

**Yindong Zhang**

**Zero-shot Semantic Segmentation using Relation Network**

Master's thesis of information technology

May 28, 2020

University of Jyväskylä  
Faculty of Information Technology

**Author:** Yindong Zhang

**Contact information:** zhangy@student.jyu.fi

**Supervisors:** Oleksiy Khriyenko

**Title:** Zero-shot Semantic Segmentation using Relation Network

**Työn nimi:** Zero-shot Semantic Segmentation using Relation Network

**Project:** Master's thesis

**Study line:** Cognitive Computing and Collective Intelligence

**Page count:** 46+17

**Abstract:** Zero-shot learning (ZSL) is widely studied in recent years to solve the problem of lacking annotations. Currently, most studies on ZSL are for image classification and object detection. But zero-shot semantic segmentation, pixel level classification, is still at its early stage. Therefore, this thesis proposes to extend a zero-shot image classification model, Relation Network (RN), to semantic segmentation tasks. This thesis modifies the structure of RN based on other state-of-the-arts semantic segmentation models (i.e. U-Net and DeepLab) and utilizes word embeddings from Caltech-UCSD Birds 200-2011 attributes and natural language processing models (i.e. word2vec and fastText). Because meta-learning is limited to binary tasks, this thesis proposes to join multiple binary semantic segmentation pipelines for multi-class semantic segmentation. It is proved by experiments that RN could improve accuracy of U-Net with the help of semantic side information on binary semantic segmentation and it could also be applied on multi-class semantic segmentation with simpler structure than the baseline model, SPNet, but higher accuracy under ZSL setting. However, the capability of RN under generalized zero-shot learning (GZSL) setting still needs improvement. This thesis also studies on how different word embeddings, network structures and data affect RN and what could be done to improve its results.

**Keywords:** Zero-shot Learning, Semantic Segmentation, Relation Network, Meta-learning

**Suomenkielinen tiivistelmä:** Zero-shot learning (ZSL) is widely studied in recent years to solve the problem of lacking annotations. Currently, most studies on ZSL are for image classification and object detection. But zero-shot semantic segmentation, pixel level classification, is still at its early stage. Therefore, this thesis proposes to extend a zero-shot image classification model, Relation Network (RN), to semantic segmentation tasks. This thesis modifies the structure of RN based on other state-of-the-arts semantic segmentation models (i.e. U-Net and DeepLab) and utilizes word embeddings from Caltech-UCSD Birds 200-2011 attributes and natural language processing models (i.e. word2vec and fastText). Because meta-learning is limited to binary tasks, this thesis proposes to join multiple binary semantic segmentation pipelines for multi-class semantic segmentation. It is proved by experiments that RN could improve accuracy of U-Net with the help of semantic side information on binary semantic segmentation and it could also be applied on multi-class semantic segmentation with simpler structure than the baseline model, SPNet, but higher accuracy under ZSL setting. However, the capability of RN under generalized zero-shot learning (GZSL) setting still needs improvement. This thesis also studies on how different word embeddings, network structures and data affect RN and what could be done to improve its results.

**Avainsanat:** Zero-shot Learning, Semantic Segmentation, Relation Network, Meta-learning

## Glossary

semantic segmentation:	Pixel level classification
loss function:	A function used to calculate the distance between prediction and ground truth while training
batch size:	The number of selected images in each episode
learning rate:	Step size in each episode
IoU:	An evaluation metric for semantic segmentation called intersection over union
seen class:	A class that is included in the training data
unseen class:	A class that is not included in the training data
ZSL:	Zero-shot learning where the model recognizes unseen classes during testing
GZSL:	Generalized zero-shot learning where the model recognizes both seen and unseen classes during testing
ZLSS:	Zero-label semantic segmentation, same as zero-shot semantic segmentation
GZLSS:	Generalized semantic segmentation, same as generalized zero-shot semantic segmentation
query image:	An image needs to be recognized
RAM:	Random access memory
feature maps:	A set of two-dimensional matrices
channels:	Depth of feature maps
embeddings:	A set of models or techniques that extract feature maps from words or images
semantic features:	A set of feature maps representing words
visual features:	A set of feature maps representing images
attribute:	A numeric value representing a visual feature of a class or an object
NLP:	Natural language processing

## List of Figures

Figure 1. Example of object detection from ILSVRC2013 (Deng et al. 2009). .....	2
Figure 2. Example of segmenting cells from medical images (Ronneberger, Fischer, and Brox 2015). .....	3
Figure 3. Example of a 1-shot 5-way problem. ....	3
Figure 4. Attributes of class “black footed albatross” and an image from CUB. ....	4
Figure 5. An example of semantic segmentation. (Everingham et al. 2012).....	7
Figure 6. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation. (Long, Shelhamer, and Darrell 2015).....	8
Figure 7. U-net architecture (example for 32x32 pixels in the lowest resolution). (Ronneberger, Fischer, and Brox 2015).....	9
Figure 8. Illustration of atrous convolution in 1-D. (Chen et al. 2017) .....	10
Figure 9. Illustration of atrous convolution pyramid pooling. (Chen et al. 2017).....	10
Figure 10. An illustration of the projection domain shift problem. (Fu et al. 2015) .....	11
Figure 11. An illustration of ALE. (Akata et al. 2013).....	12
Figure 12. An illustration of LatEm. (Xian et al. 2016) .....	13
Figure 13. An example of SSE. (Zhang & Saligrama 2015) .....	13
Figure 14. Illustration of the skip-gram model architecture. (Mikolov et al. 2013a) .....	15
Figure 15. RN pipeline for a 5-way 1-shot image classification problem with one query example. (Sung et al. 2018).....	16
Figure 16. RN architecture for zero-shot learning. (Sung et al. 2018) .....	17

Figure 17. Baseline network architecture of FSS-1000 using VGG-16 as Backbone. (Wei et al. 2019) .....	18
Figure 18. An example of SPNet pipeline including visual semantic embedding and semantic projection under ZLSS setting. (Xian et al. 2019) .....	19
Figure 19. ZS3Net pipeline under GZLSS setting. (Bucher et al. 2019).....	19
Figure 20. An example pipeline of one episode in multi-class tasks.....	21
Figure 21. Relation Network architecture for binary semantic segmentation tasks with pre-trained VGG16 and U-Net decoder. ....	24
Figure 22. Relation Network architecture for binary semantic segmentation tasks with DeepLab-v3 removing its last 4 classifier layers. ....	25
Figure 23. Comparison of FC kernel (left) and CNN kernel (right). ....	26
Figure 24. C-way 0-shot semantic segmentation pipeline. ....	27
Figure 25. Relationship between IoU and average object size of unseen classes in PASCAL under ZLSS setting.....	34
Figure 26. Relationship between IoU and average object size of unseen classes in COCO under ZLSS setting.....	35
Figure 27. Qualitative results on CUB unseen classes by U-Net based models with no-vector, attributes, word2vec, fastText and the concatenation of word2vec and fastText. ....	38
Figure 28. Qualitative results on PASCAL unseen classes under ZLSS setting by RN.....	39
Figure 29. Qualitative results on COCO unseen classes under ZLSS setting by U-Net based model with word2vec.....	40

## List of Tables

Table 1. Characteristics of tasks .....	5
Table 2. Effect of word embeddings: mIoU of binary semantic segmentation on CUB with U-Net based model. ....	30
Table 3. Effect of word embeddings module structure: mIoU of binary semantic segmentation on CUB and its attributes with U-Net based models. ....	31
Table 4. Effect of network structure: mIoU of binary semantic segmentation on CUB and CUB attributes with U-Net based model. ....	31
Table 5. Effect of word embeddings: mIoU of multi-class semantic segmentation on PASCAL with U-Net and VGG16 based models. ....	32
Table 6. Effect of word embeddings: mIoU of multi-class semantic segmentation on PASCAL with DeepLab and ResNet101 based models.....	33
Table 7. PASCAL and COCO trained U-Net based models with word2vec and VGG16 cross evaluated on each other’s testing data. ....	34
Table 8. Comparison of mIoU between SPNet on PASCAL. (SPNet data is from Xian et al. 2019) .....	37
Table 9. Comparison of mIoU between SPNet on COCO. (SPNet data is from Xian et al. 2019) .....	37

# Contents

1.	INTRODUCTION .....	1
2.	COMPUTER VISION AND ZERO-SHOT LEARNING.....	7
2.1.	SEMANTIC SEGMENTATION .....	7
2.2.	ZERO-SHOT LEARNING .....	10
2.3.	WORD EMBEDDINGS.....	14
2.4.	RELATION NETWORK .....	15
2.5.	CURRENT ZERO-SHOT SEMANTIC SEGMENTATION.....	18
3.	METHODOLOGY .....	20
3.1.	PROBLEM DEFINITION .....	20
3.2.	BASELINE: U-NET AND SPNET .....	21
3.3.	EVALUATION .....	22
4.	RN FOR ZERO-SHOT SEMANTIC SEGMENTATION .....	23
4.1.	NETWORK ARCHITECTURE .....	23
4.2.	LOSS FUNCTION .....	27
5.	EXPERIMENTS .....	28
5.1.	DATASETS AND SPLITS .....	28
5.2.	IMPLEMENTATION DETAILS .....	29
5.3.	BINARY SEMANTIC SEGMENTATION .....	29
5.3.1.	Effect of word embeddings .....	30
5.3.2.	Effect of network structure .....	30
5.4.	MULTI-CLASS SEMANTIC SEGMENTATION .....	31
5.4.1.	Effect of word embeddings .....	32
5.4.2.	Effect of network structure .....	33
5.4.3.	Effect of data.....	33
6.	DISCUSSION.....	36



6.1.	COMPARE WITH BASELINE MODELS .....	36
6.2.	QUALITATIVE RESULTS .....	37
7.	CONCLUSION.....	41
	REFERENCES .....	43
	APPENDICES .....	47
A.	PALETTE OF PASCAL VOC 2012 .....	47
B.	IMPLEMENTATION CODES OF NETWORK STRUCTURE .....	47
C.	IMPLEMENTATION CODES OF IOU COMPUTING .....	54
D.	IMPLEMENTATION CODES OF PREPROCESSING PASCAL.....	56
E.	IMPLEMENTATION CODES OF TRAINING PASCAL.....	59

# 1. Introduction

Increasing public datasets is an important boost for computer vision field. At first, computer vision datasets were rather small. For example, PASCAL VOC <sup>1</sup>(Everingham et al. 2012), as one of the most influential benchmarks established in 2005, only has 20 classes. Researchers focused more on improving algorithms instead of enriching their training data. But ImageNet (Deng et al. 2009), including 14,197,122 images covering 21841 categories, became a game changer for this filed by proving the importance of diverse and balanced data. Since the success of ImageNet, more and more huge datasets are created for all kinds of computer vision tasks, such as COCO-Stuff <sup>2</sup>(Caesar, Uijlings, and Ferrari 2018). At the beginning, these datasets only had image classification annotations, but now object detection annotations (i.e. annotate bound boxes of objects) and semantic segmentation annotations (i.e. annotate label of each pixel) are also included in many large-scale datasets.

Huge datasets also unleash potential of neural networks. Since the first ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2010, participating teams managed to decrease their error rate from more than 25% to lower than 5% on image classification (Russakovsky et al. 2015). And since AlexNet (Krizhevsky, Sutskever, and Hinton 2012) achieved 15% of error rate by utilizing neural network, this technique has been widely used in recent models.

Therefore, with the development of datasets, neural networks and graphics processing units (GPUs), many deep learning models emerged and are deployed to practical applications. For example, Visual Geometry Group (VGG) was published in ILSVRC 2014 with wider and deeper network structure than previous models, reaching 7.32% error rate on the image classification task (Zhang et al. 2015). Later, Residual Network (ResNet), as the champion of ILSVRC 2015, solves the “accuracy decreasing” problem in very deep networks. Moreover, image classification tasks sometimes come with localization, which requires models to locate the classified object (Deng et al. 2009). Image classification models could be applied on garbage classification, plants

<sup>1</sup> <http://host.robots.ox.ac.uk/pascal/VOC/voc2012/>

<sup>2</sup> <https://github.com/nightrome/cocostuff#downloads>

classification etc. But to extract more information from the real world more tasks are defined, including object detection and semantic segmentation.

Object detection can be considered as an extended task of the classification and localization task, because it requires models classifying multiple objects in one image and meanwhile recognize their locations and sizes. Figure 1 displays an example of object detection from ILSVRC2013 when it started to include object detection tasks. Object detection is a more realistic task, since there is usually more than one object in in-vehicle cameras, photos etc.

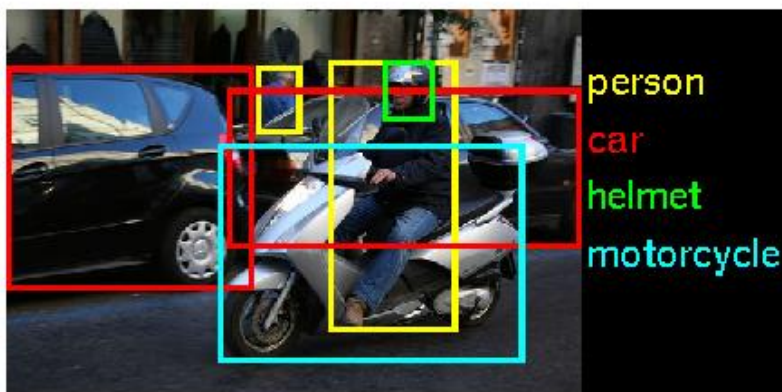


Figure 1. Example of object detection from ILSVRC2013 (Deng et al. 2009).

Furthermore, researchers propose to challenge pixel level classification - semantic segmentation. Semantic segmentation could be considered as an extension of image classification. Because the feature maps extracted by classification models could also be utilized for segmentation, researchers often develop their segmentation models based on successful classification models. One of the most common use cases of it is medical image segmentation. For example, as Figure 2 shows, on the left cells are segmented from background, and on the right different cells are segmented and classified. Besides, semantic segmentation could also be utilized in satellite image processing, facial segmentation etc. Moreover, as shown in Figure 2, it can be categorized into binary task (i.e. classify pixels into one class or background) and multi-class task (i.e. classify pixels into multiple classes and background). In current models, U-Net (Ronneberger, Fischer, and Brox 2015) has excellent performance in binary semantic segmentation tasks (e.g. medical images segmentation) and DeepLab-v3 (Chen et al. 2018) framework is more popular in multi-class segmentation tasks.

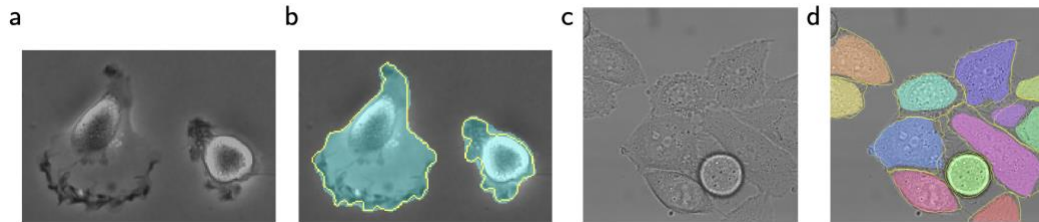


Figure 2. Example of segmenting cells from medical images (Ronneberger, Fischer, and Brox 2015).

Although neural networks are intensely studied, labelling is still a big obstacle in practical applications with supervised learning because it is impossible to label everything. Therefore, there are increasing studies on few-shot learning (FSL) and zero-shot learning (ZSL) which means predicting on classes that have been seen only a few times or never during training. FSL tasks are denoted as  $k$ -shot  $c$ -way problems, where  $k$  means the number of sample images from each class and  $c$  means the number of classes (see an example of a 1-shot 5-way problem in Figure 3) (Sung et al. 2018). And ZSL tasks are denoted as 0-shot  $c$ -way problems, because ZSL does not have sample images. FSL could also be applied on medical images as they often lack labelling. Whereas ZSL is more common in practice, such as recognizing objects on the street and recognizing handwriting text in an unseen language.

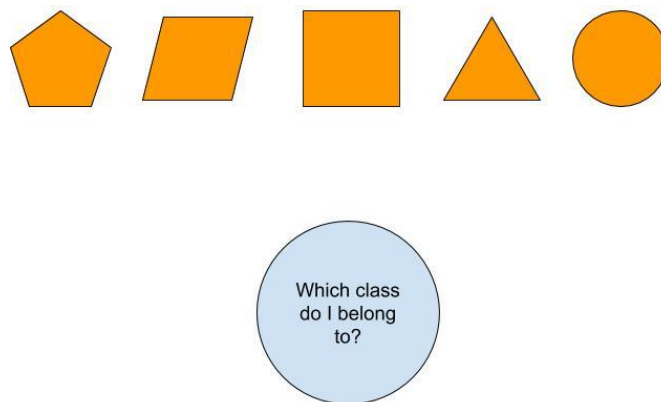



Figure 3. Example of a 1-shot 5-way problem.

In ZSL there are seen classes and unseen classes. The goal of ZSL is to transfer the knowledge learnt from seen classes to unseen classes. Although there is no labelled data for unseen classes,

we can import side information to help it, such as semantic information. Semantic side information in ZSL usually refers to attributes and natural language processing (NLP) models. Attributes are visual characteristics of a class or an object. Figure 4 shows an example image from Caltech-UCSD Birds 200-2011 (CUB) (Wah et al. 2011) with some of its class attributes and image attributes. CUB class attributes have continuous value which is “the percentage of the time (between 0 and 100) when a human thinks that the attribute is present for a given class (Wah et al. 2011)”. And CUB image attributes are Boolean values where 1 means “is present” and 0 means “is not present”. In Figure 4, the class “Black Footed Albatross” mostly likely has “hooked seabird” shape bills (59.85% is the highest among all the bill shapes), and the image on the right fulfills this characteristic of “Black Footed Albatross”.

