Machine Learning

Homework 2

Yusupha Juwara
juwara.1936515@studenti.uniroma1.it

January 14, 2024

Contents

Chapter 1

Report about Car Racing using Neural Networks

1.1 Car Racing Environment Overview

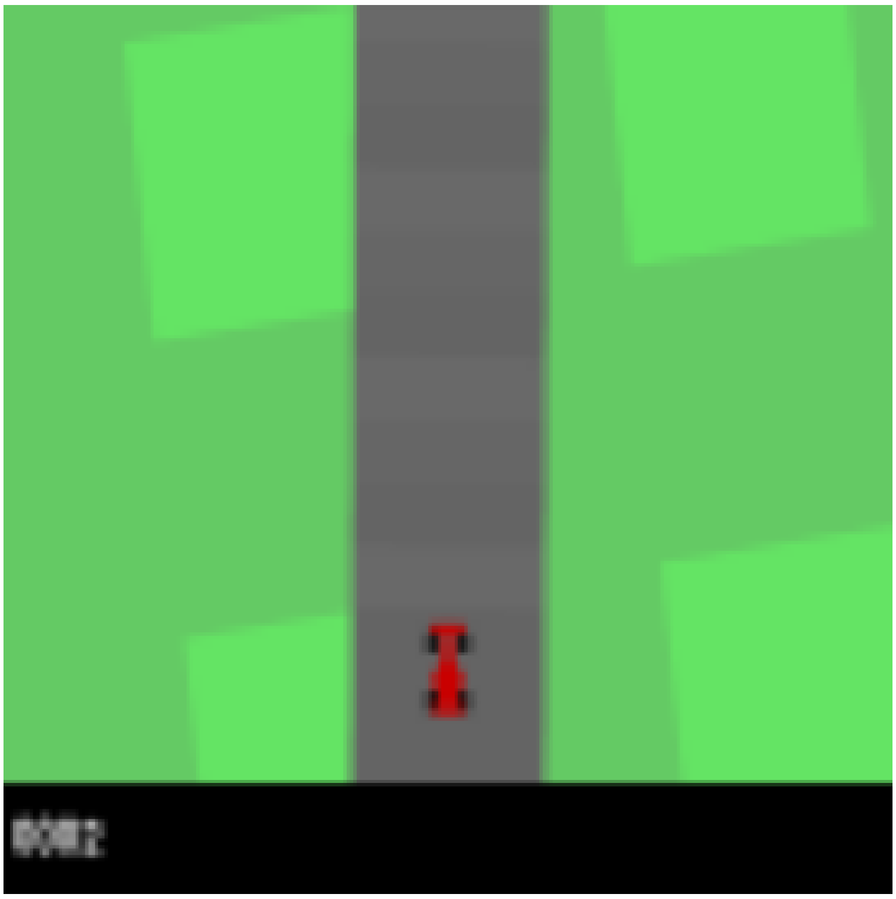

Figure 1.1: Car racing version 2

A brief overview of the CarRacing-V2 environment as taken from their website:

The easiest control task to learn from pixels – a top-down racing environment. The generated track is random every episode.

Some indicators are shown at the bottom of each frame window along with the state RGB buffer. From left to right: true speed, four ABS sensors, steering wheel position, and gyroscope. To play yourself.

This latter paragraph suggests that we can crop the frames to remove the **indicators** and, thus, have a frame size of 84×84.

1.1.1 Action Space

The action space is either continuous or discrete, representing the control inputs for the car. The agent can control the car's acceleration, steering, and braking.

```
Box([-1. 0. 0.], 1.0, (3,), float32)
```

If continuous there are 3 actions:

1. **steering**, -1 is full left, $+1$ is full right
2. **gas**
3. **breaking**

If discrete there are 5 actions:

1. **do nothing**
2. **steer left**
3. **steer right**
4. **gas**
5. **brake**

1.1.2 Observation Space

```
Box(0, 255, (96, 96, 3), uint8)
```

- **Image:** A top-down 96x96 RGB image of the car and race track.

1.1.3 Transition Dynamics

The CarRacing environment is governed by realistic car physics and dynamics. The transition dynamics involve the car's acceleration, steering, and braking inputs, influencing its position and velocity on the track.

1.1.4 Reward

The objective is to drive the car skillfully around the track. The agent receives positive rewards for covering distance and completing laps. Going off-track result in negative rewards. The goal is to maximize the cumulative reward over time.

The reward is -0.1 every frame and $\frac{+1000}{N}$ for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is $1000 - 0.1 \times 732 = 926.8$ points.

1.1.5 Starting State

The car starts at rest in the center of the road.

1.1.6 Episode Termination

The episode finishes when all the tiles are visited. The car can also go outside the playfield – that is, far off the track, in which case it will receive -100 reward and die.

1.2 Model Architectures

In this section, I describe the architecture of the convolutional neural network (CNN) used for the Car Racing. The network is designed to extract hierarchical features from input images, enabling the model to make accurate predictions.

Two types of stem layers are utilized in the base CNN feature extraction process:

BasicStem

The BasicStem module represents the default conv-batchnorm-relu stem. It consists of a sequence of operations, including a 2D convolutional layer, batch normalization, and rectified linear unit (ReLU) activation. This stem is designed to enhance the representation power of the network.

BasicStemNoBatchNorm

The BasicStemNoBatchNorm module represents the conv-relu stem without batch normalization. It simplifies the stem by excluding batch normalization, providing an alternative feature extraction option.

The network architecture is divided into two variants based on the input scale - ConvNoScale and ConvScale.

The ConvNoScale module represents a CNN with no scale applied to the input. It processes input images of size 96x96. The architecture consists of multiple BasicStem blocks, gradually reducing the spatial dimensions.

The ConvScale module represents a CNN with scale applied to the input, cropping it to 84x84. This variant is designed to capture more localized features. Similar to ConvNoScale, it comprises multiple BasicStem blocks.

These architectural components form the base feature extraction layers and the network variations used for the Car Racing model. The choice between ConvNoScale and ConvScale allows flexibility in adapting the model to different input scales, providing a versatile framework for image feature extraction.

1.3 Datasets

1.3.1 Data Augmentation

Here, I applied two different data transformation techniques to see which one performs better. The first one is just a simple reshape and convert the PIL image to tensor. Furthermore, the '.ToTensor()' automatically normalizes the data.

```
transforms.Compose([
    transforms.Resize((IMAGE_HEIGHT, IMAGE_WIDTH)),
    transforms.ToTensor(), # PIL to tensor, then normalizes in the range (0-1) by dividing by 255
    ])
```

In the second transformation technique, I applied a series of **transformations to the input image to create a variation thereof**, which can help to increase the diversity of the data and prevent overfitting. The transformations include:

- **Resizing and cropping** the image to a random size and aspect ratio using the **RandomResizedCrop transformation**. This allows the model to learn to recognize objects at different scales and positions in the image.
- **Rotating** the image by a random angle between -15 and 15 degrees using the **RandomRotation transformation**. This can help the model to learn to recognize objects from different viewpoints. Notice that the range (-15,15) is just a small tilting of the image.
- **Center cropping** the image to the desired size using the **CenterCrop transformation**. This ensures that the image has the same size and aspect ratio as the input to the model.
- **Converting the image to a tensor** and **normalizing** it using the **ToTensor** and **Normalize transformation**s. This converts the image to a format that can be processed by the model and scales the pixel values to a range between 0 and 1.

This defines **two Compose transformation**s: one for the training data and another for the test data. The training data transformation includes all the transformations, while the test data transformation only includes resizing, center cropping, converting to a tensor, and normalization.

```
(transforms.Compose([
        transforms.ToPILImage(),
        transforms.RandomResizedCrop(size=((IMAGE_WIDTH+50, IMAGE_HEIGHT+35)), scale=(0.8, 1.0)),
        transforms.RandomRotation(degrees=5), # rotates within range (-5 to 5).
        #transforms.RandomHorizontalFlip(), # randomly flips horizontally with a default probability of 50%
        transforms.CenterCrop(size=((IMAGE_WIDTH, IMAGE_HEIGHT))),
        transforms.ToTensor(), # PIL to tensor, then normalizes in the range (0-1) by dividing by 255
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
```

Also note that in the first case, both the test and train datasets have the same transform since this was intended to be simple, basic conversion to a tensor with normalization, while in the second case, they are different.

1.3.2 Efficient Dataloading Pipeline

Efficient data loading is a critical aspect of machine learning projects, particularly in avoiding the bottlenecking of the training process. To address this issue, PyTorch provides the **DataLoader** class, which efficiently loads data **batch-by-batch**. The same data loading pipeline also exists in Pytorch Lightning, which I used in this Homework. To further enhance this process, I created a new dataset class called **CarRacingDataModule**, which extends the Dataset class of Lightning and takes in a few parameters, one of which is the transform object defined above for data augmentation in order to have a robust model that generalizes well to new unseen data.

The DataLoader class and CarRacingDataModule work together to load and preprocess data in parallel, significantly reducing memory requirements and maximizing the use of computing resources. Additionally, the DataLoader class can **shuffle data**, **split it into batches**, and **load it in parallel across multiple workers**, further improving the efficiency of the data loading pipeline.

By optimizing the data loading process using the DataLoader and CarRacingDataModule classes, it is ensured that my models train faster (compared to not using it) and more efficiently.

1.3.3 Class Imbalance

In any classification task, it is common to encounter **class imbalance**, where some classes have significantly more samples than others. This can lead to poor performance on the minority classes since the model is biased towards the majority classes.

I approached this part in two ways so as to contrast the models and see their performances with respect to this imbalance problem: first by ignoring any class imbalance, and second by catering for it. To solve this (second) problem, I adjusted the loss function to give more weight to the minority class. I calculated the class weights based on the number of samples in each class.

By using these class weights in the loss function, the model can give more weight to the minority classes and prevent being biased towards the majority classes. This can improve the overall performance of the model, especially in cases of severe class imbalance.

1.4 Hyper-parameters

The success of deep learning models depends largely on selecting appropriate hyperparameters, such as the **batch size**, **learning rate**, and **number of epochs**. These hyperparameters determine the speed of convergence, overfitting, and model performance. In this project, I defined these key hyperparameters carefully and conducted experiments to fine-tune them for optimal performance. Furthermore, Pytorch Lightning helps greatly in choosing some of the hyperparameters, such as automatically finding the learning rate based on a small subset of the datasets.

Other hyper-parameters in this project include, but not only:

- **patience**: the number of epochs to wait before stopping the training if the validation loss has not improved. To do this, I used the EarlyStopping callback of Lightning.

,

1.5 Optimizers

I used the **Adam optimizer**, which is a widely used optimization algorithm for stochastic gradient descent in deep learning. The optimizer updates the weights of the neural network during training, aiming to minimize the loss function. However, selecting the right learning rate is crucial for the optimizer to achieve convergence to the optimal solution. Therefore, I used a **learning rate scheduler** to dynamically adjust the learning rate during training. This strategy is known as **learning rate annealing**, which helps to prevent overfitting and improve model generalization. The learning rate scheduler used reduces the learning rate by a certain factor when the validation loss did not improve for a certain number of epochs. By using this approach, the optimizer can converge more efficiently to the optimal solution, which results in improved model performance and better generalization.

1.6 Loss

The choice of a suitable loss function is crucial since it determines the model's ability to learn and make accurate predictions. The loss function measures the difference between the predicted output and the actual target output, and the goal of the training process is to minimize this difference.

In this project, I used the **Cross Entropy Loss**, which is a commonly used loss function in deep learning for classification tasks. It is particularly useful when the output of the model is a probability distribution over multiple classes. The loss function computes the negative log-likelihood of the predicted class probabilities given the true class labels.

It is worth mentioning two important parameters to the loss function:

- The **weight** (1.3.3) parameter in the **CrossEntropyLoss** function is used to assign different weights to each class. This is useful when dealing with Class Imbalance datasets where some classes have much fewer samples than others. By assigning higher weights to the minority class, the model is encouraged to pay more attention to these samples and learn to classify them accurately.

- The **reduction** parameter determines how the loss is aggregated over the batches. The default value, "mean", computes the mean loss over all the samples in the batch [1].

In summary, choosing an appropriate loss function is crucial for training deep learning models. The 'Cross Entropy Loss' function provided by pytorch allows for assigning weights to classes and choosing different reduction options to aggregate the loss over batches.

1.7 Overfitting and Underfitting

Overfitting and underfitting are common issues when training deep learning models.

Overfitting:

- Overfitting can occur when the model learns to fit the training data too closely, resulting in poor generalization to new, unseen data. To address this issue, regularization techniques such as dropout, weight decay, and early stopping can be used.
- Another effective approach is to adjust the learning rate of the optimizer during training, which can prevent the model from learning too quickly and overfitting to the training data. By reducing the learning rate, the optimizer can take smaller steps towards the minimum of the loss function and prevent the model from getting stuck in local optima or oscillating around the optimal solution. This approach is known as **learning rate annealing**, which helps to prevent overfitting and improve the model's generalization performance on new, unseen data.
- Furthermore, **Batch normalization** can also act as a form of regularization by stabilizing and regularizing the training process. It reduces the internal covariate shift, which is the phenomenon of the distribution of the input to a layer changing during training due to the changing weights of the previous layers. By normalizing the activations, batch normalization can help to reduce the effect of this phenomenon and make the training process more stable. Moreover, batch normalization can reduce the dependence of the model on specific weight initializations, which in turn can improve the model's generalization performance. Thus, using a combination of regularization techniques and batch normalization can help to prevent overfitting and improve the generalization performance of the models.

Underfitting:

- Underfitting occurs when the model is too simple to capture the complexity of the data, resulting in poor performance on both the training and test data. Techniques to address underfitting include increasing the model's capacity as explained in 1.4, and improving the quality (and quantity) of the training data - as explained in 1.3.1.

Addressing both overfitting and underfitting requires a combination of regularization techniques, hyperparameter tuning, and data augmentation. By effectively using these techniques, the generalization performance of the used models were improved further, resulting in better performance on new, unseen data.

1.8 Training and Validation

The process of training and validating deep learning models is crucial for their success. The training loop is a commonly used procedure that involves various inputs such as the model architecture, optimizer, learning rate, number of epochs, and several hyperparameters. The procedure initializes several variables to monitor the training progress and iterates over a specified number of epochs.

During each epoch, the model is set to training mode, and training data is fed to the model in batches. The procedure performs a forward pass through the model, calculates the Loss using a loss function that measures the difference between predicted outputs and the ground truth labels, and updates the model parameters using the Optimizers. The procedure tracks the training loss and accuracy for each epoch.

After the completion of the training loop, the model is switched to evaluation mode, and the validation data is fed to the model in batches. The procedure computes the loss between the predicted outputs and the ground truth labels and calculates the validation loss and accuracy for each epoch.

The procedure saves the model's best weights and optimizer state based on the monitored metric, which is typically the validation accuracy (as in this case). It also includes a specified **patience** parameter, which determines how many epochs the training process will wait for an improvement in the monitored metric before terminating the training early.

This procedure is a highly flexible and adaptable mechanism that allows to explore different hyperparameters, optimizer settings, and model architectures to improve the model's performance. Note that in classification tasks like the one at hand,

[1]Other options for the **reduction** parameter in **CrossEntropyLoss** include "sum", which sums the loss over all the samples in the batch, and "none", which returns the individual loss values for each sample in the batch.

it is common to choose the best model based on the best validation accuracy rather than the best validation loss, even though it is the loss that is being minimized.

1.9 Conclusion

In this work, I have played with a lot of different combinations of models, hyperparameters, such as weight to the loss function, different data transformations, etc. Since I have a lot of different combinations that, if I were to include then in the report, perhaps, I the report would be 50 pages long. Since we are told to not exceed 10 pages max, I have decided to not include them in the report here. But, they can be visualized using tensorboard in colab. I have got a lot of different results, best visualized and understand with the notebook itself. So, please, look at the visualizations and the results in the notebook.

Chapter 2

Conclusion

Firstly, we have seen that both datasets 1 and 2 were simple enough for the models and various hyperparameters to fit them well, achieving astounding results of 99% and 97% respectively!

Secondly, about performance, we have seen that the **RBF** outperforms its **Sigmoid** and **Poly** conterparts using the **SVM** with the default sk-learn values. The default values also outperform the other ones I have tried out in this homework. The SVM with the RBF kernel also outperforms the Gaussian Naive Bayes on both datasets, but does so with just a slight difference in performance results of less that 0.5%.

Thirdly, about compute time, we have seen how computational hungry is the RBF kernels. They use far more compute time than any of their counterparts! With the **gamma='scale' and 'auto'**, the RBF kernel was comparable with the others, and the time it took was almost negligible. However, if different gamma is used, especially when gamma is a float value, the RBF takes a lot of compute time for each of the parameter settings I have tried out in this homework. Some of the compute time were close to one hour (1hr) in the extreme! For the Gaussian Naive Bayes, it took far less time for the computations. The exact compute times and the performances are illustrated throughout this report for reference.

Was this result surprising that the RBF outperforms the others? We know that the RBF is the equivalent of the polynomial kernel with infinite dimension. So, it is not surprising that it outperforms!