Juan Carlos González Martínez
82122300079

Data Science Final Assignment

Predictive Model of a video game genre using Screenshots

Index

Introduction 2
Methodology 3
The Dataset 4
Prototype Model 6
Model Selection 11
Final model 13

Introduction

With the expansion of the videogame industry there has been an increase in game-related online discussions, given the tools we count on today it is normal that when discussing such matters screenshots of the discussed videogame are posted in said forums.

This has led to a necessity of organization, for both the website the discussions take place in (This could be, for example, Steam) and more specifically for the users. With this I thought about the development of a tool that could help both parties, helping the website to organize its data and the users to better understand these screenshots: A model that could predict the genre of a videogame through its screenshots.

In this report I will expand in depth and develop this model, explaining the data and methodology used.

Methodology

Starting with a dataset of different images of screenshots belonging to videogames of various genres, the first step is to organize these images by labeling them with the corresponding genre of the videogame they belong to. The next step is to generate a csv file that will contain all images references with their corresponding category,

The next course of action is to build a prototype model, this will serve as a baseline machine learning model that will allow earlier tests and performance reviews using a smaller dataset consisting of 10% the original, that will help in the development of a more advanced model.

After discussing the different types of model we could develop with their advantages and disadvantages, we will begin the building of our selected model, which will be subject to different design changes comparing different architectures and selecting the one with the best performance.

We will train this model and evaluate its ability to classify the different screenshots our dataset consists of, using different visual representations to show and evaluate its performance, comparing the improvement with the previously built prototype.

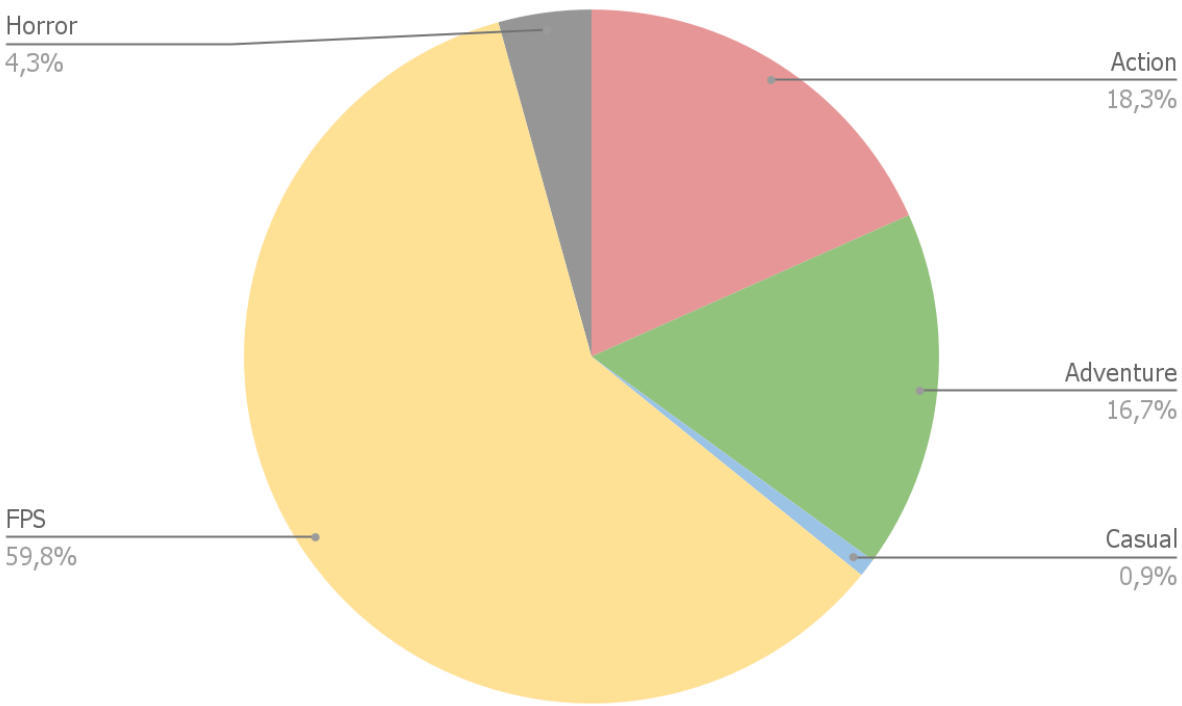
The Dataset

In the development of this model, I used a dataset consisting of a total of 57.447 images and screenshots of different video games cataloged by their different genres as shown below.

The **genres** chosen in this catalogation were those with the most related number of images, as such, I did not select genres such as Hack and Slash, where the number of images in this dataset would have been minimal. Another decision was to not take into account those genres that could be considered generical, as Indie could be, due to the ambiguity of this kind of genre.

While cataloging this data I came across games that could be considered belonging to **different genres**, for example, Half-Life 2, which can be considered both Action and FPS. Since one of the primary platforms this model is aimed to be used at is Steam, I took the decision to deal with this multi-genre games by labeling them with the predominant genre as considered by Steam, this way Half-Life 2 would be considered FPS.

Genre	**Number of Screenshots**
Action	10.527
Adventure	9.575
Casual	490
FPS	34.371
Horror	2.484
	57.447



While the genre "**Casual"** was at first contemplated to be dropped, it was finally added as a label due to the big difference of atmosphere this videogames presented with the other genres, being considered contrasting and important enough to train the model on its data, little as it may be.

The images data was obtained from the website LEVEL-DESIGN.org with each screenshot image labeled as the predominant genre, according to Steam store, of the videogame it belongs to.

Prototype Model

For the elaboration of a prototype model (*JuanCarlos_dsprototype.ipynb*) I created a new dataset in which each category is reduced to a 10% of its original size, this is due the exaggerated amount of time using the original dataset for the training could take.

As the library *scikit-learn* models are designed to work with numerical data, one of the first steps in the creation of this model was to convert the column *Genre* into numerical values.

Since the data set contains the **path** to the images instead of the actual image, the next step is to convert this path into an image the model could read and learn from and contain them in an array, using the library *cv2* which allows to read (load) and write (save) image files as NumPy arrays (ndarray). One error that emerged in this process was due to elements in the images in the list having varying shapes, which led to *NumPy* not being able to create a homogeneous array. This was solved by resizing each image to a predefined **size**, in this case 64x64, before adding them to the array.

After creating a 70% and 30% **split** of the data (For the train and the testing of the model respectively), another error appeared due to *scikit-learn*'s *RandomForestClassifier* expecting the input features to be in the form of a **2D array**. This could be easily solved by using *Numpy*'s function *Reshape* to convert the previously mentioned ndarrays containing the images data to a 2D array.

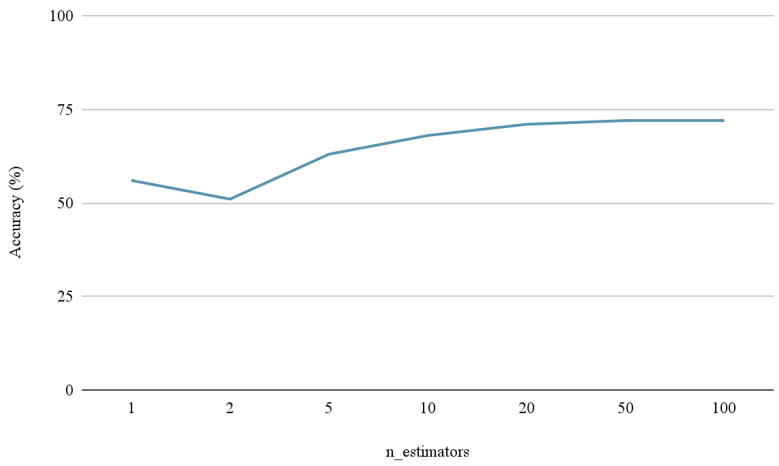
For the creation of this model (p_model) I used the next code:

```
p_model = RandomForestClassifier(
                        criterion = 'gini',
                        max_features = None,
                        n_estimators = 20,
                        verbose = 50,
                        random_state = SEED)
```

