \pi(y|x)]$$

## Stage 2: Reward Modeling

Learn rewards from preferences via the Bradley-Terry model:

$$P(y^+ \succ y^- | x) = \sigma(r_{\phi}(x, y^+) - r_{\phi}(x, y^-))$$

$$\phi^* = \arg \max_{\phi} \mathbb{E}_{(x,y^+,y^-)} \log \sigma(r_{\phi}(x, y^+) - r_{\phi}(x, y^-))$$

### Regularised Policy Optimisation

$$J(\theta) = \mathbb{E}_{x \sim \mathcal{D}, y \sim \pi_{\theta}} [r_{\phi}(x, y) - \beta D_{\text{KL}}(\pi_{\theta}(\cdot|x) \parallel \pi_{\text{ref}}(\cdot|x))]$$

where  $\beta > 0$  controls the KL penalty strength.

**Interpretation:** Encourage reward-seeking behaviour while staying close to the supervised reference policy.

# RLHF with PPO (Sampling & Reward)

## PPO-style RLHF Update

The KL-regularized reward becomes the “advantage” in PPO:

$$\hat{A}_t = r_\phi(x_t, y_t) - \beta \log \frac{\pi_\theta(y_t|x_t)}{\pi_{\text{ref}}(y_t|x_t)}$$

```
1 # RLHF PPO pseudo-code (sampling phase)
2 def rlhf_ppo_step(policy, ref_policy, reward_model, batch):
3     # Sample responses from current policy
4     responses = policy.sample(batch['prompts'])
5
6     # Compute rewards and KL penalty
7     rewards = reward_model(batch['prompts'], responses)
8     kl_penalty = beta * (policy.log_prob(responses)
9                        - ref_policy.log_prob(responses))
10    advantages = rewards - kl_penalty
11
```

**Implementation hook:** `exp08_dpo_implementation.py`

# RLHF with PPO (Policy Update)

```
1 # PPO clipped objective
2 ratio = policy.log_prob(responses) / old_policy.log_prob(responses)
3 clipped_ratio = torch.clamp(ratio, 1 - eps, 1 + eps)
4 loss = -torch.min(ratio * advantages,
5                  clipped_ratio * advantages).mean()
6
7 # Optimise policy
8 loss.backward()
9 optimizer.step()
```

**Key idea:** Treat the KL penalty as an advantage shaping term.

# Challenges with Traditional RLHF

## Computational Complexity

- Three separate training stages
- Reward model can be unstable or overfit
- Online RL training is sample inefficient
- Multiple model copies needed during training

## Optimization Issues

- Reward hacking: policy exploits reward model errors
- KL penalty tuning is sensitive ( $\beta$  hyperparameter)
- Distribution shift between stages
- Mode collapse in policy optimization

## Question

Can we optimize for human preferences **directly** without explicit reward modeling?

# Direct Preference Optimization (DPO)

## Key Insight

Reparameterize the reward model in terms of the optimal policy to eliminate the explicit reward modeling stage

### RLHF (3 stages):

- 1 Train  $\pi_{\text{ref}}$
- 2 Train  $r_{\phi}$
- 3 Train  $\pi_{\theta}$  with PPO

### Challenges:

- Complex pipeline
- Reward model errors
- Hyperparameter sensitivity

### DPO (2 stages):

- 1 Train  $\pi_{\text{ref}}$
- 2 Train  $\pi_{\theta}$  directly on preferences

### Advantages:

- Simpler pipeline
- No reward model
- More stable training
- Implicit KL regularization

# DPO Mathematical Derivation (1)

## Starting Point: Optimal Policy

The optimal policy for RLHF is:

$$\pi^*(y|x) = \frac{1}{Z(x)} \pi_{\text{ref}}(y|x) \exp\left(\frac{1}{\beta} r(x, y)\right)$$

where  $Z(x)$  is the partition function.

