Segmenting Microscopy Images

Chris Malec

Springboard DSC Track
Capstone Project #2
November, 2019

Introduction:

Biological research creates enormous quantities of image data. Often times, it requires experts to infer the boundaries of different cellular structures or anatomical features. The more computer vision can be harnessed to identify features in microscopy images, the faster actionable information can be gleaned from scans. Since hand annotation can take much, much longer than the actual data collection, this would drastically speed up and improve accuracy of data processing and interpretation, leading to more insights and discovery.

Approach:

Data Acquisition and Data Wrangling

This project will look at the identification of mitochondria within neuronal cells from Scanning Electron Microscope (SEM) images. A relatively new three dimensional imaging technique in this field is known as Focused Ion Beam - SEM, in which a three dimensional tissue sample is sequentially imaged by the SEM, and then a layer is milled away by a beam of focused ions, allowing the layer below to be imaged. This procedure was used to create a 5 micron x 5 micron x 5 micron volume of data representing neuronal cells.

Though there are many features of interest in a biological system, the mitochondria are particularly important since they create the energy for the cell. Therefore, a group of researchers took the time to hand annotate 330 images to separate out mitochondria pixels from non-mitochondria pixels (**CVLab SEM dataset**[1]). This dataset is notable in that the data is truly three dimensional, which means that the data is given as an image stack in which each pixel is

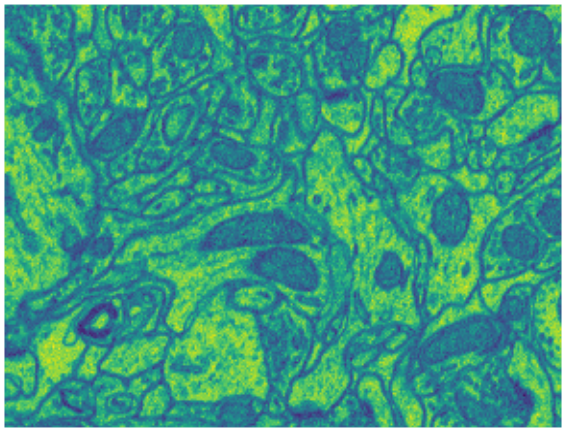

Figure 1. A single slice of the training volume, which forms a 768x1024 image

[1] CVLab SEM dataset: https://cvlab.epfl.ch/data/data-em/

actually a 5 nanometer x 5 nanometer x 5 nanometer voxel. A sample image from the stack in shown in Figure 1.

The filetype for the image was a series of multipage TIF files. There were four files, two containing training images, and the ground-truth images (images whose pixels are annotated as 1 for mitochondria and 0 for non-mitochondria), and two files containing testing images and their respective ground-truth images.

Once the files were downloaded, they could be converted to an ImageSequencer object through the PIL (Python Image Library[2]) package. This object allowed me to easily iterate through the sequence of TIF images and create a four dimensional numpy array. I chose the dimensions to be compatible with conventions used by tensorflow, which I plan on using later. Therefore,, the numpy array had size # examples x # pixels high x # pixels wide x # channels. Since electron microscopy images are in grayscale, there was only one channel, but again, I kept the four dimensional structure to keep the array in tensorflow's expected format.

I implemented a function to perform the necessary operations to convert the given file to a numpy array and used it on all four TIF files. Finally, I saved the numpy arrays for use in further notebooks, and took a look at one of the image slices to get an idea of what the data looked like.

Exploratory Analysis

I had several questions to explore about the data. First off, I visualized the 3D volume by looking at several slices along the z-axis of the imaged volume. I also created a visualization of the outside of the rectangular prism of image data. Since the resolution is roughly the same in all three dimensions, Figure 2 shows that the x-z and y-z planes look very similar to the x-y plane.

What is the average mitochondria area, the average volume?

Working with the ground-truth images for the training data, I was able to add up the average total area for each image by counting the number of pixels that were labeled as mitochondria for each image. However, I needed to also count the total number of mitochondria in each image.

This would be exceptionally tedious to do manually, so I used a simple algorithm that counted the number of 'humps' in each line of an image. Each hump was considered to be a

[2] Currently known as Pillow: https://pillow.readthedocs.io/

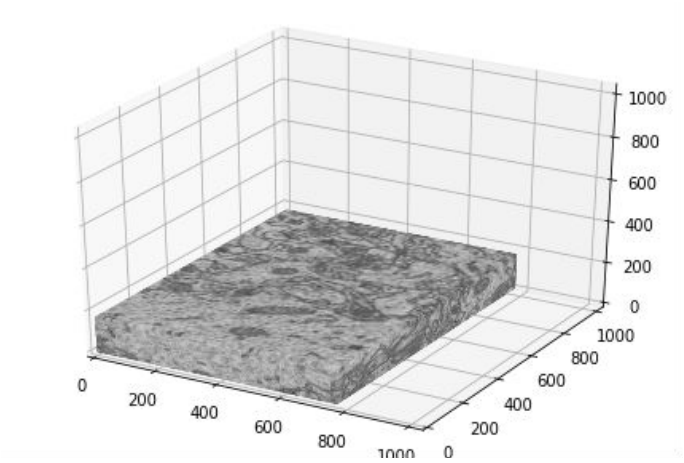

Figure 2. Image slices along the z-axis and image volume as a rectangular prism

mitochondria, and when the number of humps changed, it was assumed that the raster line had moved beyond some number of mitochondria. This algorithm agreed with my manual count for three randomly selected images.

