# Pandascore Assignment

Arnaud Yoh Massenet

**Abstract**

In this report, we present the different steps we went through to build a model to find HearthStone cards in a given image. We will discuss the way we created our dataset, the rationale behind our model choice and how we trained our model. We will then discuss some ideas for further work.

To ease the training of the model, we will randomly apply transformations of our images. Such transformations will allow the model to learn on more various images and help it to generalize well. These transformations include cropping, zooming, flipping, changing brightness, contrast, saturation, and hue. Note that the code for theses transformation is not ours.

# Contents



# List of Figures

# 1 Data Gathering

To train our model to detect hearthstone cards, we first have to create a dataset. We did this by creating an automated process with cron and shell. This allowed us to gather screenshots from gaming streams. We then used lableImg to label the screenshots we have taken. In our data set, we only labeled cards on our images to focus the training on HearthStone cards detection.

# 2 Choice of Model

To choose a model, we looked at various object-detection algorithms in the literature. The two major ones are SSD300 and YOLOv3. Both algorithms only require a single model to encapsulate the box prediction and the classification task. A major difference is that YOLOv3 is a much larger network. But more importantly, the first feature map (output) used in YOLOv3 comes from the $79^{th}$ convolutional layer. On the other hand, SSD300's first output comes from the $10^{th}$ layers. This means that with only the first 10 layers of SSD300 we will already have an output. Moreover, the $10^{th}$ layer of SSD300 is designed to detect small objects in images, which exactly what we want, since our cards are only small fractions of the whole image surface.


Figure 1: Image Example

# 3 Implementation of the Model

Based on the previous observations, we decided to build only part of the SSD300 model. This allows us to get an output (feature map) using a smaller and easier to train network. We prefer to use a small network as larger networks would require larger dataset.

SSD300 is based on the VGG16 to which we add layers to get the box and class prediction. In our modified version, we have part of the VGG16 model build:

- 2 convolutional layers with ReLU activation and a max-pooling layer
- 2 convolutional layers with ReLU activation and a max-pooling layer
- 3 convolutional layers with ReLU activation and a max-pooling layer
- 3 convolutional layers with ReLU activation. The final layer will output our feature map.

We then attach 2 convolutional layers, one that will predict the boxes and the other will take care of the classes. Both of these final layers use the feature map as an input.

We initialized the VGG16 layers using the pre-trained weights. The 2 additional layers' weights were initialized using the Xavier uniform distribution and the bias set to 0.

# 4 Training of the Model and Loss function

## 4.1 Training

### 4.1.1 Transfer Learning

With the algorithm finished, we could start training. But to further reduce the need for a large data set, we can use transfer learning. Thus, we loaded pre-trained weights from the actual SSD300 trained on the VOC dataset. Note that we can use the pre-trained weights since we do not need to worry about the weights of the other layers. Indeed, since we did not remove any layers from VGG16 that are between the input and our output, the weights that we did not use would not affect our output.

Thus, by using the pre-trained weights of VGG16 we are taking advantage of transfer learning. Note that the VOC dataset allows classifying multiple objects such as cars, trains, buses, people, dogs, cats, etc... but not cards. As a result, transfer learning will allow us to have a good basis to detect high-level features from the images. However, we still have to train our model to learn to detect more refined features to properly classify cards.

### 4.1.2 Model Training

For each input, we first randomly apply transformations to the images before resizing them to a 300 by 300-pixel dimension. We then extract 3 Tensors (RGB) such that we end up with a (3,300,300) Tensor for each image. We process input by batches of 8 images and thus we have Tensors of (8, 3, 300, 300) for each batch.

For each image, we predict two things, potential boxes based on pre-defined priors and the probabilities for each class to be the one contained in the box. In our case, we have 4 prior boxes for each cell of our output, each with a surface approximately equal to 10% of the image surface. Based on the output and the priors, we will predict

a posterior on the boxes and use it for our predictions. Regarding the class prediction, we simply assign probabilities to each possible class. Note that for both predictions, the input is the output of the partial VGG16 model. The box and class convolutions are of shape (8, 16, 38, 38) and (8, 8, 38, 38). We have 8 images per batches, 4 box predictions (we predict 4 values each) and 4 class predictions ('card' and 'background' classes). The (38,38) comes from the dimension of the output of our partial VGG.

Note that the prediction on the boxes is quite interesting. Instead of predicting 4 boundaries, we instead predict the difference between our boxes center and the center a prior box and the difference in height and width.

Since we have a small dataset, we manually separated the data into train and test sets. This ensured that we had various types of images in both sets. We ended up with 50 train images and 23 test images. Then, we trained our model for 300 epochs and we got the results presented in Figure 2.

## 4.2 Loss Function

Because we are performing predictions on boxes and classes, we need to build our loss around these two components.

### 4.2.1 Box Loss

The box loss is a sum of the absolute differences between the 4 boundaries of the predicted "good" box and the true box. "Good" boxes are those that are close enough to the true boxes and have a high prediction for the correct class. Remember that our box predictions are essentially the center position, length and with difference between the predicted box and the prior. As such we have to go through various transformation to retrieve the actual boundaries of the predicted box.

### 4.2.2 Class Loss

The classification loss is calculated using two box types. The first one is "good" boxes and the other "worst" boxes. Given $n$ "good" enough box, we selected the $3n$ "worst" boxes . The value 3 is empirical. For each selected box, we take its predicted class and select the true box with which the highest Jaccard Overlap. Then, we compute the cross-entropy loss between our predicted class and the true box class.

### 4.2.3 Final Loss

Then, we simply add both losses together to get the final loss. After that, the SGD optimizer will modify the parameters of the model. Note that we can give different weights to both losses. We settled with equal coefficients for both losses with value 1.

# 5 Figures and Results


Figure 2: Cards Detection

To generate these results, we save the model every time we beat the best loss on the test data. This means that we select the parameters that give the best results on the test data.

From Figure 2 and the Appendix we can see that the model does well to capture the position of the cards and predict the correct class. These results are in line with the loss that we get at the end of the training. After 300 epochs, we achieve good results, with validation loss as low as 1.2 average while we start at about 30 loss.

# 6 Limits of our Approach

While the presented results are satisfying, we still have a few issues with the way we approached the problem. First, we are using a big assumption on the relative size of the cards compared to the whole image. Indeed, our model is designed to capture only smaller objects. Thus, we do not know how our model would perform if the images were essentially one single big card.

Another point relates to transfer learning. While transfer learning is great when we have a small amount of data, as it provides a good enough prior on the weights, it would be still better to train our model from scratch with a large dataset of card images. Indeed, transfer learning gives us a bias towards classes that it already learned. Thus, we could potentially be near a local optimum which would prevent us to reach the global optimum.

Because of our small dataset, we drastically reduced the size of our model, forgoing the learning of complex features. This means that, because of our model size, we are most likely missing import features that could have further improved our results.

All in all, the crucial issue with our approach relates to the size of our dataset. With a larger dataset we could forgo transfer learning and train a model from scratch. By doing so, we will not have bias towards other classes and we will avoid class imbalance between our dataset and the data used for pre-training. This means that we will have less issues with local minimum traps.

# 7 Further Work

For this project, it would be interesting to see what would happen if we have access to a larger dataset of HearthStone card images. With a larger dataset, we could fully implement the SSD300 algorithm or even YOLOv3 and train using our dataset only.

We could have also try to augment our dataset using prediction results of our mode on unlabeled data. But that would assume that our model to perform well enough. This might cause a bias as well, so it should be used carefully.

On top of this, we could also refine the classification by learning to distinguish between different types of cards (Spells, Minion, Weapons). We could even go further by learning the card classes (as in the player class: Mage, Warrior, etc...).

Another idea would be to forgo box prediction and instead use image segmentation. This could allow us to better capture cards that have overlapping boxes. This overlap often happens whenever player holds many cards and we can see only a small part of the card on the image. The boxes tend to overlap on other cards especially when they are rotated and we sometimes end up with more boxes than they are cards. Segmentation could help us solve this issue as we would be predicting the exact location of the cards and rotated cards would be less of an issue.

# Appendix

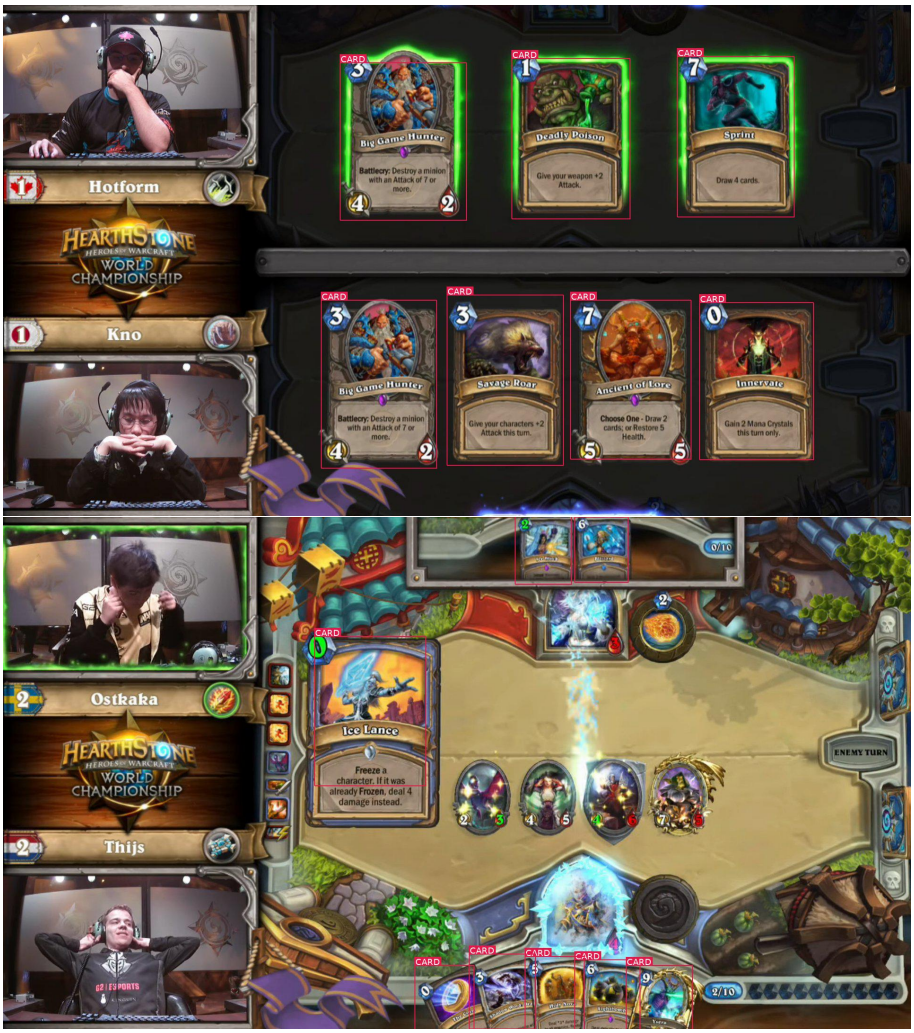


Figure 3: Cards Detection Example 1


Figure 4: Cards Detection Example 2


Figure 5: Cards Detection Example 3

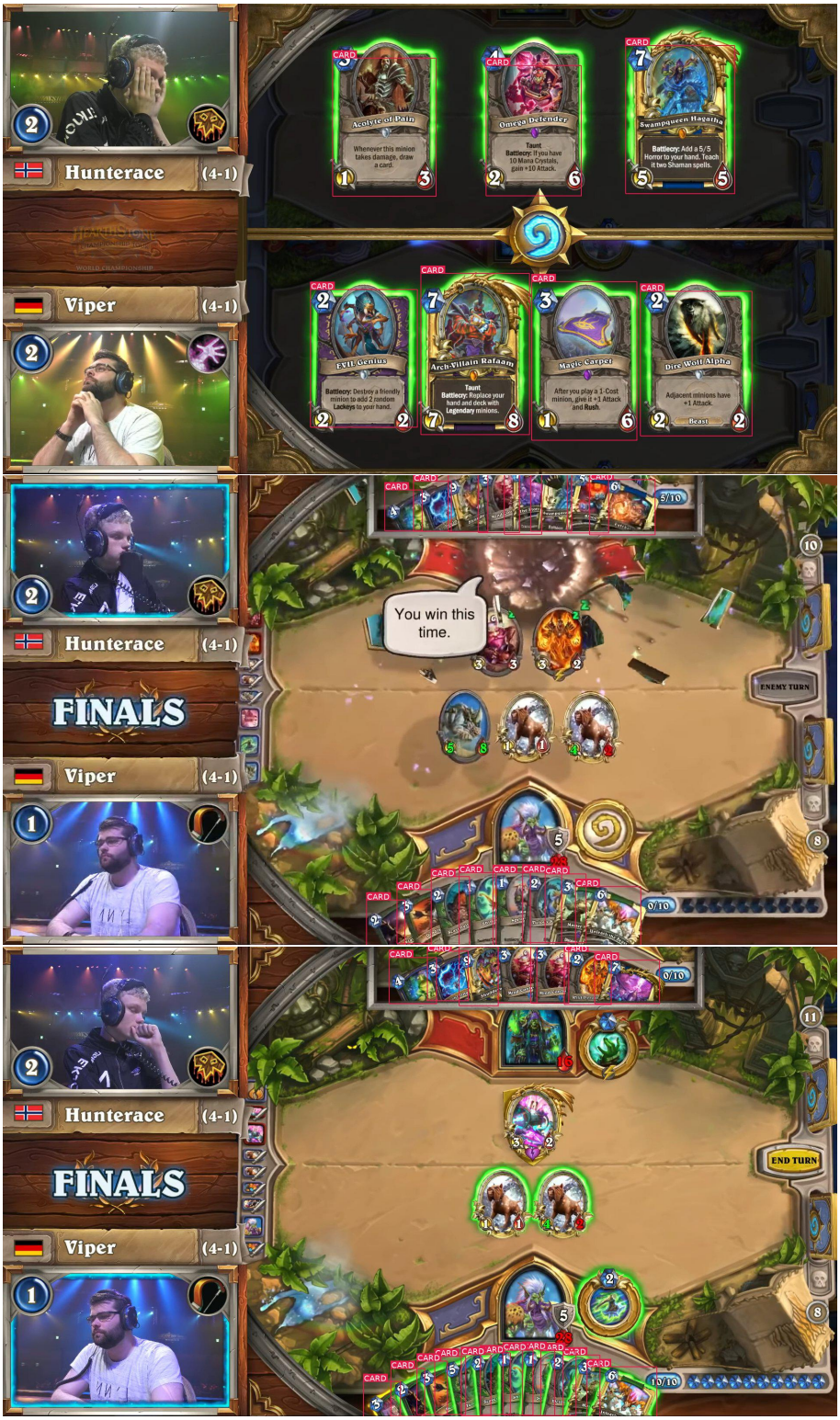


Figure 6: Cards Detection Example 4