Pytorch homework

Instructions

- Make a copy of this notebook in your own Colab and complete the questions there.
- You can add more cells if necessary. You may also add descriptions to your code, though it is not mandatory.
- Make sure the notebook can run through by *Runtime -> Run all*. **Keep all cell outputs** for grading.
- Submit the link of your notebook
- Please **enable editing or comments** so that you can receive feedback from TAs..

link textInstall PyTorch and Skorch.

```
!pip install -q torch skorch torchvision torchtext
```

```
     |████████████████████████████████| 185 kB 35.0 MB/s
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import skorch
import sklearn
import numpy as np
import matplotlib.pyplot as plt
```

1. Tensor Operations (40 points)

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

:### 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.

```
from re import X
# x is a tensor with size being (3, 2)
x = torch.Tensor([[1, 2], [3, 4], [5, 6]])
print(x)
x = torch.unsqueeze(x, -1) # Add two new dimensions to x by using the function torch.unsqueez
x =  torch.unsqueeze(x,1)
print(x.shape)
x = torch.squeeze(x, 1)# Remove the two dimensions justed added by using the function torch.s
x = torch.squeeze(x, 2)
print(x.shape)
x = x.view(6)
print("Tensor after resize: ", x)# x is now a two-dimensional tensor, or in other words a mat
print(x.shape)
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
torch.Size([3, 1, 2, 1])
torch.Size([3, 2])
Tensor after resize:  tensor([1., 2., 3., 4., 5., 6.])
torch.Size([6])
```

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.

```
# x is a tensor with size being (3, 2)
x = torch.Tensor([[1, 2], [3, 4], [5, 6]])
x.size()
# 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
print("Stack tensors ")
z = torch.stack((x,y), 0)
print("the stack of z: ", z)
print(z.shape)
# Use torch.cat and torch.unsqueeze to generate such a z
print("concatenate tensors ")
z = torch.cat((x,y), 0)
print("the torch.cat of z: ", z)
print(z.shape)
print("to unsqueeze the data")
```

```
z = torch.unsqueeze(z, 1)
print(z.shape)
```

```
Stack tensors
the stack of z:  tensor([[[ 1.,  2.],
         [ 3.,  4.],
         [ 5.,  6.]],

        [[-1., -2.],
         [-3., -4.],
         [-5., -6.]]])
torch.Size([2, 3, 2])
concatenate tensors
the torch.cat of z:  tensor([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.],
        [-1., -2.],
        [-3., -4.],
        [-5., -6.]])
torch.Size([6, 2])
to unsqueeze the data
```

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.

```
# 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.
print("to unsqueeze the data")
x = torch.unsqueeze(x,0)
print(x.shape)

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

exp = x.expand(3, -1)
print("Expanded Tensor:", exp)
exp.size()
```

```
to unsqueeze the data
torch.Size([1, 3])
Expanded Tensor: tensor([[1., 2., 3.],
        [1., 2., 3.],
```

```
        [1., 2., 3.]])
```

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.

```
# 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 th
print("mean value")
torch.mean(x)
print(x)
k=3
x_m1 = x[k]
print(x_m1)
# Compute the sum value for each row of x.
# You need to generate a tensor x_sum of size (10).
print("Sum")
print(torch.sum(x, dim=0)) # size = [1, ncol]
# Compute the max value for each row of x.
# You need to generate a tensor x_max of size (10).
print("Max")
print(torch.max(x))
# Compute the min value for each row of x.
# You need to generate a tensor x_min of size (10).
print("Min")
print(torch.min(x))
# 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 o
x_m2 = torch.randn(10, 5)
print("x_mean value")
torch.mean(x_m2)
print(x_m2)
k=3
x_m2 = x_m2[k]
print(x_m2)
```

```
          1.2747, -0.8570, -1.1846,  0.2547, -1.1219,  0.4294,  0.8655, -0.6636,
         -0.1603, -0.9577,  1.8455, -0.0880, -0.1176,  1.4293, -1.2249,  0.5246,
          1.1397, -0.6352, -0.4408,  1.4582,  0.0493,  1.4264,  1.1370, -0.9968,
         -0.9809,  0.5528, -0.6590, -0.7737, -0.4931,  0.9309, -0.1715,  0.4593,
          0.7562, -0.7132,  0.7413, -0.2494,  0.6437, -0.2087, -0.1612,  0.8058,
         -0.2579,  0.6449],
        [ 1.3088,  1.0038, -0.7025, -0.4895,  2.2443, -0.5579,  1.0837, -0.0479,
         -1.0763,  0.0034,  0.9128, -0.6735, -0.2912,  0.9961,  0.2874, -0.8188,
          0.2087,  2.2313,  0.8242,  1.0563, -0.0789, -0.9028,  0.2103, -1.9817,
          1.2797,  0.4267, -1.0700,  1.5687,  2.1278, -0.5458,  1.8857,  0.2856,
```

```
          -0.3953, -0.8918, -0.2135,  0.0359,  0.0394, -0.5803, -0.3609,  1.2043,
          -0.4164,  0.5758, -2.1096,  1.0636,  0.1053, -0.1219,  0.4086, -1.0845,
          -0.7320,  0.3663],
         [ 0.3512,  1.4783,  1.2183,  1.8444,  0.3375, -1.2745,  1.0095,  1.0794,
          -0.2481,  0.7000,  0.6696,  1.1576, -1.0951,  2.1208,  0.7194, -0.1402,
           0.9188,  0.6327,  0.7467, -0.3581,  0.4409, -1.1285,  0.6091, -0.8172,
           0.9734,  0.0638,  0.9751,  1.6680,  0.8366,  0.6696, -0.1859, -0.4323,
           0.9329,  0.8481,  0.7597, -1.8823,  1.0147,  0.6908,  0.3829,  0.8090,
           0.3277, -1.6716, -2.7289,  0.0581, -0.3082,  1.2958,  2.1448,  0.8958,
          -1.1475, -0.0933],
         [ 0.5948, -1.1587, -2.9344,  0.5049, -1.1358,  1.8283,  1.0256, -0.0394,
           0.4751, -0.2781,  0.0601, -0.3192,  0.1615, -0.3267,  0.2448,  1.2694,
          -0.2937,  1.0967, -0.9499, -0.1533,  1.0365,  0.3014,  0.5981,  0.7578,
           0.7062, -0.7812,  1.6604, -1.1127,  1.1442, -0.6625, -0.0687,  0.2210,
           0.1890,  1.9474,  0.8613, -1.0991,  1.2178, -1.4973,  0.0035,  0.6927,
          -0.1973,  0.3492,  0.1804, -0.6620, -0.8443, -0.8268, -0.8110, -1.7725,
          -1.8442, -0.0199]])
tensor([ 1.4250, -0.7966, -1.3609,  0.0917,  0.1699, -0.7673, -0.0090,  0.6805,
         0.2873, -0.5812, -1.4305, -0.5231, -0.2142, -0.0890,  1.7204,  0.9599,
         0.4978, -2.2911, -0.0957, -1.3734,  2.8082,  0.7507,  1.4813, -0.3913,
        -0.3131, -0.0203, -0.7155,  1.0847,  0.0779,  1.0867,  1.0420,  0.1514,
        -1.0449, -0.4607,  1.7630,  0.6522,  1.2171, -0.2363,  0.0608, -0.1744,
        -0.5854,  0.8131, -0.3207,  0.3327,  1.3757, -0.4524, -0.5721, -1.1262,
         0.8515,  1.1427])
Sum
tensor([ 3.7148, -2.3317, -8.4673,  1.3965,  2.8886, -3.0052,  5.5236,  3.8121,
         3.0576,  3.2446, -3.9734, -0.3218, -2.1200,  3.3347, -1.1085,  0.7441,
         3.9501, -3.2509,  4.7032, -2.1320,  0.2556,  2.8574,  2.9922, -0.2812,
         4.3808, -3.5058,  2.4437,  3.2345,  0.6644,  2.2934, -1.0142, -2.4227,
        -1.6704,  1.0953,  1.8204, -2.6881,  3.6140, -4.1130, -2.7999,  1.2580,
         0.1238, -4.0891, -4.6261, -1.4096, -0.0595,  0.1067,  2.3059, -4.8223,
        -1.4473,  3.8437])
Max
tensor(2.8082)
Min
tensor(-2.9344)
x_mean value
tensor([[ 1.0170, -0.4079, -0.0831,  0.7348,  1.2474],
        [-0.9020,  0.9491, -0.5279,  0.0155, -0.3324],
        [ 0.5718, -0.2169, -0.3735,  0.7183, -0.9384],
        [ 1.4046, -0.0272,  0.1702, -0.7667,  0.5975],
        [-1.2922,  0.2033,  1.4364, -1.4252,  0.1074],
        [ 0.6342, -0.6147, -1.2083,  0.2791,  0.5005],
        [-0.2909, -0.2979, -0.5955,  0.5165, -0.1222],
        [ 0.0982, -1.1251, -1.5430, -1.2905, -1.0654],
        [-0.3147,  0.1634, -1.0148, -0.1135, -1.5575],
        [-0.5015,  1.1539, -0.6104, -0.1824,  0.7834]])
tensor([ 1.4046, -0.0272,  0.1702, -0.7667,  0.5975])
```

Install PyTorch and Skorch.

```
!pip install -q torch skorch torchvision torchtext
```

```
        |████████████████████████████████| 185 kB 23.7 MB/s
```

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.

```
# importing required libraries
import torchvision
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import skorch
from skorch.helper import predefined_split
from torch.nn import Linear, ReLU, CrossEntropyLoss, Sequential, Conv2d, MaxPool2d, Module, S
import numpy as np
```

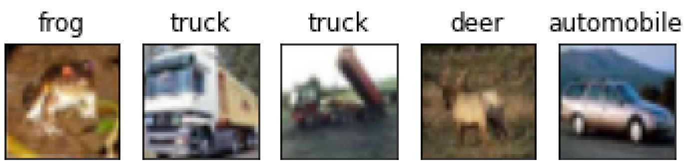
```
train = torchvision.datasets.CIFAR10("./data", train=True, download=True)
test = torchvision.datasets.CIFAR10("./data", train=False, download=True)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-
100%                          170498071/170498071 [00:02<00:00, 104748438.53it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

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

```
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) 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. You will need to tune the hyperparameters and fill the results in the table.

a) Implement convolutional layers

Implement the initialization function and the forward function of the CNN.

```
class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    # implement parameter definitions here
    self.cnn_layers = Sequential(
            Conv2d(3, 4, kernel_size=3, stride=1, padding=1),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2), #Adding max pooling also

            # Defining another 2D convolution layer
            Conv2d(4, 4, kernel_size=3, stride=1, padding=1),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),

            # Defining third 2D convolution layer
            Conv2d(4, 4, kernel_size=3, stride=1, padding=1),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2)

        )
    self.linear_layers = Sequential(
            Linear(64, 10),
            Sigmoid()
```

```
        )

    def forward(self, images):
      # implement the forward function here
      x = self.cnn_layers(images)
      x = x.view(x.size(0), -1)
      x = self.linear_layers(x)
      return x
```

