Report

Homework 2

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

Andrea Massignan
1796802

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 [1]. The task involves classifying 96x96 color images, each corresponding to one of the five distinct actions available for controlling the car.

The provided dataset is segregated into training and test sets, with each image accompanied by a label indicating the specific action associated with the car's control. The dataset structure is organized into folders, where each folder is labeled with the identifier of the corresponding action.

The primary goal of this assignment is to develop an image classification model capable of accurately predicting the action taken by the racing car based on the visual input. This entails training a model on the labeled training set and evaluating its performance on the separate test set.

The classification task is significant in understanding and predicting the decision-making process of an autonomous agent navigating a simulated racing environment. Effective solutions to this problem can have applications in various domains, such as autonomous vehicle control and reinforcement learning.

2 Methodology

The image classification task is addressed using a convolutional neural network (CNN) model. The CNN architecture is implemented using the Keras API [2] with a TensorFlow backend [3]. The model is trained on the labeled training set and evaluated on the separate test set.

Here is how I approached the problem.

3 Data Preprocessing

For the training set, data augmentation is performed to increase the model's robustness and improve its generalization capabilities. The ImageDataGenerator is configured with the following augmentation parameters:

- **Rescaling:** Normalizes pixel values to the range [0, 1] by dividing each pixel value by 255.
- **Zoom Range:** Randomly applies zooming with a range of 0.1 to introduce variations in image scale.
- **Width Shift Range:** Randomly shifts the width of the image by a fraction of 0.1 to introduce horizontal variations.

- **Height Shift Range:** Randomly shifts the height of the image by a fraction of 0.1 to introduce vertical variations.
- **Horizontal Flip:** Randomly flips images horizontally to diversify training samples.
- **Vertical Flip:** No vertical flipping is applied to maintain the orientation of the racing car images.

The augmented training data is generated in batches using the `flow_from_directory` method of the ImageDataGenerator. The generator is configured to read images from the 'trainingset' directory, resize them to a target size of (96, 96), use RGB color mode, generate batches of size 64, apply one-hot encoding for categorical labels, shuffle the data, and consider it as part of the training subset.

3.1 Validation Data Preprocessing

For the validation set, no data augmentation is applied to maintain the integrity of the original images. The ImageDataGenerator for validation is configured only with rescaling, ensuring consistency with the preprocessing applied during training.

The validation data generator is created using the `flow_from_directory` method, reading images from the 'validationset' directory, resizing them to (96, 96), and generating batches of size 64. The validation data is not shuffled, and class mode is set to 'categorical' for compatibility with the model.

3.2 Dataset Information

After configuring the generators, relevant information about the dataset is extracted, including the number of training samples, number of classes, and the input shape of the images. The class names are obtained from the generator's class indices. The extracted information is printed to the console for reference.

4 Convolutional Neural Network (CNN) Architecture

In this section, we present the architecture of the Convolutional Neural Network (CNN) designed for the image classification problem in the context of racing car behavior. The CNN model is implemented using Keras, consisting of multiple convolutional and dense layers to capture hierarchical features from the input images.

4.1 Model Architecture

The CNN model is constructed using the Sequential API in Keras, allowing the sequential stacking of layers. Let's break down each layer of the model:

1. **C1 Convolutional Layer:**
   The first convolutional layer consists of 15 filters with a kernel size of (4, 4). Rectified Linear Unit (ReLU) activation function is applied to introduce non-linearity. This layer processes the input image to capture low-level features.

2. **C2 Convolutional Layer:**
   The second convolutional layer comprises 20 filters with a kernel size of (4, 4). ReLU activation is applied, followed by a max-pooling layer with a pool size of (2, 2). This layer further extracts hierarchical features and reduces spatial dimensions through pooling.

3. **C3 Convolutional Layer:**
   The third convolutional layer includes 30 filters with a kernel size of (4, 4). ReLU activation is applied, followed by another max-pooling layer. This layer captures higher-level features and reduces spatial dimensions.

4. **Flatten Layer:**
   The flatten layer transforms the 3D tensor output from the convolutional layers into a 1D tensor, preparing it for the subsequent dense layers.

5. **D1 Dense Layer:**
   The first dense layer consists of 128 neurons with ReLU activation. A dropout layer with a rate of 0.4 is introduced to prevent overfitting.

6. **D2 Dense Layer:**
   The second dense layer comprises 96 neurons with ReLU activation. Another dropout layer with a rate of 0.4 is added.

7. **Output Layer:**
   The output layer has neurons equal to the number of classes, with softmax activation to produce class probabilities.
   This layer is responsible for predicting the action label.

4.2 Model Compilation

The model is compiled using the RMSprop optimizer with a learning rate of 0.0001. Categorical crossentropy is chosen as the loss function, as it is suitable for multi-class classification problems. The accuracy metric is used to monitor the model's performance during training.

5 Training Method with Callbacks

In this section, we outline the training methodology used to train the Convolutional Neural Network (CNN) for the racing car behavior classification problem. The training process incorporates specific callbacks to enhance the model's training efficiency and prevent overfitting.

5.1 Callback Definitions

Two crucial callbacks are employed during the training process:

1. **Early Stopping Callback:**
   The `EarlyStopping` callback is configured to monitor the validation loss (`val_loss`). If there is no improvement in validation loss for a consecutive number of epochs (patience), training is terminated early to prevent overfitting. The `restore_best_weights` parameter ensures that the model is restored to its best weights.

2. **Reduce Learning Rate on Plateau Callback:**
   The `ReduceLROnPlateau` callback dynamically adjusts the learning rate during training. It monitors the validation loss and reduces the learning rate (`factor=0.2`) if there is no improvement for a certain number of epochs (patience). The minimum learning rate is set to 1×10^{-6}.

5.2 Training Configuration

The training process is executed within a `try-except` block to handle potential interruptions. The model is trained using the `fit` method, with the following parameters:

- `train_generator` and `validation_generator:` Data generators for training and validation datasets, respectively.
- `epochs=50:` The total number of training epochs.
- `steps_per_epoch:` The number of steps (batches) to process from the training generator in one epoch.
- `validation_data:` Validation generator for evaluating the model's performance during training.
- `validation_steps:` The number of steps (batches) to process from the validation generator in one epoch.
- `callbacks:` A list containing the defined callbacks (`early_stopping` and `reduce_lr`).

5.3 Handling Interrupts

The `except KeyboardInterrupt` block handles potential manual interruptions, allowing for a graceful termination of the training process.

6 Evaluation and Results Analysis

In this section, we discuss the evaluation of the trained Convolutional Neural Network (CNN) model and analyze the obtained results.

6.1 Evaluation Code

The following code is responsible for evaluating the model on the validation dataset:

```
val_steps = validation_generator.n//validation_generator.batch_size + 1
loss, acc = model.evaluate_generator(validation_generator, steps=val_steps)
print('Test loss: %f' % loss)
print('Test accuracy: %f' % acc)
```

- `val_steps`: The variable is calculated to determine the number of steps (batches) needed to cover the entire validation dataset. It ensures that all data is considered during evaluation.
- `loss, acc`: These variables store the computed loss and accuracy values, respectively, during the evaluation.
- `model.evaluate_generator`: This method evaluates the model on the validation generator, considering the specified number of steps (`val_steps`).
- `print('Test loss: %f' % loss)`: Outputs the computed test loss.
- `print('Test accuracy: %f' % acc)`: Outputs the computed test accuracy.

6.2 Results Analysis

The results of the evaluation are as follows:

- `Test loss`: The computed loss on the validation dataset is 1.120741.
- `Test accuracy`: The computed accuracy on the validation dataset is 0.654784.

6.3 Meaning of Results

The test loss provides a quantitative measure of how well the model performs on the validation dataset. A lower loss indicates better performance.

The test accuracy, representing the proportion of correctly classified samples. This value indicates the percentage of correctly predicted labels among all validation samples.

Further analysis and potential improvements can be explored based on these results, such as fine-tuning the model architecture, adjusting hyperparameters, or augmenting the dataset.

We will try to improve by running a second diverse iteration of the CNN.

7 Classification Report Analysis

In this section, we analyze the results of a classification report obtained from a model evaluation.

7.1 Classification Report

The following is the classification report:

	precision	recall	f1-score	support
0	0.500	0.075	0.131	53
1	0.287	0.509	0.367	110
2	0.417	0.512	0.460	162
3	0.810	0.736	0.771	758
4	0.000	0.000	0.000	15
accuracy	-	-	0.638	1098
macro avg	0.403	0.367	0.346	1098
weighted avg	0.674	0.638	0.643	1098

7.2 Interpretation

The classification report provides a detailed overview of the performance of a classification model across different classes. Here is the interpretation of key metrics:

- `precision:` Precision is the ratio of correctly predicted positive observations to the total predicted positives. It is highest for class 3 (0.810), indicating a high proportion of correctly predicted positive samples.
- `recall:` Recall, or sensitivity, is the ratio of correctly predicted positive observations to the all observations in actual class. Class 3 also has the

highest recall (0.736), indicating the model captures a significant portion of actual positive samples for this class.

- `f1-score:` The F1-score is the weighted average of precision and recall. It is particularly useful when the class distribution is imbalanced. Class 3 has the highest F1-score (0.771), reflecting a balance between precision and recall.
- `support:` Support is the number of actual occurrences of the class in the specified dataset. Class 3 has the highest support (758), indicating a relatively large number of samples for this class.
- `accuracy:` Accuracy is the ratio of correctly predicted observations to the total observations. The overall accuracy is 0.638.
- `macro avg:` Macro-averaging computes the metric independently for each class and then takes the average. The macro-averaged precision, recall, and F1-score are 0.403, 0.367, and 0.346, respectively.
- `weighted avg:` Weighted averaging computes the metric for each class and then takes the weighted average based on the support. The weighted-averaged precision, recall, and F1-score are 0.674, 0.638, and 0.643, respectively.

8 Analysis of Car Behavior

In this section, we analyze the behavior of the car in the provided code that utilizes a trained model to control the car in the CarRacing-v2 environment.

8.1 Observations

The observed behavior of the car, is described below:

- **Stability:** The car exhibits a certain degree of stability, as it manages to stay on the right track for the majority of the simulation.
- **Unstable Moments:** Despite overall semi-stability, there are instances where the car exhibits some instability. As sometimes it gets outside of the track, or spins to get back on the correct path.

9 Second Iteration of the CNN

The second CNN model exhibits slight modifications in comparison to the previous model. Key changes include the optimizer, and adjustments in layer parameters.

- **Optimizer:** The model now utilizes the Adam optimizer with a specified learning rate of 0.001. This optimizer is known for its adaptive learning rate and is widely used in deep learning tasks.
- **Convolutional Layers:** The model retains three convolutional layers (C1, C2, C3), each followed by rectified linear unit (ReLU) activation. Filter sizes and pooling configurations have been adjusted for better feature extraction.
- **Pooling Layers:** Max-pooling layers follow the second and third convolutional layers to reduce spatial dimensions and capture significant features.
- **Dropout Layers:** Two dropout layers (D1, D2) are added after dense layers to prevent overfitting. Each dropout layer randomly drops a fraction of input units during training.
- **Global Average Pooling:** A Global Average Pooling layer is omitted in this model. The previous model included it, but it has been excluded here. Global Average Pooling reduces spatial dimensions and provides a form of spatial regularization.

9.1 Training Setup

- **Learning Rate Scheduler:** The model employs a learning rate scheduler that reduces the learning rate by 10% every 10 epochs. This dynamic adjustment can help the model converge more efficiently.

9.1.1 Model Evaluation and Comparison

The evaluation metrics for the second model, trained with a similar CNN architecture and data augmentation, exhibit this kind of results:

Class	Precision	Recall	F1-Score	Support
0	0.400	0.241	0.300	133
1	0.361	0.604	0.452	275
2	0.484	0.724	0.580	406
3	0.846	0.711	0.773	1896
4	0.000	0.000	0.000	39
Accuracy			0.670	2749
Macro Avg	0.418	0.456	0.421	2749
Weighted Avg	0.711	0.670	0.679	2749

9.2 Discussion

The second model demonstrates improvements in precision, recall, and F1-score across multiple classes, indicating enhanced classification performance. The macro and weighted averages also suggest an overall better balance in handling the diverse classes.

It's important to note that the absence of improvement in the recall of class 4 may be attributed to the class imbalance or inherent complexities in distinguishing this particular class.

With the absence of *Early Stopping*, the model is trained for 100 epochs. This decision increases the risk of overfitting, and careful monitoring of training and validation metrics is essential to assess the model's generalization performance.

However this choice is made to see if the model can converge to a better accuracy if trained for more epochs.

Unfortunately, the model does not converge to a much better accuracy, and it is slightly better but similar to the previous model.

This made me think that perhaps the model is uncapable of learning more, and that the dataset is not big enough to train a more complex model.

Also the performance in the driving simulation is not better than the previous model, and the car still exhibits instability.

10 Other Possible Approach I Tried

10.1 Improving the Pre-trained Model with Deep Reinforcement Learning

To enhance the performance of the pre-trained model, a Deep Reinforcement Learning (DRL) approach can be considered. Utilizing frameworks like Keras-RL or other DRL libraries allows the model to learn optimal actions through interaction with the environment.

10.2 Keras-RL Integration

Keras-RL provides seamless integration with Keras models, enabling the application of various DRL algorithms such as Deep Q-Learning (DQN) to our existing model. By incorporating reinforcement learning techniques, the model can learn a policy for controlling the car based on environmental feedback.

10.3 Training with Gym Environment

The Gym environment for CarRacing-v2, used previously for supervised learning, can now serve as a training ground for reinforcement learning. The agent explores actions, receives rewards, and learns to maximize cumulative rewards over time.

10.4 Agent Exploration Strategies

Implementing exploration strategies, such as epsilon-greedy or softmax, allows the agent to strike a balance between exploiting known actions and exploring new ones. This is crucial for discovering optimal policies.

10.5 Sequential Decision Making

Deep Reinforcement Learning enables the model to make sequential decisions, adapting its behavior over time based on feedback from the environment. This iterative learning process can lead to improved performance and robustness.

While DRL introduces additional complexity, it has the potential to significantly enhance the pre-trained model's ability to navigate the CarRacing environment intelligently.

11 Problematics

Integrating a Deep Reinforcement Learning (DRL) agent with our pre-trained model poses certain challenges. One key issue is the compatibility between the DRL frameworks, like Keras-RL, and the existing neural network model.

In my case I struggled to integrate the Keras-RL library with my model, and I was not able to make it work.

More precicely I couldn't pass the right obs as it was continously reshaping the input in the wrong way:

(1, 96, 96, 3) was forcefully transformed to (1, 1, 1, 96, 96, 3).

Thus I wasn't able to make it work, and I had to give up on this approach.