In [3]:
```
import os
import sys
import time
import random
import numpy as np
import imgaug
import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib
import matplotlib.patches as patches
from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval
from pycocotools import mask as maskUtils
import zipfile
import urllib.request
import json
import shutil
from PIL import Image, ImageDraw
```

In [4]:
```
# Root directory of the project
ROOT_DIR = os.path.abspath("../../../Mask_RCNN/")
# Import Mask RCNN
# Import mrcnn libraries
sys.path.append(ROOT_DIR)
from mrcnn.config import Config
import mrcnn.utils as utils
from mrcnn import visualize
import mrcnn.model as modellib
from mrcnn.model import log
```

```
Using TensorFlow backend.
```

In [5]:
```
# 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)
DEFAULT_DATASET_YEAR = "2019"
```

In [6]:
```
# dir = '../../../coco-abdomen/'
```

In [7]:
```
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
```

In [8]:
```
############################################################
#  Configurations
############################################################

class SkinConfig(Config):
    """Configuration for training on the toy  dataset.
    Derives from the base Config class and overrides some values.
    """
    # Give the configuration a recognizable name
    NAME = "skin"

    GPU_COUNT = 1
    # We use a GPU with 12GB memory, which can fit two images.
    # Adjust down if you use a smaller GPU.
    IMAGES_PER_GPU = 2

    # Number of classes (including background)
    NUM_CLASSES = 1 + 1  # Background + skin

    # Number of training steps per epoch
    STEPS_PER_EPOCH = 200

    # This is how often validation is run. If you are using too much hard d
rive space
    # on saved models (in the MODEL_DIR), try making this value larger.
    VALIDATION_STEPS = 70

    # Matterport originally used resnet101, but I downsized to fit it on my
graphics card
    BACKBONE = 'resnet50'

    # Skip detections with < 90% confidence
#     DETECTION_MIN_CONFIDENCE = 0.9
     # To be honest, I haven't taken the time to figure out what these do
    RPN_ANCHOR_SCALES = (8, 16, 32, 64, 128)
    TRAIN_ROIS_PER_IMAGE = 32
    MAX_GT_INSTANCES = 50
    POST_NMS_ROIS_INFERENCE = 500
    POST_NMS_ROIS_TRAINING = 1000


config = SkinConfig()
config.display()
```

```
Configurations:
BACKBONE                       resnet50
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                  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_bbox_loss': 1.0, 'rpn_class_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               50
MEAN_PIXEL                     [123.7 116.8 103.9]
MINI_MASK_SHAPE                (56, 56)
NAME                           skin
NUM_CLASSES                    2
POOL_SIZE                      7
POST_NMS_ROIS_INFERENCE        500
POST_NMS_ROIS_TRAINING         1000
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                200
TOP_DOWN_PYRAMID_SIZE          256
TRAIN_BN                       False
TRAIN_ROIS_PER_IMAGE           32
USE_MINI_MASK                  True
USE_RPN_ROIS                   True
VALIDATION_STEPS               70
WEIGHT_DECAY                   0.0001
```

In [9]:
```
class SkinDataset(utils.Dataset):

    def load_data(self, annotation_json, images_dir):
        """ Load the coco-like dataset from json
        Args:
            annotation_json: The path to the coco annotations json file
            images_dir: The directory holding the images referred to by the
json file
        """
        # Load json from file
        json_file = open(annotation_json)
        coco_json = json.load(json_file)
        json_file.close()

        # Add the class names using the base method from utils.Dataset
        source_name = "skin"
        for category in coco_json['categories']:
            class_id = category['id']
            class_name = category['name']
#             print(class_name)

            if class_id < 1:
                print('Error: Class id for "{}" cannot be less than one. (0
is reserved for the background)'.format(class_name))
                return

            self.add_class(source_name, class_id, class_name)
#             print('hi')

        # Get all annotations
        annotations = {}
        for annotation in coco_json['annotations']:
            image_id = annotation['image_id']
#             print(image_id)

            if image_id not in annotations:
                annotations[image_id] = []
            annotations[image_id].append(annotation)
#             print(annotations)


        # Get all images and add them to the dataset
        seen_images = {}
        for image in coco_json['images']:
            image_id = image['id']
            if image_id in seen_images:
                print("Warning: Skipping duplicate image id: {}".format(ima
ge))
            else:
                seen_images[image_id] = image
                try:
                    image_file_name = image['file_name']
                    image_width = image['width']
                    image_height = image['height']
                except KeyError as key:
                    print("Warning: Skipping image (id: {}) with missing ke
y: {}".format(image_id, key))

                image_path = os.path.abspath(os.path.join(images_dir, image
_file_name))
                image_annotations = annotations[image_id]

                # Add the image using the base method from utils.Dataset
```

Creating Training and Validation Data

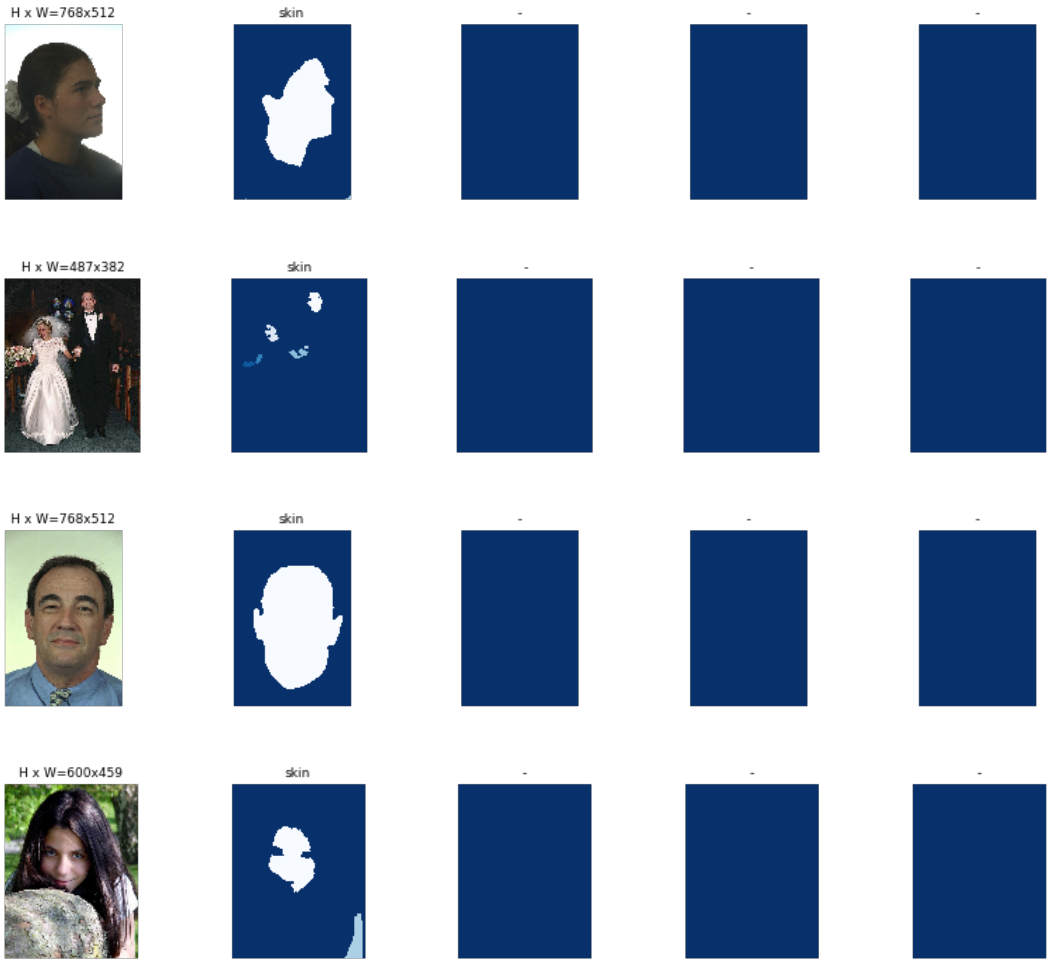
In [10]:
```
dataset_train = SkinDataset()
dataset_train.load_data('../../../coco-overall/train/instances_skin_train2019.json','../../../coco-overall/train/skin_train2019/')
dataset_train.prepare()
```

In [11]:
```
dataset_val = SkinDataset()
dataset_val.load_data('../../../coco-overall/val/instances_skin_val2019.json','../../../coco-overall/val/skin_val2019/')
dataset_val.prepare()
```

In [12]:
```
dataset_test = SkinDataset()
dataset_test.load_data('../../../test/instances_skin_test2019.json','../../../test/images/')
dataset_test.prepare()
```

Display a few images from the training dataset¶

In [13]:
```
dataset = dataset_train
image_ids = np.random.choice(dataset.image_ids, 4)

for image_id in image_ids:
    image = dataset.load_image(image_id)
    mask, class_ids = dataset.load_mask(image_id)
    visualize.display_top_masks(image, mask, class_ids, dataset.class_names
)
```



Create the Training Model and Train¶

In [14]:
```
# # Create model in training mode
# model = modellib.MaskRCNN(mode="training", config=config,
#                           model_dir=MODEL_DIR)
```

In [15]:
```
# model.
```

In [16]:
```
# # Which weights to start with?
# init_with = "last"  # imagenet, coco, or last

# if init_with == "imagenet":
#     model.load_weights(model.get_imagenet_weights(), by_name=True)
# elif init_with == "coco":
#     # Load weights trained on MS COCO, but skip layers that
#     # are different due to the different number of classes
#     # See README for instructions to download the COCO weights
#     model.load_weights(COCO_MODEL_PATH, by_name=True,
#                        exclude=["mrcnn_class_logits", "mrcnn_bbox_fc",
#                                 "mrcnn_bbox", "mrcnn_mask"])
# elif init_with == "last":
#     # Load the last model you trained and continue training
#     model.load_weights(model.find_last(), by_name=True)
```

Training

Train in two stages:

- Only the heads. Here we're freezing all the backbone layers and training only the randomly initialized layers (i.e. the ones that we didn't use pre-trained weights from MS COCO). To train only the head layers, pass layers='heads' to the train() function.
- Fine-tune all layers. For this simple example it's not necessary, but we're including it to show the process. Simply pass layers="all to train all layers.

In [17]:
```
# Train the head branches
# Passing layers="heads" freezes all layers except the head
# layers. You can also pass a regular expression to select
# which layers to train by name pattern.
start_train = time.time()
# model.train(dataset_train, dataset_val,
#             learning_rate=config.LEARNING_RATE,
#             epochs=4,
#             layers='heads')
end_train = time.time()
minutes = round((end_train - start_train) / 60, 2)
print('Training took' + str(minutes)+ 'minutes')
```

```
Training took0.0minutes
```

In [18]:
```
# Fine tune all layers
# Passing layers="all" trains all layers. You can also
# pass a regular expression to select which layers to
# train by name pattern.
start_train = time.time()
# model.train(dataset_train, dataset_val,
#             learning_rate=config.LEARNING_RATE / 10,
#             epochs=8,
#             layers="all")
end_train = time.time()
minutes = round((end_train - start_train) / 60, 2)
print('Training took' + str(minutes)+ 'minutes')
```

```
Training took0.0minutes
```

Prepare to run Inference

In [19]:
```
class InferenceConfig(SkinConfig):
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    IMAGE_MIN_DIM = 512
    IMAGE_MAX_DIM = 512
    DETECTION_MIN_CONFIDENCE = 0.85


inference_config = InferenceConfig()
```

In [20]:
```
# Recreate the model in inference mode
model = modellib.MaskRCNN(mode="inference",
                          config=inference_config,
                          model_dir=MODEL_DIR)
```

In [21]:
```
# Get path to saved weights
# Either set a specific path or find last trained weights
# model_path = os.path.join(ROOT_DIR, ".h5 file name here")
# model_path = model.find_last()
model_path = '/home/anirudh/detect/code/Umbilicus_Skin_Detection/Mask_RCNN/
logs/onlyabdomen/mask_rcnn_skin_0008.h5'
# Load trained weights (fill in path to trained weights here)
assert model_path != "", "Provide path to trained weights"
print("Loading weights from ", model_path)
model.load_weights(model_path, by_name=True)
```

```
Loading weights from  /home/anirudh/detect/code/Umbilicus_Skin_Detection/Ma
sk_RCNN/logs/onlyabdomen/mask_rcnn_skin_0008.h5
```