## Reparameterization

Solve for the reward function:

$$r(x, y) = \beta \log \frac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)} + \beta \log Z(x)$$

## Direct Preference Objective

Substitute into the preference probability and optimise directly:

$$\mathcal{L}_{\text{DPO}}(\pi_{\theta}) = -\mathbb{E}_{(x, y^{+}, y^{-})} \left[ \log \sigma \left( \beta \log \frac{\pi_{\theta}(y^{+}|x)}{\pi_{\text{ref}}(y^{+}|x)} \right) \right. \quad (1)$$

$$\left. - \beta \log \frac{\pi_{\theta}(y^{-}|x)}{\pi_{\text{ref}}(y^{-}|x)} \right) \Big]. \quad (2)$$

# DPO Loss (Log Probabilities)

```
1 def dpo_loss(policy_model, ref_model, batch, beta=0.1):
2     """Direct Preference Optimization loss"""
3     # Compute log probabilities from policy
4     policy_chosen_logps = policy_model.log_prob(
5         batch['chosen_ids'], batch['chosen_mask'])
6     policy_rejected_logps = policy_model.log_prob(
7         batch['rejected_ids'], batch['rejected_mask'])
8
9     # Reference model (frozen)
10    with torch.no_grad():
11        ref_chosen_logps = ref_model.log_prob(
12            batch['chosen_ids'], batch['chosen_mask'])
13        ref_rejected_logps = ref_model.log_prob(
14            batch['rejected_ids'], batch['rejected_mask'])
15
```

# DPO Loss (Optimization Step)

```
1  # Preference-aligned objective
2  logits = beta * ((policy_chosen_logps - ref_chosen_logps)
3                  - (policy_rejected_logps - ref_rejected_logps))
4  loss = -F.logsigmoid(logits).mean()
5
6  # Implicit KL penalty is automatically handled
7  return loss
```

**Implementation:** exp08\_dpo\_implementation.py

# Monte Carlo Tree Search (MCTS)

## Planning vs Learning

While RLHF/DPO focus on learning from preferences, MCTS combines planning and learning for sequential decision making

### Key Concepts:

- Tree-based search algorithm
- Monte Carlo simulations
- Upper Confidence Bounds (UCB)
- Balances exploration vs exploitation

### Applications:

- Game playing (Go, Chess)
- Robotics planning
- Combinatorial optimization

### Four Phases:

- 1 **Selection**: Navigate tree using UCB
- 2 **Expansion**: Add new nodes
- 3 **Simulation**: Evaluate leaf nodes
- 4 **Backpropagation**: Update statistics

### Statistics per node:

- $N(s, a)$ : Visit count
- $W(s, a)$ : Total reward
- $Q(s, a) = W(s, a)/N(s, a)$ : Mean value

# Upper Confidence bound for Trees (UCT)

## UCB1 Selection Rule

For each action  $a$  at state  $s$ , compute:

$$\text{UCB1}(s, a) = Q(s, a) + c \sqrt{\frac{\ln N(s)}{N(s, a)}}$$

Select action:  $a^* = \arg \max_a \text{UCB1}(s, a)$

**Exploitation term:**  $Q(s, a)$

- Current best estimate
- Based on empirical average
- Favors promising actions

**Exploration term:**  $c \sqrt{\frac{\ln N(s)}{N(s, a)}}$

- Confidence interval width
- Larger for less-visited actions
- Decreases as  $N(s, a)$  increases

## Theoretical Guarantee

UCT has regret bound  $O(\sqrt{\ln T / T})$  where  $T$  is number of simulations

# PUCT: Predictor + UCT

## Neural Network Enhanced MCTS

Use neural network to provide prior probabilities  $P(s, a)$  and value estimates  $V(s)$

## PUCT Selection Formula

$$\text{PUCT}(s, a) = Q(s, a) + c_{\text{puct}} P(s, a) \frac{\sqrt{N(s)}}{1 + N(s, a)}$$

### Key differences from UCT:

- Prior  $P(s, a)$  guides exploration
- Uses  $\sqrt{N(s)}$  instead of  $\ln N(s)$
- Value function  $V(s)$  for leaf evaluation
- No random rollouts needed

# PUCT Network Architecture

## Neural policy-value head:

- Input: Game state  $s$
- Policy head:  $P(s, a)$  over actions
- Value head:  $V(s) \in [-1, 1]$
- Shared convolutional representation layers

## Key Innovation

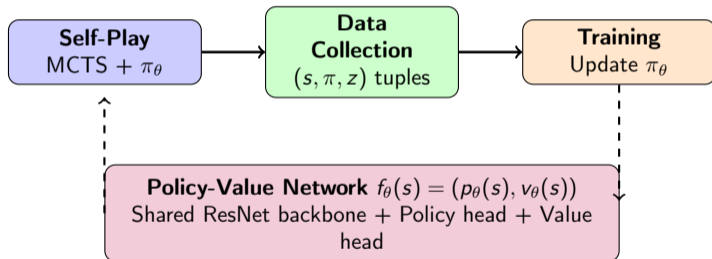
Learned priors provide an intuition that dramatically reduces search depth.

