POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Corso di Laurea Magistrale in Ingegneria Informatica

Machine Learning and Artificial Intelligence

Homework 2: Deep Learning

Student:

Pietro Basci

266004

ACADEMIC YEAR 2019/2020

Introduction

The purpose of this homework is to train a Convolutional Neural Network for image classification, tune some hyperparameters and evaluate the performance changes. Specifically, it will be considered firstly the Alexnet model and then some other types of CNNs such as VGG and ResNet. Moreover, some technique such as Transfer Learning and Data Augmentation will be used in order to try to improve the model performance. The dataset considered is the Caltech-101 dataset, which is composed by about 9000 pictures of objects belonging to 101 categories with about 40 to 800 images per category. The training of models has been performed on Google Colab that provides a free Jupyter notebook environment running entirely in the cloud and powerful computing resources. In deep learning powerful GPUs dramatically reduces the time required by the training phase and Google Colab offers Nvidia Tesla K80 GPU having 2496 CUDA cores and 12GB of VRAM.

Data Preparation

The first operations to do are load the data and create the training and test sets using the splits provided into the *train.txt* and *test.txt* files. To address that, a specific dataset class 'Caltech' has been defined. Implementing a dataset class requires to define three methods: *__init__* which implements all the logic to create those sets, *__getitem__* which define a way to access elements by index and *__len__* which returns the length of the dataset.

```
class Caltech(VisionDataset):

    def __init__(self, root, split='train', transform=None, target_transform=None):
        super(Caltech, self).__init__(root, transform=transform, target_transform=target_transform)

        # List of all classes, excluding 'BACKGROUND_Google' class
        self.classes = [d for d in os.listdir(root)
                        if os.path.isdir(os.path.join(root, d))
                        and d != 'BACKGROUND_Google']
        self.classes.sort()

        # Map classes to indexes
        self.class_to_idx = {self.classes[i]: i for i in range(len(self.classes))}

        # Map label to image name, read from txt split file (excluding 'BACKGROUND_Google' class)
        lab_to_image = pd.read_csv('Homework2-Caltech101/{}.txt'.format(split),
                                   sep='/', header=None, names=['label', 'image'])
        lab_to_image = lab_to_image.loc[lab_to_image['label'] != 'BACKGROUND_Google']

        # List of pair (image, label)
        self.samples = []

        for i, row in lab_to_image.iterrows():
            path = os.path.join(root, row['label'], row['image'])
            if os.path.isfile(path):
                #item = (path, self.class_to_idx[row['label']]) # Use only path to save ram
                item = (pil_loader(path), self.class_to_idx[row['label']])  # Use image to be faster
                self.samples.append(item)
```

In the *__init__* method, firstly the list of all classes is retrieved, to the exclusion of the 'BACKGROUND_Google' class that has to be filtered out. Than each class is associated to an integer hereinafter used as label. After that, from the txt file containing the items of the split, the label (first part of the path) and the name of the file (last part of the path) are extracted and items that belong to the 'BACKGROUND_Google' class are filtered out. Finally the pairs (image, label) are stored in the *self.samples* variable: if the hardware allows it, storing data in the main memory allows to achieve better performance at runtime, otherwise it is possible to save the pairs (image path, label) as in the commented out code.

```
def __getitem__(self, index):

    image, label= self.samples[index] # if image is used in __init__
    #path, label= self.samples[index] # if path is used in __init__
    #image = pil_loader(path)

    # Applies preprocessing when accessing the image
    if self.transform is not None:
        image = self.transform(image)

    return image, label
```

The *__getitem__* method access an element through its index, than applies preprocessing transforms and return the pair (image, label). If in the *self.samples* variable are saved the pairs (image path, label), it is needed to load the image as in the commented out code.

```
def __len__(self):
    return len(self.samples)
```

The *__len__* method returns the length of the *self.samples* list which contains all the elements of the considered dataset.

Training from scratch

For the training phase, we need a validation set on which evaluate our model for hyperparameters tuning and model selection. In deep learning, usually the K-Fold Cross Validation technique cannot be used due to long time required by the training phase. So, in practice, it is common to have a single validation set to perform tests. The split has been performed so that both the new sets are the half of the initial set and have half samples each. Since data has been read in order and no shuffle has been performed yet, elements of each class are still grouped by class inside the list. So dividing even and odd samples between the two sets ensures to both splits about half of images of each class.

```
DATA_DIR = 'Homework2-Caltech101/101_ObjectCategories'

# Prepare Pytorch train/test Datasets
train_dataset = Caltech(root=DATA_DIR, split='train', transform=train_transform)
test_dataset = Caltech(root=DATA_DIR, split='test', transform=eval_transform)

# Considering even and odd ensure to both splits about half of images of each class
train_indexes = [idx for idx in range(len(train_dataset)) if idx % 2]
val_indexes = [idx for idx in range(len(train_dataset)) if not idx % 2]

val_dataset = Subset(train_dataset, val_indexes)
train_dataset = Subset(train_dataset, train_indexes)
```

The resulting sizes of the three sets are the following:

```
Train Dataset: 2892
Validation Dataset: 2892
Test Dataset: 2893
```

A series of preprocessing transforms are defined and chained together using *Compose*. Initially, the following image transformations are considered:

```
# Define transforms for training phase
train_transform = transforms.Compose([transforms.Resize(256),      # Resizes short size of the PIL image to 256
                                      transforms.CenterCrop(224),  # Crops a central square patch of the image
                                                                   # 224 because torchvision's AlexNet needs a 224x224 input
                                      transforms.ToTensor(), # Turn PIL Image to torch.Tensor
                                      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # Normalizes tensor with mean and standard deviation
])
# Define transforms for the evaluation phase
eval_transform = transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

After that, Dataloaders are defined.

