Matthew Thomas Jackson

Real-time video super-resolution

Computer Science Tripos – Part II

Gonville and Caius College

May 7, 2020

Declaration of Originality

I, Matthew Thomas Jackson of Gonville and Caius College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I, Matthew Thomas Jackson of Gonville and Caius College, am content for my dissertation to be made available to the students and staff of the University.

Signed 

Date Thursday 7th May, 2020

Acknowledgements

I would like to thank Jin Zhu, for sharing his knowledge of super-resolution and helping me discover relevant research for this dissertation, and Professor Pietro Liò, for his advice regarding the project's direction, in addition to the structure and format of this report.

I would also like to thank my Directors of Studies, Dr Tim Jones and Dr Graham Titmus, for their advice regarding this dissertation and throughout Tripos.

Finally, I would like to thank my parents for their continued support throughout my education.

Proforma

Candidate:	**2358G**
Project Title:	**Real-time video super-resolution**
Examination:	**Computer Science Tripos — Part II, June 2020**
Word Count:	**11732**[1]
Line Count:	**2138**[2]
Originator:	2358G
Supervisors:	Prof Pietro Liò & Mr Jin Zhu

Original Aims of the Project

The aim of this project was to explore recent advances in neural-network-based video super-resolution, empirically evaluating the impact of the proposed techniques on real-time performance. After completing the evaluation of individual techniques on a baseline model, an improved model was to be designed. The extension aims were to implement a recurrent, video-based architecture and develop a package for out-of-box video super-resolution.

Work Completed

The success criteria were met and exceeded, with both extensions being implemented (an original extension was dismissed due to a change in project direction). Seven techniques across four categories — post-upsampling, residual learning, activation functions and frame-recurrence — were implemented and evaluated. Finally, a package was designed to execute the pre-trained models on low-resolution video, while adapting to the real-time performance requirements of the user.

Special Difficulties

None.

[1]The word count was computed using TeXcount, `https://app.uio.no/ifi/texcount/`.
[2]The line count was computed by `git ls-files | xargs wc -l`.

Contents

List of Figures

List of Tables

Abbreviations

HR High-resolution.

LR Low-resolution.

SR Super-resolution.

SISR Single-image super-resolution.

VSR Video super-resolution.

NN Neural network.

CNN Convolutional neural network.

MLP Multi-layer perceptron.

RNN Recurrent neural network.

LSTM Long short-term memory.

PSNR Peak signal-to-noise ratio.

SSIM Structural similarity index.

Chapter 1

Introduction

Super-resolution (SR) is the reconstruction of a high-resolution (HR) image from a corresponding low-resolution (LR) image. As with many inverse problems in graphics, SR is ill-posed due to the multitude of HR images corresponding with each LR image. However, recent advances in convolutional neural network (CNN) architectures have allowed CNN-based SR models to build effective priors about the world, greatly improving the quality of reconstruction. As deeper and increasingly complex architectures have been presented, execution speed has often been sacrificed for improved reconstruction quality, making the models impractical for real-world deployment.

This aim of this project is to explore prominent advances in CNN-based SR, evaluating their effect on the real-time performance of video super-resolution (VSR) models. A simple CNN architecture was first implemented, trained and evaluated on a VSR task. Seven techniques, such as frame-recurrence and residual learning, were then individually applied to the model, which was retrained and evaluated on the same task. From this, the techniques providing a performance benefit were identified and an improved model was designed. Finally, using pre-trained models with the improved design, a package for out-of-box VSR was constructed and released.

1.1 Motivation

On 19 March 2020, the video streaming service Netflix announced that they were lowering streaming quality, in order to handle increased traffic resulting from the COVID-19 crisis [1]. While these were exceptional circumstances, the rapidly increasing demand for video streaming, in addition to the advent of commercially available ultra-high-definition displays, makes this symptomatic of the challenges faced by streaming services. The data centres for such services face massive storage demands, often leading to lossy compression techniques being applied prior to storage. In addition to this, the latency of the service is a crucial factor in defining user experience, further compelling the use of compression techniques in order to transfer the data at a sufficient rate.

The decompression stage, which is performed locally to the user, must therefore also execute in real-time. In order to deploy CNN-based SR models to increase video resolution in this stage, their real-time performance, that being the trade-off between reconstruction quality (SR performance) and execution speed (computational performance), must be maximised.

Streaming on mobile devices, which comprises approximately a fifth of all streaming demand [2], exacerbates this problem. Firstly, the executing hardware has limited performance, making the real-time execution of computationally intensive CNN workloads challenging. In addition to this, the battery capacity of these devices necessitates that the amount of computation performed by the model be minimised, requiring models to be lightweight.

Approaching this as a hardware problem, steps are being taken by many companies to develop low-power hardware for neural network (NN) workloads. Neural processing units, such as Arm's Ethos-N series, aim to exploit the potential for parallelisation further than GPUs, allowing for efficient inference to be performed. In this project, I will explore the software approach to this problem by designing a model that maximises SR performance, whilst remaining lightweight and maintaining sufficient speed on the executing hardware.

1.2 Related work

Whilst the majority of literature on CNN-based SR focuses on improving reconstruction quality outright, multiple lightweight models have recently been proposed for fast VSR [3, 4, 5], with one claiming real-time execution speed on 1080p videos with a single GPU [6]. These papers generally present and evaluate the performance of complete models, which commonly consist of a variety of novel techniques. Therefore, comparing the performance of these models does not indicate the impact of any single technique, as would be found in an ablation study, in which a model's performance is reevaluated after a single component is removed. Other papers have claimed experiments in which the performance impact of a proposed technique is tested, however the results of these are rarely presented, with the evaluation instead favouring comparisons of the complete model to other state-of-the-art models (see the PReLU activation function, §2.4.3.1). In this project, I aim to evaluate the impact of individual techniques on the real-time performance of CNN-based SR models, rather than benchmarking against or reproducing the results of an existing model.

1.3 Report outline

Chapter 2 introduces the requisite technical background for CNN-based SR, followed by the CNN techniques selected for evaluation and finally the key aspects of project planning. **Chapter 3** describes the implementation of pipelines for SR dataset loading, model training and evaluation. **Chapter 4** presents the evaluation of the candidate techniques and reviews how the project's success criteria were fulfilled. **Chapter 5** summarises the project's results, followed by recommendations for future work in SR, along with ethical concerns.

Chapter 2

Preparation

In this chapter, the preparatory work completed prior to starting implementation is presented. Firstly, the project's starting point is described (§2.1), stating prior work and experience in relevant fields, along with a formal problem definition (§2.2). A background in SR models is then presented (§2.3), followed by a literature review of CNN-based SR (§2.4). Finally, a requirements analysis is completed (§2.5), from which a list of deliverables are constructed, and the software development tools required for implementation are given (§2.6).

2.1 Starting point

This project was researched and implemented from scratch, entirely within the official timeline. The Part IB Artificial Intelligence course provided a theoretical background, with the mathematical formulations of supervised learning and backpropagation (which are therefore not restated in this report). In addition to this, the Part II Computer Vision course provided relevant theory, in particular Bayesian inference and the formulation of convolution. Finally, the Part II Data Science unit of assessment undertaken alongside this project provided invaluable experience with Python and machine-learning libraries, in addition to guidance regarding the structure and evaluation of machine-learning projects.

2.2 Problem definition

Before exploring approaches to VSR, the problem is formally defined. VSR can be constructed from a sub-problem which we define first: single-image SR (SISR).

2.2.1 Single-image SR

Given a HR image $I^{\mathrm{HR}} \in [0,1]^{sH \times sW \times C}$, we map it to its corresponding LR image $I^{\mathrm{LR}} \in [0,1]^{H \times W \times C}$ by a degradation function

$$I^{\mathrm{LR}} = \mathcal{D}\big(I^{\mathrm{HR}}; \delta\big) \tag{2.1}$$

for degradation parameters δ, scale factor s and LR image dimensions $[H,W]$ with C colour channels. The operations performed by $\mathcal{D}$ are unknown, so we aim to find a SISR model $\mathcal{F}$ which returns an estimation I^{SR} of I^{HR}, from I^{LR} alone, as follows:

$$I^{\mathrm{SR}} = \mathcal{F}\big(I^{\mathrm{LR}}; \theta\big) \tag{2.2}$$

where θ is the parameters for $\mathcal{F}$. The goal of SISR is to find $\mathcal{F}$ and θ such that I^{SR} optimally approximates I^{HR} under some metric. Accordingly, for a given model $\mathcal{F}$ with trainable parameters θ, we aim to find

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}\big(I^{\text{HR}}, I^{\text{SR}}\big) = \arg\min_{\theta} \mathcal{L}\big(I^{\text{HR}}, \mathcal{F}(I^{\text{LR}}; \theta)\big) \tag{2.3}$$

where $\mathcal{L}$ is a loss function between I^{HR} and I^{SR}. In this report, *L2 loss* (or mean squared error) is used for $\mathcal{L}$, as is standard in SR models.

2.2.2 Video SR

VSR, the core problem of this project, is formulated by a simple extension to SISR. We consider a sequence of N HR frames $I_N^{\text{HR}} \in [0,1]^{N \times sH \times sW \times C}$ which are individually mapped to an LR sequence $I_N^{\text{LR}} \in [0,1]^{N \times H \times W \times C}$ by $\mathcal{D}$, as in (2.1). Analogously to (2.2), the goal of VSR is to find $\mathcal{F}^N$ and θ^N such that

$$I_N^{\text{SR}} = \mathcal{F}^N\big(I_N^{\text{LR}}; \theta^N\big) \tag{2.4}$$

optimally approximates I_N^{HR} under some metric. Therefore, SISR can be viewed as a subproblem of VSR, for $N = 1$. Furthermore, we can construct a subset of VSR models as

$$\mathcal{F}^N = \mathcal{F}_1 \times \ldots \times \mathcal{F}_N \tag{2.5}$$

where each $\mathcal{F}_i \in [0,1]^{H \times W \times C} \to [0,1]^{sH \times sW \times C}$ is a SISR model. By this, SISR models can be used to construct VSR models, in which each SR frame is generated from a single LR frame.

However, VSR models can leverage information from all LR frames when constructing each SR frame. This is not possible with the Cartesian product of SISR models, as in (2.5), which consider each LR frame in isolation. Therefore, we simplify the problem for the majority of this report by considering the development of SR techniques applied to SISR, until video-based techniques are specifically discussed, such as in §2.4.4.

2.3 Approaches to SR

This section explores two approaches to SR: interpolation (§2.3.1), from which a baseline method is defined (bicubic interpolation), and CNN-based SR (§2.3.2). Background theory and a definition of the CNN are first presented, before the seminal work in CNN-based SR is described. §2.4 builds upon this, by exploring advances in the field following this work.

2.3.1 Interpolation

Interpolation is the estimation of intermediate values of a function, from discrete samples of that function. The simplest multivariate form of this is *bilinear interpolation*, in which a linear polynomial (known as an interpolant) is constructed based on the four nearest neighbours of the target coordinate in the sample grid. For conciseness, the definition of linear interpolation

(which fits a linear interpolant of a single variable), is given:

$$f(x) = f(x_0) + (x - x_0)\frac{f(x_1) - f(x_0)}{x_1 - x_0} \tag{2.6}$$

where $f \in \mathbb{R} \to \mathbb{R}$ is the function being estimated at $x \in \mathbb{R}$ and $f(x_0), f(x_1)$ are the nearest neighbour samples, with $x_0 < x_1$.

Since the interpolant is linear, the approximation is not smooth and often leaves interpolation artefacts. This is improved by *polynomial interpolation*, a generalisation of linear interpolation for interpolants of any degree, with *bicubic interpolation* being a commonly used two-dimensional variant of this. This produces a third-degree polynomial based on the sixteen nearest neighbours of the target coordinate, which allow the algorithm to approximate the derivative at each of the four nearest neighbours. Due to this, the algorithm achieves superior performance to bilinear interpolation on SR tasks [7].

However, bicubic interpolation has been consistently outperformed by CNN-based models [8, 3, 6, 9, 4, 10, 5], so is typically used as a baseline for evaluation. It serves the same purpose in this project, to provide a lower bound for SR performance when implementing CNN-based models.

2.3.2 CNN-based SR

Prior to defining the CNN, NNs are introduced by reviewing the simplest form of multilayer NN, the multi-layer perceptron (MLP), and the role of non-linear activation functions. Since the SR models developed in this project execute on image data, the two-dimensional versions of the MLP and CNN are defined.

2.3.2.1 MLP

The MLP consists of a sequence of fully-connected layers of neurons, with the output (activation) of each neuron feeding into the input of every neuron in the following layer. In the case of SISR, I^{LR} acts as the input layer and I^{SR} the output layer.