**Implementation:** `exp05_puct_mcts.py`

# AlphaZero: Self-Play with PUCT

## Combining Search and Learning

AlphaZero integrates MCTS with self-supervised learning through self-play



Continuous improvement through self-play

# AlphaZero Training Objective

## Data Generation

Each self-play game generates training examples  $(s_t, \pi_t, z)$  where:

- $s_t$ : Game state at time  $t$
- $\pi_t$ : MCTS policy (visit count distribution)
- $z$ : Game outcome from  $s_t$ 's player perspective

## Loss Function

$$\mathcal{L}(\theta) = (z - v_{\theta}(s))^2 - \pi^T \log p_{\theta}(s) + c \|\theta\|_2^2$$

where:

- Value loss:  $(z - v_{\theta}(s))^2$  – MSE between outcome and predicted value
- Policy loss:  $-\pi^T \log p_{\theta}(s)$  – Cross-entropy with MCTS policy
- Regularization:  $c \|\theta\|_2^2$  – L2 weight decay

# AlphaZero Self-Improvement Loop

## Self-Improvement Loop

Better network → Better MCTS → Better training data → Better network

**Implementation:** `exp06_selfplay_alphazero.py`

## Core Components We'll Build

- ❶ **Gomoku 5×5 Environment:** Perfect information game for testing
- ❷ **Policy-Value Network:** CNN with dual heads
- ❸ **PUCT MCTS:** Tree search with neural guidance
- ❹ **Self-Play Trainer:** Data generation and model updates
- ❺ **Toy Language Model:** For DPO experiments
- ❻ **DPO Trainer:** Direct preference optimization

## Game Playing Stack

- Gomoku environment
- ResNet policy-value network
- PUCT MCTS (25-100 simulations)
- Self-play training loop
- Tournament evaluation

## Language Model Stack

- Character-level tokenizer
- Small transformer (2-4 layers)
- Synthetic preference dataset
- DPO optimisation loop
- Preference accuracy metrics

# Experimental Design Principles

## Reproducibility Requirements

- Fixed random seeds (Python, NumPy, PyTorch)
- Deterministic CUDA operations where possible
- Version tracking (PyTorch, CUDA, library versions)
- Hardware specifications (CPU, GPU, memory)
- Hyperparameter logging and checkpointing

## Performance Optimization

- Mixed precision training (AMP) when available
- `torch.compile` for model acceleration
- Batched inference for MCTS tree expansion
- Efficient tensor operations and memory management
- CPU/GPU device handling and data transfer optimization

- **AlphaZero**: Win rate vs random, previous versions, search budget analysis
- **DPO**: Preference accuracy, KL divergence, perplexity changes
- **Both**: Training loss curves, convergence analysis, hyperparameter sensitivity

# Experiment 1: Setup Verification

**Script:** `exp01_setup_verification.py`

## Objectives

- Verify PyTorch installation, device selection, and deterministic seeding
- Confirm Hugging Face tokenizer availability and simple Gomoku board checks
- Exercise AMP and `torch.compile` hooks needed for later experiments

## Expected Output (seed 42)

- Console prints device string (e.g., `cpu`) and PyTorch version
- Board diagnostic reports “Legal actions: 23/25” after placing two stones
- Mixed-precision test prints “Mixed precision training: ✓” when CUDA is available

# Experiment 1: Setup Verification Code

```
1  #!/usr/bin/env python3
2  """Verify PyTorch, CUDA, transformers, and board diagnostics"""
3
4  import torch, random, numpy as np
5
6  def setup_seed(seed=42):
7      random.seed(seed)
8      np.random.seed(seed)
9      torch.manual_seed(seed)
10     if torch.cuda.is_available():
11         torch.cuda.manual_seed_all(seed)
12
13     device = torch.device(
14         "cuda" if torch.cuda.is_available()
15         else "mps" if hasattr(torch.backends, "mps")
16             and torch.backends.mps.is_available()
17         else "cpu"
18     )
19     setup_seed(42)
20
21     print(f"Device: {device}")
22     print(f"PyTorch version: {torch.__version__}")
23     print(f"CUDA available: {torch.cuda.is_available()}")
24
```

