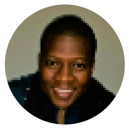


Wisdom d'Almeida Follow
Software Engineer | Machine Learning Connoisseur | Research Student
Jan 4 · 8 min read

Transfer Learning: retraining Inception V3 for custom image classification

Let's experience the power of transfer learning by adapting an existing image classifier (Inception V3) to a custom task: categorizing product images to help a food and groceries retailer reduce human effort in the inventory management process of its warehouse and retail outlets.

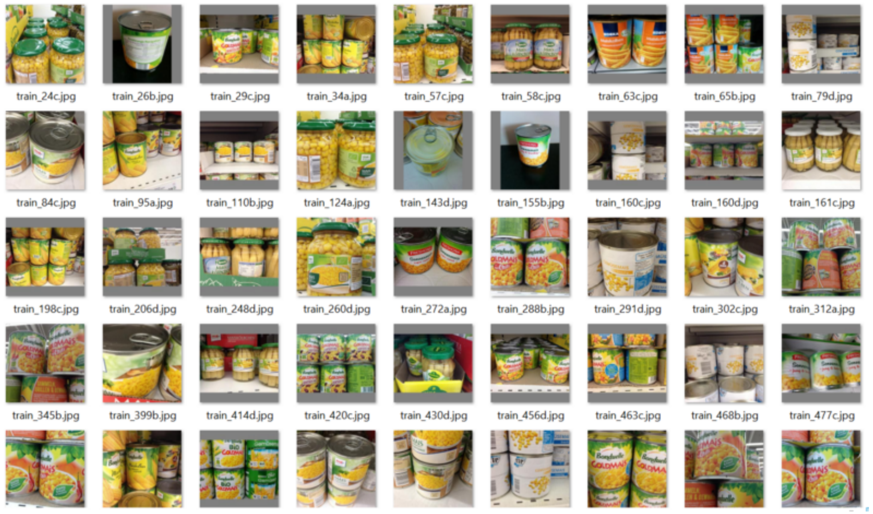
The source code of this work can be found on my GitHub repository below.

wisdal/Image-classification-transfer-learning

Image-classification-transfer-learning - Retraining Google Inception V3 model to perform custom…
github.com

Tools used: TensorFlow v1.1, Python 3.4, Jupyter.

· · ·



Some training instances

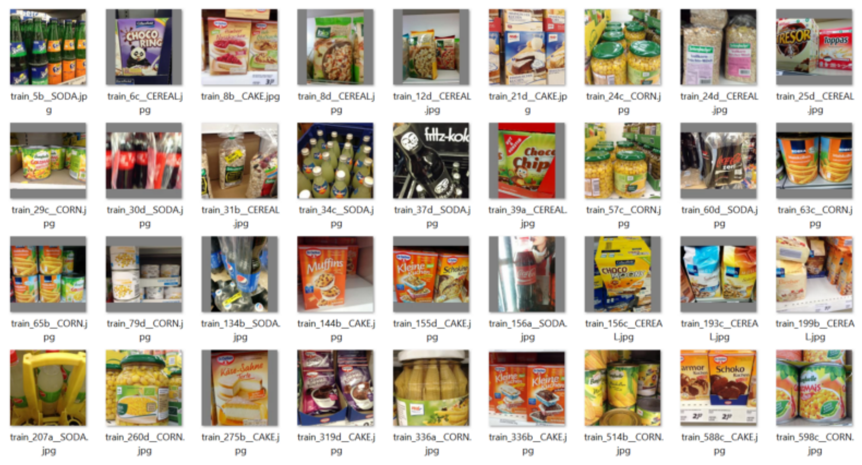
Our problem statement looks very realistic and it turns out it is a real one in fact.

The applications of Deep Neural Networks are literally on a roll. Whether it is in healthcare, transportation or retail, companies across all industries are excited about investing in building intelligent solutions. Meanwhile, let's hope human intelligence remains uncontested :).

Trending AI Articles:

1. *How Neural Networks Work*
2. *ResNets, HighwayNets, and DenseNets, Oh My!*
3. *Machine Learning for Dummies*

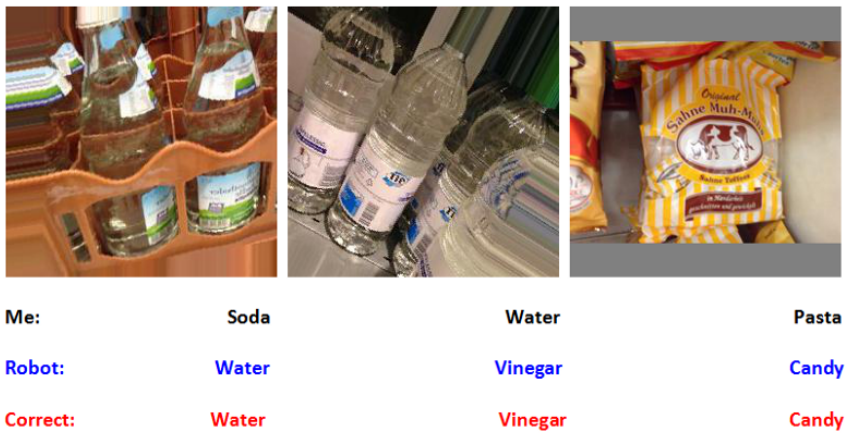
In the actual case, a solution such as a powerful image classifier can help the company track shelf inventory, categorize products, record product volume, etc. from raw products images captured in real time by dedicated devices (drones ? robots ?). And for sure, being able to identify a product and predict its category from a given picture is part of the deal and this is what this experiment is all about. We shall train a bot to predict the category of each product from images.



At the end we should be able to do something like this (note that images are now labelled according to product category)

“What an easy task!”, We think as human beings but it is not necessarily the case for robots. First of all, yes it is obvious that every **adult** should be able to correctly identify a product from its image. The first observation is that this assumption holds true as far as we consider only **adults**. A 5-years-old child, for example, can’t beat an experienced person in this task. And an “experienced person” in this case, when we zoom in, is simply someone who has seen enough product images in his life that he can easily recognize any product he has seen before, from a given image.

This concept of experience is what we try to transfer to robots when we train them with existing labelled data so that they can learn on their own how to accurately distinguish each image from a training dataset. In this sense, we use **Artificial Neural Networks** which are nothing but an imitation of how human brains actually work. The knowledge that robots build using these algorithms are later tested on unlabeled observations. They label given images based on what they have learned and that is why this kind of problem is usually referred to as **supervised learning:** robots try to map a new image instance to one of the class labels they encountered during the training phase. And talking about performance, it has been noticed that well trained robots tend to give better accuracy than humans in most cases of supervised learning. And here, you will be surprised to see how our algorithm can exceptionally outperform humans, under difficult conditions (blurred images, poor quality images, etc.).



Me:	**Soda**	**Water**	**Pasta**
Robot:	**Water**	**Vinegar**	**Candy**
Correct:	**Water**	**Vinegar**	**Candy**

Not very proud of myself :(

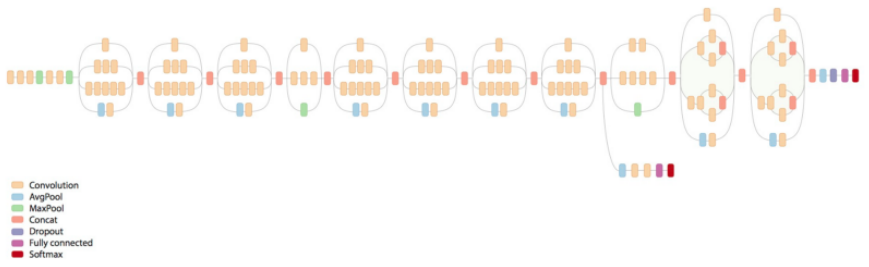
We have a dataset of products images from hackerearth to be downloaded here. How should we proceed and why are we using transfer learning ?

. . .

Why Transfer Learning ?

When we consider classifying images, we often opt to build our model from scratch for the best fit, we say. This is an option but building a deep learning model can take weeks, depending on our training dataset and on the configuration of our network (our deep learning network), not to mention the cost of resources.

But did you know there already exists some models that perform pretty well in classifying images from various categories? If so, you have probably heard of ImageNet, and its Large Visual Recognition Challenge. In this Computer Vision challenge, models try to classify a huge collection of images into 1000 classes, like “Zebra”, “Dalmatian”, and “Dishwasher”. Inception V3 is the model that Google Brain Team has built for the same. Needless to say, the model performed very well.



Schematic diagram of Inception V3

So, can we take advantage of the existence of this model for a custom image classification task like the present one? Well, the concept has a name: **Transfer learning**. It may not be as efficient as a full training from scratch, but is surprisingly effective for many applications. It allows model creation with significantly reduced training data and time by modifying existing rich deep learning models.

Why it works

In a neural network, neurons are organized in layers. Different layers may perform different kinds of transformations on their inputs. Signals travel from the first layer (input), to the last one (output), possibly after traversing the layers multiple times. As the last hidden layer, the “bottleneck” has enough summarized information to provide the next layer which does the actual classification task.

In the retrain.py script, we remove the old top layer, and train a new one on the pictures we have downloaded.

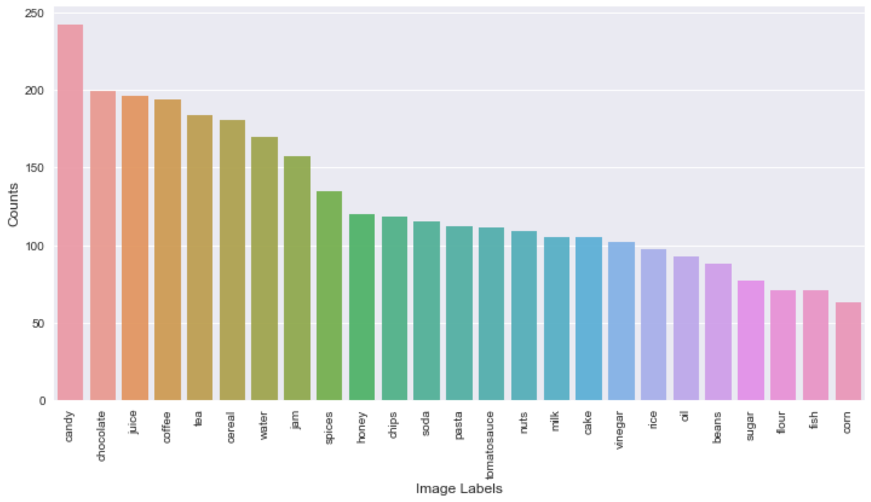
The reason our final layer retraining can work on new classes is that it turns out the kind of information needed to distinguish between all the 1000 classes in ImageNet is often also useful to distinguish between new kinds of objects.

Let’s now get our hands dirty !

. . .

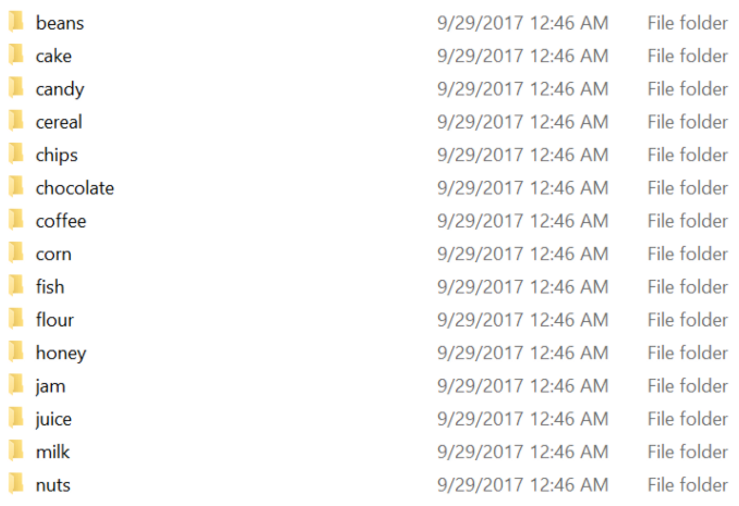
Step 1: Preprocessing images

```
label_counts = train.label.value_counts()
plt.figure(figsize = (12,6))
sns.barplot(label_counts.index, label_counts.values, alpha =
0.9)
plt.xticks(rotation = 'vertical')
plt.xlabel('Image Labels', fontsize =12)
plt.ylabel('Counts', fontsize = 12)
plt.show()
```



Distribution of images

Assuming you have already downloaded the dataset, you will notice it comes with a “train” folder that we need to set up properly. Our goal is to put each image in a subfolder representing its category. At the end, we should have x subfolders, with x the number of distinct categories.



beans	9/29/2017 12:46 AM	File folder
cake	9/29/2017 12:46 AM	File folder
candy	9/29/2017 12:46 AM	File folder
cereal	9/29/2017 12:46 AM	File folder
chips	9/29/2017 12:46 AM	File folder
chocolate	9/29/2017 12:46 AM	File folder
coffee	9/29/2017 12:46 AM	File folder
corn	9/29/2017 12:46 AM	File folder
fish	9/29/2017 12:46 AM	File folder
flour	9/29/2017 12:46 AM	File folder
honey	9/29/2017 12:46 AM	File folder
jam	9/29/2017 12:46 AM	File folder
juice	9/29/2017 12:46 AM	File folder
milk	9/29/2017 12:46 AM	File folder
nuts	9/29/2017 12:46 AM	File folder

Something like this

For this preprocessing purpose, I provide you with the pre_process.ipynb notebook.

```
for img in tqdm(train.values):
    filename=img[0]
    label=img[1]
    src=os.path.join(data_root,'train_img',filename+'.png')
    label_dir=os.path.join(data_root,'train',label)
    dest=os.path.join(label_dir,filename+'.jpg')
    im=Image.open(src)
    rgb_im=im.convert('RGB')
    if not os.path.exists(label_dir):
        os.makedirs(label_dir)
    rgb_im.save(dest)
    if not
os.path.exists(os.path.join(data_root,'train2',label)):
        os.makedirs(os.path.join(data_root,'train2',label))

rgb_im.save(os.path.join(data_root,'train2',label,filename+'
.jpg'))
```

The notebook is not only about configuring images subfolders, so be sure to check it out.

Because our dataset comes with **25 unique labels** while we **only have 3215 training images**, we need to augment the data to prevent our model from over-fitting.

```
datagen = ImageDataGenerator(
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        fill_mode='nearest')

class_size=600

src_train_dir=os.path.join(data_root,'train')
dest_train_dir=os.path.join(data_root,'train2')
it=0
for count in label_counts.values:
    #nb of generations per image for this class label in
order to make it size ~= class_size
    ratio=math.floor(class_size/count)-1
    print(count,count*(ratio+1))

dest_lab_dir=os.path.join(dest_train_dir,label_counts.index[
it])

src_lab_dir=os.path.join(src_train_dir,label_counts.index[it
])
    if not os.path.exists(dest_lab_dir):
        os.makedirs(dest_lab_dir)
    for file in os.listdir(src_lab_dir):
        img=load_img(os.path.join(src_lab_dir,file))
        #img.save(os.path.join(dest_lab_dir,file))
        x=img_to_array(img)
        x=x.reshape((1,) + x.shape)
        i=0
        for batch in datagen.flow(x,
batch_size=1,save_to_dir=dest_lab_dir, save_format='jpg'):
            i+=1
            if i > ratio:
                break
    it=it+1
```

Step 2: retraining the bottleneck and fine-tuning the model

Courtesy of Google, we have the retrain.py script to start right away. The script will download the Inception V3 pre-trained model by default.

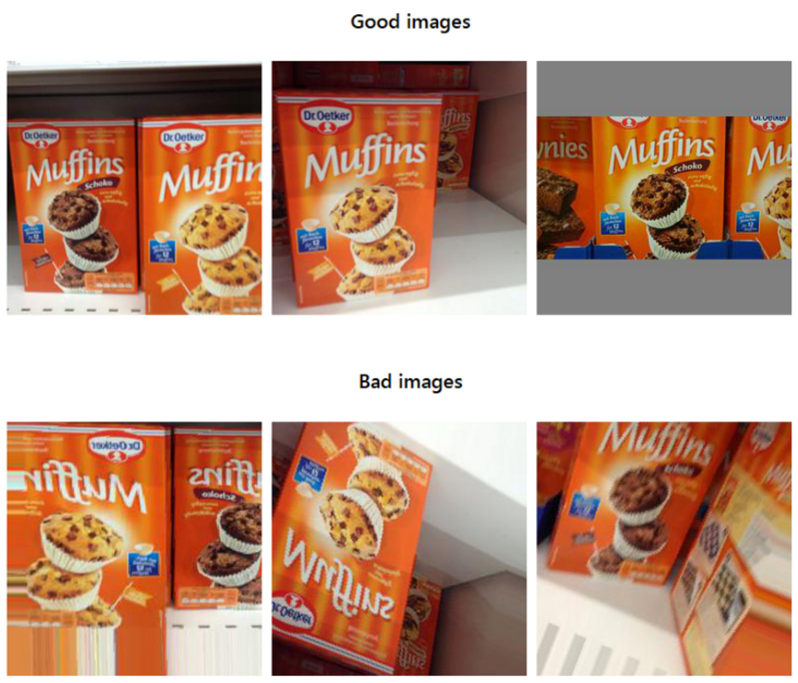
The retrain script is the core component of our algorithm and of any custom image classification task that uses Transfer Learning from Inception v3. It was designed by TensorFlow authors themselves for this specific purpose (custom image classification).

What the script does:

It trains a new top layer (bottleneck) that can recognize specific classes of images. The top layer receives as input a 2048-dimensional vector for each image. A softmax layer is then trained on top of this representation. Assuming the softmax layer contains N labels, this corresponds to learning N + 2048*N (or 1001*N) model parameters corresponding to the learned biases and weights.

The script is perfectly customizable and here is a list of parameters that can be configured:

- **image_dir**: path to folder of labeled images. Fortunately, we properly set it up during preprocessing step.
- **output_graph, intermediate_output_graphs_dir, output_labels, etc.**: where to save output files.
- **distortion feature:** my favorite. This feature alone deserves a whole paragraph. You may have noticed that images in our training set are perfect (clear, of high quality, unambiguous) but this is unfortunately not always the case in production. The algorithm may and will encounter, after being deployed, fuzzy images, sometimes dimly lit, etc.



Our algorithm should be smart enough to catch that these images represent the same thing and it is not that obvious (this is just a small example)

- […] This is what the distortion feature is all about. We intentionally randomly transform images (size, colours, orientation, etc.) in the training process to get the robot accustomed to bad images as well, to avoid loosing prediction accuracy in such circumstances.
- **how_many_training_steps**: number of epochs.
- **learning rate**.
- …

You can play with these parameters as you please. **Learning rate**, **nb. of epochs**, etc. are deterministic parameters. Use them to fine-tune your model and remember that you can use the TensorBoard at anytime to visualize the results of your training.

You may be able to get ~85% of accuracy to start with (with zero fine-tuning).

Step 3: testing the model on unseen records

Nothing crazy to do at this step. Just a little script to test the model built and saved in the previous step, on images in the "test" folder of our dataset.

Check out the test notebook for an overview of what needs to be done.

```
def run_graph(src, labels, input_layer_name,
output_layer_name,
              num_top_predictions):
    with tf.Session() as sess:
    i=0
    #outfile=open('submit.txt','w')
    #outfile.write('image_id, label \n')
    for f in os.listdir(dest):

image_data=load_image(os.path.join(dest,test[i]+'.jpg'))
        #image_data=load_image(os.path.join(src,f))
        softmax_tensor =
sess.graph.get_tensor_by_name(output_layer_name)
        predictions, = sess.run(softmax_tensor,
{input_layer_name: image_data})

        # Sort to show labels in order of confidence
        top_k = predictions.argsort()[-num_top_predictions:]
[::-1]
        for node_id in top_k:
            human_string = labels[node_id]
            score = predictions[node_id]
            #print('%s (score = %.5f) %s , %s' % (test[i],
human_string))
            print('%s, %s' % (test[i], human_string))
            #outfile.write(test[i]+', '+human_string+'\n')
        i+=1
    return 0
```

I see this product is candy
I see this product is juice
I see this product is water
I see this product is candy
0
10
20
30
40
50
60

Testing our model on bad quality images

. . .

Conclusion

That’s it! Hope the article was useful to you. Feel free to comment and suggest improvements.

I encourage you to try this and let me know in comments how much accuracy you were able to achieve. I will be happy to hear from you.

As I said before, you will surely be able to get **more than 85%** of base accuracy. The rest is up to fine tuning! In my case, the accuracy of my final model on the test set blew me away, considering how little work was required. A good understanding of how things work helps sometimes :). I think the project is a good base for anyone who wants to try image distortions or hyper-parameter tuning. This is what added me up more percentage points.

Feel free to have a look at my code on GitHub.

Resources: tensorflow.org/tutorials/image_recognition