Hyperparameter tuning methods in Transfer Learning with application to fine-grained classification

Marco Russo
Politecnico di Torino
MSc Computer Engineering

Abstract

This paper proposes a method for finding the best hyperparameters for training a neural network with transfer learning, separately for the warm-up and the fine-tuning stages.

The two-staged nature of this approach complicates the procedure, since performing two consecutive cross-validations would be a mistake due to the fact that the network has already been trained on the same samples, invalidating the estimate.

The pre-trained networks that I tested are EfficientNetB0 and DenseNet169, and the images I used for training were resized into a 130×130 *resolution, keeping the RGB channel.*

Starting with EfficientNetB0, I first performed a 3-fold cross-validation for finding the best hyperparameters for the warm-up stage, where the hyperparameters that I tuned are the learning rate, the optimizer, the regularization type (Dropout and L2 regularization) and the loss function (KL divergence and categorical cross-entropy).

Then, I performed the actual warm-up using only 90% *of the training set, after which I looked again for the best hyperparameters for the fine-tuning stage using a train-validation split where the train split consists in the previous* 90% *that the network was already warmed-up on, and the validation split consists in the remaining* 10% *of the training set. An alternative method is proposed too, but not performed. It is also showed why I didn't repeat the cross-validation on the same training set. The hyperparameters that I tuned are the number of layers that we keep frozen, the learning rate and the loss function.*

I then performed the full training with the best hyperparameters and I repeated all the previous steps with DenseNet169, comparing the final accuracies.

It is also shown what happens when we don't use augmented images for training.

Finally, it is shown how the performance varies when increasing the resolution of the training set up to 150×150*.*

The best model is the one with DenseNet169, augmentation, drop-out regularization and 250 frozen layers, with the hyperparameters discussed thoroughly in the paper, reaching a 67.3% *accuracy with* 130×130 *images, and a* 70.9% *accuracy with* 150×150 *images.*

1. Introduction

CNNs can be used for discerning many different but extremely similar classes (i.e. models of cars, flower types etc). This kind of application is called fine-grained classification and its complexity outdoes the one in standard classification, so we shall expect lower accuracy and higher training time.

Transfer learning helps reduce the training time by exploiting networks that were pre-trained on similar images, but, since it consists in two stages (warm-up and fine-tuning), the search for the best hyperparameters is non-trivial, since the best hyperparameters for the first stage may not be the best ones for the second one. Also, simply repeating the cross-validation at the second stage might make us incur in unexpected problems that we are going to discuss.

I found that these problems are often overlooked and I didn't find any paper facing these issues directly or proposing a thorough method, despite their importance and the fact that an incisive understanding of them can help us improve the overall accuracy.

For this reason, in this paper I propose a method for finding the best hyperparameters in a correct way, showing the problems that arise when some subtleties are not considered in the evaluation.

1.1. Dataset

The Stanford Dogs dataset contains images of 120 breeds of dogs from around the world. This dataset has been built using images and annotation from ImageNet for the task of fine-grained image categorization. There are 20,580 images, out of which 12,000 are used for training and 8,580 for testing. Class labels and bounding box annotations are provided for all the 12,000 images. The

dataset can be found at https://www.tensorflow.org/datasets/catalog/stanford_dogs.

As I wasn't interested in object detection for this task, I discarded the bounding box information. Also, since RAM memory in Colab is limited, I only used 8,000 out of the 12,000 training images, and 1,000 out of the 8,580 testing images. The images are preprocessed resizing them to 130x130 (keeping the RGB channel, since color can definitely help discern dog breeds) and the labels are one-hot encoded in order to be able to use the KL Divergence loss (which forced me however to use the standard categorical cross-entropy loss, not the sparse one).

Data augmentation was performed, not by using the ImageDataGenerator class that Keras provides, but by introducing some preprocessing layers directly into the model. This way the augmentation task is delegated to the GPU, so that we save CPU time and RAM. More about this will be discussed in the following section.

1.2. Methods

1.2.1 CNN architecture

The architecture of the network varies depending on the parameters that we pass to the function that builds it. Those parameters are related to the presence of the augmentation layers, the name of the pre-trained network, the type of regularization etc.

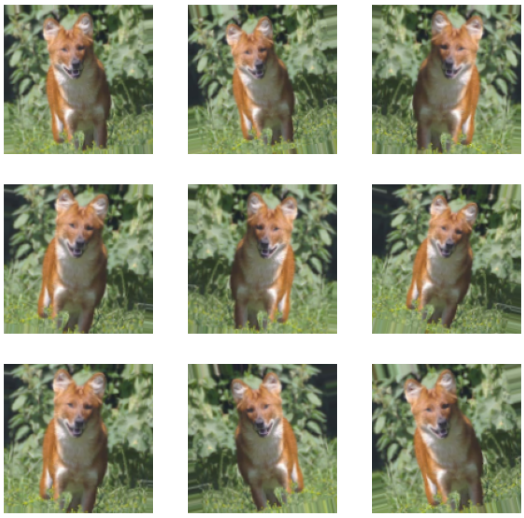
First of all I defined an input layer with shape $130 \times 130 \times 3$, since I reshaped the images into a 130×130 format and there are 3 channels (RGB).

Augmentation layers Instead of making the CPU augment the data batch-by-batch, which resulted in a waste of RAM space and CPU time, I delegated the task to the GPU by adding the augmentation layers directly into the model. These layers belong to tf.keras.layers.experimenta.preprocessing and they are a RandomFlip layer (horizontal only), a RandomZoom layer (from 0.1 out to 0.1 in) and a RandomRotation layer (I wanted a range from $-15\,\mathrm{deg}$ to $+15\,\mathrm{deg}$ as in the lab, but the layer constructor takes as input the percentage of 2π, and, since $15\,\mathrm{deg} = 0.26rad = 0.4 * 2\pi rad$, i put 0.4 as a range). All of the previous augmentation layers only work at training time; during inference, their output is just their input, unvaried.

These layers are added only if the augmentation flag is set to True.

Preprocessing layer After these layers, I added a Lambda layer (tf.keras.layers.Lambda) to preprocess the input depending on the base network that I'm using (if it's EfficientNetB0, we use the EfficientNet preprocessing function; same rationale for DenseNet169). Preprocessing functions usually zero-center/convert from RGB to BGR/rescale the images.

Figure 1. Image augmentation of a dog using the preprocessing layers



Pre-trained network I connected the pre-processing layer to the chosen base network (EfficientNetB0/DenseNet169), putting the latter in inference mode in order to keep the Batch Normalization layers in inference mode, and removing the top layers (the FC layers used for classification). A word about the two networks, both pre-trained on the ImageNet dataset:

- EfficientNetB0 is a network with the structure in Figure 2. As we can see, it uses some blocks called MBConv, which are Inverted Residual blocks. The difference with classical Residual blocks is that, while Residual blocks connect wide layers skipping the connection between the narrow layers in between (Figure 3), Inverse Residual blocks do the opposite: they connect narrow layers skipping the connection between the wide layers in between (Figure 4). On of the advantages of this approach is that less memory is required, since we connect layers with fewer parameters. The total number of layers is 236. Also notice that the network was pre-trained on 224×224 images. Other variants of EfficientNet, from B1 to B7, are pre-trained on higher resolution images.

- DenseNet169 is a network with the structure in Figure 5. In this architecture, each layer is forwardly connected to every other layer. If we have L layers, we then have $\frac{L(L+1)}{2}$ connections. This way, we alleviate the vanishing gradient problem and we improve feature reuse. The difference with the other DenseNet variants, i.e. DN-121, DN-201 and DN-264 lies in the

Figure 2. EfficientNetB0 network architecture

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Figure 3. An example of Residual blocks

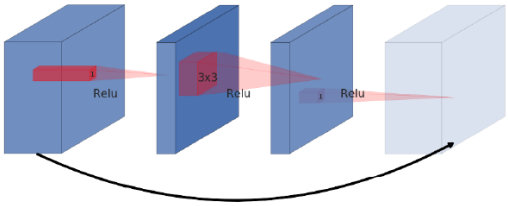
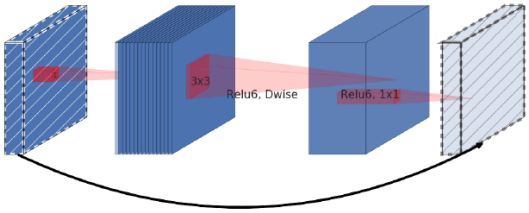
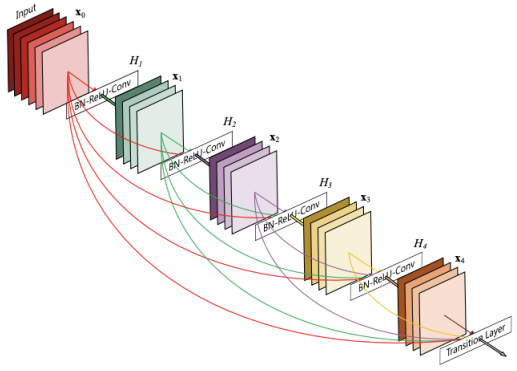


Figure 4. An example of Inverse Residual (MBConv) blocks



number of convolutional layers in each of the convolutional blocks, as we can see in Figure 6.

Figure 5. DenseNet169 network architecture



This network was pre-trained on 224×224 images too. The total number of layers is 594, so we shall expect a higher accuracy overall.

Average pooling layer An average pooling layer (tf.keras.layers.GlobalAveragePooling2D) is used to reduce the dimensionality of the input, focusing more on overall features, as opposed to max pooling.

Figure 6. DenseNet169 in detail

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112×112	7×7 conv, stride 2			
Pooling	56×56	3×3 max pool, stride 2			
Dense Block (1)	56×56	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 6$
Transition Layer (1)	56×56	1×1 conv			
	28×28	2×2 average pool, stride 2			
Dense Block (2)	28×28	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 12$
Transition Layer (2)	28×28	1×1 conv			
	14×14	2×2 average pool, stride 2			
Dense Block (3)	14×14	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 24$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 64$
Transition Layer (3)	14×14	1×1 conv			
	7×7	2×2 average pool, stride 2			
Dense Block (4)	7×7	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 16$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 32$	$\begin{bmatrix} 1 \times 1 \text{ conv} \\ 3 \times 3 \text{ conv} \end{bmatrix} \times 48$
Classification Layer	1×1	7×7 global average pool			
		1000D fully-connected, softmax			

Batch Normalization layer To help the optimizer stabilize the learning process, I added a Batch Normalization layer (tf.keras.layers.BatchNormalization), witch normalizes the input of the layer using the mean and variance of the current batch.

Dropout layer If Dropout regularization is chosen, a Dropout layer is added (tf.keras.layers.Dropout), with the specified dropout probability. Dropout is a form of regularization that, during training, randomly disables the input units with probability p (ranging from 0 to 1). During inference, the Dropout layer is of course skipped. It's a stronger form of regularization as it gives the network a sort of "robustness".

If L2 regularization is chosen, this layer is not added.

Classification layer and L2 regularization The last layer is a Dense layer (tf.keras.layers.Dense) with 120 (the number of breeds) units and softmax activation. If L2 regularization is chosen, the constructor of the Dense layer is called specifying an L2 regularization with the chosen λ.

