Deep Learning Methods for Streaming Image Reconstruction in Fixed-camera Settings

Johan Förberg

December 13, 2017

Master's Thesis
Centre for Mathematical Sciences at Lund University
Supervisor: Prof. Anders Heyden, Lund University
Deputy Supervisor: Björn Ardö, Axis Communications AB

Abstract

A streaming video reconstruction system is described and implemented as a convolutional neural network. The system performs combined 2x super-resolution and H.264 artefacts removal with a processing speed of about 6 frames per second at 1920×1080 output resolution on current workstation-grade hardware. In 4x super-resolution mode, the system can output 3840×2160 video at a similar rate. The base system provides quality improvements of 0.010–0.025 SSIM over Lanczos filtering. Scene-specific training, in which the system automatically adapts to the current scene viewed by the camera, is shown to achieve up to 0.030 SSIM additional improvement in some scenarios. It is further shown that scene-specific training can provide some improvement even when reconstructing an unfamiliar scene, as long as the camera and capture settings remain the same.

Contents

Acknowledgements

I would like to thank my supervisors: Prof. Anders Heyden and Björn Ardö. I would also like to thank the people at Axis Communications who showed interest in my work and provided valuable advice. In particular: Viktor Edpalm, Fredrik Pihl, Gunnar Dahlgren, Niclas Danielsson, Anton Jakobsson, and Xing Danielsson Fan.

Terminology

To present the point more clearly, I will adopt the common terminology of image/video processing and machine learning even when discussing more general concepts. In this paper:

Derivative and **gradient** will be used even where it would be more correct to say 'subderivative' and 'subgradient'. Where relevant, I will point out which particular subderivative was chosen.

Framerate will be used even where it would be more general to say 'samplerate'.

Performance always means quality of results and never 'processing speed'.

Pixel will be used even where it would be more general to say 'sample'.

Resolution always means spatial resolution as in 'image resolution' and never 'sample bit-depth'.

Scaling (up/down) will be used even where it would be more general to say 'resampling'.

Super-resolution always refers to single image super-resolution.

Further concepts will be defined on first mention in the text.

1 Introduction and Problem Description

1.1 Video Bandwidth Requirements

For cost and efficiency reasons, it is desirable to minimise the data bandwidth, or *bitrate*, required to transmit and store digital video. A raw video stream contains much redundancy which can be more efficiently represented for transmission and storage. It is often also necessary to discard information, but this should be done in a way that impacts the viewer minimally.

1.1.1 Video Coding

One of the most common video coding methods used today is ITU-T H.264 [1], also known as MPEG-4 part 10 AVC. An H.264 coder uses several different lossless techniques, including motion compensation and approximation by interpolation between neighbouring pixels, to encode the video more efficiently. H.264 also contains a lossy quantisation step based on the *discrete cosine transform* which is a type of frequency transform. Following the transformation, each frequency component is quantised to reduce bitrate requirements. The overall quantisation strength is controlled by a value called the *quantisation parameter* or QP. Briefly, H.264 QP values run from 0 (lossless) to 51 (very strong quantisation, severe quality loss). Along with resolution and framerate, QP is the most important factor affecting the bitrate of an H.264 coded stream.

It is very common for H.264 video to be also *chroma subsampled*, which is another form of lossy compression. Chroma subsampling means that the colour signal of the video is sampled at a lower resolution than the lightness signal. To enable this subsampling, digital video is usually represented not in RGB colour format but in $\mathrm{Y'C_BC_R}$ format. An RGB sample can be transformed into $\mathrm{Y'C_BC_R}$ using the linear mapping:[1]

$$\begin{pmatrix} Y' \\ C_B \\ C_R \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{1}$$

where each of Y', R, G, B is in the range $[0,1]$ and C_B, C_R are in the range $[-0.5,0.5]$. Y' is the lightness or *luma* channel. C_B and C_R are *chrominance* channels, which for a fixed Y' value can be loosely interpreted as a plane with 'blueness' (C_B) on one axis and 'redness' (C_R) on the other. Y' corresponds closely to the single channel available in a black-and-white television set.

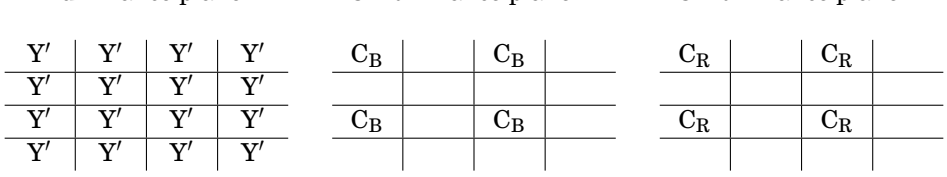


Figure 1: 4:2:0 chroma subsampling as defined by ITU-T H.264 [1]. The picture has a nominal resolution of 4×4 pixels. Chrominance samples are interpolated to obtain a complete colour picture.

Using chroma subsampling, every logical pixel of the image is associated with a unique luma sample but chrominance samples are spread out more sparsely. Figure 1 shows the 4:2:0 subsampling scheme, which is the most commonly used. Chroma subsampling is motivated by the lower sensitivity of the human eye to minute changes in chrominance, compared to luma [3, p. 87].

1.1.2 Regions of Interest

It is possible to code different parts of the image with different QP. If certain areas are known to be more interesting than others, those *regions of interest* (ROIs) may be coded with low QP, while

[1] This is the mapping for SDTV as defined by ITU-R BT.601 [2]. For HDTV, a completely different mapping is used which is unfortunately also called $\mathrm{Y'C_BC_R}$ [3, p. 296].

the remaining *background* may be coded with high QP. This can enable significant bandwidth savings. Such techniques have become popular in recent years, with the introduction of commercial implementations such as Axis Zipstream [4].

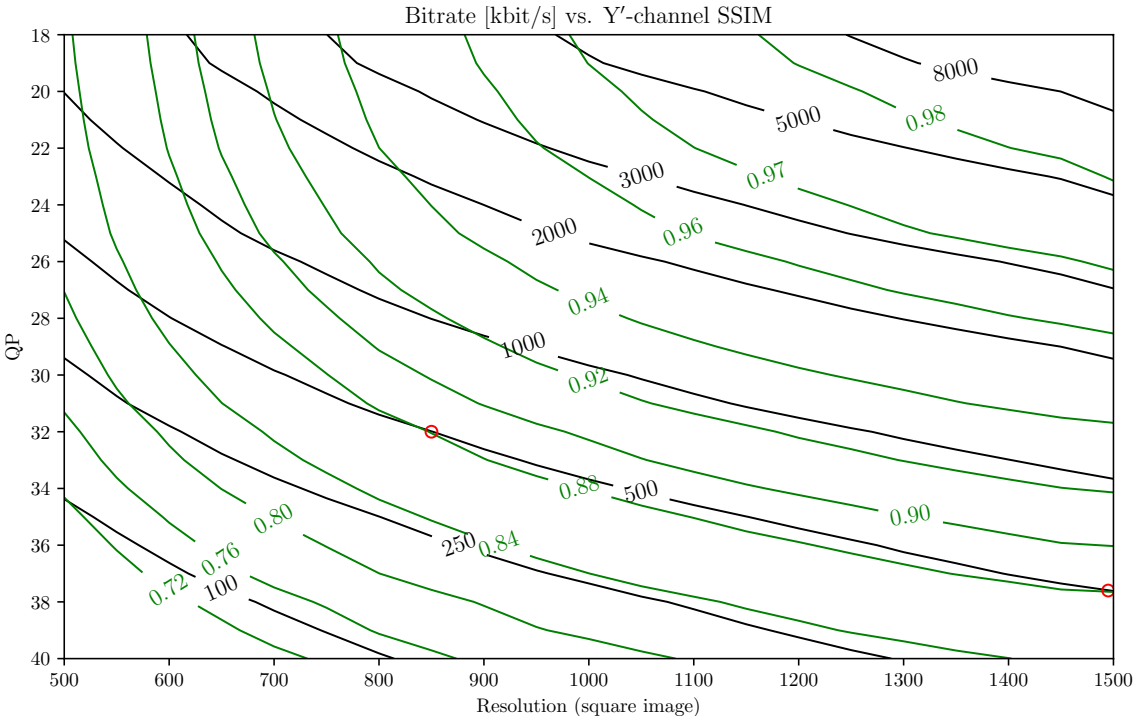


Figure 2: Example measurements of bitrate (black) and similarity index (green) between original and encoded video for a range of resolutions and compression levels.

1.1.3 Relation between Resolution, QP, Bitrate, and Quality

Figure 2 shows a simple demonstration of how QP, resolution, and bitrate correlate with perceived quality (here estimated by the SSIM index, see Section 3.6.4). In the figure, we see for instance that a quality of 0.88 by SSIM can be coded in two different ways at 500 kbit/s bitrate: either with low QP and low resolution, or high QP and high resolution. See Appendix A for details on how this test was carried out.

1.2 Image Reconstruction

Quantisation and downscaling both result in information loss. Figure 3 shows an example of noise introduced by each source. In order to improve the viewing experience, lost information should be *reconstructed* on the receiving end, to the largest extent possible. Of course it is not possible to reconstruct the exact information lost, but techniques exist that enable significant improvements.

The process of reconstructing information lost from downscaling is called *image super-resolution*. This term also includes reconstructing an image at a higher resolution than originally recorded. The process of filtering an image to remove quantisation artefacts is called *artefacts removal*.

1.3 Fixed Cameras

It is commonly thought that a good image reconstruction system should be able to upscale many different kinds of images well; such a system may be referred to as a 'general' reconstruction system. But in the context of forensic video, it is common for a camera to be viewing a fixed scene. Most cameras do not have built-in pan/tilt controls and are seldom moved. It is possible that a better 'scene-specific' reconstruction system could be designed by taking advantage of this fact.

(a) Reference (b) Scaling noise (c) Quantisation noise

Figure 3: The reference image (a) has been separately degraded by (b) downscaling to half size and then back up again with bicubic filtering, and (c) H.264 encoding at QP = 45. The signal loss in (b) and (c) is about equal when measured in peak signal-to-noise (see section 3.6.1).

There may still be a benefit to specialising on a specific camera, even if the scene is not fixed. Each camera model (and possibly each individual camera) has a characteristic influence on the output image, owing to factors such as differing optics, sensor, image processing and configuration.

1.4 Project Goals

This thesis will focus on developing an effective method for super-resolution and artefacts removal of streaming video, especially considering the scenario of fixed-camera forensic video. The finished system should meet the following goals:

1. Scene-specific reconstruction performance should be significantly better than comparable general methods: on the order of 0.01 SSIM increase and clear visual improvement.
2. The system can adapt to a new scene in an on-line fashion, alongside normal streaming operation. Peak performance is reached within a few hours.
3. Processing speed should be sufficient for at least modest real-time operation: 5 input megapixels per second or more on typical workstation-grade hardware. It should be possible to perform 2x reconstruction to at least 1920×1080 output resolution within the memory constraints of such hardware.
4. Even when viewing an unfamiliar or changing scene, performance should be good; certainly no worse than Lanczos filtering.
5. The system must not mislead the viewer by significantly misrepresenting the input video.

2 Related Work

The classic problem of image super-resolution has traditionally been addressed with sampling theory methods such as bicubic or Lanczos filtering [5]. These methods are popular due to their relative simplicity and modest computation requirements, but they often result in images that are blurry or contain artefacts such as ringing and edge overshoot.

Dong et al. [6] described the first method using convolutional networks for single-image super-resolution. Their system *SRCNN* improved upon earlier methods, notably dictionary-based methods. *SRCNN* used a bicubic upscaling filter as a preprocessing step before the neural network, and was trained using a mean squared error (MSE) loss. Later, Dong et. al described an accelerated version *FSRCNN* [7], which replaced the initial bicubic filter with a final transposed convolution layer. This improved processing speed by reducing the spatial size of the network by a factor four, except for the final layer.

Although methods like *SRCNN* and *FSRCNN* provide significant improvements, they still tend to produce blurry images in certain situations. Recent methods have identified the use of pixel-wise errors like MSE as one of the contributing factors, and have thus attempted to augment or replace MSE loss functions with other losses. Johnson et al. [8] described a 'perceptual' loss function which used an auxiliary CNN as a mapping from pixel space to 'feature space', before calculating the MSE in this space. This loss function was shown to improve reconstruction of fine detail and edges, at the cost of introducing artefacts in some of the output images. Sajjadi et al. [9] used a combination approach where the loss function is a sum of a perceptual loss, a special 'texture matching loss' intended to promote texturisation of the output image, and an auxiliary discriminator network trained using a *generative adversarial* process. Their method was successful at mitigating the artefacting tendencies of the perceptual loss, and generally at producing highly detailed images.

More radical methods such as the *pixel-recursive* method described by Dahl et al. [10] blur the line between super-resolution, understood as a resampling problem, and *image hallucination*. Their system was able to generate impressive output images from highly undersampled input, but at the cost of essentially removing the link between the ground truth and the reconstructed image.

These systems build on foundational work such as the Adam optimisation algorithm developed by Kingma & Ba [11] and the generative adversarial training method described by Goodfellow et al. [12]. Also of great importance is the SSIM similarity measure developed by Wang et al. [13], which is often used to evaluate the performance of super-resolution systems.

3 Theory

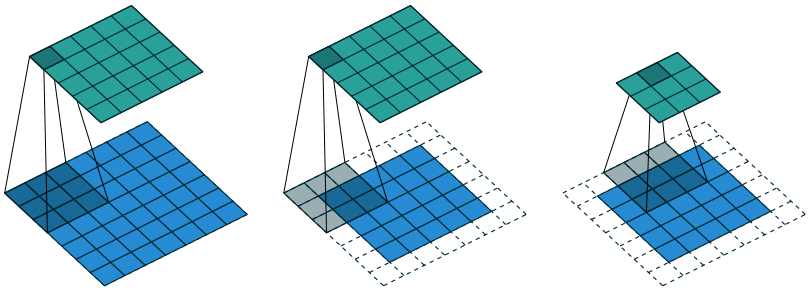
3.1 Single Image Super-resolution

The problem of single image super-resolution is fundamentally a problem of 'inventing' or *inferring* information. In a 2x super-resolution filter, four output pixels must be generated for each single input pixel, resulting in an output image which informally contains 75% inferred information and only 25% true information, pixel-by-pixel. For the case of 4x super-resolution, only 6% of the original information remains. In mathematical terms, the problem of image super-resolution is *ill-posed*: there are many possible solutions and no way of knowing which corresponds to the true original image.[2] A good super-resolution algorithm can generate a reconstruction that is—in some sense—similar to the true high-resolution image.

In this section we will briefly summarise the theory behind image super-resolution and deep learning, drawing on the basic literature including Stanford University's CS231n course [14] and Dumoulin & Visin's guide to convolutional arithmetic [15]. We will also elaborate on what it means for two images to be similar.

3.2 Convolutional Filters

For digital super-resolution, the choice of discrete convolutional filters is natural. A convolutional filter conceptually consists of a *filter kernel* represented by a matrix K of size $w \times h$. It is common to use square filters so that $w = h = s$. The filter sweeps over the image, calculating the element-wise sum of the matrix product Kx_{ij} for each window x_{ij} also of size $w \times h$. In other words, for each output pixel to be generated, the filter views a neighbourhood of the corresponding input pixel and calculates a weighted sum over this neighbourhood. Figure 4 shows graphical representations of different types of 2D convolution.



(a) 3×3 convolution, no padding

(b) 3×3 convolution, 'same' padding

(c) 3×3 convolution, strided 2×2

Figure 4: Visual demonstration of discrete convolutions. The blue array represents the input image and the green array represents the output. Reproduced from Dumoulin & Visin [15] under license.

We will also add a scalar *bias* term b after performing the convolution. Using $(*)$ to denote discrete convolution, we may write the full filter as:

$$y = K * x + b. \tag{2}$$

3.2.1 Padding, Reflection, and Receptive Field

In Figure 4a we see that the output is smaller in size than the input because convolution has to 'stop' at the edge to avoid partly falling outside the image. This is not ususally desirable, so in practice we will add zero values as padding on the edge of the image to avoid this shrinking effect. This is sometimes called 'same' padding, shown in Figure 4b.

[2] A theoretical exception would be a band-limited image sampled to at least the corresponding Nyquist rate. Such an image could be upscaled exactly to any resolution. For this thesis, we are interested in photographs or video recordings of the world, which we think of as containing essentially unlimited detail; more than can be captured by any camera.

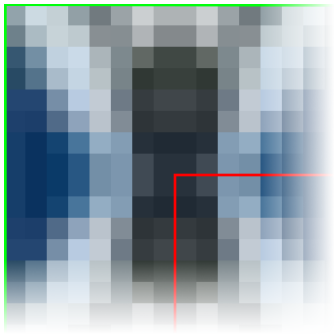


Figure 5: 8-pixel even reflection of an image, top-left corner. The red and green lines mark the image boundary before and after reflection, respectively.

If only straight zero padding were to be used, the resulting image would appear to have a dark fringe near the edges. Therefore, the input image must be padded with something else than zeros in a separate step before any other filtering. The common method is to simply 'reflect' the image at the edges. Figure 5 shows how this kind of reflection padding works.

The *receptive field* of a network is the size of the rectangle of input pixels that a single output pixel depends upon. Each successive filter (larger than 1×1) connected in series, expands the receptive field. A network of five 3×3 convolution filters in series has a receptive field size of 11×11. We see that the amount of reflection padding should be large enough to cover the receptive field.

3.2.2 Strided Convolution

In Figures 4a and 4b we see that the filter steps a single pixel in each direction on every application. Sometimes it may be desirable to use *strided convolution* as shown in Figure 4c. This can thought of as the filter 'skipping over' some windows. We see that this results in a downscaling of the image. For a stride of k in each direction the image will be downscaled to $1 / k$ of the input size.

3.2.3 Transposed Convolution

The opposite operation of strided convolution is an upscaling operation. This operation is sometimes referred to as *deconvolution*, a slightly unfortunate name as this also refers to a different operation in signals processing. To avoid confusion, many authors prefer the name *transposed convolution*.

To understand how transposed convolution works, we may express normal convolution as a matrix multiplication. Consider a 3×3 filter K operating on four 3×3 patches of a single-channel image x without any padding (see Figure 6 for a graphical representation). If the input and output images x and y are represented as 'flattened' vectors of 4 and 16 elements respectively, we may write this convolution as ordinary matrix multiplication:

$$\tilde{y} = \tilde{K}\tilde{x}$$

$$\begin{pmatrix} y_{11} \\ y_{12} \\ y_{21} \\ y_{22} \end{pmatrix} = \begin{pmatrix} K_{11} & K_{12} & K_{13} & 0 & K_{21} & K_{22} & K_{23} & 0 & K_{31} & K_{32} & K_{33} & 0 & 0 & 0 & 0 & 0 \\ 0 & K_{11} & K_{12} & K_{13} & 0 & K_{21} & K_{22} & K_{23} & 0 & K_{31} & K_{32} & K_{33} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & K_{11} & K_{12} & K_{13} & 0 & K_{21} & K_{22} & K_{23} & 0 & K_{31} & K_{32} & K_{33} & 0 \\ 0 & 0 & 0 & 0 & 0 & K_{11} & K_{12} & K_{13} & 0 & K_{21} & K_{22} & K_{23} & 0 & K_{31} & K_{32} & K_{33} \end{pmatrix} \begin{pmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{21} \\ \vdots \end{pmatrix}. \tag{3}$$

By transposing K we end up with an opposite operation, producing 16 output values $\tilde{y}'$ from 4 input values $\tilde{x}'$:

$$\tilde{y}' = \tilde{K}^T \tilde{x}'. \tag{4}$$

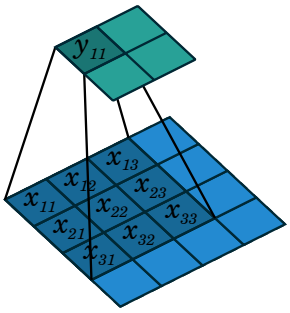


Figure 6: Visualisation of Equation 3. Adapted from Dumoulin & Visin [15] under license.

If we replace $\tilde{K}$ instead with a strided convolution (a downscaling operation)[3] we see that $\tilde{K}^T$ now describes an *upscaling* operation within the convolutional framework. Just like a (k,k) strided convolution layer will downscale the image by a factor k, a (k,k) strided transposed convolution layer will upscale the image by a factor k.

3.2.4 Filter Networks

Our super-resolution apparatus will be a *convolutional network* consisting of multiple layers of convolutional filters, the output of each layer feeding into the next (see Figure 7). Each layer will consist of a *filter stack* of multiple convolutional filters, each filter conceptually producing a separate mono-channel output image. The output image has the same width and height as the input image, because we use 'same' padding.

For efficiency, we work on *mini-batches* of several images at a time, each batch containing b individual c-channel images of size $w \times h$. Such a batch is represented as a 4-dimensional tensor of size $b \times h \times w \times c$. The filters operate uniformly on each image in the batch by convolving over all the input channels simultaneously. For each $b \times h \times w \times c$ input batch the filter stack produces an *activation volume* of size $b \times h \times w \times f$. This tensor becomes the input to the next layer. f is called the *filter depth* of the layer. The output of the last layer becomes the final result of the operation.

Assuming square filters of size s, we can collect the filter kernels of a single layer into a $c \times s \times s \times f$ size tensor and the biases into a $1 \times f$ size tensor. These tensors, together with the stride and padding settings, completely describe the convolution operation.

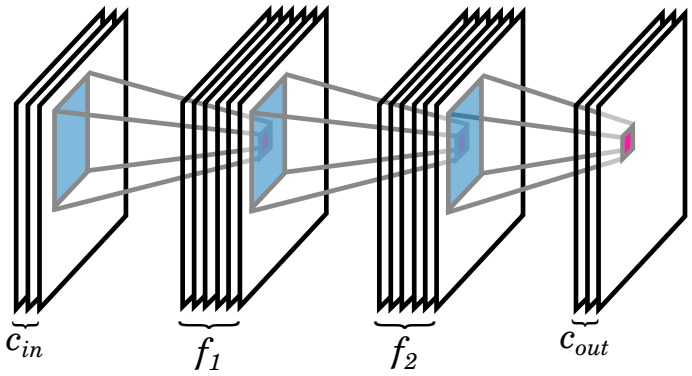


Figure 7: Graphical representation of a three-layer convolutional network.

For RGB input, we see that the first layer will convolve over $c_{in} = 3$ input channels. If we also want RGB output, the final layer should have $c_{out} = 3$ filter depth, each filter producing a single channel of R, G, B. For the remaining layers in-between, the output channels cannot be directly

[3] It may appear as if equation 3 already describes a downscaling operation, but this is because we have omitted padding.

interpreted as colours. Instead we shall think of these channels as constituting a *feature map*, each channel corresponding to some feature possibly present in the image. The value at a certain location in the activation volume corresponds to the amount of 'presence' of a particular feature in the corresponding location in the input image. The filter depth of a given layer determines the number of distinct features it can describe. We expect the feature map to move from low-level features to more high-level features as we move deeper into the filter network. The final layer will use the last feature map to reconstruct the image features in terms of concrete pixels.

Because the filter stacks operate uniformly on each input window, a fully convolutional filter network is able to process images of any size. We may use normal convolution with 'same' padding to maintain image resolution through a layer, strided convolution layers to downscale the image, and transposed convolution layers to upscale the image.

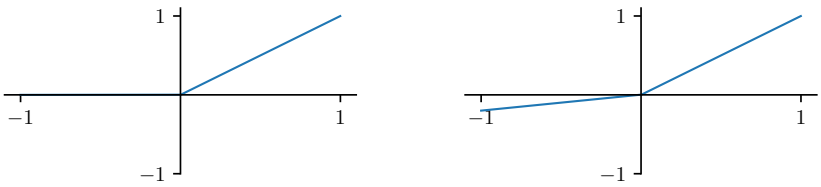
3.3 Activation Functions

Convolution is a linear operation which can be formulated as a matrix multiplication (see Section 3.2.3). Biasing can also be formulated as matrix multiplication by adding a separate 'bias dimension' to the input which is filled with ones [14]. This can be illustrated with a simple 3×3 example: given an input a, a bias b, and $\mathbf{1}$ meaning the 3×3 matrix filled with ones, we can write the bias operation as

$$a + b\mathbf{1} = \begin{pmatrix} a_{11}+b & a_{12}+b & a_{13}+b \\ a_{21}+b & a_{22}+b & a_{23}+b \\ a_{31}+b & a_{32}+b & a_{33}+b \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & 1 \\ a_{21} & a_{22} & a_{23} & 1 \\ a_{31} & a_{32} & a_{33} & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ b & b & b \end{pmatrix} = \tilde{a}\tilde{b}. \tag{5}$$

If the filter network consists only of composed linear operations (matrix multiplications), we can always simplify into only a single matrix multiplication using basic linear algebra. To avoid reducing the power of our multi-layer network into that of only one layer, non-linearity must be introduced. This is done by applying a non-linear *activation function* f element-wise to the output of each layer, except possibly the last. Comparing with equation 2, a convolutional layer with activation function can be written:

$$y = f(K * x + b). \tag{6}$$



(a) ReLU (b) Leaky ReLU, $\rho = 0.2$

Figure 8: Common activation functions.

3.3.1 Rectified Linear Unit

The *rectified linear unit* or *ReLU* (Figure 8a) is defined as

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

The electronic analogue would be an ideal diode. ReLU is differentiable everywhere except at the point $x = 0$, where we must choose a subderivative in the interval $[0, 1]$. We choose $f'(0) = 0$ and the full subderivative becomes:

$$\frac{df}{dx} = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

3.3.2 Leaky ReLU

The *leaky ReLU* (Figure 8b) adds a small 'leak resistance'. Leaky ReLU is defined as

$$f(x) = \begin{cases} x & \text{if } x > 0, \\ \rho x & \text{otherwise.} \end{cases} \tag{9}$$

Typically ρ is positive and on the order of 0.1. We also define a subderivative:

$$\frac{df}{dx} = \begin{cases} 1 & \text{if } x > 0, \\ \rho & \text{otherwise.} \end{cases} \tag{10}$$

3.4 Machine Learning

A *convolutional neural network* (CNN) typically consists of several convolutional filters connected in series, or in more complex arrangements. Instead of being calculated from a fixed distribution (like the Lanczos or bicubic filter kernels), in a CNN the kernels and biases are 'learned' dynamically during the training process. We use the term *parameters* to refer to the complete collection of all kernel and bias weights in the network. These are interpreted as a single vector which we denote θ. Aspects of the network which are controllable but not learnable during training, such as the number of layers, filter depth, kernel size etc., are called *hyperparameters*.

A common process uses several convolutional filters connected in series, with parameters initialised from random values. Let N denote the complete network and let x denote an input image to be upscaled. Then $\hat{y} = N(x)$ is the inferred super-resolution image output by the network. Further, let y denote the ground truth high-resolution image. We define a *loss function* $\mathscr{L}(y, \hat{y})$ which describes the current performance of the network by comparing the network's output $\hat{y}$ to the ground truth y and estimating their similarity. Lower values of $\mathscr{L}$ correspond to better performance.

Since we always work on mini-batches of possibly several images, the variables x, $\hat{y}$, and y will in practice come to represent batches of several images and $\mathscr{L}$ will compute the mean loss value across the batch. For a fixed choice of x, y, the output $\hat{y}$ is also fixed by the relation $\hat{y} = N(x)$. At such a fixed sample point, $\mathscr{L}$ only depends on the parameters; $\mathscr{L}(\theta)$. We may interpret this loosely as an estimate of the full field $\mathscr{L}$ (over all possible images), using the current batch as a sample.

Formally then, for a given input batch the loss is a function only of the parameters $\mathscr{L} = \mathscr{L}(\theta)$. Training the network N consists of optimising this function, choosing progressively better and better values for θ. We will study how $\mathscr{L}$ varies by the choice of parameters θ by looking at the properties of $\mathscr{L}(\theta)$ sampled over mini-batches x, y. After processing each batch we will perform an *update*, selecting a new value for θ. If a better upscaling filter than the bicubic or Lanczos filter exists, in principle the network should be able to learn it.

3.5 Optimisation

We interpret the set of possible parameters θ as constituting a *parameter space* with any given choice of θ being a vector in this space. We commonly have many thousands of parameters, so this is a very high-dimensional space. Also, in general $\mathscr{L}$ is not a convex function or any other type of more easily optimisable function, except that we require $\mathscr{L}$ to be piecewise differentiable with respect to θ.

3.5.1 Gradient Descent Methods

It is a basic fact of multivariate calculus that the gradient of a function points in the direction of greatest local increase of the function, and the magnitude of the gradient is equal to the instantaneous rate of increase.

The basic method of optimisation in machine learning consists of calculating the gradient of $\mathscr{L}$ with respect to θ at the current point in feature space, and taking a 'step' in the opposite direction. A

large magnitude of the gradient implies that we should take a longer step. By applying this general method iteratively, and sampling over a large amount of data, it is hoped that we will eventually reach a sufficient minimum of $\mathscr{L}$.

3.5.2 Simple Gradient Descent

The simplest method is to interpret the gradient literally as the step to be taken, scaled by a hyperparameter α called the *step size* or *learning rate*:

$$\theta_t = \theta_{t-1} - \alpha \nabla_\theta \mathscr{L}(\theta_{t-1}). \tag{11}$$

This method of updating θ is called *simple gradient descent*.

3.5.3 *Adam* Optimisation

Adam is a modified version of the basic gradient descent, introduced by Kingma & Ba in 2014 [11]. The update step of Adam can be written as:

$$\begin{aligned} m_0, v_0 &= 0, \\ g_t &= \nabla_\theta \mathscr{L}(\theta_{t-1}), \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t}, \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}, \\ \theta_t &= \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}. \end{aligned} \tag{12}$$

Here, m_t and v_t are exponential moving average estimates of the raw mean and raw uncentred variance of the gradient, respectively. To compensate for the fact that we set m_t and v_t to zero initially, bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$ are used for the final step. β_1, β_2, and are hyperparameters controlling the decay rate of the exponential averages and which must satisfy $0 \leq \beta_1, \beta_2 < 1$. The meaning of the hyperparameter ϵ is not explained by the authors, but it appears to be a small constant to avoid division by zero in the case that $\hat{v}_t = 0$. Kingma & Ba suggest the following defaults:

$$\begin{aligned} \beta_1 &= 0.9, \\ \beta_2 &= 0.999, \\ \epsilon &= 10^{-8}. \end{aligned} \tag{13}$$

The ratio $m_t/(\sqrt{v_t} + \epsilon)$ is interpreted as a sort of signal-to-noise estimate. Higher values of this ratio imply larger confidence in the estimate $\hat{m}_t$. This is used to scale the step size α according to the confidence of the estimated gradient direction. The use of exponential moving average acts as inertia, slowing the reaction time of the optimiser and making it more resistant to noise.

3.5.4 Step size Annealing

It is considered good practice to reduce, or *anneal*, the step size α as training proceeds. Smaller step sizes allow the optimiser to focus on a particular local minimum, but may slow down the training process. A too large step size may cause the optimiser to overstep and miss a minimum. Step size is thus a very important hyperparameter.

One way of annealing the step size is *inverse-time annealing*. With this scheme, we define:

$$\alpha(t) = \frac{\alpha_0}{1 + dt}, \tag{14}$$

where t is the current time-step, α_0 is the *base step size* or *base learning rate* and d is a hyperparameter determining how quickly the step size anneals. The step size may either anneal continuously,

or in a 'staircase' fashion where the step size is piecewise constant and reduced at regular intervals. Figure 9 shows an example of how inverse-time annealing affects the step size.

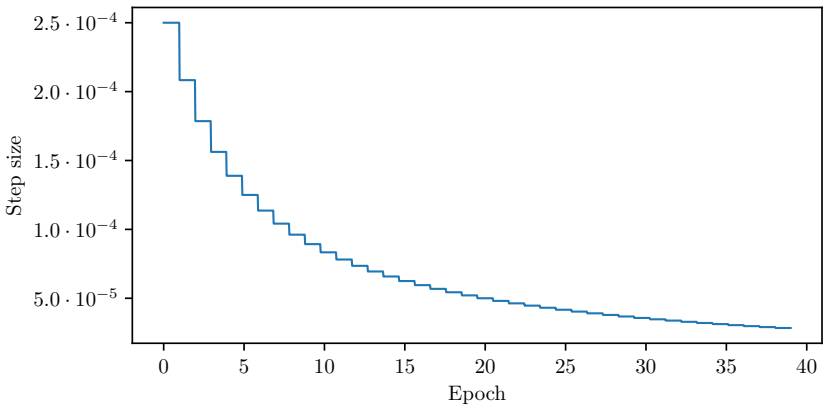
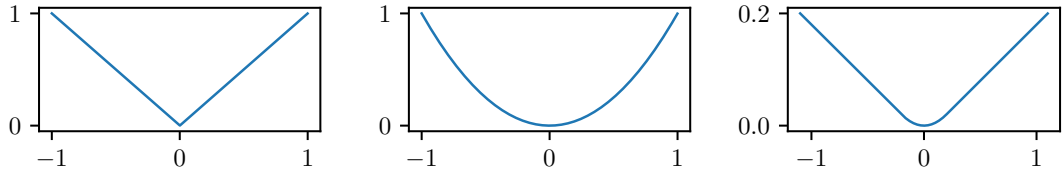


Figure 9: Example of inverse-time annealing in a 'staircase' fashion.

Sometimes the time-step in Equation 14 is replaced with the *epoch number*. An epoch refers to a single pass through the entire training set. The epoch number is the number of times the network has 'seen' at least part of every image in the training set.

3.6 Measuring Network Performance



(a) L_1 error (b) L_2 error (MSE) (c) Huber loss ($\delta = 0.2$)

Figure 10: Comparison between pixel-wise loss functions. Note that the Huber loss is scaled differently from the other losses; a deviation of ± 1 is mapped to δ and not to 1.

3.6.1 Mean Squared Error and Signal-to-noise Ratio

Clearly, the choice of loss function $\mathscr{L}$ is critical in determining network performance, because the network can only learn the behaviour encouraged by the loss function. A common choice is *mean squared error*, or MSE, a classic error estimation method used in signals processing. MSE is a simple pixel-wise comparison and is formally defined in terms of the Euclidian (L_2) norm as:

$$\mathscr{L}_{\text{MSE}}(y, \hat{y}) = \frac{1}{N} ||y - \hat{y}||_2^2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2, \tag{15}$$

where N is the number of pixel-channels in the image. E.g. for a $w \times h$ pixel image in RGB colour, we would have $N = 3wh$. The appearance of the L_2 norm encourages interpretation of the MSE as the square of geometric distance between two images in N-dimensional 'pixel space'.

MSE has the benefit of being simple to understand and to calculate. To facilitate optimisation, it is critical that the loss function is piecewise differentiable, and $\mathscr{L}_{\text{MSE}}$ clearly fulfils this requirement. From the MSE we can also calculate the *peak signal-to-noise ratio*, or PSNR, which is commonly used to assist human evaluation of network performance, and to compare different super-resolution methods. PSNR is defined as

$$\text{PSNR} = 10\log_{10}\left(\frac{L^2}{\mathscr{L}_{\text{MSE}}(y,\hat{y})}\right), \tag{16}$$

where L is the dynamic range of each sample. For the common case of 8-bit integer colour, we have $L = 255$. PSNR is measured in decibels, higher values corresponding to better network performance. Identical images have PSNR $= \infty$ dB.

3.6.2 Huber Loss

The Huber loss function [16] is a variant of MSE that attempts to diminish the impact of *outlier* values in the distribution, occasional values which deviate strongly and affect the mean error disproportionately. The mean Huber loss is defined as:

$$\mathscr{L}_{H,\delta} = \frac{1}{N}\sum_{i=1}^{N}\begin{cases}\frac{1}{2}(y_i-\hat{y}_i)^2 & \text{if } |y-\hat{y}_i| \le \delta,\\ \delta(y_i-\hat{y}_i)-\frac{1}{2}\delta^2 & \text{otherwise.}\end{cases} \tag{17}$$

The Huber loss function is quadratic for errors within δ and linear outside this interval. In this way, the Huber loss works as a differentiable and slightly re-scaled approximation of the L1 error for a suitable choice of δ. The difference between MSE and the Huber loss can be seen in Figure 10. We can calculate a decibel value for the Huber loss in the same way as MSE, using the same equation 16.

3.6.3 Shortcomings of Pixel-wise Losses

Recent work [9, 17] has highlighted certain downsides of using pure pixel-wise loss functions such as MSE or the Huber loss. Such losses are very sensitive to the exact spatial location of image features. This is in contrast to human observers who are less sensitive to translations of a few pixels. For this reason, pixel-wise losses strongly penalise high-frequency errors compared to low-frequency errors. This is sometimes in contradiction to the aim of super-resolution to generate detailed and 'plausible' images. Pure MSE-trained networks can tend to generate slightly blurry output, because risk-taking in the generation of high-frequency detail is discouraged.

Reference | 0.5x Lanczos downscaling + 2x nearest-neighbour upscaling | 0.5x Lanczos downscaling + 2x CNN upscaling

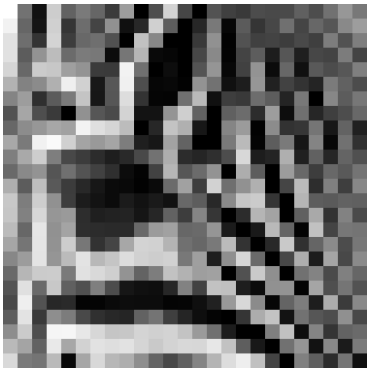
Figure 11: Example of how a network trained with pixel-wise (Huber) loss reacts to lost information.

Figure 11 shows an example of this behaviour when applied to regular patterns; in this case roof tiles running diagonally top-left to bottom-right. A network trained with pixel-wise loss is most 'concerned' with not being wrong, because the loss function penalises pixel-wise errors strongly. When 'unsure' which of two solutions is correct, the network will tend to draw the mean value of both solutions. In Figure 11, the reference image has initially been downscaled by a factor two. Due to this downscaling, it is no longer clear whether the roof tiles run top-left to bottom-right or top-right to bottom-left. When faced with this ambiguity we see that the network 'prefers' to draw both solutions on top of each other, causing the pattern to appear to run *both* ways.

A pathological example is seen in Figure 12 where an image of a zebra has been downscaled by a factor four, resulting in severe loss of detail in the stripes. Due to the downscaling, it is no longer

(a) Reference

(b) Slight affine transformation
PSNR = 17.05 dB, SSIM = 0.8358



(c) 4x Lanczos downscaling

(d) 4x Huber-trained CNN upscaling
PSNR = 17.82 dB, SSIM = 0.8020

Figure 12: The 100×100 pixel reference image (a) has been transformed in two different ways: (b) 102% upscaling followed by 1° rotation inside the same frame, and (c) Lanczos downscaling to 25×25 followed by (d) upscaling back to 100×100 pixels using a CNN trained with pixel-wise Huber loss.

clear exactly where the stripes should go. A pixel-wise loss function forces the network to generate a blurry grey superposition of stripes (Figure 12d), because it will not risk putting the stripes in the wrong place. Note that, like in Figure 11, the stripes appear to partially go 'both ways'. By contrast, Figure 12b looks almost identical to a human observer; it has the exact same features but slightly moved. We see that the PSNR measure actually prefers the downscaled image even though it looks much worse to a human observer.

Despite these shortcomings, pixel-wise losses remain common in practice. Benefits include simplicity of understanding and training as well as reduced risk of misleading the viewer; networks trained with pixel-wise losses tends to blur areas of uncertain content rather than adding inferred detail.

3.6.4 Structural Similarity Index

The desire for a more 'perceptual' measure of image similarity resulted in the *structural similarity index* (SSIM) [13]. This measure attempts to improve upon MSE by focusing more on the structure of an image rather than the absolute, pixel-by-pixel error. SSIM is typically calculated on sliding 8×8 pixel single-channel windows of the image.[4] The SSIM index for two windows x and y is then defined as

$$\mathrm{SSIM}(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}, \tag{18}$$

[4]The original implementation by Wang et al. used a circular Gaussian weighting function on the window before calculating the average and standard deviation. In this paper we use instead a simplified implementation without weighting.

where μ_x and μ_y are the mean values of x and y, σ_x^2 and σ_y^2 are their variances and σ_{xy} is their co-variance. c_1 and c_2 are constants defined by $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$ where L is the dynamic range of the image ($L = 2^8 - 1$ for 8-bit images). For the constants k_1 and k_2, Wang et al. suggest the values $k_1 = 0.01$, $k_2 = 0.03$.

When we talk about the SSIM of two images, we refer to the mean value of Equation 18 over the full image pair. Higher SSIM between a reconstructed image and the original corresponds to stronger similarity and thus better network performance. Identical images have SSIM = 1. It is common to calculate SSIM only along the Y′channel of the $\mathrm{Y'C_BC_R}$ colour space as defined by Equation 1, disregarding colour information.

SSIM is often used in the literature to evaluate different super-resolution methods, and more generally to evaluate the performance of other image transformations. In this thesis we will use SSIM as the main estimator of image similarity, combined with visual comparison of image samples to confirm the estimates.

3.6.5 Perceptual Losses

Recently, more complex 'perceptual' loss measures have been proposed [8]. Such methods feed both the ground truth y and inferred image $\hat{y}$ of the main network into an auxiliary network ϕ. Intermediate activations from different layers of ϕ are extracted and interpreted as a representation of y and $\hat{y}$ in the 'feature space' of the network ϕ. The loss is then calculated with a standard norm-based method like MSE in this feature space, instead of pixel space. By using $\phi_n(x)$ to mean 'the activations of layer n in ϕ when ϕ is fed the image x', we can write the full loss using feature-space MSE as:

$$\mathscr{L}_p(\hat{y}, y) = \frac{1}{N} ||\phi_n(\hat{y}) - \phi_n(y)||_2^2, \tag{19}$$

where N is the number of elements in the activation volume of layer ϕ_n. It is also possible to construct the loss as a weighted sum of the activations at several different layers.

While successful at producing plausible super-resolution images, such methods have been shown to exaggerate the detail content of images, resulting in 'checker-board' artefacts, unless weighted together with another loss function; see for instance Sajjadi et al. [9, fig. 4].

3.6.6 Image Hallucination

More extreme methods venture into the realm of *image hallucination*, with the explicit aim of fooling human observers into thinking that they are looking at a real picture [10]. These methods can achieve 'plausible' results at extreme upscaling factors but the correspondence between motif and generated image may be lost. E.g. for a portrait, the result may well be a very convincing portrait of a completely different person. Although certainly very interesting, image hallucination methods do not seem relevant to the objective of this thesis.

3.7 Parameter Initialisation

Initialisation refers to the starting parameters chosen before any training. These parameters should be of appropriate size so as to promote training. The basic method consist of initialising the filter kernels with random numbers and the filter biases with zeros.

3.7.1 Glorot Initialisation

One common method, as suggested by Glorot & Bengio [18], is to initialise the kernel weights K_{ij} with uniformly distributed random numbers, scaled by the inverse square root of the kernel size s:

$$K_{ij} \sim U\left[-\frac{1}{\sqrt{s}}, \frac{1}{\sqrt{s}}\right]. \tag{20}$$

4 Existing Systems

Before starting to design a completely new system, several existing open-source implementations are evaluated for the intended application. It is hoped that elements of their architecture or even implementation could be re-used for this project. Such implementations also often come bundled with pre-trained parameters, which can alleviate the need for time-consuming ground-up training.

4.1 Candidates

4.1.1 Criteria for Inclusion

To be considered a system must be available as open-source, with pre-trained parameters appropriate for a 'photo' scenario. The system must have both inference and training passed fully implemented and ready to run. Ideally, the system will also be released under a suitable 'permissive' license which allows potential re-use of source code in a proprietary product.

Each selected candidate will be described briefly in the following paragraphs.

4.1.2 *Waifu*

Waifu [19] uses a relatively simple architecture containing six convolution layers and a final transposed convolution layer. The convolution layers all have the same filter size of 3×3, while the transposed layer has 4×4 filters with 2×2 strides. The filter depth increases throughout the network, culminating in 256 channels for the final regular convolution layer. The spatial resolution is the same throughout the network (except for the final transposed layer), i.e. there is no downsampling.

The network uses leaky ReLU activation with $\rho = 0.1$ after each regular convolution layer and is trained using a Huber loss with $\delta = 0.1$ and a special linear weighting (mimicking the definition of the $\mathrm{Y'}$channel of $\mathrm{Y'C_BC_R}$) of the RGB channels before calculating the loss.

Waifu is implemented in Lua using the *Torch* library and is distributed under an MIT license. The network contains about 550,000 parameters in total.

4.1.3 *SeRanet*

SeRanet [20] introduces a special *split*/*splice* algorithm. After the first few layers the network splits into four strands, each of which generates one quarter of the output image. The quarters are interleaved to form 2×2 pixel blocks, each containing one pixel from each strand. The image is then re-assembled at full output resolution and passed through several more layers. This method is similar in some ways to the *sub-pixel* method of Shi et al. [21], except that the split occurs in the middle of the network instead of at the very end.

SeRanet is implemented in Python using the *Chainer* library and is distributed under an MIT license. The network contains about 3,300,000 parameters in total.

4.1.4 *Neural Enhance*

Neural Enhance [22] combines perceptual losses as described in Section 3.6.5 with generative adversarial training. The network consists of three distinct sub-networks: the *generator*, *discriminator* and the *perceptual loss* section. Of these only the generator is part of the inference pass and the remaining two sub-networks are only used during training.

Neural Enhance is implemented in Python using the *Theano* and *Lasagna* libraries and is distributed under the GNU Affero licence, version 3. The network consists of about 4,460,000 parameters of which 2,660,000 are needed for the inference pass.

4.2 Method

4.2.1 Test Scenes

Three different test scenes were selected for evaluation and assigned code-names:

Elite. A 24-hour time lapse of the *Elite* hotel in Lund, including a small road. A 3072×1728 JPEG frame at maximum quality was captured every minute, for a total of 1,440 frames.

Office. A 24-hour timelapse of an office interior, captured from a ceiling-mounted camera. A 1920 × 1080 JPEG frame at maximum quality was captured every minute, for a total of 1,440 frames.

Construction. A 15-month timelapse of a construction site where a building is gradually being erected. One 1280 × 720 JPEG frame at high quality was captured every five minutes, for a total of about 48,000 frames.

Sample frames of each scene are given in Appendix B.

4.2.2 Evaluation Criteria

Ten frames were randomly selected from each timelapse and used to evaluate the systems with the following process:

1. Downscale the reference frame by a factor two using ImageMagick's built-in Lanczos filter [23].
2. Upscale the frame again by a factor two using the candidate system.
3. Measure the similarity between reference and reconstructed image using SSIM (see Section 3.6.4).
4. Verify the synthetic similarity measurements by visual inspection of selected frames.

Network complexity, measured roughly by the total number of parameters and layers, is also a factor. Larger networks require more memory and take longer to train. The more complex networks are expected to perform comparatively better to justify the additional overhead.

In each case, pre-trained parameters published by the system author were used to initialise the network. The built-in Lanczos upscaling filter in ImageMagick was also included as a baseline for comparison.

4.3 Results

The numerical results of the evaluation are noted in Table 1. Image comparisons can be seen in Figure 13. In all three scenes, *Waifu* was the clear winner by SSIM. These results were also confirmed by visual inspection of a subset of the example images.

	Lanczos2x		Ne2x		Seranet2x		Waifu2x	
	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM σ
Construction	0.9220	0.0323	0.9295	0.0266	0.9377	0.0256	**0.9453**	0.0230
Elite	0.9427	0.0077	0.9437	0.0106	DNF	DNF	**0.9542**	0.0113
Office	0.9482	0.0082	0.9569	0.0073	0.9618	0.0066	**0.9696**	0.0057

Table 1: Mean value μ and standard deviation σ of the similarity for each candidate and scene. The best mean value for each scene is marked in boldface.

The performance of *Neural Enhance* ('Ne2x' in Table 1) appears quite disappointing: in no scene was it able to outperform the baseline Lanczos filter by as much as a percentage point of SSIM. N.E. appears to introduce significant noise into the images when compared to the other candidates, including colour noise.

The difference between *SeRanet* and *Waifu* in the test scenes is more subtle. *Waifu* appears to generate more controlled edges with less overshoot compared to SeRanet. This can be seen around the stalks of the flower in the leftmost column of Figure 13, or around the boom of the excavator in the centre column. *Waifu* also seems to better pronounce very thin edges; note for example how the fence in the background of the centre column almost fades into the snow with *SeRanet*, but with *Waifu* the poles are more distinct.

SeRanet was unable to process the *Elite* scene at all due to running out of memory. Tests of running *SeRanet* in software mode (as opposed to GPU mode) suggest that over 10 GB of memory would be required to run the inference pass at 5 megapixel resolution output, exceeding the 8 GB of memory available on our GTX 1080 test GPU.

Figure 13: Sample 100 × 100 pixel crops from the evaluation dataset. From the left: *Office*, *Construction*, and *Elite*. *SeRanet* was unable to process the *Elite* scene in the available memory.

4.4 Conclusion

Of the three systems evaluated, *Waifu* appears to give the best performance for the chosen test scenes. This network also has the benefit of a relatively low computational complexity, which is a necessity if we hope to be able to perform training on the camera device itself.

SeRanet had higher memory requirements than the other candidates, failing to process a 5 megapixel output scene in 8 GB of memory (a common memory size for workstation-grade GPUs at the time of writing). This problem could have been solved by provisioning a GPU with larger memory, or by splitting the image into chunks and running the inference pass on each chunk separately. However, the performance of SeRanet on the other two scenes did not inspire such efforts.

The performance of *Neural Enhance* is underwhelming considering its complexity. *Neural Enhance* is also distributed under a less-than-permissive license (GNU AGPL) which imposes significant restrictions on its use.

It is possible that these are all in fact learned behaviours and that either system could be trained to better handle the slightly noisy and sharpened images generated by a typical security camera. But as stated, the purpose of this evaluation was to select a single network architecture for further study. The pre-trained parameters supplied by the author of each system seem like a reasonable basis for such evaluation.

5 System Design

Using *Waifu* as the main inspiration, a new reconstruction filter is implemented in Python using the *Tensorflow* machine learning framework. This new system is dubbed *Vger*.

5.1 Network Layout

The *Vger* network consists of six convolution layers and a final transposed convolution layer. The filter depth of the network increases throughout the network as shown in Table 2. The network is completely 'straight' with small 3×3 filters throughout, except for the final layer. Each layer except for the last uses leaky ReLU activation with $\rho = 0.1$. In total the 2x network contains about 550,000 trainable parameters, corresponding to 2 megabytes of 32-bit floating point numbers.

Layer	#1 (C)	#2 (C)	#3 (C)	#4 (C)	#5 (C)	#6 (C)	#7 (TC)
Filter depth	16	32	64	128	128	256	3
Kernel size	3×3	3×3	3×3	3×3	3×3	3×3	$2k \times 2k$
Strides	1×1	1×1	1×1	1×1	1×1	1×1	$k \times k$

Table 2: Network structure of *Vger* for k-factor super-resolution. C and TC refer to normal convolution and transposed convolution respectively.

By modifying the last layer, it is possible to run *Vger* at different magnification factors k. In this paper, we will be focusing on the cases $k = 2$ and $k = 4$. In the $k = 2$ configuration, *Vger* is fully compatible with *Waifu* and can import *Waifu* parameters.

5.1.1 Pre- and Post-processing

Before the first layers, the following pre-processing is applied; both during training and inference:

1. 8-bit integer samples are converted to 32-bit floating point,
2. Samples are scaled by a factor 1/255, mapping the range $[0, 255]$ to $[0, 1]$, and
3. Reflection padding (7 pixels wide) is applied to the image (see section 3.2.1).

After the last layer, the following post-processing is applied:

1. 7 pixels are cropped away on each side to remove reflection padding,
2. The output samples are clamped to the interval $[0, 1]$,
3. Samples are scaled by a factor 255, mapping the range $[0, 1]$ to $[0, 255]$, and
4. 32-bit floating point samples are converted to 8-bit integers.

The clamp operation is necessary because Tensorflow appears to perform floating point/integer conversion with overflowing arithmetic. Thus, an over-saturated floating point sample of 256.0 is converted to the integer 1 and not to 255 as would be more desirable. The clamp operation fixes this.

5.2 Training

The basic training iteration for k-factor scaling is as follows:

1. Select a batch of reference images y by choosing image patches from the training set in random order and applying augmentation (see Section 5.2.4),
2. Generate x by downscaling y by a factor k variously using bicubic and Lanczos filtering,
3. Generate the reconstructed image $\hat{y} = N(x)$, by feeding x into the *Vger* network,
4. Calculate the current loss $\mathscr{L}(\hat{y}, y)$ and gradient $\nabla_\theta \mathscr{L}(\hat{y}, y)$, and

5. Update the parameters based on the loss and gradient.

This is repeated for a fixed number of steps, logging the progress and saving intermediate parameters to disk at regular intervals.

5.2.1 Optimisation

Inspired by the implementation of *Waifu*, we choose a Huber loss function for training, setting $\delta = 0.1$. The same channel weighting as in *Waifu* is used before calculating the loss, namely:

$$(R', G', B') = 3 \cdot (0.299R, 0.587G, 0.114B). \tag{21}$$

The training process uses Adam optimisation with an initial step size of $\alpha_0 = 2.5 \cdot 10^{-4}$. Inverse-time annealing is used as defined by Equation 14, with $d = 0.2$ and the variable n interpreted as the current epoch number (starting from zero). Further, the annealing is 'stair-cased' so that the step size is constant throughout each epoch and only anneals when the epoch number is incremented.

The system is trained in two different steps: *base training* starting from random weights and training on varied data, and *scene training* which starts from an already base trained model and trains further on data from only one scene. The same base model may be duplicated and further trained into several different scene models.

5.2.2 Base training

For base training, the training set used is Mr Kou's photo collection [24], a freely downloadable set of about 6,500 pictures. The set contains pictures of buildings, furniture, food, and also a few graphics that don't appear to be camera pictures. Figure 14 shows a few samples of the set. The *Kou* pictures are quite high-resolution and contain a bit of sensor noise, neither of which is ideal. Therefore, the images are scaled and cropped down to 1000×1000 pixels before training.

Figure 14: Samples from Mr Kou's photo collection

5.2.3 Scene training

Scene training works the same way as the base training, except that the parameters are initialised from an existing base model instead of from random values. The model is trained on images of the same scene, captured at different times of day using a single camera. This goes against the conventional wisdom of machine learning which is to use many different images of diverse subjects. It is expected that this will result in a certain amount of *overfitting*, that is, decreased generality of the model.

Note in particular that the entire network is re-trained, using the same step size and annealing as for base training.

5.2.4 Data Augmentation

In machine learning, it is common to modify or 'augment' the data to simulate greater variety. This can include operations such as adjusting the hue, brightness, contrast, flipping the image along the vertical or horizontal axes, etc. Usually these are applied randomly and with random parameters. For *Vger*, we use only light augmentation in the form of:

1. Random cropping of smaller patches from the image,
2. Alternating between Lanczos and Bicubic filtering for the downscaling step,
3. Occasional flips along the horizontal and/or vertical axes, and
4. Occasional application of an unsharp mask filter with random strength.

These augmentations are applied to both the input image x and the reference image y uniformly.

5.2.5 Artefacts Removal

To train artefacts removal into the network, additional processing is applied to the input image after augmentation as described in Section 5.2.4. Since the idea is for the network to learn a correspondence between compressed, downscaled input images x and uncompressed, higher-resolution images y, compression should be applied only to the input image x. Note that the filter is now trained to perform two functions in combination: artefacts removal and super-resolution.

To simulate the situation that x is a frame from a compressed video stream, the input image is passed through a Unix pipeline consisting of two *ffmpeg* [25] processes. The first process encodes a frame of RGB samples to H.264 at a specified QP. This coded data is 'piped' to the second process which decodes it back to RGB samples. With this set-up, we can simulate any QP level desired, at a moderate additional cost in terms of processing speed. We will refer to this process as *QP training*.

5.2.6 Periodic Evaluation

To facilitate tuning of the hyperparameters a periodic evaluation step is useful. At regular intervals during training, we will run an inference step against a fixed *evaluation set* of images. Crucially, the system is *not* allowed to update the parameters during the evaluation step, instead the current loss (expressed as PSNR) is simply logged. Because the system is never allowed to train directly on the evaluation set, this remains a reasonably reliable indicator of training progress. If the evaluation step has reported about the same PSNR for many steps, this indicates that training is finished.

The evaluation set for *Vger* training is simply the first 40 images in the input dataset, by filename-alphabetical order. This means that the absolute magnitude of the performance has no particular meaning, since it will depend on the 'difficulty' of the evaluation images which are essentially randomly chosen. However it does mean that the evaluation set is constant between training sessions, which is useful. We may still interpret the relative evolution of the evaluated performance as a useful indicator of training progress.

5.3 Streaming Adapter

Super-resolution systems typically include routines for processing single PNG and JPEG images on disk. To facilitate streaming operation, *Vger* also contains a GStreamer [26] adapter implemented in C using the *libpython* interpreter library (see Appendix C for a brief overview of GStreamer). The streaming adapter uses Python routines to set up a reconstruction filter at a fixed resolution which can process plain RGB buffers passed from the C adapter. The adapter exposes the filter as an *element* within the GStreamer framework. For streaming operation, the batch size is fixed at 1 for simplicity.

The GStreamer toolkit includes numerous compatible elements which can be used to set up many different kinds of video pipelines with *Vger* without any further changes to the network itself; Figure 15 shows a few interesting possibilities. Such pipelines can be constructed either using simple Unix shell scripts or in C code. The shell script interface is sufficient for many uses, including all those described in Figure 15. The C interface supports certain more advanced features such as message passing between elements and error handling.

(a)	File reader → H.264 decoder → $Y'C_BC_R$/RGB → **Vger** → → RGB/$Y'C_BC_R$ → H.264 encoder → MKV muxer → File writer

(b)	RTSP client → H.264 decoder → $Y'C_BC_R$/RGB → **Vger** → Viewer application

(c)	RTSP client → H.264 decoder → 0.5x Lanczos scaler → [1] [1] → $Y'C_BC_R$/RGB → **Vger** → RGB/$Y'C_BC_R$ → Video crop, left half of frame → [2] [1] → 2x Lanczos scaler → Video crop, right half of frame → [3] [2, 3] → Video compositor → Viewer application

Figure 15: Example pipelines. (a) processes H.264 video from a file and saves the results to a separate file in encoded form. (b) enhances video from a network stream before displaying it on the screen. (c) shows a real-time A/B comparison between *Vger* and a Lanczos scaler working on a network stream. All elements of the pipeline except for *Vger* are standard, open-source components.

6 Evaluation

6.1 Scene-specific Reconstruction

To test the *Vger* reconstruction system, additional test images are captured. For each of *Elite* and *Office*, two new consecutive 24-hour timelapses of 1,440 images each are captured. The network is trained on the first day of footage and the second day is used as the test dataset. For *Construction*, about 12,000 images are captured in a span of about two months to use as the test dataset. The training dataset for *Construction* is the same as before (see Section 4.2.1). The same camera settings are used when capturing the test data and training data.

For each scene, 100 frames are selected at random to form the test dataset. Reconstruction performance is evaluated at 2x and 4x scaling factors for both the base and the specific model, as well as a baseline measurement using ImageMagick's Lanczos filter [23]. In each case, the reference image is downscaled using Lanczos filtering at the selected scaling factor and then upscaled using the model under test. Reconstruction quality is measured using SSIM.

The models were trained using the method described in Section 5.2. The base model was trained for 40 epochs and then used as the basis for scene-specific models. The scene models were trained for a further 300 epochs on the respective scene, except for the *Construction* model which was only trained for 20 epochs due to the larger size of the *Construction* dataset. This corresponds to about:

- 10 hours for *Elite* and *Office* 2x,
- 4 hours for *Elite* and *Office* 4x,
- 20 hours for *Construction* 2x, and
- 7 hours for *Construction* 4x.

Table 3 shows the average performance for each scene. Figures 16–17 show selected crops from the output, highlighting visual differences between the different models.

2x	Lanczos		Vger Base		Vger Scene-specific		ΔLS	ΔBS
	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM μ
Elite	0.9133	0.0281	0.9288	0.0247	0.9322	0.0233	0.0189	0.0034
Office	0.9302	0.0318	0.9472	0.0320	0.9495	0.0325	0.0193	0.0023
Construction	0.9178	0.0393	0.9427	0.0271	0.9545	0.0206	0.0366	0.0118

4x	Lanczos		Vger Base		Vger Scene-specific		ΔLS	ΔBS
	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM μ
Elite	0.7327	0.0818	0.7587	0.0793	0.7672	0.0808	0.0345	0.0085
Office	0.7627	0.0623	0.8098	0.0798	0.8221	0.0906	0.0593	0.0123
Construction	0.7848	0.0872	0.8136	0.0779	0.8440	0.0618	0.0592	0.0304

Table 3: Performance measurements for 2x and 4x super-resolution showing SSIM mean μ and standard deviation σ. ΔLS and ΔBS are the mean performance improvements for the scene-specific model compared to Lanczos filtering and the base model, respectively.

For *Construction*, scene-specific training shows significant improvement for both 2x and 4x super-resolution. For *Elite* and *Office* improvement is more subtle for 2x, but for 4x a clear difference is seen. The scene models seem especially successful at drawing the fixed edges of the scene sharply without the overshoot that is typical of Lanczos filters. Performance on the *Construction* scene was decidedly more impressive than on the other scenes, which may reflect the fact that the training dataset was much larger for this scene.

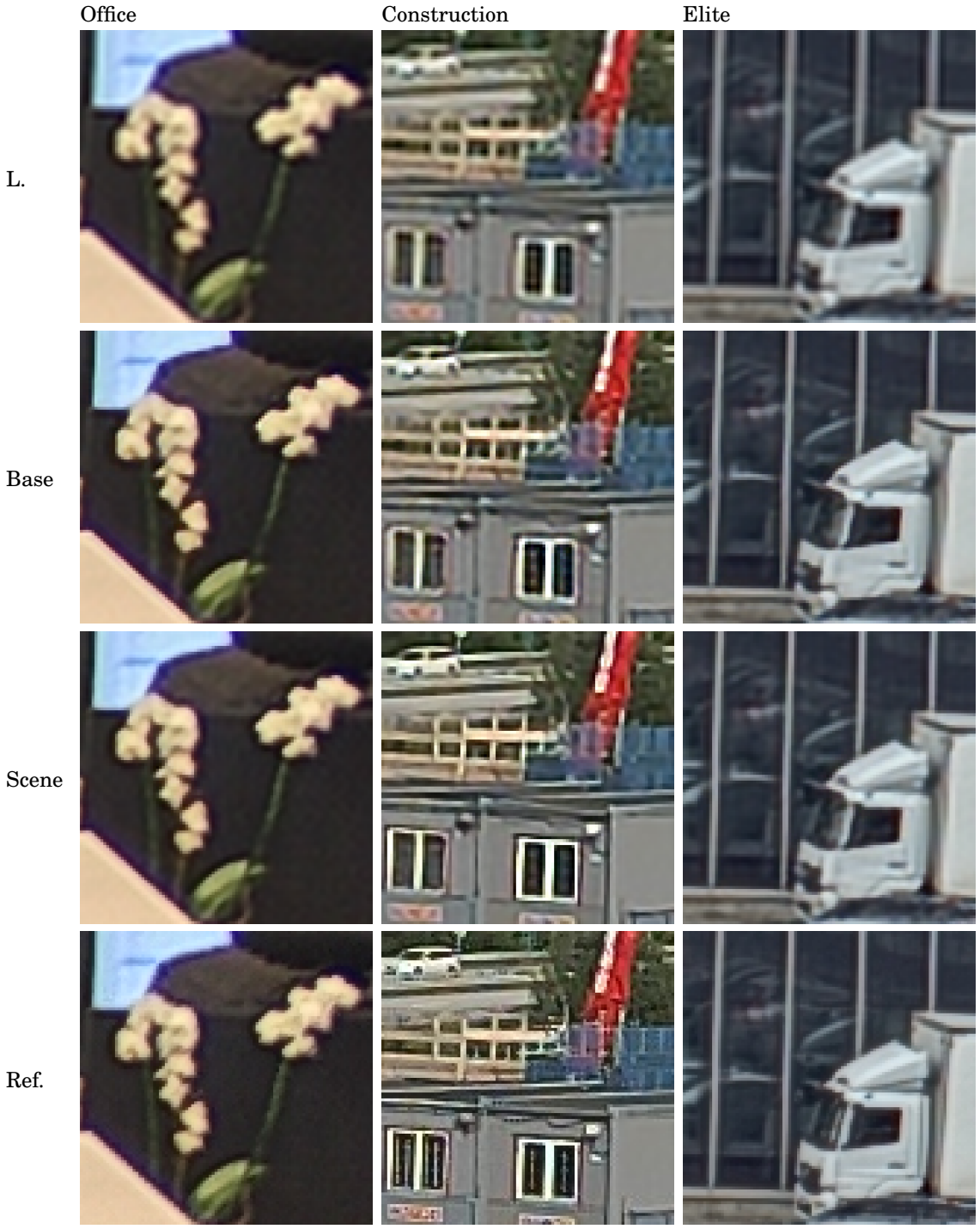


Figure 16: 2x reconstruction, 100×100 pixel samples. From the top: Lanczos filter, Vger base model, Vger scene-specific model, reference image.

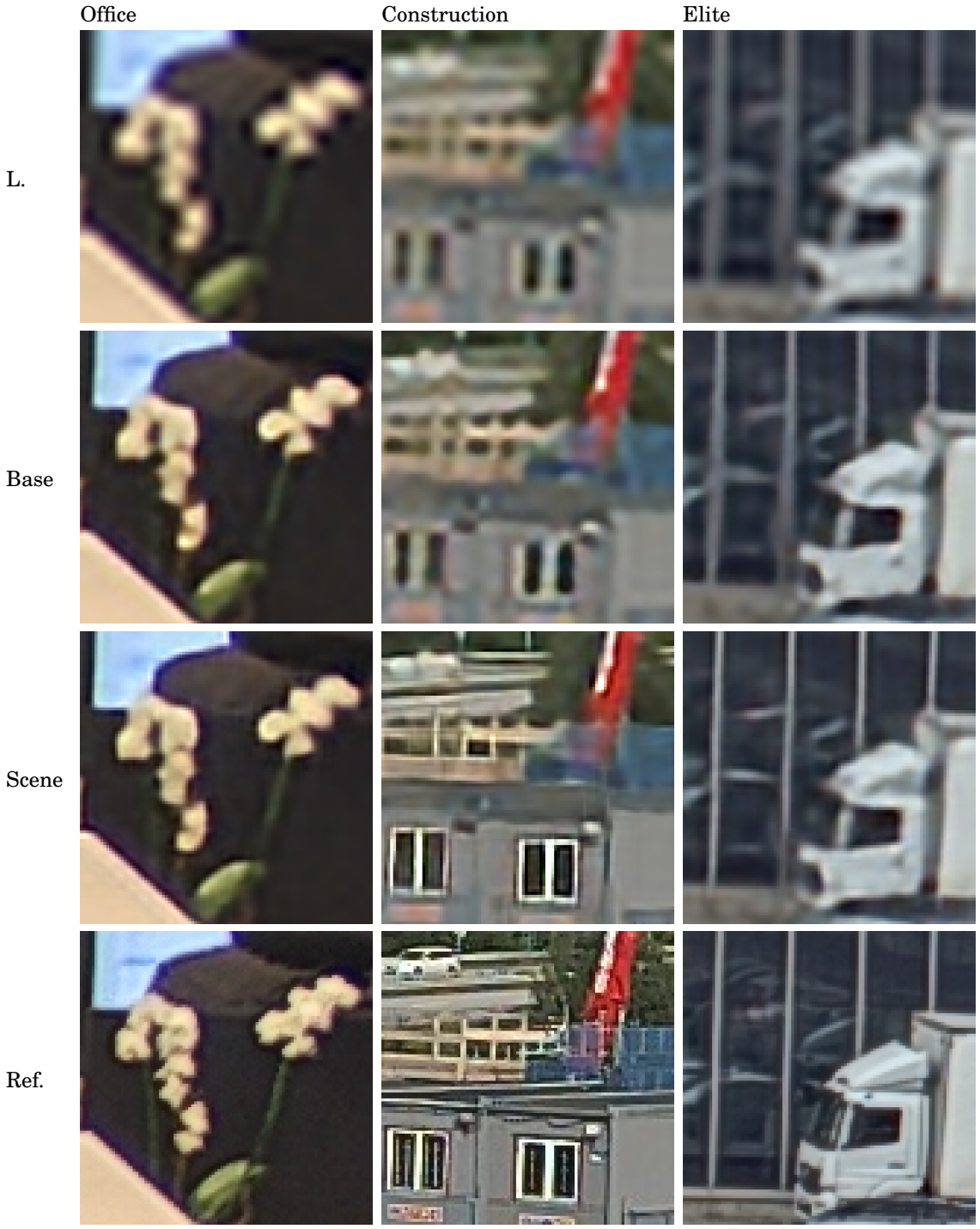


Figure 17: 4x reconstruction, 100×100 pixel samples. From the top: Lanczos filter, Vger base model, Vger scene-specific model, and reference image.

6.2 Unfamiliar Scenes

We think of scene-specific training as the model 'adapting' to new image characteristics. But what aspect of the image is the model actually adapting to? There are at least two major factors which possibly need adapting to: the physical scene that the camera is viewing and the characteristics of the capture device itself (including capture settings). At this point we cannot know if the performance gain seen in Section 6.1 is due to the model adapting to the scene, or if the model is simply 'learning' the characteristics of the camera regardless of the scene.

To improve our understanding, a new timelapse is captured using the same camera and camera settings as in the *Elite* scene, but the camera is moved to view a different scene. This scene is then reconstructed using the previous model trained on the *Elite* scene.

2x	Lanczos		Vger Base		Vger Elite		ΔLS	ΔBS
	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM μ
	0.8819	0.0264	0.9045	0.0241	0.9065	0.0208	0.0246	0.0019

4x	Lanczos		Vger Base		Vger Elite		ΔLS	ΔBS
	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM σ	SSIM μ	SSIM μ
	0.6419	0.0596	0.6710	0.0614	0.6734	0.0615	0.0314	0.0024

Table 4: Performance measurements for 2x and 4x super-resolution on an unfamiliar scene showing mean value μ and standard deviation σ. ΔLS and ΔBS are the mean performance improvements for the scene-specific *Elite* model compared to Lanczos filtering and the base model, respectively.

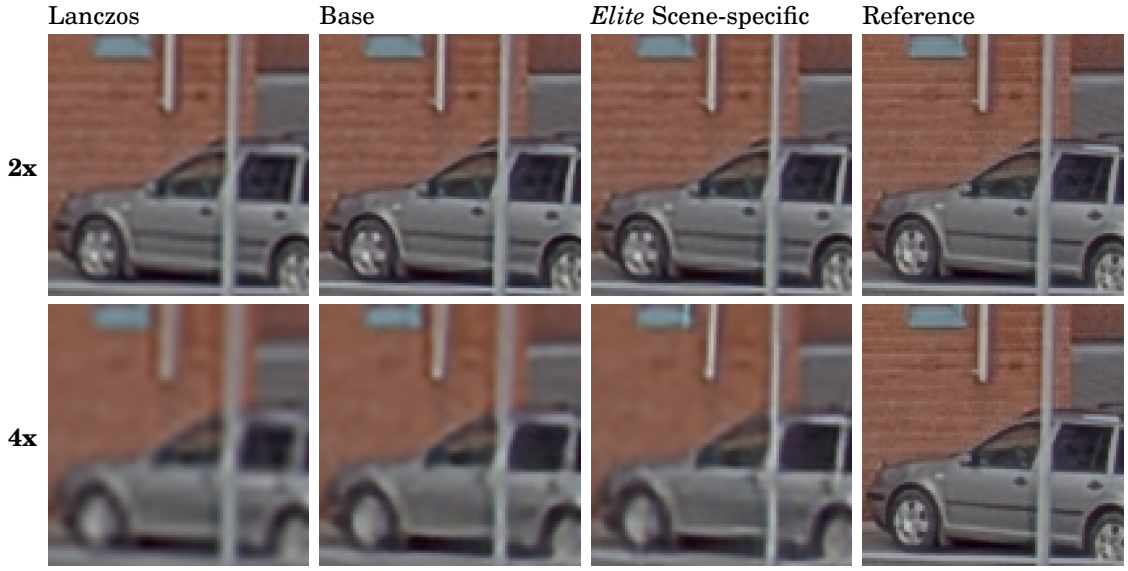


Figure 18: 2x and 4x reconstruction of an unfamiliar scene, 100 × 100 pixel samples. From the left: Lanczos filter, Vger base model, Vger *Elite* scene-specific model, reference image.

Table 4 shows the reconstruction performance measured with SSIM. There is still some improvement compared to the base model, even though the model has trained on a different scene than the one being reconstructed. This is confirmed by looking at the samples in Figure 18 and especially the case of 4x reconstruction. This indicates that at least part of the improvement seen in scene-specific training may come from the model adapting to the camera, rather than the scene.

6.3 Artefacts Removal

To evaluate the relative benefit of QP training for artefacts removal as described in Section 5.2.5, a set of different models are trained. For each of the QP values $[0, 28, 32, 36, 40, 44, 48]$, both 2x and 4x scaling models are trained (we use the shorthand QP = 0 to refer to training without any compression simulation). In each case a base model is trained from scratch, as well as a model specific to the *Elite* scene (see Section 4.2.1). Each base model is trained for 40 epochs, then duplicated and used as the basis for a scene-specific model which is trained for a further 10 epochs.

For evaluation, 90 consecutive frames of the *Elite* scene are produced by extracting the first frames from the test video described in Appendix A. This test video is then re-encoded to H.264 at a range of different QP values in the interval $[24, 50]$ using the same method as described in Appendix A. For each video QP value, the video is downscaled using *ffmpeg's* Lanczos filter and then reconstructed using each model in the set separately. The mean SSIM performance as a function of video QP and training QP is shown in Figures 19–21.

6.3.1 Relative Benefit of QP Training

To interpret Figures 19–21, note that the black contours correspond to areas of constant quality as measured by SSIM. The left-most point of each contour is the highest video QP which can be reconstructed to a given quality. Eventually, each contour curves to the right since for high enough video QP effective reconstruction becomes impossible. If the contour curves to the left before curving to the right, it means that there is a benefit to using the higher-QP trained models at this particular quality level. Also marked in each chart is the optimum model for each video QP level. The table below each chart shows the improvement over baseline training (without any QP simulation). Where the value is positive, there was a measurable benefit to using the QP training process.

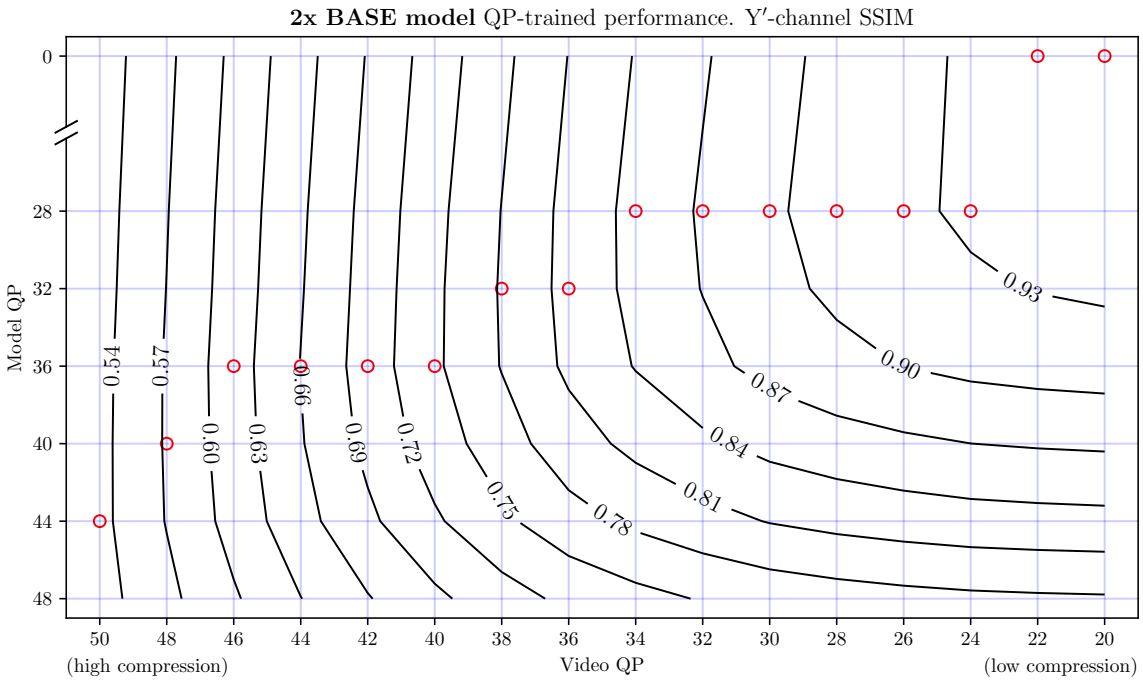
Note that the optimum model QP seems to lie a few points below the video QP in all tested cases. Note also that the improvement was significantly greater for the case of scene-specific training. For 2x reconstruction (Figure 19), note that the performance of the scene-specific model is slightly inferior to the base model for very low QP values. This result will be discussed further in Section 7.2.1.

6.3.2 Visual Effects of QP Training

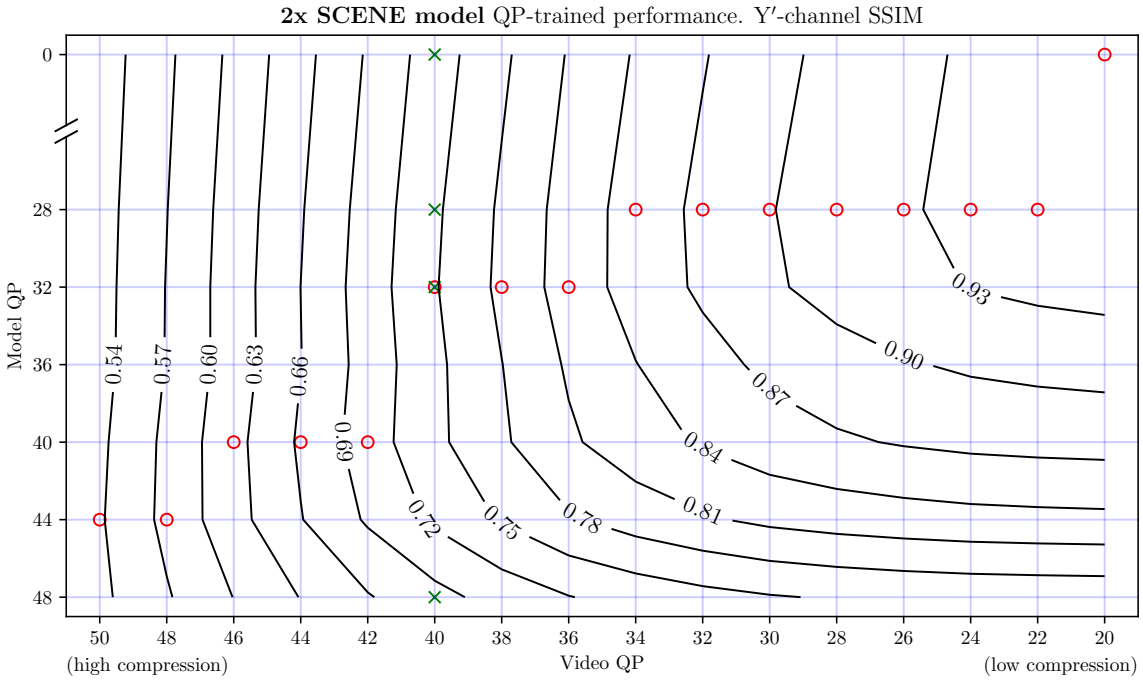
Figure 20 shows samples of the *Elite* scene encoded at QP = 40 and reconstructed with different models. We see that high-QP training seems to correspond to reduced risk-taking, as those models produce smoother output. However, it appears that this smoothness comes at the cost of lost detail.

From Figure 20, it seems clear that at such a high QP, reconstruction without any artefacts removal is not a good option. We see that the non-QP trained model reconstructs the image with severe noise and artefacts. This is to be expected, since the model has been trained on very lightly compressed images. When such a model encounters quantisation noise, it understandably 'interprets' the noise as true edges to be enhanced.

We also see the effects of using too high QP for training. The model with QP = 48 has managed to avoid any artefacting, but at the cost of blurring the image severely. Note especially how some the window reflections in the middle column of Figure 20 are almost lost when using this model.



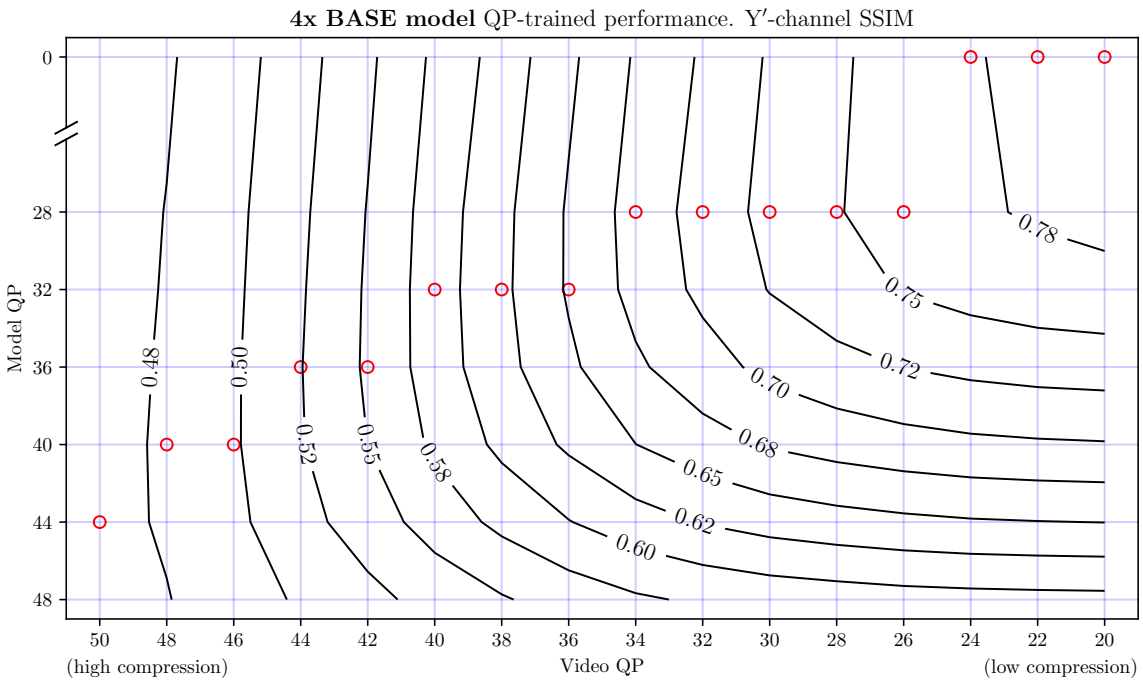
Video QP	50	48	46	44	42	40	38	36	34	32	30	28	26	24	22	20
Optimum model QP	44	40	36	36	36	36	32	32	28	28	28	28	28	28	0	0
Baseline SSIM	.5244	.5640	.6063	.6491	.6918	.7341	.7727	.8107	.8416	.8668	.8902	.9085	.9225	.9339	.9427	.9488
Optimum SSIM	.5323	.5727	.6167	.6607	.7038	.7451	.7825	.8194	.8489	.8734	.8954	.9119	.9245	.9347	.9427	.9488
Improvement	.0078	.0086	.0104	.0116	.0119	.0110	.0098	.0087	.0072	.0065	.0051	.0033	.0020	.0007	.0000	.0000



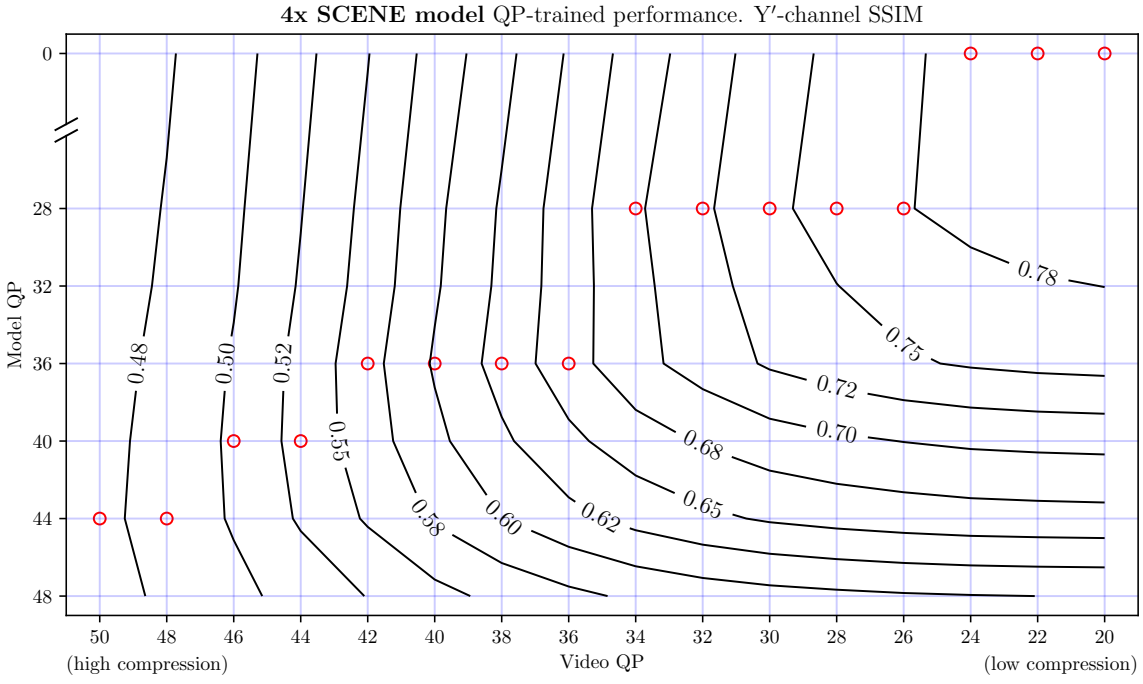
Video QP	50	48	46	44	42	40	38	36	34	32	30	28	26	24	22	20
Optimum model QP	44	44	40	40	40	32	32	32	28	28	28	28	28	28	28	0
Baseline SSIM	.5245	.5644	.6072	.6502	.6931	.7355	.7743	.8122	.8427	.8678	.8909	.9087	.9225	.9339	.9425	.9484
Optimum SSIM	.5369	.5776	.6211	.6642	.7055	.7477	.7864	.8234	.8526	.8768	.8985	.9147	.9270	.9370	.9439	.9484
Improvement	.0123	.0132	.0138	.0140	.0123	.0121	.0121	.0112	.0098	.0090	.0075	.0059	.0045	.0031	.0014	.0000

Figure 19: 2x reconstruction performance as a function of video and training QP. Grid intersections correspond to tested combinations. Red circles mark the optimum model QP for each video QP. Green crosses mark the combinations shown in Figure 20. 'Baseline SSIM' refers to the model with QP = 0, i.e. no compression simulation during training.

Figure 20: Sample 2x reconstructions of heavily compressed images using the same method as in Figure 19. The reference image has been downscaled by a factor 2 using Lanczos filtering, then H.264-encoded at QP = 40. The figure shows 2x reconstructions with various scene-trained models. Note that model QP = 32 is suggested by Figure 19 to be optimal for video QP = 40.



Video QP	50	48	46	44	42	40	38	36	34	32	30	28	26	24	22	20
Optimum model QP	44	40	40	36	36	32	32	32	28	28	28	28	28	0	0	0
Baseline SSIM	.4606	.4723	.4895	.5153	.5451	.5794	.6099	.6448	.6775	.7031	.7275	.7463	.7610	.7728	.7822	.7883
Optimum SSIM	.4678	.4783	.4972	.5241	.5535	.5880	.6197	.6526	.6851	.7094	.7324	.7485	.7616	.7728	.7822	.7883
Improvement	.0071	.0059	.0076	.0088	.0083	.0086	.0097	.0077	.0075	.0063	.0048	.0021	.0005	.0000	.0000	.0000



Video QP	50	48	46	44	42	40	38	36	34	32	30	28	26	24	22	20
Optimum model QP	44	44	40	40	36	36	36	36	28	28	28	28	28	0	0	0
Baseline SSIM	.4602	.4725	.4903	.5175	.5490	.5845	.6171	.6526	.6864	.7127	.7378	.7563	.7711	.7828	.7921	.7981
Optimum SSIM	.4706	.4823	.5042	.5333	.5662	.6027	.6345	.6651	.6965	.7210	.7443	.7606	.7734	.7828	.7921	.7981
Improvement	.0104	.0097	.0139	.0157	.0172	.0181	.0174	.0124	.0100	.0083	.0065	.0042	.0023	.0000	.0000	.0000

Figure 21: 4x reconstruction performance as a function of video and training QP. Grid intersections correspond to tested combinations. Red circles mark the optimum model QP for each video QP. 'Baseline SSIM' refers to the model with QP = 0, i.e. no compression simulation during training.

6.4 Training Characteristics

Figure 22 shows the training progress of the network as measured during the evaluation step. The absolute value of the 'performance' metric is not meaningful, because it depends strongly on the 'difficulty' of the particular evaluation set used. There is no way to be sure of when training is finished, rather one has to study the diagram and try to make an educated guess.

The models are trained on a single GTX 1080 GPU, a moderately high-end chip at the time of writing. This system can train an epoch of Vger in about 8 minutes for base training and 2 minutes for the *Elite* scene, running at about 12 examples per second.

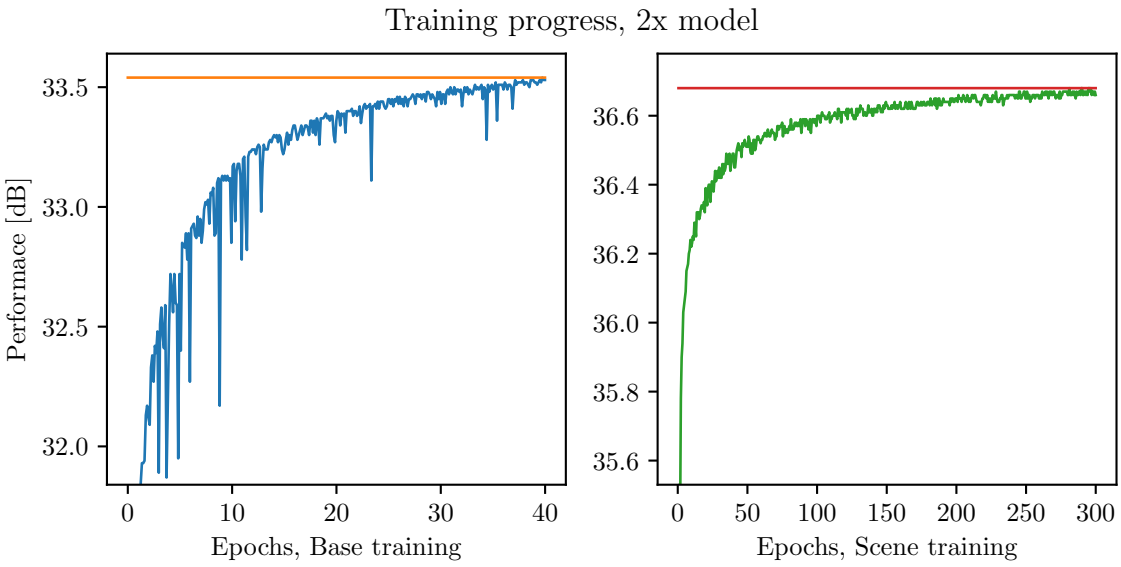


Figure 22: Base and scene-specific training on the *Elite* scene. Signal-to-noise performance is measured using the weighted Huber loss described in Section 5.2.6. Note that training and evaluation sets are different for base and scene training, so performance measurements are not directly comparable; neither are epochs the same size. Horizontal lines mark the maximum performance reached.

6.5 Processing Speed

To evaluate the processing speed, we perform streaming reconstruction at several different resolutions and measure the average processing speed in frames per second. A 5 megapixel camera is used which has a built-in hardware scaler (using a fixed-point implementation of the Lanczos filter) and thus can output video at multiple different resolutions. The network runs on a GTX 1080 with 8 GB graphics memory. FPS is measured using a timing routine inserted into the GStreamer adapter and calculated as the quotient $1/T$ where T is the average time the network takes to process a single frame of a given resolution. Figure 23 shows the results of this test.

4x processing runs slightly slower than 2x which is not surprising given that image buffers increase in size by a factor 4 when running in 4x mode compared to 2x. Otherwise, processing speed seems to be stable once a resolution of about 0.3 megapixels is reached. The maximum rate reached was about 3.1 input megapixels per second, corresponding to about six frames per second when running $800 \times 600 \rightarrow 1600 \times 1200$ super-resolution. The maximum resolution processable within the memory constraints of the GTX 1080 was $1600 \times 1200 \rightarrow 3200 \times 2400$ for 2x super-resolution and $1280 \times 960 \rightarrow 5120 \times 3840$ for 4x.

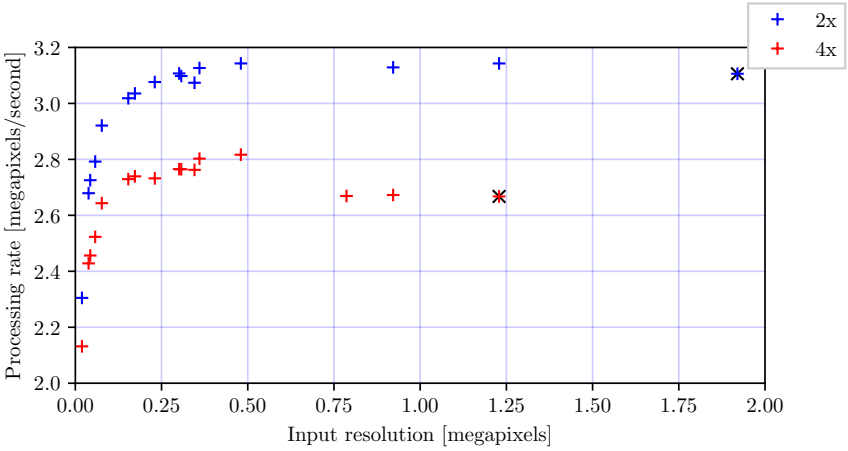


Figure 23: Processing rate of *Vger*, measured on the input side. Black crosses mark the highest resolution the system was able to process in each mode before running out of GPU memory. Note that the input to the system is capped at 29.97 frames per second.

7 Conclusion

7.1 Fulfilment of Goals

In Section 1.4, several project goals were identified. These goals will now be addressed.

Goal 1: Scene-specific reconstruction performance. Performance on the *Construction* scene was improved by 0.01 points of SSIM for 2x reconstruction and 0.03 for 4x, meeting or exceeding the set goal of 0.01 point improvement. This improvement was confirmed by visual inspection. On the other scenes, improvement was truly significant only for 4x reconstruction. In all cases, the method significantly outperformed Lanczos filtering both in SSIM measurements and by visual comparison.

Goal 2: On-line training. The *Construction* model took about 54 hours to train for only 20 epochs due to the large size of the training set. The *Elite* 4x model by contrast took only about 4 hours to train for 300 epochs. However, the *Elite* model did not perform nearly as well. Furthermore, capturing training images for *Elite* required 24 hours before training began. We conclude that the target of reaching peak performance in 'a few hours' of on-line training was not reached.

Goal 3: Processing speed. On a typical graphics workstation, *Vger* was able to process 3 megapixels per second measured on the input side, exceeding the set goal. This corresponds to 2x reconstruction of 1920×1080 pixel video at six frames per second, about a factor four less than what would be considered fully real-time. The 2x reconstruction filter was able to reconstruct at a maximum resolution of 3200×2400 before running out of GPU memory, also exceeding the set goal.

Goal 4: Unfamiliar scenes. Switching to an unfamiliar scene reduced performance compared to the scene-specific model and even be base model, but not below the baseline of Lanczos filtering specified in the project goals.

Goal 5: No significant misrepresentation of the input image. We have not seen any significant transgressions in this respect. The Huber loss of *Vger* appears to take a sufficiently conservative approach to inference, as shown in Figures 16–17.

7.2 Error Sources and Test Methodology

7.2.1 Training Duration

The 'artefacts removal' evaluation in Section 6.3 was performed with scene models that were trained for only 10 epochs, compared to the 'reconstruction performance' evaluation which used models that were trained for significantly longer. It was realised only after training had completed that 10 epochs of scene training is too short, leading to underwhelming performance. Due to the high cost in time of re-training so many different models, the test was not repeated. This only affects the 'scene-specific' tests, not the 'base' tests. In any case, the test should still be mostly valid for its main purpose of showing relative trends in QP training for a specific factor and model type.

7.2.2 Synthetic Similarity Measurements

We have used SSIM and PSNR to measure the similarity between images and thus the performance of a reconstruction method. These mathematical measures are *synthetic* in the sense that they rely on simplified statistical and arithmetic models to describe the complex phenomenon of human vision. Use of such synthetic measures is useful because it facilitates data analysis and visualisation, but the approximation of real human vision is coarse at best. One must been careful to always confirm synthetic measurements by visual inspection of select samples.

7.2.3 Compressed and Chroma Subsampled Test Video

All of the test video used in this thesis was captured with off-the-shelf IP cameras in realistic capture modes. This means that the video is both (a) compressed/quantised and (b) chroma subsampled. Effort has been made to capture the video at the highest quality possible, but the video is not captured in a raw or untouched format. This is seen as acceptable, because we are evaluating techniques for use in realistic scenarios. The *Elite* scene is captured using a 5 megapixel, 30 frames per second camera which would produce 3.6 Gbits/s of data if video were to be captured in uncompressed 24-bit RGB format. Clearly this would not be a realistic scenario, however it is something that should be considered and recognised as a possible source of error. Certain tests in this study involve compressing the video a second time and the effects of repeated compression should be considered. All tests involve upscaling and downscaling, which may exacerbate the effects of chroma subsampling. Note however that since we measure SSIM only on the Y' channel (which is never subsampled), SSIM measurements should be less affected by chroma subsampling.

7.3 Implementation Suggestions

7.3.1 Reconstruction

Using CNN upscaling with a general 'base' model is better than using only Lanczos filtering, and scene-specific training can significantly improve performance further in at least some cases. A base model can be used as a starting point for scene-specific training. It will likely suffice to perform a training step once per minute (as we did for the *Elite* scene) or once every five minutes (as we did for the *Construction* scene). Training over long periods of time (as we did with the *Construction* scene) appears to give better results than training on only a day's worth of footage (as with the *Elite* scene). For 'live' training, random crops (as described in Section 5.2.4) could be omitted, instead the full frame could be divided into suitable patches to train on, using neighbouring patches as padding instead of even reflection.

If true scene-specific training is not possible, it may still be beneficial to train a model for the particular camera to be used. Presumably, this would work best if the training scene is at least somewhat similar to the scene to be reconstructed.

There needs to be a mechanism for evaluating the performance of the current model because if either the scene or camera settings change significantly, training will likely have to start over from the base model again. A possible alternative would be to reset the step size annealing (see Section 3.5.4) to let the current model be re-trained. A periodic evaluation step on the current frame using the SSIM metric could be used to estimate current performance. A large negative change over several minutes could mean that the scene has changed and re-training is necessary.

7.3.2 Artefacts Removal

Compression effects can likely be ignored up to at least QP = 24 for 2x reconstruction and QP = 28 for 4x. Above this level, reconstruction will start to suffer unless the model can account for compression artefacts. In this project, a separate model was trained for every 4 QP values for a total of 7 models including the QP = 0 model. Judging by Figures 19–21, it might suffice to train even fewer models. Video should be reconstructed using a model trained for several QP steps less than the video QP. It is better to use a slightly too aggressive model than a too conservative one. One possible approach would be to train models at QPs 0, 25, 30, 35, and 40; selecting the highest-QP model which is at least 5 QP values below the target video QP.

7.3.3 Camera vs. Host

In this project, all work was done on the host or viewer system, with the camera only providing the video stream. It would be interesting to explore if the training process could be run on the camera itself, with the host computer only performing the inference pass. The camera could incrementally update the model as time goes by, using a built-in hardware scaler to generate training images from what the camera is currently viewing.

7.4 Further Work

Simplification of the network. Some steps in the *Vger* system are suspected to be less useful. Investigation is warranted to determine whether these steps actually contribute to network performance or could be omitted:

- Re-scaling the image samples before and after the neural network (Section 5.1.1),
- Weighting of the RGB channels before calculating the Huber loss (Section 5.2), and
- Data augmentation by using different downscaling filters (Section 5.2.4).

Other modifications to the network. There are several possible modifications which could conceivably improve performance or processing speed, such as:

- Replacing the final transposed convolution with a sub-pixel layer as described by Shi et al. [21].
- Processing video directly in $Y'C_BC_R$ colour format rather than RGB, removing the need for conversions when the source format is H.264 video.
- Processing only the luma (Y') channel of images in the neural network and using a separate e.g. Lanczos filter to upscale the chrominance components. Such a simplification was shown by Dong et al. [6] to cause only moderate performance loss in their network.
- Replacing the leaky ReLU activation function (Section 3.3.2) with a *parametric ReLU* (PReLU) function for which the amount of negative 'leak' is a learnable parameter. Experiments by Xu et al. [27] suggest that PReLU may enable increased performance over leaky ReLU.

Different loss function. For this project, a conservative loss function was chosen to help ensure that the reconstruction filter did not significantly misrepresent the scene. It is possible that a more aggressive loss function could also meet this requirement, while providing more detailed output.

Temporal filtering. The *Vger* network considers only a single frame of video at a time. It is likely that better performance could be achieved using temporal methods, which consider multiple consecutive frames at once. See for instance Caballero et al. [28].

Streaming training. A 'streaming' mode of training, where the network re-trains in parallel with normal processing, was described in the introduction of this thesis. Such a mode has not been implemented in this project, but there is no apparent technical reason why it would not be possible.

Regions of interest. It was hinted in the introduction that a ROI system of prioritised image sections could be combined with the techniques described in this paper. Such a system could transmit most of the image in low resolution (to be reconstructed using a system like *Vger*) but select certain regions to be transmitted at full resolution. Live training would be possible only on the ROIs, unless training was performed on the camera as outlined in Section 7.3.3.

8 References

[1] ITU-T H.264: Advanced video coding for generic audiovisual services, 2017. Edition 12.0.

[2] ITU-R BT.601-7: Studio encoding parameters of digital television for standard 4:3 and widescreen 16:9 aspect ratios, 2011.

[3] Charles Poynton. *Digital Video and HDTV: Algorithms and Interfaces*. Morgan Kaufmann Publishers, 2003.

[4] Axis zipstream technology: More video, less storage. https://www.axis.com/files/whitepaper/wp_zipstream_71042_en_1709_lo.pdf, 2017.

[5] Claude E. Duchon. Lanczos filtering in one and two dimensions. *Journal of Applied Meteorology*, 18(8):1016–1022, 1979.

[6] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *Proceedings of European Conference on Computer Vision*, 2014.

[7] Chao Dong, Chen Change Loy, and Xiaoou Tang. Accelerating the super-resolution convolutional neural network. In *Proceedings of European Conference on Computer Vision*, 2016.

[8] Justin Johnson, Alexandre Alahi, and Fei-Fei Li. Perceptual losses for real-time style transfer and super-resolution. *CoRR*, abs/1603.08155, 2016.

[9] Mehdi S. M. Sajjadi, Bernhard Schölkopf, and Michael Hirsch. Enhancenet: Single image super-resolution through automated texture synthesis. *CoRR*, abs/1612.07919, 2016.

[10] Ryan Dahl, Mohammad Norouzi, and Jonathon Shlens. Pixel recursive super resolution. *CoRR*, abs/1702.00783, 2017.

[11] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[12] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. In *Advances in Neural Information Processing Systems*, volume 27, 2014.

[13] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4), 2004.

[14] Andrej Karpathy and various contributors. CS231n: Convolutional neural networks for visual recognition. http://cs231n.github.io/. Stanford University course notes; accessed 17 Oct 2017.

[15] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv:1603.07285*, 2016.

[16] Peter J. Huber. Robust estimation of a location parameter. *Annals of Mathematical Statistics*, 35(1):73–101, 1964.

[17] Zhou Wang and Alan C. Bovik. Mean squared error: Love it or leave it? a new look at signal fidelity measures. *IEEE Signal Processing Magazine*, 26(1), 2009.

[18] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *PMLR*, pages 249–256, 2010.

[19] Nagadomi and various contributors. Waifu2x: Image super-resolution for anime-style art. https://github.com/nagadomi/waifu2x. Version 0.13.1-41-ga1458de from 14 Aug 2017.

[20] Kosuke Nakago. Seranet: Super resolution of picture images using deep learning. https://github.com/corochann/SeRanet. Version 6613128a from 24 Feb 2017.

[21] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P. Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. *CoRR*, abs/1609.05158, 2016.

[22] Alex J. Champandard and various contributors. Neural Enhance: Super resolution for images using deep learning. https://github.com/alexjc/neural-enhance. Version 0.3-8-g2fd67de from 30 Nov 2016.

[23] ImageMagick Studio LLC. Imagemagick software tool. http://imagemagick.org/. Version 6.8.9 from 31 July 2017.

[24] Mr Kou's photo collection. http://photosku.com/photo/category/%E6%92%AE%E5%BD%B1%E8%80%85/kou/.

[25] FFmpeg Developers. ffmpeg software tool. http://ffmpeg.org/, Version 3.4-dev-2253-g6f15f1c from 20 Sep 2017.

[26] GStreamer Team. Gstreamer software tool. https://gstreamer.freedesktop.org/. Version 1.12.0-137-gf649f0a from 17 September 2017.

[27] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *CoRR*, abs/1505.00853, 2015.

[28] Jose Caballero, Christian Ledig, Andrew P. Aitken, Alejandro Acosta, Johannes Totz, Zehan Wang, and Wenzhe Shi. Real-time video super-resolution with spatio-temporal networks and motion compensation. *CoRR*, abs/1611.05250, 2016.

Appendices

A Test Setup for Figure 2

300 frames of video were captured using an Axis Q1647 camera, encoding H264 at 3072×1728 pixels and QP = 14, of the *Elite* scene shown in Figure 25. The video was decoded to raw $\mathrm{Y'C_BC_R}$frames, cropped to 1500×1500 and saved on disk. The raw video was then encoded using varying QP and resolution using FFmpeg [25] and the libx264 software encoder. Lanczos filtering was used to downscale to the target resolution before coding, and after decoding to upscale back to the reference resolution again. In order to loosely simulate an IP camera, the following encoding parameters were chosen:

- B-frames disabled
- Constant QP, same for both I-frames and P-frames
- GOP length (keyframe interval) set to 60

The average SSIM over the $\mathrm{Y'}$channel (compared to the original video) and video bitrate were calculated using the built-in tools available in FFmpeg.

B Sample Images of the Test Scenes

Figure 24: Sample frame of the *Construction* scene.

Figure 25: Sample frame of the *Elite* scene.

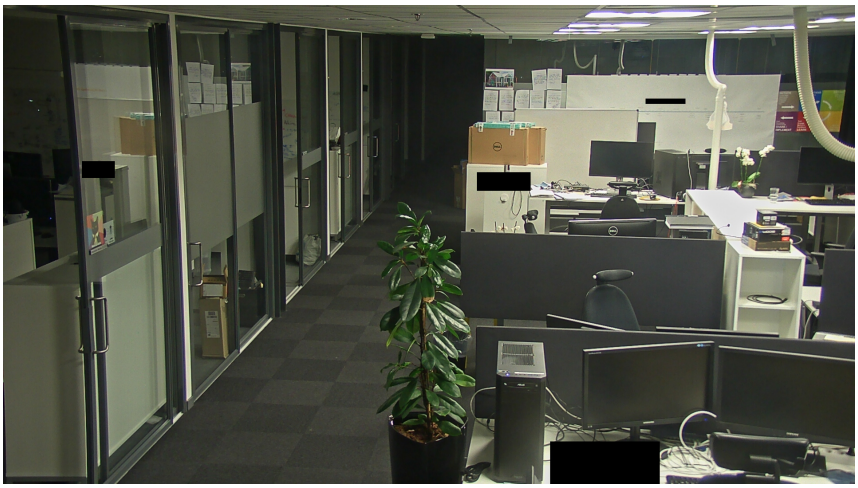


Figure 26: Sample frame of the *Office* scene. Some features have been covered.

C Brief Overview of GStreamer

The GStreamer framework [26] views a multimedia pipeline as a graph of *elements*. An element is conceptually a black-box process which may contain any number of connections, or *pads*. A pad is either an input *sink pad* or an output *src pad*. Most elements fall into the categories of either

- *sources* which produce output without accepting input,
- *sinks* which accept input but do not produce any output, or
- *filters* which accept input, apply some processing, and emit the results as output.

An example of a source would be a file reader. An example of a sink would be a file writer. An example of a filter would be a video decoder or an image reconstruction system like *Vger*.

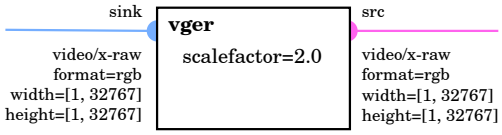


Figure 27: The *Vger* element before negotiation. It accepts and emits only raw RGB video. The video resolution is currently unconstrained, which GStreamer represents as the interval [1,32767].

Each pad has a set of capabilities or *caps* which constrain the type of media they can accept, or will emit in the case of src pads. In the case of a raw video stream, the caps may include such constraints as colour format, resolution, and framerate. Two connected pads form an edge in the graph and the pads are required to have compatible caps, in the sense that there exists a common format which satisfies the constraints of both caps. Each element also has a number of configurable processing parameters or *properties* which may affect the supported caps.

A *capsfilter* is a special passthrough element which is inserted between two pads solely to impose additional constraints on the format. For instance, the *videoscale* element can scale between many different resolutions so a capsfilter element might be necessary to select the desired resolution.

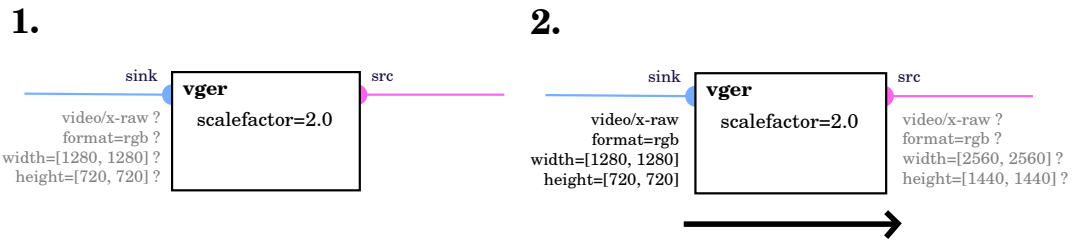


Figure 28: The *Vger* element accepts the proposed input resolution on the sink pad and pushes a proposal for the resulting output resolution on the src pad.

After the pipeline is constructed, connected elements will start negotiating the format to use on each edge. An element may 'propose' caps to a neighbouring element, which may in turn accept or reject the proposal. If accepted, the neighbouring element adjusts the caps on its other pads to match the new parameters and in turn propagates these changes as further proposals to its neighbours. The negotiation flows back and forth through the network in this manner until either the format is fixed on all edges, in which case the pipeline may start playing; or two elements fail to agree on a compatible format for an edge, in which case the pipeline fails. In more complex scenarios, the format may be dynamic and require the caps to be re-negotiated or connections modified even as the pipeline is playing. One example would be an automatic decoding element like GStreamer's *decodebin* for which the output format and even the number of output pads is dependent on the content of the input data.

Once the pipeline is playing, data flows between elements through each edge in a streaming fashion. All elements are synchronised to a common clock, which can be used to implement play/pause and seek functionality. The elements also share a message bus which can be used to signal events such as 'end of stream'.