Machine learning and Deep learning

Homework 2

Introduction

The aim of "Machine learning and Deep learning" second homework is training a *Convolutional Neural Network* (*ConvNet* or *CNN*) for image classification starting from a provided template code.

A *CNN* is a Deep Learning algorithm which can take in an input image, assign importance to various aspects in the image and be able to differentiate one picture from the other.

*ConvNet*s belong to the category of *Neural Networks* and so are computing systems inspired by the biological neural networks that constitute animal brains.

CNNs use relatively little pre-processing compared to other image classification algorithm, so this means that the network learns the filters that in traditional algorithms are hand-engineered. This independence from prior knowledge and human effort in feature design is a major advantage.

The word convolutional stems from the latin *convolvere* meaning "to roll together." In mathematics, a convolution is a specialized kind of linear operation. In decoding imaginery, *CNNs* use the priciple of convolution in place of general matrix multiplication in at least one of their layers.

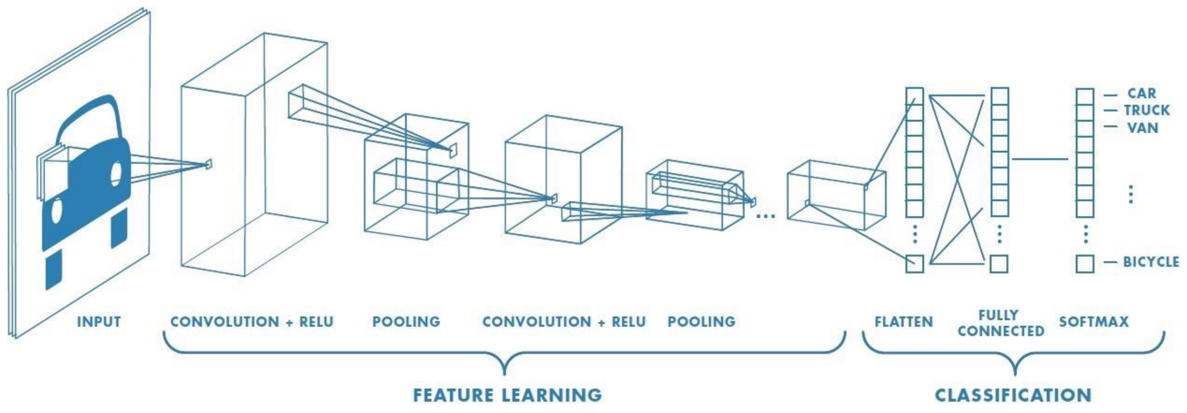


Figure 1 - Classic CNN architecture

AlexNet

The *Convolutional Neural Network* used to perform the assignment is called *AlexNet*.

AlexNet was designed by Alex Krizhevsky and published with Ilya Sutskever and Geoffrey Hinton.

It famously won the 2012 ImageNet Large Scale Visual Recognition Challenge competition achieving a top-5 error of 15.3%, more than 10.8 percentage points lower than that of the runner up.

The network, which has 60 million parameters and 650,000 neurons, contains eight layers: five of these are convolutional layers, some of which (the first, the second and last) are followed by max-pooling layers (used to down-sample the width and height of the tensors, keeping the depth same), and three are fully-connected layers with a final 1000-way softmax, which produces a distribution over the 1000 class labels in order to quantify how sure is the prediction. For our purpose, this last layer will be replaced by an equal one having only 101 output neurons, due to a smaller number of categories to recognize.

To make training phase faster, *AlexNet* uses non-saturating neurons and a very efficient GPU implementation of the convolution operation, while to reduce overfitting in the first two fully-connected layers it employs a regularization method called "dropout".

Another important feature of the *AlexNet* is the use of ReLU (Rectified Linear Unit) Nonlinearity: thanks to this activation function, the deep *CNN* can be trained much faster than using the saturating activation functions like tanh or sigmoid.

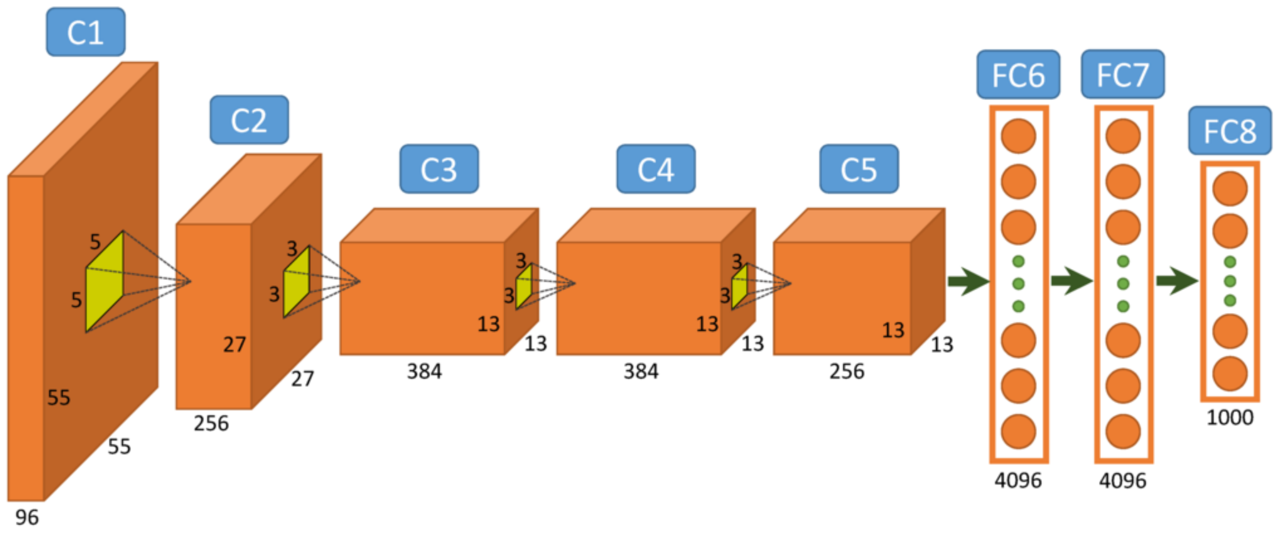


Figure 2 - AlexNet architecture

Caltech 101

In this homework Caltech 101 dataset is used to train and test *AlexNet*.

This dataset was created in September 2003 and compiled by Fei-Fei Li, Marco Andreetto, Marc 'Aurelio Ranzato and Pietro Perona at the California Institute of Technology.

Caltech 101 contains a total of **9144** images, split between 101 distinct object categories plus one background class, that in our case is the BACKGROUND_Google class. Each object category contains between 40 and 800 images. Common and popular categories such as faces tend to have a larger number of images than others.

Each image is about 300x200 pixels. Images of oriented objects such as airplanes and motorcycles were mirrored to be left to right aligned and vertically oriented structures such as buildings were rotated to be off axis.

Figure 3 - Examples of Caltech 101 images

1 – Data Preparation

The first step of the assignment consists of splitting Caltech 101 dataset into training and test set, filtering images belonging to BACKGROUND_Google class and applying data preprocessing transformations to loaded pictures.

Without BACKGROUND_Google category, dataset consists of **8677** pictures that are going to be divided in the training and test set according to the content of “train.txt” and “test.txt” files provided. Then, training set is in turn equally divided into two subsets (training set and validation set) characterized by the same number of images for each class. So, final sets are composed in this way:

- Training set: **2892** images
- Validation set: **2892** images
- Test set: **2893** images