Attribute name	Class attribute value	Image attribute value
has_bill_shape:curved_(up_or_down)	0.0	0
has_bill_shape:dagger	2.9197080292	0
has_bill_shape:hooked	1.4598540146	0
has_bill_shape:needle	0.0	0
has_bill_shape:hooked_seabird	59.8540145985	1
has_bill_shape:spatulate	26.2773722628	0
has_bill_shape:spatulate	3.6496350365	0



Black Footed Albatross

Figure 4. Attributes of class “black footed albatross” and an image from CUB.

Another useful technique for FSL and ZSL is meta-learning. Meta-learning aims to help models learn to adapt to new classes by feeding models random samples and random label space in every episode. In meta-learning, the concept of epoch is substituted by the same amount of iterations - “episodes”. A successful method of meta-learning is to learn the best initial weights from training data, and then fine tune models based on testing data. But it is also possible to train a model which does not require fine tuning (Sung et al. 2018).

Additionally, researchers also propose to do ZSL with recurrent neural networks and generative networks. The former one could update its knowledge while running which means it could

<sup>3</sup> <http://www.vision.caltech.edu/visipedia/CUB-200-2011.html>

continue learning. The latter one could generate synthetic images from semantic information and compare them with the input images. But these two types of networks are usually battery-consuming and computing-consuming which makes them difficult to be deployed on small devices (Sung et al. 2018).

Currently, most ZSL models in computer vision field are for image classification and object detection, including Relation Network (RN) (Sung et al. 2018). RN is originally designed for few-shot image classification by utilizing meta-learning, but it could be easily extended to ZSL (Sung et al. 2018). As for zero-shot semantic segmentation, it is still at its very early stage. Xian et al. (2019) propose new tasks of zero-label semantic segmentation (ZLSS) and general zero-label semantic segmentation (GZLSS) with a baseline model called Semantic Projection Network (SPNet). In ZLSS it is assumed that there are only unseen classes in testing data. For example, a segmentation model is trained on a huge dataset of forest animals, but it will be applied on sea animals which does not have any annotations. Whereas, GZLSS is more practical where both seen and unseen classes could appear during testing. For example, the forest animal trained model will be applied on jungle animals.

	K-shot	C-way	Testing classes
Binary ZLSS	$K = 0$	$C = 2$	$C_u$
Multi-class ZLSS	$K = 0$	$C \geq 2$	$C_u$
Multi-class GZLSS	$K = 0$	$C \geq 2$	$C_u \cup C_s$

Table 1. Characteristics of tasks.

As shown in Table 1, this thesis categorized all zero-shot semantic segmentation tasks into binary ZLSS, multi-class ZLSS and multi-class GZLSS. Same to ordinary binary segmentation tasks, in binary ZLSS, the model only needs to distinguish background and the target class in one image, and its training data and testing data has different target classes. As binary tasks are limited to 2-way, they do not apply to GZLSS. As for multi-class tasks, the model segments multiple classes and background in one image. In multi-class ZLSS, training data and testing data has disjointed label space, whereas multi-class GZLSS have jointed label space.

To study on ZLSS and GZLSS, one feasible idea is extending existing models. For example, Xian et al (2019) construct SPNet by removing the last classifier layer of DeepLab and appending a projection layer to it. A similar idea is to extend a zero-shot image classification model to a zero-shot semantic segmentation model. Therefore, this thesis proposes to extend RN for ZLSS and GZLSS. In the following research, this thesis aims to explore how RN could be extended, whether it has advantages over baseline models and how to improve its capability.

Following Section 2 first introduces recent state-of-the-art semantic segmentation models, ZSL models and word embedding techniques. Then it illustrates RN and current ZLSS models. Section 3 describes research questions in details and research methods. Section 4 covers how to extend RN and how to train extended models. Section 5 documents implementation details and experiments results. Section 6 compares quantitative results of RN extended models with baseline models and analyzes qualitative results of them. In the end, Section 6 summaries work of this thesis and what could be done in the future.

## 2. Computer Vision and Zero-shot Learning

### 2.1. Semantic segmentation

Semantic image segmentation refers to pixel classification, where each pixel is labeled with an object category. For example, in Figure 5, the left image is segmented into person and motorbike, displayed in the right image. It can be applied to autonomous driving, medical areas etc. Recent state-of-the-art semantic segmentation models all benefit from convolutional neural networks (CNNs) (Long, Shelhamer, and Darrell 2015; Ronneberger, Fischer, and Brox 2015; Badrinarayanan, Kendall, and Cipolla 2017; Chen et al. 2017). With CNNs, a basic image classification model consists of multiple CNN layers for extracting dense features and ends with a fully connected (FC) layer for classification. As for pixel level classification, CNN can also be applied to extract features and classify each pixel. But there is an accuracy and computation dilemma: complete predictions require CNN layers to keep the original size to contain each pixel, however faster computation needs smaller layer size.

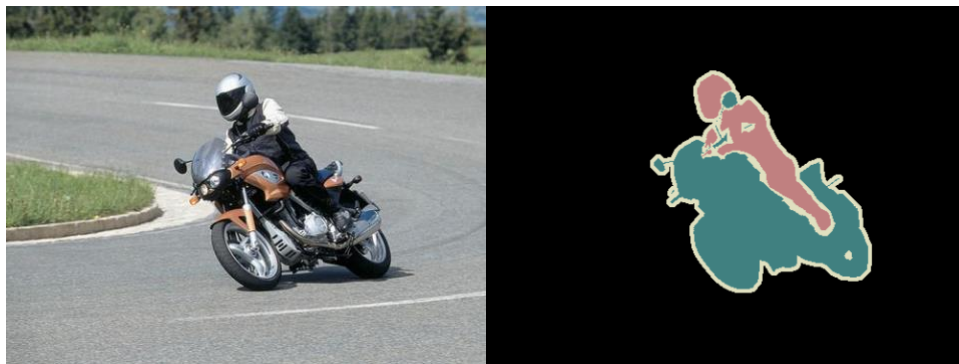


Figure 5. An example of semantic segmentation. (Everingham et al. 2012)

A common resolution is encoder-decoder structure. Fully Convolutional Networks (FCN) is the first one to utilize this structure for semantic segmentation tasks (Long, Shelhamer, and Darrell 2015). As Figure 6 demonstrates, FCN replaces last fully connected layers of pretrained image classification models with convolutional layers in order to generate heat maps and then uses



deconvolution (also called upsampling) to classify each pixel. They also propose skip connections in upsampling layers to improve FCN's accuracy.

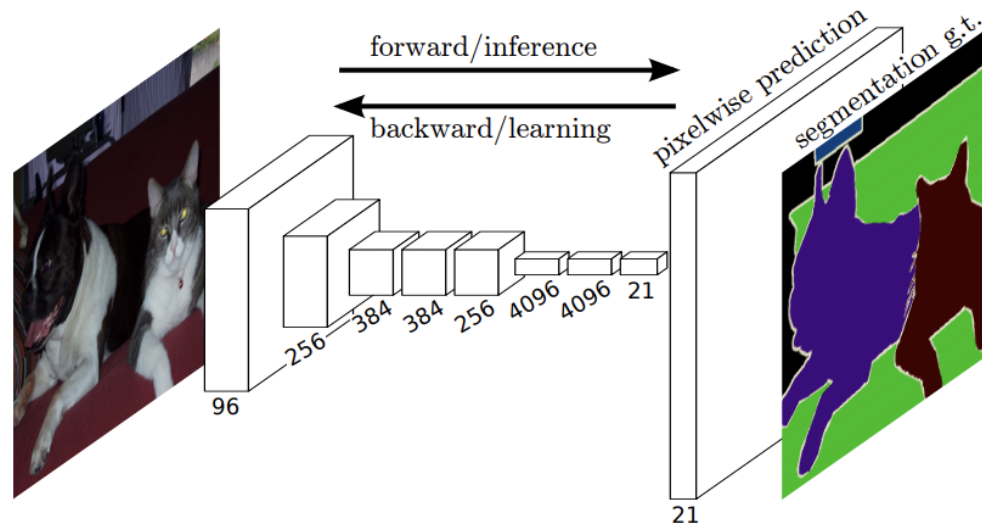


Figure 6. Fully convolutional networks can efficiently learn to make dense predictions for per-pixel tasks like semantic segmentation. (Long, Shelhamer, and Darrell 2015)

Based on FCN, U-Net is designed for medical image segmentation with more flexible network structure (Ronneberger, Fischer, and Brox 2015). As Figure 7 shows, it has a U shape, where the left part is the encoder and the right part is the decoder. Medical images usually are in large size, lack of labelling and only have one target class. Therefore, U-Net is popular in few-shot binary semantic segmentation tasks. Furthermore, because U-Net does not contain any pretrained model, it can be easily adjusted according to image size. SegNet improves FCN in terms of speed and memory consumption by utilizing max-pooling indices during upsampling (Badrinarayanan, Kendall, and Cipolla 2017).

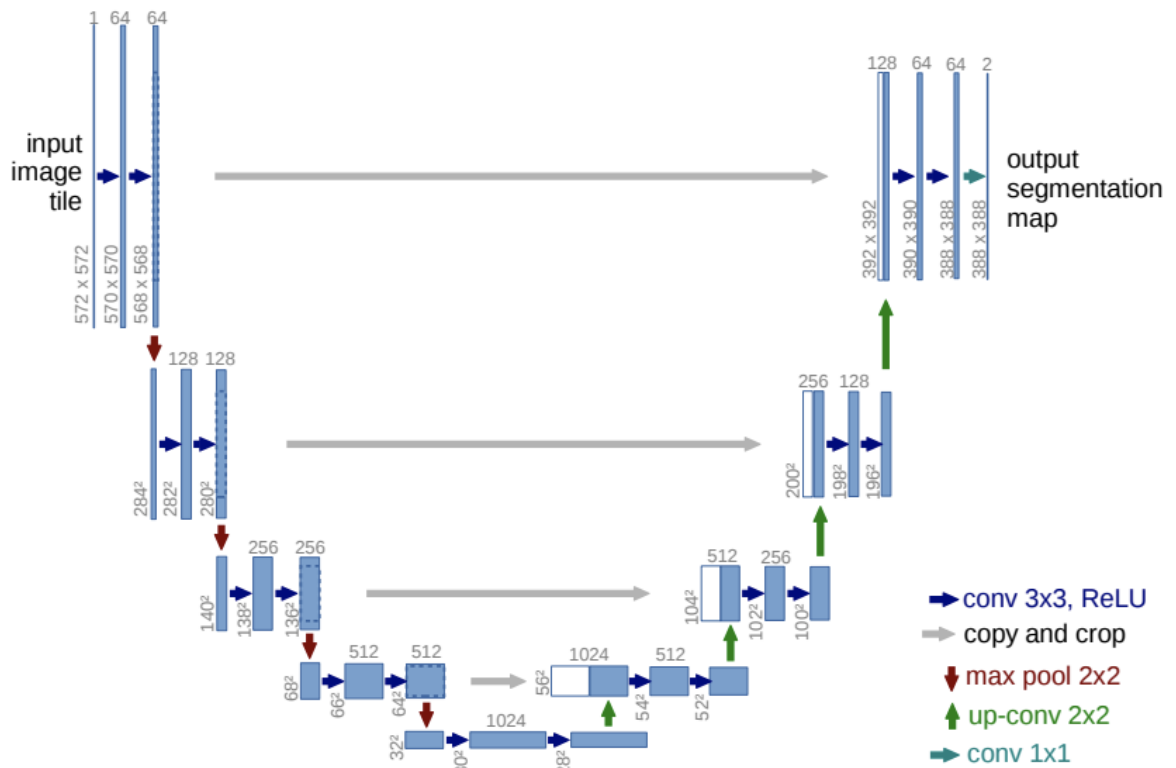


Figure 7. U-net architecture (example for 32x32 pixels in the lowest resolution). (Ronneberger, Fischer, and Brox 2015)

Furthermore, DeepLab is one of the most important semantic segmentation models because it adopts ResNet as its backbone and proposes atrous/dilated convolution and pyramid pooling (Chen et al. 2017). As mentioned in Section 1, ResNet allows models to be deep without reducing their accuracy. As for atrous convolution, Figure 8 illustrates how it works in 1-dimensional data: because neighbor pixels usually are very similar to each other, the kernel expands its cover space with ignoring some input. Figure 8 (a) is a sparse feature extraction with standard convolution, and (b) shows dense features extraction with atrous convolution with rate = 2. By doing this DeepLab is able to expand the view scope of kernels. In another point of view, it could reduce number of parameters and save memory usage. However, ignoring some input causing losing some contextual information. So, they also propose pyramid pooling to solve this problem. As Figure 9 demonstrates, multiple filters are used to provide multi-scale information.

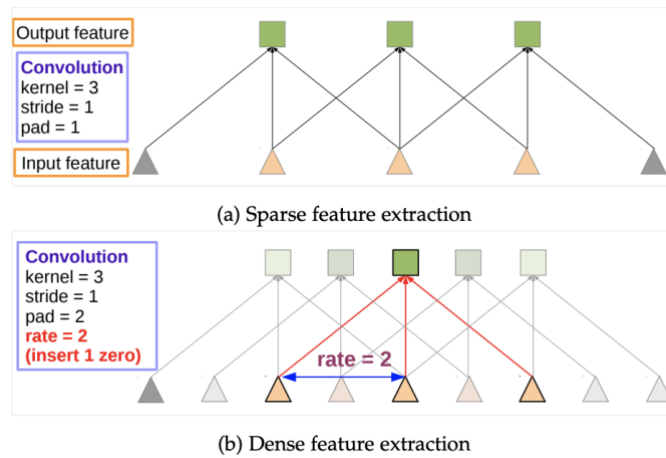


Figure 8. Illustration of atrous convolution in 1-D. (Chen et al. 2017)

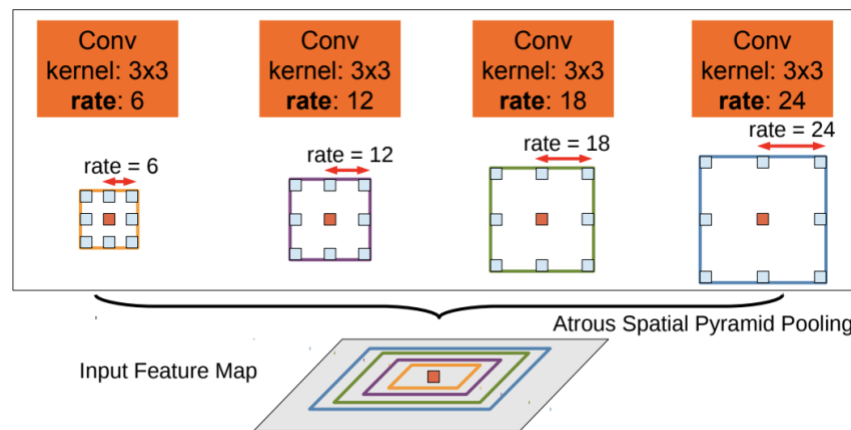


Figure 9. Illustration of atrous convolution pyramid pooling. (Chen et al. 2017)

## 2.2. Zero-shot learning

However, collecting and annotating data is expensive and time-consuming which motivates researchers to study on few-shot learning (FSL) and zero-shot learning (ZSL). If there are 5 classes and 1 sample image in each class, it is called a 5-way 1-shot problem (Sung et al. 2018). In practice, ZSL, where models have to recognize classes that they have never seen during training based on seen classes, is more common than FSL (Sung et al. 2018).

Based on attributes, Lampert, Nickisch, and Harmeling (2013) propose an intermediate task called direct attribute prediction (DAP) for zero-shot image classification. DAP predicts attributes of an input image and then searches its most similar class according to its attributes. However, this intermediate task may cause a domain shift problem where the model focuses on optimizing attribute classifiers, instead of predicting class labels (Fu et al. 2015). As Figure 10(a) displays, a zebra and a pig both have a tail, but different visual appearance. Figure 10(b)(c) represent zero-shot prototypes as red stars and predicted semantic attribute projections as blue dots. In Figure 10 (b), zebras and pigs have similar attribute space which would cause the model to predict a wrong label for the pig image. This is because it learns each attribute classifier separately but misses the relationships among attributes. Besides, human effort is required for extracting attributes from unseen classes.

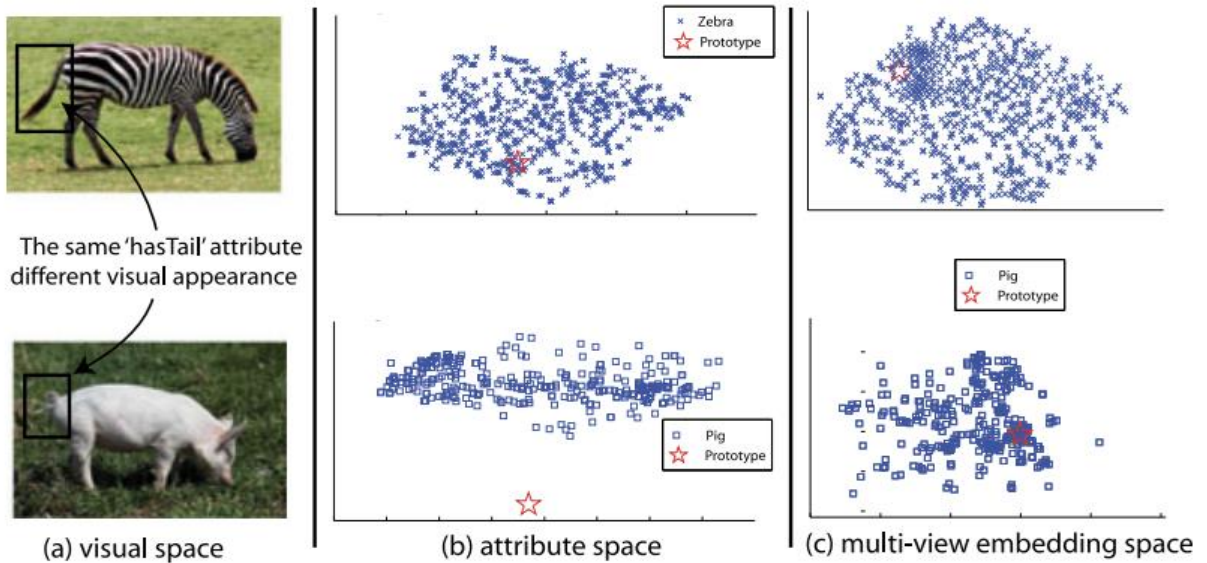


Figure 10. An illustration of the projection domain shift problem. (Fu et al. 2015)

To solve the domain shift problem, researchers try to directly map from visual space to semantic space without intermediate tasks. For example, Akata et al. (2013) propose an approach called Attribute Label Embedding (ALE) to use a bilinear function to measure the compatibility between image embeddings and label embeddings. As Figure 11 shows, its left side is image embedding and right side is label embedding. ALE uses attributes as side information for the label embedding

and measure the “compatibility” between the embedded inputs and outputs with a function  $F$ . During training, image embeddings are the input, label embeddings are the output, the goal is to optimize  $w$  to maximize the image-label pairs’ compatibility. In this way, ALE is able to directly predict on labels without intermediate tasks.

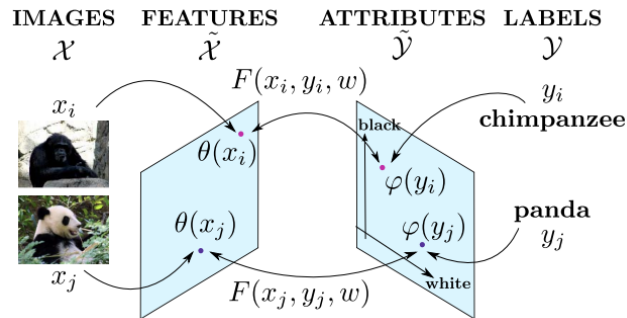


Figure 11. An illustration of ALE. (Akata et al. 2013)

Then, Akata et al. (2015) developed Structured Joint Embedding (SJE) framework based on ALE by replacing manually annotated attributes with multiple side information (e.g. attributes, hierarchical models, and text-based models). SJE includes multiple bilinear functions  $F$  for multiple semantic information sources. In another word, it uses multiple  $W_i$  to capture different visual features. However, bilinear functions are not enough for more complex tasks, so Xian et al. (2016) suggest Latent Embedding Model (LatEm) where a single bilinear function is extended to multiple linear functions to capture diverse visual features (see Figure 12).

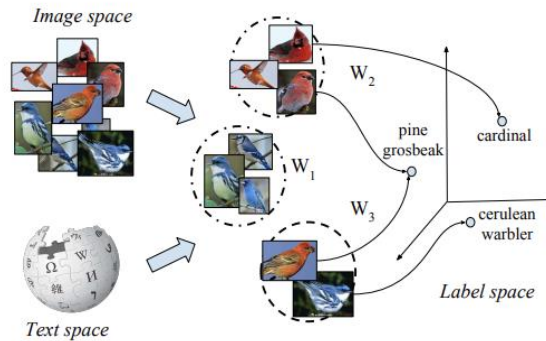


Figure 12. An illustration of LatEm. (Xian et al. 2016)

In addition to directly projecting visual space into semantic space, it is also possible to map both into a third space and optimize their embeddings together. Zhang & Saligrama (2015) propose a method called semantic similarity embedding (SSE) where both visual features and semantic features are projected into histograms separately. For example, in Figure 13 “dog” and “car” are unseen classes represented by light blue color on the left and a query image is represented on the right. Unseen classes and this query image are projected into histograms composed by seen classes separately. As it shows the image histogram is more similar to the label ‘car’ histogram, so it would be predicted as ‘car’.

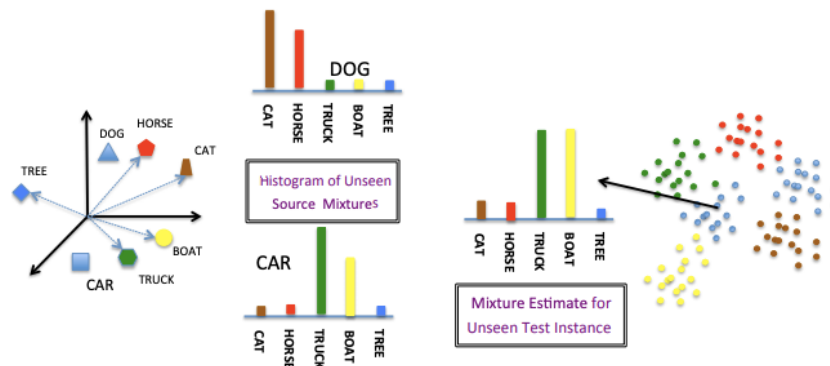


Figure 13. An example of SSE. (Zhang & Saligrama 2015)

Above models all belong to metric-based approaches: learn fixed metrics (e.g. ALE) to embed images (and labels) and then classify images by fixed classifiers (e.g. k-nearest neighbor).

Therefore, research on metric-based approaches usually focus on how to embed input data or how to recognize embedded features.

Additionally, generative ZSL is also a popular direction, because it is closer to how humans think (Bucher et al. 2019). When a human reads a description, he will have a fuzzy image in mind and recognize the target based on this fuzzy image. Similarly, generative models generate synthetic features from texts and classify query image features based on synthetic features. But complex models, including generative ones, may be too memory-consuming, battery-consuming and slow in production.

### **2.3. Word embeddings**

As mentioned above, attribute is one of the popular semantic side information ZSL (Wah et al. 2011; Xian et al. 2018). Attributes are visual characteristics annotated manually, such as “hooked seabird shape” has 59.85% probability to be recognized in class “black footed albatross” (see Figure 4). However, because of the nature of attribute datasets, extending them always require human effort.

Compared with attributes, NLP models are easier to apply to generalized tasks because they use neural network to learn word vectors (i.e. word representing) from free text in unsupervised learning method. For example, as Figure 14 shows, when Word2vec (Mikolov et al. 2013a) learns a new word  $w$ , it actually learns from its  $t$  neighbor words. The other way around, word2vec could predict word  $w$  by its neighbor words. In this way, the distance between two words are not affected by their letters but their neighbor words. For example, NLP models are able to learn the implicit relationships between countries and their capital cities because they are in similar contexts (Mikolov et al. 2013a).

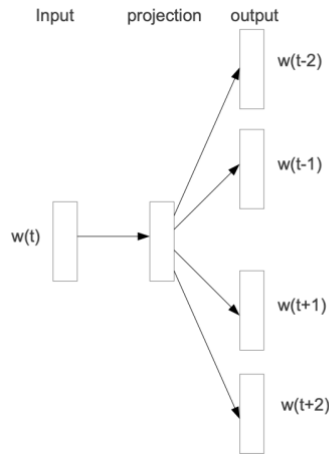


Figure 14. Illustration of the skip-gram model architecture. (Mikolov et al. 2013a)

Furthermore, fastText (Joulin et al. 2016) utilizes characters n-gram to extract word vectors within the target word itself. Different from word2vec, fastText represent a word by a set of characters instead of words. For example, the word *where* with  $n = 3$  can be represented by character n-grams: “wh”, “whe”, “her”, “ere”, “re” and itself “where” (Joulin et al. 2016). With character n-gram, fastText is not limited to the training corpus, because it could represent unseen words.

## 2.4. Relation Network

Sung et al. (2018) propose to utilize CNN to learn embeddings for query images and semantic side information and a rather flexible classifier. They stress that because meta-learning keeps feeding RN random label space during training, it could focus on learning a classifier (i.e. how to compare images and semantic information), instead of learning features of training data. Figure 15 displays that RN consists of an image embedding module and a relation module. In FSL, the image embedding module extracts feature maps from both query images and support images. Then the relation module computes relation scores of each class based on the concatenations of query features and support features. In the end, the class with the highest relation score is the prediction class for query images.



Sung et al. (2018) claim that RN could learn to adapt to new classes through meta-learning, where the model is fed by different support data in each episode. In another word, each episode selects random label space to help RN learn metrics and a classifier that can be shared by all classes. They also stress a well-established meta-learning approach is to learn a set of proper initial parameters, and then fine-tune them with new data. But RN does not require fine-tuning making it easier to be applied on general tasks.

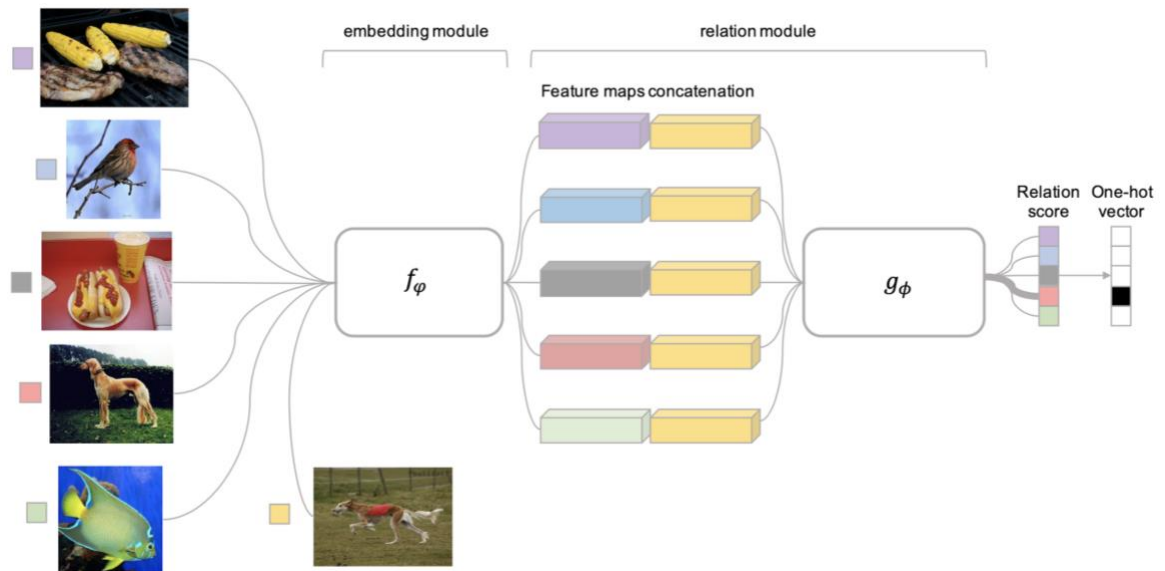


Figure 15. RN pipeline for a 5-way 1-shot image classification problem with one query example. (Sung et al. 2018)

Furthermore, they state and prove RN could be easily extended to ZSL tasks by adding a word embedding module. Semantic feature maps are concatenated with visual feature maps as input of the relation module (see Figure 16). Their experiments results demonstrate that the RN performs better than most classical zero-shot image classification models (e.g. SJE and SSE).

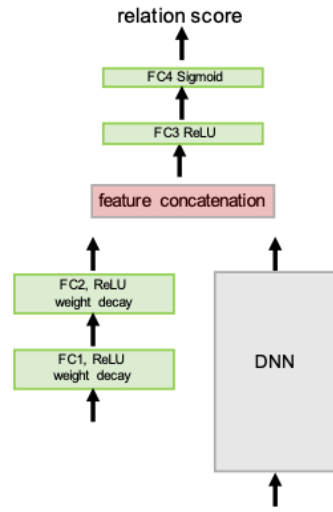


Figure 16. RN architecture for zero-shot learning. (Sung et al. 2018)

Referred to Section 2.2, RN firstly projects visual space and semantic space into a third space (same as SSE) by two embedding modules. Then it classifies images by a relation module. The biggest difference between RN and those ZSL models in Section 2.4 is that RN uses neural networks as its fixed metrics and fixed classifier.

Inspired by RN, Wei et al. (2019) build a baseline model for their few-shot segmentation dataset (i.e. FSS-1000) based on U-Net and RN. As Figure 17 displays, their model consists of encoder module, relation module and decoder module. Compared with the original U-Net, their baseline model does not only embed query images and concatenate query feature maps but also support images and support feature maps. Their proposed dataset (i.e. FSS-1000) only has binary segmentation labels, so they only implement their baseline model on binary semantic segmentation. However, since U-Net itself is designed for binary semantic segmentation tasks, their experiments cannot prove whether the RN improves their baseline model's accuracy.

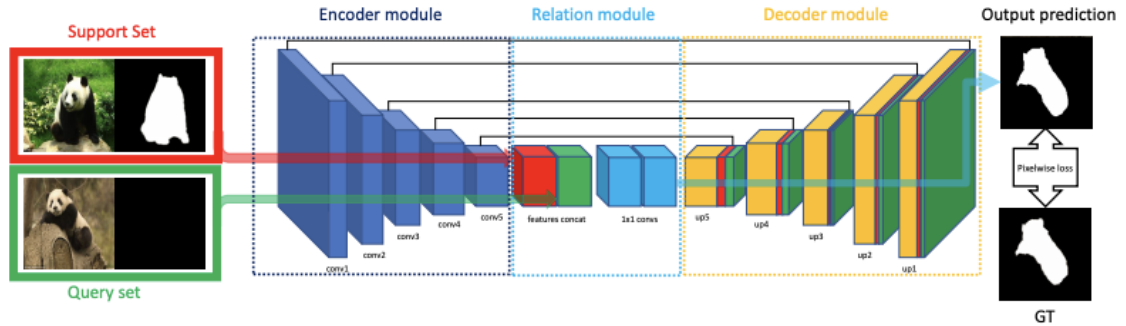


Figure 17. Baseline network architecture of FSS-1000 using VGG-16 as Backbone. (Wei et al. 2019)

## 2.5. Current zero-shot semantic segmentation

Although zero-shot semantic segmentation is still at its early stage, researchers have proven that this task is possible to complete by extending current ZSL and semantic segmentation models. For example, Xian et al. (2019) utilize the structure of DeepLab to design a zero-shot semantic segmentation model called SPNet. As shown in below figure, they remove the last classifier layer of a DNN (i.e. DeepLab or FCN) as their visual-semantic embedding, where  $a, b$  is width and height,  $d_w$  is the size of word vectors. For example, the channel size of word2vec word vectors is 300, so  $d_w$  should be 300 too. Afterwards, in semantic projection these visual-semantic features are projected into the fixed semantic space (i.e. fastText or word2vec) where word vectors are utilized as weights of the projection layer. Projection layer's output has the same depth as label space (i.e.  $|S|$  or  $|U|$ ). In the end SPNet outputs the probability of each class for each pixel. SPNet is similar to ALE as they both project visual space directly into semantic space. In addition to this, Xian et al. (2019) explain the reason why they do not use meta-learning is because it is limited to binary tasks.

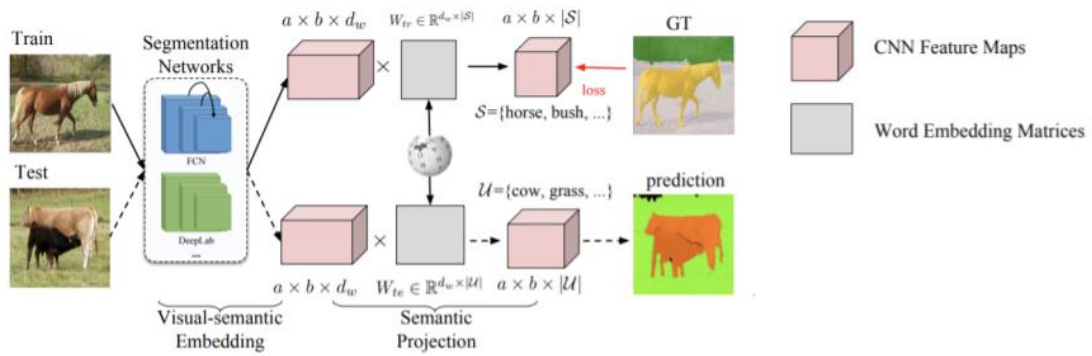


Figure 18. An example of SPNet pipeline including visual semantic embedding and semantic projection under ZLSS setting. (Xian et al. 2019)

Another related model, zero-shot semantic segmentation network (ZS3Net) (Bucher et al. 2019), is a generative ZSL model which produces synthetic image features of unseen classes (see Figure 19). However, ZS3Net is designed for semi-supervised learning where some labels of test instances are available during training, which means train classes and test classes are not disjointed in ZS3Net. Therefore, it could only be applied to GZLSS tasks. Besides, what is worth noting is that in their experiments Bucher et al. (2019) adopt a baseline model that is similar to SPNet and ALE. In their baseline model, they replace the last classifier layer of DeepLab-v3 with a CNN layer which produces word vectors. Then they calculate similarities between predicted vectors and each class’s vectors to classify query images.

