MIPROv2: Multiprompt Instruction PRoposal Optimizer Version 2

Overview

MIPROv2 is a **prompt optimizer** for DSPy programs that jointly optimizes both **instructions** and **few-shot demonstrations** for each module in a Language Model (LM) program. It uses **Bayesian Optimization** to efficiently search over the space of possible prompt configurations.

The core insight is that optimizing LM programs requires solving two interrelated challenges:

1. **The Proposal Problem**: How to efficiently generate high-quality candidate instructions and demonstrations
2. **The Credit Assignment Problem**: How to determine which prompt configurations contribute to overall performance when you only have task-level feedback

---

The Three-Step Pipeline

MIPROv2 operates in three main phases:

Step 1: Bootstrap Few-Shot Examples

The algorithm generates candidate few-shot demonstrations through **rejection sampling**:

1. Randomly sample inputs `x` from the training set
2. Run them through the LM program `Φ(x)` to generate traces
3. If the output scores well according to metric `μ` (i.e., `μ(Φ(x), x') ≥ λ`), keep the input/output traces as valid demonstration candidates
4. Repeat until you have `num_candidates` sets of demonstrations

This creates multiple candidate demo sets, each containing:

- `max_bootstrapped_demos` bootstrapped examples (from successful traces)
- `max_labeled_demos` basic examples sampled directly from training

Step 2: Propose Instruction Candidates

For each module in the program, MIPROv2 generates diverse instruction candidates using a **grounded proposer**. The proposer LM receives:

Context Component	Description
Dataset Summary	Auto-generated description of patterns in the training data
Program Summary	Description of the program's control flow and purpose
Bootstrapped Demos	Example input/outputs showing successful task completions
Tips	Randomly sampled prompt engineering suggestions (e.g., "be creative", "be concise")

The tips encourage diversity in proposed instructions. Available tips include:

- `"none"`: No additional guidance
- `"creative"`: "Don't be afraid to be creative!"
- `"simple"`: "Keep the instruction clear and concise."
- `"description"`: "Make sure your instruction is very informative and descriptive."
- `"high_stakes"`: Include a high stakes scenario
- `"persona"`: Provide the LM with a relevant persona

Step 3: Bayesian Optimization Search

This is the key innovation. Instead of random search, MIPROv2 uses a **Tree-structured Parzen Estimator (TPE)** to learn which combinations of instructions and demonstrations work best:

```
for trial in 1..num_trials:
    # Sample a configuration using TPE's learned priors
    config = TPE.sample({
        instruction_for_module_1: choice from candidates,
        instruction_for_module_2: choice from candidates,
        demos_for_module_1: choice from candidate sets,
        demos_for_module_2: choice from candidate sets,
        ...
    })

    # Evaluate on a minibatch (not full dataset)
    score = evaluate(program_with_config, minibatch)

    # Update TPE's belief about good configurations
    TPE.update(config, score)

    # Periodically do full evaluation on best candidates
    if trial % minibatch_full_eval_steps == 0:
        full_evaluate(best_averaging_config)
```

Key advantages of this approach:

1. **Efficient exploration**: TPE learns to focus on promising regions of the search space
2. **Credit assignment**: The Bayesian model learns which parameters (instructions, demos) matter most
3. **Minibatching**: Evaluating on small batches allows more trials within the same budget
4. **Joint optimization**: Instructions and demos are optimized together, capturing interactions

---

Algorithm Pseudocode

```
Algorithm: MIPROv2

Input:
  - Program Φ with m modules
  - Metric μ
  - Training data D
  - num_candidates (N)
  - num_trials (T)
  - minibatch_size (B)

Phase 1 - Bootstrap Demonstrations:
  for each module i in Φ:
    demo_candidates[i] = []
    for j in 1..N:
      demo_set = bootstrap_demos(Φ, D, μ, max_bootstrapped, max_labeled)
      demo_candidates[i].append(demo_set)

Phase 2 - Propose Instructions:
  for each module i in Φ:
    instruction_candidates[i] = []
    dataset_summary = summarize_dataset(D)
    program_summary = summarize_program(Φ)
    for j in 1..N:
      tip = random_choice(tips)
      demos = random_choice(demo_candidates[i])
      instruction = proposer_LM(dataset_summary, program_summary, demos, tip)
      instruction_candidates[i].append(instruction)

Phase 3 - Bayesian Optimization:
  Initialize TPE with uniform priors over all candidates
  best_score = -∞
  best_config = None

  for trial in 1..T:
    # Sample configuration using TPE
```

```
  config = {}
  for each module i:
    config[f"instruction_{i}"] = TPE.sample(instruction_candidates[i])
    config[f"demos_{i}"] = TPE.sample(demo_candidates[i])

  # Evaluate on minibatch
  minibatch = random_sample(D, B)
  program_configured = apply_config(Φ, config)
  score = evaluate(program_configured, minibatch, μ)

  # Update TPE
  TPE.update(config, score)

  # Periodic full evaluation
  if trial % minibatch_full_eval_steps == 0:
    best_averaging = get_best_averaging_config(TPE)
    full_score = evaluate(apply_config(Φ, best_averaging), D, μ)
    if full_score > best_score:
      best_score = full_score
      best_config = best_averaging

return apply_config(Φ, best_config)
```

Key Parameters

Parameter	Description	Default
`metric`	Evaluation function for program outputs	Required
`auto`	Preset mode: `"light"`, `"medium"`, `"heavy"`	`"light"`
`num_candidates`	Number of instruction/demo candidates per module	Set by `auto`
`max_bootstrapped_demos`	Max demos from successful traces	4
`max_labeled_demos`	Max demos directly from training	4
`prompt_model`	LM for proposing instructions	Default LM
`task_model`	LM used in the actual program	Default LM
`minibatch_size`	Size of evaluation batches	35
`minibatch_full_eval_steps`	Trials between full evaluations	5

Auto Modes

The `auto` parameter provides convenient presets:

Mode	Use Case	Trials	Candidates
`"light"`	Quick optimization, limited budget	Fewer	Fewer
`"medium"`	Balanced optimization	Medium	Medium
`"heavy"`	Maximum quality, larger budget	Many	Many

Proposer Configurations

MIPROv2 supports several flags to control what context the proposer sees:

Flag	Description
`program_aware_proposer`	Include program structure summary
`data_aware_proposer`	Include dataset characteristics summary
`tip_aware_proposer`	Include random prompt engineering tips
`fewshot_aware_proposer`	Include bootstrapped demonstrations

Comparison with Related Methods

Method	Instructions	Demos	Search Strategy	Credit Assignment
Bootstrap Random Search	✗	✓	Random	None
OPRO	✓	✗	History-based LM	LM implicit
Module-Level OPRO	✓	✗	Per-module history	Program score proxy
0-Shot MIPRO	✓	✗	Bayesian (TPE)	Surrogate model
Bayesian Bootstrap	✗	✓	Bayesian (TPE)	Surrogate model
MIPRO / MIPROv2	✓	✓	Bayesian (TPE)	Surrogate model

Key Findings from the Paper

The MIPRO paper (Opsahl-Ong et al., 2024) established several important lessons:

1. **Bootstrapped demonstrations are crucial**: For most tasks, optimizing demos alone outperforms optimizing instructions alone
2. **Joint optimization is best**: MIPRO (instructions + demos together) generally achieves the highest performance
3. **Instructions matter for conditional rules**: When tasks have subtle rules that can't be inferred from few examples, instruction optimization becomes essential
4. **Grounding helps but varies**: The usefulness of dataset/program summaries varies by task—MIPRO++ can learn optimal proposal strategies
5. **Bayesian optimization is robust**: The surrogate model handles noisy evaluations and efficiently explores the search space

---

Example Usage

```python
```

```python
import dspy
from dspy.teleprompt import MIPROv2

# Configure your LM
lm = dspy.LM('openai/gpt-4o-mini', api_key='...')
dspy.configure(lm=lm)

# Define your metric
def my_metric(prediction, example):
    return prediction.answer == example.answer

# Initialize optimizer
optimizer = MIPROv2(
    metric=my_metric,
    auto="medium",  # or "light" / "heavy"
)

# Define your program
class MyProgram(dspy.Module):
    def __init__(self):
        self.generate = dspy.ChainOfThought("question -> answer")

    def forward(self, question):
        return self.generate(question=question)

# Optimize
optimized_program = optimizer.compile(
    MyProgram(),
    trainset=train_data,
    # Optional: provide a validation set
    valset=val_data,
)

# Save for later use
optimized_program.save("optimized_program.json")
```

Practical Recommendations

1. **Start with `auto="light"`** to get a quick baseline, then increase to `"medium"` or `"heavy"` if you have compute budget
2. **For zero-shot optimization** (no few-shot demos), set `max_bootstrapped_demos=0` and `max_labeled_demos=0`

3. **For tasks with complex rules**, provide a good seed instruction that describes the rules, as optimizers struggle to infer all rules automatically
4. **Use a stronger teacher model** for bootstrapping on challenging tasks (e.g., GPT-4 instead of the task model)
5. **Monitor optimization progress**: The Bayesian model's learned importance scores can reveal which components matter most for your task

---

References

- Opsahl-Ong, K., Ryan, M. J., Purtell, J., et al. (2024). *Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs*. arXiv:2406.11695
- DSPy Documentation: https://dspy.ai/api/optimizers/MIPROv2/