Lab04 - Neural Network in PyTorch

Objectives:

In this lab, you will learn how to implement the following steps in PyTorch:

1. Load a torchvision dataset
2. Create a transformation pipeline
3. Create a Standard Neural Network
4. Define a loss function and optmizer
5. Train the network
6. Evaluate the network

Content:


```
1 from google.colab import drive
2 drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

```
1 !pip install torchinfo
```

```
Collecting torchinfo
  Downloading torchinfo-1.8.0-py3-none-any.whl (23 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.8.0
```

```
cd "gdrive/My Drive/UCCD3074_Labs/UCCD3074_Lab4"
```

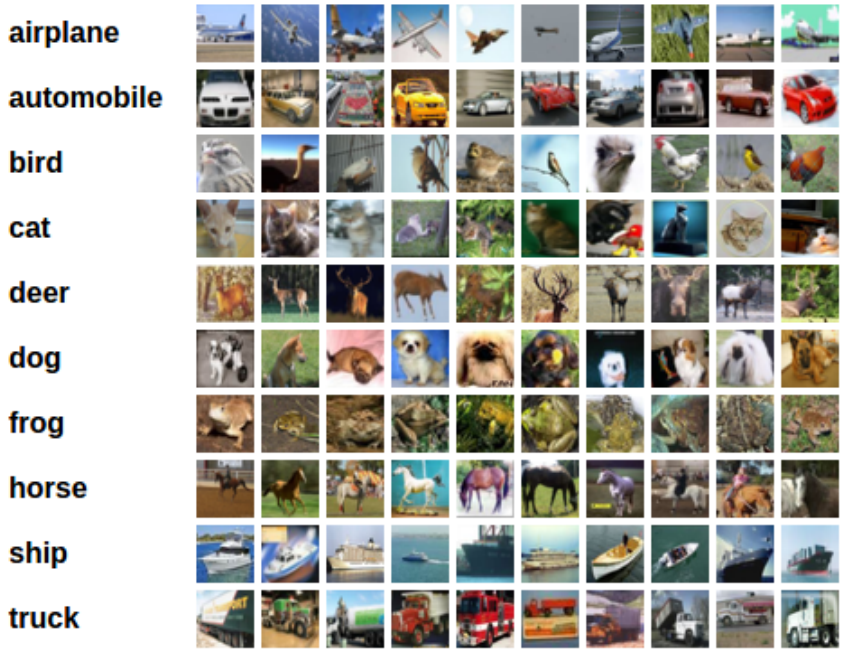
```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from torch.utils.data import DataLoader

from torchinfo import summary
from cifar10 import CIFAR10

import torchvision
import torchvision.transforms as transforms

%load_ext autoreload
%autoreload 2
```

---

1. The CIFAR10 dataset

For this tutorial, we will use the CIFAR10 dataset. The CIFAR-10 dataset consists of 60000 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. However for speed consideration, we shall subsample 10000 training images and 2000 training images.

The following figure shows 10 randomly selected samples for each class. This is a **multi-class classification** where there is only one object per image. The images in CIFAR-10 are of size (3, 32, 32), i.e. 3-channel color images of 32x32 pixels in resolution.



`torchvision.datasets`

First, we load the CIFAR10 dataset, one of the many computer vision datasets listed in `torchvision.datasets`. The dataset is an object of type `torch.utils.data.Dataset`. To reduce the training time, we have create a wrapper function `CIFAR10` in the module *cifar10.py* that loads only a subset of the CIFAR10 library.

```
1 trainset = CIFAR10(root=".", train=True, num_samples=10000, download=True)
2 testset  = CIFAR10(root=".", train=False, num_samples=2000, download=True)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./cifar-10-python.tar.gz
100%|██████████| 170498071/170498071 [00:03<00:00, 47561374.15it/s]
Extracting ./cifar-10-python.tar.gz to .
Files already downloaded and verified
```

The classes can be found in the `classes` attribute of the dataset object

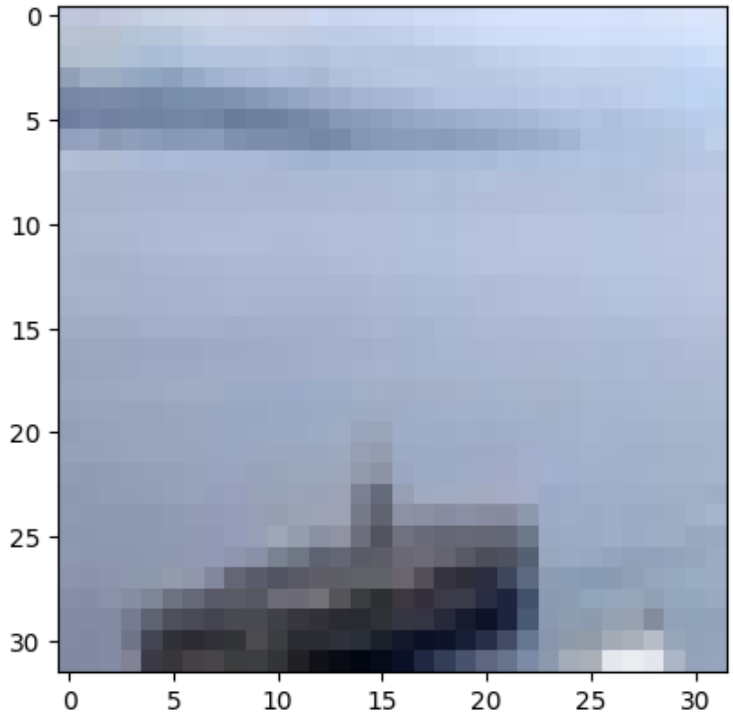
```
1 classes = trainset.classes
2 print('classes =', classes)
3 print('Number of training set:', len(trainset))
4 print('Number of test samples:', len(testset))
```

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
Number of training set: 10000
Number of test samples: 2000
```

We can access sample *i* directly by indexing the dataset `trainset[i]` which returns the tuple `(image, label)`.

```
1 img, label = trainset[0]
2 plt.imshow(img)
3 print('label =', label, ' class =', classes[label])
```

```
label = 8  class = ship
```



Each sample in the dataset is a numpy array of shape `(height, width, channel)`. The pixel values ranges from 0 to 255. In addition, the data type of the image is `uint8`.

```
print('\nType of img:', type(img))
print('Type of items in img:', img.dtype)

print('Shape of img:', img.shape)
print('Range of img:', img.min(), 'to', img.max())
```

```
Type of img: <class 'numpy.ndarray'>
Type of items in img: uint8
Shape of img: (32, 32, 3)
Range of img: 1 to 254
```

---

2. TRANSFORMING THE INPUT IMAGE

Composing the Transformation Pipeline

The torchvision package contains the [`torchvision.transforms`] library which contains the set of common image transformations. You can create a pipeline (series) of transformations by chaining them together using the command `torchvision.transform.Compose`.

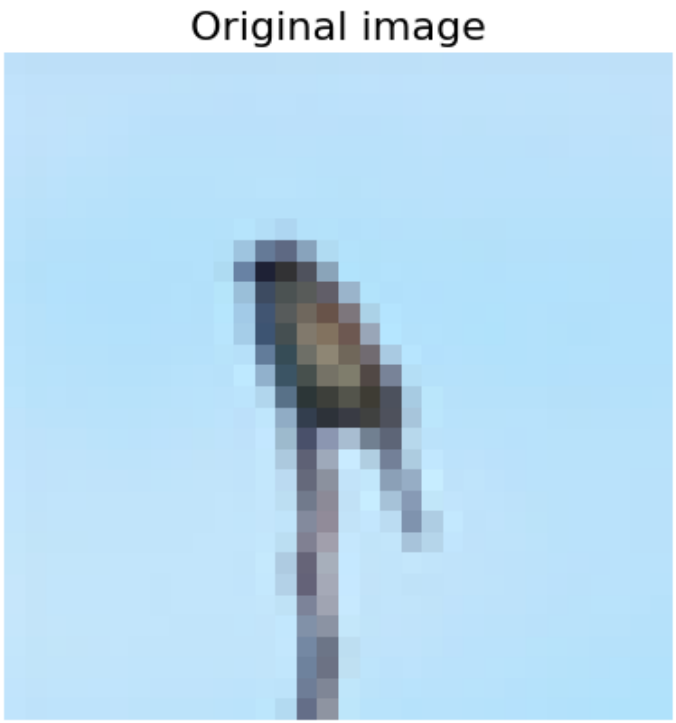
For example, the following code creates the transformation pipeline:

