Report

Homework 2

Goal: Image Classification with CNN for a self driving car in OpenAI Gym

Andrea Massignan
1796802

GitHub Repository

1 Introduction

In this homework assignment, the objective is to address the image classification problem focused on understanding the behavior of a racing car within the OpenAI Gym environment. The task involves classifying 96x96 color images, each corresponding to one of the five distinct actions available for controlling the car.

2 Methodology

The image classification task is addressed using convolutional neural network (CNN) models. The CNN architecture is implemented using the Keras API with a TensorFlow backend. The models are trained on the labeled training set and evaluated on the separate test set.

Here is how I approached the problem.

3 Data Preprocessing

In this section, I describe the data preprocessing steps implemented for training a Convolutional Neural Network (CNN) to classify images in the context of a self-driving car simulation. The preprocessing pipeline makes so that the data is appropriately formatted and augmented for robust training while maintaining consistency for validation.

The dataset was split into a training set and a validation set, each set contains images organized into subdirectories corresponding to their respective classes.

To handle the variability in image sizes and ensure compatibility with the CNN architecture, all images were resized to a target size of 96×96 pixels. The images were loaded in RGB format, which preserves the color information for classification. The batch size was set to 64.

3.1 Training Data Augmentation

To enhance the generalizability of the model, data augmentation was applied to the training set using the `ImageDataGenerator` class from Keras. Augmentation techniques included:

- **Rescaling**: All pixel values were normalized to the range [0, 1] by dividing by 255.
- **Zooming**: A random zoom range of 10% was applied to simulate variations in image scale.
- **Shifting**: Random shifts up to 10% in both width and height directions were introduced.
- **Horizontal Flipping**: Random horizontal flips were applied to account for variations in vehicle orientation.
- **Rotation**: Images were randomly rotated within a range of 20 degrees.
- **Shearing**: Shear transformations with a range of 10% were used to modify the image perspective slightly.
- **Fill Mode**: When augmentations resulted in empty areas, pixels were filled using the nearest pixel values.

3.2 Validation Data Processing

For the validation set, data augmentation was intentionally omitted to preserve the original data distribution and provide an unbiased evaluation of the model's performance. The images in this set were rescaled to the range [0, 1].

3.3 Data Generators

The training and validation data were loaded using generators, which allow for efficient batch processing:

- `train_generator`: Reads the augmented training images and their corresponding labels in batches.
- `validation_generator`: Loads validation images without augmentation for consistent evaluation.

3.4 Dataset Overview

The preprocessing pipeline gave this insights into the dataset:

- Input images have a shape of $96 \times 96 \times 3$, where the dimensions correspond to height, width, and color channels, respectively.
- The training set comprises `train_generator.n` samples distributed across `train_generator.num_classes` classes.
- The validation set consists of `validation_generator.n` samples spanning the same number of classes as the training set.
- The class names, derived from the directory structure, are: `train_generator.class_indices.keys()`.

4 Class Distribution Analysis

To better understand the distribution of samples across different classes in the training dataset, I performed an analysis of the class distribution. Uneven distribution of samples among classes, often referred to as class imbalance, can negatively affect the training process, leading to biased predictions. This analysis helps identify potential imbalances that may require corrective measures.

The number of samples per class was computed using the `train_generator.classes` attribute, which provides the class labels for all images in the training set. The class names were retrieved from the generator's `class_indices` attribute. A bar plot was generated to visualize the distribution.

4.1 Visualization of Class Distribution

The figure in the next page displays the class distribution in the training set. Each bar corresponds to a class, and its height represents the number of samples available for that class. The visualization was created using `matplotlib` this way:

- The total count of samples per class was calculated by summing the occurrences of each class label.
- A bar chart was plotted, where the x-axis represents the class names, and the y-axis represents the number of samples.

