

POLITECNICO DI TORINO

01SQIOV

# Artificial Intelligence and Machine Learning

HOMEWORK 3:  
DEEP LEARNING

Prof. Barbara CAPUTO

Prof. Paolo RUSSO

Nour SAEED

POLITECNICO DI TORINO

DEPARTMENT OF COMPUTER ENGINEERING, CINEMA, & MECHATRONICS

S250409@studenti.polito.it

January 5, 2019

## **Abstract**

This report is a required deliverable for the course of AIML. It deals with Deep Learning, more specifically MLP, CNNs and ResNet18.

Since the document that was provided was a jupyter notebook file, the report below is the PDF version of the updated document. It was commented.

The document is provided with the files.

```
1 !pip3 install -q http://download.pytorch.org/whl/cu90/torch-0.4.0-cp36-cp36m-linux_x86_64.whl
2 !pip3 install torchvision
```

## Library import

```
1 %matplotlib inline
2 import torch
3 import torchvision
4 from torchvision import models
5 import torchvision.transforms as transforms
6 from torchvision.transforms import ToPILImage
7 import torch.optim as optim
8 from IPython.display import Javascript
9
10 import torch.nn as nn
11 import torch.nn.functional as F
12
13 import matplotlib.pyplot as plt
14
15 import numpy as np
```

## Definition of functions for display

```
1 # function to show an image
2 def imshow(img):
3     img = img / 2 + 0.5     # unnormalize
4     npimg = img.numpy()
5     plt.imshow(np.transpose(npimg, (1, 2, 0)))
6     plt.show()
7
8 def plot_kernel(model):
9     model_weights = model.state_dict()
10    fig = plt.figure()
11    plt.figure(figsize=(10,10))
12    for idx, filt in enumerate(model_weights['conv1.weight']):
13        #print(filt[0, :, :])
14        if idx >= 32: continue
15        plt.subplot(4,8, idx + 1)
16        plt.imshow(filt[0, :, :], cmap="gray")
17        plt.axis('off')
18
19    plt.show()
20
21 def plot_kernel_output(model, images):
22     fig1 = plt.figure()
23     plt.figure(figsize=(1,1))
24
25     img_normalized = (images[0] - images[0].min()) / (images[0].max() - images[0].min())
26     plt.imshow(img_normalized.numpy().transpose(1,2,0))
27     plt.show()
28     output = model.conv1(images)
29     layer_1 = output[0, :, :, :]
30     layer_1 = layer_1.data
31
32     fig = plt.figure()
33     plt.figure(figsize=(10,10))
34     for idx, filt in enumerate(layer_1):
35         if idx >= 32: continue
36         plt.subplot(4,8, idx + 1)
37         plt.imshow(filt, cmap="gray")
38         plt.axis('off')
39     plt.show()
```

## Test Accuracy Function

```
1 def test_accuracy(net, dataloader):
2
3     #check accuracy on whole test set
4     correct = 0
5     total = 0
6     net.eval() #important for deactivating dropout and correctly use batchnorm accumulated statistics
7     with torch.no_grad():
8         for data in dataloader:
9             images, labels = data
10            images = images.cuda()
11            labels = labels.cuda()
12            outputs = net(images)
13            _, predicted = torch.max(outputs.data, 1)
14            total += labels.size(0)
15            correct += (predicted == labels).sum().item()
16    accuracy = 100 * correct / total
17    print('Accuracy of the network on the test set: %d %%' % (
18    accuracy))
19    return accuracy
```

## Number of Classes

Since we are using the CIFAR 100, the number of classes in 100

```
1 n_classes = 100
```

## Traditional Neural Network

In this section we are going to train a **multilayer perceptron network**.

To begin we define the class for the network. It consists of two hidden layers and a last FC layer network of 100 classes.

```
1 # function to define an old style fully connected network (multilayer perceptrons)
2 class old_nn(nn.Module):
3     def __init__(self):
4         super(old_nn, self).__init__()
5         self.fc1 = nn.Linear(32*32*3, 4096)
6         self.fc2 = nn.Linear(4096, 4096)
7         self.fc3 = nn.Linear(4096, n_classes) #last FC for classification
8
9     def forward(self, x):
10        x = x.view(x.shape[0], -1)
11        x = F.sigmoid(self.fc1(x))
12        x = F.sigmoid(self.fc2(x))
13        x = self.fc3(x)
14        return x
```

The first part of the code is related to loading the data and providing training and testing sets. It deals with all the resizingm transformations, data augmentation... Here we don't use data augmentation techniques.

```
1 #transform are heavily used to do simple and complex transformation and data augmentation
2 transform_train = transforms.Compose(
3     [
4         #transforms.RandomHorizontalFlip(),
5         transforms.Resize((32,32)),
6         transforms.ToTensor(),
7         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
8     ])
9
10 transform_test = transforms.Compose(
11     [
12         transforms.Resize((32,32)),
13         transforms.ToTensor(),
14         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
15     ])
16
17 trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
18                                         download=True, transform=transform_train)
19 trainloader = torch.utils.data.DataLoader(trainset, batch_size=256,
20                                         shuffle=True, num_workers=4,drop_last=True)
21
22 testset = torchvision.datasets.CIFAR100(root='./data', train=False,
23                                         download=True, transform=transform_test)
24 testloader = torch.utils.data.DataLoader(testset, batch_size=256,
25                                         shuffle=False, num_workers=4,drop_last=True)
26
27 dataiter = iter(trainloader)
```

