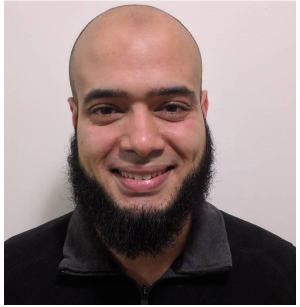
Machine Learning
PyTorch Dataset

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



PyTorch Dataset

- The **Dataset class** serves as an **abstract interface** to represent and manipulate a dataset.
- It is a part of the torch.utils.data package and is particularly useful for loading and preparing **large datasets in parallel**.
- The Dataset class is typically subclassed to create a **custom dataset** tailored for a specific problem.
    - **__len__**: This method returns the size of the dataset.
    - **__getitem__**: This method retrieves the sample at the **given index idx**.

Example for Regression

- Assume you have X, y data.
- All you need return an example corresponding to idx
- PyTorch will accumulate the batch for you

```python
class CustomRegressionDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.FloatTensor(X)
        self.y = torch.FloatTensor(y)

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

Example for Classification

```python
from PIL import Image

class CustomImageDataset(Dataset):
    def __init__(self, image_paths, labels, transform=None):
        self.image_paths = image_paths
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        image = Image.open(self.image_paths[idx]).convert('RGB')
        label = self.labels[idx]

        if self.transform:
            image = self.transform(image)

        return image, label
```

Multiple Returns

- If you have multiple items to return, just use a dictionary

```python
class CustomImageDatasetV2(Dataset):
    def __init__(self, image_paths, labels, metadata, transform=None):
        self.image_paths = image_paths
        self.labels = labels
        self.metadata = metadata
        self.transform = transform

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        image = Image.open(self.image_paths[idx]).convert('RGB')
        label = self.labels[idx]
        meta = self.metadata[idx]

        if self.transform:
            image = self.transform(image)

        sample = {'image': image, 'label': label, 'metadata': meta}
        return sample
```

Data Loader

- Finally, we create a data loader object that takes our dataset
- We configure it (e.g. batch size, shuffling)
- It reads batches of data

```python
from torch.utils.data import DataLoader

X = torch.rand(1000, 65)
y = 5 * torch.sum(X, dim=1)

dataset = CustomRegressionDataset(X, y)
data_loader = DataLoader(dataset, batch_size=32, shuffle=True)

for epoch in range(10):
    for batch_idx, (X_batch, y_batch) in enumerate(data_loader):
        print(X_batch.shape)
```

Regressor Again: Train

- Let’s now use data loader with our regressor

```python
train_dataset = CustomRegressionDataset(x_train, y_train)
train_data_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

for epoch in range(10):
    for batch_idx, (X_batch, y_batch) in enumerate(train_data_loader):
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        print(f"Epoch {epoch+1}, Batch Loss: {loss.item()}")
```

Regressor Again: Test

```python
x_test = torch.rand(10, input_dim)
y_test = 5 * torch.sum(x_test, dim=1)

test_dataset = CustomRegressionDataset(x_train, y_train)
# avoid shuffling in testing for easy debugging
train_data_loader = DataLoader(train_dataset, batch_size=1, shuffle=False)

model.eval()  # Set the model to evaluation mode
with torch.no_grad():
    for batch_idx, (X_batch, y_batch) in enumerate(train_data_loader):
        y_predict = model(X_batch)
        print(f"Prediction: {y_predict} vs gt {y_batch}", )
```

num_workers

- The num_workers argument specifies how many **subprocesses** are used
  - The default, 0, means that the data will be loaded in the **main process**
    - **slow down training** for datasets that are large or require a lot of CPU time to load.
  - data_loader = DataLoader(custom_dataset, batch_size=64, shuffle=True, **num_workers=4**)
  - Each worker will create its **own copy of the data**, which can increase memory usage
- Tip: if you want to debug, make sure to user num_workers=0

Transformations

- PyTorch provide simple API to transform (preprocess) inputs, focused on images (e.g. augment or normalize the data)
- Assume our network is doing a general image classification
  - Input size is 224
  - We expect some surrounding areas are not useful
  - So we can resize to (256x256), then crop the center of (224x224)
  - Now we can convert it to a tensor
    - Convert the PIL Image to a tensor **(also scales to [0, 1])**
-

```
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225]),
])
```

Transformations

- It was very common for Imagenet classifiers to compute pixel mean/std and normalize using it as in previous examples
- Nowadays **[0, 1]** is typically good enough, which to tensor is doing
- Or **[-1, 1]**: here is an example

```
transform2 = transforms.Compose([
    transforms.ToTensor(),  #                  # Normalize to [0, 1]
    transforms.Lambda(lambda x: 2 * x - 1)  # Normalize to [-1, 1]
])
```

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."