DSPy comprehensive end-to-end guide

02/12/2026 12:19:57

Teach me DSPy end to end for all scenarios and proper explanation of what why how

I'll teach you DSPy comprehensively - from fundamentals to advanced scenarios....

Teach me DSPy end to end for all scenarios and proper explanation of what why how

2/12/2026, 11:54:11 AM

I'll teach you DSPy comprehensively - from fundamentals to advanced scenarios. This is a complete guide to understanding and mastering DSPy.

DSPy: Complete End-to-End Guide

Part 1: What is DSPy?

DSPy (Declarative Self-improving Language Programs, yeah) is a framework for programming with language models. Think of it as replacing prompt engineering with actual programming.

The Core Problem DSPy Solves

Traditional LLM development:

```
# Traditional approach - brittle and hard to optimize
prompt = f"""You are a helpful assistant.
Given this question: {question}
And this context: {context}
Answer the question concisely."""

response = llm(prompt)
```

Problems:

- Prompts are fragile strings
- Hard to compose multiple LLM calls
- No systematic way to improve
- Each task needs manual prompt engineering

DSPy's Solution

DSPy treats prompts as **learnable parameters** rather than hand-crafted strings. It provides:

1. **Signatures**: Declare what your LM should do (input/output spec)
2. **Modules**: Composable LM operations
3. **Optimizers**: Automatically improve prompts and weights
4. **Metrics**: Define what "good" means

Part 2: Core Concepts

2.1 Signatures

Signatures define the input-output behavior of your LM call.

```
import dspy

# Simple signature - just input/output field names
class Emotion(dspy.Signature):
    """Classify the emotion in a sentence."""
    sentence = dspy.InputField()
    emotion = dspy.OutputField(desc="one of: happy, sad, angry, neutral")

# Shorthand notation
"sentence -> emotion"

# With descriptions
"question, context -> answer"
```

Why signatures? They're type-safe, composable, and can be optimized. DSPy knows what each field is for.

2.2 Modules

Modules are DSPy's building blocks - they use signatures to do LM operations.

Basic Modules

```python
# Predict - basic LM call
class EmotionClassifier(dspy.Module):
    def __init__(self):
        super().__init__()
        self.classify = dspy.Predict(Emotion)

    def forward(self, sentence):
        return self.classify(sentence=sentence)

# Usage
lm = dspy.OpenAI(model="gpt-3.5-turbo")
dspy.settings.configure(lm=lm)

classifier = EmotionClassifier()
result = classifier("I just got a promotion!")
print(result.emotion)  # "happy"
```

ChainOfThought - adds reasoning

```python
class CoTEmotionClassifier(dspy.Module):
    def __init__(self):
        super().__init__()
        # ChainOfThought adds a "rationale" field automatically
        self.classify = dspy.ChainOfThought(Emotion)

    def forward(self, sentence):
        return self.classify(sentence=sentence)

result = classifier("I just got a promotion!")
print(result.rationale)  # "The sentence mentions a promotion
```

```
which is positive..."
print(result.emotion)    # "happy"
```

Why modules? They're:

- Composable (nest them like LEGO)
- Trackable (DSPy knows the computation graph)
- Optimizable (can be improved automatically)

2.3 Data and Metrics

```
# Training data - just dicts or objects with the fields from your
signature
train_data = [
    dspy.Example(sentence="I love this!",
emotion="happy").with_inputs("sentence"),
    dspy.Example(sentence="This is terrible.",
emotion="angry").with_inputs("sentence"),
    # ... more examples
]

# Metric - defines success
def emotion_accuracy(example, prediction, trace=None):
    return example.emotion.lower() == prediction.emotion.lower()
```

Part 3: Building Blocks in Depth

3.1 All Core Modules

```
# 1. Predict - basic call
predictor = dspy.Predict("question -> answer")

# 2. ChainOfThought - adds reasoning
```

```python
cot = dspy.ChainOfThought("question -> answer")

# 3. ProgramOfThought - generates and executes code
pot = dspy.ProgramOfThought("question -> answer")

# 4. ReAct - reasoning + action loop
react = dspy.ReAct("question -> answer")

# 5. MultiChainComparison - generates multiple chains, picks best
mcc = dspy.MultiChainComparison("question -> answer")
```

3.2 Composing Modules

```python
class RAGSystem(dspy.Module):
    def __init__(self, num_passages=3):
        super().__init__()
        self.retrieve = dspy.Retrieve(k=num_passages)
        self.generate = dspy.ChainOfThought("context, question -> answer")

    def forward(self, question):
        # Retrieve relevant passages
        passages = self.retrieve(question).passages

        # Join into context
        context = "\n\n".join(passages)

        # Generate answer
        return self.generate(context=context, question=question)

# Usage
rag = RAGSystem()
result = rag("What is photosynthesis?")
print(result.answer)
```

The power: Each component can be optimized independently or together!

Part 4: Optimization - The Magic

4.1 Why Optimize?

Your initial program might work, but optimization:

- Generates better prompts than you could manually
- Finds better examples for few-shot learning
- Tunes which modules to use where
- Adapts to your specific task

4.2 Common Optimizers

```
from dspy.teleprompt import BootstrapFewShot, MIPRO,
BayesianSignatureOptimizer

# 1. BootstrapFewShot - learns good examples
optimizer = BootstrapFewShot(
    metric=emotion_accuracy,
    max_bootstrapped_demos=4,  # how many examples to learn
    max_labeled_demos=8        # max to use from training set
)

# 2. MIPRO - optimizes prompts AND examples
optimizer = MIPRO(
    metric=emotion_accuracy,
    num_candidates=10,
    init_temperature=1.0
)

# 3. BayesianSignatureOptimizer - optimizes instructions
optimizer = BayesianSignatureOptimizer(
```

```python
    metric=emotion_accuracy,
    max_iterations=20
)

# Run optimization
optimized_program = optimizer.compile(
    student=classifier,
    trainset=train_data,
    valset=val_data  # optional validation set
)
```

4.3 What Each Optimizer Does

BootstrapFewShot:

1. Runs your program on training data
2. Collects successful traces (input → output paths that worked)
3. Uses these as few-shot examples in future runs
4. **Use when**: You have a working program, want to improve accuracy

MIPRO (Multi-prompt Instruction Proposal Optimizer):

1. Generates multiple instruction variants
2. Tests different few-shot example combinations
3. Bayesian optimization to find best config
4. **Use when**: You want comprehensive optimization, have more time

BayesianSignatureOptimizer:

1. Focuses on optimizing the instruction text
2. Generates and tests instruction variants
3. Uses Bayesian optimization for efficiency
4. **Use when**: Examples are good, but instructions need work

Part 5: Real-World Scenarios

Scenario 1: Simple Classification

```
import dspy

# Setup
lm = dspy.OpenAI(model="gpt-3.5-turbo")
dspy.settings.configure(lm=lm)

# Define task
class SentimentClassification(dspy.Signature):
    """Classify sentiment of a movie review."""
    review = dspy.InputField()
    sentiment = dspy.OutputField(desc="positive, negative, or
neutral")

# Build program
class SentimentClassifier(dspy.Module):
    def __init__(self):
        super().__init__()
        self.classify =
dspy.ChainOfThought(SentimentClassification)

    def forward(self, review):
        return self.classify(review=review)

# Prepare data
train = [
    dspy.Example(review="Great movie!",
sentiment="positive").with_inputs("review"),
    dspy.Example(review="Waste of time.",
sentiment="negative").with_inputs("review"),
    # ... more
]
```

```
# Metric
def accuracy(example, pred, trace=None):
    return example.sentiment == pred.sentiment

# Optimize
optimizer = BootstrapFewShot(metric=accuracy)
optimized = optimizer.compile(SentimentClassifier(),
trainset=train)

# Use
result = optimized("This film was amazing!")
```

Scenario 2: Multi-Hop Question Answering

```
class MultiHopQA(dspy.Module):
    def __init__(self):
        super().__init__()
        # Break down into sub-questions
        self.decompose = dspy.ChainOfThought(
            "question -> sub_question1, sub_question2"
        )

        # Answer each sub-question
        self.answer_subq = dspy.ChainOfThought("question ->
answer")

        # Synthesize final answer
        self.synthesize = dspy.ChainOfThought(
            "question, answer1, answer2 -> final_answer"
        )

    def forward(self, question):
        # Decompose
```

```
        decomp = self.decompose(question=question)

        # Answer sub-questions
        ans1 = self.answer_subq(question=decomp.sub_question1)
        ans2 = self.answer_subq(question=decomp.sub_question2)

        # Synthesize
        final = self.synthesize(
            question=question,
            answer1=ans1.answer,
            answer2=ans2.answer
        )

        return final

# This can now be optimized as a single unit!
```

Scenario 3: RAG with Retrieval

```
import dspy
from dspy.retrieve.faiss_rm import FaissRM

# Setup retriever
retriever = FaissRM(
    document_chunks=your_documents,
    vectorizer=your_embedding_model
)
dspy.settings.configure(rm=retriever)

class RAG(dspy.Module):
    def __init__(self, num_passages=5):
        super().__init__()
        self.retrieve = dspy.Retrieve(k=num_passages)
        self.generate = dspy.ChainOfThought("context, question ->
```

```
answer")

    def forward(self, question):
        context = self.retrieve(question).passages
        context_str = "\n\n".join(context)
        return self.generate(context=context_str,
question=question)

# Optimize with answer correctness metric
def rag_metric(example, pred, trace=None):
    # Could use exact match, F1, or LLM-as-judge
    return example.answer.lower() in pred.answer.lower()

optimizer = MIPRO(metric=rag_metric)
optimized_rag = optimizer.compile(RAG(), trainset=train_data)
```

Scenario 4: Agent with Tools

```
class AgentWithTools(dspy.Module):
    def __init__(self, tools):
        super().__init__()
        self.tools = tools
        self.planner = dspy.ChainOfThought(
            "question, available_tools -> plan, tool_to_use,
tool_input"
        )
        self.synthesizer = dspy.ChainOfThought(
            "question, tool_output -> final_answer"
        )

    def forward(self, question):
        # Plan which tool to use
        plan = self.planner(
            question=question,
```

```
            available_tools=str(list(self.tools.keys()))
        )

        # Execute tool
        tool = self.tools[plan.tool_to_use]
        tool_output = tool(plan.tool_input)

        # Synthesize answer
        return self.synthesizer(
            question=question,
            tool_output=str(tool_output)
        )

# Define tools
tools = {
    "calculator": lambda x: eval(x),
    "search": lambda x: web_search(x),
}

agent = AgentWithTools(tools)
```

Scenario 5: Self-Refining Generator

```
class SelfRefiningWriter(dspy.Module):
    def __init__(self):
        super().__init__()
        self.generate = dspy.ChainOfThought("topic -> essay")
        self.critique = dspy.ChainOfThought("essay -> critique, suggestions")
        self.refine = dspy.ChainOfThought(
            "essay, critique -> improved_essay"
        )

    def forward(self, topic, num_refinements=2):
```

```
        # Initial generation
        result = self.generate(topic=topic)
        essay = result.essay

        # Refine multiple times
        for _ in range(num_refinements):
            critique = self.critique(essay=essay)
            refined = self.refine(
                essay=essay,
                critique=critique.suggestions
            )
            essay = refined.improved_essay

        return dspy.Prediction(essay=essay)
```

Part 6: Advanced Patterns

6.1 Assertions and Suggestions

```
import dspy
from dspy.primitives.assertions import assert_transform_module,
backtrack_handler

class AnswerWithCitations(dspy.Module):
    def __init__(self):
        super().__init__()
        self.generate = dspy.ChainOfThought("context, question ->
answer, citations")

    def forward(self, context, question):
        result = self.generate(context=context,
question=question)

        # Assert citations are present
```

```
        dspy.Assert(
            len(result.citations) > 0,
            "Answer must include citations"
        )

        # Suggest improvement
        dspy.Suggest(
            result.answer.endswith('.'),
            "Answer should end with proper punctuation"
        )

        return result

# Apply assertion handling
answer_module = assert_transform_module(
    AnswerWithCitations(),
    backtrack_handler
)
```

6.2 Typed Predictors

```
from pydantic import BaseModel, Field
from typing import List

class Person(BaseModel):
    name: str
    age: int
    occupation: str

class ExtractPeople(dspy.Signature):
    """Extract people mentioned in text."""
    text: str = dspy.InputField()
    people: List[Person] = dspy.OutputField()
```

```python
class PeopleExtractor(dspy.Module):
    def __init__(self):
        super().__init__()
        self.extract = dspy.TypedPredictor(ExtractPeople)

    def forward(self, text):
        return self.extract(text=text)

# Returns properly typed Pydantic objects!
result = extractor("John, 30, is a teacher. Mary, 25, is an
engineer.")
for person in result.people:
    print(f"{person.name}: {person.occupation}")
```

6.3 Custom Modules

```python
class CustomEnsemble(dspy.Module):
    def __init__(self, num_predictors=3):
        super().__init__()
        self.predictors = [
            dspy.ChainOfThought("question -> answer")
            for _ in range(num_predictors)
        ]
        self.vote = dspy.ChainOfThought(
            "question, answers -> best_answer, reasoning"
        )

    def forward(self, question):
        # Get predictions from all predictors
        predictions = [p(question=question) for p in
self.predictors]
        answers = [p.answer for p in predictions]

        # Vote on best answer
```

```python
        result = self.vote(
            question=question,
            answers="\n".join(f"{i+1}. {a}" for i, a in enumerate(answers))
        )

        return result
```

Part 7: Optimization Deep Dive

7.1 Understanding Bootstrapping

```python
# What happens during BootstrapFewShot:

# 1. Teacher runs on training examples
teacher = YourModule()
successful_traces = []

for example in trainset:
    prediction = teacher(**example.inputs())
    if metric(example, prediction) > threshold:
        # Save this as a good example
        successful_traces.append({
            'inputs': example.inputs(),
            'outputs': prediction
        })

# 2. Student is compiled with these examples
student = YourModule()
student.predictors[0].demos = successful_traces  # Added as few-shot examples

# Now when student runs, it sees good examples!
```

7.2 Custom Metrics

```python
# Simple exact match
def exact_match(example, pred, trace=None):
    return example.answer == pred.answer

# Partial credit
def partial_match(example, pred, trace=None):
    correct_words = set(example.answer.lower().split())
    pred_words = set(pred.answer.lower().split())
    overlap = len(correct_words & pred_words)
    return overlap / len(correct_words) if correct_words else 0

# LLM-as-judge
judge = dspy.ChainOfThought("question, gold_answer, pred_answer -> score, reasoning")

def llm_judge_metric(example, pred, trace=None):
    result = judge(
        question=example.question,
        gold_answer=example.answer,
        pred_answer=pred.answer
    )
    return int(result.score) / 10  # Assuming 0-10 scale

# Multi-aspect metric
def comprehensive_metric(example, pred, trace=None):
    correctness = exact_match(example, pred)
    has_reasoning = len(pred.get('rationale', '')) > 50
    proper_format = pred.answer.endswith('.')

    return (correctness * 0.7 + has_reasoning * 0.2 + proper_format * 0.1)
```

7.3 Optimization Strategies

```
# Strategy 1: Quick iteration
quick_optimizer = BootstrapFewShot(
    metric=my_metric,
    max_bootstrapped_demos=2,
    max_rounds=1
)
quick_model = quick_optimizer.compile(module,
trainset=small_train)

# Strategy 2: Thorough optimization
thorough_optimizer = MIPRO(
    metric=my_metric,
    num_candidates=20,
    init_temperature=1.4,
    verbose=True
)
best_model = thorough_optimizer.compile(
    module,
    trainset=full_train,
    valset=validation,
    num_trials=100
)

# Strategy 3: Staged optimization
# First optimize instructions
stage1 = BayesianSignatureOptimizer(metric=my_metric)
improved_module = stage1.compile(module, trainset=train)

# Then optimize examples
stage2 = BootstrapFewShot(metric=my_metric)
final_module = stage2.compile(improved_module, trainset=train)
```

Part 8: Production Patterns

8.1 Caching and Efficiency

```
import dspy
from functools import lru_cache

class EfficientRAG(dspy.Module):
    def __init__(self):
        super().__init__()
        self.retrieve = dspy.Retrieve(k=5)
        self.generate = dspy.ChainOfThought("context, question -> answer")

        # Cache retrievals
        self._retrieve_cached = lru_cache(maxsize=1000)(self._retrieve_impl)

    def _retrieve_impl(self, question):
        return self.retrieve(question).passages

    def forward(self, question):
        passages = self._retrieve_cached(question)
        context = "\n\n".join(passages)
        return self.generate(context=context, question=question)
```

8.2 Saving and Loading

```
# Save optimized program
optimized_module.save("my_model.json")

# Load it back
loaded_module = MyModule()  # Create fresh instance
loaded_module.load("my_model.json")
```

```
# Now it has the optimized prompts and examples!
```

8.3 Monitoring and Logging

```
class MonitoredModule(dspy.Module):
    def __init__(self):
        super().__init__()
        self.predictor = dspy.ChainOfThought("question -> answer")
        self.call_count = 0
        self.error_count = 0

    def forward(self, question):
        self.call_count += 1
        try:
            result = self.predictor(question=question)

            # Log prediction
            print(f"Call {self.call_count}: Q={question[:50]}... A={result.answer[:50]}...")

            return result
        except Exception as e:
            self.error_count += 1
            print(f"Error on call {self.call_count}: {e}")
            raise
```

Part 9: Common Pitfalls and Solutions

Pitfall 1: Not enough training data

```python
# Bad: Only 5 examples
train = [example1, example2, example3, example4, example5]

# Good: At least 50-200 for real tasks
# If you don't have labels, use bootstrapping from unlabeled data
```

Pitfall 2: Weak metrics

```python
# Bad: Too lenient
def bad_metric(example, pred, trace=None):
    return True  # Everything passes!

# Good: Meaningful measure of quality
def good_metric(example, pred, trace=None):
    return f1_score(example.answer, pred.answer) > 0.7
```

Pitfall 3: Over-optimization

```python
# Watch for overfitting to training set
optimizer = BootstrapFewShot(metric=my_metric)
optimized = optimizer.compile(module, trainset=train)

# Always validate on held-out data
val_score = evaluate(optimized, valset)
if val_score < threshold:
    print("Model might be overfitting!")
```

Part 10: Complete Example - News Summarizer

```python
import dspy
from dspy.teleprompt import BootstrapFewShot
```

```python
# 1. Setup
lm = dspy.OpenAI(model="gpt-4")
dspy.settings.configure(lm=lm)

# 2. Define signatures
class ExtractKeyPoints(dspy.Signature):
    """Extract key points from a news article."""
    article = dspy.InputField()
    key_points = dspy.OutputField(desc="3-5 bullet points")

class WriteSummary(dspy.Signature):
    """Write a concise summary from key points."""
    key_points = dspy.InputField()
    summary = dspy.OutputField(desc="2-3 sentence summary")

# 3. Build module
class NewsSummarizer(dspy.Module):
    def __init__(self):
        super().__init__()
        self.extract = dspy.ChainOfThought(ExtractKeyPoints)
        self.summarize = dspy.ChainOfThought(WriteSummary)

    def forward(self, article):
        points = self.extract(article=article)
        summary = self.summarize(key_points=points.key_points)
        return dspy.Prediction(
            key_points=points.key_points,
            summary=summary.summary
        )

# 4. Prepare data
train_data = [
    dspy.Example(
        article="Long article text here...",
```

```python
        summary="Expected summary..."
    ).with_inputs("article"),
    # ... more examples
]

# 5. Define metric
def summary_quality(example, pred, trace=None):
    # Use ROUGE or similar
    from rouge import Rouge
    rouge = Rouge()
    scores = rouge.get_scores(pred.summary, example.summary)[0]
    return scores['rouge-l']['f']

# 6. Optimize
optimizer = BootstrapFewShot(
    metric=summary_quality,
    max_bootstrapped_demos=4
)
optimized_summarizer = optimizer.compile(
    NewsSummarizer(),
    trainset=train_data
)

# 7. Use
article = "Breaking news article..."
result = optimized_summarizer(article)
print(f"Summary: {result.summary}")
print(f"Key Points: {result.key_points}")
```

Summary: When to Use What

Use Predict: Simple one-shot tasks **Use ChainOfThought**: Need reasoning/explanation **Use ProgramOfThought**: Mathematical or code execution needed **Use ReAct**: Agent-like behavior with tools **Use BootstrapFewShot**: Quick optimization with good initial program

Use MIPRO: Comprehensive optimization, have time/budget **Use BayesianSignatureOptimizer**: Focus on instruction quality

The beauty of DSPy: Start simple, add complexity only where needed, and let optimization find what works!

2/12/2026, 11:55:50 AM