Now we create an old style NN and we train

```
1 net = old_nn().
2 net = net.cuda()
3
4 criterion = nn.CrossEntropyLoss().cuda() #it already does softmax computation for use!
5 optimizer = optim.Adam(net.parameters(), lr=0.0001) #better convergency w.r.t simple SGD :)
6
7 #####TRAINING PHASE#####
8 n_loss_print = len(trainloader) #print every epoch, use smaller numbers if you wanna print loss more often!
9
10 n_epochs = 20
11
12 lossPlot=[]
13 accPlot=[]
14
15 for epoch in range(n_epochs): # loop over the dataset multiple times
16     net.train() #important for activating dropout and correctly train batchnorm
17     running_loss = 0.0
18     for i, data in enumerate(trainloader, 0):
19         # get the inputs and cast them into cuda wrapper
20         inputs, labels = data
21         inputs = inputs.cuda()
22         labels = labels.cuda()
23         # zero the parameter gradients
24         optimizer.zero_grad()
25
26         # forward + backward + optimize
27         outputs = net(inputs)
28         loss = criterion(outputs, labels)
29         loss.backward()
30         optimizer.step()
31
```

```

32 # print statistics
33 running_loss += loss.item()
34 if i % n_loss_print == (n_loss_print -1):
35     temp=running_loss / n_loss_print
36     print('[%d, %5d] loss: %.3f' %
37           (epoch + 1, i + 1, temp))
38     lossPlot.append(temp)
39     running_loss = 0.0
40 accPlot.append(test_accuracy(net,testloader))
41
42 print('Finished Training')
43

```

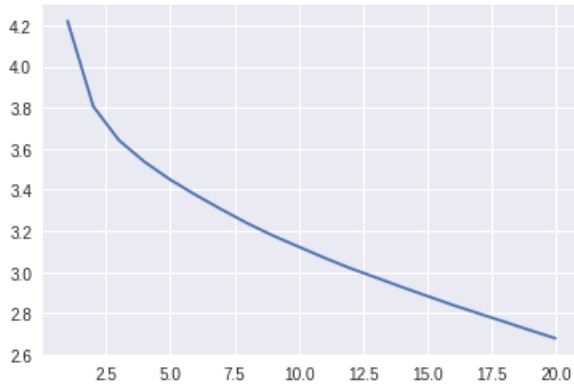
Now we plot the Loss curve

```

1 plt.plot(range(1,len(lossPlot)+1),lossPlot)
2 plt.show

```

↳ <function matplotlib.pyplot.show>



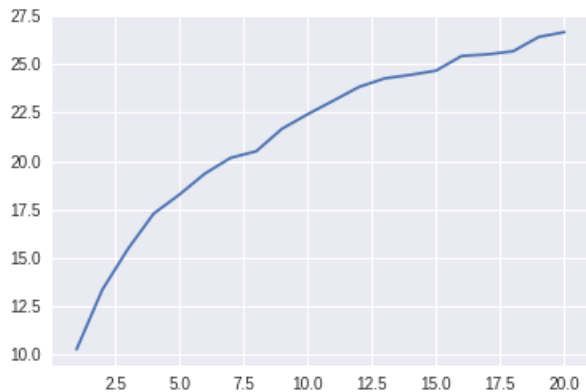
And the accuracy curve

```

1 plt.plot(range(1,len(accPlot)+1),accPlot)
2 plt.show

```

↳ <function matplotlib.pyplot.show>



## Comments

The accuracy is low as expected. Looking deeper into the given dataset, I would like to point out some point.

First of all this dataset contains images which are RGB (x3 the data) . Also, images vary alot, so the amount of feature would be huge. We would need a deeper NN.

## Convolutional Neural Networks

Let us use the CNN given.

```

1 #function to define the convolutional network
2 class CNN(nn.Module):
3     def __init__(self,s1=32,s2=32,s3=32,s4=64,o=4096):
4         super(CNN, self).__init__()
5         #conv2d first parameter is the number of kernels at input (you get it from the output value of the previous layer)
6         #conv2d second parameter is the number of kernels you wanna have in your convolution, so it will be the n. of kern
7         #conv2d third, fourth and fifth parameters are, as you can read, kernel_size, stride and zero padding :)
8         self.conv1 = nn.Conv2d(3, s1, kernel_size=5, stride=2, padding=0)
9         self.conv2 = nn.Conv2d(s1, s2, kernel_size=3, stride=1, padding=0)
10        self.conv3 = nn.Conv2d(s2, s3, kernel_size=3, stride=1, padding=0)
11        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)

```

```

12 self.conv_final = nn.Conv2d(s3, s4, kernel_size=3, stride=1, padding=0)
13 self.fc1 = nn.Linear(s4 * 4 * 4, o)
14 self.fc2 = nn.Linear(o, n_classes) #last FC for classification
15
16 def forward(self, x):
17     x = F.relu(self.conv1(x))
18     x = F.relu(self.conv2(x))
19     x = F.relu(self.conv3(x))
20     x = F.relu(self.pool(self.conv_final(x)))
21     x = x.view(x.shape[0], -1)
22     x = F.relu(self.fc1(x))
23     #hint: dropout goes here!
24     x = self.fc2(x)
25     return x
26

```

We now define the CNN

```

1 net = CNN()

```

And train

```

1 net = net.cuda()
2
3 criterion = nn.CrossEntropyLoss().cuda() #it already does softmax computation for use!
4 optimizer = optim.Adam(net.parameters(), lr=0.0001) #better convergency w.r.t simple SGD :)
5
6 #####TRAINING PHASE#####
7 n_loss_print = len(trainloader) #print every epoch, use smaller numbers if you wanna print loss more often!
8
9 n_epochs = 20
10
11 lossPlot=[]
12 accPlot=[]
13
14 for epoch in range(n_epochs): # loop over the dataset multiple times
15     net.train() #important for activating dropout and correctly train batchnorm
16     running_loss = 0.0
17     for i, data in enumerate(trainloader, 0):
18         # get the inputs and cast them into cuda wrapper
19         inputs, labels = data
20         inputs = inputs.cuda()
21         labels = labels.cuda()
22         # zero the parameter gradients
23         optimizer.zero_grad()
24
25         # forward + backward + optimize
26         outputs = net(inputs)
27         loss = criterion(outputs, labels)
28         loss.backward()
29         optimizer.step()
30
31         # print statistics
32         running_loss += loss.item()
33         if i % n_loss_print == (n_loss_print - 1):
34             temp=running_loss / n_loss_print
35             print('[%d, %5d] loss: %.3f' %
36                 (epoch + 1, i + 1, temp))
37             lossPlot.append(temp)
38             running_loss = 0.0
39         accPlot.append(test_accuracy(net,testloader))
40
41 print('Finished Training')
42

```

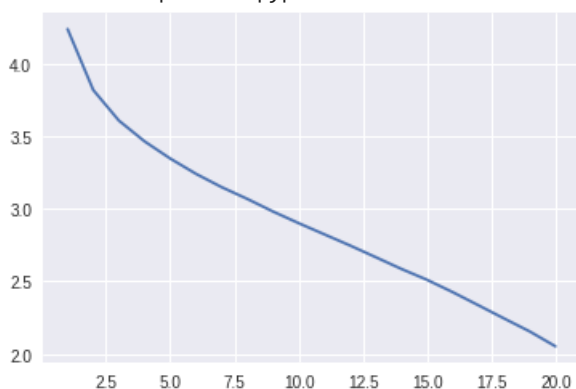
Now we plot the Loss curve

```

1 plt.plot(range(1,len(lossPlot)+1),lossPlot)
2 plt.show

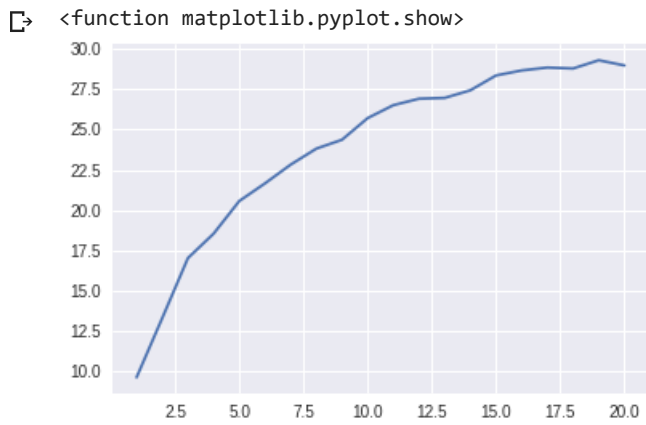
```

↗ <function matplotlib.pyplot.show>



And the accuracy curve

```
1 plt.plot(range(1, len(accPlot)+1), accPlot)
2 plt.show
```



## Comments

Even though the number of parameters to train required in CNN is less than that of a MLP, the accuracy is a little bit better than that of the MLP. With the high number of features it has to learn (noting that there are 100 classes), and considering the architecture of the following CNN, the accuracy, even a little bit low, could be expected

## More CNNs

### 128/128/128/256

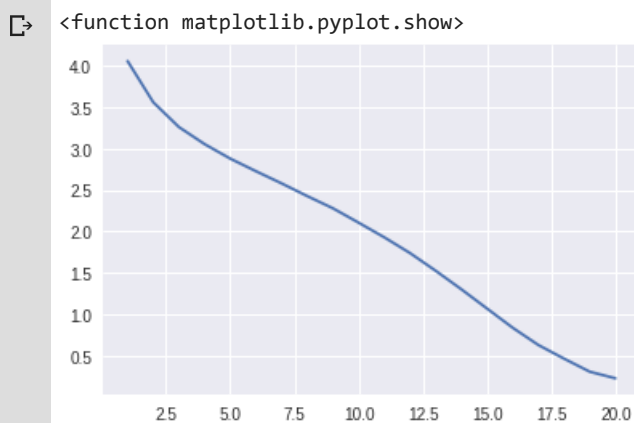
We change the number of convolutional filters to 128/128/128/256

```
1 net=CNN(128,128,128,256).
```

We train the network same way as we did above.

We plot the Loss curve

```
1 plt.plot(range(1, len(lossPlot)+1), lossPlot)
2 plt.show
```




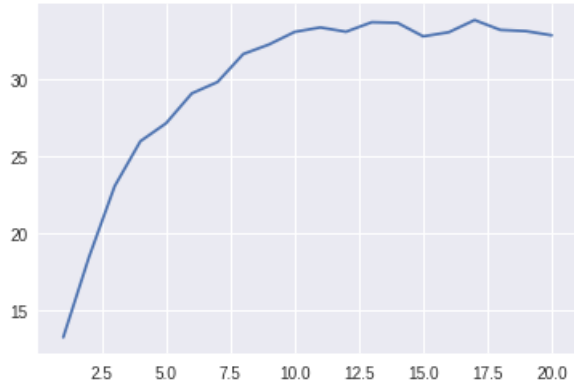
+ CODE

+ TEXT

And the accuracy curve

```
1 plt.plot(range(1, len(accPlot)+1), accPlot)
2 plt.show
```

 <function matplotlib.pyplot.show>



The time required here was 386.719 s

## 256/256/256/512


We change the number of convolutional filters to 256/256/256/512

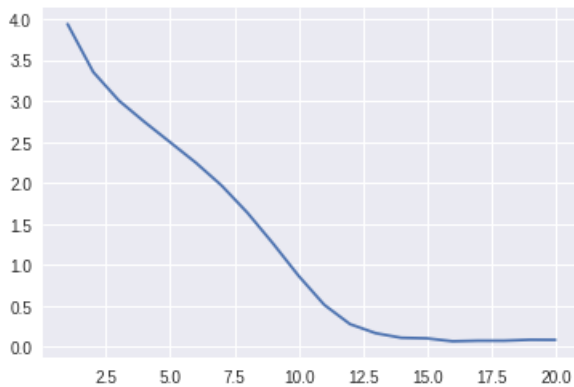
```
1 net=CNN(256,256,256,512).
```

We train the network same way as we did above.

We plot the Loss curve


```
1 plt.plot(range(1, len(lossPlot)+1), lossPlot)
2 plt.show
```

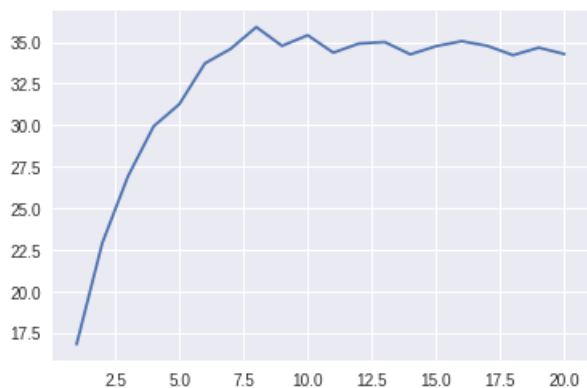
 <function matplotlib.pyplot.show>



And the accuracy curve

```
1 plt.plot(range(1, len(accPlot)+1), accPlot)
2 plt.show
```

 <function matplotlib.pyplot.show>





The time required here was 857.179 s

## 512/512/512/1024

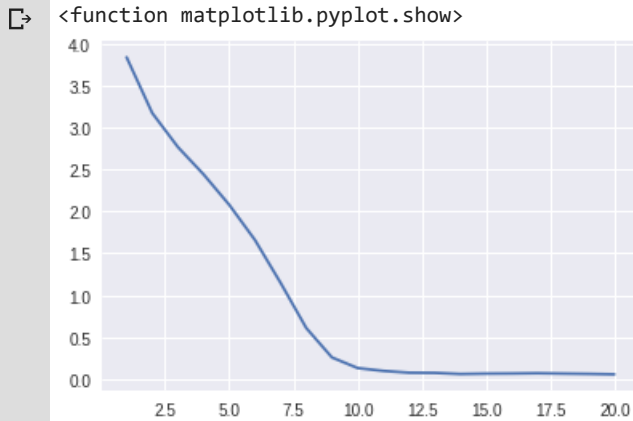
We change the number of convolutional filters to 512/512/512/1024

```
1 net=CNN(512,512,512,1024).
```

We train the network same way as we did above.

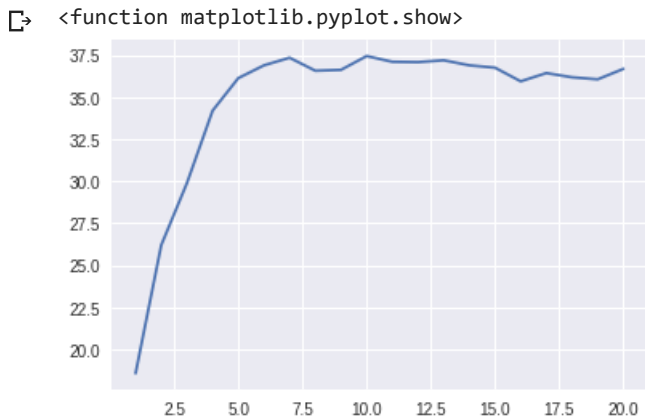
We plot the Loss curve

```
1 plt.plot(range(1,len(lossPlot)+1),lossPlot)
2 plt.show
```



And the accuracy curve

```
1 plt.plot(range(1,len(accPlot)+1),accPlot)
2 plt.show
```



The time required here was 2606.918 s

## Comments

We notice that at the end, the final accuracy is around 7% more than that of the starting one. But this accuracy isn't so much better than that of the previous cases and it takes more time than the previous cases (As we double the number the time required increases more than the double).

But what is interesting is that the rate, at which the accuracy increases (loss decreases), increases with the increase in number of convolutional filters.

Another thing to notice is that the value is always stuck in the 35% range. This could be solved by decreasing the learning rate. If that doesn't work, we should probably think about changing our architecture since our network isn't able to generalize on the test set.

It would be good to have a validation set and seeing the loss for the validation also.

## Even More CNNs

Here we will use the network with 128/128/128/256 filters

## Batch Normalization

The CNN class is updated to take into consideration batch normalization

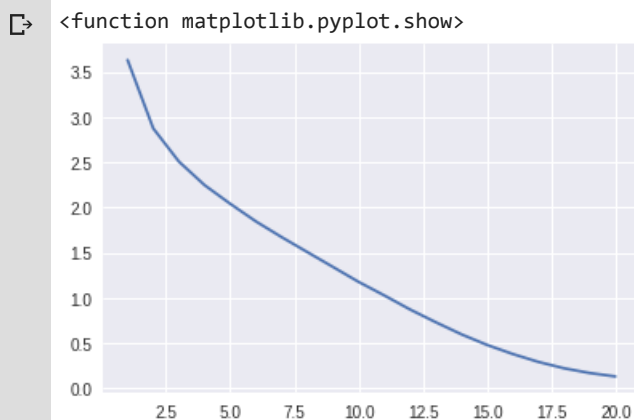
```
1 #function to define the convolutional network with normalization and dropout
2 class CNN(nn.Module):
3     def __init__(self,s1=32,s2=32,s3=32,s4=64,o=4096,BN=False,DO=False):
4         super(CNN, self).__init__()
5         #conv2d first parameter is the number of kernels at input (you get it from the output value of the previous layer)
6         #conv2d second parameter is the number of kernels you wanna have in your convolution, so it will be the n. of kern
7         #conv2d third, fourth and fifth parameters are, as you can read, kernel_size, stride and zero padding :)
8         self.conv1 = nn.Conv2d(3, s1, kernel_size=5, stride=2, padding=0)
9         self.conv1_bn = nn.BatchNorm2d(s1)
10        self.conv2 = nn.Conv2d(s1, s2, kernel_size=3, stride=1, padding=0)
11        self.conv2_bn = nn.BatchNorm2d(s2)
12        self.conv3 = nn.Conv2d(s2, s3, kernel_size=3, stride=1, padding=0)
13        self.conv3_bn = nn.BatchNorm2d(s3)
14        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
15        self.conv_final = nn.Conv2d(s3, s4, kernel_size=3, stride=1, padding=0)
16        self.conv_final_bn = nn.BatchNorm2d(s4)
17        self.fc1 = nn.Linear(s4 * 4 * 4, o)
18        self.dropout = nn.Dropout(0.5)
19        self.fc2 = nn.Linear(o, n_classes) #last FC for classification
20        self.BN=BN
21        self.DO=DO
22    def forward(self, x):
23        if(self.BN==False):
24            x = F.relu(self.conv1(x))
25            x = F.relu(self.conv2(x))
26            x = F.relu(self.conv3(x))
27            x = F.relu(self.pool(self.conv_final(x)))
28            x = x.view(x.shape[0], -1)
29            x = F.relu(self.fc1(x))
30        else:
31            x = F.relu(self.conv1_bn(self.conv1(x)))
32            x = F.relu(self.conv2_bn(self.conv2(x)))
33            x = F.relu(self.conv3_bn(self.conv3(x)))
34            x = F.relu(self.pool(self.conv_final_bn(self.conv_final(x))))
35            x = x.view(x.shape[0], -1)
36            x = F.relu(self.fc1(x))
37
38        #hint: dropout goes here!
39        if(self.DO):
40            x = self.dropout(x)
41
42        x = self.fc2(x)
43        return x
44
```

We define the NN and we train

```
1 net = CNN(128,128,128,256,4096,BN=True,DO=False)
```

Now we plot the Loss curve

```
1 plt.plot(range(1,len(lossPlot)+1),lossPlot)
2 plt.show
```

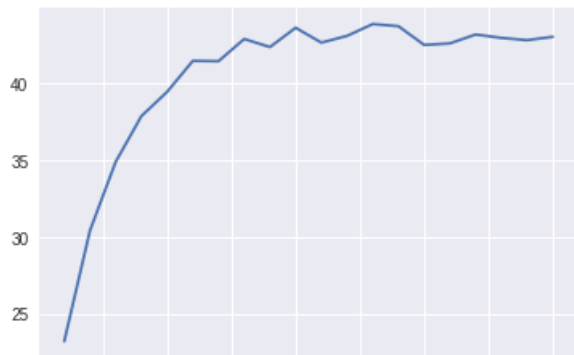


And the accuracy curve

```
1 plt.plot(range(1,len(accPlot)+1),accPlot)
2 plt.show
```



```
<function matplotlib.pyplot.show>
```



This required 408.545 s and yielded 42%

## ▼ BN with wider FC1

Here we use the batch normalization but with FC1 having 8192 neurons

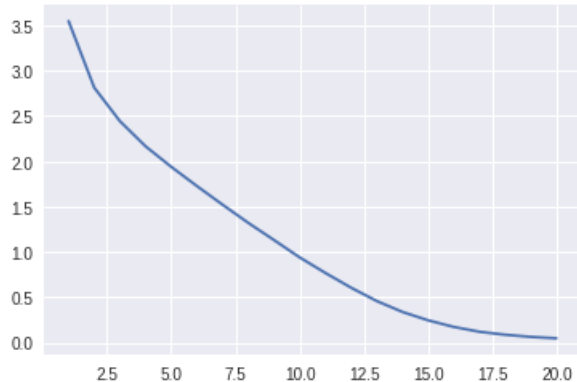
```
1 net = CNN(128,128,128,256,8192,BN=True,DO=False)
```

We train the net.

The loss curve is:

```
1 plt.plot(range(1,len(lossPlot)+1),lossPlot)
2 plt.show
```

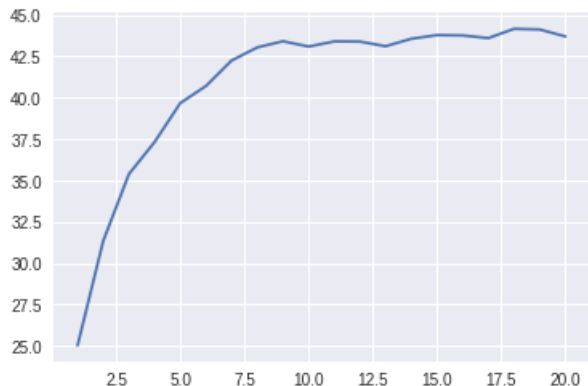
```
<function matplotlib.pyplot.show>
```



And the accuracy curve

```
1 plt.plot(range(1,len(accPlot)+1),accPlot)
2 plt.show
```

```
<function matplotlib.pyplot.show>
```



This required 471.075s and yielded 43% accuracy

## ▼ BN with 0.5 DropOut on FC1 (4096)

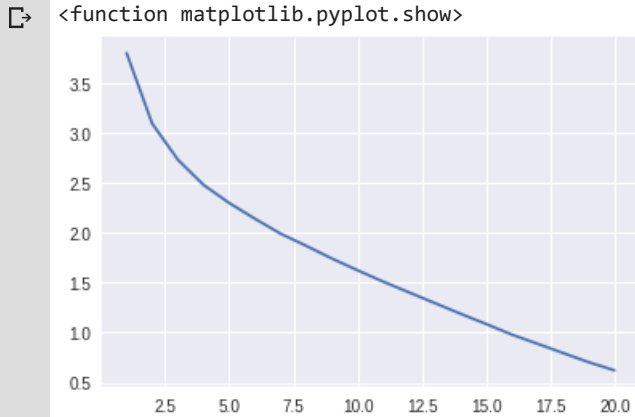
We now activate the dropout

```
1 net = CNN(128,128,128,256,4096,BN=True,DO=True)
```

We train the net.

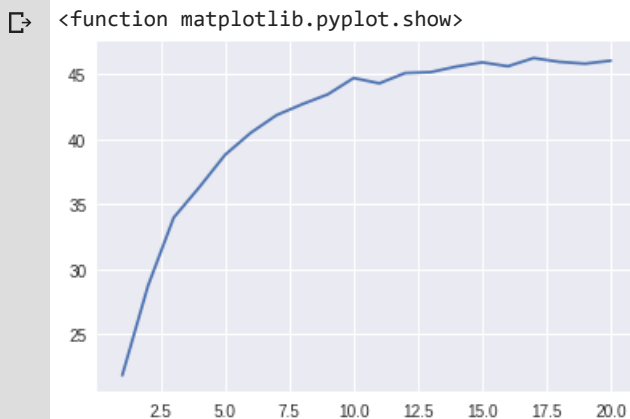
The loss curve is:

```
1 plt.plot(range(1,len(lossPlot)+1),lossPlot)
2 plt.show
```



And the accuracy curve

```
1 plt.plot(range(1,len(accPlot)+1),accPlot)
2 plt.show
```



This required 410.319s and yielded 46% accuracy

## Comments

As expected the accuracy when we applied BN would increase. We doubled the neurons on the output of FC1 but the computation time only increased 70s without that much increase in the accuracy.

Meanwhile when we applied BN with dropout of 0.5, it was able to reach more accuracy with less computational time. This was expected.

We could say that in order to increase the accuracy of a CNN, BN should be done. Dropout has helped increase the accuracy slightly. As a regularization technique, it could have solved some overfitting problem in our model. Playing with the values could help us achieve better accuracy.

Also when increasing the size of FC1 the oscillations on the accuracy at stability decreased. That was also evident in the case of dropout.

## Data Augmentation

We update our net to the required network to the basic 128/128/128/256 without batch normalization and dropout:

```
1 net = CNN(128,128,128,256,4096,BN=False,DO=False).
```

## Random Horizontal Flip

We change the data sets to include horizontal flips

```

1 #transforms are heavily used to do simple and complex transformation and data augmentation
2 transform_train = transforms.Compose(
3     [
4         transforms.RandomHorizontalFlip(),
5         transforms.Resize((32,32)),
6         transforms.ToTensor(),
7         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
8     ])
9
10 transform_test = transforms.Compose(
11     [
12         transforms.Resize((32,32)),
13         transforms.ToTensor(),
14         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
15     ])
16
17 trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
18                                         download=True, transform=transform_train)
19 trainloader = torch.utils.data.DataLoader(trainset, batch_size=256,
20                                         shuffle=True, num_workers=4, drop_last=True)
21
22 testset = torchvision.datasets.CIFAR100(root='./data', train=False,
23                                         download=True, transform=transform_test)
24 testloader = torch.utils.data.DataLoader(testset, batch_size=256,
25                                         shuffle=False, num_workers=4, drop_last=True)
26
27 dataiter = iter(trainloader)

```

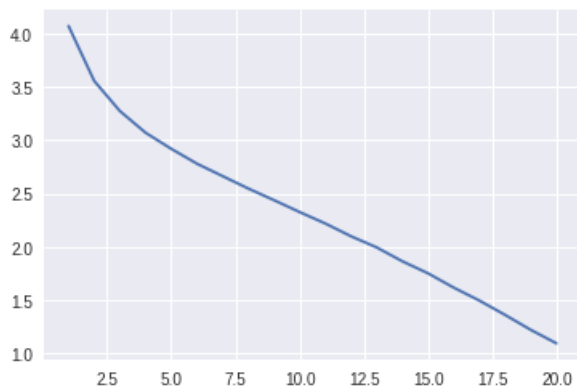
We then train the net and we plot the loss curve:

```

1 plt.plot(range(1, len(lossPlot)+1), lossPlot)
2 plt.show

```

↳ <function matplotlib.pyplot.show>



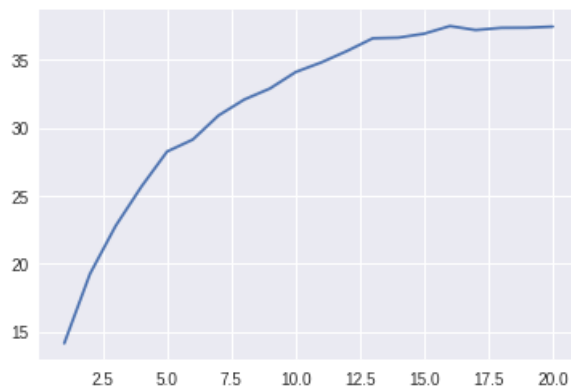
And the accuracy curve

```

1 plt.plot(range(1, len(accPlot)+1), accPlot)
2 plt.show

```

↳ <function matplotlib.pyplot.show>



It took 395.366s and gave an accuracy of around 37% (almost a 9% increase)

## ▼ Random Crops

We change the data sets now to include both horizontal flips and random crops

```

1 #transforms are heavily used to do simple and complex transformation and data augmentation
2 transform_train = transforms.Compose(

```

```

3  [
4      transforms.RandomHorizontalFlip(),
5      transforms.Resize((40,40)),
6      transforms.RandomCrop((32,32)),
7      transforms.ToTensor(),
8      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
9  ])
10
11 transform_test = transforms.Compose(
12     [
13         transforms.Resize((32,32)),
14         transforms.ToTensor(),
15         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
16     ])
17
18 trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
19                                         download=True, transform=transform_train)
20 trainloader = torch.utils.data.DataLoader(trainset, batch_size=256,
21                                         shuffle=True, num_workers=4, drop_last=True)
22
23 testset = torchvision.datasets.CIFAR100(root='./data', train=False,
24                                         download=True, transform=transform_test)
25 testloader = torch.utils.data.DataLoader(testset, batch_size=256,
26                                         shuffle=False, num_workers=4, drop_last=True)
27
28 dataiter = iter(trainloader)

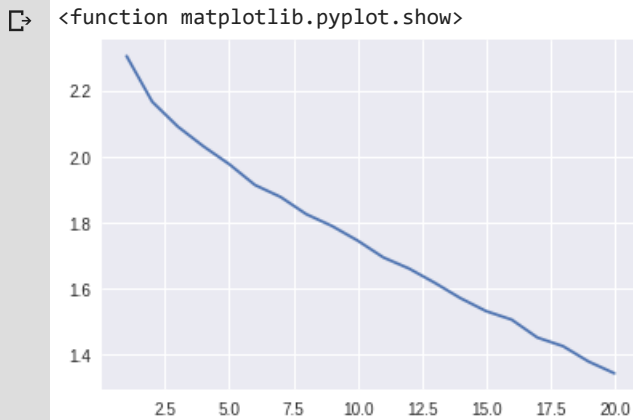
```

We then train the net and we plot the loss curve:

```

1 plt.plot(range(1, len(lossPlot)+1), lossPlot)
2 plt.show

```

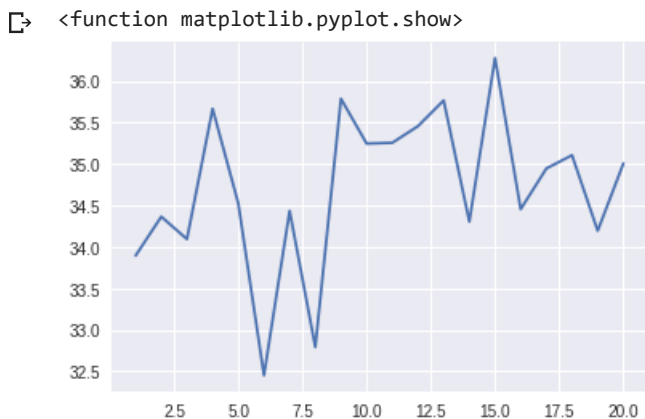


And the accuracy curve

```

1 plt.plot(range(1, len(accPlot)+1), accPlot)
2 plt.show

```



It took 419.092s and gave an accuracy of around 35%. Here it is somewhat easier to learn features since there were cropped images.

## Comments

Applying these data augmentation methods have increased our accuracy around 7%, and also the accuracy rate at the beginning (33% accuracy from first epoch when both flips and crops are applied). This method along with Batch normalization and drop could help increase the accuracy of our network

# Residual Neural Network

We are required to use ResNet18 with the following parameters:

128 batch size, 10 epochs, 224x224 resolution, Adam solver with learning rate 0.0001

We will also use data augmentation (Horizontal flips since they gave best accuracy. And since we would loss resolution from resizing 32x32 to 224x224, cropping would probably make feature extraction different). We update our dataset:

```
1 #transforms are heavily used to do simple and complex transformation and data augmentation
2 transform_train = transforms.Compose(
3     [
4         transforms.RandomHorizontalFlip(),
5         transforms.Resize((224,224)),
6         #transforms.RandomCrop((224,224)),
7         transforms.ToTensor(),
8         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
9     ])
10
11 transform_test = transforms.Compose(
12     [
13         transforms.Resize((224,224)),
14         transforms.ToTensor(),
15         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
16     ])
17
18 trainset = torchvision.datasets.CIFAR100(root='./data', train=True,
19                                         download=True, transform=transform_train)
20 trainloader = torch.utils.data.DataLoader(trainset, batch_size=128,
21                                         shuffle=True, num_workers=4,drop_last=True)
22
23 testset = torchvision.datasets.CIFAR100(root='./data', train=False,
24                                         download=True, transform=transform_test)
25 testloader = torch.utils.data.DataLoader(testset, batch_size=128,
26                                         shuffle=False, num_workers=4,drop_last=True)
27
28 dataiter = iter(trainloader)
```

And import the ResNet

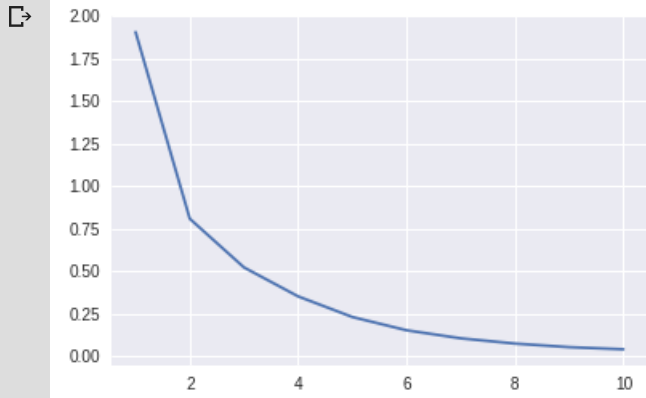
```
1 net = models.resnet18(pretrained=True)
2 net.fc = nn.Linear(512, n_classes) #changing the fully connected layer of the already allocated network
```

We train using the updated parameters

```
1 net = net.cuda()
2
3 criterion = nn.CrossEntropyLoss().cuda() #it already does softmax computation for use!
4 optimizer = optim.Adam(net.parameters(), lr=0.0001) #better convergency w.r.t simple SGD :)
5
6
7 #####TRAINING PHASE#####
8 n_loss_print = len(trainloader) #print every epoch, use smaller numbers if you wanna print loss more often!
9
10 n_epochs = 10
11
12 lossPlot=[]
13 accPlot=[]
14
15 for epoch in range(n_epochs): # loop over the dataset multiple times
16     net.train() #important for activating dropout and correctly train batchnorm
17     running_loss = 0.0
18     for i, data in enumerate(trainloader, 0):
19         # get the inputs and cast them into cuda wrapper
20         inputs, labels = data
21         inputs = inputs.cuda()
22         labels = labels.cuda()
23         # zero the parameter gradients
24         optimizer.zero_grad()
25
26         # forward + backward + optimize
27         outputs = net(inputs)
28         loss = criterion(outputs, labels)
29         loss.backward()
30         optimizer.step()
31
32         # print statistics
33         running_loss += loss.item()
34         if i % n_loss_print == (n_loss_print - 1):
35             temp=running_loss / n_loss_print
36             print('[%d, %5d] loss: %.3f' %
37                   (epoch + 1, i + 1, temp))
38             lossPlot.append(temp)
39             running_loss = 0.0
40     accPlot.append(test_accuracy(net,testloader))
41
42 print('Finished Training')
43
```

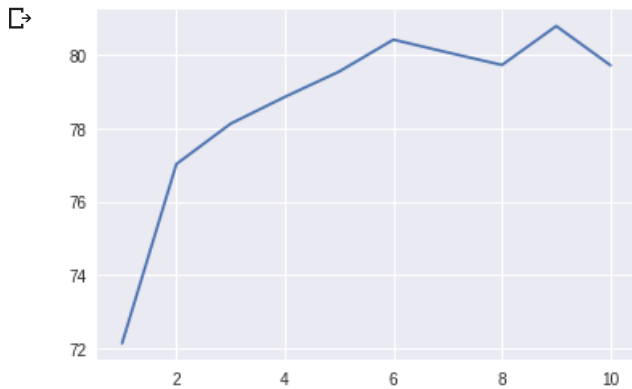
We plot the Loss curve

```
1 plt.plot(range(1, len(lossPlot)+1), lossPlot)
2 plt.show()
```



and the accuracy

```
1 plt.plot(range(1, len(accPlot)+1), accPlot)
2 plt.show()
```



The accuracy is around 80% and it took 699.921s

## Comments

The ResNet18 provides the best accuracy. This network being pretrained for imagenet, and fined tuned for our dataset acts behaves good. Using transfer learning in training networks is a good strategy that saves us time on deep networks and could substantially provide better results.

## Others

The folowing code has been used to save the models with best accuracy

```
1 #To save the file
2 txt=open("network.pt","wb")
3 torch.save(net.state_dict(), txt)
4 txt.close()
```

```
1 #To download the file
2 from google.colab import files
3 files.download("resnet.h5")
```