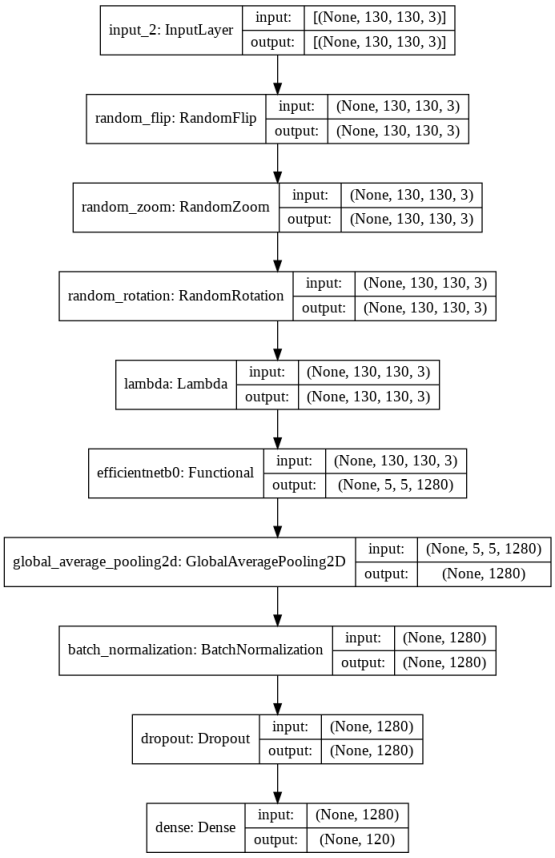
L2 regularization tries to minimize the L2-norm of the weights-vector of the Dense layer, giving it an importance proportional to the λ parameter. That is, we add a term $\frac{\lambda}{2m}\|w\|^2$ to the loss objective function that we want to minimize. This way, weights are never too high and we avoid overfitting.

As an example, in Figure 7 there is a network graph for the model with EfficientNetB0 and Dropout regularization. The DenseNet variant simply substitutes the Functional layer with another Functional layer. The L2 variant removes the Dropout layer (the regularization is put into the Dense layer).

1.2.2 Loss functions

Besides cross-entropy, we want to see whether Kullback-Leibler divergence is a good loss function for our purpose. However, this loss forces us to work with one-hot encoded labels, so, when testing with categorical cross-entropy loss, we have to use the "standard" one, not the sparse one.

Figure 7. EfficientNetB0, Dropout



Let's first consider that, in one-hot encoding, and since we have $N_c = 120$ classes (the dog breeds), a label with value i is converted into a 120-long array where all the entries are 0 except for the i-th, set to 1.

Now, the output layer of the model is a Dense layer with 120 units. The i-th unit gives an output ranging from 0 to 1, representing the probability of the i-th class, and the sum of all of them equals 1. Each unit gives always a non-zero output. Let's assume then that the ground truth of a sample is class 1 (that is, $y = [1, 0, \dots, 0]$). Now, let y_{pred} be the network output for that sample.

- Kullback-Leibler divergence: it is calculated as

$$L_{KL}(y, y_{pred}) = \sum_{1}^{N_c} y(i) \log \frac{y(i)}{y_{pred}(i)}$$

- Categorical cross-entropy: it is calculated as

$$L_{CE}(y, y_{pred}) = -\sum_{1}^{N_c} y(i) \log y_{pred}(i)$$

1.2.3 Optimizer

When compiling a model, besides the loss function we have to specify the optimizer and the learning rate. Both are hyperparameters that we want to tune, and we choose the optimizer between SGD and Adam.

1.2.4 Hyperparameter tuning

Since transfer learning consists in two stages (warm-up and fine-tuning) we can't just do only one tuning, because there are two consecutive trainings of different kind, so the hyperparameters that were the best for the warm-up stage may not be the best for fine-tuning (a trivial example is the learning rate).

The following steps are done with EfficientNetB0 first, and then with DenseNet169, so that we can compare the performances.

Hyperparameter tuning for warm-up We use 3-fold cross-validation to compare the accuracy that we reach varying the following hyperparameters:

- the optimizer (SGD/Adam)
- the loss function (KL divergence/categorical cross-entropy)
- the type of regularization (L2 with $\lambda = 0.05, 0.1$, and dropout with $p = 0.7$)
- the learning rate $(0.005, 0.01)$

However, using the best validation accuracy as the only metric is not the good idea, since there may be the risk of overfitting. If, say, model A has a slightly lower validation accuracy than model B, but the training accuracy of B is much higher than its validation accuracy, while the training accuracy of A is approximately equal to its validation accuracy, then A is preferrable. So we do two cross-validations, one with L2 regularization and the other one with Dropout, and we compare not only the best validation accuracies, but also the related validation accuracy, choosing eventually between Dropout and L2.

Hyperparameter tuning for fine-tuning At this point, we have found the best regularization method between Dropout and L2, and the related best learning rate, loss function and optimizer.

We can now do the warm-up with this configuration for a maximum of 20 epochs and using Early Stopping, monitoring the validation loss with a patience of 4 epochs (that is, if it doesn't increase in 4 epochs we stop the training, so that we furtherly avoid overfitting). We then save the resulting weights in a file. However, we can't train on the whole

training set and then repeat the cross-validation on the same set, since it's conceptually wrong.

Indeed, since the network has already been trained on those samples, when we repeat the cross-validation on them for fine-tuning we could incur in problems like overestimated validation accuracy (since the validation split contains samples that the network was warmed-up on) or overfitting (which results in underestimated validation accuracy). This was confirmed by an experiment I performed, discussed in the next subsection.

We have two main ways of proceeding:

- if we have a lot of RAM space we can do the warm-up on our 8000 training images, then download 8000 other images and repeat the cross-validation on those (never seen before by the network). This is more precise but couldn't be done on Colab for space limitations;
- we do the warm-up on the first 7040 images (the choice of the number is motivated by the fact that we take around 90% of the training set and 7040 is a multiple of 64, the batch size). Then, we tune the fine-tuning hyperparameters on a train-validation split, using the first 7040 images as a train split and the last 960 images as a validation split (the network has never been trained on these 960 images). We then compare the scores for each hyperparameter combination and we choose the combination that gives us the highest score. This gives us a less precise estimate, but it's a good compromise when we don't have a lot of space or the dataset is relatively small.

Out of the two methods that I proposed above, I used the second one. So, we perform the warm-up on the first 7040 images, then find the best hyperparameters for fine-tuning using a training-validation split where the training split consists in those 7040 images and the validation split consists in the remaining 960 images (they both come from the original training set). Instead of tuning again the optimizer and the regularization type, we keep using the best ones found for warm-up, with the difference that we increase the regularization strength to avoid overfitting (if we use Dropout we increase p, if we use L2 we increase lambda), since this stage is more delicate and we don't want to destroy what the network has learned.

So, we vary the following hyperparameters:

- K, the number of layers in the pre-trained network to keep frozen (for EfficientNetB0, we test $K = 50, 150$; for DenseNet169, we test $K = 250, 350$)
- the loss function (KL divergence/categorical cross-entropy)
- the learning rate $(0.0005, 0.0001)$

The best combination is the one that gives us the best validation accuracy.

Final complete training At this point we have found the best hyperparameters for warm-up and for fine-tuning respectively. So, we can now perform the two stages on the whole training set, using those hyperparameters. Early stopping with the same criteria is used in both stages. Validation accuracy is calculated on the testing set.

Performance without data augmentation Since enabling the augmentation layers or not may be considered as an hyperparameter itself, we repeat the warm-up and fine-tuning with the best hyperparameters (for both stages) found above, with the only difference that we disable the augmentation layers. The only purpose of this step is to show the presence of overfitting.

Choosing between the two pre-trained networks The steps above are performed with EfficientNetB0 at first, and then with DenseNet169. The one which gives us the best final accuracy is the network that we prefer.

Performance variation when training on bigger images It's expectable that by increasing the size of the images used for training we should get a higher accuracy, so, just for the sake of testing, we take the best hyperparameters for the best pre-trained network found in the previous steps and perform the full training on 150×150 images, not anymore on 130×130. We then evaluate the resulting accuracy.

1.3. Experiments

A batch size equal to 64 was used for all the experiments.

1.3.1 Using EfficientNetB0

Hyperparameter tuning and full training With the best hyperparameters for the L2 case, the mean validation accuracy is 0.6073, but the mean training accuracy is around 0.73, which is a symptom of overfitting.

Instead, with the best hyperparameters for the Dropout case, I got a mean validation accuracy equal to 0.6074 and a mean training accuracy around 0.60, so no overfitting. Both because the mean accuracy is higher for the Dropout case and because there's no overfitting I preferred Dropout regularization.

The best warm-up hyperparameters are:

- LR = 0.005
- Optimizer = SGD
- Dropout with $p = 0.7$

- Loss = KL divergence

Then, performing the warm-up on the first 7040 images and using a training-validation split as explained above, I found the best fine-tuning hyperparameters (keeping the SGD optimizer and rising p up to 0.8):

- $K = 100$
- $LR = 0.0005$
- Loss = KL divergence

With these two combinations of hyperparameters I performed a full training on the whole training set. With warm-up we reach 61.4% accuracy. After fine-tuning, I reached 65.1% accuracy.

In figure 8 and 9 there are the learning curves after warm-up and after fine-tuning.

Figure 8. After warm-up, using EfficientNetB0

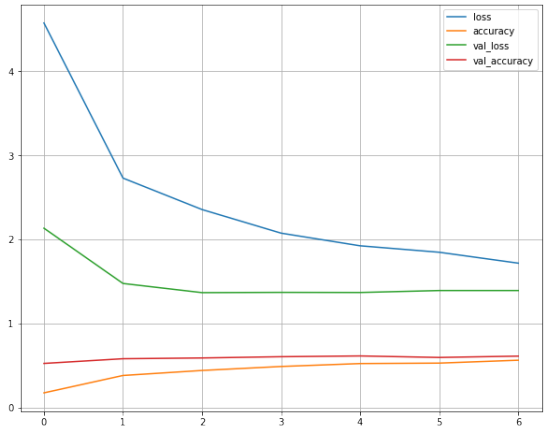
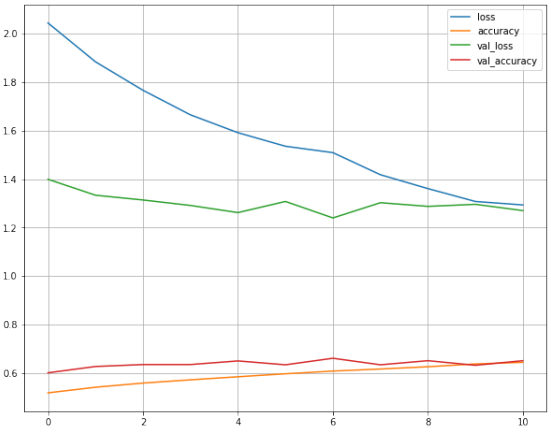


Figure 9. After fine-tuning, using EfficientNetB0

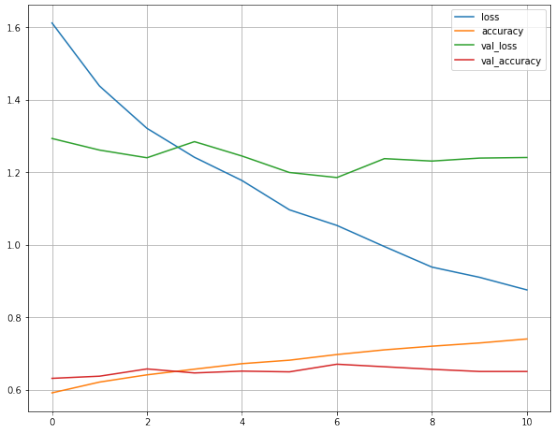


Predicting the breed from a downloaded image I tried to test the model on a downloaded image of a Leonberger dog (Figure 10), knowing that this breed corresponds to the label 103. So I resized the image into a 130×130 image and then I gave it to the model, which predicted correctly that the class it belongs to is 103.

Figure 10. A picture of a Leonberger dog downloaded from the web (https://worlddogfinder.com/it/razze/leonberger)

Performance variation without data augmentation Repeating the training without the augmentation layers, I reached approximately the same accuracy (65.1%), but strong overfitting is present (in figure 11 we can see the learning curves after fine tuning) making the model undesirable and less generalizable (even though not so much, since a strong regularization effect is already given by the dropout, and a weak one by the Batch Normalization).

Figure 11. (No augmentation) After fine-tuning, using EfficientNetB0



1.3.2 Using DenseNet169

Hyperparameter tuning and full training With the best hyperparameters for the L2 case, the mean validation accuracy is 0.6576, but the mean training accuracy is around 0.76, which is a symptom of overfitting.

Instead, with the best hyperparameters for the Dropout case, I got a mean validation accuracy equal to 0.6521 and a

mean training accuracy around 0.63, so no overfitting. For this reason, even though the validation accuracy is slightly lower, I preferred Dropout regularization to avoid overfitting.

The best warm-up hyperparameters are:

- LR = 0.005
- Optimizer = SGD
- Dropout with $p = 0.7$
- Loss = Categorical cross-entropy

Then, performing the warm-up on the first 7040 images and using a training-validation split as explained above, I found the best fine-tuning hyperparameters (keeping the SGD optimizer and rising p up to 0.8):

- $K = 250$
- $LR = 0.0001$
- Loss = Categorical cross-entropy

With warm-up I reached 65.6% accuracy (vs 61.4% with EfficientNetB0). After fine-tuning, I reached 67.3% accuracy (vs 65.1% with EfficientNetB0).

Performance variation without data augmentation I repeated the same experiment as above, getting the same results: similar accuracy (68.2%) but strong overfitting.

1.3.3 Using bigger images

With the hyperparameters found at the previous stages, and using DenseNet169, I performed the full training using images at a resolution of 150×150. As expected, I got an increase in accuracy: after warm-up I reached 70.3%, and after fine-tuning the accuracy was 70.9%.

1.3.4 Experiment: cross-validating twice on same samples

To confirm my thesis, I tried cross-validating on the whole training set for the fine-tuning stage after having performed the warm-up on the same set. As expected, I got a way higher estimated validation accuracy (79.4%), which is definitely a wrong estimate.

1.4. Conclusion

The experiments show that the model with DenseNet169 as a pre-trained network with the set of hyperparameters above outperforms the one with EfficientNetB0, which was expectable since DenseNet169 is deeper.

They also confirmed that without augmentation the CNN tends to overfit, and that by using bigger images for training we reach a higher accuracy.

This project made me face some problems that might arise when dealing with limited resources, such as the fact that using an Image Generator for data augmentation quickly fills up the RAM and slows down the performance, which is what brought me to delegate the augmentation task to the GPU by putting directly some preprocessing layers inside the model, so that the CPU is freed from this task and the RAM isn't involved.

Also, I had the intuition to consider not only the cross-validation accuracy, but also the training accuracy, by observing that sometimes the best validation accuracy was reached at the cost of high overfitting, so I learned to always contextualize and incisively examine the results that come from cross-validation.

When fine-tuning, I noticed the tendency to overfit, which made me rise the regularization strength (p for Dropout, λ for L2 regularization).

Observing the best fine-tuning hyperparameters, I noticed that keeping fewer layers frozen was better, which is a sign that ImageNet is a bit more different than Stanford Dogs than expected, which resulted in a need for recalculating the weights of more layers in the pre-trained network.

Finally, since I first tried to find the best hyperparameters for fine-tuning repeating the cross-validation (the wrong method, as I discussed above), I noticed that the estimated accuracy was very high, but when I performed the actual training (using the test set to get the validation accuracy) I noticed that the resulting validation accuracy was way lower than the estimated one, which made me realize that the method was wrong and forced me into thinking about another one.

Given the scarcity of resources and time, the tested hyperparameters are not many, but this method can be used for testing a lot more (both regarding their values and their class). The number of CV folds can be increased too.

Finally, having more RAM space, it is a good practice to use the first method that I proposed, i.e. downloading 8000 other images that the network was never warmed-up on and using them for a second cross-validation, getting a better estimate.

2. References

[1] Data augmentation (https://www.tensorflow.org/tutorials/images/data_augmentation)

[2] DenseNet (https://pytorch.org/hub/pytorch_vision_densenet/)

[3] EfficientNet B0 to B7 (https://keras.io/api/applications/efficientnet/)