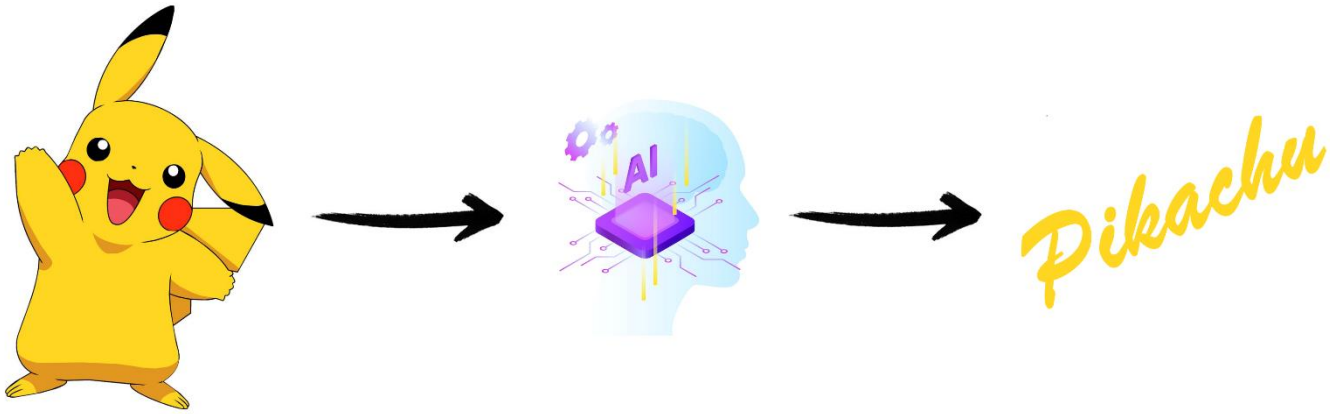
Computer Vision Fundamentals Project

Pokémon images classification

Valentin Hervé



25/06/2024

Table of contents

1. Introduction

The main aim of the project is to utilize computer vision techniques to train a model so that, given an image representing a Pokémon, it can tell which Pokémon it is. The model can be trained using images from video games, animations, drawings, or even physical objects, giving us a wide variety of data. This project is relevant as it demonstrates the practical application of image classification in the context of popular culture and gaming. This project could, for example, be used to build a larger artificial intelligence that could play Pokémon games on its own. Our project could recognize Pokémon in the wild, and another might use this data to build a team that could defeat it, for example.

You can find the project on my github account:
[1]https://github.com/Vazelek/ComputerVision_M1Project

2. Image dataset

a. The data

To obtain my data, I use an existing dataset on Kaggle, which allows me to obtain the images I want from the various sources mentioned above. The dataset I found contains around 200 images for each of the 151 first-generation Pokémon, for a total of 35,000 images. The images are already classified, but as the number of images available for each Pokémon is different, I'll have to go into each class and divide them between the training and test datasets. Here is the link of the dataset I'll use: [2]https://www.kaggle.com/datasets/echometerhhwl/pokemon-gen-1-38914.

In order to enrich this dataset, I went looking for images representing Pokémons from different sources. I also went directly into the games to take a few screenshots. You can find the added images at this link: [3]https://drive.google.com/file/d/10YbsdKGSgny1GKtRPIWGWaibJw-kA6b6/view?usp=sharing.

Examples of custom data added from the Game Freak game: [4]Pokémon Let's Go, Eevee!:

b. Preprocess the dataset

The first thing to do is to separate the data into a training and a validation dataset. The solution adopted is simply to copy a certain percentage of the data for each class into the training dataset and copy the rest into the validation dataset. We thus have 80% of the images for each class in the training dataset and 20% in the validation dataset.

The raw images could not be used directly by the models, as they vary greatly in size, shape and resolution. We need to preprocess them in order to have similar images in their shape. To do so, I chose to crop the images at their center with the following function:

```
def load_dataset(data_dir, img_size=(64, 64)):
    images = []
    labels = []
    class_names = sorted(os.listdir(data_dir))
    for label, class_name in enumerate(class_names):
        class_path = os.path.join(data_dir, class_name)
        if os.path.isdir(class_path):
            for img_name in os.listdir(class_path):
                img_path = os.path.join(class_path, img_name)
                if img_path.endswith(('.png', '.jpg', '.jpeg')):
                    try:
                        img = Image.open(img_path).convert('RGB')
                        width, height = img.size
                        min_dim = min(width, height)
                        left = (width - min_dim) / 2
                        top = (height - min_dim) / 2
                        right = (width + min_dim) / 2
                        bottom = (height + min_dim) / 2
                        img = img.crop((left, top, right, bottom))
                        img = img.resize(img_size)
                        img = np.array(img, dtype=np.float32)
                        images.append(img)
                        labels.append(label)
                    except OSError as e:
                        print(f"Skipping corrupted image: {img_path} -
{e}")
    return images, labels, class_names
```

This function does a lot of things:

- It goes through all the folders, allowing us to get the labels of the images we process,
- It converts images into the RGB format (because we can have images with transparency),
- It crops the images at their center in order to obtain square shaped images,
- It resizes the images to 64 by 64 pixels (this resolution is chosen so as to have sufficient quality to recognize features, but not too high either to have enough RAM, given that the dataset includes more than 35,000 images),

- And it returns the processed images, their labels and all the classes that exists in the dataset.

Applying this function to the training and the validation directory will allow us to have every data we need to train our models:

- Training and validation images
- Training and validation labels
- Class names

Finally, to avoid having to process these images and reseparate them each time, we save these variables using pickle. We'll then just have to load them if we run the program again.

3. Train models

a. State-of-the-art models

I have chosen to train the following 3 image classification state-of-the-art models on my dataset:

- resnet18
- densenet121
- mobilenet_v2

These different models are trained over 10 epochs, trying to achieve the best accuracy and F1 score and the minimum loss on the training data. At each epoch, the training and validation loss and accuracy are saved, along with the f1 score, so that we can see how these data evolve as training progresses. A confusion matrix is also created at the end of training. Once trained, the parameters of these models are saved so that they do not need to be re-trained.

b. Custom model

The main aim of this project is to create a custom model and to compare it with these three state-of-the-art models. To do so, we will use the Keras python library as follows:

```
def create_custom_model(input_shape, num_classes):
    model = Sequential([
        Conv2D(16, (3, 3), activation='relu', input_shape=input_shape),
        BatchNormalization(),
        MaxPooling2D((2, 2)),
        Conv2D(32, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(64, activation='relu',
kernel_regularizer=keras.regularizers.l2(0.001)),
        Dropout(0.2),
        Dense(num_classes, activation='softmax')
    ])
    model.compile(optimizer="adam",
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    return model
```

This code creates a custom made convolutional neural network (CNN) model using the Keras library:

- We first initialize a sequential model,
- We then add layers one by one:
    - Conv2D: This adds a convolutional layer with a certain number of filters, and a 3x3 kernel,
    - BatchNormalization: This normalizes the outputs of the convolutional layers to improve training speed and stability,
    - MaxPooling2D: This reduces the feature maps by taking the maximum value in each 2x2 pool,
    - Flatten: This converts the 2D matrices to a 1D vector,
    - 1st Dense: This adds a layer which will penalize large weights to prevent overfitting,
    - Dropout: This drops 20% of the units randomly during training to prevent overfitting,
    - 2nd Dense: This adds the output layer with a number of units equal to the number of classes for multi-class classification.
- We then compile the model using the Adam optimizer, using the accuracy metrics to monitor the training and testing steps.

All the parameters and the number of layers used to create this model have been tested in order to have the best results. The custom model is then trained for 10 epochs, which is a sufficient number of epochs because overfitting occurs if we go any further. Metrics such as training and validation loss and accuracy, as well as the f1 score, are saved in the history variable so that we can compare them with the one that we obtain we the state-of-the-art models.

```
for epoch in range(num_epochs):
    history = model.fit(train_images, train_labels, epochs=1,
                        validation_data=(test_images, test_labels),
                        verbose=0)
```

Once trained, the parameters of this custom model are saved so that it does not need to be re-trained.

4. Model comparison

Once we have trained all these models and retrieved the metrics, we can display them using the matplotlib library.

a. Train loss and accuracy

The main aim of training is to find the parameters of our model that give the best accuracy on our training data. Precision and loss are inversely proportional, meaning that high precision leads to low loss. [5]The loss quantifies the difference between the predicted values and the actual values, where the accuracy measures the percentage of correct predictions out of all predictions made.

Taking all this into account, we should see for each model an increase in accuracy and a reduction in loss over time. Here's the result we get:

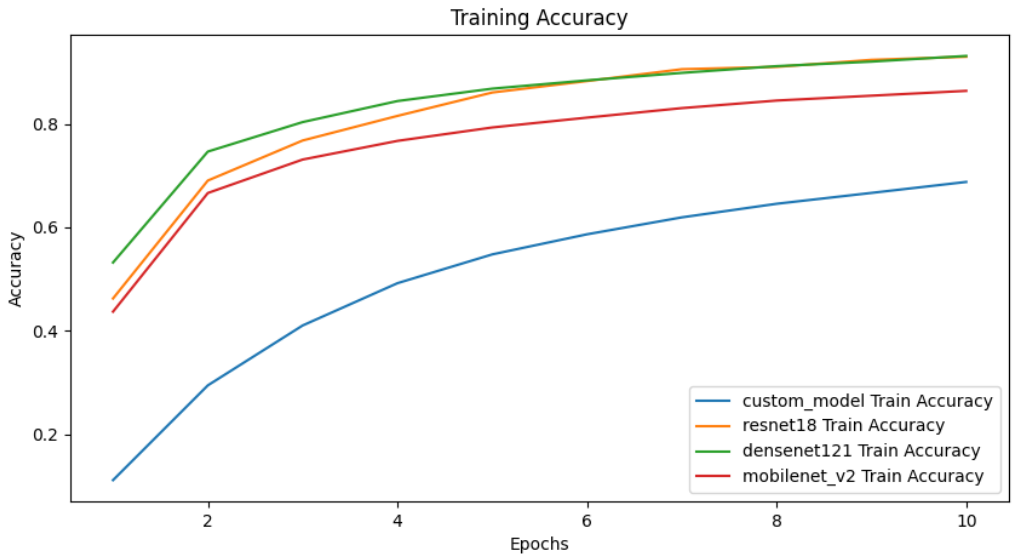


Figure 1: Training accuracy of each model by epoch

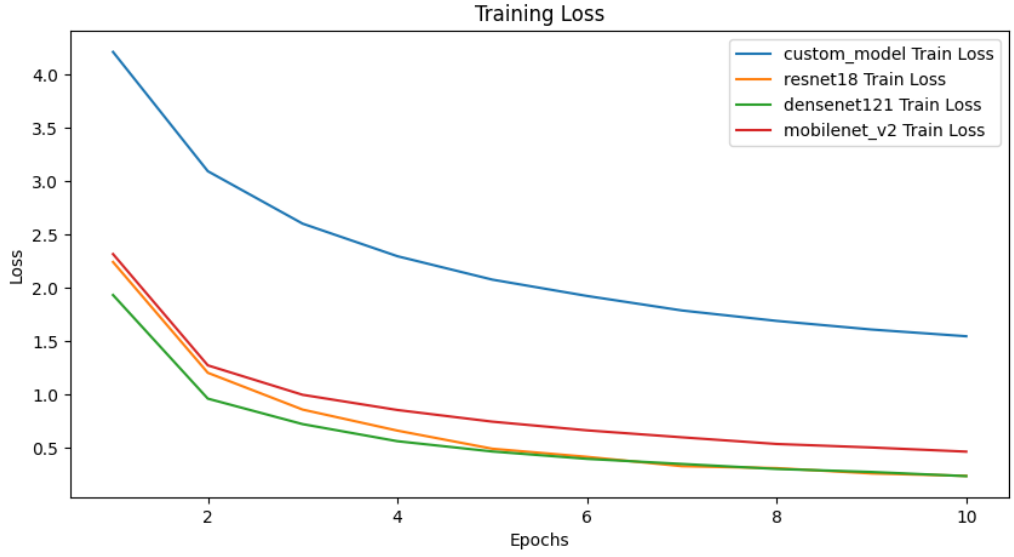


Figure 2: Training loss of each model by epoch

As expected, there has been a steady increase in precision and a steady decrease in loss over the epochs for all models. We can see from these metrics that our custom model performs less well than the state-of-the-art models, but it still has good results.

b. Validation loss and accuracy

The validation dataset allows us to "validate" our training by testing the models with data unknown to them. Whereas it was normal to observe a steady decrease in loss and a steady increase in accuracy on the test base, here we should have different data to analyze:

