Near State-of-the-Art results at Object Recognition

In this project, we will learn to:

1.Import datasets from Keras

2.Use one-hot vectors for categorical labels

3.Addlayers to a Keras model

4.Load pre-trained weights

5.Make predictions using a trained Keras model

The dataset we will be using is the CIFAR-10 dataset, which consists of 60,000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

1. Loading the Data

Let's dive right in! In these first few cells, we will import necessary packages, load the dataset, and plot some example images.

In [4]:

```
# Load necessary packages
from keras.datasets import cifar10
from keras.utils import np_utils
from matplotlib import pyplot as plt
import numpy as np
from PIL import Image
```

In [5]:

```
# load the data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

In [6]:

```
# Lets determine the dataset characteristics
print('Training Images: {}'.format(X_train.shape))
print('Testing Images: {}'.format(X_test.shape))
```

```
Training Images: (50000, 32, 32, 3)
Testing Images: (10000, 32, 32, 3)
```

In [7]:

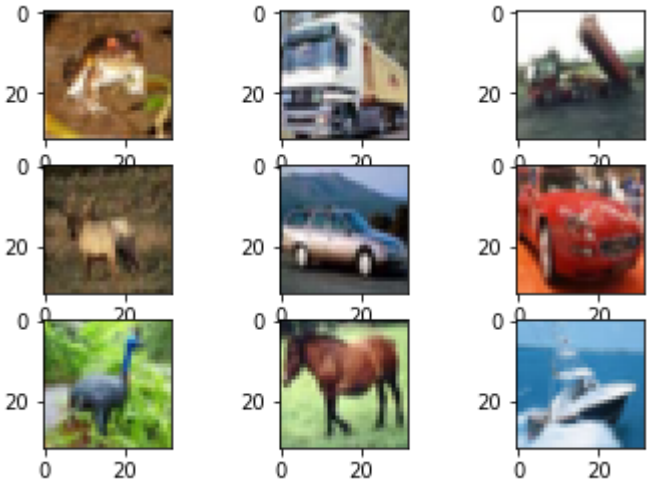
```
# Now for a single image
print(X_train[0].shape)
```

```
(32, 32, 3)
```

In [8]:

```
# create a grid of 3x3 images
for i in range(0,9):
    plt.subplot(330 + 1 + i)
    img = X_train[i].transpose([0,1,2])
    plt.imshow(img)

# show the plot
plt.show()
```



2. Preprocessing the dataset

First things first, we need to preprocess the dataset so the images and labels are in a form that Keras can ingest. To start, we'll define a NumPy seed for reproducibility, then normalize the images.

Furthermore, we will also convert our class labels to one-hot vectors. This is a standard output format for neural networks.

In [9]:

```
# Building a convolutional neural network for object recognition on CIFAR-10

# fix random seed for reproducibility
seed = 6
np.random.seed(seed)

# load the data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# normalize the inputs from 0-255 to 0.0-1.0
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
```

In [10]:

```
# class labels shape
print(y_train.shape)
print(y_train[0])
```

```
(50000, 1)
[6]
```

The class labels are a single integer value (0-9). What we really want is a one-hot vector of length ten. For example, the class label of 6 should be denoted [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]. We can accomplish this using the np_utils.to_categorical() function.

In [11]:

```
# hot encode outputs
Y_train = np_utils.to_categorical(y_train)
Y_test = np_utils.to_categorical(y_test)
num_classes = Y_test.shape[1]

print(Y_train.shape)
print(Y_train[0])
```

```
(50000, 10)
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

3. Building the All-CNN

Using the paper as a reference, we can implement the All-CNN network in Keras. Keras models are built by simply adding layers, one after another.

To make things easier for us later, we will wrap this model in a function, which will allow us to quickly and neatly generate the model later on in the project.

