# Programming Assignment: Build a CNN for image recognition.

Name: [Your-Name?]

## 0. You will do the following:

1. Read, complete, and run the code.
2. **Make substantial improvements** to maximize the accuracy.
3. Convert the .IPYNB file to .HTML file.
    - The HTML file must contain the code and the output after execution.
    - Missing **the output after execution** will not be graded.

1. Upload the .HTML file to your Google Drive, Dropbox, or Github repo. (If you submit the file to Google Drive or Dropbox, you must make the file "open-access". The delay caused by "deny of access" may result in late penalty.)
2. On Canvas, submit the Google Drive/Dropbox/Github link to the HTML file.

## Requirements:

1. You can use whatever CNN architecture, including VGG, Inception, and ResNet. However, you must build the networks layer by layer. You must NOT import the archetectures from `keras.applications`.
2. Make sure `BatchNormalization` is between a `Conv` / `Dense` layer and an `activation` layer.
3. If you want to regularize a `Conv` / `Dense` layer, you should place a `Dropout` layer **before** the `Conv` / `Dense` layer.
4. An accuracy above 70% is considered reasonable. An accuracy above 80% is considered good. Without data augmentation, achieving 80% accuracy is difficult.

## Google Colab

- If you do not have GPU, the training of a CNN can be slow. Google Colab is a good option.
- Keep in mind that you must download it as an IPYNB file and then use IPython Notebook to convert it to HTML.
- Also keep in mind that the IPYNB and HTML files must contain the outputs. (Otherwise, the instructor will not be able to know the correctness and performance.) Do the followings to keep the outputs.
- In Colab, go to `Runtime` --> `Change runtime type` --> Do NOT check `Omit code cell output when saving this notebook`. In this way, the downloaded IPYNB file contains the outputs.

## 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 x_train: ' + str(x_train.shape))
print('shape of y_train: ' + str(y_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))
```

```
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 $n \times 1$ array to a $n \times 10$ matrix.
2. Apply the function to `y_train` and `y_test`.

In [2]:
```
def to_one_hot(y, num_class=10):
     return numpy.eye(num_class)[y.reshape(-1)]

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.]
```

#### Remark: the outputs should be

- 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.]

### 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 [3]:
```
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.

### Remark:

The following CNN is just an example. You are supposed to make **substantial improvements** such as:

- Add more layers.
- Use regularizations, e.g., dropout.
- Use batch normalization.

In [7]:
```
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization, Activation
from keras.models import Sequential

model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same', input_shape=(32, 32, 3)))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Conv2D(128, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(128, (3, 3), padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(512))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

model.summary()
# seperated the activation and conv2D layers with batch nomralization
# added a second conv2d layer with same number of units before each round of max pooling
# added two conv2D layers with 128 units
# Added dropout layers to end of conv2D layers and each dense layer
# Added anoither dense layer with 512 units and dropout layers to the fully connected layers.
```

**Model: "sequential_1"**

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_6 (Conv2D) | (None, 32, 32, 32) | 896 |
| batch_normalization_6 (BatchNormalization) | (None, 32, 32, 32) | 128 |
| activation_6 (Activation) | (None, 32, 32, 32) | 0 |
| conv2d_7 (Conv2D) | (None, 32, 32, 32) | 9,248 |
| batch_normalization_7 (BatchNormalization) | (None, 32, 32, 32) | 128 |
| activation_7 (Activation) | (None, 32, 32, 32) | 0 |
| max_pooling2d_3 (MaxPooling2D) | (None, 16, 16, 32) | 0 |
| dropout_3 (Dropout) | (None, 16, 16, 32) | 0 |
| conv2d_8 (Conv2D) | (None, 16, 16, 64) | 18,496 |
| batch_normalization_8 (BatchNormalization) | (None, 16, 16, 64) | 256 |
| activation_8 (Activation) | (None, 16, 16, 64) | 0 |
| conv2d_9 (Conv2D) | (None, 16, 16, 64) | 36,928 |
| batch_normalization_9 (BatchNormalization) | (None, 16, 16, 64) | 256 |
| activation_9 (Activation) | (None, 16, 16, 64) | 0 |
| max_pooling2d_4 (MaxPooling2D) | (None, 8, 8, 64) | 0 |
| dropout_4 (Dropout) | (None, 8, 8, 64) | 0 |
| conv2d_10 (Conv2D) | (None, 8, 8, 128) | 73,856 |
| batch_normalization_10 (BatchNormalization) | (None, 8, 8, 128) | 512 |
| activation_10 (Activation) | (None, 8, 8, 128) | 0 |
| conv2d_11 (Conv2D) | (None, 8, 8, 128) | 147,584 |
| batch_normalization_11 (BatchNormalization) | (None, 8, 8, 128) | 512 |
| activation_11 (Activation) | (None, 8, 8, 128) | 0 |
| max_pooling2d_5 (MaxPooling2D) | (None, 4, 4, 128) | 0 |
| dropout_5 (Dropout) | (None, 4, 4, 128) | 0 |
| flatten (Flatten) | (None, 2048) | 0 |
| dropout_6 (Dropout) | (None, 2048) | 0 |
| dense (Dense) | (None, 512) | 1,049,088 |
| batch_normalization_12 (BatchNormalization) | (None, 512) | 2,048 |
| activation_12 (Activation) | (None, 512) | 0 |
| dropout_7 (Dropout) | (None, 512) | 0 |
| dense_1 (Dense) | (None, 10) | 5,130 |

**Total params:** 1,345,066 (5.13 MB)
**Trainable params:** 1,343,146 (5.12 MB)
**Non-trainable params:** 1,920 (7.50 KB)

In [8]:
```
from keras import optimizers
from keras.optimizers import Adam

learning_rate = 1E-5 # to be tuned!

model.compile(loss='categorical_crossentropy',
              optimizer= Adam(learning_rate=0.001),
              metrics=['acc'])

# switched optimizer from RMS.prop to Adam and adjusted learning rate to be higher
```

In [9]:
```
history = model.fit(x_tr, y_tr, batch_size=32, epochs=10, validation_data=(x_val, y_val))
```

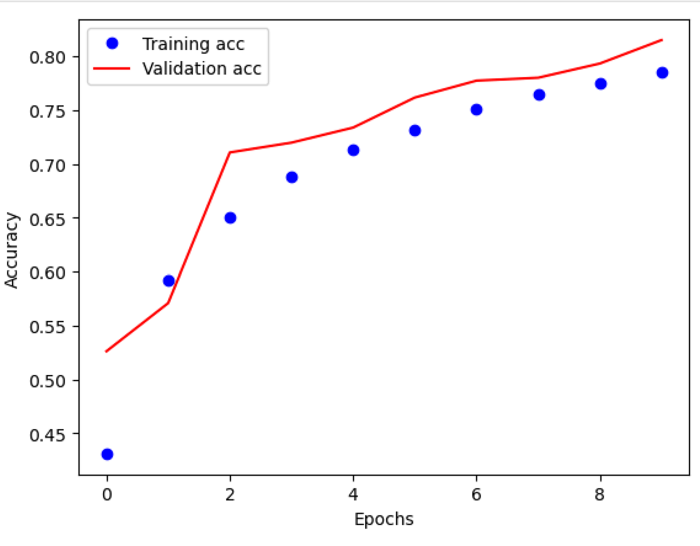
```
Epoch 1/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 145s 111ms/step - acc: 0.3434 - loss: 1.9373 - val_acc: 0.5262 - val_loss: 1.3862
Epoch 2/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 131s 105ms/step - acc: 0.5715 - loss: 1.1996 - val_acc: 0.5709 - val_loss: 1.2582
Epoch 3/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 100s 80ms/step - acc: 0.6410 - loss: 1.0093 - val_acc: 0.7106 - val_loss: 0.8227
Epoch 4/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 102s 81ms/step - acc: 0.6834 - loss: 0.8941 - val_acc: 0.7197 - val_loss: 0.7873
Epoch 5/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 117s 94ms/step - acc: 0.7094 - loss: 0.8208 - val_acc: 0.7337 - val_loss: 0.7640
Epoch 6/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 118s 94ms/step - acc: 0.7336 - loss: 0.7599 - val_acc: 0.7616 - val_loss: 0.6762
Epoch 7/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 120s 96ms/step - acc: 0.7502 - loss: 0.7163 - val_acc: 0.7773 - val_loss: 0.6345
Epoch 8/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 117s 93ms/step - acc: 0.7650 - loss: 0.6817 - val_acc: 0.7800 - val_loss: 0.6250
Epoch 9/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 119s 95ms/step - acc: 0.7763 - loss: 0.6425 - val_acc: 0.7932 - val_loss: 0.5986
Epoch 10/10
1250/1250 ━━━━━━━━━━━━━━━━━━━━ 117s 94ms/step - acc: 0.7869 - loss: 0.6135 - val_acc: 0.8150 - val_loss: 0.5329
```

In [10]:
```
import matplotlib.pyplot as plt
%matplotlib inline

acc = history.history['acc']
val_acc = history.history['val_acc']

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

Why? Previously, you used 40K samples for training; you wasted 10K samples for the sake of hyper-parameter tuning. Now you already know the hyper-parameters, so why not using all the 50K samples for training?

In [11]:
```
model.compile(loss='categorical_crossentropy',
              optimizer= Adam(learning_rate=0.001),
              metrics=['acc'])
```

In [12]:
```
#<Train your model on the entire training set (50K samples)>
#<Use (x_train, y_train_vec) instead of (x_tr, y_tr)>
#<Do NOT use the validation_data option (because now you do not have validation data)>

history = model.fit(x_train, y_train_vec, batch_size=32, epochs=10,)
```

```
Epoch 1/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 148s 90ms/step - acc: 0.7880 - loss: 0.6116
Epoch 2/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 139s 89ms/step - acc: 0.8014 - loss: 0.5760
Epoch 3/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 141s 90ms/step - acc: 0.8097 - loss: 0.5577
Epoch 4/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 138s 88ms/step - acc: 0.8162 - loss: 0.5337
Epoch 5/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 140s 90ms/step - acc: 0.8234 - loss: 0.5146
Epoch 6/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 142s 91ms/step - acc: 0.8297 - loss: 0.4840
Epoch 7/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 142s 91ms/step - acc: 0.8339 - loss: 0.4839
Epoch 8/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 145s 92ms/step - acc: 0.8365 - loss: 0.4680
Epoch 9/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 138s 88ms/step - acc: 0.8411 - loss: 0.4601
Epoch 10/10
1563/1563 ━━━━━━━━━━━━━━━━━━━━ 146s 94ms/step - acc: 0.8430 - loss: 0.4549
```

### 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 [13]:
```
loss_and_acc = model.evaluate(x_test, y_test_vec)
print('loss = ' + str(loss_and_acc[0]))
print('accuracy = ' + str(loss_and_acc[1]))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 19ms/step - acc: 0.8359 - loss: 0.4720
loss = 0.47645291686058044
accuracy = 0.8360000252723694
```

In [ ]: