Final Project

Mücahit Emre Başoğlu

Istinye University Management Information Systems

MIS027 - Introduction to Deep Learning

Ömer Ozan Evkaya

January 10,2025

Contents

Introduction

In today's world, machine learning and artificial intelligence are advancing at an incredible pace. Among the many approaches within these fields, **Convolutional Neural Networks (CNNs)** stand out as a powerful tool, particularly in image processing and recognition tasks. CNNs have the unique ability to automatically detect and learn important features from images, making them highly efficient for classification problems. Motivated by their success, this project focuses on solving an image classification task using CNNs.

For this purpose, I worked with the **CIFAR-10 dataset**, which is a widely used benchmark in computer vision. The dataset consists of 60,000 color images, each with a resolution of 32x32 pixels, categorized into 10 distinct classes, such as airplanes, automobiles, birds, cats, and more. Out of these, 50,000 images are designated for training, while 10,000 are reserved for testing. One important feature of this dataset is its balanced distribution, ensuring each class is equally represented, which eliminates potential bias during model training.

This project is structured around the following objectives:

Preparing the dataset through normalization and visualization to better understand its structure.

Implementing and training two different CNN architectures to compare their performance.

Evaluating the models using key performance metrics and identifying strengths and weaknesses in their designs.

The primary goal was to explore how different architectural choices in CNNs influence their performance. Specifically, I was curious to see how factors like model depth, number of layers, and the number of trainable parameters affect accuracy and generalization. Through this hands-on approach, I aimed to deepen my understanding of CNNs while gaining practical experience in solving real-world image classification problems.

In this report, I will discuss the details of the methodology, the models implemented, and the training process. I will then present the results, comparing the performance of the two architectures, followed by an analysis of the key findings and their implications. Finally, I will share insights gained during the process and suggest directions for future work.

Methodology

The methodology for this project was carefully structured into three main phases: **data preparation**, **model design**, and **training/evaluation**. Each step was essential to ensure a robust and well-analyzed pipeline for image classification using the CIFAR-10 dataset. Below, I provide a detailed breakdown of the entire process.

Data Preparation

The CIFAR-10 dataset was the foundation of this project, consisting of **60,000 images** (50,000 for training and 10,000 for testing), categorized into **10 distinct classes** such as airplanes, cars, and birds. Each image was of size **32x32 pixels with 3 color channels (RGB)**, making it suitable for convolutional neural networks. Here's how I prepared the data:

1. **Loading the Data**:
    - The dataset was loaded using TensorFlow's keras.datasets.cifar10 module.
    - Upon loading, the dataset was split into training and test sets, ensuring that the training set was used exclusively for learning and the test set for evaluation.

```
In [4]: (train_images, train_labels), (test_images, test_labels) = data.load_data()

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 15s 0us/step
```

2. **Normalization**:

- Since the pixel values ranged between 0 and 255, I normalized them to a 0-1 range by dividing each pixel value by 255. This step is crucial for improving the numerical stability of the model and speeding up convergence.
- Normalization helps the optimizer perform better, especially when dealing with large datasets or deep models.


```
[16]: train_images, test_images = train_images / 255.0, test_images / 255.0
```

```
[17]: model = tf.keras.models.Sequential([
```

3. **Exploratory Data Analysis (EDA)**:

- Before diving into model design, I conducted a quick EDA to visualize the dataset. Using Matplotlib, I displayed some random images to verify their quality and diversity across the classes. This also ensured that the data distribution was balanced.

4. **Insights from Data Preparation**:

- I ensured there were no missing labels or corrupted images in the dataset.
- Visual inspection confirmed that each class had sufficient representation, which is critical for training balanced models.

Model Architecture 1: The Baseline CNN

The first CNN model served as a baseline. It was deliberately kept simple to ensure the pipeline was functioning correctly before introducing complexity. The architecture comprised the following layers:

1. **Input Layer**:
    - The input layer was defined to accept images of size (32x32x3).
2. **Convolutional Layers**:
    - Three convolutional layers were used with **ReLU activation** to extract spatial features.
    - Filter sizes started at 32 and increased to 64 in subsequent layers.
    - A kernel size of (3x3) was chosen for feature extraction.
3. **Pooling Layers**:
    - Max-pooling layers followed each convolutional layer to downsample the feature maps and reduce computational cost.
4. **Dense Layers**:
    - After flattening the feature maps, I added a fully connected layer with **128 neurons**.
    - The output layer used **softmax activation** to generate probabilities for the 10 classes.

```
In [17]: model = tf.keras.models.Sequential([
                        tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
                        tf.keras.layers.MaxPooling2D(2, 2),
                        tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
                        tf.keras.layers.MaxPooling2D(2, 2),
                        tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
                        tf.keras.layers.Flatten(),
                        tf.keras.layers.Dense(128, activation='relu'),
                        tf.keras.layers.Dense(10, activation='softmax')])
```

5. **Compilation**:

- The model was compiled using the Adam optimizer, sparse categorical cross-entropy as the loss function, and accuracy as the evaluation metric.

