Data preparation

After an initial analysis of code and dataset, we can start by doing the data preparation.

Task A/B

In order to load the folder dataset of *Caltech101*, it has been created a custom class loader named *Caltech*. In particular, we load all the dataset (into the RAM, since it is relatively small, to speed up the analysis) in the $\_\_init\_\_()$ constructor. Based on the **split** parameter it loads every image identified by the paths in the *train.txt/test.txt* file discarding the so called "BACKGROUND" spurious class. Once the 101 class names have been retrieved, it is created a dictionary to convert the name into an integer. Moreover the tuples *(PILImage, class_index)* are saved into the **sample** public attribute.

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

    self.split = split
    self.loader = pil_loader

    if self.split == 'train':
      X = read_csv(str(root).split('/')[0] + '/train.txt', sep="/", header=None)
      f = open(str(root) + '/train.txt', "r")
    elif self.split == 'test':
      X = read_csv(str(root).split('/')[0] + '/test.txt', sep="/", header=None)
      f = open(str(root) + '/test.txt', "r")

    classes = [item for item in list(set(X[0])) if "BACKGROUND" not in item]
    classes.sort()
    class_to_idx = {classes[i]: i for i in range(len(classes))}

    samples = []
    for path in f.readlines():
        path = path.rstrip('\n')
        if "BACKGROUND" not in path:
            samples.append((pil_loader(root + '/101_ObjectCategories/' + path),
            class_to_idx[path.split("/")[0]]))


    self.samples = samples
    self.classes = classes
    self.class_to_idx = class_to_idx
    self.labels = list(class_to_idx.values())
    f.close()
```

On the other hand, for what concerns the $\_\_getitem\_\_()$ method, the indexing is already implemented in python and the PIL Image is already loaded, we only apply (if it exists) the transformation and we return the two elements as a tuple. Finally, our $\_\_len\_\_()$ returns the lenght of the **samples**.

```
def __getitem__(self, index):

    image, label = self.samples[index]

    if self.transform is not None:
        image = self.transform(image)

    return image, label

def __len__(self):

    length = len(self.samples)
    return length
```

Training from scratch

Task A

The most effective and simple way to split the train set in train and validation is by using the initial order of the readings in the *train.txt* file reflected in the **samples** attribute. Since every image is loaded sequentially from each class, it is obvious that a split by even/odd indexes will provide half samples per-class for each subset. For what concerns the split action, instead of using two subset, it has been chosen to create two *DataLoader* instances with a custom **sampler** called *SubsetRandomSampler* in order to not waste time in splitting the dataset, but leaving all the dirty work to the loader (one will load only the odd indexes batch by batch, the other one the even ones in the same way).

```
# Define transforms for the evaluation on test 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))
])

# Define transforms for the evaluation on validation phase
valid_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))
])

# Define transforms for the train phase
train_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))
])

# Default: Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
# With pretrained: Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))

# Load the train dataset
dataset = Caltech(DATA_DIR, transform=train_transform)

# Creating data indices for training and validation splits
train_indices = [idx for idx in range(len(dataset)) if idx % 2]
val_indices = [idx for idx in range(len(dataset)) if not idx % 2]

# Creating data samplers:
train_sampler = SubsetRandomSampler(train_indices)
valid_sampler = SubsetRandomSampler(val_indices)

# Load the test dataset
test_dataset = Caltech(DATA_DIR, split='test', transform=eval_transform)

# Create the DataLoaders
train_dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, sampler=train_sampler,
                              num_workers=1, pin_memory=True, drop_last=True)

val_dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, sampler=valid_sampler,
                              num_workers=1, pin_memory=True, drop_last=True)


test_dataloader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False,
                              num_workers=1, pin_memory=True)
```

Task B

Now that we have the validation set, we use it to evaluate the performances of each net model trained epoch by epoch and to retrieve the best model (the one with the best accuracy on the validation set) in order to use it for the test set. A possible implementation, after the training process and before the next epoch could simply iterate on the *val_dataloader* batch-by-batch, get the predictions and update the correct ones; if the accuracy (correct predictions on total samples) is better than any previous one, save it with the corresponding **net**. After

all the epochs, we will be able to use the **best_net_acc** to evaluate on the test set (the code below it is only focused for this task, some lines of code might be discarded).

```
1
2  # OTHER CODE
3
4  for epoch in range(NUM_EPOCHS):
5      for image, label in train_dataloader:
6          # DO TRAIN STUFF
7
8      # Set net to evaluation mode
9      net.train(mode=False)
10     correct = 0
11     with torch.no_grad():  # impacts the autograd engine and deactivate it to speed up computation
12         for data, target in val_dataloader:
13           data, target = data.to(DEVICE), target.to(DEVICE)
14           outputs = net(data)
15           # Get predictions
16           _, preds = torch.max(outputs.data, 1)
17
18           # Update Corrects
19           correct += torch.sum(preds == target.data).data.item()
20
21     # Calculate Accuracy
22     accuracy = correct / float(len(val_dataloader) * BATCH_SIZE)
23     if accuracy > best_acc:
24         best_acc = accuracy
25         best_net_acc = copy.deepcopy(net)
26
27     # OTHER CODE
28
29     scheduler.step()
```

Task C

The default hyperparameters were too *soft*, in particular the learning rate (too low). In order to increase the overall accuracy on the validation set and consequently on the test set, they have been chosen two others different set of hyperparameters.
The first one uses a learning rate of 0.1, a really high one, in order to cool down as soon as possible the initial values of the loss that do not provide any gain. Since we risk the loss to explode, it has been chosen a smaller step size of 10 with a big gamma of 0.6 to compensate and mantain a heavy loss decrease without explosions (fig. 1).

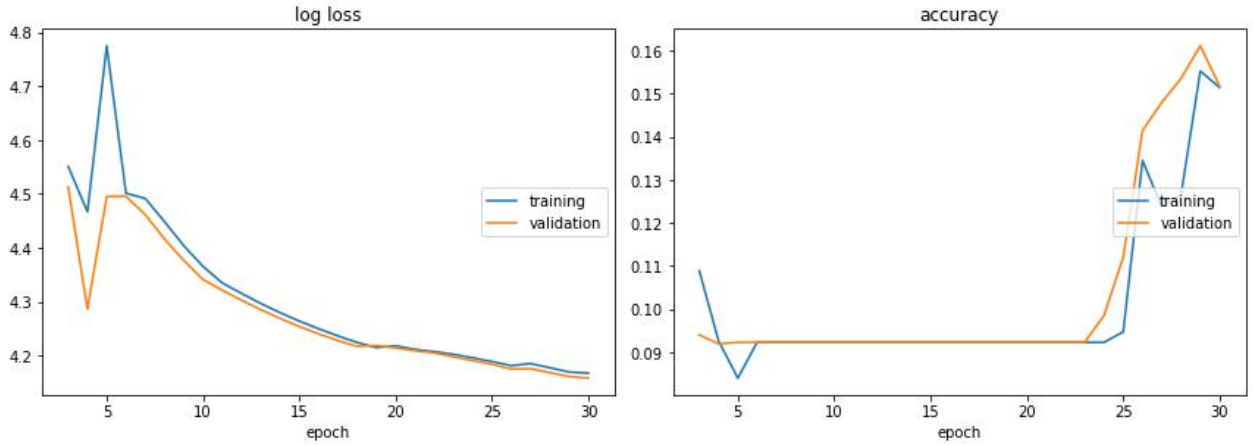

Figure 1: Log loss and accuracy for the first set

The second one uses a learning rate of 0.07 more or less. Because of the big learning rate at the previous step, it has been chosen to lower it a bit. The step size must be lowered at 10 with a total ecpochs of 40 and a gamma really high of 0.9. In both tasks it is used a high batch size of 964, a number that is divisor of the dataset in order to not waste samples (fig. 2).

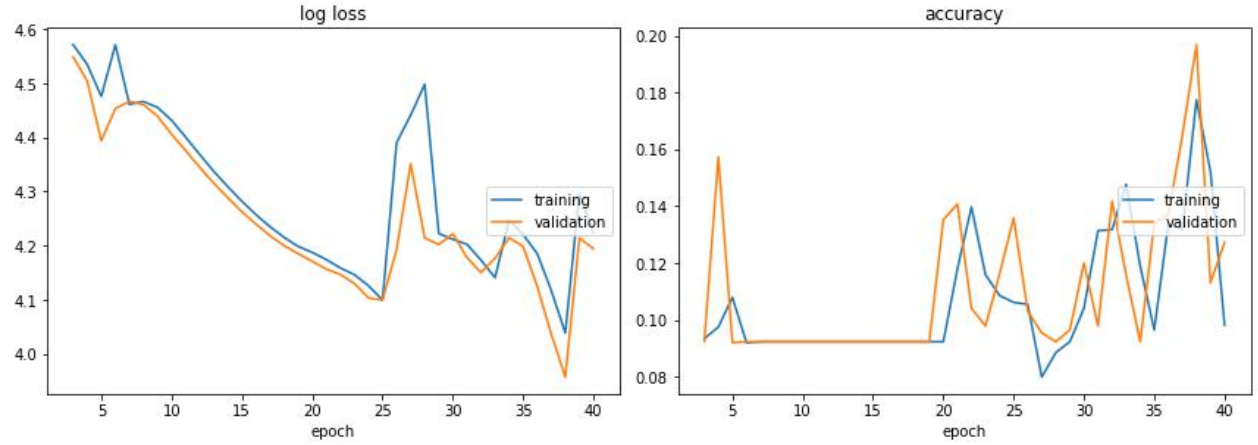

Figure 2: Log loss and accuracy for the second set

The first set performs like 16% accuracy on valid and test while the second one even better (20%). That's a gret increase w.r.t. the initial settings, but obviously it isn't enoght. We can see from the graphs that the loss and the accuracy are scattered. This leads us to a lower LR with a really higher number of epochs.

Transfer Learning

Since our dataset is a small one, we can use the weights learned by training on a large related dataset (in our case, ImageNet) as a starting point for training on the small dataset.

Task A/B

In order to load *AlexNet* using transfer learning, we pass **pretrained=True** during the instantiation of the net. Obviously, we need to change the normalization and so we work on the *train_transform*, *valid_transform* and *eval_transform* in order to modify the weights (mean and std) and set the ones from ImageNet.

```
# Modify the net = alexnet() to
net = alexnet(pretrained=True)

# Modify every transform.Normalize((0.5, 0.5, 0.5),(0.5, 0.5, 0.5)) to
transform.Normalize((0.485 , 0.456 , 0.406) , (0.229 , 0.224 , 0.225))
```

Now we are ready to run our experiments in order to improve our general accuracy using the best hyperparameters found.

Task C

Since the *pretrained* model has a bigger learning capacity due to its past train on huge amount of data, we start with a lower LR of 0.005 and a step size of 25 of 30 epochs and the same gamma of 0.5. Durig the training procedure we retrieve the accuracy and the mean loss per-epoch in order to have everything under control and supervise overfitting/underfitting. The performances with respect to the no transfer learning net, are really improved to about 80% of accuracy on both validation and test dataset (fig. 3).
As we can see, we have some margin on the training set and it could be optimized more, like for example increasing the learning rate to 0.02. The result is great, we have a plus 6% with respect to the previous set (86%, fig. 4).
But in this case the train accuracy is saturated, so maybe it would be better to decrease the learning rate at 0.003 and increase the epochs to 40 with a step size of 10 and a gamma of 0.5 (fig. 5).
At the end we reached a 83% accuracy, but we reached it slowly, trying not to overfit too much.

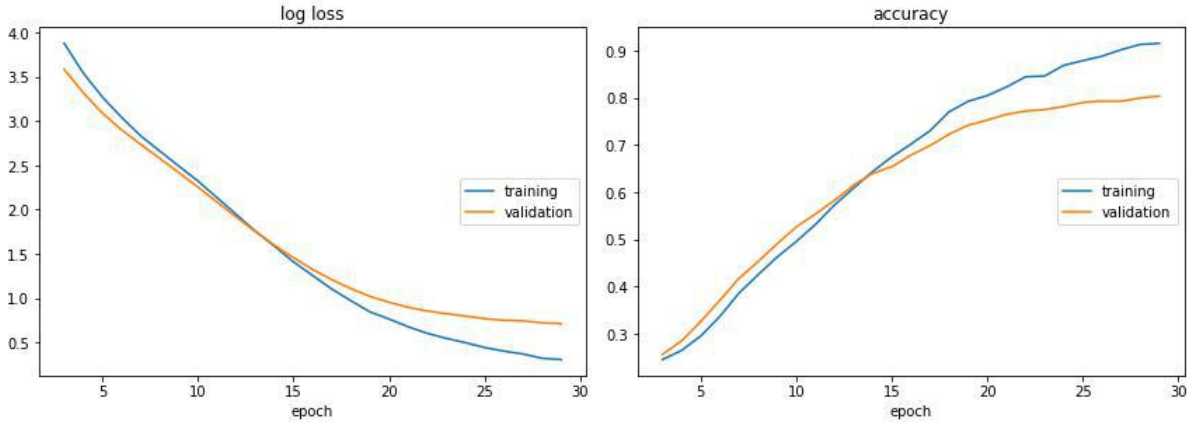

Figure 3: Log loss and accuracy for the first set of hyperparameters

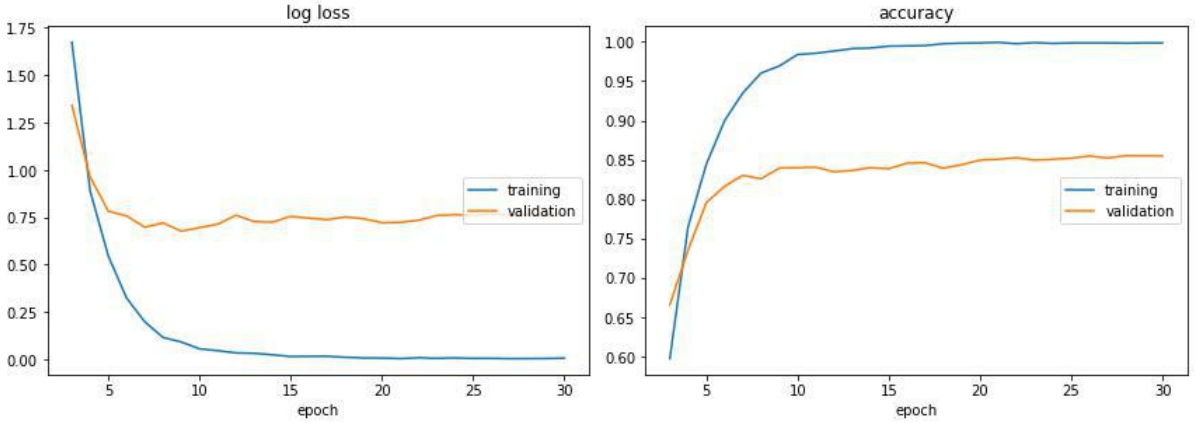

Figure 4: Log loss and accuracy for the second set of hyperparameters

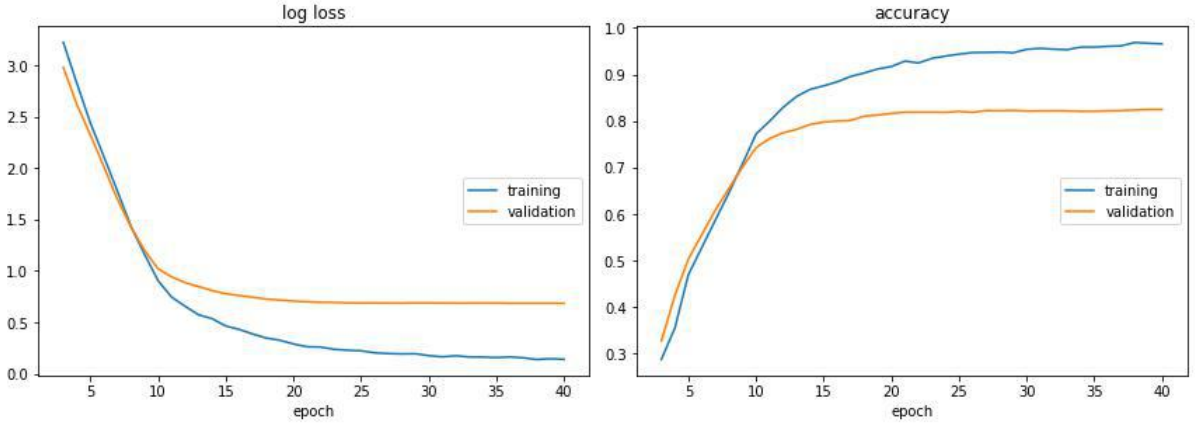

Figure 5: Log loss and accuracy for the third set of hyperparameters

Task D

Now we will use the second set of the previous step (the one that performed better) to freeze part of the net and optimize only the fully connected layers. In order to do so, the parameters need to be changed:

```
# From parameters_to_optimize = net.parameters() to
parameters_to_optimize = net.classifier.parameters()
```

And the results are pretty much the same (about 86%, fig. 6).

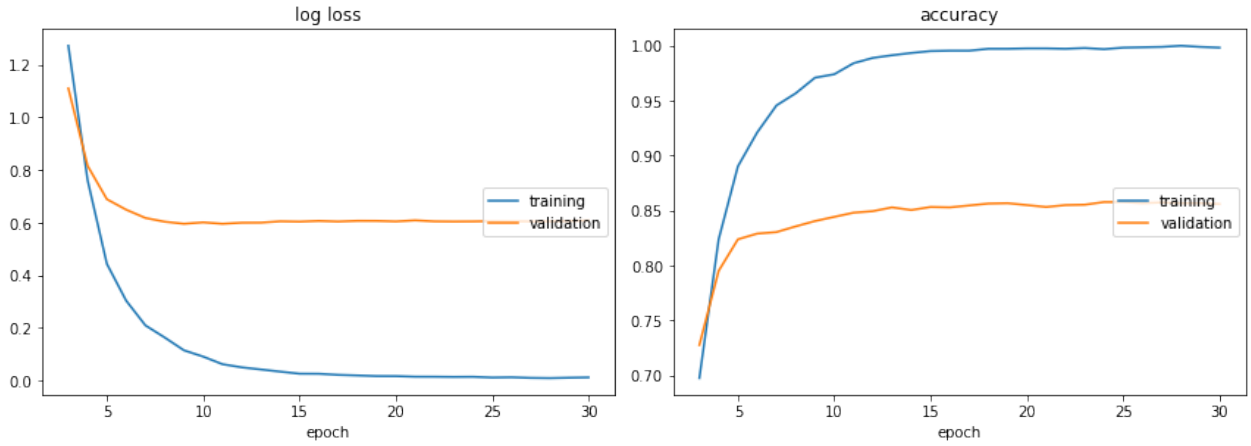

Figure 6: Log loss and accuracy for the set of hyperparameters with FC-layers only

With the difference on the validation loss that seems not so scattered as before.

Task E

As done in the previous step we freeze part of the net and we optimize only the convolutional layers. The code must be now modified in this way:

```
# From parameters_to_optimize = net.classifier.parameters() to
parameters_to_optimize = net.features.parameters()
```

Using the same parameters as before, we can see that the net performs much worse, with a final accuracy of 50% (fig. 7).

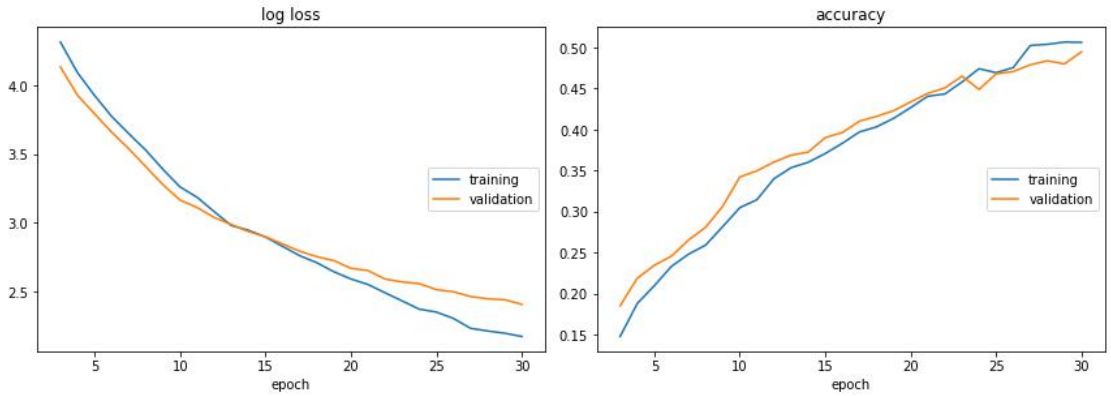

Figure 7: Log loss and accuracy for the set of hyperparameters with Conv-layers only

We can infer that the Conv-layers are much more long to train, but they are the key on the image classification. An improvement could be to fix different learning rates for Conv-layers and FC-layers at the same time passing a dictionary to the optimizer.

Data Augmentation

Task A

In order to increase our dataset without changing the labels, we used 3 transformations: *first*, *second* and *third*. The use of a random transformation leads to increase the dataset size in the sense that, for each epoch, around a half (probability of 50%) of the dataset would be different (without changing the classification label). This means that on 30 epochs the net is not only trained on the initial dataset, but on the random transformations too. For what concerns the code, the transformations are implemented as it follows:

```
train_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))
    ])

# Define transforms for the data augmentation (for the train)
first = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
    ])


second = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.RandomVerticalFlip(),
                                      transforms.RandomPerspective(),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
    ])


third = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224),
                            transforms.RandomRotation(180), transforms.RandomVerticalFlip(),
                            transforms.RandomHorizontalFlip(), transforms.RandomPerspective(),
                            transforms.ToTensor(), transforms.Normalize((0.485, 0.456,
                            0.406), (0.229, 0.224, 0.225))
                            ])
```

Since we have only one dataset where we use two dataloaders of validation and train, it has been implemented a method to change the transform of the dataset at runtime.

```
# Before it starts the train for loop,
# set the desired transform (train_transform, first, second or third)
train_dataloader.dataset.transform = train_transform  # or first or second or third

# Before it starts the validation evaluation, reset the transform to the valid
val_dataloader.dataset.transform = valid_transform
```

With the set that gave the best accuracy so far, it has been run the train with the first (fig. 8), second (fig. 9) and third (fig. 10) transform. Since the sets are quite similar, what we will see is a reduction on the final accuracy to 82% on the first, 80% on the second and 70% on the third (since the last one is the one with more transformations at once).

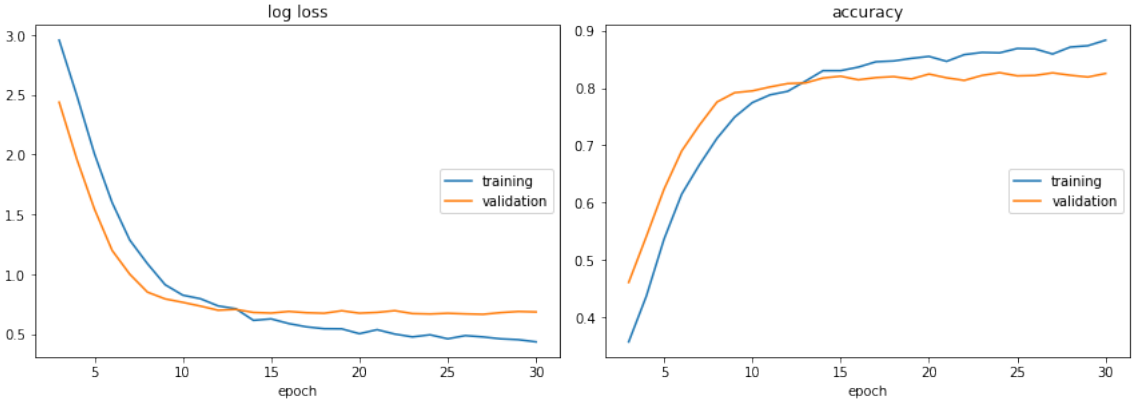

Figure 8: Log loss and accuracy for the data augmentation task with first transformation

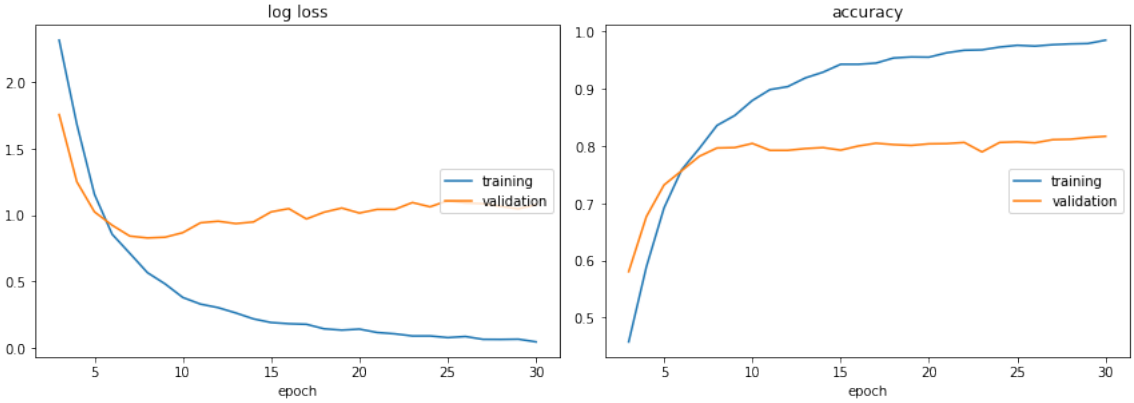

Figure 9: Log loss and accuracy for the data augmentation task with second transformation

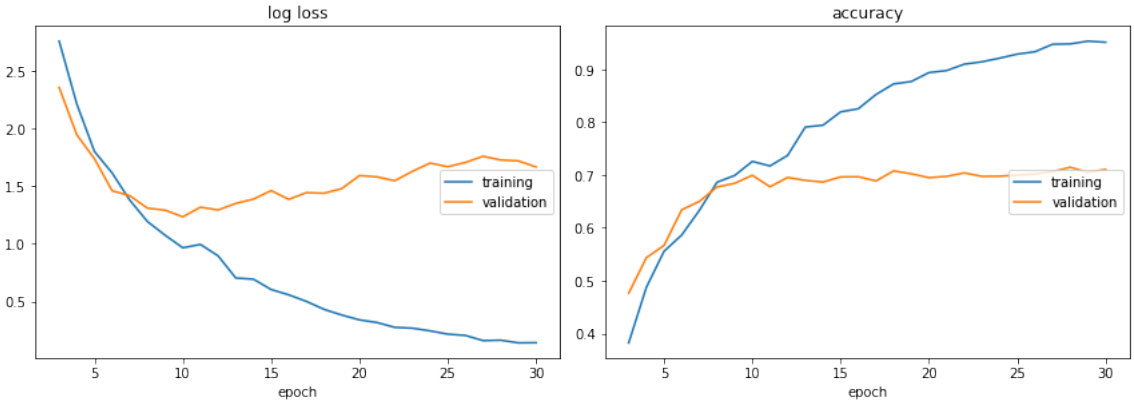

Figure 10: Log loss and accuracy for the data augmentation task with third transformation

Beyond AlexNet

In this section we're going to experiment with two different models: **resnext101_32x8d** and **inception_v3**.

resnext101_32x8d

It is a simple, highly modularized network architecture for image classification, constructed by repeating a building block that aggregates a set of transformations with the same topology. Its simple design results in a homogeneous, multi-branch architecture that has only a few hyper-parameters to set. It reached the 2nd place at the ILSVRC 2016.
We first instantiate it:

```
net = resnext101_32x8d(pretrained=True)
```

And then we modify the FC-layer to adapt it from 1000 ImageNet classes to our 101:

```
# Will be the same as Inceptionv3
net.fc = nn.Linear(2048, NUM_CLASSES)
```

Finally the parameters to optimize are all under analisys.

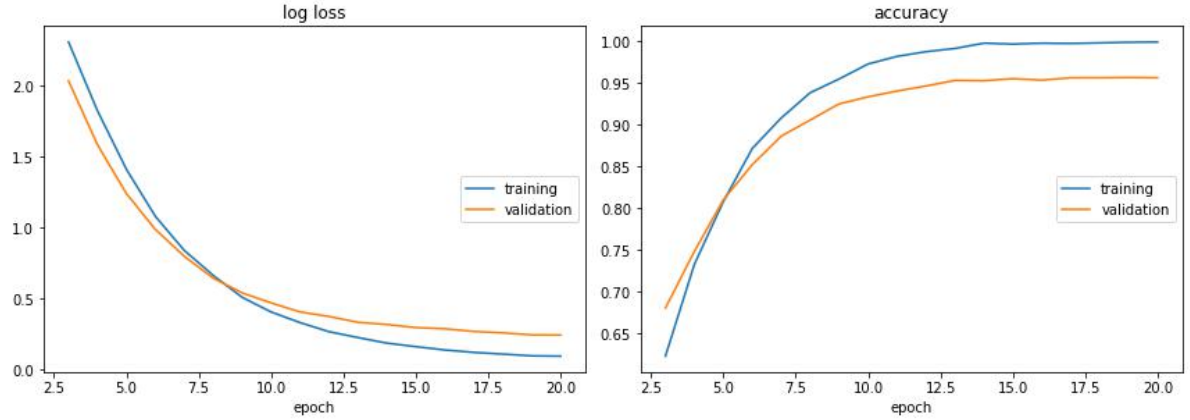

Figure 11: Log loss and accuracy using resnext101_32x8d

Using an epoch of 20 with a LR of 0.0004 and a gamma of 0.5 after 18 epochs, we reached a really good 96% (fig. 11) on test set and validation set. The batch size is really small (about 50) because this net is really deep (101 layers) and so it requires a lot of GPU memory.

inception_v3

In order to train such a big net as Inception3, it has been reduced the batch size to 50 and only the params of the initial Conv-layers and the final FC-layer are optimized (in order to optimize every layer, it is needed mode GPU power). For what concerns the learning rate, it is fixed at 0.003 and, since it takes some time to train, only 15 epochs are allowed. With this setting the accuracy on the validation and test sets is a really good 91%.
In order to instantiate the net, on Python, it is replaced the previous *net* with the new one:

```
# The aux_logits param represent an auxiliary layer that we disabled because not required
net = inception_v3(pretrained=True, aux_logits=False)
```

Now it is time to modify the FC-layer with the number of classes of our *Caltech-101* dataset:

```
# wrt AlexNet we have only one layer fully connected and it is the last one before the softwmax
net.fc = nn.Linear(2048, NUM_CLASSES)
```

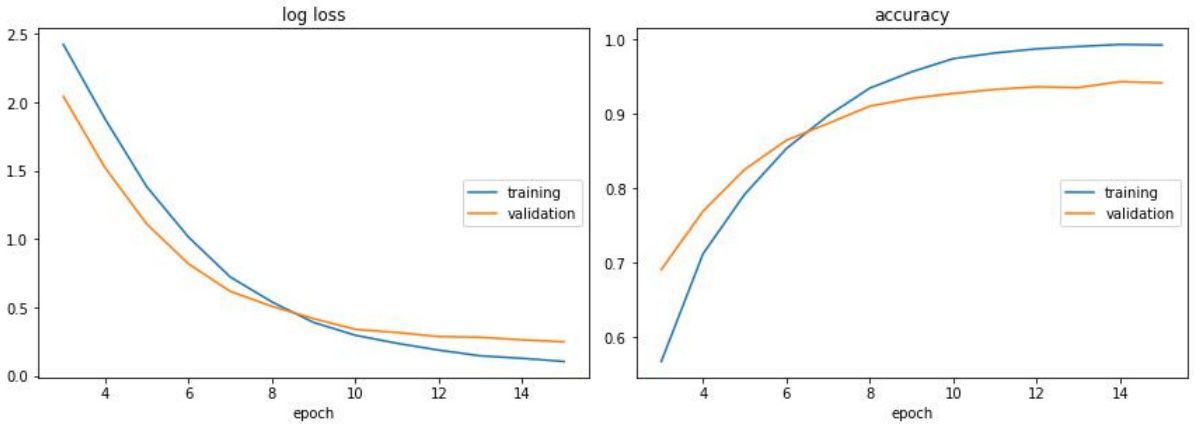

Figure 12: Log loss and accuracy using inception_v3

And specify only the first Conv-layers and the FC-layer to optimize at once:

```
# Parameters for Inception_v3, first Conv2d only and the fully connected layer
parameters_to_optimize = [{'params': net.Conv2d_1a_3x3.parameters()},
                          {'params': net.Conv2d_2a_3x3.parameters()},
                          {'params': net.Conv2d_2b_3x3.parameters()},
                          {'params': net.Conv2d_3b_1x1.parameters()},
                          {'params': net.Conv2d_4a_3x3.parameters()},
                          {'params': net.fc.parameters()}]
```

It is important to underline that the input image for inception_v3 is 299x299, so we need to modify the transforms too:

```
# Define transforms for the evaluation phase
eval_transform = transforms.Compose([transforms.Resize(320),
                                     transforms.CenterCrop(299),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.485 , 0.456 , 0.406) , (0.229 ,
    0.224 , 0.225))
])

# Define transforms for the validation phase
valid_transform = transforms.Compose([transforms.Resize(320),
                                      transforms.CenterCrop(299),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.485 , 0.456 , 0.406) , (0.229 , 0.224 ,
    0.225))
])

# Define transforms for the train phase
train_transform = transforms.Compose([transforms.Resize(320),
                                      transforms.CenterCrop(299),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.485 , 0.456 , 0.406) , (0.229 , 0.224 ,
    0.225))
])
```

And we leave everything else as it was. On the other hand, the log loss and accuracy plot is really smooth and it lead to think that there is no underfitting/overfitting (fig. 12).

Early stopping mechanism

In the code has been added (commented) the *early stopping* action. It is a technique for controlling overfitting in models, especially for neural networks, by stopping training before the weights have converged. Often we stop when the performance has stopped improving on a held-out validation set. For what concerns the code:

```
# Preparation
n_epochs_stop = 7
min_val_loss = np.Inf
best_net = None


# EARLY STOPPING IMPLEMENTATION
if valid_loss.item() < min_val_loss:
    # Save the model
    best_net = copy.deepcopy(net)
    epochs_no_improve = 0
    min_val_loss = valid_loss
else:
    epochs_no_improve += 1
    # Check early stopping condition
    if epochs_no_improve == n_epochs_stop:
        print('Early stopping!')
        # Load in the best model
        net = copy.deepcopy(best_net)
```

In this code we can see that, after every validation, if the mean loss is lower than the minimum found so far, save the net as the best one. On the other hand, if the loss is increasing for 7 consecutive times, reload the last best net with the minimum mean loss found and continue with the training in the next epoch using the last loaded net as starting point.