A Simple End-to-end Image Classification using CNN

In this project, we will train a CNN to classify images from the `CIFAR-10` dataset (www.cs.toronto.edu/~kriz/cifar.html). `CIFAR-10` is an established CV dataset used for object recognition. It is a subset of the 80 Million Tiny Images dataset and consists of 60,000 (32 × 32) color images containing 1 of 10 object classes, with 6,000 images per class.

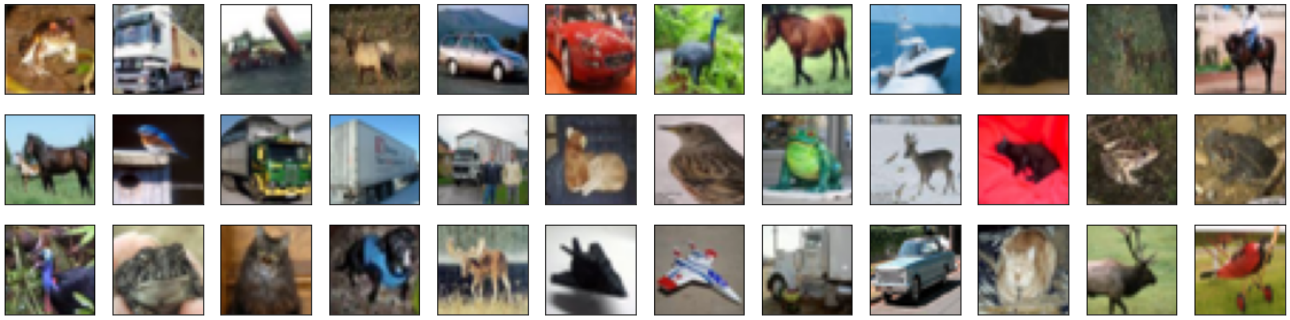
Step 1: Load the Dataset

The first step is to load the dataset into our train and test objects. Luckily, `Keras` provides the `CIFAR` dataset for us to load using the `load_data()` method. All we have to do is import `keras.datasets` and then load the data:

```
from tensorflow.keras.datasets import cifar10

# load the pre-shuffled train and test data
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Note that `Keras` splits the data into 50,000 training examples and 10,000 testing examples. We can then visualize the first 36 training images:

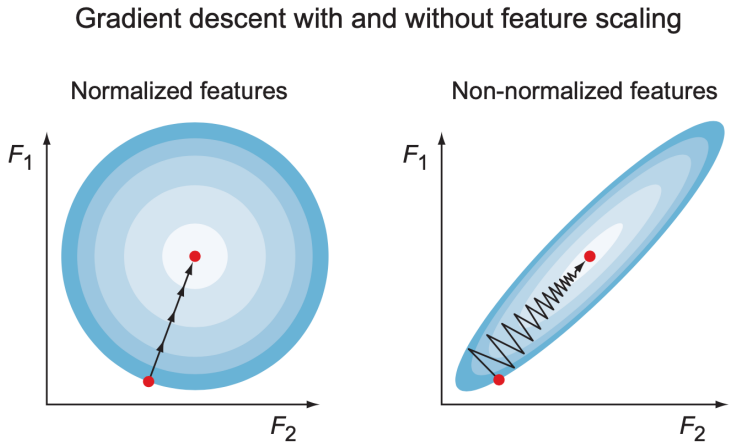
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
```



Step 2: Image Preprocessing

Based on your dataset and the problem you are solving, you will need to do **some data cleanup and preprocessing** to get it ready for your learning model. A cost function has the shape of a bowl, but it

can be an elongated bowl if the features have very different scales. Figure below shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).



Rescale the images

A typical image preprocessing involves **resizing** the images and **rescaling** the features. As the images all have the same size, we thus only rescale the input images as follows:

```
# rescale [0,255] --> [0,1]
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

Prepare the labels (one-hot encoding)

```
from tensorflow.keras.utils import to_categorical

# one-hot encode the labels
num_classes = len(np.unique(y_train))
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)
```

Split the dataset for training and validation

```
# break training set into training and validation sets
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]
```

Fun Time: how many training examples and validation examples we have after splitting (1) 5000 and 5000 (2) 10000 and 5000 (3) 5000 and 45000 (4) 45000 and 5000.

Step 3: Define the Model

You learned that the core building block of CNNs (and neural networks in general) is the layer. Most DL projects consist of stacking together simple layers that implement a form of data distillation. As you learned, the main CNN layers are convolution, pooling, fully connected, and activation functions.

Remark: How many convolutional layers should you create? How many pooling layers? It is very helpful to read about some of the most popular architectures (AlexNet, ResNet, Inception) and extract the key ideas leading to the design decisions. We will discuss the most popular CNN architectures later.

For now, let us try a smaller version of AlexNet and see how it performs with our dataset. Based on the results, we might add more layers. Our architecture will stack three convolutional layers and two fully connected (dense) layers as follows:

```
CNN: INPUT ⇒ CONV_1 ⇒ POOL_1 ⇒ CONV_2 ⇒ POOL_2 ⇒ CONV_3 ⇒ POOL_3 ⇒
DO ⇒ FC ⇒ DO ⇒ FC (softmax)
```

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten,
Dense, Dropout

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same',
activation='relu',
                 input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.3))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(10, activation='softmax'))
```

Step 4: Compile the Model

The last step before training our model is to define three more hyperparameters—a loss function, an optimizer, and metrics to monitor during training and testing:

- *Loss function*—How the network will be able to measure its performance on the training data.

- *Optimizer*—The mechanism that the network will use to optimize its parameters (weights and biases) to yield the minimum loss value. It is usually one of the variants of stochastic gradient descent.
- *Metrics*—List of metrics to be evaluated by the model during training and testing. Typically we use metrics=['accuracy'].

```
# compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
metrics=['accuracy'])
```

Step 5: Train the Model

We are now ready to train the network. In `Keras`, this is done via a call to the network's `.fit()` method (as in fitting the model to the training data):

```
from tensorflow.keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5',
verbose=1, save_best_only=True)

hist = model.fit(x_train, y_train, batch_size=32, epochs=100,
        validation_data=(x_valid, y_valid), callbacks=[checkpointer],
        verbose=2, shuffle=True)
```

When you run this cell, the training will start, and the verbose output shown in below will show one epoch at a time. Since 100 epochs of display do not fit on one page, the screenshot only shows the first 6 epochs.

```
Epoch 1/100

Epoch 00001: val_loss improved from inf to 1.36389, saving model to
model.weights.best.hdf5
1407/1407 - 21s - loss: 1.5851 - accuracy: 0.4246 - val_loss: 1.3639 -
val_accuracy: 0.5046 - 21s/epoch - 15ms/step
Epoch 2/100

Epoch 00002: val_loss improved from 1.36389 to 1.12162, saving model to
model.weights.best.hdf5
1407/1407 - 10s - loss: 1.2672 - accuracy: 0.5490 - val_loss: 1.1216 -
val_accuracy: 0.6088 - 10s/epoch - 7ms/step
Epoch 3/100

Epoch 00003: val_loss did not improve from 1.12162
1407/1407 - 10s - loss: 1.1598 - accuracy: 0.5907 - val_loss: 1.1614 -
val_accuracy: 0.5906 - 10s/epoch - 7ms/step
Epoch 4/100
```

```
Epoch 00004: val_loss improved from 1.12162 to 0.98136, saving model to
model.weights.best.hdf5
1407/1407 - 10s - loss: 1.0899 - accuracy: 0.6185 - val_loss: 0.9814 -
val_accuracy: 0.6578 - 10s/epoch - 7ms/step
Epoch 5/100

Epoch 00005: val_loss did not improve from 0.98136
1407/1407 - 10s - loss: 1.0517 - accuracy: 0.6339 - val_loss: 1.0071 -
val_accuracy: 0.6484 - 10s/epoch - 7ms/step
Epoch 6/100

Epoch 00006: val_loss did not improve from 0.98136
1407/1407 - 10s - loss: 1.0336 - accuracy: 0.6419 - val_loss: 1.2690 -
val_accuracy: 0.6032 - 10s/epoch - 7ms/step
```

Fun Time: the best model is at epoch (1) 3 (2) 4 (3) 5 (4) 6.

Fun Time: if we use the best model to predict new testing data, we expect to have the accuracy of (1) 0.60 (2) 0.62 (3) 0.64 (4) 0.66 (5) 0.68.

Step 6: Load the Best Model

Now that the training is complete, we can use the `Keras` method `load_weights()` to load into our model the weights that **yielded the best validation accuracy score**:

```
# load the weights that yielded the best validation accuracy
model.load_weights('model.weights.best.hdf5')
```

Step 7: Evaluate the Model

The last step is to evaluate our model and calculate the accuracy value as a percentage indicating how often our model correctly predicts the image classification:

```
# evaluate and print test accuracy
score = model.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', score[1])
```

When you run this cell, you will get an accuracy of about 66%. That is not bad but we can do a lot better. We revisit this project to apply these strategies and improve the accuracy to above 90%.

You can download the above code `CNN_Cifar10_Simple.ipynb` from the course website.