Pytorch

pytorch安装

```
conda install numpy mkl cffi
conda install --offline pytorch-0.2.1-py36he6bf560_0.2.1cu80.tar.bz2 #离线安装
```

---

pytorch中的卷积

1. 卷积输出是向下取整，比如2.5->3

Pytorch 中的坑

1. torch.backends.cudnn.benchmark = True 在程序刚开始加这条语句可以提升一点训练速度，没什么额外开销。我一般都会加
2. 有时候可能是因为每次迭代都会引入点临时变量，会导致训练速度越来越慢，基本呈线性增长。开发人员还不清楚原因，但如果周期性的使用torch.cuda.empty_cache()的话就可以解决这个问题。这个命令是清除没用的临时变量的。
3. BCELoss()：pred和target都是float类型
4. DataLoader装载的数据集必须尺寸一致
5. DataLoader:batch must contain tensors ,numbers,dicts or lists

detach

detach

官方文档中，对这个方法是这么介绍的。

- 返回一个新的 从当前图中分离的 Variable。
- 返回的 Variable 永远不会需要梯度
- 如果 被 detach 的Variable volatile=True， 那么 detach 出来的 volatile 也为 True
- 还有一个注意事项，即：返回的 Variable 和 被 detach 的Variable 指向同一个 tensor

detach_

官网给的解释是：将 Variable 从创建它的 graph 中分离，把它作为叶子节点。

从源码中也可以看出这一点

- 将 Variable 的grad_fn 设置为 None，这样，BP 的时候，到这个 Variable 就找不到 它的 grad_fn，所以就不会再往后BP了。
- 将 requires_grad 设置为 False。这个感觉大可不必，但是既然源码中这么写了，如果有需要梯度的话可以再手动 将 requires_grad 设置为 true

能用来干啥

如果我们有两个网络，两个关系是这样的 现在我们想用 来为 网络的参数来求梯度，但是又不想求 网络参数的梯度。我们可以这样：

```
# y=A(x), z=B(y) 求B中参数的梯度，不求A中参数的梯度
# 第一种方法
y = A(x)
z = B(y.detach())
z.backward()

# 第二种方法
y = A(x)
y.detach_()
z = B(y)
z.backward()
```

Tensor

1. 需要注意 GPU 上的 Tensor 不能直接转换为 NumPy ndarray，需要使用.cpu()先将 GPU 上的 Tensor 转到 CPU 上
2. tensor的属性

```
size()
shape()
type()
dim()
numel()
```

3. tensor类型转换

```
x = torch.ones(4, 4).float()
dtype = torch.cuda.FloatTensor # 定义默认 GPU 的 数据类型
gpu_tensor = torch.randn(10, 20).type(dtype)
```

参数初始化

```
for layer in net1:
    if isinstance(layer, nn.Linear): # 判断是否是线性层
        param_shape = layer.weight.shape
```

```
        layer.weight.data = torch.from_numpy(np.random.normal(0,
0.5, size=param_shape))
        # 定义为均值为 0，方差为 0.5 的正态分布

# 定义一个 Tensor 直接对其进行替换
net1[0].weight.data = torch.from_numpy(np.random.uniform(3, 5, siz
e=(40, 30)))
```

Variable

1. Variable 计算, 梯度,如果用一个 Variable 进行计算, 那返回的也是一个同类型的 Variable.
2. requires_grad是参不参与误差反向传播, 要不要计算梯度: variable = Variable(tensor, requires_grad=True)
3. matplolib不能绘制tensor和variable类型的变量,需要转换成numpy
4. grad在反向传播过程中是累加(accumulated)，这意味着每一次运行反向传播，梯度都会累加之前的梯度，所以反向传播之前需把梯度清零
   x.grad.data.zero_()
5. 多次自动求导
   - PyTorch 默认做完一次自动求导之后，计算图就被丢弃了，所以两次自动求导需要手动设置一个东西

```
y.backward(retain_graph=True) # 设置 retain_graph 为 True 来保留计算图
y.backward() # 再做一次自动求导，这次不保留计算图
```

---

train()和eval()

1. model.train() model.eval()一般在模型训练和评价的时候会加上这两句，主要是针对model 在训练时和评价时不同的 Batch Normalization 和 Dropout 方法模式
2. 训练和测试的 dropout 层会不一样, 训练的时候随机丢掉了一些数据, 不过测试的时候整个神经网络都是活的, 没有被 drop 掉任何连接. 所以需要设置 model.eval().
3. model.train() # 把module设成训练模式，对Dropout和BatchNorm有影响,model.eval() # 把module设置为预测模式，对Dropout和BatchNorm模块有影响

Tensor和Variable

1. PyTorch Tensors and Variables have the same API

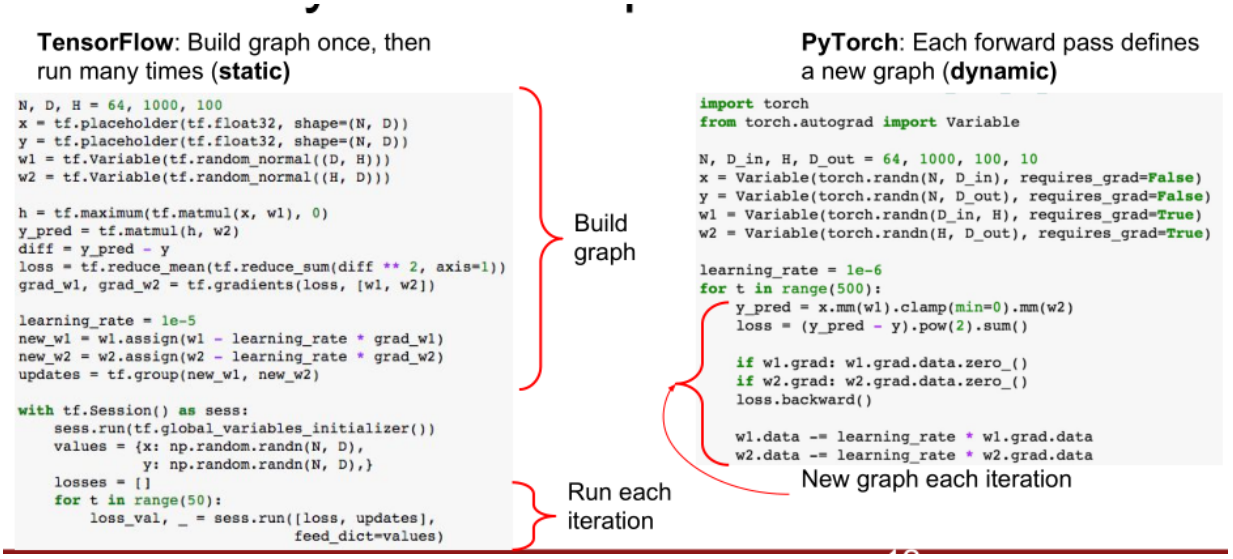
2. optimizer.step() # Update all parameters
   after computing gradients

使用预训练的模型

```
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)
```

Static vs Dynamic Graphs



优化

1. optimizer.zero_grad() # 梯度清零，等价于net.zero_grad()
2. #为不同子网络设置不同的学习率，在finetune中经常用到， 如果对某个参数不指定学习率，就使用最外层的默认学习率

```
# 为不同子网络设置不同的学习率，在finetune中经常用到
# 如果对某个参数不指定学习率，就使用最外层的默认学习率
optimizer =optim.SGD([
                {'params': net.features.parameters()}, # 学习率为1e-5
                {'params': net.classifier.parameters(), 'lr': 1e-2}
            ], lr=1e-5)
```

```
# 只为两个全连接层设置较大的学习率，其余层的学习率较小
special_layers = nn.ModuleList([net.classifier[0], net.classifier[3]])
special_layers_params = list(map(id, special_layers.parameters()))
base_params = filter(lambda p: id(p) not in special_layers_params,
                     net.parameters())

optimizer = t.optim.SGD([
            {'params': base_params},
            {'params': special_layers.parameters(), 'lr': 0.01}
        ], lr=0.001 )
```

nn.module和nn.Functional

此时读者可能会问，应该什么时候使用nn.Module，什么时候使用nn.functional呢？答案很简单，如果模型有可学习的参数，最好用nn.Module，否则既可以使用nn.functional也可以使用nn.Module，二者在性能上没有太大差异，具体的使用取决于个人的喜好。如激活函数（ReLU、sigmoid、tanh），池化（MaxPool）等层由于没有可学习参数，则可以使用对应的functional函数代替，而对于卷积、全连接等具有可学习参数的网络建议使用nn.Module。下面举例说明，如何在模型中搭配使用nn.Module和nn.functional。另外虽然dropout操作也没有可学习操作，但建议还是使用nn.Dropout而不是nn.functional.dropout，因为dropout在训练和测试两个阶段的行为有所差别，使用nn.Module对象能够通过model.eval操作加以区分。

对于不具备可学习参数的层（激活层、池化层等），将它们用函数代替，这样则可以不用放置在构造函数__init__中。对于有可学习参数的模块，也可以用functional来代替，只不过实现起来较为繁琐，需要手动定义参数parameter，如前面实现自定义的全连接层，就可将weight和bias两个参数单独拿出来，在构造函数中初始化为parameter。

```
class MyLinear(nn.Module):
    def __init__(self):
        super(MyLinear, self).__init__()
        self.weight = nn.Parameter(t.randn(3, 4))
        self.bias = nn.Parameter(t.zeros(3))
    def forward(self):
        return F.linear(input, weight, bias)
```

```
input = V(t.arange(0, 12).view(3, 4))
model = nn.Dropout()
# 在训练阶段，会有一半左右的数被随机置为0
model(input)
```

```
Variable containing:
  0   2   0   0
  8   0  12  14
 16   0   0  22
[torch.FloatTensor of size 3x4]
```

```
model.training  = False
# 在测试阶段，dropout什么都不做
model(input)
```

```
Variable containing:
  0   1   2   3
  4   5   6   7
  8   9  10  11
[torch.FloatTensor of size 3x4]
```

对于batchnorm、dropout、instancenorm等在训练和测试阶段行为差距巨大的层，如果在测试时不将其training值设为True，则可能会有很大影响，这在实际使用中要千万注意。虽然可通过直接设置training属性，来将子module设为train和eval模式，但这种方式较为繁琐，因如果一个模型具有多个dropout层，就需要为每个dropout层指定training属性。更为推荐的做法是调用model.train()函数，它会将当前module及其子module中的所有training属性都设为True，相应的，model.eval()函数会把training属性都设为False。

ModuleList

1. ModuleList也是一个特殊的module，可以包含几个子module，可以像用list一样使用它，但不能直接把输入传给ModuleList。ModuleList没有实现forward方法，ModuleList是Module的子类，当在Module中使用它的时候，就能自动识别为子module。list中的子module并不能被主module所识别，而ModuleList中的子module能够被主module所识别。这意味着如果用list保存子module，将无法调整其参数，因其未加入到主module的参数中。
2. Module能够自动检测到自己的Parameter，并将其作为学习参数。除了parameter之外，Module还包含子Module，主Module能够递归查找子Module中的parameter
3. 如果用list保存子module，将无法调整其参数，因其未加入到主module的参数中。

GPU训练

```
...

test_data = torchvision.datasets.MNIST(root='./mnist/', train=False)

# !!!!!!!! 修改 test data 形式 !!!!!!!! #
test_x = Variable(torch.unsqueeze(test_data.test_data, dim=1)).type(torch.FloatTensor)[:2000].cuda()/255.   # Tensor
test_y = test_data.test_labels[:2000].cuda()
```

```
class CNN(nn.Module):
    ...

cnn = CNN()

# !!!!!!!! 转换 cnn 去 CUDA !!!!!!!! #
cnn.cuda()    # Moves all model parameters and buffers to the GPU.
```

```
for epoch ...:
    for step, ...:
        # !!!!!!!! 这里有修改 !!!!!!!! #
        b_x = Variable(x).cuda()   # Tensor on GPU
        b_y = Variable(y).cuda()   # Tensor on GPU

        ...

        if step % 50 == 0:
            test_output = cnn(test_x)

            # !!!!!!!! 这里有修改 !!!!!!!! #
            pred_y = torch.max(test_output, 1)[1].cuda().data.squeeze()  # 将操作放去 GPU

            accuracy = torch.sum(pred_y == test_y) / test_y.size(0)
            ...

test_output = cnn(test_x[:10])

# !!!!!!!! 这里有修改 !!!!!!!! #
pred_y = torch.max(test_output, 1)[1].cuda().data.squeeze()  # 将操作放去 GPU
...
print(test_y[:10], 'real number')
```

转移至 CPU

如果你有些计算还是需要在 CPU 上进行的话呢, 比如 `plt` 的可视化, 我们需要将这些计算或者数据转移至 CPU.

```
cpu_data = gpu_data.cpu()
```

LSTM

首先需要定义好LSTM网络，需要nn.LSTM()，首先介绍一下这个函数里面的参数

input_size 表示的是输入的数据维数

hidden_size 表示的是输出维数

num_layers 表示堆叠几层的LSTM，默认是1

bias True 或者 False，决定是否使用bias

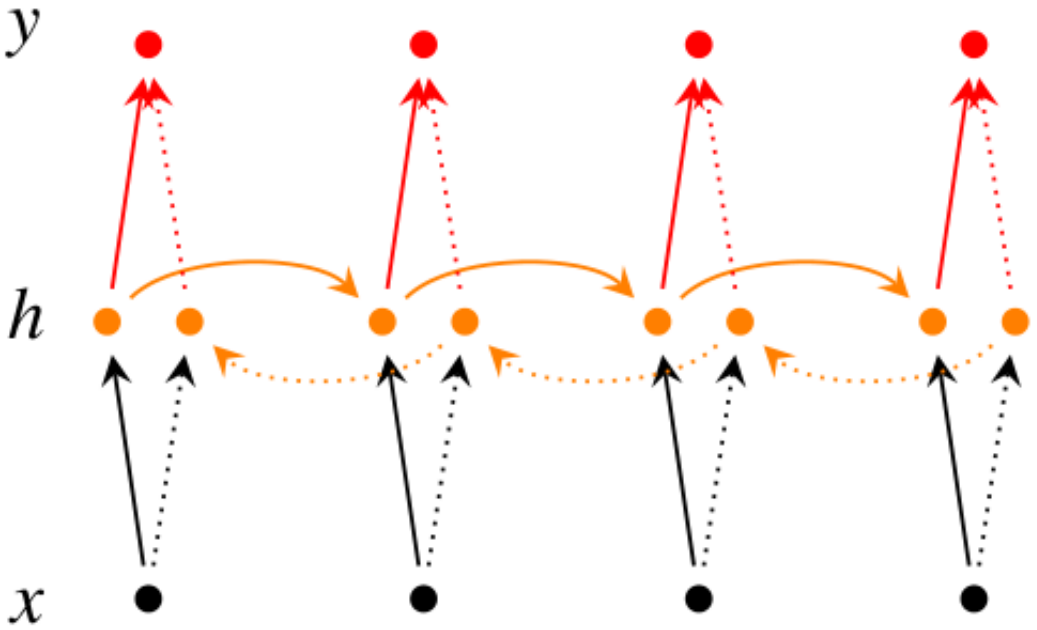
batch_first True 或者 False，因为nn.lstm()接受的数据输入是(序列长度，batch，输入维数)，这和我们cnn输入的方式不太一致，所以使用batch_first，我们可以将输入变成(batch，序列长度，输入维数)

dropout 表示除了最后一层之外都引入一个dropout

bidirectional 表示双向LSTM，也就是序列从左往右算一次，从右往左又算一次，这样就可以两倍的输出

是网络的输出维数，比如M，因为输出的维度是M，权重w的维数就是(M, M)和(M, K)，b的维数就是(M, 1)和(M, 1)，最后经过sigmoid激活函数，得到的f的维数是(M, 1)。

1. 双向RNN



$$从前往后：\overrightarrow{S_t^1} = f(\overrightarrow{U^1} * X_t + \overrightarrow{W^1} * S_{t-1} + \overrightarrow{b^1})$$

$$从后往前：\overrightarrow{S_t^2} = f(\overrightarrow{U^2} * X_t + \overrightarrow{W^2} * S_{t-1} + \overrightarrow{b^2})$$

$$输出：o_t = softmax(V * [\overrightarrow{S_t^1}; \overrightarrow{S_t^2}])$$

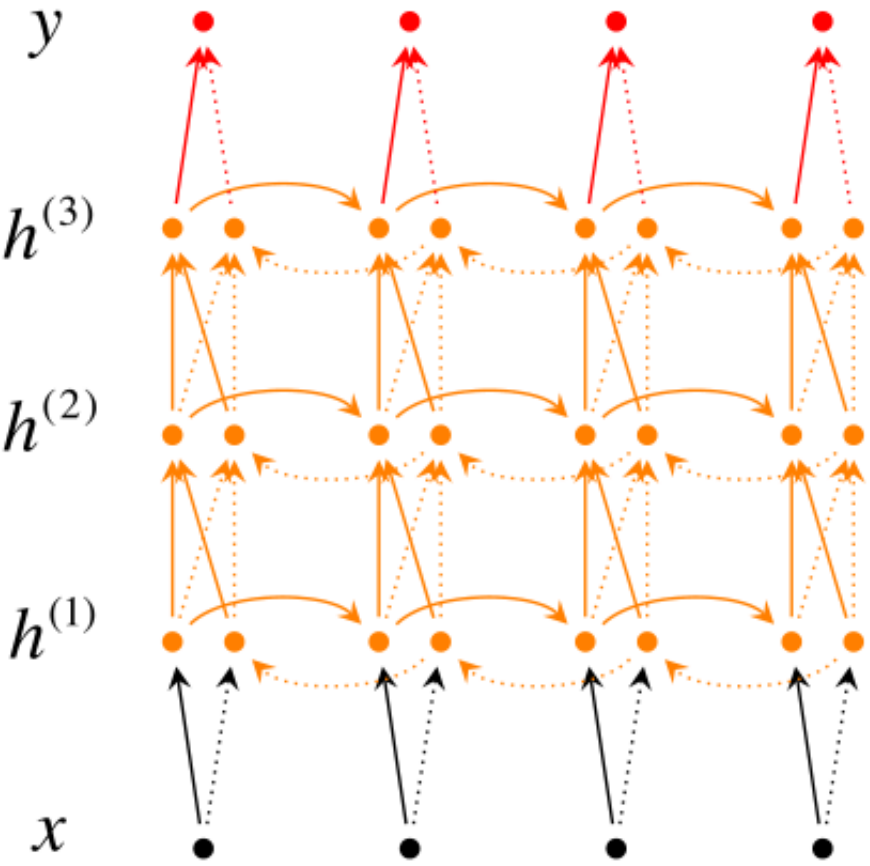
这里的$[\overrightarrow{S_t^1}; \overrightarrow{S_t^2}]$做的是一个拼接，如果他们都是1000X1维的，拼接在一起就是1000X2维的了。

双向RNN需要的内存是单向RNN的两倍，因为在同一时间点，双向RNN需要保存两个方向上的权重参数，在分类的时候，需要同时输入两个隐藏层输出的信息。

2. 深层双向RNN

深层双向RNN 与双向RNN相比，多了几个隐藏层，因为他的想法是很多信息记一次记不下来，比如你去考研，复习考研英语的时候，背英语单词一定不会就看一次就记住了所有要考的考研单词吧，你应该也是带着先前几次背过的单词，然后选择那些背过，但不熟的内容，或者没背过的单词来背吧。

深层双向RNN就是基于这么一个想法，他的输入有两方面，第一就是前一时刻的隐藏层传过来的信息$\overrightarrow{h}_{t-1}^{(i)}$，和当前时刻上一隐藏层传过来的信息$h_t^{(i-1)} = [\overrightarrow{h}_t^{(i-1)}; \overleftarrow{h}_t^{(i-1)}]$，包括前向和后向的。



我们用公式来表示是这样的:

$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)})$$

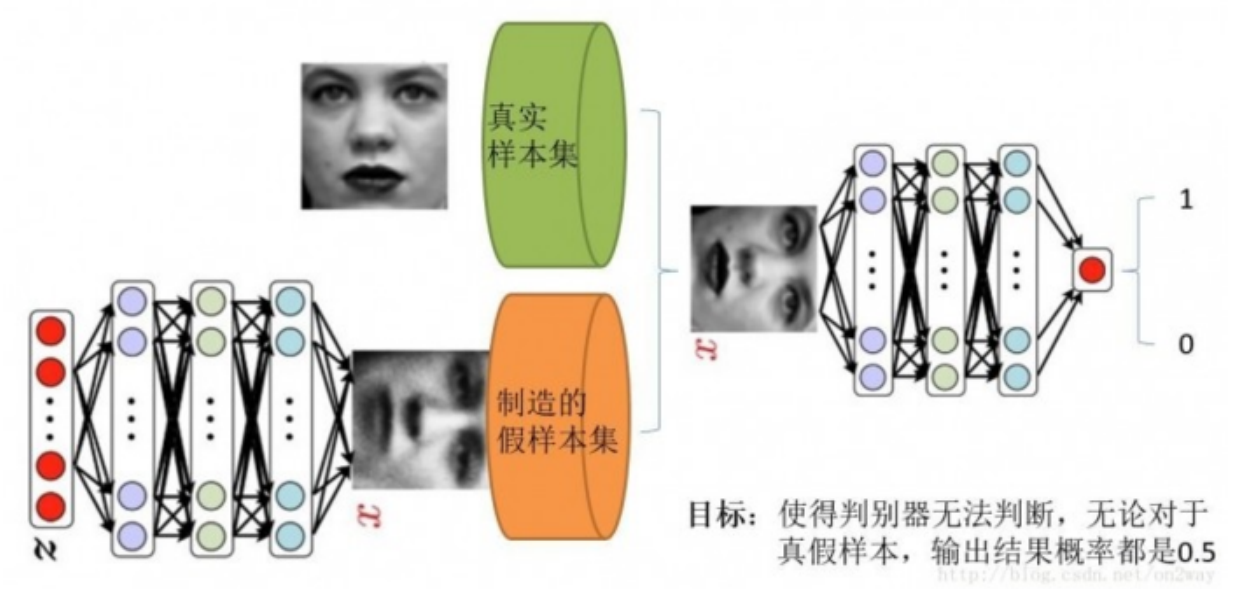
$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)})$$

然后再利用最后一层来进行分类，分类公式如下：

$$\hat{y}_t = g(Uh_t + c) = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c)$$

GAN网络的损失函数

1. 是一个二分类问题




```
d_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logit
s(logits=d_logits_real,labels=tf.ones_like(d_logits_real) * (1 - sm
ooth)))

d_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logit
s(logits=d_logits_fake,labels=tf.zeros_like(d_logits_real)))

d_loss = d_loss_real + d_loss_fake

g_loss = tf.reduce_mean(
             tf.nn.sigmoid_cross_entropy_with_logits(logits=d_logit
s_fake,
                                                 labels=tf.ones_like(d_log
its_fake))
```

2. log(D(x))+log(1-D(G(Z)))
   优化D:

$$\max_D V(D,G) = E_{x\sim p_{data}\,(x)}[log(D(x))] + E_{z\sim p_z\,(z)}[log(1-D(G(z)))]$$

优化G:

$$\min_G V(D,G) = E_{z\sim p_z\,(z)}[log(1-D(G(z)))]$$

类型转换

```
Variable(y.type(dtype).long())
```

正确率计算

```
 _, preds = scores.data.cpu().max(1)
        num_correct += (preds == y).sum()
```

重新初始化所有参数：

```
def reset(m):
    if hasattr(m, 'reset_parameters'):
        m.reset_parameters()

fixed_model.apply(reset)
```

自定义读取数据集

```
class ChunkSampler(sampler.Sampler):
    """Samples elements sequentially from some offset.
    Arguments:
        num_samples: # of desired datapoints
        start: offset where we should start selecting from
    """
    def __init__(self, num_samples, start = 0):
        self.num_samples = num_samples
        self.start = start

    def __iter__(self):
        return iter(range(self.start, self.start + self.num_sample
s))

    def __len__(self):
        return self.num_samples
```

```
NUM_TRAIN = 49000
NUM_VAL = 1000

cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, downl
oad=True,
                             transform=T.ToTensor())
loader_train = DataLoader(cifar10_train, batch_size=64, sampler=Chu
nkSampler(NUM_TRAIN, 0))

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, downloa
d=True,
                             transform=T.ToTensor())
loader_val = DataLoader(cifar10_val, batch_size=64, sampler=ChunkSa
mpler(NUM_VAL, NUM_TRAIN))

cifar10_test = dset.CIFAR10('./cs231n/datasets', train=False, downl
oad=True,
                            transform=T.ToTensor())
loader_test = DataLoader(cifar10_test, batch_size=64)
```