STA 208 Project: CNNs and Bandwidth

The folowing is the code for the project. In the first few cells we import the common libraries and classes for the experiments.

Import libraries

In [34]:

```
from __future__ import print_function

#tesnsorflow
import tensorflow as tf

#keras
import keras
from keras.layers import Dense, Conv2D, BatchNormalization, Activation
from keras.layers import AveragePooling2D, Input, Flatten
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras.callbacks import ReduceLROnPlateau
from keras.preprocessing.image import ImageDataGenerator
from keras.regularizers import l2
from keras.models import Model,Sequential
from keras.datasets import cifar10

#other
import time
import os


#plot
import matplotlib.pyplot as plt
import numpy as np


#sklearn
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.model_selection import cross_val_score
from sklearn.decomposition import PCA
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

#BW
from heapq import heappush, heappop, heapify
from collections import defaultdict
import numpy as np
import copy
from tqdm import tqdm

import numpy as np
from sklearn.decomposition import PCA
from keras.datasets import cifar10
from keras import layers
from keras import models
from keras.utils import to_categorical
from heapq import heappush, heappop, heapify
from collections import defaultdict
import numpy as np
import copy
from tqdm import tqdm
import matplotlib.pyplot as plt

```

BW calculation - To calculate the average bandwidth of a set of images we calculated the total number of bytes required for the loss less transmission of the set of images and calculated the average. For this, first we mapped the pixel/coefficient values of the dataset to a dictionary and calculated the frequency of each codeword. Then we generated binary strings to represent each codeword with **Huffman encoding**. To make sure that the codewords are integers, we quantized the pixel/coefficient values and used these quantized coefficients for later processes like reconstruction and classification. This way the effect of quantization is also reflected on the classification accuracy.

In [35]:

```
def encode(symb2freq):
    """Huffman encode the given dict mapping symbols to weights"""
    heap = [[wt, [sym, ""]] for sym, wt in symb2freq.items()]
    heapify(heap)
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

def HuffmanBW(x_test):
    """ input: x_test
        output: average BW/image in Bytes
    """
    bitstream=0
    for imnum in range(0,x_test.shape[0]):
        img = x_test[imnum,]
        bitcount=0

        if(np.min(img) < 0):
            img = img - np.min(img)
        if(np.max(img)==0):
            img[0,0]=1

        txt = "".join(map(chr, img.flatten()))
        symb2freq = defaultdict(int)

        for ch in txt:
            symb2freq[ch] += 1

        huff = encode(symb2freq)
      ##### Calculating the number of bits needed

        for p in huff:
            bitcount = bitcount+  symb2freq[p[0]]*len(p[1])
        bitstream = bitstream + bitcount
    BW = np.round(bitstream/(x_test.shape[0]*8))             ## BW in bytes
    return BW
```

Data augmentation

By randomly changing the images of each minibatch can help to solve the overfitting problem. For this, we used the following custom data augmentation generator to randomly flip and shift the training images.

In [36]:

```
def creategen(X,Y,batch_size):
    while True:
        # suffled indices
        #idx = np.random.permutation( X.shape[0])
        # create image generator
        datagen = ImageDataGenerator(

                featurewise_center=False,  # set input mean to 0 over the dataset
                samplewise_center=False,  # set each sample mean to 0
                featurewise_std_normalization=False,  # divide inputs by std of the data
                samplewise_std_normalization=False,  # divide each input by its std
                zca_whitening=False,  # apply ZCA whitening
                rotation_range=0,  # randomly rotate images in the range (degrees, 0 to
                width_shift_range=0.1,  # randomly shift images horizontally (fraction o
                height_shift_range=0.1,  # randomly shift images vertically (fraction of
                horizontal_flip=True,  # randomly flip images
                vertical_flip=False)

        batches= datagen.flow( X, Y, batch_size=batch_size,shuffle=True)

        idx0 = 0
        for batch in batches:
            idx1 = idx0 + batch[0].shape[0]
            temp = batch[0].astype('float32')
            #waveletmy2.batchwaveletcdf97mat(batch[0].astype('float32'),M,16)
            #temp = waveletmy2.batchwaveletsArrange(temp)

            yield temp/np.max(np.abs(temp)) , batch[1]

            idx0 = idx1
            if idx1 >= X.shape[0]:
                break
```

Learning rate scheduler - We used an initial learning rate of 0.001 which is reduced progressively at 80, 120, 160 and 180 over 200 epochs.

In [37]:

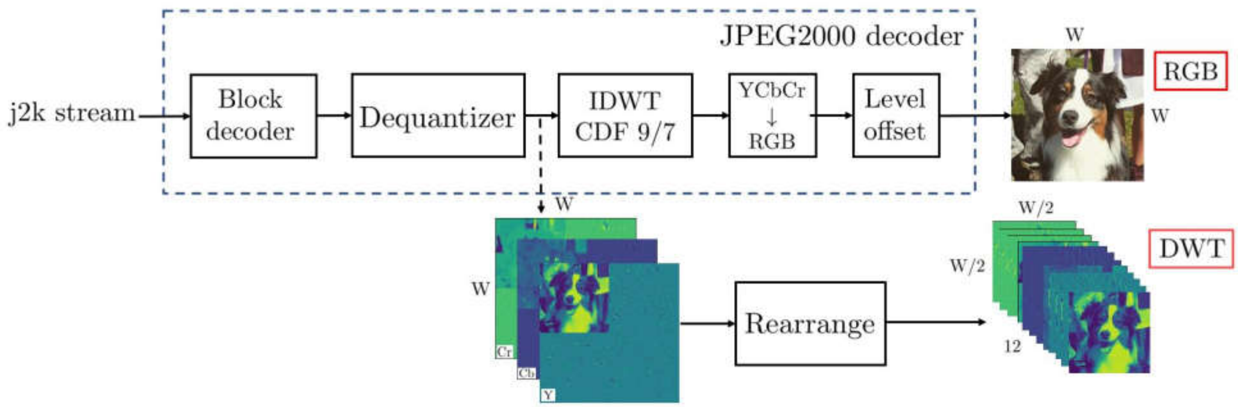
```
def lr_schedule(epoch):
    lr = 1e-3
    if epoch > 180:
        lr *= 0.5e-2
    elif epoch > 160:
        lr *= 1e-2
    elif epoch > 120:
        lr *= 1e-1
    elif epoch > 80:
        lr *= 0.5
    print('Learning rate: ', lr)
    return lr
```

Resnet - Following is the code for ResNet. We started with keras example code for CIFAR-10 and and changes as necessary.

In [38]:
```
n=1 #this is an indicator or depth (See keras Resnet implementation for CIFAR-10 for mor
#the following are the n values of the models a,b,c,d,e,f
#a: n= 4
#b: n= 3
#c: n= 2
#d: n= 3
#e: n= 2
#f: n= 1

depth = n * 6 + 2 #model depth

# Model name, depth and version
model_type = 'ResNet%d' % (depth)

def resnet_layer(inputs,
                 num_filters=64,
                 kernel_size=3, ######################## try to change this and see
                 strides=1,
                 activation='relu',
                 batch_normalization=True,
                 conv_first=True):
    """2D Convolution-Batch Normalization-Activation stack builder
    # Arguments
        inputs (tensor): input tensor from input image or previous layer
        num_filters (int): Conv2D number of filters
        kernel_size (int): Conv2D square kernel dimensions
        strides (int): Conv2D square stride dimensions
        activation (string): activation name
        batch_normalization (bool): whether to include batch normalization
        conv_first (bool): conv-bn-activation (True) or
            bn-activation-conv (False)
    # Returns
        x (tensor): tensor as input to the next layer
    """
    conv = Conv2D(num_filters,
                  kernel_size=kernel_size,
                  strides=strides,
                  padding='same',
                  kernel_initializer='he_normal',
                  kernel_regularizer=l2(1e-4))

    x = inputs
    if conv_first:
        x = conv(x)
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
    else:
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
        x = conv(x)
    return x


def resnet_v1(input_shape, depth, num_classes=10,num_filters=16,pool_size=8):
    """ResNet Version 1 Model builder [a]
    Stacks of 2 x (3 x 3) Conv2D-BN-ReLU
    Last ReLU is after the shortcut connection.
    At the beginning of each stage, the feature map size is halved (downsampled)
    by a convolutional layer with strides=2, while the number of filters is
    doubled. Within each stage, the layers have the same number filters and the
```

Faster and more accurate classification

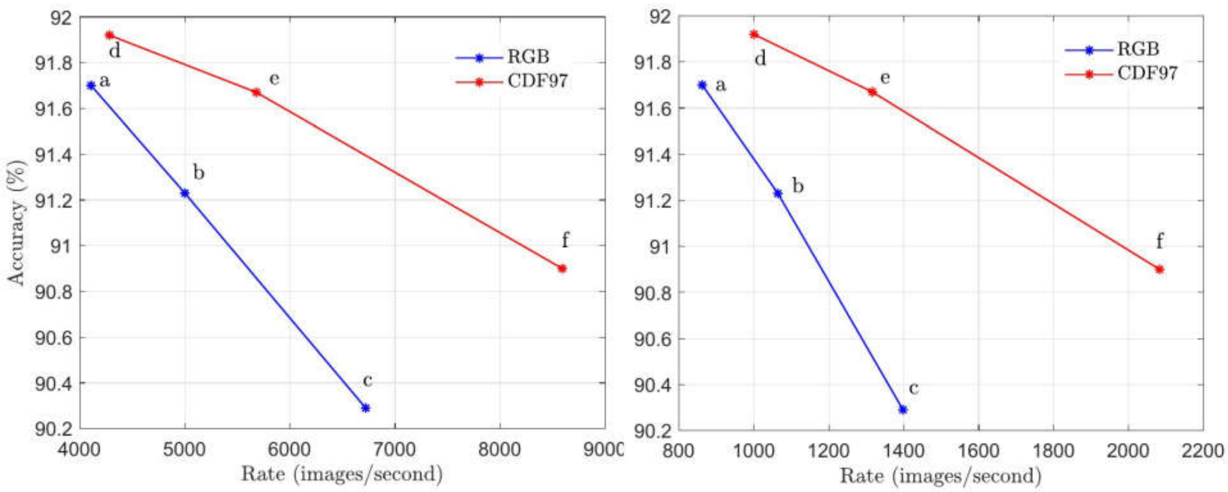
Assume a cloud based image classification scenario. Here source device (mobile phone) sends an inference image over a bandlimited channel to the server to get the class label. Server receives the inference image and feeds it to a trained classifier to predict the class label. In order to conserve limited channel bandwidth and storage capacity, source devices often encode and compress the images before transmitting to the cloud by utilizing standardized compression techniques such as JPEG2000.Because most neural networks are designed to classify images in the spatial RBG domain, the cloud currently receives and decodes the compressed j2k images back into the RGB domain before forwarding them to trained neural networks for further processing, as illustrated in the top part of the follwing figure. Thus, a natural question arises is to how to achieve faster training and inference with improved accuracy in a cloud based image classification under bandwidth, storage and computation constraints.



We claim that the conventional use of image reconstruction is unnecessary for JPEG2000 encoded classification by constructing and training a deep CNN model with the DWT coefficients with CDF 9/7 wavelets. See the bottom part of the above figure. Furthermore, we establish that more accurate classification is also possible by deploying shallower models to benefit from faster training and classification in comparison to models trained fo spatial RGB image inputs.

Experiment 1

We trained a set of ResNet models for CIFAR-10 dataset and following figures compare the test accuracy and speed for training and inference process.



(a) inference (b) training

In the above figure, (a) illustrates test accuracy vs inference speed for the CIFAR-10 data set. The blue lines represent results using reconstructed RGB images. Red curve is the result using DWT coefficients with CDF 9/7 wavelets. (b) shows the Test error vs training speed/epoch. Here rate is the number of images that go through the model in each epoch. The proposed model delivers fast and accurate classification for both training and inference. The points a,b,c,d,e and f correspond to 6 different ResNet models. The following table summerices these models.

Parameter/Domain	RGB			CDF 9/7		
Model	a	b	c	d	e	f
n	4	3	2	3	2	1
no of CONV layers	27	21	15	21	15	9
no of parameters (M)	0.37	0.27	0.18	1.79	1.17	0.55

Following is the code for the above implementation.

We used the following ResNet model (reference - keras). For each model a,b,c,d,e and f, we changed the number of residual blocks (n). For RGB domain we started with the number of filters 16 and doubled the number of filters as we downsample the input. For DWT domain we start with 64 filters and increased them by a factor of 1.5 as we down sample the input. The reason the different treatments for different domain is the number of channels and input height and width for the different domain. RGB inputs are 32x32x3 and DWT inputs are 16x61x12.

we used the following methods for preprocessing to follow the JEPG2000 encoder as closely as possible. Following are the steps.

- Level offset : deduct 128 from the inputs
- Convert RGB to YCbCr color space
- Take DWT with CDF 9/7 wavelets - we implement this linear transformation as a matrix multiplication.
- divide the input by the maximum absolute value to normalize. (why - works best with DWT according to literature)

Following are the methods.

In [39]:

```
from numpy.linalg import inv

#RGB2YCbCr - RGB to YCbCr conversion
def batchRGB2YCRCB(x_batch):
    alpha_R = 0.299
    alpha_G = 0.587
    alpha_B = 0.114
    x_batchnew = np.zeros((x_batch.shape)).astype('float32')
    for i in range(0,x_batch.shape[0]):
        #Y
        x_batchnew[i,:,:,0] = alpha_R*x_batch[i,:,:,0] + alpha_G*x_batch[i,:,:,1]
        #Cb
        x_batchnew[i,:,:,1] = (0.5/(1-alpha_B))*(x_batch[i,:,:,2]-x_batchnew[i,:,:,
        #Cr
        x_batchnew[i,:,:,2] = (0.5/(1-alpha_R))*(x_batch[i,:,:,0]-x_batchnew[i,:,:,
    return x_batchnew


#generate the matrix for CDF 9/7 transform
def getTcdf97(height):
    a1 = -1.586134342
    a2 = -0.05298011854
    a3 = 0.8829110762
    a4 = 0.4435068522

    # Scale coeff:
    k1 = 0.8128662109 # 1/1.230174104914 // 0,2,4,6
    k2 = 0.6149902344 # 1.230174104914/2 // 5038 1,3,5,7
    X1 = np.identity(height)
    X2 = np.identity(height)
    X3 = np.identity(height)
    X4 = np.identity(height)
    X5 = np.zeros((height,height)).astype('float32')
    for col in range(1,height-2,2):
        X1[col-1,col]=X1[col+1,col]=a1
    X1[height-2,height-1] = 2*a1

    #print(X1)
    for col in range(2,height-1,2):
        X2[col-1,col]=X2[col+1,col]=a2
    X2[1,0] = 2*a2
    #print(X2)
    for col in range(1,height-2,2):
        X3[col-1,col]=X3[col+1,col]=a3
    X3[height-2,height-1] = 2*a3

    #print(X1)
    for col in range(2,height-1,2):
        X4[col-1,col]=X4[col+1,col]=a4
    X4[1,0] = 2*a4

    for col in range(0,height,1):
        if(col%2==0 ):
            #print(col)
            X5[col,int(col/2)]=k1
        else:
            X5[col,int(height/2 + (col-1)/2)]=k2
    #print(X3)
    X =np.matmul(np.matmul(np.matmul(np.matmul(X1,X2),X3),X4),X5)
    return X,inv(X)

#take Level 1 DWT
def batchwaveletcdf97mat(x_batch,X,dimhalf):
    x_batchnew = np.zeros((x_batch.shape[0],dimhalf,dimhalf,12)).astype('float32'
```

Load the dataset and preprocessing - we used CIFAR-10 for all the experiemnts.

In [40]:
```
# Load the CIFAR10 data.
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
num_classes = 10
#convert to float32
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')


#level offset
X_train = X_train - 128.0
X_test = X_test - 128.0

#RGB2YCbCr - This converts RGB images to YCbCr format to facilitate compression - option
X_train = batchRGB2YCRCB(X_train)
X_test = batchRGB2YCRCB(X_test)

#generate necessary matrices for DWT cdf9/7 trandformation
M,M_inv = getTcdf97(32)

#take level-1 DWT with CDF 9/7
x_train = batchwaveletcdf97mat(X_train.astype('float32'),M,16)
x_test = batchwaveletcdf97mat(X_test.astype('float32'),M,16)

### max normalization
x_train=x_train/np.max(np.abs(x_train))
x_test=x_test/np.max(np.abs(x_test))

input_shape = x_test.shape[1:]
print('input shape to resnet: ',input_shape)
```

```
input shape to resnet:  (16, 16, 12)
```

Convert labels to one hot encoding

In [41]:
```
# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

Compile the model: we used Adam optimizer with batchsize 32. Original resnet paper uses 128 but we have limited GPU memory.

In [42]:
```
batch_size = 32
epochs = 1#200

model = resnet_v1(input_shape=input_shape, depth=depth,num_classes=num_classes,num_filte

model.compile(loss='categorical_crossentropy',optimizer=Adam(lr=lr_schedule(0)),metrics
model.summary()
print(model_type)
```

```
Learning rate:  0.001
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_4 (InputLayer)            (None, 16, 16, 12)   0
__________________________________________________________________________________________________
conv2d_28 (Conv2D)              (None, 16, 16, 64)   6976        input_4[0][0]
__________________________________________________________________________________________________
batch_normalization_22 (BatchNo (None, 16, 16, 64)   256         conv2d_28[0][0]
__________________________________________________________________________________________________
activation_22 (Activation)      (None, 16, 16, 64)   0           batch_normalizati
on_22[0][0]
__________________________________________________________________________________________________
conv2d_29 (Conv2D)              (None, 16, 16, 64)   36928       activation_
22[0][0]
__________________________________________________________________________________________________
batch_normalization_23 (BatchNo (None, 16, 16, 64)   256         conv2d_29[0][0]
__________________________________________________________________________________________________
activation_23 (Activation)      (None, 16, 16, 64)   0           batch_normalizati
on_23[0][0]
__________________________________________________________________________________________________
conv2d_30 (Conv2D)              (None, 16, 16, 64)   36928       activation_
23[0][0]
__________________________________________________________________________________________________
batch_normalization_24 (BatchNo (None, 16, 16, 64)   256         conv2d_30[0][0]
__________________________________________________________________________________________________
add_10 (Add)                    (None, 16, 16, 64)   0           activation_
22[0][0]
                                                                 batch_normalizati
on_24[0][0]
__________________________________________________________________________________________________
activation_24 (Activation)      (None, 16, 16, 64)   0           add_10[0][0]
__________________________________________________________________________________________________
conv2d_31 (Conv2D)              (None, 8, 8, 96)     55392       activation_
24[0][0]
__________________________________________________________________________________________________
batch_normalization_25 (BatchNo (None, 8, 8, 96)     384         conv2d_31[0][0]
__________________________________________________________________________________________________
activation_25 (Activation)      (None, 8, 8, 96)     0           batch_normalizati
on_25[0][0]
__________________________________________________________________________________________________
```

Set callback methods

In [43]:
```
lr_scheduler = LearningRateScheduler(lr_schedule)

callbacks = [lr_scheduler]
```

Train the model - We used a server with Titan-V GPU. (We train only for 1 epoch to show the code works. We used 200 epochs to train both RGB and DWT and models)

In [44]:
```
# Fit the model on the batches generated by datagen.flow().
model.fit_generator(creategen(x_train, y_train, batch_size=batch_size),
                    steps_per_epoch=int(np.ceil(x_train.shape[0]/32.0)),
                    epochs=epochs, verbose=0, workers=1,
                    callbacks=callbacks)
```

```
Learning rate:  0.001
```

Out[44]:
```
<keras.callbacks.History at 0x27594c02278>
```

10. Evaluate the test set - The results below is after 1 eopoch.(to show the code works)

In [45]:
```
start = time.time()
# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=1)
```

In [46]:
```
print('time per image :',(time.time()-start)*1000/10000,' ms')
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
time per image : 0.43476157188415526  ms
Test loss: 1.3211436420440674
Test accuracy: 0.5974
```

Experiment 2.1 (RGB)

The most basic test for dimensionality reduction will be RGB. Three channels are used for pictures in CIFAR10 dataset consisting of 32 x 32 images.

In [47]:
```
#Code for NN
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

#set RGB ResNet-8 parameters
image_size = 32
channel_num = 3
max_image = 255

epochs = 1#200
batch_size = 32
```

For RGB testing of accuracy and bandwidth, first designate function for calling the ResNet-8 function in a closed-off manner.

In [49]:
```
def ResCNN(x_train,y_train,x_test,y_test):

    #normalize data
    x_train = x_train.astype('float32') / max_image

    x_test = x_test.astype('float32') / max_image

    #set labels to numerical
    y_train = to_categorical(y_train)
    y_test = to_categorical(y_test)

    #set more ResNet-8 parameters
    input_shape = x_train.shape[1:]
    depth = 8

    #set number of labels
    num_classes = 10

    #call model from resnet_v1
    model = resnet_v1(input_shape=input_shape, depth=depth,num_classes=num_classes

    model.compile(loss='categorical_crossentropy',optimizer=Adam(lr=lr_schedule(0

    #set learning rate
    lr_scheduler = LearningRateScheduler(lr_schedule)

    callbacks = [lr_scheduler]

    model.fit_generator(creategen(x_train, y_train, batch_size=batch_size),
                        steps_per_epoch=int(np.ceil(x_train.shape[0]/32.0)),
                        epochs=epochs, verbose=0, workers=1,
                        callbacks=callbacks)

    #evaluate model
    test_loss, test_acc = model.evaluate(x_test, y_test)

    return test_loss, test_acc
```

In [50]:
```
#find accuracy and loss for ResNet-8
test_loss_Resnet, test_acc_Resnet = ResCNN(x_train,y_train,x_test,y_test)
```

```
Learning rate:  0.001
Learning rate:  0.001
10000/10000 [==============================] - ETA: 1: - ETA: 10s - ETA: 5 - ETA:
- ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  -
ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ET
A:  - ETA:  - ETA:  - 2s 174us/step
0.4802
```

In [51]:
```
print('Test accuracy after 1 epoch: ',test_acc_Resnet)
```

```
Test accuracy after 1 epoch:  0.4802
```

Next proccess labels and images for PCA reduction.

In [52]:
```
#fit model with PCA

#flatten
x_train_pca = x_train.reshape(x_train.shape[0],-1).astype('float32')
x_test_pca = x_test.reshape(x_test.shape[0],-1).astype('float32')

#use sklearn's pca tool to set up pca object
pca = PCA(0.99)

#fit all data
pca.fit_transform(x_train_pca)

#find projection matrix
x_train_pca_proj = pca.fit_transform(x_train_pca)

#reconstruct image
x_train_recon = pca.inverse_transform(x_train_pca_proj)

#reshape reconstructed image
x_train_recon = x_train_recon.reshape(x_train.shape).astype('float32')

#same as above for test images
x_test_pca_proj = pca.fit_transform(x_test_pca)
x_test_recon = pca.inverse_transform(x_test_pca_proj)
x_test_recon = x_test_recon.reshape(x_test.shape).astype('float32')
```

In [53]:
```
#find test accuracy and loss for PCA ResNet-8
test_loss_Resnet_PCA, test_acc_Resnet_PCA = ResCNN(x_train_recon,y_train,x_test_recon
```

```
Learning rate:  0.001
Learning rate:  0.001
10000/10000 [==============================] - ETA: 2: - ETA: 14s - ETA: 7 - ETA:
- ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  -
ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ETA:  - ET
A:  - ETA:  - ETA:  - 2s 183us/step
```

In [54]:
```
print('Test accuracy after 1 epoch: ',test_acc_Resnet_PCA)
```

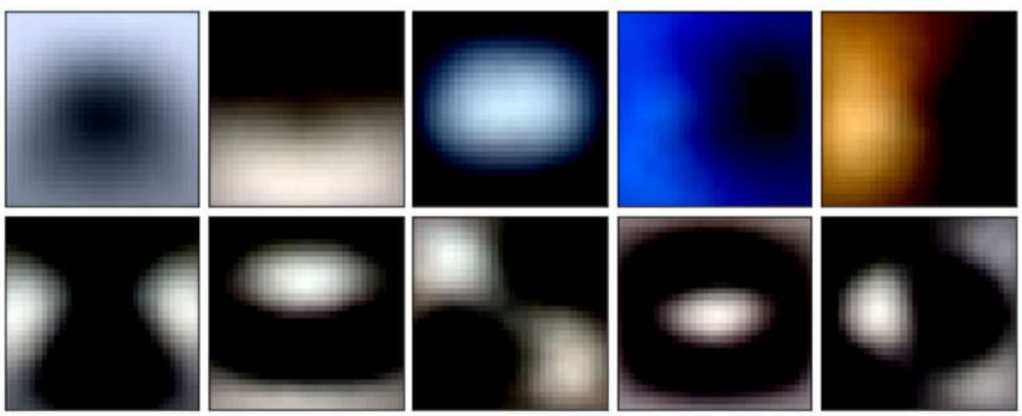
```
Test accuracy after 1 epoch:  0.4697
```

Now show the first few pca components.

In [0]:

```
#Setup a figure 8 inches by 8 inches
fig = plt.figure(figsize=(8,8))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
# plot the components, each image is 26 by 26 pixels
for i in range(10):
    ax = fig.add_subplot(5, 5, i+1, xticks=[], yticks=[])
    ax.imshow(np.reshape(pca.components_[i,:]/np.max(pca.components_[i,:]), (32,32
```

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



Display original RGB image (Red)

In [0]:

```
#show original images
#Setup a figure 8 inches by 8 inches
fig = plt.figure(figsize=(8,8))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
# plot the components, each image is 26 by 26 pixels
print('Original images:')
for i in range(10):
    ax = fig.add_subplot(5, 5, i+1, xticks=[], yticks=[])
    ax.imshow(np.reshape(x_train[i,:,:,0]/np.max(x_train[i,:,:,0]), (32,32)), cmap
```

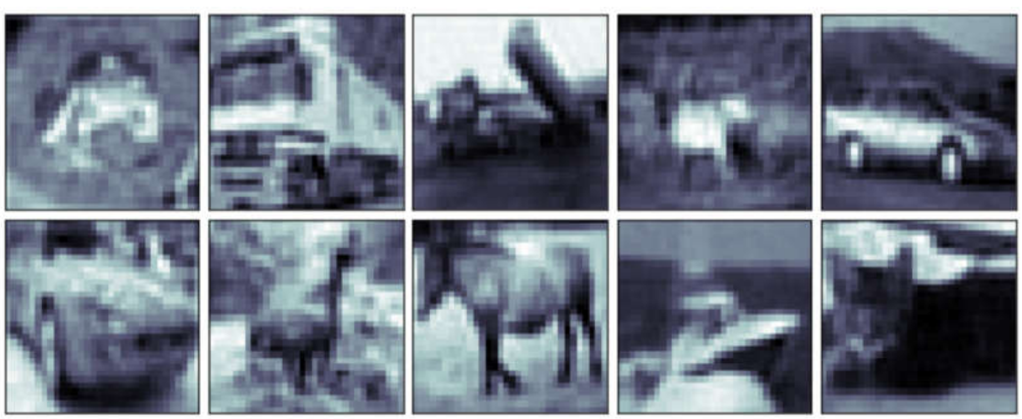
```
Original images:
```

Show reconstructed image after PCA. Notice how similar they look even with fewer components.

In [0]:

```
#recon images
fig = plt.figure(figsize=(8,8))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
# plot the components, each image is 26 by 26 pixels
print('reconstructed images:')
for i in range(10):
    ax = fig.add_subplot(5, 5, i+1, xticks=[], yticks=[])
    ax.imshow(np.reshape(x_train_recon[i,:,:,0]/np.max(x_train_recon[i,:,:,0]), (
```

```
reconstructed images:
```



Doing the bandwidth calculations,

In [56]:
```
BWtrain = HuffmanBW(x_train)
BWtrain_PCA = HuffmanBW(x_train_pca_proj)

print('average BW of the original images: ',BWtrain)
print('average BW of the PCA reduced images: ',BWtrain_PCA)
```

```
average BW of the original images:  2800.0
average BW of the PCA reduced images:  603.0
```

Let's sumarize the experiment results after 200 epochs

We obtained the following results.

Method	Test Accuracy (%)	Bandwidth (Bytes)
Original RGB	88.0	2800
PCA RGB	84.0	603

Experiment 2.2 (DCT)

We will also be applying a discrete cosine transform to the images to see the effect of this transform on the accuracy and bandwidth. Then we perform PCA to reduce dimension to reduce BW.

In [58]:
```
# DCT
from scipy.fftpack import dct, idct
from numpy import r_

#custom DCT function to do 8x8 block wise DCT
def dct2(a):
    dcted = dct(dct( a, axis=0, norm='ortho' ), axis=1, norm='ortho' )

    return dcted

def block_2d_dct(im,block_size):
    imsize = im.shape
    dct = np.zeros((4,4,64)).astype('int32')

    # Do 8x8 DCT on image (in-place)
    for i in r_[:imsize[0]:block_size]:
        for j in r_[:imsize[1]:block_size]:
            dct[int(i/block_size),int(j/block_size),:] = dct2( im[i:(i+block_size

    return dct

def block_batch_dct_2D(x_batch):
    x_batchnew = np.zeros((x_batch.shape[0],4,4,64*3)).astype('int')
    xbatch_size = x_batch.shape[0]
    n_channels = x_batch.shape[3]
    for i in range(0,xbatch_size):
        for j in range(0,n_channels):
            x_batchnew[i,:,:,j*64:(j+1)*64]=block_2d_dct(x_batch[i,:,:,j],8)
    return x_batchnew

def reduce_data(x_train, y_train, x_test, y_test, n, m):
    return (x_train[:n], y_train[:n]) , (x_test[:m], y_test[:m])
```

Let's defind the ResNet-8 model. DCT input has the shape 4x4x192. To support 192 channels we use 200 filters as the initial number of filters. Further more we change the average pooling kernel size to 1.

In [66]:

```
def ResCNN2(x_train,y_train,x_test,y_test):

    #normalize data
    x_train = x_train.astype('float32') / max_image

    x_test = x_test.astype('float32') / max_image

    #set labels to numerical
    y_train = to_categorical(y_train)
    y_test = to_categorical(y_test)

    #set more ResNet-8 parameters
    input_shape = x_train.shape[1:]
    depth = 8

    #set number of labels
    num_classes = 10

    #call model from resnet_v1
    model = resnet_v1(input_shape, depth, num_classes=num_classes,num_filters=200

    model.compile(loss='categorical_crossentropy',optimizer=Adam(lr=lr_schedule(0

    #set learning rate
    lr_scheduler = LearningRateScheduler(lr_schedule)

    callbacks = [lr_scheduler]

    model.fit_generator(creategen(x_train, y_train, batch_size=batch_size),
                        steps_per_epoch=int(np.ceil(x_train.shape[0]/32.0)),
                        epochs=epochs, verbose=0, workers=1,
                        callbacks=callbacks)

    #evaluate model
    test_loss, test_acc = model.evaluate(x_test, y_test)

    return test_loss, test_acc
```

The below code applies the transformations.

In [61]:

```
#Code for NN
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

#Reduce data for code testing
#(x_train, y_train), (x_test, y_test) = reduce_data(x_train, y_train, x_test, y_test, 50

#set RGB ResNet-8 parameters
image_size = 32
channel_num = 3
max_image = 255

epochs = 1#200
batch_size = 32
```

In [62]:

```
#Apply DCT
x_train_dct = block_batch_dct_2D(x_train.astype('float'))
x_test_dct = block_batch_dct_2D(x_test.astype('float'))

print(x_train_dct.shape)
```

```
(50000, 4, 4, 192)
```

In [63]:

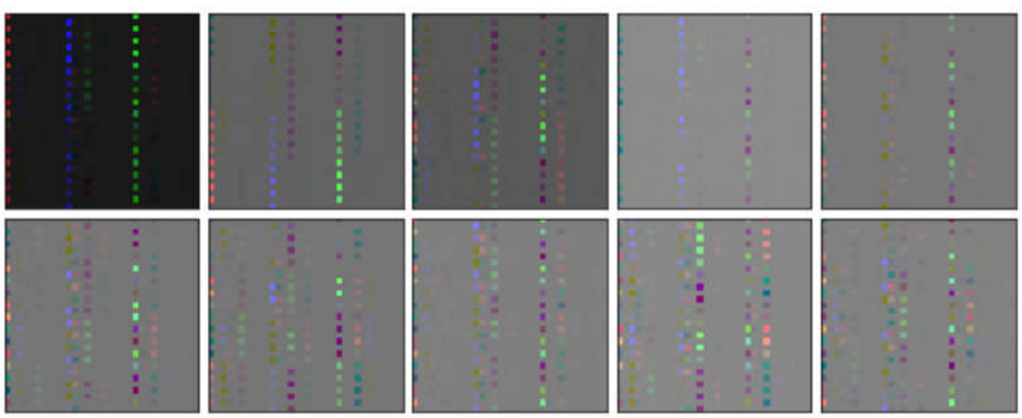
```
#fit model with PCA

#flatten
x_train_dct_flat = x_train_dct.reshape(x_train_dct.shape[0],-1).astype('float32')
x_test_dct_flat = x_test_dct.reshape(x_test_dct.shape[0],-1).astype('float32')

#use sklearn's pca tool to set up pca object
pca_dct = PCA(0.99)

#find projection matrix
x_train_dct_pca_proj = pca_dct.fit_transform(x_train_dct_flat)

#reconstruct image
x_train_dct_recon = pca_dct.inverse_transform(x_train_dct_pca_proj)

#reshape reconstructed image
x_train_dct_recon = x_train_dct_recon.reshape(x_train_dct.shape).astype('float32'

#same as above for test images
x_test_dct_pca_proj = pca_dct.fit_transform(x_test_dct_flat)
x_test_dct_recon = pca_dct.inverse_transform(x_test_dct_pca_proj)
x_test_dct_recon = x_test_dct_recon.reshape(x_test_dct.shape).astype('float32')

print(x_train_dct_recon.shape)
```

```
(50000, 4, 4, 192)
```

Let's visualize the first 10 picincipal images.

In [64]:

```
#Setup a figure 8 inches by 8 inches
fig = plt.figure(figsize=(8,8))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
# plot the components, each image is 26 by 26 pixels
for i in range(10):
    ax = fig.add_subplot(5, 5, i+1, xticks=[], yticks=[])
    image = pca_dct.components_[i,:]
    im_max = np.amax(image)
    im_min = np.amin(image)
    image = (image-im_min)/(im_max-im_min)
    ax.imshow(np.reshape(image, (32,32,3)), cmap=plt.cm.bone, interpolation='nearest'
```



The code below runs the bandwidth and training and testing. This was run on different local machines so output does not match results in report.

In [71]:
```
#Bandwidth DCT PCA calculations

BWtrain = HuffmanBW(x_train_dct_flat)
BWtrain_PCA = HuffmanBW(x_train_dct_pca_proj)

print('average BW of original DCT: ',BWtrain)
print('average BW of PCA DCT: ',BWtrain_PCA)
```

```
average BW of original DCT:  2266.0
average BW of PCA DCT:  590.0
```

In [ ]:
```
#Find accuracy and loss with DCT
test_loss_Resnet_dct, test_acc_Resnet_dct = ResCNN2(x_train_dct, y_train, x_test_dct
print('Test loss:', test_loss_Resnet_dct)
print('Test accuracy:', test_acc_Resnet_dct)
```

In [0]:
```
#Find accuracy and loss with DCT and PCA
test_loss_Resnet_dct, test_acc_Resnet_dct = ResCNN2(x_train_dct_recon, y_train, x_test_d
print('Test loss:', test_loss_Resnet_dct)
print('Test accuracy:', test_acc_Resnet_dct)
```

Let's sumarize the experiment results after 200 epochs

We obtained the following results.

Method	Test Accuracy (%)	Bandwidth (Bytes)
Original DCT	65.9	2266
PCA DCT	64.8	590

Experiment 2.3 (DWT)

In this experiment we start with Level 1 DWT coefficents with CDF 9/7 wavelet and explore methods to reduce the required bandwidth () in cloud based image classification. To create a baseline, we train a ResNet-8 with quantized CDF 9/7 DWT coefficients BW and measure the classification accuracy of CIFAR-10 dataset (say a_1) and calculate the BW with Huffman encoding. (Say BW_1). Then we perform a principal component analysis (PCA) on the vectorized dataset to reduce the dimentionality of data with the perpose of reducing bandwidth. We can calculate the average BW of these quantized PCA projections (say BW_2). Then we reconstruct the wavelet coefficients to the original dimension and train a ResNet-8 to obtain a_2 accuracy. We observed that BW_2 is much smaller than BW_1 and a_1 and a_2 are considerably close.

This observation implies than we can save bandwidth by only sending the projected DWT coefficients rather than sending the complete image with a negligible accuracy loss. We can apply this in to practice like this. We can do a PAC analysis on a large dataset like ImageNet and store the principal components at the server. Source device can calculate the projections and transmit these projections consuming smaller BW. At the server end, high dimentional image can be reconstructed using the stored PCA coefficients and feed to a classifier.

We obtained the following results.

Method	Test Accuracy (%)	Bandwidth (Bytes)
Original DWT quantized	90	1552
PCA DWT quantized	90	317

Following is the code for the implementation.

Load CIFAR-10, take DWT and flatten the data for PCA

In [72]:

```
# Load the CIFAR10 data.
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
num_classes = 10
#convert to float32
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')


#level offset
X_train = X_train - 128.0
X_test = X_test - 128.0

#RGB2YCbCr - This converts RGB images to YCbCr format to facilitate compression - option
X_train = batchRGB2YCRCB(X_train)
X_test = batchRGB2YCRCB(X_test)

#generate necessary matrices for DWT cdf9/7 trandformation
M,M_inv = getTcdf97(32)

#take level-1 DWT with CDF 9/7
x_train = batchwaveletcdf97mat(X_train.astype('float32'),M,16)
x_test = batchwaveletcdf97mat(X_test.astype('float32'),M,16)

#flattening for PCA
x_train_ori = x_train.copy()
x_train_ori = x_train_ori.astype('float32')
x_test_ori = x_test.copy()
x_test_ori = x_test_ori.astype('float32')
x_train = x_train.reshape(x_train.shape[0],-1).astype('float32')
x_test = x_test.reshape(X_test.shape[0], -1).astype('float32')

input_shape = x_train_ori.shape[1:]
print('input shape to resnet: ',input_shape)
print('Dataset shape after vectorizing: ',x_train.shape)
```

```
input shape to resnet:  (16, 16, 12)
Dataset shape after vectorizing:  (50000, 3072)
```

Calculate PCA and choose the components to preserve 99% of the variance

In [73]:

```
pca = PCA(0.99)
x_train_proj= pca.fit_transform(x_train)
x_train_proj = np.floor(x_train_proj/5)*5 #quantization

x_test_proj= pca.transform(x_test)
x_test_proj = np.floor(x_test_proj/5)*5


print('original representation',x_train.shape)
print('reduced representation: ',x_train_proj.shape)
```

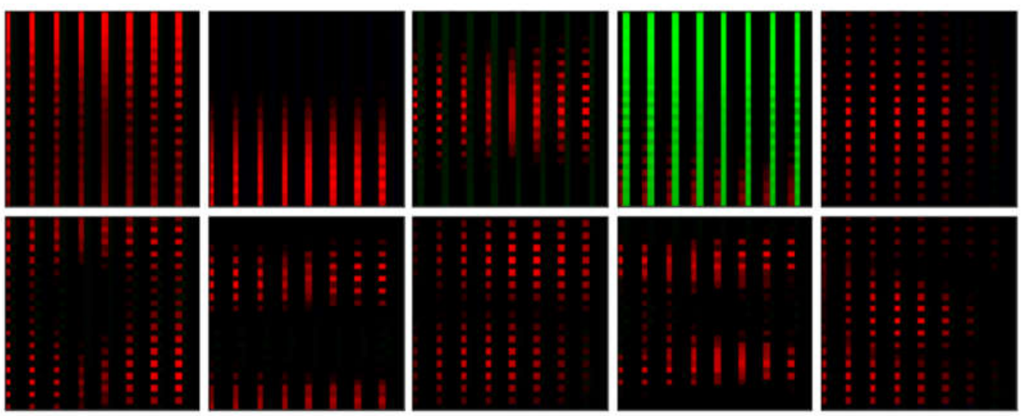
```
original representation (50000, 3072)
reduced representation:  (50000, 687)
```

With PCA we reduced the feature size from 3072 to 687. Let's display the first few principal compenents.

In [0]:

```
#Setup a figure 8 inches by 8 inches
fig = plt.figure(figsize=(8,8))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
# plot the components, each image is 26 by 26 pixels
for i in range(10):
    ax = fig.add_subplot(5, 5, i+1, xticks=[], yticks=[])
    ax.imshow(np.reshape(pca.components_[i,:]/np.max(pca.components_[i,:]), (32,32
```

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

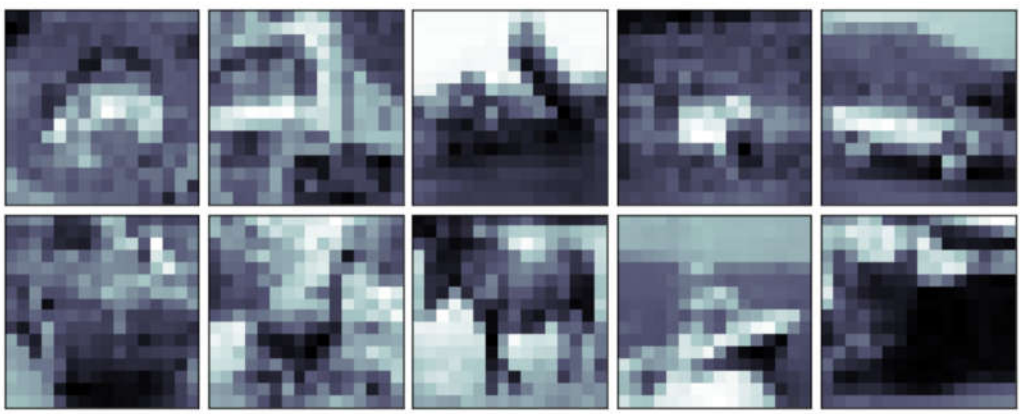


Now recontruct the images after dim reduction.For the display purposes we only show the 1st subband of 12 subbands of the images.

In [74]:

```
x_train_recon = pca.inverse_transform(x_train_proj)
x_train_recon = x_train_recon.reshape(x_train_ori.shape)

x_test_recon = pca.inverse_transform(x_test_proj)
x_test_recon = x_test_recon.reshape(x_test_ori.shape)

#show original images
#Setup a figure 8 inches by 8 inches
fig = plt.figure(figsize=(8,8))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
# plot the components, each image is 26 by 26 pixels
print('Original images:')
for i in range(10):
    ax = fig.add_subplot(5, 5, i+1, xticks=[], yticks=[])
    ax.imshow(np.reshape(x_train_ori[i,:,:,0]/np.max(x_train_ori[i,:,:,0]), (16,16

```

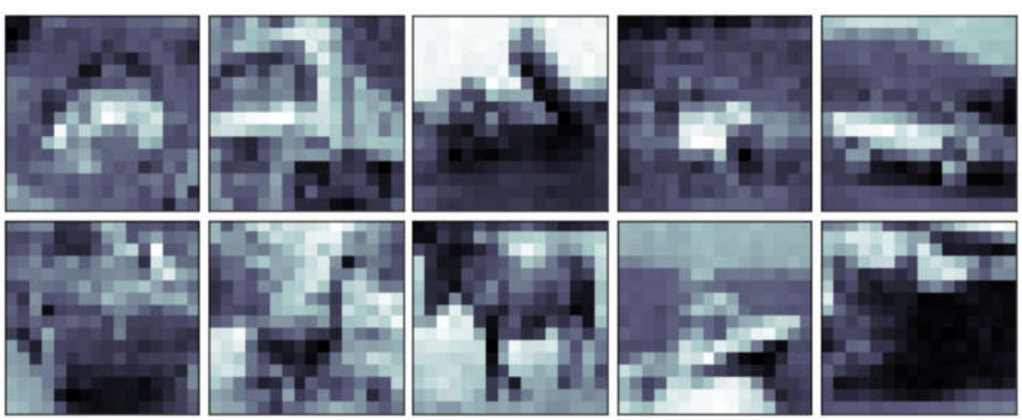
```
Original images:
```



In [75]:

```
#recon images
fig = plt.figure(figsize=(8,8))
fig.subplots_adjust(left=0, right=1, bottom=0, top=1, hspace=0.05, wspace=0.05)
# plot the components, each image is 26 by 26 pixels
print('reconstructed images:')
for i in range(10):
    ax = fig.add_subplot(5, 5, i+1, xticks=[], yticks=[])
    ax.imshow(np.reshape(x_train_recon[i,:,:,0]/np.max(x_train_recon[i,:,:,0]), (

```

```
reconstructed images:
```



Origianal and reconstructured images look similar. Now we can calculate BW_1 and BW_2.

BW_1

In [0]:
```
BWr = HuffmanBW(np.floor(x_train_ori).astype('int32'))
print('original images avg.BW/image: ',str(BWr),' Bytes')
```
```
original images avg.BW/image:  1552.0  Bytes
```

BW_2

In [0]:
```
BWp = HuffmanBW(x_train_proj.astype('int32'))
print('projections avg.BW/image: ',str(BWp),' Bytes')
```
```
projections avg.BW/image:  317.0  Bytes
```

Now lets train the network to obtain a_2.

In [76]:
```
### max normalization
x_train=x_train_recon/np.max(np.abs(x_train_recon))
x_test=x_test_recon/np.max(np.abs(x_test_recon))

input_shape = x_test.shape[1:]
print('input shape to resnet: ',input_shape)

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```
```
input shape to resnet:  (16, 16, 12)
```

Complile the ResNet-8 model

In [78]:
```
batch_size = 32
epochs = 1

model = resnet_v1(input_shape=input_shape, depth=depth,num_classes=num_classes,num_filte

model.compile(loss='categorical_crossentropy',optimizer=Adam(lr=lr_schedule(0)),metrics
model.summary()
print(model_type)
```

```
Learning rate:  0.001
____________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
====================================================================================================
input_8 (InputLayer)            (None, 16, 16, 12)   0
____________________________________________________________________________________________________
conv2d_64 (Conv2D)              (None, 16, 16, 64)   6976        input_8[0][0]
____________________________________________________________________________________________________
batch_normalization_50 (BatchNo (None, 16, 16, 64)   256         conv2d_64[0][0]
____________________________________________________________________________________________________
activation_50 (Activation)      (None, 16, 16, 64)   0           batch_normalizati
on_50[0][0]
____________________________________________________________________________________________________
conv2d_65 (Conv2D)              (None, 16, 16, 64)   36928       activation_
50[0][0]
____________________________________________________________________________________________________
batch_normalization_51 (BatchNo (None, 16, 16, 64)   256         conv2d_65[0][0]
____________________________________________________________________________________________________
activation_51 (Activation)      (None, 16, 16, 64)   0           batch_normalizati
on_51[0][0]
____________________________________________________________________________________________________
conv2d_66 (Conv2D)              (None, 16, 16, 64)   36928       activation_
51[0][0]
____________________________________________________________________________________________________
batch_normalization_52 (BatchNo (None, 16, 16, 64)   256         conv2d_66[0][0]
____________________________________________________________________________________________________
add_22 (Add)                    (None, 16, 16, 64)   0           activation_
50[0][0]
                                                                 batch_normalizati
on_52[0][0]
____________________________________________________________________________________________________
activation_52 (Activation)      (None, 16, 16, 64)   0           add_22[0][0]
____________________________________________________________________________________________________
conv2d_67 (Conv2D)              (None, 8, 8, 96)     55392       activation_
52[0][0]
____________________________________________________________________________________________________
batch_normalization_53 (BatchNo (None, 8, 8, 96)     384         conv2d_67[0][0]
____________________________________________________________________________________________________
activation_53 (Activation)      (None, 8, 8, 96)     0           batch_normalizati
on_53[0][0]
____________________________________________________________________________________________________
```

Set call backs and fit. (We train only for 1 epoch to show the code works. We used a Titan_V GPU with 200 epochs to train both original DWT and PCA models)

In [79]:
```
lr_scheduler = LearningRateScheduler(lr_schedule)


callbacks = [lr_scheduler]

# Fit the model on the batches generated by datagen.flow().
model.fit_generator(creategen(x_train, y_train, batch_size=batch_size),
                        steps_per_epoch=int(np.ceil(x_train.shape[0]/32.0)),
                        epochs=epochs, verbose=0, workers=1,
                        callbacks=callbacks)
```

```
c:\users\lahiru d. chamain\anaconda3\envs\tfgpumy\lib\site-packages\keras_preprocessing\image\numpy_array_iterator.py:127: UserWarning: NumpyArrayIterator is set to use the data format convention "channels_last" (channels on axis 3), i.e. expected either 1, 3, or 4 channels on axis 3. However, it was passed an array with shape (50000, 16, 16, 12) (12 channels).
  str(self.x.shape[channels_axis]) + ' channels).')

Learning rate:  0.001
```

Out[79]:
```
<keras.callbacks.History at 0x27614dd4d68>
```

Evaluate the testSet after 1 epoch.

In [80]:
```
start = time.time()
# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=0)
print('time per image :',(time.time()-start)*1000/10000,' ms')
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

```
time per image : 0.19965786933898927  ms
Test loss: 1.4426407526016236
Test accuracy: 0.558
```