Effective Testing in Deep Learning Framework

From Assert to Unittest with PyTorch MNIST

Seonghoon Jeong

System and Network Security Lab. (SNSec Lab.)
Division of Artificial Intelligence Engineering, Sookmyung Women's University

January 30, 2026

현실 세계의 개발자: 디버깅과 버그 수정

개발자로 취직했는데 하루 종일 버그 찾고 고치고 있습니다 원래 이런 일이 주류인가요?

EDIT DELETE REPLY

2023-06-27 12:38:18 #3800177

소프트웨어엔지니어 172.***.100.17 2298

무언가 새로운 feature들을 만들고 프로젝트를 하고 그럴 줄 알았는데 ..

기존에 존재하는 프로그램 유저들이 버그 보고하면 버그 수정하고 잘 되는지 확인하고 3달째 이런 일만 하는 중입니다. 이직해야할까요?

4

9

Source: https://www.workingus.com/forums/topic/개발자로-취직했는데-하루-종일-버그-찾고-고치고-있/

기술 면접에서의 질문

개발 중 문제가 발생하면 어떤 방법으로 해결하는 편인지

버그를 찾거나 프로젝트를 테스트하는 자신만의 프로세스가 있는지

개발자에게 제일 중요한 역량이 무엇이라고 생각하는지

가입한 개발자 커뮤니티가 있는지

가장 자신 있는 기술 분야

직무와 관련된 과목 중 성적이 제일 좋았던 과목/제일 좋아하는 과목

최근 IT 기술 중 관심 있는 것

최근에 관심 있게 참여한 컨퍼런스가 있는지

(비전공자) 전공이 직무와 안 맞는데 직무를 선택한 이유는?

(비전공자) 전공자와 비교했을 때 본인이 경쟁력이 있다고 생각하는지

(비전공자) 전공 지식을 어떻게 보완할 것인지

Source: https://corin-e.tistory.com/entry/신입-IT-개발자-면접-질문-총-정리-인성회사직무경험기술

Deep Learning Code Framework

Typical components of a PyTorch project:

- **Data Preprocessing**: Raw data cleaning and formatting.
  - Raw CAN messages → time-synchronized sensor values
  - Packet dump → a series of timestamp/payloads per stream
- **Data Pipeline**: Loading, augmentation, and batching.
- **Model**: Neural network architecture definition.
- **Training Loop**: Forward/backward pass, optimization, reporting (`wandb`).
- **Validation/Testing**: Monitoring performance on unseen data.

Bugs can hide silently in any of these interacting components.

Why Test Deep Learning Code?

- **Silent Bugs:** Models can "train" even with broken logic (e.g., incorrect broadcasting).
- **Complexity:** Interactions between data loaders, models, and training loops are complex.
- **Reproducibility:** Ensuring specific inputs yield deterministic outputs.
- **Refactoring:** Safely optimizing code without breaking functionality.

Outline

1. Introduction
2. assert
3. unittest
4. unittest demo using MNIST classifier

The `assert` Statement

What is it?

- A debugging aid that tests a condition.
- If the condition is true, nothing happens.
- If the condition is false, it raises an `AssertionError`.

When to use it?

- Verifying logical assumptions (e.g., invariants).
- Checking input contracts in internal functions.
- Checking output validity logic.

Note: Asserts can be disabled with the `-O` *flag in Python (optimization mode).*

Assert Usage: Basics

Syntax: `assert condition, message`

```
x = 10

# 1. Simple statement
assert x > 0

# 2. Statement with error message (Recommended)
assert x > 0, "x must be positive"

# 3. Checking types/shapes shortcuts
assert isinstance(x, int)

# 4. Debugging with f-strings (Very Helpful!)
assert x > 0, f"Expected positive x, got {x}"
```

Assert Usage: Input Verification

Example: Accuracy Calculation Catch invalid inputs early to prevent silent failures.

```python
def calculate_accuracy(correct, total):
    assert total > 0, "Total must be positive"
    assert 0 <= correct <= total, "Correct cannot exceed total"
    return correct / total

# Usage that triggers error
acc = calculate_accuracy(10, 0)
```

$$\text{Accuracy} = \frac{\text{correct}}{\text{total}}$$

where:

$$0 < \text{total}$$
$$0 \leq \text{correct} \leq \text{total}$$

Execution Result:

```
Traceback (most recent call last):
  File "assert_demo.py", line 8, in <module>
    acc = calculate_accuracy(10, 0)    # AssertionError: Total must be positive
          ^^^^^^^^^^^^^^^^^^^^^^^^^
  File "assert_demo.py", line 2, in calculate_accuracy
    assert total > 0, "Total must be positive"
           ^^^^^^^^^
AssertionError: Total must be positive
```

Assert Usage: Output Verification

Example: Safe Normalization Ensure mathematical operations maintain required invariants (e.g., unit vectors).

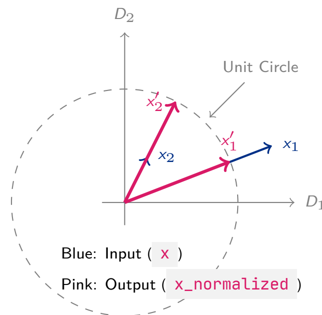
```
def safe_normalize(x, eps=1e-6):
    """Normalize vectors to unit length.

    Args:
        x: Input tensor of shape (B,D)
        eps: Small value to avoid division by zero
    Returns:
        Normalized tensor of shape (B,D)
    """
    norm = x.norm(dim=1, keepdim=True)
    x_normalized = x / (norm + eps)

    # Verify pre-condition: output vectors should have length 1.0
    # This catches bugs like forgetting 'eps' or wrong dimension
    final_norms = x_normalized.norm(dim=1)
    target = torch.ones_like(final_norms)

    assert torch.allclose(final_norms, target, atol=1e-4), \
        f"Normalization failed. Max deviation: {(final_norms - target).abs
      ().max()}"

    return x_normalized
```



Introduction to `unittest`

Python's standard library for automated testing.

Key Concepts:

- **TestCase:** The fundamental unit of testing organization in the framework.
- **Assertion:**
  - Decides whether a test passes or fails.
  - Checks for conditions (Boolean, equality, exceptions, etc.).
  - If an assertion fails, the test method fails.
- **Fixtures:** Preparation (`setUp`) and cleanup (`tearDown`) of test environment.

Test Organization

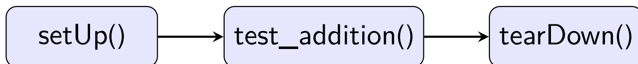
Where should `TestCase` go?

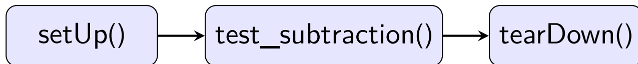
- **Separate Files (Recommended):**
  - e.g., `test_model.py` or in a `tests/` directory.
  - Keeps production code clean and lightweight.
  - Enables automatic test discovery (`python -m unittest discover`).
- **Same Module:**
  - Inside `if __name__ == '__main__':`.
  - Convenient for small scripts, but discouraged for projects.

Test Fixture (픽스처)

Fixture:

- Code executed before/after specific test functions.
- Ensures the test environment is in a known, fixed state (Pre-condition).
- Common uses: Creating DB tables (setup) or cleaning up resources (teardown).





Fresh environment for each test!

Basic unittest Structure

```python
import unittest

class TestMath(unittest.TestCase):
    def setUp(self):
        self.value = 10

    def test_addition(self):
        self.assertEqual(self.value + 5, 15)

    def test_negative(self):
        self.assertTrue(self.value - 20 < 0)

if __name__ == '__main__':
    unittest.main()
```

Execution Result

```
..
--------------------------------
Ran 2 tests in 0.000s

OK
```

Fixtures: setUp and tearDown

```python
import unittest
import os

class TestFileOps(unittest.TestCase):
    def setUp(self):
        # Run BEFORE specific test execution
        self.filepath = 'temp_test.txt'
        with open(self.filepath, 'w') as f:
            f.write('Hello')

    def tearDown(self):
        # Run AFTER (Cleanup)
        if os.path.exists(self.filepath):
            os.remove(self.filepath)

    def test_read(self):
        with open(self.filepath, 'r') as f:
            self.assertEqual(f.read(), 'Hello')
```

Execution Result

```
.
--------------------------------
Ran 1 test in 0.001s

OK
```

Common `unittest` Assertions

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIn(a, b)`	`a in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`

Note: Conventionally, `assertEqual(actual, expected)`.

https://docs.python.org/3/library/unittest.html

Unittest Demo

Demo Project Overview

A structured MNIST classifier to demonstrate testing patterns.

Structure:

- `main.py` : Entry point (No test code).
- `model.py` : The CNN architecture.
- `trainer.py` : Training loop and validation logic.
- `data_utils.py` : Data loading and preprocessing.
- **`tests/`** :
  - `test_model.py`
  - `test_trainer.py`
  - `test_data_utils.py`

The Model (`model.py`)

Architecture:

- **Conv Layers:** 2 layers with ReLU and Pooling.
- **FC Layers:** Flatten to 128 units, then 10 classes.
- **Methods:**
  - `forward(x)`: Returns logits.
  - `predict(x)`: Returns class index (0-9).
  - `predict_proba(x)`: Returns probabilities (Softmax).

```python
class MNISTClassifier(nn.Module):
    def __init__(self):
        super(MNISTClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        assert x.shape[1:] == (1, 28, 28) # ASSERTION!!!
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        return self.fc2(x)

    def predict(self, x):
        assert x.shape[1:] == (1, 28, 28) # ASSERTION!!!
        with torch.no_grad():
            return torch.argmax(self.forward(x), dim=1)
```

Test: Architecture (`test_model.py`)

```python
class TestModelArchitecture(unittest.TestCase):
    def setUp(self):
        self.model = MNISTClassifier()

    def test_layers_exist(self):
        # Verify essential components are present
        self.assertTrue(hasattr(self.model, 'conv1'))
        self.assertTrue(hasattr(self.model, 'conv2'))
        self.assertTrue(hasattr(self.model, 'fc2'))

    def test_fc2_output_size(self):
        # Final layer must have exactly 10 outputs (0-9)
        self.assertEqual(self.model.fc2.out_features, 10)
```

Test: Forward Pass Logic

```python
class TestModelForward(unittest.TestCase):
    def setUp(self):
        self.model = MNISTClassifier()

    def test_forward_output_shape(self):
        # Check if output shape is (batch_size, num_classes)
        batch_size = 8
        x = torch.randn(batch_size, 1, 28, 28)
        output = self.model(x)
        self.assertEqual(output.shape, (batch_size, 10))

    def test_forward_output_not_nan(self):
        # Ensure no numerical instability
        x = torch.randn(4, 1, 28, 28)
        output = self.model(x)
        self.assertFalse(torch.isnan(output).any())
```

Test: Prediction Logic

```python
class TestModelPrediction(unittest.TestCase):
    def test_predict_labels_range(self):
        x = torch.randn(10, 1, 28, 28)
        preds = self.model.predict(x)

        # All labels must be valid digits [0-9]
        self.assertTrue((preds >= 0).all())
        self.assertTrue((preds <= 9).all())

    def test_predict_proba_sums_to_one(self):
        x = torch.randn(4, 1, 28, 28)
        probs = self.model.predict_proba(x)

        # Sum of probabilities across classes should be 1.0
        self.assertTrue(torch.allclose(probs.sum(dim=1), torch.ones(4)))
```

Test: Gradients & Device

```python
class TestModelGradients(unittest.TestCase):
    def test_gradients_computed(self):
        x = torch.randn(4, 1, 28, 28)
        labels = torch.randint(0, 10, (4,))
        loss = nn.CrossEntropyLoss()(self.model(x), labels)
        loss.backward()

        # Verify gradients are flowing to weights
        for name, param in self.model.named_parameters():
            self.assertIsNotNone(param.grad)
            self.assertNotEqual(param.grad.norm().item(), 0)

    @unittest.skipUnless(torch.cuda.is_available(), "CUDA not available")
    def test_model_on_cuda(self):
        device = torch.device('cuda')
        model = self.model.to(device)
        self.assertEqual(next(model.parameters()).device.type, 'cuda')
```

Data Loading (`data_utils.py`)

Dataset Management:

- **Splitting:** 90% Train, 10% Val.
- **Performance:** `shuffle=True` for training.
- **Reproducibility:** Fixed seed for random split.

```python
def load_mnist_data(data_dir='./data', val_split
        =0.1):
    full_train = datasets.MNIST(..., train=True)

    val_size = int(len(full_train) * val_split)
    train_size = len(full_train) - val_size

    train_set, val_set = random_split(
        full_train, [train_size, val_size],
        generator=torch.Generator().manual_seed(42)
    )

    return (
        DataLoader(train_set, shuffle=True),
        DataLoader(val_set, shuffle=False)
    )
```

Data Preprocessing (`data_utils.py`)

```python
def normalize_image(image, mean=0.1307, std=0.3081):
    return (image - mean) / std

def denormalize_image(image, mean=0.1307, std=0.3081):
    return image * std + mean

def create_dummy_data(num_samples=100):
    # Mock data for unit testing model logic
    images = torch.randn(num_samples, 1, 28, 28)
    labels = torch.randint(0, 10, (num_samples,))
    return images, labels
```

Test: Data Utilities

```python
class TestNormalization(unittest.TestCase):
    def test_normalize_denormalize_inverse(self):
        # Validation of mathematical reversibility
        original = torch.randn(1, 28, 28)
        norm = normalize_image(original)
        recovered = denormalize_image(norm)
        self.assertTrue(torch.allclose(original, recovered, atol=1e-5))

class TestDummyData(unittest.TestCase):
    def test_create_dummy_data_custom_shape(self):
        # Ensure helper creates correct shapes for tests
        images, labels = create_dummy_data(num_samples=50)
        self.assertEqual(images.shape, (50, 1, 28, 28))
        self.assertEqual(labels.shape, (50,))
```

Training Logic (`trainer.py`)

```python
class Trainer:
    def __init__(self, model, optimizer, device):
        self.model = model.to(device)
        self.optimizer = optimizer

    def train_epoch(self, train_loader):
        self.model.train()
        total_loss = 0
        for data, target in train_loader:
            data, target = data.to(self.device), target.to(self.device)
            self.optimizer.zero_grad()
            output = self.model(data)
            loss = F.cross_entropy(output, target)
            loss.backward()
            self.optimizer.step()
            total_loss += loss.item()
        return total_loss / len(train_loader)
```

Validation Logic (`trainer.py`)

```python
    def validate(self, val_loader):
        self.model.eval()
        correct = 0
        with torch.no_grad():
            for data, target in val_loader:
                output = self.model(data)
                pred = output.argmax(dim=1)
                correct += pred.eq(target).sum().item()

        return correct / len(val_loader.dataset)

    def save_checkpoint(self, path):
        torch.save(self.model.state_dict(), path)
```

Test: Trainer Logic

```python
class TestTrainerValidation(unittest.TestCase):
    def test_validate_does_not_update_weights(self):
        # Critical: Validation should NOT change model state
        initial_w = self.model.fc2.weight.clone()
        self.trainer.validate(self.val_loader)
        self.assertTrue(torch.equal(initial_w, self.model.fc2.weight))

class TestTrainerCheckpoint(unittest.TestCase):
    def test_save_load_checkpoint(self):
        # Verify persistence
        self.trainer.save_checkpoint('ckpt.pt')
        self.model.fc2.weight.data.fill_(0) # Corrupt
        self.trainer.load_checkpoint('ckpt.pt') # Restore
        self.assertNotEqual(self.model.fc2.weight.sum(), 0)
```

Running the Tests

- **Command Line:**

```
$ python -m unittest discover -s tests -v
```

- **With Coverage:**

```
$ python -m coverage run --source=. -m unittest discover -s tests
$ python -m coverage report -m
```

Output Example

```
test_forward_output_shape (test_model.TestModelForward) ... ok
test_gradients_computed (test_model.TestModelGradients) ... ok
test_normalize_image (test_data_utils.TestNormalization) ... ok

----------------------------------------------------------------------
Ran 56 tests in 0.352s

OK
```

Test Discovery Examples

1. **Tests in Root Directory** (`./`):

```
$ python -m unittest discover -v
```

2. **Tests in** `tests/` :

```
$ python -m unittest discover -s tests -v
```

3. **Tests in Deep Subfolder** (`tests/deeplearning`):

```
$ python -m unittest discover -s tests/deeplearning -v
```

Note: `-s` *or* `--start-directory` *defines where discovery begins.*

Bonus: Testing Tokenization (NLP Example)

Scenario: Testing a text tokenizer function.

- **Goal:** Verify string processing logic.
- **Edge Cases:** Empty input, special characters, mixed case.

```python
def simple_tokenize(text):
    """Convert text to lowercase tokens."""
    if not text:
        return []
    return text.lower().strip().split()

class TestTokenizer(unittest.TestCase):
    def test_simple_case(self):
        text = "Hello Deep Learning"
        expected = ["hello", "deep", "learning"]
        self.assertEqual(simple_tokenize(text),
    expected)

    def test_special_chars_and_spaces(self):
        text = "  PyTorch  v2.0 "
        # Check stripping and splitting
        self.assertEqual(simple_tokenize(text), ["
    pytorch", "v2.0"])
```

Summary

- **Start Small:** Use `assert` for preconditions.
- **Structure:** Use `unittest.TestCase` to organize tests.
- **Scope:**
  - Test shapes and types.
  - Test mathematical invariants (probabilities sum to 1).
  - Test gradient flow (model is learning).
  - Test data pipelines (transforms are correct).
- **Automate:** Run tests frequently to catch regressions early.