

inspect_balloon_model

April 26, 2022

1 Mask R-CNN - Inspect Ballon Trained Model

Code and visualizations to test, debug, and evaluate the Mask R-CNN model.

```
[35]: import os
import sys
import random
import math
import re
import time
import numpy as np
import tensorflow as tf
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Root directory of the project
ROOT_DIR = os.path.abspath("C:/Users/majd4/Desktop/Bachelorarbeit/
                           ↳Bachelor-Arbeit-Daten/MaskRCNNProjekt/MaskRCNN_2/Mask_RCNN")

# Import Mask RCNN
sys.path.append(ROOT_DIR)  # To find local version of the library
from mrcnn import utils
from mrcnn import visualize
from mrcnn.visualize import display_images
import mrcnn.model as modellib
from mrcnn.model import log

from samples.balloon import balloon

%matplotlib inline

# Directory to save logs and trained model
MODEL_DIR = os.path.join(ROOT_DIR, "logs")

# Path to Ballon trained weights
# You can download this file from the Releases page
# https://github.com/matterport/Mask_RCNN/releases
```

```
#BALLOON_WEIGHTS_PATH = "/path/to/mask_rcnn_balloon.h5" # TODO: update this path
```

1.1 Configurations

```
[36]: config = balloon.BalloonConfig()
BALLOON_DIR = os.path.join(ROOT_DIR, "datasets/balloon")
```

```
[37]: # Override the training configurations with a few
# changes for inferencing.
class InferenceConfig(config.__class__):
    # Run detection on one image at a time
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1

config = InferenceConfig()
config.display()
```

Configurations:

BACKBONE	resnet101
BACKBONE_STRIDES	[4, 8, 16, 32, 64]
BATCH_SIZE	1
BBOX_STD_DEV	[0.1 0.1 0.2 0.2]
COMPUTE_BACKBONE_SHAPE	None
DETECTION_MAX_INSTANCES	100
DETECTION_MIN_CONFIDENCE	0.9
DETECTION_NMS_THRESHOLD	0.3
FPN_CLASSIF_FC_LAYERS_SIZE	1024
GPU_COUNT	1
GRADIENT_CLIP_NORM	5.0
IMAGES_PER_GPU	1
IMAGE_CHANNEL_COUNT	3
IMAGE_MAX_DIM	1024
IMAGE_META_SIZE	14
IMAGE_MIN_DIM	800
IMAGE_MIN_SCALE	0
IMAGE_RESIZE_MODE	square
IMAGE_SHAPE	[1024 1024 3]
LEARNING_MOMENTUM	0.9
LEARNING_RATE	0.001
LOSS_WEIGHTS	{'rpn_class_loss': 1.0, 'rpn_bbox_loss': 1.0, 'mrcnn_class_loss': 1.0, 'mrcnn_bbox_loss': 1.0, 'mrcnn_mask_loss': 1.0}
MASK_POOL_SIZE	14
MASK_SHAPE	[28, 28]
MAX_GT_INSTANCES	100
MEAN_PIXEL	[123.7 116.8 103.9]
MINI_MASK_SHAPE	(56, 56)

NAME	balloon
NUM_CLASSES	2
POOL_SIZE	7
POST_NMS_ROIS_INFERENCE	1000
POST_NMS_ROIS_TRAINING	2000
PRE_NMS_LIMIT	6000
ROI_POSITIVE_RATIO	0.33
RPN_ANCHOR RATIOS	[0.5, 1, 2]
RPN_ANCHOR_SCALES	(32, 64, 128, 256, 512)
RPN_ANCHOR_STRIDE	1
RPN_BBOX_STD_DEV	[0.1 0.1 0.2 0.2]
RPN_NMS_THRESHOLD	0.7
RPN_TRAIN_ANCHORS_PER_IMAGE	256
STEPS_PER_EPOCH	20
TOP_DOWN_PYRAMID_SIZE	256
TRAIN_BN	False
TRAIN_ROIS_PER_IMAGE	200
USE_MINI_MASK	True
USE_RPN_ROIS	True
VALIDATION_STEPS	50
WEIGHT_DECAY	0.0001

1.2 Notebook Preferences

```
[38]: # Device to load the neural network on.
# Useful if you're training a model on the same
# machine, in which case use CPU and leave the
# GPU for training.
DEVICE = "/cpu:0" # /cpu:0 or /gpu:0

# Inspect the model in training or inference modes
# values: 'inference' or 'training'
# TODO: code for 'training' test mode not ready yet
TEST_MODE = "inference"
```



```
[39]: def get_ax(rows=1, cols=1, size=16):
    """Return a Matplotlib Axes array to be used in
    all visualizations in the notebook. Provide a
    central point to control graph sizes.

    Adjust the size attribute to control how big to render images
    """
    _, ax = plt.subplots(rows, cols, figsize=(size*cols, size*rows))
    return ax
```

1.3 Load Validation Dataset

```
[40]: # Load validation dataset
dataset = balloon.BalloonDataset()
dataset.load_balloon(BALLOON_DIR, "val")

# Must call before using the dataset
dataset.prepare()

print("Images: {}\nClasses: {}".format(len(dataset.image_ids), dataset.
                                         class_names))
```

Images: 13
Classes: ['BG', 'balloon']

1.4 Load Model

```
[41]: # Create model in inference mode
with tf.device(DEVICE):
    model = modellib.MaskRCNN(mode="inference", model_dir=MODEL_DIR,
                               config=config)
```

```
[42]: # Set path to balloon weights file

# Download file from the Releases page and set its path
# https://github.com/matterport/Mask_RCNN/releases
weights_path1 = "C:/Users/majd4/Desktop/Bachelorarbeit/Bachelor-Arbeit-Daten/
                  MaskRCNNProjekt/MaskRCNN_2/Mask_RCNN/Ballonmodell10.h5"

# Or, load the last model you trained
# weights_path = model.find_last()

# Load weights
print("Loading weights ", weights_path1)
model.load_weights(weights_path1, by_name=True)
```

Loading weights C:/Users/majd4/Desktop/Bachelorarbeit/Bachelor-Arbeit-Daten/MaskRCNNProjekt/MaskRCNN_2/Mask_RCNN/Ballonmodell10.h5

1.5 Run Detection

```
[43]: image_id = random.choice(dataset.image_ids)
image, image_meta, gt_class_id, gt_bbox, gt_mask =\
    modellib.load_image_gt(dataset, config, image_id, use_mini_mask=False)
info = dataset.image_info[image_id]
print("image ID: {}.\n{} ({} {}) {}.".format(info["source"], info["id"], image_id,
                                              dataset.image_reference(image_id)))
```

```

# Run object detection
results = model.detect([image], verbose=1)

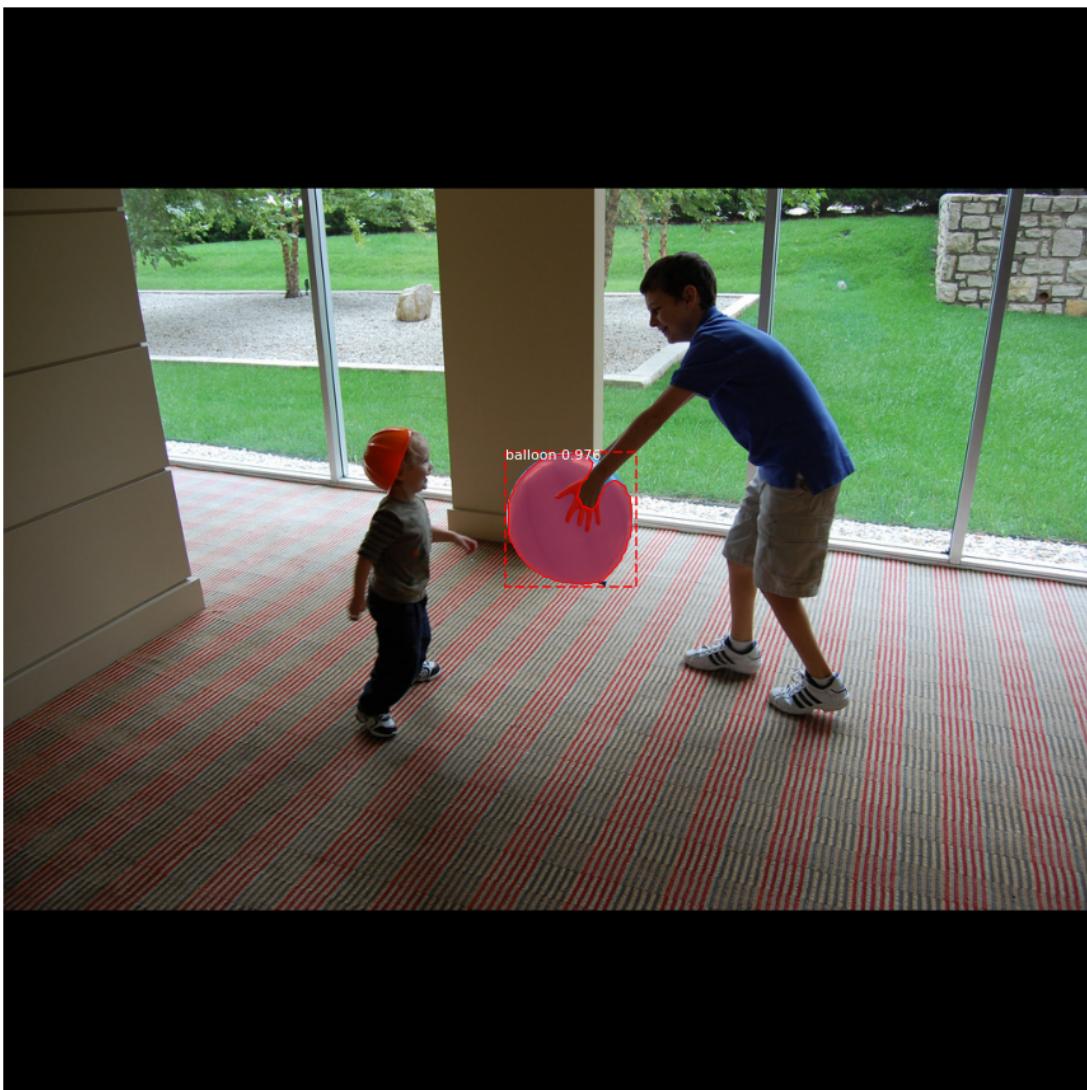
# Display results
ax = get_ax(1)
r = results[0]
visualize.display_instances(image, r['rois'], r['masks'], r['class_ids'],
                             dataset.class_names, r['scores'], ax=ax,
                             title="Predictions")
log("gt_class_id", gt_class_id)
log("gt_bbox", gt_bbox)
log("gt_mask", gt_mask)

```

image ID: balloon.2917282960_06beeee649a_b.jpg (11)
 C:\Users\majd4\Desktop\Bachelorarbeit\Bachelor-Arbeit-Daten\MaskRCNNProjekt\Mask
 RCNN_2\Mask_RCNN\datasets\balloon\val\2917282960_06beeee649a_b.jpg
 Processing 1 images

image	shape: (1024, 1024, 3)	min:	0.00000	max:
255.00000 uint8				
molded_images	shape: (1, 1024, 1024, 3)	min:	-123.70000	max:
151.10000 float64				
image_metas	shape: (1, 14)	min:	0.00000	max:
1024.00000 int32				
anchors	shape: (1, 261888, 4)	min:	-0.35390	max:
1.29134 float32				
gt_class_id	shape: (1,)	min:	1.00000	max:
1.00000 int32				
gt_bbox	shape: (1, 4)	min:	420.00000	max:
595.00000 int32				
gt_mask	shape: (1024, 1024, 1)	min:	0.00000	max:
1.00000 bool				

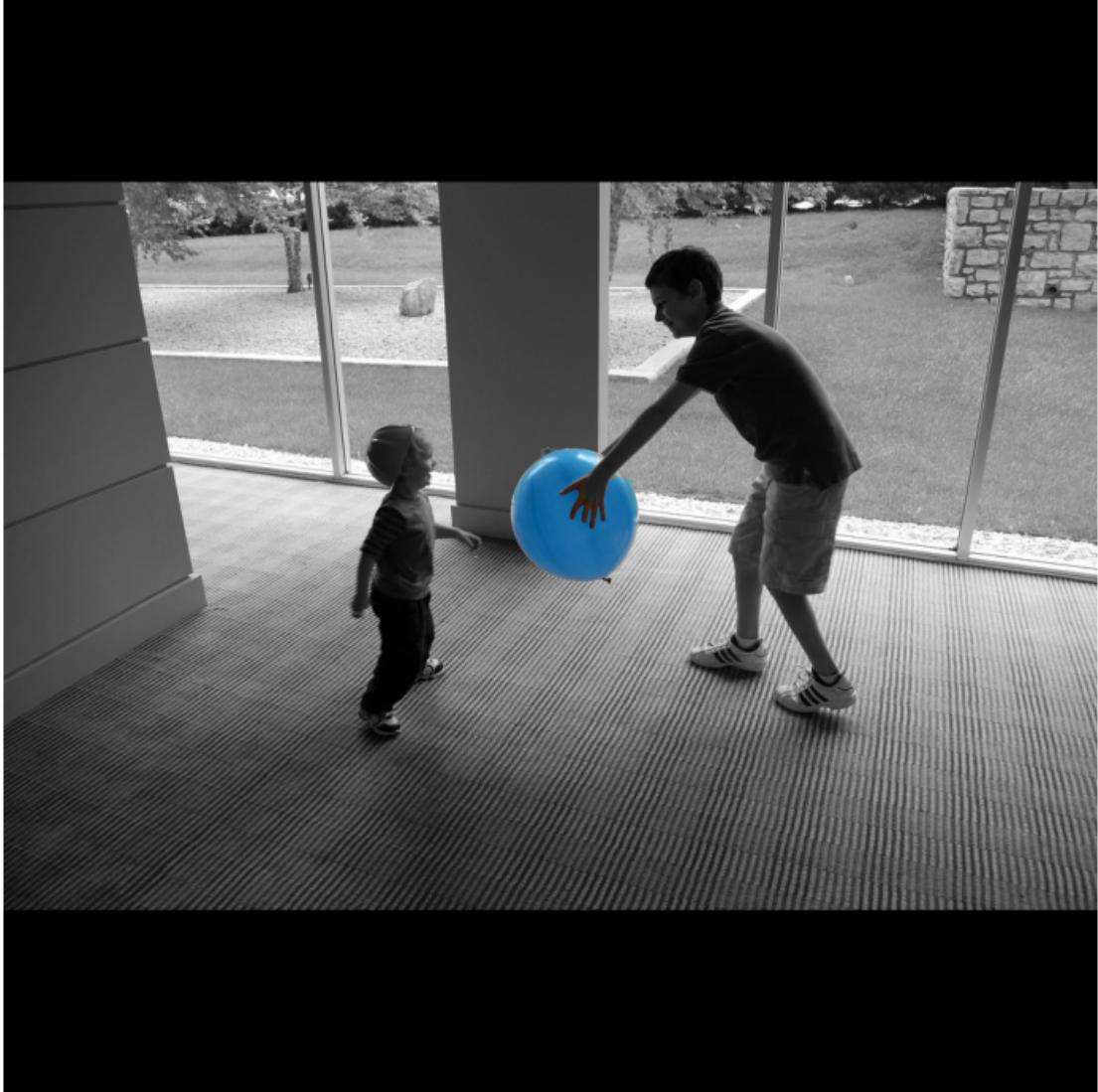
Predictions



1.6 Color Splash

This is for illustration. You can call `balloon.py` with the `splash` option to get better images without the black padding.

```
[44]: splash = balloon.color_splash(image, r['masks'])
display_images([splash], cols=1)
```



1.7 Step by Step Prediction

1.8 Stage 1: Region Proposal Network

The Region Proposal Network (RPN) runs a lightweight binary classifier on a lot of boxes (anchors) over the image and returns object/no-object scores. Anchors with high *objectness* score (positive anchors) are passed to the stage two to be classified.

Often, even positive anchors don't cover objects fully. So the RPN also regresses a refinement (a delta in location and size) to be applied to the anchors to shift it and resize it a bit to the correct boundaries of the object.

1.8.1 1.a RPN Targets

The RPN targets are the training values for the RPN. To generate the targets, we start with a grid of anchors that cover the full image at different scales, and then we compute the IoU of the anchors with ground truth object. Positive anchors are those that have an IoU ≥ 0.7 with any ground truth object, and negative anchors are those that don't cover any object by more than 0.3 IoU. Anchors in between (i.e. cover an object by IoU ≥ 0.3 but < 0.7) are considered neutral and excluded from training.

To train the RPN regressor, we also compute the shift and resizing needed to make the anchor cover the ground truth object completely.

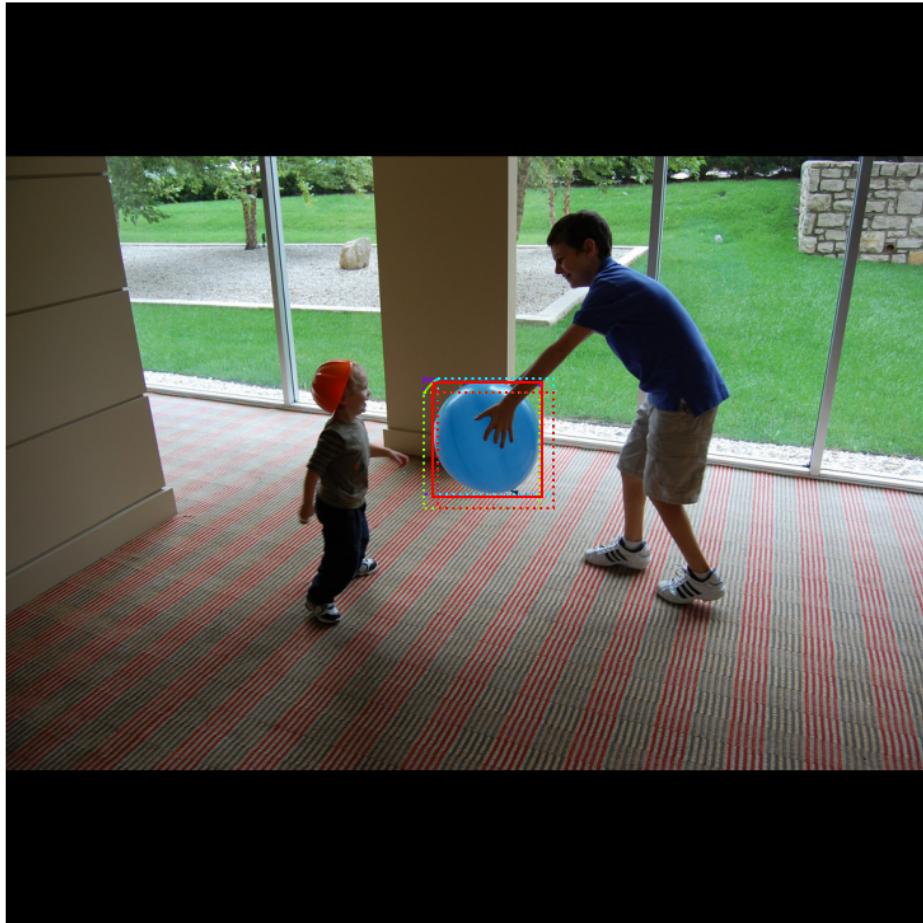
```
[45]: # Generate RPN training targets
# target_rpn_match is 1 for positive anchors, -1 for negative anchors
# and 0 for neutral anchors.
target_rpn_match, target_rpn_bbox = modellib.build_rpn_targets(
    image.shape, model.anchors, gt_class_id, gt_bbox, model.config)
log("target_rpn_match", target_rpn_match)
log("target_rpn_bbox", target_rpn_bbox)

positive_anchor_ix = np.where(target_rpn_match[:] == 1)[0]
negative_anchor_ix = np.where(target_rpn_match[:] == -1)[0]
neutral_anchor_ix = np.where(target_rpn_match[:] == 0)[0]
positive_anchors = model.anchors[positive_anchor_ix]
negative_anchors = model.anchors[negative_anchor_ix]
neutral_anchors = model.anchors[neutral_anchor_ix]
log("positive_anchors", positive_anchors)
log("negative_anchors", negative_anchors)
log("neutral anchors", neutral_anchors)

# Apply refinement deltas to positive anchors
refined_anchors = utils.apply_box_deltas(
    positive_anchors,
    target_rpn_bbox[:positive_anchors.shape[0]] * model.config.RPN_BBOX_STD_DEV)
log("refined_anchors", refined_anchors, )
```

target_rpn_match	shape: (261888,)	min: -1.00000	max:
1.00000 int32			
target_rpn_bbox	shape: (256, 4)	min: -0.97656	max:
0.50781 float64			
positive_anchors	shape: (4, 4)	min: 416.00000	max:
608.00000 float64			
negative_anchors	shape: (252, 4)	min: -64.00000	max:
1098.50967 float64			
neutral anchors	shape: (261632, 4)	min: -362.03867	max:
1322.03867 float64			
refined_anchors	shape: (4, 4)	min: 420.00000	max:
595.00000 float32			

```
[46]: # Display positive anchors before refinement (dotted) and  
# after refinement (solid).  
visualize.draw_boxes(image, boxes=positive_anchors,  
                     refined_boxes=refined_anchors, ax=get_ax())
```



