Multi View Convolutional Neural Networks Algorithm

MAYAN MENAHEM • RON MALKA • YISHAY NADAV

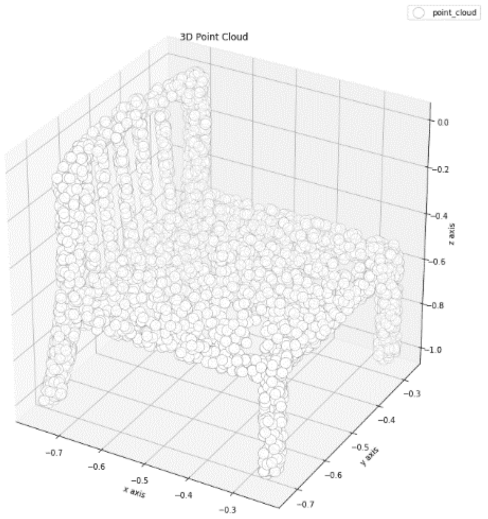
This project presents an implementation of the article “Multi-view Convolutional Neural Networks for 3D Shape Recognition”. We implemented this article using the Keras library with Tensorflow.
This report will cover the steps of implementation, from input processing to building the convolutional network model, and finally training the model on the given dataset.

Ultimately, we will present our own initiatives for improving the accuracy of the network and optimizing the time consumption.
The complete project code can be found in this GitHub repository:
https://github.com/yishayna/Multi-view-CNN-using-Keras

Chapter 1: Processing the input

Throughout the project we use the NumPy library for array manipulation. All of the transformations described below were performed without using loops, instead we made use of the optimizations done by NumPy.

The dataset includes a set of 3991 samples, each of the samples represents an object that corresponds with one of 10 classes (label types). These shapes are represented by a point cloud, which is a list of 4000 (X, Y, Z) 3D points.

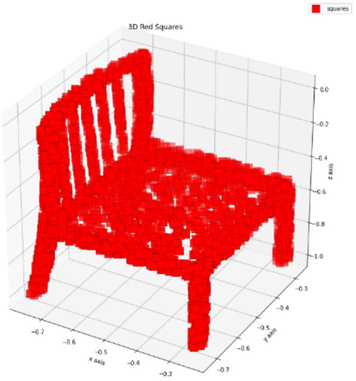


3D point cloud example

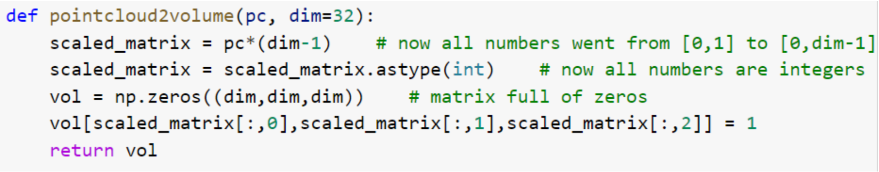
Step 1

Transform the point cloud into a volume – a 3-dimensional cube in which we place each 3D point in its place inside the cube (by setting the relevant cell to "1").

Since the sample points are within the unit cube (each coordinate is ranged between 0-1), we first multiply the entire array by the dimension (in our case, 32). Then, in order to avoid a 3-level nested for loop, we use NumPy special indexing in order to place all points inside the volume in a single line of code.



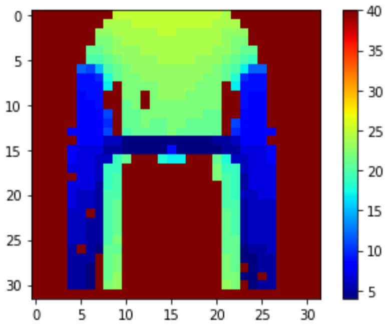
Volume example



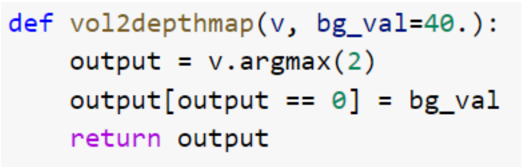
```
def pointcloud2volume(pc, dim=32):
    scaled_matrix = pc*(dim-1)     # now all numbers went from [0,1] to [0,dim-1]
    scaled_matrix = scaled_matrix.astype(int)     # now all numbers are integers
    vol = np.zeros((dim,dim,dim))     # matrix full of zeros
    vol[scaled_matrix[:,0],scaled_matrix[:,1],scaled_matrix[:,2]] = 1
    return vol
```

Step 2

Transform the volume into a depth map. This transformation will change every 3D cube into a 2D representation of the cube, where the 3rd dimension is represented by a depth value. This way of representing the sample is equivalent to a camera looking at the object from a certain angle.



Depth map example



```
def vol2depthmap(v, bg_val=40.):
    output = v.argmax(2)
    output[output == 0] = bg_val
    return output
```

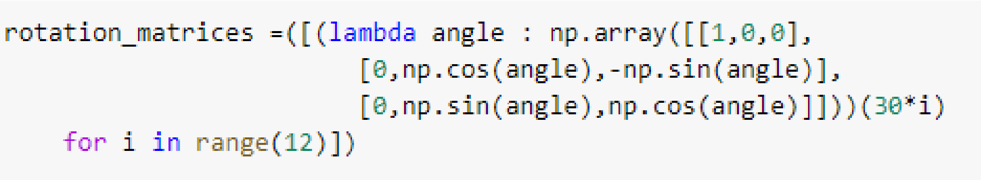
In order to make this transformation without using loops we used the NumPy function called argmax(axis). The function is applied on each sub-array along the specified axis of the array. It returns the index of the maximum element in the array. Since each element in the arrays is either "1" or "0", it will return the index of the first "1" in the array, or "0" if there are no ones in the array.
We apply these two transformations on our input data before feeding it to the CNN.

Step 3

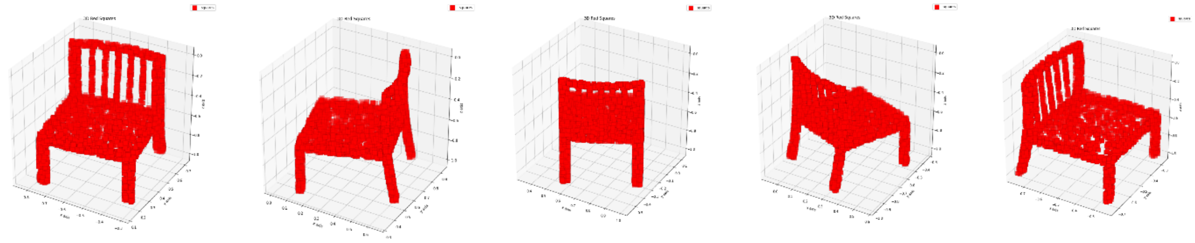
In order to increase the accuracy of our model we implement the multi-view approach.

For each sample, instead of applying the transformations on it just once, we perform 12 rotations of the object by using a 3d rotation matrix which rotates the object 30^0 around the z-axis. After each rotation we transform the point cloud into a depth map as described in parts 1 and 2, this way each sample is represented from 12 different angles.

To prevent mutation in our code we first create a list of 12 rotation matrices by incrementing the angle by 30^0 each iteration using Generator technique:



```
rotation_matrices =([(lambda angle : np.array([[1,0,0],
                              [0,np.cos(angle),-np.sin(angle)],
                              [0,np.sin(angle),np.cos(angle)]]))(30*i)
   for i in range(12)])
```

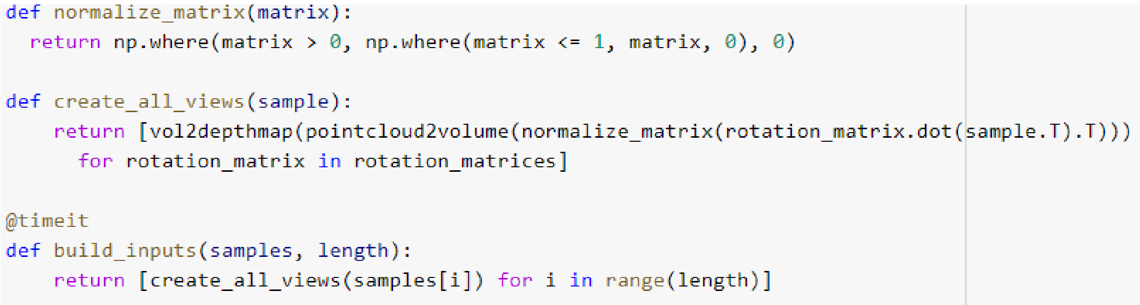


Example of object rotation

All we need to do now is put everything together. We take each sample from the training set and the test set and for each sample we do the following:

1. Multiply the sample by a rotation matrix
2. Normalize the values in the sample between 0 and 1
3. Apply the functions "pointcloud2volume" and "vol2depthmap"
4. Do this for every rotation matrix in the list

The final output is a list that contains 3991*12 depth maps of size 32*32.



```
def normalize_matrix(matrix):
  return np.where(matrix > 0, np.where(matrix <= 1, matrix, 0), 0)

def create_all_views(sample):
    return [vol2depthmap(pointcloud2volume(normalize_matrix(rotation_matrix.dot(sample.T).T)))
      for rotation_matrix in rotation_matrices]

@timeit
def build_inputs(samples, length):
    return [create_all_views(samples[i]) for i in range(length)]
```

In all the building processes of the input, we adopt functional programming techniques to speed up our performance and supply clean and readable code.

