Experimenting on Muffin vs Chihuahuas with various neural network architectures

A statistical methods for machine learning project

Jacopo Fichera

May 31, 2024

DECLARATION

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

1 Introduction

Machine learning solutions are taking an important place in everyday applications. Among the tasks we want to solve some may seem trivial. Being able to distinguish Muffins and Chihuahuas, which is more of a toy-like problem, is among those. The task itself is widely solved by many powerful and more general purpose tools nowadays but, as for the goal for this project, we want to see how close in terms of performance we can get to one of those better engineered solutions.

What we are facing **binary classification problems** category, and by the project's requirements it is requested to build a solution using Neural Networks.

The results discussed are based on best practices and empirical evidence gathered from the scientific community.

2 Machine and environment information

In order to replicate at best the results obtained for the project we give a brief on the machine used and its environments.

2.1 Machine Information

All the computations have been run locally. The system mounts Ubuntu 22.04.4 LTS (Desktop) while the Hardware components are:

- GPU: NVIDIA GeForce RTX3070Ti (6GB VRAM)
- CPU: AMD Ryzen 7 5800x 8-core processor x16
- RAM: 32 GB (2x16GB) DDR4

2.2 Development Environment

To make full use of the GPU power, which is very effective for machine learning, CUDA drivers were needed. The project ran on CUDA 11.8.

For designing the neural networks *Keras* was used. Since its latest major release (3.0) the backend, which handles the calculations, is selectable. *PyTorch* was selected as it is very popular among the research community.

Libraries and other references are listed in the Github repository[4].

3 The Muffin vs Chihuahua dataset

The Muffin vs Chihuahua set used for the project was published on Kaggle[3]. It is made out around six thousand classified images of Muffins and Chihuahuas scraped from google search. As the data is labeled we can make use of the **supervised learning** paradigm.

The images in the dataset are of various sizes but all in .jpg format.

3.1 Data distribution and normalization

The dataset, by default, is already partitioned in order to have a predetermined test set that is around 20% of the total data. Both in training and test sets around 45% of the images are muffins. This meas that we have a very small imbalance on the dataset which, by Google's[6] guidelines, is not concerning.

For the project it was decided to partition the data into:

- **Training set** (70%): Samples used to train the Neural Network
- **Validation set** (10%): A split used for two main reasons:
  - To measure learning without compromising the test set.
    We should not draw conclusions on the learning process on the test set as it is reserved for the final model evaluation.
  - To apply **Early Stopping**(4.5) as a regularization technique.
- **Test set** (20%): Used to evaluate the final obtained model.

The reason we use split the dataset into [70,10,20] parts is that it allows to not waste too much data on the validation set while still keeping the 5-fold CV partitioning as intended for the test split, covering the full dataset.

An important step when working with data, in order to make the learning process faster and more effective, is preprocessing. For the project images have been standardized ($\mu = 0, \sigma = 1$) as it improves the calculations of the gradients and grants a faster convergence.

While all images are going to be standardized, to avoid a common pitfall, it is important to calculate the mean(μ) and the standard deviation(σ) on the training set alone. If not done like this we would be having *data leakage* which might lead to overfitting as we also learn from the features of unseen data.

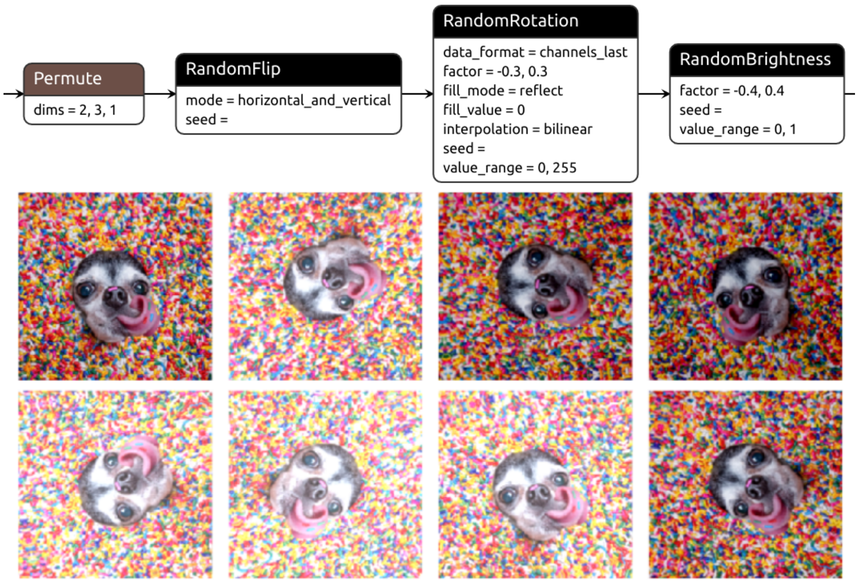
3.2 Data Augmentation

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error. Data augmentation is a process that falls under these techniques.[5, Section 5.2.2]

The idea to create more samples with small variations to extend the training set whilst making them valid is a common practice in computer vision tasks.

