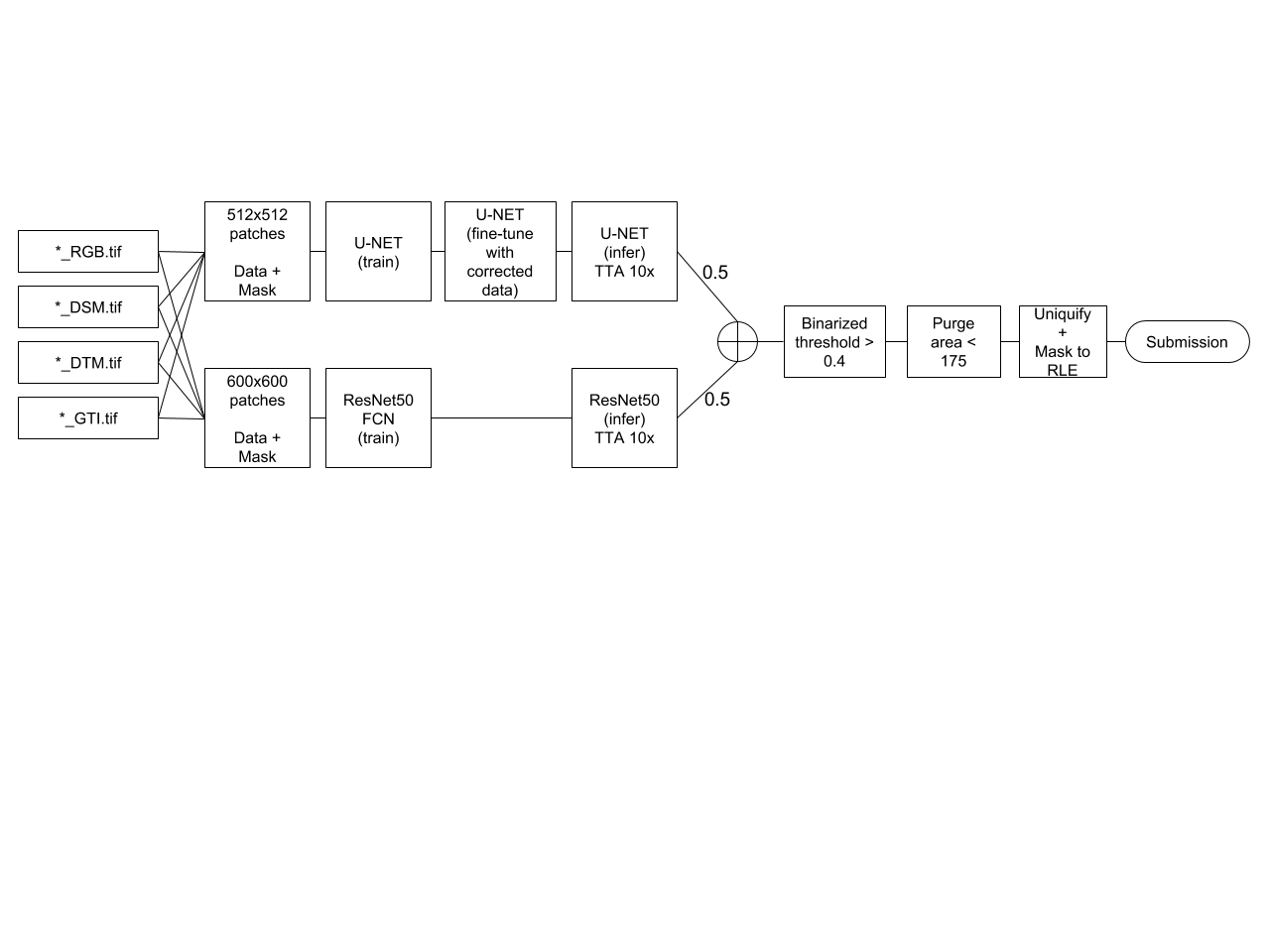
**Marathon Match - UrbanMapper3D - 5th Place Solution Description (‘kylelee’)**

**Overview**

For this particular problem I used an ensemble of a U-NET @ 512x512 and a ResNet-50 FCN @ 600x600 - each with 4 channels where 3 channels were the standard RGB and the 4th channel was the difference between DSM and DTM. For the ResNet-50 I modified weights from the standard pretrained model to account for the 4th channel in the input. Additional post-processing was done including threshold specific binarization and purging contours with small areas. Instance level annotation was done by simply looping contours using OpenCV and labeling each unique contour.



*Figure 1. Solution pipeline for 5th place in UrbanMapper3D*

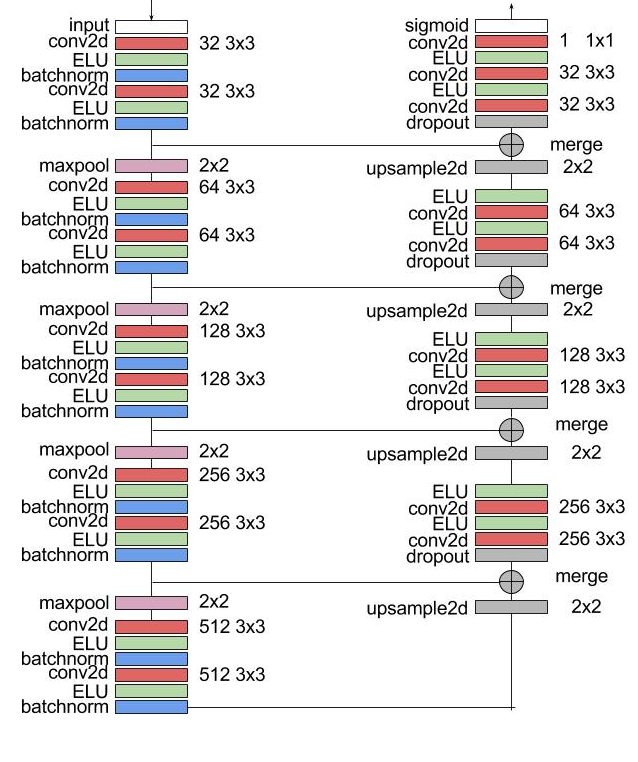
1. **Introduction**

* *Name:* Kyle Yen-Khai Lee
* *Handle:* kylelee
* *Placement you achieved in the MM:* 5th
* *About you:* I have professional experience in both hardware design (ASIC/circuits) and in data science. I have also participated and done well in a number of computer vision contests, including a few semantic segmentation and satellite imagery competitions.
* *Why did you participate in the MM?* Since there is a depth component (DSM/DTM) to this problem I wanted to see if my existing segmentation methodology used in other contests could easily be extended to this solution. This is also my first foray into Topcoder data science competitions.

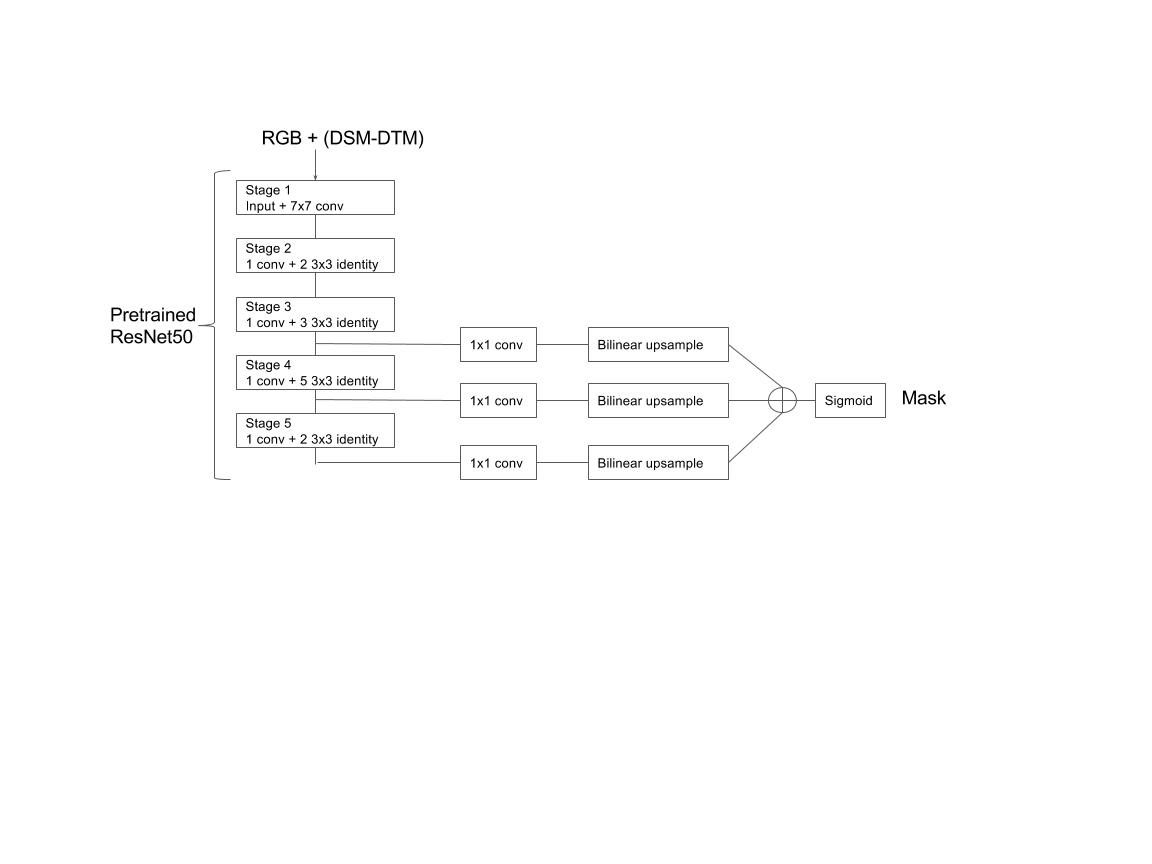
1. **Solution Development**

*How did you solve the problem? What approaches did you try and what choices did you make, and why? Also, what alternative approaches did you consider?*

* Architectures: I used a combination of the U-NET and ResNet50 FCN and treated this problem as initially a semantic segmentation problem, and only uniquify the instances at the end. The links for the original architectures are listed here while the implemented architectures are shown below:
  + U-NET: <https://arxiv.org/abs/1505.04597>
  + ResNet50: <https://arxiv.org/abs/1512.03385>

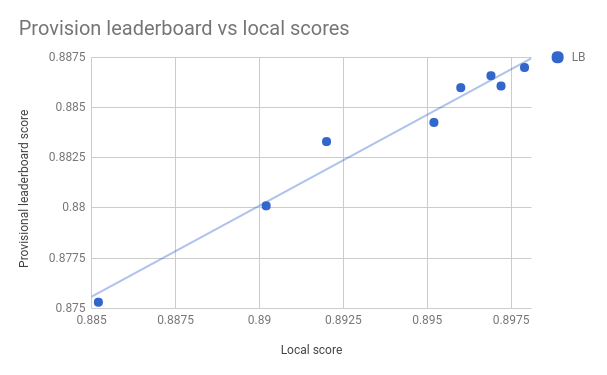


*Figure 2. U-NET architecture (with batch-normalization, ELU, and dropout)*



*Figure 3. ResNet50 FCN with 1x1 convolutions and upsampling*

* DSM/DTM: Initially, I tried out 5 channels where both DSM and DTM were separate channels in addition to RGB (where in the U-NET case I mean/standard normalized each channel). This did not work well and after further consideration, I used the delta of DSM and DTM as the only additional channel. This turned out to give qualitatively better results and I kept the channel (however I did not quantify the difference with and without this addition).
* Patching: I started out using 256x256 patches but upon further experimentation, larger patches showed a significant improvement in both local validation and provisional leaderboard. For example, for an equivalent run using the same setup and validation images a 512x512 run showed around +10,000 higher F1-score than a 256x256 run. My guess is that contextually the networks are able to visualize building-building relationships better at a larger scale.
* Data correction / fine-tuning (U-NET): Upon further review of the data I noticed that some masks enclosed both valid and invalid (darkened) regions near edges, so I had another round of patching to fix this by cutting off the ground truth masks for training mask, and fine-tuned the first U-NET step (per the two U-NET train steps in Figure 1) again. Even though provisional leaderboard and local score did not really improve I kept this step for U-NET only (but did not get around fine-tuning for ResNet50).
* Validation: I kept only around ~10% of images for local scoring (the rest used for training). I implemented patches on these validation images for inference and ensembling but used the full images for local scoring. The provisional leaderboard to local score trend appeared to be directionally consistent and did not really overfit so I used ratio this for all tuning experiments.

  
*Figure 4. Provisional leaderboard vs local scores*

1. **Final Approach**

*Please provide a bulleted description of your final approach. What ideas/decisions/features have been found to be the most important for your solution performance:*

* Adding a DSM-DTM channel: As mentioned in section 2 (Solution Development), including the difference of DSM and DTM as a separate input channel was qualitatively important for the network to segment areas where the RGB images may have been occluded.
* Larger scale patches: As mentioned in section 2 (Solution Development), using larger patch sizes provide a better context for building detection and almost a +10,000 score difference versus small patch sizes. I used a 512x512 patch for the U-NET and 600x600 patch for the ResNet50, where the 512x512 patches had steps of 256 with overlaps while the 600x600 patches had steps of 200 with overlaps. I intentionally used a slightly different patch size for ResNet50 in order to improve diversity.
* Ensemble of U-NET and ResNet50: I used a 50/50 average of both the U-NET and ResNet50 predictions. The local validation score for each was 892,600 (U-NET) and 891,600 (ResNet50) respectively, while their ensemble was 897,900. Note that the additional fine-tuning step for corrected data for the U-NET was kept in the final solution even though it may be redundant (this is described in “Solution Development”).
* Pre-trained model for ResNet50: I leveraged on the pretrained weights for a standard ResNet50 and transferred them to a 4-channel input version by copying RGB weights + the DSM-DTM weight as equivalent to the red channel (refer to src/pretrained/transfer\_weights\_res50\_4ch.py to see how this is done). I did not quantify the difference but traditionally this is much better than random initialization even for a non-RGB scenario.
* Loss functions for U-NET/ResNet50 training: I adopted a combination of binary cross-entropy and dice coefficient for the loss function in both architectures, which has shown better performance than just one or the other in many other competitions. This is represented in code by:

|  |
| --- |
| *K.binary\_crossentropy(y\_true,y\_pred)-K.log(dice\_coef(y\_true,y\_pred))* |

* Test-time cropping: In order to avoid local boundary effects I used crops on inferenced images. Specifically, for the 512x512 patches only center crops of 256x256 were used, while for 600x600 patches only center crops of 200x200 were used.
* Test-time augmentation: During test time, 10x averages of +/-90 degree rotations and horizontal/vertical flips were used. The combinations are as follows:
  + Default (no rotation, no flips)
  + Horizontal flips only
  + Vertical flips only
  + Horizontal+vertical flips
  + +90 degree rotation
  + +90 degree rotation with horizontal flips
  + +90 degree rotation with vertical flips
  + -90 degree rotation
  + -90 degree rotation with horizontal flips
  + -90 degree rotation with vertical flips
* Post-processing - binarization: A threshold of 0.4 was used for binarizing the mask after averaging both predictions. This was derived from both local validation and the provisional leaderboard.
* Post-processing - area filtering/purging: Contour areas which were below 175 were purged. Again, this was derived from both the provisional leaderboard, local validation scores, and looking at the statistics of building areas in the entire training set.
* Post-processing - uniquification: Finally, in order to convert the semantic segmentation problem to an instance segmentation one (which is the goal of the problem), I simply used a contour search in OpenCV to loop through all contour instances and iterated them to represent instances for the final RLE mask.

1. **Open Source Resources, Frameworks and Libraries**

*Please specify the name of the open source resource along with a URL to where it’s housed and it’s license type:*

* Docker, <https://www.docker.com> (Apache License 2.0)
* NVIDIA-Docker, <https://github.com/NVIDIA/nvidia-docker> (3-clause BSD license: <https://github.com/NVIDIA/nvidia-docker/blob/master/LICENSE>)
* Python 2.7, <https://www.python.org/> (PSF: <https://docs.python.org/2.7/license.html>)
* OpenCV (3-clause BSD license: <https://opencv.org/license.html>)
* Numpy, <http://www.numpy.org/>, (BSD)
* Scipy, <https://www.scipy.org/>, (BSD)
* Scikit-learn, <http://scikit-learn.org/stable>, (BSD 3-clause)
* Tdqm, <https://github.com/noamraph/tqdm>, (The MIT License)
* Pandas, <http://pandas.pydata.org/>, (3-clause BSD <https://github.com/pandas-dev/pandas/blob/master/LICENSE>)
* Keras, <https://keras.io/>, (MIT license <https://github.com/fchollet/keras/blob/master/LICENSE>)
* Keras ResNet50 pretrained weights (<https://github.com/fchollet/deep-learning-models/releases/download/v0.2/resnet50_weights_tf_dim_ordering_tf_kernels_notop.h5>) (MIT License <https://github.com/fchollet/keras/blob/master/LICENSE>)
* Tensorflow, <https://pypi.python.org/pypi/tensorflow-gpu/1.4.1> (Apache License 2.0)
* Tifffile, <https://pypi.python.org/pypi/tifffile/0.10.0> (BSD)
* Matplotlib, <https://matplotlib.org/2.0.0/> (PSF)

1. **Potential Algorithm Improvements**

*Please specify any potential improvements that can be made to the algorithm:*

* Ensemble diversity (different scales, different networks): Since my solution is purely just the ensemble of two networks, using other diverse networks (e.g. LinkNet with pretrained encoders, DenseNet FCNs, U-NET with dilated bottlenecks etc.) at higher scales should help the performance as well.
* Instance segmentation: Since I used a semantic segmentation approach while this problem is effectively instance segmentation, there may have been some conjoined instances that could be better handled. Either using some form of instance segmentation algorithm (FCIS, Mask-RCNN) or a watershedding approach to break the instances may have been helpful to improve performance.
* Learned filtering: By looking at the features of certain buildings / shapes, a second level classifier could have been trained to remove or keep contours (to improve precision). These features may include area, proximity to other buildings, convexity, etc. This could have been much better than a fixed area of 175.
* Learned thresholding: By looking at the features of certain maps, a second level regressor could have been trained to identify the optimal threshold to use. Again, this could have been much better than a fixed binarization threshold of 0.4.

1. **Algorithm Limitations**

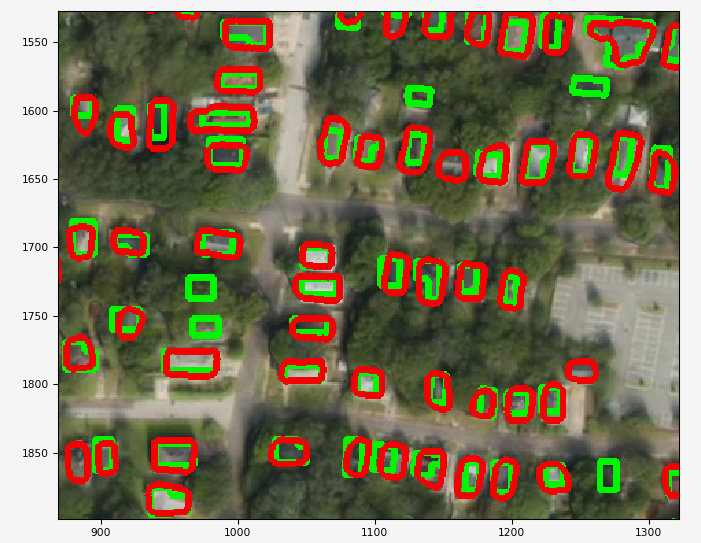
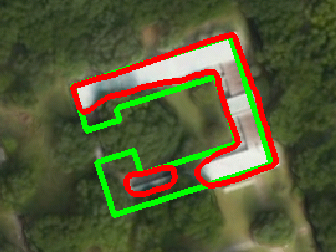
*Please specify any potential limitations with the algorithm:*

* Joined instances: Since this approach uses semantic segmentation, there are some scenarios where separate but close instances are incorrectly bound together. Two examples are shown in the figure below. As mentioned earlier, using a instance segmentation algorithm or some other post-processing approach may be able to help avoid binding or improve the performance in this regard.



*Figure 5. Examples of incorrectly conjoined instances (red = predicted; green = ground truth)*

* Hidden/occluded instances: There are a few observed instances where the ground truth is partially hidden / occluded by foliage or broken up and the network fails to predict the buildings in these areas, as shown in the figures below.



*Figure 6. Examples of occluded instances (red = predicted; green = ground truth)*

* Small buildings: Using a area threshold of 175 to eliminate contours may have resulted in many valid small buildings being eliminated. This problem may be mitigated by having a second level feature-based filtering classifier.

1. **Deployment Guide**

The steps are elaborated in README.md as part of the Github package for final testing, but is rewritten here for convenience:

* Assuming that the package has been extracted or cloned:

|  |
| --- |
| docker build -t kylelee . |

* Note that src/ as well as train.sh/test.sh files will be populated in this step. Run the container:

|  |
| --- |
| nvidia-docker run --name kylelee\_container -v <local\_data\_path>:/data -ti kylelee bash |

* Exit the container:

|  |
| --- |
| exit |

* (OPTIONAL) If final weight files are to be used directly for inference, first download the two weight files below into *src/weight\_final*:  
    
    
  U-NET: <https://drive.google.com/open?id=1ORgY4opiXLKWo7A8RQQUcJwOxVl8zAdl>  
    
  Resnet50: <https://drive.google.com/open?id=14JZNn4FDuc9go7MFEiWgWhfGSmZvqdOG>  
    
  Now populate both 512x512\_trimmed and 600x600 with weight files first:

|  |
| --- |
| docker cp src/weights\_final/unet.elu.best\_jaccard.512x512.unet\_sigmoid\_4bands\_dicebce\_trimmed.hdf5 kylelee\_container:/root/512x512\_trimmed  docker cp src/weights\_final/res50.best\_jaccard.600x600.res50\_sigmoid\_4bands\_dicebce.hdf5 kylelee\_container:/root/600x600 |

* Start the container for training or testing:

|  |
| --- |
| nvidia-docker start -a kylelee\_container -i |

1. **Final Verification**

*Please provide instructions that explain how to train the algorithm and have it execute against sample data:*The train/test steps are elaborated in README.md as part of the Github package for final testing, but is rewritten here for convenience:

1. To train the networks within the container, run the following - assuming the /data/train contains the training files.   
     
   This will sequentially prepare the areas for all three directories (splitting to tiles), then train UNET, followed by fine tuning the UNET with a different data set (where masks enclosed by black areas are trimmed), then followed by ResNet50 training. Take note that no prediction/inference on the test set is done in this step.

|  |
| --- |
| ./train.sh /data/train |

1. To generate predictions within the container, ensure that the weight files were populated (either from training completion above) or from Section 7, Step 4 (optional copying of pre-trained weights), and assuming that /data/test contains the test files / new sample data, run the following.  
     
   This will sequentially generate the predictions first for the UNET, then the ResNet50, then an ensemble followed by RLE submission to submit.txt.

|  |
| --- |
| ./test.sh /data/train /data/test submit |

1. **Feedback**

Please provide feedback on the following - what worked, and what could have been done better or differently?

* Problem Statement - The problem statement is interesting in that DSM/DTM is provided as opposed to the usual RGB + spectral band for satellite type of imagery.
* Data - As mentioned by some of the contestants in the forums, the mask data was inconsistent/noisy (https://apps.topcoder.com/forums/?module=Thread&threadID=908254) and given that scores after final testing are quite close, it would be good if this was checked/synced up before competition launch.
* Contest - No problems on this.
* Scoring - Generally no problems on this. However, I do have a minor point to pick - when submitting to the provisional leaderboard, is it possible to not stall for 2-3 hours if the file is missing (due to typo / copying for Google drive links, for example)? Rather just give an error and allow the user to submit again.

**NOTE**: Please save a copy of this template in word format. Please do not submit a .pdf