- I decided to use *gini* for the **criteria** due to:
    - Being computationally less expensive to calculate compared to others criterias, which was necessary since the model would be treating a massive amount of data.
    - Taking into account one of the categories that takes 59.8% of the data set, *gini* standed out as It tends to be less sensitive to imbalanced class distributions.
- Choosing *None* as the **max feature** was to allow the algorithm to have access to the maximum amount of information when making decisions and less sensitivity to noise in individual features (Which is very useful due to the variety of games visuals in the same genre).
- Setting **verbose** in a value higher than 10 would allow me to take track of the progress in the creation of each tree contained in the model.
- Using a random **seed** allowed reproducibility of the process.
- With **n_estimators** I tried different combinations of values, seeing how the number of trees affected performance and precision, while taking into account the time the model took to be built:

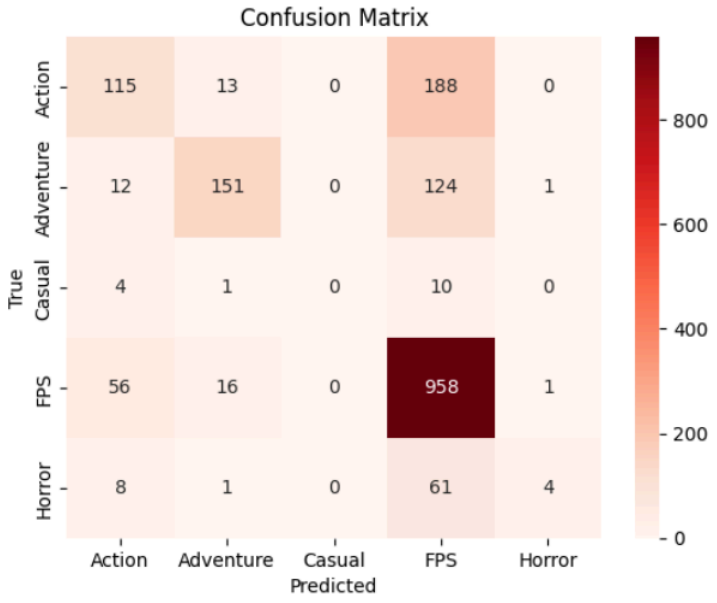
n_estimators	Time taken (min.)	Accuracy (%)
1	3.6	56
2	6.3	51
5	17.5	63
10	31.5	68
20	60.1	71
50	104.3	72
100	239.2	72

We can perceive how the accuracy increases with the number of trees steadily until 20, where the change from 20 to 50 is minimum and barely perceived from 50 to 100.



With this information, I would choose **20** as the number of trees to use in this model, as it provides us with a good performance model in a relatively short time.

We can generate the **confusion matrix** for the chosen model and visualize it using a heatmap:



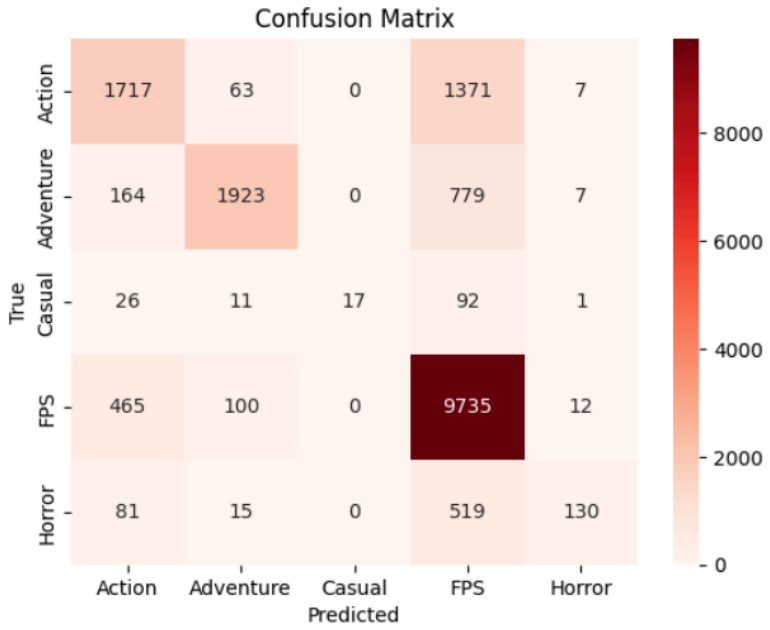
We can appreciate that the model usually predicts correctly the majority of data, however, due to the small size of the data corresponding to the “Casual” genre, it hasn't predicted any data as belonging to this category. In “Horror” we are presented with a similar situation, when having a data size of around 200 images (10% of the original dataset), the model only tried to predict 6, with 4 predicted correctly.

A remarkable point is the fact that, due to the big size of the “**FPS”** genre, the majority of images are correctly predicted to belong to this genre.

Now that we know the best values to assign to the prototype model, it is created again using the **full dataset**, which gives us the following information:

n_estimators	Time taken (min.)	Accuracy (%)
1	26.3	63
20	630.2	78

With the generated confusion matrix for 20 n_estimators using a heatmap visualization:



Where we can see that the values are approximate to those obtained in the 10% dataset, with a better accuracy due to the remarkable increase in the number of images the model was able to train with.

Model Selection

The next step is to discuss and consider the different courses of action and model architectures to build the final model with. The various options we are can consider are:

- **Convolutional Neural Network (CNN):** Most commonly used in Computer Vision as they were originally designed to map image data to an output variable, being very effective in any type of prediction problem involving image data as an input. It typically has three layers:
    - Convolutional layer: Effective for capturing hierarchical and spatial patterns in images.
    - Pooling layer: Help make the learned features more robust to variations in scale, orientation, and position.
    - Fully connected layer: Responsible for combining the high-level features learned by the previous layers to make predictions
- **Recurrent Neural Networks (RNN):** A type of Neural Network where the output from the previous step is fed as input to the current step, as such, they are designed to work with sequence prediction problems. A basic RNN consists of the following components:
    - Input layer: Responsible for receiving the input data at each time step.
    - Recurrent Connection: Allows information to be passed from one time step to the next.
    - Hidden State: Captures information about the sequence seen so far.
    - Output layer: Produces the output for the current time step based on the current input and the hidden state.
- **Dense neural networks:** One of the most straightforward configurations for a deep learning model, the term "dense" comes from the fact that each neuron in a layer is densely connected to every neuron in the subsequent layer.

When dealing with most image processing tasks RNNs are inferior to CNNs; a RNN could be used when dealing with a sequence of images, such as with a video, and even in this situation the more natural approach would be to combine both a CNN (For the image processing) with an RNN (For the sequence processing).

When dealing with image data, Dense Neural Networks lack the ability to efficiently generalize features from image pixels, struggling to efficiently learn spatial patterns and potentially leading to suboptimal performance. Dense Neural Networks, especially when dealing directly with image data, can have a large number of parameters while CNNs are designed to be more parameter-efficient for image data by sharing weights through convolutional operations, this is something we have to take into consideration due to the vast size of the dataset we are dealing with.

In conclusion, Convolutional Neural Networks prove to be more useful and prepared to deal with image classification problems such as the one we are currently treating, with their ability to learn complex patterns and object representations from images and automatically learn features from raw pixel data. CNNs require a large amount of labeled training data to generalize well. as insufficient data can lead to overfitting, something our big dataset can be useful with.

Final model

For the training and testing of the final model (*JuanCarlos_dsfinal.ipynb*) I divided the dataset using a **70 - 30** split, following the line of the prototype model, with 70% of the original data being used to train the model and 30% to test its performance.

