

Queen's

Master of Management in Artificial Intelligence

MMAI 894

Deep Learning

Prof. Ofer Shai

Team Project

March/23/2020

Team Humphrey:

Mia Bragilovski, Jimi Li, Edgar Allik, Imad Jawadi

EXECUTIVE SUMMARY

Our team has originally aimed to solve one business problem (case 1) but had to move on to identifying and solving another business problem (case 2). The second case proved to be more suitable for a classic deep learning application. Following are the summaries of business cases 1 and 2.

Case 1

Overview

For the final team project, our team aimed to develop an algorithm that will predict future financial performance and financial position of a company and detect anomalies in end-users' input of the expected results. The algorithm will apply to datasets of companies' historical results and project financial performance and financial position using data extracted from clients' ERP and CRM systems.

Objectives

The first objective is to develop an algorithm that would use the extracted financial or customer data and extrapolate results into the future within the multi-dimensional model and across all metadata objects. It will PREDICT financial performance (i.e., Profit or Loss) and financial position (i.e., Balance Sheet / Cash Flow positions) of the company.

The second objective is to develop an algorithm that would utilize data from:

- a. The generated results of the first algorithm and
- b. The end-user input

and then compare the two to DETECT anomalies in end-user inputted data as compared to the predicted results. It will then require justification or correction to the entered values. This action will apply to the same metadata in the same multi-dimensional model allowing end-users to adjust expected financial performance (Profit or Loss) and financial position (Balance Sheet / Cash Flow) of the company. Metadata is presented in Appendix – please refer to Image #1.

Results

We have done extensive research and brainstorming and have concluded that classic machine learning techniques such as an autoregressive integrated moving average (ARIMA) would garner better results for the tabular dataset we have available for this business case. Although we have millions of records in our dataset, this is insufficient for deep learning applications due to the time-series nature of the data.

In statistics and econometrics, and in time series analysis, an ARIMA model is a generalization of an autoregressive moving average model¹. Both models are fitted to time series data either to better

¹ https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average

understand the data or to predict future points in the series, i.e., to forecast. ARIMA can be implemented in both Python and R as well as software packages available on the market.

Since we wanted to apply deep learning techniques and not compromise on real-life applicability for business, we have come up with an alternative business case – Case 2.

Case 2

Overview

Our team aims to develop an image classifier that will identify unique Lego parts. We are planning to record a video from a 3D rendered model of rotating Lego parts and break them down into thousands of separate images for training. We are then planning to apply CNN and transfer learning techniques to classify real-world Lego images taken by a Smartphone. The algorithm will apply to images of Lego parts of the same color and taken against a solid monolithic background.

Application in Business

We believe this algorithm will get widespread adoption in business if deployed as an automated store assistant for home improvement and general hardware stores such as Home Depot, Lowe's, etc. Upon entering a store customers would now be able to bring in a part they want to find a replacement for, take a picture of this part and be directed to the correct location in the store where they can find the part without having to walk around the store or look for a human assistant to help them. If the part is not in stock, a customer would be directed to a nearby store where this part can be found.

Furthermore, customers may choose to not come into a store but instead take a picture of the part at home and upload it to a store portal for search across all locations and to check in-stock inventory and prices. The consumers save time by foregoing a human assistant who may or may not know the location of the part and in-stock levels, and by being able to locate the part across multiple locations. This business concept may potentially be extended to a unified search engine and a platform available to consumers across all competing stores thereby allowing consumers to not only easily locate the part but also to compare prices.

Automation of store assistants can improve customer experience and save operating expenses for the store as human assistants will no longer be required to assist with consumers inquiries of this nature. Companies will have more visibility into customer preferences and the timing of the demand for parts and products, i.e., the higher the search count for a part - the higher the demand is for this part.

Another aspect of the business case is a financial gain that the industry (the stores) will achieve by operationalizing the model. Operating expenses of the two hardware stores (Home Depot and Lowe's) are presented below in Exhibit 1. We can see that operating expenses, of which labor cost is a major part, comprise a significant portion of total revenues and these expenses continue to climb year over year for both companies (years 2017-2019 were analyzed).

Once a company has access to consumer searches, it will be in a better position to manage demand and, thereby, prices of their products. Staying competitive is largely dependent on having access to timely and accurate data supplied directly by consumers. Automated store assistant and, at a later stage, a search engine and a platform will allow companies to

1. Improve their bottom line and
2. Get access to timely data and to manage demand and competitive pricing.

Store	Financials	2019	2017	Increase / (Decrease) %
<i>The Home Depot</i>				
	Total Revenue	\$ 108,203,000	\$ 94,595,000	14.39%
	Selling, General and Administrative Expenses	\$ 19,513,000	\$ 17,132,000	13.90%
	as % of Total Revenue	18.03%	18.11%	
<i>Lowe's Companies</i>				
	Total Revenue	\$ 71,309,000	\$ 65,017,000	9.68%
	Selling, General and Administrative Expenses	\$ 17,413,000	\$ 15,129,000	15.10%
	as % of Total Revenue	24.42%	23.27%	

Exhibit # 1 – please refer to supporting Images # 2 and 3 in the Appendix

The benefit of autogenerating training set of images is that we do not have to have thousands of images per class for training as they can be generated from a 3D model and augmented by real images of parts. The downside is that we need to have a 3D model available for the parts that we want to classify. We are assuming that most engineering and technology departments would already have those models (for example, <https://www.mcmaster.com/>).

Sample Image of an autogenerated part is presented in Appendix – please refer to Image #4.

PREPROCESSING

Our data generation and preprocessing steps included:

1. Selecting Lego pieces (classes)
2. Generating 3D data
3. Processing 3D images
4. Generating real images (total of 36 images for 3 objects/classes)
5. Processing real images to align their attributes with autogenerated images
6. Splitting into Train/validate/test datasets
7. Image processing before training

1 | Selecting Lego pieces (classes)

We have originally secured 25 physical gray-colored Lego pieces that we have corresponding 3D models for. The piece selection criteria included both having a variety of different pieces (shapes, sizes, etc.) but also having similar looking pieces to ensure the classifier was trained to recognize the differences, albeit small. We chose only gray images as we did not want the classifier to focus / rely on color or shade of gray for image identification. Each Lego piece has a unique identifier which we will be using as the name of the object/class. Please refer to Image # 5 – The Catalog in the Appendix for a full list of pieces selected.

The 3D model has generated 38,880 images for each of the 25 pieces. 3 parts were selected to manually generate 12 images using different rotations and angles in order to augment our training set as the project progressed. These 3 (three) parts were selected randomly for testing of the model.

2 | Generating 3D data

We have used Studio by BrickLink (version 2.1.1), a popular free Lego 3D modelling application. Each of the Lego pieces was selected manually and uploaded into the software. A copy of every 3D piece was stored in a separate '3d_model' folder. A sample image of a 3D generated piece is shown below.

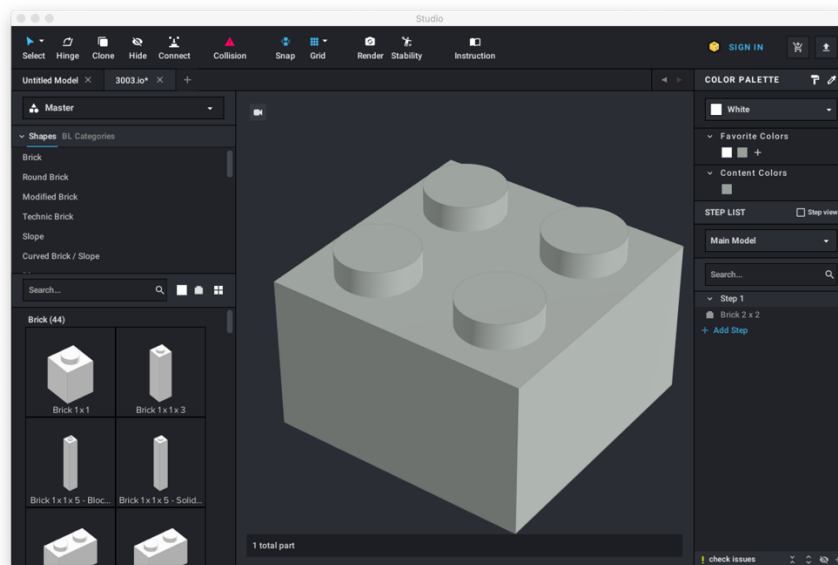


Exhibit # 1: Sample Lego piece 3003 in Studio

We then proceeded to generate 36 batches of 30 rotations each, for a total of 1,080 images per piece ('a piece' is used interchangeably with 'an object' and 'a class'). This entailed manually setting up each of the 36 batches (i.e. adjusting the X-axis value by 10-degrees) and running the batch process. Each batch generated 30 Y-axis rotated images ($360/30 = 12$ -degree increments).

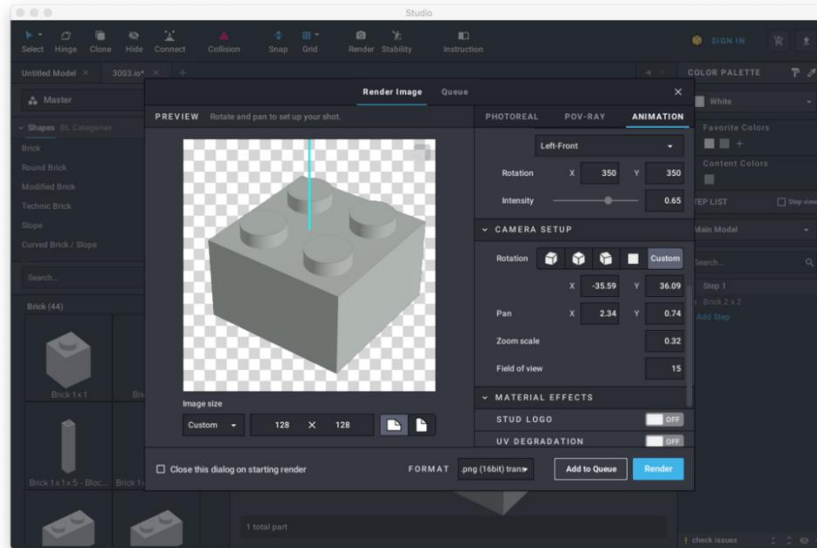


Exhibit # 3: Sample batch setup in Studio

Each image was saved as a 128x128 PNG file, of about 20KB each. Each batch took about 15 minutes to set up and another 30 minutes to render. However, this process can also be automated in the future. These images were saved in folder 'Dataset_base', which itself included subfolders per object named after the piece's ID. We have included additional subfolders for each of the batches under each class's subfolder.

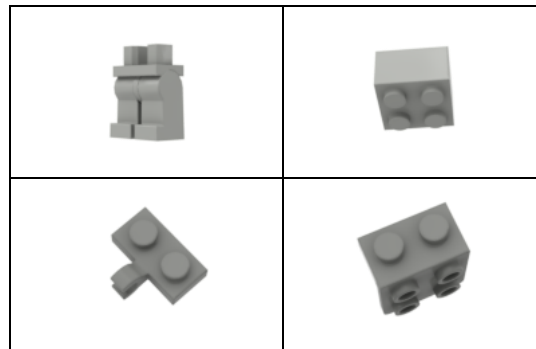


Exhibit # 2: Sample generated 3D images

3 | Processing 3D images

We still needed rotations along the Z-axis, which was achieved by rotating each of the 1,080 generated 360-degrees in 10-degree increments. Each 3D generated image generated 36 more images, so the total 3D image dataset increased to $1,080 \times 36 = 38,880$ images per object/class.

All images from 'Dataset_base' were saved to a new folder named 'Dataset_Aug' and Python script 'Image Prep Step 1.py' was executed¹. This script utilizes the PIL (Python Imaging Library) to perform

¹ https://github.com/ea5055/humphrey894/tree/master/image_prep

image rotation. We have used bicubic resampling to enlarge the images and to avoid pixelation.

```
def rotate_image(im, deg):  
    return im.rotate(deg, resample=PIL.Image.BICUBIC)
```

With 25 classes, the total size of 'Dataset_Aug' expanded to 972,000 files taking a combined total space of 7.5GB.

4 | Real image generation

To ensure the quality of our model and classification results we have decided to augment our dataset with real images. Below are the steps taken to generate 36 real images of 3 Lego objects with 12 images taken per object.

1. Selected 3 Lego pieces (with images already autogenerated) that are grey in color:




		
ID 3001	ID 3701	ID 10247

Exhibit # 5: 3 Lego pieces selected to augment the training set with real images

2. Taking pictures of Lego pieces one by one in a controlled environment. We have taken 12 pictures per Lego, 3 rotations per face. Below is the description of the tools used and the environment:

- a. DSLR + Lens
 - i. Canon 5D Mark III
 - ii. Canon 50mm F1.2L
- b. Light Source
 - i. 6000K Neutral White Circular Light Source (chosen to minimize shadows and highlights)
- c. Resolution
 - i. 5760×3840 (22.3 effective megapixels) in the entire frame
- d. Camera Settings
 - i. Shutter Speed of 1/60
 - ii. ISO of 100
 - iii. Aperture of f5
- e. Distance between camera and the Lego piece is fixed using a tripod

- f. Stationary hold (using a tripod)
- g. The same center, i.e. of how the Lego is positioned in each photograph
- h. White Background (from a letter-sized white sheet of paper)

The environment is as depicted below:

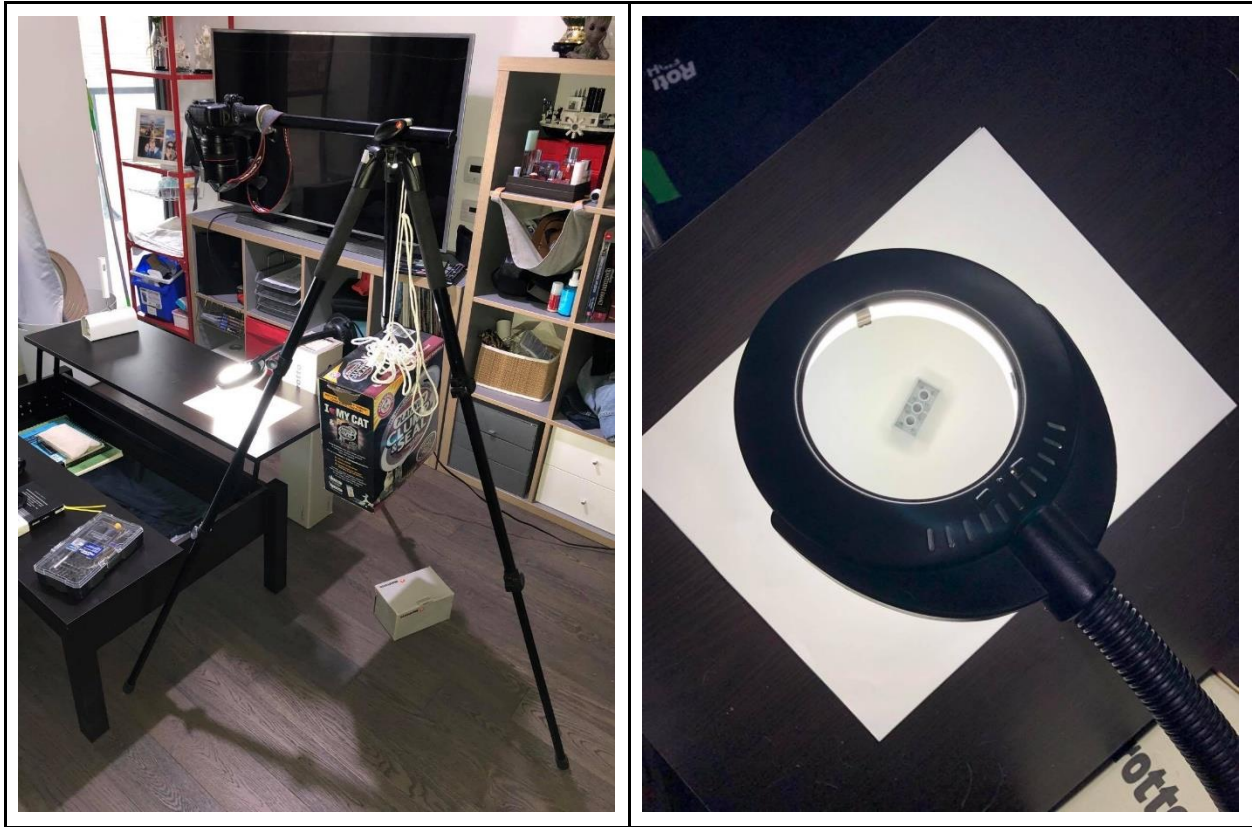


Exhibit # 6: The environment in which the 36 real images were taken

- 3. Cropped each photo such that each cropped image is:
 - a. Completely square
 - b. Lego piece location at the center with spacing like the generated images








































ID	Images Taken with rotations (manually)	Original Image (see Catalog)
3001	           	
3701	           	
10247	           	

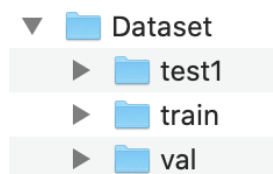
Exhibit # 7: All 36 real images added to the training set of autogenerated images

5 | Train/validation/test split

We have then created a python script (Image Prep Step 2.py file found on our GitHub page¹) to split the images and to place them in a proper folder structure for Keras' **ImageDataGenerator** function consumption. The script loops through all images in 'Dataset_Aug' folder and copies them to a new folder named 'Dataset'. It performs the following steps:

- 1) Obtains a list of all the files in 'Dataset_Aug'
- 2) Each file is assigned to a train/test/val code as per parameters (i.e. 60/30/10 split)
- 3) Each file is copied into a 'Dataset' folder while
 - a) collapsing the sub-folders under each category, and
 - b) renaming the file appropriately so there are no collisions.

The final 'Dataset' structure looks as follows. First level is the train/validate/test split into sub-folders:



Each of the train/validate/test subfolders contains sub-folders for all objects:

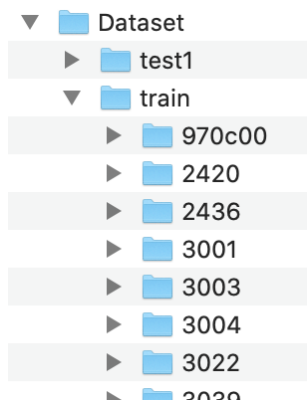


Exhibit # 8a: The file folder structure generated for each of the 25 images

Each class/object sub-folder contains all images for the class without any additional subfolders:

¹ https://github.com/ea5055/humphrey894/tree/master/image_prep





















Name	Type
 10247_anim_2_anim_00	PNG File
 10247_anim_2_anim_00_rot10	PNG File
 10247_anim_2_anim_00_rot20	PNG File
 10247_anim_2_anim_00_rot30	PNG File
 10247_anim_2_anim_00_rot50	PNG File
 10247_anim_2_anim_00_rot60	PNG File
 10247_anim_2_anim_00_rot70	PNG File
 10247_anim_2_anim_00_rot80	PNG File
 10247_anim_2_anim_00_rot90	PNG File
 10247_anim_2_anim_00_rot100	PNG File
 10247_anim_2_anim_00_rot110	PNG File
 10247_anim_2_anim_00_rot120	PNG File
 10247_anim_2_anim_00_rot130	PNG File
 10247_anim_2_anim_00_rot140	PNG File
 10247_anim_2_anim_00_rot150	PNG File
 10247_anim_2_anim_00_rot160	PNG File
 10247_anim_2_anim_00_rot170	PNG File
 10247_anim_2_anim_00_rot180	PNG File
 10247_anim_2_anim_00_rot200	PNG File
 10247_anim_2_anim_00_rot210	PNG File

Exhibit # 8b: Files created for each autogenerated image (with various rotations)

DESIGN AND EXPERIMENTATION

For this section please refer to DL Research Diary sheet in the Appendix (Image #6). GitHub links for the code can be found here:

https://github.com/ea5055/humphrey894/blob/master/hyperparameter_tuning.py

https://github.com/ea5055/humphrey894/blob/master/model_evaluation.py

https://github.com/ea5055/humphrey894/blob/master/transfer_learning.py

We will be applying deep learning (DL) techniques as they are most suitable for dealing with large datasets and for classifying images. The following three cases need to be taking into account before employing DL techniques:

1. Deep Learning (DL) does not work well with small data. To be effective, deep networks require large datasets (i.e. hundreds of thousands of inputs)
2. Deep Learning in practice is complex and expensive. Deep learning is still a very cutting-edge technique.
3. Deep networks are not easily interpreted (i.e. these are so called closed-box models)

Therefore, we can use deep learning techniques because:

- a. We are not concerned with interpretability for commercial deployment (as long as the model classifies images correctly, consumers do not need to know how it is done),
- b. We have large dataset suitable for deep networks and
- c. Once we compile the model it can be used across all stores / industries / consumers, thereby keeping maintenance costs low.

Once the business problem was established and the tools and techniques to solve it were identified, we proceeded to the design phase and development of the architecture of our network.

Our **objective** is to detect Lego parts in an environment with fixed lighting conditions based on minimal number of manually taken images (12 images per class) by using mostly (99%) 3D model autogenerated images. This ensures commercial feasibility of the model as we cannot expect the industry be taking images of each part it holds in stock. This would be cost prohibitive.

Our dataset is the autogenerated images and the real images that have the same attributes, i.e. color, scale and resolution. We are experimenting only with autogenerated images at first to see if we can deliver good prediction results on images the model has never seen before, i.e. the real images consumers would take of the parts they are looking for. We would then have to tune hyperparameters to optimize performance.

Transfer Learning. We will also be applying transfer learning technique towards the end of the experimentation to see if it will improve performance in any way in terms of training speed or accuracy.

Transfer learning focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. The advantages of transfer learning are that, there is no need in an extremely large training dataset or computational power as we are using pre-trained weights and must learn only the weights of the last few layers.¹

By using a pre-trained network to do transfer learning, we are simply adding a few dense layers at the end of the pre-trained network and learning what combination of these already learnt features helps in recognizing the objects in our testing dataset. Hence, we are training only a few dense layers. Furthermore, we can use a combination of these already learnt features to recognize other objects. This technique helps in making the training process very fast and requires less training data compared to training a convolutional network from scratch.

At the same time, we need to ensure that applying transfer learning the model does not lead to deterioration in performance, i.e., only positive transfer should be accepted.

¹ <https://medium.com/analytics-vidhya/transfer-learning-life-just-got-easier-11805a375039>

Experimentation pertinent information is presented below.

- Data was split in the following way. Red-highlighted text denotes real images (manually taken pictures), testing phase 2 has the same image set as testing phase 1 for splits 3-6 below. The first 2 splits follow 80-10-10 and 60-30-10 split, respectively.

#	Split	Training Phase	Validation Phase	Testing Phase 1	Testing Phase 2
1	80%-10%-10%	31211	3869	3800	12
2	60%-30%-10%	23247	11735	3898	12
3	100%-0%-0%	38880	8	4	
4	100%-0%-0%	38880+8	3	1	
5	60%-0%-0%	23247+8	3	1	
6	60%-0%-0%	23247+4	7	1	

Exhibit # 9 – Split details with number of images in each phase

- Batch Sizes
 - 16, 32, 64
- Resolution per image (generated and real)
 - 16x16
 - 64x64
 - 256x256
 - 224x224 (higher resolutions required for Transfer Learning, i.e. the lowest limit)
- Noise Level added on top of Generated Images
 - 0%, 20%, 50%
- Learning Rate
 - 0.001, 0.0001
- Number of nodes per layer
 - 32, 64, 128
- Dropout Rate
 - 0.25, 0.5
- Kernel Size
 - 3,4
- Strides
 - 1,2
- Max Pool Size
 - 2,3

Steps taken during hyperparameter tuning can be traced to hyperparameter_tuning.py on our GitHub page: https://github.com/ea5055/humphrey894/blob/master/hyperparameter_tuning.py

1. Started with one set of hyperparameters for a run
2. Created an image generator aimed to generate batches of images for training / validation /test phase 1/test phase 2 sets from each subfolder:

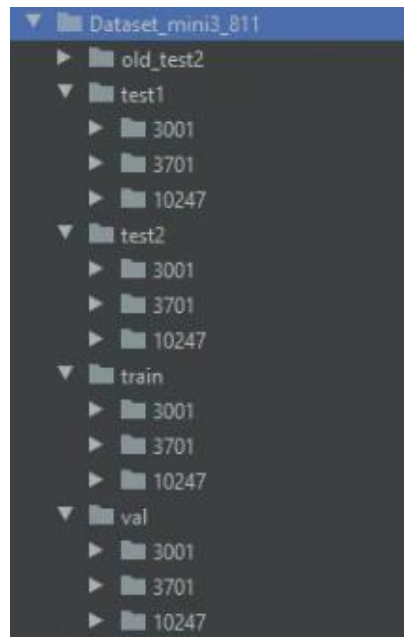


Exhibit # 10 – File structure for training/validation/test set splits

3. Built a model with the architecture as referenced in the industry best practices (best-performing architecture for the MNIST digits dataset case – please see Exhibit #11 below).
4. Saved any model with best-yet validation accuracies over all prior epochs
5. Applied an early-stop during training if validation accuracy stopped improving over half of the maximum upper limit of epochs. An early stop was disregarded when the maximum epoch was set to 5
6. Saved metrics and hyperparameters used on TensorBoard per run
 - a. Metrics include:
 - i. Validation accuracy/loss per epoch
 - ii. Training accuracy/loss per epoch
 - iii. Training /Validation/Test1/Test2 accuracies at the end of training (last epoch run)
 - iv. Average time spent training per epoch
 - b. Hyperparameters – as described above.
7. Repeat step 1 to 6 until every combination of hyperparameter combinations is processed. We have applied grid-search technique to accomplish this.

```

model = Sequential()

model.add(Conv2D(hparams[HP_NUM_UNITS_CL1], kernel_size=hparams[HP_KERNEL_SIZE_CL1], strides=hparams[HP_STRIDES_CL1], input_shape=(hpar
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Conv2D(hparams[HP_NUM_UNITS_CL2], kernel_size=hparams[HP_KERNEL_SIZE_CL2], strides=hparams[HP_STRIDES_CL2], padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(MaxPool2D(pool_size=hparams[HP_POOL_SIZE_PL1]))
model.add(Dropout(hparams[HP_DROPOUT_CL2]))

model.add(Conv2D(hparams[HP_NUM_UNITS_CL3], kernel_size=hparams[HP_KERNEL_SIZE_CL3], strides=hparams[HP_STRIDES_CL3], padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Conv2D(hparams[HP_NUM_UNITS_CL4], kernel_size=hparams[HP_KERNEL_SIZE_CL4], strides=hparams[HP_STRIDES_CL4], padding='same'))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(MaxPool2D(pool_size=hparams[HP_POOL_SIZE_PL2]))
model.add(Dropout(hparams[HP_DROPOUT_CL4]))

model.add(Flatten())
model.add(Dense(hparams[HP_NUM_UNITS_DL1]))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(hparams[HP_DROPOUT_DL1]))

model.add(Dense(hparams[HP_NUM_UNITS_DL2]))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dropout(hparams[HP_DROPOUT_DL2]))

model.add(Dense(num_class, activation='softmax'))
adam = keras.optimizers.Adam(learning_rate=hparams[HP_OPTIMIZER_PARAM], beta_1=0.9, beta_2=0.999)

model.compile(optimizer=adam, loss='categorical_crossentropy', metrics=['accuracy'])

```

Exhibit # 11 – Network Architecture

Experimentation steps

For each of the experiments we are predicting the class of the image (one of the 3 classes mentioned above). Hence, the accuracy of a random guess is 33.3%. Below is a summary of the results by combination of hypermeters used. All images included in experiments AA through NN belong to 1 class.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1						HYPERPARAMETERS							PERFORMANCE RESULTS				
2	Hyperparameters ->>> Steps	Training Phase (# of images)	Validation Phase (# of images)	Testing Phase 1 (# of images)	Testing Phase 2 (# of images)	Dataset with Real Images	% of real images used in training	Batch Size	Resolution	lr-2layermodel/01layermodel	Noise Level %	Best Time (seconds)	Best Epoch / Total Epochs	Best Train / Validation Accuracy %	Real Image Dataset	Real Image Accuracy %	Comment
3	AA	31211	3869	3800	12	Test2	0.0%	16,64,256	16,32,64	0.001/128/64	0.00%	84	5/5	100% / 99%	test2	33.00%	everything did well
4	BB	31211	3869	3800	12	Test2	0.0%	16,64,256	16,32,64	0.001/128/64	20.00%	71	5/5	100% / 99%	test2	42.00%	res 64 + batch 64 did poorly
5	CC	31211	3869	3800	12	Test2	0.0%	16,64,256	16,32,64	0.001/128/64	50.00%	57	5/5	100% / 99%	test2	39.00%	res 16 did poorly
6	DD	23247	11735	3898	12	Test2	0.0%	16,64,256	16,32,64	0.001/128/64	0.00%	50	5/5	100% / 99%	test2	33.00%	res 64 did poorly
7	EE	23247	11735	3898	12	Test2	0.0%	16,64,256	16,32,64	0.001/128/64	20.00%	61	5/5	100% / 99%	test2	39.00%	res 16 did poorly
8	FF	23247	11735	3898	12	Test2	0.0%	16,64,256	16,32,64	0.001/128/64	50.00%	128	5/5	100% / 99%	test2	39.00%	res 16 did poorly
9	GG	38880	8		4	Val, Test1, Test2	0.0%	16,64,256	16,32,64	0.001/128/64	0.00%	70	5/5	100% / 89%	val	33.00%	everything did poorly
10	HH	38880	8	3	1	Train, Val, Test1, Test2	0.2%	16,64,256	16,32,64	0.001/128/64	20.00%	19	40/50	100% / 89%	val	89.00%	res 64 + ~256 batch did well
11	II	38880	8	3	1	Train, Val, Test1, Test2	0.2%	16,64,256	16,32,64	0.001/128/64	0.00%	20	32/50	100% / 100%	val	90.00%	batch 64 did well
12	JJ	38880	8	3	1	Train, Val, Test1, Test2	0.2%	16,64,256	16,32,64	0.0001/512/1024	50.00%	23	41/50	100% / 78%	val	78.00%	batch 64 did well
13	KK	23247	8	3	1	Train, Val, Test1, Test2	0.03%	16,64,256	16,32,64	0.001/128/64	0.00%	113	21,40,43/50	100% / 89%	val	89.00%	batch 64 did well
14	LL	23247	4	7	1	Train, Val, Test1, Test2	0.017%	16,64,256	16,32,64	0.001/128/64	0.00%	32	24/50	100% / 57%	val	57.00%	res32+batch256 did well
15	MM	23247	4	7	1	Train, Val, Test1, Test2	0.017%	64	224	0.001/128/64	0.00%	908	2/50	100% / 66%	val	66.00%	TRANSFER LEARNING
16	NN	38980	8	3	1	Train, Val, Test1, Test2	0.2%	64	224	0.001/128/64	0.00%	85	15/50	98% / 62%	val	78.00%	
17																	
18	NOTES:																
19	all red numbers denote manually taken images																
20	AA-CC	applying 8/1/1 split for different combinations of hyperparameters (noises, 3 resolutions, 3 batch sizes)															
21	DD-FF	applying 6/3/1 split for different combinations of hyperparameters (noises, 3 resolutions, 3 batch sizes)															
22	GG	applying 1/0/0 split and adding 12 real images to the training, validation and testing phases															
23	HH-JJ	using only 10% of the autogenerated images and adding 8 real images to the training phase and remaining 4 - to validation and testing phases															
24	... etc.																
25	The total of 12 real images all belong to 1 class.																

Exhibit # 12 – Experimentation Research Diary sheet (also included in Appendix)

Detailed descriptions and observations from the experimentation is as follows:

- Steps (AA-GG)** We have tried different noise levels on different splits for a max of 5 epochs per run and have found that
 - Although adding noise to the autogenerated images helped, prediction accuracy did not climb above 42% (with 33% being a random guess since we are trying to classify 3 Lego parts as a Proof of Concept). This situation changes once real images are added to the training set
 - Adding noise slows down training time per epoch
 - Time per epoch for the best performing model is 50 seconds
 - We are overfitting on a training set

For Steps (DD-FF) We have changed the split and included more into validation to avoid overfitting.
- Steps (HH-JJ)** We have added 8 real images to the training set with 10% of the autogenerated images (3,888 images in training set) for a max of 50 epochs per run and have managed to reach 90% accuracy on real images.
 - when noise is added to the autogenerated images, the results become worse (accuracy drops to 78%)
 - when the batch size of 64 is used, best results are observed
 - time per epoch for the best performing model is 19 seconds
 - Peak train-validation performance occurred at 32/50 epochs
- (KK)** We have tried adding 8 real images to the training set with 60% of generated images (0.03% real images in the training set) for a max of 50 epochs per run and managed to achieve 89% accuracy on real images.
 - Time it took to train the model has significantly increased, now at 113 seconds (compared to 20 seconds in step II)
 - Peak train-validation performance occurred at 21/50 epochs

4. **(LL)** Here we have reduced the number of real images from 8 to 4 (one for each side), for a max of 50 epochs per run and have only achieved 57% accuracy on real images.
 - a. The time per epoch for the best performing model is 32 seconds
 - b. Rotating real images to obtain different angles helped the model significantly
 - c. Peak train-validation performance occurred at 24/50 epochs
5. **(MM)** Tried transfer learning NASnetmobile with the frozen ImageNet weights with similar data setup as in step LL and have achieved only 66% accuracy on real images (still better than in step LL). The model trained for 50 epochs.
 - a. time per epoch for the best performing model is 506 seconds, which is cost prohibitive and will prevent any commercial adoption.
6. **(NN)** Tried transfer learning NASnetmobile with frozen ImageNet weights with similar data setup as in step LL and have achieved only 78% accuracy on real images. The model trained for 50 epochs.
 1. time per epoch for the best performing model is 85 seconds

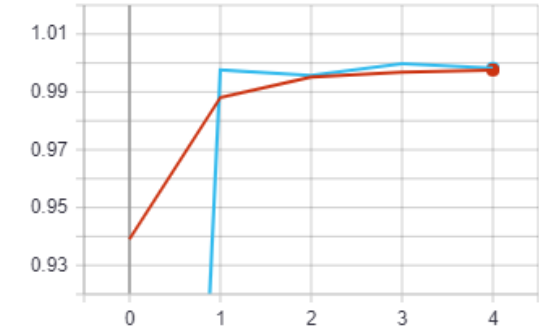
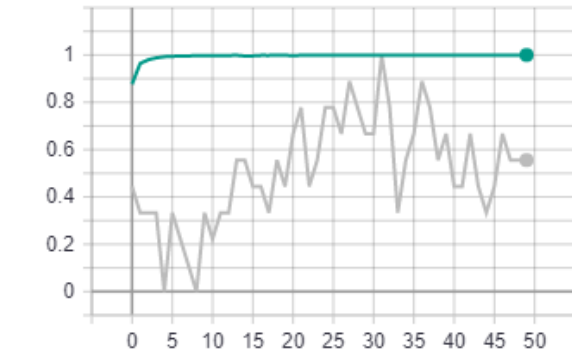
RESULTS

Based on the above results and given that commercial adoption can be ensured if accuracy levels are equal or exceed 90%, we have selected and evaluated the best model (Step II). The model has:

- a. Achieved 100% accuracy on training and validation with only 8 real images
- b. Completed training in 20 seconds per epoch and provided the best results (accuracy) when applied to 1 real image that the model has never seen before.

While transfer learning is revered about in the industry, it adds nothing to our solution and is, in fact, worse than our original, simpler solution – the CNN. Transfer Learning accuracies have also been lagging below 80% and its slow training speed (even with fixed weights) convinced us to stay with the simpler and faster CNN alternative.

Top 3 Experiments by Results, Visualized

Steps	Graphs and details of the specific run
BB	<p>epoch_accuracy</p>  <p>Validation = blue Accuracy = 42% Batch size = 256 Res = 16 Average time per epoch = 71 sec Noise = 20%</p>
II	<p>epoch_accuracy</p>  <p>Validation = grey At 31 epochs, Accuracy = 100%, test1acc=100% batch=256 res=64 noise=0 Average time per epoch = 19 sec</p>

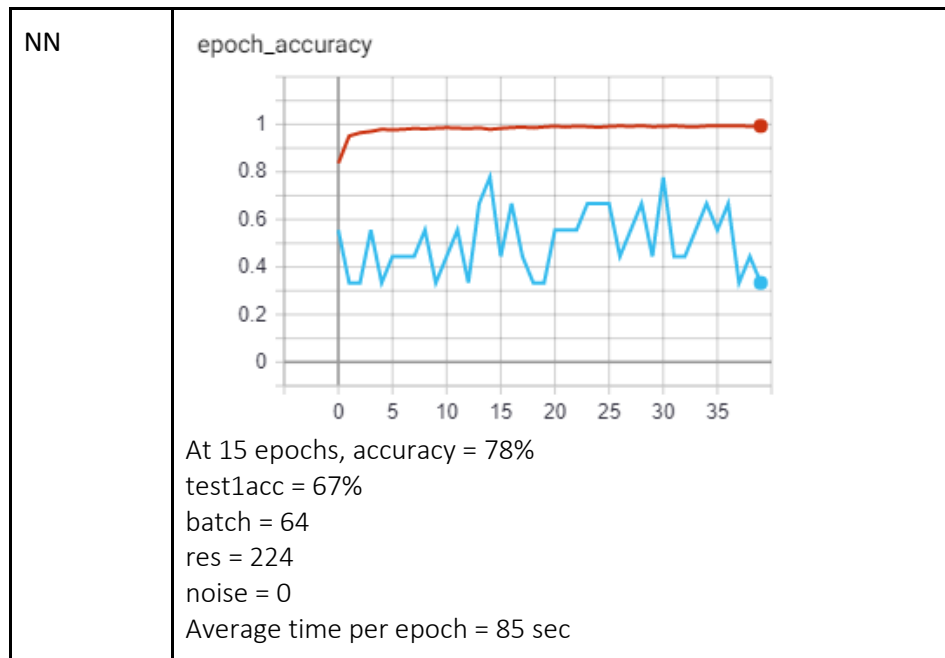


Exhibit #13 – Top 3 Experiments by Results

Therefore, we have completed this Proof of Concept model and have established that to gain commercial feasibility the model needs to meet all requirements as follows:

1. Achieve prediction accuracy at or above 90%
2. Require only a few real images to augment the autogenerated images
3. Complete training in a reasonable time (20 seconds or less).
4. Keep the model effective by including real images in small batches.

EXTENSION TO PRODUCTION

Extension to production should be approached from the below three perspectives:

1. Business viability
2. General deployment concerns
3. Model-specific concerns

Business viability

To extend our model to commercial production and ensure adoption we need to be able to operationalize it to any product our client(s) carries and make it cost effective for our clients (the industry) and their consumers to adopt.

To accomplish that we need to achieve the following:

4. 3D model like the one we used (Studio) is available to ingest images of widgets and to autogenerate a dataset for model training.

5. Clients (The Home Depot, and Lowe's Companies in our case) manually take images of widgets to augment training set with real pictures. Less than 0.2% of the overall training set is required for good performance in commercial settings.
6. Clients can take pictures of the parts they are looking to purchase in store settings (a designated area is provided on store premises) as well as at home over time as the model is trained and improved.
7. Replacing human store assistants with an automated image classifier and a search engine proves cost effective and benefits all parties, i.e. saving of time, money and an improved customer experience are key to successful adoption.

We recommended to automate taking of the real images of widgets and parts as they come off the production belt on manufacturing facilities. This will speed up generation of the training set. To ensure optimal performance, the environment in which these images are taken should be aligned with the environment for image-taking available in-store. As the model is trained and improved over time, the wider roll-out to end consumers can follow where consumers will start taking images of the widgets remotely and upload into the search/classifier engine. This will allow the industry to better manage demand and to improve its profitability.

General Deployment Concerns

In implementing AI models, practitioners use the term “operationalize” instead of a commonly used “deploy”¹. A traditional application development project has the same build, test, deploy and manage. AI projects are different because they have only two distinct phases: “training” and “inference”.

AI's **training** phase involves the selection of algorithms. It also includes the identification and selection of appropriate, clean, well-labeled data. The hyperparameter tuning is usually required to finalize a model. The model will then be validated and tested to ensure that it can adequately generalize without either overfitting of training data or underfitting for generalization.

The **inference** phase of an AI project is focused on how to apply the model to a specific use case. This phase includes ongoing evaluation of the model to determine if the system correctly responds to real-world data. This phase is based on a principle of Continuous Improvement and Continuous Development (CI/CD), where fine-tuning of the model, creation of new training dataset and further hyperparameter configurations improve the model iteratively. The inference phase also includes exploring additional use cases for the model, often going beyond the original scope.

¹<https://www.itproportal.com/features/overcoming-the-challenges-of-machine-learning-model-deployment/>

In AI world, there is no “single platform” because there are many diverse applications that can make inferences including classification, predictions, etc. Our Deep Learning model can be deployed to a cloud-based solution or to an organization's private enterprise server, or to users’ desktop applications, autonomous vehicles, or to other distributed applications. The business needs to understand security risks, deployment costs and available human resources and expertise to deploy the model, and to connect so as to provide superb customer experience and to drive future adoption of the tool.

The real-world operationalization of AI is complex. A data scientist’s model might not be “deployed” to a specific system, rather it needs to be “operationalized” in various systems, different interfaces, and deployment options, as the business requires. Therefore, the “operationalization” of models is the same regardless of the specific deployment methods.

Model-specific concerns

One of the constraints that will need to be managed is the **computing power** necessary to support satisfactory response time. Our algorithm was run with a GPU, and we know that deep learning algorithms require more computing power to create a satisfactory model as these algorithms are usually deployed to process big data. Extending our model to production requires ability to handle thousands of classes instead of just a few as in our model. Doing that can only be supported by sufficient levels of GPU.

Once the initial model is created, operational compute requirements will decrease. Therefore, careful planning is required to provide the right computing power to operationalize the program at each stage of its development and ongoing maintenance.

Ongoing evaluation and betterment of the model will be required to ensure optimal performance given any new data available and to ensure the model correctly responds to real-world data. This phase will require continuous fine-tuning of model hyperparameters, creation of new training set, and further hyperparameter configurations to iteratively improve the model. The inference phase also includes exploring additional use cases for the model, going beyond the original scope. The inference phase takes the training phase, out of the laboratory and into the “real world.”

The real images required to augment the autogenerated training set must have the **same image attributes**, i.e. monochromatic background with similar color temperature, high image resolution, and minimum shadows or highlights. To achieve this in a manufacturing line or over-the-counter application:

1. Each picture needs to be taken in an enclosed environment with no possibility of outside light pollution
2. Within the environment, an HD camera is fixed in position with a ring flashlight around it providing adequate lighting such that shadows and highlights are minimal

3. The flooring of the enclosed box should be uniquely colored (black, white, green) which contrasts well with the color of the object trying to train on

4. The flooring of the environment should have a printed marker small enough to be fully covered by the object of interest. This will ensure the object is always located at the exact center of the shot when batch cropping is being processed at a later stage.

In conclusion, our model is a prototype that, we believe, passed technical feasibility tests and showed good performance results to justify expanding it to a more in-depth commercial feasibility study and a formal business case with proper business assumptions, opportunities and risk level.

APPENDIX

Image # 1 – Case 1 Metadata

Name	Type	Description
reportingperiodname	string	Reporting period name. Required if not using startdate and enddate.
startdate	object	Opening balance date. Required if not using reportingperiodname.
enddate	object	Closing balance date. Required if not using reportingperiodname.
glaccountno	string	GL account number. Required if not using accountgroupname or startaccountno and endaccountno.
accountgroupname	string	Account group name. Required if not using glaccountno or startaccountno and endaccountno.
startaccountno	string	Starting GL account number. Required if not using glaccountno or accountgroupname.
endaccountno	string	Ending GL account number. Required if not using glaccountno or accountgroupname.
reportingbook	string	Reporting book ID. Reporting book ID. Use ACCRUAL or CASH depending on the company configuration. If Global Consolidations is enabled, you can provide a consolidation book ID instead. (Default: ACCRUAL)
adjbooks	adjbook[0...n]	Adjustment book ID's for journals that are enabled in the GL configuration. Use GAAP, TAX, and/or the IDs of user defined books. Do not append text to the ID's. If you need help, look at the Account Balances page in the UI
includereportingbook	boolean	Combine reporting book with other adjustment books. Use true to include the reporting book entries with entries from the specified adjustment books, or false to return only entries for the specified adjustment books.
statistical	string	Statistical accounts. Use either include, exclude, or only. (Default: include)
departmentid	string	Department ID. (If you pass an empty element, no filtering is performed and all department data is listed.)
dept_subs	boolean	Include department sub dimensions. (Default: true)

Name	Type	Description
locationid	string	Location ID. If you pass an empty element, no filtering is performed and all location data is listed. This field is required in a multi-base currency company.
loc_subs	boolean	Include location sub dimensions. (Default: true)
projectid	string	Project ID or project group ID.
projecttypeid	string	Project type. Do not use if project ID is set.
taskid	string	Task ID. Only available when the parent projectid is also specified.
taskid_subs	boolean	Include task sub dimensions. (Default: true)
customerid	string	Customer ID or customer group ID
customerid_subs	boolean	Include customer sub dimensions. (Default: true)
customertypeid	string	Customer type. Do not use if Customer ID is set.
vendorid	string	Vendor ID or vendor group ID
vendortypeid	string	Vendor type. Do not use if Vendor ID is set.
employeeid	string	Employee ID or employee group ID
employeetypeid	string	Employee type. Do not use if Employee ID is set.
itemid	string	Item ID or item group ID
productlineid	string	Product line. Do not use if Item ID is set.

Name	Type	Description
classid	string	Class ID or class group ID
contractid	string	Contract ID or contract group ID
warehouseid	string	Warehouse ID or warehouse group ID

Image #2 – Lowe's Companies Financials

Lowe's Companies, Inc. (LOW)

NYSE - NYSE Delayed Price. Currency in USD

☆ Add to watchlist

66.36 -3.51 (-5.02%)

At close: March 20 4:04PM EDT


Summary Company Outlook  Chart Conversations Statistics Historical Data Profile **Financials** Analysis

Show: **Income Statement** Balance Sheet Cash Flow

Annual

Income Statement

All numbers in thousands

 Get access to 40+ years of historical data with Yahoo Finance Premium. [Learn more](#)

Breakdown	TTM	1/31/2020	1/31/2019	1/31/2018	1/31/2017
Total Revenue	72,148,000	72,148,000	71,309,000	68,619,000	65,017,000
Cost of Revenue	49,205,000	49,205,000	48,401,000	45,210,000	42,553,000
Gross Profit	22,943,000	22,943,000	22,908,000	23,409,000	22,464,000
▼ Operating Expenses					
Selling General and Administrati...	15,367,000	15,367,000	17,413,000	15,376,000	15,129,000
Total Operating Expenses	16,629,000	16,629,000	18,890,000	16,823,000	16,618,000
Operating Income or Loss	6,314,000	6,314,000	4,018,000	6,586,000	5,846,000
Interest Expense	-	-	640,000	630,000	634,000
Total Other Income/Expenses Net	-	-	-	-464,000	-
Income Before Tax	5,623,000	5,623,000	3,394,000	5,489,000	5,201,000
Income Tax Expense	1,342,000	1,342,000	1,080,000	2,042,000	2,108,000
Income from Continuing Operations	4,281,000	4,281,000	2,314,000	3,447,000	3,093,000
Net Income	4,281,000	4,281,000	2,314,000	3,447,000	3,091,000
Net Income available to common s...	4,268,000	4,268,000	2,307,000	3,436,000	3,062,000
EBITDA	-	7,724,000	5,641,000	7,659,000	7,425,000

<https://finance.yahoo.com/quote/LOW/financials/>

Image # 3 – The Home Depot Financials

The Home Depot, Inc. (HD)
NYSE - NYSE Delayed Price. Currency in USD [Add to watchlist](#)

152.15 -8.98 (-5.57%)
At close: March 20 4:00PM EDT

Summary Company Outlook [Chart](#) [Conversations](#) [Statistics](#) [Historical Data](#) [Profile](#) **Financials** [Analysis](#)

Show: **Income Statement** [Balance Sheet](#) [Cash Flow](#) Annual

Income Statement All numbers in thousands

[Get access to 40+ years of historical data with Yahoo Finance Premium. Learn more](#)

Breakdown	TTM	1/31/2020	1/31/2019	1/31/2018	1/31/2017
Total Revenue	110,225,000	110,225,000	108,203,000	100,904,000	94,595,000
Cost of Revenue	72,653,000	72,653,000	71,043,000	66,548,000	62,282,000
Gross Profit	37,572,000	37,572,000	37,160,000	34,356,000	32,313,000
Operating Expenses					
Selling General and Administrati...	19,740,000	19,740,000	19,513,000	17,864,000	17,132,000
Total Operating Expenses	21,729,000	21,729,000	21,383,000	19,675,000	18,886,000
Operating Income or Loss	15,843,000	15,843,000	15,777,000	14,681,000	13,427,000
Interest Expense	1,201,000	1,201,000	1,051,000	1,057,000	972,000
Total Other Income/Expenses Net	-	-	-263,000	-	-
Income Before Tax	14,715,000	14,715,000	14,556,000	13,698,000	12,491,000
Income Tax Expense	3,473,000	3,473,000	3,435,000	5,068,000	4,534,000
Income from Continuing Operations	11,242,000	11,242,000	11,121,000	8,630,000	7,957,000
Net Income	11,242,000	11,242,000	11,121,000	8,630,000	7,957,000
Net Income available to common s...	11,242,000	11,242,000	11,121,000	8,630,000	7,957,000
EBITDA	-	18,212,000	17,759,000	16,817,000	15,436,000

<https://finance.yahoo.com/quote/HD/financials/>

Image #4 – A sample of an autogenerated part for training set of images

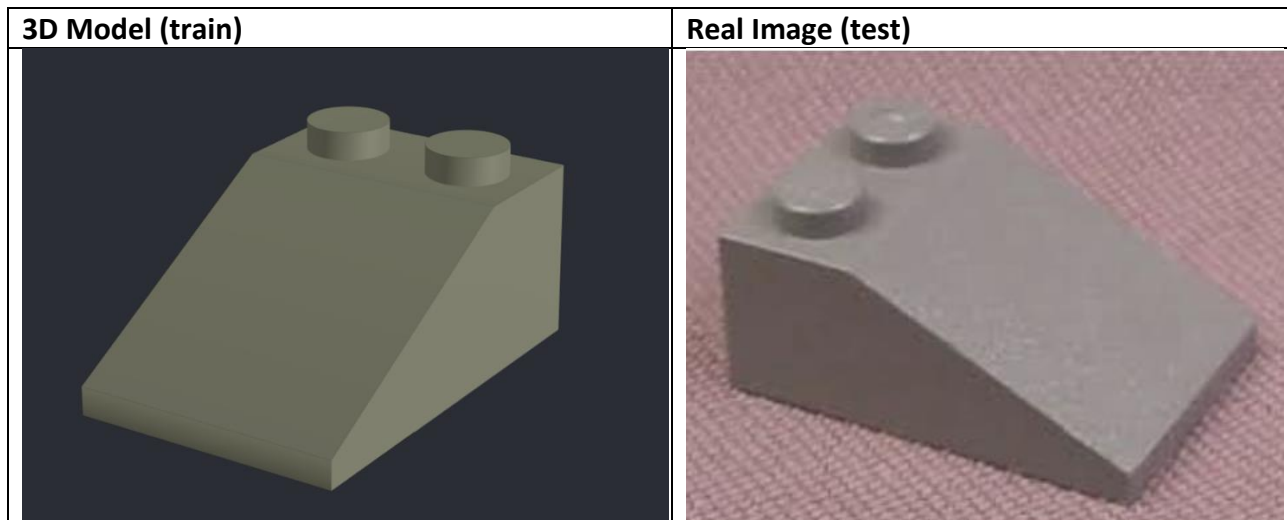







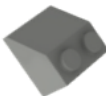







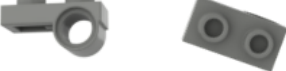


Image # 5 – The Catalog:

Name	Image
970c00	
2420	
2436	
3001	
3003	
3004	
3022	
3039	

3701	
3710	
4032	
4865	
10247	
11215	
11476	
18677	

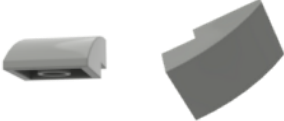

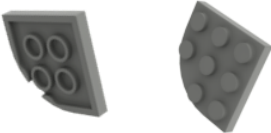
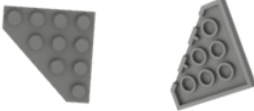
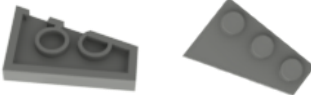
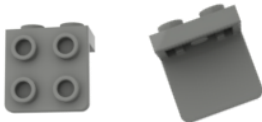
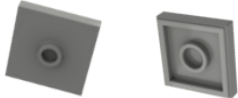

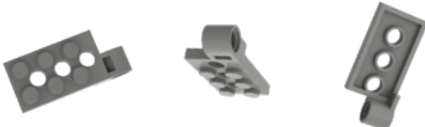
24309	
27925	
30357	
30503	
43722	
44728	
87580	
93273	
98286	

Image # 6 – Deep Learning Research Diary

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	HYPERPARAMETERS																
2	PERFORMANCE RESULTS																
3	Hyperparameters -->	Training Phase (# of images)	Validation Phase (# of images)	Testing Phase 1 (# of images)	Testing Phase 2 (# of images)	Dataset with Real Images	% of real images used in training	Batch Size Pool	Resolution Pool	lr-Layermode/01ayermode	Noise Level %	Best Time (seconds) per Epoch	Best Epoch / Total Epochs	Best Train / Validation Accuracy %	Real Image Dataset	Real Image Accuracy %	Comment
4	AA	31211	3869	3800	12	Test2	0.0%	16.64.256	16.32.64	0.001/128/64	0.00%	84	5/5	100% / 99%	test2	33.00%	everything did well
5	BB	31211	3869	3800	12	Test2	0.0%	16.64.256	16.32.64	0.001/128/64	20.00%	71	5/5	100% / 99%	test2	42.00%	res 64 + batch 64 did poorly
6	CC	31211	3869	3800	12	Test2	0.0%	16.64.256	16.32.64	0.001/128/64	50.00%	57	5/5	100% / 99%	test2	39.00%	res 16 did poorly
7	DD	23247	11735	3898	12	Test2	0.0%	16.64.256	16.32.64	0.001/128/64	0.00%	50	5/5	100% / 99%	test2	33.00%	res 64 did poorly
8	EE	23247	11735	3898	12	Test2	0.0%	16.64.256	16.32.64	0.001/128/64	20.00%	61	5/5	100% / 99%	test2	36.00%	res 16 did poorly
9	FF	23247	11735	3898	12	Test2	0.0%	16.64.256	16.32.64	0.001/128/64	50.00%	128	5/5	100% / 99%	test2	39.00%	res 16 did poorly
10	GG	3898	8	1	4	Val, Test1, Test2	0.0%	16.64.256	16.32.64	0.001/128/64	0.00%	70	5/5	100% / 99%	val	33.00%	everything did poorly
11	HH	3898+8	3	1	1	Train, Val, Test1, Test2	0.2%	16.64.256	16.32.64	0.001/128/64	20.00%	19	40/50	100% / 89%	val	89.00%	res 64 + ~256 batch did well
12	II	3898+8	3	1	1	Train, Val, Test1, Test2	0.2%	16.64.256	16.32.64	0.001/128/64	0.00%	20	32/50	100% / 100%	val	90.00%	batch 64 did well
13	JJ	3898+8	3	1	1	Train, Val, Test1, Test2	0.2%	16.64.256	16.32.64	0.0001512/1024	50.00%	23	41/50	100% / 78%	val	78.00%	batch 64 did well
14	KK	23247+8	3	1	1	Train, Val, Test1, Test2	0.03%	16.64.256	16.32.64	0.001/128/64	0.00%	113	21.40.43/50	100% / 89%	val	89.00%	batch 64 did well
15	LL	23247+4	7	1	1	Train, Val, Test1, Test2	0.017%	16.64.256	16.32.64	0.001/128/64	0.00%	32	24/50	100% / 57%	val	57.00%	res32+batch256 did well
16	MM	23247+4	7	1	1	Train, Val, Test1, Test2	0.017%	64	224	0.001/128/64	0.00%	506	2/50	100% / 66%	val	66.00%	TRANSFER LEARNING
17	NN	3898+8	3	1	1	Train, Val, Test1, Test2	0.2%	64	224	0.001/128/64	0.00%	85	15/50	98% / 62%	val	78.00%	TRANSFER LEARNING
18	NOTES:																
19	all red numbers denote manually taken images																
20	AA-CC applying 8/1/1 split for different combinations of hyperparameters (noises, 3 resolutions, 3 batch sizes)																
21	DD-FF applying 6/3/1 split for different combinations of hyperparameters (noises, 3 resolutions, 3 batch sizes)																
22	GG applying 1/0/0 split and adding 12 real images to the training, validation and testing phases																
23	HH-JJ using only 10% of the autogenerated images and adding 8 real images to the training phase and remaining 4 - to validation and testing phases																
24	... etc.																
25	The total of 12 real images all belong to 1 class.																

Image #7 – Our GitHub Page:

<https://github.com/ea5055/humphrey894>

ea5055 / humphrey894 Private

Watch 1 Star 0 Fork 0

Code Issues 0 Pull requests 0 Actions Projects 0 Security Insights

No description, website, or topics provided.

41 commits 1 branch 0 packages 0 releases

Branch: master New pull request Create new file Upload files Find file Clone or download

ea5055 Delete Image Prep Step 2_temp.py Latest commit 11db803 22 hours ago

.idea

Initial organization

last month

3d_models

Add files via upload

4 days ago

image_prep

Delete Image Prep Step 2_temp.py

22 hours ago

test_area

Testing Feb 28/20

24 days ago

README.md

Update README.md

29 days ago

hyperparameter_tuning.py

Add files via upload

23 hours ago

model_evaluation.py

Add files via upload

23 hours ago

requirements.txt

Initial test #2

29 days ago

train_val_test1_split.xlsx

Testing Feb 26/20

25 days ago

transfer_learning.py

Add files via upload

23 hours ago

README.md

MMAI2020 - 894 Deep Learning | Final Project

Team Humphrey

This is the repository for all code and supporting files (except data) related to this project.

The structure is:

1. initial_prep - this folder contains all files to process the 3D generated data into the final usable set for the algorithms. It has folders train/val/test1, and each contain 25 subfolders for each different class. The name of each subfolder is the actual ID of the Lego piece

For a quick visual map of all 25 pieces:
https://docs.google.com/document/d/1gl0Q59Egq01TREjsjBxvVd_6P2C9N8170BRVjY6lLDzY/edit