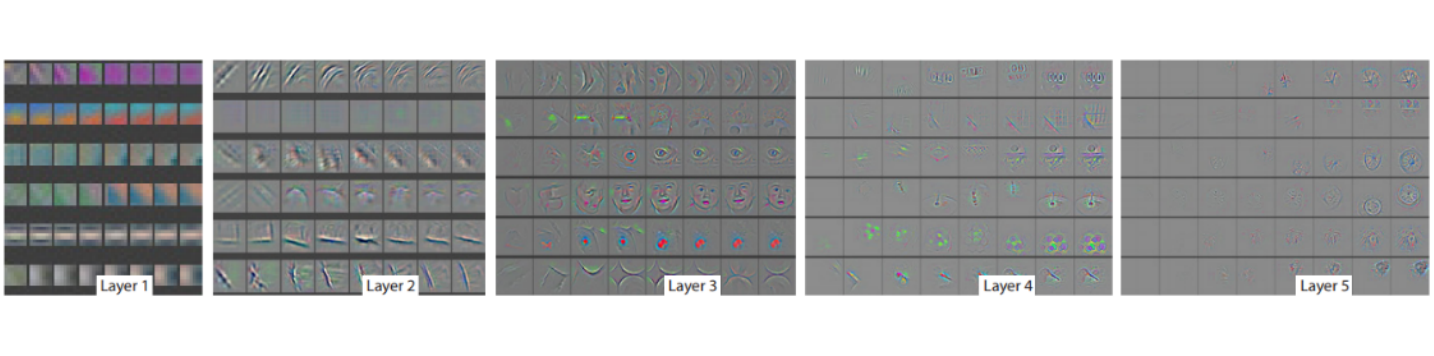
31 Oct 2017 • on python keras machine learning CNN transfer learning

The Power of Transfer Learning

1 Introduction

Deep learning with Deep Neural Network has gained huge traction in the past few years owning to the blooming of data and GPU accelerated computation. The former makes it possible to collect massive amount of training data to tune millions or even billions of parameters; the later makes running deep learning computation approachable for everyone have access to a consumer PC with GPU installed.

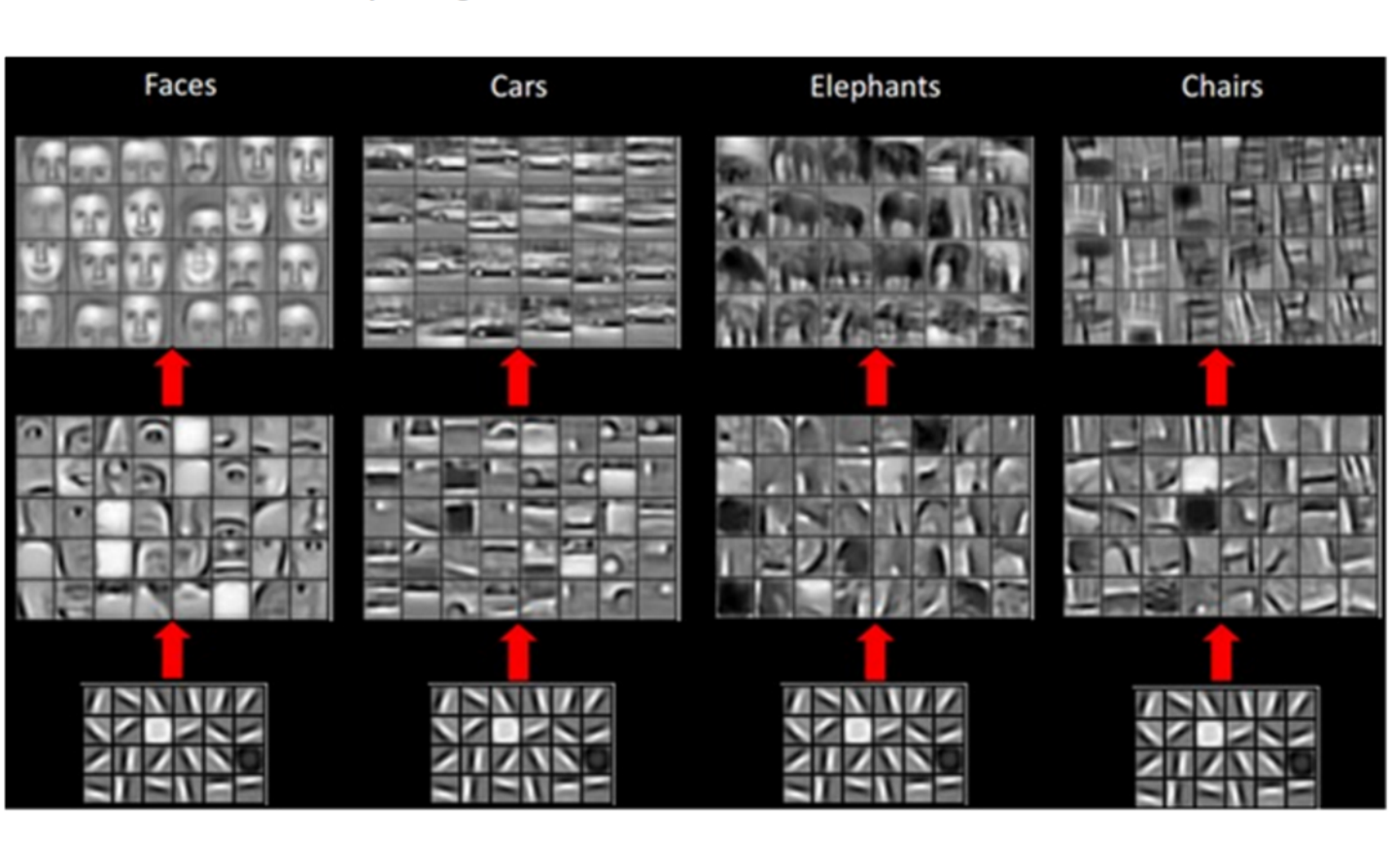
One of the most exciting fields of deep learning application is computer vision which gives machine the ability to see. One of the most influential innovations in the field of computer vision is Convolution Neurual Network(CNN), first published by LeCun et al [1], became prominence as Alex Krizhevsky used it to win 2012 ImageNet competition. The key idea of CNN is weight sharing unlike Multilayer Perceptrons (MLP) where nodes are fully connected, each input has its own weight. This allows CNN to extract patterns from images regardless of their position. First layer extracts simple patters such as edges. Following layers extract more abstract patterns based on the previous layers [2].



2 Task

Though empowered by GPU, training an image classifier with CNN still requires a long computation time. CNN itself needs to go deeper to achieve more complex tasks which requires massive amount of training data. Neither is often feasible for individuals.

As aforementioned, the first few layers extract relatively simple patterns which are roughly the same for different image classification tasks. For instance, basic patterns like edges and curves are the basic elements of every image [3] (Lee et al 2009).