In [14]:

```
# start building the model - import necessary layers
from keras.models import Sequential
from keras.layers import Dropout, Activation, Conv2D, GlobalAveragePooling2D
from keras.optimizers import SGD

def allcnn(weights=None):
    # define model type - Sequential
    model = Sequential()

    # add model layers - Convolution2D, Activation, Dropout
    model.add(Conv2D(96, (3, 3), padding = 'same', input_shape=(32, 32, 3)))
    model.add(Activation('relu'))
    model.add(Conv2D(96, (3, 3), padding = 'same'))
    model.add(Activation('relu'))
    model.add(Conv2D(96, (3, 3), padding = 'same', strides = (2,2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(192, (3, 3), padding = 'same'))
    model.add(Activation('relu'))
    model.add(Conv2D(192, (3, 3), padding = 'same'))
    model.add(Activation('relu'))
    model.add(Conv2D(192, (3, 3), padding = 'same', strides = (2,2)))
    model.add(Dropout(0.5))

    model.add(Conv2D(192, (3, 3), padding = 'same'))
    model.add(Activation('relu'))
    model.add(Conv2D(192, (1, 1), padding = 'valid'))
    model.add(Activation('relu'))
    model.add(Conv2D(10, (1, 1), padding = 'valid'))

    # add GlobalAveragePooling2D layer with Softmax activation
    model.add(GlobalAveragePooling2D())
    model.add(Activation('softmax'))

    # load the weights
    if weights:
        model.load_weights(weights)

    # return model
    return model
```

4. Defining Parameters and Training the Model

We're all set! We are ready to start training our network. In the following cells, we will define our hyper parameters, such as learning rate and momentum, define an optimizer, compile the model, and fit the model to the training data.

In [15]:

```
# define hyper parameters
learning_rate = 0.01
weight_decay = 1e-6
momentum = 0.9

# build model
model = allcnn()

# define optimizer and compile model
sgd = SGD(lr=learning_rate, decay=weight_decay, momentum=momentum, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

# print model summary
print (model.summary())

# define additional training parameters
epochs = 350
batch_size = 32

# fit the model
model.fit(X_train, Y_train, validation_data=(X_test, Y_test), epochs=epochs, batch_size=bat
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 32, 32, 96)        2688
_________________________________________________________________
activation_8 (Activation)    (None, 32, 32, 96)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 32, 32, 96)        83040
_________________________________________________________________
activation_9 (Activation)    (None, 32, 32, 96)        0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 16, 16, 96)        83040
_________________________________________________________________
dropout_3 (Dropout)          (None, 16, 16, 96)        0
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 16, 16, 192)       166080
_________________________________________________________________
activation_10 (Activation)   (None, 16, 16, 192)       0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 16, 16, 192)       331968
_________________________________________________________________
activation_11 (Activation)   (None, 16, 16, 192)       0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 8, 8, 192)         331968
_________________________________________________________________
dropout_4 (Dropout)          (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 8, 8, 192)         331968
_________________________________________________________________
activation_12 (Activation)   (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 8, 8, 192)         37056
_________________________________________________________________
activation_13 (Activation)   (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 8, 8, 10)          1930
_________________________________________________________________
```

```
global_average_pooling2d_2 ( (None, 10)                0
_________________________________________________________________
activation_14 (Activation)   (None, 10)                0
=================================================================
Total params: 1,369,738
Trainable params: 1,369,738
Non-trainable params: 0
_________________________________________________________________
None
Train on 50000 samples, validate on 10000 samples
Epoch 1/350
 2752/50000 [>.............................] - ETA: 2:22:37 - loss: 2.3020 -
acc: 0.0956

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-15-e6ef3dfcb4ab> in <module>()
     19
     20 # fit the model
---> 21 model.fit(X_train, Y_train, validation_data=(X_test, Y_test), epochs
=epochs, batch_size=batch_size, verbose = 1)

~\Anaconda3\lib\site-packages\keras\engine\training.py in fit(self, x, y, ba
tch_size, epochs, verbose, callbacks, validation_split, validation_data, shu
ffle, class_weight, sample_weight, initial_epoch, steps_per_epoch, validatio
n_steps, **kwargs)
   1037                                         initial_epoch=initial_epoch,
   1038                                         steps_per_epoch=steps_per_ep
och,
-> 1039                                         validation_steps=validation_
steps)
   1040
   1041     def evaluate(self, x=None, y=None,

~\Anaconda3\lib\site-packages\keras\engine\training_arrays.py in fit_loop(mo
del, f, ins, out_labels, batch_size, epochs, verbose, callbacks, val_f, val_
ins, shuffle, callback_metrics, initial_epoch, steps_per_epoch, validation_s
teps)
    197                     ins_batch[i] = ins_batch[i].toarray()
    198
--> 199                 outs = f(ins_batch)
    200                 outs = to_list(outs)
    201                 for l, o in zip(out_labels, outs):

~\Anaconda3\lib\site-packages\keras\backend\tensorflow_backend.py in __call_
_(self, inputs)
   2713                 return self._legacy_call(inputs)
   2714
-> 2715             return self._call(inputs)
   2716         else:
   2717             if py_any(is_tensor(x) for x in inputs):

~\Anaconda3\lib\site-packages\keras\backend\tensorflow_backend.py in _call(s
elf, inputs)
   2673             fetched = self._callable_fn(*array_vals, run_metadata=se
lf.run_metadata)
   2674         else:
-> 2675             fetched = self._callable_fn(*array_vals)
   2676         return fetched[:len(self.outputs)]
   2677
```

```
~\Anaconda3\lib\site-packages\tensorflow\python\client\session.py in __call_
_(self, *args, **kwargs)
   1437           ret = tf_session.TF_SessionRunCallable(
   1438               self._session._session, self._handle, args, status,
-> 1439               run_metadata_ptr)
   1440         if run_metadata:
   1441           proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

KeyboardInterrupt:
```

5. Woah, that's a long time...

Uh oh. It's apparent that training this deep convolutional neural network is going to take a long time, which is not surprising considering the network has about 1.3 million parameters. Updating this many parameters takes a considerable amount of time; unless, of course, you are using a Graphics Processing Unit (GPU). This is a good time for a quick lesson on the differences between CPUs and GPUs.

The **central processing unit (CPU)** is often called the brains of the PC because it handles the majority of necessary computations. All computers have a CPU and this is what Keras and Theano automatically utilize.

The **graphics processing unit (GPU)** is in charge of image rendering. The most advanced GPUs were originally designed for gamers; however, GPU-accelerated computing, the use of a GPU together with a CPU to accelarate deep learing, analytics, and engineering applications, has become increasingly common. In fact, the training of deep neural networks is not realistic without them.

The most common GPUs for deep learning are produced by NVIDIA. Furthermore, the NVIDIA Deep Learning SDK provides high-performance tools and libraries to power GPU-accelerated machine learning applications. An alternative would be an AMD GPU in combination with the OpenCL libraries; however, these libraries have fewer active users and less support than the NVIDIA libraries.

If your computer has an NVIDIA GPU, installing the CUDA Drivers and CUDA Tookit from NVIDIA will allow Theano and Keras to utilize GPU-accelerated computing. The original paper mentions that it took approximately 10 hours to train the All-CNN network for 350 epochs using a modern GPU, which is considerably faster (several orders of magnitude) than it would take to train on CPU.

If you haven't already, stop the cell above. In the following cells, we'll save some time by loading pre-trained weights for the All-CNN network. Using these weights, we can evaluate the performance of the All-CNN network on the testing dataset.

In [16]:

```
# define hyper parameters
learning_rate = 0.01
weight_decay = 1e-6
momentum = 0.9

# define weights and build model
weights = 'all_cnn_weights_0.9088_0.4994.hdf5'
model = allcnn(weights)

# define optimizer and compile model
sgd = SGD(lr=learning_rate, decay=weight_decay, momentum=momentum, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

# print model summary
print (model.summary())

# test the model with pretrained weights
scores = model.evaluate(X_test, Y_test, verbose=1)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_19 (Conv2D)           (None, 32, 32, 96)        2688
_________________________________________________________________
activation_15 (Activation)   (None, 32, 32, 96)        0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 32, 32, 96)        83040
_________________________________________________________________
activation_16 (Activation)   (None, 32, 32, 96)        0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 16, 16, 96)        83040
_________________________________________________________________
dropout_5 (Dropout)          (None, 16, 16, 96)        0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 16, 16, 192)       166080
_________________________________________________________________
activation_17 (Activation)   (None, 16, 16, 192)       0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 16, 16, 192)       331968
_________________________________________________________________
activation_18 (Activation)   (None, 16, 16, 192)       0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 8, 8, 192)         331968
_________________________________________________________________
dropout_6 (Dropout)          (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 8, 8, 192)         331968
_________________________________________________________________
activation_19 (Activation)   (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 8, 8, 192)         37056
_________________________________________________________________
activation_20 (Activation)   (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 8, 8, 10)          1930
_________________________________________________________________
global_average_pooling2d_3 ( (None, 10)                0
_________________________________________________________________
```

```
activation_21 (Activation)   (None, 10)                0
=================================================================
Total params: 1,369,738
Trainable params: 1,369,738
Non-trainable params: 0
_________________________________________________________________
None
10000/10000 [==============================] - 341s 34ms/step
Accuracy: 90.88%
```

6. Making Predictions

Using the pretrained weights, we were able to achieve an accuracy of nearly 90 percent! Let's leverage this network to make some predictions. To start, we will generate a dictionary of class labels and names by referencing the website for the CIFAR-10 dataset:

https://www.cs.toronto.edu/~kriz/cifar.html (https://www.cs.toronto.edu/~kriz/cifar.html)

Next, we'll make predictions on nine images and compare the results to the ground-truth labels. Furthermore, we will plot the images for visual reference, this is object recognition after all.

In [17]:

```
# make dictionary of class labels and names
classes = range(0,10)

names = ['airplane',
         'automobile',
         'bird',
         'cat',
         'deer',
         'dog',
         'frog',
         'horse',
         'ship',
         'truck']

# zip the names and classes to make a dictionary of class_labels
class_labels = dict(zip(classes, names))

# generate batch of 9 images to predict
batch = X_test[100:109]
labels = np.argmax(Y_test[100:109],axis=-1)

# make predictions
predictions = model.predict(batch, verbose = 1)
```

```
9/9 [==============================] - 1s 70ms/step
```

In [19]:

```
# print our predictions
print (predictions)
```

```
[[4.56472342e-18 1.05810246e-20 1.89232421e-10 2.14114854e-11
  9.99999285e-01 2.80375247e-07 3.14786662e-13 4.23812452e-07
  1.36995572e-19 1.20971122e-18]
 [2.11912264e-16 2.03867016e-17 1.56727065e-09 5.35816639e-07
  1.62562644e-10 9.99999404e-01 1.67125547e-09 2.07823074e-08
  3.65486119e-15 3.51288651e-16]
 [1.27203225e-30 5.02736423e-28 5.15396277e-24 3.60512695e-21
  1.08752228e-27 8.15215322e-22 1.00000000e+00 1.10509080e-27
  6.19246192e-32 1.22743333e-23]
 [1.51438294e-16 4.76690539e-18 2.25836949e-09 1.00000000e+00
  4.34811284e-11 1.68841135e-12 4.53171246e-13 2.66869536e-16
  7.26597001e-19 6.20504176e-17]
 [8.24998297e-36 1.00000000e+00 0.00000000e+00 0.00000000e+00
  0.00000000e+00 0.00000000e+00 0.00000000e+00 9.35651890e-36
  8.68659706e-30 8.21906932e-22]
 [5.96992736e-24 1.00000000e+00 7.54846902e-27 8.02280708e-27
  1.97139506e-30 2.73385063e-29 7.21873855e-31 5.56797325e-27
  1.61868680e-26 1.19061282e-13]
 [6.45635478e-18 2.96424244e-20 1.00464758e-11 9.99999881e-01
  6.01552066e-13 9.11104578e-08 7.88743376e-12 1.16087009e-15
  7.85877643e-21 8.54759250e-21]
 [1.66711520e-27 7.76320857e-24 5.14038596e-20 7.23866829e-18
  2.65794982e-21 1.44404203e-24 1.00000000e+00 4.44152956e-27
  6.77591273e-26 9.29662637e-24]
 [4.17356525e-25 4.33874981e-23 1.77925093e-27 2.91175471e-23
  1.00906176e-24 3.12951506e-20 1.04254110e-21 2.77667023e-21
  1.00000000e+00 2.10683519e-28]]
```

In [20]:

```
# these are individual class probabilities, should sum to 1.0 (100%)
for image in predictions:
    print(np.sum(image))
```

```
1.0
0.99999994
1.0
1.0
1.0
1.0
1.0
1.0
1.0
```

In [21]:

```
# use np.argmax() to convert class probabilities to class labels
class_result = np.argmax(predictions,axis=-1)
print (class_result)
```

```
[4 5 6 3 1 1 3 6 8]
```

In [22]:

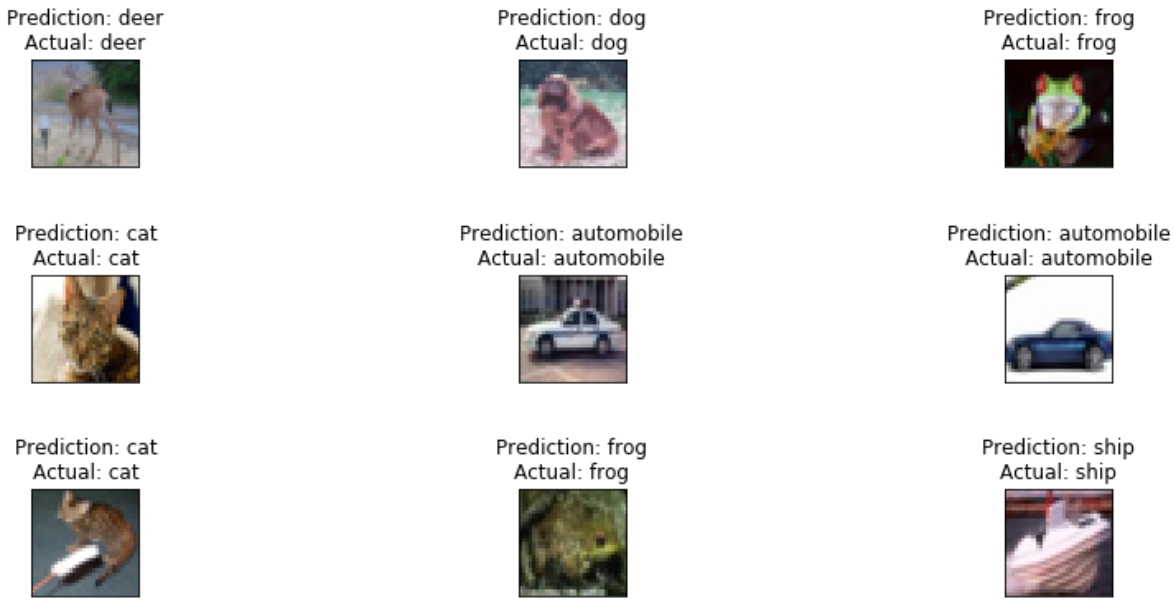
```
# create a grid of 3x3 images
fig, axs = plt.subplots(3, 3, figsize = (15, 6))
fig.subplots_adjust(hspace = 1)
axs = axs.flatten()

for i, img in enumerate(batch):

    # determine label for each prediction, set title
    for key, value in class_labels.items():
        if class_result[i] == key:
            title = 'Prediction: {}\nActual: {}'.format(class_labels[key], class_labels[lab
            axs[i].set_title(title)
            axs[i].axes.get_xaxis().set_visible(False)
            axs[i].axes.get_yaxis().set_visible(False)

    # plot the image
    axs[i].imshow(img.transpose([0,1,2]))

# show the plot
plt.show()
```



In [ ]: