# Semantic Segmentation of Cells

In this notebook, I will be exploring and testing my model for segmenting images by cells. This will serve as the starting point for **Instance Segmentation** and then **Graph Construction**

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageOps
import os
import sys
from pathlib import Path
from math import inf
from tqdm import tqdm

from torchvision.transforms import Compose, ToPILImage
import torch

top_folder = str(Path(os.getcwd()).parent.parent)
sys.path.append(top_folder)

os.environ['CUDA_LAUNCH_BLOCKING'] = "1"

%matplotlib widget
```

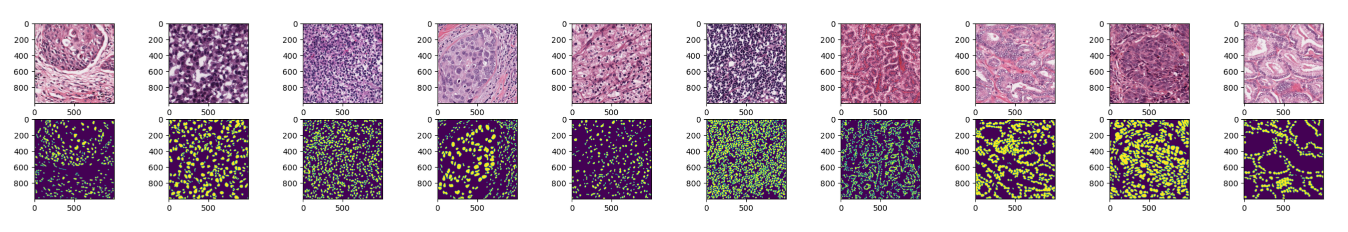
## Data Preperation

In [ ]:
```
# swap color axis because
# numpy image: H x W x C
# torch image: C x H x W

def tensor_to_numpy(tensor):
    return np.asarray(ToPILImage()(tensor))
```

### No Transforms

In [ ]:
```
from src.datasets.MoNuSeg import MoNuSeg
from torch.utils.data import DataLoader
from src.transforms.MoNuSeg import ToTensor

dl = DataLoader(MoNuSeg(os.path.join(top_folder,"data","processed","MoNuSeg_TRAIN"),transform=ToTensor()), batch_size=1, shuffle=True, num_workers=

f,ax = plt.subplots(2,10,figsize=(30,4))

for i,img in enumerate(dl):
    ax[0,i].imshow(tensor_to_numpy(img['image'].squeeze())) #as is batch 1 need to squeeze
    ax[1,i].imshow(tensor_to_numpy(img['semantic_mask'].squeeze()))
    if i==9:
        plt.show()
        break
```

Figure



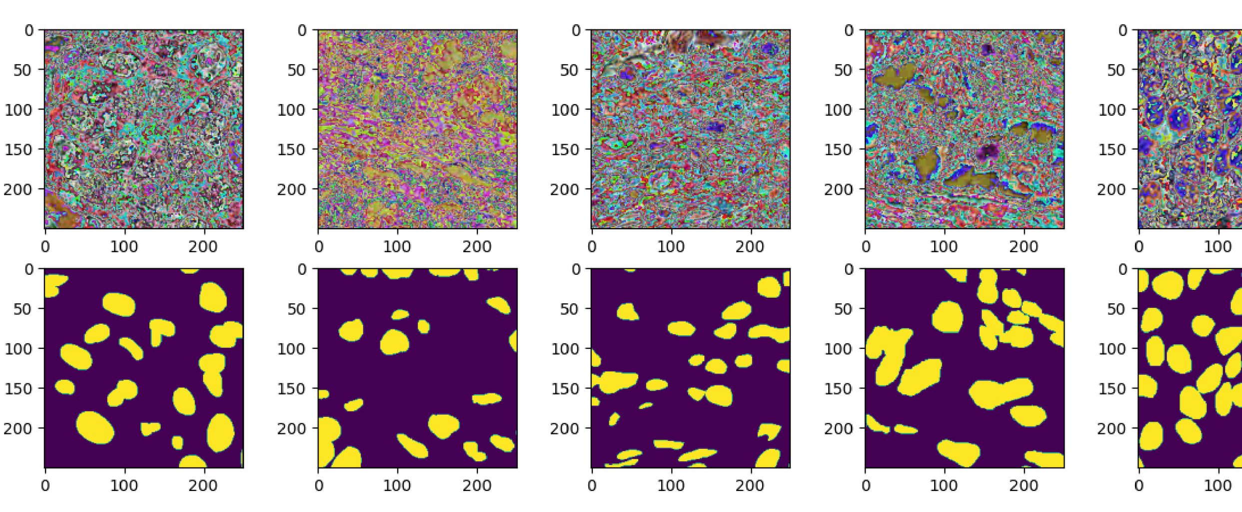
In [ ]:
```
# Figure out the normalization terms

mean_tensor,std_tensor = torch.zeros(3),torch.zeros(3)
for img in dl:
    he = img['image'].squeeze().float() #need to convert to float as is otherwise a byte (8bit depth)
    std_mean = torch.std_mean(he,dim=(1,2))
    std_tensor += std_mean[0]
    mean_tensor += std_mean[1]
mean_tensor = torch.div(mean_tensor,len(dl))
std_tensor = torch.div(std_tensor,len(dl))

print(f"The mean colour of the image is: {mean_tensor}")
print(f"The standard deviation of the images is: {std_tensor}")
```

```
The mean colour of the image is: tensor([0.6441, 0.4474, 0.6039])
The standard deviation of the images is: tensor([0.1892, 0.1922, 0.1535])
```

### Experimenting Transforms

In [ ]:
```
from src.transforms.MoNuSeg import Normalize,ToTensor,RandomCrop
from random import random

size = (250,250)
transforms = Compose([ToTensor(),Normalize([0.6441, 0.4474, 0.6039],[0.1892, 0.1922, 0.1535]),RandomCrop(size=size)])

dl_trans = DataLoader(MoNuSeg(os.path.join(top_folder,"data","processed","MoNuSeg_TRAIN"),transform=transforms), batch_size=1, shuffle=True, num_wo

f,ax = plt.subplots(2,5,figsize=(15,5))
for i,batch in enumerate(dl_trans):
    img = batch['image'].squeeze()
    mask = batch['semantic_mask'].squeeze()
    ax[0,i].imshow(tensor_to_numpy(img))
    ax[1,i].imshow(tensor_to_numpy(mask))
    if i==4:
        break
plt.show()
```

Figure



# Model

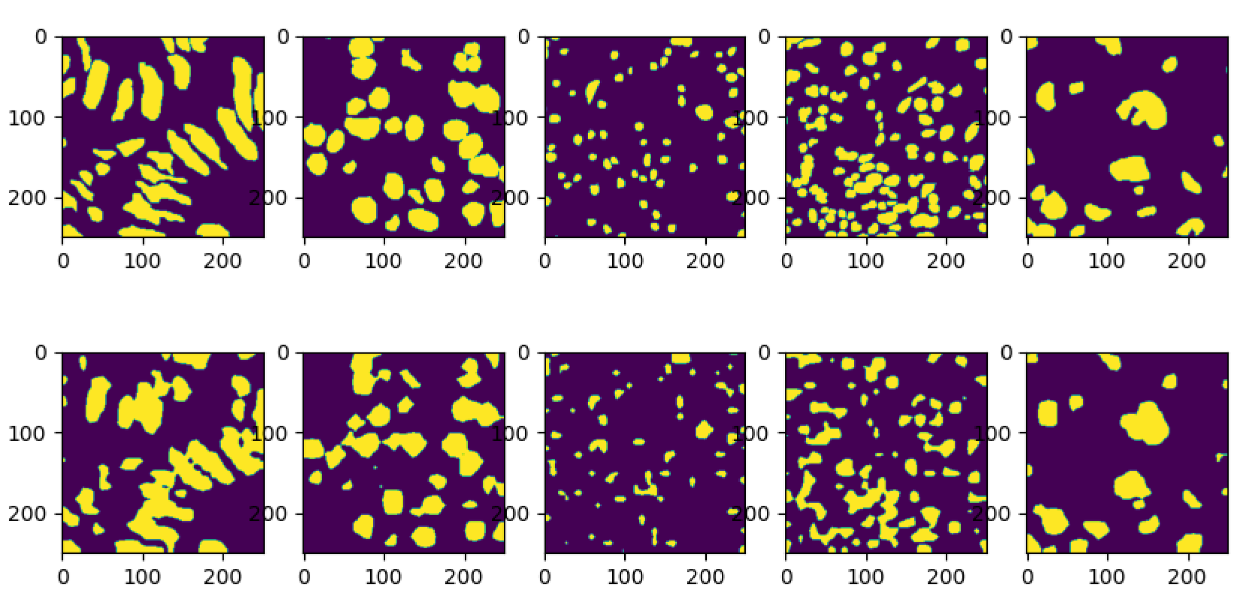
## Pretrained and built

In [ ]:
```
import mlflow

model = mlflow.pytorch.load_model("../../trained_models/cell_seg_v1.pth").cpu()
```

## Predict

In [ ]:
```
f, ax = plt.subplots(2,5,figsize=(10,5))
for i,batch in enumerate(dl_trans):
    if i ==5:
        break
    pred = model(batch['image']).detach().numpy().squeeze()
    pred[pred>0.5] = 1
    pred[pred<=0.5] = 0
    ax[0,i].imshow(batch['semantic_mask'].squeeze())
    ax[1,i].imshow(pred)
```

Figure



## Observations

If you train the model on 250x250 crops, then try to run it on the full 1000x1000 in one go, it fails horribly (detects nothing). My feeling is that this is because of size and that the smaller images of cells don't quite have the same shape dimensions when enlarged (i.e. curves a lot less slowly)

- [ ] Train on random sizes, zooms and rotates

Below you can see how a simple, pretrained model finetuned to this data set performs for not very long. On the top is the ground truth, while at the bottom is the predicted mask. There is signed of undersegmentation (cells are grouped together which should be consider distinct). I have some ideas of how to seperate these instances.

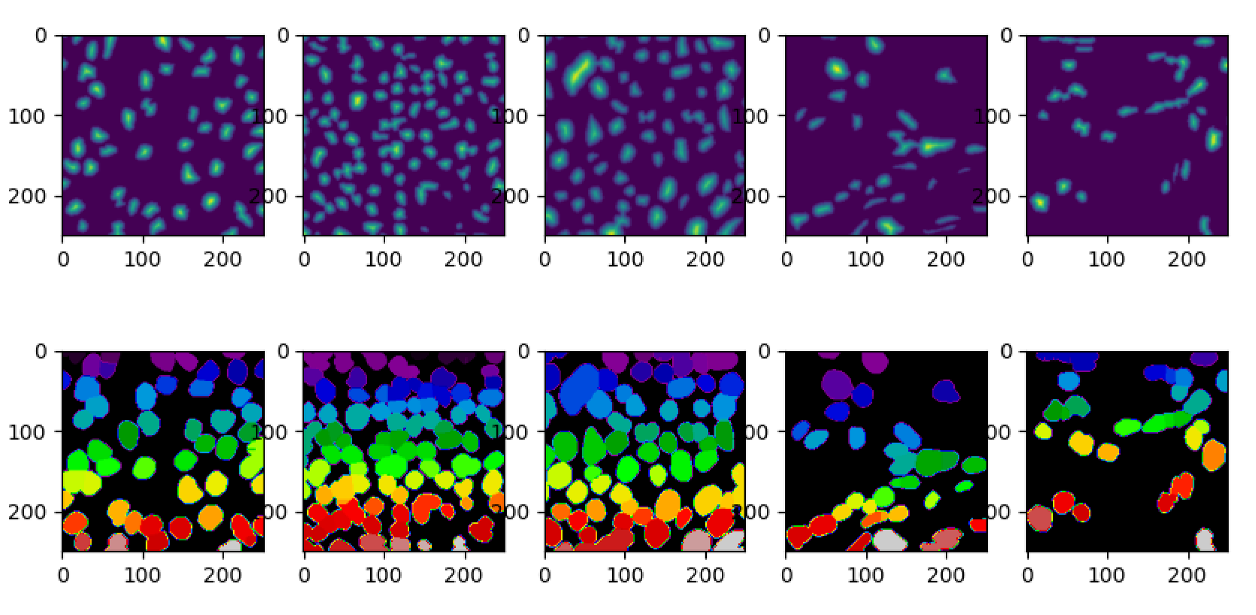
There are also some weird artifacts that are being produced:

# Instance Segmentation

In [ ]:
```
from src.model.graph_construction.cell_instance_segmentation import instance_segment
from scipy.ndimage import distance_transform_edt
#todo fix

f, ax = plt.subplots(2,5,figsize=(10,5))
for i,batch in enumerate(dl_trans):
    if i ==5:
        break
    #pred = model(batch['image']).detach().int().numpy().squeeze()
    distance = distance_transform_edt(batch['semantic_mask'].squeeze())
    ax[0,i].imshow(distance)
    ax[1,i].imshow(instance_segment(batch['semantic_mask'].squeeze().int().numpy(),peak_seperation=10,min_rel_threshold=0),cmap=plt.cm.nipy_spectra
    #ax[0,i].imshow(instance_segment(batch['semantic_mask'].squeeze().int().numpy()),cmap=plt.cm.nipy_spectral)
   # ax[1,i].imshow(pred,cmap=plt.cm.nipy_spectral)
plt.show()
```

```
c:\Users\aless\Documents\git\XAI-Cancer-Diagnosis\src\model\graph_construction\cell_instance_segmentation.py:22: FutureWarning: indices argument is deprecated and will be removed in version 0.20. To avoid this warning, please do not use the indices argument. Please see peak_local_max documentat
ion for more details.
  mask = peak_local_max(distance, footprint=np.ones((5, 5)), labels=image,
```

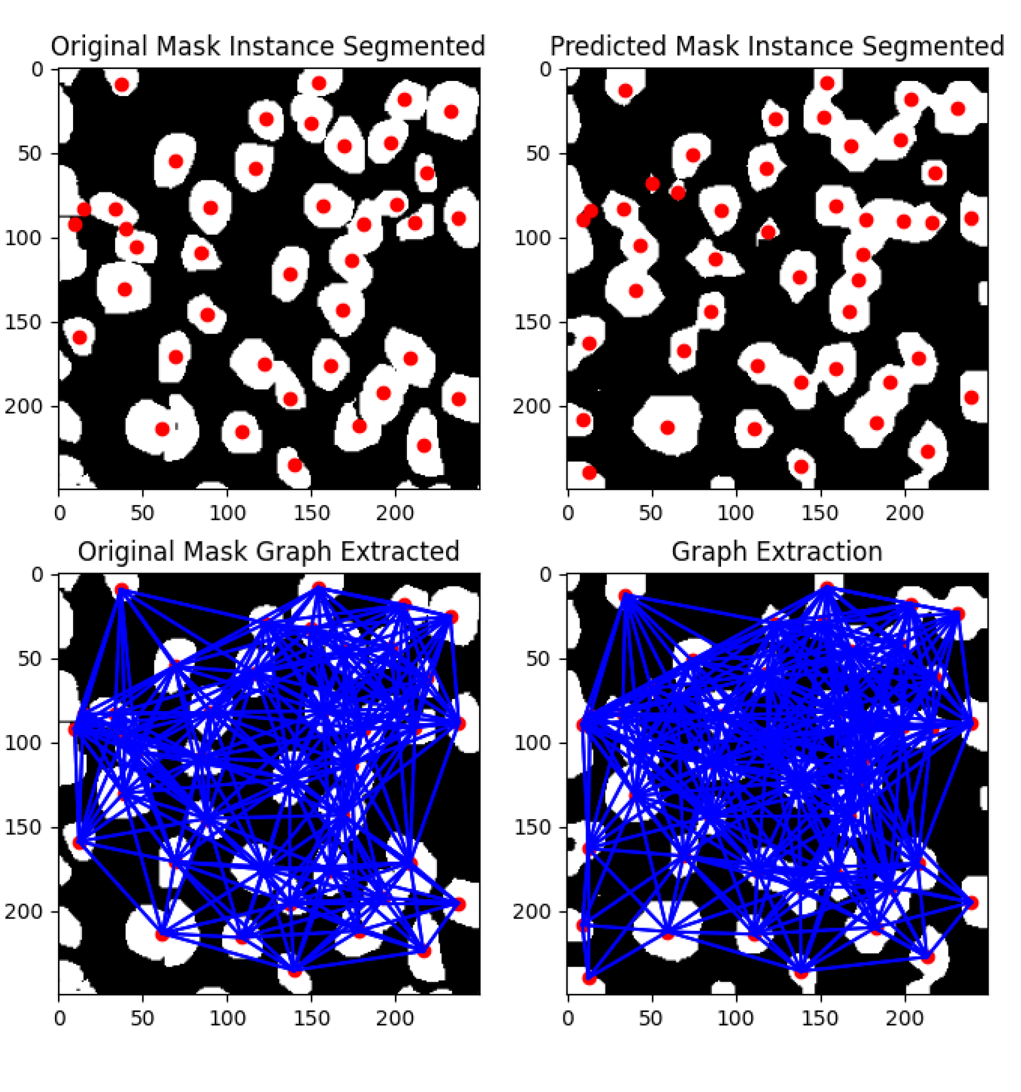
Figure



In [ ]:
```
batch = list(dl_trans)[0]
```

In [ ]:
```
from src.model.graph_construction.graph_extraction import create_featureless_graph
from src.vizualizations.graph_viz import show_graph


