# Coding Companion for Intuitive Deep Learning Part 2 (Annotated)

Code Author: Joseph Lee Wei En

Additional Annotations in Italics (or additional Python comments) by Kayla Bandy

Date Annotated: 11/26/22

The medium post for this notebook is here.

In this notebook, we'll go through the code for the coding companion for Intuitive Deep Learning Part 2 to create your very first Convolutional neural network to predict what is contained within the image (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck). We will go through the following in this notebook:

- Exploring and Processing the Data
- Building and Training our Convolutional Neural Network
- Testing out with your own images

Note that the results you get might differ slightly from the blogpost as there is a degree of randomness in the way we split our dataset as well as the initialization of our neural network.

*This is some supervised learning since we have a 'correct' label for each image.*

# Exploring and Processing the Data

We will first have to download our dataset, CIFAR-10. The details of the dataset are as follows:

- Images to be recognized: Tiny images of 32 * 32 pixels
- Labels: 10 possible labels (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck)
- Dataset size: 60000 images, split into 50000 for training and 10000 for testing

In [1]:
```
#Import needed function from needed libary
from keras.datasets import cifar10

#Split dataset into pre-determined training & test portions
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

In [2]:
```
#See the shape of the training data
print('x_train shape:', x_train.shape)
```

```
x_train shape: (50000, 32, 32, 3)
```

In [3]:
```
#See the shape of the training data's label
print('y_train shape:', y_train.shape)
```

```
y_train shape: (50000, 1)
```

We will now take a look at an individual image. If we print out the first image of our training dataset (x_train[0]):

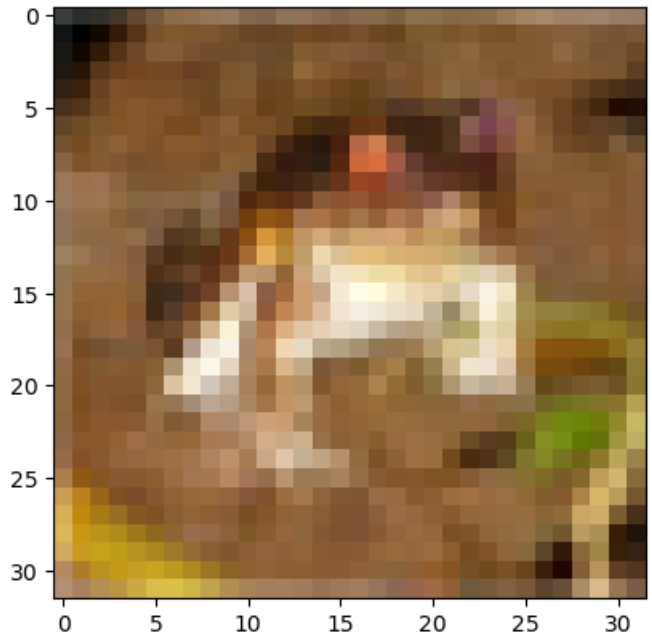
In [4]:
```
#See an item from the dataset to see what data we're working with
print(x_train[0])
```

```
[[[ 59  62  63]
  [ 43  46  45]
  [ 50  48  43]
  ...
  [158 132 108]
  [152 125 102]
  [148 124 103]]

 [[ 16  20  20]
  [  0   0   0]
  [ 18   8   0]
  ...
  [123  88  55]
  [119  83  50]
  [122  87  57]]

 [[ 25  24  21]
  [ 16   7   0]
  [ 49  27   8]
  ...
  [118  84  50]
  [120  84  50]
  [109  73  42]]

 ...

 [[208 170  96]
  [201 153  34]
  [198 161  26]
  ...
  [160 133  70]
  [ 56  31   7]
  [ 53  34  20]]

 [[180 139  96]
  [173 123  42]
  [186 144  30]
  ...
  [184 148  94]
  [ 97  62  34]
  [ 83  53  34]]

 [[177 144 116]
  [168 129  94]
  [179 142  87]
  ...
  [216 184 140]
  [151 118  84]
  [123  92  72]]]
```

In order to see the image as an image rather than a series of pixel value numbers, we will use a function from matplotlib:

In [5]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

In [6]:
```
#View item from the dataset as picture to see what we're working with
img = plt.imshow(x_train[0])
```




The labels correspond to these categories

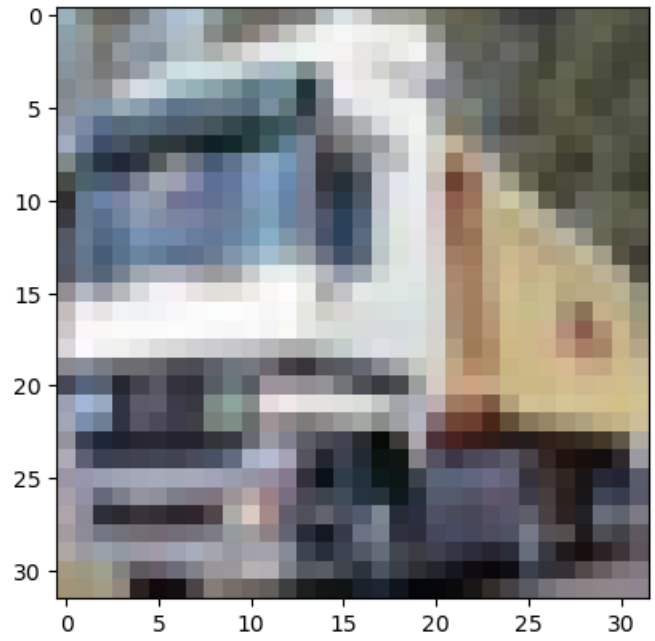
| Label | Category | Label Cont. | Category Cont. |
|---|---|---|---|
| 0 | airplane | 5 | dog |
| 1 | automobile | 6 | frog |
| 2 | bird | 7 | horse |
| 3 | cat | 8 | ship |
| 4 | deer | 9 | truck |

In [7]:
```
#Display the corresponding label
print('The label is:', y_train[0])
```

```
The label is: [6]
```

Let's explore one more image, the second image (with index 1 instead of 0) in our training dataset:

In [8]:
```
#View another image from the dataset
img = plt.imshow(x_train[1])
```




In [9]:
```
#Display the corresponding label
print('The label is:', y_train[1])
```

```
The label is: [9]
```

What we really want is the probability of each of the 10 different classes. For that, we need 10 output neurons in our neural network. Since we have 10 output neurons, our labels must match this as well. To do this, we convert the label into a set of 10 numbers where each number represents if the image belongs to that class or not. So if an image belongs to the first class, the first number of this set will be a 1 and all other numbers in this set will be a 0. To convert our labels to our one-hot encoding, we use a function in Keras:

In [10]:
```
import keras

#Convert the train & test data to allow for the categories
y_train_one_hot = keras.utils.to_categorical(y_train, 10)
y_test_one_hot = keras.utils.to_categorical(y_test, 10)
```

The conversion table now looks like this:

| Label | Category | One-Hot Encoding |
|---|---|---|
| 0 | airplane | [1 0 0 0 0 0 0 0 0 0] |
| 1 | automobile | [0 1 0 0 0 0 0 0 0 0] |
| 2 | bird | [0 0 1 0 0 0 0 0 0 0] |
| 3 | cat | [0 0 0 1 0 0 0 0 0 0] |
| 4 | deer | [0 0 0 0 1 0 0 0 0 0] |
| 5 | dog | [0 0 0 0 0 1 0 0 0 0] |
| 6 | frog | [0 0 0 0 0 0 1 0 0 0] |
| 7 | horse | [0 0 0 0 0 0 0 1 0 0] |
| 8 | ship | [0 0 0 0 0 0 0 0 1 0] |
| 9 | truck | [0 0 0 0 0 0 0 0 0 1] |

In [11]:
```
#View the one-hot encoding for a data element
print('The one hot label is:', y_train_one_hot[1])
```

```
The one hot label is: [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

A common step we do is to let the values to be between 0 and 1, which will aid in the training of our neural network. Since our pixel values already take the values between 0 and 255, we simply need to divide by 255.

*We're rescaling the values by dividing by 255.*

In [12]:
```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train / 255
x_test = x_test / 255
```

In [13]:
```
#View what the first result now looks like in the data
x_train[0]
```

Out[13]:
```
array([[[0.23137255, 0.24313726, 0.24705882],
        [0.16862746, 0.18039216, 0.1764706 ],
        [0.19607843, 0.1882353 , 0.16862746],
        ...,
        [0.61960787, 0.5176471 , 0.42352942],
        [0.59607846, 0.49019608, 0.4       ],
        [0.5803922 , 0.4862745 , 0.40392157]],

       [[0.0627451 , 0.07843138, 0.07843138],
        [0.        , 0.        , 0.        ],
        [0.07058824, 0.03137255, 0.        ],
        ...,
        [0.48235294, 0.34509805, 0.21568628],
        [0.46666667, 0.3254902 , 0.19607843],
        [0.47843137, 0.34117648, 0.22352941]],

       [[0.09803922, 0.09411765, 0.08235294],
        [0.0627451 , 0.02745098, 0.        ],
        [0.19215687, 0.10588235, 0.03137255],
        ...,
        [0.4627451 , 0.32941177, 0.19607843],
        [0.47058824, 0.32941177, 0.19607843],
        [0.42745098, 0.28627452, 0.16470589]],

       ...,

       [[0.8156863 , 0.6666667 , 0.3764706 ],
        [0.7882353 , 0.6       , 0.13333334],
        [0.7764706 , 0.6313726 , 0.10196079],
        ...,
        [0.627451  , 0.52156866, 0.27450982],
        [0.21960784, 0.12156863, 0.02745098],
        [0.20784314, 0.13333334, 0.07843138]],

       [[0.7058824 , 0.54509807, 0.3764706 ],
        [0.6784314 , 0.48235294, 0.16470589],
        [0.7294118 , 0.5647059 , 0.11764706],
        ...,
        [0.72156864, 0.5803922 , 0.36862746],
        [0.38039216, 0.24313726, 0.13333334],
        [0.3254902 , 0.20784314, 0.13333334]],

       [[0.69411767, 0.5647059 , 0.45490196],
        [0.65882355, 0.5058824 , 0.36862746],
        [0.7019608 , 0.5568628 , 0.34117648],
        ...,
        [0.84705883, 0.72156864, 0.54901963],
        [0.5921569 , 0.4627451 , 0.32941177],
        [0.48235294, 0.36078432, 0.28235295]]], dtype=float32)
```

# Building and Training our Convolutional Neural Network

Similar to our first notebook, we need to define the architecture (template) first before fitting the best numbers into this architecture by learning from the data. In summary, the architecture we will build in this post is this:

- Conv Layer (Filter size 3x3, Depth 32)
- Conv Layer (Filter size 3x3, Depth 32)
- Max Pool Layer (Filter size 2x2)
- Dropout Layer (Prob of dropout 0.25)
- Conv Layer (Filter size 3x3, Depth 64)
- Conv Layer (Filter size 3x3, Depth 64)
- Max Pool Layer (Filter size 2x2)
- Dropout Layer (Prob of dropout 0.25)
- FC Layer (512 neurons)
- Dropout Layer (Prob of dropout 0.5)
- FC Layer, Softmax (10 neurons)

For an intuition behind these layers, please refer to Intuitive Deep Learning Part 2.

*The basic idea is that we need the model to understand the relationships between pixels in order to understand the larger picture. The convolution layers slowly move over the image to understand the features that are by each other. The pooling layers then reduce the complexity by keeping only the maximum feature. the dropout layers prevent overfitting.*

We will be using Keras to build our architecture. Let's import the code from Keras that we will need to use:

In [14]:
```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
```

We then call an empty Sequential model and 'add' to this model layer by layer:

In [15]:
```
#Create an instance of the Sequential object
model = Sequential()
```

The first layer is a conv layer with filter size 3x3, stride size 1 (in both dimensions), and depth 32. The padding is the 'same' and the activation is 'relu' (these two settings will apply to all layers in our CNN). We add this layer to our empty sequential model using the function model.add().

The first number 32 refers to the depth. The next pair of numbers (3,3) refer to the filter width and size. Then, we specify activation which is 'relu' and padding which is 'same'. Notice that we did not specify stride. This is because stride=1 is a default setting, and unless we want to change this setting, we need not specify it.

If you recall, we also need to specify an input size for our first layer; subsequent layers does not have this specification since they can infer the input size from the output size of the previous layer.

All that being said, our first layer in code looks like this:

In [16]:
```
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32,32,3)))
```

Our second layer looks like this in code (we don't need to specify the input size):

In [17]:
```
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
```

The next layer is a max pooling layer with pool size 2 x 2 and stride 2 (in both dimensions). The default for a max pooling layer stride is the pool size, so we don't have to specify the stride:

In [18]:
```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

Lastly, we add a dropout layer with probability 0.25 of dropout so as to prevent overfitting:

In [19]:
```
model.add(Dropout(0.25))
```

And there we have it, our first four layers in code. The next four layers look really similar (except the depth of the conv layer is 64 instead of 32):

In [20]:
```
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

Lastly, we have to code in our fully connected layer, which is similar to what we've done in our previous post, Build your first Neural Network. However, at this point, our neurons are spatially arranged in a cube-like format rather than in just one row. To make this cube-like format of neurons into one row, we have to first flatten it. We do so by adding a Flatten layer:

In [21]:
```
model.add(Flatten())
```

Now, we have a dense (FC) layer of 512 neurons with relu activation:

In [22]:
```
model.add(Dense(512, activation='relu'))
```

We add another dropout of probability 0.5:

In [23]:
```
model.add(Dropout(0.5))
```

And lastly, we have a dense (FC) layer with 10 neurons and softmax activation:

In [24]:
```
model.add(Dense(10, activation='softmax'))
```

And we're done with specifying our architecture! To see a summary of the full architecture, we run the code:

In [25]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 32)        896

 conv2d_1 (Conv2D)           (None, 32, 32, 32)        9248

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 dropout (Dropout)           (None, 16, 16, 32)        0

 conv2d_2 (Conv2D)           (None, 16, 16, 64)        18496

 conv2d_3 (Conv2D)           (None, 16, 16, 64)        36928

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 dropout_1 (Dropout)         (None, 8, 8, 64)          0

 flatten (Flatten)           (None, 4096)              0

 dense (Dense)               (None, 512)               2097664

 dropout_2 (Dropout)         (None, 512)               0

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 2,168,362
Trainable params: 2,168,362
Non-trainable params: 0
_________________________________________________________________
```

We now fill in the best numbers after we've specified our architecture. We'll compile the model with our settings below.

The loss function we use is called categorical cross entropy, which is applicable for a classification problem of many classes. The optimizer we use here is Adam. We haven't gone through the intuition of Adam yet, but know that Adam is simply a type of stochastic gradient descent (with a few modifications) so that it trains better. Lastly, we want to track the accuracy of our model.

In [26]:
```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['acc'])
```

And now, it's time to run our training.

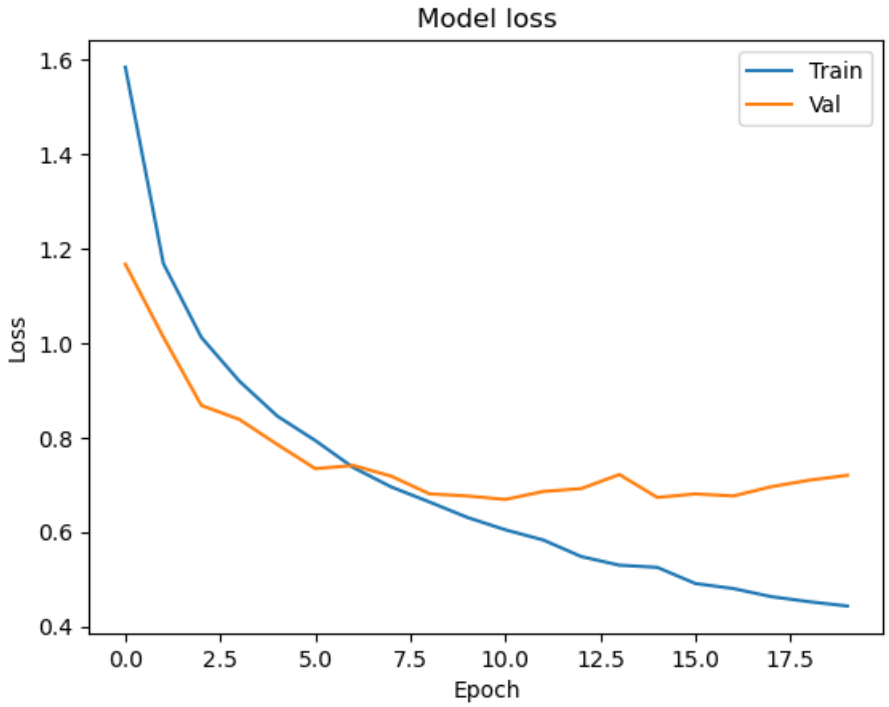
We train our model with batch size 32 and 20 epochs. We use the setting validation_split=0.2 instead of validation_data. With this shortcut, we did not need to split our dataset into a train and validation set at the start! Instead, we simply specify how much of our dataset will be used as a validation set. In this case, 20% of our dataset is used as a validation set. This will take a while on a CPU, so you might want to start training and get some coffee before coming back.

In [27]:
```
hist = model.fit(x_train, y_train_one_hot,
           batch_size=32, epochs=20,
           validation_split=0.2)
```

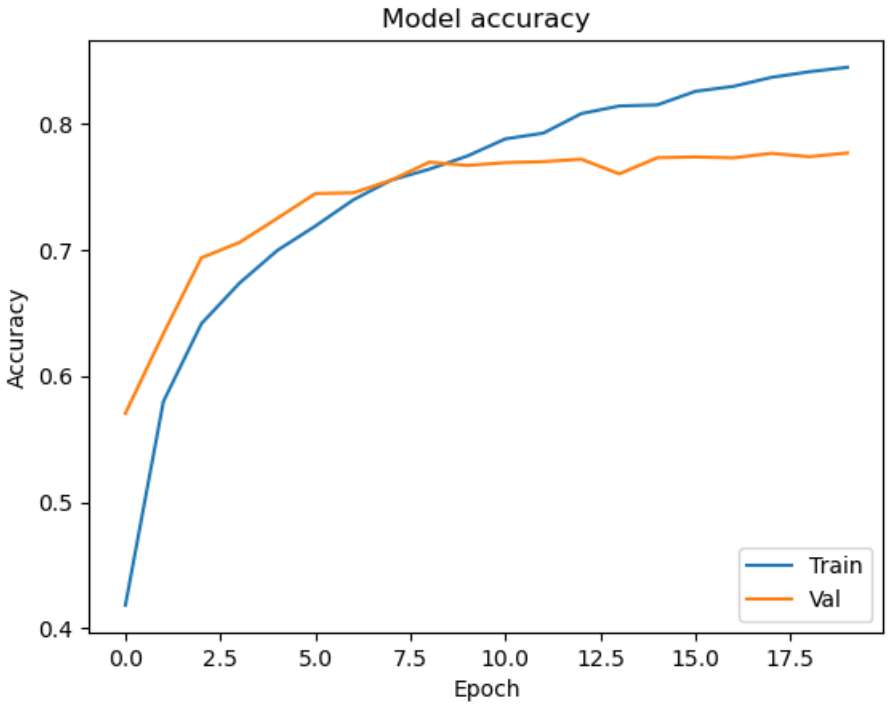
```
Epoch 1/20
1250/1250 [==============================] - 89s 70ms/step - loss: 1.5842 - acc: 0.4183 - val_loss: 1.1669 -
val_acc: 0.5703
Epoch 2/20
1250/1250 [==============================] - 88s 71ms/step - loss: 1.1684 - acc: 0.5797 - val_loss: 1.0126 -
val_acc: 0.6335
Epoch 3/20
1250/1250 [==============================] - 87s 70ms/step - loss: 1.0119 - acc: 0.6414 - val_loss: 0.8679 -
val_acc: 0.6936
Epoch 4/20
1250/1250 [==============================] - 89s 71ms/step - loss: 0.9193 - acc: 0.6735 - val_loss: 0.8382 -
val_acc: 0.7057
Epoch 5/20
1250/1250 [==============================] - 87s 69ms/step - loss: 0.8452 - acc: 0.6995 - val_loss: 0.7851 -
val_acc: 0.7249
Epoch 6/20
1250/1250 [==============================] - 85s 68ms/step - loss: 0.7934 - acc: 0.7188 - val_loss: 0.7339 -
val_acc: 0.7445
Epoch 7/20
1250/1250 [==============================] - 88s 70ms/step - loss: 0.7362 - acc: 0.7395 - val_loss: 0.7407 -
val_acc: 0.7451
Epoch 8/20
1250/1250 [==============================] - 87s 70ms/step - loss: 0.6951 - acc: 0.7552 - val_loss: 0.7179 -
val_acc: 0.7551
Epoch 9/20
1250/1250 [==============================] - 87s 69ms/step - loss: 0.6634 - acc: 0.7638 - val_loss: 0.6806 -
val_acc: 0.7694
Epoch 10/20
1250/1250 [==============================] - 87s 70ms/step - loss: 0.6306 - acc: 0.7742 - val_loss: 0.6760 -
val_acc: 0.7668
Epoch 11/20
1250/1250 [==============================] - 88s 71ms/step - loss: 0.6044 - acc: 0.7878 - val_loss: 0.6688 -
val_acc: 0.7690
Epoch 12/20
1250/1250 [==============================] - 89s 71ms/step - loss: 0.5828 - acc: 0.7924 - val_loss: 0.6856 -
val_acc: 0.7698
Epoch 13/20
1250/1250 [==============================] - 88s 71ms/step - loss: 0.5477 - acc: 0.8079 - val_loss: 0.6917 -
val_acc: 0.7717
Epoch 14/20
1250/1250 [==============================] - 88s 70ms/step - loss: 0.5295 - acc: 0.8139 - val_loss: 0.7216 -
val_acc: 0.7601
Epoch 15/20
1250/1250 [==============================] - 87s 69ms/step - loss: 0.5248 - acc: 0.8148 - val_loss: 0.6729 -
val_acc: 0.7729
Epoch 16/20
1250/1250 [==============================] - 87s 70ms/step - loss: 0.4907 - acc: 0.8255 - val_loss: 0.6805 -
val_acc: 0.7735
Epoch 17/20
1250/1250 [==============================] - 87s 70ms/step - loss: 0.4799 - acc: 0.8295 - val_loss: 0.6761 -
val_acc: 0.7728
Epoch 18/20
1250/1250 [==============================] - 88s 71ms/step - loss: 0.4628 - acc: 0.8365 - val_loss: 0.6957 -
val_acc: 0.7764
Epoch 19/20
1250/1250 [==============================] - 89s 71ms/step - loss: 0.4521 - acc: 0.8410 - val_loss: 0.7096 -
val_acc: 0.7737
Epoch 20/20
1250/1250 [==============================] - 87s 69ms/step - loss: 0.4430 - acc: 0.8446 - val_loss: 0.7198 -
val_acc: 0.7767
```

After you've done training, we can visualize the model training and validation loss as well as training / validation accuracy over the number of epochs using the below code:

In [28]:
```
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='upper right')
plt.show()
```




In [29]:
```
plt.plot(hist.history['acc'])
plt.plot(hist.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Val'], loc='lower right')
plt.show()
```




Once we are done with tweaking our hyperparameters, we can run it on our test dataset below:

In [30]:
```
#Get the model accuracy
model.evaluate(x_test, y_test_one_hot)[1]
```

```
313/313 [==============================] - 5s 16ms/step - loss: 0.7481 - acc: 0.7654
```

Out[30]:
```
0.7653999924659729
```

At this point, you might want to save your trained model (since you've spent so long waiting for it to train). The model will be saved in a file format called HDF5 (with the extension .h5). We save our model with this line of code:

In [31]:
```
model.save('my_cifar10_model.h5')
```

# Testing out with your own images

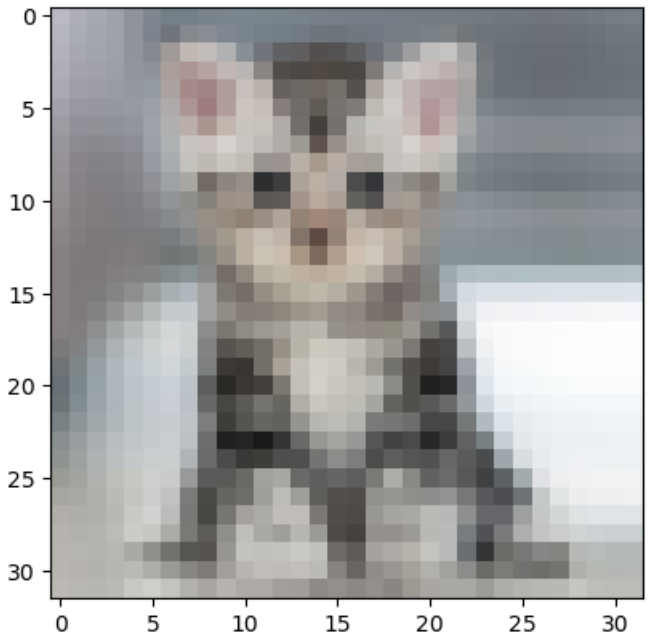
Now that we have a model, let's try it on our own images. To do so, place your image in the same directory as your notebook. For the purposes of this post, I'm going to use an image of a cat (which you can download here(link)). Now, we read in our JPEG file as an array of pixel values:

In [32]:
```
#Test additional image
my_image = plt.imread("cat.jpg")
```

The first thing we have to do is to resize the image of our cat so that we can fit it into our model (input size of 32 *32* 3).

In [33]:
```
from skimage.transform import resize
my_image_resized = resize(my_image, (32,32,3))
```

In [34]:
```
#Show the resulting augmented image
img = plt.imshow(my_image_resized)
```




And now, we see what our trained model will output when given an image of our cat, using this code:

In [35]:
```
import numpy as np
probabilities = model.predict(np.array( [my_image_resized,] ))
```

