Parameterized GAN architecture for Unannotated Image Segmentation

Ryan Nelson
M.S. Computational Biology
Carnegie Mellon University
Pittsburgh, PA 15213
ryann@andrew.cmu.edu

May 2, 2022

Abstract

*Image segmentation is a critical tool used in biological image analysis. Supervised machine learning approaches require an extensive amount of labeled image data, which can be expensive to produce and difficult to obtain. Semi-supervised approaches allow the model to leverage a small amount of labeled image data to segment unannotated images, vastly extending the capabilities of the model. We developed a framework to use hyperparameters to dynamically generate GAN model architecture based on a given configuration (*https://github.com/ryanquinnnelson/CMU-02699-Image-Segmentation-via-GANs*). We used this framework to identify architecture that is as performant as published literature but uses less memory and runtime. We improve upon this performance by pretraining with triplet loss and negative hard mining (*https://github.com/ryanquinnnelson/CMU-02699-Image-Segmentation-via-GANs2*). Additionally, we present a foundational framework for training deep learning models which provides standard training functionality, reducing code overhead and speeding up model development time (*https://github.com/ryanquinnnelson/octopus*).*

1. Introduction

Image segmentation is a technique used to identify the location and type of relevant objects within an image, often performed by classifying every pixel in an image into one of several predetermined categories. Image segmentation has many important applications in biology, one of which is improving the analysis of medical imagery. Many kinds of medical images, such as tissue samples, are taken for diagnosis and research purposes. Analyzing these images is an important step toward understanding disease and prescribing treatment. As imaging technology has improved, the quantity of medical images produced for analysis has grown exponentially. However, the number of specialists, such as radiologists, who are trained to review these images has not grown at the same pace. This resource challenge presents an opportunity to supplement the ability of medical professionals and produce diagnoses at a faster pace, which can result in better patient outcomes.

Generative Adversarial Networks (GANs) are a common deep learning model type used in image segmentation. GANs consist of two models - a generator and a discriminator. For supervised image segmentation, a generator takes raw images as input and produces a pixelwise segmentation probability map that scores how likely a given pixel is to be classified into each of the possible predefined classes. A discriminator takes generator output or target labels and the original raw image as input, then predicts whether the input the discriminator received is an annotated (labeled) segmentation map or an unannotated (generated) segmentation map. The two models compete to outperform each other: the generator tries to generate probability maps that fool the generator into thinking a generated map was annotated; the discriminator tries to correctly distinguish between annotated and unannotated maps. The goal of this process is to produce a network with higher segmentation performance than would be possible without competition.

Many architectures have been presented for GANs in image segmentation [1, 4, 5, 7]. Models are typically presented as static architectures, with a defined number of CNN layers and linear layers and defined relationship between layers. Little emphasis is placed on describing the process of achieving the published model or the trade-offs of alternative models. One of the challenges with assessing the performance of any architecture is optimizing the set of hyperparameters that defines the model itself. Does one model perform better for a given dataset because of something inherently better about its architecture or is it because the current configuration of model layers is better suited for the dataset? Is the increased memory required for a given architecture worth the performance improvement? This work analyzes the GAN architecture of DAN (Zhang, 2017) and the component architectures its authors reference (DCAN, VGG16) to elucidate this question. DAN requires a sizeable amount of memory, which requires a compromise between input image quality and memory requirements. This work parameterizes generator and discriminator components of GANs, describes a search process for tuning these parameters, and presents alternative architectures that produce high performance yet require significantly less memory and runtime. By parameterizing components, models can be tuned leveraged on a wider range of datasets than static architectures. Reducing memory requirements allows GANs to be used on cheaper commercial hardware, which can enable proliferation of the technique for fields with fewer resources. Reducing runtime means more training epochs can be performed, which enables a larger quantity of hyperparameter tests and allows a given model to undergo more training epochs.

This work also explores different approaches to supplementing conventional cross-entropy loss used by DAN with contrastive loss. We find that pretraining with triplet loss and negative hard mining can provide a boost to segmentation performance.

Additionally, developing deep learning models for image segmentation requires a significant amount of programmatic overhead, from environment setup to model generation to phase progression to metric tracking. This overhead, in turn, requires significant time to perfect, which takes up time which could otherwise be spent finding the best architectures and hyperparameters. The literature reviewed for this work does not address this issue of overhead, with bespoke code written for each published model. This work presents a Python framework named `octopus` to standardize and abstract the processes required for deep learning model development. Lightweight and simple to use, this framework strikes a balance between customization and standardization, accelerating the process of development without restricting the types of models which can be run on the framework.

2. Related Work

2.1. DCAN: Deep contour-aware networks for accurate gland segmentation

Chen et al. (2016) proposed a fully convolutional network (FCN) to improve upon state-of-the-art image segmentation methods for gland segmentation. By combining output from different CNN layers, the author's network uses multi-level contextual features to generate probability maps. This, in turn, allows the model to mitigate high levels of variation in gland shape. The authors also use transfer learning to supplement limited data supply when training the model. Almost like an autoencoder, the proposed FCN is composed of two modules: a downsampling module, and an upsampling module.

2.2. Suggestive Annotation: A Deep Active Learning Framework for Biomedical Image Segmentation

Yang et. al. (2017) proposed combining a deep neural network with active learning to more efficiently identify which images to annotate to improve model accuracy the fastest. By using a set of FCNs and uncertainty sampling, the authors achieved state-of-the-art image segmentation performance.

2.3. Deep Adversarial Networks for Biomedical Image Segmentation Utilizing Unannotated Images

Zhang et al. (2017) proposed a semi-supervised GAN model which uses annotated images alongside unannotated images in order to produce segmentations for the unannotated images. The GAN model consists of a segmentation network (SN) and an evaluation network (EN). To produce the probability map with the SN, the authors designed a version of DCAN that uses concatenation of upsampling splits rather than summation. For the EN, the authors implemented a version of VGG16. The proposed model is shown to perform better at image segmentation than state-of-the-art models on the 2015 MICCAI Gland Challenge dataset.

2.4. Learn to segment single cells with deep distance estimator and deep cell detector

Wang et al. (2019) proposed a two-pronged approach to single cell segmentation. The authors leveraged a CNN to learn the Euclidean Distance Transform (EDT) of input images, then used a Region with CNN (R-CNN) model to identify cells within the EDT. This approach improved cell count accuracy (CCA) over models that approach classification using pixels.

3. Method

3.1. octopus - a framework for running deep learning models

Most of the setup for training a deep learning model is the same: logging, environment preparation, data loading, model generation, running epochs, and metric tracking. Rather than re-write code for these standard processes for each new model, it would be more efficient to standardize these processes into a framework on which models can be run. We present `octopus`, a Python framework for running deep learning models. The `octopus` framework standardizes eight common tasks in deep learning model training:

- Logging
- Environment Setup
- Model Tracking
- Data Loading
- Model + Component initialization
- Pipeline setup (phases)
- Running the pipeline
- Cleanup

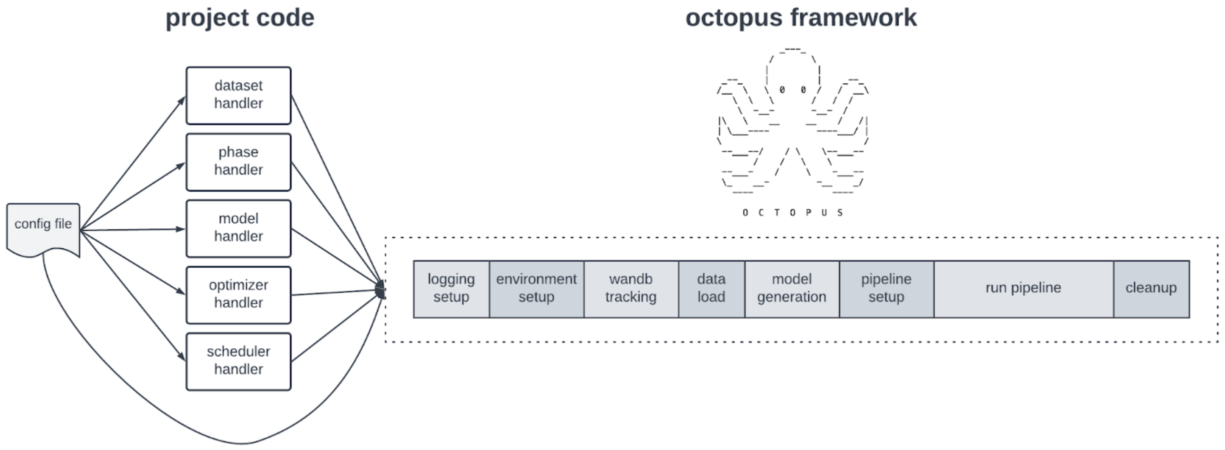


Figure 1: Diagram of input to the `octopus` framework and the modules within the framework itself

Additionally, hyperparameter tuning is often a manual process in model development. Manual tuning prevents a large range of parameter values as well as parameter combinations from being explored. `octopus` is designed to enable automated hyperparameter sweeps using Weights & Biases model tracking (https://wandb.ai/).

`octopus` handles all customization through five Handler objects and a configuration file (Figure 1). Defined in a user's project code, a Dataset Handler loads customized Dataset objects for training, validation, and test phases. A Phase Handler loads defined Training, Validation, and Test phases. Each phase has a standardized API to enable easy running of a given phase, within which a user can define custom steps. A Model Handler loads custom models based on configuration. The Optimizer Handler and Scheduler Handler both work the same way for optimizers and schedules, respectively.

For logging, octopus logs to both `stdout` and a file, ensuring that relevant information is captured for diagnosis and analysis. Logging format is predefined. For environment setup, octopus creates directories for checkpointing, output, and wandb based on user provided configuration. `octopus` also defines the PyTorch device as cpu or gpu, which is required for determining when to move the model and data to the GPU. `octopus` also handles deleting old checkpoints if specified, allowing the user to avoid focusing on manual effort during training.

For model tracking, `octopus` uses Weights & Biases (`wandb`), an API that delivers metrics and model performance statistics to a dashboard. `octopus` assumes the `wandb` package is loaded into the environment, but also provides a script which can be run first to handle the one-time installation and log in process. `wandb` tracking for the model is initialized according to user-provided configuration. For data loading, `octopus` uses a dataset handler provided by the user to load all required data into PyTorch DataLoader objects. This allows the user to fully customize all aspects of the Dataset objects for a specific problem. Users control DataLoader settings through the configuration file, as well. To enable hyperparameter sweeps, all configuration values in the framework take individual arguments. This enables each parameter to be set during a sweep.

For model and component initialization, `octopus` uses a Model Handler, Optimizer Handler, and Scheduler Handler provided by the user to generate one or more models with matching optimizers and schedulers. After initialization, `octopus` loads the components onto the GPU and adds the models to `wandb` tracking. For pipeline setup, the Phase Handler provided by the user to load training, validation, and test phases. Pipeline setup automatically handles model checkpoint saving and loading, metric gathering and tracking to `wandb`, and updating all schedulers after each epoch. `octopus` manages running the pipeline as well. Finally, cleanup steps like disconnecting from `wandb` are performed automatically by the framework.

Regarding evaluation, the framework is considered robust if it can handle multiple different types of models without any changes to the framework code. The framework is also considered a success if it allows `wandb` sweeps to be used against all framework configurations. If `octopus` is successful, other users can leverage its standardized approach to reduce the amount of code that must be written to run a new model. Model development speed should increase as well, given that users can focus on the customized code, rather than the standard processes used by all models.

3.2. ConcatenationFCN - parameterizing generator networks

Fully Convolutional Networks (FCNs) are a common deep learning model used to perform image segmentation. FCNs take a raw image and break that image down into patterns. In the case of gland images, these patterns can be used to generate a segmentation probability map which classifies individual pixels as either segment or background. Numerous FCN architectures have been proposed for image segmentation. This work focuses on extracting the common patterns within two architectures (DAN, Zheng et al. 2017; DCAN Chen et al. 2016) and developing a parameterized architecture that can be tuned for novel datasets.

Based on analysis of DAN and DCAN (Figure 2), the general pattern of an FCN for image segmentation is as follows:

1. Input block
2. One or more FCN blocks
3. Upsampling blocks
4. Combination of upsampling output
5. Mapping block

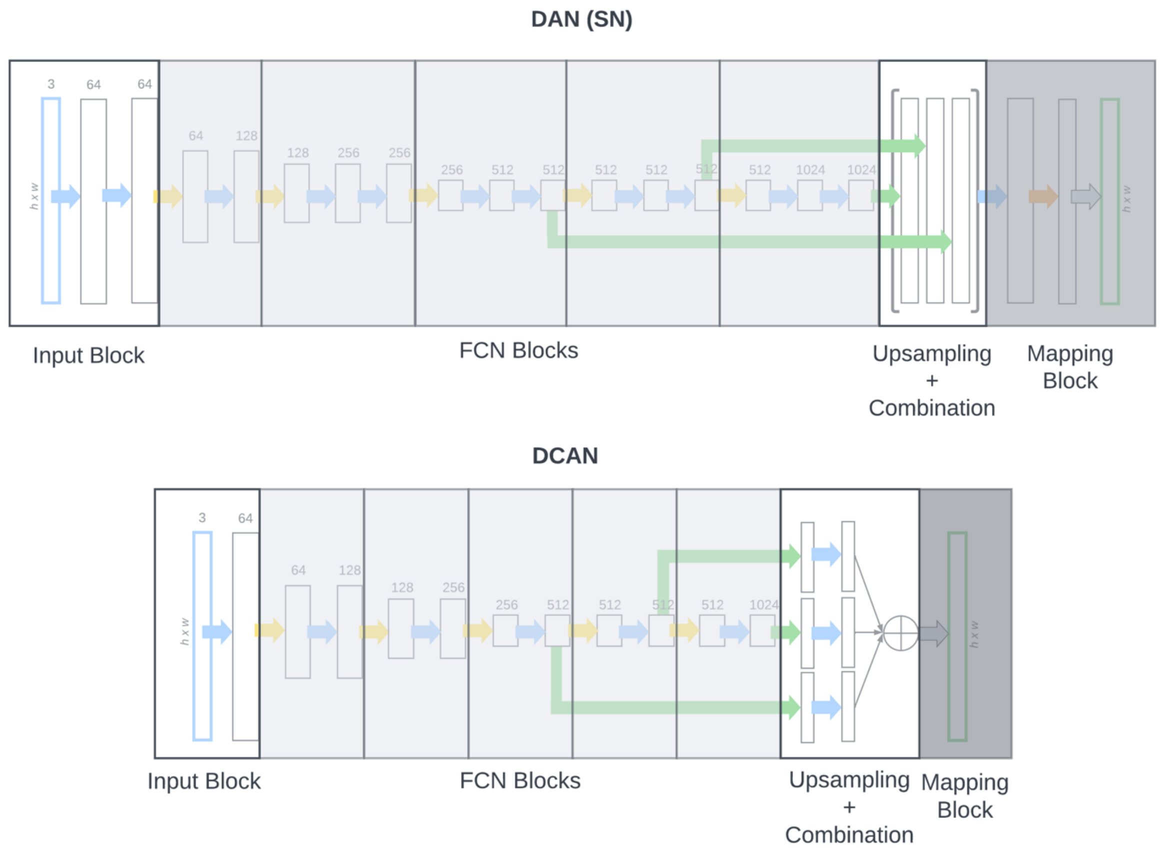


Figure 2: Diagram of FCN pattern for the segmentation networks from DAN and DCAN

The input block consists of one (DCAN) or more (DAN) CNN layers. The first CNN layer has input channels equal to the number of channels in a raw image. The number of output channels is flexible and should be optimized for the problem at hand. The number of channels remains the same for all CNN layers in the input block. Each CNN layer is followed by batch normalization and activation layers. Both DAN and DCAN use 64 output channels and a 3 x 3 kernel.

An FCN block starts with a pooling layer, followed by one or more groups of CNN, batch normalization (BN), and activation (ReLU) layers. Both DAN and DCAN use max-pooling. Placing the pooling layer at the start of an FCN block allows the output of the block to be used directly for upsampling. The number of output channels for each CNN either remains the same as the previous layer in the group or doubles in number. FCN blocks in DCAN have two groups, and the output channels double between the first group and second group (except for the second-to-last FCN block, which uses the same number of output channels). FCN blocks in DAN are less consistent. The first block has two groups; all additional blocks have three groups. Within each group, the second CNN has twice as many output channels as the first, then uses the same number of output channels for the third CNN. The second-to-last FCN block has all three CNN layers with the same number of output channels.

Each upsampling block receives input from one FCN block in the model and performs a deconvolution to upsample the current image size back to its original input size. The original image size is required so that every pixel in the image can be classified as either segment or background. Both DAN and DCAN models have three upsampling blocks, with each receiving input from one of the last three FCN blocks. DAN upsampling blocks consist of a single deconvolutional layer followed by BN and ReLU. DCAN upsampling blocks include an additional CNN layer after upsampling. (This additional layer performs classification into segments or background, bypassing the need for a separate mapping block.) Next, the model combines the output of the upsampling blocks. DAN concatenates the channels of its three upsampling blocks. DCAN sums the channels of its three upsampling blocks.

The goal of the mapping block is to produce a pixel-wise classification into segments or background. The mapping block for DAN consists of one standard CNN layer (3 x 3 kernel, BN, ReLU), a 1 x 1 kernel CNN layer, and softmax. The mapping block for DCAN is a single softmax layer.

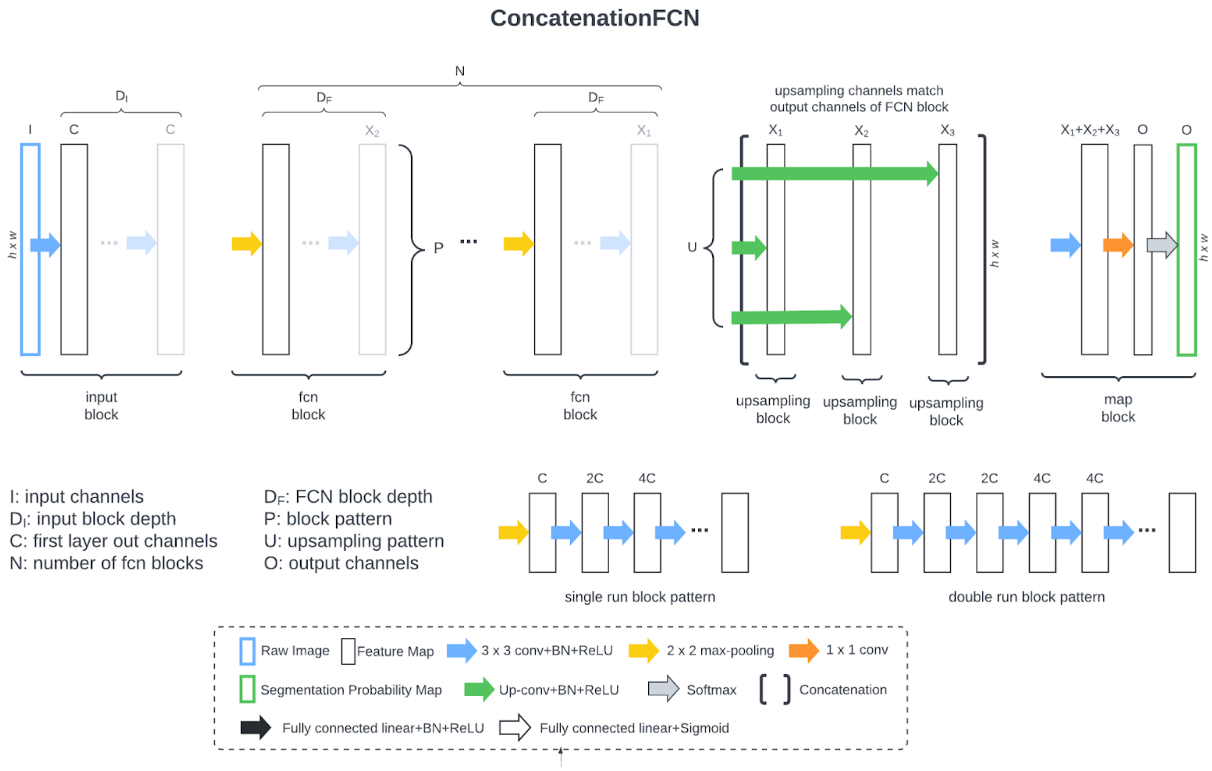


Figure 3: Diagram of ConcatenationFCN parameterized architecture for GAN generator models for image segmentation

ConcatenationFCN (Figure 3) abstracts and parameterizes these patterns so model architecture can be tuned as easily as the learning rate. All standard CNN layers have 3 x 3 kernel, stride of 1, and padding of 1. All pooling layers are set as 2 x 2 max-pooling with stride 2, padding 1, dilation 1.

Three parameters control the input block: (I) is the number of input channels; (C) is the number of output channels for the first CNN layer; (D_I) is the depth of the input block, or the number of (CNN, BN, ReLU) groups in the block. All CNN layers in the input block have the same number of output channels.

Three parameters control the FCN blocks: (N) is the number of FCN blocks; (D_F) is the depth of each FCN block; (P) is the FCN block pattern. Block pattern options can be extended, but currently consist of the following options:

- Single: the next CNN layer in the group has twice as many output channels as the previous
- Double: number of output channels doubles for every

even layer (i.e. the second CNN layer has twice as many output channels; the third CNN layer has the same as the second; the fourth CNN layer has twice as many as the third, etc.)

The number of upsampling blocks is fixed at three, although this could be extended in the future. For the ConcatenationFCN, each upsampling layer consists of a single `torch.Upsample` layer with size determined by the input size of the raw image. Upsample layers use bilinear interpolation. A single parameter controls the upsampling block pattern: (U). A single upsampling pattern (“last three”) is currently implemented, although this could also be extended in the future. "Last three" results in each of the last three FCN layers connecting to one of the three upsampling blocks. The ConcatenationFCN concatenates the layers of the upsampling blocks together and passes this to the mapping block.

The mapping block has a single parameter: (O) is the number of output channels of the model. Similar to DAN, the mapping block has one 3x3 CNN (followed by BN, ReLU), one 1x1 CNN (without BN or ReLU), and softmax.

Model architecture flexibility was tested by performing a hyperparameter sweep over all architecture parameters. Training loss, Validation loss, Intersection-over-Union (IOU) score, and pixel-wise accuracy were measured and compared against a baseline model (DAN). Note that due to memory limitations, images were downsampled from 775 x 522 to 332 x 224 before training. Models used a batch size of 2 due to memory limitations of DAN.

The loss function was defined as Multi-Class Cross Entropy (MCE), following Zhang (2017):

$$\ell(\theta_S) = \sum_{m=1}^{M} \ell_{mce}(S(X_m), Y_m) \tag{1}$$

where

X_m: annotated image

Y_m: target segmentation map

$S(X_m)$: segmentation probability map of raw image X_m

ℓ_{mce}: multi-class cross entropy loss

$\ell(\theta_S)$: loss between predicted map and target (actual) map

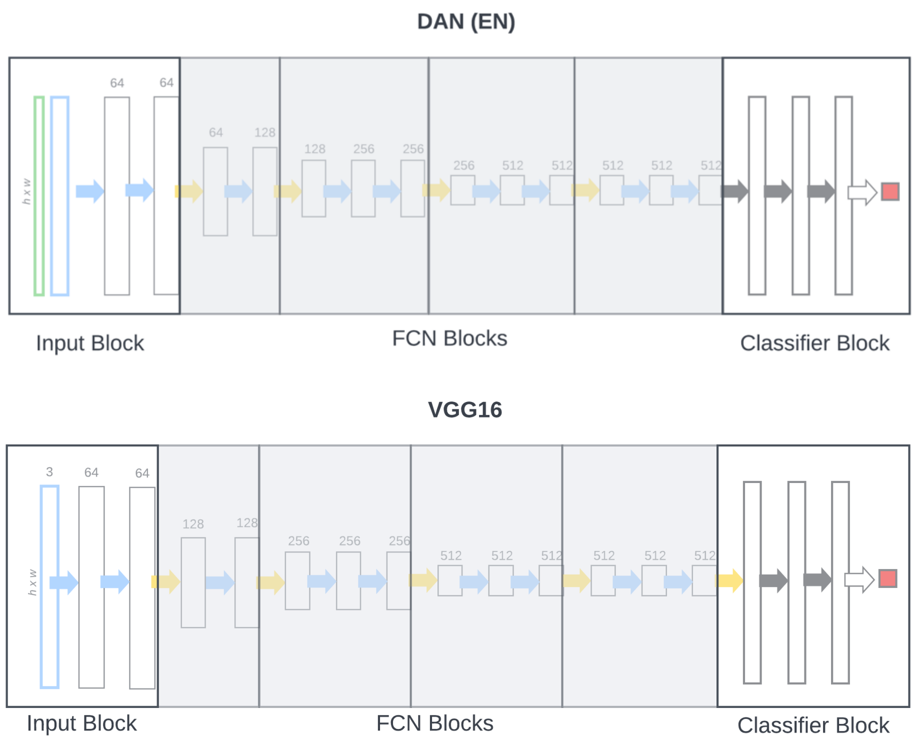


Figure 4: Diagram of model architecture pattern for the evaluation network from DAN and VGG16

3.3. FlexVGG - parameterizing discriminator networks

For image segmentation using a GAN, probability map output from the generator is used by a discriminator model to classify the image into a real segmentation map annotated by researchers or a fake segmentation map generated by the segmentation network (SN). A number of architectures have been proposed for the discriminator model inside GAN image segmentation deep learning networks. This work focuses on extracting the common patterns within two architectures (DAN, Zheng et al. 2017; VGG16 Simonyan & Zisserman 2014) and developing a parameterized architecture that can be tuned for novel datasets.

Based on the discriminator, otherwise known as an evaluation network (EN) in DAN and VGG16 (Figure 4), the general pattern of a classification VGG is as follows:

1. Input block
2. One or more FCN blocks
3. Classifier block

Input blocks consist of one or more CNN layers, with the first CNN layer receiving input and all remaining CNN layers having the same number of output channels as the first. The EN in the DAN uses as input a combination of the raw image and segmentation probability map generated by the SN. A CNN layer receives this input and outputs 64 channels. The EN has a second CNN layer with matching channels. VGG16 has an input block with two CNN layers with 64 channel output.

Each FCN block starts with a pooling layer, followed by one or more groups of CNN, batch normalization (BN), and activation (ReLU) layers. EN and VGG16 use max-pooling. The number of output channels in a layer for a group either doubles or remains the same as the previous layer. For EN blocks, the number of output channels doubles for the second layer, then remains the same if there is a third layer. The last block in the EN has all output channels the same. For VGG16 blocks, each layer in a block contains the same number of output channels, and each block starts with twice as many channels as the previous block (except the last block). VGG16 has an additional pooling layer between the FCN blocks and the classifier block. FlexVGG does not include this additional pooling layer.

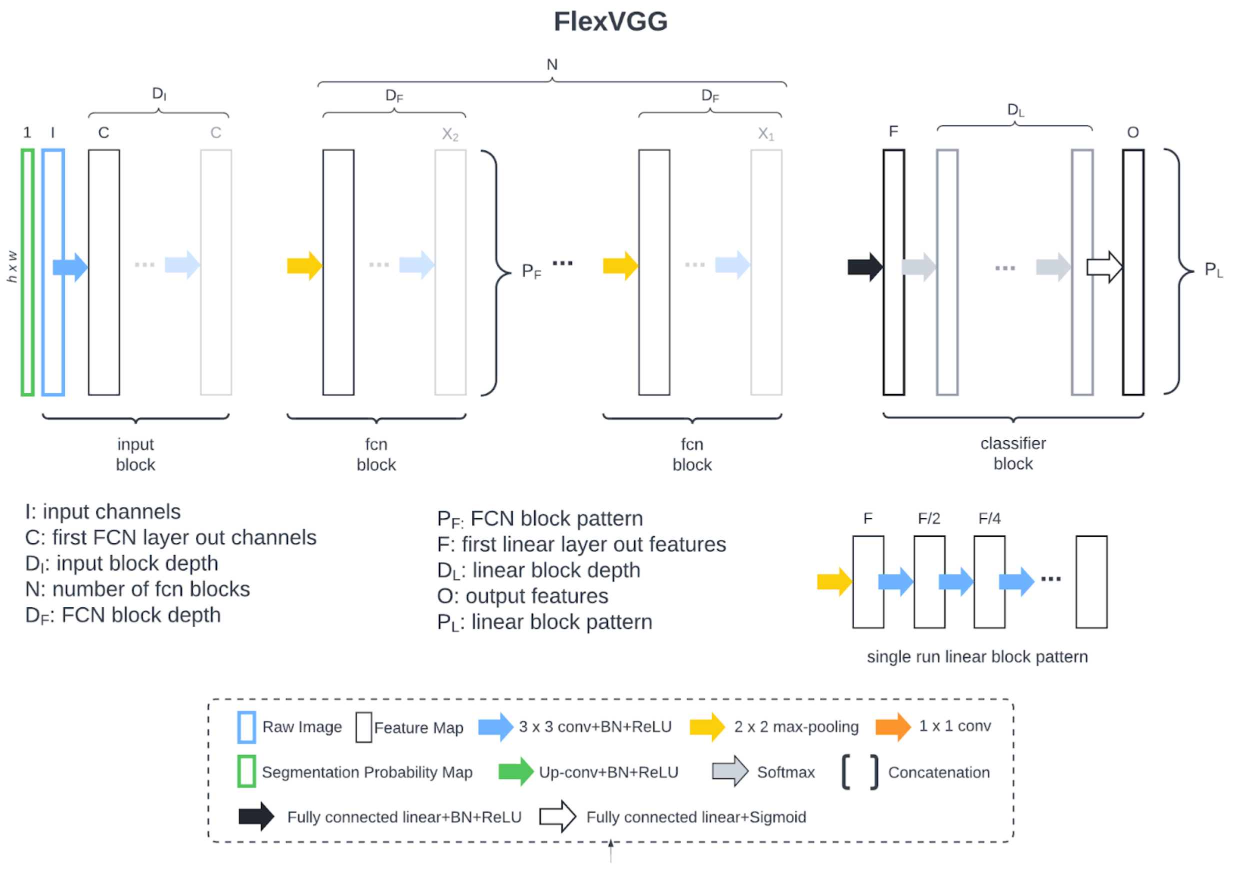


Figure 5: Diagram of FlexVGG parameterized architecture for GAN discriminator models for image segmentation

A classifier block flattens input from the FCN blocks then uses a series of one or more fully-connected linear layers (linear + BN + ReLU) to take the FCN input and classify it into one or more classes. For EN, there are two classes (real or fake). For the original implementation of VGG16, there are 1000 classes. The EN finishes the classifier block with a sigmoid activation layer to guarantee output will be between 0 and 1. VGG16 finishes with a softmax to ensure all class probabilities add to 1. These patterns have been parameterized as FlexVGG (Figure 5).

Model architecture flexibility was tested by performing a hyperparameter sweep over all architecture parameters as part of the GAN training. Discriminator accuracy, Training loss, Validation loss, Intersection-over-Union (IOU) score, and pixel-wise accuracy were measured and compared against the DAN EN. Both the DAN EN and FlexVGG architecture used the same SN to make comparison possible. Note that due to memory limitations, images were downsampled from 775 x 522 to 332 x 224 before training. Discriminator learning rate was set at 1/10 of the learning rate of the generator. The adversarial process began at Epoch 1 and continued throughout training. Models used a batch size of 4.

The loss function was defined as Binary Cross Entropy (BCE), following Zhang (2017):

$$\ell(\theta_S, \theta_E) = A + \lambda[B + C] \tag{2}$$

where

$$A = \sum_{m=1}^{M} \ell_{mce}(S(X_m), Y_m) \tag{2.1}$$

$$B = \sum_{m=1}^{M} \ell_{bce}(E[S(X_m), X_m], 1) \tag{2.2}$$

$$C = \sum_{n=1}^{N} \ell_{bce}(E[S(U_n), U_n], 0) \tag{2.3}$$

X_m: annotated image
Y_m: target segmentation map
U_n: unannotated image
λ: importance weight of discriminator loss
ℓ_{mce}: multi-class cross entropy loss
ℓ_{bce}: binary cross entropy loss
$S(X_m)$: segmentation probability map of raw image X_m
$E[S(X_m), X_m]$: score for segmentation probability map and raw image X_m (1 if annotated, 0 if unannotated)
$\ell_{bce}(E[S(X_m), X_m], 1)$: Loss between predicted annotated and actual annotated

For this method, the importance weight of discriminator loss was defined as follows:

$$\lambda = 0.1 + \frac{epoch}{100} \tag{3}$$

3.4. Tuning GAN model relationships

GAN performance depends on more than simply an optimal generator and optimal discriminator. To improve the network beyond what is possible by training each individual component, the relationship between the two models must be optimized. This relationship can be defined via the following hyperparameters:

- L_D: Learning rate of the discriminator, which defines the ratio of learning rates between generator and discriminator.
- λ: Importance weight of discriminator loss when calculating overall GAN loss.
- I_λ: Increment that lambda is increased per training epoch.
- S_{GAN}: Epoch at which the adversarial process begins. (Before this epoch, only the generator model is being trained.)

Optimizing these hyperparameters should lead to a higher performing GAN. However, the literature reviewed does not focus on these relationships. DAN (Zhang 2017) initializes importance weight at 0.1, reaching 1.0 after 30000 iterations. For DAN, the generator and discriminator were assumed to have the same initial learning rate.

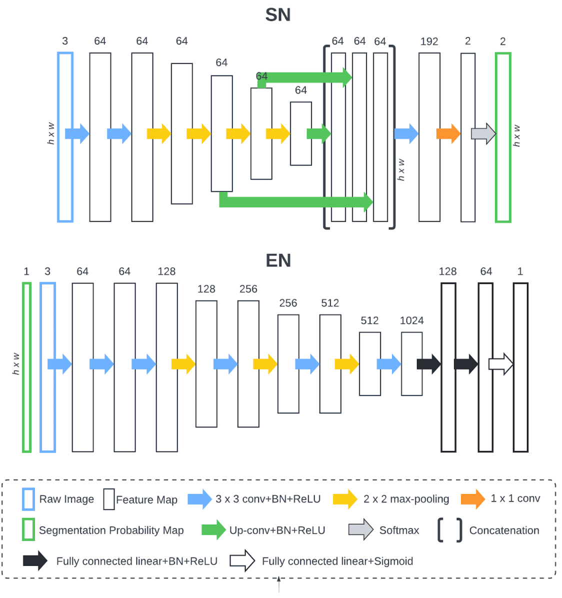


Figure 6: Architectures defined for the network which is used during GAN optimization

A sweep was performed for these hyperparameters after fixing the network architecture for ConcatenationFCN and FlexVGG based on optimal configurations from previous experiments (Figure 6). Learning rate for the discriminator was optimized in one set of experiments; importance weight and weight increment were optimized in a second set of experiments; GAN starting epoch was optimized in a third set of experiments. Loss function was defined the same as in Method 3.3. For this method, the importance weight of discriminator loss was defined as follows:

$$\lambda = \lambda_0 + \frac{epoch}{I_\lambda} \tag{4}$$

3.5. Contrastive loss with negative hard mining

It may be possible to improve GAN performance if we guide the deep learning network with additional loss functions. Contrastive loss is a common loss function which is used in image segmentation in addition to cross entropy loss. Contrastive loss can either be used as part of training or for pretraining the model. Both approaches are considered in this method. Contrastive loss is a general term to refer to loss functions that consider contrasting examples to help models learn. This type of loss can be implemented in different ways: One approach to contrastive loss takes pairs of images from different classes and uses them to train models to identify features which distinguish between the two. Another approach to contrastive loss selects pairs or triplets of pixels within a single image that belong to different classes (and in the case of triplets, two belong to the same class). When triplets are used, this approach is known as triplet loss. The approach used in this method is triplet loss.

Triplet loss requires the definition of triplets. Each triplet has three members: an anchor, a positive, and a negative. The anchor pixel is from one class; the positive pixel is from the same class; the negative pixel is from a different class. One approach to selecting negative pixels is known as negative hard mining. With negative hard mining, the goal is to choose the most difficult negative pixel for the model to distinguish from the anchor. In the case of background/foreground image segmentation, hard negatives are simply the closest background pixel to the chosen foreground pixel.

We hypothesized that adding triplet loss with negative hard mining to cross entropy loss for the generator model will lead to higher performance due to the model learning the difference between background and foreground pixels. Higher performance should translate to better accuracy and a higher IOU score than a baseline model.

Triplets were identified using the following approach. For each triplet pair desired from a given image:

1. From the foreground pixels which have not yet been selected as anchors, select a random pixel as anchor.
2. Find the closest background pixel to this anchor to be the hard negative. The search radius starts 1 pixel from the anchor (i.e. neighboring pixels), and increases by 1 pixel if no background pixels are found at that radius. If multiple background pixels are found at a given radius, one is chosen at random. The search radius increases until a background pixel is found (Figure 7).
3. For the positive, select any random foreground pixel which is not the current anchor.

This approach is linear in the number of triplets, but empirical results indicate it increases runtime by 20x from a baseline GAN model. Triplet margin loss is defined using Komorowski (2021):

$$L(a_i, p_i, n_i) = max \begin{cases} d(a_i, p_i) - d(a_i, n_i) + m \\ 0 \end{cases} \tag{5}$$

where

a_i, p_i, n_i: embeddings of anchor, positive, negative pixels, respectively

m: margin hyperparameter

$d(x, y) = \|x - y\|_2$: Euclidean distance between embeddings

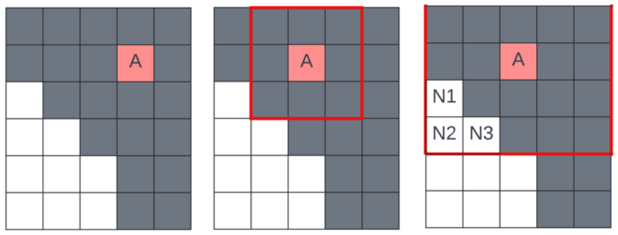


Figure 7: Diagram of search process for hard negatives of a given anchor A. Background pixels are white in this example. In this example, any of N1, N2, or N3 pixels can be chosen as the hard negative. Any of the segment pixels which are not the anchor can be chosen as the positive.

Two different uses cases for contrastive loss were explored: (1) Use triplet loss as a supplement to cross-entropy loss during training; (2) use triplet loss as a pretraining step. When considering triplet loss as a supplement, the work introduced an additional hyperparameter λ_2 to control the importance weight of triplet loss within the overall model loss. When only the generator is trained, we have:

$$\ell(\theta_S, \theta_E) = A + \lambda_2 D \tag{6}$$

where

$$D = \sum_{m=1}^{M} \sum_{i=1}^{T} L(a_i^m, p_i^m, n_i^m) \tag{6.1}$$

λ_2: importance weight of triplet loss

T: number of triplet pairs per annotated image

a_i^m: ith anchor for the mth training image

When the discriminator is being trained as well, triplet loss is added to total loss in the following manner:

$$\ell(\theta_S, \theta_E) = A + \lambda_1[B + C] + \lambda_2 D \quad (7)$$

When triplet loss is used for pretraining, we have:

$$\ell(\theta_S) = \sum_{m=1}^{M} \sum_{i=1}^{T} L(a_i^m, p_i^m, n_i^m) \quad (8)$$

The network was evaluated on the same dataset as previous methods, and hyperparameters were optimized using a grid search.

4. Experiments

4.1. Dataset

All experiments for this work utilize the 2015 MICCAI Gland Challenge dataset. The dataset comes from the University of Warwick, which contains 85 training images, 60 validation images, and 20 test images of gland tissue. Images were resized to ensure every image was 775 x 522 then resized to 332 x 224 (bilinear interpolation) to prevent memory issues. Raw images for triplet loss experiments were normalized using mean and standard deviation values calculated from all 165 images in the dataset; raw images were not normalized for other experiments. Training data used random horizontal flip and random vertical flip transformations. Dataset: https://warwick.ac.uk/fac/cross_fac/tia/data/glascontest/download/.

4.2. ConcatenationFCN hyperparameter tuning

ConcatenationFCN was optimized over the 2015 MICCAI Gland Challenge dataset using validation loss and validation IOU score as performance metrics. Models were run on a Tesla T4 GPU with 8 CPUs. Hyperparameter tuning was subdivided into several groups and a grid search performed over each group.

4.2.1. Hyperparameter Set 1 (sweep-001)

To summarize the results: (1) models with the same number of input and output channels for all CNN layers performed well and allow for very deep models (Figure 8); (2) more FCN blocks was better, with performance not improving beyond 5; (3) "Double Run" block pattern reduced memory but didn't improve performance.

Table 1: Hyperparameter Set 1

Hyperparameter	**Range Tested**
num_fcn_blocks	3,4,5,6
block_depth	1,2
first_layer_out_channels	32,64
block_pattern	single_run, double_run

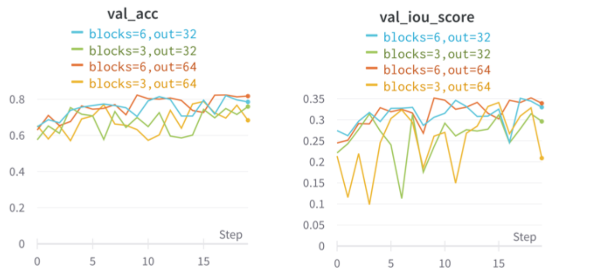


Figure 8: ConcatenationFCN Hyperparameter Set 1 sweep results (showing double_run)

4.2.2. Hyperparameter Set 2 (sweep-002)

To summarize the results: (1) adam outperformed sgd; (2) lr=0.001 outperformed other rates; (3) models built out of a single channel size performed very well and used little memory.

Table 2: Hyperparameter Set 2

Hyperparameter	**Range Tested**
lr	0.1,0.01,0.001,0.0001,0.00001
optimizer_type	adam, sgd
block_depth	1, 2
first_layer_out_channels	8, 16, 32
block_pattern	single_run, double_run

4.2.3. Optimal generator architectures

According to the metrics, several ConcatenationFCN architectures (Figures 9-10) match the performance of DAN (Table 3) while using significantly less memory and runtime. Test images were computed for all architectures as well (Figure 11).

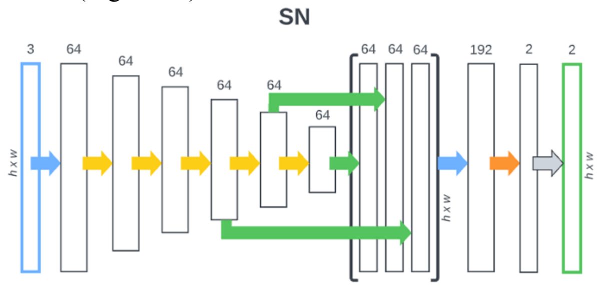


Figure 9: ConcatenationFCN-03 optimized architecture

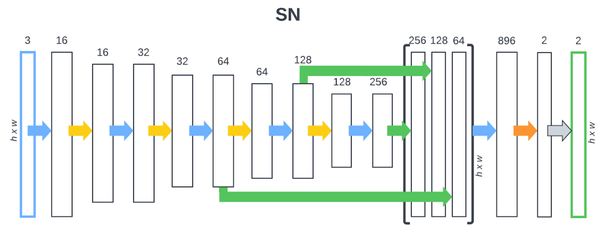


Figure 10: ConcatenationFCN-05 optimized architecture

A parameterized FCN architecture allows the model architecture itself to be easily and automatically tuned to the problem at hand, rather than approach architecture design manually. For image segmentation, parameterized FCN discovered multiple architectures that performs as well as published architecture while significantly reducing memory usage and runtime. Less memory usage means that batch size can be increased, reducing training time. However, in an effort to keep model generation as simple as possible, ConcatenationFCN does not parameterize every possible aspect of model architecture. The assumptions, such as limiting the number of upsampling layers to three or requiring that the FCN block pattern be applied to every FCN layer, may lead to suboptimal performance for some problem types.

Table 3: Runtime Metrics at Epoch 100 for comparison between DAN and ConcatenationFCN architectures

Metric	**DAN SN**	**ConcatenationFCN-003**	**ConcatenationFCN-005**
Train loss	**0.3469**	0.3549	0.3485
Validation Loss	**0.412**	0.4354	0.4283
Validation IOU Score	**0.8152**	0.7698	0.7843
Validation Accuracy	**0.987**	0.8715	0.8793
Runtime (s/epoch)	172	**9**	83
GPU Memory % (max)	94	**23**	53

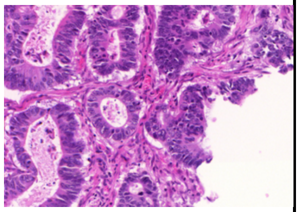
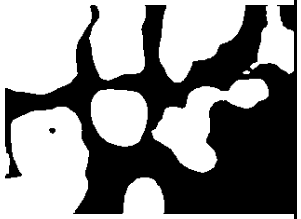
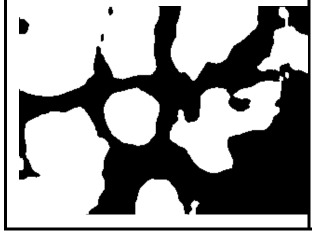
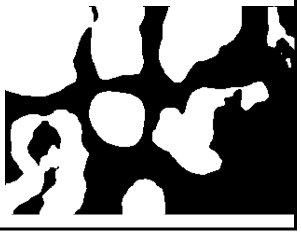
Raw Image	DAN SN
	
ConcatenationFCN-03	ConcatenationFCN-05
	

Figure 11: Test image results for DAN and ConcatenationFCN architectures

4.3. FlexVGG hyperparameter tuning

FlexVGG was optimized over the 2015 MICCAI Gland Challenge dataset using validation loss and validation IOU score as performance metrics. To make the comparison fair, both the DAN EN (Zhang 2017) and FlexVGG were run as part of the GAN using one of the optimized ConcatenationFCN architectures (ConcatenationFCN-03). Models were run on a Tesla T4 GPU with 16 CPUs. Both SN and EN start with the same learning rate (0.001).

Hyperparameter tuning was subdivided into the following steps, with a grid search performed over all hyperparameters for that step:

4.3.1. Hyperparameter Set 1 (sweep-004)

To summarize the results: (1) The best performance results from using a deeper model with a small start output channel size (64), block depth of 2, and a "Double Run" block pattern.

Using a smaller number of FCN blocks leads to memory issues because this results in a very large number of features when flattening into a linear layer. The size of the flattened layer is based on the number of CNN out channels and dimensions of the current image size, which essentially halves for every pooling layer. Using a smaller number of FCN blocks means the current image size is close to original, resulting in a large number of features.

Using a large start output channel results in the final CNN out channel very large (because the number of channels doubles in each block), also resulting in a large number of features. "Double Run" ensures the number of channels doesn't increase too quickly. Using 4 FCN blocks with a depth of 2 fits within memory; (2) one linear layer (apart from flatten) is sufficient for performance; (3) linear layer is limited to a 128-length feature vector based on memory capacity.

Table 4: Hyperparameter Set 1

Hyperparameter	**Range Tested**
num_fcn_blocks	4,5,6
depth_linear_block	1,2,4
first_linear_layer_out_features	128,512,1024
sigma	0.1, 0.5, 1.0
sigma_weight	30, 300

4.3.2. Optimal discriminator architecture

Hyperparameter sweeping found a FlexVGG model architecture (Figure 12) that performs as well as the EN in the DAN while reducing runtime and GPU memory usage (Table 2). Test images were computed for both architectures, as well (Figure 13).

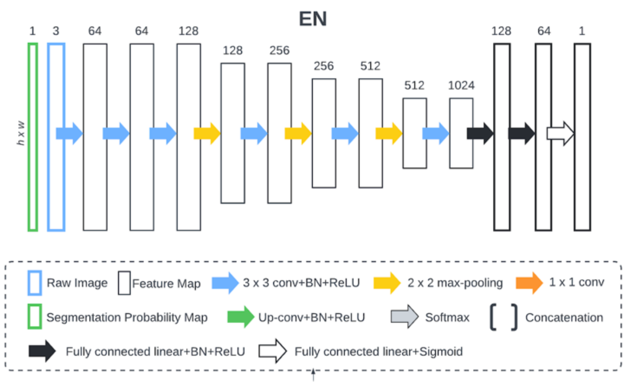


Figure 12: FlexVGG optimized architecture

A parameterized VGG architecture allows discriminator architecture to be tuned for a given problem type as easily as any other hyperparameter. This can enable faster model development for GAN and other problem types that use classification on images. However, not every part of the architecture is parameterized for FlexVGG. The assumptions that are made limit the applications of the model (i.e. every FCN block is being treated the same regarding the block pattern; the number of out features of every linear layer being defined as half of the previous layer).

Table 5: Runtime Metrics at Epoch 100 for comparison between DAN and FlexVGG architectures

Metric	**DAN EN**	**FlexVGG**
Train loss (generator)	**0.8329**	0.9575
Validation Loss (generator)	**0.4521**	0.4579
Validation IOU Score (generator)	0.7401	**0.7407**
Validation Accuracy (generator)	0.8477	**0.853**
Runtime (s/epoch)	14	**13**
GPU Memory % (max)	44	**35**
Train loss (discriminator)	1.503	**1.431**
Train loss (discriminator, unannotated)	1.094	**0.9059**
Train loss (discriminator, annotated)	**0.4086**	0.5249

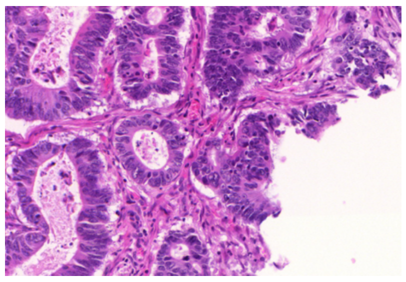
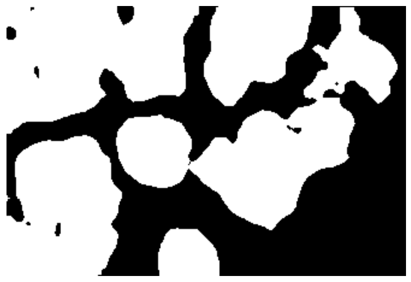
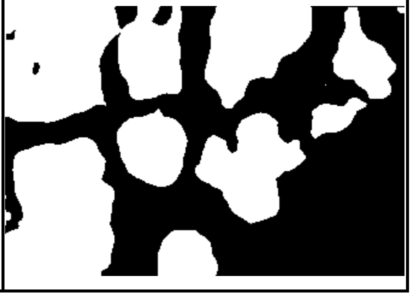
Raw Image	ConcatenationFCN-03 + DAN EN	ConcatenationFCN-03 + FlexVGG
		

Figure 13: Test image results for DAN and FlexVGG architectures

4.4. GAN Relationship Optimization

4.4.1. Discriminator learning rate (Learning Rate Ratio)

Learning rate ratio was optimized over the 2015 MICCAI Gland Challenge dataset using validation loss and validation IOU score as performance metrics. Two different hyperparameter sweeps were performed (Figures 14-15). With fixed model architecture and a starting generator (SN) learning rate of 0.001, experimentation found that the optimal learning rate for the discriminator (EN) is between 0.0001 and 0.001 when the adversarial process begins at Epoch 1. A learning rate for the discriminator that is 10% of the generator prevents the discriminator model from mastering its simpler classification task too quickly to allow the generator to compete.

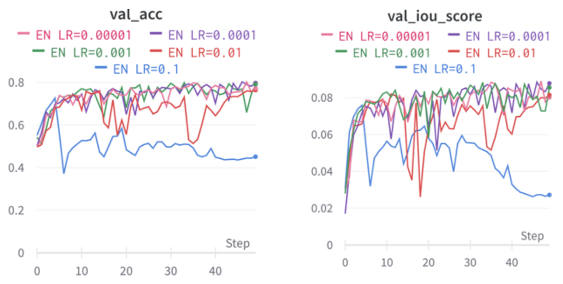


Figure 14: Sweep-008 Results for learning rate ratio optimization

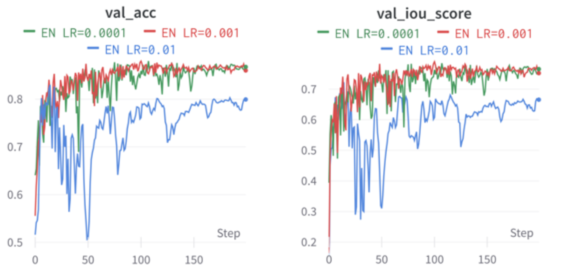


Figure 15: Sweep-009 Results for learning rate ratio optimization

4.4.2. Discriminator Importance Weight and Weight Increment

Discriminator Importance Weight (lambda) and Weight Increment (lambda_weight) were optimized over the 2015 MICCAI Gland Challenge dataset using validation loss and validation IOU score as performance metrics. Importance Weight (Figure 16) was optimized separately from Weight Increment (Figure 17). With a fixed discriminator (EN) learning rate of 0.0001, experimentation found that an importance weight between 0.05 and 0.1 produced the highest performance, although higher importance weight values produced arguably similar performance. Experimentation failed to discern an optimal weight increment, with the outcomes of all three values tested producing nearly identical performance.

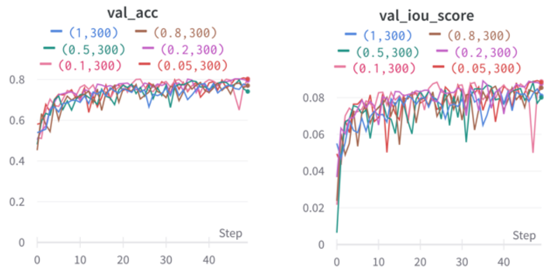


Figure 16: Sweep-007 Results for importance weight optimization

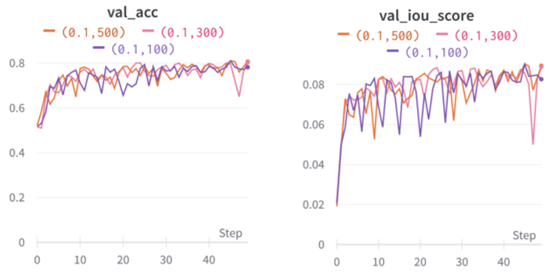


Figure 17: Sweep-007 Results for importance weight increment optimization

4.4.3. GAN starting epoch

The epoch at which the adversarial process begins (GAN start epoch) was optimized over the 2015 MICCAI Gland Challenge dataset using validation loss and validation IOU score as performance metrics (Figure 18). With a fixed discriminator learning rate, importance weight of 0.1, and weight increment of 300, experimentation found that the epoch at which the adversarial process began had little effect on validation scores, at least within the range of starts tested. Starting the adversarial process earlier (Epoch 1) allowed the generator to overcome the discriminator and improve validation scores. Additionally, the longer the adversarial process was delayed, the larger the immediate impact the process had on generator loss.

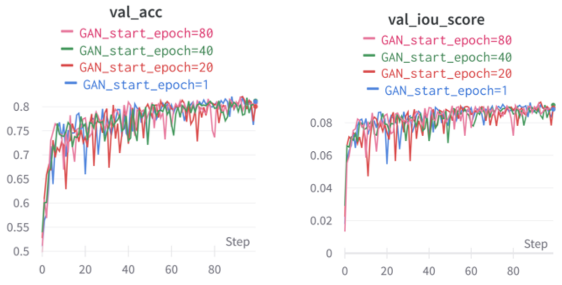


Figure 18: Sweep-006 Results for GAN start epoch optimization

Identification of optimal importance weighting strategy and learning rate for the discriminator model in a GAN allows the discriminator to benefit the training of the generator. The limitation of the method presented in this work is that the result of importance weighting or learning rate may differ depending on the model architecture for both the discriminator and generator as well as the dataset. This method assumed a fixed discriminator architecture and fixed generator architecture. The optimal relationship values which resulted from experimentation may result in suboptimal performance if different architectures are used. Additionally, this method optimized the learning rate independent of importance weight. It is possible the two hyperparameters affect each other, and that optimizing independently may result in suboptimal model performance.

4.4.4. GAN vs Generator Model only

In addition to checking different hyperparameters, we used the 2015 MICCAI Gland Challenge dataset to compare model performance of a GAN (SN plus EN) to a network with only the generator model (SN). Surprisingly, we found that model performance was nearly identical (Figure 19), with no clear statistically significant benefit when a discriminator model is added as an adversary for the generator.

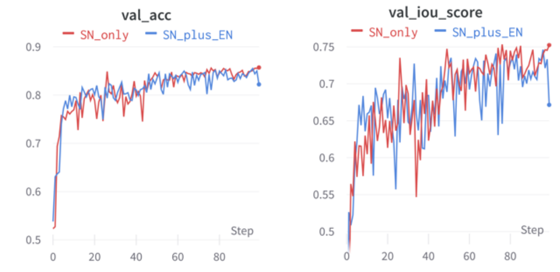


Figure 19: Model performance metrics using a GAN for image segmentation vs a generator only

4.4.5 Contrastive Loss to supplement MCE

Contrastive loss was explored on the 2015 MICCAI Gland Challenge dataset using validation loss and validation IOU score as performance metrics. The underlying network used in experiments was defined to be the ConcatenationFCN-003 + FlexVGG optimized architectures discovered in previous experiments. When triplet loss was used as a supplement to cross-entropy loss, it negatively affected performance of the model (Figure 20). Performance didn't degrade over time, but the addition of triplet loss limited the performance of the model below a model running without triplet loss, even when considering very few triplet pairs or a small importance weight.

4.4.6. Contrastive Loss to pretrain a GAN

Pretraining with contrastive loss was explored on the 2015 MICCAI Gland Challenge dataset using validation loss and validation IOU score as performance metrics. The underlying network used in the experiments was ConcatenationFCN-003 + FlexVGG optimized architectures discovered in previous experiments.

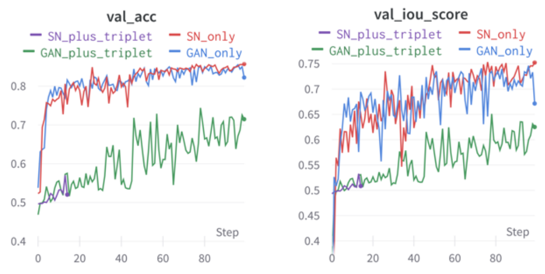


Figure 20: Metric tracking when using triplet loss to supplement model loss

The model was pretrained using 50 triplets for 100 epochs and compared against a GAN model without pretraining or triplet loss. Pretraining using 50 triplets for 200 epochs resulted in similar performance. Using triplet loss to pretrain the GAN resulted in a boost in performance for the model (Figure 21). However, this performance boost depends on the number of pretraining epochs. Pretraining using 50 triplets for 50 epochs didn't reveal improvement over a baseline model without pretraining.

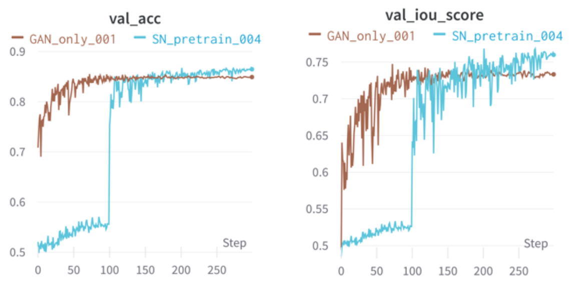


Figure 21: Metric tracking when using triplet loss to pretraining the GAN model

Identifying whether triplet loss is relevant for semantic segmentation models employing GANs will help the field determine whether to keep pursuing this type of loss or to search for a different approach. Experiment results show that triplet loss is a viable addition to GAN design.

5. Discussion

We presented a novel framework for running deep learning models, and we used this framework as the foundation for the GAN development presented in this work. A robust training framework can provide valuable benefits to machine learning researchers, especially for those just starting out in the field. Using this framework to handle the extra steps allows users to focus on model development. We identified patterns in state-of-the-art generator models and abstracted those patterns into a parameterized model called ConcatenationFCN. Demonstrating how the model can be tuned to a particular dataset, we discovered multiple architectures which performed as well as published models while using significantly less memory and runtime. We performed the same process for discriminator models, abstracting pattners into a parameterized model called FlexVGG.

The benefits of developing an approach to discover more efficient architectures may have far-reaching implications. When trying to implement published models on a limited budget for hardware, we quickly ran into memory issues. This constraint limited our analyses and required workarounds, requiring extra effort and code complexity to mitigate. Additionally, a parameterized architecture means that models can be tuned for particular datasets in novel ways. This allows models to be better optimized for a particular application.

We also analyzed the relationship between models in a GAN. The most significant and subtle findings from this analysis is that adding a discriminator adversary to challenge a segmentation network appears to have little effect on segmentation performance. If one compares the performance metrics between Methods 4.1 (Table 1) and 4.2 (Table 2), one sees that adding a discriminator model to the network resulted in worse performance at a common epoch timepoint. Given that the generator is being challenged in 4.2, it is possible its performance may be temporarily diminished or slowed. However, throughout all experimentation, the GAN model never significantly outperformed a sole generator model. This finding is worth scrutinizing - it is very possible that incorrectly chosen model architecture, GAN implementation, or hyperparameter values contributed to this result. However, if this finding holds, it could change the direction of GAN development for image segmentation applications. Finally, we demonstrated that a form of contrastive loss can be used to benefit GAN performance, and that the most effective way of doing so is to perform pretraining.

6. Future Work

A clear next step would be to challenge the finding that adding a discriminator adversary has little effect on segmentation performance. Identifying approaches to GANs which enhance segmentation performance in a statistically significant way would indicate the finding in this work does not hold. There are several avenues which have not yet been explored in this regard, including the following: (1) modification of the input into the discriminator; (2) confirmation that the implementation of GAN raining is correct in this work; (3) experimentation with larger batch sizes and larger training image sets.

In addition to focus on the GAN relationship, finding a way to optimize identifying triplet pairs is paramount for practical use of triplet loss for pretraining. The version of triplet identification from this work adds significant overhead to the model, despite being linear in the number of triplet pairs. This limits the number of triplets to a small percentage of the number of possible triplets which could be produced from an image. A more efficient version of triplet pair identification would allow pretraining to use more triplet pairs and run faster. One possible approach could be to do a single pass over each training image and identify the boundary pixels between the background and segments. Rather than search for a hard negative for each anchor, any boundary pixel on the background side could be used as a hard negative, reducing runtime complexity significantly. An additional benefit to this alternative approach would be that any boundary pixel on the segment side could be used as a hard positive. With the current approach, positives are chosen from segment pixels at random; hard positives may be able to help the generator model learn more efficiently.

Finally, although we demonstrated that pretraining with triplet loss resulted in a boost to performance, it is possible that any sort of pretraining would have had the same result. We recommend analyzing GAN pretraining using other types of loss, including cross entropy, to determine whether triplet loss or pretraining alone is the reason for performance improvement.

References

1. Hao Chen, Xiaojuan Qi, Lequan Yu, and Pheng-Ann Heng. 2016. DCAN: Deep contour-aware networks for accurate gland segmentation. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016). DOI:http://dx.doi.org/10.1109/cvpr.2016.273
2. Jacek Komorowski. 2021. MinkLoc3D: Point cloud based large-scale place recognition. *2021 IEEE Winter Conference on Applications of Computer Vision (WACV)* (2021). DOI:http://dx.doi.org/10.1109/wacv48630.2021.00183
3. Karen Simonyan, Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. 2015 ICLR *International Conference on Learning Representations* (ICLR) (2015). DOI: https://doi.org/10.48550/arXiv.1706.04737
4. Lin Yang, Yizhe Zhang, Jianxu Chen, Siyuan Zhang, and Danny Z. Chen. 2017. Suggestive annotation: A Deep Active Learning Framework for Biomedical Image Segmentation. *Medical Image Computing and Computer Assisted Intervention − MICCAI 2017* (2017), 399–407. DOI:http://dx.doi.org/10.1007/978-3-319-66179-7_46
5. Weikang Wang et al. 2019. Learn to segment single cells with deep distance estimator and deep cell detector. *Computers in Biology and Medicine* 108 (2019), 133–141. DOI:http://dx.doi.org/10.1016/j.compbiomed.2019.04.006
6. Xiangyun Zhao et al. 2021. Contrastive Learning for label Efficient Semantic Segmentation. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)* (2021). DOI:http://dx.doi.org/10.1109/iccv48922.2021.01045
7. Yizhe Zhang, Lin Yang, Jianxu Chen, Maridel Fredericksen, David P. Hughes, and Danny Z. Chen. 2017. Deep adversarial networks for biomedical image segmentation utilizing unannotated images. *Medical Image Computing and Computer Assisted Intervention − MICCAI 2017* (2017), 408–416. DOI:http://dx.doi.org/10.1007/978-3-319-66179-7_4