UNIVERSITÀ DEGLI STUDI
DI MILANO

Convolutional Neural Networks for Binary Image Classification

Francesco Pineschi

18/03/2024

Contents

1 Introduction

In recent years, Convolutional Neural Networks (CNNs) have emerged as a powerful tool in the field of computer vision, revolutionizing tasks such as image classification, object detection, and segmentation. CNNs are a class of deep neural networks specifically designed to process visual data, mimicking the human visual system's ability to extract hierarchical features from images.

The significance of CNNs lies in their ability to automatically learn features from raw pixel data, eliminating the need for manual feature engineering. By leveraging convolutional layers, pooling layers, and fully connected layers, CNNs can effectively capture spatial hierarchies and patterns within images, enabling robust and accurate classification performance.

In this study, our objective is to delve into the implementation of a CNN for binary image classification, aiming to discern its effectiveness in distinguishing between two distinct classes: Chihuahua and Muffin. Without resorting to advanced modifications, we seek to explore the CNN's performance in a simplified yet practical scenario, shedding light on its applicability and efficacy in real-world image classification tasks.

2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) represent a class of deep learning models specifically tailored for processing visual data. They have demonstrated remarkable performance in various computer vision tasks, ranging from image classification to object detection and segmentation.

CNNs mimic the hierarchical organization of the human visual system, allowing them to automatically learn and extract features from raw pixel data. This capability eliminates the need for manual feature engineering, making CNNs highly versatile and applicable to diverse datasets.

An example of CNN application in the real world includes image classification tasks, where CNNs can accurately categorize images into predefined classes. For instance, in medical imaging, CNNs have been employed to diagnose diseases based on X-ray or MRI scans, showcasing their potential in critical healthcare applications.

Key components of CNNs include convolutional layers, pooling layers, and fully connected layers. Convolutional layers apply learnable filters to input data, capturing local patterns and features. Pooling layers reduce spatial dimensions, preserving important features while reducing computational complexity. Fully connected layers integrate extracted features for classification or regression tasks.

By leveraging these components, CNNs can effectively learn hierarchical representations of input data, enabling robust and accurate predictions across various domains.

2.1 Convolutional Layers

Convolutional layers are fundamental components of Convolutional Neural Networks (CNNs) designed for processing visual data. They play a crucial role in feature extraction by applying convolution operations to input images using filters.

Filters: Filters, also known as kernels, are small matrices applied to input images during convolution. Each filter extracts specific features from the input by performing element-wise multiplication and summation operations.

Kernel Size: The kernel size refers to the spatial dimensions of the filter. It determines the receptive field of the filter and influences the types of features extracted. Common kernel sizes include 3x3 and 5x5.

Strides: Strides determine the step size of the filter as it traverses the input image during convolution. A stride of 1 means the filter moves one pixel at a time, while larger strides result in downsampling of the output feature map.

Padding: Padding is the process of adding additional border pixels to the input image before convolution. It helps preserve spatial information and prevent loss of information at the image boundaries. Common padding types include 'valid', which applies no padding, and 'same', which pads the input to ensure the output feature map has the same spatial dimensions as the input.

Activation Function: The activation function introduces non-linearity into the convolutional layer's output. Common activation functions include ReLU (Rectified Linear Unit), which introduces sparsity and addresses the vanishing gradient problem.

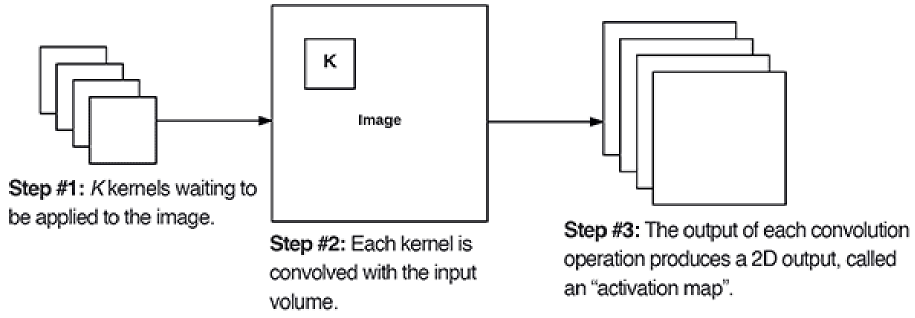

K
Image
Step #1: K kernels waiting to be applied to the image.
Step #2: Each kernel is convolved with the input volume.
Step #3: The output of each convolution operation produces a 2D output, called an "activation map".