In [22]:
```
dataset_test.image_ids
```

Out[22]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27])
```

In [32]:
```
# dataset_test.load_data()
```

```
---------------------------------------------------
TypeError      Traceback (most recent call last)
<ipython-input-32-83f4fe85c611> in <module>
----> 1 dataset_test.load_data()

TypeError: load_data() missing 2 required positional arguments: 'annotation
_json' and 'images_dir'
```

In [50]:
```
plt.imshow(r['masks'][:,:,0])
```

```
---------------------------------------------------
IndexError     Traceback (most recent call last)
<ipython-input-50-2b78084bbcd1> in <module>
----> 1 plt.imshow(r['masks'][:,:,0])

IndexError: index 0 is out of bounds for axis 2 with size 0
```

In [60]:
```
r['rois'].shape[0]
```

Out[60]:
```
0
```

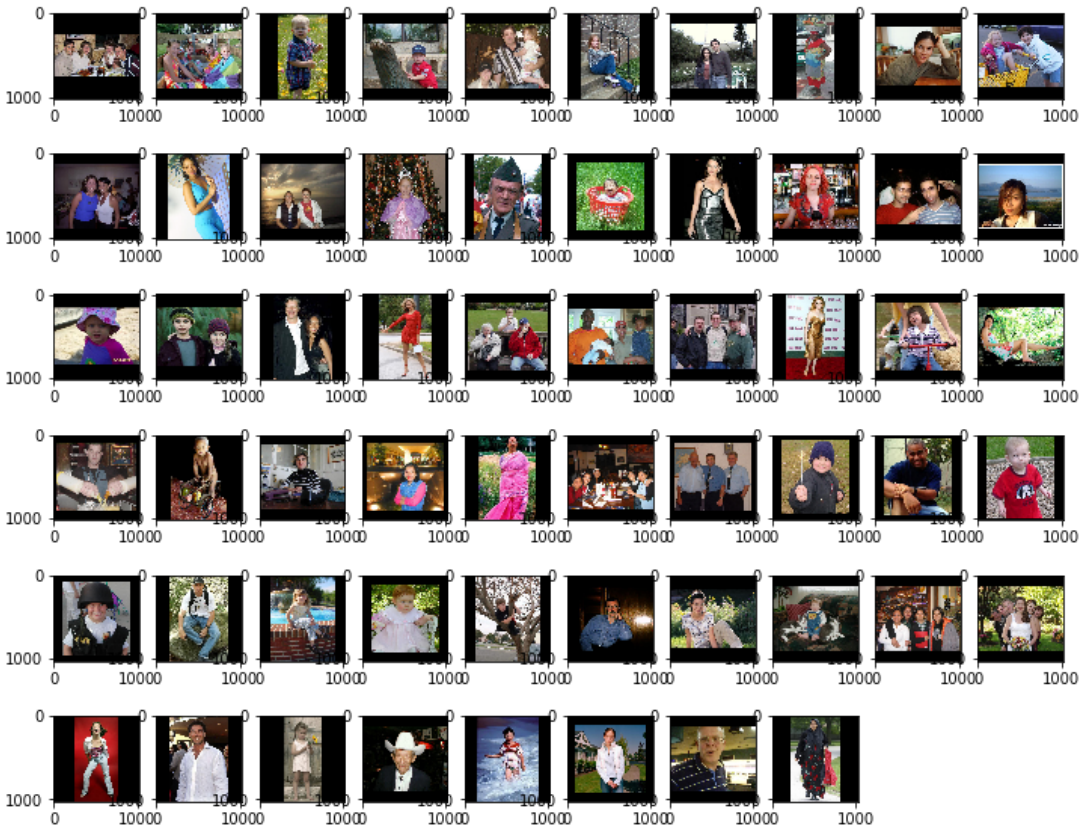
Run Inference

In [24]:
```
# Compute VOC-style Average Precision
# def compute_batch_ap(image_ids):
# image_ids = np.random.choice(dataset_val.image_ids, 28)
APs = []
accu=[]
c=0
imges = []
for image_id in dataset_val.image_ids:
    # Load image
    image, image_meta, gt_class_id, gt_bbox, gt_mask = modellib.load_image_
gt(dataset_val, config,
                                   image_id, use_mini_mask=False)
#     plt.figure()
#     plt.imshow(image)

    # Run object detection
    results = model.detect([image], verbose=0)
    r = results[0]
#     maskpre= visualize.display_instances(image, r['rois'], r['masks'], r[
'class_ids'],
#                                  dataset_test.class_names, r['scores'], fi
gsize=(5,5))
    if (r['rois'].shape[0]==0):
        c= c+1
        imges.append(image)
        continue
    maskpre=  r['masks'][:,:,0]
    acc = np.sum(gt_mask[:,:,0] == maskpre)/np.size(maskpre)
    print(acc)
    accu.append(acc)
#     plt.close()

print("mean accu = "+str(np.mean(accu)))
```

```
0.8883228302001953
0.9312505722045898
0.9929161071777344
0.9454460144042969
0.8897294998168945
0.821864128112793
0.8097152709960938
0.9730072021484375
0.9880037307739258
0.8778619766235352
0.9923906326293945
0.9144134521484375
0.9080715179443359
0.9802742004394531
0.8490505218505859
0.8486423492431641
0.9292898178100586
0.963109016418457
0.9892597198486328
0.8811836242675781
0.9814777374267578
0.8068208694458008
0.9274349212646484
0.8980464935302734
0.8304567337036133
0.6074733734130859
0.9220190048217773
0.976902961730957
0.9047889709472656
0.8497686386108398
0.9731283187866211
0.9250898361206055
0.82269287109375
0.9533119201660156
0.9834051132202148
0.9512138366699219
0.9718723297119141
0.942631721496582
0.8298835754394531
0.8323049545288086
0.8611631393432617
0.896824836730957
0.967376708984375
0.6966009140014648
0.878779411315918
0.9485712051391602
0.8484916687011719
0.9260082244873047
0.7780046463012695
0.7840280532836914
0.9734859466552734
0.8833217620849609
0.8671731948852539
0.9741239547729492
0.956303596496582
0.9822959899902344
0.9785060882568359
0.9574403762817383
0.6609439849853516
0.6231555938720703
0.8986806869506836
0.9669580459594727
0.8764801025390625
```

In [ ]:
```
for image in imges:

    plt.imshow
```

In [30]:
```
plt.figure(figsize=(12,10))# Showing the Input Data after Normalizing
x, y = 10, 6
for i in range(c):
    plt.subplot(y, x, i+1)
    plt.imshow(imges[i],interpolation='nearest')
plt.show()
```



In [ ]:
```
np.sum(gt_mask[:,:,0] == mask)/np.size(mask)
```

In [27]:
```
imges[0].shape
```

Out[27]:
```
(1024, 1024, 3)
```

In [ ]:
```
for image_id in image_ids:
    image1 = dataset_test.load_image(image_id)
    results = model.detect([image1], verbose=0)
    masked_image = image.astype(np.uint32).copy()
    colors = visualize.random_colors(5)
    y1, x1, y2, x2 = results[0]['masks']['boxes'][1]
#     msk = visualize.apply_mask(image1,,'w')
#     plt.imshow(visulaize.apply_mask(image1,results[0]['masks'],'r'))
    break
```

In [ ]:
```
image_ids
```

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

In [ ]:
```
dataset
```

In [ ]:
```
# 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_BBOX_STD
_DEV)
log("refined_anchors", refined_anchors, )
```

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

# TF 1.4 and 1.9 introduce new versions of NMS. Search for all names to sup
port 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/NonMaxSu
ppressionV2:0")
if nms_node is None: #TF 1.9-1.10
    nms_node = model.ancestor(pillar, "ROI/rpn_non_max_suppression/NonMaxSu
ppressionV3:0")

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),
])
```

In [ ]:
```
# 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])
```

In [ ]:
```
# 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 [ ]:
```
# 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 [ ]:
```
# 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: {}".forma
t(
    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: {}".forma
t(
    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: {}".forma
t(
    proposals.shape[0], recall, len(positive_anchor_ids)))
```

In [ ]:
```

```

In [ ]:
```

```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: