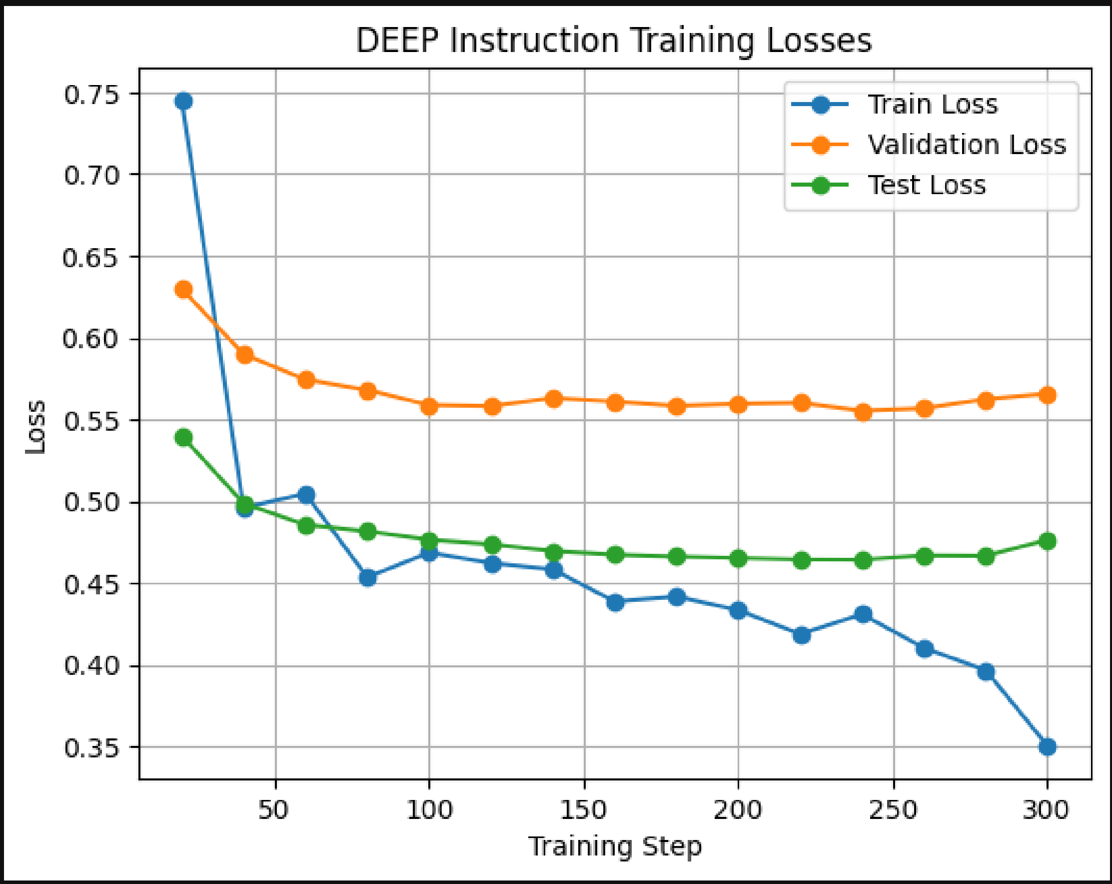
Project Analysis and Evaluation Report

1. Training Process and Loss Visualizations

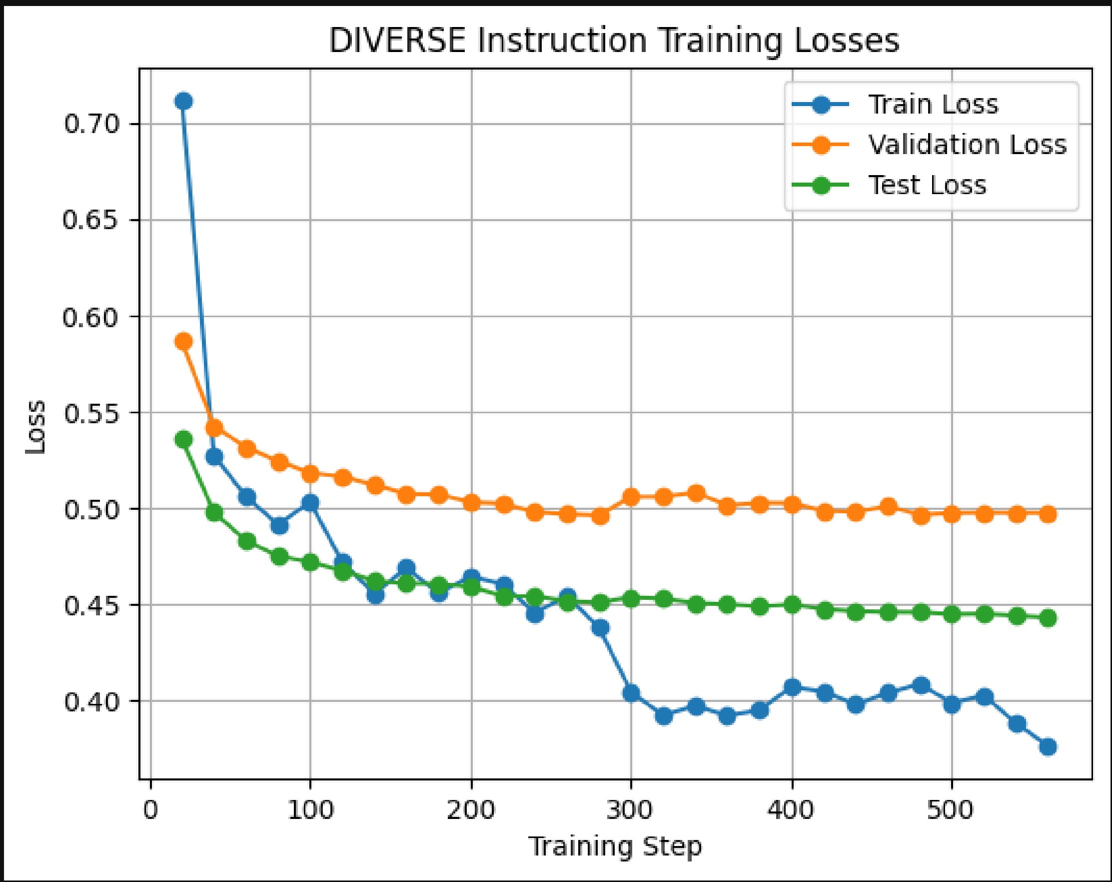
The performance of the models was monitored throughout the training process using Train, Validation, and Test Loss metrics recorded every 20 steps.

1.1. Deep Instruction Loss Graph



- Data Frequency: Values are marked at every 20-step interval.
- Analysis: In the Deep model, the Training Loss decreased steadily from approximately 0.74 to 0.35. The Test Loss reached its minimum (0.4641) at step 240, indicating the peak of the model's generalized capability.

1.2. Diverse Instruction Loss Graph



- Data Frequency: Values are marked at every 20-step interval.
- Analysis: The Diverse model's Test Loss continued to decline through step 560 (0.4432), demonstrating that the model maintained a broad learning capacity without early saturation.

---

2. Model Interpretation: Learning and Overfitting Analysis

Technical interpretations based on the loss curves are as follows:

- Learning Status: The consistent decrease in "Train Loss" for both models confirms that they successfully learned the patterns within the provided datasets.
- Overfitting Control: * Deep Model: A slight fluctuation in "Validation Loss" began after step 240; this suggests the model started trending toward memorizing the training data (overfitting) beyond this point.
  - Diverse Model: The test loss curve exhibited a smoother descent. This indicates that providing five different solution paths for each problem enhanced the model's generalization ability and reduced the risk of rote memorization.

3. Best Checkpoint Determination and Benchmark Results

The best checkpoints were determined by testing on the LiveCodeBench (release_v5) using 41 problems from the Atcoder Platform / Easy Difficulty tier.

Benchmark Comparison Table

Model	Pass@1	Solved Problems
diverse_instruction_checkpoint-step-520-epoch-0	19.5%	8/41
diverse_instruction_checkpoint-step-540-epoch-0	29.3%	12/41
diverse_instruction_checkpoint-step-560-epoch-0	29.3%	12/41

Model	Pass@1	Solved Problems
deep_instruction_checkpoint-step-220-epoch-0	33.3%	14/41
deep_instruction_checkpoint-step-240-epoch-0	29.3%	12/41
deep_instruction_checkpoint-step-260-epoch-0	31.7%	13/41

4. Model Behavioral Comparison

Based on the qualitative tests conducted using specific prompts, significant differences in how each model processes instructions were observed.

4.1. Reasoning vs. Direct Implementation

When prompted to explain the logic of an algorithm (e.g., Binary Search) before coding it (diverse/deep):

````
⌛ Loading Model and Tokenizer...
✅ Diverse checkpoint loaded successfully: /content/drive

Prompt: Calculate the distance between points (x1, y1) an
==================================================
Model Response:
Sure! Below is a Python script that calculates the Euclid

### Using Standard Math Library

```python
import math

def distance_std(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    return math.sqrt(dx*dx + dy*dy)

# Example usage:
result_std = distance_std(1, 2, 4, 6)
print(f"Standard math: {result_std}")
```


### Using NumPy for Vectors

```python
import numpy as np

def distance_numpy(x1, y1, x2, y2):
    # Create arrays from coordinates
    point1 = np.array([x1, y1])
    point2 = np.array([x2, y2])

    # Calculate difference vector and magnitude
    diff = point2 - point1
    return np.linalg.norm(diff)

# Example usage:
result_numpy = distance_numpy(1, 2, 4, 6)
print(f"Numpy: {result_numpy}")
```
````

````
⌛ Model ve Tokenizer yükleniyor...
✅ LoRA checkpoint başarıyla yüklendi: /content/drive/MyDrive/lora_project/models/deep_instruction/checkpoints/checkpoint-

Prompt: Explain the logic of a Binary Search algorithm in simple terms, then write a Python function to implement it. Afte
------------------------------
Model Yanıtı:
Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing i

### Logic

1. **Initialization**: Start with the entire array.
2. **Midpoint Calculation**: Find the middle index of the current segment.
3. **Comparison**:
   - If the target value matches the element at the midpoint, return the index.
   - If the target is greater than the element at the midpoint, search only the right half.
   - If the target is less than the element at the midpoint, search only the left half.
4. **Repeat**: Continue this process until you find the target or exhaust all possibilities (the list size becomes 0).

### Python Function Implementation

```python
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1

    return -
````
````

- Deep Model: Demonstrated superior "Chain-of-Thought" reasoning. It provided a structured, four-step logical derivation (Initialization, Midpoint Calculation, Comparison, Repeat) before writing a single line of code.
- Diverse Model: Tended to skip or minimize the formal logic steps, moving almost immediately to the Python implementation and example test cases.

4.2. Algorithmic Variety and Library Usage

In the "Distance Calculation" test, which required three different mathematical approaches(diverse/deep):

```
⌛ Loading Model and Tokenizer...
✅ Diverse checkpoint loaded successfully: /content/drive

Prompt: Calculate the distance between points (x1, y1) anc
==================================================
Model Response:
Sure! Below is a Python script that calculates the Euclide

### Using Standard Math Library

```python
import math

def distance_std(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    return math.sqrt(dx*dx + dy*dy)

# Example usage:
result_std = distance_std(1, 2, 4, 6)
print(f"Standard math: {result_std}")
```

Using NumPy for Vectors

```python
import numpy as np

def distance_numpy(x1, y1, x2, y2):
    # Create arrays from coordinates
    point1 = np.array([x1, y1])
    point2 = np.array([x2, y2])

    # Calculate difference vector and magnitude
    diff = point2 - point1
    return np.linalg.norm(diff)

# Example usage:
result_numpy = distance_numpy(1, 2, 4, 6)
print(f"Numpy: {result_numpy}")
```
```

```
⌛ Model ve Tokenizer yükleniyor...
✅ LoRA checkpoint başarıyla yüklendi: /content/drive/MyDrive/lor

Prompt: Calculate the distance between points (x1, y1) and (x2, y
------------------------------
Model Yanıtı:
Here's a Python script that does what you asked:

```python
import math

def distance_math(x1, y1, x2, y2):
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)

def distance_numpy(vector1, vector2):
    import numpy as np
    # Convert tuples to arrays
    vec1 = np.array(vector1)
    vec2 = np.array(vector2)

    # Calculate difference
    diff = vec2 - vec1

    # Calculate magnitude squared
    mag_sq = np.dot(diff, diff)

    # Return distance
    return math.sqrt(mag_sq)

def distance_custom(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    return math.sqrt(dx*dx + dy*dy)

# Test with example points
print("Distance using math:", distance_math(0, 0, 3, 4))
print("Distance using NumPy:", distance_numpy((0,0), (3,4)))
print("Distance using custom:", distance_custom(0,0, 3,4))

# Explanation for the custom function:
```

- Diverse Model: Showed a preference for modern, high-level library functions. It successfully used np.linalg.norm for the vector-based solution, which is the most efficient and standard way to find a magnitude in NumPy.
- Deep Model: Reverted to more foundational mathematical implementations. Instead of using a single high-level function, it manually calculated the magnitude using the dot product (np.dot(diff, diff)). This reflects its training on singular, deep logical paths rather than a wide variety of library-specific shortcuts.

4.3. Formatting and Pedagogy

- Deep Model: The output format is consistently more "educational," using clear headers (e.g., ### Logic, ### Python Function Implementation) to separate thinking from coding.
- Diverse Model: The output is more "developer-centric," providing clean, usable code blocks with minimal preamble, suited for users who need a quick solution rather than a tutorial.
```

5. Conclusion

In conclusion, this project demonstrates the significant impact of specialized fine-tuning strategies on the performance of the Qwen2.5-Coder-1.5B base model. By comparing the Deep Instruction and Diverse Instruction approaches, we have identified unique behavioral traits that cater to different user needs.

- Deep Instruction: This approach proved superior for tasks requiring intense logical reasoning and detailed step-by-step explanations. While it achieved a strong Pass@1 score of 33.3% at Step-220, it is more prone to fluctuations in validation loss if training extends too far.
- Diverse Instruction: This method excelled in providing flexible, modern coding solutions and better generalization. Its performance was more stable over time, reaching a consistent 29.3% Pass@1 rate by Step-540.
- Performance Gain: Both fine-tuning methods successfully elevated the base model's capabilities, transforming it from a general-purpose coder into a specialized assistant capable of following complex multi-part instructions.

Ultimately, the choice between these models depends on the use case: Deep is the ideal choice for educational tools and complex debugging where logic is paramount, whereas Diverse is better suited for rapid prototyping and versatile software development environments.