6.0 NEED OF Dataset & DataLoader CLASS

24 August 2025 02:48 AM Avinash Yadav

⭐ Why do we need the 'dataset' and 'dataloader' classes?

So there is a problem in our code that we wrote while writing a neural network pipeline using 'nn.module'

This was code that we wrote :-

PyTorch - Temp.py

```python
class MySimpleNN(nn.Module):
    def __init__(self, num_features):
        super().__init__()
        self.linear = nn.Linear(num_features, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, features):
        out = self.linear(features)
        out = self.sigmoid(out)
        return out

model = MySimpleNN(X_train_tensor.shape[1])

learning_rate = 0.1
epochs = 50

loss_function = nn.BCELoss()

optimizer = torch.optim.SGD(
    model.parameters(),
    lr=learning_rate
)

for epoch in range(epochs):
    y_pred = model(X_train_tensor)
    loss = loss_function(y_pred, y_train_tensor.view(-1, 1))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

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

A simple model with only one neuron

The training pipeline where we are building the model, optimizer, and within each epoch we are doing forward pass, loss calculation, clearing gradient, .backward

The biggest flaw in this training pipeline is that → for training the model, we are using batch gradient descent that means, to update our parameters for once we are using the whole dataset at once i.e. we are sending whole data to forward pass → on the basis of which we are calculating loss → And based on those

loss ...
clearing gradient. backward pass and parameter updation.

loss
→ And based on those loss, we are calculating gradient &

→ finally on the basis of those gradient, we are updating old gradient.

→ So, generally we do not use the batch gradient descent because of 2 bigger reasons :-

1.) The first bigger problem is that, batch gradient descent is memory inefficient

2.) Second problem is that, batch gradient descent does not give better convergence, because of which we may not be able to reach better / optimal values of the parameters.

Therefore what we were doing earlier, rather than loading the entire data, load our data in batches.

i.e. Suppose we have a data of 1000 rows, so rather than taking 1000 rows in memory and performing gradient descent on that, it is better to divided our data in 10 batches of each batch having 100 rows & for each batch perform → forward pass → loss calculation → gradient calculation → apply gradient descent.

perform → forward pass → loss calculation → gradient calculation → apply gradient descent.

Once this is done for 1 batch, the load the next batch of 100 rows and perform the same operations.

Repeat it for all the 10 batches.

⇒ This is what called as Mini Batch Gradient Descent

Simple Solution to Implement Mini Batch Gradient Descent in our above code :⇒

Same code as that of above.

```
class MySimpleNN(nn.Module):
    def __init__(self, num_features):
        super().__init__()
        self.linear = nn.Linear(num_features, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, features):
        out = self.linear(features)
        out = self.sigmoid(out)
        return out

model = MySimpleNN(X_train_tensor.shape[1])

learning_rate = 0.1
epochs = 50

loss_function = nn.BCELoss()

optimizer = torch.optim.SGD(
    model.parameters(),
    lr=learning_rate
)

batch_size = 32
n_samples = len(X_train_tensor)

for epoch in range(epochs):
    # Simply loop over the dataset in chunks of `batch_size`
    for start_idx in range(0, n_samples, batch_size):
        end_idx = start_idx + batch_size
        X_batch = X_train_tensor[start_idx:end_idx]
```

defined batch size

$\frac{320}{32} = 10$ Batches

i.e. total of 320 rows got divided into 10 Batches with each batch of size 32 rows

Starting the training process

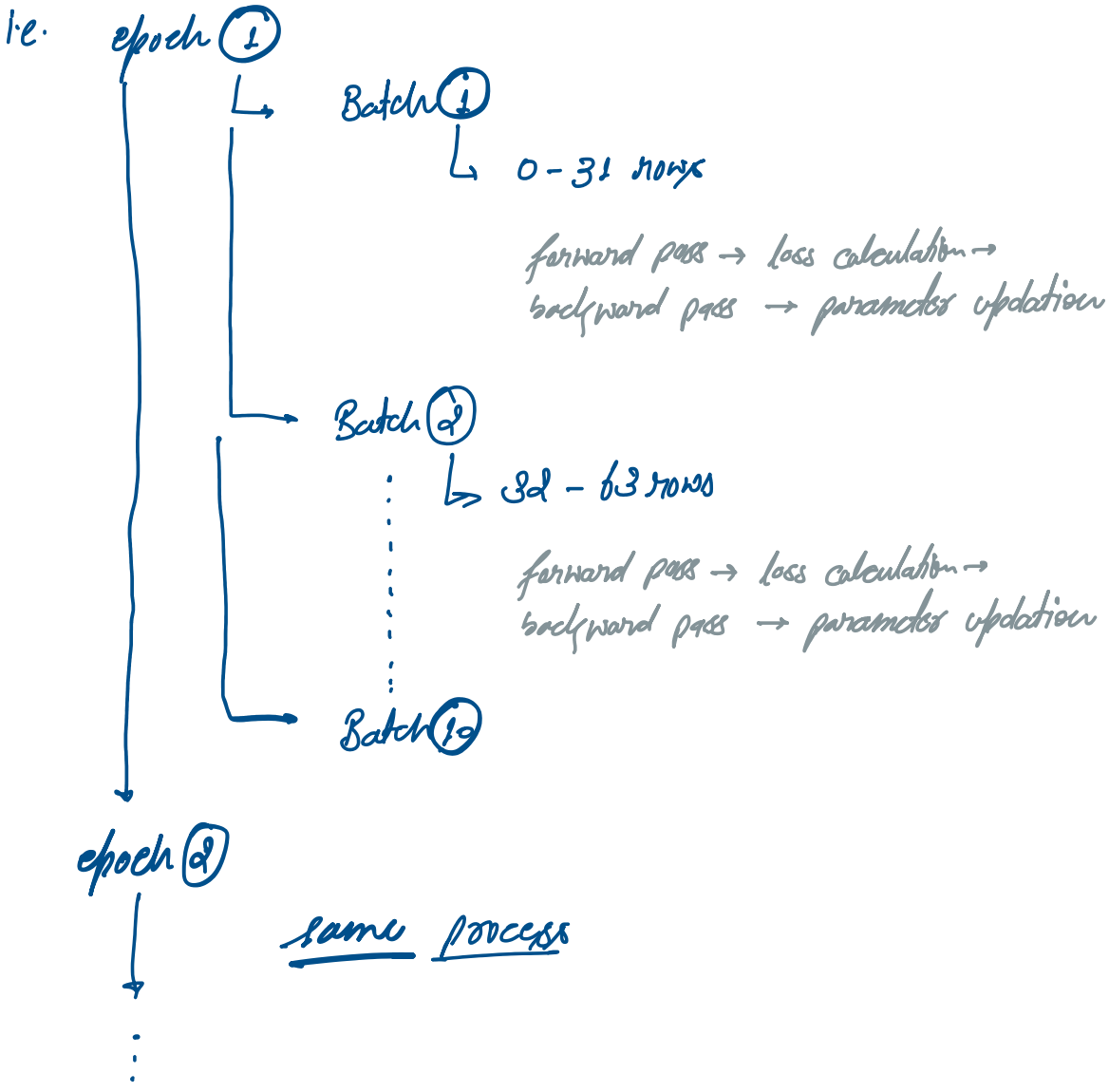
Now we are using of

we are using 2 loops 1st for epochs and for moving through batches

```
for start_idx in range(0, n_samples, batch_size):
    end_idx = start_idx + batch_size
    X_batch = X_train_tensor[start_idx:end_idx]
    y_batch = y_train_tensor[start_idx:end_idx]

    # Forward Pass
    y_pred = model(X_batch)
    loss = loss_function(y_pred, y_batch.view(-1, 1))

    # Update Step
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

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

Size 32 rows

Now for each epoch, it will take each batch of size 32 rows and perform forward pass → loss calculation → backward pass → parameter updation



epoch (50)

Same process

So now, this much simple it is to apply mini batch gradient descent using manual code although this code will work but there are some problems in this code.

So the problems are:

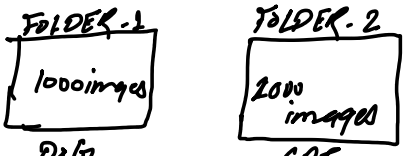
1.) No standard interface for data

that means, now we are very easily able to make the batch with the help of X_train_tensor and y_train_tensor

```
PyTorch - Temp.py

batch_size = 32
n_samples = len(X_train_tensor)

for epoch in range(epochs):
    # Simply loop over the dataset in chunks of `batch_size`
    for start_idx in range(0, n_samples, batch_size):
        end_idx = start_idx + batch_size
        X_batch = X_train_tensor[start_idx:end_idx]
        y_batch = y_train_tensor[start_idx:end_idx]

        # Forward Pass
        y_pred = model(X_batch)
        loss = loss_function(y_pred, y_batch.view(-1, 1))

        # Update Step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

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

Here we supposed that we have the complete dataset with us.
But there are some scenarios where acquiring the dataset is itself a big task

image classification
⇒ Dog v/s Cat



now from this data we have to pick the images & make our X_train_tensor and y_train_tensor
so once we have made them, then only we can create our batches.

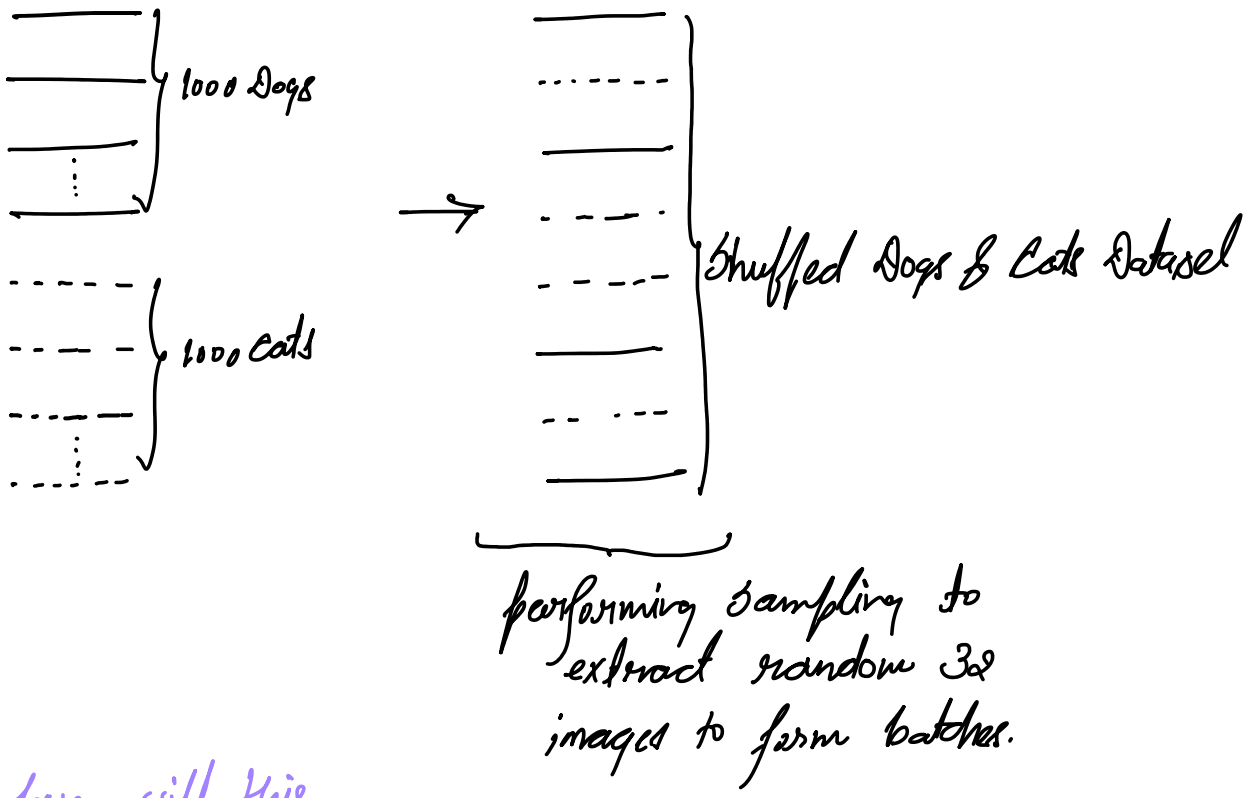
But in our discussed code, we did not mention where this logic will get implemented

2.) No Easy Way to apply transformations:-

There can be scenarios, that we wanted to make batches of 32 images and before sending each batch for training we want to apply some sort of transformation lets say RGB → BGW

But this is not mentioned in our discussed approach.

3.) Shuffling and Sampling



But how will this this shuffling and sampling will be achieved, is not discussed in our code.

4.4 Batch Management & Parallelization

suppose we have to extract batches of 32 images.
and we have to extract multiple batches parallelly
But the implementation of this parallelization is not discussed in our code.

So In order to solve all these problems, we get 2 classes from PyTorch namely 'Dataset Class' and 'Dataloader Class'