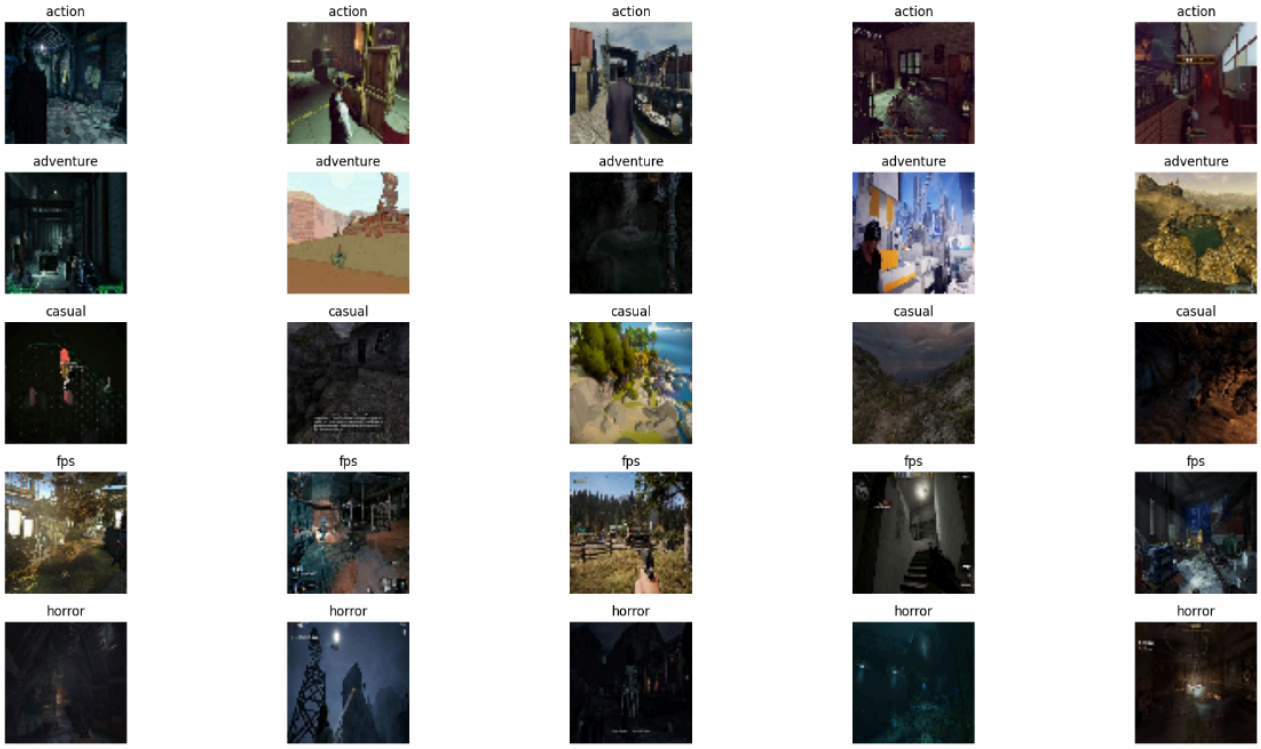
After fixing the random seed to facilitate reproducibility and assigning the test and training datasets to its corresponding variables, we create a dictionary containing the video game genres we will be using, this allows us to map numerical labels to the corresponding genres which will be helpful dealing with vector's indices and consistent mapping during the data encoding process.

A quick look at the training data distribution will ensure us the dataset has been splitted properly:

ID	Genre	Full Dataset count	70%	Training Dataset count
0	Action	10.527	7.368,9	7.368
1	Adventure	9.575	6.702,5	6.702
2	Casual	490	343	343
3	FPS	34.371	24.059,7	24.059
4	Horror	2.484	1.738,8	1.738

For the next part I created the function ***load_images***, which use is implicit in its name, given the list of the image file paths, as this is how we save images in our dataset, each image is loaded and resized to an established target value (I will use a size of 64 x 64 pixels). As we did with the prototype model, the image will be converted to a *NumPy* array, making it suitable for further processing, and saved in a list where we will store all processed images.

By using the *Ploty* graphic library we can load 5 of this processed images and **display** them, adding its corresponding genre to each of them as a label, allowing us to verify that the images are being loaded correctly and resized appropriately, and also confirm that the labels associated with the images match the actual content:



We can see some simple patterns in this sample such as the dark tone in the horror genre or the main character centered in the screen of the action screenshots.

The next function, ***data_preprocessing***, continues to prepare the dataset, converting the categorical labels in the "Genre" column into numerical values to then convert them into one-hot encoded format, this is due to each class being represented by a binary vector with all elements being zero except for the index corresponding to the class in the training model, and saved in a list. Images are loaded using our previous function and saved in another list, and their pixels are then stabilized by dividing by 255.

We now have the processed test and training datasets.

Using the *Sequential API* from the *TensorFlow* library we create a Convolutional Neural Network (**CNN**), the chosen model, with the name "*MerlinVision_CNN*". This model consist on the following layers:

1. 2D Convolutional Layer: Performs convolution on the input images by using 32 filters that move 2 pixels at a time with a 5x5 kernel size each. With a *ReLU* activation function due to its simplicity and effectiveness, using non-linearity while being computationally efficient and not suffering from the vanishing gradient problem (As it can occur with other activation functions).
2. 2D Max Pooling Layer: Execute max pooling with a 2x2 window, reducing spatial dimensions.
3. Dropout Layer: Randomly sets to zero 20% of inputs during training to prevent overfitting and reliance on specific neurons early in the network. ***
4. 2D Convolutional Layer: Another convolutional layer with 64 filters with a 3x3 kernel each. A smaller kernel (3x3) allows us to capture more fine details while the larger filters we previously used (5x5) capture more global patterns.
5. 2D Max Pooling Layer: A second max pooling layer with a 2x2 window.
6. 2D Convolutional Layer: A third convolutional layer with 128 filters and a 3x3 kernel as the last one, the increase in the number of filters allows for more complex feature learning.
7. Flatten Layer: Convert the feature map that it received from the max-pooling layer into a format that the dense layers can understand by flattening the 3D output to a 1D vector.
8. Dropout Layer: A new dropout layer with a 20% dropout rate before a dense layer to encourage it to generalize well to new data and prevent overfitting.
9. Dense Layer: A simple Layer of neurons in which each one of the 256 neurons receives input from all the neurons of the previous layer, serves as the classifier, making predictions based on the features learned by the convolutional and pooling layers.
10. Dropout Layer: A dropout layer with a 30% dropout rate added between two dense layers to introduce regularization in the fully connected part of the network.

11. Dense Layer (Output): The output layer with neurons equal to the number of genres, using the *softmax* activation function to convert raw output scores into probability distributions over the different genres.

The objective of this architecture is to progressively learn the data features, starting from simple patterns in the early layers to more complex representations in the deeper layers by starting with Convolutional layers and ending with a fully connected network. The chosen values for the parameters, such as filter sizes and the number of units, are based on the vast size of the dataset.

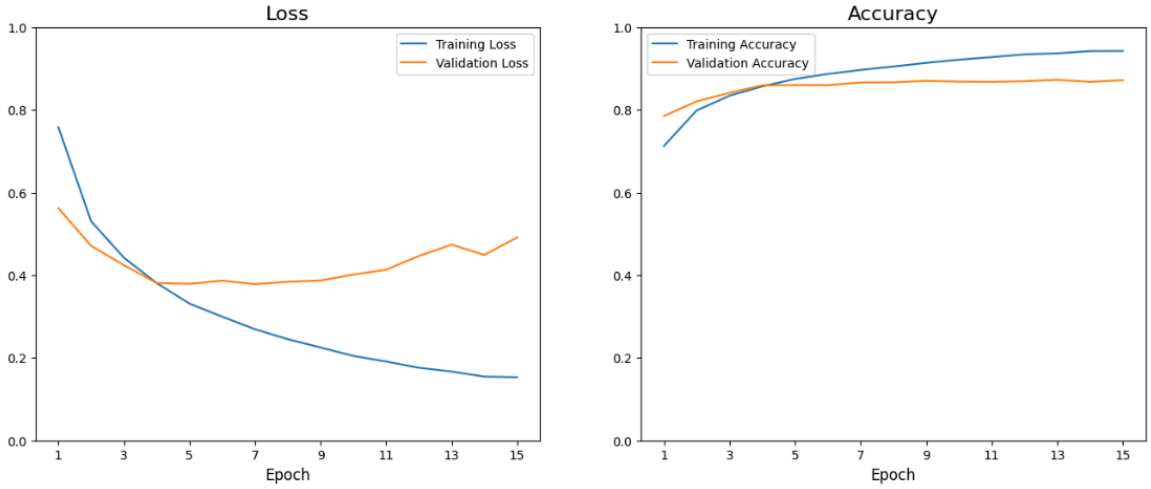
The next part of the code allows us to train the model with a chosen number of 15 epochs (Complete passes through the training dataset) giving information about the training process that we can use to compare different architectures.

For this architecture I tried **different variations**, comparing their accuracy at the epoch number 15:

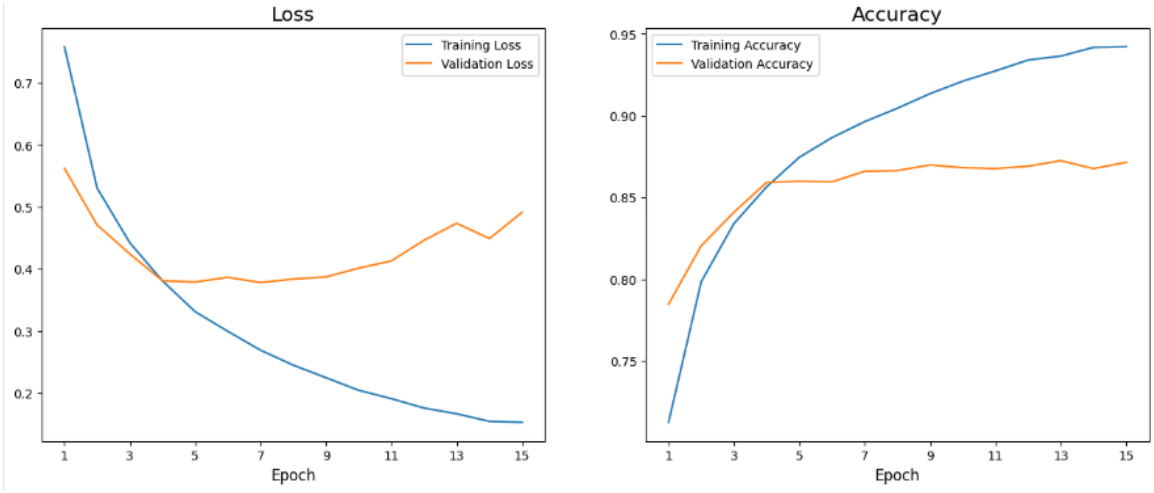
<table>
<tr><th colspan="3">Variation</th><th>Accuracy (%)</th></tr>
<tr><td>Dropout Layers (20%, 20%, 30%)</td><td colspan="2" rowspan="2">First Dropout Layer before the 3rd Convolutional Layer</td><td>83.01</td></tr>
<tr><td>Dropout Layers (30%, 40%, 50%)</td><td>87.17</td></tr>
<tr><td>Dropout Layers (30%, 40%, 50%)</td><td rowspan="3">First Dropout Layer before the 2nd Convolutional Layer</td><td>3 Max Pooling Layers
(A new one before the Flatten Layer)</td><td>86.53</td></tr>
<tr><td rowspan="2">Dropout Layers (20%, 20%, 30%)</td><td rowspan="2">90.62</td></tr>
<tr><td>2 Max Pooling Layers</td></tr>
<tr><td>94.22</td></tr>
</table>

With this we arrive at the chosen architecture, with fewer Max Pooling layers to reduce information loss and without abusing Dropout layers, as it can lead to underfitting.

Following we have a code that uses the *Matplotlib* library to create a figure with two subplots for visualizing **loss and accuracy** over epochs during the previous training, using the "history" variable that contains this training information:



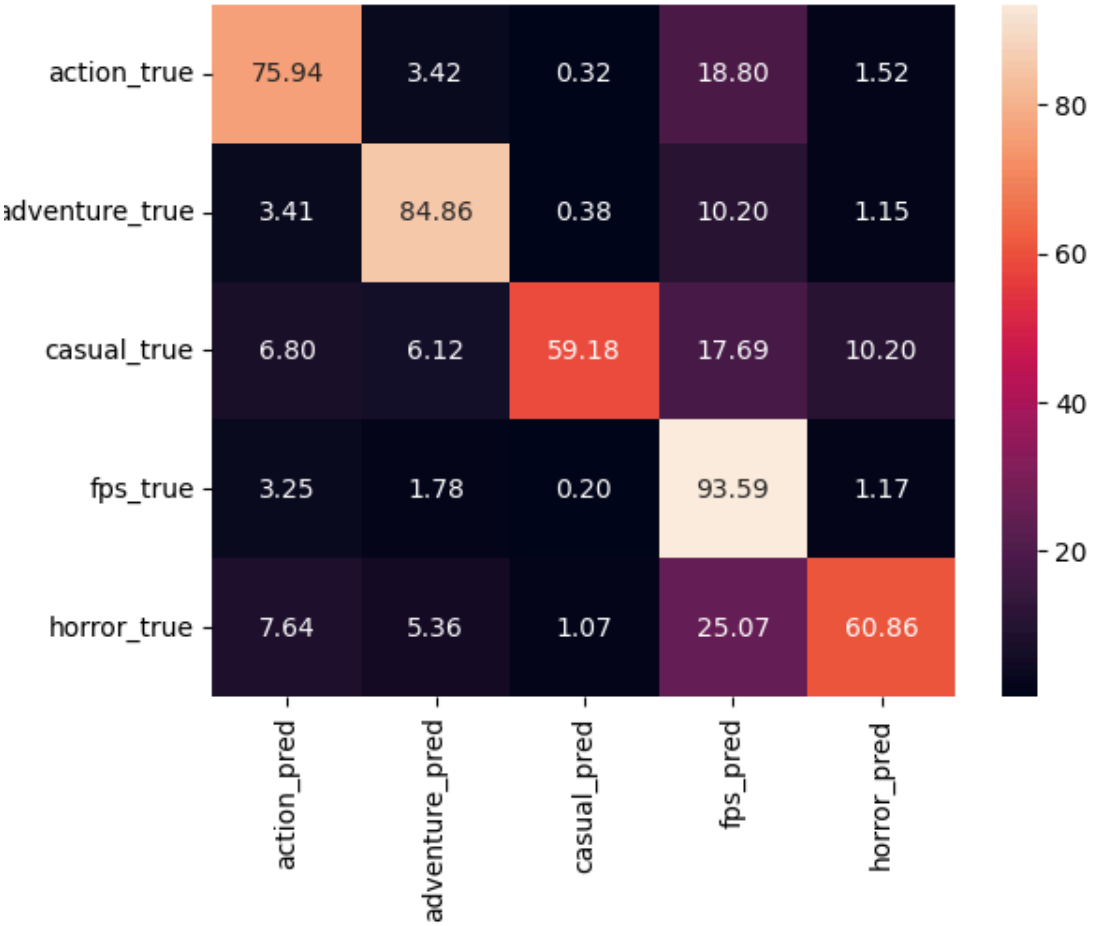
By deleting the Y axis range we can take a better look at the values changes:



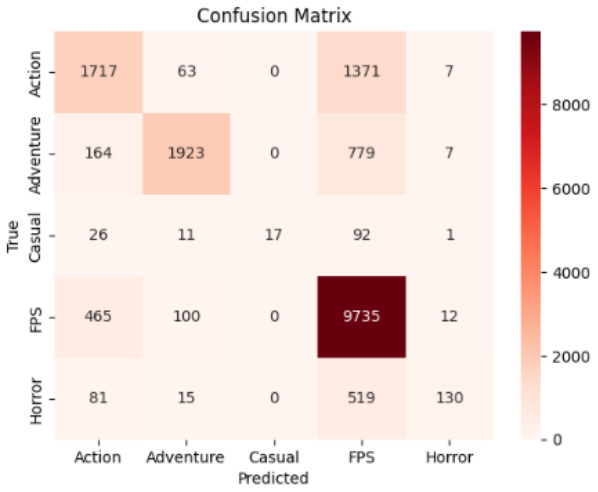
At epoch 14, there is a slight drop in validation accuracy compared to epoch 13, but the validation loss decreases, maybe due to the model focusing on specific patterns in the training data that don't generalize well to the validation set. As the model recovers from this drop in epoch 15 we shouldn't worry, but if we were looking for a more consistent and stable model we would use 13 epochs instead of 15.

The next code will use the now trained model to **predict** the output based on the test dataset we previously declared and save the predicted and true genre labels in two arrays to later evaluate the model's performance. Using the dictionary we declared when starting the program, we can map the indices values to their corresponding genre name for better visualization.

Consequently, we use the *Scikit-learn* library to generate a confusion matrix based on the true and predicted genres, normalizing it to show percentages. The confusion matrix is then converted into a *Pandas* DataFrame so we can create a **heatmap** visualization using the *Seaborn* library, this will help us have a better understanding of the model's performance:



We can easily see the clear difference in color distribution with our early prototype (On the right) where the colors look almost uniformal. Our final models exceeds in the prediction of the genres labels where even the small genre “Casual” has a 60% of correctly predicted labels, while the prototype predicted correctly 17 of 147 (a 11.56%).



Finally we will produce a **classification report** as a summary of the models performance:

Genre	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
Action	83	76	79
Adventure	88	85	86
Casual	64	59	61
FPS	90	94	92
Horror	68	61	64

Our model presents a final accuracy of **87%**