1. Convert the input image (numpy array of shape (H, W, C)) into a PIL image (`transform.ToPILImage()`).
   Many of various image transformation functions in `torchvision.transform` can only work on PIL, but not numpy arrays.
2. Randomly flip the image horizontally (`transform.RandomHorizontalFlip()`)
3. Randomly sample a sub-image of shape (24, 24) from the image (`transform.RandomCrop()`)

```
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop((24, 24))
])
```

Given an image, we can subject it to `transform` that will then perform the series of specified transformations.

```
1 img, label = trainset[1]
2 plt.imshow(img)
3 plt.title('Original image', fontsize=16)
4 _ = plt.axis("off")
5
```




```
1 transformed_img = transform(img)
2 plt.imshow(transformed_img)
3 plt.title('Transformed image')
4 _ = plt.axis("off")
```

Transforming the CIFAR dataset

For the CIFAR10 dataset, we shall perform the following transformation:

1. Convert the type from numpy *array* to a PyTorch *tensor* and change the dimension from `(H, W, C)` to `(C, H, W)` (`torchvision.transforms.ToTensor`). This is a mandatory step.
2. Normalize the data with mean and standard deviation for each channel: `output[c] = (input[c] - mean[c])/std[c]` (`torchvision.transforms.Normalize`)

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (1.0, 1.0, 1.0))
])
```

Before the transformation, the image has a shape of `(h, w, c)` and the pixel ranges between 0 and 255.

```
img, label = trainset[1]
print('Original input shape:', img.shape)
```

```
print(f'Range of values: {img.min():d} to {img.max():d}')
```

```
Original input shape: (32, 32, 3)
Range of values: 30 to 255
```

After the transformation. the image has a shape of `(c, h, w)` and the pixel values are normalized with a mean and standard deviation of roughly 0 and 1, respectively.

```
x = transform(img)
print('Transformed input shape:', x.shape)
print(f'Range of values: {x.min():.2f} to {x.max():.2f}')
```

```
Transformed input shape: torch.Size([3, 32, 32])
Range of values: -0.38 to 0.50
```

Reloading the dataset with transformation

Now that we have defined the transformation pipeline, we shall reload our dataset with `transform=transform` to apply the transformation upon loading.

```
trainset = CIFAR10(train=True,  transform=transform, num_samples=10000, download=True)
testset  = CIFAR10(train=False,  transform=transform, num_samples=2000, download=True)

print('\nNumber of training samples:', len(trainset))
print('Number of test samples:', len(testset))
```

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

Number of training samples: 10000
Number of test samples: 2000
```

Note that the the loaded image now has a shape of (C x H x W), the range is between [-1, 1], the type is *torch.FloatTensor*.

```
img, label = trainset[0]
print('Shape of img:', img.shape)
```

```
print(f'Range of img: {img.min().item():.2f} to {img.max().item():.2f}')
print('Type of img:', type(img))
print('Type of items in img:', img.dtype)
```

```
Shape of img: torch.Size([3, 32, 32])
Range of img: -0.46 to 0.45
Type of img: <class 'torch.Tensor'>
Type of items in img: torch.float32
```

---

3. Batch Loading with BatchLoader

Minibatch Gradient Descent (MGD) loads the data in batches. To do that, we need to do the following:

1. Batch the data for parallel processing
2. Shuffle the data

`DataLoader` [torch.utils.data.DataLoader] is an iterator that provides the above two features. The following code uses a dataloader to load a batch size of 4 (`batch_size=4`). The datasets are shuffled at the beginning of each epoch (`shuffle=True`). A total of 2 workers are used to load the data in parallel (`num_workers=2`).

```
trainloader = DataLoader(trainset, batch_size=4, shuffle=True, num_workers=2)
testloader  = DataLoader(testset, batch_size=4, shuffle=True, num_workers=2)
```

The following loads one batch data randomly from the training set. Since `trainloader` is an iterator, you need to use `iter` to retrieve a batch sample.

```
dataiter = iter(trainloader)
x_batch, y_batch = next(dataiter)
print('Shape of x_batch:', x_batch.shape)
print('Shape of y_batch:', y_batch.shape)
```

```
Shape of x_batch: torch.Size([4, 3, 32, 32])
Shape of y_batch: torch.Size([4])
```

---

4. Building Regular Neural Network

In this section, we shall build the following 2-layered neural network:

Layer	Name	Operation	Input Shape	OutputShape
-		view	(?, 3, 32, 32)	(?, 3072)
1	`fc1`	Linear, 50 neurons	(?, 3072)	(?, 50)
	`relu1`	Relu	(?, 50)	(?, 50)
2	`fc2`	Linear, 10 neurons, no activation	(?, 50)	(?, 10)

- `fc1` is the hidden layer. It contains 50 neurons. It uses relu for its activation
- `fc2` is the output layer. It contains 10 neurons which is equivalent to the number of classes in the task. We will not use any activation for the second layer for reasons that will be clear later.

Coding the network

The neural network architectures in PyTorch can be defined as a sub-class of `nn.Module`. We shall do

1. define all the layers inside the constructor `__init__(self)`. The modules for the layers are defined in torch.nn which include the convolutional layers, pooling layers, linear (fully connected) layers, etc.
2. perform forward propagation steps in the function `forward(self, x)`.

The following shows how to define a CNN called `Net`.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(3*32*32, 50)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        # flatten each image
        x = x.view(x.size(0), -1)

        # forward propagation for layer 1
        x = self.fc1(x)
```

```
14         x = self.relu1(x)
15
16         # forward propagation for layer 2
17         x = self.fc2(x)
18
19         return x
```

- line 1, 2, 8: The class of the network inherits from `nn.Module` and overloads the the function `__init__` and `forward`.
- line 2-6: Initialize the network
    - This function **creates the layer objects** in our network upon initialization. **No computational graph is constructed** yet at this point of time.
    - line 4: Creates the 1st linear layer, `self.fc1`.
    - line 5: Creates the activation function for layer 1, `self.relu1`.
    - line 6: Creates the 2nd linear layer, `self.fc2`.
    - Note that `self.fc1`, `self.relu1` and `self.fc2` are all `nn.Modules` objects.
- line 8-17:
    - Perform forward propagation and **creates the computational graph**
    - line 10: flattens all batch images. Here, we maintain the first dimension (`x.size(0)`) and flattens the 2nd dimension onwards. As a result, the tensor's shape changes from (B, 3, 32, 32) to (B, 3072)
    - line 13-14: performs forward propagation through layer 1
    - line 17: performs forward propagation through layer 2

Displaying the network

When you print the network, it will only list out the layers following the sequence declared in `__init__`. The does **not** reflect the actual structure of the computational graph built when you invoke the `__forward__` function.

```
1 net = Net()
2 print(net)
```

```
Net(
  (relu1): ReLU()
  (fc1): Linear(in_features=3072, out_features=50, bias=True)
```

```
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Try to swap `fc1` and `relu` declaration in the **init** function of `Net` above. Then, print the network. Observe that *printing* the the network does not reflect the true layout of the network.

```
1 net = Net()
2 print(net)
```

```
Net(
  (relu1): ReLU()
  (fc1): Linear(in_features=3072, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

To observe the correct layout of the network, you can use **`torchinfo.summary`** which also outputs the shape of the activation map given input of a specified size.

To use `summary`, you need to provide the network object and the input shape to the model

```
1 net = Net()
2 summary(net, input_size=(4, 3, 32, 32))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Net                                      [4, 10]                   --
├─Linear: 1-1                            [4, 50]                   153,650
├─ReLU: 1-2                              [4, 50]                   --
├─Linear: 1-3                            [4, 10]                   510
==========================================================================================
Total params: 154,160
Trainable params: 154,160
Non-trainable params: 0
Total mult-adds (M): 0.62
==========================================================================================
Input size (MB): 0.05
Forward/backward pass size (MB): 0.00
Params size (MB): 0.62
```

```
Estimated Total Size (MB): 0.67
==========================================================================================
```

Testing the network on the fly

In the process of constructing the network, we can apply some random input to the network even when it is still untrained. This allows us to test your network coding as we define them.

Create a random tensor with 4 samples and check if the dimension of the output is correct. What should be the shape of the output tensor?

