# Distance Loss for Improved Slit Segmentation – Project Report

Andre Ye

July 2021

## Contents

# 1 Introduction

In the podocyte slit segmentation project task, a model visually identifies the location of "slits" along the membrane. The location of these slits can then be extracted and used in the quantification of podocyte injury.

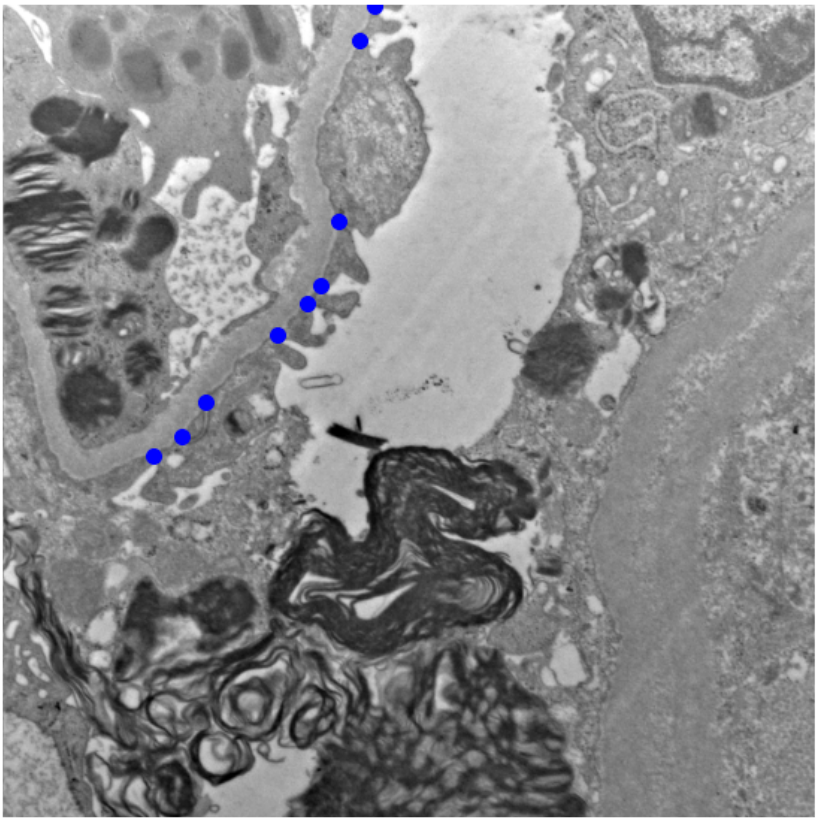

Figure 1.1: An electron microscopy cross-section with the location of slits marked in blue.

More specifically, the model takes in an electron microscopy image and outputs a segmentation map, in which "1" indicates the presence of a slit and "0" indicates the absence of a slit.

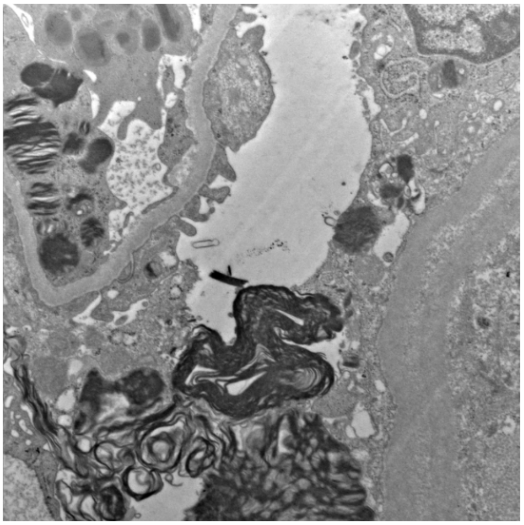

Figure 1.2: The electron microscopy image input to the model.

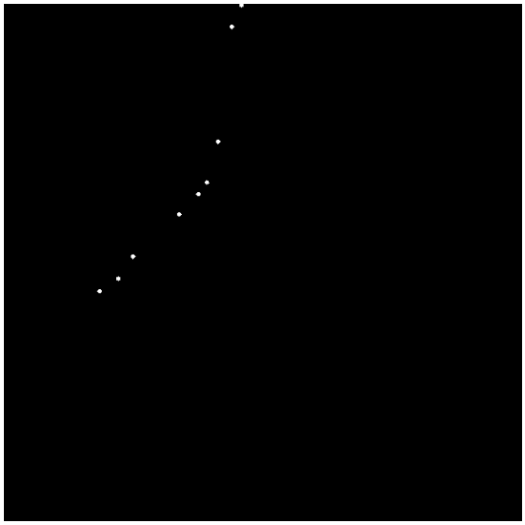

Figure 1.3: The segmentation map output of the model.

Usually, the dice coefficient is used to perform segmentation tasks. The dice coefficient is a measure of intersection over union, and can be calculated as dice $= \frac{2 \cdot \text{Area of Overlap}}{\text{Total Number of Pixels}}$. When the model perfectly predicts every pixel in the segmentation map, the dice coefficient between the prediction and the ground truth is 1, since two times the area of overlap is equal to the total number of pixels across both arrays. On the other hand, the worst possible dice coefficient is 0, in which there is no overlap at all between the prediction and ground truth arrays. Because optimizers are built to find the minimum loss function value, dice *loss* is usually formulated as $-$dice coefficient, such that the optimal dice loss is $-1$ and the worst dice loss is 0.

While dice loss suffices for most segmentation tasks, problems arise with its usage in small segmentation task problems. Dice loss quantifies the overlap between the predicted segmented region and the ground truth segmented region, but in small segmentation task problems, it is much more possible that the predicted segmented region and the ground truth segmented region do not overlap than do overlap. This leads to scenarios like in Figures 1.4 and 1.5 in which two vastly different predictions (red = prediction, blue = truth) are equivalent in goodness under the dice coefficient. The prediction in Figure 1.4 is clearly better than the prediction in Figure 1.5, because the predictions are closer to the ground truth segmented regions. However, because the predictions in Figure 1.4 don't actually overlap with the ground truth regions, the Area of Overlap component of dice loss evaluates to 0 and the dice loss itself is 0.

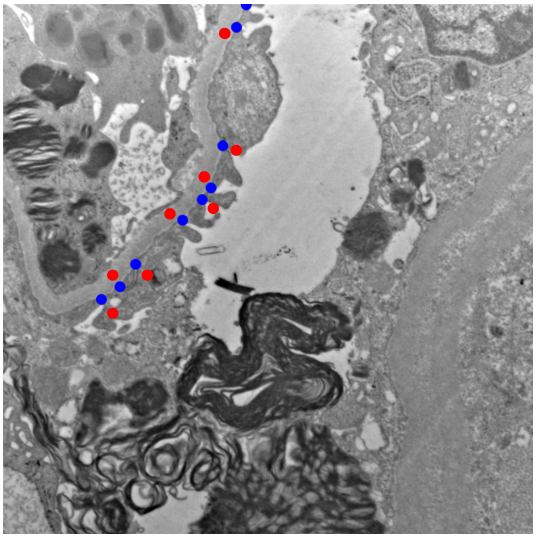

Figure 1.4: Hypothetical example of a good prediction with a dice loss of 0.

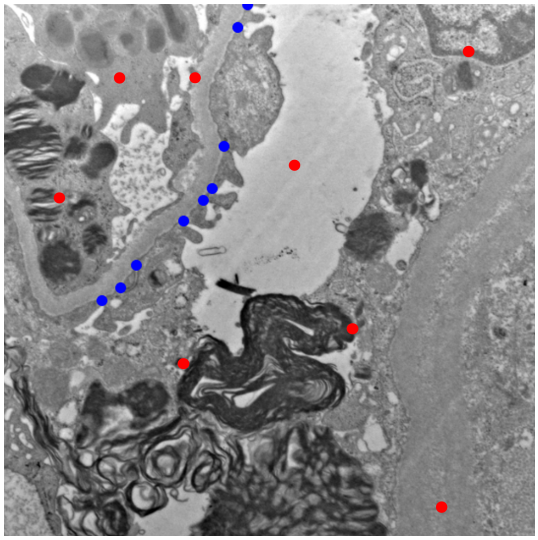

Figure 1.5: Hypothetical example of a bad prediction with a dice loss of 1.

Because of this, a model trained to perform the slit segmentation task has been observed to have erratic training behavior. The model receives a weak feedback signal for its adjustments, since any changes it makes to the position of a segmented region is more likely not to overlap with the ground truth region than to overlap with it, leading to no change in the dice loss. Thus, the model likely continually "guesses" around until one of the predicted regions happens to overlap with the true region. While the model trained with dice loss performs reasonably well, it can be hypothesized that the erratic and "hacky" nature of its learning process limits its performance potential.

The Custom Distance Loss task, then, is to attempt to improve upon the baseline dice-loss model on the slit segmentation problem by training a model on a loss function that takes into account the distance between the prediction segmentation and the ground truth segmentation to provide a richer signal that encourages more sustainable and less erratic learning. While ultimately the effort failed in creating a model with significant improvement from the dice-loss model, this report serves a summarizing purpose, documenting ideas and experiment results from the attempt.

Section 2 details the justification, formulation, and evolution of distance-based loss functions. Section 3 notes the justification and considerations of using a pretrained model as a starting point for custom distance loss experiments. Section 4 enumerates various model training procedures used to encourage model understanding and traversal of the custom distance loss. Section 5 proposes explanations for broad observed phenomena that inhibited the success of the attempt, and justifies the conclusion of the project to improve the baseline model via distance-based loss function. Lastly, Section 6 suggests directions for further exploration.

# 2 Loss Function Designs

## 2.1 Vanilla Distance Loss

The vanilla distance loss design ('vanilla' – basic) seeks to measure the average distance between the points on the predicted segmentation map and the ground truth segmentation map. It can be roughly articulated algorithmically with the following key components:

**Algorithm 1:** Key components of the vanilla distance loss design.

```
Set sum of average distances to 0;
for item in batch do
    Obtain EM image and associated ground truth segmentation map for item;
    Obtain model prediction for segmentation map on that EM image;
    Obtain Ground Truth (a_x, a_y) and Prediction (a_x̂, a_ŷ) arrays of coordinates for detected point locations;
    if no detected points in prediction then
        return maximum distance;
    end
    Match (a_x, a_y) and (a_x̂, a_ŷ) such that their average distance is minimized;
    Add the average distance between (a_x, a_y) and (a_x̂, a_ŷ) to the sum of average distances;
end
Divide the sum of average distances by the number of items;
return the mean average distance between prediction and truth across all items in batch;
```

In order to obtain the Ground Truth and Prediction arrays of coordinates from the respective segmentation maps for detected point locations, three blob-detection algorithms were considered: Laplacian of Gaussian, Difference of Gaussian, and Determinant of Hessian.

- *Laplacian of Gaussian.* Computes the Laplacian of Gaussian-smoothed images; this derivative filter allows for the detection of rapid change (edges of blobs) in images. While it is very accurate, it is also very slow. It also requires a lot of fine-tuning.
- *Difference of Gaussian.* This is a faster approximation of the Laplacian of Gaussian approach.
- *Determinant of Hessian.* Detects maxima in the determinant of Hessian of the image. It suffers from inaccurate detection of smaller blobs less than three pixels but is significantly faster than Laplacian of Gaussian or Difference of Gaussian techniques.

Because Determinant of Hessian was the fastest approach, it was used for the majority of experiments. Laplacian of Gaussian and Difference of Gaussian methods were also tested; Laplacian of Gaussian yielded poor results, likely because it dramatically slowed training speed, and Difference of Gaussian (with the right tuning) performed the same as Determinant of Hessian.

Once the $x$ and $y$ coordinates of both the ground truth and prediction segmentation maps are obtained, they are passed through a Chamfer distance function, which finds the mapping between two sets of coordinates that minimizes their average distance. If one set of coordinates is empty (i.e. the model predicts no recognizable points), the custom distance loss simply returns the maximum distance length ($\sqrt{2} \cdot$ image length). Another concern is a discrepancy in the length of the two coordinate sets – the prediction and ground truth sets of coordinates are almost always not equal in length, since the the model usually either under-predicts or over-predicts the number of points that actually exist in the ground truth segmentation map. To accommodate for this, the implementation of Chamfer distance used in this context accepts non-bijective mappings. That is, if the model under-predicts the number of slits, for instance, the Chamfer distance function will map multiple ground-truth slits to one predicted slit such that the average distance is minimized, regardless of the difference in number of slits.

The loss function is calculated for each batch in the training process; thus, the distance loss used in training does not return only the average distance between predicted and ground truth coordinates for one training item (i.e. one EM image and its associated segmentation map), but the *average* average distance across several training items. This is referred to as the mean average distance for clarity.

The Vanilla Distance Loss design is a raw reflection of the intention behind a loss function that considers distance, rather than the the intersection, between the true and predicted segmented regions. However, in training, the Vanilla Distance Loss design does not perform well, and additional aids are required to increase model performance. While the model is trained on a different, more complex loss function in later experiments, Vanilla Distance Loss still serves as an important metric for obtaining a measure of goodness better than the dice coefficient.

Implementing the Vanilla Distance Loss function in the TensorFlow framework initially appeared to be difficult, and perhaps impossible. Loss functions are conventionally implemented using native TensorFlow or Keras mathematics/backend functions, which process tensors efficiently and without friction. Moreover, this allows for easier differentiation. However, the Vanilla Distance Loss design – an all following designs – involve very complex, procedural operations that cannot be represented wholly with TensorFlow-native operations. The function eventually was implemented using TensorFlow's `tf.numpy_function()` function, which wraps a function that manipulates NumPy arrays as a TensorFlow operation. While this solution suffices in that the code runs, it is not optimal in that TensorFlow still is not able to actually differentiate functions from other libraries that do not natively manipulate tensors, especially incredibly complex ones like the Determinant of Hessian algorithm. See Section 5 for more information on this.

## 2.2 Point Loss

When training a model to minimize the Vanilla Distance Loss function, an "erasing" phenomenon occurs, in which the model progressively predicts fewer and fewer points until it predicts no points at all. (Note – the model begins by predicting several points, because it is initialized with pretrained weights. See 3.1., "The Necessity of Pretrained Weights", for more details.) After only 25 epochs of training on the Vanilla Distance Loss function, (see Figures 2.1-2.6), the majority of predicted points have been "erased".

Figure 2.1: Ground truth segmentation map for reference.

Figure 2.2: Predicted segmentation after 5 epochs.

Figure 2.3: Predicted segmentation after 10 epochs.

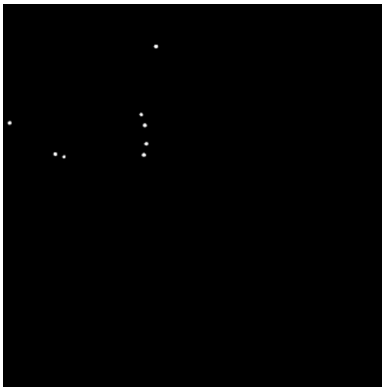

Figure 2.4: Predicted segmentation after 15 epochs.

Figure 2.5: Predicted segmentation after 20 epochs.

Figure 2.6: Predicted segmentation after 25 epochs.

This erasing phenomenon leads to erratic training performance that never improves from its initialized performance, as demonstrated by Figure 2.7. Any improvements in distance loss are always temporary and are reverted almost immediately afterwards. Leaving aside the earlier mentioned possibility of incomplete or improper differentiation of the Vanilla Distance Loss function, one possible explanation for this phenomenon is that an ambitious learning rate that would yield changes to model performance is more likely to erase than to create new points. Once

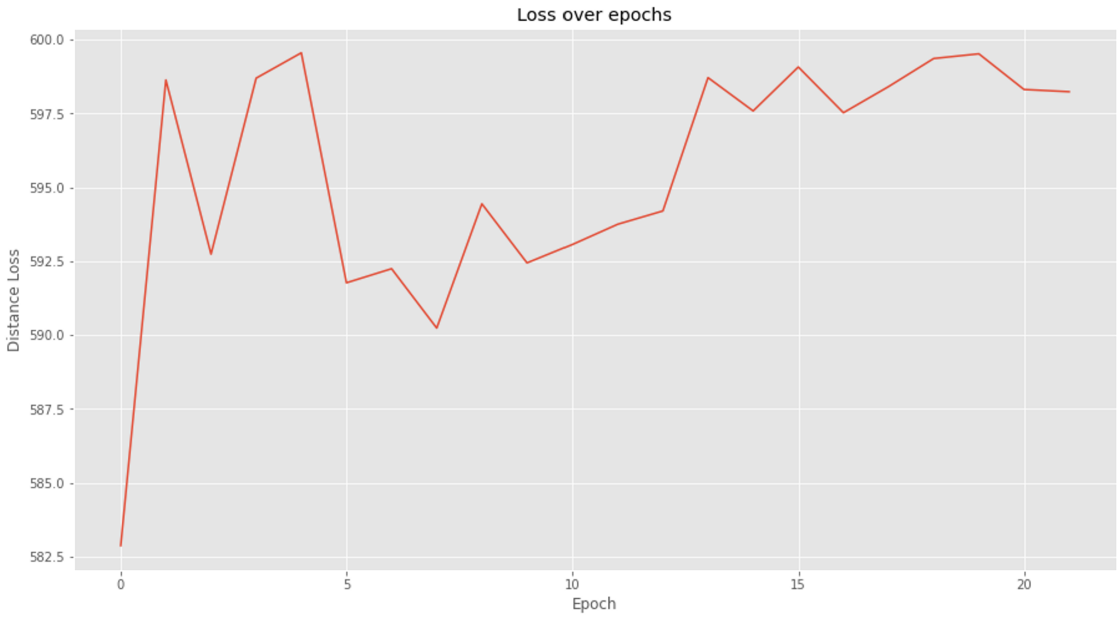


Figure 2.7: Erratic training behavior demonstrated by distance loss across epochs.

points are erased, the model cannot recover. In order to address this problem, point loss attempts to explicitly punish the *number* of predicted dots. Correctly predicting the number of points is an implicit prerequisite for minimizing the average distance between the predicted set of points and the ground truth set of points; thus, by making this implicit assumption explicit, ideally the model will be able to address this "erasing" phenomenon and obtain the necessary prerequisite skills to better predict the location of slits.

Point loss punishes the difference between the number of predicted points and the number of ground truth points. It can be algorithmically defined as follows:

**Algorithm 2:** Key components of the point loss design.

Set sum of point losses to 0;
Select penalty norms $k_s$ and $k_l$, where $k_s < k_l$; **for** *item in batch* **do**
  Obtain EM image and associated ground truth segmentation map for item;
  Obtain model prediction for segmentation map on that EM image;
  Obtain Ground Truth and Prediction arrays of coordinates for detected point locations;
  Obtain number of Ground Truth points $n_{\text{true}}$ and number of Predicted points $n_{\text{pred}}$;
  Obtain the difference between the number of points, $\Delta = n_{\text{pred}} - n_{\text{true}}$;
  **if** $\Delta > 0$ **then**
    Add $\left|\Delta^{k_s}\right|$ to the sum of point losses;
  **else**
    Add $\left|\Delta^{k_l}\right|$ to the sum of point losses;
**end**
Divide the sum of point losses by the number of items;
**return** the mean point loss across all items in the batch;

If $n_{\text{pred}} - n_{\text{true}} > 0$ (i.e. $n_{\text{pred}} > n_{\text{true}}$), the L-$k_s$ penalty is used. On the other hand, if $n_{\text{pred}} - n_{\text{true}} < 0$ (i.e. $n_{\text{pred}} < n_{\text{true}}$), the L-$k_l$ penalty is used. Because $k_l$ is defined as being larger than $k_s$, this point loss penalizes under-prediction ("erasure" of points) more severely than over-prediction. In the implementation of point loss uses in experiments, $k_s = 2$ and $k_l = 3$.

One weakness of point loss is that it fails to proportionally weight under- or over-prediction for training instances with a small number of ground-truth points. If there are six points in the ground-truth segmentation map and the model predicts three points, it misses 50% of the correct number of points. On the other hand, if there are forty points in the ground-truth segmentation map and the model predicts thirty points, it misses 25% of the correct number of points, but is punished more severely than the former instance because the absolute difference in predicted and ground-truth points is larger. To address this, a variation on point loss can be used – proportional point loss.

In proportional point loss, $\Delta = \frac{n_{\text{pred}} - n_{\text{true}}}{\max(n_{\text{pred}}, n_{\text{true}})}$. If the model predicts the number of points perfectly, $n_{\text{pred}} = n_{\text{true}}$ and $\Delta = 0$. $\Delta$ ranges from $-1$ (the worst score for under-prediction) to 1 (the worst score for over-prediction); it can be linearly scaled and exponentiated by $k_s$ or $k_l$ in point loss.

Point loss or proportional point loss can be applied to explicitly reward the formation of new points. It yields slightly improved results compared to the predictions of a model trained only on distance loss.

### 2.3 Composite Loss

Point loss alone does not provide a signal for distance, meaning that a model could hypothetically put dots anywhere. While we don't observe this behavior in a model trained only on point loss, incorporating a location-based signal into the loss function would allow for better stability. We can incorporate distance loss and point loss together by calculating both values for each training instance and returning their weighted sum.

**Algorithm 3:** Key components of the composite loss design.

Set sum of average distances $\sum DL$ to 0;
Set sum of point losses $\sum PL$ to 0;
**for** *item in batch* **do**
  Obtain EM image and associated ground truth segmentation map for item;
  Obtain model prediction for segmentation map on that EM image;
  Obtain Ground Truth and Prediction arrays of coordinates for detected point locations;
  Obtain the distance loss $DL$ based on the obtained coordinates arrays;
  Add $DL$ to the sum of average distances $\sum DL$;
  Obtain the point loss $PL$ based on the lengths of the obtained coordinate arrays;
  Add $PL$ to the sum of average distances $\sum PL$;
**end**
Divide $\sum DL$ and $\sum PL$ by the number of items to obtain the mean distance loss $\bar{DL}$ and mean point loss $\bar{PL}$ across all samples in batch;
**return** weighted sum of mean distance loss and point loss $\alpha \cdot \bar{DL} + \beta \cdot \bar{PL}$ across all samples in batch;

In the implementation of composite loss, $\alpha = 0$ and $\beta = 10$, such that $\alpha \cdot \bar{DL}$ covers the approximate same range of values as $\beta \cdot \bar{PL}$. Ideally, the model is able to optimize the loss function by reducing the distance function, but is severely penalized if begins to exhibit the "erasing phenomenon".

## 3 Pretrained Model Considerations

### 3.1 The Necessity of Pretrained Weights

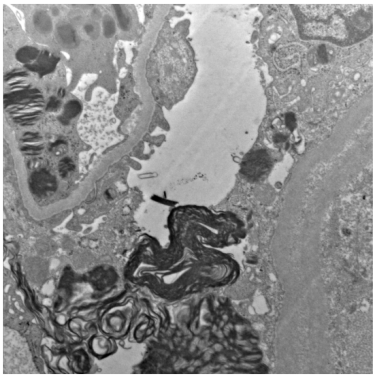

Figure 3.1: Electron microscopy input image to segmentation model.

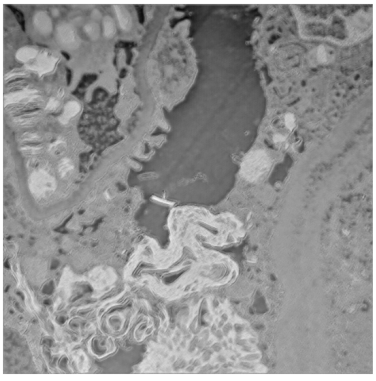

Figure 3.2: Output of new, randomly initialized model w/ lack of detectable blobs.

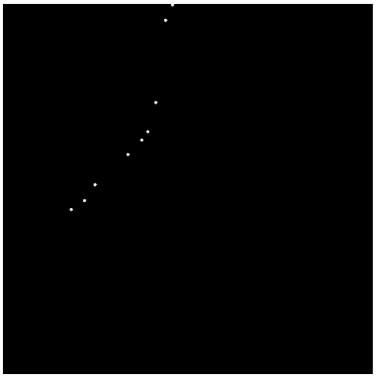

Figure 3.3: Output of using a model with pretrained weights (i.e. has basic prediction capability).

The loss function requires the model to begin by predicting reasonably dot-looking shapes to be detected (like in figure 3.3). An initialized model predicts a segmentation map without any identifiable dots (see figure

3.2). Per the Vanilla Distance Loss algorithm, the corresponding loss function would be the maximum distance ($\sqrt{2} \cdot$ image length). However, the model receives no input signal (i.e. the loss landscape is flat the maximum distance all around the optimizer) because any changes it makes to the weights is almost guaranteed to make no change to the loss function. Hence, pretrained weights are required as an initialization point such that dots can be detected. Ideally, the pretrained model will be able to access a rich, non-constant loss function that accelerates its learning of both the number and location of the segmented dots.

## 3.2 ForkNet Model Architecture

The pretrained model is a ForkNet architecture (see figure 3.4) trained on the dice loss. The ForkNet architecture takes an EM image and outputs two images: a membrane segmentation map (outputting the presence of the membrane) and a slit segmentation map (outputting the presence of a slit; this is the relevant output for improvement). The existing ForkNet model performs well, but is not very exact with dots that are close to one another.

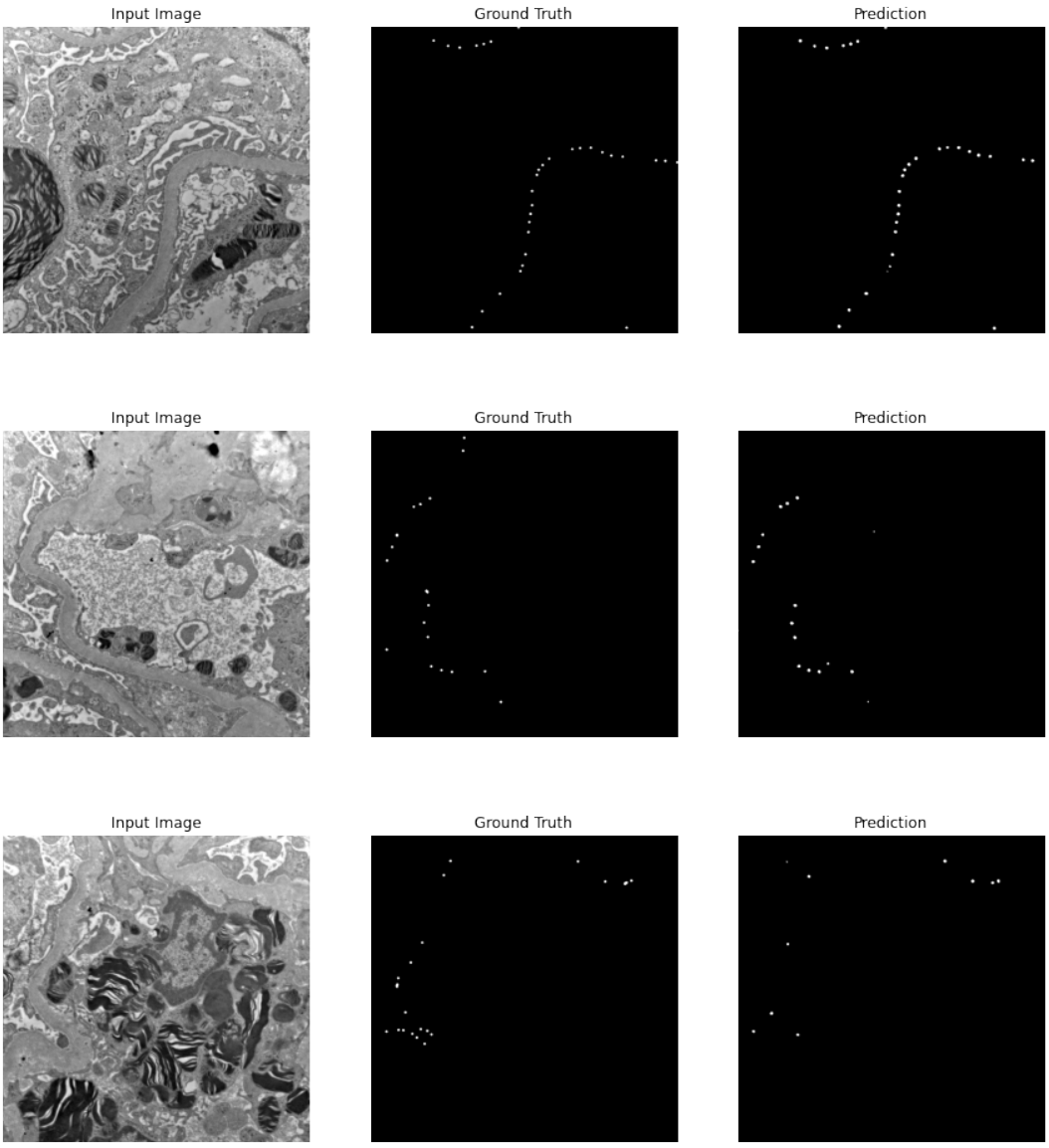


Figure 3.4: Three examples of the pretrained ForkNet predictions compared to ground truth segmentation map.

Because the ForkNet model contains two outputs but the designed loss function operates only on the slit segmentation map, the ForkNet model is trained on two losses: standard dice loss for the membrane segmentation map and the distance-based loss function for the slit segmentation map. One consideration for this multiple-output

model is that the losses must be weighted for consideration; dice loss ranges from $-1$ to 0 but various distance-based losses span a much wider range. There's an important trade-off to be examined: the relationship between the model's performance of the membrane segmentation map and the slit segmentation map. At a low-performance level, the model performance on the membrane segmentation map is correlated with its performance on the slit segmentation map, likely because the basic skills to perform decently on both are linked across both tasks. However, it seems that at a high-level, performance on the membrane segmentation map conflicts with performance on the slit segmentation map. Therefore, both weighting the dice loss very low (near 0) and weighting it too high fail to achieve relatively good performance.

# 4 Model Training Procedure

In this context, the specifics of the model training procedure proved essential to how the model responded to the loss function. Via data point-by-data point analysis of the model's development, it was discovered that the model requires a very particularly-set, well-regulated learning rate.

## 4.1 Adjusting Learning Rate and Decay Parameters

There are several key learning rate regulation parameters to be aware of in this context:

- *Initial learning rate*. This is the learning rate that the model begins with. It is rather insignificant in its own right but provides a "maximum learning rate bound", which will be relevant to how the learning rate is regulated.
- *Decay factor*. This is the factor by which the learning rate is reduced every time the regulation algorithm indicates it should be.
- *Patience*. This is the number of epochs of no improvement (the 'plateau') the LR-adjusting algorithm is willing to wait or tolerate before it reduces the learning rate by dividing it by the decay factor.

The success of the model is incredibly fragile with respect to the result of the model. If an LR-updating algorithm is too ambitious (e.g. large initial LR, small decay factor, high patience), the model often exhibits wild behavior and results in significantly poorer performance, demonstrating heavy (but not total) "erasing phenomenon" in which the model predicts 40% to 50% the quantity of the dots in the ground truth segmentation map. On the other hand, if an LR-updating algorithm is too prudent (e.g. small initial LR, large decay factor, low patience), the model learns weights that perform no better than the original model. Because of the model's sensitivity to LR-updating strategies, it is difficult to "tread the line" and find a satisfactory intermediate updating algorithm.

## 4.2 Cyclic Learning Rate

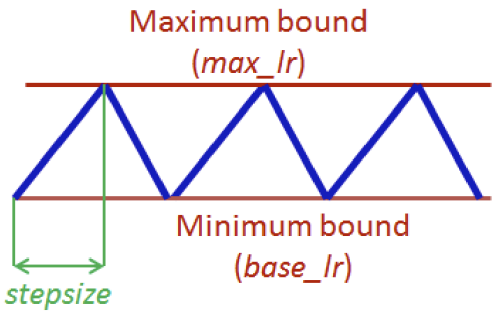


Figure 4.1: Visual depiction of a triangular LR policy from Leslie Smith.

A cyclic learning rate, rather than progressively decaying the learning rate, repeatedly decreases and increases the learning rate in a cycle. The triangular learning policy alternates between a maximum bound and a minimum bound, as depicted; the period is two times the step size, which is the number of steps the policy accommodates before 'changing direction'. The rationale behind the cyclic learning rate is that a monotonically decreasing learning rate decreases an optimizer's ability to navigate more complex loss landscapes – the optimizer progressively "loses

steam". By cycling between large and small learning rate values, the optimizer is able both to explore new solutions and to converge to satisfactory solutions.

This cyclic learning rate used in implementation is a variation of the triangular learning rate proposed originally by Leslie Smith, halving the height of the maximum bound every period as to eventually converge upon a learning rate. Using a cyclic learning rate performs better than using the more traditional Decay Learning Rate on Plateau strategy.

### 4.3 Weight Reloading

For initial experiments, the model was initialized with dice-loss-trained weights (as discussed in Section 3). This provided a starting point for the model to predict visibly detectable blobs that could be used to calculate the distance-based loss function. In weight reloading, a model initializes with the weights of a model in a previous experiment. The reloaded weights are the "best weights" from that experiment (i.e. the weights that yield the lowest loss). Using weight reloading, the model can "restart" at where the prior model had left off in terms of performance improvement.

To formalize for the sake of clarity: let $CL(M)$ be the composite loss of some model $M$, $M_{\text{orig}}$ be the original unmodified model with original dice-loss-trained weights, $M_A$ be a model trained from the weights of $M_{\text{orig}}$ on the new distance-based loss function, and $M_B$ be a model trained from the best weights of $M_A$. The key observation from multiple experiments in weight reloading is that $CL(M_B) < CL(M_A)$, even though $CL(M_{\text{orig}}) < CL(M_A)$. $M_B$, which was trained with weight reloading from the *worse*-performing $M_A$, performs better than $M_A$, which was directly trained from the *better*-performing $M_{\text{orig}}$. (Note that neither $M_A$ nor $M_B$ perform *better* than $M_{\text{orig}}$ in composite loss) This suggests a certain complexity to this optimization problem/approach, in which our conventional understandings of how solutions should be ranked – i.e. a solution derived from a good solution should be better than a solution derived from a bad solution – do not apply.

## 5 Summative Experiment Results and Discussion

Even through several dozen experiments testing out various combinations of loss functions, pretrained model considerations, and model training procedures, the best model yields marginally better performance than the original model. There are two phenomena at play here that inhibit the success of a distance-based loss function in this case:

- *Differentiability problems.* There are complications with differentiating complex functions external to the Keras/TensorFlow framework. TensorFlow's documentation website states that "Since the function takes numpy arrays, you cannot take gradients through a `numpy_function`... The resulting function is assumed stateful and will never be optimized." The design of a distance-based loss on a segmentation task inherently is associated with differentiability problems. See Section 6 for possibilities on future inquiry.

- *Learning rate adjustment sensitivity.* There is an interesting observed dynamic in which too ambitious a learning rate schedule results in failure and too prudent a learning rate schedule results in no change. Any regularization attempts to prevent the model from utterly failing (e.g. point loss/composite loss mechanism to punish "erasing phenomenon") yields a model that performs almost identically to the original model. A reliance on the performance of the loss function is built into the design of the distance-based loss, but also seems to serve as an upper ceiling for performance.

Ultimately, this project was concluded because these factors were judged to be too significant to create a significantly more successful model. It appears that this loss design task is a problem that obeys Occam's Razor on the goodness of simple solutions – it seems that attempts involving too much complexity results in a cat-and-mouse game in which an attempt to address one problem forces new problems to continually arise.

## 6 Future Directions for Exploration

Here are possible further directions for exploration:

- *Directly train the model to predict point coordinates.* Rather than outputting a segmentation map, which requires a chain of complex operations to calculate the distance function, a model that explicitly predicts the coordinates of points allows for the building a loss function explicitly with Keras/TensorFlow functions in a way that would resolve differentiability problems. The model could either be fitted with a recurrent output to

adapt towards variable quantities of points or be fitted with a fixed output and an `<empty>` token as a filler for the absence of a token.

- *Gradient accumulation.* Because the ForkNet architecture uses $640 \times 640$-sized images, it cannot be trained with a high batch size because of memory constraints. This yields erratic training behavior as the result of batch updates that are informed by a very small sample of images. Gradient accumulation is a strategy to accommodate small batch sizes by making updates after several batches have been processed rather than after every batch. There was difficulty implementing gradient accumulation in the implementation of the experiments discussed in this report because of version compatibility issues. While there is little reason to believe that it alone would dramatically increase the success of this model approach, it would have likely provided a performance boost.
- *Strongly bijective chamfer distance.* Rather than accepting many-to-many mappings of predicted coordinates to ground truth coordinates, require that the number of predicted points be equivalent to the number of ground truth points. If this condition is not satisfied, use some sort of point-based penalty (e.g. point loss). This prevents overlapping signals (in composite loss, point loss and distance loss can be said to be 'competing' against one another) from misleading an already sensitive task paradigm.