b) Tune hyperparameters

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

```
# implement hyperparameters here
model = skorch.NeuralNetClassifier(CNN, criterion=torch.nn.CrossEntropyLoss,
                                   optimizer = torch.optim.SGD, lr = 0.02, max_epochs = 80)
# implement input normalization & type cast here
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
train_y = np.array(train.targets)
#train_y = torch.from_numpy(train_y)

train_y_onehot = np.zeros((train_y.size, train_y.max() + 1))
train_y_onehot[np.arange(train_y.size), train_y] = 1

train_x = np.array([[img[:,:,0],img[:,:,1],img[:,:,2]] for img in train.data]).astype(float)

model.fit(torch.from_numpy(np.array(train_x, dtype = np.float32)), torch.from_numpy(train_y))
```

```
  epoch    train_loss    valid_acc    valid_loss     dur
-------  ------------  -----------  ------------  ------
      1        2.2564       0.1333        2.2229  7.6228
      2        2.2117       0.1569        2.2027  7.5643
      3        2.1730       0.2140        2.1594  7.6917
      4        2.1352       0.2475        2.1164  7.4184
      5        2.0722       0.3020        2.0506  7.7827
      6        2.0272       0.2980        2.0506  7.2984
      7        2.0090       0.3370        2.0130  7.3981
      8        1.9965       0.3530        1.9931  7.6817
      9        1.9888       0.3574        1.9945  7.8385
     10        1.9812       0.3549        1.9868  7.6940
     11        1.9752       0.3469        1.9929  7.5437
     12        1.9696       0.3586        1.9840  7.5971
     13        1.9642       0.3756        1.9667  7.7023
     14        1.9619       0.3817        1.9650  7.5391
     15        1.9588       0.3800        1.9606  7.8981
     16        1.9554       0.3796        1.9597  7.5926
     17        1.9533       0.3425        2.0062  7.5780
     18        1.9512       0.3598        1.9766  8.3165
```

```
     19        1.9479      0.3799        1.9641  8.3424
     20        1.9461      0.3670        1.9700  7.7122
     21        1.9440      0.3788        1.9521  7.8264
     22        1.9423      0.3934        1.9426  7.9183
     23        1.9404      0.3579        1.9905  7.6476
     24        1.9393      0.3814        1.9571  7.7644
     25        1.9364      0.3780        1.9508  7.5503
     26        1.9387      0.3446        1.9856  7.7931
     27        1.9363      0.3906        1.9464  7.7912
     28        1.9372      0.3764        1.9508  7.7820
     29        1.9351      0.3921        1.9362  7.6377
     30        1.9356      0.3687        1.9623  7.5101
     31        1.9345      0.3806        1.9463  7.7982
     32        1.9342      0.3808        1.9479  7.8631
     33        1.9326      0.3739        1.9534  7.4842
     34        1.9331      0.3849        1.9476  7.6901
     35        1.9303      0.3918        1.9329  7.7818
     36        1.9319      0.3743        1.9486  7.7786
     37        1.9302      0.3704        1.9485  7.6469
     38        1.9323      0.3787        1.9399  7.8281
     39        1.9295      0.3853        1.9358  7.7532
     40        1.9280      0.3635        1.9554  7.5893
     41        1.9273      0.3751        1.9494  7.6380
     42        1.9289      0.3937        1.9327  7.7828
     43        1.9289      0.3324        1.9669  7.7377
     44        1.9292      0.3607        1.9508  7.5954
     45        1.9298      0.3328        1.9712  7.7538
     46        1.9299      0.3709        1.9409  7.8036
     47        1.9310      0.3483        1.9646  7.7836
     48        1.9304      0.3610        1.9529  7.7242
     49        1.9280      0.3776        1.9306  7.7271
     50        1.9300      0.3577        1.9457  7.7726
     51        1.9292      0.3389        1.9560  7.6273
     52        1.9289      0.3482        1.9424  7.4246
     53        1.9284      0.3339        1.9471  7.8989
     54        1.9290      0.3600        1.9351  7.5079
     55        1.9296      0.3497        1.9406  7.5861
```

Validation loss obtained using SGD optimizer for 50 epochs is 1.9457 Validation loss obtained using SGD optimizer for 80 epochs is 1.9392

```
# implement hyperparameters here
model = skorch.NeuralNetClassifier(CNN, criterion=torch.nn.CrossEntropyLoss,
                                   optimizer = torch.optim.Adam, lr = 0.2, max_epochs = 50)
# implement input normalization & type cast here
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
train_y = np.array(train.targets)
#train_y = torch.from_numpy(train_y)

train_y_onehot = np.zeros((train_y.size, train_y.max() + 1))
train_y_onehot[np.arange(train_y.size), train_y] = 1
```

```
train_x = np.array([[img[:,:,0],img[:,:,1],img[:,:,2]] for img in train.data]).astype(float)

model.fit(torch.from_numpy(np.array(train_x, dtype = np.float32)), torch.from_numpy(train_y))
```

