Machine Learning, Homework 2 Report

Simone Palumbo 1938214, Sapienza University of Rome

2023-2024

Contents

Chapter 1

Overview

In the following report, I describe the task of devising solutions for an image classification problem to learn the behaviour of a racing car in a Gym envinronment.

In Chapter 2 I describe the preprocessing technique used on the dataset

In Chapter 3 I delineate the methods I employed to solve the problem

In Chapter 4 I present the results

Chapter 2

Preprocessing

2.1 Description of the Dataset and the Environment

The dataset was already divided into training and test set, and contains 96x96 RGB top-down images of a car and a race track, each one labelled with one out of the 5 actions available for the control of the car: 0 (do nothing), 1 (steer left), 2 (steer right), 3 (gas), 4 (brake). The car starts at rest in the center of the road and finishes when all the tiles are visited. The reward function is specified in the Gymnasium documentation.

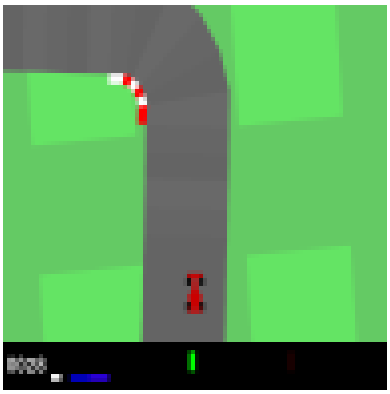


Figure 2.1: A sample from the traning set

2.2 Data Loading

The training set has the following structure:

```
train
    train/0
        train/0/0000.png
        ...
        train/0/0999.png
    train/1
        train/1/0000.png
        ...
    ...
    train/4
        ...
```

The test set has the same structure. This is the proper structure for ImageFolder to get the datasets directly from the path. First, I put it in Google Drive to store it. Then, in the data loading section, I mount the drive and create the datasets and the dataloaders thanks to ImageFolder and DataLoader. It is important to note that we have to transform our data when using ImageFolder. In fact, we are still dealing with 96x96 RGB png images, that we can't directly use as input for our model. For that, we use the transforms:

```
training_transform = transforms.Compose([
    transforms.Resize((96, 96)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15, interpolation=Image.BILINEAR),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5, 0.5))
])
```

Listing 2.1: transforms for the train set

Besides resizing the images if it's not 96x96 I do *data augmentation*. I will talk about it in Section 2.4. Finally, I apply standardization, that is treated in Section 2.5.

```
test_transform = transforms.Compose([
    transforms.Resize((96, 96)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5, 0.5))
])
```

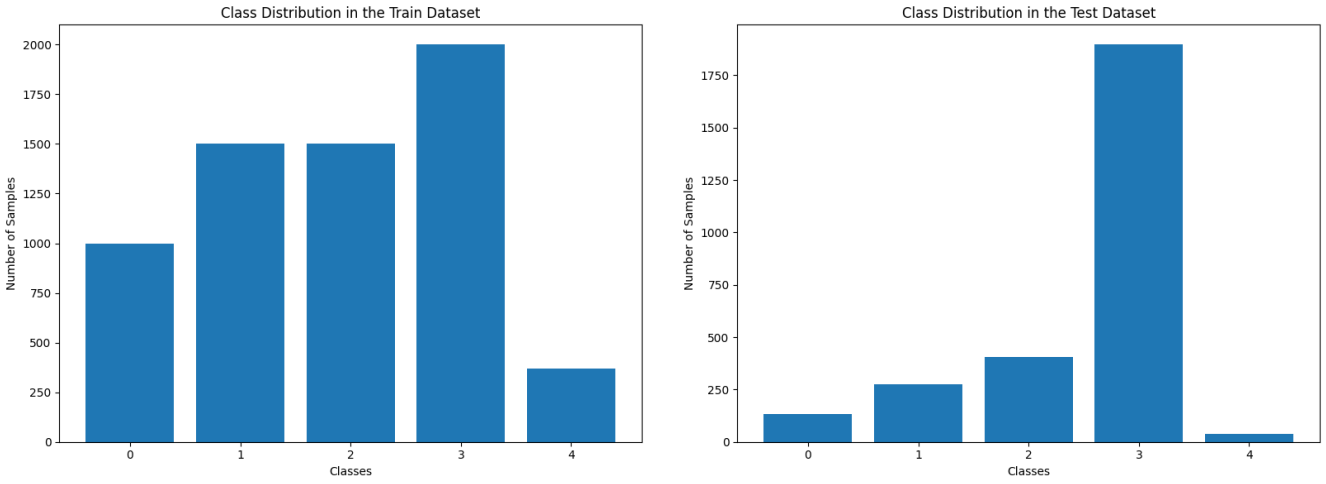
Listing 2.2: transforms for the test set

After the transformations and the loading, we have that each image is now a 3x96x96 Pytorch tensor, and we have 6369 tensors in the train set and 2749 tensors in the test set.