from scipy import ndimage

orig_mask = batch['semantic_mask'].squeeze().int().numpy()
pred_mask = model.predict(batch['image']).squeeze().int().numpy()




ins_seg_orig_mask = instance_segment(orig_mask,peak_seperation=12,min_rel_threshold=0.3)
graph_orig = create_featureless_graph(ins_seg_orig_mask,dist_threshold=100) # as stated in histopathology paper
#points_orig = list(map(lambda x:x['centre'],graph_orig.nodes.values()))
#x_orig,y_orig = map(list,zip(*points_orig))


ins_seg_pred_mask = instance_segment(pred_mask,peak_seperation=10,min_rel_threshold=0.2)
graph_pred = create_featureless_graph(ins_seg_pred_mask,dist_threshold=100)
#points_pred = list(map(lambda x:x['centre'],graph_pred.nodes.values()))
#x_pred,y_pred = map(list,zip(*points_pred))




f,ax = plt.subplots(2,2,figsize=(8,8))



ax[0,0].imshow(1-orig_mask,cmap="binary")
show_graph(graph_orig,ax[0,0],with_edges=False)
ax[0,0].set_title("Original Mask Instance Segmented")

ax[0,1].imshow(1-pred_mask,cmap="binary")
show_graph(graph_pred,ax[0,1],with_edges=False)
ax[0,1].set_title("Predicted Mask Instance Segmented")


ax[1,0].imshow(1-orig_mask,cmap="binary")
show_graph(graph_orig,ax[1,0],with_edges=True)
ax[1,0].set_title("Original Mask Graph Extracted")


ax[1,1].imshow(1-pred_mask,cmap="binary")
show_graph(graph_pred,ax[1,1],with_edges=True)
ax[1,1].set_title("Predicted Mask Graph Extracted")


plt.show()
```

Figure




### Observations

Does a good job for round nuclei. Nuclei that are thin and elongated don't really have a **centre** but instead have a **ridge**. This poses a problem for watershedding as ridges when inverted tend to have lots of local minima. This is why these cells tend to be oversegmented. This has been partially alleviated by using `peak_distance` to ignore close peaks. This "smoothing" comes at the cost of more **undersegmentation** so a balance needs to be striked.

ROUND = GOOD LONG = BAD (is it circles superimposed or just thin?)

**IDEA** - check for pertuberations

### Sparse cells:

Below is what happens when we have a region that has not too dense: