Deep Learning Tutorial

CMPE 685 Teaching Assistant: Nilesh Pandey np9207@rit.edu

1. Introduction:

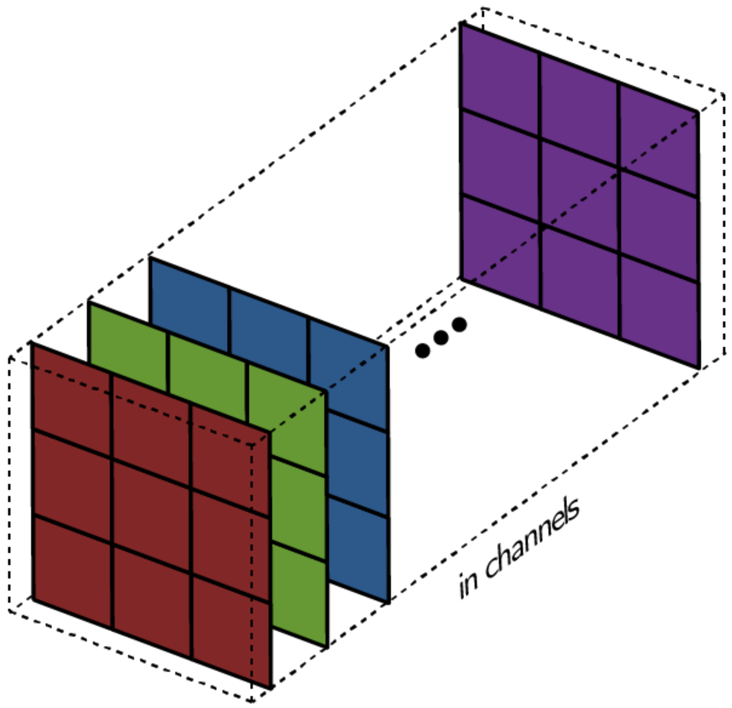
Pytorch is a deep learning framework released and maintained by facebook. Unlike tensorflow, you don't need to maintain a session to build your network graph. It works as pure python. From pytorch 1.0 it has been merged with caffe2, so it is easier to write deployment code in C++.
Pytorch tutorial https://github.com/yunjey/pytorch-tutorial
Pytorch documentation https://pytorch.org/docs/stable/index.html
Connect and ask for the pytorch community at https://discuss.pytorch.org/ .

2. Pytorch Tutorial

A. Layers

i. Convolutions

Convolution forms the basis of deep neural networks or deep learning framework. For visualization of how convolution works, visit Visualization of convolution. Convolution is a simple but effective method to learn weights that are specific to some features.



In the above image, we see a matrix (tensor) of size NxNxC. Here NxN is the size of the matrix and C is depth (number of stacked matrices). Each matrix in the above image is a filter of any type, high pass filter, low pass filter, or sobel or any filter that have yet to discover. The convolution learns weights or neurons which will extract features according to the filter they have learned. To get more intuition checkout Convolution.

Pytorch

```
torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)
```

** kernel_size = 3

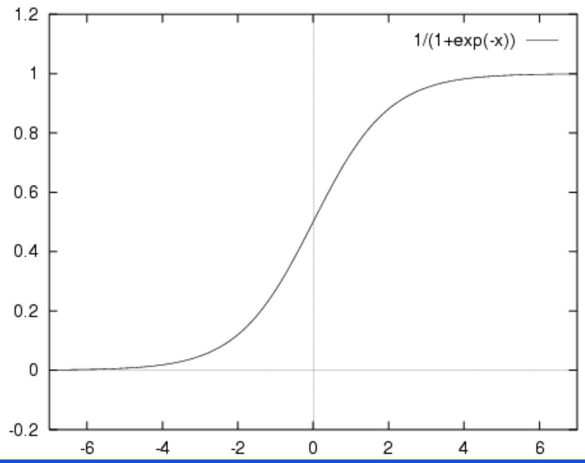
** stride = 1, if stride 2 it will reduce size by 2.

** padding = 1 for kernel_size 3, and 2 for kernel_size 5.

ii. Activation Function

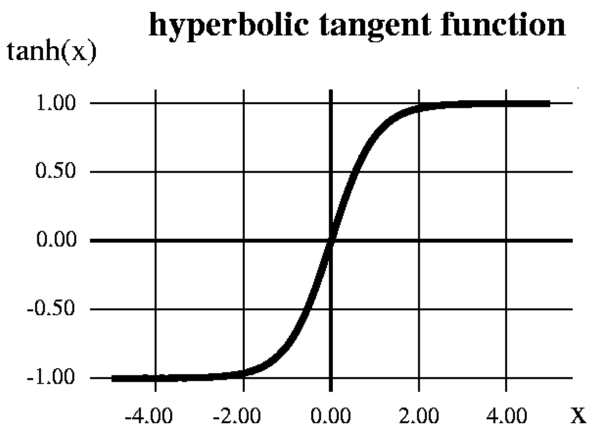
Activation functions introduce nonlinearity in the network. There are 3 commonly used activation functions. RELU is most common in CNNs.

1. Sigmoid



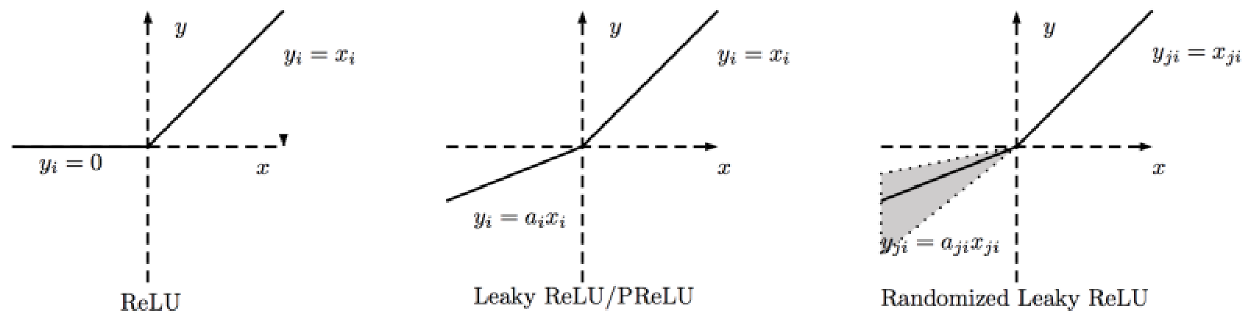
Sigmoid ranges from 0 - 1. Important parts of activation functions are the end and center part. The center part is linear. Sigmoid makes the backprograted values zero often. This causes failure in learning and sigmoid has low convergence.

2. Tanh



Tanh ranges from -1 to 1. It helps to solve the problem of gradients during backpropagation. The problem of saturation still remains with tanh.

3. Relu



Relu is free from saturation on the positive side of the graph. The negative part becomes zero, which resulted in a leaky Relu. Relu is used in CNNs because of numerical stability and fewer calculations.

```
Pytorch Implementation of Activation functions.
torch.nn.Sigmoid()
torch.nn.ReLU()
torch.nn.Tanh()
```

iii. Normalization

1. Batch Normalization

Fact 101: Normalization layers can learn weights and also they cannot learn, it depends on the type of normalization you perform in the network. Normalization can the change the whole result.

Why normalize the layers? When one can normalize the image before passing to a network. The reason is as the time new input come and pass by in the deep networks, there is a covariance shift. The covariance shift can be thought of as error in the weights learned from the previous image to the new image. This error increases over time as it gets accumulated. Which eventually gives wrong result, to minimize such error / covariance shift usage of normalization is always recommended. It is also a kind of a regularization to prevent it from overfitting. More info can be found at Batch Normalization.

torch.nn.BatchNorm2d(*num_features*)

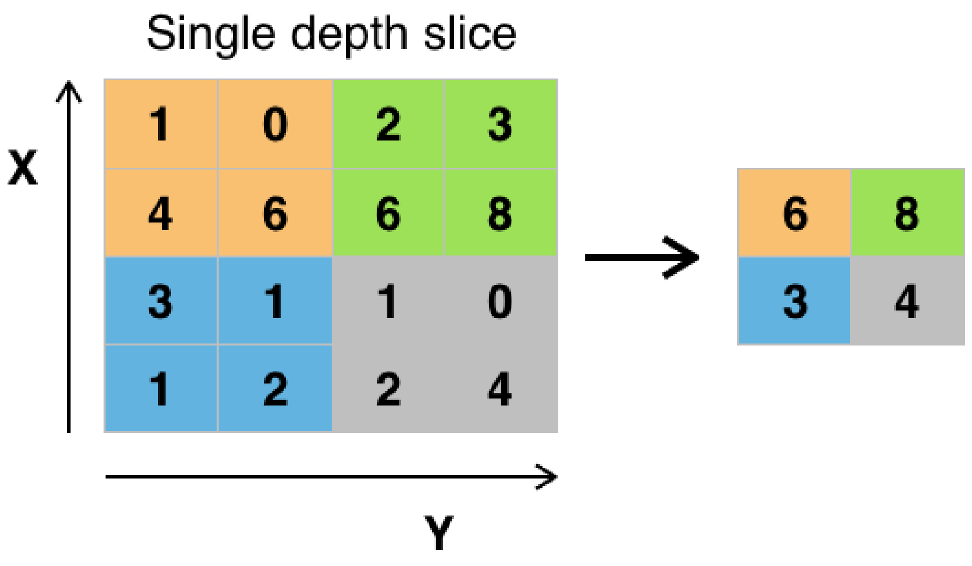
a. Drop out

Dropout works as a regularization layer. You randomly make some of the neurons zero, which gives 0 output. To be more precise the output of the convolution when passed to the activation layer decides the actual output, if we apply dropout layer then the activation layer forces some of the output to be zero.
It helps with overfitting of the neural network. Since proposal of normalization layer, dropout use is rare.
torch.nn.Dropout(P), where is probability of weights to be dropped or make zero. Usually P is kept at 0.5.

b. Pooling

1. Max Pool

We have seen functions of above layers, which sounds important and it is. The Maxpool just does the down sampling.



Example of Maxpool with a 2x2 filter and a stride of 2

Max pool literally means taking the maximum value from the kernel_sized window. In the above example kernel_sized is 2. It is non learnable layer. Nowadays, people prefer to replace maxpool by convolution layer with stride 2, which does the same thing but with learnable weights. Depending on the resource available one needs to select between the two.

2. Average Pool

As the name suggests, instead of max pool the layer does averaging.
torch.nn.AvgPool2d(*kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True*)

*** I prefer torch.nn.AdaptiveAvgPool2d((H,W)) which does the average but gives the desired size of the input image.
If input is 64x64x7 than output of the torch.nn.AdaptiveAvgPool2d((7,7)) will be 7x7x7.

c. Fully Connected layer

Fully connected layer is the final layer, surprisingly these layers have more information than the rest of the above layers. Simply, because the rest of the layers help network to learn the filters which are important to extract information rather than task of decision making which is done by the final layer.
torch.nn.Linear(input,output)
torch.nn.Linear(4096,1024)
torch.nn.Linear(1024,10), here 10 is number of classes.
** If classes are very less then don't directly jumpy from 4096 or 2048 to number of classes.

iv. Wrapping

Before running your model you need something which can hold your model together while backpropagation and forward propagation. Pytorch has

1. first_out_put = torch.nn.Sequential(

```
----Name---of----layers----
torch.nn.Conv2d()
torch.nn.Conv2d()
torch.nn.Relu()
torch.nn.AvgPool2d()
)
```

Layers in Sequential cannot be accessed individually, or cannot be frozen individually. Deciding which layers go in sequential is also important. Sometime one may prefer to write their whole code has module rather than wrapping in Sequential.

2. Output1 = torch.nn.Conv2d()
Output2 = torch.nn.Conv2d()
Output3 = torch.nn.Relu()
Output4 = torch.nn.AvgPool2d()

Here all layers are accessible individually, but it is hectic to write everything individually.

Using mix of nn.Sequential is good way to write pytorch code.

Example Toy VGG code in pytorch

```
Class
VGG(nn.Module):

    def __init__(self, features, num_classes=1000,
    init_weights=True):
        super(VGG, self).__init__()
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
    )
        if init_weights:
            self._initialize_weights()
```

```
def forward(self, x):
    x = self.features(x)
    x = self.avgpool(x)
    x = x.view(x.size(0), -1) #resize
    x = self.classifier(x)
    return x
```

v. Loading Models

Let's say you have the model which is ready to run. Two things are very important while running model. Detailed article can be read here Saving and Loading Models

1. Saving

`torch.save(VGG.state_dict(), PATH)` will save the parameters learned by your model. If you don't provide the above line then you won't be able to retrieve the model again without training, it is also important if one wish to resume one's model later.

2. Loading

```
model = VGG()
model.load_state_dict(torch.load(PATH))
model.eval() # Only use when testing
```

```
To load the model, one need to define the model object, which will
load the weights learned during training.
```

vi. Data Loader

There are actually many ways to load data in pytorch. We will divide the methods is 2 section, one which is directly provided by pytorch and another is custom method. More examples of pre-defined datasets in pytorch is here.

1. Pytorch Method

a. ImageFolder

```
A generic data loader where the images are arranged in this way:
root/dog/xxx.png
root/dog/xxy.png
root/dog/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/asd932_.png

Then using ImageFolder is good idea.

def ImageLoader():
    def loader(transform):
        data = torchvision.datasets.ImageFolder(PATH, transform=transform)
        data_loader = torchvision.utils.data.DataLoader(data, shuffle=True,
batch_size=batch_size,
                                 num_workers=4)

        return data_loader

    return loader


def ImageLoader_Transform(dataloader, image_size=4):
    transform = transforms.Compose([
        transforms.Resize(image_size),
        transforms.CenterCrop(image_size),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    loader = dataloader(transform)

    return loader

Loader = ImageLoader()
Dataloader = ImageLoader_Transform(Loader,256)

for x, y in Dataloader:
    print(x) # image
    print(y) # image label
```

2. Custom Method

Custom Method is very important, when your data doesn't follows the above structure. Like exporting image and it's mask which needs to be synchronized. You cannot have different image and different mask. Complete example for exporting mask and corresponding image code is provided below.

```
class DataCV(data.Dataset):
    """Dataset for Custom folders.
    """
    def __init__(self):
        super(DataCV, self).__init__()
        # base setting

    def name(self):
        return "DataCV"

    def __getitem__(self, index):
        return image


    def __len__(self):
        return len(self.im_names)
```

Example:

```
/data/train/imagesA/
/data/train/imagesMask/
Here, we have two folders one with image and another with mask of the image.
Such data needs a text file to synchronize the data.

“
train.txt
000003_0.jpg 000003_1.jpg
000004_0.jpg 000004_1.jpg
000005_0.jpg 000005_1.jpg
000006_0.jpg 000006_1.jpg
000007_0.jpg 000007_1.jpg
000008_0.jpg 000008_1.jpg
000009_0.jpg 000009_1.jpg
000011_0.jpg 000011_1.jpg

”
class CMPE(torch.utils.data.Dataset):
    """
Dataset for CMPE-685.
    """
    def __init__(self):
        super(CMPE, self).__init__()

        # base setting

        self.root = '/home/np9207/nile/data'
        self.datamode = 'train' # train or test or self-define
        self.data_list = "train.txt"
        self.fine_height = 256
        self.fine_width = 256
        self.radius = 3
        self.data_path = osp.join(self.root, self.datamode)
## Transform
        self.transform = transforms.Compose([
transforms.Resize((self.fine_height,self.fine_width)),\
                transforms.ToTensor(),   \
                transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

        # load data list
        srcs = []
        masks = []
        with open(osp.join(self.root, self.data_list), 'r') as f:
            for line in f.readlines():
                src, mask = line.strip().split()
                srcs.append(src)
                masks.append(mask)

        self.srcs = srcs
        self.masks = mask

    def name(self):
        return "CMPE"

    def __getitem__(self, index):
        src = self.srcs[index]
        masks = self.masks[index]

        source = Image.open(osp.join(self.data_path, 'image', src))
        masks = Image.open(osp.join(self.data_path, 'imagesMask', masks))


        source = self.transform(source)  # [-1,1]
        masks = self.transform(masks)  # [-1,1]

        dataset = {"src":source,"mask":masks}
        return dataset

    def __len__(self):
        return len(self.srcs)
```

3. Data Sampler

Data sampling for training and testing is important, especially when we have a dataset without train and test set. The following example illustrates how to implement data sampling in pytorch.

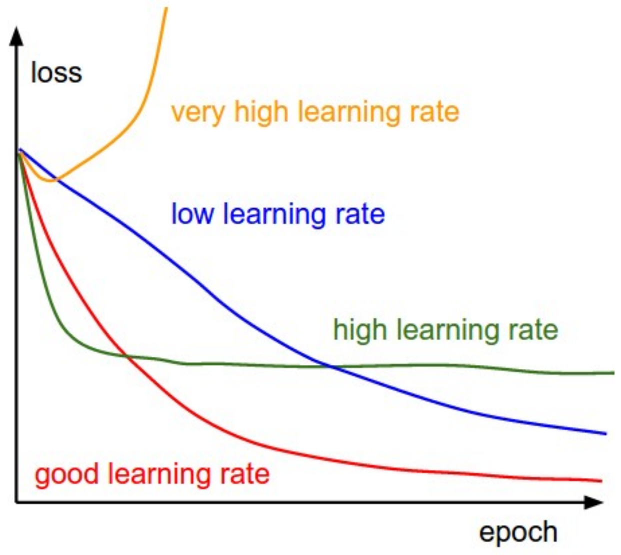
```
data = ImageFolder(root="/home/np9207/cmpe/lo/Train/", transform=transform())
train_size = int(0.9 * len(data)) # 0.9 is ratio for training
test_size = len(data) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(data, [train_size,
test_size])
```

vii. Hyperparameter tuning

Hyperparameter are the variable that can be changed in the model. Hyperparameter tuning itself is big research area.

1. Learning Rate

Learning Rate is very vague. It is the hyperparameter which controls the rate at which deep neural network learn. If it is set high, it will overshoot, if it is less then it might not even learn. Definition of slow or fast learning rate is purely experimental. A good article on learning rate with more visual explanation.



a. Observe Loss value, if it is decreasing or increasing constantly then learning rate is good.

b. If there is wavy curve in loss, than you might need to decrease the learning rate.
c. In my experience 0.0001 and 0.0002 can have very different effect on the model output. Other than experimenting with different learning rate. Researchers have proposed methods like learning rate decay.
d. Learning rate is decayed over the iteration, where the learning rate is slowly changed.

```
epoch = 0
n_epochs = 200
decay_epoch = 100
lr_scheduler_G = torch.optim.lr_scheduler.LambdaLR(optimizer_G,
lr_lambda=LambdaLR(n_epochs, epoch, decay_epoch).step)

*** end of the epoch ***
lr_scheduler_G.step()
```

2. Optimizers

Optimizers adjust the weights in every layer according to the difference in loss. Adjusting the weights in each layer basically means learning weights. There many different optimizers in deep neural networks.

1. Adam

There is great article on Adam. Adam is most used optimizer in deep learning, and it adjusts the momentum and learning dynamically.

```
torch.optim.Adam(VGG.parameters(),lr=0.01)
```

3. Epochs

Epochs define how long one wish to train their model. Longer training in some networks makes the result stable, where as there are network which explode from longer training. Depending upon the loss curve, one can make a smart guess, else it is experimental on how long one should keep their model running.

viii. Loss

Loss functions decides how good is your predicted result. Loss value is then back propagated to every layer according to which optimizers adjust the weight of the layers. Some of the very used loss functions are given below. Mentioned loss functions are used almost 99% of the time in any research.

1. Binary cross entropy Loss / Cross entropy Loss

If you might have studied logistic regression, then the name might have sound familiar. The loss is borrowed from the logistic regression literature.

```
torch.nn.BCELoss()
```

2. L1 Loss / L2 Loss

There is a great article on Least absolute error (L1) and Least square error (L2).

```
L2_Loss = torch.nn.MSELoss()
L1_Loss = torch.nn.L1Loss()
```

References:

https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1
https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f
https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c
https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/
https://pytorch.org/tutorials/beginner/saving_loading_models.html
https://pytorch.org/docs/stable/torchvision/datasets.html
https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10
https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1
https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c
https://towardsdatascience.com/demystifying-cross-entropy-e80e3ad54a8
http://rishy.github.io/ml/2015/07/28/l1-vs-l2-loss/