Saliency Detection Implementation

Team 25: 0616076 朱彥勳、0616212 呂旻軒

Github repo link

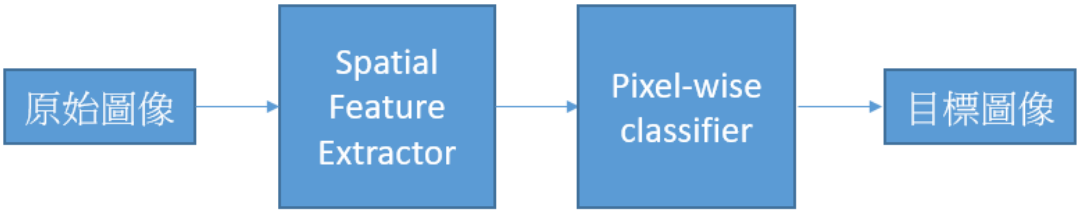
https://github.com/yenhsun1223/AI_Final_Project

Introduction:

在人類複雜的視覺系統中，為了避免大腦的認知能力過載，會採用視覺空間的注意機制，也就是大腦會篩選部分看似重要的信息進行處理。而在電腦視覺裡，也有如同人類視覺系統中的機制，稱為顯著物偵測(Salient Object Detection)。電腦透過降低複雜度的方式，以達成偵測影像中最具代表性的物件之目的，並加速對於物件辨別的速度和準確度。

最早的顯著物偵測技術往往只結合了人自行定義，或是容易辨識的特徵，像是強烈對比的顏色以及光的強度等等。透過特徵值的比對能取代實際分割圖像，需要大量的 ground truth 以及相似圖像來判斷顯著物體，這樣的設計不僅影像的複雜度不能太高，相似圖像的數據集也必須講求充足。採用 residual connection(RCRNet)的架構並有效地利用 CNN 的技術，在 spatial feature extractor 內的建立數層 convolution layers，能為顯著物偵測提供新的思路，像是無須大量的相似圖像就能獲得極高的偵測準確度，並且能明確地標註顯著物突出區域的邊界。

concept figure(RCRNet 簡易架構):



Related work:

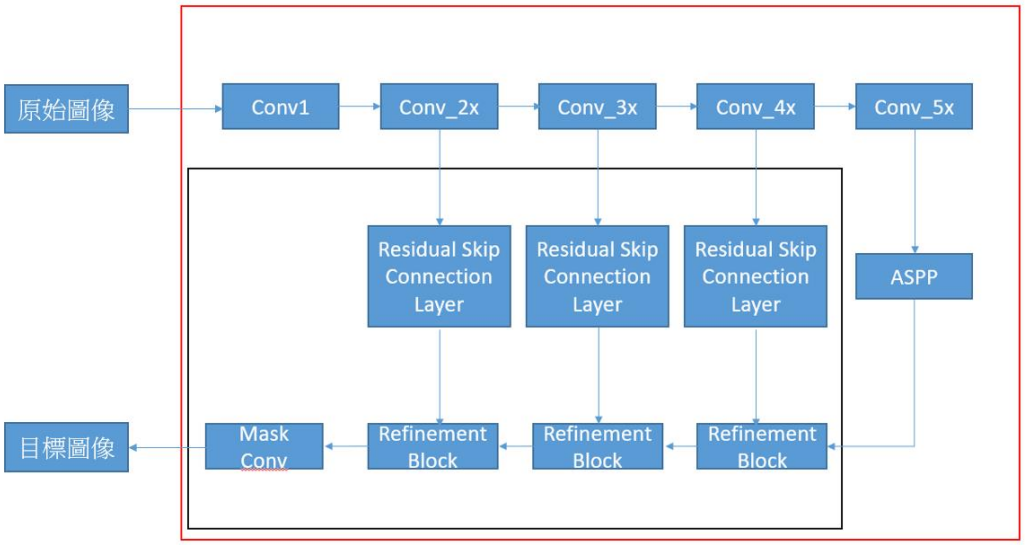
現有的顯著物偵測方式需要依賴大量且密集的 video frames 來維持具有可靠性的結果，但若採用連續的 video frames(將一段影片以極短的秒數分割成一張一張的圖片)，必須考慮如何能夠有效率地處理這些大筆資料，並且顯著物的偵測也難以保證擁有一治性的結果。倘若解決了上述的問題，訓練集也極有可能隨著網路的加深，出現了準確率下降的跡象。

