

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.

1.0.1 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.

1.0.2 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.

1.1 How can I learn PyTorch?

Justin Johnson has made an excellent [tutorial \(https://github.com/jcjohnson/pytorch-examples\)](https://github.com/jcjohnson/pytorch-examples) for PyTorch.

You can also find the detailed [API doc \(http://pytorch.org/docs/stable/index.html\)](http://pytorch.org/docs/stable/index.html) here. If you have other questions that are not addressed by the API docs, the [PyTorch forum \(https://discuss.pytorch.org/\)](https://discuss.pytorch.org/) is a much better place to ask than StackOverflow.

Install PyTorch and Skorch.

```
In [1]: 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
```

1.2 0. Tensor Operations (5 points)


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

1.2.1 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](https://pytorch.org/docs/stable/torch.html#torch.squeeze) (<https://pytorch.org/docs/stable/torch.html#torch.squeeze>), [torch.unsqueeze](https://pytorch.org/docs/stable/torch.html#torch.unsqueeze) (<https://pytorch.org/docs/stable/torch.html#torch.unsqueeze>) and [torch.Tensor.view](https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view) (<https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view>). Please read the documents of the functions, and finish the following practice.

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

x.shape
# Add two new dimensions to x by using the function torch.unsqueeze, so that the size
x = x.unsqueeze(1).unsqueeze(-1)
print(x.shape)
# Remove the two dimensions just added by using the function torch.squeeze, and cha
x = x.squeeze(1).squeeze(-1)
print(x.shape)
# x is now a two-dimensional tensor, or in other words a matrix. Now use the function
x = x.view(6)
print(x.shape)
```



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

1.2.2 2) Tensor concatenation and stack

Tensor concatenation and stack are operations to combine small tensors into big tensors. The corresponding functions are [torch.cat](https://pytorch.org/docs/stable/torch.html#torch.cat) (<https://pytorch.org/docs/stable/torch.html#torch.cat>) and [torch.stack](https://pytorch.org/docs/stable/torch.html#torch.stack) (<https://pytorch.org/docs/stable/torch.html#torch.stack>). Please read the documents


of the functions. and finish the following practice.

```
In [3]: # 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,:,:]

# Use torch.stack to generate such a z
z = torch.stack([x,y])  ##stack can combine two tensor at the new dimentstion
print(z[0,:,:])
# Use torch.cat and torch.unsqueeze to generate such a z
z = torch.cat([x.unsqueeze(0), y.unsqueeze(0)], dim=0)  ## by adding a new dimension
print(z[1,:,:])
```



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

1.2.3 3) Tensor expansion

Tensor expansion is to expand a tensor into a larger tensor along singleton dimensions. The corresponding functions are [torch.Tensor.expand](https://pytorch.org/docs/stable/tensors.html#torch.Tensor.expand) (<https://pytorch.org/docs/stable/tensors.html#torch.Tensor.expand>) and [torch.Tensor.expand_as](https://pytorch.org/docs/stable/tensors.html#torch.Tensor.expand_as) (https://pytorch.org/docs/stable/tensors.html#torch.Tensor.expand_as). Please read the documents of the functions, and finish the following practice.

```
In [4]: # 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,:,:]

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

# [TO DO]
# Then expand the new tensor to the target tensor by using torch.Tensor.expand.
z = x.expand(2, -1)  ## expand twice at the the first dimension and hold the origina
print(z.shape)
# print(z[0,:]) verify the result
# print(z[1,:])
```

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

1.2.4 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](https://pytorch.org/docs/stable/torch.html#torch.mean) (<https://pytorch.org/docs/stable/torch.html#torch.mean>), [torch.sum](https://pytorch.org/docs/stable/torch.html#torch.sum) (<https://pytorch.org/docs/stable/torch.html#torch.sum>), [torch.max](https://pytorch.org/docs/stable/torch.html#torch.max) (<https://pytorch.org/docs/stable/torch.html#torch.max>), [torch.min](https://pytorch.org/docs/stable/torch.html#torch.min) (<https://pytorch.org/docs/stable/torch.html#torch.min>), [torch.topk](https://pytorch.org/docs/stable/torch.html#torch.topk) (<https://pytorch.org/docs/stable/torch.html#torch.topk>), and finish the following practice.

```
In [5]: # x is a random tensor with size being (10, 50)
x = torch.randn(10, 50)

##for all these function we just need to handle with the first dimension
# 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
x_mean = torch.mean(x, 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).
x_sum = torch.sum(x, 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).
x_max, _ = torch.max(x, 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).
x_min, _ = torch.min(x, dim=1)
print(x_min.shape)

# Compute the top-5 values for each row of x.
# You need to generate a tensor x_top of size (10, 5), and x_top[k, :] is the top-5 values
x_5top, _ = torch.topk(x, 5, dim=1)
print(x_5top.shape)

tensor(-0.2006)
torch.Size([10])
torch.Size([10])
torch.Size([10])
torch.Size([10, 5])
```

1.3 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.