1.8.2 1.b RPN Predictions

Here we run the RPN graph and display its predictions.

```
[47]: # Run RPN sub-graph  
pillar = model.keras_model.get_layer("ROI").output # node to start searching  
from
```

```

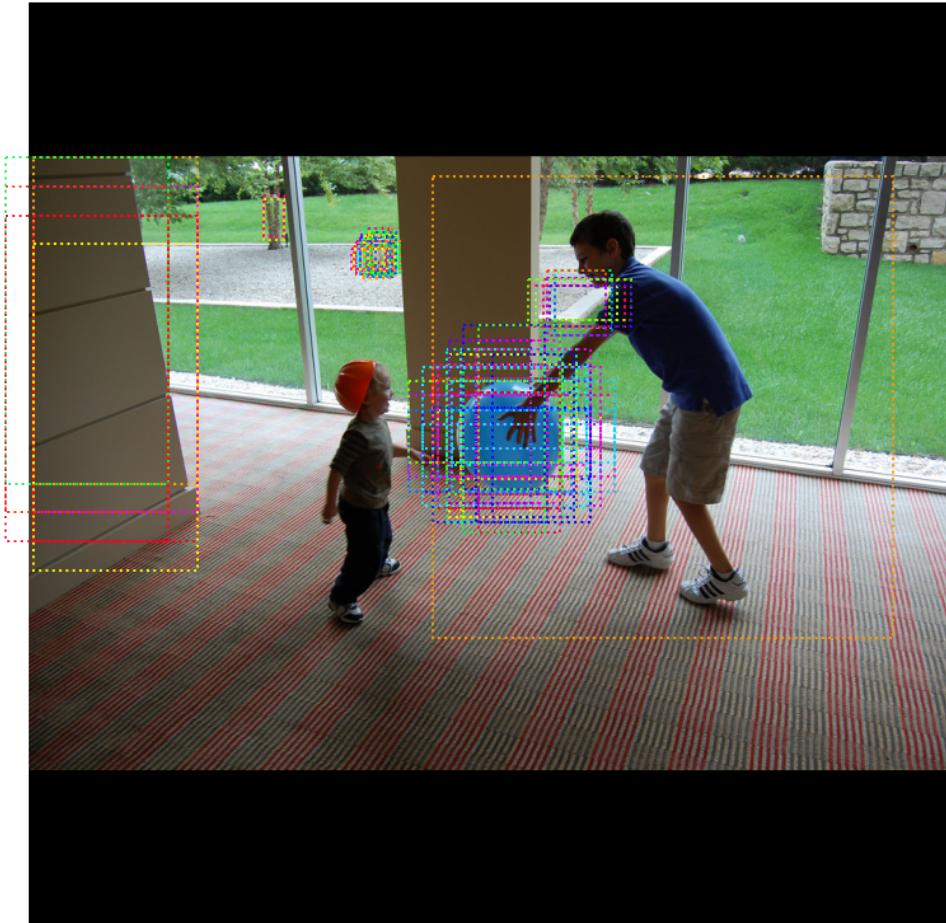
# TF 1.4 and 1.9 introduce new versions of NMS. Search for all names to support ↴
↪TF 1.3~1.10
nms_node = model.ancestor(pillar, "ROI/rpn_non_max_suppression:0")
if nms_node is None:
    nms_node = model.ancestor(pillar, "ROI/rpn_non_max_suppression/
↪NonMaxSuppressionV2:0")
if nms_node is None: #TF 1.9-1.10
    nms_node = model.ancestor(pillar, "ROI/rpn_non_max_suppression/
↪NonMaxSuppressionV3:0")

rpn = model.run_graph([image], [
    ("rpn_class", model.keras_model.get_layer("rpn_class").output),
    ("pre_nms_anchors", model.ancestor(pillar, "ROI/pre_nms_anchors:0")),
    ("refined_anchors", model.ancestor(pillar, "ROI/refined_anchors:0")),
    ("refined_anchors_clipped", model.ancestor(pillar, "ROI/
↪refined_anchors_clipped:0")),
    ("post_nms_anchor_ix", nms_node),
    ("proposals", model.keras_model.get_layer("ROI").output),
])

```

rpn_class	shape: (1, 261888, 2)	min:	0.00000	max:
1.00000 float32				
pre_nms_anchors	shape: (1, 6000, 4)	min:	-0.35390	max:
1.29134 float32				
refined_anchors	shape: (1, 6000, 4)	min:	-30.89931	max:
31.85384 float32				
refined_anchors_clipped	shape: (1, 6000, 4)	min:	0.00000	max:
1.00000 float32				
post_nms_anchor_ix	shape: (1000,)	min:	0.00000	max:
5214.00000 int32				
proposals	shape: (1, 1000, 4)	min:	0.00000	max:
1.00000 float32				

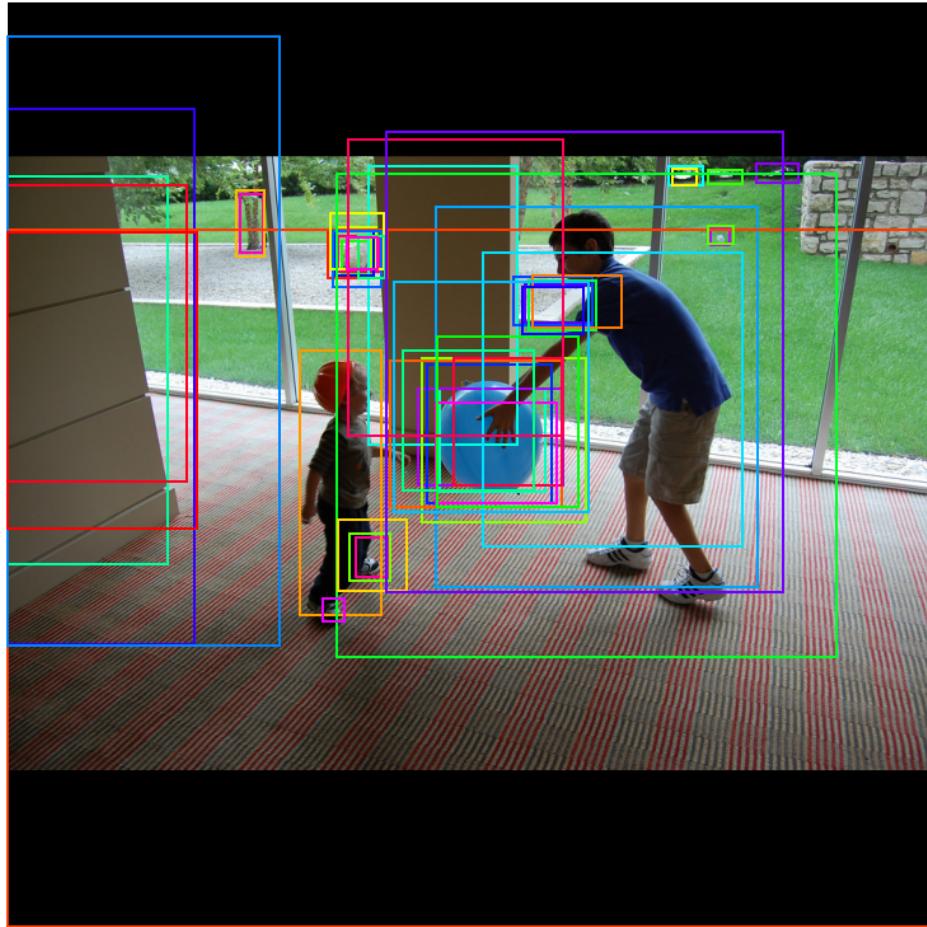
```
[48]: # Show top anchors by score (before refinement)
limit = 100
sorted_anchor_ids = np.argsort(rpn['rpn_class'][:, :, 1].flatten())[::-1]
visualize.draw_boxes(image, boxes=model.anchors[sorted_anchor_ids[:limit]], ↴
↪ax=get_ax())
```



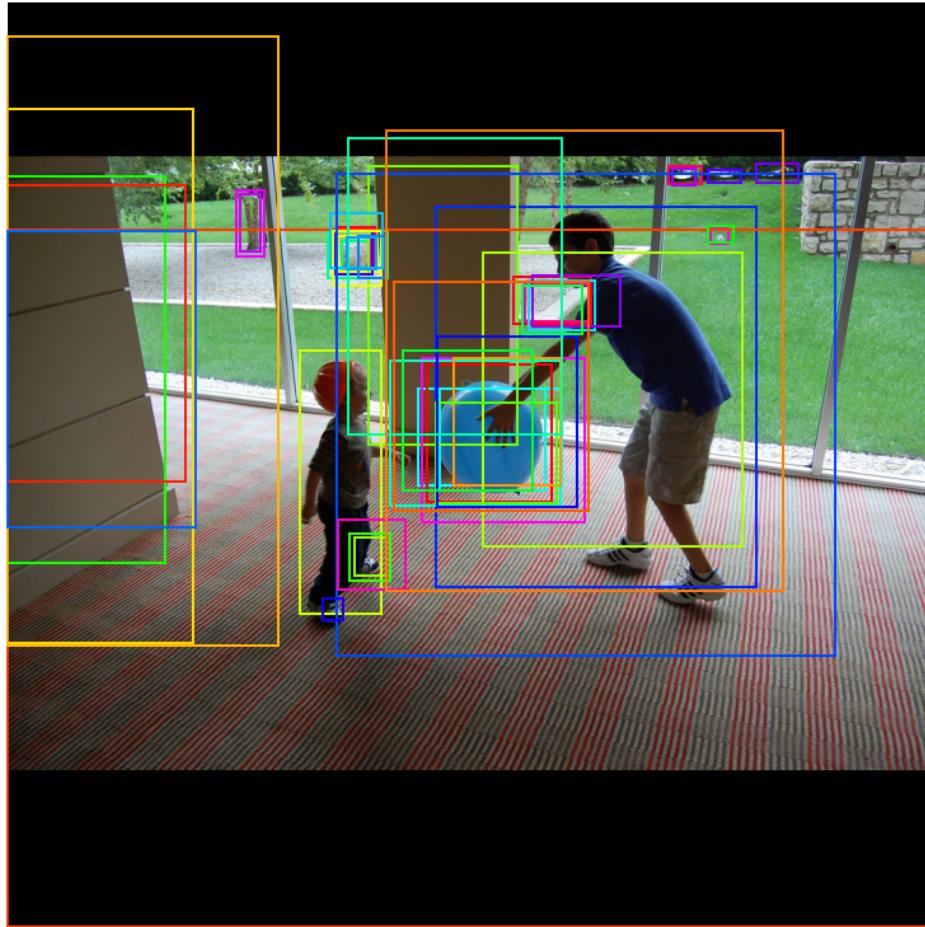
```
[49]: # Show top anchors with refinement. Then with clipping to image boundaries
limit = 50
ax = get_ax(1, 2)
pre_nms_anchors = utils.denorm_boxes(rpn["pre_nms_anchors"][0], image.shape[:2])
refined_anchors = utils.denorm_boxes(rpn["refined_anchors"][0], image.shape[:2])
refined_anchors_clipped = utils.denorm_boxes(rpn["refined_anchors_clipped"][0], ▾
    ↳image.shape[:2])
visualize.draw_boxes(image, boxes=pre_nms_anchors[:limit],
                     refined_boxes=refined_anchors[:limit], ax=ax[0])
visualize.draw_boxes(image, refined_boxes=refined_anchors_clipped[:limit], ▾
    ↳ax=ax[1])
```



```
[50]: # Show refined anchors after non-max suppression
limit = 50
ixs = rpn["post_nms_anchor_ix"][:limit]
visualize.draw_boxes(image, refined_boxes=refined_anchors_clipped[ixs], ↴
    ↪ax=get_ax())
```



```
[51]: # Show final proposals
# These are the same as the previous step (refined anchors
# after NMS) but with coordinates normalized to [0, 1] range.
limit = 50
# Convert back to image coordinates for display
h, w = config.IMAGE_SHAPE[:2]
proposals = rpn['proposals'][0, :limit] * np.array([h, w, h, w])
visualize.draw_boxes(image, refined_boxes=proposals, ax=get_ax())
```



1.9 Stage 2: Proposal Classification

This stage takes the region proposals from the RPN and classifies them.

1.9.1 2.a Proposal Classification

Run the classifier heads on proposals to generate class probabilities and bounding box regressions.

```
[52]: # Get input and output to classifier and mask heads.  
mrcnn = model.run_graph([image], [  
    ("proposals", model.keras_model.get_layer("ROI").output),  
    ("probs", model.keras_model.get_layer("mrcnn_class").output),  
    ("deltas", model.keras_model.get_layer("mrcnn_bbox").output),
```

```

        ("masks", model.keras_model.get_layer("mrcnn_mask").output),
        ("detections", model.keras_model.get_layer("mrcnn_detection").output),
    ])

```

proposals	shape: (1, 1000, 4)	min:	0.00000	max:
1.00000 float32				
probs	shape: (1, 1000, 2)	min:	0.00000	max:
1.00000 float32				
deltas	shape: (1, 1000, 2, 4)	min:	-4.25648	max:
4.27291 float32				
masks	shape: (1, 100, 28, 28, 2)	min:	0.00000	max:
1.00000 float32				
detections	shape: (1, 100, 6)	min:	0.00000	max:
1.00000 float32				

```

[53]: # Get detection class IDs. Trim zero padding.
det_class_ids = mrcnn['detections'][0, :, 4].astype(np.int32)
det_count = np.where(det_class_ids == 0)[0][0]
det_class_ids = det_class_ids[:det_count]
detections = mrcnn['detections'][0, :det_count]

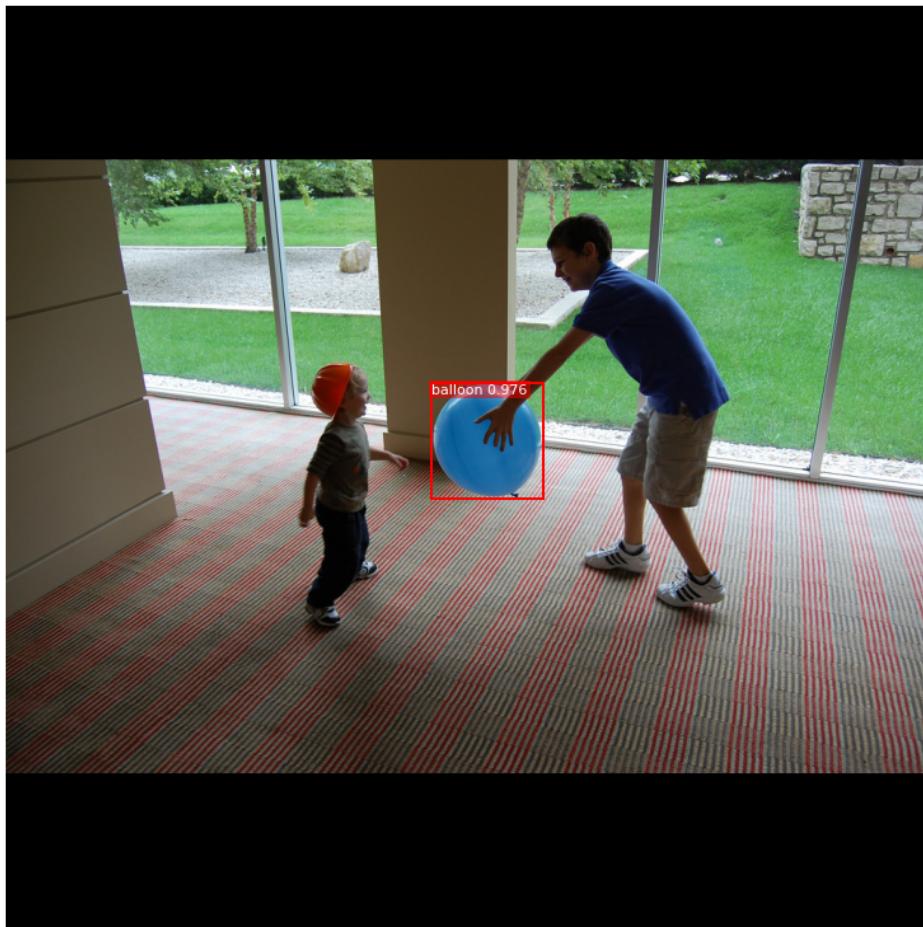
print("{} detections: {}".format(
    det_count, np.array(dataset.class_names)[det_class_ids]))

captions = ["{} {:.3f}{}".format(dataset.class_names[int(c)], s) if c > 0 else ""
            for c, s in zip(detections[:, 4], detections[:, 5])]
visualize.draw_boxes(
    image,
    refined_boxes=utils.denorm_boxes(detections[:, :4], image.shape[:2]),
    visibilities=[2] * len(detections),
    captions=captions, title="Detections",
    ax=get_ax())

```

1 detections: ['balloon']

Detections



1.9.2 2.c Step by Step Detection

Here we dive deeper into the process of processing the detections.

```
[54]: # Proposals are in normalized coordinates. Scale them
      # to image coordinates.
h, w = config.IMAGE_SHAPE[:2]
proposals = np.around(mrcnn["proposals"][0] * np.array([h, w, h, w])).astype(np.
    ↪int32)

# Class ID, score, and mask per proposal
roi_class_ids = np.argmax(mrcnn["probs"][0], axis=1)
```

```

roi_scores = mrcnn["probs"][0, np.arange(roi_class_ids.shape[0]), roi_class_ids]
roi_class_names = np.array(dataset.class_names)[roi_class_ids]
roi_positive_ixs = np.where(roi_class_ids > 0)[0]

# How many ROIs vs empty rows?
print("{} Valid proposals out of {}".format(np.sum(np.any(proposals, axis=1)), proposals.shape[0]))
print("{} Positive ROIs".format(len(roi_positive_ixs)))

# Class counts
print(list(zip(*np.unique(roi_class_names, return_counts=True))))

```

1000 Valid proposals out of 1000
14 Positive ROIs
[('BG', 986), ('balloon', 14)]

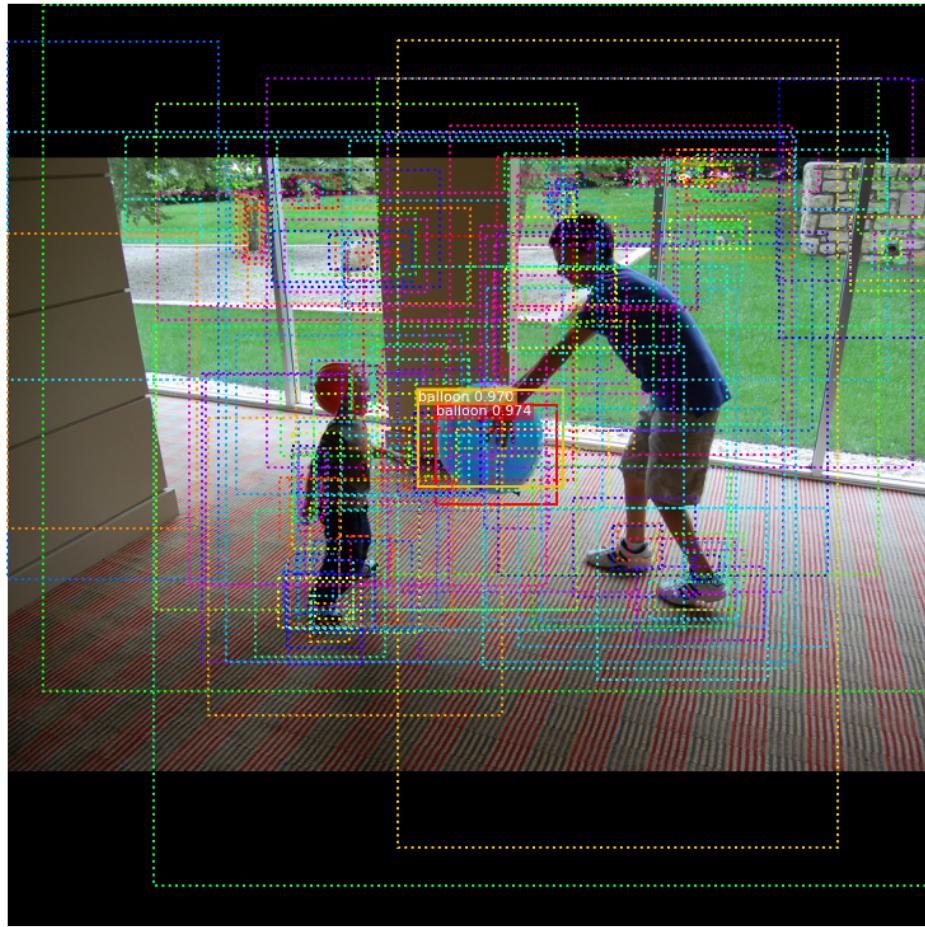
[55]: *# Display a random sample of proposals.*
Proposals classified as background are dotted, and
the rest show their class and confidence score.

```

limit = 200
ixs = np.random.randint(0, proposals.shape[0], limit)
captions = ["{} {:.3f}{}".format(dataset.class_names[c], s) if c > 0 else ""
            for c, s in zip(roi_class_ids[ixs], roi_scores[ixs])]
visualize.draw_boxes(image, boxes=proposals[ixs],
                      visibilities=np.where(roi_class_ids[ixs] > 0, 2, 1),
                      captions=captions, title="ROIs Before Refinement",
                      ax=get_ax())

```

ROIs Before Refinement



Apply Bounding Box Refinement

```
[56]: # Class-specific bounding box shifts.  
roi_bbox_specific = mrcnn["deltas"][0, np.arange(proposals.shape[0]),  
    ↪roi_class_ids]  
log("roi_bbox_specific", roi_bbox_specific)  
  
# Apply bounding box transformations  
# Shape: [N, (y1, x1, y2, x2)]  
refined_proposals = utils.apply_box_deltas(  
    proposals, roi_bbox_specific * config.BBOX_STD_DEV).astype(np.int32)  
log("refined_proposals", refined_proposals)
```

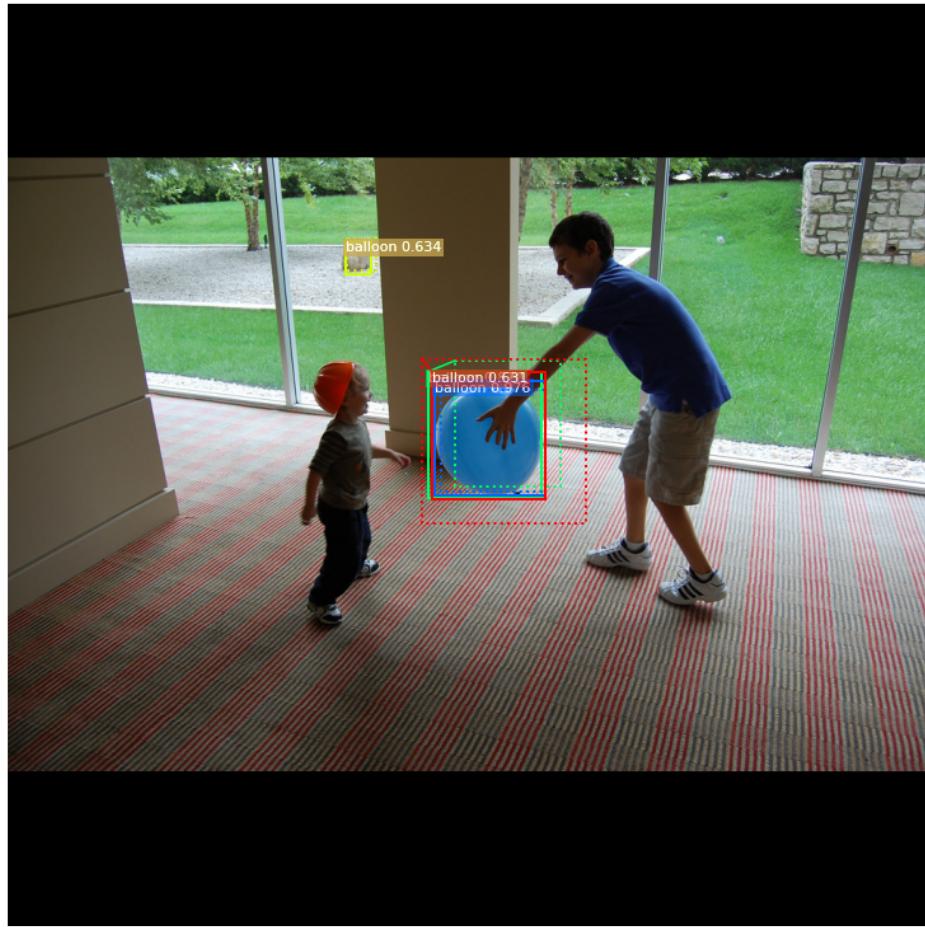
```

# Show positive proposals
# ids = np.arange(roi_boxes.shape[0]) # Display all
limit = 5
ids = np.random.randint(0, len(roi_positive_ixs), limit) # Display random
→sample
captions = ["{} {:.3f}{}".format(dataset.class_names[c], s) if c > 0 else ""
            for c, s in zip(roi_class_ids[roi_positive_ixs][ids], ↵
→roi_scores[roi_positive_ixs][ids])]
visualize.draw_boxes(image, boxes=proposals[roi_positive_ixs][ids],
                      refined_boxes=refined_proposals[roi_positive_ixs][ids],
                      visibilities=np.where(roi_class_ids[roi_positive_ixs][ids] ↵
→> 0, 1, 0),
                      captions=captions, title="ROIs After Refinement",
                      ax=get_ax())

```

roi_bbox_specific	shape: (1000, 4)	min: -4.25648	max:
4.27291 float32			
refined_proposals	shape: (1000, 4)	min: -493.00000	max:
1804.00000 int32			

ROIs After Refinement



Filter Low Confidence Detections

```
[57]: # Remove boxes classified as background
keep = np.where(roi_class_ids > 0)[0]
print("Keep {} detections:\n{}".format(keep.shape[0], keep))
```

```
Keep 14 detections:
[ 0   1   3   4   8  10  13  15  19  27  38  39  49 143]
```

```
[58]: # Remove low confidence detections
keep = np.intersect1d(keep, np.where(roi_scores >= config.
    DETECTION_MIN_CONFIDENCE)[0])
print("Remove boxes below {} confidence. Keep {}:\n{}".format(
```

```
config.DETECTION_MIN_CONFIDENCE, keep.shape[0], keep))
```

Remove boxes below 0.9 confidence. Keep 4:
[0 3 4 27]

Per-Class Non-Max Suppression

```
[59]: # Apply per-class non-max suppression
pre_nms_boxes = refined_proposals[keep]
pre_nms_scores = roi_scores[keep]
pre_nms_class_ids = roi_class_ids[keep]

nms_keep = []
for class_id in np.unique(pre_nms_class_ids):
    # Pick detections of this class
    ixs = np.where(pre_nms_class_ids == class_id)[0]
    # Apply NMS
    class_keep = utils.non_max_suppression(pre_nms_boxes[ixs],
                                            pre_nms_scores[ixs],
                                            config.DETECTION_NMS_THRESHOLD)
    # Map indices
    class_keep = keep[ixs[class_keep]]
    nms_keep = np.union1d(nms_keep, class_keep)
    print("{:22}: {} -> {}".format(dataset.class_names[class_id][:20],
                                    keep[ixs], class_keep))

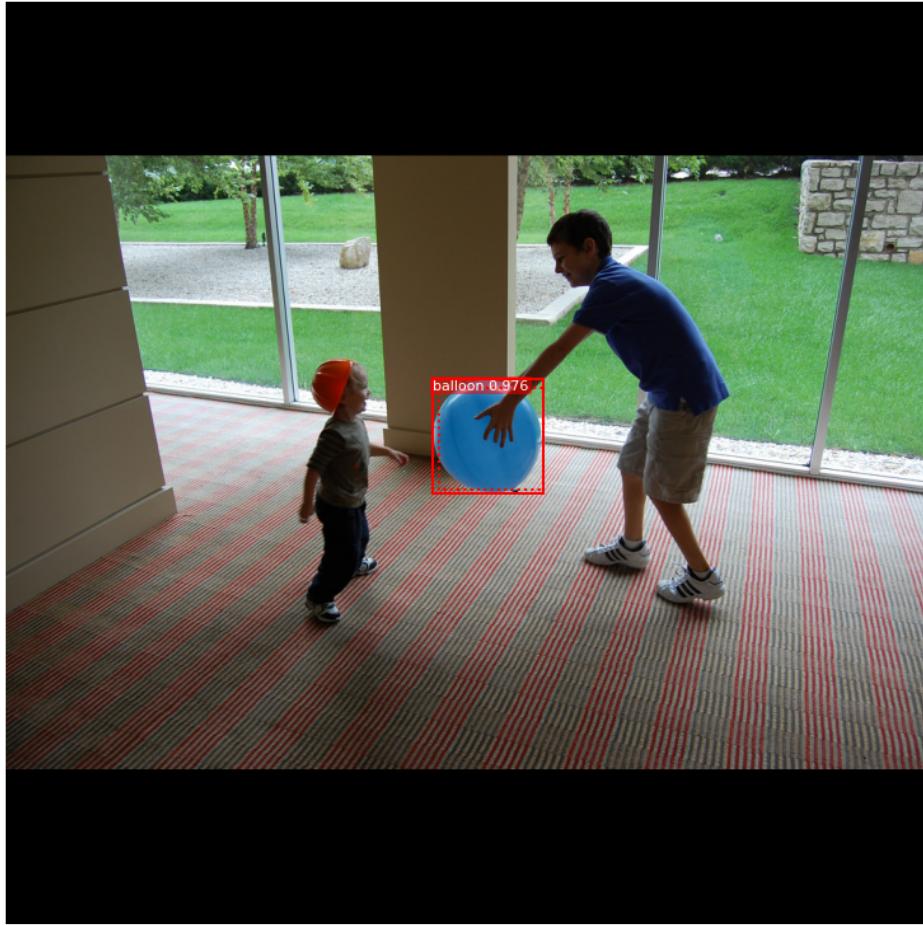
keep = np.intersect1d(keep, nms_keep).astype(np.int32)
print("\nKept after per-class NMS: {}\n".format(keep.shape[0], keep))
```

balloon : [0 3 4 27] -> [0]

Kept after per-class NMS: 1
[0]

```
[60]: # Show final detections
ixs = np.arange(len(keep)) # Display all
# ixs = np.random.randint(0, len(keep), 10) # Display random sample
captions = ["{} {:.3f}{}".format(dataset.class_names[c], s) if c > 0 else ""
            for c, s in zip(roi_class_ids[keep][ixs], roi_scores[keep][ixs])]
visualize.draw_boxes(
    image, boxes=proposals[keep][ixs],
    refined_boxes=refined_proposals[keep][ixs],
    visibilities=np.where(roi_class_ids[keep][ixs] > 0, 1, 0),
    captions=captions, title="Detections after NMS",
    ax=get_ax())
```

Detections after NMS



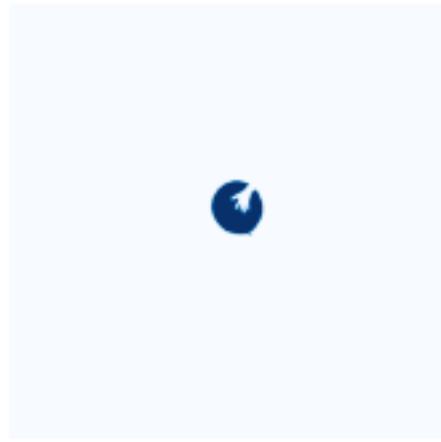
1.10 Stage 3: Generating Masks

This stage takes the detections (refined bounding boxes and class IDs) from the previous layer and runs the mask head to generate segmentation masks for every instance.

1.10.1 3.a Mask Targets

These are the training targets for the mask branch

```
[61]: display_images(np.transpose(gt_mask, [2, 0, 1]), cmap="Blues")
```



1.10.2 3.b Predicted Masks

```
[62]: # Get predictions of mask head
mrcnn = model.run_graph([image], [
    ("detections", model.keras_model.get_layer("mrcnn_detection").output),
    ("masks", model.keras_model.get_layer("mrcnn_mask").output),
])

# Get detection class IDs. Trim zero padding.
det_class_ids = mrcnn['detections'][0, :, 4].astype(np.int32)
det_count = np.where(det_class_ids == 0)[0][0]
det_class_ids = det_class_ids[:det_count]

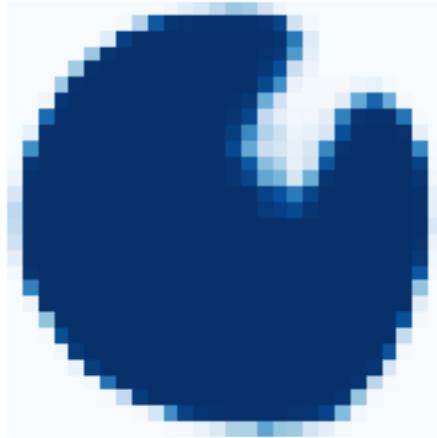
print("{} detections: {}".format(
    det_count, np.array(dataset.class_names)[det_class_ids]))
```

```
detections           shape: (1, 100, 6)      min: 0.00000 max:
1.00000 float32
masks                shape: (1, 100, 28, 28, 2) min: 0.00000 max:
1.00000 float32
1 detections: ['balloon']
```

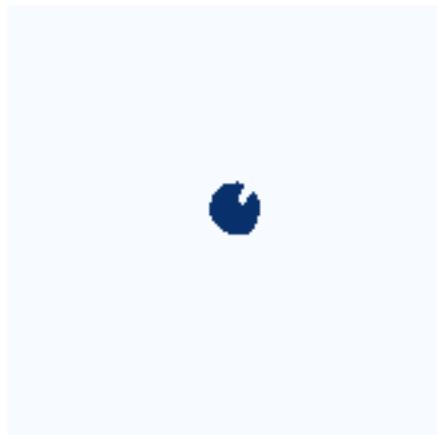
```
[63]: # Masks
det_boxes = utils.denorm_boxes(mrcnn["detections"][0, :, :4], image.shape[:2])
det_mask_specific = np.array([mrcnn["masks"][0, i, :, :, c]
                             for i, c in enumerate(det_class_ids)])
det_masks = np.array([utils.unmold_mask(m, det_boxes[i], image.shape)
                      for i, m in enumerate(det_mask_specific)])
log("det_mask_specific", det_mask_specific)
log("det_masks", det_masks)
```

```
det_mask_specific      shape: (1, 28, 28)          min: 0.00000 max:  
1.00000 float32  
det_masks              shape: (1, 1024, 1024)        min: 0.00000 max:  
1.00000 bool
```

```
[64]: display_images(det_mask_specific[:4] * 255, cmap="Blues", interpolation="none")
```



```
[65]: display_images(det_masks[:4] * 255, cmap="Blues", interpolation="none")
```



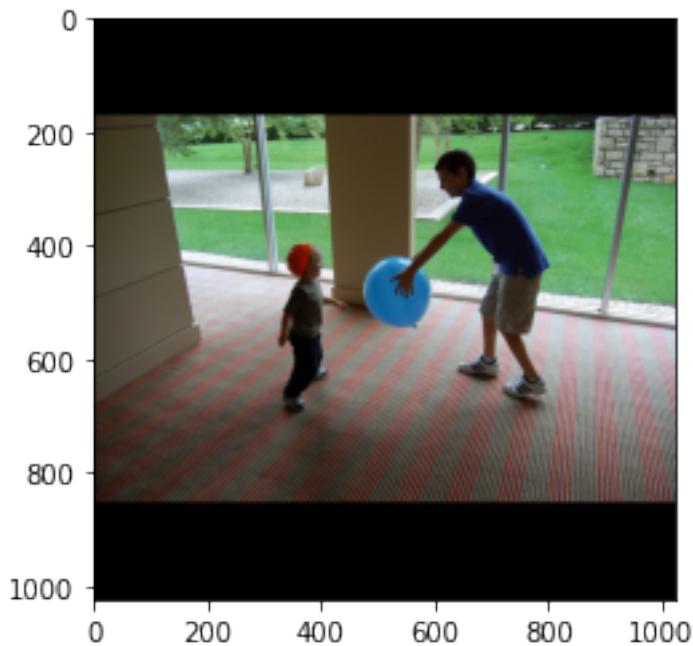
1.11 Visualize Activations

In some cases it helps to look at the output from different layers and visualize them to catch issues and odd patterns.

```
[66]: # Get activations of a few sample layers
activations = model.run_graph([image], [
    ("input_image", tf.identity(model.keras_model.
    ↪get_layer("input_image").output)),
    ("res2c_out", model.keras_model.get_layer("res2c_out").output),
    ("res3c_out", model.keras_model.get_layer("res3c_out").output),
    ("res4w_out", model.keras_model.get_layer("res4w_out").output), # ↪
    ↪for resnet100
    ("rpn_bbox", model.keras_model.get_layer("rpn_bbox").output),
    ("roi", model.keras_model.get_layer("ROI").output),
])
])
```

input_image	shape: (1, 1024, 1024, 3)	min: -123.70000	max:
151.10001 float32			
res2c_out	shape: (1, 256, 256, 256)	min: 0.00000	max:
27.69915 float32			
res3c_out	shape: (1, 128, 128, 512)	min: 0.00000	max:
28.80073 float32			
res4w_out	shape: (1, 64, 64, 1024)	min: 0.00000	max:
58.61688 float32			
rpn_bbox	shape: (1, 261888, 4)	min: -8.21568	max:
109.06531 float32			
roi	shape: (1, 1000, 4)	min: 0.00000	max:
1.00000 float32			

```
[67]: # Input image (normalized)
_ = plt.imshow(modellib.unmold_image(activations["input_image"])[0], config))
```



```
[68]: # Backbone feature map  
display_images(np.transpose(activations["res2c_out"] [0,:,:,:4], [2, 0, 1]),  
             cols=4)
```

