CNN

December 20, 2020

[4]:
```
%matplotlib inline

import torch
import torchvision
import torchvision.transforms as transforms

import matplotlib.pyplot as plt
import numpy as np

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import random
import numpy as np

import os

from shutil import copy2
```

[5]:
```
in_submission = os.path.exists('/flags/isgrader.flag')
perform_computation = not in_submission

if in_submission:
    assert os.path.exists('./cifar_net.pth'), 'The trained network for CIFAR␣
 ↪was not stored properly. ' + \
                                              'Please read and follow the␣
 ↪instructions/important notes.'

    assert os.path.exists('./mnist_net.pth'), 'The trained network for MNIST␣
 ↪was not stored properly. ' + \
                                              'Please read and follow the␣
 ↪instructions/important notes.'

    copy2('./cifar_net.pth', './cifar_net_submitted.pth')
    copy2('./mnist_net.pth', './mnist_net_submitted.pth')
```

1 *Assignment Summary

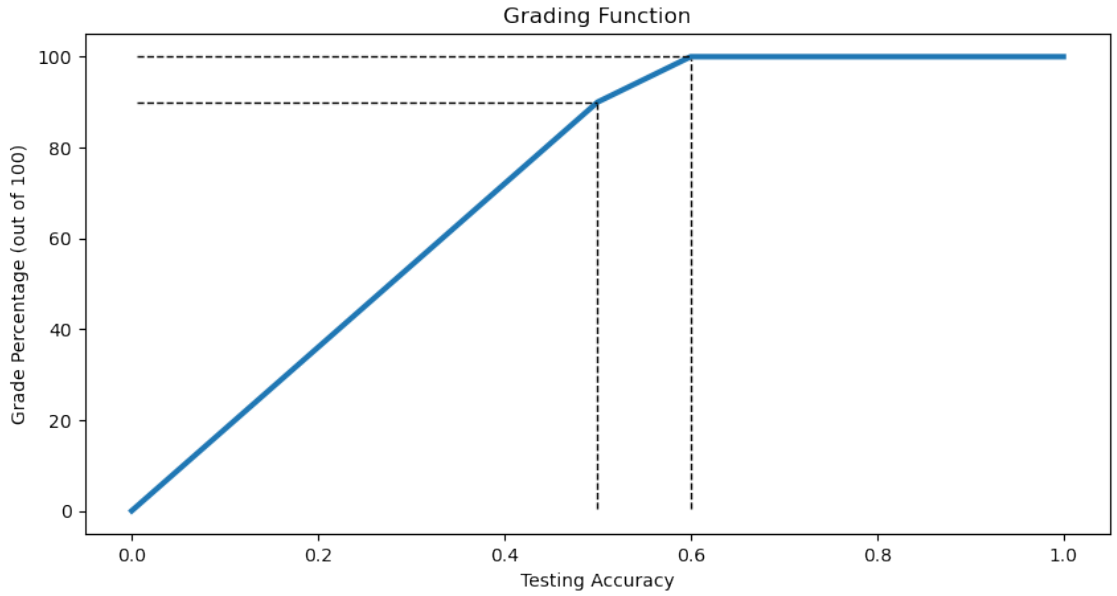
Go through the CIFAR-10 tutorial at https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html, and ensure you can run the code. Modify the architecture that is offered in the CIFAR-10 tutorial to get the best accuracy you can. Anything better than about 93.5% will be comparable with current research.

Redo the same efforts for the MNIST digit data set.

Procedural Instructions:

This assignment is less guided than the previous assignments. You are supposed to train a deep convolutional classifier, and store it in a file. The autograder will load the trained model, and test its accuracy on a hidden test data set. Your classifier's test accuracy will determine your grade for each part according to the following model.

[6]:
```
fig, ax = plt.subplots(1, 1, figsize=(10, 5), dpi=100)
ax.plot([0., 0.5, 0.6, 1.], [0., 90., 100., 100.], lw=3)
ax.axhline(y=90, xmin=0.05, xmax=.5, lw=1, ls='--', c='black')
ax.axvline(x=0.5, ymin=0.05, ymax=.86, lw=1, ls='--', c='black')
ax.axhline(y=100, xmin=0.05, xmax=.59, lw=1, ls='--', c='black')
ax.axvline(x=0.6, ymin=0.05, ymax=.95, lw=1, ls='--', c='black')
ax.set_xlabel('Testing Accuracy')
ax.set_ylabel('Grade Percentage (out of 100)')
ax.set_title('Grading Function')
None
```



2 Important Notes

You **should** read these notes before starting as these notes include crucial information about what is expected from you.

1. **Use Pytorch**: The autograder will only accept pytorch models.

- Pytorch's CIFAR-10 tutorial at https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html is the best starting point for this assignment. However, we will not prohibit using or learning from any other tutorial you may find online.

2. **No Downloads**: The coursera machines are disconnected from the internet. We already have downloaded the pytorch data files, and uploaded them for you. You will need to disable downloading the files if you're using data collector APIs such as `torchvision.datasets`.

- For the CIFAR data, you should provide the `root='/home/jovyan/work/course-lib/data_cifar', download=False` arguments to the `torchvision.datasets.CIFAR10` API.
- For the MNIST data, you should provide the `root='/home/jovyan/work/course-lib/data_mnist', download=False` arguments to the `torchvision.datasets.MNIST` API.

3. **Store the Trained Model**: The autograder can not and will not retrain your model. You are supposed to train your model, and then store your best model with the following names:

- The CIFAR classification model must be stored at `./cifar_net.pth`.
- The MNIST classification model must be stored at `./mnist_net.pth`.
- Do not place these file under any newly created directory.
- The trained model may **not exceed 1 MB** in size.

4. **Model Class Naming**: The neural models in the pytorch library are subclasses of the `torch.nn.Module` class. While you can define any architecture as you please, your `torch.nn.Module` must be named `Net` exactly. In other words, you are supposed to have the following lines somewhere in your network definition:

```
import torch.nn as nn
class Net(nn.Module):
    ...
```

5. **Grading Reference Pre-processing**: We will use a specific randomized transformation for grading that can be found in the `Autograding and Final Tests` section. Before training any model for long periods of time, you need to pay attention to the existence of such a testing pre-processing.

6. **Training Rules**: You are able to make the following decisions about your model:

- You **can** choose and change your architecture as you please.
  - You can have shallow networks, or deep ones.
  - You can customize the number of neural units in each layer and the depth of the network.
  - You are free to use convolutional, and non-convolutional layers.
  - You can employ batch normalization if you would like to.
  - You can use any type of non-linear layers as you please. `Tanh`, `Sigmoid`, and `ReLU` are some common activation functions.
  - You can use any kind of pooling layers you deem appropriate.
  - etc.

- You **can** initialize your network using any of the methods described in `https://pytorch.org/docs/stable/nn.init.html`.
  - Some common layer initializations include the Xavier (a.k.a. Glorot), and orthogonal initializations.
  - You may want to avoid initializing your network with all zeros (think about the symmetry of the neural units, and how identical initialization may be a bad idea considering what happens during training).
- You **can** use and customize any kind of optimization methods you deem appropriate.
  - You can use any first order stochastic methods (i.e., Stochastic Gradient Descent variants) such as Vanilla SGD, Adam, RMSProp, Adagrad, etc.
  - You are also welcome to use second order optimization methods such as newton and quasi-newton methods. However, it may be expensive and difficult to make them work for this setting.
  - Zeroth order methods (i.e., Black Box methods) are also okay (although you may not find them very effective in this setting).
  - You can specify any learning rates first order stochastic methods. In fact, you can even customize your learning rate schedules.
  - You are free to use any mini-batch sizes for stochastic gradient computation.
  - etc.
- You **can** use any kind of loss function you deem effective.
  - You can add any kind of regularization to your loss.
  - You can pick any kind of classification loss functions such as the cross-entropy and the mean squared loss.
- You **cannot** warm-start your network (i.e., you **cannot** use a pre-trained network).
- You **may** use any kind of image pre-processing and transformations during training. However, for the same transformations to persist at grading time, you may need to apply such transformations within the neural network's `forward` function definition.
  - In other words, we will drop any `DataLoader` or transformations that your network may rely on to have good performance, and we will only load and use your neural network for grading.

3 1. Object Classification Using the CIFAR Data

3.1 1.1 Loading the Data

[6]:
```
message = 'You can implement the pre-processing transformations, data sets,␣
 ↪data loaders, etc. in this cell. \n'
message = message + '**Important Note**: Read the "Grading Reference␣
 ↪Pre-processing" bullet above, and look at the'
message = message + ' test pre-processing transformations in the "Autograding␣
 ↪and Final Tests" section before'
message = message + ' training models for long periods of time.'
print(message)

transform = transforms.Compose(
    [transforms.RandomAffine(degrees=30, translate=(0.01, 0.01), scale=(0.9, 1.
 ↪1),
```

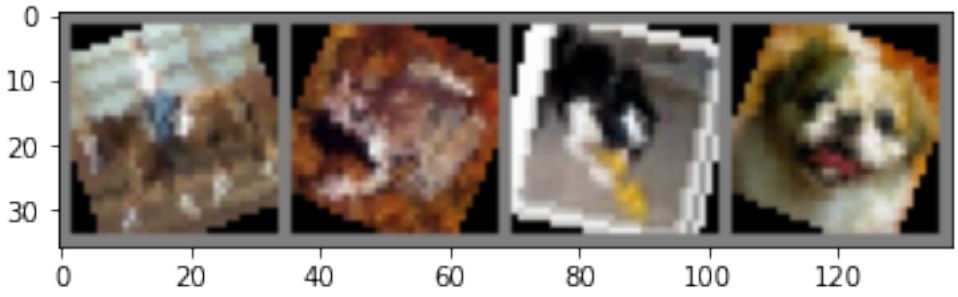
```
                                                shear=None, resample=0,␣
↪fillcolor=0),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='/home/jovyan/work/course-lib/
↪data_cifar', train=True,
                                        download=False, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='/home/jovyan/work/course-lib/
↪data_cifar', train=False,
                                       download=False, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# your code here
# raise NotImplementedError
```

You can implement the pre-processing transformations, data sets, data loaders, etc. in this cell.
Important Note: Read the "Grading Reference Pre-processing" bullet above, and look at the test pre-processing transformations in the "Autograding and Final Tests" section before training models for long periods of time.

[7]:
```
message = 'You can visualize some of the pre-processed images here (This is␣
↪optional and only for your own reference).'
print(message)

# your code here
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()
# print(images.shape, labels.shape)
# show images
imshow(torchvision.utils.make_grid(images))
```

```
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))

# raise NotImplementedError
```

You can visualize some of the pre-processed images here (This is optional and only for your own reference).




```
horse  frog   dog   dog
```

4 1.2 Defining the Model

Important Note: As mentioned above, make sure you name the neural module class as `Net`. In other words, you are supposed to have the following lines somewhere in your network definition:

```
import torch.nn as nn
class Net(nn.Module):
   ...
```

[8]:
```
message = 'You can define the neural architecture and instantiate it in this␣
 ↪cell.'
print(message)

# your code here
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 84)
```

```
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.alpha_dropout ( F.celu( self.conv1(x), 0.9 ), 0.1 ) )
        x = self.pool(F.alpha_dropout ( F.celu( self.conv2(x), 0.9 ), 0.1 ) )
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()

# raise NotImplementedError
```

```
You can define the neural architecture and instantiate it in this cell.
```

5 1.3 Initializing the Neural Model

It may be a better idea to fully control the initialization process of the neural weights rather than leaving it to the default procedure chosen by pytorch.

Here is pytorch's documentation about different initialization methods: https://pytorch.org/docs/stable/nn.init.html

Some common layer initializations include the Xavier (a.k.a. Glorot), and orthogonal initializations.

[9]:
```
message = 'You can initialize the neural weights here, and not leave it to the␣
 ↪library default (this is optional).'
# print(message)
# print(net)
# print(nn.init.xavier_normal_(net.conv1.weight, gain = 1)[0][1])

gain = nn.init.calculate_gain('leaky_relu', 0.2)
# print(gain)
nn.init.xavier_normal_(net.conv1.weight, gain = gain)
nn.init.xavier_normal_(net.conv2.weight, gain = gain)
nn.init.xavier_normal_(net.fc1.weight, gain = gain)
nn.init.xavier_normal_(net.fc2.weight, gain = gain)
nn.init.xavier_normal_(net.fc3.weight, gain = gain)

# your code here
# raise NotImplementedError
```

[9]:
```
Parameter containing:
tensor([[ 5.6357e-02, -5.7237e-01, -1.9227e-01,  3.5276e-01,  1.5889e-02,
         -3.0655e-02, -1.1477e-01, -2.6419e-01, -4.6648e-01, -3.6514e-01,
```

```
  1.8320e-03, -1.4103e-01,  3.0335e-01, -4.4770e-01,  3.7262e-01,
  9.8880e-02,  3.1616e-02,  3.4646e-01, -7.0389e-02, -3.6781e-02,
  5.3349e-02, -6.2918e-03,  2.3753e-02, -2.7932e-01,  5.8881e-02,
 -2.2654e-01,  5.4420e-02,  4.5163e-02, -3.5548e-01, -8.7781e-02,
  1.2602e-02,  2.8328e-02,  1.0181e-01,  5.7333e-02,  2.8151e-01,
 -1.4122e-01,  2.0142e-01,  1.4189e-01,  2.4303e-01, -9.7615e-02,
 -1.5289e-01, -7.6305e-02,  9.0132e-03,  1.2161e-01,  2.2335e-01,
 -9.2094e-02,  4.1822e-02,  2.4520e-02,  3.1333e-01, -9.7074e-02,
  7.9003e-03,  1.9889e-01,  6.4882e-02, -3.3168e-01,  2.2195e-03,
  1.0699e-01, -1.1570e-01, -1.4014e-01,  6.3598e-02, -4.3367e-02,
 -2.9626e-01,  5.9932e-02, -8.5125e-02,  1.7677e-01, -2.9792e-02,
  2.0367e-01,  1.6780e-01, -1.2766e-01,  2.7474e-01,  5.8528e-01,
  2.0881e-02,  2.5769e-02,  1.9387e-02,  1.5440e-02, -8.3341e-02,
 -2.6347e-02, -9.5011e-02, -9.1210e-02,  6.5793e-02, -1.0610e-01,
 -4.8047e-02,  9.6641e-02,  3.3572e-01, -3.1943e-02],
[-1.4287e-01, -1.2551e-01, -3.0368e-01,  7.2406e-02, -3.3848e-02,
  2.8797e-01,  4.2244e-02,  4.8308e-01, -5.8211e-03,  2.1226e-01,
 -1.1462e-01, -2.3376e-01,  1.2084e-01, -7.8696e-02,  1.8946e-01,
 -7.3145e-01,  1.9512e-02,  3.3439e-02, -2.6247e-01, -2.1557e-01,
  7.1777e-02, -1.5159e-01,  1.1369e-01, -5.6243e-02, -2.6536e-01,
 -1.8286e-01, -1.4801e-01, -1.5381e-01, -4.3885e-02, -1.1039e-01,
  8.2280e-02,  1.8122e-01,  3.9329e-01,  6.9683e-02,  2.0933e-01,
  2.5323e-01,  1.0736e-01, -4.5527e-01, -2.2858e-01, -5.9781e-02,
  1.6518e-01, -5.2610e-01,  1.1359e-01, -1.3514e-01,  2.8027e-01,
  1.4920e-01, -1.3077e-01, -2.4052e-01, -1.4249e-01, -6.6867e-02,
  1.5957e-01,  9.3918e-02,  3.0786e-01, -1.3435e-01,  1.1930e-01,
 -2.0876e-01, -1.6279e-01,  1.4887e-01,  7.0640e-02,  9.2085e-02,
  5.0997e-02, -3.7722e-01,  1.4636e-02,  2.6825e-02,  4.1854e-01,
  4.8295e-01, -1.1012e-01, -1.4560e-01, -2.5963e-01, -5.1714e-01,
 -2.8271e-01,  1.6538e-01, -4.5566e-02, -2.1117e-01,  1.1939e-01,
  3.6428e-01, -1.1277e-01, -7.9249e-02, -2.8576e-01,  7.2201e-02,
 -2.3814e-01, -6.6396e-02, -5.0738e-02,  7.9678e-02],
[-1.7736e-01,  5.1598e-02, -8.0045e-02,  3.4264e-01,  3.0480e-02,
  1.9024e-01, -5.5382e-02,  1.0410e-01,  1.3185e-01, -9.2655e-02,
 -1.0193e-02, -3.0634e-01,  6.8325e-01,  1.8568e-01,  1.5515e-02,
 -2.7689e-01, -4.3870e-03,  2.9618e-02, -1.5443e-01,  1.1869e-01,
 -2.2667e-01, -2.7908e-02,  9.9806e-02,  4.3436e-02,  2.3372e-01,
 -2.8898e-01,  3.4412e-01, -1.5063e-01, -1.9508e-01, -1.1664e-01,
  4.5659e-01, -1.0583e-02,  5.9261e-02, -2.4056e-01, -2.4717e-01,
 -1.9299e-01,  3.1494e-01,  1.4013e-01, -1.1024e-01,  5.5101e-01,
  8.2056e-02, -1.0688e-01, -5.7323e-01,  1.3217e-01,  1.6468e-01,
 -1.2147e-01,  9.9240e-02, -9.8410e-02, -3.3439e-01, -2.7947e-01,
 -1.6440e-01, -3.0235e-01,  3.3811e-01, -8.5181e-02, -2.2276e-01,
 -7.8766e-02, -1.5263e-01,  8.3321e-02, -2.5953e-01, -8.2214e-02,
 -3.9445e-02,  3.0574e-01,  1.5996e-01,  3.3068e-03, -3.3405e-02,
 -1.2668e-02,  2.8024e-02,  1.6450e-01,  4.3098e-01,  3.9647e-01,
 -1.3222e-01, -2.9570e-01, -4.2121e-02,  2.2588e-01,  1.1487e-01,
```

```
  6.4150e-02,  6.1511e-01,  4.8616e-01, -3.3537e-01,  1.0363e-01,
  2.8042e-01,  3.6883e-02,  1.0564e-01,  3.3853e-02],
[-2.5114e-01,  2.5296e-01, -3.4024e-01,  1.2084e-02,  3.0593e-01,
  2.1252e-01,  1.8095e-01, -1.1274e-03, -1.7093e-01, -2.7501e-01,
 -1.5514e-01, -5.2956e-02,  8.1694e-03, -2.0902e-01,  1.5002e-01,
  1.1265e-01, -1.5147e-01, -3.2352e-02, -4.5631e-02, -2.3955e-01,
 -6.1551e-02, -1.5825e-01, -1.1144e-01, -2.0094e-01, -1.2815e-01,
 -2.6222e-01,  1.9756e-01,  1.1111e-01, -8.7974e-02, -1.3272e-02,
  5.9845e-02,  7.6398e-02,  4.9043e-01,  1.7757e-01,  2.5848e-01,
  1.0073e-01,  5.3677e-02, -2.4046e-01, -1.9582e-01, -8.9336e-02,
 -3.5395e-02,  1.0422e-01,  6.3197e-02, -1.7998e-01, -2.7556e-01,
 -2.1635e-01,  3.9627e-01,  3.4270e-01, -9.0712e-02,  2.3995e-01,
  1.7661e-02, -7.6225e-01,  1.3341e-01, -1.1023e-01,  1.7279e-01,
 -9.5565e-02,  9.6804e-02, -6.4068e-02,  3.3468e-01,  1.8564e-01,
  1.2972e-01,  2.4640e-01,  3.0372e-01,  7.0441e-02,  1.6023e-01,
  8.4757e-02,  1.7130e-01, -1.4909e-01, -3.4024e-02,  2.1822e-01,
 -2.8112e-01,  8.7139e-02,  2.5591e-01,  2.0940e-02, -4.2286e-02,
 -3.4182e-02, -1.9318e-01,  6.5683e-02,  6.8024e-02,  2.3900e-02,
  2.1914e-01, -1.4344e-01,  6.1908e-02, -2.6176e-01],
[ 2.7695e-01, -1.4677e-01, -3.8288e-01, -2.8686e-01,  2.3575e-01,
  2.6026e-01,  1.1376e-01, -8.8420e-02, -3.8512e-02, -8.9706e-02,
  2.6090e-03,  2.0950e-01,  7.8637e-03, -1.4400e-01,  7.1656e-02,
 -1.5774e-02,  2.1151e-01, -2.2109e-01, -2.2296e-01,  5.6504e-02,
 -6.4266e-02,  5.9733e-02,  3.0200e-02,  8.6510e-02, -1.4417e-01,
  1.2987e-01,  1.4940e-01,  1.9789e-01,  2.7932e-02, -2.7234e-01,
 -3.0857e-03,  6.3350e-02, -3.8683e-02,  3.0689e-01,  2.1268e-02,
 -6.3168e-02,  6.6890e-02,  3.6477e-02,  3.5054e-01,  1.8016e-01,
 -2.4453e-01, -5.3037e-02,  2.1386e-01, -2.5422e-01,  1.6195e-01,
 -1.7849e-01, -1.2001e-01, -3.4013e-01, -3.0474e-01, -5.3331e-02,
 -4.0822e-01, -6.9431e-02, -1.9704e-01, -1.5429e-01, -9.5290e-02,
  3.1784e-02,  1.3397e-01, -2.4906e-02,  1.2692e-01,  1.9201e-01,
  2.1656e-01, -4.9341e-02,  2.1126e-01,  8.4020e-02, -1.9558e-01,
 -1.6109e-01,  1.2835e-01, -1.9660e-01,  4.1441e-02,  6.3078e-02,
  3.4472e-02,  4.3027e-01, -2.2067e-01,  1.7986e-01, -1.1423e-01,
 -1.6928e-01,  4.3820e-02,  7.1923e-03,  6.0644e-01, -3.5829e-03,
  7.2551e-02,  1.1640e-01, -1.8541e-01, -1.5635e-01],
[-2.1829e-01,  1.3198e-02, -1.8516e-02,  2.5223e-01,  4.6984e-01,
  2.0192e-01, -2.4541e-03,  1.3264e-03, -1.2411e-01,  1.1096e-01,
  8.0693e-02,  3.4367e-01, -4.3801e-01,  2.9330e-01,  7.9872e-02,
 -6.3877e-02, -1.1491e-01, -3.3989e-01, -3.4840e-01,  2.0145e-01,
 -1.0776e-01,  2.9436e-02,  2.5727e-02,  8.4071e-02, -3.5124e-01,
 -2.8266e-01, -4.8944e-02, -9.8320e-03, -1.4651e-01,  6.5699e-02,
 -1.3797e-03,  3.4837e-01, -4.5528e-01, -2.5027e-01, -5.3873e-02,
 -1.4847e-01, -2.3001e-01,  6.2103e-03,  2.1135e-01,  8.9528e-02,
  8.4843e-03,  4.5435e-01, -1.4549e-01, -7.0560e-02, -1.0713e-01,
  2.9778e-01,  3.6068e-02,  1.4925e-01,  2.6395e-01,  4.0520e-02,
 -4.6358e-01, -1.2518e-01,  1.6923e-01,  1.3119e-01, -1.8362e-01,
```

```
 -1.4798e-01, -1.3696e-01, -1.5092e-01,  7.5446e-02,  5.2320e-01,
  2.1685e-01, -2.3435e-01, -2.8392e-01, -1.0799e-01,  1.4681e-01,
 -3.2183e-01, -2.4970e-01,  6.7017e-02, -1.6698e-01, -1.4579e-01,
 -5.9909e-02, -2.8341e-01,  1.4270e-01,  2.6721e-01,  1.1617e-01,
 -1.3872e-01, -4.0880e-02, -3.0559e-01, -2.8004e-01, -1.7852e-01,
 -2.3322e-01, -5.7424e-02, -6.9812e-02, -4.3875e-01],
[ 1.4803e-01, -2.8287e-02,  2.3016e-01,  1.2255e-01, -1.7716e-01,
 -2.7432e-01, -3.2045e-01, -5.0095e-02,  1.6244e-01, -1.5493e-01,
 -4.7367e-01, -2.2239e-01,  7.5064e-02,  3.0221e-01, -1.9083e-01,
  2.5725e-01,  1.5535e-01, -2.4761e-01,  3.0607e-01,  3.9399e-01,
 -2.4244e-01,  3.7524e-01,  6.3887e-02, -3.3996e-01, -4.2087e-01,
 -2.8961e-01, -4.3412e-01, -4.6093e-01,  9.7828e-02,  1.6098e-01,
  2.3932e-02,  6.0791e-02, -3.1599e-01, -2.0776e-01,  3.3874e-03,
  1.8710e-01,  4.9153e-01,  3.9362e-02,  6.5794e-02, -4.6653e-02,
 -2.3654e-02, -1.3575e-01,  1.9829e-01,  3.4025e-01,  2.1740e-01,
 -1.4090e-01,  4.4301e-02, -7.4785e-02, -3.8981e-01,  2.7045e-01,
 -1.4842e-01,  1.6878e-01,  1.9129e-02, -2.5052e-02,  1.3350e-01,
  1.7763e-01, -3.5206e-01, -5.1879e-03, -1.1356e-01,  4.5171e-02,
  1.5941e-01, -1.6636e-01,  4.1143e-01,  4.5795e-02,  1.8885e-01,
 -5.1516e-02, -2.4019e-01,  1.2035e-02, -3.8022e-01,  2.4373e-01,
 -7.8521e-03, -1.4874e-01,  3.0901e-01, -5.7663e-02, -1.4538e-01,
  5.0089e-01,  7.9227e-03,  3.2145e-01, -6.3430e-01, -5.4571e-02,
  1.3149e-01,  1.1945e-01, -9.1978e-02, -8.6299e-03],
[ 3.7864e-01,  1.8267e-01, -6.7252e-02, -3.7073e-01,  1.6750e-01,
 -1.9829e-01, -2.7435e-01,  6.6342e-02,  1.1564e-02, -5.7632e-03,
  1.8172e-01, -2.2983e-01, -1.5945e-01, -1.0888e-01, -4.1332e-02,
 -2.0253e-01,  3.8462e-01,  5.9854e-02,  1.4519e-01,  2.2041e-01,
 -1.3803e-01, -1.1459e-01, -1.0269e-01,  1.6570e-01,  2.7439e-01,
  1.1665e-01, -1.8338e-01, -7.9853e-02, -1.1711e-01,  1.7318e-01,
 -4.2921e-02, -2.0291e-01, -1.4502e-01,  2.6721e-01,  2.1020e-01,
 -9.1814e-02, -3.2996e-02,  1.1118e-01, -4.9127e-02, -8.7966e-02,
  5.4983e-01, -2.3186e-02, -1.1571e-01, -9.1178e-03, -1.4955e-01,
 -5.0527e-01,  2.1511e-01, -2.7926e-02,  3.6865e-01,  1.8860e-01,
 -3.6383e-01, -6.1829e-02,  3.1185e-02, -1.6100e-01, -2.1153e-01,
  2.3060e-01, -7.3172e-02,  3.1812e-01, -3.2850e-02, -1.7950e-02,
  4.7916e-01, -2.0995e-01,  4.0421e-02,  1.1724e-01,  5.3159e-03,
  2.6358e-02, -2.6817e-02,  5.1149e-01,  1.7594e-01, -1.7615e-01,
  2.4486e-01,  4.9151e-02, -1.7575e-02,  2.0307e-02,  2.0452e-01,
  1.7204e-01,  1.5992e-01, -3.0848e-04,  1.2951e-02, -1.4242e-02,
  2.5711e-01, -9.2082e-02, -1.6201e-01, -1.0717e-01],
[ 9.1361e-02,  8.8633e-02,  2.0936e-03, -7.7930e-02, -5.2807e-02,
  4.2270e-01, -1.1950e-01, -2.5940e-01, -3.4273e-01, -5.0240e-02,
  1.8330e-01, -8.0949e-02, -1.6395e-01,  4.2793e-01, -1.1413e-01,
 -1.2600e-01, -1.7648e-01, -1.7492e-02, -3.3463e-01,  2.3947e-01,
  2.8103e-01,  1.5552e-01,  2.1530e-01, -1.4636e-01,  3.4401e-01,
  8.5932e-02,  7.7948e-02,  1.5488e-01,  2.3861e-02, -1.3984e-02,
 -1.2314e-01, -2.9769e-02, -1.4965e-01, -3.9650e-02, -1.5975e-01,
```

```
        -1.4156e-01, -3.5829e-02,  3.6817e-01,  4.3477e-02, -2.3845e-02,
        -2.0211e-01,  2.2383e-01, -1.8799e-01, -5.8834e-02,  1.4153e-01,
         1.3193e-01, -2.8347e-01,  1.1249e-01, -2.1537e-02, -7.1126e-02,
        -3.5373e-01,  1.9959e-01,  5.2148e-03, -8.9737e-02, -1.4113e-01,
        -3.0688e-01, -1.1073e-01, -4.3460e-01,  2.6287e-01,  6.0816e-02,
         1.1992e-01, -1.2726e-01, -2.7171e-01,  2.1565e-01,  1.4563e-02,
         5.4969e-03, -3.2290e-02,  1.9975e-01, -7.2639e-02,  3.2805e-01,
         2.2416e-01,  3.2883e-01,  2.2714e-01,  2.3267e-01,  2.7093e-01,
        -2.3700e-01, -1.1139e-01,  6.6238e-03, -1.2569e-01,  1.0841e-01,
         2.6257e-01,  4.0143e-02,  2.7002e-01,  3.1151e-01],
       [-5.7267e-02, -2.9808e-02, -1.3352e-01, -1.3403e-01,  2.2078e-01,
        -1.7722e-01, -3.8478e-01, -1.8254e-01, -2.4623e-01, -7.1869e-02,
        -2.9041e-02, -2.9726e-01, -1.0848e-01,  2.6827e-01,  1.6193e-01,
         7.9765e-02,  1.1349e-01, -3.1918e-01,  1.3737e-01,  4.7371e-02,
         1.3945e-01, -2.1798e-01, -1.9175e-01,  2.6003e-01,  1.1763e-01,
        -2.1227e-02,  5.4342e-02,  2.8385e-01, -1.8673e-01,  7.0220e-03,
         2.2718e-01, -7.1472e-02,  1.0523e-01, -2.5278e-01,  3.3969e-01,
         3.9268e-03,  1.7917e-01,  1.3475e-01, -5.5493e-02,  4.0179e-02,
         2.3442e-01, -4.2687e-02,  3.6130e-01, -2.2219e-01, -1.0952e-01,
        -4.5922e-01, -1.7287e-01, -3.3347e-01, -2.3483e-03, -3.9109e-02,
         2.8194e-01, -8.1000e-02,  2.2845e-01, -1.2642e-01, -3.7563e-01,
         4.8331e-02,  3.5460e-02, -2.0102e-01,  2.3031e-01,  6.6626e-02,
         3.7405e-02, -5.1715e-02,  9.4839e-02,  1.5683e-01, -3.1503e-03,
         1.5726e-01, -2.0818e-01, -2.5991e-03,  5.5908e-02,  4.7303e-02,
        -2.4958e-01, -3.3956e-01, -3.1722e-02, -3.9263e-02, -7.6564e-02,
         1.9682e-01,  1.2334e-01,  4.8991e-02,  2.8561e-02,  1.1722e-01,
        -4.9265e-02,  9.1157e-02,  1.7787e-01, -1.8632e-01]],
      requires_grad=True)
```

