HW5-Coding(5)

January 11, 2024

1 Homework 5: Convolutional neural network (30 points)

In this part, you need to implement and train a convolutional neural network on the CIFAR-10 dataset with PyTorch. ### What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

1.0.1 Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.
- GPU PyTorch TensorFlow GPU CUDA
-
- TensorFlow PyTorch)
- ## How can I learn PyTorch?

Justin Johnson has made an excellent tutorial for PyTorch.

You can also find the detailed API doc here. If you have other questions that are not addressed by the API docs, the PyTorch forum is a much better place to ask than StackOverflow.

Install PyTorch and Skorch.

[2]:
```
!pip install -q torch skorch torchvision torchtext
```

[2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import skorch
import sklearn
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

1.1 0. Tensor Operations (5 points)

Tensor operations are important in deep learning models. In this part, you are required to get famaliar to some common tensor operations in PyTorch.

1.1.1 1) Tensor squeezing, unsqueezing and viewing

Tensor squeezing, unsqueezing and viewing are important methods to change the dimension of a Tensor, and the corresponding functions are torch.squeeze, torch.unsqueeze and torch.Tensor.view. Please read the documents of the functions, and finish the following practice.

[9]:
```
# x is a tensor with size being (3, 2)
x = torch.Tensor([[1, 2],
                  [3, 4],
                  [5, 6]])
print(x.shape)

# Add two new dimensions to x by using the function torch.unsqueeze
x = torch.unsqueeze(torch.unsqueeze(x, -1), 1)
print(x.shape)

# Remove the two dimensions just added by using the function torch.squeeze
x = torch.squeeze(torch.squeeze(x, -1), 1)
print(x.shape)

# x is now a two-dimensional tensor, or in other words a matrix. Now use the␣
 ↪function torch.Tensor.view and change x to a one-dimensional vector with␣
 ↪size being (6).
x = x.view(-1)
print(x.shape)
```

```
torch.Size([3, 2])
torch.Size([3, 1, 2, 1])
torch.Size([3, 2])
torch.Size([6])
```

1.1.2 2) Tensor concatenation and stack

Tensor concatenation and stack are operations to combine small tensors into big tensors. The corresponding functions are torch.cat and torch.stack. Please read the documents of the functions, and finish the following practice.

[10]:
```
# x is a tensor with size being (3, 2)
x = torch.Tensor([[1, 2], [3, 4], [5, 6]])

# y is a tensor with size being (3, 2)
y = torch.Tensor([[-1, -2], [-3, -4], [-5, -6]])

# Our goal is to generate a tensor z with size as (2, 3, 2), and z[0,:,:] = x,␣
↪z[1,:,:] = y.

# Use torch.stack to generate such a z
# pass
z = torch.stack([x, y])
print(z[0,:,:])
# Use torch.cat and torch.unsqueeze to generate such a z
# pass
z = torch.cat([x.unsqueeze(0), y.unsqueeze(0)], dim = 0 )
print(z[1,:,:])
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
tensor([[-1., -2.],
        [-3., -4.],
        [-5., -6.]])
```

1.1.3 3) Tensor expansion

Tensor expansion is to expand a tensor into a larger tensor along singleton dimensions. The corresponding functions are torch.Tensor.expand and torch.Tensor.expand_as. Please read the documents of the functions, and finish the following practice.

[11]:
```
# x is a tensor with size being (3)
x = torch.Tensor([1, 2, 3])

# Our goal is to generate a tensor z with size (2, 3), so that z[0,:,:] = x,␣
↪z[1,:,:] = x.

# [TO DO]
# Change the size of x into (1, 3) by using torch.unsqueeze.
# pass
x = torch.unsqueeze(x, 0)
print(x.shape)

# [TO DO]
# Then expand the new tensor to the target tensor by using torch.Tensor.expand.
# pass
```

```
z = x.expand(2, -1)
print(z.shape)
```

```
torch.Size([1, 3])
torch.Size([2, 3])
```

1.1.4 4) Tensor reduction in a given dimension

In deep learning, we often need to compute the mean/sum/max/min value in a given dimension of a tensor. Please read the document of torch.mean, torch.sum, torch.max, torch.min, torch.topk, and finish the following practice.

