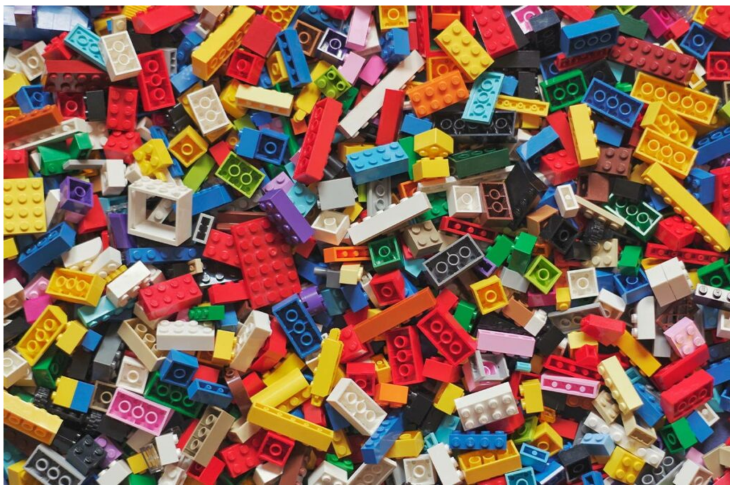
Machine Learning with Tensorflow

opencampus.sh

Classification of LEGO Bricks Based on Images

Project documentation

Jule Kuhn, Daniel Mansfeldt, Tabea Rahm



https://absatzwirtschaft-1cf93.kxcdn.com/wp-content/uploads/2022/08/lego-1024x676.jpg

January 27, 2023

1 Introduction

LEGO is the most well-known and popular brand producing building block systems. It was even ranked the "world's most powerful brand" in February 2015 [1]. In total, there are more than 3700 unique LEGO bricks, with the 10 most common bricks corresponding to 20% of all elements [2, 3]. The number of LEGO parts produced all over the world sums up to over 600 billion (as of 2015 [4]), which corresponds to around 80 bricks per person. Most of these bricks are made from ABS (Acrylonitrile Butadiene Styrene), a polymer that cannot be recycled. Only recently, the LEGO company started to produce their first prototypes of bricks made from recycled plastic bottles. This is, however, still work in progress. The company plans on making their products recyclable from 2030 - an ambition the company itself calls 'bold' [5].
With this in mind, a LEGO sorting system could help to efficiently make use of the existing LEGO bricks by being able to reusing them in the various building kits. Different attempts at such image classifying and brick sorting systems have been made in the past: The app 'BrickIt' was released in 2021 and aims to identify the different LEGO bricks in a picture to then give ideas and instructions of what can be built with them [6]. Various blog posts describe the authors' efforts in programming classifiers for images of LEGO bricks, some of them even built whole sorting machine systems [7, 8, 9].
In this project, our aim was to build and train a neural net that can identify and classify different LEGO bricks in images. The dataset and methods used to design and train the neural net are described in Section 2. Section 4 shows the final model and the results in training, validation and testing in comparison to a simple baseline model as described in Section 3.

2 Data and Methods

This section describes the properties of the used data set as well es the methods for handling the data, e.g. image preprocessing and finding model hyper parameters.

2.1 Dataset

The data which was used for this project is a data set published on kaggle with the title 'Images of LEGO Bricks' [10]. The data set contains computer rendered pictures of 50 different bricks. The rendering was done with the software 'Autodesk Maya 2020' by 800 different angles with a two camera setup, so it's 40,000 pictures in total. The pictures have the dimension of 400 x 400 pixels and 3 RGB channels. A sample image is shown in Figure 1.

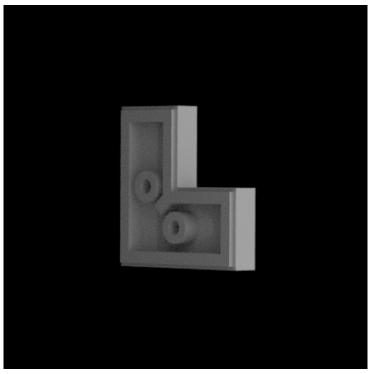


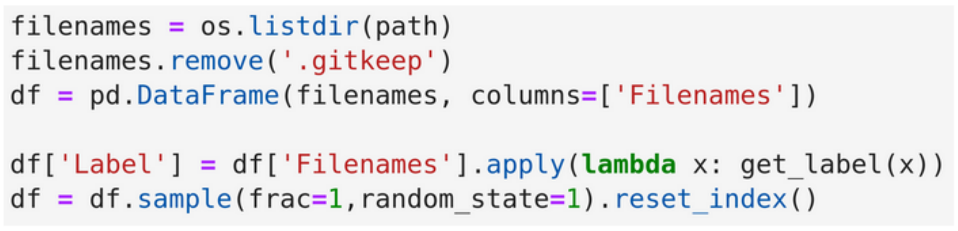
Figure 1: Sample of a rendered brick called 'flat tile corner 2x2'.

2.2 Methods

In this subsection, the methods of image preprocessing and of finding perfect model hyper parameters are described.