## Experiment 2: Training Infrastructure

**Script:** `exp02_standard_training_header.py`

### Objectives

- Provide reusable dataclass configs, checkpoint/save utilities, and logging scaffolding
- Exercise AMP-aware checkpoint saves that handle `torch.compile` wrapped modules
- Benchmark inference throughput and DQN updates with reproducible seeds

### Expected Output (seed 42)

- Console shows “Config save/load: ✓” followed by checkpoint save/load confirmations
- TensorBoard log directory created under `test_logs/` with scalar entries
- Throughput line like “Model throughput: 850.0 inferences/sec” and a DQN loss scalar

## Experiment 2: Config & Checkpoint Code

```
1 @dataclass
2 class ExperimentConfig:
3     learning_rate: float = 1e-4
4     batch_size: int = 32
5     beta: float = 0.1          # DP0 temperature
6     c_puct: float = 1.0       # MCTS exploration
7     num_simulations: int = 25 # MCTS budget
8     device: str = str(device)
9     seed: int = 42
10
11 class CheckpointManager:
12     """Handles model checkpointing with torch.compile support"""
13
14     def save_checkpoint(self, model, optimizer, epoch, metrics, config):
15         model_state = (model.state_dict() if not hasattr(model, "_orig_mod")
16                        else model._orig_mod.state_dict())
17
18         checkpoint = {
19             "model_state_dict": model_state,
20             "optimizer_state_dict": optimizer.state_dict(),
21             "epoch": epoch,
22             "metrics": metrics,
23             "config": asdict(config),
24         }
25         torch.save(checkpoint, checkpoint_path)
```

## Experiment 3: Gomoku 5×5 Environment

**Script:** `exp03_gomoku_environment.py`

### Objectives

- Implement a 5×5 Gomoku environment with legal action masks and terminal detection
- Validate perspective switching, cloning, and horizontal/diagonal win checks
- Produce board summaries for downstream MCTS/AlphaZero experiments

### Expected Output (seed 42)

- Sequence of “Testing ...” lines concluding with “All tests passed! ✓”
- Demonstration game trace printing moves and rewards, finishing with detected winner
- Environment specification footer confirming board size, action count, and observation shape

## Experiment 3: Gomoku Environment Snippet

```
1 class Gomoku5x5:
2     """5x5 Gomoku with perspective-based observations"""
3
4     def reset(self):
5         self.board = np.zeros((5, 5), dtype=np.int8)
6         self.current_player = 1 # Black starts
7         return self._get_observation(), {}
8
9     def _get_observation(self):
10        current_stones = (self.board == self.current_player)
11        opponent_stones = (self.board == -self.current_player)
12        obs = np.stack([current_stones, opponent_stones], axis=0)
13        return torch.from_numpy(obs.astype(np.float32))
14
15    def step(self, action):
16        row, col = divmod(action, 5)
17        self.board[row, col] = self.current_player
18
19        winner = self._check_winner()
20        if winner != 0:
21            reward = 1.0 if winner == self.current_player else -1.0
22            return self._get_observation(), reward, True, False, {}
23
24        self.current_player *= -1
25        return self._get_observation(), 0.0, False, False, {}
26
```

# Experiment 4: Policy-Value Network

**Script:** `exp04_policy_value_network.py`

## Objectives

- Build a shared-convolution trunk with dual policy/value heads for Gomoku boards
- Support legal action masking, AMP-aware forward passes, and optional `torch.compile`
- Provide a trainer utility for supervised policy/value updates from self-play data

## Expected Output (seed 42)

- Reports total parameter count ( $\approx 1.1\text{M}$ ) and marks “Architecture: ✓”
- Training step summary with scalar losses and “Evaluation: ✓” once metrics computed
- Demo section prints value estimate ( $\approx 0.0$  at reset) and top-5 move probabilities

## Experiment 4: Policy-Value Network Snippet

```
1 class PolicyValueNet(nn.Module):
2     """ResNet-style CNN with policy and value heads"""
3
4     def __init__(self, input_channels=2, hidden_channels=64,
5                   num_residual_blocks=4):
6         super().__init__()
7         self.stem = nn.Sequential(
8             nn.Conv2d(input_channels, hidden_channels, 3, padding=1),
9             nn.BatchNorm2d(hidden_channels), nn.ReLU(),
10        )
11        self.residual_blocks = nn.ModuleList(
12            ResidualBlock(hidden_channels) for _ in range(num_residual_blocks)
13        )
14        self.policy_head = nn.Sequential(
15            nn.Conv2d(hidden_channels, 2, 1), nn.BatchNorm2d(2),
16            nn.ReLU(), nn.Flatten(), nn.Linear(50, 25),
17        )
18        self.value_head = nn.Sequential(
19            nn.Conv2d(hidden_channels, 1, 1), nn.BatchNorm2d(1),
20            nn.ReLU(), nn.Flatten(), nn.Linear(25, 64),
21            nn.ReLU(), nn.Linear(64, 1), nn.Tanh(),
22        )
23
```

## Experiment 5: PUCT MCTS Implementation

**Script:** `exp05_puct_mcts.py`

### Objectives

- Implement PUCT selection with visit counts, priors, and mean value updates
- Support batched neural guidance by coupling to the policy-value net
- Provide temperature control, Dirichlet noise, and performance benchmarks

### Expected Output (seed 42)

- “Testing MCTS node: ✓” followed by “MCTS search: ✓” when Gomoku + network available
- Benchmark lines such as “25 simulations: 0.087s per search” (CPU numbers slightly higher)
- Demo trace listing moves with root values and top-3 action probabilities until termination

## Experiment 5: MCTS Node Snippet

```
1 class MCTSNode:
2     def __init__(self, prior=0.0, parent=None):
3         self.parent = parent
4         self.children = {}
5         self.N, self.W, self.Q = 0, 0.0, 0.0
6         self.P = prior
7
8     def select_action(self, c_puct):
9         def puct_value(action, child):
10             if child.N == 0:
11                 return float('inf')
12             bonus = c_puct * child.P * math.sqrt(self.N) / (1 + child.N)
13             return child.Q + bonus
14         return max(self.children, key=lambda a: puct_value(a, self.children[a]))
15
16     def backup(self, value):
17         self.N += 1
18         self.W += value
19         self.Q = self.W / self.N
20         if self.parent:
21             self.parent.backup(-value)
22
```

# Experiment 6: AlphaZero Self-Play Training

**Script:** `exp06_selfplay_alphazero.py`

## Objectives

- Combine MCTS policy targets with replay buffer for AlphaZero-style updates
- Track self-play outcomes, buffer growth, and supervised training loss
- Provide evaluation hooks against random opponents for quick smoke tests

## Expected Output (seed 42)

- “Experience buffer: ✓” and “Self-play trainer: ✓” when MCTS is available
- Demo logs summarizing games (e.g., “Game 2: Black wins in 17 moves”) and buffer size
- Evaluation block with win/draw/loss rates (random baseline  $\approx 33$ )

## Experiment 6: Self-Play Loop Snippet

```
1 class SelfPlayTrainer:
2     def play_game(self, env):
3         """Collect one self-play game and return training triples."""
4         trajectories = []
5         env.reset()
6         while not env.is_terminal():
7             temp = 1.0 if len(trajectories) < 10 else 0.0
8             action_probs, _ = self.mcts.search(env, 25, temp)
9             policy = torch.zeros(25)
10            for action, prob in action_probs.items():
11                policy[action] = prob
12            trajectories.append((env._get_observation().clone(), policy, env.current_player))
13            action = np.random.choice(list(action_probs), p=list(action_probs.values()))
14            env.step(action)
15
16        winner = env.winner()
17        def target(player):
18            return 1.0 if winner == player else -1.0 if winner else 0.0
19        return [Experience(state, policy, target(player))
20                for state, policy, player in trajectories]
21
```

# Experiment 7: Toy Language Model for DPO

**Script:** `exp07_toy_causal_lm.py`

## Objectives

- Build a lightweight tokenizer and causal transformer compatible with DPO training
- Generate a synthetic preference dataset with (prompt, chosen, rejected) triples
- Provide sampling helpers for quick qualitative evaluation of generated text

