Homework II
Machine Learning and Deep Learning

Lidia Fantauzzo, s273117
Polytechnic University of Turin, April 27th, 2020

Training the AlexNet Convolutional Neural Network for image classification on Caltech-101.

Description of the Dataset

Caltech-101 is a dataset of digital images containing a total of 9.146 images, split between 101 distinct object categories, such as animal, faces, means of transportation, and a background category. Common categories such as faces tend to have a larger number of images than others. Images of oriented objects such as airplanes and motorcycles were mirrored to be left to right aligned and vertically oriented structures such as buildings were rotated to be off axis. This dataset have some advantages because for instance, almost all the images within each category are uniform in image size. Although this advantage can be a downside since the images are not always representative of practical inputs that the algorithm might later expect to see. Moreover the dataset represents only a small fraction of possible object categories and it is unbalanced because some categories are not represented as well as others, containing as few as 31 images.

Figure 1: Images from Caltech-101.

AlexNet

AlexNet consists 5 Convolutional Layers and 3 Fully Connected Layers. The first two Convolutional layers are followed by the Overlapping Max Pooling layers that are usually used to downsample the width and height of the tensors, keeping the depth same. The

third, fourth and fifth convolutional layers are connected directly. The fifth convolutional layer is followed by an Overlapping Max Pooling layer, the output of which goes into a series of two fully connected layers. The last fully connected layer has been changed with a new fully connected layer with 101 outputs to use it with the Caltech dataset. ReLU nonlinearity is applied after all the convolution and fully connected layers. The ReLU function is given by $f(x) = max(0, x)$. The implementation of this convolutional neural network is provided by pytorch. Another code model has been already provided me to implement it and to do the basic steps of training and testing.

Loss function

The loss function used in this work is the Cross Entropy Loss, whose expression for a single image is the following

$$loss(x, class) = -x[class] + log\left(\sum_{j=0}^{100} e^{x[j]}\right),$$

where x is a vector of dimension 101 containing the output of the last layer. The loss function measures how near to the correct label we are and so during the training phase, the optimizer helps the loss function to improve the prediction. After every minibatch of 256 images, the loss function in averaged and used to update the weights.

1 Creation of the dataset

A structure of code to implement the class for the creation of the dataset has been already provided. So the code consists in a class called Caltech in which the list of images of the dataset has to be uploaded from the folders. The first function in this class, called *___init___*, creates an instance of the class Caltech. The dataset was already splitted in train and test sets and two *.txt* files contained the names and the categories of each image. So the constructor can be called with a parameter that specifies which set has to be uploaded. A dictionary containing a unique number for each class is created with the *os* function *listdir*. Then the chosen file txt is read and all the strings identifying an image and its class are stored in a list; only the images belonging to the class 'BACKGROUND_Google' are ignored and discarded., in this way the number of classes becomes 101. Another parameter has to be passed to this function, which is the transform parameter that defines the transformations that have to be applied to the images before training or testing. The class Caltech contains also the *___getitem___* function, that is automatically called when the operator [] is applied to an element of Caltech class. This function has to return the image corresponding to the given index inside the brackets. In order to do this, the string, corresponding to the index, is read from the previous list and is splitted in name of the image and class of belonging. The function returns the image, loaded with the use of the *pil_loader* function that opens

the path and the label of the image, provided by the previous dictionary, as shown in the following code

```
def __getitem__(self, index):
        string=self.images[index].split("/")
        classe=string[0]
        image, label =
            pil_loader("./Caltech101/101_ObjectCategories/"+self.images[index]),
            self.Dict[classe]
        ...
        return image, label
```

At the end, a function to return the length, namely the number of elements, in the current dataset, is defined.

2 Training from scratch

In order to train the convolutional neural network it is necessary to have a validation methods to tune the hyper-parameters and choose the best model. In deep learning, it is often not sustainable to do k-fold cross validation, due to time required to complete trainings, so in this case is better having a single validation. So the division of the dataset in training and testing set was already done when the instance of the Caltech class was created, because as said, the 'test' or 'train' parameter was passed. Here also the transform parameter was passed: the images had to be resized, cropped to $224x224$ because torchvision's AlexNet needs this kind of image inputs, then they have been converted to a tensor and normalize with mean and standard deviation both equals to 0.5. Then the training set had to be splitted in validation and training sets. The size of the two sets has been required to be the same. So in order to split them, keeping the balanced the presence of each class in the two sets, the validation set was created taking the indexes that are multiple of two and the remaining are put in the training set.

```
train_dataset_pre = Caltech(DATA_DIR, split='train', transform=train_transform)
test_dataset = Caltech(DATA_DIR, split='test', transform=eval_transform)
train_indexes = [x for x in range(len(train_dataset_pre)) if not(x % 2) == 0]
val_indexes = [x for x in range(len(train_dataset_pre)) if x % 2 == 0]
train_dataset = Subset(train_dataset_pre, train_indexes)
val_dataset = Subset(train_dataset_pre, val_indexes)
```