epoch	train_loss	valid_acc	valid_loss	dur
1	2.3616	0.1000	2.3612	8.6093
2	2.3612	0.1000	2.3612	9.9424
3	2.3612	0.1000	2.3612	8.3205
4	2.3612	0.1000	2.3612	7.8114
5	2.3612	0.1000	2.3612	7.7383
6	2.3612	0.1000	2.3612	7.7371
7	2.3612	0.1000	2.3612	7.6811
8	2.3612	0.1000	2.3612	7.6484
9	2.3612	0.1000	2.3612	7.7050
10	2.3612	0.1000	2.3612	7.9149
11	2.3612	0.1000	2.3612	7.7035
12	2.3612	0.1000	2.3612	8.2284
13	2.3612	0.1000	2.3612	7.7088
14	2.3612	0.1000	2.3612	7.6084
15	2.3612	0.1000	2.3612	7.7143
16	2.3612	0.1000	2.3612	7.8593
17	2.3612	0.1000	2.3612	7.6511
18	2.3612	0.1000	2.3612	7.8716
19	2.3612	0.1000	2.3612	10.9168
20	2.3612	0.1000	2.3612	8.3139
21	2.3612	0.1000	2.3612	8.2782
22	2.3612	0.1000	2.3612	9.4205
23	2.3612	0.1000	2.3612	8.5033
24	2.3612	0.1000	2.3612	7.8255
25	2.3612	0.1000	2.3612	8.9048
26	2.3612	0.1000	2.3612	9.9808
27	2.3612	0.1000	2.3612	9.4702
28	2.3612	0.1000	2.3612	9.6823
29	2.3612	0.1000	2.3612	10.1683
30	2.3612	0.1000	2.3612	9.4410
31	2.3612	0.1000	2.3612	10.5238
32	2.3612	0.1000	2.3612	12.1579
33	2.3612	0.1000	2.3612	8.7332
34	2.3612	0.1000	2.3612	8.1706
35	2.3612	0.1000	2.3612	8.3132
36	2.3612	0.1000	2.3612	8.1552
37	2.3612	0.1000	2.3612	8.2883
38	2.3612	0.1000	2.3612	8.0469
39	2.3612	0.1000	2.3612	8.2140
40	2.3612	0.1000	2.3612	8.2633
41	2.3612	0.1000	2.3612	8.2459
42	2.3612	0.1000	2.3612	8.1746
43	2.3612	0.1000	2.3612	8.2453
44	2.3612	0.1000	2.3612	8.1371
45	2.3612	0.1000	2.3612	8.7439
46	2.3612	0.1000	2.3612	8.3979
47	2.3612	0.1000	2.3612	8.2403
48	2.3612	0.1000	2.3612	8.2722
49	2.3612	0.1000	2.3612	8.1669
50	2.3612	0.1000	2.3612	8.1890

```
<class 'skorch.classifier.NeuralNetClassifier'>[initialized](
  module_=CNN(
    (cnn_layers): Sequential(
      (0): Conv2d(3, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
```

Adam optimizer could achieve 2.3612

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

Hint: You may need more epochs for SGD than Adam.

#channel for each layer \ optimizer	SGD	Adam
(128, 128, 128)		
(256, 256, 256)		
(512, 512, 512)		

c) Use larger CNN model

> Indented block

Add more Convolution/BatchNorm/Pooling/DropOut/Linear layers to improve the accuracy.

```
class CNNLarge(nn.Module):
  def __init__(self):
    super(CNNLarge, self).__init__()
    # implement parameter definitions here
    self.cnn_layers = Sequential(
            Conv2d(3, 4, kernel_size=3, stride=1, padding=1),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2), #Adding max pooling also

            # Defining another 2D convolution layer
            Conv2d(4, 4, kernel_size=3, stride=1, padding=1),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),

            # Defining third 2D convolution layer
            Conv2d(4, 8, kernel_size=3, stride=1, padding=1),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),

            # Defining fourth 2D convolution layer
            Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2),
```

```
            # Defining fifth 2D convolution layer
            Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            ReLU(inplace=True),
            MaxPool2d(kernel_size=2, stride=2)

        )
    self.linear_layers = Sequential(
            Linear(32, 10)
        )

  def forward(self, images):
    # implement the forward function here
    x = self.cnn_layers(images)
    x = x.view(x.size(0), -1)
    x = self.linear_layers(x)
    return x

# implement hyperparameters here
model = skorch.NeuralNetClassifier(CNNLarge, criterion=torch.nn.CrossEntropyLoss,
                                   optimizer = torch.optim.SGD, lr = 0.02, max_epochs = 50)
# implement input normalization & type cast here
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
train_y = np.array(train.targets)
#train_y = torch.from_numpy(train_y)

train_y_onehot = np.zeros((train_y.size, train_y.max() + 1))
train_y_onehot[np.arange(train_y.size), train_y] = 1

train_x = np.array([[img[:,:,0],img[:,:,1],img[:,:,2]] for img in train.data]).astype(float)

model.fit(torch.from_numpy(np.array(train_x, dtype = np.float32)), torch.from_numpy(train_y))
```

```
  epoch    train_loss    valid_acc    valid_loss      dur
-------  ------------  -----------  ------------  -------
      1        2.2608       0.2042        2.1160   8.4619
      2        1.9980       0.2766        1.9392   8.4601
      3        1.8505       0.3473        1.7859   8.2478
      4        1.7526       0.3628        1.7747   8.3604
      5        1.6938       0.3786        1.6956   8.5037
      6        1.6556       0.3923        1.6679   8.3975
      7        1.6272       0.4118        1.6227   8.4142
      8        1.6061       0.4194        1.5975   8.5926
      9        1.5875       0.4146        1.5976   9.4106
     10        1.5706       0.4283        1.5837   8.5247
     11        1.5544       0.4286        1.5701   8.5465
     12        1.5415       0.3656        1.7803   8.6729
     13        1.5263       0.4024        1.6732  11.3805
     14        1.5180       0.4232        1.5944  11.6643
     15        1.5050       0.4184        1.6173  11.9321
     16        1.4906       0.4324        1.5705  10.6959
     17        1.4830       0.4374        1.5637   9.7695
     18        1.4742       0.4401        1.5571   9.1306
```

```
     19        1.4693       0.4411        1.5321  9.2395
     20        1.4623       0.4429        1.5604  8.4874
     21        1.4520       0.4378        1.5846  8.6886
     22        1.4475       0.4306        1.6232  8.7709
     23        1.4356       0.4464        1.5482  8.8958
     24        1.4340       0.4372        1.5922  8.5705
     25        1.4292       0.4603        1.5276  8.5893
     26        1.4172       0.4540        1.5488  9.2090
     27        1.4140       0.4513        1.5469  8.7739
     28        1.4109       0.4559        1.5338  8.7279
     29        1.4060       0.4597        1.5139  8.6909
     30        1.3977       0.4374        1.5963  8.5998
     31        1.3965       0.4642        1.5009  8.5772
     32        1.3926       0.4575        1.5342  8.5082
     33        1.3864       0.4620        1.5183  8.4655
     34        1.3873       0.4774        1.4784  9.7116
     35        1.3794       0.4670        1.4861  8.8878
     36        1.3757       0.4695        1.4838  11.4243
     37        1.3697       0.4779        1.4915  8.7505
     38        1.3688       0.4635        1.5149  8.7212
     39        1.3695       0.4661        1.4914  8.5417
     40        1.3608       0.4493        1.5576  8.5625
     41        1.3597       0.4741        1.4798  8.6888
     42        1.3676       0.4820        1.4628  8.6457
     43        1.3596       0.4723        1.4801  8.5798
     44        1.3559       0.4724        1.4850  8.6249
     45        1.3505       0.4763        1.4725  8.6494
     46        1.3485       0.4780        1.4737  8.6372
     47        1.3464       0.4834        1.4576  8.6884
     48        1.3438       0.4827        1.4550  8.7124
     49        1.3405       0.4620        1.5084  8.6828
     50        1.3427       0.4775        1.4682  8.5796
<class 'skorch.classifier.NeuralNetClassifier'>[initialized](
  module_=CNNLarge(
    (cnn_layers): Sequential(
      (0): Conv2d(3, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
```

Using a larger architecture, I could obtain much better loss of 1.4682. This clearly proves that this problem requires complexity and using bigger models can give better results