Frame interpolation via adaptive separable convolution, adversarial training and post-processing pixel shaders

Sergio Pedri, 1618800, La Sapienza University, Rome, 2019

Abstract

This project picks up from the work of the "Video frame interpolation via adaptive separable convolution" paper [1], using a similar network with the same separable convolution operation to generate the output frames. In addition to that, the network training has been modified with the addition of a GAN [2] component, using a dedicated fully connected stack on top of a pretrained VGG-19 network [3]. The cost function has also been altered to include an L_1 factor, and a custom pixel shader has been applied to the output frames to try to shift some specific work away from the network itself, giving more room for the available weights to focus on the motion flow estimation task.

1 Separable convolution

Forward pass

This method is the core of the frame interpolation network, and it is a technique that is used to interpolate an input 2D image using an estimated 2D convolution kernel that is different for each pixel in the image.
That is, given I an $[m, n]$ image and K an $[m, n]$ array of 2D kernels, the result is given by [1]:

$$\hat{I}_{(x,y)} = K_{(x,y)} * P_{(x,y)},$$

where $P_{x,y}$ is a patch of the input image I at x, y.

In the case of the separable convolution, we replace the single kernel K with a pair of 1D kernels K^v and K^h. Each of these two kernels will have a depth d equal to the desired size of the image patches used to calculate the value of each pixel in the resulting image $\hat{I}$. In order to center each 2D kernel on the target image pixel, let's assume that d is an odd number and consider $\bar{d} = \lfloor \frac{d}{2} \rfloor$. More formally, given I an input $[m, n, c]$ image (with $c = 3$ if it's an RGB image) and K^h, K^v a pair of $[m, n, d]$ kernels, the resulting image is given by:

$$\hat{I}_{(x,y,z)} = \sum_{i=0}^{d} \sum_{j=0}^{d} I_{(x-\bar{d}+i, y-\bar{d}+j, z)} K^v_{(x,y,i)} K^h_{(x,y,j)}.$$

The formula above takes care of centering the patches $P_{(x,y,z)}$ on each source pixel $I_{(x,y,z)}$, and assumes the use of implicit padding - that is, the sum factors where the source coordinates in I lie outside the image boundaries are simply ignored.

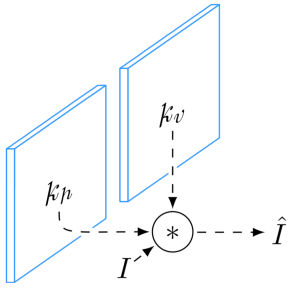

Figure 1: Separable convolution with two kernels K^h, K^v

Backward pass

Let's consider a single patch $P_{(x^*,y^*)}$ from the input image I centered around pixel $I_{(x^*,y^*)}$, the corresponding pixel $\hat{I}_{(x^*,y^*)}$ in the resulting image $\hat{I}$ and the pair of 1D kernels $K^v_{(x^*,y^*)}$ and $K^h_{(x^*,y^*)}$ used to produce that pixel (we will omit the kernels coordinates from now on to make the notation clearer). The forward pass of the separable convolution operation involves the following vectors:

$$I = \begin{bmatrix} I_{(1,1)} & I_{(1,2)} & \dots & I_{(1,n)} \\ I_{(2,1)} & I_{(2,2)} & \dots & I_{(2,n)} \\ \vdots & \vdots & \ddots & \vdots \\ I_{(m,1)} & I_{(m,2)} & \dots & I_{(m,n)} \end{bmatrix},$$

the output image $\hat{I}$ of the same shape as I and the two kernels contributing to $\hat{I}_{(x^*,y^*)}$:

$$K^v = \begin{bmatrix} K^v_1 \\ K^v_2 \\ \vdots \\ K^v_d \end{bmatrix}, K^h = \begin{bmatrix} K^h_1 & K^h_2 & \dots & K^h_d \end{bmatrix}.$$

Assuming that the pixel taken into consideration is not near the edges of the input image I, so that no padding has to be applied, the resulting pixel $\hat{I}_{(x^*,y^*)}$ is given by:

$$\sum_{i,j} \begin{bmatrix} I_{(x^*-\bar{d},y^*-\bar{d})}K_1^vK_1^h & \dots & I_{(x^*+\bar{d},y^*-\bar{d})}K_1^vK_d^h \\ I_{(x^*-\bar{d},y^*-\bar{d}+1)}K_2^vK_1^h & \dots & I_{(x^*+\bar{d},y^*-\bar{d}+1)}K_2^vK_d^h \\ \vdots & \ddots & \vdots \\ I_{(x^*-\bar{d},y^*+\bar{d})}K_d^vK_1^h & \dots & I_{(x^*+\bar{d},y^*+\bar{d})}K_d^vK_d^h \end{bmatrix}.$$

Since the input is fixed, the target gradients to calculate are just $\frac{\delta C}{\delta K^v}$ and $\frac{\delta C}{\delta K^h}$. The whole backward pass is computed in two passes: first the gradient $\frac{\delta C}{\delta K}$ with respect to the separable kernel is computed, then its components are distributed across the two original kernels K^v and K^h. The forward pass for this target pixel $\hat{I}_{(x^*,y^*)}$ is the same as a classic 2D convolution between the image patch $P_{(x^*,y^*)}$ and the kernel K, so the gradient $\frac{\delta C}{\delta K}$ is given by:

$$P_{(x^*,y^*)} * \frac{\delta C}{\delta \hat{I}_{(x^*,y^*)}},$$

which is equivalent to a simple elementwise product, as the output gradient $\frac{\delta C}{\delta \hat{I}_{(x^*,y^*)}}$ is a single pixel. As the input and output images have an arbitrary number of channels (generally 3, for RGB images), which are all processed using the same kernel K, the backpropagated gradient for all channels is given by:

$$\frac{\delta C}{\delta K_{(x^*,y^*)}} = \sum_{z=0}^{c} P_{(x^*,y^*,z)} \odot \frac{\delta C}{\delta \hat{I}_{(x^*,y^*,z)}}.$$

The last step required is to extract the two components $\frac{\delta C}{\delta K^v}$ and $\frac{\delta C}{\delta K^h}$ from the partial gradient. Each component of K_i^v influences the i-th row of $\frac{\delta C}{\delta K}$, while each component of K_j^h influences the j-th columns of $\frac{\delta C}{\delta K}$, proportionally to the corresponding element of the other vector for each coordinate. The gradients for the two vectors are therefore given by:

$$\frac{\delta C}{\delta K^v} = \begin{bmatrix} \sum_j \frac{\delta C}{\delta K}_{(1,j)} K_j^h \\ \vdots \\ \sum_j \frac{\delta C}{\delta K}_{(d,j)} K_j^h \end{bmatrix},$$

$$\frac{\delta C}{\delta K^h} = \begin{bmatrix} \sum_i \frac{\delta C}{\delta K}_{(i,1)} K_i^v & \dots & \sum_i \frac{\delta C}{\delta K}_{(i,d)} K_i^v \end{bmatrix}.$$

2 Network architecture

Frame interpolation

The first part of the network [2] is responsible for the actual frame interpolation, ie. it's the part that takes pairs of consecutive frames as inputs, and produces the interpolated frame as output. This is done through an encoder-decoder with skip connections, that learns the optical flow estimation for pairs of frames, and then an output section where the 4 separable kernels are computed and applied. As shown in the original paper, using separable kernels reduces the number of operations per output pixels from $O(n^2)$ to $O(n)$ for each output pixel.

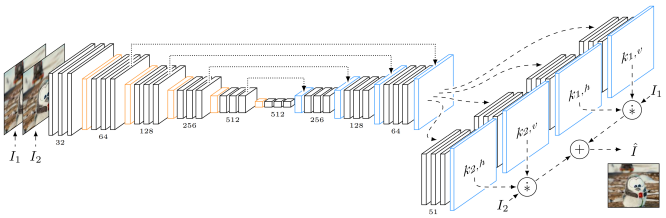

Figure 2: Encoder-decoder and separable convolution

Perceptual pass

The frame interpolation network leverages a pretrained VGG19 network [3] during training, that is used to extract the image features for each generated frame, in order to apply the perceptual loss [4] to the output.
This is done by propagating each generated frame and ground truth frame through the network, then extracting the features of a specific convolution layer and applying the loss function to the difference between those features.

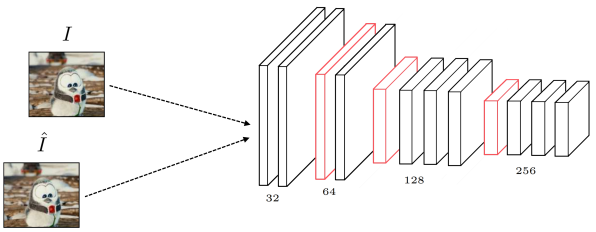

Figure 3: VGG19 up to `conv4_4`

Adversarial network

The final building block of the full training network is an adversarial network [4], which is stacked on top of the pretrained VGG19 network. It is composed of a series of custom trained convolution layers, with a final fully connected stack that reduces the number of output features down to one, which is used to produce the binary classification result for the discriminator network.

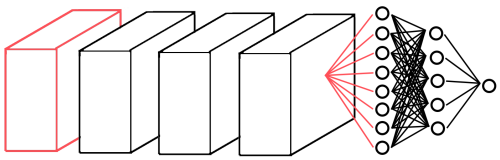

Figure 4: Convolution and fully connected stack

Putting it all together

All of these building blocks come together during training, where we end up with the following architecture [5]:

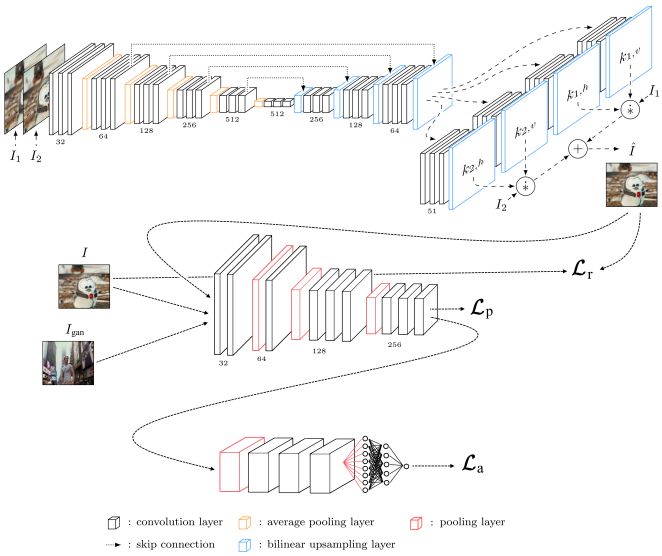

Figure 5: The final training architecture

3 Loss function

Given two input frames I_0 and I_1, the network is trained to generate the intermediate frame $\hat{I}$ using the following loss function:

$$\mathcal{L} = \mathcal{L}_p + \gamma_r \mathcal{L}_r + \gamma_a \mathcal{L}_a,$$

which is a linear combination of four different losses. The coefficients that were found to be effective when training the network were $\gamma_r = 0.4$ and $\gamma_a = 0.8$.

Perceptual loss $\mathcal{L}_p$, calculates the accuracy of the model based on high-level features of the generated frame:

$$\mathcal{L}_p = \|\Phi(\hat{I}) - \Phi(I)\|^2,$$

where $\Phi(x)$ is the output of the `relu4_4` layer of the VGG19 network (before the last pooling layer).

Reconstruction loss $\mathcal{L}_r$, which is the L_1 loss on the intermediate frame and it is used to help the model generate accurate colors in the generated frames, as the perceptual loss tends to prioritize sharpness and accurate image features over exact color reproduction:

$$\mathcal{L}_r = \|\hat{I} - I\|^1.$$

Adversarial loss $\mathcal{L}_a$, used to minimize artifacts in the generated frames, which are often small enough not to be perfectly addressed by the perceptual loss, while still being visible when looking at a high resolution frame produced by the model:

$$\mathcal{L}_a = -log(\sigma(D(\hat{I}))),$$

where D is the output (a single value in $[0, 1]$) of the discriminator network on a generated frame.

4 Dataset

The dataset was generated from a collection of around 2600 popular movies and TV series episodes, with a resolution of either 1920×1080 (FullHD) or 3840×2160 (UHD). Each video contributed to the dataset with a series of 80 1s-long clips, evenly spaced across the original video. The frames extracted from every clip were resized to 1280×720, and those with not enough difference with the following ones or with not enough variance were excluded. Furthermore, we used a threshold on the maximum L_2 difference between frame pairs to identify scene changes, and limited to 3 the maximum number of frames from a single scene sequence, to avoid long series of frames with a similar content. The resulting dataset contained 285.939 different frames, for a total of 95.313 different samples, each one with the two frames I_0, I_1 and the ground truth.

Data augmentation

As the model was trained using 160×160 images, it was possible to expand the dataset by clipping a large number of different samples from the available frames [6]. The dataset augmentation pipeline is composed of the following operations:

Resize: as clipping small images from a large 1280×720 frame would cause the resulting samples to never show large visual structures (eg. a city skyline), each group of frames was first randomly resized (with bilinear sampling, keeping the aspect ratio) to a target resolution between 1280×720 and 327×184.

Crop and flow: each resized image was then clipped to 160×160, and a random offset in the $[0, 12]$ range along both the X and Y axes was then applied to the frames I_0 and I_1. This allowed to increase the flow intensity in the input frames, even in scenes with small spatial movement.

Flip: the preprocessed frames were then randomly flipped either vertically or horizontally, to make the dataset more symmetric.

Sorting: the resulting frames I_0 and I_1 then had a 50% chance of being swapped (effectively reversing the time axis for the frames group) to further prevent the model from being biased.

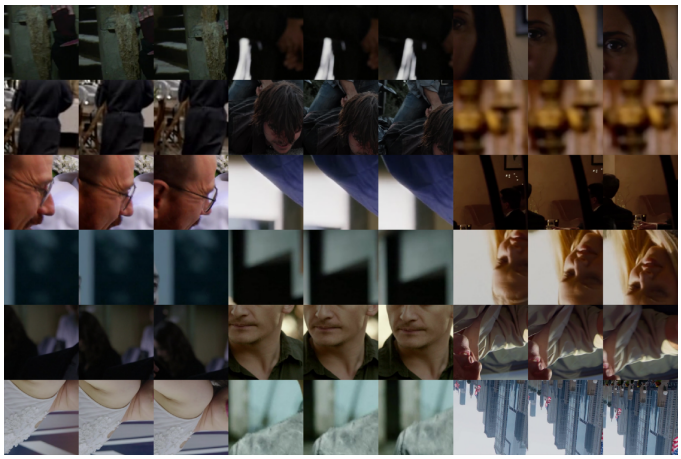

Figure 6: Sample training batches

5 Inference

During the inference pass, an additional pixel shader is applied on the interpolated frames to improve the output quality. This is done both to reduce flickering in specific cases where the network doesn't produce optimal results (eg. when there is a moving object behind an overlayed, static text) and to improve temporal consistency across frames. As mentioned before, the reason why this step is performed through a pixel shader and not just by modifying the network structure is to avoid overfitting the network just on these specific cases, which could reduce the overall quality of the outputs in more general situations.
The pixel shader works by taking two consecutive frames I_1 and I_2 as input, along with the generated frame $\hat{y}$ and a blend map $\mathcal{B}(\hat{y})$ [7], which is the result of a Gaussian blur applied to a processed difference between the two input frames, and applying the following formula:

$$\mathcal{B}(\hat{I})_{(x,y)} = \frac{\sum_{c=0}^{3} \|I_1(x,y,c) - I_2(x,y,c)\|}{3} * K_{(x,y)},$$

where K is a Gaussian 2D kernel of a given radius. Once this blend map is computed, it is passed as an additional parameter to the pixel shader, that uses it to compute the right value for each output pixel.

Algorithm 1: pixel shader for $\hat{I}$

if $\mathcal{B}(\hat{I})_{(x,y)} < 6$ **then**
- r $\leftarrow \frac{I_{1(x,y,0)}+I_{2(x,y,0)}}{2}$;
- b $\leftarrow \frac{I_{1(x,y,1)}+I_{2(x,y,1)}}{2}$;
- g $\leftarrow \frac{I_{1(x,y,2)}+I_{2(x,y,2)}}{2}$;
- d $\leftarrow$ `clamp`$(\mathcal{B}(\hat{I})_{(x,y)}, 2, 6)$ [7];
- f $\leftarrow$ `inverse_lerp`$(d, 2, 6)$ [8];
- r $\leftarrow$ `lerp`$(f, r, \hat{I}_{(x,y,0)})$ [8];
- g $\leftarrow$ `lerp`$(f, g, \hat{I}_{(x,y,1)})$;
- b $\leftarrow$ `lerp`$(f, b, \hat{I}_{(x,y,2)})$;
- **return** (r,g,b);

else
- **return** $\hat{I}_{(x,y)}$;

The effect of the pixel shader is more easily noticeable on static areas across frames with a similar color: applying the pixel shader makes the output color more accurate and reduces the flickering that can be produced as a result of small luminance differences in the generated pixels, that can be difficult to catch for the loss function.

Figure 7: Pre-processing for the pixel shader buffers

Luminosity loss

In order to improve the temporal consistency across frames for areas with a uniform color and to reduce the flickering effect there, other possible solutions have been experimented with as well. For instance, adding a fourth loss function component $\mathcal{L}_l$ that is calculated on the luminance

difference between each pixel from the generated frame and the ground truth frame, according to the following formula:

$$\mathcal{L}_l = \|\Lambda(\hat{I}) - \Lambda(I)\|^2,$$

where

$$\Lambda(I)_{(x,y)} = 0.299 I_{(x,y,0)} + 0.587 I_{(x,y,1)} + 0.114 I_{(x,y,2)}.$$

This $\mathcal{L}_l$ loss consists of the L_2 loss on the perceived brightness [9] of the two images, and takes into account the way the human eye reacts to different wavelengths (as the eye sensibility is not the same across the RGB spectrum). After some testing this didn't really seem to contribute much to the generated pixels, so it was set aside.

6 Results

The training was done on a single GTX1080, and it was stopped after around 70 hours: considering the single, less powerful GPU, this network was trained for a comparable amount of time with respect to the original sepconv network. The network was initialized with appropriate default values [5], and it was trained using the Adam optimizer [6] with default parameters ($\beta_1 = 0.9$, $\beta_2 = 0.999$) and a learning rate initially equal to 0.0006, and then using an exponential decay with a value of 0.99. A similar setup was used for the discriminator optimizer, with the main difference being a slightly smaller initial learning rate value of 0.0001. The batch size had to be set to just 6 samples, due to memory constraints (a single GTX1080 only has 8GB of VRAM).

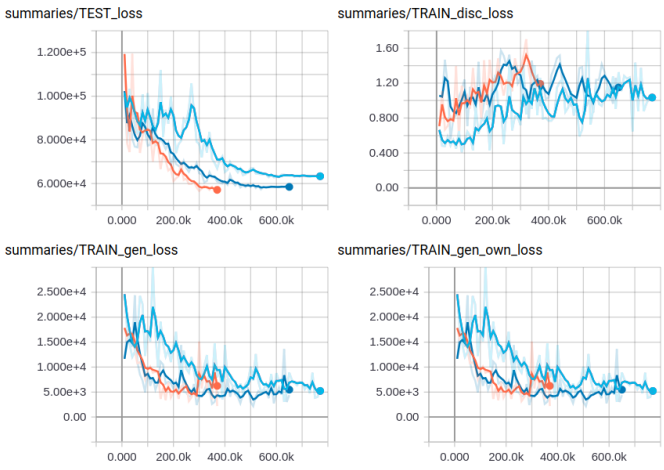

Figure 8: Combined losses for the 3 best models

The final model that was picked among the best ones [8] is the one in dark blue, as it produced the best scores for both the generative and test losses, and resulted in the best looking images during manual evaluation. In total, over 100 different models have been trained, while testing different network configurations and settings for the customizable parameters involved in the setup.

