Mask R-CNN - Inspect Trained Model

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

In [26]:
```
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
import skimage

# Root directory of the project
ROOT_DIR = os.path.abspath("../")

# 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 mrcnn.config import Config
from mrcnn import utils
import mrcnn.model as modellib
from mrcnn import visualize
from mrcnn.model import log


%matplotlib inline

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

# Local path to trained weights file
COCO_MODEL_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")
# Download COCO trained weights from Releases if needed
if not os.path.exists(COCO_MODEL_PATH):
    utils.download_trained_weights(COCO_MODEL_PATH)

# Path to Shapes trained weights
SHAPES_MODEL_PATH = os.path.join(ROOT_DIR, "logs/ped20181117T1835/mas
k_rcnn_ped_0001.h5")
```

Configurations

In [5]:
```
class PedConfig(Config):
    """Configuration for training on the toy shapes dataset.
    Derives from the base Config class and overrides values specific
    to the toy shapes dataset.
    """
    # Give the configuration a recognizable name
    NAME = "ped"

    # Train on 1 GPU and 8 images per GPU. We can put multiple images
 on each
    # GPU because the images are small. Batch size is 8 (GPUs * image
s/GPU).
    GPU_COUNT = 1
    IMAGES_PER_GPU = 2

    # Number of classes (including background)
    NUM_CLASSES = 1 + 3  # background + 3 shapes

    # Use small images for faster training. Set the limits of the sma
ll side
    # the large side, and that determines the image shape.
    IMAGE_MIN_DIM = 480
    IMAGE_MAX_DIM = 640
    IMAGE_CHANNEL_COUNT = 3
    USE_MINI_MASK = False

    # Use smaller anchors because our image and objects are small
    RPN_ANCHOR_SCALES = (8, 16, 32, 64, 128)  # anchor side in pixels

    # Reduce training ROIs per image because the images are small and
 have
    # few objects. Aim to allow ROI sampling to pick 33% positive ROI
s.
    TRAIN_ROIS_PER_IMAGE = 32

    # Use a small epoch since the data is simple
    STEPS_PER_EPOCH = 100

    # use small validation steps since the epoch is small
    VALIDATION_STEPS = 5

config = PedConfig()
config.display()
```

```
Configurations:
BACKBONE                       resnet101
BACKBONE_STRIDES               [4, 8, 16, 32, 64]
BATCH_SIZE                     2
BBOX_STD_DEV                   [0.1 0.1 0.2 0.2]
COMPUTE_BACKBONE_SHAPE         None
DETECTION_MAX_INSTANCES        100
DETECTION_MIN_CONFIDENCE       0.7
DETECTION_NMS_THRESHOLD        0.3
FPN_CLASSIF_FC_LAYERS_SIZE     1024
GPU_COUNT                      1
GRADIENT_CLIP_NORM             5.0
IMAGES_PER_GPU                 2
IMAGE_CHANNEL_COUNT            3
IMAGE_MAX_DIM                  640
IMAGE_META_SIZE                16
IMAGE_MIN_DIM                  480
IMAGE_MIN_SCALE                0
IMAGE_RESIZE_MODE              square
IMAGE_SHAPE                    [640 640   3]
LEARNING_MOMENTUM              0.9
LEARNING_RATE                  0.001
LOSS_WEIGHTS                   {'rpn_class_loss': 1.0, 'rpn_bbox_los
s': 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                           ped
NUM_CLASSES                    4
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              (8, 16, 32, 64, 128)
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                100
TOP_DOWN_PYRAMID_SIZE          256
TRAIN_BN                       False
TRAIN_ROIS_PER_IMAGE           32
USE_MINI_MASK                  False
USE_RPN_ROIS                   True
VALIDATION_STEPS               5
WEIGHT_DECAY                   0.0001
```

In [13]:
```
class PedDataset(utils.Dataset):
    """Generates the shapes synthetic dataset. The dataset consists o
f simple
    shapes (triangles, squares, circles) placed randomly on a blank s
urface.
    The images are generated on the fly. No file access required.
    """

    def load_ped(self, load_path="../Datasets/", istrain=True, st_ind
ex=None, batch_size=20):
        """Generate the requested number of synthetic images.
        count: number of images to generate.
        height, width: the size of the generated images.
        """
        # Add classes
        self.add_class("ped", 1, "bike")
        self.add_class("ped", 2, "car")
        self.add_class("ped", 3, "person")

        if istrain is True:
            self.X = np.load(load_path+'X_train.npy')
            self.X_mask = np.load(load_path+'X_mask_train.npy')
            self.y = np.load(load_path+'y_train.npy')


        if istrain is False:
            self.X = np.load(load_path+'X_val.npy')
            self.X_mask = np.load(load_path+'X_mask_val.npy')
            self.y = np.load(load_path+'y_val.npy')

        self.N = self.X.shape[0]

        if istrain:
            self.X = self.X[st_index:st_index+batch_size]
            self.X_mask = self.X_mask[st_index:st_index+batch_size]
            self.y = self.y[st_index:st_index+batch_size]

        for i in range(self.X.shape[0]):
            self.add_image("ped", image_id=i, path=None)

    def load_image(self, image_id, istrain=True):
        """Generate an image from the specs of the given image ID.
        Typically this function loads the image from a file, but
        in this case it generates the image on the fly from the
        specs in image_info.
        """
        image = self.X[image_id]
        if image.ndim != 3:
            image = skimage.color.gray2rgb(image)
        return image

    def image_reference(self, image_id):
        """Return the shapes data of the image."""
        return self.X[image_id].shape

    def load_mask(self, image_id):
```

```
        """Generate instance masks for shapes of the given image ID.
        """
        mask = np.zeros((self.X_mask.shape[1], self.X_mask.shape[2],
1))
        mask[:, :, 0] = np.logical_not(self.X_mask[image_id]).astype(
np.bool_)
#         print('masks are bool!')
        class_ids = np.array([self.y[image_id]+1])
        return mask, class_ids.astype(np.int32)


    def prepare(self):
        """Prepares the Dataset class for use.

        TODO: class map is not supported yet. When done, it should ha
ndle mapping
              classes from different datasets to the same class ID.
        """

        self.num_classes = int(max(self.y[:]) + 2)
        self.class_ids = np.arange(self.num_classes)
        self.class_names = ["BG", "BIKE", "CAR", "PERSON"]
        self.num_images = self.X.shape[0]
        self._image_ids = np.arange(self.num_images)

        # Mapping from source class and image IDs to internal IDs
        self.class_from_source_map = {"{}.{}".format(info['source'],
info['id']): id
                                      for info, id in zip(self.class_
info, self.class_ids)}
        self.image_from_source_map = {"{}.{}".format(info['source'],
info['id']): id
                                      for info, id in zip(self.image_
info, self.image_ids)}

        # Map sources to class_ids they support
        self.sources = list(set([i['source'] for i in self.class_info
]))
        self.source_class_ids = {}
        # Loop over datasets
        for source in self.sources:
            self.source_class_ids[source] = []
            # Find classes that belong to this dataset
            for i, info in enumerate(self.class_info):
                # Include BG class in all datasets
                if i == 0 or source == info['source']:
                    self.source_class_ids[source].append(i)
```

In [14]:
```
# Run one of the code blocks

config = PedConfig()

# Shapes toy dataset
# import shapes
# config = shapes.ShapesConfig()

# MS COCO Dataset
# import coco
# config = coco.CocoConfig()
# COCO_DIR = "path to COCO dataset"  # TODO: enter value here
```

In [15]:
```
# 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       0.7
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                  640
IMAGE_META_SIZE                16
IMAGE_MIN_DIM                  480
IMAGE_MIN_SCALE                0
IMAGE_RESIZE_MODE              square
IMAGE_SHAPE                    [640 640   3]
LEARNING_MOMENTUM              0.9
LEARNING_RATE                  0.001
LOSS_WEIGHTS                   {'rpn_class_loss': 1.0, 'rpn_bbox_los
s': 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                           ped
NUM_CLASSES                    4
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              (8, 16, 32, 64, 128)
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                100
TOP_DOWN_PYRAMID_SIZE          256
TRAIN_BN                       False
TRAIN_ROIS_PER_IMAGE           32
USE_MINI_MASK                  False
USE_RPN_ROIS                   True
VALIDATION_STEPS               5
WEIGHT_DECAY                   0.0001
```

Notebook Preferences

In [16]:
```
# 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"
```

In [17]:
```
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
```

Load Validation Dataset

In [27]:
```
# Build validation dataset
dataset = PedDataset()
dataset.load_ped(istrain=False)

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

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

```
Images: 90
Classes: ['BG', 'BIKE', 'CAR', 'PERSON']
```

Load Model

In [24]:
```
# Create model in inference mode
with tf.device(DEVICE):
    model = modellib.MaskRCNN(mode="inference", model_dir=MODEL_DIR,
                              config=config)

# # Set weights file path
# if config.NAME == "shapes":
#     weights_path = SHAPES_MODEL_PATH
# elif config.NAME == "coco":
#     weights_path = COCO_MODEL_PATH
# Or, uncomment to load the last model you trained
weights_path = SHAPES_MODEL_PATH

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

```
Loading weights  /home/paperspace/Documents/PedNet/logs/ped20181117T1
835/mask_rcnn_ped_0001.h5
Re-starting from epoch 1
```

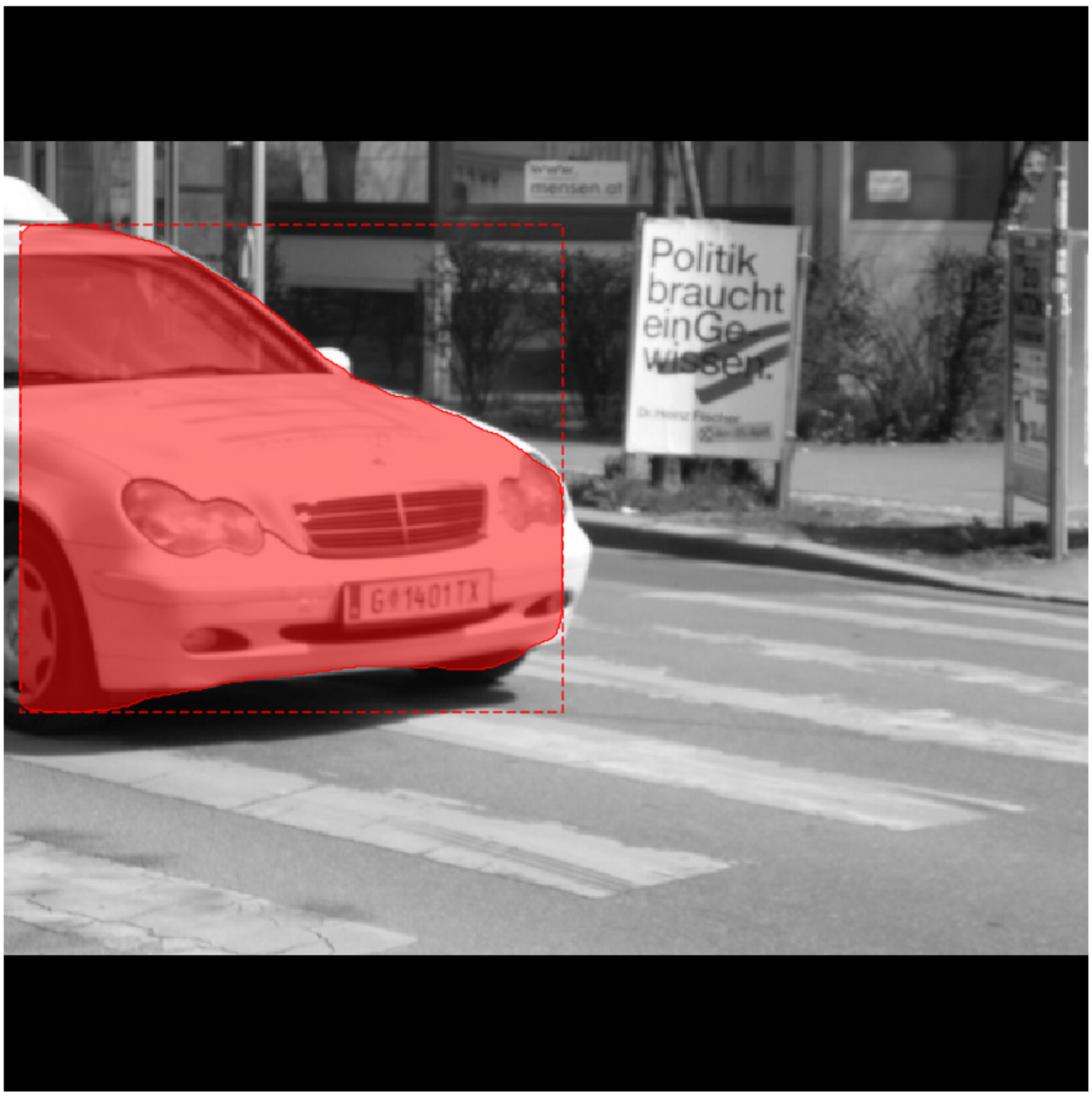
Run Detection

In [28]:

```
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=F
alse)
info = dataset.image_info[image_id]
print("image ID: {}.{} ({}) {}".format(info["source"], info["id"], im
age_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_id
s'],
                            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)
```

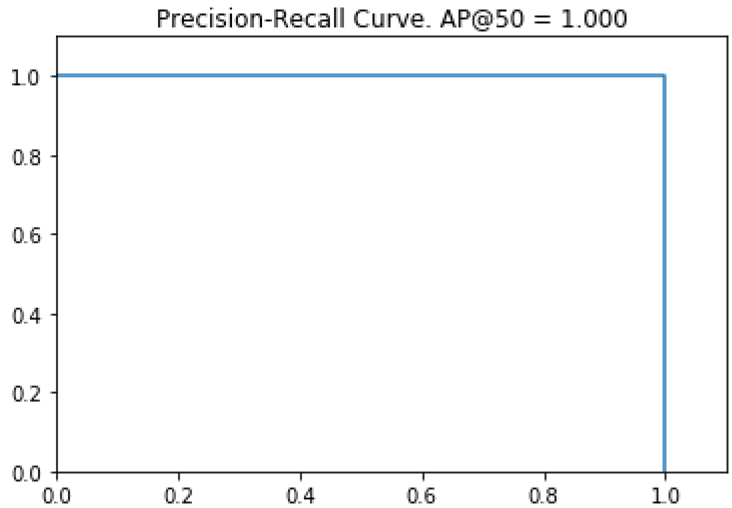
```
image ID: ped.63 (63) (480, 640)
Processing 1 images
image                    shape: (640, 640, 3)         min:    0.00000
  max:  255.00000  float64
molded_images            shape: (1, 640, 640, 3)      min: -123.70000
  max:  151.10000  float64
image_metas              shape: (1, 16)               min:    0.00000
  max:  640.00000  int64
anchors                  shape: (1, 102300, 4)        min:   -0.14164
  max:    1.04149  float32
gt_class_id              shape: (1, 1)                min:    2.00000
  max:    2.00000  int32
gt_bbox                  shape: (1, 4)                min:    0.00000
  max:  433.00000  int32
gt_mask                  shape: (640, 640, 1)         min:    0.00000
  max:    1.00000  bool
```

Predictions

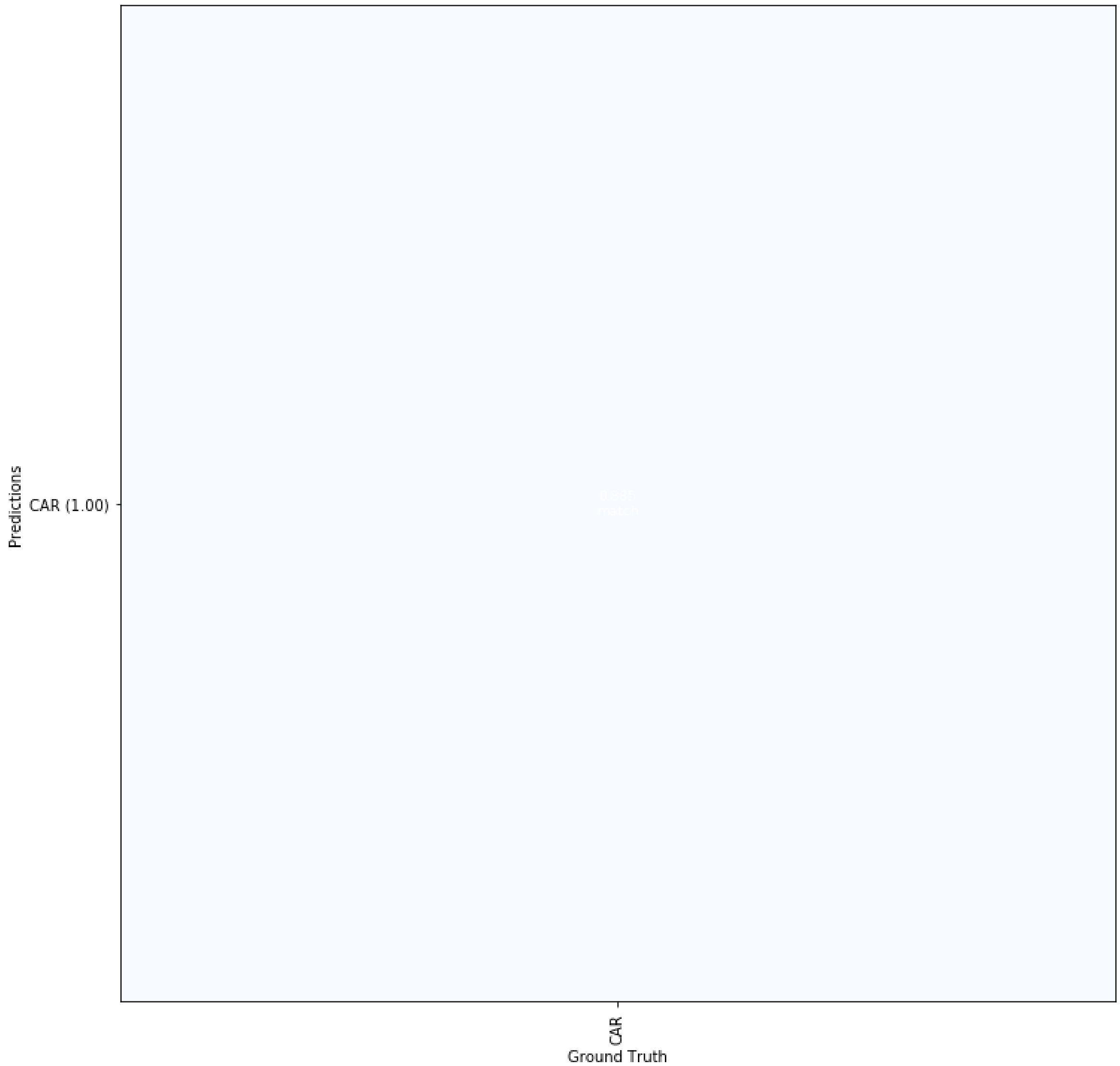


Precision-Recall

In [29]:
```
# Draw precision-recall curve
AP, precisions, recalls, overlaps = utils.compute_ap(gt_bbox, gt_clas
s_id, gt_mask,
                                          r['rois'], r['class_ids'],
r['scores'], r['masks'])
visualize.plot_precision_recall(AP, precisions, recalls)
```



In [30]:
```python
# Grid of ground truth objects and their predictions
visualize.plot_overlaps(gt_class_id, r['class_ids'], r['scores'],
                        overlaps, dataset.class_names)
```



Compute mAP @ IoU=50 on Batch of Images

In [31]:
```
# Compute VOC-style Average Precision
def compute_batch_ap(image_ids):
    APs = []
    for image_id in image_ids:
        # Load image
        image, image_meta, gt_class_id, gt_bbox, gt_mask =\
            modellib.load_image_gt(dataset, config,
                                   image_id, use_mini_mask=False)
        # Run object detection
        results = model.detect([image], verbose=0)
        # Compute AP
        r = results[0]
        AP, precisions, recalls, overlaps =\
            utils.compute_ap(gt_bbox, gt_class_id, gt_mask,
                              r['rois'], r['class_ids'], r['scores'],
 r['masks'])
        APs.append(AP)
    return APs

# Pick a set of random images
image_ids = np.random.choice(dataset.image_ids, 10)
APs = compute_batch_ap(image_ids)
print("mAP @ IoU=50: ", np.mean(APs))
```

```
mAP @ IoU=50:  1.0
```

Step by Step Prediction

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.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.

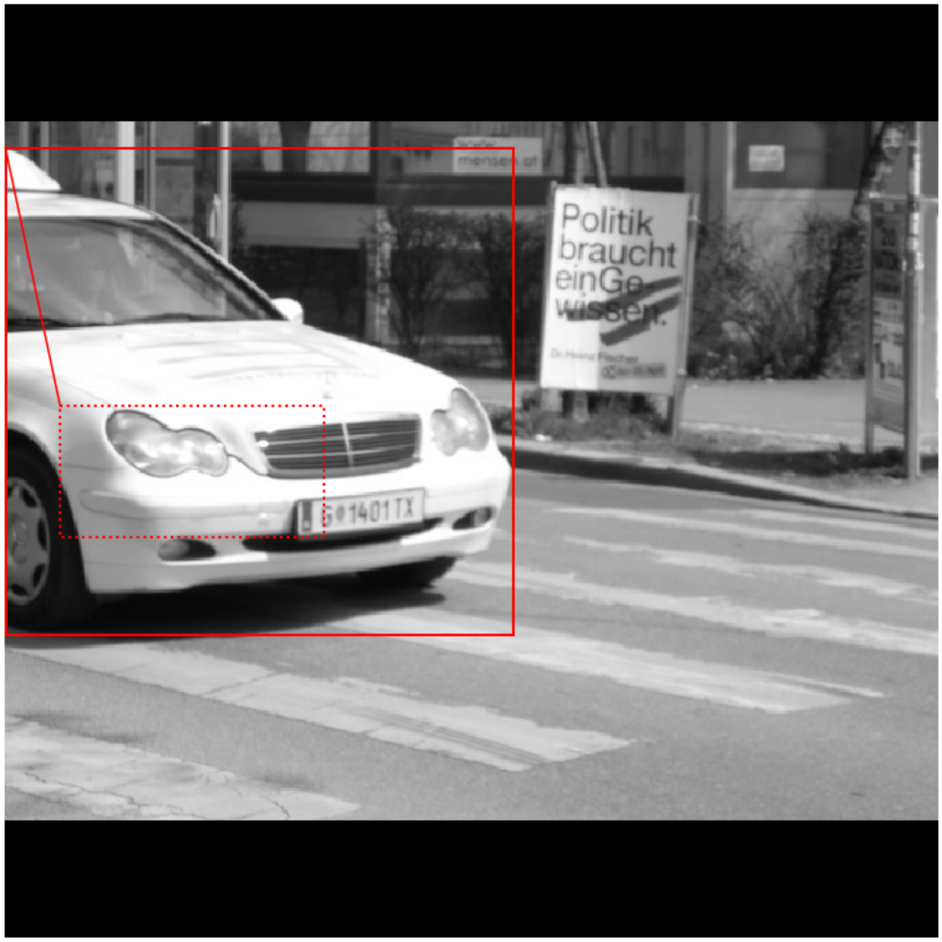
To train the RPN regressor, we also compute the shift and resizing needed to make the anchor cover the ground truth object completely.

In [32]:

```
# Generate RPN trainig 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_BB
OX_STD_DEV)
log("refined_anchors", refined_anchors, )
```

```
target_rpn_match         shape: (102300,)             min:   -1.00000
  max:    1.00000  int32
target_rpn_bbox          shape: (256, 4)              min:   -6.02146
  max:    6.54337  float64
positive_anchors         shape: (1, 4)                min:   37.49033
  max:  365.25483  float64
negative_anchors         shape: (255, 4)              min:   -6.62742
  max:  646.62742  float64
neutral anchors          shape: (102044, 4)           min:  -90.50967
  max:  666.50967  float64
refined_anchors          shape: (1, 4)                min:   -0.00002
  max:  432.99994  float32
```

In [33]:
```
# Display positive anchors before refinement (dotted) and
# after refinement (solid).
visualize.draw_boxes(image, boxes=positive_anchors, refined_boxes=ref
ined_anchors, ax=get_ax())
```



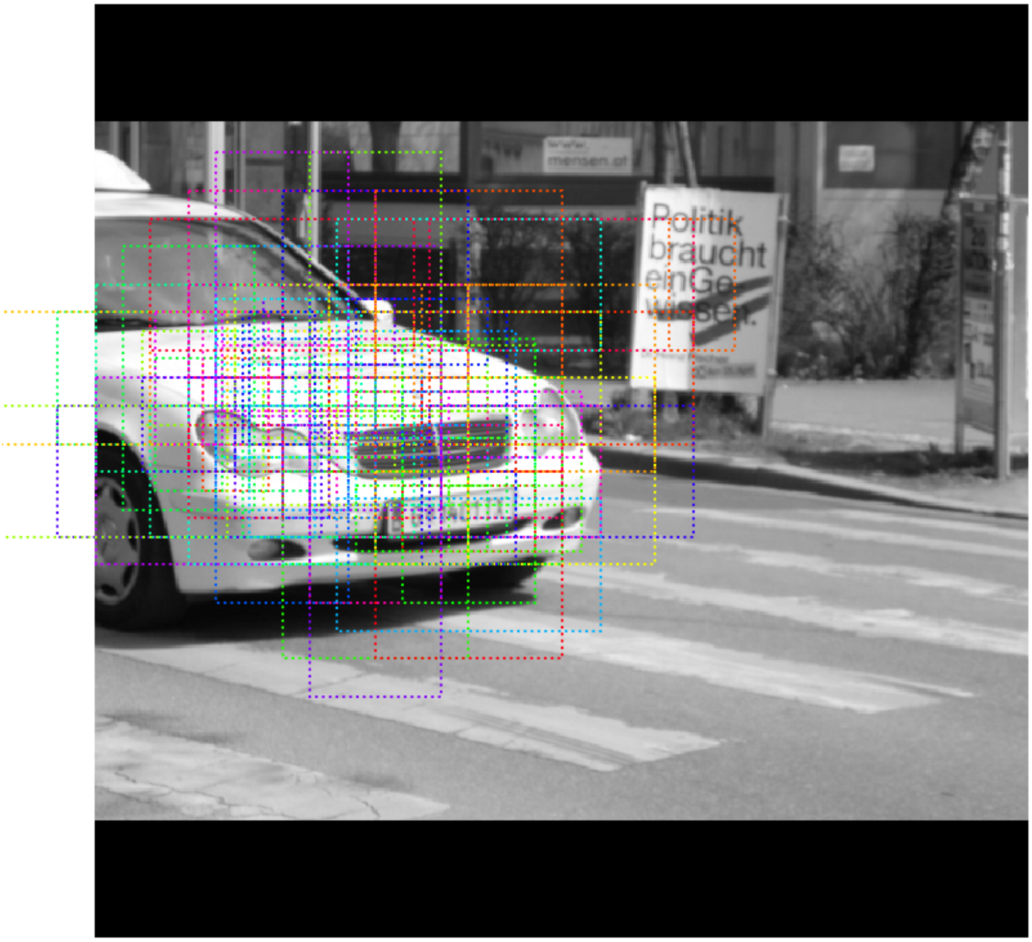
1.b RPN Predictions

Here we run the RPN graph and display its predictions.

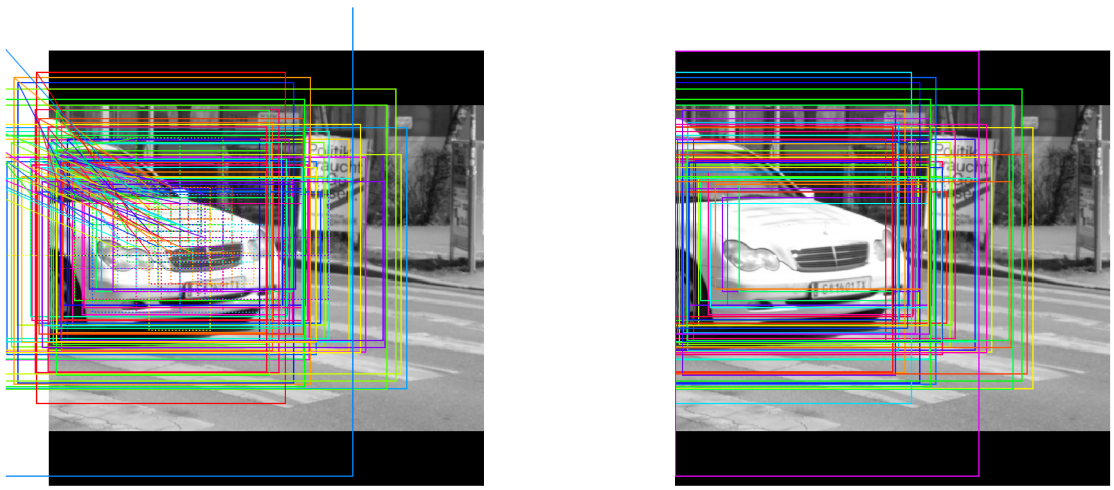
In [34]:
```
# Run RPN sub-graph
pillar = model.keras_model.get_layer("ROI").output  # node to start s
earching from

# 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/No
nMaxSuppressionV2:0")
if nms_node is None: #TF 1.9-1.10
    nms_node = model.ancestor(pillar, "ROI/rpn_non_max_suppression/No
nMaxSuppressionV3:0")

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_a
nchors_clipped:0")),
    ("post_nms_anchor_ix", nms_node),
    ("proposals", model.keras_model.get_layer("ROI").output),
])
```

```
rpn_class                shape: (1, 102300, 2)        min:    0.00000
  max:    1.00000  float32
pre_nms_anchors          shape: (1, 6000, 4)          min:   -0.14164
  max:    1.04149  float32
refined_anchors          shape: (1, 6000, 4)          min:  -87.76527
  max:   88.22453  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: 2062.00000  int32
proposals                shape: (1, 1000, 4)          min:    0.00000
  max:    1.00000  float32
```

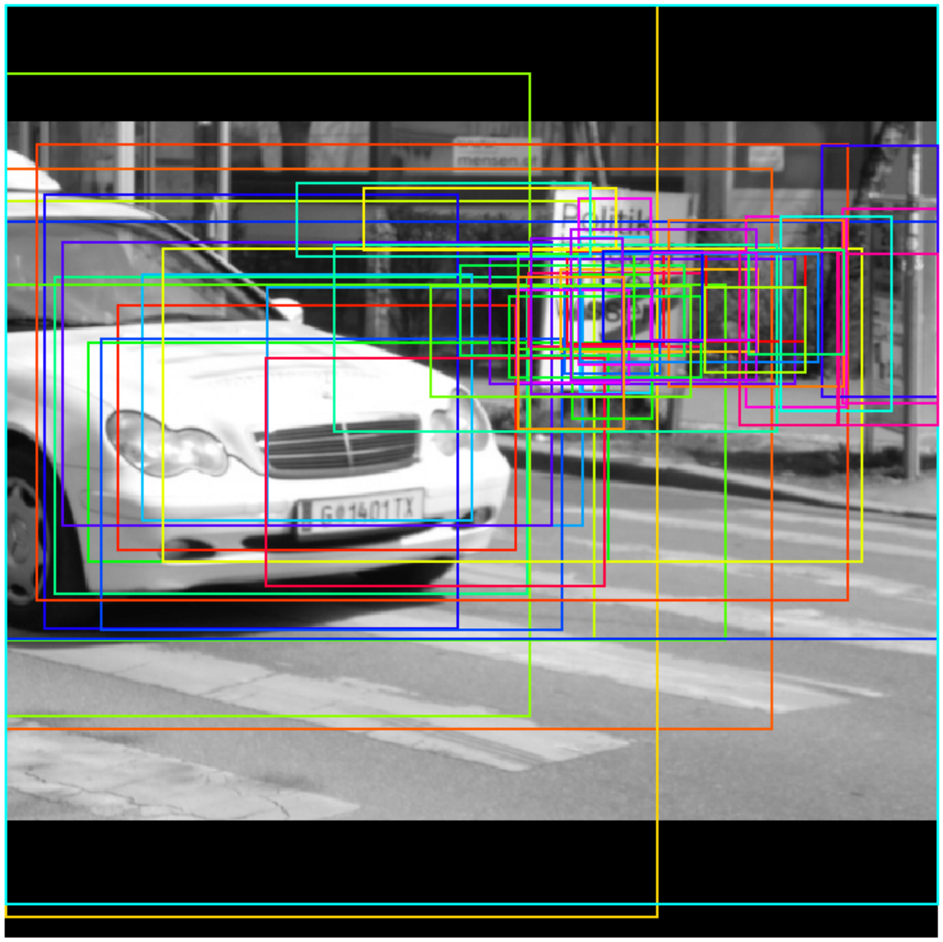
In [35]:
```
# Show top anchors by score (before refinement)
limit = 100
sorted_anchor_ids = np.argsort(rpn['rpn_class'][:,:,1].flatten())[::-1]
visualize.draw_boxes(image, boxes=model.anchors[sorted_anchor_ids[:limit]], ax=get_ax())
```



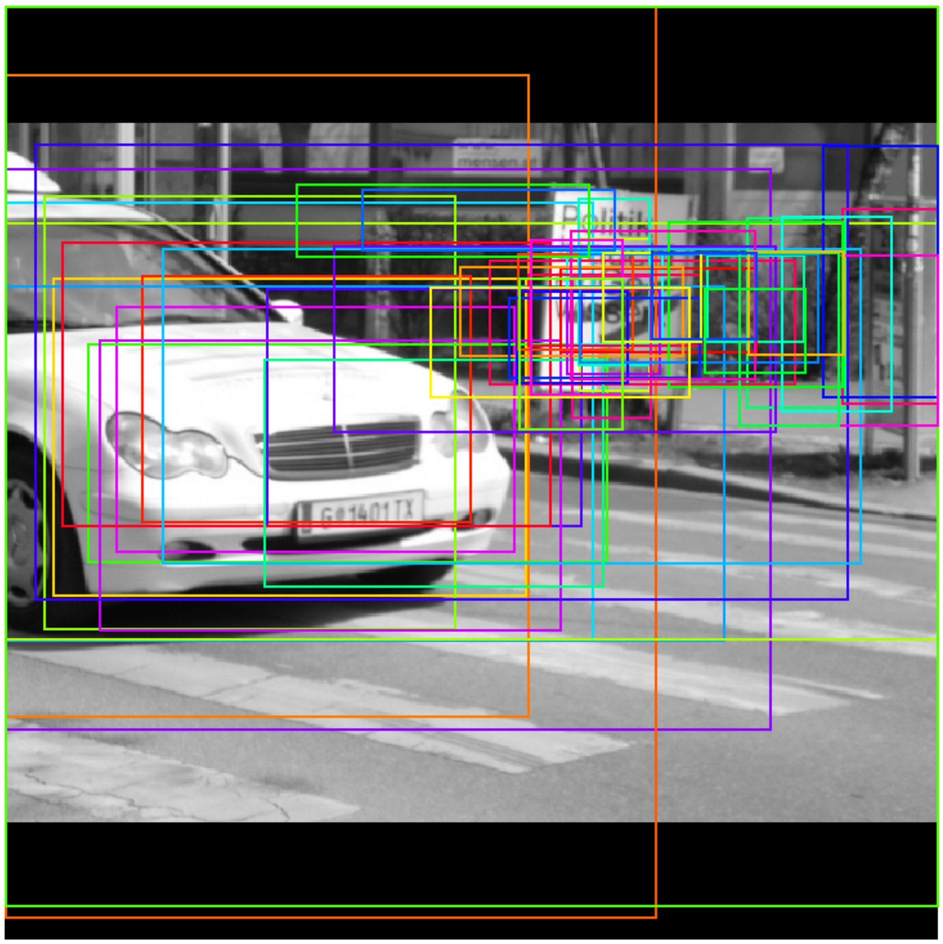
In [36]:
```
# Show top anchors with refinement. Then with clipping to image bound
aries
limit = 50
ax = get_ax(1, 2)
pre_nms_anchors = utils.denorm_boxes(rpn["pre_nms_anchors"][0], image
.shape[:2])
refined_anchors = utils.denorm_boxes(rpn["refined_anchors"][0], image
.shape[:2])
refined_anchors_clipped = utils.denorm_boxes(rpn["refined_anchors_cli
pped"][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[:li
mit], ax=ax[1])
```



In [37]:
```
# 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())
```



In [38]:
```
# 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())
```



In [39]:
```
# Measure the RPN recall (percent of objects covered by anchors)
# Here we measure recall for 3 different methods:
# - All anchors
# - All refined anchors
# - Refined anchors after NMS
iou_threshold = 0.7

recall, positive_anchor_ids = utils.compute_recall(model.anchors, gt_bbox, iou_threshold)
print("All Anchors ({:5})       Recall: {:.3f}  Positive anchors: {}".format(
    model.anchors.shape[0], recall, len(positive_anchor_ids)))

recall, positive_anchor_ids = utils.compute_recall(rpn['refined_anchors'][0], gt_bbox, iou_threshold)
print("Refined Anchors ({:5})   Recall: {:.3f}  Positive anchors: {}".format(
    rpn['refined_anchors'].shape[1], recall, len(positive_anchor_ids)))

recall, positive_anchor_ids = utils.compute_recall(proposals, gt_bbox, iou_threshold)
print("Post NMS Anchors ({:5})  Recall: {:.3f}  Positive anchors: {}".format(
    proposals.shape[0], recall, len(positive_anchor_ids)))
```

```
All Anchors (102300)       Recall: 0.000  Positive anchors: 0
Refined Anchors ( 6000)   Recall: 0.000  Positive anchors: 0
Post NMS Anchors (   50)  Recall: 1.000  Positive anchors: 3
```

Stage 2: Proposal Classification

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

2.a Proposal Classification

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

In [40]:
```
# 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").out
put),
])
```

```
proposals                shape: (1, 1000, 4)          min:    0.00000
  max:    1.00000  float32
probs                    shape: (1, 1000, 4)          min:    0.00000
  max:    1.00000  float32
deltas                   shape: (1, 1000, 4, 4)       min:   -5.11462
  max:    4.14306  float32
masks                    shape: (1, 100, 28, 28, 4)   min:    0.00022
  max:    0.99996  float32
detections               shape: (1, 100, 6)           min:    0.00000
  max:    2.00000  float32
```

In [41]:

```
# 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: ['CAR']
```

Detections

2.c Step by Step Detection

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

In [42]:
```
# 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
25 Positive ROIs
[('BG', 975), ('CAR', 25)]
```

In [43]:
```
# 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 el
se ""
            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 Refinemen
t",
                     ax=get_ax())
```

ROIs Before Refinement

Apply Bounding Box Refinement

In [44]:
```
# 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.int
32)
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 r
andom sample
captions = ["{} {:.3f}".format(dataset.class_names[c], s) if c > 0 el
se ""
            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:   -3.38738
  max:    3.11136  float32
refined_proposals        shape: (1000, 4)            min: -191.00000
  max:  771.00000  int32
```

ROIs After Refinement

Filter Low Confidence Detections

In [45]:

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

```
Keep 25 detections:
[  0   1   2   3   4   6   7   8   9  10  12  14  15  17  57  79 160
161
 170 186 236 258 292 437 585]
```

In [46]:
```
# Remove low confidence detections
keep = np.intersect1d(keep, np.where(roi_scores >= config.DETECTION_M
IN_CONFIDENCE)[0])
print("Remove boxes below {} confidence. Keep {}:\n{}".format(
    config.DETECTION_MIN_CONFIDENCE, keep.shape[0], keep))
```

```
Remove boxes below 0.7 confidence. Keep 23:
[  0   1   2   3   4   6   7   8   9  10  12  14  15  17  57  79 160
161
 170 186 236 258 585]
```

Per-Class Non-Max Suppression

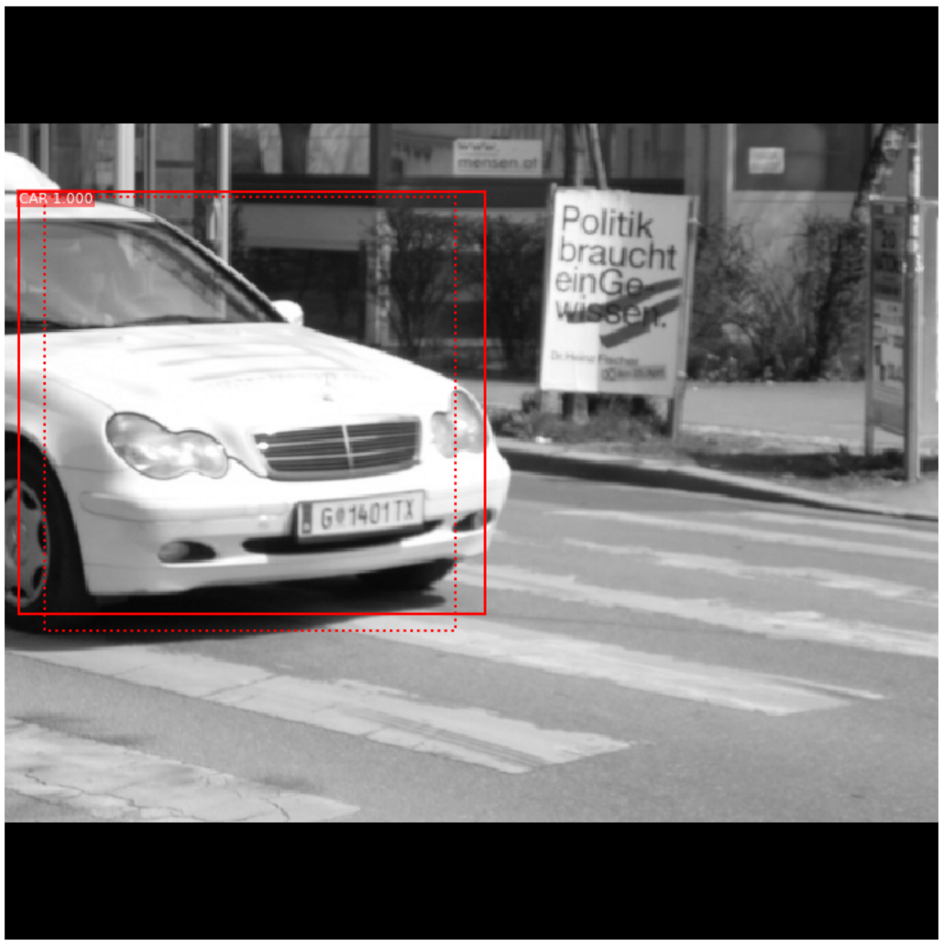
In [47]:
```
# Apply per-class non-max suppression
pre_nms_boxes = refined_proposals[keep]
pre_nms_scores = roi_scores[keep]
pre_nms_class_ids = roi_class_ids[keep]

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_THRE
SHOLD)
    # Map indicies
    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
))
```

```
CAR                   : [  0   1   2   3   4   6   7   8   9  10  12
14  15  17  57  79 160 161
 170 186 236 258 585] -> [3]

Kept after per-class NMS: 1
[3]
```

In [48]:

```
# 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



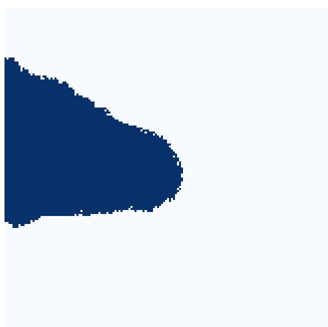
Stage 3: Generating Masks

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.

3.a Mask Targets

These are the training targets for the mask branch

In [49]:
```
display_images(np.transpose(gt_mask, [2, 0, 1]), cmap="Blues")
```



3.b Predicted Masks

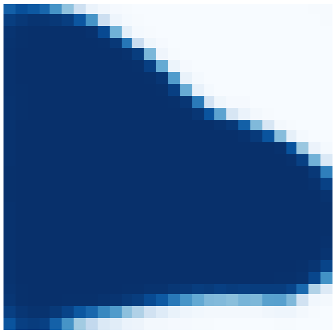
In [50]:
```
# Get predictions of mask head
mrcnn = model.run_graph([image], [
    ("detections", model.keras_model.get_layer("mrcnn_detection").out
put),
    ("masks", model.keras_model.get_layer("mrcnn_mask").output),
])

# 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:    2.00000  float32
masks                    shape: (1, 100, 28, 28, 4)   min:    0.00022
  max:    0.99996  float32
1 detections: ['CAR']
```

In [51]:
```
# Masks
det_boxes = utils.denorm_boxes(mrcnn["detections"][0, :, :4], image.s
hape[: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:    0.00022
  max:    0.99996  float32
det_masks                shape: (1, 640, 640)         min:    0.00000
  max:    1.00000  bool
```

In [52]:
```
display_images(det_mask_specific[:4] * 255, cmap="Blues", interpolati
on="none")
```



In [53]:
```
display_images(det_masks[:4] * 255, cmap="Blues", interpolation="non
e")
```

Visualize Activations

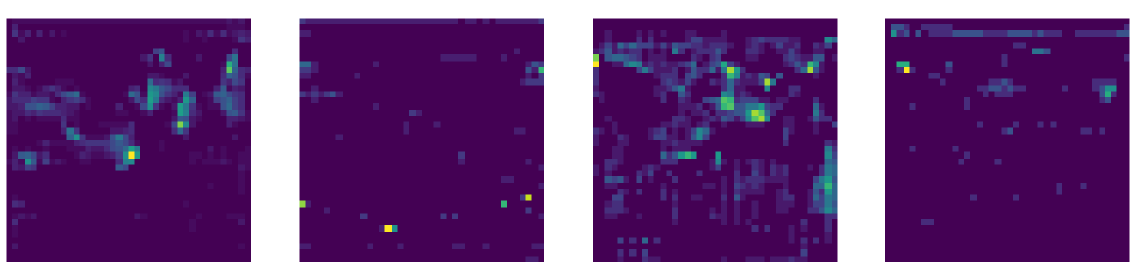
In some cases it helps to look at the output from different layers and visualize them to catch issues and odd patterns.

In [54]:
```
# Get activations of a few sample layers
activations = model.run_graph([image], [
    ("input_image",        model.keras_model.get_layer("input_image")
.output),
    ("res4w_out",          model.keras_model.get_layer("res4w_out").o
utput),  # for resnet100
    ("rpn_bbox",           model.keras_model.get_layer("rpn_bbox").ou
tput),
    ("roi",                model.keras_model.get_layer("ROI").output
),
])
```

```
input_image              shape: (1, 640, 640, 3)      min: -123.70000
  max:  151.10001  float32
res4w_out                shape: (1, 40, 40, 1024)     min:    0.00000
  max:   57.82650  float32
rpn_bbox                 shape: (1, 102300, 4)        min:  -15.51891
  max:  205.08542  float32
roi                      shape: (1, 1000, 4)          min:    0.00000
  max:    1.00000  float32
```

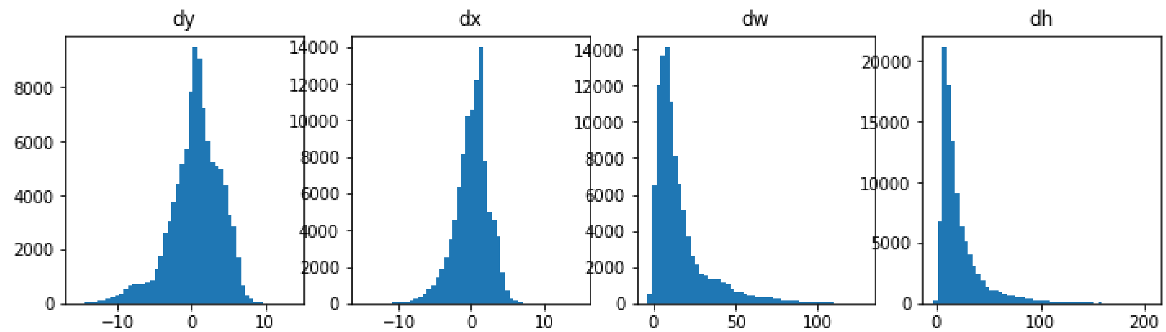
In [55]:
```
# Input image (normalized)
_ = plt.imshow(modellib.unmold_image(activations["input_image"][0],co
nfig))
```

In [56]:
```
# Backbone feature map
display_images(np.transpose(activations["res4w_out"][0,:,:,:4], [2, 0
, 1]))
```



In [57]:

```
# Histograms of RPN bounding box deltas
plt.figure(figsize=(12, 3))
plt.subplot(1, 4, 1)
plt.title("dy")
_ = plt.hist(activations["rpn_bbox"][0,:,0], 50)
plt.subplot(1, 4, 2)
plt.title("dx")
_ = plt.hist(activations["rpn_bbox"][0,:,1], 50)
plt.subplot(1, 4, 3)
plt.title("dw")
_ = plt.hist(activations["rpn_bbox"][0,:,2], 50)
plt.subplot(1, 4, 4)
plt.title("dh")
_ = plt.hist(activations["rpn_bbox"][0,:,3], 50)
```



In [58]:

```
# Distribution of y, x coordinates of generated proposals
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("y1, x1")
plt.scatter(activations["roi"][0,:,0], activations["roi"][0,:,1])
plt.subplot(1, 2, 2)
plt.title("y2, x2")
plt.scatter(activations["roi"][0,:,2], activations["roi"][0,:,3])
plt.show()
```