

## Introduction au Machine Learning - Enise - Centrale Lyon

2024-2025

Emmanuel Dellandréa

### ✓ TD7 – Convolutional Neural Networks

The objective of this tutorial is to use the PyTorch library for building, training, and evaluating CNN models.

#### ✓ Sequence 1: Training a CNN to classify CIFAR10 images

The goal is to apply a Convolutional Neural Net (CNN) model on the CIFAR10 image dataset and test the accuracy of the model on the basis of image classification.

Be sure to check the PyTorch tutorials and documentation when needed:

<https://pytorch.org/tutorials/>

<https://pytorch.org/docs/stable/index.html>

[+ Code](#)[+ Texte](#)

You can test if GPU is available on your machine and thus train on it to speed up the process

```

1 import torch
2
3 # check if CUDA is available
4 train_on_gpu = torch.cuda.is_available()
5
6 if not train_on_gpu:
7     print("CUDA is not available. Training on CPU ...")
8 else:
9     print("CUDA is available! Training on GPU ...")

```

Next we load the CIFAR10 dataset

```

1 import numpy as np
2 from torchvision import datasets, transforms
3 from torch.utils.data.sampler import SubsetRandomSampler
4
5 # number of subprocesses to use for data loading
6 num_workers = 0
7 # how many samples per batch to load
8 batch_size = 20
9 # percentage of training set to use as validation
10 valid_size = 0.2
11
12 # convert data to a normalized torch.FloatTensor
13 transform = transforms.Compose(
14     [transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
15 )
16
17 # choose the training and test datasets
18 train_data = datasets.CIFAR10("data", train=True, download=True, transform=transform)
19 test_data = datasets.CIFAR10("data", train=False, download=True, transform=transform)
20
21 # obtain training indices that will be used for validation
22 num_train = len(train_data)
23 indices = list(range(num_train))
24 np.random.shuffle(indices)
25 split = int(np.floor(valid_size * num_train))
26 train_idx, valid_idx = indices[split:], indices[:split]
27
28 # define samplers for obtaining training and validation batches
29 train_sampler = SubsetRandomSampler(train_idx)
30 valid_sampler = SubsetRandomSampler(valid_idx)
31
32 # prepare data loaders (combine dataset and sampler)
33 train_loader = torch.utils.data.DataLoader(
34     train_data, batch_size=batch_size, sampler=train_sampler, num_workers=num_workers
35 )
36 valid_loader = torch.utils.data.DataLoader(
37     train_data, batch_size=batch_size, sampler=valid_sampler, num_workers=num_workers
38 )

```

```

39 test_loader = torch.utils.data.DataLoader(
40     test_data, batch_size=batch_size, num_workers=num_workers
41 )
42
43 # specify the image classes
44 classes = [
45     "airplane",
46     "automobile",
47     "bird",
48     "cat",
49     "deer",
50     "dog",
51     "frog",
52     "horse",
53     "ship",
54     "truck",
55 ]

```

CNN definition (this one is an example)

```

1 import torch.nn as nn
2 import torch.nn.functional as F
3
4 # define the CNN architecture
5
6
7 class Net(nn.Module):
8     def __init__(self):
9         super(Net, self).__init__()
10         self.conv1 = nn.Conv2d(3, 6, 5)
11         self.pool = nn.MaxPool2d(2, 2)
12         self.conv2 = nn.Conv2d(6, 16, 5)
13         self.fc1 = nn.Linear(16 * 5 * 5, 120)
14         self.fc2 = nn.Linear(120, 84)
15         self.fc3 = nn.Linear(84, 10)
16
17     def forward(self, x):
18         x = self.pool(F.relu(self.conv1(x)))
19         x = self.pool(F.relu(self.conv2(x)))
20         x = x.view(-1, 16 * 5 * 5)
21         x = F.relu(self.fc1(x))
22         x = F.relu(self.fc2(x))
23         x = self.fc3(x)
24         return x
25
26
27 # create a complete CNN
28 model = Net()
29 print(model)
30 # move tensors to GPU if CUDA is available
31 if train_on_gpu:
32     model.cuda()

```

Loss function and training using SGD (Stochastic Gradient Descent) optimizer

```

1 import torch.optim as optim
2
3 criterion = nn.CrossEntropyLoss() # specify loss function
4 optimizer = optim.SGD(model.parameters(), lr=0.01) # specify optimizer
5
6 n_epochs = 30 # number of epochs to train the model
7 train_loss_list = [] # list to store loss to visualize
8 valid_loss_min = np.Inf # track change in validation loss
9
10 for epoch in range(n_epochs):
11     # Keep track of training and validation loss
12     train_loss = 0.0
13     valid_loss = 0.0
14
15     # Train the model
16     model.train()
17     for data, target in train_loader:
18         # Move tensors to GPU if CUDA is available
19         if train_on_gpu:
20             data, target = data.cuda(), target.cuda()
21         # Clear the gradients of all optimized variables
22         optimizer.zero_grad()
23         # Forward pass: compute predicted outputs by passing inputs to the model
24         output = model(data)

```

```

25     # Calculate the batch loss
26     loss = criterion(output, target)
27     # Backward pass: compute gradient of the loss with respect to model parameters
28     loss.backward()
29     # Perform a single optimization step (parameter update)
30     optimizer.step()
31     # Update training loss
32     train_loss += loss.item() * data.size(0)
33
34     # Validate the model
35     model.eval()
36     for data, target in valid_loader:
37         # Move tensors to GPU if CUDA is available
38         if train_on_gpu:
39             data, target = data.cuda(), target.cuda()
40         # Forward pass: compute predicted outputs by passing inputs to the model
41         output = model(data)
42         # Calculate the batch loss
43         loss = criterion(output, target)
44         # Update average validation loss
45         valid_loss += loss.item() * data.size(0)
46
47     # Calculate average losses
48     train_loss = train_loss / len(train_loader)
49     valid_loss = valid_loss / len(valid_loader)
50     train_loss_list.append(train_loss)
51
52     # Print training/validation statistics
53     print(
54         "Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}".format(
55             epoch, train_loss, valid_loss
56         )
57     )
58
59     # Save model if validation loss has decreased
60     if valid_loss <= valid_loss_min:
61         print(
62             "Validation loss decreased ({:.6f} --> {:.6f}). Saving model ...".format(
63                 valid_loss_min, valid_loss
64             )
65         )
66         torch.save(model.state_dict(), "model_cifar.pt")
67         valid_loss_min = valid_loss

```

Does overfit occur? If so, do an early stopping.

```

1 import matplotlib.pyplot as plt
2
3 plt.plot(range(len(train_loss_list)), train_loss_list)
4 plt.xlabel("Epoch")
5 plt.ylabel("Loss")
6 plt.title("Performance of Model 1")
7 plt.show()

```

Now loading the model with the lowest validation loss value

```

1 model.load_state_dict(torch.load("./model_cifar.pt"))
2
3 # track test loss
4 test_loss = 0.0
5 class_correct = list(0.0 for i in range(10))
6 class_total = list(0.0 for i in range(10))
7
8 model.eval()
9 # iterate over test data
10 for data, target in test_loader:
11     # move tensors to GPU if CUDA is available
12     if train_on_gpu:
13         data, target = data.cuda(), target.cuda()
14     # forward pass: compute predicted outputs by passing inputs to the model
15     output = model(data)
16     # calculate the batch loss
17     loss = criterion(output, target)
18     # update test loss
19     test_loss += loss.item() * data.size(0)
20     # convert output probabilities to predicted class
21     _, pred = torch.max(output, 1)
22     # compare predictions to true label
23     correct_tensor = pred.eq(target.data.view_as(pred))

```

```

24     correct = (
25         np.squeeze(correct_tensor.numpy())
26         if not train_on_gpu
27         else np.squeeze(correct_tensor.cpu().numpy())
28     )
29     # calculate test accuracy for each object class
30     for i in range(batch_size):
31         label = target.data[i]
32         class_correct[label] += correct[i].item()
33         class_total[label] += 1
34
35 # average test loss
36 test_loss = test_loss / len(test_loader)
37 print("Test Loss: {:.6f}\n".format(test_loss))
38
39 for i in range(10):
40     if class_total[i] > 0:
41         print(
42             "Test Accuracy of %5s: %2d%% (%2d/%2d)"
43             % (
44                 classes[i],
45                 100 * class_correct[i] / class_total[i],
46                 np.sum(class_correct[i]),
47                 np.sum(class_total[i]),
48             )
49         )
50     else:
51         print("Test Accuracy of %5s: N/A (no training examples)" % (classes[i]))
52
53 print(
54     "\nTest Accuracy (Overall): %2d%% (%2d/%2d)"
55     % (
56         100.0 * np.sum(class_correct) / np.sum(class_total),
57         np.sum(class_correct),
58         np.sum(class_total),
59     )
60 )

```

## Experiments:

Build a new network with the following structure.

- It has 3 convolutional layers of kernel size 3 and padding of 1.
- The first convolutional layer must output 16 channels, the second 32 and the third 64.
- At each convolutional layer output, we apply a ReLU activation then a MaxPool with kernel size of 2.
- Then, three fully connected layers, the first two being followed by a ReLU activation.
- The first fully connected layer will have an output size of 512.
- The second fully connected layer will have an output size of 64.

Compare the results obtained with this new network to those obtained previously.

## ✓ Sequence 2: Working with pre-trained models.

PyTorch offers several pre-trained models <https://pytorch.org/vision/0.8/models.html>

We will use ResNet50 trained on ImageNet dataset (<https://www.image-net.org/index.php>). Use the following code with the files `imagenet-simple-labels.json` that contains the imagenet labels and the image `dog.png` that we will use as test.

```

1 import json
2 from PIL import Image
3 from torchvision import models
4
5 # Choose an image to pass through the model
6 test_image = "dog.png"
7
8 # Configure matplotlib for pretty inline plots
9 %matplotlib inline
10 %config InlineBackend.figure_format = 'retina'
11
12 # Prepare the labels
13 with open("imagenet-simple-labels.json") as f:
14     labels = json.load(f)
15
16 # First prepare the transformations: resize the image to what the model was trained on and convert it to a tensor
17 data_transform = transforms.Compose(
18     [
19         transforms.Resize((224, 224)),

```

```

20     transforms.ToTensor(),
21     transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
22 ]
23 )
24 # Load the image
25
26 image = Image.open(test_image)
27 plt.imshow(image), plt.xticks([]), plt.yticks([])
28
29 # Now apply the transformation, expand the batch dimension, and send the image to the GPU
30 # image = data_transform(image).unsqueeze(0).cuda()
31 image = data_transform(image).unsqueeze(0)
32
33 # Download the model if it's not there already. It will take a bit on the first run, after that it's fast
34 model = models.resnet50(pretrained=True)
35 # Send the model to the GPU
36 # model.cuda()
37 # Set layers such as dropout and batchnorm in evaluation mode
38 model.eval()
39
40 # Get the 1000-dimensional model output
41 out = model(image)
42 # Find the predicted class
43 print("Predicted class is: {}".format(labels[out.argmax()]))

```

## Experiments:

Study the code and the results obtained. Possibly add other images downloaded from the internet.

Experiment with other pre-trained CNN models.

## ✓ Sequence 3: Transfer Learning

For this work, we will use a pre-trained model (ResNet18) as a descriptor extractor and will refine the classification by training only the last fully connected layer of the network. Thus, the output layer of the pre-trained network will be replaced by a layer adapted to the new classes to be recognized which will be in our case ants and bees. Download and unzip in your working directory the dataset available at the address :

[https://download.pytorch.org/tutorial/hymenoptera\\_data.zip](https://download.pytorch.org/tutorial/hymenoptera_data.zip)

Execute the following code in order to display some images of the dataset.

```

1 import os
2
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import torch
6 import torchvision
7 from torchvision import datasets, transforms
8 import torch.nn as nn
9 import torch.optim as optim
10 from torch.optim import lr_scheduler
11
12 # Data augmentation and normalization for training
13 # Just normalization for validation
14 data_transforms = {
15     "train": transforms.Compose(
16         [
17             transforms.RandomResizedCrop(
18                 224
19             ), # ImageNet models were trained on 224x224 images
20             transforms.RandomHorizontalFlip(), # flip horizontally 50% of the time - increases train set variability
21             transforms.ToTensor(), # convert it to a PyTorch tensor
22             transforms.Normalize(
23                 [0.485, 0.456, 0.406], [0.229, 0.224, 0.225]
24             ), # ImageNet models expect this norm
25         ]
26     ),
27     "val": transforms.Compose(
28         [
29             transforms.Resize(256),
30             transforms.CenterCrop(224),
31             transforms.ToTensor(),
32             transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
33         ]
34     ),
35 }
36
37 data_dir = "hymenoptera_data"

```

```

38 # Create train and validation datasets and loaders
39 image_datasets = {
40     x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x])
41     for x in ["train", "val"]
42 }
43 dataloaders = {
44     x: torch.utils.data.DataLoader(
45         image_datasets[x], batch_size=4, shuffle=True, num_workers=0
46     )
47     for x in ["train", "val"]
48 }
49 dataset_sizes = {x: len(image_datasets[x]) for x in ["train", "val"]}
50 class_names = image_datasets["train"].classes
51 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
52
53 # Helper function for displaying images
54 def imshow(inp, title=None):
55     """Imshow for Tensor."""
56     inp = inp.numpy().transpose((1, 2, 0))
57     mean = np.array([0.485, 0.456, 0.406])
58     std = np.array([0.229, 0.224, 0.225])
59
60     # Un-normalize the images
61     inp = std * inp + mean
62     # Clip just in case
63     inp = np.clip(inp, 0, 1)
64     plt.imshow(inp)
65     if title is not None:
66         plt.title(title)
67     plt.pause(0.001) # pause a bit so that plots are updated
68     plt.show()
69
70
71 # Get a batch of training data
72 inputs, classes = next(iter(dataloaders["train"]))
73
74 # Make a grid from batch
75 out = torchvision.utils.make_grid(inputs)
76
77 imshow(out, title=[class_names[x] for x in classes])
78
79

```

Now, execute the following code which uses a pre-trained model ResNet18 having replaced the output layer for the ants/bees classification and performs the model training by only changing the weights of this output layer.

```

1 import copy
2 import time
3
4
5 def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
6     since = time.time()
7
8     best_model_wts = copy.deepcopy(model.state_dict())
9     best_acc = 0.0
10
11     epoch_time = [] # we'll keep track of the time needed for each epoch
12
13     for epoch in range(num_epochs):
14         epoch_start = time.time()
15         print("Epoch {}/{}".format(epoch + 1, num_epochs))
16         print("-" * 10)
17
18         # Each epoch has a training and validation phase
19         for phase in ["train", "val"]:
20             if phase == "train":
21                 scheduler.step()
22                 model.train() # Set model to training mode
23             else:
24                 model.eval() # Set model to evaluate mode
25
26             running_loss = 0.0
27             running_corrects = 0
28
29             # Iterate over data.
30             for inputs, labels in dataloaders[phase]:
31                 inputs = inputs.to(device)
32                 labels = labels.to(device)
33
34                 # zero the parameter gradients

```

```

35         optimizer.zero_grad()
36
37         # Forward
38         # Track history if only in training phase
39         with torch.set_grad_enabled(phase == "train"):
40             outputs = model(inputs)
41             _, preds = torch.max(outputs, 1)
42             loss = criterion(outputs, labels)
43
44             # backward + optimize only if in training phase
45             if phase == "train":
46                 loss.backward()
47                 optimizer.step()
48
49         # Statistics
50         running_loss += loss.item() * inputs.size(0)
51         running_corrects += torch.sum(preds == labels.data)
52
53     epoch_loss = running_loss / dataset_sizes[phase]
54     epoch_acc = running_corrects.double() / dataset_sizes[phase]
55
56     print("{} Loss: {:.4f} Acc: {:.4f}".format(phase, epoch_loss, epoch_acc))
57
58     # Deep copy the model
59     if phase == "val" and epoch_acc > best_acc:
60         best_acc = epoch_acc
61         best_model_wts = copy.deepcopy(model.state_dict())
62
63     # Add the epoch time
64     t_epoch = time.time() - epoch_start
65     epoch_time.append(t_epoch)
66     print()
67
68     time_elapsed = time.time() - since
69     print(
70         "Training complete in {:.0f}m {:.0f}s".format(
71             time_elapsed // 60, time_elapsed % 60
72         )
73     )
74     print("Best val Acc: {:.4f}".format(best_acc))
75
76     # Load best model weights
77     model.load_state_dict(best_model_wts)
78     return model, epoch_time
79
80
81 # Download a pre-trained ResNet18 model and freeze its weights
82 model = torchvision.models.resnet18(pretrained=True)
83 for param in model.parameters():
84     param.requires_grad = False
85
86 # Replace the final fully connected layer
87 # Parameters of newly constructed modules have requires_grad=True by default
88 num_ftrs = model.fc.in_features
89 model.fc = nn.Linear(num_ftrs, 2)
90 # Send the model to the GPU
91 model = model.to(device)
92 # Set the loss function
93 criterion = nn.CrossEntropyLoss()
94
95 # Observe that only the parameters of the final layer are being optimized
96 optimizer_conv = optim.SGD(model.fc.parameters(), lr=0.001, momentum=0.9)
97 exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
98 model, epoch_time = train_model(
99     model, criterion, optimizer_conv, exp_lr_scheduler, num_epochs=10
100 )
101

```

## Experiments:

Study the code and the results obtained.

Modify the code and add an "eval\_model" function to allow the evaluation of the model on a test set (different from the learning and validation sets used during the learning phase). Study the results obtained.

Now modify the code to replace the current classification layer with a set of two layers using a "relu" activation function for the middle layer. Renew the experiments and study the results obtained.

Experiment with other models and datasets.

