Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Object Classification and Counting with Count-ception Network

Relatore: Prof. Giacomo Boracchi
Correlatore: Diego Carrera

Tesi di laurea di: Zhou Yinan
Matr. 872686

Anno Accademico 2017-2018

Acknowledgements

To start, I would like to thank my supervisor, prof. Giacomo Boracchi, for his guidance and effort through the whole thesis work. Also I would like to thank my co-supervisor Diego Carrera for his support and help during this project.

I would like to thank them for introducing me into the field of computer vision and deep learning. I appreciate a lot for the GPU supporting.

Abstract

Steller sea lions in the western Aleutian Islands have declined 94 percent in the last 30 years. The endangered population are focus of conservation efforts which require annual population counts. Currently, it takes biologists up to four months to count sea lions from the thousand of images NOAA Fisheries collects each year. The results of these counts are time-sensitive and automating the annual population count will free up critical resources, and allow experts to focus on core research.

Computer vision with deep learning methods is a hot topic in both research and industry. Various algorithms are invented and improved for image classification and object detection. However there are not so many algorithms optimized for object counting which is a common demand in many situations, like endangered animal protection and biological imaging analysis. The purpose of this thesis is to design an algorithm optimized for simultaneously object classification and counting which could automate preliminary information gathering for various researches.

Sommario

I leoni marini di Steller nelle isole Aleutine occidentali sono diminuiti del 94 percento negli ultimi 30 anni. La popolazione in via di estinzione è al centro degli sforzi di conservazione che richiedono un numero annuale di popolazione. Al momento, i biologi impiegano fino a quattro mesi per contare i leoni marini dalle migliaia di immagini raccolte da NOAA Fisheries ogni anno. I risultati di questi conteggi sono sensibili al fattore tempo e l'automazione del conteggio annuale della popolazione libererà risorse critiche e consentirà agli esperti di concentrarsi sulla ricerca di base.

La visione artificiale con metodi di apprendimento profondi è un tema caldo sia nella ricerca che nell'industria. Vari algoritmi sono inventati e migliorati per la classificazione delle immagini e il rilevamento degli oggetti. Tuttavia non ci sono molti algoritmi ottimizzati per il conteggio degli oggetti, che è una domanda comune in molte situazioni, come la protezione degli animali in via di estinzione e l'analisi delle immagini mediche. Lo scopo di questa tesi è di progettare un algoritmo ottimizzato per la classificazione e il conteggio di oggetti simultanei che potrebbe automatizzare la raccolta di informazioni preliminari per varie ricerche.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Steller sea lions in the western Aleutian Islands have declined 94 percent in the last 30 years. The endangered western population, found in the North Pacific, are focus of conservation efforts which require annual population counts. Specifically trained scientists at NOAA Fisheries Alaska Fisheries Science Center conduct these surveys using airplanes and unoccupied aircraft systems to collect aerial images. Having accurate population estimates enables scientists to better understand factors that may be contributing to lack of recovery of Stellers in this area. Currently, it takes biologists up to four months to count sea lions from the thousand of images NOAA Fisheries collects each year. Once individual counts are conducted, the tallies must be reconciled to confirm their reliability. The results of these counts are time-sensitive. Automating the annual population count will free up critical resources, and allow experts to focus on core research. Plus, advancements in computer vision applied to aerial population counts may also greatly benefit other endangered species.

Since Alex re-introduced CNN architecture into computer vision field, deep learning methods become the state-of-art for image classification, localization, segmentation, as well as many other vision recognization tasks. Image classification is the task of giving an input image, outputting a corresponding label. The algorithm is usually trained with very large labeled dataset and the outputs are constrained to the labels we have in the training samples. Image localization and segmentation are more difficult than classification. Localization algorithm outputs a bounding box around each object and a corresponding label. In order to do this, the training data needs to have both the labels and the bounding boxes indicating the object location. This kind of dataset occupies large amount of time from human labelers and thus it is more difficult to construct. Segmentation algorithm requires pixel-leveled prediction for each object and this kind of training is even more time consuming.

The purpose for this thesis is to develop an algorithm for simultaneous classification and counting. Given an input image, we output the label and the corresponding count for each object. There are various kinds of situations where we could apply this algorithm to, for example, the counting problem for endangered animals like the sea lions described above. By using our algorithm, generating sea lion counting numbers in each image can be fully automated. Also in biological imaging, we could use this

Figure 1: Sea lion image

algorithm to produce precise counts for cells and tissues, helping doctors to analyze the disease. Moreover, the training dataset is a lot easier to construct: we only need to have a labeled dot for each object, no bounding boxes are required. This kind of dataset is more common than bounding box labeled or pixel-level labeled dataset.

The main challenge for this thesis is the inherent difficulty of our sea lion images which have small and similar objects but complex background environment. Moreover, each image has quite high resolution which makes it difficult for managing algorithm speed and memory consumption. As we can see in Figure 1, dense sea lion distribution on the rocks makes it difficult to classify sea lion type and generate precise counts.

In this thesis, we developed two major solutions for multi-object counting problem:

1. Reformulate the problem of object counting as object classification and use CNN to solve it.

2. Build a regression network optimized for object counting.

Our algorithm focuses on the second solution which uses Count-ception architecture [1] and fully convolutional mechanism [3] to generate counting numbers directly. The first solution is used as a baseline. We show that our algorithm which is optimized for object counting achieves superior performance than using classification to generate object counts. We use the sea lions dataset provided by NOAA Kaggle Competition to run our experiments.

The thesis is structured as:

- In Chapter 1, we review deep learning algorithms in computer vision field, including image classification, object detection and localization algorithms.

- In Chapter 2, we introduce Count-ception architecture and explain why this can solve object counting problems.

- In Chapter 3, we present our major contribution which is a modified version of Count-ception architecture, able to perform object classification and counting at the same time.
- In Chapter 4, we deal with dataset construction and preprocessing techniques.
- In Chapter 5, algorithm performance is analyzed.
- Finally, conclusions and further improvements are provided.[1]

[1]The code developed for this thesis can be found in https://github.com/marioZYN/Thesis

Chapter 1

Deep Learning Methods In Computer Vision

Image classification, object detection and localization are one of the major challenges in computer vision. In this chapter we briefly introduce these problems and the state-of-art algorithms. Then we analyze how we may use these algorithms to solve our sea lion counting problem.

1.1 Image classification

Image classification is the task of giving an input image and outputting the corresponding label. Before convolution neural network in image classification, people use handcrafted features from images and exploit these features for classifying images. It is a challenge, even for experts to design a feature extraction algorithm suitable for various vision recognization tasks. Convolutional neural network(CNN) is a special kind of deep learning architecture used in computer vision, which is composed of convolution and pooling layers. The convolution layer makes use of a set of learnable filters. Each filter learns how to extract features and patterns present in the image. The filter is convolved across the width and height of the input image, and a dot product operation is computed to give an activation map. Different filters which detect different features are convolved with the input image and the activation maps are stacked together to form the input for the next layer. By stacking more activation maps, we can get more abstract features. However, as the architecture becomes deeper, we may consume too much memory and in order to solve this problem, pooling layers are used to reduce the dimension of the activation maps. There are two types of pooling layers: max pooling and average pooling. As the name states, max pooling keeps the maximum value within the filter and discards all the rest, while average pooling keeps the average value. By discarding some values in each filter, we reduce the dimension of the activation maps and thus reduce the number

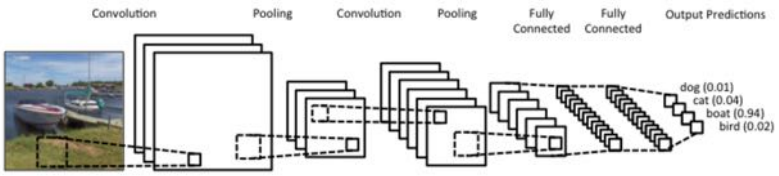


Figure 1.1: Convolutional neural network

of parameters we need to learn and this makes deep CNN architecture possible.

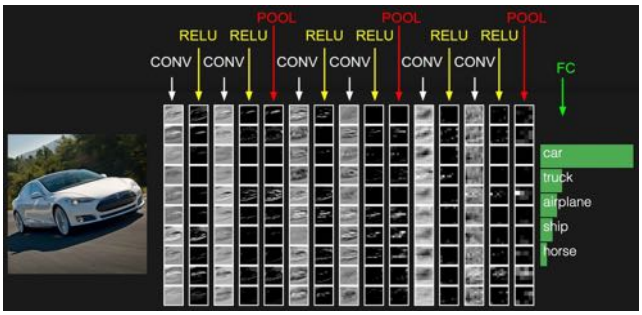


Figure 1.2: CNN architecture for image classification

After each convolution operation, an activation function is added to decide whether a certain neuron fires or not. There are different kinds of activation functions having different characteristics as illustrated in Figure 1.3. Sigmoid functions squashes the output into a value between zero and one and it was the most popular activation function back in days since it has nice interpretation as a saturating "firing rate" of a neuron. However sigmoid activation function has three major problems:

1. Saturated neurons "kill" the gradients.
2. Sigmoid outputs are not zero-centered which hurts gradient descent process.
3. Exponential function is a bit compute expensive.

ReLU(Rectified Linear Unit) activation function is used to avoid the drawbacks of sigmoid functions. ReLU activation function does not have saturation problem and while the largest gradient value for sigmoid function is $1/4$, the gradient for ReLU function is either 1 or 0. Theoretically ReLU activation function has larger convergence rate than sigmoid function. The problem for ReLU function is that when we have a negative input value, the gradient is zero. It seems to behave like our neurons which can fire or not, but in reality this can create dead ReLU nodes. Since the value and gradient are all zero when the input value is negative, it can happen that some neurons can never fire again. This is called the "dead neuron phenomenon". In order to solve this problem, leaky ReLU is used. Leaky ReLU does not have

zero gradient at the negative part of the axis, but a small positive value, thus when necessary the output value can grow back to non zero, avoiding dead neuron problem. Nowadays leaky ReLU is the most commonly used activation function in deep learning architectures.

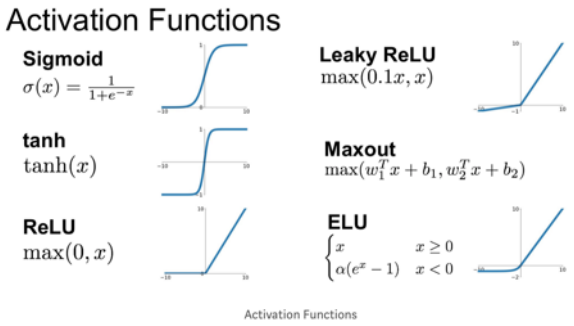


Figure 1.3: Activation functions

The general idea for CNN architecture is to stack several convolution and pooling layers to extract features from images, then it uses fully connected layers to exploit these features for classification. By using CNN architecture we don't need to design feature extraction algorithm, instead we can exploit gradient descent, letting convolution filters learn the weights from our training dataset. So convolution and pooling layers in CNN actually build up an automatic way for extracting features.

We want CNN architecture to generate classification results, and one way to do this is to output probability scores for each class. Suppose we have to classify each image among N possible classes, we can make our CNN architecture generate N values, each value representing the probability of being a certain class. Since this is a probability distribution, the summation of these N values is one. In CNN architecture, softmax layer is inserted at last in order to squeeze the output between zero and one. Softmax function amplifies probability of largest x_i but still assigns some probability to smaller x_i. It is defined in Formula 1.1.

$$P(y_i|x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \tag{1.1}$$

1.2 Object detection and localization

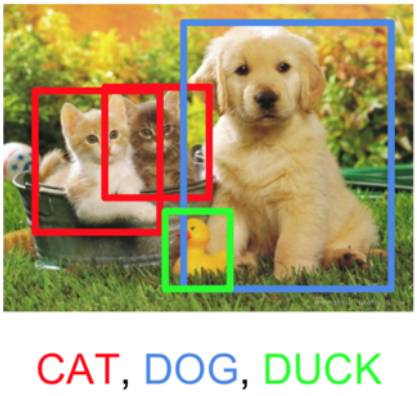


Figure 1.4: Object detection and localization

Object detection and localization is a more difficult task than image classification, because you need to first find possible object locations and then perform object classification. Given an input image possibly containing multiple objects, we need to generate a bounding box around each object and classify the object type, as illustrated in Figure 1.4. The general idea is to output the class label as well as the coordinates of the four corners of the bounding box. Outputting the class label is a classification problem and generating bounding box coordinates can be seen as a regression problem. In fact, each bounding box can be represented as a four value tuple: (x, y, w, h) which stands for coordinates of the center point, width and height of the bounding box. We combine the classification loss and regression loss as the final loss for our architecture.

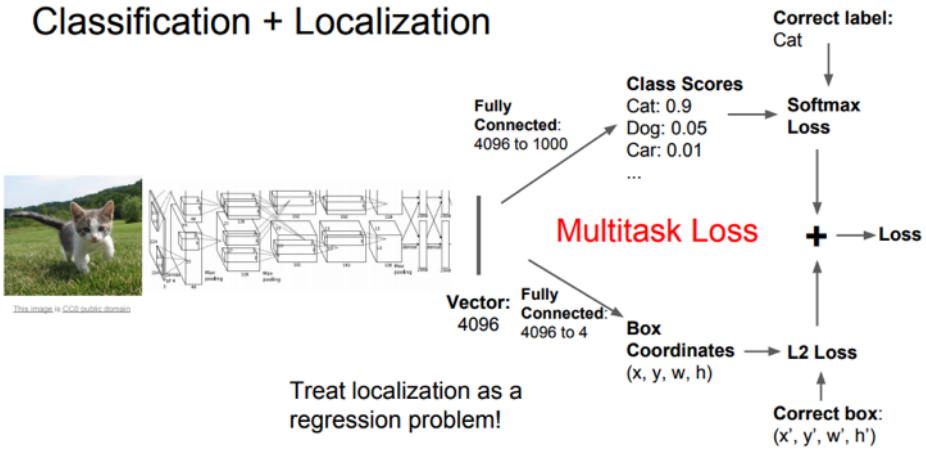


Figure 1.5: Classification and localization

Object detection and localization is a two-step problem, first we need to find which parts of the image may contain an object, second we need to classify each

part. We call the region in the image where may exist an object "region of interests"(ROI). One straight forward way to find ROI is to use a sliding window to generate all possible ROI. Since we don't know neither the location nor the size of each object, we need to test many positions and scales which is time consuming and not feasible. There exists various kinds of object detection and localization algorithms differing in network structure and ROI proposing techniques. Just like image classification algorithms, the performance of detection and localization boosted since deep learning architectures are used in this field, as we can see it in Figure 1.6. In the following sections, we will introduce some sate-of-art object detection and localization algorithms and by studying them, we may get inspired of how to solve multi-object counting problem.

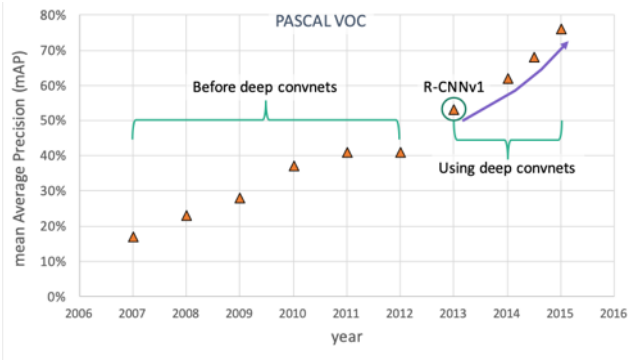


Figure 1.6: Object detection renaissance

1.2.1 R-CNN

R-CNN is short for Regional Convolutional Neural Network. The purpose of this algorithm is to generate a bounding box around each possible object and a corresponding label. Given an input image, we first generate ROI from the image and then use CNN to classify each region.

The pipeline for this algorithm is the following:

1. Build a CNN model and train it from scratch or download a pre-trained image classification model.

2. Fine-tune model for detection. Discard the final fully connected layer and modify it according to domain dependent problems.

3. Extract region proposals for all images by using external algorithms like selective search, and for each region, wrap it to CNN size and use CNN to classify its type. It can be a certain object or background.

4. Train one binary SVM per class to classify region features in order to check if object exists in this region.

5. For each class, train a linear regression model to map from extracted features of CNN to offsets of ground truth boxes in order to make up for the wrong positions of ROI.

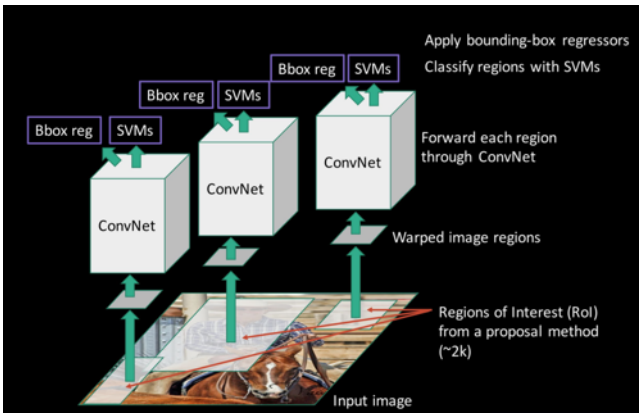


Figure 1.7: R-CNN architecture

In R-CNN algorithm, the region proposal method is independent from the algorithm itself, and we can use whatever region proposal algorithm we like. In the paper, the author suggests to use "selective search". Selective search is used to separate image into different sized areas which may contain objects. The general idea is to first use edge detection methods for creating fine grilled chunks and then greedily merge similar ones to create ROI. It is a greedy algorithm, starting from bottom-up segmentation and merging regions at different scales. This method can find "blob-like" regions containing similar pixel values. For each image, there are around $2k$ regions of proposals.

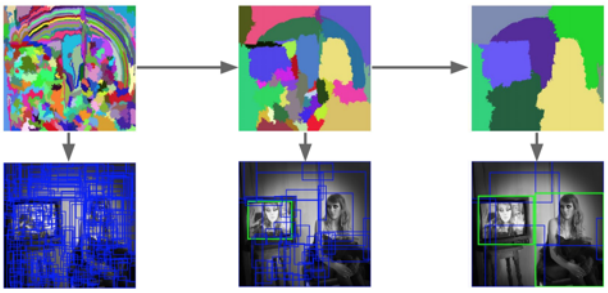


Figure 1.8: Selective search

R-CNN was the start-of-art algorithm for object detection since 2010. The ad hoc training procedures for R-CNN is the following:

- Fine-tune network with softmax classifier
- Train post-hoc linear SVMs

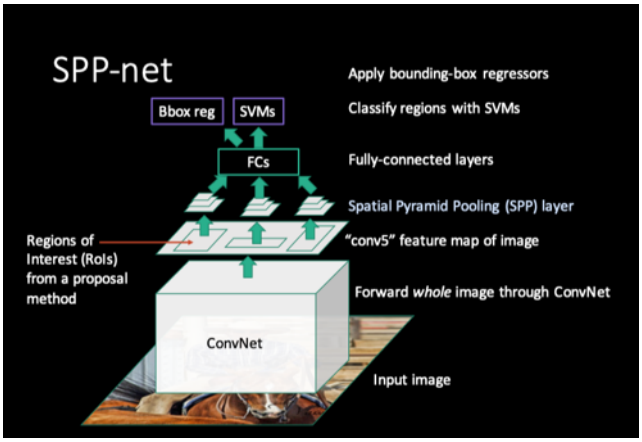


Figure 1.9: SPP-net

- Train post-hoc bounding-box regressors

There are three major disadvantages for this algorithm, first of all it is slow at test time because it needs to run both the selective search and full forward path for each region proposal. With VGG16 as the base architecture, it takes around 47s per image of size 224 x 224. Second, we spend large amount of time training SVMs and regressors, but they can only be used for this specific problem. For different problems we need to train SVMs and regressors again from scratch. Third, the whole architecture is not end-to-end and this complex multistage training pipeline is difficult to implement.

1.2.2 SPP-net

SPP-net solves the problem of slow inference time in R-CNN architecture by sharing computation. Recall that in R-CNN, we first generate ROI and then classify each ROI using CNN. Since the original image may be large, we could generate a lot of ROI regions and each of them needs to be forwarded through a CNN, which is time consuming. Now SPP-net swaps the order by first forwarding the original image through a CNN architecture, and then generate ROI from the feature map we get out of convolution. Besides sharing computation, SPP-net also invents a new pooling operation called "Spatial Pyramid Pooling".

In CNN architecture, we have convolution and pooling layers to extract features from images, and then we use fully connected layers to exploit these features for classification. However there is a technical issue in the training and testing of the CNNs: most of CNNs require a fixed input size (e.g. 224x224), which limits both the aspect ratio and the scale of the input image. When applied to images of arbitrary sizes, current CNN methods mostly fit the input image to the fixed size, either via cropping or via warping. However both methods have drawbacks: cropping can destroy whole image structure and wrapping can cause distortion.

Figure 1.10: Crop and warp

So why do we need to have fixed sized input? In fact, convolution and pooling operations do not require fixed size input, only fully connected layers do. When we forward various sized images through convolution and pooling layers, what we get are only tensors with different shapes. On the other hand, fully connected layers require fixed sized input by their definition. SPP-net creates spatial pyramid pooling layers to deal with this issue. By adding SPP layer right before fully connected layer, we can create fixed sized feature vector from tensors in different shapes. SPP-layer is actually a sequence of max pooling operations combined together as described in Figure 1.11. Each max pooling in SPP divides the input tensor evenly to a predefined number of areas. For example, the blue max pooling area always divides the whole tensor into 16 parts, no matter what shape the input tensor has. Each part will be max pooled to generate a single output value. Also note that 256-d is the number of channels of the input tensor, thus the final feature vector's length is calculated as: $(16 + 4 + 1) \times 256 = 5376$.

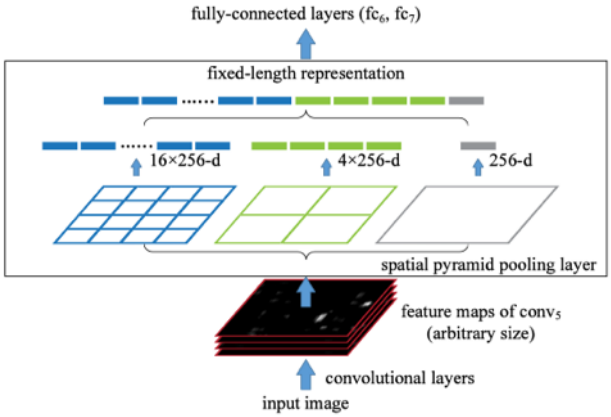


Figure 1.11: Spatial pyramid pooling

SPP-net fixes one issue with R-CNN: it makes testing fast, but it still inherits the rest of R-CNN's problems:

- Ad hoc training objectives.
- Training is slow, taking a lot of disk space.

SPP-net also creates a new issue: we can not update parameters below SPP layer during training, since the SPP layer combines a sequence of max pooling operations of

different filter sizes, the gradient can not flow through it because there is overlapping within these filters.

1.2.3 Fast R-CNN

Fast R-CNN was invented as an improved version of R-CNN architecture. It exploits the idea of swapping the order of convolution and ROI proposing which makes it fast at test time, like SPP-net. It is also a network trained end-to-end, avoiding the complex multistage architecture in R-CNN. Fast R-CNN has higher mean average precision than R-CNN and SPP-net. The network is described in Figure 1.12.

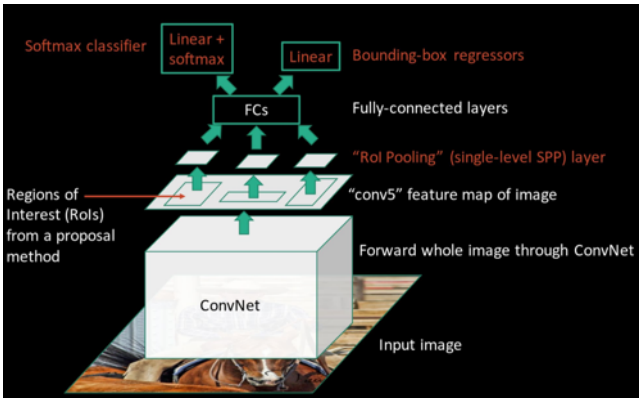


Figure 1.12: Fast R-CNN architecture

After regions of proposals are generated, fast R-CNN architecture uses "ROI Pooling" to wrap ROI into a predefined size. ROI pooling is actually a single-level SPP layer. Fast R-CNN architecture can be trained end-to-end and we don't need to train separate modules like SVMs in R-CNN architecture. By using only a single-level SPP layer, gradients can flow across ROI pooling layer and training the whole network at once is possible. Fast R-CNN is a lot faster than R-CNN, and according to the author, the training phase is 8 times faster than R-CNN and it only takes 0.32 seconds to make an inference at test time. However this inference time does not include generating regions of proposals, and we still need to use other methods like selective search to find ROI. Since we have to use selective search as an independent method to find region of proposals, why don't we implement region proposing methods inside the network and let it learn how to generate proposals during training. Faster R-CNN was created based on this idea.

1.2.4 Faster R-CNN

Summarizing all the algorithms above, we can realize that they all follow a similar pattern. First, we use some method to generate region proposals. Second, we use

classification algorithms to classify each area and use regression to fine-tune bounding box positions. So far, faster R-CNN is the most powerful algorithm exploiting this two-step approach for object detection. Faster R-CNN inserts a "Region Proposal Network"(RPN) after the last convolutional layer of fast R-CNN, thus avoids using selective search to generate ROI. RPN is trainable to produce region proposals directly and there is no need to use external region proposing methods. The rest of modules are the same as fast R-CNN. The whole structure is demonstrated in Figure 1.13 and there are four major components in this network:

1. Convolution layers. Faster R-CNN exploits the basic idea of CNN, using convolution and pooling layers to extract feature maps from original image. These feature maps are later used in RPN and fully connected layers.

2. Region Proposal Network. For each pixel value in the feature maps, 9 anchors are created as candidates for region proposals and RPN uses softmax to classify each anchor as foreground or background (A foreground anchor contains an object). Then it uses bounding box regression to modify anchors for more precise proposals.

3. ROI pooling. This layer gathers feature maps and region proposals, making it into fixed length feature vector for fully connected layers.

4. Classification. Use the fixed length feature vector for classification and final bounding box regression.

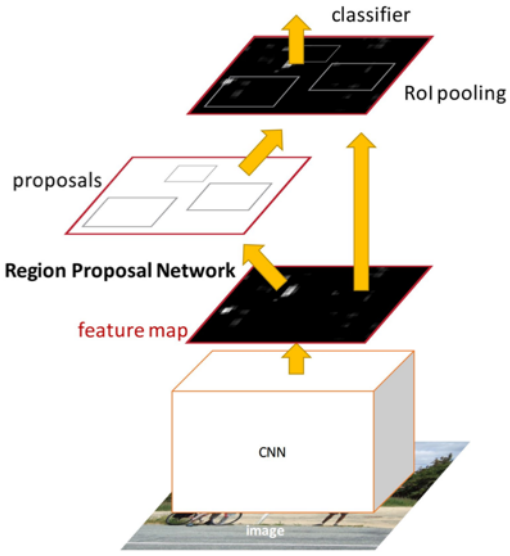


Figure 1.13: Faster R-CNN architecture

RPN is a small network for classifying object or non-object and it can regress bounding box locations as well. A sliding window is used on the convolutional feature maps

and the position of the sliding window provides location information with reference to the image. As we can see in Figure 1.15, the convolutional feature map has 256 channels and after another 3 x 3 convolution to gather local information, we can generate a feature vector of length 256 for each pixel. This feature vector is used for both classification and bounding box regression.

Since we do not know in advance the size and shape of objects, for each pixel in the feature map we create k anchor boxes around it as region proposal candidates. The reason for creating multiple anchor boxes is to match the proposal to the ground truth as close as possible. In Figure 1.14, we can see that the green box is more precise. Anchors are represented by 4 values $(c_x, c_y, width, height)$ and they can act as foreground or background. So finally we get $2k$ scores and $4k$ coordinates as the output of RPN.

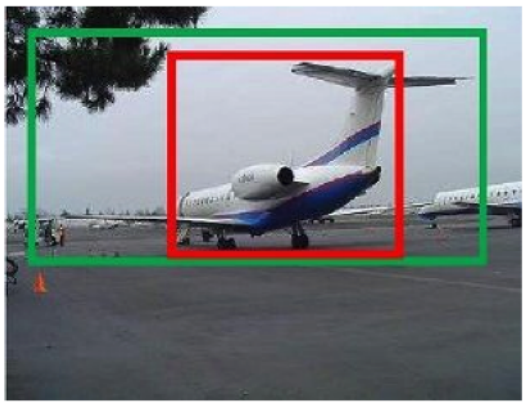


Figure 1.14: Plane anchor boxes

How many anchors will we generate for each image? Suppose the original image is 800 x 600 x 3, and we use VGG16 as our feature extraction network. VGG16 downsamples an image to its $1/16$, so if set $k = 9$, the number of anchors is calculated as:

$$ceil(800/16) \times ceil(600/16) \times 9 = 50 \times 3 \times 9 = 17100$$

There are quite a lot anchors compared to selective search which generates around $2k$ proposals per image. Actually during training we sort anchor candidates by their classification scores and randomly select 256 positive and 256 negative ones as a training batch. Intersection of union (IoU) is used as a measure of the common area between an region proposal and ground truth bounding box. If IoU is larger than 0.7, we think it as a positive anchor and if it is smaller than 0.3, we view it as negative. Those anchors having $0.3 < \text{IoU} < 0.7$ do not get involved in the training phase.

RPN learns a regressor to map pixel values form the feature map to ROI bounding box coordinates. With RPN we do not need to generate ROI candidates as a

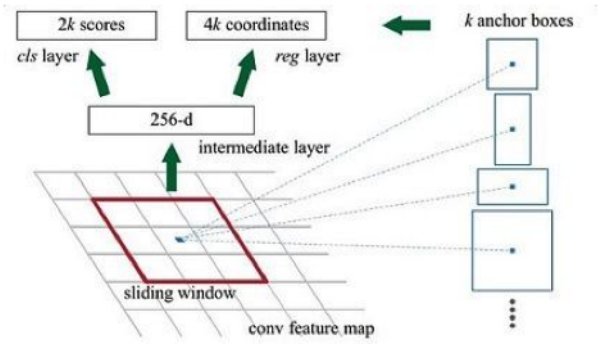


Figure 1.15: RPN

preliminary step, and this makes inference a lot quicker. Figure 1.16 is a speed comparison among object detection algorithms we just introduced. All these algorithms exploit the idea of the two-step approach: First generate region proposals from original images. Second use classification techniques to classify each area and use regression to fine-tune bounding boxes.

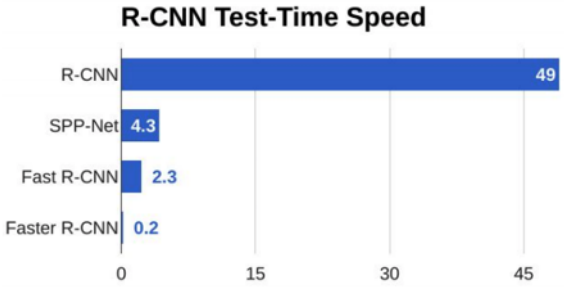


Figure 1.16: R-CNN test speed

1.3 Counting objects by classification and detection

Can we use image classification and detection techniques for solving object counting problem? Recall that the problem we face is giving an input image, we need to estimate sections in the image and this requires both classification and object counting. Also note that the dataset we have does not include a bounding box around each object, but a colored dot at the center. Here is how we may use image classification and detection techniques to solve object counting problem:

- Classification: We can use a sliding window to generate patches from an image, and then use CNN architecture to classify each patch. Suppose we have N kinds of objects in the image, and then including the background patch, we are now dealing with a $N+1$ classification problem. During inference, we first separate the image into patches and then classify each patch. We sum up all the patches

to get the final result.

- Detection and localization: Since we do not have a bounding box around each object, in order to train an object detection network we need to first create a bounding box from the center dot. By approximating the size of each object, we could create the bounding box manually. Then we can do a faster R-CNN architecture to localize each object. During inference, each bounding box prediction indicates an object existence.

In principle both of the methods should work, but problems do exist for each of them. If we generate object counts by sliding window and image classification, we naturally assume that each patch maximumly contains a single object. In order to make count prediction precise, we need to carefully set the patch size so that most of the patches contain only one object or no object at all. As for object detection, it seems over-kill here because we don't need to predict object location while we only want to estimate object counts. In our sea lion counting problem, we tried to use sliding window and image classification as a start up. We first construct the training dataset by manually extracting patches with sea lions in the center. Then we train a CNN which learned how to classify each patch correctly. Finally we use the trained network to estimate sea lion numbers from testing images.

Chapter 2

Count-ception Architecture

In this chapter, we talk about Count-ception architecture which is based on Inception modules and fully convolutional network. Inception network is proposed by Google team [1], and fully convolutional network is first analyzed in [5].

2.1 Inception Network

The Inception network is an important milestone of CNN classifiers. Before Inception network, the most straight forward way of improving the performance of deep neural networks is by increasing their size [5]. This includes both increasing the depth - the number of levels - of the network and its width: the number of units at each level. This is an easy and safe way of training higher quality models, only when we have a large amount of labeled training data. Also bigger size typically means a large number of parameters, which makes the enlarged network more prone to overfitting. The Inception network is carefully designed in terms of speed and accuracy while keeping the computational budget constant. The network exploits a good local network topology (network within a network) and then stack these modules on top of each other. The performance is verified by GoogleLeNet, a 22-layer deep network which won ILSVRC14 competition.

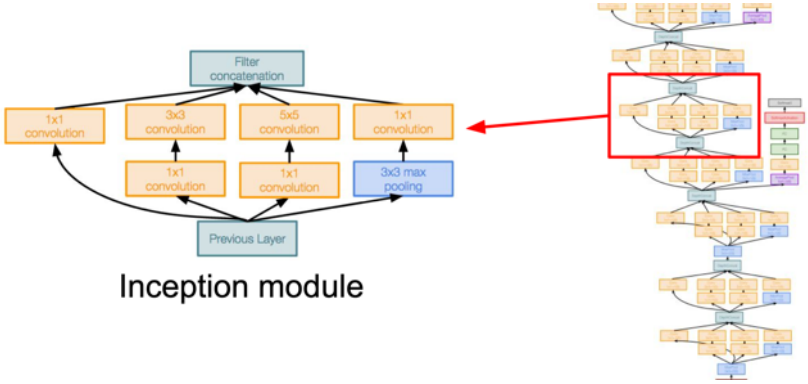
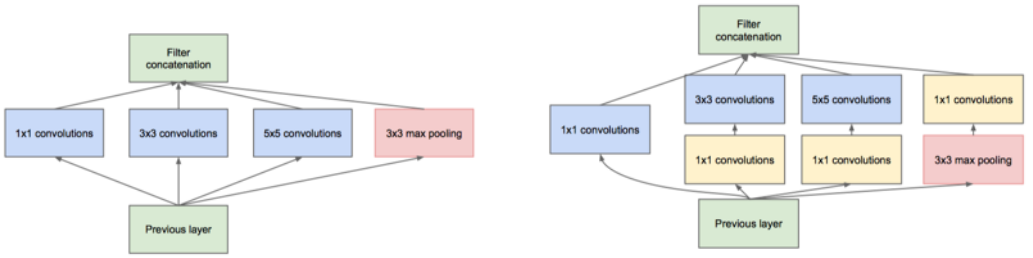


Figure 2.1: GoogleLeNet

The main difference between Inception module and normal CNN convolution layer is that Inception module uses various sizes of filters in each convolution layer while CNN uses only one. Convolution filter sizes define how much local information we would like to collect. When we increase its size, we tend to collect more spatial information and can create more sparse features. By using different filter sizes, we can learn both the sparse and non-sparse features in each layer, thus increasing the width of the network. The outputs of each filter are stacked together to form the input of the next stage. As we can see in Figure 2.2, there are three shapes of filter: 1x1, 3x3 and 5x5. Choosing these specific filter sizes is not mandatory and we could use other sizes we like. Note that in order to stack output tensors, we need to have outputs with same dimension from each filter and we can achieve this by using padding. The 1x1 filter is used for dimension reduction by decreasing output channels.

In Figure 2.2a, we can see the naive implementation of an Inception module. This implementation, however, has one big issue, the number of channels in the output tensor can explode. Since pooling operation does not change the channel number and we stack all intermediate tensors along the depth, the final output tensor will have much more channels than the input tensor. This problem becomes even more pronounced when we chain more Inception modules. The second implementation structure solves channel exploding problem, as we can see in Figure 2.2b. Whenever the computational requirements increase too much, we can apply a 1x1 convolution to reduce the dimension.



(a) Inception module, naive version (b) Inception module with dimension reductions

Figure 2.2: Inception module

Inception module is the fundamental element in Inception network. By concatenating various sizes of filters in each layer and using 1x1 filters to reduce dimensions, the network can grow wider and deeper. Since Inception module is invented, several improvements are made over the years. However in this thesis, we only use the idea of stacking various sizes of filters and ignore other tricks. So the details of the improvements for Inception networks are not discussed here, only a summary is provided.

- Inception v1 concatenates Inception modules to make the network wider and deeper. [5]
- Inception v2 uses two 3 x 3 filters to replace 5 x 5 filters, decomposes n x n filers into 1 x n and n x 1 filters in order to increase computation speed. [6]
- Inception v3 introduces RMSprop optimization algorithm, factorized 7 x 7 filters and batch normalization. [6]
- Inception v4 exploits the idea of ResNet. [4]

2.2 Fully Convolutional Network

FCN is short for fully convolutional neural net which is a convolutional network without fully connected layers. The whole network is built by convolution and pooling layers only. CNN architectures can be seen as a pipeline structure: first using several convolution and pooling layers to extract features from images, then using fully connected layers to exploit these features for classification. This structure has one disadvantage, once we set up the architecture we can not change the input image size anymore, otherwise we can not forward the tensor into fully connected layers. So it is very common to see that in many CNN architectures, the first step is to crop or warp the input image into a certain size. In many cases, the damage for this resizing operation is underestimated. SPP-net uses spatial pyramid pooling to

create fixed length feature vector, while fully convolutional network discards fully connected layers to accept different sized input images.

In fact, passing tensors through fully connected layers can be seen as a convolution operation. We can convert any fully connected layer into convolution, with one-to-one map on the weights. The number of neurons in the next fully connected layer is equal to the number of filters in the converted convolution layer and each convolution filter size is equal to the input tensor size. Let's see an example, suppose we have a CNN doing three class classification with a 128-neuron fully connected layer and we get a tensor with size 2 x 2 x 256 after several convolution and pooling operations, as shown in Figure 2.3. If we want to pass this tensor through fully connected layers, we need to first stretch it into a long vector of size 1024. Ignoring the bias, the weight matrices in fully connected layers are of size 1024 x 128 and 128 x 3. We can convert fully connected layers into convolution layers using the following steps:

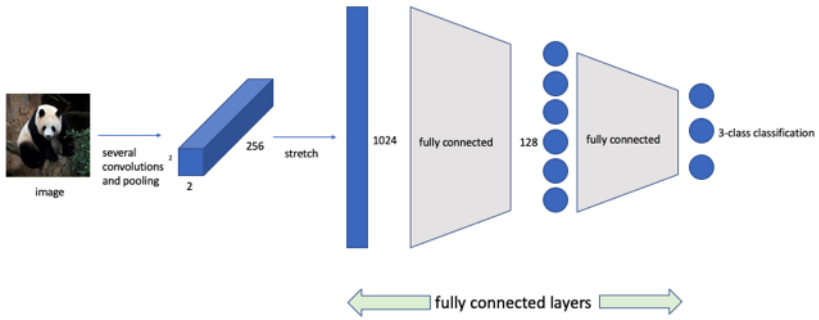


Figure 2.3: Normal CNN architecture

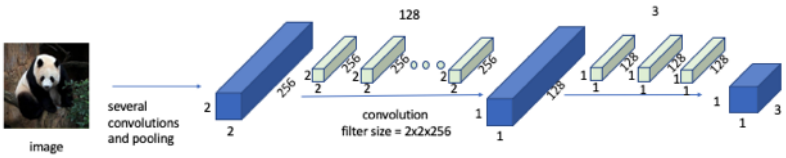


Figure 2.4: Fully Convolutional Network

1. Do not stretch the 2 x 2 x 256 tensor, instead keep the tensor unchanged.

2. Build 128 filters, each filter is a tensor of 2 x 2 x 256 which is equal to the input tensor. Passing 1024 length vector through fully connected layers can be seen as doing convolution with no padding and stride one.

3. Build 3 filters, each filter is 1 x 1 x 128 to replace the last layer in the fully connected architecture.

There is a one-to-one map between weights in fully connected layers and weights in convolution filters, as described in the Figure 2.4. In general, converting any fully connected layers into convolution operations has the following rules:

- Passing vector through fully connected layers is equivalent to doing convolution with no padding and stride one.
- The filter size is equal to the size of input tensor, and number of filters is equal to the number of neurons in the next fully connected layer.

Converting fully connected layers into convolution layers has several benefits. First, we do not need to reshape the image when we have different image size. As long as the input image size is no smaller than the filter, we can directly forward it through the network. This means we can train the network in a fully convolutional way without resizing the input image. Second, we do not get a single vector at the output, instead we get a tensor. This means if we take a trained CNN and convert its fully connected layers into convolution, when we feed the network with an image having larger size than the image size we used in training, we will not get a single probability vector but a heat map at the output. Each pixel value in the heat map is a probability value coming from a receptive field with size equal to the training image size. Thus we can exploit the heat map information for further processing, for example doing spatial correlation using the object location information. Third, modern deep learning frameworks like tensorflow and pytorch have optimization for convolution operations, so by doing computation in a fully convolutional manner, we can get the result faster than doing computation batch-wise.

Although SPP-net discards the non-convolutional portion of classification nets to make a feature extractor, it can not be learned end-to-end. Alternatively, fully convolutional network converts fully connected layers to convolution operation which makes it possible to train the whole network all at once.

If we analyze the performance of FCN, we will see that stochastic training under FCN is equivalent to batch-wise training under normal CNN [3]. A real-valued loss function composed with a FCN defines a task. If the loss function is a sum over the spatial dimensions of the final layer, $\ell(x;\theta) = \sum_{ij} \ell^{'}(x_{ij};\theta)$, its gradient will be a sum over the gradients of each of its spatial components. Thus stochastic gradient descent on ℓ computed on whole images will be the same as stochastic gradient descent on $\ell^{'}$, taking all of the final layer receptive fields as a mini-batch. When these receptive fields overlap significantly, both feedforward computation and back propagation are much more efficient when computed layer-by-layer over an entire image instead of independently patch-by-patch.

2.3 Count-ception Architecture

Count-ception network is introduced in [1]. The author uses Inception modules to build a network targeting at counting objects in the image. The whole network is a fully convolutional neural net and no pooling layer is used. There is no pooling layers in the architecture in order to not lose pixel information and make calculating receptive field easier. The network is shown in Figure 2.5, which is used for regression, each 32 x 32 region produces a 1 x 1 x 1 tensor indicating the number of objects contained in that region. After each convolution, batch normalization and leaky ReLU activation are used in order to speed up convergence. The 3 x 3 convolutions are padded so they do not reduce the tensor size and there are only two points in the network where the size is reduced.

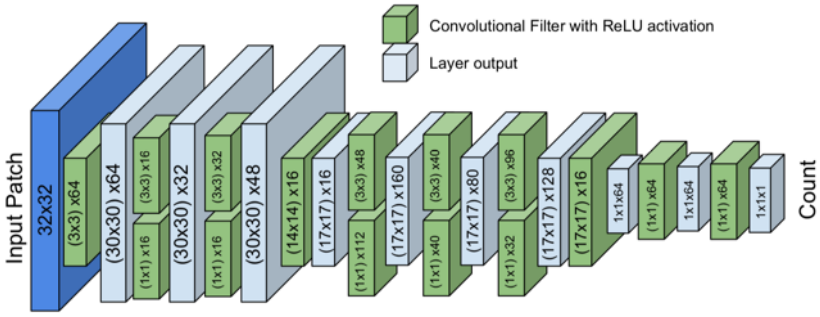


Figure 2.5: Count-ception Architecture

Instead of taking the entire image as input and producing a single prediction for the number of objects, Count-ception is a smaller network that is run over the image to produce an intermediate count map [1]. This smaller network is trained to count the number of objects in its receptive field. Moreover, the input image I is processed in a fully convolutional way to produce a matrix $F(I)$ that represents the counts of objects for a specific receptive field r x r of a sub-network that performs the counting.

A prediction map is generated using an input image and Count-ception architecture, and we need to define a loss function in order to use gradient descent to update filter weights. The target we are learning is generated from the dot labeled image and by using convolution we create a target map from this dotted image as illustrated in Figure 2.6. The convolution filter size is the same as the receptive field in Count-ception architecture. Figure 2.7 illustrates the training pipeline. Pixel-wised L1 loss is calculated between prediction map and target map.

$$min||F(I) - T||_1$$

The whole procedure for Count-ception is the following:

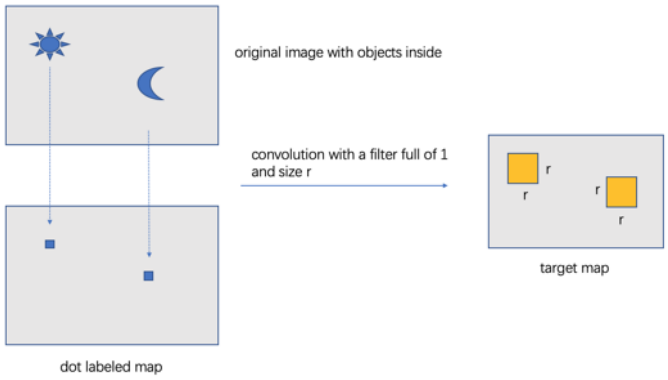


Figure 2.6: Target map construction

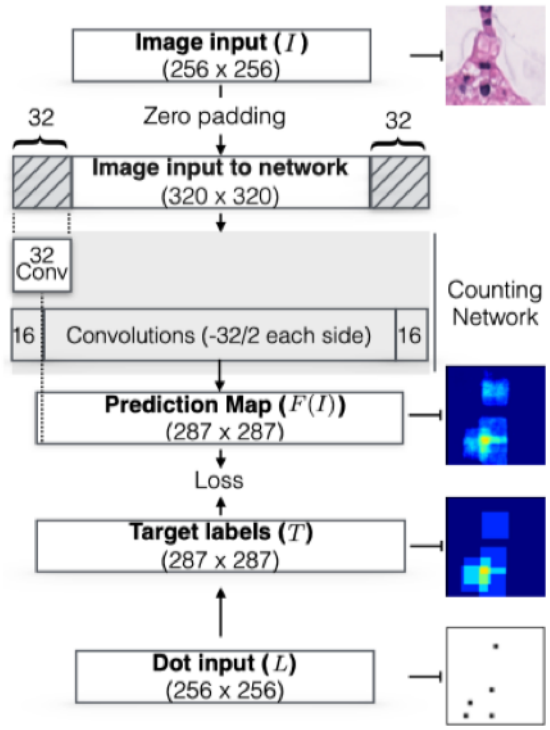


Figure 2.7: Count-ception pipeline

1. Pad the input image in order to deal with objects near the boarder.
2. Calculate the prediction map using Count-ception architecture.
3. Calculate the target map using convolution.
4. Calculate the loss between prediction map and target map. The loss function is L1 loss.
5. Use gradient descent to update the weights.
6. Sum up all the pixel values in prediction map in order to get object count prediction.

After we have trained the Count-ception network, we can calculate the prediction map for each input image. In order to get the number of objects in the image, we sum up all the pixel values in the prediction map. Note that due to convolution operation, each pixel in the input image is counted redundantly. The network is designed intentionally to count each cell multiple times in order to average over possible errors. With a stride of 1, each target is counted once for each pixel in its receptive field. We can adjust the object counts using Formula 2.1. $F(I)$ stands for the prediction map, and r is the receptive field size. In the example above, the receptive field of one pixel in the output is 32 x 32, so r equals 32. Each pixel in the input image is counted 32 x 32 times.

$$counts = \frac{\sum_{x,y} F(I)}{r^2} \tag{2.1}$$

Using Count-ception network, we sacrifice the ability to localize each cell exactly with x, y coordinates; however in many applications, accurate counting is more important that exact localization. Another issue with this approach is that a correct overall count may not come from correctly identifying cells and could be the network adapting to the average prediction for each regression [1]. One common example is when the training data contains many images without objects, the network may predict 0 in order to minimize the loss. A solution to this is to first train on a more balanced dataset and then take the well performing networks fine-tuning it on more sparse datasets. In our modified version of Count-ception network, we invent a weight balanced layer to deal with this issue. The details are provided in chapter 3.

Chapter 3

Count-ception with Classification

In this chapter, we will present our major contribution which is a modified version of Count-ception architecture, able to do classification and counting at the same time.

3.1 Architecture Overview

Before we introduce our network, let's first briefly review the original Count-ception architecture. As illustrated in Figure 2.5, each 32 x 32 area in the original image produces one output value. The entire architecture is a fully convolutional network, and each basic element is an Inception module stacking two sizes of filters. By convolution, Count-ception architecture generates a prediction map which is computed with the target map to calculate the loss. The original Count-ception architecture is used to count objects in the image, but it can not do classification, because there is only one convolution filter in the last convolution layer, thus only a single heat map is generated. Moreover the dataset used in Count-ception paper is easier to analyze compared to our sea lion dataset. A comparison between their cell image and our sea lion image is shown in Figure 3.1. They used medical imaging pictures which have very simple background environment (black) and each image is relatively small, with 256 x 256 x 3 pixels. Our sea lion dataset, however, has far more complex background, including sea, grass, and rocks and the image has quite high resolution, around 3000 x 4000 x 3 each, occupying around 5MB.

(a) Cell image (b) Sea lion image

Figure 3.1: Dataset comparison

We modified the Count-ception architecture to meet our demands, and the architecture is show in Figure 3.2. The main changes are the following:

- The receptive filed is changed from 32 x 32 to 48 x 48 by increasing the first convolution filter size to 19 x 19 x 64.
- The output is changed from 1 x 1 x 1 to 1 x 1 x 5 by increasing the number of channels in the last convolution filter.

These modifications are made in order to make the network work for our sea lion dataset and integrate classification functionality. Increasing the receptive field is necessary because our objects in the image are larger than medical imaging cells used in the original Count-ception architecture. We need to set the receptive field so that it could cover the largest possible object. Since we have totally 5 types of sea lions to count, the number of convolution filters in the last layer is increased accordingly.

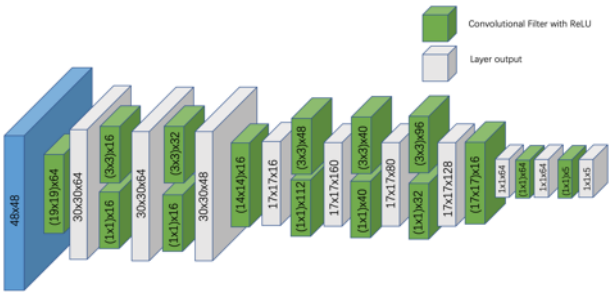


Figure 3.2: Modified Count-ception architecture

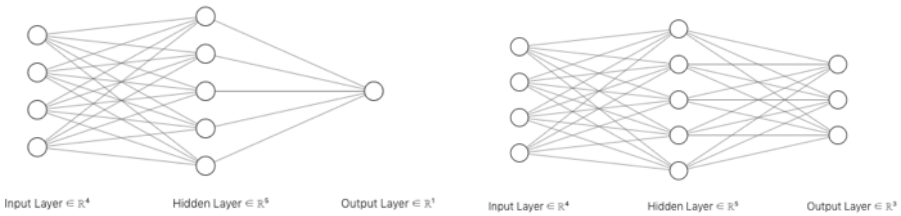
3.2 Design Choices

In the development of this thesis, we encountered some challenges due to the inherent difficulty of the sea lion dataset. In order to exploit Count-ception architecture for simultaneously object classification and counting, we did a few experiments and some major considerations are discussed below.

3.2.1 Classification

Recall that the original Count-ception architecture is used to generate a counting number for only one kind of object, and that's why we get a 1 x 1 x 1 tensor as the output value. But now we can have multiple types of objects in each image, and we would like to generate a counting number for each type of them. A straightforward way to do it is to add more channels in the last convolution layer, so that we can generate more outputs. As we can see in Figure 3.2, we have five convolution filters in the last layer while the original Count-ception network has only one. It is obvious that by increasing the number of filters, we can get more output values. The question is: Does it make sense?

To answer this question, let's examine the network architecture more in detail. Recall that Count-ception architecture is fully convolutional, meaning that there are no fully connected layers in this network. The fully connected layers are all converted into convolution layers, as the last two green blocks in Figure 3.2 shows. If we convert them back and look at the network in fully connected way, we will see that adding more convolution filters is equivalent to adding more neurons. When we do multi-class classification with CNN architecture, the number of neurons in the last layer is equal to the number of classes we want to distinguish. So it is reasonable to directly add more convolution filters to do classification. Also note that when we use Count-ception architecture to convolve with an input image, we will get a heat map as output. Now with 5 filters in the last layer, we will get five heat maps, one for each type of sea lion.



(a) Binary classification (b) multi-class classification

Figure 3.3: CNN classification

3.2.2 Receptive field and activation area

The receptive field in a convolutional neural network refers to the part of the image that is visible to one filter at a time. More specifically it defines the local area of the input which generates a single pixel value in the output. The receptive filed in the original Count-ception architecture is 32 x 32 and we increase it to 48 x 48, because we want it to fit our sea lions. Count-ception architecture is a small network that is run over the image to produce an intermediate count map, and it is trained to count the number of objects in its receptive field [1]. So should we design the network to make the receptive field large enough to cover each object? In fact we do not need to. We only need to design the network to make the "activation area" able to cover a single object, not necessarily the receptive field. We define the activation area as the local region in the dot label image which produces a positive value in the output. In [1], the authors didn't mention activation area because the objects in their cell images are already smaller than the receptive field (32 x 32). However in our experiment with the sea lion dataset, we find that using 32 x 32 receptive field gives bad performance, because the activation area is not large enough to cover each sea lion.

Recall that in chapter 2 we explained how the target construction network works. The dot label images are all black except the dots indicating object centers, and by using convolution we can generate a target map which is our learning objective. Since we are doing convolution with square filters and stride one, the positive values in the target map are in the form of squares too and each square has the same size as the receptive field, as illustrated in Figure 3.4.

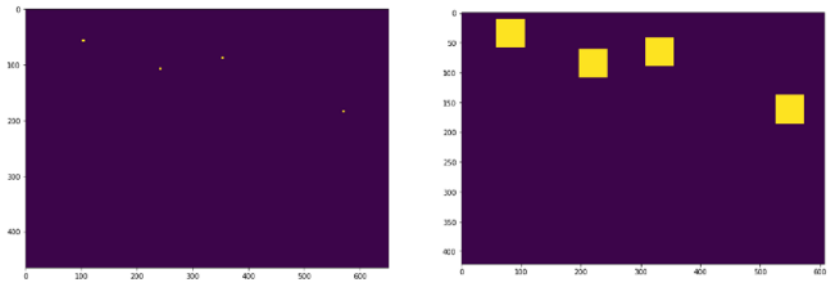
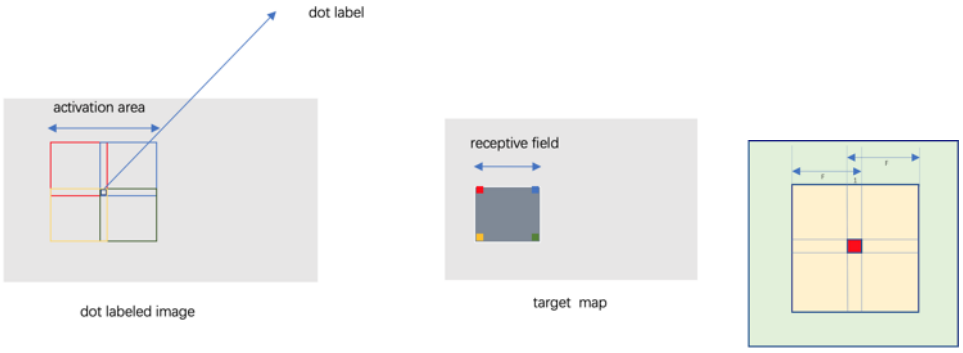


Figure 3.4: Dot label image to target map

If we analyze the convolution operation step by step, we will see that each dot label will produce a positive output only when it is inside the filter. When the dot labels are at the corners of the filter, we have the activation area boundary. So the relationship between the activation area A and filter size F is calculated by $A = 2 \times F - 1$. Also note that the filter size is equal to the receptive filed of Count-ception architecture, so the activation area is almost twice the size of the receptive filed.



(a) Relation (b) Detail

Figure 3.5: Activation area and receptive field

As long as the activation area is larger than the object size, we can proceed with the training process. Of course, we can also design the network to make the receptive field larger than the object size, however it will require more parameters to learn. The largest type of sea lion is the adult male which is around 96 x 96, so we decide to make the receptive field equal to 48 x 48.

3.2.3 Weight balance layer

Count-ception network is convolved with the input image to generate a prediction map, which is computed with the target map to calculate the L1 loss. Note that here we are calculating L1 loss with a matrix, and if we expand this formula into pixel values, we will see that the overall loss is the summation of each individual pixel loss.

$$loss = ||P - T||_1$$

$$loss = \sum_{x,y} ||P_{x,y} - T_{x,y}||_1$$

By expanding the loss function, we see that doing stochastic training with fully convolutional network is equivalent to do batch-wise training with normal CNN. So training Count-ception network directly with output maps has the same performance compared to do training with manually extracted patches and corresponding target values, as illustrated in Figure 3.6.

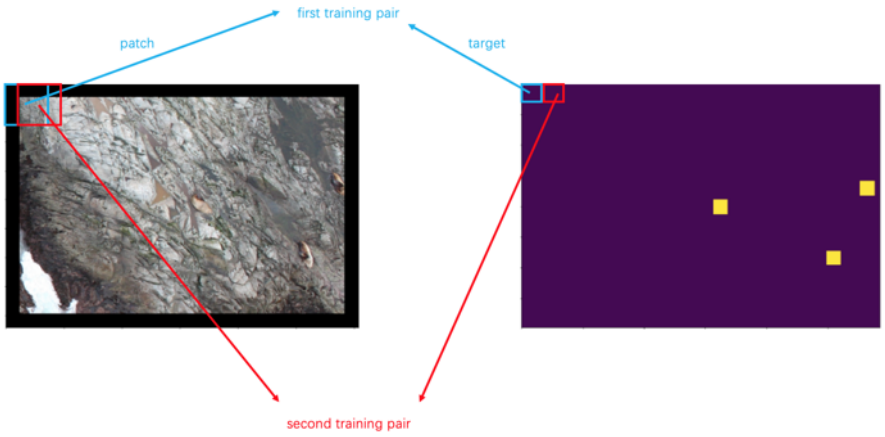


Figure 3.6: Batch-wise training

So what will happen if our target map has a low density (having very few positive areas), like the target map in Figure 3.6? If we examine the training phase in batch-wise, we'll see that there are a lot more negative patches (contain no object) than positive patches in our mini-batch. If we train our network with this unbalanced distributed training samples, our network will learn to always produce zero in order to minimize the loss.

Actually having unbalanced dataset is a common issue in machine learning and deep learning, and there are basically three ways to deal with it:

1. Upsample the minor class to enrich the dataset.
2. Downsample the major class.
3. Modify the loss function to give more weights to the minor class.

Here we adopt the third approach, because we have multiple objects in each image, thus it is not an easy task to resample the image and balance objects. In order to balance the loss function, we insert a weight balance layer L into our network and the loss function is changed to:

$$loss = ||P \times L - T \times L||_1$$

The weight balance layer has the same dimension as the prediction map, and it contains all ones except the positive output area in the target map which will have pixel value w (w>1). It is doing element-wise product with both the prediction map and the target map.

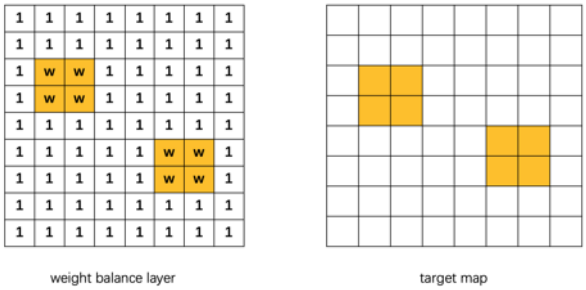


Figure 3.7: Weight balance layer

W can be chosen manually by ourselves or we can use the following formula to make positive and negative patches equal weighted.

$$w = \frac{width_T \times height_T - sum(T)}{sum(T)}$$

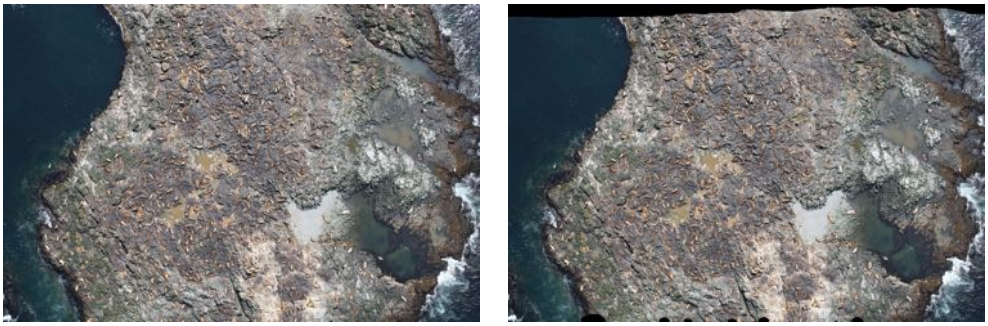
We design the weight balance layer in order to deal with low density target map issue, and the experiment results show that it is necessary if we want to produce positive counting numbers.

Chapter 4

Dataset Construction and Preprocessing

4.1 Steller Sea Lion Dataset

The goal of Steller Sea Lion Count competition is to estimate the number of each type of sea lions in a given image. The different types of sea lions are: adult males, subadult males, adult females, juveniles and pups. There are totally 947 training images and 18639 testing images. For each training image, we have two versions: the original image and the one with colored dots in the center of each sea lion. Different images may have different sizes but all the sizes are around 4500 x 3000 x 3, thus the image is quite large occupying around 5MB. The large amount of high resolution images introduce two major problems. First, during training we need to deal with memory consumption and we may need to split the image into patches or resize the image in order to fit them into GPU's memory. Second, in the testing phase, we need to have short inference time due to the huge amount testing images. In Figure 4.1, a sampled training image pair is provided. Different dot color indicates different sea lion types:



(a) Original image (b) Labeled image

Figure 4.1: Training image pair

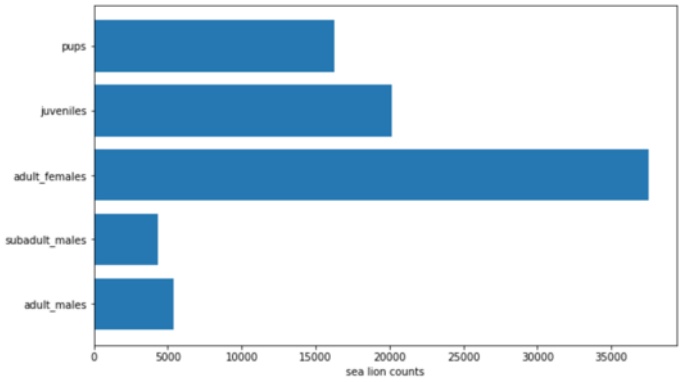


Figure 4.2: Sea lion types distribution

- red: adult males
- magenta: subadult males
- brown: adult females
- blue: juveniles
- green: pups

As we can see in the Figure 4.1, there are some black regions in the labeled images. The black regions are added by the data provider in order to filter out controversial sea lions (even the experts find it difficult to distinguish the sea lion type). Another thing to notice is that the number of sea lions per image varies a lot. We can have an image containing more than 900 sea lions or an image containing only 3. Also it is not a uniform distribution for different sea lion types. Figure 4.2 is a summary of sea lion type distribution in the whole training dataset.

In our experiments, we construct our own testing set from the 947 training images. More specifically, all the images with id $750 - 947$ are used as test set which are never been seen in the training phase. These testing images are used as the indicator of algorithm performance.

4.2 Data Preprocessing

In order to construct the training dataset, some data preprocessing is needed. First of all, we use blob detection to get the color of each centered dot and its coordinates. Then we can use these coordinates to construct dot labeled inputs which are required in Count-ception architecture. In order to deal with GPU memory consumption, we exploit two methods: image scaling and separation. Data augmentation is used to balance sea lion types and improve classification performance.

4.2.1 Blob Detection

In computer vision, blob detection methods aim at detecting regions in a digital image that differ in properties, such as brightness or color, compared to surrounding regions. Roughly speaking, a blob is a region of an image in which some properties are constant or approximately constant; all the points in a blob can be considered in some sense to be similar to each other.

A dot in labeled images is a blob which contains the similar pixel values within a small region, thus we can use blob detection to find the center coordinates of the dot. After we get the center coordinates, we can use RGB values to classify its color and get the corresponding sea lion type. Luckily we don't need to implement blob detection algorithm from scratch, there are many open source implementations for this algorithm and in this thesis we use the version provided by OpenCV. Here are the decision rules we used as a look up table in order to classify dot color using its RGB values:

Algorithm 4.1 Dot color classification

```
if R > 255 and B < 25 and G < 25:
        then red
elif R > 255 and B > 255 and G < 25:
        then magenta
elif R < 75 and B < 50 and 150 < G < 200:
        then green
elif R < 75 and 150 < B < 200 and G < 75:
        then blue
elif 60 < R < 120 and B < 50 and G < 75:
        then brown
```

4.2.2 Target Map Construction

In order to calculate the loss for Count-ception architecture, we need to construct the target map manually. After we get the coordinates for each sea lion, we can construct a dot label image indicating positions of each sea lion. This dot image has the same dimension as the original image and has 255 pixel value at each sea lion position while all the other pixel values are zero. Padding is used to deal with sea lions near the image boarder.

(a) Original image (b) dot labeled image

Figure 4.3: Pre-processed images

4.2.3 Image resizing and separation

The original sea lion image has quite high resolution and it can consume too much memory. When we train our neural network with GPU, we not only need to load the image, but also need to store the parameters and all the intermediate values which are used to calculate gradients in back propagation. In order to deal with this memory issue, we mainly exploit image scaling and separation.

In computer graphics and digital imaging, image scaling refers to the resizing of a digital image. When scaling a vector graphic image, the graphic primitives that make up the image can be scaled using geometric transformations, without loss of image quality. When scaling a raster graphics image, a new image with a higher or lower number of pixels must be generated. In case of decreasing the pixel number this usually results in a quality loss. There are quite some image rescaling algorithms, like Nearest-neighbor interpolation, bilinear and bicubic algorithm, and etc. The details of these algorithms are not our concern, and we are more interested in the downsampling effect. The sea lion images are raster graphics and we half its size to reduce memory consumption. As a result we lose some pixel information and everything becomes smaller. There are mainly two effects:

- By losing some pixel values, the information left may not be sufficient to distinguish each sea lion type.
- When we downsample the image, everything becomes smaller and thus we need re-design the receptive field size in order to better fit each sea lion.

Another method to deal with memory consumption is to separate each image into tiles and then process each tile to sum up the counts as the final result. By using this method, we do not lose pixel values and each object does not shrink. However, when we generate tiles using a non-overlapping sliding window, we may cut off some objects and this can hurt performance.

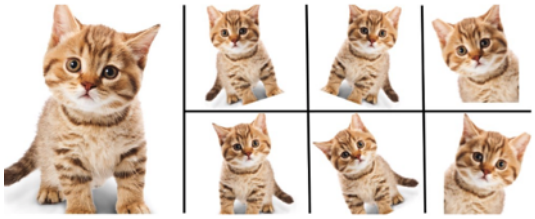


Figure 4.4: Data augmentation

In a word, both image scaling and separation have disadvantages and in fact using these two methods is just a compromise due to the limited memory of GPU.

4.2.4 Data Augmentation

Data augmentation is a common technique used to improve the performance of neural networks. It is a common knowledge that the more data an ML algorithm has access to, the more effective it can be [2]. Overfitting is a challenge for deep learning models due to insufficient amount of data and complex network structures. We can not simplify the deep learning architecture when we have a non trivial problem but we can increase the training data by data augmentation. Given an input image, we can rotate, flip and zoom it to enrich our dataset. Data augmentation is working because we manually create more learnable samples to provide for our network. It is known that CNN has translation invariance due to the calculating mechanism of convolution, but when the object has different scales or is rotated in the testing images, it is not an easy task for the network to still recognize it. Thus by manually creating more possible poses of the objects, CNN can perform better.

In Steller sea lion dataset, we use image rotation and flipping to create more training samples and improve classification performance. We did not use zooming operation because the images are taken by a drone, which flied in the same height, thus all the sea lions have the same scale.

Chapter 5

Performance Analysis

In this chapter, we present performance of our algorithm. First of all, a baseline approach is provided, using sliding window and CNN. Then we will use Count-ception architecture to do binary counting which does not require any classification. Finally, we discuss our modified version of Count-ception network, able to do classification and counting at the same time.

5.1 Baseline Approach

We use sliding window patch extraction and CNN patch classification as a baseline approach to our sea lion counting problem. The whole pipeline for this approach is provided here:

1. Extract 96 x 96 patches from original images to construct training dataset.
2. Train our CNN network which is able to do 6 class classification (5 sea lion types + background).
3. In testing phase, we manually divide the image into patches and gather the classification results as the counting prediction.

5.1.1 Patch extraction and training set construction

We already have each sea lion coordinates by using blob detection introduced in chapter 4 and we use these coordinates as patch centers. Each patch is of size 96 x 96. Including background patch, we have totally 6 classes of patches, and a sample of them is shown in Figure 5.1.

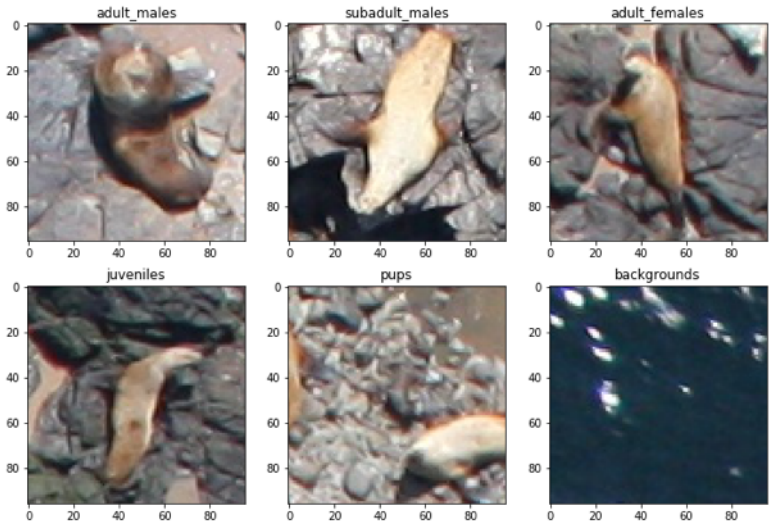


Figure 5.1: Patch types

From all the training images, we extract 1000 patches for each sea lion type, and among them 85% are used as training, 15% are used as validation. We train our network in a mini-batch manner and each mini-batch contains 30 patches under uniform sea lion distribution.

5.1.2 CNN overview

Our CNN architecture takes a 96 x 96 x 3 patch as input and classifies it among six possible labels. The whole network is shown in figure 5.2.

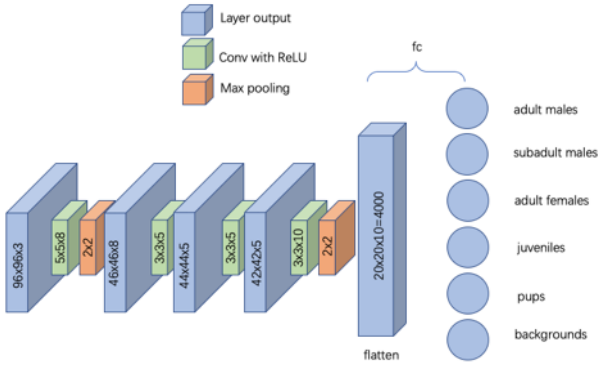


Figure 5.2: CNN architecture

The last layer in this CNN is a fully connected layer and we append a softmax layer after it to convert the outputs into classification scores. The sea lion class with the highest score will be our prediction.

5.1.3 Performance

This network is trained for 100 epochs under 10^{-5} learning rate. Since it is a classification network, we use cross entropy loss with Adam optimization and in order to have a better sense of the network performance, we add accuracy as an evaluation metric. The training and validation performance is shown in Figure 5.3. As we can see, overfitting happens after around 35 epochs. We use the model with the best validation score to conduct testing.

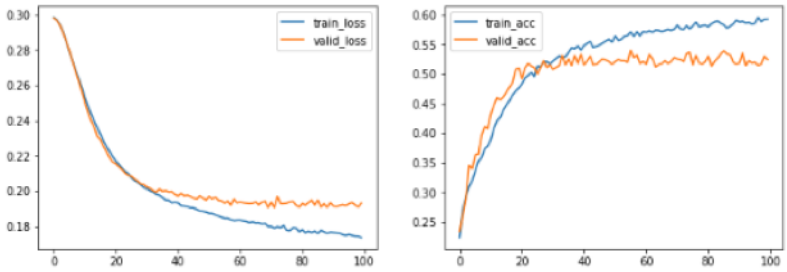


Figure 5.3: Baseline training performance

We have a six-class classification problem with uniform data distribution and the accuracy we achieved after training is around 50%. So it is actually not bad for our CNN to classify patches. As mentioned before, we can get sea lion count prediction by first separating the whole image and then sum up classification results. The test dataset we construct has totally 184 images and the testing performance is summarized in the Table 5.1.

sea lion types	adult_males	subadult_females	adult_females	juveniles	pups
average count error	25.77	40.98	7.53	199.75	314.83

Table 5.1: Baseline testing performance

We have very high counting error for juveniles and pups and there are two major reasons for this:

1. Juveniles and pups are inherently difficult to detect because they have smaller sizes compared to other sea lion types. Also pups look like rocks a lot.

2. By using non-overlapping patches with classification, we assume that the sea lions lie at the center of each patch, and there are maximumly one sea lion in each patch. However in the test images, sea lions not always lie in the patch centers, and pups for most of the time lie very near to other sea lions (possibly their mothers), so the above assumption does not hold.

The baseline approach treats counting problem as classification problem. Although we could achieve quite good results on classification, the counting performance is not so good due to the limitation of the algorithm itself.

5.2 Binary Counting with Count-ception

We create a modified version of Count-ception architecture to generate binary counts for our sea lion images. This network has receptive filed size 48 x 48 and it generates a single prediction map indicating the number of all sea lion types. We design this architecture as a startup for our final algorithm and use it to verify that our Count-ception network works for sea lion dataset.

With learning rate equals to 10^{-5}, the network is trained for 50 epochs under Adam optimization. We use 50 training images and each of them are divided into 25 non-overlapping tiles in order to deal with GPU's memory issue. One training tile example is illustrated in Figure 5.4. Thus there are totally 1250 tiles, among them 85% of them are used in training and the rest are used as validation dataset. In the testing phase, each image is first separated into 25 tiles and then sea lion count predictions of each tile are summed up to generate the final result.

Figure 5.4: Training tile example

The learning process is illustrated in Figure 5.5, both the train loss and valid loss decrease as more training epochs are done. We perform redundant counting from highly overlapping receptive fields and in order to get the final counting prediction, we normalize the summation of all the pixel values in the prediction map. Train count error decreases during training while validation count error fluctuates quite a lot, but there is a decreasing trend. The figure indicates that if we increase the training epochs, we may get better results. However since our goal is to do multi-object counting, not binary counting, the result is sufficient to show that our Count-ception architecture could handle sea lion dataset.

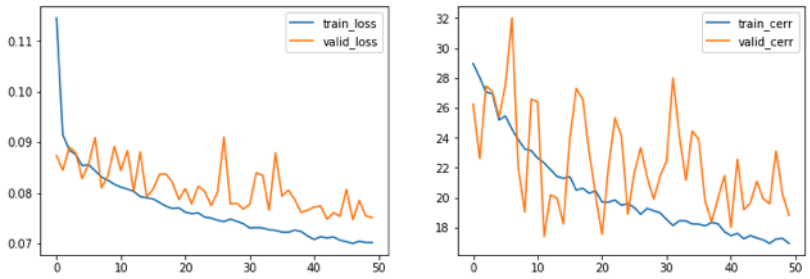
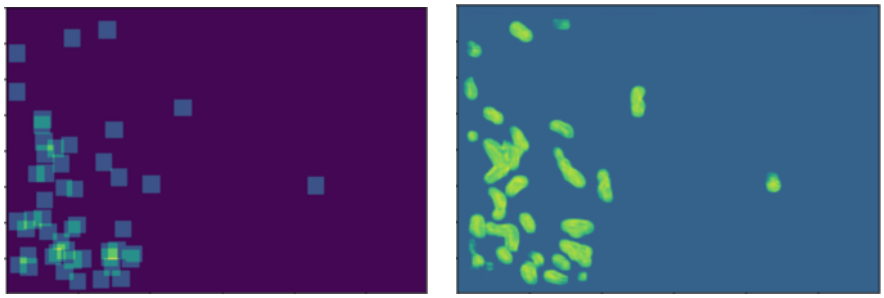


Figure 5.5: Sea lion binary learning process

The results of sea lion binary counting are not bad, and as we can see in Figure 5.6, the algorithm not only learns the sea lion positions but also sea lion shapes, even though our learning objective is squares.



(a) Target map (b) Predicted map

Figure 5.6: Sea lion binary counting performance

In our experiments with sea lion binary classification, we discovered that enlarging the receptive field size of Count-ception architecture is necessary for recognizing sea lions. We need to carefully set its size to cover the largest object. As mentioned in chapter 4, we can deal with GPU memory issue by either rescaling the image or separating the image. The result above is achieved by image separation, but we also tried rescaling. By rescaling the image to 20% of its original size, we could feed the whole image into our network. However the performance is poor due to the fact that we lose a lot of detail information. In the end, we decide to use image separation to deal with memory issue.

After modifying the receptive field size of Count-ception architecture, sea lion binary counting works and the average counting error is illustrated in Table 5.2. The numbers verify that our network is able to process images with complex backgrounds, and using full resolution images under larger receptive field size achieves the best result.

algorithms	32x32 receptive field 0.5 resolution	48x48 receptive field 0.5 resolution	48x48 receptive field 1.0 resolution
average count error	284.864	184.164	130.141

Table 5.2: Sea lion binary testing performance

The testing count error is high because our algorithm tends to generate false positives from rocks. Some rocks have very similar shapes and colors compared to sea lions. For example, the testing image shown in Figure 5.7 is full of brown rocks and our Count-ception algorithms predicts 1584 sea lions while the true count is 100.

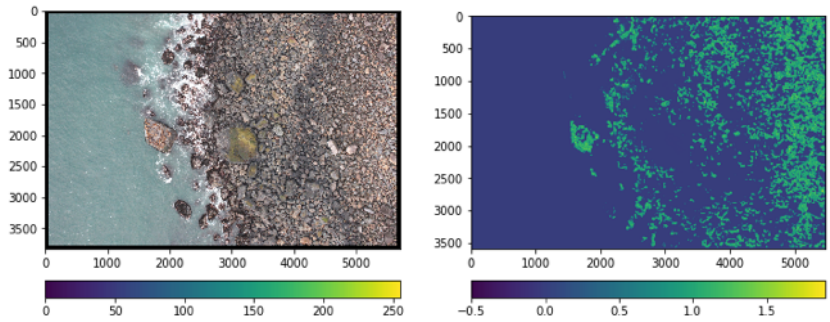


Figure 5.7: Bad case

5.3 Multi-class Counting with modified Count-ception

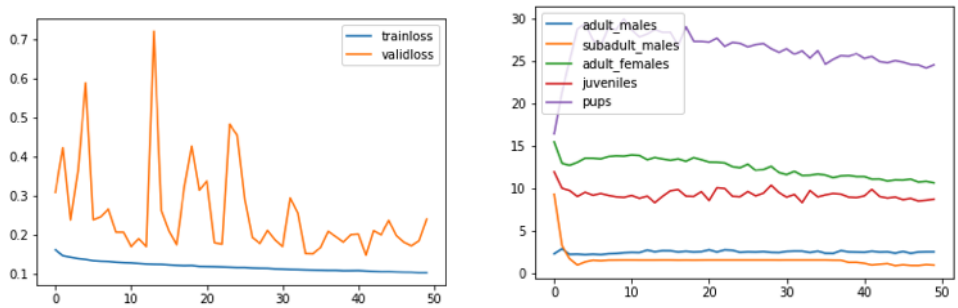
We generate multi-class heat maps by increasing the number of filters in the last convolution layer. Compared with sea lion binary counting, we are now facing five class counting which requires more parameters. This multi-class Count-ception architecture is a regression network which outputs five pixel values from a single receptive field. These values can be seen as scores for different sea lion types, for example if the area in the receptive field is sea water then all the five scores should be close to zero.

During our experiments, we encountered unbalanced sea lion distribution problem. Unlike sea lion binary counting where we only have a single heat map, here we have five heat maps and sea lions are distributed among them. Due to unbalanced distribution, some heat maps can contain a lot positive pixel values (like adult_females) while others can have very few (like adult_males). The total loss is computed as a summation of each individual pixel loss and this could lead to "always zero prediction". In order to deal with this problem, we create a weight balance layer which was discussed in chapter 4.

The whole network is trained for 50 epochs with 10^{-5} learning rate under Adam optimization. We separate each training image into 25 tiles like what we did in sea lion binary counting, however not all the tiles are used here. Even though we have

weight balance layer to adjust loss, it will not work if there are no sea lions at all. So we need to make sure that a tile contains at least one sea lion for each type. This is actually a very strict requirement and from 500 training images, we can only create 195 legal tiles. 85% of these tiles are used for training and the rest are for validation.

The train and valid loss are shown in Figure 5.8a, both of them decrease with epochs, but the decreasing trend is not so significant, indicating the inherent difficulty of learning multi sea lion objects at the same time. The fluctuating behavior of validation is mostly due to the fact that we do not have so much validation data. In Figure 5.8b, training count error for each individual sea lion type is shown. Note that for pups, the training count error is low at the beginning because the initial weights happen to make its count error low. From this figure, we can notice that there is an error decreasing trend, even though it's a slow process.



(a) Train & valid loss (b) Train count error

Figure 5.8: Sea lion complete loss

In order to understand the learning process for each sea lion type, train and valid count error is illustrated together in Figure 5.9. For some sea lion type like adult females, the train and valid error is decreasing more significantly than others which means that it is easier to distinguish adult females. For juveniles and pups, the error is not decreasing so much, indicating the difficulty for recognizing them.

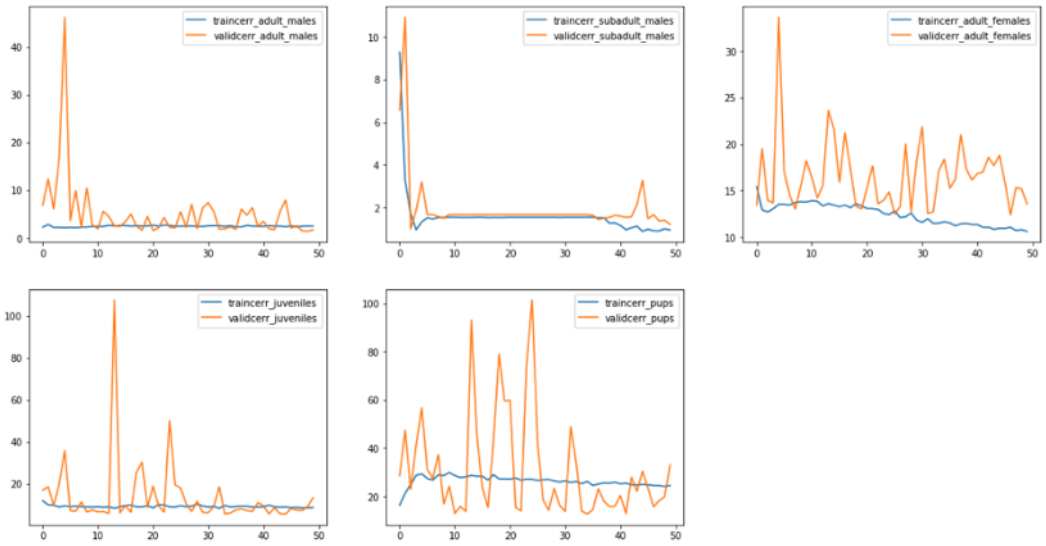


Figure 5.9: Multi-class sea lion count error

Given an input image patch, the predicted heat maps are shown in Figure 5.10. The most challenging issue for our algorithm is false positive prediction. As we can see in the heat maps, the algorithm works well for recognizing the existence of sea lions, but it works not so well for correctly classifying the sea lion type. Due to the similarity of each sea lion, our algorithm tends to generate a higher counting number than the real value. There are two major issues for our modified Count-ception network to be applied on sea lions dataset:

1. False positives from other types of sea lions and background noises like rocks.
2. It can happen that we produce correct sea lion counting number, but the sea lion activation area in the heat map is not correct. For example, the subadult males prediction in Figure 5.10.

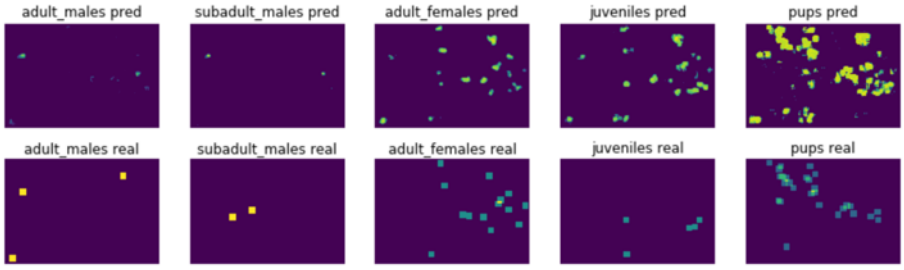


Figure 5.10: Sea lion heat maps with five classes

The testing performance is shown in Fable 5.3. Even though the previous heat maps indicate a non satisfying classification result, our algorithm which is trained under less data and fewer epochs, performs better than the baseline approach. Our modified

Count-ception network optimizes object counting by calculating the loss directly from the heat maps, and each pixel value in the input image is calculated redundantly to reduce variance. After inserting a weight balance layer, our network is able to deal with unbalanced class distribution. Moreover, during the experiments, we discovered a bonus behavior of the network which enables us to use very few positive samples to conduct training. Normally, in order to train a convolutional neural network, we need to feed a sufficient number of positive and negative samples which can be time and space consuming. However when we train a network in a fully convolutional way, positive and negative training samples are generated implicitly from convolution. For example, in our multi-sea lion counting problem, receptive fields in the input image are naturally divided into backgrounds and sea lions depending on the output pixel value. Training the network in a fully convolutional way saves us from manually extracting patches for constructing mini-batch and using weight balance layer enables us to train the network with very sparse images.

sea lion types	adult_males	subadult_females	adult_females	juveniles	pups
average count error	47.0	6.3	40.0	21..3	35.8

Table 5.3: Count-ception testing performance

Conclusions

This thesis concerns automated object classification and counting in high resolution images. The approach we take is to use a fully convolutional network based Count-ception architecture to regress a spatial density map across the image. Training our model end-to-end in a fully convolutional manner is beneficial to the images which have objects overlapping or very close to each other. High resolution images make GPU training difficult because it will occupy a lot of memory, and we tackle this problem by dividing the whole image into several tiles. The original Count-ception architecture only deals with single object counting, and we append more filters in the final convolution layer in order to integrate classification functionality. During our experiments we find that unbalanced object distribution harms the performance and moreover low density target maps will force our algorithm to output zero. We solve this problem by inserting a weight balance layer which assigns weight coefficients in the loss function. Positive pixel values will be assigned a weight factor larger than zero pixels.

Multi-objects counting in crowded images is an extremely time consuming task encountered in many real-world applications. In a lot of situations we do not need to predict exact locations for each object, thus we would like to avoid using complex models for object detection like faster R-CNN. We proposed this modified version of Count-ception architecture for simultaneously object classification and counting, and testify it on the sea lion dataset provided by NOAA. Since the sea lion dataset is inherently difficult with complex backgrounds and very similar objects, we can foresee that our algorithm can be applied for many other problems as well, including biology, remote sensing, surveillance, and etc.

Further improvements

In this thesis, there are some challenges we could not fully solve. Also there are some ideas we do not have time to test and we leave them as possible further improvements.

High resolution images

As mentioned in the chapter 4, in order to deal with GPU's memory consumption we can resort to image separation or image scaling. However both methods have disadvantages and can harm performance. We process the whole image in a fully convolutional way in order to deal with images having object clumping and overlapping, but by doing image separation we may cutoff object clusters or object itself at the very beginning. For achieving better object counting performance, it's better to use GPU with high memory.

More complex Count-ception architecture

In the thesis, we basically keep the stucture of the original Count-ception architecture. Instead of making it deeper, we keep the depth and only change the receptive filed size. The reason to do it is to make the number of parameters as small as possible in the concern of both convergence speed and memory usage. Actually it is natural to design a more complex network structure when we are facing a difficult dataset, so it is worth a try in the future.

Spatial correlation in heat maps

We generate object counts by doing redundant counting and summing up all the pixel values in the predicted map. We would say this is a very preliminary usage of heat maps. Fully convolutional networks generate heat maps which contain spatial information and we could exploit this for better performance. For example, after getting the predicted sea lion heat maps, we could combine some primer knowledge (pups appear near their mums, etc...) to do further processing, like constructing a CRF (Conditional Random Field) to do spatial correlation.

Bibliography

[1] Joseph Paul Cohen, Henry Z. Lo, and Yoshua Bengio. Count-ception: Counting by fully convolutional redundant counting. *CoRR*, abs/1703.08710, 2017.

[2] Luis Perez Jason Wang. The effectiveness of data augmentation in image classification using deep learning. 2017.

[3] Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(4):640–651, 2017.

[4] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[5] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[6] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.