Handwritten digit Recognition

Below neural net recognizes hand written digits from MNIST dataset. The layering of the network is summarized and shown below.

`conv --> conv --> max pool --> conv --> conv --> max pool --> conv --> conv --> conv`

see comments in code below for details of `input size`, `padding`, `kernel size`, `output size` and `Receptive field` for each layer

```
# shown below are the definitions of the layers of the network

# input = 30x30x1 (padding=1)| kernels = (3x3x1)x32 | output = 28x28x32  | RF = 3x3
self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1) #input -? OUtput? RF

# input = 30x30x32 (padding=1) | kernels = (3x3x32)x64 | output = 28x28x64 | RF = 5x5
self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)

# input = 28x28x64 | maxpool = 2x2 | output = 14x14x64 | RF = 10x10
self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

# input = 16x16x64 (padding=1) | kernels = (3x3x64)x128 | output = 14x14x128 | RF = 12x12
self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)

# input = 16x16x128 (padding=1) | kernels = (3x3x128)x256 | output = 14x14x256 | RF = 14x14
self.conv4 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)

# input = 14x14x256 | maxpool = 2x2 | output = 7x7x256 | RF = 28x28
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

# input = 7x7x256 | kernel = (3x3x256)x512 | output = 5x5x512 | RF=30x30
self.conv5 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3)

# input = 5x5x512 | kernel = (3x3x512)x1024 | output = 3x3x1024 | RF=32x32
self.conv6 = nn.Conv2d(in_channels=512, out_channels=1024, kernel_size=3)

# input = 3x3x1024 | kernel = (3x3x1024)x10 | output = 1x1x10 | RF=34x34
self.conv7 = nn.Conv2d(in_channels=1024, out_channels=10, kernel_size=3)
```

code is well commented to document the details

1. Data representation: MNIST data set is used. Each input image in the training set is 28x28. There are 6000 images in the training dataset and 2000 in the test dataset
2. the model has an accuracy of 98%. The results were evaluated using the test dataset from MNIST

```
0%|          | 0/469 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:42: UserW
Training phase: loss=0.06000012531876564 batch_id=468: 100%|██████████| 469/469 [00:19<00:00, 24.61it/s]
Testing phase: total_test_loss=527.0729377493262 total_correct=9819: 100%|██████████| 79/79 [00:01<00:00, 4
Test set: Average loss: 0.0527, Accuracy: 9819/10000 (98%)
```

3. Loss function: The negative log likelihood loss (nll_loss) was used. It is useful to train a classification problem with C classes and since this is a classification problem, nll_loss was used..

```
from __future__ import print_function
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
```

```
# this class defines the CNN or convulational neural network
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # shown below are the definitions of the layers of the network

        # input = 30x30x1 (padding=1)| kernels = (3x3x1)x32 | output = 28x28x32  | RF = 3x3
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1) #input -? OUtput? RF

        # input = 30x30x32 (padding=1) | kernels = (3x3x32)x64 | output = 28x28x64 | RF = 5x5
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)

        # input = 28x28x64 | maxpool = 2x2 | output = 14x14x64 | RF = 10x10
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # input = 16x16x64 (padding=1) | kernels = (3x3x64)x128 | output = 14x14x128 | RF = 12x12
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)

        # input = 16x16x128 (padding=1) | kernels = (3x3x128)x256 | output = 14x14x256 | RF = 14x14
        self.conv4 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1)

        # input = 14x14x256 | maxpool = 2x2 | output = 7x7x256 | RF = 28x28
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        # input = 7x7x256 | kernel = (3x3x256)x512 | output = 5x5x512 | RF=30x30
        self.conv5 = nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3)

        # input = 5x5x512 | kernel = (3x3x512)x1024 | output = 3x3x1024 | RF=32x32
        self.conv6 = nn.Conv2d(in_channels=512, out_channels=1024, kernel_size=3)

        # input = 3x3x1024 | kernel = (3x3x1024)x10 | output = 1x1x10 | RF=34x34
        self.conv7 = nn.Conv2d(in_channels=1024, out_channels=10, kernel_size=3)

    def forward(self, x):
        # seems to define the forward propogation of the neural network
        x = self.pool1(F.relu(self.conv2(F.relu(self.conv1(x)))))
        x = self.pool2(F.relu(self.conv4(F.relu(self.conv3(x)))))
        x = F.relu(self.conv6(F.relu(self.conv5(x))))
        #x = F.relu(self.conv7(x))
        x = self.conv7(x)
        x = x.view(-1, 10)
        return F.log_softmax(x)
```

```
!pip install torchsummary
from torchsummary import summary
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
```

```
model = Net().to(device)
# print a summary of the model for an input of size 28x28x1 (l x w xchannels)
summary(model, input_size=(1, 28, 28))
```

```
Requirement already satisfied: torchsummary in /usr/local/lib/python3.7/dist-packages (1.5.1)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 28, 28]             320
            Conv2d-2           [-1, 64, 28, 28]          18,496
         MaxPool2d-3           [-1, 64, 14, 14]               0
            Conv2d-4          [-1, 128, 14, 14]          73,856
            Conv2d-5          [-1, 256, 14, 14]         295,168
         MaxPool2d-6            [-1, 256, 7, 7]               0
            Conv2d-7            [-1, 512, 5, 5]       1,180,160
            Conv2d-8           [-1, 1024, 3, 3]       4,719,616
            Conv2d-9             [-1, 10, 1, 1]          92,170
================================================================
Total params: 6,379,786
Trainable params: 6,379,786
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 1.51
Params size (MB): 24.34
Estimated Total Size (MB): 25.85
----------------------------------------------------------------
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:42: UserWarning: Implicit dimension choice for log_
```

```

torch.manual_seed(1)
batch_size = 128

kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
# data loader for training ???.  download=True means download the dataset (training images and labels for images
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
                    transform=transforms.Compose([
                        transforms.ToTensor(),
                        transforms.Normalize((0.1307,), (0.3081,))  # need to understand why we normalize?  even
                    ])),
    batch_size=batch_size, shuffle=True, **kwargs)
# data loader for testing???  download=True means download the dataset (training images and labels for images) i
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False, transform=transforms.Compose([
                        transforms.ToTensor(),
                        transforms.Normalize((0.1307,), (0.3081,))
                    ])),
    batch_size=batch_size, shuffle=True, **kwargs)

```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ../data
```

9913344/? [02:35<00:00, 63716.93it/s]

```
Extracting ../data/MNIST/raw/train-images-idx3-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ../data
```

29696/? [00:36<00:00, 812.75it/s]

```
Extracting ../data/MNIST/raw/train-labels-idx1-ubyte.gz to ../data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ../data/
```

1649664/? [00:36<00:00, 45691.54it/s]

```
# Decorate an iterable object, returning an iterator which acts exactly like the original iterable, but prints a
from tqdm import tqdm
# this is the function that trains the specified 'model' using the 'train_loader', optimizer and others
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    pbar = tqdm(train_loader)
    for batch_idx, (data, target) in enumerate(pbar):  # target is the 'label' for this image..
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()  # for each batch, zero the gradient (from processing of prior batch)

        output = model(data)   # predict using the model for the given data (batch); for each batch, output.shap

        #if (batch_idx in (0, 468)):
        #  print(f"\noutput.shape={output.shape}; prediction for first image in batch: output[0]={output[0]}")

        # The negative log likelihood loss.  The negative log likelihood loss. It is useful to train a classific
        # https://pytorch.org/docs/master/generated/torch.nn.functional.nll_loss.html
        loss = F.nll_loss(output, target)  # compute the loss

        loss.backward()   # do backward propogation; update gradients of tensors in dynamic computation graph, s

        optimizer.step()  # use the updated gradients to adjust the weight of each kernel

        pbar.set_description(desc= f'Training phase: loss={loss.item()} batch_id={batch_idx}')

# this functions tests the trained model 'model' by comparing the predictions done by the trained model with the
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    pbar = tqdm(test_loader)
    with torch.no_grad():   # during testing/inferencing, backprop is not needed; so disable autograd.
        for data, target in pbar:   # target is the 'label' for this image..
            data, target = data.to(device), target.to(device)
            output = model(data)   # make the prediction for the given test data
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability using argmax()
            #  get the correct number of predictions for this batch
            correct += pred.eq(target.view_as(pred)).sum().item()   # tensor.eq() method: computes if one elemen
            pbar.set_description(desc= f'Testing phase: total_test_loss={test_loss} total_correct={correct}')

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

```

model = Net().to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

for epoch in range(1, 2):
    train(model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
```

```
  0%|          | 0/469 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:42: UserWarr
Training phase: loss=0.06000012531876564 batch_id=468: 100%|██████████| 469/469 [00:19<00:00, 24.61it/s]
Testing phase: total_test_loss=527.0729377493262 total_correct=9819: 100%|██████████| 79/79 [00:01<00:00, 40.83i
Test set: Average loss: 0.0527, Accuracy: 9819/10000 (98%)
```

Adding two digits using NN

The fully connected NN shown below performs addition of 2 digits.

Given the input below

```
X[0:5]=tensor([[0., 4.],
        [5., 2.],
        [4., 9.],
        [7., 9.],
        [1., 9.]])
```

the output/predicted value from the NN is below

```
tensor([[ 4.0000],
        [ 7.0000],
        [13.0000],
        [16.0000],
        [10.0000]], grad_fn=<AddmmBackward>)
```

The layering of the fully connected dense network is as shown below:

`input layer --> hidden layer 1 (10 neurons) --> hidden layer 2 ( 10 neurons) --> output layer (1 neuron)`

```
model = torch.nn.Sequential(
    torch.nn.Linear(in_features=10*2, out_features=10),    # input size is 20; one hot encoded input
    torch.nn.Linear(in_features=10, out_features=10),
    torch.nn.Linear(in_features=10, out_features=1),
    )
```

2. Data representation: the input is represented using one hot encoding. The two digits to be added (from 0 to 9) are represented using a vector of length 20.
3. Data generation strategy: random numbers are generated for the training dataset. And the generated numbers are scaled from 0 to 9 and are one hot encoded. See code for details.
4. Shown further above are some sample results..
5. Loss function: MSE loss function torch.nn.MSELoss() is used as it is well suited for continuous values

```
############
```

```
# https://discuss.pytorch.org/t/convert-int-into-one-hot-format/507/25
############
import torch

def one_hot_embedding(labels, num_classes):
    """Embedding labels to one-hot form.

    Args:
      labels: (LongTensor) class labels, sized [N,].
      num_classes: (int) number of classes.

    Returns:
      (tensor) encoded labels, sized [N, #classes].
    """
    y = torch.eye(num_classes)
    return y[labels]

one_hot_embedding([0,1,2,3,4,5,6,7,8,9],10)
```

```
tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

```
#######
#  input to NN is **one hot encoded**
#######
import torch
import numpy as np
from tqdm import tqdm
from  torchsummary import summary

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

N = 1000  # number of samples
input_dim_size = 2  # input dimension
output_dim_size = 1  # output dimension

X = torch.rand(N, input_dim_size)  # training data; dim = 1000 x 2; values is 0 < x < 1
X *= 10  # scale it to 0 < x < 10
X = X.floor()
X.to(device)
print(f"X.shape={X.shape};  X[:8]={X[:8]}") #X.mean()={X.mean()}; X.std()={X.std()};"): RuntimeError: Can only c

y = torch.sum(X, axis=-1).reshape(-1, output_dim_size)  # training label/target; dim = sum(X) transforms 1000x2
print(f"torch.sum(X, axis=-1).shape={torch.sum(X, axis=-1).shape}")
print(f"y.shape={y.shape}")
#print(f"y.shape={y.shape}")

X_onehot = one_hot_embedding(X.long(),10)    # X[0:,:] or X is vector of size 1000x2; X_onehot is 1000x2x10
print(f"X_onehot.shape={X_onehot.shape}")
print(f"X[0:5]={X[0:5]}")    # 5x2
print(f"X_onehot[0:5]={X_onehot[0:5]}")    # 5x2x10
# reshape
X_onehot = X_onehot.reshape(N, 10*2)  # 1000x2x10 to 1000x20
print(f"X_onehot[0:5]={X_onehot[0:5]}")    # 5x2x10
X_onehot.to(device)

lr = 1e-2  # Learning rate
```

```
lr = 1e-2  # Learning rate

# model: 2 inputs --> 1 neuron (output neuron)
# model = torch.nn.Sequential(torch.nn.Linear(in_features=input_dim_size, out_features=output_dim_size))  # mode

# ----------------------------------------------------------------
#         Layer (type)               Output Shape         Param #
# ================================================================
#             Linear-1             [-1, 1, 1, 10]             210          # 20 inputs x 10 neurons = 200; 10 bi
#             Linear-2             [-1, 1, 1, 10]             110          # 10 inputs x 10 neurons = 100; 10 bi
#             Linear-3              [-1, 1, 1, 1]              11          # 10 inputs x 1 neuron = 10; 1 biases
# ================================================================
# model: 20 inputs --> 10 neurons --> 10 neurons --> 1 neuron (output)
model = torch.nn.Sequential(
    #torch.nn.Linear(in_features=input_dim_size, out_features=10),    # input size is 2; input as a number (not
    torch.nn.Linear(in_features=10*2, out_features=10),    # input size is 20; one hot encoded input
    torch.nn.Linear(in_features=10, out_features=10),
    torch.nn.Linear(in_features=10, out_features=1),
    )
model.to(device)

criterion = torch.nn.MSELoss()  # loss function
optimizer = torch.optim.Adam(model.parameters(), lr=lr)  # optimizer

#pbar_epochs = tqdm(range(1000))
for epoch in range(1000):
    #y_pred = model(X)  # forward step; y_pred is 1000x1; X is 1000x2
    model.to(device)
    y_pred = model(X_onehot)  # forward step; y_pred is 1000x1; X_onehot is 1000x20
    loss = criterion(y_pred, y)  # compute loss; y is 1000x1; loss is 1000x1
    loss.backward()  # backprop (compute gradients)
    if epoch == 0:
      print(f"X.shape={X.shape}; y.shape={y.shape}; y_pred.shape={y_pred.shape}; type(loss)={type(loss)}")   # l
    optimizer.step()  # update weights (gradient descent step)
    optimizer.zero_grad()  # reset gradients
    if epoch % 200 == 0:
      print(f"epoch={epoch}, loss={loss.item():.6f}")
    #pbar.set_description(desc=f"[EPOCH]: {epoch}, [LOSS]: {loss.item():.6f}")

#summary(model, input_size=(1,1,2))  # model has 3 parameters:  2 weights, one each for each input + 1 bias = 3
summary(model, input_size=(1,1,20))

#print(f"model( torch.tensor([[1.,2.], [3.,4.], [100.,200.], [1000.,2000.]]) )={model( torch.tensor([[1.,2.], [3
model( X_onehot[0:5] )
```

```
X.shape=torch.Size([1000, 2]);  X[:8]=tensor([[0., 4.],
        [5., 2.],
        [4., 9.],
        [7., 9.],
        [1., 9.],
        [3., 8.],
        [2., 3.],
        [3., 4.]])
torch.sum(X, axis=-1).shape=torch.Size([1000])
y.shape=torch.Size([1000, 1])
X_onehot.shape=torch.Size([1000, 2, 10])
X[0:5]=tensor([[0., 4.],
        [5., 2.],
        [4., 9.],
        [7., 9.],
        [1., 9.]])
X_onehot[0:5]=tensor([[[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
         [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.]],
```

```
        [[0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]],

        [[0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]],

        [[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]]])
X_onehot[0:5]=tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.,
         0., 0.],
        [0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.,
         0., 0.],
        [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 1.],
        [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 1.],
        [0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 1.]])
X.shape=torch.Size([1000, 2]); y.shape=torch.Size([1000, 1]); y_pred.shape=torch.Size([1000, 1]); type(loss)=
epoch=0, loss=101.791016
epoch=200, loss=0.000000
epoch=400, loss=0.000000
epoch=600, loss=0.000000
epoch=800, loss=0.000000
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1             [-1, 1, 1, 10]             210
            Linear-2             [-1, 1, 1, 10]             110
            Linear-3              [-1, 1, 1, 1]              11
================================================================
Total params: 331
Trainable params: 331
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
```

Combining digit recognition with addition.

This is a TODO and couldn't be completed.

The idea is that

- the output of the digit recognition NN is a x1x10 (for 10 classes) tensor.
- argmax() is used to identify the digit the imge represents (maximum probablity of the 10 classes).
- The identified digit needs to one hot encoded.
- this one hot encoded input, along with a random input number one hot encoded, becomes the input of the above addition NN. so the input here is of the shape 1x20 (2 digits, one hot encoded)
- the input to this composite network is both the random digit and the image..
- the output needs to be the recoginized digit and the added value.

✓ 0s completed at 5:06 AM