UNIVERSITÀ DEGLI STUDI DI MILANO
FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

Master in
Computer Science

STATISTICAL METHODS FOR MACHINE LEARNING
NEURAL NETWORKS FOR BINARY IMAGE CLASSIFICATION

Professor: Nicolò Cesa-Bianchi

Student:
Lorenzo Romeo
Matr. Nr. 28815A

Academic year 2023-2024

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

Contents

Chapter 1

Introduction

This project consists of experimenting with different architectures for solving a binary classification problem. In particular, the problem requires distinguishing images of muffins from those of chihuahuas, a famous problem as the two entities appear to be comically similar. This similarity, however, suggest that this is a challenging problem to solve.
The development of the project begins with the exploration of the dataset and the definition of an adequate data preprocessing with the definition of training and test sets. Subsequently, the three chosen architectures are defined: a Multi Layer Perceptron, a Convolutional Neural Network, and finally a more advanced Convolutional Neural Network. Various experiments will be performed by applying data augmentation to the dataset and subsequently performing hyperparameter tuning on the defined architectures. We will show how the performance of the models evolves with the optimizations just presented, and at last perform a final training using a 5-fold cross validation, which will produce the final evaluations for the models. Finally, the last section proposes explanations of the results observed in the experimentation phase.

Chapter 2

Dataset

The dataset used was taken from Kaggle, and consists of 3199 photos of Chihuahuas and 2718 photos of muffins. it is important to specify that the dataset contains many low quality photos, as they contain images that are not relevant to our problem. Despite of that it was chosen to not edit manually or automatically the provided dataset.
The description of the creator of the dataset indicates that the images were obtained from Google Images, and that no preprocessing was performed, except for the removal of duplicate images.
To load the dataset *tensorflow.keras.preprocessing.image_dataset_from_directory* was used:

```
image_dataset_from_directory(
    training_dir,
    labels='inferred',
    label_mode='binary',
    image_size=(128, 128),
    interpolation='nearest',
    batch_size=None,
    shuffle=True,
    color_mode='rgb'
)
```

This method was used because it allows to obtain data from a directory, associate the correct label to the datapoints and carry out an initial normalization of the data. In particular, it is possible to specify the size of the images, the way in which they are interpolated and specify RGB as the color space to use. It also allows for shuffling the dataset for improved randomicity. The interpolation was chosen as *nearest*, because after experimentation it yield better training results compared to other tecniques, like *lanczos3*.
This operation is executed for both the folders containing the test and training images, obtainig two *tf.data.Dataset* datasets.

Subsequently, data normalization is carried out, transporting the RGB values of the images into the range [0,1):

```
tf.image.convert_image_dtype(image, dtype=tf.float32)
```

Finally, the datasets obtained are converted into two Numpy arrays, in order to make it easier to manipulate and modify the size of the dataset. These arrays are then used to create training and test sets:

```
test_size_perc = 0.2

dataset = ds_train.concatenate(ds_test)

X = np.array(list(dataset.map(lambda X, y: X).as_numpy_iterator())
    )
y = np.array(list(dataset.map(lambda X, y: y).as_numpy_iterator())
    )

test_size = int(len(X) * test_set_perc)
train_size = len(X) - test_size

X_train = X[:train_size]
y_train = y[:train_size]

X_test = X[train_size:]
y_test = y[train_size:]
```

As shown in the code, the size of the test set was set as 20% of the dataset.

To recap, the result of this preprocessing are the numpy arrays X, that contains the datapoints, and y, that contains the associated labels so that $\forall i \in [0, n]$ y_i is the label of X_i.

Figure 1 shows the final images with their corresponding labels (0 if chihuahua or 1 if muffin).

Figure 1: A sample from the dataset

Chapter 3

Models

In this chapter the chosen model architectures will be defined and described. All the models are defined as Keras Sequentials.

3.1 Multi Layer Perceptron

The first architecture defined is a Multi Layer Perceptron (MLP): a feed-forward neural network that is composed of multiple layers of fully connected neurons.

```
model_mlp = models.Sequential([

    Input(shape=(128, 128, 3)),
    layers.Flatten(),

    layers.Dense(512, activation='relu'),

    layers.Dense(256, activation='relu'),
    layers.Dropout(0.2),

    layers.Dense(128, activation='relu'),
    layers.Dropout(0.2),

    layers.Dense(64, activation='relu'),

    layers.Dense(1, activation='sigmoid')
])
```

The model has three Dense layers with a decreasing number of neurons and Dropout layers to randomically disable neurons to mitigate overfitting.
The first layer, the Flatten, is used to convert the input images to 1-dimensional arrays. The hidden layers use the *ReLU* activation function while the last layer is a single neuron with a *sigmoid* activation function, that returns the probability of the input of having label 1 (muffin).

3.2 Convolutional Neural Network

The second model defined is a Convolutional Neural Network (CNN): a feed-forward neural network widely used for computer vision tasks like image classification. It uses special layers: the convolutions, that apply filters to a sliding matrix over the image to extrapolate features, and maxPooling layers to reduce dimesionality and accentuate the features.

```
model_cnn = models.Sequential([
    Input(shape=input_shape),

    layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu'
    ),
    layers.MaxPooling2D((2,2)),

    layers.Conv2D(64,(3,3), activation='relu'),

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])
```

The first convolutional layer is used to detect rough features, like edges or colors. The second is used to find more complex features. The maxPooling is used to reduce the spatial dimensions and to accentuate the features detected by the convolutional layer. Lastly a Flatten layer is used to input the processed data into the last Dense layer. This layer is used to integrate the features learned in the previous convolutions to form an understanding of the features and then, through a neuron with the *sigmoid* activation function, produces the final prediction.

3.3 Advanced Convolutional Neural Network

The third model is a more complex, but still fairly simple, convolutional model.

```
model_cnn_1 = models.Sequential([
    Input(shape=input_shape),

    layers.Conv2D(32, (3,3), activation='relu'),
    layers.MaxPooling2D((3,3)),
    layers.Dropout(0.2),

    layers.Conv2D(64,(3,3), activation='relu'),
    layers.MaxPooling2D((3,3)),
    layers.Dropout(0.2),

    layers.Conv2D(64,(3,3), activation='relu'),
    layers.Dropout(0.5),
```

```
14
15      layers.Flatten(),
16      layers.Dense(64, activation='relu'),
17      layers.Dense(32, activation='relu'),
18      layers.Dense(1, activation='sigmoid')
19  ])
```

This model is similar to 3.2, but introduces an additional convolutional layer, maxPooling layer and Dense layer. The Convolution should be able to identify finer features, and the last Dense layer should provide better reasoning capabilities. Dropouts layers are also introduced to reduce and prevent overfitting.

3.4 Models building

The models are defined as Keras sequentials, and have the same compiling parameters. The chosen optimizer is *adam* because it's efficient and tipically yields good results. The chosen loss function is *binary crossentropy* because it's a good loss function for binary classification problems, and then, as requested, also a zero one loss function. This will be explained better in section 7.2.

```
1  def zero_one_loss(y_true, y_pred):
2      return tf.reduce_mean(tf.square(tf.round(y_pred) - y_true))
3
4  for cont, i in enumerate(models_list):
5      i.compile(
6          optimizer = 'adam',
7          loss = 'binary_crossentropy',
8          metrics = ['binary_accuracy', zero_one_loss]
9      )
10     models_json[model_names[cont]] = i.to_json()
```

Chapter 4

Data Augmentation

The first tecnique introduced to improve the performance of the model is the data agumentation. This tecnique consists in the artificial generation of new training examples, starting from the existing ones. This approach prevents overfitting helping the model to generalize more and potentially exposes the model to variations of real world usage. The chosen augmentation consists in random horizontal flips and rotations and was impemented using Keras Layers:

```
data_augmentation_layers = [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
    ]

def data_augmentation(images):
    for layer in data_augmentation_layers:
        images = layer(images)
    return images
])
```

Subsequenlty, an augmented dataset is created with the original and augmented images. The images were intertwined in the dataset, putting an augmented datapoint after every other image:

```
X_augm = np.zeros((len(X)*2, input_shape[0], input_shape[1],
   input_shape[2]))
y_augm = np.zeros((len(y)*2, 1))

index_norm = 0
index_augm = 0
for i in range(len(X)*2):
    if(i%2==0):
        X_augm[i] = X[index_norm]
        y_augm[i][0] = y[index_norm]
        index_norm += 1
    else:
```

Figure 2: Example of augmentation operations on an image

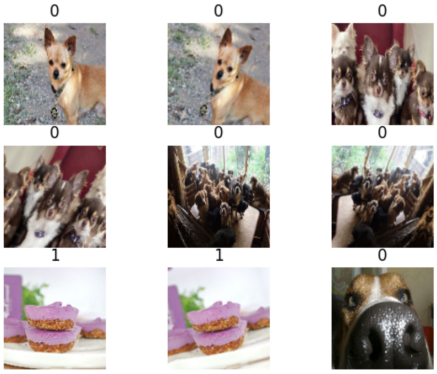

Figure 3: A sample of the augmented dataset

```
12          X_augm[i] = augmented_images[index_augm]
13          y_augm[i][0] = y[index_augm]
14          index_augm += 1
```

This operation was introduced in order to distribute the images eavenly and simplify the subsequent creation of training and test sets.
Image 2 shows 9 different augmentations on an image, while image 3 shows a sample of the augmented dataset. It's important to specify that putting the augmented version of an image immediately after it may not be optimal. To prevent any problems it's important to shuffle the batches before traning.

Chapter 5

Hyperparameter Tuning

The hyperparameters are all the parameters that have to be defined before training a model and cannot be learned through fitting. To improve models is a common practice to execute some kind of tuning to find better hyperparameters. This technique involves evaluating models using different hyperparameters, and finally selecting the set of hyperparameters that gave the best results. Furthermore, it is advisable to carry out the tests using a (not too small) subset of the dataset, in order to use fewer resources and therefore have faster computations. In our case 50% of the dataset was used for these operations.
To simplify the process *kerasTuner* was used to execute the tuning and choose automatically the iterations of hyperparameters to try.

For the MLP model the dropouts, the numbers of neurons of the dense layers and the learning rate were parametrized. The following is the constructor method of the model with chosen ranges for the parameters:

```
def build_model_mlp(hp):
    dropouts = hp.Float("dropout", min_value = 0.2, max_value =
   0.5, step = 0.1)

    model = models.Sequential([

        Input(shape=input_shape),
        layers.Flatten(),

        layers.Dense(256, activation='relu'),

        layers.Dense(hp.Int("dense_1", min_value=int(image_size
   [0]/2), max_value=256, step=32), activation='relu'),
        layers.Dropout(dropouts),

```

```
        layers.Dense(hp.Int("dense_2", min_value=int(image_size[0]/4), max_value= 256, step=32), activation='relu'),
        layers.Dropout(dropouts),

        layers.Dense(hp.Int("dense_3", min_value=int(image_size[0]/6), max_value= 256, step=32), activation='relu'),

        layers.Dense(1, activation='sigmoid')
    ])

    model.compile(
        optimizer = keras.optimizers.Adam(hp.Choice('learning_rate', values=[1e-2, 1e-3])),
        loss = 'binary_crossentropy',
        metrics = ['binary_accuracy', 'accuracy', zero_one_loss]
    )

    return model
```

For the first CNN model the filter number for the convolutions, the neurons number for the dense layer and the learning rate were parametrized:

```
def build_model_cnn(hp):
    model = models.Sequential([
        Input(shape=input_shape),

        layers.Conv2D(hp.Choice("conv 1", [32, 64]), (3,3), activation='relu'),
        layers.MaxPooling2D((3,3)),

        layers.Conv2D(hp.Choice("conv 3", [32, 64]),(3,3), activation='relu'),

        layers.Flatten(),
        layers.Dense(hp.Choice("dense", [32, 64, 128]), activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])

    model.compile(
        optimizer = keras.optimizers.Adam(hp.Choice('learning_rate', values=[1e-2, 1e-3])),
        loss = 'binary_crossentropy',
        metrics = ['binary_accuracy', 'accuracy', zero_one_loss]
    )

    return model
```

For the second CNN model, the same parameters as the simple CNN model were chosen, but also the dropout probability:

```
def build_model_cnn1(hp):
    dropouts = hp.Float("dropout_flatten_layer", min_value = 0.2, max_value = 0.5, step = 0.1)

    model = models.Sequential([
        Input(shape=input_shape),

        layers.Conv2D(
            filters = hp.Int('conv_1_filter', min_value=32, max_value=128, step=32),
            kernel_size = hp.Choice('conv_2_kernel', values = [3,5]),
            activation = 'relu'
        ),
        layers.Dropout(dropouts),

        layers.Conv2D(
            filters = hp.Int('conv_2_filter', min_value=32, max_value=64, step=16),
            kernel_size = hp.Choice('conv_2_kernel', values = [3,5]),
            activation = 'relu'
        ),
        layers.MaxPooling2D((2,2)),
        layers.Dropout(dropouts),

        layers.Conv2D(
            filters = hp.Int('conv_3_filter', min_value=16, max_value=64, step=16),
            kernel_size = hp.Choice('conv_2_kernel', values = [3,5]),
            activation = 'relu'
        ),
        layers.Dropout(dropouts),

        layers.Flatten(),
        layers.Dense(
            units = hp.Int('dense_1_units', min_value=32, max_value=128, step=16),
            activation = 'relu'
        ),
        layers.Dense(
            units = hp.Int('dense_2_units', min_value=16, max_value=32, step=16),
            activation = 'relu'
        ),
```

```
38
39
40          layers.Dense(1, activation='sigmoid')
41      ])
42
43      model.compile(
44          optimizer = keras.optimizers.Adam(hp.Choice('learning_rate
    ', values=[1e-2, 1e-3])),
45          loss = 'binary_crossentropy',
46          metrics = ['binary_accuracy', 'accuracy', zero_one_loss]
47      )
48
49      return model
```

As previously mentioned KerasTuned was used to calculate the RandomSearch of the hyperparameters. This method, as suggested by the name, samples random points in the search space. GridSearch would probably have found better hyperparameter combinations, but was removed afer some experimentation. The reason being that an exaustive search of all the possible hyperparameters would have take an extremely long time to execute, also trying combinations very similar to each other. RandomSearch, on the other hand, explores more the search space, and is more likely to find better combinations by chance.
For the RandomSearch these parameters were chosen:

- *validation_accuracy* as the metric to optimize;
- 50 maximum trials to find the best combination of parameters;
- 20 epochs of training per trial.

```
1  def tune(model_name, model_builder):
2      tuner = keras_tuner.RandomSearch(
3          model_builder,
4          project_name = f"project {model_name}",
5          objective='val_accuracy',
6          distribution_strategy=tf.distribute.MirroredStrategy(),
7          max_trials=50,
8          overwrite = True
9          )
10
11     tuner.search(reduced_X_train, reduced_y_train, epochs=20,
    validation_data=(reduced_X_test, reduced_y_test))
12     return tuner
```

After searching, these were the best found parameters:

For the MLP model:

```
{'dropout': 0.30000000000000004,
 'dense_1': 128,
 'dense_2': 128,
 'dense_3': 117,
 'learning_rate': 0.001}
```

With a best val_accuracy of 0.7702702879905701, and 00h 44m 51s of processing.

For the first CNN model:

```
{'dropout': 0.30000000000000004,
 'conv 1': 32,
 'conv 3': 32,
 'dense': 32,
 'learning_rate': 0.001}
```

With a best val_accuracy of 0.8800675868988037, and 00h 46m 18s of processing.

And, at last, for the bigger CNN model:

```
{'dropout': 0.30000000000000004,
 'conv_1_filter': 32,
 'conv_2_kernel': 3,
 'conv_2_filter': 32,
 'conv_3_filter': 64,
 'dense_1_units': 96,
 'dense_2_units': 32,
 'learning_rate': 0.001}
```

After more than 2 hours of processing.

It's important to point out that given the random nature of the approach and the relatively low number of trials the results will hardly be optimal, and could be worse in real world applications than the default parameters. With greater resources and time it's possibile to find better hyperparameters, either by increasing the number of trials or by using an exaustive search tecnique like GridSearch.

Chapter 6

K-fold cross validation

K-fold cross validation is a validation technique that consists in dividing the dataset into K subsets, also called folds. Iteratively the model is trained K times, using a different training set and test set in each iteration. In particular each time a fold is chosen, which will be used as a test set, while the remaining part of the dataset will be used for training. Using this technique it's possible to verify the capabilities of the architecture by exposing it to a greater number of different images in the training and test sets.
Specifically, a 5-fold cross validation was used in this project.
In figure 4 it is possible to see how the folds are distributed on the dataset used in this project. Each column represents an iteration of the algorithm: the line represents the dataset, its red portion represents the current fold, that is the test set, while the remaining blue part represents the training set.
To implement the algorithm *sklearn.model_selection.KFold()* was used. This method requires the number of folds, and after having applied the datapoints and labels of the whole dataset returns, for each iteration, the indexes of the dataset which will be part of the training set and which will be part of the test set as follows:

```
Fold 0:
  Train: index=[1184 1185 1186 ... 5914 5915 5916]
  Test:  index=[   0    1    2 ... 1181 1182 1183]
  Train indexes:  1184 ->  5916
  Test indexes:      0 ->  1183

Fold 1:
  Train: index=[   0    1    2 ... 5914 5915 5916]
  Test:  index=[1184 1185 1186 ... 2365 2366 2367]
  Train indexes:     0 ->  5916
  Test indexes:   1184 ->  2367

Fold 2:
  Train: index=[   0    1    2 ... 5914 5915 5916]
```

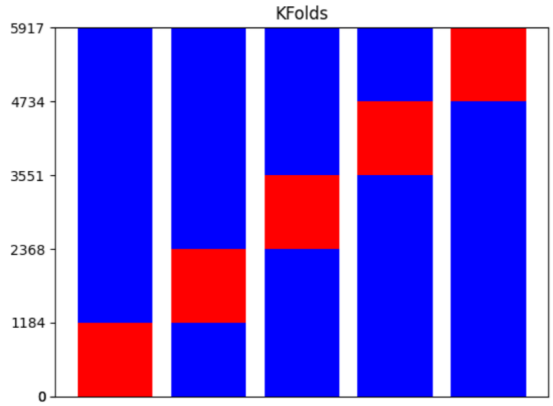

Figure 4: Distribution of the folds in the dataset

```
  Test:  index=[2368 2369 2370 ... 3548 3549 3550]
  Train indexes:     0 ->  5916
  Test indexes:   2368 ->  3550

Fold 3:
  Train: index=[   0    1    2 ... 5914 5915 5916]
  Test:  index=[3551 3552 3553 ... 4731 4732 4733]
  Train indexes:     0 ->  5916
  Test indexes:   3551 ->  4733

Fold 4:
  Train: index=[   0    1    2 ... 4731 4732 4733]
  Test:  index=[4734 4735 4736 ... 5914 5915 5916]
  Train indexes:     0 ->  4733
  Test indexes:   4734 ->  5916
```

The algorithm implementation used in the project is as follows:

```
def k_fold_cross_validation(model_json, X, y, num_folds,
    num_epochs, verbose=True):
     verbosity = 1 if verbose else 0
     show_plot = True if verbose else False

     folds = []
     histories = []

     for k, (training_index, test_index) in enumerate(KFold(
    n_splits = num_folds).split(X, y)):
          model = models.model_from_json(model_json)


          print(f'-> Fold {k+1}/{num_folds}')

```

```
14          if verbose: print("  splitting...")
15
16          # splitting
17          X_train, X_test = X[training_index], X[test_index]
18          y_train, y_test = y[training_index], y[test_index]
19
20          if verbose: print("  training...")
21          # training
22          history = model.fit(
23              X_train,
24              y_train,
25              validation_data = (X_test, y_test),
26              epochs = num_epochs,
27              verbose = verbosity,
28              shuffle='batch',
29              callbacks = callbacks.EarlyStopping(
30                      monitor='val_loss',
31                      patience=10,
32                      restore_best_weights=True
33                  )
34          )
35
36          # scores
37          evaluation = model.evaluate(X_test, y_test, return_dict=
    True)
38          evaluation['fold_nr'] = k+1
39          folds.append(evaluation)
40
41          current_history = history.history
42          current_history['fold_nr'] = k+1
43          histories.append(current_history)
44
45          if(show_plot):
46              fig, axes = plt.subplots(nrows=1, ncols=3, figsize
    =(10, 3))
47              history_frame = pd.DataFrame(history.history)
48              history_frame.loc[:, ['loss', 'val_loss']].plot(title=
    'Losses', ax=axes[0])
49              history_frame.loc[:, ['binary_accuracy', '
    val_binary_accuracy']].plot(title="Binary accuracy", ax=axes
    [1])
50              history_frame.loc[:, ['zero_one_loss', '
    val_zero_one_loss']].plot(title="zero-one loss", ax=axes[2])
51              plt.tight_layout(pad=2.0)
52              plt.show()
53
54      return folds, histories
```

This algorithm allows, given a model, the dataset, a number of folds and a maximum number of epochs, to obtain all the evaluations on the folds and to show the graphs of how the various metrics and losses evolve during learning.
Furthermore, a callback has been inserted in the Keras fitting method, responsible for stopping the training when it is no longer bringing significant improvements on the validation loss. The callback stops execution and returns the model that had the least validation loss. This way we have an automatic system of adjusting the number of epochs based on the moment the model starts to overfit.
One detail that may be relevant is that the algorithm does not take as a parameter a compiled model, but asks for its JSON representation. That's because at each fold the model needs to be rebuilt. From one of the last versions of Keras skipping this step would not start a new fit, but would start building on the weights of the previous one.

Chapter 7

Experiments

In this chapter several experiments will be reported based on the techniques presented in previous chapters. Initially, the capabilities of the models will be evaluated using a simple training technique, while subsequently, 5-fold cross validation will be used to verify the robustness of the best models.

7.1 Training

To carry out the preliminary tests and make them faster, the Keras *fit* method was used:

```
batch_size = 32

history = model.fit(
                X_train,
                y_train,
                validation_data = (X_test, y_test),
                epochs = num_epochs,
                verbose = verbosity,
                batch_size = batch_size,
                shuffle='batch',
                callbacks = callbacks.EarlyStopping(
                        monitor='val_loss',
                        patience=10,
                        restore_best_weights=True
                    )
            )
```

It's important to note the use of batches and how a shuffle is applied to them before they are used. As explained in the chapter 6, the EarlyStopping callback has also been inserted, responsible for preventing overfitting and restoring the best weights with respect to the validation loss.

7.2 Zero-one loss

One of the metrics used in this project is the zero-one loss. This is one of the simplest metrics: it gives 0 if the classification is correct and 1 otherwise.

$$\text{zero-one loss} = \frac{1}{n}\sum_i^n (\hat{y}_i - y_i)^2$$

Where n is the number of data points and $\hat{y}_i$ is the i-th prediction. Since the classification is binary the squared difference will be 1 if y_i and $\hat{y}_i$ are different, 0 otherwise. By averaging the errors we can obtain our metric.
To be able to use it with keras it was necessary to create a function that implements it:

```
def zero_one_loss(y_true, y_pred):
    return tf.reduce_mean(tf.square(tf.round(y_pred) - y_true))
```

TensorFlow functions were used to allow Keras to apply all the optimizations that can be applied during training so it won't fall back on op-by-op mode.

7.3 Fist training

The first training was done to the initial models and dataset without any modifications.

7.3.1 MLP

In figure 5 are shown the graphs and the following are the metrics:

```
{
  "binary_accuracy": 0.7650042176246643,
  "loss": 0.5132174491882324,
  "zero_one_loss": 0.23512423038482666
 }
```

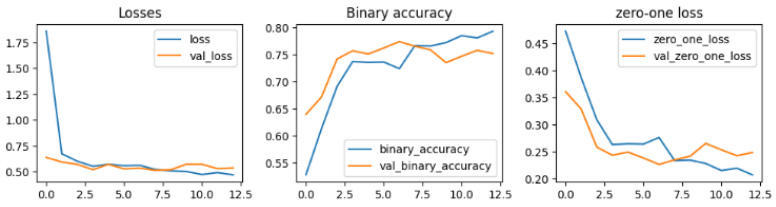

Figure 5: Performance of the Multi Layer Perceptron model

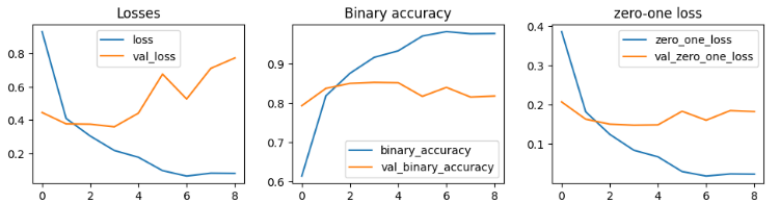

Figure 6: Performance of the CNN model

7.3.2 CNN

In figure 6 are shown the graphs and the following are the metrics:

```
{
  "binary_accuracy": 0.8520709872245789,
  "loss": 0.35923945903778076,
  "zero_one_loss": 0.14813099801540375
 }
```

7.3.3 Large CNN

In figure 7 are shown the graphs and the following are the metrics:

```
{
  "binary_accuracy": 0.9171597361564636,
  "loss": 0.23139923810958862,
  "zero_one_loss": 0.08293373882770538
 }
```

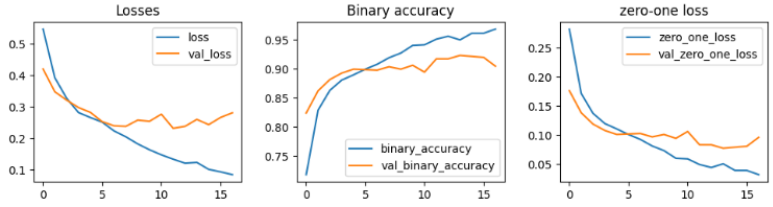

Figure 7: Performance of the larger CNN model

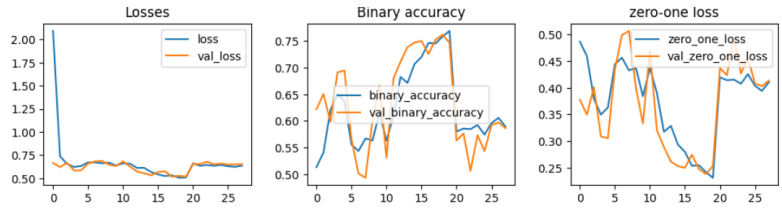

Figure 8: Performance of the Multi Layer Perceptron model after augmentation

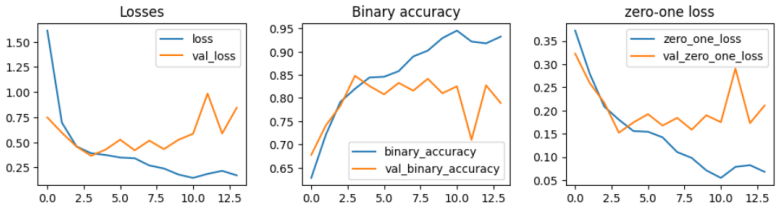

Figure 9: Performance of the CNN model after augmentation

7.4 Training after augmentation

The second training was done with the initial models and dataset with the augmented datapoints.

7.4.1 MLP

In figure 8 are shown the graphs and the following are the metrics:

```
{
  "binary_accuracy": 0.7525352239608765,
  "loss": 0.5182831287384033,
  "zero_one_loss": 0.24748632311820984
 }
```

7.4.2 CNN

In figure 9 are shown the graphs and the following are the metrics:

```
{
  "binary_accuracy": 0.8478873372077942,
  "loss": 0.3653579354286194,
  "zero_one_loss": 0.15216779708862305
 }
```

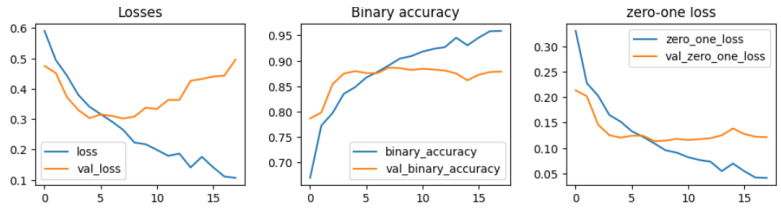

Figure 10: Performance of the larger CNN model after augmentation

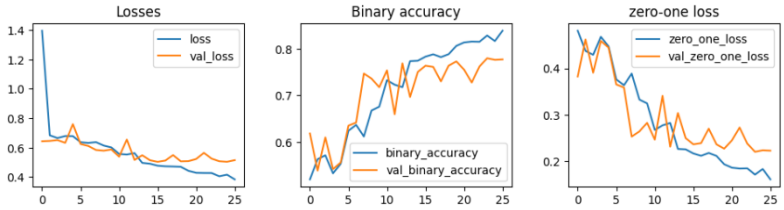

Figure 11: Performance of the Multi Layer Perceptron model after hyperparameter tuning

7.4.3 Large CNN

In figure 10 are shown the graphs and the following are the metrics:

```
{
  "binary_accuracy": 0.8866197466850281,
  "loss": 0.30190277099609375,
  "zero_one_loss": 0.11347731947898865
 }
```

7.5 Training after Hyperparameter tuning

The third training was done to the tuned models and the not-augmented dataset.

7.5.1 MLP

In figure 11 are shown the graphs and the following are the metrics:

```
{
  "accuracy": 0.7540152072906494,
  "binary_accuracy": 0.7540152072906494,
  "loss": 0.5146805644035339,
  "zero_one_loss": 0.24585875868797302
}
```

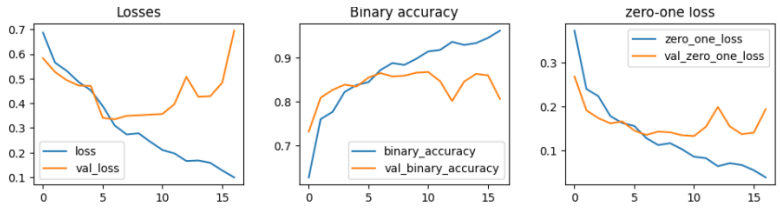

Figure 12: Performance of the CNN model after hyperparameter tuning

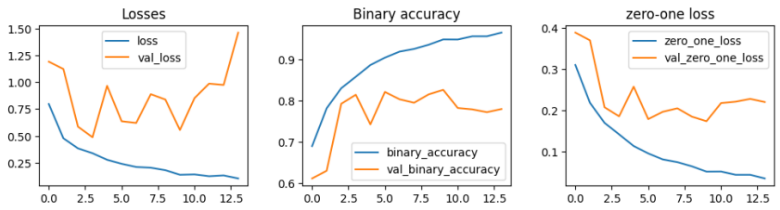

Figure 13: Performance of the larger CNN model after hyperparameter tuning

7.5.2 CNN

In figure 12 are shown the graphs and the following are the metrics:

```
{
  "accuracy": 0.8622146844863892,
  "binary_accuracy": 0.8622146844863892,
  "loss": 0.3359849452972412,
  "zero_one_loss": 0.13775065541267395
 }
```

7.5.3 Large CNN

In figure 13 are shown the graphs and the following are the metrics:

```
{
  "accuracy": 0.8165680766105652,
  "binary_accuracy": 0.8165680766105652,
  "loss": 0.6330058574676514,
  "zero_one_loss": 0.18344049155712128
 }
```

7.6 5-Fold cross validation

The 5-Fold cross validation was done to the most promising models: the initial ones.

It's important to underline that the graphs show a lot of overfitting, but that's fine thanks to the parameter *restore_best_weights* of the EarlyStopping callback.

7.6.1 MLP

In figure 14 are shown the graphs of the folds and the following are the metrics:


```
{
  "binary_accuracy": 0.7618243098258972,
  "loss": 0.4889039695262909,
  "zero_one_loss": 0.2381756752729416,
  "fold_nr": 1
 },
 {
  "binary_accuracy": 0.787162184715271,
  "loss": 0.48280003666877747,
  "zero_one_loss": 0.2128378450870514,
  "fold_nr": 2
 },
 {
  "binary_accuracy": 0.7514792680740356,
  "loss": 0.5155483484268188,
  "zero_one_loss": 0.24850152432918549,
  "fold_nr": 3
 },
 {
  "binary_accuracy": 0.7886728644371033,
  "loss": 0.47827431559562683,
  "zero_one_loss": 0.21123038232326508,
  "fold_nr": 4
 },
 {
  "binary_accuracy": 0.761622965335846,
  "loss": 0.5095245242118835,
  "zero_one_loss": 0.238393634557724,
  "fold_nr": 5
 }
```

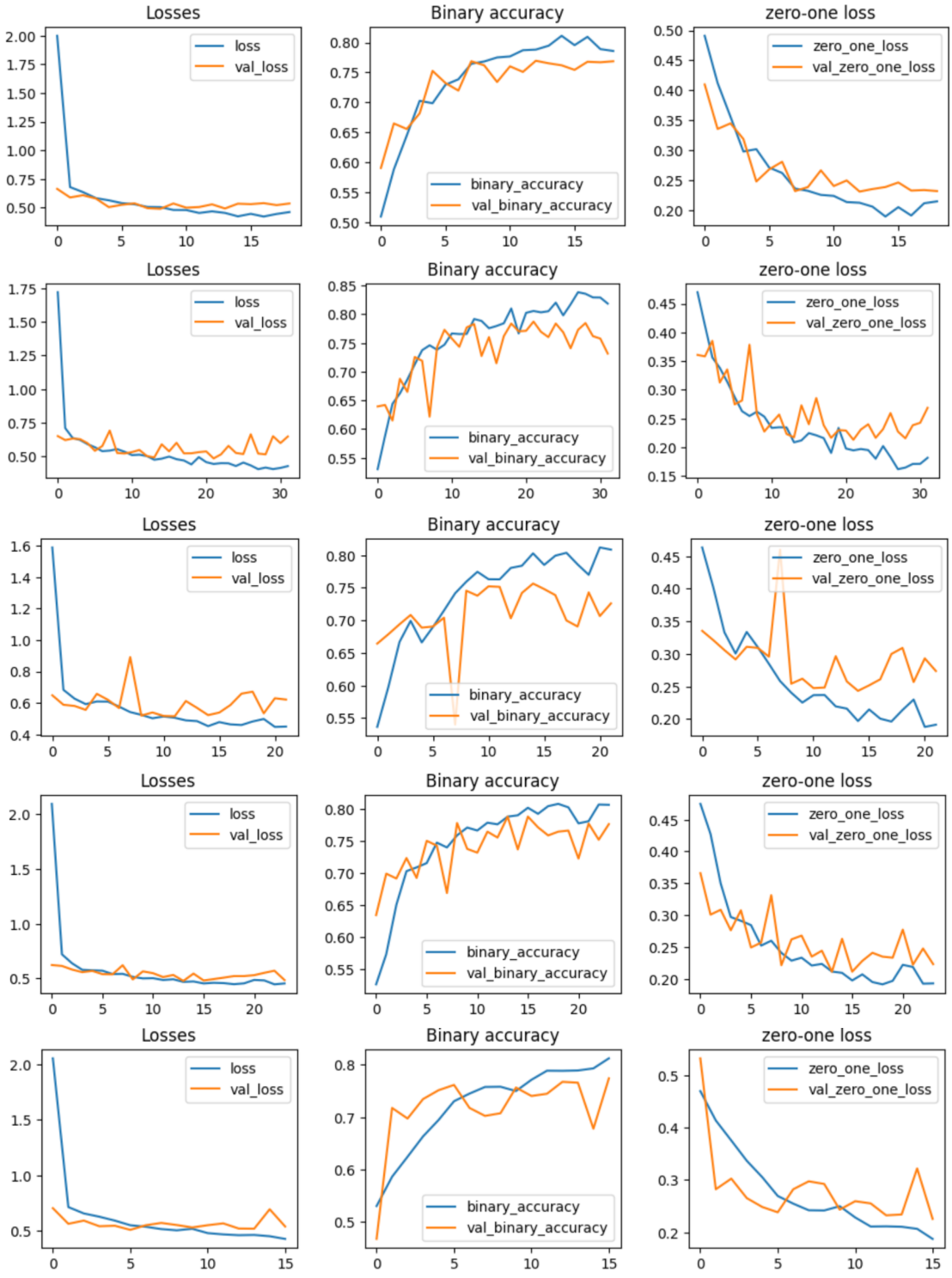

Figure 14: Folds of MLP (in order)

7.6.2 CNN

In figure 15 are shown the graphs of the folds and the following are the metrics:


```
{
 "binary_accuracy": 0.8631756901741028,
 "loss": 0.338460773229599,
 "zero_one_loss": 0.1368243247270584,
 "fold_nr": 1
},
{
 "binary_accuracy": 0.8682432174682617,
 "loss": 0.3290727138519287,
 "zero_one_loss": 0.1317567527294159,
 "fold_nr": 2
},
{
 "binary_accuracy": 0.8596788048744202,
 "loss": 0.34884706139564514,
 "zero_one_loss": 0.14025719463825226,
 "fold_nr": 3
},
{
 "binary_accuracy": 0.8402366638183594,
 "loss": 0.39834722876548767,
 "zero_one_loss": 0.1597646027803421,
 "fold_nr": 4
},
{
 "binary_accuracy": 0.8258664608001709,
 "loss": 0.6515231132507324,
 "zero_one_loss": 0.17423169314861298,
 "fold_nr": 5
}
```

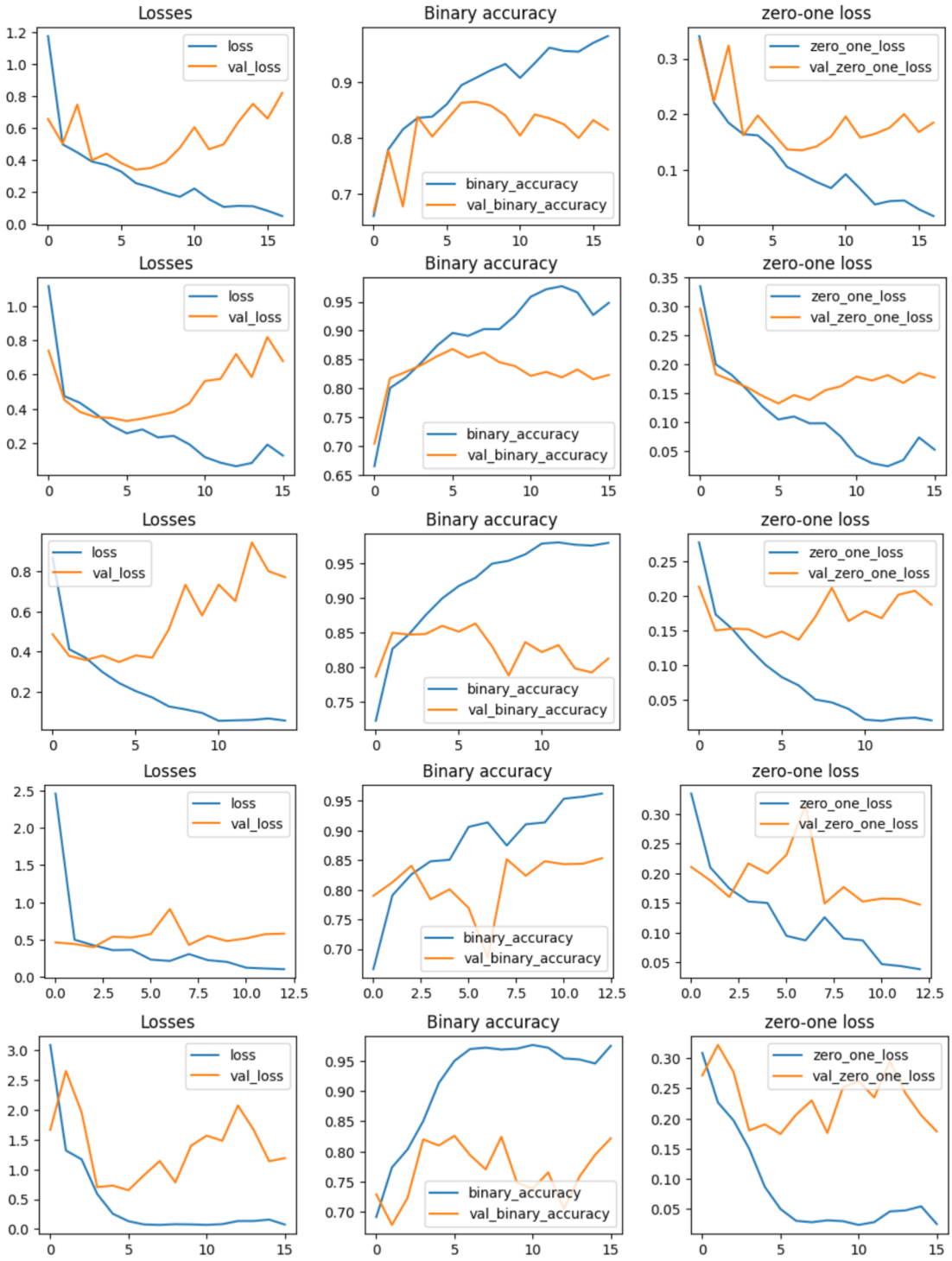

Figure 15: FoldS of CNN (in order)

7.6.3 Large CNN

In figure 16 are shown the graphs and the following are the metrics:


```
{
  "binary_accuracy": 0.9045608043670654,
  "loss": 0.2435985803604126,
  "zero_one_loss": 0.09543918818235397,
  "fold_nr": 1
 },
 {
  "binary_accuracy": 0.9104729890823364,
  "loss": 0.26119473576545715,
  "zero_one_loss": 0.08952702581882477,
  "fold_nr": 2
 },
 {
  "binary_accuracy": 0.8892645835876465,
  "loss": 0.2817454934120178,
  "zero_one_loss": 0.11064188927412033,
  "fold_nr": 3
 },
 {
  "binary_accuracy": 0.88165682554245,
  "loss": 0.2985360026359558,
  "zero_one_loss": 0.11829773336648941,
  "fold_nr": 4
 },
 {
  "binary_accuracy": 0.9087066650390625,
  "loss": 0.24582195281982422,
  "zero_one_loss": 0.0913524404168129,
  "fold_nr": 5
 }
```

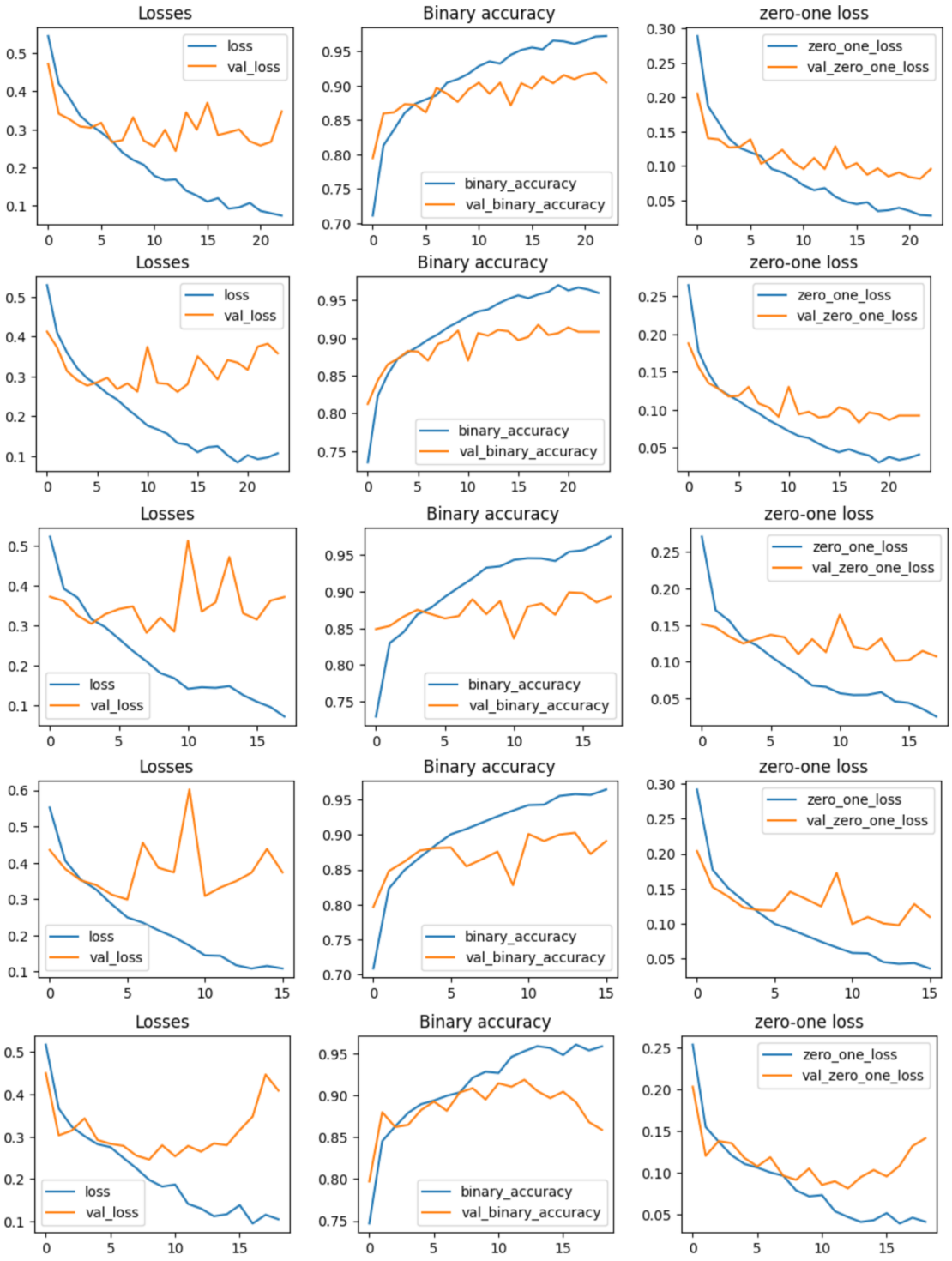

Figure 16: Folds of the bigger CNN model (in order)

Chapter 8

Discussion

As shown in the experiments, the model that performs best is in any case the one defined manually, trained without the augmented dataset. It's interesting that the model trained with the augmented dataset performs worse than the default model, and the reasons could be the following:

- The models may be too simple to effectively capture the features of the augmented images;
- The augmentation may not bring examples that are different enough from the original images, pushing the model to overfit more.

Similarly, the hyperparameter search produced models worse than the default ones. Particurly, the tuned larger CNN model performs worse than the tuned smaller CNN model. The reason is that our search methodology, RandomSearch, is likely to converge to a local minimum, as it searches only a subset of the possible hyperparameter combinations. With better resources and more time it would be possible to carry out an exhaustive search, for example using GridSearch, which would most likely produce the best possible model.
Furthermore, it is possible to notice that the multilayer perceptron performs much worse than other architectures, this suggests that the structure is too simple to be able to correctly extrapolate the information necessary to make acceptable predictions.
It is also interesting to note that the bigger CNN model performs better than the smaller CNN model, thus showing that the additional layers, although complicating the architecture, were able to make a positive change. It's also relevant to notice how the same thing is not true if the hyperparameters are not chosen carefully: the bigger CNN model performs worse compared to the simple CNN model after hyperparameter tuning, proving that a bigger model does not necessairily mean better predictions, and that there are a lot of things to consider when

expanding a model. Either way, both convolutional networks obtain acceptable results, albeit significantly better than the MLP model.
Finally, as it's possbile to notice from the section dedicated to 5-fold cross validation (7.6) all models are fairly stable and have the same performance regardless of the traing and test sets.