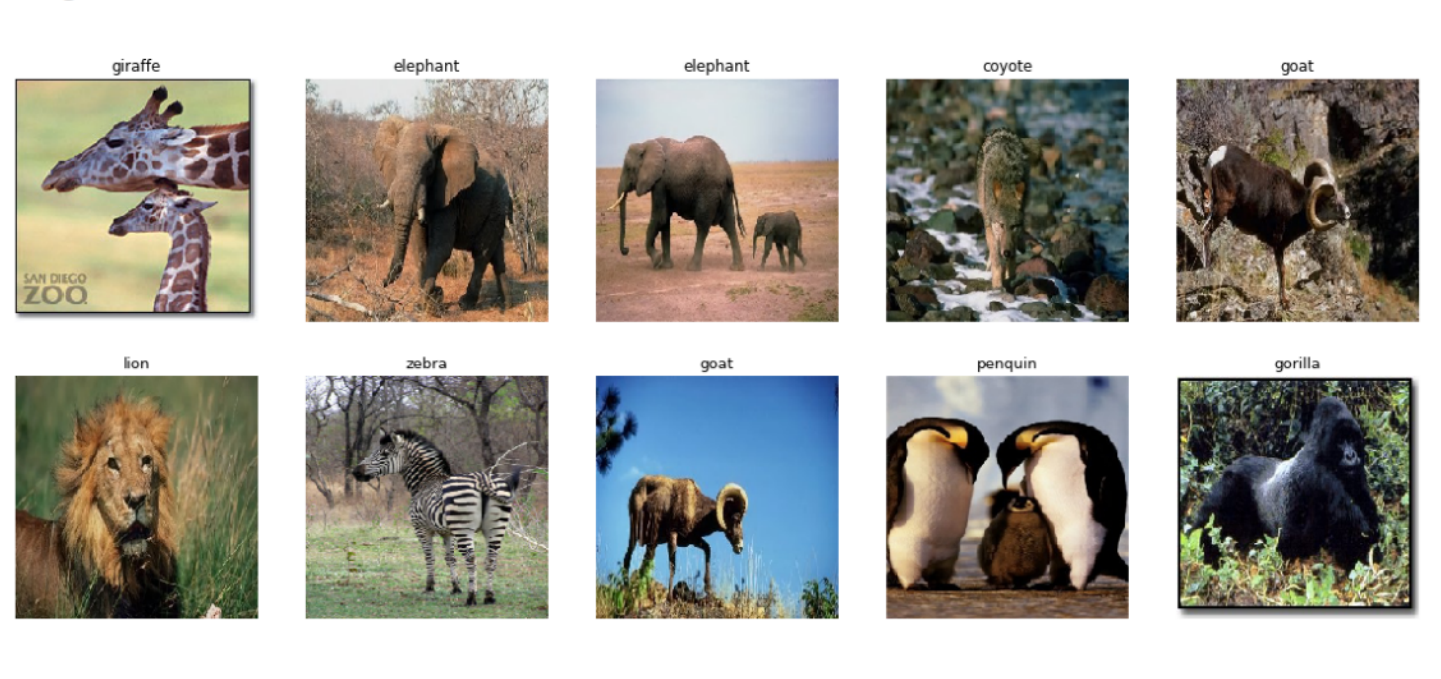


Naturally we would want to find a way to reuse the less abstract and less task oriented layers pre-trained by massive data sets. This is when transfer learning come to the rescue. Transfer learning allows us to reuse the less abstract layers of a pre-trained model for a similar tasks and achieve a different goal by training the more abstract layers with way less data and effort due to the massive reduction of trainable parameters.

In this study, I look to use transfer learning with Keras to train an animal image classifier with 19 classes and very few training data. Then I look to use the trained model to achieve near effortless object localization.

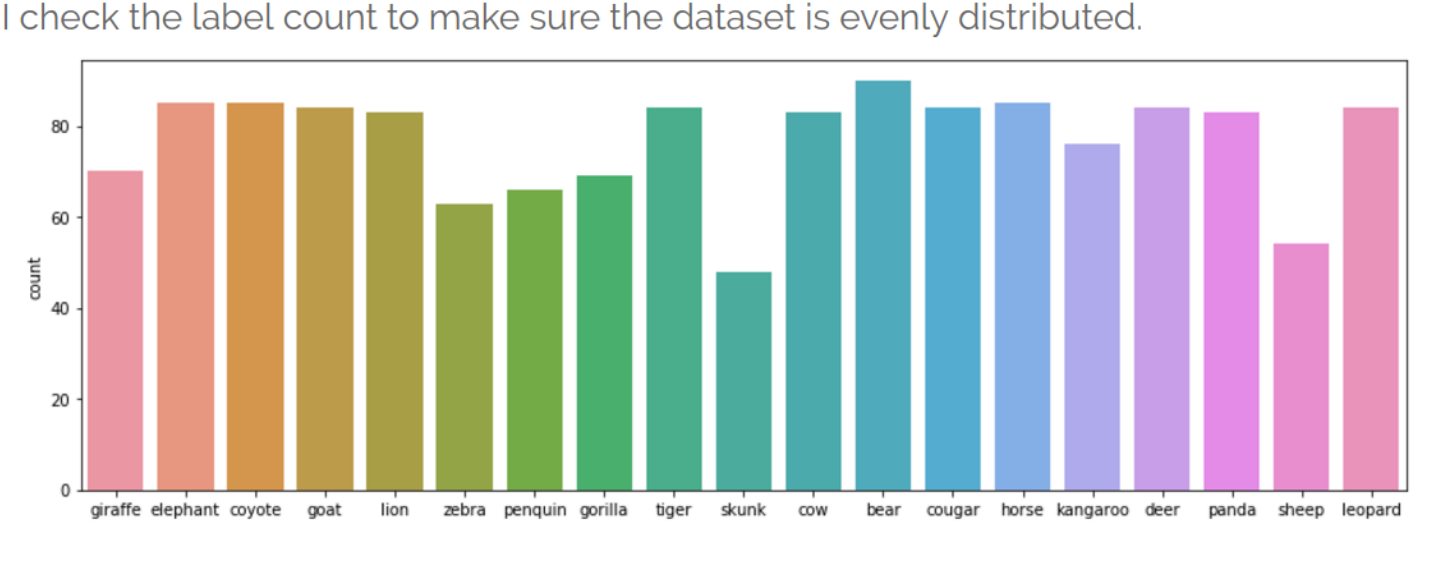
3 Dataset

The KTH-ANIMALS dataset[4] contains only 1740 images in total and are of different shapes from ~200 to ~400 pixels width and height. They are loaded as 224 by 224 pixels images. Below shows the images loaded.



As can been seen in the training data, the images are taken from different angles in the wild. This gives us even more troubles. For example, a more complex model is required to learn a goat looking from the front as well as from the side when the goat horn appears in different shapes from different angle.

I check the label count to make sure the dataset is evenly distributed.



I then cut the data into training, validation and test set containing 1460, 180 and 100 images respectively. Leaving us merely 78 images on average per animal class which is far from sufficient to train a CNN model from stretch to classify such a diverse dataset. I download another 40 images from Google and add to the test set (test set has 140 images in total).

4 Evaluation Metrics

Classification Accuracy and Log Loss are used as evaluation metrics. Classification Accuracy gives me the percentage of correctly classified images. Log Loss will give me more information about how confident the model is for correct predictions as well as how far from correct the incorrect predictions are. Log Loss for multi-class classification is defined by[6]:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{i,j}\log(p_{i,j})$$

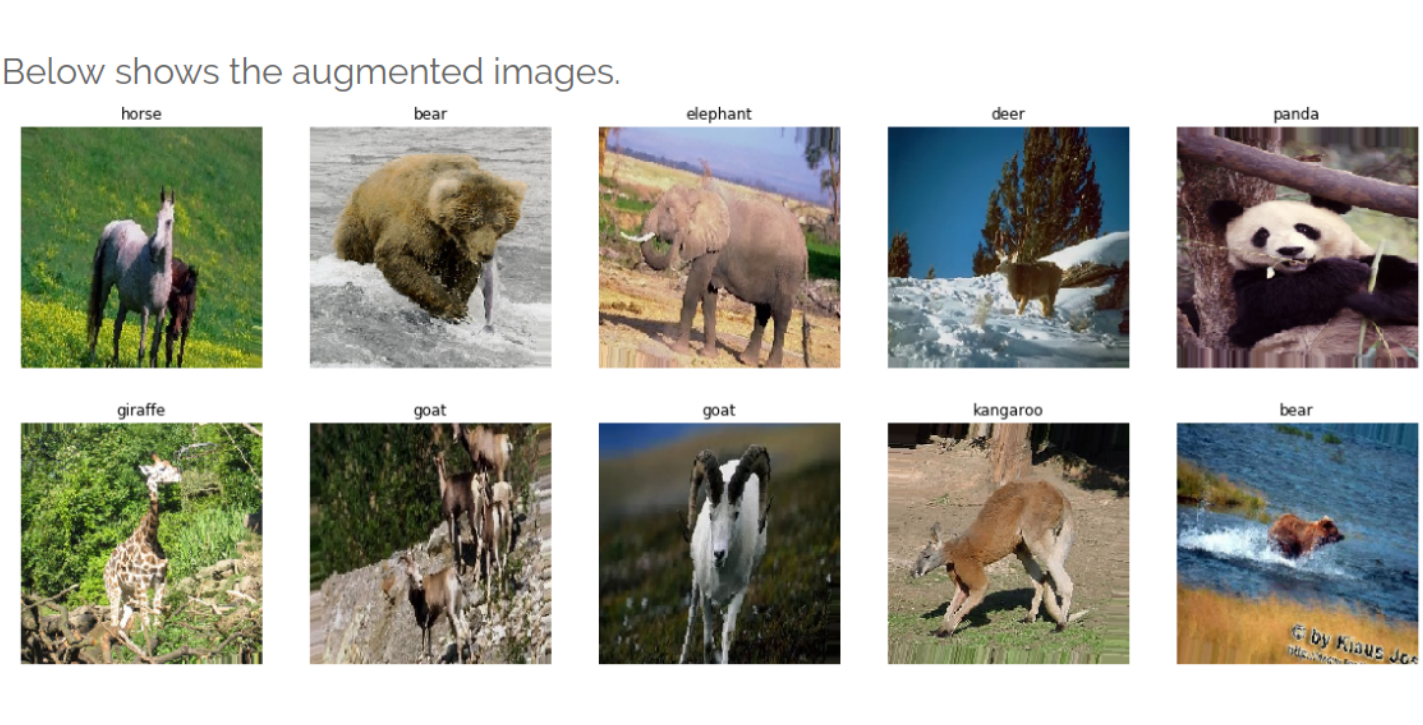
5 Data Preprocessing

All image pixels are scaled by 1/255 when being loaded into 4D tensor.

Image Augmentation is applied to training data to combat overfitting and to make the models more robust.

```
datagen = ImageDataGenerator(
    rotation_range=10,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
)
```

Below shows the augmented images.



6 Benchmark Model

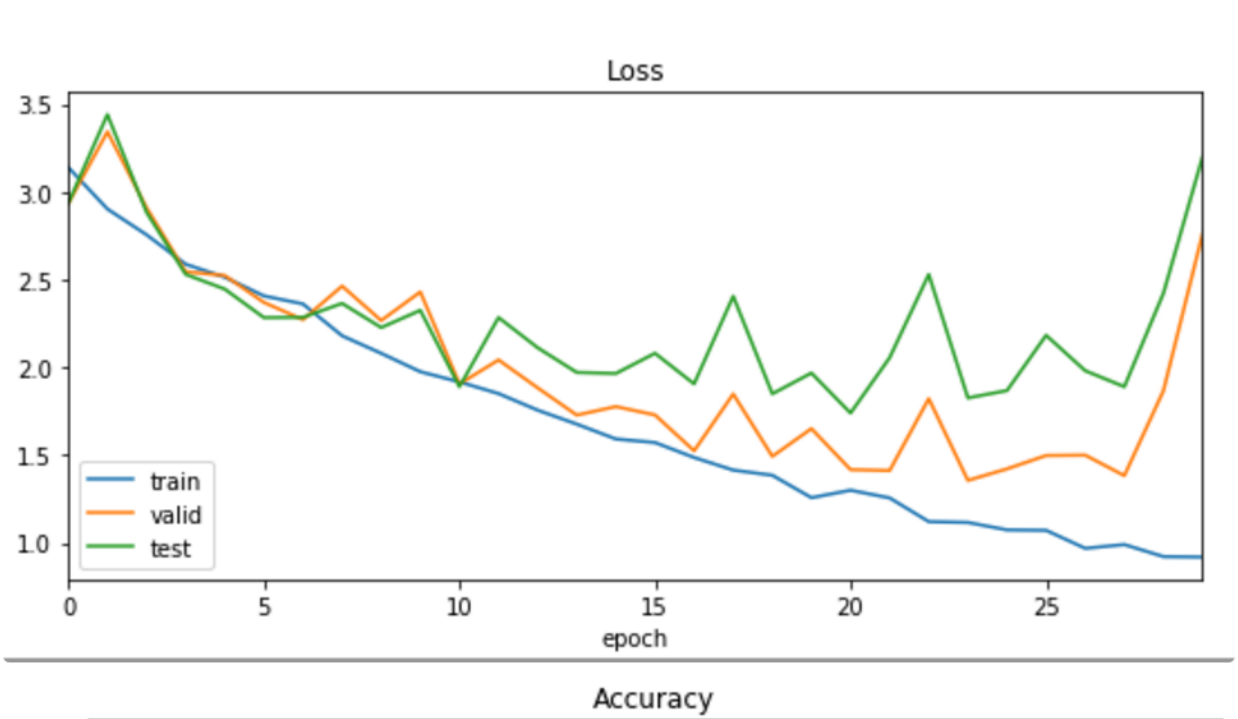
The benchmark model - a CNN model with 5 Convolution layers with similar architecture to VGG19 except that every block has only one CNN layer and Global Average Pooling is used instead of Fully Connected layer to reduce overfitting.

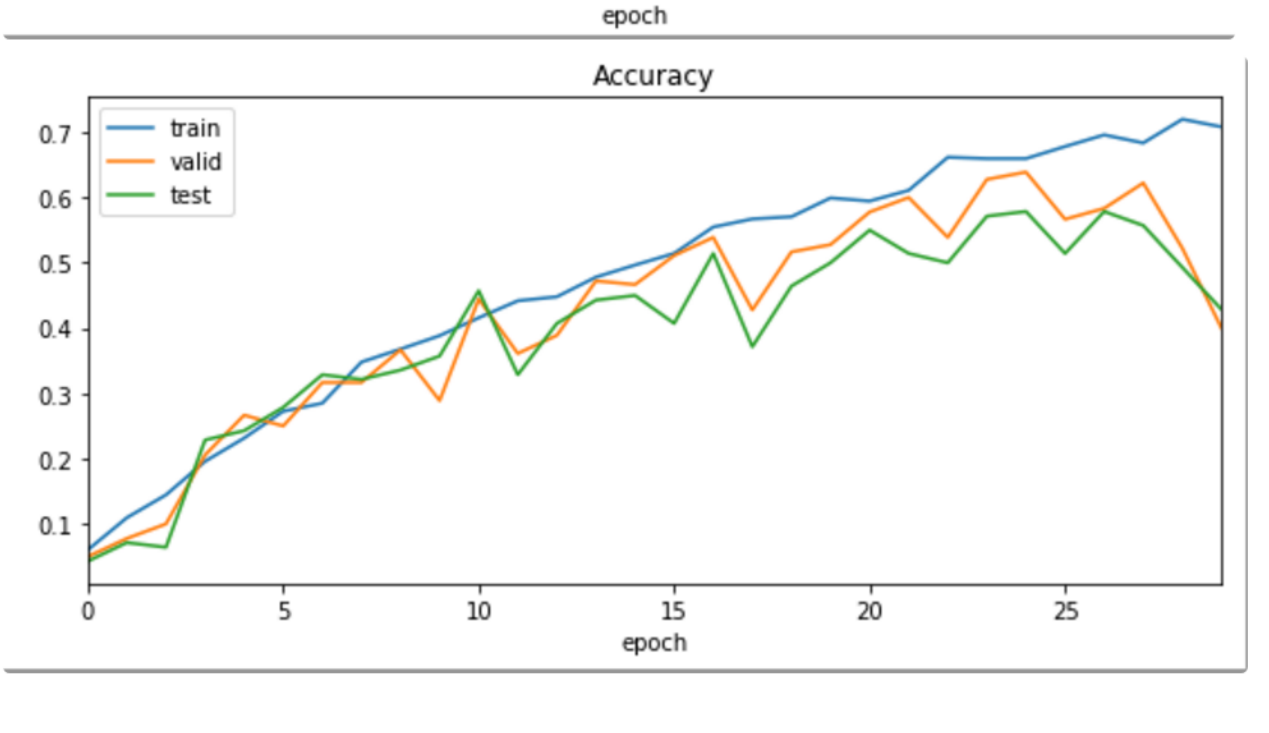
```
Layer (type)                 Output Shape              Param #
=================================================================
Input (InputLayer)           (None, 224, 224, 3)       0
_________________________________________________________________
Conv_1 (Conv2D)              (None, 224, 224, 64)      1792
_________________________________________________________________
MaxPool_2 (MaxPooling2D)     (None, 112, 112, 64)      0
_________________________________________________________________
Conv_3 (Conv2D)              (None, 112, 112, 128)     73856
_________________________________________________________________
MaxPool_4 (MaxPooling2D)     (None, 56, 56, 128)       0
_________________________________________________________________
Conv_5 (Conv2D)              (None, 56, 56, 256)       295168
_________________________________________________________________
MaxPool_5 (MaxPooling2D)     (None, 28, 28, 256)       0
_________________________________________________________________
Conv_7 (Conv2D)              (None, 28, 28, 512)       1180160
_________________________________________________________________
MaxPool_7 (MaxPooling2D)     (None, 14, 14, 512)       0
_________________________________________________________________
Dropout_8 (Dropout)          (None, 14, 14, 512)       0
_________________________________________________________________
Conv_9 (Conv2D)              (None, 14, 14, 512)       2359808
_________________________________________________________________
MaxPool_10 (MaxPooling2D)    (None, 7, 7, 512)         0
_________________________________________________________________
Dropout_11 (Dropout)         (None, 7, 7, 512)         0
_________________________________________________________________
GAP_12 (GlobalAveragePooling (None, 512)               0
_________________________________________________________________
Dropout_13 (Dropout)         (None, 512)               0
_________________________________________________________________
Output (Dense)               (None, 19)                9747
=================================================================
Total params: 3,920,531
Trainable params: 3,920,531
Non-trainable params: 0
_________________________________________________________________
```

A simpler model tend to easily overfit(or underfit if too simple) while a deeper model cannot learn due to the limitation of the insufficient number of training data.

It took 259.6 seconds, 29 epochs to finish training the benchmark model. 8.95 seconds per epoch. Achieve 57% test accuracy and 1.83 log loss at the 23th epoch.

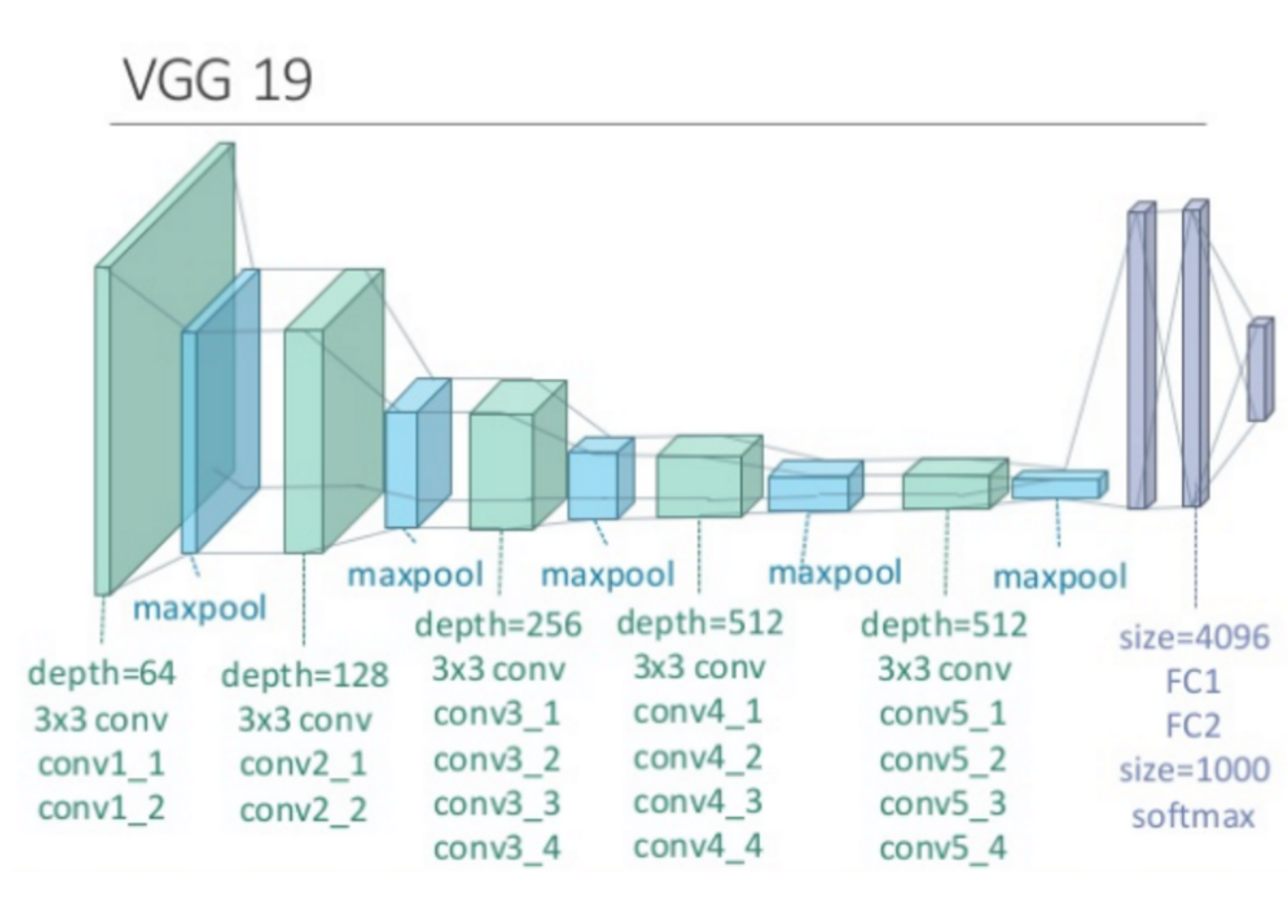
	epoch	metric	train	valid	test
46	23	loss	1.117020	1.356680	1.826603
47	23	acc	0.659589	0.627778	0.571429





7 Transfer Learning Model

The transfer learning model - a VGG19 with weights pre-trained on ImageNet dataset with top layers(two Fully Connected layers) replaced with a top model.



Top Model below is used to replace the two FC layers and the output layer (in purple above).

```
Layer (type)                 Output Shape              Param #
=================================================================
GAP (GlobalAveragePooling2D) (None, 512)               0
_________________________________________________________________
Dropout (Dropout)            (None, 512)               0
_________________________________________________________________
FC (Dense)                   (None, 19)                9747
_________________________________________________________________
Softmax (Activation)         (None, 19)                0
=================================================================
Total params: 9,747
Trainable params: 9,747
Non-trainable params: 0
_________________________________________________________________
```

7.1 Generate bottleneck features

To train the top model above, we need to pass the training, validation and test dataset through the VGG19(without top layers included) to obtain the input features (bottleneck features) for the top model.

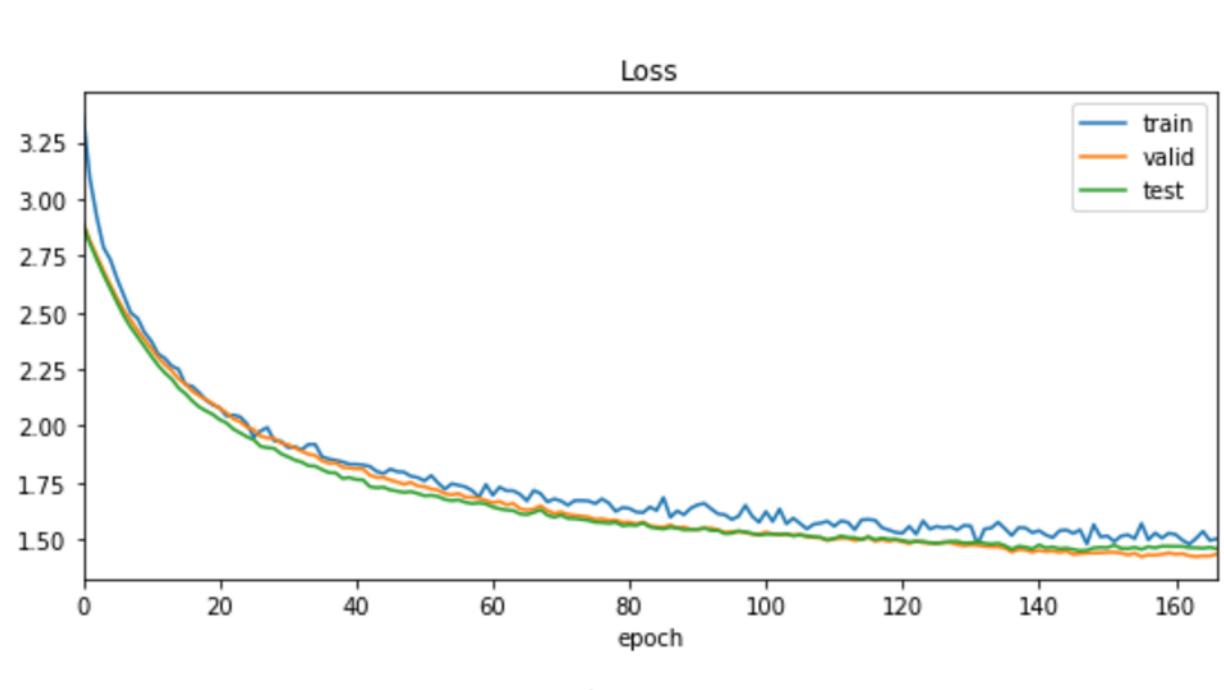
```
# Load pre-trained VGG19
pretrained_model = applications.VGG19(include_top=False, weights='imagenet')

# Generator for augmented training data
train_X_generator = datagen.flow(X_train, batch_size=batch_size, shuffle=False)

# Generate bottleneck features
bottleneck_features_train = pretrained_model.predict_generator(train_X_generator,
steps=len(X_train) / batch_size)
bottleneck_features_valid = pretrained_model.predict(X_valid)
bottleneck_features_test = pretrained_model.predict(X_test)
```

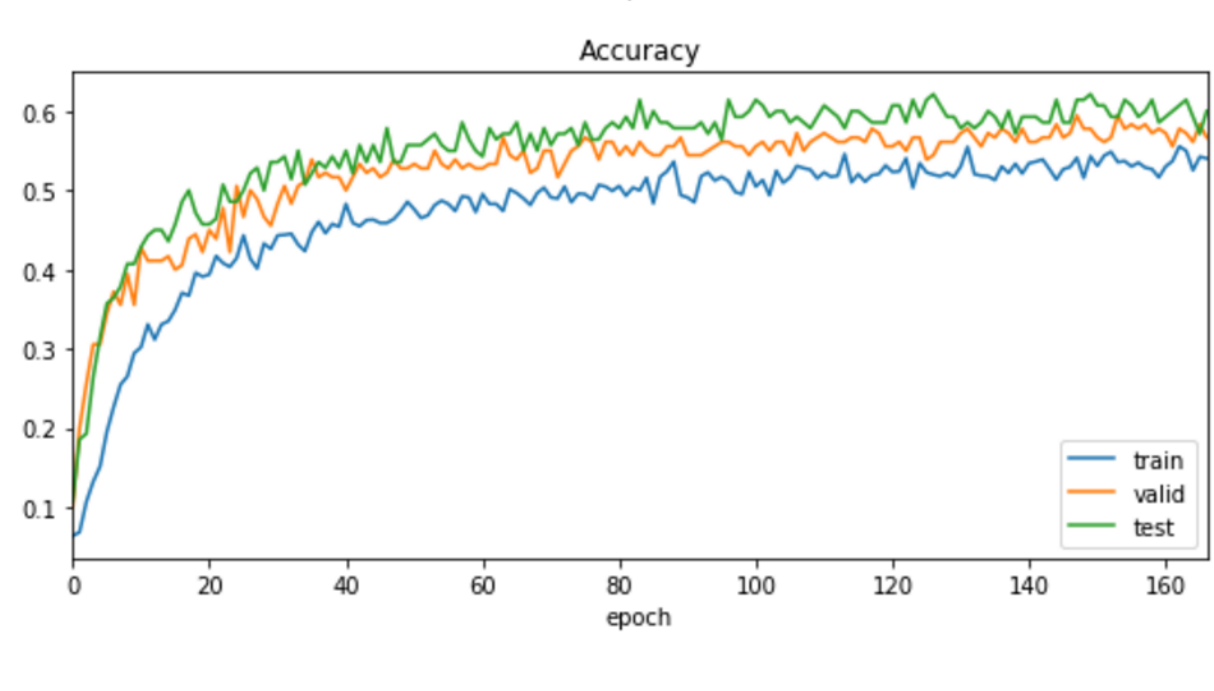
7.2 Train the top model

I then use the bottleneck features as the input to train the weight of the top model.

It took 31.31 seconds, 166 epochs to finish training the top model. Merely 0.19 second per epoch. Achieve 61% test accuracy and 1.45 log loss at the 155st epoch.

	epoch	metric	train	valid	test
310	155	loss	1.569475	1.423375	1.455613
311	155	acc	0.530137	0.583333	0.607143





The power of transfer learning is already showing! Training the top model is 47x faster while achieving a way better result. +4% in test accuracy.
One can stop here if speed is very important in the application and the accuracy is acceptable. I pursuit a better result by making the best out of transfer learning by fine tuning the last block of VGG19.

7.3 Fine tune VGG19

I connect the top model trained about to the pre-trained VGG19(with top layers removed) to build a new model.

Freeze the first 4 CNN blocks of VGG19 to fine tune the last (5th) block of VGG19 as well as our top model. By freezing the first 4 CNN blocks, we can freeze the hard learned pre-trained weights applicable to general image classification task. Fine tuning the more abstract layers so that the last CNN block learns the high level features related to this task.

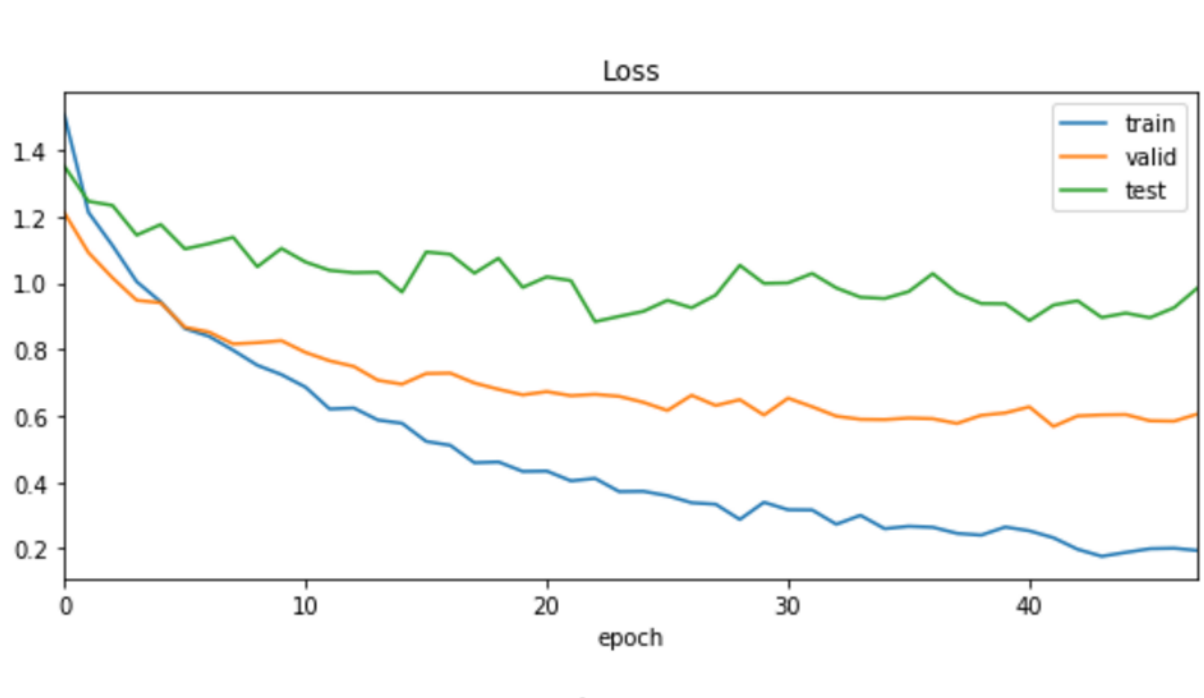
```
model = Sequential()

# Add VGG19 (without top layers)
for l in pretrained_model.layers:
    model.add(l)

# Add my Top Model
for l in top_model.layers:
    model.add(l)

## Lock layers until the last ConvNet block
lock_until = 17
for n, layer in enumerate(model.layers):
    if n < lock_until:
        layer.trainable = False
    else:
        layer.trainable = True
```

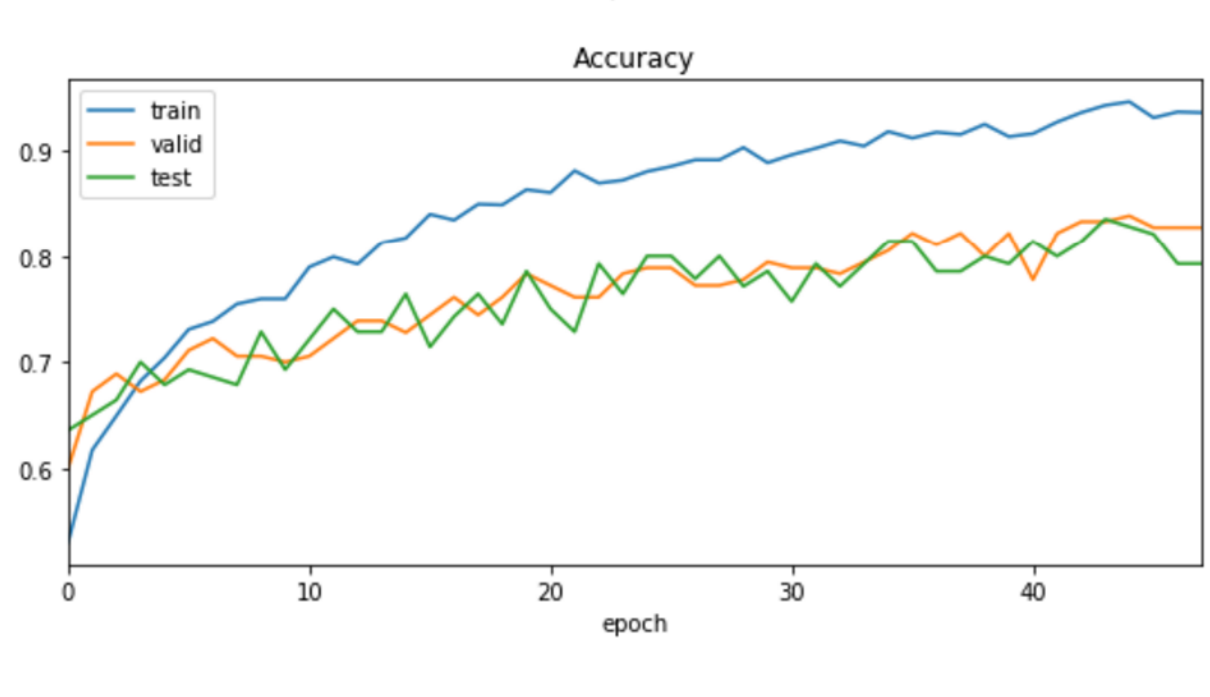
Here is the model with weights trainable from block5_conv1.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         (None, None, None, 3)     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, None, None, 64)    1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, None, None, 64)    36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, None, None, 64)    0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, None, None, 128)   73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, None, None, 128)   147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, None, None, 128)   0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, None, None, 256)   295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_conv4 (Conv2D)        (None, None, None, 256)   590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, None, None, 256)   0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, None, None, 512)   1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_conv4 (Conv2D)        (None, None, None, 512)   2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, None, None, 512)   0
_________________________________________________________________
GAP (GlobalAveragePooling2D) (None, 512)               0
_________________________________________________________________
Dropout (Dropout)            (None, 512)               0
_________________________________________________________________
FC (Dense)                   (None, 19)                9747
_________________________________________________________________
Softmax (Activation)         (None, 19)                0
=================================================================
Total params: 20,034,131
Trainable params: 9,448,979
Non-trainable params: 10,585,152
_________________________________________________________________
```

It took 600.55 seconds, 47 epochs to finish fine tuning the model. 12.7 seconds per epoch. Achieve 80% test accuracy and 0.93 log loss at the 41st epoch. Another +19% improvement in test accuracy. It's impossible to train a model from stretch to achieve 80% test accuracy with so little date. Transfer learning making it all possible with the expense of more computation time (varies depends on the benchmark and transfer model complexity).

	epoch	metric	train	valid	test
82	41	loss	0.232594	0.568131	0.933875
83	41	acc	0.927397	0.822222	0.800000

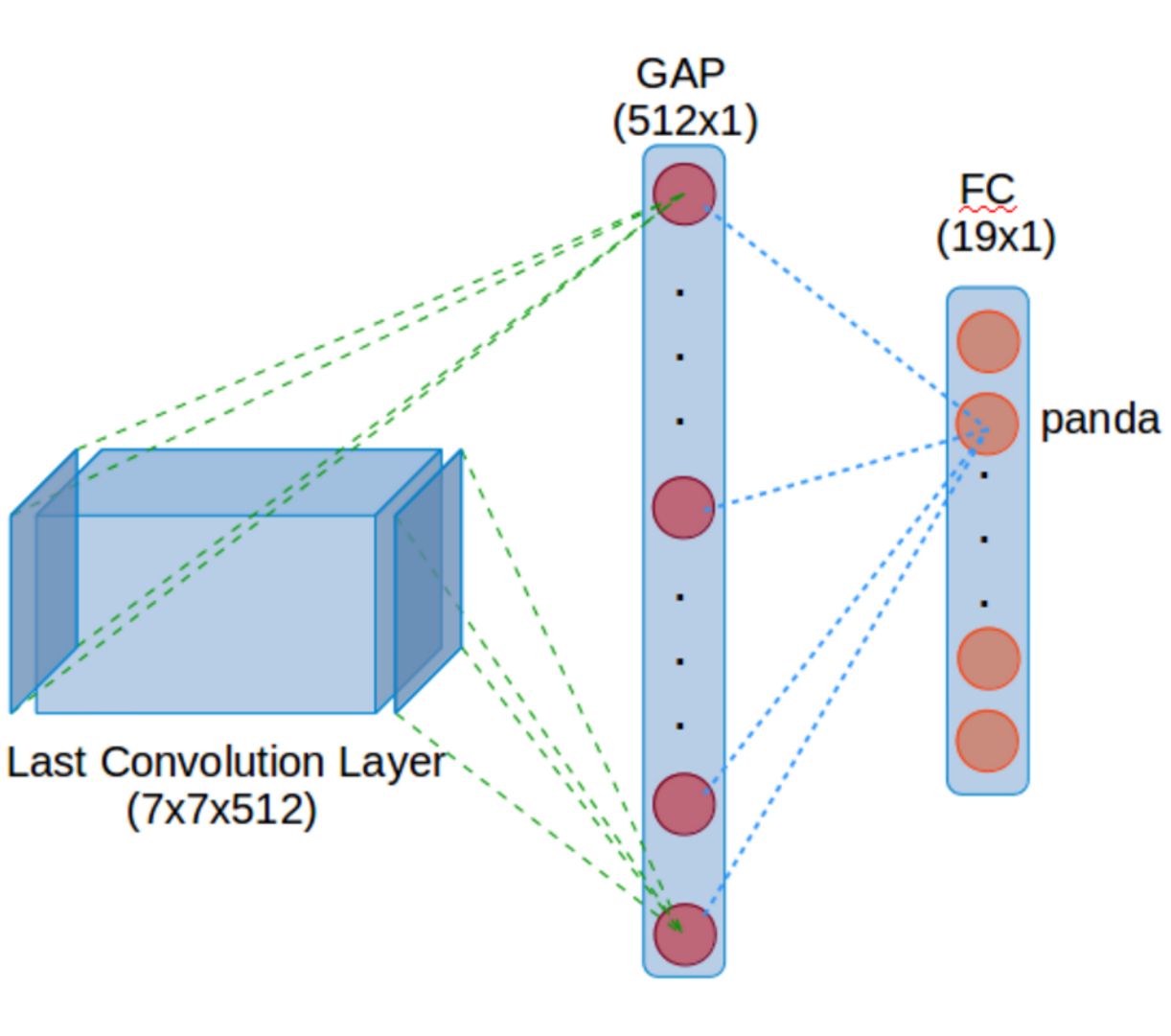




8 Object Localization

I use Global Average Pooling(GAP) instead of Fully Connected(FC) layers not only for reducing overfitting but also it brings an interesting byproduct - weights of the last CNN can be used for nearly effortless object localization. [7][8]

The idea behind is simple. When using GAP layer before the final FC layer. The weights (512x1 vector - blue dotted lines below) connecting the GAP to each of the output node, e.g. panda, tell us which of those 512 feature maps activates the output the most. And each of the 512 nodes in GAP layer is calculated averaging a 7x7 feature map, green dotted lines below.



Therefore multiplying 7x7x512 by 512x1 gives us the weighted feature map - the activation map which allow us to visualize which part of the image is activating that particular output class. Below I will show it in action.

Build a new model

First create a new model with the trained model above adding an extra output from the last Convolution Layer.

```
last_cnn_layer = -6  # the last convolution layer
output_layer = -2    # the last layer (-1) is softmax activation in the current setting
new_model = Model(inputs=model.input, outputs=(model.layers[last_cnn_layer].get_output_at(1),
model.layers[output_layer].get_output_at(1)))
```

Get convolution output and prediction

Pass the image through the model to obtain the last convolution output as well as the predicted class

```
# get filtered images from convolutional output + model prediction vector
last_conv_output, pred_vec = new_model.predict(img_path_to_tensor(img_path))
last_conv_output = np.squeeze(last_conv_output)  # (1, 14, 14, 512) --> (14, 14, 512)
pred = np.argmax(pred_vec) # get predicted class
```

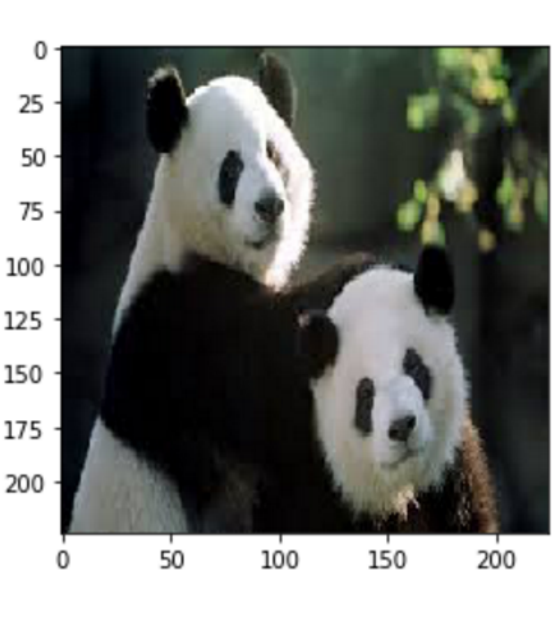
Get the FC layer weight (512x1)

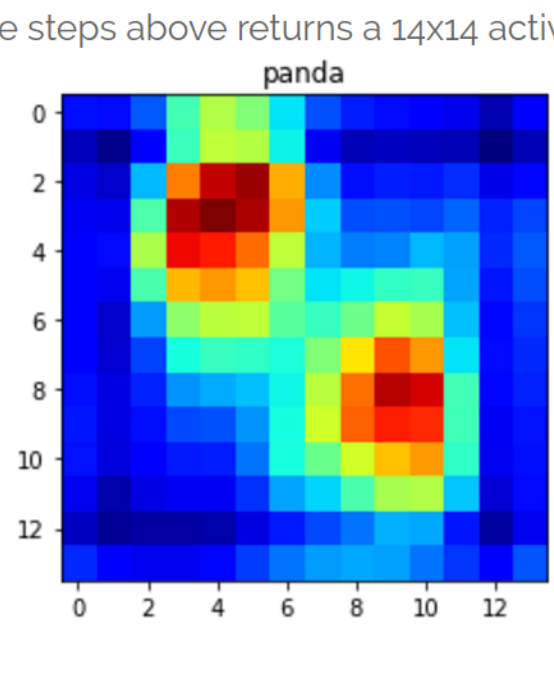
Get the FC layer weight that connected to the predicted class. Multiply the last convolution output (14x14x512 - green dotted lines) by the FC layer weight (512x1 - blue dotted lines)

```
all_fc_layer_weights = model.layers[output_layer].get_weights()[0]  # 512x19 (19 classes)
fc_layer_weights = all_fc_layer_weights[:, pred] * -1 # 512x1

# 14x14x512 * 512x1 = 14x14
activation_map = np.dot(last_conv_output, fc_layer_weights)
```

Visualize

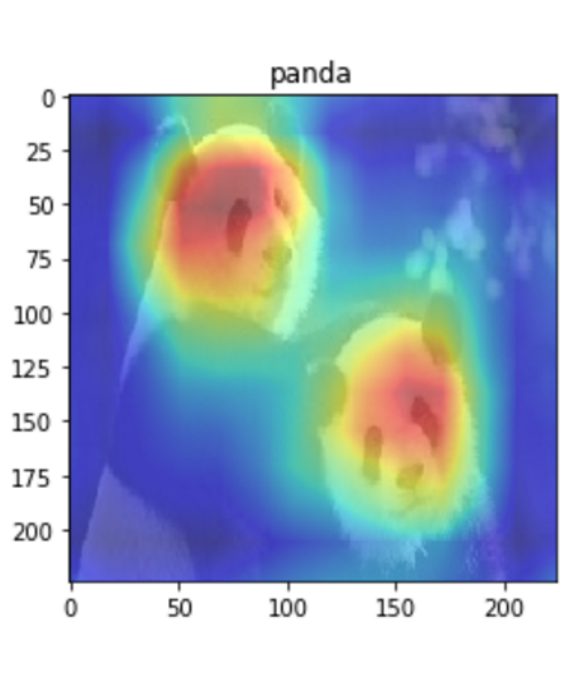
Passing below image with two pandas through the steps above returns a 14x14 activation map





below.

Upsampling the activation map 16 times (224x224) and overlay it on top of the original image shows that the model predicts panda mostly due to the iconic look of the face of pandas. Especially the eyes.



This method offers a near effortless object localization by simply passing an image through a trained CNN and some simple matrix calculation.

9 Conclusion

This study demonstrated how powerful transfer learning is by outperforming a non-trivial CNN model trained from stretch with just a fraction of the training time when the training data is insufficient. Also how pretrained model can be further fine-tuned to obtain massive performance improvement that could not otherwise be achievable if training a new CNN model. We can see huge potentials applying transfer learning in real life applications especially in testing out new ideas quickly without having to collect a huge amount of train data and a long training time.

We have also demonstrated an interesting byproduct of CNN model using Global Average Pooling layer between the last Convolution layer and the Fully Connected output layer. Potentially when dealing with images with multiple objects, we could use some of the highest scored (not just the best) FC weights to build the activation map to obtain the location of objects recognized by the mode then further feed the sub-images containing the objects through the CNN again to obtain the classes for all bounding boxes. Other studies have been made to utilize this method for more robust object localization, such as the study by Singh and Lee [9] where they expand the bounding box by randomly turning off parts of the training images to force the network to learn other parts of the object (not just the most iconic part like the panda face in our example).

Reference

[1] http://yann.lecun.com/exdb/publis/pdf/lecun-99.pdf
[2] https://arxiv.org/pdf/1311.2901.pdf
[3] https://www.cs.princeton.edu/~rajeshr/papers/icml09-ConvolutionalDeepBeliefNetworks.pdf
[4] https://www.csc.kth.se/~heydarma/Datasets.html
[5] https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html
[6] https://www.kaggle.com/wiki/LogLoss
[7] http://cnnlocalization.csail.mit.edu/Zhou_Learning_Deep_Features_CVPR_2016_paper.pdf
[8] https://alexisbcook.github.io/2017/global-average-pooling-layers-for-object-localization/
[9] http://krsingh.cs.ucdavis.edu/krishna_files/papers/hide_and_seek/my_files/iccv2017.pdf

© 2017 Edwin He. All rights reserved.