Methodology:

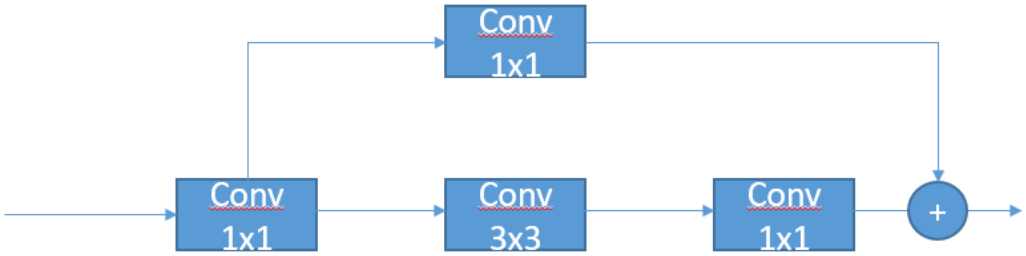
RCRNet 完整架構:

Spatial Feature Extractor(red frame-black frame)

Pixel-wise Classifier(black frame)



首先，先介紹 RCRNet 架構中的精華: ResNet。ResNet 出現的意義就是解決”隨著網路加深，準確率卻下降”的問題，這種全新架構的網路同時使用了兩種不同的 mapping 方式: identity mapping 以及 residual mapping。若將 Residual skip connection layer 放大來看，如下圖所示:



可見上方的路徑為 identity mapping；下方的路徑為 residual mapping。透過這樣的架構，一旦網路達到最佳的情況，residual mapping 將被 push 成 0，也就是剩下的訓練只會交給 identity mapping 來執行，identity mapping 透過 shortcut connection 的連線方式繼續加深網路，以達到兼顧網路深度以及準確率的兩全。另外，residual mapping 的設計是為了降低引數的數目，我們也可以稱這樣的設計為 ”bottleneck design”。像是第一層 1x1 的 convolution layer 能將高維降至低為維，再透過最後一層 1x1 的 convolution layer 將維度升回原始維度。在本次的實作中，採用的 ResNet 為 ResNet50(https://download.pytorch.org/models/resnet50-19c8e357.pth)，圖下為建構 residual mapping 三層 convolution layers 的方式:

```
self.conv1 = conv1x1(inplanes, planes)
self.bn1 = nn.BatchNorm2d(planes)
self.conv2 = conv3x3(planes, planes, stride, dilation, dilation)
self.bn2 = nn.BatchNorm2d(planes)
self.conv3 = conv1x1(planes, planes * self.expansion)
self.bn3 = nn.BatchNorm2d(planes * self.expansion)
```

再透過檢測網路狀況產生最後的 output:

```
residual = x

out = self.conv1(x)
out = self.bn1(out)
out = self.relu(out)

out = self.conv2(out)
out = self.bn2(out)
out = self.relu(out)

out = self.conv3(out)
out = self.bn3(out)

if self.downsample is not None:
    residual = self.downsample(x)

out += residual
out = self.relu(out)

return out
```

回到一開始的完整架構圖，左上方的"原始圖像"，是來自於 DAIVIS(https://davischallenge.org/)的 training data，這個網站提供大量的 JPEGImages(原始圖像)和 Annotaion(image 經過 model 後產生的最理想化結果)。而右方的 Spatial feature extractor 涵蓋的部分包括五個 convolution layers 以及 Atrous Spatial Pyramid Pooling(ASPP)。使用的方式如下:

```
block0 = self.resnet.conv1(x)
block0 = self.resnet.bn1(block0)
block0 = self.resnet.relu(block0)
block0 = self.resnet.maxpool(block0)

block1 = self.resnet.layer1(block0)
block2 = self.resnet.layer2(block1)
block3 = self.resnet.layer3(block2)
block4 = self.resnet.layer4(block3)
block4 = self.aspp(block4)
return block1, block2, block3, block4
```

其中在第五層 convolution layer 中使用的是具有 dilation rate=2 的 convolution layer，目的是為了讓輸入影像的 size 能夠等同於輸出影像的 size。在第五層 convolution layer 後附加了 ASPP module 而非 Residual skip connection layer，是因為若持續使用 Residual skip connection layer，在最後 resize 的過程中，很有可能因為 pooling 導致影像的變形，且其中要被提取的信息也會變形，從而限制了識別的準確度。為了減少這樣的損失，選擇移除 pooling 層之後，ASPP 就會因此誕生。

Pixel-wise Classifier 是產生目標圖像前的最後步驟，以程式的表現方式如圖下:

```
bu1 = self.refinement1(block3, block4)
bu1 = F.interpolate(bu1, size=block2.shape[2:], mode="bilinear", align_corners=False)
bu2 = self.refinement2(block2, bu1)
bu2 = F.interpolate(bu2, size=block1.shape[2:], mode="bilinear", align_corners=False)
bu3 = self.refinement3(block1, bu2)
bu3 = F.interpolate(bu3, size=shape, mode="bilinear", align_corners=False)
seg = self.decoder(bu3)
return seg
```

Pixel-wise Classifier 是由三個 refinement blocks 組合而成，而 refinement block 的設計是用來減少 downsampling process 過程中 spatial details 的流失。透過 bu1 定義一個 refinement block，負責匯集來自第四個 convolution layer(block 3)和第五個 convolution layer(block 4)的通道信息，並組合兩者，後面的 refinement block 也依此類推。

Pixel-wise Classifier 的輸出即為 output salient image。

Experiments:

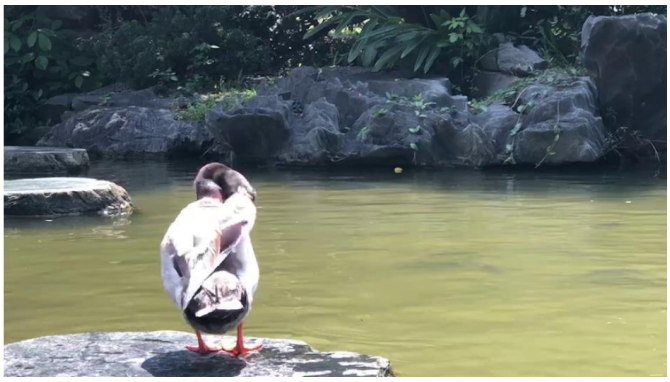
首先，借助著 yaml 擁有人性化的資料序列標準，利用 yaml 存入所有 training data 和 testing data 的資料夾資料，如下圖所示:

```
DAVIS2016:
    image_dir: JPEGImages/480p
    label_dir: Annotations/480p
    image_ext: .jpg
    label_ext: .png
    default_label_interval: 1
    video_split:
        train: ['bear', 'bmx-bumps', 'boat', 'breakdance-flare',
                'bus', 'car-turn', 'dance-jump', 'dog-agility',
                'drift-turn', 'elephant', 'flamingo', 'hike',
                'hockey', 'horsejump-low', 'kite-walk', 'lucia',
                'mallard-fly', 'motocross-bumps', 'motorbike',
                'paragliding', 'rhino', 'rollerblade', 'scooter-gray',
                'soccerball', 'stroller', 'surf', 'swing', 'tennis',
                'train']
```

```
mydataset_test:
    image_dir: JPEGImages
    label_dir: Annotations
    image_ext: .jpg
    label_ext: .png
    default_label_interval: 1
    video_split:
        test: ['mydataset']
```

其中 training data 來自於網路上現有的資源(DAVIS2016)；testing data 則來自於在學校池塘邊所錄製的一段影片中以極短的秒數分割而成的圖像。

原始圖像:



人工製作的顯著物標註圖像(實驗的理想結果):
(放在 test data 內，因此不會用於 train 的步驟，但有助於檢測不同 training model 的差異和效果)

執行指令之介紹:

1. 透過指令“`!CUDA_VISIBLE_DEVICES=0 python3 train.py --data data/datasets --checkpoint models/image_pretrained_model.pth --save-folder final/my_models --epochs 10`“ 能決定要 train 的 model(image_pretrained_model.pth)、以及 epochs 的數量。
2. 透過指令” `!CUDA_VISIBLE_DEVICES=0 python inference.py --data data/datasets --dataset mydataset_test --split test --checkpoint final/my_models/video_epoch-10.pth --results-folder final/my_results_Epoch10`” 能決定要使用的 trained model 來進行 testing。

實驗結果(以探討和對於程式修改的想法為主軸):

1. Epoch 不同以及 testing data 對於結果的影響

做法: 重複執行相同程式並將每次的結果分別存入
下圖為上述之原始圖像的實驗結果(Epoch=1 之圖像):

Epoch=10 之圖像:

Epoch=20 之圖像:

前兩張結果除了腳的部份，其他的差異性並不是很大，推測是因為動物本身的顏色和水池的顏色相差甚大，並且動物的本體也沒有和後方的石頭有過多的重疊，因此對於電腦而言較易判讀。但位於動物左側的石頭也被判讀成動物，推測是 epoch 數目不夠大；的確，在第三張圖像，誤判的資訊少上許多。

2. Overfitting 之問題

在下個階段，我們選用另一張動物本體和後方背景有重疊的圖像來檢測 epoch 個數不同的差異。

原始圖像:

Epoch=1 之圖像:

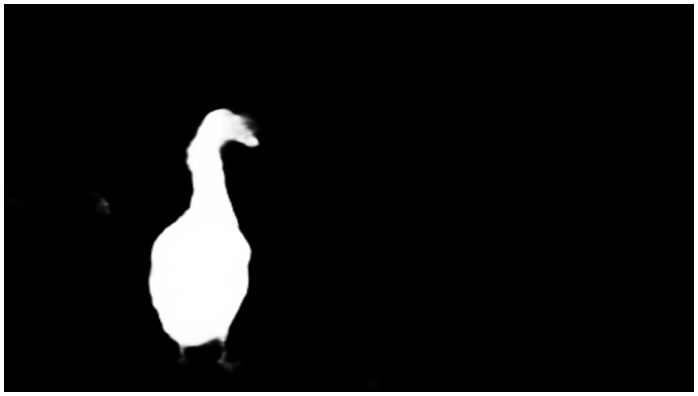
Epoch=10 之圖像:

Epoch=20 之圖像:

對於顏色重疊，也就是動物頭部的部分，epoch=20 之圖像的確創造的效果

較好，不僅動物的輪廓更加明顯，誤判的比例也少上許多。但觀察腳的輪廓，隨著 epoch 數增加卻變的越來越模糊，因此額外測試 epoch=50 之圖像，來觀察結果是否相同於推論的方向。

Epoch=50 之圖像:



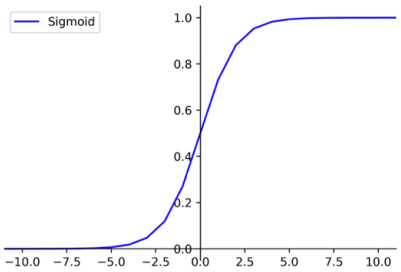
確實，此張圖像已經幾乎看不見腳的輪廓，推測是遇到 overfitting 的問題，由於模型過多學習 training data，因而降低了泛化能力(generalization ability)。

3. 相同 epoch 之前提下，Sigmoid/Tanh/Relu 的效能發揮

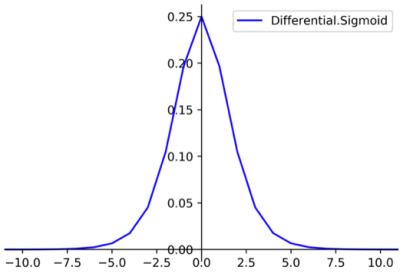
在類神經網路所使用的 activation function 中，幾乎都是利用非線性的方程式來解決各式各樣的非線性問題。一旦類神經網路缺少了 activation function，會使得各個 convolution layer 的輸入和輸出只存在著線性的關係，而單純訓練只能處理線性問題的類神經網路所產生的訓練模型，將缺乏測試的意義。

在這次實作中，我們分別使用不同的 activation function 來進行探討。雖然 sigmoid、tanh、relu 皆為常見的 activaion function，但是使用上也絕非毫無準則，隨意的使用很容易造成梯度消失(vanishing gradient)和梯度爆炸(exploding gradients)的問題。

其中 sigmoid 公式為 $f(z) = \frac{1}{1+\exp(-z)}.$ ，能將任意實數對映到(0,1)的區間，對於在特徵相差大的情況下能有不錯的效果，但 sigmoid 同時也有許多缺點，包含計算複雜且涉及除法，在多層 epoch 的重複呼叫下計算量驚人，另外在進行 backpropagation 時，需要將當層的導數和之前各層的導數進行乘積，然而 sigmoid 導數最大值僅為 1/4，這代表每進行一層的乘積，會使得整體快速的壓縮，達到 converge 導致梯度消失，無法發揮深層網路的效益，而這也在我們的實作中體現，使用 sigmoid 的效果有些瑕疵。



Sigmoid

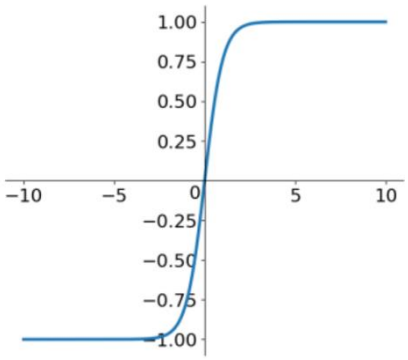


Sigmoid導數

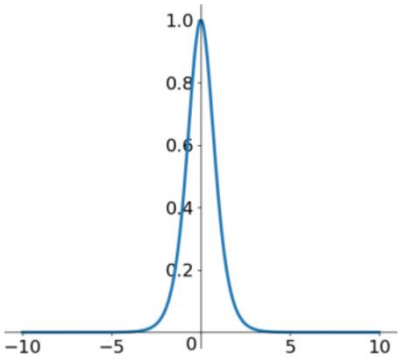
(圖片取自網路)

$$\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

而 tanh 公式為 ，能將任意實數對應到(-1,1)區間，相較於 sigmoid，tanh 屬於 0-center，在提取特徵時如果特徵差距明顯效果會非常好，因為在迴圈中會不斷擴大特徵效果，且其導數最大值為 1，雖然仍存在梯度消失的問題，但達到 converge 使得梯度消失的速率遠低於 sigmoid，所以這也解釋在我們的結果圖中，tanh 在過濾顯著物時的效果更好。



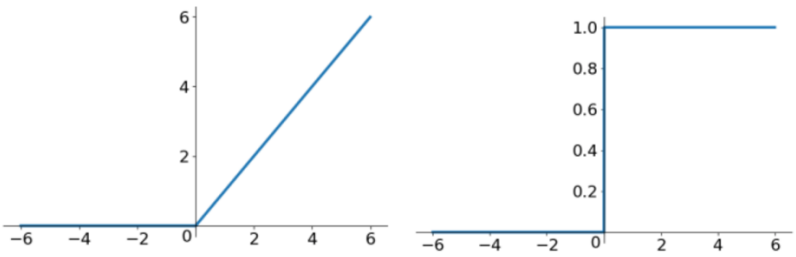
tan h



tan h導數

(圖片取自網路)

最後，relu 公式為 max(0,x)，為目前最經常被使用的 activate 函數，由於它計算簡單，且不像前者公式涉及除法，因此在多層 epoch 的大量計算下，速度方面有不小的突破，另外它成功解決了梯度消失的問題，由於其導數在正數情況下都為 1，所以在進行 backpropagation 時，各層數可以不會受到過小的導數影響，而造成更新時立馬就達到 converge，而可以走完每個 layer，避免掉梯度消失取得最好的 feature，在我們的實作中，可以看見使用 relu 的情況下，training 的時間大幅降低，且最後的成果也和 tanh 一樣能準確的標示出顯著物。



ReLU　　ReLU導數

(圖片取自網路)

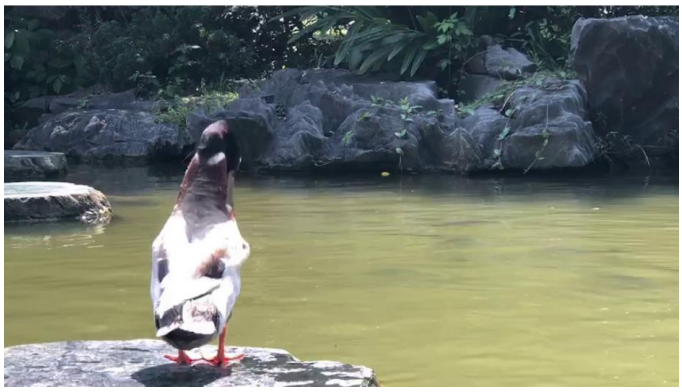
使用 sigmoid 且 epoch=10 之圖像:

使用 tanh 且 epoch=10 之圖像:

使用 relu 且 epoch=10 之圖像:

雖然在動物本身並沒有太大的差異，但在相同 epoch 的前提下，relu 對於將被景誤判成動物的失誤率較其他兩者小。若選用 sigmoid 失誤比例更高的圖像上(如下圖)，relu 優秀的發揮就更容易顯現出來。

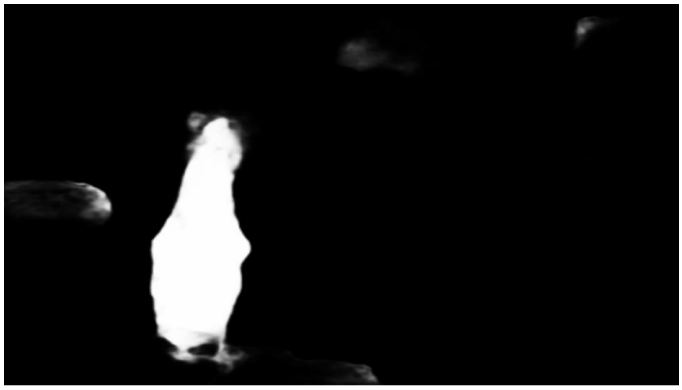
改選用的原始圖像:



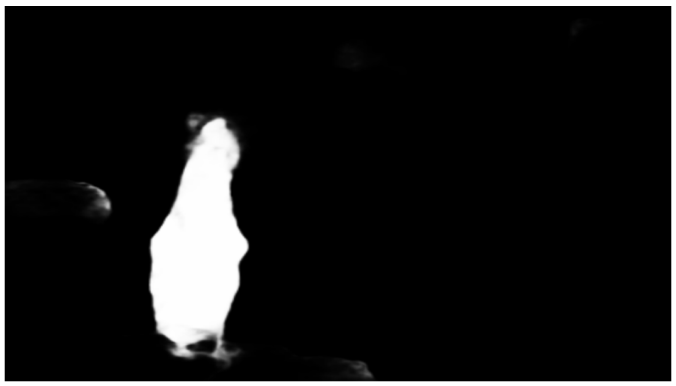
使用 sigmoid 且 epoch=10 之圖像:

選用這張圖象的目的是在使用 sigmoid 之後，這張圖象在背景的誤判率比上一張圖像高上許多，因而較易觀測使用不同的 activation function 下的效能比對。

使用 tanh 且 epoch=10 之圖像:



使用 relu 且 epoch=10 之圖像(效果改善的更加明顯):



確實，使用 relu 後，背景的誤判率少上許多。

4. 相同 epoch 之前提下，ResNet 50/101 的效能發揮

從上述對於 ResNet 的介紹，可以得知 ResNet 透過 bottleneck design 來降低引數的數目，故 ResNet 出現的意義就是解決”隨著網路加深，準確率卻下降”的問題。而在常用的 ResNet 有分為 ResNet50 和 ResNet101，在前面的實驗結果都屬 ResNet50，但我們仍針對 ResNet101 來進行兩者的比對。使用 ResNet 101 的圖像如下圖所示:

對於兩個 ResNet 的 model 之建立，可見下方兩行:

ResNet50: `model = ResNet(Bottleneck, [3, 4, 6, 3], **kwargs)`

ResNet101: `model = ResNet(Bottleneck, [3, 4, 23, 3], **kwargs)`

對於第一層 convolution layer，由於沒有降維的功能並且所有 ResNet 都擁有相同的規格，因此不用特別進行記錄。所以”`[3, 4, 6, 3]`”中的第一個變數代表的是第二個 convolution layer 中的 block 個數，往後推至第四個 convolution layer 中 ResNet50 和 ResNet101 有著 17 個 blocks 的差距，並且加上下圖的描述:

```
self.conv1 = conv1x1(inplanes, planes)
self.bn1 = nn.BatchNorm2d(planes)
self.conv2 = conv3x3(planes, planes, stride, dilation, dilation)
self.bn2 = nn.BatchNorm2d(planes)
self.conv3 = conv1x1(planes, planes * self.expansion)
self.bn3 = nn.BatchNorm2d(planes * self.expansion)
```

可知道每一層 block 裡涵蓋 3 層的 convolution layer，因此 17 個 blocks 的差距相當於 51 個 convolution layer 的差距。起初，我們的想法是由於 overfitting 的問題造成 test 完的結果與 ground truth 相差甚遠。因此我們單純取 DAVIS2016 裡面其中 50 張圖片當作新的 training data，並把 test data 設為相同的 50 張圖片，由於 training data 和 test data 完全相同，因此無論用任何方法都不會遇到 overfitting 問題，然而最後測試的結果:

從圖片中清晰可見結果仍然與 ground truth 相差甚遠。最後經由我們的推測是 training data 的數量還遠遠不及 ResNet101 中適合的採樣數量，才會因為經過過多的 convolution layers 後產生結果的變異。

Conclusion:

在報告中，我們透過實作論文的架構，體會到 RCRnet 在進行顯著物辨識的能力，它並不需要大量的測試集及 ground truth 來驗證答案，而是透過多層的 convolution layer 及 Residual skip connection layer，有效的提取少量測試集中的重要資訊，同時減少 overfitting 的發生，使得我們在實作時能快速且正確的標示顯著物輪廓，而除了實作 RCRnet 外，我們還實驗不同 epoch、不同 Activate function 以及不同的 resnet 架構對於顯著物辨識的影響，發現在提取特徵值時，convolution 的次數以及 activate 的方式對結果有不小的影響，若訓練次數過少或使用容易收斂的 function，便會失去深層網路的意義，無法精確的提取特徵，而讓 RCRnet 無法完美的過濾出顯著物，出現不少誤判，然而若訓練次數過多也不一定會出現更好的結果，過多有可能出現 overfitting 的情況，太過貼近於 training sets，造成在 testing sets 上有部分的顯著物輪廓消失，因此可得知如何定義 epoch 和 Activate function 是影響 RCRnet 結果的重要因素。而透過這些實驗，我們更加瞭解這些參數的含義，也對於整個深層網路該如何操作有更深的體會。

Reference:

https://kinpzz.com/publication/iccv19_semi_vsod/
https://github.com/Kinpzz/RCRNet-Pytorch/tree/8d9f0fe0c7ad651db7578b2d96741de11036ef82
https://github.com/kazuto1011/deeplab-pytorch
https://github.com/speedinghzl/pytorch-segmentation-toolbox
https://github.com/AlexHex7/Non-local_pytorch