In [1]:
```
# Import required packages
import numpy as np
import cv2
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report
from sklearn.linear_model import LogisticRegression
from tensorflow import keras
import tensorflow
from tensorflow.keras.models import Sequential,Model
from tensorflow.keras.layers import Dense, Dropout, Flatten,Input,Concatenate
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import LeakyReLU
from tensorflow.keras.utils import to_categorical
import time
from tensorflow.keras import datasets, layers, models
```

1. Load the datasets

For the project, we provide a training set with 50000 images in the directory `../data/images/` with:

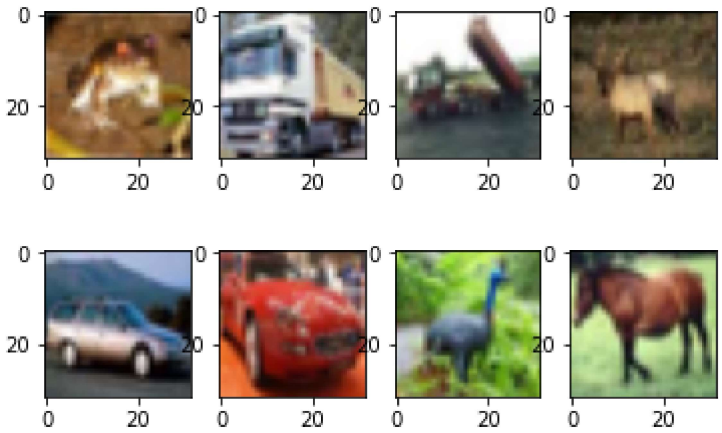
- noisy labels for all images provided in `../data/noisy_label.csv` ;
- clean labels for the first 10000 images provided in `../data/clean_labels.csv` .

In [2]:
```
# [DO NOT MODIFY THIS CELL]

# Load the images
n_img = 50000
n_noisy = 40000
n_clean_noisy = n_img - n_noisy
imgs = np.empty((n_img,32,32,3))
for i in range(n_img):
    img_fn = f'../data/images/{i+1:05d}.png'
    imgs[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)


# Load the labels
clean_labels = np.genfromtxt('../data/clean_labels.csv', delimiter=',', dtype=
"int8")
noisy_labels = np.genfromtxt('../data/noisy_labels.csv', delimiter=',', dtype=
"int8")
```

For illustration, we present a small subset (of size 8) of the images with their clean and noisy labels in `clean_noisy_trainset` . You are encouraged to explore more characteristics of the label noises on the whole dataset.

In [3]:
```
# [DO NOT MODIFY THIS CELL]

fig = plt.figure()

ax1 = fig.add_subplot(2,4,1)
ax1.imshow(imgs[0]/255)
ax2 = fig.add_subplot(2,4,2)
ax2.imshow(imgs[1]/255)
ax3 = fig.add_subplot(2,4,3)
ax3.imshow(imgs[2]/255)
ax4 = fig.add_subplot(2,4,4)
ax4.imshow(imgs[3]/255)
ax1 = fig.add_subplot(2,4,5)
ax1.imshow(imgs[4]/255)
ax2 = fig.add_subplot(2,4,6)
ax2.imshow(imgs[5]/255)
ax3 = fig.add_subplot(2,4,7)
ax3.imshow(imgs[6]/255)
ax4 = fig.add_subplot(2,4,8)
ax4.imshow(imgs[7]/255)

# The class-label correspondence
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# print clean labels
print('Clean labels:')
print(' '.join('%5s' % classes[clean_labels[j]] for j in range(8)))
# print noisy labels
print('Noisy labels:')
print(' '.join('%5s' % classes[noisy_labels[j]] for j in range(8)))
```

```
Clean labels:
 frog truck truck  deer   car   car  bird horse
Noisy labels:
  cat   dog truck  frog   dog  ship  bird  deer
```



2. The predictive model

We consider a baseline model directly on the noisy dataset without any label corrections. RGB histogram features are extracted to fit a logistic regression model.

2.1. Baseline Model

In [4]:
```
# [DO NOT MODIFY THIS CELL]
# RGB histogram dataset construction
no_bins = 6
bins = np.linspace(0,255,no_bins) # the range of the rgb histogram
target_vec = np.empty(n_img)
feature_mtx = np.empty((n_img,3*(len(bins)-1)))
i = 0
for i in range(n_img):
    # The target vector consists of noisy labels
    target_vec[i] = noisy_labels[i]

    # Use the numbers of pixels in each bin for all three channels as the feat
ures
    feature1 = np.histogram(imgs[i][:,:,0],bins=bins)[0]
    feature2 = np.histogram(imgs[i][:,:,1],bins=bins)[0]
    feature3 = np.histogram(imgs[i][:,:,2],bins=bins)[0]

    # Concatenate three features
    feature_mtx[i,] = np.concatenate((feature1, feature2, feature3), axis=None
)
    i += 1
```

In [5]:
```
# [DO NOT MODIFY THIS CELL]
# Train a logistic regression model
clf = LogisticRegression(random_state=0).fit(feature_mtx, target_vec)
```

For the convenience of evaluation, we write the following function `predictive_model` that does the label prediction. **For your predictive model, feel free to modify the function, but make sure the function takes an RGB image of numpy.array format with dimension $32 \times 32 \times 3$ as input, and returns one single label as output.**

In [6]:
```
# [DO NOT MODIFY THIS CELL]
def baseline_model(image):
    '''
    This is the baseline predictive model that takes in the image and returns
 a label prediction
    '''
    feature1 = np.histogram(image[:,:,0],bins=bins)[0]
    feature2 = np.histogram(image[:,:,1],bins=bins)[0]
    feature3 = np.histogram(image[:,:,2],bins=bins)[0]
    feature = np.concatenate((feature1, feature2, feature3), axis=None).reshap
e(1,-1)
    return clf.predict(feature)
```

2.2. Model I

Since our baseline model is based on simple logistic regression function, it extract the images' features as a matrix that contains the number of pixels between few RGB depth intervals. So we used another classical feature exactor algorithm called CNN (Convolutional Neural Network) to build our Model I, and CNN is widely used in nowadays computer vision development espically facial recognition and self-driving cars.

In [7]:
```
# [BUILD A MORE SOPHISTICATED PREDICTIVE MODEL]
# write your code here...


#Here we prepare the training sets and testing sets,
#since first 10000 imgs has their clean label, which is exactly the desired pr
ediction results that we want
#So the first 10k imgs and their clean labels are regarded as our validation d
atasets
#10k-50k imgs and their noisy labels are putting in the training datasets.
#(To avoid overfitting problem, we did not put first 10k imgs and their noisy
 labels into training datasets)

test_x = np.empty((n_noisy,32,32,3))
test_y = np.empty(n_noisy)
valid_x = np.empty((n_clean_noisy,32,32,3))
valid_y = np.empty(n_clean_noisy)

for i in range(n_noisy):
    img_fn = f'../data/images/{10000+i+1:05d}.png'
    test_x[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)
    test_y[i] = noisy_labels[10000+i]

for i in range(n_clean_noisy):
    img_fn = f'../data/images/{i+1:05d}.png'
    valid_x[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)
    valid_y[i]= clean_labels[i]

#Convert the label to one-hot encoding
test_y = to_categorical(test_y)
valid_y = to_categorical(valid_y)

test_x = test_x.astype("float32")
test_x = test_x/255

valid_x = valid_x.astype("float32")
valid_x = valid_x/255


batch_size = 64
epochs = 10
num_classes = 10

#Model Building Process
cnn_model = Sequential()
cnn_model.add(Conv2D(32, kernel_size=(3, 3),activation='linear',input_shape=(3
2,32,3),padding='same'))
cnn_model.add(LeakyReLU(alpha=0.1))
cnn_model.add(MaxPooling2D((2, 2),padding='same'))
cnn_model.add(Conv2D(64, (3, 3), activation='linear',padding='same'))
cnn_model.add(LeakyReLU(alpha=0.1))
cnn_model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
cnn_model.add(Conv2D(128, (3, 3), activation='linear',padding='same'))
cnn_model.add(LeakyReLU(alpha=0.1))
cnn_model.add(MaxPooling2D(pool_size=(2, 2),padding='same'))
cnn_model.add(Flatten())
```

```
cnn_model.add(Dense(128, activation='linear'))
cnn_model.add(LeakyReLU(alpha=0.1))
cnn_model.add(Dense(num_classes, activation='softmax'))

cnn_model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.
optimizers.Adam(),metrics=['accuracy'])


#Model Training
start = time.time()
cnn_train = cnn_model.fit(test_x, test_y, batch_size=batch_size ,epochs=epochs
,verbose=1,validation_data=(valid_x, valid_y))
end = time.time()
print("Runtime of the model 1's evaluation is %d seconds" % (end - start))
```

```
Epoch 1/10
625/625 [==============================] - 36s 58ms/step - loss: 2.2649 - acc
uracy: 0.1658 - val_loss: 1.9423 - val_accuracy: 0.4064
Epoch 2/10
625/625 [==============================] - 37s 59ms/step - loss: 2.2243 - acc
uracy: 0.2083 - val_loss: 1.9122 - val_accuracy: 0.4871
Epoch 3/10
625/625 [==============================] - 36s 58ms/step - loss: 2.1921 - acc
uracy: 0.2351 - val_loss: 1.7741 - val_accuracy: 0.5253
Epoch 4/10
625/625 [==============================] - 36s 57ms/step - loss: 2.1545 - acc
uracy: 0.2605 - val_loss: 1.6988 - val_accuracy: 0.5519
Epoch 5/10
625/625 [==============================] - 36s 57ms/step - loss: 2.1048 - acc
uracy: 0.2829 - val_loss: 1.6496 - val_accuracy: 0.5528
Epoch 6/10
625/625 [==============================] - 36s 58ms/step - loss: 2.0271 - acc
uracy: 0.3093 - val_loss: 1.6169 - val_accuracy: 0.5406
Epoch 7/10
625/625 [==============================] - 37s 58ms/step - loss: 1.9108 - acc
uracy: 0.3506 - val_loss: 1.7004 - val_accuracy: 0.4798
Epoch 8/10
625/625 [==============================] - 36s 57ms/step - loss: 1.7536 - acc
uracy: 0.4049 - val_loss: 1.7061 - val_accuracy: 0.4494
Epoch 9/10
625/625 [==============================] - 37s 59ms/step - loss: 1.5667 - acc
uracy: 0.4678 - val_loss: 1.9327 - val_accuracy: 0.3841
Epoch 10/10
625/625 [==============================] - 36s 58ms/step - loss: 1.3602 - acc
uracy: 0.5368 - val_loss: 2.0940 - val_accuracy: 0.3525
Runtime of the model 1's evaluation is 364 seconds
```

As the above CNN model shows, the model training process has quite long computational time and it acheives the maximum validation accuracy at middle epochs, then decrease when learning models' accuracy keeps increasing. This may cause overfitting problems when we introduce new images that our computers have never seen. So here we built another simpler CNN structure and it has better results with less running time.

In [8]:

```
#Model1 Another Simpler CNN Structure

#Set the 10001st -50000th pic as training images(noisy labels)
#Set the 1st-10000th pic as validation images since it has clean labels(the co
rrect answer)
test_x = np.empty((n_noisy,32,32,3))
test_y = np.empty(n_noisy)
valid_x = np.empty((n_clean_noisy,32,32,3))
valid_y = np.empty(n_clean_noisy)

for i in range(n_noisy):
    img_fn = f'../data/images/{10000+i+1:05d}.png'
    test_x[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)
    test_y[i] = noisy_labels[10000+i]

for i in range(n_clean_noisy):
    img_fn = f'../data/images/{i+1:05d}.png'
    valid_x[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)
    valid_y[i]= clean_labels[i]

#test_y = to_categorical(test_y)
#valid_y = to_categorical(valid_y)

test_x = test_x.astype("float32")
test_x = test_x/255

valid_x = valid_x.astype("float32")
valid_x = valid_x/255

#Build CNN model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3
)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))

model.compile(optimizer='adam',
              loss=tensorflow.keras.losses.SparseCategoricalCrossentropy(from_
logits=True),
              metrics=['accuracy'])

start = time.time()
history = model.fit(test_x, test_y, epochs=10,
                    validation_data=(valid_x, valid_y))
end = time.time()
print("Runtime of the model 1's evaluation is %d seconds" % (end - start))

#Overall validation results is 0.4479 much better than Baseline Model
```

```
Epoch 1/10
1250/1250 [==============================] - 23s 18ms/step - loss: 2.2798 - a
ccuracy: 0.1383 - val_loss: 2.0983 - val_accuracy: 0.2825
Epoch 2/10
1250/1250 [==============================] - 22s 18ms/step - loss: 2.2494 - a
ccuracy: 0.1772 - val_loss: 1.9291 - val_accuracy: 0.4051
Epoch 3/10
1250/1250 [==============================] - 23s 18ms/step - loss: 2.2279 - a
ccuracy: 0.2004 - val_loss: 1.9410 - val_accuracy: 0.4387
Epoch 4/10
1250/1250 [==============================] - 22s 18ms/step - loss: 2.2074 - a
ccuracy: 0.2181 - val_loss: 1.8337 - val_accuracy: 0.4594
Epoch 5/10
1250/1250 [==============================] - 22s 18ms/step - loss: 2.1903 - a
ccuracy: 0.2297 - val_loss: 1.8775 - val_accuracy: 0.4618
Epoch 6/10
1250/1250 [==============================] - 22s 18ms/step - loss: 2.1700 - a
ccuracy: 0.2445 - val_loss: 1.7764 - val_accuracy: 0.4980
Epoch 7/10
1250/1250 [==============================] - 22s 17ms/step - loss: 2.1472 - a
ccuracy: 0.2542 - val_loss: 1.7493 - val_accuracy: 0.4878
Epoch 8/10
1250/1250 [==============================] - 23s 18ms/step - loss: 2.1179 - a
ccuracy: 0.2691 - val_loss: 1.8079 - val_accuracy: 0.4810
Epoch 9/10
1250/1250 [==============================] - 22s 18ms/step - loss: 2.0840 - a
ccuracy: 0.2813 - val_loss: 1.7351 - val_accuracy: 0.4751
Epoch 10/10
1250/1250 [==============================] - 23s 18ms/step - loss: 2.0410 - a
ccuracy: 0.2971 - val_loss: 1.8071 - val_accuracy: 0.4454
Runtime of the model 1's evaluation is 224 seconds
```

As the above result indicates, it has more stable prediction result so we regard it as our first model.

In [16]:
```python
def model_I(image):
    '''
    This function should takes in the image of dimension 32*32*3 as input and
 returns a label prediction
    '''
    # write your code here..
    image = image.reshape(1,32,32,3)
    image= image.astype("float32")
    image= image/255

    label= model.predict(image)
    label = np.argmax(np.round(label), axis=1)
    return label
```

2.3. Model II

In [10]:
```
# [ADD WEAKLY SUPERVISED LEARNING FEATURE TO MODEL I]

# write your code here...

def model_II(image):
    '''
    This function should takes in the image of dimension 32*32*3 as input and
 returns a label prediction
    '''
    # write your code here...
```

3. Evaluation

For assessment, we will evaluate your final model on a hidden test dataset with clean labels by the `evaluation` function defined as follows. Although you will not have the access to the test set, the function would be useful for the model developments. For example, you can split the small training set, using one portion for weakly supervised learning and the other for validation purpose.

In [11]:
```
# [DO NOT MODIFY THIS CELL]
def evaluation(model, test_labels, test_imgs):
    y_true = test_labels
    y_pred = []
    for image in test_imgs:
        y_pred.append(model(image))
    print(classification_report(y_true, y_pred))
```

In [ ]:
```
# [DO NOT MODIFY THIS CELL]
# This is the code for evaluating the prediction performance on a testset
# You will get an error if running this cell, as you do not have the testset
# Nonetheless, you can create your own validation set to run the evlauation
n_test = 10000
test_labels = np.genfromtxt('../data/test_labels.csv', delimiter=',', dtype="i
nt8")
test_imgs = np.empty((n_test,32,32,3))
for i in range(n_test):
    img_fn = f'../data/test_images/test{i+1:05d}.png'
    test_imgs[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)
evaluation(baseline_model, test_labels, test_imgs)
```

In [13]:
```
n_test = 10000
test_labels = np.genfromtxt('../data/clean_labels.csv', delimiter=',', dtype=
"int8")
test_imgs = np.empty((n_test,32,32,3))
for i in range(n_test):
    img_fn = f'../data/images/{i+1:05d}.png'
    test_imgs[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)

import time
start = time.time()
evaluation(baseline_model, test_labels, test_imgs)
end = time.time()
print("Runtime of the baseline model's evaluation is %d seconds" % (end - star
t))
```

```
              precision    recall  f1-score   support

           0       0.32      0.43      0.37      1005
           1       0.18      0.29      0.22       974
           2       0.22      0.04      0.07      1032
           3       0.19      0.12      0.14      1016
           4       0.24      0.48      0.32       999
           5       0.22      0.13      0.16       937
           6       0.26      0.35      0.30      1030
           7       0.29      0.04      0.07      1001
           8       0.28      0.43      0.34      1025
           9       0.19      0.11      0.14       981

    accuracy                           0.24     10000
   macro avg       0.24      0.24      0.21     10000
weighted avg       0.24      0.24      0.21     10000

Runtime of the baseline model's evaluation is 2 seconds
```

The overall accuracy is 0.24, which is better than random guess (which should have a accuracy around 0.10). For the project, you should try to improve the performance by the following strategies:

- Consider a better choice of model architectures, hyperparameters, or training scheme for the predictive model;
- Use both `clean_noisy_trainset` and `noisy_trainset` for model training via **weakly supervised learning** methods. One possible solution is to train a "label-correction" model using the former, correct the labels in the latter, and train the final predictive model using the corrected dataset.
- Apply techniques such as k-fold cross validation to avoid overfitting;
- Any other reasonable strategies.

In [17]:
```
import time
start = time.time()
evaluation(model_I, test_labels, test_imgs)
end = time.time()
print("Runtime of the model 1's evaluation is %.4f minutes" % ((end - start)/6
0))
```

```
              precision    recall  f1-score   support

           0       0.29      0.61      0.39      1005
           1       0.51      0.50      0.51       974
           2       0.28      0.46      0.35      1032
           3       0.29      0.26      0.28      1016
           4       0.29      0.29      0.29       999
           5       0.36      0.22      0.27       937
           6       0.68      0.23      0.34      1030
           7       0.60      0.35      0.44      1001
           8       0.61      0.50      0.55      1025
           9       0.51      0.51      0.51       981

    accuracy                           0.39     10000
   macro avg       0.44      0.39      0.39     10000
weighted avg       0.44      0.39      0.39     10000

Runtime of the model 1's evaluation is 3.8611 minutes
```

4. Save Label Predictions

In [ ]:
```
imgs_test = np.empty((10000,32,32,3))
for i in range(10000):
    img_fn = f'../data/images/{i+1:05d}.png'
    imgs_test[i,:,:,:]=cv2.cvtColor(cv2.imread(img_fn),cv2.COLOR_BGR2RGB)
```

In [ ]:
```
baseline_pred = []
for i in imgs_test:
    baseline_pred.append(baseline_model(i)[0])

model1_pred = model_I(imgs_test)
model1_pred[:10]
```

In [ ]:
```
#Save csv file
import pandas as pd

results = {
    "Baseline": baseline_pred,
    "Model I": model1_pred
}

results_df = pd.DataFrame(results)
results_df.to_csv("../output/label_prediction.csv")
```

In [ ]: