ML/DL HW

Giorgio Crepaldi 242108

January 2019

Chapter 1

HW 1

1.1 Principal Component Visualization

1.1.1 Reprojection of a single image

After loading the whole dataset of images I computed the set of principal components based on it by using the PCA class. I changed the number of PC I wanted to use by switching the argument x in the PCA(x) function with 2/6/60 values as it computes the whole set ranked by eigenvalues and picks the first x PCs.
The last 6 PCs case was a little bit trickier as it required to compute and store the whole set of ranked PCs, take the last 6 PCs from the components_ field in the PCA class and substitute the components_ value with the new one. This guarantees that we only use the last 6 PCs when executing the PCA inverse_transform method.
I reconstructed the 10th image from the dog subset(figure 1.1).
As the number of PC decreases we start to see a face instead of a dog, that's because the dataset has a huge number of face images compared to the ones containing a dog.

1.1.2 Scatter plot using different PC couples

In order to get the PC couples (1,2) (3,4) (10,11) I used the PCA() function again for getting the whole PC set from the components_ field and then I just picked the elements I needed from such array.
Plotting those 3 different scenarios(figure 1.1.2) reveals a good separation in the first case, especially if we take a look at blue and purple points which are the most present. In the other two cases the points seem to be more and more messed up together. This is because of the ranking per eigenvalues done on the PCs.

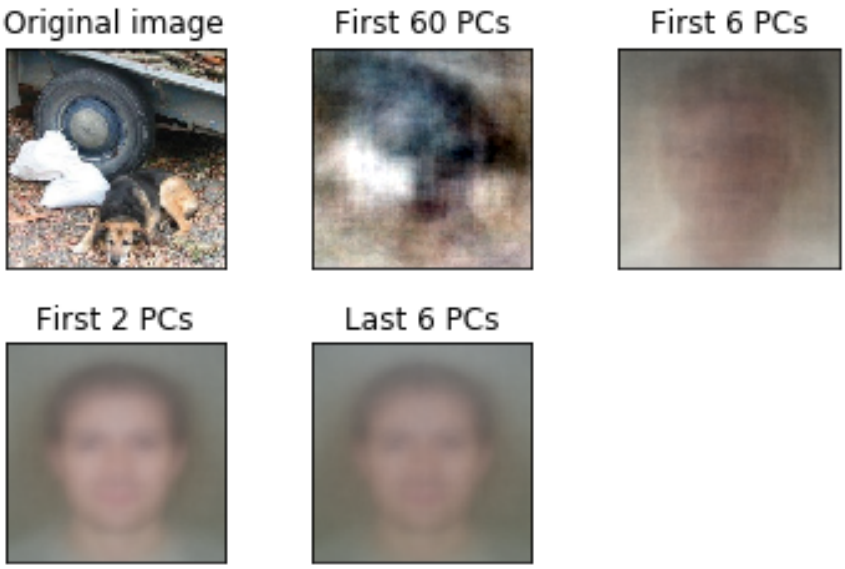

Figure 1.1: Reconstructed images

1.1.3 Deciding number of PCs

The best way to decide how many PCs to use to preserve data is to plot the variance in function of the number of PCs(figure 1.5), choosing a value that uses less PCs as possible while keeping a good accuracy.
In this case i would say that a value of approximately 100 PCs is good as it guarantees a variance higher than 80%.

1.1.4 Bayesian Classifier results

The accuracy values I obtain using Bayesian Classifier are:
Accuracy = 0.724770642202 for the full set of PC
Accuracy = 0.51376146789 for PC(1,2)
Accuracy = 0.440366972477 for PC(3,4)
We notice that a small number of PC(2) is enough to obtain a decent accuracy, contrary to the previous method. That's because the Bayesian classifier works on probability and works fine even with such a low amount of PC.

1.1.5 Optional: plot decision boundaries

See fig. 1.6 for decision boundaries plot.

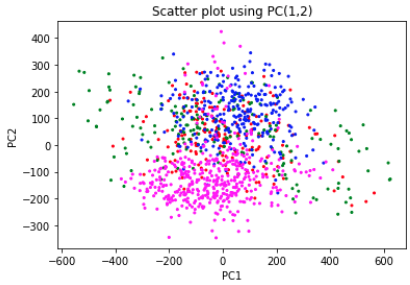

Figure 1.2: Scatter plot using PC(1,2)

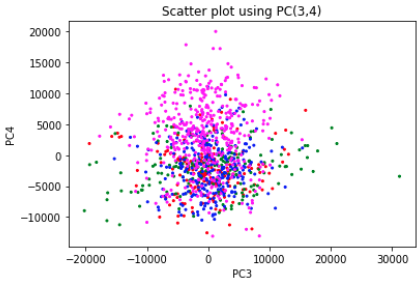

Figure 1.3: Scatter plot using PC(3,4)

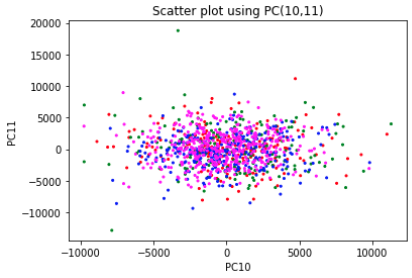

Figure 1.4: Scatter plot using PC(10,11)

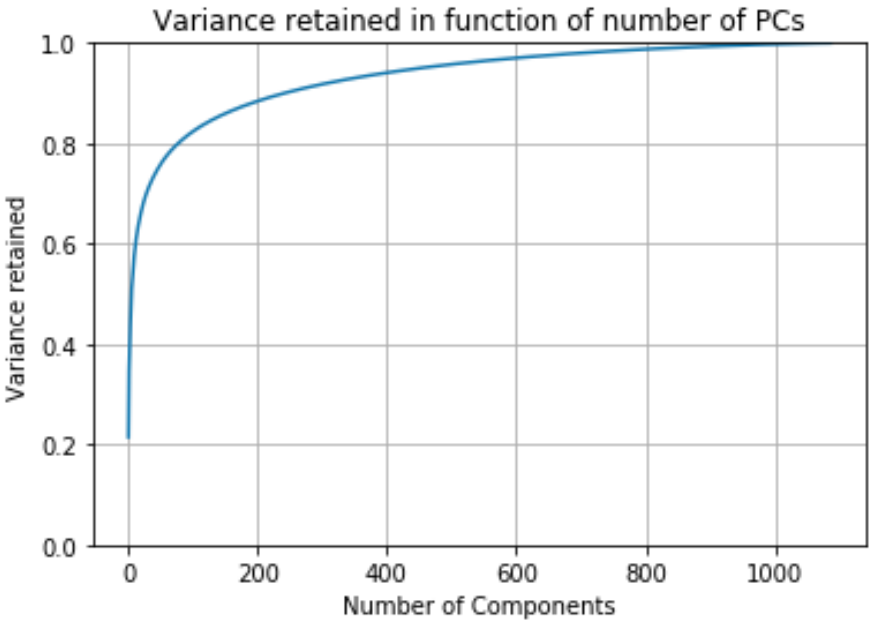

Figure 1.5: Variance plot

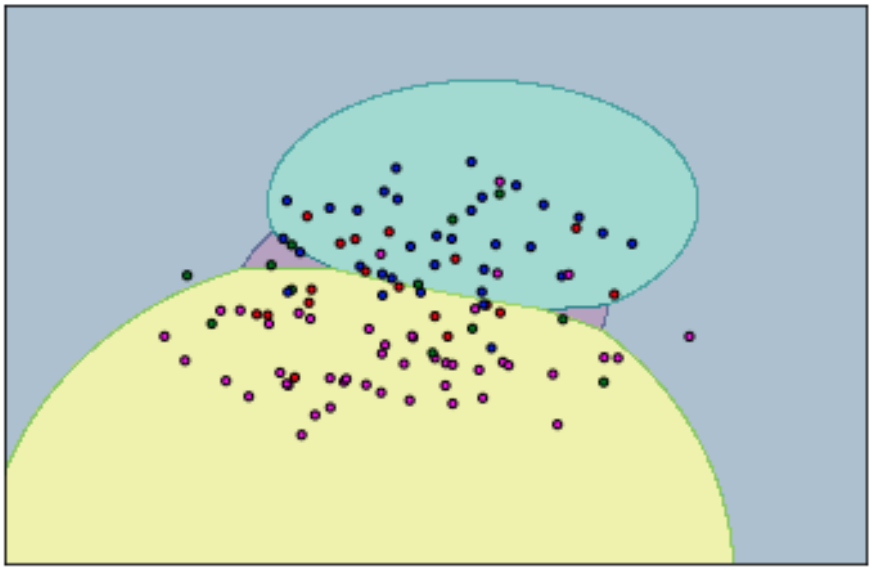

Figure 1.6: Classifier decision boundaries

Chapter 2

HW 2

2.1 Linear

2.1.1 Boundary change on validation set

I did a plot of the decision boundaries using C values ranging from 10^{-3} to 10^{3}. C is inversely proportional to λ, which is the margin.
It's visible that increasing C causes a better classification, but we don't want to increase it too much in order to have a bigger margin which is important for testing. See fig. 2.1 for different boundaries using different C for both linear and rbf svm.

2.1.2 Best C on test set

By plotting how the accuracy varies in function of C(fig. 2.2) we can see that the best C is 0.1 corresponding to 0.83 accuracy which is good. A low value of C means we are using a greater margin, which seems a good choice because green and orange labeled objects are very messed up together. See fig. 2.2 for boundaries using linear svm with best C.

2.2 RBF

2.2.1 Difference with linear svm

What is different from linear svm is that the boundaries are not linear any more as they depend on 2 parameters rather than just one. In this case we see that as C increases (keeping gamma as default) the boundaries get more confusing and less accurate. That's why the best C is very low: C=0.001.

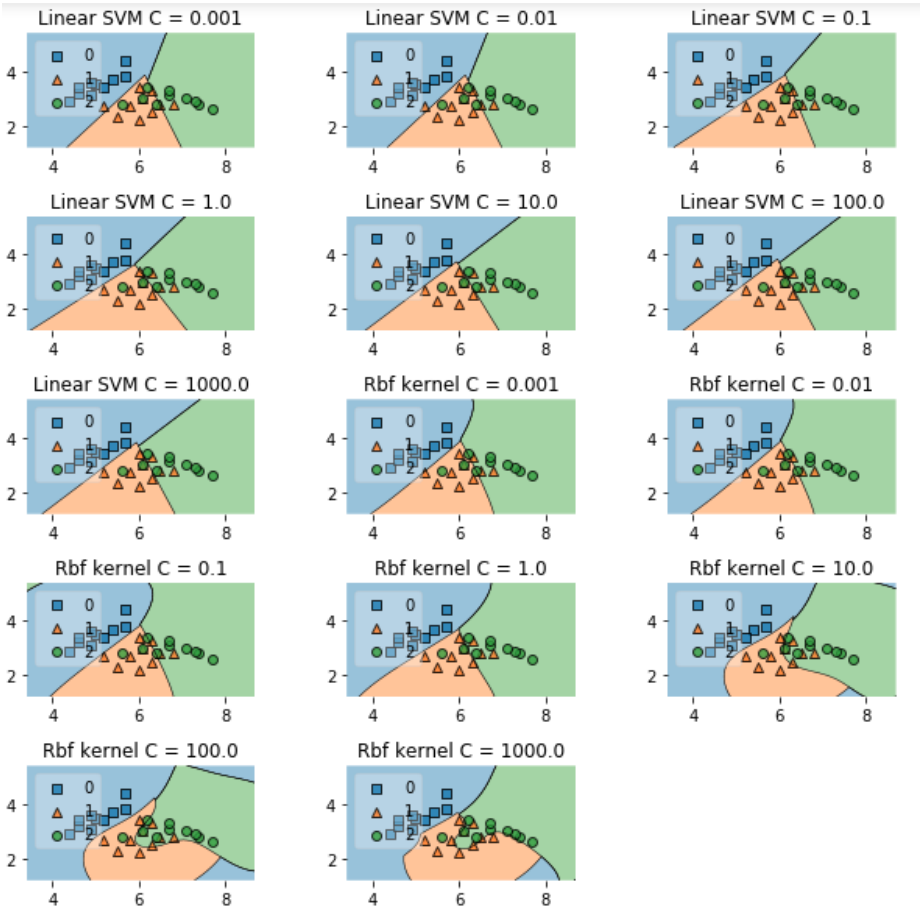

Figure 2.1: Linear and RBF boundaries plot

2.2.2 RBF svm: Grid Search

The grid search is used to obtain the best couple of C and gamma values after computing a matrix containing the score for every combination of the two. In this phase we use the validation set to obtained the best configuration to be used on the test set.
See fig. 2.4 for grid search heatmap.
See fig. 2.5 for the boundaries obtained using such parameters on the test set.

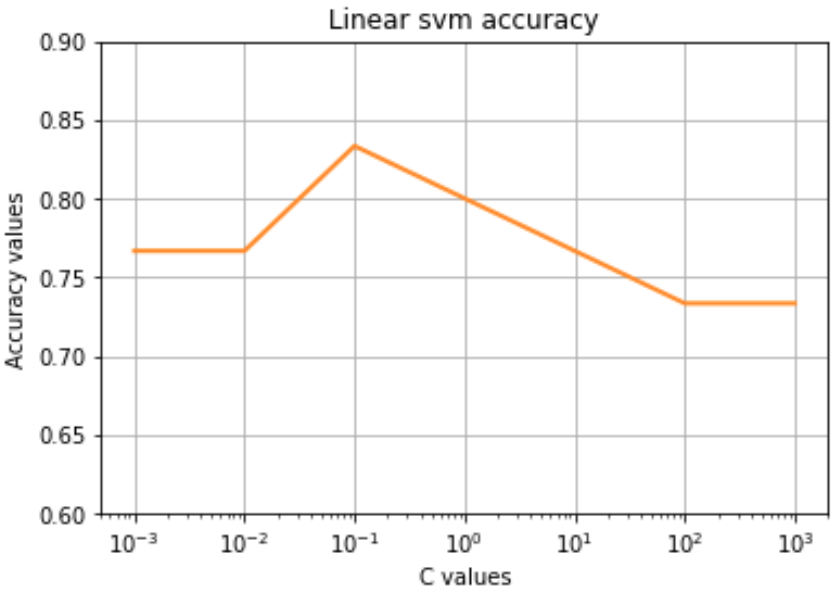

Figure 2.2: Linear accuracy plot

2.3 K-Fold

2.3.1 Test set final score

Using a k-fold cross validation we obtain an accuracy value of 0.81 which is a little bit lower than previous cases. This may be due to the fact that k-fold computes an average among all scores, ending with a more reliable value but not necessarily a better one. See fig. 2.6 for results.

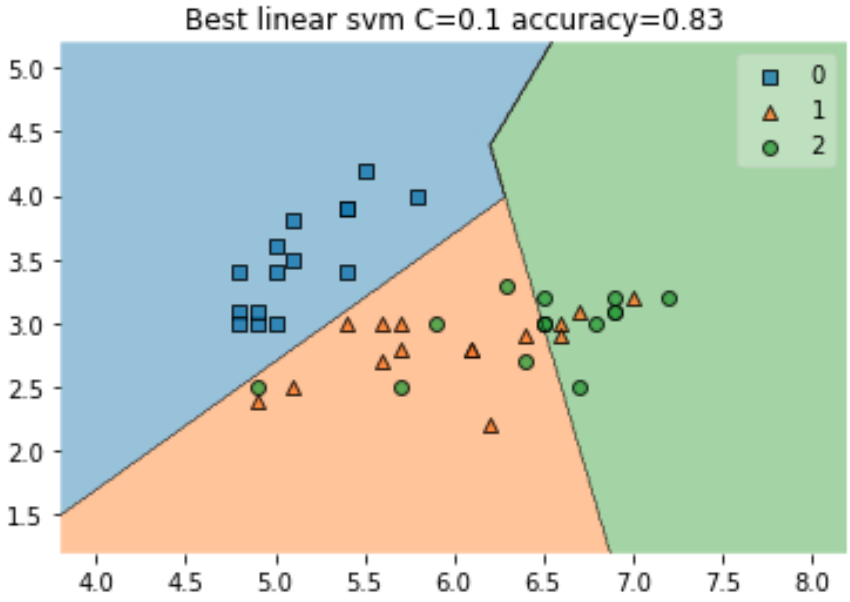

Figure 2.3: Linear svm best C boundaries

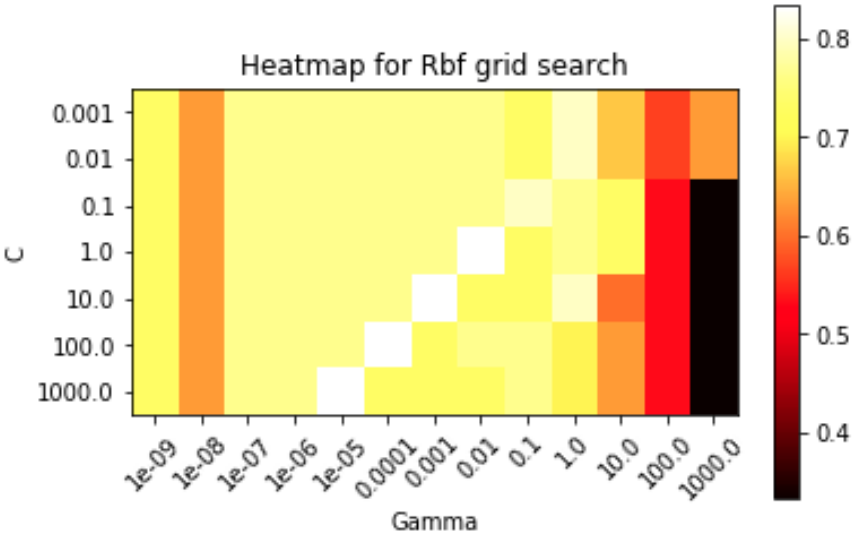

Figure 2.4: Heatmap for rbf svm grid search

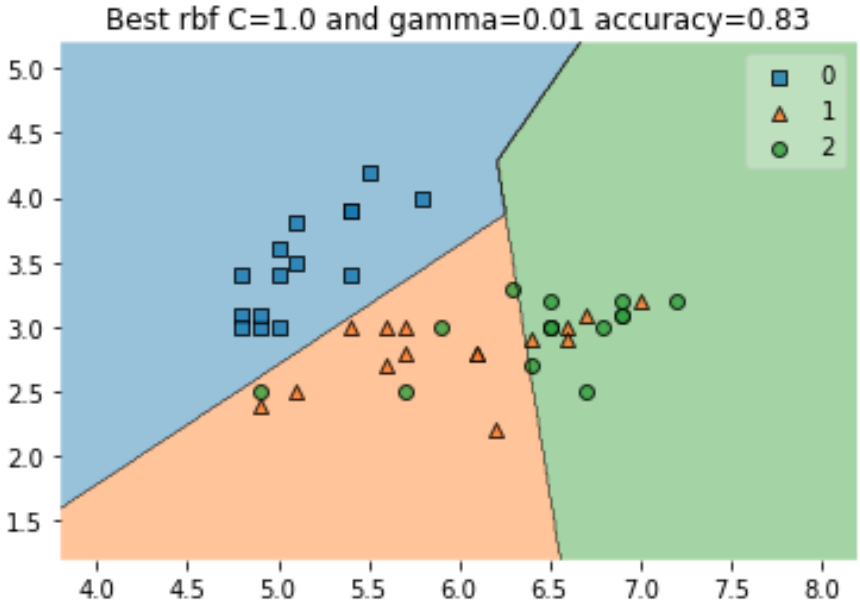

Figure 2.5: Rbf svm with grid search boundaries

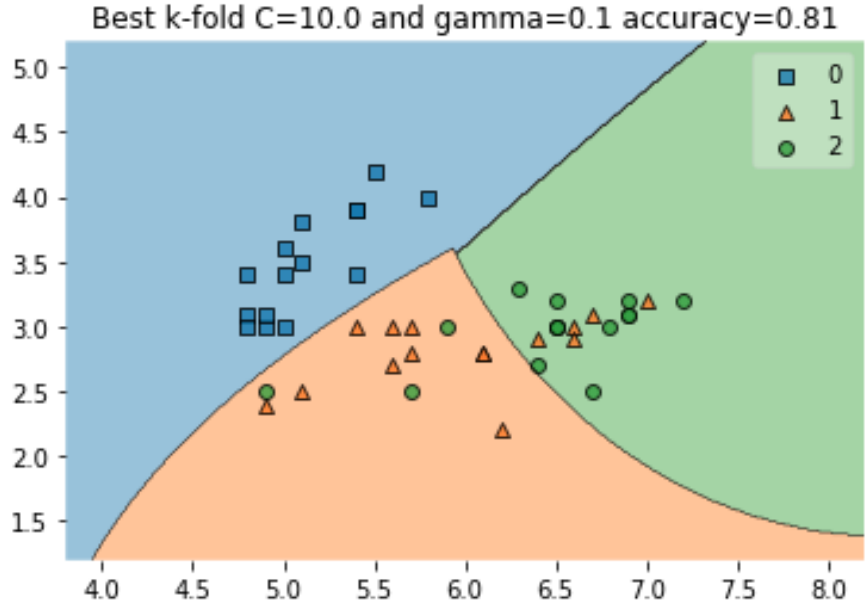

Figure 2.6: K-fold boundaries

Chapter 3

HW 3

3.1 1/6: Old nn

Loss = 2.678
Accuracy = 26%
Computation time = 3m 59s
The final accuracy we obtain using such a simple nn is very low as we may expect using such a simple architecture. See fig. 3.1 for loss and accuracy plot. You can see the code for plotting loss and accuracy below:

```
# store every loss in array
loss_values.append(running_loss / n_loss_print)
  .....
epoch_values = np.arange(0, n_epochs, step = 1)

plt.title(mauvemauve'mauvemauveLossmauvemauve mauve
    mauvemauvefunctionmauvemauvemauve'mauve)
plt.plot(loss_values, marker = mauve'mauveomauve', color = mauve'
    mauveredmauvemauve'mauve)
plt.xlabel(mauvemauve'mauvemauveEpochmauve')
plt.ylabel(mauvemauve'mauvemauveLossmauvemauve'mauve)
plt.xticks(epoch_values)

plt.show()

plt.title(mauvemauve'mauvemauveAccuracymauve mauvefunctionmauve')
plt.plot(accuracy_values, marker = mauve'mauveomauve', color = mauve'
    mauvebluemauvemauve'mauve)
plt.xlabel(mauvemauve'mauvemauveEpochmauve')
plt.ylabel(mauvemauve'mauvemauveAccuracymauve')
plt.xticks(epoch_values)
plt.show()
```

And the old_nn class as requested:

```
class old_nn(nn.Module):
    def __init__(self):
        super(old_nn, self).__init__()
        self.fc1 = nn.Linear(32*32*3, 4096)
        self.fc2 = nn.Linear(4096, 4096)
        self.fc3 = nn.Linear(4096, n_classes) #last FC for classification

    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
```

3.2 2/6: CNN

Loss = 2.182
Accuracy = 28%
Computation time = 3m 44s
For the second part I just used the CNN net instead of the old_nn. In this case we notice a small increase in the accuracy and a faster reach of the max accuracy: 15 epochs instead of 20. I expected a better result as the CNN has hidden layers which extract the image features.

3.3 3/6: CNN using different filters

3.3.1 128/128/128/256

Loss = 0.146
Accuracy = 32%
Computation time = 6m 48s
In this case the final accuracy is 32% but the max accuracy of 33% has been reached at the 10th epoch, which is even faster than the previous case. Another thing to be noticed is the massive gain in training loss just by increasing filter dimension. The computation time gets(and will still get in the next cases) longer and longer as there is a greater number of values to be computed.

3.3.2 256/256/256/512

Loss = 0.075
Accuracy = 34%
Computation time = 15m 37s

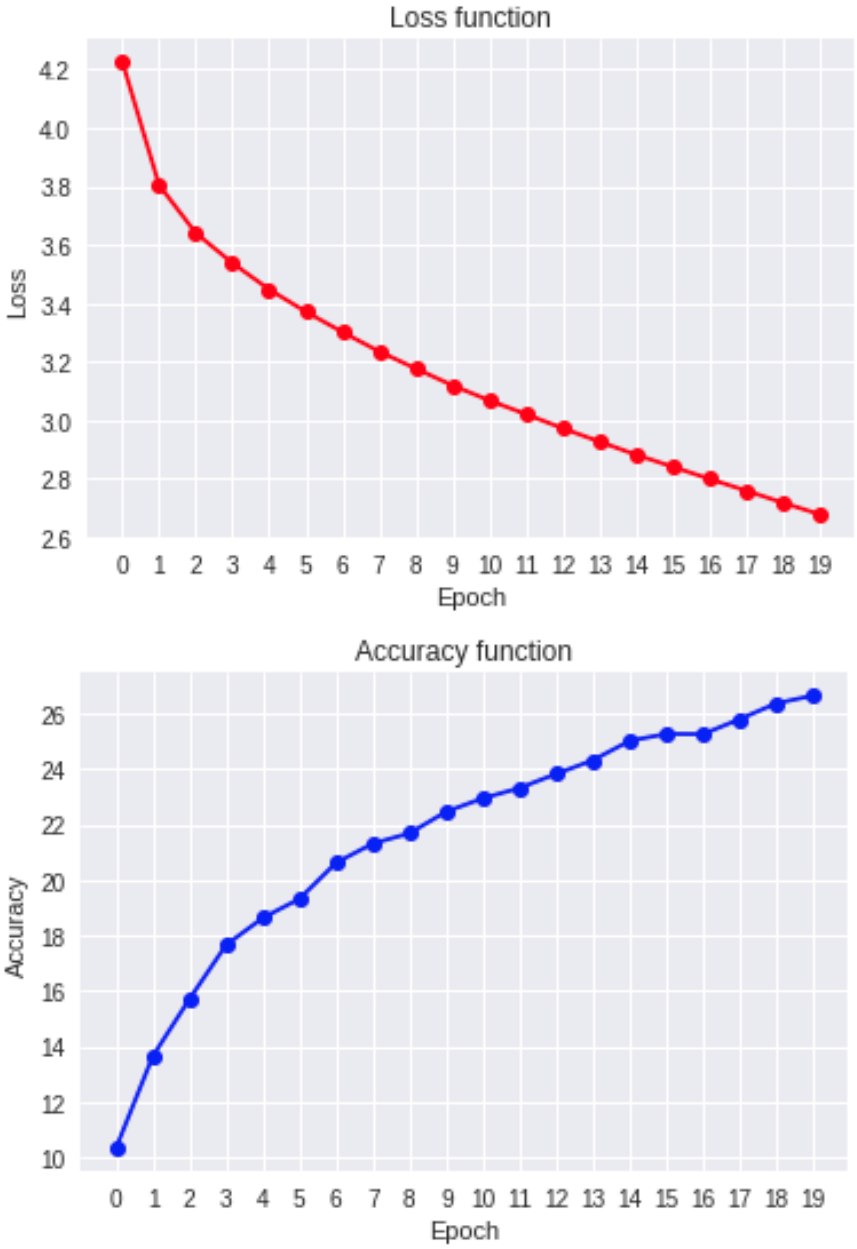

Figure 3.1: Old nn: loss and accuracy

In this case the final accuracy is 34% but it oscillates between 34% and 35% from the 7th epoch. We also notice that the loss function is already stabilized at the 13th epoch thus revealing a maybe unnecessary number of epochs.

3.3.3 512/512/512/1024

Loss = 0.070
Accuracy = 35%
Computation time = 47m 54s
In this case the final accuracy is 35%, oscillating between 35% and 37% yet at the 6th epoch when it reaches the max value. We can notice a loss behaviour similar to the previous case, featured by a continuous value from the 9th epoch already. The computation time is now 47 minutes and 54 seconds, which is a huge increase compared to the no-gain in accuracy. As the loss doesn't decrease from the 9th epoch and the accuracy seems to start decreasing after oscillating for the majority of the epochs it seems we are even beyond overfitting. There's no point in going on if there's no improvement not only for the accuracy but for the loss too.

3.3.4 Final assumptions on filter dimension

As we learn from the results increasing the filter dimension has the main benefit of fastening the reach of the max accuracy process in terms of epochs needed and getting way better results for training loss, but slowing down the computation time with a fixed number of epochs. I would say that it is a good idea to increase the filter dimension if we make sure to reduce the number of epochs, thus mitigating the whole computation time while maintaining a good accuracy and risking not to reach overfitting.

3.4 4/6: CNN using optimization techniques

3.4.1 a: Batch normalization

Loss = 0.128
Accuracy = 42%
Computation time = 6m 43s
Applying Batch Normalization means to normalize the output of every convolutional layer, decreasing the impact of previous layers to succeeding ones. This is visible in a faster learning process resulting in a better accuracy score.

3.4.2 b: BN + Increasing FC1 neuron number

Loss = 0.044
Accuracy = 45%
Computation time = 8m 11s

Increasing neuron number causes a more flexible classification due to the greater number of elaborating units.

3.4.3 c: BN + Dropout on FC1

Loss = 0.623
Accuracy = 46%
Computation time = 7m 04s
Using the dropout means to negate the contribute of some neurons at random in the learning phase. This is used to soften the co-dependency between neurons in order to avoid overfitting. When we plot the accuracy we notice it doesn't saturate as fast as in the other cases. See fig. 3.2 for loss and accuracy plot as it is the best performing CNN I found. Below you can see the CNN class:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        #conv2d first parameter is the number of kernels at input (you get
            it from the output value of the previous layer)
        #conv2d second parameter is the number of kernels you wanna have
            in your convolution, so it will be the n. of kernels at output.
        #conv2d third, fourth and fifth parameters are, as you can read,
            kernel_size, stride and zero padding :)
        self.conv1 = nn.Conv2d(3, 128, kernel_size=5, stride=2, padding=0)
        self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding
            =0)
        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding
            =0)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv_final = nn.Conv2d(128, 256, kernel_size=3, stride=1,
            padding=0)
        self.fc1 = nn.Linear(256 * 4 * 4, 4096)
        self.fc2 = nn.Linear(4096, n_classes) #last FC for classification
        self.bn1 = nn.BatchNorm2d(128)
        self.bn2 = nn.BatchNorm2d(128)
        self.bn3 = nn.BatchNorm2d(128)
        self.bnf = nn.BatchNorm2d(256)
        self.drop = nn.Dropout(0.5)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        x = F.relu(self.pool(self.bnf(self.conv_final(x))))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
```

```
        #hint: dropout goes here!
        x = self.drop(x)
        x = self.fc2(x)
        return x


    ####RUNNING CODE FROM HERE:

start = time.time()

#transform are heavily used to do simple and complex transformation and
    data augmentation
transform_train = transforms.Compose(
    [
        #transforms.RandomHorizontalFlip(),
     transforms.Resize((32,32)),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

transform_test = transforms.Compose(
    [
     transforms.Resize((32,32)),
     transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
     ])

trainset = torchvision.datasets.CIFAR100(root=mauve'mauve./mauvedata
    mauve', train=True,
                                        download=True, transform=
                                            transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256,
                                          shuffle=True, num_workers=4,
                                              drop_last=True)

testset = torchvision.datasets.CIFAR100(root=mauve'mauve./mauvedata
    mauve', train=False,
                                       download=True, transform=
                                           transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=256,
                                         shuffle=False, num_workers=4,
                                             drop_last=True)
```

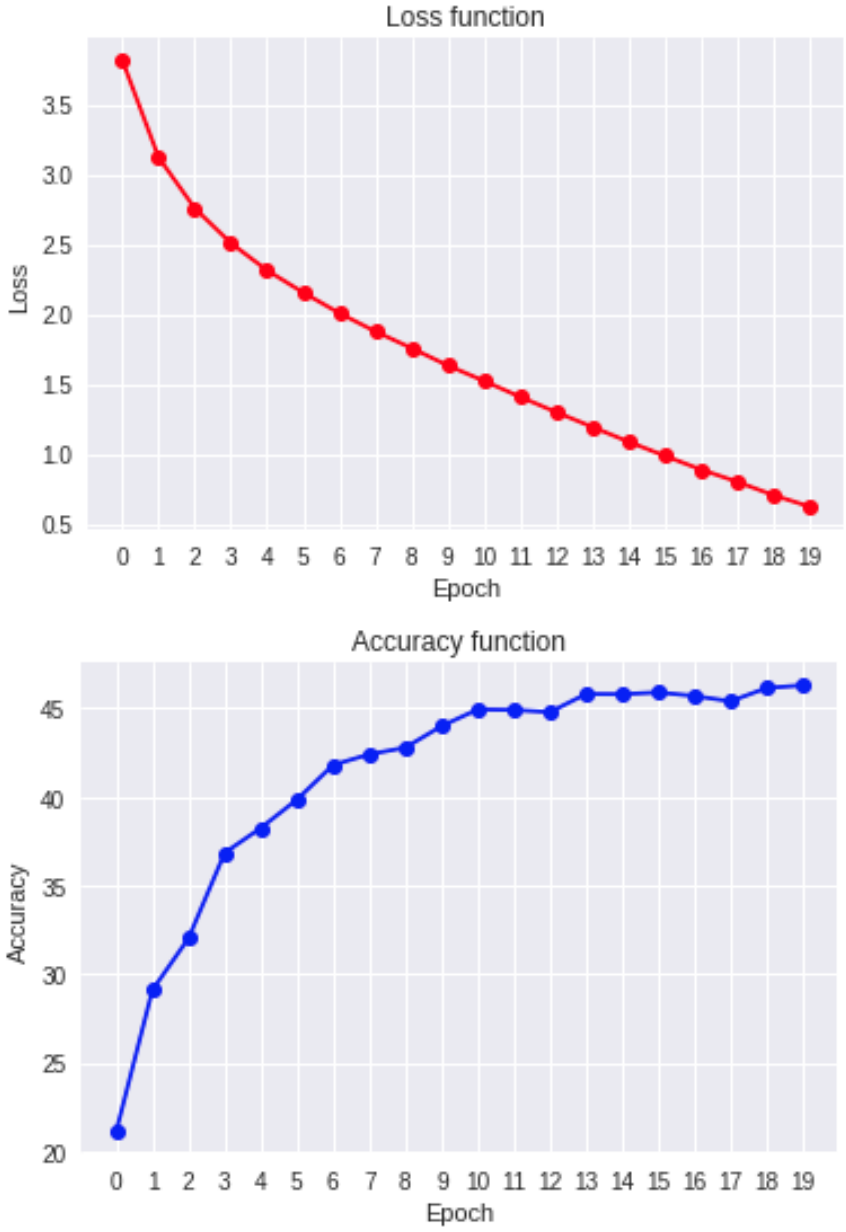

Figure 3.2: CNN using BN+dropout

3.5 5/6: Data augmentation

Data augmentation techniques permit to increase the size of the training set by modifying the images we already have. This is possible because the network

doesn't recognize a modified image as the same as its details are positioned in different sections of it, allowing to increase the dataset without overfitting. In this part of the homework I chose not to use Batch Normalization to observe only the contribute of the data augmentation to the overall accuracy. That being said I noticed that using Batch Normalization amplifies the effect of the data augmentation in terms of accuracy gain.

3.5.1 a: Random horizontal flipping

Loss = 0.996
Accuracy = 35%
Computation time = 6m 23s
Horizontal flipping of the images simply consists in flipping them horizontally.

3.5.2 b: Random crop

Loss = 2.114
Accuracy = 34%
Computation time = 6m 36s
Random crop is taking a random smaller section of the image and resizing it to the original image dimension. We can notice a greter loss value when compared to horizontal flipping.

3.6 6/6: ResNet18

Loss = 0.040
Accuracy = 80%
Computation time = 1h 4m 11s
In this case we use the pre-trained network ResNet18 and adapt it to our problem. We also use horizontal flip for data augmentation as it is the one which showed the best improvement in the previous step. Using such a deep pre-trained network guarantees a huge increase in accuracy. The complexity is clear by considering the computation time which is way longer. See fig. 3.3 for the results. I just uncommented the part of the code which used the ResNet and applied Horizontal Flip in the trasformations to do this part of the homework:

```
#for Residual Network:
net = models.resnet18(pretrained=True)
net.fc = nn.Linear(512, n_classes) #changing the fully connected layer of
    the already allocated network
####
...
#transform are heavily used to do simple and complex transformation and
    data augmentation
transform_train = transforms.Compose(
    [
```

```
        transforms.RandomHorizontalFlip(),
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
       ])

transform_test = transforms.Compose(
     [
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
        ])

trainset = torchvision.datasets.CIFAR100(root=mauve'mauve./mauvedata
     mauve', train=True,
                                          download=True, transform=
                                              transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,
                                            shuffle=True, num_workers=4,
                                              drop_last=True)

testset = torchvision.datasets.CIFAR100(root=mauve'mauve./mauvedata
     mauve', train=False,
                                         download=True, transform=
                                             transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=128,
                                           shuffle=False, num_workers=4,
                                             drop_last=True)
```

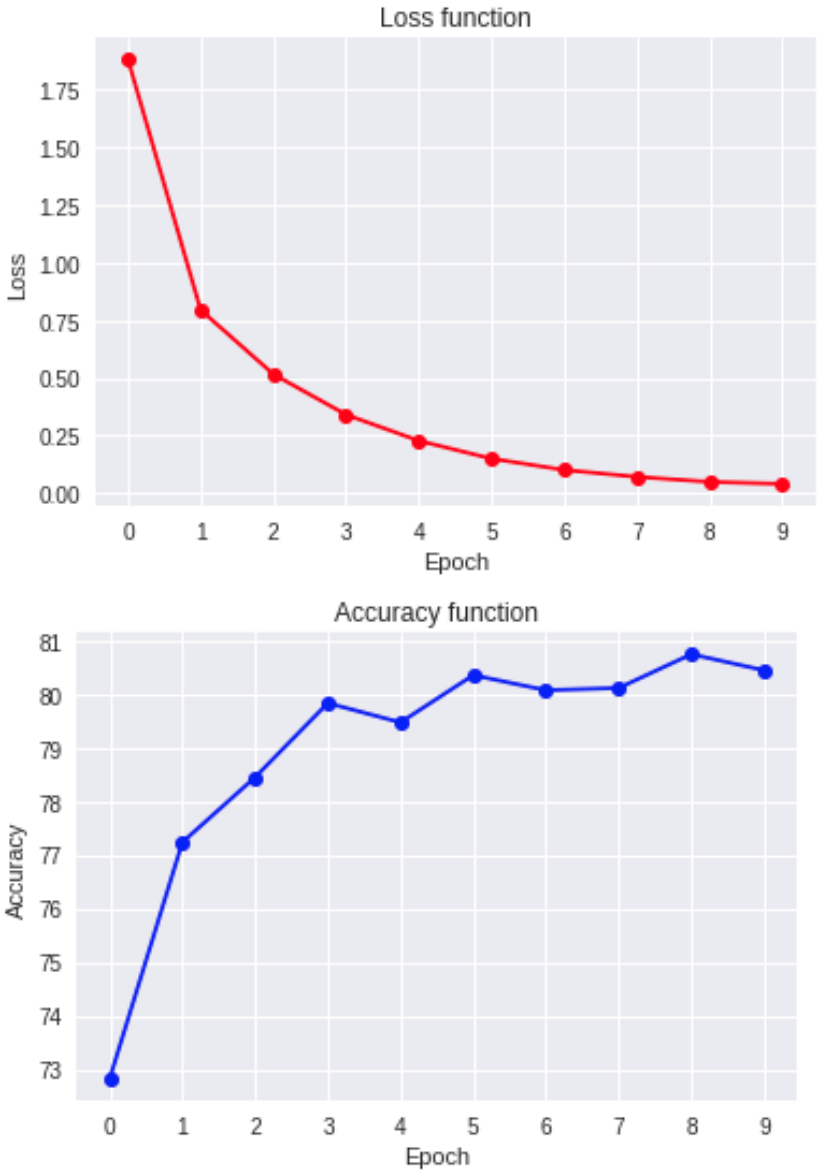

Figure 3.3: Resnet18 loss and accuracy