**Process**

Initially I began with a binaryCrossEntropy loss and a sigmoid top layer—partially because the sigmoid has the very nice interpretation of representing the likelihood that our model M thinks pixel (I, J) should be 0 or 1, and binaryCrossEntropy as punishing how badly the model’s predictions diverge from the actual distribution in the image. However, this caused the model to output a blank prediction—there was a class imbalance problem. Furthermore the training loss was negative—I had forgotten to normalize outputs.

I chose the Focal Tversky loss over the more common Dice Loss because the Dice loss treats false negatives and false positives with equal weight, which may not be a wise assumption; furthermore, while not attaining the best precision/recall metrics, Focal Tversky tends to ensure a good balance between precision and recall and is more stable. Experimenting with its hyperparameters, I found the recommended gamma of 4/3 to be the most useful.

After running several hundred epochs, I chose the lr = 1e-4 with the Adam optimizer because it balanced learning the loss curve well, and it never got “stuck” before (at least) a few hundred epochs. I chose 0.4 for my dropout rate both to control overfitting and also to ensure that the model moved towards a good training precision, as 0.5 dropout and higher made this much more difficult.

I chose he-normal because it is a relatively common kernel initializer.

Initially I composed the architecture using the Imperative API, however, retroactively this was a huge mistake: I couldn’t concatenate layers or save and load up my model with easy serialization. Furthermore, my model structure remained opaque and unintuitive. I made the decision to refactor-- actually, to write the whole thing from the ground up, again, using the Functional API—and it was the best decision I made, really simplified so many things tremendously.

I think my data augmentation strategy also really helped me; at first I tried to use Keras’ built-in ImageDataGenerator functionality with flow(), but discovered that zipping up two different generators, even with the same seed, did not jointly preserve the relationship between images and masks. I turned then to Albumentations and its data augmentation API to build a data augmentation pipeline that would randomly apply transformations while preserving those pair-wise relationships; it worked like a charm, and probably did more both than any amount of hyperparameter tuning given the low sample complexity and the highly imbalanced nature of the dataset.

Going forward, I think I’d like to train a variational autoencoder to synthesize inputs for my u-net, and research other transforms or means of data augmentation that would give the model a better intuition about the actual underlying distribution.