UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Physics and Technology

Inference Guided Few-Shot Segmentation

Joel Burman

FYS-3941: Master's Thesis in Applied Physics and Mathematics - 30stp
June 2022

“If you can’t convince them, confuse them.”
–Harry S Truman

“It is more fun to talk with someone who doesn’t use long, difficult words but rather short, easy words like, ‘What about lunch?”
–Winnie the Pooh

Abstract

Few-shot segmentation has in recent years gotten a lot of attention. The reason is its ability to segment images from classes based on only a handful of labeled support images. This opens up many possibilities when the need for a big dataset is removed. To do this a few-shot segmentation network need to extract as much quality information from each support image as possible.

In this thesis we are exploring if an existing few-shot segmentation network can be improved by making the inference phase more target class specific. To do this we are introducing our Inference Guided Few-Shot Segmentation (IGFSS) method. It can be applied to an existing few-shot segmentation network. It changes the inference phase from a static network to one that adapts certain class specific parts of the network to each new target class. We tested our method with the Self-Guided Cross-Guided (SGCG) network as backbone. Here we optimized either the prototypes or the decoder. We used the Pascal dataset to compare the results from both methods. This is done on a fixed list from the dataset to be able to make a fair comparison.

In the 5-shot setup, where new classes are segmented based on 5 support images. Here we get a solid improvement when our method is applied to both the prototypes and the decoder. The mean IoU score was increased with 3.7% and 7.5% respectively. The dataset was analysed with regard to image and object distributions. This gives us a better understanding of the results of our IGFSS method.

While our IGFSS method does benefit all classes this could be a first step towards a Class-Adaptive Inference Guided Few-Shot Segmentation method.

Acknowledgement

Thanks you Michel for continuously guiding me through this masters thesis. You made the light in the end of the tunnel glow brighter with each meeting.

Thanks to the UiT Machine Learning Groups Ping-Pong team, for continuous motivation and creativity boosts throughout the whole process of this thesis.

Contents

List of Figures

List of Tables

1 Introduction

Image analysis has had a fast development in the last few years together with the rest of the field of machine learning. Improvement has come from a steady stream of new methods utilising new hardware-technologies that greatly increases computation capabilities. This has solved a lot of previous problems such as running image recognition networks in real time and being able to train large scale networks. Nowadays face detection networks can run easily in the background on a phone in real time without any problem. While large networks such as DALL-E [19] with 3.5 billion parameters that can produce high quality images from a written caption. Today the machine learning field is working on other bottlenecks that are preventing further improvement. One of them is the need for big labeled dataset. Without this training networks becomes much harder and often impossible to get good results.

1.1 Semantic Image Segmentation

In this thesis we will take a close look at the image recognition technique *semantic image segmentation*. This means that we want to classify each pixel in an input image to either a target class or background. The applications for semantic image segmentation range from self driving cars [44] to medical diagnostics [14][16]. Self driving cars need to make quick decisions from visual inputs to keep on the road as well as detecting and avoiding obstacles [22]. In the medical field there are a lot of employees working with analysing

different kinds of images while looking for diseases. If this could be completely automated or at least give the employees a helping hand we could increase the efficiency of the health system [15].

In recent years it has become more common that the absent of big quality datasets is the main bottleneck [57][58]. Since the machine learning models are learning from data, the performance of the model is highly correlated to the quality of the dataset it has trained on. Generating big labeled datasets is expensive, especially for image data labeled on a pixel level. This has led to an increased interest in methods that are less dependent on large labeled datasets for a specific task. There are many alternatives that remove the need for large labeled datasets.

Unsupervised learning removes the labels all together from the datasets. This way big datasets can still be accessed, but it is hard to implement good methods and keep track of what they have learnt and why. The score on segmentation tasks are generally much lower for unsupervised methods compared to supervised methods with limited training data [24].

Semi-supervised learning combines a small set of labeled data with a large set of unlabeled data. These work good for one specific task with additional unlabeled data. However they are not general enough to work on new classes with with just a few training images [63].

An other alternative is to take an existing network and re-train it, often called fine-tuning, on a new class. This can give good results if the new class is similar to what the network originally trained on. However to get good results on a new class it typically relies on more images to get good performance [45].

Few-shot learning is also a potential solution that addresses the problem with large labeled datasets. Here the focus is to be able to just need use a hand full of training images when a new class is introduced. This addresses both problems from unsupervised and semi-supervised learning.

1.2 Few-Shot Segmentation

To solve the need for large datasets for labeled data the field of few-shot segmentation has developed. In this thesis we will take a closer look at few-shot methods and dive into the field of few-shot learning. The main reason this field has gained an increased popularity the last few years is for its ability to learn from a very small dataset of the target class [57]. With just one training image from a new class we can still produce an accurate segmentation

map.

With few-shot learning we are shifting the mindset from "learning a task" to "learning how to learn tasks". So instead of having a fixed set of classes that the network can label, few-shot networks can be given a new target class to label. During the training phase the network will have be given small sets of images from the same class. Some labeled and some to label, this structure prepares the network to label unseen classes in the future. To do this we are still dependent on having large labeled datasets of many classes to train the models. Optimally the type of training inputs should be from a similar environment to the target class we want to segment in the end. This means that if the goal is to segment trees from satellite radar images, it is not preferred to train on real life images of cats and dogs.

When the model is trained we can introduce a new class with only one or a few labeled training examples. These training examples are often referred to as *support images*. Since the model has a focus on "learning how to learn" it can extract the information from the support images. This is then used by the model to check each pixel in the input image, *query image*, if they are belonging to the target class or background.

1.2.1 Limitations of Few-Shot Segmentation

Few-shot models are also dependent on a good training dataset for good performance. They need to be very general in order to adapt to new classes and tend to be class agnostic. It is still important that the classes the network has been training on is somewhat close to the classes it should be used on. Few-shot segmentation networks tend to have trouble if the type of images vary to much from training to testing [5].

During the training phase we want to get the network as general as possible to handle unknown new classes. Then we are interested in finding broad features in the training dataset. Using too specific information from each training example can make it hard to find common weights for the network that work good for all classes.

In the inference phase however we want to maximize usage of the information given from the few support images in the target class. So by using the network in the same setting for the training and inference phase we risk to oversee some information from the support set. Most few-shot segmentation do not change anything in the inference phase. Therefore they do not fully leverage the information from the support set. There have been some recent adaptations to change the structure of the network in the inference phase [5]. This however

has not been done for networks particularly designed for few-shot segmentation networks.

1.3 Research hypothesis

We hypothesize that it is very important to use the information from the few support images in the best possible way since they contain a new target class to the network.
Our hypothesis is that:
The results from a few-shot network can be improved by utilizing the information from the support set in the inference phase in a more extensive way. This can be done by changing certain class specific parts of the network by optimizing it to the current target class.

This would make it possible to explore different options after a computational costly training process has been made. That means that the method can be added to an existing pretrained model and potentially improve its results. However to utilize more information from the support set we are likely needing to increase the computations for each output from the inference phase.

1.4 Contribution

In this thesis we will explore the possibility to use the information from the support set in a better way. We will also take a in depth look at the dataset to be able to discuss the outcome from our results.

1.4.1 Primary contribution

We suggest to test our hypothesis by adding an addition step in the inference phase. Where the network adapts the weights in certain key parts of the network to the support images before the final prediction of the segmentation map is made. Training the whole network on the support set is not feasible. This would overfit the network quickly since there are simply to few images to train on. Instead we believe that by training certain key parts that are class specific we can improve the general performance without overfitting the network. We call our method *Inference-Guided Few-Shot Segmentation* (IGFSS). Our IGFSS will be applied to the Self-Guided Cross-Guided network (SGCG) network [57], a recent state of the art few-shot segmentation network. It uses prototypes to concentrate all the information from the support images and

uses them to predict a segmentation map. It also uses a non linear upscaling decoder to produce the segmentation map that could benefit from our IGFSS method. This makes the SGCG network a good candidate to test our if our hypothesis work on a established network.

To adapt the weights of the SGCG network we will use fine-tune either the prototypes or the decoder. The prototypes have the combined information from the support set and represents the target class as a whole. By changing the prototypes they have potential to improve the information given from the support set. The decoder is the last step for generating the final predicted segmentation map. So changing this to be more specific for the target class should have a potential big impact on the output.

The fine-tuning is done using the support images as a tiny training set to train and only update that given part. This will increase the computations needed for segmenting each image and have to be considered if the gained performance is worth it. In contrast to fine-tune the whole network fine-tuning just a single part of the network do not tend to overfit the network as fast.

To measure the improvement the results will be compared to the previous non fine-tuned method. We will do this continuously during the fine-tuning phase to get an understanding how it affects the results. This will give a direct comparison between the original model and different stages of our alternative model.

1.4.2 Secondary Contribution

We will also analyse the Pascal dataset [13]. We are not aware of an in-depth analysis of this dataset with regard to few-shot segmentation. We will look at how the classes are distributed on a image and pixel level. Starting with looking the amount of images in each class for an overview of the dataset. Then we will look closer at how different classes are distributed within the same image. Finally the size ratio between class objects will be explored to see if it can have any correlation with the result. This will give us a better understanding on the behaviour of the change in results for different classes.

1.5 Structure

This thesis is structured as follows:
In Part I we will discuss the background theory to for Few-Shot segmentation networks. This will be done by discussing the different important building

blocks needed. Starting from the basics of neural networks in Chapter 2 and how to optimize them in Chapter 3. We will then focus on networks for image recognition and segmentation in Chapter 4 and 5. Then an introduction to the few-shot learning field in Chapter 6 prepare us for more complex few-shot segmentation networks in Chapter 7. Parts of the content is based on the pre-thesis project [6].

In Part II we will take a closer look at our Inference Guided Few-Shot Segmentation (IGFSS) method in Chapter 8. We will test it on a recent state of the art Few-Shot segmentation model SGCG [57]. We look into how it can be improved by utilizing more of the information given from the support images by our IGFSS method. The Pascal dataset used for the experiments will be analysed in Chapter 9 for better understanding the results . We will then compare the results from our Inference Guided model to the original SGCG model in Chapter 10. Here we also discuss our improvements as well as how to develop it further in the future. Finally we sumarize out thesis in Chapter 11.

Part I

Background

2

Artificial Neural Networks

An artificial neural network (NN) can be seen as a simple function with multiple input values. The output size is fixed for a certain network but can vary a lot from one to several output values depending on the structure of the network. So a NN can be seen as a simple function

$$f_\theta(\mathbf{x}) = \mathbf{y}. \tag{2.1}$$

Here $\mathbf{x}$ is the input values, $\mathbf{y}$ is the output value(s) and f is the function or network itself with parameters $\boldsymbol{\theta}$. A NN is constructed of several smaller operations in a repeating pattern. These have parameters that need to be optimized with respect to each other for the NN to give a good output. There are linear operations with parameters $\boldsymbol{\theta}$ that can be updated and gives the network it is ability to adapt to the task. Then there are non linear operations that are predefined without parameters that are changed for the task. These non linear operations make sure that the whole network can find more then just a complex linear transformation of the input data.

2.1 Fully Connected Neural Networks

In a fully connected Neural Network (FCN) each of the linear operations is a multi dimensional function between the nodes in the network. The structure of a general FCN can be seen in Figure 2.1. Here each variable in the input $\mathbf{x}$ is put through a linear transformation $\mathbf{w}^T\mathbf{x} + b$ where $\mathbf{x} = [x_1, x_2, \dots x_n]^T$ and $\mathbf{w} = [w_1, w_2, \dots w_n]^T$ are column vectors and b is a scalar. The vector multiplication results in a weighted sum between the product of each weight, w_i, and it is corresponding x_i value. The result, z, from the linear transformation is a scalar that is sent to a non linear activation function, $\sigma(\bullet)$.

$$z = \mathbf{w}^T\mathbf{x} + b$$
$$h = \sigma(z) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

This is done for each node in each layer, with separate weights, $\boldsymbol{w}$, and bias, b, for each node. The process is repeated with the output from the previous layer as the input to the current layer. This is done through all the l hidden layers and finally the output $\hat{y}$ is calculated.

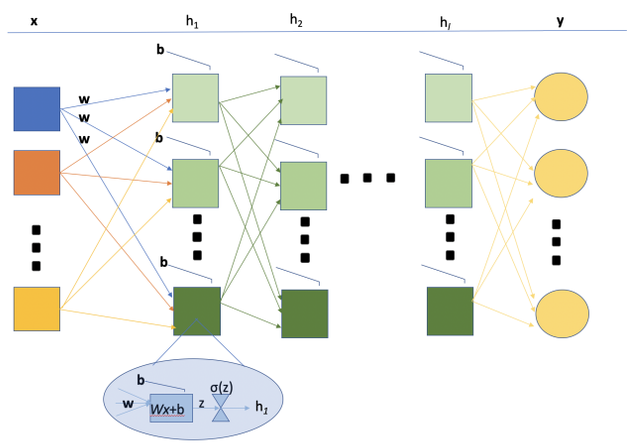


Figure 2.1: Structure of a fully connected neural network. With input $\mathbf{x}$ and output $\hat{y}$ and hidden layers $\mathbf{h}_1$ to $\mathbf{h}_l$ in between. Each box is a node with output calculated by a linear function $\mathbf{z} = \mathbf{wx}+b$ followed by a non linear activation function $\mathbf{h} = \sigma(\mathbf{z})$. The output form each node is used in the layer further right in the network until the output circles, y, is calculated.

More specifically we can look at Figure 2.1. On the left side the input, $\mathbf{x}$, is used to calculated a weighted sum for each node in the layer to the right, with different weights for each node. Then the non linear activation function is applied on each of the weighted sums. Note that each node uses all nodes in

the previous layers as input. This is then repeated for all the layers moving right until an output is achieved.

The operations between the nodes in the network can be summarized with:

$$h_r = \sigma(\mathbf{wh} + b)$$

Since each node in one layer has the same input, $\mathbf{x}$, but different weights, $\mathbf{w}$, we can summarize the weights to a matrix $\mathbf{W}$. This can be used to express all steps from the input $\mathbf{x}$ to the first hidden layer h_1 with $\mathbf{h}_1 = \sigma(\mathbf{Wx} + \mathbf{b})$. Or more general from step r-1 to r:

$$\mathbf{h}_r = \sigma(\mathbf{W}_r\mathbf{h}_{r-1} + \mathbf{b}_r) \tag{2.2}$$

Note that the bias, $\mathbf{b}$, is now represented with a vector containing the bias for each node in the first hidden layer. The activation function $\sigma(\bullet)$ is used on each value in the input vector so we get the resulting vector $\mathbf{h}_1$ out. This can then be repeated for each layer in the network and we have a quite compact way of representing the network.

$$\begin{aligned}
\mathbf{h}_1 &= \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \\
\mathbf{h}_2 &= \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2) \\
&\vdots \\
\mathbf{y} &= \sigma(\mathbf{W}_{l+1}\mathbf{h}_l + \mathbf{b}_{l+1})
\end{aligned}$$

In the end we can express the whole network with a chain of functions depending on the output from the previous functions. The structure with alternating linear functions with changeable weights and nonlinear static functions will give the network ability for very complex transformations using a set of simple equations.

2.2 Activation functions

Activation functions are non linear functions that are used after the linear transformation in each node in networks. This results in a non linear transformation

in each node of the network and makes it possible to find advanced feature representations from the input.

The input range and the derivative of activation functions need to be defined on the whole real line. This is to make sure that the network can handle all types of input. The derivative is used for optimizing the network and will be discussed further in the Section 3.2

There are several different activation functions, the most commonly used once are *Sigmoid*, *Tanh*, *Rectified Linear Unit (ReLU)* and *Softmax*. These we can see in the following table.

Activation function	f(x)	Output range
Sigmoid	$f(x) = \frac{e^x}{e^x+1} = \frac{1}{1+e^{-x}}$	{0, 1}
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	{-1, 1}
ReLu	$f(x) = max\{0, x\}$	$\{0, \infty\}$
Softmax	$f(x) = \frac{e^{x_i}}{\sum *J_{j=1} e^{x_j}}\ for\ i = 1, ..., J$	{0, 1}

Table 2.1: Most commonly used activation functions.

The softmax function is most commonly used for classification in the last layer in a network. It will calculate the estimated probability for an input for belonging to each of J different classes. Sigmoid and Tanh are used mostly in smaller networks since they have some problems for scaling up in deep networks, more on this in Chapter 3.5. ReLu is mostly used in deep networks since it is better at handling this problem.

3

Network Optimisation

In the network there is a need for different parameters for each linear transformation to get a flexible function with a fixed structure. This means that the amount of parameters are related to the width and depth (amount of nodes in each layer and number of layers) of the network. The amount of parameters quickly exceeds thousands even for smaller networks so manually adjusting them is not a viable option. Instead we want to optimize the network for a certain task by minimize the difference between the predicted result and true result. Since there are so many parameters optimizing the network by exploring the whole parameter space is extremely computational demanding. Instead we can look how the network is performing on data with known ground truth result, $\boldsymbol{y}$, and compare the ground truth with the output from the network, $\hat{\boldsymbol{y}}$. Then we can change the parameters in the network and see how it affects the results. By using some clever techniques known as *gradient decent* and *backpropagation* we can change the parameters so that the *loss* which is a function of the difference between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ gets smaller.

3.1 Loss function

By looking at the difference between $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ we can give a score on how well the network performed called *loss*. In Equation 3.1 we have a general expression for the *loss function* that does this

$$\mathcal{L} = \epsilon(\boldsymbol{y}, \hat{\boldsymbol{y}}).$$

There is a variety of *loss functions* that can be used to calculate loss. Different loss functions reward and penalize on different criterion when calculating the loss. Some common ones are mean square error (MSE) and cross entropy [61] loss

$$MSE = \sum_{i=1}^{m} (\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)^2 \tag{3.1}$$

$$CrossEntropyLoss = -\sum_{i=1}^{m} (\boldsymbol{y_i} ln(\hat{\boldsymbol{y}}_{\boldsymbol{i}})). \tag{3.2}$$

3.2 Gradient decent

Gradient decent is an iterative method to find a local minimum by taking repetitive steps in the opposite direction of the gradient. In our network we want to minimize the loss function, this can be seen as minimizing the difference between $\mathbf{y}$ and $\hat{\boldsymbol{y}}$. This is done by slightly changing all parameters in the network step by step to find lower values of the loss [23].

By changing the parameters in the negative direction of it is derivative the loss should decrease. Since we can not tell how much to move each parameter to find its optimal value we will change all by scaling the derivative with a value called *step size*. After all parameters have been updated the process can be repeated until the loss do not improve by decreasing in value. Then we know that we have found a local minimum. In Figure 3.1 we can see an example of a loss with only one parameter.

Depending on where we start on this line we might end up in a local minimum. In higher dimensions saddle points are more common then minimum points so this problem is not all to important for neural networks [21]. This effect also leads to that local minimums are generally closer in value to the global minimum. So even if we get stuck in a local minimum we will have similar results for the network compared to finding the global minimum.

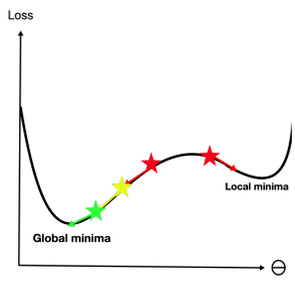


Figure 3.1: Example of gradient decent with one parameter, θ. Three steps, orange, yellow, green each reducing the loss to the global minimum. The single red star will get stuck in the local minimum.

3.2.1 Training in Batches

While the network is training we can choose how often we would like to update all parameters. We can first consider both extremes, update after each input or after all inputs. The first alternative will update after each input and is called Stochastic Gradient Decent (SGD) if the GD optimiser is used. The downside is that we will not know if the update is in the general direction of the training data since it is specific to just one example. This tends to give an unstable improvement of the networks performance. The second option, updating after each *epoch*, where epoch refer to running through the whole training set once. This will make sure that we update the parameters according to the general direction of the training data. This however is very slow as we only do one update for each epoch.

To get the benefits from both extremes a middle ground is often chosen. We can update the weights when a certain amount of samples has made it trough the network and the loss is calculated for each sample. This is referred to as training in *batches* where each batch has m samples. This can increase the training efficiency significantly when utilizing tensor multiplication for the whole batch at once [1][11]. The bottleneck of computational speed often comes from the memory limit in the GPU used for the computations. Larger batch size will generate larger tensor multiplications and for larger networks these becomes very large. In addition to computational speed the training will become smoother since it is updated with respect to a mean of each batch instead of every sample.

3.3 Adaptive Moment Estimation

One problem with Gradient decent is that it can be really slow, especially if the loss surface is relatively flat. For some sections it might be better with a longer step size. But if we use a too long step size we might get stuck overshooting a minimum point if the loss surface is to steep. A solution to this is to use an adaptive step size by using momentum. This means that we will use a portion of the previous steps first derivative and adding it to the next step. This makes the step size increase for continuous steps in the same direction. A well used algorithm for this is ADAM short for *Adaptive Moment Estimation* [25]. In addition to momentum ADAM also uses an adaptive factor, $\hat{v}$. This is similar to momentum, $\hat{m}$, but uses the second derivative. The adaptive factor is used to scale the momentum such that big changes in the second derivative gives lower effect of the momentum. The algorithm for this can be seen as the following

$$\begin{aligned}
\boldsymbol{g}_{i-1} &\leftarrow \nabla_{\boldsymbol{\theta}} \epsilon(\boldsymbol{\theta}_{i-1}) \\
m_{i-1} &\leftarrow \beta_1 m_{i-2} + (1-\beta_1)\boldsymbol{g}_{i-1} \\
v_{i-1} &\leftarrow \beta_2 v_{i-2} + (1-\beta_2)\boldsymbol{g}_{i-1}^2 \\
\hat{m}_{i-1} &\leftarrow \frac{m_{i-1}}{1-\beta_1} \\
\hat{v}_{i-1} &\leftarrow \frac{v_{i-1}}{1-\beta_2} \\
\boldsymbol{\theta}_i &\leftarrow \boldsymbol{\theta}_{i-1} - \frac{\gamma}{\sqrt{\hat{v}_{i-1}} + \xi}\hat{m}_{i-1}.
\end{aligned} \tag{3.3}$$

$\boldsymbol{g}_{i-1}$ is the gradients of the parameter θ. The gradient is used to calculate $\hat{m}_{i-1}$ and $\hat{v}_{i-1}$ by adjusting m_{i-1} and v_{i-1} since they are biased estimators of the gradient moments. β_1, β_2, ξ and γ are hyperparameters with default values suggested by the authors $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\xi = 10^{-8}$. The learning rate γ have to be adapted to the specific training case.

3.4 Backpropagation

When we know how the network is performed compared to the ground truth $\boldsymbol{y}$ we can calculate the derivative of the loss with respect to each parameter in the network. That will tell us how much each parameter contributed to the loss. To do this efficiently we use *backpropagation* [12], this utilises the chain structure of the network to calculate the derivatives. This structure gives parameters

dependencies to each other that can be used to calculate derivatives for one layer at the time and not the whole network. By using the chain rule while derivation we can calculate the derivative for each specific parameter in the network.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \frac{\hat{\boldsymbol{y}}}{\partial W_l} \\ &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \frac{\partial \sigma(z_l)}{z_l} \frac{\mathbf{W}_l^T \mathbf{h}_{l-1} + b_l}{\partial \mathbf{W}_l} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \sigma'(z_l) h_{l-1}\end{aligned}$$

Then for **b**:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{b}_l} &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \frac{\hat{\boldsymbol{y}}}{\partial b_l} \\ &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \frac{\partial \sigma(z_l)}{z_l} \frac{\mathbf{W}_l^T \mathbf{h}_{l-1} + b_l}{\partial \mathbf{b}_l} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{b}_l} &= \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{y}}} \sigma'(z_l)\end{aligned}$$

Starting from the end of the network we can calculate the derivatives for the last layer and use them to easily repeat the process for the previous layer. This way we can get all the parameters derivatives efficiently by calculating one layer at the time propagating backwards in the network.

3.5 Vanishing and exploding gradients

While doing the backpropagation in larger networks problems with *vanishing gradients* or *exploding gradients* can occur [3]. This problem comes from that the derivative is used in backpropagation over and over. So if the derivative is continuously smaller then 1 in magnitude the effect of the backpropagation will decrease exponentially. The opposite applies when the derivative is continuously larger then 1, then the effect will increase exponentially. In Table 3.1 we can see the range of the derivative of the most commonly used activation functions. Note that Sigmoid will always be less then 1 and Tanh will only be one while its output is close to 0.

This is the reason why ReLu [52] activation functions are commonly used

Activation function	Derivative range
Sigmoid	[0, 0.25]
Tanh	[0, 1]
ReLu	0 or 1

Table 3.1: Most commonly used activation functions and their derivative range.

in larger networks [1]. While Sigmoid and Tanh is often used in smaller networks.

3.6 Over fitting and Regularization

When the network has been trained on a dataset it will learn how to solve the tasks for that dataset. We would like the network to be general and work as well as possible on new unlabeled similar data. It is common that by training for to long we can get better and better results on the training data. But for new data the performance is decreasing after a while. We then say that the network is *over fitted* [55] on the training data. This means that the network has been to optimized to solve a certain task. We can think of this as memorising the answer to a question instead of understanding an overview of the subject.

To test this it is common to divide the labeled dataset into *training*, *test* and *validation* datasets. Here the network is trained on the training data and the network is updated from the errors to optimize all parameters in the network. Then the validation data is used for checking the performance of the network on "unseen" data. The networks are not updated from this. This can be used to choose what settings to be used for hyperparameters[1]. Finally the test dataset can be used as a final new unseen test after the whole training is completed. This is done by measuring how the network is performing on data that has never seen before. It represents a more realistic situation that is closer to a real life situation.

If the validation or test results indicated that the network is over fitting, there exists *regularization* techniques to improve the network. By regulating how the network is training we can increase the variety for the networks tasks. This can be done by introducing noise to the input of the network [4]. This prevents the network for being too reliant on specific parts of the input. The idea of not being reliant of parts of the network and still get good results comes again in other regularization techniques as well.

1. Parameters that is chosen before the training is stated and are fixed for the whole training session. Example: Batch size, training time and learning rate

3.6.1 Dropout

During training the network can sometimes use some weights or filters more than others. This could give a good result during training, especially on a small training dataset as it might not need all parameters. However this can cause bad result on the test dataset as the network do not become general enough to tackle new data. Here *dropout* [43] can make sure that the network can not rely on just some parts of the network. During training while using dropout the network will set a chosen rate of parameters to 0. After each round through the network a new set of parameters is set to 0. This will force the network become independent of any specific parameter for its predictions making it more general.

The training time while using dropout will increase as the optimization task becomes harder when not all parameters is updated [10]. The end result however can often be improved quite a bit on the test set making dropout well used in many networks.

3.6.2 Batch normalization

Another technique that utilizes the batch structure while training a network is *batch normalization* [20]. It can be used to improve its training properties by smoothing out the parameter landscape and make the optimization faster [38]. This also works as a regularization technique. The process is to normalize every batch that goes into a layer, this makes the different samples less spread. The mean is set to 0 and variance to 1 for the input X in each batch. Then two trainable parameters, γ and β are used to scale and shift the standardized batch.

$$\hat{\mathrm{X}}_i = \frac{1}{m}\sum_{i=1}^{m}\frac{\mathrm{X}_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{3.4}$$

$$\mathbf{Y}_i = \gamma\hat{\mathrm{X}}_i + \beta \tag{3.5}$$

During training μ is the mean of the batch with m samples in each batch. σ^2 is the variance of the batch and ϵ is small non 0 positive number that is preventing division by 0. γ and β are trainable parameters.

When the network is used in the test phase the global μ and σ^2 is optimal to use. These can be calculated by a running mean while the network is training to avoid extra calculations of computing it separately. This also gives the ability

to stop before all batches are used and still have the global μ and σ^2 for the batches used.

4

Convolutional Neural Networks

A convolutional neural network (CNN) [28] is mostly used for its good qualities on grid data inputs. They are able to process the information from all the pixels in an image to a latent feature space. This representation can then be used for classification, reconstruction or other tasks. In this chapter we will take a closer look on how convolutions work and how they can be used in neural networks. Then we will look at the architecture of some well used networks.

4.1 Convolutions in 2D

Convolution is done between two matrices, an image (larger matrix with information in each pixel) and a filter (often 3x3 size). The convolution can be seen as sliding the filter across the image and then calculate the sum of the pairwise multiplication for each position. This will result in a scalar that can be used as a single pixel value in the filtered image. The computation of the convolution follows three steps:

- The filter will be places on positions where it can overlap each value in the filter with the image.

- Then a pairwise multiplication is made between the filter and the corresponding pixels in the image.

- The sum of the multiplications is the resulting value in the filtered image, with corresponding position to where the filter was placed.

The filter can then be repositioned on the image until a filtered version of the whole image is calculated. These steps are summarized in Equation 4.3. Note that these equations are in context of CNNs and not convolutions in signal processing.

$$\mathbf{y} = \mathbf{x} * \boldsymbol{\omega} \tag{4.1}$$

$$y(a.b) = \sum_{i=0}^{h_\omega - 1} \sum_{j=0}^{w_\omega - 1} x(a+i, b+j)\omega(h_\omega + 1, w_\omega + j) \tag{4.2}$$

$$a = 1, \ldots, h_\mathbf{y}, b = 1, \ldots, w_\mathbf{y} \tag{4.3}$$

Here * is the notation for convolution, $\mathbf{X}$ is the input, $\mathbf{Y}$ is the output and $\boldsymbol{\omega}$ is the filter.

How many positions the filter is moved across the image for each calculation is called stride. For a two dimensional image one has to move with a stride of minimum [1,1], where each number represent the stride in different dimensions. The larger the stride is, the smaller the filtered image will become. Since by moving the filter more it will cover the image faster and get fewer outputs for the filtered image.

A few examples with sizes of inputs, filters and corresponding output can be seen in Figure 4.1

When the filter is moved to the edge of the image it has to stop to still overlap. This will result in a cropping effect on the filtered image, with the cropping size depending on the size of the filter. One way to handle the size reduction of the filtered image is to use padding, it can reduce or remove the size reduction. Padding will make the original image larger by adding extra pixels outside the original image. Most commonly the added pixels are filled with the value 0 and is then called zero-padding. The thickness of the extra edge can vary, to remove the cropping effect by adding half the filter size. By using padding the filter can go over all pixels of the original image the same way, but it will result in that the edges of the filtered image will get a frame of values close to the values in the padding.

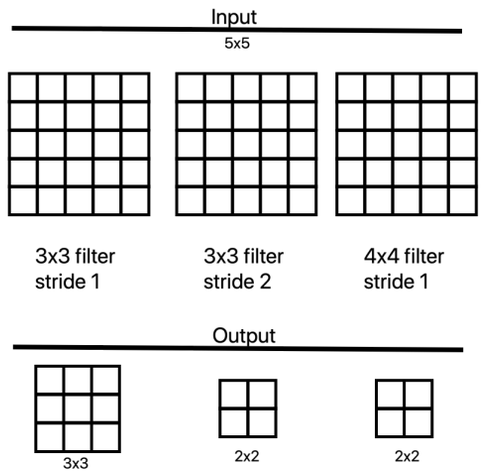


Figure 4.1: Example of sizes before and after 2D convolutions with different stride and filter size.

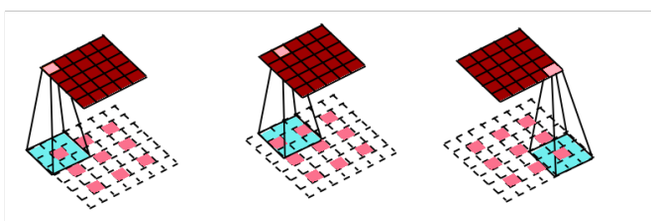


Figure 4.2: Illustration of transpose convolution or deconvolution. With three steps of the filter process.

There are several factors that will decide the final size of the filtered image. Equation 4.4 can be used to calculate it.

$$w_{out} = \frac{w_{in} + 2p - f}{s} + 1 \tag{4.4}$$

Where w_{out} is the filtered image size, w_{in} is the original image size, p is the amount of padding, f is the filter size and s is the stride.

There are also ways to do convolutions to increase the size of input image, this can, a bit simplified, be seen as backwards convolution. It is called *deconvolution* or *transpose convolution* [40] and is done by zero padding around each of the values in the image instead of the edge. As we can see in Figure 4.2 Then use normal convolution on the padded image.

4.2 Convolutional layer

In a convolutional neural network (CNN) [28] there are several convolutional layers. They are organized the same way as in a fully connected network (FCN) with weights changed to filters. In this section we will take a closer look on how convolutional layers work.

The connections works similar to a FCN with some key differences. The weight vector, $\boldsymbol{w}_i$, that transforms the input x_i to z_i is replaced with a filter for each weight scalar. The nodes in the network are replaced with *channels*, C. In a normal RGB the input channels are represented with a color, red, green or blue. Then each input channel are filtered with a set amount of filters in each layer. The amount of filters in each layer represent the amount of output channels that layer has. So that the output channels for one layers is the input channels for the next layer.

The convolution of a layer can be seen mathematically in Equation 4.5. Note that we still have an activation function $\sigma(\bullet)$ that will introduce non-linearity to the network.

$$\boldsymbol{Y} = \sigma(\Omega * \boldsymbol{X} + \boldsymbol{B}) \tag{4.5}$$

We want to use different filters in each layer of the network to find a variation of features throughout the network.

So in Figure 2.1 we replace $\mathbf{W}$ with filters of size $h \times w$. The bias term B will be a filter for each output channel. All of these can be optimized with backpropagation.

To filter an input with C_{out} amount of filters in one layer we will transform the input of size $C_{in} \times w_{in} \times h_{in}$ to $C_{out} \times w_{out} \times h_{out}$. We will need one filter for each of the channels in the output. This can be represented with a tensor of size $C_{out} \times W \times H$ that replaces the weight matrix, $\mathbf{W}$, from the FCN.

The benefit of this is that we can use a filter on the whole input image instead of having a single weight scalar for each pixel in the image. This largely decreases the amount of parameters needed for a CNN compared to a FCN for images.

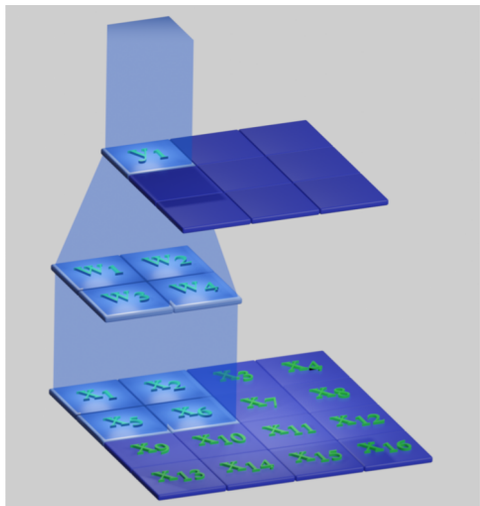


Figure 4.3: Visualization of convolution in the backpropagation in a CNN. Image from [32].

4.3 Back-propagation in CNNs

Backpropagation in CNNs are quite similar to FCNs that were discussed earlier with the main difference of the convolution operation. We can use the convolution in the backpropagation to calculate the gradients and update the parameters for each layer as before. In Figure 4.3 we can see how the convolution is used in backpropagation.

We will change the order of the convolution so that the output is filtered to the input. This will send one pixel through the filter to a larger grid. Thereby keeping the dependencies between layers from the forward pass.

4.4 Receptive field and non-linearity

When using convolutions on images with one filter only linear representations can be found of the input image. Non linear representations are found by using convolutional layers with activation functions. Then as the input image is filtered deeper in the network, more complex features are filtered out. By having long chains of filters also benefits in the way that each filter gets input from a larger part of the input image. Since we will compress the information of several pixels into one pixel and repeat this the *receptive field* of the layer will increase as the layers get deeper and deeper. The receptive field is the amount of pixels that gives information to a single output pixel. Often the

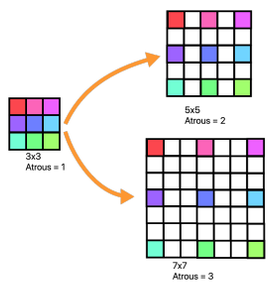


Figure 4.4: Example of Atrous filter made by expanding a 3x3 filter to 5x5 and 7x7 by adding zeros (no color) between original values (with colors).

receptive field is notated as $w \times h$ where w is the width and h is the height of the receptive field.

Example: A single 3×3 filter will have a receptive field of 3×3, while two 3×3 filters in a sequence will have a receptive field of 5×5. This means that each pixel in the output will have inputs from 5×5 pixels from the input before both filters.

The benefit of this compared to having just a single 5×5 filter is that with two filters we can start to find more complex representations then a single filter. If we add a activation function after each filter we will be able to find more advanced features with two small filter compared to one big. In addition to finding more advanced features, the amount of parameters for the two 3×3 filters is 18 while a single 5×5 filter uses 25. This can significantly reduce the computations needed for finding advanced features in a deep convolutional network.

4.4.1 Atrous convolution

Another way to increase the receptive field is to use *Atrous convolution* [8]. Then a filter is increased in size by adding zeros between each value in the filter as seen in Figure 4.4. This will result in a filter with larger size while having the original edge values spread to edge of the new filter. This can be beneficial for convolutional layers trying to find objects and classify them in an image. There are however some drawbacks to this method as it generates griding artifacts [56]. It exists methods to reduce this effect [56] but it is not all ways necessary in a feature representation of an image.

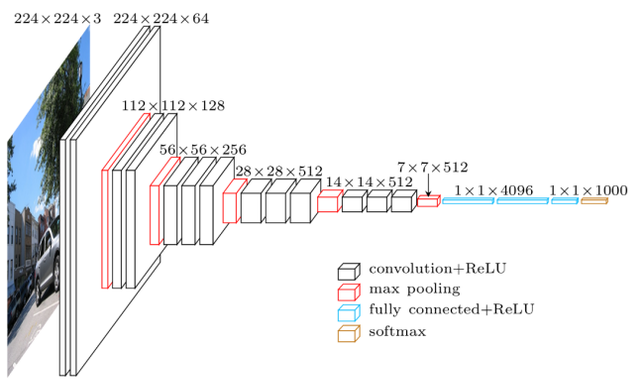


Figure 4.5: Architecture of VGG-16 network, the last output from the soft-max can be altered to fit the amount of classes that should be predicted. Image from towardsdatascience.com[1].

4.5 Convolutional Classification Models

In recent years convolutional networks have been the most used for working with images and grid data. They have proved to be well suited for the task and quite intuitive to work with. There are a wide variety of structures and applications and alternatives of each model. In this chapter we will look at two well used models for image classification and there main features.

4.5.1 VGG-networks

VGG is a well used deep convolutional network that works good on grid data [41]. There are several variations of VGG networks with different amount of layers where 16 and 19 layers are the most common. The structure is based on the same principles, in this chapter we will focus on the VGG-16 network architecture.

VGG networks uses sequences of convolutional layers followed by a max pooling layer. All convolutional filters have a size 3×3, stride 1 and padding to same size. The max pooling layer has a size 2×2 with stride 2. This makes the network easy to use since all blocks alter the the width and height of the input in a predictive way. The structure can be seen in Figure 4.5

As we discussed in the previous section there are advantages with using several

1. Link to blog post where image is taken from: https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c

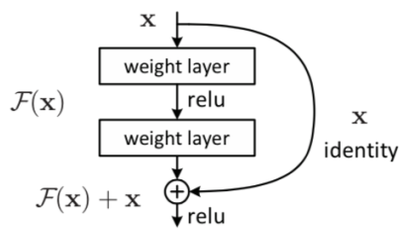


Figure 4.6: A building block in a ResNet with residual connection marked as identity. Image taken from [18].

small filters. We can find more complex non linear features compared to having one large filter. It is also more parameter efficient with several small filters compared to larger filters.

The last three layers in all VGG networks are fully connected layers. the output from these will be the same as the amount of classes that should be classified. Finally a soft max function is used for classification after the last layer.

4.5.2 ResNet

ResNet has been a very popular network for image processing since it was introduced in 2015 [18]. It addresses the problem that deeper CNNs do not necessarily produce better results when more layers are used. Deep CNNs (20+ layers) do not seem to give better results with more layers. In theory deeper networks will find higher level features as the input gets more and more processed. This however applies for the training of the network and it is not guaranteed that this will transform to the test phase. In reality deeper networks are harder to train due to problems like vanishing and exploding gradients. These can make it almost impossible to optimize the early layers in the network. This will make the training progress slow or even impossible when no improvement is made.

ResNet solves this problem by using residual connections in the network enabling it to use more then 100 layers efficiently. In Figure 4.6 we can see a block of 2 layers with a residual connection past them. This connection makes it possible for the derivative to pass trough the backpropagation without passing trough any activation function. Thereby it will effectively remove a big part of the vanishing/exploding gradient problem. ResNet also uses batch normalization between each convolution layer and activation function. This also helps smoothing out the parameter space [20].

Mathematically we can describe a block with a residual connection with Equation 4.6. Here $\mathcal{F}(\boldsymbol{x}, \{W_i\})$ is the block itself with convolutional layers. The $+W_s\boldsymbol{x}$ term is the residual connection that is passed on and unchanged by the operations inside the block. Here W_s is a linear projection that is used if the dimensions of $\boldsymbol{x}$ and $\boldsymbol{y}$ is not the same.

$$\boldsymbol{y} = \mathcal{F}(\boldsymbol{x}, \{W_i\}) + W_s\boldsymbol{x} \tag{4.6}$$

Due to its good properties ResNet is now very well known and used as a base for many other machine learning networks. If the ResNet is used for classification it will have a fully connected layer in the end to output the same size as the possible classes.

5

Semantic Image Segmentation

Semantic segmentation aims to classify each pixel in an image to a class corresponding to the object that pixel belongs to, the result can be seen in Figure 5.1. This is done to create a *segmentation map* that will represent each class with a specific label. The label is often shown as a specific color when presenting the segmentation map. If we are only interested in looking on one or a few classes we can use a "background class" for all the pixels that are not belonging to the classes of interest.

Image segmentation is useful for a variety of tasks, either to help a human in decision making or fully automate a task. For medical images it can be used on x-ray images to separate different organs or finding cancer [2]. In autonomous vehicles it is used for helping the car understand the environment and separate different objects [22]. Note that most methods do not separate between objects from the same class. This means that two persons standing next to each other will look like one big silhouette in the segmentation map.

A problem with training networks for semantic image segmentation is that producing labels on a pixel level is quite resource expensive. There exist some big open source datasets with corresponding segmentation maps available. But for more specific image types it can be hard getting good results without producing new training data.

Figure 5.1: Sample from the PASCAL Context 2010 dataset. Up: The sample image. Down: The segmentation map.

5.1 Encoder-Decoder approach to segmentation

To classify each pixel in an image there are several different approaches. One way is to use fully convolutional layers followed by an interpolation layer in the end to upscale the segmentation maps to the original size [31]. Another common approach is to use deconvolution and un-pooling layers instead of interpolation [36][34]. This can make it possible to do non linear up scaling that can be trained for specific tasks.

5.1.1 Encoder

The strength of a CNN is that it is good to represent the important features in an input image with just a few values. An *encoder* takes input and encodes(transforms) it from an input space to a feature space representation. Where the feature space representation is a lower dimensional representation of the data. This is normally impossible for humans to understand.

When we mention encoders in machine learning we often think of networks that encode an input to a feature space in a given size of an input image. This can be seen as compressing the information in the input image, while still trying to keep as much information as possible. The encoder can then be used for representing an input image with either one or several dimensions. This means that we can represent the content in an image with a fixed size data structure. These can then be compared to other encoded images or processed further, by for example creating new images.

There are a lot of similarities in the first part of a CNN for classifying images and an encoder. By using a trained classification network we can use the first part of the network to get a pretrained encoder. The amount of compression in the encoder can be chosen by selecting the how many layers to use from the classification network.

5.1.2 Decoder and deconvolution

While the encoder is good at compressing an input to a smaller size, a *decoder* is good at expanding compressed information to a larger size. The encoder can extracted a feature space representation of an image. Then we can use a decoder to try to generate a segmentation map of the original input with the original size. On images this is mostly done with deconvolutional layers. The result can be compared to linear interpolation but with a non linear up scaling. This means that the decoder network can use the structure of the neighboring pixels to fill in the blank space, compared to just filling them with a set transformation of the values at the closest pixels. Another crucial advantage is that the decoder do not need to have an input from the same amount of channels as the output. It can therefore be given an input straight from a complex feature space rather then just a smaller version of the output image.

5.2 Intersection over Union

When the network is trained it is important to be able to measure how well it is performing by comparing the predicted results to the ground truth. The loss function measures this in a way. Using the loss function to compare different networks performance is not suitable. The loss function often dependent on the size of the output and can be different for each network. To solve this we need a standardized measurement that can be used on one or many classes. It also needs to be normalized in the manner that it is independent of how big representation each class has. The Standard way of doing this for a segmentation map is the *intersection over union* (IoU). As the name implies, it finds the ratio between the intersection and union between the prediction and ground truth. The IoU is calculated with the following expression

$$IoU_c = \frac{Pred_c \cap GT_c}{Pred_c \cup GT_c}$$

where we calculate the IoU for class c. Here $Pred_c$ is the pixels predicted to

class c and GT_c is the pixels belonging to class c in the ground truth. It is also possible to calculate the mean IoU for all classes. Then it is easier to look at the IoU with regard to true positive (TP), false positive (FP) and false negative (FN) as the following expression

$$IoU = \frac{TP}{TP + FP + FN}.$$

/6

Few Shot Learning

While humans can easily learn to recognise new kinds of objects by just looking at one example, machines struggle with this [47]. This ability motivates the exploration of the *few shot learning* field of machine learning. In this field of machine learning the focus is to have a network that can learn to adapt with very limited training data. While training the network, the focus is on "learning how to learn" and not the learning of a specific task.

The goal is to have a network that can learn a new task with very limited amount of examples of the task. This however does not remove the need of a big dataset all together. We still need to train the network on a similar tasks (classification, image segmentation etc.) to have a good base for the network to learn quickly from.

The core difference is that in a few shot learning network we can add a new class with just a few examples [51]. We will not achieve as good results as having big labeled dataset of a new class and use supervised learning methods. With few shot learning methods we can get decent result with as little as one to five training examples of a new target class. For many problems the datasets does not exist and make the standard supervised learning methods unusable. Also the labor needed for creating a big dataset with labeled images or even labeled segmentation maps is expensive for a large dataset [37]. Some datasets requires medical experts to label them, an example is x-ray images while looking for certain diseases.

6.1 Terminology

When we talk about few shot learning the terminology *shot* and *way* is often used, ex: 5-shot 3-way. Here the amount of *shots* are referring to the amount of training data for each class and *ways* are the amount of classes. In this example there are only $5 * 3 = 15$ training samples, also called *support set*. The test set is called *query*, note that the query can be an single image in the test phase of a 1-shot network.

Few show learning can be seen as solving the following problem:
We are looking at one image, *query*, and want to classify it by comparing it to another set of images, *support set*. We can only choose one class from the support set and therefore need to learn what the query is most similar to in the support set.

6.2 Few-shot classification via prototypical learning

There are many different ways of solving a few-shot learning problem [26][27][33], in this section we will discuss one method that can be extended for use in few shot segmentation as well.

When training a few-shot network *episodes* are often used. An episode is a small set of training data that should represent how the network get data for the inference phase. So if a few-show network gets 5 support images to classify one query image the training data is divided into episodes of 6 images, 5 support and 1 query.

In few-shot networks it is common to use encoders as the first step in the network. The encoder can find a feature space representation for both the support and query image. To reduce the training time needed for the encoder using a pretrained network that is trained on a similar task but on different data is popular. Here a VGG-16 or ResNet are common examples for image data.

The pretrained network can be modified to an encoder by removing layers from the end of the network. The amount of layers removed will effect how narrow the output from the encoder will be. If only a few of the first layers the network will produce a low level feature representation of the input. This will represent the general traits of the input rather then specific details. Deeper in the network the receptive field increases and high level features are found. These are more

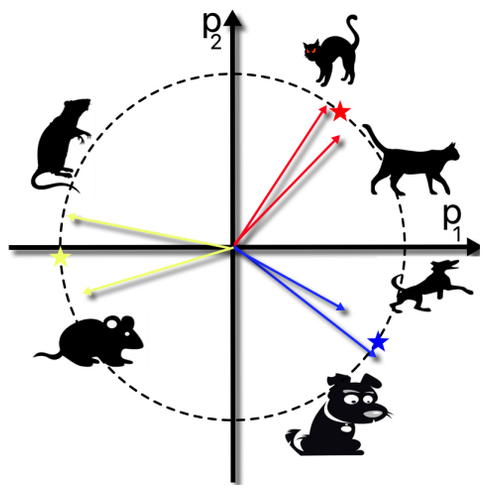


Figure 6.1: Visual example of prototypes from image inputs represented by arrows in two dimension, p_1 and p_2. The mean of normalized prototypes of each class is represented by a star.

complex shapes, for examples eyes in photos or specific animals. Since we are interested in being able to find new classes we want to use low level feature representation to have a more general view of the input [58].

This feature mapping can be used to compare different inputs by checking for similarities. By taking the global mean of each pixel in the encoded input, we will get a vector representing the feature space mapping of the input. This vector representation is called a *prototype* and can be used to compute similarity score between different prototypes. We will discuss in more detail the commonly used similarity score *cosine similarity* in the next section. The prototype can be calculated by

$$p_C = \frac{1}{W}\frac{1}{H}\sum_{w=1}^{W}\sum_{h=1}^{H} I_{C\times W\times H}$$

and will have the mean value of each filter used in the last layer. The main idea is that inputs that represent the same class will have similar prototypes. Therefore they will be mapped together in the prototype space as seen in the Figure 6.1.

Since each vector corresponds to a certain input image we can find a representation for each class by taking the mean of vectors from the same class. These mean vectors are then normalized to make sure they are comparable.

By just running our few training samples and taking the normalized mean for each class, we can compare them to each other. If we then want to classify a new input test image we can pass it through the network and see what class vector it is closest to. Since all vectors are normalized one can also use the *cosine similarity* to just compare the direction of the new test vector from the new image with the mean class vectors. In the next chapter cosine similarity is explained in more details.

This method does not need any training on the specific query set that we are interested in. We can make a pretrained model by using a network that has been trained on classification on a similar input dataset as the query dataset. The pretrained network can then be used as the encoder for the prototype network model described above.

Few-shot models are still dependent on big labeled datasets to generate a good variety of episodes for training. Lacking this can result in a network that has difficulties in adapting to new classes. A network might not be general enough to be able to find good prototype vectors if it gets images from a new domain [5].
Ex: A network trained on personal images from the Flickr dataset will most likely have trouble with finding good prototypes from medical images.

The pretrained model can be improved by using fine tuning with the support set that is available. The accuracy can be improved compared to not using fine tuning [9]. This will help if the pretrained model is trained on a dataset that has a different type of images then the support set.

6.3 Cosine similarity

Cosine similarity is a distance metric, used to measure similarity between two non-zero vectors of an inner product space. It can be derived from the Euclidean dot product formula:

$$\mathbf{A}^T \cdot \mathbf{B} = |\mathbf{A}| \cdot |\mathbf{B}| cos(\theta)$$

$$similarity = cos(\theta) = \frac{\mathbf{A}^T \cdot \mathbf{B}}{|\mathbf{A}| \cdot |\mathbf{B}|} \tag{6.1}$$

Finding the cosine of an angle, θ, between two vectors is the same as finding the normalized inner product of the vectors. This is a useful method to test how similar vectors in feature space are to each other. Looking at the angle instead of the distance between points tends to be more stable in higher dimensions.

7

Few Shot Segmentation

The few-shot classification approach can also be extended to solve few-shot segmentation problems. Here we want to make a segmentation map of a query image, while we just have a few support images with corresponding segmentation maps. For quite a lot of cases it can be interesting to just look at one class and compare it to the background. In that way we can filter out an interesting object from an image. For example this can be applied to find diseases in medical images [14]. The basic idea is to classify each pixel in the image to either one of the classes from the support set or as background. Few-shot segmentation will be the main topic of this thesis and in this chapter we will take a closer look how it is done.

7.1 Non-parametric vs Decoder-based Approach

There are two main directions for doing few-shot segmentation. The first *non-parametric direction* focuses on comparing the prototype with each pixel in the query image using a fixed distance function like the cosine similarity. This is a very general approach and have little tendency to overfit. Here we can have networks with very limited amount of trainable parameters. We can use a pretrained encoder to get a feature space representation of the input images. The encoder can be trained to make the network perform better but in theory it can be used it without any training. For upscaling the feature space sized prediction, a linear algorithm is often used for the predicted segmentation map.

This makes non-parametric networks easy to train and understand, here PANet [49] is a representative example for the first direction. This direction however has its limitations, since it is so general it can have trouble with performing on more complex classes. It is not adapting for each new target class and therefore not using all information from the support set.

The second direction is the *decoder-based direction* which are often larger networks with more parameters. Here the fixed upscaling functions are changed to a decoder which can learn non linear upscaling. This makes the decoder-based networks harder to train but also increases its potential. The distance function used to compare prototypes to feature space pixels is often changed to a parameter based module that can be trained. A typical example of the decoder-based direction is CANet [58]. Here the network is using the support set more for active training and adapts to a specific target class. Some parts of the network adapt iteratively according to the support set to give a more specialised output.

We will take a closer look at both examples, PANet and CANet. Then we will dig deeper into the second decoder-based direction. We are interested in this direction since it has bigger potential to extract more usable information from a new support set. It also uses decoders for the upscaling that can be specialised for a certain target class. We are interested in finding ways to improve the class specific parts in few-shot segmentation networks.

7.1.1 Non-parametric Direction, PANet

Based on a similar method as a prototypical classification network [42] PANet uses prototypes from the feature representation of the input to produce segmentation maps. PANet also introduce an extra step, prototype alignment regularization (PAR). Where the produced segmentation map is reused as the ground truth in a second pass of the network. An overview of the method can be seen in Figure 7.1.

We will have a support set with images and their corresponding true segmentation maps. Each image is pass through a convolutional network that works as an encoder. PANet uses the first five layers of a VGG-16 network pretrained on Image-net. The encoder has the same weights for all images independent whether they come from support or query. These are then masked with a one hot encoded mask for each class in the segmentation map. The mask is downsized to the same size as the feature space by max pooling to keep the one hot encoded structure of the mask. After the feature representation has been masked by class, they are globally average pooled. The result is a vector for each class, a *prototype vector* that represent the average pixel value for that

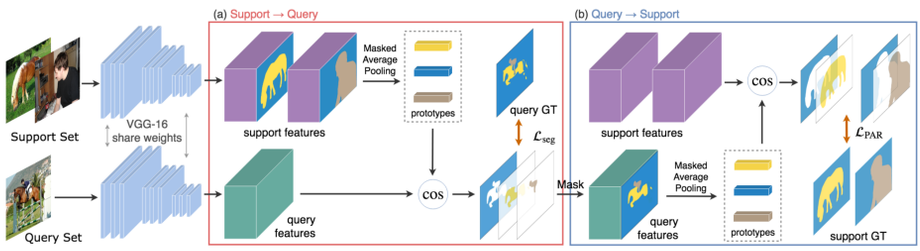


Figure 7.1: An overview of a 1-shot PANet example. The input from support and query are both mapped to the same feature space(left side). Then block (a) performs a support-to-query few-shot segmentation. Where the prototypes are calculated by masked average pooling and the query image is segmented by cosine similarity (cos in figure). The loss $\mathcal{L}_{seg}$ is calculated between between the segmentation result and the ground truth mask. Then the process is repeated in block (b) with swapped rolls for support and query. The generated segmentation result is then used as ground truth mask when the loss $\mathcal{L}_{par}$ is calculated. Image from [49].

class after the encoder.

Then the query image is also passed through the same encoder, the feature space representation can then be compared pixel wise with the prototypes for each class. The cosine similarity is calculated between each pixel and the set of prototype vectors. Each pixel will be mapped to represent the most similar class. Then the segmentation map will be up scaled with linear interpolation to match in input size. This is the output from the network while it is in the test phase. The test phase is described in left side of Figure 7.1, with the VGG-16 encoder and $(a)Support \rightarrow Query$.

The second part of the network $(b)Query \rightarrow Support$ is only used in the training phase. Here the procedure is repeated with the tasks for the query and support swapped. The feature space representation of the support and query is reused to calculate new prototypes. The segmentation map generated from part (a) is used as the ground truth to mask the query image to calculate the new prototypes. Then the same procedure as in (a) is used. Cosine similarity between each pixel from the encoded support images to find segmentation maps. These segmentation maps can then be compared to the original segmentation maps for the support set.

Then the loss is calculated by adding the loss from part (a) and (b). First the segmentation loss from part (a) is calculated by

$$\mathcal{L}_{seg} = -\frac{1}{N}\sum_{x,y}\sum_{p_j \in P} \mathbb{1}[M_q^{(x,y)} = j] log(\tilde{M}_{q,j}^{(x,y)}).$$

Here x and y are the indexes for the spatial locations, N is the total number of spatial locations, $x \times y$. P contains the C different classes $\{p_1, ..., p_{C-1}, p_{bg}\}$ including the background class p_{bg}. $\mathbb{1}[\bullet = \bullet]$ is the indicator function[1]. $M^{(x,y)_q}$ is the ground truth segmentation map and its estimate $\tilde{M}_{q,j}^{(x,y)}$ is a probability map calculated by

$$\tilde{M}_{q,j}^{(x,y)} = \frac{exp(-\alpha d_{css}(F_q^{(x,y)}, p_j))}{\sum_{p_j \in P} exp(-\alpha d_{css}(F_q^{(x,y)}, p_j))}.$$

Here α is a scaling parameter and d_{css} is the cosine similarity distance metric function. F_q denotes the query feature map and p_j is the prototype from class j.

Then the loss for part (b) is calculated by:

$$\mathcal{L}_{PAR} = -\frac{1}{CKN}\sum_{c,k,x,y}\sum_{p_j \in P} \mathbb{1}[M_q^{(x,y)} = j] log(\tilde{M}_{q,j}^{(x,y)})$$

Where $k = 1, ..., K$ are the support images.

Then the total loss is calculated as a sum of both losses with $\mathcal{L}_{PAR}$ scaled by parameter λ.

$$\mathcal{L} = \mathcal{L}_{seg} + \lambda \mathcal{L}_{PAR}$$

While training the network the parameters in the encoder are updated. These are the only trainable parameters in the network. The rest of the network does not have trainable parameters and only uses fixed functions. The loss function is calculated in two parts, one from the first part of the network and one from the second. The first part the loss is calculated between the generated query segmentation map and the query ground truth segmentation map. The second part of the network the loss is calculated between the generated support segmentation map and the support ground truth segmentation map. Both the query and support losses are simply added together to generate the complete loss that can be backpropagated to improve the network.

1. A function that is either 0 or 1, can be seen if test where True is 1 and False is 0.

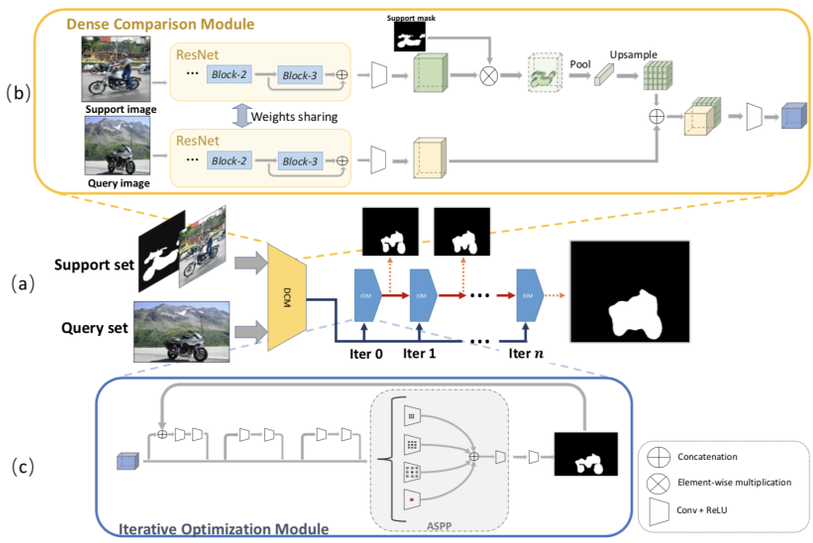


Figure 7.2: An overview of 1-shot CANet for semantic segmentation. (a) The network structure. (b) The Dense Comparison Module. (c) The Iterative Optimization Module. Image from [58].

7.1.2 Decoder-based Direction, CANet

The second direction of few-shot segmentation is well represented by the CANet network. It iterativly improves the the query segmentation map by a step by step improvement guided by the support set. This decoder-based approach have the ability to improve the results over time.

CANet is divided into two parts, a dense comparison module (DCM) and an interative optimization module (IOM). The DCM extracts the information in the support set and the query image and the IOM generates segmentation maps for the query iterative with improving results. An overview of the method can be seen in Figure 7.2

In the DCM the encoder uses the first three blocks from a ResNet pretrained on ImageNet to map the images to feature space. The weights are shared for the encoder for both query and support set. The feature space representation of the support set is then point wise multiplied with the corresponding segmentation map. This is then global average pooled to a vector that is expanded to the same size as the feature space. The result is a representation of the support set that has the same prototype vector for each pixel location. Finally the expanded prototype tensor is concatenated with the encoded query image. This is passed through a convolutional layer with a ReLu activation function. The result is the output from the DCM.

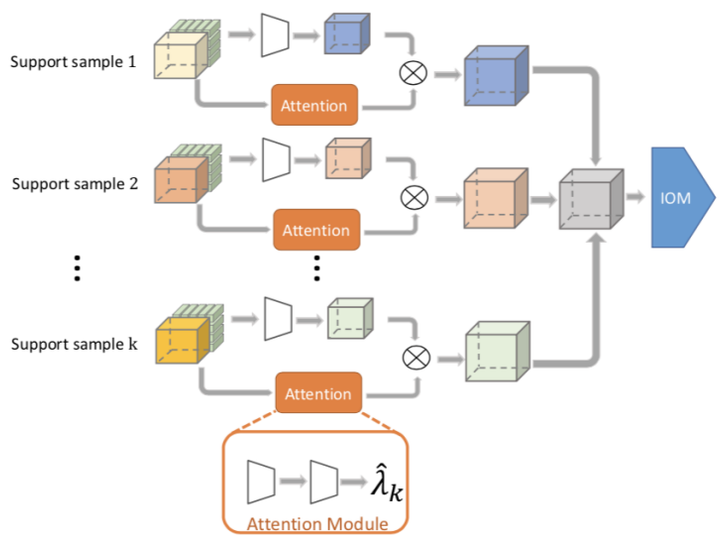


Figure 7.3: Attention mechanism for k-shot semantic segmentation. The softmax function is used on the scalar output to normalize the it so that all scalars from support examples sum to 1. Image from [58].

If there is more then one support image, the amount of information used for each image has to be decided. Here attention is calculated for all the images and used as a scaling for the the output of the DCM. In Figure 7.3 we can see how attention is used to scale the contribution from each support image output. The attention part consists of 2 convolutional layers, the last one only has 1 filter and is followed by global average pooling. This results in a scalar that is used in a softmax function to get a weight for each support image. Then the output from each support image is added together to make one output for the DCM.

In the IOM part of the network convolution layers with skip connections are used to further process the output from the DCM. After 3 blocks with 2 layers in each, the input are processed with an Atrous Spatial Pyramid Pooling (ASPP) [8] layer. Here different 3x3 filters are up scaled to a bigger size while leaving just 0 on the additional spots in the filter, as seen in Figure 7.4. This helps with finding bigger patterns in the image by increasing the receptive field while not increasing the amount of parameters compared to a 3x3 filter. The size and spread of the values in the ASPP vary for different filters.

Finally after the ASPP the IOM uses a few convolutional filters to generate the output segmentation map.

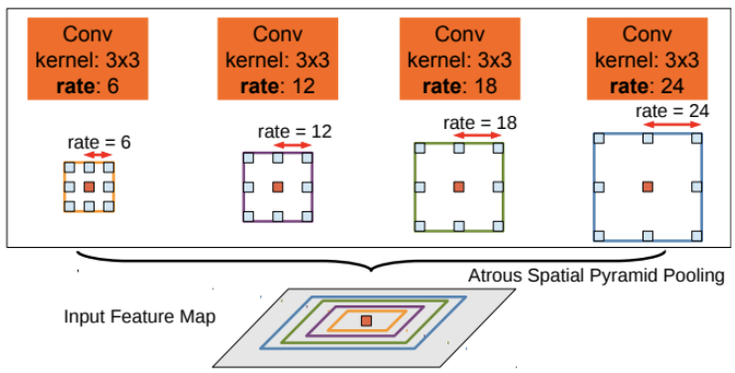


Figure 7.4: Atrous Spatial Pyramid Pooling (ASPP). It is classifying the center pixel (orange) by exploiting multi-scale features using multiple parallel filters with different rates. The receptive field are shown in different colors on the feature map. The figure is taken from [8].

This IOM part of the network is then iterativly repeated where the segmentation map is concatenated before the first convolutional layer. This is to make the network improve on the first prediction. To prevent it to overfit and rely on the previous segmentation map a form of dropout is used so that some of the returned segmentation maps just 0 as value.

7.1.3 Decoder-based Direction, PFENet

The Prior Guided Feature Enrichment Network (PFENet) [46] also uses the decoder-based method to segment images. It focus on using prior segmentation maps, Y_Q, to guide the decisions for generating the query segmentation map. An overview of the PFEnet can be seen in Figure 7.5.

The motivation to use prior segmentation maps is that they can allow to use deeper encoders that generate higher level features. These are according to the authors [46] more class specific and are therefore bad at generalizing to other unseen classes after training. The prior segmentation map, Y_Q, is therefore made from an encoder that has not been fine-tuned by training the few shot learning network. By having the encoder fixed it will not be biased to any of the classes in the training set.

To calculate Y_Q a encoder, $\mathcal{F}$, is used. It is made from a pretrained ResNet with the first 4 layers for extracting high level features. The encoder is used on the support and query input images, $\mathcal{I}_S$ and $\mathcal{I}_Q$ to generate the feature representations X'_S, X_Q. Then the support segmentation map, M_S, is rescaled and pointwise multiplied on X'_S to get X_S. We have

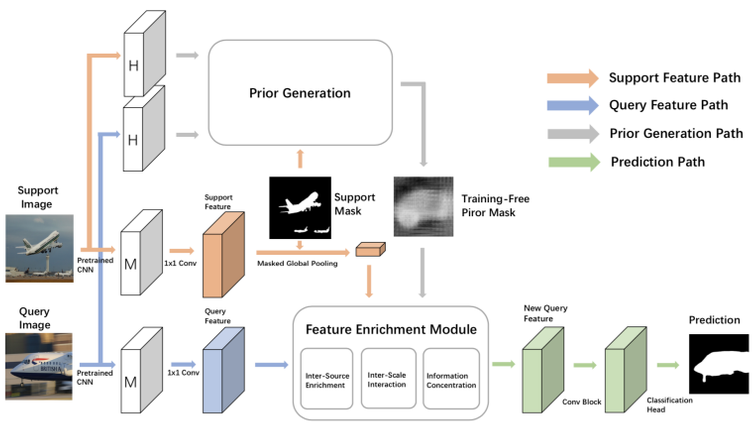


Figure 7.5: Overview of the Prior Guided Feature Enrichment Network, with the prior generation and Feature Enrichment Module. White blocks marked with M and H represent the middle- and high-level features from the backbone. Image taken from [46].

$$X_Q = \mathcal{F}(\mathcal{I}_Q)$$
$$X_S = \mathcal{F}(\mathcal{I}_S) \bullet M_S$$

where $\bullet$ represents the pointwise multiplication. Since we have masked X_S so that the background is zero it is removed from the feature representation of the support image.

Then the maximum of the cosine similarity between each pixel in X_Q and all pixels in X_S is used in the corresponding pixel in Y'_Q

$$Y'_{Q[s,t]} = \frac{max}{h,w \in \{1,2,...,H,1,2,...,W\}}(cos(X_{Q[s,t]}, X_{S[h,w]})),$$

here $cos(*,*)$ represent the cosine similarity 6.1.

Finally Y_Q is computed by normalizing Y'_Q with min max normalization

$$Y_Q = \frac{Y_Q - max(Y_Q)}{max(Y_Q) - min(Y_Q) + \epsilon},$$

where ϵ is set to $1e-7$.

Y_Q will then be used as an input in the *Feature Enrichment Module* (FEM) to produce the generated segmentation map for the query image. An overview of the module can be seen in Figure 7.6.

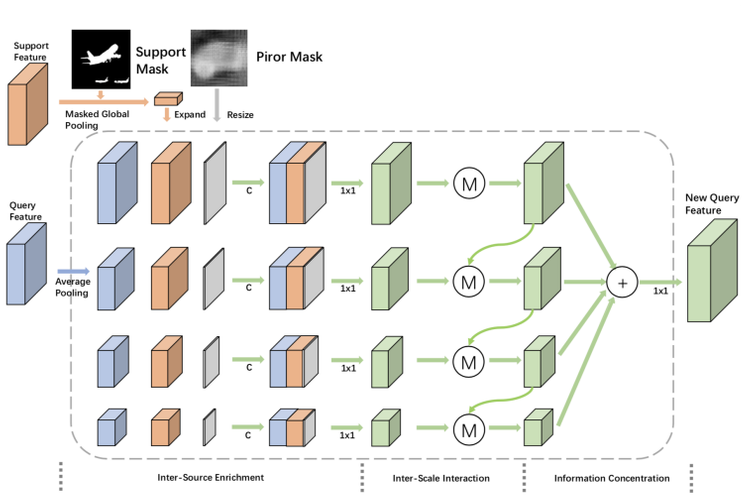


Figure 7.6: Example of the FEM module (dashed box) with four scales and a top-down path. C represent concatenation, 1x1 is a convolution and Circled M represent inter-scale merging module. All activation functions are ReLu. Image taken from [46].

There are two other inputs for the FEM produced from the query and support image. Both are encoded with a different encoder from the one used for the prior mask. The new encoder is produced in the same way but with fewer layers to produce mid-level features for better generalization.

The encoded support image is then masked with the support mask and GAP is the used to produce a prototype as we have seen in CANet and PANet. For the encoded query image a 1×1 convolution is used before it enters the FEM.

In the FEM there are three different stages, *Inter-Source Enrichment*, *Inter-Scale Interaction* and *Information Concentration*. The first, Inter-Source Enrichment, scales in the inputs to a set of n spatial sizes, $B = [B^1, B^2, ..., B^n]$, where $B^1 > B^2 > ... > B^n$. The three different inputs are scaled to each of the n sizes in B. The support prototype is simply expanded to the correct size. Adaptive average pooling is used on the query features and the prior mask is resized to fit each size by bi-linear interpolation. When we have n different scales of each of the three inputs they are concatenated to n different blocks and sent to the next stage in the FEM.

In the Inter-Scale Interaction stage of the FEM a top down approach is used for combining information from each of the n different scales. Here information is passed from finer features to the more coarse ones as the inputs are more and more compressed. This is shown in Figure 7.6 as the circled M with how the

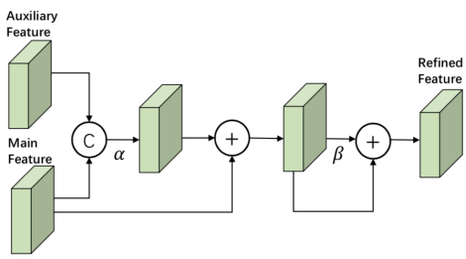


Figure 7.7: Visual illustration of the inter-scale merging module M. C is concatenation and + is pixel-wise addition. α means 1×1 convolution and β represents two 3×3 convolutions. If a features do not have auxiliary features, no concatenation with the auxiliary feature is made and the refined feature is produced only by the main feature with α and β. All activation functions are ReLU. Image taken from [46].

information is passed down. The module can be seen as

$$X^i_{Q,new} = \mathcal{M}(X^{Main,i}_{Q,m}, X^{Aux,i}_{Q,m}),$$

where $X^i_{Q,new}$ is the refined feature for the scale i, B^i. $X^{Main,i}_{Q,m}$ and $X^{Aux,i}_{Q,m}$ is the main and auxiliary feature inputs for B^i.

We take a closer look at this module in Figure 7.7 where the two inputs Auxiliary Feature comes from the previous layer and the Main Feature comes from the current layer. In the module both inputs are concatenated and put through a 1×1 convolution, α. Then the Main Feature input are pointwise added to the result. This is followed by another pointwise addition between the previous result and a 3×3 convolution, β, of it to produce the refined feature map, $X^i_{Q,new}$. Note that for the first layer the auxiliary feature input does not exist, this is then replaced with an empty tensor of the same size.

In the final step, Information Concentration, all $X^i_{Q,new}$ are reshaped to the output size of $h \times w \times c$ and concatenated with each other. Then a final 1×1 convolutional filter is used to produce the output query map, $X_{Q,new}$. The Information Concentration step can be seen as

$$X_{Q,new} = \mathcal{F}(X^1_{Q,new} \oplus X^2_{Q,new} \ldots \oplus X^n_{Q,new}).$$

The loss functions is calculated with cross entropy loss, this is done for each of the refined feature maps, n $X^i_{Q,new}$, in the FEM module to get $\mathcal{L}^i_1$. Then again for the final prediction, $\mathcal{L}_2$. The total loss is computed by calculating the mean of the n $\mathcal{L}^i_1$ and and adding the last loss,

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}^i_1 + \mathcal{L}_2.$$

7.2 Self-guided and Cross-guided network

The self-guided and cross-guided network (SGCG) [57] addresses the problem that some information is unavoidably lost from the masked *Global Average Pooling* (GAP) when obtaining the prototypes. SGCG introduces two prototypes; one primary and one auxiliary that represent the networks correct and incorrect segmentation map prediction on the support set. These help guiding the prediction of the query image. An overview of the whole model can be seen in Figure 7.8.

The SGCG network is based on prototypes like PANet. SGCG uses the same encoder for both support and query images as before. It also uses decoders later in the network that also share weights for support and query. The problem of lost information from the prototypes is solved by a *Self Guided Module* (SGM). The SGM will generate a loss for the support image and two prototypes for the true and false predicted support mask. The two prototypes are then used with the encoded query image to produce a predicted segmentation mask for the query image.

This method can be applied to different existing networks by using them as a backbone. The backbone is used in the query and support *Feature Processing Module* (FPM) and decoder as we calculate the segmentation maps in different parts of the SGCG network.

Self-Guided Module

Here we will take a closer look at the Self-Guided Module (SGM), an overview can be seen in Figure 7.9.

To get the inputs to the the SGM a masked GAP is used on the encoded support image, F_s, to produce a prototype,

$$v_s = \frac{\sum_{i=1}^{hw} F_s(i)\mathbb{1}[M_s(i) = 1]}{\sum_{i=1}^{hw} \mathbb{1}[M_s(i) = 1]}.$$

Here i is the spatial position and hw is the height and width of the encoded image. F_S is the encoded support image and $\mathbb{1}[\bullet]$ is the indicator function. M_s is the binary segmentation map for the support image downsampled to the same size as F_s.

Then the SGM uses the encoded support image, F_s, the prototype from F_s, v_s,

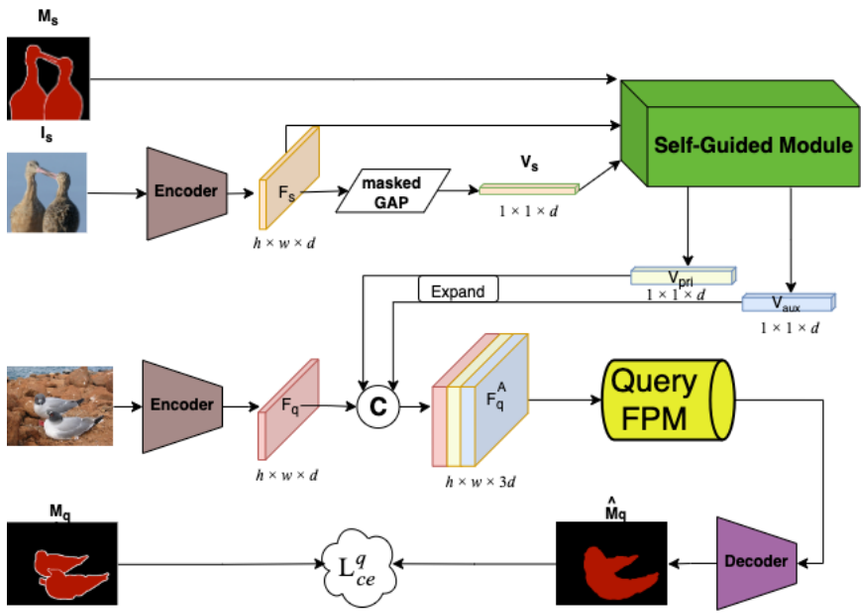


Figure 7.8: Overview of Self-Guided Cross-Guided (SGCG) network. It uses the encoded support image, F_s to generate a prototype, v_s. These are used by the Self-guided Module to produce two new prototypes, v_{pri} and v_{aux}. These are used with the encoded query image in the Feature Processing Module (FPM) to generate the predicted segmentation mask, $\hat{M}_s$.

and the mask to the support image, M_s as inputs. First v_s is expanded to the size of F_s, $h \times w \times d$, and becomes V_s. It is then duplicated to size $h \times w \times 2d$ and concatenated with F_s to become

$$F_{sv} = Concat([F_s, V_s, V_s]).$$

Then F_{sv} is passed through the support *Feature Processing module* (FPM). This module changes depending on the backbone of the network. If the decoder in the backbone are taking a single feature map as input [58][54] the Single-Scale FPM is used. Then if the decoder for the backbone needs several feature map inputs [46] the Multi-Scale FPM will be used.

The decoder then produces the predicted segmentation map for the support image, $\hat{M}_s$. The decoder consists of two blocks of two convolutional layers in each, the blocks are followed by an ASPP layer and lastly a convolutional layer. In each block the first layer in the is a convectional layer followed by a convolutional layer with the same output size as input. The output from

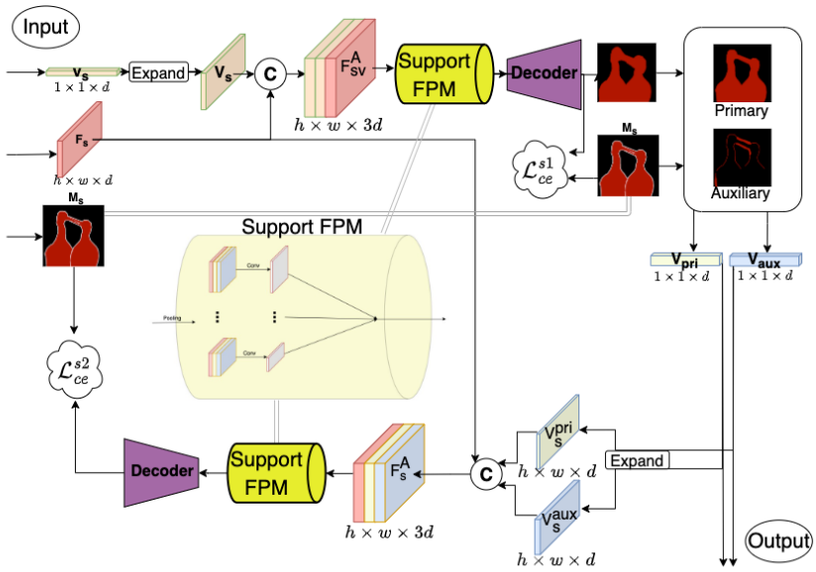


Figure 7.9: Detailed overview of the Self-Guided Module, (SGM). It uses the encoded feature map, F_s, from the support image and its masked GAP prototype, v_s as inputs. The SMG outputs two new prototypes, v_{pri} and v_{aux}. It also calculate two losses, $\mathcal{L}_{ce}^{s1}$ and $\mathcal{L}_{ce}^{s2}$. The first is from the ground truth segmentation map, M_s and the predicted segmentation map by using v_s, $\hat{M}_s$. The second is from M_s and the predicted segmentation map by using v_{pri} and v_{aux}.

each layer is then element-wise added together as the output for one block. A soft-max function is then used on the decoded image to get a predicted probability mask

$$P_{s1} = softmax(\mathcal{D}(FPM_s(F_{sv}))).$$

To get a binary predicted segmentation mask we will use the argmax function on P_{s1} to get

$$\hat{M}_s = argmax(P_{s1}).$$

This prediction is used to calculate a first cross-entropy loss with

$$\mathcal{L}_{ce}^{s1} = -\frac{1}{hw}\sum_{i=1}^{hw}\sum_{c_j\in[0,1]} \mathbb{1}[M_s(i) = c_j] log(P_{c_j}^{s1}(i)) \tag{7.1}$$

Here 1 is the target class and 0 is the background class. $P_{ce}^{s1}(i)$ is the prediction of pixel i belonging to class c_j.

The network will then compare $\hat{M}_s$ and M_s to find two new binary masks. One for where the true predicted pixels for class $c_j = 1$, called the Main mask in Figure 7.9.

$$M_{Main} = \mathbb{1}[M_S = \hat{M}_s]\mathbb{1}[M_S = 1].$$

Then one for where the predicted pixels for class $c_j = 1$ where false, called the Loss mask

$$M_{Loss} = \mathbb{1}[M_S \neq \hat{M}_s]\mathbb{1}[M_S = 1].$$

These masks are used for masked GAP of F_s to calculate two prototypes. One primary v_{pri} from the Main mask by

$$v_{pri} = \frac{\sum_{i=1}^{hw} F_s(i)\mathbb{1}[M_{Main}(i) = 1]}{\sum_{i=1}^{hw}\mathbb{1}[M_{Main}(i) = 1]}$$

and one auxiliary v_{aux} from the Loss mask

$$v_{aux} = \frac{\sum_{i=1}^{hw} F_s(i)\mathbb{1}[M_{Loss}(i) = 1]}{\sum_{i=1}^{hw}\mathbb{1}[M_{Loss}(i) = 1]}.$$

When we have computed the new prototypes, we want to use them by using them to generate a new segmentation map for the support image. This is done by repeating the previous process, stacking the prototypes to size $h \times w \times d$ and concatenate them by

$$F_s^A = Concat([F_s, V_s^{pri}, V_s^{aux}]).$$

Then the same Feature Processing Module is used with a decoder to produce a second prediction probability map,

$$P_{s2} = softmax(\mathcal{D}(FPM_s(F_{sv}^A))).$$

That is used to calculate a second cross-entropy loss for the support image,

$$\mathcal{L}_{ce}^{s2} = -\frac{1}{hw}\sum_{i=1}^{hw}\sum_{c_j \in [0,1]} \mathbb{1}[M_s(i) = c_j] log(P_{s2}^{c_j}(i)) \tag{7.2}$$

Using the query

In the last part of the network we will use the query image to generate a segmentation much like we did in the first part. We encode the query image to generate the feature representation, F_q. Then the two prototypes, v_{pir} and v_{aux} are stacked and concatenated with F_q

$$F_q^A = Concat([F_q, V_q^{pri}, V_q^{aux}]).$$

Then F_q^A is put through the query FPM, which has the same structure as the support FPM but different filters. The output is passed through the same decoder as previously to generate the predicted probability mask, P_q, and predicted binary segmentation mask, $\hat{M}_q$, for the query image.

$$P_q = softmax(\mathbb{D}(FPM_q(F_q^A))).$$

$$\hat{M}_q = argmax(P_q). \tag{7.3}$$

During training we can use M_q to calculate the stochastic-gradient loss for the query

$$\mathcal{L}_{ce}^{q} = -\frac{1}{hw}\sum_{i=1}^{hw}\sum_{c_j \in [0,1]} \mathbb{1}[M_q(i) = c_j] log(P_q^{c_j}(i)). \tag{7.4}$$

With that we can also add all losses in the network to a total loss

$$\mathcal{L} = \mathcal{L}_{ce}^{s1} + \mathcal{L}_{ce}^{s2} + \mathcal{L}_{ce}^{q}.$$

Here $\mathcal{L}_{ce}^{s1}$ and $\mathcal{L}_{ce}^{s2}$ are the loss from Equation 7.1 and 7.2.

Cross-guided for k-shot

To change from 1-shot to k-shot[2] we do not need to retrain the network but we will do some modifications. The Cross-Guided Module (CGM) is based on calculating the predicted probability mask for each support image and finding a weights sum of that. Here each weight represent how good the model thinks that corresponding predicted probability mask is. Then a softmax function of the sum and the argmax function generates the final predicted segmentation map for the query image.

First we have to change some of the notations from the 1-shot case. We will have a set of k support images, $\{I_s^1, I_s^2, ..., I_s^K\}$. With corresponding ground truth segmentation maps, $\{M_s^1, M_s^2, ..., M_s^K\}$. The predicted probability mask P^k can be calculated by our 1-shot model for query P_q^k and support P_s^k.

$$\mathcal{G}(I_q|I_s) = P_k. \tag{7.5}$$

We can utilize that we have k support image to find the best $\hat{M}_s^k$ for each of the k I_s. This is done by using one by one I_s as query, I_s^q, and calculating a $\hat{M}_s^{q|k}$ for each of the k-1 I_s not in use as query. Then the mean IoU between each $\hat{M}_s^{q|k}$ and M_s^q is calculated

$$U_s^k = \frac{1}{K}\sum_{i=1}^{K} IoU(\hat{M}_s^{i|k}|M_s^i) \tag{7.6}$$

and used as the weight for that I_s^q.

When this is done for all k I_s we can calculate the predicted probability mask, P_q^k, for the true query. These are used to calculate the predicted probability mask for the query by

2. k-shot indicates that k>1 so that there are several support images.

$$\hat{P}_q = softmax(\frac{1}{K}\sum_{k=1}^{K} U_s^k \mathcal{G}(I_q|I_s^k)). \tag{7.7}$$

To get the final predicted segmentation map for the query we use the argmax function

$$\hat{M}_q = argmax(P_q)$$

as in the 1-shot case.

7.3 Transductive Inference

As models for Few-shot learning have improved and become more complex, the results have somewhat stagnated [7]. One approach to overcome this is to look into the inference phase. The models which we explored in this thesis focused on meta learning where they focus on the "learning to learn" concept. This focus can have the effect of an *inductive* approach to the problem. Meaning that instead of solving the segmentation problem of the current target class the networks tries to find a solution for segmenting all possible classes. Getting a general solution is great if possible but for few-shot learning the results are limited by the small support set.

To solve this, a more *transductive* approach can be used in the inference phase of few-shot learning networks. This means that the network will only try to solve the problem for the current target class. Removing the need for being a general network, the results on the target class could improve. During training we do not want to change the inductive approach. We want to have a general network capable of handling new unseen classes. During the inference phase however, the network can make small changes to the weights to segment the new target class better.

These changes aims to tweak the parameters in certain components of the network by fine-tuning them. Fine-tuning a network network is closely related to *transfer learning*, where we focus on transferring knowledge from one network to another. This can be done by using a pretrained network as a base to retrain it for a new dataset. Fine-tuning an existing network can improve the training time significantly compared to training a network from scratch, especially if the new dataset is similar to the one used before.

The limitations for fine-tuning is that we need the network we are interesting in using to be trained all ready. This limits the use significantly, however we can use parts of a pretrained network as a part of a new network. We have seen examples of this in the encoders in the few-shot segmentation networks described previously.

The fine-tuning for transductive inference is done by using the support set as a training set while freezing all parameters outside the ones in the certain component. One example is the Region Proportion Regularized Inference network (RePRI) [5]. This network is fine-tuning the last classification layer of a non few-shot segmentation network [62]. By doing this a network that is not meant for few-shot segmentation tasks can get good performance in a few-shot segmentation setting.

Using fine-tuning in the inference phase for few-shot segmentation tasks has previously been limited to general segmentation networks, such as [5]. We hypothesize that the improvements can also be obtained for decoder-based few-shot segmentation networks. This would switch the inductive approach of a few-shot segmentation network to a transductive approach, making it more target class specific and potentially improve the results.

In Part II of this thesis we will look in detail on our method that can apply a transductive inference phase to an inductive few-shot segmentation network.

Part II

Method and Results

8 Method

In this chapter we will discuss our Inference Guided Few-Shot Segmentation method (IGFSS). In particular in the context with the Self-Guided Cross-Guided network (SGCG) [57] as backbone. However, note that this approach can be generalized to other prototype or decoder based networks as well. Our network is inspired by the Transductive Inference–RePRI [62] that fine-tune the last classification layer in the inference phase. Our IGFSS network will instead focus on improving the inference phase for decoder-based Few-Shot segmentation networks.

We will first go through the underlying idea of our IGFSS method. Then we will take a closer look on how it is applied to the SGCG backbone network. Starting with how the prototypes can be improved by our IGFSS method, followed by how it is applied to the decoder.

8.1 Inference Guided Few-Shot Segmentation

Our Inference Guided Few-Shot Segmentation method is designed to improve the inference phase for few-shot segmentation networks. We aim to make the inference phase more class specific to improve the results by utilizing more information from the support set. This is done by optimizing the parameters in certain class specific parts of the network. The changed parts will be optimized for the current target class. The rest of the network is left unchanged.

The SGCG backbone network is a meta learning few-shot segmentation network that focus on "learning to learn" training. This makes the network capable of segmenting new target classes with just a few support images. However, the inference phase is limited to the same segmentation approach as in the training phase. With our new Inference Guided method applied the SGCG network should be able to adapt to new complex classes better.

All the information that the network gets to segment a new target class query image is contained in the prototypes from the support images. This crucial class specific part is therefore a good candidates to optimize with our IGFSS method. The encoder and decoder also has an important role in interpreting the input image to feature space and back. The decoder is the last step for generating the predicted segmentation map and have a big impact on the final results. When the SGCG network is trained on the Pascal dataset it is trained without the target class and therefor not optimized for that. This indicates that there is room for improvement in this part of the SGCG network.

Since the encoder has its backbone from a pretrained ResNet trained on a very large and variable dataset it is assumed that it is quite general. It should be robust enough to handle new classes which the SGCG has not been trained on directly. It is not unlikely that the pretrained ResNet in the encoder have had a class similar to a new target class during its training phase. Therefore we have not explored the possibility for fine-tuning this part of the network, but rather focused on the other two alternatives, the decoder and prototypes.

8.1.1 Inference Guided Prototypes

The goal is to improve the output by fine-tuning the primary and auxiliary prototype on the support image before it is used on the query image. The prototypes are the only information about the target class passed on to the query part of the network. It is therefore crucial that it contains as much high quality information about the target class as possible. Inference guiding the prototypes will increase computing expenses for the network. But since this will only be done after the training of the network it will not affect the time consuming training phase.

When our IGFSS method is applied the support images are passed through the whole network to obtain the primary and auxiliary prototype. The fine-tuning will then be done by freezing all parameters in the network except the primary and auxiliary prototype. Then the query part of the network is used with the support image as query image during the fine-tuning phase. The setup used can be seen in Figure 8.1 and compared to the full version in Figure 7.8. During the fine-tuning process only using a small part of the network is used

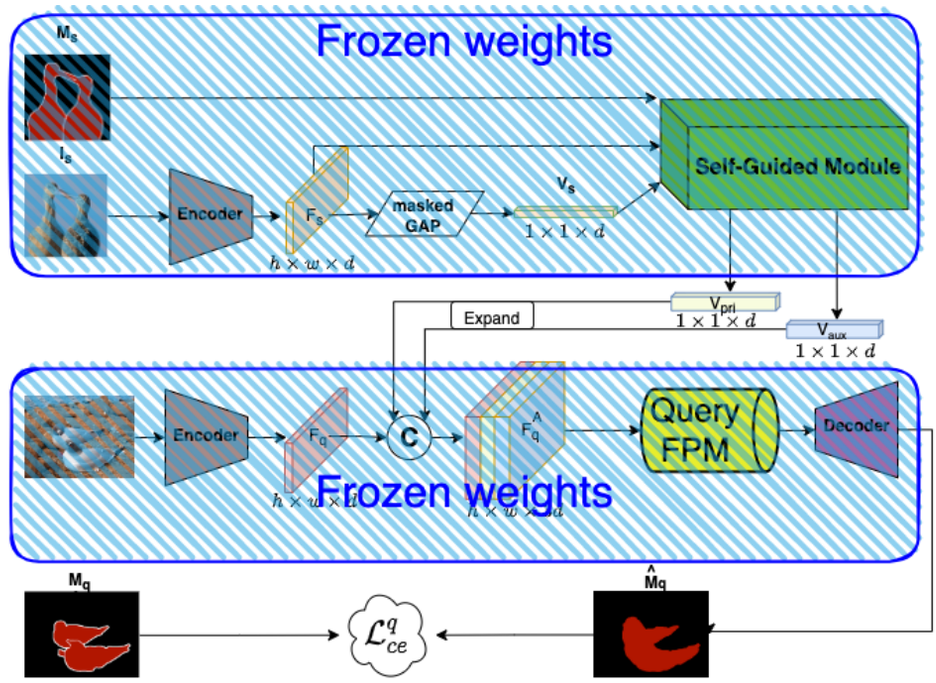


Figure 8.1: Structure of our IGFSS method applied to the Self-Guided Cross-Guided network [57] to fine-tune its prototypes. With the weights frozen in blue.

so it is a reasonable fast process, especially compared to retraining the whole network.

To update the weights, a single cross entropy loss is calculated like before between the predicted output, $\tilde{M}_s$, and the true support mask M_s. This is backpropagated through the frozen network and only updates the primary and auxiliary prototypes.

The method above describes well the **1-shot** case where we have a few hyper-parameters that we can change to affect the end result. Here the number of epochs for fine-tuning combined with the step size will be important to give a new result compared to the method without fine-tuning. Too many epochs will make the network overfit on the single support image and give bad results on the true query image.

For the **5-shot** case we will have more options since we get a tiny dataset for fine-tuning. Here two main decisions have to be made. A visualization of the combinations can be seen in Table 8.1 and will be refereed to in the next paragraph.

A B C 1 2	1 set of prototypes	5 sets of prototypes
Use mean	1 A	2 A
Use fixed IoU as scale	1 B	2 B
Use updated IoU as scale	1 C	2 C

Table 8.1: Overview of options for 5-shot fine-tuning.

First we choose the number of primary and auxiliary prototype to be used. Either a single set of primary and auxiliary prototypes for all support images *(1)* or a separate set for each of the 5 support images *(2)*. If separate sets are used the query output is obtained by first calculating a query prediction for each prototype set and then taking a mean of all those predictions. Secondly we choose how much the loss from each support image should affect the updating of the prototypes. The simplest and most computational efficient way is to treat all support images equal by using a standard mean of the losses *(A)*. Taking inspiration from [5] as described in Section 7.3 we can use a weighted average of the different losses. The IoU calculated between the support image prediction and its true label can also be used as weight. The IoU for each support image can then decide how much the loss for that support image should be scaled. This will make the loss from a well predicted support image have more effect on the fine-tuning. The support IoU for scaling the losses can either be calculated once in the beginning *(B)* or updated for each fine-tuning epoch *(C)*.

To choose what method that should be used for the inference guided prototypes we need to consider the dataset we want to segmentate. On a general basis one set of prototypes should give a more stable training as it is training in a batch setting. This will give a more robust network that can handle outliers better. The quality of the support images are uncertain so it can be wise to use a safe method, with the mean of all losses. Using IoU as weights can be beneficial for some classes, but not necessary for all classes.

8.1.2 Inference Guided Decoder

Another part of the network that can be fine-tuned to change the output is the decoder. This is the last part of the network and will have a big impact on the final segmentation map. By using our IGFSS method on the decoder we hope to make it more class specific. In that way an improvement on trickier target classes can hopefully be achieved.

The decoder can be fine-tuned in a similar manner as the prototypes as discussed previously. Here we will also freeze most of the network and only use

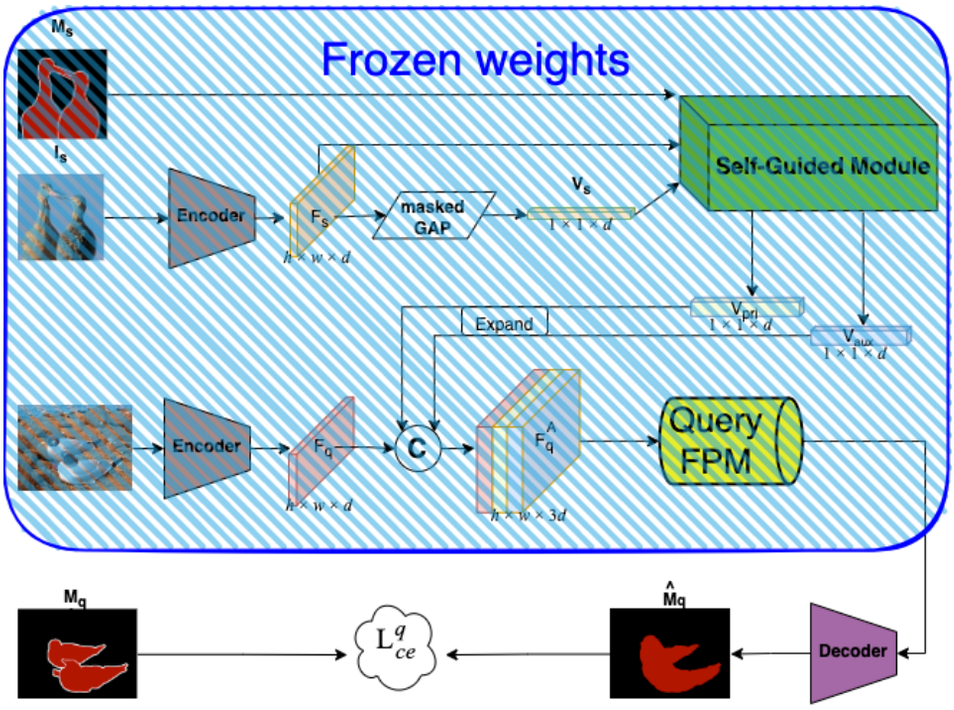


Figure 8.2: Overview of our IGFSS method applied to the Self-Guided Cross-Guided network [57] with modifications for fine-tuning its decoder. Only the bottom part of the network is used during the fine-tuning phase.

the final part of the network when fine-tuning the decoder. An overview can be seen in Figure 8.2.

To fine-tune the decoder we will use the support images as query images, as for the prototype fine-tuning. In the 1-shot case the support image will be trained on itself. For the 5-shot case each support image will have all support images as support images including itself.

In both cases we will then save the output from the Query Featuring Processing Module (FPM) for each support as query image. This will speed up the actual fine-tuning process since we do not need to use the whole network in each step.

Since the decoder is in the last step of the network it can be seen as a small network trained on a dataset with only 5 images. This will make the fine-tuning fast after the initial run through the whole network to get the input to the decoder. The loss is calculated with cross entropy between the labeled support mask and the output from the decoder. Then the decoder is updated with backpropagation.

In this method we are only focusing on the decoder in the query step of the network. Note that this decoder is also used in the self guided module for for finding the loss between the generated support prediction and its label. This is only done when training the whole network and not affecting the query output in the test phase.

9

Dataset Analysis

In this chapter we will take a closer look at the $Pascal - 5^i$ dataset proposed by OSLSM [39], combining the PASCAL VOC 2012 [13] and SBD dataset [17]. In this thesis we are referring to the $Pascal - 5^i$ dataset as just the Pascal dataset. It is a dataset with labeled segmentation maps for each image in the dataset. The dataset contains everyday images from a wide range of scenarios. There are 20 different classes in the dataset, cat, dog etc. A full overview can be seen in Table 9.2 and some example images can be seen in Figure 9.1

We will take a closer look at how the different classes in the dataset are distributed. This will give us a better chance to interpret the result from different models in Chapter 10.

Figure 9.1: Example of images from the Pascal dataset.

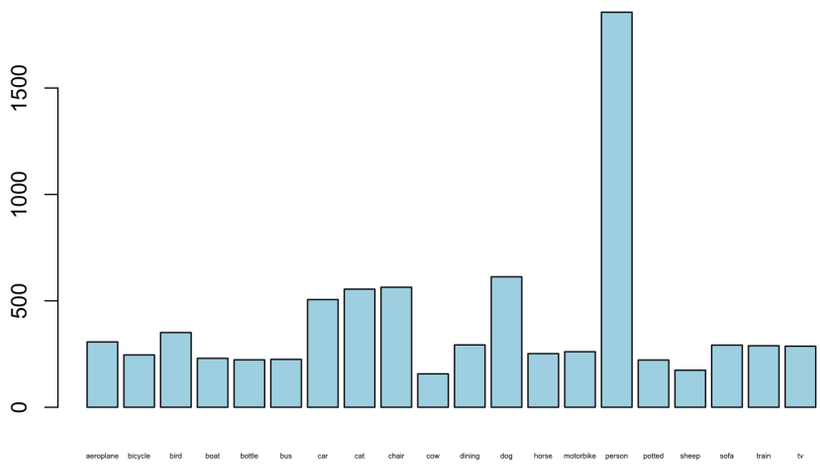


Figure 9.2: A visual overview of how the classes in the Pascal dataset are distributed.

9.1 Image distribution

Together the two datasets have 5953 unique images, labeled on pixel level with 20 different classes. When using the dataset for training the SGCG network the dataset is split into 4 splits with 5 classes in each split. Here one split is used as a validation set and the remaining 3 splits as a training set.

Split #	Validation classes in split
0	1, 2, 3, 4, 5
1	6, 7, 8, 9, 10
2	11, 12, 13, 14, 15
3	16, 17, 18, 19, 20

Table 9.1: Overview of the data splits with their corresponding validation class. The remaining classes are used as training classes.

The class label and the amount of sample images in each class can be seen in Table 9.2. Here the classes are sorted in alphabetic order. It can be several classes in a single picture but normally not more then four different classes in one picture.

The dataset is cleaned by only using images where a class has at least 2*32*32 labeled pixels following the procedure of [57]. This is done since the SGCG-encoder downsizes the image to 1/32 of the original size. The cleaning will remove images containing mostly background except for small objects labeled

Class number	Class	# Sample images after removing small objects
1	aeroplane	307
2	bicycle	246
3	bird	351
4	boat	230
5	bottle	223
6	bus	225
7	car	506
8	cat	555
9	chair	564
10	cow	157
11	dining table	293
12	dog	613
13	horse	252
14	motorbike	261
15	person	1856
16	potted plant	222
17	sheep	174
18	sofa	292
19	train	289
20	tv-monitor	287

Table 9.2: Overview of pascal dataset with amount of images in each class.

to a class. In this procedure 213 images are removed and 5740 images remain with the class distribution shown in Table 9.1 and for a visual reference in Figure 9.2.

The labeled segmentation map for each image is a gray scale image with values for each pixel according to is class label. Here pixels with value 15 are labeled as class 15 "person" and so on. All pixels that do not belong to a class is classified as background with label 0. Each labeled object also has a border defined 255, this can be used as *ignore pixels* when training on segmentation of images. As the name implies these border pixels are ignored by the network since they are easily missclassified both by the network and the human labeling the pixel in the first place. Hence instead of arguing about what class these should belong to we can simply ignore them.

We can clearly see from Figure 9.2 that the "Person" class is over represented with 3 times the amount of the second largest class. The smallest class are the "Cow" class with less then a $\frac{1}{10}$ of the images in the "Person" class and a $\frac{1}{4}$ of the images in the second larges class. This will suggest that for getting a good

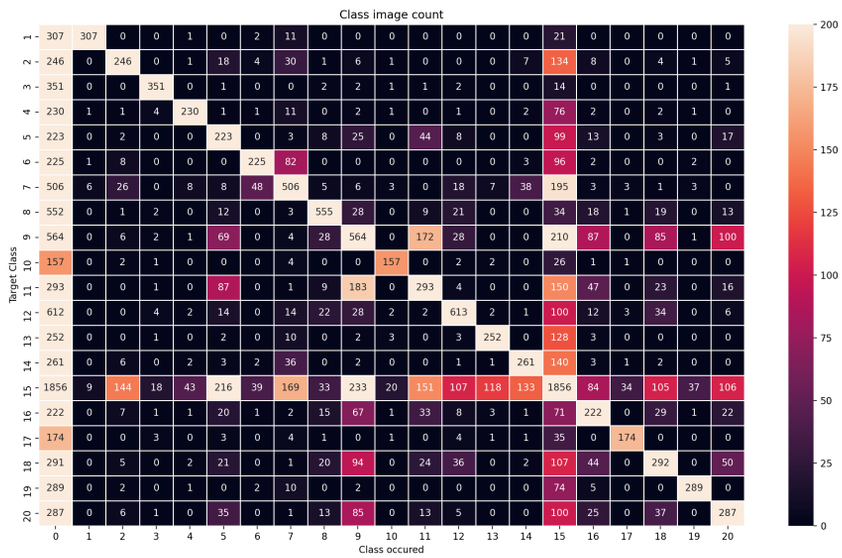


Figure 9.3: An overview of the the classes in the Pascal dataset are distributed. Here we can see how often other classes appears in images from the target class. The amount of images in each class can be read from the diagonal. Note that the background class, 0, is almost present in every image.

mean IoU score it is much more important to do well on the "Person" class than the rest. This however is not always an easy task since it is quite a bit variation within the person class. The reason for this is different clothing and postures, sometime its even appearing as commercial signs on top of other classes.

9.2 Image and Pixel Ratio

The amount of images in each class have given us a good indication on how the images are distributed in the Pascal dataset. Now we will focus more on the ratio how multiple classes are distributed within the images. This will give us a better understanding on how classes are distributed within the same images. Later we will look at the ratio between classes on a pixel level too. Dominant classes will have large objects compared to other classes and make them easier to segment. This is since large objects can easily be represented with high level features such as shapes [50].

We will start by taking a closer look at how multiple classes are distributed within the same images in Figure 9.3. Here we can see how often other classes

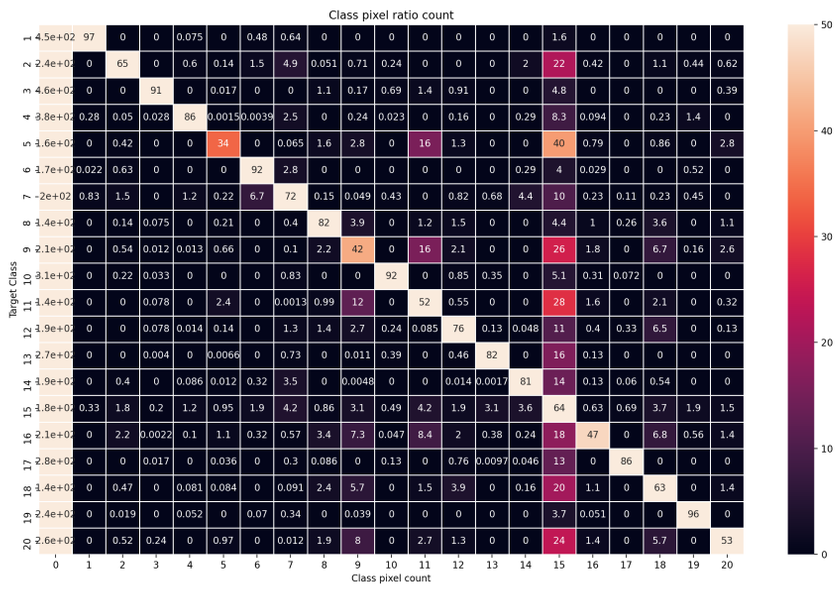


Figure 9.4: An overview of how the pixel ratio between classes in the Pascal dataset are distributed. Here we can see for each "Target class" how many pixels in percent are distributed compared with the other classes. Note that the background class, 0, is left out when calculating the percent rate and therefore exceeds 100% on its own.

appears in images from the target class. On the white diagonal we can see the amount of images in each class. Then the row represents where other classes are in those images. Note that class 0 is the background class and are almost present in every image.

It is easy to see that class 15 "Person" stands out with much more color and are often represented with other classes. It is present in a significant part of almost every other class. It is also the largest additional class to every other class. This indicates that class 15 strongly affects the dataset and the performance of the networks trained on segmenting images from it.

We can then look at the pixel ratio in the images to see how much each class is represented. This can give an indication if a class is the main objective of images it is part of. An overview is seen in Figure 9.4, here class 15 "Person" stands out as well but not as dominant as before. Here the extreme cases are class 5 ("Bottle"). Here there are more pixels labeled "Person" than "Bottle" in the bottle images. Other extreme cases are class 1 and 19 ("Aeroplane" and "Train") that are almost exclusive labeled objects in there images. It is still clear

that class 15 has much to say as it is still the largest class in addition to the the "Target Class" when we look at pixel ratio.

When training a network for segmenting images from the Pascal dataset the class score will have very mixed contributions to the overall score. This will be an important factor when we look at the results in the next chapter.

10 Experiments and results

In this chapter we will look at experiments with our Inference Guided Few-Shot Segmentation method (IGFSS), with the Self-Guided Cross-Guided network (SGCG) [57] as background. We will start with the reproduction of the SGCG network. Then we will look at how our IGFSS method affects the results of the SGCG network. Finally we will compare and discuss the results.

10.1 Reproduction of the Self-Guided Cross-Guided network

To make sure that the SGCG network give consistent results we tried to reproduce the results claimed by the authors. This will also give an indication on the spread of the results form different training runs and test sets.

The trained SGCG model will be used as a baseline to be able to compare it to our IGFSS method with SGCG as backbone. We will test two sets of weights for the baseline model. The reproduced model we trained and the weights provided from the original authors of the SGCG network. These will later be referred to as *Reproduced weights* and *Original weights*. Using two sets of weights will give an indication if our method gives consistent results.

	Split 1	Split 2	Split 3	Split 4	Mean
SGCG 1-shot	63.0	70.0	56.5	57.7	61.8
Replicated 1-shot	61.8	70.0	56.1	55.7	60.9

Table 10.1: Result from 1-shot training of the Self-Guided Cross-Guided network with the result from the paper on the first row and our represented result on the second row.

10.1.1 Experimental Setup Self-Guided Cross-Guided network

The SGCG network was trained on the Pascal dataset with the recommended hyperparameters from the provided code at `https://github.com/zbf1991/SCL`. There was one exception for the hyperparameters, the batch size was set to 1 instead of 4. This was due to a memory constrain on our available hardware. The backbone used for the SGCG network was the PFENet network [46], this was chosen since it gave the best mIoU score.

During training and validation common practice was followed and the dataset was split in the same 4 splits with classes divided in alphabetic order as the authors. In the training phase the network was validated on 1000 pairs of support-query images that are sampled randomly from the validation set. The evaluation is made with the IoU metric comparing the foreground-background overlap. The SGCG network was only trained in the 1-shot setting following the method of the authors. The same weights where used for evaluation on both the 1-shot and 5-shot setting.

10.1.2 Results Self-Guided Cross-Guided network

The reproduced result were slightly worse than the ones claimed by the authors. The best results were given at epoch 99, 36, 33 and 63 for split 1-4. A comparison is seen in Table 10.1 where we see that our reproduction matched the reported results in [57] exactly for split 2, while being close for the other splits.

To reduce uncertainty in the results a fixed list of validation data is used. This list is generated from an random sampling from the dataset. Since this split contains images from all classes in the dataset we have to make sure to use the model **not** trained on the class of the current image. So for the fine-tuning validation four trained models are used on the fixed validation set. This makes sure that the model always get a new class that its never been training on. However it is likely that the model have seen object from that class in training but then the object have been labeled as background. This especially true for class 15 ("People") since it is appearing in all other classes as we discussed in

the Chapter 9.

For the rest of the results the fixed list of query-support pairs was used for a fair comparison between different setups of our IGFSS method. The results from the fixed list can be seen in Table 10.2. Here we can see that there is quite a big spread in IoU score for different classes. Here class 15 stands out with the lowest score, this is also by far the largest class in the dataset as we discussed in Chapter 9. This also makes class 15 have the largest potential to increase the overall performance for the network from a single class.

The IoU score tends to stay in the same range between different setups of weights and amount of shots. This is a good indication of small variance coming from change of weights or amount of support images.

Note that the results from our fixed list are lower then the first reproduction. This comes from the fact that our random selected list includes challenging examples.

10.2 Inference Guided Few-Shot Segmentation on Prototypes

The prototypes are carrier of the information given by the support set, improving them should be able to increase the IoU score. In this section we will take a closer look at the results from our IGFSS method used on the prototypes.

10.2.1 Experimental Setup Prototypes

When using our IGFSS method on the SGCG network we keep all hyperparameters the same as for the reproduction setup. The additional learning rate for the fine-tuning of the prototypes was added. It is set to 0.025 for the 1-shot case and increased to 1.0 for the 5-setup to compensate for slower learning. This can be compared to the 0.0025 learning rate for the original training. It is a substantial difference but for the fine-tuning we are not interested in the optimal solution for the support set since this would most probably overfit the network. Instead we are looking for a slightly more target class specific prototype.

To fine-tune the prototypes one pass through the whole network is done once. This give us a benchmark of the same query-support set used for comparison later. The prototypes are saved and all other weights are frozen. When the

class	Name	1-shot original	1-shot reproduced	5-shot original	5-shot reproduced
1	aeroplane	80	81	81	82
2	bicycle	36	37	36	37
3	bird	78	82	76	82
4	boat	70	65	66	66
5	bottle	57	42	63	46
6	bus	88	89	86	89
7	car	63	65	65	66
8	cat	92	90	92	89
9	chair	21	26	26	29
10	cow	92	93	92	93
11	dining table	25	18	22	16
12	dog	88	88	87	85
13	horse	86	89	88	89
14	motorbike	79	77	82	81
15	person	9	10	8	9
16	potted plant	35	34	34	35
17	sheep	92	92	92	92
18	sofa	76	62	68	64
19	train	81	83	82	83
20	tv-monitor	39	36	37	36
mean		58	58	59	58

Table 10.2: Result for the SGCG network on the fixed list of query support set. On the rows we can see IoU score for each class and mean IoU on the last row.

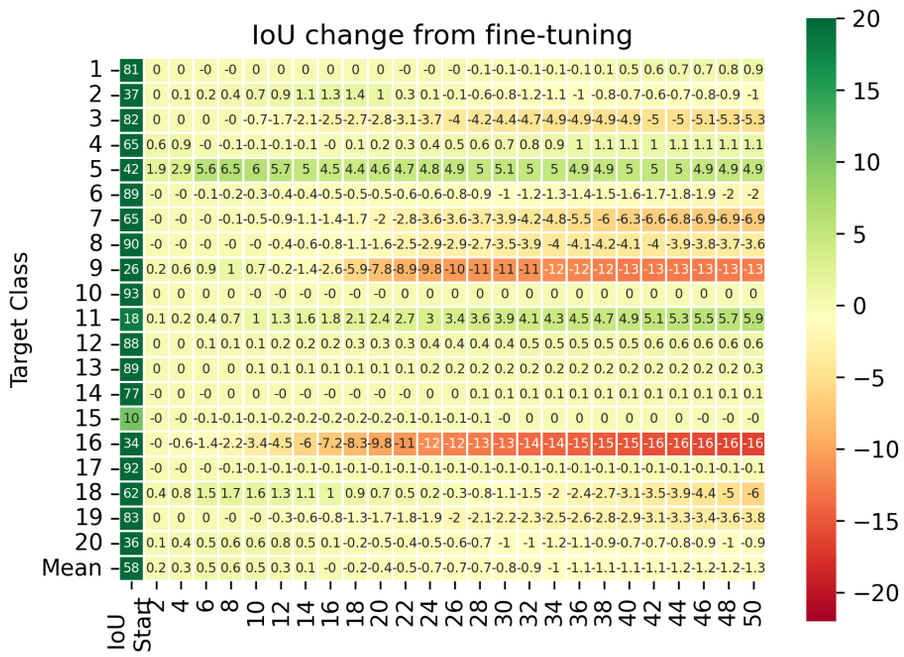


Figure 10.1: Results from our IGFSS **1-shot** method on the **prototypes** with **reproduced** weights and the learning rate is set to 0.025.
The rows represent a Target class from the dataset, the last row is the mean for all classes. The columns will represent the fine-tuning epoch. The first column, IoU Start, is the IoU score in % before any fine-tuning has been done. The other values in the matrix are the difference between the fine-tuned and non-fine-tuned IoU score in %.

support images are used as query to fine-tune the prototypes only the query part of the network is used to save computation time. We are also continuously comparing our IGFSS method to the SGCG original to get good overview on the improvement of our method.

10.2.2 Results Prototypes 1-Shot

In the 1-shot setting for the prototypes our IGFSS method did not manage to get an significant improvement of the mIoU score. It was expected that our model would have some trouble since it only got one support image. The results also indicated that our method starts over-fitting quite fast.

In Figure 10.1 a comparison of the results before and after our IGFSS method

was applied to the SGCG network can be seen. Here the mIoU peaks in the first few epochs with an 0.6% increase before it drops to consistent negative results after 16 epochs. There are some significant improvements in IoU score for class 5 and 11 and significant decreasing IoU score for class 7, 9 and 16 The negative overall trend is consistent after the first 50 epochs with only class 11 still having significant positive change in IoU score.

10.2.3 Results Prototypes 5-shot

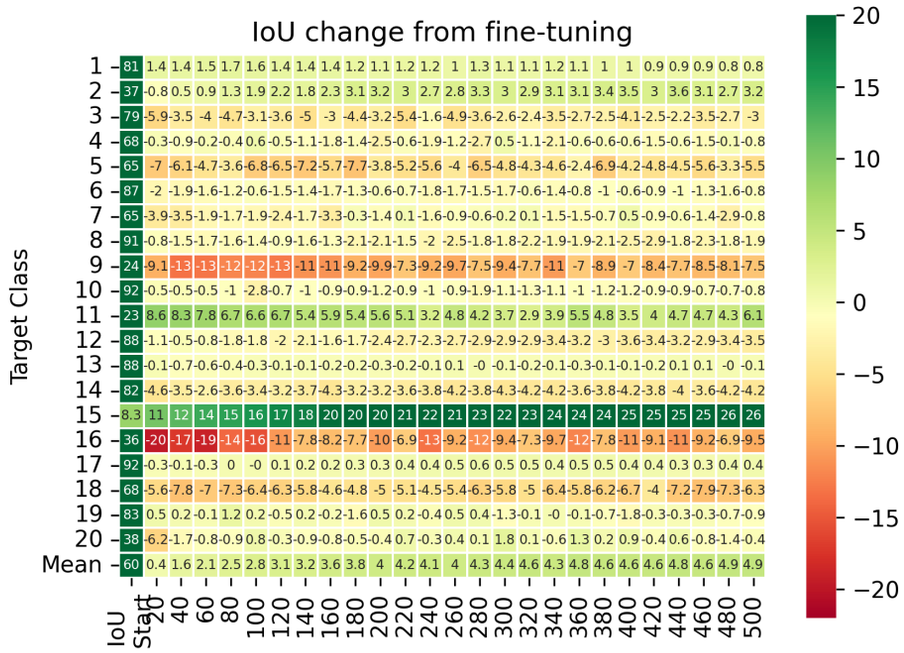
When we look at the results from the 5-shot IGFSS method we were expecting better results than for the 1-shot cases. All 6 different 5-shot alternatives where tested, one or several sets of prototypes and IoU score as weights. The best result was given from alternative 1A where we used one set of prototypes and the mean loss. For the 1B alternative we got similar improvements in IoU score, but as big as for 1A. A before and after comparison with our IGFSS method on both alternative 1A and 1B can be seen in Figure 10.2 Otherwise there were no significant increase in mIoU score for the other alternatives, so we will focus on them further in this thesis.

The time needed to fine-tune each picture increased around 5 times as expected. In the 5-shot case the fine-tuning took much longer to get a peak result. To compensate for this the learning rate was increased from 0.025 to 1.0. This is a substantial increase in learning rate to get a peak in mIoU score of 4.9% at epoch 480. The result seem to be quite stable after 320 epochs. After 500 epochs the mIoU score decreases.

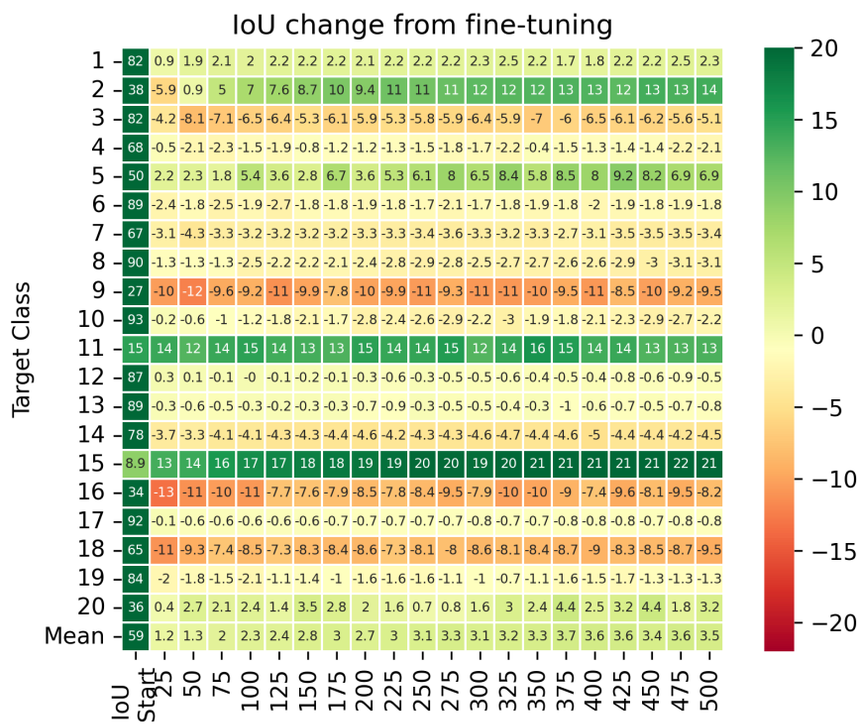
For alternative 1A our IGFSS method significantly increased the IoU score for class 2, 11 and 15 with 3-26 %. These were to begin with low scoring classes, so the potential increase was high. The decrease in score where significant for class 5, 9, 14, 1 and 18, with around 4-10% decrease in IoU score. Overall we have more classes with decreasing score but the mIoU was still increasing much with help of class 15. This class is much larger then the other classes as discussed in Chapter 9.

The 1B alternative got similar results but with some notable differences, here there is a strong increase for class 2, 5, 11 and 20 compared to alternative 1A. Class 15 has less of an increase but still at 21% increase.

It seems like more classes has a good response on the weighted loss of alternative 1B. But the overall score increases more from the mean loss of alternative 1A, with a high contribution of the dominant class 15.



(a) 5-shot 1A



(b) 5-shot 1B

Figure 10.2: Results from our IGFSS **5-shot** method on the **prototypes** with alternative (a)**1A** and (b)**1B**, the learning rate is set to 1.0.
The rows represent a Target class from the dataset, the last row is the mean for all classes. The columns will represent the fine-tuning epoch. The first column, IoU Start, is the IoU score in % before any fine-tuning has been done. The other values in the matrix are the difference between the fine-tuned and non-fine-tuned IoU score in %.

10.3 Inference Guided Few-Shot Segmentation on Decoder

Our IGFSS method was also applied to the decoder in the SGCG network. The decoder is the last step for producing the final segmentation, so making the decoder a bit more specialised on the target class could improve the results. We will first look at the results for the 1-shot case and then the 5-shot case.

10.3.1 Experimental Setup Decoder

Similar to the prototype case we set all hyperparameters according to the SGCG authors setup to get the best comparison. The additional learning rate for the fine-tuning of the decoder are set to 0.025 for both the 1-shot and 5-shot case.

When our IGFSS method is used we start with passing the query-support set through the whole network. Then all weights are frozen except for the decoder. We use the whole network to segment each support image as query image one by one. To save computation time, the inputs for the decoder are saved for each support image. In the fine-tuning process the decoder is then used as a small network with the saved prepossessed support images as input. This greatly decreases the training time compared to using the whole network.

We are continuously comparing our IGFSS method to the SGCG original every tenth epoch. This will give us a good overview on the improvement of our method.

10.3.2 Results Decoder 1-shot

For the 1-shot case the network only has one support image to train on, so we are not expecting great results. In Figure 10.3 we can see an comparison before and after applying our IGFSS method to the SGCG network. The results still looks a promising with a consistent increase of mIoU score 1.4% after 40+ epochs. It seems like most of this improvement comes from class 2, 11 and 15. Here we can consider class 15 as the main contributor for the increasing overall performance since it is the largest class and has the highest increase in IoU score, 14%. It is also worth noting that all of the classes with a large improvement had a low IoU before our Inference Guided method.

Even if the mIoU score increased, most classes had a decrease in IoU score. Here class 8 and 18 sticks out with more then 10% decrease followed by class

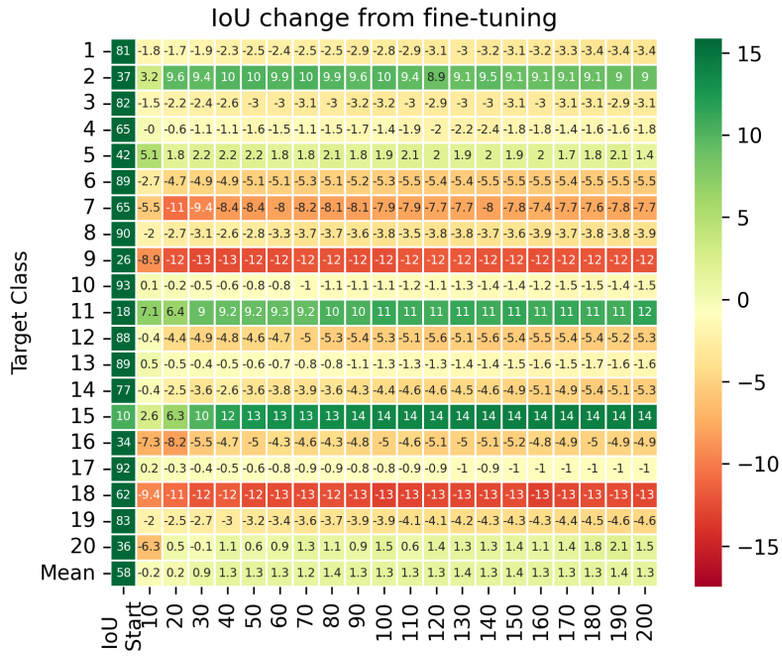


Figure 10.3: Results from our IGFSS **1-shot** method on the **decoder** with **reproduced** weights and the learning rate is set to 0.025.
The rows represent a Target class from the dataset, the last row is the mean for all classes. The columns will represent the fine-tuning epoch. The first column, IoU Start, is the IoU score in % before any fine-tuning has been done. The other values in the matrix are the difference between the fine-tuned and non-fine-tuned IoU score in %.

6, 7, 9, 12, 14 and 19 who all decreased around 5%. This shows how big impact a large class can have if it is performing very poorly from the start.

10.3.3 Result Decoder 5-Shot

When using five support images instead of one we are expecting more improvement from our IGFSS method. It is very easy to overfit on just one image, having a few more should make parameter optimization more general.

Our IGFSS method seems to work good in general for the Pascal data set as we can see in Figure 10.5. With a substantial improvement for the mIoU after around 60 epochs that is quite consistent the rest of the way to 200 epochs. The increase in mIoU is mostly contributed from class 2, 11 and 15 which had week results from before. Class 15 stands out significantly with the initially

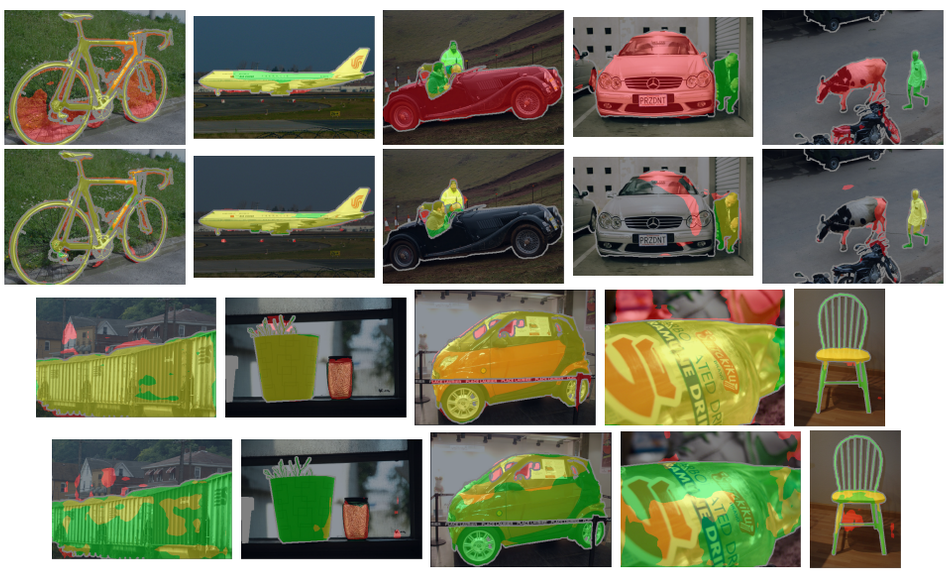
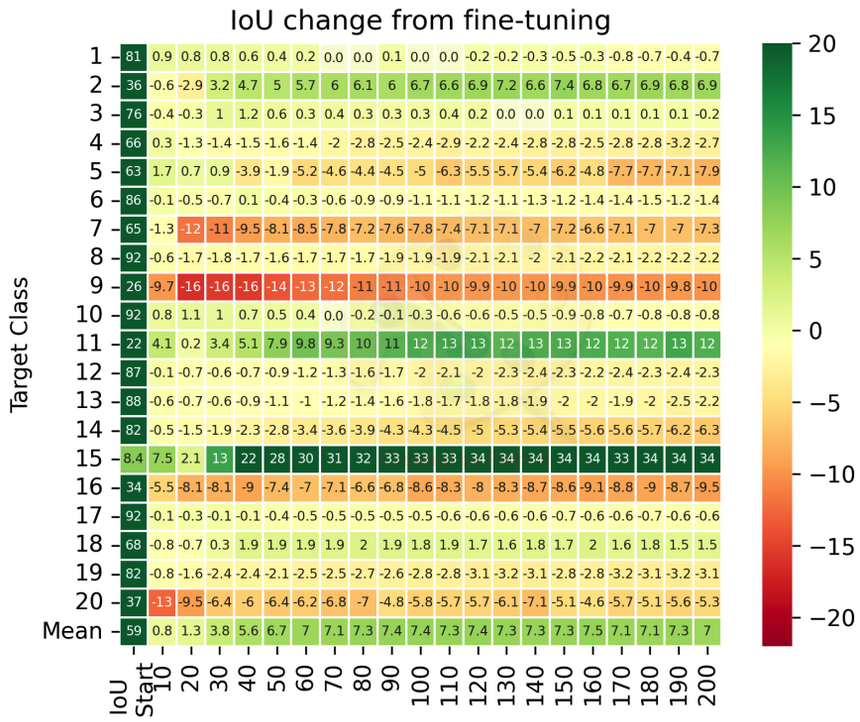


Figure 10.4: Example images before and after Inference Guided method. Here green is the true label that has not been predicted, yellow is true prediction, red is wrongly predicted areas and gray is ignored pixels. The first row is bad examples from the SGCG network that got improved with our IGFSS method in the second row. The third row is good examples from the SGCG network that got worse with our IGFSS method in the forth row.

lowest IoU score and the highest increase. For the reproduced weights the IoU score for class 15 started at 8.9% and increased with 40% IoU score to 49%! The results for the original weights also follows this pattern with a little less increase.

Not all low scoring classes has been improved, class 9 and 16 started with a low IoU score of 29% and 36% and decreased with more than 10%. In Figure 10.4 we can see some examples of images with overlaying predicted segmentation maps. These are examples with significant changes to the predicted segmentation map after 200epochs with IoU score going in both directions. Here yellow is correctly classified, green is ground truth and res is wrong predictions. In the top section we have images that improved and the bottom images with decreasing results.



(a) Original weights.

(b) Reproduced weights.

Figure 10.5: Results from our IGFSS **5-shot** method on the **decoder** with (a)**original** and (b)**reproduced** weights and the learning rate is set to 0.025. The rows represent a Target class from the dataset, the last row is the mean for all classes. The columns will represent the fine-tuning epoch. The first column, IoU Start, is the IoU score in % before any fine-tuning has been done. The other values in the matrix are the difference between the fine-tuned and non-fine-tuned IoU score in %.

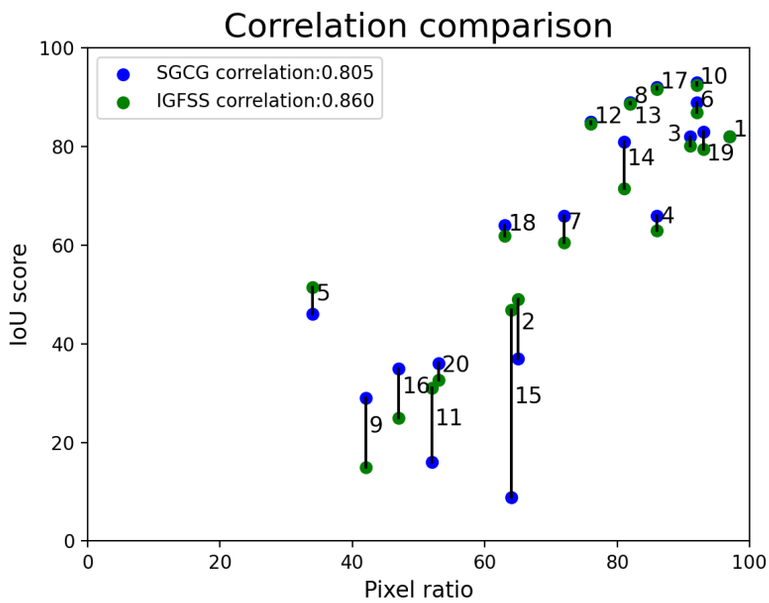


Figure 10.6: Comparison between class IoU score (SGCG in blue and IGFSS in green) and class pixel ratio from Figure 9.4 . The class pixel ratio is the amount of pixels in the target class compared to other labeled classes, excluding the background class. The numbers represent class numbers. There is a strong correlation between both axis, 0.805 and 0.860 for SGCG and IGFSS respectively.

10.3.4 IoU and Object Size Correlation

The results from the class IoU score have a lot of variation between classes. The reason for this seems to be correlated to the size of the target class object in each image. Images with a larger object compared to the other labeled classes tend to give a better IoU score.

In Figure 10.6 we can see how the different classes are distributed with respect to pixel ratio and IoU score from SGCG network and our IGFSS method. The pixel ratio is comparing the amount of pixels from a target class against all other labeled classes. The background class is ignored for this metric. There is a strong positive correlation between pixel ratio and IoU score. When our IGFSS method is applied to the SGCG network the correlation is increasing from 0.805 to 0.860.

This correlation could be a good factor for deciding if our IGFSS method should be applied to an existing method or specific target class.

Method	mIoU	Epoch	Learning rate
SGCG(PFENet) 1-shot	58.2	–	–
IGFSS (SGCG) 1-shot prototypes	58.8	8	0.025
IGFSS (SGCG) 1-shot decoder	**59.6**	80	0.025
SGCG(PFENet) 5-shot	58.5	–	–
IGFSS (SGCG) 5-shot prototypes 1A	63.4	480	1.0
IGFSS (SGCG) 5-shot prototypes 1B	62.2	375	1.0
IGFSS (SGCG) 5-shot prototypes 1C	60.4	320	1.0
IGFSS (SGCG) 5-shot prototypes 2A	58.8	440	1.0
IGFSS (SGCG) 5-shot prototypes 2B	58.6	160	1.0
IGFSS (SGCG) 5-shot prototypes 2C	58.8	160	1.0
IGFSS (SGCG) 5-shot decoder	**65.4**	70	0.025

Table 10.3: Mean IoU score for different alternatives of our IGFSS method on the SGCG network on our fixed query-support list. The learning rate and epoch is for the fine-tuning in the inference phase. All other parameters are shared for all alternatives.

10.4 Result comparison and discussion

We have now looked at the results from our IGFSS method used on both the prototypes and decoder one by one. We will now summarize and compare the results from the different setups. We will also see how they compare to recent state of the art networks. Then we will look at the the results in relation to our previous data analyse to see if we can understand the behaviour of our IGFSS method better. An overview of the mIoU score for all alternatives of our IGFSS model can be found in Table 10.3.

1-shot

For our IGFSS method on both the prototypes and decoder we did not expect any big improvements in the 1-shot setting. Most likely it would overfit on the single support image quite fast. A bit surprisingly our IGFSS method manages to increase the result in both cases. However it is debatable if the prototype became any better with such a short span of improvement until it had a decreased mIoU score.

The decoder had better results in the 1-shot case, here we got some consistent improvement of mIoU after 40 epochs. After this not much changes for either the mIoU or for the class IoU. This is a good sign as it makes the method less sensitive for tuning the right number of epochs in the inference phase on unseen data.

For both the prototypes and the decoder the positive results are however well spread between the classes. For most classes we have a negative result which indicates that it is not worth using this method for all cases. If possible the results from our IGFSS and its backbone should be compared to a small validation set.

5-shot

In the 5-shot case we expected our IGFSS method to do a much better result. Here we are less likely to overfit and have the possibility to get some more general learning while fine-tuning on the support set.

For the prototypes we manged to get a significant improvement of mIoU that was consistent after 200 epochs with a learning rate increased to 1.0. The mIoU score is positive all the way which is a good indication of the method. But as with the 1-shot setups the increase in mIoU comes mostly from the increase in just a few classes and mostly class 15.

The decoder got even better results with consistent increase in mIoU for all epochs. This mostly comes from a very large increase in the IoU of class 15. There is some significant increase of IoU score for a few other classes as well. But the majority of classes sees a small decrease in IoU score like all other setups. Some classes also have a significant decrease in IoU score.

Compared to other networks

To get a reference point on how this performance compares to previous networks we can look at Table 10.4. Here we can see how previous state of the art methods performed few-shot segmentation on the Pascal dataset. On our fixed list of query-support images the SGCG network had worse results then the ones stated by [57]. So for a fair comparison we have the mIoU score for our IGFSS method and the SGCG network on the same fixed list marked with *.

Computation speed

To apply the method to a real problem it is also important that it can be done efficiently. For the prototypes our IGFSS method needed many epochs to obtain a solid improvement. The combination of more epochs and a larger network to back propagate through led to a quite long inference phase for the 5-shot setup. The decoder work more like a mini network here where we can back

Method	Backbone	1-shot	5-shot
OSLSM (BMVC'17) [39]	vgg16	40.8	44.0
SG-One [60]	vgg16	46.3	47.1
PANet (ICCV'19) [49]	vgg16	48.1	55.7
PGNet (ICCV'19) [59]	resnet50	56.0	58.5
CRNet (CVPR'20) [29]	resnet50	55.7	58.8
RPMMs (ECCV'20) [53]	resnet50	56.3	57.3
FWB (ICCV'19) [35]	resnet101	56.2	59.9
PPNet*(ECCV'20) [30]	resnet50	51.5	62.0
DAN (ECCV'20) [48]	resnet101	58.2	60.5
CANet (CVPR'19) [58]	resnet50	55.4	57.1
PFENet (TPAMI'20) [46]	resnet50	60.8	61.9
SGCG (PFENet) [57]	resnet50	61.8	62.9
SGCG* (PFENet) [57]	resnet50	58.2	58.5
ours IGFSS-Prototype* (SGCG)	resnet50	58.8	62.2
ours IGFSS-Decoder* (SGCG)	resnet50	59.6	65.4

Table 10.4: Overview of the mIoU score from few shot segmentation networks on the Pascal dataset with 1-shot and 5-shot settings.
* Tested on our fixed list.

propagate only to the weights being updated. This gives the decoder setup a significantly faster inference speed. It is also worth considering the amount of epochs needed to get consistent positive results. The prototypes got better results after 200 epochs while the decoder had stable good results after 50-70 epochs.

In our experiments we have used a single Nvidia 1080titan graphics card. In the 5-shot cases it takes 2 seconds per image for the standard SGCG network. When we apply our IGFSS method to the prototypes it takes 47 second per image in the inference phase to fine-tune 200 epochs. This can be compared to the decoder setup which only takes 12 seconds. Note that these inference times should only be seen as guidelines as the code has not been optimized for performance.

11

Future work and Conclusion

In this chapter we will discuss how our IGFSS method can be further improved in the future. Then we will end this thesis with a short conclusion.

11.1 Future work

Our IGFSS method have shown that it can be used to improve the overall results and in particular the results for some specific classes. There is however always room for improvement, that can be considered in future work. In particular, current short comings are the decreased IoU score for some classes. To minimize the negative results of specific classes and in the same time maximizing the positive results a class specific stop criterion could be introduced. This could be done in the form of an *early stopping*, where we stop the fine-tuning when a specific criterion is met for each query support set.

Finding the optimal criterion would require further experiments, some possible candidates are when the value from a loss function or IoU score increases. To test these criterion we can't use the query image, since we do not have the true label. In the 5-shot setting we could use the support images as query one by one while having the remaining 4 as support.

We could also set a epoch by analysing the class data, here object size seem to have a large impact on the potential improvement and could be a good candidate. This approach would work for both the 1-shot and 5-shot setting.

To future evaluate the generalizability of our purposed method, a test on a different dataset such as the COCO dataset could be done. Since this is a larger and more variable data set than the Pascal dataset it should give a good indication on how reliable our new IGFSS method is. Another way to further test the generalizability would be to use our IGFSS on a new backbone.

In our experiments we used the same hyperparameters as the SGCG. Our additional learning rate for the fine-tuning was set to 0.025 and 1.0 after a minor parameter search. Future exploration of the hyperparameter space could benefit the results but is considered to be out of the scope of this thesis.

11.2 Conclusion

In this thesis we have focused on few-shot segmentation networks. We introduced our *Inference-Guided Few-Shot Segmentation* (IGFSS) that can be applied to an existing decoder-based few-shot segmentation network. Our method aims to improve the usage of information from the support set in the inference phase. This is done by fine-tuning either the decoder or the prototypes in the inference phase of a pretrained backbone network. We evaluated our IGFSS method on the Self-Guided Cross-Guided (SGCG) [57] network. Our IGFSS method outperforms the SGCG baseline with a considerable margin in the multi-shot case, both when applied to the prototypes and the decoder.

Moreover we analysed the Pascal dataset [13] used in our experiments to better understand the results. Here we discovered a strong correlation between the IoU score and object size of the target class. When our IGFSS method was applied the correlation increased even more.

Bibliography

[1] Forest Agostinelli et al. "Learning Activation Functions to Improve Deep Neural Networks." In: *arXiv:1412.6830 [cs, stat]* (Apr. 2015). arXiv: 1412.6830. URL: http://arxiv.org/abs/1412.6830 (visited on 05/11/2022).

[2] Spyridon Bakas et al. "Advancing The Cancer Genome Atlas glioma MRI collections with expert segmentation labels and radiomic features." eng. In: *Scientific Data* 4 (Sept. 2017), p. 170117. ISSN: 2052-4463. DOI: 10.1038/sdata.2017.117.

[3] Y. Bengio, P. Simard, and P. Frasconi. "Learning long-term dependencies with gradient descent is difficult." In: *IEEE Transactions on Neural Networks* 5.2 (Mar. 1994). Conference Name: IEEE Transactions on Neural Networks, pp. 157–166. ISSN: 1941-0093. DOI: 10.1109/72.279181.

[4] Chris M. Bishop. "Training with Noise is Equivalent to Tikhonov Regularization." en. In: *Neural Computation* 7.1 (Jan. 1995), pp. 108–116. ISSN: 0899-7667, 1530-888X. DOI: 10.1162/neco.1995.7.1.108. URL: https://direct.mit.edu/neco/article/7/1/108-116/5828 (visited on 06/07/2022).

[5] Malik Boudiaf et al. "Few-Shot Segmentation Without Meta-Learning: A Good Transductive Inference Is All You Need?" en. In: *arXiv:2012.06166 [cs]* (Mar. 2021). arXiv: 2012.06166. URL: http://arxiv.org/abs/2012.06166 (visited on 05/10/2022).

[6] Joel Burman. *An overview of Few-Shot Segmentation*. 2021.

[7] Tianshi Cao, Marc Law, and Sanja Fidler. *A Theoretical Analysis of the Number of Shots in Few-Shot Learning*. Tech. rep. arXiv:1909.11722. arXiv:1909.11722 [cs, stat] type: article. arXiv, Feb. 2020. DOI: 10.48550/arXiv.1909.11722. URL: http://arxiv.org/abs/1909.11722 (visited on 06/01/2022).

[8] Liang-Chieh Chen et al. "DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs." In: *arXiv:1606.00915 [cs]* (May 2017). arXiv: 1606.00915. URL: http://arxiv.org/abs/1606.00915 (visited on 05/11/2022).

[9] Wei-Yu Chen et al. "A Closer Look at Few-shot Classification." In: *arXiv:1904.04232 [cs]* (Jan. 2020). arXiv: 1904.04232. URL: http://arxiv.org/abs/1904.04232 (visited on 05/11/2022).

[10] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. "Improving deep neural networks for LVCSR using rectified linear units and dropout." In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. ISSN: 2379-190X. May 2013, pp. 8609–8613. DOI: 10.1109/ICASSP.2013.6639346.

[11] Dipankar Das et al. "Distributed Deep Learning Using Synchronous Stochastic Gradient Descent." In: *arXiv:1602.06709 [cs]* (Feb. 2016). arXiv: 1602.06709. URL: http://arxiv.org/abs/1602.06709 (visited on 05/11/2022).

[12] Geoffrey E. Hinton and Ronald J. Williams David E. Ruineihart. *Learning Internal Representations by Error Propagation*.

[13] Mark Everingham et al. "The Pascal Visual Object Classes (VOC) Challenge." en. In: *International Journal of Computer Vision* 88.2 (June 2010), pp. 303–338. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-009-0275-4. URL: http://link.springer.com/10.1007/s11263-009-0275-4 (visited on 05/11/2022).

[14] Abdur R. Feyjie et al. "Semi-supervised few-shot learning for medical image segmentation." In: *arXiv:2003.08462 [cs]* (Apr. 2020). arXiv: 2003.08462. URL: http://arxiv.org/abs/2003.08462 (visited on 05/11/2022).

[15] Yu Gordienko et al. "Deep Learning with Lung Segmentation and Bone Shadow Exclusion Techniques for Chest X-Ray Analysis of Lung Cancer." In: *arXiv:1712.07632 [cs]* 754 (2019). arXiv: 1712.07632, pp. 638–647. DOI: 10.1007/978-3-319-91008-6_63. URL: http://arxiv.org/abs/1712.07632 (visited on 05/11/2022).

[16] Stine Hansen et al. "Unsupervised supervoxel-based lung tumor segmentation across patient scans in hybrid PET/MRI." en. In: *Expert Systems with Applications* 167 (Apr. 2021), p. 114244. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2020.114244. URL: https://www.sciencedirect.com/science/article/pii/S0957417420309623 (visited on 06/14/2022).

[17] Bharath Hariharan et al. "Simultaneous Detection and Segmentation." In: *arXiv:1407.1808 [cs]* (July 2014). arXiv: 1407.1808. URL: http://arxiv.org/abs/1407.1808 (visited on 05/11/2022).

[18] Kaiming He et al. "Deep Residual Learning for Image Recognition." In: *arXiv:1512.03385 [cs]* (Dec. 2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385 (visited on 05/11/2022).

[19] *Hierarchical Text-Conditional Image Generation with CLIP Latents*. en. Number: arXiv:2204.06125 arXiv:2204.06125 [cs]. Apr. 2022. URL: http://arxiv.org/abs/2204.06125 (visited on 06/14/2022).

[20] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *arXiv:1502.03167 [cs]* (Mar. 2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167 (visited on 05/11/2022).

[21] Kenji Kawaguchi. "Deep Learning without Poor Local Minima." In: *Advances in Neural Information Processing Systems*. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/hash/f2fc990265c712c49d51a18a32b39f0c-Abstract.html (visited on 06/07/2022).

[22] Çağrı Kaymak and Ayşegül Uçar. "A Brief Survey and an Application of Semantic Image Segmentation for Autonomous Driving." In: *arXiv:1808.08413 [eess]* (Aug. 2018). arXiv: 1808.08413. URL: http://arxiv.org/abs/1808.08413 (visited on 05/11/2022).

[23] J. Kiefer and J. Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." In: *The Annals of Mathematical Statistics* 23.3 (Sept. 1952). Publisher: Institute of Mathematical Statistics, pp. 462–466. ISSN: 0003-4851, 2168-8990. DOI: 10.1214/aoms/1177729392. URL: https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-23/issue-3/Stochastic-Estimation-of-the-Maximum-of-a-Regression-Function/10.1214/aoms/1177729392.full (visited on 05/11/2022).

[24] Wonjik Kim, Asako Kanezaki, and Masayuki Tanaka. "Unsupervised Learning of Image Segmentation Based on Differentiable Feature Clustering." In: *IEEE Transactions on Image Processing* 29 (2020). arXiv:2007.09990 [cs], pp. 8055–8068. ISSN: 1057-7149, 1941-0042. DOI: 10.1109/TIP.2020.3011269. URL: http://arxiv.org/abs/2007.09990 (visited on 05/24/2022).

[25] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In: *arXiv:1412.6980 [cs]* (Jan. 2017). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 05/11/2022).

[26] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. "Siamese Neural Networks for One-shot Image Recognition." en. In: (), p. 8.

[27] Brenden M Lake et al. "One shot learning of simple visual concepts." en. In: (), p. 6.

[28] Yann LeCun and Yoshua Bengio. "Convolutional networks for images, speech, and time series." In: *The handbook of brain theory and neural networks*. Cambridge, MA, USA: MIT Press, Oct. 1998, pp. 255–258. ISBN: 978-0-262-51102-5. (Visited on 05/11/2022).

[29] Weide Liu et al. *CRNet: Cross-Reference Networks for Few-Shot Segmentation*. Tech. rep. arXiv:2003.10658. arXiv:2003.10658 [cs] type: article. arXiv, Mar. 2020. DOI: 10.48550/arXiv.2003.10658. URL: http://arxiv.org/abs/2003.10658 (visited on 06/03/2022).

[30] Yongfei Liu et al. *Part-aware Prototype Network for Few-shot Semantic Segmentation*. Tech. rep. arXiv:2007.06309. arXiv:2007.06309 [cs] type: article. arXiv, Sept. 2020. DOI: 10.48550/arXiv.2007.06309. URL: http://arxiv.org/abs/2007.06309 (visited on 06/03/2022).

[31] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation." In: *arXiv:1411.4038 [cs]*

(Mar. 2015). arXiv: 1411.4038. URL: http://arxiv.org/abs/1411.4038 (visited on 05/11/2022).

[32] Harald Lykke Joakimsen. *Weakly Supervised Semantic Segmentation*. 2021.

[33] M.G. Miller, N.E. Matsakis, and P.A. Viola. "Learning from one example through shared densities on transforms." en. In: *Proceedings IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000 (Cat. No.PR00662)*. Vol. 1. Hilton Head Island, SC, USA: IEEE Comput. Soc, 2000, pp. 464–471. ISBN: 978-0-7695-0662-3. DOI: 10.1109/CVPR.2000.855856. URL: http://ieeexplore.ieee.org/document/855856/ (visited on 05/11/2022).

[34] Rahul Mohan. "Deep Deconvolutional Networks for Scene Parsing." In: *arXiv:1411.4101 [cs, stat]* (Nov. 2014). arXiv: 1411.4101. URL: http://arxiv.org/abs/1411.4101 (visited on 05/11/2022).

[35] Khoi Nguyen and Sinisa Todorovic. *Feature Weighting and Boosting for Few-Shot Segmentation*. Tech. rep. arXiv:1909.13140. arXiv:1909.13140 [cs] type: article. arXiv, Sept. 2019. DOI: 10.48550/arXiv.1909.13140. URL: http://arxiv.org/abs/1909.13140 (visited on 06/03/2022).

[36] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. "Learning Deconvolution Network for Semantic Segmentation." In: 2015, pp. 1520–1528. URL: https://openaccess.thecvf.com/content_iccv_2015/html/Noh_Learning_Deconvolution_Network_ICCV_2015_paper.html (visited on 05/11/2022).

[37] *Pricing | Data Labeling Service*. en. URL: https://cloud.google.com/ai-platform/data-labeling/pricing (visited on 05/11/2022).

[38] Shibani Santurkar et al. "How Does Batch Normalization Help Optimization?" In: *arXiv:1805.11604 [cs, stat]* (Apr. 2019). arXiv: 1805.11604. URL: http://arxiv.org/abs/1805.11604 (visited on 05/11/2022).

[39] Amirreza Shaban et al. "One-Shot Learning for Semantic Segmentation." In: *arXiv:1709.03410 [cs]* (Sept. 2017). arXiv: 1709.03410. URL: http://arxiv.org/abs/1709.03410 (visited on 05/11/2022).

[40] Evan Shelhamer, Jonathan Long, and Trevor Darrell. *Fully Convolutional Networks for Semantic Segmentation*. Tech. rep. arXiv:1605.06211. arXiv:1605.06211 [cs] version: 1 type: article. arXiv, May 2016. URL: http://arxiv.org/abs/1605.06211 (visited on 06/07/2022).

[41] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *arXiv:1409.1556 [cs]* (Apr. 2015). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556 (visited on 05/11/2022).

[42] Jake Snell, Kevin Swersky, and Richard S. Zemel. "Prototypical Networks for Few-shot Learning." In: *arXiv:1703.05175 [cs, stat]* (June 2017). arXiv: 1703.05175. URL: http://arxiv.org/abs/1703.05175 (visited on 05/11/2022).

[43] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. ISSN: 1533-7928. URL: http://jmlr.org/papers/v15/srivastava14a.html (visited on 05/11/2022).

[44] Kirill Svyatov et al. "Scenes Segmentation in Self-driving Car Navigation System Using Neural Network Models with Attention." In: June 2019, pp. 278–289. ISBN: 978-3-030-24307-4. DOI: 10.1007/978-3-030-24308-1_23.

[45] Nima Tajbakhsh et al. "Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning?" In: *IEEE Transactions on Medical Imaging* 35.5 (May 2016). Conference Name: IEEE Transactions on Medical Imaging, pp. 1299–1312. ISSN: 1558-254X. DOI: 10.1109/TMI.2016.2535302.

[46] Zhuotao Tian et al. "Prior Guided Feature Enrichment Network for Few-Shot Segmentation." In: *arXiv:2008.01449 [cs]* (Aug. 2020). arXiv: 2008.01449. URL: http://arxiv.org/abs/2008.01449 (visited on 05/11/2022).

[47] Oriol Vinyals et al. *Matching Networks for One Shot Learning*. Tech. rep. arXiv:1606.04080. arXiv:1606.04080 [cs, stat] type: article. arXiv, Dec. 2017. URL: http://arxiv.org/abs/1606.04080 (visited on 06/08/2022).

[48] Haochen Wang et al. "Few-Shot Semantic Segmentation with Democratic Attention Networks." en. In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi et al. Vol. 12358. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 730–746. ISBN: 978-3-030-58600-3 978-3-030-58601-0. DOI: 10.1007/978-3-030-58601-0_43. URL: https://link.springer.com/10.1007/978-3-030-58601-0_43 (visited on 06/03/2022).

[49] Kaixin Wang et al. "PANet: Few-Shot Image Semantic Segmentation with Prototype Alignment." In: *arXiv:1908.06391 [cs]* (Feb. 2020). arXiv: 1908.06391. URL: http://arxiv.org/abs/1908.06391 (visited on 05/10/2022).

[50] Xinlong Wang et al. "SOLOv2: Dynamic and Fast Instance Segmentation." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 17721–17732. URL: https://proceedings.neurips.cc/paper/2020/hash/cd3afef9b8b89558cd56638c3631868a-Abstract.html (visited on 06/16/2022).

[51] Yaqing Wang et al. "Generalizing from a Few Examples: A Survey on Few-Shot Learning." In: *arXiv:1904.05046 [cs]* (Mar. 2020). arXiv: 1904.05046. URL: http://arxiv.org/abs/1904.05046 (visited on 05/11/2022).

[52] Bing Xu et al. "Empirical Evaluation of Rectified Activations in Convolutional Network." In: *arXiv:1505.00853 [cs, stat]* (Nov. 2015). arXiv: 1505.00853. URL: http://arxiv.org/abs/1505.00853 (visited on 05/11/2022).

[53] Boyu Yang et al. *Prototype Mixture Models for Few-shot Semantic Segmentation*. Tech. rep. arXiv:2008.03898. arXiv:2008.03898 [cs] type: article. arXiv, Sept. 2020. DOI: 10.48550/arXiv.2008.03898. URL: http://arxiv.org/abs/2008.03898 (visited on 06/03/2022).

[54] Yuwei Yang et al. "A New Local Transformation Module for Few-shot Segmentation." In: *arXiv:1910.05886 [cs]* (Nov. 2019). arXiv: 1910.05886. URL: http://arxiv.org/abs/1910.05886 (visited on 05/11/2022).

[55] Xue Ying. "An Overview of Overfitting and its Solutions." en. In: *Journal of Physics: Conference Series* 1168 (Feb. 2019), p. 022022. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/1168/2/022022. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1168/2/022022 (visited on 05/11/2022).

[56] Fisher Yu, Vladlen Koltun, and Thomas Funkhouser. "Dilated Residual Networks." In: *arXiv:1705.09914 [cs]* (May 2017). arXiv: 1705.09914. URL: http://arxiv.org/abs/1705.09914 (visited on 05/11/2022).

[57] Bingfeng Zhang, Jimin Xiao, and Terry Qin. "Self-Guided and Cross-Guided Learning for Few-Shot Segmentation." In: *arXiv:2103.16129 [cs]* (Mar. 2021). arXiv: 2103.16129. URL: http://arxiv.org/abs/2103.16129 (visited on 05/11/2022).

[58] Chi Zhang et al. "CANet: Class-Agnostic Segmentation Networks with Iterative Refinement and Attentive Few-Shot Learning." In: *arXiv:1903.02351 [cs]* (Mar. 2019). arXiv: 1903.02351. URL: http://arxiv.org/abs/1903.02351 (visited on 05/11/2022).

[59] Chi Zhang et al. "Pyramid Graph Networks With Connection Attentions for Region-Based One-Shot Semantic Segmentation." en. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Seoul, Korea (South): IEEE, Oct. 2019, pp. 9586–9594. ISBN: 978-1-72814-803-8. DOI: 10.1109/ICCV.2019.00968. URL: https://ieeexplore.ieee.org/document/9010760/ (visited on 06/03/2022).

[60] Xiaolin Zhang et al. "SG-One: Similarity Guidance Network for One-Shot Semantic Segmentation." In: *IEEE Transactions on Cybernetics* 50.9 (Sept. 2020). Conference Name: IEEE Transactions on Cybernetics, pp. 3855–3865. ISSN: 2168-2275. DOI: 10.1109/TCYB.2020.2992433.

[61] Zhilu Zhang and Mert R. Sabuncu. "Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels." In: *arXiv:1805.07836 [cs, stat]* (Nov. 2018). arXiv: 1805.07836. URL: http://arxiv.org/abs/1805.07836 (visited on 05/11/2022).

[62] Hengshuang Zhao et al. *Pyramid Scene Parsing Network*. Tech. rep. arXiv:1612.01105. arXiv:1612.01105 [cs] type: article. arXiv, Apr. 2017. URL: http://arxiv.org/abs/1612.01105 (visited on 06/08/2022).

[63] Zhi-Hua Zhou. "A brief introduction to weakly supervised learning." In: *National Science Review* 5.1 (Jan. 2018), pp. 44–53. ISSN: 2095-5138. DOI: 10.1093/nsr/nwx106. URL: https://doi.org/10.1093/nsr/nwx106 (visited on 06/16/2022).