MNIST problem using PyTorch

```
from torchvision import datasets
import torchvision.transforms as transforms

transform = transforms.ToTensor()
```

Convert data to torch.FloatTensor

```
train_data = datasets.MNIST(root = 'data',
                            train = True,
                            transform = transform)
# similarly for test_data (train = False)

train_loader = torch.utils.data.DataLoader(
                            train_data,
                            batch_size = b_size,
                            num_workers = workers)

# Inspecting the data:

dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy()
```

one batch of the data

one image ← img = np.squeeze(images[1])

Defining the Model:

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

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 10)
```

to prevent overfiting ← `self.dropout = nn.Dropout(0.2)`

```
    def forward(self, x):
```

flatten ← `x = x.view(-1, 28*28)`

apply layer ← `x = F.relu(self.fc1(x)); x = self.dropout(x)`

same for fc2.

```
        x = self.fc3(x)
        return x
```

#question: why do we need activation functions?
to scale the outputs of a layer so that they are a consistent, small value.

```
model = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
# Finally, train the model!
n_epochs = 30   # start small!

for epoch in range(n_epochs):
    train_loss = 0
    for data, target in data_loader:
        optimizer.zero_grad() → clear all the gradients

        output = model(data) forward pass
        loss = criterion(output, target) compute loss

        loss.backward() → backward propagation
        optimizer.step() → parameter update

        train_loss += loss.item() * data.size(0)

    train_loss = train_loss/len(train_loader.dataset)
```

```
# test!      model.eval()
for data, target in test_loader:
    output = model(data)
    loss = criterion(output, target)
    test_loss += loss.item() * data.size(0)
    _, pred = torch.max(output, 1)
    correct =
        np.squeeze(pred.eq(target.data.view_as(pred)))
```

* Question: nn.CrossEntropy, does both of Softmax and NLLLoss. What if we want the output of the model as class probabilities instead of scores?

Answer: use Softmax and NLLLoss separately.

```
    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

test_loss = test_loss/len(test_loader.dataset)
```

```
for i in range(num_classes):
    if class_total[i] > 0:
        accuracy[i] = np.sum(class_correct[i]) /
                      np.sum(class_total[i]))

overall_accuracy = np.sum(class_correct) / np.sum(class_total)
```

Model Validation

```
from torch.utils.data.sampler import SubsetRandomSampler
```

we use this to sample some validation data

```
train_Sampler = SubsetRandomSampler(train_idx)
valid_Sampler = SubsetRandomSampler(valid_idx)
```

```
train_loader( as before , sampler=train_Sampler)
```

we do the same for valid-sampler.

Question: why do we need validation?

Answer: Knowing when to stop training to avoid overfitting
(when valid-loss starts increasing but train_loss keep decreasing)

OpenCV

```
import cv2
import numpy as np
import matplotlib.image as mpimg
```

load → `image = mpimg.imread('img.png')`

make black/white → `gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)`

the filter →
```
filt = np.array([[-1, -2, -1],
                 [0, 0, 0],
                 [1, 2, 1]])
```

Convolute → `filtered_image = cv2.filter2D(gray, -1, filt)`