1 Attention UNets for Image Segmentation

In this article we will talk about the basics of image segmentation in computer vision, how a machine learning model called a UNet is well suited for tackling the problem, how to train these networks more efficiently, and finally how to improve on the basic design by implementing bottleneck attention and local attention in the decoding steps. Mathematical symbols are used as appropriate but the paper is written such that a high school math education is sufficient and anything more complicated is accompanied with a verbal expalantion. Full model architecture diagrams and algorithm pseudo code are provided at the end for the sake of completeness. The complete source code is also available on github along with the google colab python notebook used for training.

1.1 The Problem

Before we talk about image segmentation we must first describe the easier problems of image registration and image bounding. For image registration, we have a list of labels for instance "dog", "cat", "human" and we would like to determine if any given image contains any items matching those labels. Mathematically we can think of the problem as finding a function F_θ that takes as input a 256 by 256 color image (which we can convert into $3 \cdot 256 \cdot 256$ numbers) and outputs three numbers $F_\theta(x) = (d, c, h)$ with $0 \leq d, c, h \leq 1$. We interpret these numbers as the certainty that the picture contains the corresponding label. If $d = 0$ then the model is certain that the image does not contain a dog. If $d = 1$ then the model is certain that the image contains a dog. If $d = 0.7$ then the model is fairly sure that the image contains a dog. The subscript θ refers to the fact that the function we are trying to find is determined by a (large) number of parameters. To figure out what these parameters should be, we first collect a set of images S

$$S \subset [0,1]^{3\cdot 256\cdot 256} \text{ is the set of images for which labels are known}$$

and the labels for this set are given by a known function

$$g : S \to [0,1]^3 \text{ maps images in S to the corresponding label probabilities}$$

If we work very hard and label every possible image in existance then $F_\theta = g$ and we are done, life however is not so easy. It is important to note that the labels we generate ourselves will just be 0 or 1 since, as humans, we are pretty good at telling the three apart. In the event that we find an image for which we are uncertain, we can always just throw it out. The next step is to create our model F_θ which for now is just a very complicated non linear function that is easy for a computer to differentiate. Since we don't know what the parameters θ should be we initialize them to some small random numbers. We now have a model and it will likely spit out garbage (it's a bit like writing a book by successively picking random words from a dictionary). What's important though is that we can define a loss function L:

$$L(F_\theta, g) = \sum_{s\in S} ||F_\theta(s) - g(s)||$$

which is to say we just add up all the distances between what the model assigned and what we know to be true. The smaller the loss, the better the model, if the loss is 0 then the model agrees with known labels. Our new goal is to find parameters θ that make the loss as small as possible, to do that we do gradient descent. That is to say we compute the derivative of the loss function with respect to θ and then we adjust θ in the opposite direction and do it again and again. This is the basis on which modern machine learning is built.

The next version of the problem is producing bounding boxes. For each image, if the model detects a label, we would like for it to draw a tight box around the object. Mathematically our new model would now output $5k$ numbers instead of k numbers for each of the k labels. For each label, the model needs to output a confidence value between 0 and 1, it now also outputs the coordinates of the top left corner of the bounding box, the height of the bounding box, and the width of the bounding box for a total of 4 extra numbers per label. The job of producing labeled data has now gotten a lot harder.

1.1.1 Producing Labeled Data

Machine learning is usually limited by two things: processing power to run the algorithm and quality labeled data. For some machine learning tasks like training language models or auto encoders we can simply feed a large corpus of data like wikipedia text, public domain text, common crawl and google image results into our models. The upside of this approach is that these models can learn the general shape of the data, on the other hand we pick up some new issues. Training a language model on reddit discourse might make an AI able to converse like a person but it will also internalize all of the racisim, bigotry, and missinformation that can be found on the platform. The nature of AI training is that without direct intervention all information is treated equally. In particular if a particular sentiment is oft repeated in the corpus, it is much more likely for the AI to pick up on it. If an AI sees "the earth is flat" written more often than "the earth is round", that AI is much more likely to later reproduce "the earth is flat". In the case of using random image data you run into the problem of including obscene, hateful, or violent imagery that the model might choose to reproduce even when it isn't asked for explicitly. Finally a new and budding problem is that more and more content on the internet is AI generated, therefore scraping internet content is more and more likely to produce AI content rather than human generated content. Training AIs on AI generated content creates a feedback loop that amplifies negative biases and degrades quality (since at the very least the AI isn't seeing anything new).

Back to the problem at hand, for image segmentation and classification we are in need of labeled data. There is good news and bad news. The bad news is that it requires thousands of man hours to create (often being outsourced to poor developing countries or poor graduate students), the good news is that a lot of this work has already been done. Furthermore advances in AI allow us to bootstrap our earlier results. In the case of image classification we might start with a collection of labeled images, then give our AI a much larger collection to classify. We can now go back over the AI classifications and quickly mark them as correct or incorrect. Correct classifications can now be added to our labeled pool and in this way quickly make the pool bigger. In the case of genearting bounding boxes, we might use a more robust, deterministic, algorithm that lets us draw boxes that snap to features in the scene. As long as this algorithm can extract different shapes from the picture (without any idea what those shapes are), it can dramatically speed up the labeling process. In the case of image segmentation, instead of drawing boxes we need labels for each individual pixel. An algorithm like photoshops "magic select" lets us quickly add similar regions together to make a mask.

1.1.2 Our Dataset

Our model was trained on about 12,700 labeled images from the DeepFashion dataset. This dataset had 24 unique labels and featured models in a variety of poses. The largest limitation of the dataset is that pictures were shot with studio lighting against a flat white backdrop. As a result any model trained from the dataset would not be able to properly interpet background objects. However this limitation can be rectified by either training a second model that filters out backgrounds from people (a 2 label segmentation model) or by fine tuning the finished model on a larger corpus containing noisy backgrounds. Most of the images featured female models which is appropriate since there is a far greater variety of female presenting clothing. The models also included a variety of races which is especially important as models that are not trained on dark skin tones are likely to misinterpret them. The diversity of a training pool should match the diversity of the overall population for which the model is deployed. As an example if the model was to be used by an arabic brand it would make sense to train it using pictures of predominantly middle eastern models and clothing styles popular in that region. Since the dataset contains exclusively professional models, it is biased towards people with specific proportions (taller than average, thinner than average, more symmetric than average). In the case of clothing segmentation this is less of a problem (although training for images of children or obese people may be necessary depending on the application). Furthermore the data set did not contain any images of underwear or swimwear.

1.2 Image Segmentation

Image segmentation is much like image registration, the difference is that instead of tagging the entire image with whether or not it contains a label, we now tag each individual pixel with whatever label it happens to belong to. While this task may sound a lot harder, it isn't actually a very large step from registration and creating bounding boxes. If the network learns what certain labels look like and roughly where they are, it often times isn't a big stretch to then label the pixels.

1.3 Instance Segmentation

The next step up from image segmentation is instance segmentation. Now instead of simply labeling each pixel with a label, we want our model to differentiate between different instances of that label. In the case where different instances are expected to remain connected, we can simply train a regular segmentation model to output one extra channel containing a boundary mask. However if instances are disconnected (like a person standing in front of a car) then the task may have to be done itteratively: the model might first generate bounding boxes for different labels and then extract instances one at a time from each box until the entire image has been masked.

2 Challenges of Clothing Segmentation

In this section we go over some of the many challenges that make clothing segmentation a particularly tricky problem

2.1 Textures and Patterns

Clothing comes in a variety of different textures, colours, and designs. An article of clothing might contain large macroscopic elements such as text and pictures, it may also be heterogenous in design with different portions being different colors. Since both tops and bottoms can be made of the same types of materials, it is not possible to determine what type of clothing something is if we are restricted to looking at a small region. Since looking at colours alone is unsufficient for locally segmenting clothing, a model needs to be able to infer texture information. There are both microscopic and macroscopic types of texture information that can be inferred from an image. Examples of microscopic data are the bumps of leather or the crosshatching of threads, while this data is excellent for locally identifying contiguous regions, it also requires resolutions that are often unavailable. Macroscopic texture information can include relative focus due to depth of field, or things like whether light is more specularly or diffusely reflected. This type of information is available even for lower resolutions but it may not be sufficient to classify items. For instance we have less reflection information for portions of the image that are not directly illuminated.

2.2 Pose and Macroscopic Shape

If we were to look at just the silhoutte of a clothing article it might not always be sufficient to identify what type of clothing it is or even if it is clothing at all. While T shirts and tight fitting pants have fairly consistent sillhoutes, this can still be complicated by the pose of the wearer. Side profiles can obscure the shape of the sleeves. In the case of long sleeves or loose clothing the effect becomes even more complicated. The model would have to properly identify sleeves amongst folds of fabric in order to determine the general features of the article.

2.3 Distantly Coupled Features

The more information that a network has to pay attention to, the more memory it is going to take and the slower it will run and train. Networks that using only local feature information have trouble identifying items that are much larger than the local window. For example a network looking at dress might think the top portion is a top and the bottom portion is a skirt. Even if the network could infer that the waist portion would have to belong to a dress, that information would not be able to propogate to the tips. The final prediction from such a network would be the top portion of the dress being a top, the middle portion being a dress and the bottom portion being a skirt. Rompers are even more complicated because they look like a dress except for the small portion separating the legs (which could get a bit obscured if the leg portion is flowing). To solve this problem the network needs a way to efficiently transfer information across large spatially separated regions. We can interpret this process of taking local feature information and producing global feature information as being a type of consensus/clustering problem which is already non trival.

2.4 Landmarks and Context

Features like rings and bracelets often occupy only a few pixels for a lower resolution image. In order to accurately label them the network needs to identify landmarks (where in the image the fingers and wrist are) then restrict its analysis to only those regions. Without this additional context, the local signal is too weak and we end up over identifying or (more likely) failing to identify the feature entirely. In order to keep track of this additional context information, the model requires even more memory and processing power.

3 UNet Design

The UNet design has three components: an encoder, a bottleneck, and a decoder. The encoder takes a signal with spatially localized information and peforms successive transformations that reduce spatial information (where exactly things are) but increase the amount of feature information (what things look like). Each block in the encoder cuts the spatial information in half but doubles the amount of feature information (with the exception of the first block that has only 3 features for the color channels). Early blocks use pixel information to find fine features like edge segments, later blocks then use this to build lines, then shapes and textures.

The bottleneck of the network is where we stop reducing spatial information and then use the high level feature information collected in order to make broad inferences about the picture. For example our network takes a 3 x 512 x 512 image to a 1536 x 16 x 16 vector in the bottleneck. This means that each 1536 x 1 x 1 pixel describes a 3x 32 x 32 pixel block in the original image. The network can use this information to infer roughly what objects are in each block but it might not have enough information to determine what the pixel boundaries look like. Furthermore the bottleneck is the only place where we can make inferences between very distant blocks.

The last portion of the network is the decoder. The decoder takes the output of the bottleneck which has lots of feature information and low spatial information and essentially does the opposite of the encoder: increasing spatial information while decreasing feature information. We pass residual information directly from the encoder to the decoder at each resolution scale so that it can use it to better determine where the boundaries between different regions are. As an example the exact position of an edge is useful when the network is trying to determine the exact boundary between two classes, however it is impractical to carry all of this fine detail information through the encoder, bottleneck, and back up through the decoder. Since the bottleneck is capable of performing inference using larger scale feature information (shapes and textures), it is much more practical to reference this information only when it becomes relevant again.

The last portion of our network uses two convolutions to reduce the 96 pixel channels down to 24 label predictions plus 1 boundary prediction. The former is passed through a softmax layer before being intepretted as a probability distribution, the latter is passed through a sigmoid layer.

4 Measuring Loss

4.1 Intersection Over Union

Let us interpret a ground truth mask for a label j as a function that maps the discrete pixel coordinates in the image to 1 if the pixel belongs to that label and 0 otherwise

$$f_j : \mathbb{Z} \cap [0, 512) \times \mathbb{Z} \cap [0, 512) \to \{0, 1\} \qquad \sum_j f_j = 1$$

Similarly let us interpret our prediction for the ground truth as a function g_j which can take any real value between 0 and 1.

$$g_j : \mathbb{Z} \cap [0, 512) \times \mathbb{Z} \cap [0, 512) \to [0, 1] \qquad \sum_j g_j = 1$$

We can define $I_j = \int f_j \cdot g_j$ as the total weighted area of their intersection and $U_j = \int f_j + g_j - I_j$ as the total weighted area of their union (If g_j only takes values of 0 or 1 then these are exactly the areas of their intersection and union). The intersection over union measure of accuracy is just

$$A_j = \frac{I_j + \epsilon}{U_j + \epsilon}$$

where ϵ is a small number to keep us from dividing by zero. Since networks are designed around decreasing loss rather than increasing accuracy we instead care about $L_j = 1 - A_j$. Summing over all of the labels gives us a servicable loss function.

While IoU provides a good measure of the accuracy of the network, overall it is not a good loss function. The issue is that not all of our labels appear in each picture. Guessing that a label appears in a picture where it doesn't instantly increases that portion of the IOU loss from 0 to 1. As a result the network chooses to never suggest a rare label, even if the network had previously learned to predict that label with 90% accuracy. The second issue with this system is that all labels are treated equally despite their relative size within the image. In some cases this would be desirable, however in our case predicting rings and neclaces is not as important as correctly predicting tops and bottoms. Both problems can be addressed by weighting each individual IoU score by the mass of each label as it appears in the picture (or doing something else proportional to this).

The last property of interest for IoU loss is that the individual pixel response is essentially linear. Changing a prediction from 0 to 0.1 incurs roughly the same change in loss as going from 0.9 to 1. On one hand this encourages sparcity, on the other hand it provides a weaker training signal when our prediction for a label is very bad. While its use as a loss function is not quite ideal, IoU scores are an effective metric for analyzing the results of different models.

4.1.1 When good enough is perfect

Imagine that our model predicts the correct category for each pixel as being 0.51%, this means that if for each pixel, we take the predicted label with highest probability, then our prediction will perfectly match the ground truth. Ideally we would like to use a loss function that takes into account the fact that we will use the highest probability for each each label, however this process is not differentiable (as all loss functions must be). This is also another reason why IoU is not an ideal loss function: the network will try to make already strong predictions stronger just as often as it makes weak predictions stronger, even though the former will have little or no effect on the final pixel accracy.

4.2 Cross Entropy

Weighted cross entropy is the loss function that we ended up using for training. With functions f_j and g_j as above, cross entropy loss is given by

$$L = -\sum_j w_j \cdot \int f_j \cdot \ln(g_j)$$

Mathematically this is the sum of KL divergences for each each distribution. Practically speaking though, this is just a pixel accuracy loss where small predictions for a label are penalized incredibly heavily (mathematically the loss for the logarithm becomes infinite when g_j goes to 0 so we cut it off at some maximum value). The values of w_j are just weights we can apply to each label. In our case it helps to increase the weights of jewelry relative to clothing so that we punish the network for failing to learn them. Cross entropy has the downside of discouraging sparsity, the network is incentivized to hedge its bet on each label (i.e every pixel is predicted to be every label with a small probability) just because the penalty for completely missing a label is massive.

4.2.1 The Trouble With Weights

The network has trouble finding things like rings, bracelets, and neclaces because they are very thin. The decoder portion of the network seeks to create consensus between different regions, as such a rare label like [ring] often gets subsumed by the more common [skin] label. By increasing the weight of [ring] relative to [skin] in the loss function, we force the network not to subsume [ring] signals amidst [skin] signals. However if the weight becomes too high, the network will aggresively guess that [ring]s exist wherever they might be simply because that incurs less overall loss compared to failing to identify where a [ring] is.

By looking at the relative mass of each label compared to the relative mass of our predictions, we can identify if the network is under or overpredicting and adjust the weight accordingly. This however is not a magic bullet, it could be that the network is failing to classify a label simply because it does not have enough bandwidth or training examples. Even though increasing the corresponding weight causes the model to identify pixels with that label more often, that does not mean that those additional classifications are correct.

4.3 Regularization

If we know what the distribution of the output is supposed to look like, we can penalize the network for failing to take it into account. If we look at any individual mask, it should be uniform in its interior region and then sharply drop off, in other words we want the discrete gradient of each mask to be sparse. One method to achieve this result is to the L^p semi norm with $p < 1$. Since the gradients blow up we use a cut off:

$$||x||_p = (|x| + \epsilon)^p - \epsilon^p$$

This norm promotes one big jump over multiple little jumps, however it still amplifies minute variations. We can try to ignore small variations by clamping near 0.

$$||x||_p = \max((|x| + \epsilon)^p - \epsilon^p - c, 0)$$

Another scheme we might try to promote accuracy is reweighting the pixel score not by label but by position. This means that we increase the weight of pixels that appear along the boundary of an item relative to the interior. In order to use this technique it is imperative that the labeled information is as accurate as possible, otherwise these descrepencies will be greatly amplified by the model and lead to strange results.

5 Preprocessing, Postprocessing, And User Response

5.1 Preprocessing

While there are some tasks that machine learning models are well equiped for, there are others that can be better handled by conventional programming techniques. As an example we can either increase the network bandwidth so that it is able to identify features on vastly different scales, or we can first rescale the input so that the network only needs to learn a narrow band of variations. We can train our network to deal with noisy backgrounds or we can train a separate network that segments out people from a background before passing the segmented image along. However it might turn out that a network capable of segmenting people from general backgrounds ends up encoding enough feature information that it might as well perform the clothing segmentation as well.

5.2 Postprocessing

The simples method to produce a segmentation given label probabilities for each pixel is to assign to each pixel the label of highest probability. While quick and easy, this method has a number of down sides. First of all the network might be able to segment out an article of clothing but have trouble reaching consensus as to what that article is. The network might end up thinking that pixels near the boundary of a pair of pants are slightly more likely to belong to a skirt. To fix this issue we implemented a post process step where a segmentation of the image into unlabeled regions is used to build a collection of masks. Those mask are then used to pool statistical information from the segmentation and output one cohesive label for the mask. Any leftover pixels are then added to whichever mask has the most similar statistics or promoted to masks in their own right if their certainty is high enough (since jewelry is sometimes too small to generate a large enough mask in the first step to meet the threshold).

Ideally the network would output a perfect segmentation and we would have a single mask per connected label. In practise the network ends up predicting a lot of extrenuous boundaries. Even worse, the network will sometimes miss a boundary and as a result two different labels end up pooled together. Extra boundaries are not serious from a practical standpoint but they are a strong indicator that the network has trouble segmenting individual items (in other words the bottleneck needs to be more robust). Extra boundaries make the algorithm run a bit slower but don't make the output that much worse. If the network thinks that an item is actually two different items there isn't much more to do at that point. Missing boundaries on the other hand can have catastrophic concequences for our accuracy, especially when the [top] and [pants] are mixed together or a [top] and the [background]. We used a custom Binary cross entropy loss that penalized false negatives at 20 times the rate of false positives to greatly reduce the chance of missing boundaries at the expense of creating a lot of incorrect ones. Furthermore we used a simple expand contract algorithm that is capable of closing small pixel gaps (but fails when the boundary information becomes sufficiently noisy).

Our post processing algorithm has a number of parameters that can be tuned to give better results on an image by image basis. Since this was mostly a broad level research exercise, we did not look into more robust algorithms like vectorization (since they require orders of magnitude more computational power to run)

5.3 User Response

Machine learning models do not exist in a vacuum but rather have some kind of real world use case. It is often possible to have a user instantly perform actions that might take some significant processing power otherwise. Instead of manually rescaling an image we can have the user drag and resize the image so that the head appears inside a target oval. A user could also resolve label confusion using a single press (is this a [dress] or a [romper]?)

6 Normalization and Regularization

One of the big differences between dealing with mathematical models vs practical machine learning is the issue of finite precision arithmatic. Machine learning is generally performed with 32 bit floating point numbers, these are stored as $1.F \cdot 2^{E-127}$ where F is the 23 bit fractional part of the normalized number, E is an 8 bit integer between 1 and 254 (so the exponent can range between -126 and 127). For example $2 = 1.0 \cdot 2^1$ so $F = 0$ and $E = 128$, the number 3 however can be written in binary as 11 or $1.1 \cdot 10$ so again $E = 1$ and $F = 1 \cdot 2^{23}$ depending on how we order the bits. This means that consecutive floating point numbers greater than 2^{23} must be at least 1 apart. Our very large floating point numbers turn into integers! Beyond that they get even worse as consecutive floating point numbers start differing by a factor of some power of 2 all the way to 2^{104}. Therefore if we want any hope of doing computation with any degree of accuracy we want to make sure that we deal with small enough numbers. Even if we tune our model so that all of the activations stay between -1 and 1, it is still possible that gradients could become very large or very small (since we are ultimately multiplying them all together during back propogation). We must be especially vigilant of intermediate calculations where we end up adding together numbers of different magnitudes. For instance in floating point arithmatic $0.1 + 2^{24} - 2^{24} = 0$ while $2^{24} - 2^{24} + 0.1 = 0.1$, floating point arithmatic is not commutative for numbers of vastly different scales!

For complex numerical systems where such computations are unavoidable we might use double (or even higher) precision numbers, we might also employ techniques such as reordering large summations in order of scale to effectively deal with the case above. While these techniques are more accurate they are orders of magnitude slower. Most hardware that we deal with is optimized for 32 bit (or even 16 bit) computations, it is much easier to change our code to suit the hardware than it is changing the hardware to suit our code. To that end we employ a number of computation techniques to stay within a domain where finite precision remains a robust approximation.

6.1 Weight Decay

The process of weight decay is simply taking all of our model parameters θ, computing their L^2 norm $||\theta||_2^2$ and adding it to the loss function (with a suitably scaled constant parameter μ). We are essentially telling the algorithm that at a certain point it is not worth trying to improve the accuracy if it were to make the weights too big. Since the gradient of $\mu||\theta||_2^2$ with respect to θ is just $2\mu\theta$ we can just subtract $2\mu\lambda\theta$ from θ during each itteration of the algorithm to get this effect. Most machine learning libraries already implement weight decay in the optimizer and allow for different values of μ for different parameters so that they can be decayed in batches.

6.2 Skipped Connections

Imagine we have a model given by $F_{\theta_1} \circ G_{\theta_2} \circ H_{\theta_3}$ with $F, G, H : \mathbb{R}^n \to \mathbb{R}^n$. If we add a cost function $c : \mathbb{R}^n \to \mathbb{R}$ to the head we can compute the gradients:

$$x_1 = H_{\theta_3}(x_0) \qquad x_2 = G_{\theta_2}(x_1) \qquad x_3 = F_{\theta_1}(x_2) \qquad y = c(x_3)$$

$$\begin{aligned}
\left.\frac{\partial c}{\partial \theta_1}\right|_{x_2} &= \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \left.\frac{\partial F}{\partial \theta_1}\right|_{x_2,\theta_1} \\
\left.\frac{\partial c}{\partial \theta_2}\right|_{x_2} &= \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \left.\frac{\partial F}{\partial x}\right|_{x_2,\theta_1} \cdot \left.\frac{\partial G}{\partial \theta_2}\right|_{x_1,\theta_2} \\
\left.\frac{\partial c}{\partial \theta_1}\right|_{x_2} &= \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \left.\frac{\partial F}{\partial x}\right|_{x_2,\theta_1} \cdot \left.\frac{\partial G}{\partial x}\right|_{x_1,\theta_2} \cdot \left.\frac{\partial H}{\partial \theta_1}\right|_{x_0,\theta_1}
\end{aligned}$$

Each block in the backpropogation step is a matrix multiplication. We can see how the derivatives of shallow layers (closer to the input x_0) has to pass through all of the gradients of the deeper layers. Now let us consider a new model employing skipped connections: $(F_{\theta_1} + \mathbb{1}) \circ (G_{\theta_2} + \mathbb{1}) \circ (H_{\theta_3} + \mathbb{1})$

$$x_1 = H_{\theta_3}(x_0) + x_0 \qquad x_2 = G_{\theta_2}(x_1) + x_1 \qquad x_3 = F_{\theta_1}(x_2) + x_2 \qquad y = c(x_3)$$

$$\begin{aligned}
\left.\frac{\partial c}{\partial \theta_1}\right|_{x_2} &= \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \left.\frac{\partial F}{\partial \theta_1}\right|_{x_2,\theta_1} \\
\left.\frac{\partial c}{\partial \theta_2}\right|_{x_2} &= \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \left[\left.\frac{\partial F}{\partial x}\right|_{x_2,\theta_1} + \mathbb{1}\right] \cdot \left.\frac{\partial G}{\partial \theta_2}\right|_{x_1,\theta_2} \\
\left.\frac{\partial c}{\partial \theta_1}\right|_{x_2} &= \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \left[\left.\frac{\partial F}{\partial x}\right|_{x_2,\theta_1} + \mathbb{1}\right] \cdot \left[\left.\frac{\partial G}{\partial x}\right|_{x_1,\theta_2} + \mathbb{1}\right] \cdot \left.\frac{\partial H}{\partial \theta_1}\right|_{x_0,\theta_1}
\end{aligned}$$

We now see that instead of having a third order term in θ_1 we now have one third order term, two second order terms, and one first order term. This means that even if the gradients of F and G are very small, the network can still get information about H instead of getting stuck. Furthermore if the functions in the network are expected to be somewhat close to the identity, then this decomposition makes it much easier for the network to get close to the solution. It is in general easier for an arbitrary layer to learn how to map to a small number than it is for an arbirary layer to learn the identity mapping.

An often overlooked consideration for using skipped connections is that the layers involved need to be able to learn the target map. For example lets take a network containing generic layers F, G, H that all end with ReLU activation (the results are always positive definite). In other words regardless of the choice of parameters, $F, G, H : \mathbb{R}^{\mathrm{n}} \to [0, \infty)^n$. while the original network would be able to take an input of $(2, 2)$ and map it to $(1, 1)$ the second network would not be able to. It is natural to ask the question: if this is a real phenomenon then why don't models that make this mistake simply break. The answer is that there is eventually some kind of linear mapping that is able to perform subtraction for us. Let us take a simple example of a network whose input is two dimension, whose intermediate steps are four dimensionsal and whose output is two dimensional terminating with an intermediate layer. Our network might start by mapping $(2,2)$ to $(2,2,0,0)$, due to our naive skipped connections the coordinates can no longer increase, however the network can learn to treat the extra features as negative. The network can then take $(2,2,0,0)$ to $(2,2,1,1)$ and the final linear layer can be $(x_1,x_2,x_3,x_4) \mapsto (x_1-x_3, x_2-x_4)$. Due to our mistake the network had to sacrifice half of its bandwidth to account for it. At worst such a mistake would double the number of required features and therefore quadruple the memory and computation requirements.

The method we propose is to reabsorb the skip before the ReLU activation. Given the ReLU function r

$$r : \mathbb{R}^n \to \mathbb{R}^n \qquad r(x)_i = \begin{cases} x_i, & \text{for } x_i \geq 0 \\ 0, & \text{for } x_i < 0 \end{cases} \qquad \frac{\partial r_i}{\partial x_j} = \delta_{ij} \cdot \mathcal{X}_{x_j \geq 0}$$

The network $r \circ (F_{\theta_1} + \mathbb{1}) \circ r \circ (G_{\theta_2} + \mathbb{1}) \circ r \circ (H_{\theta_3} + \mathbb{1})$ has gradients

$$x_1 = r(H_{\theta_3}(x_0) + x_0) \qquad x_2 = r(G_{\theta_2}(x_1) + x_1) \qquad x_3 = r(F_{\theta_1}(x_2) + x_2) \qquad y = c(x_3)$$

$$\left.\frac{\partial c}{\partial \theta_1}\right|_{x_2} = \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \frac{\partial r}{\partial x} \cdot \left.\frac{\partial F}{\partial \theta_1}\right|_{x_2,\theta_1}$$

$$\left.\frac{\partial c}{\partial \theta_2}\right|_{x_2} = \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \frac{\partial r}{\partial x} \cdot \left[\left.\frac{\partial F}{\partial x}\right|_{x_2,\theta_1} + \mathbb{1} \right] \cdot \frac{\partial r}{\partial x} \cdot \left.\frac{\partial G}{\partial \theta_2}\right|_{x_1,\theta_2}$$

$$\left.\frac{\partial c}{\partial \theta_1}\right|_{x_2} = \left.\frac{\partial c}{\partial x}\right|_{x_3} \cdot \frac{\partial r}{\partial x} \cdot \left[\left.\frac{\partial F}{\partial x}\right|_{x_2,\theta_1} + \mathbb{1} \right] \cdot \frac{\partial r}{\partial x} \cdot \left[\left.\frac{\partial G}{\partial x}\right|_{x_1,\theta_2} + \mathbb{1} \right] \cdot \frac{\partial r}{\partial x} \cdot \left.\frac{\partial H}{\partial \theta_1}\right|_{x_0,\theta_1}$$

Where each r derivative is just a diagonal projection onto the space of activated features. As before we get terms that are first order in F, G, H but now they are also third order in Dr. The obvious downside of this approach is that we lose this gradient information if the projections do not line up. The model would have to learn to keep features lined up between layers so that there is straight shot whenever possible. Alternatively the activation rate would need to be high compared to the number of layers (e.g 4 blocks would need an activation rate of over 75% to guarantee first order information leaking through for the bottom most layer).

A good solution is to employ a combination of both regular skipped connections between blocks along with these partial skipped connections within blocks. We just need to make sure that each block has an appropriate range (don't terminate a block with a ReLU activation).

6.3 Normalization

Normalization is the process of taking a set of variables with a given set of statistics and modifying them to have a more desired set of statistic in such a way that the information conveyed by the variables is preserved as accurately as possible. Given a vector $x \in \mathbb{R}^n$ with coordinates x_i, and a small regularization number ϵ, we define the normalization function N to be:

$$\mu(x) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma(x,\epsilon) = \sqrt{\epsilon + \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu(x))^2}$$

$$N(x) = \frac{x - \mu(x)}{\sigma(x,\epsilon)}$$

That is we shift the coordinates so that their new mean is 0 and their new variance is 1. The parameter ϵ is there to keep us from dividing by zero and is generally around 10^{-5}, the actual value does not matter too much. An important consideration when normalizing is that we lose two features worth of information due to the two imposed constraints. However if we apply a linear mapping to our normalized data the statistics μ and σ remain (mostly) known, this means that repeated normalization doesn't cause catastrophic data loss in large networks that alternate between linear layers and normalization. (This is not a proof by any means because the coordinates x_i in a network are not independent random variables). Finally most machine learning libraries allow you to adjust the output by applying an affine transformation $x_i \mapsto (x_i + b_i) \cdot w_i$, where b and w are learned weight and bias vectors. If the normalization layer feeds into an affine linear layer (which it almost always does) then this process is mathematically redundant. It might make a small difference when factoring in the effects of weight decay, it also amplifies the learning rate for the gradient in this particular direction. At least for the purpose of evaluation, it is best that the weight and bias of the normalization are factored into the adjescent affine layer.

Looking at our specific network, each layer is composed of a four dimensional tensor: $x \in \mathbb{R}^{B\times C\times H\times W}$ where B is the batch size, C is the number of features, H is the height and W is the width. The difficulty involved in normalization is not the function but rather choosing the set over which we normalize.

6.4 Batch Normalization

For batch normalization we break the vector into C subvectors, one for each channel

$$x^i \in \mathbb{R}^{B\times 1\times H\times W} \qquad 1 \leq i \leq C$$

then we apply the normalization algorithm to each x^i and then glue them back together. The idea behind batch normalization is that if our batch size is sufficiently large, then the statistics of the batch will be representative of the statistics of the total population for each individual channel. In this ideal case, batch normalization does not end up losing any information at all since the batch mean and variance become equal to the population mean and variance (a constant). In evaluation mode, batch normalization does not compute the batch statistics (since the model might be evaluated with batch size of 1) but instead uses a running average of the mean and variance seen during training. This can be a significant increase in performance. However it may not be possible to create batches large enough to represent the total population due to GPU memory limitations, sharding large batches across multiple GPUs and then computing batch statistics across them is one possiblity, however this might involve an excessive amount of GPU synchronization depending on the model. Another possible solution is to track batch statistics from the previous itteration and then use them to compute a biased estimate for the mean and variance of each subbatch. As an example if the batch of size B is split across into n mini batches, the normalization step at the ith iteration for the jth mini batch would be:

$$\begin{aligned}
\hat{\mu}_{i,j} &= \mu(x^j) \\
\mu_{i,j} &= \frac{\hat{\mu}_{i,j}}{n+1} + \frac{1}{n+1}\sum_k \hat{\mu}_{i-1,k} \\
\hat{\sigma}_{i,j} &= \sigma(x^j, \epsilon) \\
\sigma^2_{i,j} &= \frac{\hat{\sigma}^2_{i,j}}{n+1} + \frac{1}{n+1}\sum_k \hat{\sigma}^2_{i-1,k} \\
N(x^j) &= \frac{x^j - \mu_{i,j}}{\sigma_{i,j}}
\end{aligned}$$

This is only a sketch of the idea and it will take some work to stop the statistics from drifting over time. We can think of this algorithm as a first order approximation, we can make the algorithm more robust by computing higher order terms (the velocity and acceleration) of $\hat{\mu}$ and $\hat{\sigma}$ as functions of i and factoring them in:

$$\mu_i \approx \frac{5}{2}\mu_{i-1} - 2\mu_{i-2} + \frac{1}{2}\mu_{i-3}$$

These numbers look a bit like magic but it's really simple. The velocity at time i is estimated by $v_i = (\mu_i - \mu_{i-1})/\Delta t$, the acceleration at time i is estimated by $a_i = (v_i - v_{i-1})/\Delta t$ and the predicted average is given by $\mu_i = \mu_{i-1} + v_{i-1}\Delta t + \frac{1}{2}a_{i-1}\Delta t^2$ where Δt cancels and we expand everything in terms of μ_i

Using batch normalization with undersized batches leads to instability during training (since the statistics of unrelated samples become coupled together) and the network produces different results depending on which inputs are bundled together. The model will spit out complete garbage in evaluation mode because the moving average will often times not be close to the mini batch mean.

6.5 Local Feature Normalization

This type of normalization is very rarely used because most of the time others are better. We break the vector into $N \times H \times W$ subvectors and compute statistics across the channel for each individual coordinate:

$$x^i \in \mathbb{R}^{1\times C\times 1\times 1} \qquad 1 \le i \le N \times H \times W$$

As mentioned earlier this leads to a significant amount of data loss when C is small. We use local feature normalization with a learnable weight parameter right before passing our output into the softmax function. Since the softmax function is mean invariant, removing the mean does not actually lose any information. Since our softmax function ideally outputs a single large logit, the scale coming in doesn't matter that much either. We used local normalization so that the variables fed into the softmax function were as regularly distributed as possible, that way the gradients would be most stable (softmax contains a bunch of e^x terms that want to blow up and ruin your day)

6.6 Group Normalization

For group normalization we break our channel into G groups containing C/G features each. We compute the statitsics across the group and the spatial dimensions (but not batch). The special case where $G = C$ is called instance normalization. The latter has the issue that an input that simulataneously actuates a feature at every spatial position would have that type of activation suppressed. In some cases that behaviour is desirable but in our case using group normalization seemed to work better. Group normalization allows you to pool information across channels rather than the batch in order to make the normalization more consistent. By splitting the features into groups, the network can learn to group related features together in order to better preserve their statistics. If all the features end up in one large group, this becomes layer normalization.

