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

In this project, we will be deploying a convolutional neural network (CNN) for object recognition. More specifically, we will be using the All-CNN network published in the 2015 ICLR paper, "Striving For Simplicity: The All Convolutional Net". This paper can be found at the following link:

https://arxiv.org/pdf/1412.6806.pdf (https://arxiv.org/pdf/1412.6806.pdf)

This convolutional neural network obtained state-of-the-art performance at object recognition on the CIFAR-10 image dataset in 2015. We will build this model using Keras, a high-level neural network application programming interface (API) that supports both Theano and Tensorflow backends. You can use either backend; however, I will be using Theano.

In this project, we will learn to:

- Import datasets from Keras
- Use one-hot vectors for categorical labels
- Addlayers to a Keras model
- Load pre-trained weights
- 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 [23]:

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

In [24]:

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

In [25]:

```
#determine characteristics for dataset
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)
```

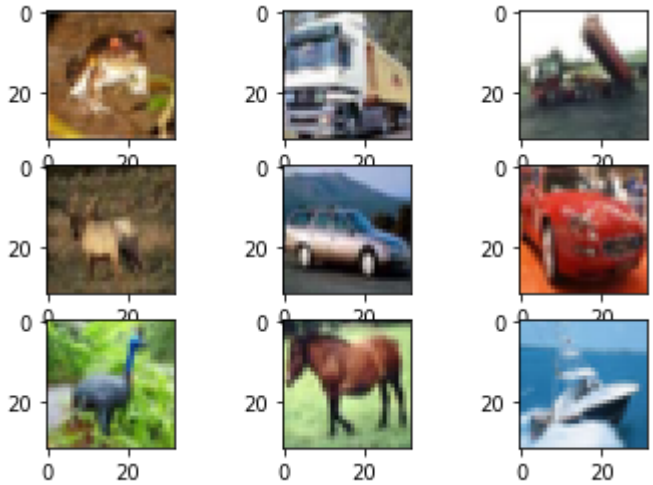
In [26]:

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

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

In [27]:

```
#create grid of images
for i in range (9):
    plt.subplot(331+i)
    img=X_train[i]
    plt.imshow(img)
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 [28]:

```
# 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 [29]:

```
# class labels shape
print(y_train.shape)
print(y_train[0])
print(y_train[1])
print(y_train[2])
print(y_train[3])
print(y_train[4])
print(y_train[5])
print(y_train[6])
print(y_train[7])
print(y_train[8])
```

```
(50000, 1)
[6]
[9]
[9]
[4]
[1]
[1]
[2]
[7]
[8]
```

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 [30]:

```
# 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.

Model All-CNN-C

- Input 32 × 32 RGB image
- 3 × 3 conv. 96 ReLU
- 3 × 3 conv. 96 ReLU

- 3 × 3 max-pooling stride 2
- 3 × 3 conv. 192 ReLU
- 3 × 3 conv. 192 ReLU
- 3 × 3 max-pooling stride 2
- 3 × 3 conv. 192 ReLU
- 1 × 1 conv. 192 ReLU
- 1 × 1 conv. 10 ReLU
- global averaging over 6 × 6 spatial dimensions
- 10 or 100-way softmax

In [31]:

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import Flatten
from keras.constraints import maxnorm
from keras.optimizers import SGD
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
```

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 [41]:

```
def allcnn(weights = None):

    # Define model type -Sequential
    model = Sequential()

    # add model layers
    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 Global average pooling layer with softmax activation
    model.add(GlobalAveragePooling2D())
    model.add(Activation('softmax'))

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

    # Return the model
    return model
```

In [ ]:

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

# build the model
model = allcnn()

# define an optimizer and compile this 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
```

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).

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 [45]:

```
# 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))
```

```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_77 (Conv2D)           (None, 32, 32, 96)        2688
_________________________________________________________________
activation_57 (Activation)   (None, 32, 32, 96)        0
_________________________________________________________________
conv2d_78 (Conv2D)           (None, 32, 32, 96)        83040
_________________________________________________________________
activation_58 (Activation)   (None, 32, 32, 96)        0
_________________________________________________________________
conv2d_79 (Conv2D)           (None, 16, 16, 96)        83040
_________________________________________________________________
dropout_21 (Dropout)         (None, 16, 16, 96)        0
_________________________________________________________________
conv2d_80 (Conv2D)           (None, 16, 16, 192)       166080
_________________________________________________________________
activation_59 (Activation)   (None, 16, 16, 192)       0
_________________________________________________________________
conv2d_81 (Conv2D)           (None, 16, 16, 192)       331968
_________________________________________________________________
activation_60 (Activation)   (None, 16, 16, 192)       0
_________________________________________________________________
conv2d_82 (Conv2D)           (None, 8, 8, 192)         331968
_________________________________________________________________
dropout_22 (Dropout)         (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_83 (Conv2D)           (None, 8, 8, 192)         331968
_________________________________________________________________
activation_61 (Activation)   (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_84 (Conv2D)           (None, 8, 8, 192)         37056
_________________________________________________________________
activation_62 (Activation)   (None, 8, 8, 192)         0
_________________________________________________________________
conv2d_85 (Conv2D)           (None, 8, 8, 10)          1930
_________________________________________________________________
global_average_pooling2d_9 ( (None, 10)                0
```

```
_________________________________________________________________
activation_63 (Activation)   (None, 10)                0         
=================================================================
Total params: 1,369,738
Trainable params: 1,369,738
Non-trainable params: 0
_________________________________________________________________
None
10000/10000 [==============================] - 192s 19ms/step
Accuracy: 87.51%
```

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 [64]:

```
# 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[5000:5009]
labels = np.argmax(Y_test[5000:5009],axis=-1)

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

```
9/9 [==============================] - 0s 18ms/step
```

In [65]:

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

```
[[1.3459610e-29 6.8407724e-31 3.9189782e-27 6.2767672e-26 8.0752359e-23
  3.1965896e-19 2.3385702e-32 1.0000000e+00 8.2312272e-42 4.4530160e-34]
 [2.9330703e-15 1.1057101e-13 1.0581191e-09 3.4009782e-09 6.1784175e-11
  2.5881693e-11 1.0000000e+00 6.5168121e-16 1.6841737e-14 9.3355174e-14]
 [5.9528077e-07 4.8546012e-06 1.1152735e-14 8.7186812e-13 6.1035758e-15
  2.1005621e-12 3.8139729e-15 7.9155652e-15 9.9999112e-01 3.4526001e-06]
 [1.3220615e-09 3.0639661e-11 2.3889177e-07 4.0163370e-04 9.9835253e-01
  2.6616890e-06 8.4733537e-10 1.2428993e-03 1.3201207e-10 3.1573087e-08]
 [1.0795381e-17 2.1299791e-28 7.0847924e-15 2.8770266e-18 1.0000000e+00
  1.1652163e-18 4.5249335e-20 3.1212805e-25 8.5485208e-25 2.7915496e-21]
 [4.1387779e-34 5.8433843e-34 2.4034080e-31 3.5970451e-16 4.8830182e-27
  1.0000000e+00 3.8305063e-33 1.0469726e-29 5.0297913e-39 1.1014534e-29]
 [2.0205304e-12 5.5099798e-15 5.4173898e-02 4.3243603e-03 9.3361390e-01
  7.6611545e-03 5.4133608e-08 2.2661030e-04 4.3216441e-14 7.1630923e-11]
 [2.7922780e-18 4.0997023e-17 7.3098515e-16 1.0000000e+00 2.9802929e-12
  7.1233228e-09 3.4356944e-14 4.4420626e-16 6.0556228e-15 9.4337674e-18]
 [1.7642828e-13 6.2568642e-15 1.3204602e-12 6.6823965e-08 9.9999994e-01
  6.0801128e-11 3.3653549e-11 1.4839313e-12 2.5129090e-13 1.5052009e-14]]
```

In [66]:

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

```
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
```

In [67]:

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

```
[7 6 8 4 4 5 4 3 4]
```

In [68]:

```
# 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)

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