[12]:
```
# x is a random tensor with size being (10, 50)
x = torch.randn(10, 50)

# Compute the mean value for each row of x.
# You need to generate a tensor x_mean of size (10), and x_mean[k, :] is the␣
↪mean value of the k-th row of x.
# pass
x_mean = x.mean(dim=1)
print(x_mean[3, ])

# Compute the sum value for each row of x.
# You need to generate a tensor x_sum of size (10).
# pass
x_sum = x.sum(dim=1)
print(x_sum.shape)

# Compute the max value for each row of x.
# You need to generate a tensor x_max of size (10).
# pass
x_max, _ = x.max(dim=1)
print(x_max.shape)

# Compute the min value for each row of x.
# You need to generate a tensor x_min of size (10).
# pass
x_min, _ = x.min(dim=1)
print(x_min.shape)

# Compute the top-5 values for each row of x.
# You need to generate a tensor x_mean of size (10, 5), and x_top[k, :] is the␣
↪top-5 values of each row in x.
# pass
x_xtop, _ = torch.topk(x, k=5, dim=1, largest=True, sorted=True)
print((x_xtop.shape))
```

```
tensor(0.1405)
torch.Size([10])
torch.Size([10])
torch.Size([10])
torch.Size([10, 5])
```

1.2 Convolutional Neural Networks

Implement a convolutional neural network for image classification on CIFAR-10 dataset.

CIFAR-10 is an image dataset of 10 categories. Each image has a size of 32x32 pixels. The following code will download the dataset, and split it into `train` and `test`. For this question, we use the default validation split generated by Skorch.

[3]:
```
train = torchvision.datasets.CIFAR10("./data", train=True, download=True)
test = torchvision.datasets.CIFAR10("./data", train=False, download=True)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

The following code visualizes some samples in the dataset. You may use it to debug your model if necessary.

[6]:
```
def plot(data, labels=None, num_sample=5):
  n = min(len(data), num_sample)
  for i in range(n):
    plt.subplot(1, n, i+1)
    plt.imshow(data[i], cmap="gray")
    plt.xticks([])
    plt.yticks([])
    if labels is not None:
      plt.title(labels[i])

train.labels = [train.classes[target] for target in train.targets]
plot(train.data, train.labels)
```

1.2.1 1) Basic CNN implementation

Consider a basic CNN model

- It has 3 convolutional layers, followed by a linear layer.
- Each convolutional layer has a kernel size of 3, a padding of 1.
- ReLU activation is applied on every hidden layer.

Please implement this model in the following section. The hyperparameters is then be tuned and you need to fill the results in the table.   CNN

- 3
- 3 1
- ReLU

a) Implement convolutional layers (10 Points) Implement the initialization function and the forward function of the CNN.

[3]:
```
class CNN(nn.Module):
    def __init__(self, channels):
        super(CNN, self).__init__()
        # implement parameter definitions here
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        # pass
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=channels,␣
 ↪kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=channels, out_channels=channels, 3,␣
 ↪padding=1)
        self.conv3 = nn.Conv2d(in_channels=channels, out_channels=channels, 3,␣
 ↪padding=1)
        self.fc = nn.Linear(channels * 32 * 32, 10)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    def forward(self, images):
        # implement the forward function here
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        # pass
        images = images.float()
        images = F.relu(self.conv1(images))
        images = F.relu(self.conv2(images))
        images = F.relu(self.conv3(images))
        images = images.view(images.size(0), -1)
        images = self.fc(images)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return images
```

b) Tune hyperparameters Train the CNN model on CIFAR-10 dataset. We can tune the number of channels, optimizer, learning rate and the number of epochs for best validation accuracy.

[4]:

```
# implement hyperparameters, you can select and modify the hyperparameters by␣
 ↪yourself here.

optimize = [torch.optim.SGD, torch.optim.Adam]
learning_rate = [1e-3]
channel = [16, 32, 64]


train_data_normalized = torch.Tensor(train.data/255)
train_data_normalized = train_data_normalized.permute(0,3,1,2)

for l in learning_rate:
  for o in optimize:
    for c in channel:
      print(f'The channel was {c}, the learning rate was {l} and the optimizer␣
 ↪was {str(o)}')

      cnn = CNN(channels = c)

      model = skorch.NeuralNetClassifier(cnn, criterion=torch.nn.
 ↪CrossEntropyLoss,
                                         device="cuda",
                                         optimizer=o,
                                        # optimizer__momentum=0.90,
                                         lr=l,
                                         max_epochs=50,
                                         batch_size=512,
                                         callbacks=[skorch.callbacks.
 ↪EarlyStopping(lower_is_better=True)])
      # implement input normalization & type cast here
      model.fit(train_data_normalized, torch.LongTensor(train.targets))
```

```
The channel was 16, the learning rate was 0.001 and the optimizer was <class
'torch.optim.sgd.SGD'>
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        2.3023       0.1000        2.3016
1.8696
      2        2.3014       0.1001        2.3006
1.4992
      3        2.3004       0.1002        2.2996
1.6483
      4        2.2993       0.1003        2.2985
1.4937
      5        2.2982       0.1026        2.2973
1.4817
      6        2.2970       0.1084        2.2959
```

```
1.6490
      7          2.2955        0.1191          2.2943
1.6139
      8          2.2939        0.1261          2.2925
1.5891
      9          2.2920        0.1352          2.2904
1.7191
     10          2.2898        0.1441          2.2879
1.6297
     11          2.2870        0.1536          2.2848
1.5851
     12          2.2836        0.1639          2.2809
1.5934
     13          2.2794        0.1746          2.2760
1.7376
     14          2.2740        0.1833          2.2698
1.5488
     15          2.2672        0.1979          2.2620
1.5256
     16          2.2587        0.2089          2.2523
1.7559
     17          2.2482        0.2197          2.2404
1.5962
     18          2.2354        0.2299          2.2260
1.6180
     19          2.2200        0.2397          2.2088
1.6428
     20          2.2017        0.2477          2.1885
1.4556
     21          2.1804        0.2568          2.1652
1.4557
     22          2.1565        0.2635          2.1396
1.4744
     23          2.1308        0.2695          2.1127
1.5492
     24          2.1045        0.2754          2.0859
1.4287
     25          2.0791        0.2785          2.0606
1.4330
     26          2.0558        0.2838          2.0380
1.5790
     27          2.0353        0.2874          2.0186
1.4708
     28          2.0179        0.2900          2.0022
1.4527
     29          2.0031        0.2937          1.9882
1.5850
     30          1.9903        0.2973          1.9762
```

```
1.5017
     31        1.9790       0.3011        1.9654
1.4780
     32        1.9687       0.3045        1.9555
1.5058
     33        1.9590       0.3079        1.9462
1.6080
     34        1.9498       0.3139        1.9373
1.4211
     35        1.9410       0.3188        1.9288
1.4571
     36        1.9324       0.3243        1.9205
1.5877
     37        1.9241       0.3280        1.9125
1.4352
     38        1.9161       0.3324        1.9048
1.4467
     39        1.9084       0.3349        1.8974
1.4845
     40        1.9010       0.3399        1.8902
1.5439
     41        1.8938       0.3413        1.8833
1.4759
     42        1.8869       0.3453        1.8766
1.4976
     43        1.8803       0.3480        1.8702
1.5401
     44        1.8740       0.3506        1.8640
1.4488
     45        1.8679       0.3526        1.8580
1.4856
     46        1.8621       0.3540        1.8523
1.6838
     47        1.8564       0.3570        1.8469
1.5008
     48        1.8510       0.3594        1.8416
1.4871
     49        1.8459       0.3609        1.8366
1.5410
     50        1.8409       0.3613        1.8318
1.5418
The channel was 32, the learning rate was 0.001 and the optimizer was <class
'torch.optim.sgd.SGD'>
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        2.3021       0.1041        2.3013
2.8145
      2        2.3004       0.1489        2.2995
```

```
2.6793
       3        2.2986        0.1506         2.2975
2.5704
       4        2.2965        0.1301         2.2953   2.4280
       5        2.2941        0.1228         2.2925   2.5590
       6        2.2911        0.1244         2.2892   3.1142
       7        2.2875        0.1309         2.2850   3.2769
       8        2.2829        0.1431         2.2798   2.8112
       9        2.2771        0.1596         2.2732
2.6166
      10        2.2698        0.1762         2.2647
2.7641
      11        2.2602        0.1919         2.2536
2.6307
      12        2.2479        0.2082         2.2394
2.6573
      13        2.2321        0.2180         2.2213
2.7342
      14        2.2124        0.2290         2.1989
2.6742
      15        2.1884        0.2427         2.1725
2.6494
      16        2.1611        0.2525         2.1431
2.7171
      17        2.1320        0.2595         2.1131
2.7030
      18        2.1035        0.2667         2.0848
2.7112
      19        2.0776        0.2721         2.0598
2.6295
      20        2.0554        0.2777         2.0386
2.7505
      21        2.0369        0.2822         2.0212
2.5853
      22        2.0215        0.2849         2.0067
2.6423
      23        2.0086        0.2890         1.9944
2.6975
      24        1.9973        0.2925         1.9836
2.6076
      25        1.9872        0.2953         1.9738
2.6293
      26        1.9778        0.3010         1.9647
2.6288
      27        1.9688        0.3049         1.9561
2.8176
      28        1.9602        0.3063         1.9477
2.6918
```

```
     29        1.9517       0.3116        1.9395
2.6841
     30        1.9434       0.3166        1.9315
2.7937
     31        1.9351       0.3232        1.9236
2.9334
     32        1.9269       0.3274        1.9157
2.8763
     33        1.9188       0.3329        1.9079
2.8930
     34        1.9107       0.3356        1.9003
3.2919
     35        1.9029       0.3395        1.8928
3.1846
     36        1.8952       0.3441        1.8855
3.0136
     37        1.8877       0.3453        1.8785
3.0824
     38        1.8805       0.3474        1.8717
3.0156
     39        1.8736       0.3494        1.8653
2.9976
     40        1.8671       0.3527        1.8591
3.0877
     41        1.8609       0.3533        1.8533
2.9834
     42        1.8551       0.3548        1.8479
2.9826
     43        1.8496       0.3567        1.8427
2.9340
     44        1.8444       0.3586        1.8378
3.0666
     45        1.8396       0.3605        1.8333
3.0049
     46        1.8350       0.3623        1.8290
2.9624
     47        1.8308       0.3651        1.8250
2.9678
     48        1.8267       0.3667        1.8212
2.9123
     49        1.8229       0.3679        1.8177
2.8755
     50        1.8193       0.3696        1.8143
2.9986
The channel was 64, the learning rate was 0.001 and the optimizer was <class
'torch.optim.sgd.SGD'>
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
```

```
      1         2.2999        0.1120          2.2971
5.2526
      2         2.2950        0.1218          2.2919
4.9220
      3         2.2895        0.1216          2.2857  4.8481
      4         2.2826        0.1310          2.2777
5.0554
      5         2.2736        0.1529          2.2669
4.8972
      6         2.2613        0.1813          2.2523
4.8617
      7         2.2446        0.2026          2.2324
5.0537
      8         2.2222        0.2200          2.2062
4.9247
      9         2.1936        0.2322          2.1742
4.9192
     10         2.1604        0.2488          2.1395
4.8985
     11         2.1265        0.2574          2.1063
5.1471
     12         2.0949        0.2722          2.0765
5.0504
     13         2.0660        0.2828          2.0491
5.0806
     14         2.0390        0.2953          2.0234
4.9728
     15         2.0135        0.3068          1.9994
4.8894
     16         1.9901        0.3146          1.9781
4.8898
     17         1.9698        0.3214          1.9601
5.0013
     18         1.9527        0.3285          1.9453
4.6879
     19         1.9386        0.3322          1.9332
4.7185
     20         1.9267        0.3350          1.9232
4.7069
     21         1.9166        0.3388          1.9145
4.8730
     22         1.9077        0.3404          1.9070
4.7222
     23         1.8998        0.3424          1.9000
4.7212
     24         1.8925        0.3440          1.8934
4.9435
     25         1.8857        0.3471          1.8872
```

```
5.0120
     26        1.8794       0.3491        1.8815
4.9071
     27        1.8734       0.3509        1.8760
5.1758
     28        1.8677       0.3522        1.8708
5.1380
     29        1.8623       0.3537        1.8656
4.9481
     30        1.8571       0.3561        1.8609
4.9800
     31        1.8521       0.3558        1.8563  5.4353
     32        1.8473       0.3578        1.8518
5.1319
     33        1.8427       0.3589        1.8472
4.9452
     34        1.8382       0.3615        1.8425
5.0308
     35        1.8337       0.3629        1.8381
4.9283
     36        1.8292       0.3650        1.8337
4.9094
     37        1.8249       0.3683        1.8296
4.8871
     38        1.8207       0.3700        1.8255
5.0013
     39        1.8166       0.3711        1.8214
4.9111
     40        1.8126       0.3715        1.8175
4.9419
     41        1.8086       0.3716        1.8135
4.9686
     42        1.8046       0.3731        1.8096
4.8710
     43        1.8008       0.3758        1.8058
4.9026
     44        1.7969       0.3763        1.8021
5.1728
     45        1.7932       0.3778        1.7985
5.2594
     46        1.7894       0.3779        1.7948
5.2541
     47        1.7857       0.3788        1.7912
5.0514
     48        1.7820       0.3803        1.7876
5.3261
     49        1.7783       0.3810        1.7841
5.0562
```

```
     50        1.7746       0.3819        1.7804
5.2772
The channel was 16, the learning rate was 0.001 and the optimizer was <class
'torch.optim.adam.Adam'>
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        1.8488       0.4405        1.5880
2.1092
      2        1.5285       0.4807        1.4754
1.4926
      3        1.4184       0.4920        1.4440
1.4509
      4        1.3420       0.5363        1.3145
1.5134
      5        1.2640       0.5557        1.2645
1.6366
      6        1.2032       0.5642        1.2354
1.4786
      7        1.1497       0.5737        1.2122
1.4639
      8        1.1042       0.5810        1.1967
1.6642
      9        1.0645       0.5827        1.1885
1.4675
     10        1.0287       0.5853        1.1848
1.5633
     11        0.9958       0.5882        1.1829
1.6633
     12        0.9635       0.5900        1.1732
1.4790
     13        0.9336       0.5959        1.1675
1.5211
     14        0.9071       0.5989        1.1675
1.5810
     15        0.8830       0.6014        1.1697  1.6498
     16        0.8628       0.5991        1.1860  1.5684
     17        0.8463       0.6043        1.1800  1.5647
Stopping since valid_loss has not improved in the last 5 epochs.
The channel was 32, the learning rate was 0.001 and the optimizer was <class
'torch.optim.adam.Adam'>
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        1.8419       0.4501        1.5560
2.7578
      2        1.4838       0.4990        1.4230
2.7789
      3        1.3575       0.5249        1.3367
2.9298
```

```
      4        1.2650       0.5562        1.2526
2.7506
      5        1.1903       0.5699        1.2085
2.7766
      6        1.1195       0.5903        1.1619
2.7301
      7        1.0474       0.6011        1.1307
2.8848
      8        0.9762       0.6080        1.1117
2.7578
      9        0.9106       0.6153        1.0970
2.7570
     10        0.8492       0.6209        1.1026  2.8909
     11        0.7926       0.6262        1.1078  2.7543
     12        0.7423       0.6199        1.1382  2.7523
     13        0.7030       0.6220        1.1651  2.6920
Stopping since valid_loss has not improved in the last 5 epochs.
The channel was 64, the learning rate was 0.001 and the optimizer was <class
'torch.optim.adam.Adam'>
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        1.7742       0.4654        1.4959
4.9542
      2        1.4214       0.5256        1.3271
5.0191
      3        1.2805       0.5337        1.3249
5.0329
      4        1.1849       0.5756        1.2055
4.9213
      5        1.0875       0.5998        1.1314
4.9105
      6        1.0037       0.6161        1.1106
5.0302
      7        0.9147       0.6194        1.1166  4.8979
      8        0.8371       0.6238        1.1196  4.8899
      9        0.7662       0.6238        1.1438  4.9137
     10        0.7078       0.6197        1.1802  5.0204
Stopping since valid_loss has not improved in the last 5 epochs.
```

Write down **validation accuracy** of your model under different hyperparameter settings. Note the validation set is automatically split by Skorch during `model.fit()`.

#channel for each layer	optimizer	SGD	Adam
16		0.3613	0.6043
32		0.3696	0.6220
64		0.3819	0.6197

1.2.2 2) Full CNN implementation (10 points)

Based on the CNN in the previous question, implement a full CNN model with max pooling layer.

- Add a max pooling layer after each convolutional layer.
- Each max pooling layer has a kernel size of 2 and a stride of 2.

Please implement this model in the following section. The hyperparameters is then be tuned and fill the results in the table. You are also required to complete the questions.

a) Implement max pooling layers Similar to the CNN implementation in previous question, implement max pooling layers.

[6]:
```
class CNN_MaxPool(nn.Module):
    def __init__(self,channels):
        super(CNN_MaxPool, self).__init__()
        # implement parameter definitions here
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=channels,␣
 ↪kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=channels, out_channels=channels, 3,␣
 ↪padding=1)
        self.conv3 = nn.Conv2d(in_channels=channels, out_channels=channels, 3,␣
 ↪padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc = nn.Linear(channels * 4 * 4, 10)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    def forward(self, images):
        # implement the forward function here
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        images = images.float()
        images = F.relu(self.conv1(images))
        images = self.pool(images)
        images = F.relu(self.conv2(images))
        images = self.pool(images)
        images = F.relu(self.conv3(images))
        images = self.pool(images)
        images = images.view(images.size(0), -1)
        images = self.fc(images)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        return images
```

b) Tune hyperparameters Based on the better optimizer found in the previous problem, we can tune the number of channels and learning rate for best validation accuracy.

[7]:

```
# implement hyperparameters, you can select and modify the hyperparameters by␣
↪yourself here.
learning_rate = [1e-4]
channel = [16, 32, 64]
# Select the better optimizer by the result shown in the previous problem, you␣
↪can select and modify it by yourself here.
better_optimizer = torch.optim.Adam

train_data_normalized = torch.Tensor(train.data/255)
train_data_normalized = train_data_normalized.permute(0,3,1,2)

for l in learning_rate:
    for c in channel:
      print(f'The channel was {c}, the learning rate was {l}')

      cnn = CNN_MaxPool(channels = c)

      model = skorch.NeuralNetClassifier(cnn, criterion=torch.nn.
↪CrossEntropyLoss,
                                   device="cuda",
                                   optimizer=better_optimizer,
                                   lr=l,
                                   max_epochs=50,
                                   batch_size=256,
                                   callbacks=[skorch.callbacks.
↪EarlyStopping(lower_is_better=True)])
      # implement input normalization & type cast here
      model.fit(train_data_normalized, torch.LongTensor(train.targets))
```

```
The channel was 16, the learning rate was 0.0001
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        2.2904       0.1483        2.2464
1.6172
      2        2.1098       0.2743        2.0010
1.1716
      3        1.9816       0.2989        1.9379
1.1853
      4        1.9221       0.3272        1.8772
1.3423
      5        1.8557       0.3562        1.8055
1.2601
      6        1.7853       0.3769        1.7474
1.2338
      7        1.7325       0.3892        1.7060
1.3598
      8        1.6936       0.4004        1.6745
```

```
1.2577
      9        1.6637       0.4078         1.6490
1.2484
     10        1.6397       0.4128         1.6277
1.2435
     11        1.6198       0.4195         1.6096
1.4338
     12        1.6027       0.4267         1.5941
1.2394
     13        1.5879       0.4324         1.5802
1.2567
     14        1.5748       0.4354         1.5680
1.2290
     15        1.5631       0.4398         1.5568
1.3180
     16        1.5525       0.4442         1.5468
1.2734
     17        1.5427       0.4501         1.5376
1.2090
     18        1.5336       0.4544         1.5290
1.3671
     19        1.5252       0.4571         1.5211
1.2738
     20        1.5173       0.4590         1.5136
1.1364
     21        1.5099       0.4618         1.5066
1.1105
     22        1.5030       0.4642         1.5000
1.1440
     23        1.4964       0.4670         1.4937
1.1733
     24        1.4901       0.4699         1.4879
1.1235
     25        1.4840       0.4728         1.4822
1.2580
     26        1.4782       0.4741         1.4766
1.1354
     27        1.4727       0.4750         1.4713
1.1519
     28        1.4674       0.4773         1.4663
1.1752
     29        1.4622       0.4794         1.4614
1.1096
     30        1.4571       0.4817         1.4566
1.1074
     31        1.4522       0.4844         1.4520
1.1333
     32        1.4475       0.4864         1.4476
```

```
1.3321
     33        1.4428       0.4872        1.4432
1.1371
     34        1.4382       0.4881        1.4389
1.1324
     35        1.4337       0.4901        1.4347
1.1373
     36        1.4293       0.4918        1.4305
1.1809
     37        1.4249       0.4931        1.4264
1.1323
     38        1.4207       0.4945        1.4225
1.1026
     39        1.4164       0.4954        1.4187
1.2285
     40        1.4123       0.4974        1.4148
1.1756
     41        1.4082       0.4988        1.4109
1.2780
     42        1.4042       0.5008        1.4071
1.1071
     43        1.4002       0.5028        1.4036
1.1156
     44        1.3963       0.5042        1.4000
1.0883
     45        1.3924       0.5058        1.3964
1.2562
     46        1.3886       0.5076        1.3929
1.3020
     47        1.3849       0.5098        1.3894
1.1344
     48        1.3812       0.5113        1.3860
1.1567
     49        1.3775       0.5121        1.3825
1.2061
     50        1.3740       0.5128        1.3792
1.1291
The channel was 32, the learning rate was 0.0001
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        2.2316       0.2693        2.0385
1.4971
      2        1.9780       0.3181        1.9140
1.4968
      3        1.8640       0.3640        1.8010
1.6392
      4        1.7552       0.3979        1.7090
1.5602
```

```
      5         1.6730        0.4178         1.6434
1.5724
      6         1.6142        0.4307         1.5934
1.5481
      7         1.5697        0.4460         1.5546
1.4835
      8         1.5350        0.4547         1.5236
1.4542
      9         1.5072        0.4621         1.4983
1.5190
     10         1.4845        0.4686         1.4776
1.5926
     11         1.4651        0.4779         1.4595
1.4701
     12         1.4482        0.4847         1.4434
1.5429
     13         1.4335        0.4879         1.4293
1.4540
     14         1.4205        0.4944         1.4168
1.4837
     15         1.4089        0.4983         1.4060
1.4929
     16         1.3986        0.5023         1.3960
1.5204
     17         1.3890        0.5060         1.3870
1.5871
     18         1.3802        0.5098         1.3787
1.4465
     19         1.3719        0.5138         1.3710
1.5009
     20         1.3642        0.5163         1.3635
1.4504
     21         1.3569        0.5181         1.3566
1.4646
     22         1.3499        0.5210         1.3501
1.5039
     23         1.3432        0.5236         1.3438
1.4552
     24         1.3367        0.5236         1.3379  1.5805
     25         1.3305        0.5253         1.3323
1.4698
     26         1.3245        0.5282         1.3267
1.5021
     27         1.3186        0.5308         1.3212
1.4621
     28         1.3128        0.5326         1.3159
1.4453
     29         1.3072        0.5350         1.3105
```

```
1.5120
     30        1.3017       0.5373        1.3052
1.4677
     31        1.2963       0.5387        1.3002
1.5693
     32        1.2910       0.5421        1.2953
1.5502
     33        1.2858       0.5435        1.2905
1.4665
     34        1.2806       0.5449        1.2857
1.4535
     35        1.2755       0.5483        1.2810
1.4499
     36        1.2704       0.5496        1.2765
1.5073
     37        1.2654       0.5503        1.2719
1.6197
     38        1.2605       0.5509        1.2673
1.7351
     39        1.2556       0.5519        1.2629
1.5843
     40        1.2508       0.5542        1.2584
1.5823
     41        1.2459       0.5563        1.2540
1.5619
     42        1.2412       0.5571        1.2497
1.4999
     43        1.2364       0.5590        1.2455
1.4913
     44        1.2317       0.5615        1.2412
1.5563
     45        1.2271       0.5629        1.2370
1.6428
     46        1.2225       0.5647        1.2329
1.4798
     47        1.2179       0.5668        1.2288
1.5729
     48        1.2134       0.5682        1.2249
1.5736
     49        1.2090       0.5688        1.2209
1.5438
     50        1.2046       0.5702        1.2171
1.3991
The channel was 64, the learning rate was 0.0001
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        2.1464       0.3092        1.9339
2.1533
```

```
       2        1.8240        0.3776        1.7582
2.1884
       3        1.6660        0.4231        1.6407
2.2465
       4        1.5776        0.4492        1.5584
2.2363
       5        1.5184        0.4701        1.4993
2.2044
       6        1.4741        0.4857        1.4553
2.2599
       7        1.4384        0.4956        1.4204
2.1999
       8        1.4078        0.5055        1.3921
2.1791
       9        1.3819        0.5169        1.3664
2.2640
      10        1.3591        0.5260        1.3448
2.1652
      11        1.3386        0.5312        1.3249
2.1339
      12        1.3199        0.5389        1.3074
2.2700
      13        1.3026        0.5459        1.2912
2.2839
      14        1.2866        0.5508        1.2763
2.2272
      15        1.2716        0.5563        1.2629
2.3535
      16        1.2576        0.5619        1.2501
2.3623
      17        1.2443        0.5662        1.2380
2.3617
      18        1.2315        0.5692        1.2270
2.2513
      19        1.2194        0.5740        1.2163
2.2911
      20        1.2079        0.5771        1.2062
2.2833
      21        1.1967        0.5806        1.1965
2.3257
      22        1.1860        0.5836        1.1873
2.3147
      23        1.1757        0.5866        1.1785
2.5167
      24        1.1657        0.5889        1.1701
2.2488
      25        1.1560        0.5918        1.1620
2.1878
```

```
     26         1.1466        0.5939         1.1541
2.3689
     27         1.1376        0.5969         1.1464
2.1535
     28         1.1288        0.5989         1.1391
2.1696
     29         1.1202        0.6021         1.1319
2.1288
     30         1.1118        0.6047         1.1246
2.2984
     31         1.1036        0.6073         1.1178
2.1223
     32         1.0955        0.6092         1.1111
2.1786
     33         1.0877        0.6107         1.1049
2.1347
     34         1.0802        0.6124         1.0987
2.1286
     35         1.0728        0.6135         1.0926
2.1851
     36         1.0656        0.6145         1.0871
2.1227
     37         1.0587        0.6163         1.0817
2.3853
     38         1.0518        0.6177         1.0793
2.1707
     39         1.0453        0.6226         1.0714
2.2874
     40         1.0389        0.6255         1.0667
2.2098
     41         1.0326        0.6275         1.0621
2.2192
     42         1.0266        0.6304         1.0577
2.1843
     43         1.0207        0.6320         1.0535
2.2130
     44         1.0149        0.6331         1.0496
2.4063
     45         1.0093        0.6348         1.0458
2.1557
     46         1.0037        0.6368         1.0420
2.1941
     47         0.9983        0.6388         1.0383
2.1705
     48         0.9931        0.6394         1.0347
2.2612
     49         0.9880        0.6414         1.0312
2.2446
```

```
          50          0.9829          0.6433          1.0277
2.2944
```

Write down the **validation accuracy** of the model under different hyperparameter settings.

#channel for each layer	validation accuracy
16	0.5128
32	0.5702
64	0.6433

For the best model you have, test it on the test set.

[8]:
```
# implement the same input normalization & type cast here
test_data_normalized = torch.Tensor(test.data/255)
test_data_normalized = test_data_normalized.permute(0,3,1,2)
test.predictions = model.predict(test_data_normalized)
sklearn.metrics.accuracy_score(test.targets, test.predictions)
```

[8]:
```
0.6438
```

How much **test accuracy** do you get? What can you conclude for the design of CNN structure and tuning of hyperparameters? (5 points)

Your Answer: 0.6438 It can be concluded that increasing the number of channels helps to improve the accuracy.

[ ]:
```

```