Batch Size for the DataLoader I chose a batch size of 64 for the data loaders. During model training, we pass samples in "batches" and reshuffle the data at every epoch (shuffle = True) to reduce model overfitting. Changing the batch_size value could be interesting to observe how performances would have changed.

2.3 Label Distribution

We can visualize the frequency of each category. In our case, *both the train set and the test set are unbalanced*, as shown in Figure 2.2.



(a) Label distribution for train set

(b) Label distribution for test set

Figure 2.2: These two histograms show us that both the train set and the test set are unbalanced

What is the problem with unbalanced dataset? The fact that the dataset is unbalanced, implies that *accuracy is not a good metrics*, and (weighted) F1-score should be preferred. This is due to the fact that *the model is designed to maximixe accuracy*, and so it will favor the larger class and sometimes even ignore the smaller class if the data is highly unbalanced. So basically, if we will get an high accuracy, doesn't mean that the model is good.

2.4 Data Augmentation:

As introduced in section 2.2 I used data augmentation to reduce the overfitting of the two models. This technique consists on slightly modifying the images with several transformation.

2.4.1 Resizing and Rescaling

I resized the images at the start to be sure that they were all 96x96, but it can be used in more creative ways, as shown in Subsection 2.4.4.

2.4.2 Random Rotation and Interpolation:

I used RandomRotation with 15 degrees. That means that each time this transformation is applied a random rotation in the [-15, 15] degree is chosen and applied to the image. When rotating an image, potential empty spaces or overlapping pixels may arise. To address this, interpolation methods are employed to estimate pixel values at the transformed coordinates. I tried two methods: nearest neighbor interpolation and bilinear interpolation.

Nearest neighbor interpolation: is the simplest method to interpolate an image. This interpolation selects the value of the nearest pixel in the original image to assign to the rotated pixel. The nearest neighbor interpolation will result in a rotated image with blocky and pixelated edges. This is because each pixel in the rotated image takes the value of the nearest pixel in the original image, resulting in a stair-step effect.

Bilinear interpolation: calculates a weighted average of the four nearest pixels in the original image. This interpolation, produces a smoother and more visually appealing rotated image than the previous method. This is achieved by considering the weighted average of neighboring pixels, resulting in smoother transitions between colors. *The following analysis and performances shown in this report are done by using bilinear interpolation.*

2.4.3 Random Horizontal Flip:

This transformation horizontally flips the image randomly with a given probability. I have chosen 50%, which is actually the default value.

2.4.4 Other thing I could do

Another possible transformation would have been to Resize, making the image larger, and then Random Crop the image. This essentially would have zoomed and cropped out a region from the image. It would be necessary to apply the transformation in the test set as well, in order to get consistency. However, we don't want any randomness in the test set, so we can't use RandomCrop on it. A work around would have been to use a non-random type of cropping such as CenterCrop.

2.5 Normalization

Standardization (or feature scaling) is a preprocessing technique. It's important to note that not all algorithms require standardized features. However, it's generally considered good practice to standardize features. By default, when converting the PNGs images in tensors, Pytorch transforms.ToTensor() will normalize values such that they are in the [0,1] range. However, it could be better to have values standardized in the [-1, 1] range for gradient descent[1], and so I used transforms.Normalize() with mean = (0.5, 0.5, 0.5) and std = (0.5, 0.5, 0.5), to standardize afterwards. Note that we have three values for mean and standard variation instead of one because we are dealing with RGB images, and so we have a value for each channel.

[1] used in ADAM or SDG optimizer, section 3.1

Chapter 3

Methods and Algorithms

3.1 Architecture of the Models and Optimizer

I decided to use these two CNN-based approaches to solve the problem:

- CNN1: 3 convolutional layers, each with max pooling, and 2 fully connected layers. It uses ADAM optimizer.
- CNN2: 5 convolutional layers, each with avg pooling, and 2 fully connected layers. It uses SDG optimizer.

