Weather Classification Project Report

Student Name: Edoardo Scarpel
Number ID: 2089096
January 25, 2024

Università degli Studi di Padova

Contents

Table of Contents

Chapter 1

Introduction

1.1 Project Description

In this report, we take a look at the classification of weather images extracted from well-known datasets: the Meteorological Watch Dog (MWD), Atmospheric Circulation Data Centre (ACDC), and SynDrone dataset. The aim is to employ advanced deep learning techniques, specifically using Python and PyTorch, to develop and compare the performance of two distinct convolutional neural network (CNN) architectures: a simple CNN with 4 layers and a Residual Network (ResNet).

The use of deep learning models for weather classification has gained momentum due to their ability to automatically learn and recognize features in images. For this project Python, Pytorch and OpenCV have been used, with these tools the aim is to not only classify weather images accurately but also shed light on the comparative effectiveness of a basic CNN architecture against the more sophisticated ResNet.

The approach involves the preprocessing of the dataset, the design and implementation of two distinct CNN architectures, and an in-depth evaluation of their performance. The simplicity of the basic CNN and the depth of the ResNet are chosen deliberately to assess the trade-off between model complexity and accuracy. Through this comparative study, the aim is to offer insights into the suitability of each architecture for weather image classification tasks.

This report is structured to provide a comprehensive overview of the methodology used, starting from the code description, then following with the description of every CNN architecture used and in the end showing the results of this study.

Chapter 2

Brief code description

2.1 Libraries description

For simplicity in this project, a Jupyter Notebook has been used since this tool is much more helpful in showing the results of each step of the code.

Several external libraries have been used to develop the code, the most important are OpenCV and PyTorch to develop the core of the code: the first one has been used to read the images from the datasets and then make some preprocessing to properly feed the CNNs; the second one instead has been used to develop and train the two CNN architectures and to create the appropriate data structures used in training, validation and test phases. Other libraries such as pandas, scikit-learn, matplotlib, and seaborn have been used to manage the data and to show all the results more elegantly.

In addition to the external libraries, the code incorporates two key files: 'utils.py' and 'models.py'. Within 'utils.py', a set of functions are designed to simplify the code and enhance its speed. Meanwhile, 'models.py' houses the classes for the two CNNs used in this study, along with essential functions such as 'train', 'valid', and 'test' for evaluating performance at various stages. These files play crucial roles in supporting the functionality and efficiency of the weather image classification framework.

2.2 Code sections description

Here a brief review of the sections encountered in the Jupyter Notebook to have an overview of the code. An in-depth view of the core parts is offered in the next chapters.

Taking a look at the Jupyter Notebook the first section regards data extraction. In this section, the paths of the images are read and the labels from the folders or the file names are extracted. Then a bunch of images as examples are shown. In the end, the images are extracted using the list of paths, resized to a dimension of (256, 256), and each channel is normalized to a range of 0-1 instead of 0-256 to ensure speed and stability in the training process.

In the second section, the datasets containing the images and labels are split into training and validation datasets in 80% and 20% of the original datasets respectively. All the datasets are then transformed into tensors. In the end, Dataset and Dataloader objects are created from the latter. Hyperparameters for the simple CNN are also defined.

Passing to the training and validation phase of the simple CNN, a ConvolutionalModel object is created and initialized with the proper datasets and hyperparameters. Then, a 100 epochs training phase is run and the train and validation functions from the 'models.py' file are used to train and validate, respectively, the model and keep track of the performance of the training phase. Plots of the training and validation evolutions are displayed. In the end, a comparison between prediction and real label is proposed to have a look at the performance of the trained net.

Then, similarly to the section before, the training and validation phase is run using a ResNet. Also in this case a ResNet object is initialized and a 100 epochs training phase is run. The initial parameters are fixed specifically for the net in question. As before, plots and results of this phase are displayed to have a look at the goodness of the model.

In the last section, the test dataset is used to test both nets. Images and labels are extracted, resized, normalized, and then converted into Dataloader as done in the above sections. An accuracy test and a comparison between prediction and real label are proposed to have a look at the performance of the trained net for both nets.

2.3 External functions description

Let's take a closer look at a set of functions embedded in the 'utils.py' and 'models.py' files. In this section a description of the functions embedded in the project is proposed.

```
def show_images_with_text(images: list, labels: list = None, title: str = None,
↪  figsize: tuple = None)
```

This function is a concise tool for displaying a grid of images from a given list along with their corresponding labels. It accepts parameters such as the list of images (‘images‘), optional labels (‘labels‘), an optional title (‘title‘), and an optional figure size (‘figsize‘). The images are arranged in a grid, and if labels are provided, they are displayed below each image. This function is handy for visualizing images and associated information in a clear and concise manner.

```
def plot_img(image: np.ndarray, title: str = "Image")
```

This function quickly plots an image with an optional title. It takes an image as a NumPy array (‘image‘) and converts it from BGR to RGB format for proper display. The resulting image is then shown with a specified title, and the function is designed for swift visualization of images.

```
def random_color_generator()
```

This function generates a random color in hexadecimal form.

```
def show_plot(signals: list, title: str, x_label: str, y_label: str, x_range: list =
↪  None, y_range: list = None)
```

This function efficiently plots a graph of signals with specified characteristics. It takes a list of signals (‘signals‘), a title (‘title‘), and labels for the x and y axes (‘x_label‘ and ‘y_label‘). Optional parameters include custom ranges for the x and y axes (‘x_range‘ and ‘y_range‘). The function uses the Seaborn library to create a line plot, with each signal uniquely colored and labeled.

```
def compute_mean_std(loader: DataLoader) -> tuple
```

This function efficiently calculates the mean and standard deviation from a DataLoader containing a list of images. It takes the DataLoader (‘loader‘) as input, extracts a batch of images, and computes the mean and standard deviation across different dimensions. The resulting tuple contains the mean and standard deviation values.

```
def read_images(path: str) -> list
```

This function reads images from a specified path and returns a list of image-label pairs. It navigates through the directory structure, extracting labels and corresponding image file paths. The resulting list, containing tuples of labels and image paths, serves as a convenient representation of the image data.

```
def train(train_loader: DataLoader, model: nn.Module, epoch: int, criterion:
↪  nn.modules.loss, optimizer: torch.optim, device, pbar: tqdm = None) -> (list, list)
```

This function is designed for training a PyTorch model using a specified training dataset. The key parameters include the training data loader (‘train_loader‘), the neural network model (‘model‘), the current epoch number (‘epoch‘), the loss function (‘criterion‘), the optimizer (‘optimizer‘), and the device on which the model should be loaded (‘device‘). Optionally, a progress bar (‘pbar‘) using ‘tqdm‘ can be provided for visual tracking of the training progress.

Within the function, the model is set to training mode (‘model.train()‘), and the training loop iterates over batches from the training loader. For each batch, the input images and corresponding labels are processed. The function handles GPU casting, forward pass, computation of the loss, and backpropagation steps. Additionally, it keeps track of the epoch-wise loss, predictions, and ground truth labels.

At the end of each epoch, the function prints a summary of the training performance, including the average loss, its standard deviation, accuracy, and the time taken for the epoch. The final results, such as the mean loss and accuracy, are returned as a tuple. This function serves as a fundamental component in the training process, providing insights into model performance during the training phase.

```
def validate(validation_loader: DataLoader, model: nn.Module, epoch: int, criterion:
↪  nn.modules.loss, device, pbar: tqdm = None) -> (list, list)
```

The ‘validate‘ function assesses a PyTorch model’s performance on a validation dataset. It takes parameters such as the validation data loader (‘validation_loader‘), the neural network model (‘model‘), the current epoch number (‘epoch‘), the loss function (‘criterion‘), the device on which the model should be loaded (‘device‘), and an optional progress bar (‘pbar‘). Operating in evaluation mode (‘model.eval()‘), it computes and reports the validation loss, accuracy, and time taken. This function is necessary for evaluating model performance during the validation phase.

```
def test(test_loader: DataLoader, model: nn.Module, criterion: nn.modules.loss, device)
↪  -> (list, list)
```

The ‘test‘ function evaluates a PyTorch model on a provided test dataset. It takes parameters including the test data loader (‘test_loader‘), the neural network model (‘model‘), and the loss function (‘criterion‘). Operating in evaluation mode (‘model.eval()‘), it computes and reports the test loss, accuracy, and the time taken for the evaluation. This function is instrumental for gauging the model’s performance on unseen data during the testing phase.

2.4 External classes description

Let’s take a closer look at a set of classes embedded in the ’utils.py’ and ’models.py’ files. In this section a description of the classes embedded in the project is proposed.

```
class ImagesLabelsDataset(Dataset):
    def __init__(self, images_array, labels_array, transforms=None)
    def __getitem__(self, idx)
    def __len__(self)
```

The ‘ImagesLabelsDataset‘ class defines a custom PyTorch dataset for efficiently loading images and labels from provided arrays. It inherits from the PyTorch ‘Dataset‘ class. The constructor (‘__init__‘) takes in arrays of images (‘images_array‘) and corresponding labels (‘labels_array‘), with an optional parameter for image transformations (‘transforms‘). The class implements methods to retrieve a specific item (‘__getitem__‘) and obtain the dataset’s length (‘__len__‘). This class provides a convenient and customizable interface for handling image and label data within the PyTorch ecosystem.

```
class EarlyStopper:
    def __init__(self, patience=1, min_delta=0)
    def early_stop(self, validation_loss)
```

The ‘EarlyStopper‘ class implements early stopping in model training based on validation loss. Initialized with parameters like ‘patience‘ (epochs with no improvement before stopping) and

‘min_delta‘ (minimum change in validation loss), it tracks epochs without improvement and the minimum validation loss.

The ‘early_stop‘ method checks if the current validation loss exceeds the minimum by more than ‘min_delta‘. If so, it resets the counter and updates the reference loss. If not, it increments the counter, triggering early stopping if patience is reached. This class helps prevent overfitting by stopping training when validation performance stabilizes.

```
class ConvolutionalModel(nn.Module):
    def __init__(self, output_size, dropout=0.32)
    def forward(self, x)
```

The ‘ConvolutionalModel‘ class defines a straightforward Convolutional Neural Network (CNN) for image classification, employing four convolutional layers. Initialized with the desired output size and an optional dropout rate, this PyTorch model inherits from ‘nn.Module‘.

The architecture comprises four convolutional blocks (‘ConvBlock‘), each consisting of a 2D convolutional layer, batch normalization, ReLU activation, and max-pooling. The model concludes with a dropout layer, a dense block (‘DenseBlock‘) with two linear layers and ReLU activations, ultimately leading to the output layer.

The ‘forward‘ method processes input data through the defined architecture, producing the final output. This ‘ConvolutionalModel‘ serves as a concise and adaptable tool for image classification tasks, providing a balance between simplicity and performance.

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None)
    def forward(self, x)

class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes)
    def _make_layer(self, block, planes, blocks, stride=1)
    def forward(self, x)
```

The ‘ResidualBlock‘ class defines a residual block for a Residual Neural Network (ResNet). It consists of two convolutional blocks (‘ConvBlock‘). The first block (‘conv1‘) includes a convolutional layer, batch normalization, and ReLU activation, while the second block (‘conv2‘) consists of a convolutional layer and batch normalization. The block also supports downsampling, and the residual connection is added to the output after the second convolutional block.

The ‘ResNet‘ class is the overarching architecture of a ResNet model. It utilizes the ‘ResidualBlock‘ class to construct a deep neural network. The model includes an initial convolutional layer, followed by four layers (‘layer0‘ to ‘layer3‘), each comprising multiple residual blocks. The number of blocks in each layer is specified by the ‘layers‘ parameter. The model concludes with average pooling, flattening, and a fully connected layer (‘fc‘) for classification.

This ResNet serves as a more solid model to compare to the simpler CNN above described.

Chapter 3

Results using a simple CNN

3.1 Qualitative analysis of the model

For this project has been decided to use at first a simple convolutional neural network composed of four convolutional layers, then compare it to a more sophisticated one like ResNet and see what are the differences in terms of performance. From a theoretical point of view the pros of using a simpler architecture could be:

- Simpler models are computationally efficient, accelerating both training and inference.
- Simplicity enhances interpretability, facilitating model understanding and analysis.
- Simpler models are less prone to overfitting, offering better generalization.

While the cons could be:

- Simpler models may struggle with intricate patterns and varied datasets.
- Simpler models may not be adequate for highly complex or specialized image classification tasks.

The ‘ConvolutionalModel‘ class consists of four convolutional layers grouped into distinct blocks:

- ConvBlock 1:
  - Input: 3 channels (RGB images)
  - Output: 6 channels
  - Convolution: Kernel size of 4x4, ReLU activation, Batch Normalization
  - MaxPooling: Kernel size of 5x5
- ConvBlock 2:
  - Input: 6 channels

  - Output: 16 channels
  - Convolution: Kernel size of 5x5, ReLU activation, Batch Normalization
  - MaxPooling: Kernel size of 2x2

- ConvBlock 3:
  - Input: 16 channels
  - Output: 32 channels
  - Convolution: Kernel size of 8x8, ReLU activation, Batch Normalization
  - MaxPooling: Kernel size of 4x4

- ConvBlock 4:
  - Input: 32 channels
  - Output: 120 channels
  - Convolution: Kernel size of 4x4, ReLU activation, Batch Normalization
  - Flatten operation followed by dropout with rate specified by ‘dropout‘ parameter

- DenseBlock:
  - Fully connected layer: Input size of 120, output size of 84 with ReLU activation
  - Fully connected layer: Input size of 84, output size determined by ‘output_size‘

The use of Batch Normalization, ReLU activation, and MaxPooling contributes to the network's learning capacity. The presence of dropout aids in preventing overfitting during training.

3.2 Results of training, validation and test phase

The training and validation phase has been done using different configurations of hyperparameters and using the three datasets 'MWD', 'ACDC' and 'SynDrone'.

3.2.1 MWD Dataset

Starting from the MWD dataset, the training phase has been done using Cross-entropy as a loss function and Adam Optimization Algorithm to update net weights. The hyperparameters used are the following:

- Number of epochs: 100
- Learning rate: 10^{-4}
- Weight decay factor (L2-penalty): 10^{-4}
- Dropout: 0.4

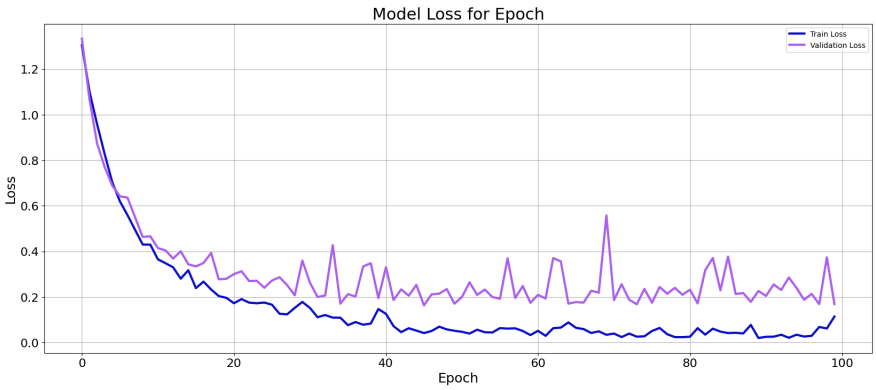

Figure 3.1: MWD Model Loss for Epoch - 32 batch size

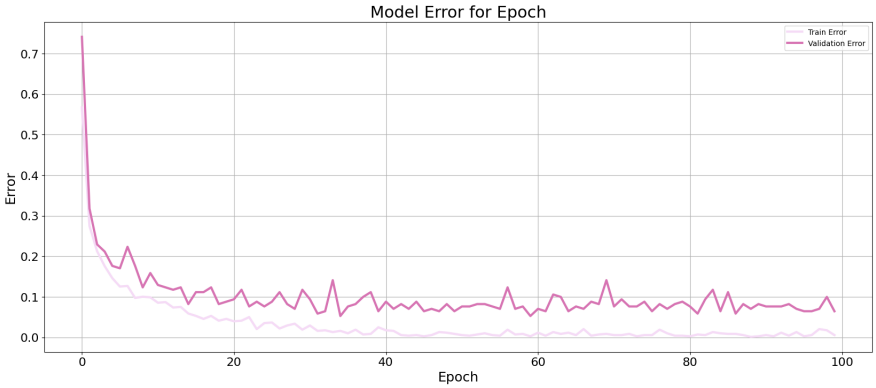

Figure 3.2: MWD Model Error for Epoch - 32 batch size

Starting from the case of a batch size equal to 32, the results are depicted in Fig.[3.1], Fig.[3.2] and Fig.[3.3]. The training and validation phase ended up with an accuracy of 100% and a loss of 0.022 ± 0.023 for the training set, and an accuracy of 92% and a loss of 0.183 ± 0.102 for the validation set.

From the loss and error plots depicted in Fig.[3.1] and Fig.[3.2] we can clearly see an improvement in the training and validation of the model, but we also notice a lot of oscillations in the evolution of the validation loss and error, this can be due to overfitting. To overcome this problem the learning rate can be lowered or the dropout factor can be increased, but by doing this it was not observed an increase in the performance. The oscillation problem sometimes can be attributed not only to overfitting but also to the case in which the validation set is too small and/or the batch size is too small.

In the end, applying the trained model to the test dataset has been observed an overall accuracy of 94% with an average loss of about 0.192 ± 0.248. From Fig.[3.3] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model struggles a little in recognizing the sixth image as 'shine' because it is fooled by the clouds and recognizes the image as 'cloudy'. This can be due to overfitting since the model isn't able to generalize.

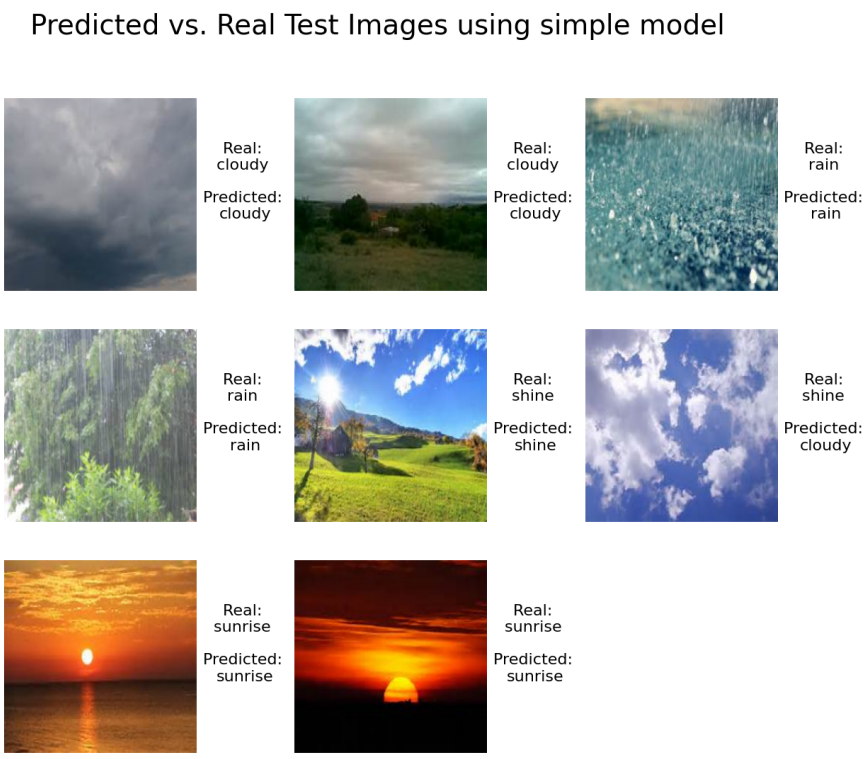

Figure 3.3: MWD Predicted vs. Real Test Images - 32 batch size

Trying to minimize the noise in the validation loss and error evolution the batch size has been increased to 64 and another test has been done with the same configuration and hyperparameters as before. The results are depicted in Fig.[3.4], Fig.[3.5] and Fig.[3.6].

The training and validation phase ended up with an accuracy of 100% and a loss of 0.011 $\pm$ 0.007 for the training set, and an accuracy of 93% and a loss of 0.251 $\pm$ 0.109 for the validation set.

From the loss and error plots depicted in Fig.[3.4] and Fig.[3.5] we can observe an evolution similar to the one of the previous case but we notice the noise in the validation loss and error has been radically reduced, but also that the model does not improve a lot in the validation phase. The first result is because the batch size has increased so the single batches are more similar to each other in terms of distribution of the images. The second result is again due to overfitting, the model can't still generalize and improve itself significantly.

In the end, applying the trained model to the test dataset has been observed an overall accuracy of 95% with an average loss of about 0.153 $\pm$ 0.216. From Fig.[3.6] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model, also in this case, struggles to recognize correctly the sixth image.

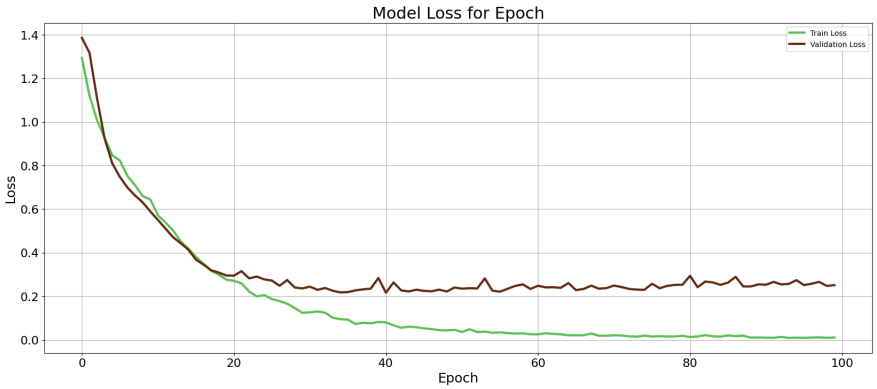

Figure 3.4: MWD Model Loss for Epoch - 64 batch size

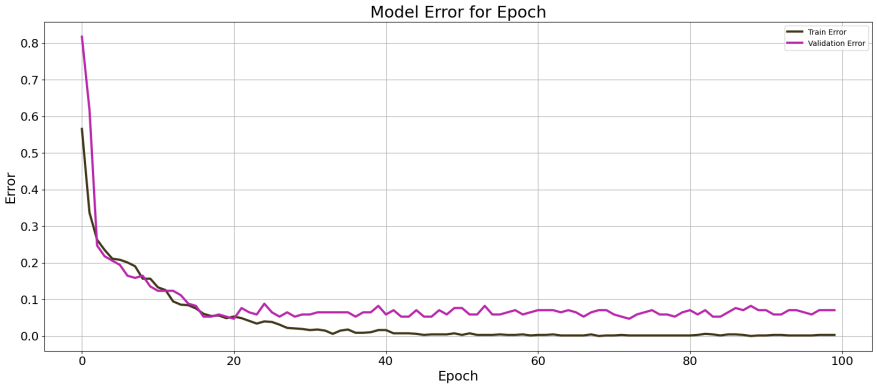

Figure 3.5: MWD Model Error for Epoch - 64 batch size

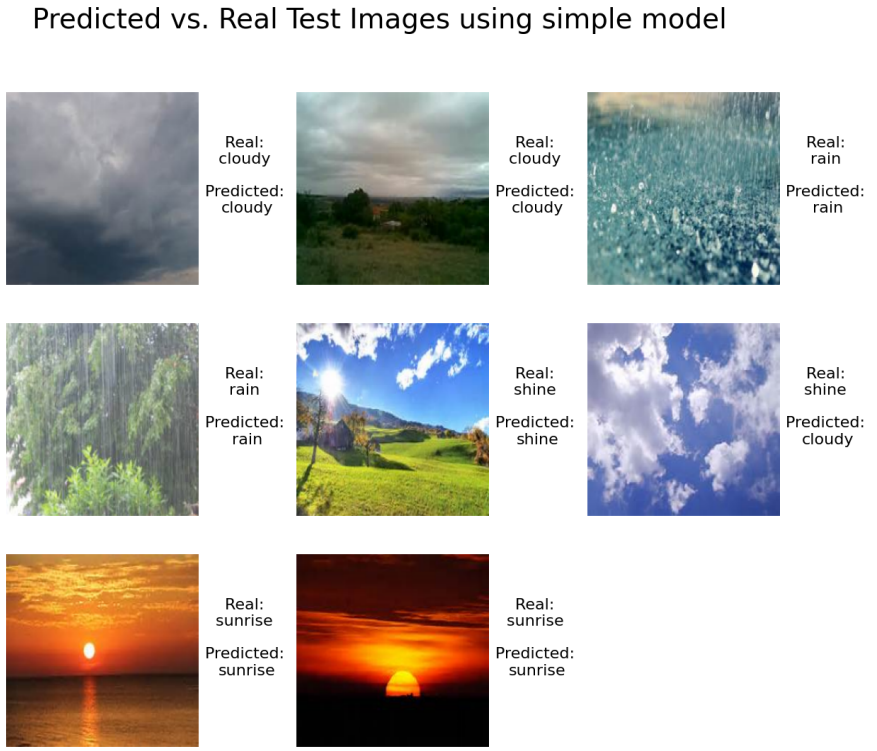

Figure 3.6: MWD Predicted vs. Real Test Images - 64 batch size

3.2.2 ACDC Dataset

Passing at the ACDC dataset, the training phase has been done using Cross-entropy as a loss function and Adam Optimization Algorithm to update net weights. The hyperparameters used are the following:

- Number of epochs: 100
- Learning rate: 10^{-4}
- Weight decay factor (L2-penalty): 10^{-3}
- Dropout: 0.4

Again, starting from the case of a batch size equal to 32, the results are depicted in Fig.[3.7], Fig.[3.8] and Fig.[3.9]. The training and validation phase ended up with an accuracy of 100% and a loss of 0.022 $\pm$ 0.023 for the training set, and an accuracy of 98% and a loss of 0.183 $\pm$ 0.102 for the validation set.

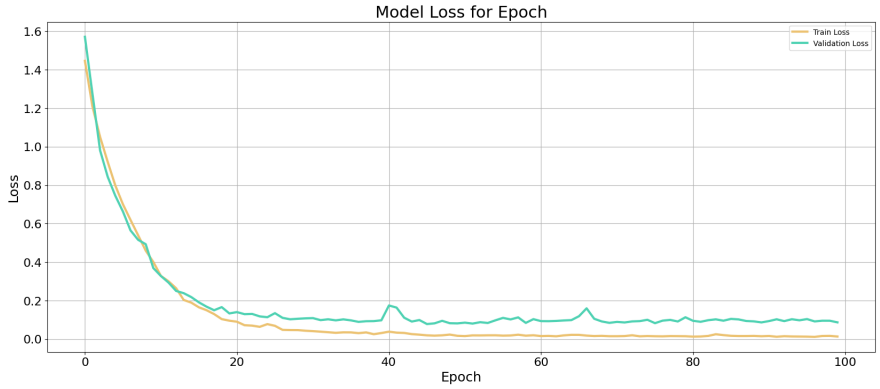

Figure 3.7: ACDC Model Loss for Epoch - 32 batch size

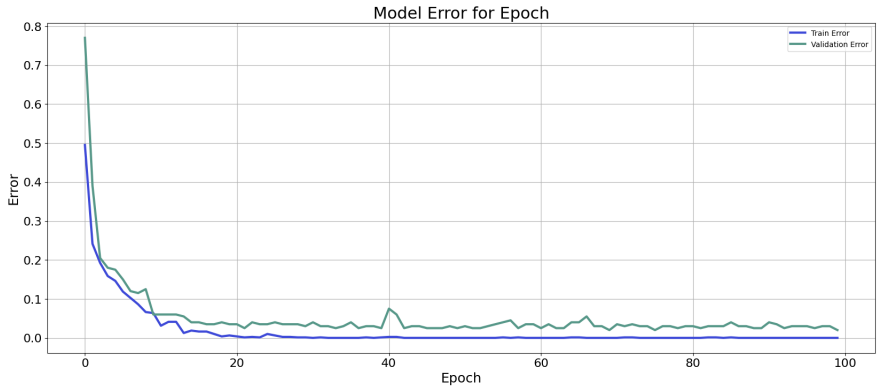

Figure 3.8: ACDC Model Error for Epoch - 32 batch size

From the plots depicted in Fig.[3.7] and Fig.[3.8] we can see that in this case, the model is showing pretty good results since there is no more noise in the validation loss and error evolution

and the gap between training and validation performance is small, so we can say that there is no overfitting. So in the end the performance in this case is really good.

Applying the trained model to the test dataset has been observed an overall accuracy of 92% with an average loss of about 0.271 ± 0.121, a little lower than previous tests. From Fig.[3.9] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model correctly predicts all the depicted images, even the harder ones like the last picture where the snow scenario is harder to recognize.

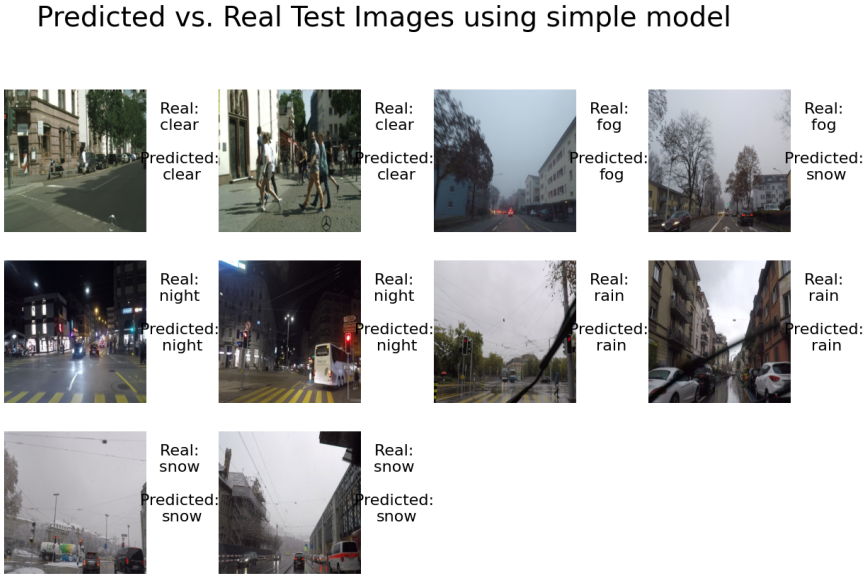

Figure 3.9: ACDC Predicted vs. Real Test Images - 32 batch size

Increasing the batch size to 64 another test has been done with the same configuration and hyperparameters as before. The results are depicted in Fig.[3.10], Fig.[3.11] and Fig.[3.12]. The training and validation phase ended up with an accuracy of 100% and a loss of 0.008 ± 0.003 for the training set, and an accuracy of 98% and a loss of 0.134 ± 0.134 for the validation set.

In this case, as before, we can observe a similar evolution of the training and validation phase but with better results and performance, thanks to the increase of the batch size.

Applying the trained model to the test dataset has been observed an overall accuracy of 95% with an average loss of about 0.173 ± 0.069. From Fig.[3.12] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model correctly predicts all the depicted images, even the harder ones like the last picture where the snow scenario is harder to recognize.

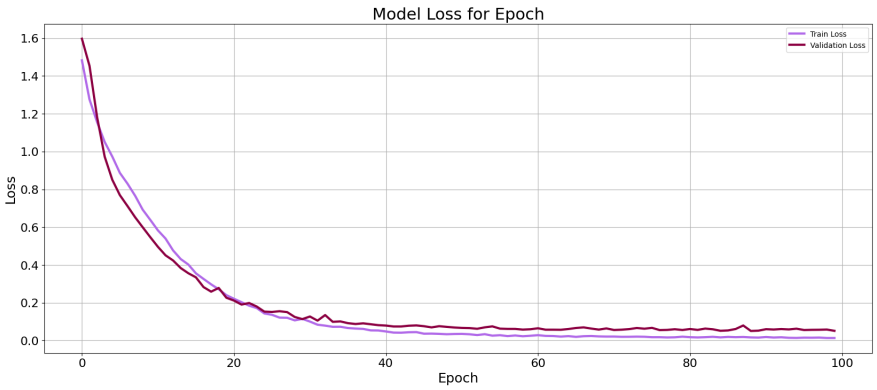

Figure 3.10: ACDC Model Loss for Epoch - 64 batch size

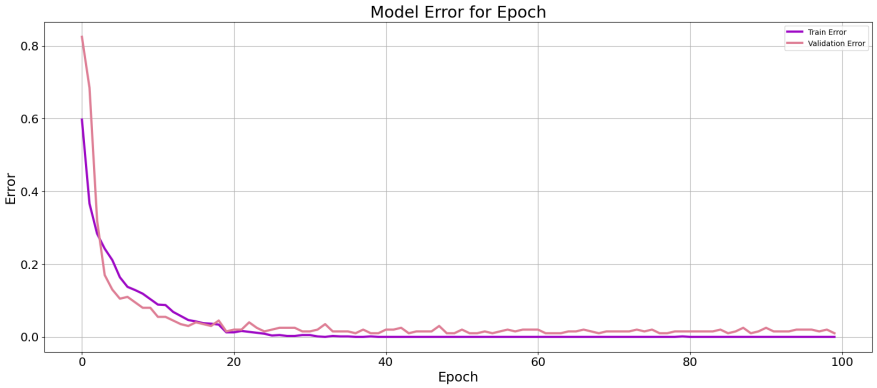

Figure 3.11: ACDC Model Error for Epoch - 64 batch size

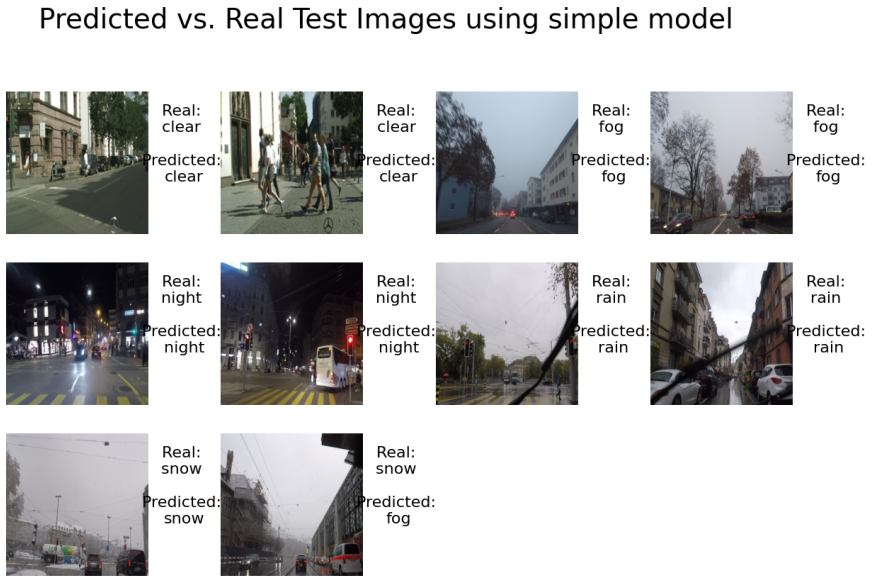

Figure 3.12: ACDC Predicted vs. Real Test Images - 64 batch size

3.2.3 SynDrone Dataset

Using the SynDrone dataset the training phase has been done using Cross-entropy as a loss function and Adam Optimization Algorithm to update net weights. The hyperparameters used are the following:

- Number of epochs: 100
- Learning rate: 10^{-4}
- Weight decay factor (L2-penalty): 10^{-3}
- Dropout: 0.32

The results, both for the case with a batch size of 32 and 64, are quite straightforward and really good. The results are depicted in Fig.[3.13], Fig.[3.14], Fig.[3.15], Fig.[3.16], Fig.[3.17] and Fig.[3.18]. The training and validation phase using a batch size equal to 32 ended up with an accuracy of 100% and a loss of 0.003 ± 0.002 for the training set, and an accuracy of 100% and a loss of 0.001 ± 0.000 for the validation set. Similarly, the training and validation phase using a batch size equal to 64 ended up with an accuracy of 100% and a loss of 0.005 ± 0.001 for the training set, and an accuracy of 100% and a loss of 0.008 ± 0.004 for the validation set.

By looking at the plots of the loss and error, the model takes up to 50 epochs to reach a loss of about 0.0 and it takes a little bit more than 10 steps to reach an accuracy of about 100%. These results can be explained by the nature of the dataset: the four weather scenarios are quite different from each other and there is a lot less noise in the images. The model can clearly recognize and differentiate the various scenarios and obtain these astonishing results.

Applying the trained model to the test dataset has been observed an overall accuracy of 100% with an average loss of about 0.002 ± 0.001 and 0.002 ± 0.001 for the 32 and 64 batch size respectively. From Fig.[3.15] and Fig.[3.18] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model correctly predicts all the depicted images.

3.2.4 Summary of the results

		Train		Validation		Test	
Dataset	Batch Size	Loss	Acc.	Loss	Acc.	Loss	Acc.
MWD	32	0.022 ± 0.023	100%	0.183 ± 0.102	92%	0.192 ± 0.248	94%
MWD	64	0.011 ± 0.007	100%	0.251 ± 0.109	93%	0.153 ± 0.216	95%
ACDC	32	0.004 ± 0.003	100%	0.057 ± 0.068	98%	0.271 ± 0.121	92%
ACDC	64	0.008 ± 0.003	100%	0.134 ± 0.134	98%	0.173 ± 0.069	95%
SynDrone	32	0.003 ± 0.002	100%	0.001 ± 0.000	100%	0.002 ± 0.001	100%
SynDrone	64	0.005 ± 0.001	100%	0.008 ± 0.004	100%	0.004 ± 0.001	100%

Table 3.1: Sample Table

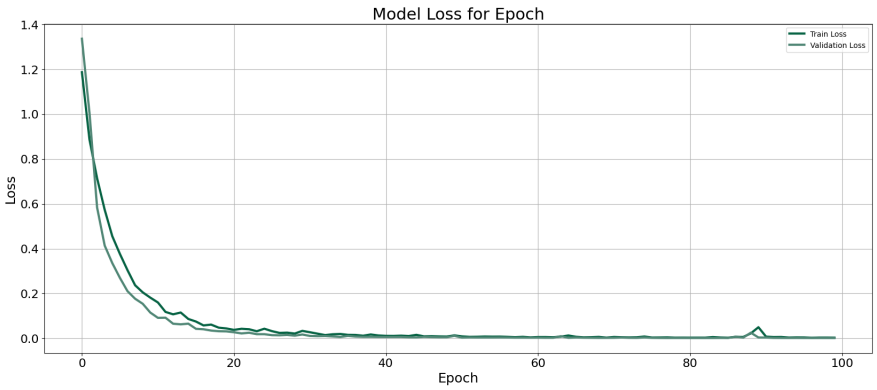

Figure 3.13: SynDrone Model Loss for Epoch - 32 batch size

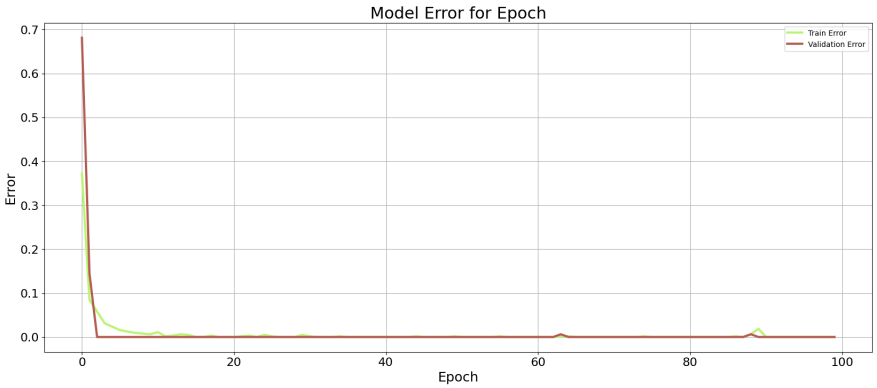

Figure 3.14: SynDrone Model Error for Epoch - 32 batch size

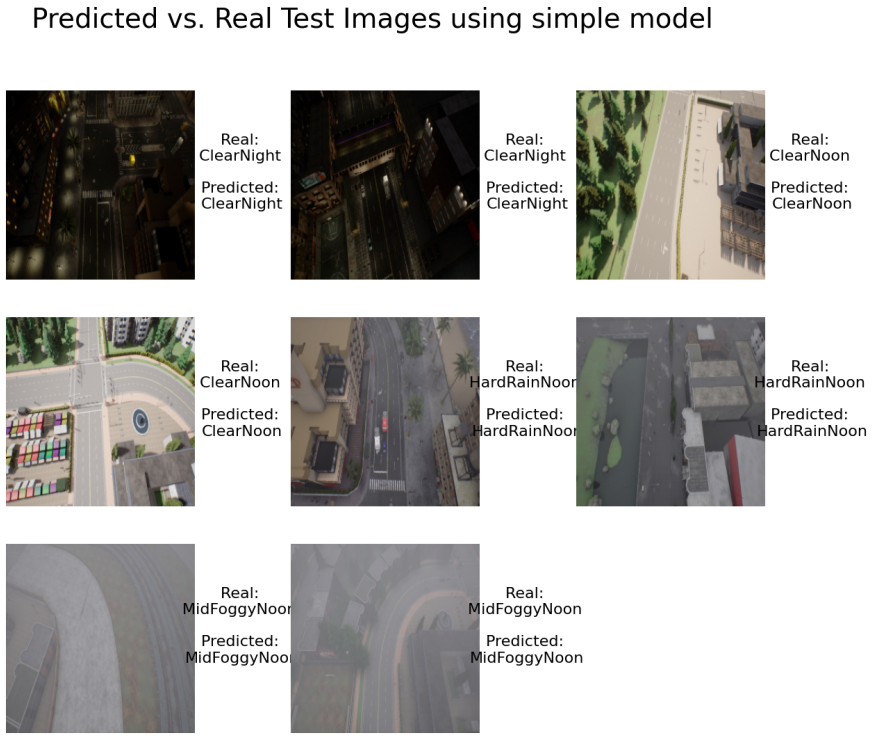

Figure 3.15: SynDrone Predicted vs. Real Test Images - 32 batch size

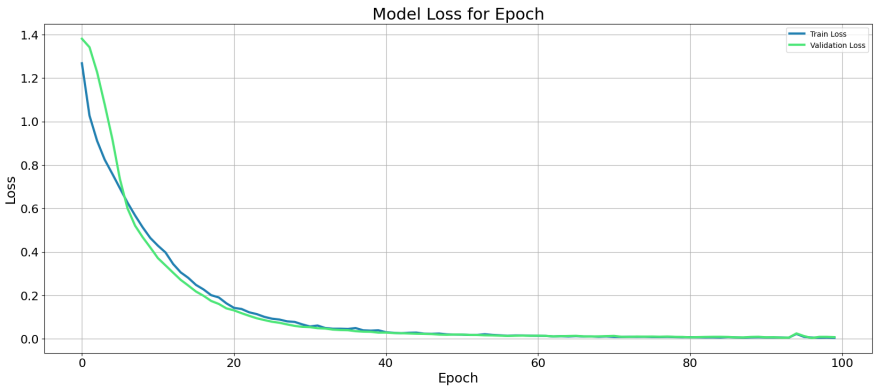

Figure 3.16: SynDrone Model Loss for Epoch - 64 batch size

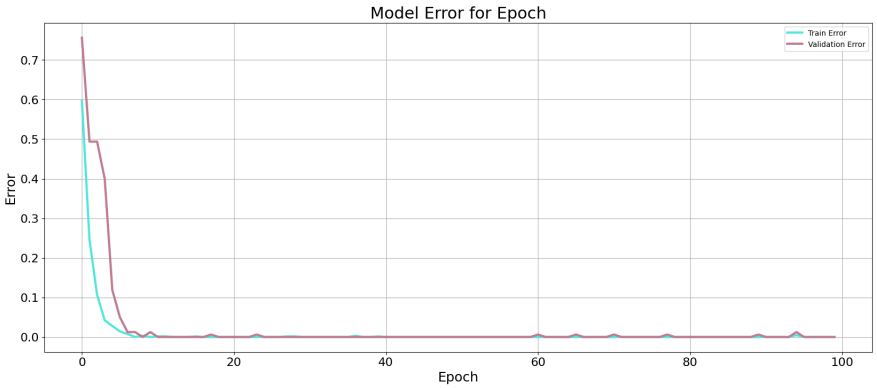

Figure 3.17: SynDrone Model Error for Epoch - 64 batch size

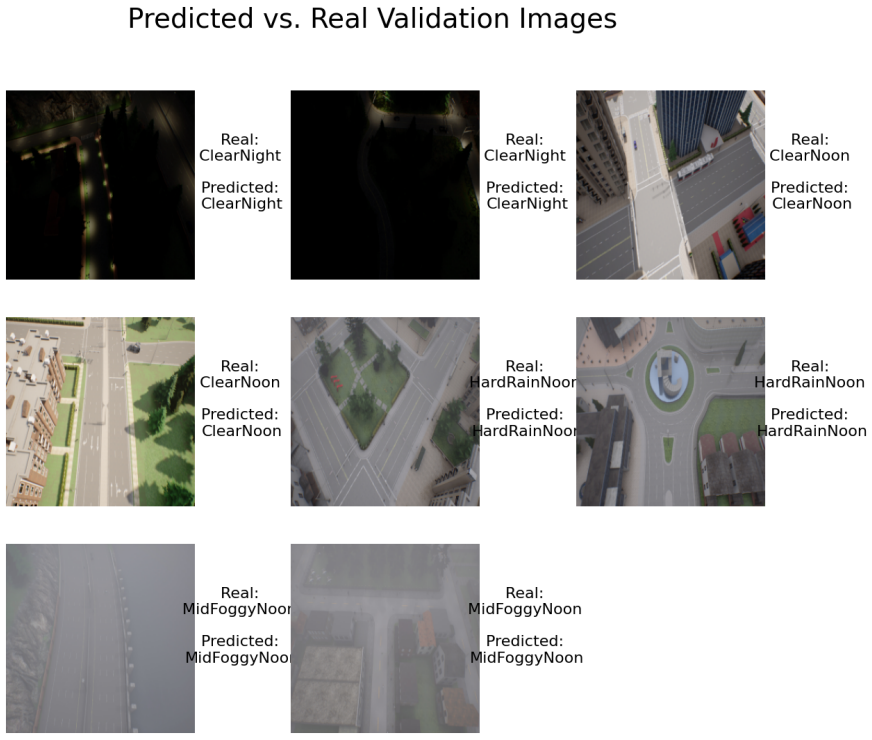

Figure 3.18: SynDrone Predicted vs. Real Test Images - 64 batch size

Chapter 4

Results using ResNet

4.1 Qualitative analysis of the model

For this project, the second model considered is a Residual Network, also called ResNet. This type of architecture is particularly effective for image classification tasks and is known for its ability to handle deep network training challenges thanks to the use of the residual.

In ResNets, the residual is used by introducing a skip connection or shortcut path between layers. The output of a previous layer is added directly to the output of the subsequent layer. This mechanism allows the network to learn and propagate the residual information (the difference between the input and output of a layer) during training. It addresses the vanishing gradient problem and facilitates the training of very deep neural networks by ensuring smoother gradient flow through the network.

From a theoretical point of view the pros of using a more complex architecture (e.g. ResNet) could be:

- ResNets enable the effective learning of hierarchical features through skip connections, allowing the network to capture intricate patterns and representations in data.
- Complex architectures, like ResNet, have a higher capacity to learn intricate relationships within the data, potentially leading to superior performance on complex tasks.

While the cons could be:

- On smaller datasets or less complex tasks, a highly complex model may be prone to overfitting, capturing noise in the data rather than generalizing well to new, unseen examples.
- The increased depth and complexity of ResNet result in higher computational requirements during training and inference, making it more resource-intensive.

'ResidualBlock' and 'ResNet' classes make up a complete architecture composed in the following way.

Starting from the Residual Block:

- Input: Tensor with shape [batch_size, in_channels, image_height, image_width]
- ConvBlock 1:
  - Convolutional Layer (3x3 kernel, padding=1)
  - Batch Normalization
  - ReLU Activation
- ConvBlock 2:
  - Convolutional Layer (3x3 kernel, padding=1)
  - Batch Normalization
- Downsample Connection (if specified)
- ReLU Activation
- Output: Tensor with shape [batch_size, out_channels, image_height, image_width]

Then ResNet achitercture is built in the following way (for this project):

- Input: Tensor with shape [batch_size, 3, image_height, image_width]
- Initial Convolutional Layer:
  - Convolutional Layer Convolutional Layer (7x7 kernel, stride=2, padding=3)
  - Batch Normalization
  - ReLU Activation
- Max Pooling Layer (3x3 kernel, stride=2, padding=1)
- Layer 0:
  - Sequence of Residual Blocks (Number of Blocks: 3)
- Layer 1:
  - Sequence of Residual Blocks (Number of Blocks: 4, Stride: 2)
- Layer 2:
  - Sequence of Residual Blocks (Number of Blocks: 6, Stride: 2)
- Layer 3:
  - Sequence of Residual Blocks (Number of Blocks: 3, Stride: 2)
- Downsample Connection Global Average Pooling (7x7 kernel)
- Fully Connected Layer (Linear Layer) with Output Channels: 'number of classes'
- Output: Tensor with shape [batch_size, num_classes]

4.2 Results of training, validation and test phase

The training and validation phase has been done using different configurations of hyperparameters and using the three datasets 'MWD', 'ACDC' and 'SynDrone'.

4.2.1 MWD Dataset

Starting from the MWD dataset, the training phase has been done using Cross-entropy as a loss function and Adam Optimization Algorithm to update net weights. The hyperparameters used are the following:

- Number of epochs: 50
- Learning rate: 10^{-4}
- Weight decay factor (L2-penalty): 10^{-4}

Starting from the case of a batch size equal to 32, the results are depicted in Fig.[4.1], Fig.[4.2] and Fig.[4.3]. The training and validation phase ended up with an accuracy of 97% and a loss of 0.101 ± 0.084 for the training set, and an accuracy of 93% and a loss of 0.185 ± 0.164 for the validation set.

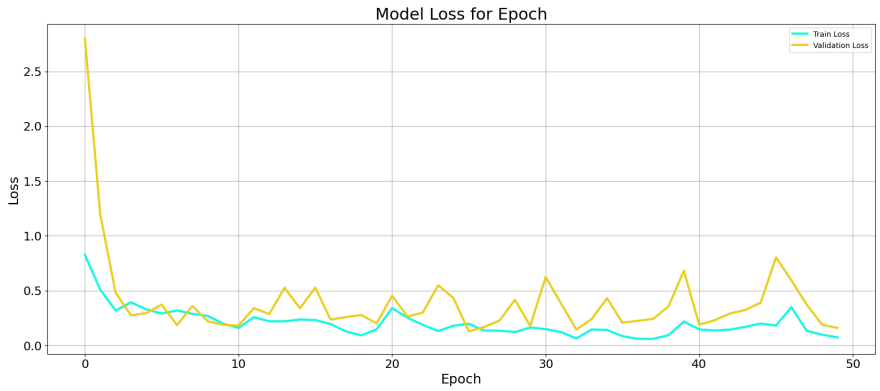

Figure 4.1: MWD ResNet Loss for Epoch - 32 batch size

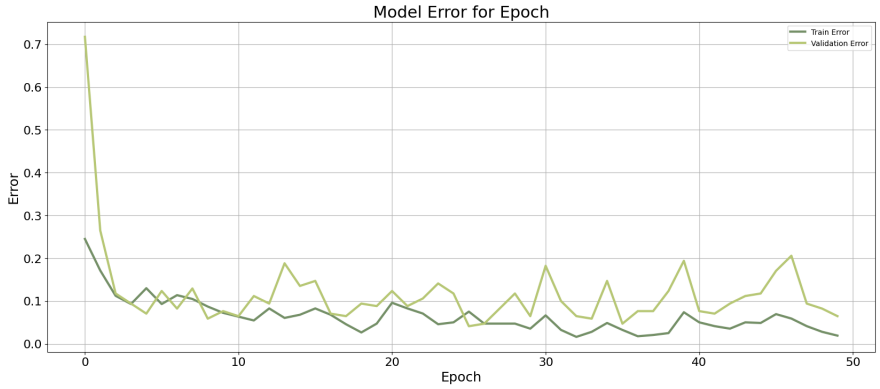

Figure 4.2: MWD ResNet Error for Epoch - 32 batch size

As we can see from the plots in Fig.[4.1] and Fig.[4.2] there is a lot of noise both in the training and validation phases. This phenomenon can be explained by the batch size and the complexity of the model. Regarding the first problem, we already noticed, using the simpler model, that by increasing the batch size to 64 it is possible to reduce overfitting problems and the noise in the training and validation phase, as well as improving the overall performance of the model. In this case, this problem is emphasized by the complexity of the net, which directly connects with the second problem. These types of architectures need to be fed with a lot of samples to train accurately, the datasets taken into consideration have only 1000 samples at most, and increasing the batch size by more than 64 didn't make much more difference because, on the other hand, the number of batches would be reduced. So, in general, the net is too complex to deal with such a small number of samples. To obtain better performance a larger dataset has to be taken into consideration.

Applying the trained model to the test dataset has been observed an overall accuracy of 96% with an average loss of about 0.105 $\pm$ 0.120. From Fig.[4.3] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model correctly predicts almost all the depicted images. Again, the model struggles to correctly predict the sixth image where it confuses the clouds in the sky with the case in which the weather is 'cloudy', rather than 'shine'. So, even in this case, the model isn't able to generalize and fully predict the images.

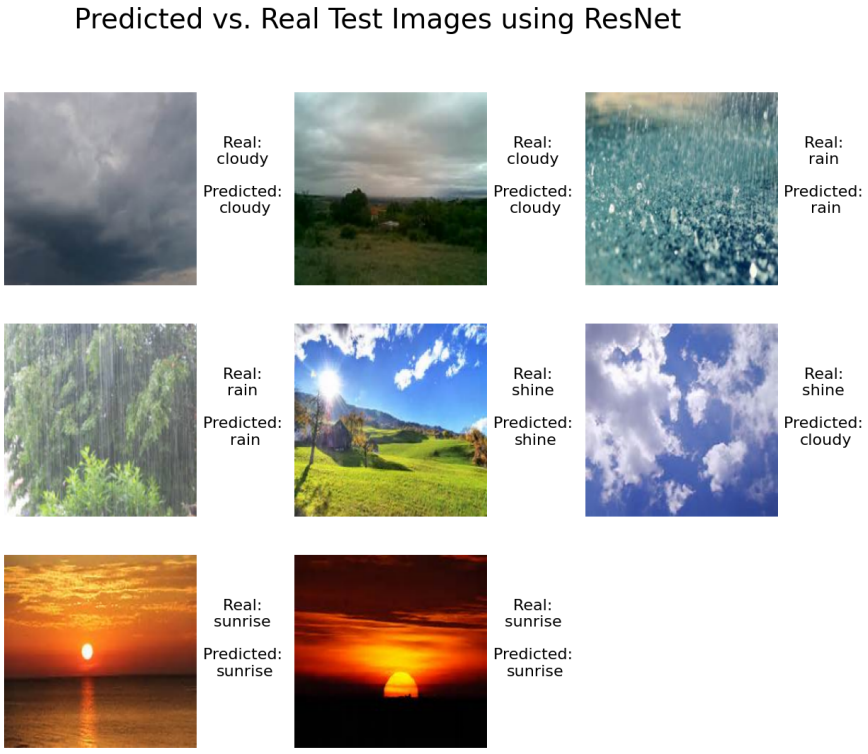

Figure 4.3: MWD ResNet Test for Epoch - 32 batch size

Passing to tha case in which the batch size has been increased to 64, with the same configuration and hyperparameters as before, the results are depicted in Fig.[4.4], Fig.[4.5] and Fig.[4.6]. The training and validation phase ended up with an accuracy of 100% and a loss of 0.008 $\pm$ 0.010 for the training set, and an accuracy of 95% and a loss of 0.187 $\pm$ 0.124 for the validation set.

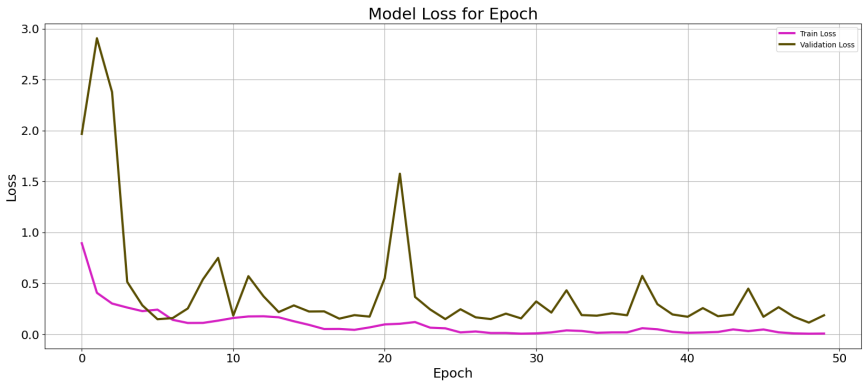

Figure 4.4: MWD ResNet Loss for Epoch - 64 batch size

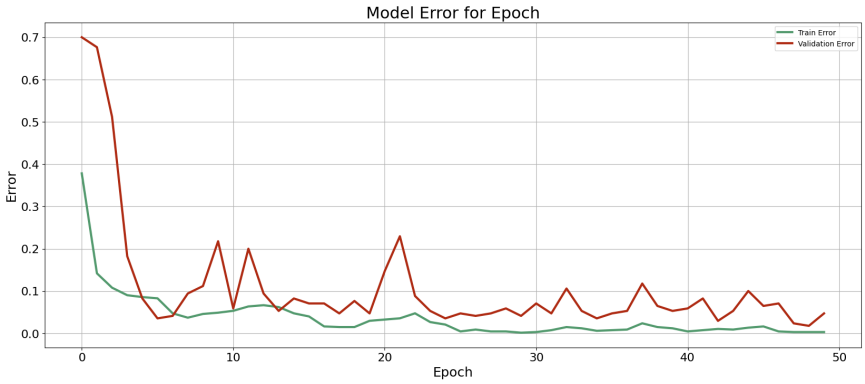

Figure 4.5: MWD ResNet Error for Epoch - 64 batch size

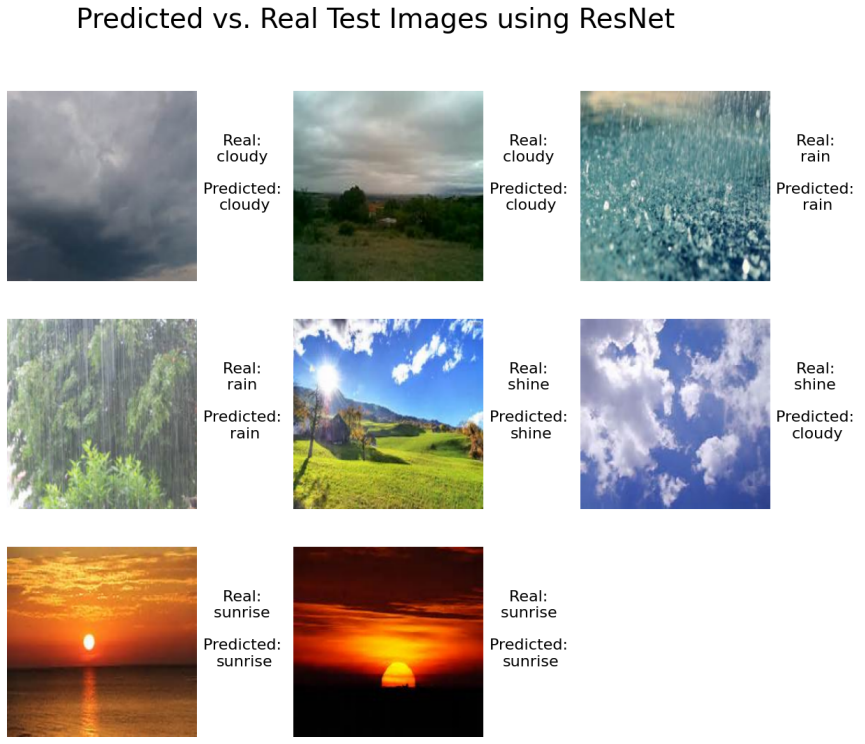

Figure 4.6: MWD ResNet Test for Epoch - 64 batch size

In this case, as we can observe from Fig.[4.4] and Fig.[4.5], there was a general improvement in the performance of the trained model but still, there is a lot of noise in the training and validation phase. Increasing the batch size didn't help a lot, but still, we can state that the overall performance of the model is pretty good, slightly more than the ones of the simpler model, but not so much as we would expect from an architecture like this.

Applying the trained model to the test dataset has been observed an overall accuracy of 96% with an average loss of about 0.105 ± 0.120. From Fig.[4.3] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model correctly predicts almost all the depicted images. Again, the model struggles to correctly predict the sixth image where it confuses the clouds in the sky with the case in which the weather is 'cloudy', rather than 'shine'. So, even in this case, the model isn't able to generalize and fully predict the images.

4.2.2 ACDC Dataset

Using the ACDC dataset, the training phase has been done using Cross-entropy as a loss function and Adam Optimization Algorithm to update net weights. The hyperparameters used are the following:

- Number of epochs: 50
- Learning rate: 10^{-4}
- Weight decay factor (L2-penalty): 10^{-3}

In the case of a batch size equal to 32, the results are depicted in Fig.[4.7] and Fig.[4.8]. The training and validation phase ended up with an accuracy of 100% and a loss of 0.004 ± 0.012 for the training set, and an accuracy of 97% and a loss of 0.099 ± 0.083 for the validation set.

In the case of a batch size equal to 64, the results are depicted in Fig.[4.10] and Fig.[4.11]. The training and validation phase ended up with an accuracy of 100% and a loss of 0.000 ± 0.000 for the training set, and an accuracy of 98% and a loss of 0.030 ± 0.041 for the validation set.

Applying the trained model to the test dataset has been observed an overall accuracy of 96% with an average loss of about 0.161 ± 0.121 in the case of a batch size equal to 32, instead if we increase the size to 64 we observe an overall accuracy of 97% with an average loss of about 0.141 ± 0.107. From Fig.[4.3] and Fig.[4.6] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model correctly predicts almost all the depicted images. In the case the batch size is equal to 32 we can observe some struggles from the net to recognize the last image as 'snow' since the weather is very similar to 'fog', instead with the batch size equal to 64 this problem is overcome and the trained model correctly predicts all the images.

From the MWD dataset we can observe an overall improvement in the prediction and in the performance, in fact, the evolution of the training and validation phase, both for the batch size equal to 32 and 64, is less noisy and in the end more constant with a loss and an error almost close

to zero. This can be in part explained by the bigger size of the dataset and by the distribution of the images which are very similar to each other in the same class, but also different from the one of other classes.

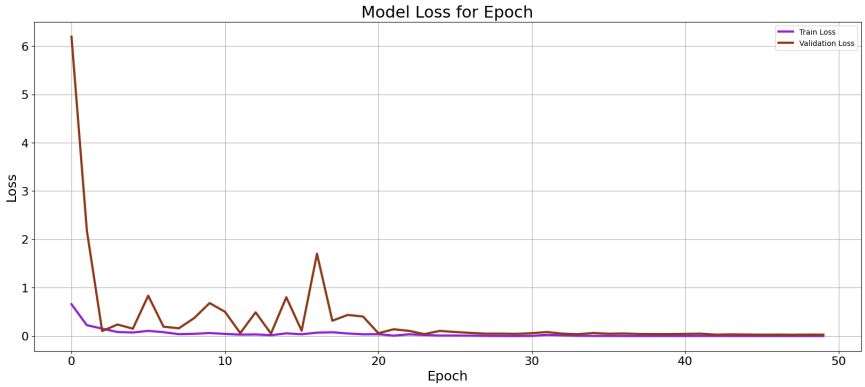

Figure 4.7: ACDC ResNet Loss for Epoch - 32 batch size

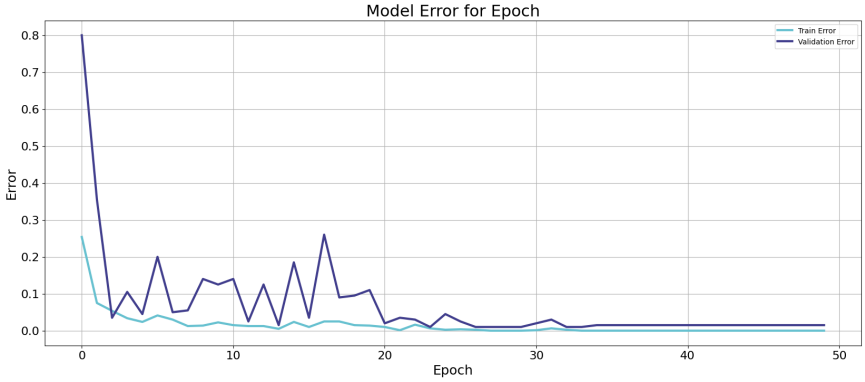

Figure 4.8: ACDC ResNet Error for Epoch - 32 batch size

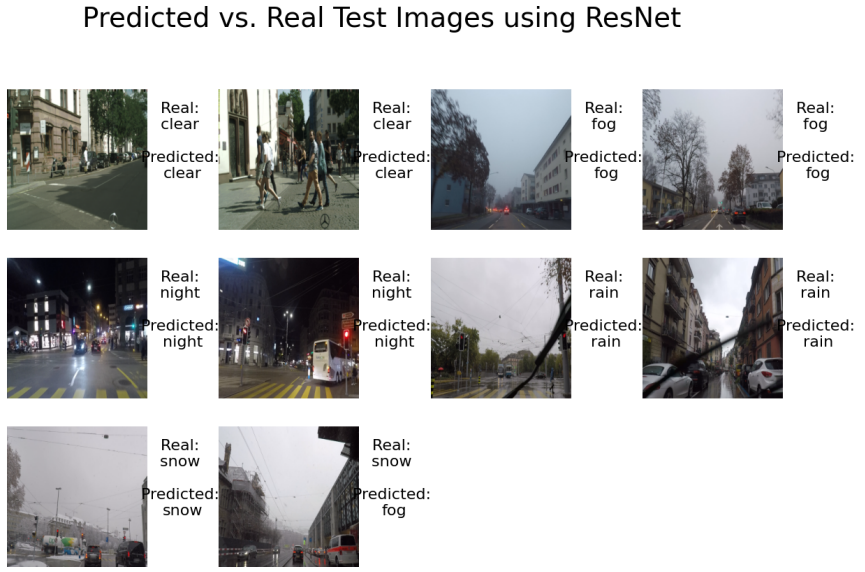

Figure 4.9: ACDC ResNet Test for Epoch - 32 batch size

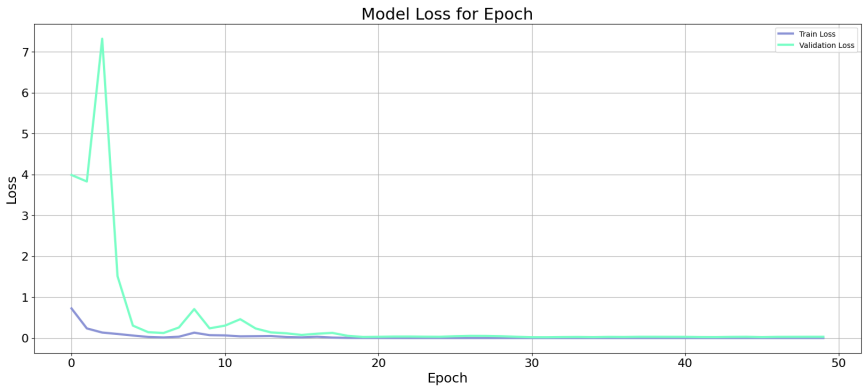

Figure 4.10: ACDC ResNet Loss for Epoch - 64 batch size

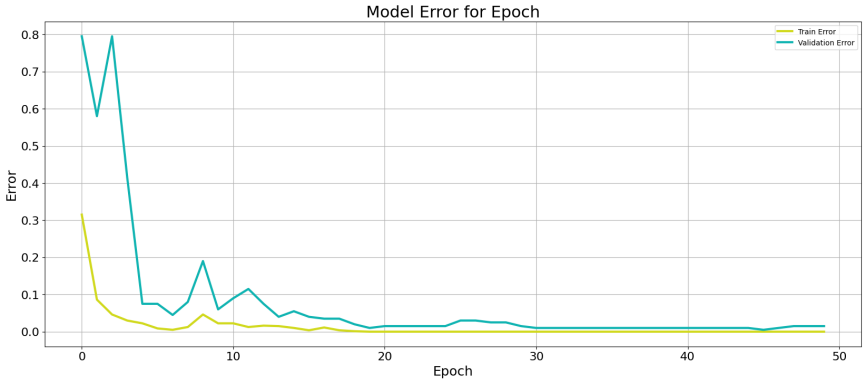

Figure 4.11: ACDC ResNet Error for Epoch - 64 batch size

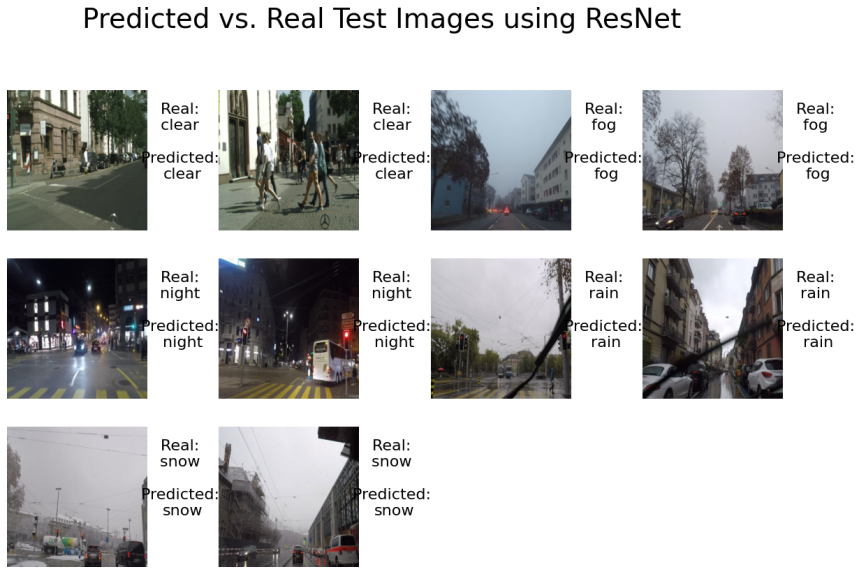

Figure 4.12: ACDC ResNet Test for Epoch - 64 batch size

4.2.3 SynDrone Dataset

Using the SynDrone dataset, the training phase has been done using Cross-entropy as a loss function and Adam Optimization Algorithm to update net weights. The hyperparameters used are the following:

- Number of epochs: 50
- Learning rate: 10^{-4}
- Weight decay factor (L2-penalty): 10^{-4}

In the case of a batch size equal to 32, the results are depicted in Fig.[4.13] and Fig.[4.14]. The training and validation phase ended up with an accuracy of 100% and a loss of 0.002 ± 0.004 for the training set, and an accuracy of 100% and a loss of 0.001 ± 0.002 for the validation set.

In the case of a batch size equal to 64, the results are depicted in Fig.[4.16] and Fig.[4.17]. The training and validation phase ended up with an accuracy of 100% and a loss of 0.000 ± 0.000 for the training set, and an accuracy of 100% and a loss of 0.000 ± 0.000 for the validation set.

Applying the trained model to the test dataset has been observed an overall accuracy of 100% with an average loss of about 0.000 ± 0.000 in the case of a batch size equal to 32, instead if we increase the size to 64 we observe an overall accuracy of 100% with an average loss of about 0.000 ± 0.000. From Fig.[4.15] and Fig.[4.18] we can observe some examples of predicted labels of images extracted from the test dataset and we can see that the model correctly predicts all the depicted images.

Again, the results obtained using this dataset are really good since we got a 100% accuracy in all the training, validation, and test datasets, for both the case with a batch size equal to 32 and 64. The reasonings for these results can be found, as for the simpler CNN, in the nature of the SynDrone dataset, the distribution and clarity of the images.

4.2.4 Summary of the results

		Train		Validation		Test	
Dataset	Batch Size	Loss	Acc.	Loss	Acc.	Loss	Acc.
MWD	32	0.101 ± 0.084	96%	0.185 ± 0.164	96%	0.105 ± 0.120	96%
MWD	64	0.008 ± 0.010	100%	0.187 ± 0.124	95%	0.144 ± 0.188	96%
ACDC	32	0.004 ± 0.012	100%	0.099 ± 0.083	97%	0.161 ± 0.121	96%
ACDC	64	0.000 ± 0.000	100%	0.030 ± 0.041	98%	0.141 ± 0.107	97%
SynDrone	32	0.002 ± 0.004	100%	0.001 ± 0.002	100%	0.000 ± 0.000	100%
SynDrone	64	0.000 ± 0.000	100%	0.000 ± 0.000	100%	0.000 ± 0.000	100%

Table 4.1: Sample Table

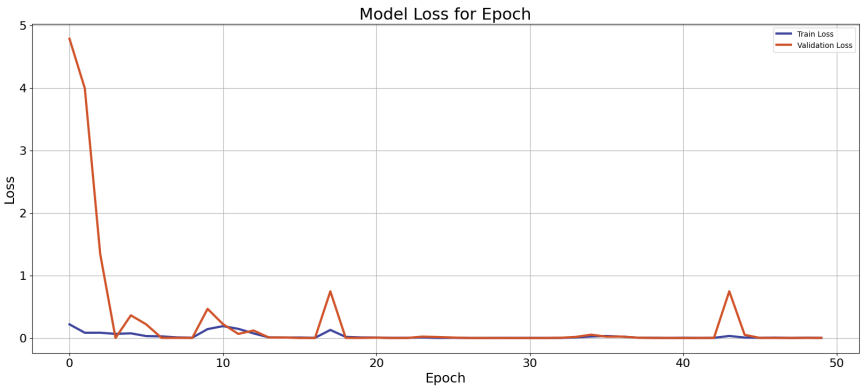

Figure 4.13: SynDrone ResNet Loss for Epoch - 32 batch size

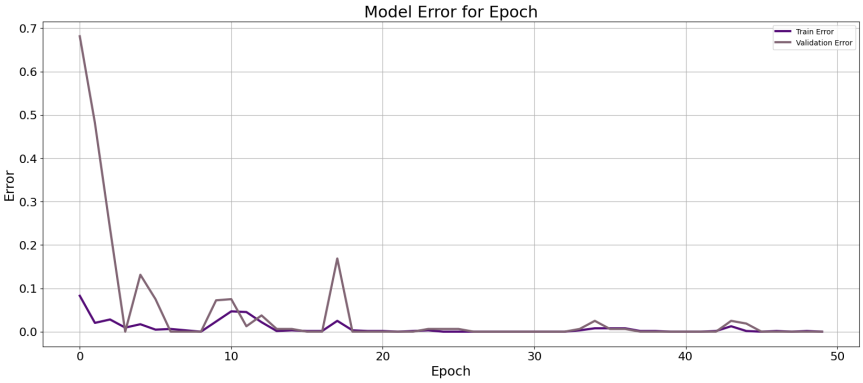

Figure 4.14: SynDrone ResNet Error for Epoch - 32 batch size

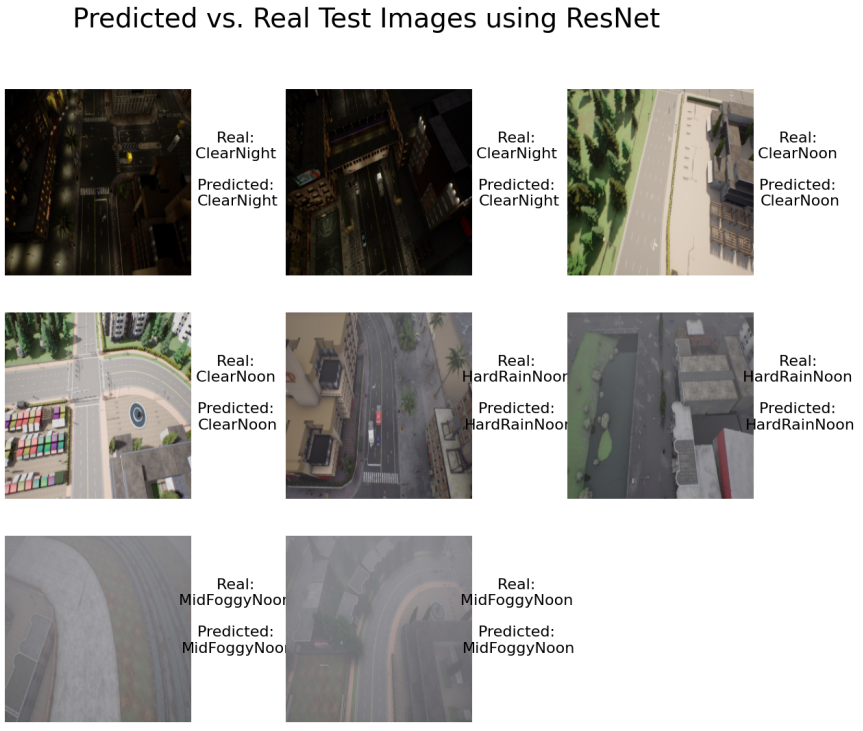

Figure 4.15: SynDrone ResNet Test for Epoch - 32 batch size

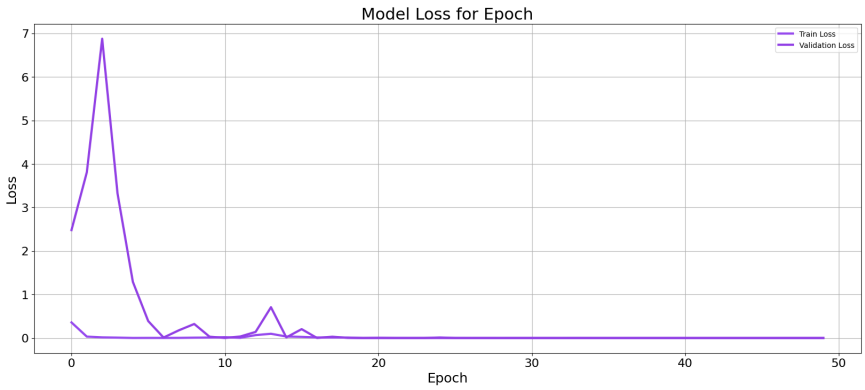

Figure 4.16: SynDrone ResNet Loss for Epoch - 64 batch size

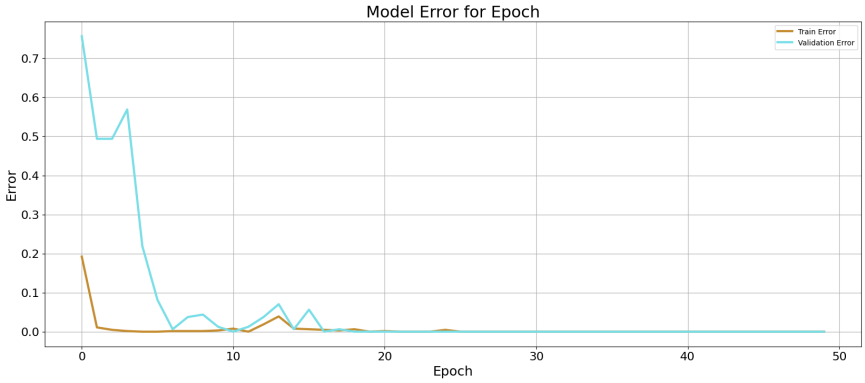

Figure 4.17: SynDrone ResNet Error for Epoch - 64 batch size

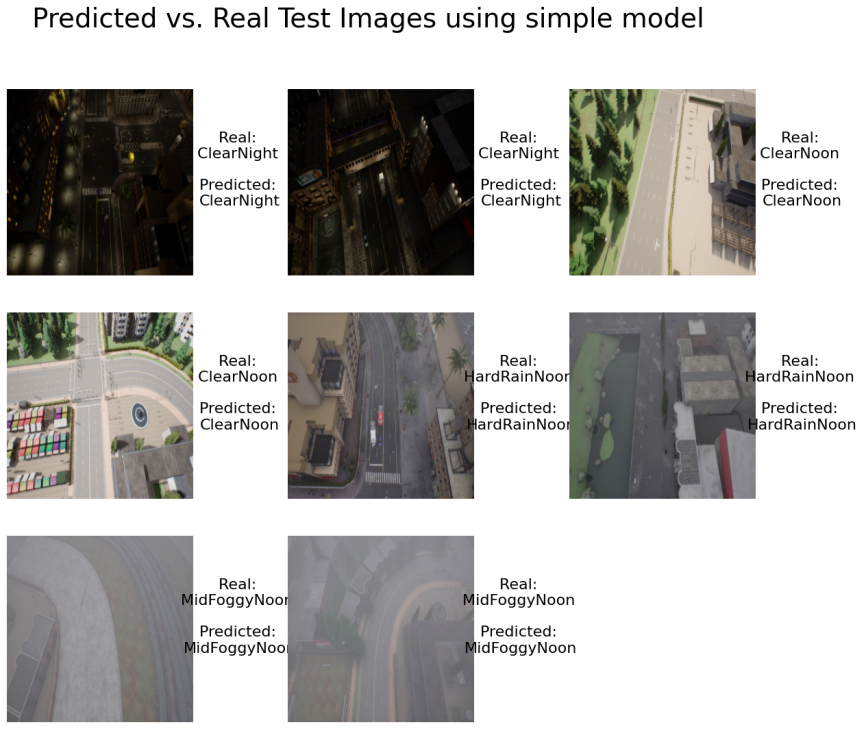

Figure 4.18: SynDrone ResNet Test for Epoch - 64 batch size

Chapter 5

Conclusions

From the results observed above it is possible see that in general a good result has been reached. In the training phase, for a batch size equal to 64, a consistent 100% accuracy has been achieved, also in the test phase the accuracy was over 94%. Similar, but worse, results are observed in the case where the batch size is equal to 32. These results demonstrate the effectiveness of these architectures in learning and generalizing patterns from the given dataset. The high training accuracy suggests that the models are capable of fitting the training data well, moreover, the success of both architectures in maintaining high accuracy on the test set indicates their ability to avoid overfitting, emphasizing the effectiveness of regularization techniques employed in the models.

Additionally, the observation that a batch size of 64 outperforms a batch size of 32 in terms of overall performance, as indicated by higher test set accuracy, aligns with a common trend in deep learning. Larger batch sizes often contribute to more stable and efficient training. A batch size of 64 likely enables the models to have a superior generalization performance on the test set.

We also observe some differences from the simpler CNN compared to ResNet in the training phase, since in the first case the evolution of the validation phase is in general less noisier than the one of ResNet. This behavior directly comes from the nature of the models: the first model is a simple CNN with only four layers, so it has a really simple architecture, while the second model has a much more complex architecture compared to the first one. The complexity of the ResNet model requires that the model is properly fed, i.e. much larger datasets and possibly more classes.

In summary, the consistently high accuracy during training and strong generalization to the test set highlight the efficacy of both the simpler CNN and ResNet architectures. In the end, the simpler model is preferred over ResNet for its speed in the training and validation phase and for the results and performance reached that are not so much different than the one of ResNet. Also, a batch size of 64 is preferred over smaller ones.

All the trained models using a batch size equal to 64, shown in the Figures of this report, are properly saved into the project folder 'trained_models'. These files contain only the values of the weights and not the entire models.

Bibliography

[1] Rahul Agarwal. "Complete Guide to the Adam Optimization Algorithm". In: builtin, 2023. URL: https://builtin.com/machine-learning/adam-optimization.

[2] Amartya Banerjee. "Classification of Images in Multi-class Weather Dataset (MWD) using Deep-Learning in PyTorch". In: Jovian, 2023. URL: https://jovian.com/amartyabanerjee99/04-zero2gans-course-project#C74.

[3] Chirag Daryani. "Classification of Weather Images using ResNet-34 in PyTorch". In: Medium, 2020. URL: https://medium.com/swlh/classification-of-weather-images-using-resnet-34-in-pytorch-7e86b2b24dcf.

[4] Nouman. "Writing ResNet from Scratch in PyTorch". In: Paperspace, 2022. URL: https://blog.paperspace.com/writing-resnet-from-scratch-in-pytorch/.

[5] Lucas Silva. "Weather Image Classification with Pytorch". In: Kaggle, 2023. URL: https://www.kaggle.com/code/lucas142129silva/weather-image-classification-with-pytorch/.