Figure 1: Illustration of a Convolutional Neural Network (CNN) architecture.

2.2 Pooling Layers

Pooling layers are essential components in Convolutional Neural Networks (CNNs) used for down-sampling and feature reduction. They help reduce the spatial dimensions of the input feature maps, making the network more computationally efficient and reducing overfitting.

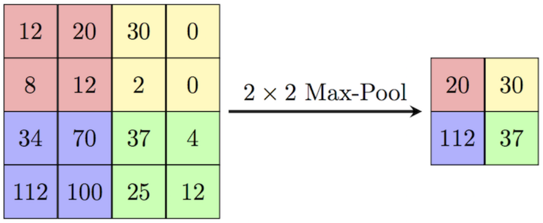
Pooling operations typically fall into three main categories:

Max Pooling: Max pooling selects the maximum value from each subregion of the input feature map. It retains the most salient features while discarding less important ones. Max pooling is the most common type of pooling operation used in CNNs due to its simplicity and effectiveness.

Average Pooling: Average pooling computes the average value from each subregion of the input feature map. It provides a smoothed representation of the features and is less prone to overfitting compared to max pooling.

Global Pooling: Global pooling computes a single value for each feature map by applying a pooling operation across the entire map. It reduces the spatial dimensions to a single value per feature map, often used as the input to the final classification layer of the network.

Among these categories, max pooling is exclusively chosen for this project due to its simplicity and capability.



12	20	30	0
8	12	2	0
34	70	37	4
112	100	25	12

20	30
112	37

Figure 2: Illustration of Max Pooling (2x2) performed on a matrix (4x4) of integers.

2.3 Fully Connected Layers

Fully connected layers, also known as dense layers, play a vital role in Convolutional Neural Networks (CNNs) by integrating the features extracted by convolutional and pooling layers for final classification or regression tasks. Dense layers connect every neuron in one layer to every neuron in the next layer, enabling global information propagation throughout the network.

In Keras, fully connected layers are implemented using the `Dense` layer. These layers require input data to be in the form of a one-dimensional vector. However, the output of convolutional and pooling layers is typically a three-dimensional tensor. Therefore, before passing the output to the dense layers, the tensor is flattened into a one-dimensional vector using the `Flatten` layer.

The `Flatten` layer serves the purpose of reshaping the output of the preceding convolutional and pooling layers into a format suitable for input to the dense layers. It collapses the spatial dimensions of the feature maps into a single vector while preserving the relationships between the features.

3 Dataset Modeling

In this chapter, we discuss how the dataset is partitioned, the utilization of augmentation techniques, and the establishment of stopping criteria.

3.1 Dataset Partitioning

The dataset is partitioned into three subsets: training, validation, and test sets.

- **Training Set**:
  - Used for model training.
  - Contains examples of input (features) and output (labels/targets).
  - During training, the model adjusts its parameters (weights and biases in neural network models) to learn the relationship between input and output.
  - Proportion: 80% of the original dataset.
- **Validation Set**:
  - Used for evaluating model performance during training.
  - Helps in adjusting model parameters and selecting the best hyperparameter configurations.
  - After each training iteration, the model is evaluated on this set to monitor overfitting and generalization.
  - Proportion: 12.5% of the training set.
- **Test Set**:
  - Used for final evaluation of model performance.
  - Contains data unseen during training or validation.
  - The model's performance on this set provides an estimate of its accuracy and predictive capability in real-world scenarios.
  - Proportion: 20% of the original dataset.

3.2 Augmentation

Data augmentation is a technique used to artificially increase the diversity of the training dataset by applying various transformations to the original images. This helps in improving the model's generalization ability and reducing overfitting. Common data augmentation methods include rotation, scaling, flipping, and other transformations.

It's important to note that augmentation is only applied to the training set and not the validation set to ensure unbiased evaluation of model performance. By training with augmented data, the models can better handle variations and complexities present in real-world data, leading to more reliable predictions.

In the code, an ImageDataGenerator object named `augmentation` is defined with several augmentation parameters:

- **rescale**: Normalizes the pixel values of the images.
- **rotation_range**: Specifies the range of rotation in degrees.
- **width_shift_range** and **height_shift_range**: Define the range of horizontal and vertical shifts, respectively.
- **shear_range**: Determines the range of shearing transformations.
- **zoom_range**: Sets the range of zooming transformations.
- **channel_shift_range**: Specifies the range of channel shifts.
- **horizontal_flip**: Enables horizontal flipping of the images.

Two data generators are then created for the training and validation sets using the `flow_from_dataframe` method of the `augmentation` object. These generators take dataframes containing file paths and corresponding labels as input and generate batches of augmented images during training.

3.3 Early Stopping Callback

EarlyStopping is a technique used to halt the training of a model if no improvement in performance on the validation set is observed for a specified number of epochs. This helps prevent overfitting and saves computational resources by stopping training when further optimization is unlikely to yield better results.

- **Monitor**: The "val_loss" parameter is monitored to determine when to stop training. The validation loss is a metric that measures the model's performance on the validation set.
- **Patience**: The "patience" parameter specifies the number of epochs to wait before stopping training if no improvement in the validation loss is observed. In this case, training will be stopped if the validation loss does not improve for 5 consecutive epochs.
- **Start From Epoch**: The "start_from_epoch" parameter specifies the epoch from which to start monitoring the validation loss for early stopping. This allows the model to train for a certain number of epochs before early stopping is applied. In this example, monitoring starts from epoch 15.

The EarlyStopping callback is added to the list `stop_early`, which will be passed to the `fit` method of the model during training. This ensures that early stopping is applied during the training process if the specified conditions are met.

4 Model Implementation

In this chapter, we detail the implementation of three distinct CNN models for binary image classification of Chihuahua and Muffin images. Each model is progressively more complex, with additional layers and dropout regularization applied to improve generalization performance.

4.1 Model 1: Simplest Model

The first model is constructed as follows:

- **Sequential Model Definition**: The model is defined as a sequential stack of layers, allowing for easy construction of CNN architectures.
- **Convolutional Layers**: Two convolutional layers are employed, each followed by max-pooling layers for downsampling.
- **Flattening and Dense Layers**: The feature maps are flattened into a one-dimensional vector, followed by a dense layer with ReLU activation.
- **Output Layer**: The final layer consists of a single neuron with sigmoid activation for binary classification.

```
model1 = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),

    Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),

    Flatten(),
    Dense(units=64, activation='relu'),

    Dense(units=1, activation='sigmoid')
])
```

4.2 Model 2: Enhanced Model with Dropout

In the second model, dropout regularization is introduced to prevent overfitting. Dropout randomly sets a fraction of input units to zero during training, preventing the network from relying too heavily on any individual feature. This encourages the network to learn more generalized patterns and reduces overfitting by promoting independence among neurons. It is constructed as follows:

- **Convolutional Layers**: Similar to Model 1, two convolutional layers with max-pooling are used.
- **Dropout Regularization**: Dropout layers with a dropout rate of 0.25 are inserted after each max-pooling layer and before the dense layer.

- **Dense Layers**: The architecture concludes with a dense layer with ReLU activation and an output layer with sigmoid activation.

```
model2 = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Flatten(),
    Dense(units=64, activation='relu'),
    Dropout(0.3),

    Dense(units=1, activation='sigmoid')
])
```

4.3 Model 3: Final Model with Additional Convolutional Layer

In the third model, an extra convolutional layer is added to further enhance feature extraction:

- **Convolutional Layers**: Three convolutional layers with increasing filter counts are utilized, each followed by max-pooling and dropout layers.
- **Flattening and Dense Layers**: The architecture is concluded with flattening, a dense layer with ReLU activation, dropout regularization, and an output layer with sigmoid activation.

```
model3 = Sequential([
    Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2),
    Dropout(0.25),

    Flatten(),
    Dense(units=64, activation='relu'),
    Dropout(0.3),

    Dense(units=1, activation=’sigmoid’)
])
```

4.4 Training and Augmentation

All three models are trained using a training dataset with and without augmentation techniques to enhance accuracy. During training, the data augmentation methods are applied to the images, effectively increasing the dataset’s diversity and aiding the model in learning more robust features. This approach significantly improves the model’s ability to generalize to unseen data, as observed from the analysis showing a notable enhancement in validation accuracy and a reduction in validation loss.

5 Results

The training results for the three models (Model 1, Model 2, and Model 3) are summarized below, including their performance with and without data augmentation techniques.

5.1 Model 1: Simplest Model

The accuracy and loss of Model 1 without data augmentation are suboptimal, with an accuracy of approximately 70% and a relatively high validation loss.

