UNIVERSITÀ DEGLI STUDI DI MILANO
FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

Master in
Computer Science

STATISTICAL METHODS FOR MACHINE LEARNING
FINAL REPORT ON NEURAL NETWORKS FOR THE BINARY CLASSIFICATION

Teacher: Prof. Nicolò Cesa-Bianchi

Final report written by:
Samuele Simone
Matr. Nr. 11910A

Academic year 2022-2023

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

The purpose of this project was to discover and work on different neural network models in order to solve an image classification problem. Specifically the recognition of chihuahuas from muffins, which because of the similarities, turns out to be a non-trivial task for a machine to perform. In the following chapters we will go into detail about the various models and through experiments and graphs figure out which model will work best to solve this task.

Chapter 2

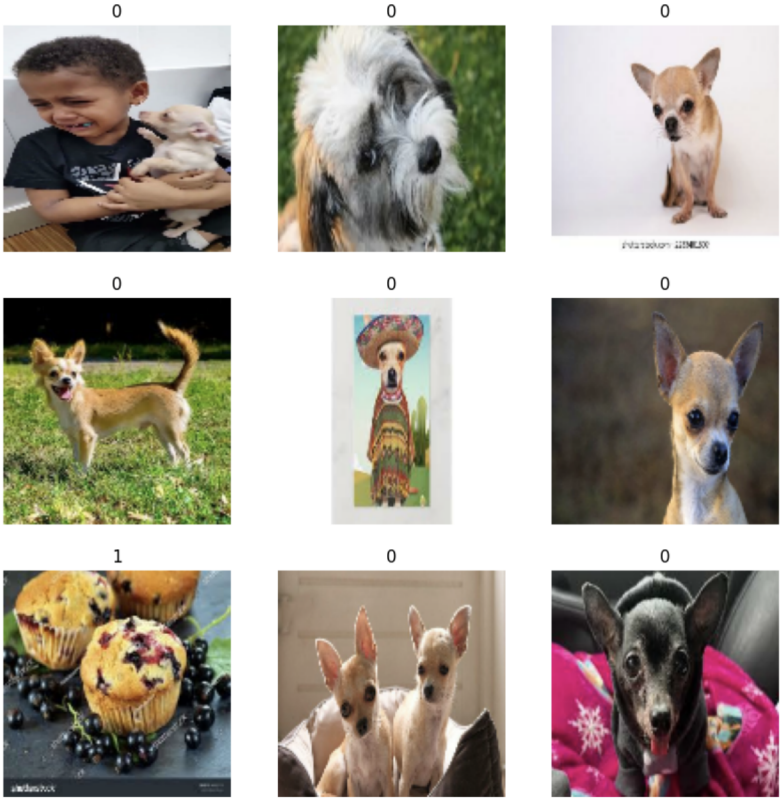
Dataset

The dataset under consideration is called "Muffin vs Chihuahua", taken from Kaggle [1]. It consists of about 6000 images taken from Google Images. Duplicate images have been removed. The first operation that was performed was to download the dataset through the Kaggle API and divide the dataset into training set and test set. After that, thanks to the Tensorflow library and specifically Keras, the images were loaded so that the neural networks could be trained. Here the code for the train_images loading using `tf.keras.utils.image_dataset_from_directory`

```
train_images = tf.keras.utils.image_dataset_from_directory(
    train_folder,
    labels="inferred",
    label_mode="int",
    class_names=None,
    color_mode="rgb",
    batch_size=32,
    image_size=(128, 128),
    shuffle=True,
    seed=None,
    validation_split=None,
    subset=None,
    interpolation="bilinear",
    follow_links=False,
    crop_to_aspect_ratio=False,
)
```

As you can see there are different properties such as `label_mode = int`, so labels as integers. Or `color_mode ="rgb"` in which the images are working in this red blue green space. By making these parameters explicit, it is possible to obtain a more custom configuration that suits our needs. The same process was applied for the test_images. To give an idea to the reader, here a sample of images found within the dataset: As you can see muffin label is 0 and chihuahua label is 1.

Figure 1: A sample of images found within the dataset



2.1 Preprocessing

Certainly when dealing with data it is good to always make some improvement such as cleaning the data so that you have better results. In fact, the dataset was delivered without having duplicate images, which already represents a first step of preprocessing. In addition, I applied the **Data augmentation** [2] that is a technique in machine learning used to reduce overfitting when training a machine learning model, by training models on several slightly-modified copies of existing data. With the help of Keras, it is possible to achieve date augmentation in the following way:

```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
    ]
)
```

Here is graphically what the transformation looks like: In order to complete the

Figure 2: Data augmentation applied in a specific image

preprocessing I applied the data augmentation transformation to the input and I

performed normalization of pixel values in the range [0, 255] to values in the range [0, 1], indeed this helps stabilize the training of the model. To conclude, this line `train_images = train_images.prefetch(tf.data.AUTOTUNE)` optimizes the data loading process during model training, allowing the model to work more efficiently and reducing the waiting time between data batches.

Chapter 3

Neural Network Models

In this chapter we are going to examine the models that have been made in order to solve the binary classification task. All models were trained using the principle of K-Fold cross validation. In this case the `K = 5`. For the results of the experiments please refer to the Chapter 4 while for the discussion of the experiments please refer to the Chapter **??**.

3.1 K-fold cross validation (K = 5)

Cross-validation is a statistical method used to estimate the skill of machine learning models.The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called k-fold cross-validation. When a specific value for k is chosen, it may be used in place of k in the reference to the model such as k=5 becoming 5-fold cross-validation, the one used in this project. [3]. The general rules are the following:

1. Shuffle the dataset randomly
2. Split the dataset into k groups
3. For each group:
    - Take the group as a hold out or test data set
    - Take the remaining groups as a training data set
    - Fit a model on the training set and evaluate it on the test set
    - Retain the evaluation score and discard the model
4. Summarize the skill of the model using the sample of model evaluation scores

Let's take a look at its mathematical formalization. Let S be our entire dataset [4]. We partition S in K subsets (also known as folds) $S_1, \ldots, S_K$ of size m/K each (assume for simplicity that K divides m).The K-fold CV estimate of $\mathbb{E}\left[\ell_{\mathcal{D}}(A)\right]$ on S, denoted by $\ell_S^{\text{cv}}(A)$, is then computed as follows: we run A on each training part S_{-i} of the folds $i = 1, \ldots, K$ and obtain the predictors $h_1 = A\left(S_{-i}\right), \ldots, h_K = A\left(S_{-K}\right)$. We then compute the (rescaled) errors on the testing part of each fold,

$$\ell_{S_i}\left(h_i\right) = \frac{K}{m} \sum_{(\boldsymbol{x},y)\in S_i} \ell\left(y, h_i(\boldsymbol{x})\right)$$

Finally, we compute the CV estimate by averaging these errors

$$\ell_S^{\text{cv}}(A) = \frac{1}{K} \sum_{i=1}^{K} \ell_{S_i}\left(h_i\right)$$

3.2 Multi-Layer Perceptron (MLP)

A multilayer perceptron (MLP) is a feedforward artificial neural network, consisting of fully connected neurons with a nonlinear kind of activation function, organized in at least three layers, notable for being able to distinguish data that is not linearly separable. [5] During the project, I developed 3 different MLP models so that I could conduct different experiments and understand the performance of them.

3.2.1 First MLP Model

Here are the MLP architecture:

- `inputs = tf.keras.Input(shape=input_shape)`: model's input with the input_shape size. In this case (128, 128, 3)
- `x = layers.Flatten()(inputs)`: Converts 2D input (an image) to a 1D vector
- `x = layers.Dense(256, activation='relu')(x)`: Fully connected layer (dense) with 256 neurons and ReLU activation
- `x = layers.Dropout(0.5)(x)`: This dropout layer introduces regularization into the network. The parameter 0.5 indicates that 50% of the neurons exiting this layer are randomly switched off during training.
- `x = layers.Dense(128, activation='relu')(x)`: Other fully connected layer with 128 neurons and ReLU activation.

- `x = layers.Dropout(0.5)(x)`: Another dropout layer for further regularization.
- `activation = 'sigmoid'`: Here the activation function for the last layer is specified. In your case, you are using 'sigmoid,' which is commonly used in binary classification problems where the output must be a probability between 0 and 1 for each class.
- `outputs = layers.Dense(num_classes, activation=activation)(x)`. num_classes = 1. This is the last layer of the model, which returns the output. Then these are the technical details of the setup:
  - `batch_size = 32`. Indicates how many data examples are processed together before updating the weights.
  - `optimizer='adam'`. The optimizer is the algorithm that adjusts the model weights during training to minimize the cost function. It is a stochastic gradient descent method.
  - `loss = 'binary_crossentropy'`. It specifies the cost function (or loss function) that will be used to evaluate how well the model is fitting the data during training. This cross-entropy loss is user for binary (0 or 1) classification applications.
  - `metrics=[zero_one_loss_func]`: this param specifies the evaluation metrics that will be calculated during model training. In this case I used a custom function called `zero_one_loss_func`. It requires two parameters: `y_true` , the real associated label and the `y_pred`, the ones predicted by the model. Then:
    - Transforms predicted probabilities into binary labels (0 or 1)
    - Compare the predicted binary labels with the actual labels
    - Calculate the zero-one loss as the average of the errors.

3.2.2 Hypertuning parameter for MLP

Hyper-parameters are parameters that are not directly learnt within estimators. In order to obtain better results I decided to do an hypertuning of the params using the **GridSearchCV** from the library `sklearn.model_selection`. GridSearchCV exhaustively considers all parameter combinations from a paramater grid [6]. Due the amount of the time spent for the exhaustive combination I choose just two different learning rate `0.001` and `0.01`. After wrapping the mlp keras into a KerasClassifier I can integrate the scikit-learn flow and run GridSearch. Here the results:

```
Highest score is 0.5403 with {'learning_rate': 0.001, 'optimizer':
    <class 'keras.optimizers.adam.Adam'>}

List in descending order:
0.5403 --> {'learning_rate': 0.001, 'optimizer': <class 'keras.
    optimizers.adam.Adam'>}
0.5119 --> {'learning_rate': 0.01, 'optimizer': <class 'keras.
    optimizers.adam.Adam'>}
```

So in this case with a learning rate of `0.001` the accuracy is better.

3.2.3 First MLP Model after Hypertuning

I trained the same mlp but with different learning rate as discussed above. The results are quite similar as you can see from the Table 2 so I spent some times to create other two mlp architecture.

3.2.4 Second MLP Model

In the second MLP Model I added two fully connected layer(dense) after the 0.5 Dropout of the first MLP Model.

Layer (type)	**Output Shape**	**Param #**
input_7 (InputLayer)	[(None, 128, 128, 3)]	0
flatten_5 (Flatten)	(None, 49152)	0
dense_15 (Dense)	(None, 256)	12,583,168
dropout_10 (Dropout)	(None, 256)	0
dense_16 (Dense)	(None, 128)	32,896
dropout_11 (Dropout)	(None, 128)	0
dense_17 (Dense)	(None, 64)	8,256
dropout_12 (Dropout)	(None, 64)	0
dense_18 (Dense)	(None, 32)	2,080
dropout_13 (Dropout)	(None, 32)	0
dense_19 (Dense)	(None, 1)	33
Total params: 12,626,433		
Trainable params: 12,626,433		
Non-trainable params: 0		

Also in this case, the performance are not so high as reported in the Table. I executed a 10 epoch training and here the results:

Epoch	**Loss**	**Accuracy**
1/10	1083.8823	0.4843
2/10	10.3901	0.5398
3/10	9.8943	0.5405
4/10	9.4855	0.5398
5/10	9.1378	0.5400
6/10	8.7128	0.5405
7/10	8.3303	0.5407
8/10	7.9671	0.5407
9/10	7.6214	0.5407

3.2.5 Third MLP Model

The third MLP Model has 3 fully connected layers (512,256,128) with a Dropout for each layer of 0.3. Here the results:

Epoch	Loss	Accuracy
1	900.3068	0.5039
2	27.8657	0.5172
3	16.0389	0.5398
4	14.6262	0.5394
5	13.1948	0.5400
6	12.0603	0.5407
7	11.3178	0.5405
8	10.7011	0.5394
9	9.8406	0.5400
10	9.1993	0.5400

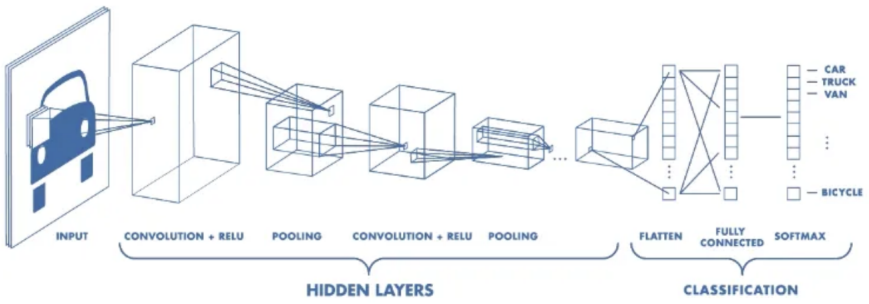
3.3 Convolutional Neural Network (CNN)

A **Convolutional Neural Network**, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be. [7]

3.3.1 CNN Model

During the project, I trained a CNN model to solve the binary classification problem. Here are the details of the architecture:

Figure 3: Example of CNN architecture



- `inputs = tf.keras.Input(shape=input_shape)`. Define a input layer for the model.
- `x = layers.Rescaling(1.0 / 255)(inputs)`. Rescaling for pixel normalization.
- `x = layers.Conv2D(128, 3, strides=2, padding="same", activation="relu")(x)`. This is a 2D convolution layer with 128 filters, a kernel size of 3x3, a stride of 2 (i.e., displacement of 2 pixels at a time), and a ReLU activation function. This layer extracts features from the image.
- `x = layers.MaxPooling2D(3, strides=2, padding="same")(x)`.This is a 2D max-pooling layer with a pool size of 3x3, a stride of 2, and equal "padding." Max-pooling reduces the size of the feature maps extracted from the convolution layer.
- `Middle block the same of the previous two step`.
- `x = layers.GlobalAveragePooling2D()(x)`.This layer performs "Global Average Pooling" on the resulting feature maps. Global Average Pooling calculates the average of the values in each feature map, producing a compact representation of the features.
- `x = layers.Dropout(0.5)(x)`. This is a dropout level that helps prevent overfitting. It randomly deactivates 50% of the neurons during training.
- `activation = "sigmoid"`.The activation function for the output layer is defined as "sigmoid," which is commonly used for binary classification problems.
- `outputs = layers.Dense(num_classes, activation=activation)(x)`.this is the output layer that produces the final predictions of the model. The

number of neurons corresponds to the number of classes specified by the num_classes parameter, and the activation function is "sigmoid" for the binary classification problem.

Then these are the technical details of the setup:

- `batch_size = 32`. Indicates how many data examples are processed together before updating the weights.
- `optimizer=keras.optimizers.Adam(1e-3)`. The optimizer is the algorithm that adjusts the model weights during training to minimize the cost function. It is a stochastic gradient descent method. I specified the learning rate to `0.001`
- `loss = 'binary_crossentropy'`. It specifies the cost function (or loss function) that will be used to evaluate how well the model is fitting the data during training. This cross-entropy loss is user for binary (0 or 1) classification applications.
- `metrics=[zero_one_loss_func]`: this param specifies the evaluation metrics that will be calculated during model training. In this case I used a custom function called `zero_one_loss_func`. It requires two parameters: `y_true` , the real associated label and the `y_pred`, the ones predicted by the model. Then:
  - Transforms predicted probabilities into binary labels (0 or 1)
  - Compare the predicted binary labels with the actual labels
  - Calculate the zero-one loss as the average of the errors.

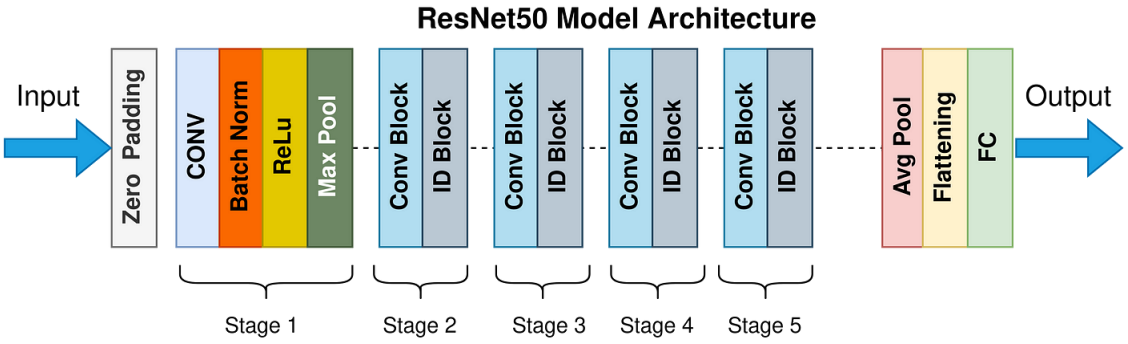
The result of the experiments is reported in the Table 3.

3.4 Deep Residual learning (ResNet50)

ResNet-50 is a convolutional neural network that is 50 layers deep. You can load a pretrained version of the neural network trained on more than a million images from the ImageNet database [8].The pretrained neural network can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. As a result, the neural network has learned rich feature representations for a wide range of images.The main innovation in ResNet is the introduction of **residual blocks**. In a residual block, the input is added to the output, allowing the model to learn the differences between the two rather than trying to learn the entire mapping. This makes training much more stable for very deep networks.ResNet-50

is often used for **transfer learning**.As a result, the model weights have already learned high-level representations that can be used for various computer vision tasks.

Figure 4: ResNet50 Architecture



3.4.1 ResNet50 Model

Here are the details of the architecture:

- `base_model = ResNet50(`
  `weights='imagenet',`
  `include_top=False,`
  `input_tensor=Input(shape=input_shape))`. Loads the ResNet50 pre-trained model.
  `weights='imagenet'` indicates that we want to use the pre-trained weights provided by ImageNet.
  `include_top=False` means that we do not want to include the fully connected layers (top layers) of the model, and
  `input_tensor=Input(shape=input_shape)` specifies the shape of the input tensor.

- `x = base_model.output`
  `x = GlobalAveragePooling2D()(x)`
  `x = Dense(256, activation='relu')(x)`
  `predictions = Dense(num_classes, activation='sigmoid')(x)`. It adds a Global Average Pooling layer and a fully connected layer. Here, the output of the ResNet50 model is passed through a 2D Global Average Pooling layer, which averages the spatial features for each channel. Then, a second fully connected layer with 256 units and ReLU activation is added, followed by an output layer with a number of units equal to num_classes and sigmoid

activation. The latter layer will return the probabilities associated with each class.

- `model = Model(inputs=base_model.input, outputs=predictions)`.Using the Model object, model inputs (the same inputs as the ResNet50 model) and outputs (the predictions tensor created in the previous step) are specified to create the final model.
- `for layer in base_model.layers: layer.trainable = False`.In this for loop, all layers of the basic ResNet50 model are iterated and their weights are set as non-trainable (they will not be updated during training of the new model). This is useful when using pre-trained weights for feature extraction and you want to prevent them from being overwritten during training. Then these are the technical details of the setup:
  - `batch_size = 32`. Indicates how many data examples are processed together before updating the weights.
  - `optimizer=keras.optimizers.Adam(1e-3)`. The optimizer is the algorithm that adjusts the model weights during training to minimize the cost function. It is a stochastic gradient descent method. I specified the learning rate to `0.001`
  - `loss = 'binary_crossentropy'`. It specifies the cost function (or loss function) that will be used to evaluate how well the model is fitting the data during training. This cross-entropy loss is user for binary (0 or 1) classification applications.
  - `metrics=['accuracy']`: this param specifies the evaluation metrics that will be calculated during model training.

Chapter 4

Experiments

Here the tables results for each experiment that I conducted:

Table 1: First MLP training results for each fold and epoch

Fold	Epoch	Loss	zero_one_loss_func	Val_Loss	Val_zero_one_loss_func
1	1	841.5640	0.4819	0.7778	0.4574
1	2	0.8091	0.4618	0.7777	0.4584
1	3	0.7503	0.4589	0.7795	0.4584
1	4	0.6898	0.4604	0.7803	0.4584
1	5	7.5421	0.4615	0.7684	0.4564
1	6	0.6897	0.4607	0.7778	0.4564
1	7	0.6899	0.4607	0.7784	0.4564
1	8	0.7288	0.4603	0.7786	0.4564
1	9	0.6897	0.4609	0.7786	0.4564
1	10	0.7136	0.4598	0.7782	0.4564
2	1	920.0229	0.4724	0.7597	0.4522
2	2	2.3125	0.4600	0.7584	0.4532
2	3	0.7153	0.4623	0.6895	0.4522
2	4	0.8337	0.4622	0.6886	0.4522
2	5	0.6901	0.4614	0.6891	0.4522
2	6	0.6899	0.4597	0.6890	0.4522
2	7	0.6898	0.4609	0.6890	0.4522
2	8	0.7029	0.4594	0.6890	0.4522
2	9	0.6902	0.4620	0.6890	0.4522
2	10	0.6903	0.4620	0.6890	0.4522
3	1	593.7564	0.4803	0.6951	0.4432

Continued on next page

Table 1 – Continued from previous page

Fold	Epoch	Loss	zero_one_loss_func	Val_Loss	Val_zero_one_loss_func
	2	3.0590	0.4624	0.6904	0.4432
	3	0.6901	0.4641	0.6895	0.4432
	4	0.6900	0.4641	0.6889	0.4432
	5	0.7200	0.4620	0.6885	0.4432
	6	0.6898	0.4618	0.6882	0.4432
	7	0.6895	0.4647	0.6881	0.4432
	8	0.6907	0.4638	0.6880	0.4432
	9	0.6897	0.4641	0.6880	0.4432
	10	0.6901	0.4635	0.6880	0.4432
4	1	907.8840	0.4852	0.7944	0.4707
	2	2.0251	0.4560	0.7090	0.4718
	3	0.6919	0.4543	0.7087	0.4707
	4	0.6900	0.4563	0.7087	0.4707
	5	0.6895	0.4548	0.7088	0.4707
	6	0.6896	0.4570	0.7089	0.4707
	7	0.6918	0.4550	0.7001	0.4697
	8	0.6896	0.4575	0.7002	0.4697
	9	0.6894	0.4560	0.7003	0.4697
	10	0.6893	0.4550	0.7003	0.4697
5	1	830.7402	0.5075	0.6967	0.4719
	2	1.6454	0.4563	0.6913	0.4708
	3	0.8036	0.4571	0.6909	0.4708
	4	0.6914	0.4576	0.6907	0.4708
	5	0.6896	0.4563	0.6907	0.4708
	6	0.6895	0.4553	0.6908	0.4708
	7	0.6895	0.4560	0.6908	0.4708
	8	0.6892	0.4571	0.6909	0.4708
	9	0.6893	0.4566	0.6910	0.4708
	10	0.6893	0.4563	0.6910	0.4708

Table 2: First MLP after hypertraining results for each fold and epoch

Fold	Epoch	Loss	zero_one_loss_func	Val_Loss	Val_zero_one_loss_func
1	1	6402.9297	0.4792	0.6918	0.4755
	2	20.6926	0.4580	0.6920	0.4755
	3	6.5519	0.4575	0.6919	0.4755

Continued on next page

Table 2 – Continued from previous page

Fold	Epoch	Loss	zero_one_loss_func	Val_Loss	Val_zero_one_loss_func
	4	4.8472	0.4567	0.6927	0.4755
	5	7.7787	0.4554	0.6933	0.4755
	6	6.7850	0.4577	0.6923	0.4755
	7	6.2771	0.4549	0.6919	0.4755
	8	3.6706	0.4567	0.6923	0.4755
	9	3.5163	0.4559	0.6924	0.4755
	10	2.6841	0.4552	0.6918	0.4755
2	1	4105.2227	0.4698	0.6890	0.4460
	2	18.9675	0.4612	0.6884	0.4460
	3	9.2161	0.4670	0.6878	0.4460
	4	7.5382	0.4630	0.6877	0.4460
	5	2.5178	0.4637	0.6878	0.4460
	6	4.8316	0.4659	0.6885	0.4460
	7	1.9393	0.4645	0.6874	0.4460
	8	3.9615	0.4638	0.6883	0.4460
	9	1.3100	0.4636	0.6876	0.4460
	10	2.0382	0.4631	0.6880	0.4460
3	1	4279.5918	0.4742	0.6892	0.4533
	2	13.3945	0.4664	0.6890	0.4533
	3	4.3161	0.4611	0.6892	0.4533
	4	9.3893	0.4650	0.6889	0.4533
	5	3.5593	0.4632	0.6888	0.4533
	6	3.7687	0.4584	0.6892	0.4533
	7	1.6367	0.4630	0.6888	0.4533
	8	1.7411	0.4600	0.6888	0.4533
	9	1.7792	0.4611	0.6887	0.4533
	10	1.8078	0.4595	0.6892	0.4533
4	1	5847.2910	0.4888	0.7012	0.5255
	2	18.9971	0.5070	0.6919	0.4745
	3	0.9379	0.4684	0.6917	0.4745
	4	0.6986	0.4563	0.6919	0.4745
	5	0.6900	0.4560	0.6923	0.4745
	6	0.6908	0.4555	0.6917	0.4745
	7	0.6889	0.4552	0.6924	0.4745
	8	0.6894	0.4565	0.6918	0.4745
	9	0.8700	0.4573	0.6921	0.4745
	10	0.6897	0.4575	0.6920	0.4745
5	1	6920.2822	0.5217	0.7348	0.4537

Continued on next page

Table 2 – Continued from previous page

Fold	Epoch	Loss	zero_one_loss_func	Val_Loss	Val_zero_one_loss_func
	2	0.7554	0.4621	0.6871	0.4537
	3	0.7348	0.4623	0.6888	0.4537
	4	0.8745	0.4623	0.6887	0.4537
	5	0.6921	0.4608	0.6884	0.4537
	6	0.6897	0.4608	0.6887	0.4537
	7	0.6899	0.4608	0.6885	0.4537
	8	0.6901	0.4630	0.6891	0.4537
	9	0.6973	0.4605	0.6886	0.4537
	10	0.6905	0.4633	0.6885	0.4537

Table 3: CNN training results for each fold and epoch

Fold	**Epoch**	**Loss**	**Zero-One Loss**	**Val Loss**	**Val Zero-One Loss**
1	1	0.6377	0.3554	0.5714	0.2873
	2	0.5656	0.2574	0.5050	0.2379
	3	0.5084	0.2201	0.4841	0.1914
	4	0.4641	0.1962	0.4494	0.2269
	5	0.4789	0.2112	0.5268	0.2297
	6	0.4630	0.1944	0.3895	0.1577
	7	0.4250	0.1747	0.3806	0.1577
	8	0.4113	0.1712	0.3621	0.1494
	9	0.4060	0.1697	0.4321	0.1792
	10	0.4003	0.1707	0.3355	0.1320
2	1	0.6440	0.3730	0.6200	0.3092
	2	0.5646	0.2619	0.5208	0.2438
	3	0.5163	0.2400	0.5359	0.2862
	4	0.4898	0.2184	0.5103	0.2622
	5	0.4666	0.2042	0.4583	0.1719
	6	0.4564	0.1928	0.4790	0.2299
	7	0.4502	0.1965	0.4241	0.1712
	8	0.4152	0.1761	0.4199	0.1626
	9	0.4045	0.1711	0.4113	0.1841
	10	0.4077	0.1727	0.4501	0.2206
3	1	0.6494	0.3746	0.5797	0.2698
	2	0.5625	0.2673	0.5442	0.2747
	3	0.5195	0.2351	0.4789	0.2098
	4	0.4979	0.2179	0.4449	0.1910

	5	0.4535	0.1954	0.4147	0.1535
	6	0.4456	0.1936	0.4026	0.1653
	7	0.4246	0.1732	0.3914	0.1639
	8	0.4340	0.1847	0.3959	0.1566
	9	0.4014	0.1629	0.4562	0.2286
	10	0.3970	0.1678	0.3694	0.1549
4	1	0.6432	0.3631	0.5496	0.2645
	2	0.5696	0.2663	0.5117	0.1941
	3	0.5180	0.2360	0.4826	0.1787
	4	0.4866	0.2074	0.4352	0.1756
	5	0.4693	0.2009	0.4211	0.1662
	6	0.4525	0.1919	0.4434	0.2037
	7	0.4295	0.1830	0.4131	0.1597
	8	0.4221	0.1842	0.3998	0.1587
	9	0.4010	0.1623	0.3793	0.1503
	10	0.4030	0.1696	0.3813	0.1576
5	1	0.6642	0.3964	0.6219	0.3389
	2	0.5809	0.2812	0.5462	0.2440
	3	0.5439	0.2500	0.5333	0.2583
	4	0.5147	0.2266	0.4992	0.1932
	5	0.4811	0.2050	0.4485	0.2135
	6	0.4596	0.1990	0.4555	0.2094
	7	0.4549	0.1940	0.4256	0.1764
	8	0.4268	0.1753	0.5045	0.2706
	9	0.4198	0.1791	0.4268	0.1660
	10	0.4197	0.1801	0.4139	0.1847

Bibliography

[1] Samuel Cortinhas. muffin-vs-chihuahua-image-classification. https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification, 2023.

[2] Wikipedia. Data augmentation. https://en.wikipedia.org/wiki/Data_augmentation, 2023.

[3] Jason Brownlee. A gentle introduction to k-fold cross-validation. https://machinelearningmastery.com/k-fold-cross-validation/, 2020.

[4] Nicolò Cesa-Bianchi. Hyperparameter tuning and risk estimates. https://cesa-bianchi.di.unimi.it/MSA/Notes/crossVal.pdf, 2023.

[5] Wikipedia. Multilayer perceptron. https://en.wikipedia.org/wiki/Multilayer_perceptron, 2023.

[6] Scikit Learn. Tuning the hyper-parameters of an estimator. https://scikit-learn.org/stable/modules/grid_search.html, 2023.

[7] Mayank Mishra. Convolutional neural networks, explained. https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939, 2020.

[8] Imagenet. http://www.image-net.org.