6 1.4 Defining The Loss Function and The Optimizer

[10]:
```
message = 'You can define the loss function and the optimizer of interest here.'
print(message)

# your code here
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
# optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
optimizer = optim.Adagrad(net.parameters(), lr=0.01, lr_decay=0,␣
 ↪weight_decay=0, initial_accumulator_value=0, eps=1e-10)
# raise NotImplementedError
```

```
You can define the loss function and the optimizer of interest here.
```

7 1.5 Training the Model

Important Note: In order for the autograder not to time-out due to training during grading, please make sure you wrap your training code within the following conditional statement:

```
if perform_computation:
   # Place any computationally intensive training/optimization code here
```

[11]:
```
if perform_computation:
    message = 'You can define the training loop and forward-backward␣
 ↪propagation here.'
    print(message)

    # your code here
    for epoch in range(15):  # loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:    # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0

    print('Finished Training')

# raise NotImplementedError
```

```
You can define the training loop and forward-backward propagation here.
[1,  2000] loss: 1.827
[1,  4000] loss: 1.642
[1,  6000] loss: 1.569
[1,  8000] loss: 1.528
[1, 10000] loss: 1.469
[1, 12000] loss: 1.490
```

```
[2,  2000] loss: 1.411
[2,  4000] loss: 1.414
[2,  6000] loss: 1.409
[2,  8000] loss: 1.380
[2, 10000] loss: 1.391
[2, 12000] loss: 1.378
[3,  2000] loss: 1.340
[3,  4000] loss: 1.342
[3,  6000] loss: 1.332
[3,  8000] loss: 1.347
[3, 10000] loss: 1.334
[3, 12000] loss: 1.321
[4,  2000] loss: 1.301
[4,  4000] loss: 1.315
[4,  6000] loss: 1.299
[4,  8000] loss: 1.290
[4, 10000] loss: 1.284
[4, 12000] loss: 1.300
[5,  2000] loss: 1.261
[5,  4000] loss: 1.269
[5,  6000] loss: 1.279
[5,  8000] loss: 1.254
[5, 10000] loss: 1.285
[5, 12000] loss: 1.267
[6,  2000] loss: 1.249
[6,  4000] loss: 1.258
[6,  6000] loss: 1.267
[6,  8000] loss: 1.255
[6, 10000] loss: 1.237
[6, 12000] loss: 1.246
[7,  2000] loss: 1.226
[7,  4000] loss: 1.222
[7,  6000] loss: 1.227
[7,  8000] loss: 1.259
[7, 10000] loss: 1.237
[7, 12000] loss: 1.231
[8,  2000] loss: 1.213
[8,  4000] loss: 1.216
[8,  6000] loss: 1.220
[8,  8000] loss: 1.222
[8, 10000] loss: 1.222
[8, 12000] loss: 1.209
[9,  2000] loss: 1.221
[9,  4000] loss: 1.202
[9,  6000] loss: 1.194
[9,  8000] loss: 1.205
[9, 10000] loss: 1.195
[9, 12000] loss: 1.222
```

```
[10,  2000] loss: 1.188
[10,  4000] loss: 1.199
[10,  6000] loss: 1.184
[10,  8000] loss: 1.189
[10, 10000] loss: 1.189
[10, 12000] loss: 1.185
[11,  2000] loss: 1.195
[11,  4000] loss: 1.185
[11,  6000] loss: 1.187
[11,  8000] loss: 1.177
[11, 10000] loss: 1.174
[11, 12000] loss: 1.186
[12,  2000] loss: 1.184
[12,  4000] loss: 1.178
[12,  6000] loss: 1.170
[12,  8000] loss: 1.193
[12, 10000] loss: 1.166
[12, 12000] loss: 1.146
[13,  2000] loss: 1.157
[13,  4000] loss: 1.174
[13,  6000] loss: 1.171
[13,  8000] loss: 1.171
[13, 10000] loss: 1.169
[13, 12000] loss: 1.162
[14,  2000] loss: 1.160
[14,  4000] loss: 1.169
[14,  6000] loss: 1.174
[14,  8000] loss: 1.149
[14, 10000] loss: 1.159
[14, 12000] loss: 1.147
[15,  2000] loss: 1.144
[15,  4000] loss: 1.143
[15,  6000] loss: 1.157
[15,  8000] loss: 1.139
[15, 10000] loss: 1.158
[15, 12000] loss: 1.163
Finished Training
```

