**Borzillo Caterina**, matricola **1808187**

# MACHINE LEARNING COURSE
HOMEWORK 2

## Introduction: Image Classification Problem
The most popular artificial neural networks used for image analysis are Convolutional Neural Networks, also called CNN. In particular,  I'm going to use a CNN to solve an image classification problem starting from a dataset containing about 8000 images. The images the model has to classify  belong to 8 classes and the subjects of the pictures are objects typically used in a home environment. Thanks to the random combination of the seven numbers of my matricola, the sub-classes of pictures my model has to classify are: "cake_server", "marmalades", "pasta_bowl", "plastic_food_container", "squeegees", "tangerines", "tea_drink_bottle" and  "vegetable_chips_&_crisps".

## CNNs
As I mentioned, I will use Convolutional Neural Networks which are complex deep neural networks used for image classification because of its high accuracy. A CNN is based on a hierarchical model that works on building a structure composed of an input layer, an output layer and few or many hidden layers which act following some patterns until the output is processed.

## Where and how
I will perform this task using the open source TensorFlow library on Colab and in order to manipulate all the important features and high-level APIs concerning neural networks I'll use Keras library, which supports almost all the models of a NN. I choose Keras on Tensorflow because Keras' simple use joins with its flexibility since Keras itself integrates deeply and completely with low-level Tensorflow functionality.

## Data preparation and data pre-processing
First of all, I have to deal with the image preparation step to get the images ready to be fed to the deep learning network. To do this I need to organise the data in order to structure the directories with images within my drive; so, I create a directory called 'dataset' in which there are initially all the images of the dataset divided by the 8 classes. Then, through some scripts I make two sub-directories 'train' with the training data and 'test' with the test data. The number of images in the test set is approximately one third of the number of images in the training set for each class of objects; that's because it's important for the model to practice on a large number of training images before being ready to classify them in the most correct way. Images were put together in the train set and the test set after being mixed with each other.

Now, one of the most important things we must take care when we use Convolutional Neural Networks to classify images is the data pre-processing step. It consists in converting and transforming each input image data into floating-point tensor before it is fed into Convolutional Neural Networks.

Fortunately, Keras provides utilities to perform this process in a very easy way. The Keras module with image-processing tools contains the class ImageDataGenerator, which automatically lets you turn image files on drive into batches of preprocessed tensors.

The first thing to do is to rescale the images in order to bring multiple image files to a common scaling for comparison. I rescaled both the training and testing images from the range 0 -255 to 0-1.
Even if the size of the dataset is quite large (about 8000 total images), since the fact that CNN's are models that work better with as many images as possible, I used the data augmentation technique which consists in artificially creating new training data from existing training data. It is made by modifying and manipulating images through some operations like shifts, flips, zooms and others. In particular I applied the zoom, rotation, the width and height shift, and the horizontal flip operations. What the generator does is to run through the image data and apply random transformations to each individual image as it is passed to the model so that it never encounters the same image twice while training. The advantage is also that, since that the model trains different types of images, a bit distorted, a bit modified, it help to learn only the most important feature of the image instead of learning the noisy and meaningless details of a certain training image; in this way, the model will be more able to recognize shapes and informations about images that it has never seen before. Note that in the test generator I only rescaled the data and the reason is the benefit of augmenting the images is only for training data, since that it can decrease the test performance.

Since I will use the MobileNet pre-trained model to solve the classification problem, I often have experienced, instead of the image rescaling, the use of the preprocessing function for inputs of MobileNet model. The use of this function is recommended when it is used for the Transfer Learning technique and when you're not so much fine-tuning the pre-trained model, and you're using almost all the knowledge (namely, the weights of layers) of the pre-trained model.

**Approach used: Transfer Learning**
Now I will focus on the first method that I used for classifying images: Transfer Learning technique. Transfer Learning approach is based on the idea of utilizing knowledge previously acquired for a certain problem or task to solve others related ones. This method is really useful and efficient when we want to reach more accurate results without training a deep network from scratch. With Transfer Learning it's possible to take advantage of the features that the pre-trained model has learned through the different hidden layers; in fact, once the pre-trained model is loaded, we have 2 possibilities: the first is to use all the knowledge of the pre-trained model for our model changing only the last fully connected layer (based on the number of classes of the classification problem), the second is to "freeze" only a certain number of hidden layers of the pre-trained model (that are usually the firsts layers - because they learns the main feature on the image - ) and to add multiples layers in addition to the freezed ones in order to improve our model with customized layers.
The first method I just mentioned is used when we can find a trained model that works very well on its task and we know that that specific task is similar to ours. Regarding this, the expression of "freezing layers" concerns the fact that we don't want to modify  the weights

of the pre-trained model for our task. The benefit of not updating the weights of some layers is the time, because it cut down the computational time for training.
Otherwise, the second approach usually is applied in those cases in which the 2 tasks are different but however there is some information that the new model wants to keep about the "old" model and there is some other information that the new model wants to acquire for its specific and particular task. As I said before, usually, the first layers of the pre-trained model are frozen, and only the end layers will be modified.

**For this classification problem I'll try both Transfer Learning technique implementations and then I'll see the differences between the two approaches.**

**MODEL A: transfer learning model (using MobileNet)**
First, I use MobileNet pre-trained model which is a class of light weight deep convolutional neural networks that are faster and smaller in performance than other popular models. Even if MobileNet is a class of small and low-power models the tradeoff is the accuracy metric, but I'll try it anyway because it still performs very well and the reduction of the accuracy is relatively small. An important thing to take into account is the fact that MobileNet is a model that has not been trained on the dataset I'm using now, but it is trained with ImageNet; it means that it should not be so easy for it to recognize on the first attempt and with a short training time the images which belong to household objects. Even if my result will not be so brilliant as if MobileNet had trained itself for my particular type of images, it's important to know that pre-trained model during its training session learnt something that are very useful for our Transfer Learning model which are the general features (referred to the lower layers of the network). In the end of the model, I take care of adding some output layers (fully connected layers) in order to receive as a result of the classification problem one of the 8 categories of pictures.

So, after data have been pre-processed (in this case I used the preprocessing mobilenet function) with the ImageDatagenerator function we apply the 'flow_from_directory' function to give the model access to files directly from the directory on my drive with the train and the test sets. In 'flow_from_directory' I have to also mention the target size argument, which specifies the dimensions for which the images have to be automatically resized as input for my model, in my case is (224,224). Another important parameter is the 'shuffle' one which by default it's True and it refers to the different order in which we want to give the data on every epoch. The parameter shuffle setted to true is ideal for training data because in this way the model learns more by receiving during its training various scattered categories of images. Regarding the testing set, if we want to compute the confusion matrix it's good to put the shuffle parameter setted to False.

Now I load the MobileNet model from keras application, and I'll set the include_top parameter to False and it means that I'm not including the last layer of MobileNet network, so I can add my customized output layer for my 8-image classification problem. Since MobileNet has been trained with images of ImageNet dataset it's important to set the weights parameter as 'imagenet'.
To complete my transfer learning model I add as final layers a GlobalAveragePooling2D layer, then 2 Dense layers (1024 and 512 units) and then finally a Dense layer with 8 units to

recognize the 8 classes of images. As activation function for the end Dense layer I use the 'softmax' function, which is very useful because it converts the scores to a normalized probability distribution and assigns probabilities to each class in our multi-class problem.

Now there is an interesting thing that has to be specified: the number of trainable layers. Since in this first transfer learning model  I'm just extracting the features learned by the pre-trained model, in this case I want only to set as trainable=True the 3 fully connected layers that I added at the end of my model. In other words, now I'm freezing all the weights of MobileNet model (so I set to trainable=False all the layers of MobileNet) and I'm training only the final layers (namely, only 1,578,504 parameters up to 4,807,368 will be updated).

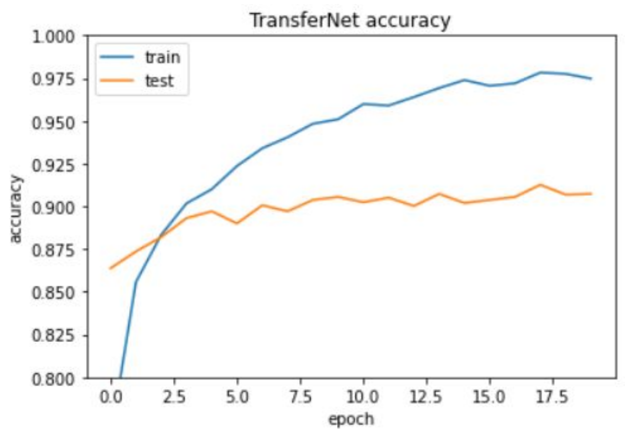
So I have my 'transfer model' obtained by the MobileNet model with the replacement of the last output  layer with a customized output layer written by me. All the layers of my transfer model belong to the MobileNet model except the last and all the trainable layers in my transfer model are the ones I added at the end of the network, while all the layers of MobileNet I have inherited are not trainable.

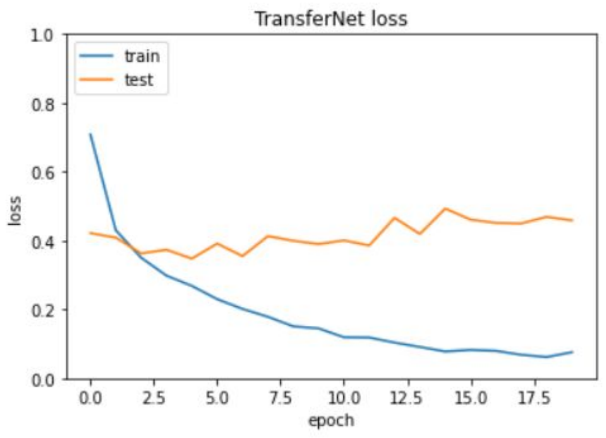
Now I'm going to compile the model and I'm going to use the compile keras function which is included in the group of metric functions, used for judging the performance of my model. In the compile function I specify the Adam optimizer which is a stochastic gradient descent method computationally very efficient. Stochastic gradient descent is an optimization algorithm that estimates the error gradient for the current state of the model using examples from the training dataset and then updates the weights of the model based on the error gradient. The quantity of weights that are updated during training are dictated by the 'learning rate'. In other words, the learning rate says how much the model wants to learn every time it finishes an epoch of training.  At the beginning I set the learning rate, that usually is a small positive value, at 0.0005 and then I used the decay rate to decrease at each step the learning rate to prevent the fact that the model starts learning too much about the training samples of the dataset.
Another important parameter is the accuracy metric that evaluates my classification model and that expresses the number of correct predictions on the total number of predictions. Finally, there is the specification of the 'categorical_crossentropy'  loss function which is the loss function used where we're dealing with the classification of two or more classes.

There is now the most important step that is fitting the model by calling the fit function which is used to train my neural network. After training we'll have the loss and accuracy results on the training set and we'll also have the loss and the accuracy parameters about the validation set, which is my test set of images. I trained my model  for 20 epochs.

**Results: MODEL A**
At the end of training the loss and the accuracy values about the training set are very good (loss: 0.071 and acc: 0.978) and from the start of the first epoch to the 20th epoch the loss value is only decreasing and the accuracy value is only increasing. Actually, the results value already after the first epoch were acceptable: that's because my model hasn't started learning the features of my images from skratch, but it has used the knowledge of the pre-trained model MobileNet to recognize the images.

The results about the validation loss and the validation accuracy (val_loss: 0.458, val_acc: 0.907) in general are fine but however quite worse than the values related to the training set. From one side, these results are normal since the fact that usually the loss and accuracy values are worse for the validation set because the model tries to classify images that it has never seen before (contrary to how it happens for the training images). On the other hand, if we plot the results on a graph we'll see that there is a substantial difference between the train and the test outcome and we'll also observe the fact that the development of the orange line (test line) remains quite stopped around a certain value and is not so much improving during the training. That's because I recall that the trainable layers are only the ones of the last layer that I add to my transfer model to have as output the classification of 8 images. On the contrary, all the layers of MobileNet except the last have been 'frozen' and so their weights were already fixed.





To conclude, it seems that modifying only the last layers and freezing all the layers before the last one it's not a great idea, only for the fact that the model does not improve itself so much, but it's like it has already learned everything.
For this reason, I tried to modify something in my previous transfer model by doing 2 main things: adding some additional fully connected layers and increasing the number of trainable layers.

**MODEL B1: first fine-tuned model**
Starting from the model I have trained before, I built a similar model with the differences about the trainable layers and about the structure of the network.
Another thing that I changed is the way of preprocessing images before putting them into the network. In the first model I trained earlier, the images had been preprocessed through preprocessing function of mobilenet (taken from keras), and through data augmentation. For this model, that I'll call 'fine-tuned model' instead of 'transfer learning model', I do not use the preprocessing function of MobileNet but I only rescaled the coloured images from 0-255 to 0-1 and I apply the data augmentation technique also this time.

As I did for the previous trained model, I load from Keras MobileNet with the difference that I cut off not only the last layer of the network, but also the last 20 (trainable) layers of the model. In this way, I 'fine-tuned' the pre-trained model in order to let the new 'fine-tuned' model to work better on the customized dataset. The reason why I'm trying to operate with fine-tuning technique is that the Transfer Learning approach takes a great advantage from the dataset we're dealing with: if the dataset in which the pre-trained model has worked with is very similar to the available dataset, also without fine-tuning the pre-trained model there is the possibility to have good results. On the other hand, if the dataset has nothing to deal with the original dataset, only the TransferLearning approach could be not enough to have a satisfactory outcome. In this early mentioned case, the fine tuning technique is the most recommendable.

In this image classification problem, I only know that the original dataset and the actual dataset (the one I'm working on) are different. I experimented for now only the transfer learning approach by removing from the pre-trained model the final layer and by adding a suitable final one for my specific problem, but I don't know yet if doing the 'fine-tuning' technique could make the situation better or worse. We'll see it with the fine-tuned model I will train.

**MODEL B1: structure**
Let's describe the structure and the depth of this fine-tuned model.
Recalling that in neural networks the first layers detect simpler and more general patterns and the peculiar and more specific patterns are detected only moving forward towards the architecture, I choose to remove from MobileNet pre-trained model the last 4 trainable layers (in particular 2 convolutional layers has been eliminated) and to add a global average pooling layer, and then 4 final dense layer included the last one with the 8 units of classification. Then I created my new fine-tuned model by calling the Model function which instantiated a model which has as input the pre-trained model input according to the shape of my dataset images, and as output the output specified with the 8 units.

**MODEL B1: trainable layers**
Another thing that is changed from the previous transfer learning model is the layers that I'm training during my training session and the ones I leave frozen. While before I have made trainable only the output layers, now I'm also training a convolutional layer (by MobileNet) and the fully connected layers I added at the end of my network.
Now the number of trainable parameters are: 1,133,032.

Then I set the learning rate of my optimizer function equal to 0.0005 and a
decay rate = learning rate/epochs.
I compile the model with Adam optimizer, with the 'categorical crossentropy' loss function (used for multi-classes problems) and I specify the accuracy metric.

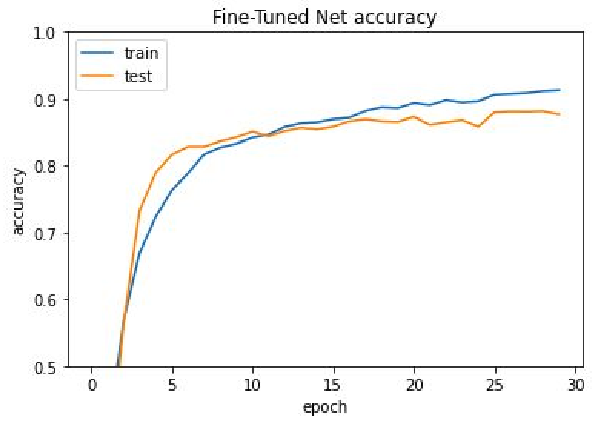
**MODEL B1: results**
First I trained the model for 10 epochs with a learning rate of 0.0005 and a
decay rate = learning rate/50. The results were very good and they seemed to be growing as concerning both the validation accuracy and validation loss. For this reason I tried to decrease the learning rate to 0.0004 and to train the model for 20 epochs. The result is even better for loss, val_loss and val_acc metrics. For the accuracy metric on training samples it is the same as before.  This means that the model continued to train and to improve its performance.

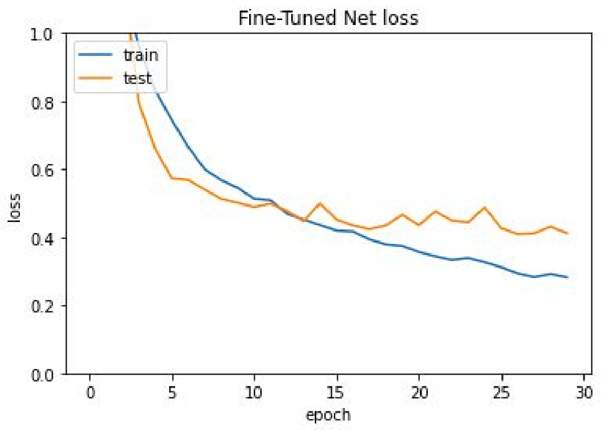
I tried the last time for 30 epochs.
Training the model for 30 epochs did decrease the validation_loss and did not vary the validation_accuracy. The loss and acc metrics are improved both.

Result table:

1) 10 epochs: loss: 0.51, accuracy: 0.84, val_loss: 0.49, val_accuracy: 0.85
2) 20 epochs: loss: 0.42 accuracy: 0.84, val_loss: 0.45, val_accuracy: 0.87
3) 30 epochs: loss: 0.26 accuracy: 0.92, val_loss: 0.41, val_accuracy: 0.88

**MODEL B1: comments**
An interesting thing is that the metrics results about the training set and the ones of the testing set are very similar. And it's possible to note that observing the outcomes of case 1) the loss and val_loss, and the accuracy and cal_accuracy have similar values. It means that the model doesn't recognize better the training images for which it trained for (usually, val and acc metrics results are better than val_loss and val_acc). So one possible reason is that the model hasn't been trained enough times.
Then, after seeing the result of the training with 30 epochs we see that actually the loss metric on the training set is decreased a lot and also the accuracy is higher than before. The value of val_loss and val_accuracy has improved but not so high. That could denote the fact that if the model it's trained for more epochs (so for more time) it may suffer from overfitting. Overfitting is a quite big problem in machine learning. It happens when the model learns the details and the noise in the training data with the result of not being able to recognize the images that belong to the test set because "too general". This leads us to say that the fine-tuned model is no longer trainable and that 30 epochs are sufficient.





**Comparing transfer learning model (MODEL A) and fine-tuned model (MODEL B1)**
Comparing the model A with the model B1, we note that in the first training case the performance did not improve so much during the epochs. That's because we used all the knowledge of the pre-trained model to train and test images belonging to a new dataset. However, the model succeeded to obtain a decent performance and to identify a lot of images of the 8 classes. But this (transfer learning technique just used) is not the best choice for this particular case (dataset not the same as the original).

