

# CV Final Project Report

系級	電子一	組員學號	R07943017	組員姓名	謝世暉
系級	電子一	組員學號	R07943018	組員姓名	周育辰
系級	電子一	組員學號	R07943127	組員姓名	林彥伯

## Subject

MTK Project

## Outline

- Algorithms
  - Cost-Volume Filtering
  - Local Expansion
  - Training MCCNN
  - Max Disparity estimation
  - Real data rectification
  - Comparison and Robustness check
- Synthetic data results
- Real data results
- Bokeh-Effect

## 1. Algorithms

我們在這次 Final Project 中使用的方法是延續作業使用的 Cost-Volume Filtering 以及 Graph cut - local expansion。以下會介紹兩個方法的演算法以及我們做出的改變。

### Cost-Volume Filtering (CVF)

CVF [1]的演算法相當簡單，助教在課堂上也有做詳細的介紹了，這邊就不再做重複的說明。作業四中，我們用四級 pipeline 的方式來簡單描述這個演算法：

- Cost computation: Truncated cost
- Cost aggregation: Guided image filter
- Disparity Optimization: WTA
- Disparity Refinement: Left-right check+ hole filling+ weighted median filter

由於這次 Final 比起作業，要多了很多 imperfection，包含光影的不同、雜訊、模糊等等，而這些在作業中都沒有出現，因此需要額外處理。我們的處理方式是改掉作業四中 truncated cost 的部分，換成用 MCCNN 來算。MCCNN 在計算 cost 的時候，會做 patch-wise 的 matching，且由於他在 training 的過程中有將 patch 做 augmentation，因此 MCCNN 對於雜

訊、光影、不完美的 rectify 都相對 robust 許多。

我們使用的 MCCNN [2]是 pre-train 在 Middlebury 的 training data 上的，我們也有嘗試使用 pre-train 在 Kitti 和 Kitti2015 訓練集的 model，結果上沒有差太多。除了 cost 從 truncated cost 改成 MCCNN 之外，其他部分沒有什麼改變(只有參數的微調)，做出來的結果如下：

- MTK Synthetic avg-time: 49 seconds
- MTK Synthetic avg-err: 1.915%
- Middlebury V2 bad-1.0: 4.44%

每張圖在我們的工作站上約跑 49 秒，平均 error 是 1.915%，已經是還不錯的結果了，如果使用 truncated cost，沒有將圖片經過任何前處理的話，結果會幾乎全部壞掉，也可以從這邊看出 MCCNN 對光影改變的 robust 程度。

我們也拿了作業四的四張圖來測試是否有所進步，使用 truncated cost 的同一份 code 在作業中得到 5.18 的 bad-1.0 error，換成 MCCNN 之後進步到 4.44，已經幾乎跟使用 graph cut+ truncated cost 的錯誤率(4.37)相當了。

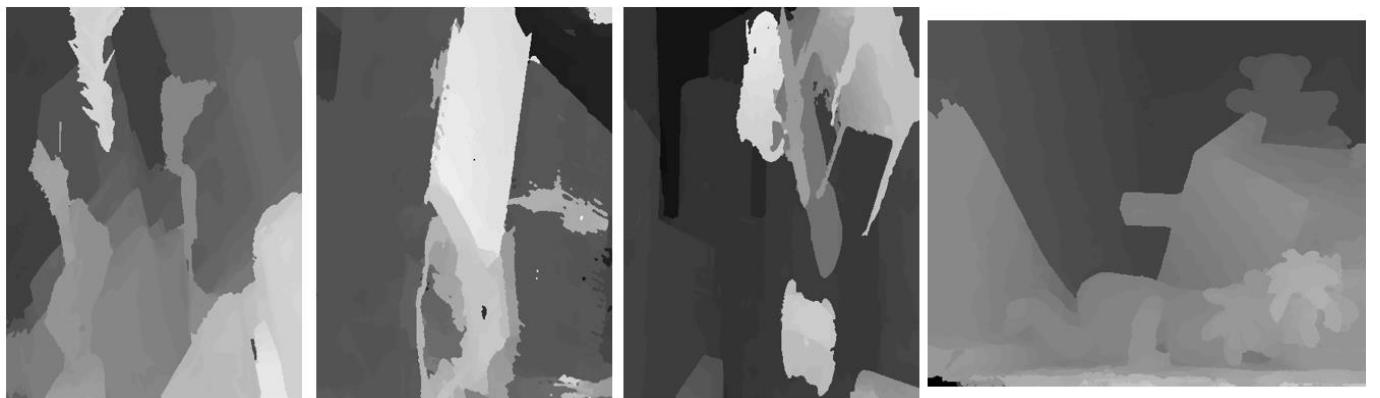


Fig.1. CVF+ pre-trained MCCNN results on MTK data and MiddleBury data

在 Real data 上，在 data 經過 rectify 後(後面會再說明如何 rectify)，直接套用上述方法也能解出很不錯的結果：

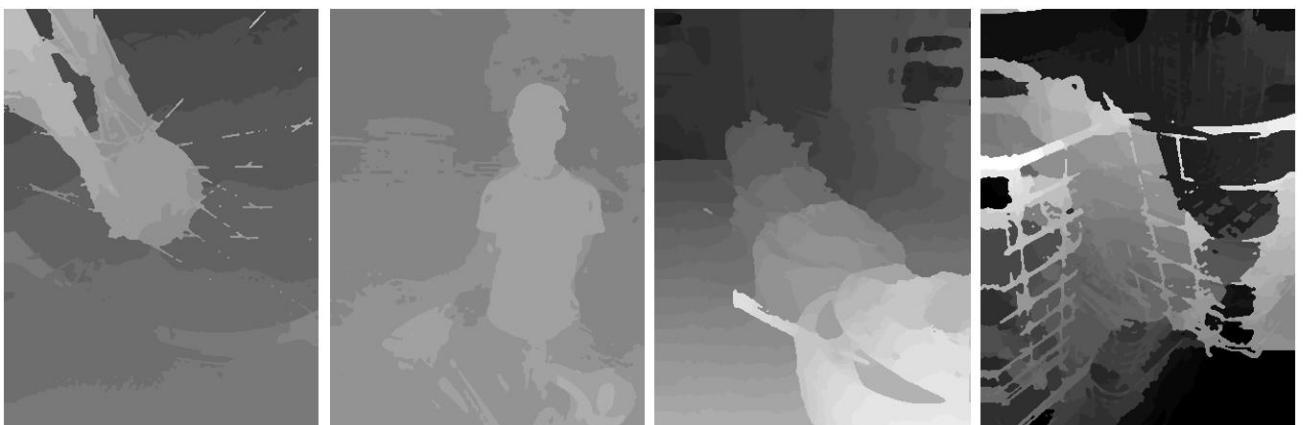


Fig.2. CVF+ pre-trained MCCNN real data results

## Graph cut - Local Expansion (LE)

我們實作的另一種方法是 Local Expansion [3]，由於這部分上課沒有提到，可能也比較少人在作業中實作這個方法，演算法的部分會做比較詳細的介紹。

由於現實生活中大部分物體都是由平面組成，通常深度圖也都會是一些連續的平面，僅在少部分的接合處有不連續存在，這個方法從這個假設出發，希望能用 graph cut 解出 cost 小且又連續的結果。

不同於 CVF，這邊每個 pixel 會被 assign 一個 3D vector  $(a_p, b_p, c_p)$  作為 label。假設一個 pixel 位於影像中的  $(x, y, 1)$  這個位置，則這個 pixel 的 disparity 就會是兩個 vector 的內積：

$$\text{Disparity} = (a_p, b_p, c_p) \cdot (x, y, 1) = a_p x + b_p y + c_p$$

每一個 3D label 可以看成是 pixel 在一個 3D 空間中位於的平面的參數，其中 3D 空間的維度分別是  $(x, y, \text{disparity})$ 。因此，同一個 label 的 pixel 會屬於同一個 3D 平面，但未必會擁有同樣的 disparity。

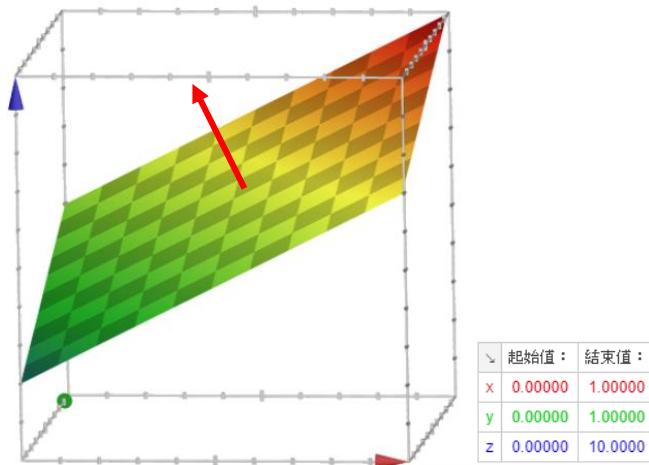


Fig.3. 一個  $(a_p, b_p, c_p) = (5, 3, 2)$  的示意圖

此方法是用 Graph cut 去 iteratively optimize 下面這個 Energy function:

$$E(f) = \sum_{p \in \Omega} \phi_p(f_p) + \lambda \sum_{(p,q) \in \mathcal{N}} \psi_{pq}(f_p, f_q)$$

其中  $\phi_p(f_p)$  是 data term，代表將 pixel p assign 給 label  $f_p$  的 cost，而  $\psi_{pq}(f_p, f_q)$  則是 smoothness term，代表將相鄰的兩個 pixel  $(p, q)$  assign 到不同的 label  $(f_p, f_q)$  的 cost。下面分別介紹這兩項的數學式與意義。

### Data cost

$$\phi_p(f_p) = \sum_{s \in W_p} \omega_{ps} \rho(s|f_p). \quad (1)$$

Data cost 的式子如(1)所示，包含了 cost computation 以及 cost aggregation 兩個部分。對於每一個 pixel p，在計算 cost 時會開一個中心點在 p 的 window  $W_p$  (Figure 4)，並計算每一個在 window  $W_p$ 中的 pixel s 的 cost ( $\rho(s|f_p)$ )，最後再做加權平均加總起來。

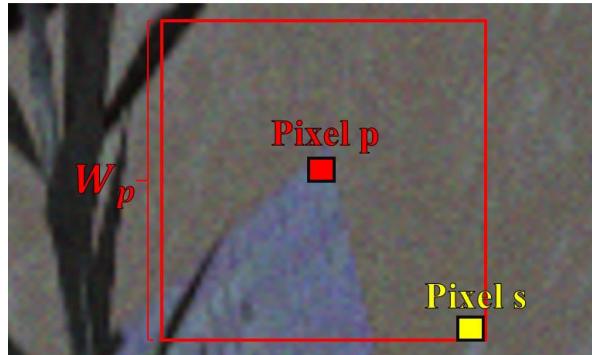


Fig.4. Data cost window 示意圖

$\rho(s|f_p)$ 的意思就是將 pixel s assign 給 label  $f_p$ 的純 cost (without aggregation)，paper 中使用 truncated cost，而我們將其換成 MCCNN。這邊值得注意的是，在 CVF 中計算 cost 的時候通常會先將每個 disparity 都算出一張 cost map，假設 max disparity 是 60 則會算出  $61*W*H$  的 cost map set，以便後續使用。但在這邊由於 disparity 會有小數(因為 label 本身未必由整數組成)，如果直接四捨五入會喪失太多資訊，而 paper 中是做 bilinear interpolation 找出 cost 的內插值，但這樣每次都會需要重新計算，會花費比較多時間，我們選擇一個折衷方案：假設 max disparity 是 60，我們會算出  $601*W*H$  的 cost map set，以 0.1 格做為一個單位，因此若 disparity 是 10.46，我們會直接在小數點下第一位四捨五入到 10.5，並以  $10.5*10$  作為 index 找出該點 pixel 的 cost。好處是喪失較少資訊，速度也不會變慢，但需要用到更多的 memory。

$\omega_{ps}$ 在論文中使用 guide image filtering 的 weight，而我們有實作 guided image filter 和 bilateral filter 兩種版本，由於 guided image filter 是一個  $O(1)$ 的 filter，好處明顯是可以比較快，但我們實作的結果發現 bilateral filter 的結果比 guided image filter 要好得多，但由於 Window size 很大( $41*41$ )，用 bilateral 的會花很久的時間，兩者大約差 3 倍的時間(performance 大約差 0.2%)。

## Smoothness cost

Smoothness cost 的式子如下：

$$\psi_{pq}(f_p, f_q) = \max(w_{pq}, \epsilon) \min(\bar{\psi}_{pq}(f_p, f_q), \tau_{\text{dis}}).$$

$$w_{pq} = e^{-\|I_L(p) - I_L(q)\|_1/\gamma} \quad \bar{\psi}_{pq}(f_p, f_q) = |d_p(f_p) - d_p(f_q)| + |d_q(f_q) - d_q(f_p)|$$

這個 cost 的設計相當的聰明，它的概念最主要在  $\bar{\psi}_{pq}(f_p, f_q)$  這一項上。假設 p 和 q 的 label 分別是  $f_p$  和  $f_q$ ，則  $\bar{\psi}_{pq}(f_p, f_q)$  就是 p 同時帶入 plane  $f_p, f_q$  的 disparity difference，加上 q 同時帶入 plane  $f_p, f_q$  的 disparity difference (如 Figure 5 所示)。如果  $f_p = f_q$ ，則  $\bar{\psi}_{pq}(f_p, f_q)$  就會是 0。因此只會 penalize 相鄰 pixel 不同平面這件事。

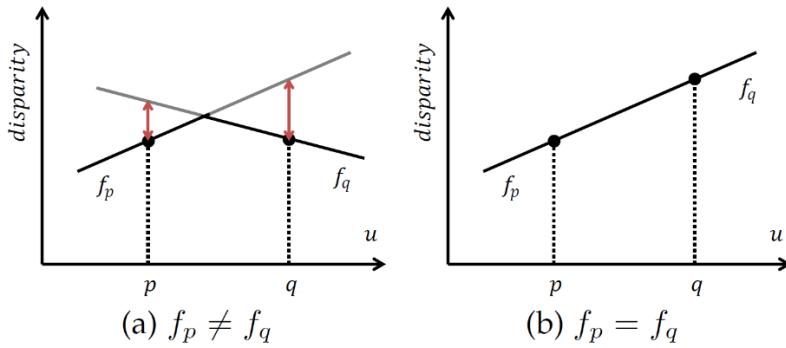


Fig.5. Smoothness Cost

這個 cost 最聰明的地方在於它符合 submodularity 的特性，這個特性要求 cost 符合下列不等式：

$$\psi(\alpha, \alpha) + \psi(\beta, \gamma) \leq \psi(\beta, \alpha) + \psi(\alpha, \gamma)$$

需要符合這個不等式的原因是之後 graph cut 需要做 alpha expansion，而符合這個特性是 alpha expansion 能正確做 graph cut 的必要條件，相關原因在 2001 年的一篇論文[4]中解釋的相當詳細，這邊也不再贅述。

由於這是個設計的很好的 cost，我們就延續使用這個 cost，不再做修改。

### Expansion moves

前面提到每個 pixel label 會是一個 3D vector，因此 label 的可能性會有無窮多種，又不可能做 exhaustive search，local expansion 關鍵的地方就在於提出能比較快找出一組不錯的解的方法。以概念上來說，這個方法就是希望將整張圖中好的 label 擴散出去，取代掉比較差的 label，並不斷重複做直到收斂。

我們借用 paper 上的圖做更詳細的解釋：

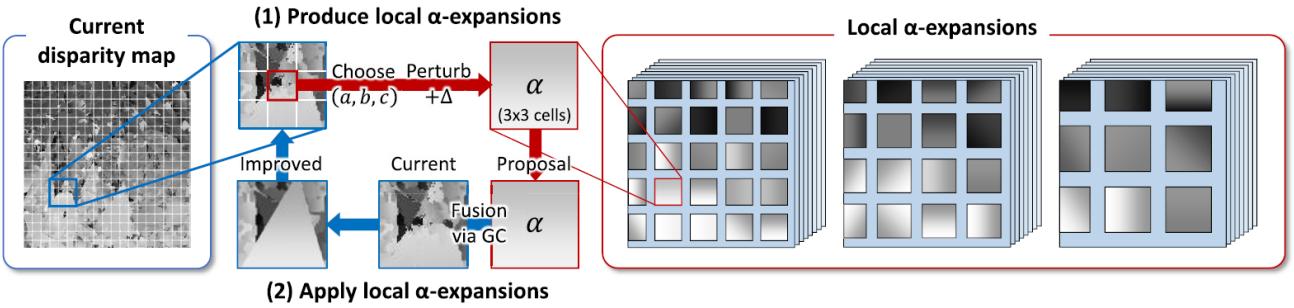


Fig.6. Local expansion flow

他們提出的方法如下，給定目前的 label map，會在每個 local window 隨機選擇一個 label，並且將這個 label 向外擴張，建立一張 proposal。Proposal 就是由幾個獨立的小區域所組成(如下圖)，每個小區域只有一個 label，因此會是一個 3D 平面，而這個 label 會從目前的 label map 中隨機選出來，並加上一點 perturbation。小區域的 window size 也有不同的選擇( $5*5$ 、 $15*15$ 、 $25*25$ )，以來做到不同程度的 spatial propagation。可以觀察到每個小區域之間會被隔開，因此彼此之間會是 independent。

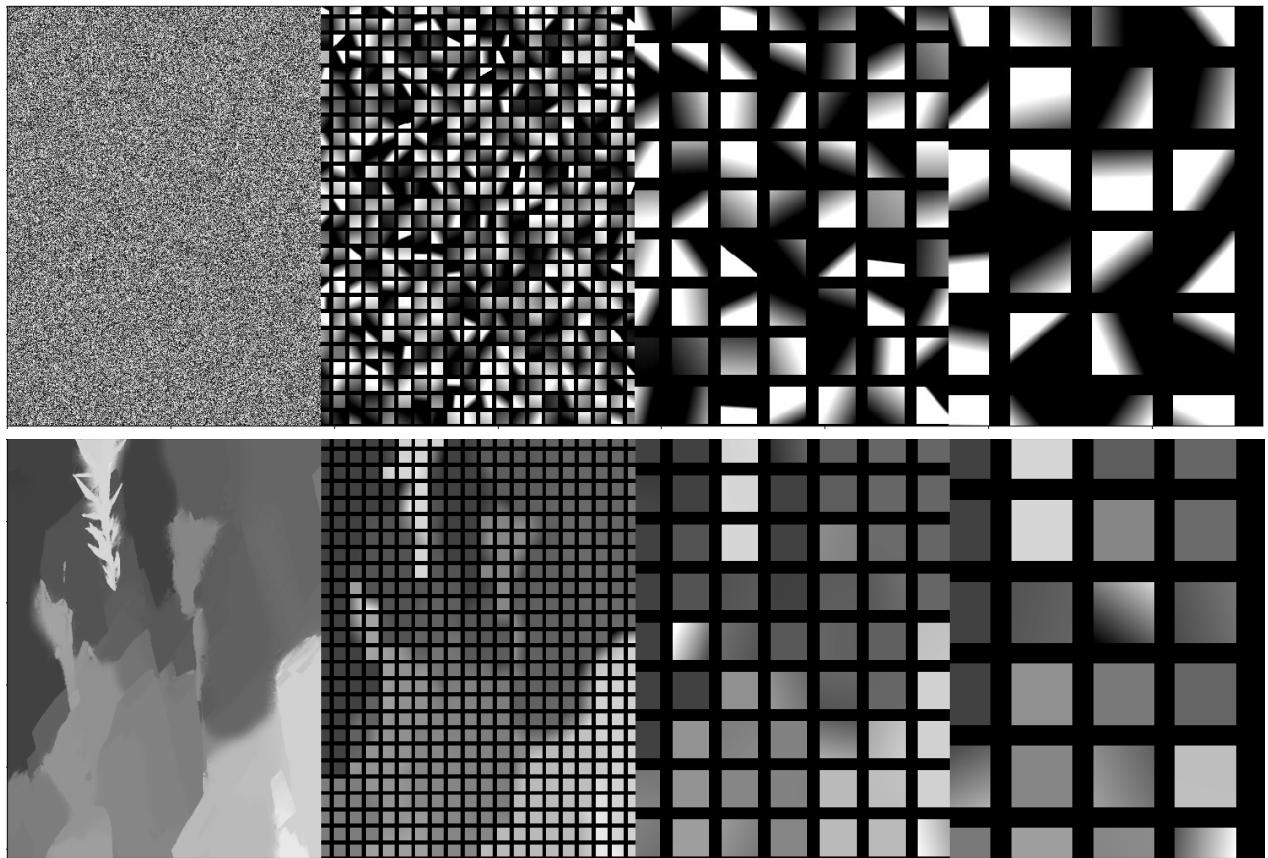


Fig.7. 上: random initialized label map and its proposals; 下: initialized with CVF and its proposals

有了 proposal 之後，下一步會將 proposal 和 current label 做 fusion，做 fusion 的方式是使用 graph cut。由於每個小區域只有一個 label，同時彼此之間又沒有交互關係，可以將 fusion 視為每個小區域獨立去做 alpha expansion。Proposal 的 label 會是  $\alpha$ ，current label 則是  $\bar{\alpha}$ ，並讓

graph cut 做切割，使一部份的 pixel label 保持原 label，另一部份則換為 $\alpha$ 。實作上，我們是整張圖一次做 graph cut，在 proposal 沒有 label 的地方(上圖黑色橫條、直條區域)則給他 infinite 的 data cost，讓 graph cut 一定不會切到那條 edge。

Alpha expansion 比較麻煩的地方在於需要建立 auxiliary node，如下圖的 a、b node。並且 edge weight 也需要考慮清楚哪一條 edge 應該給什麼 weight 才合理，如果沒有完全清楚 alpha expansion 的概念，很容易接錯而 energy 就不會嚴格遞減，因此這部分我們也花了很多時間在 debug。