```
In [6]: 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.

```
In [7]: 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)
```



1.3.1 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.

1.3.1.1 a) Implement convolutional layers (10 Points)

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

```
In [8]: 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)*****
    ## 3 (rgb three dimension) * 32 * 32
    self.conv1 = nn.Conv2d(3, channels, kernel_size=3, padding=1)    ##(3-1)/2 + 1 = 1
    self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)    ##similar
    self.conv3 = nn.Conv2d(channels, channels, kernel_size=3, padding=1)    ##similar
    self.fc = nn.Linear(channels * 32 * 32, 10)    ##10 categories
    # *****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)*****
    x = F.relu(self.conv1(images))
    ##print("x_1", x.shape)
    x = F.relu(self.conv2(x))
    ##print("x_3", x.shape)
    x = F.relu(self.conv3(x))
    ##print("x_5", x.shape)
    x = x.view(x.size(0), -1)
    ##print("x_6", x.shape)
    x = self.fc(x)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return x    ##image after processing
```

1.3.1.2 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.

```
In [14]: # implement hyperparameters, you can select and modify the hyperparameters by yourself

##paras = [[torch.optim.SGD, 1e-3, 128],[torch.optim.Adam, 1e-3, 128],[torch.optim.Ad
optimize = [torch.optim.SGD, torch.optim.Adam]
learning_rate = 1e-3
channel = [128, 256, 64]

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

for o in optimize:
    for c in channel:
        print(f'The channel was {c}, the learning rate was {learning_rate} and the opti
        cnn = CNN(channels = c)

        model = skorch.NeuralNetClassifier(cnn, criterion=torch.nn.CrossEntropyLoss,
                                           device="cuda",
                                           optimizer=o,
                                           # optimizer__momentum=0.90,
                                           lr=learning_rate,
                                           max_epochs=25,
                                           batch_size=64,
                                           callbacks=[skorch.callbacks.EarlyStopping(lower_is

        # implement input normalization & type cast here
        model.fit(train_data_normalized, np.asarray(train.targets, dtype= np.int64))
```

The channel was 128, the learning rate was 0.001 and the optimizer was <class 'torch.optim.sgd.SGD'>

epoch	train_loss	valid_acc	valid_loss	dur
1	2.2399	0.2403	2.1277	18.6095
2	2.0169	0.3126	1.9395	18.4208
3	1.9055	0.3410	1.8715	18.5584
4	1.8522	0.3574	1.8254	18.6854
5	1.8116	0.3726	1.7888	18.6897
6	1.7757	0.3842	1.7537	18.6787
7	1.7400	0.3984	1.7169	18.5943
8	1.7043	0.4136	1.6815	18.5994
9	1.6733	0.4230	1.6540	18.6909
10	1.6475	0.4293	1.6315	18.7263
11	1.6241	0.4372	1.6104	18.7275
12	1.6014	0.4433	1.5896	18.6507
13	1.5794	0.4504	1.5693	18.7153
14	1.5578	0.4556	1.5491	18.8290
15	1.5367	0.4618	1.5295	20.0487
16	1.5168	0.4688	1.5108	21.8507

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
128	0.5059	0.5835
256	0.5340	0.5827
64	0.4977	0.5981

1.3.2 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.

1.3.2.1 a) Implement max pooling layers

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

```
In [10]: 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)*****
    ## 3 (rgb three dimension) * 32 * 32
    self.conv1 = nn.Conv2d(3, channels, kernel_size=3, padding=1)      ##(3-1)/2 + 1 = 1
    self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, padding=1) ##similar
    self.conv3 = nn.Conv2d(channels, channels, kernel_size=3, padding=1) ##similar
    self.fc = nn.Linear(channels * 4 * 4, 10)                       ##10 categories
    # *****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)*****
    x = F.relu(self.conv1(images))
    ##print("x_1", x.shape)
    x = F.max_pool2d(x, kernel_size=2, stride = 2)
    ##print("x_2", x.shape)
    x = F.relu(self.conv2(x))
    ##print("x_3", x.shape)
    x = F.max_pool2d(x, kernel_size=2, stride = 2)
    ##print("x_4", x.shape)
    x = F.relu(self.conv3(x))
    x = F.max_pool2d(x, kernel_size=2, stride = 2)
    ##print("x_5", x.shape)
    x = x.view(x.size(0), -1)
    ##print("x_6", x.shape)
    x = self.fc(x)
    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return x ##image after processing
```