## Expected Output (seed 42)

- “Tokenizer: ✓” with reported vocab size (default 512) and model parameter count
- Sample generation snippet showing prompt/completion pairs from the tiny model
- Preference dataset preview printing tokenized chosen vs rejected responses

## Experiment 7: Tokenizer & LM Snippet

```
1 class SimpleTokenizer:
2     def __init__(self, vocab_size=512):
3         chars = string.ascii_letters + string.digits + string.punctuation + " "
4         specials = ["<PAD>", "<BOS>", "<EOS>", "<UNK>"]
5         vocab_chars = list(set(chars)[:vocab_size - len(specials)])
6         self.vocab = specials + vocab_chars
7         self.token_to_id = {token: i for i, token in enumerate(self.vocab)}
8
9 class SimpleTransformerLM(nn.Module):
10     def __init__(self, vocab_size, embed_dim=128, num_layers=2):
11         super().__init__()
12         self.token_embedding = nn.Embedding(vocab_size, embed_dim)
13         self.position_embedding = nn.Embedding(128, embed_dim)
14         self.layers = nn.ModuleList(
15             TransformerBlock(embed_dim, num_heads=4) for _ in range(num_layers)
16         )
17         self.lm_head = nn.Linear(embed_dim, vocab_size, bias=False)
18
```

## Experiment 8: DPO Loss and Training

**Script:** `exp08_dpo_implementation.py`

### Objectives

- Implement DPO loss with implicit KL regularisation against a frozen reference model
- Support mixed-precision training loops and dataset iteration over preference batches
- Track accuracy, reward margins, and KL drift for post-lecture analysis

### Expected Output (seed 42)

- “Testing DPO loss...” followed by “DPO loss computation: ✓” and “Training step: ✓”
- Demo run logging training/validation losses ( $\approx 0.69 \rightarrow 0.55$ ) and accuracy improving  $>60$
- Beta sweep summary such as “Beta 0.1: Accuracy = 0.62, KL = 0.08” for sensitivity checks

## Experiment 8: DPO Loss Snippet

```
1 def dpo_loss(policy_chosen_logps, policy_rejected_logps,
2               reference_chosen_logps, reference_rejected_logps, beta):
3     policy_diff = policy_chosen_logps - policy_rejected_logps
4     reference_diff = reference_chosen_logps - reference_rejected_logps
5
6     logits = beta * (policy_diff - reference_diff)
7     loss = -F.logsigmoid(logits).mean()
8     accuracy = (logits > 0).float().mean()
9
10    reward_margin = beta * ((policy_chosen_logps - reference_chosen_logps)
11                             - (policy_rejected_logps - reference_rejected_logps))
12
13    return {
14        "loss": loss,
15        "accuracy": accuracy,
16        "reward_margin": reward_margin.mean(),
17    }
```

## Experiment 8: Mixed Precision Step

```
1 with torch.autocast(device_type="cuda" if amp_enabled else "cpu"):
2     policy_chosen = compute_log_likelihood(policy_model, chosen_ids, chosen_mask)
3     policy_rejected = compute_log_likelihood(policy_model, rejected_ids, rejected_mask)
4     with torch.no_grad():
5         ref_chosen = compute_log_likelihood(ref_model, chosen_ids, chosen_mask)
6         ref_rejected = compute_log_likelihood(ref_model, rejected_ids, rejected_mask)
7     loss_dict = dpo_loss(policy_chosen, policy_rejected, ref_chosen, ref_rejected, beta)
8
```

## Experiment 9: Integration Testing & Benchmarks

**Script:** `exp09_integrated_capstone_test.py`

### Comprehensive Test Suite

- **Component Tests:** Individual module functionality
- **Integration Tests:** End-to-end pipeline validation
- **Reproducibility Tests:** Fixed seed consistency
- **Performance Benchmarks:** Throughput and timing measurements

### Expected Output (seed 42)

- Console banner “LECTURE 11 INTEGRATION TEST SUITE” followed by eight PASS entries
- Summary line “Success Rate: 9/9 tests passed” (CPU runs may skip performance timings gracefully)
- JSON report saved to `runs/integration_suite/integration_report.json`

## Experiment 9: Runner Skeleton

```
1 from lecture11.experiments import EXPERIMENT_ROOT, RUNS_ROOT
2
3 class IntegrationTestSuite:
4     def __init__(self):
5         self.test_results: dict[str, dict[str, object]] = {}
6         self.artifacts_dir = (RUNS_ROOT / "integration_suite")
7         self.artifacts_dir.mkdir(parents=True, exist_ok=True)
8         self.configs_dir = self.artifacts_dir / "configs"
9         self.checkpoints_dir = self.artifacts_dir / "checkpoints"
10        for folder in (self.configs_dir, self.checkpoints_dir):
11            folder.mkdir(parents=True, exist_ok=True)
12
13    def run_all_tests(self) -> dict[str, object]:
14        print("=" * 60)
15        print("LECTURE 11 INTEGRATION TEST SUITE")
16        self.test_results["environment"] = self.test_environment_setup()
17        self.test_results["integration"] = self.test_full_integration()
18        return self.generate_test_report()
```

# Key Implementation Insights: AlphaZero

- Temperature annealing: high early temperature, greedy late in games.
- Dirichlet noise only at the root to encourage exploration.
- Perspective handling: always view the board from the current player.
- Experience replay via reservoir sampling for diverse batches.
- Evaluate against previous checkpoints and random baselines.

# Key Implementation Insights: DPO

- Keep the reference model frozen to avoid distribution shift.
- Tune  $\beta$ : larger values emphasise preference margins but weaken KL penalties.
- Sum sequence-level log-probs with proper padding masks.
- Log-sigmoid retains numerical stability during optimisation.
- Track preference accuracy and implicit KL divergence each epoch.

## Common Pitfalls

Masking bugs, missing regularisation, or device placement errors quickly destabilise training.

# Hyperparameter Sensitivity: AlphaZero

- $c_{\text{puct}}$ : explore strength (0.5–2.0).
- Number of simulations: 10–100 per move.
- Temperature decay: when to switch to greedy play.
- Dirichlet  $\alpha$ : controls exploration noise at the root.
- Optimiser settings: learning rate, weight decay, gradient clipping.

# Hyperparameter Sensitivity: DPO

- $\beta$  in  $[0.01, 1.0]$  sets preference strength.
- Learning rate typically lower than SFT baselines.
- Batch size and sequence length tied to GPU memory.
- Warmup steps and scheduler smoothing for stable optimisation.

## Experimental Design Tips

- Grid or random search across the most sensitive knobs.
- Multiple random seeds for statistically meaningful comparisons.
- Inspect learning curves and ablations to isolate components.

## Computational Scaling

- Distributed MCTS and parallel rollouts.
- Model or tensor parallelism for larger networks.
- Gradient accumulation for big batches.
- Mixed precision training for throughput.
- Efficient data loading and preprocessing pipelines.

## Methodological Extensions

- Curriculum or multi-task learning for complex domains.
- Transfer learning and continual adaptation.
- Human-in-the-loop refinement and evaluation.

# Scaling to Real Applications: Use Cases

- Game playing: Chess, Go, StarCraft, Poker.
- Language models: RLHF/DPO for chat assistants.
- Robotics: Manipulation, navigation, dexterous control.
- Science: Protein folding, drug discovery, theorem proving.
- Finance: Trading strategies, risk management, portfolio optimisation.

- Constitutional AI and multi-layered preference learning.
- RLAIIF: AI-generated feedback to scale supervision.
- Iterative DPO pipelines with refreshed preference data.
- Multi-objective optimisation balancing safety, helpfulness, style.
- Modelling preference uncertainty and annotator disagreement.

# Research Frontiers: Planning

- MuZero-style learned dynamics for model-based planning.
- Gumbel AlphaZero and improved sampling strategies.
- Continuous-action and partial-observability extensions to MCTS.
- Multi-agent search algorithms for competitive settings.

## Emerging Trends

Foundation models with RLHF/DPO, multi-modal preference learning, and scalable oversight methods.

## Key Takeaways: Technical

- RLHF/DPO removes the reward model by optimising preferences directly.
- Neural-guided MCTS powers AlphaZero-style planning loops.
- Self-improvement requires careful data pipelines and evaluation.
- Implementation details (logging, seeding, hardware) matter for reproducibility.

## Key Takeaways: Perspective

- Modern RL blends search, learning, and preferences in one system.
- Rigorous evaluation means multiple metrics and statistical tests.
- Grow complexity gradually: validate components before scaling.
- Ethics and safety considerations are integral to deployment.