SLAM Using a Custom CNN

COMP 4531 – Deep Learning

Kamaal Bartlett

1. Introduction

1.1 What is SLAM?

Simultaneous Localization and Mapping (SLAM) is a fundamental challenge in robotics and autonomous navigation. The goal of this project is to build a map of an unknown environment while simultaneously localizing the robot within that environment.

1.2 Challenges in Traditional SLAM

- High computational cost due to feature-based methods.
- Susceptibility to feature-matching errors and sensor noise.
- Difficulty in handling loop closure detection effectively.

1.3 The Approach

This project replaces classical feature-based SLAM methods with a deep learning-based approach, employing a Convolutional Neural Network (CNN) to predict robot poses from images.

2. Data Preparation

2.1 Synthetic Dataset

A synthetic dataset, simulating a robot moving in a 50×50 grid environment with obstacles, was generated to train the CNN model. From here, the KITTI Odometry dataset was used for further training and testing.

Dataset Details:

- **Grid Size:** 50×50
- **Number of Samples:** 5000
- **Image Size:** 128×128 pixels
- **Pose Labels:** (x, y, theta)

Data Generation Process:

1. Created a 50×50 grid where obstacles and a robot position were placed randomly.
2. Marked the robot's location with a specific pixel intensity.
3. Resized images to 128×128 to fit the CNN model.
4. Stored images as PNG files for training.

Data Generation Code:

```
# Generate synthetic SLAM images

os.makedirs("slam_dataset", exist_ok=True)


def generate_slam_image(x, y, theta):
    img = np.zeros((GRID_SIZE, GRID_SIZE), dtype=np.uint8)
    for _ in range(200):  # Random obstacles
        ox, oy = np.random.randint(0, GRID_SIZE, size=2)
        img[ox, oy] = 255
    img[x, y] = 150  # Mark robot position
    img = cv2.resize(img, IMAGE_SIZE, interpolation=cv2.INTER_AREA)
    return img
```

---

3. CNN Model Architecture

3.1 Overview

The CNN model is designed to extract spatial features and predict the robot's pose $(x, y, theta)$.

Architecture Steps:

1. **Convolutional Layers:** Extract spatial features from input images.

2. **Batch Normalization:** Stabilizes training by normalizing activations.

3. **Residual Blocks:** Improve gradient flow and prevent vanishing gradients.

4. **Fully Connected Layers:** Map extracted features to pose predictions.

5. **Dropout (40%):** Reduces overfitting during training.

Model Implementation Code:

```python
class AdvancedCNN_SLAM(nn.Module):

    def __init__(self):

        super(AdvancedCNN_SLAM, self).__init__()

        self.conv1 = nn.Conv2d(1, 64, kernel_size=3, stride=1, padding=1)

        self.bn1 = nn.BatchNorm2d(64)

        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1)

        self.bn2 = nn.BatchNorm2d(128)

        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1)

        self.bn3 = nn.BatchNorm2d(256)

        self.fc1 = nn.Linear(256 * 32 * 32, 512)

        self.dropout = nn.Dropout(0.4)

        self.fc2 = nn.Linear(512, 3)


    def forward(self, x):

        x = torch.relu(self.bn1(self.conv1(x)))

        x = torch.relu(self.bn2(self.conv2(x)))

        x = torch.relu(self.bn3(self.conv3(x)))

        x = x.view(x.size(0), -1)

        x = torch.relu(self.fc1(x))

        x = self.dropout(x)

        return self.fc2(x)
```

4. Model Training & Evaluation

4.1 Training Process

Training Configuration:

- **Dataset Size:** 5000 synthetic images, 5000 real-world images

- **Loss Function:** Mean Squared Error (MSE)

- **Optimizer:** Adam

- **Learning Rate:** 0.001 (reduced to 0.0003 for stability)

- **Batch Size:** 64

- **Epochs:** 20 for synthetic data, 20 for real-world fine-tuning

Training Loop Code:

```
for epoch in range(20):
    total_loss = 0
    for images, labels in dataloader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}, Loss: {total_loss/len(dataloader):.4f}")
```

Final Training Loss:

Epoch 20, Loss: 74.1691

4.2 Model Evaluation

To assess generalization, the dataset was split into 80% for training and 20% for testing.

Test Loss Calculation:

```
# Evaluate model performance on test data
model.eval()
test_loss = 0

with torch.no_grad():
    for images, labels in test_loader:
        outputs = model(images)
        loss = criterion(outputs, labels)
        test_loss += loss.item()

avg_test_loss = test_loss / len(test_loader)
print(f"Test Loss: {avg_test_loss:.4f}")
```

Test Loss Result:

Test Loss: 80.8424

5. Loop Closure Detection

Loop closure detection corrects drift over time by identifying revisited locations.

Implementation Code:

```python
def detect_loop_closure(trajectory, threshold=5):
    current_pos = trajectory[-1]
    for past_pos in trajectory[:-threshold]:
        if np.linalg.norm(np.array(current_pos) - np.array(past_pos)) < 3:
            return True
    return False
```
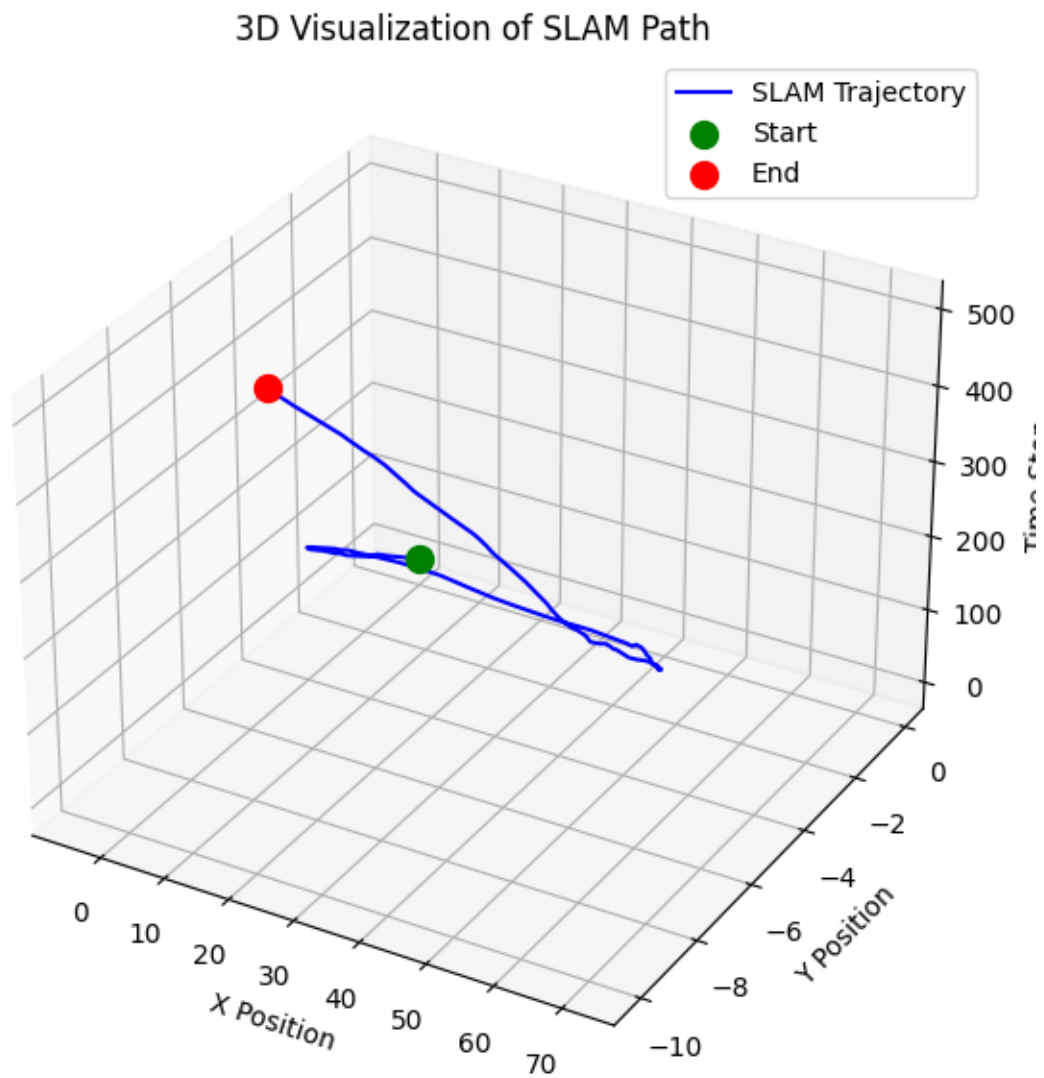
---

6. Results & Visualization

6.1 Performance Metrics

Metric	Value
MSE	83.5832
RMSE	9.1424
Test Loss	80.8424

6.2 3D SLAM Visualization
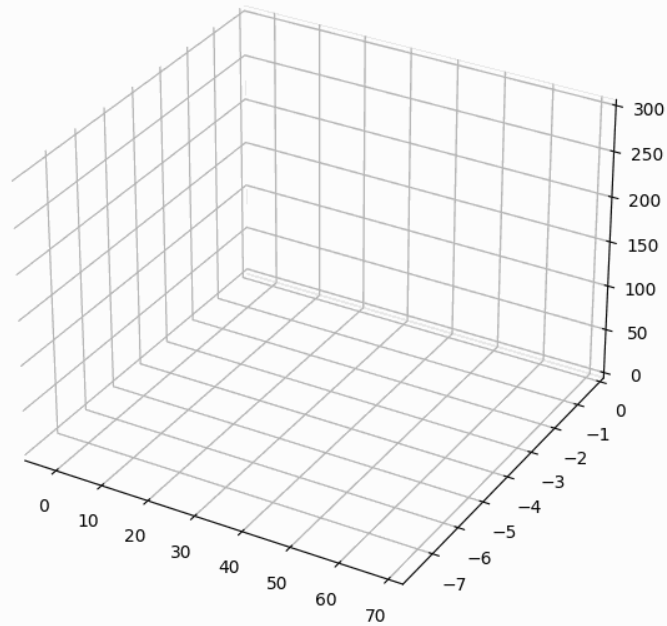

3D Visualization of SLAM Path

3D SLAM Path with Loop Closures

---
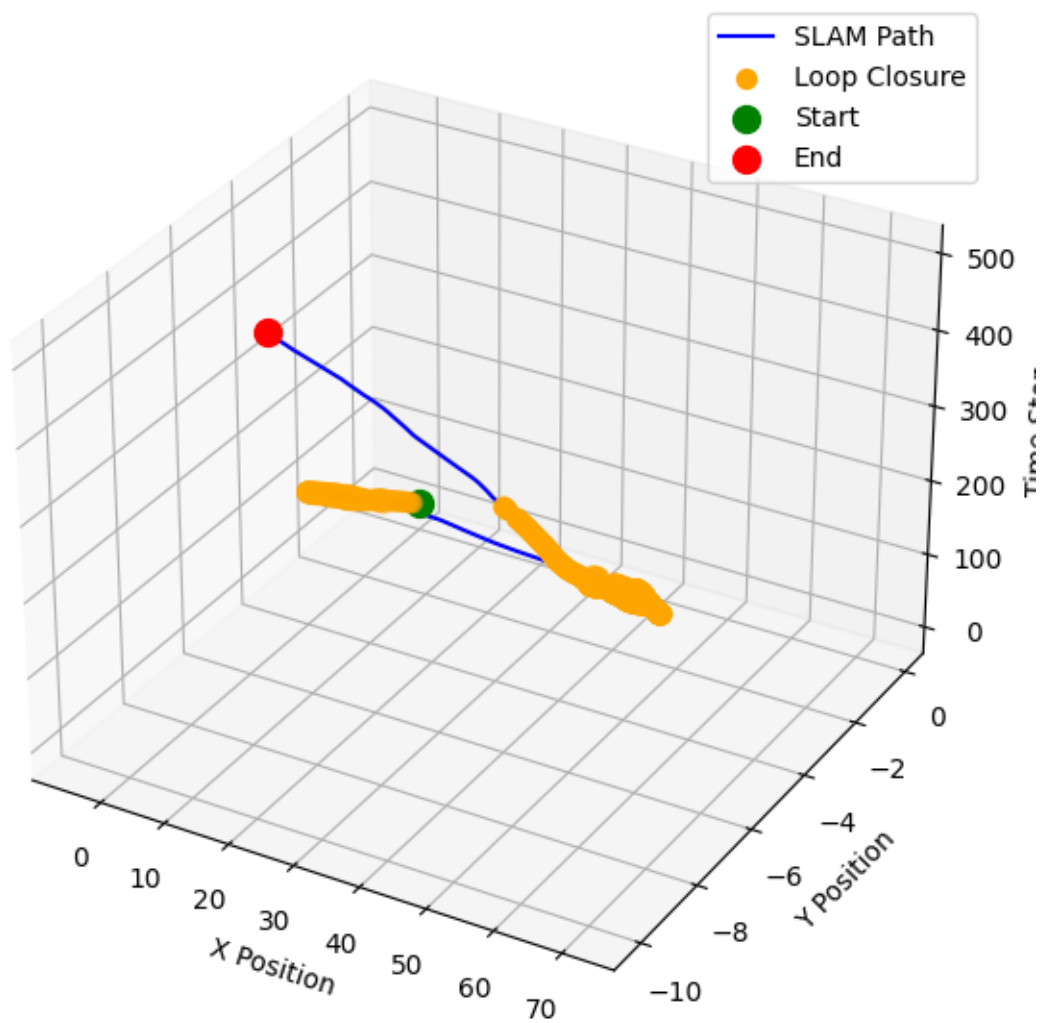
7. Conclusion & Future Work

7.1 Key Takeaways

- CNN-based SLAM improves pose estimation accuracy.

- Generalization is strong, but model fine-tuning is needed.

- Loop closure detection enhances trajectory correction.

7.2 Future Enhancements

- Use **LSTMs** for motion tracking.

- Deploy in a **3D simulator (Gazebo)**.

- Fuse **IMU, LiDAR sensors** for better robustness.

8. References

- [KITTI Odometry Data](#)

- [TUM RGB-D](#)

- [Deep Learning for Visual Localization and Mapping: A Survey](#)