Pokémon Cards Classification Using a Computer Vision Model

Diego Surot de la Vega

1 Introduction

This is a project that aims to classify Pokémon Trading Card Game (PTCG) cards by their type (Fire, Metal, Psychic, etc.) using a computer vision model approach. I worked on two main components to achieve this objective:

- **Classification Model**: A PyTorch-based neural network trained on a dataset of labeled card images to accurately predict the type of Pokémon cards
- **Real-Time Detection System**: A Python script using OpenCV that captures live video from a camera, detects the presence of a card, and overlays the predicted type directly onto the video feed in real time

2 Inspiration

This project was greatly inspired by my fascination towards Pokémon and the PTCG. In this regard, I also drew additional inspiration from the following works which provided a general outline for the task:

- Playing Card Classifier using Convolutional Neural Networks (CNNs) (Wijekoon, 2024)
- Identifying Pokémon Cards (Peixoto, 2021)

3 Dataset

3.1 Description

The dataset for this project, *PokemonCards* was retrieved from HuggingFace (TheFusion21, 2024). It contains PTCG cards released in English from 1999 to 2022, starting with the Base Set (1999) and ending with the Silver Tempest set (2022). It originally contained 13,139 cards, exclusively Pokémon monsters cards, with image resolutions ranging from 600×825 to 734×1024. After removing 51 cards due to missing images, the dataset was reduced to 13,087 cards. For each card, the dataset includes the following data fields:

- *id*
- *image_url*
- *caption*
- *name*
- *hp*
- *set_name*

For the purpose of this project, only *id*, *image_url*, and *caption* were considered. In addition to that, a new data field called *type* was created for the training of the model.

id	image_url	caption	name	hp	set
pl1-1	https://images...	A Stage 2 Pokemon Card of type **Lightning**	Ampharos	130	Platinum

Figure 1: Dataset Overview

3.2 Preprocessing Pipeline

The dataset preprocessing steps involved the following tasks:

- Each card was downloaded and assigned their corresponding ID from the dataset
- Extracted the typing information from the *caption* column and created a new column named *type* for each card
- Cards that could not be downloaded were logged to a *.txt* file, and then removed from the processed dataset
- Each Pokémon type was mapped to a unique numerical index and stored in a new column *class_idx*, moving from categorical to numerical labels

Pokémon Type	Class Index
Darkness	0
Colorless	1
Grass	2
Water	3
Metal	4
Psychic	5
Lightning	6
Dragon	7
Fire	8
Fighting	9
Fairy	10

Table 1: Mapping of Pokémon Types to Numerical Indices

4 Classifier Model

4.1 Dataset and Splits

I used a standard split of 70% / 15% / 15% split for training, validation, and testing, respectively. Moreover, splits were stratified to ensure each subset had the same class distribution as the original dataset. This was done specifically to (try to!) mitigate unbalanced class problems due to the fact that Dragon, Metal, and Fairy type cards had much less cards than more popular types like Water and Grass, for example.

Dataset Split	Size (cards)
Training Set	9,160
Validation Set	1,963
Test Set	1,964

Table 2: Sizes of Training, Validation, and Test Sets

4.2 Transfer Learning Approach

I used ResNet18 from PyTorch's *torchvision.models*. For this model in particular, I replaced the final fully connected layer with a new linear layer outputting 11 classes (one for each Pokémon type!).

4.3 Hyperparameters

Standard values were set for every hyperparameter, for the most part. Additionally, Data Augmentation parameters were included and tuned.

- **Epochs**: 10.
- **Batch Size**: 32.
- **Learning Rate**: 0.001, with Adam as an optimizer.
- **Augmentations**: I included random crops, flips, rotations, and color jitter so the model could generalise to varied card orientations and lighting conditions.
- **Image Size 224×224**: This was done to ensure consistency with the pre-training setup of ResNet.

