1. model.train()和model.eval()用法和区别

1.1 model.train()

model.train()的作用是**启用 Batch Normalization 和 Dropout。**

如果模型中有BN层(Batch Normalization）和Dropout，需要在训练时添加model.train()。model.train()是保证BN层能够用到每一批数据的均值和方差。对于Dropout，model.train()是随机取一部分网络连接来训练更新参数。

1.2 model.eval()

model.eval()的作用是**不启用 Batch Normalization 和 Dropout。**

如果模型中有BN层(Batch Normalization）和Dropout，在测试时添加model.eval()。model.eval()是保证BN层能够用全部训练数据的均值和方差，即测试过程中要保证BN层的均值和方差不变。对于Dropout，model.eval()是利用到了所有网络连接，即不进行随机舍弃神经元。

训练完train样本后，生成的模型model要用来测试样本。在model(test)之前，需要加上model.eval()，否则的话，有输入数据，即使不训练，它也会改变权值。这是model中含有BN层和Dropout所带来的的性质。

在做one classification的时候，训练集和测试集的样本分布是不一样的，尤其需要注意这一点。

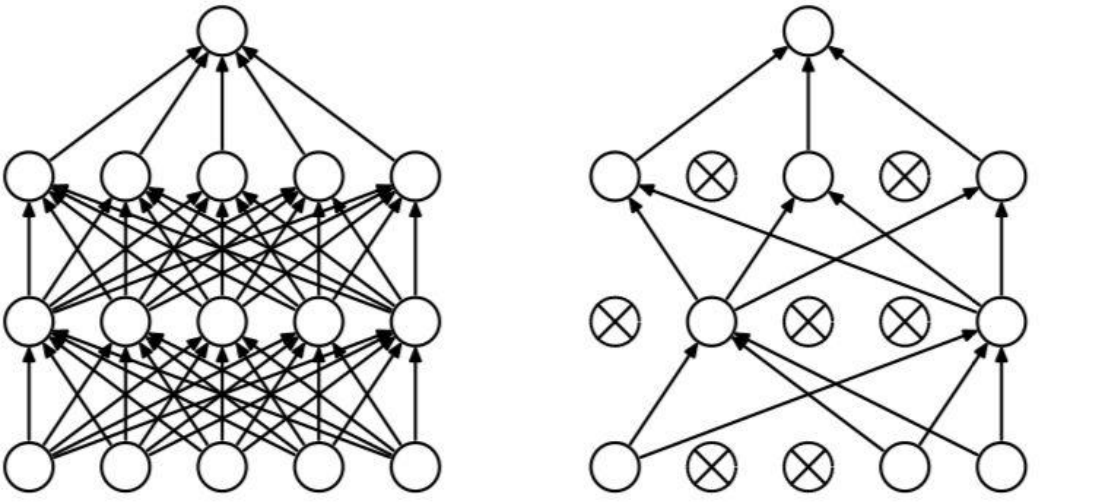
1.3 分析原因

使用PyTorch进行训练和测试时一定注意要把实例化的model指定train/eval。model.eval()时，框架会自动把BN和Dropout固定住，不会取平均，而是用训练好的值，不然的话，一旦test的batch_size过小，很容易就会被BN层导致生成图片颜色失真极大！！！！！！

```
# 定义一个网络
class Net(nn.Module):
    def __init__(self, l1=120, l2=84):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, l1)
```

```
        self.fc2 = nn.Linear(l1, l2)
        self.fc3 = nn.Linear(l2, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    # 实例化这个网络
    Model = Net()

    # 训练模式使用.train()
    Model.train(mode=True)

    # 测试模型使用.eval()
    Model.eval()
```

为什么PyTorch会关注我们是训练还是评估模型？最大的原因是dropout和BN层（以dropout为例）。这项技术在训练中随机去除神经元。



(a) Standard Neural Net　　(b) After applying dropout.

dropout

想象一下，如果右边被删除的神经元（叉号）是唯一促成正确结果的神经元。一旦我们移除了被删除的神经元，它就迫使其他神经元训练和学习如何在没有被删除神经元的情况下保持准确。这种dropout提高了最终测试的性能，但它对训练期间的性能产生了负面影响，因为网络是不全的。

下面我们看一个我们写代码的时候常遇见的错误写法：

在这个特定的例子中，似乎每50次迭代就会降低准确度。

如果我们检查一下代码， 我们看到确实在train函数中设置了训练模式。

```
def train(model, optimizer, epoch, train_loader, validation_loader):
    model.train() # ??????????? 错误的位置
    for batch_idx, (data, target) in experiment.batch_loop(iterable=train_loader):
        # model.train() # 正确的位置，保证每一个batch都能进入model.train()的模式
        data, target = Variable(data), Variable(target)
        # Inference
        output = model(data)
        loss_t = F.nll_loss(output, target)
        # The iconic grad-back-step trio
        optimizer.zero_grad()
        loss_t.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            train_loss = loss_t.item()
            train_accuracy = get_correct_count(output, target) * 100.0 / len(target)
            experiment.add_metric(LOSS_METRIC, train_loss)
            experiment.add_metric(ACC_METRIC, train_accuracy)
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx, len(train_loader),
                100. * batch_idx / len(train_loader), train_loss))
            with experiment.validation():
                val_loss, val_accuracy = test(model, validation_loader) # ???????????
                experiment.add_metric(LOSS_METRIC, val_loss)
```

```
24                  experiment.add_metric(ACC_METRIC,
    val_accuracy)
```

这个问题不太容易注意到，在循环中我们调用了test函数。

```
1  def test(model, test_loader):
2      model.eval()
3      # ...
```

在test函数内部，我们将模式设置为eval。这意味着，如果我们在训练过程中调用了test函数，我们就会进eval模式，直到下一次train函数被调用。这就导致了每一个epoch中只有一个batch使用了dropout ，这就导致了我们看到的性能下降。

修复很简单我们将model.train() 向下移动一行，让其在训练循环中。理想的模式设置是尽可能接近推理步骤，以避免忘记设置它。修正后，我们的训练过程看起来更合理，没有中间的峰值出现。

2. model.eval()和torch.no_grad()的区别

在PyTorch中进行validation/test时，会使用model.eval()切换到测试模式，在该模式下：

1. 主要用于通知dropout层和BN层在train和validation/test模式间切换：

- 在train模式下，dropout网络层会按照设定的参数p设置保留激活单元的概率（保留概率=p); BN层会继续计算数据的mean和var等参数并更新。
- 在eval模式下，dropout层会让所有的激活单元都通过，而BN层会停止计算和更新mean和var，直接使用在训练阶段已经学出的mean和var值。
- 该模式不会影响各层的gradient计算行为，即gradient计算和存储与training模式一样，只是不进行反向传播（back probagation)。

而with torch.no_grad()则主要是用于停止autograd模块的工作，以起到加速和节省显存的作用。它的作用是将该with语句包裹起来的部分停止梯度的更新，从而节省了GPU算力和显存，但是并不会影响dropout和BN层的行为。

如果不在意显存大小和计算时间的话，仅仅使用model.eval()已足够得到正确的validation/test的结果；而with torch.no_grad()则是更进一步加速和节省gpu空间（因为不用计算和存储梯度），从而可以更快计算，也可以跑更大的batch来测试。