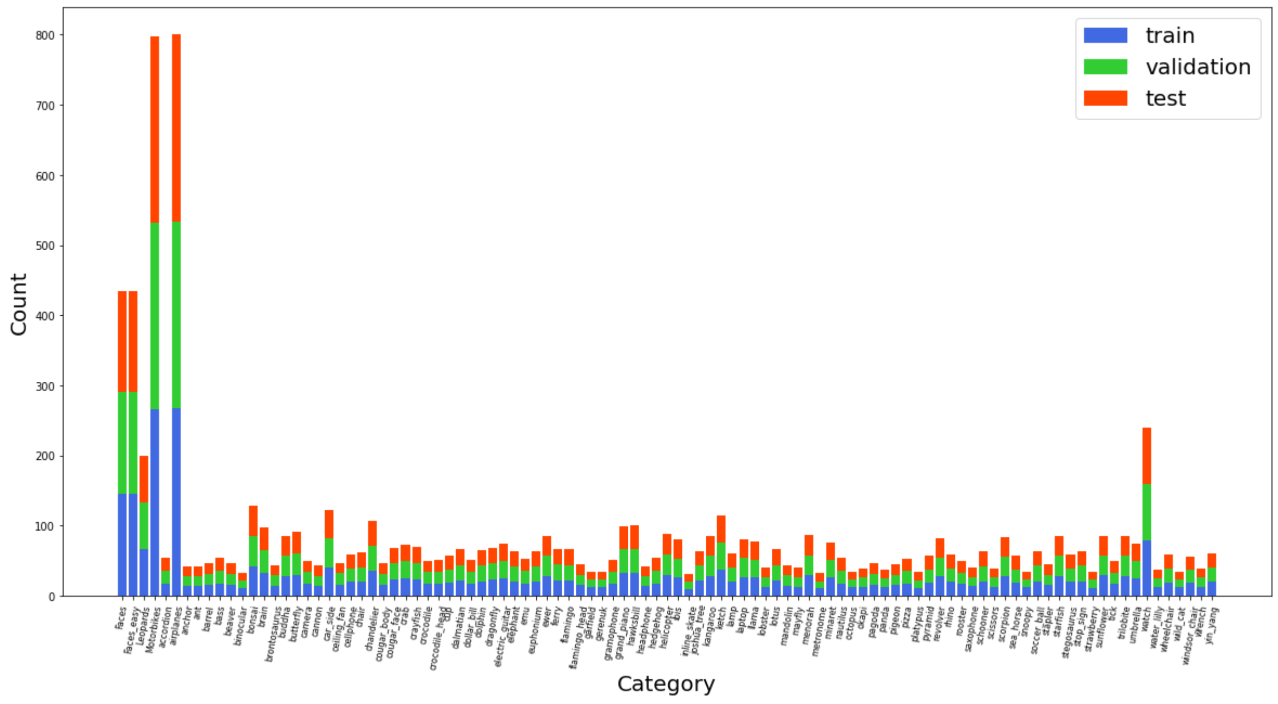


Figure 4 - Number of images by category in Caltech-101

The previous plot shows the image distribution among Caltech-101 categories. As we can see, class partitioning is strongly unbalanced because few categories are represented by a huge number of images, while majority of these does not contain many pictures: indeed, most categories only have about 50 images (which typically is not enough for a neural network to learn to high accuracy).

Analyzing 5 most represented categories and 5 least represented ones we can see that total number of images per class is very different and, in some cases, is very far from mean value of **86** pictures per category.

Top 5 categories

Category	*Training set*	*Validation set*	*Test set*	*Total*
airplanes	267	267	266	800
Motorbikes	266	266	266	798
Faces	145	145	145	435
Faces_easy	145	145	145	435
watch	79	80	80	239

Bottom 5 categories

Category	*Training set*	*Validation set*	*Test set*	*Total*
platypus	11	11	12	34
wild_cat	12	11	11	34
binocular	11	11	11	33
metronome	11	10	11	32
inline_skate	10	11	10	31

Statistics presented in the previous table show firstly as images belonging to the same class are rightly balanced among sets, and secondly as the biggest total number of images per category is very far from the smallest one: indeed, “airplanes” class contains 800 images, while “inline_skate” only 31.

On the other hand, also looking again previous bar plot, it is possible to notice that this evident imbalance concerns only few categories, while most of them are very similar in number of pictures.

After dataset splitting, images are processed by some transformations.

Transformations provided by starting code are taken from *torchvision.transforms* library by PyTorch and perform following operations:

- **Resize**: images are resized, so now the shortest side has a length of 256 pixels while the other side is scaled to maintain the aspect ratio of the images.
- **Center Crop**: images are cropped into a 224 by 224 pixels square images taken from the center of the original ones.
- **Conversion to tensor**: PIL (Python Imaging Library) images are converted to tensors, so each picture is coded into numbers. An appropriate function separates the three colors that every pixel of the pictures is comprised of, that are red, green and blue. This operation essentially turns one image into three images (one tinted red, one green and one blue). Then, the pixels of each tinted image are converted into the brightness of their color, from 0 to 255. These values are divided by 255, so they can be in a range of 0 to 1. Now images are data structures composed by lots of numbers and are called Torch Tensors.
- **Normalization**: tensor images are normalized with mean and standard deviation so that *CNN* can perform better. There are three values in the mean and standard deviation to match each RGB picture (initially all three values are equal to 0.5, in both cases)

2 - Training from scratch

Second step of the assignment consists of training from scratch the *CNN*.

In general, training a deep network from scratch requires a very large labeled data set to design an architecture able to learn features and model. This technique is good in case of large number of output categories, but due to the large amount of data and slow rate of learning, these networks typically take a lot of time to be trained effectively and so training from scratch is the less common strategy. Clearly, using a small dataset, model accuracy will not be satisfactory (as in this case).

At first, *CNN* is initialized by random weights, so this basically means that the network is completely guessing. Once it makes its prediction, it will check how wrong it is using a loss function and then update its weights to make a better prediction next time.

In this case, loss function is represented by *Cross-entropy*. In information theory, it represents the measure of the difference between two probability distributions for a given random variable or set of events. Here, probability distributions (*p* and *q)* are respectively the distribution of true labels and the estimated probability distribution of the current epoch model for each label.

So, *cross-entropy* loss can be described as:

$$H(p,q) = -\sum_{x} p(x)\, log\, q(x)$$

Where *x* represents one of Caltech-101 classes at each iteration.

Clearly, a lower loss is better, because corresponds to higher probability values.

Losses are averaged across observations for each minibatch and are studied at every epoch both when entire training dataset is passed forward and backward through the neural network.

At first step of the first epoch of training phase, the same probability is given to all classes. This value corresponds to a random extraction of one class from the set and it is equal to 1 over the number of classes. So, theoretically, at the beginning loss should be:

$$L = -\ln\left(\frac{1}{101}\right) \approx 4.615$$

where *ln* is the natural logarithm.

Once the code is implemented correctly, it is possible to verify this theoretical result performing the first running of python program.

First training of the network is tuned by hyperparameters provided, that are summarized in the following table.

Learning Rate	0.001
Batch Size	256
Number of Epochs	30
Step Size	20
Gamma	0.1
Optimizer	SGD
Momentum	0.9
Weight decay	5e-5

Results about loss trend computed on training and validation set are shown below.

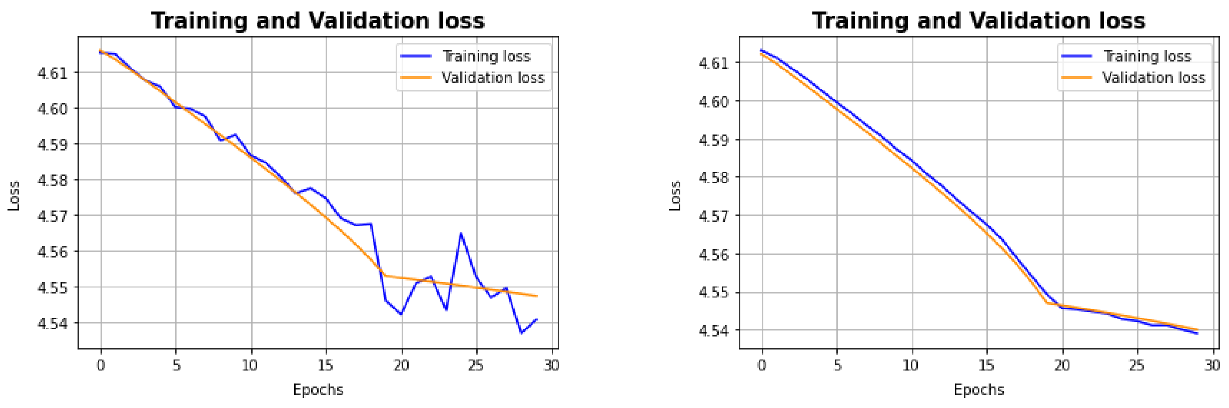


Figure 5

As we can see, starting loss is strongly near to theoretical value calculated before. Moreover, as expected by learning process, loss value is overall decreasing, firstly faster and then slower (due to learning rate reduction).

As said before, two plots above show training and validation loss, but are not calculated in the same way: indeed in the first case training cross-entropy loss represents the average among samples of the only last mini-batch considered, while validation cross-entropy is computed considering all validation batches in that epoch.

Instead, in the second plot, both training and validation loss are computed over all batches of respective dataset. Contrary to first loss representation, where training trend is subject to last batch bias, in the second one it is possible to notice that curves are strongly similar and practically overlapping.

Relatively high values and the fact that training loss continues to decrease at the end of the plot reveal that the model is unable to learn the training dataset at all. This behavior is called *underfitting*.

This is also noticeable from low accuracy scores obtained, that converges to a value of about **9%** both on training and validation set at the end of 30th epoch.

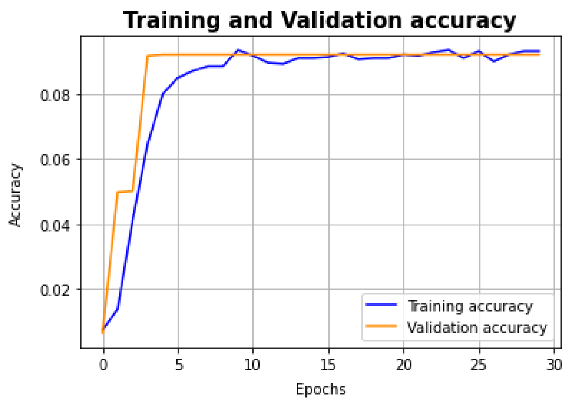
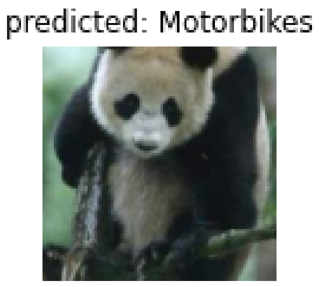


Figure 6

Finally, performing final model found during previous phase on test set, accuracy score obtained is approximately equal to **9%**, in line with one caught before.

Comprehensibly, insufficient quality of training phase leads poor results in terms of class prediction: indeed, model classifies majority of images as belonging to most represented categories. It is possible to be noticed from random samples extracted.



Very similar loss and accuracy scores are also obtained using different splits of dataset, so, after several trials, it is possible to say that the distribution of images among training and validation set does not affect results significantly.

After this first attempt, model is trained from scratch with different sets of hyperparameters.

During next trials, training loss is going to be always considered as the average of training (or validation) set batch losses computed at every epoch. Results reported correspond to final loss and accuracy mean value got by running two times the code.

Every trial performed is characterized by changing learning rate and one parameter among step size, optimizer and number of epochs, keeping all other parameters the same.

As a start, learning rate is increased by one order of magnitude and ten epochs are added, in an attempt to improve accuracy score.

So, hyperparameters used appears as follows:

Learning Rate	0.01
Batch Size	256
Number of Epochs	40
Step Size	20
Gamma	0.1
Optimizer	SGD
Momentum	0.9
Weight decay	5e-5

And here results are plotted.

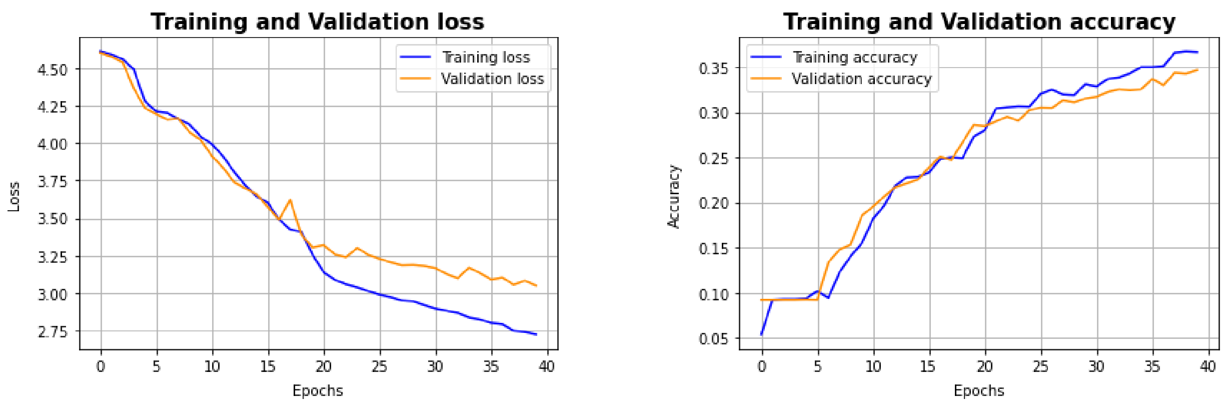


Figure 7

Learning rate hyperparameter points out the amount that the weights are updated during training and controls the rate at which the model learns, so higher values allow the model to be faster adapted to the problem.

And that is what is happened increasing learning rate to 0.01: now cost function achieves a final value about **3.05** on validation set and about **2.75** on training set, that are smaller than scores got before.

Even accuracy gets better thanks to this net configuration, being equal to **34.64%** on validation set. So now the model is almost four times more accurate than before!

Following attempt is characterized by a further gain of learning rate and by a step size reduction in order to drop the learning rate by a factor every 15 epochs, and so 2 times during the 40 epochs.

Summarizing:

Learning Rate	0.1
Batch Size	256
Number of Epochs	40
Step Size	15
Gamma	0.1
Optimizer	SGD
Momentum	0.9
Weight decay	5e-5

Results are presented below:

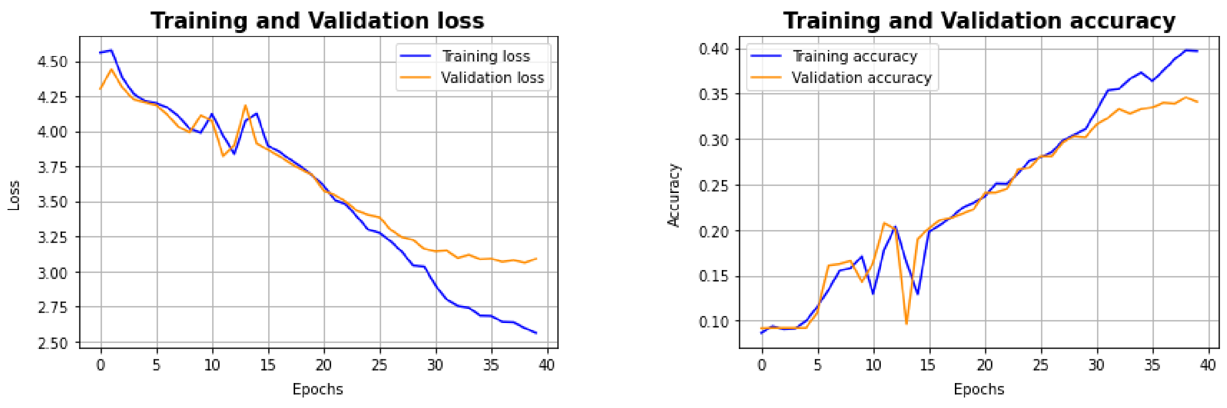


Figure 8

As we can see, learning rate is now too large, then it involves very large weight updates and the performance of the model (as shown by losses on training and validation dataset) oscillates over initial epochs. Oscillating performance is said to be caused by weights that diverge.

It is also noticeable that the model gets best improvements when learning rate decreases (after 15th epoch). Towards the end of the training phase (after 30th epoch) they begin to see signs of *overfitting* due to loss divergence. Loss score on training set is about **2.7**, on validation set is about **3.1** and final accuracy on validation set is about **34%**.

It must be said that performing several times the model by this learning rate very different plots about loss and accuracy are obtained, as evidence of the impossibility to converge to the same optimal solution. Indeed, sometimes losses diverged after few epochs and accuracy collapsed.

Nesterov Accelerated Gradient

Setting again learning rate to 0.001, optimizer is tried to be changed.

During previous trials, *SGD* (Stochastic Gradient Descent) with standard Momentum is adopted, but now *NAG* (Nesterov accelerated gradient) is used.

SGD is an iterative optimization algorithm that estimates the error gradient for the current state of the model using examples (mini-batches) from the training dataset, then updates the weights of the model using the back-propagation of errors algorithm, referred to as simply backpropagation.

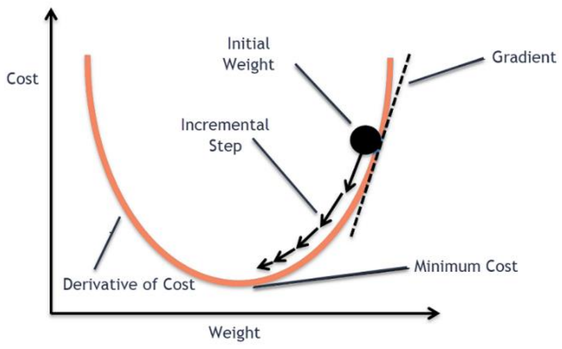


Figure 9 - 2D representation of SGD

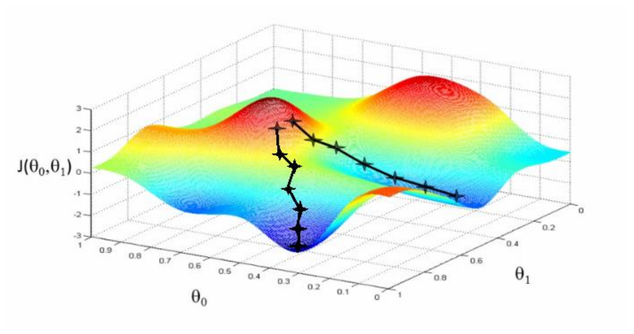


Figure 10 - 3D representation of SGD

If SGD uses standard momentum, it accumulates exponentially decaying moving average of past gradients and continues to move in their direction, so instead of using only the gradient of the current step to guide the search, standard momentum also accumulates the gradient of the past steps to determine the direction to go.

On the other hand, *NAG* exploits Nesterov momentum. The difference between Nesterov and standard momentum is where the gradient is evaluated: Nesterov’s momentum calculate the gradient not with respect to the current step but with respect to the future step and then adds a correction factor to the gradient, updating the weights.

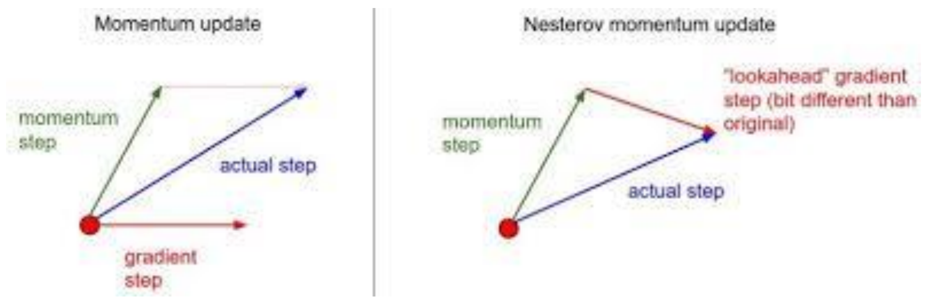


Figure 11 - Standard momentum and Nesterov's momentum different approach

But despite of optimization algorithm change, running code with the first set of hyperparameters presented provides very similar results, without significantly optimizing the descent faster.

Adaptive Moment Estimation

After *NAG*, another optimizer called [*Adam*]() (Adaptive Moment Estimation) is used.

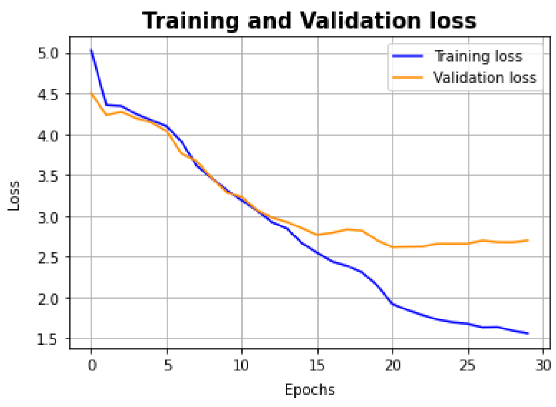
Adam is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision. It is a solution that combines advantages of two other methods, that are Adaptive Gradient Algorithm (*AdaGrad*) and Root Mean Square Propagation (*RMSProp*). Indeed, Adam not only adapts the parameter learning rates based on the average first moment (the mean) as in *RMSProp*, but also makes use of the average of the second moments of the gradients (the uncentered variance).

Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and some parameters (β_1 and β_2) control the decay rates of these moving averages.

For this trial, hyperparameter set is so composed:

Learning Rate	0.001
Batch Size	256
Number of Epochs	30
Step Size	20
Gamma	0.1
Optimizer	Adam
Momentum	0.9
Weight decay	5e-5
β_1	0.9
β_2	0.999

Results are shown below.



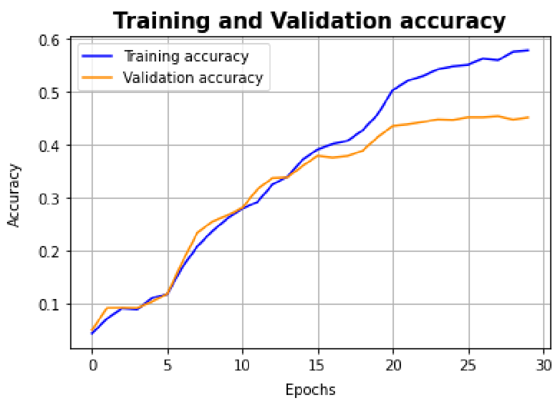


Figure 12

Analyzing first graph we can see that at the 30th epoch plot of training loss continues to decrease, while the plot of validation loss decreases to a certain point (20th epoch) and then begins slightly increasing. This behavior reveals *overfitting*, so the model has learned the training dataset too well and now is less good to generalize to new unseen data, resulting in an increase in generalization error.

Anyway, final training loss is about **1.5** and validation loss is about **2.7**, that are the lowest value obtained until now.

Regarding accuracy trend, it is possible to notice that after 20th epoch it keeps growing on training set, while it slows improving on validation set.

Final accuracy score on validation set is equal to **46.2%,** that is much better than score obtained from *SGD* (**9%**) with the same parameters.

Last attempt is performed decreasing learning rate by one order of magnitude, increasing the number of epochs and using Adam optimizer.

Learning Rate	0.0001
Batch Size	256
Number of Epochs	50
Step Size	20
Gamma	0.1
Optimizer	Adam
Momentum	0.9
Weight decay	5e-5
β_1	0.9
β_2	0.999

Results are represented by following plots.

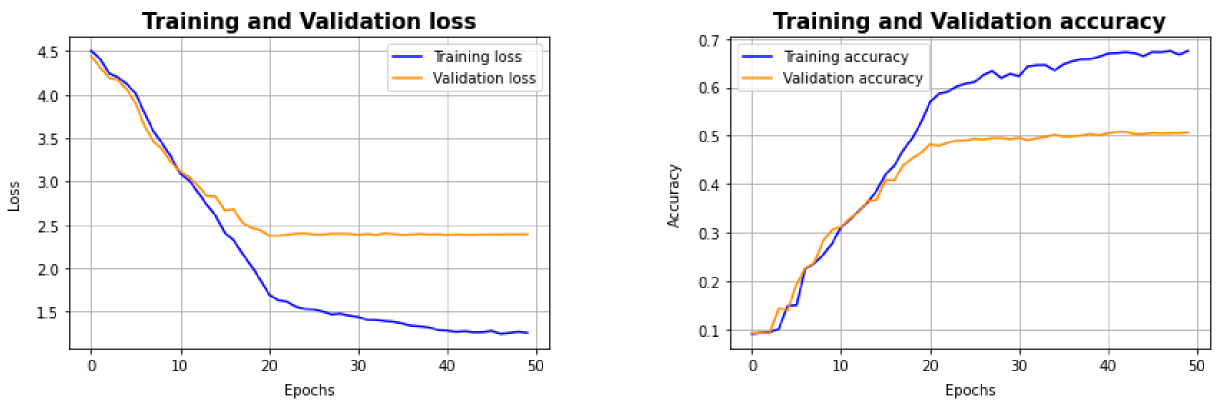


Figure 13

Decreasing learning rate by a factor of ten, final validation loss is now equal to **2.4**, that is lower than value obtained before. As expected, using a greater number of epochs than previous experiment, it is more evident the gap between losses, so in this case model is more affected by *overfitting*. Indeed, network performances on validation set stop improving after 20th epoch, while continue on training set. Anyway, accuracy on validation set increases up to **50.62%**, that is better than before.

As a conclusion of this first part of the report, it is possible to say that training a network from scratch is not a good choice in case of small datasets: in fact low accuracy results obtained reveal that model is not sufficiently reliable. So, in order to get better performances, it is required to use other two techniques, which are transfer learning and data augmentation.

3 – Transfer Learning

As humans do not learn everything from the ground up and leverage and transfer their knowledge from previously learnt domains to newer domains and tasks, likewise in the context of deep learning there is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones.

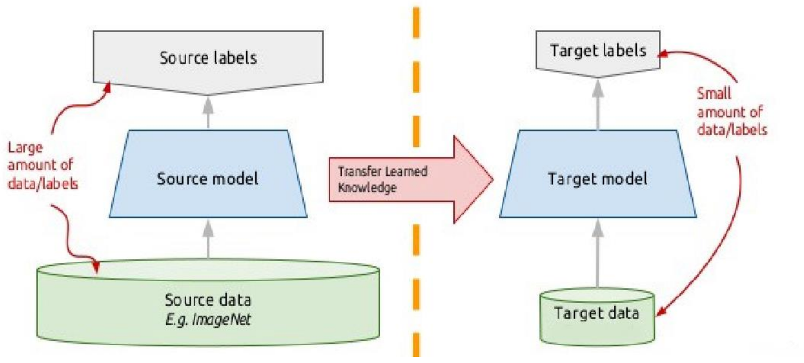


Figure 14 - Idea behind Transfer Learning

So, in practice, in very few cases an entire Convolutional Neural Network is trained from scratch (with random initialization), especially if we have not a dataset of sufficient size. Instead, it is common to pretrain starting weights of a *ConvNet* on a very large dataset.

Here, *AlexNet* will be pretrained by using ImageNet, a large visual database of human annotated photographs which contains over 14 million images divided into more than 20.000 categories.

Before proceeding further, it is necessary to change batch normalization on training, validation and test set according to ImageNet statistics, that are:

- *Means* = [0.485, 0.456, 0.406]
- *Stds* = [0.229, 0.224, 0.225]

The two vectors are composed by three values, each of them represents the mean (or the standard deviation) of one of RGB colors (Red, Green or Blue) on ImageNet pictures.

After these preliminary operations, we are ready to apply **finetuning**.

So, now we start with the pretrained model and update all model's parameters for our task (in essence retraining all the layers of the *ConvNet*).

To see the effectiveness of this method, first set of hyperparameters performed is composed by values provided initially, that are shown on next page.

Learning Rate	0.001
Batch Size	256
Number of Epochs	30
Step Size	20
Gamma	0.1
Optimizer	SGD
Momentum	0.9
Weight decay	5e-5

Results obtained are plotted below.

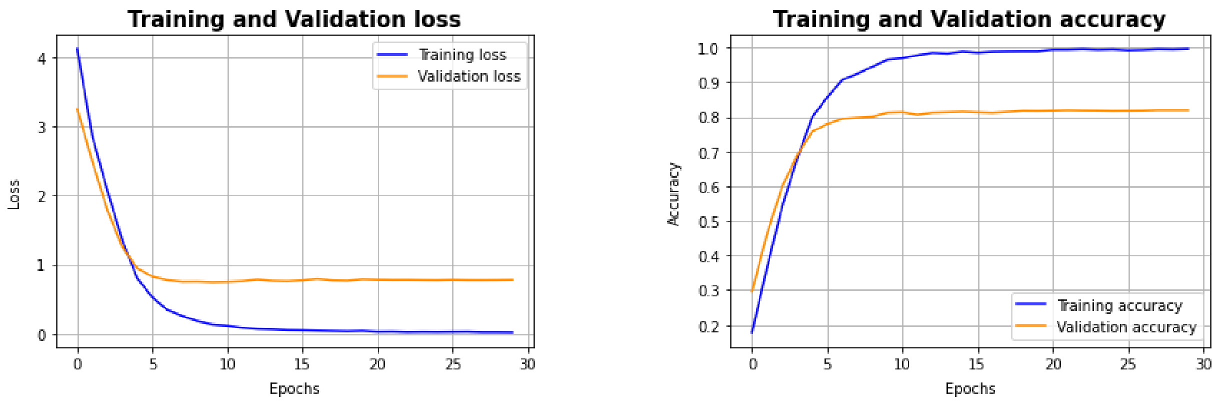


Figure 15

Positive effects of transfer learning are evident: validation loss is now about **0.78** and respective final accuracy is **81.95%**, while training loss converges to zero.

Even accuracy score on test set is extremely better, achieving a value of **82.85%** by using final model obtained from training phase.

Both loss and accuracy converge to best values within 15th epoch, so now the model can learn faster.

Then, next attempt will be performed increasing learning rate by one order of magnitude and using 15 as number of epochs and 10 as step size.

Summarizing parameters used are:

Learning Rate	0.01
Batch Size	256
Number of Epochs	15
Step Size	10
Gamma	0.1
Optimizer	SGD
Momentum	0.9
Weight decay	5e-5

Results are presented below.

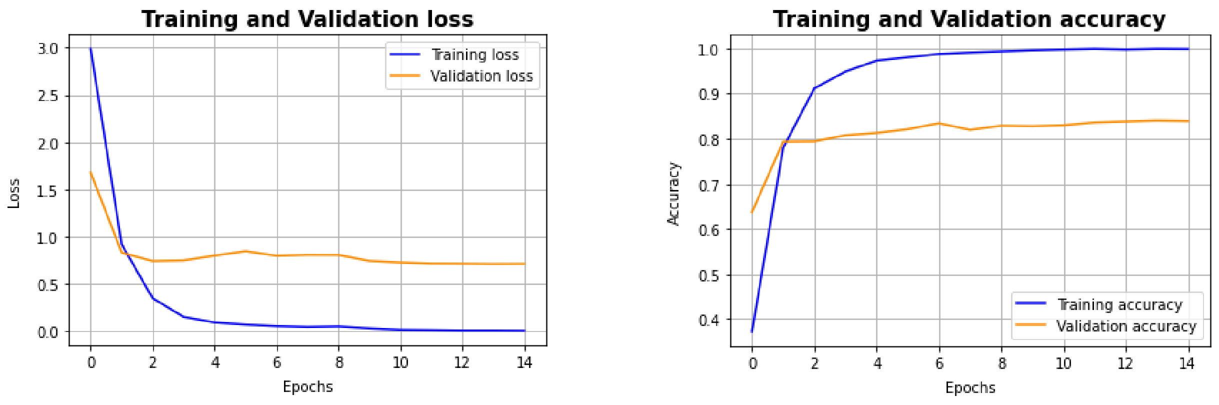


Figure 16

As expected, a greater learning rate leads a faster convergence to the optimal solution: in fact, already after 3rd epoch a very low loss score is obtained. Between 4th and 9th epoch it gets worse and then it improves a little after the step size, when learning rate decrease.

Anyway, loss validation value is now lower than one got during previous experiment, so it is about **0.73**. Even validation set accuracy is quietly better and is equal to **83.74%**.

In order to prevent loss validation worsening between 4th and 9th epoch, step size is decreased, keeping all other parameters the same.

Results are very similar but slightly better (final validation loss is now about **0.69** and respective accuracy is equal to **83.95**%).

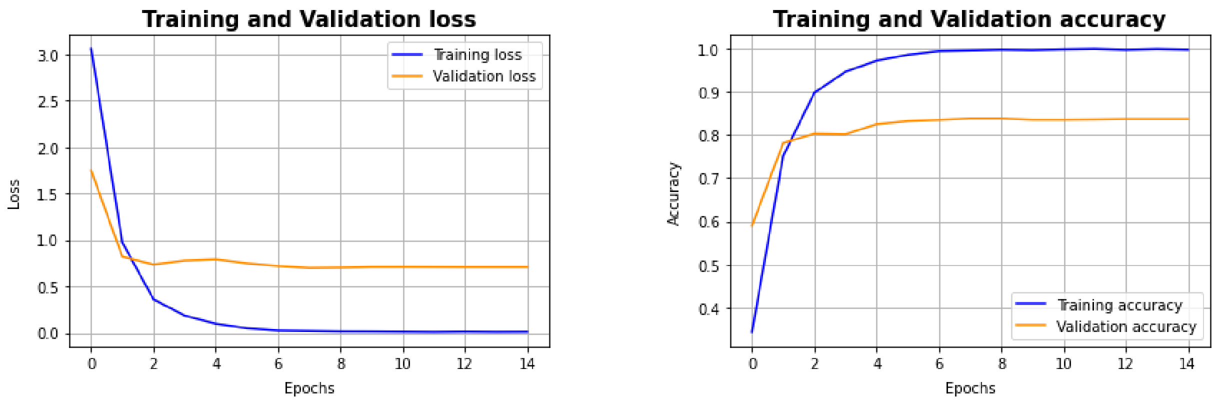


Figure 17

Fourth transfer learning attempt is performed by decreasing learning rate to 0.0001.

Learning Rate	0.0001
Batch Size	256
Number of Epochs	40
Step Size	30
Gamma	0.1
Optimizer	SGD
Momentum	0.9
Weight decay	5e-5

But now results are not so good as before, because model learns too slowly to converge to previous loss and accuracy results, neither increasing to 40 the number of epochs.

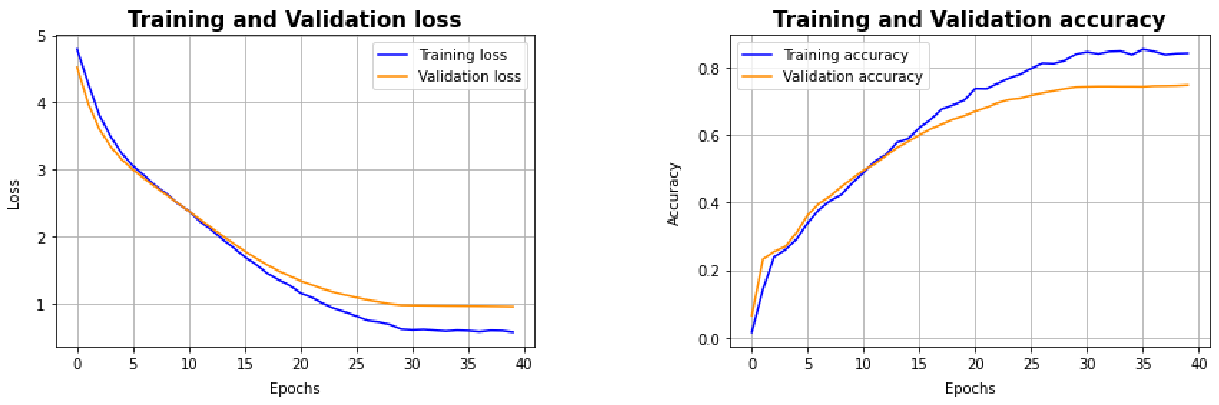


Figure 18

Validation loss stops to **0.96** and validation accuracy got applying final model from training phase is about **75%**.

After these, some of these sets of hyperparameters are performed using *Adam* optimizer. All the results are then collected in the following table and represent the mean value obtained by performing *AlexNet* two times for each configuration, highlighting as best the one with lowest validation loss result. In this case, lowest loss coincided also with the highest accuracy on validation and test set.

Learning Rate	*Optimizer*	*Num epochs*	*Step size*	*Training Loss*	*Validation Loss*	*Training Accuracy*	*Validation Accuracy*	*Test Accuracy*
0.01	SGD	15	5	0.013	0.686	99.64%	83.95%	84.24%
0.01	SGD	15	10	0.006	0.725	99.86%	83.74%	84.12%
0.001	SGD	30	20	0.021	0.777	99.61%	81.95%	82.85%
0.001	Adam	30	20	0.013	1.758	99.72%	72.05%	70.16%
0.0001	SGD	40	30	0.573	0.956	84.34%	75.19%	75.74%
0.0001	Adam	40	30	0.002	0.753	99.89%	83.85%	83.40%

As we can see, the first usage of *Adam* optimization algorithm (with *LR* = 0.001) provides the worst results of transfer learning trials, but still better than all ones obtained training the model from scratch.

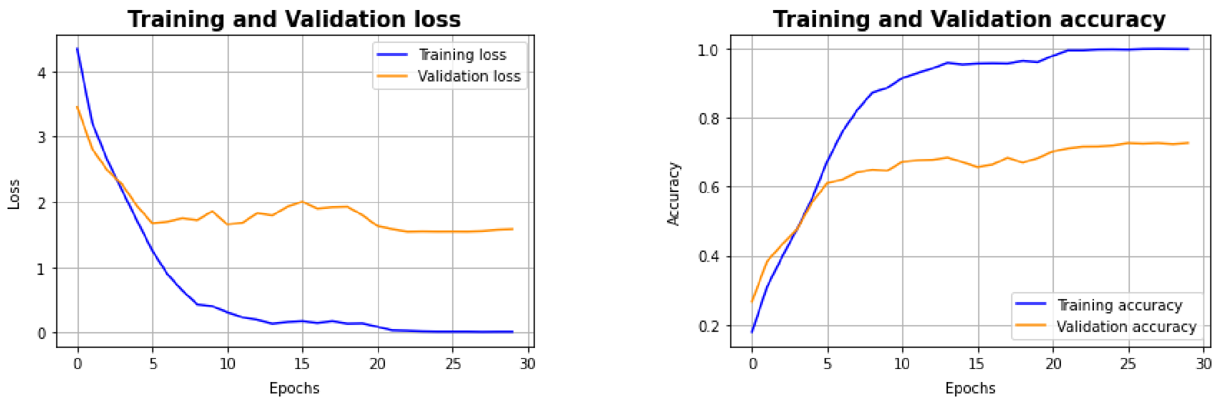


Figure 19

Moreover, it is possible to notice that loss validation line diverges from training one, so that reveals *overfitting*.

Second usage of Adam provides us very good results that are very similar to best ones obtained with SGD optimizer with LR = 0.01.

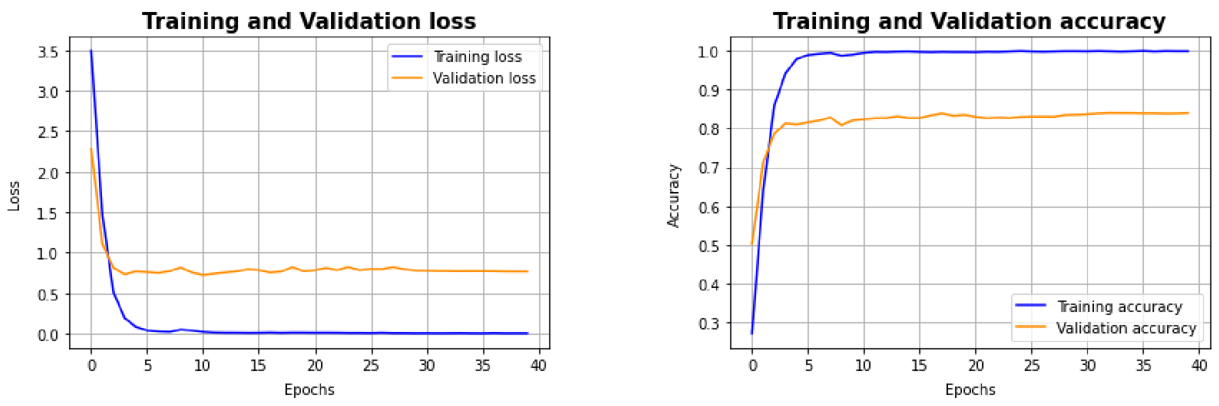


Figure 20

Figure 20 shows that the network converges rapidly (within 5th epoch) to approximate best loss and accuracy values, so we could have been used a smaller number of epochs.

At the end of tests, it is possible to say that transfer learning technique has a very important positive impact on *AlexNet* performances, getting validation loss scores from about **2.4** to **0.7** and accuracy results from about **50%** to **84%** with respect to training from scratch**.**

As shown in previous table, best outcomes are obtained from the following set of hyperparameters.

Learning Rate	0.01
Batch Size	256
Number of Epochs	15
Step Size	5
Gamma	0.1
Optimizer	SGD
Momentum	0.9
Weight decay	5e-5

Then, it is applied *fine-tuning* to the *ConvNet* with this configuration, but this time keeping earlier layers fixed (due to overfitting concerns) and only *fine-tuning* fully connected layers.

Freezing convolutional layers, validation loss is lower than the experiment performed training whole network: indeed, final value is now equal to **0.631** (before it was **0.686**), while training loss is greater (**0.15**).

Accuracy trend is very similar to previous experiment and final score on training set is equal to **96.58%** (slightly worse than before), while accuracy on validation set is **82.78%** (slightly lower than before).

Accuracy on test set is **82.29%**, in line with validation one and with previous experiment.

Following two plots show the results of this experiment.

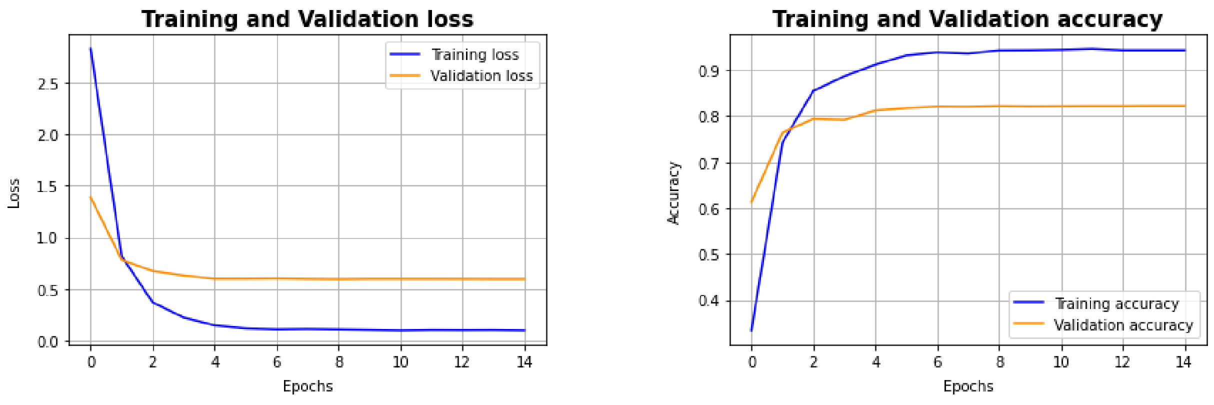


Figure 21

Freezing convolutional layers has not improved network accuracy significatively, but the main benefit consists of saving time as demonstrated by the only **5** minutes and **36** seconds to train the network.

After this, only the 5 convolutional layers of *AlexNet* are trained.

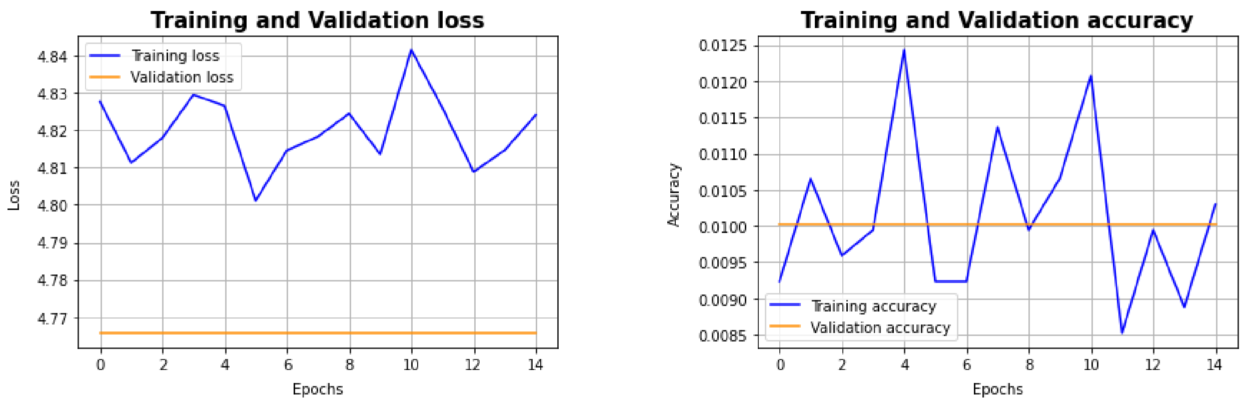


Figure 22

As we can see from graphs, loss and accuracy on validation set have constant trends that stand about **4.76** for loss and **1.00%** for accuracy.

Instead, loss and accuracy on training set have an oscillatory trend between **4.80** and **4.84** for loss and between **0.85%** and **1.25%** for accuracy score.

So, freezing the fully-connected layers extremely negative results are obtained due to the fact that the final fully-connected layers are generally assumed to capture information that is relevant for solving the specific task and if they are not trained, they will contain elements that indicate which features are relevant to classify an image into one of 1000 object categories from ImageNet while features to recognize and distinguish Caltech 101 classes.

On the other hand, the earlier features of the *ConvNet* contain more generic features that should be useful to many tasks, so, freezing convolutional layers, it is possible to obtain good results.

4 – Data Augmentation

After transfer learning, another technique used to prevent overfitting and to improve *AlexNet* performance is called *Data augmentation*.

Data augmentation is a technique to artificially expand training set size by the creation of new training data from existing training data. So, it involves the creation of modified versions of images in the dataset. In this way, it is possible to improve the ability of the fit models to generalize what they have learned to new images.

Transforms include a range of operations from the field of image manipulation, such as shifts, flips, zooms, and much more.

Below are presented examples of image transformations of some randomly chosen picture from Caltech-101 dataset.

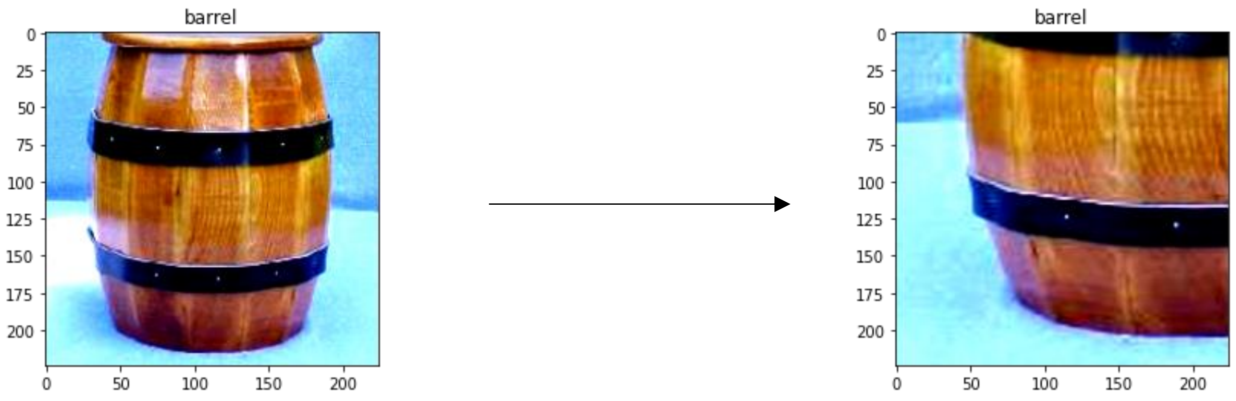


Figure 23 – Random resized crop of 224x224 pixels

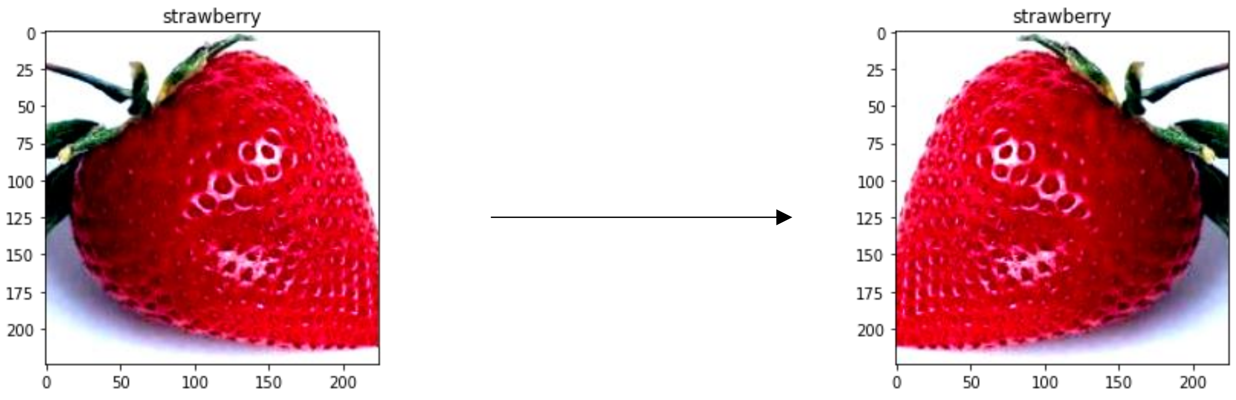


Figure 24 - Horizontal flip

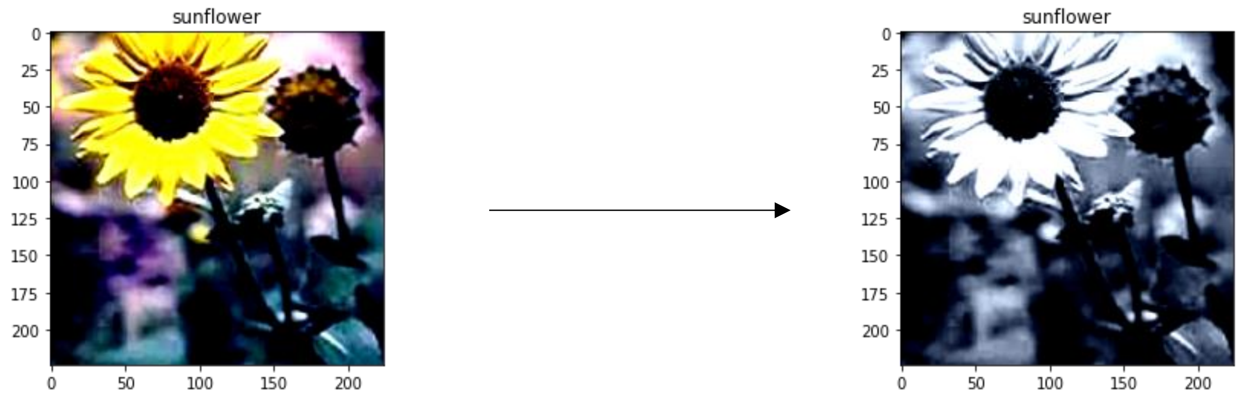


Figure 25 - Gray scale colors conversion

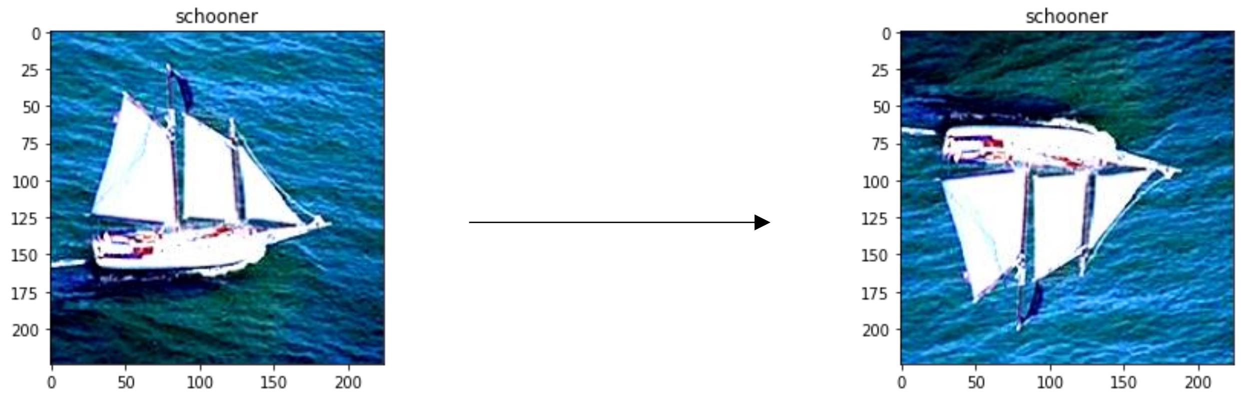


Figure 26 - Vertical flip

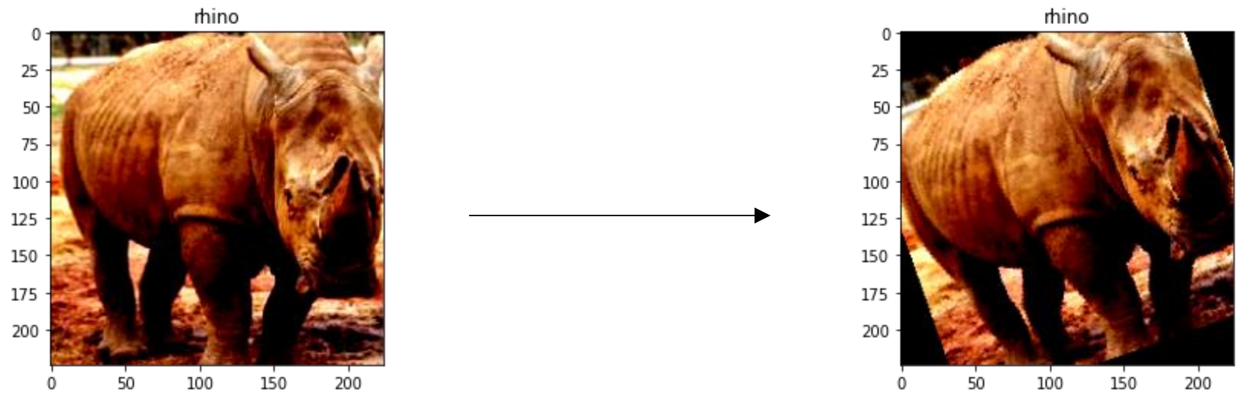


Figure 27 – Rotation

Results presented in previous points of report are obtained only performing a resize and a center crop of each image, as described during data preparation treatment, but now different sets of transformations are applied.

In the following trials the best hyperparameter set found before is used, training all the layers of the *ConvNet* and applying different transformation functions only to training split of dataset.

Outcomes are computed as means of results obtained running two times code and are summarized in the table below.

Transformation functions	*Training Loss*	*Validation Loss*	*Training Accuracy*	*Validation Accuracy*	*Test Accuracy*
RandomResizedCrop(size = 224), RandomHorizontalFlip(probability = 0.5)	0.474	1.020	85.56%	73.78%	82.55%
Resize(size = 256), CenterCrop(size = 224), RandomGrayscale(probability = 0.5)	0.039	0.712	97.46%	82.88%	84.05%
Resize(size = 256), CenterCrop(size = 224), RandomRotation(degrees = (-90, 90)), RandomVerticalFlip(probability = 0.5)	0.294	1.317	89.88%	69.77%	74.12%

By these choices, it is not possible to appreciate any improvements. Indeed, in the best case, validation loss is very similar to one got without data augmentation and test accuracy is almost the same.

Worst case is that in which rotation and vertical flips of images decrease of about 10% precision on test accuracy.

Last experiment consists of applying best transformation functions found on training split, adding *TenCrop* function to evaluation transformations on test dataset.

This function from PyTorch library crops PIL images into four corners and the central crop plus the flipped version of these.

To avoid exceeding GPU memory limit, batch size is now changed to **64**.

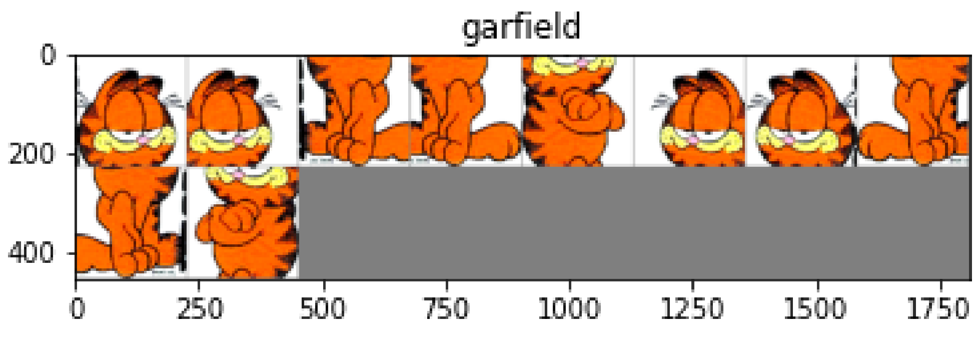


Figure 28 – Example of split in ten crops

So, it is considered the mean of the features over the crops. The evaluation in this more accurate way provides a lower accuracy on test images, achieving a score equal to **79.80%**.

5 – (Extra) Beyond AlexNet

Within previous chapters of this report, all results discussed are obtained from *AlexNet*. But *AlexNet* it is not the only possible *ConvNet* architecture: indeed, in recent years, we witnessed the birth of numerous *CNNs*, which have become progressively deeper and more accurate, getting better results in terms of top-1 score (check about equality of the class having the highest probability and the target label) and top-5 score (check about the comprehension of target label in one of top 5 predictions) during the different editions of ImageNet Large Scale Visual Recognition Challenge (*ILSVRC*).

Some of these pretrained models are performed, tuning best hyperparameters found on *AlexNet* experiments, preprocessing images according to the best configuration from data augmentation step and decreasing the batch size to **64** in order to avoid out of GPU memory problems.

As first, *VGG-16* is applied. *VGG-16* is the convolutional neural network model by some researchers from the University of Oxford. It was one of the famous models submitted to ILSVRC-14, competition in which achieved 92.7% top-5 test accuracy.

Second model performed is *ResNeXt-50* by UC San Diego and Facebook AI Research (FAIR). This is an evolution of *ResNet* model, that implies one more dimension called "*cardinality*". ResNeXt became the 1st Runner Up of ILSVRC 2016 classification task.

Results are collected in the table below.

Model	*Training Loss*	*Validation Loss*	*Training Accuracy*	*Validation Accuracy*	*Test Accuracy*
VGG-16	0.009	0.460	99.47%	89.25%	90.34%
ResNeXt-50 (32 x 4d)	0.009	0.169	99.75%	95.66%	95.48%

As we can see, in both cases there are significant improvements of loss and accuracy score, but ResNext-50 turns out to be the best.

Indeed, through ResNext-50 it is possible to get a very low validation loss (**0.169**) and an accuracy on test set even over **95%**. Trends of these magnitudes are represented in the following graphs.

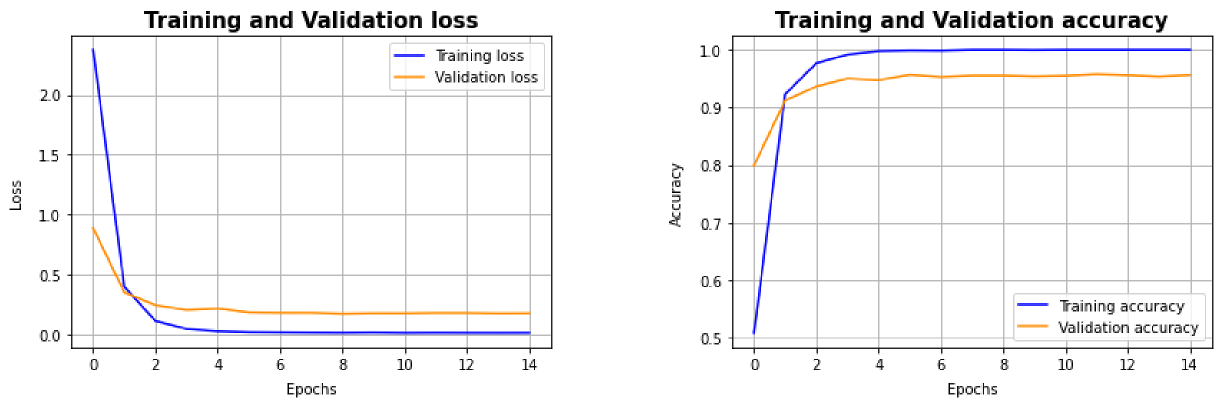


Figure 29

References

[1] AlexNet Documentation

https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] Caltech 101 dataset

http://www.vision.caltech.edu/Image_Datasets/Caltech101/

[3] PyTorch image transformation library

https://pytorch.org/docs/stable/torchvision/transforms.html

[4] Cross entropy

https://en.wikipedia.org/wiki/Cross_entropy

[5] Nesterov Accelerated Gradients

https://arxiv.org/pdf/1607.01981.pdf

[6] Adaptive Moment Estimation

https://arxiv.org/pdf/1412.6980.pdf

[7] ImageNet dataset

www.image-net.org

[8] ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

http://www.image-net.org/challenges/LSVRC/