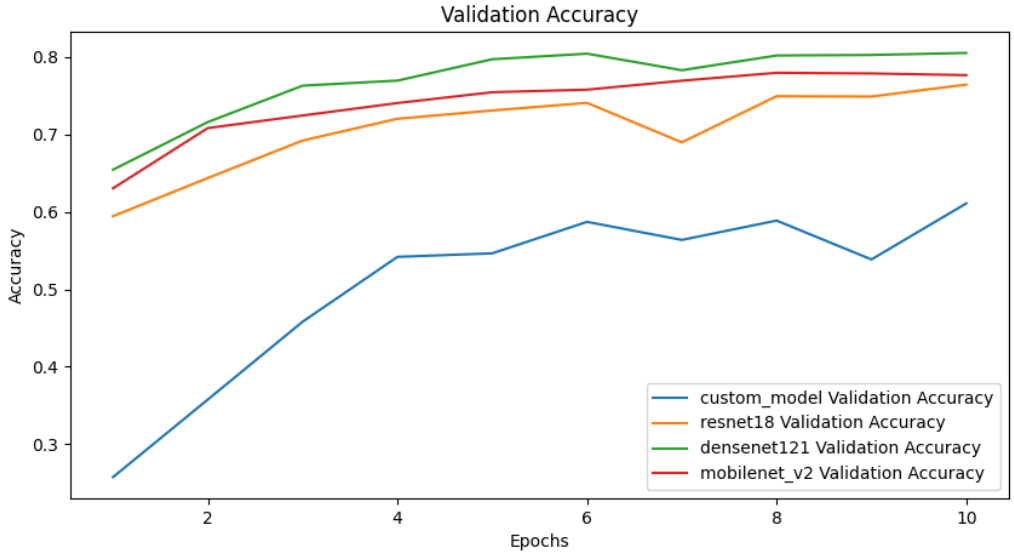


Figure 3: Validation accuracy of each model by epoch

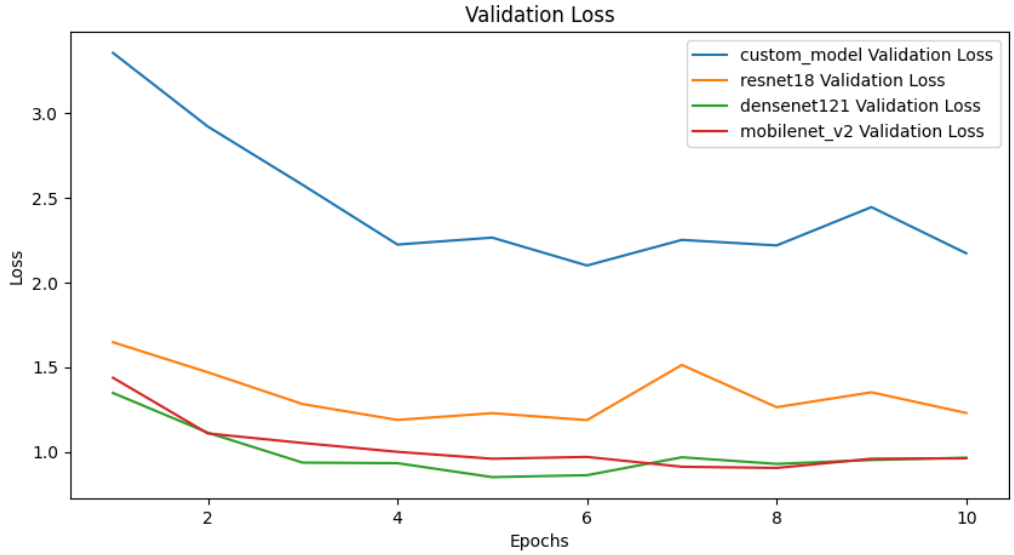


Figure 4: Validation loss of each model by epoch

We can see here that the loss initially decreases for each model on the validation base, but then starts to increase. This is linked to the concept of "overfitting". This happens when a

model trains too much on a particular set of data and is unable to adapt to other, unfamiliar data. If a model trains too much on fixed data, then it will be excellent at recognizing them, but will fail to find commonalities with unknown data. This is why, in order to prevent this, I've tried to include layers like Dropout to limit overfitting.

c. F1 score

[6]The F1 score is an important metrics used in images classification. It takes in account both the precision and the recall. The precision represents the accuracy of positive predictions, it is the number of true positive divided by the total of positive predicted. The recall represents the capacity of the model to identify actual positive cases. It is the number of true positive divided by the actual number of positive. Combining these two metrics gives the F1 score. It can be better than accuracy, for example, because it also takes into account false positives and false negatives, giving a more important understanding of our model's current performance.

Here is what we got as F1 score in our validation dataset for each epoch:

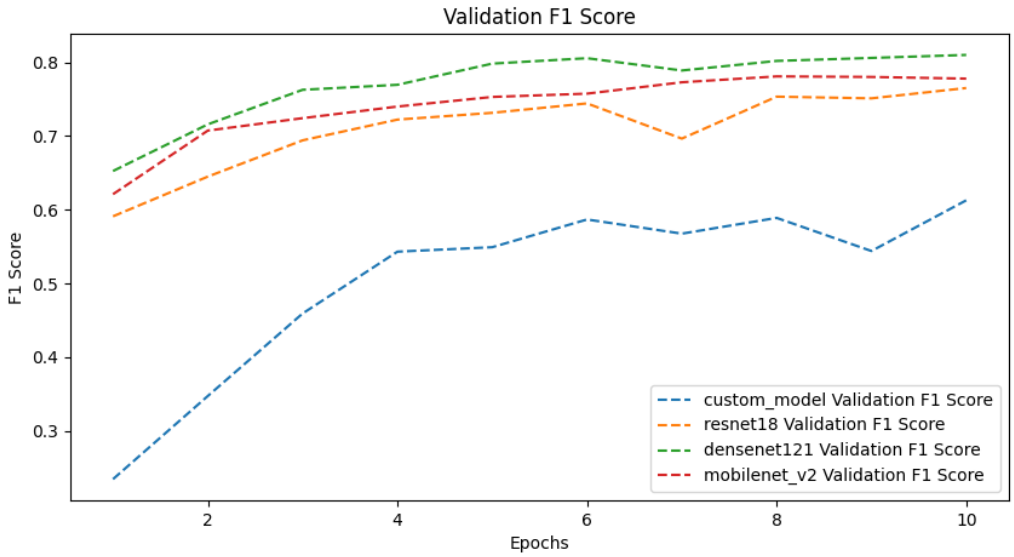


Figure 5: Validation F1 score of each model by epoch

In our specific case, we can see that the f1 score follows the overall accuracy trend on the validation dataset.

d. Confusion matrices

The confusion matrix measures the quality of a classification system. Each row corresponds to a real class, and each column to an estimated class. In a way, it shows where the model went wrong. For a high-performance model, we should see a diagonal on the confusion matrix, meaning that the images have been predicted correctly, whereas a low-performance model will have its results scattered all over the confusion matrix.

Here are the confusion matrices for our four models:

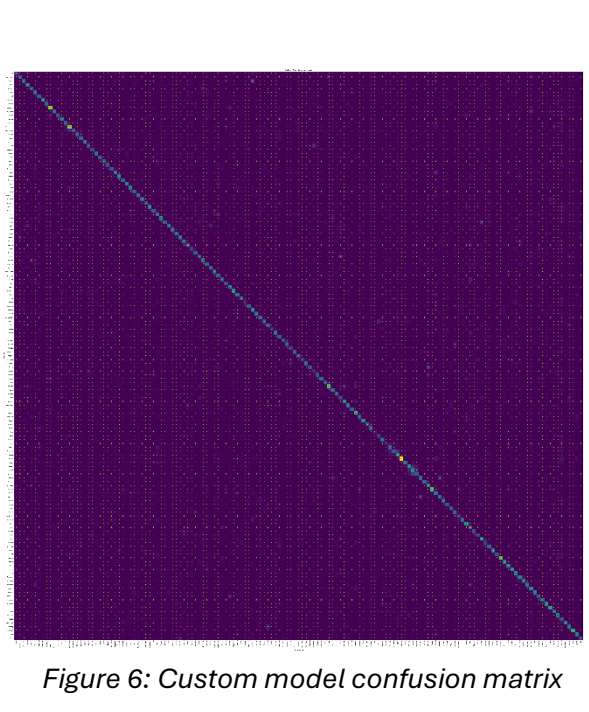


Figure 6: Custom model confusion matrix

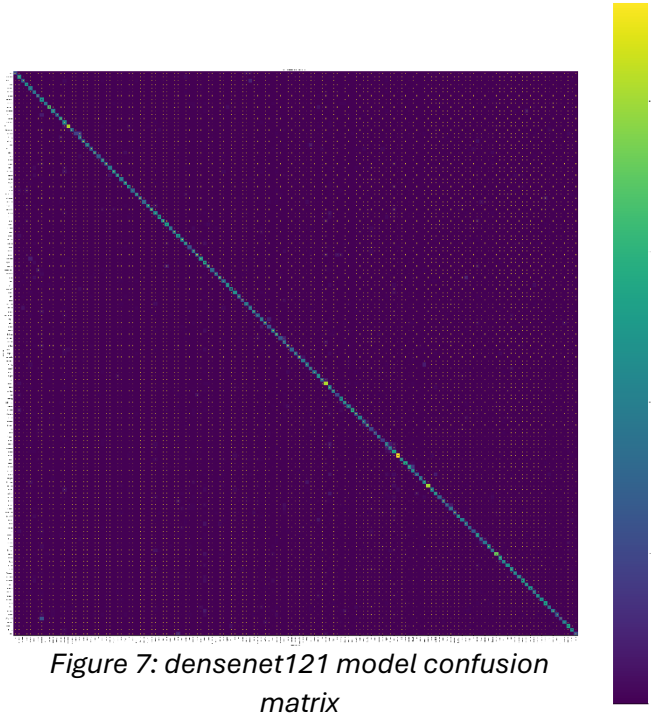


Figure 7: densenet121 model confusion matrix

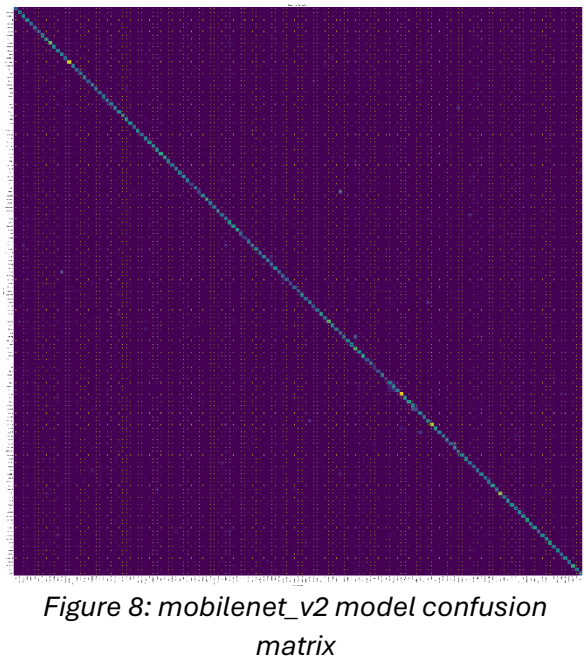


Figure 8: mobilenet_v2 model confusion matrix

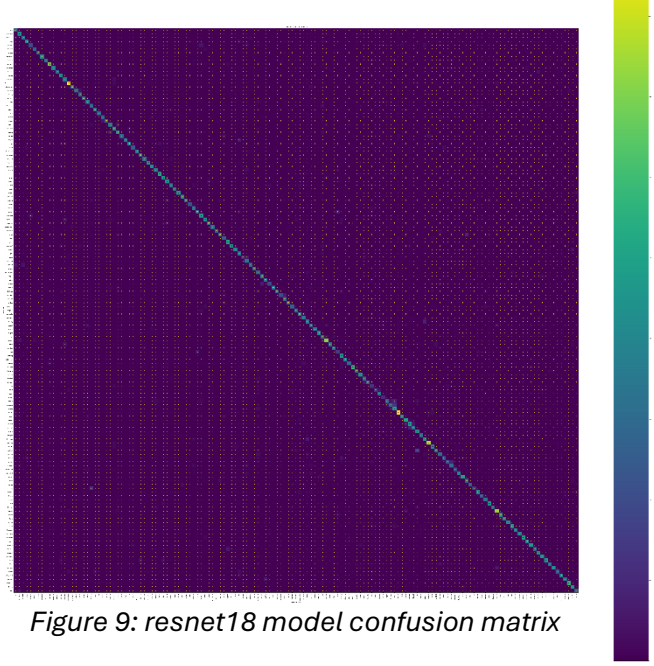


Figure 9: resnet18 model confusion matrix

With this number of classes, this is unreadable as it is, but we can see the diagonal I was talking about for all these models. We can zoom in and see what we have:

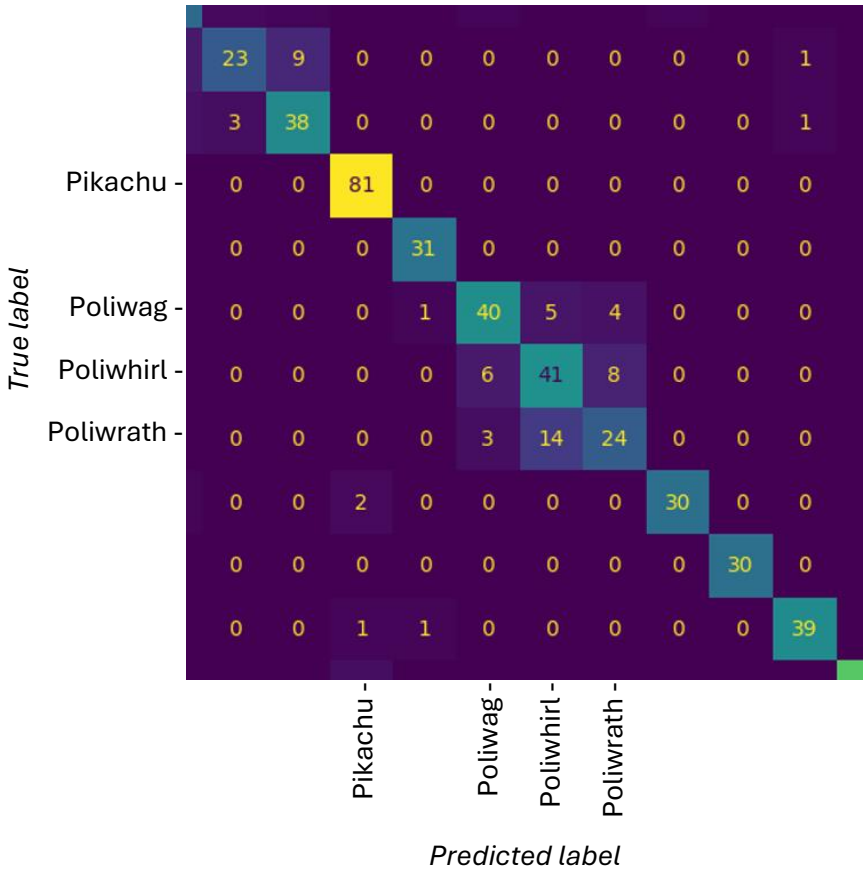


Figure 10: Custom model confusion matrix

As we can see, with our custom model, 81 Pikachu have been well predicted. We can also see that some of them are not, but most of them are. We can also see a lot of errors for Poliwag, Poliwhirl and Poliwrath. The model seems to be confused with them. This is likely due because these 3 Pokémons have a very similar design:

