Pokémon Image Classification Challenge

Task 3

Team LLMH

Basics of Task 3

- The team
  - GALLO Lorenzo 72719
  - LEKBOURI Lina 72697
  - LICHTNER Marc 72690
  - WERCK Hugo 72692
- The work
  - Best public score: 0.17175
  - Best private score: 0.19786
  - Leaderboard position in the private leaderboard: 15/18

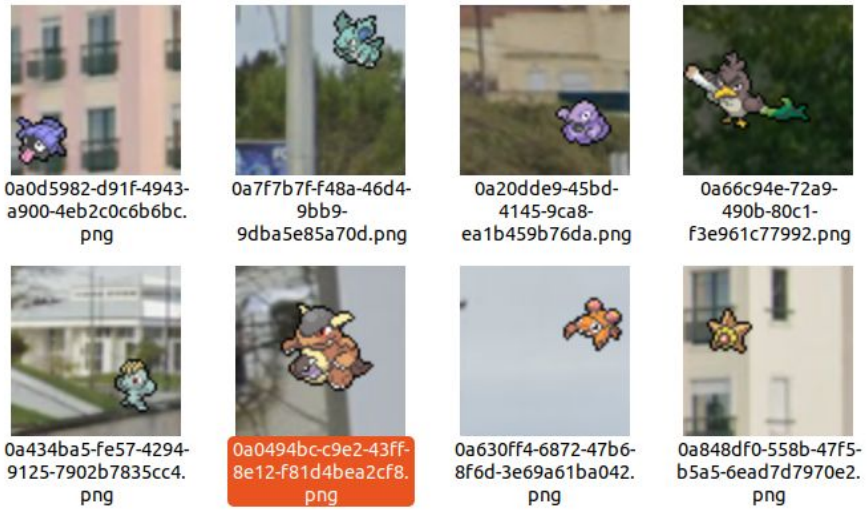
Task 1: Multilayer Perceptron (MLP) Classification

Feedbacks and summaries

Data Exploration: Summary

- **Dataset look:**

Train and Test images folder



train_labels.csv

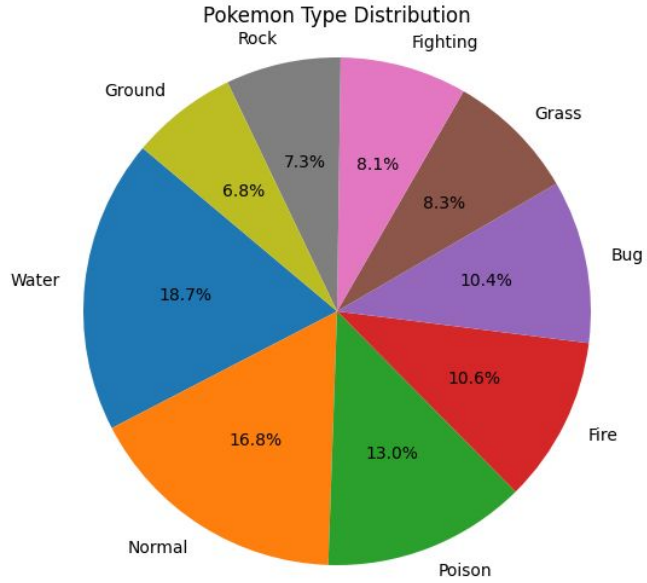
	Id	label
0	6fc9045b-9983-41e2-be2d-8796ecd97412	Normal
1	874716ce-9048-4e8a-b980-5ed9a5c0110e	Poison
2	c3613b20-ead8-48e1-8c8d-2f219d8e19d4	Normal
3	c7264ebc-ba44-460a-9b2b-df23c04783bc	Normal
4	a72045db-8fae-458b-993e-23d2aab1a5c6	Normal

Data Exploration: Summary

- **Global information and class distribution:**

- 3600 in the Train folder
- 9 labels
- 0 duplicates
- size of each images: 64*64
- background: nothing to do with the pokémon

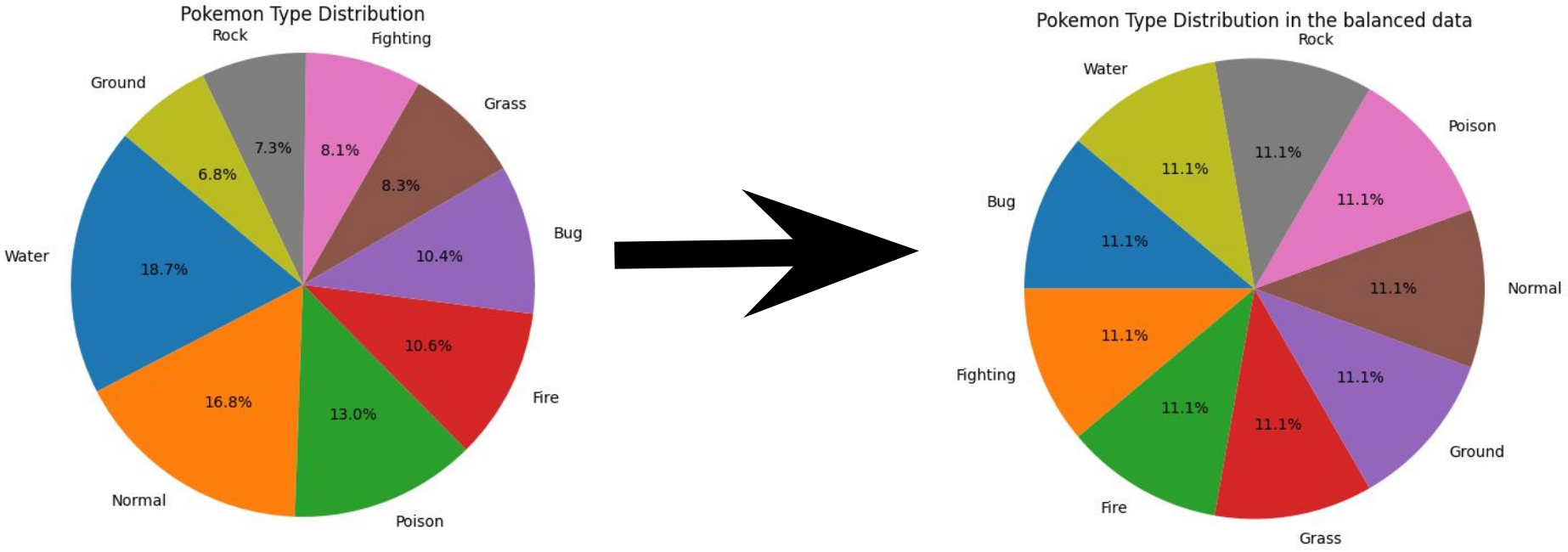
class distribution



⇒ **imbalance**

Data Preprocessing: Summary

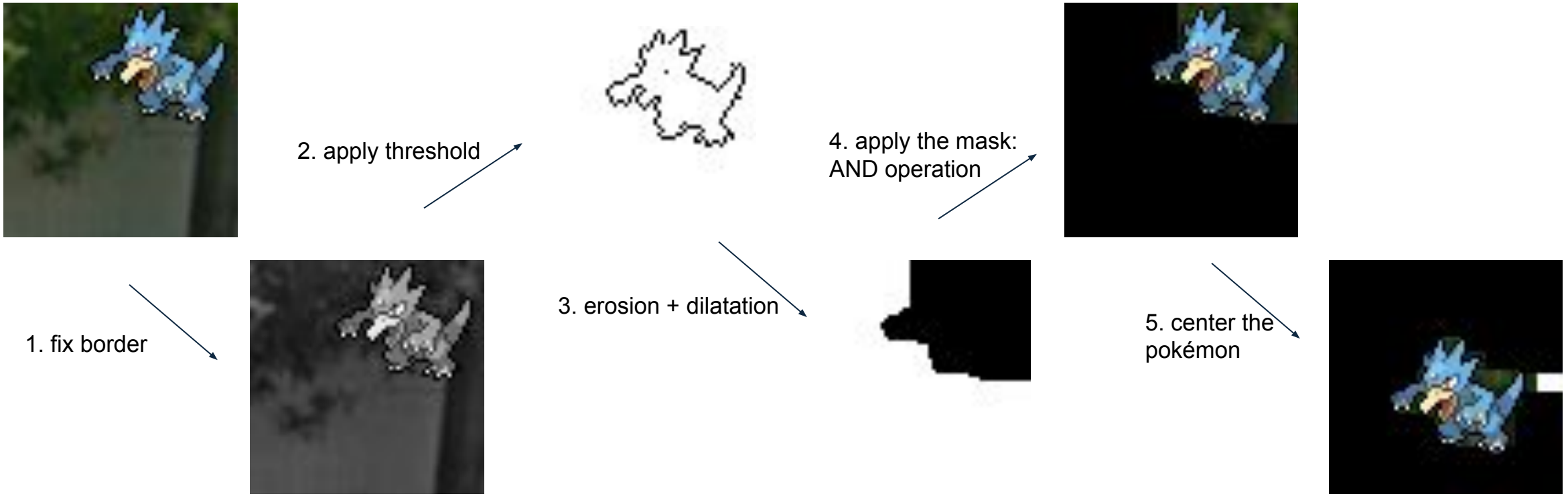
- **Balancing the data:** undersampling



Data Preprocessing: Summary

- **Removing the background + centering the pokémon**

  ⇒ to help the model to focus more on the pokémon itself



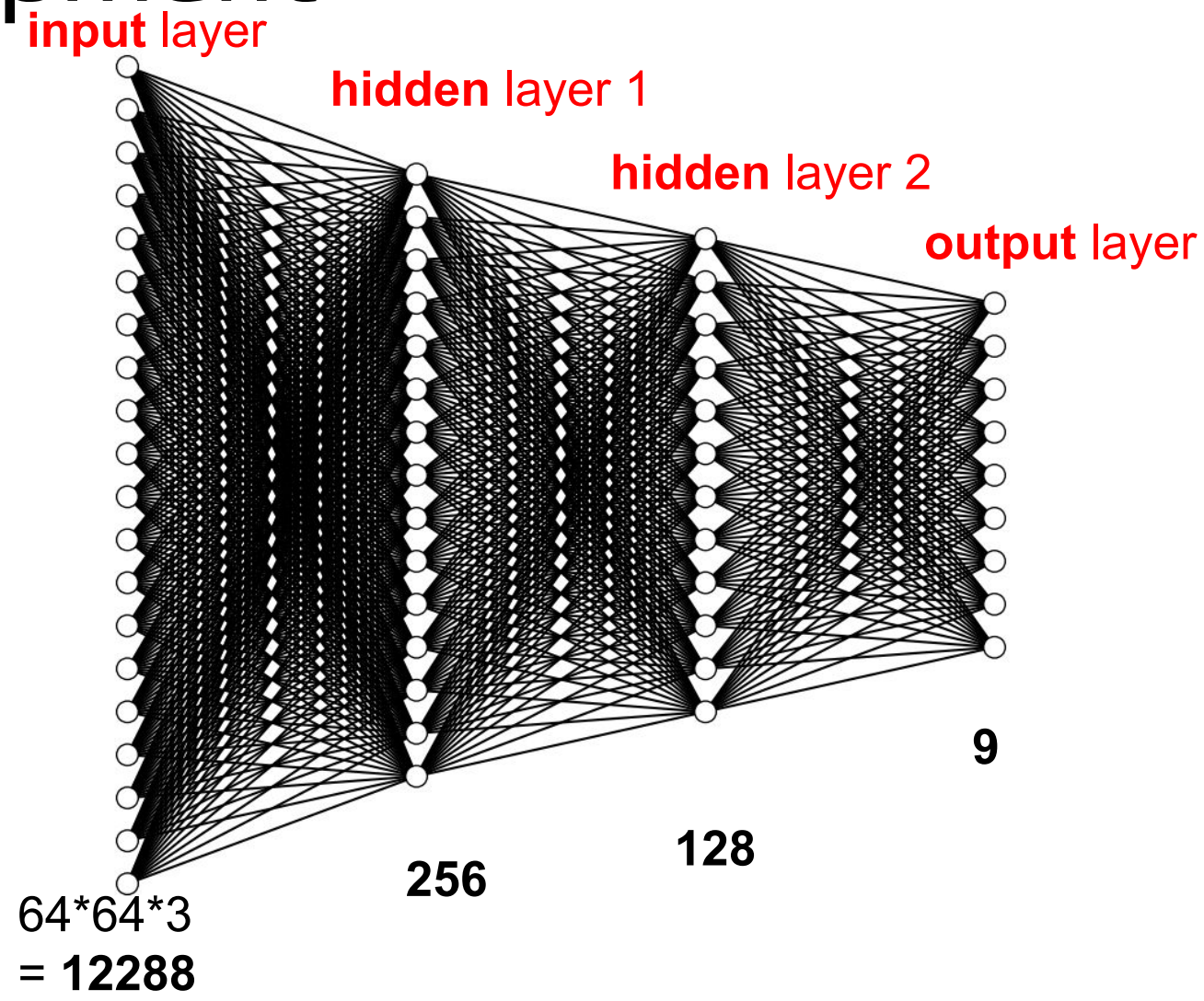
Model Development

- MLP architecture diagram
- Justification for chosen architecture
- Ablation study

Model Development

- **MLP architecture diagram:**

(Given the big numbers of neurons, we can not represent the real numbers on this slide, so we take random numbers of neurons respecting orders of magnitude.)



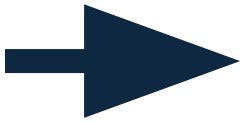
number of neurons:

Model Development

- **Justification for chosen architecture**

4 layers:

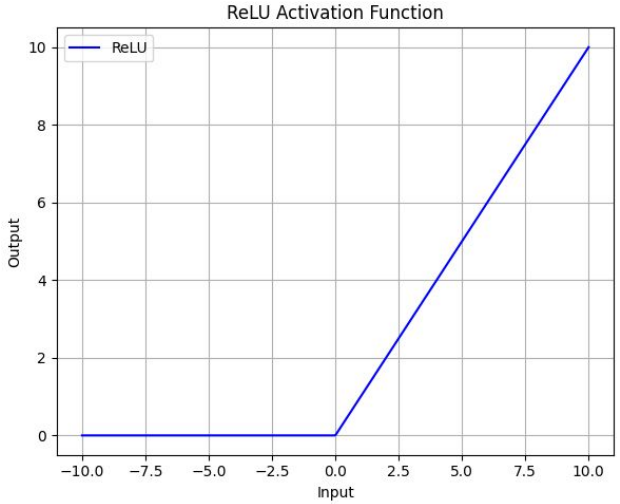
- input layer: **64*64*3 = 12 288 neurons** (number of pixels in each image by the RGB channel);
- hidden layer 1: 256 neurons;
- hidden layer 2: 128 neurons;
- output layer: **9 neurons** (number of primary types of pokémon).



The hidden architecture giving the best result after testing.

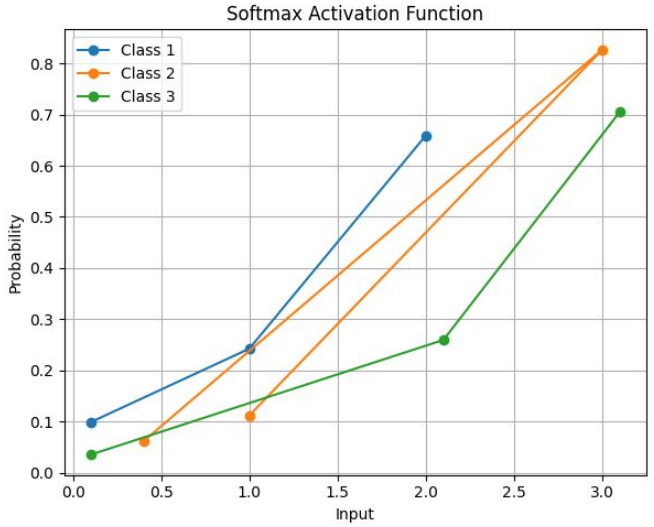
Model Development

- **Justification for activations**



Activation functions:

- **ReLU** (from input to h1 and from h1 to h2): efficient and improve generalization (avoid gradient vanishing).
- **Softmax** (from h2 to output): Converts raw scores into probabilities, making the model interpretable.



Model Development

- **Ablation study conclusions:**
  - ReLU significantly improves **training efficiency and performance**;
  - Softmax is essential for **classification tasks**;
  - Deeper architectures capture **more complex representations**.

Training Efficiency

- Training time
- GPU usage
- Strategies employed for efficient training.

Training Efficiency

Training time

As we converged rapidly to a maximum. We were able to complete the training within a **couple of minutes**. (local run)

Using the GPU and using early stopping, the training was completed within **24 seconds**.

Training Efficiency

GPU usage:

By using the **GPU**, it took **1min59** to compute 200 epochs whereas it took **4min54** with the **CPU**. This is what we used to take advantage of the **GPU if available**:

```
[ ]  # Check if GPU is available
     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     print(f"Using device: {device}")
```

Training Efficiency

- Strategies employed for **efficient training**:

  - **Early stopping**: Avoid useless computations by stopping the training when the model don’t improve anymore

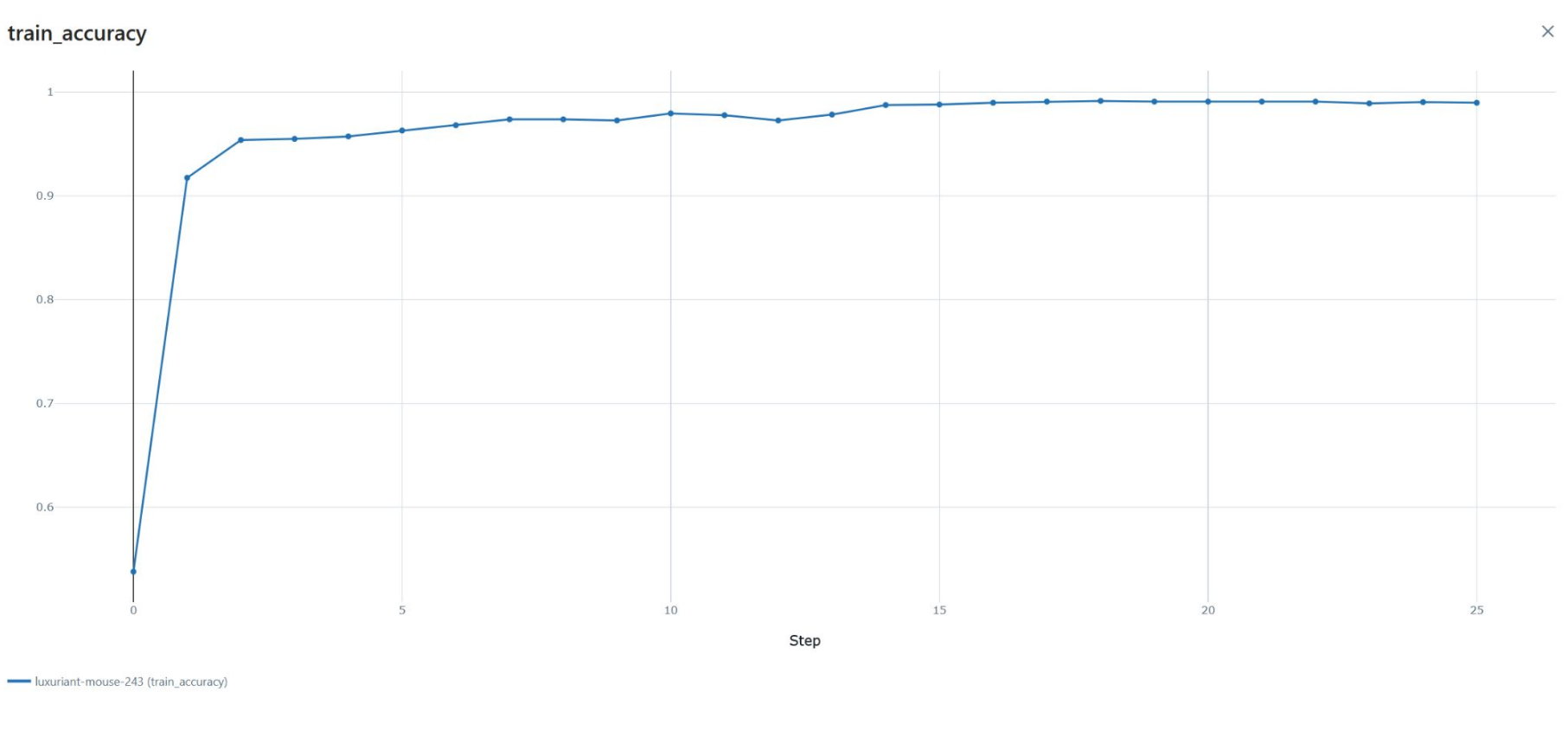
  - **Batch size**: In order to select the batch sizes, we followed the “linear scaling rule”. We chose a batch size of 32 and a learning rate of 0.001.

Performance Evaluation

- Justification of the metrics used
- Interpretation of results

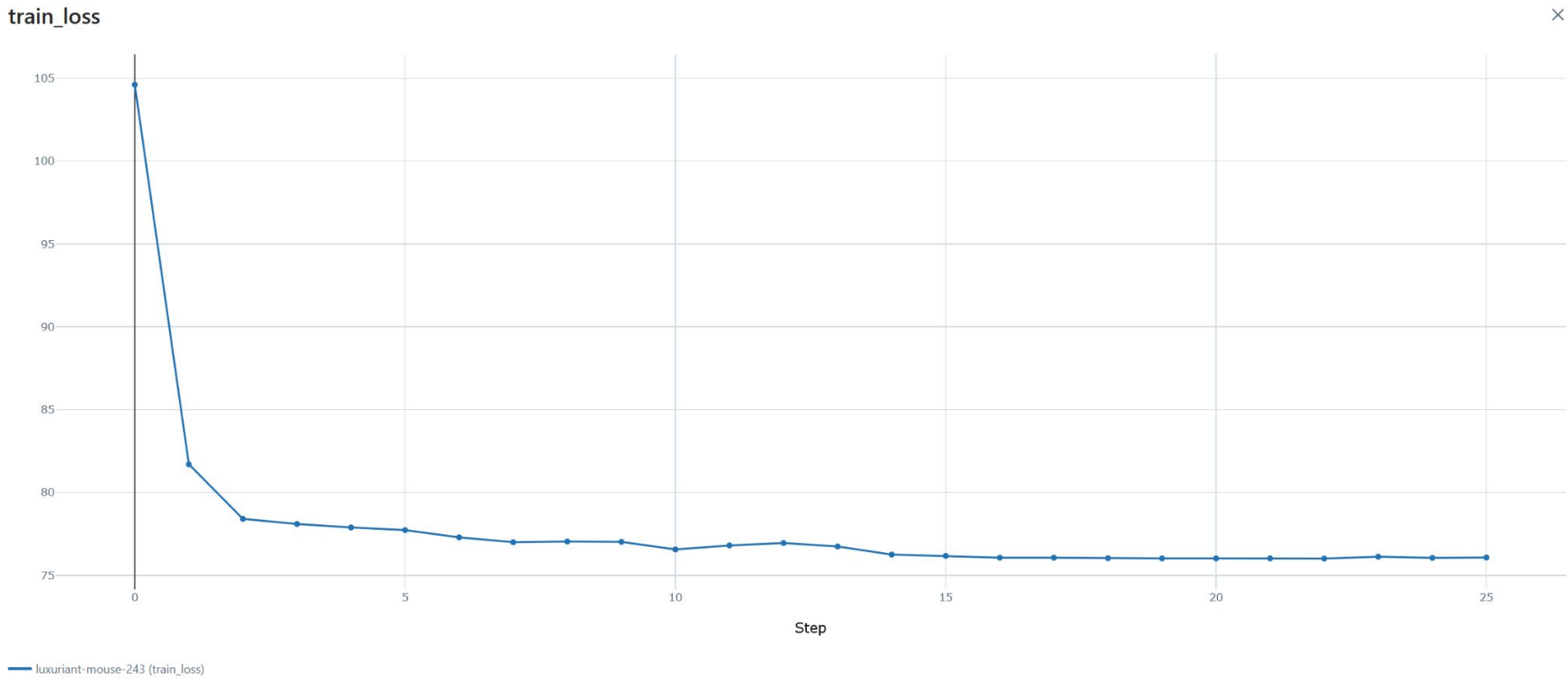
Performance Evaluation

- **Train accuracy**
  - Measures the percentage of correctly classified training samples, indicating how well the model fits the training data.
  - The curve shows an increasing trend, reaching over **95%** (against 85% without centering). This suggests the model is learning very effectively from the training data, images being centered.



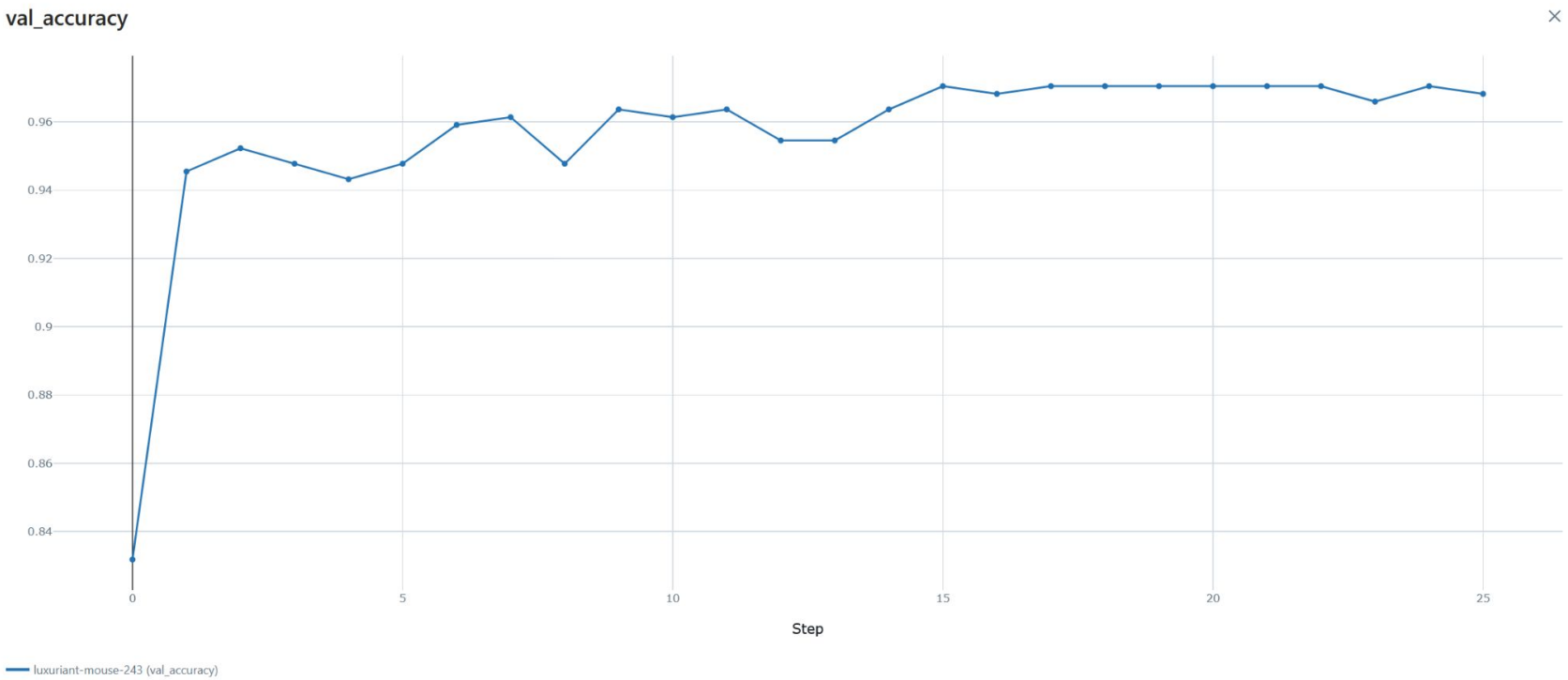
Performance Evaluation

- **Train loss**
  - Represents the error during training, helping to track convergence and detect overfitting.
  - The loss decreases rapidly at **76%** (against 82% without centering), indicating successful learning.



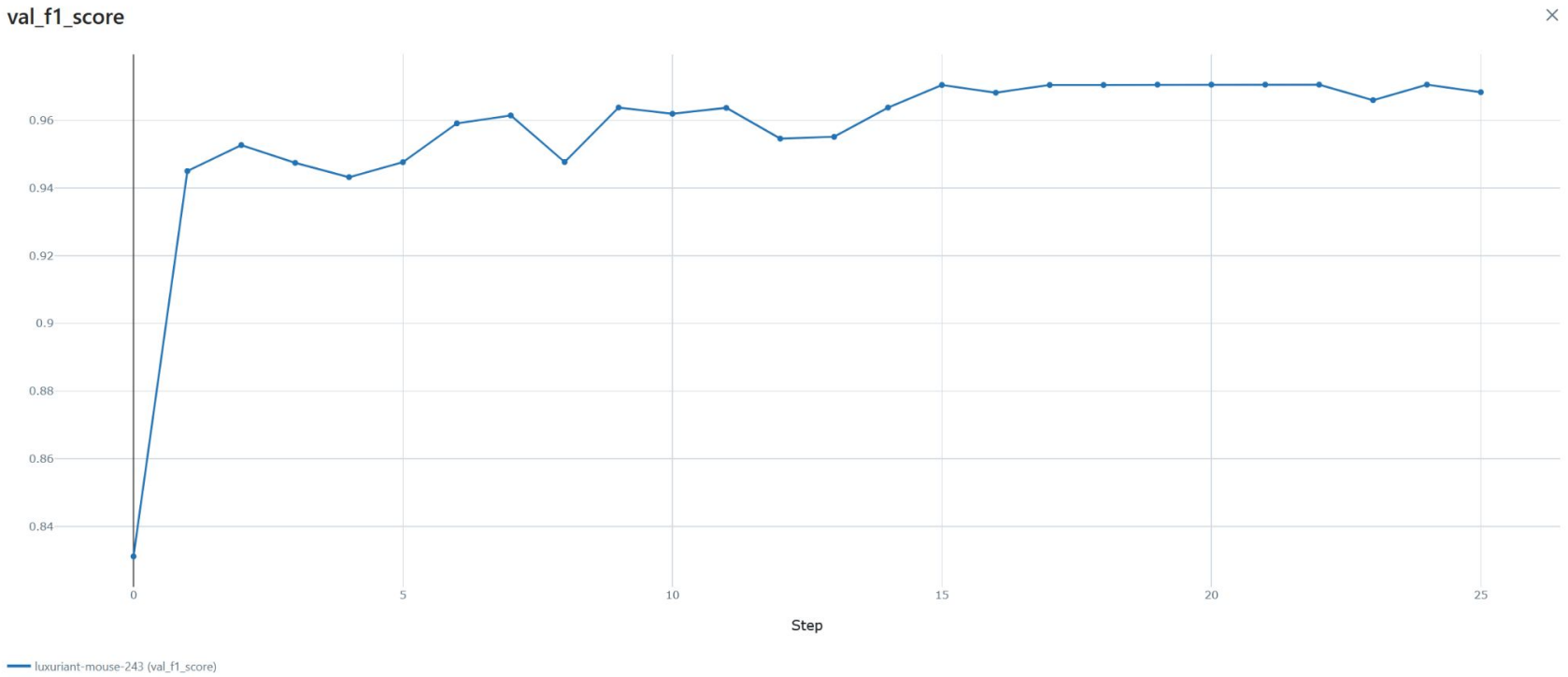
Performance Evaluation

- **Validation accuracy**
  - Evaluates performance on unseen data, reflecting generalization ability.
  - The curve fluctuates around **96%** (against 30% without centering !) which suggests the model generalize well with that improvement.



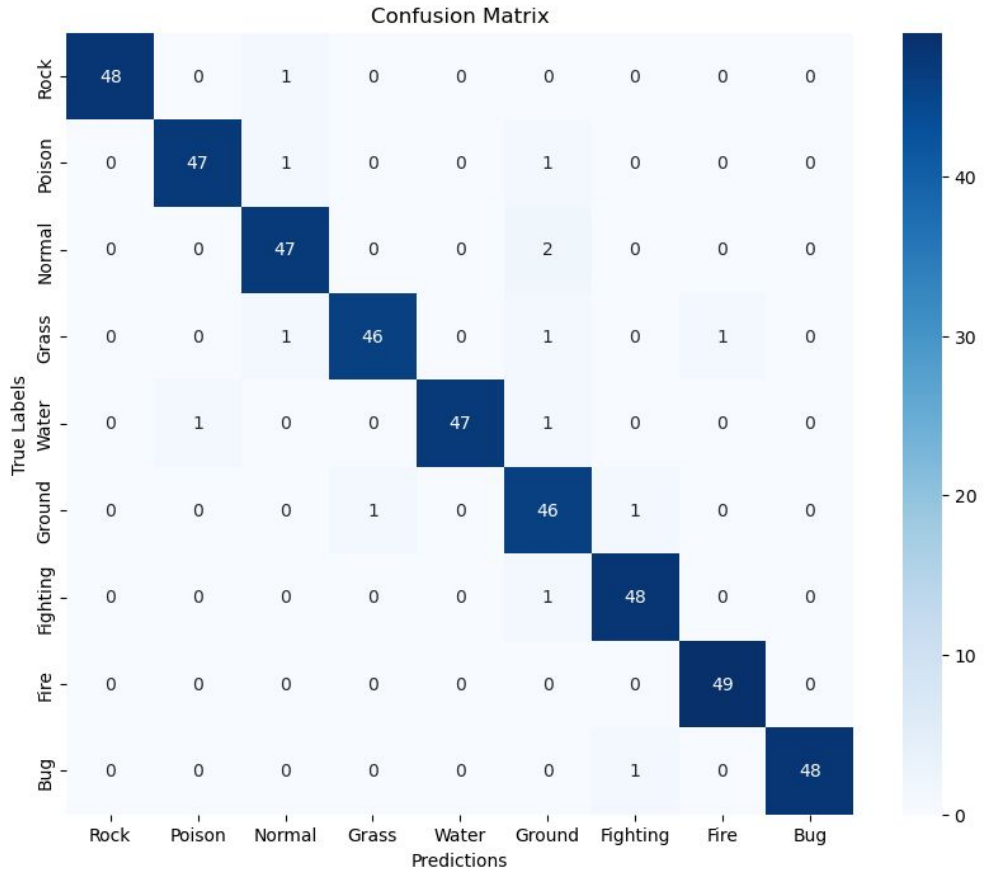
Performance Evaluation

- **Macro F1-score**
  - Balances precision and recall, crucial for handling class imbalances.
  - The high variance suggests unstable predictions across classes.
  - **96%** now with less variance than before, and f1 at 30% without centering.



Performance Evaluation

- **Confusion matrix**
  - Provides a detailed evaluation of the model's performance beyond a single accuracy score.
  - Strong performance on every class, only few mistakes.



Task 2: Convolutional Neural Network (CNN) Development

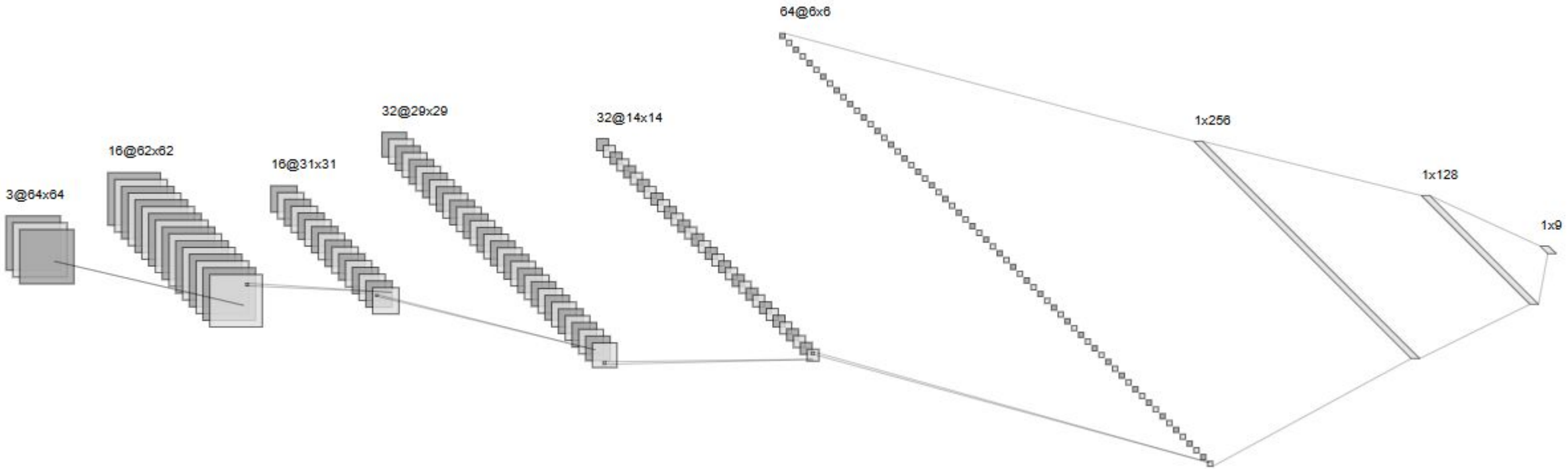
Feedbacks and summaries

CNN Model Development

- CNN architecture diagram
- Explanation and justification for design choices using quantitative arguments.
- Ablation study

CNN Model Development

CNN Architecture diagram



CNN Model Development

• Justification for chosen architecture

6 layers (feature extraction):

- **Input layer:** 64×64 RGB image →**3 channels**, 64×64 pixels = **3@64×64**
  → the pokémon images are colored (RGB) 64*64 sized images.
- **Conv Layer 1 + ReLU:** 16 filters, 3×3 kernel→ output: **16@62×62**
  → Extracts low-level features such as edges, corners, and textures, using the ReLU.
- **Max Pool 1:** 2×2→ output: **16@31×31**
  →Reduces the spatial dimensions (by half) while preserving the most important features.
- **Conv Layer 2 + ReLU:** 32 filters, 3×3 → output: **32@29×29**
  →Extracts more abstract and complex features such as shapes and textures by learning from the low-level features provided by the first convolutional layer.
- **Max Pool 2:** 2×2 → output: **32@14×14**
  →Further reduces spatial dimensions to retain only the most important features and reduce the risk of overfitting.
- **Conv Layer 3 + ReLU:** 64 filters, 3×3 → output: **64@12×12**
  →Extracts high-level and more abstract features, such as object parts or larger shapes.
- **Max Pool 3:** 2×2→ output: **64@6×6**
  →Final reduction of spatial dimensions to prepare for the classification layers, it ensures that the most critical features are retained while reducing computation for the classifier.

CNN Model Development

• Justification for chosen architecture

Classifier layers (flatten + fully connected):

- **Flatten Layer**: 64×6×6 = 2,304 neurons
  →Converts the feature map from the last convolutional layer ((64 \times 6 \times 6)) into a 1D vector of size (2,304) neurons.
- **Fully connected layer 1**: 2,304 → 512 neurons *(with ReLU and Dropout)*
  →Reduces the dimensionality of the feature vector while preserving important information. (The **ReLU activation** enables the network to learn complex patterns in the data and the **Dropout layer** helps prevent overfitting by randomly deactivating neurons during training, improving generalization.)
- **Fully connected layer 2 (output)**: 512 → 9 neurons *(number of Pokémon types)*
  → Produces the final class scores, one for each Pokémon type, using the softmax activation function.

CNN Model Development

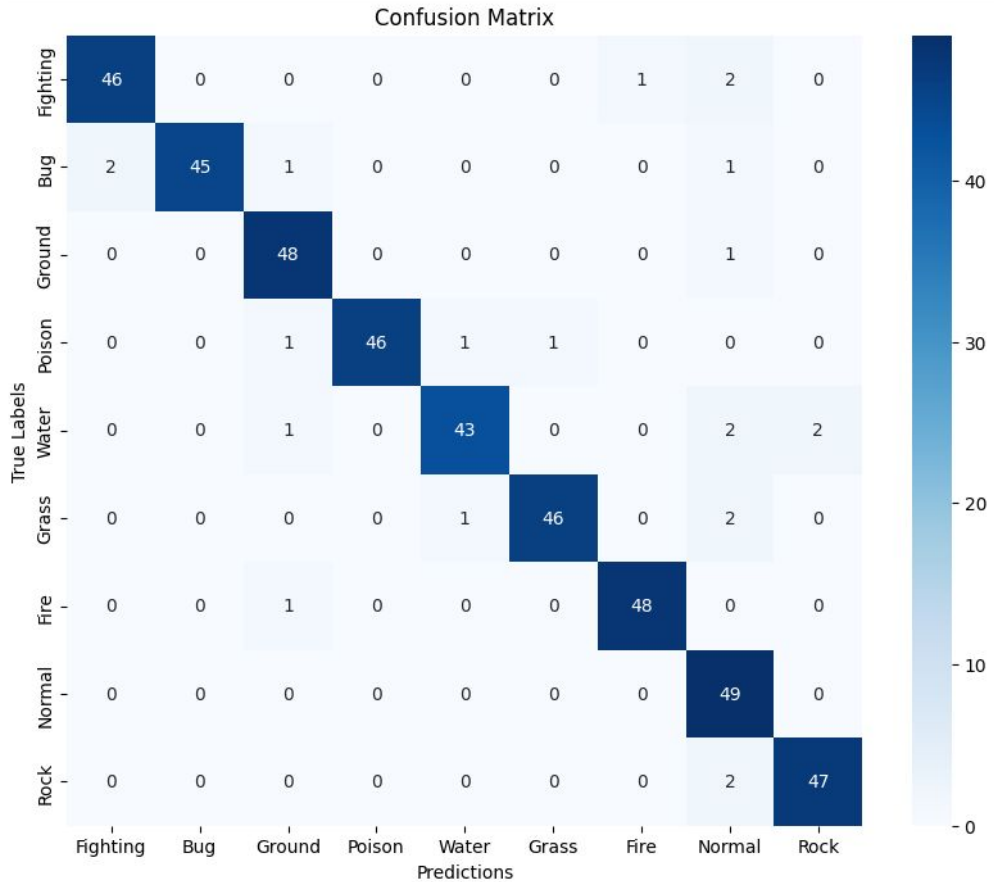
To justify our choice, we tried to use **different architectures** of CNN Model, but none of them give us a better result than the one with the architecture we choose.

CNN Model Development

- Trying out a different architecture:
  LeNet architecture:

1. **Input:** images of size 64x64.
2. **Convolutional Layers:**
   - **Conv1:** 6 filters of size 5x5, output 6x28x28, activation **ReLU**.
   - **Pooling1:** Sub-sampling (pooling) 2x2, output 6x14x14.
   - **Conv2:** 16 filters of size 5x5, output 16x10x10, activation **ReLU**.
   - **Pooling2:** Sub-sampling 2x2, output 16x5x5.
3. **Fully Connected Layers:**
   - **FC1:** 400 (16x5x5) → 120 neurons, activation **ReLU**.
   - **FC2:** 120 → 84 neurons, activation **ReLU**.
   - **FC3 (Output):** 84 → Number of classes
4. **Output:**
   - Class probabilities through **Softmax** activation.

F1-score obtained: 0.9548

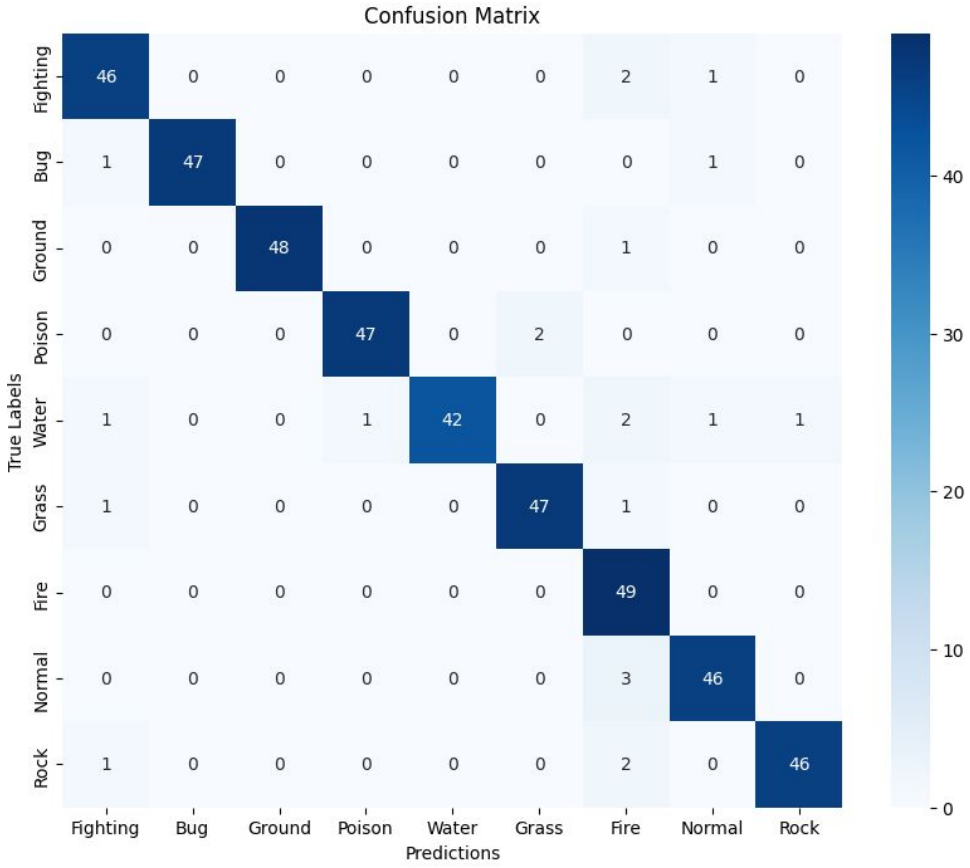


CNN Model Development

- Trying out a different architecture:
  Enhanced LeNet architecture

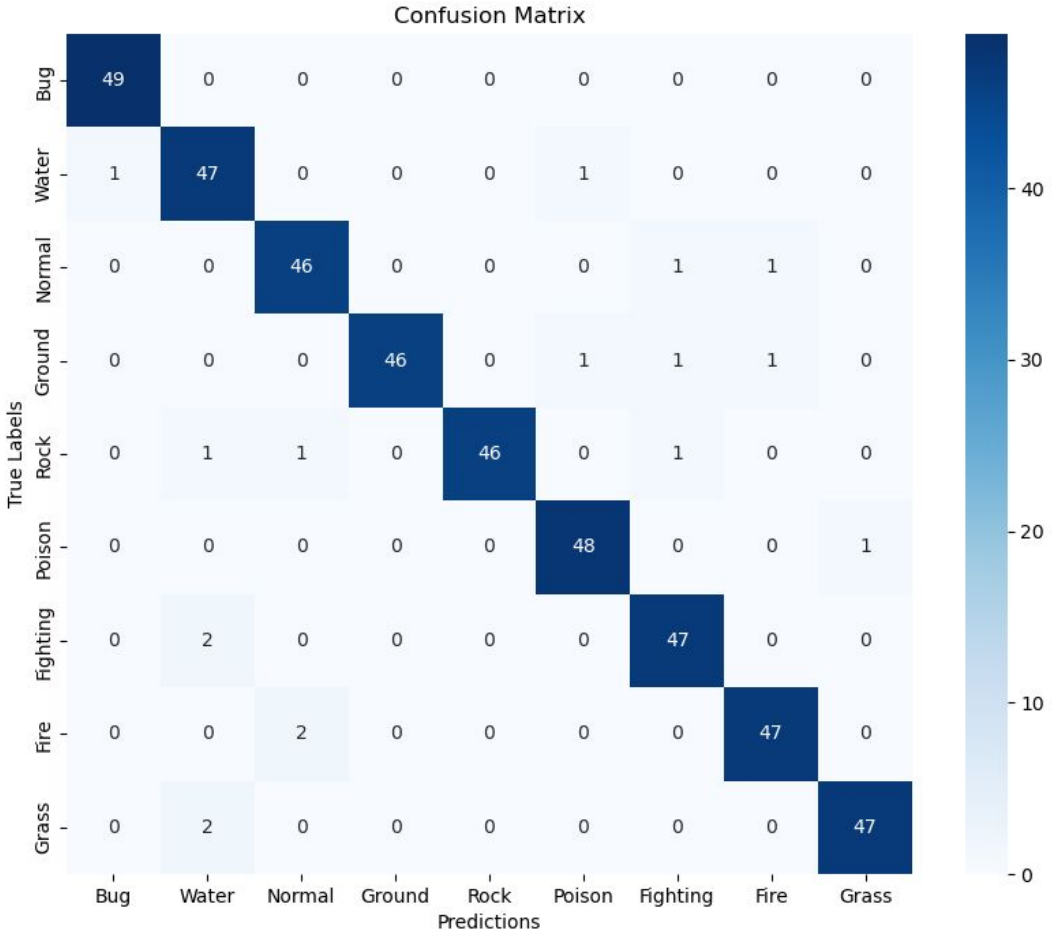
1. **Input:** Images with three color channels (RGB) of size 64 \times 64.
2. **Feature Extraction:**
   - **Conv1:** 3x3 kernel → Output: 16x62x62.
     Followed by **max pooling** → 16x31x31.
   - **Conv2:** 3x3 kernel → Output: 32x29x29.
     Followed by **max pooling** → 32x14x14.
   - **Conv3:** 3x3) kernel → Output: 64x12x12.
     Followed by **max pooling** → 64x6x6).
3. **Classification Head:** Fully connected layers for classification:
   - **Flatten:** Converts the feature map 64x6x6 into a flat vector of size 2304.
   - **FC1:** Fully connected layer with 2304→512 neurons. Activation: **ReLU**.
   - **Dropout:** Dropout with a probability of 0.5 for regularization.
   - **FC2:** Fully connected layer with 512→num_classes.
4. **Output:** The final layer outputs logits for each class, which can be converted to probabilities, using **softmax**.

F1-score obtained: 0.9507

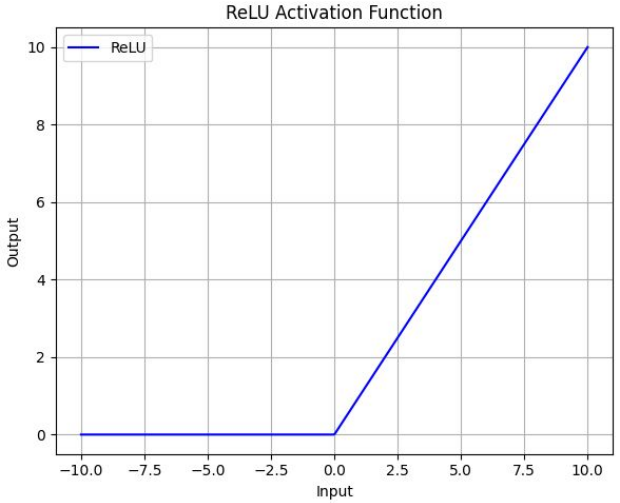


CNN Model Development

Result with the model we choose:



CNN Model Development



- **Justification for activations**

ReLU (Rectified Linear Unit) is used after each convolutional and fully connected layer because It introduces non-linearity, helping the network to learn complex patterns.

ReLU is:

- Computationally **efficient;**
- Helps prevent **vanishing gradients;**
- Activates only **positive values** → creates sparse representations.

No activation in output layer, because we use the **CrossEntropyLoss** → This function **internally applies Softmax**, which transforms logits into class probabilities, the most important part for **classification**.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

CNN Model Development

- **Ablation study:**

Key Experiments:

- **Remove Conv Layer 3:** Test the effect of fewer convolutional layers, analyze the importance of deeper convolutional layers in extracting high-level features.
- **Reduce Filters:** Analyze the impact of fewer filters in each layer.
- **Remove Max Pooling:** Replace pooling with strided convolutions.
- **Remove Dropout:** Evaluate the risk of **overfitting**.
- **Skip FC1:** Directly connect Flatten to output layer.
- **Change Activations:** Replace ReLU with LeakyReLU or ELU.

Every component contributes to **generalization and performance**.

The **full model with 3 conv layers, ReLU, dropout, and 2 FC layers** gave the **best results** on validation accuracy and confusion matrix.

Training Efficiency

- Training time
- GPU usage (if applicable)
- Techniques to address class imbalance and overfitting

Training Efficiency

Training time

As we converged rapidly to a maximum. We were able to complete the training within **2m13**. (local run)

Using the GPU and using early stopping, the training was completed within **19s**.

Performance comparison (CNN vs MLP)

- Side-by-side performance metrics, with tables and plots
- Analysis of improvements and remaining challenges

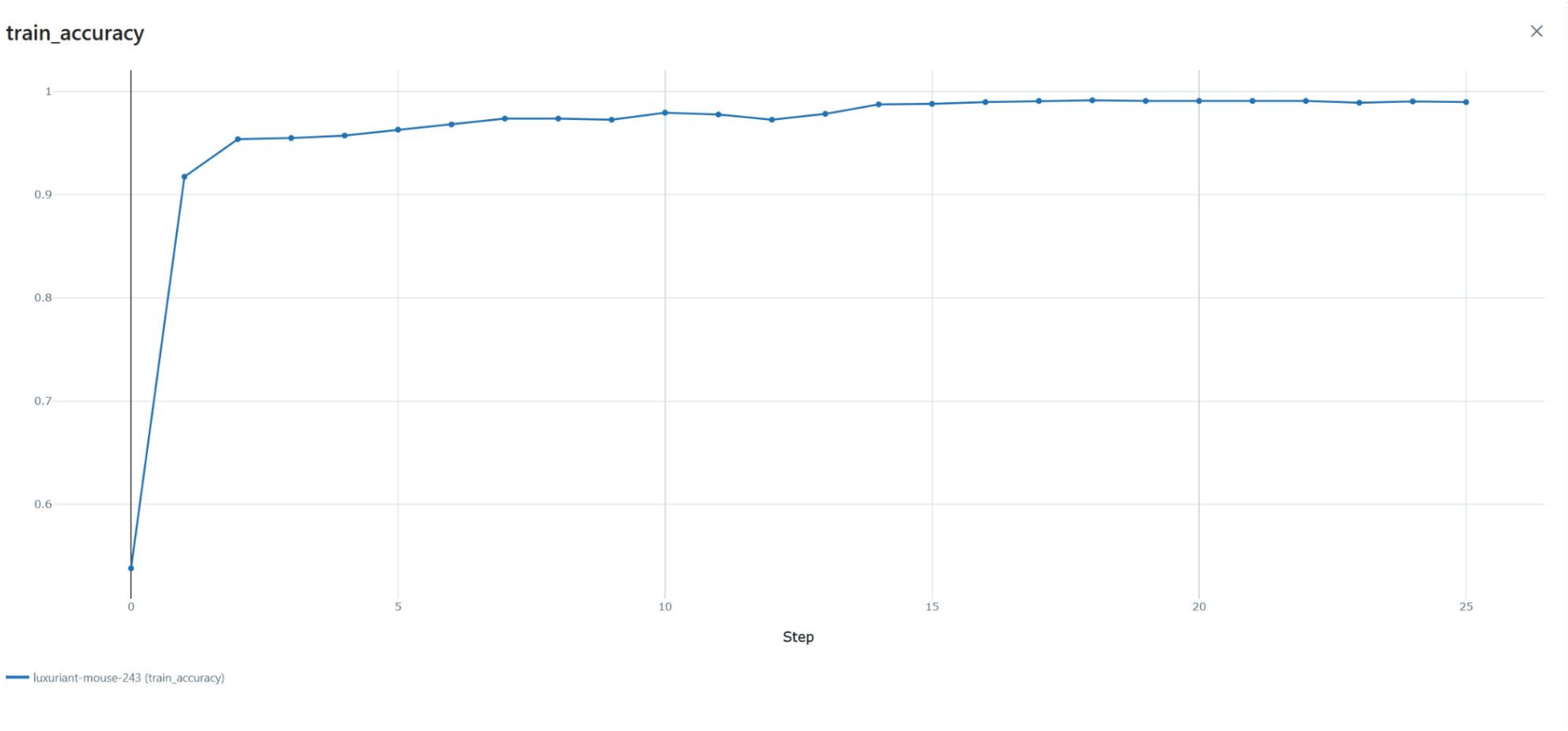
Performance comparison (CNN vs MLP)

Training accuracy :

- The MLP achieves a **high validation accuracy**, demonstrating its ability to generalize well to unseen data when benefiting from effective preprocessing techniques. This indicates that the simplifications introduced during **preprocessing** help the MLP perform competitively on validation datasets.
- The CNN shows a **slightly higher validation** accuracy compared to the MLP. This suggests that the CNN's architecture, designed to capture spatial hierarchies in image data, provides a marginal advantage in generalization, even when preprocessing benefits the MLP significantly.

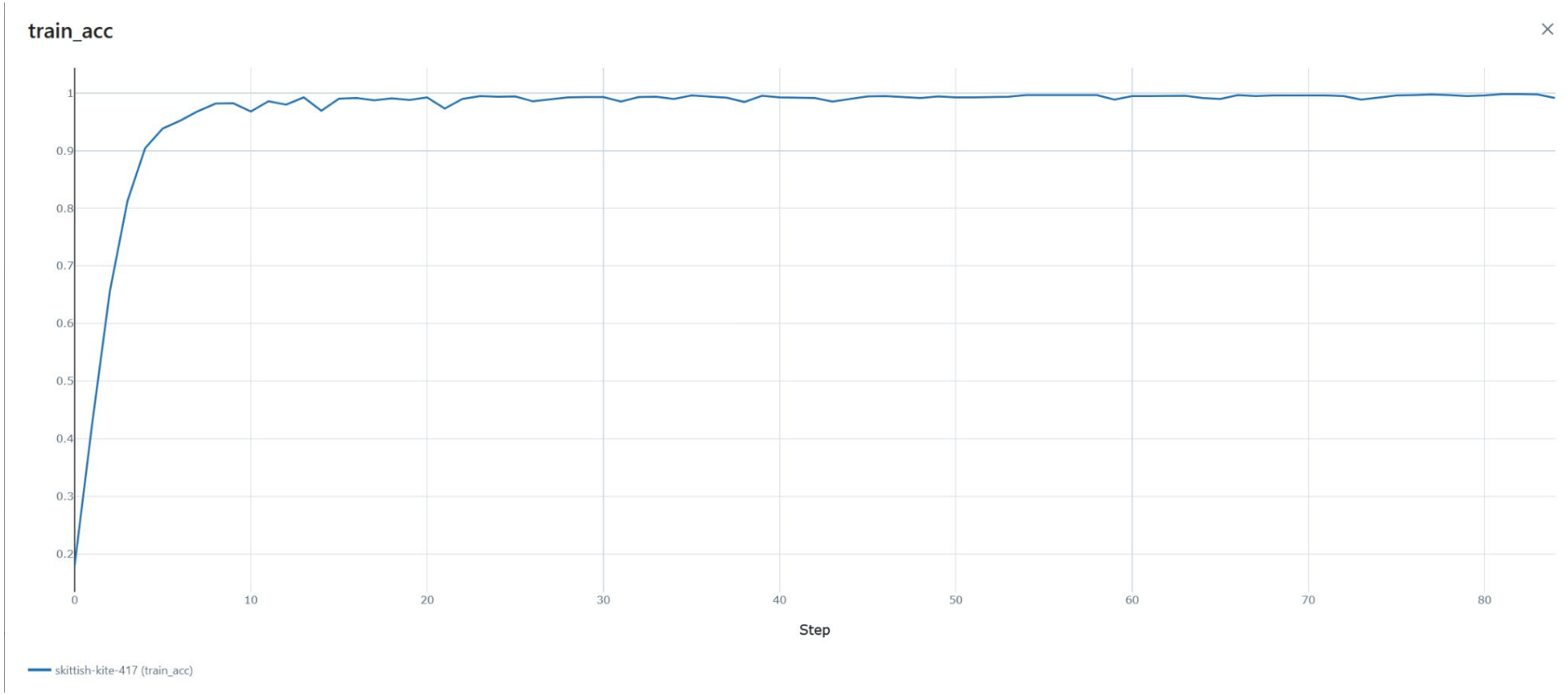
Performance comparison (CNN vs MLP)

Training accuracy of the **MLP**:



Performance comparison (CNN vs MLP)

Training accuracy of the **CNN**:



Performance comparison (CNN vs MLP)

Training Loss:

- The MLP starts with a **high training loss** but rapidly decreases to **stabilize** around 76%. This indicates that while the MLP can reduce errors significantly, it may still have difficulty minimizing the loss further, potentially due to its simpler architecture or overfitting tendencies.
- In contrast, the CNN demonstrates a **sharp decrease in training loss**, stabilizing at a much lower value of around 1%. This suggests that the CNN is more effective at minimizing errors on the training data, benefiting from its ability to learn complex patterns and features inherent in image data.

Performance comparison (CNN vs MLP)

F1 Score:

- The MLP achieves a **high F1 score**, indicating a good balance between precision and recall on the validation set. This suggests that the MLP, with **effective preprocessing**, can effectively classify instances, maintaining a strong performance in terms of both false positives and false negatives.
- The CNN also demonstrates a **high F1 score**, slightly outperforming the MLP. This reflects the CNN's ability to capture complex patterns in the data, leading to better precision and recall. The CNN's architecture allows it to **generalize** well and **maintain high performance metrics** across different evaluation criteria.

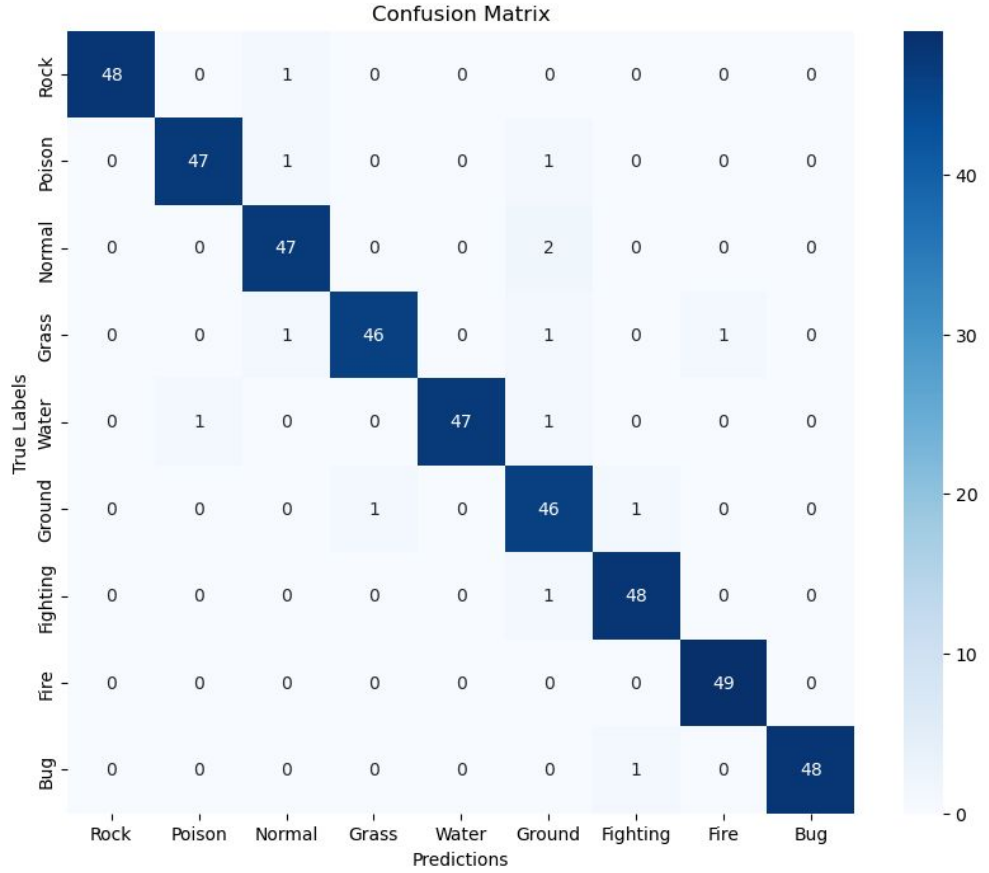
Performance comparison (CNN vs MLP)

Confusion matrix:

- The CNN's confusion matrix displays a **prominent diagonal**, reflecting **accurate predictions** across most classes. The MLP demonstrates slightly better performance in terms of correct classifications, as evidenced by the higher sum of the diagonal in its confusion matrix (427 vs 423).
- Despite the CNN's architectural advantages in capturing spatial hierarchies, the MLP's simpler architecture, combined with effective preprocessing, allows it to achieve better overall classification accuracy in this case.
- Both models face challenges in distinguishing between similar classes, but the MLP appears to handle these distinctions more effectively in this dataset.

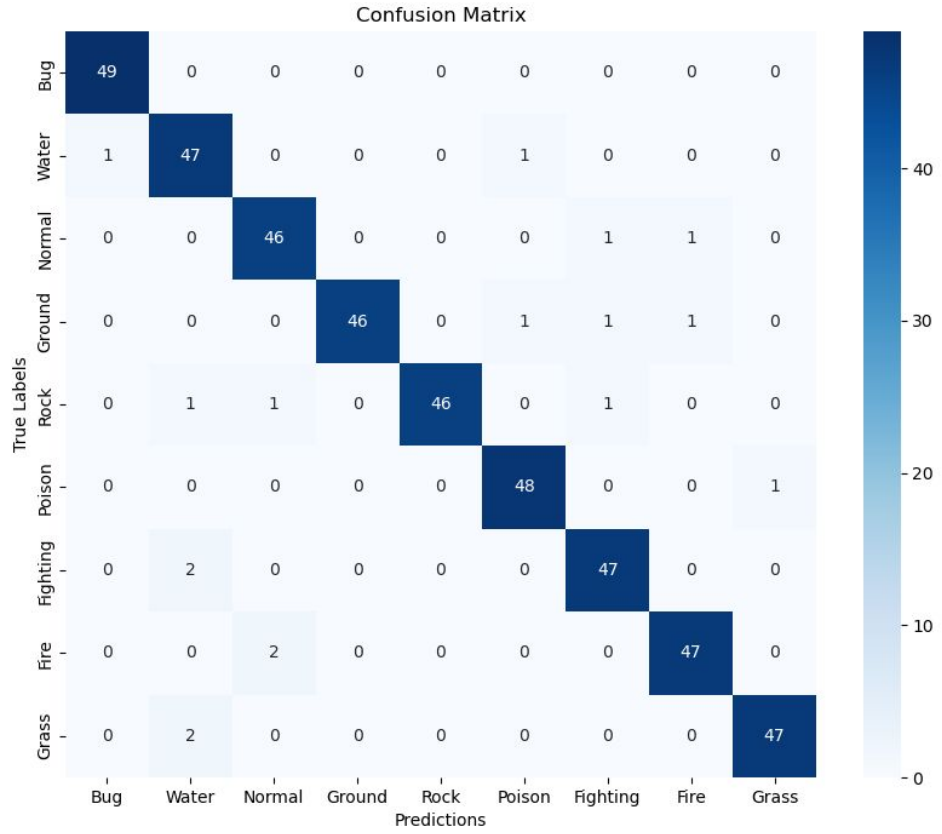
Performance comparison (CNN vs MLP)

Confusion matrix of the **MLP**:



Performance comparison (CNN vs MLP)

Confusion matrix of the **CNN**:



Task 3: Transfer Learning and Fine-Tuning

Note: Pokémon have a new environment

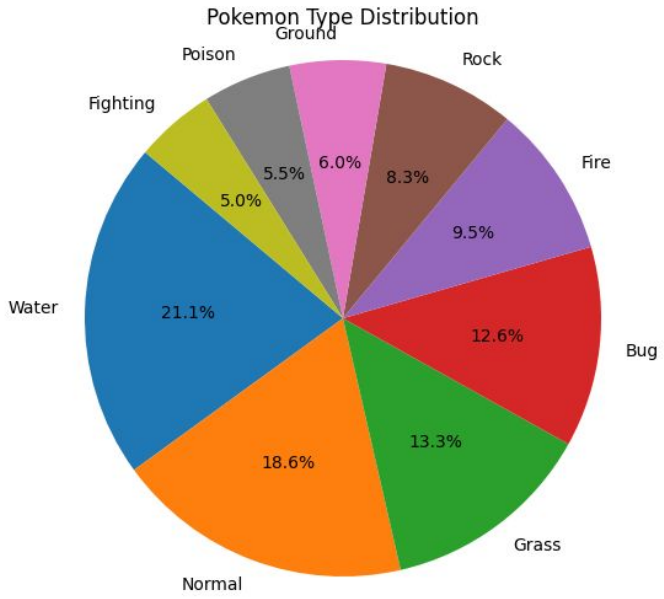
For this final task, we are using a more complex dataset, since we can't easily split the pokémon from its background as depicted here. So, we can't use anymore our strategy to remove the background and center the pokémon.

Primary type: Poison

Primary type: Grass

Primary type: Normal

Moreover, we are still dealing with unbalanced data, as depicted by this pie chart.

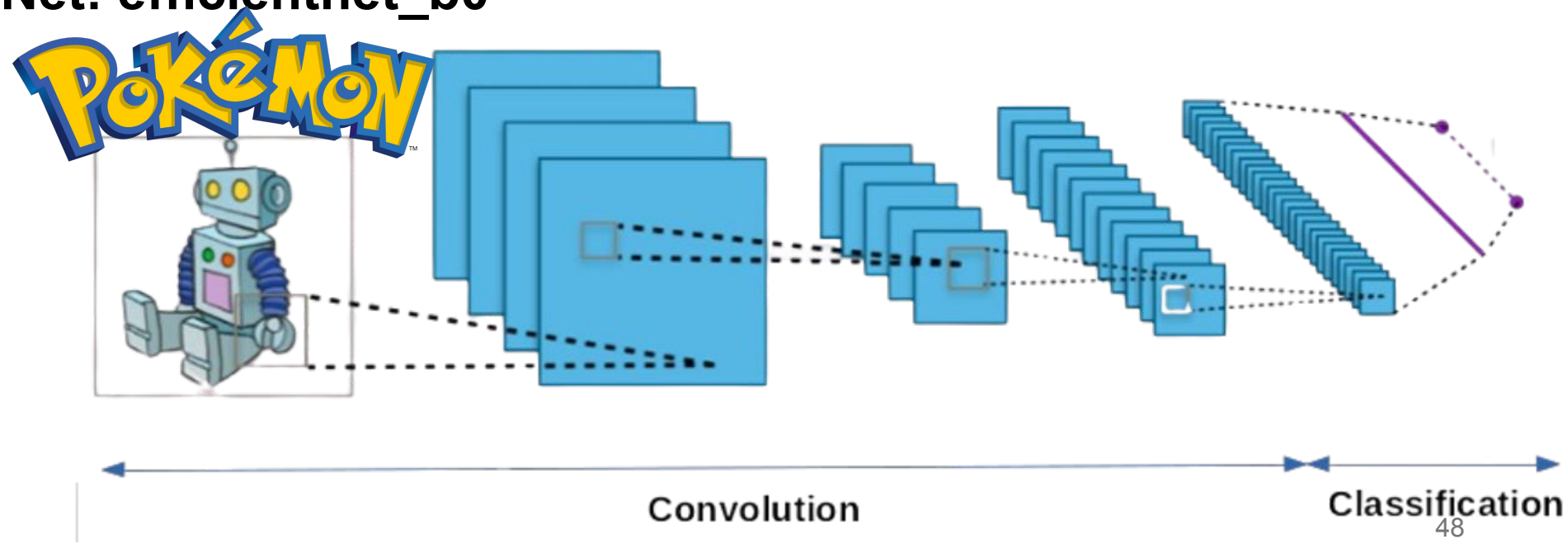


Transfer Learning Pipeline Development

- Models tested
- Details on the MLP in front of the feature extractor
- Explanation and justification for design choices using quantitative arguments.
- Ablation study

Transfer Learning Pipeline Development

- Models tested:
  - **Resnet: resnet50**
  - **VGG: vgg16**
  - **EfficientNet: efficientnet_b0**



Transfer Learning Pipeline Development

- Details on the **MLP** in front of the feature extractor

  We used two different MLPs during our test, and one happened to give better result than the other.

→a simple Linear layer

```
model.classifier[1] = nn.Linear(model.classifier[1].in_features, num_classes)
```

→a more complexe layer with ReLU and Dropout

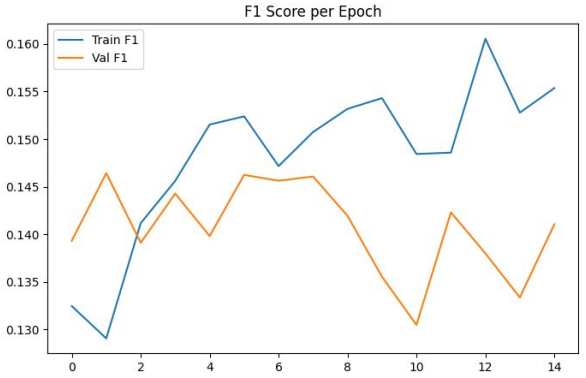
```
model.classifier = nn.Sequential(
    nn.Linear(model.classifier[1].in_features, 512),
    nn.ReLU(),
    nn.Dropout(0.3),
    nn.Linear(512, num_classes)
)
```

Transfer Learning Pipeline Development

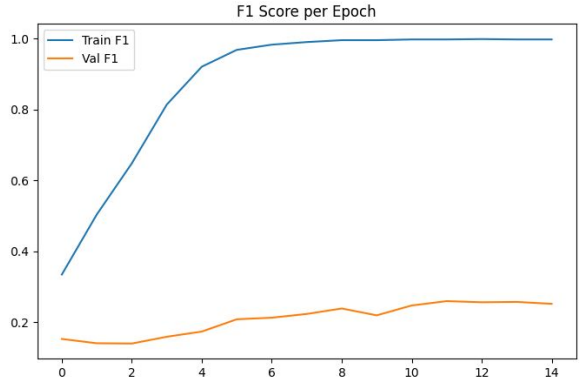
- Details on the **MLP** in front of the feature extractor (example with EfficientNet)

  The simple linear layer was the one giving the best result as depicted here:

```
model.classifier[1] = nn.Linear(model.classifier[1].in_features, num_classes)
```

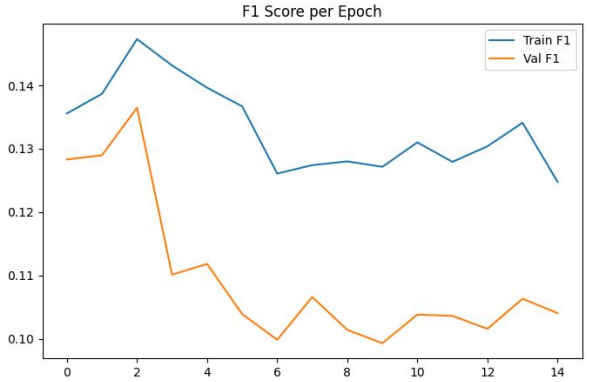


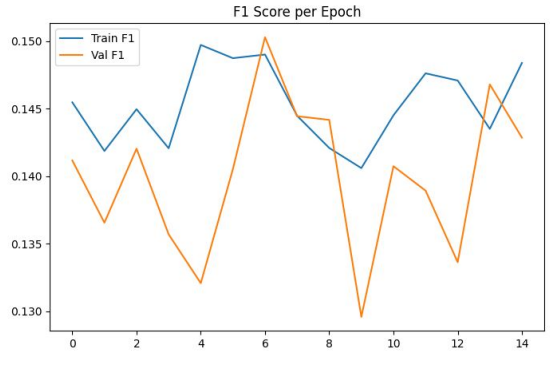
With a best Val F1: 0.2596



Without the simple linear layer

With a best Val F1: 0.1503

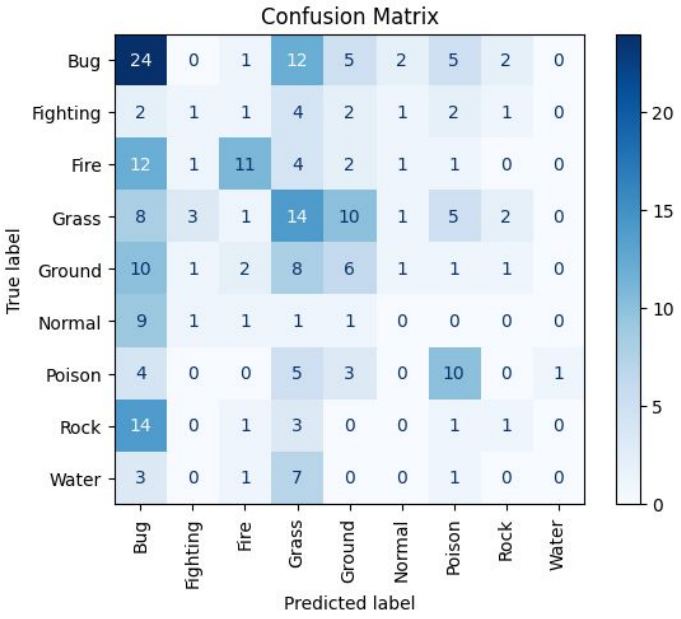




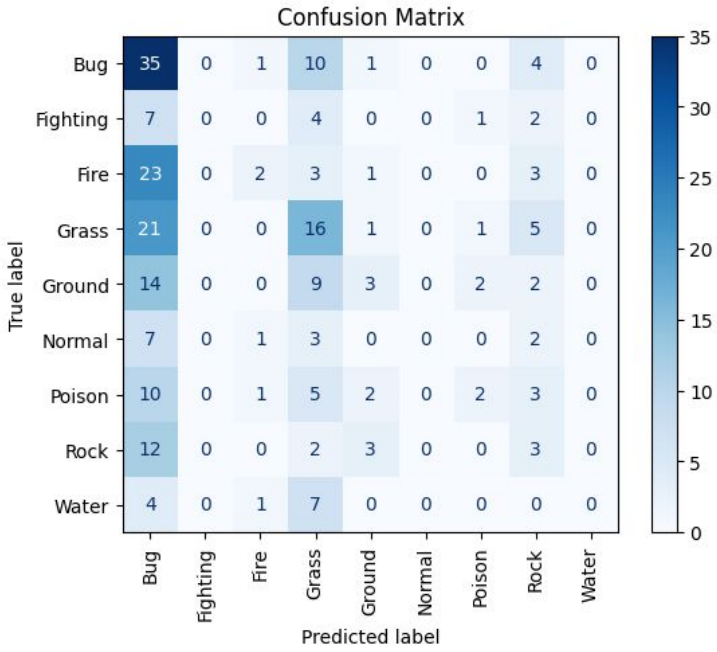
Transfer Learning Pipeline Development

- Explanation and justification for design choices using quantitative arguments.

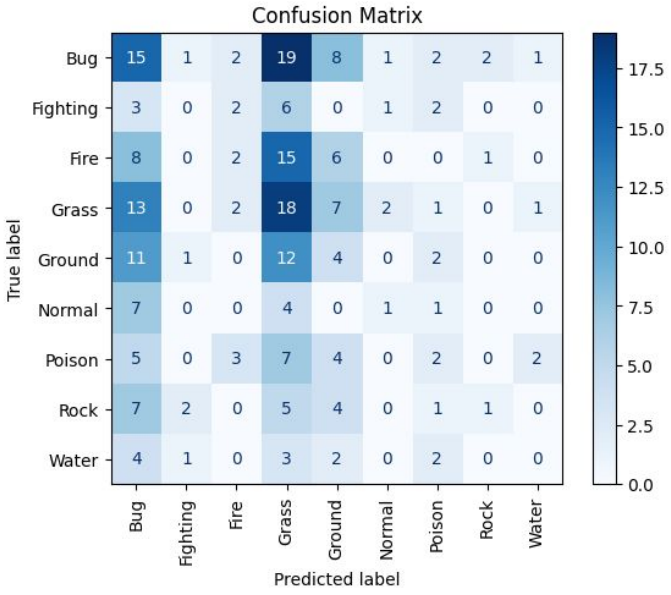
EfficientNet best f1: 0.25



ResNet best f1: 0.19



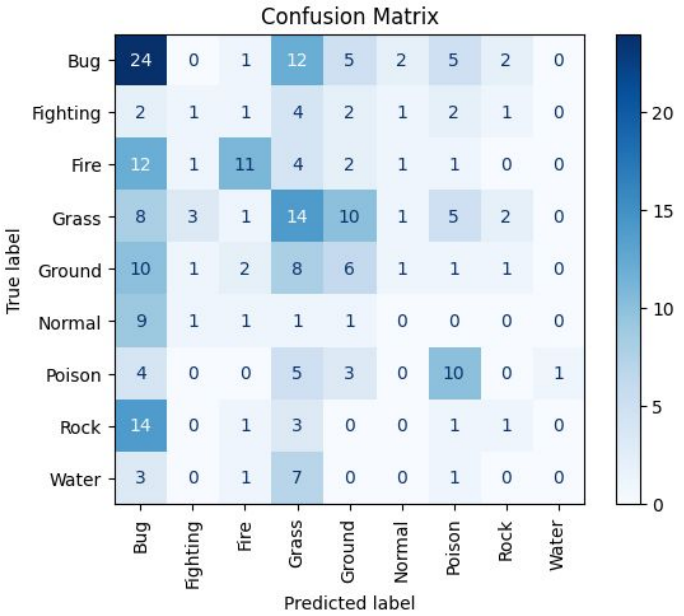
VGG best f1: 0.12



Transfer Learning Pipeline Development

- Explanation and justification for design choices using quantitative arguments.

EfficientNet best f1: 0.25



As we can see in the previous slide, this model seems to be the best

though the other ones seem to have similar results. However, EfficientNet gives a better f1 score and it seems to predict more classes than the other one that predict 2 classes more. Given the f1 score and the cohesion matrix, EfficientNet seems to be the best choice for our work.

In conclusion, we choose the **EfficentNet model**, trained on ImageNet dataset.

Transfer Learning Pipeline Development

- Data regularization:

  When using pre-trained models, it's best to normalize images

  with ImageNet statistics since the model we are using is trained with the ImageNet dataset:

  mean = [0.485, 0.456, 0.406]

  std = [0.229, 0.224, 0.225]

  This keeps the input distribution aligned with what the model expects, ensuring better transfer learning performance. Using our custom dataset stats is only recommended when training from scratch (what we did before: mean = [0.4464, 0.4480, 0.4158],std = [0.1823, 0.1728, 0.1813]).

```
# Data augmentation and regularization
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

Transfer Learning Pipeline Development

- Ablation study

→model: changing the model will have a big impact on all the training phase, since they don't have the same architecture.

→MLP: after freezing the pretrained model we implement an MLP that we trained. As we explained, this architecture also have a big impact on the training because it is the one used in this phase.

→ learning rate: we tested different cases with different learning rate, the one that seems to be the best is: **1e-5**.

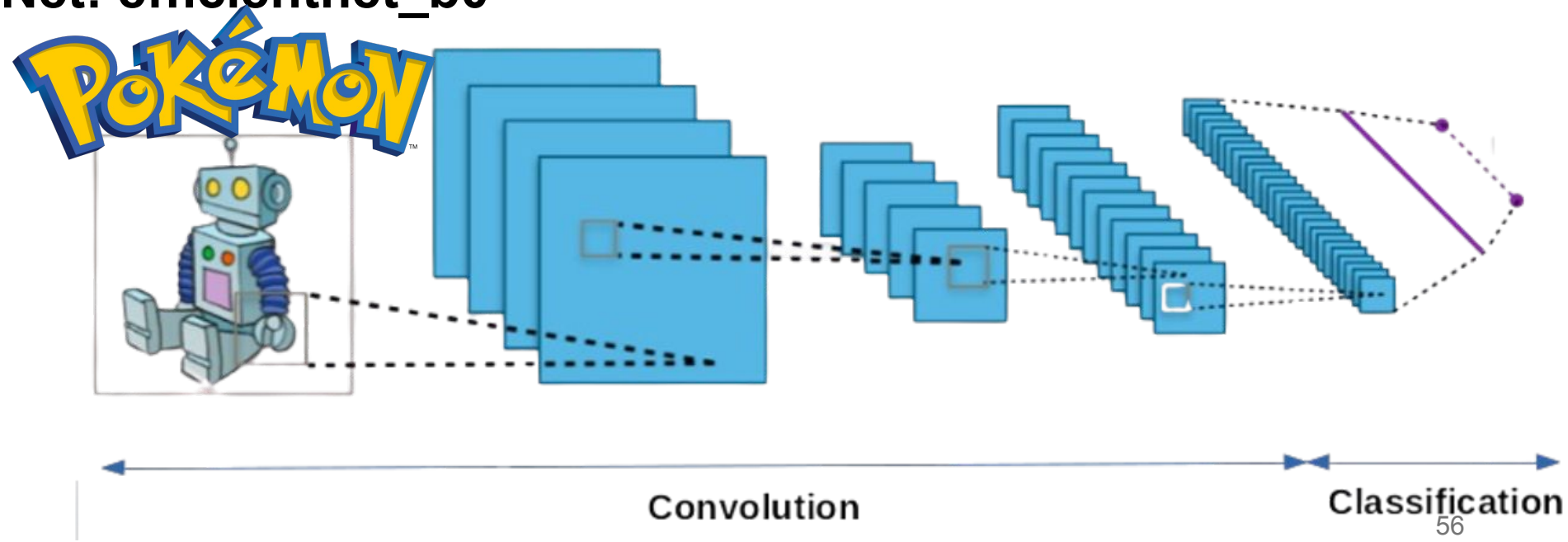
→ number of epoch: as we saw in the first task thanks to early stopping, training an MLP can be done in about **15 epochs**. Notice that we try with bigger numbers of epoch, but none of them show better results than with 15.

Fine-Tuning Pipeline Development

- Models tested.
- Data augmentation.
- Details on the training procedure and hyperparameters.
- Explanation and justification for design choices using quantitative arguments.
- Ablation study

Fine-Tuning Pipeline Development

- Models tested:
  - **Resnet: resnet50**
  - **VGG: vgg16**
  - **EfficientNet: efficientnet_b0**



Fine-Tuning Pipeline Development

- Data normalization:

  It is essential for **fine-tuning** because it helps improve the model's generalization ability by artificially increasing the diversity of the training dataset.

  This is especially important when working with limited data (as we do), as it reduces **overfitting** and allows the model to learn more reliable features.

  Data augmentation is performed using several transformations:

  - images are resized to 224x224 pixels,
  - randomly flipped horizontally,
  - rotated by up to 15 degrees,
  - and their brightness, contrast, saturation, and hue are slightly modified using color jittering.

Finally, the images are converted to tensors and normalized using the mean and standard deviation values from the ImageNet dataset.

```python
# Data augmentation and regularization
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])

val_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

Fine-Tuning Pipeline Development

- Details on the training procedure and hyperparameters.

  →To find the best hyperparameters (**batch size and learning rate**), we train our model with different ones and print the accuracy and f1 score for each configuration.

  At many times, we face **overfitting**, because our Training accuracy flew directly to 1. Facing that, we avoid the configuration that presented overfitting.

Final results was: **batch size = 32** as our previous work and **learning rate = 1e-5** for the first training and **= 1e-4** for the second one (when defreezing all the layer).

Fine-Tuning Pipeline Development

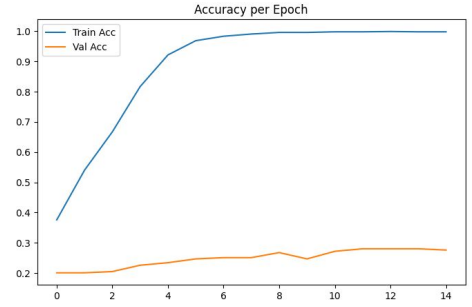
- Explanation and justification for design choices using quantitative arguments. (F1 score before and after fine-tuning)

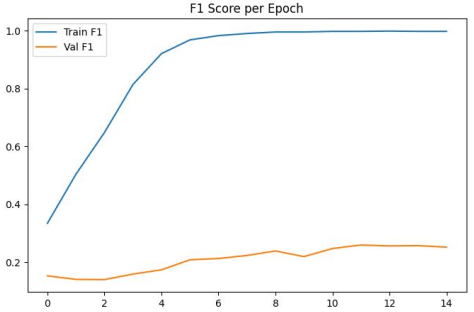
Unfreeze all layer (overfitting?)

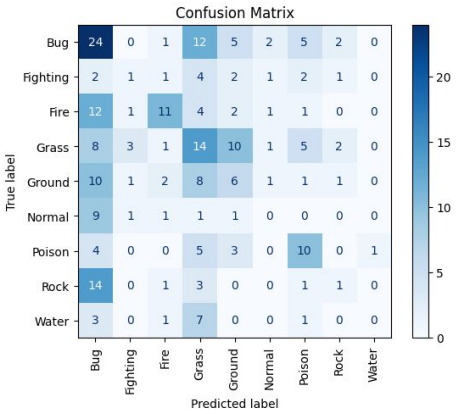
~0.17→0.2596

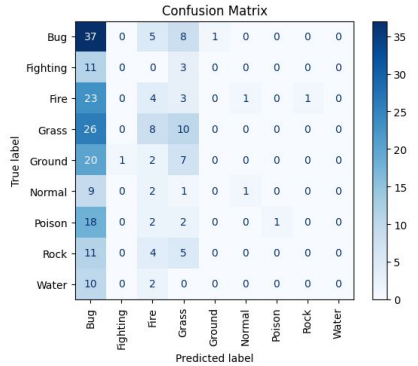
Unfreeze just 1 or 2 layers

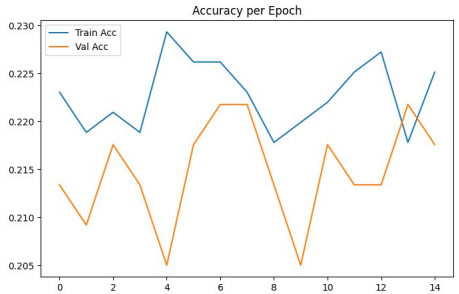
~0.12→ 0.1503

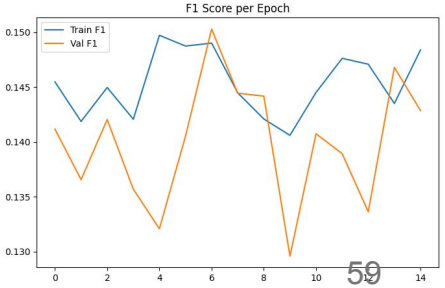












Fine-Tuning Pipeline Development

- Explanation and justification for design choices using quantitative arguments.

⇒ With the previous slide, we clearly notice that the configuration where we **unfreeze all the layers** is the one giving the best result. However, seeing the f1 and accuracy graphs, we may still suspect that our model is kind of **overfitting** with the training dataset.

Fine-Tuning Pipeline Development

- Ablation study

→ Base model selection: We started by comparing different pre-trained models (ResNet, VGG and EfficientNet) and selected **EfficientNet** for its superior performance.

→ Fine-tuning: We tested multiple strategies: freezing all layers except the last, unfreezing only a few layers, fully unfreezing from the beginning, and a two-stage approach (**freezing first, then unfreezing all layers after 15 epochs)**, which gave the best results.

→ Learning rate: Various **learning rates** were tested (1e-5, 1e-4, 1e-3), and the best results came from a two-phase setup: **1e-5 for the frozen stage, followed by 1e-4 after unfreezing.** That result was kind of surprising because we thought that the learning for the fine-tuning phase must be way smaller than the one before, but this configuration gave us the best results.

→ Epoch number: We also evaluated the impact of training duration (15 vs. 30 vs. 50 epochs), and found that **15 + 30 epochs** provided a good balance between convergence and generalization.

→ Batch size: Finally, we tried different **batch sizes**, and although small (batch size = 32), it worked best given memory constraints and model stability, as in the first 2 tasks.

Training Efficiency

- Training time
- GPU usage
- Strategies employed for efficient training (early stopping, batch sizes, etc.)

Training Efficiency

- Training time
  - Running locally with GPU = Nvidia Geforce RTX 4070 & CPU = AMD Ryzen 7 7745HX
  - With CUDA enabled, the **transfer learning** part took **53s** and the **fine tuning** part took **2m42s**.
  - Running only on the CPU, the **transfer learning** part took **9m25s** and the **fine tuning** part took **30m52**.

Note

For the next part: comparison part, we used the **CNN implemented from scratch in Task 2** on the new dataset.

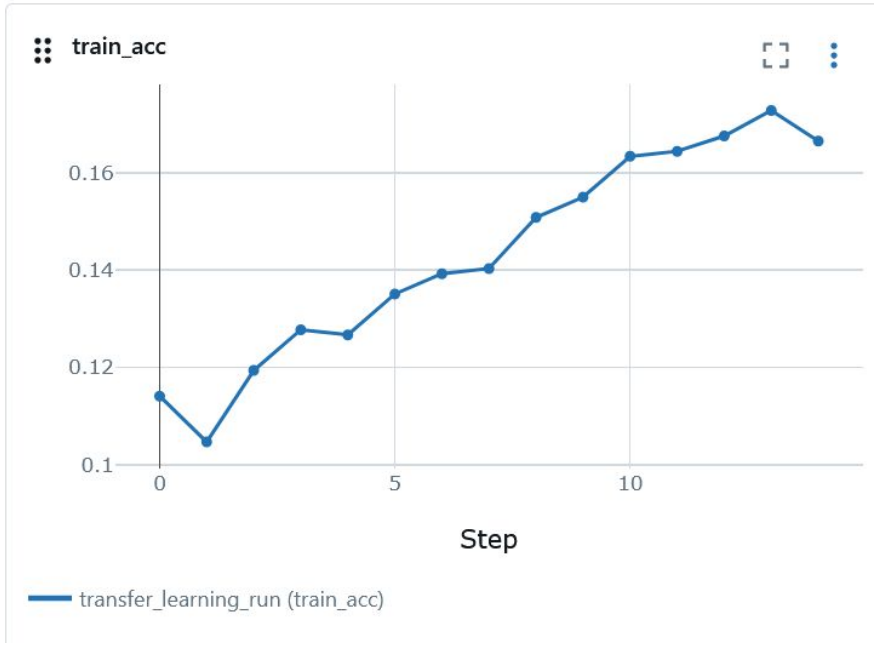
Moreover, as the dataset is more complex, we could not use the preprocessing tool we created (removing the background and centering the pokémon) so the results may be lower from the ones in Task 2.

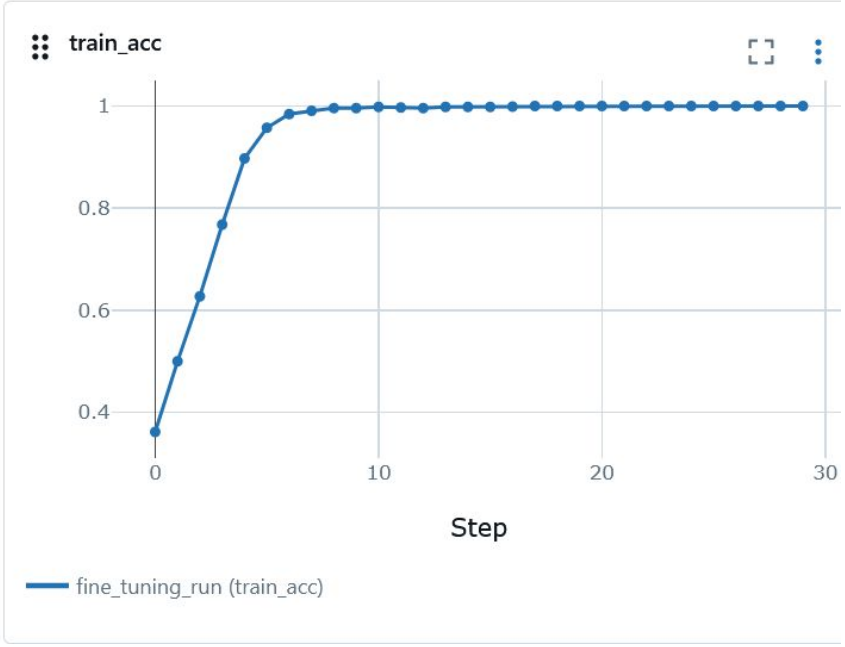
Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

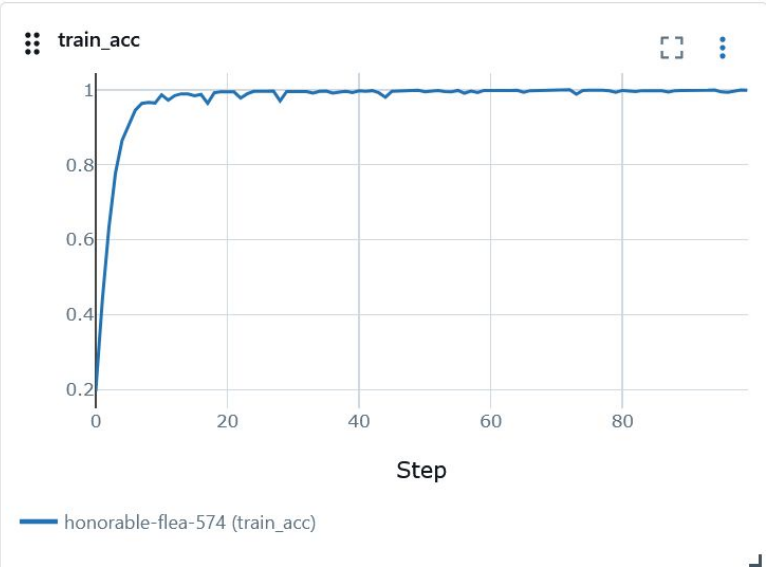
- Side-by-side performance metrics, with tables and plots
- Analysis of improvements and remaining challenges

Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

- **Training accuracy plots**: from left to to right = Transfer Learning, Fine Tuning, CNN (exact same order will be used in







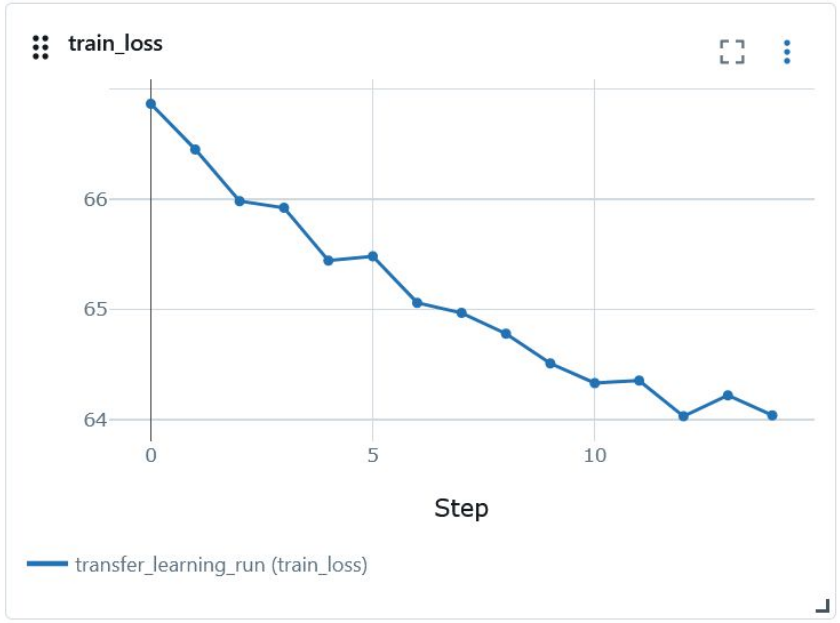
Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

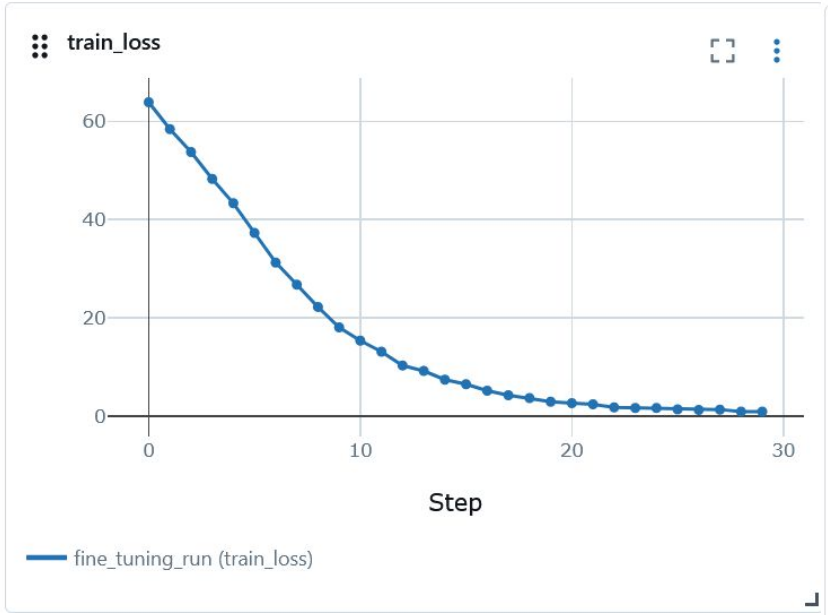
- Training accuracy **analysis**

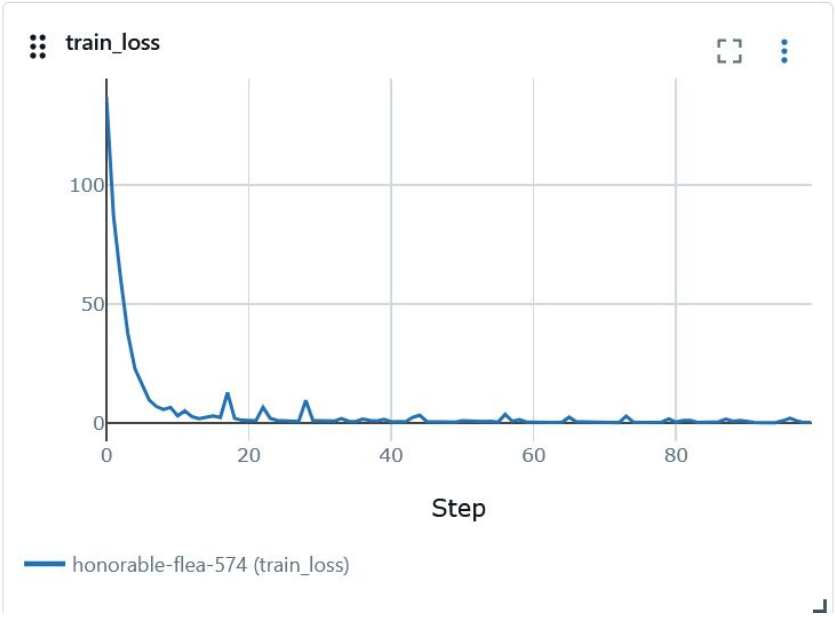
  - The CNN model reaches perfect training accuracy (~1.0) after about 10 steps, indicating strong learning capacity from scratch.
  - Fine-tuning converges even faster, achieving near-perfect accuracy in fewer than 5 steps, likely due to pre-learned features being quickly adapted.
  - Transfer learning progresses much more slowly and plateaus below 0.25, suggesting limited flexibility from frozen layers or not optimal hyperparameters.

Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

- **Training loss plots**







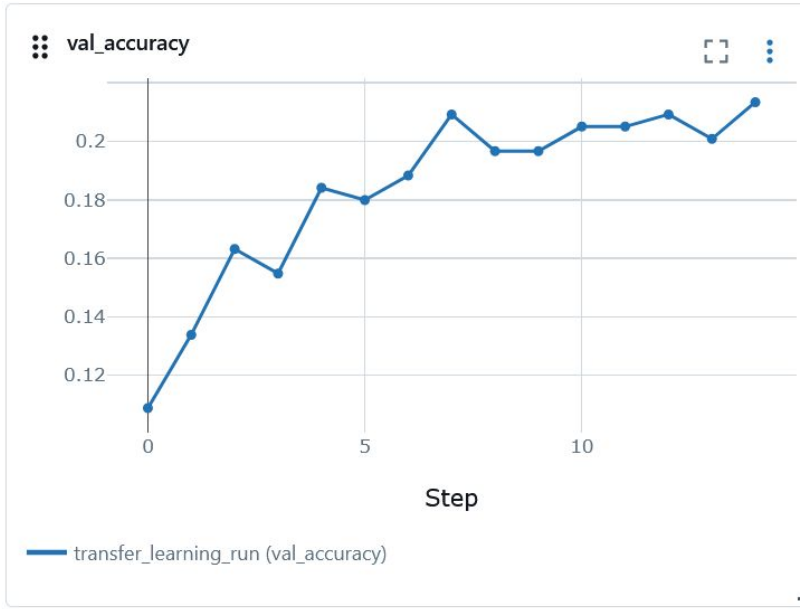
Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

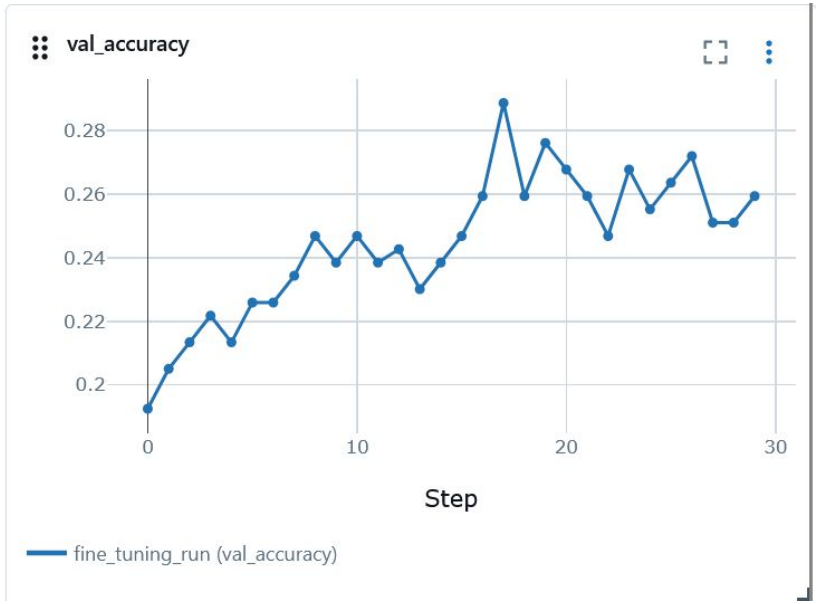
- Training loss **analysis**

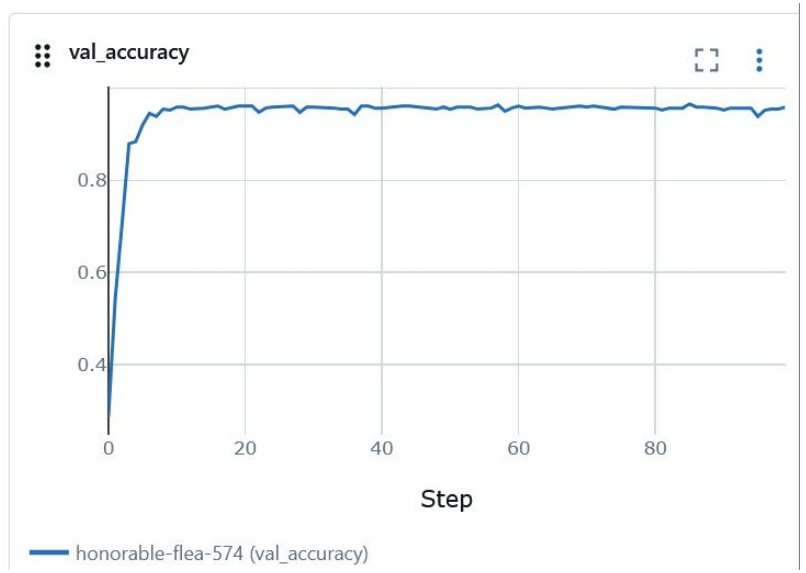
  - The training loss decreases fastest and most significantly in the fine-tuning and CNN models, with fine-tuning reaching near-zero loss in fewer steps.
  - The transfer learning model shows much slower progress, indicating limited capacity to adapt due to frozen layers.
  - The CNN model exhibits some fluctuations, but still reaches very low loss values.

Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

- **Validation accuracy plots**





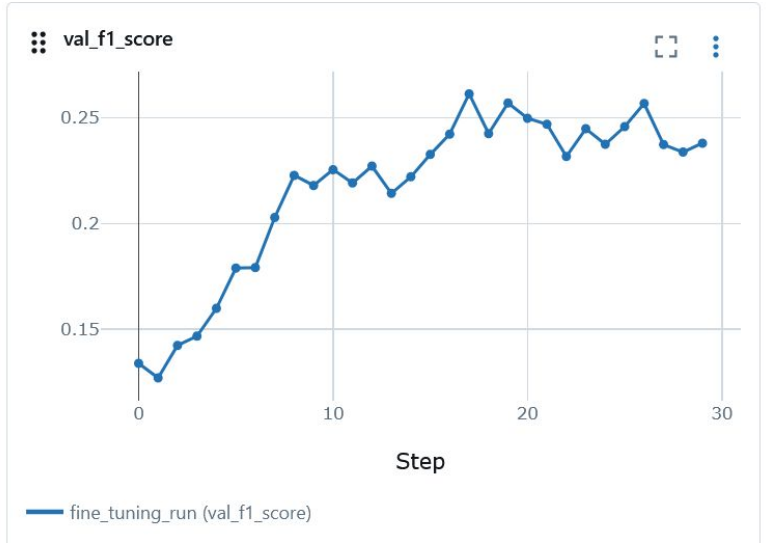


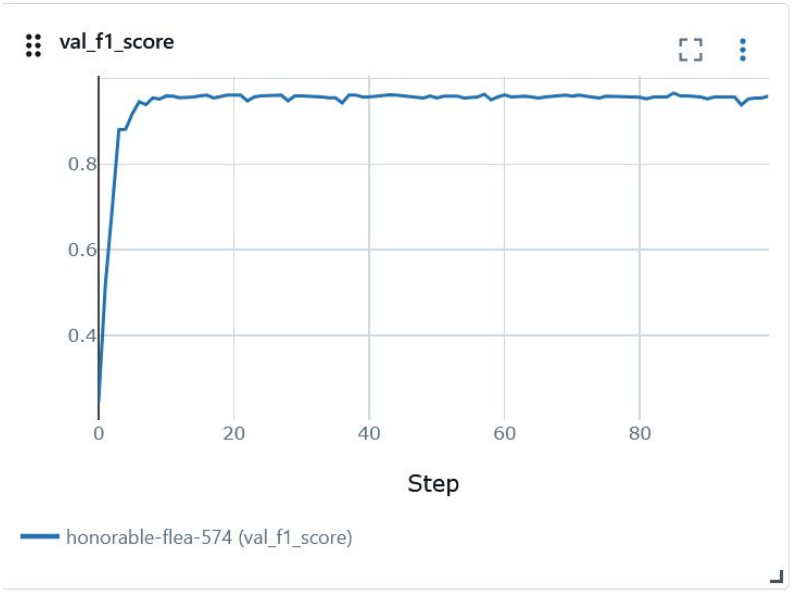
Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

- Validation accuracy **analysis**
  - Fine-tuning shows better generalization than transfer learning, with validation accuracy peaking around 0.28.
  - However, the CNN model achieves very high validation accuracy (~0.95), suggesting it fits the data very well — though the risk of overfitting remains, considering the gap between training and validation performance.
  - Transfer learning lags behind, reaching only ~0.21.

Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

- **F1 Score plots**



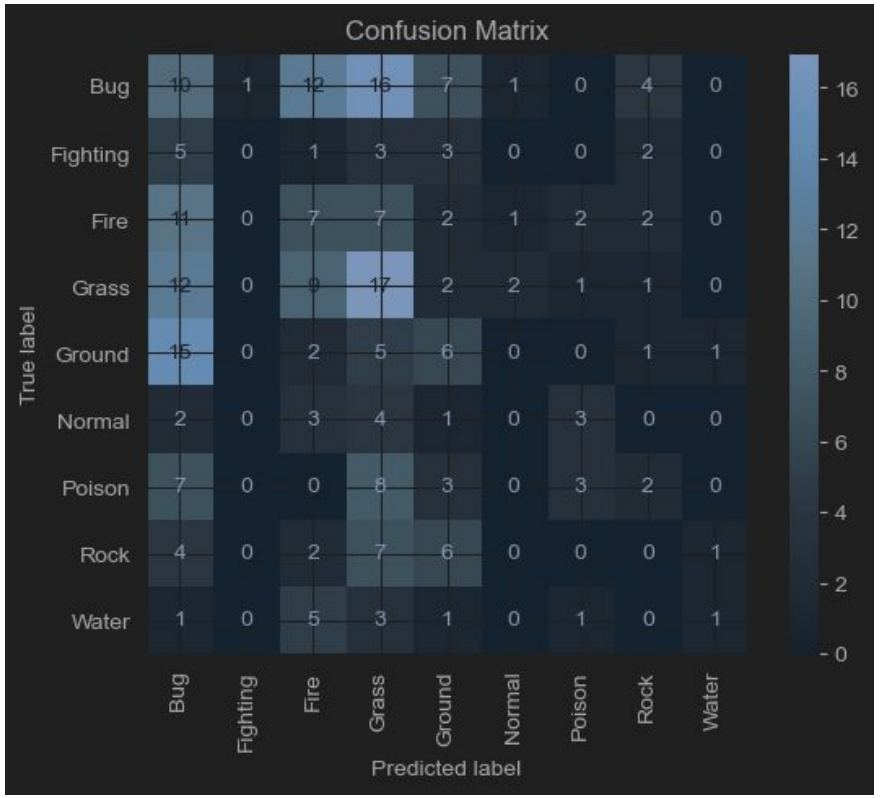


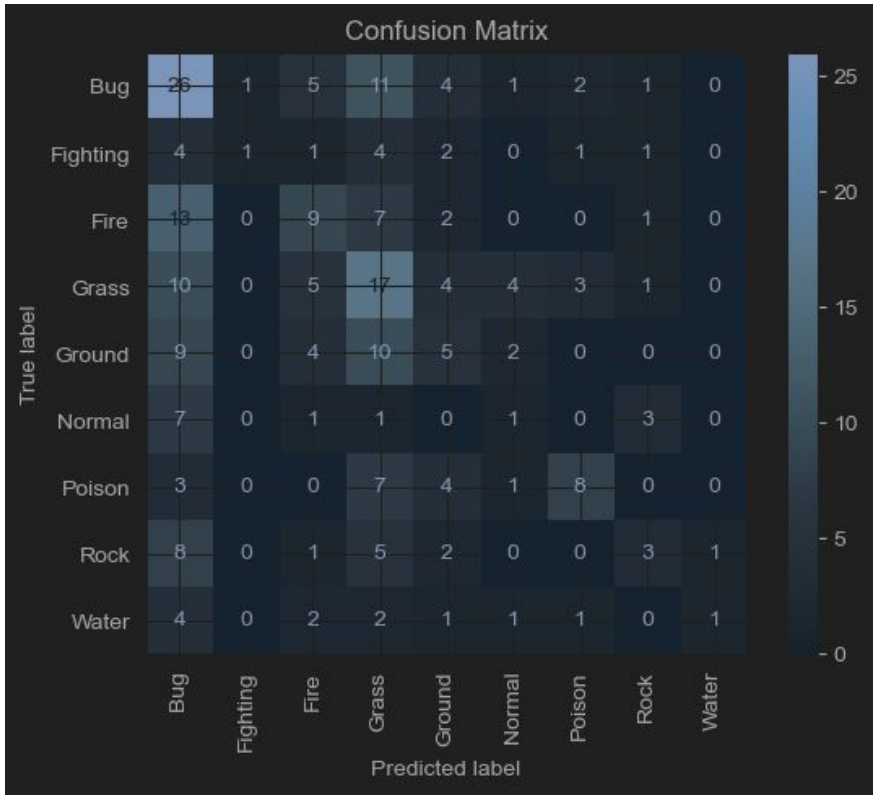
Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

- F1 Score **analysis**
  - Transfer Learning shows a modest F1 score improvement, reaching around 0.18, indicating limited adaptation without further training.
  - Fine-Tuning performs better, steadily increasing to an F1 score of about 0.26, showing the benefits of adapting the pre-trained model more deeply.
  - Custom CNN from Scratch outperforms both, achieving an F1 score above 0.9, suggesting that the model architecture and training from scratch were highly effective for this specific dataset.

Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

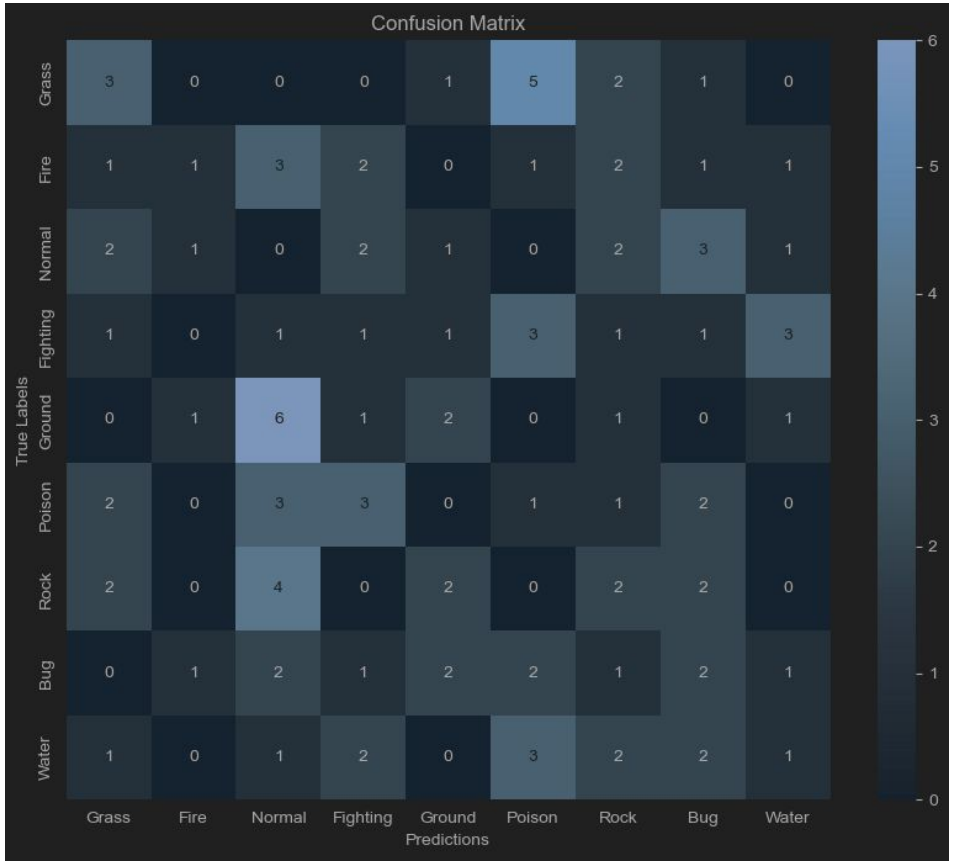
- **Confusion matrix** of Transfer Learning (left) and Fine Tuning (right)





Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

- **Confusion matrix** of CNN



Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

- Confusion matrix **analysis**
  - CNN: Shows moderate performance with noticeable misclassifications.
  - Transfer Learning: Demonstrates improved accuracy with fewer misclassifications.
  - Fine-Tuning: Achieves the highest accuracy, indicating the best performance among the three models.

Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

Observed **Improvements**

- Fine-Tuning achieved the best performance, demonstrating near-perfect accuracy. This indicates strong learning capacity and excellent adaptation to the dataset.
- Fine-Tuning clearly outperforms both basic Transfer Learning and the CNN trained from scratch, showing the benefits of unfreezing and adjusting pre-trained layers.
- Training loss decreases rapidly for the Fine-Tuning model, confirming effective learning.

Performance comparison (Transfer Learning vs Fine-Tuning vs CNN from scratch)

Remaining Challenges

- The gap between training and validation accuracy, especially in the CNN model trained from scratch, suggests that while the model learns the training data well, there may still be room to improve generalization.
- Transfer Learning without fine-tuning shows limited performance on this dataset, which may indicate that the pre-trained features alone are not fully aligned with the task requirements.
- A moderate train/validation divergence across all methods could also come from characteristics of the dataset, such as class imbalance or domain-specific features.

Lessons learned and insights gained

- How small details like learning rate or MLP layer might have big impacts on the result
- How to use a pretrained model on a different dataset
- The importance of considering the balance between model complexity, computation time, and final performance to choose the most suitable approach based on project constraints
- Identifying specific use cases where each approach excels: CNN from scratch for very specialized tasks without relevant pre-trained models, transfer learning for rapid deployments with limited resources, and fine-tuning for optimal performance

What went wrong

- Training time was really long during this phase (to be more efficient, we use the GPU available for free on kaggle)
- Finding improvement by making choice between the model, the MLP architecture, the learning rate, the number of epoch and then the fine-tuning phase: which layer to defreeze. It was really hard to find the good one that would make really good improvements.

What went great

- Using kaggle GPU to train our model
- Understanding each part of transfer learning and fine-tuning
- Trying to work with a different dataset of pokémon images.