PyTorch for Deep Learning

Thursday, 19 December 2024 15:19

Lets break it down into 3 parts :

I : Dataset handling :

Inherit from pytorch Dataset

```
class PlayingCardDataset(Dataset):
    def __init__(self):
        pass

    def __len__(self):
        pass

    def __getitem__(self, idx):
        return
```

- `__init__` → init method
- `__len__` → Dataloader needs to know how many examples we have
- `__getitem__` → takes in index & return 1 item

```
class PlayingCardDataset(Dataset):
    def __init__(self, data_dir, transform=None):
        self.data = ImageFolder(data_dir, transform=transform)

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

    def __getitem__(self, idx):
        return self.data[idx]

    @property
    def classes(self):
        return self.data.classes
```

- `__init__` → init takes data directory, transform
- `ImageFolder` → Image folder helps in giving the images class labels if they are grouped accordingly.
- `return len(self.data)` } return data count
- `return self.data[idx]` ↳ return a particular item based on index
- `classes` → To get the classes of data

Now the transformation : Mainly to make sure all the i/p have consistent data sizes.

```
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
])

data_dir = '/kaggle/input/cards-image-datasetclassification/train'
dataset = PlayingCardDataset(data_dir, transform)
```

- `transform` } to resize & convert it into tensor form
- `data_dir`, `dataset` } reassigning the dataset

Now we have dataloaders, that will batch the data based on set batch size. Dataloader () wraps our dataset while feeding into our model.

PYTORCH Model :

```
class SimpleCardClassifer(nn.Module):
    def __init__(self, num_classes=53):
        super(SimpleCardClassifer, self).__init__()
        # Where we define all the parts of the model
①       self.base_model = timm.create_model('efficientnet_b0', pretrained=True)
②       self.features = nn.Sequential(*list(self.base_model.children())[:-1])

        enet_out_size = 1280
        # Make a classifier
③       self.classifier = nn.Linear(enet_out_size, num_classes)

    def forward(self, x):
        # Connect these parts and return the output
④       x = self.features(x)
        output = self.classifier(x)
        return output
```

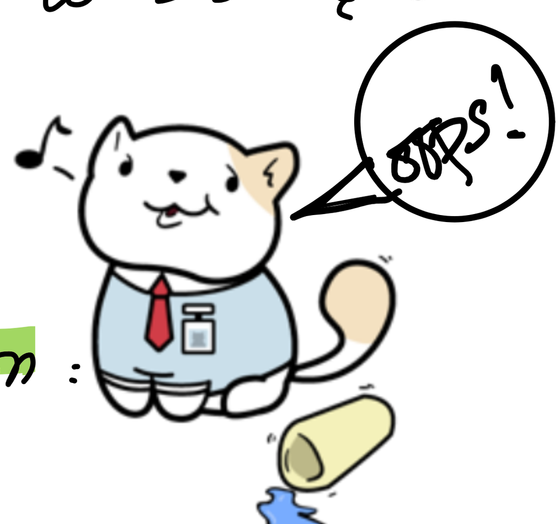
- `(nn.Module)` } inheriting from.
- `num_classes=53` → setting as our o/p classes
- `super(...).__init__()` → Calling constructor of parent class.
- `nn.Sequential` → arrange them in sequential order to process it
- `self.base_model.children())[:-1]` → extract all layers except last.
- `self.classifier` → A fully connected dense layer that maps 1280 to 53
- `forward` ↳ is called by default once model is initialed

In detail what does 1, 2, 3 & 4 do?

① we chose the model we want, here its Efficientnet Bo. & its pretrained on image net

② Here we are separating the data layers of the trained model from its last layers as the last layer maps to 1208 classes as it was trained on imagenet data. When we pass our i/p into features, we have a vector o/p which contain feature map. This isnt yet the o/p as it has not mapped the feature space to classes.

③ self. classifier is defined with required class_num & a linear mapping converts 1208 class mapping to 53. & hence the o/p layer.



TRAINING LOOP :

declaring the LOSS function :

```
# Loss function
criterion = nn.CrossEntropyLoss()
# Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

we use entropy loss as it is classifier task. & the optimiser we are using is adam.

```
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
])

train_folder = '../input/cards-image-datasetclassification/train/'
valid_folder = '../input/cards-image-datasetclassification/valid/'
test_folder = '../input/cards-image-datasetclassification/test/'

train_dataset = PlayingCardDataset(train_folder, transform=transform)
val_dataset = PlayingCardDataset(valid_folder, transform=transform)
test_dataset = PlayingCardDataset(test_folder, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
```

Now, we have to resize all our data. split them as test, train, Validation. & separately load the data using data loader. mentioning the batch size. Note that we shuffled training data but not the test / validation data.

```
num_epochs = 5
train_loss, val_losses = [], []

model = SimpleCardClassifer(num_classes=53)

for epoch in range(num_epochs):
    # Set the model to train
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * images.size(0)
    train_loss = running_loss / len(train_loader.dataset)
    train_losses.append(train_loss)
```

- `train_loss, val_losses = [], []` } lists to track losses.
- `loss = criterion(outputs, labels)` } comparing the actual labels & o/p
- `loss.backward()` → Bck propogation
- `optimizer.step()` → takes a step

Now move on to training the data.

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

use this for the model to use GPU if available.

we can make the model & dataset to use the same device by mentioning :

```
model.to(device)
```

```
images, labels = imges.to(device),
                 labels.to (device)
```

We can use tqdm for progress from tqdm.notebook use tqdm & wrap it in the forloop.