4.4 Training Process and Results

The entire training process takes about 40 to 45 minutes on an RTX 3060 using CUDA cores. As for the training, throughout the 10 epochs the model steadily improved in both training and validation performance, with training accuracy rising from about 81.6% to 94.8%, and validation accuracy reaching as high as 95.4%. Although there was a very *sharp* dip on the fourth epoch, it recovered quickly. By the final epoch, the train and validation accuracies were both in the mid-90% range, and the loss values had consistently decreased.

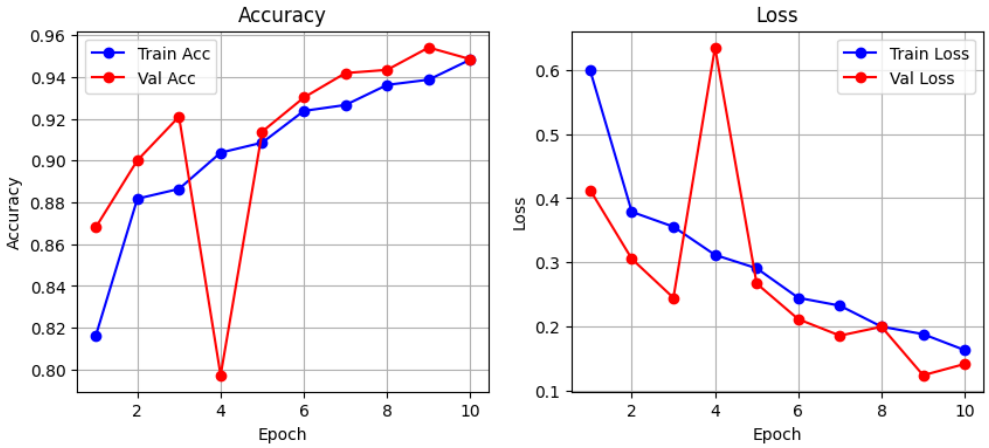

Figure 2: Accuracy and Loss during Training

When evaluated on the test set, the final model achieved a 95.82% accuracy with a loss of 0.1286. The model performed well on training and validation data, but also maintained high accuracy on unseen cards. Finally, model weights were saved to be used for the second part of the project: live detection.

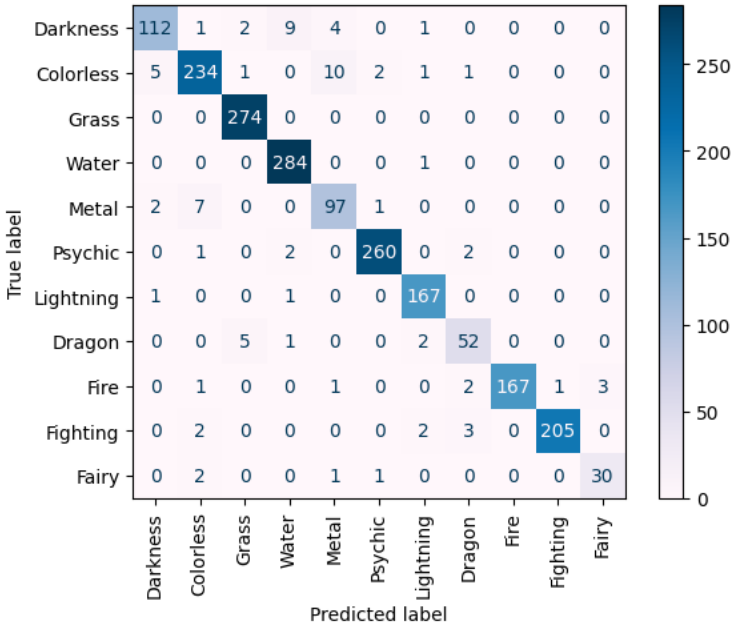

Figure 3: Confusion Matrix for Test Set

5 Real-Time Detection System

5.1 Warning!

This is a *very* raw implementation of real-time detection using OpenCV. My original idea was to also train a separate model on the detection of card shapes, but I did not know how to proceed in this regard nor was I able to find a suitable dataset for the task. In its current iteration, the model can only detect vertical cards, only one card at a time, and under ideal cold light conditions. Nevertheless, it was a good exercise of learning how OpenCV works.

5.2 Contour-Based Approach

I used the IP stream app to stream live video directly to my classification script. Each incoming video frame is first converted to grayscale using OpenCV's cv2.cvtColor function, which simplifies the image and makes it easier to detect edges. To reduce noise and smooth out the image, I applied a Gaussian blur with cv2.GaussianBlur. After blurring, I used the Canny edge detection method (cv2.Canny) with threshold values of 60 and 155, although I also experimented with 100 and 200 to find the best settings under my -abysmal- different lighting conditions. This process creates a binary edge map that highlights the significant edges in each frame.
Next, the script looks for contours in the edge map using cv2.findContours. It focuses on the largest external contour, operating under the assumption that the Pokémon card will be the most prominent rectangular shape in the frame. By approximating each contour to a polygon with four corners using cv2.approxPolyDP, the script identifies potential card boundaries based on their rectangular shape. Once a suitable four cornered contour is found, the script calculates its bounding rectangle with cv2.boundingRect and crops this area from the original frame. This effectively isolates the card, removing any surrounding background and is then ready for classification by the trained model.

5.3 Classification and Overlay

After detecting and cropping the Pokémon card from each frame, the preprocessing pipeline ensures that the image is in the optimal format for classification. The cropped region is first resized to 224×224, aligning

with ResNet18's expected input size. The image is then converted from its original color space to RGB and, following this, the RGB image is transformed into a tensor with dimensions corresponding to channels, height, and width (C×H×W) using PyTorch's transforms.ToTensor(). Once the ROI is properly transformed, it is fed into the ResNet18 model, which processes the image and identifies the most likely type based on the prediction. For visual feedback in the live video feed, I added a bounding box around the detected card region and overlaid the predicted type name along with a corresponding icon. This was done using OpenCV's drawing functions, and the annotated frames were displayed in real-time via cv2.imshow.

5.4 Testing

I tested the real-time classification on at least 10 separate occasions under different lighting conditions. Best results were achieved under cold natural light between 10:00 and 11:30, and with a black background as a base. Also, for both tests, a bottle was used to stabilise the camera.
Tests were conducted with cards from the Paradox Rift (2023) and Obsidian Flames (2024) sets. These cards were not included in the training dataset. Below are clickable links to YouTube videos with two of the testing instances.

Early Attempt
For this early attempt, 4 cards across 4 different Pokémon types were tested. All four cards were correctly classified; however, as I mentioned before, due to the raw implementation of the detection, only vertical cards can be classified. This was apparent for the third card, that could not be classified at first because it was rotated to the left.

Latest Attempt
For the latest attempt, 10 cards across 9 different Pokémon types were tested. I consider a win Colorless and Metal cards were not misclassified, due to their similar color appearance. Also, the second fire card was also correctly classified, even though a part of it was of a different color. The last card, a Lightning type, was misclassified, which I assume it might be attributed to the lighting conditions.

6 References

References

[1] OpenCV. (n.d.). *OpenCV: Open Source Computer Vision Library*. Retrieved January 5, 2025, from `https://opencv.org`

[2] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research, 12*, 2825–2830. `https://scikit-learn.org`

[3] Peixoto, H. (2021). *ptcg-detection* [Source code]. GitHub. `https://github.com/hugopeixoto/ptcg-detection`

[4] PyTorch. (n.d.). *PyTorch: An open source machine learning framework*. Retrieved January 5, 2025, from `https://pytorch.org`

[5] TheFusion21. (2024). *PokemonCards [Dataset]*. Hugging Face. Retrieved December, 2024, from `https://huggingface.co/datasets/TheFusion21/PokemonCards`

[6] Wijekoon, H. (2024). *pytorch-cnn-playing-cards-classifier* [Source code]. GitHub. `https://github.com/hiroonwijekoon/pytorch-cnn-playing-cards-classifier`