Creating Labels

In this step, we're establishing the groundwork for our Pokémon classification system by generating a list of labels, which correspond to the different Pokémon species we aim to classify. This process is crucial because these labels will later serve as the target variables for our machine learning model.

Here’s a breakdown of the code:

1. Setting the Directory: We specify the path to the directory containing our data. In this case, the path is "/kaggle/input/pokemonclassification/PokemonData". This directory contains subdirectories named after each Pokémon species, and each subdirectory holds images of that specific Pokémon.
2. Listing Directories: Using os.listdir(directory), we retrieve the names of all subdirectories within our specified path. These names directly correspond to the Pokémon species we have images for, thus serving as our labels for classification.
3. Counting Labels: The len(labels) function counts the total number of unique labels (or Pokémon species) that we have data for. This number is important as it tells us the scope of our classification task in terms of variety.
4. Outputting Labels: Finally, the print(labels) statement outputs the list of Pokémon species names. This output is not just a validation of the successful extraction but also provides a clear view of all the Pokémon species we will be working with. For instance, some of the Pokémon species listed are 'Golbat', 'Beedrill', 'Caterpie', etc.

Converting Images to Numpy Arrays and Preprocessing

In this step, we are processing the images to prepare them for use in our machine learning model. This involves several crucial steps, each important for ensuring the images are in a format suitable for effective training:

1. Function Definition: The function input_target_split is designed to handle the preprocessing of images. It takes two parameters: train_dir (the directory where the images are stored) and labels (the list of Pokémon species).
2. Initializing Data Structures:
    - stored: A dictionary to keep track of the numeric labels assigned to each Pokémon species.
    - dataset: A list that will hold tuples of images and their corresponding labels.
3. Image Processing Loop:
    - For each label, corresponding to a Pokémon species, the function navigates to the folder containing its images.
    - It then iterates over each image in the folder, applying several preprocessing steps:
        - Image Loading and Resizing: load_img loads an image and resizes it to 150x150 pixels. This uniform size is crucial for training the model as it expects inputs to be of a consistent shape.
        - Conversion to Array and Normalization: img_to_array converts the loaded image into a numpy array. The pixel values of the image are then normalized by dividing by 255.0. This normalization step brings pixel values into the [0,1] range, improving model convergence during training.

- Each processed image and its label (encoded as an integer `count`) are appended as a tuple to the `dataset` list.

4. Post-Processing:
    - The labels are printed out as each is completed, giving feedback on the progress.
    - The dataset is shuffled to ensure that the model does not learn any unintentional patterns from the order of the data.
    - Finally, the dataset is split into features (`X`) and labels (`y`), which are converted into numpy arrays for compatibility with machine learning frameworks.
5. Return Values: The function returns two numpy arrays, `X` and `y`, which represent the features (images) and labels (integer indices corresponding to Pokémon species), respectively.
6. Function Execution: The function `input_target_split` is called with the directory containing the Pokémon data and the list of labels, and it returns the prepared feature and label arrays `X` and `y`.

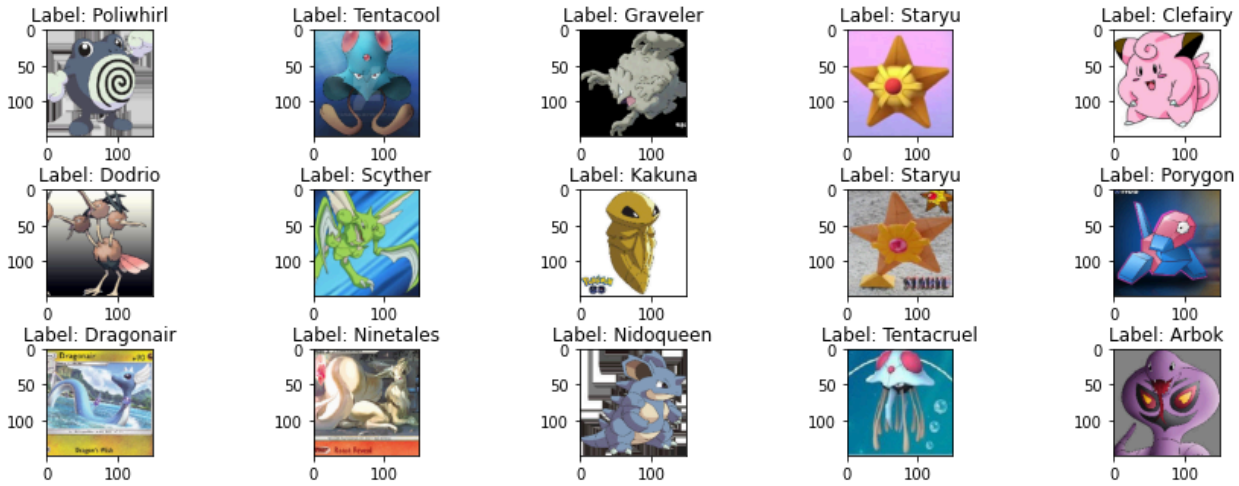
Visualizing the Images and Their True Labels

After preprocessing and loading the images, visualizing them alongside their true labels is an invaluable step. This helps to verify that the images have been correctly processed and labeled, and provides a clear visual reference for understanding the dataset's composition. Here's how this visualization is structured:

1. Setting Up the Plot:

- We use matplotlib.pyplot to create a figure with a specified size, here 15x9 inches, which provides enough space to view multiple images clearly.
- We initialize a counter n to manage the subplot indices.

2. Loop for Displaying Images:
    - A loop runs 15 times (as indicated in the code), creating subplots for each image. This number could represent a batch or a subset of the total dataset to keep the visualization manageable and meaningful.
    - In each iteration, the subplot is configured using plt.subplot, specifying a grid of 5x5. Adjustments are made for spacing between images using plt.subplots_adjust.
    - The image is displayed using plt.imshow(X[i]), which shows the preprocessed image from our dataset X.
    - Each image is titled with its corresponding label using plt.title(f'Label: {labels[y[i]]}'). This label is fetched from the labels list using the index from array y, which provides a direct mapping of images to their Pokémon species.
3. Displaying the Plot:
    - After setting up all subplots, the complete visualization is displayed. This step is crucial for a visual inspection of the dataset, ensuring that each image is correctly processed and labeled.
4. Unique Labels Check:
    - The output np.unique(y) lists all unique label indices present in the dataset. This provides a validation point to ensure that all intended Pokémon species are represented in the dataset and have been correctly indexed.



Train Test Split and Encoding of Labels

This section of the code deals with preparing the dataset for training by splitting it into training and test sets and encoding the labels. Here's a step-by-step breakdown:

1. Train Test Split:
    - Using `train_test_split` from the `sklearn.model_selection` module, the dataset `X` (features, i.e., images) and `y` (labels) are divided into training (`X_train`, `y_train`) and testing (`X_test`, `y_test`) sets. The split ratio is specified as 22% for the test set, with a `random_state` of 42 to ensure reproducibility of results.
    - This split helps in validating the model on unseen data, ensuring that it generalizes well beyond the data it was trained on.
2. Checking Distribution of Labels:
    - After the split, `np.unique(y_train, return_counts=True)` and `np.unique(y_test, return_counts=True)` are used to print the unique labels and their counts in the training and test sets, respectively. This check is crucial to ensure that both splits

have a fair representation of all Pokémon species, preventing any bias toward certain classes.

3. Image Data Augmentation:
    - An ImageDataGenerator is set up for the training data with parameters designed to introduce randomness into the training process by augmenting the images. These augmentations include flipping, rotation, zoom, shifts, and shear. Such transformations help in making the model robust to variations in new data that it might encounter after deployment.
    - A separate ImageDataGenerator for the test data is created without any augmentations, as the test set should ideally represent the real-world scenario without modifications.
4. Fitting Data Generators:
    - The fit method is applied to X_train with datagen and to X_test with testgen. This step computes any necessary statistics to perform the specified augmentations. For instance, if z-score normalization is used, this would calculate the mean and standard deviation of the dataset.
5. One-Hot Encoding of Labels:
    - The labels y_train and y_test are converted from integer forms to one-hot encoded formats using np.eye(nb)[y_train] and np.eye(nb)[y_test]. nb is the total number of unique labels (species), and np.eye(nb) creates an identity matrix of size nb. One-hot encoding transforms the label into a binary vector which is more suitable for classification tasks with multiple classes, ensuring that the model treats each class as equally distinct.

Freezing the DenseNet201 Model

In this step, we're configuring the DenseNet201 model to use it as a base for ourPokémon image classification. This approach, using a pre-trained model, is known as transfer learning, which can significantly speed up the training process and improve the model's performance, especially when wehave a relatively small dataset. Here's what's happening in ourcode:

1. Model Setup:
    - DenseNet201 Configuration: DenseNet201 is initialized with include_top=False, which means that the top (or final) fully connected layers are not included. This is typical when using DenseNet for transfer learning, as wewill add ourown classifier suited to ourspecific task (150 classes of Pokémon).
    - Weights: The model weights pre-trained on ImageNet are loaded to give the model a head start in learning visual patterns.
    - Input Shape: The input shape is set to (150, 150, 3), matching the preprocessing done earlier where images were resized to 150x150 pixels with 3 color channels (RGB).
2. Layer Freezing:
    - The layers up to the 675th layer of the DenseNet201 model are set to layer.trainable = False. This freezes the weights of these layers during training, which means that they will not be updated. Freezing is used to preserve the learned features that are usually general while speeding up training and reducing the risk of overfitting on the specific Pokémon data.

- Layers from 675 onwards are set to `layer.trainable = True`, allowing them to update during training. This is where the model can learn features more specific to the Pokémon dataset.

3. Downloading Weights:
    - The weights for DenseNet201 are downloaded from TensorFlow's servers, ensuring that wehave the latest pre-trained models. This download step happens automatically when the weights are not found locally.

Constructing the Feedforward Network for Pokémon Classification

In this part of ourproject, we're building the final layers of ourclassification model, which will specifically tailor the DenseNet201 base to the task of classifying 150 different Pokémon. Here's a breakdown of how we're constructing this feedforward network:

1. Sequential Model:
    - we're using Keras' `Sequential` model, which is a linear stack of layers. This is appropriate for a feedforward neural network where wehave a clear beginning and end, with data flowing sequentially through transformations.
2. Adding the Base Model:
    - The `base_model` (pre-configured DenseNet201) is added as the initial layer of the `Sequential` model. Since it's already set up to handle the input shape and preprocessing, it acts as a sophisticated feature extractor.
3. Global Average Pooling:

- After the base model, a GlobalAveragePooling2D layer is added. This layer reduces each feature map in the output of DenseNet201 to a single average value, significantly reducing the number of parameters in the model. This step not only helps to minimize overfitting but also reduces computational complexity.

4. Dense Output Layer:
    - A Dense layer with nb (150) neurons is added, one for each Pokémon class. The activation function used is softmax, which is standard for multi-class classification tasks as it outputs a probability distribution over the classes.
5. Compilation:
    - The model is compiled with the Adam optimizer, a popular choice that adapts learning rates during training and offers good convergence on a wide range of tasks. The learning rate is set to 0.001.
    - The loss function is categorical_crossentropy, suitable for multi-class classification problems where each class is represented as a one-hot encoded vector, as is the case with ourPokémon labels.
    - The metric used to evaluate the model is accuracy, which will provide a straightforward interpretation of performance during training and testing.

Evaluation of the Pokémon Classification Model

The classification report we ha've generated provides detailed insights into the performance of our model across all 150 Pokémon species. Here's an analysis of some key metrics and what they indicate about our model's effectiveness:

1. Precision:
    - Measures the accuracy of positive predictions. Formulated as the ratio of true positives to the sum of true and false positives. High precision relates to a low false positive rate.
    - Most Pokémon have very high precision, often hitting 1.00, meaning that when a class is predicted, it is often correct.
2. Recall:
    - Also known as sensitivity, it measures the ability of the model to find all the relevant cases (all Pokémon of a specific type).
    - Many classes also have a recall of 1.00, showing that the model is excellent at capturing all relevant instances. However, there are a few, like Marowak and Vulpix, where recall is significantly lower, suggesting these Pokémon could be harder to detect or are confused with others.
3. F1-Score:
    - The F1-score is the weighted harmonic mean of precision and recall. The highest possible value of an F1-score is 1.0, indicating perfect precision and recall, and the lowest possible value is 0, if either the precision or the recall is zero.
    - The F1-scores are generally very high, showing a good balance between precision and recall. This is particularly important in a balanced dataset where the impact of each class is similar.
4. Support:

- The number of actual occurrences of the class in the specified dataset. It's vital for assessing the evaluation metrics within the context of how many examples were available.
- Some Pokémon like Butterfree have very few samples (1), which might make their perfect scores slightly less robust compared to classes with more samples.

5. Overall Accuracy:
    - The overall accuracy of the model is 90%, which is quite high, especially considering the diversity and similarity among Pokémon species, making it a challenging classification problem.
6. Macro and Weighted Averages:
    - Macro-average computes the metric independently for each class and then takes the average (hence treating all classes equally), resulting in 0.90.
    - Weighted average accounts for class imbalance by weighting the average of each class's score by the number of true instances when calculating the average.

Key Takeaways

- our model performs excellently across most Pokémon classes, with high marks in precision, recall, and F1-score.
- There are some classes where the performance metrics are lower, which could be due to various factors like fewer training samples or more complex features that are harder to learn.
- Overall, the model demonstrates robust generalization capability across a wide variety of Pokémon types, making it a successful classification system for this particular task.

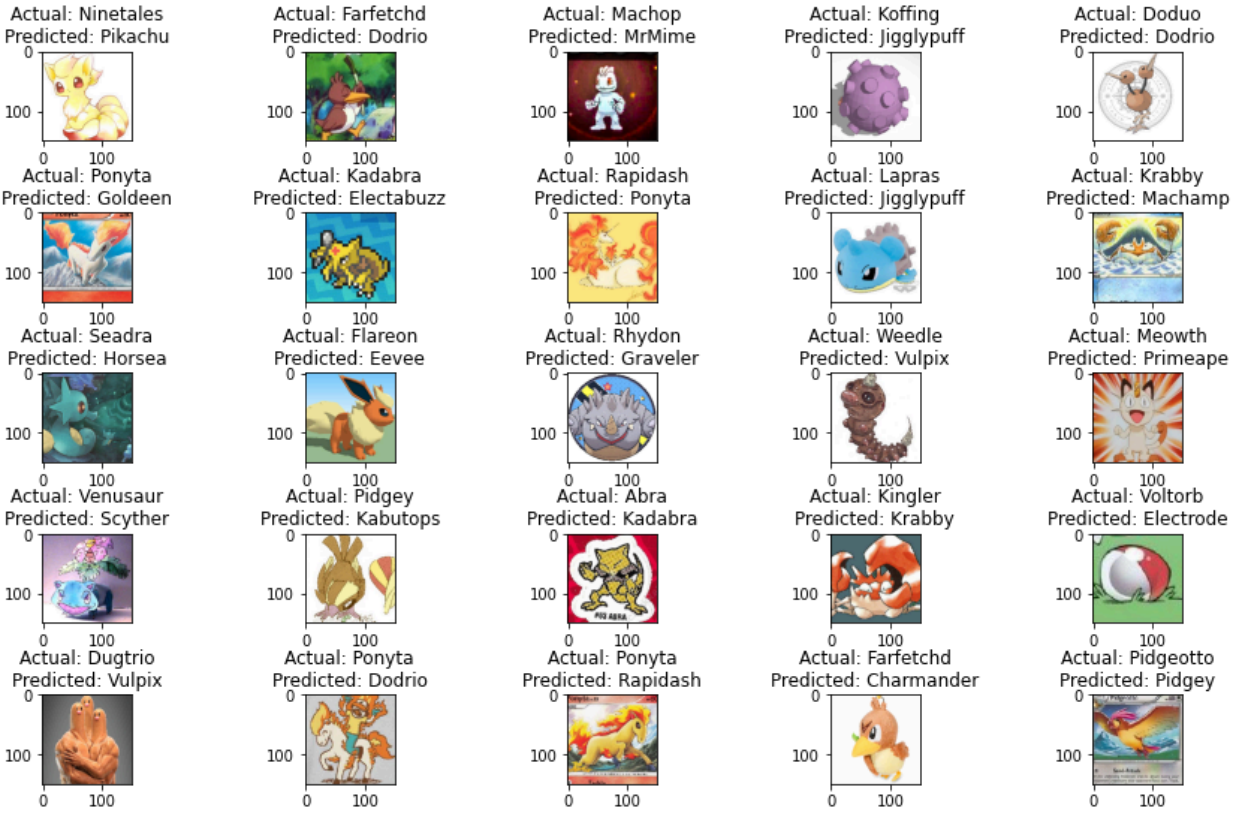
Visualizing Wrongly Classified Images

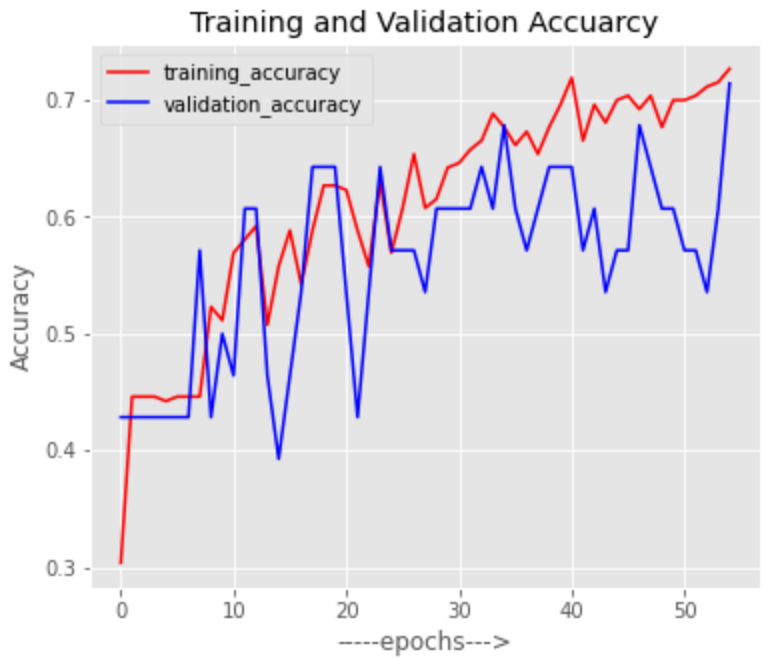
Our approach to visualizing wrongly classified images is a great way to gain insights into where the model might be struggling and to identify potential patterns or characteristics that could be causing the misclassifications. Here's a breakdown of how Our visualization code works and its importance:

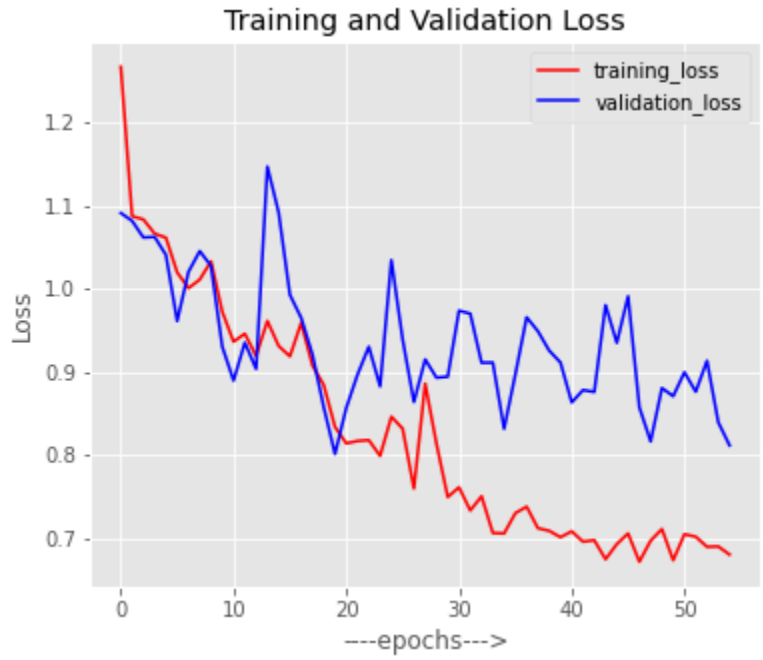
1. Setting Up the Plot:
    - A figure is initialized with a size of 15x9 inches, which provides ample space to display multiple images clearly.
    - `plt.figure(figsize=(15, 9))` sets up a new figure for plotting.
2. Loop Through Test Set:
    - The loop iterates through each image in the test dataset (`X_test`). For each image, it checks if the predicted label (`y_pred[i]`) matches the actual label (`y_true[i]`).
    - `y_pred` should be derived similarly to how `y_true` is obtained, typically using `np.argmax()` on the model's output probabilities if they are not already in this format.
3. Condition for Visualization:
    - If a prediction is incorrect, the code increments the counter `n`.
    - Only the first 25 misclassified images are considered for visualization, which is a manageable number for detailed inspection without overcrowding the figure.
4. Plotting the Images:
    - Each misclassified image is plotted in a subplot within a 5x5 grid. The images are added one by one to the subplots as they are encountered.

- plt.subplot(5, 5, n) dynamically adds a new subplot for each misclassified image.
- plt.subplots_adjust(hspace=0.8, wspace=0.3) adjusts the spacing between these subplots to prevent labels and images from overlapping, improving readability.

5. Image and Title:
   - plt.imshow(X_test[i]) displays the image.
   - The title of each subplot is set to show both the actual and predicted labels, providing immediate visual feedback on the nature of the error (plt.title(f'Actual: {labels[y_true[i]]}\nPredicted: {labels[y_pred[i]]}')).




Training and Validation Accuarcy
training_accuracy
validation_accuracy
Accuracy
0.7
0.6
0.5
0.4
0.3
0
10
20
30
40
50
----epochs--->


Training and Validation Loss
training_loss
validation_loss
Loss
1.2
1.1
1.0
0.9
0.8
0.7
0
10
20
30
40
50
----epochs--->