Artificial Intelligence and Machine Learning

Homework 3: Deep Learning

Jeanpierre Francois S243920

January 18, 2019

1 Description

In this homework I explored the tip of the iceberg regarding deep learning and neural networks applications.
The language used in this homework is python, the framework implemented is pytorch.
The homework was written and run onto google's Colab platform.
The images dataset used for training and testing the networks is the CIFAR100 imported from torchvision.
Whithin the homework I worked with three different kinds of neural network:

1. Old neural network
2. Convolutional neural network
3. ResNet-18

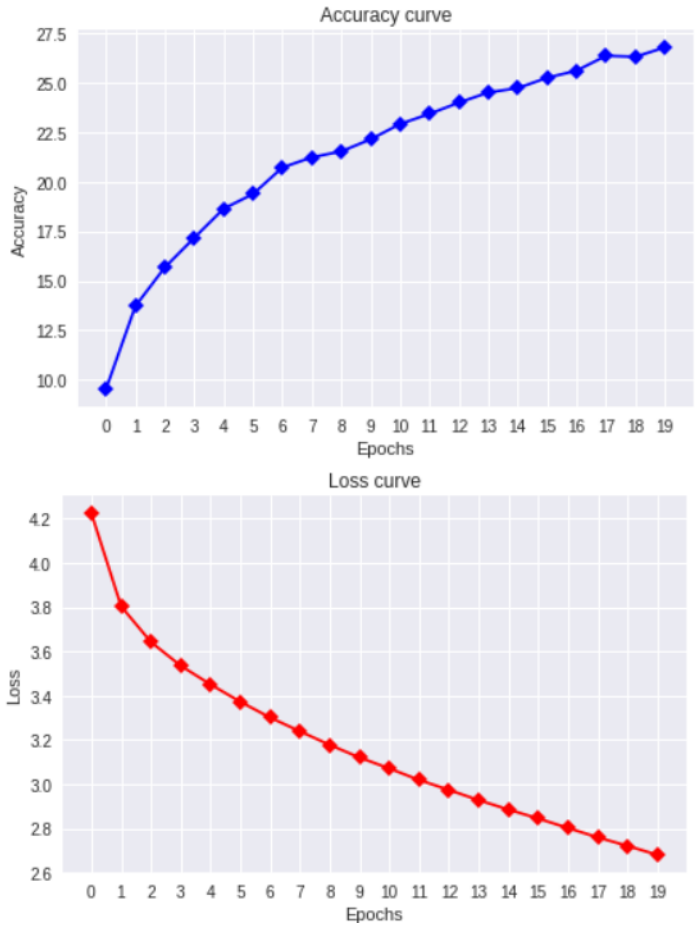
2 Old Neural Networks

The so called old neural network implemented is a simple traditional net, provided with forward and backward propagation, composed of three fully connected layers: two hidden and one for the classification. The transformations comprohends a resize and a normalization of the input images with a batch size of 256. The chosen solver is Adam with a learning rate of 0.0001.

```
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)

    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
```

```
transform_train = transforms.Compose(
    [
     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='./data', 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='./data', train=False,
                                        download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=256,
                                         shuffle=False, num_workers=4,drop_last=True)


dataiter = iter(trainloader)
```

Storing at each epoch the accuracy and loss obtained, within 20 epochs I obtained a final score on the test set of 26% with a loss of 2.680 in 4 minutes and 7 seconds as shown in the subsequent results.



Despite the simplicity of the deployed net I got an accuracy of 26% that may seem low is still a pretty good result working with 100 classes, the starting chance was of 1% and this neural network rose it up of 26 times, which is nice.

3 Convolutional Neural Networks

Convolutional neural networks (cnn) are neural net applied to images where each hidden unit is connected to a small patch of the input and shares weights across space. This networks are capable of learning filters without prior knowledge.
In this practice multiple different cnn have been tested.
I trained the cnn from scratch using the various network configuration and data augmentation requested in the homework obtaining the following results.

Net configuration	Accuracy	Time	Loss
2of6_32/32/32/64	30%	00:04:30	2.116
3of6_128/128/128/256	32%	00:06:17	0.149
3of6_256/256/256/512	35%	00:14:12	0.076
3of6_512/512/512/1024	36%	00:42:32	0.049
4of6_a	43%	00:06:31	0.122
4of6_b	44%	00:07:50	0.048
4of6_c	45%	00:06:30	0.641
5of6_a	35%	00:05:51	0.827
5of6_b	33%	00:06:20	2.101
Best cnn found	58%	00:50:36	0.544

Every cnn deployed for the work share a similar base architecture: four convolutional layers, two fully connected with ReLu and max pooling strategies.

3.1 2of6

This is the first convolutional network deployed with a number of convolutional filters of 32/32/32/64. As we can see from the accuracy obtained it's way better in classifying CIFAR100's images than a simple neural network composed of fully connected layers as the one implemented in the previous öld nn.

3.2 3of6

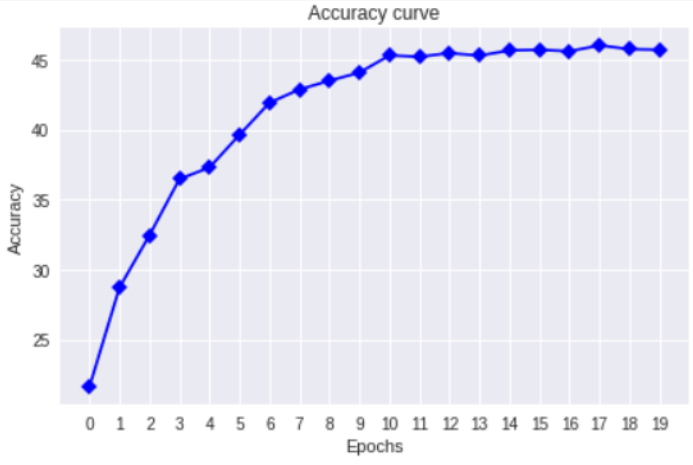
The computational time grows as I increment the number of convolutional filters from 32/32/32/64 to 128/128/128/256, then again to 256/256/256/512 and finally to 512/512/512/1024. Incrementing the number of filters will result in a slow and slower learning.
Regarding the accuracy scores on the test set I saw them grow less and less doubling the number of filters so I guess the accuracy improvement obtained by only adding filters is limited.

3.3 4of6

Applying batch normalization after each convolutional layer normalizes the flowing data in the whole network speeding up convergence. It results in an great increment in accuracy from the previous 32% up to 43%.
Trying to use a wider fully connected layer (fc1 8096 neurons) brings a longer training time for a little improvement of around 1% in accuracy but a strong decrease for the loss probably because it overfits onto training set.

Dropout randomly sets neurons activations to zero during the training process to avoid overfitting and it increase slightly the accuracy but also the loss.
4of6_c was the best cnn configuration found among the proposed:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 128, kernel_size=5, stride=2, padding=0)
        self.conv1_bn = nn.BatchNorm2d(128)
        self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv2_bn = nn.BatchNorm2d(128)
        self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
        self.conv3_bn = nn.BatchNorm2d(128)
        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.conv_final_bn = nn.BatchNorm2d(256)
        self.fc1 = nn.Linear(64 * 4 * 4 * 4, 4096)
        self.dropout = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(4096, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1_bn(self.conv1(x)))
        x = F.relu(self.conv2_bn(self.conv2(x)))
        x = F.relu(self.conv3_bn(self.conv3(x)))
        x = F.relu(self.pool(self.conv_final_bn(self.conv_final(x))))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))

        x = self.dropout(x)
        x = self.fc2(x)
        return x
```



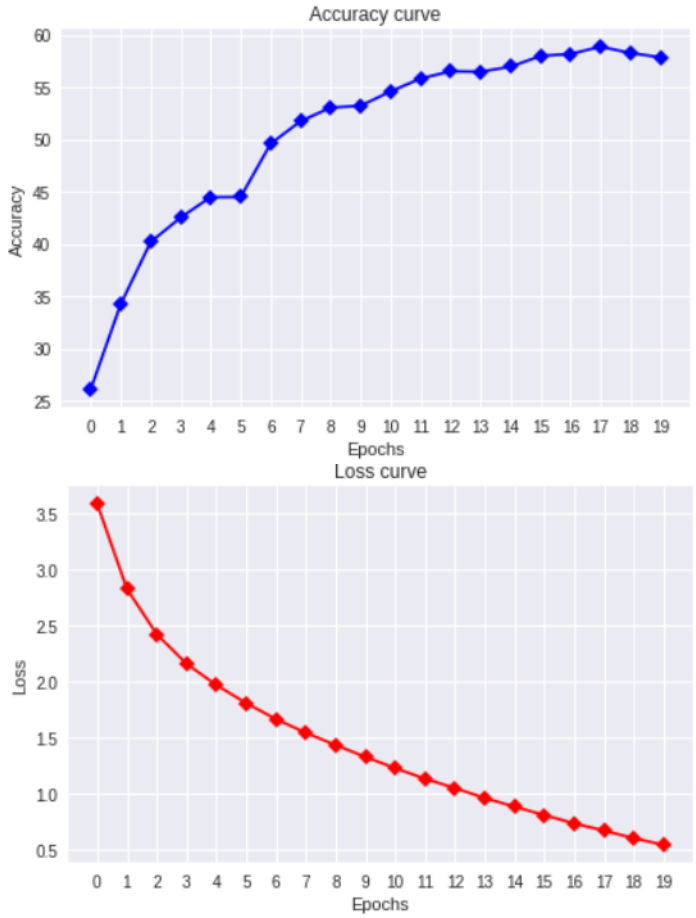
3.4 5of6

Whith a limited dataset, data augmentation techniques comes in handy to limit overfitting on a small training set and, by changing input during training, let the network generalize better.
In this case random horizontal flipping gave a better score and lower loss than random cropping probably because our dataset is composed of 32x32 rgb images and for the latter technique we have to first resize them to 40x40 and only then crop.

3.5 Best cnn results

By combying some of the techniques seen in the previous configurations while focusing on performance the best result I obtained, out of the ones tested, through applying batch normalization to every convolutional layer, dropout to limit overfitting, random horizontal flips as data augmentation, a number of filters of 512/512/512/1024 with a wider fully connected layer with 8096 neurons is 58% of accuracy with a low loss as 0.544 with the following components:

```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 512, kernel_size=5, stride=2, padding=0)
        self.conv1_bn = nn.BatchNorm2d(512)
        self.conv2 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=0)
        self.conv2_bn = nn.BatchNorm2d(512)
        self.conv3 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=0)
        self.conv3_bn = nn.BatchNorm2d(512)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv_final = nn.Conv2d(512, 1024, kernel_size=3, stride=1, padding=0)
        self.conv_final_bn = nn.BatchNorm2d(1024)
        self.fc1 = nn.Linear(64 * 4 * 4 * 4 * 4, 8096)
        self.dropout = nn.Dropout(p=0.5)
        self.fc2 = nn.Linear(8096, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1_bn(self.conv1(x)))
        x = F.relu(self.conv2_bn(self.conv2(x)))
        x = F.relu(self.conv3_bn(self.conv3(x)))
        x = F.relu(self.pool(self.conv_final_bn(self.conv_final(x))))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))

        x = self.dropout(x)
        x = self.fc2(x)
        return x
```



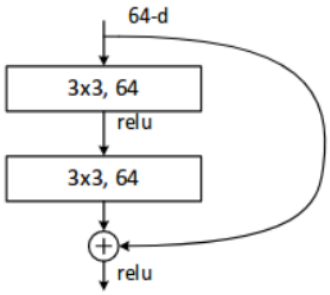
4 ResNet-18

ResNet architecture solves the problem of the vanishing gradient in ultra-deep networks, where accuracy gets saturated and then degrades rapidly, using residual functions.
The one I used was ResNet-18-layer taken from pytorch framework, pretrained on imagenet, with horizontal random flip as data augmentation.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

ResNet Architectures

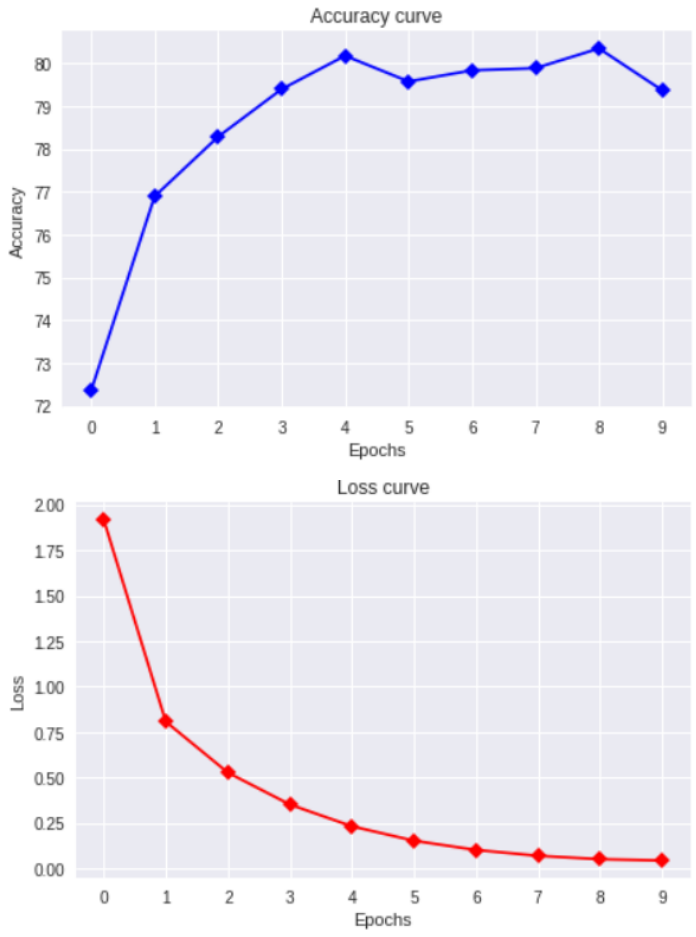
The ResNet-18 blocks are two layer deep



Even if its training is very slow, having it pretrained on the vast dataset of imagenet before finetunig it onto CIFAR100 for ten epochs resulted in really good scores.

Net configuration	Accuracy	Time	Loss
6of6	79%	01:01:01	0.044

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


dataiter = iter(trainloader)
```



ResNet gave the best results out of the tested networks not only because is a deeper cutting edge research network but also because it was pretrained on the vast imagenet database and only then finetuned on our CIFAR100.