2.2.1 Image Preprocessing

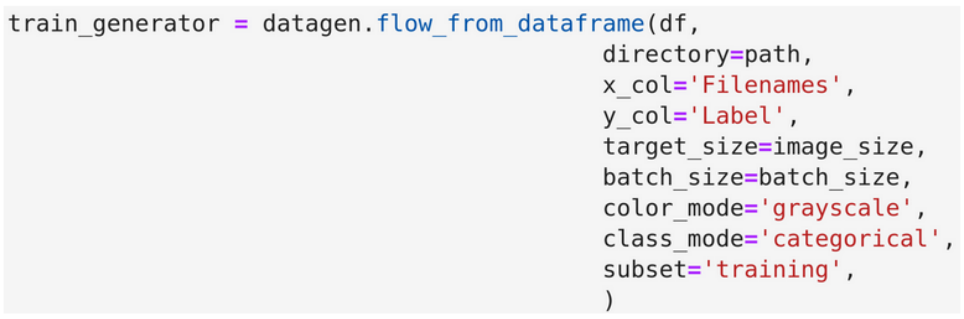
To use the images from the data set for training and validation, the images had to be loaded in a preprocessing step. The 'dataframe' package from the pandas library was used to load the pictures and to get the labels for the class definition from the file names of the pictures. The pandas data frame is shown in Figure 2.



```python
filenames = os.listdir(path)
filenames.remove('.gitkeep')
df = pd.DataFrame(filenames, columns=['Filenames'])

df['Label'] = df['Filenames'].apply(lambda x: get_label(x))
df = df.sample(frac=1,random_state=1).reset_index()
```

Figure 2: Code snippet to load the pictures into a pandas data frame and get the labels from the file names.

After loading the images into a data frame, the training generator and the validation generator is set up by using 'flow from dataframe'. The data generator to do this is shown in Figure 3.



```python
train_generator = datagen.flow_from_dataframe(df,
                                              directory=path,
                                              x_col='Filenames',
                                              y_col='Label',
                                              target_size=image_size,
                                              batch_size=batch_size,
                                              color_mode='grayscale',
                                              class_mode='categorical',
                                              subset='training',
                                              )
```

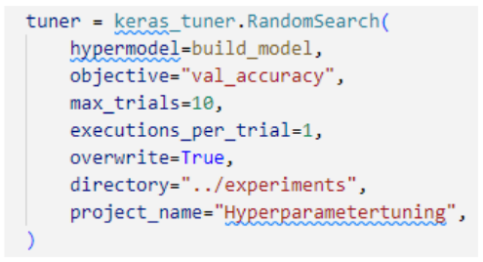
Figure 3: Code snippet to get the training generator and validation generator using 'flow from dataframe'.

2.2.2 Tuner

To find the best hyper parameters for the model, the Keras tuner package was used. The tuner chooses a set of hyper parameters from a defined search space and builds and evaluates multiple models with varied hyper parameters automatically. To build the models, the tuner is provided with an according function. Finally, only the model with the best results in accuracy and loss is kept and retrained.

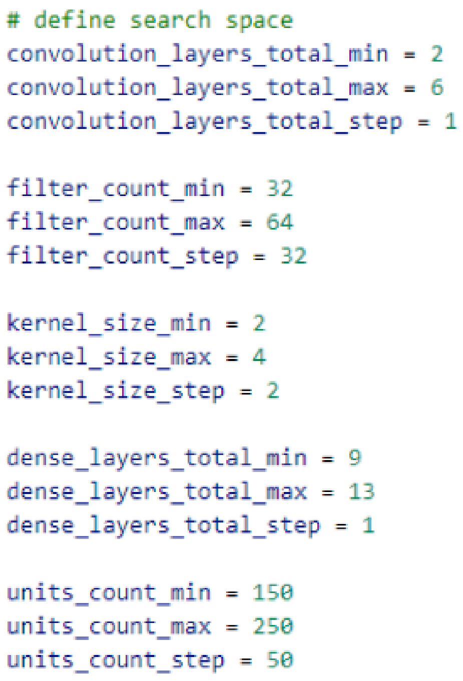
In Figure 4 the tuner, as it was used in this project, is shown.

The initial search space for this project was inspired by the baseline model and is shown in Figure 5. To better approximate the best hyperparameters, the search space was refined manually multiple times.



```python
tuner = keras_tuner.RandomSearch(
    hypermodel=build_model,
    objective="val_accuracy",
    max_trials=10,
    executions_per_trial=1,
    overwrite=True,
    directory="../experiments",
    project_name="Hyperparametertuning",
)
```

Figure 4: Definition of the tuner as it was used in this project.



```
# define search space
convolution_layers_total_min = 2
convolution_layers_total_max = 6
convolution_layers_total_step = 1

filter_count_min = 32
filter_count_max = 64
filter_count_step = 32

kernel_size_min = 2
kernel_size_max = 4
kernel_size_step = 2

dense_layers_total_min = 9
dense_layers_total_max = 13
dense_layers_total_step = 1

units_count_min = 150
units_count_max = 250
units_count_step = 50
```

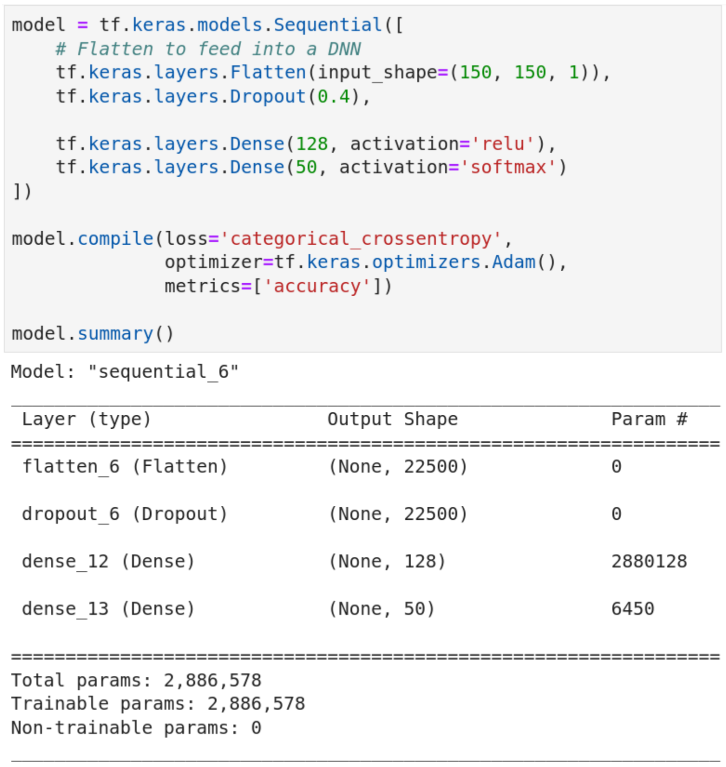
Figure 5: Definition of a search space as an initial base for the tuner.

3 Baseline Model

We designed a simple baseline model to be used as a benchmark for the training of further model architectures as described in Section 4. The model summary is shown in Figure 6. As can be seen, the baseline model is a simple dense neural network (DNN) with one dropout and one hidden dense layer. The final dense layer consists of 50 neurons to represent the 50 different LEGO bricks. Since the problem at hand is a classification task, the chosen loss function is categorical cross-entropy, and the Adam optimizer is used with the default learning rate of 0.001.

We decided to monitor training and validation accuracy to assess the model's quality. The resulting curves as a function of training epoch are shown in Figure 7 along with training and validation loss. Training was stopped when the validation loss was not improving anymore for 10 epochs by implementing an early stopping callback.

Already with this simple DNN architecture, the model is able to classify around 50% of the images in the validation set correctly.



```
model = tf.keras.models.Sequential([
    # Flatten to feed into a DNN
    tf.keras.layers.Flatten(input_shape=(150, 150, 1)),
    tf.keras.layers.Dropout(0.4),

    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(50, activation='softmax')
])

model.compile(loss='categorical_crossentropy',
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])

model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_6 (Flatten)         (None, 22500)             0

 dropout_6 (Dropout)         (None, 22500)             0

 dense_12 (Dense)            (None, 128)               2880128

 dense_13 (Dense)            (None, 50)                6450

=================================================================
Total params: 2,886,578
Trainable params: 2,886,578
Non-trainable params: 0
_________________________________________________________________
```

Figure 6: Summary of the baseline model listing the different model layers in the dense neural network.

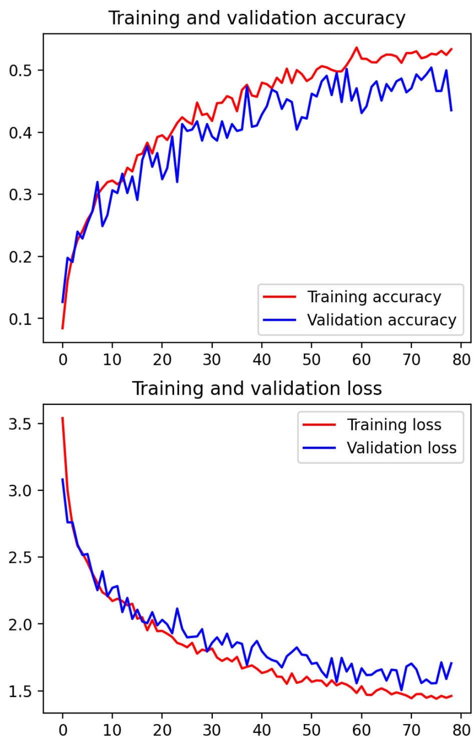


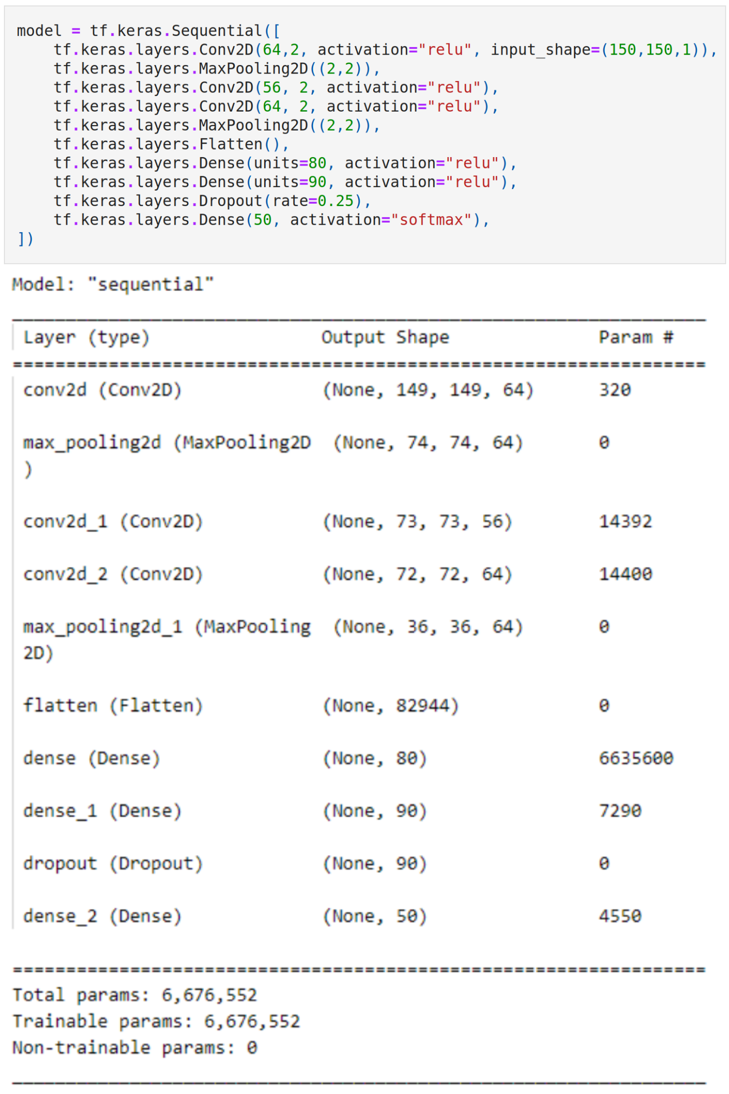
Figure 7: Accuracy and loss of the baseline model as a function of training epoch.

4 Results

This section describes the final model including training and validation, testing, and problems which occurred during designing and using the model.

4.1 Final Model

The summary of the final model that was obtained with the help of the Keras hyperparameter tuner is shown in Figure 8. The hyperparameters were found by the tuner after refining the search space manually a few times. The final model is a convolutional neural network (CNN) consisting of three convolutional layers with 64, 56 and 64 filters, respectively, two 2x2 pooling layers (one after the first and the second after the last convolutional layer), two hidden dense layers and a dropout layer before the output layer, which is the same as in the baseline model.
By including convolutional layers, we hoped to improve the model's performance by highlighting the distinct features of the different brick types.



```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64,2, activation="relu", input_shape=(150,150,1)),
    tf.keras.layers.MaxPooling2D((2,2)),
    tf.keras.layers.Conv2D(56, 2, activation="relu"),
    tf.keras.layers.Conv2D(64, 2, activation="relu"),
    tf.keras.layers.MaxPooling2D((2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=80, activation="relu"),
    tf.keras.layers.Dense(units=90, activation="relu"),
    tf.keras.layers.Dropout(rate=0.25),
    tf.keras.layers.Dense(50, activation="softmax"),
])
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 149, 149, 64)      320

 max_pooling2d (MaxPooling2D  (None, 74, 74, 64)       0
 )

 conv2d_1 (Conv2D)           (None, 73, 73, 56)        14392

 conv2d_2 (Conv2D)           (None, 72, 72, 64)        14400

 max_pooling2d_1 (MaxPooling  (None, 36, 36, 64)       0
 2D)

 flatten (Flatten)           (None, 82944)             0

 dense (Dense)               (None, 80)                6635600

 dense_1 (Dense)             (None, 90)                7290

 dropout (Dropout)           (None, 90)                0

 dense_2 (Dense)             (None, 50)                4550

=================================================================
Total params: 6,676,552
Trainable params: 6,676,552
Non-trainable params: 0

_________________________________________________________________
```

Figure 8: Summary of the final CNN

4.2 Training and Validation

Again, training and validation accuracy and loss were monitored to assess the model's performance. The resulting plots are shown in Figure 9. These plots are the curves obtained after retraining the model a couple of times, always reusing the already trained model weights. We can see that validation accuracy is stable just below 70% after the first few training epochs. Training accuracy, however, rises above 90%, which indicates overfitting especially in combination with the increase in validation loss after the first few epochs.