Chapter 2: Building the model

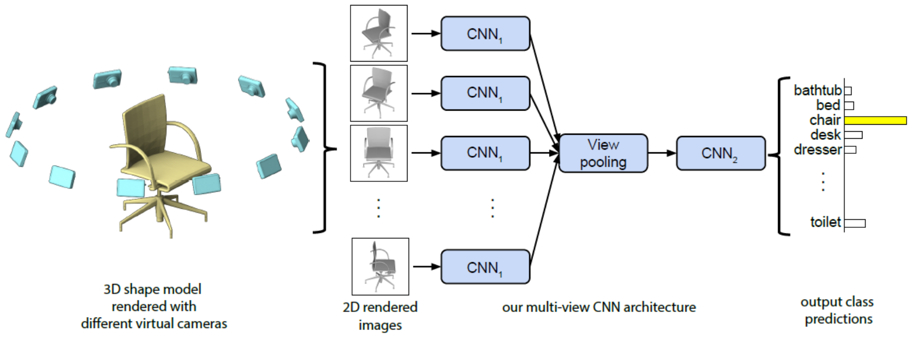
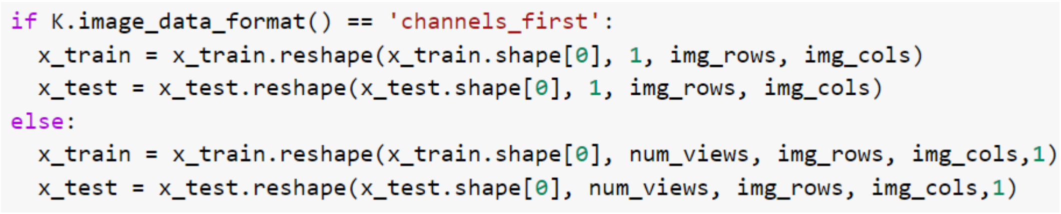


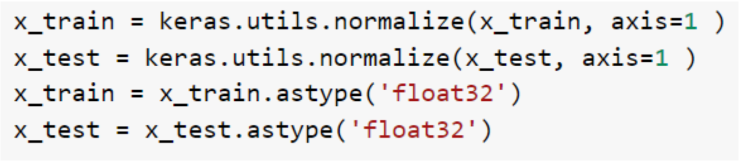
Illustration taken from the article

After processing the input and creating 12 views for each sample, our input is a list that holds all the samples, each sample from 12 distinct views.
Before feeding the depth maps to the model we need to perform a few manipulations on our input to make it compatible with the Sequential model.



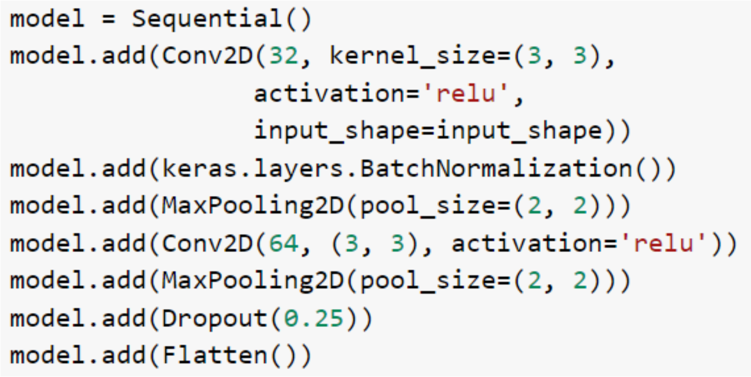
```
if K.image_data_format() == 'channels_first':
  x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
  x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
else:
  x_train = x_train.reshape(x_train.shape[0], num_views, img_rows, img_cols,1)
  x_test = x_test.reshape(x_test.shape[0], num_views, img_rows, img_cols,1)
```

The first thing we need to do is reshape the train and test lists into 4-dimensional lists of shape (3991, 12, 32, 32), where 3991 is the number of samples and 12 is the number of views. Each of the depth maps is of shape 32*32. The second step will be to normalize the arrays and convert them from type integer to type float.

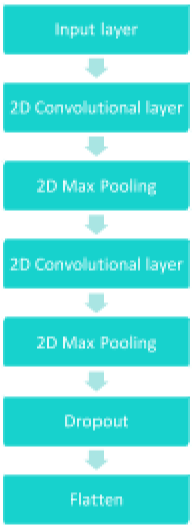


```
x_train = keras.utils.normalize(x_train, axis=1 )
x_test = keras.utils.normalize(x_test, axis=1 )
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

After preparing the input lists for the model we define the model itself.



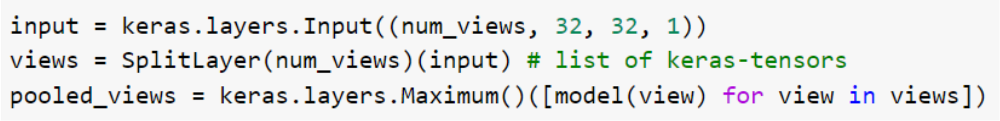
```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(keras.layers.BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
```



The model is composed of two CNNs – first we define the first CNN, it is composed of 6 layers –

- Conv2D – a 2D convolution layer that extracts features from a source image. It can help the machine to learn specific characteristics of an image.
- Batch normalization - normalizes the activation of the previous layer.
- MaxPooling2D – takes the maximum of each pool size across the array. Here pool size is set to be a window of size 2*2. This layer reduces the image dimensionality without losing important features or patterns.
- Dropout – prevents the model from over fitting on the data.
- Flatten – Flattening transforms a two-dimensional matrix of features into a vector that can be fed into a fully connected neural network classifier.

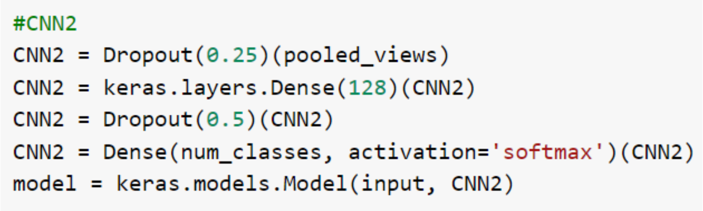
We then take the input and split it into 12 views, where each view has a list of all the training samples from this specific view angle.



