

Sapienza University of Rome



MACHINE LEARNING

COURSE CODE: 1022858

Homework 2.

CNN - Image Classification

Author:

CARMINE FABBRI

January 5, 2024

Contents

1	Introduction	2
2	The dataset	2
2.1	Data preprocessing	3
2.1.1	Tools used	3
2.1.2	Preparing the dataset	3
3	Architecture	3
3.1	Design choice	3
3.2	First CNN model	4
3.2.1	Data preprocessing	4
3.2.2	Model architecture	5
3.2.3	Optimizer	7
3.3	Second CNN model	8
3.3.1	Data preprocessing	8
3.3.2	Model architecture	8
3.3.3	Optimizer	11
4	The results	12
4.1	Performance evaluation metrics	12
4.2	Model performance comparison	13
4.3	First CNN	13
4.3.1	Hyperparameter tuning	14
4.4	Second CNN	15
4.4.1	Hyperparameter tuning	16
4.5	Confusion matrix	17
4.6	Computational training time	18
5	Conclusion	18

1 Introduction

This is my report for the machine learning class's second assignment. To correctly classify images and learn the behavior of a racing car in a Gym environment, I worked on creating a customized CNN architecture. Below is an explanation of the procedures used to pre-process the input data, train the various models, and assess the performance using different methods of evaluation.

2 The dataset

The dataset, which is separated into training and test sets, consists of 96 x 96 color pictures labeled with one of the five possible actions that can be used to operate the vehicle.

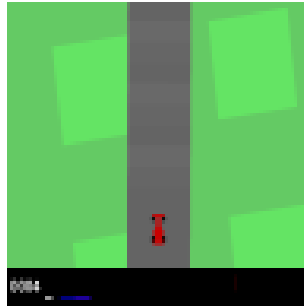


Figure 1: Dataset sample

The dataset is given as collections of photos arranged into folders identified by the action's ID.

The identified actions are the following:

- 0: do nothing
- 1: steer left
- 2: steer right
- 3: gas
- 4: brake

2.1 Data preprocessing

2.1.1 Tools used

Because Python has well-documented libraries and modules, that's why I chose it as a programming language. Instead of starting from scratch, I used widely accessible tools like *Tensorflow*, *Keras*, *Matplot*, and *Numpy* to build, train, and assess the convolutional neural network.

For ease of reading and code composition, I've made the source code available as a Jupyter Notebook (with a.ipynb extension). This method facilitates the creation of code snippets and aids in code understanding. Additionally, the notebook's checkpoints allow you to save partial findings and instantly see debugging details and images.

2.1.2 Preparing the dataset

An essential first step in the creation of neural networks and other machine learning models is data preprocessing. Improving the data's quality and usefulness will help the neural network identify trends and generate precise predictions. An effective neural network model depends critically on proper data preparation, which can also have a big impact on the model's performance.

To improve the effectiveness of the solution, in addition to designing a capable neural network, it is necessary to pre-process the data in such a way as to make the model more accurate and effective.

In the following chapters, the preprocessing methodology used on the data of each model will be described in detail.

3 Architecture

3.1 Design choice

Even though there are many pre-defined architectures or pre-trained models that perform very well, in this homework - for learning purposes - each of the convolutional neural networks is designed in a custom way from the ground.

Considering also the fact that the datasets used for training are characterized by a fair amount of noise, my goal was to define two neural networks that would allow balancing the accuracy obtained on the validation dataset

with the behavior of the vehicle driven through the use of the model in the gym environment.

3.2 First CNN model

In the first CNN in order to extract hierarchical characteristics from the input image, the architecture alternates between convolutional and max-pooling layers. Fully connected layers are then added for classification. The ReLU activation function is used for non-linearity, and the softmax activation in the output layer provides class probabilities for multi-class classification.

3.2.1 Data preprocessing

Augmenting the training dataset is a critical step in training a Convolutional Neural Network (CNN) for image classification. Data augmentation is a method used to add different changes to the pre-existing images to artificially improve the diversity of the training set. For this purpose, I have used the *ImageDataGenerator* method from the Keras library.

A screenshot of a code editor window with a dark background and light-colored text. The code is written in Python and defines a train_datagen object using the ImageDataGenerator class. The parameters specified are rescale=1./255, shear_range=0.2, zoom_range=0.2, and horizontal_flip=True. The code is numbered from 1 to 7.

```
1 # Data augmentation
2 train_datagen = ImageDataGenerator(
3     rescale=1./255,
4     shear_range=0.2,
5     zoom_range=0.2,
6     horizontal_flip=True
7 )
```

Figure 2: Data preprocessing used for the first CNN

Parameters of the Data Augmentation Method:

- *Rescale*: This normalization guarantees that the input data is processed by the neural network on consistent scales, which improves convergence when training
- *Shear range*: A shear range of 0.2 is applied, introducing variations in the orientation of objects within the images

- *Zoom range*: A zoom range of 0.2 is specified, allowing the model to learn from variations in object sizes within the images
- *Horizontal flip*: Enabling this transformation adds horizontally flipped versions of the images to the training set, promoting robustness to changes in object orientation

3.2.2 Model architecture

The architecture alternates between convolutional and max-pooling layers. Fully connected layers are then added for classification. Every pooling layer is of type **MaxPooling2D**, while every convolutional layer is of type **Conv2D**. The final layer uses the *Softmax* activation function, whereas the activation function in the other layers is *ReLU* (*Rectified Linear Unit*). Furthermore, every layer has its *padding* attribute set to 'same' to guarantee that the input and output's spatial dimensions don't change. When there is crucial information that we do not want to lose on the edge of the image, we employ this feature.

- **Input Layer**: This convolutional layer takes in input a shape with the following dimension: (96 x 96 x 3) and then processes the input image using 32 *filters* of size 3x3 (*kernel*), applying the Rectified Linear Unit (ReLU) activation function
- **MaxPooling Layer**: This pooling layer is responsible for reducing the spatial dimensions (2x2) of the input volume while retaining important information. It is a form of down-sampling that helps control the computational complexity of the model and makes the learned features more robust
- **Convolutional Layer**: The second convolutional layer continues to extract higher-level features with 64 *filters* of size 3x3 (*kernel*). ReLU activation is applied, and "same" padding maintains the spatial dimensions
- **MaxPooling Layer**: Another max-pooling layer further reduces the spatial dimensions of the feature maps obtained from the second convolutional layer

- **Convolutional Layer:** The third convolutional layer with 128 *filters* continues to capture more complex and abstract features in the data
- **MaxPooling Layer:** The final max-pooling layer further reduces the spatial dimensions of the feature maps
- **Flatten layer:** Flattens the 3D output to a 1D vector, preparing the data for the fully connected layers
- **Dense layer:** The first fully connected layer with 128 units applies the ReLU activation function to introduce non-linearity
- **Dense layer (Output layer):** The final dense layer with softmax activation produces the output probabilities for the different classes

The total number of parameters is shown below:

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 96, 96, 32)	896
max_pooling2d_6 (MaxPooling2D)	(None, 48, 48, 32)	0
conv2d_7 (Conv2D)	(None, 48, 48, 64)	18496
max_pooling2d_7 (MaxPooling2D)	(None, 24, 24, 64)	0
conv2d_8 (Conv2D)	(None, 24, 24, 128)	73856
max_pooling2d_8 (MaxPooling2D)	(None, 12, 12, 128)	0
flatten_2 (Flatten)	(None, 18432)	0
dense_4 (Dense)	(None, 128)	2359424
dense_5 (Dense)	(None, 5)	645
Total params: 2453317 (9.36 MB)		
Trainable params: 2453317 (9.36 MB)		
Non-trainable params: 0 (0.00 Byte)		

Figure 3: Summary of the first CNN

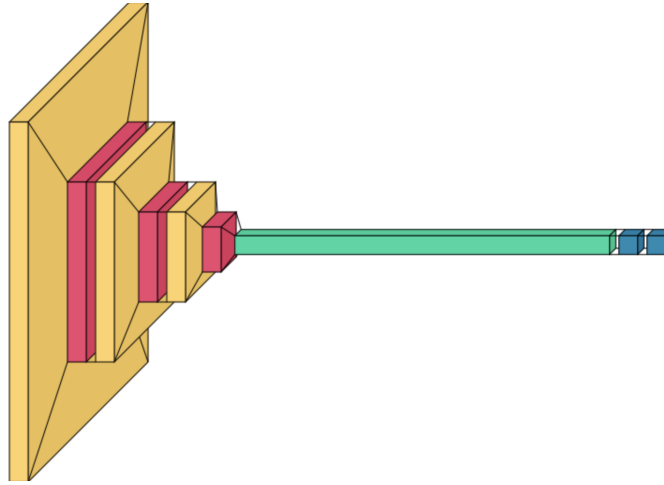


Figure 4: First CNN architecture

From the following image, we can see the CNN architecture and also how the input image is transformed when it goes through the different layers.

The various convolutional layers are represented by the yellow boxes, while the various pooling levels are represented by the red boxes. The flattening layer is then illustrated by using the green box, and in the end, the two last dense layers - represented by the blue - are visible.

3.2.3 Optimizer

An essential part of the training procedure is the choice of the optimizer. Based on the calculated gradients of the loss function with respect to the weights, the optimizer is in charge of updating the neural network's weights. The main objective is to reduce the loss function as much as possible so that the neural network may gradually learn and become more efficient.

CNN performance, convergence, and training speed can all be greatly impacted by the optimizer selection.

For this CNN I have used the **Adam** optimizer. Adaptive Moment Estimation is what Adam stands for. It combines the benefits of *RMSProp* and *AdaGrad*, two additional well-known optimizers. Adam's adaptive learning rates for every parameter are among its most important characteristics. It enables faster convergence by modifying each weight's learning rate according to historical gradients.

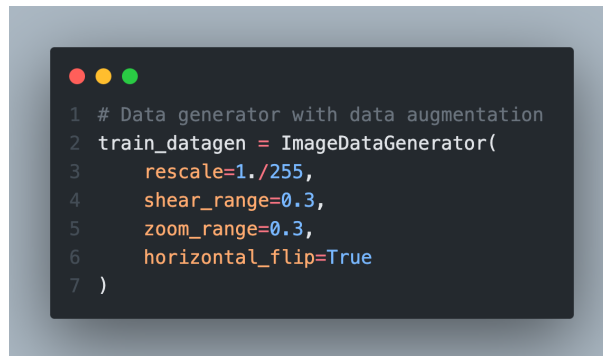
The *Categorical Crossentropy* loss function was employed in the model compilation process, which makes it appropriate for multi-class classification problems in which a sample may belong to more than one class.

3.3 Second CNN model

The second CNN's design alternates between max-pooling and convolutional layers as well. In this instance, to try a different type of architecture, I utilized the *elu* (*Exponential Linear Unit*) for the convolutional layers, the *ReLU* function for the dense layer, and the *softmax* activation function for the resulting output layer.

3.3.1 Data preprocessing

Also in this case the method that I have used to preprocess the data is the *ImageDataGenerator* but with slightly minor differences.

A screenshot of a code editor with a dark background and light-colored text. The code is written in Python and defines a train_datagen object using ImageDataGenerator. The code includes parameters for rescaling, shearing, zooming, and horizontal flipping. The code is as follows:

```
1 # Data generator with data augmentation
2 train_datagen = ImageDataGenerator(
3     rescale=1./255,
4     shear_range=0.3,
5     zoom_range=0.3,
6     horizontal_flip=True
7 )
```

Figure 5: Data preprocessing used for the second CNN

3.3.2 Model architecture

In this case, I used the *elu* (*Exponential Linear Unit*) function for the convolutional layers, the *ReLU* function for the dense layer, and the *softmax* activation function for the final output layer to experiment with a different sort of architecture. It is also important to notice, that in this type of architecture, the *padding* property is not used.

