7CCSMPRJ Final Year

Few-Shot Semantic Segmentation with Deep Metric Learning in a Supervised and Multi-Dimensional Context

Final Project Report

Anon

Abstract

Semantic image segmentation is the problem of identifying whether individual pixels that make up an image belong to a specific object category. Given the data challenge in the real-world, increasing focus has been given to few-shot segmentation where models seek to learn from only a few labelled support images. Deep learning models have come to dominate the landscape by performance and recently deep metric learning has started to garner significant attention in its ability to leverage such deep architectures. This project seeks to implement and evaluate deep metric learning models in a few shot learning context. The project will compare object segmentation for two tasks, supervised segmentation (using only object class annotation information) and multi-dimensional segmentation (using object and part annotations) for object segmentation. The comparison will look to address whether the additional information provided by individual parts delivers better accuracy for overall object segmentation in a deep metric learning few-shot context.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 15,000 words.

Acknowledgements

Contents

Chapter 1

Introduction

1.1 Overview

Semantic image segmentation is the problem of identifying whether individual pixels that make up an image belong to a specific object category with the aim of partitioning an image with semantic labels. Semantic segmentation is an important area of focus in the field of computer vision given the rise of numerous applications which require fine-grained segmentation, including medical imaging, autonomous driving, and video surveillance amongst others.

Recently deep learning models have started to dominate the computer vision landscape delivering strong performance improvements in terms of accuracy rates compared to historic algorithms. These historic wide-ranging algorithms include, amongst others, thresholding [29], region segmentation[27], k-means clustering [9], conditional random fields [30] and sparsity-based methods [37] to name a few. Following the advent of deep learning models, Fully Convolutional Neural Networks ("FCNs") [19] have arisen as one of the most popular architectures for semantic segmentation, given their performance and efficiency in end-to-end training for pixel level classification. However, training these models typically require a large number of images in the training dataset which can be challenging in certain domains where there is limited data availability. These models can also suffer from a lack of generalisability for new object classes [34].

These issues have given rise to increased focus on few-shot segmentation where models seek to learn from only a few labelled support images. Few-shot segmentation aims to segment a query image from unseen object classes, in contrast to those object classes used for training, given a "few" support images containing the same unseen object class and ground truth labels for the support images.

Deep Metric Learning ("DML") has garnered significant attention recently in its ability to leverage deep architectures for metric learning and its ostensible suitability for semantic segmentation in a few shot context. Deep Metric Learning seeks to learn an embedding space such that pixels of the same object class are close together and pixels from other object classes are further away using a defined distance metric [20]. Deep metric learning has been applied to semantic segmentation by utilising FCNs as feature extractors and utilising a deep metric learning loss function which incorporates a distance metric between features to train the network.

This project seeks to implement and evaluate deep metric learning models in a few shot learning context. The project will compare object segmentation for two tasks: (1) supervised segmentation - using only object class annotation information for training; (2) multi-dimensional segmentation - using both object class annotations and multi-dimensional annotation information of the respective objects (i.e., individual parts composing an object class). The comparison will look to address whether the additional information provided by individual parts of objects aid the model in coping with diverse appearance in objects with different parts, poses or subcategories, and therefore result in better accuracy for overall object segmentation in a deep metric learning few-shot context.

1.2 Motivation and Significance

The motivation for undertaking the project lies principally in both the increasing number of applications which rely upon semantic segmentation and the increasing need to be able to perform such fine-grained segmentation without large training datasets and with the ability to generalise to new classes.

In the real world, there are a large number of applications where there is: (1) no access to a large number of training examples for a new object class or where such examples would require significant time to collect; (2) fine-grained semantic labelling for training is significantly time consuming to annotate for a large number of training examples (as opposed to a classification task) and is potentially prohibitively expensive; or (3) the number of object classes is not fixed, where many classes of interest simply do not have sufficient data and constitute a heavy tail for the image distribution in the real world. These attributes result in traditional deep learning methods not performing well and therefore the need for a few-shot learning approach.

One particular application where this is particularly relevant is in the area of video surveillance and facial recognition. Given the size of the population, there is an unknown number of classes (i.e., individual people) and the number of training images for a particular individual will be potentially fairly small. Therefore, the field requires few-shot segmentation to be able to handle such a task. The benefits are particularly clear within policing for example, where a small number of support images of a potential suspect can be used to identify whether such individual was involved in a crime.

Another area which benefits from few-shot segmentation is medical image processing for diagnosis and treatment. There are a number of areas where there are limited sized medical datasets which have been built up to date, given the fairly recent application within medical imaging, or also for cases of rare diagnoses. In these areas few shot segmentation is particularly important and the benefits of such techniques are significant as doctors can make more precise diagnoses.

Thus, this project will aim to test whether the provision of part information of an object can help improve the accuracy of object segmentation in a few-shot context and thereby potentially benefit object segmentation within these practical settings.

1.3 Research Aims and Objectives

The project seeks to implement and evaluate deep metric learning models in a few shot learning context such that the models will aim to segment images into object classes having only received minimal support examples from previously unseen object classes.

The project will seek to explore two principal types of segmentation: supervised and multi-dimensional supervised semantic segmentation. Traditional supervised semantic segmentation consists of segmenting an image such that each pixel is associated with a specific object class, with the model trained in a supervised manner with ground truth labelled training data. By way of example an individual pixel would have pixel labels referring to the object (e.g., 0 for "background", 1 for "aeroplane" and 2 for "bicycle" respectively).

In comparison for multi-dimensional supervised semantic segmentation, the model will still aim to segment the image such that each pixel is associated with a specific object class, however it will be trained using multi-dimensional (object part) ground truth pixel annotations for the respective object classes in addition to the ground truth object class pixel annotations. For the multi-dimension annotation, each pixel of the ground truth part class pixel annotations is associated with the part label corresponding to the different parts of the objects. By way of example an individual pixel would have pixel labels referring to the constituent part of the object (e.g., 1 for "body", 2 for "wing" for the "aeroplane" object class, 3 for "wheels", 4 for "body" for the "bicycle" object class respectively).

There exists a large number of different state-of-the art FCN architectures given the rapid development of the field over the past few years [38], in addition to varying strategies for addressing some of the weaknesses (integrating long-range dependencies and global contextual information [13]) of FCNs. Equally within deep metric learning there exist a number of different objectives functions, training strategies and prediction methodologies [32]. This project seeks to additionally evaluate segmentation performance in implementing alternative formulations of some of the key parameters from the base experimental setup.

Therefore, there are a number of key objectives that the project seeks to achieve:

- Implement a few-shot Deep Metric Learning Model for traditional supervised semantic segmentation;
- Implement a few-shot Deep Metric Learning Model for multi-dimensional supervised semantic segmentation;
- Test the performance of traditional supervised semantic segmentation using the standard few-shot learning image segmentation benchmarks (e.g., one-shot, five-shot);
- Test the performance for multi-dimensional supervised semantic segmentation in a few-shot context;
- Evaluate the impact of varying design choices (e.g., prediction methodology, loss functional choices, FCN backbone architecture);

1.4 Organisation

This paper will introduce the project along multiple lines and is organised as follows:

Chapter 2 (Background) provides a critical review of the relevant literature about the key thematic areas pertaining to semantic segmentation utilising deep metric learning models in a few-shot context including how these are applied to the project.

Chapter 3 (Objectives and Specification:) provides a detailed outline of the problem specification and the detailed requirements to achieve aims and objectives of the project, including the issues to be addressed.

Chapter 4 (Design and Implementation:): describes the design and implementation of the few-shot deep metric learning models and application of the background methodology in detail.

Chapter 5 (Results and Evaluation) evaluates and compares the performance of image segmentation produced by semantic segmentation and multi-dimensional segmentation using the standard image segmentation benchmarks. Additionally, the results from altering key model parameters of the base experimental design are evaluated.

Chapter 6 (Legal, Social, Ethical and Professional Issues) outlines the legal, social ethical and professional issues within the context of the project and the procedures taken during design and implementation to account for the respective issues.

Chapter 7 (Conclusion) concludes on the achievement of objectives of the project and furthermore, outlines potential areas of further studies.

Chapter 2

Background

2.1 Overview

This literature review examines the past work pertaining to deep metric learning models in a few shot context.

The chapter is divided into three main sections corresponding to the key thematic areas of the project. The first section of the literature review is concerned with image segmentation in a few-shot context including alternative training and testing formulation as compared to traditional deep learning image segmentation. The first section will also examine the implementation choices of few-shot learning and performance of differing approaches. The second section is concerned with fully convolutional networks with the purpose of outlining popular architectures which will act as the backbone to the deep metric learning model and examines strategies to address the inherent feature downsampling deficiency in these networks. The third section of the literature review is concerned with deep metric learning in the context of semantic segmentation and varying functional choices for these models in literature. It will also examine some of the objective functions used previously in a deep metric learning context.

2.2 Few Shot Learning

Few shot learning and the problem of learning from limited training examples has been a key area of focus in the field computer vision. This is particularly the case given the rise of deep neural networks within the field and some of the inherent shortfalls that are observed including the typical requirement for large datasets to train on, overfitting training data or not generalising well to new object classes. This is in contrast to humans' ability to generalise and learn new objects after having only been given limited examples and even only one example at the extreme. Few shot learning seeks to emulate this generalisability and capability of learning from limited data [34].

The approach to few-shot segmentation which fine tunes a pre-trained network to recognise an unseen object class by using a limited number of support images from the unseen object class has not seen good performance given the tendency for these networks to overfit the previous training data, even with regularization techniques or data augmentation given the small training set [34].

The space has therefore focused on a meta-learning approach where the model seeks to learn how to generalise the few-shot task itself rather than learning specific object classes - given the limited data. Therefore, for semantic segmentation the model learns to segment images given a set of training tasks (using object classes separated for training) and evaluates the model using a set of test tasks (using unseen object classes). The model learns the segmentation task itself and uses one set of segmentation problems to help solve other unrelated sets. For semantic segmentation, the few shot problem is typically formulated as a C-way, K-shot task, where c represents the number of object classes and k represents the number of support images (with associated ground truth pixel-level annotations) associated with each object class to segment the query images [26]. The meta-learning approach is typically adopted by structuring the training of the network into episodes with each comprising a support set of k training images per object class and a query image. Figure 1 outlines this few-shot meta learning approach to training and testing and Figure 2 outlines the aim of the segmentation process.

Figure 2.1: 2-Way, 2-Shot Experiment Training and Testing

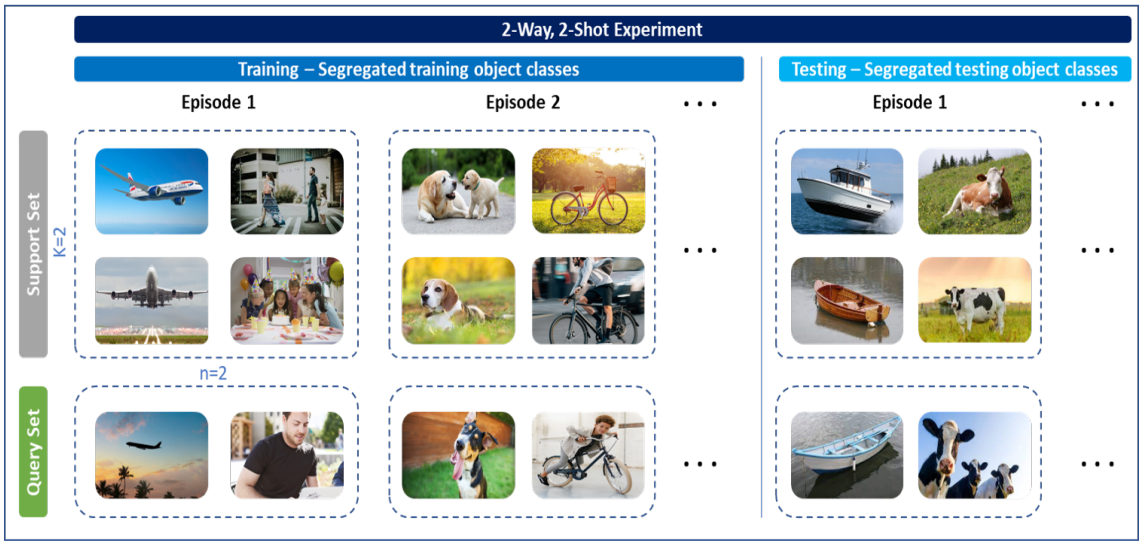
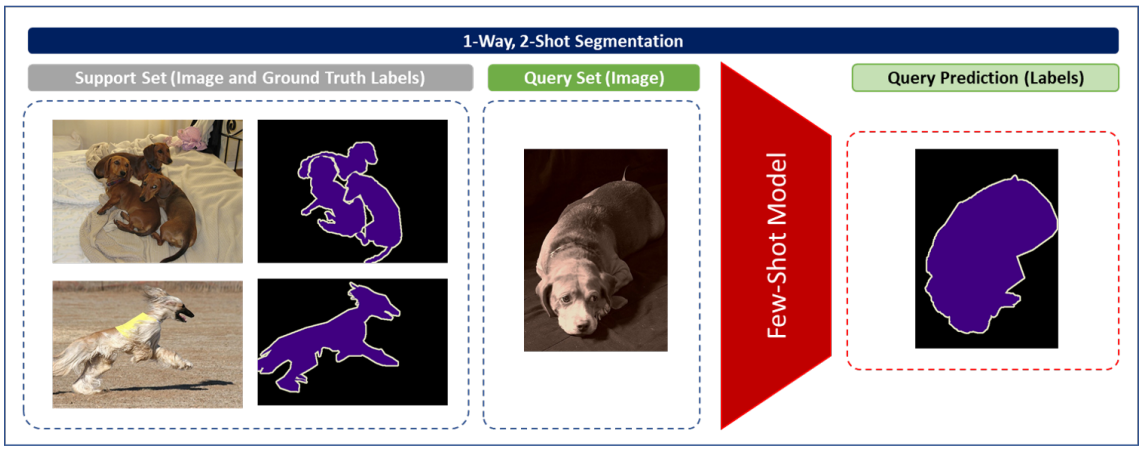


Figure 2.2: 1-Way, 2-Shot Segmentation Objective



Vinayls et al. [41] first outlined this meta-learning training approach in the Matching Nets

architecture for few-shot classification. Support and query images were input into the deep neural network to generate features and weighted nearest neighbour matching was performed to classify query images, however this was in relation to image classification. Snell et al. [36] proposed a Prototypical Network to represent each object class with one feature vector (prototype) for image classification, establishing the prototypical framework.

With more relevance for semantic segmentation, Shaban et al. [34] proposed a model for few-shot semantic segmentation using a conditioning branch to generate a vector of parameters from the support set and these parameters are used in the segmentation branch to perform segmentation of query image features using logistic regression. The paper also introduced the few-shot PASCAL-5^i benchmark semantic segmentation dataset which splits the PASCAL VOC 2012 dataset [11] into 4 folds (each with 5 object classes) with evaluation based on 3 folds for training and 1 unseen fold for testing. This benchmark dataset has been used in numerous few-shot papers subsequently for model comparisons and therefore used as the base dataset for this project.

As opposed to the parametric fine-tuning approach, Dong et Xing [26] applied a meta-learning and prototypical framework. They proposed a two branch fully convolutional network with shared weights. The first branch is a prototype learner which takes the support set and outputs the prototypes (comprising mean embedding vectors of each class) and the segmentation branch which takes the query image and prototypes and outputs the segmentation mask. However, the model involved three stages and complex configurations for training.

This prototypical approach was simplified and advanced with PANet [42] which sought to learn class feature prototypes from a few support images, having passed the support images through the network. The query image is passed through the network to generate query features and segmentation is undertaken through matching query features to learned prototypes features of object classes. A similarity matrix (using cosine similarity) from each query feature to support prototypes is generated and the prediction of an object class for a query feature is based on choosing the class that maximise the cosine similarity in the similarity matrix. Similar to other few-shot models, this prediction which utilises a distance metric in a learned embedding space lies in the deep metric learning paradigm which is elaborated on in section 2.4. However, in contrast to other deep metric learning loss functions and that used in this project which are based directly on the distance metric between support and query features (without any prediction), PANet's loss function is based on applying cross-entropy loss to the predicted segmented image (using the distance metric for prediction). Therefore, while PANet utilises a distance metric for prediction, the learned embedding space is driven by categorical loss of the prediction compared to the ground truth labels. The model also introduces further regularization during training by performing the prototype learning from the query image to the support set.

PANet presents the closest few-shot model to that used in this project with the key difference being the use of a deep metric learning loss function. Given the use of a distance metric in both PANet and this model, this project additionally compares the efficacy of the resulting embedding space by using a direct deep metric learning loss function or an indirect metric categorical loss function.

Employing a similar approach in a one-way context, the SG-One model [45] extracts a representative vector of the respective object class through masked average pooling of the

support features and generates a similarity matrix ("guidance map") using cosine similarity to each of the query features. The query features are element-wise multiplied by the guidance map to emphasise the "similar" query features and dampen the "dis-similar" query features. The fused query features and guidance map are further processed by two convolutional layers to output a class prediction which is employed in the cross-entropy loss function to optimise the network.

In the context of the multi-dimensional segmentation, it is also important to note the recently published PPNet [23] which is a part-aware prototype network for few-shot semantic segmentation. The model draws on SG-One and PANet in its prototypical approach, however rather than segmenting based on object prototypes it segments objects based on part-aware prototypes and maps the parts back to objects. These prototypes are referred to as part-aware prototypes as they are identified using K-means clustering rather than using actual part information. Similar to PANet, a distance matrix using cosine similarity is generated to the part-aware prototypes per class and max pooling is applied to generate an object class prediction which is employed with cross-entropy loss.

For the multi-dimensional task in the project, the base experiment setup uses part information in addition to object information in the deep metric loss function and prediction (i.e., segmentation of the image) utilises the distance to the support object prototypes. However similar to PPNet, this project also considers using only part information to segment the image by using the distance to part-prototypes and mapping back to the relevant objects for object segmentation.

2.3 Fully Convolutional Neural Networks

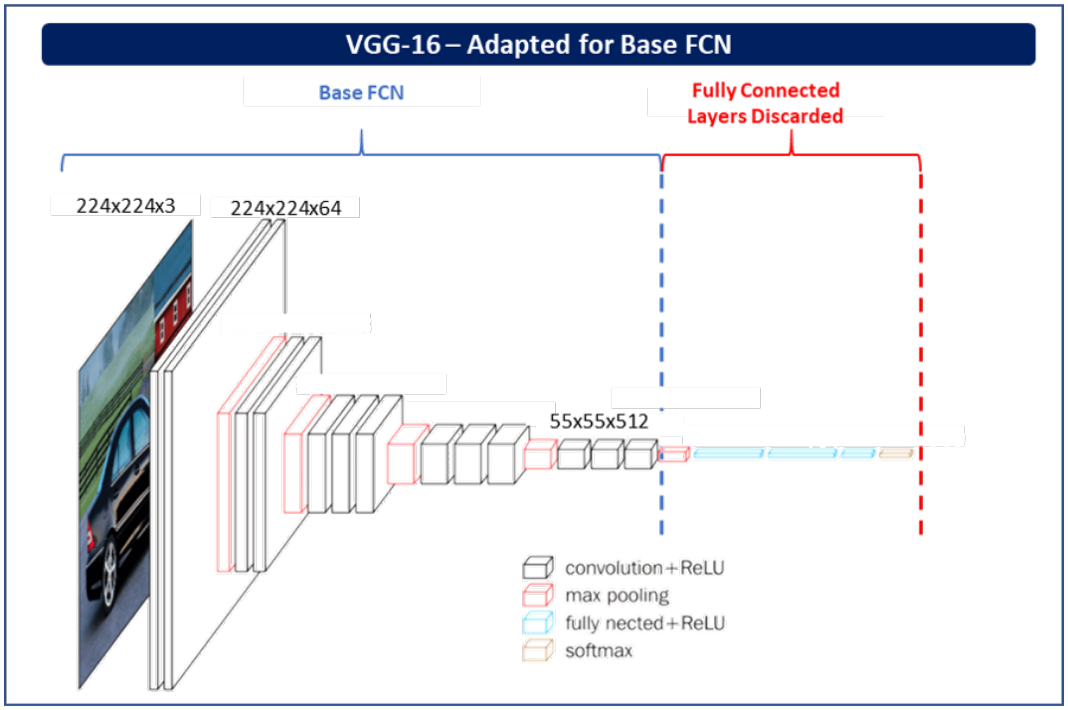
A fully convolutional neural network will act as the backbone architecture for the deep metric learning model in the project. FCN models have emerged as a key deep learning architecture for semantic segmentation, delivering state-of-the art performance. Whilst convolutional neural networks have been around for a significant period of time [38][24] within the field of computer vision usurping traditional handcrafted algorithms (SIFT, HOG etc.), FCNs were first proposed by Long et al.[19] in their seminal 2016 paper.

FCN models utilise the traditional CNN architecture, however the fully connected layers are removed to be replaced by a 1x1 convolution with the same channel dimensions as the number of object classes. This thereby turns the CNN from a whole image classification task into a pixel-wise classification task by learning a probability segmentation map or embedding vector for each pixel. This ability to learn such an embedding space for each pixel is the reason why the FCN architecture is ideally suited to act as the base feature extractor for a deep metric learning model. In contrast for a deep metric learning model, the learned embedding space is based around a distance metric (rather than as a classification maximizer), so the final layer of the FCN in a deep metric model typically has a high dimension to provide a more expressive embedding (e.g., 512) (rather than the number of object classes).

A number of these FCN networks use base CNN architectures then remove the fully connected layers, with some of the most commonly used being VGG [35] and ResNet [15] variants. This project uses the first 13 convolutional and pooling layers of VGG-16 (fully connected layers removed) as a base feature extractor for support and query images which generates

downsampled feature maps (as described in the paragraphs below) with 512 dimensions which is shown in figure 2.3. The project also considers the impact of using ResNet50 as an alternative architecture.

Figure 2.3: Base FCN Architecture



The FCNs consist of stacking layers of three types; convolutional, non-linear activation or pooling layers, where the layers are locally-connected and receive weighed inputs from a local neighbourhood (size of kernel) known as the receptive field [24]. An inherent downside of the FCN and CNN structure more broadly is their more limited ability to integrate global contextual information, which is important to be able to solve ambiguity from features at the local level. This is due to the fact the receptive field can only grow linearly with the number of layers and the pooling layers which help deliver spatial invariance and reduce computational cost also resulting in downsampling of dimensions [13]. This downsampling and resultant smaller feature map delivers less fine-grained and coarse segmentation results when upsampled naively through simple bilinear interpolation. The developed FCN architectures in research have been in part to address this downsampling problem and introducing global context to achieve higher accuracy segmentation.

There are a number of methods that have been developed to address the issue. In their original FCN paper, Long et al. [19] utilised a skip architecture to fuse the predictions of shallower layers with later layers. This combines the finer, more local features of shallower layers with the coarser more global layers thereby introducing global context. Parsenet [22] has built on this approach by using global average pooling to extract global information and combining this with feature maps from later layers (having been through L2 normalization) achieving near state-of-the-art performance on PASCAL VOC 2012.

In contrast, Dilatednet [44] and DeepLab (and its successive versions) [6] utilise a different approach to increase the receptive field without losing resolution by utilising dilated/atrous

convolution. This method utilises a dilated kernel to expand the receptive field which will increase in exponential manner through layers rather than linearly. The DeepLab models also integrate a fully connected pairwise Conditional Random Field (CRF) to further refine the pixel-wise prediction by incorporating global context. The CRF models long-range dependencies between pixels and recover linkages lost due to the spatial invariance of FCNs.

An alternative approach has been to apply deconvolutional or unpooling layers to upsample the feature maps back to the images original dimensions. Deconvnet [28], SegNet [2] and U-Net [31] employ this approach by adding an opposite deconvolutional network to the original convolutional network and effectively learn the upsampling for finer segmentation.

The base models used in the project utilises bi-linear interpolation to upsample the feature maps output from the network to reduce the impact of architectural choices.

2.4 Deep Metric Learning

The field of metric learning has been a significant area of research [22] and seeks to directly learn a distance function which is fine-tuned to a particular task. Within a classification context, a distance metric is sought within higher-dimensional feature space such that objects of the same class are close together and objects of different classes are further away. The approach is typically combined with nearest-neighbour methods, clustering or other distance or similarity approaches for the final classification (i.e., prediction methodology) once the embedding space has been learned. The metric learning approach has a number of advantages particularly in relation to robustness to overfitting by learning a distance metric, making it more suitable for a few shot context [4]. The field of deep metric learning has evolved following the advent of deep learning methods. In contrast to learning a specifically handcrafted distance metric for a particular task as with metric learning, in deep metric learning a neural network architecture is used to directly learn an embedding space by optimizing over a loss function which is built around some distance metric (e.g., euclidean or cosine for example). Whilst much of the literature within deep metric learning in computer vision has focused on its application within image classification [39] [10], it has started being used in a semantic segmentation context [1] [12] [8].

Three key elements which typically specify a DML model and highly influence its performance are the objective/loss function, sampling strategy of inputs to the loss function and prediction strategy.

There are a number of loss functions that have been developed for deep metric learning, albeit most of these loss functions have been developed in a classification context. The deep metric learning loss functions are based on a distance metric between the generated embeddings with the aim that the embeddings of the same class are close together and embeddings of a different class are further away. Unlike other typical image segmentation loss functions (e.g., categorical loss) there is no actual prediction of the relevant classes required, only the distance between features is used for the loss. One of the earliest loss functions is the contrastive loss function [39] which seeks to minimize the distance of similar objects and maximise that of dissimilar objects and has been previously implemented through a Siamese network with shared weights - a sample image input into each network with the distance maximised or minimised depending on whether images are the same object class or not respectively. Inspired by this

approach the triplet loss function [17] was developed which takes an anchor sample, a sample of the same class and a sample of a different class thereby combining both the inter-class distance and intra-class distance in one function. Further loss functions have been developed including quadruple loss, structured loss, N-pair loss, magnet loss, clustering loss, mixed loss [32]. Kevin Musgrave et al. [25] have built an open-source library, PyTorch Metric Learning, to implement a number of published loss functions and sampling strategies.

However, it should be noted that the metric learning libraries have been built primarily for classification tasks, therefore a number of the standardised loss functions from the library are not computationally designed for semantic segmentation and need to be implemented manually. Therefore, this project draws on a significantly smaller field of available loss functions designed for semantic segmentation and which require bespoke implementation based on their specification in research publications. Additionally, this project further looks at adapting the loss function to take into account multi-dimensional (part) information which, to the authors knowledge, has not been explored in literature within the deep metric learning space. In this project, the discriminative loss function developed by Brabandere et al. [5] has been utilised which was designed for semantic segmentation and based on large margin nearest neighbour classification [43]. The loss function and its implementation for semantic segmentation and multi-dimensional segmentation are discussed in detail in Chapter 4.

In a semantic segmentation context, the sampling strategy is less relevant compared to classification tasks given the high number samples (i.e., pixels rather than whole images), but it will still have an influence on training and memory requirements, therefore must be considered. This project utilises all pixels in the feature map from the support images to generate the prototype vectors for each object class.

Having learned an embedding space, there exist a number of prediction strategies in DML models to generate the actual predicted segmentation from the feature output of the model. The K-Nearest Neighbour approach has been employed successfully in a supervised context treating segmentation as a pixel-wise retrieval problem [8] and lends itself to few-shot learning [26]. Alternatives, particularly in the unsupervised or instance segmentation space use clustering methods or more complex masking methods [12]. Given the prototypical nature of the loss function used in the project, for the base prediction methodology the euclidean distance from each of the query features to the different support prototype vectors for each object class is used to determine the predicted object class (i.e., the class with the minimum euclidean distance is chosen).

There has been limited relevant research on utilising part information for object segmentation in a deep metric learning context, except for broader research on individual part segmentation [21]. This project therefore builds on the deep metric learning loss function to incorporate the parts information and this is fully outlined in Chapter 4.

Chapter 3

Objectives & Specification

3.1 Problem Overview and Objectives

3.1.1 Problem Overview:

The project seeks to implement and evaluate deep metric learning models in a few shot learning context such that the models will aim to segment images into object classes having only received minimal support examples from previously unseen object classes. The project will seek to compare two principal types of segmentation, supervised and multi-dimensional. In supervised segmentation, the few-shot DML model is trained using ground truth labelling of object classes (e.g., person, dog, cat). In multi-dimensional segmentation, the few-shot DML model is trained using ground truth labelling of both object classes (e.g., dog) and their respective part classes (e.g., head, legs, tail). The resulting models will aim to segment a query image into the respective object classes using the support images. The main contribution for this project is that it will evaluate whether the provision of multi-dimensional part information can aid the model in coping with diverse appearance in objects with different parts, poses or subcategories, and therefore result in better accuracy for overall object segmentation in a DML few-shot context.

3.1.2 Problem Definition:

The object classes, C, are split into two non-overlapping sets C_{train} and C_{test}, with the DML model trained on C_{train} and evaluated on C_{test}. Under the few-shot meta-learning framework each iteration of training and testing is defined as an "episode". In a C-way, K-shot experiment, an episode, i, consists of:

1. Chosen Object Classes: randomly sampling c object classes, $c_i \in C_{train}$ or C_{test} (depending on whether training or testing respectively);

2. Query Set: randomly sampling q query images and respective ground truth object-label pairs, $Q_i = \{(I_1^c, L_1^c), ..., (I_q^c, L_q^c)\}$, from c_i (the chosen c-way object classes);

3. Support Set: randomly sampling a support set of k images and object-label pairs (k-shot) for each of the chosen c-way object classes, $S_i = \{S_i^1, S_i^2, ..., S_i^c\}$ where $S_i^c =$

$\{(I_1^c, L_1^c), ..., (I_k^c, L_k^c)\}$

The object labels consist of binary masks for each of the c-classes and the background classes, therefore have dimension c+1.

Training consists of N_{train} episodes, $\{(S_i, Q_i)\}_{i=1}^{N_{train}}$, where the model generates support features, S_i^f, and query features, Q_i^f, with a forward pass through the network and the resultant distance metrics between the two sets of features for each of the object classes is used in the DML loss function, $Loss_{DML} = f(d(S_i^f, Q_i^f))$. Stochastic Gradient Descent is used to optimise the models parameters during the N_{train} episodes. As each episode contains randomly selected classes from C_{train} the model is trained to generalise and learn the task of segmentation using a support set rather than individual classes.

Testing consists of N_{test} episodes, $\{(S_i, Q_i)\}_{i=1}^{N_{test}}$, where the post-training DML model is used to generate support features and query features. A prediction strategy is employed to generate a prediction of the object class of the query features. The base prediction strategy in this project is the euclidean distance from the query features to a support prototype from the object classes. This prediction of the object class of the query features or each of the q query images, L_q^{pred}, is compared to the ground truth query labels, L_q^c, and this is done for each of the N_{test} episodes and the results evaluated based on benchmark metrics outlined in the methodology section.

In the case of multi-dimensional segmentation, in addition the provision of object class labels, L^c, the support set and query set consist of an image, object-label and part-label triplets, $\{(I^c, L^c, P^c)\}$. The part labels consist of binary masks for each of the multi-dimensional parts for each of the c-classes and the background class. Given the number of parts is variable for each object class, the dimensions of the part labels vary each episode. During training the features of each of the part classes (i.e., output features for pixels of a certain part) are used in addition to those from object classes in the DML loss function.

3.1.3 Additional Analysis:

Beyond the main comparison of the two types of segmentation, the project also evaluates the impact of a number of key parameters within the models.

1. The first parameter to be assessed is the use of part prototypes instead of object prototypes in the prediction methodology to understand whether segmenting for parts and mapping the resulting prediction to objects performs better than using an object prototype.

2. The second parameter to be assessed is the use of a metric learning based categorical loss function compared to the deep metric learning discriminative loss function to understand whether the learned distance based-embedding space based on categorical prediction is more effective than distance alone.

3. The third parameter to be assessed is the use of ResNet50 backbone architecture as compared to VGG-16 to understand the impact on performance and the embedding space.

3.2 Requirements

3.2.1 Functional Requirements:

The following functional requirements outline the core functionality that the few-shot deep metric learning models must have at an abstract level. The models must:

- Import images from an external dataset;
- Process images to standardised dimensions that the network is trained on;
- Utilise a few-shot data loader to input a support and query set per episode to the model;
- Functionality to amend n-way and k-shot parameters;
- Load a base network architecture and have flexibility to alter the architecture;
- Implement deep metric learning loss function adapted for semantic segmentation;
- Loss function to incorporate multi-dimensional parts information;
- Predict object classes based on features using a metric learning prediction strategy;
- Able to evaluate the performance of the model using image segmentation benchmarks;

3.2.2 Non-Functional Requirements:

Whilst adhering to the functional requirements listed above, the system must also meet several non-functional requirements which detail the overall qualities it must possess. These include:

- Suitable design setup for training and testing;
- Easily amendable hyper parameters and model configuration;
- Well documented code and functions in-line with PEP 8 conventions;
- Optimised loss functions and memory management within model;
- Suitable checkpoints and saving functionality for configurations, models and results;

3.3 Technical Specifications

The few-shot deep metric learning models for the project have been written in Python 3.8. Python was chosen given its concise, readable code and its open-source software being supported by high-quality documentation and a large and active community of developers. From a deep learning standpoint, the language benefits from an abundance of machine learning and associated libraries and frameworks. Additionally, a significant amount of code made available concurrently with published research in the field has been implemented in python, thereby aligning with common practice and facilitating experiments from previous research in developing the project.

The deep learning library for implementing the few-shot deep metric learning models for the project was PyTorch 1.7. Whilst not the leading framework by adoption more broadly,

PyTorch is significantly used in the research community for deep learning given its flexibility when designing bespoke neural networks. This is particularly relevant for the project given the deep metric learning context requiring non-standard loss functions with further tailoring for multi-dimensional parts segmentation and few-shot context.

The models were implemented on a Tesla P100 16GB graphics processing unit. Unlike traditional image segmentation which requires only one image to be passed through the model per iteration (that can be combined into batches), few-shot semantic segmentation requires multiple images forming the support and query set to be passed through the model and computationally intensive loss functions per episode thereby significantly increasing the memory requirements. This results in the need to utilise a GPU to train the models within a reasonable timeframe.

The key specifications are set out below:

1. Programming Language: Python 3.8
2. Deep Learning Framework: PyTorch
3. Key Python Libraries: Cuda, Numpy, Pandas, MatplotLib
4. Hardware specifications: Single core hyper threaded Xeon Processor @2.3Ghz, 12GB RAM, Tesla P100 16GB GPU

Chapter 4

Design & Implementation

This chapter outlines the design and implementation of the few-shot DML models for both supervised multi-dimensional segmentation. The chapter outlines the key design choices in achieving the objectives of the project with reference to the relevant background literature outlined in Chapter 2. Additionally, the chapter details the implementation of the design and key aspects of the respective code.

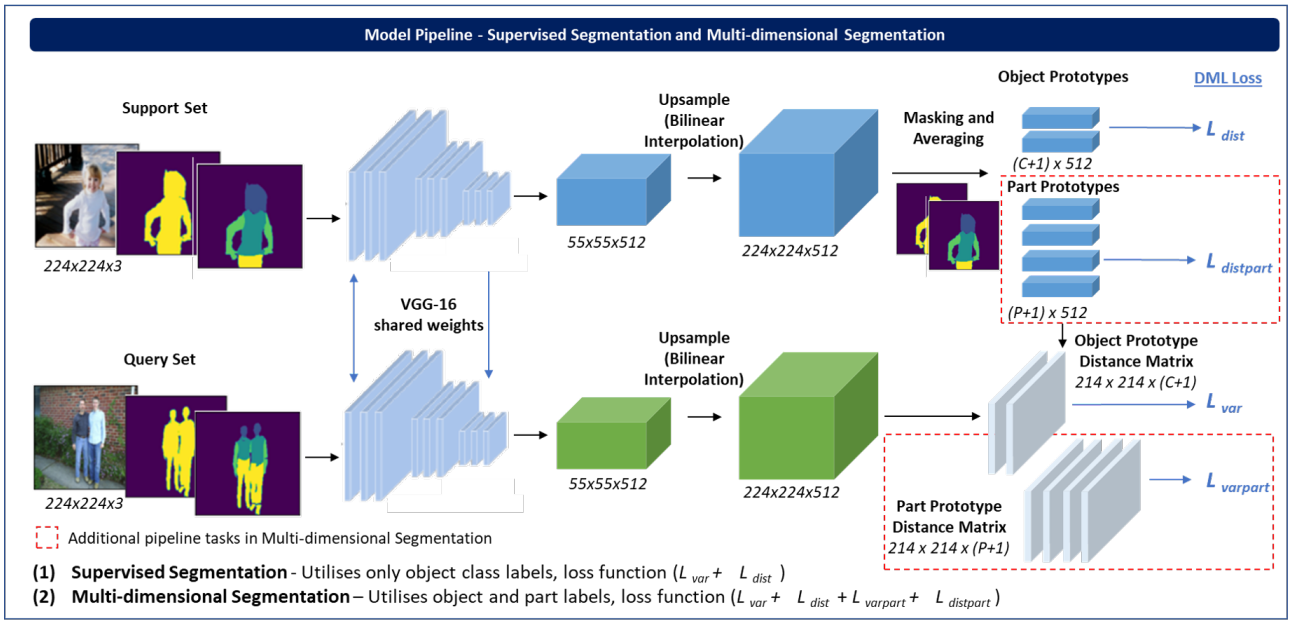
The chapter is split into eight sub-sections with each sub-section presenting the design and implementation of a key aspect of the few-shot DML models. The sub-sections cover the following areas:

1. Model Overview
2. Dataset
3. Few-Shot Dataloading
4. Network Architecture
5. Loss functions
6. Prediction Methodology
7. Evaluation
8. Code Structure

4.1 Model Overview

An overview of the model pipeline is presented in the figure and description below. The model pipeline highlights the training procedure for both supervised segmentation and multi-dimensional segmentation.

Figure 4.1: Model Training Pipeline



The input to the models is the support set and query set, each comprising images with associated object and part ground-truth label masks. The support images and query images are passed through the backbone VGG-16 FCN to generate support and query feature maps respectively. These feature maps represent a downsampled version of the image with pixels converted into embedding vectors with dimension 512. Both feature maps are then upsampled back to the original image dimensions using bilinear interpolation. Mean embedding vectors for objects and parts (prototypes) are generated from the support feature map by using the support object and part labels to mask the feature map and averaging over all respective object and part features. The euclidean distance from support and object prototypes to all query features are used to generate object and part distance matrices, and the distances are used as the basis for calculating inter-cluster loss (L_{var} and $L_{varpart}$). The distances between support object and part prototypes are used as the basis for calculating intra-cluster loss (L_{dist} and $L_{distpart}$). The inter-cluster and intra-cluster loss for the DML loss functions are based on the distance between feature vectors and aim to push features of the same object/part class "close" together and different object/part classes "far" away over training. Once training has been completed and such an embedding space learnt, the predicted segmentation of a query image is undertaken by determining which support object prototype each query feature is closest to by its euclidean distance.

Full details of the design decisions and implementation of all the key components of the models are outlined in the following sections.

4.2 Dataset

4.2.1 Design:

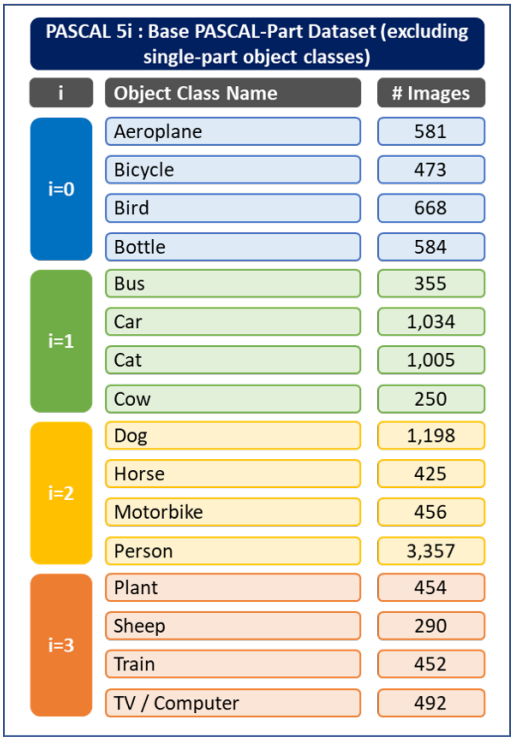
The dataset consists of images with associated object class labels and part class labels that the few-shot DML models will be trained and evaluated on. The dataset chosen for the project is the PASCAL-5^i [34] structure using base images and segmentations from PASCAL-Part [7] dataset.

The decision was taken to use the PASCAL-5^i structure as within few-shot semantic segmentation publications the dataset has become established as one of the benchmark datasets. Therefore, there exists comparable performance metrics from few-shot semantic segmentation models in the space that this project can reasonably be compared to in terms of performance. PASCAL-5^i uses base images and segmentations from the PASCAL VOC 2012 dataset [11] with SBD [14] augmentation which is a benchmark dataset in broader visual object category recognition and detection (i.e., non-few-shot) and used in the annual PASCAL VOC Challenge to evaluate state of the art image segmentation models. The dataset contains 20 object classes: aeroplane, bicycle, boat, bus, car, motorbike, train, bottle, chair, dining table, potted plant, sofa, TV/monitor, bird, cat, cow, dog, horse, sheep, and person. The dataset contains 12,032 images with each image having an associated pixel-level object segmentation annotations. To align with the few-shot framework, PASCAL-5^i splits the dataset into 4 folds with each fold containing 5 classes. For evaluation of models in the field, training is conducted on 3 folds (15 classes) and testing is conducted on 1 unseen fold (5 classes).

Given the comparison of few-shot supervised segmentation with multi-dimensional segmentation, the project requires a dataset which additionally includes annotations for individual parts of object classes rather than only object level annotations. The PASCAL-Part dataset is a set of additional annotations for PASCAL VOC 2010 which provides segmentation masks for both object classes and each of the parts of the objects. The dataset provides part level segmentation for 16 out of the 20 object classes with four classes without consistent set of parts (boat, chair, table, sofa). With the aim of providing a fair comparison between supervised segmentation and multi-dimensional segmentation, the PASCAL-Part dataset was used as the base dataset with the PASCAL-5^i structure as it contains both object-level and part-level annotations for images. The dataset contains 10,104 images with each image having an associated pixel-level object and part segmentation annotations. The decision was taken to remove the four classes which do not have part level segmentation to provide a consistent comparison and a more accurate basis to assess the impact of part information, therefore PASCAL-5^i splits the dataset into 4 folds with each fold containing 4 classes.

The below figure outlines the split of object classes and number of images in each class under the PASCAL-5^i structure:

Figure 4.2: Project Dataset



The part-level annotations contained in the PASCAL-Part dataset are highly granular with each object class typically being segmented into between 8-13 individual parts. This level of granularity for individual parts is not required and is highly detrimental to the networks ability to differentiate individual part features from unseen classes given the few-shot context based on experiments during the development of the multidimensional model. Therefore, for each of the object classes the individual granular parts have been merged to achieve between 3-4 parts per object class. This serves to provide consistency on the number of parts within the model and also not overload the discriminatory ability of the model for individual parts within the embedding space. As described above, the 4 classes (boat, chair, table, sofa) without part segmentation are removed from the dataset. The individual parts for each of the object classes are outlined below.

Figure 4.3: Project Object-Part Mapping

PASCAL-Part Amended Object-Part Mapping

	Object	Object Index	Part	Part Index
i=0	Aeroplane	1	Body	1
			Wing	2
	Bicycle	2	Wheels	3
			Body	4
	Bird	3	Body	5
			Wings	6
			Feet	7
	Bottle	5	Cap	9
			Body	10
i=1	Bus	6	Body	11
			Wheels	12
			Windows	13
	Car	7	Body	14
			Wheels	15
			Windows	16
	Cat	8	Head	17
			Body	18
			Legs	19
			Tail	20
	Cow	10	Head	22
			Body	23
			Legs	24
			Tail	25

	Object	Object Index	Part	Part Index
i=2	Dog	12	Head	27
			Body	28
			Legs	29
			Tail	30
	Horse	13	Head	31
			Body	32
			Legs	33
			Tail	34
	Motorbike	14	Wheels	35
			Body	36
	Person	15	Head	37
			Body	38
			Arm	39
			Legs	40
i=3	Plant	16	Pot	41
			Plant	42
	Sheep	17	Head	43
			Body	44
			Legs	45
			Tail	46
	Train	19	Body	48
			Windows	49
	TV / Comp.	20	Screen	50
			Periphery	51

4.2.2 Implementation:

Prior to the few-shot implementation of support and query sets, the code first needs to access individual images and associated object labels and part labels from the dataset. This has been implemented with the ”VOC” class in ”pascal_loader.py” from the ”dataloaders” folder. The VOC class inherits its data loading class properties from the standard PyTorch ”DataLoader” class. The VOC class maps indices to data samples from the PASCAL-Part dataset. The ”getitem()” protocol retrieves an individual sample from the dataset object once initialised. The index of the datasample is its position in the ”trainparts.txt” document within the ”data” folder which contains the names of all the datasamples. Each data sample will contain the name of the image, original image, normalised image converted to a tensor, object mask for the image and part mask for the image. The VOC class is used to initially load the full dataset, which is then split by class, processed to only incorporate the relevant object labels and loaded in a few-shot framework outlined in the next sub-section.

The object and part segmentation masks for each image in the PASCAL Parts dataset are contained in individual matlab source files (.mat). The project has incorporated open-source code [40] to manipulate the annotations of the PASCAL-Part dataset and load the respective object and part annotations for an image in numpy array format that will subsequently be converted to a tensor for the model. Each object and part have been given an associated label with the mapping specified in ”part2obj.py”. The ”_mat2map” function creates the object class mask which is an array the same size of the image where each pixel is labelled between 0-20 for its respective object class (0 representing background) and likewise the part class mask generates an array with labels between 0-51 for each of the possible parts amongst all the objects.

Sample code for the VOC class and the _mat2map function is outlined below:

Figure 4.4: getitem() protocol in VOC class loads datasample from PASCAL-Part dataset

```
def __getitem__(self, idx):
    # Fetch data
    id_ = self.ids[idx]
    image = Image.open(os.path.join(self._image_dir, f'{id_}.jpg'))
    imsize = np.array(image).shape
    data = loadmat(os.path.join(self._label_dir,
                                f'{id_}.mat'))['anno'][0, 0]
    n_objects = data['objects'].shape[1]

    objects = []
    for obj in data['objects'][0, :]:
        objects.append(PascalObject(obj))

    cls_labels, inst_labels, part_labels = self._mat2map(imsize,objects)

    sample = {'id': id_,
              'image': image,
              'cls_labels':cls_labels,
              'part_labels':part_labels}
```

Figure 4.5: Creates object and part masks from .mat files

```
def _mat2map(self,imsize,objects):
    # Create masks from .mat annotations in numpy format
    shape = imsize[:-1]  # first two dimensions, ignore color channel
    cls_mask = np.zeros(shape, dtype=np.uint8)
    inst_mask = np.zeros(shape, dtype=np.uint8)
    part_mask = np.zeros(shape, dtype=np.uint8)
    for i, obj in enumerate(objects):
        class_ind = obj.class_ind
        mask = obj.mask

        inst_mask[mask > 0] = i + 1
        cls_mask[mask > 0] = class_ind

        if obj.n_parts > 0:
            for p in obj.parts:
                part_name = p.part_name
                pid = PIMAP[class_ind][part_name]
                part_mask[p.mask > 0] = pid

        # For classes without parts the whole object is segmented
        if class_ind in [4,9,11,18]:
            part_mask[mask > 0] = PIMAP[class_ind]['main']

    return cls_mask, inst_mask, part_mask
```

4.3 Few-Shot Dataloading

4.3.1 Design:

The few-shot dataloading functionality is such that for each episode in a C-way, K-shot experiment the model receives as input a support set of k images with associated object and part labels for each of the chosen c-way object classes and a query set with q images with associated object and part labels chosen from the c-way object classes. Therefore, the support set will contain (c x k) image, object label, part label triplets and the query set will contain q image, object, part label triplets randomly chosen from the c-classes in an episode.

The dataloading functionality has been designed such that all images and labels in the support set and query set are converted to the PyTorch tensor data structure for computational efficiency given vectorisation of PyTorch algorithms and ability to utilising gpu capabilities. Additionally, there are a number of image and object-part label processing steps that are required prior to being an input for the model. The first pre-processing procedure is to resize all images and labels in the support and query to standardised dimensions of 224 x 224 pixels and normalized using standard VGG-16 means (0.485, 0.456, 0.406) and standard deviations (0.229, 0.224, 0.225). Whilst the FCN backbone architecture and few-shot algorithm can accept any size image (given it does not have any fully connected layers), the base model of VGG-16 has been pre-trained on ImageNet with images of these dimensions and normalized therefore retaining consistency should benefit the starting point for feature extraction. The second pre-processing step is to amend the support and query object and part labels such that only the object classes (and associated part classes) selected in an episode are contained in the labels, as certain images contain multiple object classes. This ensures the labelling for an episode only contains the relevant classes and the model is trained or tested on the correct ground-truth annotations.

The dataloading functionality is designed such that the input to the model in each episode will contain the following items:

1. Class ids of chosen object classes contained in episode (i.e., c randomly selected from the respective train and test classes)

2. Part ids of all parts from all the chosen object classes contained in the episode.

3. Support images - Each of the images in the support set. Tensor with dimensions (c x k x 3 x 224 x 224) where the 3 dimensions represent RGB

4. Support object labels - For each support image, binary masks for each of the object classes in the episode and background. Tensor with dimensions (c x k x 224 x 224 x c+1) where the c+1 dimensions represent binary masks for each of the classes and the background

5. Support part labels - For each support image, binary masks for each of the part classes in the episode and background. Tensor with dimensions (c x k x 224 x 224 x p+1) where the p+1 dimensions represent binary masks for each of the part classes in the episode and the background

6. Query images - Each of the images in the support set. Tensor with dimensions (q x 3 x 224 x 224)

7. Query object labels - For each query image, binary masks for each of the object classes in the episode and background. Tensor with dimensions (q x 224 x 224 x c+1)

8. Query part labels - For each query image, binary masks for each of the part classes in the episode and background. Tensor with dimensions (q x 224 x 224 x p+1)

4.3.2 Implementation:

The project utilised the few-shot code published for PANet [42] as a base and adapted it for the PASCAL-Parts dataset, parts segmentation and optimised dataloading to fully utilise the tensor structure for all objects. The main function for few-shot data loading is "voc-fewshot" function in "fewshot_loader.py" from the "dataloaders" folder. The voc-fewshot function initially loads the full dataset using the VOC class described in section 4.1. The full data set is then split into subsets corresponding to the object classes that the data samples belong to. The c-way object classes are randomly chosen for each episode and random samples from these chosen object classes are paired to form the support set and query.

For the first pre-processing procedure of resizing all images and labels to standardized dimensions the "Resize" function was used within "transforms.py" from the "dataloaders" folder. To optimise processing speed for images only PyTorch in-built functions are used. The images are resized using "torchvision.transforms.Resize" function which uses bi-linear interpolation. For resizing of the labels, given the need to preserve the values of the pixels which reference the respective class labels, the nearest neighbour interpolation method is used instead. The sample code is outlined in the figure below.

Figure 4.6: Resizes image dimensions to 224x224

```
class Resize(object):
    """
    Resize images/masks to given size

    Args:
        size: output size
    """
    def __init__(self, size):
        self.size = size

    def __call__(self, sample):
        img, cls_labels, part_labels =  sample['image'], sample['cls_labels'], sample['part_labels']

        img = tr_F.resize(img, self.size)
        cls_labels = torfunc.interpolate(torch.Tensor(cls_labels).unsqueeze(dim=0).unsqueeze(dim=0),
                                         self.size, mode='nearest').squeeze().squeeze().numpy()
        part_labels = torfunc.interpolate(torch.Tensor(part_labels).unsqueeze(dim=0).unsqueeze(dim=0),
                                         self.size, mode='nearest').squeeze().squeeze().numpy()

        sample['image'] = img
        sample['cls_labels'] = cls_labels
        sample['part_labels'] = part_labels
        return sample
```

For the second pre-processing procedure of amending the support and query object and part labels such that only the chosen object classes (and associated part classes) of that episode are shown, the "classMask" and "partMask" functions are used. The functions accept the original label array and the relevant object classes and part classes for that episode and creates a mask containing only those labels. The sample code is outlined in the figure below.

Figure 4.7: Create Object and Part Masks for episode from original labels

```
def classMask(classlabel, class_ids):

    # H x W x (C+1) array with each of the (C+1) object masks
    mask = torch.zeros_like(classlabel).unsqueeze(dim=2)
    for i in class_ids:
        classmask = torch.where(classlabel == i, 1, 0).unsqueeze(dim=2)
        mask = torch.cat([mask,classmask],dim=2)

    # HxW object mask with each object indexed to pbject ids array
    mask_flat = torch.argmax(mask,dim=2)

    # Converts the background class layer into a background mask
    mask[:,:,0] = torch.where(mask_flat == 0, 1, 0)

    return mask, mask_flat

def partMask(partlabel, part_ids):

    # H x W x (P+1) array with each of the (P+1) part masks
    mask = torch.zeros_like(partlabel).unsqueeze(dim=2)
    for i in part_ids:
        partmask = torch.where(partlabel == i, 1, 0).unsqueeze(dim=2)
        mask = torch.cat([mask,partmask],dim=2)

    # HxW part mask with each part indexed to part ids array
    mask_flat = torch.argmax(mask,dim=2)

    # Converts the background class layer into a background mask
    mask[:,:,0] = torch.where(mask_flat == 0, 1, 0)

    return mask, mask_flat
```

4.4 Network Architecture and Training Hyperparameters

4.4.1 Design:

The backbone network acts as the feature extractor taking the image tensor as an input and generating a feature map consisting of vector embeddings. The feature map's dimensions are a downsampled version of the original image dimensions and once upsampled back (using bi-linear interpolation) to the original image size each vector can be considered a feature embedding of a pixel in the original image. The aim is to adjust the weights of the network through training such that the network has learnt to encode a feature space where pixel embeddings from the same object class are close and different object classes are far away, therefore is discriminatory.

The project uses VGG-16 [35] as the base backbone FCN network architecture for the DML models. To convert VGG-16 to an FCN, only the first 13 convolutional and pooling layers are used with the final layer using a linear activation function. The model encodes a 224x224x3 image tensor into 55x55x512 feature map. VGG-16 was chosen as the base architecture given that

it is widely used a backbone for many image segmentation models. Additionally, its relatively shallower depth and complexity compared to other newer architectures reduces memory requirements given the lower number of parameters and from experiments takes a shorter amount of time to train on PASCAL-5i. The network is initialised with weights from a pre-trained network using ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [33] which is similar to other works [34] [26]. Using pre-trained weights is an example of transfer learning [18] where the network has already learnt some features prior to being trained on PASCAL-5i, and from experiments has resulted in lower training times than the standardised Kaiming He initialisation [16]. For training, the model uses stochastic gradient descent with momentum with a learning rate of 1e-06 and momentum term of 0.9. The model also applies a learning rate scheduler with gamma decay rate of 0.1 each 10,000 episodes. The number of training iterations used is 30,000 episodes.

The model additionally has the functionality to use ResNet-50[15] as an alternative network architecture. The final-two average pooling and fully connected layers are removed. The network is additionally initialised with weights from training on ImageNet. The model encodes a 224x224x3 image tensor into a 28x28x2048 feature map. ResNet-50 is another popular backbone architecture for image segmentation models and therefore provides a useful comparison for the base VGG-16 architecture.

The base upsampling procedure to resize the feature maps back to the original size of the image uses simple bi-linear interpolation. This simple upsampling procedure does however lead to less fine-grained and coarse segmentation results.

4.4.2 Implementation:

The FewShotNet class, which inherits the standard PyTorch neural network model features, in "network.py" within the "models" folder is the main network module and imports the VGG-16 or ResNet-50 architectures as the "encoder" object. The model combines all the support images and query images into a single tensor with shape (n-way * k-shot + q) x 3 x 224 x 224, effectively converting the set into a mini-batch, for input into the VGG-16 or Resnet-50 encoder. The combined support and query features are output from the model with shape (n-way * k-shot + q) x 512 x H' x W' where H' and W' refers to the downsampled dimensions of the feature maps. The support and query features are separated into individual tensors which will be the input for the loss function. The sample code for the model is outlined in the figure below.

Figure 4.8: Model importing encoder to generate support and query features

```
def forward(self, supp_imgs, qry_imgs):
    """
    Args:
        supp_imgs: support images
            way x shot x 3 x H x W
        qry_imgs: query images
            q x 3 x H x W
    """
    n_ways = supp_imgs.shape[0]
    n_shots = supp_imgs.shape[1]
    n_queries = qry_imgs.shape[0]
    img_size = supp_imgs.shape[-2:]

    # (Ways*Shot+Queries) x 3 x H x W
    imgs_concat = torch.cat([supp_imgs.view(n_ways*n_shots,3,*img_size)]
                            +[qry_imgs], dim=0)

    # Extract features
    # (Way*Shots + Q) x 512 x H' x W'
    img_fts = self.encoder(imgs_concat)
    fts_size = img_fts.shape[-2:]

    # Separate support and query features
    #W x Sh x V(512) x H' x W'
    supp_fts = img_fts[:n_ways * n_shots].view(
         n_ways, n_shots, -1, *fts_size)

    # Q x V(512) x H' x W'
    qry_fts = img_fts[n_ways * n_shots:]

    return supp_fts, qry_fts
```

4.5 Loss Functions

4.5.1 Design:

The deep metric learning loss function constitutes the main mechanism to encode a feature space where pixel embeddings from the same object class are close and different object classes are far away based on a specified distance metric. Training the network to learn the discriminatory feature encoding is achieved through minimising the loss function using stochastic gradient descent and backpropagation to adjust the weights of the network through training. In contrast to semantic image segmentation where categorical loss is regularly used (e.g., cross entropy loss between the predicted labels and ground truth labels), deep metric learning loss functions are based on the distances between feature embeddings of pixels from the same object class or different object classes. As described in the background section, many of the deep metric learning loss functions have been designed for the task of classification rather than semantic segmentation and furthermore have not been designed for a few-shot context. Additionally, to the knowledge of this author none of the published literature has considered the incorporation

of multi-dimensional parts information into a deep-metric learning loss function. Therefore, for the task of supervised semantic segmentation, a deep metric loss function which fulfils the semantic segmentation and few-shot context is implemented and this loss function has been adapted for multi-dimensional semantic segmentation.

In this project, we use an adapted version of the discriminative loss function developed by Brabandere et al. [5]. The discriminative loss function was designed specifically for semantic segmentation, based on large margin nearest neighbour classification [43], and was the first to use a loss based on distance metric learning principles for the task of semantic segmentation with deep networks. The loss function is computationally efficient and designed for semantic segmentation as it acts on the distances of an object classes pixels to the mean embedding vector of that object class (prototypes), rather than the distance matrix of all pixels which would be computationally intractable. This also aligns it closer to the prototypical approach taken in few shot learning models. The loss function consists of three terms:

$$L_{var} = \frac{1}{C} \sum_{c=1}^{C} \frac{1}{N_c} \sum_{i=1}^{N_c} [\|\mu_c - x_i\| - \delta_v]_+^2 \tag{4.1}$$

$$L_{dist} = \frac{1}{C(C-1)} \sum_{c_A=1}^{C} \sum_{c_B=1}^{C} [2\delta_d - \|\mu_{c_A} - \mu_{c_B}\|]_+^2 \tag{4.2}$$

$$L_{reg} = \frac{1}{C} \sum_{c=1}^{C} \|\mu_c\| \tag{4.3}$$

$$L = \alpha L_{var} + \beta L_{dist} + \gamma L_{reg} \tag{4.4}$$

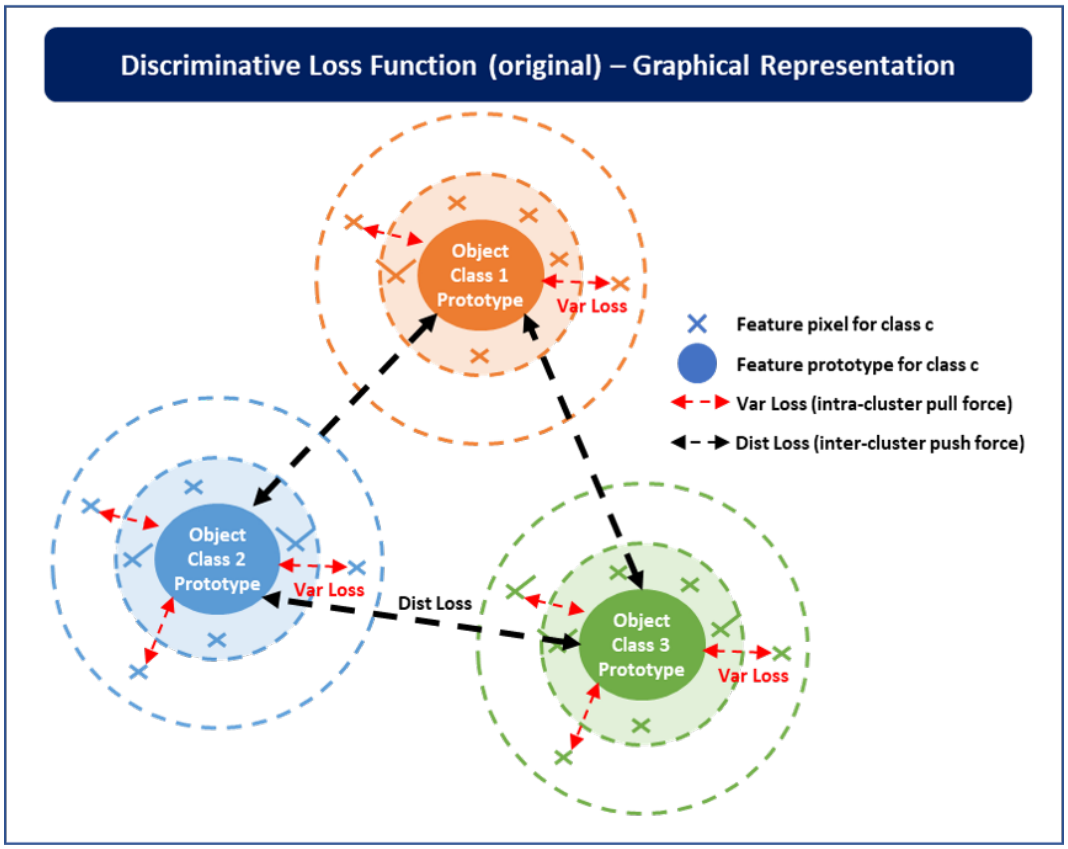
1. Variance term - Induces pixels of the same class to be close together (intra cluster pull force). The variance loss is constructed as the average distance from a pixel feature to the mean pixel feature of that object class. Therefore, the closer object classes are to the mean (i.e., closer all pixels of a class are together) the lower the variance loss.

2. Distance term - Induces pixels of different classes to be further away from each other (inter cluster push force). The distance loss is constructed as a target distance ($2\delta_d$) less the average distance between the mean pixel features of all the object classes. Therefore, the further away the mean feature vectors from each other (i.e., closer to the target distance) the lower the loss function.

3. Regularization term - Smaller induction for clusters to be nearer origin to bound distances.

The variance and distance term are hinged so they are only active to a certain distance to allow for better representation within the feature space. The hyperparameters $\alpha = \beta = 1$ and $\gamma = 0.001$, with $d_d > 2d_v$ are used in the original paper [5]. A graphical depiction of the loss function is outlined below:

Figure 4.9: Original Discriminative Loss Function



The loss function has been adapted for this project to cater for the two tasks of (1) few-shot supervised segmentation and (2) few-shot multidimensional segmentation.

(1) **Supervised Segmentation**:
For supervised segmentation, the loss function has been adapted such that only the variance and distance term from the previously outlined discriminative loss function are included. From experimentation, the inclusion of the regularization term does not have a meaningful impact on the segmentation accuracy.

Variance loss: The loss function has been adapted for the task of few-shot segmentation by explicitly associating the support features to the query features. In the variance term, the mean support feature vector for each of the c object classes in a c-way, k-shot experiment is defined as the mean of the support features for the respective object classes generated by the model (i.e., the support prototype). There will be (c+1) prototypes generated in a c-way, k-shot experiment (the additional class is the background class) and each prototype will have dimensions of 1x512. The average distance is then calculated between each of the query features and the mean support feature vector of the same object class. The variance term therefore induces the query features of an object class to be close to the mean of the support features of that object class. This is a natural target as in the prediction stage of the model, outlined in the next subsection, the base prediction methodology is to assign the object class of a query feature pixel to the object class of the support prototype that is closest. The loss function has been optimised such

that in practice it is more efficient to first calculate the euclidean distance matrix between each of the support prototypes and the query feature map, resulting in a tensor with size q x 224 x 224 x (c+1). When the argmin function is applied to the distance matrix it effectively provides the object segmentation prediction by assigning the pixel to the nearest support object class. The variance term is calculated by multiplying the distance matrix by the object-class query mask composed of binary masks for each of the classes (with the same dimensions) so for each layer only the distances for that object class to the respective support vector are retained and the average taken. In contrast to the original discriminative loss function the d_v has been set to 0 therefore all pixels of an object class regardless of distance are induced to be as close to the prototype vector of that class as possible.

Distance loss: The distance term is defined as target distance, δ_{target}, minus the average distance between the support prototypes for each class. The number of distance combinations between the support prototypes are C(C-1) combinations. This will induce an inter-cluster push force between the support prototypes and lead to a higher distance between object classes therefore more discrimination. Through experimentation, the target distance was set at 250 which results in the variance loss and distance loss being fairly similar and decreasing at similar rates.

The supervised segmentation loss function is defined below, where for the base experiments the distance function, d, refers to the euclidean distance:

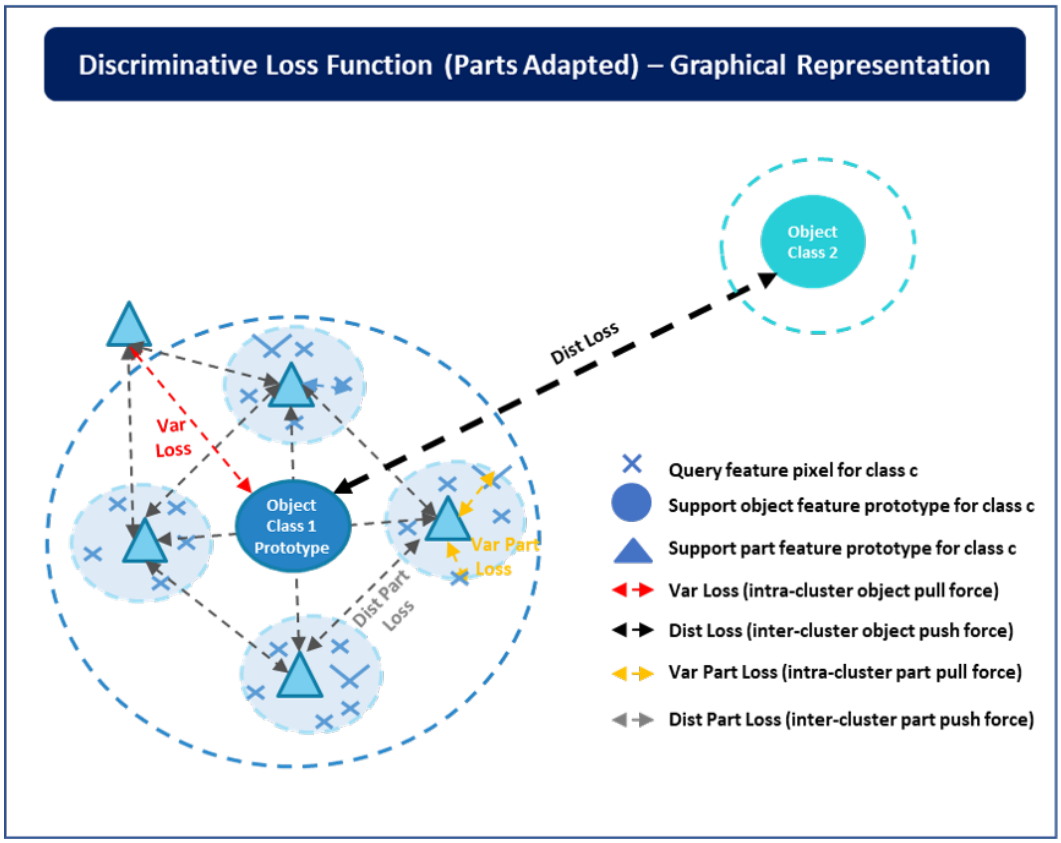
$$L_{var} = \frac{1}{C} \sum_{c=1}^{C} \frac{1}{N_c} \sum_{i=1}^{N_c} d(\mu_c^{supp}, x_i^{query}) \tag{4.5}$$

$$L_{dist} = \frac{1}{C(C-1)} \sum_{c_A=1}^{C} \sum_{c_B=1}^{C} [\delta_{target} - d(\mu_{c_A}^{supp} - \mu_{c_B}^{supp})] \tag{4.6}$$

$$L = L_{var} + L_{dist} \tag{4.7}$$

(2) **Multi-dimensional Parts Segmentation:**
The loss function for multi-dimensional parts segmentation has been built on the loss function for supervised segmentation. The aim of multi-dimensional parts segmentation is to assess whether the inclusion of parts information can improve performance and discriminative ability of object segmentation in a few-shot learning context. This is implemented through the loss function with the inclusion of parts-specific loss terms that act under the same principles of inter-cluster pull forces and intra-cluster push forces. Conceptually within the embedding space, this is achieved by creating separated feature clusters for the individual parts of an object whilst still keeping these part clusters at a close enough distance to the overall object cluster to allow segmentation. The different object clusters should still be separated such that it is possible to discriminate between them. Therefore, the loss function is balanced by separating out the individual parts of an object whilst maintaining sufficient distance between objects. This conceptual illustration of the varying forces and distances implied by the loss function are shown in the figure below:

Figure 4.10: Parts Adapted Discriminative Loss Function



This has been designed in the part loss function by adapting the variance loss and keeping the distance loss unchanged from supervised segmentation and introducing two additional terms, variance part loss and the distance part loss.

Variance Loss: The original variance loss has been adapted such that it is now defined as average distance between the support object prototype and the respective support part prototypes, which induces the part prototypes to not be too far from the object prototype. The variance loss is hinged at the target variance distance as we would like the part prototypes to not be too close together and lose discriminative ability of the individual parts.

Distance Loss: The distance loss term stays the same as supervised segmentation.

Variance part loss: The variance part loss is defined as the average distance from query features to the respective support part feature prototype that the query feature belongs to. The support part feature prototypes are defined as the mean of the feature vectors belonging to a respective part. There will be (p+1) part prototypes generated in a c-way, k-shot experiment (the additional class is the background class) with p defined as the sum of the number of parts for the c object classes. The aim of part variance term is to pull the individual query part feature vectors close to the respective support part prototype, thereby bringing the part cluster closer together.

Distance part loss: the distance part loss which is calculated as the target part distance minus the average of the intra-part cluster distance for each object classes in an episode. The intra-part cluster distance for an object class is defined as the average distance between each of the part support prototypes. This term therefore induces the individual parts for each of the

object class to be more separated and reach the target part distance. Through experimentation, the target part distance was set at 75, target variance distance set at 100 and the target distance from the distance loss term continues to be set at 250. This ensures that the distance between individual part prototypes is significantly smaller than the distance between overall object prototypes.

The multi-dimensional parts segmentation loss function is defined below:

$$L_{var} = \frac{1}{C} \sum_{c=1}^{C} \frac{1}{P_c} \sum_{i=1}^{P_c} \left[d(\mu_c^{supp}, \mu_{p,c}^{supp}) - \delta_{vartarget} \right]_+ \tag{4.8}$$

$$L_{dist} = \frac{1}{C(C-1)} \sum_{c_A=1}^{C} \sum_{c_B=1}^{C} \left[\delta_{target} - d(\mu_{c_A}^{supp} - \mu_{c_B}^{supp}) \right] \tag{4.9}$$

$$L_{varpart} = \frac{1}{C} \sum_{c=1}^{C} \frac{1}{P_c} \sum_{i=1}^{P_c} \frac{1}{N_c} \sum_{i=1}^{N_c} d(\mu_{p,c}^{supp}, x_i^{query}) \tag{4.10}$$

$$L_{partdist} = \frac{1}{C(C-1)} \sum_{c_A=1}^{C} \sum_{c_B=1}^{C} \left[\frac{1}{P(P-1)} \sum_{p_A=1}^{P_c} \sum_{p_B=1}^{P_c} \left[\delta_{parttarget} - d(\mu_{p_A}^{supp} - \mu_{p_B}^{supp}) \right] \right] \tag{4.11}$$

$$L = L_{var} + L_{dist} + L_{varpart} + L_{distpart} \tag{4.12}$$

Distance metric: Given the specification of both the supervised and the multi-dimensional parts loss functions, the remaining key aspect to fully describe the few-shot distance metric learning task is the distance metric itself. This project uses the euclidean distance as the base distance metric between the feature vectors in the embedding space. During experimentation, using the cosine similarity metric resulted in similar but slightly worse results, therefore the euclidean distance was chosen.

Cross-entropy loss comparison: The loss function code has additionally been given the functionality to calculate the cross-entropy loss of the predicted object segmentation alongside the actual DML loss function used in the model. The cross-entropy loss does not contribute to the discriminative loss function used as the base of the project and therefore is not part of the loss gradient and backpropagation. However, it provides a useful reference point during training and as expected decreases over the training horizon as the model becomes more discriminative between object classes. The cross-entropy loss is calculated by using the inverse of the euclidean distance matrix (therefore closer distances have higher values) as the logits input (i.e., raw predictions of the class) to be compared to the ground-truth labels. This project also considers the performance of using the categorical loss as the primary loss function and associated evolution of the discriminative loss function in the background.

4.5.2 Implementation:

The loss function is defined by the "DisctriminativeLoss" class in "discriminative_loss.py". The class takes the support features, query features, respective object, and part masks as inputs. The class outputs the variance loss, distance loss, variance part loss, distance part loss and distance matrix from support prototypes (i.e., the prediction after argmin function is applied). The loss function uses a number of underlying functions to perform the operations with the

main six defined below.

The first function of note is used to generate the support prototypes. The function takes support sets feature maps as an input and upsamples the feature map to the original image size using bilinear interpolation with resulting tensor of size (c-way * k-shot) x 214 x 214 x 512. The upsampled feature map is multiplied by each of the object class support masks so only the relevant features are used, and the mean vector of each object class is calculated (the prototypes). Each support object prototype has dimensions 1x512 and together all support prototypes form a single tensor with dimension (c+1)x512. The sample code is shown below.

Figure 4.11: Extracts support prototypes

```
def getPrototypes(self, supp_fts, supp_mask, n_ways, n_shots, n_partclasses):
    """
    Extract support prototypes from feature maps

    Args:
        supp_fts: input features, expect shape: way x shots x 512 x H' x W'
        supp_mask: binary mask, expect shape: way x shots x H x W x (C+1)

    """

    # Upscales feature maps through bilinear interpolation: (way x shots) x H x W x V
    supp_fts = supp_fts.view(n_ways*n_shots,supp_fts.shape[2],
                             supp_fts.shape[3],supp_fts.shape[4])
    supp_fts = F.interpolate(supp_fts, size=supp_mask.shape[2:4],
                             mode='bilinear',align_corners=False)
    supp_fts = supp_fts.permute(0,2,3,1)  #

    # Reshape maske(way x shots): x H x W x V
    supp_mask = supp_mask.view(n_ways*n_shots,supp_mask.shape[2],
                               supp_mask.shape[3],supp_mask.shape[4])

    # Applies mask to upsized feature maps for average: (way x shots) x V x (C+1)
    fts_prots = torch.stack([torch.stack([torch.sum(supp_fts[j]*
                                                    supp_mask[j][:,:,i][...,None],dim=(0, 1)) /
                                          (torch.sum(supp_mask[j][:,:,i])+ 1e-5)
          for i in range(n_partclasses)],dim=0) for j in range(n_ways*n_shots)],dim=0)
    fts_prots = fts_prots.permute(0,2,1)

    # Applies mean across Ways and Shots: V x (C+1)
    mask_b = torch.sum(supp_mask,dim=(1,2))
    mask_b = torch.where(mask_b != 0, 1, 0)
    fts_prots = torch.sum(fts_prots,dim=0) / (torch.sum(mask_b,dim=0)+ 1e-5)

    return fts_prots
```

The second function of note is used to generate the distance matrix which calculates the euclidean distance from each pixel of the query feature map (size: qx55x55x512) to the support prototypes (size: c+1 x 512). The distance matrix has resulting size q x 55 x 55 x (c+1) where each of the c+1 layers has the distance of the feature map to one of the c+1 support prototypes. The sample code is shown below:

Figure 4.12: Generates Euclidean Distance Matrix

```
def calEucDist(self, fts, prototypes, n_queries,n_classes,fts_dim):
    """
    Calculate the distance between features and prototypes as distance matrix

    Args:
        fts: input features, expect shape: Q x V x H' x W'
        prototype: prototype of one semantic class expect shape: V x (C+1)
    """

    # Calculates euclidean distance matrix: Q x (C+1) x H' x W'
    prototypes = prototypes.permute(1,0)
    fts = fts.permute(0,2,3,1)
    pred = torch.stack([torch.sqrt(torch.sum(torch.square(fts - prototypes[i]),dim=3))
              for i in range(n_classes)],dim=1)

    return pred
```

The third function of note is used to calculate the variance loss for supervised segmentation. This is implemented by multiplying the distance matrix by the object-class query mask composed of binary masks for each of the classes (with the same dimensions) so for each layer only the distances for that object class to the respective support vector are retained and the average taken. The sample code is shown below:

Figure 4.13: Calculates Variance Loss

```
def withinClusterLoss(self, var_loss, qry_mask, pred):
    """
    Calculate the average distance between features and respective prototypes
    using by applying binary masks to distance matrix and averaging

    Args:
       pred: distance matrix, expect shape: Q x (C+1) x H x W
       qry_mask: binary masks for semantic class expect shape: # Q x H x W x (C+1)
    """

    qry_mask = qry_mask.permute(0,3,1,2)
    var_loss = var_loss + (torch.sum(pred*qry_mask) /
                           (torch.sum(qry_mask)+ 1e-5))
    return var_loss
```

The fourth function of note is used to calculate the distance loss. This is implemented as the target distance minus the average distance between the support object prototypes. The sample code is shown below:

Figure 4.14: Calculates Distance Loss

```
def betweenClusterLoss(self, dist_loss,prototypes,target):
    """
    Calculate the target distance - average distance between object prototypes

    Args:
       prototype: prototype for each semantic class expect shape: V x (C+1)
       target: target distance between prototypes
    """
    n_class = prototypes.shape[1]
    cum_dist = 0
    n_conn = 0

    if n_class == 1:  # no inter cluster loss
        dist_loss += torch.tensor(0).cuda()
    else:
        for i in range(n_class):
            for j in range(i+1, n_class):
                cum_dist = cum_dist + torch.norm(prototypes[:,i]-prototypes[:,j])
                n_conn = n_conn + 1

    if n_class==1:
        dist_loss += torch.tensor(0).cuda()
    else:
        dist_loss += torch.maximum(target - (cum_dist / n_conn),
                                   torch.tensor(0).cuda())
    return dist_loss
```

The fifth function of note is used to calculate the alternative variance loss for multi-dimensional segmentation. This is implemented by calculating the distance between the part prototypes and the object prototype and applying the hinge function at the minimum target distance. The sample code is shown below:

Figure 4.15: Calculates Variance Loss

```
def withinObjectPartLoss(self, var_loss, obj_prototypes, part_prototypes,
                         n_classes, part_idx, target):
    """
    Calculate the average distance between part prototypes and respective
    object prototypes and applying the hinge function at minimum target distance

    Args:
       obj_prototypes: object prototypes, expect shape: V x (C+1)
       part_prototypes: part prototypes, expect shape: V x (P+1)
    """

    # (C+1) x V
    obj_prototypes = obj_prototypes.permute(1,0)
    # (P+1) x V
    part_prototypes = part_prototypes.permute(1,0)

    for i in range(n_classes):
        # background class is excluded
        if i == 0:
            var_loss += torch.tensor(0).cuda()
        else:
        # calculate distance between part prototypes and object prototype
            part_prots = torch.index_select(part_prototypes, 0, torch.where(part_idx==i)[0])
            n_partsclass = part_prots.shape[0]
            obj_prot = obj_prototypes[i,:]

            cum_dist = torch.sum(torch.sqrt(torch.sum(
                torch.square(part_prots - obj_prot),dim=1)),dim=0)
            cum_dist = (cum_dist / n_partsclass).long()

            var_loss += torch.maximum(cum_dist - target,
                                      torch.tensor(0).cuda())

    return var_loss
```

The sample code for part variance loss is not shown as it is functionally similar to variance loss functions with the exception it utilises the individual part prototypes and part query masks as input.

The sixth function of note is used to calculate the distance part loss. This is implemented as the target part distance minus the average distance between the support part prototypes for each object class. The sample code is shown below:

Figure 4.16: Calculates Part Distance Loss

```
def betweenPartClusterLoss(self, dist_loss,prototypes,target,n_classes,part_idx):
    """
    Calculate the average distance between part prototypes

    Args:
       prototype: part prototype for each semantic class expect shape: V x (P+1)
       target: target distance between part prototypes
    """

    # average part distance for each of the classes
    cumdist_loss = []

    for i in range(n_classes):
        dist_loss_part = 0
        cum_dist = 0
        n_conn = 0

        # background class is excluded
        if i == 0:
            dist_loss_part += torch.tensor(0).cuda()
        else:
        # for each of the other classes calculate distance between part prototypes
            supp_chosen = torch.index_select(prototypes, 1, torch.where(part_idx==i)[0])
            n_partsclass = supp_chosen.shape[1]

            for i in range(n_partsclass):
                if n_partsclass==1:  # no inter cluster loss
                    break
                for j in range(i+1, n_partsclass):
                    cum_dist = cum_dist + torch.norm(supp_chosen[:,i]-supp_chosen[:,j])
                    n_conn = n_conn + 1

            if n_partsclass==1:
                dist_loss_part += torch.tensor(0).cuda()
            else:
                dist_loss_part += torch.maximum(target - (cum_dist / n_conn), torch.tensor(0).cuda())
        cumdist_loss.append(dist_loss_part)

    # Mean of per class average distance (excluding any zero distance)
    cumdist_loss = torch.stack([i for i in cumdist_loss],dim=0)
    cumdist_loss = torch.sum(cumdist_loss) / (torch.sum(torch.where(cumdist_loss != 0, 1, 0))+ 1e-6)
    dist_loss = dist_loss + cumdist_loss

    return dist_loss
```

4.6 Prediction Methodology:

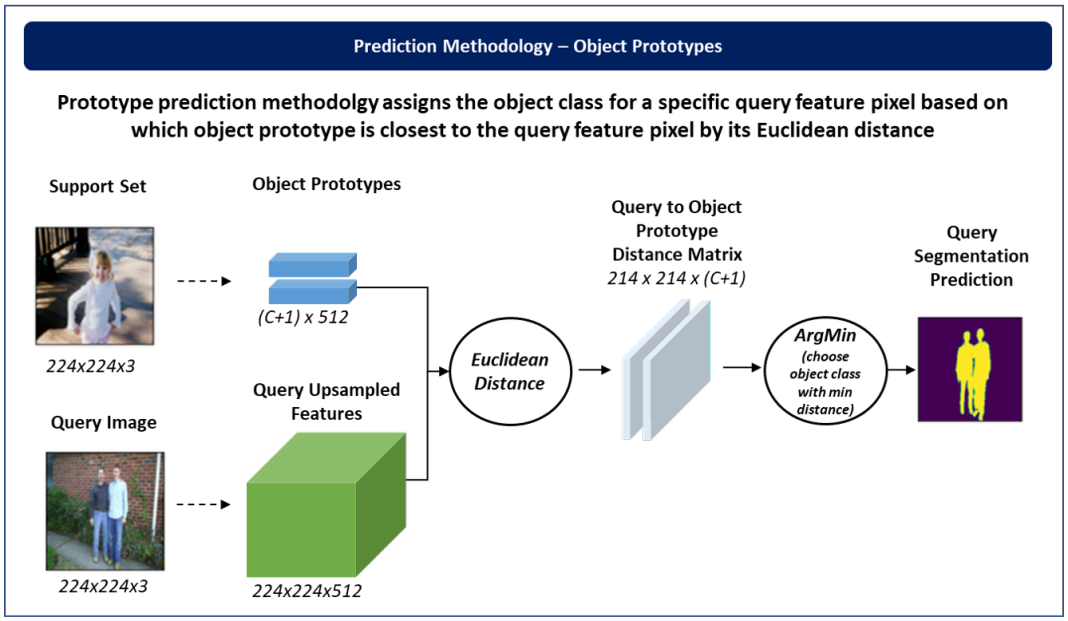
4.6.1 Design:

The prediction methodology determines how the upsampled query feature map can be segmented into respective object classes (i.e., each pixel is assigned an object class) using the upsampled support feature map and the associated ground truth support object labels.

The base methodology used in this project for prediction is to firstly calculate the mean support feature vector (support prototype) for each object class using the support features. The euclidean distance between the query feature map vectors and the support prototypes from each class is calculated to generate a distance matrix. The prediction of the object class is generated when the argmin function is applied to the distance matrix such that for each pixel the object class which has the smallest distance from the query feature pixel to the respective support prototype is chosen. The resulting prediction is a labelled array with the same dimensions of the original image where each pixel takes the value of the respective object class.

The figure below shows the base prediction methodology.

Figure 4.17: Prediction Methodology



4.6.2 Implementation:

For the support prototype methodology, the function to generate the distance matrix ("calEucDist") has already been outlined in the previous section and generates a distance matrix with size q x 55 x 55 x (c+1) where each of the c+1 layers has the distance feature map to one of the c+1 object support prototypes. The distance matrix is upsampled to the original image size of 224x224 using bilinear interpolation. The argmin function is used to generate the predicted object class for each pixel. The sample code is shown below:

```
# Upsample distance matrix to image size: C x H x W
query_pred = F.interpolate(pred, size=img_size, mode='bilinear',
                    align_corners=False)

# Apply argmin to generate predicted labels: H x W
qry_pred_flat = torch.argmin(query_pred.squeeze(dim=0),
                             dim=0).cpu().numpy()

# Ground truth segmentation from binary class masks
qry_masks_flat = torch.argmax(query_class_masks[0],
                              dim=2).cpu().numpy()
```

4.7 Code Structure:

4.7.1 Design:

The code has been given a modular structure similar to other published deep learning models which provides the key benefit of being able to adapt individual components (e.g., loss function or network architecture) without changing the full code.

4.7.2 Implementation:

The code has been structured such that the three main files in the directory are (1) "config.py" (2) "train.py" (3) "test.py". The "config.py" file contains all the hyperparameters and settings for training and testing of an experiment. The "train.py" file runs the training phase of the model for a specific experiment, with the model and graphs of the loss functions saved in the "experiments" folder to ensure if training is interrupted there is only minimal loss given the fairly lengthy training time of c.5 hours per model using technical specifications set out in Chapter 3. The "test.py" file accesses the saved model from training and calculates evaluation metrics using the test dataset, as well as saving sample visualisations of predictions.

The other folders within the directory contain the files that control each of the key aspects in the models. These are set out below with a description of their functions:

1. dataloaders - Contains the "pascal_loader.py" and "fewshot_loader.py" modules to load the PASCAL-Part dataset in a few shot context, in addition to the PASCAL-Part helper module to load parts segmentation.

2. loss - Contains the supervised and multi-dimensional segmentation DML loss function

3. models - Contains the base VGG-16 and Resnet-50 network architectures

4. utils - Contains the code to generate evaluation metrics and various experiment logging functions

The figure below outlines the directory structure and files:

Model File Structure

Folder Structure		Description
train.py		*Code to run training of model*
test.py		*Code to test model after evaluation*
config.py		*Contains configuration and hyperparameters*
data		
	VOCdevkit	*PASCAL-VOC 2010 source data folder*
dataloaders		
	fewshot_loader.py	*Loads samples in a few-shot context*
	pascal_loader.py	*Loads pascal images and annotations*
	part2obj.py	*Map of parts to objects*
	transforms.py	*Resizing and Normalization of images*
experiments		*Folder to contain saved experiments*
loss		
	discriminative_loss.py	*Contains Object and Part discriminative loss function*
models		
	network.py	*Network structure to load model architecture*
	vgg.py	*VGG-16 torch model structure*
	resnet.py	*ResNet-100 torch model structure*
	pre-trained_model	*Folder containing pre-trained weights*
utils		
	metric.py	*Contains module to calculate evaluation metrics*
	utils.py	*Contains functions for saving experiments*

Chapter 5

Results & Evaluation

This chapter reports the results and evaluates the performance of the few-shot deep metric learning models developed for semantic segmentation and multi-dimensional segmentation. The performance is analysed using the standard image segmentation benchmarks and compared against previous published results within the space. In addition, this section considers the impact from alternate design choices to the base model for some key criteria including network architecture and prediction methodology.

The chapter is split into four sub-sections which cover the following areas:

1. Base experimental setup
2. Testing criteria and benchmark evaluation metrics
3. Results and evaluation of semantic segmentation and multi-dimensional segmentation models
4. Results and evaluation of alternative model specifications

5.1 Base Experimental Setup

As described in the implementation section, the base models are initialised with the VGG-16 network architecture with weights pre-trained on ImageNet. During training, the model is optimised using stochastic gradient descent with an initial learning rate of 1e-06 and momentum of 0.9. A learning rate scheduler is applied with gamma decay rate of 0.1 each 10,000 episodes. The number of training iterations used is 30,000 episodes with a batch size of 1. The base prediction methodology for segmentation from the query feature maps is defined by the object class which has the minimum euclidean distance to the respective support object class prototypes.

The models are trained on 3 folds of the PASCAL-5^i with base PASCAL-Part dataset and 4 object classes without parts excluded (i.e., 12 object classes for training with 4 per fold) and evaluated on the remaining fold (i.e., 4 unseen object classes). In cross-validation style, for each task of supervised segmentation and multi-dimensional segmentation four models are trained each using a different split and testing fold with the results recorded.

The baseline for the two few-shot DML segmentation models is the segmentation performance of VGG-16 network pre-trained on ImageNet without any training on the Pascal 5i dataset.

5.2 Testing Criteria and Benchmark Evaluation Metrics

For evaluation, the models are tested on 3 runs of 1,000 randomly sampled episodes from the unseen testing object classes and the average performance of these results is taken. The models deliver quite consistent results between each of the 3 runs.

There are two principal metrics that are common across published research to evaluate the performance of few-shot segmentation models, mean Intersection-over-Union ("mean-IoU") [42][34][45] and binary Intersection-over-Union ("binary-IoU") [45][26].

The mean-IoU metric takes the mean of the Intersection-over-Union for each of the c-way foreground object classes in an experiment. The IoU for a particular class c is defined as:

$$IoU_c = \frac{TP_c}{TP_c+FP_c+FN_c} \tag{5.1}$$

TP_c is the total number of pixels which are true positives between the predicted segmentation mask and ground truth label mask for a particular class c over the test set (i.e. 1,000 episodes) and where FP_c and FN_c are the same for False Positives and False Negatives respectively. The mIoU is the average of IoUs over the testing object classes (i.e., 4 unseen object classes with the background class disregarded and only the foreground object classes are considered):

$$mIoU = \frac{1}{c_{test}} \sum_{c=1}^{c_{test}} IoU_c \tag{5.2}$$

The binary-IoU metric treats all object categories as one foreground class and averages the IoU of foreground and background. The bIoU is therefore defined as the mean of IoU for foreground classes and background classes:

$$bIoU = \frac{1}{2} \sum_{c=bg,fg} \frac{TP_c}{TP_c+FP_c+FN_c} \tag{5.3}$$

Bg and fg refer to the background and foreground (i.e. combined foreground classes). TP_c, FP_c and FN_c are the total number of pixels which are True Positives, False Positives and False Negatives respectively between the predicted segmentation mask and ground truth label mask for foreground and background over the test set (i.e., 1,000 episodes).

5.3 Results and Evaluation of Semantic Segmentation and Multi-dimensional Segmentation Models

5.3.1 One-Way, One-Shot Task - Mean IoU

The mean IoU results from one-way, one-shot task for the supervised segmentation, multi-dimensional segmentation, baseline initialisation and comparison from previous published papers [23] are outlined in the figures below. The figure compares the mean IoU metric for each

of the four folds.

One Shot, One Way – Mean IoU Performance Comparison

Model	Pascal 5i - Testing Split				Mean
	split-1	split-2	split-3	split-4	
Benchmark					
(0) Model-Initialised[1]	**28.1**	**33.4**	**29.5**	**23.0**	**28.5**
Project Models					
(1) Supervised[1]	**38.9**	**52.2**	**47.2**	**33.6**	**43.0**
(2) Multidimensional[1]	**37.3**	**48.6**	**46.2**	**33.1**	**41.3**
Comparison Models					
OSLSM	33.6	55.3	40.9	33.5	40.8
co-FCN	31.7	50.6	44.9	32.4	41.1
SG-one	40.2	58.4	48.4	38.4	46.3
PANet	42.3	58.0	51.1	41.2	48.1
PPNet	52.7	62.8	57.4	47.7	55.2
CANet	52.5	65.9	51.3	51.9	55.4
PGNet	56.0	66.9	50.6	50.4	56.0
FWB	51.3	64.5	56.7	52.2	56.2

Note: (1) PASCAL-5^i structure with base PASCAL-Part dataset (additional annotations for PASCAL VOC 2010) and removal of four non-part object classes

It can be seen that the Supervised Segmentation model performance in a one-shot, one-way setting of 43.0% mean IoU places it in the bottom-half of previously published few-shot research. The performance of the model is broadly in-line with expectations given the simpler end-to-end network design to allow a comparison between supervised and multi-dimensional segmentation. Additionally, the Supervised Segmentation model utilises a DML loss function, as compared to other publications typically using a categorical loss, and furthermore does not utilise any decoder module or post-processing techniques to refine the results.

It can be seen that the multi-dimensional segmentation model performance in a one-shot, one-way setting of 41.3% mean IoU likewise places it in the bottom-half of previously published few-shot research. The lower performance as compared to Supervised Segmentation suggests that the inclusion of parts information using a deep-metric learning loss function does not provide useful object segmentation information but rather degrades such discrimination. Conceptually, this lower performance compared to Supervised Segmentation does not come as a surprise within the deep metric context of the model. The models use object prototypes from the support set as the basis for prediction of a query pixel's object class by choosing the object prototype which is closest. The multi-dimensional loss function learns an embedding space such that individual part prototypes for an object are separated, this results in the respective object feature space being larger as compared to Supervised Segmentation where the object features are pushed as close together as possible. The resulting larger object feature space in multi-dimensional segmentation leads to lower ability to discriminate objects using object prototypes as prediction as there will be more overlap between object regions in the embedding

space. The discrimination of parts within the embedding space is therefore not fully utilised. An alternative prediction methodology is outlined in the next section which uses the individual part support prototypes for prediction to see whether this would result in a higher performance.

5.3.2 One-Way, Five-Shot Task - Mean IoU

The mean IoU results from one-way, five-shot task for the supervised segmentation, multi-dimensional segmentation, baseline initialisation and comparison from previous published papers are outlined in the figures below. The figure compares the mean IoU metric for each of the four folds.

Figure 5.1: mIoU: One Way, Five Shot Comparison

Five Shot, One Way – Mean IoU Performance Comparison

Model	Pascal 5i - Testing Split				Mean
	split-1	split-2	split-3	split-4	
Benchmark					
(0) Model-Initialised[1]	**35.1**	**43.9**	**37.5**	**29.3**	**36.5**
Project Models					
(1) Supervised[1]	**43.0**	**58.7**	**47.5**	**35.2**	**46.1**
(2) Multidimensional[1]	**41.1**	**55.3**	**46.6**	**34.7**	**44.4**
Comparison Models					
OSLSM	35.9	58.1	42.7	39.1	43.9
co-FCN	37.5	50.0	44.1	33.9	41.4
SG-one	41.9	58.6	48.6	39.4	47.1
PANet	51.8	64.6	59.8	46.5	55.7
PPNet	60.3	70.0	69.4	60.7	65.1
CANet	55.5	67.8	51.9	53.2	57.1
PGNet	57.7	68.7	52.9	54.6	58.5
FWB	54.8	67.4	62.2	55.3	59.9

Note: (1) PASCAL-5^i structure with base PASCAL-Part dataset (additional annotations for PASCAL VOC 2010) and removal of four non-part object classes

The supervised segmentation and multi-dimensional segmentation models show a similar relative result in the five-shot, one-way task with mean IoU of 46.1% and 44.4% respectively with the uplift in performance between 1- shot and 5-shot in-line with the majority of other models (c.3.1%). This relatively small uplift can likely be explained by the prototype approach to prediction which will amalgamate features and not fully capture diversity of object features from the different support images compared to, for example a KNN prediction methodology.

5.3.3 One-Way, One-Shot, Five-Shot Tasks - Binary IoU

The binary IoU results from one-way, one-shot and five-shot tasks for the supervised segmentation, multi-dimensional segmentation, baseline initialisation and comparison from previous published papers are outlined in the figures below. The figure compares the average binary-IoU

over the four folds. The results for binary-IoU for both one-shot and five-shot are likewise comparable to previous research, albeit at the bottom of the range.

Figure 5.2: bIoU: One Way, Five Shot Comparison

One Way, One Shot and Five Shot – Binary IoU Performance Comparison

Model	One Shot	Five Shot
Benchmark		
(0) Model-Initialised[1]	**52.2**	**56.5**
Project Models		
(1) Supervised[1]	**57.5**	**60.9**
(2) Multidimensional[1]	**56.5**	**60.0**
Comparison Models		
OSLSM	61.3	61.5
co-FCN	60.1	60.2
SG-one	63.9	65.9
PANet	66.5	70.7
PPNet	70.9	77.45
CANet	66.2	69.6
PGNet	69.9	70.5

Note: (1) PASCAL-5^i structure with base PASCAL-Part dataset (additional annotations for PASCAL VOC 2010) and removal of four non-part object classes

5.3.4 One-Way, One-Shot, Five-Shot Tasks - Visualisations

The figures below show a visual comparison of example query images and predictions from the supervised segmentation and multidimensional segmentation models for one-shot and five-shot tasks. It can be seen in the examples that both models deliver broadly similar results on the same images, albeit with some differences. It is worth noting that the DML models can also deal with occlusion as shown in the fourth image of Figure 5.4 where the motorbike is split by the person riding it. For both models it can be seen that the segmentation is somewhat coarse around the objects with limited fine-grained detail. This is as a result of the downsampling of the feature maps from the model 55x55x512 and subsequent bilinear upsampling to 214x214x512 and a loss of information at the individual pixel level.

Figure 5.3: Visualisations

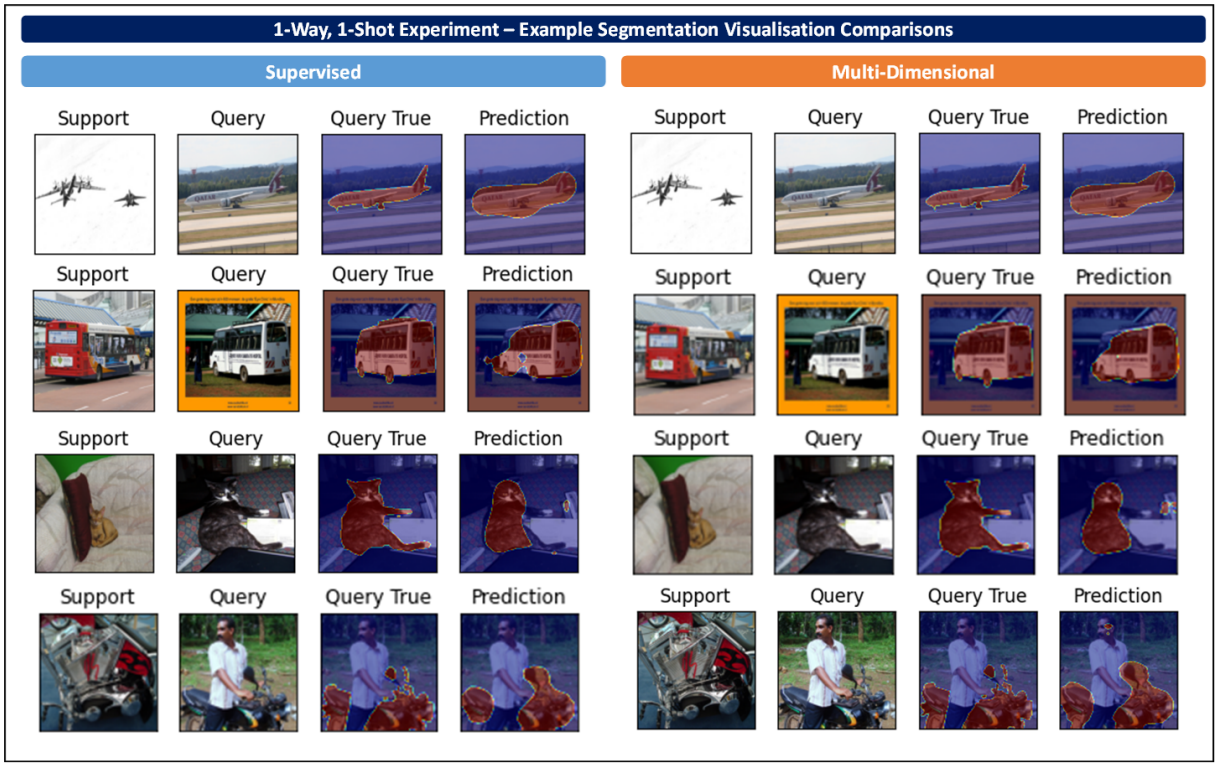
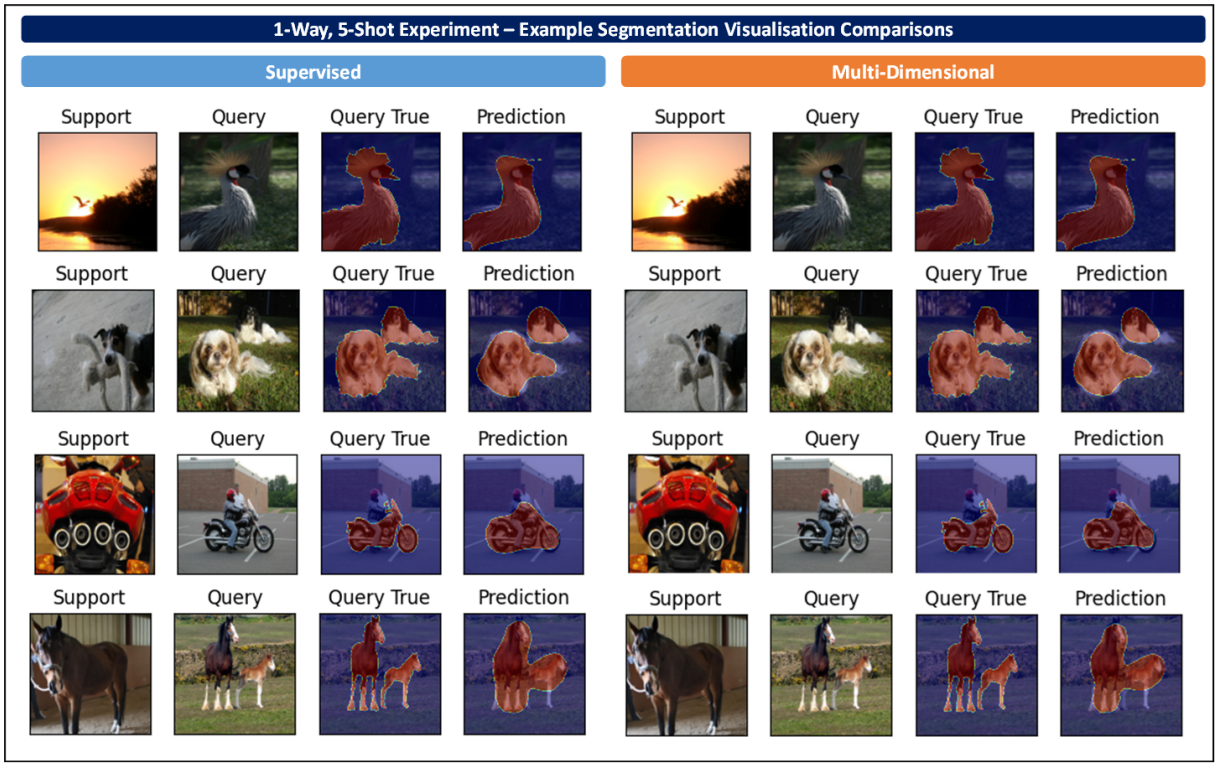


Figure 5.4: Visualisations



5.3.5 One-Way, One-Shot, Five-Shot Tasks - Average Loss Evolution over Training

The figures below show the evolution of the DML loss function over training for supervised Segmentation and multi-dimensional segmentation.

It can be seen for supervised Segmentation, the variance loss and distance loss fall at similar rates over the training period. The resulting average variance loss at the end of training is c.66 which implies the feature pixels are on average within 66 units from support object prototypes in the embedding space. The resulting average distance loss at the end of training is 93 which implies the support prototypes are on average 157 units (250 target- 93) away from each other. This implies that the support prototypes are well separated within the embedding space and the support features from different objects should not overlap on average.

The resulting distances for multi-Dimensional Segmentation likewise suggests well separated parts and objects within the embedding space, albeit with more room for overlap given the average distance of c.94 units between parts implied by the var part loss.

Whilst not used in the loss function for backpropagation and gradient descent, the cross-entropy loss using the predicted segmentation from the model is also calculated as another benchmark of model performance and which can be seen to be reducing over the training period.

Figure 5.5: Visualisations

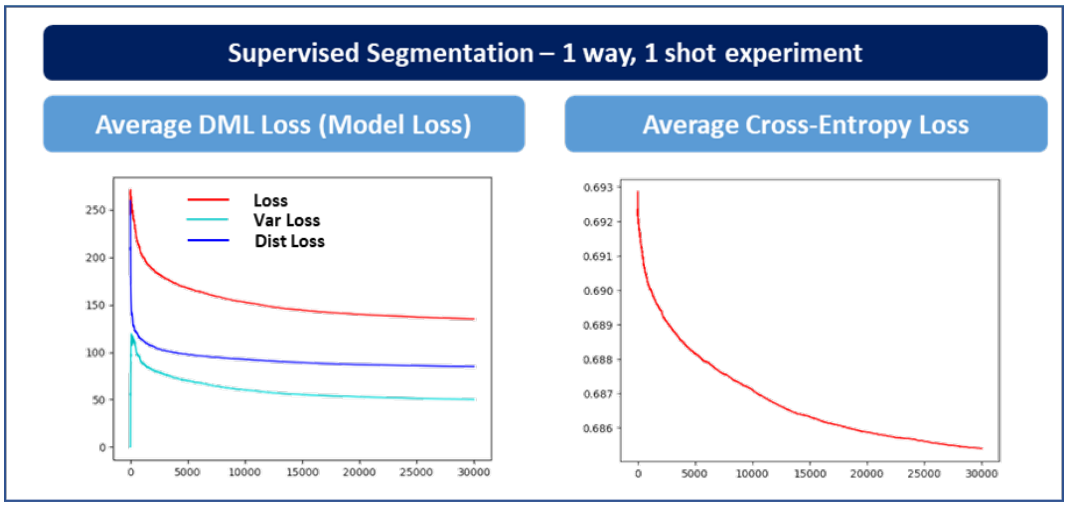
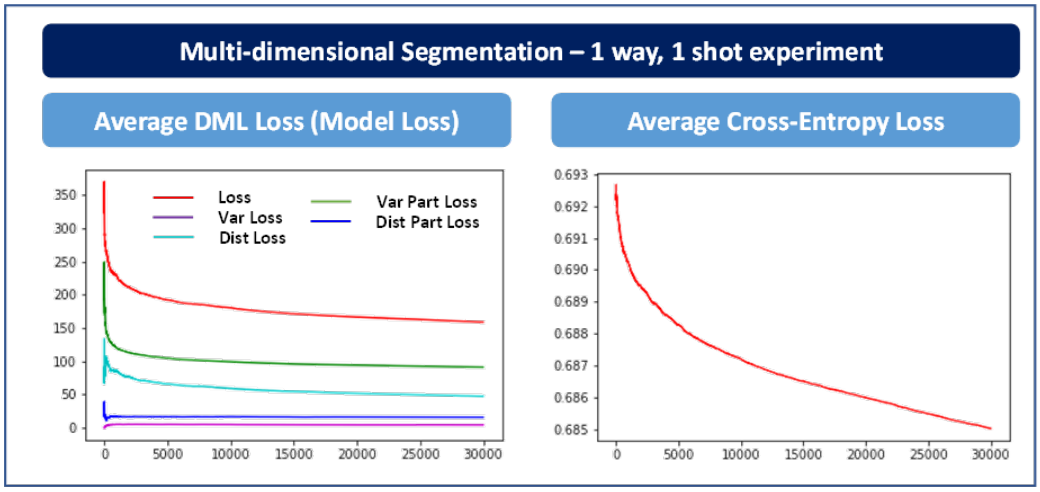


Figure 5.6: Visualisations



5.4 Results and Evaluation of Alternative Model Specifications

Beyond the main comparison of the two types of segmentation, the project also evaluates the impact of a number of key parameters within the models which were outlined in Chapter 4 "Design".

5.4.1 Prediction Methodology: Object Prototype (base) vs Part Prototype Prediction

The first parameter to be assessed is the use of support part prototypes instead of support object prototypes for the prediction methodology to understand whether segmenting by parts and mapping the resulting prediction to objects performs better than using an object prototype. For the prediction methodology, the query feature is assigned to the part which has the closest part prototype amongst the part prototypes in the support set. Once the predicted part labels have been established for the query image, these parts are mapped to their respective object to generate the object segmentation. The intention is therefore to create an embedding space with well separated parts and assign an object class based on a part prediction to account for diverse appearance in objects with different parts, poses or subcategories.

As shown in the figure below, the parts prediction delivers significantly worse results than Supervised and Multidimensional models using object prototypes as the prediction methodology. This result is likely due to the high number of parts that the model is being asked to predict therefore transforming a 1-way object segmentation task into a p-way part segmentation task and the part features not being suitably far apart in the embedding space even after training.

The mean IoU results for all folds of PASCAL-5^i are outlined below.

Figure 5.7: mIoU: Part Prototype Prediction Methodology

Parts Prediction Methodology – 1-way, 1-shot Mean IoU

Model	Pascal 5i - Testing Split				Mean
	split-1	split-2	split-3	split-4	
Alternative – Parts Prediction					
Multidimensional[1]	**29.0**	**36.0**	**35.5**	**20.6**	**30.3**
Base Project Models					
(0) Model-Initialised[1]	28.1	33.4	29.5	23.0	28.5
(1) Supervised[1]	38.9	52.2	47.2	33.6	43.0
(2) Multidimensional[1]	37.3	48.6	46.2	33.1	41.3

Note: (1) PASCAL-5^i structure with base PASCAL-Part dataset (additional annotations for PASCAL VOC 2010) and removal of four non-part object classes

5.4.2 Loss Function: DML Discriminative Loss (direct distance)(base) vs Cross-Entropy Loss (indirect distance)

The second parameter to be assessed is the use of a metric learning based categorical loss function (Cross Entropy Loss) compared to the deep metric learning discriminative loss function to understand whether the learned distance based-embedding space based on categorical prediction is more effective than distance alone within a DML loss function. The categorical function tested is the cross-entropy loss function. The prediction of objects labels continues to be based on the euclidean distance between query features and support prototypes and the prediction is compared to ground-truth labels in the cross-entropy loss function. Therefore, the loss function indirectly uses a distance metric for prediction. As shown in the figure below, the use of a categorical loss function results in significantly higher accuracy than the DML loss function. Interestingly, even though performance is high, observing the segmentation loss evolution (which the model is not optimised on) shows a weak DML embedding space as it can be seen that all object classes are highly concentrated in the embedding space (given the low variance and high distance loss).

The mean IoU results for the first fold of PASCAL-5^i is outlined below and graphs of the average loss evolution.

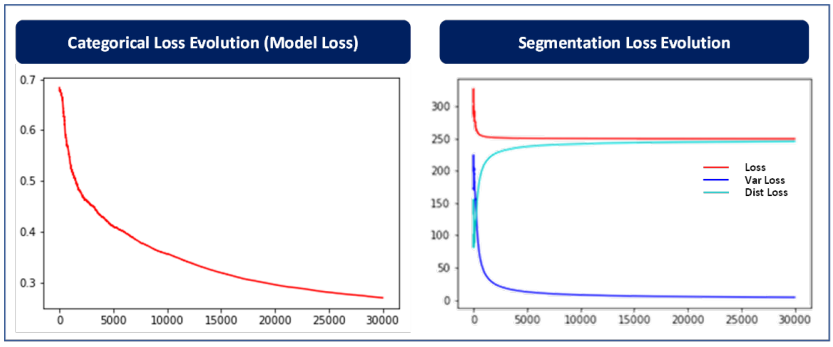
Figure 5.8: mIoU: Model Cross-Entropy Loss

Categorical Loss – 1-way, 1-shot Mean IoU

Model	Pascal 5i - Testing Split
	split-1
Alternative – Categorical Loss	
Supervised[1]	**42.8**
Base Project Models	
(0) Model-Initialised[1]	28.1
(1) Supervised[1]	38.9
(2) Multidimensional[1]	37.3

Note: (1) PASCAL-5^i structure with base PASCAL-Part dataset (additional annotations for PASCAL VOC 2010) and removal of four non-part object classes

Figure 5.9: Average Loss Evolution - Cross Entropy Loss



5.4.3 Network Architecture: VGG-16 (base) vs ResNet50

The third parameter to be assessed is the use of the pre-trained ResNet50 backbone architecture, compared to pre-trained VGG-16, to understand the impact of alternative deeper architectures on performance and the embedding space

It can be seen that using the ResNet50 architecture results in a decrease in performance compared to VGG-16. It is interesting to note that without any training on Pascal 5i, the performance of the architecture is significantly lower than VGG-16 on initialisation. This may be a result of the larger depth of ResNet50 and the larger downsampling of the feature map (28x28x2024) for ResNet50 vs VGG-16 (55x55x512) and subsequent bilinear interpolation up-sampling needed leading to more coarse segmentation. The embedding space is also significantly more concentrated than with the VGG-16 base architecture with high distance loss and low variance loss.

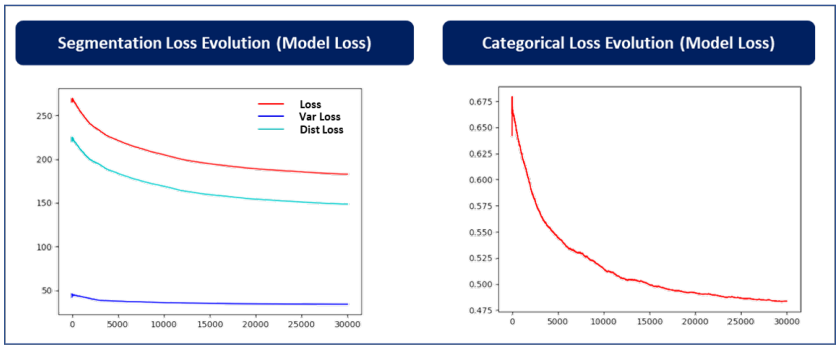
Figure 5.10: mIoU: ResNet50 Architecture

ResNet50 Architecture – 1-way, 1-shot Mean IoU

Model	Pascal 5i - Testing Split
	split-1
Alternative – ResNet Architecture	
Model-Initialised[1]	**16.6**
Supervised[1]	**31.3**
Base Models	
(0) Model-Initialised[1]	28.1
(1) Supervised[1]	38.9
(2) Multidimensional[1]	37.3

Note: (1) PASCAL-5^i structure with base PASCAL-Part dataset (additional annotations for PASCAL VOC 2010) and removal of four non-part object classes

Figure 5.11: Average Loss Evolution - ResNet50 Architecture



Chapter 6

Legal, Social, Ethical and Professional Issues

This chapter outlines the legal, social ethical and professional issues within the context of the project.

6.1 British Computer Society Code of Conduct & Code of Good Practice

Throughout the project, care has been taken to abide by the British Computing Society's Code of Conduct (COC) and Code of Good Practice. These guidelines primarily set out a code of ethics and standards of best practice that should govern individual decisions and behaviours in software development. Whilst a number of the guidelines are applicable to the project, the rule in relation to the safeguarding and protection of intellectual property is the most applicable:

"You shall:...b. have due regard for the legitimate rights of Third Parties*.have due regard for the legitimate rights of third parties;" [3]

This rule has been duly reflected in the research, design and implementation of the software and all reports produced. The project code makes use of a number of open-source third-party libraries and frameworks in addition to adapting where applicable the code from projects in the same field published online. The report also outlines a number of ideas from various authors in research publications within the space. This use of third-party code, ideas and information is accepted within the field of computer science in the interests of progress and to build on the work already compiled by previous individuals. However, it is necessary to give due credit to these third-parties and ensure there is no infringement on their intellectual property rights. In that regard, a significant amount of care has been taken to source and reference each idea, code and information where it has been prepared by third-parties so that I do not claim ownership of the work performed by others.

6.2 Social and Ethical Issues

One of the key social and ethical issues surrounding the project, is the potential unethical use of the few-shot semantic segmentation within a surveillance context. The use of the project within surveillance may become unethical if it involves sensitive information where consent is not given (e.g., facial recognition) or where such surveillance leads to breaches of human rights. It is not intended that the project technology be applied by third-parties in such a manner, rather the main applications were focused on the fields of medicine and security.

Chapter 7

Conclusion and Future Work

In conclusion, the project has achieved its goal of building models for the tasks of supervised segmentation and for multidimensional segmentation using a DML framework in a few-shot context. The final segmentation results produced by this project achieved high levels of performance which were comparable with previously published results and highlight the capabilities of a DML approach within few-shot learning

The results highlight that, whilst using object prototypes for prediction, the inclusion of part information in the DML loss function does not assist the model for object segmentation. Optimising for parts in the loss function degrades the embedding space by expanding the distance that an object occupies, resulting in less well-defined spaces and more likelihood of another object being crossing-over in this space.

The alternative experiments highlighted that even the use of part prototypes for prediction (rather than object prototypes) does not benefit overall object segmentation performance, given the few-shot context and small number of support images likely impacting the ability to predict the larger number of parts in any task.

Interestingly, the alternative experiments also highlight that the use of a Categorical Loss function with prediction based on a distance metric results in distinctly better performance than using a DML loss function directly.

The main area where further work can be carried out is the utilisation of parts information in a non-few shot experiment context. One of the main features of part information is that the individual parts should be more readily identifiable to the model than an overall object because of the varying poses etc. However, in a few-shot context this ability does not seem capable of being taken advantage of given the small amounts of parts information to train on given the support size. Therefore, the project could be extended to examine whether part information benefits object segmentation with a higher amount of training data.

References

[1] Konstantinos G. Derpanis Adam W. Harley and Iasonas Kokkinos. Learning dense convolutional embeddings for semantic segmentation. *ICLR 2016*, 2016.

[2] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation, 2016.

[3] The Chartered Institute for IT BCS. Bcs code of conduct. `https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/`, 2021.

[4] Aurélien Bellet, Amaury Habrard, and Marc Sebban. A survey on metric learning for feature vectors and structured data, 2014.

[5] Bert De Brabandere, Davy Neven, and Luc Van Gool. Semantic instance segmentation with a discriminative loss function, 2017.

[6] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs, 2017.

[7] Xianjie Chen, Roozbeh Mottaghi, Xiaobai Liu, Sanja Fidler, Raquel Urtasun, and Alan Yuille. Detect what you can: Detecting and representing objects using holistic models and body parts, 2014.

[8] Yuhua Chen, Jordi Pont-Tuset, Alberto Montes, and Luc Van Gool. Blazingly fast video object segmentation with pixel-wise metric learning, 2018.

[9] Nameirakpam Dhanachandra, Khumanthem Manglem, and Yambem Jina Chanu. Image segmentation using k -means clustering algorithm and subtractive clustering algorithm. *Procedia Computer Science*, 54:764–771, 2015. Eleventh International Conference on Communication Networks, ICCN 2015, August 21-23, 2015, Bangalore, India Eleventh International Conference on Data Mining and Warehousing, ICDMW 2015, August 21-23, 2015, Bangalore, India Eleventh International Conference on Image and Signal Processing, ICISP 2015, August 21-23, 2015, Bangalore, India.

[10] Stan Z. Li Dong Yi, Zhen Lei. Deep metric learning for practical person re-identification. *2014 22nd International Conference on Pattern Recognition*, 2014.

[11] Mark Everingham, S. Eslami, Luc Van Gool, Christopher Williams, John Winn, and Andrew Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111, 01 2014.

[12] Alireza Fathi, Zbigniew Wojna, Vivek Rathod, Peng Wang, Hyun Oh Song, Sergio Guadarrama, and Kevin P. Murphy. Semantic instance segmentation via deep metric learning, 2017.

[13] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and Jose Garcia-Rodriguez. A review on deep learning techniques applied to semantic segmentation, 2017.

[14] Bharath Hariharan, Pablo Arbeláez, Lubomir Bourdev, Subhransu Maji, and Jitendra Malik. Semantic contours from inverse detectors, 2011.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[17] Elad Hoffer and Nir Ailon. Deep metric learning using triplet network, 2018.

[18] Minyoung Huh, Pulkit Agrawal, and Alexei A. Efros. What makes imagenet good for transfer learning?, 2016.

[19] Trevor Darrell Jonathan Long, Evan Shelhamer. Fully convolutional networks for semantic segmentation. *CVPR*, 2015.

[20] Mahmut Kaya and H. Bilge. Deep metric learning: A survey. *Symmetry*, 11:1066, 08 2019.

[21] Kevin Lin, Lijuan Wang, Kun Luo, Yinpeng Chen, Zicheng Liu, and Ming-Ting Sun. Cross-domain complementary learning using pose for multi-person part segmentation. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(3):1066–1078, Mar 2021.

[22] Wei Liu, Andrew Rabinovich, and Alexander C. Berg. Parsenet: Looking wider to see better, 2015.

[23] Yongfei Liu, Xiangyi Zhang, Songyang Zhang, and Xuming He. Part-aware prototype network for few-shot semantic segmentation, 2020.

[24] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey, 2020.

[25] Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. Pytorch metric learning, 2020.

[26] Eric P Xing Nanqing Dong. Few-shot semantic segmentation with prototype learning. *British Machine Vision Conference*, 2018.

[27] Richard Nock and Frank Nielsen. Statistical region merging. *IEEE transactions on pattern analysis and machine intelligence*, 26:1452–8, 12 2004.

[28] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation, 2015.

[29] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.

[30] Nils Plath, Marc Toussaint, and Shinichi Nakajima. Multi-class image segmentation using conditional random fields and global classification, 01 2009.

[31] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[32] Karsten Roth, Timo Milbich, Samarth Sinha, Prateek Gupta, Björn Ommer, and Joseph Paul Cohen. Revisiting training strategies and generalization performance in deep metric learning, 2020.

[33] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge, 2015.

[34] Amirreza Shaban, Shray Bansal, Zhen Liu, Irfan Essa, and Byron Boots. One-shot learning for semantic segmentation, 2017.

[35] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[36] Jake Snell, Kevin Swersky, and Richard S. Zemel. Prototypical networks for few-shot learning, 2017.

[37] J.-L. Starck, M. Elad, and D.L. Donoho. Image decomposition via the combination of sparse representations and a variational approach. *IEEE Transactions on Image Processing*, 14(10):1570–1582, 2005.

[38] Farhana Sultana, Abu Sufian, and Paramartha Dutta. Evolution of image segmentation using deep convolutional neural network: A survey. *Knowledge-Based Systems*, 201-202:106062, Aug 2020.

[39] Yann LeCun Sumit Chopra, Raia Hadsell. Learning a similarity metric discriminatively, with application to face verification. *CVPR*, 2005.

[40] Stavros Tsogkas. Pascal-part classes. `https://github.com/tsogkas/pascal-part-classes`, 2015.

[41] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning, 2017.

[42] Kaixin Wang, Jun Hao Liew, Yingtian Zou, Daquan Zhou, and Jiashi Feng. Panet: Few-shot image semantic segmentation with prototype alignment, 2020.

[43] Kilian Weinberger, J. Blitzer, and L. Saul. Distance metric learning for large margin nearest neighbor classification, 01 2006.

[44] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions, 2016.

[45] Xiaolin Zhang, Yunchao Wei, Yi Yang, and Thomas Huang. Sg-one: Similarity guidance network for one-shot semantic segmentation, 2020.

Appendix A

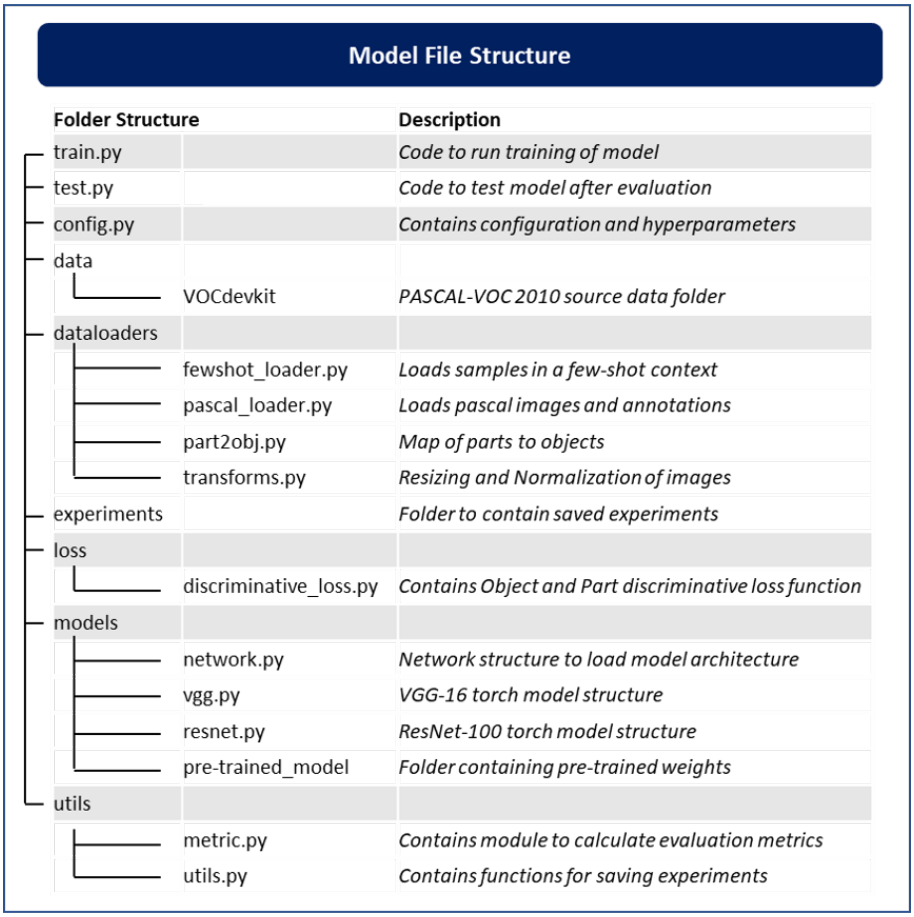
Source Code

A.1 Structure overview

The code has been structured such that the three main files in the directory are (1) "config.py" (2) "train.py" (3) "test.py". The "config.py" file contains all the hyperparameters and settings for training and testing of an experiment. The "train.py" file runs the training phase of the model for a specific experiment, with the model and graphs of the loss functions saved in the "experiments" folder to ensure if training is interrupted there is only minimal loss given the fairly lengthy training time of c.5 hours per model using technical specifications set out in Chapter 3. The "test.py" file accesses the saved model from training and calculates evaluation metrics using the test dataset, as well as saving sample visualisations of predictions. The other folders within the directory contain the files that control each of the key aspects in the models. These are set out below with a description of their functions:

1. "dataloaders" - Contains the "pascal_loader.py" and "fewshot_loader.py" modules to load the PASCAL Parts dataset in a few shot context, in addition to the PASCAL Parts helper module to load parts segmentation.
2. "loss" - Contains the loss functions
3. "models" - Contains the base VGG-16 and ResNet-50 network architectures
4. "utils" - Contains the code to generate evaluation metrics and various experiment logging functions

The figure below outlines the directory structure and files:



A.2 Key Model Files

The source code for the key files utilised in the models are specified below. The code uses the model published by "PANet: Few-Shot Image Semantic Segmentation with Prototype Alignment" [42] as a base and this has been altered for the tasks of semantic segmentation and multi-dimensional segmentation in a DML context. The code also draws on other open source code and all references to these have been included in the code comments.

The source code for the following key model files are shown below:

1. config.py
2. train.py
3. test.py
4. discriminative_loss.py
5. fewshot_loader.py
6. pascal_loader.py
7. part2obj.py
8. transforms.py

9. network.py
10. metric.py
11. utils.py

A.2.1 config.py:

```
"""
@author: A Kumarathas

Configuration Script:
Contains configuration and hyperparameters for training and testing of
experiments
"""

current_experiment = 'Exp1_Seg_1w1s_VGG_data0_local'

#Training configurations
loss_type = 'no_parts' # 'parts' or 'no_parts'
chkpts_dir = './experiments/'
data_name = 'VOC'
label_sets = 'parts0'
input_size = (224, 224)
path = {
    'log_dir': './runs',
    'init_path': './models/pretrained_model/vgg16-397923af.pth',
    'VOC':{'data_dir': './data/VOCdevkit/VOC2010',
           'data_split': 'trainparts',},}
n_steps = 30000
batch_size = 1
task = {'n_ways': 1,
        'n_shots': 1,
        'n_queries': 1,}
model_align = {'align': True,}
gpu_id = 0
optim = {'lr': 1e-06,
         'momentum': 0.9,
         'weight_decay': 0.0005,}
lr_milestones = [10000, 20000, 30000]
ignore_label = 255
print_interval = 50
save_pred_every = 500
calc_val_every = 500
n_valsteps = 100
architecture = 'VGG16'   #'ResNet50' or 'VGG16'

#Test configurations
max_label = 20
n_runs = 3
n_teststeps = 1000
seed = 1234
prediction_type = 'no_parts' # 'parts' or 'no_parts'
save_image_every = 100
visfolder = '/visualisations'
```

A.2.2 train.py:

```
"""
@author: A anonymous

Training Script:
Code to run training of model

Adapted code from "PANet: Few-Shot Image Semantic Segmentation with
Prototype Alignment" by Kaixin Wang et al. at https://github.com/kaixin96/PANet
"""

import os
import shutil

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim
from torch.utils.data import DataLoader
from torch.optim.lr_scheduler import MultiStepLR
import torch.backends.cudnn as cudnn
from torchvision.transforms import Compose
import matplotlib.pyplot as plt

from models.network import FewShotNet
from loss.discriminative_loss import DiscriminativeLoss
from dataloaders.fewshot_loader import voc_fewshot
from dataloaders.transforms import Resize, ToTensorNormalize
from util.utils import CLASS_LABELS, set_seed
from util.utils import config_experiment, save_experiment,config_logger,
    save_configfile,save_testfile_run,save_testfile_final
from util.utils import evaluate_val,log_losses,log_vallosses
from config import *


def train():

    set_seed(seed)

    # Set up an experiment
    experiment, exp_logger = config_experiment(current_experiment, resume=True)
    step = experiment['step']
    loss_hist = experiment['loss']
    avgloss_hist = experiment['avg_loss']
    log_loss = experiment['log_loss']
    val_hist = experiment['val_loss']
    exp_logger.info('training started/resumed at epoch ' + str(step))
    val_seed = 0

    #Prepare data
    transforms = Compose([Resize(size=input_size)])
    labels = CLASS_LABELS[data_name][label_sets]
    vallabels = CLASS_LABELS[data_name]['parts'] - CLASS_LABELS[data_name][
    label_sets]

```

```
    # Load Training Datasets
    dataset = voc_fewshot(
        base_dir=path[data_name]['data_dir'],
        split=path[data_name]['data_split'],
        transforms=transforms,
        to_tensor=ToTensorNormalize(),
        labels=labels,
        max_iters=n_steps * batch_size,
        n_ways=task['n_ways'],
        n_shots=task['n_shots'],
        n_queries=task['n_queries'])

    trainloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
                             num_workers=1, pin_memory=True, drop_last=True)

    # Load Model
    model = FewShotNet(pretrained_path=path['init_path'],
                       cfg=model_align, model_arch = architecture)
    model = nn.DataParallel(model.cuda(), device_ids=[gpu_id, ])
    model.load_state_dict(experiment['model_state_dict'])
    model.train()

    # Training Optimisation
    optimizer = torch.optim.SGD(model.parameters(), **optim)
    scheduler = MultiStepLR(optimizer, milestones=lr_milestones, gamma=0.1)

    # Load Loss function
    model_loss_fn = DiscriminativeLoss()

    #Save configurations of experiment
    save_configfile(current_experiment, model_loss_fn)

    # Training
    for i_iter, sample_batched in enumerate(trainloader):

        step += 1
        if step >= n_steps:
            break

        # Prepare input
        # Way x Shots x Imgdim x H x W
        support_images = sample_batched['support_images'][0, ...].cuda()

        # Q x Imgdim(3) x H x W
        query_images = sample_batched['query_images'][0, ...].cuda()

        # Way x Shots x H x W x (C+1)
        support_class_masks = sample_batched['support_class_mask'][0, ...].cuda
    ()

        # Q x H x W x (C+1)
        query_class_masks = sample_batched['query_class_mask'][0, ...].cuda()

        # Way x Shots x H x W x (C+1)
        support_part_masks = sample_batched['support_part_mask'][0, ...].cuda()

```

```
        # Q x H x W x (C+1)
        query_part_masks = sample_batched['query_part_mask'][0, ...].cuda()

        # P
        part_idx = [torch.tensor([0])] + sample_batched['part_classidx_full']
        part_idx = torch.cat([i for i in part_idx], dim=0).cuda()

        # Forward and Backward
        optimizer.zero_grad()

        # Generate Support and Query features
        support_fts, query_fts = model(support_images, query_images)

        n_ways = support_images.shape[0]
        n_shots = support_images.shape[1]
        n_queries = query_images.shape[0]
        img_size = support_images.shape[-2:]

        # Implement loss function on features and generate query seg prediction
        pred, _, var_loss, dist_loss,var_part_loss,dist_part_loss =
    model_loss_fn(support_fts,

    query_fts,

    support_class_masks,

    query_class_masks,

    support_part_masks,

    query_part_masks,

    n_ways,

    n_shots,

    n_queries,

    part_idx,

    img_size,

    loss_type)

        if loss_type == 'parts':
            loss = var_loss + dist_loss + var_part_loss + dist_part_loss
        else:
            loss = var_loss + dist_loss

        # Need 1/pred for euclidean distance as ce loss function assumes input
    is probability map
        ce_loss = F.cross_entropy(torch.reciprocal(pred)*20,torch.argmax(
    query_class_masks,dim=3))

        loss.backward()
        optimizer.step()
```

```
150         scheduler.step()
151 
152         # Log loss statistics
153         var_loss = var_loss.clone().detach().data.cpu().numpy()
154         dist_loss = dist_loss.clone().detach().data.cpu().numpy() if dist_loss
    != 0 else 0
155         var_part_loss = var_part_loss.clone().detach().data.cpu().numpy()
156         dist_part_loss = dist_part_loss.clone().detach().data.cpu().numpy() if
    dist_part_loss != 0 else 0
157         loss = loss.clone().detach().data.cpu().numpy()
158         ce_loss = ce_loss.clone().detach().data.cpu().numpy()
159 
160         loss_hist, log_loss, avgloss_hist = log_losses(loss_hist, log_loss,
    avgloss_hist,
161                                                        var_loss, dist_loss,
    var_part_loss,
162                                                        dist_part_loss, loss,
    ce_loss, step)
163         avgloss = avgloss_hist['loss'][-1]
164         avgvloss = avgloss_hist['var_loss'][-1]
165         avgdloss = avgloss_hist['dist_loss'][-1]
166         avgvploss = avgloss_hist['var_part_loss'][-1]
167         avgdploss = avgloss_hist['dist_part_loss'][-1]
168         avgceloss = avgloss_hist['crossentropy_loss'][-1]
169 
170         exp_logger.info(
171             f'step {step}: loss: {loss:.3f}, vloss: {var_loss:.3f}, dloss: {
    dist_loss:.3f},ce:{ce_loss:.3f} ')
172         # exp_logger.info(
173         #     f'step {step}: loss: {loss:.3f}, vloss: {var_loss:.3f}, dloss: {
    dist_loss:.3f}, vploss: {var_part_loss:.3f}, dploss: {dist_part_loss:.3f}')
174         # exp_logger.info(
175         #     f'step {step}: loss: {loss:.3f}, vploss: {var_part_loss:.3f},
    dploss: {dist_part_loss:.3f}')
176 
177 
178         # print loss and take snapshots
179         if (step + 1) % print_interval == 0:
180             exp_logger.info(f'step {step}: av loss: {avgloss:.3f},av vloss: {
    avgvloss:.3f}, av dloss: {avgdloss:.3f}, ce:{avgceloss:.3f}')
181             # exp_logger.info(f'step {step+1}: av loss: {avgloss:.3f},av vloss:
    {avgvloss:.3f}, av dloss: {avgdloss:.3f}, av vploss: {avgvploss:.3f}, av
    dploss: {avgdploss:.3f}, ce:{avgceloss:.3f}')
182             # exp_logger.info(f'step {step+1}: av loss: {avgloss:.3f}, av vploss
    : {avgvploss:.3f}, av dploss: {avgdploss:.3f}, ce:{avgceloss:.3f}')
183 
184 
185             # Plot and save loss history
186             plt.plot(avgloss_hist['loss'], 'r')
187             plt.plot(avgloss_hist['var_loss'], 'b')
188             plt.plot(avgloss_hist['dist_loss'], 'c')
189             # plt.plot(avgloss_hist['var_part_loss'], 'g')
190             # plt.plot(avgloss_hist['dist_part_loss'], 'm')
191             plt.savefig(chkpts_dir+current_experiment+'/avglosshistory.png')
192             plt.close()
193 
```

```
            plt.plot(avgloss_hist['crossentropy_loss'], 'r')
            plt.savefig(chkpts_dir+current_experiment+'/avgcehistory.png')
            plt.close()

        # calculate validation loss
        if (step + 1) % calc_val_every == 0:
            ########EVALUATION########
            model.eval()
            with torch.no_grad():
                print('###### Validation Loss ##########')
                val_seed += 1
                set_seed(seed + val_seed)
                valdataset = voc_fewshot(base_dir=path[data_name]['data_dir'],
                                         split=path[data_name]['data_split'],
                                         transforms=transforms,
                                         to_tensor=ToTensorNormalize(),
                                         labels=vallabels,
                                         max_iters=n_valsteps * batch_size,
                                         n_ways=task['n_ways'],
                                         n_shots=task['n_shots'],
                                         n_queries=task['n_queries'])

                valloader = DataLoader(valdataset, batch_size=batch_size,
shuffle=False,
                                       num_workers=1, pin_memory=True, drop_last
=False)



                val_loss = evaluate_val(valloader, model, model_loss_fn,
loss_type)
                val_hist = log_vallosses(val_loss, val_hist, n_valsteps)

                valloss = val_hist['loss'][-1]
                valvloss = val_hist['var_loss'][-1]
                valdloss = val_hist['dist_loss'][-1]
                valvploss = val_hist['var_part_loss'][-1]
                valdploss = val_hist['dist_part_loss'][-1]
                valceloss = val_hist['crossentropy_loss'][-1]
                exp_logger.info(f'step {step}: loss: {valloss:.3f}, vloss: {
valvloss:.3f}, dloss: {valdloss:.3f}, vploss: {valvploss:.3f}, dploss: {
valdploss:.3f},celoss: {valceloss:.3f}')

                # Plot and save loss history
                plt.plot(val_hist['loss'], 'r')
                plt.plot(val_hist['var_loss'], 'b')
                # plt.plot(val_hist['var_part_loss'], 'g')
                plt.savefig(chkpts_dir+current_experiment+'/vallosshistory.png')
                plt.close()

            model.train()
            set_seed(seed)

        # save experiment
        if (step + 1) % save_pred_every == 0:
            save_experiment({'model_state_dict': model.state_dict(),
```

```
                                'step': step,
                                'loss': loss_hist,
                                'log_loss': log_loss,
                                'avg_loss': avgloss_hist,
                                'val_loss': val_hist}, current_experiment)

if __name__ == '__main__':
    train()
```

A.2.3 test.py:

```
"""
@author: A anonymous

Evaluation Script:
Code to evaluate model after training

Adapted code from "PANet: Few-Shot Image Semantic Segmentation with
Prototype Alignment" by Kaixin Wang et al. at https://github.com/kaixin96/PANet
"""


import os
import shutil

import tqdm
import numpy as np
import torch
import torch.optim
import torch.nn as nn
from torch.utils.data import DataLoader
import torch.backends.cudnn as cudnn
from torchvision.transforms import Compose
import matplotlib.pyplot as plt

from models.network import FewShotNet
from loss.discriminative_loss import DiscriminativeLoss
from dataloaders.fewshot_loader import voc_fewshot
from dataloaders.transforms import Resize, ToTensorNormalize
from util.metric import Metric
from util.utils import CLASS_LABELS, set_seed
from util.utils import config_experiment, save_experiment,config_logger,
    save_configfile,save_testfile_run,save_testfile_final
from util.utils import evaluate_val,log_losses,log_vallosses
from config import *


def test():

    #Load model and loss function for prediction
    model = FewShotNet(pretrained_path=path['init_path'], cfg=model_align,
    model_arch = architecture)
    model = nn.DataParallel(model.cuda(), device_ids=[gpu_id,])
    try:
```

```
        experiment = torch.load(chkpts_dir+current_experiment+'/chkpt.pth',
    map_location=lambda storage, loc: storage)
        print("Loaded checkpoint model")
    except Exception as e:
        print("No trained model found")
        experiment, exp_logger = config_experiment(current_experiment,
                                                   resume=True)

    model.load_state_dict(experiment['model_state_dict'])
    model_loss_fn = DiscriminativeLoss()
    model.eval()

    #Prepare data
    transforms = [Resize(size=input_size)]
    transforms = Compose(transforms)
    labels = CLASS_LABELS[data_name]['parts'] - CLASS_LABELS[data_name][
    label_sets]
    print('Testing Labels', labels)

    # Creat visualisation directory
    os.makedirs(chkpts_dir+current_experiment+visfolder, exist_ok=True)


    # Evaluation script
    metric = Metric(max_label=max_label, n_runs=n_runs)
    with torch.no_grad():
        for run in range(n_runs):
            print(f'### Run {run + 1} ###')
            set_seed(seed + run)

            print(f'### Load data ###')

            dataset = voc_fewshot(
                 base_dir=path[data_name]['data_dir'],
                 split=path[data_name]['data_split'],
                 transforms=transforms,
                 to_tensor=ToTensorNormalize(),
                 labels=labels,
                 max_iters=n_teststeps * batch_size,
                 n_ways=task['n_ways'],
                 n_shots=task['n_shots'],
                 n_queries=task['n_queries'])

            testloader = DataLoader(dataset, batch_size=batch_size, shuffle=
    False,
                                    num_workers=1, pin_memory=True, drop_last=
    False)
            print(f"Total # of Data: {len(dataset)}")

            for step, sample_batched in enumerate(testloader):
                print(step)

                label_ids = list(sample_batched['class_ids'])
                part_ids = list(sample_batched['part_ids'])

                # Load images and Masks
```

```
                # Way x Shots x Imgdim x H x W
                support_images = sample_batched['support_images'][0,...].cuda()

                # Q x Imgdim(3) x H x W
                query_images = sample_batched['query_images'][0,...].cuda()

                # Way x Shots x H x W x (C+1)
                support_class_masks = sample_batched['support_class_mask'
    ][0,...].cuda()

                # Q x H x W x (C+1)
                query_class_masks = sample_batched['query_class_mask'][0,...].
    cuda()

                # Way x Shots x H x W x (C+1)
                support_part_masks = sample_batched['support_part_mask'][0,...].
    cuda()

                # Q x H x W x (C+1)
                query_part_masks = sample_batched['query_part_mask'][0,...].cuda
    ()

                # P
                part_idx = [torch.tensor([0])] + sample_batched['
    part_classidx_full']
                part_idx = torch.cat([i for i in part_idx], dim=0).cuda()

                # Generate Support and Query features
                support_fts, query_fts = model(support_images, query_images)

                n_ways = support_images.shape[0]
                n_shots = support_images.shape[1]
                n_queries = query_images.shape[0]
                img_size = support_images.shape[-2:]

                # Generate Query Segmentation Prediction
                query_pred,query_pred_part, _, _, _, _ = model_loss_fn(
    support_fts, query_fts,

    support_class_masks, query_class_masks,

    support_part_masks, query_part_masks,
                                                                        n_ways,
    n_shots, n_queries, part_idx,
                                                                        img_size,
     loss_type)


                # ground truth class mask
                qry_masks_flat = torch.argmax(query_class_masks[0],dim=2).cpu().
    numpy()

                # prediction of class mask using class prototypes ('no_parts')
    or part prototypes ('parts')
                if prediction_type == 'no_parts':
                    # euclidean
```

```
                    qry_pred_flat = torch.argmin(query_pred.squeeze(dim=0),dim
    =0).cpu().numpy()

            else:
                part_idx = part_idx.cpu().numpy()
                query_pred_part = torch.reciprocal(query_pred_part)*20
                qry_pred_flat = torch.argmax(query_pred_part.squeeze(dim=0),
    dim=0).cpu().numpy()

                # maps parts to classes
                lookup = {}
                for i in range(len(part_idx)):
                    lookup[i] = part_idx[i]

                # replaces each part by its class
                for i in range(len(lookup)):
                    qry_pred_flat = np.where(qry_pred_flat==i,lookup[i],
    qry_pred_flat)


            metric.record(qry_pred_flat,
                          qry_masks_flat,
                          labels=label_ids, n_run=run)


            if (step + 1) % save_image_every == 0:
                sup_disp = sample_batched['support_images_t'][0,0,0].cpu().
    numpy().transpose(1, 2, 0)
                qry_disp = sample_batched['query_images_t'][0,0].cpu().numpy
    ().transpose(1, 2, 0)

                fig = plt.figure()
                ax1 = fig.add_subplot(1,4,1)
                ax1.set_title("Support")
                ax1.imshow(sup_disp)
                ax1.axes.xaxis.set_visible(False)
                ax1.axes.yaxis.set_visible(False)
                ax2 = fig.add_subplot(1,4,2)
                ax2.imshow(qry_disp)
                ax2.set_title("Query")
                ax2.axes.xaxis.set_visible(False)
                ax2.axes.yaxis.set_visible(False)
                ax3 = fig.add_subplot(1,4,3)
                ax3.imshow(qry_disp)
                ax3.imshow(qry_masks_flat, cmap='jet',alpha=0.5)
                ax3.axes.xaxis.set_visible(False)
                ax3.axes.yaxis.set_visible(False)
                ax3.set_title("Query True")
                ax4 = fig.add_subplot(1,4,4)
                ax4.imshow(qry_disp)
                ax4.imshow(qry_pred_flat, cmap='jet',alpha=0.5)
                ax4.axes.xaxis.set_visible(False)
                ax4.axes.yaxis.set_visible(False)
                ax4.set_title("Prediction")

                fig.savefig(chkpts_dir+current_experiment+visfolder+'/'+
```

```
                                'run'+str(run)+'image'+str(step+1)+'plot.png')
                    plt.close(fig)

            classIoU, meanIoU = metric.get_mIoU(labels=sorted(labels), n_run=run
    )
            classIoU_binary, meanIoU_binary = metric.get_mIoU_binary(n_run=run)
            print(f'classes in run: {labels}' )
            print(f'classIoU: {classIoU}')
            print(f'mean classIoU: {meanIoU}')
            print(f'binary classIoU: {classIoU_binary}')
            print(f'mean binary classIoU: {meanIoU_binary}')
            print('_____________________')
            save_testfile_run(current_experiment, run, labels,
                              classIoU, meanIoU,
                              classIoU_binary, meanIoU_binary)



    classIoU, classIoU_std, meanIoU, meanIoU_std = metric.get_mIoU(labels=sorted
    (labels))
    classIoU_binary, classIoU_std_binary, meanIoU_binary, meanIoU_std_binary =
    metric.get_mIoU_binary()
    print('##### Final Result #####')
    print(f'classIoU: {classIoU}')
    print(f'mean classIoU: {meanIoU}')
    print(f'binary classIoU: {classIoU_binary}')
    print(f'mean binary classIoU: {meanIoU_binary}')

    #save metrics in test log file
    save_testfile_final(current_experiment, labels, classIoU ,classIoU_std,
                        meanIoU, meanIoU_std,
                        classIoU_binary, classIoU_std_binary,
                        meanIoU_binary, meanIoU_std_binary)

if __name__=='__main__':
    test()
```

A.2.4 discriminative_loss.py:

```
"""
@author: A anonymous

Discriminative Loss Script:
Contains Loss function for Supervised and Multi-Dimensional Segmentation

The Supervised and Multi-dimensional loss in this function is an
adapted version of the discriminative loss function taken from
"Semantic Instance Segmentation with a Discriminative Loss Function"
by Bert De Brabandere, Davy Neven, Luc Van Gool at https://arxiv.org/abs
    /1708.02551
"""


import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
import matplotlib.pyplot as plt


# Loss parameters
var_class_target = 0
var_part_target = 0
dist_class_target = 250
dist_part_target = 0

class DiscriminativeLoss(torch.nn.Module):
    '''
    A class designed purely to run a givan loss on a batch of samples.
    input is a batch of samples (as autograd Variables) and a batch of labels (ndarrays),
    output is the average loss (as autograd variable).
    '''
    def __init__(self):
        super(DiscriminativeLoss, self).__init__()
        self.loss = self.disc_loss

    def forward(self, supp_fts, qry_fts,
                supp_class_masks, qry_class_masks,
                supp_part_masks, qry_part_masks,
                n_ways, n_shots, n_queries, part_idx, img_size,loss_type):

        pred, pred_part, var_loss, dist_loss, var_part_loss, dist_part_loss = self.loss(supp_fts, qry_fts,

            supp_class_masks, qry_class_masks,

            supp_part_masks, qry_part_masks,

            n_ways, n_shots, n_queries, part_idx, img_size,loss_type)

        return pred, pred_part, var_loss, dist_loss, var_part_loss, dist_part_loss


    def disc_loss(self, supp_fts, qry_fts,
                supp_class_masks, qry_class_masks,
                supp_part_masks, qry_part_masks,
                n_ways, n_shots, n_queries, part_idx, img_size,loss_type):
        '''
        Supervised and Multi-dimensional loss function

        Args:
            supp_fts: support features, expect shape: W x Sh x V(512) x H' x W'
            qry_fts: qry features, expect shape: Q x V(512) x H' x W'

            supp_class_mask: binary object masks for support images: # W x Sh x H x W x (C+1)
            qry_class_mask: binary object masks for query images:: # Q x H x W x (C+1)

            supp_part_mask: binary part masks for support images: # W x Sh x H x
```

```
 W x (P+1)
            qry_part_mask: binary part masks for query images:: # Q x H x W x (P
+1)

        Return:
            Loss components and predicted segmentation
        '''

        n_classes = supp_class_masks.shape[4]
        n_partclasses = supp_part_masks.shape[4]
        fts_dim = supp_fts.shape[-2:]

        # Get prototypes for each part class with vector dimension length
        # V x P
        supp_part_prototypes = self.getPrototypes(supp_fts,supp_part_masks,
n_ways,
                                                  n_shots,n_partclasses)

        # Get prototypes for each class with vector dimension length
        # V x C
        supp_class_prototypes = self.getPrototypes(supp_fts,supp_class_masks,
n_ways,
                                                   n_shots, n_classes)

        #Distance from class prototypes to query feature map
        #Q x C x H' x W'
        pred = self.calEucDist(qry_fts, supp_class_prototypes, n_classes)

        # Upsize pred is Q x C x H x W
        pred = F.interpolate(pred, size=img_size, mode='bilinear', align_corners
=False)


        ###### Compute loss ######
        var_loss = 0
        dist_loss = 0
        var_part_loss = 0
        dist_part_loss = 0

        ###### Class Var Loss - Avg. Within Cluster Distance ######
        var_loss = self.withinClusterLoss(var_loss, qry_class_masks,
                                          pred,var_class_target)


        ###### Class Dist Loss - Between Cluster Distance ######
        dist_loss = self.betweenClusterLoss(dist_loss,supp_class_prototypes,
                                            dist_class_target)

        if loss_type == 'parts':
            #Distance from part prototypes to query feature map
            #Q x C x H' x W'
            pred_part = self.calEucDist(qry_fts, supp_part_prototypes,
n_partclasses)

            # Upsize pred is Q x C x H x W
            pred_part = F.interpolate(pred_part, size=img_size, mode='bilinear',
```

```
    align_corners=False)

            ##### Var Loss - Avg. Object-Part Distance ######
            var_loss = self.withinObjectPartLoss(var_loss, supp_class_prototypes,
                                                 supp_part_prototypes, n_classes,
                                                 part_idx, var_class_target)

            ##### Part Var Loss - Avg. Within Part Cluster Distance ######
            var_part_loss = self.withinClusterLoss(var_part_loss, qry_part_masks,
                                               pred_part, var_part_target)

            ###### Part Dist Loss - Between Part Cluster Distance ######
            dist_part_loss = self.betweenPartClusterLoss(dist_part_loss, supp_part_prototypes,
                                                         dist_part_target, n_classes,part_idx)
        else:
            pred_part = torch.tensor(0).cuda()
            var_part_loss += torch.tensor(0).cuda()
            dist_part_loss += torch.tensor(0).cuda()

        return pred, pred_part, var_loss, dist_loss, var_part_loss, dist_part_loss


    def withinClusterLoss(self, var_loss, qry_mask, pred, var_target):
        """
        Calculate the average distance between features and respective prototypes
        using by applying binary masks to distance matrix and averaging

        Args:
           pred: distance matrix, expect shape: Q x (C+1) x H x W
           qry_mask: binary masks for semantic class expect shape: # Q x H x W x (C+1)
        """

        qry_mask = qry_mask.permute(0,3,1,2)
        var_loss += torch.maximum((torch.sum(pred*qry_mask) / (torch.sum(qry_mask)+ 1e-5))
                                  - var_target,
                                  torch.tensor(0).cuda())
        return var_loss


    def betweenClusterLoss(self, dist_loss,prototypes,target):
        """
        Calculate the target distance - average distance between object prototypes

        Args:
           prototype: prototype for each semantic class expect shape: V x (C+1)
           target: target distance between prototypes
```

```
        """
        n_class = prototypes.shape[1]
        cum_dist = 0
        n_conn = 0

        if n_class == 1:  # no inter cluster loss
            dist_loss += torch.tensor(0).cuda()
        else:
            for i in range(n_class):
                for j in range(i+1, n_class):
                    cum_dist +=  torch.norm(prototypes[:,i]-prototypes[:,j])
                    n_conn += 1

        if n_class==1:
            dist_loss += torch.tensor(0).cuda()
        else:
            dist_loss += torch.maximum(target - (cum_dist / n_conn),
                                       torch.tensor(0).cuda())
        return dist_loss


    def withinObjectPartLoss(self, var_loss, obj_prototypes, part_prototypes,
                             n_classes, part_idx, target):
        """
        Calculate the average distance between part prototypes and respective
        object prototypes and applying the hinge function at minimum target
    distance

        Args:
           obj_prototypes: object prototypes, expect shape: V x (C+1)
           part_prototypes: part prototypes, expect shape: V x (P+1)
        """

        # (C+1) x V
        obj_prototypes = obj_prototypes.permute(1,0)
        # (P+1) x V
        part_prototypes = part_prototypes.permute(1,0)

        for i in range(n_classes):
            # background class is excluded
            if i == 0:
                var_loss += torch.tensor(0).cuda()
            else:
            # calculate distance between part prototypes and object prototype
                part_prots = torch.index_select(part_prototypes, 0, torch.where(
    part_idx==i)[0])
                n_partsclass = part_prots.shape[0]
                obj_prot = obj_prototypes[i,:]

                cum_dist = torch.sum(torch.sqrt(torch.sum(torch.square(
    part_prots - obj_prot),dim=1)),dim=0)
                cum_dist = (cum_dist / n_partsclass).long()

                var_loss += torch.maximum(cum_dist - target,
                                          torch.tensor(0).cuda())

```

```
212         return var_loss
213
214     def betweenPartClusterLoss(self, dist_loss,prototypes,target,n_classes,
    part_idx):
215         """
216         Calculate the average distance between part prototypes
217
218         Args:
219            prototype: part prototype for each semantic class expect shape: V x (
    P+1)
220            target: target distance between part prototypes
221         """
222
223         # average part distance for each of the classes
224         cumdist_loss = []
225
226         for i in range(n_classes):
227             dist_loss_part = 0
228             cum_dist = 0
229             n_conn = 0
230
231             # background class is excluded
232             if i == 0:
233                 dist_loss_part += torch.tensor(0).cuda()
234             else:
235             # for each of the other classes calculate distance between part
    prototypes
236                 supp_chosen = torch.index_select(prototypes, 1, torch.where(
    part_idx==i)[0])
237                 n_partsclass = supp_chosen.shape[1]
238
239                 if n_partsclass==1:  # no inter cluster loss
240                     cum_dist += torch.tensor(0).cuda()
241                 else:
242                     for i in range(n_partsclass):
243                         for j in range(i+1, n_partsclass):
244                             cum_dist += torch.norm(supp_chosen[:,i]-supp_chosen
    [:,j])
245                             n_conn += + 1
246
247                 if n_partsclass==1:
248                     dist_loss_part += torch.tensor(0).cuda()
249                 else:
250                     dist_loss_part += torch.maximum(target - (cum_dist / n_conn)
    , torch.tensor(0).cuda())
251             cumdist_loss.append(dist_loss_part)
252
253         # Mean of per class average distance (excluding any zero distance)
254         cumdist_loss = torch.stack([i for i in cumdist_loss],dim=0)
255         cumdist_loss = torch.sum(cumdist_loss) / (torch.sum(torch.where(
    cumdist_loss != 0, 1, 0))+ 1e-6)
256         dist_loss += cumdist_loss
257
258         return dist_loss
259
260     def calEucDist(self, fts, prototypes, n_classes):
```

```
        """
        Calculate the distance between features and prototypes as distance matrix

        Args:
            fts: input features, expect shape: Q x V x H' x W'
            prototype: prototype of one semantic class expect shape: V x (C+1)
        """

        # Calculates euclidean distance matrix: Q x (C+1) x H' x W'
        prototypes = prototypes.permute(1,0)
        fts = fts.permute(0,2,3,1)
        pred = torch.stack([torch.sqrt(torch.sum(torch.square(fts - prototypes[i]),dim=3))
                    for i in range(n_classes)],dim=1)

        return pred


    def getPrototypes(self, supp_fts, supp_mask, n_ways, n_shots, n_partclasses):
        """
        Extract support prototypes from feature maps

        Args:
            supp_fts: input features, expect shape: way x shots x 512 x H' x W'
            supp_mask: binary mask, expect shape: way x shots x H x W x (C+1)

        """

        # Upscales feature maps through bilinear interpolation: (way x shots) x H x W x V
        supp_fts = supp_fts.view(n_ways*n_shots,supp_fts.shape[2],
                                 supp_fts.shape[3],supp_fts.shape[4])
        supp_fts = F.interpolate(supp_fts, size=supp_mask.shape[2:4],
                                 mode='bilinear',align_corners=False)
        supp_fts = supp_fts.permute(0,2,3,1)  #

        # Reshape masks: (way x shots) x H x W x (C+1)
        supp_mask = supp_mask.view(n_ways*n_shots,supp_mask.shape[2],
                                   supp_mask.shape[3],supp_mask.shape[4])

        # Applies mask to upsized feature maps for average: (way x shots) x V x (C+1)
        fts_prots = torch.stack([torch.stack([torch.sum(supp_fts[j]*
                                                        supp_mask[j][:,:,i][...,None],dim=(0, 1)) /
                                              (torch.sum(supp_mask[j][:,:,i])+ 1e-5)
                  for i in range(n_partclasses)],dim=0) for j in range(n_ways*n_shots)],dim=0)
        fts_prots = fts_prots.permute(0,2,1)

        # Applies mean across Ways and Shots: V x (C+1)
        mask_b = torch.sum(supp_mask,dim=(1,2))
        mask_b = torch.where(mask_b != 0, 1, 0)
```

```
        fts_prots = torch.sum(fts_prots,dim=0) / (torch.sum(mask_b,dim=0)+ 1e-5)

        return fts_prots
```

A.2.5 fewshot_loader.py:

```
"""
@author: A anonymous

Fewshot_Loader Script:
Loads samples in a few-shot context

Adapted code from "PANet: Few-Shot Image Semantic Segmentation with
Prototype Alignment" by Kaixin Wang et al. at https://github.com/kaixin96/PANet
"""

import os
import random
import torch
import numpy as np

from .pascal_loader import VOC, PairedDataset
from .part2obj import get_pimap


PIMAP = get_pimap()

def voc_fewshot(base_dir, split, transforms, to_tensor, labels, n_ways, n_shots,
     max_iters,
                n_queries=1):
    """
    Args:
        base_dir:
            VOC dataset directory
        split:
            which split to use
            choose from ('train', 'val', 'trainval', 'trainaug')
        transform:
            transformations to be performed on images/masks
        to_tensor:
            transformation to convert PIL Image to tensor
        labels:
            object class labels of the data
        n_ways:
            n-way few-shot learning, should be no more than # of object class
    labels
        n_shots:
            n-shot few-shot learning
        max_iters:
            number of pairs
        n_queries:
            number of query images
    """
    voc = VOC(base_dir=base_dir, split=split, transforms=transforms, to_tensor=
    to_tensor)
```

```
    voc.add_attrib('basic', attrib_basic, {})

    # Load image ids for each class
    sub_ids = []
    for label in labels:
        with open(os.path.join(voc._id_dir, voc.split,
                               'class{}.txt'.format(label)), 'r') as f:
            sub_ids.append(f.read().splitlines())
    # Create sub-datasets and add class_id attribute
    subsets = voc.subsets(sub_ids, [{'basic': {'class_id': cls_id}} for cls_id
    in labels])

    # Choose the classes of queries
    cnt_query = np.bincount(random.choices(population=range(n_ways), k=n_queries
    ), minlength=n_ways)
    # Set the number of images for each class
    n_elements = [n_shots + x for x in cnt_query]
    # Create paired dataset
    paired_data = PairedDataset(subsets, n_elements=n_elements, max_iters=
    max_iters, same=False,
                                pair_based_transforms=[
                                    (fewShot, {'n_ways': n_ways, 'n_shots':
    n_shots,
                                               'cnt_query': cnt_query})])
    return paired_data

def fewShot(paired_sample, n_ways, n_shots, cnt_query):
    """
    Postprocess paired sample for fewshot settings

    Args:
        paired_sample:
            data sample from a PairedDataset
        n_ways:
            n-way few-shot learning
        n_shots:
            n-shot few-shot learning
        cnt_query:
            number of query images for each class in the support set
        coco:
            MS COCO dataset
    """
    ###### Compose the support and query image list ######
    cumsum_idx = np.cumsum([0,] + [n_shots + x for x in cnt_query])
    # print('CUMSUM',cumsum_idx)

    # support class ids
    class_ids = [paired_sample[cumsum_idx[i]]['basic_class_id'] for i in range(
    n_ways)]

    # support part ids
    part_ids = [PIMAP[class_ids[i]]['index'] for i in range(n_ways)]
    part_ids_full = [part for subpartlist in part_ids for part in subpartlist]
    part_class = [[class_ids[i] for j in range(len(part_ids[i]))] for i in range
    (n_ways)]
    part_classidx = [[i+1 for j in range(len(part_ids[i]))] for i in range(
```

```
    n_ways)]
97  part_classidx_full = [idx for subpartlist in part_classidx for idx in
    subpartlist]
98
99  #New - All Tensors
100 # support images: ways x shots x 3 x H x W
101 support_images = torch.stack([torch.stack([paired_sample[cumsum_idx[i] + j][
    'image']
102                                             for j in range(n_shots)],dim=0)
    for i in range(n_ways)],dim=0)
103
104 # support images(non-normalized): ways x shots x 3 x H x W
105 support_images_t = torch.stack([torch.stack([paired_sample[cumsum_idx[i] + j
    ]['image_t']
106                                             for j in range(n_shots)],dim=0)
    for i in range(n_ways)],dim=0)
107
108 # support image class labels (including other classes): way x shots x H x W
109 support_class_labels = torch.stack([torch.stack([paired_sample[cumsum_idx[i]
     + j]['cls_labels']
110                                             for j in range(n_shots)],dim=0)
    for i in range(n_ways)],dim=0)
111
112 # support image part labels (including other classes): way x shots x H x W
113 support_part_labels = torch.stack([torch.stack([paired_sample[cumsum_idx[i]
    + j]['part_labels']
114                                             for j in range(n_shots)],dim=0)
    for i in range(n_ways)],dim=0)
115
116 # query images: Q x 3 x H x W
117 query_images = torch.stack([paired_sample[cumsum_idx[i+1] - j - 1]['image']
    for i in range(n_ways)
118                 for j in range(cnt_query[i])],dim=0)
119
120 # query images(non-normalized): Q x 3 x H x W
121 query_images_t = torch.stack([paired_sample[cumsum_idx[i+1] - j - 1]['
    image_t'] for i in range(n_ways)
122                 for j in range(cnt_query[i])],dim=0)
123
124 # query part labels (including other classes): Q x H x W
125 query_class_labels = torch.stack([paired_sample[cumsum_idx[i+1] - j - 1]['
    cls_labels'] for i in range(n_ways)
126                 for j in range(cnt_query[i])],dim=0)
127
128 # query part labels (including other classes): Q x H x W
129 query_part_labels = torch.stack([paired_sample[cumsum_idx[i+1] - j - 1]['
    part_labels'] for i in range(n_ways)
130                 for j in range(cnt_query[i])],dim=0)
131
132 # support class mask (only classes in way): way x shot x H x W x (C+1) (
    where C is no. classes +1 for background)
133 # each layer of mask has 1s for relevant class pixels and 0 otherwise
134 supp_class_mask = torch.stack([torch.stack([classMask(support_class_labels[
    way,shot,:,:], class_ids)[0]
135                                             for shot in range(n_shots)],dim
    =0) for way in range(n_ways)],dim=0)
```

```
136
137     # query part mask only that class: Q x H x W x (C+1)
138     query_class_mask = torch.stack([classMask(paired_sample[cumsum_idx[i+1] - j
        - 1]['cls_labels'],
139                                     class_ids)[0] for i in range(n_ways) for j in
        range(cnt_query[i])],dim=0)
140
141     # support part mask only those classes in way: way x shot x H x W x (C+1) (
        where C is no. classes +1 for background)
142     # each layer of mask has 1s for relevant class pixels and 0 otherwise
143     supp_part_mask = torch.stack([torch.stack([partMask(support_part_labels[way,
        shot,:,:], part_ids_full)[0]
144                                                 for shot in range(n_shots)],dim
        =0) for way in range(n_ways)],dim=0)
145
146     # query part mask only that class: Q x H x W x (C+1)
147     query_part_mask = torch.stack([partMask(paired_sample[cumsum_idx[i+1] - j -
        1]['part_labels'],
148                                     part_ids_full)[0] for i in range(n_ways) for j
         in range(cnt_query[i])],dim=0)
149
150
151     return {'class_ids': class_ids,
152             'part_ids': part_ids,
153             'part_ids_full': part_ids_full,
154             'part_class': part_class,
155             'part_classidx_full': part_classidx_full,
156
157             'support_images': support_images,
158             'support_images_t': support_images_t,
159             'support_class_labels': support_class_labels,
160             'support_part_labels': support_part_labels,
161             'support_class_mask': supp_class_mask,
162             'support_part_mask': supp_part_mask,
163
164             'query_images': query_images,
165             'query_images_t': query_images_t,
166             'query_class_labels': query_class_labels,
167             'query_part_labels': query_part_labels,
168             'query_class_mask': query_class_mask,
169             'query_part_mask': query_part_mask
170
171           }
172
173 def attrib_basic(_sample, class_id):
174     """
175     Add basic attribute
176
177     Args:
178         _sample: data sample
179         class_id: class label asscociated with the data
180             (sometimes indicting from which subset the data are drawn)
181     """
182     return {'class_id': class_id}
183
184
```

```
def classMask(classlabel, class_ids):

    # H x W x (C+1) array with each of the (C+1) object masks
    mask = torch.zeros_like(classlabel).unsqueeze(dim=2)
    for i in class_ids:
        classmask = torch.where(classlabel == i, 1, 0).unsqueeze(dim=2)
        mask = torch.cat([mask,classmask],dim=2)

    # HxW object mask with each object indexed to pbject ids array
    mask_flat = torch.argmax(mask,dim=2)

    # Converts the background class layer into a background mask
    mask[:,:,0] = torch.where(mask_flat == 0, 1, 0)

    return mask, mask_flat

def partMask(partlabel, part_ids):

    # H x W x (P+1) array with each of the (P+1) part masks
    mask = torch.zeros_like(partlabel).unsqueeze(dim=2)
    for i in part_ids:
        partmask = torch.where(partlabel == i, 1, 0).unsqueeze(dim=2)
        mask = torch.cat([mask,partmask],dim=2)

    # HxW part mask with each part indexed to part ids array
    mask_flat = torch.argmax(mask,dim=2)

    # Converts the background class layer into a background mask
    mask[:,:,0] = torch.where(mask_flat == 0, 1, 0)

    return mask, mask_flat
```

A.2.6 pascal_loader.py:

```
"""
@author: A anonymous

pascal_loader Script:
Load pascal part base dataset

Adapted code from "PANet: Few-Shot Image Semantic Segmentation with
Prototype Alignment" by Kaixin Wang et al. at https://github.com/kaixin96/PANet
"""


import os
import numpy as np
from PIL import Image
import torch
import random
from scipy.io import loadmat
from skimage.io import imread
from skimage.measure import regionprops
from torch.utils.data import Dataset

```

```
from .part2obj import get_pimap


PIMAP = get_pimap()

class BaseDataset(Dataset):
    """
    Base Dataset

    Args:
        base_dir:
            dataset directory
    """
    def __init__(self, base_dir):
        self._base_dir = base_dir
        self.aux_attrib = {}
        self.aux_attrib_args = {}
        self.ids = []  # must be overloaded in subclass

    def add_attrib(self, key, func, func_args):
        """
        Add attribute to the data sample dict

        Args:
            key:
                key in the data sample dict for the new attribute
                e.g. sample['click_map'], sample['depth_map']
            func:
                function to process a data sample and create an attribute (e.g.
    user clicks)
            func_args:
                extra arguments to pass, expected a dict
        """
        if key in self.aux_attrib:
            raise KeyError("Attribute '{0}' already exists, please use '
    set_attrib'.".format(key))
        else:
            self.set_attrib(key, func, func_args)

    def set_attrib(self, key, func, func_args):
        """
        Set attribute in the data sample dict

        Args:
            key:
                key in the data sample dict for the new attribute
                e.g. sample['click_map'], sample['depth_map']
            func:
                function to process a data sample and create an attribute (e.g.
    user clicks)
            func_args:
                extra arguments to pass, expected a dict
        """
        self.aux_attrib[key] = func
        self.aux_attrib_args[key] = func_args

```

```
    def del_attrib(self, key):
        """
        Remove attribute in the data sample dict

        Args:
            key:
                key in the data sample dict
        """
        self.aux_attrib.pop(key)
        self.aux_attrib_args.pop(key)

    def subsets(self, sub_ids, sub_args_lst=None):
        """
        Create subsets by ids

        Args:
            sub_ids:
                a sequence of sequences, each sequence contains data ids for one subset
            sub_args_lst:
                a list of args for some subset-specific auxiliary attribute function
        """

        indices = [[self.ids.index(id_) for id_ in ids] for ids in sub_ids]
        if sub_args_lst is not None:
            subsets = [Subset(dataset=self, indices=index, sub_attrib_args=args)
                       for index, args in zip(indices, sub_args_lst)]
        else:
            subsets = [Subset(dataset=self, indices=index) for index in indices]
        return subsets

    def __len__(self):
        pass

    def __getitem__(self, idx):
        pass


class VOC(BaseDataset):
    """
    Base Class for VOC Dataset

    Args:
        base_dir:
            VOC dataset directory
        split:
            which split to use
            choose from ('train', 'val', 'trainval', 'trainaug')
        transform:
            transformations to be performed on images/masks
        to_tensor:
            transformation to convert PIL Image to tensor
    """
    def __init__(self, base_dir, split, transforms=None, to_tensor=None):
        super().__init__(base_dir)
```

```
        self.split = split
        self._image_dir = os.path.join(self._base_dir, 'JPEGImages')
        self._label_dir = os.path.join(self._base_dir, 'SegmentationParts')
        self._id_dir = os.path.join(self._base_dir, 'ImageSets', 'Segmentation')
        self.transforms = transforms
        self.to_tensor = to_tensor

        with open(os.path.join(self._id_dir, f'{self.split}.txt'), 'r') as f:
            self.ids = f.read().splitlines()

    def __len__(self):
        return len(self.ids)

    def __getitem__(self, idx):
        # Fetch data
        id_ = self.ids[idx]
        image = Image.open(os.path.join(self._image_dir, f'{id_}.jpg'))
        imsize = np.array(image).shape
        data = loadmat(os.path.join(self._label_dir,
                                    f'{id_}.mat'))['anno'][0, 0]
        n_objects = data['objects'].shape[1]

        objects = []
        for obj in data['objects'][0, :]:
            objects.append(PascalObject(obj))

        cls_labels, inst_labels, part_labels = self._mat2map(imsize,objects)

        sample = {'id': id_,
                  'image': image,
                  'cls_labels':cls_labels,
                  'part_labels':part_labels}

        # Image-level transformation (Resizing)
        if self.transforms is not None:
            sample = self.transforms(sample)

        image_t = np.array(sample['image']).transpose(2, 0, 1)
        sample['image_t'] = image_t

        # Transform to tensor
        if self.to_tensor is not None:
            sample = self.to_tensor(sample)
            image_t = torch.from_numpy(image_t)
            sample['image_t'] = image_t

        # Add auxiliary attributes
        for key_prefix in self.aux_attrib:
            # Process the data sample, create new attributes and save them in a
    dictionary
            aux_attrib_val = self.aux_attrib[key_prefix](sample, **self.
    aux_attrib_args[key_prefix])
            for key_suffix in aux_attrib_val:
                # one function may create multiple attributes, so we need suffix
     to distinguish them
                sample[key_prefix + '_' + key_suffix] = aux_attrib_val[
```

```
    key_suffix]

        return sample

    def load_image(self, idx):
        id_ = self.ids[idx]
        image = Image.open(os.path.join(self._image_dir, f'{id_}.jpg'))
        return image


    def _mat2map(self,imsize,objects):
        ''' Create masks from the annotations
        Python implementation based on
        http://www.stat.ucla.edu/~xianjie.chen/pascal_part_dataset/trainval.tar.gz

        Read the annotation and present it in terms of 3 segmentation mask maps (
        i.e., the class maks, instance maks and part mask). pimap defines a
        mapping between part name and index (See part2ind.py).
        '''

        shape = imsize[:-1]  # first two dimensions, ignore color channel
        cls_mask = np.zeros(shape, dtype=np.uint8)
        inst_mask = np.zeros(shape, dtype=np.uint8)
        part_mask = np.zeros(shape, dtype=np.uint8)
        for i, obj in enumerate(objects):
            class_ind = obj.class_ind
            mask = obj.mask

            inst_mask[mask > 0] = i + 1
            cls_mask[mask > 0] = class_ind

            if obj.n_parts > 0:
                for p in obj.parts:
                    part_name = p.part_name
                    pid = PIMAP[class_ind][part_name]
                    part_mask[p.mask > 0] = pid

            # For classes without parts the whole object is segmented
            if class_ind in [4,9,11,18]:
                part_mask[mask > 0] = PIMAP[class_ind]['main']

        return cls_mask, inst_mask, part_mask

class PascalBase(object):
    def __init__(self, obj):
        self.mask = obj['mask']


class PascalObject(PascalBase):
    def __init__(self, obj):
        super(PascalObject, self).__init__(obj)

        self.class_name = obj['class'][0]
        self.class_ind = obj['class_ind'][0, 0]
```

```
235
236         self.n_parts = obj['parts'].shape[1]
237         self.parts = []
238         if self.n_parts > 0:
239             for part in obj['parts'][0, :]:
240                 self.parts.append(PascalPart(part))
241
242 class PascalPart(PascalBase):
243     def __init__(self, obj):
244         super(PascalPart, self).__init__(obj)
245         self.part_name = obj['part_name'][0]
246
247
248
249 class PairedDataset(Dataset):
250     """
251     Make pairs of data from dataset
252
253     When 'same=True',
254         a pair contains data from same datasets,
255         and the choice of datasets for each pair is random.
256         e.g. [[ds1_3, ds1_2], [ds3_1, ds3_2], [ds2_1, ds2_2], ...]
257     When 'same=False',
258             a pair contains data from different datasets,
259             if 'n_elements' <= # of datasets, then we randomly choose a subset
    of datasets,
260                 then randomly choose a sample from each dataset in the subset
261                 e.g. [[ds1_3, ds2_1, ds3_1], [ds4_1, ds2_3, ds3_2], ...]
262             if 'n_element' is a list of int, say [C_1, C_2, C_3, ..., C_k], we
    first
263                 randomly choose k(k < # of datasets) datasets, then draw C_1,
    C_2, ..., C_k samples
264                 from each dataset respectively.
265                 Note the total number of elements will be (C_1 + C_2 + ... + C_k
    ).
266
267     Args:
268         datasets:
269             source datasets, expect a list of Dataset
270         n_elements:
271             number of elements in a pair
272         max_iters:
273             number of pairs to be sampled
274         same:
275             whether data samples in a pair are from the same dataset or not,
276             see a detailed explanation above.
277         pair_based_transforms:
278             some transformation performed on a pair basis, expect a list of
    functions,
279             each function takes a pair sample and return a transformed one.
280     """
281     def __init__(self, datasets, n_elements, max_iters, same=True,
282                  pair_based_transforms=None):
283         super().__init__()
284         self.datasets = datasets
285         self.n_datasets = len(self.datasets)
```

```
        self.n_data = [len(dataset) for dataset in self.datasets]
        self.n_elements = n_elements
        self.max_iters = max_iters
        self.pair_based_transforms = pair_based_transforms
        if same:
            if isinstance(self.n_elements, int):
                datasets_indices = [random.randrange(self.n_datasets)
                                    for _ in range(self.max_iters)]
                self.indices = [[(dataset_idx, data_idx)
                                 for data_idx in random.choices(range(self.
    n_data[dataset_idx]),
                                                                k=self.
    n_elements)]
                                for dataset_idx in datasets_indices]
            else:
                raise ValueError("When 'same=true', 'n_element' should be an
    integer.")
        else:
            if isinstance(self.n_elements, list):
                self.indices = [[(dataset_idx, data_idx)
                                 for i, dataset_idx in enumerate(
                                     random.sample(range(self.n_datasets), k=len
    (self.n_elements)))
                                 for data_idx in random.sample(range(self.n_data
    [dataset_idx]),
                                                               k=self.n_elements
    [i])]
                                for i_iter in range(self.max_iters)]
            elif self.n_elements > self.n_datasets:
                raise ValueError("When 'same=False', 'n_element' should be no
    more than n_datasets")
            else:
                self.indices = [[(dataset_idx, random.randrange(self.n_data[
    dataset_idx]))
                                 for dataset_idx in random.sample(range(self.
    n_datasets),
                                                                  k=n_elements)]
                                for i in range(max_iters)]

    def __len__(self):
        return self.max_iters

    def __getitem__(self, idx):
        sample = [self.datasets[dataset_idx][data_idx]
                  for dataset_idx, data_idx in self.indices[idx]]
        if self.pair_based_transforms is not None:
            for transform, args in self.pair_based_transforms:
                sample = transform(sample, **args)
        return sample


class Subset(Dataset):
    """
    Subset of a dataset at specified indices.

    Args:
```

```
        dataset:
            The whole Dataset
        indices:
            Indices in the whole set selected for subset
        sub_attrib_args:
            Subset-specific arguments for attribute functions, expected a dict
    """
    def __init__(self, dataset, indices, sub_attrib_args=None):
        self.dataset = dataset
        self.indices = indices
        self.sub_attrib_args = sub_attrib_args

    def __getitem__(self, idx):
        if self.sub_attrib_args is not None:
            for key in self.sub_attrib_args:
                # Make sure the dataset already has the corresponding attributes
                # Here we only make the arguments subset dependent
                #   (i.e. pass different arguments for each subset)
                self.dataset.aux_attrib_args[key].update(self.sub_attrib_args[
    key])
        return self.dataset[self.indices[idx]]

    def __len__(self):
        return len(self.indices)
```

A.2.7 part2obj.py:

```
"""
@author: A anonymous

Part2obj Script:
Map objects to respective parts

Adapted code from Stavros Tsogkas - https://github.com/tsogkas/pascal-part-
    classes
and with Python implementation based on http://www.stat.ucla.edu/~xianjie.chen/
    pascal_part_dataset/trainval.tar.gz

"""

def get_class_names():
    classes = {1: 'aeroplane',
               2: 'bicycle',
               3: 'bird',
               4: 'boat',
               5: 'bottle',
               6: 'bus',
               7: 'car',
               8: 'cat',
               9: 'chair',
               10: 'cow',
               11: 'table',
               12: 'dog',
               13: 'horse',
               14: 'motorbike',
```

```
                15: 'person',
                16: 'pottedplant',
                17: 'sheep',
                18: 'sofa',
                19: 'train',
                20: 'tvmonitor'}
    return classes


def get_pimap():
    pimap = {}

    # [aeroplane]
    pimap[1] = {}
    pimap[1]['body']        = 1
    pimap[1]['stern']       = 1
    pimap[1]['lwing']       = 2                 # left wing
    pimap[1]['rwing']       = 2                 # right wing
    pimap[1]['tail']        = 1
    for ii in range(1, 10 + 1):
        pimap[1][('engine_%d' % ii)] = 2 # multiple engines
    for ii in range(1, 10 + 1):
        pimap[1][('wheel_%d' % ii)] = 1  # multiple wheels
    pimap[1]['index'] = [1,2]

    # [bicycle]
    pimap[2] = {}
    pimap[2]['fwheel']      = 3                 # front wheel
    pimap[2]['bwheel']      = 3                 # back wheel
    pimap[2]['saddle']      = 4
    pimap[2]['handlebar']   = 4                 # handle bar
    pimap[2]['chainwheel']  = 4                 # chain wheel
    for ii in range(1, 10 + 1):
        pimap[2][('headlight_%d' % ii)] = 4
    pimap[2]['index'] = [3,4]

    # [bird]
    pimap[3] = {}
    pimap[3]['head']        = 5
    pimap[3]['leye']        = 5                 # left eye
    pimap[3]['reye']        = 5                 # right eye
    pimap[3]['beak']        = 5
    pimap[3]['torso']       = 5
    pimap[3]['neck']        = 5
    pimap[3]['lwing']       = 6                 # left wing
    pimap[3]['rwing']       = 6                 # right wing
    pimap[3]['lleg']        = 7                 # left leg
    pimap[3]['lfoot']       = 7                # left foot
    pimap[3]['rleg']        = 7                # right leg
    pimap[3]['rfoot']       = 7                # right foot
    pimap[3]['tail']        = 5
    pimap[3]['index'] = [5,6,7]

    # [boat]
    # only has silhouette mask
    pimap[4] = {}
```

```
    pimap[4]['main'] = 8
    pimap[4]['index'] = [8]

    # [bottle]
    pimap[5] = {}
    pimap[5]['cap']          = 9
    pimap[5]['body']         = 10
    pimap[5]['index'] = [9,10]

    # [bus]
    pimap[6] = {}
    pimap[6]['frontside']    = 11
    pimap[6]['leftside']     = 11
    pimap[6]['rightside']    = 11
    pimap[6]['backside']     = 11
    pimap[6]['roofside']     = 11
    pimap[6]['leftmirror']   = 11
    pimap[6]['rightmirror'] = 11
    pimap[6]['fliplate']     = 11                # front license plate
    pimap[6]['bliplate']     = 11                # back license plate
    for ii in range(1, 10 + 1):
        pimap[6][('door_%d' % ii)] = 11
    for ii in range(1, 10 + 1):
        pimap[6][('wheel_%d' % ii)] = 12
    for ii in range(1, 10 + 1):
        pimap[6][('headlight_%d' % ii)] = 11
    for ii in range(1, 20 + 1):
        pimap[6][('window_%d' % ii)] = 13
    pimap[6]['index'] = [11,12,13]


    # [car]         # car has the same set of parts with bus
    pimap[7] = {}
    pimap[7]['frontside']    = 14
    pimap[7]['leftside']     = 14
    pimap[7]['rightside']    = 14
    pimap[7]['backside']     = 14
    pimap[7]['roofside']     = 14
    pimap[7]['leftmirror']   = 14
    pimap[7]['rightmirror'] = 14
    pimap[7]['fliplate']     = 14                # front license plate
    pimap[7]['bliplate']     = 14                # back license plate
    for ii in range(1, 10 + 1):
        pimap[7][('door_%d' % ii)] = 14
    for ii in range(1, 10 + 1):
        pimap[7][('wheel_%d' % ii)] = 15
    for ii in range(1, 10 + 1):
        pimap[7][('headlight_%d' % ii)] = 14
    for ii in range(1, 20 + 1):
        pimap[7][('window_%d' % ii)] = 16
    pimap[7]['index'] = [14,15,16]

    # [cat]
    pimap[8] = {}
    pimap[8]['head']         = 17
    pimap[8]['leye']         = 17                # left eye
```

```
    pimap[8]['reye']        = 17                # right eye
    pimap[8]['lear']        = 17                # left ear
    pimap[8]['rear']        = 17                # right ear
    pimap[8]['nose']        = 17
    pimap[8]['torso']       = 18
    pimap[8]['neck']        = 18
    pimap[8]['lfleg']       = 19                # left front leg
    pimap[8]['lfpa']        = 19               # left front paw
    pimap[8]['rfleg']       = 19               # right front leg
    pimap[8]['rfpa']        = 19               # right front paw
    pimap[8]['lbleg']       = 19               # left back leg
    pimap[8]['lbpa']        = 19               # left back paw
    pimap[8]['rbleg']       = 19               # right back leg
    pimap[8]['rbpa']        = 19               # right back paw
    pimap[8]['tail']        = 20
    pimap[8]['index'] = [17,18,19,20]

    # [chair]
    # only has sihouette mask
    pimap[9] = {}
    pimap[9]['main'] = 21
    pimap[9]['index'] = [21]

    # [cow]
    pimap[10] = {}
    pimap[10]['head']       = 22
    pimap[10]['leye']       = 22                # left eye
    pimap[10]['reye']       = 22                # right eye
    pimap[10]['lear']       = 22                # left ear
    pimap[10]['rear']       = 22                # right ear
    pimap[10]['muzzle']     = 22
    pimap[10]['lhorn']      = 22                # left horn
    pimap[10]['rhorn']      = 22                # right horn
    pimap[10]['torso']      = 23
    pimap[10]['neck']       = 23
    pimap[10]['lfuleg']     = 24               # left front upper leg
    pimap[10]['lflleg']     = 24               # left front lower leg
    pimap[10]['rfuleg']     = 24               # right front upper leg
    pimap[10]['rflleg']     = 24               # right front lower leg
    pimap[10]['lbuleg']     = 24               # left back upper leg
    pimap[10]['lblleg']     = 24               # left back lower leg
    pimap[10]['rbuleg']     = 24               # right back upper leg
    pimap[10]['rblleg']     = 24               # right back lower leg
    pimap[10]['tail']       = 25
    pimap[10]['index'] = [22,23,24,25]

    # [table]
    # only has silhouette mask
    pimap[11] = {}
    pimap[11]['main'] = 26
    pimap[11]['index'] = [26]


    # [dog]
    # dog has the same set of parts with cat,
    pimap[12] = {}
```

```
    pimap[12]['head']        = 27
    pimap[12]['leye']        = 27                # left eye
    pimap[12]['reye']        = 27                # right eye
    pimap[12]['lear']        = 27                # left ear
    pimap[12]['rear']        = 27                # right ear
    pimap[12]['nose']        = 27
    pimap[12]['torso']       = 28
    pimap[12]['neck']        = 28
    pimap[12]['lfleg']       = 29                # left front leg
    pimap[12]['lfpa']        = 29               # left front paw
    pimap[12]['rfleg']       = 29               # right front leg
    pimap[12]['rfpa']        = 29               # right front paw
    pimap[12]['lbleg']       = 29               # left back leg
    pimap[12]['lbpa']        = 29               # left back paw
    pimap[12]['rbleg']       = 29               # right back leg
    pimap[12]['rbpa']        = 29               # right back paw
    pimap[12]['tail']        = 30
    pimap[12]['muzzle']     = 27            # except for the additionalmuzzle
    pimap[12]['index'] = [27,28,29,30]


    # [horse]
    # horse has the same set of parts with cow,
    # except it has hoof instead of horn
    pimap[13] = {}
    pimap[13]['head']       = 31
    pimap[13]['leye']       = 31                # left eye
    pimap[13]['reye']       = 31                # right eye
    pimap[13]['lear']       = 31                # left ear
    pimap[13]['rear']       = 31                # right ear
    pimap[13]['muzzle']     = 31
    pimap[13]['torso']      = 32
    pimap[13]['neck']       = 32
    pimap[13]['lfuleg']     = 33               # left front upper leg
    pimap[13]['lflleg']     = 33               # left front lower leg
    pimap[13]['rfuleg']     = 33               # right front upper leg
    pimap[13]['rflleg']     = 33               # right front lower leg
    pimap[13]['lbuleg']     = 33               # left back upper leg
    pimap[13]['lblleg']     = 33               # left back lower leg
    pimap[13]['rbuleg']     = 33               # right back upper leg
    pimap[13]['rblleg']     = 33               # right back lower leg
    pimap[13]['tail']       = 34
    pimap[13]['lfho']       = 33
    pimap[13]['rfho']       = 33
    pimap[13]['lbho']       = 33
    pimap[13]['rbho']       = 33
    pimap[13]['index'] = [31,32,33,34]

    # [motorbike]
    pimap[14] = {}
    pimap[14]['fwheel']     = 35
    pimap[14]['bwheel']     = 35
    pimap[14]['handlebar']  = 36
    pimap[14]['saddle']     = 36
    for ii in range(1, 10 + 1):
        pimap[14][('headlight_%d' % ii)] = 36
```

```
    pimap[14]['index'] = [35,36]

    # [person]
    pimap[15] = {}
    pimap[15]['head']       = 37
    pimap[15]['leye']       = 37                    # left eye
    pimap[15]['reye']       = 37                     # right eye
    pimap[15]['lear']       = 37                     # left ear
    pimap[15]['rear']       = 37                     # right ear
    pimap[15]['lebrow']     = 37                     # left eyebrow
    pimap[15]['rebrow']     = 37                     # right eyebrow
    pimap[15]['nose']       = 37
    pimap[15]['mouth']      = 37
    pimap[15]['hair']       = 37

    pimap[15]['torso']      = 38
    pimap[15]['neck']       = 38

    pimap[15]['llarm']      = 39                    # left lower arm
    pimap[15]['luarm']      = 39                    # left upper arm
    pimap[15]['lhand']      = 39                    # left hand
    pimap[15]['rlarm']      = 39                    # right lower arm
    pimap[15]['ruarm']      = 39                    # right upper arm
    pimap[15]['rhand']      = 39                    # right hand

    pimap[15]['llleg']      = 40                # left lower leg
    pimap[15]['luleg']      = 40                # left upper leg
    pimap[15]['lfoot']      = 40                # left foot
    pimap[15]['rlleg']      = 40                # right lower leg
    pimap[15]['ruleg']      = 40                # right upper leg
    pimap[15]['rfoot']      = 40                # right foot
    pimap[15]['index'] = [37,38,39,40]

    # [pottedplant]
    pimap[16] = {}
    pimap[16]['pot']        = 41
    pimap[16]['plant']      = 42
    pimap[16]['index'] = [41,42]

    # [sheep]
    # sheep has the same set of parts with cow
    pimap[17] = {}
    pimap[17]['head']       = 43
    pimap[17]['leye']       = 43                # left eye
    pimap[17]['reye']       = 43                # right eye
    pimap[17]['lear']       = 43                # left ear
    pimap[17]['rear']       = 43                # right ear
    pimap[17]['muzzle']     = 43
    pimap[17]['lhorn']      = 43                # left horn
    pimap[17]['rhorn']      = 43                # right horn
    pimap[17]['torso']      = 44
    pimap[17]['neck']       = 44
    pimap[17]['lfuleg']     = 45              # left front upper leg
    pimap[17]['lflleg']     = 45              # left front lower leg
    pimap[17]['rfuleg']     = 45              # right front upper leg
    pimap[17]['rflleg']     = 45              # right front lower leg
```

```
    pimap[17]['lbuleg']      = 45                # left back upper leg
    pimap[17]['lblleg']      = 45                # left back lower leg
    pimap[17]['rbuleg']      = 45                # right back upper leg
    pimap[17]['rblleg']      = 45                # right back lower leg
    pimap[17]['tail']        = 46
    pimap[17]['index'] = [43,44,45,46]

    # [sofa]
    # only has sihouette mask
    pimap[18] = {}
    pimap[18]['main'] = 47
    pimap[18]['index'] = [47]

    # [train]
    pimap[19] = {}
    pimap[19]['head']        = 48
    pimap[19]['hfrontside'] = 48                   # head front side
    pimap[19]['hleftside']  = 48                   # head left side
    pimap[19]['hrightside'] = 48                   # head right side
    pimap[19]['hbackside']  = 48                  # head back side
    pimap[19]['hroofside']  = 48                   # head roof side

    for ii in range(1, 10 + 1):
        pimap[19][('headlight_%d' % ii)] = 48

    for ii in range(1, 10 + 1):
        pimap[19][('coach_%d' % ii)] = 49

    for ii in range(1, 10 + 1):
        pimap[19][('cfrontside_%d' % ii)] = 49   # coach front side

    for ii in range(1, 10 + 1):
        pimap[19][('cleftside_%d' % ii)] = 49   # coach left side

    for ii in range(1, 10 + 1):
        pimap[19][('crightside_%d' % ii)] = 49  # coach right side

    for ii in range(1, 10 + 1):
        pimap[19][('cbackside_%d' % ii)] = 49   # coach back side

    for ii in range(1, 10 + 1):
        pimap[19][('croofside_%d' % ii)] = 49   # coach roof side
    pimap[19]['index'] = [48,49]

    # [tvmonitor]
    pimap[20] = {}
    pimap[20]['screen']      = 50
    pimap[20]['index'] = [50]

    return pimap
```

A.2.8 transforms.py:

```
"""
@author: A anonymous
```

```
3
4  Transforms Script:
5  Data transforms for resizing and normalisation of images
6
7  Adapted code from "PANet: Few-Shot Image Semantic Segmentation with
8  Prototype Alignment" by Kaixin Wang et al. at https://github.com/kaixin96/PANet
9  """
10
11 import random
12
13 from PIL import Image
14 from scipy import ndimage
15 import numpy as np
16 import torch
17 import torch.nn.functional as torfunc
18 import torchvision.transforms.functional as tr_F
19
20
21 class Resize(object):
22     """
23     Resize images/masks to given size
24
25     Args:
26         size: output size
27     """
28     def __init__(self, size):
29         self.size = size
30
31     def __call__(self, sample):
32         img, cls_labels, part_labels =  sample['image'], sample['cls_labels'],
       sample['part_labels']
33
34         img = tr_F.resize(img, self.size)
35         cls_labels = torfunc.interpolate(torch.Tensor(cls_labels).unsqueeze(dim
       =0).unsqueeze(dim=0),
36                                          self.size, mode='nearest').squeeze().
       squeeze().numpy()
37         part_labels = torfunc.interpolate(torch.Tensor(part_labels).unsqueeze(
       dim=0).unsqueeze(dim=0),
38                                          self.size, mode='nearest').squeeze().
       squeeze().numpy()
39
40         sample['image'] = img
41         sample['cls_labels'] = cls_labels
42         sample['part_labels'] = part_labels
43         return sample
44
45 class ToTensorNormalize(object):
46     """
47     Convert images/masks to torch.Tensor
48     Scale images' pixel values to [0-1] and normalize with predefined statistics
49     """
50     def __call__(self, sample):
51
52         img, cls_labels, part_labels = sample['image'], sample['cls_labels'],
       sample['part_labels']
```

```
53
54          img = tr_F.to_tensor(img)
55          img = tr_F.normalize(img, mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
        0.225])
56          cls_labels = torch.Tensor(cls_labels).long()
57          part_labels = torch.Tensor(part_labels).long()
58
59          sample['image'] = img
60          sample['cls_labels'] = cls_labels
61          sample['part_labels'] = part_labels
62          return sample
```

A.2.9 network.py:

```
1  """
2  @author: A anonymous
3
4  Network Script:
5  Code to load model architecture and pre-trained models
6
7  Adapted code from "PANet: Few-Shot Image Semantic Segmentation with
8  Prototype Alignment" by Kaixin Wang et al. at https://github.com/kaixin96/PANet
9  """
10
11 from collections import OrderedDict
12
13 import torch
14 import torch.nn as nn
15 import torch.nn.functional as F
16
17 from .vgg import Encoder
18 from .resnet import resnet50 as Net50
19
20 class FewShotNet(nn.Module):
21     """
22     Fewshot Segmentation model
23
24     Args:
25         in_channels:
26             number of input channels
27         pretrained_path:
28             path of the model for initialization
29         cfg:
30             model configurations
31         model_arch:
32             model architecture (VGG16 or ResNet50)
33     """
34     def __init__(self, in_channels=3, pretrained_path=None, cfg=None, model_arch
        = None):
35         super().__init__()
36         self.pretrained_path = pretrained_path
37         self.config = cfg or {'align': False}
38         self.model_arch = model_arch
39
40         # Encoder
```

```
        if self.model_arch == 'ResNet50':
            self.encoder = nn.Sequential(OrderedDict([('backbone', Net50(cfg=cfg
    )),]))

        else:
            self.encoder = nn.Sequential(OrderedDict([
                ('backbone', Encoder(in_channels, self.pretrained_path)),]))


    def forward(self, supp_imgs, qry_imgs):
        """
        Args:
            supp_imgs:
                support images (way x shot x 3 x H x W)
            qry_imgs:
                query images (q x 3 x H x W)
        """
        n_ways = supp_imgs.shape[0]
        n_shots = supp_imgs.shape[1]
        n_queries = qry_imgs.shape[0]
        img_size = supp_imgs.shape[-2:]

        # Concatenate support and query images into one batch

        # (Ways*Shot+Queries) x 3 x H x W
        imgs_concat = torch.cat([supp_imgs.view(n_ways*n_shots,3,*img_size)]
                                +[qry_imgs], dim=0)

        # Extract features

        # (Way*Shots + Q) x V(512) x H' x W'
        img_fts = self.encoder(imgs_concat)
        fts_size = img_fts.shape[-2:]

        # Separate support and query features

        #W x Sh x V(512) x H' x W'
        supp_fts = img_fts[:n_ways * n_shots].view(
             n_ways, n_shots, -1, *fts_size)

        # Q x V(512) x H' x W'
        qry_fts = img_fts[n_ways * n_shots:]

        return supp_fts, qry_fts
```

A.2.10 metric.py:

```
"""
@author: A anonymous

Metric Script:
Contains module for computing evalutation metrics

Adapted code from "PANet: Few-Shot Image Semantic Segmentation with
Prototype Alignment" by Kaixin Wang et al. at https://github.com/kaixin96/PANet
```

```
"""

import numpy as np

class Metric(object):
    """
    Compute evaluation result

    Args:
        max_label:
            max label index in the data (0 denoting background)
        n_runs:
            number of test runs
    """
    def __init__(self, max_label=20, n_runs=None):
        self.labels = list(range(max_label + 1))  # all class labels
        self.n_runs = 1 if n_runs is None else n_runs

        # list of list of array, each array save the TP/FP/FN statistic of a testing sample
        self.tp_lst = [[] for _ in range(self.n_runs)]
        self.fp_lst = [[] for _ in range(self.n_runs)]
        self.fn_lst = [[] for _ in range(self.n_runs)]

    def record(self, pred, target, labels=None, n_run=None):
        """
        Record the evaluation result for each sample and each class label, including:
            True Positive, False Positive, False Negative

        Args:
            pred:
                predicted mask array, expected shape is H x W
            target:
                target mask array, expected shape is H x W
            labels:
                only count specific label, used when knowing all possible labels in advance
        """
        assert pred.shape == target.shape

        if self.n_runs == 1:
            n_run = 0

        # array to save the TP/FP/FN statistic for each class (plus BG)
        tp_arr = np.full(len(self.labels), np.nan)
        fp_arr = np.full(len(self.labels), np.nan)
        fn_arr = np.full(len(self.labels), np.nan)

        if labels is None:
            labels = self.labels
        else:
            labels = [0,] + labels

        for j, label in enumerate(labels):
            # Get the location of the pixels that are predicted as class j
```

```
            idx = np.where(np.logical_and(pred == j, target != 255))
            pred_idx_j = set(zip(idx[0].tolist(), idx[1].tolist()))
            # Get the location of the pixels that are class j in ground truth
            idx = np.where(target == j)
            target_idx_j = set(zip(idx[0].tolist(), idx[1].tolist()))

            if target_idx_j:  # if ground-truth contains this class
                tp_arr[label] = len(set.intersection(pred_idx_j, target_idx_j))
                fp_arr[label] = len(pred_idx_j - target_idx_j)
                fn_arr[label] = len(target_idx_j - pred_idx_j)

        self.tp_lst[n_run].append(tp_arr)
        self.fp_lst[n_run].append(fp_arr)
        self.fn_lst[n_run].append(fn_arr)

    def get_mIoU(self, labels=None, n_run=None):
        """
        Compute mean IoU

        Args:
            labels:
                specify a subset of labels to compute mean IoU, default is using all classes
        """
        if labels is None:
            labels = self.labels
        # Sum TP, FP, FN statistic of all samples
        if n_run is None:
            tp_sum = [np.nansum(np.vstack(self.tp_lst[run]), axis=0).take(labels)
                      for run in range(self.n_runs)]
            fp_sum = [np.nansum(np.vstack(self.fp_lst[run]), axis=0).take(labels)
                      for run in range(self.n_runs)]
            fn_sum = [np.nansum(np.vstack(self.fn_lst[run]), axis=0).take(labels)
                      for run in range(self.n_runs)]

            # Compute mean IoU classwisely
            # Average across n_runs, then average over classes
            mIoU_class = np.vstack([tp_sum[run] / (tp_sum[run] + fp_sum[run] + fn_sum[run])
                                    for run in range(self.n_runs)])
            mIoU = mIoU_class.mean(axis=1)

            return (mIoU_class.mean(axis=0), mIoU_class.std(axis=0),
                    mIoU.mean(axis=0), mIoU.std(axis=0))
        else:
            tp_sum = np.nansum(np.vstack(self.tp_lst[n_run]), axis=0).take(labels)
            fp_sum = np.nansum(np.vstack(self.fp_lst[n_run]), axis=0).take(labels)
            fn_sum = np.nansum(np.vstack(self.fn_lst[n_run]), axis=0).take(labels)

            # Compute mean IoU classwisely and average over classes
```

```
            mIoU_class = tp_sum / (tp_sum + fp_sum + fn_sum)
            mIoU = mIoU_class.mean()

            return mIoU_class, mIoU

    def get_mIoU_binary(self, n_run=None):
        """
        Compute mean IoU for binary scenario
        (sum all foreground classes as one class)
        """
        # Sum TP, FP, FN statistic of all samples
        if n_run is None:
            tp_sum = [np.nansum(np.vstack(self.tp_lst[run]), axis=0)
                      for run in range(self.n_runs)]
            fp_sum = [np.nansum(np.vstack(self.fp_lst[run]), axis=0)
                      for run in range(self.n_runs)]
            fn_sum = [np.nansum(np.vstack(self.fn_lst[run]), axis=0)
                      for run in range(self.n_runs)]

            # Sum over all foreground classes
            tp_sum = [np.c_[tp_sum[run][0], np.nansum(tp_sum[run][1:])]
                      for run in range(self.n_runs)]
            fp_sum = [np.c_[fp_sum[run][0], np.nansum(fp_sum[run][1:])]
                      for run in range(self.n_runs)]
            fn_sum = [np.c_[fn_sum[run][0], np.nansum(fn_sum[run][1:])]
                      for run in range(self.n_runs)]

            # Compute mean IoU classwisely and average across classes
            mIoU_class = np.vstack([tp_sum[run] / (tp_sum[run] + fp_sum[run] +
    fn_sum[run])
                                    for run in range(self.n_runs)])
            mIoU = mIoU_class.mean(axis=1)

            return (mIoU_class.mean(axis=0), mIoU_class.std(axis=0),
                    mIoU.mean(axis=0), mIoU.std(axis=0))
        else:
            tp_sum = np.nansum(np.vstack(self.tp_lst[n_run]), axis=0)
            fp_sum = np.nansum(np.vstack(self.fp_lst[n_run]), axis=0)
            fn_sum = np.nansum(np.vstack(self.fn_lst[n_run]), axis=0)

            # Sum over all foreground classes
            tp_sum = np.c_[tp_sum[0], np.nansum(tp_sum[1:])]
            fp_sum = np.c_[fp_sum[0], np.nansum(fp_sum[1:])]
            fn_sum = np.c_[fn_sum[0], np.nansum(fn_sum[1:])]

            mIoU_class = tp_sum / (tp_sum + fp_sum + fn_sum)
            mIoU = mIoU_class.mean()

            return mIoU_class, mIoU
```

A.2.11 utils.py:

```
"""
@author: A Kumarathas

```

```
Util Script:
Contains functions for setting up and saving experiments, logging data,
running validation

Adapted code from "PANet: Few-Shot Image Semantic Segmentation with
Prototype Alignment" by Kaixin Wang et al. at https://github.com/kaixin96/PANet
"""

import random
import logging
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

from models.network import FewShotNet
from config import *

CLASS_LABELS = {
    'VOC': {
        'parts': set([1, 2, 3, 4, 6, 7, 8, 10, 12, 13, 14, 15, 16, 17, 19, 20]),
        'parts0': set([6, 7, 8, 10, 12, 13, 14, 15, 16, 17, 19, 20]),
        'parts1': set([1, 2, 3, 4, 12, 13, 14, 15, 16, 17, 19, 20]),
        'parts2': set([1, 2, 3, 4, 6, 7, 8, 10, 16, 17, 19, 20]),
        'parts3': set([1, 2, 3, 4, 6, 7, 8, 10, 12, 13, 14, 15]),
    },
}

def set_seed(seed):
    """
    Set the random seed
    """
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

def config_experiment(name, resume=True):

    exp = {}
    os.makedirs(chkpts_dir+name, exist_ok=True)
    logger = config_logger(name)

    if resume:

        try:
            exp = torch.load(chkpts_dir+name+'/chkpt.pth',map_location=lambda
    storage, loc: storage)
            logger.info("loading checkpoint, experiment: " + name)
            return exp, logger
        except Exception as e:
            logger.warning('checkpoint does not exist. creating new experiment')

    model = FewShotNet(pretrained_path=path['init_path'], cfg=model_align,
    model_arch = architecture)
    model = nn.DataParallel(model.cuda(), device_ids=[gpu_id,])
```

```
58
59      exp['model_state_dict'] = model.state_dict()
60      exp['step'] = 0
61      exp['loss'] = {'loss': [], 'var_loss':[], 'dist_loss':[],'var_part_loss':[],
         'dist_part_loss':[],'crossentropy_loss':[]}
62      exp['log_loss'] = {'loss': 0, 'var_loss':0, 'dist_loss': 0,'var_part_loss'
        :0, 'dist_part_loss':0,'crossentropy_loss':0}
63      exp['avg_loss'] = {'loss': [], 'var_loss':[], 'dist_loss':[],'var_part_loss'
        :[], 'dist_part_loss':[],'crossentropy_loss':[]}
64      exp['val_loss'] = {'loss': [], 'var_loss':[], 'dist_loss':[],'var_part_loss'
        :[], 'dist_part_loss':[],'crossentropy_loss':[]}
65
66      return exp, logger
67
68
69  def save_experiment(exp, name):
70      torch.save(exp,chkpts_dir+name+'/chkpt.pth')
71
72
73  def config_logger(current_exp):
74      logger = logging.getLogger(current_exp)
75      # logger.propagate = False
76      logger.setLevel(logging.DEBUG)
77      handler = logging.StreamHandler()
78      handler.setLevel(logging.DEBUG)
79      handler2 = logging.FileHandler(chkpts_dir+current_exp+'/log')
80      handler2.setLevel(logging.INFO)
81      formatter = logging.Formatter(
82              '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
83      handler.setFormatter(formatter)
84      handler2.setFormatter(formatter)
85      logger.addHandler(handler)
86      logger.addHandler(handler2)
87
88      return logger
89
90  def save_configfile(name, model_loss_fn):
91      file = open(chkpts_dir+name+"/config.txt", "w")
92      file.write("##### Training Configurations #####" + "\n"+
93                 "Experiment = " + repr(current_experiment) + "\n"+
94                 "Data Name = " + repr(data_name) + "\n"+
95                 "Train Labels = " + repr(label_sets) + "\n"+
96                 "Input Size = " + repr(input_size) + "\n"+
97                 "Train Steps = " + repr(n_steps) + "\n"+
98                 "Train task = " + repr(task) + "\n"+
99                 "Optimisation = " + repr(optim) + "\n"+
100                "LR Milestones = " + repr(lr_milestones) + "\n"+
101                "Seed = " + repr(seed) + "\n"+
102                "Loss = " + repr(type(model_loss_fn).__name__) + "\n"+
103                "Loss type = " + repr(loss_type) + "\n"+
104                "Model architecture = " + repr(architecture) + "\n"+
105                "##### Testing Configurations #####" + "\n"+
106                "Runs = " + repr(n_runs) + "\n"+
107                "Test steps = " + repr(n_teststeps) + "\n")
108     file.close
109
```

```
def save_testfile_run(name, run, labels, classIoU, meanIoU,
                      classIoU_binary, meanIoU_binary):

    with open(chkpts_dir+name+'/testresults.txt', 'a') as file:
        file.write(f'### Run {run + 1} ###' + "\n"+
                   f'classes in run: {labels}' + "\n"+
                   f'classIoU: {classIoU}' + "\n"+
                   f'mean classIoU: {meanIoU}' + "\n"+
                   f'binary classIoU: {classIoU_binary}' + "\n"+
                   f'mean binary classIoU: {meanIoU_binary}' + "\n"+
                   '_______________________' + "\n")

def save_testfile_final(name, labels, classIoU ,classIoU_std,
                        meanIoU, meanIoU_std,
                        classIoU_binary, classIoU_std_binary,
                        meanIoU_binary, meanIoU_std_binary):

    with open(chkpts_dir+name+'/testresults.txt', 'a') as file:
        file.write('----- Final Result -----' + "\n" +
                   f'classes in run: {labels}' + "\n"+
                   f'classIoU mean: {classIoU}' + "\n"+
                   f'classIoU std: {classIoU_std}' + "\n"+
                   f'meanIoU mean: {meanIoU}' + "\n"+
                   f'meanIoU std: {meanIoU_std}' + "\n"+
                   f'classIoU_binary mean: {classIoU_binary}' + "\n" +
                   f'classIoU_binary std: {classIoU_std_binary}' + "\n" +
                   f'meanIoU_binary mean: {meanIoU_binary}' + "\n" +
                   f'meanIoU_binary std: {meanIoU_std_binary}' + "\n" +
                   '_______________________' + "\n")

def evaluate_val(valloader,model,model_loss_fn,loss_type):

    log_loss = {'loss': 0, 'var_loss':0, 'dist_loss': 0,'var_part_loss':0, '
    dist_part_loss':0, 'crossentropy_loss':0}

    for i_iter, sample_batched in enumerate(valloader):

        print('Val iteration',i_iter)

        # Prepare input
        # Way x Shots x Imgdim x H x W
        support_images = sample_batched['support_images'][0,...].cuda()

        # Q x Imgdim(3) x H x W
        query_images = sample_batched['query_images'][0,...].cuda()

        # Way x Shots x H x W x (C+1)
        support_class_masks = sample_batched['support_class_mask'][0,...].cuda()

        # Q x H x W x (C+1)
        query_class_masks = sample_batched['query_class_mask'][0,...].cuda()

        # Way x Shots x H x W x (C+1)
        support_part_masks = sample_batched['support_part_mask'][0,...].cuda()

        # Q x H x W x (C+1)
```

```
        query_part_masks = sample_batched['query_part_mask'][0,...].cuda()

        # P (Not utilised)
        part_idx = [torch.tensor([0])] + sample_batched['part_classidx_full']
        part_idx = torch.cat([i for i in part_idx],dim=0).cuda()

        support_fts, query_fts = model(support_images, query_images)

        n_ways = support_images.shape[0]
        n_shots = support_images.shape[1]
        n_queries = query_images.shape[0]
        img_size = support_images.shape[-2:]

        pred, _, var_loss, dist_loss,var_part_loss,dist_part_loss =
    model_loss_fn(support_fts,

    query_fts,

    support_class_masks,

    query_class_masks,

    support_part_masks,

    query_part_masks,

    n_ways,

    n_shots,

    n_queries,

    part_idx,

    img_size,

    loss_type)

        if loss_type == 'parts':
            loss = var_loss + dist_loss + var_part_loss + dist_part_loss
        else:
            loss = var_loss + dist_loss


        ce_loss = F.cross_entropy(1/pred,torch.argmax(query_class_masks,dim=3))

        # Log loss
        var_loss = var_loss.clone().detach().data.cpu().numpy()
        dist_loss = dist_loss.clone().detach().data.cpu().numpy() if dist_loss
    != 0 else 0
        var_part_loss = var_part_loss.clone().detach().data.cpu().numpy()
        dist_part_loss = dist_part_loss.clone().detach().data.cpu().numpy() if
    dist_part_loss != 0 else 0
        loss = loss.clone().detach().data.cpu().numpy()
        ce_loss = ce_loss.clone().detach().data.cpu().numpy()

```

```
        log_loss['var_loss'] += var_loss
        log_loss['dist_loss'] += dist_loss
        log_loss['var_part_loss'] += var_part_loss
        log_loss['dist_part_loss'] += dist_part_loss
        log_loss['loss'] += loss
        log_loss['crossentropy_loss'] += ce_loss

    return log_loss

def log_losses(loss_hist,log_loss,avgloss_hist,
                var_loss,dist_loss,var_part_loss,dist_part_loss,loss,ce_loss,
    step):

    loss_hist['var_loss'].append(var_loss)
    loss_hist['dist_loss'].append(dist_loss)
    loss_hist['var_part_loss'].append(var_part_loss)
    loss_hist['dist_part_loss'].append(dist_part_loss)
    loss_hist['loss'].append(loss)
    loss_hist['crossentropy_loss'].append(ce_loss)

    # Cumulative Loss
    log_loss['var_loss'] += var_loss
    log_loss['dist_loss'] += dist_loss
    log_loss['var_part_loss'] += var_part_loss
    log_loss['dist_part_loss'] += dist_part_loss
    log_loss['loss'] += loss
    log_loss['crossentropy_loss'] += ce_loss

    # Avg Loss over time
    avgloss_hist['loss'].append(log_loss['loss'] / (step))
    avgloss_hist['var_loss'].append(log_loss['var_loss'] / (step))
    avgloss_hist['dist_loss'].append(log_loss['dist_loss'] / (step))
    avgloss_hist['var_part_loss'].append(log_loss['var_part_loss'] / (step))
    avgloss_hist['dist_part_loss'].append(log_loss['dist_part_loss'] / (step))
    avgloss_hist['crossentropy_loss'].append(log_loss['crossentropy_loss'] / (
    step))

    return loss_hist, log_loss, avgloss_hist

def log_vallosses(val_loss, val_hist, n_valsteps):

    val_hist['loss'].append(val_loss['loss'] / n_valsteps)
    val_hist['var_loss'].append(val_loss['var_loss'] / n_valsteps)
    val_hist['dist_loss'].append(val_loss['dist_loss'] / n_valsteps)
    val_hist['var_part_loss'].append(val_loss['var_part_loss'] / n_valsteps)
    val_hist['dist_part_loss'].append(val_loss['dist_part_loss'] / n_valsteps)
    val_hist['crossentropy_loss'].append(val_loss['crossentropy_loss'] /
    n_valsteps)

    return val_hist
```