So I pick up the model and fine-tune it (model B1). The model B1 in addition to having a good part of layers from MobileNet, it also has other new layers and above all has more trainable layers that may be trained on the images of the new suitable dataset.
The results are much better for model B1 which, even if thanks to the help of the pre-trained model (which ha fornito to it the most general features), however had to adapt to the new dataset by updating the weights of the layers of MobileNet and training the new ones just added to the model.

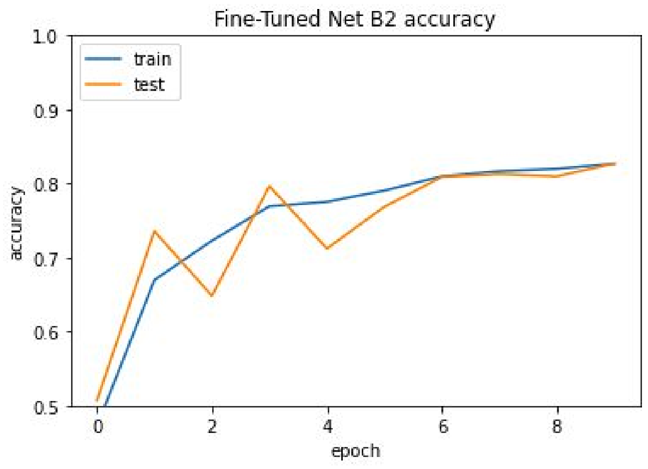
**MODEL B2: second fine-tuned model**
Lastly, I want to make a final variation of the fine-tuned model by adding some convolutional layers. Recall that in the previous fine-tuned model I have only added fully connected layers to the pre-trained model and made trainable more layers than before. Now, I'm going to cut off the last 4 convolutional layers of MobileNet and then in their place attach 2 customized convolutional layers followed by a global average pooling layer which reduces the spatial dimensions of the tensor in input , a flatten layer and then 5 final fully connected layers. The job of the flatten layer is to convert the pooled feature map to a single vector that finally is passed to the fully connected layers. After that the fully connected layers, which are simply feed forward neural networks and in cnn's structure are found in the last few layers, connect every neuron in one layer to every neuron in another layer. The first dense layer starts with a vector of 256 nodes and then the nodes of the vector decrease to 128, then 64 and at the end the last dense layer has the final output vector composed by 8 nodes, which are the 8 categories of the images in the dataset.
After the two convolutional layers I added on the network, I used the Batch Normalization method which is a regularization method used for reducing the overfitting problem. In general, it has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep neural networks.
Regarding the number of parameters that have to be updated during training, I set as 7 the number trainable layers, in particular the 2 convolutional layers that I added to the model and the 5 fully connected layers at the end of the network with a total of 2.927.052 trainable parameters, about ¾ of the total parameters. Remember that in the convolutional layers the number of parameters is independent from the size of the input and depends on the size of the kernel we apply and the number of kernels we apply. On the contrary, in the fully connected layers the number of parameters depends on the size of the input and of the output vector. For example the fully connected layer called "dense_11" has the number of parameters that is equal to 300(input)x256(output)+256. Note that the number of parameters of the convolutional neural networks are many more respects to the fully connected layers ones, that's because there are a lot of convolutional layers and because the number of kernels applied on each convolutional layer is very high.
After compiling the model with Adam optimizer with the learning and the decay rate as usual, the categorical_crossentropy loss function, and after specifying the accuracy metric I fit the fine-tuned model for 20 epochs.

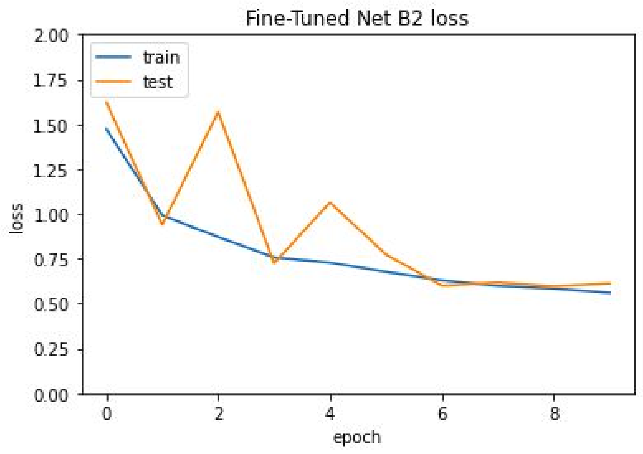
**MODEL B2: results**
After 20 epochs of training the results are not so good if compared with the previous fine-tuned model. The loss and the accuracy about the training set are fine because loss is about 0.39 and the accuracy about 0.88. But the thing is that this model can suffer from overfitting, maybe because it is trained for so many epochs. That's quite evident when we

see the values of the validation loss and validation accuracy. The first is 0.76 and the second is 0.81 which are values that are relatively distant from the loss and the accuracy of the training images. Also, plotting the history of all the training epochs we note that the test metrics (both loss and accuracy) are not increasing in a linear way, but go up and down continuously and at the end of the 20th epoch it's unclear whether they will stop soon or not. However, for example, the difference of the validation loss between the first epoch and the last is only about 0.2 because after the 1st epoch the val_loss is 0.99 and at the end of the 20th is 0.76 and during the training the value fluctuates. So I decided to train the model for only 10 epochs instead of 20 because I thought that after 10 epochs the model would have learned enough to recognize the dataset images, giving the fact that the model was already taken part of the pre-trained knowledge.
Training for 10 epochs the fine-tuned model gives better results. The loss and the validation loss metrics are closer than before and the validation accuracy is improved from 0.76 to 0.50. The validation accuracy is improved at 0.84.
So I think that is enough to train this model for 10 epochs instead of 20 because it gives more desirable results.

Results tables:

4) 10 epochs: loss: 0.39, accuracy: 0.88, val_loss: 0.78, val_accuracy: 0.81
5) 20 epochs: loss: 0.36 accuracy: 0.89, val_loss: 0.51, val_accuracy: 0.85

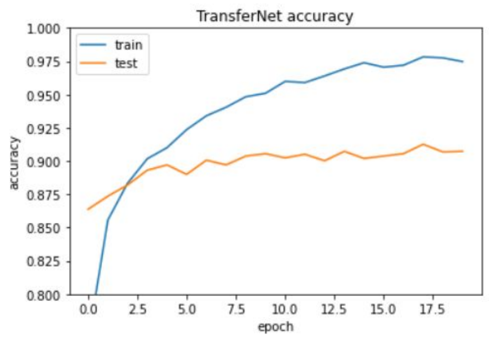


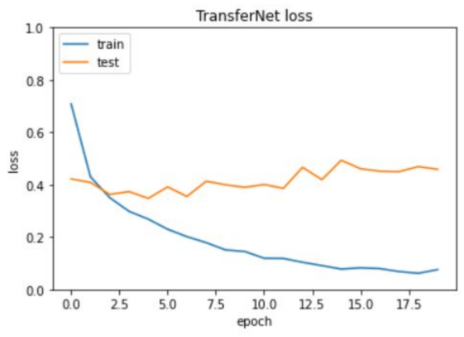


**Computational time: considerations**
Now I'm going to make a few considerations on the computational training time. Usually, the time a convolutional neural network takes for training is high. For this reason, it's extremely important and necessary to use a GPU instead of a CPU in the processing. That's because parallel computing is recommended even when processing a simple neural network, so for convolutional neural networks, which in addition dealt with images, GPU is very meaningful. In my study case, the use of a transfer learning approach has a big impact on the computational training time because instead of training a CNN from zero I use a pre-built and pre-trained model whose weights were already fixed. In particular, the MODEL A, that is the model in which I have frozen a very big part of parameters, the computational time for training was about half a minute/a minute for epoch.ANd, As I have early said, that's because in the model that I trained and evaluated the parameters that have been updated

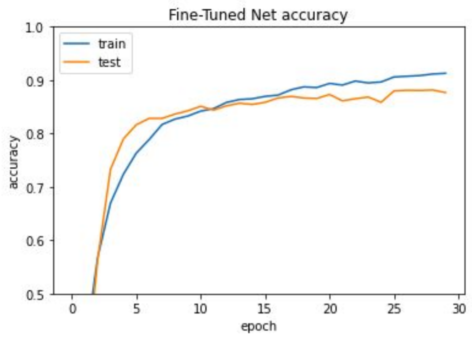
were not so many compared to the total number of parameters. Comparing the three models (MODEL A, MODEL B1 and MODEL B2) I expected that the model that takes more training time is MODEL B2, because it is the most “fine-tuned” model.
Another thing that has an impact on computational time is the data augmentation. I tried to train a model without using the data augmentation technique and I noticed that the time for completion of 10 epochs was reduced. However, data augmentation techniques can be very useful for example when the dataset is not so big and it helps to reduce overfitting. Furthermore, in general, a model that is trained with the application of distortions on images can be more prepared than another that sees all images just as they are without malformation.

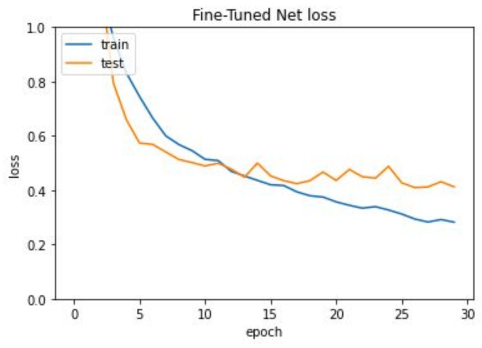
**Conclusions:**
I want to compare the models’ plots about the metrics to see the differences.
**MODEL A: transfer model (MobileNet)**



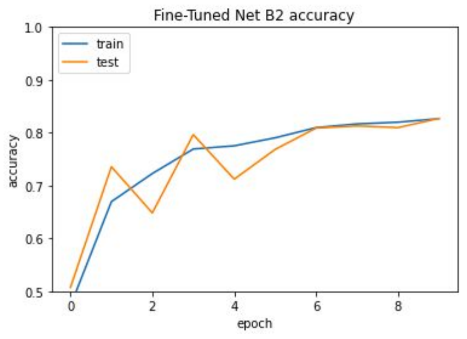


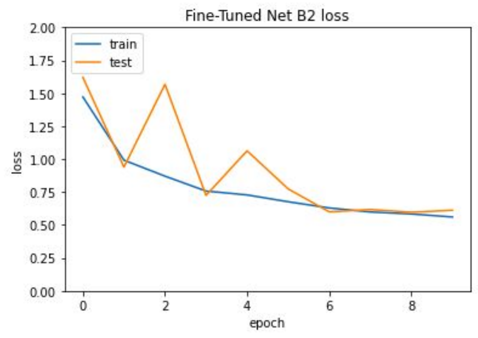
**MODEL B1: first fine-tuned model (little retouching)**





**MODEL B2: second fine-tuned model (big retouching)**





Final considerations:

- MODEL B1 is the most stable model. It is somewhere between the transfer model and the “strong” fine-tuned model.
- MODEL A is not so much improving its performance during training because it used all the knowledge of MobileNet model, which was not trained with the same dataset.
- MODEL B2 performance goes up and down and it might have been fine-tuned a bit too much with the additional convolutional layers that did not improve the situation with respect to the performance of the MODEL B1.