During the development of the project only the first few models were defined without the use of data augmentation. They were later enriched with this functionality as it provided a general increase in performance.

The augmentation in *Keras* can be defined as part of the model. The used procedure is the one shown in the graph of the image 1.



dims = 2, 3, 1

mode = horizontal_and_vertical
seed =

data_format = channels_last
factor = -0.3, 0.3
fill_mode = reflect
fill_value = 0
interpolation = bilinear
seed =
value_range = 0, 255

factor = -0.4, 0.4
seed =
value_range = 0, 1

Figure 1: The first image shows the augmentation structure used in the project. Below are shown various transformations done on the same (first of grid) image. The "permutation" layer present in the augmentation has only been added to work around some techical issues given by the torch backend and dataloading.

4 Hyperparameters Space

Many machine learning algorithms are not strictly algorithms as they have some parameters to set before the training process can start, called *Hyperparameters*. Those are considered families of learning algorithms of which Neural Networks are part of. They are defined as $\{A_\theta : \theta \in \Theta\}$ where θ is an instance of the possible hyperparameters space Θ.

Due to the *no-free lunch theorem* there is no general setting that works best for any situation. This means that the hyperparameters, while often good in enough in some "default" ranges, are specific to the learning problem. It's common practice to apply some heuristics on these parameters and to later tune them in order to build a better model.

Some parameters on which we should like to shine some light on are:

4.1 Image size and channels

The size of the input is an important element when designing a prediction model as we face two possible problems:

- Images of different sizes require special techniques to be elaborated
- Large input size can lead to a high number of parameters

Downscaling an image has also the advantage of acting as a regularization technique as the images we are providing show less detail and enhance recognizable features. This means that in most cases, if the image size is not reduced too much, the model tends to generalize better.

In the development of the project images were kept RGB and resized to (224×224) as it has been adopted by many solid models, like *VGG-16*(7.1.2).

4.2 Loss function

The loss function (l) is used to measures the difference between the predicted value $\hat{y}$ and the real value y. In statistical learning it is part of the specification of a learning problem (D, l) where D is the data distribution. As D is unknown the test set is created to measure the error of the predictor that is with a good probability, by the *Chernoff-Hoeffding bound*, close to the statistical risk.[]

For the project the **Binary Cross Entropy loss** was used:

$$l(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

It is the negative log-likelihood which measures the probability of observing the data assuming specific values for the parameters.

The loss function, compared to alternatives like the **hinge loss**, has the advantage of giving a probabilistic interpretation of the model outputs while also giving a harsher penalization to miss-classifications.

It was requested to run *k-fold CV* to calculate the model's risk estimates and report them according to the values of the **0-1 loss**: $l(y, \hat{y}) = 1(y \neq \hat{y})$.

4.3 Optimizer

It is an algorithm that optimizes a given objective function.

Most optimizers are gradients based with **Stochastic Gradient Descent**(SGD) being one of the first, and still relevant today. An step for SGD is defined as:

$$w := w - \eta \nabla J(w)$$

.

For the project SGD was the to go to as it still isn't outclassed by other methods[16] and also has few advantages compared to other optimizers such as having a lower memory footprint and fewer hyperparameters[8] to tune.

The default learning rate (η) is set to 0.01 but later fine-tuned.

4.4 Batch size

While the optimizer choice fell for SGD we actually run a variant of it being the **mini-batched** version. This version simply takes a small subset of the training data and averages the calculated gradients before updating the weights of the model. This usually leads to a faster convergence and also better results in general while having smaller memory requirements.

Another thing to consider is that, in practice, it was observed that lager batches tend to degrade the model's quality[7]

The initial batch size was set to 32 as it is a recommended starting value.

4.5 Epochs

An epoch is a training step computed on the full training set. Based on the machine learning algorithm a single iteration might be sufficient, as for k-NN, but that's not the case for NNs as they use gradient based optimization techniques.