Frame comparisons

Here are some comparisons with the original sepconv network [9]. All the images from the new network that are shown in this section do not have the pixel shader applied to them, in order to better illustrate the raw output of the network itself. For each frames pair, the frame on the left is produced by the sepconv network, and the one on the right by the network from this project.

Figure 9: Frame comparisons

In the first pair, the frame produced by the new network doesn't show the artifacts that are visible in the top left corner of the other frame, and has a less pronounced ghosting effect near the rear propulsors of the X-Wing. In the second pair, the face of the character at the center is more detailed than in the frame produced by sepconv, and all the artifacts are replaced by a reduced ghosting effect (especially noticeable over the person on the left). Also, the hands and arms of the character at the center are much more defined. In the third pair, notice the lack of artifacts and more accurate interpolation of the two sparks at the bottom right of the frame. In the last pair, the sepconv

network produces particularly noticeable artifacts, which are made more obvious by the contrast with the horizontal lines on the road.
The following figure [10] highlights a series of details from the previous sequence of frames. For each pair of details, the one on the left was generated by the sepconv network, and the one on the right by the new network. It can be seen how the sepconv network tends to produce pretty noticeable artifacts, as mentioned before, which are completely absent in the frames produced by the new network.

Figure 10: Specific details from the comparisons

Conclusions

In general, the main difference between the two networks is the tendency of the sepconv network to produce noticeable artifacts and distorsions, while the network from this project tends to produce results that are more similar to a motion blur, therefore being more natural looking in the context of frame interpolation performed on movies or TV series (which were also the source of the dataset). This can be explained by considering the presence of an additional adversarial training component, as those artifacts would be particularly noticeable by an external discriminator even if they didn't contribute much to the actual perceptual loss, which is how they probably managed to get past the training of the original sepconv network.

7 Future work

In order to capture wide movements, especially in high resolution images, where the distance travelled by pixels from one frame to another can be much larger than values observed when working with low resolutions, large convolution kernels are required. But, even with the memory optimizations brought by the use of separable kernels, working with high resolution images means that any increase of the size of the adaptive kernels can require a large amount of memory. Another intuition behind the following extension is that as the distance between an output pixel $\hat{I}_{(x,y,z)}$ and pixels in its source patch $P_{(x,y,z)}$ increases, the relative density of information required from each pixel decreases, so just using a larger, dense kernels would not be cost-effective.

Composite dilation

To expand the area covered by the adaptive kernels while also increasing the ratio between the area of each image patch and the number of kernels required to process it, we define an extension of the adaptive separable convolution operation, which can naturally be reduced to the operation described above if needed. Let's define d^{in} to be the length of the dense, inner kernel, and d^{out} to be the depth of the outer, dilated kernel. In order to keep the composite kernel centered on any given pixel in the source image, d_{in} and d_{out} must be an odd and even number respectively. As a result, the size of each image patch $P_{(x,y)}$ will be equal to $(d_{\text{in}} + 2d_{\text{out}})^2$ pixels, but as each kernel is separable and decomposed into two vectors K^v and K^h, the actual number of parameters to train will be equal to $2(d_{\text{in}}+d_{\text{out}})$. In contrast with a fully dense adaptive kernel with the same parameters, the difference in covered area is given by the following expression:

$$\begin{aligned}\Delta A &= A_{\text{dilated}} - A_{\text{dense}} \\ &= (d_{\text{in}} + 2d_{\text{out}})^2 - (d_{\text{in}} + d_{\text{out}})^2 = 2d_{\text{in}}d_{\text{out}} + 3d_{\text{out}}^2.\end{aligned}$$

This means that the average amount of covered area for each parameter increases from $\frac{d}{2}$ to $\frac{d_{\text{in}}^2}{2d} + 2d_{\text{out}}$. It can also be observed that the dense convolution is just a special case of the composite convolution, where $d_{\text{out}} = 0$.
Another important detail about the composite dilation is the pattern of the dilated section of the convolution kernel, which alternates two different configurations to reduce to a minimum the unavoidable blindspots of the kernel caused by the dilation itself [11]. Both patterns have 8 fixed pixels near their corners and the center of each edge, then the first one distributes the remaining pixels next to the corners, and the second one at the center of each edge.

The forward and backward passes are similar to the ones described in the previous sections, with the additional logic to handle the dilated part of each image pattern. The whole operation can be thought of as a normal local convolution between an image patch $P_{(x,y,z)}$ and a

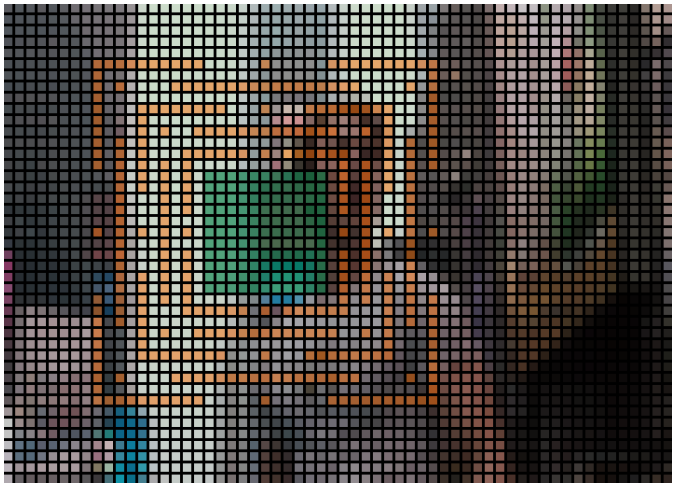

Figure 11: Composite dilation, $d_{\text{in}} = 11$ and $d_{\text{out}} = 10$

separable kernel K, with the difference that the patch is not necessarily a contiguous area of the input image, but it combines a central, dense section of the image of area d_{in}^2 and an outer square ring of area $d^2 - d_{\text{in}}^2$ that is built by aggregating the sparse pixels at the right coordinates, according to the pattern of each dilated convolution layer. The forward expression is therefore described by the following formula:

$$\hat{I}_{(x,y,z)} = \sum_{i=0}^{d_{\text{in}}} \sum_{j=0}^{d_{\text{in}}} I_{(x-\bar{d}_{\text{in}}+i, y-\bar{d}_{\text{in}}+j, z)} K^v_{(x,y,i)} K^h_{(x,y,j)}$$
$$+ \sum_{i=0}^{d_{\text{out}}} \sum_{j=0}^{d_{\text{out}}} I_{(x+\mathcal{M}(i), y+\mathcal{M}(j), z)} K^v_{(x,y,i+d_{\text{in}})} K^h_{(x,y,j+d_{\text{in}})}$$

Where $\mathcal{M}$ is a mapping function that returns the correct offset for the source pixel in the input frame, according to the current relative offset in the operation. Notice how the separable kernels are not indexed in any particular way during this dilated convolution: in order for it to be seen as a more general case of the classic separable convolution, the kernel weights used for the dense convolution are just stored in the kernel elements in the $[0, d_{\text{in}})$ range, while the weights in the following positions, if present, are used for the dilated part of the convolution. As mentioned above, the backward pass doesn't need to be modified, other than just applying an inverse $\mathcal{M}$ function to make sure to sample the input pixels from their correct, offset coordinates in the input frame.

References

[1] Simon Niklaus, Long Mai, Feng Liu. *Video Frame Interpolation via Adaptive Separable Convolution.* arXiv/1708.01692, 2017 1

[2] Ian J. Goodfellow, Jean Pouget-Abadie,Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. *Generative Adversarial Nets.* arXiv/1406.2661, 2014 1

[3] Karen Simonyan, Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition.* arXiv/1409.1556, 2014 1

[4] Justin Johnson, Alexandre Alahi, Li Fei-Fei. *Perceptual Losses for Real-Time Style Transfer and Super-Resolution.* arXiv/1603.08155, 2016 2

[5] A. Aghajanyan. *Convolution aware initialization.* arXiv/1702.06295, 2017 5

[6] Diederik P. Kingma, Jimmy Lei Ba. *Adam: a method for stochastic optimization.* arXiv/1412.6980, 2014 5

[7] en.wikipedia.org/wiki/Clamping_(graphics) 4

[8] en.wikipedia.org/wiki/Linear_interpolation 4

[9] en.wikipedia.org/wiki/Luma_(video) 5