Source: [7]https://www.deviantart.com/pokeyfakemon/art/060-Poliwag-Poliwhirl-Poliwrath-933239429 by PokeyFakemon

e. Training time

The last but not least metrics that we have is the time the models have spent training:

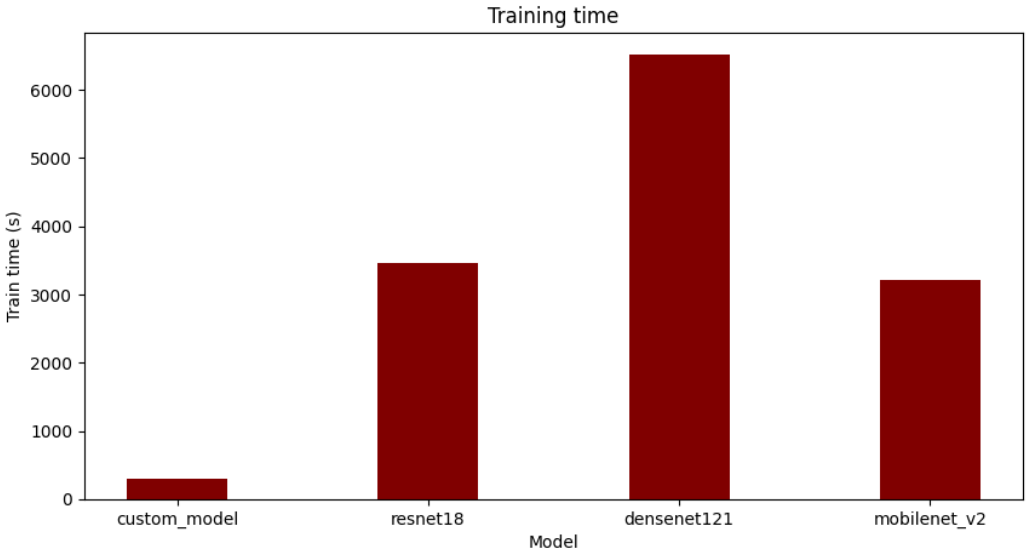


Figure 11: Models training time

Here we can see that our custom model took much less time to train than the others (due to the model's small number of layers). Overall, the training time was not interminable, given the large size of the dataset. The longest model took just under two hours to complete ten epochs.

f. Model comparison - conclusion

If we look at the performance of the models, particularly in relation to the f1 score, we see that densenet is the most efficient model with an F1 score close to 0.8 (/1), with our custom model the least efficient (around 0.6). But if we also take execution time into account, we realize that the densenet model is almost twice as slow to train as resnet or mobilenet, which nevertheless have F1 scores very close to densenet. We can therefore conclude that the mobilenet model is the most efficient on our database in terms of F1 score / training speed ratio.

5. Prediction tests

To see the results of training our models, I decided to add a python script that takes as input a folder containing images and displays the classification obtained from these images according to each model. Here is an example of result we obtain:

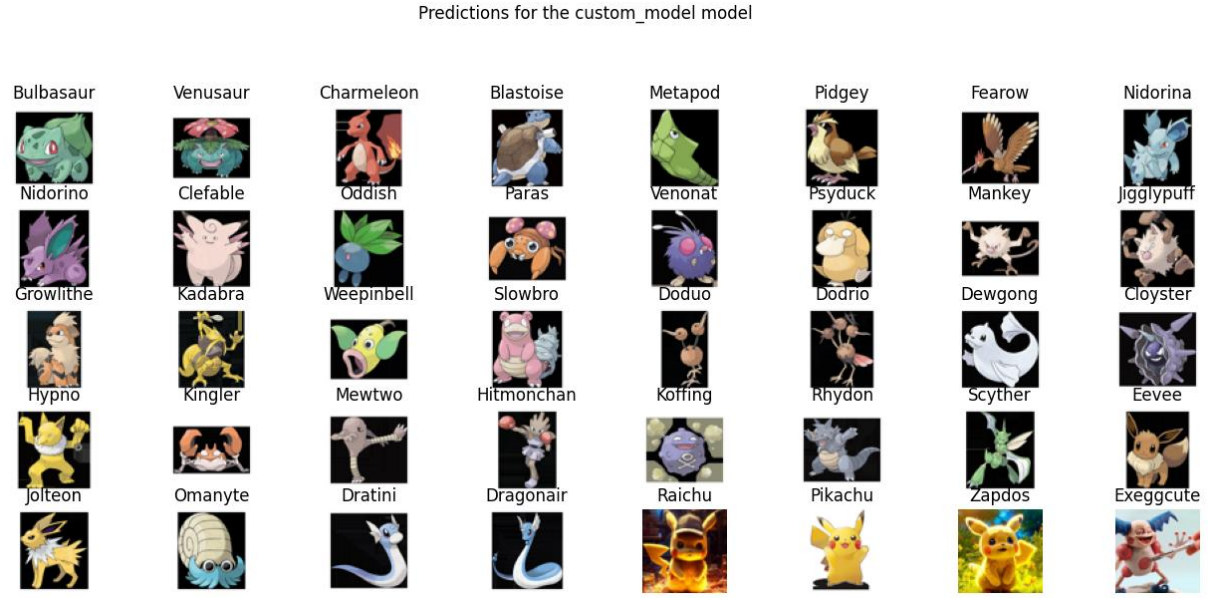


Figure 12: Class prediction with the custom model

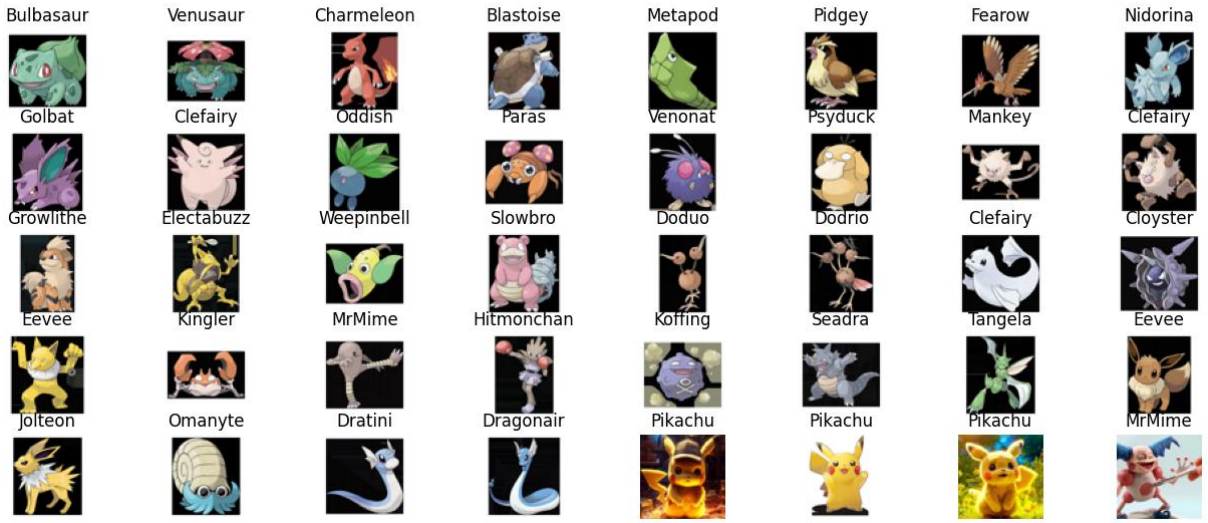


Figure 13: Class prediction with the densenet model

We can see mainly the same good results, with a little more precision from the densenet model overall, even if sometimes the custom model gives a true result unlike the densenet model (for Scyther or Kadabra for instance).

6. References

[1] Valentin Hervé's GitHub project repository: https://github.com/Vazelek/ComputerVision_M1Project
[2] Kaggle, *Gen 1 Pokemon (35k images)*, ECHOMETER HHWL: https://www.kaggle.com/datasets/echometerhhwl/pokemon-gen-1-38914
[3] CustomeImages folder: https://drive.google.com/file/d/10YbsdKGSgny1GKtRPIWGWaibJw-kA6b6/view?usp=sharing
[4] *Pokémon Let's Go, Eevee!*, 2018, Game Freak
[5] Geeks for Geeks, *What is the relationship between the accuracy and the loss in deep learning?*: https://www.geeksforgeeks.org/what-is-the-relationship-between-the-accuracy-and-the-loss-in-deep-learning/
[6] Geeks for Geeks, *F1 Score in Machine Learning*: https://www.geeksforgeeks.org/f1-score-in-machine-learning/
[7] Image by PokeyFakemon, DeviantArt: https://www.deviantart.com/pokeyfakemon/art/060-Poliwag-Poliwhirl-Poliwrath-933239429