Mayank Agarwal Follow
Deep Learning and Computer Vision Enthusiast
Apr 22, 2017 · 4 min read

Understanding and Coding Inception Module in Keras

Introduction

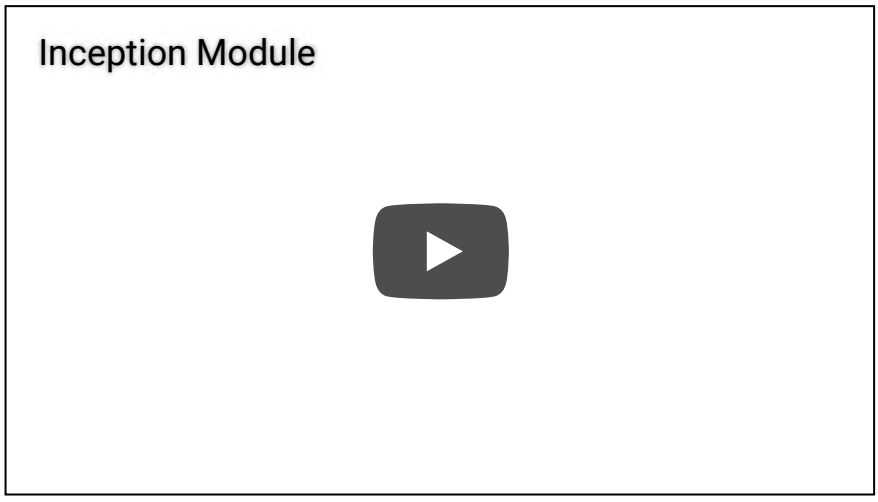
A typical Convolutional neural network (CNN) is made up of stacked convolutional layers in combination with max pooling and dropout. For larger datasets such as Imagenet, deeper architectures are used to get better results and dropout is used to prevent overfitting.

GoogLeNet achieved the new state of the art in the *ImageNet Large-Scale Visual Recognition Challenge 2014*. GoogLeNet was constructed by stacking Inception layers to create a deep convolutional neural network.

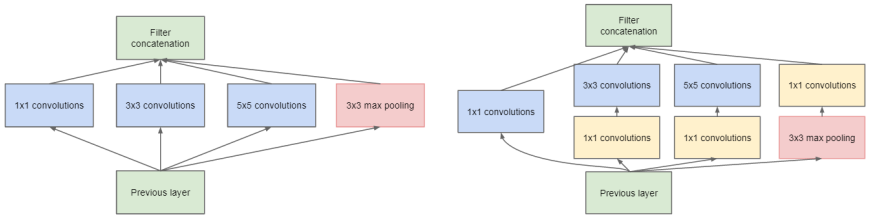
In this blog, I would describe the intuition behind the Inception module. I would also show how one can easily code an Inception module in Keras.

Inception Module

In a typical CNN layer, we make a choice to either have a stack of 3x3 filters, or a stack of 5x5 filters or a max pooling layer. In general all of these are beneficial to the modelling power of the network. The inception module suggests the use of all of them.



This means instead of adding a particular filter size layer, we add all 1x1, 3x3, 5x5 filters and perform convolution on the output from the previous layers. Since pooling has been essential for the success of current CNNs, the inception module also includes an additional pooling path.The output of all the filters are concatenated and passed on as input to the next layer.



(a) Inception module, naïve version (b) Inception module with dimension reductions

Coding Inception Module using Keras

We will build a simple architecture with just one layer of inception module using keras. Make sure you have already installed keras beforehand. We will train the architecture on the popular CIFAR-10 dataset which consists of 32x32 images belonging to 10 different classes.

Keras has the functionality to directly download the dataset using the cifar10.load_data() function. Once downloaded the function loads the data ready to use.

```
from keras.datasets import cifar10
(X_train, y_train), (X_test, y_test) = cifar10.load_data()
```

Each image is represented as 32x32 pixels each for red, blue and green channels. Each pixel has a value between 0–255. Next, we normalize the values to 0–1.

```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0
```

In order to best model the classification model, we convert y_test and y_train to one hot representations in the form of a binary matrix.

```
from keras.utils import np_utils
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
```

The latest Keras functional API allows us to define complex models. In order to create a model, let us first define an input_img tensor for a 32x32 image with 3 channels(RGB).

```
from keras.layers import Input
input_img = Input(shape = (32, 32, 3))
```

Now, we feed the input tensor to each of the 1x1, 3x3, 5x5 filters in the inception module.

```
from keras.layers import Conv2D, MaxPooling2D

tower_1 = Conv2D(64, (1,1), padding='same',
activation='relu')(input_img)
tower_1 = Conv2D(64, (3,3), padding='same',
activation='relu')(tower_1)


tower_2 = Conv2D(64, (1,1), padding='same',
activation='relu')(input_img)
tower_2 = Conv2D(64, (5,5), padding='same',
activation='relu')(tower_2)

tower_3 = MaxPooling2D((3,3), strides=(1,1), padding='same')
(input_img)
tower_3 = Conv2D(64, (1,1), padding='same',
activation='relu')(tower_3)
```

We learn 64 1x1 filters on the input_img tensor and then we learn 64 3x3 filters on the tower_1 tensor. Similarly, we make tower_2, tower_3 tensors.

Note: We provide input_img tensor to tower_2 and tower_3 as input so all the 3x3, 5x5 filters and the max pooling layers are performed on the same input.

The padding is kept same so that the output shape of the Conv2D operation is same as the input shape. So, the final output of each filter of tower_1, tower_2 and tower_3 is same. Thus we can easily concatenate these filters to form the output of our inception module.

```
output = keras.layers.concatenate([tower_1, tower_2,
tower_3], axis = 3)
```

Concatenate operation assumes that the dimensions of tower_1, tower_2, tower_3 are the same, except for the concatenation axis.

We flatten the output to a one dimensional collection of neurons which is then used to create a fully connected neural network as a final classifier

```
from keras.layers import Flatten, Dense
output = Flatten()(output)
out    = Dense(10, activation='softmax')(output)
```

Thus we obtain a fully connected neural network with final layer having 10 neurons one corresponding to each class.

We can now create the model

```
from keras.models import Model
model = Model(inputs = input_img, outputs = out)
# print model.summary()
```

Now we compile and fit the model with SGD optimizer

```
from keras.optimizers import SGD
epochs = 25
lrate = 0.01
decay = lrate/epochs
sgd = SGD(lr=lrate, momentum=0.9, decay=decay,
nesterov=False)
```

```
model.compile(loss='categorical_crossentropy',
optimizer=sgd, metrics=['accuracy'])
```

```
model.fit(X_train, y_train, validation_data=(X_test,
y_test), epochs=epochs, batch_size=32)
```

We can also store our model in a JSON file and store the results as HDF5 format.

```
from keras.models import model_from_json
import os
```

```
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
```

```
model.save_weights(os.path.join(os.getcwd(), 'model.h5'))
```

We can calculate the accuracy of our model for evaluation

```
scores = model.evaluate(X_test, y_test, verbose=0)
print("Accuracy: %.2f%%" % (scores[1]*100))
```

ar