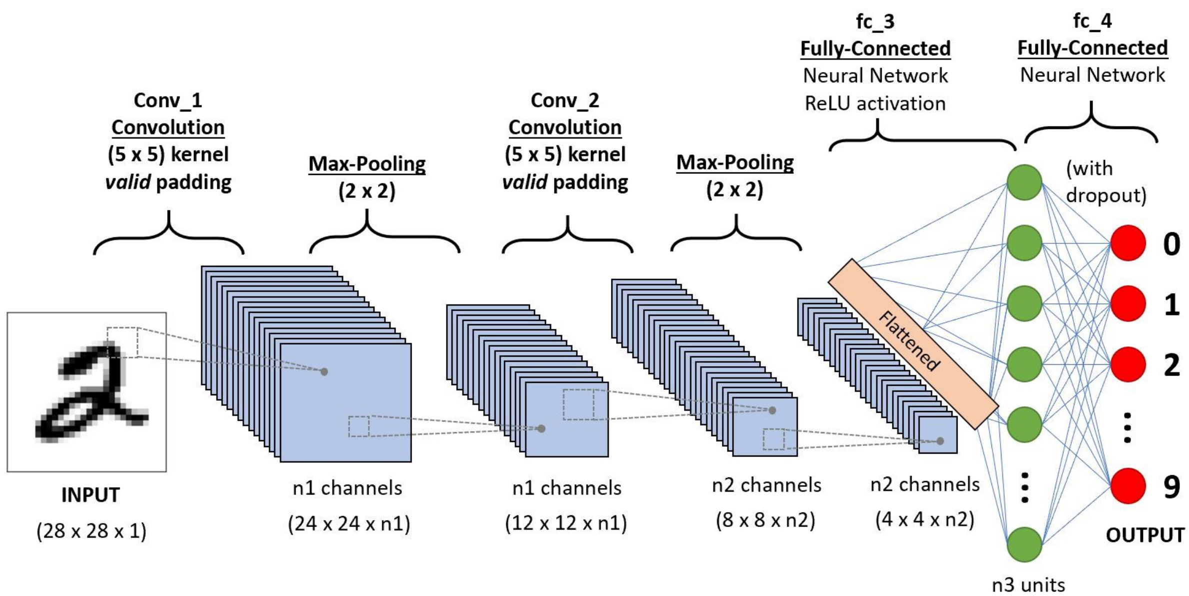
Clase 12, enero 6, 2021

Deep Learning using LeNet and AlexNet

Comment: LeNet and AlexNet

LeNet



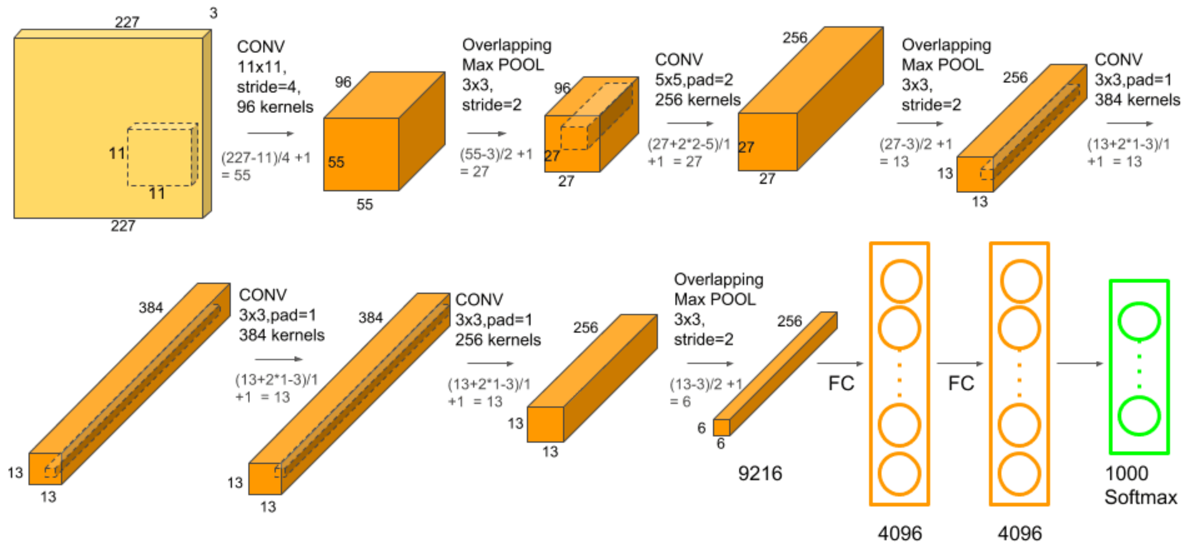
LeNet: these networks have a large number of weights and biases; overfitting should be attended

Article: LeNet

Comment: LeNet

Comment: Convolutional Neural Networks

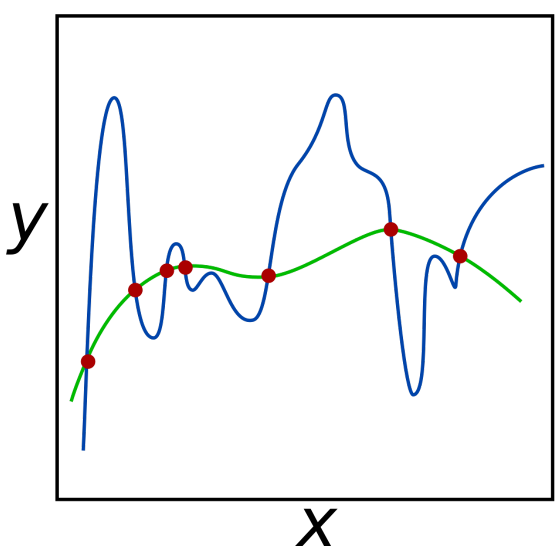
AlexNet

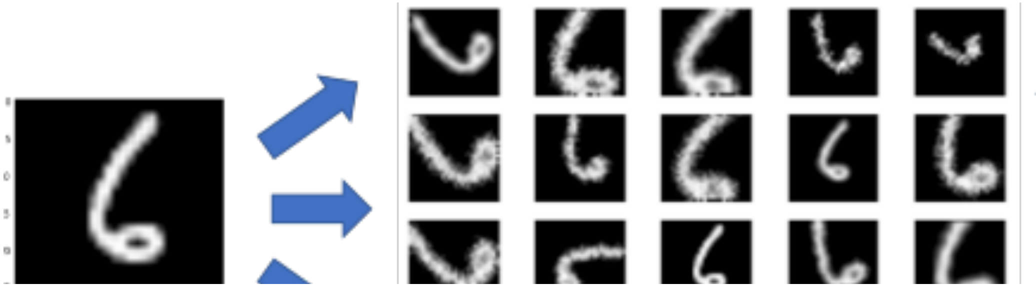


Article: AlexNet

Comment: AlexNet

A method to reduce overfitting: Data Augmentation



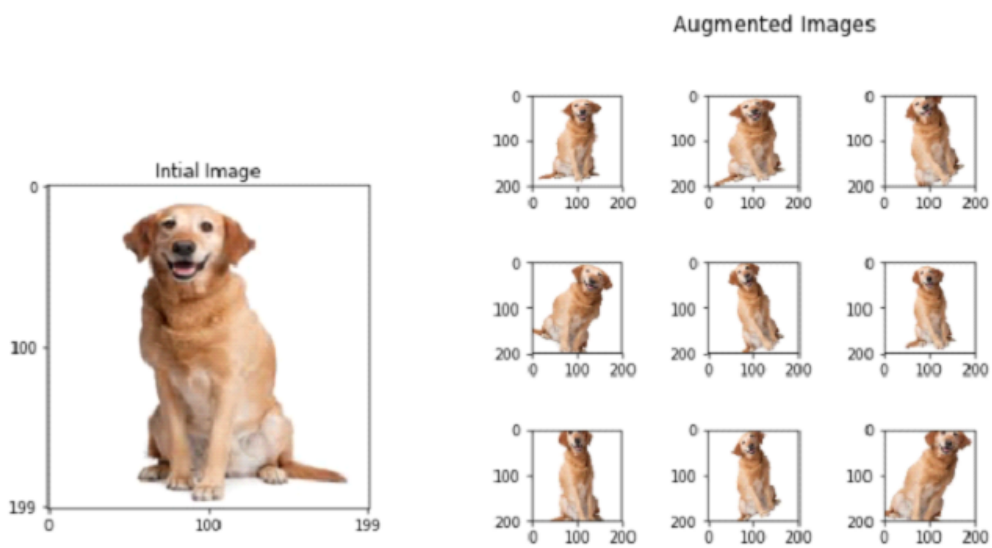


Data Augmentation

Paper: Augmentation overview

Deep networks are heavily reliant on big data to avoid overfitting:

Transforming an image



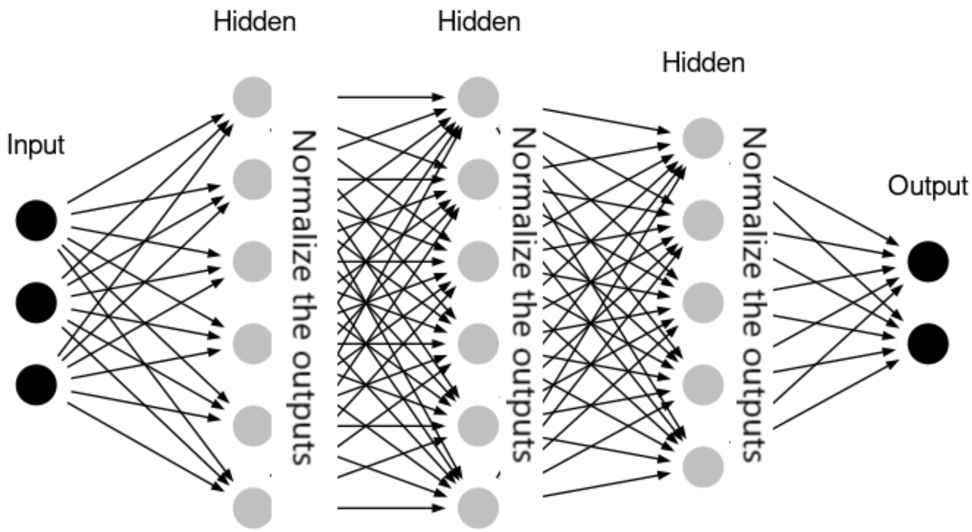
Transforming a curve

Keras: Image Preprocessing

Comment: About data augmentation for Deep Learning

Another way of reducing overfitting is using batch normalization

Paper: Batch normalization



Batch normalization helps to reduce the overfitting and accelerates the convergence of the network during training

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

Comment: batch normalization

Deep Learning: LeNet

If you use tensorflow-GPU, run the following cell

In [1]:
```
import tensorflow as tf

physical_devices = tf.config.experimental.list_physical_devices('GPU')
print("physical_devices-------------", len(physical_devices))
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

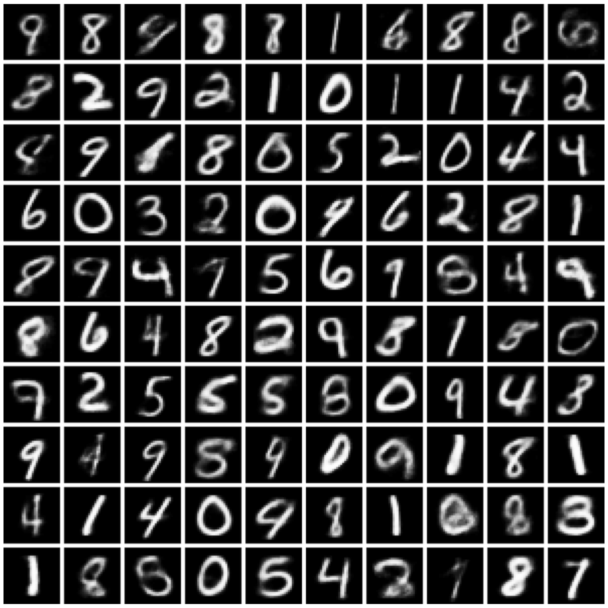
```
physical_devices------------- 1
```

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

from keras.models import Sequential
from keras.layers import Dense, Conv2D, Activation, Dropout, Flatten, MaxPool
from keras.layers import BatchNormalization
from keras.utils import plot_model
from keras import optimizers

import time

np.random.seed(10)
```

```
Using TensorFlow backend.
```

Data of the System to be analyzed: mnist

The MNIST database



Generation or extraction of the raw data

In [3]:
```
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

In [4]:
```
print("x_train, y_train type", type(x_train), type(y_train))
print("x_test, y_test type", type(x_test), type(y_test))
```

```
x_train, y_train type <class 'numpy.ndarray'> <class 'numpy.ndarray'>
x_test, y_test type <class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

In [5]:
```
print("x_train shape", x_train.shape)
print("y_train shape", y_train.shape)
print("x_test shape", x_test.shape)
print("y_test shape", y_test.shape)
```

```
x_train shape (60000, 28, 28)
y_train shape (60000,)
x_test shape (10000, 28, 28)
y_test shape (10000,)
```

Analysis of the raw data

In [6]:
```
image_index = 7777 # You may select anything up to 60,000
print(y_train[image_index]) # The label is 8
plt.imshow(x_train[image_index], cmap='Greys')
plt.show()
```

```
8
```

Transformation of the raw data

In [7]:
```
# Reshaping the array to 4-dims so that it can work with the Keras API
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
#input_shape = (28, 28, 1)

# Making sure that the values are float so that we can get decimal points aft
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

# Normalizing the RGB codes by dividing it to the max RGB value.
x_train /= 255.0
x_test /= 255.0
print('x_train shape:', x_train.shape)
print('y_train shape:', y_train.shape)
print('x_test shape:', x_test.shape)
print('y_test shape:', y_test.shape)
```

```
x_train shape: (60000, 28, 28, 1)
y_train shape: (60000,)
x_test shape: (10000, 28, 28, 1)
y_test shape: (10000,)
```

In [8]:
```
y_train[0:15]
```

Out[8]:
```
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1], dtype=uint8)
```

Definition of the neural network architecture

In [9]:
```
# Creating a Sequential Model and adding the layers

def architecture(batch_normalization, dropout, input_shape, activation):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(5,5), input_shape=input_shape))

    model.add(MaxPooling2D(pool_size=(2, 2)))
    if batch_normalization:
        model.add(BatchNormalization())    #The recomendaton is to perform ba

    model.add(Conv2D(64, kernel_size=(5,5), input_shape=input_shape))

    model.add(MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten()) # Flattening the 2D arrays for fully connected layer

    model.add(Dense(1024))
    if dropout:
        model.add(Dropout(0.2))
    if batch_normalization:
        model.add(BatchNormalization())  #The recomendaton is to perform batc
    model.add(Activation(activation))

    model.add(Dense(10,activation='softmax'))

    return model
```

Generating a model of deep neural network

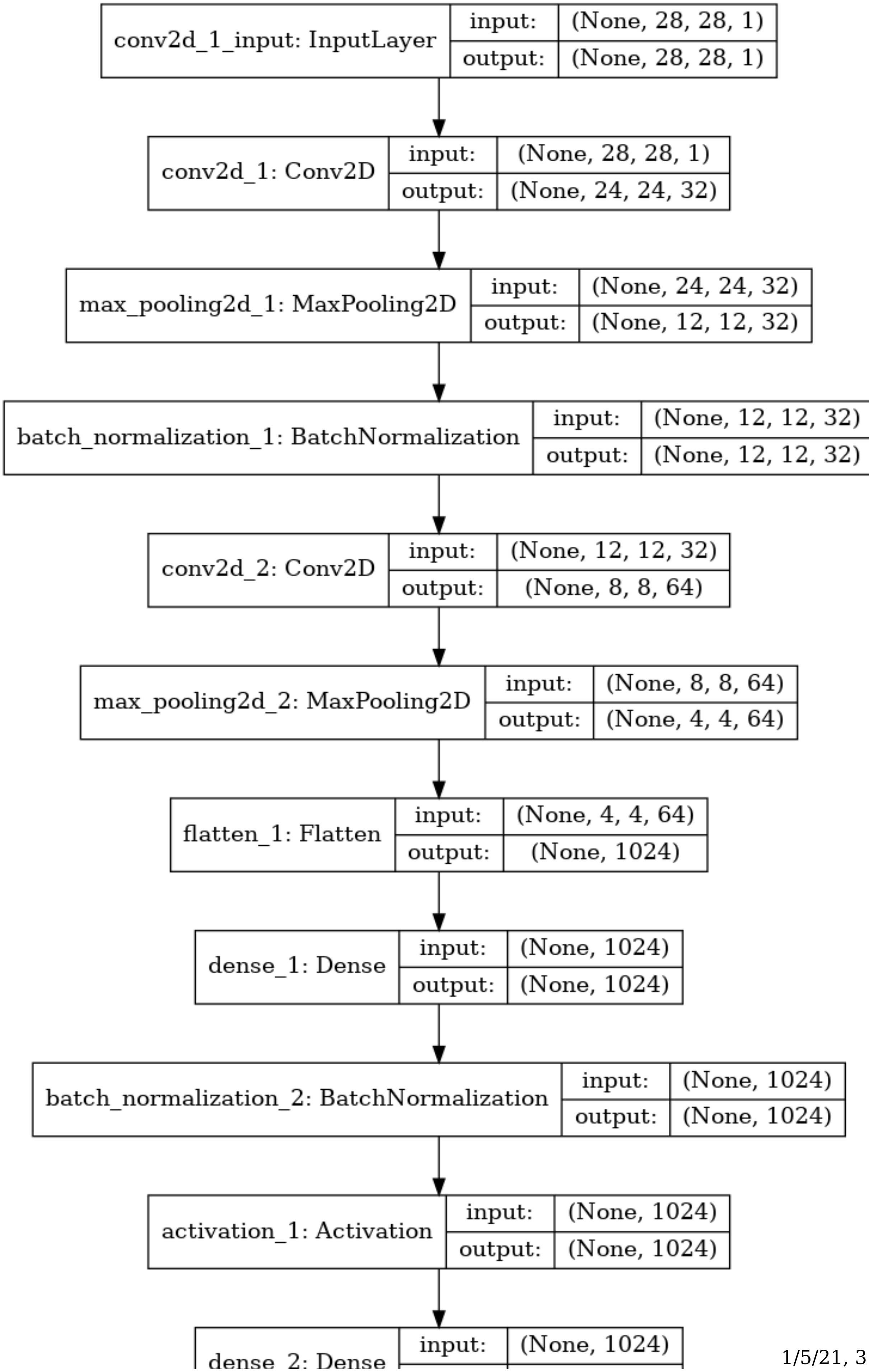
Playing with batch normalization and dropout, you will see that batch normalization improves better the network. Remember that batch normalization is applied before the activation.

Paper: Batch Normalization

In [10]:
```
batch_normalization=True
dropout=False
input_shape = (28, 28, 1)
activation = 'relu'

LeNet_model = architecture(batch_normalization, dropout, input_shape, activat
```

In [11]:
```
# Plotting the architecture

plot_model(LeNet_model, to_file='LeNet.png', show_shapes=True, show_layer_nam
```

Out[11]:


conv2d_1_input: InputLayer
input: (None, 28, 28, 1)
output: (None, 28, 28, 1)
conv2d_1: Conv2D
input: (None, 28, 28, 1)
output: (None, 24, 24, 32)
max_pooling2d_1: MaxPooling2D
input: (None, 24, 24, 32)
output: (None, 12, 12, 32)
batch_normalization_1: BatchNormalization
input: (None, 12, 12, 32)
output: (None, 12, 12, 32)
conv2d_2: Conv2D
input: (None, 12, 12, 32)
output: (None, 8, 8, 64)
max_pooling2d_2: MaxPooling2D
input: (None, 8, 8, 64)
output: (None, 4, 4, 64)
flatten_1: Flatten
input: (None, 4, 4, 64)
output: (None, 1024)
dense_1: Dense
input: (None, 1024)
output: (None, 1024)
batch_normalization_2: BatchNormalization
input: (None, 1024)
output: (None, 1024)
activation_1: Activation
input: (None, 1024)
output: (None, 1024)
dense_2: Dense
input: (None, 1024)

In [12]:
```
LeNet_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 24, 24, 32)        832       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0         
_________________________________________________________________
batch_normalization_1 (Batch (None, 12, 12, 32)        128       
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 64)          51264     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 4, 4, 64)          0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              1049600   
_________________________________________________________________
batch_normalization_2 (Batch (None, 1024)              4096      
_________________________________________________________________
activation_1 (Activation)    (None, 1024)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 10)                10250     
=================================================================
Total params: 1,116,170
Trainable params: 1,114,058
Non-trainable params: 2,112
_________________________________________________________________
```

[Keras: compiling methods]

Compiling the model

In [13]:
```
#Compiling the model

lr = 0.001

LeNet_model.compile(optimizer=optimizers.Adam(learning_rate=lr,beta_1=0.9, be
              loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Running the model

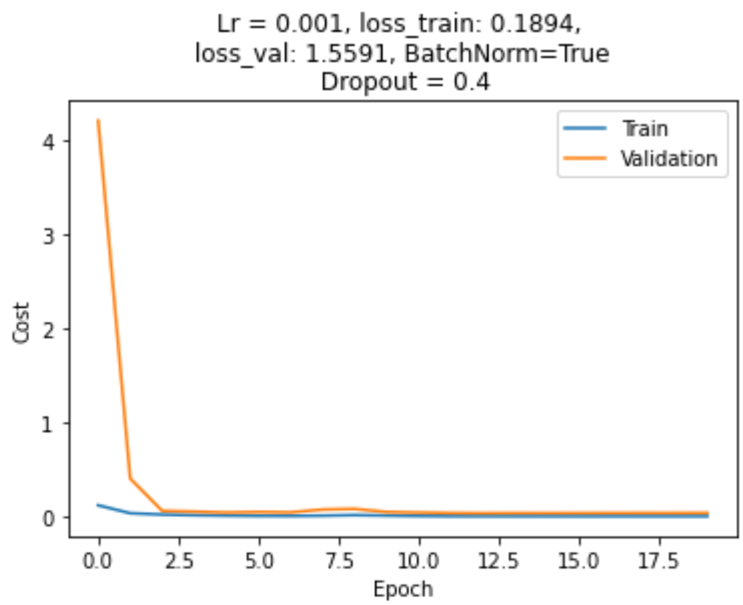
In [14]:
```
start_time = time.time()

num_epochs=20

history = LeNet_model.fit(x_train, y_train, batch_size=256, epochs=num_epochs

end_time = time.time()
print("Time for training: {:10.4f}s".format(end_time - start_time))
```

```
Train on 50400 samples, validate on 9600 samples
Epoch 1/20
```

```
50400/50400 [==============================] - 3s 53us/step - loss: 0.1167 -
accuracy: 0.9630 - val_loss: 4.2014 - val_accuracy: 0.1065
Epoch 2/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0321 -
accuracy: 0.9899 - val_loss: 0.4013 - val_accuracy: 0.8586
Epoch 3/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0182 -
accuracy: 0.9940 - val_loss: 0.0565 - val_accuracy: 0.9809
Epoch 4/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0110 -
accuracy: 0.9965 - val_loss: 0.0491 - val_accuracy: 0.9865
Epoch 5/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0067 -
accuracy: 0.9982 - val_loss: 0.0398 - val_accuracy: 0.9884
Epoch 6/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0042 -
accuracy: 0.9990 - val_loss: 0.0450 - val_accuracy: 0.9876
Epoch 7/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0034 -
accuracy: 0.9993 - val_loss: 0.0410 - val_accuracy: 0.9886
Epoch 8/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0062 -
accuracy: 0.9981 - val_loss: 0.0717 - val_accuracy: 0.9818
Epoch 9/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0125 -
accuracy: 0.9959 - val_loss: 0.0783 - val_accuracy: 0.9804
Epoch 10/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0073 -
accuracy: 0.9975 - val_loss: 0.0451 - val_accuracy: 0.9892
Epoch 11/20
50400/50400 [==============================] - 1s 28us/step - loss: 0.0031 -
accuracy: 0.9992 - val_loss: 0.0390 - val_accuracy: 0.9901
Epoch 12/20
50400/50400 [==============================] - 1s 28us/step - loss: 7.1629e-0
4 - accuracy: 0.9999 - val_loss: 0.0335 - val_accuracy: 0.9911
Epoch 13/20
50400/50400 [==============================] - 1s 28us/step - loss: 2.4544e-0
4 - accuracy: 1.0000 - val_loss: 0.0319 - val_accuracy: 0.9920
Epoch 14/20
50400/50400 [==============================] - 1s 28us/step - loss: 1.7695e-0
4 - accuracy: 1.0000 - val_loss: 0.0328 - val_accuracy: 0.9921
Epoch 15/20
50400/50400 [==============================] - 1s 28us/step - loss: 1.2038e-0
4 - accuracy: 1.0000 - val_loss: 0.0325 - val_accuracy: 0.9923
Epoch 16/20
50400/50400 [==============================] - 1s 29us/step - loss: 1.0140e-0
4 - accuracy: 1.0000 - val_loss: 0.0330 - val_accuracy: 0.9926
Epoch 17/20
50400/50400 [==============================] - 1s 28us/step - loss: 9.0561e-0
5 - accuracy: 1.0000 - val_loss: 0.0336 - val_accuracy: 0.9928
Epoch 18/20
50400/50400 [==============================] - 1s 28us/step - loss: 7.2238e-0
5 - accuracy: 1.0000 - val_loss: 0.0338 - val_accuracy: 0.9924
Epoch 19/20
50400/50400 [==============================] - 1s 28us/step - loss: 6.3028e-0
5 - accuracy: 1.0000 - val_loss: 0.0339 - val_accuracy: 0.9923
Epoch 20/20
50400/50400 [==============================] - 1s 28us/step - loss: 5.3360e-0
5 - accuracy: 1.0000 - val_loss: 0.0346 - val_accuracy: 0.9922
```

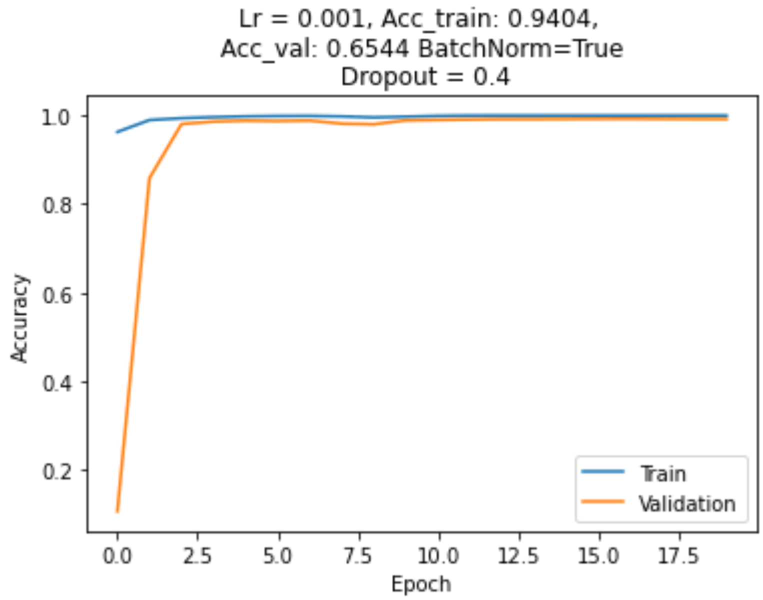
Plotting the loss function

In [15]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Lr = 0.001, loss_train: 0.1894, \n loss_val: 1.5591, BatchNorm=Tru
plt.ylabel('Cost')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
#plt.ylim(top=13)
#plt.ylim(bottom=0)
plt.show()
```



Plotting the accuracy

In [16]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Lr = 0.001, Acc_train: 0.9404, \n Acc_val: 0.6544 BatchNorm=True \
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='lower right')
plt.show()
```



In [17]:
```
# Predicting the image associated to the each sample in the test set (X_test)
predictions = LeNet_model.predict(x_test)
```

In [18]:
```
print(type(predictions))
print(predictions.shape)
```

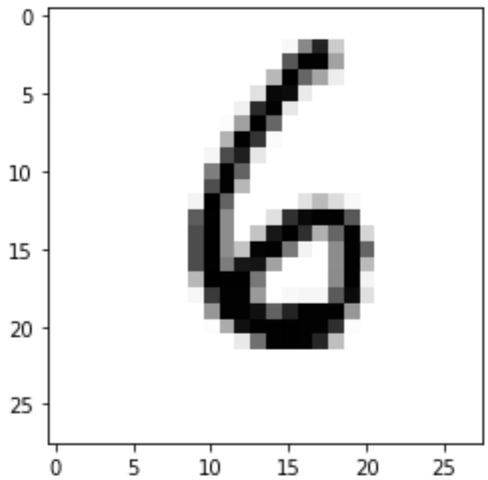
```
<class 'numpy.ndarray'>
(10000, 10)
```

In [19]:
```
sample = 91
print(predictions[sample])
print("\nPredicted digit:", np.argmax(predictions[sample]))
```

```
[5.7965465e-13 1.8259214e-12 4.5605764e-10 4.8307799e-12 3.7656635e-11
 6.1520482e-09 1.0000000e+00 1.1678162e-16 1.6059101e-10 9.9855061e-14]

Predicted digit: 6
```

Displaying the image associated to this sample.

In [20]:
```
plt.imshow(x_test[sample], cmap='Greys')
plt.show()
```

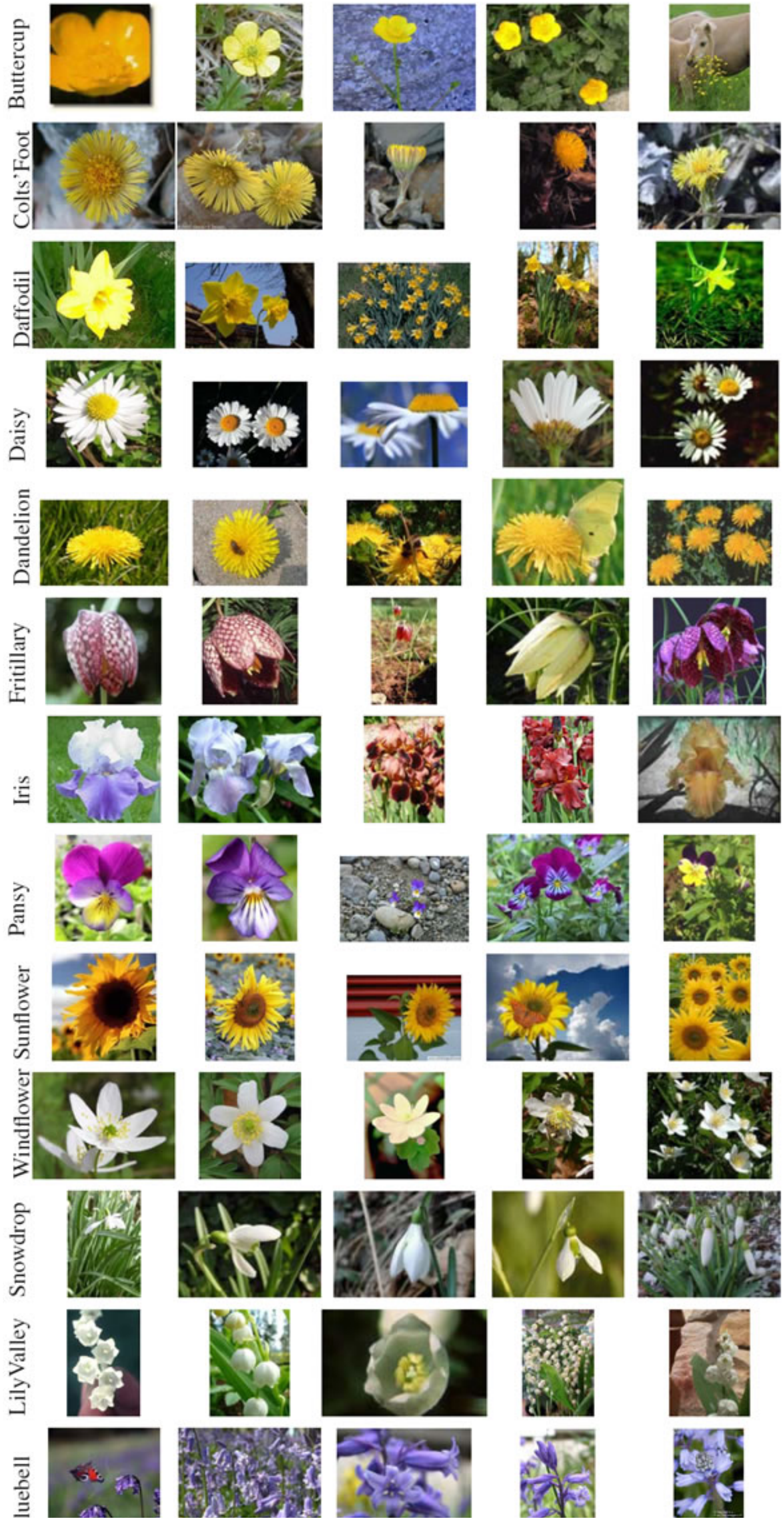


Deep Learning: AlexNet

In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
import time

from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Activation, Dense, Flatten
from keras.layers import Activation, Dropout, BatchNormalization
from keras.utils import plot_model
from keras import optimizers

np.random.seed(10)
```

Data of the System to be analyzed: oxflowers17

The oxflowers17 database


Buttercup
Colts'Foot
Daffodil
Daisy
Dandelion
Fritillary
Iris
Pansy
Sunflower
Windflower
Snowdrop
LilyValley
luebell

Generation or extraction of the raw data

Install the library tflearn to get the data.

TFLearn library

In [22]:
```
import tflearn.datasets.oxflower17 as oxflower17
train_x, train_y = oxflower17.load_data(one_hot=True)
```

```
WARNING:tensorflow:From /home/bokhimi/anaconda3/envs/tf-gpu/lib/python3.8/sit
e-packages/tensorflow/python/compat/v2_compat.py:96: disable_resource_variabl
es (from tensorflow.python.ops.variable_scope) is deprecated and will be remo
ved in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
```

Analysis of the raw data

The oxflower17 dataset consists of 1360 colour images (224 pixels high and 224 pixes width) of flowers in 17 classes, with 80 images per class. All images will be used for training. Before running the model, it will be indicated the ratio of samples that will be used for validation.

The 17 classes are:

index	class name
0	Daffodil
1	Snowdrop
2	Daisy
3	ColtsFoot
4	Dandelion
5	Cowslip
6	Buttercup
7	Windflower
8	Pansy
9	LilyValley
10	Bluebell
11	Crocus
12	Iris

index	class name
13	Tigerlily
14	Tulip
15	Fritillary

Viewing one sample from the data sets

We define a dictionary to associate the class number to a class name.

In [23]:
```
dic = {0: 'Daffodil', 1: 'Snowdrop', 2: 'Daisy', 3: 'ColtsFoot', 4: 'Dandelio
       5: 'Cowslip', 6: 'Buttercup', 7: 'Windflower', 8: 'Pansy', 9:'LilyVall
       10: 'Bluebell', 11: 'Crocus', 12: 'Iris', 13: 'Tigerlily', 14:'Tulip',
       15: 'Fritillary', 16: 'Sunflower'}
```

Next, we show a sample: its target and image.

In [24]:
```
# Plotting the content of a sample

sample = 72

plt.imshow(train_x[sample]);
print('y =',  np.squeeze(train_y[sample]))

for i in [i for i,x in enumerate(train_y[sample]) if x == 1]:
    print('')

print('y =',  i, ';', 'the sample', sample, 'corresponds to a(an)', dic[i])
```

```
y = [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

y = 0 ; the sample 72 corresponds to a(an) Daffodil
```

Transformation of the raw data

In [25]:
```
print('the shape is', train_x.shape)
```

```
the shape is (1360, 224, 224, 3)
```

In [26]:
```
print(train_x[0][0:5][0:2])
```

```
[[[0.06666667 0.05882353 0.10980392]
  [0.09019608 0.08235294 0.13333334]
  [0.13333334 0.1254902  0.1764706 ]
  ...
  [0.31764707 0.31764707 0.31764707]
  [0.30980393 0.30980393 0.30980393]
  [0.29411766 0.29411766 0.29411766]]

 [[0.12941177 0.12156863 0.17254902]
  [0.10588235 0.09803922 0.14901961]
  [0.09803922 0.09019608 0.14117648]
  ...
  [0.30980393 0.30980393 0.30980393]
  [0.23529412 0.23529412 0.23529412]
  [0.24313726 0.24313726 0.24313726]]]
```

In [27]:
```
print(train_y[0])
```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Transformation of the raw data

The raw data are jet renormalized. We do not do anything more

In [28]:
```
print('train_x shape:', train_x.shape)
print('train_y shape:', train_y.shape)
```

```
train_x shape: (1360, 224, 224, 3)
train_y shape: (1360, 17)
```

In [29]:
```
from sklearn.model_selection import train_test_split

# Choose your test size to split between training and testing sets:
train_x, test_x, train_y, test_y = train_test_split(train_x,train_y, test_siz
```

In [30]:
```
print(train_x.shape)
print(test_x.shape)
print(train_y.shape)
print(test_y.shape)
```

```
(1224, 224, 224, 3)
(136, 224, 224, 3)
(1224, 17)
(136, 17)
```

Definition of the neural network architecture

Keras has two different modes to define the architecture:

1. The sequential model. It is a sequential stack of layers.

2. The functional API. It is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

In the present case, we will use the sequential mode for constructing the architecture of the network.

Keras: Sequential model API

Keras: Convolutional layers

Keras: Pooling layers)

Keras: Batch Normalization

In [31]:
```
# Creating a Sequential Model and adding the layers

def architecture(batch_normalization, dropout, input_shape, activation):

    # Creating a sequential model
    model = Sequential()

    # 1st Convolutional layer
    model.add(Conv2D(filters=96, activation=activation, input_shape=input_sha
      kernel_size=(11,11), strides=(4,4), padding='valid', kernel_initializer
    # Pooling
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='valid'))
    if batch_normalization:
        model.add(BatchNormalization())

    # 2nd Convolutional Layer
    model.add(Conv2D(filters=256, activation=activation, kernel_size=(5,5), \
                     strides=(1,1), padding='valid'))
    # Pooling
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='valid'))
    if batch_normalization:
        model.add(BatchNormalization())

    # 3rd Convolutional Layer
    model.add(Conv2D(filters=384, activation=activation, kernel_size=(3,3), s

    # 4th Convolutional Layer
    model.add(Conv2D(filters=384, activation=activation, kernel_size=(3,3), s

    # 5th Convolutional Layer
    model.add(Conv2D(filters=256, activation=activation, kernel_size=(3,3), s
    # Pooling
    model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='valid'))
    if batch_normalization:
        model.add(BatchNormalization())

    # Passing it to a dense layer
    model.add(Flatten())
    if dropout:
        model.add(Dropout(0.4))

    # 1st Dense Layer
    model.add(Dense(512, activation=activation, input_shape=(224*224*3,), ker
    # Add Dropout to prevent overfitting
    if dropout:
        model.add(Dropout(0.4))
    if batch_normalization:
        model.add(BatchNormalization())

    # 2nd Dense Layer
    model.add(Dense(512, activation=activation, kernel_initializer = 'he_unif
    model.add(Activation('relu'))
    # Add Dropout
    if dropout:
        model.add(Dropout(0.4))
    if batch_normalization:
        model.add(BatchNormalization())
```

Generating a model of deep neural network

In [32]:

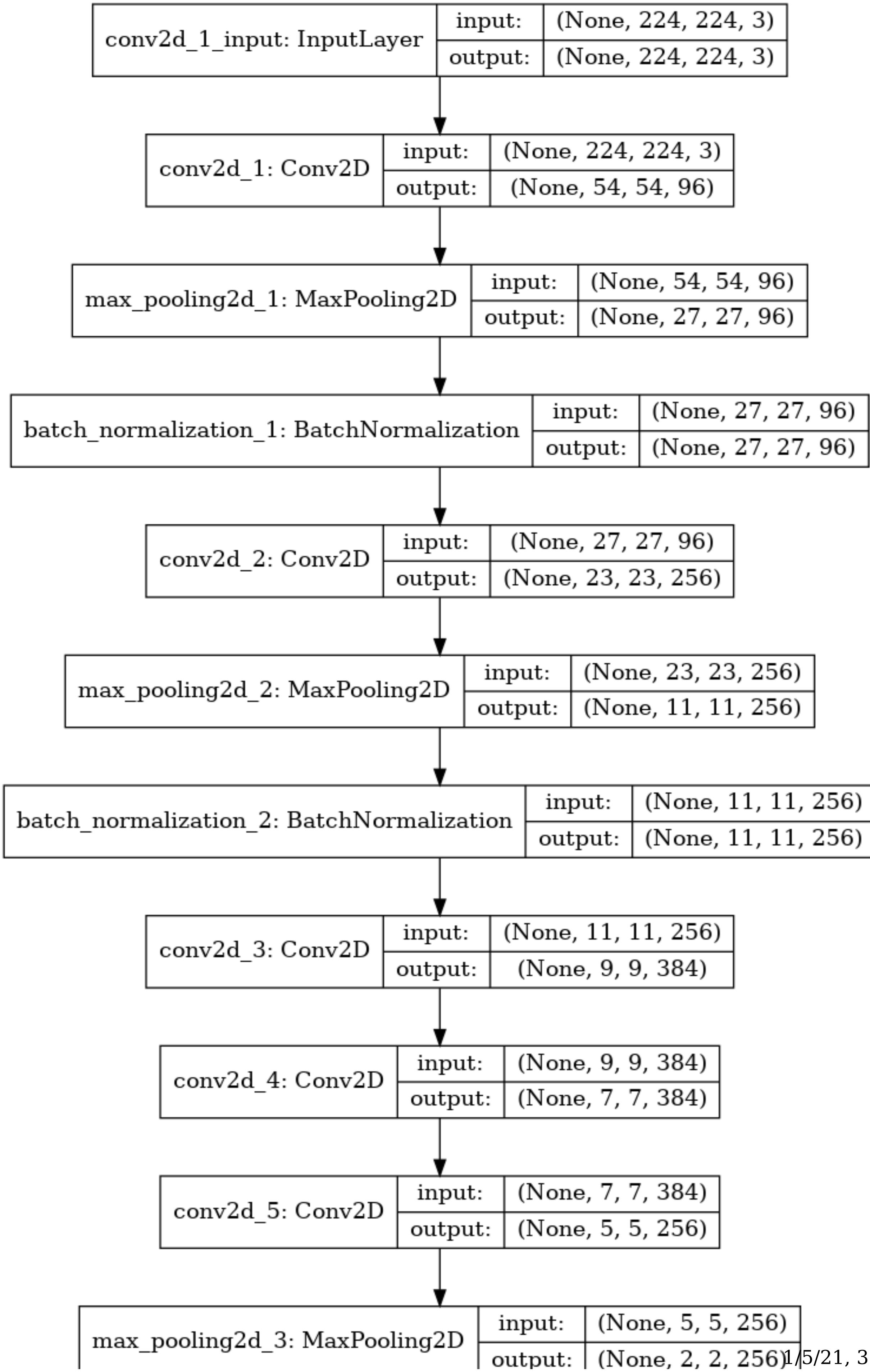
```
# Generating the model using the defined architecture

batch_normalization=True
dropout=True
one_image = (224, 224, 3)
activation = 'relu'

oxflower17_model = architecture(batch_normalization, dropout, one_image, acti
```

```
WARNING:tensorflow:From /home/bokhimi/anaconda3/envs/tf-gpu/lib/python3.8/sit
e-packages/tensorflow/python/ops/resource_variable_ops.py:1659: calling BaseR
esourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops) w
ith constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
```

In [33]:

```
plot_model(oxflower17_model, to_file='oxflower17_model.png', show_shapes=True
```

Out[33]:


conv2d_1_input: InputLayer
input: (None, 224, 224, 3)
output: (None, 224, 224, 3)
conv2d_1: Conv2D
input: (None, 224, 224, 3)
output: (None, 54, 54, 96)
max_pooling2d_1: MaxPooling2D
input: (None, 54, 54, 96)
output: (None, 27, 27, 96)
batch_normalization_1: BatchNormalization
input: (None, 27, 27, 96)
output: (None, 27, 27, 96)
conv2d_2: Conv2D
input: (None, 27, 27, 96)
output: (None, 23, 23, 256)
max_pooling2d_2: MaxPooling2D
input: (None, 23, 23, 256)
output: (None, 11, 11, 256)
batch_normalization_2: BatchNormalization
input: (None, 11, 11, 256)
output: (None, 11, 11, 256)
conv2d_3: Conv2D
input: (None, 11, 11, 256)
output: (None, 9, 9, 384)
conv2d_4: Conv2D
input: (None, 9, 9, 384)
output: (None, 7, 7, 384)
conv2d_5: Conv2D
input: (None, 7, 7, 384)
output: (None, 5, 5, 256)
max_pooling2d_3: MaxPooling2D
input: (None, 5, 5, 256)
output: (None, 2, 2, 256)

In [34]:
```
oxflower17_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 54, 54, 96)        34944     
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 27, 27, 96)        0         
_________________________________________________________________
batch_normalization_1 (Batch (None, 27, 27, 96)        384       
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 23, 23, 256)       614656    
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 11, 11, 256)       0         
_________________________________________________________________
batch_normalization_2 (Batch (None, 11, 11, 256)       1024      
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 9, 9, 384)         885120    
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 7, 7, 384)         1327488   
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 5, 5, 256)         884992    
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 2, 2, 256)         0         
_________________________________________________________________
batch_normalization_3 (Batch (None, 2, 2, 256)         1024      
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800    
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0         
_________________________________________________________________
batch_normalization_4 (Batch (None, 512)               2048      
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656    
_________________________________________________________________
activation_1 (Activation)    (None, 512)               0         
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0         
_________________________________________________________________
batch_normalization_5 (Batch (None, 512)               2048      
_________________________________________________________________
dense_3 (Dense)              (None, 17)                8721      
=================================================================
Total params: 4,549,905
Trainable params: 4,546,641
Non-trainable params: 3,264
_________________________________________________________________
```

Compiling the model

In [35]:
```
#Compiling the model using Adam as optimizer

lr = 0.001  # Learning rate

oxflower17_model.compile(loss='categorical_crossentropy', metrics=['accuracy'
optimizer=optimizers.Adam(learning_rate=lr,beta_1=0.9, beta_2=0.999, amsgrad=
```

Running the model

In [36]:
```
start_time = time.time()

batch_size=32
num_epochs = 50

history = oxflower17_model.fit(train_x, train_y, batch_size=batch_size,\
       epochs=num_epochs, validation_data=(test_x,test_y),verbose=1, shuffle=


end_time = time.time()
print("Time for training: {:10.4f}s".format(end_time - start_time))
```

```
Train on 1224 samples, validate on 136 samples
Epoch 1/50
1224/1224 [==============================] - 2s 2ms/step - loss: 2.9391 - acc
uracy: 0.1462 - val_loss: 40.2617 - val_accuracy: 0.0441
Epoch 2/50
1224/1224 [==============================] - 1s 1ms/step - loss: 2.4698 - acc
uracy: 0.2451 - val_loss: 5.7297 - val_accuracy: 0.1471
Epoch 3/50
1224/1224 [==============================] - 1s 1ms/step - loss: 2.0694 - acc
uracy: 0.3619 - val_loss: 4.0435 - val_accuracy: 0.2647
Epoch 4/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.9486 - acc
uracy: 0.3660 - val_loss: 2.3372 - val_accuracy: 0.2868
Epoch 5/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.8283 - acc
uracy: 0.4044 - val_loss: 2.5977 - val_accuracy: 0.3015
Epoch 6/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.7595 - acc
uracy: 0.4257 - val_loss: 1.9638 - val_accuracy: 0.4559
Epoch 7/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.6529 - acc
uracy: 0.4371 - val_loss: 1.9267 - val_accuracy: 0.4191
Epoch 8/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.5097 - acc
uracy: 0.4959 - val_loss: 2.0047 - val_accuracy: 0.4706
Epoch 9/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.4617 - acc
uracy: 0.5016 - val_loss: 2.3358 - val_accuracy: 0.3235
Epoch 10/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.4467 - acc
uracy: 0.5049 - val_loss: 2.1216 - val_accuracy: 0.3676
Epoch 11/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.4191 - acc
uracy: 0.5204 - val_loss: 1.6400 - val_accuracy: 0.4779
Epoch 12/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.3123 - acc
```

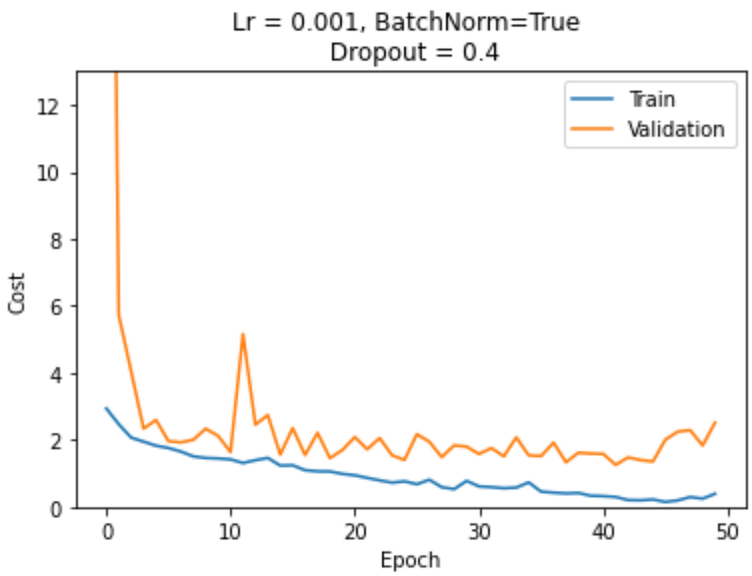
```
uracy: 0.5556 - val_loss: 5.1617 - val_accuracy: 0.1691
Epoch 13/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.3954 - acc
uracy: 0.5172 - val_loss: 2.4555 - val_accuracy: 0.3529
Epoch 14/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.4642 - acc
uracy: 0.5131 - val_loss: 2.7454 - val_accuracy: 0.3235
Epoch 15/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.2348 - acc
uracy: 0.5662 - val_loss: 1.5613 - val_accuracy: 0.5662
Epoch 16/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.2445 - acc
uracy: 0.5801 - val_loss: 2.3547 - val_accuracy: 0.3971
Epoch 17/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.0969 - acc
uracy: 0.6291 - val_loss: 1.5511 - val_accuracy: 0.4559
Epoch 18/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.0636 - acc
uracy: 0.6389 - val_loss: 2.2117 - val_accuracy: 0.4706
Epoch 19/50
1224/1224 [==============================] - 1s 1ms/step - loss: 1.0597 - acc
uracy: 0.6291 - val_loss: 1.4577 - val_accuracy: 0.5221
Epoch 20/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.9872 - acc
uracy: 0.6560 - val_loss: 1.6981 - val_accuracy: 0.5147
Epoch 21/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.9452 - acc
uracy: 0.6871 - val_loss: 2.0846 - val_accuracy: 0.4118
Epoch 22/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.8678 - acc
uracy: 0.7059 - val_loss: 1.7226 - val_accuracy: 0.4926
Epoch 23/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.7931 - acc
uracy: 0.7181 - val_loss: 2.0558 - val_accuracy: 0.4559
Epoch 24/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.7258 - acc
uracy: 0.7443 - val_loss: 1.5422 - val_accuracy: 0.5368
Epoch 25/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.7646 - acc
uracy: 0.7279 - val_loss: 1.4043 - val_accuracy: 0.5588
Epoch 26/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.6794 - acc
uracy: 0.7508 - val_loss: 2.1713 - val_accuracy: 0.4265
Epoch 27/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.8126 - acc
uracy: 0.7288 - val_loss: 1.9466 - val_accuracy: 0.4265
Epoch 28/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.5905 - acc
uracy: 0.7917 - val_loss: 1.4886 - val_accuracy: 0.5956
Epoch 29/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.5326 - acc
uracy: 0.8276 - val_loss: 1.8373 - val_accuracy: 0.5368
Epoch 30/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.7779 - acc
uracy: 0.7484 - val_loss: 1.8009 - val_accuracy: 0.4706
Epoch 31/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.6135 - acc
uracy: 0.7884 - val_loss: 1.5818 - val_accuracy: 0.5368
Epoch 32/50
```

```
1224/1224 [==============================] - 1s 1ms/step - loss: 0.5922 - acc
uracy: 0.7917 - val_loss: 1.7567 - val_accuracy: 0.5000
Epoch 33/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.5589 - acc
uracy: 0.8047 - val_loss: 1.5121 - val_accuracy: 0.5662
Epoch 34/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.5783 - acc
uracy: 0.7998 - val_loss: 2.0773 - val_accuracy: 0.5662
Epoch 35/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.7335 - acc
uracy: 0.7557 - val_loss: 1.5412 - val_accuracy: 0.5882
Epoch 36/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.4572 - acc
uracy: 0.8480 - val_loss: 1.5193 - val_accuracy: 0.5735
Epoch 37/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.4279 - acc
uracy: 0.8578 - val_loss: 1.9197 - val_accuracy: 0.5588
Epoch 38/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.4046 - acc
uracy: 0.8529 - val_loss: 1.3339 - val_accuracy: 0.6176
Epoch 39/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.4177 - acc
uracy: 0.8546 - val_loss: 1.6187 - val_accuracy: 0.5588
Epoch 40/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.3340 - acc
uracy: 0.8946 - val_loss: 1.5935 - val_accuracy: 0.5074
Epoch 41/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.3233 - acc
uracy: 0.8815 - val_loss: 1.5850 - val_accuracy: 0.6103
Epoch 42/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.2993 - acc
uracy: 0.9101 - val_loss: 1.2564 - val_accuracy: 0.6103
Epoch 43/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.2061 - acc
uracy: 0.9346 - val_loss: 1.4779 - val_accuracy: 0.6103
Epoch 44/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.1994 - acc
uracy: 0.9314 - val_loss: 1.4005 - val_accuracy: 0.7132
Epoch 45/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.2234 - acc
uracy: 0.9289 - val_loss: 1.3579 - val_accuracy: 0.6324
Epoch 46/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.1498 - acc
uracy: 0.9518 - val_loss: 2.0061 - val_accuracy: 0.5515
Epoch 47/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.1963 - acc
uracy: 0.9322 - val_loss: 2.2492 - val_accuracy: 0.5368
Epoch 48/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.2955 - acc
uracy: 0.9118 - val_loss: 2.2917 - val_accuracy: 0.4926
Epoch 49/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.2477 - acc
uracy: 0.9232 - val_loss: 1.8336 - val_accuracy: 0.6618
Epoch 50/50
1224/1224 [==============================] - 1s 1ms/step - loss: 0.3940 - acc
uracy: 0.8791 - val_loss: 2.5183 - val_accuracy: 0.5147
```

- Note: if you run `fit()` again, the `model` will continue training, starting with the parameters it has already learnt, instead of reinitializing them.

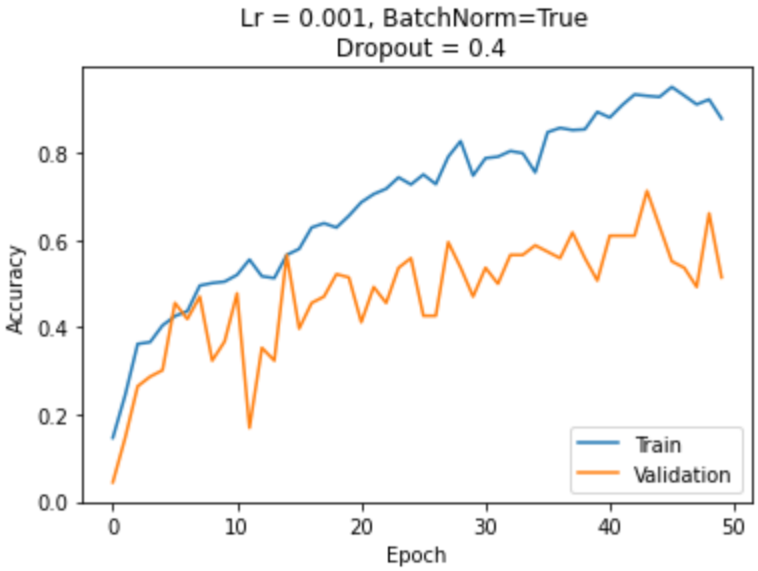
Plotting the loss function

In [37]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Lr = 0.001, BatchNorm=True \n Dropout = 0.4')
plt.ylabel('Cost')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.ylim(top=13)    # The instruction is used to limit the upper value of the
plt.ylim(bottom=0)  # The instruction is used to limit the lower value of the
plt.show()
```



Plotting the accuracy

In [38]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Lr = 0.001, BatchNorm=True \n Dropout = 0.4')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='lower right')
plt.show()
```



Data augmentation

shear_range, zoom_range, and horizontal_flip are some of the parameter available in Keras that define the transformation of the images

[Keras: Data augmetation]

Generating a model of deep neural network

In [39]:
```
# Generating the model using the defined architecture

batch_normalization=True
dropout=True
one_image = (224, 224, 3)
activation = 'relu'

oxflower17_model = architecture(batch_normalization, dropout, one_image, acti
```

Compiling the model

In [40]:
```
#Compiling the model using Adam as optimizer

lr = 0.001  # Learning rate

oxflower17_model.compile(loss='categorical_crossentropy', metrics=['accuracy'
optimizer=optimizers.Adam(learning_rate=lr,beta_1=0.9, beta_2=0.999, amsgrad=
```

In [41]:
```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
    shear_range = 0.2,
    zoom_range = 0.2,
    horizontal_flip = True
)
```

Running the model

Comment: Keras flow method

This process requires long times, depending of the number of steps per epoch, the number of epochs and the number of images that will be generated during the data augmentation (batch_size)

In [42]:
```
# compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied)
train_datagen.fit(train_x)

train_generator = train_datagen.flow(
    train_x,
    train_y,
    batch_size = 32,
    shuffle=True
)
```

In [43]:
```
get_steps_augment = 64

print ("X_train shape: " + str(train_x.shape[0]))
steps = int(train_x.shape[0]/get_steps_augment)
print("Augmentation steps = {}".format(steps))
```

```
X_train shape: 1224
Augmentation steps = 19
```

In [44]:
```
start_time = time.time()

num_epochs = 100

history = oxflower17_model.fit(train_generator, steps_per_epoch=steps,\
        epochs=num_epochs, validation_data=(test_x, test_y), verbose=1, shuff


end_time = time.time()
print("Time for training: {:10.4f}s".format(end_time - start_time))
```

```
Epoch 1/100
19/19 [==============================] - 3s 176ms/step - loss: 3.1353 - accur
acy: 0.1168 - val_loss: 44.1296 - val_accuracy: 0.0515
Epoch 2/100
19/19 [==============================] - 3s 184ms/step - loss: 2.6787 - accur
acy: 0.1764 - val_loss: 23.8188 - val_accuracy: 0.1029
```

```
Epoch 3/100
19/19 [==============================] - 3s 177ms/step - loss: 2.5314 - accur
acy: 0.2346 - val_loss: 8.1675 - val_accuracy: 0.1471
Epoch 4/100
19/19 [==============================] - 3s 182ms/step - loss: 2.3450 - accur
acy: 0.2681 - val_loss: 6.4789 - val_accuracy: 0.1324
Epoch 5/100
19/19 [==============================] - 4s 188ms/step - loss: 2.1867 - accur
acy: 0.3240 - val_loss: 4.2634 - val_accuracy: 0.1691
Epoch 6/100
19/19 [==============================] - 3s 179ms/step - loss: 2.0995 - accur
acy: 0.3373 - val_loss: 2.7618 - val_accuracy: 0.2426
Epoch 7/100
19/19 [==============================] - 4s 188ms/step - loss: 2.1194 - accur
acy: 0.3141 - val_loss: 2.4809 - val_accuracy: 0.2868
Epoch 8/100
19/19 [==============================] - 3s 178ms/step - loss: 2.1393 - accur
acy: 0.3339 - val_loss: 2.3831 - val_accuracy: 0.3162
Epoch 9/100
19/19 [==============================] - 3s 176ms/step - loss: 1.9776 - accur
acy: 0.3613 - val_loss: 1.7023 - val_accuracy: 0.4779
Epoch 10/100
19/19 [==============================] - 3s 177ms/step - loss: 1.8667 - accur
acy: 0.3832 - val_loss: 1.9932 - val_accuracy: 0.3676
Epoch 11/100
19/19 [==============================] - 4s 190ms/step - loss: 1.8851 - accur
acy: 0.3832 - val_loss: 1.9633 - val_accuracy: 0.3897
Epoch 12/100
19/19 [==============================] - 3s 173ms/step - loss: 1.7090 - accur
acy: 0.4195 - val_loss: 2.2833 - val_accuracy: 0.3015
Epoch 13/100
19/19 [==============================] - 3s 181ms/step - loss: 1.8009 - accur
acy: 0.3921 - val_loss: 2.4853 - val_accuracy: 0.2647
Epoch 14/100
19/19 [==============================] - 4s 184ms/step - loss: 1.7312 - accur
acy: 0.4095 - val_loss: 2.4010 - val_accuracy: 0.3456
Epoch 15/100
19/19 [==============================] - 3s 182ms/step - loss: 1.6900 - accur
acy: 0.4161 - val_loss: 2.1207 - val_accuracy: 0.3676
Epoch 16/100
19/19 [==============================] - 3s 178ms/step - loss: 1.6009 - accur
acy: 0.4391 - val_loss: 1.6227 - val_accuracy: 0.4412
Epoch 17/100
19/19 [==============================] - 3s 167ms/step - loss: 1.8838 - accur
acy: 0.4304 - val_loss: 2.3893 - val_accuracy: 0.3750
Epoch 18/100
19/19 [==============================] - 3s 184ms/step - loss: 1.8702 - accur
acy: 0.3799 - val_loss: 2.2472 - val_accuracy: 0.3603
Epoch 19/100
19/19 [==============================] - 4s 204ms/step - loss: 1.5802 - accur
acy: 0.4523 - val_loss: 2.1612 - val_accuracy: 0.3971
Epoch 20/100
19/19 [==============================] - 3s 179ms/step - loss: 1.6811 - accur
acy: 0.4401 - val_loss: 1.8556 - val_accuracy: 0.4706
Epoch 21/100
19/19 [==============================] - 3s 180ms/step - loss: 1.6443 - accur
acy: 0.4293 - val_loss: 1.9464 - val_accuracy: 0.4632
Epoch 22/100
19/19 [==============================] - 3s 176ms/step - loss: 1.5040 - accur
```

```
acy: 0.4984 - val_loss: 1.8450 - val_accuracy: 0.4706
Epoch 23/100
19/19 [==============================] - 3s 173ms/step - loss: 1.5123 - accur
acy: 0.4880 - val_loss: 2.0540 - val_accuracy: 0.3382
Epoch 24/100
19/19 [==============================] - 3s 177ms/step - loss: 1.5558 - accur
acy: 0.4675 - val_loss: 2.4349 - val_accuracy: 0.3603
Epoch 25/100
19/19 [==============================] - 3s 179ms/step - loss: 1.5712 - accur
acy: 0.4589 - val_loss: 2.2818 - val_accuracy: 0.4265
Epoch 26/100
19/19 [==============================] - 4s 202ms/step - loss: 1.4107 - accur
acy: 0.5082 - val_loss: 2.5848 - val_accuracy: 0.3824
Epoch 27/100
19/19 [==============================] - 3s 182ms/step - loss: 1.4136 - accur
acy: 0.5214 - val_loss: 1.5447 - val_accuracy: 0.5221
Epoch 28/100
19/19 [==============================] - 3s 169ms/step - loss: 1.5443 - accur
acy: 0.4897 - val_loss: 2.1851 - val_accuracy: 0.3750
Epoch 29/100
19/19 [==============================] - 3s 177ms/step - loss: 1.3854 - accur
acy: 0.5378 - val_loss: 1.7810 - val_accuracy: 0.4338
Epoch 30/100
19/19 [==============================] - 3s 175ms/step - loss: 1.3356 - accur
acy: 0.5493 - val_loss: 1.8170 - val_accuracy: 0.4485
Epoch 31/100
19/19 [==============================] - 3s 171ms/step - loss: 1.3850 - accur
acy: 0.5154 - val_loss: 1.4959 - val_accuracy: 0.5368
Epoch 32/100
19/19 [==============================] - 3s 176ms/step - loss: 1.3862 - accur
acy: 0.5137 - val_loss: 4.7515 - val_accuracy: 0.2279
Epoch 33/100
19/19 [==============================] - 4s 186ms/step - loss: 1.3504 - accur
acy: 0.5296 - val_loss: 2.4606 - val_accuracy: 0.3897
Epoch 34/100
19/19 [==============================] - 3s 169ms/step - loss: 1.3227 - accur
acy: 0.5565 - val_loss: 1.7868 - val_accuracy: 0.4559
Epoch 35/100
19/19 [==============================] - 3s 181ms/step - loss: 1.3556 - accur
acy: 0.5428 - val_loss: 2.0456 - val_accuracy: 0.3750
Epoch 36/100
19/19 [==============================] - 3s 177ms/step - loss: 1.2548 - accur
acy: 0.5493 - val_loss: 2.9642 - val_accuracy: 0.3015
Epoch 37/100
19/19 [==============================] - 3s 171ms/step - loss: 1.3546 - accur
acy: 0.5325 - val_loss: 1.7061 - val_accuracy: 0.4779
Epoch 38/100
19/19 [==============================] - 3s 175ms/step - loss: 1.2781 - accur
acy: 0.5634 - val_loss: 3.0292 - val_accuracy: 0.3382
Epoch 39/100
19/19 [==============================] - 3s 182ms/step - loss: 1.2352 - accur
acy: 0.5757 - val_loss: 1.4557 - val_accuracy: 0.5147
Epoch 40/100
19/19 [==============================] - 4s 198ms/step - loss: 1.1643 - accur
acy: 0.6096 - val_loss: 1.2538 - val_accuracy: 0.5588
Epoch 41/100
19/19 [==============================] - 4s 185ms/step - loss: 1.2196 - accur
acy: 0.5855 - val_loss: 1.5317 - val_accuracy: 0.5147
Epoch 42/100
```

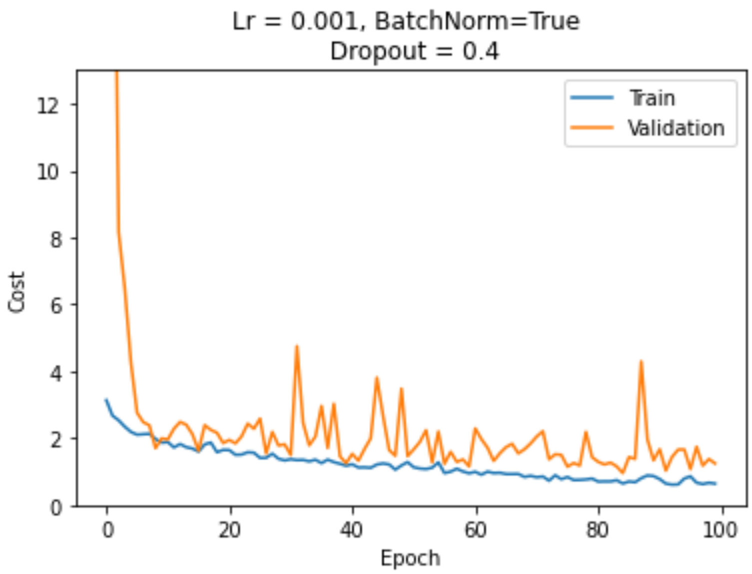
```
19/19 [==============================] - 3s 183ms/step - loss: 1.1232 - accur
acy: 0.6201 - val_loss: 1.3263 - val_accuracy: 0.5147
Epoch 43/100
19/19 [==============================] - 3s 182ms/step - loss: 1.1279 - accur
acy: 0.6151 - val_loss: 1.6795 - val_accuracy: 0.4632
Epoch 44/100
19/19 [==============================] - 3s 173ms/step - loss: 1.0944 - accur
acy: 0.6054 - val_loss: 1.9967 - val_accuracy: 0.4485
Epoch 45/100
19/19 [==============================] - 3s 184ms/step - loss: 1.2131 - accur
acy: 0.5789 - val_loss: 3.8101 - val_accuracy: 0.2132
Epoch 46/100
19/19 [==============================] - 3s 182ms/step - loss: 1.2451 - accur
acy: 0.5773 - val_loss: 2.6766 - val_accuracy: 0.3897
Epoch 47/100
19/19 [==============================] - 3s 174ms/step - loss: 1.2224 - accur
acy: 0.5771 - val_loss: 1.6436 - val_accuracy: 0.4853
Epoch 48/100
19/19 [==============================] - 3s 182ms/step - loss: 1.0597 - accur
acy: 0.6266 - val_loss: 1.4739 - val_accuracy: 0.5147
Epoch 49/100
19/19 [==============================] - 3s 167ms/step - loss: 1.1985 - accur
acy: 0.5993 - val_loss: 3.4810 - val_accuracy: 0.3015
Epoch 50/100
19/19 [==============================] - 3s 181ms/step - loss: 1.2855 - accur
acy: 0.5724 - val_loss: 1.4647 - val_accuracy: 0.5515
Epoch 51/100
19/19 [==============================] - 4s 196ms/step - loss: 1.1585 - accur
acy: 0.6096 - val_loss: 1.6666 - val_accuracy: 0.5000
Epoch 52/100
19/19 [==============================] - 3s 182ms/step - loss: 1.0927 - accur
acy: 0.6151 - val_loss: 1.8896 - val_accuracy: 0.4559
Epoch 53/100
19/19 [==============================] - 3s 175ms/step - loss: 1.1366 - accur
acy: 0.6233 - val_loss: 2.2382 - val_accuracy: 0.3676
Epoch 54/100
19/19 [==============================] - 3s 182ms/step - loss: 1.1186 - accur
acy: 0.5921 - val_loss: 1.2781 - val_accuracy: 0.5956
Epoch 55/100
19/19 [==============================] - 3s 168ms/step - loss: 1.3007 - accur
acy: 0.5942 - val_loss: 2.1996 - val_accuracy: 0.4265
Epoch 56/100
19/19 [==============================] - 3s 169ms/step - loss: 0.9481 - accur
acy: 0.6610 - val_loss: 1.2351 - val_accuracy: 0.6029
Epoch 57/100
19/19 [==============================] - 3s 174ms/step - loss: 0.9999 - accur
acy: 0.6842 - val_loss: 1.5923 - val_accuracy: 0.4779
Epoch 58/100
19/19 [==============================] - 3s 179ms/step - loss: 1.0877 - accur
acy: 0.6053 - val_loss: 1.2925 - val_accuracy: 0.5662
Epoch 59/100
19/19 [==============================] - 4s 202ms/step - loss: 1.0050 - accur
acy: 0.6464 - val_loss: 1.3629 - val_accuracy: 0.5735
Epoch 60/100
19/19 [==============================] - 3s 176ms/step - loss: 0.9799 - accur
acy: 0.6644 - val_loss: 1.1559 - val_accuracy: 0.6176
Epoch 61/100
19/19 [==============================] - 3s 169ms/step - loss: 0.9721 - accur
acy: 0.6592 - val_loss: 2.2908 - val_accuracy: 0.3676
```

```
Epoch 62/100
19/19 [==============================] - 3s 180ms/step - loss: 0.9081 - accur
acy: 0.7023 - val_loss: 1.9711 - val_accuracy: 0.4485
Epoch 63/100
19/19 [==============================] - 3s 168ms/step - loss: 1.0169 - accur
acy: 0.6421 - val_loss: 1.7153 - val_accuracy: 0.5074
Epoch 64/100
19/19 [==============================] - 3s 179ms/step - loss: 0.9569 - accur
acy: 0.6760 - val_loss: 1.3210 - val_accuracy: 0.5882
Epoch 65/100
19/19 [==============================] - 3s 184ms/step - loss: 0.9669 - accur
acy: 0.6859 - val_loss: 1.5475 - val_accuracy: 0.4779
Epoch 66/100
19/19 [==============================] - 3s 181ms/step - loss: 0.9368 - accur
acy: 0.6849 - val_loss: 1.7305 - val_accuracy: 0.4559
Epoch 67/100
19/19 [==============================] - 4s 185ms/step - loss: 0.9283 - accur
acy: 0.6711 - val_loss: 1.8319 - val_accuracy: 0.4485
Epoch 68/100
19/19 [==============================] - 3s 179ms/step - loss: 0.9256 - accur
acy: 0.7038 - val_loss: 1.5465 - val_accuracy: 0.5000
Epoch 69/100
19/19 [==============================] - 4s 192ms/step - loss: 0.8423 - accur
acy: 0.7072 - val_loss: 1.6679 - val_accuracy: 0.5000
Epoch 70/100
19/19 [==============================] - 3s 183ms/step - loss: 0.8748 - accur
acy: 0.6990 - val_loss: 1.8416 - val_accuracy: 0.4779
Epoch 71/100
19/19 [==============================] - 3s 172ms/step - loss: 0.8596 - accur
acy: 0.7072 - val_loss: 2.0492 - val_accuracy: 0.5074
Epoch 72/100
19/19 [==============================] - 3s 183ms/step - loss: 0.8557 - accur
acy: 0.7039 - val_loss: 2.2084 - val_accuracy: 0.4118
Epoch 73/100
19/19 [==============================] - 3s 168ms/step - loss: 0.7537 - accur
acy: 0.7534 - val_loss: 1.3678 - val_accuracy: 0.5221
Epoch 74/100
19/19 [==============================] - 3s 176ms/step - loss: 0.8908 - accur
acy: 0.6941 - val_loss: 1.5174 - val_accuracy: 0.5735
Epoch 75/100
19/19 [==============================] - 3s 179ms/step - loss: 0.7747 - accur
acy: 0.7401 - val_loss: 1.5004 - val_accuracy: 0.5735
Epoch 76/100
19/19 [==============================] - 3s 176ms/step - loss: 0.8416 - accur
acy: 0.7106 - val_loss: 1.1474 - val_accuracy: 0.6397
Epoch 77/100
19/19 [==============================] - 3s 173ms/step - loss: 0.7483 - accur
acy: 0.7204 - val_loss: 1.2569 - val_accuracy: 0.6250
Epoch 78/100
19/19 [==============================] - 3s 166ms/step - loss: 0.7368 - accur
acy: 0.7380 - val_loss: 1.1814 - val_accuracy: 0.6324
Epoch 79/100
19/19 [==============================] - 3s 177ms/step - loss: 0.8005 - accur
acy: 0.7432 - val_loss: 2.1879 - val_accuracy: 0.4485
Epoch 80/100
19/19 [==============================] - 3s 179ms/step - loss: 0.7923 - accur
acy: 0.7303 - val_loss: 1.4326 - val_accuracy: 0.5809
Epoch 81/100
19/19 [==============================] - 3s 179ms/step - loss: 0.6940 - accur
```

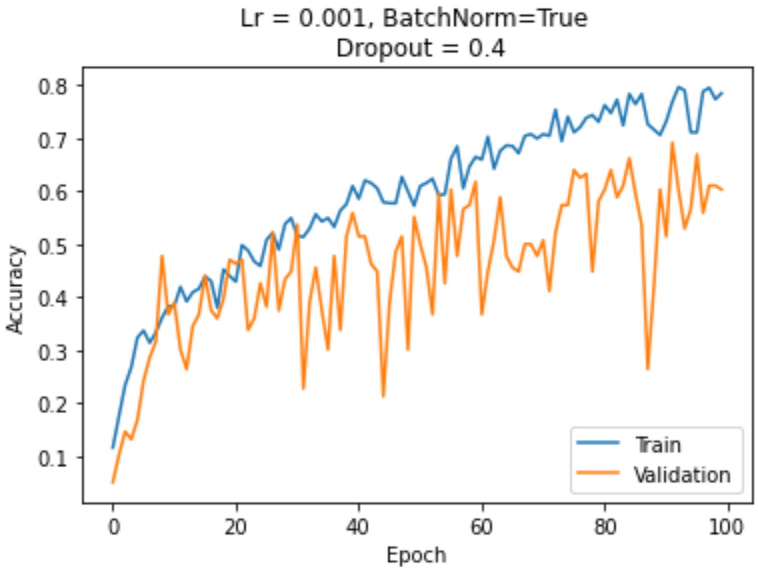
```
acy: 0.7620 - val_loss: 1.2920 - val_accuracy: 0.6029
Epoch 82/100
19/19 [==============================] - 3s 182ms/step - loss: 0.7084 - accur
acy: 0.7467 - val_loss: 1.2149 - val_accuracy: 0.6397
Epoch 83/100
19/19 [==============================] - 3s 173ms/step - loss: 0.7390 - accur
acy: 0.7723 - val_loss: 1.2683 - val_accuracy: 0.5882
Epoch 84/100
19/19 [==============================] - 3s 172ms/step - loss: 0.7429 - accur
acy: 0.7237 - val_loss: 1.1573 - val_accuracy: 0.6103
Epoch 85/100
19/19 [==============================] - 3s 181ms/step - loss: 0.6451 - accur
acy: 0.7829 - val_loss: 0.9571 - val_accuracy: 0.6618
Epoch 86/100
19/19 [==============================] - 3s 173ms/step - loss: 0.6972 - accur
acy: 0.7637 - val_loss: 1.4342 - val_accuracy: 0.5956
Epoch 87/100
19/19 [==============================] - 4s 194ms/step - loss: 0.6841 - accur
acy: 0.7829 - val_loss: 1.3808 - val_accuracy: 0.5368
Epoch 88/100
19/19 [==============================] - 3s 183ms/step - loss: 0.8232 - accur
acy: 0.7260 - val_loss: 4.2957 - val_accuracy: 0.2647
Epoch 89/100
19/19 [==============================] - 3s 173ms/step - loss: 0.9306 - accur
acy: 0.7158 - val_loss: 1.9639 - val_accuracy: 0.4191
Epoch 90/100
19/19 [==============================] - 3s 175ms/step - loss: 0.8674 - accur
acy: 0.7056 - val_loss: 1.3379 - val_accuracy: 0.6029
Epoch 91/100
19/19 [==============================] - 3s 178ms/step - loss: 0.7736 - accur
acy: 0.7312 - val_loss: 1.6693 - val_accuracy: 0.5147
Epoch 92/100
19/19 [==============================] - 3s 183ms/step - loss: 0.6425 - accur
acy: 0.7681 - val_loss: 1.0308 - val_accuracy: 0.6912
Epoch 93/100
19/19 [==============================] - 3s 179ms/step - loss: 0.6113 - accur
acy: 0.7961 - val_loss: 1.4377 - val_accuracy: 0.6029
Epoch 94/100
19/19 [==============================] - 3s 175ms/step - loss: 0.6179 - accur
acy: 0.7895 - val_loss: 1.6587 - val_accuracy: 0.5294
Epoch 95/100
19/19 [==============================] - 4s 189ms/step - loss: 0.8173 - accur
acy: 0.7106 - val_loss: 1.6605 - val_accuracy: 0.5662
Epoch 96/100
19/19 [==============================] - 3s 179ms/step - loss: 0.8470 - accur
acy: 0.7106 - val_loss: 1.0752 - val_accuracy: 0.6691
Epoch 97/100
19/19 [==============================] - 4s 185ms/step - loss: 0.6691 - accur
acy: 0.7878 - val_loss: 1.7435 - val_accuracy: 0.5588
Epoch 98/100
19/19 [==============================] - 3s 177ms/step - loss: 0.6627 - accur
acy: 0.7945 - val_loss: 1.1784 - val_accuracy: 0.6103
Epoch 99/100
19/19 [==============================] - 4s 187ms/step - loss: 0.6669 - accur
acy: 0.7730 - val_loss: 1.3755 - val_accuracy: 0.6103
Epoch 100/100
```

- Note: if you run `fit()` again, the `model` will continue training, starting with the parameters it has already learnt, instead of reinitializing them.

In [45]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Lr = 0.001, BatchNorm=True \n Dropout = 0.4')
plt.ylabel('Cost')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper right')
plt.ylim(top=13)    # The instruction is used to limit the upper value of the
plt.ylim(bottom=0)  # The instruction is used to limit the lower value of the
plt.show()
```



In [46]:
```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Lr = 0.001, BatchNorm=True \n Dropout = 0.4')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='lower right')
plt.show()
```



In [47]:
```
# Predicting the image associated to the each sample in the test set (X_test)
predictions = oxflower17_model.predict(test_x)
```

In [48]:
```
print(type(predictions))
print(predictions.shape)
```

```
<class 'numpy.ndarray'>
(136, 17)
```

In [49]:
```
# Predicting the image associated to the sample
# np.argmax returns the index of the maximum value
sample = 17
prediction = np.argmax(predictions[sample])
print("Prediction number=", prediction, ', it corresponds to a', dic[predicti


# Plotting the content of a sample

plt.imshow(train_x[sample]);
print('\ny =',  np.squeeze(train_y[sample]))

for i in [i for i,x in enumerate(train_y[sample]) if x == 1]:
    print('')

print('y =',  i, ';', 'the sample', sample, 'corresponds to a(an)', dic[i])
```

```
Prediction number= 3 , it corresponds to a ColtsFoot

y = [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

y = 3 ; the sample 17 corresponds to a(an) ColtsFoot
```

In [ ]: