In [16]:
```
import os
import time
import random
import collections

import numpy as np
import pandas as pd
from PIL import Image
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

import torch
import torchvision
from torchvision.transforms import ToPILImage
from torchvision.transforms import functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
from torchvision.models.detection.mask_rcnn import MaskRCNNPredictor
```

In [17]:
```
TRAIN_CSV = "train.csv"
TRAIN_PATH = "train"
TEST_PATH = "test"

WIDTH = 704
HEIGHT = 520
TEST = False

DEVICE = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

RESNET_MEAN = (0.485, 0.456, 0.406)
RESNET_STD = (0.229, 0.224, 0.225)

BATCH_SIZE = 2

# No changes tried with the optimizer yet.
MOMENTUM = 0.9
LEARNING_RATE = 0.001
WEIGHT_DECAY = 0.0005

# Changes the confidence required for a pixel to be kept for a mask.
# Only used 0.5 till now.
MASK_THRESHOLD = 0.5

# Normalize to resnet mean and std if True.
NORMALIZE = False


# Use a StepLR scheduler if True. Not tried yet.
USE_SCHEDULER = False

# Amount of epochs
NUM_EPOCHS = 8


BOX_DETECTIONS_PER_IMG = 539


MIN_SCORE = 0.59
```

In [18]:
```
class Compose:
    def __init__(self, transforms):
```

```
        self.transforms = transforms

    def __call__(self, image, target):
        for t in self.transforms:
            image, target = t(image, target)
        return image, target

class VerticalFlip:
    def __init__(self, prob):
        self.prob = prob

    def __call__(self, image, target):
        if random.random() < self.prob:
            height, width = image.shape[-2:]
            image = image.flip(-2)
            bbox = target["boxes"]
            bbox[:, [1, 3]] = height - bbox[:, [3, 1]]
            target["boxes"] = bbox
            target["masks"] = target["masks"].flip(-2)
        return image, target

class HorizontalFlip:
    def __init__(self, prob):
        self.prob = prob

    def __call__(self, image, target):
        if random.random() < self.prob:
            height, width = image.shape[-2:]
            image = image.flip(-1)
            bbox = target["boxes"]
            bbox[:, [0, 2]] = width - bbox[:, [2, 0]]
            target["boxes"] = bbox
            target["masks"] = target["masks"].flip(-1)
        return image, target

class Normalize:
    def __call__(self, image, target):
        image = F.normalize(image, RESNET_MEAN, RESNET_STD)
        return image, target

class ToTensor:
    def __call__(self, image, target):
        image = F.to_tensor(image)
        return image, target


def get_transform(train):
    transforms = [ToTensor()]
    if NORMALIZE:
        transforms.append(Normalize())

    # Data augmentation for train
    if train:
        transforms.append(HorizontalFlip(0.5))
        transforms.append(VerticalFlip(0.5))

    return Compose(transforms)
```

In [19]:

```
def rle_decode(mask_rle, shape, color=1):
    '''
    mask_rle: run-length as string formated (start length)
    shape: (height,width) of array to return
    Returns numpy array, 1 - mask, 0 - background
```

```
    '''
    s = mask_rle.split()
    starts, lengths = [np.asarray(x, dtype=int) for x in (s[0:][::2], s[1:][::2])]
    starts -= 1
    ends = starts + lengths
    img = np.zeros(shape[0] * shape[1], dtype=np.float32)
    for lo, hi in zip(starts, ends):
        img[lo : hi] = color
    return img.reshape(shape)
```

In [20]:

```
class CellDataset(Dataset):
    def __init__(self, image_dir, df, transforms=None, resize=False):
        self.transforms = transforms
        self.image_dir = image_dir
        self.df = df

        self.should_resize = resize is not False
        if self.should_resize:
            self.height = int(HEIGHT * resize)
            self.width = int(WIDTH * resize)
        else:
            self.height = HEIGHT
            self.width = WIDTH

        self.image_info = collections.defaultdict(dict)
        temp_df = self.df.groupby('id')['annotation'].agg(lambda x: list(x)).reset_inc
        for index, row in temp_df.iterrows():
            self.image_info[index] = {
                    'image_id': row['id'],
                    'image_path': os.path.join(self.image_dir, row['id'] + '.png'),
                    'annotations': row["annotation"]
                    }

    def get_box(self, a_mask):
        ''' Get the bounding box of a given mask '''
        pos = np.where(a_mask)
        xmin = np.min(pos[1])
        xmax = np.max(pos[1])
        ymin = np.min(pos[0])
        ymax = np.max(pos[0])
        return [xmin, ymin, xmax, ymax]

    def __getitem__(self, idx):
        ''' Get the image and the target'''

        img_path = self.image_info[idx]["image_path"]
        img = Image.open(img_path).convert("RGB")

        if self.should_resize:
            img = img.resize((self.width, self.height), resample=Image.BILINEAR)

        info = self.image_info[idx]

        n_objects = len(info['annotations'])
        masks = np.zeros((len(info['annotations']), self.height, self.width), dtype=np
        boxes = []

        for i, annotation in enumerate(info['annotations']):
            a_mask = rle_decode(annotation, (HEIGHT, WIDTH))
            a_mask = Image.fromarray(a_mask)

            if self.should_resize:
                a_mask = a_mask.resize((self.width, self.height), resample=Image.BILIN
```

```
            a_mask = np.array(a_mask) > 0
            masks[i, :, :] = a_mask

            boxes.append(self.get_box(a_mask))

        # dummy labels
        labels = [1 for _ in range(n_objects)]

        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.as_tensor(labels, dtype=torch.int64)
        masks = torch.as_tensor(masks, dtype=torch.uint8)

        image_id = torch.tensor([idx])
        area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0])
        iscrowd = torch.zeros((n_objects,), dtype=torch.int64)

        # This is the required target for the Mask R-CNN
        target = {
            'boxes': boxes,
            'labels': labels,
            'masks': masks,
            'image_id': image_id,
            'area': area,
            'iscrowd': iscrowd
        }

        if self.transforms is not None:
            img, target = self.transforms(img, target)

        return img, target

    def __len__(self):
        return len(self.image_info)
```

In [21]:
```
df_train = pd.read_csv(TRAIN_CSV, nrows=5000 if TEST else None)
ds_train = CellDataset(TRAIN_PATH, df_train, resize=False, transforms=get_transform(tr
dl_train = DataLoader(ds_train, batch_size=BATCH_SIZE, shuffle=True,
                      num_workers=2, collate_fn=lambda x: tuple(zip(*x)))
```

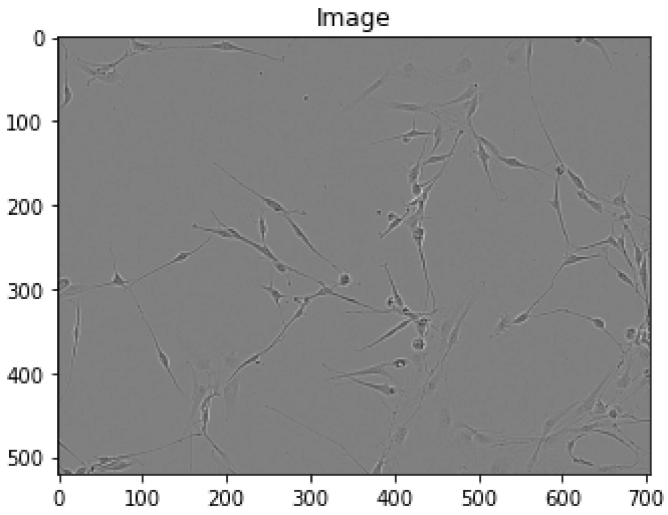
In [22]:
```
def get_model():
    # This is just a dummy value for the classification head
    NUM_CLASSES = 2

    if NORMALIZE:
        model = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True,
                                                                   box_detections_per_
                                                                   image_mean=RESNET_M
                                                                   image_std=RESNET_ST
    else:
        model = torchvision.models.detection.maskrcnn_resnet50_fpn(pretrained=True,
                                                                  box_detections_per_i

    # get the number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    # replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, NUM_CLASSES)

    # now get the number of input features for the mask classifier
    in_features_mask = model.roi_heads.mask_predictor.conv5_mask.in_channels
    hidden_layer = 256
    # and replace the mask predictor with a new one
    model.roi_heads.mask_predictor = MaskRCNNPredictor(in_features_mask, hidden_layer,
```

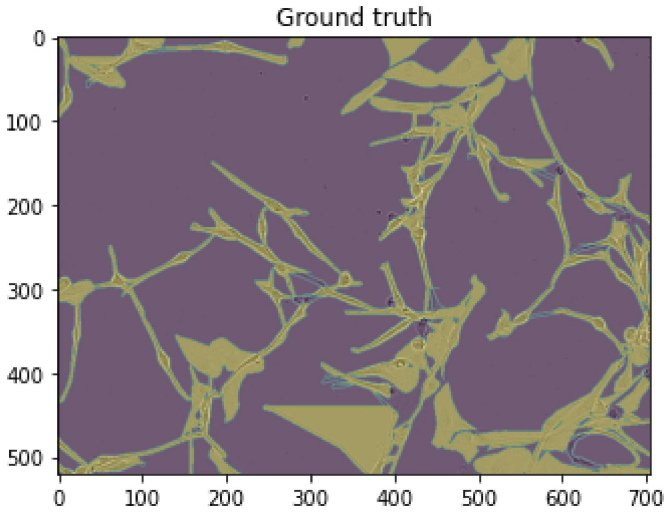
```
    return model


# Get the Mask R-CNN model
# The model does classification, bounding boxes and MASKs for individuals, all at the
# We only care about MASKS
model = get_model()
model.to(DEVICE)

# TODO: try removing this for
for param in model.parameters():
    param.requires_grad = True

model.train();
```

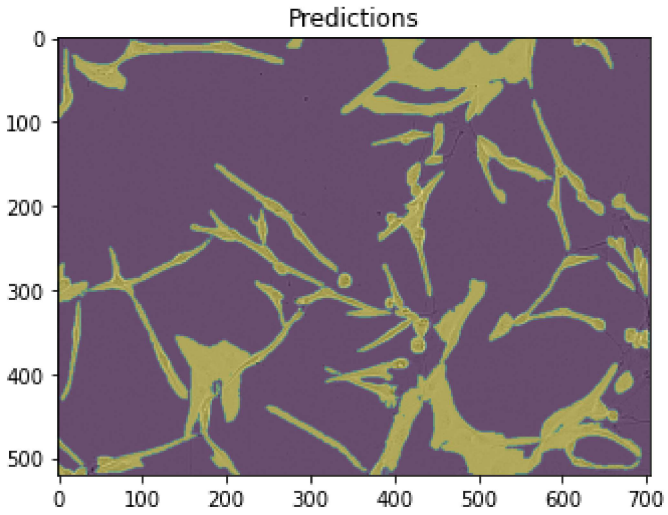
In [23]:
```
#import matplotlib.image as mpimg
#test_image = mpimg.imread("7ae19de7bc2a.png")
#plt.imshow(test_image)
#plt.show()
```

In [24]:
```
def analyze_train_sample(model, ds_train, sample_index):

    img, targets = ds_train[sample_index]
    plt.imshow(img.numpy().transpose((1,2,0)))
    plt.title("Image")
    plt.show()

    masks = np.zeros((HEIGHT, WIDTH))
    for mask in targets['masks']:
        masks = np.logical_or(masks, mask)
    plt.imshow(img.numpy().transpose((1,2,0)))
    plt.imshow(masks, alpha=0.3)
    plt.title("Ground truth")
    plt.show()

    model.eval()
    with torch.no_grad():
        preds = model([img.to(DEVICE)])[0]

    plt.imshow(img.cpu().numpy().transpose((1,2,0)))
    all_preds_masks = np.zeros((HEIGHT, WIDTH))
    for mask in preds['masks'].cpu().detach().numpy():
        all_preds_masks = np.logical_or(all_preds_masks, mask[0] > MASK_THRESHOLD)
    plt.imshow(all_preds_masks, alpha=0.4)
    plt.title("Predictions")
    plt.show()
```

In [36]:
```
device = torch.device('cpu')
model=get_model()
model.load_state_dict(torch.load('pytorch_model-e8back.bin',map_location=device))
analyze_train_sample(model, ds_train, 1)
```







In [45]:
```
class CellTestDataset(Dataset):
    def __init__(self, image_dir, transforms=None):
        self.transforms = transforms
        self.image_dir = image_dir
        self.image_ids = [f[:-4]for f in os.listdir(self.image_dir)]

    def __getitem__(self, idx):
        image_id = self.image_ids[idx]
        image_path = os.path.join(self.image_dir, image_id + '.png')
        image = Image.open(image_path).convert("RGB")

        if self.transforms is not None:
            image, _ = self.transforms(image=image, target=None)
```

```
        return {'image': image, 'image_id': image_id}

    def __len__(self):
        return len(self.image_ids)
```