6. **Purpose**:

- This baseline model allowed me to verify the effectiveness of the basic architecture and served as a reference for comparison with more complex models.

Training and Evaluation

Both models were trained for 20 epochs with the following settings:

Batch Size: 32 (small enough to fit into GPU memory while retaining sufficient gradient accuracy).

Optimizer: Adam, chosen for its adaptability and computational efficiency.

Loss Function: Sparse categorical cross-entropy, suitable for multi-class problems.

Validation Data: Test set used during training to monitor performance.

To evaluate the models:

Accuracy and Loss: Tracked for both training and validation sets across epochs.

Comparative Analysis: I compared the training curves to identify overfitting and assess whether the deeper model improved performance.

Insights from Methodology

The implementation of both baseline and deeper architectures provided key insights into the role of model complexity, training dynamics, and optimization techniques:

1. **Baseline Model Effectiveness**:
    - The baseline model served as a reliable starting point, demonstrating that even a relatively shallow network could achieve acceptable accuracy on the CIFAR-10 dataset.
    - Its straightforward architecture ensured fast training times and minimal computational overhead, making it a practical choice for initial experimentation.
2. **Impact of Deeper Architectures**:
    - The deeper model, with additional convolutional layers and parameters, significantly improved its ability to capture complex features from the dataset.
    - This enhancement in feature extraction translated into better generalization on unseen data, as observed from improved validation accuracy.
    - However, this improvement came at the cost of increased training times and memory usage, highlighting the trade-off between performance and computational efficiency.
3. **Regularization Techniques**:
    - The use of **dropout layers** effectively reduced overfitting by ensuring the model did not rely too heavily on specific neurons during training.
    - **Batch normalization** played a critical role in stabilizing the learning process, particularly in the deeper architecture, by normalizing layer inputs and allowing the network to train faster and with fewer epochs.
    - Without these techniques, the deeper model exhibited signs of overfitting early in the training phase.
4. **Optimization and Loss Dynamics**:
    - The Adam optimizer proved to be a robust choice for both architectures, efficiently handling gradient updates and achieving smooth convergence.
    - Sparse categorical cross-entropy effectively handled the multi-class classification task, ensuring the loss function aligned well with the problem requirements.
5. **Scalability of the Pipeline**:

- The modular design of the pipeline ensured that changes, such as adding layers or incorporating additional techniques, were seamlessly integrated.
- This flexibility is particularly important when exploring further optimizations or experimenting with additional datasets and models in the future.

These insights underline the importance of starting simple, analyzing the effectiveness of fundamental designs, and incrementally building complexity to improve performance while addressing challenges like overfitting and computational constraints.

Results

Two different models were trained and evaluated on the dataset to compare their performance. Both models underwent 20 epochs of training, and the validation accuracy and loss were tracked throughout the process.

Model 1:

Initial performance: Accuracy started at 37.15%, and validation accuracy was 53.71%. The corresponding validation loss was 1.2899.

Improvement trend: Accuracy increased steadily across epochs, reaching a final training accuracy of 93.37%. However, the validation accuracy peaked at 71.86% in epoch 11 and declined slightly afterward.

Overfitting observation: The validation loss began to increase after epoch 7, indicating potential overfitting. Despite high training accuracy, the final validation accuracy was 70.56%, with a loss of 1.4247.

Model 2:

Initial performance: Accuracy started at 33.46%, with a validation accuracy of 58.95% and a validation loss of 1.1538.

Improvement trend: Accuracy improved steadily to 93.31% by the final epoch. Validation accuracy reached 78.15% in epoch 17, surpassing Model 1's best result.

Overfitting observation: Validation loss began increasing significantly after epoch 8. While validation accuracy remained relatively stable, the final validation accuracy was 77.26%, with a loss of 1.1483.

Comparison:

Validation accuracy: Model 2 outperformed Model 1 in terms of peak validation accuracy (78.15% vs. 71.86%).

Overfitting: Both models exhibited overfitting, but Model 2 maintained a higher validation accuracy throughout training.

Training time: Model 2 required significantly longer training time per epoch (36-38 ms/step) compared to Model 1 (3 ms/step).

These results highlight that while Model 2 achieved better generalization, its longer training time may limit scalability for larger datasets or real-time applications. Further optimization and regularization techniques could help mitigate overfitting and improve performance.