```
1/1 [==============================] - 0s 124ms/step
```

In [36]:
```
#Display probabilities for all categories possible
probabilities
```

Out[36]:
```
array([[7.6545883e-05, 2.3943103e-08, 2.2310249e-03, 2.2996087e-01,
        6.0544703e-03, 4.3813643e-01, 6.4969230e-05, 3.2343820e-01,
        3.5516980e-05, 1.9645443e-06]], dtype=float32)
```

In [37]:
```
number_to_class = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
index = np.argsort(probabilities[0,:])
print("Most likely class:", number_to_class[index[9]], "-- Probability:", probabilities[0,index[9]])
print("Second most likely class:", number_to_class[index[8]], "-- Probability:", probabilities[0,index[8]])
print("Third most likely class:", number_to_class[index[7]], "-- Probability:", probabilities[0,index[7]])
print("Fourth most likely class:", number_to_class[index[6]], "-- Probability:", probabilities[0,index[6]])
print("Fifth most likely class:", number_to_class[index[5]], "-- Probability:", probabilities[0,index[5]])
```

```
Most likely class: dog -- Probability: 0.43813643
Second most likely class: horse -- Probability: 0.3234382
Third most likely class: cat -- Probability: 0.22996087
Fourth most likely class: deer -- Probability: 0.0060544703
Fifth most likely class: bird -- Probability: 0.002231025
```

As you can see, the model has accurately predicted that this is indeed an image of a cat. Now, this isn't the best model we have and accuracy has been quite low, so don't expect too much out of it. This post has covered the very fundamentals of CNNs on a very simple dataset; we'll cover how to build state-of-the-art models in future posts. Nevertheless, you should be able to get some pretty cool results from your own images (some images that you can try this out on are in the GitHub folder).

*This run was comparable for accuracy and loss, yet this same image used in the original notebook was identified as a dog, and cat was the 3rd most likely option. This is interesting because I otherwise did not change any settings for the model.*