



CS 598 FINAL PROJECT - LEGO IMAGE RECOGNITION

KerryAnn DeMeester, Megan Rapach



APRIL 25, 2019

DR. ZHENG
CS 598

Table of Contents

| | |
|--|----|
| Problem Description..... | 2 |
| Approach | 2 |
| Tools & Technologies | 2 |
| Data Collection | 3 |
| Project Structure..... | 3 |
| Tutorial for Use..... | 7 |
| Project Setup..... | 8 |
| Retraining the Network | 8 |
| Additional Information Regarding the Training Process (can skip for tutorial)..... | 9 |
| Using the Retrained Model..... | 11 |
| Results | 12 |

Problem Description

LEGO is famous for selling sets of bricks that can be built into well-known objects. For example, they have sets for certain car models, buildings, and elements from popular movies such as Hogwarts Castle from the Harry Potter series. These sets contain many bricks - large LEGO sets can be comprised of up to 75,000 pieces.

Within a LEGO set, the bricks are usually separated into smaller bags. When these bags are opened, the builder is left with piles and piles of LEGO bricks. On top of being overwhelming, it can be very difficult to find a needed piece in this sea of LEGO pieces. Often, this can be demotivating to builders and rather straining on the mind and body (eyes, neck, etc.).

For our final project, we attempt to find a solution to this problem using image recognition to recognize and classify LEGO bricks based on color and size. This would help alleviate the stress faced by LEGO enthusiasts when searching for needed LEGO pieces in a large pile of bricks.

Approach

As a proof of concept, we trained our image recognition model with images of 4 different color LEGO bricks of 3 sizes.

| Colors: | Sizes: |
|---------|--------|
| Red | 2x2 |
| Yellow | 2x3 |
| Green | 2x4 |
| Blue | |

Tools & Technologies

For our project, we used several tools to implement our algorithm. These tools include **TensorFlow**, **Transfer Learning**, and **TensorBoard**.

TensorFlow is a free, open-source machine learning tool that contains several libraries and other resources for numerical computation machine learning projects. Popular use cases for TensorFlow include voice and sound recognition, text-based analysis applications, image recognition, time series analysis, and video detection. For our project, we use TensorFlow to train a classifier to classify our LEGO bricks by color and size.

Transfer Learning refers to a machine learning method in which a model that has been previously developed is reused as the starting point for a new model. Transfer Learning is a common approach to deep learning, which uses a pre-trained model as a starting point to more quickly develop neural networks to be used on related problems. We used a previously trained model for our LEGO image recognition called ImageNet. ImageNet was already pre-loaded with over 1,000 classes. We added our 12 new classes (one for every combination of color and size LEGO brick) and retrained the model. We then used a neural network called MobileNet to re-train ImageNet into a new model to compare only the new classifiers we had specified from our sample data of LEGO bricks.

To visualize the performance of our image recognition model, we used **TensorBoard**. TensorBoard is a dashboard that shows a graphical representation of the accuracy and cross

entropy of the learning model. This tool was very useful because it allowed us to see how our re-training of the existing model was affected by the addition of our new classes, as well as see how well our model performed in terms of accuracy.

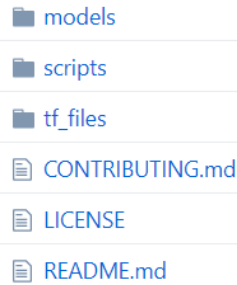
Data Collection

In order to re-train the ImageNet model, we had to provide our own images of the LEGO bricks. We used the photo burst feature on our iPhones to take several pictures of each classifier, then made a directory for each classifier and placed the images in the corresponding folder. We attempted to make the images of each classifier as similar as possible, using the same back-drop (a white sheet of printer paper) and taking the pictures from similar angles at roughly the same distance from each brick. In total, we took around 3,100 images of LEGO bricks!

| Classifier | Number of Pictures Taken |
|------------|--------------------------|
| Blue 2x2 | 233 |
| Blue 2x3 | 252 |
| Blue 2x4 | 229 |
| Green 2x2 | 208 |
| Green 2x3 | 183 |
| Green 2x4 | 221 |
| Red 2x2 | 259 |
| Red 2x3 | 226 |
| Red 2x4 | 321 |
| Yellow 2x2 | 400 |
| Yellow 2x3 | 235 |
| Yellow 2x4 | 349 |

Project Structure

Because our project is rather large, we will not provide all source code in this document. Instead, we will show our project structure and explain important areas. The entire project source code can be accessed at <https://github.com/meganrapach/lego-image-recognition>



The Models folder contains a .txt file that includes all 1,000 labels that came with the pre-trained ImageNet model. This is one of the files that gets affected when we re-train the model with our 12 additional classifiers.

The Scripts folder contains all of the Python scripts required to re-train our model and evaluate test images. This folder also contains a script called label_image.py, which is used to analyze a test image and assigns a label to it. The source code for label_image.py is provided below.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys
import time

import numpy as np
import tensorflow as tf

def load_graph(model_file):
    graph = tf.Graph()
    graph_def = tf.GraphDef()

    with open(model_file, "rb") as f:
        graph_def.ParseFromString(f.read())
    with graph.as_default():
        tf.import_graph_def(graph_def)

    return graph

def read_tensor_from_image_file(file_name, input_height=299, input_width=299,
                               input_mean=0, input_std=255):
    input_name = "file_reader"
    output_name = "normalized"
    file_reader = tf.read_file(file_name, input_name)
    if file_name.endswith(".png"):
```

```

        image_reader = tf.image.decode_png(file_reader, channels = 3,
                                           name='png_reader')
    elif file_name.endswith(".gif"):
        image_reader = tf.squeeze(tf.image.decode_gif(file_reader,
                                                       name='gif_reader'))
    elif file_name.endswith(".bmp"):
        image_reader = tf.image.decode_bmp(file_reader, name='bmp_reader')
    else:
        image_reader = tf.image.decode_jpeg(file_reader, channels = 3,
                                           name='jpeg_reader')

    float_caster = tf.cast(image_reader, tf.float32)
    dims_expander = tf.expand_dims(float_caster, 0);
    resized = tf.image.resize_bilinear(dims_expander, [input_height, input_width])
    normalized = tf.divide(tf.subtract(resized, [input_mean]), [input_std])
    sess = tf.Session()
    result = sess.run(normalized)

    return result

def load_labels(label_file):
    label = []
    proto_as_ascii_lines = tf.gfile.GFile(label_file).readlines()
    for l in proto_as_ascii_lines:
        label.append(l.rstrip())
    return label

if __name__ == "__main__":
    file_name = "tf_files/lego_photos/Yellow 2x2/IMG_4985.jpg"
    #file_name = "tf_files/flower_photos/daisy/3475870145_685a19116d.jpg"
    model_file = "tf_files/retrained_graph.pb"
    label_file = "tf_files/retrained_labels.txt"
    input_height = 224
    input_width = 224
    input_mean = 128
    input_std = 128
    input_layer = "input"
    output_layer = "final_result"

    parser = argparse.ArgumentParser()
    parser.add_argument("--image", help="image to be processed")
    parser.add_argument("--graph", help="graph/model to be executed")
    parser.add_argument("--labels", help="name of file containing labels")
    parser.add_argument("--input_height", type=int, help="input height")

```

```
parser.add_argument("--input_width", type=int, help="input width")
parser.add_argument("--input_mean", type=int, help="input mean")
parser.add_argument("--input_std", type=int, help="input std")
parser.add_argument("--input_layer", help="name of input layer")
parser.add_argument("--output_layer", help="name of output layer")
args = parser.parse_args()
```

```
if args.graph:
    model_file = args.graph
if args.image:
    file_name = args.image
if args.labels:
    label_file = args.labels
if args.input_height:
    input_height = args.input_height
if args.input_width:
    input_width = args.input_width
if args.input_mean:
    input_mean = args.input_mean
if args.input_std:
    input_std = args.input_std
if args.input_layer:
    input_layer = args.input_layer
if args.output_layer:
    output_layer = args.output_layer
```

```
graph = load_graph(model_file)
t = read_tensor_from_image_file(file_name,
                                input_height=input_height,
                                input_width=input_width,
                                input_mean=input_mean,
                                input_std=input_std)
```

```
input_name = "import/" + input_layer
output_name = "import/" + output_layer
input_operation = graph.get_operation_by_name(input_name);
output_operation = graph.get_operation_by_name(output_name);
```

```
with tf.Session(graph=graph) as sess:
    start = time.time()
    results = sess.run(output_operation.outputs[0],
                        {input_operation.outputs[0]: t})
    end=time.time()
```



```

results = np.squeeze(results)














top_k = results.argsort()[-5:][::-1]
labels = load_labels(label_file)

print('\nEvaluation time (1-image): {:.3f}s\n'.format(end-start))
template = "{} (score={:.5f})"
for i in top_k:
    print(template.format(labels[i], results[i]))

```

Finally, the `tf_files` folder contains all our training and test images for the LEGO bricks. This folder is broken into two directories, the training set `lego_photos`, and the test set `test_images`. The test set contains all the images that we used after training the model when making classifications. Each folder is further broken down into subdirectories for each classifier. The folder structure of the `tf_files/lego_photos` directory is shown below.

Branch: master ▾ [lego-image-recognition](#) / [tf_files](#) / [lego_photos](#) /

| | |
|--|----------------------|
|  KerryAnn-DeMeester | Delete .gitignore |
| .. | |
|  Blue_2x2 | Delete .gitignore |
|  Blue_2x3 | Delete .gitignore |
|  Blue_2x4 | Delete .gitignore |
|  Green_2x2 | Add files via upload |
|  Green_2x3 | Delete .gitignore |
|  Green_2x4 | Delete .gitignore |
|  Red_2x2 | first commit |
|  Red_2x3 | added Red_2x3 |
|  Red_2x4 | Added Red_2x4 |
|  Yellow_2x2 | first commit |
|  Yellow_2x3 | added Yellow_2x3 |
|  Yellow_2x4 | Added Yellow_2x4 |

Tutorial for Use

This section provides step-by-step instructions for replicating our project implementation. Note that this can be done for any set of new classifiers, as long as the images are stored correctly under the `tf_files` directory.

Our tutorial uses Linux shell commands, so it is recommended to follow this tutorial using a Linux machine, if possible.

Project Setup

The first thing we must do is install TensorFlow on our machine. This can be done by running the following command.

```
pip install --upgrade "tensorflow==1.7.*"
```

Next, we clone the git repository that contains the required python scripts.

```
git clone https://github.com/googlecodelabs/tensorflow-for-poets-2
cd tensorflow-for-poets-2
```

After we have cloned the git repository, we can add our images that we will use to retrain the model to the `tf_files` folder. First, create a directory under `tf_files` to contain all the training images. We call this `lego_photos`. Under `lego_photos`, create a subdirectory for each of the 12 classifiers. The file structure for `tf_files` should look like this:

```
tf_files/
  lego_photos/
    Blue_2x2/
    Blue_2x3/
    Blue_2x4/
    Green_2x2/
    Green_2x3/
    Green_2x4/
    Red_2x2/
    Red_2x3/
    Red_2x4/
    Yellow_2x2/
    Yellow_2x3/
    Yellow_2x4/
```

Retraining the Network

To retrain the model with our LEGO dataset, we first must configure and retrain the MobileNet neural network. There are two variables we must set to configure the MobileNet.

1. Input image resolution (128, 160, 192, or 224 px). Note that using a higher image resolution will take more processing time, but results in better classification accuracy.
2. The relative size of the model as a fraction of the largest MobileNet (1.0, 0.75, 0.50, or 0.25)

We will use 224 px and 0.25 for this project.

These variables can be set in the Linux shell:

```
IMAGE_SIZE=224
ARCHITECTURE="mobilenet_0.25_${IMAGE_SIZE}"
```

Before we retrain the model, start TensorBoard in the background to monitor the training progress. To run TensorBoard, open a new Linux shell and run the following command.

```
tensorboard --logdir tf_files/training_summaries &
```

Note that the above command will fail if there is already an instance of TensorBoard running. If this is the case, the running instance can be terminated by running

```
pkill -f "tensorboard"
```

Now, we can retrain the neural network. Because we cloned the git repository from codelab, we already have all of the scripts required to do the retraining. All we have to do is run a python script with the settings we would like. Run the following in the Linux shell:

```
python -m scripts.retrain \  
  --bottleneck_dir=tf_files/bottlenecks \  
  --how_many_training_steps=500 \  
  --model_dir=tf_files/models/ \  
  --summaries_dir=tf_files/training_summaries/"${ARCHITECTURE}" \  
  --output_graph=tf_files/retrained_graph.pb \  
  --output_labels=tf_files/retrained_labels.txt \  
  --architecture="${ARCHITECTURE}" \  
  --image_dir=tf_files/lego_photos
```

This script only retrains the final layer of the ImageNet network by downloading the pretrained model, adding a new final layer, and training that layer on the LEGO images we added. While this step may take a while, it should complete in a reasonable amount of time.

Even though the original ImageNet does not contain any of the LEGO pieces we are using to train, the ability to differentiate among 1,000 classifiers is useful in distinguishing between objects. By using this model, we are taking that information as input to the final classification layer that distinguishes our LEGO classifiers.

Note that in the above command we run the script with only 500 training steps. Accuracy can be improved if this number is increased. The default value for this parameter is 4,000 steps.

Additional Information Regarding the Training Process (can skip for tutorial)

Here we discuss in detail how the retraining process works. This section contains no action items or Linux commands.

The retrain script has us specify a directory for the bottleneck values for each image. Bottleneck is an informal term that refers to the layer before the final output layer. This is the layer that does the classification. The term is used to imply that this layer slows down the network. Because this layer is near the output, representation is much more compact here than in the main body of the network.

Because every image is reused several times during retraining, calculations for the layers behind the bottleneck for each image takes a significant amount of time. However, because these lower layers of the network are not being modified, their outputs are cached and reused. Specifying a bottleneck directory for this script saves the values to that directory, caching them for reuse if the model is retrained again.

Once the retrain script finishing generating the bottleneck files, it begins the actual training of the final layer of the neural network. Each step of the training processes randomly selects 10 training images, locates their bottleneck values from the cache directory, and uses them as input to the final layer to generate predictions. These predictions are then compared against the actual labels, and the results of this comparison are used to update the weights of the final layer via backpropagation.

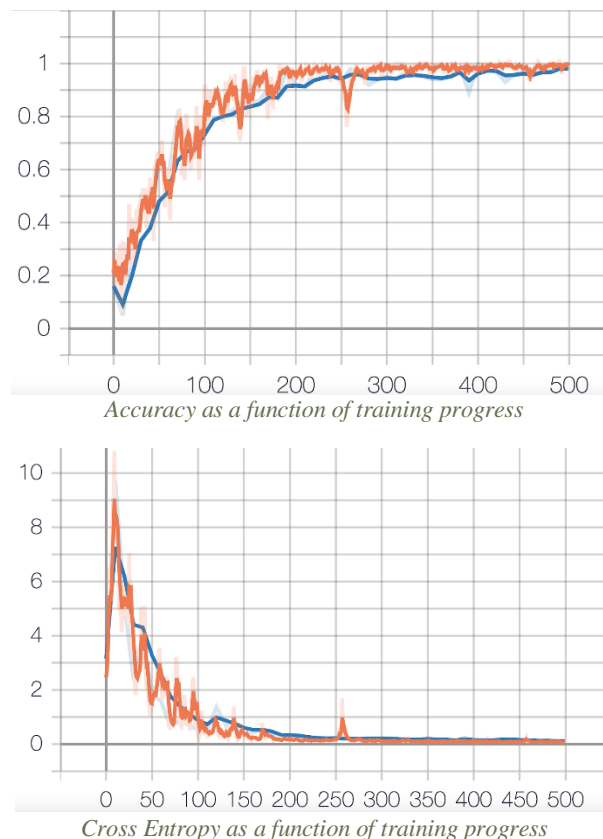
As the training progresses, a series of step outputs will be printed to the Linux shell. Each step's output shows training accuracy, validation accuracy, and cross entropy. A brief description of each of these measures is provided below.

Training accuracy – the percentage of the images used in the current training batch that were labeled with correct classifiers.

Validation accuracy – the precision (the percentage of correctly-labelled images) on a randomly-selected group of images from a different set

Cross entropy – a loss function that gives us insight into how well the learning process is progressing. For this value, lower numbers are better.

Below, we have provided graphs of our model's accuracy and cross entropy as the training progressed. This can be viewed by opening TensorBoard and clicking on the figure's name.



Both images show two lines, one orange and one green. The orange line shows the accuracy and cross entropy of the model on the training data, while the green line shows the validation accuracy and cross entropy on the test set. Because the test set was not used for training, this measure is much more indicative of the performance of the model. If we were to see the case

where the training accuracy continues to rise while the validation accuracy decreases, the model would be “overfitting”. This means the model has begun to memorize the training set instead of understanding the general patterns in the data.

After training completes, the accuracy of the model should be between 85% and 99%. This value varies due to the randomization of the training process.

Using the Retrained Model

The retrain script writes data to two files:

- `tf_files/retrained_graph.pb` – contains a version of the selected network with a final layer retrained on our classifiers
- `tf_files/retrained_labels.txt` – a .txt file containing labels

Now we are ready to test our model with a test image of a LEGO brick. First, we create a new directory under `tf_files` called `test_images` to hold the photos we will use to test. We create subdirectories for each classifier to keep our test images organized. The file structure will resemble that of the training folder. We have produced it below for reference.

```
tf_files/  
  test_images/  
    Blue_2x2/  
    Blue_2x3/  
    Blue_2x4/  
    Green_2x2/  
    Green_2x3/  
    Green_2x4/  
    Red_2x2/  
    Red_2x3/  
    Red_2x4/  
    Yellow_2x2/  
    Yellow_2x3/  
    Yellow_2x4/
```

As an example, save the following image as `IMG_6852.jpg` to `tf_files/Red_2x2`



Once the image is saved to the aforementioned directory, we can run the `label_image.py` script to test the model's ability to classify this image as a Red 2x2 LEGO brick. Run the following command to classify the image.

```
python -m scripts.label_image \
  --graph=tf_files/retrained_graph.pb \
  --image=tf_files/test_images/Red_2x2/IMG_6852.jpg
```

Each execution of this script will print a list of LEGO labels, usually with the correct LEGO classifier on top. Results may look like the following.

```
Evaluation time (1-image): 0.383s

Red 2x2 (score = 0.97167)
Yellow 2x2 (score = 0.02371)
Red 2x3 (score = 0.00324)
Yellow 2x3 (score = 0.00133)
Red 2x4 (score = 0.00002)
```

This result indicates that the model predicted with ~97% accuracy that the image we selected is a Red 2x2 LEGO brick, which is the correct classification.

To test more images, upload additional test images and run `label_image.py` again with the corresponding path to the image.

Results

To test how well we were able to retrain our model to classify LEGO pieces according to the categories we defined, we ran the `label_image.py` script on several training and test images. Our results are provided below.

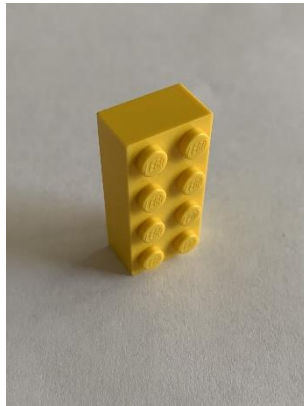
First, we used one training image of each class to verify that the model was able to correctly identify an image of each type of LEGO brick. The model did in fact correctly identify each training image into the appropriate class.

We then tried classifying each category with a test image from its class to see if the algorithm was able to identify the class using an image not included in the training set. The table below summarizes our results.

| Class | Training Image (Control) | Test Image |
|-----------|--------------------------|------------|
| Blue 2x2 | Correct | Correct |
| Blue 2x3 | Correct | Correct |
| Blue 2x4 | Correct | Correct |
| Green 2x2 | Correct | Correct |
| Green 2x3 | Correct | Correct |
| Green 2x4 | Correct | Correct |
| Red 2x2 | Correct | Correct |

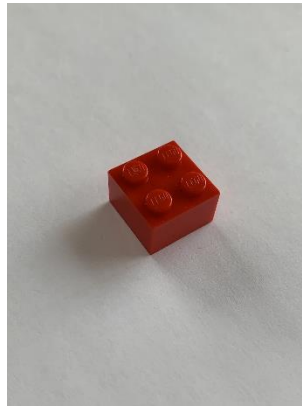
| | | |
|------------|---------|---------|
| Red 2x3 | Correct | Correct |
| Red 2x4 | Correct | Correct |
| Yellow 2x2 | Correct | Correct |
| Yellow 2x3 | Correct | Correct |
| Yellow 2x4 | Correct | Correct |

Because our training images contain photos of LEGO bricks taken from various angles, we decided to test the model's ability to classify bricks that are positioned differently. For this test, we only looked at one brick of every color. We chose Yellow 2x4, Red 2x2, Blue 2x2 and Green 2x3. To test the different angles, we used a test image of a standing Yellow 2x4, an upright Red 2x3 turned with a slight camera angle, an upside down Blue 2x2, and an upright Green 2x3 with almost no camera angle.



Evaluation time (1-image): 0.308s

yellow 2x4 (score=0.95240)
red 2x4 (score=0.04247)
red 2x3 (score=0.00262)
yellow 2x3 (score=0.00245)
yellow 2x2 (score=0.00004)



Evaluation time (1-image): 0.383s

red 2x2 (score=0.97167)
yellow 2x2 (score=0.02371)
red 2x3 (score=0.00324)
yellow 2x3 (score=0.00133)
red 2x4 (score=0.00002)



Evaluation time (1-image): 0.400s

blue 2x2 (score=0.97682)
green 2x2 (score=0.02169)
blue 2x3 (score=0.00108)
red 2x2 (score=0.00036)
green 2x3 (score=0.00004)



Evaluation time (1-image): 0.313s

green 2x3 (score=0.90018)
blue 2x3 (score=0.05721)
green 2x4 (score=0.03480)
blue 2x4 (score=0.00244)
yellow 2x3 (score=0.00233)

Each of these images was classified correctly by the model.