POLITECNICO DI TORINO

Intelligence and Machine Learning

2019-2020

HOMEWORK2

Antonino Musmeci

HOMEWORK 2

The dataset used in this homework is Caltech101.
Caltech 101 is a data set of digital images contains a total of 9,146 images, split between 101 distinct object categories (faces, watches, ants, pianos, etc.) and a background category, about 40 to 800 images per category. Provided with the images are labels describing the outlines of each image.

You are going to train AlexNet, a Convolutional Neural Network for image classification. AlexNet contained eight layers; the first five were convolutional layers, some of them followed by max-pooling layers, and the last three were fully connected layers.

Data Preparation

We use the provided template to build a class to read the file train.txt and test.txt to divide the dataset in train set and test set and filter out the background class. Of the 9,146 images, 5784 are used as training set, and 2893 as test set. We used the train_test_split function from the sklearn.model_selection library. We need to divide the training set in training set and evaluation set. We use stratify parameter to equally distribute samples of each class in train and evaluation set

In [ ]:
```
import numpy as np
from sklearn.model_selection import train_test_split
indices = np.arange(len(train_dataset))
x1, x2, y1, y2, idxtrain, idxeval = train_test_split(
    train_dataset.imageroots, train_dataset.labels, indices, test_size=1/2,stratify=train_dataset.labels, random_state = 42)
eval_dataset = Subset(test_dataset, idxeval)
train_dataset = Subset(train_dataset, idxtrain)
```

Training from scratch

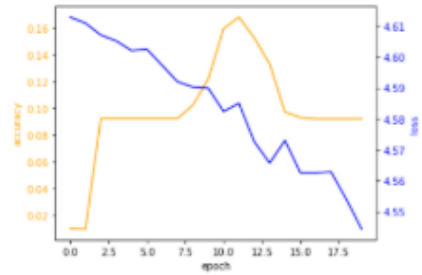
We use the training set to build the model and after each epoch we test each model on the validation set. Then we select the best best performing model and test it on the detaset. The current implementation using SGD with momentum for 30 epochs with an initial learning rate of 0.001 and a decaying policy after 20 epochs

In [ ]:
```
old_accuracy = 0

# Start iterating over the epochs
for epoch in range(NUM_EPOCHS):
    # Iterate over the dataset
    for images, labels in train_dataloader:
        # train the model....
    accuracy = testModel(net,eval_dataloader)
    if(accuracy > old_accuracy):
        torch.save(net.state_dict(), 'model.pt')
        old_accuracy = accuracy
    # Step the scheduler
    scheduler.step()
net.load_state_dict(torch.load('model.pt'))
testModel(net,test_dataloader)
```

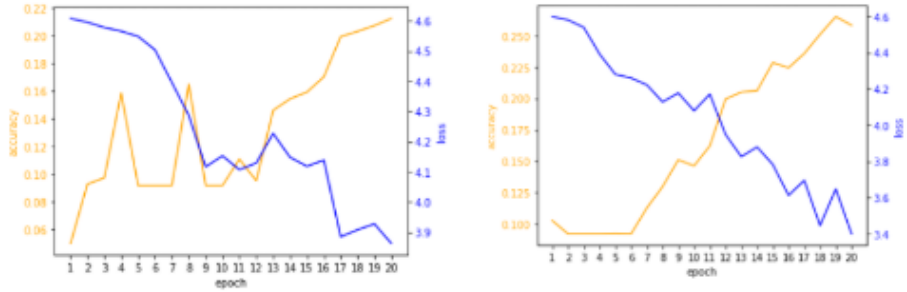
The first hyperparameter to optimize is the learning rate. We train and test our network over 20 epoch. With lr = 0.001 we see that the loss decrease too slowly. We try to increase learning rate

lr = 0.001

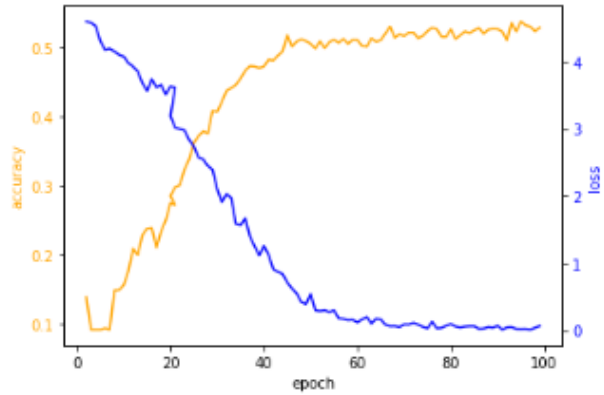


We try different value of learning rate. with lr = 0.005 we get an accuracy of 0.217, with lr = 0.01 we get an accuracy of 0.26

lr = 0.005 lr = 0.01

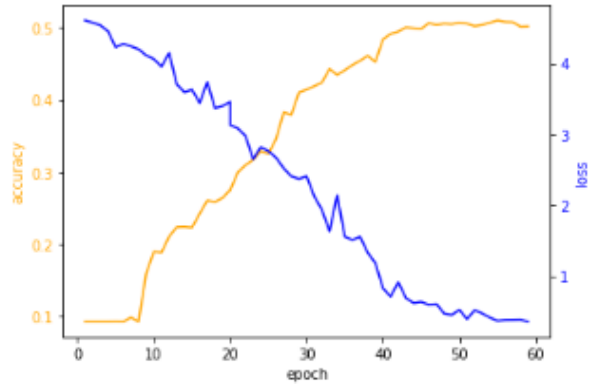


With the lr = 0.01 that give us the highest accuracy we try to optimize the number of epoch.As we can see from the plot the loss can apparently continue to decrease if we use more epoch. Now we train and test our network over 100 epoch and plot the accuracy and the loss to show in which epoch the model converges



We can see that after about 60 epoch the loss stop to decrease and accuracy increase very slowly

Using more epoch won't directly translate into a better accuracy, becouse the network could overfit on the training data. One way to try to prevent overfitting is to decrease learning rate after a certain number of epochs



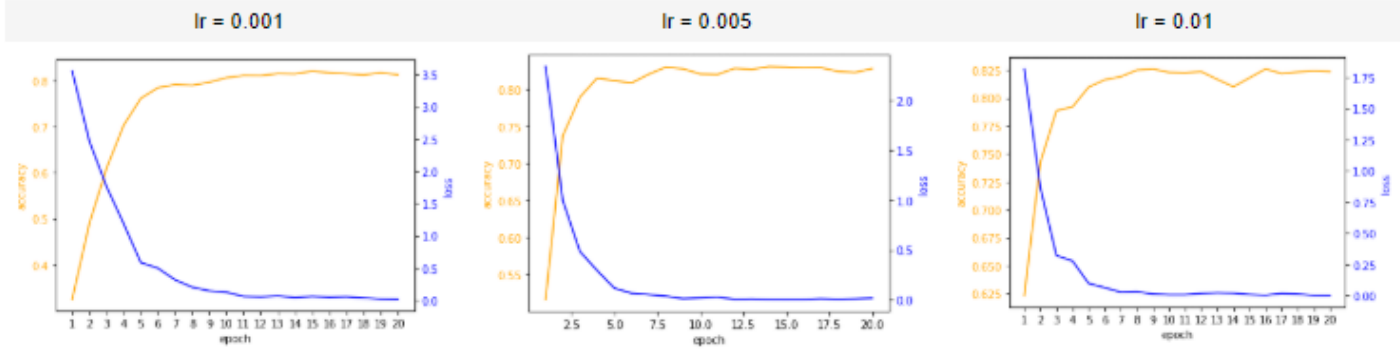
With deeplearning models to get better result we need more data but the dataset that we are using is too small. A solution to this problem is to use transfer learning. Insted of using random weight to inizialize the net we use a pretrained model We load AlexNet with weights trained on the ImageNet dataset a large dataset with milion of images.

To use this pretrained model on our dataset we need to change the Normalize function of Data Preprocessing to Normalize using ImageNet's mean and standard deviation.

Then we run again the training phase trying different hyperparameter

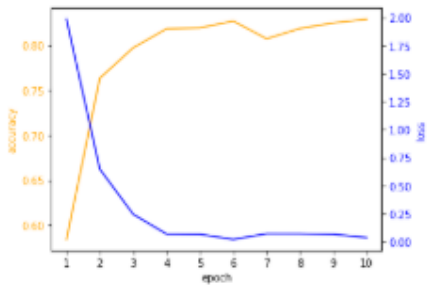
In [ ]:
```
...
transforms.Normalize(  mean=[0.485, 0.456, 0.406],
                       std=[0.229, 0.224, 0.225] )
...

net = torchvision.models.alexnet(pretrained=True)
```

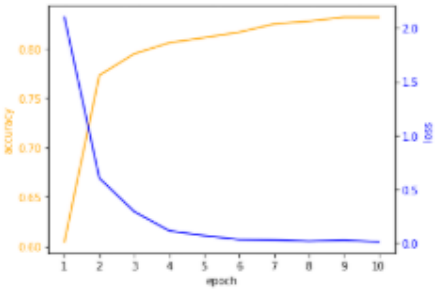


As we can see from the plot above with the pretrained model the loss decreases very quickly and, as expected, the accuracy obtained is higher then before. with lr = 0.01 the loss converge after about 10 epoch, they are enought to train the model

lr = 0.01, epochs = 10, accuracy = 81%

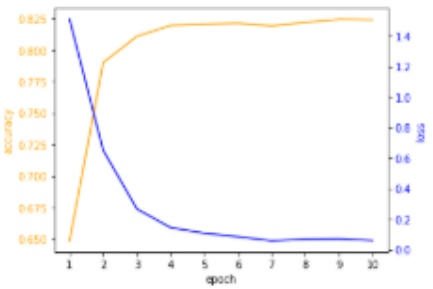


lr = 0.01, epochs = 10,
STEP_SIZE = 5, accuracy = 83%

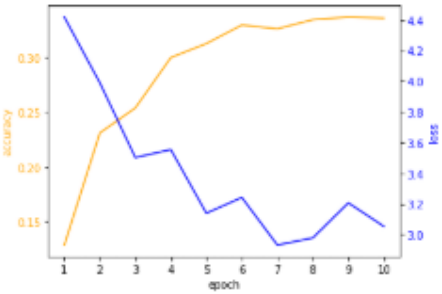


When we use transfer learning, we can also “freeze” part of the network and only train certain layers. We use the best hyperparameters obtained in the previous step: lr = 0.01, epochs = 10, STEP_SIZE = 5. First we train only the fully connected layers and then only the convolutional layers

Only full connected layer,
lr = 0.01, epochs = 10



Only convolutional layer,
lr = 0.01, epochs = 10, STEP_SIZE = 5



As we can see from the plot above if we freeze the convolution layers and only train the fully connected layers, the model works quite well and we get 81% accuracy on the test set. Instead if we only train the convolutional layer the model does not work very well even if we use a high value of the learning rate, the loss decreases slowly and we get only 37% accuracy on the test set. With more epochs we can obtain 50% accuracy but increasing the number of epochs we lose part of advantage of freezing the network. We can not get an higher accuracy probably because the first layers are less specific than the last

DATA AUGMENTATION

The more the data, the better our models will be, in principle. But every data collection process is associated with a cost. This cost can be in terms of dollars, human effort, computational resources and off course time consumed in the process. Therefore, we may need to augment existing data to increase the data size that we feed to our classifiers and to compensate for the cost involved in further data collection. Data augmentation is a strategy that enables practitioners to significantly increase the diversity of data available for training models, without actually collecting new data. Data augmentation techniques such as cropping, flipping, rotation are commonly used to train large neural networks. We perform data augmentation on our dataset using different sets of trasformation

In [ ]:
```
train_transform= transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.RandomHorizontalFlip(p=0.5),
                                     transforms.ToTensor()])


train_transform= transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.RandomHorizontalFlip(p=0.5),
                                     transforms.RandomVerticalFlip(p=0.5),
                                     torchvision.transforms.RandomApply([transforms.ColorJitter( brightness=0.5, contrast=0.5,
                                                                         saturation=0.5)], p=0.5),
                                     transforms.ToTensor()])


train_transform= transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.RandomRotation(20),
                                     torchvision.transforms.RandomApply([transforms.ColorJitter( brightness=0.5, contrast=0.5,
                                                                         saturation=0.5)], p=0.5),
                                     transforms.ToTensor()])

train_transform= transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.RandomPerspective(distortion_scale = 0.5),
                                     transforms.RandomVerticalFlip(p=0.5),
                                     transforms.ToTensor()])
```

transforms.RandomHorizontalFlip(p=0.5),	transforms.RandomHorizontalFlip(p=0.5), transforms.RandomVerticalFlip(p=0.5), transforms.ColorJitter	transforms.RandomRotation transforms.ColorJitter	transforms.RandomPerspective(distortion_scale=0.5), transforms.RandomVerticalFlip(p=0.5),

We apply this sets of preprocessing. All the experiments are conducted using the best parameters found before, without freezing components. With the first we get 84% accuracy, with the second 80% accuracy, with the third 80% accuracy, with the fourth 82% accuracy. We can see that only with the first set of trasformations we increase the result. the other trasformations don't help our model to perform better and the accuracy in all the cases are worse than the accuracy we obtain without transformations

(Extra) Beyond AlexNet

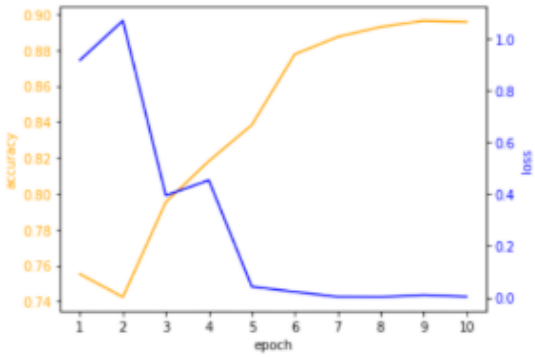
In this extra section we decide to experiment with other two different architecture VGG16 and ResNet152

The VGG16 architecture consists of twelve convolutional layers, some of which are followed by maximum pooling layers and then four fully-connected layers and finally a 1000-way softmax classifier.

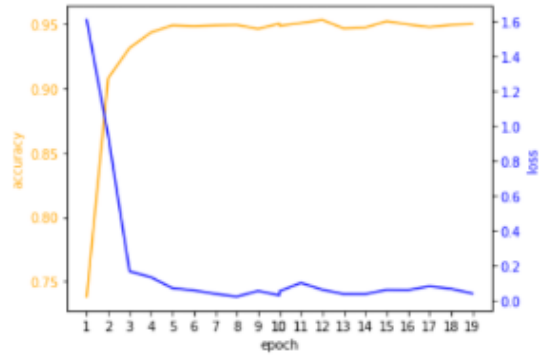
ResNet was the winner of ImageNet challenge in 2015. The fundamental breakthrough with ResNet was it allowed us to train extremely deep neural networks with 150+ layers successfully.

VGG and ResNet are more complex then AlexNet, they have lots of layers so they are slow to train and the network architecture has large numbers of weights so require more ram. A solution to handle this is to redure the batch size. we set batch size to 64 for VGG and to 16 for ResNet

VGG16,
lr = 0.01, epochs = 10,STEP_SIZE = 5,
BATCH_SIZE = 64



ResNet152,
lr = 0.001, epochs = 20, STEP_SIZE = 10,
BATCH_SIZE = 16



As expected using this more deeper networks we got a better result. With VGG we obtain 0.89 test accuracy and with ResNet we obtain 0.96 test accuracy