ADAM[1] is an optimization algorithm that is actually an extension of SGD[2]. These algorithm are designed to update the weights of the CNN (and neural networks in general) during training.

3.2 CNN1

The architecture of CNN1 is shown in this diagram that I have done in Google Draw:

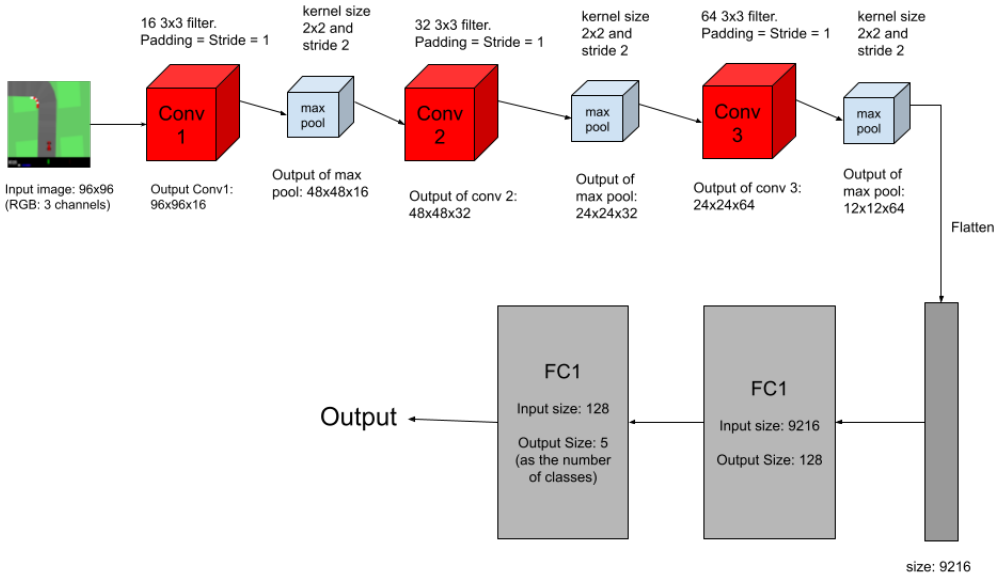


Figure 3.1: Architecture of CNN1

[1] ADAptive Moment estimation
[2] Stochastic Gradient Descent

As shown above, the output of the convolutional layers have to be flattened to be a suitable input for the fully connected part of the network.

Parameters of the model:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 16, 96, 96]             448
         MaxPool2d-2          [-1, 16, 48, 48]               0
            Conv2d-3          [-1, 32, 48, 48]           4,640
         MaxPool2d-4          [-1, 32, 24, 24]               0
            Conv2d-5          [-1, 64, 24, 24]          18,496
         MaxPool2d-6          [-1, 64, 12, 12]               0
            Linear-7                [-1, 128]       1,179,776
            Linear-8                  [-1, 5]             645
================================================================
Total Trainable params: 1,204,005
----------------------------------------------------------------
```

3.3 CNN2

The architecture of CNN2 is shown in this other diagram, also done in Google Draw:

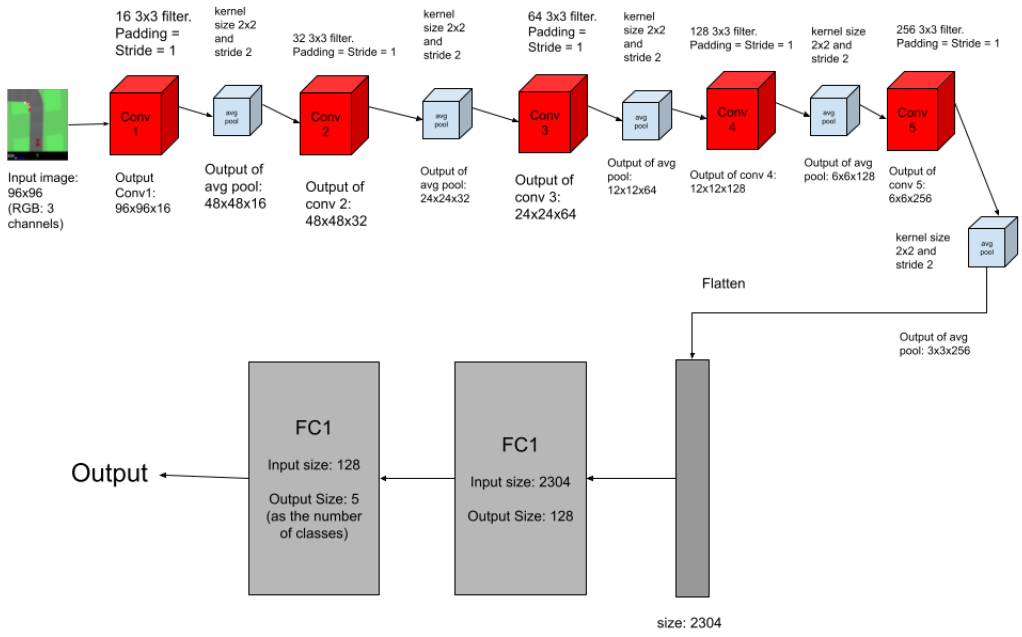


Figure 3.2: Architecture of CNN2

3.3.1 Average Pooling vs Max Pooling

In CNN2 I opted for average pooling instead of max pooling. Max pooling, extracting the maximum value from each window, tends to preserve the most important features. Conversely, average pooling calculates the average value within each window, providing a smoothed representation of the features. *Average pooling tends to be less sensitive to outliers than max pooling and can provide a more generalized representation of the input, helping in mitigating overfitting.*

Parameters of the model:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 16, 96, 96]             448
         AvgPool2d-2          [-1, 16, 48, 48]               0
            Conv2d-3          [-1, 32, 48, 48]           4,640
         AvgPool2d-4          [-1, 32, 24, 24]               0
            Conv2d-5          [-1, 64, 24, 24]          18,496
         AvgPool2d-6          [-1, 64, 12, 12]               0
            Conv2d-7         [-1, 128, 12, 12]          73,856
         AvgPool2d-8          [-1, 128, 6, 6]               0
            Conv2d-9          [-1, 256, 6, 6]         295,168
        AvgPool2d-10          [-1, 256, 3, 3]               0
           Linear-11                [-1, 128]         295,040
           Linear-12                  [-1, 5]             645
================================================================
Total Trainable params: 688,293
----------------------------------------------------------------
```

As shown above, due to the architecture of the model, *the total number of trainable parameters is almost half than in CNN1.* This is because, in a CNN, the majority of the parameters lies in the fully connected layers, and in CNN2 these layers have less neurons since they receive a smaller input, due to the presence of more convolutional layers preceding them, which process the images and reduce their size.

Chapter 4

Evaluation

4.1 Metrics Used

I used:

- Accuracy, that measures the proportion of correctly classified instances among the total number of instances.

$$Accuracy = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

  While accuracy provides an overall view of the model's performance, it might not be suitable for imbalanced datasets where the classes are not evenly distributed, that is the case for our datasets.

- Weighted Precision, that measures the ability to avoid false positives:

$$w_k * Precision_k = w_k * \frac{\text{True Positives}_k}{\text{True Positives}_k + \text{False Positivesx}_k}$$

  Note the subscript of Precision, since we are dealing with multiclass classification and we have to compute Precision for each class k = 0, ..., 5.

- Weighted Recall, that measures the ability to avoid false negatives:

$$w_k * Recall_k = w_k * \frac{\text{True Positives}_k}{\text{True Positives}_k + \text{False Negatives}_k}$$

  Recall is crucial when the cost of false negatives is high. It indicates how effectively the model can identify positive instances.

- Weighted F1-Score, that is the harmonic mean of precision and recall:

$$w_k * F1 - Score = w_k * 2 \times \frac{\text{Precision}_k \times \text{Recall}_k}{\text{Precision}_k + \text{Recall}_k}$$

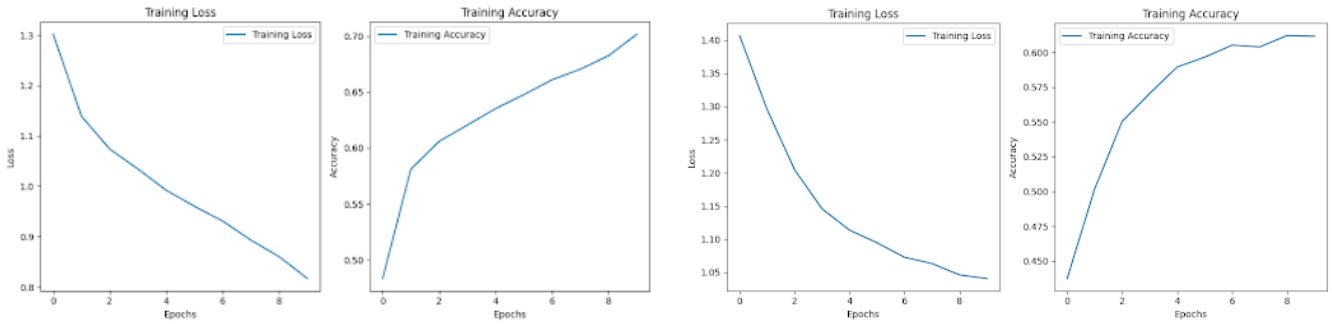
  It is particularly useful when you want to seek a balance between identifying positive instances correctly and minimizing false positives and false negatives.

For the weighted metrics we have $w_k = \frac{\text{No. of samples in class k}}{\text{Total number of samples}}$. To get a single value for these metrics, we do an average of the values for each class.

4.2 Results

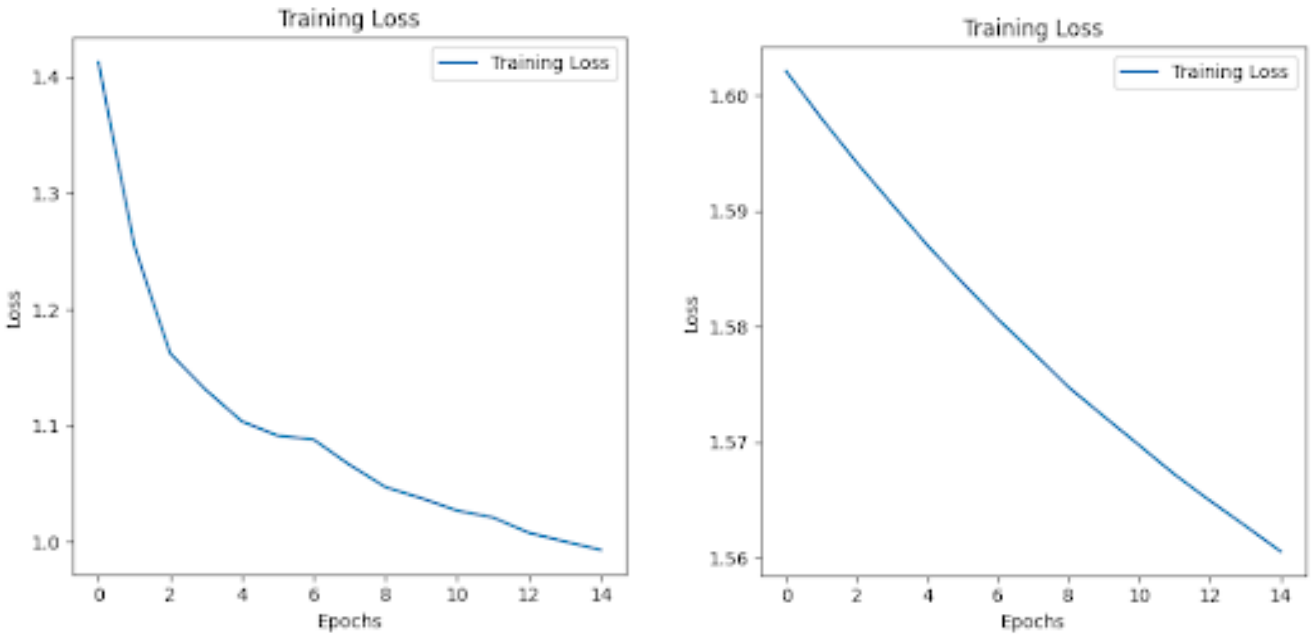
4.2.1 Importance of Data Augmentation

It significantly reduces overfitting. For instance, CNN1, with a learning rate of 0.001 and 10 epochs, increases its accuracy from 58.24% to 65.19% when Data Augmentation is applied, while its training performance tends to be higher when Data Augmentation is not utilized, as shown in Figure 4.1. Note that this is just an example: the actual comparison, using more reliable metrics for unbalanced dataset than accuracy, will be done in Section 4.2.3.



(a) CNN1 Training Performances without Data Augmentation (b) CNN1 Training Performances with Data Augmentation

Figure 4.1: As shown, the performances are better without Data Augmentation, but this doesn't mean that the model is better, since it's overfitting the data.



(a) CNN1 Training Performances (b) CNN2 Training Performances

Figure 4.2: Comparison between training performances of CNN1 and CNN2

4.2.2 Training Performances

Using num_epochs = 15 and learning rate = 0.01, we got the training performances shown in Figure 4.2. CNN2 training performance compared to CNN1 ones presents a smoother and higher loss. However, that doesn't mean that CNN2 is definitively worse that CNN1, since the comparison should be done on the test set. Moreover, this comparison was done with hyperparameters that are the best for CNN1, as shown in Subsection 4.2.3, but not for CNN2.

4.2.3 Hyperparameter Tuning:

I've chosen to tune the learning rate hyperparameter, in order explore the impact of varying the step size in optimization on convergence and overall performances. Additionally, I'm considering tuning for the number of epochs, even though it's feasible to design an algorithm that automatically halts the model at its optimal epoch to prevent overfitting. I tried value for num_epoch from [10, 15, 30] and learning rate in [0.1, 0.01, 0.001]. Fixing every other parameter and using data augmentation, I obtained the results shown in Table 4.1 and 4.2.

Tuning for CNN1: The best choice is learning rate = 0.001 and epoch = 15, since it maximizes the weighted F1-Score. Note that there are choices that have higher accuracy, but it's not of interest for us. As the confusion

Table 4.1: Performances on the test set for CNN1, for different values for Epoch and Learning Rate

Epoch	Learning Rate	Accuracy	W-Avg-Precision	W-Avg-Recall	W-Avg-F1-Score
10	0.001	65,19%	0.72	0.65	0.67
10	0.01	68.97%	0.48	0.69	0.56
10	0.1	68.97%	0.48	0.69	0.56
15	0.001	67,30%	0.72	0.67	0.69
15	0.01	64.68%	0.70	0.65	0.66
15	0.1	68.97%	0.48	0.69	0.56
30	0.001	57,80%	0.71	0.58	0.61
30	0.01	68.97%	0.48	0.69	0.56
30	0.1	68.97%	0.48	0.69	0.56

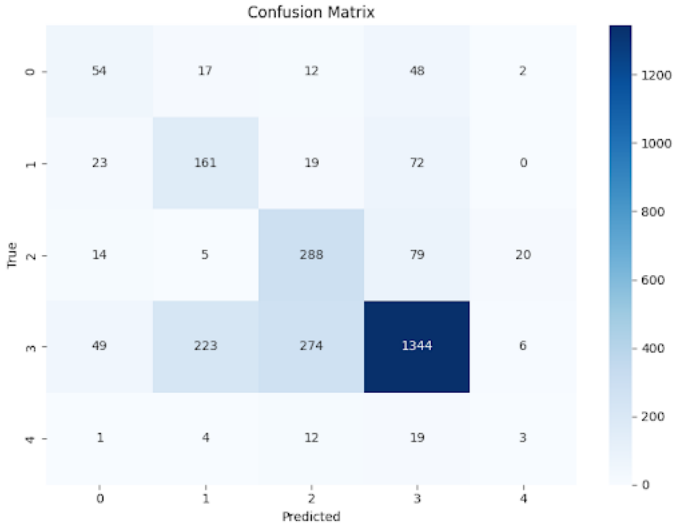


Figure 4.3: Confusion Matrix for CNN1 for hyperparameters (lr = 0.001, epoch = 15)

matrix depicts in Figure 4.3, the model is able to predict correctly most of the sample of class 3, the majority class. 19 samples of classes 4 are predicted as samples of classes 3. This could seems neglectable, but class 4 has only 39 samples. Indeed, just 3 samples of classes 4 are predicted correctly.

Table 4.2: Performances on the test set for CNN2, for different values for Epoch and Learning Rate

Epoch	Learning Rate	Accuracy	W-Avg-Precision	W-Avg-Recall	W-Avg-F1-Score
10	0.001	68.97%	0.48	0.69	0.56
10	0.01	68.97%	0.48	0.69	0.56
10	0.1	68.97%	0.63	0.69	0.66
15	0.001	68,97%	0.48	0.69	0.56
15	0.01	68,97%	0.48	0.69	0.56
15	0.1	69,73%	0.65	0.70	0.67
30	0.001	68.97%	0.48	0.69	0.56
30	0.01	68.97%	0.48	0.69	0.56
30	0.1	63.95%	0.66	0.64	0.64

Tuning for CNN2: The best choice is learning rate = 0.1 and epoch = 15. As for CNN1, the best is chosen based on weighted F1-score value, so it may not be the best for the accuracy.

All predicted class 3 case: Both in CNN1 and CNN2, for several configuration of learning rate and epoch, the performances are Accuracy = 68.97%, Precision = 0.48, Recall = 0.69 and F1-Score = 0.56. In this case, the confusion matrix looks like the one in Figure 4.5. So, even if the accuracy is higher, the performances are clearly not good. In fact, the model is predicting each sample as class 3, getting it right for the majority of the dataset, since class 3 is the bigger one, but obviously getting it wrong for all the other classes. This is recognizable from the low F1-Score.

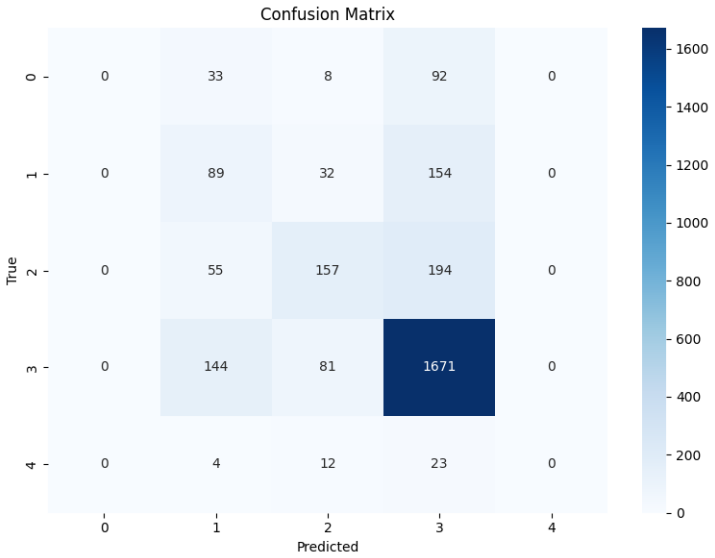


Figure 4.4: Confusion Matrix for CNN1 for hyperparameters (lr = 0.1, epoch = 15)

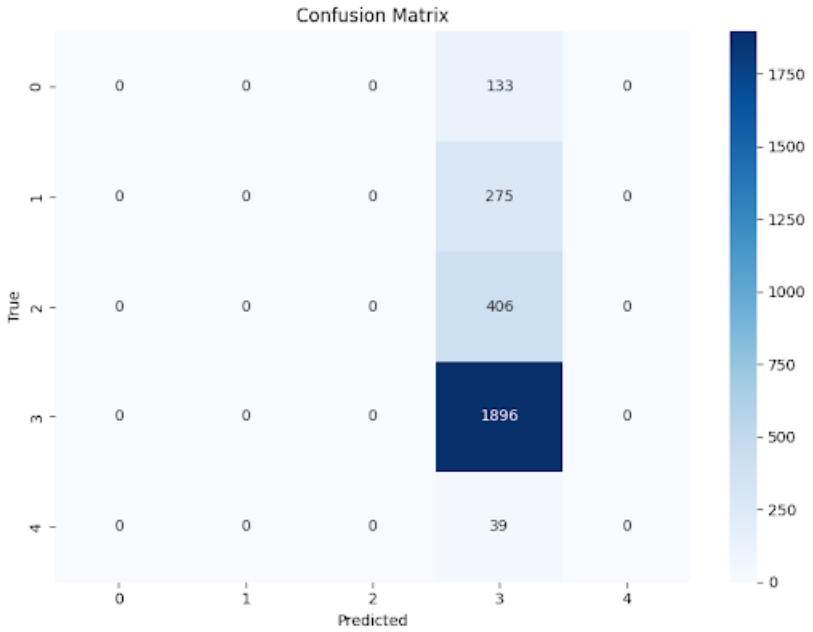


Figure 4.5: Confusion Matrix for the all predicted class 3 case

4.3 Conclusions

Neither of the two models presents outstanding performances, and their results are actually similar. Both models have a tendency of overfitting the data. I thought that CNN2, using average pool, that tends to reduce overfitting, as explained in Subsection 3.3.1, would have had better results. However, I believe differences in the optimizer and the number of layers have altered this prediction.

Criticality with Gymnasium and Box2D: to check how my agent behaves and be able to share the video of the car race, I have dedicated considerable time attempting to install Gymnasium on my machine, working on my Colab, but for some errors I never had the possibility to make it work correctly.