CNN for butterfly multiclass species classification

Stefano Bozzi

autumn 2025

Abstract

This report analyzes the efficiency of a CNN model in classifying butterfly images as a function of three different hyperparameters.

First, an introduction and a theoretical overview are presented. Next, the processing and organization of the dataset are explained, followed by the analysis of the model's efficiency with varying hyperparameters, and finally the conclusions.

The appendix contains the code used. The project is also available at the following GitHub link: github.com/Stefano-Bozzi/LepidopterAI

Contents

1 Introduction **2**
- 1.1 CNN - a theoretical overview . . . 2
  - 1.1.1 Convolution . . . 2
  - 1.1.2 Activation function . . . 3
  - 1.1.3 Pooling . . . 3

2 Data collection and manipulation **3**
- 2.1 Train, Validation and Test . . . 4
- 2.2 Data Augmentation . . . 4

3 Model Creation and hyperparameters selection **5**
- 3.1 Dropout rate . . . 6
- 3.2 Learning rate and kernel regularizer . . . 7

4 Results **9**

5 Conclusions **12**

A Code **13**
- A.1 Data collection and manipulation . . . 13
  - A.1.1 image load . . . 13
  - A.1.2 train val test split . . . 14
  - A.1.3 data augmentation . . . 14
- A.2 Model creation and hyperparameters selection . . . 15
  - A.2.1 generic parametric function . . . 15
  - A.2.2 dropout rates . . . 16
  - A.2.3 grid search . . . 16
- A.3 Results and metrics . . . 17

1 Introduction

Convolutional Neural Networks (CNNs) were first developed at the end of 20th century for character recognition [1]. They are now central to deep learning, especially in image classification. Like regular neural networks, CNNs have neurons with learnable weights and biases and use loss functions. The key difference is that CNNs assume the input is an image, exploiting spatial structure to reduce parameters and improve efficiency.

1.1 CNN - a theoretical overview

To understand why is impractical to use a fully connected FFNN for images let consider an RGB image 200×200. The number of parameters for each neuron in the first hidden layer is $200 \times 200 \times 3 = 120{,}000$, i.e. $W \in \mathbb{R}^{120{,}000 \times m_1}$.
Even with only $m_1 = 50$ neurons in the first hidden layer, the number of weights amounts to $120{,}000 \times 50 = 6{,}000{,}000$, a number too large [1].

The key idea of CNN is to extract local correlations by detecting patterns in the image, and then combine the extracted information distributed across multiple layers through filters to obtain compact representations that reduce redundancy and improve generalization respect to a FFNN.

The main steps of a CNN can be summarized into three stages:

1. Convolution
2. Activation function
3. Pooling

Before the convolution there is an *input layer*, and after pooling we typically have one or more *fully connected layers*.

1.1.1 Convolution

Given the input $x(t)$, performing a convolution on the input is:

$$s(t) = \int x(t-a)w(a)\,da$$

where the function $w(a)$ is used to extract local behavior from the data [2]. If we consider the case of a two-dimensional image, the signal becomes:

$$s(i,j) = (x * w)(i,j) = \sum_{n,m} x(i-m, j-n)\, w(m,n)$$

$$= \sum_{n',m'} w(i-m', j-n')\, x(m',n') = (w * x)(i,j),$$

where $i, j = 1, \ldots, (\#\,\text{pixels})$. We want the weight function to be *local*.

If we restrict $m, n = -2, -1, 0, 1, 2$, i.e. we correlate 5 input pixels in each direction, we end up with 25 weights, which describe the localized structure in a given region of our grid.

Let us imagine an input matrix $x(i,j)$ of size 3×4, which we want to convolve with weights W of size 2×2. The convolution between these two matrices yields a matrix Z of size 2×3, where each element is the weighted sum of the corresponding 4 input pixels:

$$\begin{bmatrix} a & d & g & l \\ b & e & h & m \\ c & f & i & n \end{bmatrix} * \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} \Rightarrow \begin{bmatrix} A & C & E \\ B & D & F \end{bmatrix}$$

with:

$$A = \alpha a + \gamma b + \beta d + \delta e$$

$$B = \alpha b + \gamma c + \beta e + \delta f$$

In this way, starting from the inputs, we apply the convolution and obtain the components of Z (which are fewer in number).

1.1.2 Activation function

So far, the convolutional filter is just a localized affine transformation. Therefore, it is necessary to apply a non-linear activation function f to the elements of Z.

1.1.3 Pooling

Pooling performs a dimensionality reduction. There are several methods, but for example, consider a 3×3 grid. Divide Z into 3×3 regions, and for each region take the maximum value. this is known as *max-pooling*.

2 Data collection and manipulation

For this project, all images are obtained from a Kaggle dataset [3]. The data are organized as follows: a folder named `train` containing all the images labeled as `Image_n` with $n = 1, 2, \ldots,$ and a CSV file with one column with `filename` and `class` separated by a comma, where the class is written in letters.

First, the CSV file was read using the `pandas` [4] library. Then, only 10 classes were selected from the dataset. A script was implemented to automatically select only the images belonging to the chosen classes. In addition, a series of checks on file format and integrity were performed to ensure that the images could be properly loaded using TensorFlow[5] and Keras[6].

Images were also resized to a resolution of 100×100 pixels.

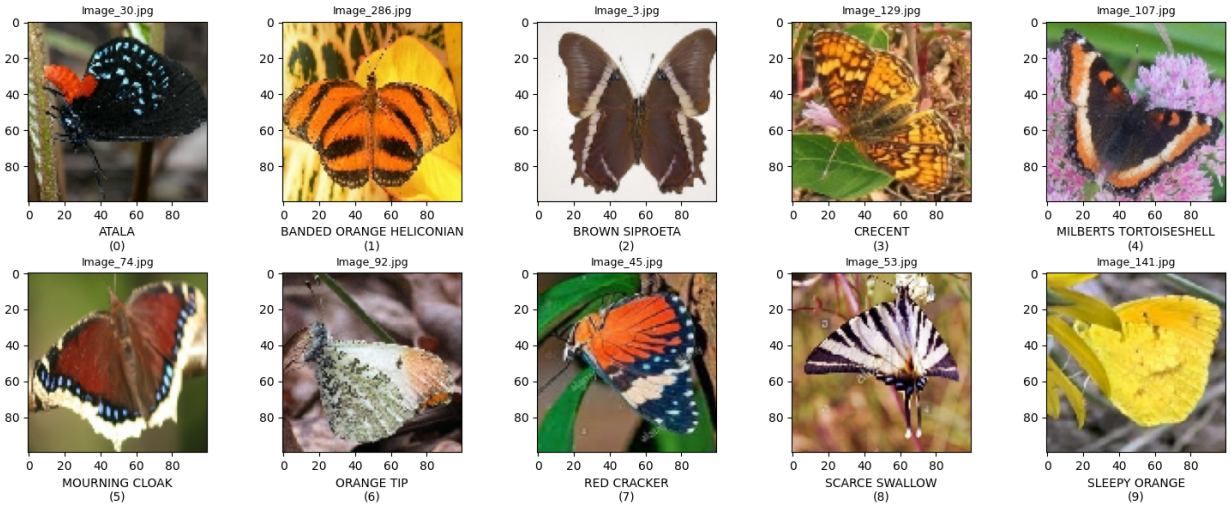


Figure 1: *Example images from the 10 selected butterfly classes. Each label shows the class name and the numeric label.*

2.1 Train, Validation and Test

After collecting and selecting the images, the dataset was split into smaller subsets to perform training and evaluate the model.

A split of 70% training, 15% validation, and 15% testing was chosen. The training and validation sets are used to fit the model and prevent overfitting, while the test set provides an unbiased assessment of the model's performance. This split was performed using the `train_test_split` function from Scikit-Learn [7], with the `stratify` option enabled to ensure a uniform sampling across all classes.

Dataset	**Number of Images**	**Image Size**	**Channels**
Training (X_{train})	711	100×100	3
Validation (X_{val})	152	100×100	3
Test (X_{test})	153	100×100	3

Table 1: *Dataset splits with image shapes and channels.*

2.2 Data Augmentation

The training dataset consists of 711 images, which is relatively small, corresponding to approximately 70 images per class. Training deep learning models on such small datasets can make it difficult for the model to extract meaningful features and may lead to poor convergence or overfitting. To address this issue and improve both convergence and generalization, data augmentation techniques were applied using TensorFlow. These techniques include rotations, flips, zooms, and translations, which effectively increase the diversity of the training data and help the model generalize better [8].

Parameter	**Value**	**Description**
rotation_range	15	Random rotation of images in degrees
width_shift_range	0.10	Random horizontal shift (fraction of total width)
height_shift_range	0.10	Random vertical shift (fraction of total height)
zoom_range	0.15	Random zoom in/out
horizontal_flip	True	Random horizontal flip
vertical_flip	True	Random vertical flip
fill_mode	'nearest'	Strategy for filling in newly created pixels

Table 2: *Data augmentation configuration using `ImageDataGenerator`[9].*

In this project, data augmentation was applied only to the training dataset, to keep the performance evaluation as faithful as possible to the original dataset and to avoid artificial transformations that introduce bias or distort the true estimate of the generalization capability of the model.

For each image in the training dataset, two new augmented images were generated.

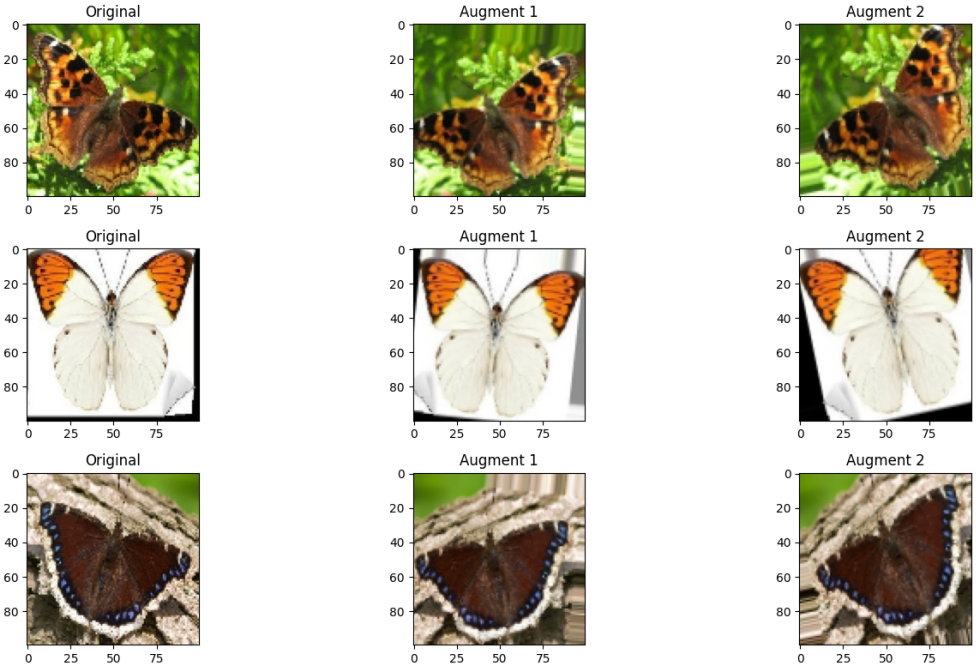


Figure 2: *Example of data augmentation. Each row shows the original image alongside two augmented versions.*

3 Model Creation and hyperparameters selection

This report illustrates how the performance of the constructed CNN varies with different hyperparameters. For this project, the hyperparameters chosen for variation were the dropout rate, the learning rate, and the lambda (L2 regularization constant). Performing a full grid search would have been too computationally demanding, given the available hardware. Therefore, a two-step optimization approach wass adopted: first, tuning the dropout rate, followed by a smaller grid search over the learning rate and the lambda.

First, a generic parametric function was implemented to build the CNN model, allowing the specification of hyperparameters.

The model was created using the Keras `Sequential` API [10][1]. It starts with an input layer, followed by three convolutional blocks, each composed of a `Conv2D` layer (with 32, 64, and 32 filters, respectively) and a `MaxPooling2D` layer with a pooling size of $(2, 2)$. After flattening the feature maps, a fully connected `Dense` layer with 64 units is added. The `Dropout` layer is then applied. Finally, the output layer is defined with a number of units equal to the number of classes, producing logits for classification.

Other model hyperparameters or architecture configurations are illustrated below.
For a complete description of the implementation, the code is provided in the appendix.

Hyperparameter / Configuration	Value
Optimizer	Adam
Learning rate	to be specified (default = 0.0001)
Loss function	SparseCategoricalCrossentropy
Metric	Accuracy
Batch size	8
Epochs	50
Early stopping	Yes
Monitor	Validation loss
Patience	5
Restore best weights	True

Table 3: *Training configuration and hyperparameters.*

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 98, 98, 32)	896
max_pooling2d (MaxPooling2D)	(None, 49, 49, 32)	0
conv2d_1 (Conv2D)	(None, 47, 47, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 23, 23, 64)	0
conv2d_2 (Conv2D)	(None, 21, 21, 32)	18,464
max_pooling2d_2 (MaxPooling2D)	(None, 10, 10, 32)	0
flatten (Flatten)	(None, 3200)	0
dense (Dense)	(None, 64)	204,864
dense_1 (Dense)	(None, 10)	650

Total params: 243,370 (950.66 KB)

Figure 3: *Model summary with hyperparameters set to their default values.*

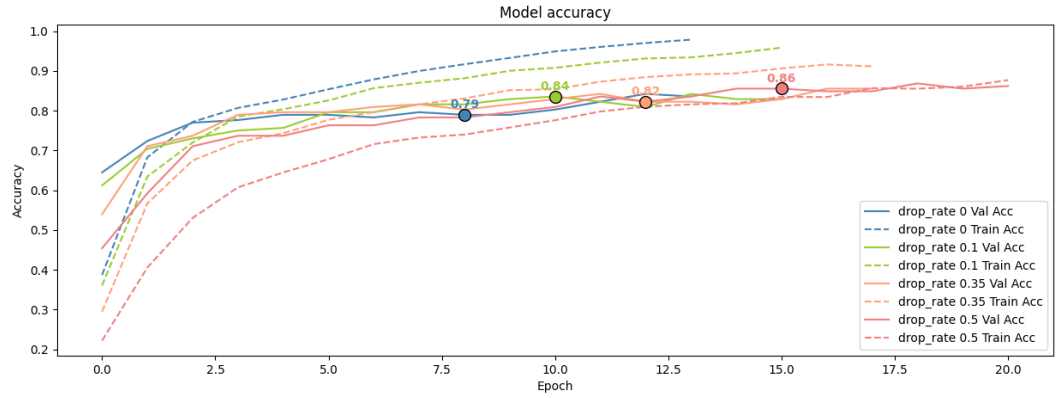
3.1 Dropout rate

Dropout[11] is a technique designed to reduce overfitting in deep neural networks with many parameters. The idea is to randomly drop a subset of neurons (and their connections) during each training iteration. This process prevents excessive co-adaptation of units and forces the network to learn more robust representations. Conceptually, dropout can be seen as training an ensemble of many "thinned" subnetworks and averaging their predictions. At test time, the full network is used with appropriately scaled weights.

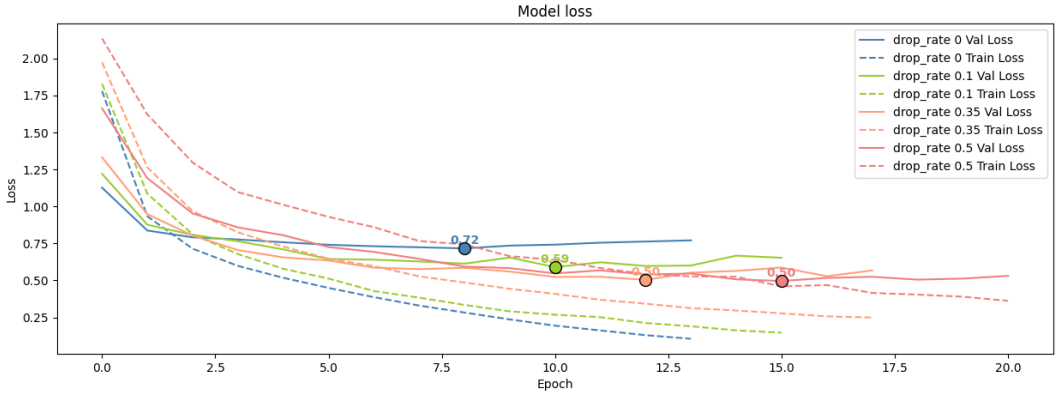
The percentage of neurons in a given layer that are randomly dropped during training is known as the *dropout rate*. In this project, four different values were chosen for comparison: `[0, 0.10, 0.35, 0.5]`. Four separate training runs were performed with identical models varying only the dropout rate. The model with the highest validation accuracy was then selected.

It is possible that at the end of training the validation accuracy appears higher than the training accuracy. This happens because training accuracy is computed with dropout enabled, while validation accuracy is evaluated using the full net. To obtain a fair estimate of the training accuracy one would need to re-evaluate the trained model on the training set after the training. However, this step is not necessary, since only the validation accuracy is used to select the best performing model.

Figures 4 (a) and (b) show a comparison of model accuracy and loss during training respectively, in function og epochs.



(a) Model accuracy



(b) Model loss

Figure 4: *Comparison of accuracy (a) and loss (b) for models with different dropout rates. Dashed lines indicate training values, solid lines indicate validation values. Values at best epochs are highlighted with a marker and annotated with the corresponding value.*

3.2 Learning rate and kernel regularizer

The other hyperparameters chosen for this project are the *learning rate* and the *kernel regularizer*. The learning rate (η) controls the step size in gradient descent updates: a small value leads to very slow convergence, while a large value can cause oscillations or even divergence from the optimal solution [12]. In the Adam optimizer expression for updating each parameter θ_j, the learning rate appears as η_t [12]:

$$\theta_{t+1,j} = \theta_{t,j} - \eta_t \frac{\hat{m}_{t,j}}{\sqrt{\hat{s}_{t,j}} + \epsilon}, \tag{1}$$

where $\hat{m}_t$ and $\hat{s}_t$ are bias-corrected estimates of the first and second moments of the gradient, calculated using β_1 and β_2 hyperparameters ($\beta_1 = 0.9$, $\beta_2 = 0.999$) The ϵ is a small constant to prevent division by zero ($\epsilon = 10^{-7}$) [13].

Kernel regularizers are often employed to reduce overfitting. The $\mathcal{L}_2$ regularizer adds a penalty term to the loss function proportional to the squared magnitude of the weights [14][15]. This encourages the network to keep weights small, improving generalization. In this project this is implemented in Keras through the parameter `kernel_regularizer=regularizers.l2(`λ`)`, where λ controls the strength of the penalty.

A grid search was performed to evaluate the best combination of learning rate and regularization factor λ. The values explored in the search were:

$$lr\_val = \{10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}, \quad \lambda\_val = \{0, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}\}.$$

After training, the models were evaluated on both the training and validation datasets. To better visualize the results, the accuracies were plotted using heatmaps generated with `matplotlib`.

Learning rate (lr)	**L2** (λ)	**Validation Accuracy**
$1e{-}5$	0	0.74342
$1e{-}5$	$1e{-}5$	0.76316
$1e{-}5$	$1e{-}4$	0.77632
$1e{-}5$	$1e{-}3$	0.75658
$1e{-}5$	$1e{-}2$	0.73684
$1e{-}4$	0	0.83553
$1e{-}4$	$1e{-}5$	0.87500
$1e{-}4$	$1e{-}4$	0.82237
$1e{-}4$	$1e{-}3$	0.86184
$1e{-}4$	$1e{-}2$	**0.88158**
$1e{-}3$	0	0.82895
$1e{-}3$	$1e{-}5$	0.84211
$1e{-}3$	$1e{-}4$	0.86184
$1e{-}3$	$1e{-}3$	0.85526
$1e{-}3$	$1e{-}2$	0.81579
$1e{-}2$	0	0.13158
$1e{-}2$	$1e{-}5$	0.13158
$1e{-}2$	$1e{-}4$	0.13158
$1e{-}2$	$1e{-}3$	0.13158
$1e{-}2$	$1e{-}2$	0.13158

Table 4: *Model accuracy evaluated on validation set.*

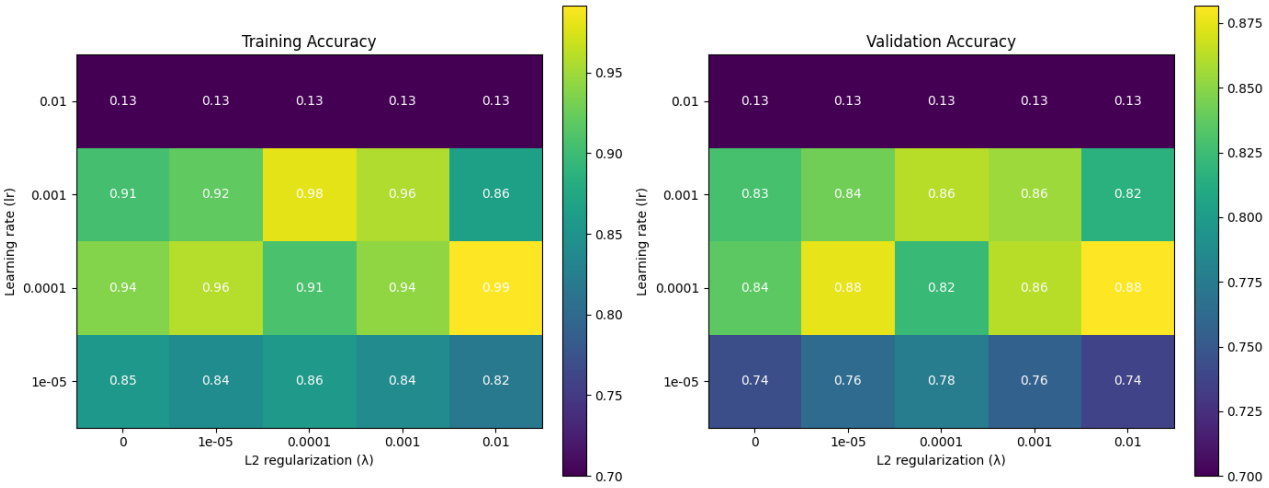


Figure 5: *Model Accuracy evaluated on train set (left) and validation set (right).*

From both the heatmap visualization and the tabulated results it is clear that the combination $lr = 1 \times 10^{-4}$ and $\lambda = 1 \times 10^{-2}$ achieves the highest validation accuracy (0.88158). This configuration is therefore selected as the best performing model.

4 Results

The performance of the final model is evaluated on **test dataset**, that the model had never seen during models training or best performing model selection. The following metrics are reported [16]:

- **Accuracy**: the proportion of all correct predictions. It describes how the model performs across all classes

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

- **Precision**: the ratio of true positives to all positive predictions. The precision reflects how reliably the model is in classifying samples as positive

$$Precision = \frac{TP}{(TP + FP)}$$

- **Recall**: the ratio of true positives to all actual positive examples. The recall measures the model's ability to detect Positive samples

$$Recall = \frac{TP}{(TP + FN)}$$

- **F1-score**: the harmonic mean of precision and recall. It provides a single metric that balances both, being high only when both precision and recall are high [17]

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

where TP, FP, TN, and FN refer respectively to the number of *True Positives*, *False Positives*, *True Negatives*, and *False Negatives*.

Being this project a *multi-class classification* these definitions are generalized using a one-vs-rest approach. For each class C_i, the class is considered as the "positive" class, while all other classes are treated as "negative."

Metrics are computed for each class individually, then the results are summerized using two types of averages [18]:

- **Macro-average**: the arithmetic mean of the per-class metrics, treating all classes equally regardless of their size. This is particularly useful for evaluating performance enhancing behaviour on minor classes.
- **Weighted-average**: the mean of the per-class metrics weighted by the number of examples in each class (*support*). It is useful as an overall performance indicator.

The results are summerize in Tabel 5.

A **confusion matrix** is also presented in Fig 6 to analyze classification performance per class, revealing patterns of misclassification and helping to identify which classes are often confused.

Class	Precision	Recall	F1-score	Support
0	1.000	0.800	0.889	15
1	0.778	0.933	0.848	15
2	0.875	0.933	0.903	15
3	1.000	0.867	0.929	15
4	0.923	0.857	0.889	14
5	0.769	1.000	0.870	20
6	1.000	1.000	1.000	14
7	1.000	0.643	0.783	14
8	0.938	1.000	0.968	15
9	1.000	1.000	1.000	16
Accuracy			0.908	153
Macro avg	0.928	0.903	0.908	153
Weighted avg	0.923	0.908	0.907	153

Table 5: *Metrics for each class and overall averages.*

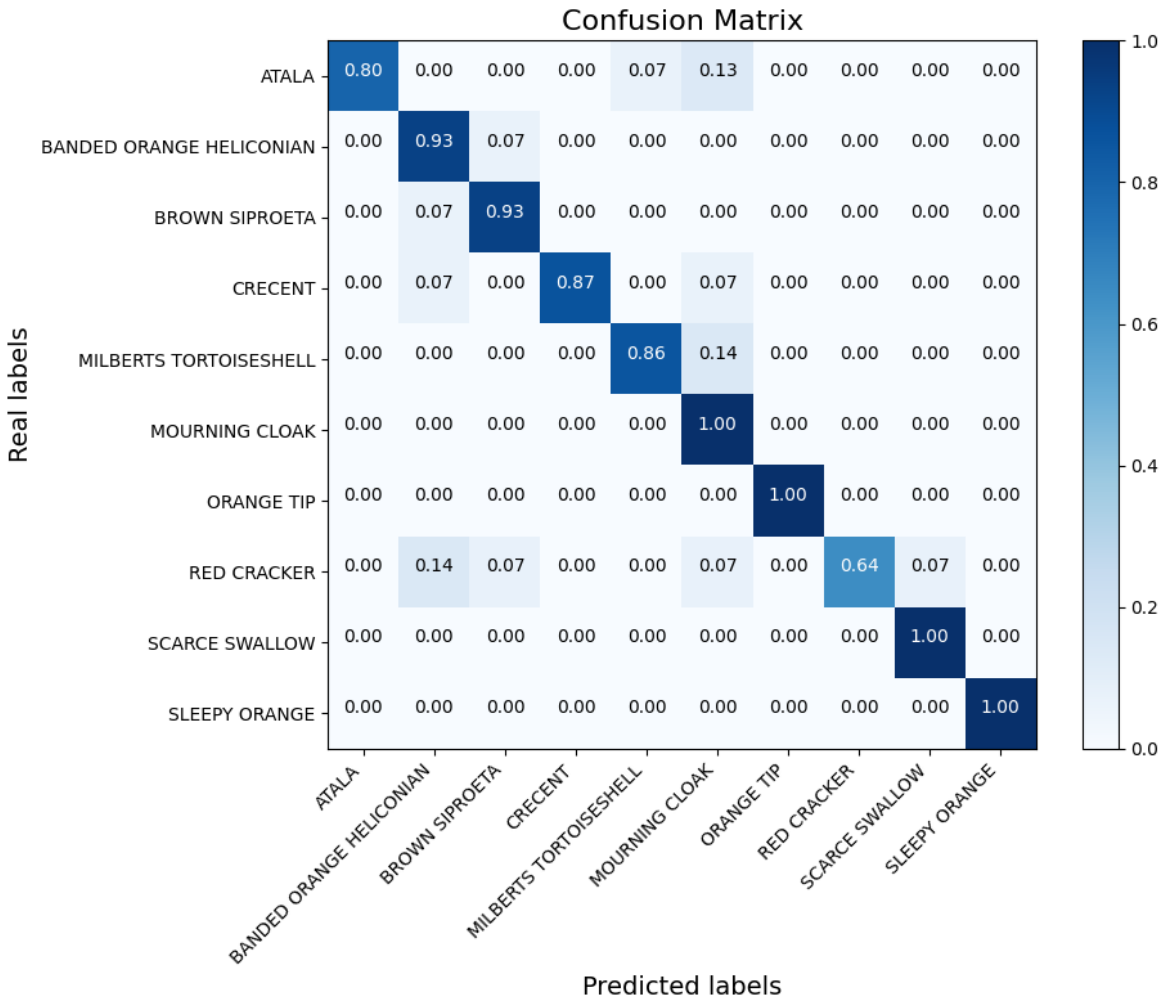
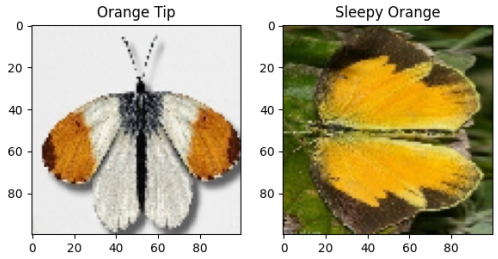


Figure 6: *Confusion Matrix with species names.*

From Table 5 and Fig 6 is clear that the CNN with the final hyperparameters performs reasonably well across all classes achieving an accuracy of $\approx 90\%$. Species with big uniform color ares are classified most accurately, such as the *Sleepy Orange*, with its uniform yellow color and brown borders, and the *Orange Tip*, with its uniform white wings with orange tips. The most frequently misclassified class is the *Red Cracker* with blue wings from top view and half orange wing from bottom view, which is predicted 14% of the time as a *Banded Orange Heliconian*, and 7% of the time as a *Brown Siproeta*, *Mourning Cloak*, or *Scarce Swallow*.



(a) Best classified species side by side.

(b) Worst classified species (left) and species it is confused with (right).

Figure 7: *Visualization of best and worst classified species.*

5 Conclusions

In this project, a CNN was implemented and evaluated for the multiclass classification of different species of butterfly images by varying three iperparameters: dropout rate, learning rate, and L2 regularization. The study involved three main phases: data collection and manipulation, model creation and hyperparameter selection, and performance evaluation.

10 butterfly species were selected and all images were resized to 100×100 pixels. To augment the number of images per class, data augmentation techniques were applied only to training set — including rotations, flips, translations, and zooms.

The CNN architecture consisted of three convolutional blocks followed by a fully connected layer, a dropout layer, and an output layer. A parametric approach was followed. Dropout rates between 0 and 0.5 were tested. A grid search over learning rates and L2 regularization values is then performed. The best combination of these three hyperparameters is:

- $dropout\ rate = 0.5$
- $lr = 1 \times 10^{-4}$
- $\lambda = 1 \times 10^{-2}$

The test set accuracy was approximately 90%. This is a good result, especially considering that the training set contained only about 70 images per class before data augmentation. The best-classified images were those without complex patterns and with large uniform color regions. The worst-classified classes had similar, complex patterns and often displayed varying colors across different images depending on the viewpoint.

Further investigations recommended, like a finer search around the optimal hyperparameters or if hardware allows, a large-scale grid search on all hyperparameters. Alternatively random search could be explored.

Other directions initially considered but later postponed—and which could be explored in future work—include modifying the number of layers, the type of optimizer, or the convolutional filter kernel sizes.

A Code

Including all the code would result in an appendix longer than the report itself. Since the full code and notebook are available on GitHub at the following GitHub link: github.com/Stefano-Bozzi/LepidopterAI, only the main parts are shown here, omitting comments, imports, array conversions, and similar steps.

A.1 Data collection and manipulation

A.1.1 image load

```
folder_path = "./archive/"
# Check if the image format is valid and verify if TensorFlow can load the image,
# otherwise mark it as invalid.

images_path = "train"
images_folder_path = os.path.join(folder_path, images_path)
invalid_images = []
dataset = []
target_size = (100, 100)

# I can check all extension supported from PIL
supported_exts = tuple(ext.lower() for ext in Image.registered_extensions().keys())

for filename in df['filename']:
    file_path = os.path.join(images_folder_path, filename)
    print(filename)
    if os.path.isfile(file_path):

        # Check if the file extension is a supported image format
        if not filename.lower().endswith(supported_exts):
            print(f"Invalid Extension for {filename}.")
            invalid_images.append(filename)
            continue

        # Try loading and preprocessing the image
        try:
            img = tf.keras.preprocessing.image.load_img(file_path,
            target_size=target_size)
            numeric_img = tf.keras.preprocessing.image.img_to_array(img)
            numeric_img = numeric_img / 255.
            dataset.append(numeric_img)
        except Exception as e:
            print(f"Error loading {filename}: {e}.")
            invalid_images.append(filename)
    else:
        print(f"Skipping {filename}, not a file.")
print()
print("images checked")
dataset_np = np.array(dataset)
```

```
print()
print("Invalid images list:", invalid_images)

# Update the df with valid images only
df = df[~df['filename'].isin(invalid_images)].reset_index(drop=True)

# Create array for labels
labels = df['label'].values
# change from string label to numeric label
le = LabelEncoder()
labels_encoded = le.fit_transform(labels)
```

A.1.2 train val test split

```
X_temp, X_test, y_temp, y_test = train_test_split(
    dataset_np, labels_encoded, test_size=0.15,
    random_state=seed_value, stratify=labels_encoded)

# here test size is to ensure that is 15% of total, so 17.6% of 85% of total
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.176, random_state=seed_value, stratify=y_temp)
```

A.1.3 data augmentation

```
# augmentation confiug
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.10,
    height_shift_range=0.10,
    zoom_range=0.15,
    horizontal_flip=True,
    vertical_flip=True,
    fill_mode='nearest'
)

# new images for each existing one
n_copies = 2

X_train_aug = []
y_train_aug = []

for i in range(len(X_train)):
    img = X_train[i]
    label = y_train[i]

    # expansion to shape (1, H, W, C)
    img_expanded = np.expand_dims(img, axis=0)

    aug_iter = datagen.flow(img_expanded, batch_size=1, seed=seed_value)

    for _ in range(n_copies):
```

```
        aug_img = next(aug_iter)[0]
        X_train_aug.append(aug_img)
        y_train_aug.append(label)

# new X,y _train
X_train = np.concatenate([X_train, np.array(X_train_aug)], axis=0)
y_train = np.concatenate([y_train, np.array(y_train_aug)], axis=0)
```

A.2 Model creation and hyperparameters selection

A.2.1 generic parametric function

```
input_shape = (target_size[0], target_size[1], 3)
number_of_classes = len(selected_species)

def build_cnn_model(
    input_shape,
    number_of_classes,
    kernel_size=(3, 3),
    learning_rate=0.0001,
    l2_reg=0.0,
    dropout_rate=0.0
):
    model = models.Sequential()

    model.add(layers.Input(shape=input_shape))

    model.add(layers.Conv2D(32, kernel_size, activation='relu',
    kernel_regularizer=regularizers.l2(l2_reg)))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(64, kernel_size, activation='relu',
    kernel_regularizer=regularizers.l2(l2_reg)))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Conv2D(32, kernel_size, activation='relu',
    kernel_regularizer=regularizers.l2(l2_reg)))
    model.add(layers.MaxPooling2D((2, 2)))

    model.add(layers.Flatten())

    model.add(layers.Dense(64, activation='relu',
    kernel_regularizer=regularizers.l2(l2_reg)))

    if dropout_rate > 0:
        model.add(layers.Dropout(dropout_rate))

    model.add(layers.Dense(number_of_classes))

    optimizer = optimizers.Adam(learning_rate=learning_rate)
    model.compile(
```

```
        optimizer=optimizer,
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=['accuracy']
    )
    return model
```

A.2.2 dropout rates

```
batch_size=8
dropout_rates = [0,0.10,0.35,0.5]
num_try = len(dropout_rates)
CNN_s_drop = [None] * num_try
history_drop = [None] * num_try

epochs = 50
best_val_acc = 0.

for i in range(num_try):
    start_time = time.time()

    # create model
    CNN_s_drop[i] = build_cnn_model(input_shape, number_of_classes,
    dropout_rate=dropout_rates[i])
    CNN_s_drop[i] .summary()

    # train
    early_stop = EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True
    )
    history_drop[i] = CNN_s_drop[i].fit(
        X_train, y_train,
        epochs=epochs,
        validation_data=(X_val, y_val),
        callbacks=[early_stop],
        batch_size=batch_size,
        verbose=1
    )
```

A.2.3 grid search

```
lr_val = [1e-5, 1e-4, 1e-3, 1e-2]
lmd_val = [0, 1e-5, 1e-4, 1e-3, 1e-2]

num_try_lr = len(lr_val)
num_try_l2 = len(lmd_val)
CNN_s = [[None] * num_try_l2 for _ in range(num_try_lr)]
history = [[None] * num_try_l2 for _ in range(num_try_lr)]

epochs = 50
```

```
best_val_acc = 0.

for i in range(num_try_lr):
    for j in range(num_try_l2):
        start_time = time.time()
        # create model
        CNN_s[i][j] = build_cnn_model(input_shape, number_of_classes,
        dropout_rate=best_drop, learning_rate=lr_val[i], l2_reg=lmd_val[j])
        CNN_s[i][j] .summary()
        # train
        early_stop = EarlyStopping(
            monitor='val_loss',
            patience=5,
            restore_best_weights=True
        )
        history[i][j] = CNN_s[i][j].fit(
            X_train, y_train,
            epochs=epochs,
            validation_data=(X_val, y_val),
            callbacks=[early_stop],
            batch_size=batch_size,
            verbose=1
        )
```

A.3 Results and metrics

```
# Evaluate model
for i in range(num_try_lr):
    for j in range(num_try_l2):
        # best ephoc
        best_epoch = history[i][j].history['val_loss'].index(
            min(history[i][j].history['val_loss'])
        )
        # --- Evaluation ---
        train_eval = CNN_s[i][j].evaluate(X_train, y_train, verbose=0)
        val_eval = CNN_s[i][j].evaluate(X_val, y_val, verbose=0)

        print(
            f"   Evaluated -> Train acc={train_eval[1]:.5f}, "
            f"Val acc={val_eval[1]:.5f}\n"
        )

# results on testset

# 1. predictions
CNN_best = CNN_s[best_i][best_j]
y_pred_prob = CNN_best.predict(X_test)
y_pred = np.argmax(y_pred_prob, axis=1)

# 2. True labels
```

```
y_true = y_test

# 3. Classification report
print(classification_report(y_true, y_pred, digits=3))

# 5. Accuracy on test set
test_loss, test_acc = CNN_best.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_acc:.3f}")
```

References

[1] Reticonvoluzionali_e_ricorrenti. https://gitlab.com/cbarbieri/MachineLearning-con-applicazioni/-/blob/master/Lab%20Computazionale/Sessione3-RetiNeurali/RetiConvoluzionali_e_ricorrenti.ipynb?ref_type=heads.

[2] Notes10_neuralnetsii_advacedarchitectures. https://gitlab.com/cbarbieri/MachineLearning-con-applicazioni/-/blob/master/Lezioni%20(PDF)/Notes10_NeuralNetsII_AdvacedArchitectures%20(provvisorio).pdf?ref_type=heads.

[3] Butterfly image classification dataset. https://www.kaggle.com/datasets/phucthaiv02/butterfly-image-classification.

[4] The pandas development team. pandas-dev/pandas: Pandas. https://doi.org/10.5281/zenodo.3509134, 2020. Zenodo. doi:10.5281/zenodo.3509134.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] François Chollet et al. Keras. https://keras.io, 2015.

[7] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[8] data augmentation, tensorflow. https://www.tensorflow.org/tutorials/images/data_augmentation?hl=it.

[9] Imagedatagenerator, tensorflow. https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator.

[10] keras sequential, tensorflow. https://www.tensorflow.org/api_docs/python/tf/keras/Sequential.

[11] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 06 2014.

[12] Notes08_allenamento. https://gitlab.com/cbarbieri/MachineLearning-con-applicazioni/-/blob/master/Lezioni%20(PDF)/Notes08_Allenamento.pdf?ref_type=heads.

[13] Adam, tensorflow. https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam.

[14] L2 regularization, keras. https://keras.io/api/layers/regularizers/.

[15] L2 regularization, tensorflow. https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/L2.

[16] How to evaluate deep learning models: Key metrics explained. https://www.digitalocean.com/community/tutorials/deep-learning-metrics-precision-recall-accuracy.

[17] F1 score. https://it.wikipedia.org/wiki/F1_score.

[18] Weighted and macro averaging. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html.