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:

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[1]. 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

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

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.

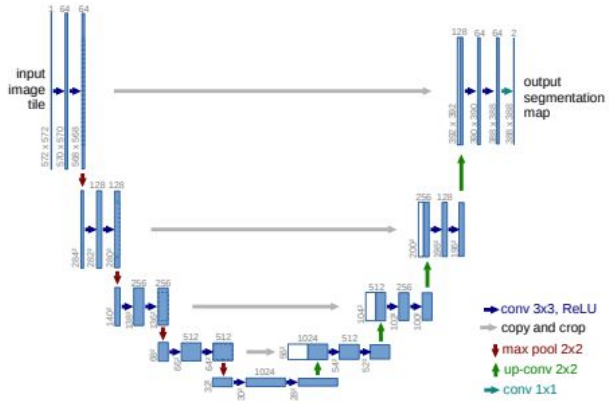

Figure 10. A graphical depiction of U-net reproduced from the cited publication.

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[2]. 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[3].

The model is trained with default parameters using the Adam optimizer[4], 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.

---

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

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

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

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

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 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.