Homework 0: Warmup of Google Colab and PyTorch

Shukai Gong

1 Google Colab

Google Colab enables users to write and execute Python in the browser, with zero configuration required, **free access to GPUs**, and easy sharing. To utilize the free GPU provided by google, click on "Runtime" -> "Change Runtime Type". There are three options under "Hardward Accelerator", select "GPU". Doing this will restart the session, so make sure you change to the desired runtime before executing any code.

1.1 Codes and Shell Commands

- torch.cuda.is_available(): check if the GPU is available. It outputs True if the GPU is available.
- !nvidia-smi: check the GPU status. smi stands for System Management Interface. It outputs the GPU information, including the GPU model, memory usage, and the processes that are using the GPU.

```
Fri Jun 21 03:50:02 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |                    0 |
| N/A   45C    P8               9W /  70W |      3MiB / 15360MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

Figure 1: Output of !nvidia-smi

- !gdown --id <file_id>: download files from Google Drive. The <file_id> can be found in the shareable link of the file. For example, the file id of the link https://drive.google.com/open?id=123456789abcdefg is 123456789abcdefg.
- !ls: list the files in the current directory. !ls -l lists the files in a detailed format.
- from google.colab import drive: mount Google Drive to Google Colab. This allows users to access files in Google Drive. Then by drive.mount('/content/drive'), the user can access the working files in the Google Drive by /content/drive/My Drive/.
- !pwd: print the current working directory.
- !cd <path>: change the current working directory to <path>.
- !mkdir <dir_name>: create a new directory named <dir_name>.

2 PyTorch

Pytorch is a machine learning framework in Python. It has two main features:

- **Tensor computation**: N-dimensional array, similar to numpy but with GPU acceleration.
- **Automatic differentiation**: PyTorch can automatically compute the gradients of the tensors for training DNNs.

Figure 2 shows the pipeline of deep learning. Now we will introduce some basic operations in PyTorch to implement the pipeline.

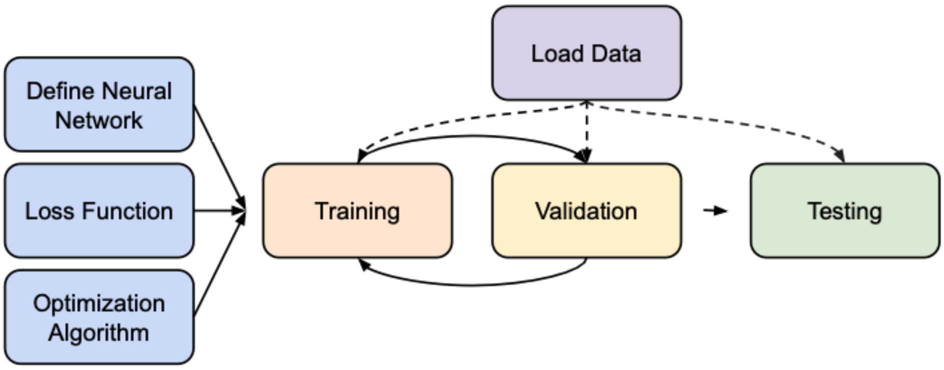

Figure 2: Pipeline of Deep Learning

2.1 Load Data

2.1.1 Dataset and DataLoader

`Dataset` and `Dataloader` are two important attributes of `torch.utils.data`.

- `Dataset`: An abstract class representing a dataset. It stores data samples and expected values.
- `DataLoader`: It is an iterator that groups data in batches, enables multiprocessing.

A typical example of using `Dataset` and `DataLoader` is shown in the following code. First we define a dataset class `MyDataset` that inherits from `torch.utils.data.Dataset`.

```python
from torch.utils.data import Dataset, DataLoader
class MyDataset(Dataset):
    '''
    x: np.ndarray, feature matrix
    y: np.ndarray, labels if available, otherwise None
    '''
    # Initialize the dataset, read data and preprocess
    def __init__(self, x, y=None):
        if y is None:
            self.y = y
        else:
            self.y = torch.FloatTensor(y)
        self.x = torch.FloatTensor(x)
    # Returns one sample at a time
```

```python
    def __getitem__(self, idx):
        if self.y is None:
            return self.x[idx]
        return self.x[idx], self.y[idx]
    # Returns the size of the dataset
    def __len__(self):
        return len(self.x)
```

Then we can create a `DataLoader` object to load the data in batches. Figure 3 shows the mechanism of `DataLoader`. It relies on the `Dataset` object to load the data. We can set the batch size and shuffle the data by setting the parameters of `DataLoader`. Note that we should set `shuffle=True` when training the model, but set `shuffle=False` when evaluating the model.

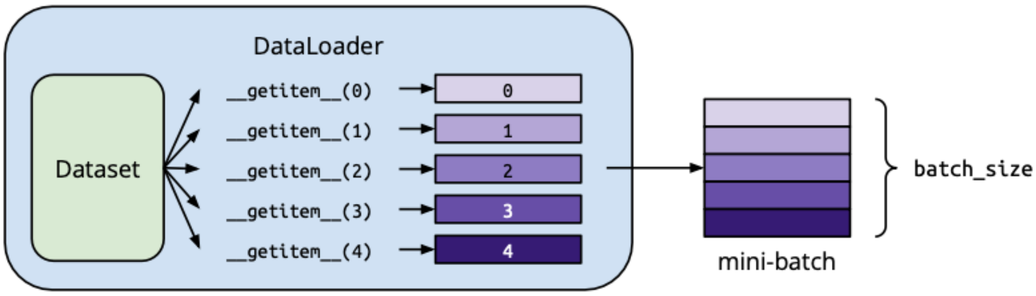

Figure 3: Mechanism of DataLoader

```python
# Create a DataLoader object
dataset = MyDataset(file)
dataloader = DataLoader(dataset, batch_size, shuffle=True)
```

2.1.2 Tensors

Tensors are the basic data structure in PyTorch. They are similar to `numpy.ndarray` but can be used on GPUs. Real world data can be abstracted as tensors: 1D tensor as audio, 2D tensor as black-white images, 3D tensor as RGB images. Here we look at some operations associated with tensors:

Operation	Description
`x=torch.tensor([[1,2],[3,4]])` `x=torch.from_numpy(np.array([[1,2],[3,4]]))`	Create a tensor `[[1,2],[3,4]]`
`x=torch.zeros([1,2,3])`	Create a tensor with shape `[1,2,3]` filled with zeros
`x=torch.ones([1,2,3])`	Create a tensor with shape `[1,2,3]` filled with ones
`x.unsqueeze(dim)`	Add a dimension at the `dim` position
`x.squeeze(dim)`	Remove the dimension at the `dim` position
`torch.cat[x,y,z], dim=d`	Concatenate tensors `x`, `y`, `z` along the `d` dimension.
`torch.FloatTensor(x)`	Convert a numpy array `x` to a 32-bit float tensor
`torch.longTensor(x)`	Convert a numpy array `x` to a 64-bit signed integer tensor
`x.to(device)`	Move tensor `x` to device, where `device` is `'cpu'` or `'cuda'`
`x.requires_grad=True`	Set the tensor `x` to require gradient

Table 1: Operations on Tensors

An illustration of the mechanism of automatic gradient calculation of Tensors is shown in figure 4.

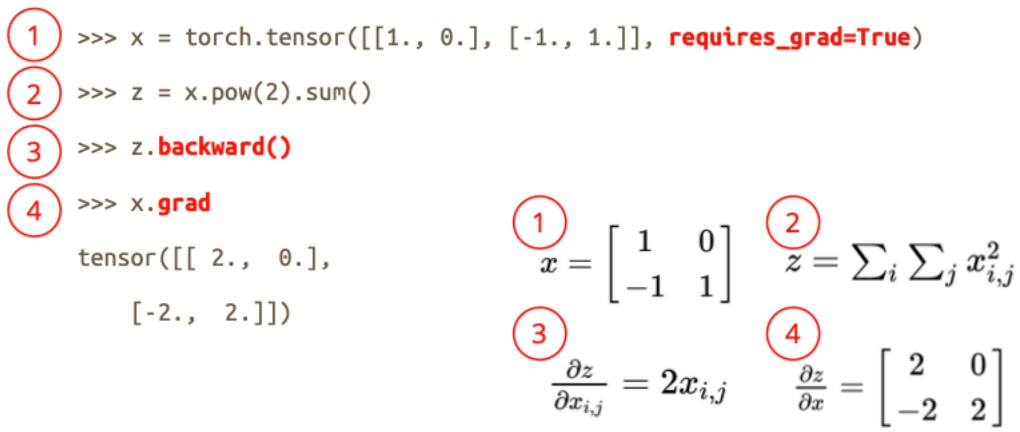

Figure 4: Mechanism of Automatic Gradient Calculation

2.2 Training and Validation

2.2.1 Define Neural Network

We use `torch.nn` to define the neural network layers. One of the most commonly used layer is **Linear Layer**, which is a fully connected layer. The input and output of a linear layer are both tensors. The output tensor is calculated by $\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$. One linear layer can be constructed by

```
layer = torch.nn.Linear(in_features, out_features)
```

where `in_features` is the size of the input tensor and `out_features` is the size of the output tensor. The `nn.Linear` layer has two attributes: `weight` and `bias`:

- `weight`: `layer.weight.shape = (out_features, in_features)`
- `bias`: `layer.bias.shape = (out_features)`

2.2.2 Non-linear Activation Functions

Nonlinear activation functions are used to introduce nonlinearity to the neural network.

- **ReLU**: Rectified Linear Unit, $f(x) = \max(0, x)$, `torch.nn.ReLU()`
- **Sigmoid**: $f(x) = \dfrac{1}{1 + e^{-x}}$, `torch.nn.Sigmoid()`

A typical example of building a neural network is shown in the following code. We define a class `MyModel` that inherits from `torch.nn.Module`. The `forward` function defines the forward pass of the neural network that compute the output of our neural network.

```
import torch.nn as nn
class MyModel(nn.Module):
    # Initialize the model and define the network layers
    def __init__(self):
        super(MyModel, self).__init__()
        self.net = nn.Sequential(
```

```python
            nn.Linear(10, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )
    # Compute output of the neural network
    def forward(self, x):
        return self.net(x)
```

2.2.3 Loss Function

Different loss functions are adopted for different tasks:

- **Regression** ⇒ Mean Squared Error, `criterion = torch.nn.MSELoss()`
- **Classification** ⇒ Cross Entropy Loss, `criterion = torch.nn.CrossEntropyLoss()`

Then we can compute the loss by `loss = criterion(model_output, expected_value)`

2.2.4 Optimization Algorithm

Gradient-based optimization algorithms that adjus network parameters are adopted to reduce error. We can use `torch.optim` to define the optimizer. For example, we can use Stochastic Gradient Descent (SGD) as the optimizer:

```python
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum = 0)
```

After defining the optimizer, in the training loop, for every batch of data, we can update the parameters by

- Call `optimizer.zero_grad()` to reset the gradients of the parameters.
- Compute the loss by `loss = criterion(model_output, expected_value)`
- Call `loss.backward()` to backpropagate gradients of prediction loss.
- Call `optimizer.step()` to update the parameters.

A sample code is given below: (A simplified version for readers to understand the basic structure of the deep learning pipeline)

```python
# Dataset and DataLoader
dataset = MyDataset(file)
train_loader = DataLoader(train_dataset, batch_size = 16, shuffle = True)
valid_loader = DataLoader(valid_dataset, batch_size = 16, shuffle = True)
test_loader = DataLoader(test_dataset, batch_size = 16, shuffle = False)

# Model, Loss, Optimizer
model = MyModel().to('cuda')
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0)

# Training Loop
for epoch in range(epochs):           # iterate over epochs
    model.train()                     # set model to training mode
```

```python
    for x, y in train_loader:       # iterate over data
        optimizer.zero_grad()       # Reset gradients
        pred = model(x)             # Forward pass
        loss = criterion(pred, y)   # Compute loss
        loss.backward()             # Backpropagation
        optimizer.step()            # Update parameters

# Validation Loop
model.eval()                        # set model to evaluation model
loss_record = []                    # record the loss
for x, y in valid_loader:
    x, y = x.to('cuda'), y.to('cuda')   # move data to GPU
    with torch.no_grad():           # disable gradient calculation
        pred = model(x)             # Forward pass
        loss = criterion(pred, y)   # Compute loss
    loss_record.append(loss.item()) # record the loss
mean_valid_loss = sum(loss_record) / len(loss_record) # compute the mean loss
```

It's worth noticing that

- `model.train()` and `model.eval()` are used to set the model to training mode and evaluation mode respectively. The difference between the two modes is that **some layers like dropout and batch normalization behave differently in training and evaluation mode.**
- `torch.no_grad()` is used to disable gradient calculation. It is used when we don't need to compute the gradient, for example, when evaluating the model.

2.3 Testing

After the model is trained and validated on the training and validation set, we can give our prediction on the test set. The quality of the model on the test set can be evaluated on platforms like Kaggle.

```python
# Test Loop
model.eval()                        # set model to evaluation model
preds = []                          # record the predictions
for x in test_loader:
    x = x.to('cuda')                # move data to GPU
    with torch.no_grad():           # disable gradient calculation
        pred = model(x)             # Forward pass
        preds.append(pred)          # record the predictions
```

2.4 Save and Load Model

- **Save the model:** as a dictionary, `torch.save(model.state_dict(), 'model_path')`
- **Load the model:** load the dictionary, `ckpt = torch.load('model_path')`, then load the model by `model.load_state_dict(ckpt)`