Experiment 7 - Introduction to Deep Learning with PyTorch

The goal of this experiment is to introduce you to deep learning frameworks. There are many frameworks for deep learning such as TensorFlow, PyTorch, Keras, ... etc. In this lab you will learn the basics of PyTorch. The reason why we choose PyTorch is that it is pythonic, easy to learn, and the most pupular framework in academic communities. This experimet is based on PyTorch tutorials.

PyTorch installation

PyTorch is a Python-based scientific computing package serving two broad purposes: a replacement for NumPy to use the power of GPUs and other accelerators, and an automatic differentiation library that is useful to implement neural networks. It is primarily developed by Facebook AI Research lab (FAIR).

Please follow PyTorch's instruction of installation for your system. If you want GPU acceleration, please install the matched version of CUDA in advance. Please visit CUDA Toolkit Archive.

Note that you can use Google Colab without local installation of PyTorch.

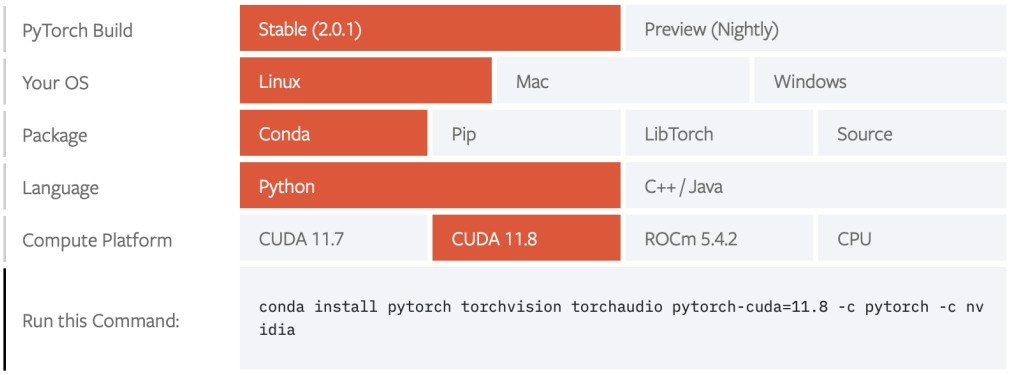

Figure 1.1: PyTorch installation.

1.1 Tensors

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters. Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other specialized hardware to accelerate computing.
Let's start by importing pytorch and numpy

```
import torch
import numpy as np
```

Tensor Initialization

Tensors can be initialized in various ways. Take a look at the following examples:

1. Directly from data
   Tensors can be created directly from data. The data type is automatically inferred.

   ```
   data = [[1, 2], [3, 4]]
   x_data = torch.tensor(data)
   ```

2. From a NumPy array
   Tensors can be created from NumPy arrays

   ```
   np_array = np.array(data)
   x_np = torch.from_numpy(np_array)
   ```

3. From another tensor:
   The new tensor retains the properties (shape, datatype) of the argument tensor, unless explicitly overridden.

   ```
   x_ones = torch.ones_like(x_data) # retains the properties of x_data
   print(f"Ones Tensor: \n {x_ones} \n")

   x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the
       datatype of x_data
   print(f"Random Tensor: \n {x_rand} \n")
   ```

4. With random or constant values:
   shape is a tuple of tensor dimensions. In the functions below, it determines the dimensionality of the output tensor.

   ```
   shape = (2, 3,)
   rand_tensor = torch.rand(shape)
   ones_tensor = torch.ones(shape)
   ```

```
zeros_tensor = torch.zeros(shape)

print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
```

Tensor Attributes

Tensor attributes describe their shape, datatype, and the device on which they are stored.

```
tensor = torch.rand(3, 4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Tensor Operations

Over 100 tensor operations, including transposing, indexing, slicing, mathematical operations, linear algebra, random sampling, and more are described here.
Each of them can be run on the GPU (at typically higher speeds than on a CPU). If you're using Colab, allocate a GPU by going to Edit > Notebook Settings.

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
  tensor = tensor.to('cuda')
  print(f"Device tensor is stored on: {tensor.device}")
```

Try out some of the operations from the list. If you're familiar with the NumPy API, you'll find the Tensor API a breeze to use.

1. Standard numpy-like indexing and slicing:

   ```
   tensor = torch.ones(4, 4)
   tensor[:,1] = 0
   print(tensor)
   ```

2. Joining tensors
   You can use `torch.cat` to concatenate a sequence of tensors along a given dimension. See also `torch.stack`, another tensor joining op that is subtly different from `torch.cat`.

   ```
   t1 = torch.cat([tensor, tensor, tensor], dim=1)
   print(t1)
   ```

3. Multiplying tensors

```
# This computes the element-wise product
print(f"tensor.mul(tensor) \n {tensor.mul(tensor)} \n")
# Alternative syntax:
print(f"tensor * tensor \n {tensor * tensor}")
```

This computes the matrix multiplication between two tensors.

```
print(f"tensor.matmul(tensor.T) \n {tensor.matmul(tensor.T)} \n")
# Alternative syntax:
print(f"tensor @ tensor.T \n {tensor @ tensor.T}")
```

4. In-place operations
Operations that have a _ suffix are in-place. For example: `x.copy_(y), x.t_()`, will change x.

```
print(tensor, "\n")
tensor.add_(5)
print(tensor)
```

Bridge with NumPy

Tensors on the CPU and NumPy arrays can share their underlying memory locations, and changing one will change the other.

Tensor to NumPy array: A change in the tensor reflects in the NumPy array.

```
t = torch.ones(5)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")

t.add_(1)
print(f"t: {t}")
print(f"n: {n}")
```

NumPy array to Tensor: Changes in the NumPy array reflects in the tensor.

```
n = np.ones(5)
t = torch.from_numpy(n)

np.add(n, 1, out=n)
print(f"t: {t}")
print(f"n: {n}")
```

1.2 A Gentle Introduction to `torch.autograd`

`torch.autograd` is PyTorch's automatic differentiation engine that powers neural network training. In this section, you will get a conceptual understanding of how autograd helps a neural network train.

Background

Neural networks (NNs) are a collection of nested functions that are executed on some input data. These functions are defined by parameters (consisting of weights and biases), which in PyTorch are stored in tensors.

Training a NN happens in two steps:

Forward Propagation: In forward prop, the NN makes its best guess about the correct output. It runs the input data through each of its functions to make this guess.

Backward Propagation: In backprop, the NN adjusts its parameters proportionate to the error in its guess. It does this by traversing backwards from the output, collecting the derivatives of the error with respect to the parameters of the functions (gradients), and optimizing the parameters using gradient descent.

Differentiation in Autograd

Let's take a look at how `autograd` collects gradients. We create two tensors a and b with `requires_grad=True`. This signals to `autograd` that every operation on them should be tracked.

```
import torch

a = torch.tensor([2., 3.], requires_grad=True)
b = torch.tensor([6., 4.], requires_grad=True)
```

We create another tensor Q from a and b.

$$Q = 3a^3 - b^2$$

```
Q = 3*a**3 - b**2
```

Let's assume a and b to be parameters of an NN, and Q to be the error. In NN training, we want gradients of the error w.r.t. parameters, i.e.

$$\frac{\partial Q}{\partial a} = 9a^2$$

$$\frac{\partial Q}{\partial b} = -2b$$

When we call `.backward()` on Q, autograd calculates these gradients and stores them in the respective tensors' `.grad` attribute.
We need to explicitly pass a `gradient` argument in `Q.backward()` *because it is a vector*. `gradient` is a tensor of the same shape as Q, and it represents the gradient of Q w.r.t. itself, i.e.

$$\frac{dQ}{dQ} = 1$$

Equivalently, we can also aggregate Q into a scalar and call backward implicitly, like `Q.sum().backward()`.

```
external_grad = torch.tensor([1., 1.])
Q.backward(gradient=external_grad)
```

Gradients are now deposited in `a.grad` and `b.grad`

```
# check if collected gradients are correct
print(9*a**2 == a.grad)
print(-2*b == b.grad)
```

Task 1: *Use autograd to compute the gradients of* Y *w.r.t.* x_1 *and* x_2 *at the point* $(x_1, x_2) = (1, 1)$. *Where*

$$Y = (3x_1 - 2x_2 - 2)^2.$$

Verify your results by computing the gradients analytically.

1.3 Building Models with PyTorch

Neural networks can be constructed using the `torch.nn package`. Now that you had a glimpse of `autograd`, `nn` depends on `autograd` to define models and differentiate them. An `nn.Module` contains layers, and a method `forward(input)` that returns the `output`.
A typical training procedure for a neural network is as follows:

- Define the neural network that has some learnable parameters (or weights)
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss (how far is the output from being correct)
- Propagate gradients back into the network's parameters
- Update the weights of the network, typically using a simple update rule: `weight = weight - learning_rate * gradient`

For example, look at this network that classifies digit images. It is a simple feed-forward network. It takes the input, feeds it through two hidden layers one after the other, and then finally gives the output, which is 10 neurons each corresponds to a different class (digit 0, 1, ..., 9).

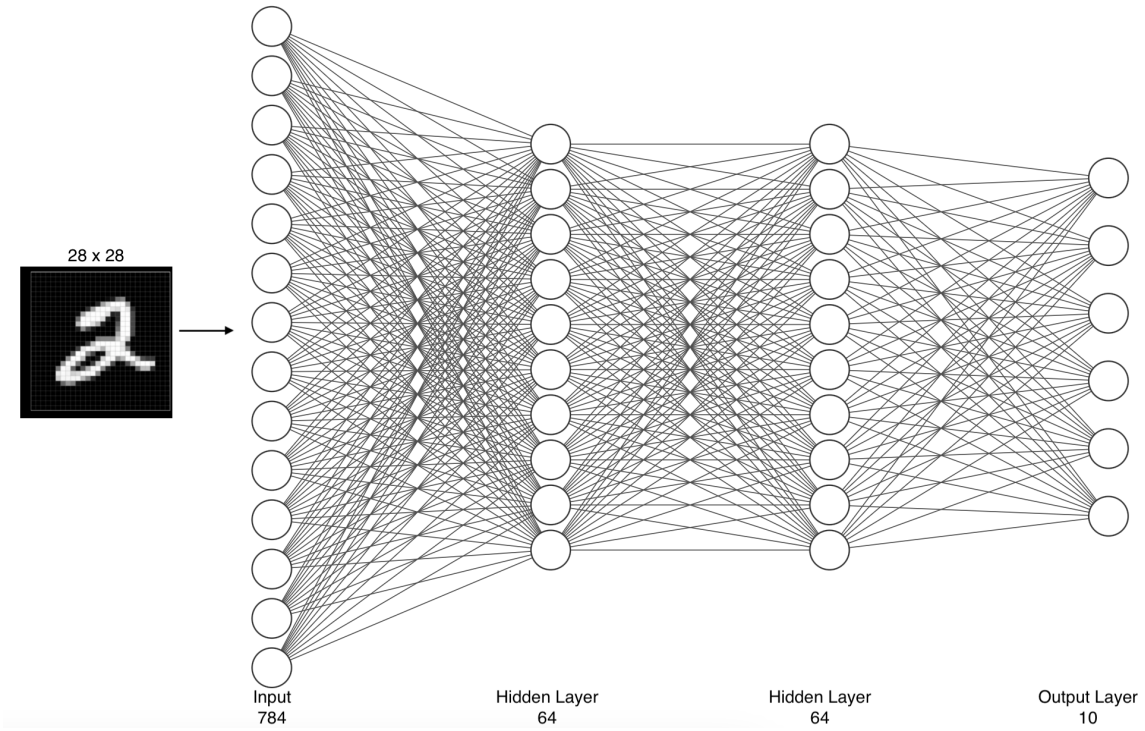

Figure 1.2: MLP for digit classification.

Now, let's build and train the MLP shown in Figure 1.2.

Define the network

Let's define this network:

```
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # an affine operation: y = Wx + b
        # 784 is the input dimension, and 68 is the output dimenstion of
            the first hidden layer
```

```
        self.fc1 = nn.Linear(784, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        # apply the first layer with relu activation
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


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

You just have to define the `forward` function, and the `backward` function (where gradients are computed) is automatically defined for you using `autograd`. You can use any of the Tensor operations in the `forward` function.

The learnable parameters of a model are returned by `net.parameters()`

```
params = list(net.parameters())
print(len(params))

for p in params:
  print(p.size())
```

***Task 2**: Identify what are the parameters that are printed in the previous code.*

Let's try a random input. Note: expected input size of this network is 784.

```
input = torch.randn(1, 784)
out = net(input)
print(out)
```

Note that `torch.nn` only supports mini-batches. The entire `torch.nn` package only supports inputs that are a mini-batch of samples, and not a single sample. That's the reason why we add an additional dimension for the input tensor.

***Task 3**: Try the previous network with a random mini-batch of size 4 and print its output.*

Define a Loss function and optimizer

As we have a classification task, let's use a Classification Cross-Entropy (CE) loss, where

$$CE(\hat{y}, y) = -\sum_{c=1}^{C} y_c \; log \; \hat{y}_c$$

where y is the one-hot encoding of the target class, $\hat{y}$ is the predicted probabilities of the network, and C is the number of classes.
For the optimizer, we will use a simple stochastic gradient descent (SGD) optimizer, which has a simple update rule: `weight = weight - learning_rate * gradient`

```
import torch.optim as optim

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

Note that the input to `nn.CrossEntropyLoss()` is expected to contain the unnormalized logits for each class. That's why we do not use softmax activation for the last layer of the network.

What about data?

To train the network, you need to iterate over your dataset and feed it to the network as mini-batches in the training loop. PyTorch provides two classes to make this process easier: `Dataset` and `Dataloader`. The `Dataset` and `DataLoader` classes encapsulate the process of pulling your data from storage and exposing it to your training loop in batches.
The `Dataset` is responsible for accessing and processing single instances of data. The `DataLoader` pulls instances of data from the `Dataset` (either automatically or with a sampler that you define), collects them in batches, and returns them for consumption by your training loop. The `DataLoader` works with all kinds of datasets, regardless of the type of data they contain.
PyTorch domain libraries provide a number of pre-loaded datasets (such as MNIST) that subclass `torch.utils.data.Dataset` and implement functions specific to the particular data. They can be used to prototype and benchmark your model. In this experiment, we will use the MNIST dataset, which contains images of handwritten digits. It has a training set of 60,000 examples, and a test set of 10,000 examples. Each example comprises a 28×28 grayscale image and an associated label from one of 10 classes (digit 0, 1, ..., 9).

Loading a Dataset

Here is an example of how to load the MNIST dataset from `TorchVision`.
We load the MNIST Dataset with the following parameters:

- `root` is the path where the train/test data is stored.

- `train` specifies training or test dataset.
- `download=True` downloads the data from the internet if it's not available at `root`.
- `transform` and `target_transform` specify the feature and label transformations.

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

Iterating and Visualizing the Dataset

We can index `Datasets` manually like a list: `training_data[index]`. We use `matplotlib` to visualize some samples in our training data.

```
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title("digit:" + str(label))
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

Preparing your data for training with DataLoaders

The `Dataset` retrieves our dataset's features and labels one sample at a time. While training a model, we typically want to pass samples in "minibatches", reshuffle the data at every epoch to reduce model overfitting, and use Python's `multiprocessing` to speed up data retrieval.
`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=4, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=4, shuffle=True)
```

Iterate through the DataLoader

We have loaded that dataset into the `DataLoader` and can iterate through the dataset as needed. Each iteration below returns a batch of `train_features` and `train_labels` (containing `batch_size=4` features and labels respectively). Because we specified `shuffle=True`, after we iterate over all batches the data is shuffled.

```
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

Post-Lab: Read the Creating a Custom Dataset for your files section in PyTorch tutorials and implement a dataloader for the dataset that will be provided by your instructor. Iterate over some of the training examples and visualize them.

Train the network

This is when things start to get interesting. We simply have to loop over our data iterator, and feed the inputs to the network and optimize.

```
for epoch in range(2): # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(train_dataloader, 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(torch.flatten(inputs,1))
        iteration_loss = loss(outputs, labels)
        iteration_loss.backward()
        optimizer.step()

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

print('Finished Training')
```

***Task 4**: What is the meaning of* `epoch`, `forward pass`, `backward pass`. *What is the effect of* `torch.flatten(inputs, 1)`, *and* `optimizer.step()`?

To save our trained model, we can use the following code:

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

Test the network on the test data

We have trained the network for 2 passes over the training dataset. But we need to check if the network has learnt anything at all.
We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions. The outputs are energies for the 10 classes. The higher the energy for a class, the more the network thinks that the image is of the particular class. So, we get the prediction by finding the index of the highest energy. Let's load back in our saved model (note: saving and re-loading the model wasn't necessary here, we only did it to illustrate how to do so):

```
net = Net()
net.load_state_dict(torch.load(PATH))
```

Let's look at how the network performs on the whole testing dataset.

```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for
```

```
    our outputs
with torch.no_grad():
    for data in test_dataloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(torch.flatten(images,1))
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct
    // total} %')
```

***Task 5**: Train the network in the previous example, but instead of using 2 hidden layers, try 3 hidden layers.*

***Task 6**: Train the network in the previous example using Adam optimizer.*

Training on GPU

The training in the previous example was done on CPU. But how do we train our model on GPU?
Just like how you transfer a Tensor onto the GPU, you transfer the neural net onto the GPU. Let's first define our device as the first visible cuda device if we have CUDA available:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Assuming that we are on a CUDA machine, this should print a CUDA device:

print(device)
```

The rest of this section assumes that `device` is a CUDA device.
Then these methods will recursively go over all modules and convert their parameters and buffers to CUDA tensors:

```
net.to(device)
```

Remember that you will have to send the inputs and targets at every step to the GPU too:

```
inputs, labels = data[0].to(device), data[1].to(device)
```

***Task 7**: Train the network in the previous example on GPU. Do you notice significant speedup? if not, try to increase the size of your network.*

1.4 EXTRA (Optional) - Visualizing models, data, and training with tensorboard

PyTorch integrates with TensorBoard, a tool designed for visualizing the results of neural network training runs. This tutorial illustrates some of its functionality.