Deep Learning

Task 1

Team Rocket

Members:

David Castro nº 60973
Guilherme Antunes nº 70231
José Morgado nº 59457
Ricardo Rodrigues nº 72054

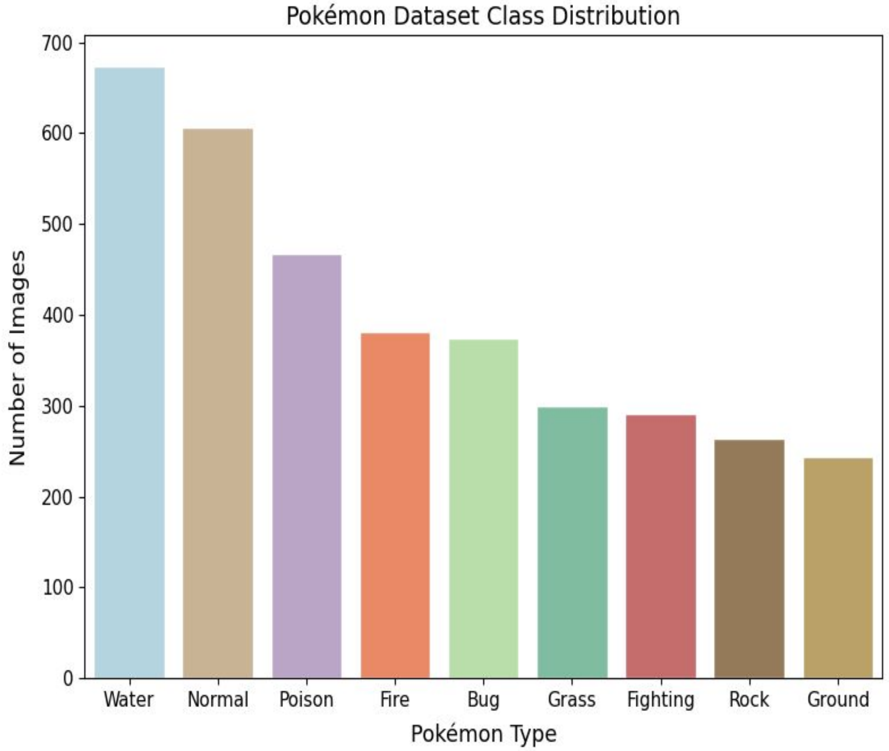
Dataset Analysis

In this graph we can see all the distribution of all classes. There are 3600 total images, and 9 classes in total.

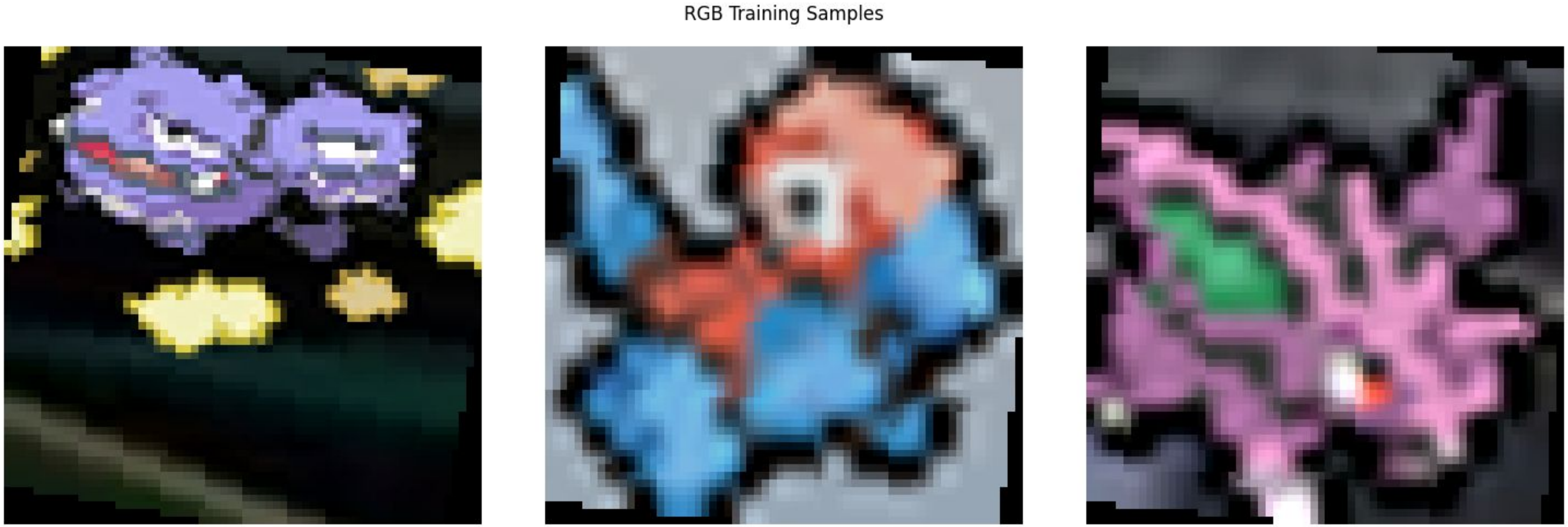
Water: 674
Normal: 606
Poison: 467
Fire: 381
Bug: 374
Grass: 299
FIghting: 291
Rock: 264
Ground: 244

With this distribution we'd expect a better performance identifying water/normal types since they have more than double the samples of the least prevalent type.

One of the most significant breakthroughs in our project came after inspecting the dataset images. We discovered that each Pokémon was outlined by a distinctive black silhouette. Leveraging this, we implemented a function to detect and group black pixel coordinates of the silhouette, then used the xmin, ymin, xmax, ymax method to draw a bounding box around the pokemon. We cropped each Pokémon with 5% padding and resized it to 64x64 pixels, reducing background clutter. This allowed the model to focus solely on the Pokémon, leading to a much more accurate and efficient training process. As a result, our model's F1 score saw a dramatic improvement, jumping from 0.22 to 0.8975 on Kaggle.



The cropping process is not perfect, especially when the background has other dark colors but it works for a generous amount of entries



Model Structure

Our model was trained on an MLP (Multilayer Perceptron) architecture, in order to classify and label Pokémon images from a dataset.

The input layer of our MLP takes a flattened 64x64 image (after preprocessing, such as cropping and resizing) as a vector. Given that each image is resized to 64x64 pixels, this results in an input size of 4096 features (64 * 64). Our model has two hidden layers, both fully connected, one with 128 neurons and the other with 256, followed by a Batch Normalization to stabilize learning and speed up convergence. The activation function used in both layers was ReLU, which introduced non-linearity to our model and increased its expressiveness.

A dropout rate of 20% is applied after each hidden layer. This helps reduce overfitting by randomly setting a fraction of input units to zero at each update during training, thereby forcing the model to generalize better. The dropout rate is set at 20%, as we already apply Batch Normalization to each hidden layer, which further helps prevent overfitting.

The final layer of the MLP, the output layer, is a fully connected layer with num_classes neurons, corresponding to the number of Pokémon categories in the dataset. A softmax activation is applied (implicitly by the CrossEntropyLoss function) to produce the probability distribution over all classes.

The model is trained using Cross-Entropy Loss for classification, and the optimizer of choice is Adam. To adapt the learning rate throughout training, a StepLR learning rate scheduler is used with a step size of 5 epochs and a gamma of 0.1, helping the model converge more smoothly.

To train our model we split the training dataset into train, test and validation datasets, with a split of 70/15/15.

Performance Metrics

The total time it took to train our best model was 3709.72 seconds, or 1.1h, using a Google Colab runtime with a T4 GPU.

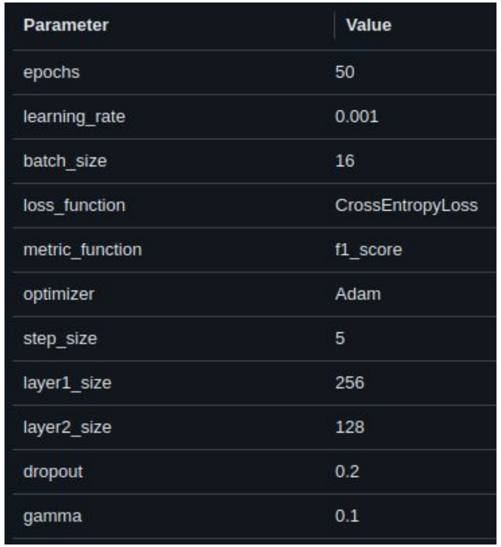
We got a local test accuracy of 90.37%

Test F1 Score: 90.22%

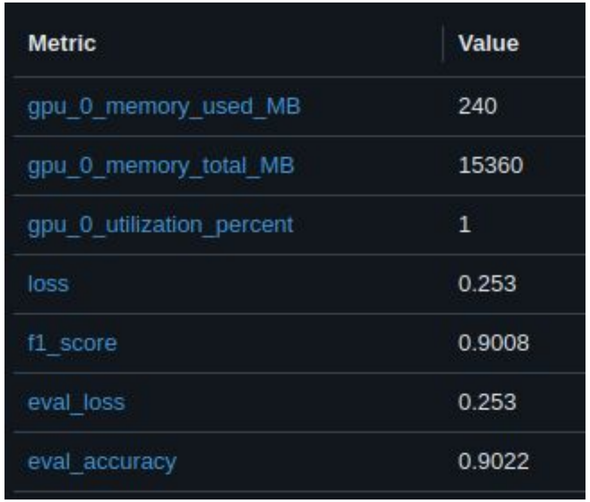
These metrics were obtained from mflow.

Since our class distribution is skewed, accuracy is not a great way to predict model reliability.

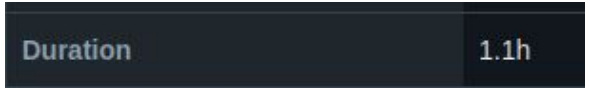
This high F1 score indicates that our model has good balance between precision and recall scores across all classes.



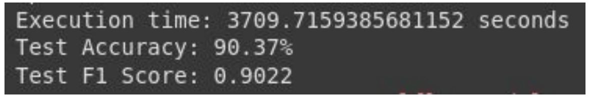
Parameter	Value
epochs	50
learning_rate	0.001
batch_size	16
loss_function	CrossEntropyLoss
metric_function	f1_score
optimizer	Adam
step_size	5
layer1_size	256
layer2_size	128
dropout	0.2
gamma	0.1



Metric	Value
gpu_0_memory_used_MB	240
gpu_0_memory_total_MB	15360
gpu_0_utilization_percent	1
loss	0.253
f1_score	0.9008
eval_loss	0.253
eval_accuracy	0.9022



Duration	1.1h



```
Execution time: 3709.7159385681152 seconds
Test Accuracy: 90.37%
Test F1 Score: 0.9022
```

Kaggle Submission

Task 1:

Public Score - 0.91046

Private Score - 0.89633

Leaderboard Ranking - 2nd

Parameters - batch size 16, 50 epochs, learning rate 0.001, step size 5, layer 1 256, layer 2 128, dropout 0.2, gamma 0.1

Insights Gained

We learned that MLPs are not great at distinguishing the Pokemón from the background, this explains why our results improved so much after removing the background so the Pokemón takes up most of the image.

What went wrong

Before cropping the images to remove the background we were struggling to understand how some groups were getting score considerably higher than the rest, this made us lose quite some time tweaking the model to try and find what we were missing.

What went great

The best thing that happen to us on this task was realizing that, since MPLs have a hard time distinguishing the background from the target, we could try to remove the background to help the model. And it worked very well, this change was what gave us the most expressive improvement.

Deep Learning

Task 2

Team Rocket

Members:

David Castro nº 60973
Guilherme Antunes nº 70231
José Morgado nº 59457
Ricardo Rodrigues nº 72054

Dataset Analysis

Initially we tried to use the same dataset we used in task 1, with the cropped images to remove the background, and while we got very good results locally, on Kaggle our scores were around 0.20. We went back to the original dataset and immediately got a score of 0.52.

We believe this happened because of two reasons:

1 - CNNs are better at distinguishing the important part of the image from the background, so while removing the background helped our MLP model, the same improvements weren't seen on the CNN model;

2 - Our CNN model had a small kernel size and a stride of 1, thus it can pick up on details more easily. By removing the background we essentially made the pokémon take up the almost the entire image. We believe that our model was trying to pick up details that in reality were not there, hence the poor performance.

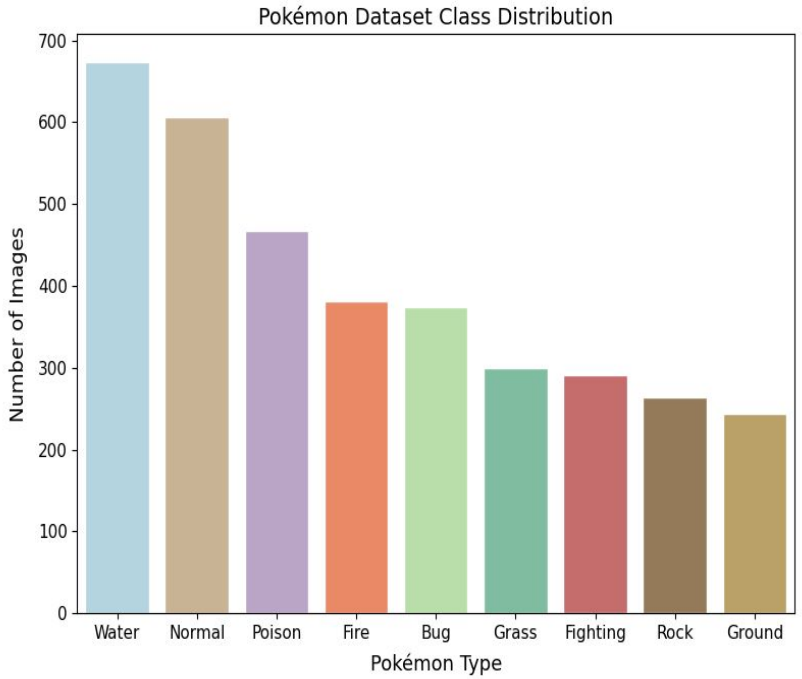
Dataset Analysis - Continuation

Due to the results we obtained initially, we decided to keep the original dataset, with the same class imbalances, and focused on designing a CNN that could take advantage of the Pokémon images' spatial structure.

The original dataset consists of 3600 images of 64x64 pixels in RGBA mode, with each image having a Pokémon on top of noisy background.

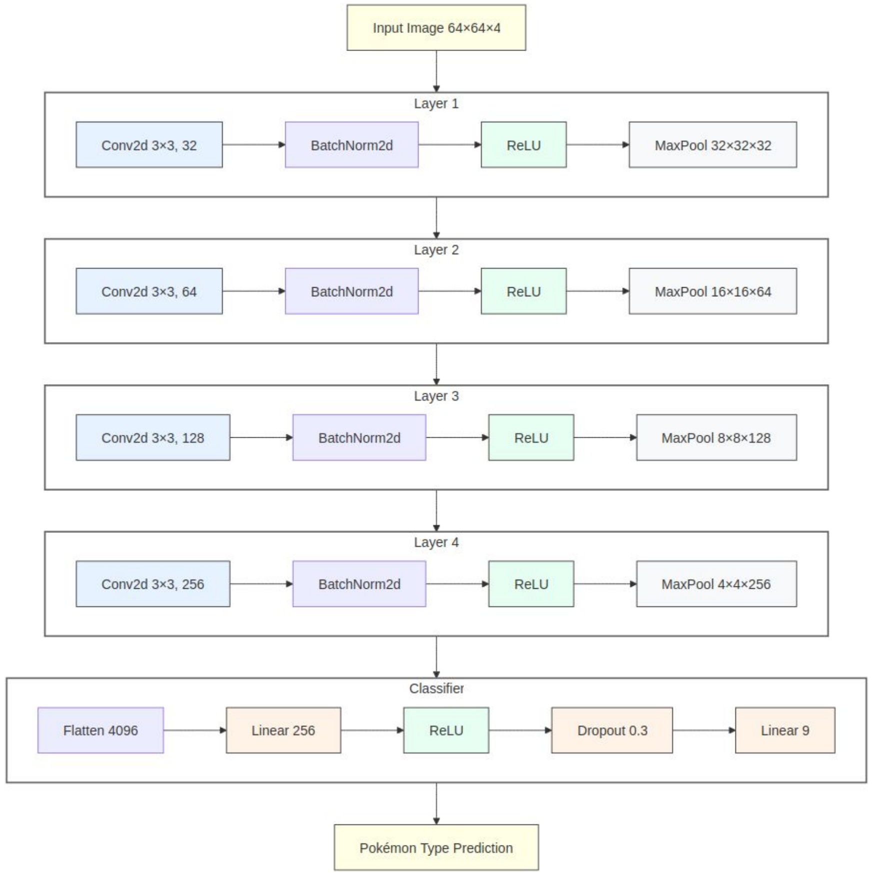
On this graph we can see the distribution of the Pokemón types.

The model might have a better accuracy for predicting the types with more representation in the training dataset, like water and normal, while it might have a harder time predicting the types with less representation like rock and ground.



Model Structure

While the MLP flattened images into 1D vectors, losing spatial relationships, our CNN preserves spatial structure through convolutional layers that detect features regardless of their position in the image. Our CNN uses a 4-stage convolutional structure, where each stage progressively learns more complex features. Each layer applies a 2D convolution, followed by batch normalization, a ReLU activation, and max pooling, which reduces the spatial size while keeping the most important information. Thanks to this structure, our model became much more robust at recognizing Pokémon even when they weren't centered, which meant we no longer needed to crop or center the images like we did with the MLP. On top of that, our CNN uses fewer parameters than an MLP would for the same task, making it more efficient and helping to prevent overfitting.



Performance Metrics

All models we tried had a batch size of 16, a learning rate of 0.001, and ReLu activation on every layer.

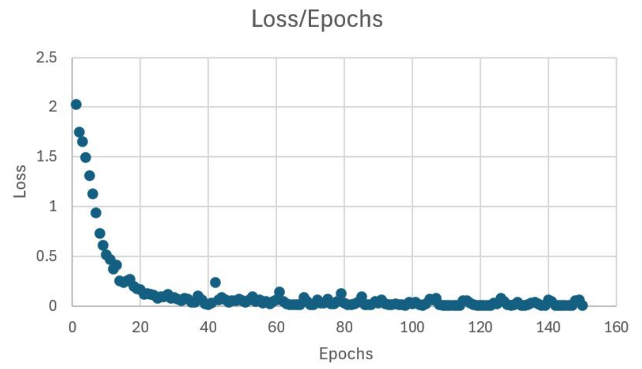
The first model we tried had two convolutional layers and got a Kaggle F1 score of 0.52, our second model had three layers and got a score of 0.9 on Kaggle. Both of these models were trained with 50 epochs.

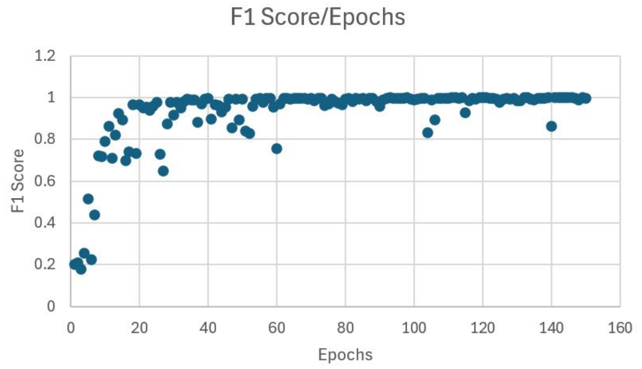
Our final model had four layers and we got scores between 0.99 and 1. Initially, we trained it with 150 epochs and we noticed that after around 50 to 70 epochs, the F1 scores for each epoch were stabilizing at values ranging from 0.97 to 1, with epochs with a score of 1 becoming more common after that point.

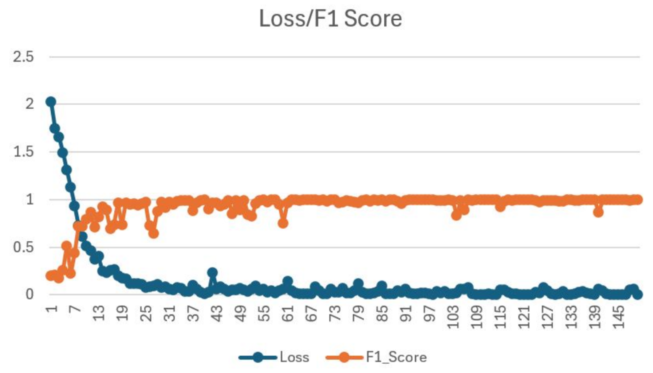
Performance Metrics - Training

We can see in these graphs that over the epochs the loss and F1 score behave as expected. The relationship between the loss and the F1 score is also as expected, higher F1 scores have better losses.

The most important thing to notice is that after around 50 epochs both the loss and F1 score seem to stabilize. This shows that 50 epochs are enough to train the model, and this conclusion is supported by our Kaggle results, both private and public.







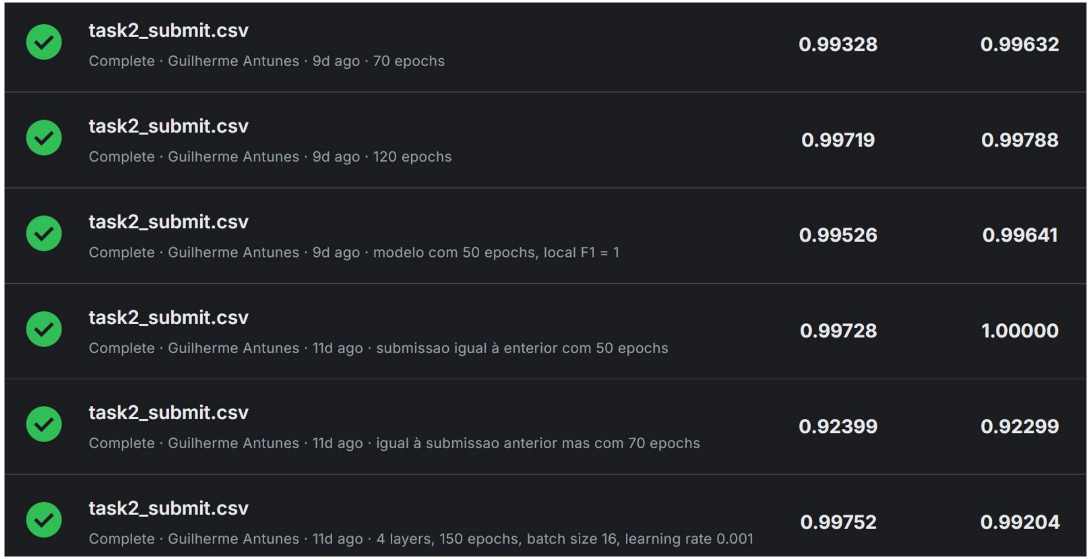
Performance Metrics - Continuation

We wanted to see if more epochs were actually needed, or if we could stop earlier and see similar results.

We trained the model with 50, 70, 120 and 150 epochs. All of our submissions had Kaggle scores higher than 0.99, except for our first submission with 70 epochs, which had a score of 0.92. We trained the model again with 70 epochs and got a score higher than 0.99, so it seems the score of 0.92 was just bad luck.

As for local scores, our final model was consistently above 0.99.

Our private Kaggle scores confirmed that the model converged at 50 epochs, all our submissions had scores above 0.99, except for the submission with a public score of 0.92, the private score for this one was also 0.92.

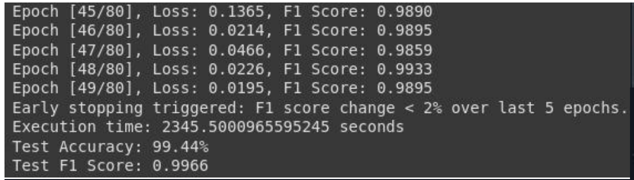


✓	task2_submit.csv Complete · Guilherme Antunes · 9d ago · 70 epochs	0.99328	0.99632
✓	task2_submit.csv Complete · Guilherme Antunes · 9d ago · 120 epochs	0.99719	0.99788
✓	task2_submit.csv Complete · Guilherme Antunes · 9d ago · modelo com 50 epochs, local F1 = 1	0.99526	0.99641
✓	task2_submit.csv Complete · Guilherme Antunes · 11d ago · submissao igual à enterior com 50 epochs	0.99728	1.00000
✓	task2_submit.csv Complete · Guilherme Antunes · 11d ago · igual à submissao anterior mas com 70 epochs	0.92399	0.92299
✓	task2_submit.csv Complete · Guilherme Antunes · 11d ago · 4 layers, 150 epochs, batch size 16, learning rate 0.001	0.99752	0.99204

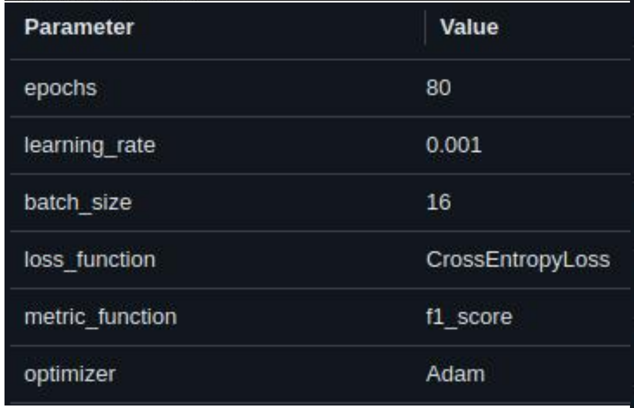
Training metrics

After realizing it was easy to train a model to reach an F1 score of 0.98 and up, we implemented an early stop technique to train on the least amount of epochs as possible. Our function stops training when the variance of the F1 score across the last 5 epochs is less than 2%.

With this method, our final model completed training in 49 epochs, taking 43 minutes on Google Colab, well below the assigned budget. Noteworthy was also the low resource usage, about 3% per epoch, which stayed constant throughout training.

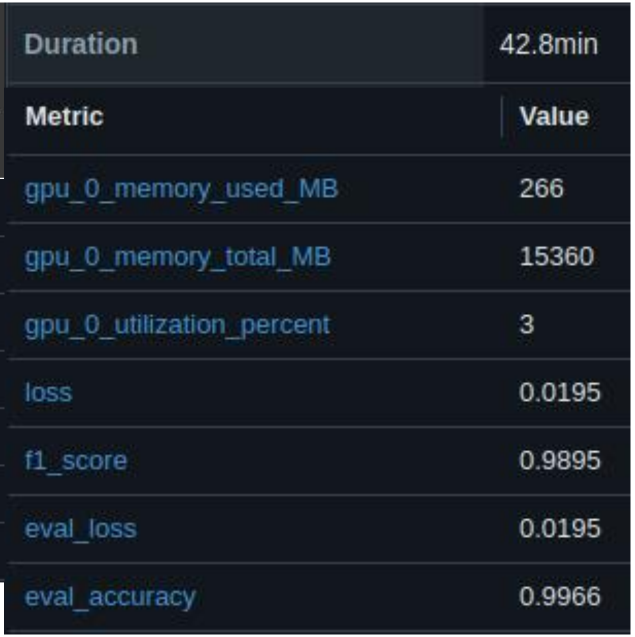


```
Epoch [45/80], Loss: 0.1365, F1 Score: 0.9890
Epoch [46/80], Loss: 0.0214, F1 Score: 0.9895
Epoch [47/80], Loss: 0.0466, F1 Score: 0.9859
Epoch [48/80], Loss: 0.0226, F1 Score: 0.9933
Epoch [49/80], Loss: 0.0195, F1 Score: 0.9895
Early stopping triggered: F1 score change < 2% over last 5 epochs.
Execution time: 2345.5000965595245 seconds
Test Accuracy: 99.44%
Test F1 Score: 0.9966
```



Parameter	Value
epochs	80
learning_rate	0.001
batch_size	16
loss_function	CrossEntropyLoss
metric_function	f1_score
optimizer	Adam

The epochs value here refers to the predefined maximum number of iterations



Duration	42.8min

Metric	Value
gpu_0_memory_used_MB	266
gpu_0_memory_total_MB	15360
gpu_0_utilization_percent	3
loss	0.0195
f1_score	0.9895
eval_loss	0.0195
eval_accuracy	0.9966

Kaggle Submission

Task 2:

Public Score - 0.99204

Private Score - 0.99752

Leaderboard Ranking - 3rd

Parameters: 49 epochs, batch size - 16, learning rate - 0.001

Execution time: 2345.5 seconds, or 39.1 minutes

Test Accuracy: 99.81%

Test F1 Score: 0.9976

What went wrong

Initially we were training the model using the dataset with cropped images and we were getting scores around 0.95, however, in Kaggle we got scores around 0.19. We were able to fix this issue by using the base, unchanged dataset, and got a score of 0.52 on Kaggle. Now, even though the backgrounds were noisy, at least they were consistent across all images, which the CNN was able to handle much better.

Since our model was able to reach near-perfect results just by adjusting the number of layers, we didn’t experiment much with other factors like learning rates, activation functions, regularization, or data augmentation, which could have been explored if needed.

What went great

In our opinion the best thing in this task was achieving a great model very early, after switching to the original training dataset, we started by trying different layer amounts (2, 3 and 4), and our results improved along with the number of layers (0.52, 0.90, 0.99) after just 50 epochs of training.

Another positive point was how efficient training turned out to be, managing to train our final model in only 49 epochs and 43 minutes with the use of an early stop technique, staying well below the time and resource budget.

Deep Learning

Task 3

Team Rocket

Members:

David Castro nº 60973
Guilherme Antunes nº 70231
José Morgado nº 59457
Ricardo Rodrigues nº 72054

Transfer learning in this context

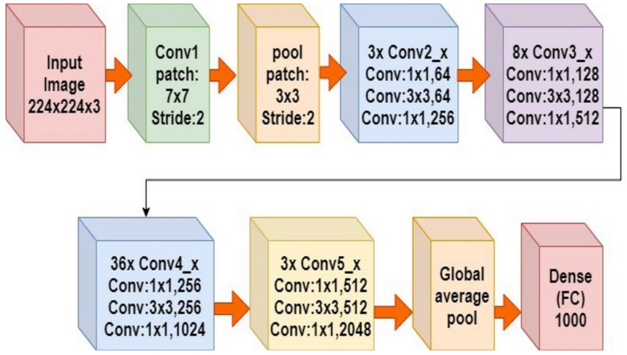
On the updated dataset provided, the complexity of the images was greatly increased, with a much higher resolution and more coherent images, it was now exponentially harder to be able to systematically identify what features and sections of the image were relevant for classification. Due to this and the relative small size of around 1200 training examples for such a complex task, it would not be feasible to train a model from scratch and get a decent result; considering this, the natural solution is to use transfer learning. Using a pre-trained model, such as ResNet, provides us with multiple advantages, namely: better efficiency with our limited data, capability of handling complex backgrounds and detection of visual features, faster convergence and robustness to real-world variation.

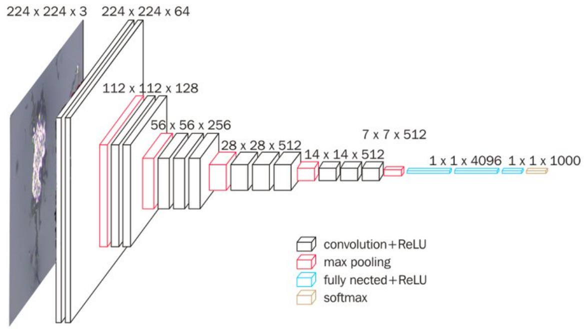
Model selection and justification

As mentioned previously, the size of our training data was relatively small, but it had a very important feature: our images were composed of diverse real-world backgrounds, with a superimposed image of the Pokémon. What this meant is that, by using a model pre-trained on images of the real world, such as ResNet, which was trained on ImageNet, its convolutional layers would already be capable of detecting different edges, textures and patterns in natural settings, helping separate these complex backgrounds from the Pokémon in the foreground without requiring extensive training, making the most out of our small dataset.

Model selection and justification

The pre-trained model we opted to use was ResNet50, with its architecture depicted in the top right figure. Initially, we attempted to use other ResNet models with less layers, but due to their poor performance, we gradually increased the number of layers of the model we used, settling on this for its balance of depth and computational efficiency. Another model we experimented with was VGG16, depicted in the bottom right, but its performance was also not comparable.


Input Image 224x224x3
Conv1 patch: 7x7 Stride:2
pool patch: 3x3 Stride:2
3x Conv2_x Conv:1x1,64 Conv:3x3,64 Conv:1x1,256
8x Conv3_x Conv:1x1,128 Conv:3x3,128 Conv:1x1,512
36x Conv4_x Conv:1x1,256 Conv:3x3,256 Conv:1x1,1024
3x Conv5_x Conv:1x1,512 Conv:3x3,512 Conv:1x1,2048
Global average pool
Dense (FC) 1000



Fine-tuning and Adaptation

The goal of applying transfer learning to this task is to use the base capabilities of feature detection of a model trained on a very large dataset, to adapt to our very specific task, which does not have a large training set. Having this goal in mind, we adapted the ResNet50 model for our problem using the following approach: first, we replaced the classification head with a custom MLP, featuring a single hidden layer with 512 nodes, ReLU activation and 50% dropout; then, we froze every layer except for the third and fourth and, with this setup, started our training using a very low learning rate. Replacing the classification head was virtually mandatory for this challenge, as the original model was trained to classify between 1000 classes, while we only have 9.

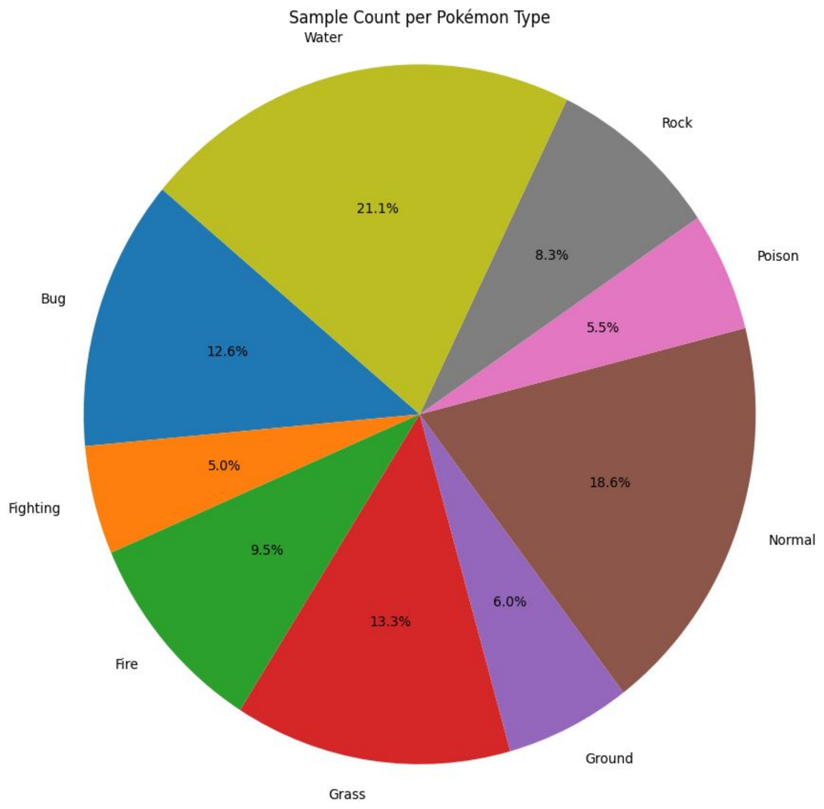
Fine-tuning and Adaptation

Regarding the freezing of layers, our reasoning was as follows: ResNet50's initial layers are known to extract low-level, primitive features from images (for example edges, textures). Since our dataset shares a similar domain with ImageNet (the dataset used to pretrain ResNet50, comprising over 1.2 million images), we chose to retain these early layers exactly as they are, leveraging the pretrained weights without modification. On the other hand, we opted to keep the final two layers unfrozen as these layers are responsible for learning high-level, abstract features that are more task-specific. While these were originally fine-tuned for ImageNet's classification tasks - which differ from ours - the nature of our problem is fairly similar. Therefore, we allow these layers to be updated during training, but with a very low learning rate to make only minimal adjustments. This approach is the one that got us the best results.

Training Split

Given the limited size of the dataset and the noticeable class imbalance, we employed stratified splitting to ensure that each subset maintained a proportional representation of all classes. We divide the data into training (80%), validation (10%), and te (10%) sets

As shown in the pie chart, the dataset contains significant variability in class frequency, ranging fron as low as 5.0% (Fighting) and 5.5% (Poison) to as high as 21.1% (Water) and 18.6% (Normal). Without stratification, random splitting could result in some classes being completely absent from certain subse affecting model evaluation and learning stability.



Data Augmentation

Given the limited size of the dataset, data augmentation was essential to mitigate overfitting and improve generalization. Deep convolutional networks such as ResNet-50 are prone to overfit when trained on small datasets, augmentation introduces controlled variability in the training images, which encourages the model to learn without compromising the semantic identity of the target classes.

In our project, spatial and photometric augmentations were applied, including horizontal flipping, random rotations, and moderate adjustments to brightness, contrast, and saturation. All images were resized to 224x224 to match the input size expected by ResNet.

Although MixUp and CutMix are popular augmentation techniques for improving generalization, they were found to be counterproductive in this context. Both methods are based on the assumption that interpolating images or replacing image regions across classes can produce semantically meaningful hybrids, and that labels can be similarly interpolated. This assumption breaks down in scenarios involving semantically rich and visually distinct categories such as Pokémon types. For example, a hybrid image that mixes a Bulbasaur with a Gengar does not correspond to a coherent “part-Grass, part-Ghost” entity. Instead, it creates a visually confusing input that misleads the model. This confusion is exacerbated by the small size of the dataset. In large datasets, the impact of occasional noisy or nonsensical examples is diluted. In contrast, small datasets amplify the influence of such examples, destabilizing learning and degrading generalization. Training with MixUp or CutMix resulted in slower convergence and lower F1 scores (0.33, locally our best model had 0.49) especially for underrepresented classes, with Fighting and Poison classes having F1 scores of 0 and 0.15, respectively. Consequently, these augmentations were excluded from the final model.

Regularization and Class Imbalance

To address class imbalance, we tried several methods. Focal Loss was initially considered due to its popularity in object detection tasks with severe foreground-background imbalance. It modifies the standard CrossEntropy loss by applying a modulating factor that emphasizes harder-to-classify examples. While theoretically appealing, Focal Loss performed poorly in this context. Class-weighted CrossEntropy was also tested by assigning higher loss weights to rarer classes, using weights calculated from the inverse class frequencies to penalize misclassification of underrepresented classes more heavily.

Instead of manipulating the loss function, a more stable and effective strategy was adopted, the use of WeightedRandomSampler during data loading. This approach also uses the same class-based weights to ensure that underrepresented classes are sampled more frequently during training, effectively balancing the class distribution in each mini-batch.

	Focal Loss	**Cross Entropy (weights)**	**Weighted Sampler**
F1 Score(Total/ Type)	0.4293	0.5268	0.4904
Bug	0.0909	0.2727	0.4348
Fighting	0.2857	0.2222	0.1818
Fire	0.6087	0.6667	0.6087
Grass	0.7500	0.8387	0.7333
Ground	0.5000	0.6667	0.4286
Normal	0.4815	0.4667	0.5538
Poison	0.3636	0.5000	0.4444
Rock	0.2353	0.4545	0.3750
Water	0.5484	0.6531	0.6531
Kaggle Private	0.27065	0.25041	0.29650
Kaggle Public	0.29711	0.26483	0.33726

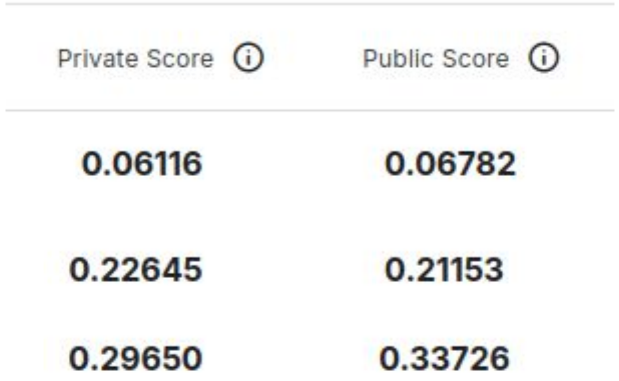
As already, we introduced dropout in the classifier head with a probability of 0.5. Additionally, we employed a ReduceLROnPlateau learning rate scheduler, which dynamically reduced the learning rate when validation loss plateaued. This adaptive adjustment helped the model escape local minima and converge toward flatter, more generalizable solutions.

We also implemented early stopping, but only began monitoring after the 10th epoch to allow the model sufficient time to start learning meaningful patterns before potentially halting.

Finally, we experimented with gradual unfreezing of earlier layers (layers 1 and 2), intending to fine-tune more of the model. However, this approach actually degraded performance, likely due to overfitting or disruption of pretrained feature representations, and was therefore not used in the final setup.

Performance Evaluation and Interpretation

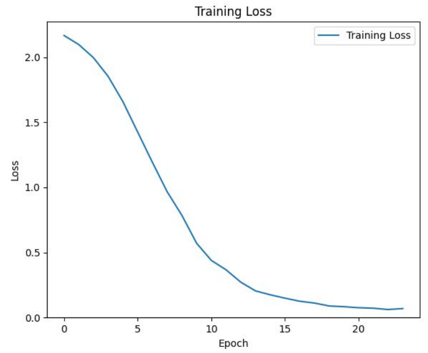
In order to evaluate the performance of our approach, we will look into 3 main results and their corresponding private Kaggle scores. First, we have what we define as the baseline: the ResNet50 model with only its head swapped, untrained. For this case, we swapped the head just for compatibility purposes with the 9 classes. For the second result, we used the previously described transfer learning approach with no data augmentation other than normalization. For our final attempt, we used all the tools we could gather, which includes the data augmentation.

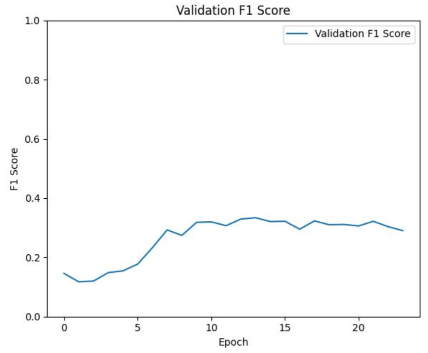


Private Score	Public Score
0.06116	0.06782
0.22645	0.21153
0.29650	0.33726

Performance Evaluation and Interpretation

On the right are represented the graphs for the progress of our training loss and F1 score when not using data augmentation. This approach was not expected to obtain better results than with data augmentation, but we did it anyway to confirm and serve as a base of comparison. By comparing these two graphs we are able to see the effect of our low amount of training data: while our training loss converges smoothly, our F1 score struggles to increase, reflecting how our model overfits on our data, being unable to generalize despite continued training. Thanks to our early stopping we are able to stop fairly quickly, after just about 20 epochs. Despite not being an amazing performance, it is still a huge increase in comparison to the previous approach, highlighting the value of transfer learning in this challenge.

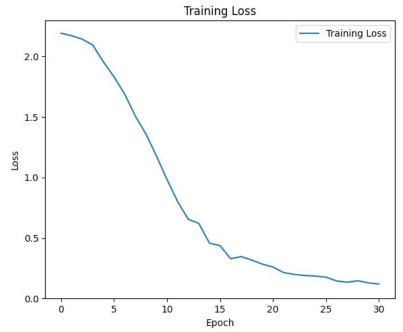


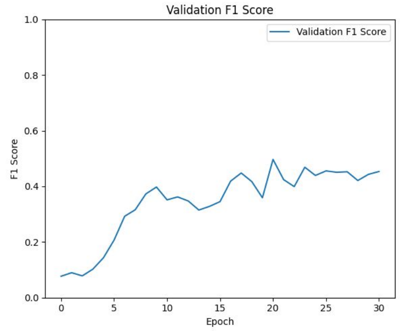


0.22645 0.21153

Performance Evaluation and Interpretation

Finally, on the right we can see the visualizations for the progress of our training loss and F1 score of our best attempt. Here, we are using data augmentation, such as random rotations, horizontal flips, affine transformations and added color jitter. These make it harder for the training loss to converge, since the model doesn't overfit as easily, which is really good and the objective, as we can see in our F1 score graph. This graph shows us that with this technique we don't reach a plateau as easily, instead going up and down more erratically, but showing an overall constant improvement. Here we are also able to observe why the early stopping doesn't activate as fast, but doing so after a mere 30 epochs, making it a fairly quick and cheap training process.





0.29650 0.33726

Insights Gained

By developing our model we learned how to make use of meaningful feature learning despite having scarce data. And we learned how to achieve stability in the presence of class imbalances.

What went wrong

We found it hard to find relevant free information to improve our model’s performance with Transfer Learning on small datasets with class imbalance.

What went great

We were able to achieve good overall results despite dealing with data scarcity, which mimics real world constraints, showing our solution is scalable and realistic.

We were also able to make effective use of transfer learning and training process.