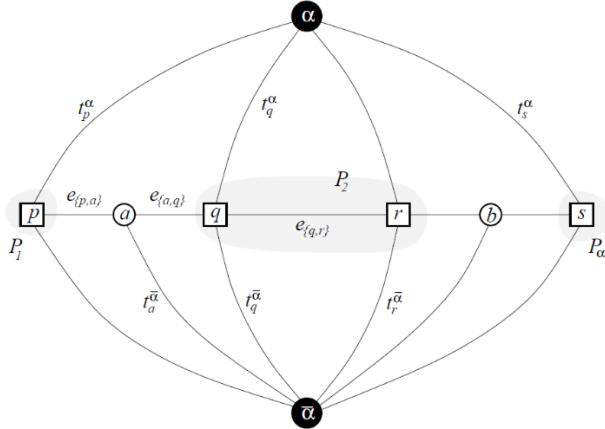


Fig.8. An example of a 1D alpha expansion graph establishment

我們跟原論文在這邊也有一點不同，原論文是用 random 的方式做初始化，但這樣會讓 graph cut 花比較多時間才能收斂，我們因為已經有先做出 CVF 的方法，就直接從 CVF 的結果做初始化，但這會有一個問題，就是 CVF 的結果是 disparity，而我們要的是 3D 平面的三個參數( $a_p, b_p, c_p$ )。我們解決的方式也很簡單，就是在每個 pixel 附近開一個 window，並且用 linear regression 去找出一個最小 error 的平面。

以一個  $7*7$  的 window 為例，假設下面這樣的情況:

$d_{i,j}$						$d_{i,j+6}$
			$d_{i+3,j+3}$			
$d_{i+6,j}$						$d_{i+6,j+6}$

$$\begin{bmatrix} i & j & 1 \\ i+1 & j & 1 \\ i+2 & j & 1 \\ \vdots & & \\ i+6 & j+6 & 1 \end{bmatrix} \begin{bmatrix} a_p \\ b_p \\ c_p \end{bmatrix} = \begin{bmatrix} d_{0,0} \\ d_{1,0} \\ d_{2,0} \\ \vdots \\ d_{6,6} \end{bmatrix}$$

$$A \quad x \quad b$$

Fig.9. Local window plane fitting

其中  $d_{i+3,j+3}$  是中心 pixel 的 disparity，附近 pixel 的 disparity 則是從  $d_{i,j}$  到  $d_{i+6,j+6}$ 。我們所解的問題就如 figure 9 右圖所示，找出  $x$  使得  $\|Ax - b\|^2$  最小的一個簡單的線性迴歸問題。我們另外也會在矩陣上乘上 bilateral weight  $w$ ，以避免被跟中心 pixel 不像的 pixel 影響到，因此實際上是解  $\min_x \|\omega(Ax - b)\|^2$  這樣的問題。

使用 CVF 做初始化和隨機初始化相比，大約可以讓 graph cut 少掉一半的 iteration，也能達到差不多的結果。下面記錄幾張 graph cut 從隨機初始化逐漸進化的過程：

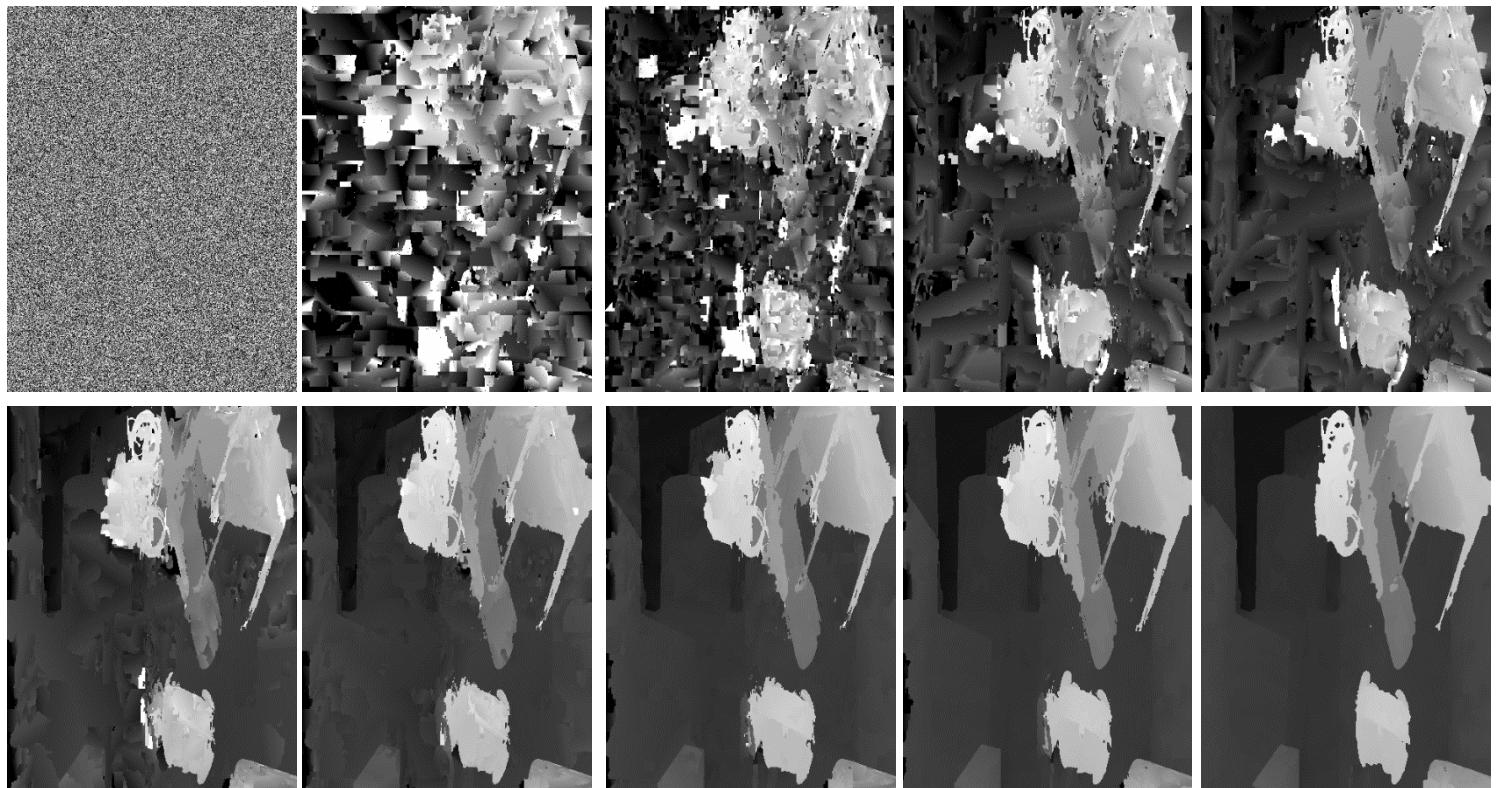


Fig. 10. Graph cut results in each iterations

Graph cut 的最大缺點就是花費的時間太久，我們的設定跟 paper 不太一樣，outer iteration 設定為 3 個， $5*5/15*15/25*25$  的 proposal 在每個 outer iteration 分別跑 32/16/16 個 iteration。而因為 post-processing 要做 left-right check，總共需要  $3*(32+16+16)*2=384$  個 iterations 才能跑完。每個 iteration 要做的事情包含建立 proposal，計算 data cost、smoothness cost、建 graph、maximum flow 解 graph。大概每個 iteration 要 20 秒的時間，因此總共要約 8000 秒才能解完一張影像，非常耗時。事實上，原 paper 即便用 C++寫、使用 guided image filter weight、加上 GPU 加速，也大約要數百秒到上千秒才能解完一張，所以本質上就是一個花時間的解法。

我們一樣將這個方法測試在 MTK 合成測資以及作業四的 middlebury 測資，結果如下：

- MTK Synthetic avg-time: ~8100 seconds
- MTK Synthetic avg-err: 1.678%
- Middlebury V2 bad-1.0: 3.63%

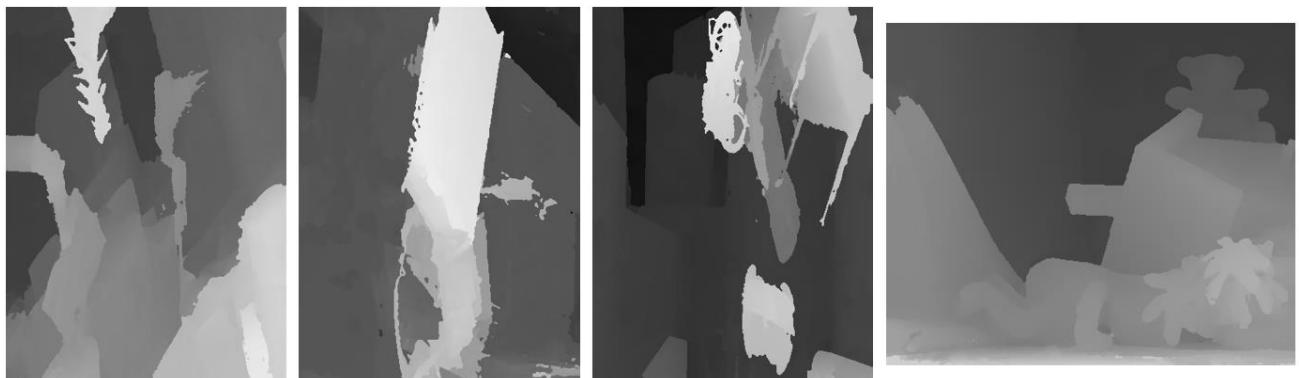


Fig. 11. Graph cut + pre-trained MCCNN results on MTK data and MiddleBury data

在 MTK 的 data 上有大約 0.24% 的進步，而在作業的 MiddleBury 測資上，則得到了 3.63% 的 bad pixel 1.0 錯誤率，比起之前不用 MCCNN 的 graph cut 解出來的 4.37% 在近一步的進步，以上圖中的 teddy 為例，桌面的部分通常之前都會解的不太好，但會成 MCCNN 後則變的連續且自然許多。

Graph cut- local expansion 跟 CVF 不一樣的地方在於，這個方法解出來的是連續的 disparity，下面以兩張 real data 為例，左圖是 CVF 解出來的結果、右圖是 LE 的結果，可以明顯得看出 CVF 有 disparity 的斷層，而 LE 的結果就相當的連續。