```
# Dataloaders iterate over pytorch datasets and transparently provide useful functions (e.g. parallelization and shuffling)
train_dataloader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, num_workers=4, drop_last=True)
val_dataloader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)
test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False, num_workers=4)
```

The first CNN considered is AlexNet. It is composed by eight layers: the first five are convolutional and the other three are fully-connected.

After the model defining, since the AlexNet model has been developed for the ImageNet dataset which is composed by 1000 classes, the last layer of the fully connected layers has to be readapted to the number of the Caltech-101 classes.

```
net = alexnet() # Loading AlexNet model

# AlexNet has 1000 output neurons, corresponding to the 1000 ImageNet's classes
# We need 101 outputs for Caltech-101
net.classifier[6] = nn.Linear(4096, NUM_CLASSES) # nn.Linear in pytorch is a fully connected layer
                                                 # The convolutional layer is nn.Conv2d

# We just changed the last layer of AlexNet with a new fully connected layer with 101 outputs
```

Then, the loss function, the optimizer and the scheduler are defined. In this phase, also the parameter to optimize are chosen: in this case, all the parameters of AlexNet will be optimized.

```
# Define loss function
criterion = nn.CrossEntropyLoss() # for classification, we use Cross Entropy

# Choose parameters to optimize
# To access a different set of parameters, you have to access submodules of AlexNet
# (nn.Module objects, like AlexNet, implement the Composite Pattern)
# e.g.: parameters of the fully connected layers: net.classifier.parameters()
# e.g.: parameters of the convolutional layers: net.features.parameters()
parameters_to_optimize = net.parameters() # In this case we optimize over all the parameters of AlexNet

# Define optimizer
# An optimizer updates the weights based on loss
# We use SGD with momentum
optimizer = optim.SGD(parameters_to_optimize, lr=LR, momentum=MOMENTUM, weight_decay=WEIGHT_DECAY)

# Define scheduler
# A scheduler dynamically changes learning rate
# The most common schedule is the step(-down), which multiplies learning rate by gamma every STEP_SIZE epochs
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=STEP_SIZE, gamma=GAMMA)
```

In the training phase, for each epoch, we iterate over all the dataset using more than one image at a time. In particular, at each iteration, the model is used to make prediction on a batch of n images (where n is defined by the *batch_size* parameter), then the average loss for those n samples is computed and finally weights are updated according to accumulated gradients.

```
current_step = 0
# Start iterating over the epochs
for epoch in range(NUM_EPOCHS):
  print('Starting epoch {}/{}, LR = {}'.format(epoch+1, NUM_EPOCHS, scheduler.get_lr()))

  running_loss = 0.0
  # Iterate over the dataset
  for images, labels in train_dataloader:
    # Bring data over the device of choice
    images = images.to(DEVICE)
    labels = labels.to(DEVICE)

    net.train() # Sets module in training mode

    # PyTorch, by default, accumulates gradients after each backward pass
    # We need to manually set the gradients to zero before starting a new iteration
    optimizer.zero_grad() # Zero-ing the gradients

    # Forward pass to the network
    outputs = net(images)

    # Compute loss based on output and ground truth
    loss = criterion(outputs, labels)
    running_loss += loss.item()

    # Log loss
    if current_step % LOG_FREQUENCY == 0:
      print('Step {}, Loss {}'.format(current_step, loss.item()))

    # Compute gradients for each layer and update weights
    loss.backward()  # backward pass: computes gradients
    optimizer.step() # update weights based on accumulated gradients

    current_step += 1

  train_loss.append(running_loss/len(train_dataloader))  # compute average loss
  train_acc.append(get_accuracy(net, train_dataloader)) # compute accuracy on train set
```

For each epoch, train loss and accuracy are traced to make a compare with validation loss and accuracy.

At the end of each epoch, a test on the validation set has been performed in order to select the best performing model which will be used for the final evaluation on the test set.

```
# Evaluate the model on the validation set
net = net.to(DEVICE) # this will bring the network to GPU if DEVICE is cuda
net.train(False) # Set Network to evaluation mode

running_loss = 0.0
running_corrects = 0
for images, labels in val_dataloader:
  images = images.to(DEVICE)
  labels = labels.to(DEVICE)

  # Forward Pass
  outputs = net(images)

  # Compute loss based on output and ground truth
  loss = criterion(outputs, labels)
  running_loss += loss.item()

  # Get predictions
  _, preds = torch.max(outputs.data, 1)

  # Update Corrects
  running_corrects += torch.sum(preds == labels.data).data.item()

# Calculate Accuracy
accuracy = running_corrects / float(len(val_dataset))

print('Test Accuracy on Validation: {}'.format(accuracy))

# Update the best model
if accuracy > best_accuracy:
  best_accuracy = accuracy
  best_net = copy.deepcopy(net)

val_acc.append(accuracy)
val_loss.append(running_loss/len(val_dataloader))

# Step the scheduler
scheduler.step()
```

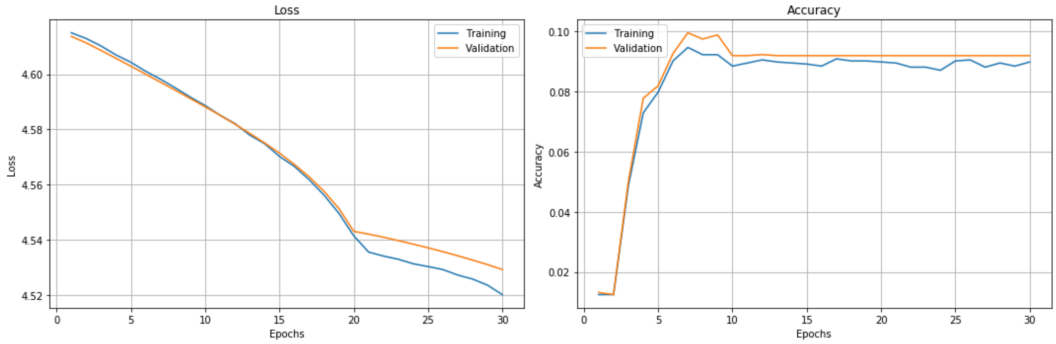
The model has been trained multiple times using different sets of hyperparameters. Specifically, different values of these hyperparameters has been considered:

- *learning_rate* which defines the size of each parameter update. A small learning rate leads the network parameters not to change very much in each iteration. As a result, the network could take a long time to train (model converges too slowly). An high learning rate instead leads the network parameters to change a lot in each iteration (the model converges fast but accuracy is sub-optimal);
- *batch_size* which defines the dimension of the batch of images that is used in each iteration (in each iteration, each weight is updated once);
- *num_ephocs* that defines the number of time all training data is used to update the weights (in each epoch, each weight is updated multiple times, i.e. the number of batches);
- *step_size* that defines the number of epochs after which the learning rate decreases.

An appropriate learning rate depends on the batch size and the stage of training: a large batch size allows a larger learning rate, while a smaller batch size requires a smaller learning rate. It is also useful to reduce the learning rate as training progresses so that the update of weights will be higher in the early stages of training and lower in the later stages.

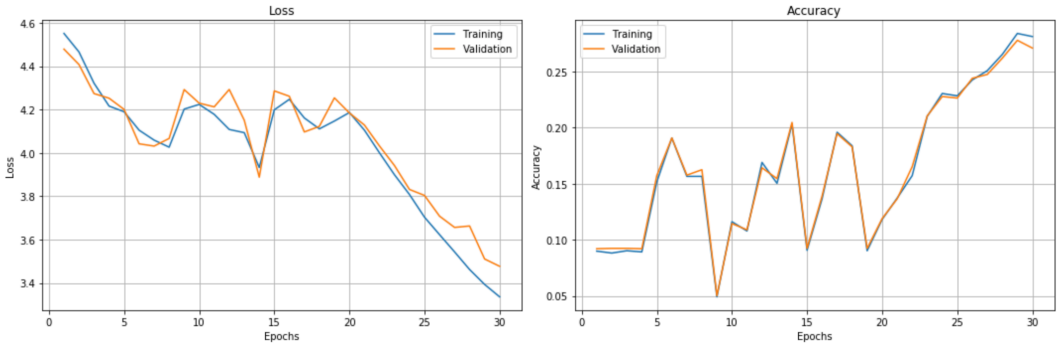
The results for each set of hyperparameters are reported in the following page.

Training using:
learning_rate = 1e-3; batch_size = 256; num_epochs=30; step_size=20.



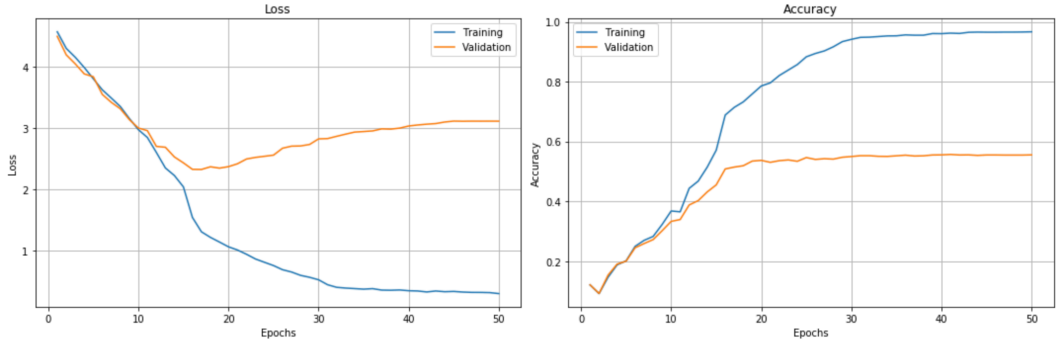
Due to low value of learning rate loss decreases very slow but with a smooth trend. The network trains very slow, indeed accuracy after 30 epochs is still under 0.1.

Training using:
learning_rate = 1e-1; batch_size = 256; num_epochs=30; step_size=20.



Increasing too much the learning rate lead the curves to be very noisy. This is due to the fact that the size of the network weights updates is too high.

Training using:
learning_rate = 1e-2; batch_size = 64; num_epochs=50; step_size=15.



As shown in the chart, using an higher learning rate with a lower batch that leads to an higher number of updates of the network weights (more iterations), allows to achieve a better accuracy (0.56 on validation set). However after the 18th epoch the model starts to overfit the training data as demonstrated by the huge gap between training and validation accuracies and also losses curves with the validation loss that actually begins to increase after the 20th epoch.

Finally, the best model found, has been tested on the test set.

```
best_net = best_net.to(DEVICE) # this will bring the network to GPU if DEVICE is cuda
best_net.train(False) # Set Network to evaluation mode

running_corrects = 0
for images, labels in tqdm(test_dataloader):
  images = images.to(DEVICE)
  labels = labels.to(DEVICE)

  # Forward Pass
  outputs = best_net(images)

  # Get predictions
  _, preds = torch.max(outputs.data, 1)

  # Update Corrects
  running_corrects += torch.sum(preds == labels.data).data.item()

# Calculate Accuracy
accuracy = running_corrects / float(len(test_dataset))

print('Test Accuracy: {}'.format(accuracy))
```

Output:

```
100%|██████████| 46/46 [00:05<00:00,  8.24it/s]Test Accuracy: 0.5468371932250259
Best Accuracy on Validation: 0.5567081604426003
```

Transfer Learning

In order to achieve very good performance, a Deep Convolutional Neural Network needs very large data-set to train good features. Since the Caltech-101 dataset is very small, to obtain better result the Transfer Learning technique has been considered. Transfer Learning is a technique which focuses on storing the knowledge gained while solving one problem and applying it to a different but related problem. The idea is to start from a CNN pre-trained on a large dataset and fine-tune the model on the new dataset. In this way, weights learned by training on a large related dataset are used as a starting point for training on the small dataset. So, the pre-trained AlexNet on ImageNet dataset has been fine-tuned on the new dataset and evaluated.

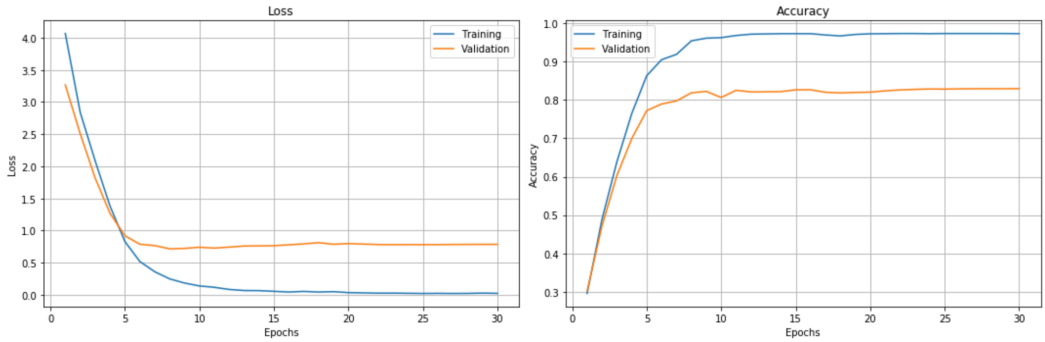
```
net = alexnet(pretrained=True) # Loading AlexNet model (pretrained on ImageNet)
net.classifier[6] = nn.Linear(4096, NUM_CLASSES)  # Changed the last layer of AlexNet
```

The Normalize function of the preprocessing data has been modified to normalize using ImageNet's mean and standard deviation.

```
# Define transforms for training phase
train_transform = transforms.Compose([transforms.Resize(256),      # Resizes short size of the PIL image to 256
                                      transforms.CenterCrop(224),  # Crops a central square patch of the image
                                                                   # 224 because torchvision's AlexNet needs a 224x224 input!
                                                                   # Remember this when applying different transformations, otherwise you get an error
                                      transforms.ToTensor(), # Turn PIL Image to torch.Tensor
                                      transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)) # Normalizes tensor with mean and standard deviat
])
# Define transforms for the evaluation phase
eval_transform = transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
])
```

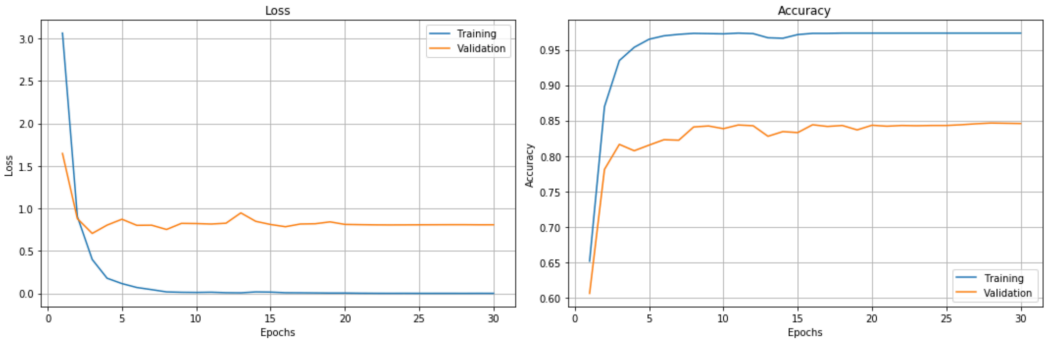
The results obtained using different sets of hyperparameters are reported below.

Training using:
learning_rate = 1e-3; batch_size = 256; num_epochs=30; step_size=20.



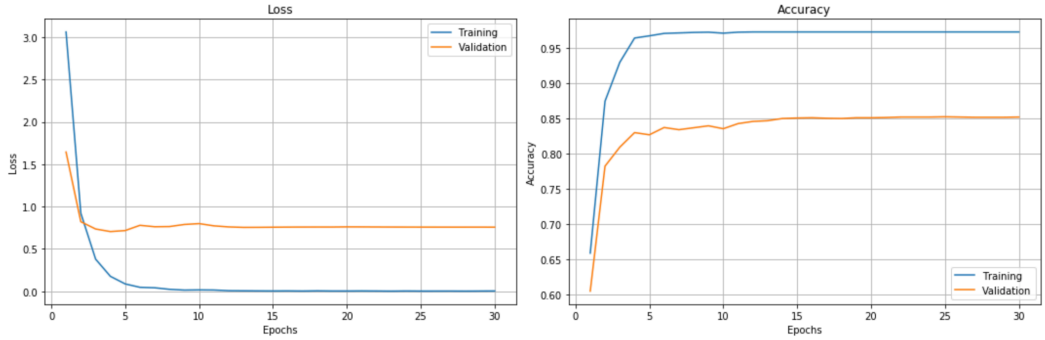
Comparing the results to the ones obtained in the previous experiment, it is clear that, thanks to the fact that the model is already trained on images, the accuracy start at around 30% after the first epoch and rapidly grow reaching about the max in less than 10 epochs. Similarly the loss function sharply decrease within the first 5 epochs and than the two curves starts to assume different values: the validation loss remains around 0.8 while the train loss continues to decrease until it reaches values near to 0. This means that from this point the model starts to overfit the trining data and there will be no performance increase on the validation data as confirmed also by the accuracies curves.

Training using:
learning_rate = 1e-2; batch_size = 256; num_epochs=30; step_size=20.



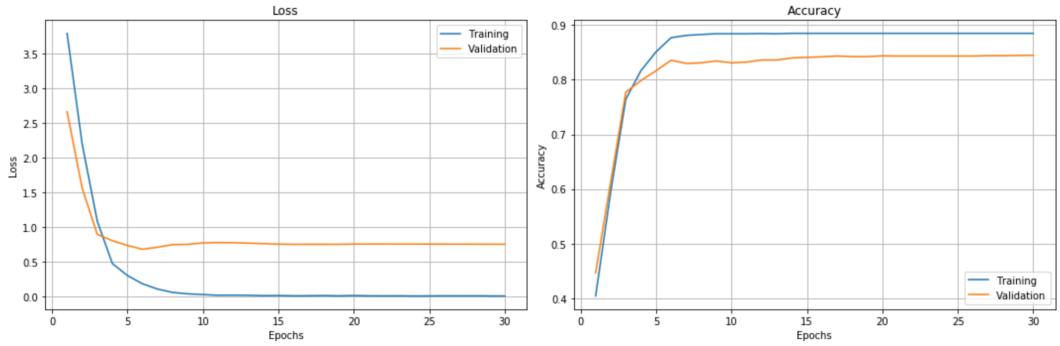
Incrementing the learning rate, we further reduce the rise time of the accuracy which also starts from an higher value already after the first epoch (more than 0.60). Due to the higher value of the learning rate both curves are quite wiggled before the 20th epoch. After that the learning rate decreases and the curves become smoother.

Training using:
learning_rate = 1e-2; batch_size = 256; num_epochs=30; step_size=10.



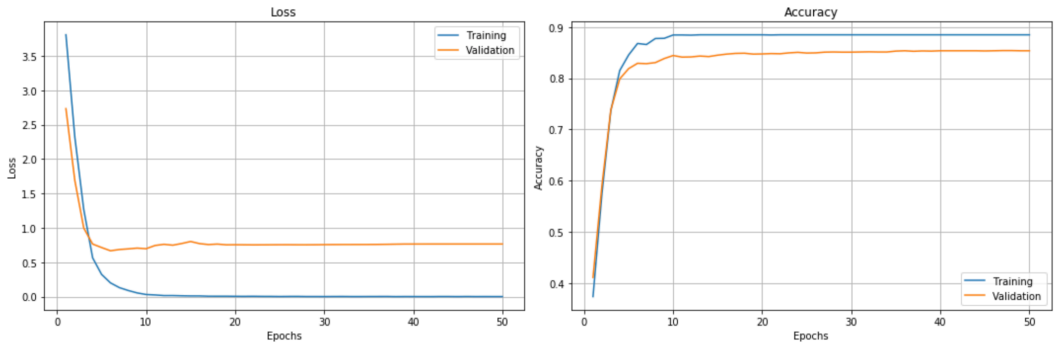
Decreasing the step size to 10, allows to achieve the best accuracy on the validation (near to 0.86) among all the tests performed on different sets of hyperparameters.

Training using:
learning_rate = 1e-2; batch_size = 512; num_epochs=30; step_size=10.



Incrementing the batch size, the gap between the two accuracy is reduced. The accuracy on the training set remains under 0.9 while the accuracy on the validation is steel the same (around 0.85).

Training using:
learning_rate = 1e-2; batch_size = 512; num_epochs=50; step_size=20.



Increasing the number of epochs to 50 there are no significantly change on both losses and accuracies, so 30 epoch can be considered enough.

Then, the best model found has been tested on the test data and the accuracy reached about 0.85.

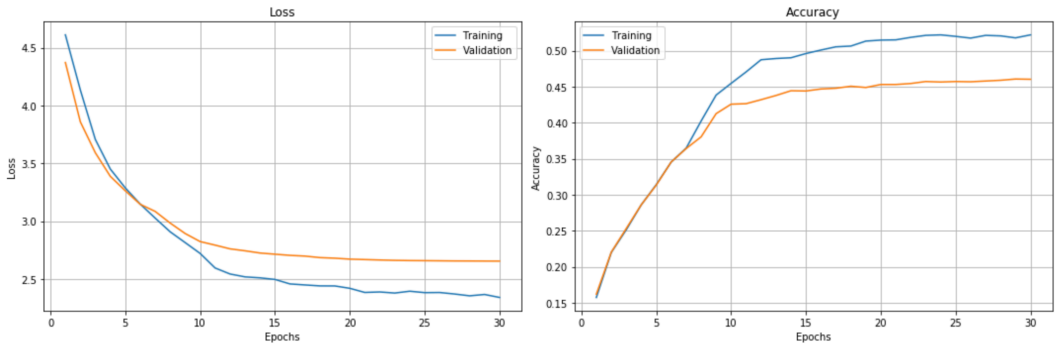
```
100%|██████████| 6/6 [00:05<00:00,  1.11it/s]Test Accuracy: 0.8454891116488075
Best Accuracy on Validation: 0.8540802213001383
```

Now, using the best hyperparameters obtained in the previous step, the pre-trained AlexNet has been fine-tuned “freezing” part of the network and so training only certain layers. This can reduce the time needed by training phase but could worse accuracy. In the first experiment, only the convolutional layers parameters have been optimized (fully connected layers “frozen”).

```
# Choose parameters to optimize
# To access a different set of parameters, you have to access submodules of AlexNet
# (nn.Module objects, like AlexNet, implement the Composite Pattern)
# e.g.: parameters of the fully connected layers: net.classifier.parameters()
# e.g.: parameters of the convolutional layers: net.features.parameters()
parameters_to_optimize = net.features.parameters() # In this case we optimize over the convolutional layers parameters of AlexNet
#parameters_to_optimize = net.classifier.parameters() # In this case we optimize over the fully connected layers parameters of AlexNet
#parameters_to_optimize = net.parameters() # In this case we optimize over all the parameters of AlexNet
```

The results obtained are reported below.

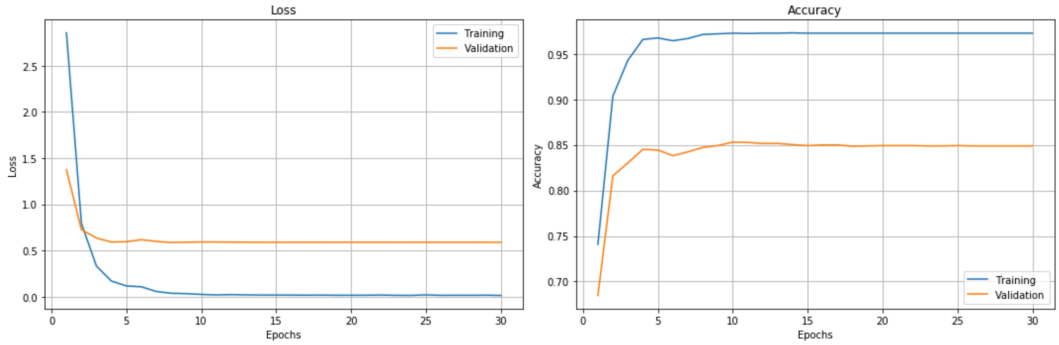
Training using:
learning_rate = 1e-2; batch_size = 256; num_epochs=30; step_size=10.



In the second experiment, only the fully connected layers parameters have been optimized (convolutional layers “frozen”).

```
# Choose parameters to optimize
# To access a different set of parameters, you have to access submodules of AlexNet
# (nn.Module objects, like AlexNet, implement the Composite Pattern)
# e.g.: parameters of the fully connected layers: net.classifier.parameters()
# e.g.: parameters of the convolutional layers: net.features.parameters()
#parameters_to_optimize = net.features.parameters() # In this case we optimize over the convolutional layers parameters of AlexNet
parameters_to_optimize = net.classifier.parameters() # In this case we optimize over the fully connected layers parameters of AlexNet
#parameters_to_optimize = net.parameters() # In this case we optimize over all the parameters of AlexNet
```

Training using:
learning_rate = 1e-2; batch_size = 256; num_epochs=30; step_size=10.



As shown by the charts, in the first case we have an important reduction of the performances with respect to the previous training where we optimized over all the parameter of the model. This can be explained with the fact that in this case we are only optimizing the convolutional layers parameters that is the part of the CNN that is already good to extract features as it has been pre-trained on huge quantity of images. While the fully connected layers parameters that should be readapted to the problem are not optimized. Indeed, in second case, despite we optimize only the fully connected layers parameters we obtain performance equivalent to the complete training.

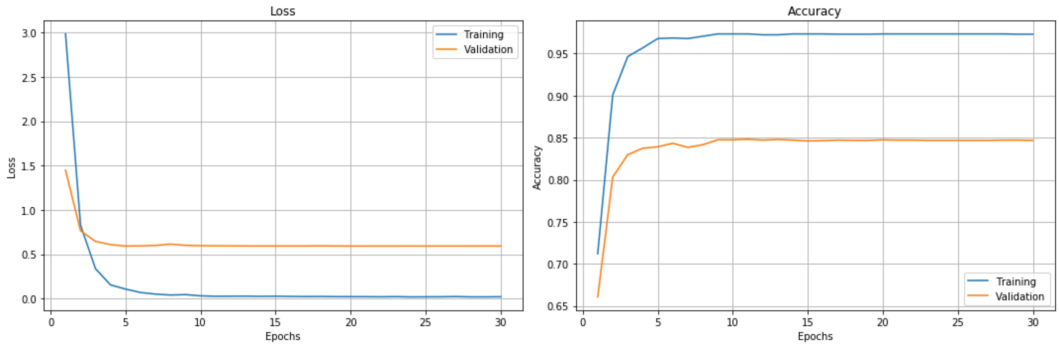
Data Augmentation

As said above, deep learning needs very high amount of data to achieve good performance. So, it is common to artificially increase the dataset size by applying label-preserving transformations to images. Data augmentation prevents overfitting by increasing cardinality of data. Some other transformations have been applied to the data such as RandomHorizontalFlip, RandomRotation, RandomPerspective and ColorJitter.

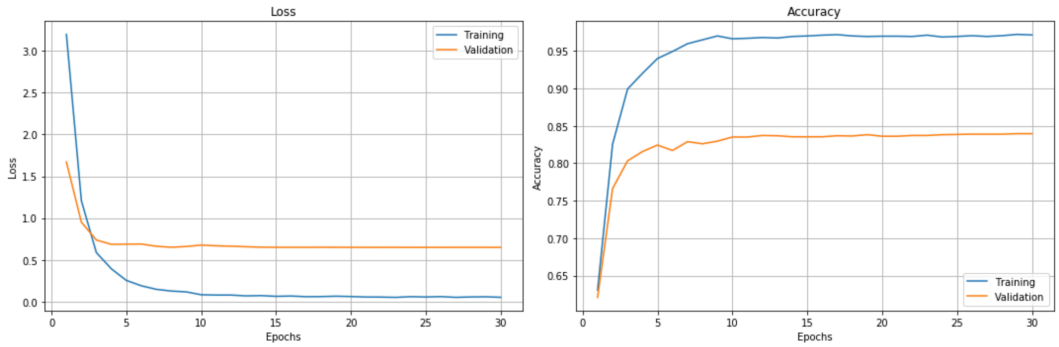
```python
# Define transforms for training phase
train_transform = transforms.Compose([transforms.Resize(256),      # Resizes short size of the PIL image to 256
                                      transforms.CenterCrop(224),  # Crops a central square patch of the image
                                                                   # 224 because torchvision's AlexNet needs a 224x224 input!
                                                                   # Remember this when applying different transformations, otherwise you get an error
                                      transforms.RandomHorizontalFlip(),  # Horizontally flip the given PIL Image randomly with a given probability
                                      transforms.RandomRotation(15),      # Rotate the image by angle
                                      transforms.RandomPerspective(distortion_scale=0.4), # Performs Perspective transformation of the given PIL Image ran
                                      transforms.ColorJitter(brightness=0.4, contrast=0.4, saturation=0.4, hue=0.1),  # Randomly change the brightness, co
                                      transforms.ToTensor(), # Turn PIL Image to torch.Tensor
                                      transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225)) # Normalizes tensor with mean and standard deviat
])
```

Training using:
learning_rate = 1e-2; batch_size = 256; num_epochs=30; step_size=10.

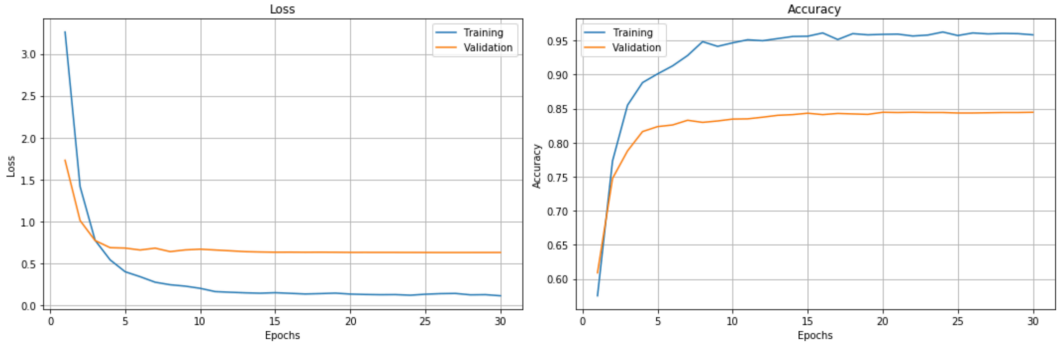
Applying RandomHorizontalFlip transform on training data.



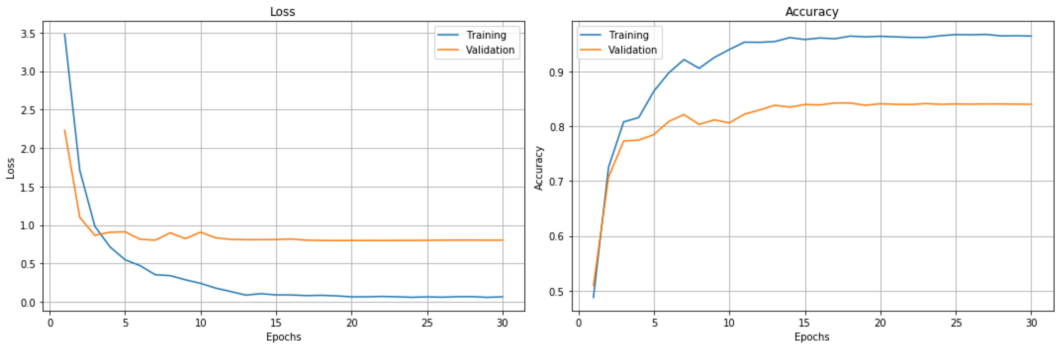
Applying RandomHorizontalFlip and RandomRotation transforms on training data.



Applying RandomHorizontalFlip, RandomRotation and ColorJitter transforms on training data.



Applying RandomHorizontalFlip, RandomRotation, RandomPerspective and ColorJitter transforms on training data.



Applying different combination of transforms on trining data no much changes involves the model performance. This could be due to the fact that data used for test are not too much different from data used for training so using data augmentation does not significantly change results.

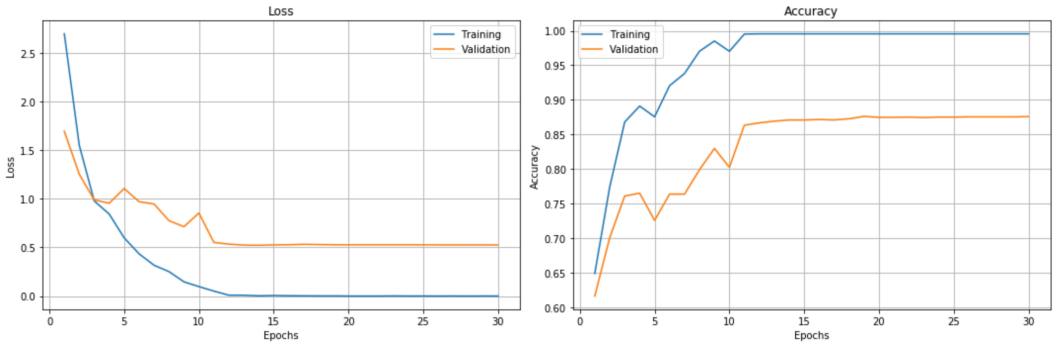
Beyond AlexNet

In this section, other types of Convolutional Neural Networks have been used: VGG-16 and ResNet-50.

VGG-16

```
net = vgg16(pretrained=True) # Loading VGG-16 model (pretrained on ImageNet)
net.classifier[6] = nn.Linear(4096, NUM_CLASSES)  # Changed the last layer of VGG-16
```

Training using:
learning_rate = 1e-2; batch_size = 32; num_epochs=30; step_size=10.



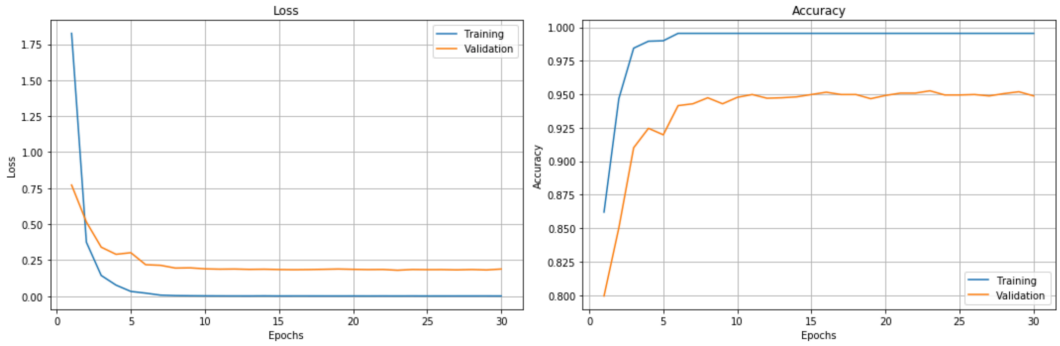
Results on test set.

```
100%|██████████| 91/91 [00:28<00:00,  4.02it/s]Test Accuracy: 0.8686484618043553
Best Accuracy on Validation: 0.876210235131397
```

ResNet-50

```
net = resnet50(pretrained=True) # Loading ResNet-50 model (pretrained on ImageNet)
net.fc = nn.Linear(2048, NUM_CLASSES) # Changed the last layer of ResNet-50
```

Training using:
learning_rate = 1e-2; batch_size = 32; num_epochs=30; step_size=10.



Results on test set.

```
100%|██████████| 91/91 [00:21<00:00,  5.06it/s]Test Accuracy: 0.944348427238161
Best Accuracy on Validation: 0.9526279391424619
```

Thanks to the great number of layers, this kind of CNN allows to achieve very high score (0.95 on validation set)