CMPE 685 Computer Vision

Pytorch Tutorial

Nilesh Pandey

Abu M. N. Taufique

Index

1. Numpy ↔ Pytorch
2. Basic Syntax
3. Neural Network
4. Loss and Optimization
5. Training
6. Hyperparameter : Learning Rate | Epoch
7. Hyperparameter : Suggestions

Numpy ↔ Pytorch

Pytorch is inspired by numpy operations, which can be performed on GPU.

Basic Numpy ↔ Pytorch Operation.

```
1  x = np.random.randn(5,3)
```

```
1  X_1 = torch.from_numpy(x)
```

```
1  X
```

```
tensor([[-0.8175, -0.4168,  0.9458],
        [ 0.9310,  0.1324, -0.1400],
        [ 1.4400, -1.1683, -1.7375],
        [-0.1767, -0.3250,  0.3472],
        [ 0.2042,  0.8053, -0.4884]], dtype=torch.float64)
```

```
1  Z = torch.Tensor(x)
```

```
1  Z
```

```
tensor([[-0.8175, -0.4168,  0.9458],
        [ 0.9310,  0.1324, -0.1400],
        [ 1.4400, -1.1683, -1.7375],
        [-0.1767, -0.3250,  0.3472],
        [ 0.2042,  0.8053, -0.4884]])
```

```
1  Z = Z.type(torch.float16)
```

```
1  Z
```

```
tensor([[-0.8174, -0.4167,  0.9458],
        [ 0.9312,  0.1324, -0.1400],
        [ 1.4404, -1.1680, -1.7373],
        [-0.1768, -0.3250,  0.3472],
        [ 0.2041,  0.8052, -0.4883]], dtype=torch.float16)
```

Torch.from_numpy() maintains the datatype which is important

Syntax

Basic Syntax of NN class

- Right hand code is simplest NN without any Output or any Operation.
- All code will follow the same pattern irrespective of version.

- Pytorch closely follows Python, so pythonic classes can be used

And functions like general python functions.

- The 2nd example is simple neural net

With 2 layers.

Example:

nn.Conv2d
nn.Batchnorm2d
nn.Relu

```
class VGGBlock(nn.Module):
    def __init__(self,values=***, values=***):
        super(VGGBlock, self).__init__()
#         *** Bunch of Modules and Functions ****

    def forward(self, x):
#         *** operations on the Input ***
        return out
```

```
class VGGBlock(nn.Module):

    def __init__(self, act_func=nn.ReLU(inplace=True)):

        super(VGGBlock, self).__init__()

        self.conv1 = nn.Conv2d(in_channels, middle_channels, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(middle_channels)
        self.conv2 = nn.Conv2d(middle_channels, out_channels, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.act = act_func

    def forward(self, x):

        out = self.conv1(x)
        out = self.bn1(out)

        out = self.conv2(out)
        out = self.bn2(out)

        out = self.act(out)

        return out
```

VGG – 7(example)

```python
import torch.nn as nn
from torchsummary import summary

class VGG(nn.Module):
    def __init__(self, num_classes=1000, init_weights=True):
        super(VGG, self).__init__()

        self.features = nn.Sequential(
                nn.Conv2d(3, 64, kernel_size=3, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2),

                nn.Conv2d(64, 128, kernel_size=3, padding=1),
                nn.BatchNorm2d(128),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2),

                nn.Conv2d(128, 256, kernel_size=3, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),
                nn.Conv2d(256, 256, kernel_size=3, padding=1),
                nn.BatchNorm2d(256),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(kernel_size=2, stride=2),


            )

        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))

        self.classifier = nn.Sequential(
            nn.Linear(256 * 7 * 7, 1024),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(1024, 1024),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(1024, 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)
        x = self.classifier(x)
        return x
```

```python
model = VGG().cuda()
```

```python
summary(model,(3,224,224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
       BatchNorm2d-2         [-1, 64, 224, 224]             128
              ReLU-3         [-1, 64, 224, 224]               0
         MaxPool2d-4         [-1, 64, 112, 112]               0
            Conv2d-5        [-1, 128, 112, 112]          73,856
       BatchNorm2d-6        [-1, 128, 112, 112]             256
              ReLU-7        [-1, 128, 112, 112]               0
         MaxPool2d-8          [-1, 128, 56, 56]               0
            Conv2d-9          [-1, 256, 56, 56]         295,168
      BatchNorm2d-10          [-1, 256, 56, 56]             512
             ReLU-11          [-1, 256, 56, 56]               0
           Conv2d-12          [-1, 256, 56, 56]         590,080
      BatchNorm2d-13          [-1, 256, 56, 56]             512
             ReLU-14          [-1, 256, 56, 56]               0
        MaxPool2d-15          [-1, 256, 28, 28]               0
AdaptiveAvgPool2d-16            [-1, 256, 7, 7]               0
           Linear-17                 [-1, 1024]      12,846,080
             ReLU-18                 [-1, 1024]               0
          Dropout-19                 [-1, 1024]               0
           Linear-20                 [-1, 1024]       1,049,600
             ReLU-21                 [-1, 1024]               0
          Dropout-22                 [-1, 1024]               0
           Linear-23                 [-1, 1000]       1,025,000
================================================================
Total params: 15,882,984
Trainable params: 15,882,984
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 157.87
Params size (MB): 60.59
```

Loss and Optimization

```
optimizer = torch.optim.Adam(filter(lambda p: p.requires_grad, model.parameters()), lr=3e-4)

'''
filter(lambda p: p.requires_grad, model.parameters())

Only layers which are not frozen can be only passed to optimizer. Frozen layers are not passed to optimizer.


optimizer.zero()

optimizer accumulates loss over batch when backpropogating the loss, so it is important to empty the
optimizer.


optimizer.step()

optimizer.step() performs the update on each layer, this step is performed only if we have loss present.




'''
```

```
criterion = nn.BCEWithLogitsLoss().cuda()

'''

Loss is the difference between the ground truth and the predicted value. Loss needs to be backpropagated,
and the optimizer updates the values in each layer.

Loss and optimizer work together.

'''
```

Training

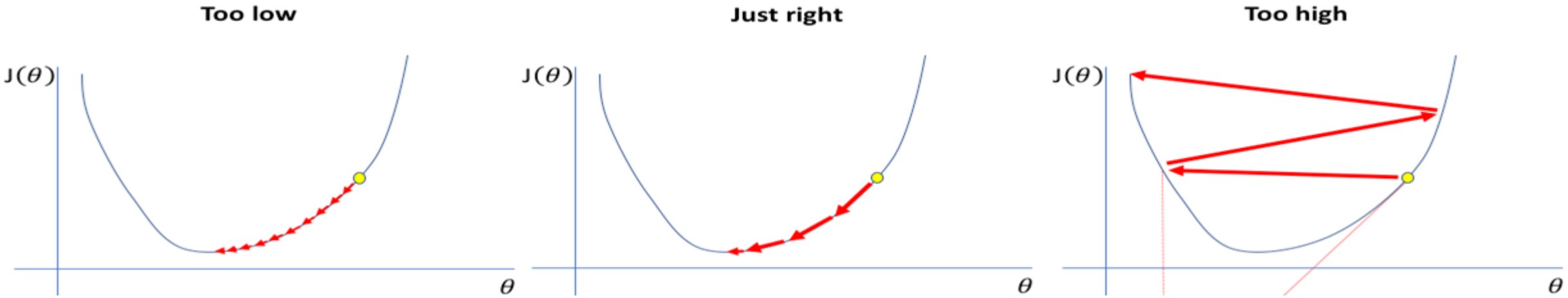
Training can be done as simple as the example.

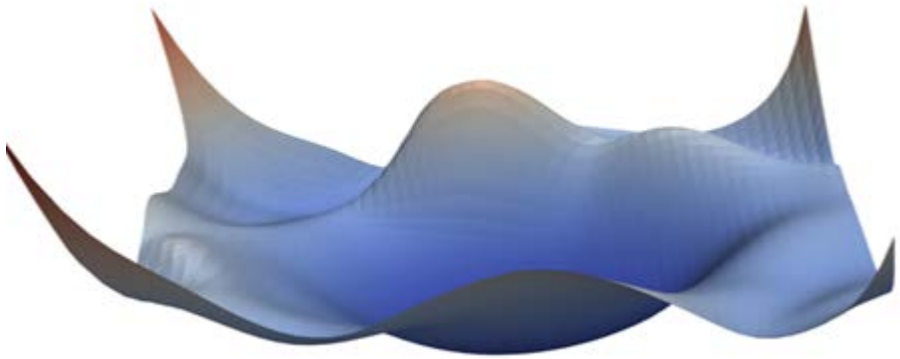
The steps in Training include.

- Loop in data
- Decide GPU/CPU
- Initiate model.train()
- Initiate optimizer to zero
- Calculate Loss
- Backprop Loss
- Update the layers
- Save the model

```
model.train()

for i, (input, target) in tqdm(enumerate(train_loader), total=len(train_loader)):
    input = input.cuda()

    target = target.cuda()

    output = model(input)

    loss = criterion(output, target)


    optimizer.zero_grad()

    loss.backward()

    optimizer.step()

    if i%500==0:
        fig, ax = plt.subplots(1,2, figsize = (8,4))
        ax[0].imshow(target[0,0,:,:].detach().cpu(),cmap="gray")
        ax[1].imshow(output[0,0,:,:].detach().cpu(),cmap="gray")
        plt.show()
```

Hyper parameters : Learning Rate



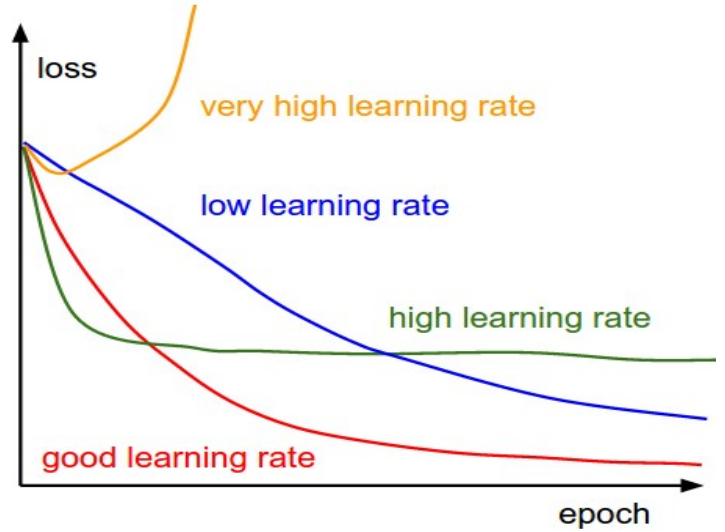


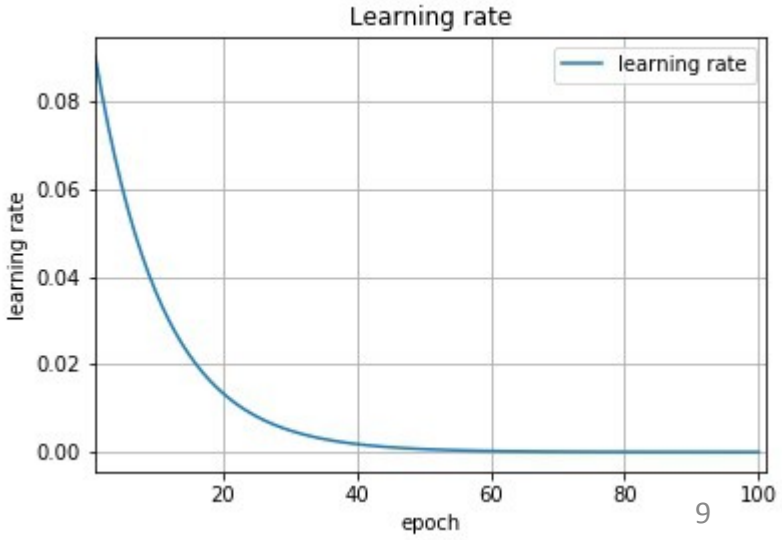
Hyper parameters : Learning Rate

- Learning rate

1. LR based on Loss
    1. Loss ↑ / ↓
    2. Convergence Should not be very fast

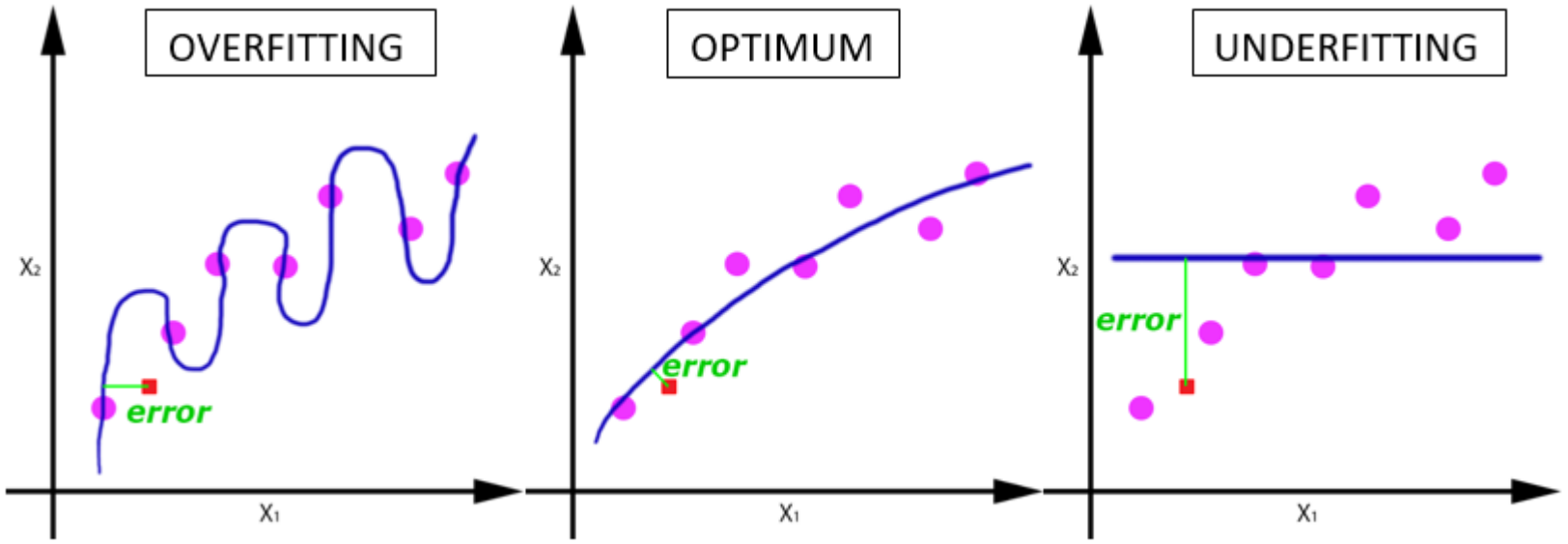
2. LR Scheduling
    1. Learning Rate decay
    2. Optimizer



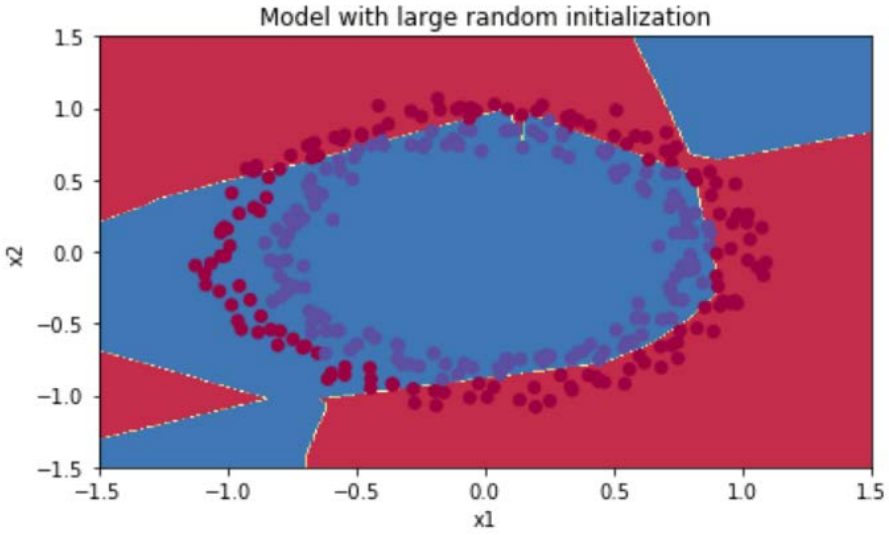


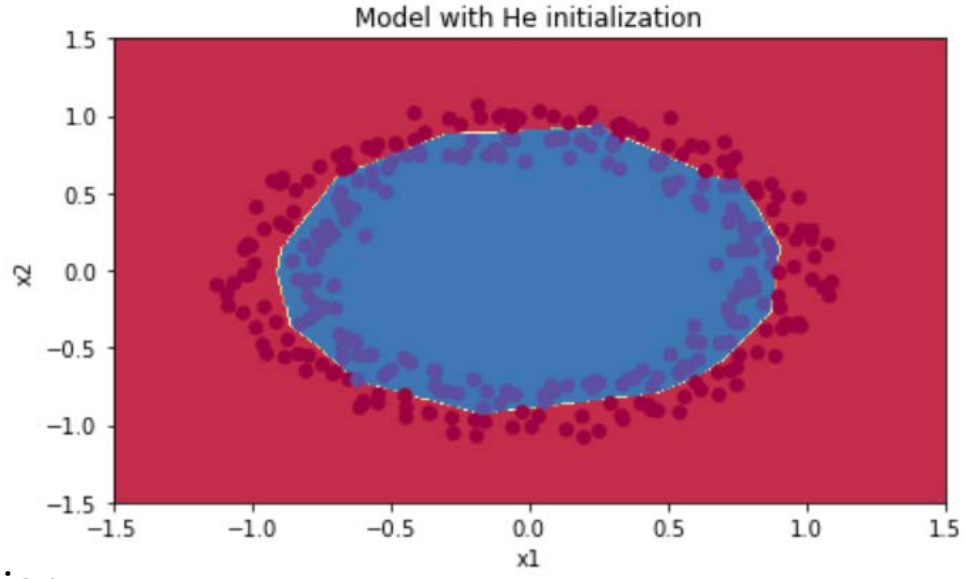
Hyper parameters : Epochs

- Higher Number of Epochs
  - First Image
- Lower Number of Epochs
  - Last Image
- Optimal Number of Epochs
  - Middle Image



Hyper parameter: Weight Initialization





Result after training network with two different weight initialization.

- Initialization is a very important hyper parameter to get better optimization.

```
''''
torch.nn.init.uniform_(tensor, a=0, b=1)
torch.nn.init.normal_(tensor, mean=0, std=1)
torch.nn.init.kaiming_normal_()
torch.nn.init.xavier_normal_()
''''
def weights_init(m):
    if isinstance(m, nn.Conv2d):
        torch.nn.init.kaiming_normal_(m.weight.data)
        torch.nn.init.kaiming_normal_(m.bias.data)

model.apply(weights_init)
```

Useful 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/