Homework 4: Build a CNN for Image Recognition

Name: Tyler Bryk

1. Data Preparation

1.1. Load data

In [1]:
```
from keras.datasets import cifar10
import numpy

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

print('shape of y_train: ' + str(y_train.shape))
print('shape of x_train: ' + str(x_train.shape))
print('shape of x_test: ' + str(x_test.shape))
print('shape of y_test: ' + str(y_test.shape))
print('number of classes: ' + str(numpy.max(y_train) - numpy.min(y_train) + 1))
```

```
Using TensorFlow backend.
shape of x_train: (50000, 32, 32, 3)
shape of y_train: (50000, 1)
shape of x_test: (10000, 32, 32, 3)
shape of y_test: (10000, 1)
number of classes: 10
```

1.2. One-hot encode the labels

In the input, a label is a scalar in $\{0, 1, \cdots, 9\}$. One-hot encode transform such a scalar to a 10-dim vector. E.g., a scalar `y_train[j]=3` is transformed to the vector `y_train_vec[j]=[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`.

1. Define a function `to_one_hot` that transforms an $m \times 1$ array to a $m \times 10$ matrix.

2. Apply the function to `y_train` and `y_test`.

In [2]:
```
def to_one_hot(y, num_classes=10):
    results = numpy.zeros((len(y), num_class))
    for i, label in enumerate(y):
        results[i, label] = 1.
    return results

y_train_vec = to_one_hot(y_train)
y_test_vec = to_one_hot(y_test)

print('Shape of y_train_vec: ' + str(y_train_vec.shape))
print('Shape of y_test_vec: ' + str(y_test_vec.shape))

print(y_train[0])
print(y_train_vec[0])
```

```
Shape of y_train_vec: (50000, 10)
Shape of y_test_vec: (10000, 10)
[6]
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

In [3]:
```
# Convert Image Data to Float32
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# Standardize Data with Z-scores
mean = numpy.mean(x_train, axis=(0,1,2,3))
std = numpy.std(x_train, axis=(0,1,2,3))
x_train = (x_train - mean) / (std + 1e-7)
x_test = (x_test - mean) / (std + 1e-7)
```

1.3. Randomly partition the training set to training and validation sets

Randomly partition the 50K training samples to 2 sets:

- a training set containing 40K samples
- a validation set containing 10K samples

In [4]:
```
rand_indices = numpy.random.permutation(50000)
train_indices = rand_indices[0:40000]
valid_indices = rand_indices[40000:50000]

x_val = x_train[valid_indices, :]
y_val = y_train_vec[valid_indices, :]

x_tr = x_train[train_indices, :]
y_tr = y_train_vec[train_indices, :]

print('Shape of x_tr: ' + str(x_tr.shape))
print('Shape of y_tr: ' + str(y_tr.shape))
print('Shape of x_val: ' + str(x_val.shape))
print('Shape of y_val: ' + str(y_val.shape))
```

```
Shape of x_tr: (40000, 32, 32, 3)
Shape of y_tr: (40000, 10)
Shape of x_val: (10000, 32, 32, 3)
Shape of y_val: (10000, 10)
```

2. Build a CNN and tune its hyper-parameters

1. Build a convolutional neural network model
2. Use the validation data to tune the hyper-parameters (e.g., network structure, and optimization algorithm)
   - Do NOT use test data for hyper-parameter tuning!!
3. Try to achieve a validation accuracy as high as possible.

In [5]:
```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, Activation
from keras.layers.normalization import BatchNormalization
from keras.models import Sequential
from keras.regularizers import l2

model = Sequential()
model.add(Conv2D(32, (3,3), padding='same', kernel_regularizer=l2(1e-4), activation="relu", input_shape=(32,32,3)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3,3), padding='same', kernel_regularizer=l2(1e-4), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Dropout(0.2))
model.add(Conv2D(64, (3,3), padding='same', kernel_regularizer=l2(1e-4), activation="relu"))
model.add(BatchNormalization())
model.add(Conv2D(64, (3,3), padding='same', kernel_regularizer=l2(1e-4), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Dropout(0.3))
model.add(Conv2D(128, (3,3), padding='same', kernel_regularizer=l2(1e-4), activation="relu"))
model.add(BatchNormalization())
model.add(Conv2D(128, (3,3), padding='same', kernel_regularizer=l2(1e-4), activation="relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Flatten())
model.add(Dense(512))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.4))
model.add(Dense(10))
model.add(BatchNormalization())
model.add(Activation('softmax'))
```

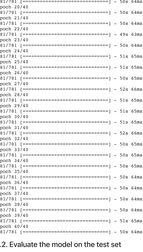
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 32)        128
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 16, 16, 64)        36928
_________________________________________________________________
batch_normalization_4 (Batch (None, 16, 16, 64)        256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
batch_normalization_5 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 8, 8, 128)         147584
_________________________________________________________________
batch_normalization_6 (Batch (None, 8, 8, 128)         512
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1049088
_________________________________________________________________
batch_normalization_7 (Batch (None, 512)               2048
_________________________________________________________________
activation_1 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5130
_________________________________________________________________
batch_normalization_8 (Batch (None, 10)                40
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 1,345,166
Trainable params: 1,343,166
Non-trainable params: 1,940
_________________________________________________________________
```

In [6]:
```
from keras import optimizers
from keras.preprocessing.image import ImageDataGenerator

gen = ImageDataGenerator(rotation_range=15,
                         width_shift_range=0.1,
                         height_shift_range=0.1,
                         shear_range=0.2,
                         zoom_range=0.1,
                         horizontal_flip=True)

gen.fit(x_tr)
train_generator = gen.flow(x_tr, y_tr, batch_size=64)
test_generator = ImageDataGenerator().flow(x_val, y_val, batch_size=64)

model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=0.001, decay=1e-6),
              metrics=['accuracy'])

history = model.fit_generator(train_generator, steps_per_epoch=40000/64, epochs=40, validation_data=test_generator, validation_steps=10000/64)
```

```
Epoch 1/40
625/625 [==============================] - 42s 67ms/step - loss: 1.4541 - accuracy: 0.5121 - val_loss: 0.9724 - val_accuracy: 0.6383
Epoch 2/40
625/625 [==============================] - 42s 67ms/step - loss: 1.1017 - accuracy: 0.6489 - val_loss: 0.9180 - val_accuracy: 0.7078
Epoch 3/40
625/625 [==============================] - 42s 68ms/step - loss: 0.9648 - accuracy: 0.6999 - val_loss: 0.8764 - val_accuracy: 0.7241
Epoch 4/40
625/625 [==============================] - 41s 66ms/step - loss: 0.8882 - accuracy: 0.7270 - val_loss: 0.7428 - val_accuracy: 0.7729
Epoch 5/40
625/625 [==============================] - 42s 67ms/step - loss: 0.8387 - accuracy: 0.7444 - val_loss: 0.8329 - val_accuracy: 0.7653
Epoch 6/40
625/625 [==============================] - 42s 67ms/step - loss: 0.8049 - accuracy: 0.7560 - val_loss: 0.8565 - val_accuracy: 0.7665
Epoch 7/40
625/625 [==============================] - 42s 67ms/step - loss: 0.7808 - accuracy: 0.7675 - val_loss: 0.7876 - val_accuracy: 0.7876
Epoch 8/40
625/625 [==============================] - 42s 67ms/step - loss: 0.7617 - accuracy: 0.7760 - val_loss: 0.7773 - val_accuracy: 0.8074
Epoch 9/40
625/625 [==============================] - 42s 66ms/step - loss: 0.7474 - accuracy: 0.7815 - val_loss: 0.8030 - val_accuracy: 0.8002
Epoch 10/40
625/625 [==============================] - 42s 66ms/step - loss: 0.7249 - accuracy: 0.7909 - val_loss: 0.5733 - val_accuracy: 0.8044
Epoch 11/40
625/625 [==============================] - 42s 66ms/step - loss: 0.7145 - accuracy: 0.7969 - val_loss: 0.9419 - val_accuracy: 0.8152
Epoch 12/40
625/625 [==============================] - 42s 66ms/step - loss: 0.7036 - accuracy: 0.7997 - val_loss: 0.6393 - val_accuracy: 0.8076
Epoch 13/40
625/625 [==============================] - 42s 67ms/step - loss: 0.6994 - accuracy: 0.8030 - val_loss: 0.5725 - val_accuracy: 0.8365
Epoch 14/40
625/625 [==============================] - 42s 66ms/step - loss: 0.6889 - accuracy: 0.8055 - val_loss: 0.6632 - val_accuracy: 0.8008
Epoch 15/40
625/625 [==============================] - 42s 66ms/step - loss: 0.6746 - accuracy: 0.8124 - val_loss: 0.5092 - val_accuracy: 0.8456
Epoch 16/40
625/625 [==============================] - 42s 66ms/step - loss: 0.6696 - accuracy: 0.8156 - val_loss: 0.5663 - val_accuracy: 0.8384
Epoch 17/40
625/625 [==============================] - 42s 66ms/step - loss: 0.6706 - accuracy: 0.8146 - val_loss: 0.6523 - val_accuracy: 0.8565
Epoch 18/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6605 - accuracy: 0.8184 - val_loss: 0.6145 - val_accuracy: 0.8458
Epoch 19/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6567 - accuracy: 0.8209 - val_loss: 0.6421 - val_accuracy: 0.8353
Epoch 20/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6596 - accuracy: 0.8223 - val_loss: 0.3942 - val_accuracy: 0.8530
Epoch 21/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6466 - accuracy: 0.8251 - val_loss: 0.6067 - val_accuracy: 0.8385
Epoch 22/40
625/625 [==============================] - 42s 66ms/step - loss: 0.6471 - accuracy: 0.8229 - val_loss: 0.5245 - val_accuracy: 0.8259
Epoch 23/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6419 - accuracy: 0.8208 - val_loss: 0.8233 - val_accuracy: 0.8298
Epoch 24/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6336 - accuracy: 0.8307 - val_loss: 0.4700 - val_accuracy: 0.8462
Epoch 25/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6388 - accuracy: 0.8276 - val_loss: 0.5400 - val_accuracy: 0.8606
Epoch 26/40
625/625 [==============================] - 42s 67ms/step - loss: 0.6337 - accuracy: 0.8329 - val_loss: 0.4744 - val_accuracy: 0.8364
Epoch 27/40
625/625 [==============================] - 42s 67ms/step - loss: 0.6286 - accuracy: 0.8335 - val_loss: 0.5554 - val_accuracy: 0.8480
Epoch 28/40
625/625 [==============================] - 42s 66ms/step - loss: 0.6285 - accuracy: 0.8339 - val_loss: 0.5141 - val_accuracy: 0.8493
Epoch 29/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6237 - accuracy: 0.8366 - val_loss: 0.8269 - val_accuracy: 0.8449
Epoch 30/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6193 - accuracy: 0.8379 - val_loss: 0.6516 - val_accuracy: 0.8445
Epoch 31/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6176 - accuracy: 0.8383 - val_loss: 0.5224 - val_accuracy: 0.8589
Epoch 32/40
625/625 [==============================] - 42s 67ms/step - loss: 0.6128 - accuracy: 0.8399 - val_loss: 0.9207 - val_accuracy: 0.8479
Epoch 33/40
625/625 [==============================] - 42s 67ms/step - loss: 0.6120 - accuracy: 0.8411 - val_loss: 0.4678 - val_accuracy: 0.8698
Epoch 34/40
625/625 [==============================] - 42s 67ms/step - loss: 0.6134 - accuracy: 0.8412 - val_loss: 0.5255 - val_accuracy: 0.8681
Epoch 35/40
625/625 [==============================] - 42s 67ms/step - loss: 0.6062 - accuracy: 0.8421 - val_loss: 0.4633 - val_accuracy: 0.8624
Epoch 36/40
625/625 [==============================] - 42s 67ms/step - loss: 0.6113 - accuracy: 0.8414 - val_loss: 0.4884 - val_accuracy: 0.8503
Epoch 37/40
625/625 [==============================] - 41s 66ms/step - loss: 0.6022 - accuracy: 0.8436 - val_loss: 0.5688 - val_accuracy: 0.8629
Epoch 38/40
625/625 [==============================] - 42s 67ms/step - loss: 0.5996 - accuracy: 0.8451 - val_loss: 0.5783 - val_accuracy: 0.8639
Epoch 39/40
625/625 [==============================] - 41s 66ms/step - loss: 0.5999 - accuracy: 0.8450 - val_loss: 0.3326 - val_accuracy: 0.8593
Epoch 40/40
625/625 [==============================] - 41s 66ms/step - loss: 0.5995 - accuracy: 0.8464 - val_loss: 0.5971 - val_accuracy: 0.8702
```

In [7]:
```
import matplotlib.pyplot as plt

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
epochs = range(len(acc))

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



3. Train (again) and evaluate the model

- To this end, you have found the "best" hyper-parameters.
- Now, fix the hyper-parameters and train the network on the entire training set (all the 50K training samples)
- Evaluate your model on the test set.

3.1. Train the model on the entire training set

In [8]:
```
gen = ImageDataGenerator(rotation_range=15,
                         width_shift_range=0.1,
                         height_shift_range=0.1,
                         shear_range=0.2,
                         zoom_range=0.1,
                         horizontal_flip=True)

gen.fit(x_train)
train_generator = gen.flow(x_train, y_train_vec, batch_size=64)

model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=0.001, decay=1e-6),
              metrics=['accuracy'])

history = model.fit_generator(train_generator, steps_per_epoch=50000/64, epochs=40)
```

```
Epoch 1/40
781/781 [==============================] - 52s 66ms/step - loss: 0.6239 - accuracy: 0.8400
Epoch 2/40
781/781 [==============================] - 50s 64ms/step - loss: 0.6187 - accuracy: 0.8405
Epoch 3/40
781/781 [==============================] - 50s 64ms/step - loss: 0.6162 - accuracy: 0.8402
Epoch 4/40
781/781 [==============================] - 50s 64ms/step - loss: 0.6115 - accuracy: 0.8442
Epoch 5/40
781/781 [==============================] - 51s 65ms/step - loss: 0.6103 - accuracy: 0.8436
Epoch 6/40
781/781 [==============================] - 51s 66ms/step - loss: 0.6114 - accuracy: 0.8461
Epoch 7/40
781/781 [==============================] - 49s 63ms/step - loss: 0.6091 - accuracy: 0.8450
Epoch 8/40
781/781 [==============================] - 50s 64ms/step - loss: 0.6091 - accuracy: 0.8459
Epoch 9/40
781/781 [==============================] - 50s 65ms/step - loss: 0.5970 - accuracy: 0.8490
Epoch 10/40
781/781 [==============================] - 50s 64ms/step - loss: 0.6001 - accuracy: 0.8497
Epoch 11/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5925 - accuracy: 0.8507
Epoch 12/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5953 - accuracy: 0.8490
Epoch 13/40
781/781 [==============================] - 50s 64ms/step - loss: 0.6024 - accuracy: 0.8488
Epoch 14/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5960 - accuracy: 0.8526
Epoch 15/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5881 - accuracy: 0.8502
Epoch 16/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5884 - accuracy: 0.8532
Epoch 17/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5898 - accuracy: 0.8515
Epoch 18/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5878 - accuracy: 0.8525
Epoch 19/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5820 - accuracy: 0.8562
Epoch 20/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5816 - accuracy: 0.8553
Epoch 21/40
781/781 [==============================] - 49s 63ms/step - loss: 0.5809 - accuracy: 0.8536
Epoch 22/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5756 - accuracy: 0.8572
Epoch 23/40
781/781 [==============================] - 51s 65ms/step - loss: 0.5781 - accuracy: 0.8570
Epoch 24/40
781/781 [==============================] - 51s 65ms/step - loss: 0.5774 - accuracy: 0.8577
Epoch 25/40
781/781 [==============================] - 50s 65ms/step - loss: 0.5763 - accuracy: 0.8565
Epoch 26/40
781/781 [==============================] - 51s 65ms/step - loss: 0.5779 - accuracy: 0.8545
Epoch 27/40
781/781 [==============================] - 51s 65ms/step - loss: 0.5732 - accuracy: 0.8571
Epoch 28/40
781/781 [==============================] - 51s 65ms/step - loss: 0.5716 - accuracy: 0.8584
Epoch 29/40
781/781 [==============================] - 51s 65ms/step - loss: 0.5717 - accuracy: 0.8595
Epoch 30/40
781/781 [==============================] - 52s 66ms/step - loss: 0.5751 - accuracy: 0.8562
Epoch 31/40
781/781 [==============================] - 50s 65ms/step - loss: 0.5668 - accuracy: 0.8597
Epoch 32/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5725 - accuracy: 0.8584
Epoch 33/40
781/781 [==============================] - 50s 65ms/step - loss: 0.5654 - accuracy: 0.8603
Epoch 34/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5666 - accuracy: 0.8605
Epoch 35/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5617 - accuracy: 0.8594
Epoch 36/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5649 - accuracy: 0.8613
Epoch 37/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5677 - accuracy: 0.8599
Epoch 38/40
781/781 [==============================] - 51s 65ms/step - loss: 0.5659 - accuracy: 0.8599
Epoch 39/40
781/781 [==============================] - 50s 64ms/step - loss: 0.5618 - accuracy: 0.8635
```

3.2. Evaluate the model on the test set

DO NOT used the test set until now. Make sure that your model parameters and hyper-parameters are independent of the test set.

In [9]:
```
loss_and_acc = model.evaluate(x_test, y_test_vec)
print('loss = ' + str(loss_and_acc[0]))
print('accuracy = ' + str(loss_and_acc[1]))
```

```
10000/10000 [==============================] - 3s 337us/step
loss = 0.5268663731117381
accuracy = 0.8659999966621399
```