Statistical Methods for Machine Learning project

Edoardo Carrer
Matricola: 967549

December 2023

Contents

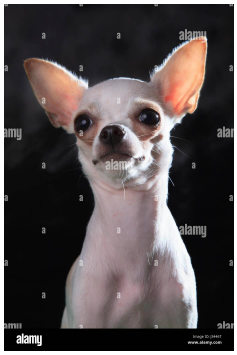
1 Introduction

In recent years, deep learning techniques have demonstrated remarkable performance in various computer vision tasks, including image classification. This report presents a performance analysis of 4 different neural network architectures, implemented with Keras, a type of predictors inspired by human brain biology with 9 pairs of training hyperparameters (3 batch sizes and 3 learning rates) via a binary classification problem based on an image dataset downloaded from Kaggle named muffin vs chihuahua. The project focuses on evaluating the impact of network architecture choices and hyperparameter tuning on the final estimate of cross-validation risk.

2 Methodology

2.1 Dataset and Data Pre-processing

The dataset used in this project is Kaggle's Muffin vs Chihuahua, consisting of 5917 images with 3199 images of chihuahuas (Figure 1) and 2718 images of muffins (Figure 2), of different sizes and types, as in the following examples.



(a) Chihuahua 1

(b) Chihuahua 2

(c) Chihuahua 3

Figure 1: dataset's images of Chihuahuas

(a) Muffin 1

(b) Muffin 2

(c) Muffin 3

Figure 2: dataset's images of Muffins

The images are initially divided into train and test folders, to perform the cross-validation technique they are moved from these folders to two new folders with all the images divided into the two classes, then the pre-processing phase can begin. At this stage, the images are converted from RGB format to grayscale; the images are resized to 64x64, to facilitate efficient training, and the pixel value is normalized in the range [0, 1].

2.2 Neural Network Architectures

To build an interesting analysis, four distinct neural networks architectures was chosen.

The first architecture is a Fully connected neural network with 4 hidden layer of 1024, 512, 256 and 128 neurons and it was chosen to evaluate the ability of this type of network in performing image classification tasks compared to convolutional networks(Figure 3).

The other three networks are convolutional neural networks, known to perform well in image classification tasks because they also take into account spatial information, composed of convolutional layers, which apply filters to all patches of the images generating in output a feature map, and pooling layers to reduce the dimensionality of feature maps. In this experiment, a convolutional layer with a 3x3 filter was used, followed by maxpooling layers, that takes only the max value inside the filter window, with a 2x2 filter and a classifier consisting of a 128-neuron dense layer before output. The first one has a feature extractor of 2 convolutional layers of 16 and 32 filters in output (Figure 4), chosen as the basis of comparison for performance, the second model, on the other hand, mantains the same depth with 2 layers but doubles the number of the filters (Figure 5), finally the third one is more deeper and it uses 4 convolutional layers of 16, 32, 64 and 128 filters (Figure 6).

Each layer has the ReLU activation function, which replaces all negative values with 0 avoiding the recurring problem of vanishing gradients, except for the output layer that uses the sigmoid function to perform binary classification, which is very often chosen in this type of problem because the interval is in [0, 1] and has an easy gradient to compute during back-propagation. It was also chosen a regularization technique, called dropout, often used in neural networks to prevent overfitting, which is implemented as a layer that randomly sets a fraction of input units to zero during training; in the experiments it was used before the output layers.

```
model = keras.Sequential([
    keras.layers.Flatten(),
    keras.layers.Dense(1024, activation='relu'),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dense(256, activation='relu'),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.1),
    keras.layers.Dense(1, activation='sigmoid')
])
```

Figure 3: First model architecture

```
model = keras.Sequential([
    keras.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(64, 64, 1)),
    keras.layers.MaxPool2D((2, 2)),
    keras.layers.Conv2D(32, (3, 3), activation='relu'),
    keras.layers.MaxPool2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.1),
    keras.layers.Dense(1, activation='sigmoid')
])
```

Figure 4: Second model architecture

```
model = keras.Sequential([
    keras.layers.Conv2D(64, (3, 3), activation='relu', input_shape=(64, 64, 1)),
    keras.layers.MaxPool2D((2, 2)),
    keras.layers.Conv2D(128, (3, 3), activation='relu'),
    keras.layers.MaxPool2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.1),
    keras.layers.Dense(1, activation='sigmoid')
])
```

Figure 5: Third model architecture

```
model = keras.Sequential([
    keras.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(64, 64, 1)),
    keras.layers.Conv2D(32, (3, 3), activation='relu'),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPool2D((2, 2)),
    keras.layers.Conv2D(128, (3, 3), activation='relu'),
    keras.layers.MaxPool2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dropout(0.1),
    keras.layers.Dense(1, activation='sigmoid')
])
```

Figure 6: Fourth model architecture

2.3 Training Hyperparameters

Different sets of training hyperparameters was used to highlight as much as possible the performance differences between them. All architectures use Adam optimizers that represent the state of the art, with an adaptive learning rate for each parameter and an exponentially decreasing average of the past gradient, and binary cross entropy as a loss function during training, which punishes more harshly predictions that are too far from the actual label. The technique of stopping early during training was chosen to avoid the waste of computational resources and the risk of overfitting by monitoring a performance metric, validation loss in this case, for a number of epochs called patience, in which no improvement in the metric occurs the iteration is stopped. The hyperparameters tested were learning rates with values 0.01, 0.001 and 0.0001, batch sizes with values 16, 64, 256 and a patience of 5. During preliminary testing, it was noted that a lower or higher value for learning rate did not lead to interesting results, while for batch size it was a timing problem for less than 16 and a resource problem for more than 256.

2.4 Cross-Validation

A 5-fold cross-validation strategy is used to calculate the risk estimates. The entire dataset is divided into five subsets of 1183 (or 1184) images and each architecture is trained and validated on five different combinations of these subsets, where each time the validation subset changes and the training set consists of the entire dataset minus the latter subset for a total of 4732 images. This approach provides a more robust assessment of the model's ability to generalize to the chosen dataset. For fold creation, it is important to take into account that the classes are unbalanced, so in the project StratifiedKFold was used to generate folds with the same percentage of classes as the entire dataset.

3 Results and Discussion

The results of the experiments are presented and discussed with a focus on the impact of the network architecture and hyperparameters choices on the final cross-validated risk estimate and they are visualized in a matrix with also the mean number of epochs.

3.1 Network Architecture and Hyperparameter Influence

It is evident how the feed forward architecture encountered the worst results, at the same batch size we can see how a lower learning rate lead to a better result, where a batch of 256 has the best result in 14.6 epochs. Having a lower learning rate lead to slower change on neuron weights that help in go through the minimum of the validation loss.

Regarding the three convolutional models we can make some considerations, the dataset is not particularly large and this is probably the reason why in any

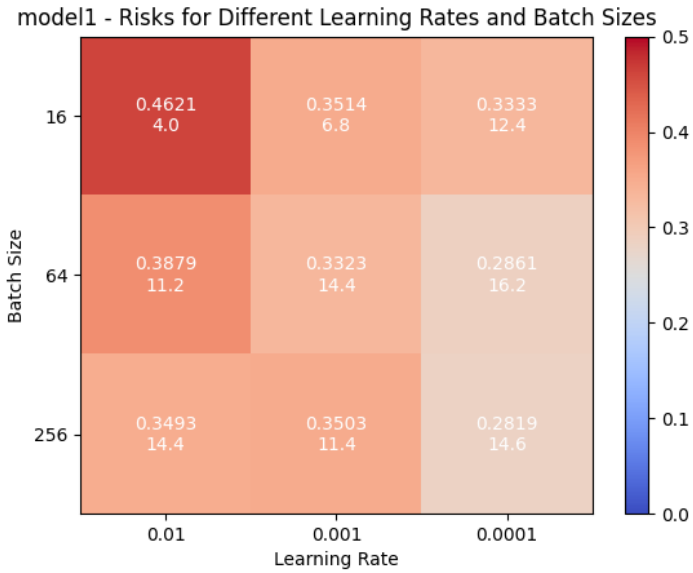


Figure 7: Risks matrix of first model

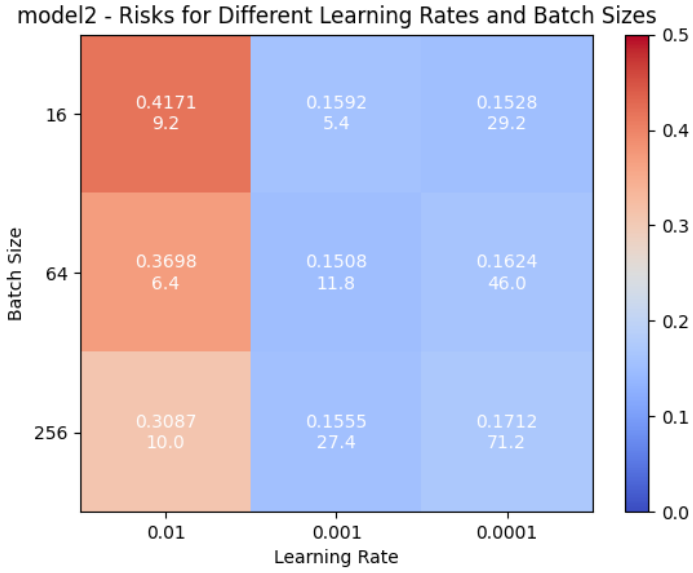


Figure 8: Risks matrix of second model

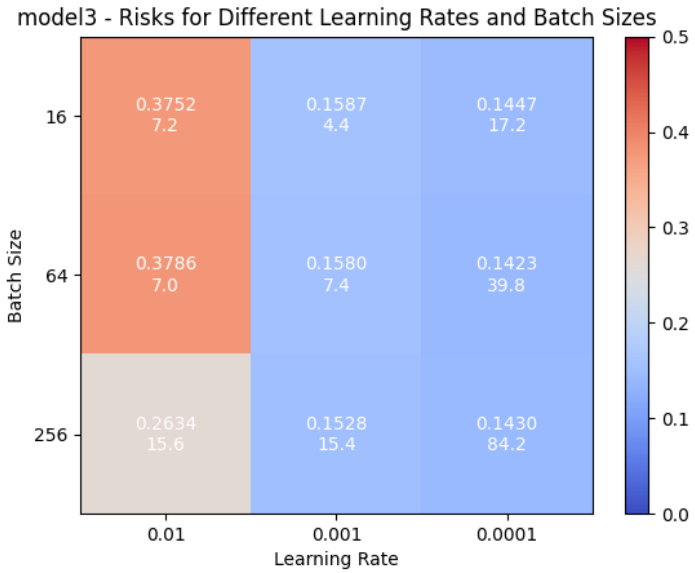


Figure 9: Risks matrix of third model

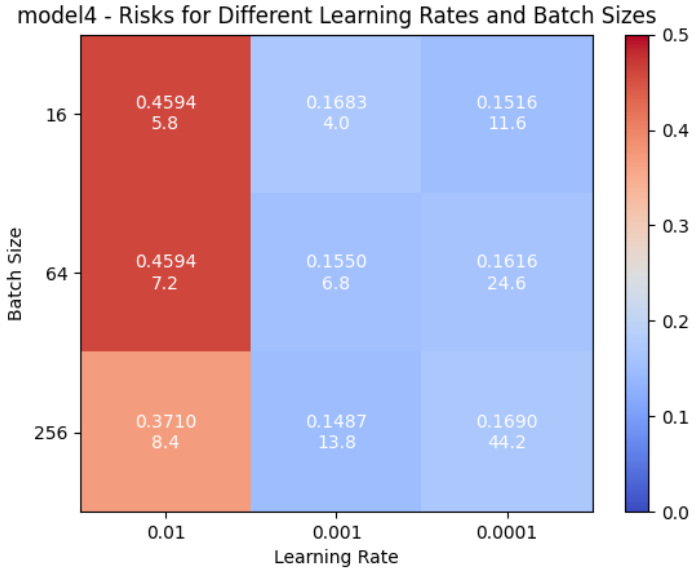


Figure 10: Risks matrix of fourth model

case the results do not differ too much, we can however see that for example the fourth model, i.e. the deepest, has very bad results with the highest learning rate, in some case worst than the feed forward architecture, presumably because having more parameters makes it more hard to reach convergence with a high learning rate. The third model is the architecture that lead to the best performance overall, where the best predictor obtain a 0.1423 for a batch size of 64 and a learning rate of 0.0001 after 39.8 epochs, and in general is the best with the lowest learning rate.

It's also intresting how the number of epochs decrease in the fourth model, realistically being the deepest model lead to an early overfitting.

More generally, it seems that in this problem the increase in depth did not lead to an increase in performance even in the best cases, while the third model is better, albeit slightly, on all combinations of hyperparameters, thus demonstrating how using larger filters while maintaining the same depth was the best choice for this problem.

The highest learning rate value turns out to be for all models and batch values the worst, with training lasting a few epochs. For the same batch size smaller values of learning rate lead to more epochs, hence more time before reaching an eventual overfitting.

3.2 Cross-Validated Risk Estimate

Zero-one loss is used to calculate the cross-validated risk estimate, i.e., the average zero-one loss estimated for each fold, and is useful for binary classification because it takes into account only the error on the final binary prediction, and not the distance between the prediction and the actual label, which can be 0 or 1.

The best configuration for the first model is with a batch size of 256 and a learning rate of 0.0001 after a mean of 14.6 epochs with a risk of 0.2819.

The best configuration for the second model is with a batch size of 64 and a learning rate of 0.001 after a mean of 11.8 epochs with a risk of 0.1508.

The best configuration in terms of risks in this test is 0.1423 for the third model with a batch size of 64 and 0.0001 of learning rate after a mean of 39.8 epochs.

The best configuration for the fourth model is with a batch size of 256 and a learning rate of 0.001 of learning rate after a mean of 13.8 epochs with risk of 0.1487.

It is clear how the worst configuration is with the lowest batch size and the highest learning rate for all the models.

3.3 Reproducibility

Below are all the parameters used to be able to reproduce the experiment, the models can be seen in Figures 1, 2, 3 and 4

- Batch size: 16, 64, 128
- Learning rate: 0.01, 0.001, 0.0001
- Activation function for hidden layer: ReLU
- Activation function for ouput layer: Sigmoid
- Optimizer: Adam
- Fold numbers: 5
- Loss function during training: Binary Cross-Entropy
- Loss function for risk estimate: Zero-one loss
- Max number of epochs: 100
- Dropout layer parameter value: 0.1
- Early stopping metrics: validation loss
- Early stopping patience: 5