7 Local Attention

The code behind implementing local attention looks a bit daunting but the idea is relatively simple. Our goal is to perform the attention algorithm between different pixels, however we want to limit ourselves to pixels that are in a small window so that the computation does not become massive. We start by fixing a block size which is either 2 or 4, it needs to divide both the height and the width. Next we partition the spatial dimensions by the block size. For example if the block size is 2 and the width and height are both 32, this would take a 32×32 vector to a $16 \times 2 \times 16 \times 2$ vector. It would be possible to run attention on each subblock of 2×2 pixels but this would not be very effective. There would be no way for a pixel in the bottom left corner of the block to see it's immediate neighbours below or to the left. To solve this issue we make the key subblocks 3 times bigger in both spatial directions. We do this by first padding the boundary with blocks, then for each key subblock we concatenate the 8 adjescent blocks. This process satisfies the expression (with x equal to zero where it would be out of bounds)

$$f(x)[h, w, i + n \cdot b, j + m \cdot b] = x[h - 1 + n, w - 1 + m, i, j] \qquad 0 \leq i, j < b, \quad 0 \leq n, m \leq 2$$

Next we have to compute a suitable mask for the weight matrix. The first part of the mask is simply setting the weight to $-\infty$ for keys that are out of bounds. The second part of the mask assigns a learned parameter for every relative coordinate. i.e the value of the weight matrix for query at local coordinate (x_1, y_1) and key local coordinate (x_2, y_2) would be biased by some number depending on $(x_1 - x_2)$ and $(y_1 - y_2)$, we precompute this mask to greatly speed up inference. Note that for a block size of b, we have $9b^4$ components in the weight matrix but only $16b^2$ unique pairs of offsets. We compute a mask to represent where each parameter in the smaller matrix appears in the bigger matrix then copy them into place using a tensor product at run time. Note that this implementation of local attention is not completely symmetric. For $b = 2$, a query at horizontal position 0 can see two keys to the left but three keys to the right. We can make this process symmetric by masking out relative coordinates whose offset is larger than the block size, however this does throw away 'valid' information. We ended up choosing to keep this information and letting the network learn to ignore it if necessary

7.1 Computational Complexity

To simplify the expression we will make the height and width equal to W, the input and output features equal to N, the number of heads n times the key dimension d also equal to N and the block size equal to b. For pure numerical complexity we have:

- query, key, value projections: $O(3 \cdot H^2 \cdot N^2)$
- output projections: $O(H^2 \cdot N^2)$
- weight matrix computation: $O(9 \cdot b^2 \cdot H^2 \cdot N)$
- softmax computation: $O(9b^2 \cdot n \cdot H^2)$ (with larger factor for computing exponetials and division)
- weight to value: $O(9b^2 \cdot H^2 \cdot N)$

Looking at complexity alone it might seem that local attention has a similar computation cost to a convolution with kernel size $b + 1$, however it ends up being significantly slower. This is because we are not factoring in the cost of all the extra memory shaping steps we have to do. Furthermore carrying out a large number of matrix multiplications is less efficient than carrying out one large matrix multiplication, again due to memory alignment and algorithm efficiencies. Introducing local attention reduced loss by 20% but drastically increased training times to the point where we were better off simply excluding it and increasing the number of features by 50% instead. The impact of local attention is highest for the outermost layers due to the large spatial resolution. If we were to build a larger model with more data and more training resources, using local attention in 2 or 3 of the deepest UpBlocks might be worth it. Better hardware optimization and driver level support might one day make local self attention a viable upgrade. We did not exhaust the possibility that, after training to exhaustion, scaling the model size ends up inferior to implementing local self attention. At the very least though we determined that a 64 base feature size model with local attention was less effective than a 96 base feature size model with an extra convolution step in each upBlock, both models were roughly 500MB but the local attention model had 33% longer training itterations.

7.2 FLOOD and SEGMENT Algorithms

The FLOOD algorithm is a fairly straightforward implementation of floodfill. Starting at a cell, we claim it, then consider the four orthogonal cells. If the cell is unclaimed we claim it and repeat the process. If the algorithm finds a boundary cell, it claims it but a boundary cell is not allowed to claim another boundary cell. This means that the flood can only be stopped by a boundary two cells thick. Since the symmetric discrete gradient of our target segmentation produces boundaries that are two cells thick, this lets us perfectly recover the shape of the boundary. If we had used a simple discrete gradient, then the position of the boundary would end up ambiguous. The proposed algorithm is not particularly efficient, it performs roughly four times as many comparisons and uses four times the memory than strictly necessary.

Starting with the first unclaimed cell, we use FLOOD to generate a mask, then add the mask to stack if it is sufficiently large, then we repeat the process until every pixel has been claimed. Each pixel now belongs to at MOST one mask in the stack.

7.3 DUST_FILL Algorithm

The resut of our previous segmentation was a series of masks on which we will make the image segmentation constant. However these masks do not completely cover the image and there may be sizable gaps in between. Furthermore these masks may completely exclude small features like jewelry. The Dust fill algorithm solves both of these problems in an efficient manner. To start we need to figure out the statistics of each mask. To that end we take our segmentation and sum it over each mask along the spatial axes. This gives us a 2D array that stores the sum of probablities of each mask and label. Next we create a new mask for each label and initialize it to 0. Similarly we set the stats of the mask to be 1 for the corresponding label and 0 otherwise.

In the first pass of our algorithm we look at each unmasked pixel: if the label of highest probability for that pixel exceeds a preset threshold, we "promote" the pixel to the corresponding label mask. Otherwise we add the pixel coordinates to a stack along with patience value of 1. Finally we create a new stack to be used as a swap buffer and begin the main loop. For each pixel in the stack we check if it borders any masks. If it does not then we put the coordinate in the swap buffer and continue. If it does border at least one mask, we compute its corresponding normalized vector of label probabilities and compare it to the normalized vector of label probabilities for that mask. We do this for each mask it borders (a maximum of four) and consider only the closest mask. If the distance between probability vectors is less than a given threshold multiplied by patience then we add the pixel to the corresponding mask. If the mask is not one of our promoted masks we also add the probability vector to the corresponding stat vector. If the distance is greater than the threshold, we put the coordinate back into the swap buffer and multiply the patience value by our impatience parameter. This guarantees that a pixel will eventually be added to some mask. Once the stack is exhuasted we swap it with the swap buffer and repeat the process until every pixel has been assigned to a mask. We use L_1 norm in our implementation because the segmentation array is already L_1 normalized and it is marginally faster to compute. Finally the statistics are used to assign the most probable label to each mask and combine together to yield a segmentation.

By adjusting the join threshold and impatience parameter we can ensure that pixels are not simply mapped to the closest mask but instead have a chance to find the mask that has the closest statistics. The algorithm run time is proportional to the number of unmapped pixels and the orthogonal pixel distance between masks, in practise it is quite fast compared to floodfill. After some basic optimizations, the post processing adds less than a second on average to the execution time, the performance degrades to a few seconds if the boundary segmentation is extremely noisy.

Algorithm 1 Basic Gradient Descent Algorithm

```
Require: $k \in \mathbb{N}$                                   ▷ number of labels
Require: $F_\theta : [0,1]^{3\cdot256\cdot256} \to [0,1]^k$   ▷ model with adjustable parameters $\theta$
Require: $S \subset [0,1]^{3\cdot256\cdot256}$                ▷ subset of images with known labels
Require: $g : S \to [0,1]^k$                                  ▷ function that maps elements of S to their labels
Require: $0 < \lambda \in \mathbb{R}$                         ▷ learning rate
Require: $n \in \mathbb{N}$                                   ▷ maximum itterations
Require: $\theta_0 \in \mathbb{R}^m$                          ▷ initial guess for $\theta$, can be set to random
  $i \leftarrow 0$
  while $i < n$ do
      $L_i(\theta) \leftarrow \sum_{s\in S} ||F_\theta(s) - g(s)||$
      $d\theta \leftarrow \nabla_\theta L_i(\theta)|_{\theta_i}$
      $\theta_{i+1} \leftarrow \theta_i - \lambda d\theta$
      $i \leftarrow i + 1$
  end while
```

Algorithm 2 FLOOD

```
Require: $W$ : INT                                              ▷ Field Width
  Enumerate:
      $$E = \left\{ \begin{array}{rl} \text{CELL} & = 0, \\ \text{BOUNDARY} & = 1, \\ \text{CLAIMED} & = 2 \end{array} \right\}$$
Require: $H$ : INT                                              ▷ Field Height
Require: data: array$[0, H-1][0, W-1]$ of E      ▷ 2D array of enumerated cells
Require: mask: array$[0, H-1][0, W-1]$ of E                      ▷ Empty Mask
Require: $x$: INT                                         ▷ seed x coordinate
Ensure: $0 \leq x < W$
Require: $y$: INT                                         ▷ seed y coordinate
Ensure: $0 \leq y < H$
Ensure: mask $= 0$
Ensure: data$[y, x] \in \{\text{CELL}, \text{BOUNDARY}\}$
  Declare stack : stack of (INT, INT, INT)
  stack.push($(y, x, 0)$)
  Declare filled ← 0
  while size(stack) > 0 do
      y, x, b ← stack.pop()
      if $x < 0$ or $x \geq W$ or $y < 0$ or $y \geq H$ then
          continue
      end if
      if data[y,x] = CELL or (data[y,x] = BOUNDARY and b = 0) then
          b ← 1 if data[y,x] = BOUNDARY else 0
          data[y,x] ← CLAIMED
          stack.push($(y + 1, x, b)$)
          stack.push($(y - 1, x, b)$)
          stack.push($(y, x + 1, b)$)
          stack.push($(y, x - 1, b)$)
          filled ← filled +1
      end if
  end whileOutput: filled
```

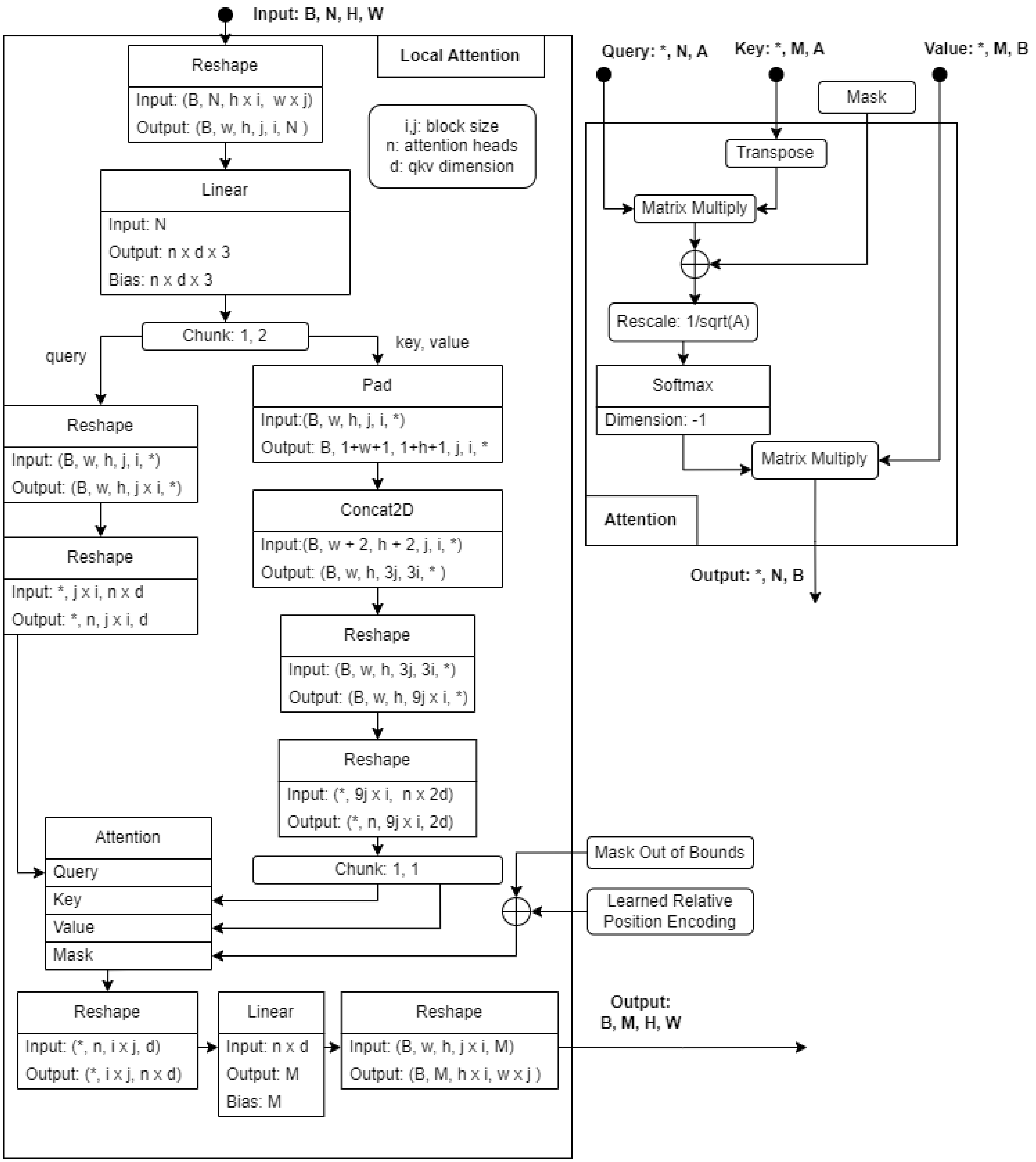
Algorithm 3 SEGMENT

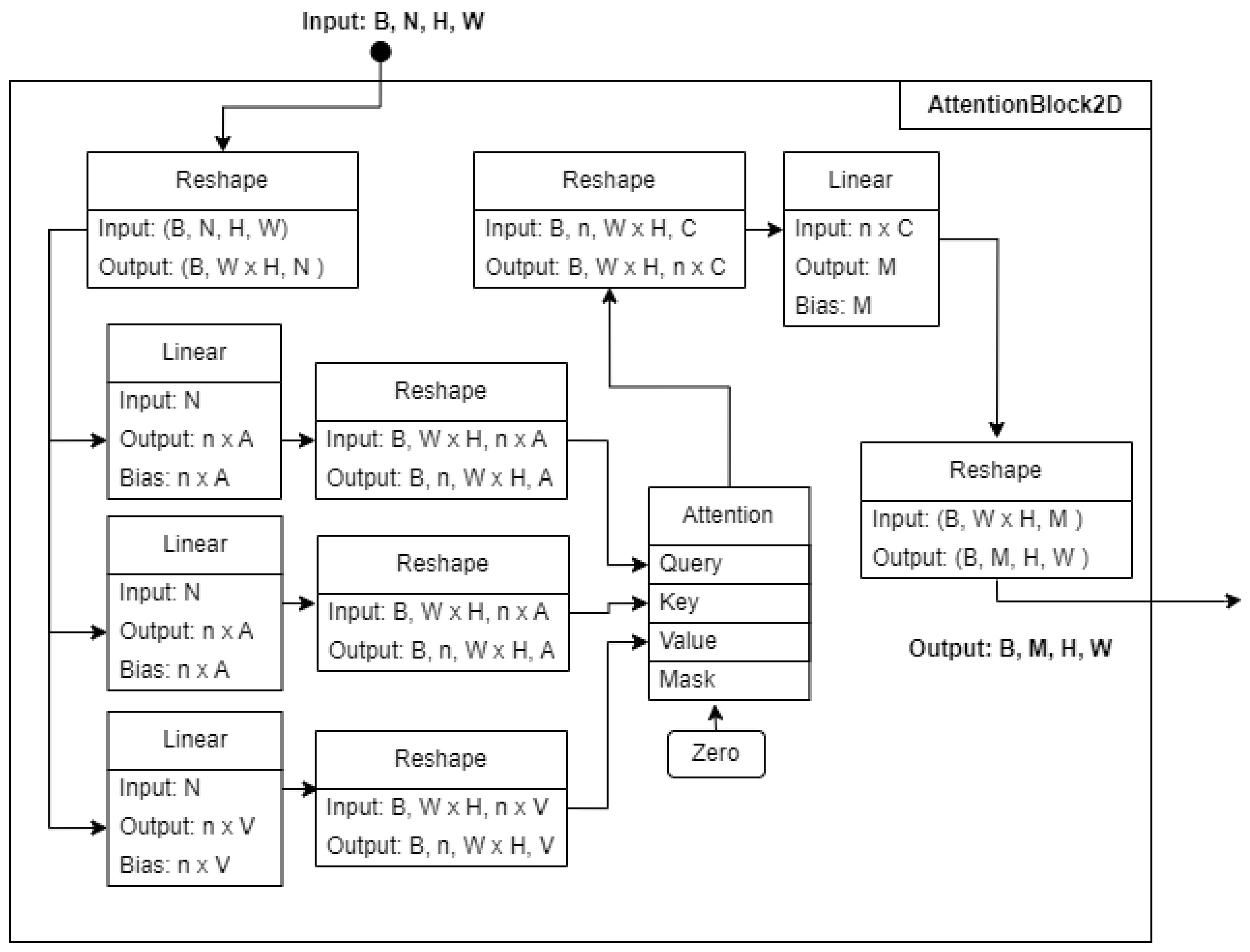
```
Require: $W$ : INT                                                    ▷ Field Width
Require: $H$ : INT                                                    ▷ Field Height
Require: $T$: INT                  ▷ Dust Threshold: minimum mask size to put in stack
Require: data: array$[0, H-1][0, W-1]$ of E            ▷ 2D array of enumerated cells
  Declare masks : stack of ( array$[0, H-1][0, W-1]$ of INT )
  (y, x) ← any of (s,t) such that data[s,t] = 0 or data[s,t] = 1
  while exists (y, x) do
    Declare mask : array$[0, H-1][0, W-1]$ of INT ← 0
    mass ← FLOOD(W, H, &data, &mask, x, y)            ▷ modifies data, mask in place
    if mass ≥ T then
      masks.push(mask)                   ▷ Add masks to the stack if they are large enough
    end if
    (y, x) ← any of (s,t) such that data[s,t] = 0 or data[s,t] = 1        ▷ Get Next Point
  end while
  Output masks
```

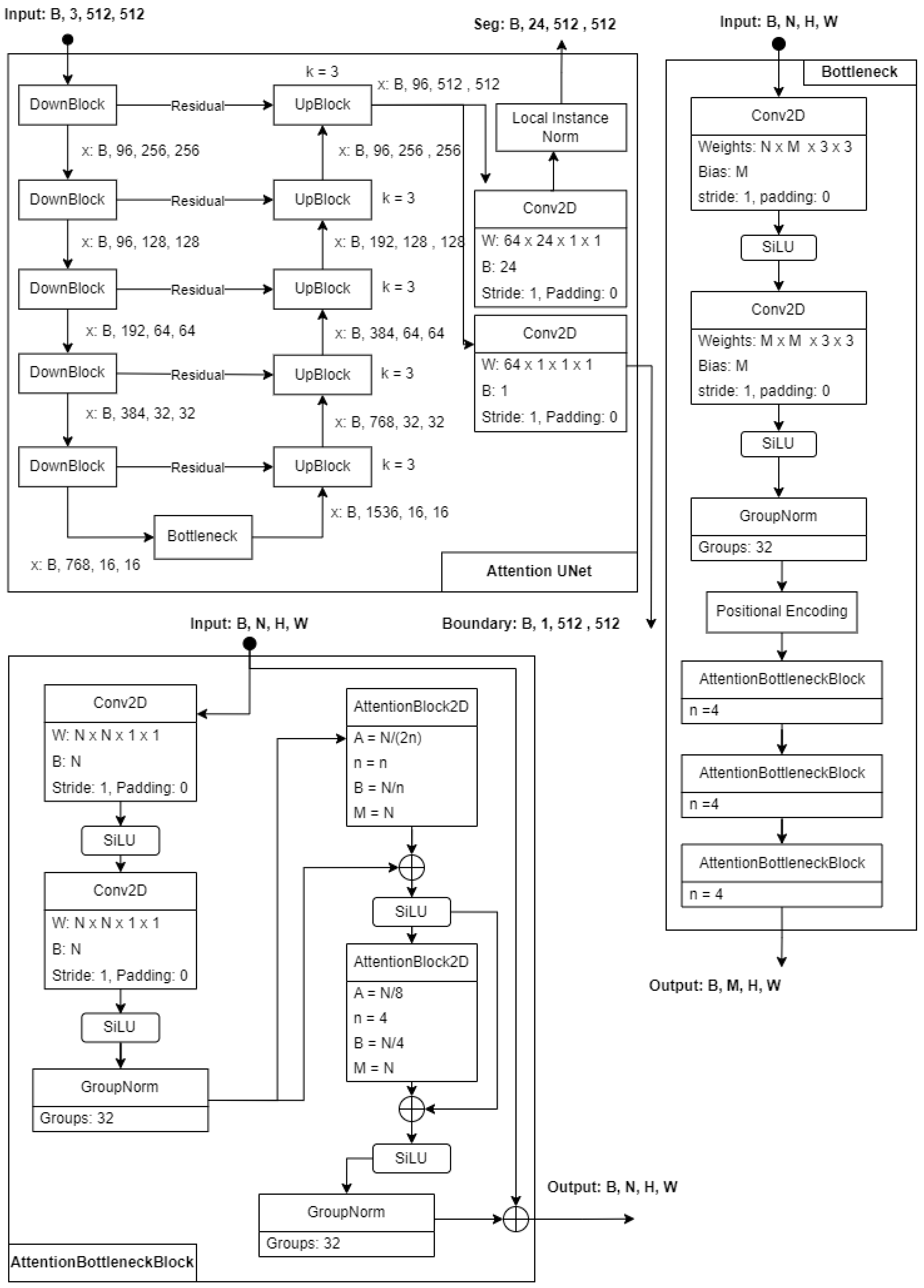
Algorithm 4 DUST_FILL

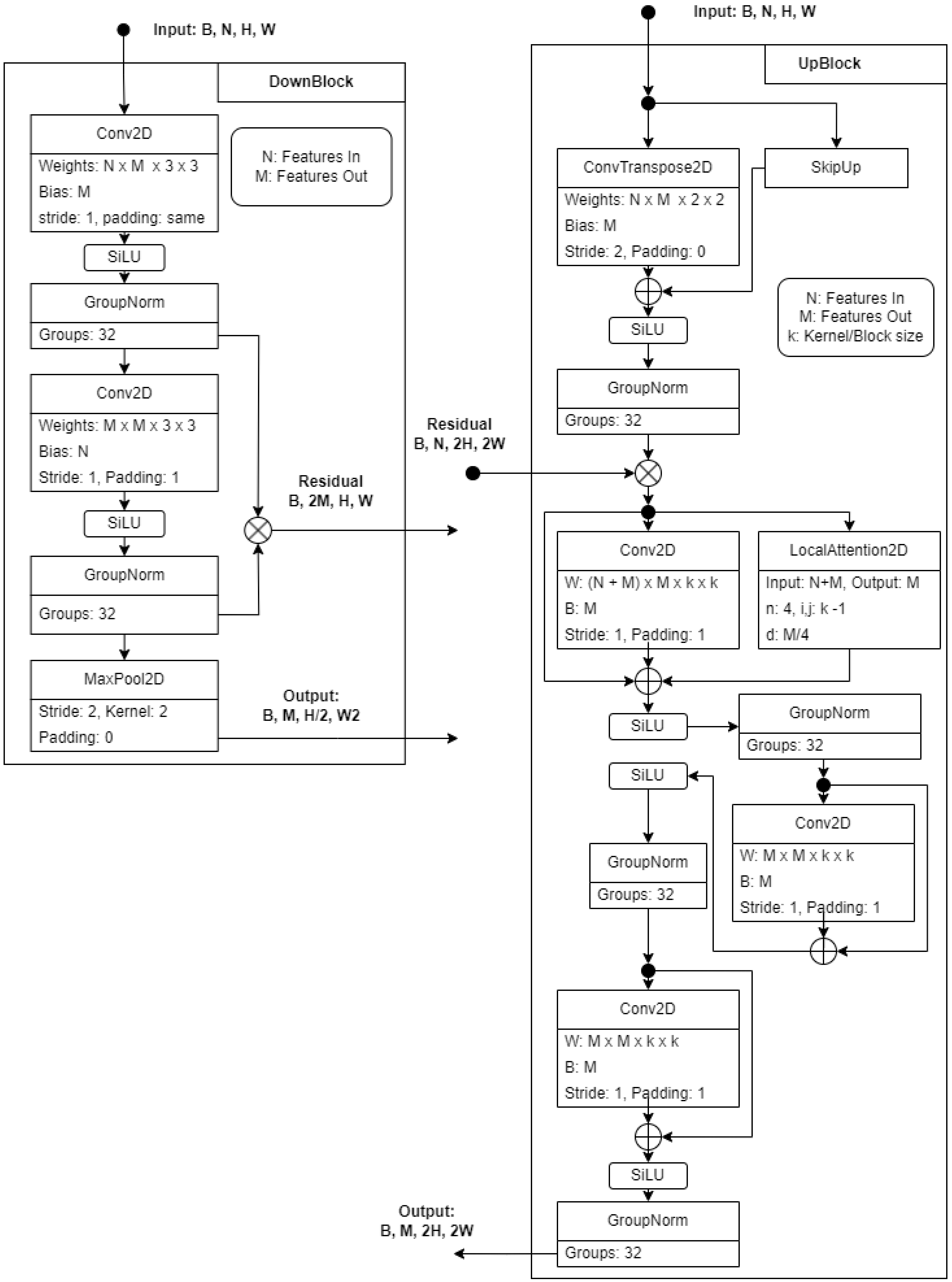
```
Require: $W$ : INT                                          ▷ Field Width
Require: $H$ : INT                                          ▷ Field Height
Require: $L$: INT                                           ▷ Number of Labels
Require: seg: array$[0, H-1][0, W-1][0, L-1]$ of FLOAT      ▷ Label probability of each pixel
Require: m: stack of ( array$[0, H-1][0, W-1]$ of INT )
Require: new_threshold: array $[0, L-1]$ of FLOAT           ▷ Threshold to promote pixel to mask
Require: join_threshold: FLOAT                              ▷ Minimum Distance to join pixel to a mask
Require: impatience: FLOAT                                  ▷ Relaxes join threshold each cycle
  Declare $M \leftarrow$ size(masks)
  Declare masks : array$[0, M+L-1][0, H-1][0, W-1]$ of INT
  Declare stats: array$[0, M+L-1][0, L-1]$ of FLOAT
  Declare mapping: array$[0, H-1][0, W-1]$ of INT ← -1
  Declare $S = \{(s,t) \in (\text{INT}, \text{INT}) : 0 \leq s < H, \quad 0 \leq t < W\}$    ▷ Define for itteration
  for $k \leftarrow 0$ to $M-1$ do
      mask ← m.pop()
      masks[k] ← mask
      mapping[s, t]← $k$ if  mask[s,t] = 1  for  $(s,t) \in S$
      stats[k] ← $\sum_{(s,t)\in S}$ seg[s,t] · mask[s,t]        ▷ Accumulate pixel statistics for each mask
  end for
  for $k \leftarrow 0$ to $L-1$ do
      masks[M + k] ← 0                                    ▷ Allocate a mask for each label for promotion
      stats[M+k] ← 0
      stats[M+k, k] ← 1                                   ▷ These fixed masks have constant stats
  end for
  Declare stack: stack of (INT, INT, FLOAT)               ▷ Stack of pixels to sort, and their patience
  Declare swap: stack of (INT, INT, FLOAT)
  for $(s,t) \in S$ do
      if mapping[s,t] ≥ 0 then                            ▷ Skip mapped pixels
          continue
      end if
      Declare label ← argmax(seg[s,t])
      if seg[s,t, label]≥ new_threshold[label] then       ▷ Promote Pixels that exceed threshold
          mapping[s,t] ← label
          masks[M + label, s, t] ← 1
      else
          stack.push((s,t, 1.0))                          ▷ Mark for further processing
      end if
  end for
```

```
while size(stack) $> 0$ do
    while size(stack) $> 0$ do
        (y, x, patience) $\leftarrow$ stack.pop()
        dist $\leftarrow$ 1000                                        ▷ Smallest distance
        mask $\leftarrow -1$                                          ▷ Closest mask by distance
        border $\leftarrow$ 0                     ▷ Flag whether the pixel is adjescent to a mask
        for $(s,t) \in \{(y+1,x),(y-1,x),(y,x+1),(y,x-1)\}$ do
            if InBounds(s, t) and mapping[s, t] $\geq 0$ then
                border $\leftarrow$ 1
                Declare tempm $\leftarrow$ mapping[s, t]
                Declare tempd $\leftarrow$ Norm(Normalize(seg[y,x]) - Normalize(stats[tempm]))
                if tempd< join_threshold $\cdot$ patience and tempd < dist then
                    dist $\leftarrow$ tempd
                    mask $\leftarrow$ tempm
                end if
            end if
        end for
        if border = 0 then
            swap.push((y,x, patience))          ▷ Patience doesn't change until we border a mask
            continue
        end if
        if mask $= -1$ then
            swap.push((y,x, patience $\cdot$ impatience))     ▷ Increases threshold for next itteration
            continue
        end if
        masks[mask,y,x] $\leftarrow$ 1
        mapping[y,x] $\leftarrow$ mask
        if mask < M then
            stats[mask] $\leftarrow$ stats[mask] + seg[y,x]              ▷ Accumulate pixel information
        end if
    end while
    stack, swap $\leftarrow$ swap, stack                                  ▷ Swap the stacks
end while
Declare output: array$[0, H-1][0, W-1]$ of INT
output$\leftarrow \sum_{k=0}^{M+L-1}$ masks[k] $\cdot$ argmax(stats[k])
Output output                          ▷ multiply each mask by its maximal label and output
```


Input: B, N, H, W
Local Attention
Reshape
Input: (B, N, h x i, w x j)
Output: (B, w, h, j, i, N)
i,j: block size
n: attention heads
d: qkv dimension
Linear
Input: N
Output: n x d x 3
Bias: n x d x 3
Chunk: 1, 2
query
key, value
Reshape
Input: (B, w, h, j, i, *)
Output: (B, w, h, j x i, *)
Pad
Input:(B, w, h, j, i, *)
Output: B, 1+w+1, 1+h+1, j, i, *
Reshape
Input: *, j x i, n x d
Output: *, n, j x i, d
Concat2D
Input:(B, w + 2, h + 2, j, i, *)
Output: (B, w, h, 3j, 3i, *)
Reshape
Input: (B, w, h, 3j, 3i, *)
Output: (B, w, h, 9j x i, *)
Reshape
Input: (*, 9j x i, n x 2d)
Output: (*, n, 9j x i, 2d)
Attention
Query
Key
Value
Mask
Chunk: 1, 1
Mask Out of Bounds
Learned Relative Position Encoding
Reshape
Input: (*, n, i x j, d)
Output: (*, i x j, n x d)
Linear
Input: n x d
Output: M
Bias: M
Reshape
Input: (B, w, h, j x i, M)
Output: (B, M, h x i, w x j)
Output: B, M, H, W
Query: *, N, A
Key: *, M, A
Value: *, M, B
Mask
Transpose
Matrix Multiply
Rescale: 1/sqrt(A)
Softmax
Dimension: -1
Matrix Multiply
Attention
Output: *, N, B


Input: B, N, H, W
AttentionBlock2D
Reshape
Input: (B, N, H, W)
Output: (B, W x H, N)
Linear
Input: N
Output: n x A
Bias: n x A
Reshape
Input: B, W x H, n x A
Output: B, n, W x H, A
Linear
Input: N
Output: n x A
Bias: n x A
Reshape
Input: B, W x H, n x A
Output: B, n, W x H, A
Linear
Input: N
Output: n x V
Bias: n x V
Reshape
Input: B, W x H, n x V
Output: B, n, W x H, V
Attention
Query
Key
Value
Mask
Zero
Reshape
Input: B, n, W x H, C
Output: B, W x H, n x C
Linear
Input: n x C
Output: M
Bias: M
Reshape
Input: (B, W x H, M)
Output: (B, M, H, W)
Output: B, M, H, W


Input: B, 3, 512, 512
Seg: B, 24, 512 , 512
Input: B, N, H, W
Bottleneck
DownBlock
Residual
UpBlock
k = 3
x: B, 96, 512 , 512
Local Instance Norm
x: B, 96, 256, 256
x: B, 96, 256 , 256
Conv2D
W: 64 x 24 x 1 x 1
B: 24
Stride: 1, Padding: 0
x: B, 96, 128, 128
x: B, 192, 128 , 128
x: B, 192, 64, 64
x: B, 384, 64, 64
Conv2D
W: 64 x 1 x 1 x 1
B: 1
Stride: 1, Padding: 0
x: B, 384, 32, 32
x: B, 768, 32, 32
x: B, 1536, 16, 16
Bottleneck
x: B, 768, 16, 16
Attention UNet
Boundary: B, 1, 512 , 512
Conv2D
Weights: N x M x 3 x 3
Bias: M
stride: 1, padding: 0
SiLU
Conv2D
Weights: M x M x 3 x 3
Bias: M
stride: 1, padding: 0
SiLU
GroupNorm
Groups: 32
Positional Encoding
AttentionBottleneckBlock
n =4
AttentionBottleneckBlock
n =4
AttentionBottleneckBlock
n = 4
Output: B, M, H, W
Input: B, N, H, W
Conv2D
W: N x N x 1 x 1
B: N
Stride: 1, Padding: 0
SiLU
Conv2D
W: N x N x 1 x 1
B: N
Stride: 1, Padding: 0
SiLU
GroupNorm
Groups: 32
AttentionBlock2D
A = N/(2n)
n = n
B = N/n
M = N
SiLU
AttentionBlock2D
A = N/8
n = 4
B = N/4
M = N
SiLU
GroupNorm
Groups: 32
Output: B, N, H, W
AttentionBottleneckBlock


Input: B, N, H, W
DownBlock
Conv2D
Weights: N x M x 3 x 3
Bias: M
stride: 1, padding: same
N: Features In
M: Features Out
SiLU
GroupNorm
Groups: 32
Conv2D
Weights: M x M x 3 x 3
Bias: N
Stride: 1, Padding: 1
SiLU
GroupNorm
Groups: 32
Residual
B, 2M, H, W
MaxPool2D
Stride: 2, Kernel: 2
Padding: 0
Output:
B, M, H/2, W2
Input: B, N, H, W
UpBlock
ConvTranspose2D
Weights: N x M x 2 x 2
Bias: M
Stride: 2, Padding: 0
SkipUp
N: Features In
M: Features Out
k: Kernel/Block size
SiLU
GroupNorm
Groups: 32
Residual
B, N, 2H, 2W
Conv2D
W: (N + M) x M x k x k
B: M
Stride: 1, Padding: 1
LocalAttention2D
Input: N+M, Output: M
n: 4, i,j: k -1
d: M/4
SiLU
GroupNorm
Groups: 32
SiLU
Conv2D
W: M x M x k x k
B: M
Stride: 1, Padding: 1
GroupNorm
Groups: 32
Conv2D
W: M x M x k x k
B: M
Stride: 1, Padding: 1
SiLU
GroupNorm
Groups: 32
Output:
B, M, 2H, 2W