I applied the same reasoning to find the total number of mitochondria in the volume, by applying my earlier function to each slice, I assumed that when the number of shapes counted changes, then the image slices had moved beyond one of the mitochondria. In this way, I counted the average number of pixels per mitochondria and average number of voxels per mitochondria. I multiplied by 25 nm^2 to get an average area of 70,013.9 nm^2 (a square about 53 pixels on a side) and by 125 nm^3 to get an average volume of 13,430,224.3 nm^3 (a cube about 48 pixels on a side).

In later steps, these estimates helped me determine how large an image must be in order to contain at least one mitochondria on average.

Are the cells spherical, elliptical, some other odd shape?

By calculating the edges of the mitochondria from the ground-truth images, I could reconstruct a

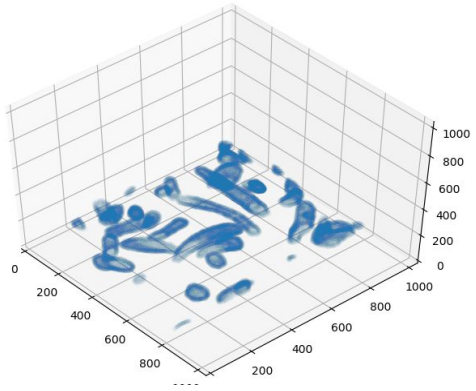

Figure 3. D scatterplot of mitochondria surfaces.

3D scatterplot that displays the surfaces of the 3D mitochondria, shown in Figure 3. Some mitochondria are definitely spherical, while others are stretched out. Most shapes appear to be convex, which is good, because my counting algorithm wouldn't work well if the mitochondria routinely hooked around or made odd 's' shapes.

How much variation is there in contrast, is there a difference in the two regions?

The microscope image is really a measurement of the intensity of electrons reflected to each pixel from the surface. The microscope operator will typically try to prevent saturation of the pixel intensity to either zero or maximum. Indeed, creating a histogram of pixel intensities, Figure 4 reveals that the pixels do not span the entire possible range, and there is no pixels at saturation value.

When the two regions are plotted separately, it can be seen that the mitochondria have more lower intensity pixels than the general image. This can also be seen from casual inspection of the image slices. However, the pixel intensity range in the mitochondria region entirely overlaps the pixel intensity range of the non-mitochondria region. This means that though the mitochondria by and large reflect less electrons than the non-mitochondria regions, this cannot be used as a good method of identifying the two regions.

It's good to know that the solution to the problem at hand cannot be achieved simply by setting a threshold level and taking pixels with a value lower than the threshold as mitochondria. It will actually be worthwhile to train a neural net to segment the images.

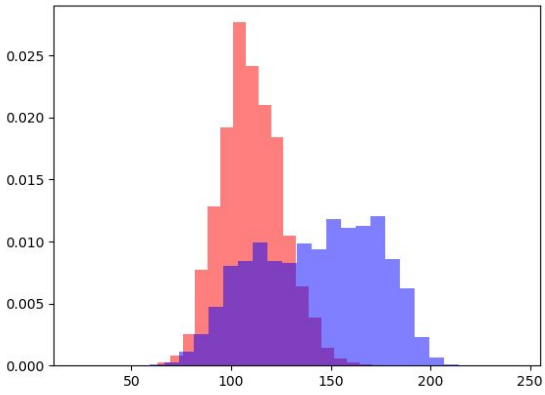

Figure 4. Histograms of mitochondria (red) and non-mitochondria (blue) pixel intensities.

Statistical Inference

An important question in the Data Science problem is how to present the data to the machine learning model. Should the data be presented as a series of image slices? Or maybe the data should be presented as a series of volumes. The thicker the volumes used, the less training examples, but perhaps they are more useful examples.

With this issue in mind, I wanted to look at image correlation between different slices along the z-axis. By this I mean, testing how correlated each image slice was with the image at the bottom of the stack. Subsequent image slices are not independent, but pictures at different depths of many of the same cellular structures are expected to be. Presumably, the bottom image will be correlated perfectly with itself, a little less with the image above it, and a little less with the image above that.

