Chapter-5

December 22, 2018

0.1 Deep Learning in Computer Vision

Topics Covered:

1. Building Image Classifier on MNIST Dataset
2. Working of Conv in PyTorch
3. Why Pooling?
4. Why use View?
5. Transfer Learning
6. Visualizing Outputs from Intermediate Layers
7. Visualizing Weights of Intermediate Layers

```
In [1]: # Necessary Imports
        import torch, torchvision
        import time
        import numpy as np
        import matplotlib.pyplot as plt
        from torch.autograd import Variable
        import torchvision.transforms as transforms
        import torch.nn as nn
        import torch.optim as optim
        import torch.nn.functional as F
```

0.1.1 Challenges of using Linear Layers/FCN Layers?

- Loss of Spatial Information.
  - Flattening an image into 1D array loses all the spatial information about the image.
- High complexity (number of weights for all layers in FCN)

0.1.2 Building an Image Classifier on MNIST Dataset

Steps:

1. Getting the Data
2. Pre-process (flattening, resizing -- if required, split the dataset - train_test_split)
3. Build model (CNN)
4. Train and Validate the model.

5. Test on Unseen Data (Test Dataset)

Step-1: Getting the Data

```
help(torchvision.datasets)
```

Output:

```
Help on package torchvision.datasets in torchvision:

NAME
    torchvision.datasets

PACKAGE CONTENTS
    cifar
    coco
    fakedata
    folder
    lsun
    mnist
    omniglot
    phototour
    semeion
    stl10
    svhn
    utils
```

It's clear that we have mnist dataset available in torchvision datasets utility class. Let's go ahead and import mnist dataset.

```
In [5]: import torchvision.datasets.mnist as mnist
```

```
In [9]: # required transformations
        transformation = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.130
                                                                                          (0.308
```

```
In [11]: train_dataset = mnist.MNIST('data/', train=True, transform=transformation, download=Tr
         test_dataset = mnist.MNIST('data/', train=False, transform=transformation, download=Tr
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Processing...
Done!
```

```
In [14]: train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
         test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=True)
```

```
In [36]: # utility function to convert a tensor into an image
         def show_image(tensor, cmap=None):
             image = tensor.numpy()[0]
             mean = 0.1307
             std_dev = 0.3081
             image = ((mean * image) + std_dev)
             plt.imshow(image, cmap) # show gray-scaled version of the image
             plt.show()
```

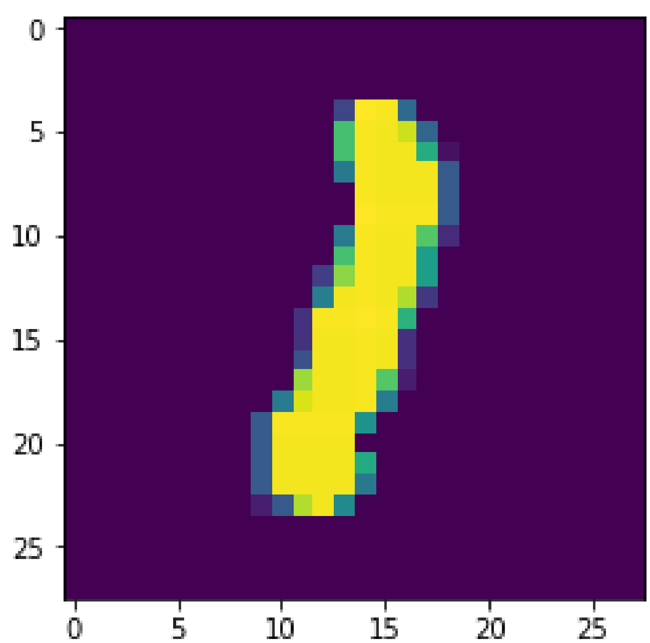
```
In [29]: next(iter(train_loader))[0] # returns batch of 32 images, randomly
```

```
Out[29]: tensor([[[[-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   ...,
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242]]],


                 [[[-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   ...,
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242]]],


                 [[[-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   ...,
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242]]],


                 ...,


                 [[[-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   ...,
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
                   [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242]]],
```

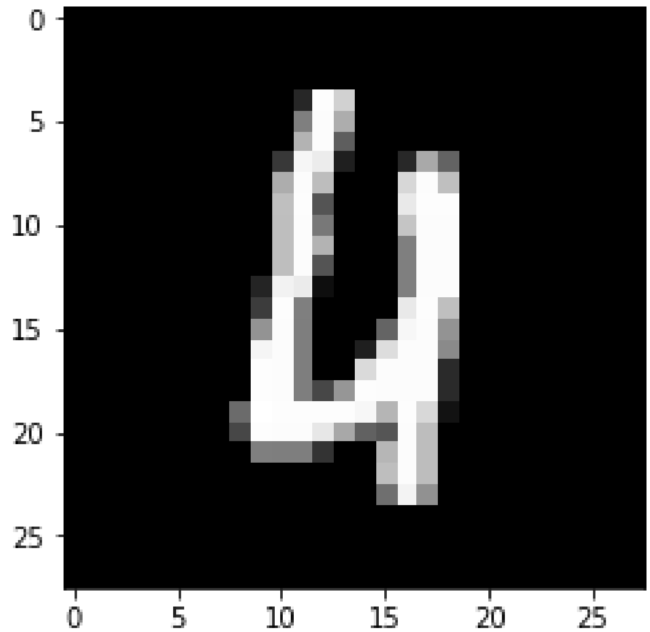
```
[[[-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  ...,
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242]]],


[[[-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  ...,
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242],
  [-0.4242, -0.4242, -0.4242,  ..., -0.4242, -0.4242, -0.4242]]]])
```

In [39]:
```
print("Showing image in color mode")
show_image(next(iter(train_loader))[0][0])
print("Showing image in grayscale mode")
show_image(next(iter(train_loader))[0][0], cmap='gray')
```

Showing image in color mode



```
Showing image in grayscale mode
```



0.2 Building Model

Generally, a CNN is composed of following layers:

1. Conv2d
2. MaxPooling2d
3. ReLU
4. View
5. Linear Layer (FC Layer)
6. Dropout

In [42]:
```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2d_dropout = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)
    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.max_pool2d(self.conv2d_dropout(self.conv2(x)), 2)
        x = x.view(-1, 320) # why?
```

```
            x = F.relu(self.fc1(x))
            x = F.dropout(x, training=self.training)
            x = self.fc2(x)
            return F.log_softmax(x)
```

0.3 How does Convolution works in PyTorch?

Take an example of a 1D array. Let's try to apply a conv filter on the array.

```
conv = nn.Conv1d(1, 1, 3, bias=False)
```

Let's look at the docs of `nn.Conv1d`:

```
class Conv1d(_ConvNd)
 |  Applies a 1D convolution over an input signal composed of several input
 |  planes.
 |
 |  In the simplest case, the output value of the layer with input size
 |  (N, C_{in}, L) and output (N, C_{out}, L_{out}) can be
 |  precisely described as:
```

$$\text{out}(N_i, C_{out_j}) = \text{bias}(C_{out_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{out_j}, k) \star \text{input}(N_i, k)$$

```
 |  Attributes:
 |      weight (Tensor): the learnable weights of the module of shape
 |          (out_channels, in_channels, kernel_size)
 |      bias (Tensor):   the learnable bias of the module of shape
 |          (out_channels)
 |
 |  Examples::
 |
 |      >>> m = nn.Conv1d(16, 33, 3, stride=2)
 |      >>> input = torch.randn(20, 16, 50)
 |      >>> output = m(input)
```

```
In [76]: net = nn.Conv1d(16, 33, 3, stride=2)
         print("Net.weight: ", net.weight[0][0])
         print("Net bias: ", net.bias[0][0])
         input_ = torch.randn(20, 16, 50)
         print("Shape of input tensor: {}".format(input_.shape))
         output = net(input_)
         print("Input", input_[0][0])
         print("output", output[0][0])
```

```
Net.weight:  tensor([ 0.0898, -0.0290, -0.0184], grad_fn=<SelectBackward>)
Net bias:  tensor(-0.0234, grad_fn=<AliasBackward>)
Shape of input tensor: torch.Size([20, 16, 50])
```

```
Input tensor([ 0.4412,  1.1871,  1.2538,  0.9038, -0.4481,  1.3991, -0.9111,  0.2678,
         1.1055,  1.9549, -1.1615, -1.0567,  0.5989,  0.1253,  0.7138,  1.2872,
         1.2061, -0.4276, -1.0316, -2.1270,  0.6886,  0.5826, -1.3476,  0.6016,
        -0.3757, -1.1891,  0.6825, -0.0293, -1.0270,  0.8466,  1.6183, -1.1577,
        -1.6786,  1.4142, -0.6939, -0.8775, -1.0439,  0.1629,  1.2870, -1.2488,
        -0.8653,  0.8719,  0.2489, -2.0293, -2.2702,  0.8735,  0.9727,  0.5769,
        -0.4099, -0.9260])
output tensor([-0.4045,  0.4693, -0.6951, -0.4687, -0.0146, -0.4754,  0.5161, -0.1440,
        -0.2339, -0.2184,  0.3562, -0.5512, -0.7846, -0.8485, -0.1358,  0.3537,
        -0.2544,  0.1960, -0.3285,  0.1912, -0.1416,  0.0757,  0.1543,  1.5208],
       grad_fn=<SelectBackward>)


/home/kushashwa/.local/lib/python3.6/site-packages/ipykernel_launcher.py:3: UserWarning: inval
  This is separate from the ipykernel package so we can avoid doing imports until
```

```
In [64]: net.weight[0][0]
```

```
Out[64]: tensor([-0.0673,  0.1119, -0.0590], grad_fn=<SelectBackward>)
```

```
In [79]: net.weight[0][0]
```

```
Out[79]: tensor([ 0.0898, -0.0290, -0.0184], grad_fn=<SelectBackward>)
```

```
In [91]: net.bias[0]
```

```
Out[91]: tensor(-0.0234, grad_fn=<SelectBackward>)
```

```
In [88]: print(input_[0][0][:3] * net.weight[0][0])
         print("input_, net.weight, net.bias: {}/{}/{}".format(input_[0][0][:3], net.weight[0]
```

```
tensor([ 0.0396, -0.0344, -0.0231], grad_fn=<ThMulBackward>)
input_, net.weight, net.bias: tensor([0.4412, 1.1871, 1.2538])/tensor([ 0.0898, -0.0290, -0.01


/home/kushashwa/.local/lib/python3.6/site-packages/ipykernel_launcher.py:2: UserWarning: inval
```

```
In [104]: np.dot(input_[0][0][:3].detach().numpy(), net.weight[0][0].detach().numpy()) + net.b
```

```
Out[104]: -0.041310795
```

```
In [98]: type(net.weight)
```

```
Out[98]: torch.nn.parameter.Parameter
```

0.4 Why Pooling?

Used just after a convolution layer, to reduce the data size to process. Also helps in reducing the size of feature maps. Also, forces algorithm to not focus on small changes in position. **(how?)**

0.5 Why View?

Generally, we use `torch.Tensor.view()` (https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view) function at the end of a network, because for FC (Fully Connected) or Linear layers, we need to flatten the data to 1D.

While flattening, it's important to make sure that two different images don't mix-up. That's why we input the first argument to `torch.Tensor.view()` as `-1`.

Let's look at the docs of this function.

```
In [166]: print(help(torch.Tensor.view))
```

```
Help on method_descriptor:

view(...)
    view(*args) -> Tensor

    Returns a new tensor with the same data as the :attr:`self` tensor but of a
    different size.

    The returned tensor shares the same data and must have the same number
    of elements, but may have a different size. For a tensor to be viewed, the new
    view size must be compatible with its original size and stride, i.e., each new
    view dimension must either be a subspace of an original dimension, or only span
    across original dimensions :math:`d, d+1, \dots, d+k` that satisfy the following
    contiguity-like condition that :math:`\forall i = 0, \dots, k-1`,

    .. math::

      stride[i] = stride[i+1] \times size[i+1]

    Otherwise, :func:`contiguous` needs to be called before the tensor can be
    viewed.

    Args:
        args (torch.Size or int...): the desired size

    Example::

        >>> x = torch.randn(4, 4)
        >>> x.size()
        torch.Size([4, 4])
        >>> y = x.view(16)
        >>> y.size()
        torch.Size([16])
        >>> z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
        >>> z.size()
        torch.Size([2, 8])
```

```
None
```

```
In [164]: torch.Tensor.view?

In [158]: x = np.array([[[[8], [9]], [[2], [0.3]]], [[[8], [9]], [[2], [0.3]]]])
          x = x.reshape((2, 1, 2, 2)) # reshape to (2, 1, 2, 2) - like a batch of 2 images of

In [159]: x.view() # view the array x

Out[159]: array([[[[8. , 9. ],
                   [2. , 0.3]]],


                 [[[8. , 9. ],
                   [2. , 0.3]]]])

In [160]: x_tensor = torch.from_numpy(x) # conversion of numpy array to a pytorch tensor
          # reference: https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#conve

In [161]: print(x_tensor.shape) # verifying

torch.Size([2, 1, 2, 2])


In [167]: x_tensor.view(-1, 4)
          # -1 means: don't touch the first dimension (only if the second dimension satisfies
          # if doesn't satisfy, then -1 is inferred from other dimensions

Out[167]: tensor([[8.0000, 9.0000, 2.0000, 0.3000],
                  [8.0000, 9.0000, 2.0000, 0.3000]], dtype=torch.float64)

In [171]: x_tensor.view(-1, 2) # example: this should return (4, 2) tensor

Out[171]: tensor([[8.0000, 9.0000],
                  [2.0000, 0.3000],
                  [8.0000, 9.0000],
                  [2.0000, 0.3000]], dtype=torch.float64)

In [174]: x_tensor.view(-1, 1) # example: this should return (8, 1) tensor

Out[174]: tensor([[8.0000],
                  [9.0000],
                  [2.0000],
                  [0.3000],
                  [8.0000],
                  [9.0000],
                  [2.0000],
                  [0.3000]], dtype=torch.float64)
```

0.6 Training the Model

In [197]:
```
def fit(epoch, optimizer, model, data_loader, volatile=False, phase='training'):
    if(phase == 'training'):
        model.train()
    if(phase == 'evaluation'):
        model.evaluate()
        volatile=True #why?

    running_loss = 0.0
    running_correct = 0

    for batch_idx, (data, target) in enumerate(data_loader):
#         print(data)
#         if data.is_cuda():
#             data, target = data.cuda(), target.cuda()
        data, target = Variable(data, volatile), Variable(target)
        if phase == 'training':
            optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        running_loss += F.nll_loss(output, target, size_average=False).data[0]
        preds = output.data.max(dim=1, keepdim=True)[1]
        running_correct += preds.eq(target.data.view_as(preds)).cpu().sum()
        if phase == 'training':
            loss.backward()
            optimizer.step()

    loss = running_loss/len(data_loader.dataset)
    accuracy = 100. * running_correct/len(data_loader.dataset)
    print("Loss: {}, accuracy: {}".format(loss, accuracy))
    return loss, accuracy
```

In [198]:
```
model = Net()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

In [199]:
```
optimizer
```

Out[199]:
```
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.01
    momentum: 0.5
    nesterov: False
    weight_decay: 0
)
```

In [200]:
```
train_losses, train_accuracy = [], []
val_losses, val_accuracy = [], []
```

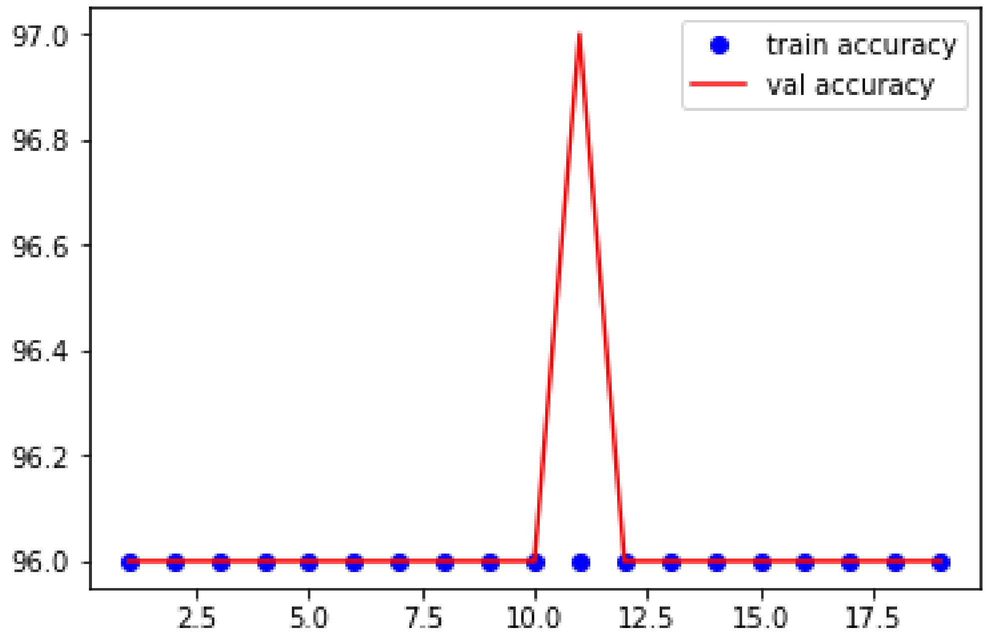
```
In [204]: for epoch in range(1, 20):
              epoch_loss, epoch_accuracy = fit(epoch, optimizer, model, train_loader, volatile=
              val_epoch_loss , val_epoch_accuracy = fit(epoch,optimizer, model, test_loader, vo
              train_losses.append(epoch_loss)
              train_accuracy.append(epoch_accuracy)
              val_losses.append(val_epoch_loss)
              val_accuracy.append(val_epoch_accuracy)
```

```
/home/kushashwa/.local/lib/python3.6/site-packages/ipykernel_launcher.py:16: UserWarning: Impli
  app.launch_new_instance()
/usr/local/lib/python3.6/dist-packages/torch/nn/functional.py:52: UserWarning: size_average and
  warnings.warn(warning.format(ret))
/home/kushashwa/.local/lib/python3.6/site-packages/ipykernel_launcher.py:20: UserWarning: inval


Loss: 0.1206219345331192, accuracy: 96
Loss: 0.12402940541505814, accuracy: 96
Loss: 0.11611782014369965, accuracy: 96
Loss: 0.11210104078054428, accuracy: 96
Loss: 0.11929728090763092, accuracy: 96
Loss: 0.1225995272397995, accuracy: 96
Loss: 0.11394266784191132, accuracy: 96
Loss: 0.11298033595085144, accuracy: 96
Loss: 0.11729313433170319, accuracy: 96
Loss: 0.11354488879442215, accuracy: 96
Loss: 0.115715891122818, accuracy: 96
Loss: 0.11816882342100143, accuracy: 96
Loss: 0.11266960203647614, accuracy: 96
Loss: 0.11292938143014908, accuracy: 96
Loss: 0.11479900777339935, accuracy: 96
Loss: 0.12624315917491913, accuracy: 96
Loss: 0.11431062966585159, accuracy: 96
Loss: 0.1180247887969017, accuracy: 96
Loss: 0.1105051189661026, accuracy: 96
Loss: 0.12324003875255585, accuracy: 96
Loss: 0.11057820171117783, accuracy: 96
Loss: 0.10865706950426102, accuracy: 97
Loss: 0.10824849456548691, accuracy: 96
Loss: 0.11275696009397507, accuracy: 96
Loss: 0.10923698544502258, accuracy: 96
Loss: 0.11321546882390976, accuracy: 96
Loss: 0.10587245225906372, accuracy: 96
Loss: 0.11644107848405838, accuracy: 96
Loss: 0.10572720319032669, accuracy: 96
Loss: 0.10592672973871231, accuracy: 96
Loss: 0.10624512284994125, accuracy: 96
Loss: 0.12218598276376724, accuracy: 96
Loss: 0.1034180149435997, accuracy: 96
```

```
Loss: 0.114189513027668, accuracy: 96
Loss: 0.10513007640838623, accuracy: 96
Loss: 0.11694205552339554, accuracy: 96
Loss: 0.10253959894180298, accuracy: 96
Loss: 0.10903823375701904, accuracy: 96
```

In [205]:
```
plt.plot(range(1,len(train_accuracy)+1),train_accuracy,'bo',label = 'train accuracy'
plt.plot(range(1,len(val_accuracy)+1),val_accuracy,'r',label = 'val accuracy')
plt.legend()
```

Out[205]: <matplotlib.legend.Legend at 0x7fa4d4396940>



0.7 Transfer Learning

- Reusing pre-trained algorithm on a dataset similar to what we are using. Training from scratch not required.
- Freeze most of the layers and fine tune params for some of the layers only.

In [3]:
```
from torchvision import models
```

In [208]:
```
vgg = models.vgg16(pretrained=True) # load the pretrained model VGG16
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /home/kushashwa/.torc
100%|| 553433881/553433881 [00:18<00:00, 30166213.80it/s]
```

```
In [210]: # let's look at the VGG network
          print(vgg)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

In the above summary of VGG16 Model, there are 2 Sequential models:

1. Features: - Has the layers that we are going to freeze **2. Classifier:** - Has the layers that we are going to learn.

In [211]:
```
# freeze the weights for features layers
for param in vgg.features.parameters():
    param.requites_grad = False
    # prevent optimizer from updating the weights
```

In [217]:
```
# let's fine tune the classifier layers
# for dogs and cats?
vgg.classifier[6].out_features = 2
```

In [218]:
```
optimizer = optim.SGD(vgg.classifier.parameters(), lr=0.001, momentum=0.5)
# only pass classifier parameters for optimization
```

Quick Tips to improve accuracy

In [229]:
```
# changing dropout from 0.2 to 0.5
for layer in vgg.classifier.children():
    if(type(layer) == nn.modules.dropout.Dropout):
        layer.p = 0.5
```

In [230]:
```
# performing data augmentation to add more data: flipping, mirroring, rotating with
train_transform = transforms.Compose([transforms.Resize((224, 224)),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation(0.2),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406], [0.
```

0.8 Visualizing Outputs from Intermediate Layers

We can visualize outputs from Intermediate Layers, using PyTorch utility function: `register_forward_hook`

In [4]:
```
vgg = models.vgg16(pretrained=True)
```

In [5]:
```
class LayerActivations():
    features=None

    def __init__(self,model,layer_num):
        self.hook = model[layer_num].register_forward_hook(self.hook_fn)

    def hook_fn(self,module,input,output):
        self.features = output.cpu().data.numpy()

    def remove(self):
        self.hook.remove()
```

In [6]:
```
conv_out = LayerActivations(vgg.features, 0)
```

Let's create a sample image

```
In [52]: img = []
         for i in range(64):
             img.append(np.random.randn(16, 16, 3))
```

```
In [53]: img = np.array(img)
         img.shape
```

```
Out[53]: (64, 16, 16, 3)
```

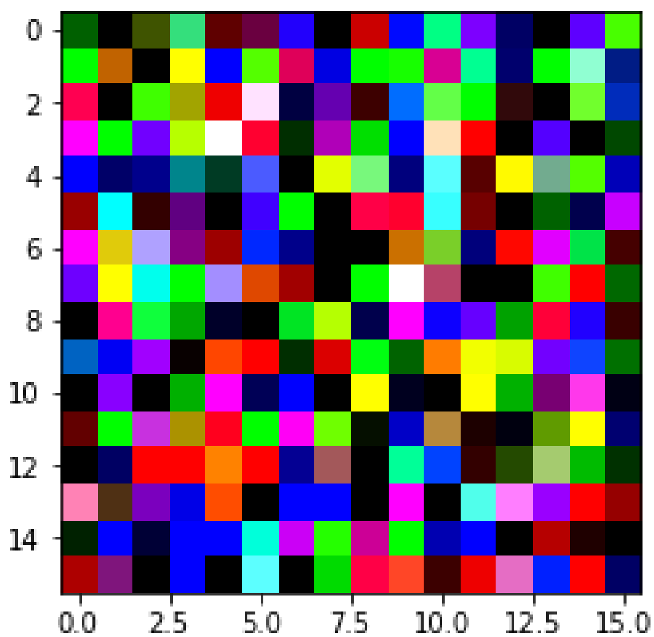
This is the image created.

```
In [54]: img = img.reshape((64, 16, 16, 3))
```

```
In [55]: plt.imshow(img[0])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
```

```
Out[55]: <matplotlib.image.AxesImage at 0x7f5cf7f02ba8>
```




```
In [56]: img = img.reshape((64, 3, 16, 16))
```

```
In [57]: img = torch.from_numpy(img).float()
```

```
In [58]: print(img.shape)
```

```
torch.Size([64, 3, 16, 16])
```

This wasn't working on my PC, lack of free RAM you know? ;)
Thanks to Google Collabs :D Implemented it here: (on a random input image)
https://colab.research.google.com/drive/12UH6rwswmxIuvIvvHT-wjB0LNVj-Txw9

0.9 Visualizing weights of Intermediate Layers

In [65]:
```
cnn_weights = vgg.state_dict()['features.0.weight'].cpu()
```

In [68]:
```
# directly taken from the book
fig = plt.figure(figsize=(30,30))
fig.subplots_adjust(left=0,right=1,bottom=0,top=0.8,hspace=0,wspace=0.2)
for i in range(30):
    ax = fig.add_subplot(12,6,i+1,xticks=[],yticks=[])
    plt.imshow(cnn_weights[i])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255]
```