Convolutional Neural Networks

Giovanni Garifo

2019/01/14

1 Introduction

The CIFAR100 dataset is composed of 100 classes containing 600 images each. The images have a resolution of 32x32 pixels and are in RGB format. Of the 600 images available for a given class, 500 are part of the training set, and 100 of the test set.

We want to train and test different kinds of neural networks (hereinafter NN), in order to evaluate their accuracy.

We are using *Pytorch* to implement our NN. For each kind of NN, a python class has been written. Each class is composed of two methods: an initialization method, that instantiate the NN, defining its architecture, and a $forward$ method, that implements the forward pass of the NN. The *backward* pass is implemented by Pytorch itself, thanks to its computational graph, which allows to perform efficiently the backpropagation phase by applying the chain rule for the computation of the gradient.

For each kind of NN we are going to take a look at some plots in order to have some insights about their performances. The networks will be trained and tested over 20 epochs.

2 Fully Connected Neural Network

In this kind of network every neuron receives as input the output produced by all the neurons of the previous layer, except for the input layer, which naturally doesn't have any input other then the input values themselfs.

The following code illustrates the architecture of our first neural network. As we can see there are three fully connected layer. The first takes as input the features, being a fully connected layer, each neuron of this layer (there are 4096 neurons, the same as the output features of the first layer) takes as input every feature, and produces an output that is received as input from all the neurons of the second fully connected layer. The same concept applies to the output layer, this final layer has a number of neurons that is equal to the number of classes.

```
def __init__(self):
    super(old_nn, self).__init__()

    # First fully connected layer, perform a linear (z=w*x+b) transformation
    self.fc1 = nn.Linear(32*32*3, # number of input features
                         4096) # number of output features

    # from first hidden layer to second hidden layer
    self.fc2 = nn.Linear(4096, 4096)

    # from second hidden layer to output layer
    self.fc3 = nn.Linear(4096, NUM_CLASSES)
```

As we can see from the code below, the forward pass of our first Neural Network is very straightforward: it's just a concatenation of fully connected layers and activation functions that are used to introduce non linearity in our model: in this case we are using a *sigmoid*.

```
def forward(self, x):
    x = x.view(x.shape[0], -1)
    x = F.sigmoid(self.fc1(x))
    x = F.sigmoid(self.fc2(x))
    x = self.fc3(x)
    return x
```

If we train and test our network over 20 epochs, and if we plot the loss and the accuracy for each epoch, we see that the accuracy of our model is slowly increasing, thus performing badly. This because we are using an old style NN, which is not able to extract the necessary knowledge from the feature. We'll see that more advanced models will perform much better. As we can see from the first plot, the loss is slowly decreasing, and it can apparently continue to decrease if we use more epochs, but this won't directly translate into a better accuracy, because the NN will overfit on the training data.

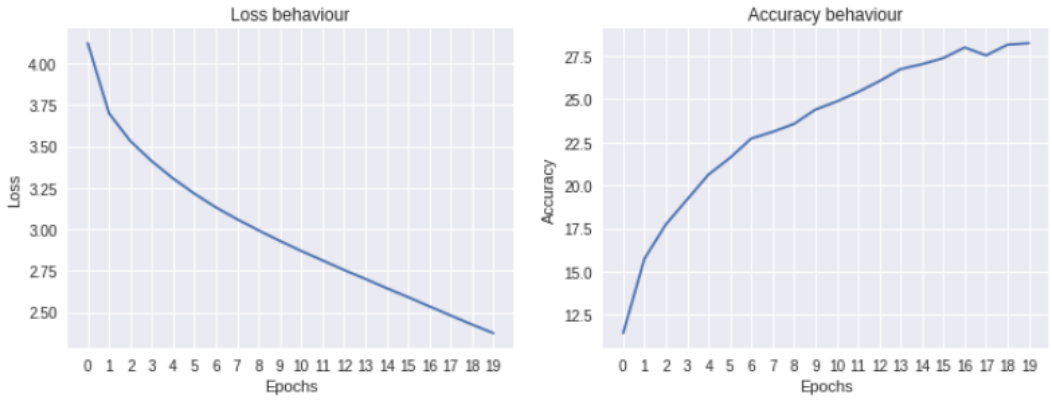

Figure 1: Loss and Accuracy over epochs for the Fully Connected NN

3 Convolutional Neural Network

The convolutional neural networks (hereinafter CNN) make use of the so called *convolutional layers*. A convolutional layer is composed of neurons that apply a convolution transformation on the input features: this kind of transformation allows the model to make a better generalization of the input features, in order to classify correctly the samples.

We'll test different convolutional architectures: a first set of architectures are based on a pure CNN, with no other layers in addition to the convolutional ones. A second set does use more advanced techniques such as batch normalization and droput to improve the generalization of the model, thus to reduce overfitting. Then we'll use a state of the art pre-trained CNN, called $ResNet18$, to see how well a pre-trained network with only few minor adjustements performs.

3.1 Standard CNN

Below there's a standard implementation of a convolutional architecure: there are a sequence of convolutional layers and pooling layers: the latter are used to reduce the input dimensionality, in this case we're using a max pooling techniques, that takes only the maximum value for each group of features, the dimension of the group depends on some filtering parameters (kernel size, stride, padding).

```
def __init__(self):
    super(CNN_1234, self).__init__()

    self.conv1 = nn.Conv2d(3, 32, kernel_size=5, stride=2, padding=0)
    self.conv2 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=0)
    self.conv3 = nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=0)
    self.conv_final = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=0)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
    self.fc1 = nn.Linear(4*4*64, 4096)
    self.fc2 = nn.Linear(4096, NUM_CLASSES) #last FC for classification
```

The forward method is very similar to the fully connected NN one, the only main difference to notice is that for the convolutional network is not used a sigmoid but a *ReLU* activation function: it's main advantage is that it doesn't saturate to 1 the output value, it's disadvantage is that it saturate to zero a negative value.

Here are the plots for the behaviour of the accuracy and the loss across the different training and testing epochs. The different kind of convolutional filter used is writted in the caption.

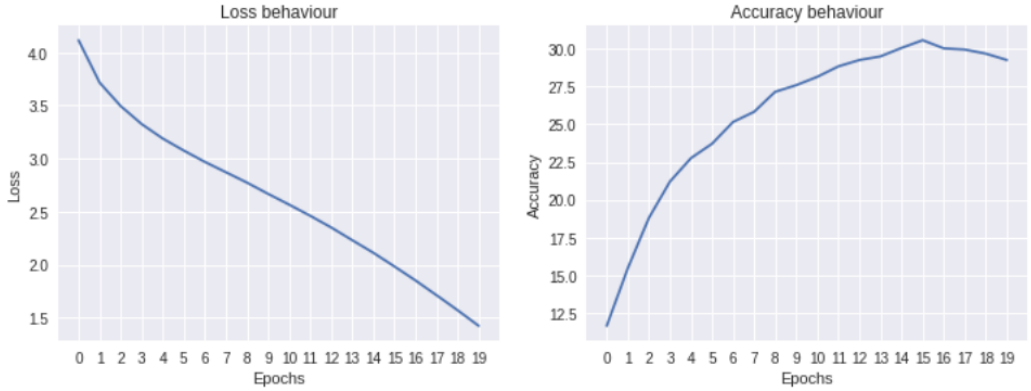

Figure 1: CNN_1 with convolutional filters for each layer: 32/32/32/34

For CNN_1 we are not getting a much better result with respect to the NN, we can also notice that from epoch 15 our accuracy is starting to decrease, this can be a signal of overfitting of the model on the training samples. Now we'll try to increase the number of filters in order to see if this improves the accuracy of the model. Please note that by doing this, we're also increasing the training time.

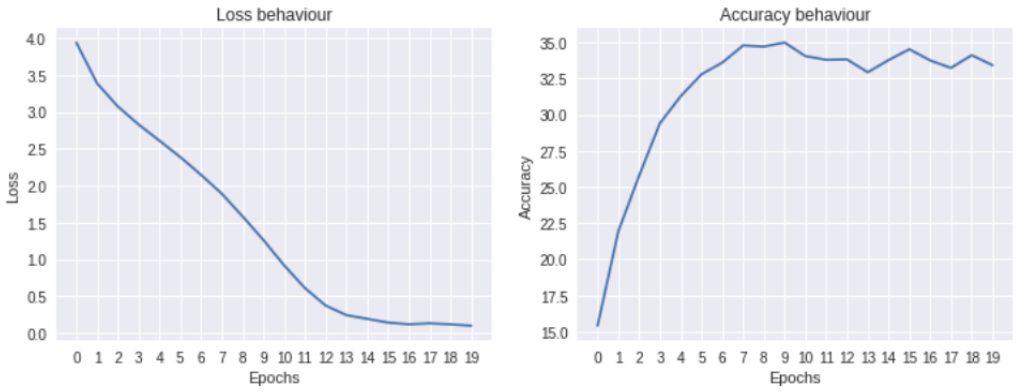

Figure 2: CNN_2 with convolutional filters for each layer: 128/128/128/256

For CNN_2 the accuracy have increased by five percent, but the training time almost doubles. We can also see that the accuracy doesn't increase anymore after the sixth epoch, and it also starts to spikes

up and down: another signal of overfitting. This is the first time that the model is able to reach the minimum loss, as it can be seen from the first plot.

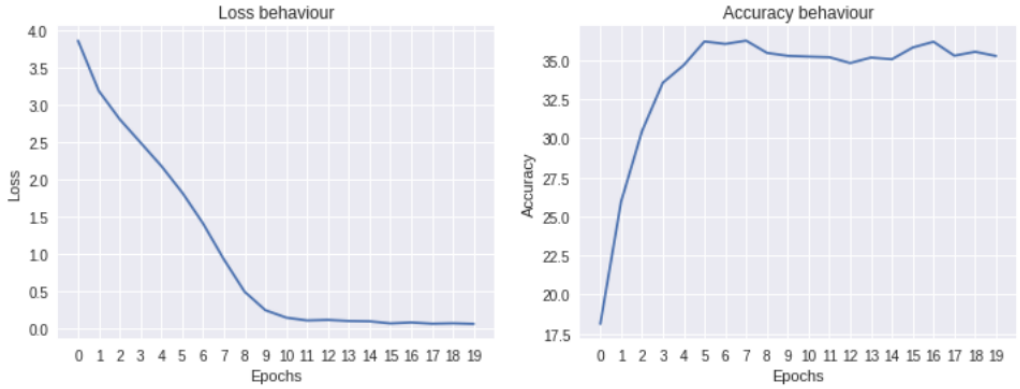

Figure 3: CNN_3 with convolutional filters for each layer: 256/256/256/512

CNN_3 performs practically the same as CNN_2, but with much worse training time. We can see that the loss decrease slighly faster, thus that the model reaches it's maximum accuracy after just 5 epochs.