I implemented a function to calculate the Pearson correlation coefficient of two series of pixel intensities, as well as a 95% confidence interval. I could find built-in functions to get the correlation coefficient, but the confidence interval functionality didn't work like I wanted it to. I then performed the statistical test on each layer of the training data in turn to create a graph of correlation coefficient vs distance from the bottom image shown in Figure 5.

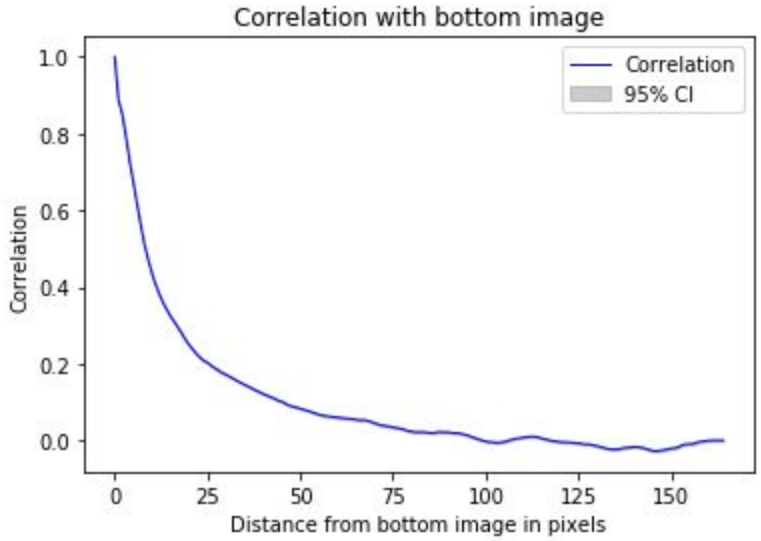

Figure 5. Correlation coefficient vs distance from the bottom image slice.

Another important concept in image segmentation, particularly for microscopy data (which can be scarce) is data augmentation. Data augmentation involves transforming the image by shifting, resizing, flipping, distorting, or otherwise altering the image to increase the volume of training examples as well as to avoid overfitting.

I was interested to know how well a transformation such as flipping the image from left to right might decorrelate adjacent layers. It turns out that such operations are very effective, so that

data augmentation can be used to greatly increase the number of training examples that appear independent of each other. Figure 6 Shows a plot similar to Figure 5, but the images above the bottom image have been flipped about the horizontal axis.

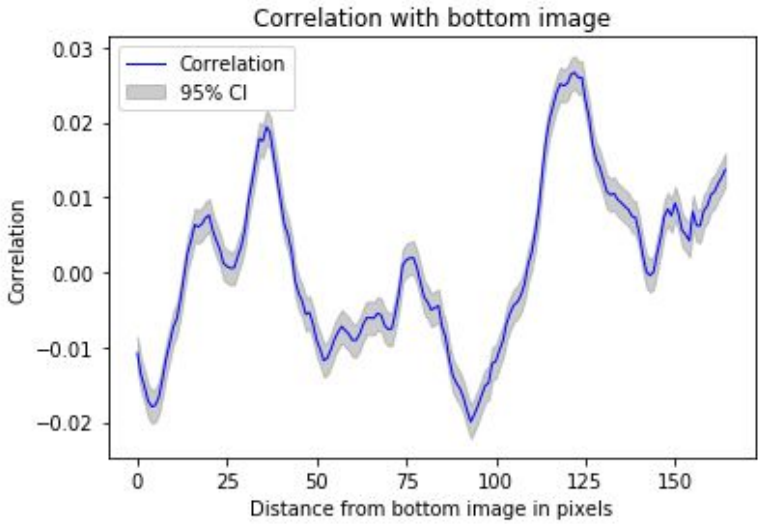

Figure 6. Correlation coefficient vs distance from the bottom image slice after a horizontal flip.

Noise and distortions were other transformations I looked at, with the results summarized in Figure 7. Adding random noise, which amounted to adding a random pixel intensity drawn from a normal distribution, affect the correlation of adjacent images as expected. Greater noise lead to images being less correlated.

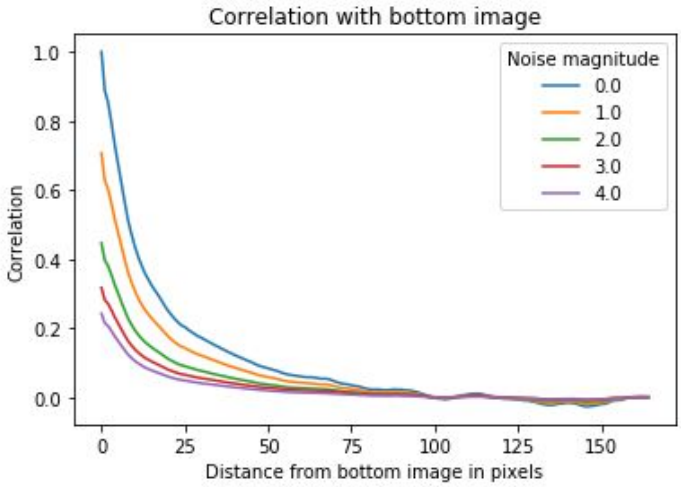

Figure 7. Correlation plots with different magnitudes of random noise.

Finally, distortions were introduced by creating a random vector field that displaced pixel intensities from the original image and then interpolated back to the original grid. This resulted in

the borders of features being altered. Two parameters, alpha and sigma control the distortion. Higher alpha creates larger distortions, meaning that the original pixels are moved farther away on average.

Higher sigma smooths out the distortions so that the borders of features remain roughly intact though the exact shape may be changed. These distortions caused a decorrelation of adjacent image slices. Figure 9 demonstrates that even for somewhat large distortions, the effect is less than for a transformation such as a flip. It is important to keep in mind, that decorrelating the images is not the entire goal of the transformations. Training a model that is robust to small changes from the training data and so performs well on unseen data is an equally important goal. Examples of this type of transformation are shown in Figure 9.

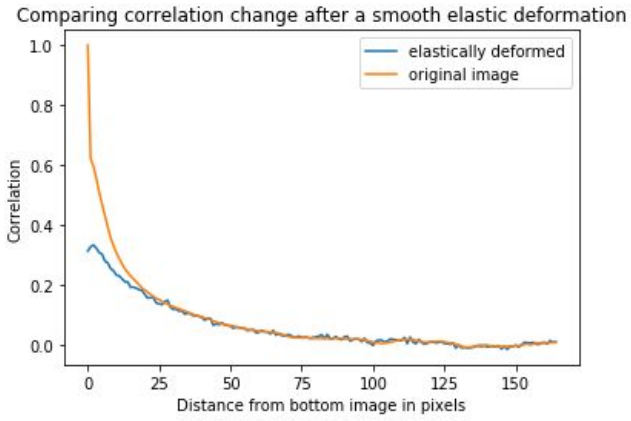

Figure 8. Correlation vs distance from bottom image for no transformation and a distortion described here.

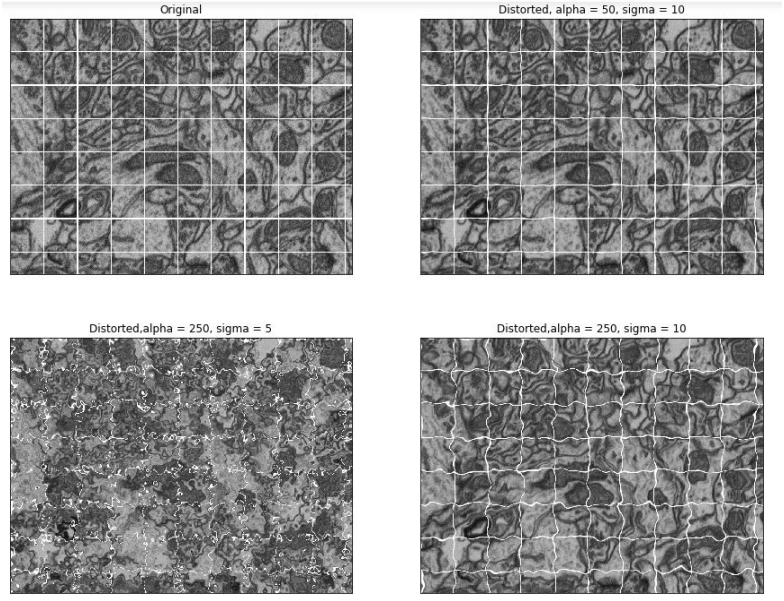

Figure 9. Several examples of an elastic deformation as described in the text for different values of alpha and sigma. Alpha increases the distortion, while sigma controls the smoothness.

Baseline Model

Machine Learning - Deep Convolutional Neural Nets:

Most tasks involving images can be effectively handled by a class of neural nets known as convolutional neural nets, oftentimes shortened to CNN. CNN's apply an operation known as a 'convolution' to an image. This amounts to multiplying each small patch of pixels (for example a 3x3 square) by a filter of the same size, and then taking the average value of the resulting numbers. This is done for many image patches over and over and over again. The eventual output of the net could be a set of categories, a number, or another image. The overall concept is that convolutions avoid creating fully connected layers and thus drastically decrease the number of connections while maintaining the important relationships between pixels that are near each other.

In order to train the net, the output is compared with the true output, a loss is produced, and the weights of the net updated in order to minimize the loss. Variations of gradient descent optimization have been developed specifically for neural nets, as the loss surface is not convex, in other words there are many local minima or long ridges that make classic optimization algorithms converge incredibly slowly.

U-Net Deep Neural Net Architecture:

For image segmentation, an effective architecture that has emerged is known as U-Net[3]. The name is a reference to the architecture, as when it is graphed, it actually looks like the letter U. The basic idea is that convolution operations are followed by max-pooling until the image is only a few pixels wide and many filters deep. This is the center at the bottom of the 'U,' and after this point, the neural net applies upsampling operations, followed by convolutions. In each upsampling layer, the filters are concatenated with filters from the downsampling layer of equal size, which provides context for the upsampling layer. The final layer has one output for each pixel in the original image's size is another image, except binarized, so that pixels belonging to a class of interest are one and pixels that do not belong to this class are zero.

[3]U-Net: Convolutional Networks for Biomedical Image Segmentation, Ronneberger et al, 2015, https://arxiv.org/pdf/1505.04597.pdf

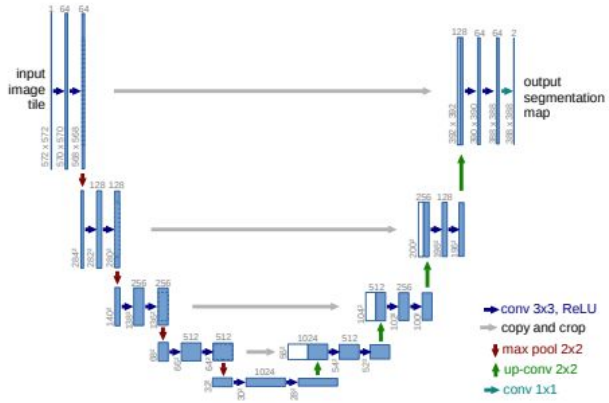

Figure 10. A graphical depiction of U-net reproduced from the cited publication.

Building the model in Tensorflow and Keras

I have implemented this neural net using Tensorflow and Keras, borrowing the basic architecture from a u-net built in Tensorflows github page[4]. Keras is a wrapper for Tensorflow and simplifies some aspects of building a neural network by creating a network as a series of specialized layers. This net is mostly built from convolutional layers that apply convolutional filters, deconvolutional layers that turn reverse the convolution operation, and max-pooling layers which downsample a 2x2 square of pixels into one pixel. Rather than use large filters (5x5, 7x7) the network uses only 3x3 filters, but cascades them to create an effective 5x5 filter, a concept borrowed from ResNet[5].

The model is trained with default parameters using the Adam optimizer[6], which has quickly become one of the most popular optimizers for neural nets since its publication. I only trained the net for 10 epochs to reach a baseline.

Evaluation and the Dice Coefficient

A common metric to evaluate the similarity between two binary images is the Dice Coefficient. The Dice Coefficient is the number of pixels that are the same between two images divided by the total number of pixels in the two images labeled as 1.

$$Dice\ Coefficient\ =\ \frac{2\cdot\Sigma(X\cap Y)}{\Sigma(X\cup Y)}$$

When two images have no pixels labeled ‘1’ in common, the coefficient is 0, and when all coefficients labeled ‘1’ are the same, the coefficient is 1. The dice coefficient can be turned into

[4] Image segmentation tutorial: https://github.com/tensorflow/models/tree/master/samples/outreach/blogs/segmentation_blogpost

[5] Deep Residual Learning for Image Recognition, He et al., 2015, https://arxiv.org/abs/1512.03385

[6] Adam: A method for stochastic optimization, Kingma et al, 2014, https://arxiv.org/abs/1412.6980

a loss function by subtracting it from 1, so that 0 minimizes the loss and maximizes the Dice coefficient.

The baseline model does not do particularly well, with a Dice Coefficient of 0.09 on the test data, however it outputs an image of probabilities of the same shape as the input image. The test loss 0.26 is only somewhat worse than the training loss 0.22, the model is simply not fitting the data. There could be many reasons for this, which will be addressed in the extended model. As a note, a goal I had was to improve on the Dice Coefficient of 0.87 reported in the original u-net paper making use of this dataset.

Extended Model

Dicing up images

To make computation easier, as well as to make the images square, I divided each 768x1024 image into twelve 256x256 images. This amounted to dividing the image into a 3x4 grid of smaller square images. Square images traverse through the convolutional net easier, since Keras does not allow for uneven zero padding with its convolutional layers. This could theoretically be dealt with by accessing the Tensorflow object directly.

The number of training examples when dividing into 12 squares becomes 1980 up from 165, all at the resolution of the original images. Since the net is optimized in mini-batches, the computer will only need to deal with about 5-20 images at a time.

Data Augmentation

For many deep learning applications, and particularly microscopy data, data augmentation is a technique that becomes necessary. Neural Nets require an immense amount of data to train, however microscopy data can be scarce since it is expensive to collect, and labeling the data is more expensive still, as experts can be in short supply. Data augmentation is the process of applying transformations to existing data to create new data. Transformations can include adding noise, flipping the image, rotating the image, adding shifts in the coordinate axes, or anything else that may alter the image in some noticeable way. Data augmentation both improves the training error, it provides a certain amount of regularization by preventing the model from overfitting on a small number of images.

It is important to note here, that the original paper describing U-Net image segmentation used only 30 training examples to achieve effective image segmentation. Therefore data augmentation can decrease both the train and test errors, and in this way it is a regularization technique because it improves the response to unseen data.

In Table 1, I applied data augmentation to the baseline model and looked at how the affected the training and validation loss. I interpreted transformations that made the training loss lower to effectively generate images new enough to create a non-zero gradient that points towards a

better solution. The transformations that had a training/validation loss that were closer together were effective at regularizing the training process.

Transformation	Training Loss	Validation Loss
None	**0.302**	**0.905**
Vertical Flip	**0.425**	**0.955**
Zoom 0.2	**0.160**	**0.891**
Rotate 45	**0.136**	**0.489**
Rotate 90	**0.143**	**0.284**
Height/Width Shift 0.1	**0.112**	**0.345**
Height/Width Shift 0.3	**0.122**	**0.457**
Elastic Deformation	**0.063**	**0.205**
Random Noise	**0.100**	**0.285**

Table 1. Data Augmentation Training/Validation losses on on epoch of the baseline model

For the final data augmentation pipeline, I selected rotation, elastic deformation, vertical flip, and horizontal flip.

Training - optimizing hyperparameters

Next, I put data augmentation to the side and tuned some of the network hyper-parameters. The hyperparameters for Neural nets are particularly difficult to optimize since the loss surface is not convex. Therefore gradient descent does not work as well as for algorithms such as Support Vector Machine or Logistic Regression, which deal with convex search spaces. The Adam optimizer has quickly become well regarded for neural nets.

Below, I briefly review some of the major parameters of a convolutional neural network that I can adjust to improve the performance of my model.

- **Optimizer (Adam)**
    - **Learning Rate** - Typically the most important parameter, too fast results in a poor fit, too slow results in slow convergence.
    - **β_0 and β_1** - Momentum terms that prevent solution from sticking in local minima, the default values are rarely changed.
    - **Epsilon** - A smoothing term to prevent zero-division, usually left at default, though some image categorization nets have reported good outcomes for higher values.

- **Loss Function**
    - **Dice Loss** - A loss function created from the Dice coefficient (1-D). It more directly optimizes for the metric of interest.
    - **Log(Dice Loss)** - Increases the gradient of the dice loss since it lies between 0 and 1, improving the convergence.
    - **Binary Cross-Entropy Loss** - A common loss function for binary valued prediction models.
- **Filter size** - Larger filters are more difficult to compute, but could capture more information. Large filters are not always better, as relevant patterns may be small.
- **Number of layers** - The 'depth' of the network, the more layers the more parameters the model has to fit.
- **Activation function** - The non-linearity applied after each layer, 'relu' is the usual choice. As long as the output of the function falls in an appropriate range the exact choice is usually not important, but it has to be non-linear, otherwise a ten layer network doesn't function any differently than a one layer network.
- **Batch Size** - The number of training examples used to calculate one step of optimization. Larger batch sizes are more representative of the training set, however for precisely this reason, intermediate batch sizes can perform better by providing more variance.
- **Normalization type**
    - **Batch Normalization** - Normalizes the weights per feature for a single mini-batch. Typically batch normalization provides little downside and a faster-training network.
    - **Layer Normalization** - Normalizes the weights per training example for a set of features.
    - **No Normalization** - This is a perfectly valid choice, though sometimes leads to the network taking longer to train.

I didn't adjust all of these, and definitely not in a grid search style. There were simply too many parameters to tune in a purely systematic fashion. This is a strong argument for getting some more computing power to develop this model further. However, I adjusted a number of parameters by changing them from the baseline model one at a time and looking at the Dice Coefficient after one epoch for the training and validation sets as well as the time it took the model to train one epoch. The results are summarized in Table 2.

Parameter	Value	Train Dice Coefficient	Validation Dice Coefficient	Time to train epoch (s)
Default	**N/A**	**0.50**	**0.18**	**2396**
Learning Rate	**1E-1**	**0.52**	**0.10**	**2300**
Learning Rate	**1E-5**	**0.77**	**0.03**	**2271**
Epsilon	**1E-4**	**0.93**	**0.78**	**800**
Epsilon	**0.1**	**0.13**	**0.09**	**2246**
Batch Size	**5**	**0.88**	**0.80**	**820**
Batch Size	**10**	**0.81**	**0.73**	**767**
Batch Size	**20**	**0.79**	**0.24**	**2486**
Loss Function	**Dice Loss**	**0.73**	**0.63**	**759**
Loss Function	**Log Dice Loss**	**0.84**	**0.69**	**742**
Normalization	**Layer**	**0.35**	**0.47**	**1688**
Normalization	**None**	**0.35**	**0.52**	**734**
Model Depth	**Add two 16 filter layers at beginning and end**	**0.38**	**0.12**	**1018**
Model Depth	**Add 2048 filter layer at the center**	**0.37**	**0.12**	**2626**
Model Depth	**Removed two 32 filter layers**	**0.81**	**0.61**	**2296**
Filter Size	**Effective 5x5**	**0.44**	**0.06**	**3497**
Filter Size	**True 5x5**	**0.29**	**0.08**	**3286**

Table 2. Models with one value changed from default, along with the dice coefficient on the training and validation sets, and the time to train one epoch

From Table 2, we can see a lower learning rate, a value of epsilon of 1E-4, small batch sizes, using the Dice loss, and removing the 16 filter layer from my baseline network all lead to gains in model accuracy. It is worth noting that all the parameters do not operate independently of each other, therefore, when I tried to train a model with a combination of all the best values shown here, the final Dice coefficient on the training set was 0.08. So some guess-and-check methodology is still necessary until computing power becomes large enough to process the

models on a grid search and try all possible combinations. A knowledge of the effects of the parameters and diagnosing individual training runs can significantly speed up the search by offering additional guidance for selecting hyperparameters.

During training, an easy way to determine the quality of the fit on unseen data is to use a validation set, and apply the model to this validation set at the end of each epoch. While this method is in some ways less biased than using a portion of the training data, over time it can still produce overfitting. Over time, the model will do slightly better on the validation data by chance, which causes overfitting by biasing which models are selected.

This is seen by the fact that the average dice coefficient of truly unseen test data is lower than that of the validation data.

Findings

The Dice Coefficient of the Test data in the best model was 0.89, which did improve on the original paper for this data set.

The model's efficacy can be visualized by viewing the sections of predicted vs actual segmentations, as well as incorrectly classified regions. We can see in Figure 11 that the predicted and ground-truth values are remarkably similar. Looking at the difference between the prediction and the ground-truth, we can see that the edges are the most difficult to classify regions.

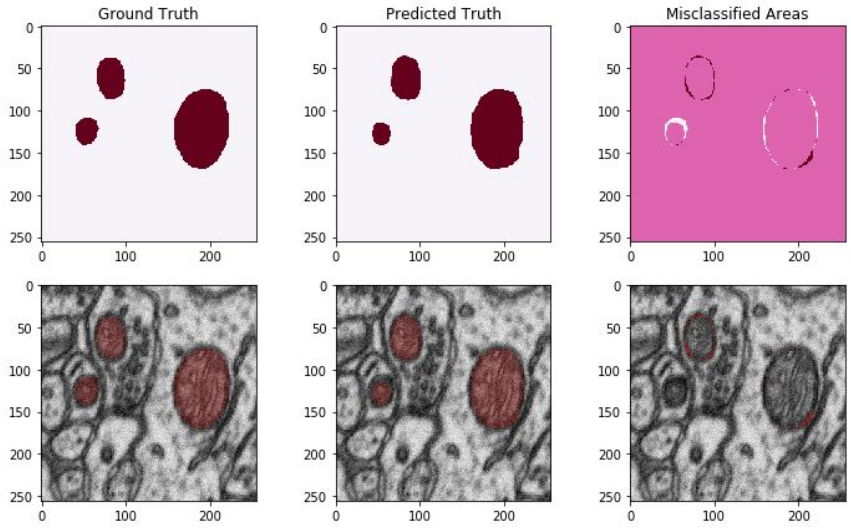

Figure 11. An example segmented image from the Test set

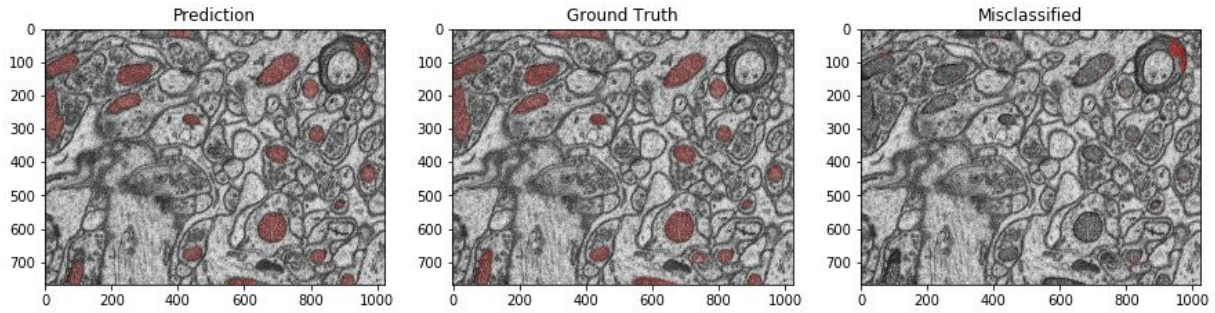

Figure 12. A fullsize image generated by predicting on many 256x256 windows

Figure 11 shows one of the 256x256 slices, but I'd like to see how the model does on a full size image from the test set. The model should work on any square image, but the original images aren't square. I therefore implemented a function that applied the model to several 256x256 squares in the image and took the average over all such predictions on a pixel by pixel basis. The result is shown in Figure 12. The full size image has a small spot in the upper right hand corner that was misidentified, but otherwise, the mitochondria can be seen to be effectively segmented.

Future Work

There are many directions to take to further improve the performance of this and similar models. One is to look at using the data in a more "three dimensional form" rather than as a sequence of two dimensional images. This could be done by grouping nearby layers together along the 'channel' axis and feeding it through the U-net model as-is, or it could be done by adding a fifth axis to represent depth and using truly three dimensional convolutions. In both cases there would have to be a fair bit of adjustment to the data augmentation pipeline, and I would have to write a custom generator. Although the combination Tensorflow/Keras supports three dimensional convolutions, most of the transformations in the ImageDataGenerator class only support four-dimensional tensors.

From my experimentation with data augmentation, it is clear that this can have a profound effect on both the training and the test error. Therefore it makes sense to study in detail the effects of various transformations, particularly those that create vacant pixels, like rotation and shift. In general, it seemed that there may be smarter ways to do data augmentation, as not all transformations are fully independent of each other, and some transformations cause parts of the image to be under-represented in the training set (like a rotation clipping the corners off).

Another pathway for future work is simply to collect more data. A major weakness of this particular dataset is that the train and test data are from the same microscopy run. This means that though they are separated by many cells, and do not show any appreciable correlation coefficient with each other, the training data and the test data are not truly independent. The

microscope state (magnification, brightness, etc) was also likely very similar over the whole run, and so the model might not do well with data from another microscope session. However, many common differences between microscope sessions can be well modeled by the data augmentation transformations, so this may be less of a concern.

Finally, many choices made here were related to what parameters could make the problem tractable for my available computing power. Given the variety of cloud computing services available, such as Google Cloud, Amazon Web Services, or Microsoft Azure, it would make sense to simply use more computers to speed up calculations once some basic parameters have been worked out. However, since one pays for compute time, it would be dangerous to just naively set up a grid-search and let the model train. I believe the best model presented here could easily be improved by allowing the optimization algorithm to work for many more epochs, perhaps in the hundreds.

Client Recommendations

Automatic segmentation promises to greatly accelerate the analysis of microscopy images. In turn, this can lead to new discoveries and more complicated data collection procedures being used as routine diagnostics. In order to use the algorithm effectively, I suggest adjusting how metrics are reported and planning for an explainable AI[7] layer to give the user an indication of how the model chose to label certain pixels. Though neural nets are normally thought of as 'black boxes' whose decisions cannot be explained, it is possible to find the filters that were most strongly activated by a certain section of the image, so the user can directly see what patterns the computer was using to identify a certain region as a mitochondria.

For research or medical applications, it is very important that the experts be able to trust that the segmentation algorithm is producing results equal to that, or even better than, a human. To that end, the Dice coefficient is probably not actually the metric to report to clients. What researchers want to know is how many mitochondria were correctly segmented. A number such as 75/302 or 300/302 mitochondria correctly segmented would allow researchers to decide if the convenience offered by the algorithm is worth any decreased accuracy.

In addition, putting the identification in terms of precision and recall would help medical professionals and researchers understand the results in a way they are accustomed to. In applications where the segmentation is related to identifying the presence or absence of disease it would be important for ethical and legal reasons to have a standard metric used in the medical profession to assess the quality of the model.

[7] https://www.darpa.mil/attachments/XAIProgramUpdate.pdf

As with other image recognition technology, the possibility exists that the model could eventually do better than humans[8], in which case it will be important to add some interpretative layer to the model. Otherwise, researchers may be overly tempted to override the algorithm's decisions even when their own decisions may be worse.

Finally, as more microscopy data is acquired, it is important to incorporate this data into the model. That way, the effect of different tissue samples and different microscope states will be taken into account. It may also help to adopt a standard magnification when taking images, or to at least make sure to record the magnification so that images can be resized if need be. At least in the near term, the model should be treated as a work in progress, as the current training data is drawn from one microscope run.

8

https://www.forbes.com/sites/michaelthomsen/2015/02/19/microsofts-deep-learning-project-outperforms-humans-in-image-recognition/#6025dd9c740b