- **Input Layer:** This convolutional layer takes in input a shape with the following dimension: (96 x 96 x 3) and then processes the input image using 32 *filters* of size 3x3 (*kernel*), applying ELU activation function. This function is known for mitigating the vanishing gradient problem and allowing negative values, providing smoother learning characteristics.
- **MaxPooling Layer:** Max-pooling with a 2x2 window is applied to reduce the spatial dimensions, capturing the most important features and promoting translational invariance
- **Convolutional Layer:** Another convolutional layer with 32 *filters* is added, extracting higher-level features from the downsampled image
- **MaxPooling Layer:** Another max-pooling layer further reduces spatial dimensions
- **Flatten layer:** Flattens the 3D output to a 1D vector, preparing the data for the fully connected layers
- **Dense layer:** The first fully connected layer with 128 units applies the Rectified Linear Unit (ReLU) activation function, introducing non-linearity
- **Dropout layer:** During training, dropout is used to randomly deactivate 50
- **Second Dense layer:** Another fully connected layer with 64 units and ReLU activation
- **Dense layer (Output layer):** The final dense layer with softmax activation produces probabilities for each class.

The total number of parameters is shown in the summary below:

```
Model: "sequential_6"
```

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 94, 94, 32)	896
max_pooling2d_12 (MaxPooling2D)	(None, 47, 47, 32)	0
conv2d_13 (Conv2D)	(None, 45, 45, 32)	9248
max_pooling2d_13 (MaxPooling2D)	(None, 22, 22, 32)	0
flatten_6 (Flatten)	(None, 15488)	0
dense_15 (Dense)	(None, 128)	1982592
dropout (Dropout)	(None, 128)	0
dense_16 (Dense)	(None, 64)	8256
dense_17 (Dense)	(None, 5)	325

```

Total params: 2001317 (7.63 MB)
Trainable params: 2001317 (7.63 MB)
Non-trainable params: 0 (0.00 Byte)

```

Figure 6: Summary of the second CNN

The second CNN architecture and how the input is elaborated when it goes through the different layers is shown below:

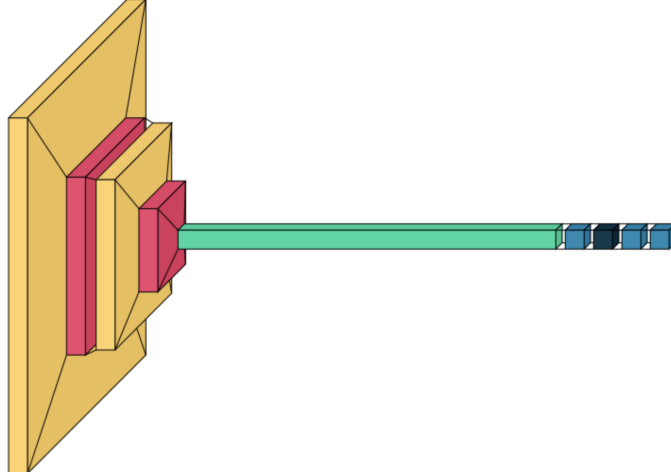


Figure 7: Second CNN architecture

Even in this CNN the yellow boxes represent the different convolutional layers, and the red boxes represent the different pooling levels. The flattening layer is shown using the green box and the two final dense layers, are indicated by the 'light' blue color, finally, the dropout layer is represented by the dark blue box.

3.3.3 Optimizer

RMSprop optimizer was used during the model compilation. RMSprop is an optimization algorithm commonly used in training neural networks. It belongs to the family of adaptive learning rate methods. The key idea behind RMSprop is to adaptively adjust the learning rates for each parameter during training.

During the compilation of the model, the loss function used was the *Categorical Crossentropy* suitable for multi-class classification tasks where each sample can belong to multiple classes.

4 The results

4.1 Performance evaluation metrics

To optimally evaluate the performance of the CNNs, I analyzed fundamental metrics, such as accuracy, f1, precision, and recall, obtained as follows:

- Accuracy: $\frac{\text{True Positives} + \text{True Negatives}}{\text{All Instances}}$
- Recall: $\frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$
- Precision: $\frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$

The first model was trained with 15 epochs, the following graph shows the training accuracy over epochs:

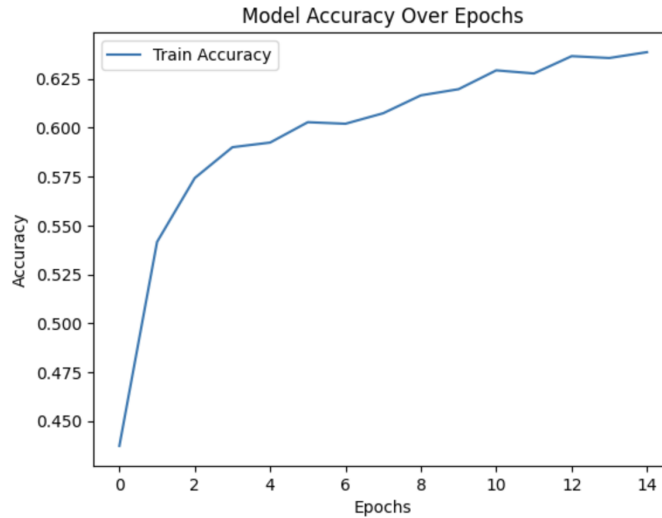


Figure 8: Train accuracy over epochs - First CNN

Instead, the second CNN was trained with only 10 epochs:

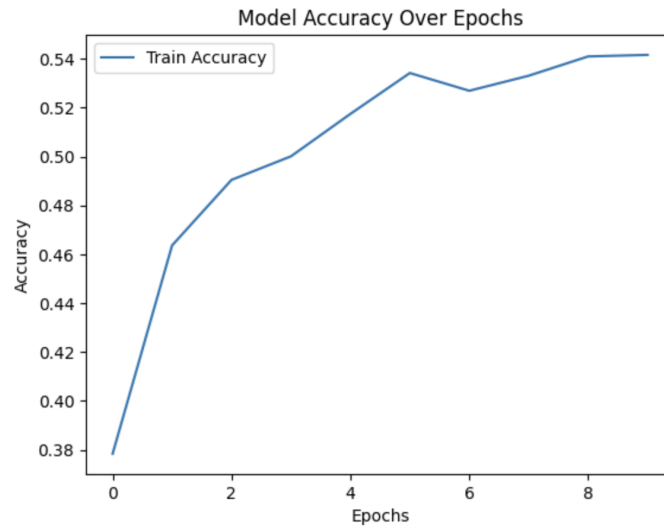


Figure 9: Train accuracy over epochs - Second CNN

4.2 Model performance comparison

The comparison of the different algorithms in each exercise was done not only by taking parameters like precision and recall into account but also by analyzing the behavior in the Gym environment to validate the behavior of the car that uses the trained CNN.

4.3 First CNN

The model exhibited a test accuracy of approximately 60.31%. Let's analyze the performance metrics provided in the classification report:

Test Accuracy: 0.6031284103310295				
Classification Report:				
	precision	recall	f1-score	support
0	0.20	0.50	0.29	133
1	0.35	0.57	0.43	275
2	0.47	0.70	0.56	406
3	0.88	0.60	0.72	1896
4	0.06	0.08	0.06	39
accuracy			0.60	2749
macro avg	0.39	0.49	0.41	2749
weighted avg	0.72	0.60	0.64	2749

Figure 10: First CNN - Performance metrics

The model does reasonably well in categorizing Class 3, with excellent precision and recall. Classes 0 through 2 have poorer precision and recall values, indicating areas for improvement. Class 4 performs extremely poorly across all parameters. The weighted average metrics provide an overall evaluation that takes into account the dataset's class imbalances.

4.3.1 Hyperparameter tuning

Additionally, I experimented with the first CNN using other parameter values, such as:

- *batch_size*: I've changed the size of the batch from 32 \rightarrow 64
- *epochs*: The number of epochs was reduced from 15 \rightarrow 12
- *filter*: I've also increased the number of filters in the penultimate dense layer from 128 \rightarrow 256

The outcome in terms of accuracy was marginally different because of those small hyperparameter adjustments:

Test Accuracy: 0.6187704619861768				
Classification Report:				
	precision	recall	f1-score	support
0	0.35	0.39	0.37	133
1	0.33	0.59	0.42	275
2	0.43	0.79	0.56	406
3	0.86	0.62	0.72	1896
4	0.00	0.00	0.00	39
accuracy			0.62	2749
macro avg	0.39	0.48	0.41	2749
weighted avg	0.71	0.62	0.64	2749

Figure 11: First CNN with hyperparameter changes - Performance metrics

The model's overall accuracy was improved by those hyperparameter adjustments. In actuality, it succeeds well not only when measured by pure performance metrics - as reported in the image above - but also when put to the test in a gym environment.

Both versions of the first CNN model have performed very well in the vehicle gameplay by scoring high scores.

4.4 Second CNN

The second model achieved a test accuracy of approximately 62.71%. Let's delve into the performance metrics provided in the classification report:

Test Accuracy: 0.6271371407784649				
Classification Report:				
	precision	recall	f1-score	support
0	0.39	0.27	0.32	133
1	0.31	0.66	0.42	275
2	0.45	0.77	0.57	406
3	0.87	0.63	0.73	1896
4	0.00	0.00	0.00	39
accuracy			0.63	2749
macro avg	0.40	0.47	0.41	2749
weighted avg	0.71	0.63	0.64	2749

Figure 12: Second CNN - Performance metrics

In particular, the model predicts Class 3 with high precision, recall, and F1-score, and it does so fairly well. Performance in classes 0, 1, and 2 is

inconsistent, with different recall and precision metrics. Unfortunately, class 4 had zero recall and an undefinable F1 score because no positive predictions (true positives) were received in this class. The weighted average metrics offer a comprehensive assessment that takes into account the dataset's class imbalances.

Class 4 is not being predicted correctly, It could be due to a lack of representation in the training data and imbalanced class distribution.

4.4.1 Hyperparameter tuning

I have also tried the second CNN with different values of parameters like:

- *batch_size*: I've changed the size of the batch from 32 \rightarrow 64
- *epochs*: The number of epochs was reduced from 10 \rightarrow 8
- *dropout*: The value of the dropout layer, was reduced from 0.5 \rightarrow 0.3

Thanks to those minor hyperparameter tuning, the result in terms of accuracy was slightly different:

```
Test Accuracy: 0.663150236449618
Classification Report:
```

	precision	recall	f1-score	support
0	0.46	0.30	0.36	133
1	0.33	0.51	0.40	275
2	0.47	0.75	0.58	406
3	0.84	0.71	0.77	1896
4	0.00	0.00	0.00	39
accuracy			0.66	2749
macro avg	0.42	0.45	0.42	2749
weighted avg	0.70	0.66	0.67	2749

Figure 13: Second CNN with hyperparameter changes - Performance metrics

As we can see from the image above, the overall accuracy of the model on the test set is 66.31% and is higher than the previously presented version of the CNN. Overall, while the model demonstrates decent accuracy, there is room for improvement, particularly in handling imbalanced classes and enhancing predictions for minority classes.

Also, the first version, even if it has a lower accuracy score, in the gym environment performed better.

4.5 Confusion matrix

The following confusion matrix was plotted using the following libraries: *Matplotlib* and *Seaborn*. From those images, we can see the difference in terms of precision between each model

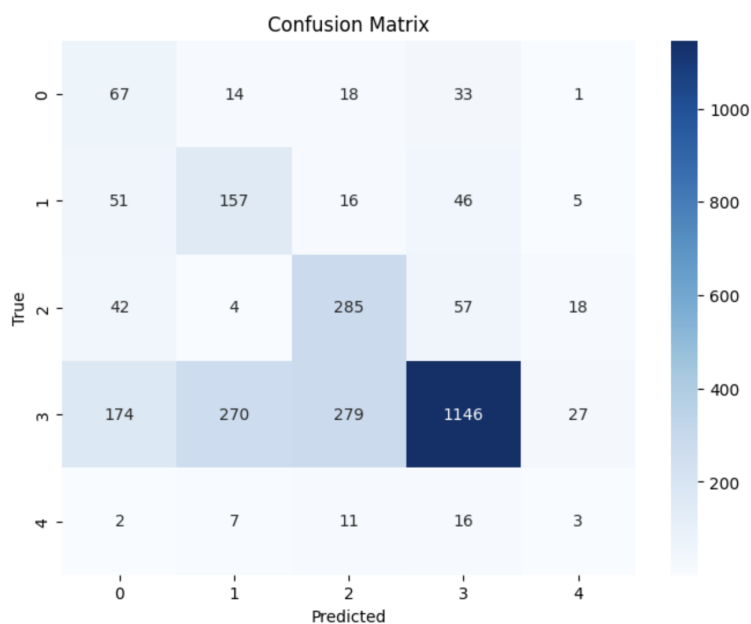


Figure 14: Error matrix for the first CNN

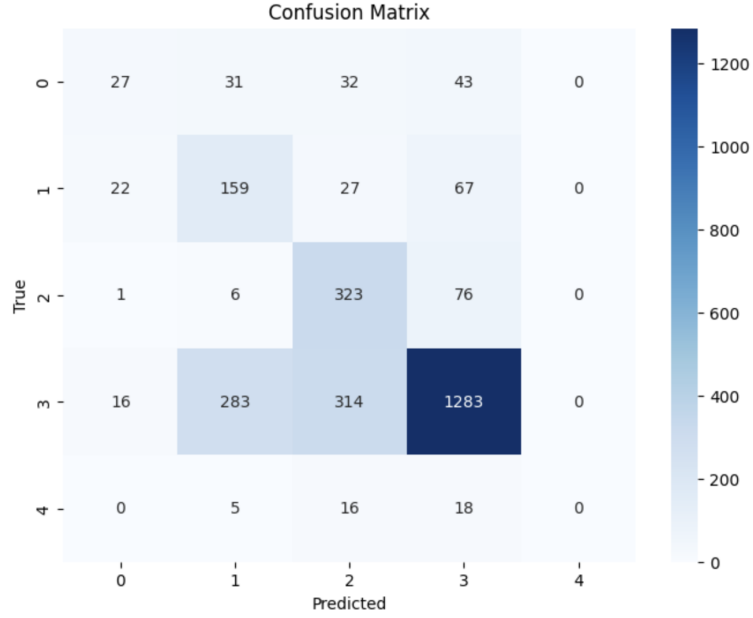


Figure 15: Error matrix for the second CNN

4.6 Computational training time

During the training phase of various models, I observed that the first CNN takes between 15 and 25 seconds per epoch, so the total estimated training time is about 271 seconds. Instead, the training time of the second model was about 225 seconds, 15 seconds for each epoch.

5 Conclusion

In conclusion, even though the first model does have a lower accuracy on the test set, in the gym environment it outperformed the second CNN model. The first model saved with the following name *first_cnn_model.h5* scored more than 830 points in the CarRacing-v2 game. Instead, the *second_cnn_model.h5* did not perform that well, in fact, the final score was near 330 points and the car's behavior was not stable.