

GoogLeNet in Keras

JUNE 2016 • JOE MARINO



GoogLeNet (Inception V1) in Keras.

In this new era of deep learning, a number of software libraries have cropped up, each promising users speed, ease of use, and compatibility with state-of-the-art models and techniques. The go-to library in the Caltech vision lab has been **Caffe**, an open-source library developed by **Yangqing Jia** and maintained by the **Berkeley Vision and Learning Center (BVLC)**. It has been the gold standard in terms of speed and offering the latest pre-trained models (AlexNet, GoogLeNet, etc.). However, for trying anything other than what's already been done, Caffe can be rigid and difficult to adapt. Diving into the C++ code to implement a new layer can be a daunting task, and don't get me started on those .prototxt files! For research, I need a deep learning library that I

can easily adapt to whatever experiment I'm on, so I went searching for greener pastures and found a python library developed by François Chollet that runs on top of Theano and Tensorflow. Installing Keras and either of these backend libraries is fairly easy (just pip install), and Keras itself achieves an excellent balance of simplicity and adaptability.

However, Keras doesn't contain the degree of pre-trained models that come complete with Caffe. There are a number of github repositories by devoted Keras followers hosting implementations of AlexNet, VGG, GoogLeNet, etc., but from what I could tell, these models didn't exactly correspond to the models I had worked with in Caffe. As I often work with GoogLeNet (which is also referred to as Inception V1), I took it upon myself to transfer the weights from Caffe into an exact replica in Keras. The following is a walkthrough of my method.

Download the Keras Model

Before we begin, if you're not interested in getting into the details of how to implement GoogLeNet in Keras yourself, feel free to just download the model. Keras models are defined by two files: a json file containing the model architecture and an hdf5 file containing the model's weights. You can use the link below to download a zip folder containing the architecture and weight files for GoogLeNet.

 [Keras GoogLeNet](#)

To load the model, simply run the following:

```
from googlenet_custom_layers import PoolHelper,LRN
from keras.models import model_from_json
model =
model_from_json(open('googlenet_architecture.json').read()),custom_objects=
```

```
{"PoolHelper": PoolHelper, "LRN": LRN})
```

Caltech

```
model.load_weights('googlenet_weights.h5')
```

ABOUT

RESEARCH

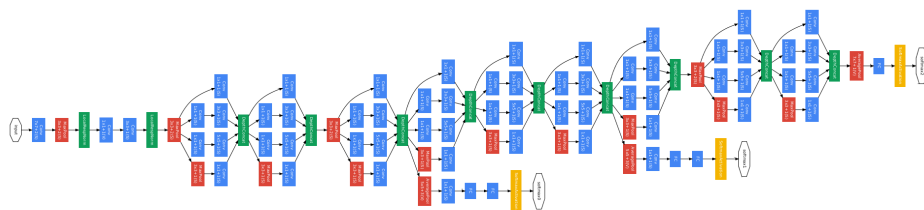
BLOG

CONTACT

I've also created a [GitHub Gist](#) that contains a script with the entire model definition.

I've laid this blog post out in two sections, constructing the network architecture and transferring the weights, however, these two phases went hand in hand. Along the way, I used the weights to generate activations to verify the network architecture.

Constructing the Network Architecture



The behemoth that sits above is GoogLeNet. As part of an ensemble of other similar models trained by the researchers at Google, GoogLeNet achieved a top-5 error rate of 6.67% on the [2014 ImageNet classification challenge](#). What that means is this: if you have an image of an object that is contained in the 1,000 object classes of the ImageNet dataset (all sorts of animals, household objects, vehicles, etc.), 93.33% of the time the correct object class will be contained in the GoogLeNet ensemble's top five predictions. Considering that ImageNet consists of many fine-grained object categories and that some images contain multiple object categories, this is an incredible feat, nearly on par with human performance. While at first glance the model may appear incredibly complex, upon closer inspection, the overall structure of the model can be broken down into a few

basic sections: the stem, the inception modules, auxiliary classifiers, and finally the output classifier.

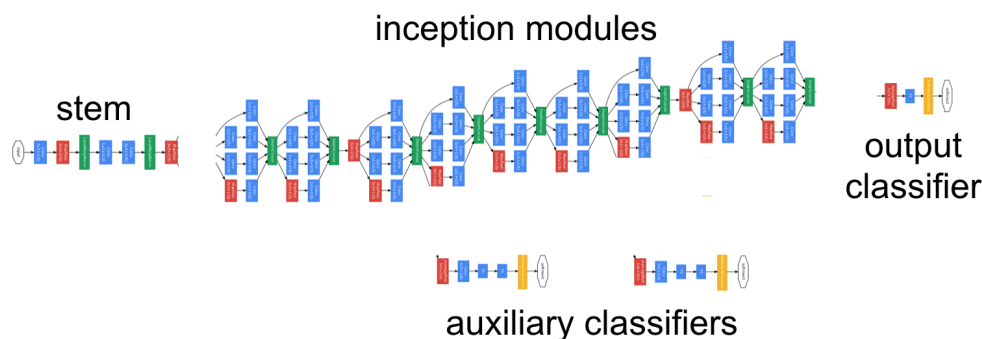
Caltech

ABOUT

RESEARCH

BLOG

CONTACT

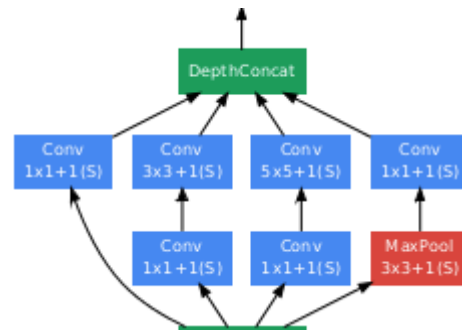


GoogLeNet starts with a sequential chain of convolution, pooling, and local response normalization operations, in a similar fashion to previous convolutional neural network models, such as AlexNet. Later papers on the inception architectures refer to this initial segment as the 'stem'. Shown below, the stem stands in contrast to the rest of the GoogLeNet architecture, which is primarily made up of what are referred to as 'inception' modules. The authors cite technical issues for including the stem rather than training a network made up entirely of inception modules. It will be interesting to see whether this stem section remains a part of future networks.

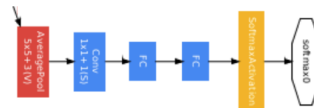


The basic building block of GoogLeNet, the inception module, is a set of convolutions and poolings at different scales, each done in parallel, then concatenated together. Along the way, 1×1 convolutions are used to reduce the dimensionality of inputs to convolutions with larger filter sizes. This approach results in a high performing model with drastically fewer parameters. GoogLeNet, in fact, has a factor of 12 times fewer parameters

than AlexNet. Why the name inception, you ask? Because the module represents a *network within a network*. Don't get the reference, go watch Christopher Nolan's *Inception*...this is what happens when we let computer scientists have a sense of humor.



The above diagram shows an inception module. GoogLeNet contains nine of these modules, sequentially stacked, with two max pooling layers along the way to reduce the spatial dimensions. Due to the depth of this architecture, the authors added two auxiliary classifiers branching from the main network structure. The purpose of these classifiers is to amplify the gradient signal back through the network, attempting to improve the earlier representations of the data. However, with the introduction of batch normalization, these classifiers have been ignored in recent models.



Finally, we get to the output classifier, which performs an average pooling operation followed by a softmax activation on a fully connected layer.



In total, the network uses the standard operations: convolution, pooling, normalization, and fully-connected

Caltech

ABOUT

RESEARCH

BLOG

CONTACT

Unbeknownst to me, each of these operations are performed differently across different software libraries, so each operation required some hacking to convert from Caffe and Keras.

Let's start by looking at the input. Opening GoogLeNet's `deploy.prototxt` file from Caffe, we first see

```
name: "GoogLeNet"
input: "data"
input_dim: 10
input_dim: 3
input_dim: 224
input_dim: 224
```

What we have is a network named GoogLeNet that takes a 4-D input blob "data" with input dimensions (10, 3, 224, 224), i.e. batches of 10 images, each with 3 channels (note: in BGR order!), of size 224×224 . In Keras, working with the [Functional API](#), this is equivalently written as

```
input = Input(shape=(3, 224, 224))
```

The batch size is omitted for the time being, as it gets set once we train or test with the model. Moving on, let's look at the first convolutional layer in Caffe:

```
layer {
  name: "conv1/7x7_s2"
  type: "Convolution"
  bottom: "data"
  top: "conv1/7x7_s2"
  param {
    lr_mult: 1
```

```
        decay_mult: 1
    },
    param {
        lr_mult: 2
        decay_mult: 0
    }
    convolution_param {
        num_output: 64
        pad: 3
        kernel_size: 7
        stride: 2
        weight_filler {
            type: "xavier"
            std: 0.1
        }
        bias_filler {
            type: "constant"
            value: 0.2
        }
    }
}

layer {
    name: "conv1/relu_7x7"
    type: "ReLU"
    bottom: "conv1/7x7_s2"
    top: "conv1/7x7_s2"
}
```

What we see is a convolutional layer named "conv1/7x7_s2" that takes input from "data", applies a set of $64 \ 7 \times 7$ convolutional filters with a stride of 2 and a padding of 3, then passes the

activations through the ReLU layer "conv1/relu". In Keras, we can implement this using a Convolution2D layer as follows:

Caltech

ABOUT

RESEARCH

BLOG

CONTACT

```
conv1_7x7_s2 = Convolution2D(64,7,7, subsample=(2,2),
border_mode='same', activation='relu',
name='conv1/7x7_s2')(input)
```

We use the `subsample` and `border_mode` keyword arguments to handle the stride and padding respectively. By setting `border_mode='same'`, we tell Keras that we want to pad the input with zeros such that the spatial output size of the layer (if we were to neglect the stride) would be the same as the input size. For a filter size of 7×7 , this would correspond to a padding of 3, as in Caffe. Next, we move on to a pooling layer:

```
layer {
  name: "pool1/3x3_s2"
  type: "Pooling"
  bottom: "conv1/7x7_s2"
  top: "pool1/3x3_s2"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
```

This max-pooling layer takes input from "conv1/7x7_s2" and uses a 3×3 window with stride 2 to subsample the maximum activations from the convolution. Since the layer's prototxt definition does not specify any padding, this is considered to be a 'valid' operation, i.e. only take maxima at locations where the window completely overlaps the input. In Keras, we would implement this using a `MaxPooling2d` layer:

pool1_3x3_s2 = MaxPooling2D(pool_size=(3,3), strides=(2,2), border_mode='valid', name='pool1/3x3_s2')(conv1_7x7_s2)

Caltech

ABOUT

RESEARCH

BLOG

CONTACT

However, **this operation is not consistent from Caffe to Keras.** After some frustration, I found that Caffe actually does pad the end of both spatial dimensions. In this way, with indexing ranging from (0, 0) to (111, 111), the first pooling location is centered at (1, 1), but the last pooling location is centered at (111, 111). Valid at the beginning, same at the end. My solution: zero-pad the result from the convolutional layer, then use a custom layer to remove the zeros from the beginning of both dimensions (the first row and column). Thus,

```
conv1_zero_pad = ZeroPadding2D(padding=(1, 1))
(conv1_7x7_s2)
pool1_helper = PoolHelper()(conv1_zero_pad)
pool1_3x3_s2 = MaxPooling2D(pool_size=(3,3), strides=
(2,2), border_mode='valid', name='pool1/3x3_s2')
(pool1_helper)
```

where `PoolHelper` is a custom layer, implemented as

```
class PoolHelper(Layer):
    def __init__(self, **kwargs):
        super(PoolHelper, self).__init__(**kwargs)

    def call(self, x, mask=None):
        return x[:, :, 1:, 1:]

    def get_config(self):
        config = {}
        base_config = super(PoolHelper, self).get_config()
```

Sure enough, this does the trick. Now on to normalization. In Caffe, the local response normalization layers are defined as

```
layer {
  name: "pool1/norm1"
  type: "LRN"
  bottom: "pool1/3x3_s2"
  top: "pool1/norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
```

What do the hyperparameters `local_size`, `alpha`, and `beta` mean? In Caffe, the local response normalization is performed for each example by normalizing along the feature (channel) dimension. This normalization is performed only over a small window, the *local size*, over features at every spatial location. The parameters α and β come in through the normalization equation:

$$LRN(x_{f,r,c}) = \left(k + \frac{\alpha}{n} \sum_{i=f-\frac{n}{2}}^{f+\frac{n}{2}} x_{i,r,c}^2 \right)^\beta. \quad (1)$$

Let's deconstruct this equation. The local response normalization of the input $x_{f,r,c}$ at a particular feature f at row r and column c is given by an offset k plus a scaling coefficient α times the average squared input over the centered window of size n (the local size), all taken to the power of β . Again, note

that the window extends only over the feature map, meaning that each spatial location is normalized separately. In Caffe, k is set to 1.

Caltech

ABOUT

RESEARCH

BLOG

CONTACT

In the latest version of Keras, the only form of normalization is the `BatchNormalization` layer. And while this layer is useful and versatile, especially with the widespread adoption and nice theoretical motivations of batch normalization, it does not perform local response normalization in the same way as Caffe. **Previous versions** of Keras included a `LRN2D` layer, which was adapted from `pylearn2`, however I had to make modifications to this code to obtain the identical operation performed by Caffe. The layer definition is below:

```
class LRN(Layer):
    def __init__(self, alpha=0.0001,k=1,beta=0.75,n=5,
**kwargs):
        self.alpha = alpha
        self.k = k
        self.beta = beta
        self.n = n
        super(LRN, self).__init__(**kwargs)

    def call(self, x, mask=None):
        b, ch, r, c = x.shape
        half_n = self.n // 2 # half the local region
        input_sqr = T.sqr(x) # square the input
        extra_channels = T.alloc(0., b, ch + 2*half_n, r,
c) # make an empty tensor with zero pads along channel
dimension
        input_sqr = T.set_subtensor(extra_channels[:,
half_n:half_n+ch, :, :],input_sqr) # set the center to be
the squared input
```

```

        scale = self.k # offset for the s
        norm_alpha = self.alpha / self.n
        alpha

        for i in range(self.n):
            scale += norm_alpha * input_sqr[:, i:i+ch, :,
:]

        scale = scale ** self.beta
        x = x / scale
        return x

def get_config(self):
    config = {"alpha": self.alpha,
              "k": self.k,
              "beta": self.beta,
              "n": self.n}

    base_config = super(LRN, self).get_config()
    return dict(list(base_config.items()) +
list(config.items()))

```

I have added comments to explain the steps along the way. Now we can add the LRN layer to our network:

```
pool1_norm1 = LRN(name='pool1/norm1')(pool1_3x3_s2)
```

Now you might be thinking to yourself, "We're only three layers in! This is going to take forever!" Not to fear, after accounting for the differences in pooling and normalization between Caffe and Keras, the rest of the network is a straightforward conversion. The rest of the stem can be completed similarly to the first three layers. I'll spare you the prototxt version, but in Keras, the first inception module looks like this:

```
inception_3a_1x1 = Convolution2D(64,1,1,
border_mode='same', activation='relu',
```

```

name='inception_3a/1x1')(pool2_3x3_s2)

inception_3a_3x3_reduce = Convolution2D(96,1,1,
border_mode='same', activation='relu',
name='inception_3a/3x3_reduce')(pool2_3x3_s2)

inception_3a_3x3 = Convolution2D(128,3,3,
border_mode='same', activation='relu',
name='inception_3a/3x3')(inception_3a_3x3_reduce)

inception_3a_5x5_reduce = Convolution2D(16,1,1,
border_mode='same', activation='relu',
name='inception_3a/5x5_reduce')(pool2_3x3_s2)

inception_3a_5x5 = Convolution2D(32,5,5,
border_mode='same', activation='relu',
name='inception_3a/5x5')(inception_3a_5x5_reduce)

inception_3a_pool = MaxPooling2D(pool_size=(3,3), strides=
(1,1), border_mode='same', name='inception_3a/pool')
(pool2_3x3_s2)

inception_3a_pool_proj = Convolution2D(32,1,1,
border_mode='same', activation='relu',
name='inception_3a/pool_proj')(inception_3a_pool)

inception_3a_output = merge([inception_3a_1x1,
inception_3a_3x3, inception_3a_5x5,
inception_3a_pool_proj], mode='concat', concat_axis=1,
name='inception_3a/output')

```

We see 1×1 , 3×3 , and 5×5 convolutions with varying numbers of filters in addition to a pooling layer. At the end, the outputs of

these different pathways are concatenated using a 'merge' layer. The next inception module will then take the output from this concatenated layer. The rest of the inception layers are exactly identical, but with different numbers of filters.

Finally, we get to the classifiers. Since the auxiliary classifiers are not included in Caffe's deploy model, I will focus on the final classifier (the auxiliary classifiers are nearly identical). At the beginning of each classifier branch is an *average* pooling layer. In Caffe, this is written as

```
layer {
  name: "pool5/7x7_s1"
  type: "Pooling"
  bottom: "inception_5b/output"
  top: "pool5/7x7_s1"
  pooling_param {
    pool: AVE
    kernel_size: 7
    stride: 1
  }
}
```

In Keras, we can use the `AveragePooling2D` layer to implement this:

```
pool5_7x7_s1 = AveragePooling2D(pool_size=(7,7), strides=
(1,1), name='pool5/7x7_s2')(inception_5b_output)
```

We then come to a dropout layer.

```
layer {
  name: "pool5/drop_7x7_s1"
  type: "Dropout"
  bottom: "pool5/7x7_s1"
```

```

top: "pool5/7x7_s1"
dropout_param {
  dropout_ratio: 0.4
}
}

```

This is simply implemented using Keras' `Dropout` layer.

However, before this point, I flatten the input to get rid of the spatial dimensions, which by this point, are both of size 1.

```

loss3_flat = Flatten()(pool5_7x7_s1)
pool5_drop_7x7_s1 = Dropout(0.4, name='pool5/drop_7x7_s1')
(loss3_flat)

```

...and finally the output softmax layer.

```

layer {
  name: "loss3/classifier"
  type: "InnerProduct"
  bottom: "pool5/7x7_s1"
  top: "loss3/classifier"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 1000
    weight_filler {
      type: "xavier"
    }
  }
}

```

```

        bias_filler {
          type: "constant"
          value: 0
        }
      }
    }
  }
}
layer {
  name: "prob"
  type: "Softmax"
  bottom: "loss3/classifier"
  top: "prob"
}

```

This is implemented in Keras using a `Dense` layer (with default linear activation) followed by a softmax `Activation` layer.

```

loss3_classifier = Dense(1000, name='loss3/classifier')
(pool15_drop_7x7_s1)
loss3_classifier_act = Activation('softmax', name='prob')
(loss3_classifier)

```

Finally, in Keras, we need to turn the set of layers into a model by specifying the input and output.

```

googlenet = Model(input=input, output=
[loss1_classifier_act, loss2_classifier_act, loss3_classifier_act])

```

And that's all there is to it! At this point, to train the model, we would simply need to compile the it (likely using `'categorical_crossentropy'` for the loss function) and start feeding in batches of training examples. Don't forget: in order to train, you will also want to add the `W_regularizer` argument to each of the convolution and fully-connected layers. Caffe specifies the regularization hyperparameter as being 0.0002.

Caltech

However, the main point of reconstructing the network in Keras was to take advantage of the pre-trained weights from Caffe. In the next section, I'll walk through the process of transferring these weights over.

Transferring the Weights

GoogLeNet's weights are contained in a caffemodel file and can be accessed by loading them in Caffe. With the deploy.prototxt file, this is done as follows:

```
import caffe
MODEL_DEF = 'path/to/train_val.prototxt'
MODEL_WEIGHTS = 'path/to/bvlc_googlenet.caffemodel'
net = caffe.Net(MODEL_DEF,MODEL_WEIGHTS,caffe.TEST)
```

The model's weights and biases are accessed through `net.params`, whereas each layer's activations are accessed through `net.blobs`. Therefore, to copy over the weights, we can run the following script:

```
for layer_name in net.params.keys():
    weights = np.copy(net.params[layer_name][0].data)
    biases = np.copy(net.params[layer_name][1].data)
    model_layer = googlenet.get_layer(name=layer_name)
    model_layer.set_weights([weights,biases])
```

We run through all of the layers, copying over the weights and biases from Caffe, then set those parameters in the corresponding layer in Keras. Unfortunately, if we run this script, we will encounter two errors, one of which will be obvious, and the other one not. The first error occurs due to the fact that the weights for fully-connected layers are transposed between Caffe and Keras. This is simply a convention between libraries. We can remedy this error by adding the following:

The other error is more difficult to catch. Caffe and Keras do not implement convolutional layers in the same manner. Keras (really Theano) performs convolution, whereas Caffe performs correlation. For more information on how these differ, see [this explanation](#). In real terms, this simply means that we need to rotate each of the filters by 180°:

```
if 'conv' in layer_name or 'proj' in layer_name or (('1x1'
in layer_name or '3x3' in layer_name or '5x5' in
layer_name) and 'inception' in layer_name):
    for i in range(weights.shape[0]): # go through each
filter
        for j in range(weights.shape[1]): # go through
each channel
            weights[i, j] = np.rot90(weights[i, j], 2) #
rotate it (twice)
```

Upon adding these two catches into our copying script, we can now copy the weights over. And there you have it, GoogLeNet in Keras!

Testing the Model

Just to make sure everything went off without a hitch, let's try running a sample image through both networks to verify that the activations are consistent. While copying over the weights, this was how I debugged the Keras model. Let's use the sample image used by Caffe, that adorable tabby cat kitten:



First we'll preprocess the image by subtracting the channels means, changing the channel ordering, switching the spatial and channel dimensions, cropping the image, and adding an extra dimension for the batch.

```
img = imread('cat.jpg', mode='RGB')
height,width = img.shape[:2]
img = img.astype('float32')
# subtract means
img[:, :, 0] -= 123.68
img[:, :, 1] -= 116.779
img[:, :, 2] -= 103.939
img[:,:,:,[0,1,2]] = img[:,:,:,[2,1,0]] # swap channels
img = img.transpose((2, 0, 1)) # re-order dimensions
img = img[:,(height-224)//2:(height+224)//2,(width-224)//2:(width+224)//2] #crop
img = np.expand_dims(img, axis=0) # add dimension for batch
```

Now we can run the image through Caffe. We will use the data layer to contain a single example and pass our image into this layer. We then call the network's forward function to propagate the activations through the layers.

```
net.blobs['data'].reshape(1,3,224,224)
net.blobs['data'].data = img
output = net.forward()
```

To get the activations in Keras, we will define a function that takes in a model, a layer, and an input and returns that layer's activations.

```
import theano
def get_activations(model, layer, X_batch):
    get_activations =
    theano.function([model.layers[0].input,K.learning_phase()],
        layer.output, allow_input_downcast=True)
    activations = get_activations(X_batch,0)
    return activations
```

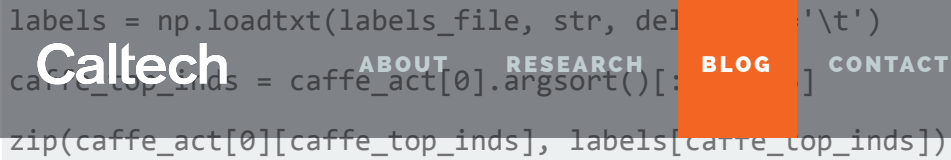
We can now compare activations between the two networks at any layer. If we have a layer called `layer_name`, we can get the activations as:

```
caffe_act = net.blobs[layer_name].data
layer = googlenet.get_layer(name=layer_name)
keras_act = get_activations(googlenet,layer,img)
```

Rather than show the activations for each and every layer, I will show the final output of each model by setting

`layer_name='prob'`. In Caffe, the image of the kitten results in

```
labels_file = caffe_root +
'data/ilsvrc12/synset_words.txt'
```



Caltech

ABOUT RESEARCH **BLOG** CONTACT

```
labels = np.loadtxt(labels_file, str, delimiter='\t')
caffe_top_inds = caffe_act[0].argsort()[::-1][:5]
zip(caffe_act[0][caffe_top_inds], labels[caffe_top_inds])
```

```
[(0.79362965, 'n02123394 Persian cat'),
 (0.081304103, 'n02127052 lynx, catamount'),
 (0.074268937, 'n02123159 tiger cat'),
 (0.029699767, 'n02123045 tabby, tabby cat'),
 (0.0024248438, 'n04589890 window screen')]
```

While in Keras, we get the following:

```
keras_top_inds = keras_act[0].argsort()[::-1][:5]
zip(keras_act[0][keras_top_inds], labels[keras_top_inds])
```

```
[(0.79363048, 'n02123394 Persian cat'),
 (0.081303641, 'n02127052 lynx, catamount'),
 (0.074268863, 'n02123159 tiger cat'),
 (0.029699543, 'n02123045 tabby, tabby cat'),
 (0.0024248327, 'n04589890 window screen')]
```

Immediately, we see that the network struggled with this image.
Honestly...Persian cat?



I'm no cat expert, but that looks to be a tabby. I guess object classification hasn't been solved. Comparing the probabilities between the two networks, we see that they match up to about four decimal places. I'm not sure specifically where this comes from, but I imagine it has something to do with the number of bits used to store each activation throughout the networks. For all intents and purposes, the networks produce identical outputs. Now get out there and have some fun with GoogLeNet!

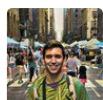
16 Comments**Blog****1 Login** ▾

♥ Recommend

🔗 Share

Sort by Best ▾

Join the discussion...

**Joe Marino** Mod • 2 months ago

Since making this post, François Chollet has released new (a.k.a. better) models for image recognition, including Inception v3 and ResNet. So if you're looking for something even better than GoogLeNet, check out his [repository of pre-trained models](#).

1 ^ | ▾ • Reply • Share ▾

**Shawn Wang** • 2 months ago

Awesome! I run the code with TitanX, but it seems a little slow. The forwarding path will cost 20 seconds with `batch_size=128`.

Callech

The caffe model seems only 562.841 ms. Is there a reason for this? Thanks!

ABOUT

RESEARCH

BLOG

CONTACT

^ | v · Reply · Share ›

**Anurag Priyadarshi** · 2 months ago

Excellent walkthrough! I must say you took up a daunting task. Decoding, rewriting and then explaining googLeNet can be a blog within a blog within a blog (codename inception ;)

^ | v · Reply · Share ›

**Evgeniy Koryagin** · 2 months ago

Hi! Thanks for the googlenet port to keras. Could you suggest how to reimplement LRN for using TensorFlow as backend instead of Theano (or for backend abstraction). I'm struggling with this.

^ | v · Reply · Share ›

**Joe Marino** Mod → Evgeniy Koryagin · 2 months ago

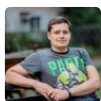
I'm not well versed on TensorFlow, so it would take me a little while to figure out how to convert the functions from Theano to TensorFlow. However, since making this post, François Chollet has released a more recent version of the inception architecture as well as the ResNet architecture for Keras, neither of which use the LRN layer. You can find those models [here](#).

1 ^ | v · Reply · Share ›

**Evgeniy Koryagin** → Joe Marino · 2 months ago

Thanks for the reply. Anyway, I'd prefer Inception v1 to v3 since it's faster on forward pass. Francois doesn't have it

^ | v · Reply · Share ›

**alex_lazarev** · 3 months ago

BTW how to train this model with your own images? The most interesting question is what "y" should be for [model.fit](#)? I mean shape and class value.

^ | v · Reply · Share ›

**Joe Marino** Mod → alex_lazarev · 2 months ago

You can train on your own images using the [ImageDataGenerator](#) class.

Fine-tuning with your own data is also further explained [here](#).

^ | v · Reply · Share ›

Caltech

ABOUT
alex_lazarevRESEARCH
Joe Marino • 2 months ago

BLOG

CONTACT

One more thing, the GoogLeNet (InceptionV3) model now is in applications part of Keras. And contains even pre-trained weights. There is also an example about fine tune that model. I did try to follow the example and failed with some strange error. Here's the ticket [https://github.com/fchollet/ke...](https://github.com/fchollet/keras/issues/1000)

^ | v • Reply • Share ›



alex_lazarev → Joe Marino • 2 months ago

Yeah that's what I'm doing right now. But if VGG16 is pretty simple to understand, GoogleLeNet is not so simple about output dimension and how to represent classes. Also from the example you've mentioned there is pretty everything is ok when we're training for binary problem, but for multiclass that example does not work so good:
I'm trying to classify images from Oxford 102 flower dataset (see my repo [https://github.com/Arsey/keras...](https://github.com/Arsey/keras/blob/master/oxford102.py) and there are couple issues:

1. with trained top model that has 99% on train data and 78% on validation data, I'm getting by some reason max 35% accuracy on test data o_o. Pretty strange isn't it?
2. if we proceed to the latest step (fine-tuning), again by some reason it starts on the 1st epoch from 11% (it should start at higher point as we had bottlenecks with 78% acc) then finishes max at 76% (validation) and again by some reason has even lower accuracy on test data - 11%

I'm pretty new in this area so guess there is something from my side. But again I couldn't find fine-tuning for Keras+VGG16+multiclass to see wha't wrong with my trial.

^ | v • Reply • Share ›

alex_lazarev → alex_lazarev
• 2 months ago

I've finally found the issue! On training process I'm using rescale for RGB values to 1. / 255, but I had forgotten to add

Caltech

ABOUT

RESEARCH

BLOG

CONTACT

**alex_lazarev** • 3 months ago

Hi! Inside of get_activations function there is K.learning_phase(). What is the "K" package here? I suppose it should be "from keras import backend as K"

^ | v • Reply • Share ›

**Joe Marino** Mod → alex_lazarev • 3 months ago

Yes, that's correct.

^ | v • Reply • Share ›

**Janpreet Singh** • 3 months ago

Hello,

I am facing problem to understand the step after the input(3,224,224) step. It's given that the filter size is 7 x 7 and a padding of 3. But, when I do the calculation i.e $224 + 6 - 7/4$, it is not coming out to be an integer, if you could explain that or

CONTACT

jmarino [at] caltech.edu

1200 E. California Blvd.

MC 136-93

Pasadena, 91125 CA

CONNECT

© Copyright 2016 Joseph Marino.

Caltech

ABOUT

RESEARCH

BLOG

CONTACT