1.3.2.2 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.


```
In [12]: # implement hyperparameters, you can select and modify the hyperparameters by yourself
learning_rate = 1e-3
channel = [128, 256, 512]
# Select the better optimizer by the result shown in the previous problem, you can se
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 c in channel:
    print(f'The channel was {c}, the learning rate was {learning_rate}')

    cnn = CNN_MaxPool(channels = c)

    model = skorch.NeuralNetClassifier(cnn, criterion=torch.nn.CrossEntropyLoss,
                                      device="cuda",
                                      optimizer=better_optimizer,
                                      lr=learning_rate,
                                      max_epochs=25,
                                      batch_size=64,
                                      callbacks=[skorch.callbacks.EarlyStopping(lower_is_

# implement input normalization & type cast here
model.fit(train_data_normalized, np.asarray(train.targets, dtype= np.int64))
```

The channel was 128, the learning rate was 0.001

epoch	train_loss	valid_acc	valid_loss	dur
1	1.6302	0.5116	1.3833	6.8544
2	1.2627	0.5905	1.1546	6.6204
3	1.0903	0.6371	1.0228	6.6239
4	0.9807	0.6520	0.9703	6.6118
5	0.9036	0.6672	0.9552	6.7928
6	0.8479	0.6753	0.9477	6.6467
7	0.8029	0.6704	0.9713	6.6586
8	0.7666	0.6775	0.9315	6.6561
9	0.7329	0.6874	0.9154	6.6624
10	0.7052	0.6929	0.9034	6.6734
11	0.6797	0.7030	0.8787	6.6715
12	0.6563	0.7043	0.8751	6.6505
13	0.6352	0.7102	0.8741	6.6705
14	0.6161	0.7083	0.8801	6.6648
15	0.5989	0.7032	0.8967	6.6386
16	0.5843	0.6981	0.9185	6.6706
17	0.5720	0.7022	0.9346	6.6751

Stopping since valid_loss has not improved in the last 5 epochs.

The channel was 256, the learning rate was 0.001

epoch	train_loss	valid_acc	valid_loss	dur
1	1.5570	0.5630	1.2261	15.3096
2	1.1413	0.6396	1.0208	15.3128
3	0.9681	0.6682	0.9453	15.3371
4	0.8629	0.6898	0.8897	15.3621
5	0.7843	0.6953	0.8873	15.3246
6	0.7196	0.6979	0.8989	15.3408
7	0.6671	0.6908	0.9296	15.3628
8	0.6231	0.7050	0.9051	15.3356
9	0.5850	0.7120	0.8980	15.3359

Stopping since valid_loss has not improved in the last 5 epochs.

The channel was 512, the learning rate was 0.001

epoch	train_loss	valid_acc	valid_loss	dur
1	1.5001	0.5907	1.1893	37.0213
2	1.0907	0.6525	0.9865	37.0427
3	0.9130	0.6784	0.9127	37.0198
4	0.7916	0.6939	0.8747	37.0396
5	0.7010	0.7024	0.8819	37.0127
6	0.6302	0.6868	0.9553	37.0334
7	0.5742	0.6899	0.9898	37.0341
8	0.5188	0.6913	1.0638	37.0439

Stopping since valid_loss has not improved in the last 5 epochs.

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

#channel for each layer	validation accuracy
128	0.7022
256	0.7120
512	0.6913

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

```
In [16]: # implement the same input normalization & type cast here

##define the best model parameters to train again and get the test accuracy
train_data_normalized = torch.Tensor(train.data/255)
train_data_normalized = train_data_normalized.permute(0, 3, 1, 2)
cnn = CNN_MaxPool(channels = 256)
model = skorch.NeuralNetClassifier(cnn, criterion=torch.nn.CrossEntropyLoss,
                                   device="cuda",
                                   optimizer=torch.optim.Adam ,
                                   lr=1e-3,
                                   max_epochs=25,
                                   batch_size=64,
                                   callbacks=[skorch.callbacks.EarlyStopping(lower_is_

# implement input normalization & type cast here
model.fit(train_data_normalized, np.asarray(train.targets, dtype= np.int64))
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)

Out[16]: 0.7041
```

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: The test accuracy I get is 0.7041.

The conclusions in this hw I get are as follows.

Using pooling layer in the CNN process can help training CNN model to be faster and better.

And when tuning of hyperparameters, set `optimizer=torch.optim.Adam` in the model setting can have a better model.

Having more channels for each layer can have a more complex model to get a better performace with longer training time but too much channels may lead to overfitting problem and then lead to a worse validation accuracy.