8 1.6 Storing the Model

[12]:
```python
message = 'Here you should store the model at "./cifar_net.pth" .'
print(message)

# your code here
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

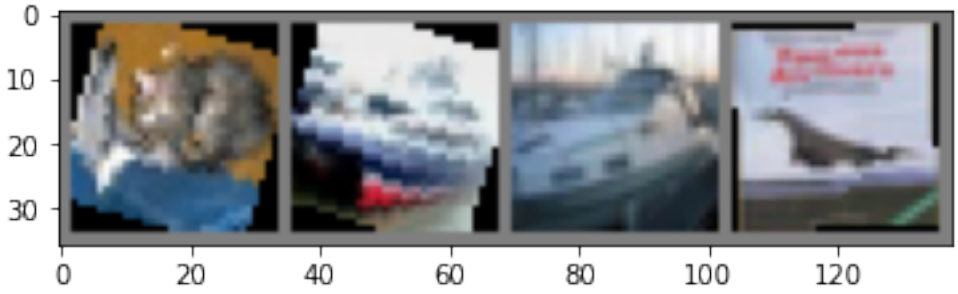
```
# raise NotImplementedError
```

Here you should store the model at "./cifar_net.pth" .

9 1.7 Evaluating the Trained Model

[13]:
```
message = 'Here you can visualize a bunch of examples and print the prediction␣
 ↪of the trained classifier (this is optional).'
print(message)

# your code here

dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))

# raise NotImplementedError
```

Here you can visualize a bunch of examples and print the prediction of the trained classifier (this is optional).



GroundTruth: cat ship ship plane

[14]:
```
message = 'Here you can evaluate the overall accuracy of the trained classifier␣
 ↪(this is optional).'
print(message)

# your code here

correct = 0
total = 0
with torch.no_grad():
```

```
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

# raise NotImplementedError
```

```
Here you can evaluate the overall accuracy of the trained classifier (this is
optional).
Accuracy of the network on the 10000 test images: 57 %
```

[15]:
```
message = 'Here you can evaluate the per-class accuracy of the trained␣
 ↪classifier (this is optional).'
print(message)

# your code here

class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))

# raise NotImplementedError
```

```
Here you can evaluate the per-class accuracy of the trained classifier (this is
optional).
Accuracy of plane : 58 %
Accuracy of   car : 69 %
Accuracy of  bird : 46 %
```

```
Accuracy of   cat : 39 %
Accuracy of  deer : 45 %
Accuracy of   dog : 46 %
Accuracy of  frog : 69 %
Accuracy of horse : 64 %
Accuracy of  ship : 72 %
Accuracy of truck : 59 %
```

9.1 1.8 Autograding and Final Tests

[16]:
```
assert 'Net' in globals().keys(), 'The Net class was not defined earlier. ' + \
                                  'Make sure you read and follow the␣
 ↪instructions provided as Important Notes' + \
                                  '(especially, the "Model Class Naming" part).'

cifar_net_path = './cifar_net_submitted.pth' if in_submission else './cifar_net.
 ↪pth'

assert os.path.exists(cifar_net_path), 'You have not stored the trained model␣
 ↪properly. '+  \
                                       'Make sure you read and follow the␣
 ↪instructions provided as Important Notes.'

assert os.path.getsize(cifar_net_path) < 1000000, 'The size of your trained␣
 ↪model exceeds 1 MB.'



if 'net' in globals():
    del net
net = Net()
net.load_state_dict(torch.load(cifar_net_path))
net = net.eval()

# Disclaimer: Most of the following code was adopted from Pytorch's␣
 ↪Documentation and Examples
# https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

transformation_list = [transforms.RandomAffine(degrees=30, translate=(0.01, 0.
 ↪01), scale=(0.9, 1.1),
                                               shear=None, resample=0,␣
 ↪fillcolor=0),
                       transforms.ToTensor(),
                       transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]

test_pre_tranformation = transforms.Compose(transformation_list)
```

```
cifar_root = '/home/jovyan/work/course-lib/data_cifar'
testset = torchvision.datasets.CIFAR10(root=cifar_root, train=False,
                                       download=False,␣
 ↪transform=test_pre_tranformation)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=1)


class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
print('----------')
print(f'Overall Testing Accuracy: {100. * sum(class_correct) /␣
 ↪sum(class_total)} %%')
```

```
Accuracy of plane : 59 %
Accuracy of   car : 67 %
Accuracy of  bird : 45 %
Accuracy of   cat : 40 %
Accuracy of  deer : 46 %
Accuracy of   dog : 47 %
Accuracy of  frog : 68 %
Accuracy of horse : 65 %
Accuracy of  ship : 72 %
Accuracy of truck : 60 %
----------
Overall Testing Accuracy: 57.4 %%
```

[18]:
```
# "Object Classification Test: Checking the accuracy on the CIFAR Images"
```

10 2. Digit Recognition Using the MNIST Data

10.1 2.1 Loading the Data

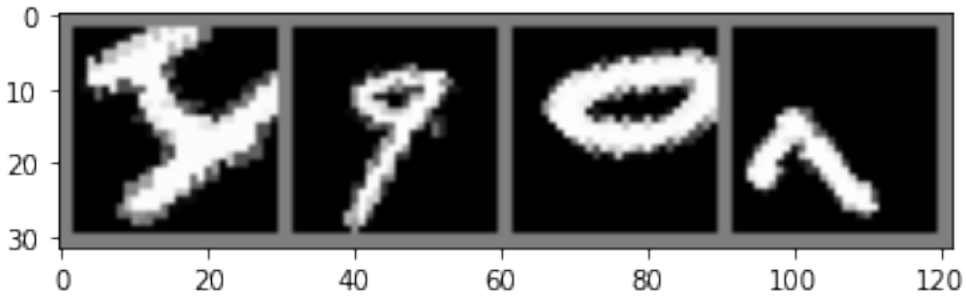
[23]:
```
message = 'You can implement the pre-processing transformations, data sets,␣
 ↪data loaders, etc. in this cell. \n'
message = message + '**Important Note**: Read the "Grading Reference␣
 ↪Pre-processing" bullet, and look at the'
message = message + ' test pre-processing transformations in the "Autograding␣
 ↪and Final Tests" section before'
message = message + ' training models for long periods of time.'
print(message)

# your code here

transform = transforms.Compose( [transforms.RandomAffine(degrees=60,␣
 ↪translate=(0.2, 0.2), scale=(0.5, 2.),
                                                  shear=None, resample=0,␣
 ↪fillcolor=0),
                        transforms.ToTensor(),
                        transforms.Normalize((0.5,), (0.5,))] )

trainset = torchvision.datasets.MNIST(root='/home/jovyan/work/course-lib/
 ↪data_mnist', train=True,
                                       download=False, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.MNIST(root='/home/jovyan/work/course-lib/
 ↪data_mnist', train=False,
                                      download=False, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

classes = ('0', '1', '2', '3',
           '4', '5', '6', '7', '8', '9')

# raise NotImplementedError
```

You can implement the pre-processing transformations, data sets, data loaders, etc. in this cell.
Important Note: Read the "Grading Reference Pre-processing" bullet, and look at the test pre-processing transformations in the "Autograding and Final Tests" section before training models for long periods of time.

[24]:
```
message = 'You can visualize some of the pre-processed images here (This is␣
 ↪optional and only for your own reference).'
```

```
print(message)

# your code here
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join('%5s' % classes[labels[j]] for j in range(4)))

# raise NotImplementedError
```

You can visualize some of the pre-processed images here (This is optional and only for your own reference).




```
    4     9     0     7
```

11 2.2 Defining the Model

Important Note: As mentioned above, make sure you name the neural module class as `Net`. In other words, you are supposed to have the following lines somewhere in your network definition:

```
import torch.nn as nn
class Net(nn.Module):
    ...
```

[25]:
```
message = 'You can define the neural architecture and instantiate it in this␣
 ↪cell.'
print(message)

# your code here
# defining the model architecture

# class Net(nn.Module):
#     def __init__(self):
#         super(Net, self).__init__()
#         self.conv1 = nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=1)
#         self.pool = nn.MaxPool2d(2, 2)
#         self.batch = nn.BatchNorm2d(4)
#         self.relu = nn.ReLU(inplace=True)
#         self.conv2 = nn.Conv2d(4, 4, kernel_size=3, stride=1, padding=1)
#         self.fc1 = nn.Linear(4 * 7 * 7, 10)

#     def forward(self, x):
#         x = self.pool( self.relu( self.batch( self.conv2( self.pool( self.
 ↪relu( self.batch( self.conv1(x) ) ) ) ) ) ) )
#         x = x.view(x.size(0), -1)
#         x = self.fc1(x)
#         return x

##############################

# class Net(nn.Module):
#   def __init__(self):
#       super(Net, self).__init__()

#       self.cnn_layers = nn.Sequential(
#           # Defining a 2D convolution layer
#           nn.Conv2d(1, 4, kernel_size=3, stride=1, padding=1),
#           nn.BatchNorm2d(4),
#           nn.ReLU(inplace=True),
#           nn.MaxPool2d(kernel_size=2, stride=2),
#           # Defining another 2D convolution layer
#           nn.Conv2d(4, 4, kernel_size=3, stride=1, padding=1),
#           nn.BatchNorm2d(4),
#           nn.ReLU(inplace=True),
#           nn.MaxPool2d(kernel_size=2, stride=2),
#       )

#       self.linear_layers = nn.Sequential(
#           nn.Linear(4 * 7 * 7, 10)
#       )
```

```
#   # Defining the forward pass
#   def forward(self, x):
#       x = self.cnn_layers(x)
#       x = x.view(x.size(0), -1)
#       x = self.linear_layers(x)
#       return x

 ###################################

# class Net(nn.Module):
#     def __init__(self):
#         super(Net, self).__init__()
#         self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
#         self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
#         self.conv2_drop = nn.Dropout2d()
# #         self.conv2_drop = nn.AlphaDropout(p = 0.5)
#         self.fc1 = nn.Linear(320, 50)
#         self.fc2 = nn.Linear(50, 10)

#     def forward(self, x):
#         x = F.relu(F.max_pool2d(self.conv1(x), 2))
#         x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
#         x = x.view(-1, 320)
#         x = F.relu(self.fc1(x))
#         x = F.dropout(x, training=self.training)
#         x = self.fc2(x)
#         return F.log_softmax(x)

 ###################################

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 4, 3, 1)
        self.conv2 = nn.Conv2d(4, 8, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(1152, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
```

```
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output

net = Net()

# raise NotImplementedError
```

```
You can define the neural architecture and instantiate it in this cell.
```

12 2.3 Initializing the Neural Model

It may be a better idea to fully control the initialization process of the neural weights rather than leaving it to the default procedure chosen by pytorch.

Here is pytorch's documentation about different initialization methods: https://pytorch.org/docs/stable/nn.init.html

Some common layer initializations include the Xavier (a.k.a. Glorot), and orthogonal initializations.

[26]:
```
message = 'You can initialize the neural weights here, and not leave it to the␣
 ↪library default (this is optional).'
print(message)

# your code here

gain = nn.init.calculate_gain('leaky_relu', 0.15)
# print(gain)

# nn.init.orthogonal_(net.conv1.weight, gain = gain)
# nn.init.orthogonal_(net.conv2.weight, gain = gain)
# nn.init.orthogonal_(net.fc1.weight, gain = gain)
# nn.init.orthogonal_(net.fc2.weight, gain = gain)

# nn.init.xavier_normal_(net.conv1.weight, gain = gain)
# nn.init.xavier_normal_(net.conv2.weight, gain = gain)
# nn.init.xavier_normal_(net.fc1.weight, gain = gain)
# nn.init.xavier_normal_(net.fc2.weight, gain = gain)

# raise NotImplementedError
```

```
You can initialize the neural weights here, and not leave it to the library
default (this is optional).
```

13 2.4 Defining The Loss Function and The Optimizer

[27]:
```
message = 'You can define the loss function and the optimizer of interest here.'
print(message)

# your code here

import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
# optimizer = optim.Adagrad(net.parameters(), lr=0.01, lr_decay=0,␣
 ↪weight_decay=0, initial_accumulator_value=0, eps=1e-10)
# opmtimizer = optim.Adam(net.parameters(), lr=0.01)
# optimizer = optim.RMSprop(net.parameters(), lr=0.01, alpha=0.99, eps=1e-08,␣
 ↪weight_decay=0, momentum=0, centered=False)

# raise NotImplementedError
```

```
You can define the loss function and the optimizer of interest here.
```

14 2.5 Training the Model

Important Note: In order for the autograder not to time-out due to training during grading, please make sure you wrap your training code within the following conditional statement:

```
if perform_computation:
   # Place any computationally intensive training/optimization code here
```

[28]:
```
if perform_computation:
    message = 'You can define the training loop and forward-backward␣
 ↪propagation here.'
    print(message)

    # your code here
    for epoch in range(15):  # loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
```

```
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
            if i % 2000 == 1999:    # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0

    print('Finished Training')

#     raise NotImplementedError
```

```
You can define the training loop and forward-backward propagation here.
[1,  2000] loss: 2.277
[1,  4000] loss: 2.155
[1,  6000] loss: 2.094
[1,  8000] loss: 1.999
[1, 10000] loss: 1.886
[1, 12000] loss: 1.764
[1, 14000] loss: 1.686
[2,  2000] loss: 1.568
[2,  4000] loss: 1.516
[2,  6000] loss: 1.460
[2,  8000] loss: 1.425
[2, 10000] loss: 1.391
[2, 12000] loss: 1.373
[2, 14000] loss: 1.322
[3,  2000] loss: 1.314
[3,  4000] loss: 1.292
[3,  6000] loss: 1.268
[3,  8000] loss: 1.245
[3, 10000] loss: 1.254
[3, 12000] loss: 1.234
[3, 14000] loss: 1.225
[4,  2000] loss: 1.201
[4,  4000] loss: 1.194
[4,  6000] loss: 1.172
[4,  8000] loss: 1.163
[4, 10000] loss: 1.175
[4, 12000] loss: 1.154
[4, 14000] loss: 1.152
[5,  2000] loss: 1.123
[5,  4000] loss: 1.133
```

```
[5,  6000] loss: 1.142
[5,  8000] loss: 1.107
[5, 10000] loss: 1.143
[5, 12000] loss: 1.089
[5, 14000] loss: 1.108
[6,  2000] loss: 1.094
[6,  4000] loss: 1.079
[6,  6000] loss: 1.094
[6,  8000] loss: 1.103
[6, 10000] loss: 1.075
[6, 12000] loss: 1.039
[6, 14000] loss: 1.059
[7,  2000] loss: 1.060
[7,  4000] loss: 1.063
[7,  6000] loss: 1.069
[7,  8000] loss: 1.082
[7, 10000] loss: 1.066
[7, 12000] loss: 1.047
[7, 14000] loss: 1.075
[8,  2000] loss: 1.056
[8,  4000] loss: 1.027
[8,  6000] loss: 1.034
[8,  8000] loss: 1.040
[8, 10000] loss: 1.065
[8, 12000] loss: 1.058
[8, 14000] loss: 1.026
[9,  2000] loss: 1.042
[9,  4000] loss: 1.046
[9,  6000] loss: 1.040
[9,  8000] loss: 1.051
[9, 10000] loss: 1.014
[9, 12000] loss: 1.033
[9, 14000] loss: 1.013
[10,  2000] loss: 1.033
[10,  4000] loss: 1.016
[10,  6000] loss: 1.009
[10,  8000] loss: 1.016
[10, 10000] loss: 1.013
[10, 12000] loss: 0.987
[10, 14000] loss: 1.015
[11,  2000] loss: 0.994
[11,  4000] loss: 0.990
[11,  6000] loss: 1.012
[11,  8000] loss: 1.010
[11, 10000] loss: 1.003
[11, 12000] loss: 0.995
[11, 14000] loss: 0.992
[12,  2000] loss: 0.974
```

```
[12,  4000] loss: 1.007
[12,  6000] loss: 1.003
[12,  8000] loss: 1.000
[12, 10000] loss: 0.986
[12, 12000] loss: 0.980
[12, 14000] loss: 0.980
[13,  2000] loss: 0.978
[13,  4000] loss: 0.975
[13,  6000] loss: 0.976
[13,  8000] loss: 0.985
[13, 10000] loss: 0.988
[13, 12000] loss: 0.962
[13, 14000] loss: 0.958
[14,  2000] loss: 0.976
[14,  4000] loss: 0.961
[14,  6000] loss: 1.011
[14,  8000] loss: 0.973
[14, 10000] loss: 0.956
[14, 12000] loss: 0.983
[14, 14000] loss: 0.967
[15,  2000] loss: 0.942
[15,  4000] loss: 0.970
[15,  6000] loss: 0.958
[15,  8000] loss: 0.954
[15, 10000] loss: 0.964
[15, 12000] loss: 0.984
[15, 14000] loss: 0.973
Finished Training
```

15 2.6 Storing the Model

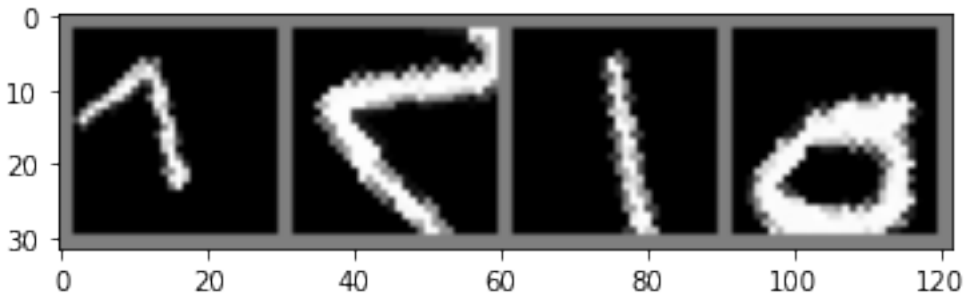
[29]:
```python
message = 'Here you should store the model at "./mnist_net.pth" .'
print(message)

# your code here

PATH = './mnist_net.pth'
torch.save(net.state_dict(), PATH)

# raise NotImplementedError
```

```
Here you should store the model at "./mnist_net.pth" .
```

16 2.7 Evaluating the Trained Model

[30]:
```
message = 'Here you can visualize a bunch of examples and print the prediction␣
 ↪of the trained classifier (this is optional).'
print(message)

# your code here

dataiter = iter(testloader)
images, labels = dataiter.next()

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))

# raise NotImplementedError
```

```
Here you can visualize a bunch of examples and print the prediction of the
trained classifier (this is optional).
```




```
GroundTruth:       7     2     1     0
```

[31]:
```
message = 'Here you can evaluate the overall accuracy of the trained classifier␣
 ↪(this is optional).'
print(message)

# your code here

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
```

```
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

# raise NotImplementedError
```

Here you can evaluate the overall accuracy of the trained classifier (this is optional).
Accuracy of the network on the 10000 test images: 68 %

[32]:
```
message = 'Here you can evaluate the per-class accuracy of the trained␣
 ↪classifier (this is optional).'
print(message)

# your code here

class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))

# raise NotImplementedError
```

Here you can evaluate the per-class accuracy of the trained classifier (this is optional).
Accuracy of   0 : 72 %
Accuracy of   1 : 93 %
Accuracy of   2 : 58 %
Accuracy of   3 : 67 %
Accuracy of   4 : 73 %
Accuracy of   5 : 67 %
Accuracy of   6 : 69 %

```
Accuracy of     7 : 64 %
Accuracy of     8 : 50 %
Accuracy of     9 : 67 %
```

16.1 2.8 Autograding and Final Tests

[33]:
```
assert 'Net' in globals().keys(), 'The Net class was not defined earlier. ' + \
                                  'Make sure you read and follow the␣
 ↪instructions provided as Important Notes' + \
                                  '(especially, the "Model Class Naming" part).'

mnist_net_path = './mnist_net_submitted.pth' if in_submission else './mnist_net.
 ↪pth'

assert os.path.exists(mnist_net_path), 'You have not stored the trained model␣
 ↪properly. '+  \
                                       'Make sure you read and follow the␣
 ↪instructions provided as Important Notes.'

assert os.path.getsize(mnist_net_path) < 1000000, 'The size of your trained␣
 ↪model exceeds 1 MB.'

if 'net' in globals():
    del net
net = Net()
net.load_state_dict(torch.load(mnist_net_path))
net = net.eval()

# Disclaimer: Most of the following code was adopted from Pytorch's␣
 ↪Documentation and Examples
# https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

transformation_list = [transforms.RandomAffine(degrees=60, translate=(0.2, 0.
 ↪2), scale=(0.5, 2.),
                                               shear=None, resample=0,␣
 ↪fillcolor=0),
                       transforms.ToTensor(),
                       transforms.Normalize((0.5,), (0.5,))]

test_pre_tranformation = transforms.Compose(transformation_list)

mnist_root = '/home/jovyan/work/course-lib/data_mnist'
testset = torchvision.datasets.MNIST(root=mnist_root, train=False,
                                     download=False,␣
 ↪transform=test_pre_tranformation)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=1)
```

```
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
print('----------')
print(f'Overall Testing Accuracy: {100. * sum(class_correct) /␣
 ↪sum(class_total)} %%')
```

```
Accuracy of     0 : 89 %
Accuracy of     1 : 98 %
Accuracy of     2 : 72 %
Accuracy of     3 : 81 %
Accuracy of     4 : 86 %
Accuracy of     5 : 82 %
Accuracy of     6 : 83 %
Accuracy of     7 : 76 %
Accuracy of     8 : 66 %
Accuracy of     9 : 81 %
----------
Overall Testing Accuracy: 82.19 %%
```

[34]:
```
# "Digit Recognition Test: Checking the accuracy on the MNIST Images"
```

[ ]:
```

```