Transfer Learning

In [2]:
```
import torch
import numpy as np
import matplotlib.pyplot as plt
import pickle
from tqdm import tqdm
```

In [3]:
```
# DATA PARSING
# If this is running on GPU cluster, no change required
# Otherwise, download CIFAR-10 dataset from
# https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz and set path
path='../../../../CIFAR-10/cifar-10-batches-py/'
data=np.zeros((0,32,32,3))
labels=[]
for i in range(1,6):
    with open(path+'data_batch_'+str(i), 'rb') as fo:
        dat = pickle.load(fo)
        r = dat['data'][:,:1024*1].reshape((10000,32,32,1))
        g = dat['data'][:,1024:2048].reshape((10000,32,32,1))
        b = dat['data'][:,2048:3072].reshape((10000,32,32,1))
        rgb = np.concatenate((r,g,b),axis=3)
        data = np.vstack((data,np.float32(rgb)/255))
        labels += dat['labels']
labels = np.array(labels)
# data -> 50000 X 32 X 32 X 3 array with training data
# labels -> 50000 labels ranging from 0 to 9
```

[2 points] Plot 3 random images corresponding to each label from the training data and indicate the name of the class label.

[0 points] Now, we perform some pre-processing operations to get our training datasets.

1. Split the data and labels into 2 sets, first one containing labels 0 to 4, and second one from 5 to 9.
2. Generate one hot encoded targets based on the labels for the 2 sets.
3. Store them in data1, labels1, data2 and labels2.

In [4]:
```
data1 = np.zeros((0,32,32,3))
labels1 = []
data2 = np.zeros((0,32,32,3))
labels2 = []
```

```
for i in range(5):
    x = data[labels==i]
    data1 = np.vstack((data1,x))
    labels1 += [i]*len(x)
for i in range(5,10):
    x = data[labels==i]
    data2 = np.vstack((data2,x))
    labels2 += [i-5]*len(x)
labels1 = np.array(labels1)
labels2 = np.array(labels2)

temp = np.zeros((len(labels1),5))
for i in range(len(labels1)):
    temp[i,labels1[i]] = 1
labels1 = temp
temp = np.zeros((len(labels2),5))
for i in range(len(labels2)):
    temp[i,labels2[i]] = 1
labels2 = temp

torch_data1 = data1.transpose((0,3,1,2))
torch_data2 = data2.transpose((0,3,1,2))
```

[3 points] Create a simple convolutional network to classify the training data. The network structure should be as follows:

- Layer 1: Kernel size 4, stride 2, output channels 5, bias enabled, ReLU activation
- Layer 2: Kernel size 4, stride 1, output channels 10, bias enabled, ReLU avtication
- Layer 3: Kernel size 4, stride 1, output channels 20, bias enabled, ReLU activation
- Layer 4: Kernel size 4, stride 1, output channels 40, bias enabled, ReLU activation
- Layer 5: Fully connected layer followed by sigmoid activation

Refer to https://github.com/ameykusurkar/pytorch-image-classifier/blob/master/main.py for help from this section onwards, but note that `torchvision.transforms` is not required since we already have the data in the required format.

```
In [28]: import torch.nn as nn
         import torch.nn.functional as F
         import torch.optim as optim
         from torch.autograd import Variable

         n=5
         class Net(nn.Module):
             def __init__(self):
                 super(Net, self).__init__()
                 self.conv1 = YOUR CODE HERE
```

```
        self.conv2 = YOUR CODE HERE
        self.conv3 = YOUR CODE HERE
        self.conv4 = YOUR CODE HERE
        self.fc = YOUR CODE HERE

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = YOUR CODE HERE
        x = YOUR CODE HERE
        x = YOUR CODE HERE
        x = F.sigmoid(YOUR CODE HERE)
        return x
```

```
In [ ]: # print the network structure
        # Using a GPU is highly recommended since training will take a while otherwise
        net = Net().cuda()
        print(net)
```

[5 points] Create a function that trains the network using the provided data. However it should only train the part of the network that is passed in as a parameter.

1. Training data must be randomly sampled to obtain a batch of data which is a subset of the whole training dataset at every iteration.
2. Use the Adam optimizer and BCELoss function (Binary Cross Entropy Loss).
3. Store the loss as well as the accuracy of the network on the training data at every iteration and return them in arrays at the end.

```
In [33]: # to_train can be net.paramaters OR net.fc.parameters OR net.conv1.parameters so that o
         def train(tdata,tlabel,to_train):
             criterion = YOUR CODE HERE
             losslist = []
             acc = []
             epochs = YOUR CODE HERE
             batch = YOUR CODE HERE
             learning_rate = YOUR CODE HERE
             optimizer = YOUR CODE HERE
             for k in tqdm(range(epochs)):
                 for l in range(int(len(tdata)/batch)):
                     inds = np.random.randint(0,len(tdata)-1,batch)
                     inputs = Variable(torch.FloatTensor(tdata[inds]).cuda())
                     targets = Variable(torch.FloatTensor(tlabel[inds]).cuda())
                     optimizer.zero_grad()
                     prediction = YOUR CODE HERE
                     loss = YOUR CODE HERE
                     loss.backward()
                     optimizer.step()
                     losslist.append(loss.data.cpu().numpy())
```

```
                acc.append(np.mean(np.argmax(prediction.data.cpu().numpy(),1)==np.argmax(tl

        return losslist,acc
```

[5 points] Initialize the network, train the complete network (net.parameters) on data1 (the first 5 classes) and plot the loss and accuracy vs iterations on the same graph. Print the final loss and accuracy as well. Set the learning rate, number of iterations and batch size such that the loss is gradually and smoothly decreasing and converging. The accuracy at the end of training must be at least 35 %. Suggested parameters are: batch size greater than 300, learning rate in the order of 1e-5 and at least 100 iterations for these parameters.

```
In [ ]: # Initialize net
        net = Net().cuda()
        x1,a1 = train(YOUR CODE HERE)
        ax = range(len(x1))
        plt.plot(ax,x1,ax,a1)
        plt.show()
        print(x1[-1])
        print(a1[-1])
```

[2 points] Without reinitializing the network, train only the fully connected layer (net.fc.parameters) now on data2 (the next 5 classes). Do not change any hyper parameters such as learning rate or batch size. Plot the loss and accuracy and print the final values like before.

```
In [ ]: x2,a2 = train(YOUR CODE HERE)
        ax = range(len(x2))
        plt.plot(ax,x2,ax,a2)
        plt.show()
        print(x2[-1])
        print(a2[-1])
```

[3 points] Now repeat the process in the opposite order. Initialize the net again, train the whole network on data2, generate the same plots as before, and then without reinitializing the net, train only the fully connected layer on data1 and generate the plots. Do not change any hyperparameters.

```
In [ ]: # Initialize net
        net = Net().cuda()
        x3,a3 = train(YOUR CODE HERE)
        ax = range(len(x3))
        plt.plot(ax,x3,ax,a3)
        plt.show()
        print(x3[-1])
        print(a3[-1])
```

```
In [ ]: x4,a4 = train(YOUR CODE HERE)
        ax = range(len(x4))
        plt.plot(ax,x4,ax,a4)
        plt.show()
        print(x4[-1])
        print(a4[-1])
```

[5 points] Plot the loss vs iterations graphs obtained in the previous 4 training operations on the same graph, to visualize the effects of transfer learning. Explain the results obtained, based on the training regimen. Comment on the performance of transfer learning in each setting.

```
In [ ]: ax = range(len(x1))
        plt.plot(ax,x1,ax,x3,ax,x2,ax,x4)
        plt.legend(['random init 1','random init 2','transfer learning 1','transfer learning 2']
        plt.show()
```

[0 points] Create a network with more layers, pooling layers, and more filters and try to increase accuracy as much as possible. Play around with the hyperparameters to understand how they affect the training process. No need to submit anything for this part.