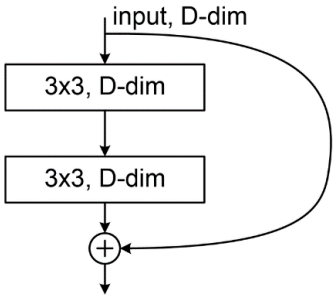
Problem1:

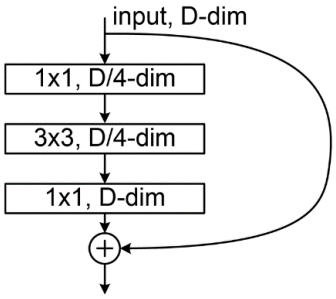
Please explain what is the residual block (two types) and give the pros and cons of each.

Residual Block 可以藉由計算結果與 input 相加作為 output 來避免梯度消失問題，減少 loss 卡在 critical points 的機會。

結構示意圖:



Basic Residual Block



Bottleneck Residual Block

1. Basic Residual Block

由兩個連續的 3x3 convolution layers 組成，output 為通過兩層 convolution layers 計算的結果與 input 之相加(shortcut connection)，這樣可以避免在遇到 critical points 時立刻停下，導致容易卡在鞍部而不是 minimum 的問題，進而繼續降低 loss。

Pros:

相較於 Bottleneck Residual Block 結構更為簡單，對於較淺的模型比較合適。

Cons:

計算效率較差、參數多，對於層數較深的模型表現不如 Bottleneck Residual Block 好。

2. Bottleneck Residual Block

由三個連續的 convolution layers 組成，分別為 1x1、3x3、1x1，同樣 output 為經過三層 convolution layers 的 data 與 input 相加。

此種結構是為了減少參數的使用以及加快計算效率，頭尾的 1x1 convolution layers 可以在保留原始數據特徵的情況下增減 data 的維度，且數據皆不受其他數據所影響，因此第一層(1x1)對數據降維來減少參數的使用，第二層(3x3)可以學習局部空間特徵，再經由第三層(1x1)將維度升回，讓其可以與 input 相加。

Pros:

減少參數及計算量，讓模型更容易訓練，訓練速度也較快，適合層數較深的模型。

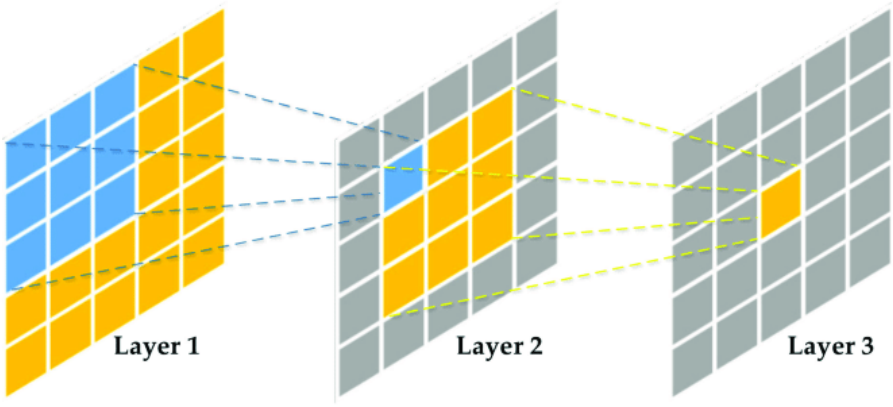
Cons:

結構較複雜，不適合較淺的模型。

Problem2:

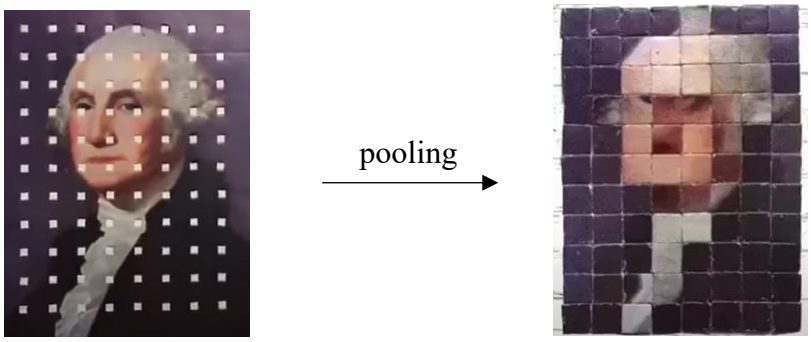
Please explain what is the receptive field and how to adjust the receptive field in the neural network.