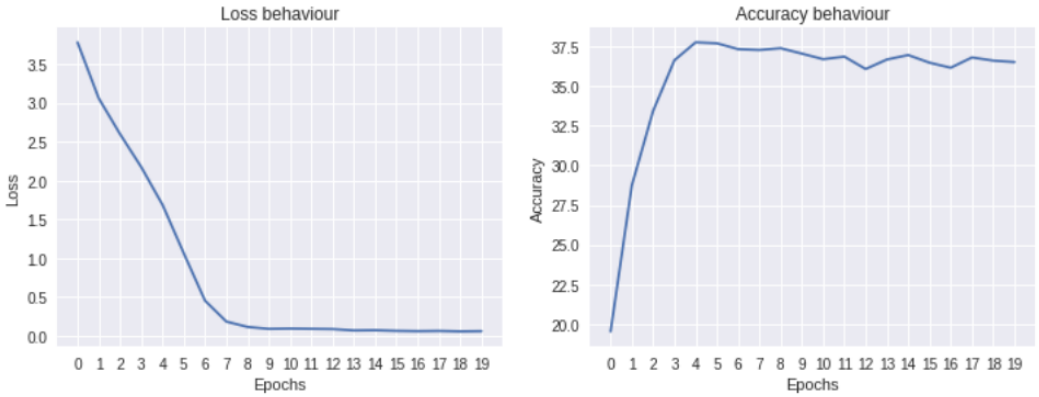

Figure 4: CNN_4 with convolutional filters for each layer: 512/512/512/1024

With CNN_4 we're actually increasing the accuracy by some percentage value, but the training time increased much more in comparison to the accuracy gain obtained: in the next chapter we'll see if there are better ways to improve the accuracy, without sacrifice too much the training time. Also for this CNN, the time to reach the minimum loss is slightly lower: the CNN is fast to extract features from the training data, but it's unable to generalize well.

A possible interpretation of the results for this set of CNN is that given the fact that on the first layers the convolutions are extracting very general features (edges, gradients...), we are actually extracting too much information from the images: our model is in practice unable to recognize anything because it recognize too much. At each convolutional layer, the filters are learned from the input data. if the filters are too much, later on during the testing, the model may not perform well, because it won't be able to recognize a singular pattern in the image. We are also prone to overfitting: the network is learning too much from the training data, this is why the accuracy is slowly decreasing over time. The techniques that we'll use for the second set of CNN are purposely developed to decrease the chance of overfitting.

3.2 Advanced techniques for CNN: batch normalization, dropout and data augmentation

Now we'll apply some of the common techniques that can be used to lower the overfitting of a network. In particular, we'll use batch normalization, dropout and data augmentation. All this techniques will be applied to the CNN_2, the CNN with 128/128/128/256 convolutional filters, that is the one with the best trade-off between accuracy and execution time.

Batch normalization is a normalization techniques that normalize the output of each convolutional layer of the network, it's implemented by adding the so called normalization layers to the network. This layers allows to decrease the probability to have some neurons that always receives bigger values compared to the other neurons of the same layer, in fact, the batch normalization layer actually standardizes the output feature of a convolutional layer.

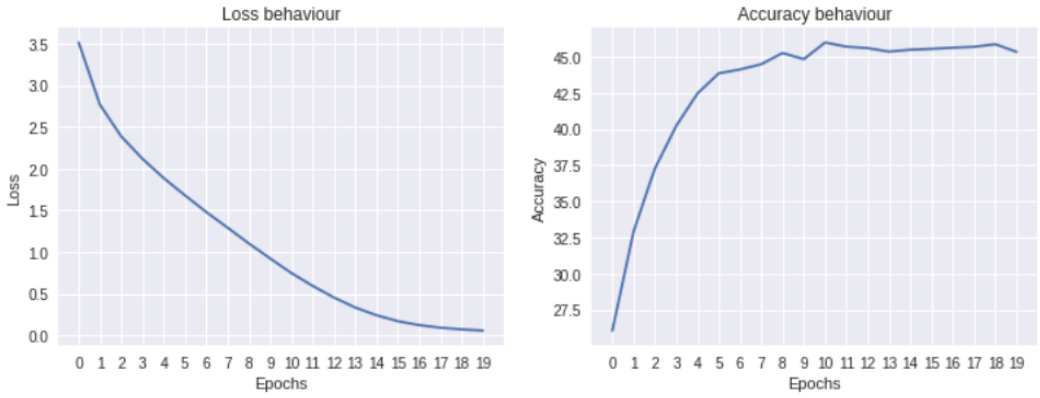

Figure 5: CNN_5 uses only batch normalization

As we can see for CNN_5, by using the batch normalization, not only the accuracy of the network have increased by more then 10%, but also the overfitting of the network has decreased as well: we no longer see a decrease in accuracy over time. Our model is still unable to generalize well, given that the accuracy is not able to pass the threshold of 50%.

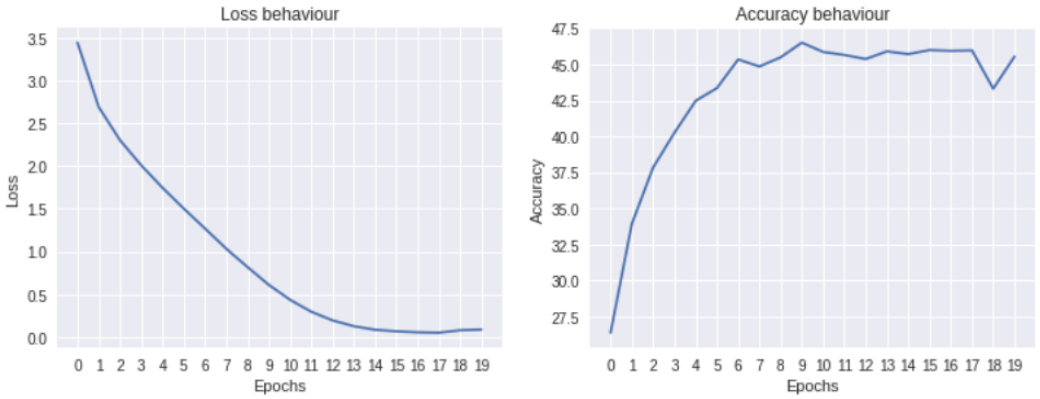

Figure 6: CNN_6 uses only batch normalization, but with a wider FC layer

In CNN_6 we're only substituting the second-last FC layer with a more wider one, composed of 8192 neurons instead of 4096. We are not gaining any accuracy by making this change, also the loss behaviour is the same as the previous CNN.

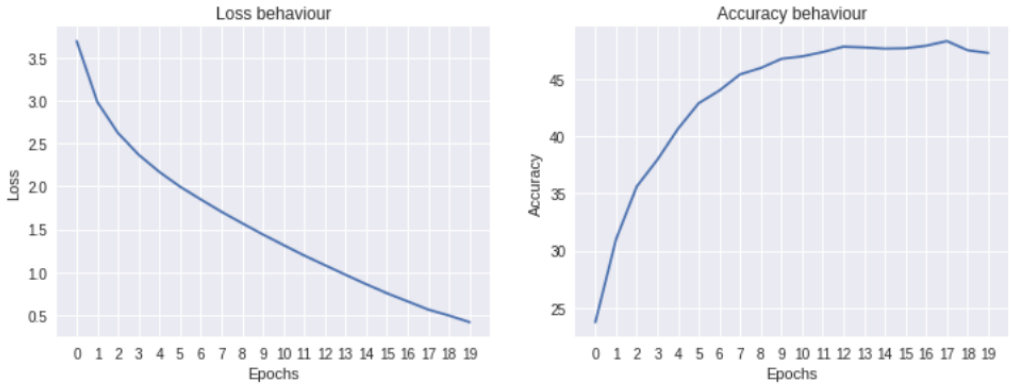

Figure 7: CNN_7 uses batch normalization with dropout on second-last FC layer

With CNN_7, we are using the dropout regularization technique: we are using a dropout parameter of 0.5, so there's a 50% probability that a neuron is shutted down on the last second-last fully connected layer, this will help in reducing overfitting and co-adaptations of the neurons of the layer. We can see a more smooth accuracy curve, but there is no particular increase in the overral accuracy of the model.

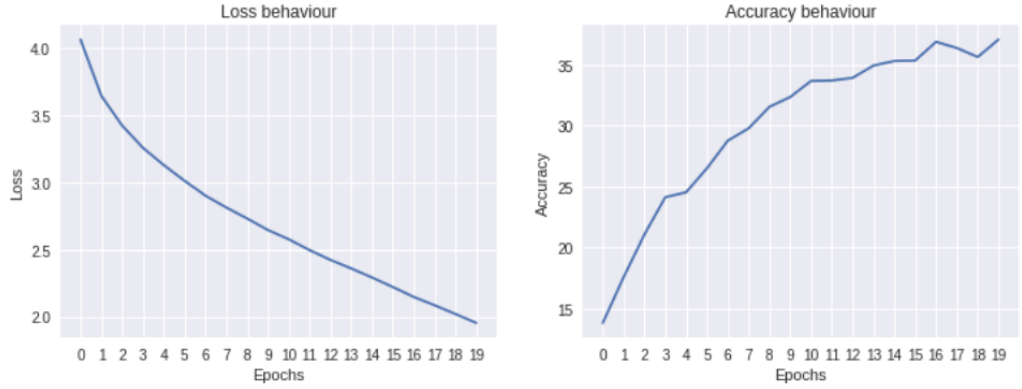

Figure 8: CNN_8 uses only data augmentation techniques

In CNN_8 we are not using anymore any regularization techniques, but only data augmentation: we are doing horizontal flips and random cropping of the images. As we can see, this won't increase so much the accuracy, but is always a good idea to augment the data: this techniques are use to have more variance, and are particularly useful if the dataset is not composed of a lot of samples.

3.3 Pre-trained CNN: ResNet18

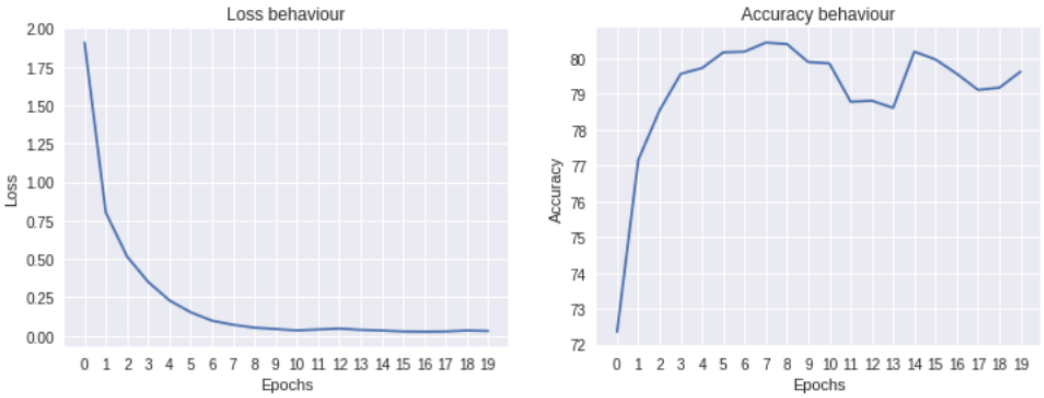

Figure 9: ResNet18 pre-trained with only the last FC layer modified

The last CNN that we used is ResNet18, a state of the art network for image recognition. We are using a pre-trained network, this means that it has already been trained on other sample data. We just need to fine-tune the network on our dataset, in order to tweak the weights and improve as much as possible the accuracy. We used some data augmentation techniques as for the previous CNN, in particular the random horizontal flip, because the random crop didn't bring any accuracy increase.

We can see that this network is the one that gives the highest accuracy, in fact, if our task is to build a classifier for common problems, such as image recognition tools, is always a good choice to start from a pre-trained state of the art network instead of writing a new model from scratch. Even if there are some spikes in the accuracy, they're in the order of 1%, so completely normal. As it's possible to see from the first plot, the loss is minimized after just 8 epochs: they're enough to train the model.