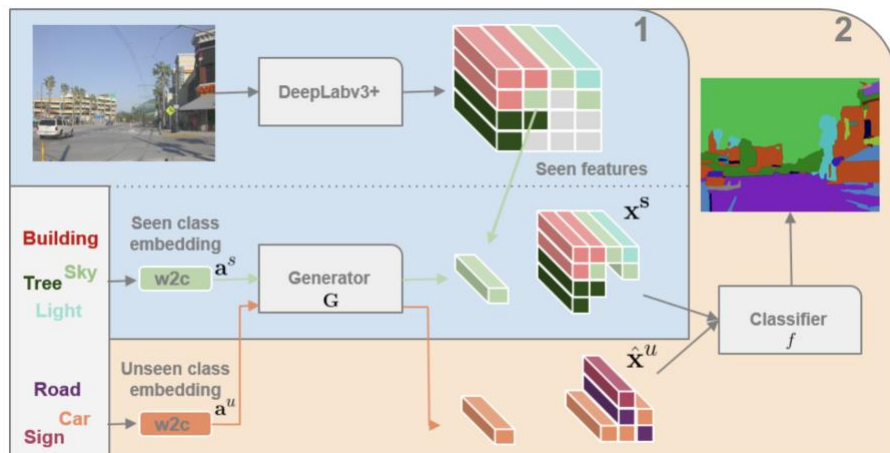


Figure 19. ZS3Net pipeline under GZLSS setting. (Bucher et al. 2019)

### 3. Methodology

#### 3.1. Problem definition

In ZSL, each episode’s input data consists of a query set and a support set that share the same label space. In this thesis, the query set refers to query images  $Q = \{(x_{i,j,k}, y_{i,j,k})\}_{i,j,k=1}^{w,h,n}$ , where  $x_{i,j,k}$  is the pixel at coordinate  $i, j$  of image  $k$ ,  $y_{i,j,k}$  is the label of that pixel, and  $w, h, n$  is the width, height, batch size of images. And let denote seen classes as  $C_s$  and unseen classes as  $C_u$ . The support set refers to semantic side information  $S_{train} = \{v_c ; c \in C_s\}$  where  $v_c$  is attributes or word vectors of class  $c$ . The goal of model is to predict scores  $r_{i,j,k,c}$  of each pixel over each class, and minimize the distance  $f_d$  between prediction scores and ground truth  $y_{i,j,k,c}$ :

$$r_{i,j,k,c} \leftarrow \operatorname{argmin} \sum_{i=1}^w \sum_{j=1}^h \sum_{k=1}^n \sum_{c=1}^C f_d(r_{i,j,k,c}, y_{i,j,k,c}) \quad (1)$$

Because of the meta-learning setting, in each episode a random combination of  $Q$  and  $S_{train}$  is selected. And classes, that are not included in  $S_{train}$ , are ignored in this episode (Xian et al. 2019). In another word, pixels belonging to unselected classes should not affect loss calculation and accuracy calculation.

In multi-class training, each query image features are combined with each class’s semantic information as a pair. For example, if there are  $k$  images and  $c$  classes in one episode, this episode has  $k \times c$  pairs. Figure 20 displays an example pipeline of one episode, where 1 query image and 5 classes are selected. Firstly, semantic and visual feather maps are concatenated as the input of relation module. Then the relation module computes relation scores for each class over each pixel. During multi-class testing, under ZLSS setting the support set  $S_{test} = \{v_c ; c \in C_u\}$ , but under GZLSS setting  $S_{test} = \{v_c ; c \in C_s \cup C_u\}$ . In the binary ZLSS task, training classes and testing classes are disjointed, and each image is only paired with its target class (i.e.  $k$  query images result in  $k$  pairs).

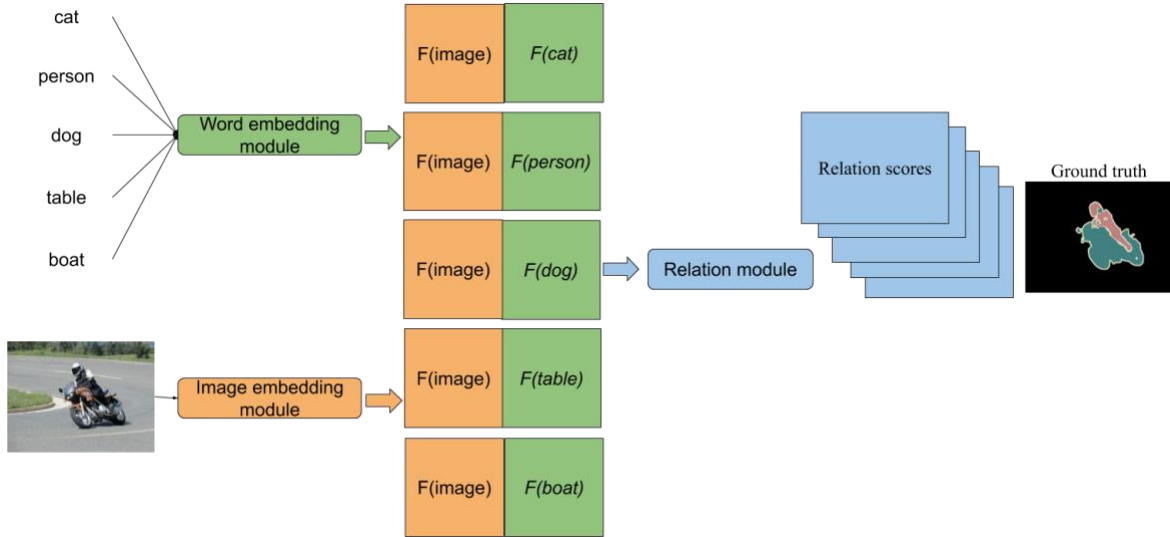


Figure 20. An example pipeline of one episode in multi-class tasks.

This thesis studies on:

- How to extend RN from zero-shot image classification to binary ZLSS, multi-class ZLSS and multi-class GZLSS?
- Does RN have advantages over baseline models?
- What factors would affect the capability of RN and how could it be improved?

### 3.2. Baseline: U-Net and SPNet

For binary zero-shot semantic segmentation tasks U-Net (see Figure 7) is set as the baseline model. Because in these tasks, similar classes could make them be considered as few-shot tasks as well. Moreover, U-Net is designed for few-shot binary semantic segmentation which concatenate downsampling feature maps with upsampling ones. Therefore, this thesis aims to investigate if semantic side information could improve the accuracy of U-Net.

SPNet (see Figure 18) is set as the baseline model for multiclass semantic segmentation tasks. The crucial part of SPNet is projecting image features into semantic space which is flexible in terms of label space and does not require fine tuning. But using DeepLab framework makes their model

heavy to train and apply to applications, so this thesis studies on if RN could outperform it with a simpler architecture.

### 3.3. Evaluation

All models are evaluated by the average intersection over union (mIoU) (2) over classes.  $I_i$  refers to the intersection area of prediction and ground truth of class  $i$ , and  $U_i$  refers to their union area.

$$mIoU = \left( \sum_{i=0}^n \frac{I_i}{U_i} \right) \div n \quad (2)$$

And under GZLSS setting, the models are evaluated by mIoU of unseen classes, mIoU of seen classes, and harmonic mean (H) of them:

$$H = \frac{2 * mIoU_{seen} * mIoU_{unseen}}{mIoU_{seen} + mIoU_{unseen}} \quad (3)$$

## 4. RN for zero-shot semantic segmentation

### 4.1. Network architecture

RN consists of three modules: a word embedding module  $f_{\phi_1}(v_c)$  for extracting semantic features from classes, an image embedding module  $f_{\phi_2}(x_{i,j,k})$  for extracting visual features from query images, and a relation module  $g_\varphi(f_1 \oplus f_2)$  for computing relation scores for each visual features and semantic features pair. Therefore, RN predicts relation scores  $r_{i,j,k,c}$  for each pixel over each class (see Equation 4). The goal of RN is to learn metrics (i.e. embeddings  $f_{\phi_1}$  and  $f_{\phi_2}$ ) and a classifier (i.e. relation module  $g_\varphi$ ) for categorizing the concatenation of feature maps into a class. In semantic segmentation, besides metrics, it also learns a set of classifiers for categorizing concatenations on pixels into classes.

$$r_{i,j,k,c} = g_\varphi \left( f_{\phi_1}(v_c) \oplus f_{\phi_2}(x_{i,j,k}) \right) \quad (4)$$

Relation module is the key point of how to extend RN from image classification to semantic segmentation. Inspired by U-Net and baseline model of FSS-1000 dataset (Wei et al. 2019), relation module uses the same structure as the decoder of U-Net. Based on this, Figure 21 demonstrates RN architecture for the binary ZLSS task (also called 0-shot 2-way task): word embedding module is on the top which transforms semantic information into proper channels for concatenation, image embedding is on the left which is ImageNet pretrained VGG16 (Zhang et al. 2015), and relation module is on the right which predicts scores for each pixel. As it shows, the relation module consists of upsampling and CNN layers, where feature maps from image embedding module and word embedding module are utilized to improve its accuracy.



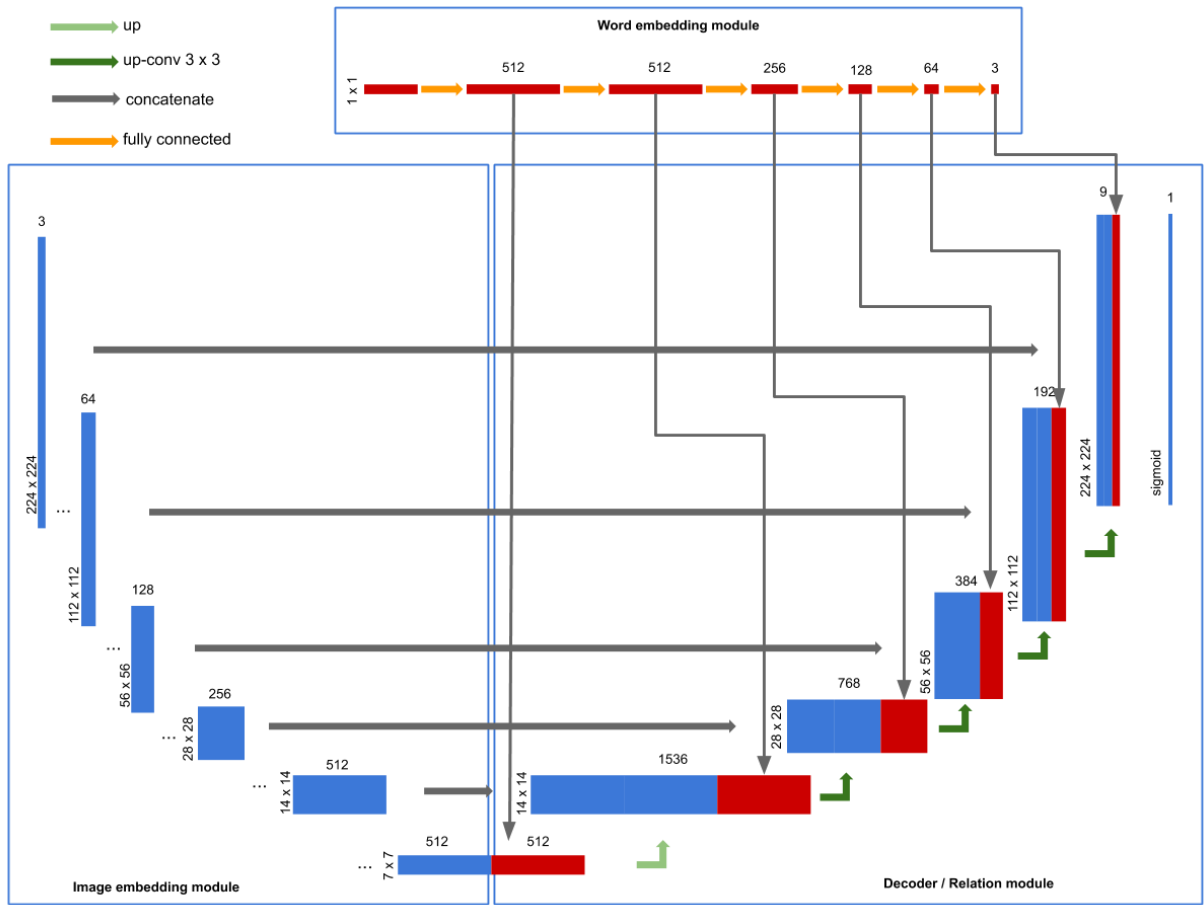


Figure 21. Relation Network architecture for binary semantic segmentation tasks with pre-trained VGG16 and U-Net decoder.

Inspired by SPNet, another method is using a DNN as image embeddings and directly classify on the concatenation of semantic feature maps and visual feature maps by CNN layers. Figure 22 displays RN architecture of using DeepLab-v3 (without its last 4 classifier layers) as image embeddings, ImageNet pretrained ResNet101 (He et al. 2016) as its backbone, FC layers as word embeddings and upsampling and CNN layers as relation module.

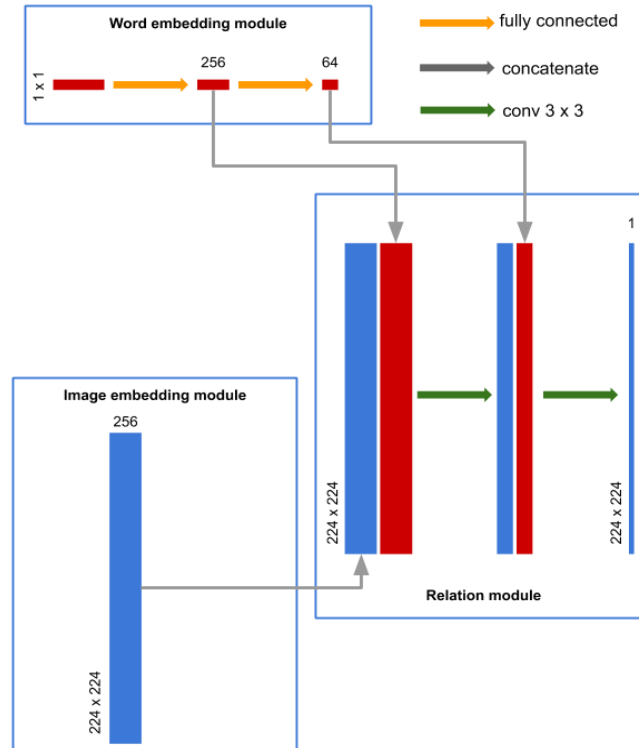


Figure 22. Relation Network architecture for binary semantic segmentation tasks with DeepLab-v3 removing its last 4 classifier layers.

Moreover, the word embedding module could be composed by FC layers or upsampling and CNN layers. Neural networks learn feature maps from input data by kernels. In fully connected (FC) layers, kernels have the same size as the input, but in convolutional neural networks (CNN) layers kernel size is smaller than input (e.g.  $3 \times 3$ ). As Figure 23 demonstrates, with the same input (i.e.  $5 \times 5$ ) FC kernel on the left has the same size of input but CNN kernel on the right is  $3 \times 3$ , and each kernel multiplies with all channels' feature maps and sums up its results. Therefore, when the input data size is  $10 \times 15 \times 5 \times 5$ , both FC and CNN layers have 5 kernels, step size is 1 and no padding, the output of FC layer would be  $10 \times 5 \times 5 \times 5$ , but the output of CNN layer would be  $10 \times 5 \times 3 \times 3$ . Compared with FC, CNN focuses on a part of the input data at a time, which is helpful in Computer Vision. For example, when we recognize a cat from an image, what helps us are those key features (e.g., ears and eyes), instead of the whole image. Therefore, using a smaller kernel helps models to learn more useful feature maps.

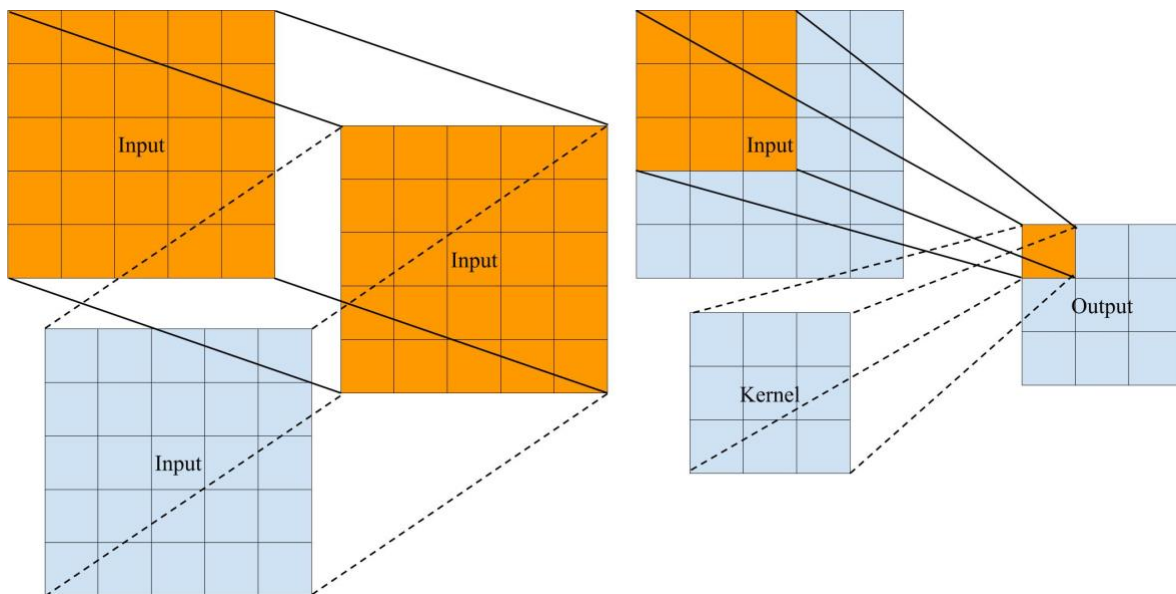


Figure 23. Comparison of FC kernel (left) and CNN kernel (right).

With FC layers its input is 2-dimensional data (i.e. batch size  $\times$  channel size). FC layers transform the channel size of semantic feature maps (e.g. from 300 to 512), so semantic feature maps need to be repeated in width and height before concatenation. With CNN layers, its input is 4-dimensional data (i.e. batch size  $\times$  channel size  $\times$  width  $\times$  height) which can be upsampled and transformed to the proper size. Using FC layers is closer to the original concept of RN where each pixel is concatenated with the same semantic feature maps and makes model size smaller in this case. But the other one may provide more context information for semantic segmentation models.

In the end, relation module uses sigmoid at its last layer to restrict relation scores between 0 and 1 representing the probability of this pixel belonging to the target class. In the binary ZLSS task, if a score is higher than or equal to 0.5, this pixel is predicted as target class, otherwise it is predicted as background.

In  $C$ -way 0-shot semantic segmentation tasks, as shown in Figure 24, RN predicts relation scores for each word-image pair and compare relation scores among classes. Then for each pixel the class with the highest relation score is its prediction class. With this network architecture, a multi-class segmentation task is essentially a series of binary segmentation tasks.

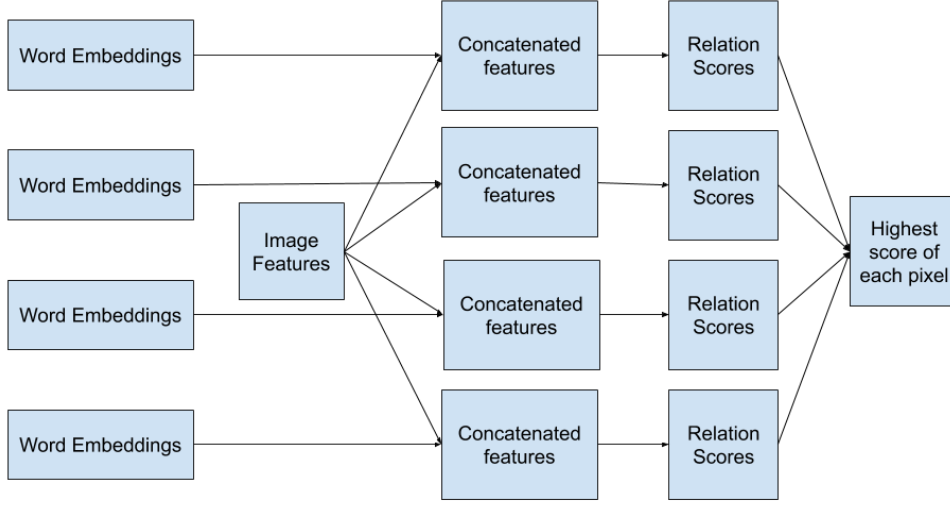


Figure 24. C-way 0-shot semantic segmentation pipeline.

## 4.2. Loss function

In both binary segmentation training and multiclass segmentation training, binary cross entropy (BCE) loss (5) is adopted for backpropagation, where  $r_{i,j,k,c}$  is the relation score of a pixel over class  $c$  and  $y_{i,j,k,c}$  is the ground truth of it. If this pixel belongs to class  $c$ , its ground truth is 1, otherwise it is 0. Mean square error (MSE) could also be used here but does not make much difference. As for categorical cross entropy, experiments show that its convergence speed is not as fast as BCE.

$$\omega, r_{i,j,k,c} \leftarrow \operatorname{argmin} \sum_{i=1}^w \sum_{j=1}^h \sum_{k=1}^n \sum_{c=1}^C w(-y_{i,j,k,c} \log(r_{i,j,k,c}) - (1 - y_{i,j,k,c}) \log(1 - r_{i,j,k,c})) \quad (5)$$

## 5. Experiments

### 5.1. Datasets and splits

Sung et al. (2018) used Animals with Attributes (AWA2) (Xian et al. 2018) and CUB for zero-shot image classification. They both provide attributes information which could be input of the word embedding module. However, AWA2 does not have segmentation labels. Whereas, besides 200 bird species and corresponding 312 numeric attributes values, CUB also provides rough segmentation labels for each image. And PASCAL VOC 2012 (Everingham et al. 2012) and COCO-Stuff are also popular semantic segmentation datasets (Xian et al. 2019; Bucher et al. 2019). PASCAL contains 2913 images with segmentation annotations, covering 20 classes and 1 background class. COCO has 123K annotated images, including 182 classes and 1 background class.

So, experiments are made on CUB, PASCAL and COCO in this thesis. The train/test split of CUB is provided by Sung et al. (2018) from their GitHub repository: 150 seen classes and 50 unseen classes. Because there is only one target class in each image, its class split is the same as its image split and this dataset can be utilized in both binary segmentation and multiclass segmentation. As for PASCAL and COCO, I directly adopt the class train/test splits proposed by Xian et al. (2019): split classes based on whether they are included in ImageNet 1K because the pre-trained models are trained on it. In PASCAL the last 5 classes are unseen classes (i.e., potted plant, sheep, sofa, train and tv/monitor) and the rest 15 ones are seen classes. In COCO 15 classes are unseen classes (i.e. cow, giraffe, suitcase, frisbee, skateboard, carrot, scissors, cardboard, clouds, grass, playing field, river, road, tree and wall-concrete) and the rest 167 are seen classes. The train/test splits from PASCAL and COCO are directly used as the image split. Unseen classes are ignored in loss function during training, seen classes are ignored in accuracy calculation under ZLSS setting, and no class is ignored in accuracy calculation under GZLSS setting.

As for word embeddings, all datasets could be trained with NLP models (i.e., word2vec and fastText). Following choices of Xian et al. (2019), my experiments use Google News (Mikolov et

al. 2013b) pre-trained word2vec model, Common Crawl (Mikolov et al. 2017) pre-trained fastText model and their concatenation. When a class has multiple words, their word vectors are summed.

## 5.2. Implementation details

Experiments are implemented with PyTorch (Paszke et al. 2019). Pre-trained models and DeepLab-v3 framework are imported from PyTorch sub-package “models”. The U-Net based model uses VGG16 as its image embeddings module and DeepLab-v3 uses ResNet101 as its backbone. Both VGG16 and ResNet101 are pre-trained on ImageNet 1K. Implementation codes of preprocessing, training and testing on PASCAL and models are demonstrated in Appendix B, C, D, and E.

For pre-processing the data, all the input images, which are in JPG format, are resized to size (224,224) and normalized with mean value [0.485, 0.456, 0.406] and std value [0.229, 0.224, 0.225] to be aligned with ImageNet pre-trained models implemented by PyTorch team. All the labels, which are in PNG format, are converted to classes index. Specifically, each pixel in binary segmentation labels (i.e. CUB labels) are converted to 0 as background or 1 as target classes. As for multi-class segmentation labels (i.e. PASCAL and COCO-Stuff) 0 indicates background, -1 indicates difficult or ambiguous pixels, positive numbers represent classes. PASCAL annotations are represented by 22 different colors, and during preprocessing they are converted to numbers from -1 to 20 (see its palette in Appendix A). COCO annotations are indexed images where 0-181 refers to classes and 255 refers to background (COCO do not annotate difficult pixels). Besides, it is worth noting that annotation images are resized by OpenCV (Bradski 2000) with “INTER\_NEAREST” parameter to avoid generating new pixels.

## 5.3. Binary semantic segmentation

While training each model, the total episode number is 50,000, batch size is 32, learning rate is  $1e-6$  and is reduced by 50% every 5,000 episodes, and Adam is chosen as optimizer.

### 5.3.1. Effect of word embeddings

Compared with U-Net, combination of U-Net and RN has higher mIoU over classes, as demonstrated in Table 1. Among four different word embeddings, the model with CUB attributes has the best accuracy which achieves 81.18% on unseen classes and 81.84% on seen classes. But the accuracy difference between the attributes-based model and other three is smaller than 3%, so this may be not enough to imply that in semantic segmentation tasks, human annotated visual side information is more helpful than NLP models. Besides, different from experiments of Xian et al. (2019), concatenation of word2vec and fastText does not have obvious advantages compared with other word embeddings.

	unseen class mIoU (%)	seen class mIoU (%)	H (%)
No word embeddings	74.69	74.41	74.55
CUB attributes	<b>81.18</b>	<b>81.84</b>	<b>81.51</b>
word2vec	78.69	78.82	78.75
fastText	79.79	79.92	79.86
word2vec + fastText	78.51	78.69	78.60

Table 2. Effect of word embeddings: mIoU of binary semantic segmentation on CUB with U-Net based model.

### 5.3.2. Effect of network structure

When use FC layers in the word embedding module, semantic feature map size is always  $channels \times 1 \times 1$ , and then they are repeated to the same width and height as the visual features. And when use CNN layers, word embeddings are upsampled and then transformed by CNN layers with  $3 \times 3$  size kernels. As below table displays, after training two U-Net based RN models with the same semantic information (i.e. CUB attributes), the one uses FC layers performs slightly better than the one uses CNN layers. Besides, using FC layers decreases model size, because its kernel size is only  $1 \times 1$ . Therefore, following experiments all adopt FC layers in their word embeddings.

	unseen class mIoU (%)	seen class mIoU (%)	H (%)
FC layers	<b>81.18</b>	<b>81.84</b>	<b>81.51</b>
CNN layers	78.12	78.79	78.45

Table 3. Effect of word embeddings module structure: mIoU of binary semantic segmentation on CUB and its attributes with U-Net based models.

It is obvious that the U-Net framework performs better than DeepLab in this binary task. In Table 4 both models are trained and evaluated with CUB images and CUB attributes and the U-Net based model outperforms DeepLab based model in all criteria.

	unseen class mIoU (%)	seen class mIoU (%)	H (%)
U-Net_VGG16	81.18	81.84	81.51
DeepLab_Resnet101	72.85	74.17	73.51

Table 4. Effect of network structure: mIoU of binary semantic segmentation on CUB and CUB attributes with U-Net based model.

#### 5.4. Multi-class semantic segmentation

In multi-class training, the total episode number is 50,000 for PASCAL, 100,000 for COCO. The initial learning rate is  $1e-4$  and is reduced by 50% every 10,000 episodes, and optimizer is Adam. The batch size for U-Net based models is 32, but for DeepLab based models is 8 because of the limit of random access memory (RAM) (i.e. 16 GB). After trying using checkpoints and adjusting activation functions' parameters to reduce memory usage, I still had to reduce its batch size. In each episode, 10 classes are randomly selected from all classes, and then unseen classes are excluded from the label space. Because COCO has many more classes (i.e. 182 classes), to speed its converging speed, labels are randomly selected within the label space of each episode.

When calculating the loss, unselected classes are ignored (i.e. unselected seen classes and all unseen classes). During testing, under ZLSS setting label space is unseen classes (i.e. 5 classes in



PASCAL and 15 classes in COCO) and under GZLSS setting label space is both seen and unseen classes.

#### 5.4.1. Effect of word embeddings

As Table 5 and Table 6 show, three kinds of word vectors are adopted to study the effect of word embeddings on PASCAL. With U-Net based models, similar experiment results as Xian et al. (2019) are collected: the concatenation of word2vec and fastText has the best performance among them. However, the concatenated vectors have poor performance with the DeepLab based model.

Moreover, although label space is randomly chosen during training, their loss is still able to decrease to  $1e-7$  and the best ZLSS mIoU is reached within the first 15,000 episodes. Furthermore, in GLZSS the mIoU over seen classes is much higher than the mIoU over unseen classes. These indicate models are able to converge with seen classes, but is overfitting on PASCAL, causing poor performance with unseen classes. Besides, in Table 5 the model with concatenated vectors ends with higher loss than the other two, which indicates its overfitting is less serious. One reason for overfitting is that training data is not diverse enough because there are only 20 classes in PASCAL.

	ZLSS mIoU (%)	GZLSS unseen mIoU (%)	GZLSS seen mIoU (%)	GZLSS H (%)	Lowest loss
word2vec	40.09	10.87	51.03	17.92	$1e-7$
fastText	41.84	15.75	52.56	24.23	$1e-7$
w2v + ft	48.93	16.09	50.49	24.40	$1e-4$

Table 5. Effect of word embeddings: mIoU of multi-class semantic segmentation on PASCAL with U-Net and VGG16 based models.

	ZLSS mIoU (%)	GZLSS unseen mIoU (%)	GZLSS seen mIoU (%)	GZLSS H (%)	Lowest loss
word2vec	41.38	17.13	67.80	27.35	1e-3
fastText	40.02	7.6	58.66	13.45	1e-3
w2v + ft	36.18	5.27	39.14	9.29	1e-3

Table 6. Effect of word embeddings: mIoU of multi-class semantic segmentation on PASCAL with DeepLab and ResNet101 based models.

#### 5.4.2. Effect of network structure

Although DeepLab has a more complex structure, it does not always have better results than U-Net in this case. By comparing Table 5 and Table 6, it is observed that DeepLab based models are more difficult to converge, which indicates its overfitting problem as less serious than U-Net based models. The first model (i.e. the one uses word2vec) in Table 5 does not have the highest ZLSS mIoU, but it reaches the best harmonic mean among above two tables. However, the other two DeepLab based models have rather poor performance, so it is hard to say whether DeepLab has better capability on multi-class tasks.

#### 5.4.3. Effect of data

As there are only 20 classes in PASCAL, the label spaces are highly overlapped among episodes. However, meta learning aims to feed random and various classes to the model to make it adapt to new classes. As mentioned in Section 4.4.1, RN is overfitting on PASCAL, so to investigate whether more diverse training data could boost its performance, RN is also trained and evaluated on COCO. With word2vec as semantic side information, VGG16 as image embedding module, U-Net as relation module, RN is able to reach 37.37% of ZLSS mIoU, 6.42% of GZLSS harmonic mean (3.80% over unseen classes and 20.55% over seen classes). Furthermore, models trained on these two datasets are cross evaluated on each other. As shown in Table 6, COCO trained model surpasses PASCAL trained model in ZLSS of both datasets.

ZLSS mIoU (%)	PASCAL-evaluated	COCO-evaluated
PASCAL-trained	40.09	5.09
COCO-trained	<b>78.69</b>	<b>37.37</b>

Table 7. PASCAL and COCO trained U-Net based models with word2vec and VGG16 cross evaluated on each other’s testing data.

Meanwhile, size of objects could also affect prediction scores. Below two figures demonstrate the relationship between IoU and average object size of unseen classes from PASCAL and COCO under ZLSS setting. For PASCAL classes and most COCO classes, it is obvious that IoU has a significantly positive correlation with object size.

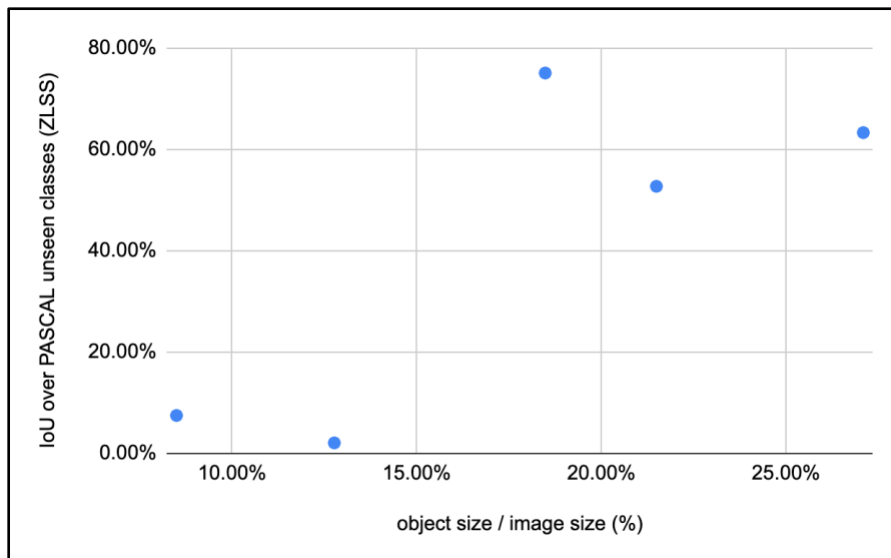


Figure 25. Relationship between IoU and average object size of unseen classes in PASCAL under ZLSS setting.

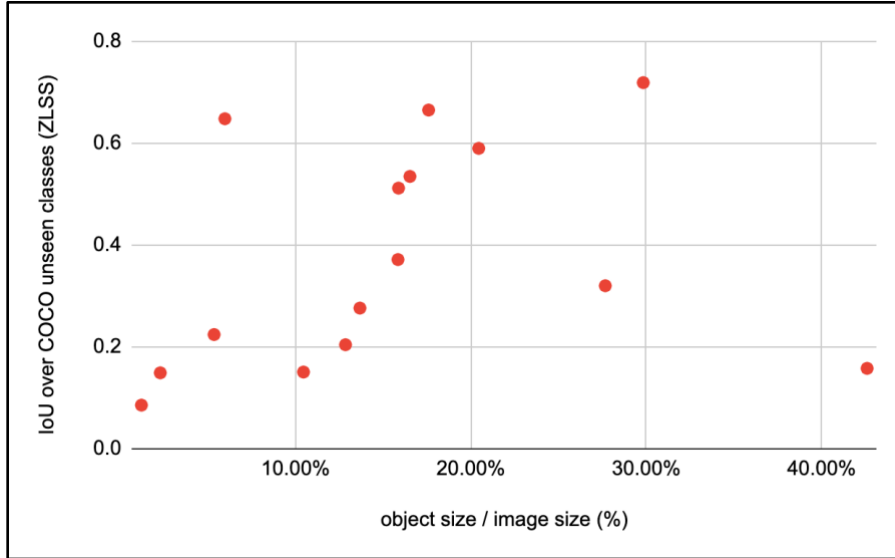


Figure 26. Relationship between IoU and average object size of unseen classes in COCO under ZLSS setting.