Receptive field 是代表可以影響計算過程中一個像素的範圍，如下圖，Layer 1 藍色部分即為 Layer 2 藍色像素的 Receptive field。



它的大小決定了該模型捕捉局部/全局特徵的能力，若圖片中的主體很大，receptive field 就不適合太小，反之若主體小，就不要使用太大的 receptive field，但也要注意越大的 receptive field 也就代表越大的計算量及越差的計算效率。

另外，padding、pooling、stride 等也是影響 receptive field 的原因之一，若想要增加邊緣的 receptive field，可以適當的增加 padding 來達成，想要更快速的掃過大面積視野，就可以增加 stride 大小，或是先做 pooling 再做後續 convolution 計算，如下示意圖。



Problem3:

Please give some methods to achieve feature map upsampling. Explain them with codes and images.

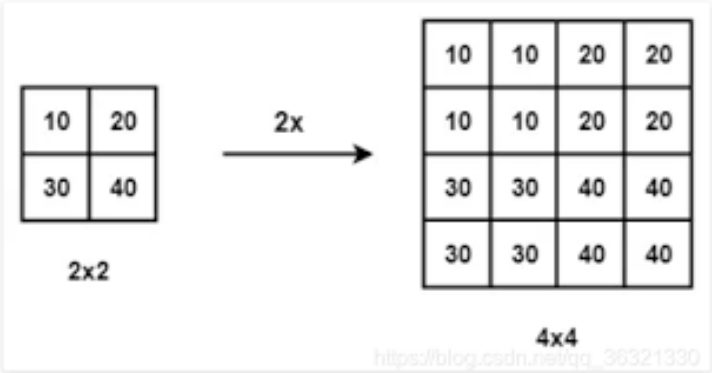
Upsampling 的功能是提升圖片的分辨率，但單純的放大圖片並沒有辦法增加圖片訊息，因此透過不同 upsampling 的方式使放大後的圖片擁有更多的訊息。以下為幾種常見的 upsampling 方式。

1. **Nearest Neighbour Interpolation**

PyTorch:

torch.nn.functional.interpolate(input, size, scale_factor, mode='nearest', align_corners, recompute_scale_factor, antialias)

直接將距離最近的原像素數據填充至放大後的像素中，這種方式簡單快速，但也無法有效增加圖片訊息。

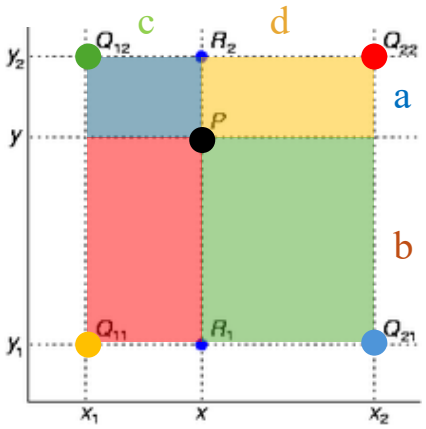


2. **Bilinear Interpolation**

PyTorch:

torch.nn.functional.interpolate(input, size, scale_factor, mode='bilinear', align_corners, recompute_scale_factor, antialias)

這種方式與 Nearest Neighbour Interpolation 類似，但是填充將該點包圍的四點之加權平均，這個方法可以有效避免鋸齒的出現。以欲填充之點與外圍任一點圍起之矩形面積與總面積的比例作為反向像素點之權重，如下圖，P 為要填充的點，Q_{11}、Q_{12}、Q_{22}、Q_{21} 為四個原先存在的點，以顏色區分各 Q 點計算權重之面積。



$$P = \frac{(a \bullet d \bullet Q_{11} + a \bullet c \bullet Q_{21} + b \bullet d \bullet Q_{12} + b \bullet c \bullet Q_{22})}{(a + b) \bullet (c + d)}$$

