Comparative Analysis of Neural Architectures: CNNs for Vision and BiLSTMs for NLP

Responsible AI (AI 354) - Lab Assignment 2

Devesh Singh Chauhan
Roll No: I23MA002
M.Sc. Mathematics, SVNIT Surat

Assigned: January 29, 2026

1 Introduction

This lab investigates the impact of architectural depth in Convolutional Neural Networks (CNNs) for image classification and the effect of regularization (dropout) in Bidirectional LSTMs (BiLSTMs) for sentiment analysis. Additionally, we evaluate the robustness of NLP models against noisy input data, a critical aspect of Responsible AI.

2 Methodology

2.1 Dataset Description

- **Vision Task:** Kaggle Clothes Dataset. 6,000 training images and 1,500 test images across 15 classes (e.g., Blazer, Jeans, Hoodie). Images were resized to $64 \times 64 \times 3$.
- **NLP Task:** Amazon Reviews Dataset. 15,000 samples balanced between positive and negative sentiments. Preprocessing included tokenization and padding to a sequence length of 100.

2.2 Model Architectures

1. **CNN (Vision):** Compared a "Shallow" 1-layer CNN against a "Standard" 2-layer CNN. Both used ReLU activation, BatchNorm, and MaxPooling, ending with a fully connected layer.
2. **BiLSTM (NLP):** A Bidirectional LSTM with embedding dimension 100 and hidden dimension 64. Dropout was applied at the embedding and LSTM layers to test generalization.

3 Results Analysis

3.1 Q1: CNN Depth Comparison

We trained two variants of the CNN for 10 epochs on a CPU.

Model Architecture	Final Training Loss	Macro F1 Score
Shallow CNN (1 Layer)	2.0564	0.3341
Standard CNN (2 Layers)	1.8368	**0.4292**

Table 1: Impact of Convolutional Depth on Performance.

Observation: Increasing the depth from 1 to 2 layers resulted in a significant performance jump ($\approx +9.5\%$ F1 Score). The 2-layer network was able to learn more complex spatial hierarchies, reducing the training loss from 2.05 to 1.83.

3.2 Q2: BiLSTM Regularization Robustness

We experimented with dropout rates of 0.2, 0.4, and 0.6.

Dropout Rate	Training Loss	Validation F1
0.2	**0.3144**	**0.8464**
0.4	0.3344	0.8239
0.6	0.4171	0.7950

Table 2: Effect of Dropout on BiLSTM Generalization.

Optimal Configuration: The model with **Dropout = 0.2** performed best, achieving the lowest loss and highest validation F1. Higher dropout rates (0.6) underfitted the data, as evidenced by the significantly higher training loss (0.4171).

3.2.1 Robustness Analysis

The best performing model (Dropout 0.2) was tested on clean data versus noisy data (simulated spelling mistakes/synonyms).

- **Clean Test F1:** 0.8412
- **Noisy Test F1:** 0.7967

The model showed reasonable robustness, dropping only $\approx 4.5\%$ in performance when noise was introduced. This suggests the BiLSTM learned semantic context rather than just memorizing specific keywords.

3.3 Visual Summary

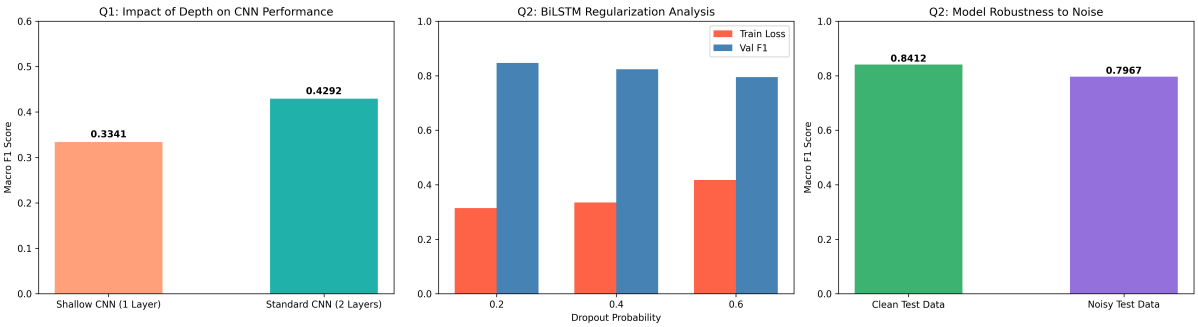

Figure 1: Comparative analysis of CNN Depth (Left), BiLSTM Dropout (Center), and Noise Robustness (Right).

4 Conclusion

- **Vision:** Depth matters. Even a small increase in convolutional layers significantly improves feature extraction capabilities in CNNs.
- **NLP:** For this dataset, lower regularization (Dropout 0.2) was optimal. The BiLSTM architecture proved robust, maintaining high accuracy even when input text contained noise, satisfying a key requirement for reliable AI systems.

A Code Implementation

```
import torch
import torch.nn as nn
import pandas as pd
# ... [Dataset Loading Code] ...

# --- CNN Model ---
class BetterCNN(nn.Module):
    def __init__(self, num_classes, num_conv_layers=2):
        super(BetterCNN, self).__init__()
        self.layer1 = nn.Sequential(nn.Conv2d(3, 32, 3, 1), nn.ReLU(), nn.MaxPool2d(2))
        self.layer2 = nn.Sequential(nn.Conv2d(32, 64, 3, 1), nn.ReLU(), nn.MaxPool2d(2))
        self.fc = nn.Linear(64*4*4, num_classes)
    # ... [Forward Pass] ...

# --- BiLSTM Model ---
class BiLSTM(nn.Module):
    def __init__(self, vocab_size, dropout=0.2):
        super(BiLSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, 100)
        self.lstm = nn.LSTM(100, 64, bidirectional=True, batch_first=True)
        self.fc = nn.Linear(128, 1)
    # ... [Forward Pass] ...

# --- Noise Injection for Evaluation ---
def get_noisy_batch(inputs, noise_level=0.1):
    mask = torch.rand(inputs.shape) < noise_level
    noisy_inputs[mask] = torch.randint(1, vocab_size, inputs.shape)[mask]
    return noisy_inputs
```