This allowed to keep the classes balanced only because it was known that the images in the folder were sorted by class so alternately placing an element in one set and one in the other, it keeps them balanced. In the provided code, the name of the training class created had to be changed because otherwise it entered in conflict with the subset function, as shown in the code above. Then all the sets were divided in batches, which are the groups of samples that are propagated through the network one at the time for training. Moreover, the stochastic gradient descend with momentum, equals to 0.9,

was defined as optimizer, which updates the weights based on loss. The update of the weights were regularized by the weight decay, that penalizes large weights which were often symptom of overfitting. So the new weights would be updated every time using this algorithm $w_i \longleftarrow w_i - \eta\delta L/\delta w_i - \eta\lambda w_i$, where $\lambda/2$ is the weight decay. The scheduler used here was the step down scheduler that multiplies learning rate by gamma every 'step_size' epochs. The initial hyper-parameters that has been used to train and validate the CNN are the following

```
BATCH_SIZE = 256
LR = 1e-3
MOMENTUM = 0.9
WEIGHT_DECAY = 5e-5
NUM_EPOCHS = 30
STEP_SIZE = 20
GAMMA = 0.1
```

The CNN was trained on the training set NUM_EPOCHS time, and every 5 epochs it was validated on the validation set and the accuracy of the classification was calculated. The initial results obtained with these hyper-parameters were really disappointing: the accuracy on the validation was very low (the best value was 0.1578) and then it stabilized, and the loss decreased too slowly during the epochs. So this suggested to increase the learning rate, initially a little to avoid the risk to skip the minimum. So with all the parameter left as before and with a LR= 0.01, the results obtained were shown in Figure 2 . The accuracy improved so much because it reached 0.2756, but the loss did not decrease so much. The next step was to increase again the LR in order to make the loss

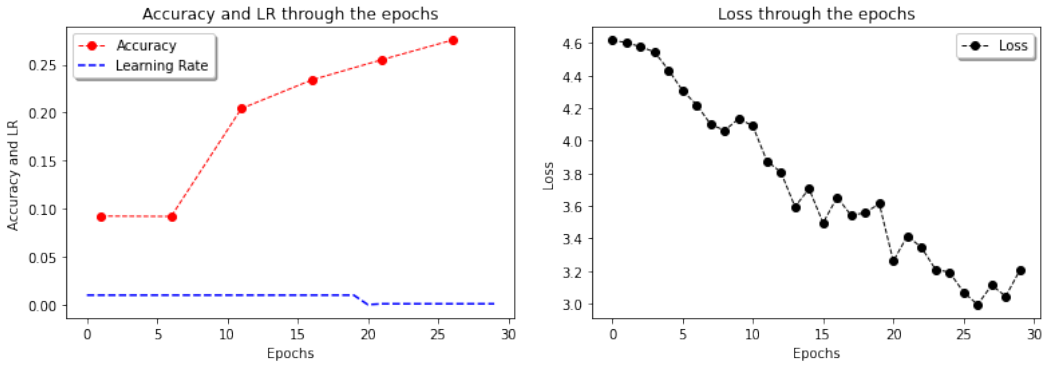

Figure 2: On the left: Accuracy and LR trends with respect to the epochs. On the right: Loss trend with respect to the epochs. The changed value si only $LR = 1e - 2$.

converge quickly, but since it was a risk to go so fast because the minimum ccould be missed and the loss could diverge, the STEP_SIZE had to be decreased, in this way the LR started with an high value but it shrank faster. So the LR was set to 0.1 and STEP_SIZE to 10 and the results obtained are shown in Figure 3. It can be seen that the accuracy was a little bit worse and the loss decreased as before, something had to

be improved. Since the accuracy have had a trend that hinted that it could increase, increasing the epochs, another attempt was to set NUM_EPOCHS to 50 and to try to set LR to 0.2. The results were worse in this case, the accuracy was low and stabilizes after 31 epochs, while the loss decreased and then started to increase a little. So it seemed that was better to leave LR= 0.1 and it seemed useless to use more than 30 epochs.

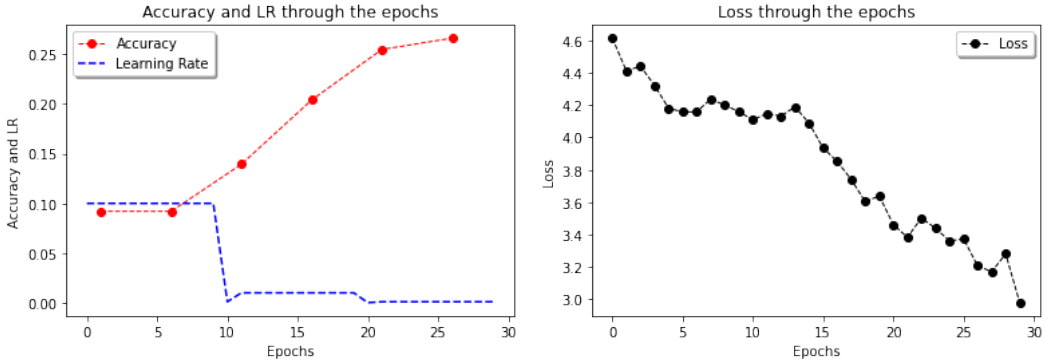

Figure 3: On the left: Accuracy and LR trends with respect to the epochs. On the right: Loss trend with respect to the epochs. The changed values are LR= $1e-1$ and STEP_SIZE = 10.

In order to further improve the model, it seemed good to decrease the LR more times, setting the STEP_SIZE to 7, but with a smaller factor, so setting GAMMA to 0.5 instead of 0.1. The results obtained are shown in Figure 4. The accuracy was better than before and the loss reached a low level. Other changes have been done, such as decreasing STEP_SIZE to 5 and set GAMMA to 0.7, but the best results got worst. So, after training the CNN with the best hyper-parameters, the convolution neural network obtained was tested on the testing set. The result was 0.4877 of accuracy. A summary of what has been done in shown in the table below.

LR	NUM_EPOCHS	STEP_SIZE	GAMMA	best VAL Accuracy	at epoch
0.001	30	20	0.1	0.1578	26
0.01	30	20	0.1	0.2756	26
0.1	30	10	0.1	0.2598	26
0.2	50	10	0.1	0.2062	31
0.1	30	7	0.5	0.4759	26
0.1	30	5	0.7	0.4476	26

3 Transfer Learning

Transfer learning is a technique that reuses parts of a previously trained model on a new network tasked for a different but similar problem. In this case, an AlexNet's pre-trained convolutional neural network is used. This model has already been trained on ImageNet

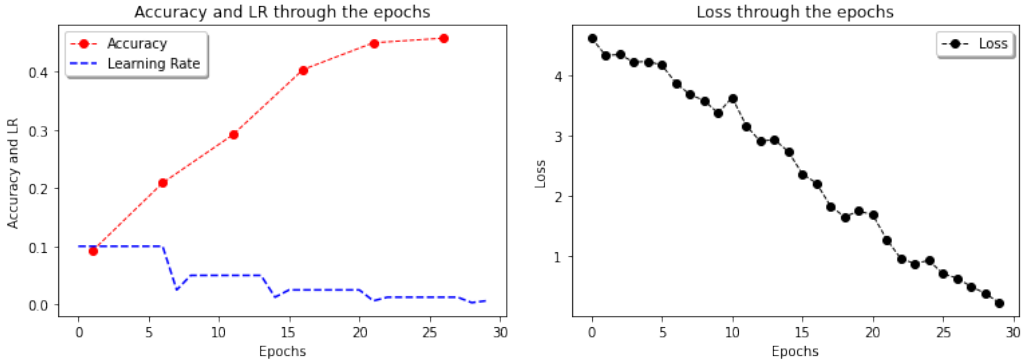

Figure 4: On the left: Accuracy and LR trends with respect to the epochs. On the right: Loss trend with respect to the epochs. The changed values are LR= $1e-1$, STEP_SIZE= 7 and GAMMA $= 0.5$.

with a millions of images from 1000 classes. The process to use a pre-trained model is listed below.

- Load the pre-trained weights from a network trained on a large dataset, here Alexnet on ImageNet.
- Freeze some layers depending on similarity of new task to original dataset; in this case Caltech is very similar to Imagenet.
- Replace the upper layers of the network with a custom classifier, because the number of outputs must be set equal to the number of classes. This has been already done since AlexNet has been used since the beginning of the assignment.
- Train only the classifier layers not frozen for the task thereby optimizing the model for smaller dataset.

The AlexNet CNN was loaded with the pre-trained weights and this required to change the normalization mean and standard deviation differently and set them respectively to $(0.485, 0.456, 0.406)$ and $(0.229, 0.224, 0.225)$, for the three dimensions. For this initial part of hyper-parameters tuning, all the parameters of the pre-trained AlexNet were optimized, none of them was frozen. All the hyper-parameters attempts are shown in table below.
The initial attempt has been done with a middle ground learning rate to see what happened. The results were already better then before but maybe increasing the LR they could be better. So after increasing the LR to 0.008, the accuracy improved. The best accuracy was reached before the thirtieth epoch, so the epochs were decreased and the LR was decreased every 10 epochs to make more precise weights updates. The results were a little bit worst. So next step was to increase a little bit the LR and the accuracy improved again. Then the attempt was to make the LR decrease slowly increasing GAMMA ad the result was better. Since the best accuracy was reached at the last epoch, the number of

LR	NUM_EPOCHS	STEP_SIZE	GAMMA	best VAL Accuracy	at epoch
0.001	30	20	0.5	0.8309	26
0.008	30	20	0.5	0.8499	21
0.008	20	10	0.3	0.8393	16
0.01	30	10	0.3	0.8558	26
0.01	30	10	0.5	0.8675	26
0.01	50	10	0.5	0.8624	31

epochs was increased to 50 but the best result was reached at the 31st epoch anyway, so it was not so useful increasing them. The accuracy obtained with the best hyper-parameters on the testing set was 0.8598.

The second part consisted in freezing some layers in order to retrain only the weights in the other layers. The optimal idea was to freeze the first convolutional layers because they learn the basic features of the images so they could be reused for different tasks. On the contrary, freezing the fully connected layers was not a great idea, since they are really specialize on the specific task of the CNN so is always better to retrained them. So both approaches have been tried and the results are shown in the table below. The parameters used for these training were the hyper-parameters that gave the best performance in the previous tuning, so LR= 0.01,NUM_EPOCHS= 30, STEP_SIZE= 10 and GAMMA= 0.5.

Frozen layer	best VAL Accuracy	TEST Accuracy	at epoch
Convolutional	0.8592	0.8613	26
Fully Connected	0.5211	0.5237	26

As said freezing the fully connected layers instead of the convolutional layers, worse results were obtained.

4 Data Augmentation

It is really important to have a large dataset to train the CNN, so since in this case the Caltech dataset is small, data augmentation is an approach that can be useful. Data augmentation virtually enlarge the number of images of the dataset, because at each epoch the dataloader applies a fresh set of random operations "on the fly" with some probability, in such a way that the label of the images remains the same. So the size of the dataset remains the same but it is like the neural network is trained on an bigger one. The transformations used were:

- RandomGrayscale(0.5): converts the image to grayscale with a probability of 0.5;
- RandomHorizontalFlip(0.5): horizontally flippes the image with a probability of 0.5;

- RandomRotation(degrees=45, center=(0,0)): rotates the image with an angle of 45° with respect to the center. If the center is omitted, the center of rotation is the center of the image otherwise the coordinates that are provided. The origin of the axis is the upper left corner;
- RandomCrop(size=100): crops the image with a square crop of 100x100;
- RandomErasing(p=0.5, scale=(0.02, 0.33)): randomly selects a rectangle region in an image and erases its pixels. Range is the range of proportion of erased area against input image.
- ColorJitter(brightness, contrast, saturation): randomly changes the brightness, contrast and saturation of an image. Different values were tested.

The results obtained are shown in the table; all of them were done with transfer learning with the same best hyper-parameters as before. Here some augmentation policies were

Transformation	best VAL Accuracy	TEST Accuracy
HorizontalFlip	0.8485	0.8482
RandomRotation 45°, (0,0)	0.7538	0.8019
RandomRotation 45°, ceter image	0.7804	0.8130
RandomGrayscale	0.8475	0.8486
RandomCrop	0.7690	0.7700
ColorJitter(0.5,0.2,0.1)	0.8471	0.8503
ColorJitter(0.3,0.5,0.3)	0.8483	0.8558

worsening accuracy, maybe because the training and test sets are very similar to each others.

5 Beyond AlexNet: ResNet

In neural network architectures having a very deep network with a lot of hidden layers would increase the accuracy but unfortunately having a lot of hidden layers paves way to the vanishing / exploding gradients problem which actually makes the neural network unusable. Vanishing gradients is the phenomenon where the weights are updated so slowly that they stop updating after a while and exploding gradients is the phenomenon where large updates are made to the neural network weights. The Residual Network (ResNet) solves this problem and allows deeper networks. The new construction is the identity connection between the layers, namely there is a connection between the input of the layer and the ReLu non-linearity. This new "path" can be taken by the gradients during the back-propagation so they don't have to encounter any weight layer, hence, there won't be any change in the value of computed gradients. Two ResNet architectures were tested, one with 18 layers and another with 34 layers. The hyper-parameters were tuned and the best found were LR= 0.03,NUM_EPOCHS= 16, STEP_SIZE= 8 and

GAMMA= 0.7. For ResNet3 the number of batches had to be reduced because larger models consume GPU memory, but they are more accurate. As expected, the more layers

Model	BATCH_SIZE	VAL Accuracy	TEST Accuracy
ResNet18	256	0.9362	0.9340
ResNet34	128	0.9481	0.9423

are used, the more precise the network.