In [47]:
```
ds_test = CellTestDataset(TEST_PATH, transforms=get_transform(train=False))
ds_test[0]
```

Out[47]:
```
{'image': tensor([[[0.5020, 0.5020, 0.4980,  ..., 0.4980, 0.4941, 0.4902],
         [0.5020, 0.5020, 0.4941,  ..., 0.5020, 0.5020, 0.5020],
         [0.5098, 0.5098, 0.5020,  ..., 0.5137, 0.5020, 0.5137],
         ...,
         [0.4980, 0.4980, 0.4980,  ..., 0.5333, 0.5216, 0.5176],
         [0.5098, 0.5098, 0.5098,  ..., 0.5373, 0.5451, 0.5373],
         [0.4941, 0.5059, 0.5020,  ..., 0.5686, 0.5569, 0.5490]],

        [[0.5020, 0.5020, 0.4980,  ..., 0.4980, 0.4941, 0.4902],
         [0.5020, 0.5020, 0.4941,  ..., 0.5020, 0.5020, 0.5020],
         [0.5098, 0.5098, 0.5020,  ..., 0.5137, 0.5020, 0.5137],
         ...,
         [0.4980, 0.4980, 0.4980,  ..., 0.5333, 0.5216, 0.5176],
         [0.5098, 0.5098, 0.5098,  ..., 0.5373, 0.5451, 0.5373],
         [0.4941, 0.5059, 0.5020,  ..., 0.5686, 0.5569, 0.5490]],

        [[0.5020, 0.5020, 0.4980,  ..., 0.4980, 0.4941, 0.4902],
         [0.5020, 0.5020, 0.4941,  ..., 0.5020, 0.5020, 0.5020],
         [0.5098, 0.5098, 0.5020,  ..., 0.5137, 0.5020, 0.5137],
         ...,
         [0.4980, 0.4980, 0.4980,  ..., 0.5333, 0.5216, 0.5176],
         [0.5098, 0.5098, 0.5098,  ..., 0.5373, 0.5451, 0.5373],
         [0.4941, 0.5059, 0.5020,  ..., 0.5686, 0.5569, 0.5490]]]),
 'image_id': '7ae19de7bc2a'}
```

In [49]:
```
def rle_encoding(x):
    dots = np.where(x.flatten() == 1)[0]
    run_lengths = []
    prev = -2
    for b in dots:
        if (b>prev+1): run_lengths.extend((b + 1, 0))
        run_lengths[-1] += 1
        prev = b
    return ' '.join(map(str, run_lengths))


def remove_overlapping_pixels(mask, other_masks):
    for other_mask in other_masks:
        if np.sum(np.logical_and(mask, other_mask)) > 0:
            mask[np.logical_and(mask, other_mask)] = 0
    return mask
```

In [50]:
```
model.eval();

submission = []
for sample in ds_test:
    img = sample['image']
    image_id = sample['image_id']
    with torch.no_grad():
        result = model([img.to(DEVICE)])[0]

    previous_masks = []
    for i, mask in enumerate(result["masks"]):

        # Filter-out low-scoring results. Not tried yet.
        score = result["scores"][i].cpu().item()
        if score < MIN_SCORE:
```

```
            continue

        mask = mask.cpu().numpy()
        # Keep only highly likely pixels
        binary_mask = mask > MASK_THRESHOLD
        binary_mask = remove_overlapping_pixels(binary_mask, previous_masks)
        previous_masks.append(binary_mask)
        rle = rle_encoding(binary_mask)
        submission.append((image_id, rle))

    # Add empty prediction if no RLE was generated for this image
    all_images_ids = [image_id for image_id, rle in submission]
    if image_id not in all_images_ids:
        submission.append((image_id, ""))

df_sub = pd.DataFrame(submission, columns=['id', 'predicted'])
df_sub.head()
```

Out[50]:

	id	predicted
0	7ae19de7bc2a	355113 7 355815 9 356518 11 357221 12 357923 1...
1	7ae19de7bc2a	275 4 978 8 1682 10 2387 12 3092 14 3797 15 45...
2	7ae19de7bc2a	139298 10 139309 1 140001 21 140705 23 141409 ...
3	7ae19de7bc2a	1855 8 2559 9 3262 11 3966 12 4670 12 5374 12 ...
4	7ae19de7bc2a	109839 5 110541 7 111244 8 111946 10 112649 10...