```
input = keras.layers.Input((num_views, 32, 32, 1))
views = SplitLayer(num_views)(input) # list of keras-tensors
pooled_views = keras.layers.Maximum()([model(view) for view in views])
```

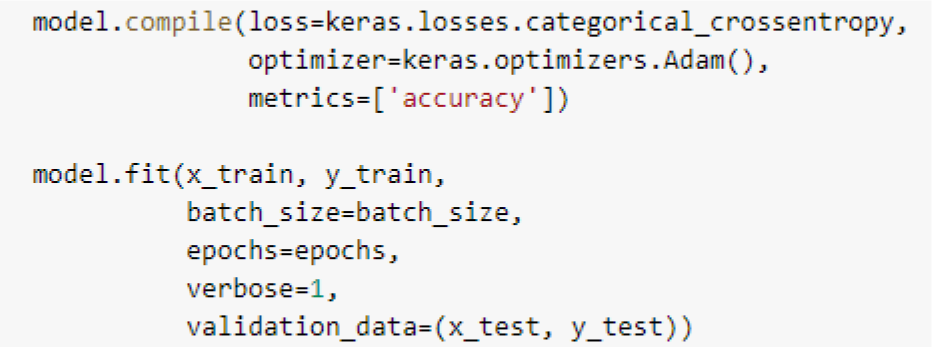
After this split we get 12 CNNs that train on each of the 12 views. SplitLayer inherits from keras.layers.Layer and implements the "call" method, which represents the logic behind the layer and is called by the model during training. The logic of the SplitLayer is dividing the input layer into 12 layers.

After the training is over, we implement the View Pooling (see article) by taking the maximum from each output layer.



```
#CNN2
CNN2 = Dropout(0.25)(pooled_views)
CNN2 = keras.layers.Dense(128)(CNN2)
CNN2 = Dropout(0.5)(CNN2)
CNN2 = Dense(num_classes, activation='softmax')(CNN2)
model = keras.models.Model(input, CNN2)
```

In the last step of the model we train the second CNN by adding two Dense layers in which the results of the convolutional layers are fed through neural layers to generate a prediction, and two Dropout layers in order to prevent overfitting on our data.

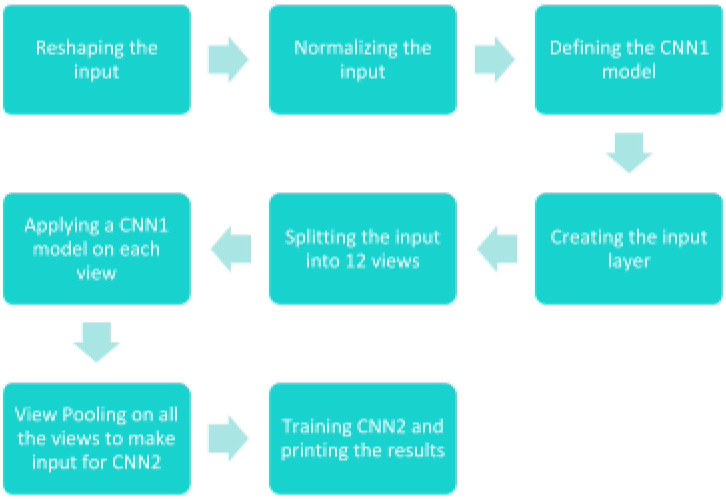


```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
```

We now compile the model and start the training. Our initial input is a list that holds all the samples from all the views together. It will be split later in the process to separate views like we described before.
Notice we are using an optimizer called Adam to improve the runtime of our training. Also, we use categorical_crossentropy in our model which increases the accuracy results by large scale. Why we chose this loss function will be explained in more detail in the upcoming sections.

We can summarize this part with this scheme of the model flow from top to bottom:


Reshaping the input
Normalizing the input
Defining the CNN1 model
Creating the input layer
Splitting the input into 12 views
Applying a CNN1 model on each view
View Pooling on all the views to make input for CNN2
Training CNN2 and printing the results

Chapter 3: Training Results and optimizations

Part 1 : Training Results

Our model shows qualified results. We can see that the model accuracy advance between epochs is monotonically increasing and converges up to 1. Simultaneously the loss rate decreases down to 0.
In 12 epochs, the highest accuracy rate we got is 0.928 and the lowest loss was 0.218.

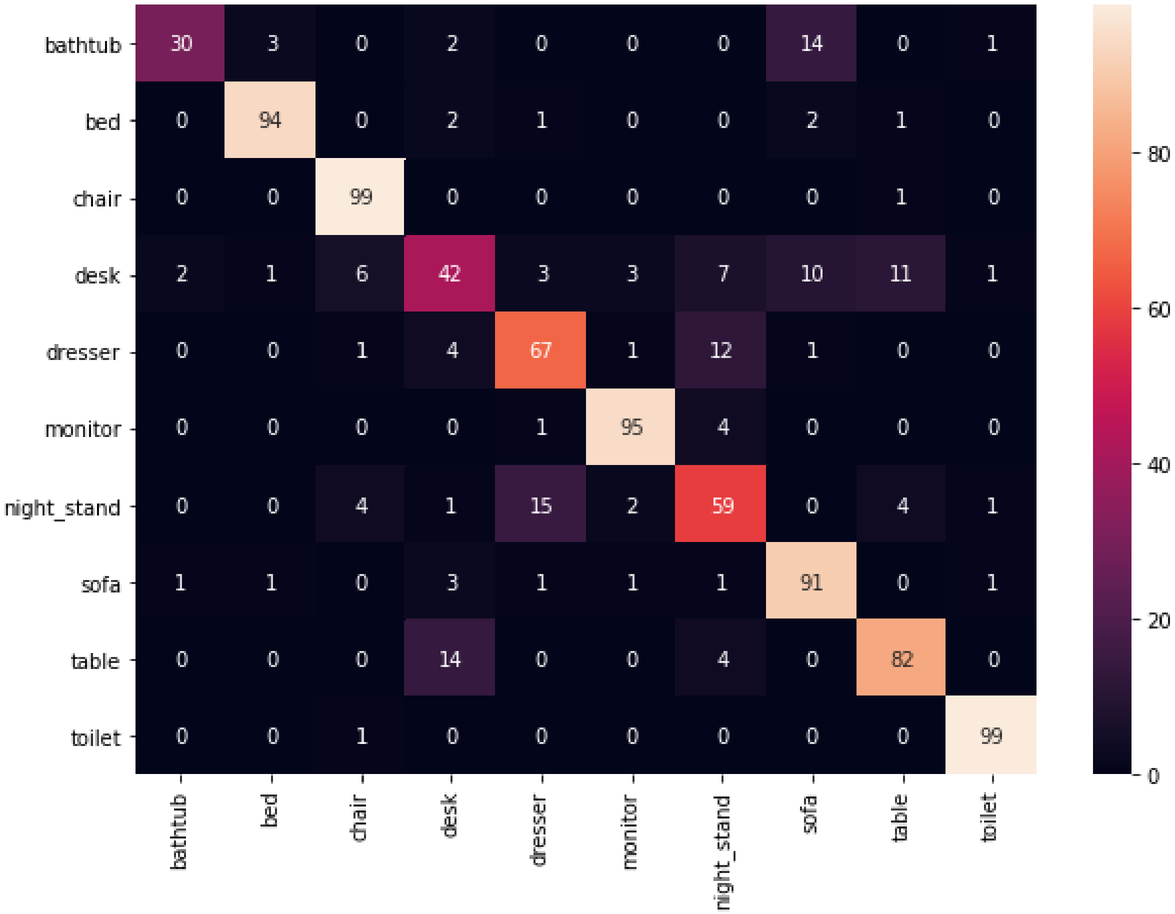
The more we increase the number of epochs, the network accuracy rate converges monotonically to 1, and the loss rate to 0.

The fact that the results of the last epoch in training on the input data, and the result of the test dataset are in related values, points to that the manipulations on the input data do not cause model overfitting. These positive outcomes are mainly due to the variance of the data and the different views.

```
Epoch 12/12
32/32 [==============================] - 3s 85ms/step - loss: 0.2184 -
accuracy: 0.9288 - val_loss: 0.5386 - val_accuracy: 0.8348
Test loss: 0.538562536239624
Test accuracy: 0.8348017334938049
```

Another measure of the quality of the classification process is the confusion matrix that we can see below. By definition of the confusion matrix, the rows indicate the true number of the objects and the columns the number of identification of the objects by the network, note that the number is represented as a percentage. As we can see in our matrix the objects were classified correctly most of the time, which means that our model succeeded in classifying the objects as we expected.

Confusion Matrix



	bathtub	bed	chair	desk	dresser	monitor	night_stand	sofa	table	toilet
bathtub	30	3	0	2	0	0	0	14	0	1
bed	0	94	0	2	1	0	0	2	1	0
chair	0	0	99	0	0	0	0	0	1	0
desk	2	1	6	42	3	3	7	10	11	1
dresser	0	0	1	4	67	1	12	1	0	0
monitor	0	0	0	0	1	95	4	0	0	0
night_stand	0	0	4	1	15	2	59	0	4	1
sofa	1	1	0	3	1	1	1	91	0	1
table	0	0	0	14	0	0	4	0	82	0
toilet	0	0	1	0	0	0	0	0	0	99

As part of the model training, we print the training results in the following manner:

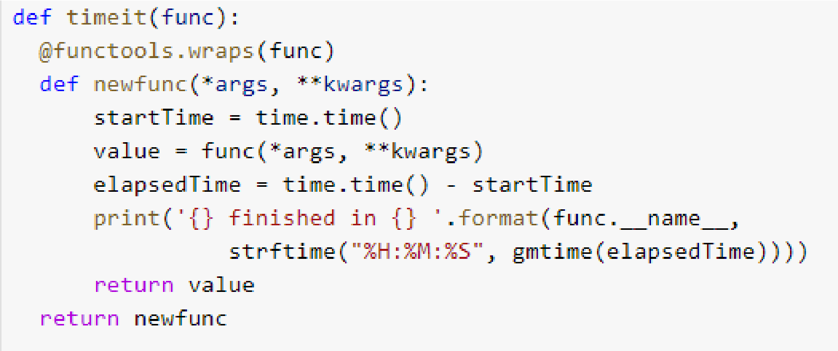
1. We print the shape of the samples of the dataset and test dataset.
2. All the epochs of the model are printed through the process.
3. After the model finished we printed the two values Test loss, Test accuracy which returned as the output of model.evaluate.
4. We print the confusion matrix and present it as an image.
5. We print the total time of the training process using the decorator technique.

Train Results

```
samples_train shape: (3991, 4000, 3), labels_train shape: (3991,)
samples_train shape: (908, 4000, 3), labels_train shape: (908,)
build_inputs finished in 00:00:11
build_inputs finished in 00:00:02
Train
(3991, 12, 32, 32)
x_train shape: (3991, 12, 32, 32, 1)
3991 train samples
908 test samples
Epoch 1/12
32/32 [==============================] - 3s 103ms/step - loss: 1.9787 - accuracy: 0.4951 -
val_loss: 1.8563 - val_accuracy: 0.6597
Epoch 2/12
32/32 [==============================] - 3s 86ms/step - loss: 0.7949 - accuracy: 0.7507 -
val_loss: 0.8529 - val_accuracy: 0.7324
Epoch 3/12
32/32 [==============================] - 3s 86ms/step - loss: 0.6006 - accuracy: 0.8036 -
val_loss: 0.7208 - val_accuracy: 0.7797
Epoch 4/12
32/32 [==============================] - 3s 86ms/step - loss: 0.4963 - accuracy: 0.8389 -
val_loss: 0.7895 - val_accuracy: 0.7412
Epoch 5/12
32/32 [==============================] - 3s 86ms/step - loss: 0.4421 - accuracy: 0.8562 -
val_loss: 0.6128 - val_accuracy: 0.8073
Epoch 6/12
32/32 [==============================] - 3s 86ms/step - loss: 0.3808 - accuracy: 0.8785 -
val_loss: 0.6529 - val_accuracy: 0.7941
Epoch 7/12
32/32 [==============================] - 3s 87ms/step - loss: 0.3391 - accuracy: 0.8877 -
val_loss: 0.6053 - val_accuracy: 0.7863
Epoch 8/12
32/32 [==============================] - 3s 85ms/step - loss: 0.3021 - accuracy: 0.8970 -
val_loss: 0.5794 - val_accuracy: 0.8227
Epoch 9/12
32/32 [==============================] - 3s 86ms/step - loss: 0.2747 - accuracy: 0.9020 -
val_loss: 0.5459 - val_accuracy: 0.8381
Epoch 10/12
32/32 [==============================] - 3s 86ms/step - loss: 0.2553 - accuracy: 0.9088 -
val_loss: 0.5506 - val_accuracy: 0.8293
Epoch 11/12
32/32 [==============================] - 3s 85ms/step - loss: 0.2325 - accuracy: 0.9256 -
val_loss: 0.6009 - val_accuracy: 0.8194
Epoch 12/12
32/32 [==============================] - 3s 85ms/step - loss: 0.2184 - accuracy: 0.9288 -
val_loss: 0.5386 - val_accuracy: 0.8348
Test loss: 0.538562536239624
Test accuracy: 0.8348017334938049
[[30  3  0  2  0  0  0 14  0  1]
 [ 0 94  0  2  1  0  0  2  1  0]
 [ 0  0 99  0  0  0  0  0  1  0]
 [ 2  1  6 42  3  3  7 10 11  1]
 [ 0  0  1  4 67  1 12  1  0  0]
 [ 0  0  0  0  1 95  4  0  0  0]
 [ 0  0  4  1 15  2 59  0  4  1]
 [ 1  1  0  3  1  1  1 91  0  1]
 [ 0  0  0 14  0  0  4  0 82  0]
 [ 0  0  1  0  0  0  0  0  0 99]]
train_multi finished in 00:00:39
```

Part 2 : Optimizations

Measuring the runtime using decorator:



```
def timeit(func):
  @functools.wraps(func)
  def newfunc(*args, **kwargs):
      startTime = time.time()
      value = func(*args, **kwargs)
      elapsedTime = time.time() - startTime
      print('{} finished in {} '.format(func.__name__,
                strftime("%H:%M:%S", gmtime(elapsedTime))))
      return value
  return newfunc
```

We used a decorator named “timeit” to measure the runtime of the input processing and the model training. We observed an improvement in the input processing stage, that derives from the functional programming we applied and the fact that the entire process of applying the transformations avoids mutations.
We also observed improvements in the training stage runtime for various reasons such as: using keras libraries for normalization, avoiding adding unnecessary layers to the model, and putting things in the right order.

Adadelta vs Adam optimizer:

We first tried using Adadelta optimizer. In Adadelta you don't require an initial learning rate constant to start with. We got high accuracy results but the network didn’t quite converge like we wanted to. We then tried using Adam optimizer. Adam combines the good properties of Adadelta and RMSprop and hence tends to do better for most of the problems. With Adam optimizer we obtained better results although the runtime was increased slightly.

categorical_crossentropy vs binary_crossentropy loss function:

We started with the “categorical_crossenthropy” loss function which is more suited for models in which each object is compatible with one of multiple classes, for example in our project an object can be either a chair, or a table, or a bed, and so on. The model guesses which of the 10 classes an object represents for each sample in the dataset, based on the learning process and a score between 0 to 1 given to each sample for each of the labels.

In order to improve the accuracy we tried changing the loss function to “binary_crossentropy”. This loss function produced better results, but we were confused since this function performs well in models that require binary classification of objects (1 or 0, yes or no), for example, given an object the model will decide if the object is a chair or not, a table or not, and so on. This type of model is different from ours and we tried to understand why this happened. Finally, we figured out that the cause for the increase in accuracy was that for each sample, instead of having one output with a single guess of 1 of 10 possible labels, we received an output of a vector of size 10, in which each label received a score of either 0 or 1 - and so in our model a bathtub for example, which is the first label, will receive an output of [1,0,0,0,0,0,0,0,0,0], and so the model will be “right” 10 times instead of one time, since it not only guessed that the object is a bathtub, but it also guesses that it is not a table, not a toilet, not a bed, and so on.

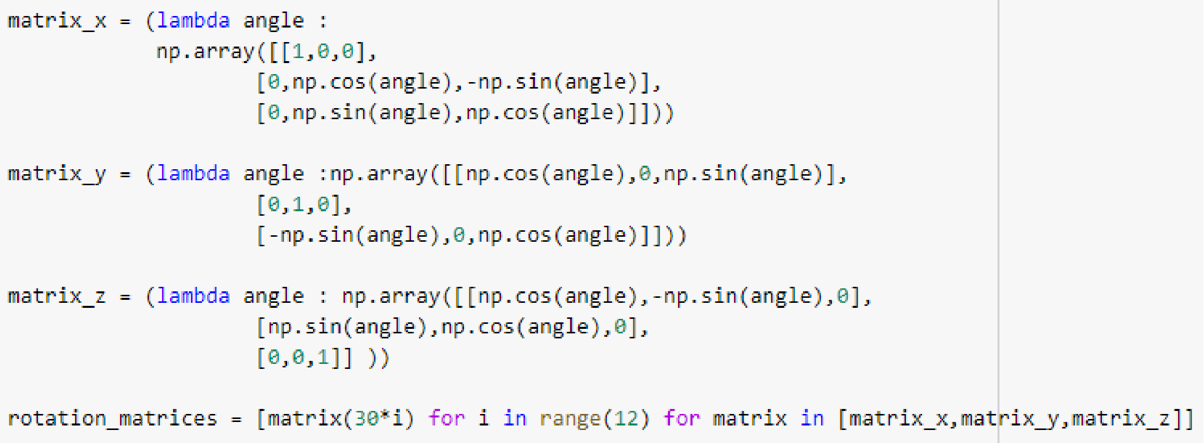
After we understood the reason for this anomaly we changed the loss function back to “categorical_crossenthropy”.

Chapter 4: Being Creative

Our creative initiative for the model was adding 2 more rotation matrices that rotate each object around the other 2 axises, so we get a rotation around all 3 axises. This simulates a camera looking at the objects from all possible angles in a spherical manner.

This change led to an increase in the accuracy and a decrease in the loss, and the confusion matrix showed better results. Still, the improvements were not significant, probably due to the fact that most of the features of the objects are less noticeable when looking from the top and bottom sides. In addition, the time consumption was increased 3 times. In our opinion the mild increase in accuracy was not worth the increase in time consumption.

The code and the results of this part are detailed below.



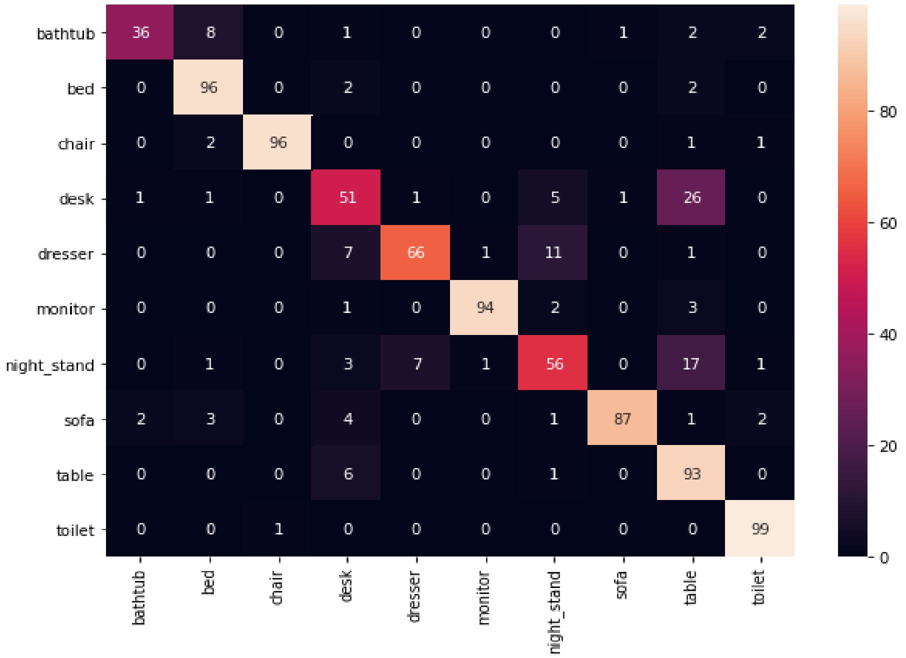
```
matrix_x = (lambda angle :
            np.array([[1,0,0],
                    [0,np.cos(angle),-np.sin(angle)],
                    [0,np.sin(angle),np.cos(angle)]]))

matrix_y = (lambda angle :np.array([[np.cos(angle),0,np.sin(angle)],
                    [0,1,0],
                    [-np.sin(angle),0,np.cos(angle)]]))

matrix_z = (lambda angle : np.array([[np.cos(angle),-np.sin(angle),0],
                    [np.sin(angle),np.cos(angle),0],
                    [0,0,1]] ))

rotation_matrices = [matrix(30*i) for i in range(12) for matrix in [matrix_x,matrix_y,matrix_z]]
```

Confusion Matrix



	bathtub	bed	chair	desk	dresser	monitor	night_stand	sofa	table	toilet
bathtub	36	8	0	1	0	0	0	1	2	2
bed	0	96	0	2	0	0	0	0	2	0
chair	0	2	96	0	0	0	0	0	1	1
desk	1	1	0	51	1	0	5	1	26	0
dresser	0	0	0	7	66	1	11	0	1	0
monitor	0	0	0	1	0	94	2	0	3	0
night_stand	0	1	0	3	7	1	56	0	17	1
sofa	2	3	0	4	0	0	1	87	1	2
table	0	0	0	6	0	0	1	0	93	0
toilet	0	0	1	0	0	0	0	0	0	99

Train Results

```
samples_train shape: (3991, 4000, 3), labels_train shape: (3991,)
samples_train shape: (908, 4000, 3), labels_train shape: (908,)
Train
(3991, 36, 32, 32)
x_train shape: (3991, 36, 32, 32, 1)
3991 train samples
908 test samples
Epoch 1/12
32/32 [==============================] - 5s 157ms/step - loss: 2.4446 - accuracy: 0.3603 -
val_loss: 1.4554 - val_accuracy: 0.5529
Epoch 2/12
32/32 [==============================] - 4s 129ms/step - loss: 0.9176 - accuracy: 0.7196 -
val_loss: 0.8796 - val_accuracy: 0.7203
Epoch 3/12
32/32 [==============================] - 4s 130ms/step - loss: 0.6321 - accuracy: 0.7913 -
val_loss: 0.7050 - val_accuracy: 0.7731
Epoch 4/12
32/32 [==============================] - 4s 131ms/step - loss: 0.5117 - accuracy: 0.8291 -
val_loss: 0.6208 - val_accuracy: 0.7996
Epoch 5/12
32/32 [==============================] - 4s 131ms/step - loss: 0.4542 - accuracy: 0.8512 -
val_loss: 0.6007 - val_accuracy: 0.8051
Epoch 6/12
32/32 [==============================] - 4s 131ms/step - loss: 0.3965 - accuracy: 0.8685 -
val_loss: 0.5324 - val_accuracy: 0.8216
Epoch 7/12
32/32 [==============================] - 4s 130ms/step - loss: 0.3554 - accuracy: 0.8815 -
val_loss: 0.5968 - val_accuracy: 0.8106
Epoch 8/12
32/32 [==============================] - 4s 129ms/step - loss: 0.3123 - accuracy: 0.8993 -
val_loss: 0.5582 - val_accuracy: 0.8139
Epoch 9/12
32/32 [==============================] - 4s 128ms/step - loss: 0.2912 - accuracy: 0.9025 -
val_loss: 0.4898 - val_accuracy: 0.8381
Epoch 10/12
32/32 [==============================] - 4s 129ms/step - loss: 0.2593 - accuracy: 0.9163 -
val_loss: 0.4848 - val_accuracy: 0.8425
Epoch 11/12
32/32 [==============================] - 4s 128ms/step - loss: 0.2399 - accuracy: 0.9226 -
val_loss: 0.4534 - val_accuracy: 0.8579
Epoch 12/12
32/32 [==============================] - 4s 128ms/step - loss: 0.2091 - accuracy: 0.9321 -
val_loss: 0.4710 - val_accuracy: 0.8524
Test loss: 0.4710197150707245
Test accuracy: 0.8524228930473328
[[36  8  0  1  0  0  0  1  2  2]
 [ 0 96  0  2  0  0  0  0  2  0]
 [ 0  2 96  0  0  0  0  0  1  1]
 [ 1  1  0 51  1  0  5  1 26  0]
 [ 0  0  0  7 66  1 11  0  1  0]
 [ 0  0  0  1  0 94  2  0  3  0]
 [ 0  1  0  3  7  1 56  0 17  1]
 [ 2  3  0  4  0  0  1 87  1  2]
 [ 0  0  0  6  0  0  1  0 93  0]
 [ 0  0  1  0  0  0  0  0  0 99]]
train_multi finished in 00:01:01
```

Summary

In this project we deepened our understanding of object recognition, machine learning, classification models and representation of 3D objects as 2-dimensional arrays.

We explored various methods of optimizations of runtime and learning rate of convolutional neural networks, and found different ways of improving the accuracy and loss rates of our model.

We also experimented with functional programming in python, particularly using the NumPy library and generators. We noticed a great performance improvement once we applied all of our transformations using only functional programming.

To summarize the results we described in detail throughout this report, we think that the best practice for our model is to rotate the object around the z-axis since this approach led to the best results in terms of time consumption while still maintaining high accuracy rate and low loss rate.