- Read
- Write
- Glob
- Grep

DSPy GEPA Optimizer

Goal

Optimize complex agentic systems using LLM reflection on full execution traces with Pareto-based evolutionary search.

When to Use

- **Agentic systems** with tool use
- When you have **rich textual feedback** on failures
- Complex multi-step workflows
- Instruction-only optimization needed

Inputs

Input	Type	Description
`program`	`dspy.Module`	Agent or complex program
`trainset`	`list[dspy.Example]`	Training examples
`metric`	`callable`	Must return `(score, feedback)` tuple
`reflection_lm`	`dspy.LM`	Strong LM for reflection (GPT-4)
`auto`	`str`	“light”, “medium”, “heavy”

Outputs

Output	Type	Description
`compiled_program`	`dspy.Module`	Reflectively optimized program

Workflow

Phase 1: Define Feedback Metric

GEPA requires metrics that return textual feedback:

```
def gepa_metric(example, pred, trace=None):
    """Must return (score, feedback) tuple."""
    is_correct = example.answer.lower() in pred.answer.lower()

    if is_correct:
        feedback = "Correct. The answer accurately addresses the question."
    else:
        feedback = f"Incorrect. Expected '{example.answer}' but got '{pred.answer}'.
The model may have misunderstood the question or retrieved irrelevant information."

    return is_correct, feedback
```

Phase 2: Setup Agent

```
import dspy

def search(query: str) -> list[str]:
    """Search knowledge base for relevant information."""
    results = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')(query,
k=3)
    return [r['text'] for r in results]

def calculate(expression: str) -> float:
    """Safely evaluate mathematical expressions."""
    return dspy.PythonInterpreter({}).execute(expression)

agent = dspy.ReAct("question -> answer", tools=[search, calculate])
```

Phase 3: Optimize with GEPA

```
dspy.configure(lm=dspy.LM("openai/gpt-4o-mini"))

optimizer = dspy.GEPA(
    metric=gepa_metric,
    reflection_lm=dspy.LM("openai/gpt-4o"),  # Strong model for reflection
    auto="medium"
)

compiled_agent = optimizer.compile(agent, trainset=trainset)
```

Production Example

```python
import dspy
from dspy.evaluate import Evaluate
import logging

logger = logging.getLogger(__name__)

class ResearchAgent(dspy.Module):
    def __init__(self):
        self.react = dspy.ReAct(
            "question -> answer",
            tools=[self.search, self.summarize]
        )

    def search(self, query: str) -> list[str]:
        """Search for relevant documents."""
        results = dspy.ColBERTv2(
            url='http://20.102.90.50:2017/wiki17_abstracts'
        )(query, k=5)
        return [r['text'] for r in results]

    def summarize(self, text: str) -> str:
        """Summarize long text into key points."""
        summarizer = dspy.Predict("text -> summary")
        return summarizer(text=text).summary

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

def detailed_feedback_metric(example, pred, trace=None):
    """Rich feedback for GEPA reflection."""
    expected = example.answer.lower().strip()
    actual = pred.answer.lower().strip() if pred.answer else ""

    # Exact match
    if expected == actual:
        return 1.0, "Perfect match. Answer is correct and concise."

    # Partial match
    if expected in actual or actual in expected:
        return 0.7, f"Partial match. Expected '{example.answer}', got 
'{pred.answer}'. Answer contains correct info but may be verbose or incomplete."

    # Check for key terms
    expected_terms = set(expected.split())
    actual_terms = set(actual.split())
    overlap = len(expected_terms & actual_terms) / max(len(expected_terms), 1)

    if overlap > 0.5:
        return 0.5, f"Some overlap. Expected '{example.answer}', got '{pred.answer}'. 
Key terms present but answer structure differs."

    return 0.0, f"Incorrect. Expected '{example.answer}', got '{pred.answer}'. The 
agent may need better search queries or reasoning."

def optimize_research_agent(trainset, devset):
    """Full GEPA optimization pipeline."""

    dspy.configure(lm=dspy.LM("openai/gpt-4o-mini"))

    agent = ResearchAgent()

    # Convert metric for evaluation (just score)
    def eval_metric(example, pred, trace=None):
        score, _ = detailed_feedback_metric(example, pred, trace)
        return score

    evaluator = Evaluate(devset=devset, metric=eval_metric, num_threads=8)
    baseline = evaluator(agent)
    logger.info(f"Baseline: {baseline:.2%}")

    # GEPA optimization
    optimizer = dspy.GEPA(
        metric=detailed_feedback_metric,
        reflection_lm=dspy.LM("openai/gpt-4o"),
        auto="medium",
        enable_tool_optimization=True  # Also optimize tool descriptions
    )

    compiled = optimizer.compile(agent, trainset=trainset)
    optimized = evaluator(compiled)
    logger.info(f"Optimized: {optimized:.2%}")

    compiled.save("research_agent_gepa.json")
    return compiled
```

Tool Optimization

GEPA can jointly optimize predictor instructions AND tool descriptions:

```
optimizer = dspy.GEPA(
    metric=gepa_metric,
    reflection_lm=dspy.LM("openai/gpt-4o"),
    auto="medium",
    enable_tool_optimization=True  # Optimize tool docstrings too
)
```

Best Practices

1. **Rich feedback** - More detailed feedback = better reflection
2. **Strong reflection LM** - Use GPT-4 or Claude for reflection
3. **Agentic focus** - Best for ReAct and multi-tool systems
4. **Trace analysis** - GEPA analyzes full execution trajectories

Limitations

- Requires custom feedback metrics (not just scores)
- Expensive: uses strong LM for reflection
- Newer optimizer, less battle-tested than MIPROv2
- Best for instruction optimization, less for demos