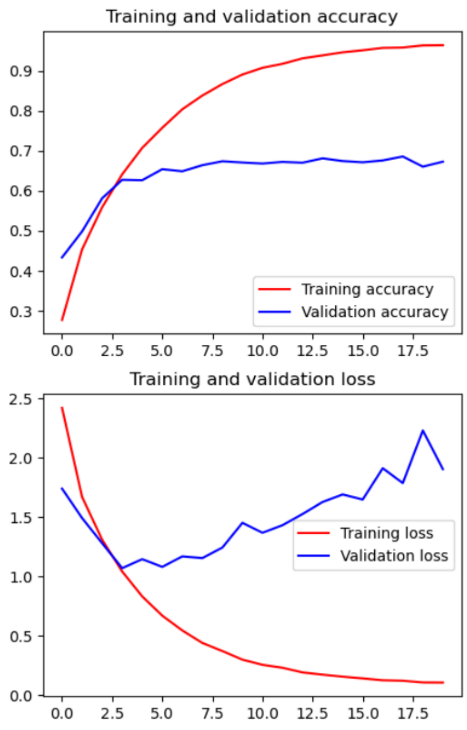
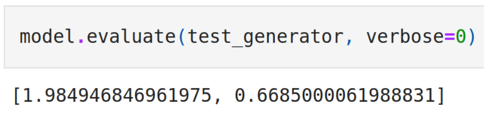


Figure 9: Training and validation accuracy and loss of the final CNN as a function of training epoch for the last round of retraining.

10% of the dataset were used as a test set to evaluate the model's performance on previously unseen data. The resulting metrics can be seen in Figure 10. The test accuracy is 67% and comparable to the final validation accuracy.
Hence, we can say that the final model's performance is considerably improved compared to the baseline model and that it performs comparably well on unseen pictures in the test set.



```
model.evaluate(test_generator, verbose=0)
```

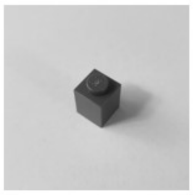
```
[1.984946846961975, 0.6685000061988831]
```

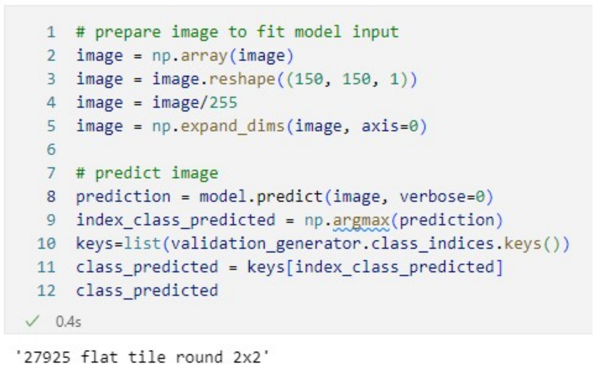
Figure 10: Test loss and accuracy of the final CNN.

4.3 Testing the Model with real pictures

To test the trained model, two pictures each of 7 different real LEGO bricks were taken with a conventional smartphone camera. The pictures were cut to squared format and set to black and white, before given to the model to predict its class. The following screenshots show the prediction results of the pictures. In general, none of predictions were right, but the predictions varied and were at least related to the real class of the respective brick. Moreover, the orientation of the brick on the picture didn't have an impact on the prediction result, so the predicted class stayed the same for every picture of the same brick.

Pictures of a 1x1 and a 2x1 brick were classified as 'flat tile round 2x2', while pictures of a 3x2 brick, a flat 6x2 brick, a 8x1 brick, a 2x2 roof brick and a round pin brick were all predicted to be a 'brick 2x4'.



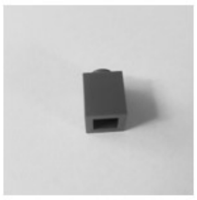


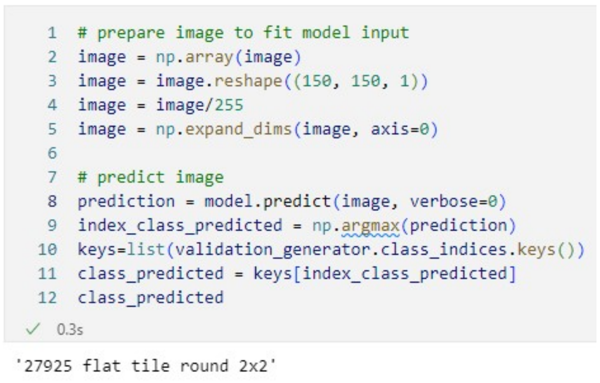
```python
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

✓ 0.4s

```
'27925 flat tile round 2x2'
```

(a)





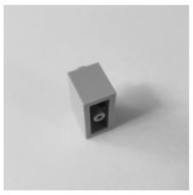
```python
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

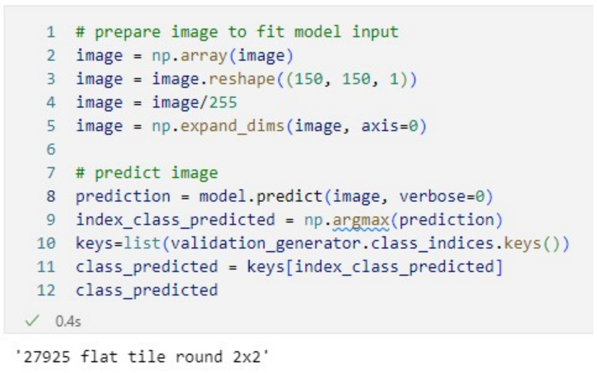
✓ 0.3s

```
'27925 flat tile round 2x2'
```

(b)

Figure 11: The screenshots show the model predicting a 1x1 brick being a 'flat tile round 2x2', which is not right.



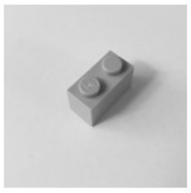


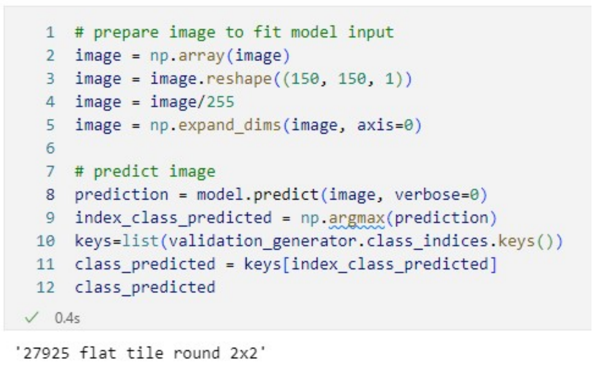
```python
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

✓ 0.4s

```
'27925 flat tile round 2x2'
```

(a)





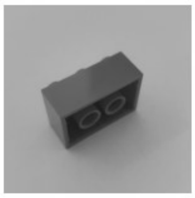
```python
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

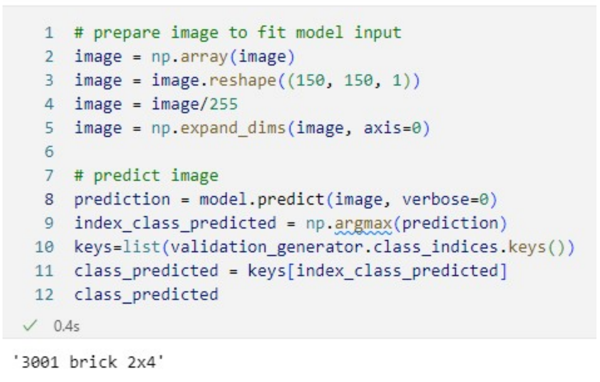
✓ 0.4s

```
'27925 flat tile round 2x2'
```

(b)

Figure 12: The screenshots show the model predicting a 2x1 brick being a 'flat tile round 2x2', which is not right.



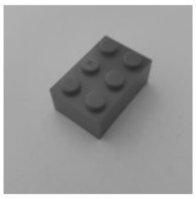


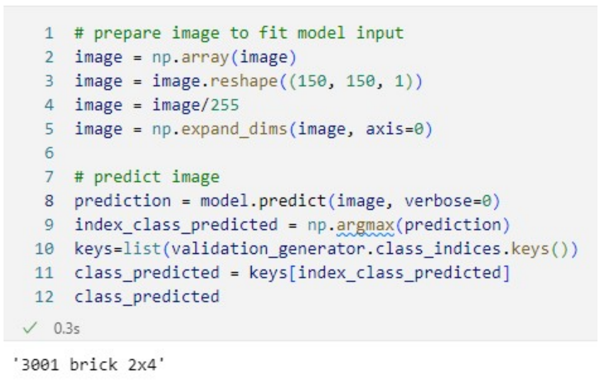
```python
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

✓ 0.4s

```
'3001 brick 2x4'
```

(a)





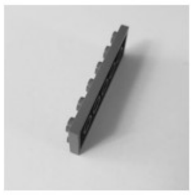
```python
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

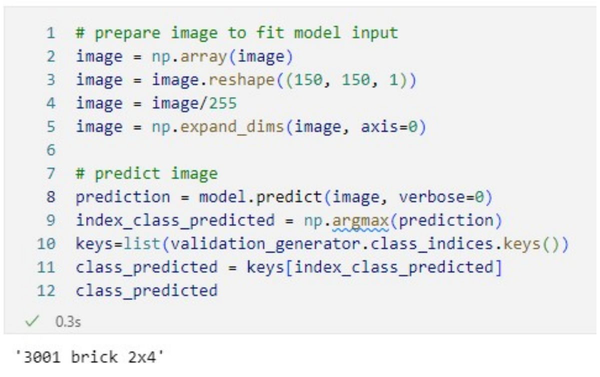
✓ 0.3s

```
'3001 brick 2x4'
```

(b)

Figure 13: The screenshots show the model predicting a 3x2 brick being a 'brick 2x4', which is not right.



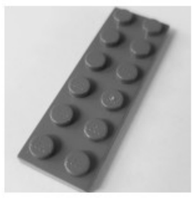


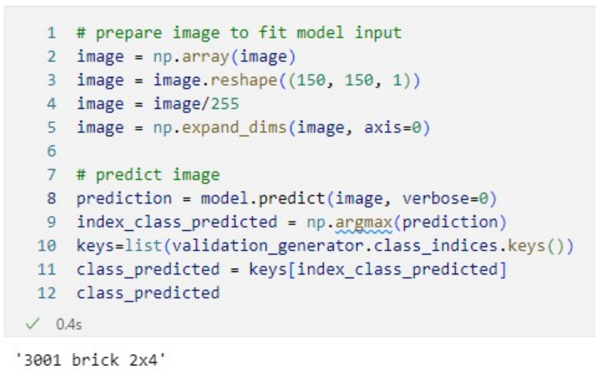
```python
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

✓ 0.3s

```
'3001 brick 2x4'
```

(a)





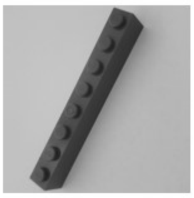
```python
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

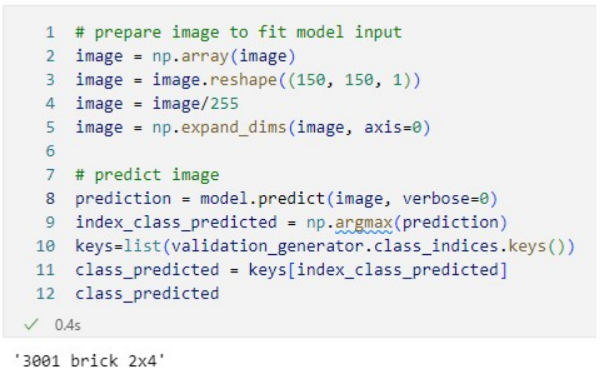
✓ 0.4s

```
'3001 brick 2x4'
```

(b)

Figure 14: The screenshots show the model predicting a flat 6x2 brick being a 'brick 2x4', which is not right.



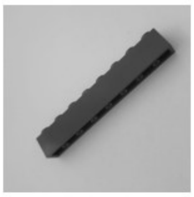


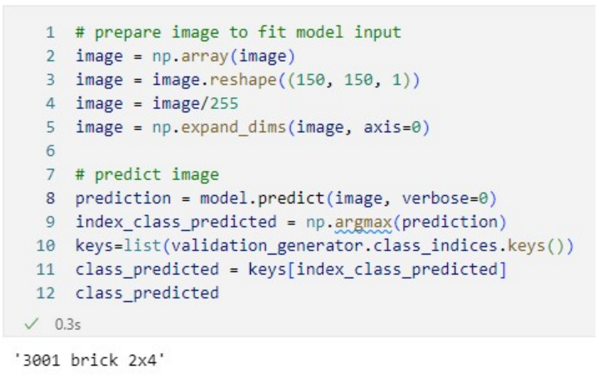
```
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

✓ 0.4s

```
'3001 brick 2x4'
```

(a)





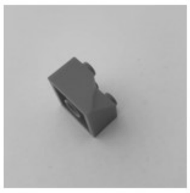
```
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

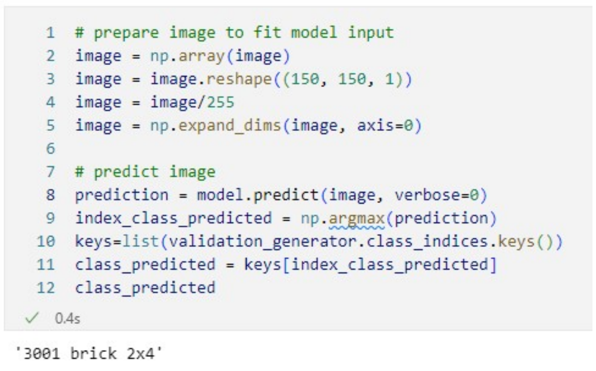
✓ 0.3s

```
'3001 brick 2x4'
```

(b)

Figure 15: The screenshots show the model predicting a 8x1 brick being a 'brick 2x4', which is not right.



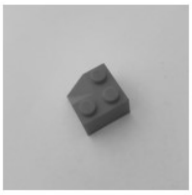


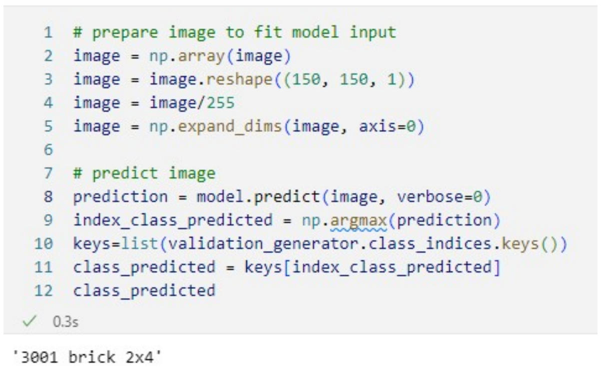
```
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

✓ 0.4s

```
'3001 brick 2x4'
```

(a)





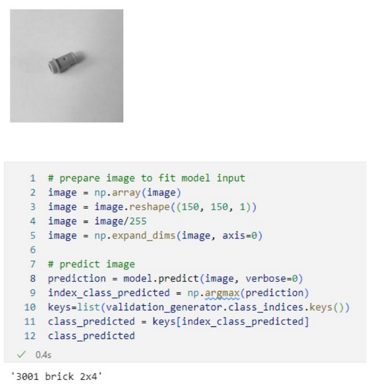
```
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```

✓ 0.3s

```
'3001 brick 2x4'
```

(b)

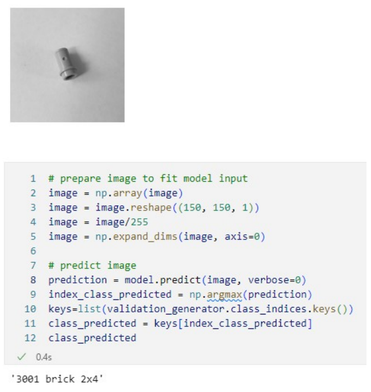
Figure 16: The screenshots show the model predicting a 2x2 roof brick being a 'brick 2x4', which is not right.



```
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```
✓ 0.4s

```
'3001 brick 2x4'
```

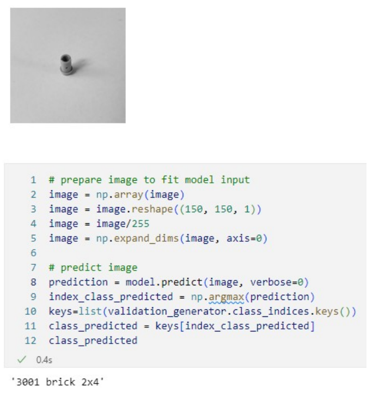
(a)



```
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```
✓ 0.4s

```
'3001 brick 2x4'
```

(b)



```
# prepare image to fit model input
image = np.array(image)
image = image.reshape((150, 150, 1))
image = image/255
image = np.expand_dims(image, axis=0)

# predict image
prediction = model.predict(image, verbose=0)
index_class_predicted = np.argmax(prediction)
keys=list(validation_generator.class_indices.keys())
class_predicted = keys[index_class_predicted]
class_predicted
```
✓ 0.4s

```
'3001 brick 2x4'
```

(c)

Figure 17: The screenshots show the model predicting a round pin brick being a 'brick 2x4', which is not right.

4.4 Problems

During the implementation of this project, some problems occurred, which are going to be mentioned in this subsection.

One problem was inexplicable overfitting. It turned out, that this was because the training and the validation generators contained completely different classes. Shuffling the data could solve this issue.

Another error which happened several times, was that previously saved models could not be loaded (Error: 'RuntimeWarning: Unexpected end-group tag: Not all data was converted'). Unfotunately there was no solution found for this issue so far.

Another issue was the definition of a reasonable grid of hyper parameters to prevent a running out of memory error. This had to be figured out by trial and error.

Additionally, sometimes the models were too large and too many pooling layers led to an error, when trying to reduce the image further than possible.

References

[1] Kathryn Dill. *Lego Tops Global Ranking Of The Most Powerful Brands In 2015.* URL: https://www.forbes.com/sites/kathryndill/2015/02/19/lego-tops-global-ranking-of-the-most-powerful-brands-in-2015/. (Access date: 23.01.2023).

[2] Tom Alphin. *2019 Most Common LEGO Parts.* URL: https://brickarchitect.com/2019/2019-most-common-lego-parts/. (Access date: 20.01.2023).

[3] Toby Woolcock. *100 Fun Facts You Probably Didn't Know About Lego.* URL: https://bricksfans.com/lego-facts/. (Access date: 20.01.2023).

[4] Dan Anthony. *The message is the medium.* URL: https://ipo.blog.gov.uk/2016/12/13/the-message-is-the-medium/. (Access date: 23.01.2023).

[5] LEGO. *Recycled materials.* URL: https://www.lego.com/en-gb/sustainability/environment/recycledmaterials. (Access date: 20.01.2023).

[6] BrickIt. *Build new things from your old bricks.* URL: https://brickit.app/. (Access date: 20.01.2023).

[7] Alejandro J Rod. *Part II-Tensorflow model training of a lego bricks image classifier using mobilenetv2.* URL: https://medium.com/lego-brick-image-classification-running-on-jetson/tensorflow-implementation-of-lego-bricks-image-classifier-using-mobilenetv2-782284a6446e. (Access date: 20.01.2023).

[8] C. Thomas Brittain. *Training a CNN to Classify LEGOs.* URL: https://ladvien.com/lego-deep-learning-classifier-cnn/. (Access date: 23.01.2023).

[9] Cameron Coward. *Use TensorFlow and Raspberry Pi to Build an Automatic LEGO Sorter.* URL: https://www.hackster.io/news/use-tensorflow-and-raspberry-pi-to-build-an-automatic-lego-sorter-4f43840ba9ab. (Access date: 23.01.2023).

[10] Joost Hazelzet. *Images of LEGO Bricks.* URL: https://www.kaggle.com/datasets/joosthazelzet/lego-brick-images. (Access date: 20.01.2023).