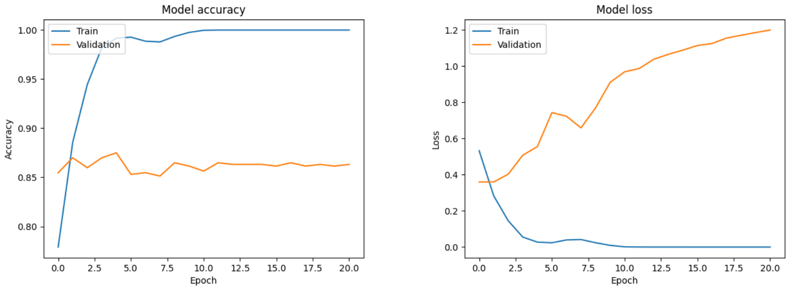


Figure 3: Accuracy and Loss of Model 1 (without augmentation)

5.2 Model 2: Enhanced Model with Dropout

Model 2 demonstrates improved performance compared to Model 1, achieving a higher validation accuracy and lower validation loss. The introduction of dropout regularization helps mitigate overfitting.

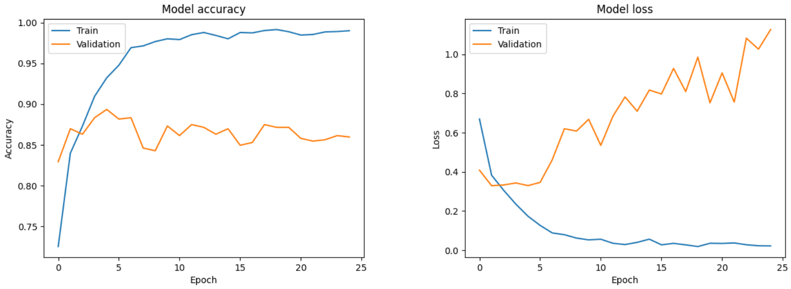


Figure 4: Accuracy and Loss of Model 2 (without augmentation)

5.3 Model 3: Additional Convolutional Layer

Model 3, the most complex architecture among the three models, exhibits slightly superior performance compared to Model 2. The additional convolutional layer enhances feature extraction, leading to further improvements in validation accuracy and loss.

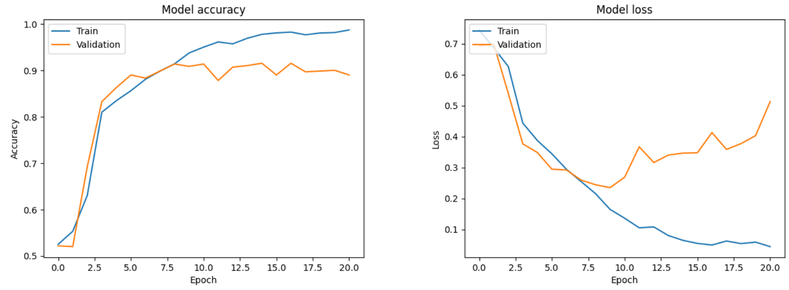


Figure 5: Accuracy and Loss of Model 3 (without augmentation)

5.4 Impact of Data Augmentation

When training the models using augmented data, the performance significantly improves across all models in terms of both accuracy and loss. Data augmentation increases the diversity of the training dataset, enabling the models to generalize better to unseen data.

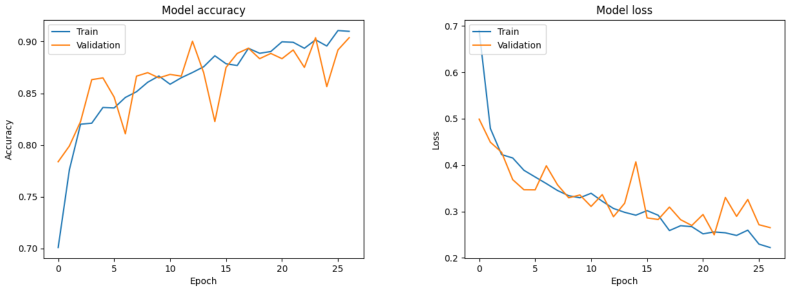


Figure 6: Impact of Data Augmentation on Model 1 Performance

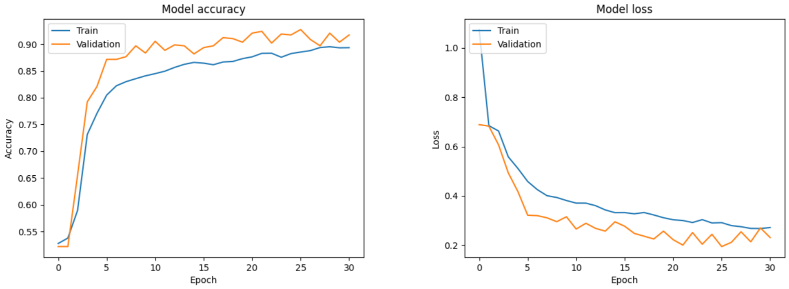


Figure 7: Impact of Data Augmentation on Model 2 Performance

5.5 Best Model: Model 3 with Augmentation

Among the three models trained with data augmentation, Model 3 emerges as the top performer. Its validation accuracy and loss demonstrate the most significant improvements, showcasing the effectiveness of the additional convolutional layer and dropout regularization in handling complex patterns within the dataset.

In summary, Model 3 trained with data augmentation exhibits the highest validation accuracy and the lowest validation loss, making it the optimal choice for binary image classification tasks involving Chihuahua and Muffin classes.

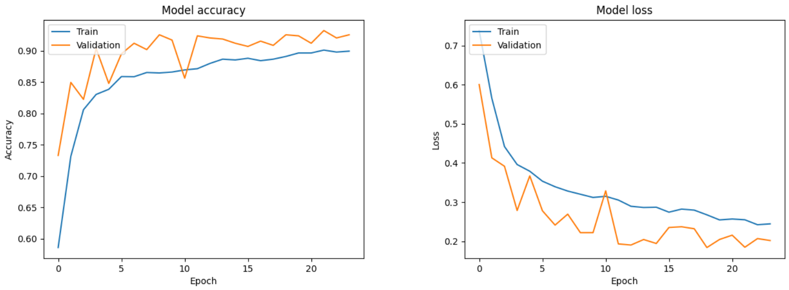


Figure 8: Impact of Data Augmentation on Model 3 Performance

6 Hyperparameter Tuning

Hyperparameter tuning is a critical step in optimizing the performance of a convolutional neural network (CNN) model. It involves the systematic search for the best combination of hyperparameters that maximize the model's performance on validation data. In this context, hyperparameters refer to parameters that are set before the learning process begins and are not updated during training.

To perform hyperparameter tuning, a Bayesian optimization approach is employed using the Keras Tuner library. The objective is to maximize the validation accuracy of the CNN model.

First, a model builder function is defined (`model_builder`) that constructs a convolutional neural network based on specified hyperparameters. The function takes an instance of `HyperParameters` (denoted as `hp`) as input, which allows for tuning of various model architecture parameters such as the number of filters, kernel size, dropout rates, and number of units in dense layers.

The model constructed by `model_builder` closely resembles the architecture of the third model (`model3`) described earlier, which consists of multiple convolutional layers followed by dropout regularization and dense layers.

Next, a Bayesian optimization tuner (`hp_tuner`) is initialized with the following parameters:

- **Objective**: The goal is to maximize the validation accuracy (”val_accuracy”) of the model.
- **Max Trials**: The maximum number of trials (model configurations) to evaluate during the search.
- **Directory and Project Name**: These parameters specify the location to save the results of the optimization process.
- **Overwrite**: If set to `False`, existing optimization results will not be overwritten.

The search space for hyperparameters is defined within the tuner, specifying the ranges and step sizes for each hyperparameter (e.g., filters, dropout rates, number of units).

The tuner then conducts the search (`hp_tuner.search`) using the training data generator (`train_gen_aug`) and validation data generator (`val_gen`). During the search, the model is trained for a fixed number of epochs (50 epochs in this case) while evaluating its performance on the validation set.

After the search completes, the best set of hyperparameters is retrieved (`best_hp`) based on the highest validation accuracy achieved. These hyperparameters are then used to build the final optimized model (`hypermodel`) by calling `hp_tuner.hypermodel.build`.

The resulting `hypermodel` is then trained using the `train_model` function with augmentation enabled.

The best hyperparameters found through the hyperparameter tuning process are as follows:

- **Convolution 1 Filters**: 64
- **Convolution 2 Filters**: 128
- **Convolution 3 Filters**: 256
- **Dropout Hidden Layer**: 0.2
- **Num Units (Dense Layer)**: 128
- **Dropout Flatten Layer**: 0.4

These hyperparameters represent the optimal configuration discovered through the Bayesian optimization search, aiming to maximize the performance of the CNN model for binary image classification.

7 Conclusion

Summary of key findings from implementing AlexNet for binary image classification. Reflection on the effectiveness of CNNs in classifying images of Chihuahua and Muffin classes.

8 References

List of cited works and resources used in the preparation of the paper.