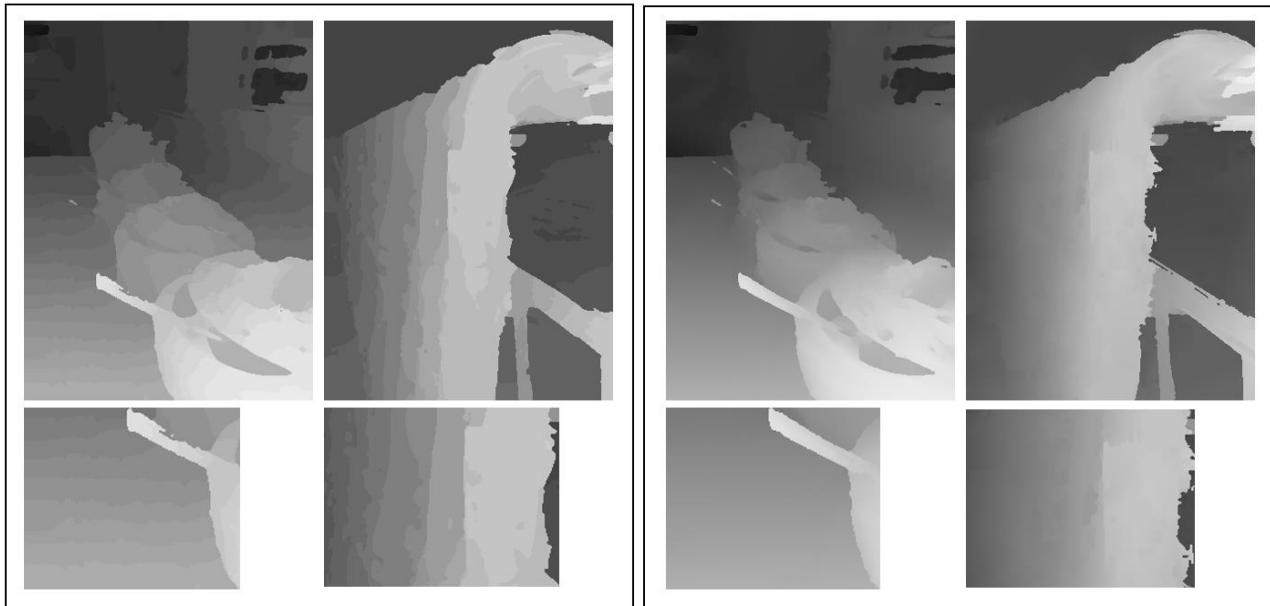


Fig. 12. CVF vs LE on real data

雖然 LE 的解會比較連續，但我們覺得 real data 上其實用 CVF 的方法就足夠了，主要原因

還是因為 graph cut 需要大量時間來跑，但結果並沒有好到能彌補所花費的時間。下圖是用 graph cut 解出來的一些 real data 的結果。

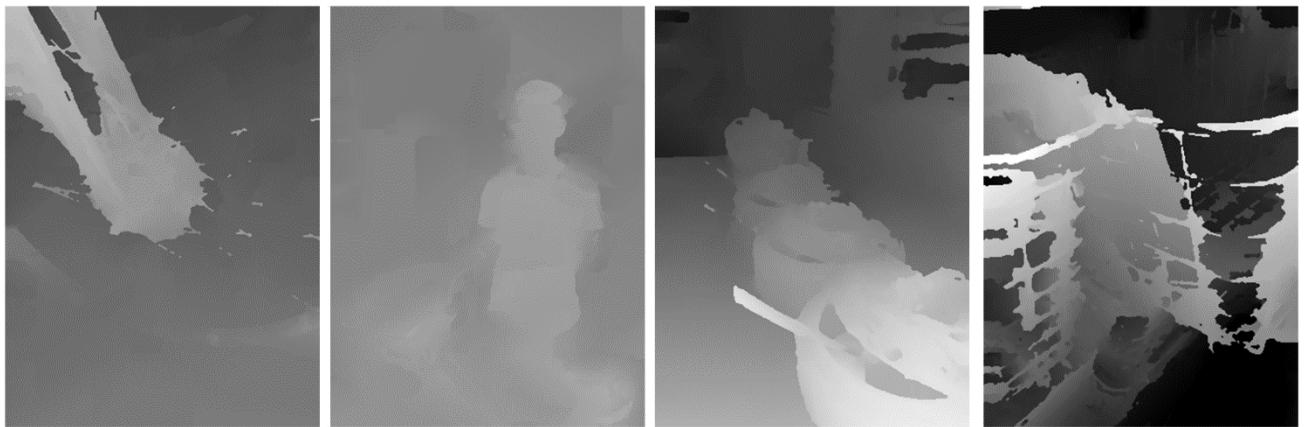


Fig. 13. Graph cut + pre-trained MCCNN results on real data

### MCCNN training

上述的方法都是用其他人 train 好的 MCCNN，在合成的結果有點難以做出突破，因此我們也嘗試了自己來 train MCCNN。MCCNN 的架構如下圖：

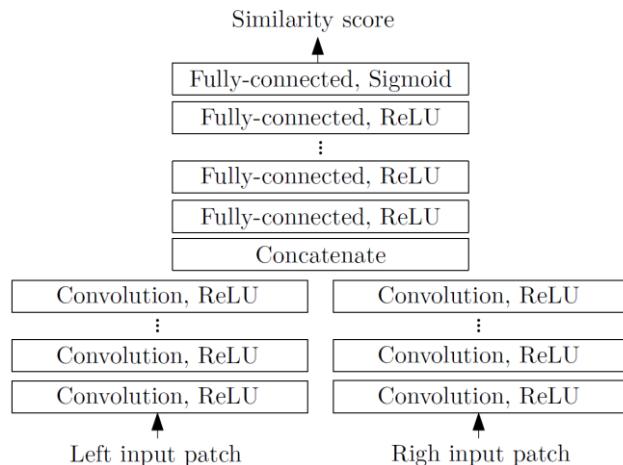


Fig. 14. MCCNN model

MCCNN 的 input 是左右兩個 patch，經過一個 shared weight CNN model，會得到兩條 feature vector，concatenate 在一起之後經過幾層的 fc，最後 output 一個 0 到 1 的 score。在 training 上，每一張左圖都會對到一張正確的右圖和錯誤的右圖，因此訓練集中會有剛好 50% 的 positive match 和 50% 的 negative match。在選擇 negative match 上，是在正確的 match 的水平方向附近幾個 pixel 擷取取得。整個 model 用 binary cross-entropy 做 training，是一個二元的分類器。

我們使用 keras 建這個 model，整體的 hyper parameter 使用跟 MCCNN 論文中 middlebury model 的參數一樣，訓練資料使用 10 張 MTK 合成圖和 10 張 MiddleBury 的 training image，

避免只用 10 張 MTK 合成圖可能造成 overfitting 的現象，最終一共會擷取出 300 多萬張 patch。

我們也有另外做 data augmentation，讓左右圖同時做 horizontal flip/vertical flip、rotation、shearing，來增加資料多樣性，避免只學到特定的 match。Validation 切 training 的 10%，在 validation loss 上升後做 early stopping。

最後大約 train 了 50 個 epoch 左右，這部分由於時間關係，我們就只有嘗試它和 CVF 的搭配，結果如下：

- MTK Synthetic avg-time: 369.5 seconds
- MTK Synthetic avg-err: 1.429%
- Middlebury V2 bad-1.0: 6.47%

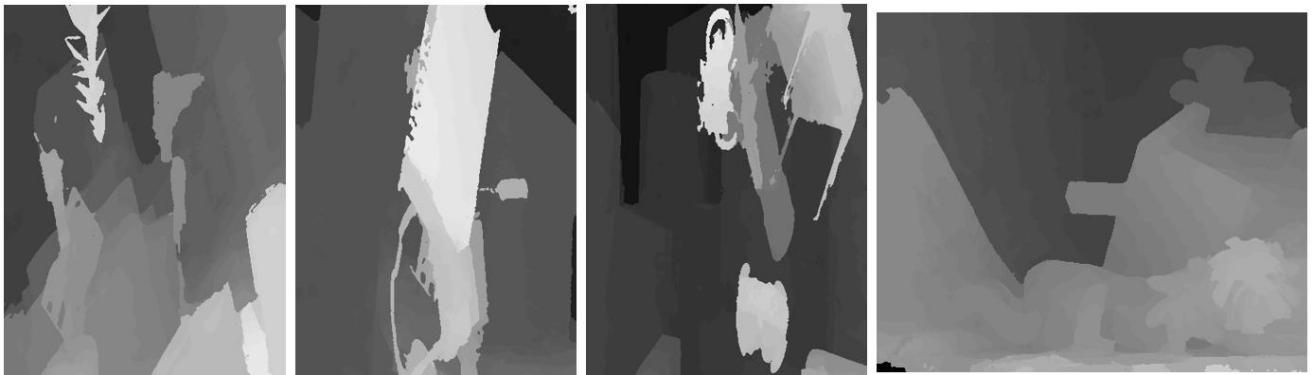


Fig. 15. CVF+ MCCNN we trained results on MTK data and MiddleBury data

雖然合成資料跟 pre-trained MCCNN 相比有約 0.5% 的進步，在作業四的 MiddleBury data 上則退步到 6.47%，雖然不比 pre-trained 的 MCCNN 來的好，但也不至於到 overfitting 的地步。為求驗證，我們也有進一步拿它來解 real data：

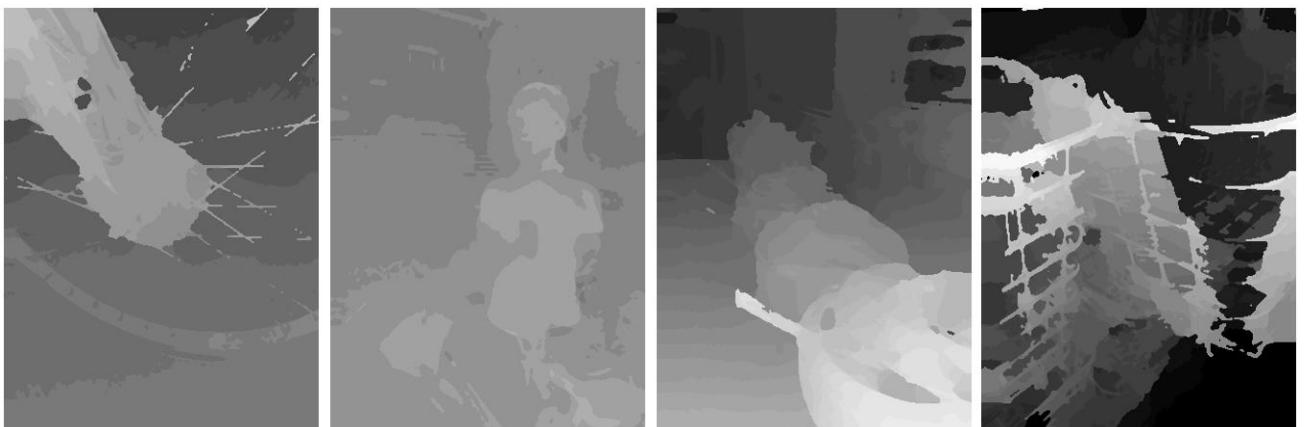


Fig. 16. CVF+ MCCNN we trained results on real data

就效果上，其實跟 pre-trained MCCNN 沒有太大差別，但如果仔細看細節的話，我們自己

train 的結果還是略遜一籌，而時間上由於是我們自己用 keras 呀出來的，inference 的時間遠不及 pre-trained model 用 chainer 的速度，大約慢了 7~8 倍，因此 real data 我們還是保留用 pre-trained MCCNN 來解。

### Max disparity estimation

我們在 maximum disparity 上，不是設定一個寫死的值，而是使用 ORB 來判定。第一步我們先將 ORB matching 中所有 y 軸相同的 match 挑出來，再從其中找出 x 方向 shift 最大的 match 作為我們的 disparity (實作上我們會在多+5 以避免 ORB 沒 match 好)。以下兩圖為例，ORB 找出的 disparity 分別為 50 和 19。

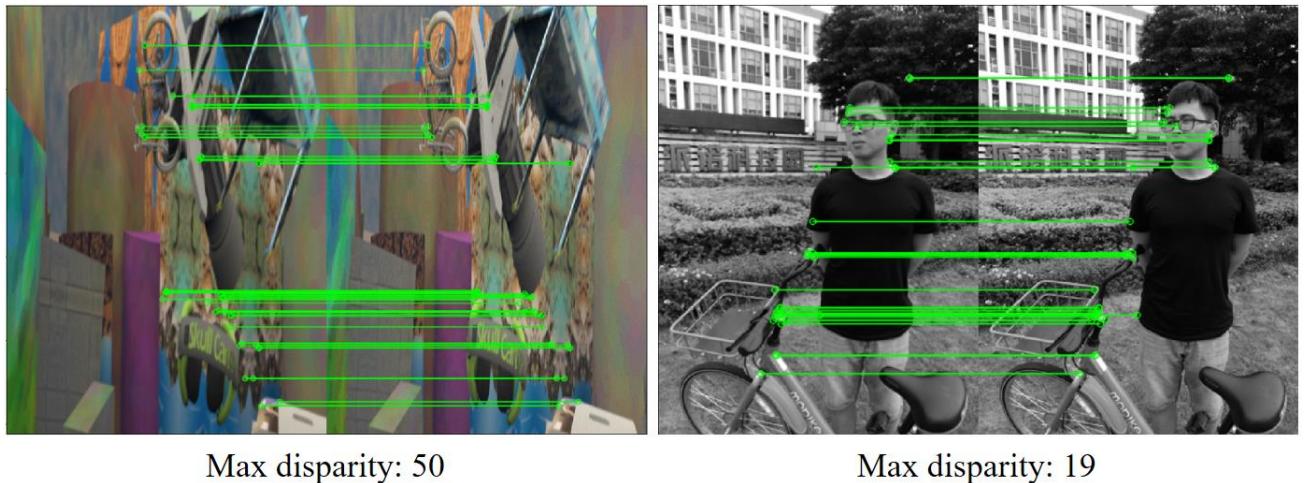


Fig. 17. Finding maximum disparity using ORB

### Real data rectification

由於 real image 會有負的 disparity 存在，我們處理的方法同樣使用 ORB 做 matching，看是否存在負的 disparity (如下圖)，若存在，我們會將右圖往左 shift 相對應的量，使得負的 disparity 不再存在，而這也是我們判斷合成和真實影響的方法。

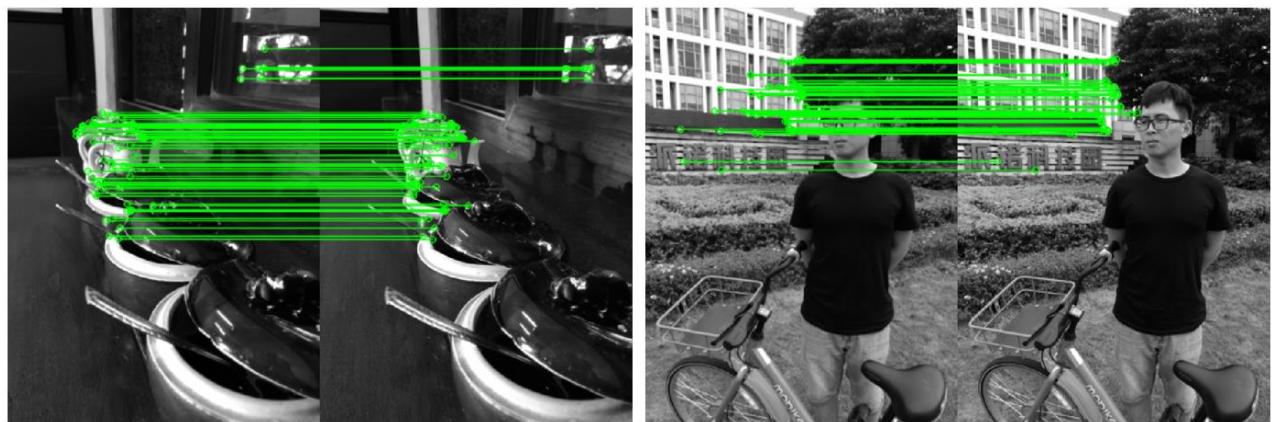


Fig. 18. Finding negative disparity using ORB

## Comparison and Robustness check

這邊將上述提到的方法做一個總結:

Method used	CVF	CVF	LE
MCCNN	Pre-trained	Trained ourselves	Pre-trained
MTK Synthetic avg-time	49 secs	369.5 secs	~8100 secs
MTK Synthetic avg-error	1.915%	1.429%	1.678%
MiddleBury V2 bad-1.0	4.44%	6.47%	3.63%
Best Qualitative real?	✓		

Table. 1. Methods performance summary

```

Architecture:          x86_64
CPU op-mode(s):       32-bit, 64-bit
Byte Order:           Little Endian
CPU(s):               8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):            1
NUMA node(s):         1
Vendor ID:            GenuineIntel
CPU family:           6
Model:                158
Model name:           Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz
Stepping:              9
CPU MHz:              3799.940
CPU max MHz:          4200.0000
CPU min MHz:          800.0000
BogoMIPS:              7200.00
Virtualization:       VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):    0-7

```

Fig. 19. Machine Spec (GPU: 1080Ti)

從上面的 table 可以看出每個方法都有它的優缺點，CVF+pre-trained MCCNN 在 runtime 上勝過其他兩個方法，而 CVF+MCCNN we trained 在 MTK 合成資料上做到最小的 error rate，LE 則在解作業四的四張圖上 error 最小。我們最終上傳的結果，合成資料我們選用 CVF+MCCNN we trained，真實資料我們選用 CVF+pre-trained MCCNN 來解。

由於助教有提到說會將影像做 transformation 再做 testing，我們在最後也額外嘗試了將合成影像稍微做點簡單的 transform 後，再拿前面的方法重新跑一遍，也算是 overfitting 的一個測試。我們的 transformation 如下圖:

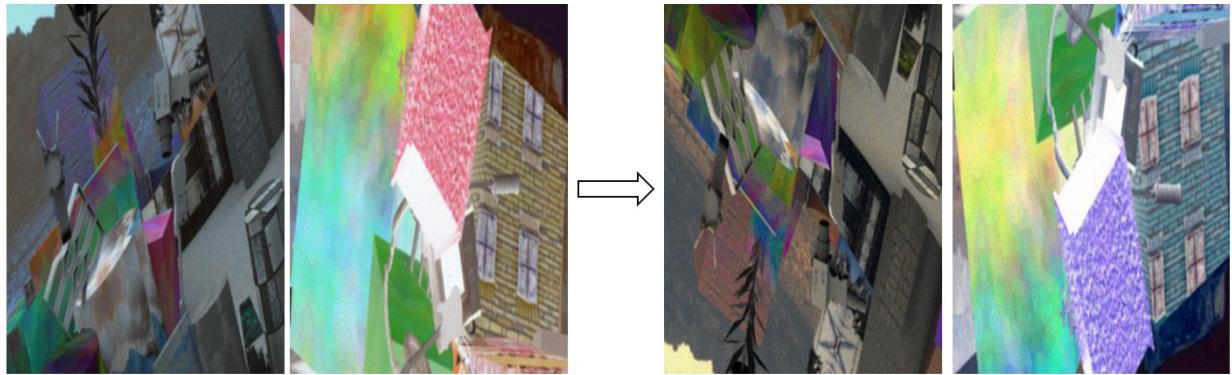


Fig. 20. Synthetic data transformation

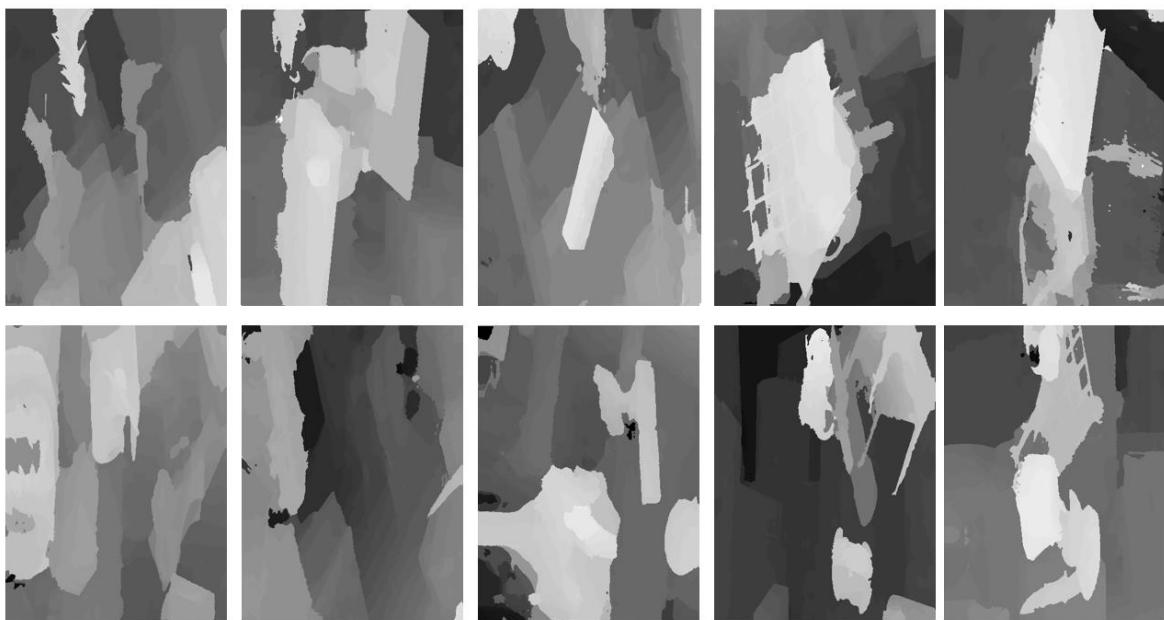
我們使用的 transform 包含將影像上下顛倒、RGB 轉 BGR、亮度和對比度的微調、以及加上微量的 noise。Ground truth 的部分再經過上下顛倒後得到。結果如下：

Method used	CVF	CVF
MCCNN	Pre-trained	Trained ourselves
Transformed image error	2.117%	2.193%

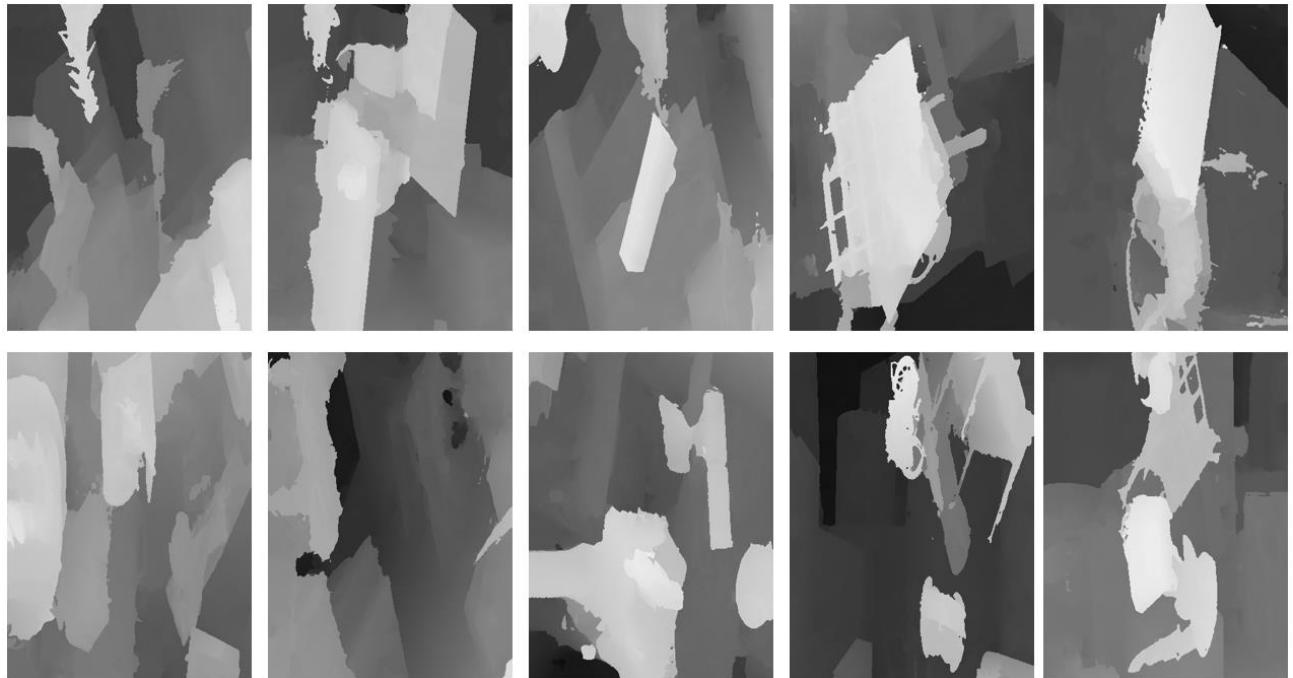
LE 由於時間太久就先不做測試，可以看到結果 error rate 在另外兩個方法都有所上升，而 pre-trained MCCNN 反過來打敗了我們自己 train 的 MCCNN，我猜測可能我們的 MCCNN 對於亮度、顏色和 noise 比 pre-trained 的要敏感，因為 pre-trained MCCNN 在 training 時有做比我們多很多的 augmentation，而我們僅做翻轉、shear、rotate 這種較簡單的 augmentation 方法。

## 2. Synthetic data result

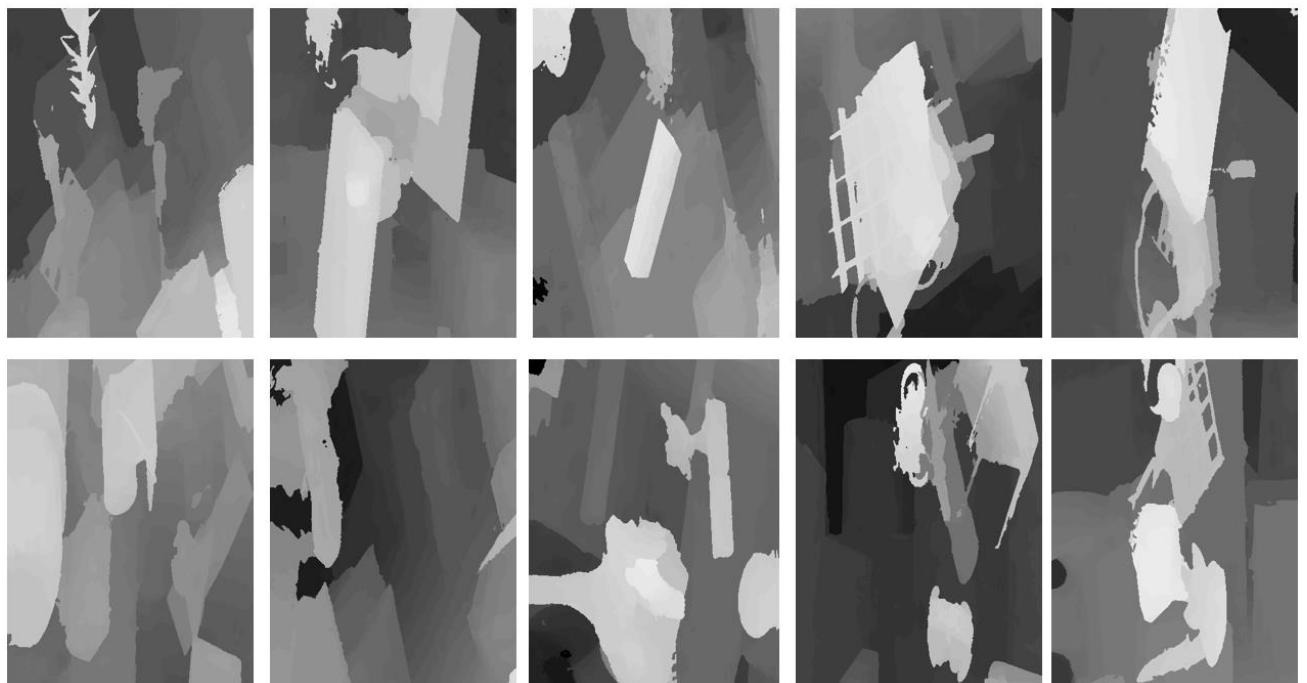
CVF+ pre-trained MCCNN:



LE+ pre-trained MCCNN:

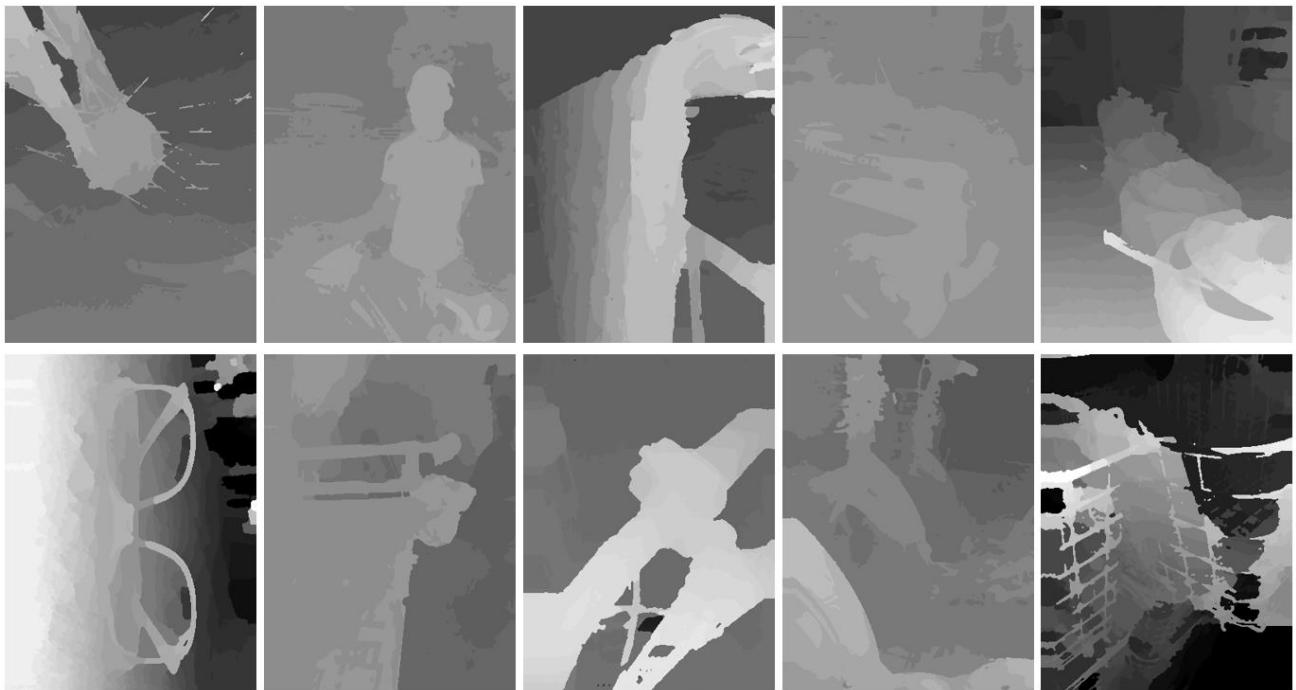


CVF+MCCNN we trained (Submitted) :

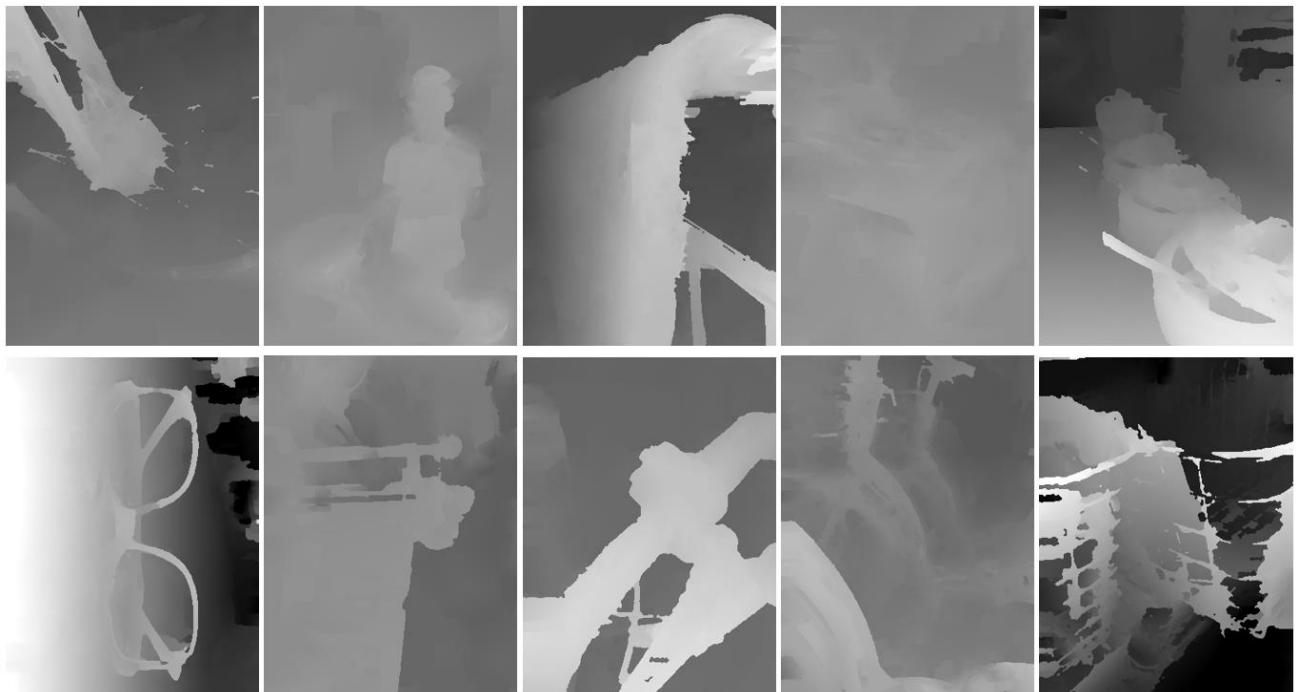


### 3. Real data results

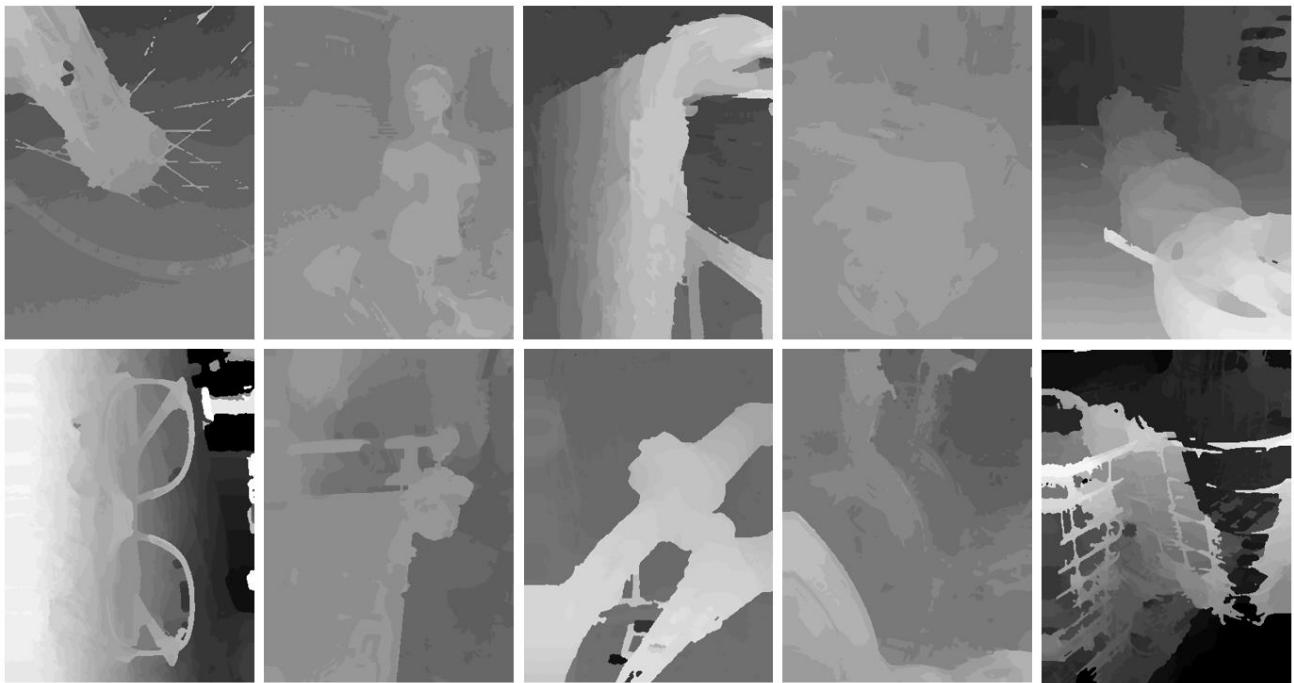
**CVF+ pre-trained MCCNN (Submitted):**



**LE+ pre-trained MCCNN:**



**CVF+MCCNN we trained:**

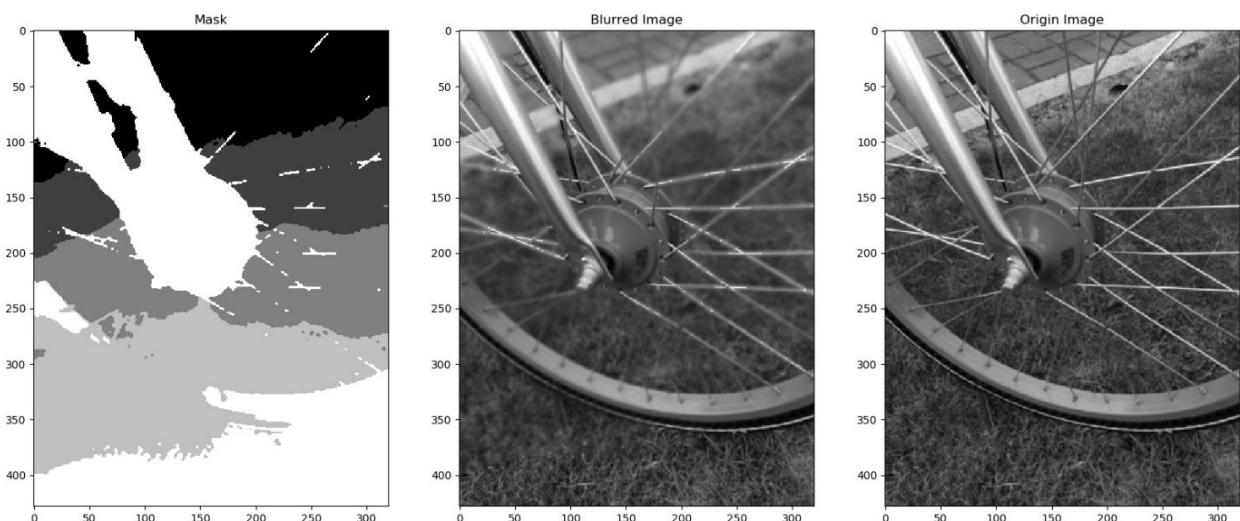


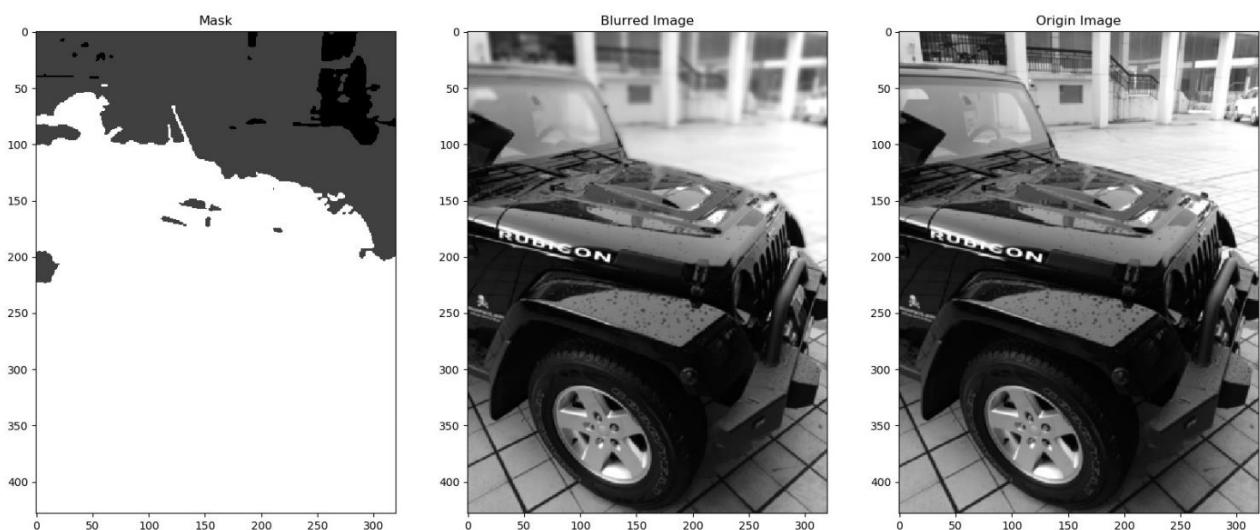
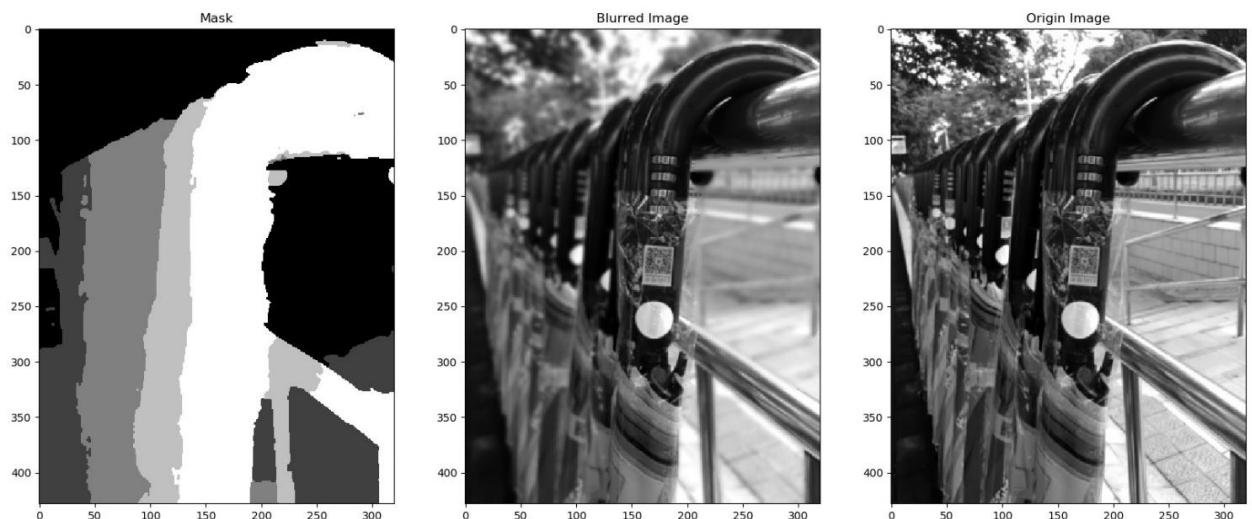
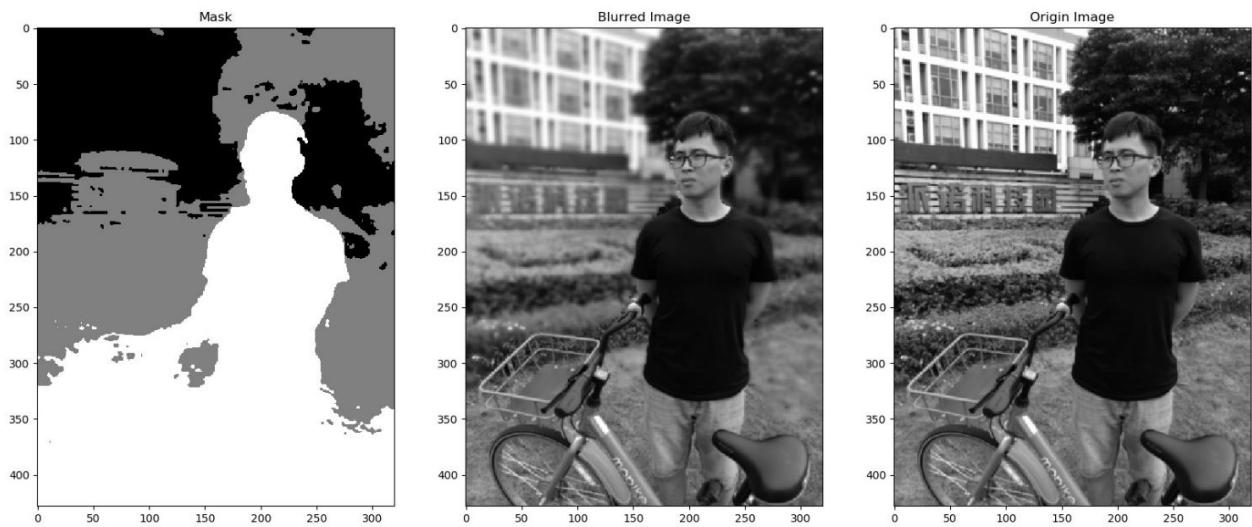
#### 4. Bokeh-Effect

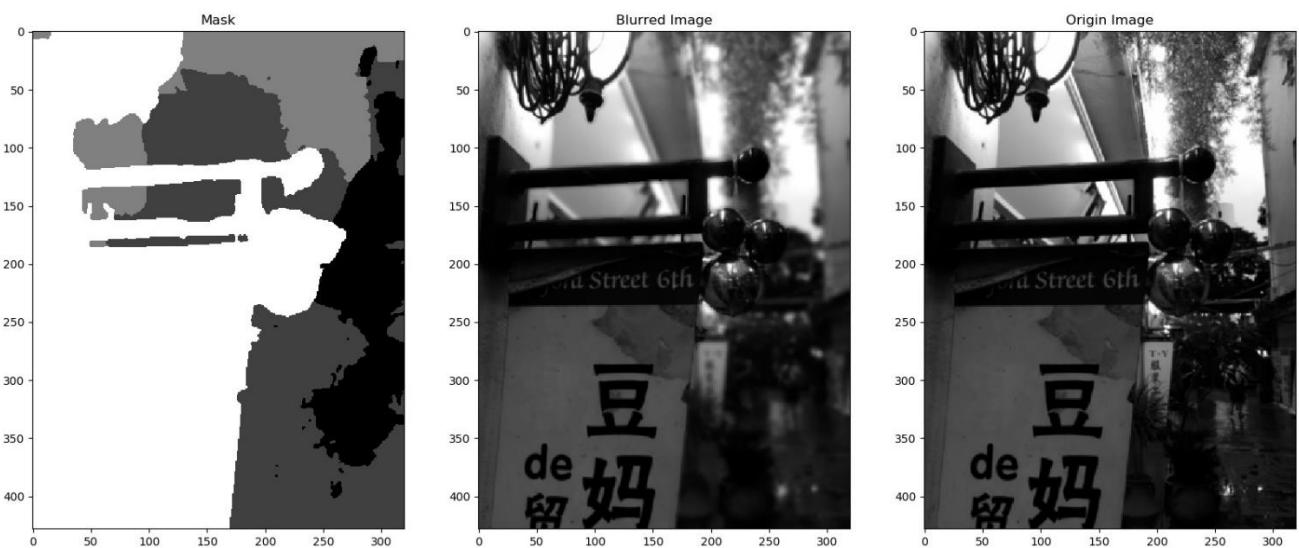
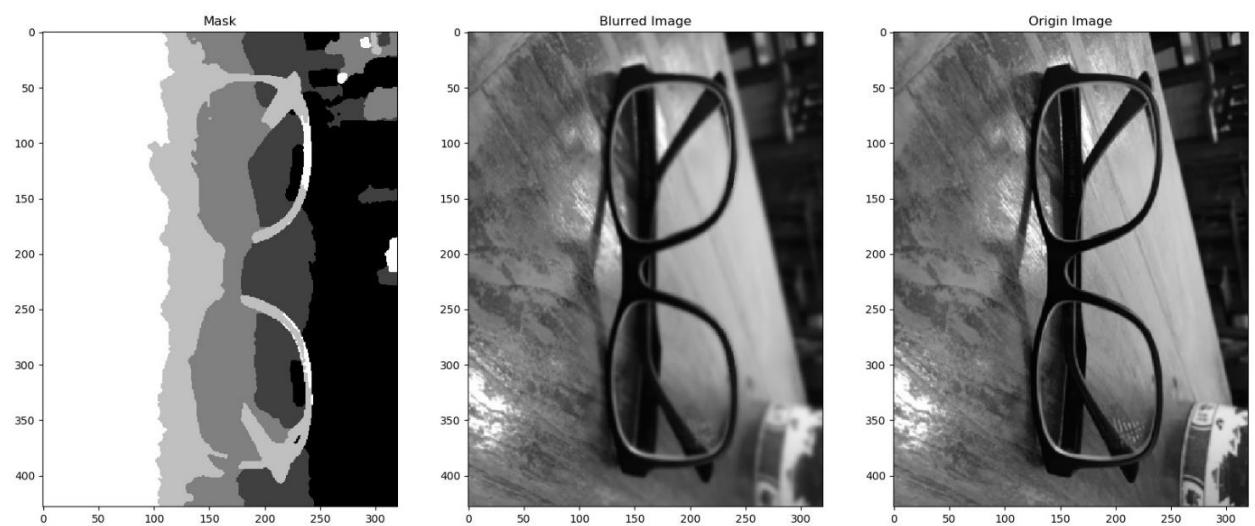
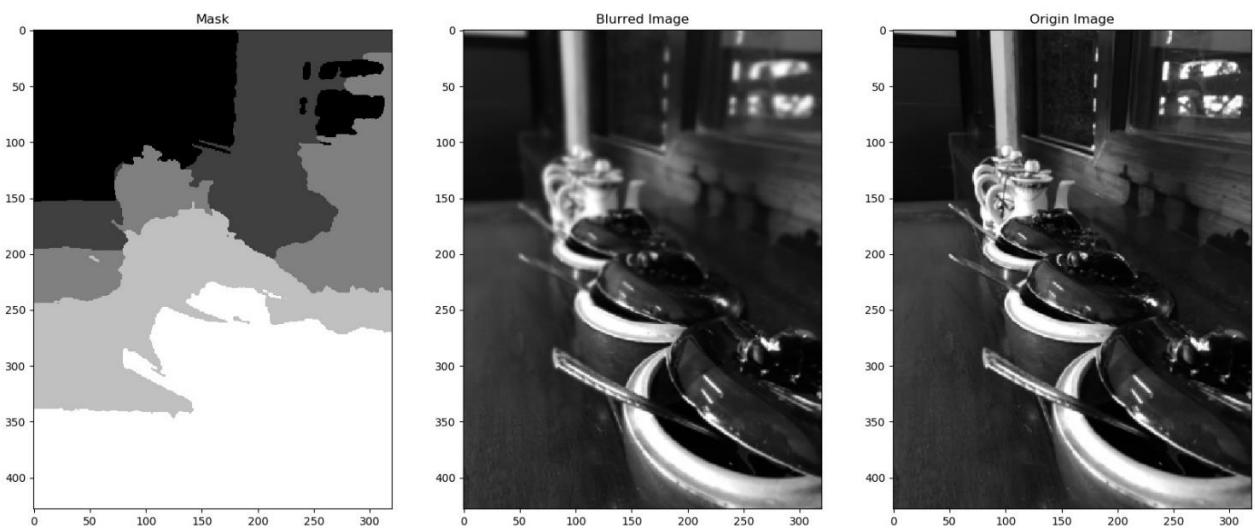
由於 Real data 的部分沒有 Ground truth 可以量化數據，因此為了驗證我們 Disparity 的好壞，我們試著將圖片分成前景與背景，將背景的部分虛化來模擬相機的散景效果。

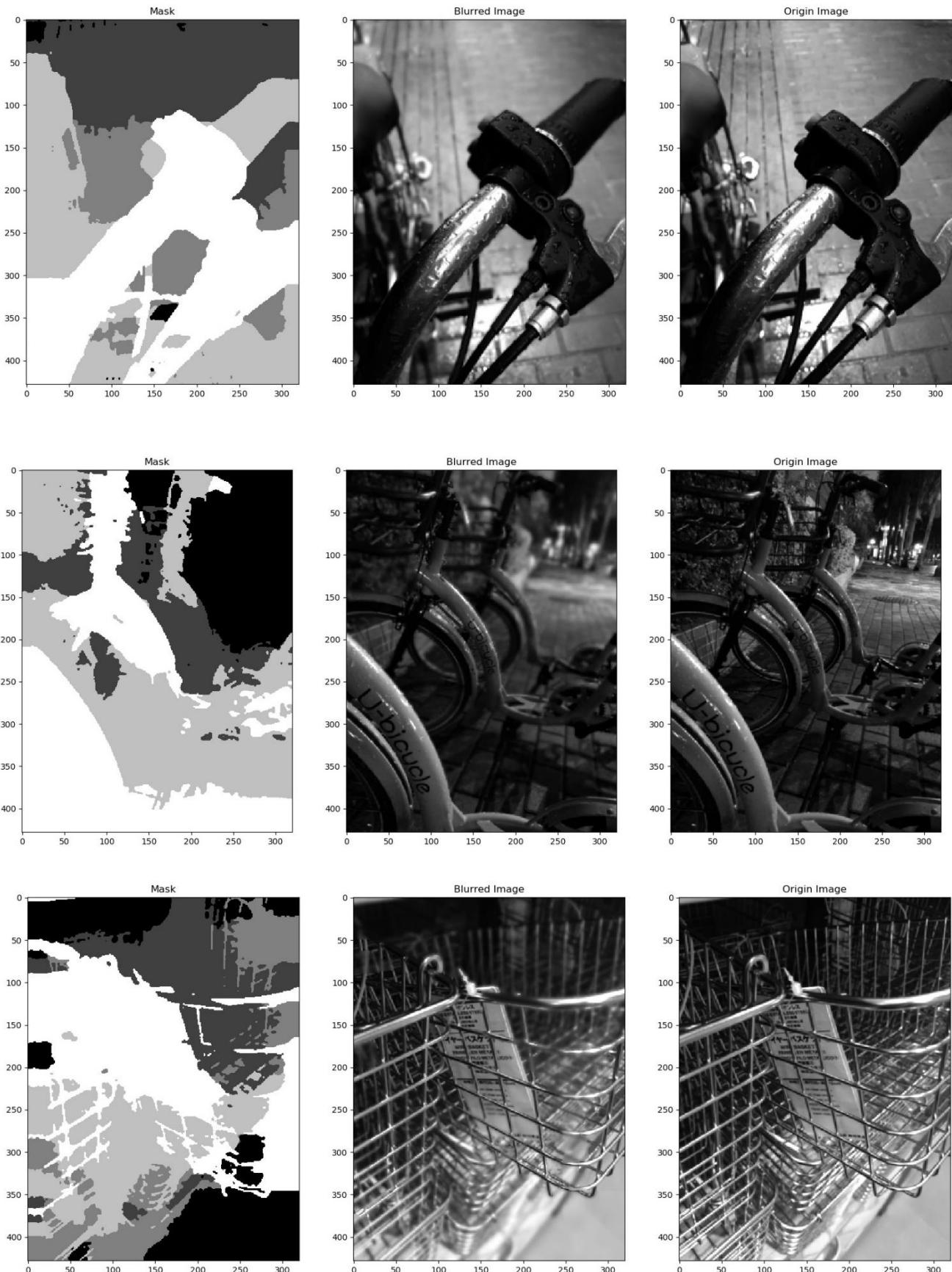
我們的方法是先將原本算出來的 disparity map 做 histogram equalization，接著把 disparity map 分成 5 個等級作為虛化的 mask，利用這 5 個等級的 mask 我們接著分別對原圖做 0 ~ 4 次的 Gaussian blur，如此我們便可得到前景清晰且背景有不同模糊等級的結果。

#### Results:









從以上 10 張 Real data 的背景虛化效果來看，除了第一張腳踏車輪胎以及最後一張網狀籃子有比較明顯的問題外，其他張照片的虛化效果都蠻不錯的，前景部分沒有被模糊掉，邊界的部分也算蠻精確的，因此我們認為我們的 Disparity map 解出來的結果算是挺不錯的。

## Code Dependencies

- Pymaxflow (<https://github.com/pmneila/PyMaxflow>)
- Cupy
- Chainer (GPU version)
- keras

## References

- [1] A. Hosni, C. Rhemann, M. Bleyer, C. Rother, and M. Gelautz. Fast cost-volume filtering for visual correspondence and beyond. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(2):504–511, 2013.
- [2] Zbontar, Jure, and Yann LeCun. "Stereo matching by training a convolutional neural network to compare image patches." *Journal of Machine Learning Research* 17.1-32 (2016): 2.
- [3] Taniai, Tatsunori, et al. "Continuous 3D label stereo matching using local expansion moves." *IEEE transactions on pattern analysis and machine intelligence* 40.11 (2018): 2725-2739.
- [4] Boykov, Yuri, Olga Veksler, and Ramin Zabih. "Fast approximate energy minimization via graph cuts." *IEEE Transactions on pattern analysis and machine intelligence* 23.11 (2001): 1222-1239.
- [5] MCCNN code reference: <https://github.com/t-taniai/mc-cnn-chainer>