```
net = Net()

with torch.no_grad():
    inp = torch.rand(4, 3, 32, 32)
    out = net(inp)

print('Shape of out:', out.shape)
print('Out:\n', out)
```

```
Shape of out: torch.Size([4, 10])
Out:
 tensor([[-0.1110, -0.2790, -0.0325,  0.0510,  0.0390, -0.0013, -0.1899,  0.0988,
          0.2316,  0.0156],
        [-0.0551, -0.1703, -0.0645,  0.2323, -0.0193, -0.0657, -0.1534,  0.1640,
          0.1220, -0.0440],
        [-0.1198, -0.2969, -0.0267, -0.0734,  0.0577,  0.0869, -0.2153,  0.1813,
          0.0600, -0.0578],
        [-0.0379, -0.1851, -0.1154,  0.0925,  0.0672, -0.0322, -0.1672,  0.0978,
          0.1464,  0.0086]])
```

Module vs Functional

`nn.module` classes have attributes, e.g., `fc1` has internal attributes like `self.weight` and `self.bias`. However, some layers , e.g., *sigmoid*, *relu*, *dropout*, etc., are attributeless. Rather than explicitly create a layer for these attributeless layers, we can use a **functional** approach to define these layer. You can use the function either in **`torch.nn.functional`** (e.g., `nn.functional.avg_pool2d`) or **`torch`** (e.g., `torch.sigmoid`, `torch.sin`, `torch.softmax`) to perform forward propagation.

To use the functional modules, you just have to call the function directly just like any function without having to declare them first (that's why they are called functionals. For example:

```
import torch.nn.functional as F
...
class Net(nn.Module):
    def __init__():
        ...

    def forward():
        ...
        x = torch.relu(x)
```

Note that all layers implemented using functional are invisible to `print` and `torchinfo.summary`. So, next time when you print the layers of this network, remember that it may not be telling you the whole story.

1. Modify the class `Net` above to implement the `relu1` using the functional approach.
2. Use `print` to display the network. Observe that it does not print the functional `relu` layer although it is in the network.
3. Use `torchinfo.summary` to display the network. Observe that it does not print the functional `relu` layer although it is in th network.

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(3*32*32, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        # flatten each image
        x = x.view(x.size(0), -1)

        # forward propagation for layer 1
        x = self.fc1(x)
        x = torch.relu(x)

        # forward propagation for layer 2
```

```
        x = self.fc2(x)

        return x
```

```
net = Net()
print(net)
```

```
Net(
  (fc1): Linear(in_features=3072, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

```
summary(net, input_size=(4, 3, 32, 32))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Net                                      [4, 10]                   --
├─Linear: 1-1                            [4, 50]                   153,650
├─Linear: 1-2                            [4, 10]                   510
==========================================================================================
Total params: 154,160
Trainable params: 154,160
Non-trainable params: 0
Total mult-adds (M): 0.62
==========================================================================================
Input size (MB): 0.05
Forward/backward pass size (MB): 0.00
Params size (MB): 0.62
Estimated Total Size (MB): 0.67
==========================================================================================
```

---

4. Training the model

Now, we are ready to train our model.

Create the network

Now, let's create the network and transfer them to the GPU if available. **Remember to enable the GPU for this notebook** by setting `Edit > Notebook settings > Hardware accelerator` to `GPU`.