3. Unpooling

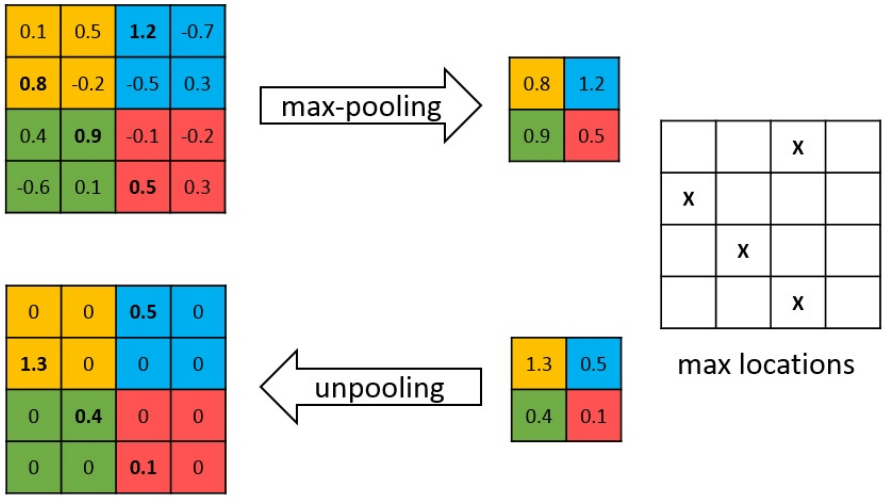
PyTorch:

torch.nn.MaxUnpool2d(kernel_size, stride, padding)

e.g.

```
pool = nn.MaxPool2d(2, stride=2, return_indices=True)
unpool = nn.MaxUnpool2d(2, stride=2)
output, indices = pool(input)
result = unpool(output, indices)
#也可以自訂 result 大小，結果會像是在 flatten 的情況下向後補 0，再排列成自定義大小。
# result = unpool(output, indices, output_size=torch.Size([1, 1, 5, 5]))
```

這個方式是先紀錄下原先做 Max pooling 時最大值的位置，在完成中間的計算後，將結果填充至紀錄的位置，其餘位置則補上 0，這樣可以最大限度保留原訊息，如下圖。



4. Transpose convolution

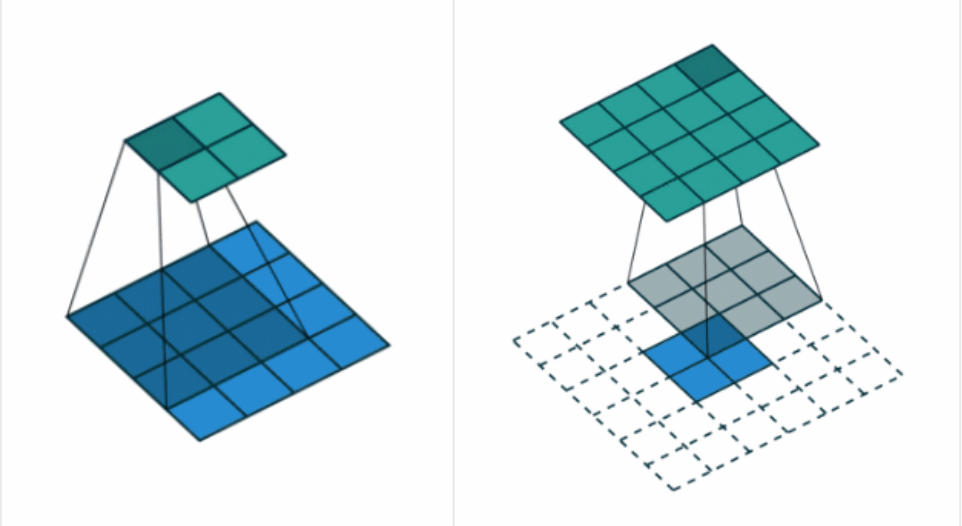
Pytorch:

torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride, padding, output_padding, groups, bias, dilation, padding_mode='zeros', device, dtype)

e.g.

```
m = nn.ConvTranspose2d(16, 33, 3, stride=2)
#也可自訂 kernel 形狀
# m = nn.ConvTranspose2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
output = m(input)
```