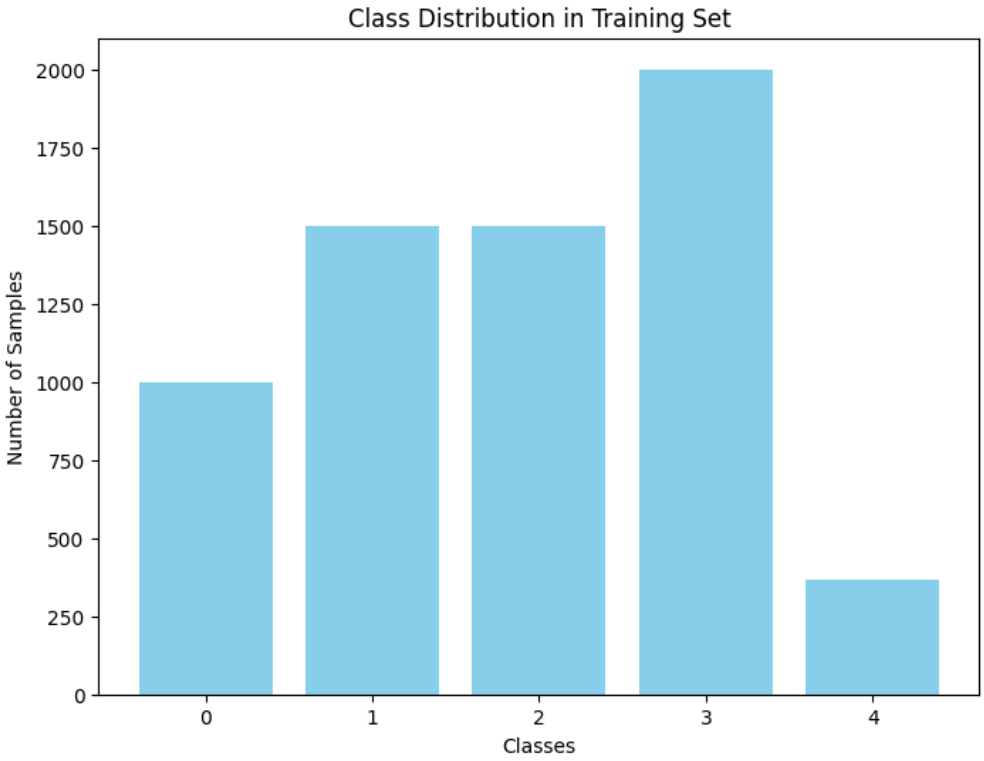


Figure 1: Class distribution in the training set.

4.2 Observations

The class distribution in the training dataset, as depicted in Figure 1, reveals variability in the representation of different classes:

- **Class 3** is the most represented, with approximately 2000 samples.
- **Classes 1 and 2** are moderately represented, each containing around 1500 samples.
- **Class 0** has fewer samples, with just above 1000 samples.
- **Class 4** is significantly underrepresented, containing fewer than 500 samples.

This distribution indicates an imbalance in the dataset, with Class 4 being particularly underrepresented and Class 3 dominating the dataset. Such imbalances may lead to biased model performance, favoring overrepresented classes and leading to overfitting

4.3 Creating Separate Generators for Minority Classes

To address the observed class imbalance in the dataset, I implemented a strategy to augment the minority classes separately. This approach focuses on increasing the representation of the underrepresented classes in the training data, and this is done by re-applying the augmentation techniques to the images in these classes. The goal is to balance the class distribution and improve the model's ability to generalize across all classes.

4.3.1 Target Classes and Data Augmentation

The classes identified as minority classes in the dataset were **Class 0** and **Class 4**.

The augmentation techniques applied to the minority classes included **normalizing** pixel values to the range [0, 1], applying random **zooms** up to 20% to simulate changes in scale, introducing random **horizontal and vertical shifts** up to 20%, randomly **flipping** images horizontally, performing random **rotations** up to 30 degrees, applying **shear transformations** up to 20%, and filling any empty areas created during transformations using the **nearest pixel values**.

4.3.2 Implementation

Separate generators were created for the minority classes using the `ImageDataGenerator` class, targeting their respective directories within the training dataset. Only the directories corresponding to the minority classes (`Class 0` and `Class 4`) were processed. Each generator was configured to shuffle the data and output augmented samples in batches of 64 images, resized to 96×96 pixels.

4.3.3 Outcome

For each minority class, a generator was successfully created, provided that the class directory existed in the training dataset. If a directory was not found, the process skipped that class with a notification.

4.4 Computing Class Weights

To further address the issue of class imbalance in the training dataset, class weights were computed. Class weights help the model prioritize learning from underrepresented classes by assigning higher importance to their corresponding loss during training. This ensures that the model does not become biased toward overrepresented classes.

4.4.1 Methodology

The class weights were calculated using the `class_weight.compute_class_weight` function from `sklearn.utils`. This function computes the weights inversely proportional to the frequency of each class, ensuring that all classes are equally considered during training. The computation steps are as follows:

- The unique class labels were extracted from `train_generator.classes`.
- The class weights were calculated using the formula:

$$\text{weight}_i = \frac{\text{total samples}}{\text{number of classes} \times \text{samples in class}_i}$$

- The weights were converted into a dictionary for easy integration during model training.

4.4.2 Results

The computed class weights were:

Class Weights: `class_weights_dict`

These weights assign higher values to underrepresented classes (e.g., Class 4), while overrepresented classes (e.g., Class 3) receive lower weights.

4.4.3 Significance

By incorporating class weights during model training, the loss function will penalize misclassifications of underrepresented classes more heavily. This adjustment should help to improve the model's performance on minority classes without artificially modifying the dataset or oversampling.

In this case specifically, the class weights are the following:

```
Class Weights: {0: 1.2738, 1: 0.8492, 2: 0.8492, 3: 0.6369, 4: 3.4520}
```

4.5 Combining Original and Augmented Generators

To ensure a comprehensive dataset for training, the original training generator was combined with the augmented generators created for the minority classes. This approach effectively merges the balanced representation from the augmentation process with the natural variability of the original dataset, improving the overall robustness of the model.

4.5.1 Methodology

The combination was implemented using the `itertools.chain` function, which allows for the seamless concatenation of multiple generators. These steps were performed:

- The original training generator (`train_generator`) was combined with the augmented generators created for the minority classes.
- A wrapper generator function, `combined_generator`, was defined to yield data batches from the combined generator.
- The iterator for the combined generator was reset to ensure a clean start during training.

4.5.2 Advantages

The combined generator should offer:

- **Increased Diversity**: Combines naturally occurring samples with augmented samples, providing more diverse training data.
- **Class Balance**: Includes additional samples for minority classes, mitigating class imbalance.
- **Seamless Integration**: Maintains the batch-wise data flow required for efficient training in deep learning frameworks.

5 Model Definition

In this section, I define two Convolutional Neural Network (CNN) architectures, thought for the image classification task: a `First CNN` and a `Second CNN`. Both models leverage multiple convolutional blocks to extract hierarchical features from input images, followed by dense layers for classification. These architectures differ in their complexity, optimization strategies, and regularization techniques.

5.1 First CNN Architecture

The `First CNN` model is a sequential architecture consisting of three convolutional blocks, each followed by pooling and dropout layers to reduce overfitting. The key components are as follows:

- **First Convolutional Block:** Includes two convolutional layers with 32 filters of size 3×3 and ReLU activation, followed by max pooling (2×2) and dropout (30%).
- **Second Convolutional Block:** Contains two convolutional layers with 64 filters of size 3×3 and ReLU activation, followed by max pooling and dropout (30%).
- **Third Convolutional Block:** Comprises two convolutional layers with 128 filters of size 3×3 and ReLU activation, followed by max pooling and dropout (40%).
- **Dense Layers:** A fully connected layer with 512 units and ReLU activation is followed by a 50% dropout layer. The output layer uses a softmax activation to classify the images into the desired number of classes.

The model is compiled with the RMSprop optimizer and a learning rate of 0.001, employing categorical cross-entropy as the loss function. This architecture serves as a strong baseline for the task.

5.2 Second CNN Architecture

The `Second CNN` model builds on the `First CNN` architecture by incorporating several enhancements aimed at improving performance:

- **Batch Normalization:** Added after each convolutional layer to stabilize learning and accelerate convergence.
- **Leaky ReLU Activation:** Replaces the standard ReLU activation, allowing for small gradients even for negative inputs, which prevents neuron "dying."
- **Global Average Pooling:** Used instead of flattening after the convolutional blocks, reducing the risk of overfitting by minimizing the number of parameters.
- **Kernel Regularization:** A regularization term ($L2$) is applied to the dense layer to further mitigate overfitting.
- **Adam Optimizer:** A more advanced optimization algorithm with a learning rate of 0.0001, offering adaptive learning rates for better convergence.

The architecture retains three convolutional blocks but augments each with the additional features described above. Dropout is applied at various points to introduce regularization.

6 Model Training

The training phase involves optimizing the `First CNN` and `Second CNN` models using the preprocessed dataset and the combined generators. This section outlines the training strategy, including the use of callbacks for dynamic learning rate adjustment, early stopping, and model checkpointing.

6.1 Callbacks

To monitor and enhance the training process, three key callbacks were employed:

- **Early Stopping:** This callback monitors the validation loss and stops training if there is no improvement for 10 consecutive epochs. Additionally, it restores the model weights corresponding to the best validation performance to prevent overfitting.
- **ReduceLROnPlateau:** This callback reduces the learning rate by a factor of 0.2 if the validation loss does not improve for 3 consecutive epochs, ensuring gradual convergence.
- **Model Checkpoint:** The best model, based on validation accuracy, is saved to a file (`models/best_model.keras`) during training. This ensures the best performing model is preserved even if training is interrupted.

6.2 Training Configuration

The training process was configured with this parameters:

- **Epochs:** Both models were trained for up to 100 epochs, with early stopping enabled to terminate training when no further improvement is observed.
- **Steps Per Epoch:** Calculated as the total number of samples in the training set divided by the batch size, rounded up to ensure all samples are utilized.
- **Validation Steps:** Similarly, computed for the validation set.
- **Class Weights:** The previously computed class weights were applied to emphasize learning from underrepresented classes.

If the training process was interrupted manually (e.g., via a keyboard interrupt), the models were saved at their current state to avoid loss of progress.

6.3 Model Saving

Upon completion of the training, the final versions of both models were saved to disk for future use:

- **First CNN:** Saved as `models/final_model_deep.keras`.
- **Second CNN:** Saved as `models/final_model_improved.keras`.

7 Creating and Evaluating the Combined Model

To leverage the strengths of both the `First CNN` and `Second CNN` architectures, a combined model was created by averaging the predictions of the two individual models.

7.1 Architecture of the Combined Model

The combined model was constructed as follows:

- **Input Layer:** A single input layer was defined with the shape matching the input dimensions of both `First CNN` and `Second CNN`. To ensure compatibility, the input shapes of the two models were verified to be identical.
- **Predictions from Individual Models:** The predictions from both `First CNN` and `Second CNN` were obtained by passing the input layer through each pre-trained model.
- **Averaging Layer:** The outputs of the two models were combined using an averaging layer, which computes the element-wise mean of the predictions. This ensemble strategy smooths out individual model errors and improves robustness.
- **Output Layer:** The averaged predictions serve as the final output of the combined model, representing the probabilities for each class.

7.2 Compilation and Saving

The combined model was compiled with this configuration:

- **Loss Function:** Categorical cross-entropy was used as the loss function, consistent with the individual models.
- **Optimizer:** The Adam optimizer with a learning rate of 0.0001 was selected for its adaptive learning capabilities.
- **Metrics:** Accuracy was used as the primary evaluation metric.

7.3 Model Evaluation Summary

The results, including the loss, accuracy, and classification metrics, are summarized here:

7.3.1 Validation Metrics

Model	Loss	Accuracy
First CNN	1.3909	0.6162
Second CNN	1.3409	0.6744
Combined CNN	1.3079	0.7159

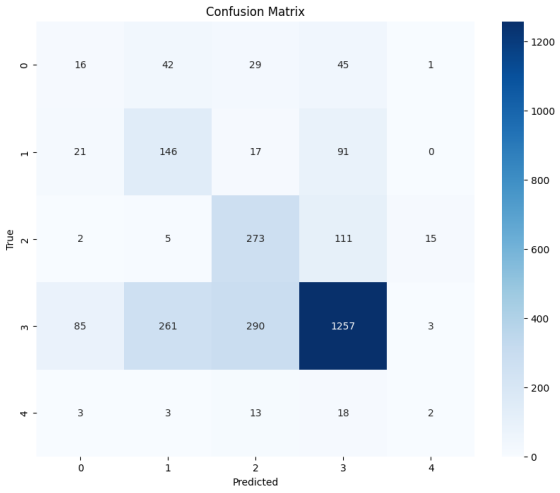
Table 1: Validation Loss and Accuracy for All Models

7.3.2 Classification Reports

The detailed classification reports for each model, which provide precision, recall, F1-score, and support for all classes, are presented in the following tables.

	precision	recall	f1-score	support
0	0.126	0.120	0.123	133
1	0.319	0.531	0.399	275
2	0.439	0.672	0.531	406
3	0.826	0.663	0.736	1896
4	0.095	0.051	0.067	39
accuracy			0.616	2749
macro avg	0.361	0.408	0.371	2749
weighted avg	0.674	0.616	0.633	2749

Classification Report

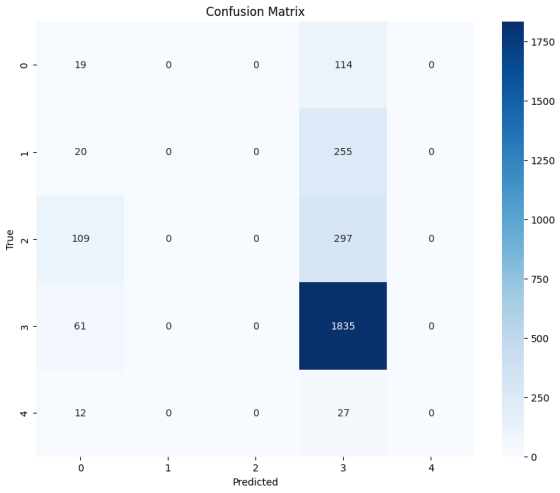


Confusion Matrix

Table 2: Evaluation Metrics for the First CNN

	precision	recall	f1-score	support
0	0.086	0.143	0.107	133
1	0.000	0.000	0.000	275
2	0.000	0.000	0.000	406
3	0.726	0.968	0.830	1896
4	0.000	0.000	0.000	39
accuracy			0.674	2749
macro avg	0.162	0.222	0.187	2749
weighted avg	0.505	0.674	0.577	2749

Classification Report

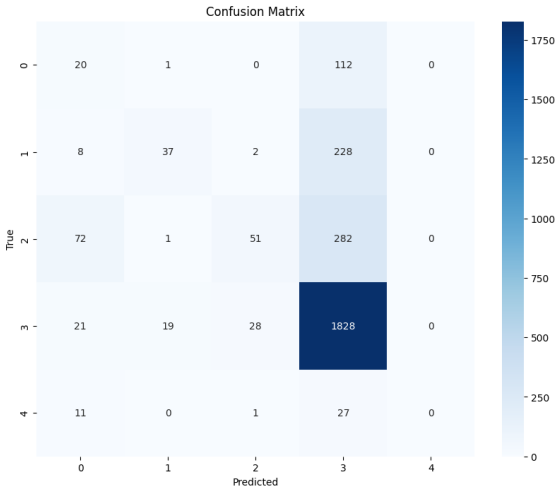


Confusion Matrix

Table 3: Evaluation Metrics for Second CNN

	precision	recall	f1-score	support
0	0.152	0.150	0.151	133
1	0.638	0.135	0.222	275
2	0.622	0.126	0.209	406
3	0.738	0.964	0.836	1896
4	0.000	0.000	0.000	39
accuracy			0.704	2749
macro avg	0.430	0.275	0.284	2749
weighted avg	0.672	0.704	0.637	2749

Classification Report



Confusion Matrix

Table 4: Evaluation Metrics for the Combined CNN

7.3.3 Observations

- The **First CNN** achieved the lowest accuracy (61.62%) and the highest loss (1.3909), indicating its comparatively limited capacity to generalize on the validation set.
- The **Second CNN** outperformed the `First CNN`, achieving an accuracy of 67.44% and a slightly lower loss of 1.3409, highlighting the benefits of batch normalization, Leaky ReLU, and regularization techniques.
- The **Combined Model** achieved the highest validation accuracy of 71.59%, with the lowest loss of 1.3079, demonstrating its effectiveness in leveraging the strengths of both `First CNN` and `Second CNN`.

7.4 Model Deployment in Gymnasium Environment

The best-performing model was deployed in the `CarRacing-v3` environment from Gymnasium. The goal was to control the car using the model's predictions, mapping the classification outputs to predefined continuous actions, even though it was an optional task.

7.4.1 Environment Setup

The environment was initialized using this parameters:

- **Domain Randomization:** Disabled to maintain consistent environmental conditions.
- **Render Mode:** Set to `rgb_array` to enable observation rendering.

7.4.2 Predefined Actions

A set of continuous actions was predefined to control the car based on the model's predictions. These actions include no action, steering left, steering right, accelerating, and braking.

7.4.3 Performance in the Environment

The car was controlled using this approach:

- Observations from the environment were preprocessed and fed into the trained model.
- The model's predicted class was mapped to a corresponding predefined action.

- The environment was stepped with the selected action until the episode terminated.

The interaction with the environment was recorded as a video for further analysis and demonstration purposes.

7.5 Results

Alas, the model was not able to control the car effectively in the Gymnasium environment. The car often veered off the track or failed to navigate turns, indicating a lack of robustness in the model's predictions. The performance was inconsistent, with occasional successes but frequent failures in controlling the car effectively. I'm afraid this is due to the overfitting, which even if mitigated by the ensemble model, was still present in the final model, due to the lack of data and the class imbalance.

8 Reinforcement Learning Approach

An optional component of this project involved implementing a reinforcement learning (RL) approach to solve the `CarRacing-v3` task using a Deep Q-Network (DQN) agent. This approach was designed to train a neural network to autonomously control the car by interacting with the environment and learning from its actions. However, due to hardware limitations and the extensive training time required, this component was not completed to its full extent.

8.1 Overview of the Reinforcement Learning Approach

The RL implementation followed a classical DQN architecture with the following elements:

- **Deep Q-Network (DQN):** A convolutional neural network that predicts Q-values for a set of predefined continuous actions.
- **Action Space:** A set of continuous steering, gas, and brake actions was defined for controlling the car.
- **Replay Memory:** A memory buffer to store past experiences, which were replayed during training to stabilize learning.
- **Target Model Updates:** A secondary target model was used to stabilize training by providing consistent Q-value estimates.

8.2 Implementation Details

The implementation included utility functions for preprocessing frames, seeding randomness, and saving results, along with a configuration class to manage hyperparameters. The training loop leveraged a reward system to guide the agent's learning and included mechanisms to skip frames for efficiency. A test function was also defined to evaluate a trained model in a live environment.

8.3 Challenges and Limitations

These were the challenges that were encountered during the implementation:

- **Hardware Constraints:** The training process requires significant computational resources due to the large state space and the need for extensive interactions with the environment.
- **Training Time:** DQN training is time-intensive, and achieving convergence would require a lot of time of uninterrupted computation.
- **Incomplete Execution:** Due to the aforementioned constraints, the RL approach was implemented but not fully executed to produce conclusive results.

8.4 Inspired by Autonomous Networking Course

This approach was inspired by an example presented during the *Autonomous Networking* course last year. It was demonstrated the application of reinforcement learning for autonomous control tasks, which provided a foundation and inspiration for the design of this RL agent.

8.5 Future Work

With access to better hardware resources and more time, we could improve the implementation, for example:

- Full training and testing of the DQN agent to validate its performance in the `CarRacing-v3` environment.
- Exploration of advanced RL algorithms, such as Double DQN.
- Integration of domain-specific reward shaping to better guide the agent's learning process.

9 Conclusion

In this project, I addressed the image classification task for a self-driving car in the OpenAI Gym environment using convolutional neural networks. I implemented two CNN architectures, the `First CNN` and `Second CNN`, and combined them to create an ensemble model. The combined model achieved the highest validation accuracy of 71.59%, outperforming the individual models.

However, the model's performance in the Gymnasium environment was suboptimal, indicating the need for further improvements to enhance robustness and generalization.

The reinforcement learning approach using a DQN agent was partially implemented but not fully executed due to hardware constraints and training time limitations. Future work could focus on completing the RL implementation and exploring advanced algorithms to improve the agent's performance.