```
net = Net()
if torch.cuda.is_available():
    net = net.cuda()
```

Define the optimizer

To update the parameters, we can use the `torch.optim` package which contains various formulas for optimizing the parameter. For this practical, we use `nn.optim.SGD` optimizer ($w := w - \alpha \frac{dcost}{dw}$) to update `net.parameters`. The update (`optimizer.step()`) will commence only after forward-backward propagation.

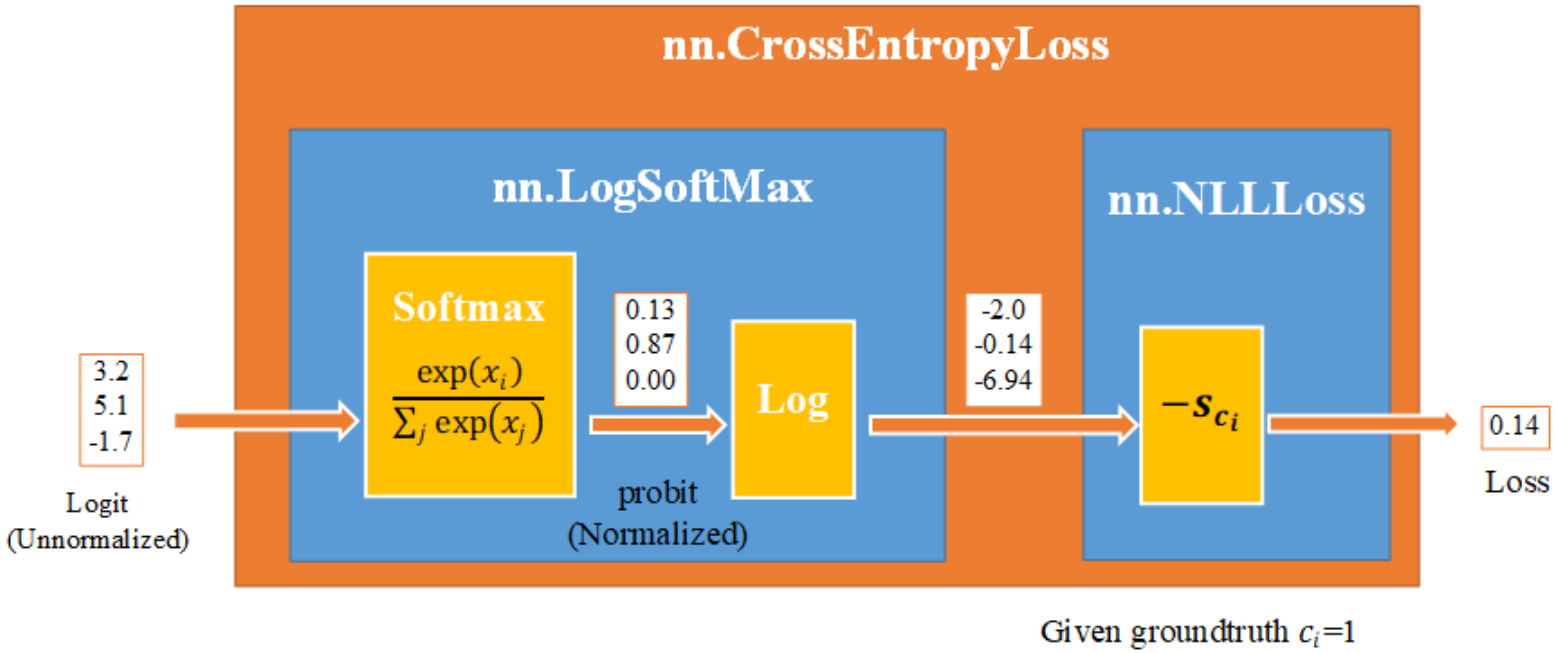
```
optimizer = optim.SGD(net.parameters(), lr=0.001)
```

Define the cost function

Since we are performing **multi-class classification**, we shall use the cross entropy loss. There are at least 3 ways to implement the loss function:

1. log softmax activation + negative log likelihood
    1. Use `log_softmax` activation for the last (classification) layer.
    2. Use `NLLLoss` (negative log likelihood) to compute the cost. Here, the input to the function is the *log likelihood* values. Hence, the input to `NLLLoss` must have applied `log` prior to calling the function.
2. softmax activation + negative likelihood:
    1. Use `softmax` function for the last (classification) layer
    2. Perform `log` on the output of the layer
    3. Use `NLLLoss` (negative log likelihood) to compute the cost.
3. No activation + cross entropy loss
    1. No activation for the last (classification) layer
    2. Use `CrossEntropyLoss` which will implements both `log_softmax` + `NLLLoss` in a single function or module.

Since our last layer has no activation, we shall be implementing the 3rd method. We shall use the `nn.CrossEntropyLoss` as criterion to compute the loss function. The block operations of `nn.CrossEntropyLoss` is sumarized in the diagram below:




```
1 cross_entropy = nn.CrossEntropyLoss()
```

Set to training mode

The following code tells the model that you are in training mode. Some layers like dropout, batchnorm etc. behave differently during training and testing.

```
1 net.train()
```

```
Net(
  (fc1): Linear(in_features=3072, out_features=50, bias=True)
```

```
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Train the model

Now we are ready to train our model for 6 epochs.

```
num_epochs = 6
loop_per_val = 500

for e in range(num_epochs):

    running_loss = 0
    running_count = 0

    for i, (x_batch, y_batch) in enumerate(trainloader):

        # transfer input to GPU
        if torch.cuda.is_available():
            x_batch = x_batch.cuda()
            y_batch = y_batch.cuda()

        # set grad to zero
        net.zero_grad()

        # forward propagation
        yhat_batch = net(x_batch)

        # compute cost
        cost = cross_entropy(yhat_batch, y_batch)

        # backward propagation
        cost.backward()

        # update model
        optimizer.step()

        # compute running loss and validation
        running_loss += cost.item()
        running_count +=1
```

```
34
35         # display the averaged loss value
36         if i % loop_per_val == loop_per_val - 1:
37             train_loss = running_loss / loop_per_val
38             running_loss = 0.
39             running_count = 0
40             print(f'[Epoch {e+1:2d}/{num_epochs:d} Iter {i+1:5d}/{len(trainloader)}]: train_loss = {train_loss:.4f}')
```

```
[Epoch  1/6 Iter   500/2500]: train_loss = 2.2809
[Epoch  1/6 Iter  1000/2500]: train_loss = 2.2338
[Epoch  1/6 Iter  1500/2500]: train_loss = 2.2025
[Epoch  1/6 Iter  2000/2500]: train_loss = 2.1718
[Epoch  1/6 Iter  2500/2500]: train_loss = 2.1296
[Epoch  2/6 Iter   500/2500]: train_loss = 2.0966
[Epoch  2/6 Iter  1000/2500]: train_loss = 2.0666
[Epoch  2/6 Iter  1500/2500]: train_loss = 2.0139
[Epoch  2/6 Iter  2000/2500]: train_loss = 1.9946
[Epoch  2/6 Iter  2500/2500]: train_loss = 2.0040
[Epoch  3/6 Iter   500/2500]: train_loss = 1.9671
[Epoch  3/6 Iter  1000/2500]: train_loss = 1.9573
[Epoch  3/6 Iter  1500/2500]: train_loss = 1.9324
[Epoch  3/6 Iter  2000/2500]: train_loss = 1.9078
[Epoch  3/6 Iter  2500/2500]: train_loss = 1.8786
[Epoch  4/6 Iter   500/2500]: train_loss = 1.8665
[Epoch  4/6 Iter  1000/2500]: train_loss = 1.8805
[Epoch  4/6 Iter  1500/2500]: train_loss = 1.8792
[Epoch  4/6 Iter  2000/2500]: train_loss = 1.8299
[Epoch  4/6 Iter  2500/2500]: train_loss = 1.8326
[Epoch  5/6 Iter   500/2500]: train_loss = 1.8349
[Epoch  5/6 Iter  1000/2500]: train_loss = 1.8215
[Epoch  5/6 Iter  1500/2500]: train_loss = 1.7926
[Epoch  5/6 Iter  2000/2500]: train_loss = 1.7969
[Epoch  5/6 Iter  2500/2500]: train_loss = 1.7862
[Epoch  6/6 Iter   500/2500]: train_loss = 1.8040
[Epoch  6/6 Iter  1000/2500]: train_loss = 1.7800
[Epoch  6/6 Iter  1500/2500]: train_loss = 1.7570
[Epoch  6/6 Iter  2000/2500]: train_loss = 1.7379
[Epoch  6/6 Iter  2500/2500]: train_loss = 1.7653
```

- Line 2, 6-7, 31-40: Computes the averaged loss over 500 iterations
- Line 9: load the batch sample, namely `inputs` with a shape of (4, 3, 32, 32) and `labels` of size (4,). where batch_size = 4 and #channel = 3.

- Line 20: forward propagation to compute the scores for current model
- Line 23: compute the loss value
- Line 26: backpropagation. The gradients for `net.parameters` are computed.
- Line 29: use the optimizer to update `net.parameters` based on the gradient values computed from backpropagation

Retraining the model

Note that if you simply re-run the training cell directly above, it will continue the previous training the model. You can observe it by seeing that the starting cost value is very low and continues from the previous training session. To train the model from scratch again, you can create a new network by running the code `net = Net()` to start clearn.

---

5. Predicting Using the Trained Model

We can use our trained model to predict on the testset. Expected accuracy is around 38%. The performance is better than random guess. If we randomly guess, we can expect the performance is around 10% since there are 10 classes and the classes are well distributed.

Set to evaluation mode

The following code tells the model that you are in evaluation or testing mode. Some layers like dropout, batchnorm etc. behave differently during training and testing.

```
1 net.eval()
```

```
Net(
  (fc1): Linear(in_features=3072, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Evaluate the model

Now we are ready to evaluate our model.

```
 1 # running_correct to accumulate number of correct predictions
 2 running_corrects = 0
 3
 4 for x_batch, y_batch in testloader:
 5
 6     # transfer to GPU
 7     if torch.cuda.is_available():
 8         x_batch = x_batch.cuda()
 9         y_batch = y_batch.cuda()
10
11     # do not want to compute the gradient
12     with torch.no_grad():
13
14         # forward propagation to get class scores
15         yhat_batch = net(x_batch)
16
17         # get the predicted class
18         ypred_batch = yhat_batch.argmax(axis=1)
19
20         # compute the running correct
21         running_corrects += (ypred_batch == y_batch).double().sum()
22
23 accuracy = running_corrects / len(testset)
24 print(f'Accuracy = {accuracy*100:.2f}%')
```

```
Accuracy = 37.70%
```

- Line 4: Loads the `input` (shape = (4, 3, 32, 32)) and `target` (shape=(4,))
- Line 12: Since we are no longer training the model, there is no need to the compute gradient anymore. The wrapper `with torch.no_grad()` temporarily sets all `requires_grad` flag to false. This will save more memory and speed up the process.
- Line 15: Perform foward propagation to get the class scores. `scores` is a FloatTensor of size (4, 10). Batch size is 4 and there are 10 classes.
- Line 18: Gets the predicted class for each score. `preds` is a LongTensor of size (4,) indicating the class index for each batch sample.
- Line 2 and 21 is used to accumulates the number of correctly predicted samples throughout the batch iteration.
- Line 23: Computes the testing accuracy

Analysis

The standard NN should achieve a test accuracy of around 37%. So, is it learning anything useful? The answer is yes, because 37% is still better than a random guess where you can expect a performance of around 10% because there are 10 classes in the network. Admittedly, the performance is far from satisfactory and this not the best that deep learning can do. In the next lab, we shall improve the performance of CIFAR10 using CNN networks.

Conclusion

In this lab, you have implemented neural network using PyTorch. In the process, you have learnt how to

1. Use the **data loader** to load batch data
2. Implement a **transformation pipeline** to preprocess images using `torchvision.transforms`.
3. Use a PyTorch's **optimizer** (`nn.optim.SGD`) to train the network
4. Define a **custom neural network** class using `nn.Module`.
5. How to implement the **cost function**
6. Understand the difference between `nn.Module` and `nn.functional`.