Image Classification Using CNN Deep Learning Project

Abstract

Computer vision is evolving rapidly day-by-day. It's one of the reason is deep learning. When we talk about computer vision, a term convolutional neural network (abbreviated as CNN) comes in our mind because CNN is heavily used here. Examples of CNN in computer vision are face recognition, image classification etc. It is similar to the basic neural network. CNN also have learnable parameter like neural network i.e., weights, biases etc.

Introduction

The classification problem is to categorize all the pixels of a digital image into one of the defined classes. Image classification is the most critical use case in digital image analysis. Image classification is an application of both supervised classification and unsupervised classification. In supervised classification, we select samples for each target class. We train our neural_network on these target class samples and then classify new samples. In unsupervised classification, we group the sample images into clusters of images having similar properties. Then, we classify each cluster into our intended classes.

Data Description

This dataset contains 6,899 images from 8 distinct classes compiled from various sources. The classes include airplane, car, cat, dog, flower, fruit, motorbike and person.

The important points that distinguish this dataset from others are:

- Images are colored as compared to the black and white texture of MNIST
- Each image is 150 x 150 pixel
- 6899 training images and 2054 testing images

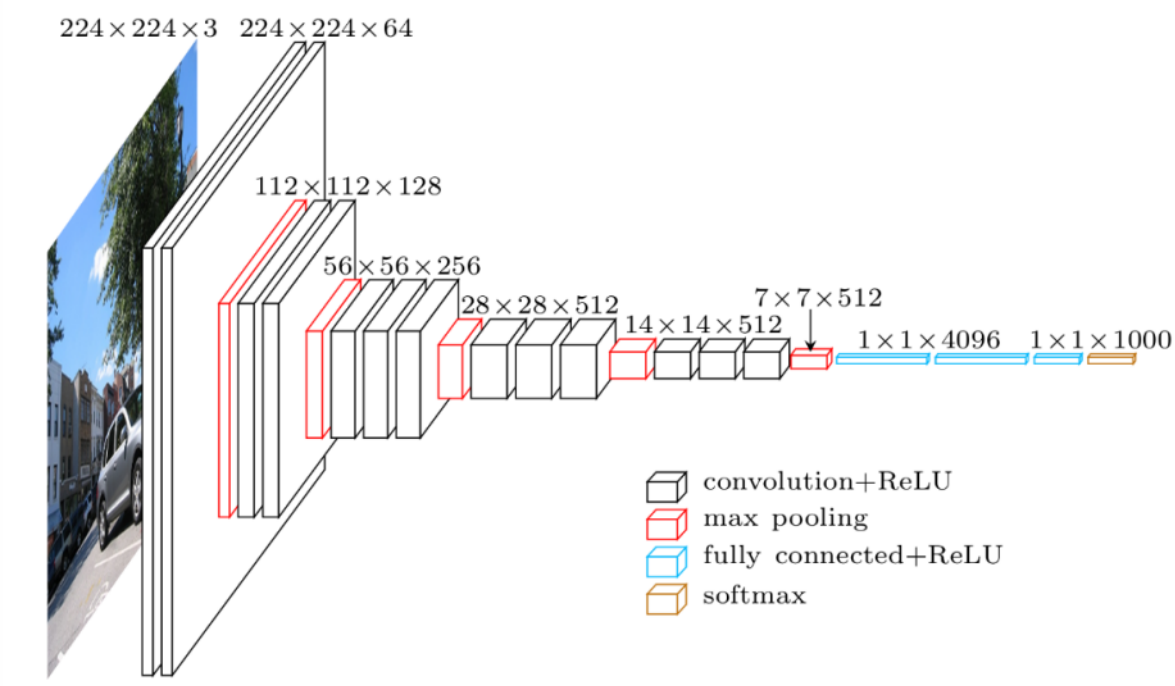
Image Classification

The complete image classification pipeline can be formalized as follows:

- Our input is a training dataset that consists of *N* images, each labelled with one of 8 different classes.
- Then, we use this training set to train a classifier to learn what every one of the classes looks like.
- In the end, we evaluate the quality of the classifier by asking it to predict labels for a new set of images that it has never seen before. We will then compare the true labels of these images to the ones predicted by the classifier.

Why Should we use CNN

Problem with Feedforward Neural Network
Suppose you are working with MNIST dataset, you know each image in MNIST is 28 x 28 x 1(black & white image contains only 1 channel). Total number of neurons in input layer will 28 x 28 = 784, this can be manageable. What if the size of image is 1000 x 1000 which means you need 10^6 neurons in input layer. Oh! This seems a huge number of neurons are required for operation. It is computationally ineffective right. So here comes Convolutional Neural Network or CNN. In simple word what CNN does is, it extract the feature of image and convert it into lower dimension without loosing its characteristics. In the following example you can see that initial the size of the image is 224 x 224 x 3. If you proceed without convolution then you need 224 x 224 x 3 = 100, 352 numbers of neurons in input layer but after applying convolution you input tensor dimension is reduced to 1 x 1 x 1000. It means you only need 1000 neurons in first layer of feedforward neural network.



Project Prerequisites

The prerequisite to develop and execute image classification project is Keras .

Steps for image classification using CNN:

1. Load the dataset from google drive

```
train_data_dir = '/content/drive/MyDrive/Datasets_classification/train'
validation_data_dir = '/content/drive/MyDrive/Datasets_classification/test'
```

2. Convert image matrix into vector to feed into first layer

As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size.

```
img_width, img_height = 150, 150
if K.image_data_format() == 'channels_first':
    input_shape = (3, img_width, img_height)
else:
    input_shape = (img_width, img_height, 3)
```

3. Import the required layers and modules to create our convolution neural net architecture

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense
from keras import backend as K
```

4. Create the sequential model and add the layers

The next step was to build the model. I used two convolutional blocks comprised of convolutional and max-pooling layer. I have used relu as the activation function for the convolutional layer. On top of it I used a flatten layer and followed it by two fully connected layers with relu and sigmoid as activation respectively.

The following lines of code below define the convolutional base using a common pattern: a stack of Conv2D and MaxPooling2D layers.

```
model = Sequential()
model.add(Conv2D(32, (2, 2), input_shape = input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size =(2, 2)))

model.add(Conv2D(32, (2, 2)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size =(2, 2)))

model.add(Conv2D(64, (2, 2)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size =(2, 2)))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(8))
model.add(Activation('sigmoid'))

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

We'll use the Sequential () function which is probably the easiest way to define a deep learning model in Keras. It lets you add layers on one by one. Use the Keras_Conv2D function to create a 2-dimensional convolutional layer. We use the activation function ReLu. Then use the MaxPooling2D function to add a 2D max pooling layer, with pooling filter sized 2X2. Add we add more convolution and pooling layers with 32 and 64 filters. Finally, flatten the output and define the fully_connected_layers. Add Dense layers on top

To complete our model, we will feed the last output tensor from the convolutional base (of shape (4, 4, 64)) into one or more Dense layers to perform classification. Dense layers take vectors as input (which are 1D), while the current output is a 3D tensor. First, you will flatten (or unroll) the 3D output to 1D, then add one or more Dense layers on top. Our dataset has 8 output classes, so you use a final Dense layer with 8 outputs and an activation.

Dropout

Another technique to reduce overfitting is to introduce Dropout to the network, a form of regularization.

When you apply Dropout to a layer it randomly drops out (by setting the activation to zero) a number of output units from the layer during the training process. Dropout takes a fractional number as its input value, in the form such as 0.1, 0.2, 0.4, etc. This means dropping out 10%, 20% or 40% of the output units randomly from the applied layer.

SoftMax / Logistic Layer

SoftMax or Logistic layer is the last layer of CNN. It resides at the end of FC layer. Logistic is used for binary classification and SoftMax is for multi-classification.

5. Configure the optimizer and compile the mode

After compiling our model, we will train our model by fit() method, then evaluate it.

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

l

We use rmsprop as the optimizer and cross entropy as the loss.

6. View the model summary for better understanding of model architecture

```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 149, 149, 32)      416
_________________________________________________________________
activation_10 (Activation)   (None, 149, 149, 32)      0
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 74, 74, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 73, 73, 32)        4128
_________________________________________________________________
activation_11 (Activation)   (None, 73, 73, 32)        0
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 36, 36, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 35, 35, 64)        8256
_________________________________________________________________
activation_12 (Activation)   (None, 35, 35, 64)        0
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 17, 17, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 18496)             0
_________________________________________________________________
dense_4 (Dense)              (None, 64)                1183808
_________________________________________________________________
activation_13 (Activation)   (None, 64)                0
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_5 (Dense)              (None, 8)                 520
_________________________________________________________________
activation_14 (Activation)   (None, 8)                 0
=================================================================
Total params: 1,197,128
Trainable params: 1,197,128
Non-trainable params: 0
```

Above, you can see that the output of every Conv2D and MaxPooling2D layer is a 3D tensor of shape (height, width, channels). The width and height dimensions tend to shrink as you go deeper in the network. The number of output channels for each Conv2D layer is controlled by the first argument (e.g., 32 or 64). Typically, as the width and height shrink, you can afford (computationally) to add more output channels in each Conv2D layer.

It's used to see all parameters and shapes in each layers in our models. You can observe that total parameters are 1,197,128 and total trainable parameters are 1,197,128. Non-trainable parameter is 0.

7. Data Augumentation

The practice of Data Augmentation is an effective way to increase the size of the training set. Augumenting the training examples allow the network to "see" more diversified, but still representative, datapoints during training.
Normalize the data between 0–1 by dividing train data and test data with 255 then convert all labels into one-hot vector with *to_catagorical ()* function.

The following code defines a set of augumentations for the training-set: *rotation*, *shift*, *shear*, *flip*, and *zoom*.

```
train_datagen = ImageDataGenerator(rescale=1. / 255,
                                   shear_range=0.2,
                                  zoom_range=0.2,
                                  horizontal_flip=True)
```

```
test_datagen = ImageDataGenerator(rescale=1. / 255)
```

Whenever the dataset size is small, data augmentation should be used to create additional training data.

Also, I created a data generator to get our data from our folders and into Keras in an automated way. Keras provides convenient python generator functions for this purpose.

```
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical')
```

```
Found 6899 images belonging to 8 classes.
```

```
validation_generator = test_datagen.flow_from_directory(
    validation_data_dir,
    target_size=(img_width, img_height),
    batch_size=batch_size,
    class_mode='categorical')
```

```
Found 2054 images belonging to 8 classes.
```

Target size code is for
resizing every image by 150 X 150 size so all the images will be of the same size and shape and class mode is chosen as categorical because we have 8 classes we need to go with. Generator function identified number of training and testing images and how much classes are there.

8. Train the model

```
nb_train_samples = 6899
nb_validation_samples = 2054
epochs = 25
batch_size = 16
```

Next I trained the model for 25 epochs with a batch size of 16.

Batch size is one of the most important hyperparameters to tune in deep learning. I prefer to use a larger batch size to train my models as it allows computational speedups from the parallelism of GPUs. However, it is well known that too large of a batch size will lead to poor generalization. On the one extreme, using a batch equal to the entire dataset guarantees convergence to the global optima of the objective function. However, this is at the cost of slower convergence to that optima. On the other hand, using smaller batch sizes have been shown to have faster convergence to good results. This is intuitively explained by the fact that smaller batch sizes allow the model to start learning before having to see all the data. The downside of using a smaller batch size is that the model is not guaranteed to converge to the global optima. Therefore, it is often advised that one starts at a small batch size reaping the benefits of faster training dynamics and steadily grows the batch size through training.

```
c=model.fit_generator(
    train_generator,
    steps_per_epoch=nb_train_samples // batch_size,
    epochs=epochs,
    validation_data=validation_generator,
    validation_steps=nb_validation_samples // batch_size)
```

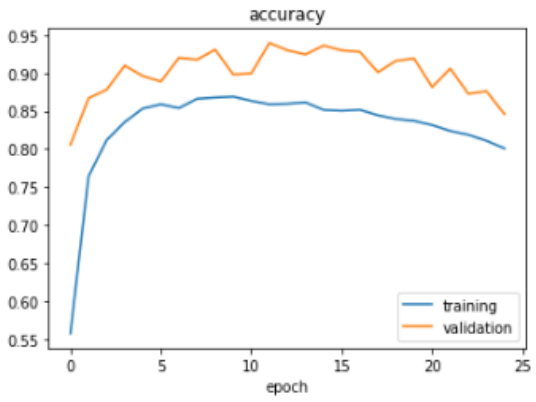
After training we got more than 80% accuracy and more than 90% validation accuracy. It is not actually bad at all.

```
Epoch 1/25
431/431 [==============================] - 3095s 7s/step - loss: 0.3440 - accuracy: 0.3967 - val_loss: 0.1161 - val_accuracy: 0.8057
Epoch 2/25
431/431 [==============================] - 55s 128ms/step - loss: 0.1633 - accuracy: 0.7450 - val_loss: 0.0850 - val_accuracy: 0.8667
Epoch 3/25
431/431 [==============================] - 55s 127ms/step - loss: 0.1273 - accuracy: 0.8112 - val_loss: 0.0830 - val_accuracy: 0.8779
Epoch 4/25
431/431 [==============================] - 56s 129ms/step - loss: 0.1125 - accuracy: 0.8333 - val_loss: 0.0589 - val_accuracy: 0.9097
Epoch 5/25
431/431 [==============================] - 55s 127ms/step - loss: 0.1034 - accuracy: 0.8563 - val_loss: 0.0682 - val_accuracy: 0.8960
Epoch 6/25
431/431 [==============================] - 55s 127ms/step - loss: 0.0976 - accuracy: 0.8601 - val_loss: 0.0732 - val_accuracy: 0.8887
Epoch 7/25
431/431 [==============================] - 54s 126ms/step - loss: 0.1026 - accuracy: 0.8488 - val_loss: 0.0686 - val_accuracy: 0.9199
Epoch 8/25
431/431 [==============================] - 54s 126ms/step - loss: 0.0948 - accuracy: 0.8677 - val_loss: 0.0582 - val_accuracy: 0.9175
Epoch 9/25
431/431 [==============================] - 55s 127ms/step - loss: 0.0958 - accuracy: 0.8675 - val_loss: 0.0503 - val_accuracy: 0.9307
Epoch 10/25
431/431 [==============================] - 55s 128ms/step - loss: 0.0915 - accuracy: 0.8738 - val_loss: 0.0738 - val_accuracy: 0.8979
Epoch 11/25
431/431 [==============================] - 55s 127ms/step - loss: 0.1002 - accuracy: 0.8645 - val_loss: 0.0750 - val_accuracy: 0.8994
Epoch 12/25
431/431 [==============================] - 54s 126ms/step - loss: 0.0980 - accuracy: 0.8593 - val_loss: 0.0515 - val_accuracy: 0.9395
Epoch 13/25
431/431 [==============================] - 54s 125ms/step - loss: 0.1083 - accuracy: 0.8614 - val_loss: 0.0657 - val_accuracy: 0.9297
Epoch 14/25
431/431 [==============================] - 54s 125ms/step - loss: 0.1140 - accuracy: 0.8612 - val_loss: 0.0655 - val_accuracy: 0.9243
Epoch 15/25
431/431 [==============================] - 54s 125ms/step - loss: 0.1067 - accuracy: 0.8552 - val_loss: 0.0626 - val_accuracy: 0.9360
Epoch 16/25
431/431 [==============================] - 54s 126ms/step - loss: 0.1082 - accuracy: 0.8527 - val_loss: 0.0566 - val_accuracy: 0.9297
Epoch 17/25
431/431 [==============================] - 54s 124ms/step - loss: 0.1130 - accuracy: 0.8517 - val_loss: 0.0604 - val_accuracy: 0.9277
Epoch 18/25
431/431 [==============================] - 54s 126ms/step - loss: 0.1189 - accuracy: 0.8445 - val_loss: 0.0833 - val_accuracy: 0.9009
Epoch 19/25
431/431 [==============================] - 55s 127ms/step - loss: 0.1210 - accuracy: 0.8412 - val_loss: 0.0695 - val_accuracy: 0.9160
Epoch 20/25
431/431 [==============================] - 55s 127ms/step - loss: 0.1274 - accuracy: 0.8283 - val_loss: 0.0629 - val_accuracy: 0.9189
Epoch 21/25
431/431 [==============================] - 55s 128ms/step - loss: 0.1182 - accuracy: 0.8414 - val_loss: 0.0925 - val_accuracy: 0.8813
Epoch 22/25
431/431 [==============================] - 54s 126ms/step - loss: 0.1317 - accuracy: 0.8327 - val_loss: 0.0758 - val_accuracy: 0.9058
Epoch 23/25
431/431 [==============================] - 54s 126ms/step - loss: 0.1356 - accuracy: 0.8227 - val_loss: 0.1014 - val_accuracy: 0.8726
Epoch 24/25
431/431 [==============================] - 55s 127ms/step - loss: 0.1355 - accuracy: 0.8129 - val_loss: 0.1163 - val_accuracy: 0.8760
Epoch 25/25
431/431 [==============================] - 54s 126ms/step - loss: 0.1391 - accuracy: 0.8075 - val_loss: 0.0952 - val_accuracy: 0.8457
```

9.Evaluate the Model

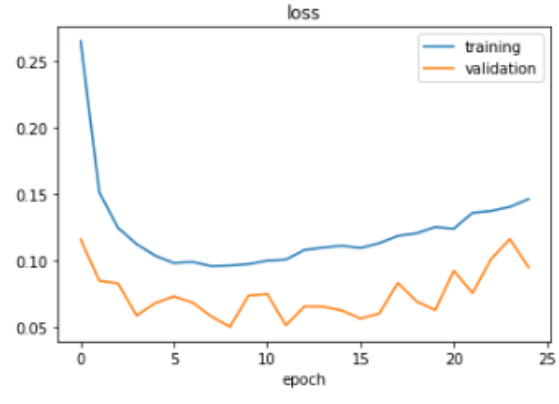
```
plt.plot(c.history['accuracy'])
plt.plot(c.history['val_accuracy'])
plt.legend(['training','validation'])
plt.title('accuracy')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```




```
plt.plot(c.history['loss'])
plt.plot(c.history['val_loss'])
plt.legend(['training','validation'])
plt.title('loss')
plt.xlabel('epoch')
```

```
Text(0.5, 0, 'epoch')
```



We got an accuracy validation accuracy between 85-95%. Initially our accuracy for trainng data was at 40% but it in the remaining epochs it was able to maintain an accuracy between 80-85%. Loss for both training and validation is low

10. Save the model

```
model.save_weights('model_saved.h5')
```

11. Make a dictionary to map to the output classes and make predictions from the model

```
train_generator.class_indices
```

```
{'airplane': 0,
 'car': 1,
 'cat': 2,
 'dog': 3,
 'flower': 4,
 'fruit': 5,
 'motorbike': 6,
 'person': 7}
```

12. Predictions from model

```
import cv2
image=cv2.imread('/content/depositphotos_8807359-stock-photo-mixed-race-man-smiling.jpg')
img=cv2.resize(image,(150,150))
```

```
import numpy as np
img=np.reshape(img,[1,150,150,3])
```

```
s=model.predict_classes(img)
```

```
/usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/sequential.py:450: U
  warnings.warn('`model.predict_classes()` is deprecated and '
```

```
s
```

```
array([0])
```

I just uploaded an airplane image and it predicted it as class 0 which is airplane

Conclusion and Future scope

Thus, deep learning is indeed possible with less data. One could argue that this was fairly easy as car and bus look quite different even for the naked eye. Can we extend this and make a benign/malignant cancer classifier? Sure, we can but the key is using data augmentation whenever data-set size is small. Another approach could be using transfer learning using pre-trained weights.

Generally, they make some key findings from exploring the deeper architectural approach:

- The very deep architecture worked well on small and large datasets.
- Deeper networks decrease classification error.
- Max-pooling achieves better results than other, more sophisticated types of pooling.
- Generally going deeper degrades accuracy; the shortcut connections used in the architecture are important.
- The ability to build and deploy a deep learning model in around few lines of code is remarkable and shows how fast the world of AI is developing. This article shows a simple and efficient way of using different python libraries to build and deploy and model.

Model Applicability

1. **Stock Photography and Video Websites**. It's fueling billions of searches daily in stock websites. It provides the tools to make visual content discoverable by users via search.
2. **Visual Search for Improved Product Discoverability**. Visual Search allows users to search for similar images or products using a reference image they took with their camera or downloaded from internet.
3. **Security Industry**. This emerging technology is playing one of the vital roles in the security industry. Many security devices have been developed that includes drones, security cameras, facial recognition biometric devices, etc.
4. **Healthcare Industry.** Microsurgical procedures in the healthcare industry powered by robots use computer vision and image recognition techniques.
5. **Automobile Industry**. It can be used for decreasing the rate of road accidents, follow traffic rules and regulations in order, etc.

GitHub

https://github.com/johnahjohn/Image-Classification-Using-CNN

Dataset

https://www.kaggle.com/prasunroy/natural-images