## 6. Discussion

### 6.1. Compare with baseline models

In binary ZLSS, compared with the baseline model U-Net, RN is able to improve the accuracy with the help of semantic side information (see Table 1). But in multi-class tasks, accuracy of RN depends on how diverse training data is, as shown in Table 8 and Table 9 where their best results are picked up for comparison. In both tables, SPNet adopts concatenation of word2vec and fastText as its word embeddings and DeepLab as its framework. But RN uses VGG16 and U-Net as its image embedding module and relation module separately. In Table 8 RN uses concatenation of word2vec and fastText as well, but in Table 9 it uses word2vec only.

When RN is trained on PASCAL, its ZLSS mIoU is close to SPNet but its GZLSS accuracy is much better than SPNet. And as expected, a bigger dataset (i.e. COCO-Stuff) could boost performance of RN with meta-learning. After only 50,000 episodes, RN achieves higher mIoU than SPNet in both ZLSS and GZLSS tasks on COCO. But its loss (i.e.  $5e-2$ ) is still big and its mIoU on seen classes (i.e. 20.07%) is rather low, compared with other experiments, implying it is underfitting.

However, Xian et al. (2019) is able to increase the GZLSS accuracy of SPNet dramatically by reducing its prediction scores on seen classes (i.e. calibration). After calibrating, SPNet-C could maintain its mIoU on seen classes and increase its mIoU on unseen classes. On the other hand, although calibrating RN could improve its unseen mIoU a little, it would decrease seen mIoU dramatically, resulting in lower harmonic mean.

	ZLSS mIoU (%)	GZLSS unseen mIoU (%)	GZLSS seen mIoU (%)	GZLSS H (%)
RN	48.93	16.09	50.49	24.40
SPNet	49.50	0.01	75.51	0.02
SPNet-C		29.33	76.84	42.45

Table 8. Comparison of mIoU between SPNet on PASCAL. (SPNet data is from Xian et al. 2019)

	ZLSS mIoU (%)	GZLSS unseen mIoU (%)	GZLSS seen mIoU (%)	GZLSS H (%)
RN	37.37	3.80	20.55	6.42
SPNet	35.20	0.20	34.05	0.33
SPNet-C		8.33	34.52	13.42

Table 9. Comparison of mIoU between SPNet on COCO. (SPNet data is from Xian et al. 2019)

## 6.2. Qualitative results

According to mIoU (see Table 2), the effect of different word embeddings is small. But in Figure 27, it is obvious that attributes boost the model’s accuracy and fastText performs better than word2vec. This is because in Figure 27 prediction scores (i.e. from 0 to 1) are directly shown in images, but in Table 1 prediction scores are converted to prediction labels (i.e. 0 or 1) to calculate the mIoU.

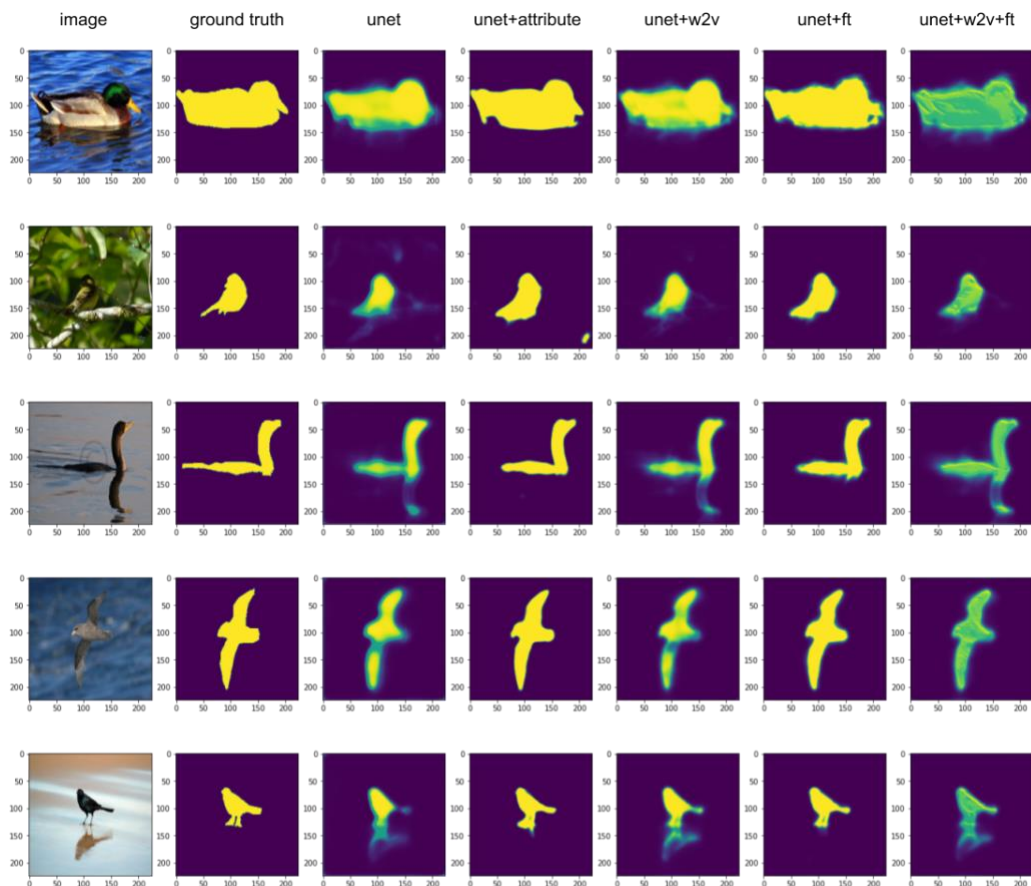


Figure 27. Qualitative results on CUB unseen classes by U-Net based models with no-vector, attributes, word2vec, fastText and the concatenation of word2vec and fastText.

Figure 28 displays the segmentation results on PASCAL predicted by different RNs. The first three models from the left are trained on PASCAL with the same U-Net structure but different word embeddings. The fourth one adopts DeepLab structure and word2vec embeddings. The last one has the same setting as the first model, but is trained on COCO. Different from the above binary task, each predicted pixel in Figure 28 is converted to RGB format by PASCAL palette. Because in ZLSS the search scope is unseen classes, all seen classes are represented by black color as the background. And ZLSS results on COCO are shown in Figure 29 made by a U-Net based model with wor2vec embeddings. Seen classes are drawn in black, but unseen classes are represented directly by their class indexes. At each row, expected classes are listed on the left.

As analyzed in Section 4.4.3, big objects (e.g. trains and sofas) are easier to be recognized than small objects. Additionally, the poor recognition of monitors may be caused by their colorful screens which is misleading when the data is not diverse enough. Although the COCO trained model could segment monitors well, it still has difficulty on sheep and glass (see the glass part of trains in Figure 28).

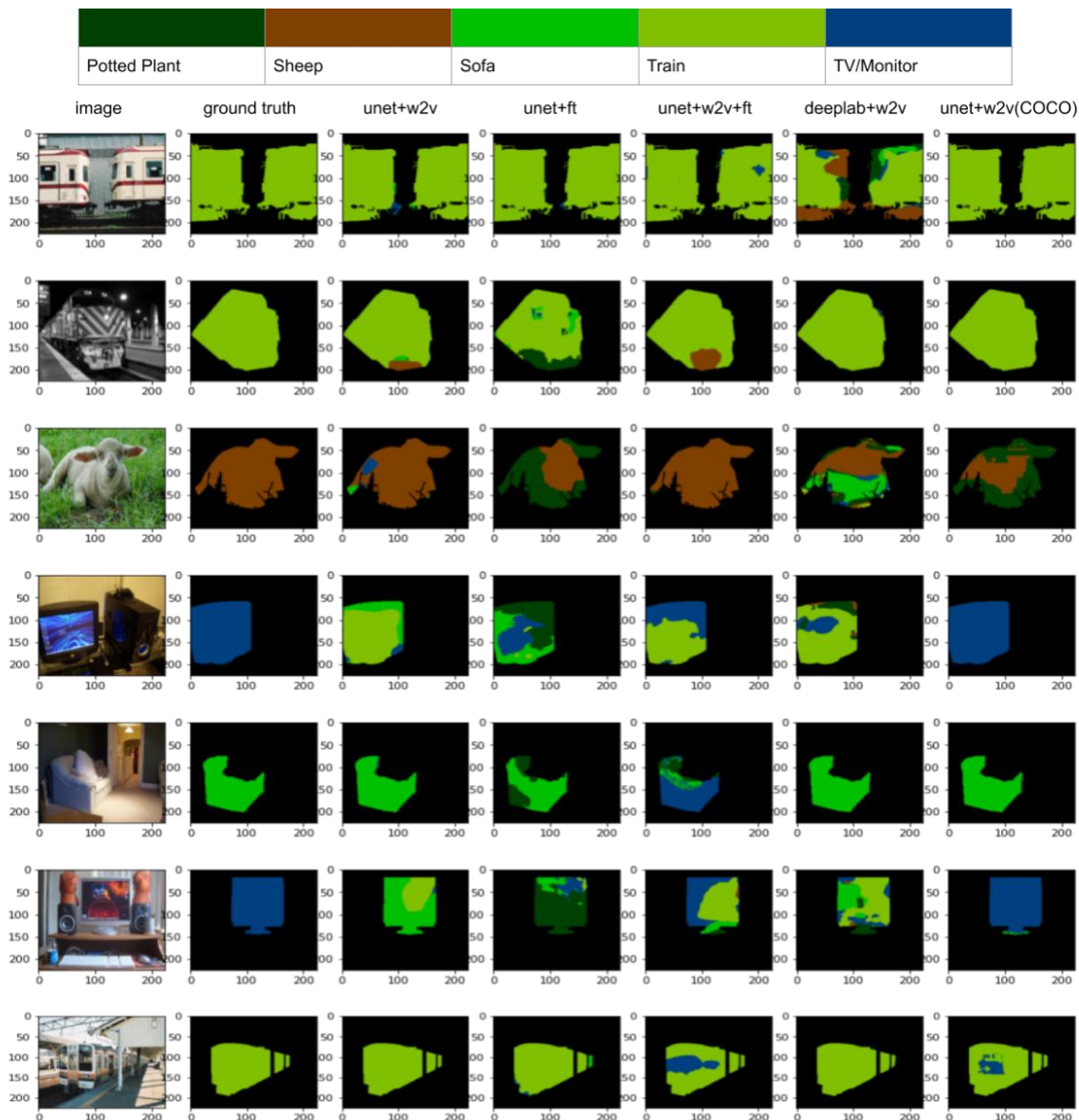


Figure 28. Qualitative results on PASCAL unseen classes under ZLSS setting by RN.





Figure 29. Qualitative results on COCO unseen classes under ZLSS setting by U-Net based model with word2vec.

## 7. Conclusion

This thesis proves it is feasible to extend RN from zero-shot image classification tasks to ZLSS and GZLSS tasks. Because semantic feature maps of each class are concatenated with visual feature maps separately, RN is limited to binary semantic segmentation. But this thesis manages to do multi-class semantic segmentation tasks by joining multiple binary semantic segmentation pipelines. U-Net and SPNet frameworks are referred to in network structure designing. ImageNet pretrained DNNs (i.e. VGG16, ResNet101 and DeepLab-v3) are adopted in image embedding module, attributes and NLP models are utilized as word embeddings.

In the binary ZLSS task on CUB, although there is only one object in each image and classes are similar to each other, RN could improve semantic segmentation accuracy with the help of semantic information. And in multi-class tasks, the effect of data is obvious. Because RN is trained by meta-learning, more diverse training data could boost its accuracy. As above experiments shown, in ZLSS tasks, RN has a close mIoU as SPNet on PASCAL, but outperforms it on COCO. However, under GZLSS setting, although RN has obviously higher harmonic mean than SPNet, SPNet can effectively improve its accuracy by calibrating, whereas calibrating does not help RN. The poor performance in GZLSS tasks may be caused by U-Net structure, as U-Net was originally designed for binary segmentation tasks and may not be suitable for multi-class ones. Moreover, DeepLab could achieve better harmonic mean than U-Net but has a bigger model size and needs more time to train.

In terms of practical application, U-Net based RN has a smaller model size and a simpler architecture than SPNet and could maintain a similar ZLSS mIoU. But as the number of classes grows in GZLSS tasks, U-Net based models have rather low mIoU. DeepLab based models may have better results in multi-class tasks but costs more RAM usage.

In experiments, the effect of different settings is also studied for optimizing the capability of RN. In binary segmentation, although attributes are more helpful than NLP models, they are more expensive to extend and apply on real problems. Overall, NLP models have unstable performance

in experiments, so it is hard to make a conclusion which one is the best in this case. While choosing between U-Net and DeepLab, the former one has an advantage when the search space is small.

Nevertheless, while designing network structure, this thesis simply removes the last 4 classifier layers of DeepLab and appended upsampling and CNN layers to it. But there may be other options on how to utilize DeepLab framework. Moreover, different semantic side information (e.g. Ontologies) could also be utilized to improve their accuracy. Because this thesis is a part of preparation of a scientific paper, the capability of DeepLab based models and more diverse semantic information could be further investigated based on this thesis in future work. And it is worth noting because of limited GPU RAM, DeepLab based models were trained with batch size = 8, which would influence convergence.

## References

- Akata, Zeynep, Scott Reed, Daniel Walter, Honglak Lee, and Bernt Schiele. "Evaluation of output embeddings for fine-grained image classification." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2927-2936. 2015.
- Akata, Zeynep, Florent Perronnin, Zaid Harchaoui, and Cordelia Schmid. "Label-embedding for attribute-based classification." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 819-826. 2013.
- Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla. "Segnet: A deep convolutional encoder-decoder architecture for image segmentation." *IEEE transactions on pattern analysis and machine intelligence* 39, no. 12 (2017): 2481-2495.
- Bradski, G. "The OpenCV Library".Dr. Dobb's Journal of Software Tools. 2000.
- Bucher, Maxime, V. U. Tuan-Hung, Matthieu Cord, and Patrick Pérez. "Zero-Shot Semantic Segmentation." In *Advances in Neural Information Processing Systems*, pp. 466-477. 2019.
- Caesar, Holger, Jasper Uijlings, and Vittorio Ferrari. "Coco-stuff: Thing and stuff classes in context." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1209-1218. 2018.
- Chen, Liang-Chieh, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L. Yuille. "Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs." *IEEE transactions on pattern analysis and machine intelligence* 40, no. 4 (2017): 834-848.
- Chen, Liang-Chieh, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. "Encoder-decoder with atrous separable convolution for semantic image segmentation." In *Proceedings of the European conference on computer vision (ECCV)*, pp. 801-818. 2018.

- Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database." In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248-255. Ieee, 2009.
- Everingham, M., L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. "The pascal visual object classes challenge 2012 (voc2012) results (2012)." In URL <http://www.pascal-network.org/challenges/VOC/voc2011/workshop/index.html>. 2011.
- Fu, Yanwei, Timothy M. Hospedales, Tao Xiang, and Shaogang Gong. "Transductive multi-view zero-shot learning." *IEEE transactions on pattern analysis and machine intelligence* 37, no. 11 (2015): 2332-2345.
- He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778. 2016.
- Joulin, Armand, Edouard Grave, Piotr Bojanowski, Matthijs Douze, H erve J egou, and Tomas Mikolov. "Fasttext. zip: Compressing text classification models." *arXiv preprint arXiv:1612.03651* (2016).
- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In *Advances in neural information processing systems*, pp. 1097-1105. 2012.
- Lampert, Christoph H., Hannes Nickisch, and Stefan Harmeling. "Attribute-based classification for zero-shot visual object categorization." *IEEE transactions on pattern analysis and machine intelligence* 36, no. 3 (2013): 453-465.
- Long, Jonathan, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431-3440. 2015.

Mikolov, Tomas, Edouard Grave, Piotr Bojanowski, Christian Puhersch, and Armand Joulin. "Advances in pre-training distributed word representations." *arXiv preprint arXiv:1712.09405*. 2017.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. "Distributed representations of words and phrases and their compositionality." In *Advances in neural information processing systems*, pp. 3111-3119. 2013a.

Mikolov, Tomas, Kai, Chen, Greg, Corrado, and Jeffrey, Dean. "GoogleNews-vectors-negative300.bin.gz - Efficient estimation of word representations in vector space".*arXiv preprint arXiv:1301.3781*. 2013b.

Mottaghi, Roozbeh, Xianjie Chen, Xiaobai Liu, Nam-Gyu Cho, Seong-Whan Lee, Sanja Fidler, Raquel Urtasun, and Alan Yuille. "The role of context for object detection and semantic segmentation in the wild." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 891-898. 2014.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen et al. "PyTorch: An imperative style, high-performance deep learning library." In *Advances in Neural Information Processing Systems*, pp. 8024-8035. 2019.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-net: Convolutional networks for biomedical image segmentation." In *International Conference on Medical image computing and computer-assisted intervention*, pp. 234-241. Springer, Cham, 2015.

Russakovsky, Olga, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang et al. "Imagenet large scale visual recognition challenge." *International journal of computer vision* 115, no. 3 (2015): 211-252.

Sung, Flood, Yongxin Yang, Li Zhang, Tao Xiang, Philip HS Torr, and Timothy M. Hospedales. "Learning to compare: Relation network for few-shot learning." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1199-1208. 2018.

- Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818-2826. 2016.
- Wah, Catherine, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. "The caltech-ucsd birds-200-2011 dataset." 2011.
- Wei, Tianhan, Xiang Li, Yau Pun Chen, Yu-Wing Tai, and Chi-Keung Tang. "Fss-1000: A 1000-class dataset for few-shot segmentation." *arXiv preprint arXiv:1907.12347* (2019).
- Xian, Yongqin, Christoph H. Lampert, Bernt Schiele, and Zeynep Akata. "Zero-shot learning—A comprehensive evaluation of the good, the bad and the ugly." *IEEE transactions on pattern analysis and machine intelligence* 41, no. 9 (2018): 2251-2265.
- Xian, Yongqin, Subhabrata Choudhury, Yang He, Bernt Schiele, and Zeynep Akata. "Semantic projection network for zero-and few-label semantic segmentation." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 8256-8265. 2019.
- Xian, Yongqin, Zeynep Akata, Gaurav Sharma, Quynh Nguyen, Matthias Hein, and Bernt Schiele. "Latent embeddings for zero-shot classification." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 69-77. 2016.
- Zhang, Ziming, and Venkatesh Saligrama. "Zero-shot learning via semantic similarity embedding." In *Proceedings of the IEEE international conference on computer vision*, pp. 4166-4174. 2015.
- Zhang, Xiangyu, Jianhua Zou, Kaiming He, and Jian Sun. "Accelerating very deep convolutional networks for classification and detection." *IEEE transactions on pattern analysis and machine intelligence* 38, no. 10 (2015): 1943-1955.

# Appendices

## A. Palette of PASCAL VOC 2012

[0,0,0]	[128,0,0]	[0,128,0]	[128,128,0]	[0,0,128]	[128,0,128]
background	Aeroplane	Bicycle	Bird	Boat	Bottle
[0,128,128]	[128,128,128]	[64,0,0]	[192,0,0]	[64,128,0]	[192,128,0]
Bus	Car	Cat	Chair	Cow	DiningTable
[64,0,128]	[192,0,128]	[64,128,128]	[192,128,128]	[0,64,0]	[128,64,0]
Dog	Horse	Motorbike	Person	PottedPlant	Sheep
[0,192,0]	[128,192,0]	[0,64,128]	Others		
Sofa	Train	Tv/Monitor	border/difficult pixels		

## B. Implementation codes of network structure

```

import torch
import torch.nn as nn
from torchvision import models
import torch.nn.functional as F

class RelationNetworkWithVGG16(nn.Module):
    def __init__(self,vgg16,class_num,vec_d):
        super(RelationNetworkWithVGG16, self).__init__()

```



```

self.vgg16 = vgg16
for p in vgg16.parameters():
    p.requires_grad = False
self.up1 = nn.Upsample(size=(14,14))
self.conv1 = nn.Conv2d(in_channels=1536,out_channels=512,kernel_size=3,padding=1)
self.up2 = nn.Upsample(size=(28,28))
self.conv2_1 = nn.Conv2d(in_channels=512,out_channels=256,kernel_size=3,padding=1)
self.conv2_2 = nn.Conv2d(in_channels=768,out_channels=256,kernel_size=3,padding=1)
self.up3 = nn.Upsample(size=(56,56))
self.conv3_1 = nn.Conv2d(in_channels=256,out_channels=128,kernel_size=3,padding=1)
self.conv3_2 = nn.Conv2d(in_channels=384,out_channels=128,kernel_size=3,padding=1)
self.up4 = nn.Upsample(size=(112,112))
self.conv4_1 = nn.Conv2d(in_channels=128,out_channels=64,kernel_size=3,padding=1)
self.conv4_2 = nn.Conv2d(in_channels=192,out_channels=64,kernel_size=3,padding=1)
self.up5 = nn.Upsample(size=(224,224))
self.conv5_1 = nn.Conv2d(in_channels=64,out_channels=3,kernel_size=3,padding=1)
self.conv5_2 = nn.Conv2d(in_channels=9,out_channels=3,kernel_size=3,padding=1)
self.conv6 = nn.Conv2d(in_channels=3,out_channels=class_num,kernel_size=1)

self.fc1 = nn.Linear(vec_d,512)
self.fc2 = nn.Linear(512,512)
self.fc3 = nn.Linear(512,256)
self.fc4 = nn.Linear(256,128)
self.fc5 = nn.Linear(128,64)
self.fc6 = nn.Linear(64,3)

def forward(self,imgs,vecs):
    pre_x_112 = self.vgg16[0:5](imgs) # 64*112*112
    pre_x_56 = self.vgg16[5:10](pre_x_112) # 128*56*56
    pre_x_28 = self.vgg16[10:17](pre_x_56) # 256*28*28
    pre_x_14 = self.vgg16[17:24](pre_x_28) # 512*14*14
    pre_x_7 = self.vgg16[24:31](pre_x_14) # 512*7*7

    vecs_1 = F.relu(self.fc1(vecs),inplace=True) # 512
    vecs_2 = F.relu(self.fc2(vecs_1),inplace=True) # 512
    vecs_3 = F.relu(self.fc3(vecs_2),inplace=True) # 256
    vecs_4 = F.relu(self.fc4(vecs_3),inplace=True) # 128
    vecs_5 = F.relu(self.fc5(vecs_4),inplace=True) # 64
    vecs_6 = F.relu(self.fc6(vecs_5),inplace=True) # 3

    vecs_1 = vecs_1.unsqueeze(2).unsqueeze(3)
    vecs_2 = vecs_2.unsqueeze(2).unsqueeze(3)
    vecs_3 = vecs_3.unsqueeze(2).unsqueeze(3)
    vecs_4 = vecs_4.unsqueeze(2).unsqueeze(3)

```

```

vecs_5 = vecs_5.unsqueeze(2).unsqueeze(3)
vecs_6 = vecs_6.unsqueeze(2).unsqueeze(3)

Y = torch.cat((pre_x_7,vecs_1.repeat(1,1,7,7)),1) # 1024*7*7
y = self.up1(pre_x_7)
y = torch.cat((y,pre_x_14,vecs_2.repeat(1,1,14,14)),1) # 1536*14*14
y = F.relu(self.conv1(y),inplace=True) # 512*14*14

y = self.up2(y) # 512*28*28
y = F.relu(self.conv2_1(y),inplace=True) # 256*28*28
y = torch.cat((y,pre_x_28,vecs_3.repeat(1,1,28,28)),1) # 768*28*28
y = F.relu(self.conv2_2(y),inplace=True) # 256*28*28

y = self.up3(y) # 256*56*56
y = F.relu(self.conv3_1(y),inplace=True) # 128*56*56
y = torch.cat((y,pre_x_56,vecs_4.repeat(1,1,56,56)),1)
y = F.relu(self.conv3_2(y),inplace=True) # 128*56*56

y = self.up4(y) # 128*112*112
y = F.relu(self.conv4_1(y),inplace=True) # 64*112*112
y = torch.cat((y,pre_x_112,vecs_5.repeat(1,1,112,112)),1)
y = F.relu(self.conv4_2(y),inplace=True) # 64*112*112

y = self.up5(y) # 64*224*224
y = F.relu(self.conv5_1(y),inplace=True) # 3*224*224
y = torch.cat((y,imgs,vecs_6.repeat(1,1,224,224)),1)
y = F.relu(self.conv5_2(y),inplace=True) # 3*224*224

y = torch.sigmoid(self.conv6(y))
return y

```

```

class RelationNetworkWithVGG16NoVec(nn.Module):
    def __init__(self,vgg16,class_num):
        super(RelationNetworkWithVGG16NoVec, self).__init__()
        self.vgg16 = vgg16
        for p in vgg16.parameters():
            p.requires_grad = False
        self.up1 = nn.Upsample(size=(14,14))
        self.conv1 = nn.Conv2d(in_channels=1024,out_channels=512,kernel_size=3,padding=1)
        self.up2 = nn.Upsample(size=(28,28))
        self.conv2_1 = nn.Conv2d(in_channels=512,out_channels=256,kernel_size=3,padding=1)
        self.conv2_2 = nn.Conv2d(in_channels=512,out_channels=256,kernel_size=3,padding=1)
        self.up3 = nn.Upsample(size=(56,56))
        self.conv3_1 = nn.Conv2d(in_channels=256,out_channels=128,kernel_size=3,padding=1)

```

```

self.conv3_2 = nn.Conv2d(in_channels=256,out_channels=128,kernel_size=3,padding=1)
self.up4 = nn.Upsample(size=(112,112))
self.conv4_1 = nn.Conv2d(in_channels=128,out_channels=64,kernel_size=3,padding=1)
self.conv4_2 = nn.Conv2d(in_channels=128,out_channels=64,kernel_size=3,padding=1)
self.up5 = nn.Upsample(size=(224,224))
self.conv5_1 = nn.Conv2d(in_channels=64,out_channels=3,kernel_size=3,padding=1)
self.conv5_2 = nn.Conv2d(in_channels=6,out_channels=3,kernel_size=3,padding=1)
self.conv6 = nn.Conv2d(in_channels=3,out_channels=class_num,kernel_size=1)

```

```

def forward(self,imgs):
    pre_x_112 = self.vgg16[0:5](imgs) # 64*112*112
    pre_x_56 = self.vgg16[5:10](pre_x_112) # 128*56*56
    pre_x_28 = self.vgg16[10:17](pre_x_56) # 256*28*28
    pre_x_14 = self.vgg16[17:24](pre_x_28) # 512*14*14
    pre_x_7 = self.vgg16[24:31](pre_x_14) # 512*7*7

    y = self.up1(pre_x_7)
    y = torch.cat((y,pre_x_14),1) # 1024*14*14
    y = F.relu(self.conv1(y),inplace=True) # 512*14*14

    y = self.up2(y) # 512*28*28
    y = F.relu(self.conv2_1(y),inplace=True) # 256*28*28
    y = torch.cat((y,pre_x_28),1)
    y = F.relu(self.conv2_2(y),inplace=True) # 256*28*28

    y = self.up3(y) # 256*56*56
    y = F.relu(self.conv3_1(y),inplace=True) # 128*56*56
    y = torch.cat((y,pre_x_56),1)
    y = F.relu(self.conv3_2(y),inplace=True) # 128*56*56

    y = self.up4(y) # 128*112*112
    y = F.relu(self.conv4_1(y),inplace=True) # 64*112*112
    y = torch.cat((y,pre_x_112),1)
    y = F.relu(self.conv4_2(y),inplace=True) # 64*112*112

    y = self.up5(y) # 64*224*224
    y = F.relu(self.conv5_1(y),inplace=True) # 3*224*224
    y = torch.cat((y,imgs),1)
    y = F.relu(self.conv5_2(y),inplace=True) # 3*224*224

    y = torch.sigmoid(self.conv6(y))
    return y

```

```

class RelationNetworkWithVGG16AllConv(nn.Module):

```

```

def __init__(self,vgg16,class_num,vec_d):
    super(RelationNetworkWithVGG16, self).__init__()
    self.vgg16 = vgg16
    for p in vgg16.parameters():
        p.requires_grad = False
    self.up1 = nn.Upsample(size=(14,14))
    self.conv1 = nn.Conv2d(in_channels=1536,out_channels=512,kernel_size=3,padding=1)
    self.up2 = nn.Upsample(size=(28,28))
    self.conv2_1 = nn.Conv2d(in_channels=512,out_channels=256,kernel_size=3,padding=1)
    self.conv2_2 = nn.Conv2d(in_channels=768,out_channels=256,kernel_size=3,padding=1)
    self.up3 = nn.Upsample(size=(56,56))
    self.conv3_1 = nn.Conv2d(in_channels=256,out_channels=128,kernel_size=3,padding=1)
    self.conv3_2 = nn.Conv2d(in_channels=384,out_channels=128,kernel_size=3,padding=1)
    self.up4 = nn.Upsample(size=(112,112))
    self.conv4_1 = nn.Conv2d(in_channels=128,out_channels=64,kernel_size=3,padding=1)
    self.conv4_2 = nn.Conv2d(in_channels=192,out_channels=64,kernel_size=3,padding=1)
    self.up5 = nn.Upsample(size=(224,224))
    self.conv5_1 = nn.Conv2d(in_channels=64,out_channels=3,kernel_size=3,padding=1)
    self.conv5_2 = nn.Conv2d(in_channels=9,out_channels=3,kernel_size=3,padding=1)
    self.conv6 = nn.Conv2d(in_channels=3,out_channels=class_num,kernel_size=1)

    self.vec_up1 = nn.Upsample(size=(7,7))
    self.vec_conv1 =
nn.Conv2d(in_channels=vec_d,out_channels=512,kernel_size=3,padding=1)
    self.vec_up2 = nn.Upsample(size=(14,14))
    self.vec_conv2 =
nn.Conv2d(in_channels=512,out_channels=512,kernel_size=3,padding=1)
    self.vec_up3 = nn.Upsample(size=(28,28))
    self.vec_conv3 =
nn.Conv2d(in_channels=512,out_channels=256,kernel_size=3,padding=1)
    self.vec_up4 = nn.Upsample(size=(56,56))
    self.vec_conv4 =
nn.Conv2d(in_channels=256,out_channels=128,kernel_size=3,padding=1)
    self.vec_up5 = nn.Upsample(size=(112,112))
    self.vec_conv5 = nn.Conv2d(in_channels=128,out_channels=64,kernel_size=3,padding=1)
    self.vec_up6 = nn.Upsample(size=(224,224))
    self.vec_conv6 = nn.Conv2d(in_channels=64,out_channels=3,kernel_size=3,padding=1)

def forward(self,imgs,vecs):
    pre_x_112 = self.vgg16[0:5](imgs) # 64*112*112
    pre_x_56 = self.vgg16[5:10](pre_x_112) # 128*56*56
    pre_x_28 = self.vgg16[10:17](pre_x_56) # 256*28*28
    pre_x_14 = self.vgg16[17:24](pre_x_28) # 512*14*14

```

```

pre_x_7 = self.vgg16[24:31](pre_x_14) # 512*7*7

vecs = vecs.unsqueeze(2).unsqueeze(3)
vecs_1 = self.vec_up1(vecs)
vecs_1 = self.vec_conv1(vecs_1)
vecs_2 = self.vec_up2(vecs_1)
vecs_2 = self.vec_conv2(vecs_2)
vecs_3 = self.vec_up3(vecs_2)
vecs_3 = self.vec_conv3(vecs_3)
vecs_4 = self.vec_up4(vecs_3)
vecs_4 = self.vec_conv4(vecs_4)
vecs_5 = self.vec_up5(vecs_4)
vecs_5 = self.vec_conv5(vecs_5)
vecs_6 = self.vec_up6(vecs_5)
vecs_6 = self.vec_conv6(vecs_6)

Y = torch.cat((pre_x_7,vecs_1),1) # 1024*7*7
y = self.up1(pre_x_7)
y = torch.cat((y,pre_x_14,vecs_2),1) # 1536*14*14
y = F.relu(self.conv1(y),inplace=True) # 512*14*14

y = self.up2(y) # 512*28*28
y = F.relu(self.conv2_1(y),inplace=True) # 256*28*28
y = torch.cat((y,pre_x_28,vecs_3),1) # 768*28*28
y = F.relu(self.conv2_2(y),inplace=True) # 256*28*28

y = self.up3(y) # 256*56*56
y = F.relu(self.conv3_1(y),inplace=True) # 128*56*56
y = torch.cat((y,pre_x_56,vecs_4),1)
y = F.relu(self.conv3_2(y),inplace=True) # 128*56*56

y = self.up4(y) # 128*112*112
y = F.relu(self.conv4_1(y),inplace=True) # 64*112*112
y = torch.cat((y,pre_x_112,vecs_5),1)
y = F.relu(self.conv4_2(y),inplace=True) # 64*112*112

y = self.up5(y) # 64*224*224
y = F.relu(self.conv5_1(y),inplace=True) # 3*224*224
y = torch.cat((y,imgs,vecs_6),1)
y = F.relu(self.conv5_2(y),inplace=True) # 3*224*224

y = torch.sigmoid(self.conv6(y))
return y

```

```

class Identity(nn.Module):
    def __init__(self):
        super(Identity, self).__init__()
    def forward(self, x):
        return x

class RelationNetworkWithDeepLab(nn.Module):
    def __init__(self, deeplab, class_num, vec_d):
        super(RelationNetworkWithDeepLab, self).__init__()
        self.deeplab = deeplab
        for p in deeplab.backbone.parameters():
            p.requires_grad = False
        self.conv1 = nn.Conv2d(in_channels=512, out_channels=64, kernel_size=3, padding=1)
        self.conv2 =
nn.Conv2d(in_channels=128, out_channels=class_num, kernel_size=3, padding=1)

        self.fc1 = nn.Linear(vec_d, 256)
        self.fc2 = nn.Linear(256, 64)

    def forward(self, x, vecs):
        vecs_1 = F.relu(self.fc1(vecs), inplace=True)
        vecs_2 = F.relu(self.fc2(vecs_1), inplace=True)

        vecs_1 = vecs_1.unsqueeze(2).unsqueeze(3)
        vecs_2 = vecs_2.unsqueeze(2).unsqueeze(3)

        y = self.deeplab(x)['out']
        y = torch.cat((y, vecs_1.repeat(1, 1, 224, 224)), 1)
        y = F.relu(self.conv1(y))
        y = torch.cat((y, vecs_2.repeat(1, 1, 224, 224)), 1)
        y = self.conv2(y)
        y = torch.sigmoid(y)
        return y

def initModel(vec_d, model=0, allConv=False):
    if model == 0:
        vgg16 = models.vgg16(pretrained=True)
        print("load pre-trained vgg16")
    if vec_d > 0:
        if not allConv:
            relation_network = RelationNetworkWithVGG16(vgg16.features, 1, vec_d)
        else:
            relation_network = RelationNetworkWithVGG16(vgg16.features, 1, vec_d)
    else:

```

```

    relation_network = RelationNetworkWithVGG16AllConv(vgg16.features,1)
    return relation_network
elif model == 1:
    # in not pre-trained deeplabv3_resnet101, its backbone is ImageNet pre-trained resnet101
    deeplab = torch.hub.load('pytorch/vision:v0.6.0', 'deeplabv3_resnet101', pretrained=False)
    deeplab.classifier[4] = Identity()
    print("load pre-trained deepLab")
    relation_network = RelationNetworkWithDeepLab(deeplab,1,vec_d)
    return relation_network

```

### C. Implementation codes of IoU computing

```

import numpy as np
import torch
from torch.utils.data import DataLoader, TensorDataset

def mIoU_of_class(prediction, predict_label, target, target_label):
    target_args = torch.where(target == target_label)
    target_size = target_args[0].shape[0]
    if target_size == 0:
        return None
    else:
        intersection = prediction[target_args] == predict_label
        intersection = torch.sum(intersection)
        predict_args = torch.where(prediction == predict_label)
        predict_size = predict_args[0].shape[0]
        union = target_size + predict_size - intersection
        mIoU = intersection.float()/union.float()
    return mIoU

# test_classes is search space starting from 0.
# 0 is background class
def IoU_per_class(model, test_features, test_labels, word_vectors, test_classes, test_batch, GPU,
calibrate_classes, calibrate):
    if (test_classes < 0).any():
        return False

    test_data = TensorDataset(test_features, test_labels)
    test_loader = DataLoader(test_data, batch_size=test_batch, shuffle=False)
    test_size = test_features.shape[0]

    test_classes = np.sort(test_classes)

```

```

if test_classes[0] == 0:
    includeBack = True
else:
    includeBack = False
class_num = len(test_classes)
test_vectors = torch.tensor([word_vectors[int(c-1)] for c in test_classes if c > 0]).view(-1,
word_vectors.shape[1]).float().cuda(GPU) # -1*300

class_acc = [None] * class_num
predict_total = None

for batch_features, batch_labels in test_loader:
    batch_size = batch_features.shape[0]
    support_features = test_vectors.repeat(batch_size,1) # -1*300*1*1 -> -1*256*28*28

    query_features = batch_features.repeat(1,test_vectors.shape[0],1,1).view(-1,3,224,224)
    query_features = query_features.cuda(GPU).float()

    relations =
model(query_features,support_features).view(batch_size,test_vectors.shape[0],224,224)
    if includeBack:
        background_scores = 1-torch.max(relations,1)[0].view(-1,1,224,224)
        scores = torch.cat((background_scores,relations),1)
    else:
        scores = relations

    if calibrate_classes is not None and len(calibrate_classes) > 0:
        scores[:,calibrate_classes,:,:] = scores[:,calibrate_classes,:,:] * calibrate

    prediction = torch.max(scores,1)[1]

    if predict_total is None:
        predict_total = prediction.cpu().detach()
    else:
        predict_total = torch.cat((predict_total,prediction.cpu().detach()))

# ignore unselected classes
test_labels = test_labels.view(-1,224,224)
select_args = np.where(np.isin(test_labels,test_classes))
test_labels = test_labels[select_args]
predict_total = predict_total[select_args]

for c_i in range(class_num):
    mIoU = mIoU_of_class(predict_total, c_i, test_labels, test_classes[c_i])

```



```

if mIoU is not None:
    class_acc[c_i] = mIoU.item()
return class_acc

```

## D. Implementation codes of preprocessing PASCAL

```

import scipy.io as sio
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import os
from torchvision import transforms
import torchvision.datasets.folder as torch_folder
from torch.utils.data import DataLoader, TensorDataset
from torch.optim.lr_scheduler import StepLR
import matplotlib.pyplot as plt
import torchvision.models as models
from gensim.models.keyedvectors import KeyedVectors
from gensim.models import word2vec
from torch.utils.checkpoint import checkpoint
drive_path = "./"

# step 0: read train.txt and val.txt
train_txt = drive_path + "data/VOC2012/train.txt"
train_loc = []
f = open(train_txt, 'r')
for line in f.readlines():
    train_loc.append(line.strip())
train_loc = np.array(train_loc)
f.close()

val_txt = drive_path + "data/VOC2012/val.txt"
test_loc = []
f = open(val_txt, 'r')
for line in f.readlines():
    test_loc.append(line.strip())
test_loc = np.array(test_loc)
f.close()
sio.savemat(drive_path + "data/VOC2012/split.mat", {'train_loc': train_loc, 'test_loc': test_loc})

# Step 1: preprocess images
def preprocess_batch_img(batch_path):

```

```

batch_num = len(batch_path)
input_batch = torch.zeros([batch_num,3,224,224], dtype=torch.float64)
preprocess = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
for i in range(batch_num):
    path = batch_path[i].strip()
    print(path)
    input_image = torch_folder.pil_loader(path)
    input_tensor = preprocess(input_image)
    input_batch[i] = input_tensor
return input_batch

```

```

image_root = drive_path + "data/VOC2012/JPEGImages/"

```

```

train_img_path = [image_root + i + ".jpg" for i in train_loc]
train_features = preprocess_batch_img(train_img_path)
train_features = train_features.numpy()
print(train_features.shape)
sio.savemat(drive_path +
"data/VOC2012/images_224_224_RGB_train.mat",{ 'features':train_features })

```

```

test_img_path = [image_root + i + ".jpg" for i in test_loc]
test_features = preprocess_batch_img(test_img_path)
test_features = test_features.numpy()
print(test_features.shape)
sio.savemat(drive_path +
"data/VOC2012/images_224_224_RGB_test.mat",{ 'features':test_features })

```

```

# Step 2: prepare seen classes, unseen classes and their vectors

```

```

# PASCAL VOC 2012 has 20 classes:

```

```

class_names =

```

```

["Aeroplane","Bicycle","Bird","Boat","Bottle","Bus","Car","Cat","Chair","Cow","Dining_table"
,"Dog","Horse","Motorbike","Person","Potted_plant","Sheep","Sofa","Train","Tv_monitor"]

```

```

seen_c = range(15)

```

```

unseen_c = range(15,20)

```

```

word2vec_path = drive_path + "data/word2vec/GoogleNews-vectors-negative300.bin"

```

```

model = KeyedVectors.load_word2vec_format(word2vec_path, binary=True)

```

```

word2vectors = []

```

```

for c_name in class_names:

```

```

words = c_name.strip().split('_')
vec = np.array([0] * 300, dtype='float64')
for w in words:
    if w in model.vocab:
        vec += model[w]
word2vectors.append(vec)
word2vectors = np.array(word2vectors)
print(word2vectors.shape)

# pre-trained fastText model with common crawl
import io
fastText_path = drive_path + "data/fastText/crawl-300d-2M.vec"
fin = io.open(fastText_path, 'r', encoding='utf-8', newline='\n', errors='ignore')
n, d = map(int, fin.readline().split())
lines = fin.readlines()
crawl_voc_list = {} # read word index first to avoid running out of RAM
for i in range(n):
    word = lines[i].split()[0]
    crawl_voc_list[word]=i

ftvectors = []
for c_name in class_names:
    words = c_name.strip().split('_')
    vec = np.array([0] * 300, dtype='float64')
    for w in words:
        if w in crawl_voc_list:
            index = crawl_voc_list[w]
            tokens = [float(num) for num in lines[index].split()[1:]]
            vec += tokens
    ftvectors.append(vec)
ftvectors = np.array(ftvectors)
print(ftvectors.shape)

sio.savemat(drive_path +
"data/VOC2012/matfiles/classes_info.mat", {'class_names':class_names,'seen_c':seen_c,'unseen_
c':unseen_c,'word2vectors':word2vectors,'ftvectors':ftvectors })

# Step 3: read and resize labels and save to mat
colors of each class above. [0,0,0] is background.
class_color =
[[0,0,0],[128,0,0],[0,128,0],[128,128,0],[0,0,128],[128,0,128],[0,128,128],[128,128,128],[64,0,0]
,[192,0,0],[64,128,0],[192,128,0],[64,0,128],[192,0,128],[64,128,128],[192,128,128],[0,64,0],
[128,64,0],[0,192,0],[128,192,0],[0,64,128]]

```

```

def preprocessLabel(image_index):
    print(image_index)
    import cv2
    label = cv2.imread(drive_path + "data/VOC2012/SegmentationClass/"+image_index+".png")
    label = cv2.resize(label, (224,224), interpolation=cv2.INTER_NEAREST)
    label = np.array(label)
    # background: 0; classes: 1-20; other pixels(segmentation borders): -1
    l_converted = np.zeros((224,224),dtype=int) -1
    for i in range(len(class_color)):
        color = class_color[i]
        args = np.where((label == [color[2],color[1],color[0]]).all(-1)) # opencv read in BGR not RGB
        l_converted[args] = i
    return l_converted

train_labels = np.empty((train_loc.shape[0],1,224,224), dtype=int)
for i in range(train_loc.shape[0]):
    index = train_loc[i]
    train_labels[i,0,:,:) = preprocessLabel(image_index=index)
print(train_labels.shape)

test_labels = np.empty((test_loc.shape[0],1,224,224), dtype=int)
for i in range(test_loc.shape[0]):
    index = test_loc[i]
    test_labels[i,0,:,:) = preprocessLabel(image_index=index)
print(test_labels.shape)

sio.savemat(drive_path +
"data/VOC2012/seg_class_labels.mat",{ 'train_labels':train_labels,'test_labels':test_labels })

background: 0; classes: 1-20; other pixels(segmentation borders): -1
train_labels = sio.loadmat(drive_path + "data/VOC2012/seg_class_labels.mat")['train_labels']
test_labels = sio.loadmat(drive_path + "data/VOC2012/seg_class_labels.mat")['test_labels']
print(train_labels.shape)
print(test_labels.shape)

```

## E. Implementation codes of training PASCAL

```

import scipy.io as sio
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import os

```

```

from torchvision import transforms
import torchvision.datasets.folder as torch_folder
from torch.utils.data import DataLoader, TensorDataset
from torch.optim.lr_scheduler import StepLR
from gensim.models.keyedvectors import KeyedVectors
from gensim.models import word2vec
import argparse
import models
from iou import IoU_per_class

BATCH_SIZE = args.batch_size
TEST_BATCH_SIZE = args.test_batch_size
EPISODE = args.episode
LEARNING_RATE = args.learning_rate
LOAD_MODEL = args.load_model
relation_model_path = args.root_path + "models/" + args.relation_model_file_name + ".pkl"
best_zsl_model_path = args.root_path + "models/" + args.relation_model_file_name +
"_best_zsl.pkl"
best_gzsl_model_path = args.root_path + "models/" + args.relation_model_file_name +
"_best_gzsl.pkl"
VEC = args.vec
IMG_MODEL = args.img_model
gpu_list = list(map(int, args.gpu_list.split(",")))
GPU = gpu_list[0]
bce_w = args.bce_weights

# step 1: read train/test split
split = sio.loadmat(args.root_path + "data/VOC2012/split.mat")
train_loc = split['train_loc']
print(train_loc.shape)
test_loc = split['test_loc']
print(test_loc.shape)

# Step 2: resize images
train_features = sio.loadmat(args.root_path +
"data/VOC2012/images_224_224_RGB_train.mat")['features']
print(train_features.shape)
test_features = sio.loadmat(args.root_path +
"data/VOC2012/images_224_224_RGB_test.mat")['features']
print(test_features.shape)

# Step 3: prepare seen classes, unseen classes and their vectors
class_info = sio.loadmat(args.root_path + "data/VOC2012/matfiles/classes_info.mat")
class_name = class_info['class_name']

```

```

seen_c = class_info['seen_c'][0]
unseen_c = class_info['unseen_c'][0]
if VEC == 0: # word2vec
    word_vectors = class_info['word2vectors']
    vec_d = 300
    print(word_vectors.shape)
    print("load word2vec")
elif VEC == 1: # fastText
    word_vectors = class_info['ftvectors']
    vec_d = 300
    print(word_vectors.shape)
    print("load fastText")
elif VEC == 2: # word2vec::fastText
    word2vectors = class_info['word2vectors']
    ftvectors = class_info['ftvectors']
    word_vectors = np.concatenate((word2vectors,ftvectors),1)
    vec_d = 600
    print(word_vectors.shape)
    print("load np.cat(word2vec, fastText)")

# Step 4: read and resize labels
train_labels = sio.loadmat(args.root_path +
"data/VOC2012/matfiles/seg_class_labels.mat")['train_labels']
test_labels = sio.loadmat(args.root_path +
"data/VOC2012/matfiles/seg_class_labels.mat")['test_labels']
print(train_labels.shape)
print(test_labels.shape)

# Step 5: read image's classes from annotation files (skipped)
# Step 6: prepare dataset
train_features = torch.from_numpy(train_features)
train_labels = torch.from_numpy(train_labels)
train_data = TensorDataset(train_features, train_labels)
test_features = torch.from_numpy(test_features)
test_labels = torch.from_numpy(test_labels)

# Step 7: define and init models
relation_network = models.initModel(vec_d, IMG_MODEL)
relation_network = torch.nn.DataParallel(relation_network, device_ids=gpu_list)
relation_network.cuda(GPU)

relation_network_optim =
torch.optim.Adam(relation_network.parameters(),lr=LEARNING_RATE)
relation_network_scheduler = StepLR(relation_network_optim,step_size=10000,gamma=0.5)

```

```

if LOAD_MODEL:
    print("load model: ",LOAD_MODEL)
    relation_network.load_state_dict(torch.load(relation_model_path))

last_accuracy = args.last_accuracy
last_H = args.last_H

# Step 8: episode training
for episode in range(EPIISODE):
    relation_network.train()
    relation_network_scheduler.step(episode)

    train_loader = DataLoader(train_data,batch_size=BATCH_SIZE,shuffle=True)
    batch_features, batch_labels = train_loader.__iter__().next()

    # episode_classes = np.unique(batch_labels)
    episode_classes = np.unique(np.random.randint(20, size=10)+1)
    filtered = np.isin(episode_classes, seen_c+1)
    filtered = np.where(filtered)
    sample_features = torch.tensor([word_vectors[int(c-1)] for c in
    episode_classes[filtered]]).view(-1,vec_d).float().cuda(GPU) # -1*300
    class_num = sample_features.shape[0]
    sample_features = sample_features.repeat(BATCH_SIZE,1)

    batch_features = batch_features.cuda(GPU).float() # -1*3*224*224
    batch_features = batch_features.repeat(1,class_num,1,1).view(-1,3,224,224)

    relations =
    relation_network(batch_features,sample_features).view(BATCH_SIZE,class_num,224,224)

    for c in range(1,21): # ignore other classes
        if c not in episode_classes[filtered]:
            args = np.where(batch_labels == c)
            batch_labels[args] = -1

    args = torch.where(batch_labels != -1) # only seen labels left
    one_hot_index = batch_labels[args]

    for c_i in range(class_num): # re-order labels
        c = episode_classes[filtered][c_i]
        c_args = np.where(one_hot_index == c)
        one_hot_index[c_args] = c_i+1

```

```

one_hot_labels =
torch.zeros(one_hot_index.shape[0],class_num+1).scatter_(1,one_hot_index.view(-1,1).long(),1)
one_hot_weights =
torch.ones(one_hot_index.shape[0],class_num+1).scatter_(1,one_hot_index.view(-
1,1).long(),bce_w)
one_hot_labels = one_hot_labels[:,1:]
one_hot_weights = one_hot_weights[:,1:]
relations = relations[args[0],:,args[2],args[3]]

bce = nn.BCELoss(weight=one_hot_weights).cuda(GPU)
loss = bce(relations, one_hot_labels.cuda(GPU))

loss.backward()
# update
relation_network_optim.step()
relation_network.zero_grad()

if (episode+1)%100 == 0:
    torch.save(relation_network.state_dict(),relation_model_path)
    print("episode:",episode+1,"loss",loss.item()," models saved!")
if (episode+1)%1000 == 0:
    relation_network.eval()
    zsl_label_space = np.array(range(16,21))
    zsl_acc_per_class = IoU_per_class(relation_network, test_features, test_labels, word_vectors,
zsl_label_space, TEST_BATCH_SIZE, GPU, None, None)
    print(zsl_acc_per_class)
    zsl_acc_per_class = [acc for acc in zsl_acc_per_class if acc is not None]
    zsl_mIoU = sum(zsl_acc_per_class) / len(zsl_acc_per_class)
    print('zsl = %.4f' % (zsl_mIoU))

if zsl_mIoU > last_accuracy:
    last_accuracy = zsl_mIoU
    torch.save(relation_network.state_dict(), best_zsl_model_path)
    print("Last zsl accuracy updated! Best zsl model saved!")
else:
    print("last zsl accuracy is ", last_accuracy)

gzsl_label_space = np.array(range(1,21))
gzsl_acc_per_class = IoU_per_class(relation_network, test_features, test_labels,
word_vectors, gzsl_label_space, TEST_BATCH_SIZE, GPU, None, None)
print(gzsl_acc_per_class)
gzsl_acc_unseen = [gzsl_acc_per_class[c] for c in unseen_c]
gzsl_acc_unseen = [acc for acc in gzsl_acc_unseen if acc is not None]
unseen_mIoU = sum(gzsl_acc_unseen) / len(gzsl_acc_unseen)

```



```
gzsl_acc_seen = [gzsl_acc_per_class[c] for c in seen_c]
gzsl_acc_seen = [acc for acc in gzsl_acc_seen if acc is not None]
seen_mIoU = sum(gzsl_acc_seen) / len(gzsl_acc_seen)
H = (2 * seen_mIoU * unseen_mIoU) / (seen_mIoU + unseen_mIoU)
print('gzsl: seen=% .4f, unseen=% .4f, h=% .4f' % (seen_mIoU, unseen_mIoU, H))
```