可以將其看成 convolution 的反向操作，如下圖，左方為 convolution 在 kernel=3、No padding、stride=1 時 input 為 4x4 所得到的 2x2 output 結果，右方就是做 transpose convolution 將 2x2 變回 4x4。



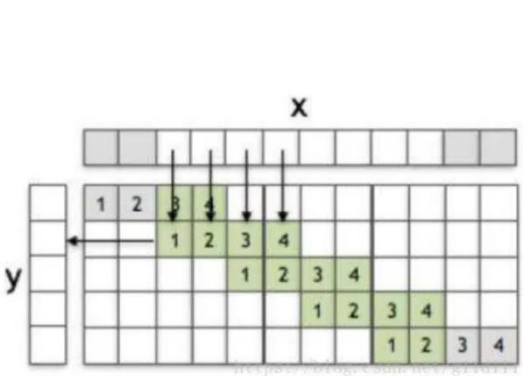
我們可以將左方的 convolution 以矩陣形式列出，以 **x** 為 4x4 input，**y** 為 2x2 output，**w** 為 3x3 convolution layer，並將 **w** 表示成 4x16 的稀疏矩陣 **C**。

$$\boldsymbol{Cx = y}, \qquad \boldsymbol{if\ C_{ij} = 0, No\ connection\ between\ x_j\ and\ y_i}$$

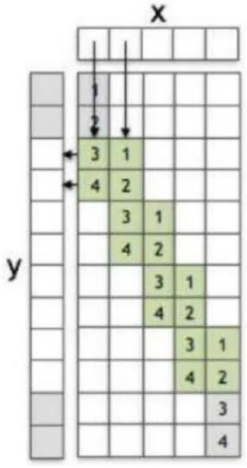
C **矩陣為**

$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 & 0 \\ 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} & 0 \\ 0 & 0 & 0 & 0 & 0 & w_{0,0} & w_{0,1} & w_{0,2} & 0 & w_{1,0} & w_{1,1} & w_{1,2} & 0 & w_{2,0} & w_{2,1} & w_{2,2} \end{pmatrix}$$

做 transpose convolution 時，該動作剛好就是將前面的 output **y** 與 $\boldsymbol{C^T}$ 做矩陣乘法，因此得到 $\boldsymbol{C^T y = x}$，對應的操作即為將 kernel 以中心做對稱，並對 **y** 做 full zero padding 後做 convolution。



Convolution



Transpose convolution

5. Sub-pixel convolution

PyTorch:

torch.nn.PixelShuffle(upscale_factor)

e.g. 放大 3 倍

pixel_shuffle = nn.PixelShuffle(3)

output = pixel_shuffle(input)

若我們希望 output 放大成 input(a x a)的 n 倍，則要生成 n^2 個 feature map(channel)，並將這 n^2 個 feature map(channel)的每個像素排列，將同個位置的像素排成 n x n 的小部分，再由 a x a 個小部分組成 na x na 的 output，如下圖。

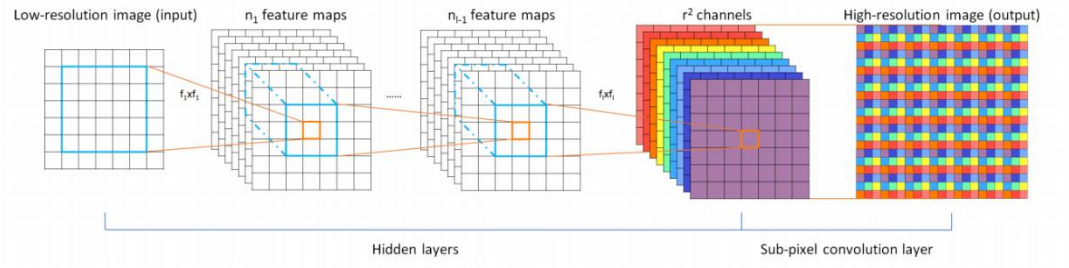


Figure 1. The proposed efficient sub-pixel convolutional neural network (ESPCN), with two convolution layers for feature maps extraction, and a sub-pixel convolution layer that aggregates the feature maps from LR space and builds the SR image in a single step.