The initial number of epochs has been set to 15 and later replaced by the flexible **Early Stopping** regularization technique.

The error on the validation set is used as a proxy to estimate the generalization error and, if there hasn't been any improvement for a set amount of epochs (*patience*=5) the training procedure is interrupted.

In *keras* this is obtained by increasing the maximum number of epochs and adding to the *fit* procedure the callback:

```
keras.callbacks.EarlyStopping(
        monitor='val_loss', patience=5, verbose=1, mode='min'
)
```

By default, once finished, the best network weights are restored.

4.6 Hyperparameter Tuning

While having fixed hyperparameters can lead to satisfying results it is desirable to have the best possible configuration of the learning process for the model. The procedure of learning hyperparameters can be obtained in various ways, such as nested **k-fold CV**. In the project we opted for the exposed *Bayesian Optimization Tuner* that is provided by *KerasTuner*. It works iteratively by placing priors on the unknown function and to update them in the form of posterior distribution over the target. The distribution is then used to construct an acquisition function that determines the next query point and the process is repeated. The higher the amount of observations the more confident the algorithm becomes of certain regions in space.[11]

The tuning process was applied in two ways, as described below.

Tuning the NN structure In the context of Neural Networks the hyperparameters skyrocket as the entire structure of the network is a set of hyperparameters itself. Therefore, we wanted to learn a possibly better structure than the ones defined by trial and error. This is shown in depth in the section 6.1.1.

Tuning the learning hyperparameters In order to improve the learning process for a specific model we learn the following list of hyperparameters on it:

- Learning rate: set between [1e-5, 1e-2] with a step of 2 on log sampling.
- Batch size: $\in \{8, 16, 32, 64\}$ to avoid too large values.
- Momentum (of SGD): set between [0.5, 1] with a linear step of 0.05

5 Naive MLP

The first network structure we tailored on is the classic **Multilayer Perceptron**(MLP), not to be confused with the first proposed Perceptron by McCulloch and Pitts[9] which does share some similarities with it.

These networks are feedforward, meaning they propagate information only to the following layers at evaluation time.

Each layer of the neural network of this structure is made of units, also called neurons, that connect to all the ones of the following layer, thus they are called **fully connected**. In each neuron the inputs are weighted by a matrix of coefficients, weights (w), summed and passed to an activation function(σ):

$$g(x) = \sigma(w^T x)$$

The activation function is usually non-linear. The computed value is what is passed as an input to all the neurons of the next layer. The parameters that the predictor has to learn during the training process are the values of the weight matrix. Of course the values of the matrix will be zero ($w_{i,j} = 0$) when $(i, j) \notin E$ where E are the edges of the acyclic graph that connect two neurons, as such connections do not exist.

To implement a simple MLP using Keras only two layers types are needed:

```
# This defines an input object
x = keras.layers.Input(shape=(32,))
# This defines a fully connected layer that takes the previous one in input.
x = keras.layers.Dense(128, activation="relu")(x)
```

Most neural network are trained using the **backpropagation** algorithm.

5.1 Proposed model

Two proposals were made for the creation of a MLP.

First Model A simple fully connected network that *Flattens* the input (3x224x224) and is followed by two hidden layers of 1024 and 256 units respectively. At top of the model there is a single unit layer with sigmoid activation acting as classifier.

Second Model Instead of having two hidden layers this model has only one of 128 units while being structurally identical to the first model.

5.2 Observed Results

Both models overfit. The second one, which is way smaller than the first one, still has too many parameters and leads to overfitting. Regularization techniques could be applied to increase the performance of the network but, considering CNN are both generally better and easier to implement when working with images, we shift our focus towards CNN's and leave DNNs behind.

6 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks introduce a powerful tool that fits very well when working in the context of image elaboration. This tool, which gives the name to the networks, is the **convolution layer**.

Convolution It applies the homonymous mathematical operation by combining two functions to produce a third one, in the discrete case:

$$(f * g)(n) = \sum f(m)g(n-m)$$

Where, in the context of images, f could represent an image and g a kernel.

More intuitively the kernel combines the pixels of an image in a window altered them values based on the neighborhood. In neural network this is not any different as the weighted inputs are convoluted by the kernel specification.

In image processing convolution is a basic step for many tasks, and it becomes powerful in neural networks as the model learns the kernel matrices during the training process. The power in applying convolutions to the data is that it alters its information to better show features of the image; i.e. from simple lines in the beginning of the network to complex shapes in deeper convolutions.

In keras this is obtained by the **Conv2D**[12] layer object, such as:

```
x = keras.layers.Conv2D(
    filters=64, kernel_size=(3,3), padding="same", activation="relu"
)
```

Pooling In order to reduce the amount of parameters and improve generalization one can use the **Pooling** layer. The pooling operation simply down-samples the feature map of the data. In *keras* this is achieved by any implementation of the pooling operation. As empirical performance between the various methods[1] doesn't favor any precise method, the **MaxPooling2D**[13] was used.

For a long time CNNs have been the state of the art for various fields involving machine learning, including Computer Vision. Now that transformers have become popular CNNs might have been outclassed, but the question is still open for debate.[15] Besides being very effective the fact that they are both easier to train and less memory constraining were other good reasons to favor CNNs instead of the simple MLPs approach.

6.1 Model Structure

First Model For the first model, shown in Figure 2, each convolution layer, two in total, is followed by pooling one. A good rule when designing CNN, confirmed by empirical evidence i.e. on *VGG-16* 7.1.2, is to increase the number of filters in deeper layers as the down-sampling reduces the power of representation of the network. This rule was applied to the definition of the model.

k	*accuracy*	l_{0-1}
0	0.8666	0.1334
1	0.9029	0.0971
2	0.8707	0.1293
3	0.8935	0.1065
4	0.8808	0.1192
avg	0.8829	0.1171

k	*accuracy*	l_{0-1}
0	0.8910	0.1090
1	0.8970	0.1030
2	0.8377	0.1623
3	0.8986	0.1014
4	0.8986	0.1014
avg	0.8846	0.1154

Table 1: Left the first model's *k-fold-CV* estimates, right the second one's.

In its first definition no image augmentation procedure was used. The model was overfitting the training set with $acc = 88\%$ and $l = 0.6591$.

We redefined the model using augmentation (3.2). This resulted in a similar model in terms of accuracy but a noticeable decrease in loss meaning the model is more certain about more classifications. The k-fold-CV estimates, after learning the optimal learning parameters, reported in 1 are coherent with the final measured model performance being: $acc = 89.36\%$ and $l_{0-1} = 0.1014$.

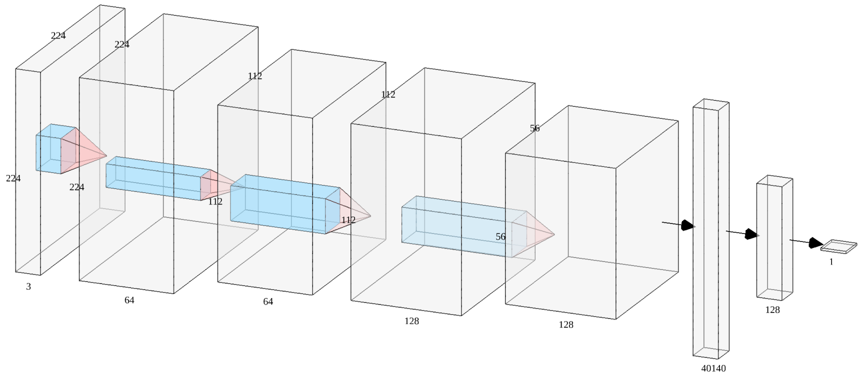


Figure 2: Graphical representation of the first CNN model.
Image generated via the NN-SVG tool

Second Model To reduce overfitting on the first model we tried defining a model with fewer parameters. The structure of the model is defined in Figure 3 and has, only on the first Convolution layer, a wider kernel of size (5×5) in the hope of the network to capture a broader context.

For this network the learning parameters have not been trained but the optimal configuration for the first conv network was used instead. The risk estimates for the network performance are reported in the Table 1.

The final model is in line with the expected performance as it measured $acc = 89.70\%$ and $l_{0-1} = 0.1030$ on the test set. If we tuned the training hyperparameters specifically for this architecture we might have gotten a better model.

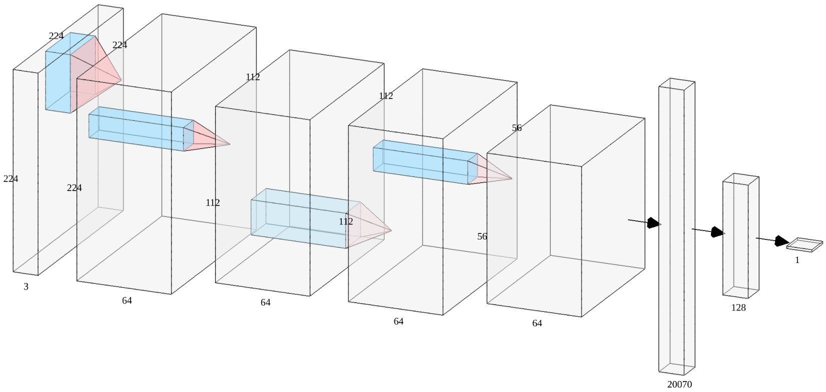


Figure 3: Graphical representation of the second CNN model.
Image generated via the NN-SVG tool

6.1.1 Autotuned CNN

The presented CNNs yield discrete results which might be for their simplicity so a better and more complex structure should be identifiable in some way. Instead of going on by process of trial and error we try to learn the architecture.

In order to do achieve this result while still testing a good amount of combinations, the search space was confined to the following variables:

convolutional_layers The amount of $(CONV \rightarrow POOL)$ layers to build.
For each of these, the pooling layer parameters were fixed.

filters_i Number of filters, a power of two from 16 to 256.
The setting is referred to the specific convolution layer i

kernel_i The square size of the kernel chosen from the set of values: $\{3, 5\}$.
The setting is referred to the specific convolution layer i

hidden_layers Number of dense layers that follow the convolutional ones

units_i Number of units for the dense layer i

The optimization process has run for a total of 40 iterations and resulted in various models with a close performance. From the results a trend indicates that networks with more convolutional layers generally lead to better results.

Auto-tuned Model The best performing model is of the characterized by 4 convolution layers and 2 dense ones. At top of the network a single dense neuron layer with sigmoid activation is put for final classification.
The structure of the network is shown in the Figure 4.

The learning parameters for the CNN were also fine-tuned which led to a final model with $acc = 92.31\%$ and $l_{0-1} = 0.0769$ which is just a little below average when compared with the *k-Fold CV* estimates reported in Table 2.

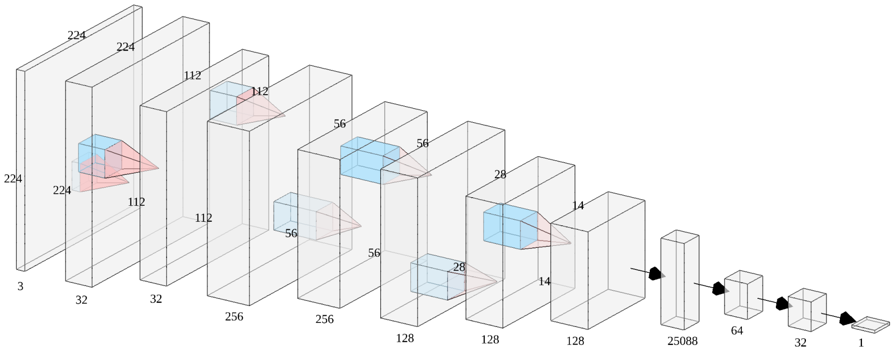


Figure 4: Graphical representation of the autotuned CNN model structure
Image generated via the NN-SVG tool

k	$accuracy$	l_{0-1}
0	0.9248	0.0752
1	0.9316	0.0684
2	0.9524	0.0676
3	0.9265	0.0735
4	0.9205	0.0795
avg	0.9272	0.0728

Table 2: Left the first model's *k-fold-CV* estimates, middle second, right the thrid one.

7 Case Studies

While the handcrafted NNs yielded relatively acceptable results we ask ourselves: how do other, well known, model perform with such a trivial task?
To answer this question two "case studies" were selected: Xception and VGG16.

7.1 Xception

The Xception[2] model was developed by Google and is an evolution of the Inception model presented during the ImageNet Recognition Challenge of 2012.

The Inception family of models are defined by a low count of parameters compared to competitors but, what makes *Xception* special, is that the use of depthwise separable convolutions throughout the network[14].

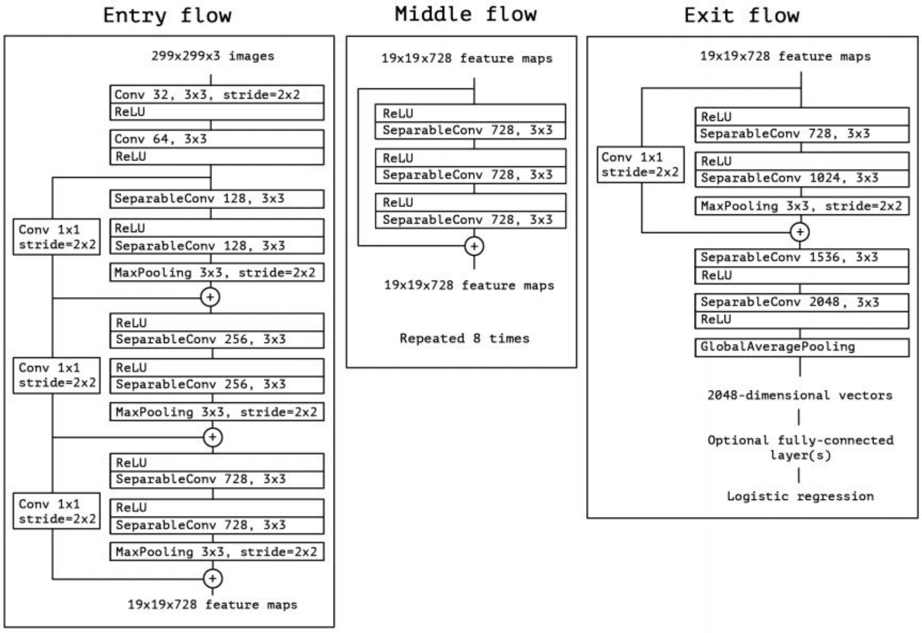
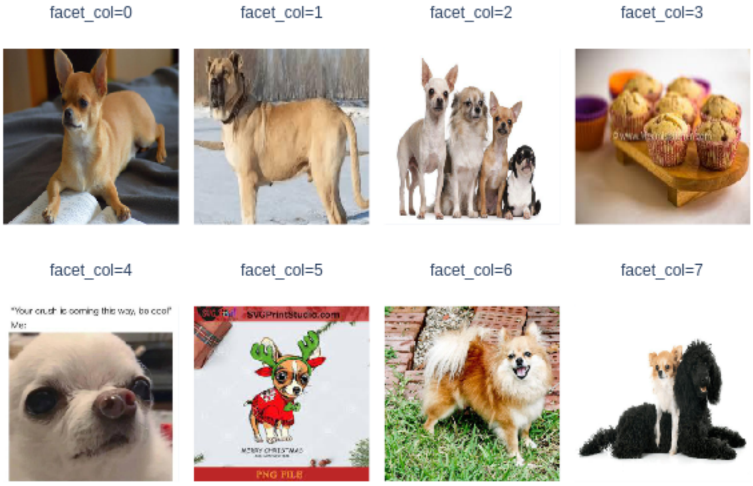


Figure 5: Xception is made of a total of 71 layers.
Image by the author of the original paper Francois Chollet[2]

7.1.1 Xception applied to Muffin vs Chihuahuas

To measure the performance of Xception on the Muffin vs Chihuahuas task we first see the pretrained model in action to later fine tune it on our dataset.

Pretrained Model The pretrained model used in the project for the evaluations was trained on the*Imagenet* dataset. The dataset does not provide a label "muffin" so it was mapped to "bakery"(415) and "bagel"(931) which for our intent is close enough. Predictions might not include either bakery or chihuahua as there are other classes, in this case we consider them as misclassifications.



facet_id	y	$\hat{y}$
0	chihuahua	['Chihuahua', 'toy_terrier', 'miniature_pinscher']
1	chihuahua	['bull_mastiff', 'bloodhound', 'Great_Dane']
2	chihuahua	['Chihuahua', 'toy_terrier', 'Mexican_hairless']
3	muffin	['French_loaf', 'tray', 'hamper']
4	chihuahua	['Chihuahua', 'Pomeranian', 'papillon']
5	chihuahua	['envelope', 'packet', 'handkerchief']
6	chihuahua	['Pomeranian', 'keeshond', 'Pekinese']
7	chihuahua	['standard_poodle', 'toy_poodle', 'miniature_poodle']

Figure 6: Some images and relative predictions of *Xception* on our dataset. Interestingly for facets 1 and 6 the prediction, while misclassified by our means, are actually correct as the images in the dataset are not really a chihuahuas.

When running the network on the test set the performance was low. It only correctly classified 67.65% of the samples. This was to be expected as the classifier is both untrained on muffins and has more labels that might actually fit the given image. We expect the model to work better once fine-tuned.

Fine-tuned Model In order to fine tune the model we have to define a new one that wraps around Xception. The model has to use the pre-processing on data used by Xception as requested by the keras documentation. It is also necessary to output a single prediction value instead of the previous 1000 classes.

During the training process, which often takes very little time, we freeze the network to avoid destroying information already built by the previous training process and only work up on the last layer of the *Xception*

The K-fold CV estimates results lead to a better model than the previously built ones as shown in the Table 7.1.1. The final trained model has has $acc = 99.16\%$ and $l_{0-1} = 00.84$ which is consistent with the expected values.

k	*accuracy*	l_{0-1}
0	0.9983	0.0017
1	0.9916	0.0084
2	0.9932	0.0068
3	0.9975	0.0025
4	0.9940	0.0059
avg	0.9949	0.0051

k	*accuracy*	l_{0-1}
0	0.9958	0.0118
1	0.9916	0.0084
2	0.9949	0.0051
3	0.9873	0.0127
4	0.9907	0.0093
avg	0.9905	0.0095

Table 3: Left the *Xception K-fold estimates*, right the *VGG-16* ones.

7.1.2 Vgg16

Developed by Visual Geometry Group (VGG) at the University of Oxford the VGG-16[10] is a deep CNN model that was proposed for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014 where it achieved top results in object detection and image classification.

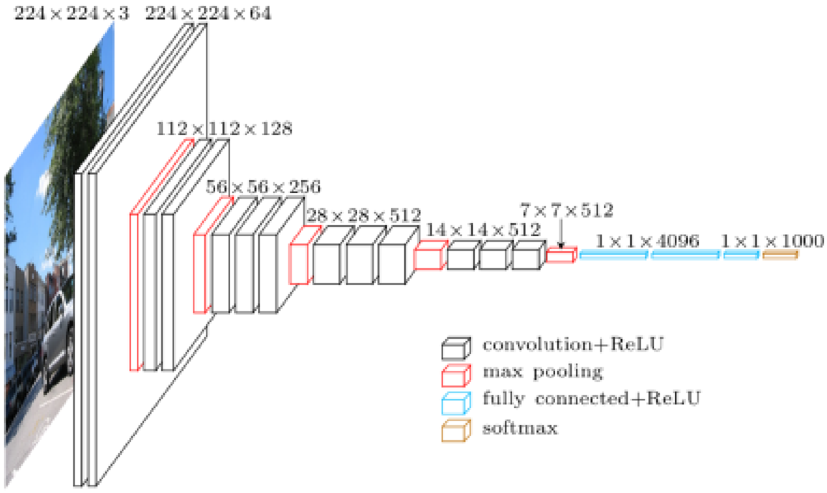


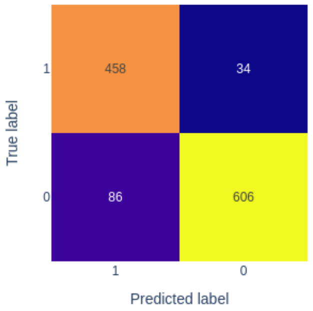
Figure 7: The VGG-16 model is characterized by 13 convolutionakl layers among which at times a max pooling layer has been placed. After those layers the network is closed by 3 fully connected layers with a final softmax classifier. *Image by Davi Frossari (https://www.cs.toronto.edu/ frossard/about/)*

Compared to Xception this model has more parameters

7.1.3 VGG-16 applied to Muffin vs Chihuahuas

The ideas we applied on Xception and the general methodology (7.1.1) are analogous to the ones applied to *VGG-16*.

Pretrained Model As for *Xception*, our used instance model of *VGG-16* was trained on Imagenet. When running the network on the test set we had poor



reference	*precision*	*recall*	*f1*	*support*
muffin	0.8419	0.9309	0.8842	544
chihuahua	0.9469	0.8757	0.9099	640
accuracy	-	-	0.8986	1184

Figure 8: Metrics of the first CNN developed model

performance as for the other analized model. It only correctly classified 57.01% of the samples, yielding a worse performance than *Xception*.

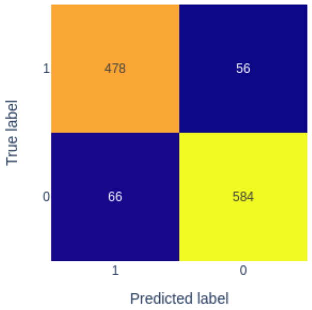
Fine-tuned Model The fine tuning process on the *VGG-16* uses its dedicated pre-processing function on the data. After a short training period we ran the K-fold CV estimates which are reported in Table 7.1.1 which is in line with the final model having $acc = 99.16\%$ and $l_{0-1} = 0.084$. Overall the performance of the classifier is on pair with the one of *Xception* with one drawback being: it has more parameters. So, from a practical standpoint, *Xception* could be preferable as the two models is not that far different in terms of results.

8 Conclusions

For the various models the final accuracy and 0-1 loss was given. Those metrics are good to express concisely the evaluation of a model.

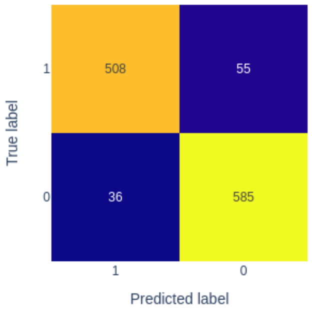
We can go deeper with our insight by measuring other metrics before drawing conclusions as reported in the Figures: 8, 9, 10, 11 and 12.

It's hard to understand why the CNN models are underfitting, are the models too simple to capture the complexity of the features? If that were the case a good way to increase the performance could be adding more layer or simply more parameters. Could the image augmentation technique be generating too noisy images? Adding other samples to the dataset is another path worth considering as the dataset is not huge but so far, as expected, the more complex and better trained pre-tuned models perform better under every aspect.



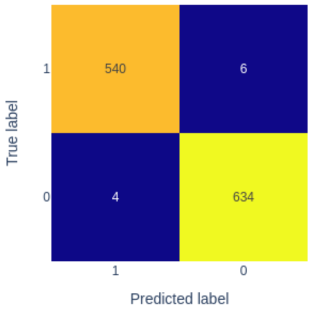
reference	*precision*	*recall*	*f*1	*support*
muffin	0.8951	0.8787	0.8868	544
chihuahua	0.8985	0.9125	0.9054	640
accuracy	-	-	0.8970	1184

Figure 9: Metrics of the second CNN developed model



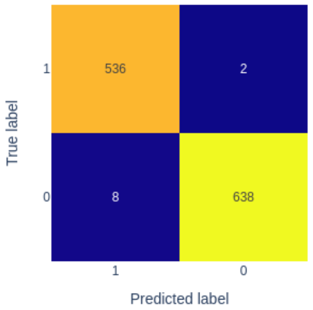
reference	*precision*	*recall*	*f*1	*support*
muffin	0.9023	0.9338	0.9178	544
chihuahua	0.9420	0.9141	0.9278	640
accuracy	-	-	0.9231	1184

Figure 10: Metrics of the autotuned CNN model



reference	*precision*	*recall*	*f*1	*support*
muffin	0.9890	0.9926	0.9908	544
chihuahua	0.9937	0.9906	0.9922	640
accuracy	-	-	0.9916	1184

Figure 11: Metrics of the Xception transfer learned model



reference	*precision*	*recall*	*f*1	*support*
muffin	0.9963	0.9853	0.9908	544
chihuahua	0.9876	0.9969	0.9922	640
accuracy	-	-	0.9916	1184

Figure 12: Metrics of the VGG-16 transfer learned model

References

[1] Florentin Bieder, Robin Sandkühler, and Philippe C. Cattin. Comparison of methods generalizing max- and average-pooling, 2021.

[2] François Chollet. Xception: Deep learning with depthwise separable convolutions, 2017.

[3] Samuel Cortinhas. Muffin vs chihuahua, 2023.

[4] Jacopo Fichera. Muffin vs Chihuahua SMML Project Repository, May 2024.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] Google. Imbalanced data, 2023.

[7] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima, 2017.

[8] Lightly. Which optimizer should i use for my ml project?

[9] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[10] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[11] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms, 2012.

[12] Keras Development Team. Keras documentation - conv2d layer.

[13] Keras Development Team. Keras documentation - maxpooling2d layer.

[14] Christian Versloot. Understanding separable convolutions.

[15] Zeyu Wang, Yutong Bai, Yuyin Zhou, and Cihang Xie. Can cnns be more robust than transformers?, 2023.

[16] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning, 2018.