Each neuron in the intermediate (hidden) or output layers computes a linear combination of the activation map of its previous layer plus a constant (bias) term, then applies a non-linear activation function to determine its activation,

$$a_{ij}^{(l)} = g\left(b + \sum_m \sum_n w_{mn} \cdot a_{mn}^{(l-1)}\right) \tag{2.7}$$

where w and b are respectively the weights and bias of the neuron at (i, j), $a^{(l)}$ is the activation map of layer l and g is the activation function. For an input vector x, we therefore define the input layer as $a^{(0)} = x$. Figure 2.1 demonstrates an example of a one-dimensional MLP architecture.

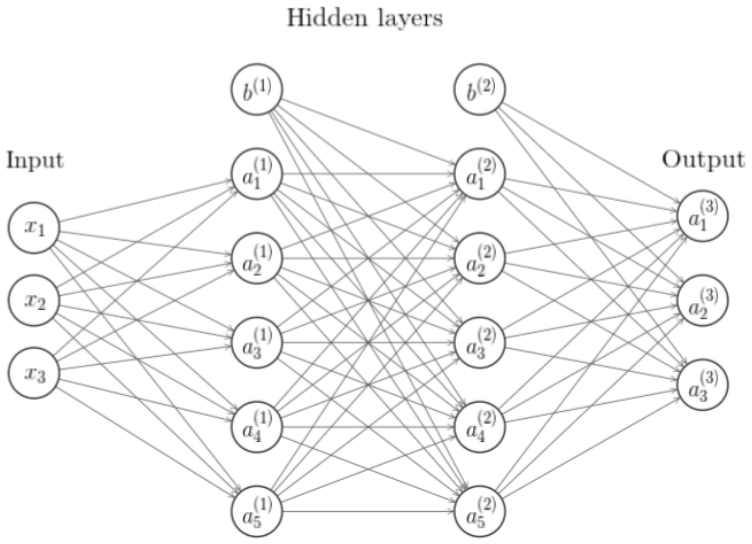


Figure 2.1: Example MLP architecture — approximates $f \in \mathbb{R}^3 \to \mathbb{R}^3$ with two hidden layers.

2.3.2.2 Non-linear activation functions

By the Universal Approximation Theorem [11, 12], MLPs with a finite number of neurons are able to approximate any function up to an arbitrary level of precision, if and only if the network's activation function is non-polynomial. For this reason, networks with linear activation functions $x \mapsto c \cdot x$ (or no activation function, for $c = 1$) are unable to learn all (more specifically, non-linear) functions.

For this reason, we require non-linear (and ideally, non-polynomial) activation functions in order for a network to learn complex patterns. Originally, NNs commonly used the *hyperbolic tangent function*

$$g(x) = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.8}$$

or the *logistic sigmoid function*

$$g(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.9}$$

where x is the input signal of the activation function and e is Euler's number.

Both of these are differentiable and monotonic, however as $|x|$ increases their gradients approach 0. This leads to the *vanishing gradient problem*, wherein the partial derivatives of the loss function with respect to neurons' weights becomes vanishingly small as they backpropagate through the network, leading to the front layers of the network training very slowly. Due to this, the *rectified linear unit* (ReLU) was proposed [13],

$$g(x) = \max(0, x). \tag{2.10}$$

ReLU is a piecewise linear function, mapping the negative part of its domain to zero and retaining the remainder. In addition to being cheaper to compute, it has a constant gradient, decreasing the likelihood of a vanishing gradient and therefore leading to faster learning. Consequently, most NNs, including SR models, use ReLU activations in their hidden layers. Variants

and uses of ReLU in SR models are discussed further in §2.4.3.

2.3.2.3 CNN

CNNs are a specialised form of NN, useful when handling data in a grid topology. While the MLP consists exclusively of fully-connected layers, which use general matrix multiplication, CNNs also contain at least one convolutional layer, which performs convolution between the previous layer's activations and a kernel. In practice, most machine learning libraries implement cross-correlation [14, p. 329], which is an equivalent operation to convolution except the kernel is not flipped. Therefore, we define two-dimensional convolutional layers using cross-correlation, the activation map for which is defined as

$$\begin{aligned} a_{ij}^{(l)} &= g(b + w * a^{(l-1)}) \\ &= g\left(b + \sum_m \sum_n a_{i+m,j+n}^{(l-1)} \cdot w_{mn}\right). \end{aligned} \tag{2.11}$$

By sharing parameters, CNNs are able to learn shift-invariant patterns with improved memory requirements and computational efficiency compared to MLPs [14, p. 333]. This makes them suitable for image data, where invariance to translation is necessary and layers can be very large. Figure 2.2 demonstrates an example of two-dimensional convolution, with a multi-channel kernel and input.

Dong et al. [8] first proposed the use of shallow CNNs for SISR with a model named SRCNN, exceeding state-of-the-art performance. The model consists of three convolution layers with ReLU activations, applied to an interpolated input image. Since then, a variety of advances have been made in CNN-based SR, which are detailed in §2.4.

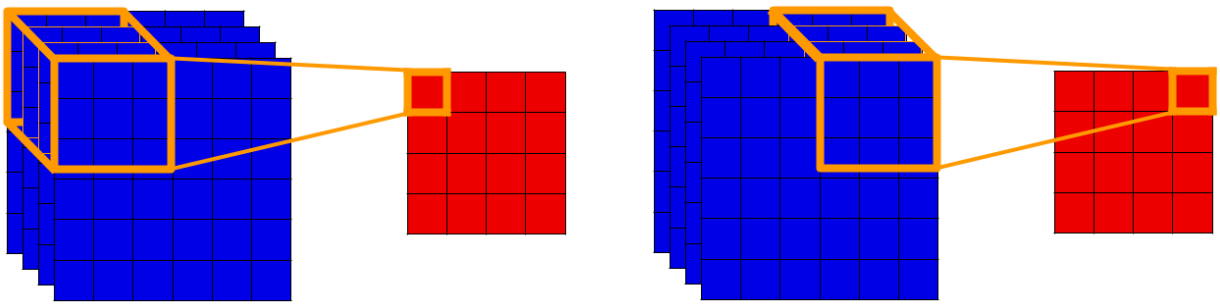


Figure 2.2: Example of two-dimensional, multi-channel convolution.

2.4 Advances in CNN-based SR

In this section, a survey of recent literature on CNN-based SR is presented, with the key contributions from various proposed models being defined and their theoretical benefits contrasted. These form the basis for the implementation detailed in Chapter 3 and their performance is contrasted in Chapter 4.

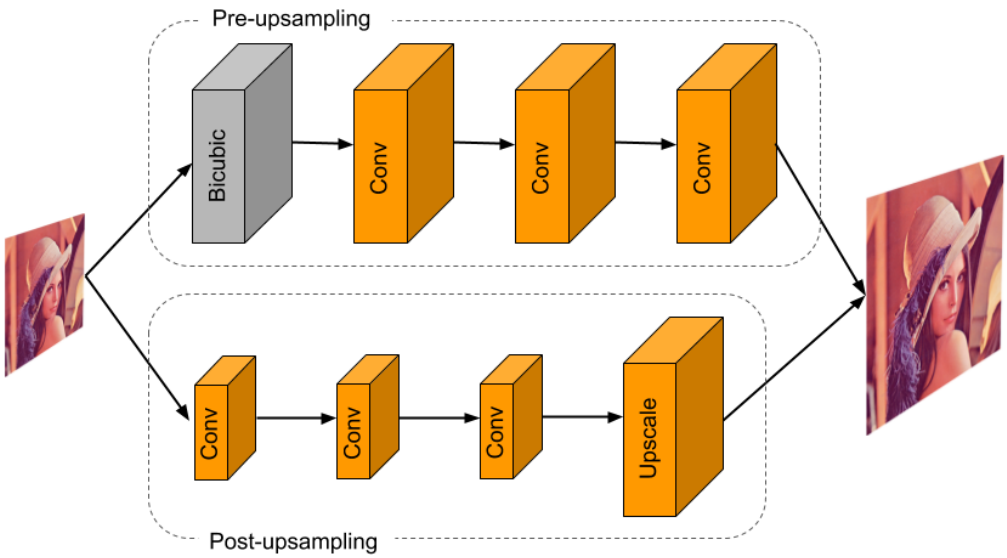


Figure 2.3: Pre- vs. post-upsampling — **orange** operations contain learnable parameters, while **grey** operations are predefined.

2.4.1 Post-upsampling

In the seminal work on CNNs for SR [8], the input image is upsampled by bicubic interpolation in a pre-processing step, prior to network inference. This approach, known as *pre-upsampling*, makes the network input s^2 larger than I^{LR}. By instead feeding I^{LR} directly to the network, the convolution operations occur in LR space over the same information, allowing for an $\mathcal{O}(s^2)$ improvement in time and memory complexity (assuming constant stride).

In order to achieve this, the upsampling operation is applied after feature extraction has been completed (Figure 2.3). This approach is known as *post-upsampling*, for which two upsampling operations have become dominant: *deconvolution* and *sub-pixel convolution*.

2.4.1.1 Deconvolution

Presenting a model named FSRCNN, Dong et al. [3] proposed upsampling after feature extraction with a deconvolution (or transpose convolution) layer [15]. Deconvolution is equivalent to the backwards pass of the convolution operation, thereby making the output dimensionality greater than the input.

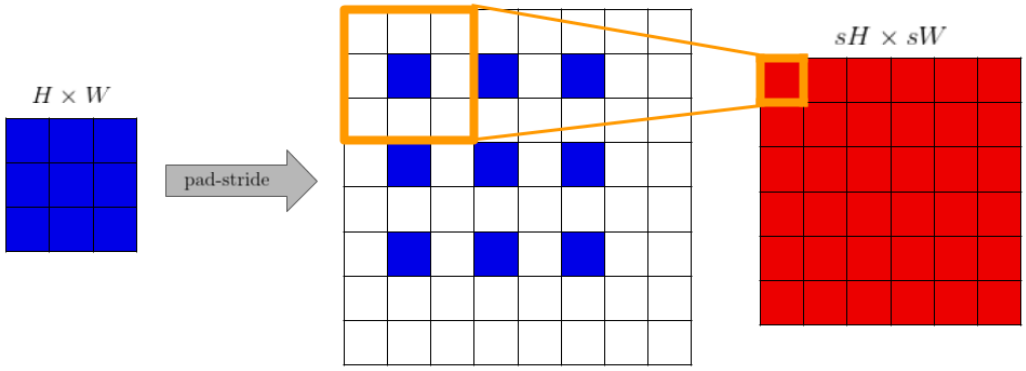
With convolution kernels, a stride of k produces an output map with a resolution of $1/k$ times the input resolution. Therefore, an equal stride with a deconvolution kernel produces an output with resolution k times greater than the input resolution. By setting $k = s$, where s is the desired upscaling factor, I^{SR} is produced directly from LR space (Figure 2.4a).

Furthermore, bicubic interpolation (see §2.3.1) can be viewed as a variant of the deconvolution operation for a predefined kernel [16]. By making the weights of the upsampling kernel learnable, deconvolution allows the upsampling operation to be fit to the training dataset, improving performance.

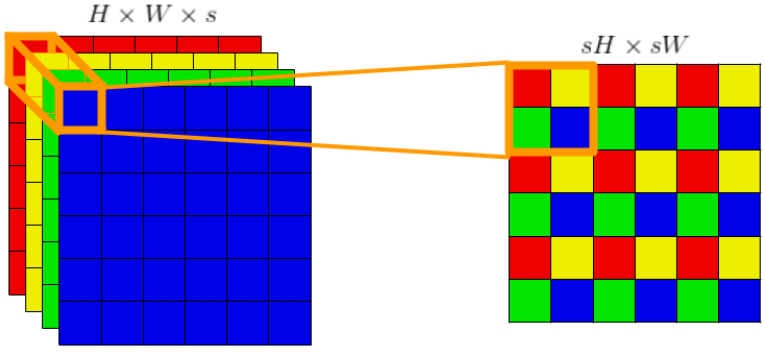
2.4.1.2 Sub-pixel convolution

Shi et al. [6] proposed an alternative upsampling operation in their ESPCN model, sub-pixel convolution. In this, the final convolution layer produces an output of shape $H \times W \times s^2C$, which is then reshaped by a depth-to-space transformation (Figure 2.4b). This rearranges the data so values are moved in spatial blocks from the channel dimension to the height and width dimensions [17], producing an output, I^{SR}, of shape $sH \times sW \times C$ for a block size s. Therefore, sub-pixel convolution contains no learnable parameters and upsamples from LR to HR space with no change to the data.

In a further work, Shi et al. [18] demonstrated mathematical equivalence between the deconvolution and sub-pixel convolution operations, thereby concluding that there is no need for explicit upsampling, as performed by deconvolution. Despite this, multiple SR models published following this have opted for the deconvolution operation to move between LR and HR space [19], thereby implying there is some performance difference despite their theoretical equivalence. In order to test this hypothesis, the two methods are empirically evaluated against each other in §4.4.2.



(a) Deconvolution



(b) Sub-pixel convolution

Figure 2.4: Post-upsampling methods for 2× upsampling ($s = 2$).

2.4.2 Residual learning

NNs become increasingly difficult to train as their depth grows due to the vanishing gradient problem (see §2.3.2.2). In theory, deep NNs should be able to at least equal the performance of shallow NNs, by using an identical mapping in their later layers. However, empirical evidence

of a 56-layer model being outperformed by a 20-layer model with the same architecture was presented by He et al. [20], suggesting that deep networks are incapable of learning all the mappings they may represent through gradient descent.

As a solution to this, the same work proposed *residual networks*, which achieve significantly improved results with deeper architectures. These work by adding 'skip connections', which forward the output of a non-adjacent previous layer to the current layer (Figure 2.5), where some operation is performed between it and the previous layer's output. Commonly, this is the element-wise sum of the layers, for which the output of a residual block at layer l is defined as

$$a_{ij}^{(l)} = g\left(a_{ij}^{(l-1)} + \sum_{\delta \in \Delta} a_{ij}^{(l-\delta)}\right) \tag{2.12}$$

where Δ denotes the lengths of skip connections into layer l. This is equivalent to stacking the residual inputs and performing a 1×1 convolution over them.

2.4.2.1 Local residual blocks

The SR models presented earlier, SRCNN [8], FSRCNN [3] and ESPCN [6], used shallow network architectures, consisting of $2 - 8$ hidden convolutional layers. Following these, Lim et al. [9] presented EDSR, a residual network with 69 hidden convolutional layers which exceeded state-of-the-art performance. The massive increase in parameters, from 12K in FSRCNN to 8M in EDSR, makes its performance improvement unsurprising, however it demonstrates the potential of deep residual networks for SR. This used *local residual blocks*, in which blocks of residual layers are connected in a non-residual architecture (Figure 2.5b). §2.4.2.2 presents an extension to this, *global cascading*.

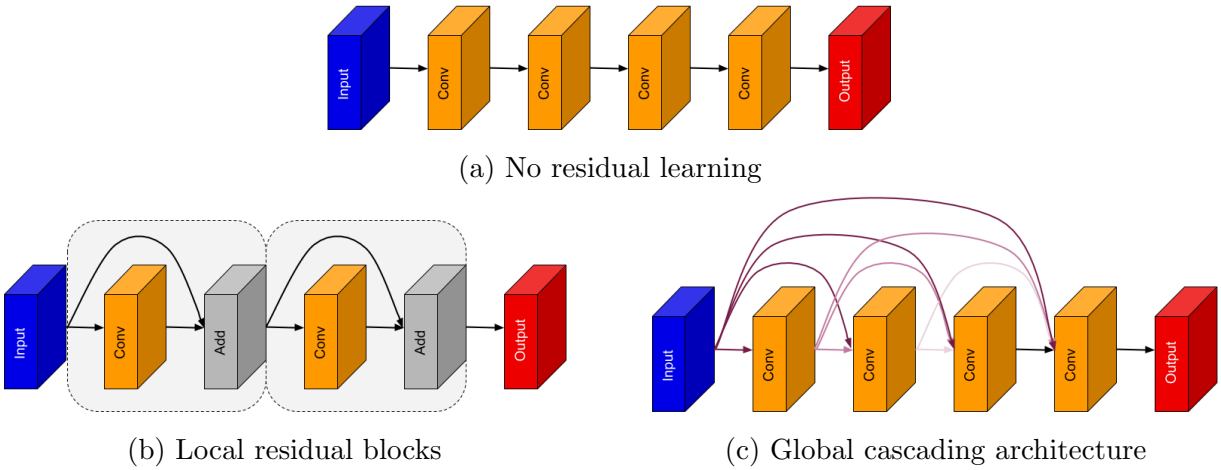
2.4.2.2 Global cascading

Following EDSR, Ahn et al. [4] proposed CARN, a *cascading residual network*. In addition to the local residual blocks described earlier, CARN uses a global cascading architecture, in which skip connections from all previous residual blocks feed into later residual layers (Figure 2.5c). This architecture is equivalent to recursively applying the residual block structure to a chain of residual blocks. The performance of CARN was slightly below EDSR, however the network was optimised for speed and memory efficiency so achieved this with six times fewer parameters, hence demonstrating the effectiveness of global cascading in SR models. The real-time performance impact of the two residual learning variants against an equivalent non-residual network is evaluated in §4.4.1.

2.4.3 Activation functions

2.4.3.1 ReLU variants

A major disadvantage of ReLU is that it has zero gradient when the unit is not active ($x < 0$). Therefore, units which become inactive will never be activated again as backpropagation does not adjust their weight, creating 'dead' features. This leads to slower convergence, in addition



(a) No residual learning

(b) Local residual blocks

(c) Global cascading architecture

Figure 2.5: Residual learning methods — inspired by Wang et al. [19]

to inefficient use of the network's parameters. The *Leaky ReLU* function [21] provides a solution to this, being defined as follows:

$$g(x) = \max(x, 0) + \alpha \min(x, 0) \tag{2.13}$$

for some constant $\alpha \in [0, 1]$ (often 0.01). This assigns a non-zero slope to the negative part of the domain, preventing neurons from dying.

The concept of Leaky ReLU was taken further by He et al. [22] who proposed the *Parametric ReLU* (PReLU) function, which makes α a trainable parameter rather than using a constant. PReLU was accredited by the authors as the reason for their model's state-of-the-art performance on the ImageNet classification task [23], demonstrating its effectiveness.

Dong et al. [3] proposed PReLU activations in their FSRCNN model, claiming experiments which verified improved performance of PReLU-activated networks compared to their ReLU-activated counterparts. Despite this, to the best of my knowledge no published SR models have since used PReLU activations, generally selecting ReLU instead. For this reason, empirically verifying this hypothesis would generate a useful result, as completed in §4.4.3.

2.4.3.2 Activation removal

An alternative solution to ReLU's dead feature problem is proposed by Li et al. [5]: removing many of the activation operations from the network. They argue that useful information is partially discarded by ReLU, so by decreasing the number of activation operations the network retains more information, allowing a smaller, more efficient network to be designed. This is verified in their paper by measuring model performance over a range of activation removal ratios, concluding that removing 1/2 of the activation operations from a typical CNN architecture (with every convolution layer ReLU-activated) optimises SR performance. Since this paper was published recently (Sept. 2019), this hypothesis has not been fully explored in the literature. Therefore, empirically verifying it would again provide a useful result (see §4.4.3).

2.4.4 Frame-recurrence

The techniques discussed in §2.4.1–2.4.3 are applicable to both SISR and VSR models. However, since video frames are temporally correlated, VSR provides the opportunity to leverage the mutual information between LR frames when generating each SR frame. Performing this effectively would allow models to achieve higher SR performance, or achieve equal performance with a shallower architecture and, if performed efficiently, higher computational performance.

The original approach to this was *multi-frame SR*, in which a sliding window is passed over I_N^{LR}, thereby using adjacent LR frames as input for each SR frame. This was taken further by methods for warping the adjacent LR frames onto the central LR frame, for example by optical flow [24] or a trainable motion compensation network [25]. However, due to LR frames being reprocessed in multiple SR predictions, in addition to the cost of processing multiple frames per prediction, multi-frame SR is inherently inefficient and therefore limits computational performance. As a solution to this, recurrent neural networks (RNNs) have been proposed for VSR.

2.4.4.1 RNN

In contrast to the feedforward architectures described earlier, RNNs are feedback, meaning the output from each recurrent unit is saved and fed back as input on the next forward propagation. For the t-th forward propagation in a sequence, the activation map of a recurrent unit at layer l is defined by

$$a^{(l),\langle t\rangle} = \phi\big(a^{(l-1),\langle t\rangle}, a^{(l),\langle t-1\rangle}, \theta^{(l)}\big) \tag{2.14}$$

where ϕ is the operation performed by the recurrent unit and $\theta^{(l)}$ is the trainable parameter set for layer l.

Many operations have been proposed for ϕ in VSR models, for instance the concatenation of $a^{(l-1),\langle t\rangle}$ and $a^{(l),\langle t-1\rangle}$ in the channel dimension (FRVSR [10]) and the Long Short-Term Memory (LSTM) unit (DRRNet [26], see §2.4.4.2). The choice of ϕ for this project was influenced by implementation-specific details, which are therefore discussed further in §3.2.1.1.

2.4.4.2 LSTM

In a basic RNN architecture [27], ϕ consists of a single hyperbolic tangent operation applied to the concatenation of $a^{(l-1),\langle t\rangle}$ and $a^{(l),\langle t-1\rangle}$. However, despite this being theoretically capable of handling long-term dependencies — that is, maintaining useful state from much earlier in the input sequence — these have extensively been shown to fail at this task [28]. LSTM units were proposed by Hochreiter and Schmidhuber [29], as a solution to this problem.

An LSTM unit is composed of a cell, which is stored and used in the next forward propagation, in addition to three gates: input, output and forget. Essentially, the roles of these are as follows:

- **Cell,** $c^{\langle t\rangle}$ — Stores information ('dependencies') from previous sequence elements.
- **Input gate,** $i^{\langle t\rangle}$ — Controls the extent to which $c^{\langle t\rangle}$ is updated by the input, $a^{(l-1),\langle t\rangle}$.

- **Forget gate,** $f^{\langle t \rangle}$ — Controls the extent to which $c^{\langle t \rangle}$ is maintained from $c^{\langle t-1 \rangle}$.
- **Output gate,** $o^{\langle t \rangle}$ — Controls the extent to which $c^{\langle t \rangle}$ determines the output, $a^{(l),\langle t \rangle}$.

Formally, these are defined by

$$\begin{aligned} i^{\langle t \rangle} &= \sigma\big(w_i \cdot [a^{(l),\langle t-1 \rangle}, a^{(l-1),\langle t \rangle}] + b_i\big) \\ f^{\langle t \rangle} &= \sigma\big(w_f \cdot [a^{(l),\langle t-1 \rangle}, a^{(l-1),\langle t \rangle}] + b_f\big) \\ o^{\langle t \rangle} &= \sigma\big(w_o \cdot [a^{(l),\langle t-1 \rangle}, a^{(l-1),\langle t \rangle}] + b_o\big) \\ \tilde{c}^{\langle t \rangle} &= \tanh\big(w_c \cdot [a^{(l),\langle t-1 \rangle}, a^{(l-1),\langle t \rangle}] + b_c\big) \\ c^{\langle t \rangle} &= \sigma\big(f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + i^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}\big) \\ a^{(l),\langle t \rangle} &= o^{\langle t \rangle} \circ \tanh\big(c^{\langle t \rangle}\big) \end{aligned} \tag{2.15}$$

where $\{w_i, w_f, w_o, w_c\}$ and $\{b_i, b_f, b_o, b_c\}$ are respectively the weights and biases of the LSTM unit, $[x, y]$ denotes the concatenation of vectors x and y and $\circ$ denotes the element-wise (Hadamard) product. By systematically updating c over the sequence, the cell state can be effectively added to and removed from, allowing dependencies to be carried through long sequences.

2.5 Requirements analysis

This section presents the high-level requirements of the project, based on the success criteria given in the project proposal (Appendix A). Upon completion of the literature review detailed in §2.4, refinements to the success criteria were made.

2.5.1 Success criteria refinements

The original success criteria involved the implementation and training of existing SR models (e.g. FSRCNN, ESPCN), aiming to reproduce their reported performance. Since each model implements a variety of techniques, comparing their overall performance would not allow the relative contribution of each technique to the model's performance to be understood, as in an ablation study. Therefore, this criterion was amended to be the implementation of a core CNN-based model, which would be individually extended and reevaluated with the techniques given in §2.4.

In addition to this, another criterion stated that an application shall be implemented to "apply the SR model to an input video". The application was left purposefully vague, as its role was to demonstrate a possible use case of VSR. However, creating an application purely for demonstration purposes would not provide general insight into VSR, so the criterion was modified to be the implementation of a VSR inference pipeline, as would be found in such an application.

Finally, as an extension criterion, a general-purpose VSR package could be implemented from the inference pipeline, providing an API for out-of-box upscaling of videos. This would have real-world applications outside of this project, as it would allow developers with no machine

learning experience to integrate VSR into their applications, as well as providing an evaluation framework for SR researchers.

2.5.2 Project deliverables

From these refined success criteria, a set of deliverables (Table 2.1) were defined and categorised by type, priority and difficulty, as follows:

- **Type** (Functional or non-functional) — Functional deliverables are specific functions of the system being implemented, whereas non-functional deliverables are measurements of the operation of the system.
- **Priority** (Core, desired or optional) — The deliverables required to fulfil the core success criteria are labeled 'core'. Those which fulfil extension criteria are labelled 'desired' or 'optional', based on the extent to which they contribute to the core aim of the project.
- **Difficulty** (Low, medium or high) — The estimated difficultly in achieving or implementing a deliverable.

The core non-functional deliverable is that a CNN-based model will be implemented and trained, achieving superior SR performance to the baseline method, bicubic interpolation. As an extension, the model should achieve this with real-time computational performance (see §4.1) on the VSR task.

Table 2.1: Project deliverables.

Type	Deliverable	Priority	Difficulty
Functional	Data gathering	Core	Low
	Data pre-processing	Core	Medium
	Baseline SR model	Core	Low
	Core SR model	Core	Medium
	Advanced CNN techniques	Core	High
	Frame-recurrent model	Desired	High
	Training pipeline	Core	Medium
	Technique evaluation	Core	High
	Inference pipeline	Core	Medium
	VSR package	Optional	Medium
Non-functional	Outperform baseline	Core	Medium
	Outperform in real-time	Optional	High

2.6 Software engineering tools

2.6.1 Programming language and libraries

The project was developed in **Python 3.6**, using a virtual environment created with **Virtualenv** in order to isolate the project's Python environment from the system implementation. Python was selected due to it being supported by many machine learning libraries, namely **TensorFlow's** implementation of the **Keras API**, which was used for NN implementation and training. In addition to this, standard Python libraries for scientific computing and visualisation were used, such as **Numpy** and **Matplotlib**. Finally, **Jupyter** notebooks were used for fast data exploration, visualisation and model execution (see §2.6.2).

2.6.2 Computational resources

The project implementation was completed on my personal machine (2 GHz Intel Core i5 processor with 8 GB RAM), however due to NN execution being computationally intensive, model training and evaluation required additional resources. Since NNs are highly parallelisable, GPU execution offers large speedups and is standard when benchmarking computational performance. Therefore, **Google Colaboratory**, an online Jupyter notebook environment offering free execution time on Tesla K80 GPUs, was used for model training and evaluation.[1]

2.6.3 Version control and backup

The project source code was developed on my personal machine, in addition to being pushed to a Git repository hosted on **GitHub** upon the completion of tasks. This allowed the repository to be transformed between working states, making roll-back to previous versions simple and allowing different features to be concurrently developed by branching.

Overleaf, an online LaTeX editor, was used for report writing. In addition to this automatically saving the report, it was also frequently pushed to a Git repository and downloaded to my personal machine nightly. Finally, both the project report and source code were regularly backed up from my personal machine with the cloud storage service **iCloud**.

2.6.4 Software licence

The source code for both model training and the inference package were made publicly available on GitHub. Since the dependencies were not released in source code or binary format with the project, it was not necessary to comply with their licences. Therefore, the **MIT licence** was selected due to it being permissive and having high licence compatibility, allowing developers to easily use the project in their own applications.

[1] The original project proposal refers to a HiKey 960 development board, which was loaned from Arm for the execution of SR models. It was decided that, in addition to the overhead in setting up and implementing an inference pipeline on the board, this would contribute little to the primary goal of the project, so the board was not used.

Chapter 3

Implementation

In this chapter, the implementation of the deliverables outlined in the previous chapter is presented. This is completed in three sections, which are ordered by relative dependency: data handling (§3.1), model training (§3.2) and the inference package (§3.3). Finally, an overview of the project repository is given (§3.4).

3.1 Data handling

This section details how SR datasets were gathered (§3.1.1), pre-processed (§3.1.2) and efficiently loaded for use in model training and evaluation (§3.1.3).

3.1.1 Gathering datasets

In order to train a supervised learning algorithm, a set

$$(x_1, y_1), \ldots, (x_n, y_n) \tag{3.1}$$

of training samples is required, where y_i is the value of the target variable for a given input x_i. SR datasets consist of $(I^{\mathrm{LR}}, I^{\mathrm{HR}})$ pairs (or $(I_N^{\mathrm{LR}}, I_N^{\mathrm{HR}})$ for VSR datasets), which are generated by a degradation function $\mathcal{D}$ applied to each HR image or video frame, as in (2.1). Often, $\mathcal{D}$ performs a simple, uniform downsampling such as bicubic interpolation (see §2.3.1), however specialised datasets exist for specific, real-world degradations (e.g. JPEG compression [30]). Since specific degradations are not the focus of this project, generic SR datasets using bicubic downsampling were selected (Table 3.1).

Naturally, the recurrent (video-based) models required video data for training, so a VSR training dataset (REDS [31]) was selected. However, in order to maximise the performance of single-image models, the number of unique images in the training dataset should also be maximised. Due to the correlation between frames in a video, training such a model on video data would provide little additional benefit compared to training on a single frame from each video. Therefore, a SISR dataset (DIV2K [32, 33]) with a large number of images was selected for the training of feed-forward (single-image) models.

The selected datasets are common benchmark datasets for their respective tasks, with both training datasets being sufficiently large in total number of frames and definition (DIV2K and REDS containing 1080p and 720p frames respectively) for both model training and validation. In addition to this, the models were designed to perform SR for a specific upsampling factor s, requiring that all datasets used the same s. The selected datasets all contained subsets with $s = 4$, so the models were trained for $\times 4$ upsampling.

Table 3.1: Datasets used for model training and testing.

Task	Subset	Dataset	Total number of frames	Number of videos
SISR	Train	DIV2K [32, 33]	900	-
VSR	Train	REDS [31]	24000	240
	Test	Vid4 [34]	167	4

3.1.2 Pre-processing

In order to efficiently and accurately train a supervised learning model, it is crucial that the data being trained on is pre-processed. This stage of the training pipeline commonly includes data cleaning, normalisation, augmentation and feature selection, although the applicability of each of these varies by dataset. In this section, two key pre-processing stages applied to the selected SR datasets are presented: data augmentation (§3.1.2.1) and mini-batching (§3.1.2.2).

3.1.2.1 Data augmentation

The core aim of any supervised learning model is to perform well on new and unseen data, in addition to that on which it was trained. This ability is known as *generalisation* and it requires the model's variance — that is, the extent to which it overfits to the training data — to be minimised. A determining factor in this is the ratio of trainable parameters to the number of training samples [14, p. 110], making deep NNs particularly prone to overfitting.

One method of reducing model variance is to generate additional training samples through *data augmentation*. In this, existing training samples are augmented to produce new training samples, thereby improving the parameter-to-sample ratio. Therefore, the SR datasets were augmented by randomly flipping and rotating each image, increasing the number of samples by a factor of 8. In addition to this, the images were randomly cropped in order to perform mini-batching, as described in §3.1.2.2.

3.1.2.2 Mini-batching

In order to understand why data batching is used in model training, we first review NN optimisation. In order to train a NN with learnable parameters θ to minimise some objective function $J(\theta)$ over the training dataset, we require an optimisation algorithm. A common first-order algorithm for NN training is *gradient descent*, which iteratively updates θ as follows:

$$\theta := \theta - \eta \cdot \nabla J(\theta) \tag{3.2}$$

where η denotes the learning rate. While this is guaranteed to converge to a local minima for non-convex error surfaces [35], each update step requires the computation of $J(\theta)$ over the entire dataset, making it slow for large datasets. Since SR models are trained on image data, which is very large, gradient descent would therefore take too long to converge.

As a solution to this, *mini-batch gradient descent* [36] computes an approximation of the

gradient of the $J(\theta)$ using a subset (mini-batch) of the dataset, as follows:

$$\theta := \theta - \frac{\eta}{k} \cdot \sum_{i=1}^{k} \nabla J_i(\theta) \tag{3.3}$$

where k is the mini-batch size and $J_i(\theta)$ is the objective function for the i-th sample in the dataset. This means each update step requires less time to compute, lowering overall training time. Furthermore, if k can be tuned such that the entire mini-batch can fit in memory, while being sufficiently large to give an accurate approximation of $\nabla J(\theta)$, the mini-batch can be efficiently computed by executing operations in parallel over all samples.

In order to implement training with mini-batches, a tensor containing k images must be constructed. However, since the resolution of the dataset images was variable, concatenating the images did not produce a tensor with uniform shape (of the form $k \times H \times W \times C$), as is required for model training. Therefore, in order to create a mini-batch tensor with uniform shape, multiple methods were considered:

- **Padding** — Each of the images are zero-padded to the maximum resolution of any image in the mini-batch. A masking layer is then applied, such that the output of the padded data is ignored in model training and evaluation. However, this method can become very inefficient since redundant operations over the padding data are still executed.
- **Interpolation** — Bicubic interpolation is used to resize all of the images to the same resolution. However, increasing image resolution requires more convolution operations over the same information, while decreasing resolution loses information and can introduce interpolation artefacts [37].
- **Truncation** — Images are randomly cropped to the minimum resolution. While this leads to less information per sample, the information density is maintained, allowing operations to be performed efficiently over the batch. In addition to this, periodically recropping the images allows all image data to be incorporated into the training process, meaning no information or computation is wasted. Therefore, this method was selected for producing mini-batch tensors with uniform shape.

3.1.3 Data loading pipeline

In order for `Keras` models to be trained and evaluated, they require data in the form of TensorFlow `tf.Tensor` objects. A data loading pipeline was constructed to generate mini-batched `tf.Tensor` pairs from the selected datasets (see §3.1.1), which could be fed to the models. An overview of the pipeline's implementation is given in Figure 3.1.

3.1.3.1 Dataset abstraction

In contrast to nominally typed languages such as Java and C++, in which any object must be exactly or a subtype of a given type, Python uses duck typing [38]. This requires only that an object supports the methods and attributes called from it, meaning that rather than strict interfaces, Python uses informal interfaces named protocols.

In order to make the architecture extensible to new datasets, protocols for image and video datasets were designed. These specified the implementation of an LR-HR generator, which returned the file paths of corresponding image or video pairs from the given dataset. By implementing this protocol, the low-level details of each dataset were abstracted from the data loading pipeline, allowing it to be constructed independently of any dataset.

3.1.3.2 Efficient data loading

Given the file paths to LR-HR pairs, the data loading pipeline then had to load the data so it could be pre-processed and fed to a Keras model for training. A simple Python-based approach to this was to load the data into a NumPy array, then pass it to the model through a Python dictionary with the Keras `feed_dict` method. However, since this loads the data in Python, it is subject to Python's global interpreter lock (GIL), which restricts interpreter execution to a single thread. This greatly slows the data loading, creating a bottleneck and thereby preventing high GPU utilisation when training the models.

An alternative solution was to load the data through TensorFlow's `tf.data.Dataset` object.[1] Since the data loading is performed by TensorFlow's efficient C++ implementation, it is significantly faster than the Python-based method and avoids Python's GIL. In addition to this, the object contains methods for data batching and mapping pre-processing functions over the dataset elements. Therefore, this solution was implemented by the `build_dataset` method of the `DataLoader` subclasses, which return a loaded `tf.data.Dataset` object.

3.2 Model training

This section details how models were designed, implemented and trained in order to evaluate the techniques presented in §2.4. The design and implementation of the models is first described (§3.2.1), followed by the construction of the training pipeline (§3.2.2) and finally the tuning of the models (§3.2.3).

3.2.1 Model implementation

Prior to outlining the design and implementation of the core model, the justification of which recurrent unit was selected for the frame-recurrent extension is presented.

3.2.1.1 Recurrent unit selection

§2.4.4 introduces RNNs for VSR, however the choice of recurrent unit was partially based on implementation-specific details. Keras provides support for a limited selection of recurrent units: simple (fully-connected), Gated Recurrent Units [39] and LSTM [29] (see §2.4.4.2). Each of these involve a fully-connected operation over the input, which is costly and unsuitable given image data, as discussed in §2.3.2.3. However, Keras also provides support for a variant of

[1] Data loading with the `tf.data.Dataset` object was inspired in part by an online guide: `https://towardsdatascience.com/how-to-quickly-build-a-tensorflow-training-pipeline-15e9ae4d78a0`.

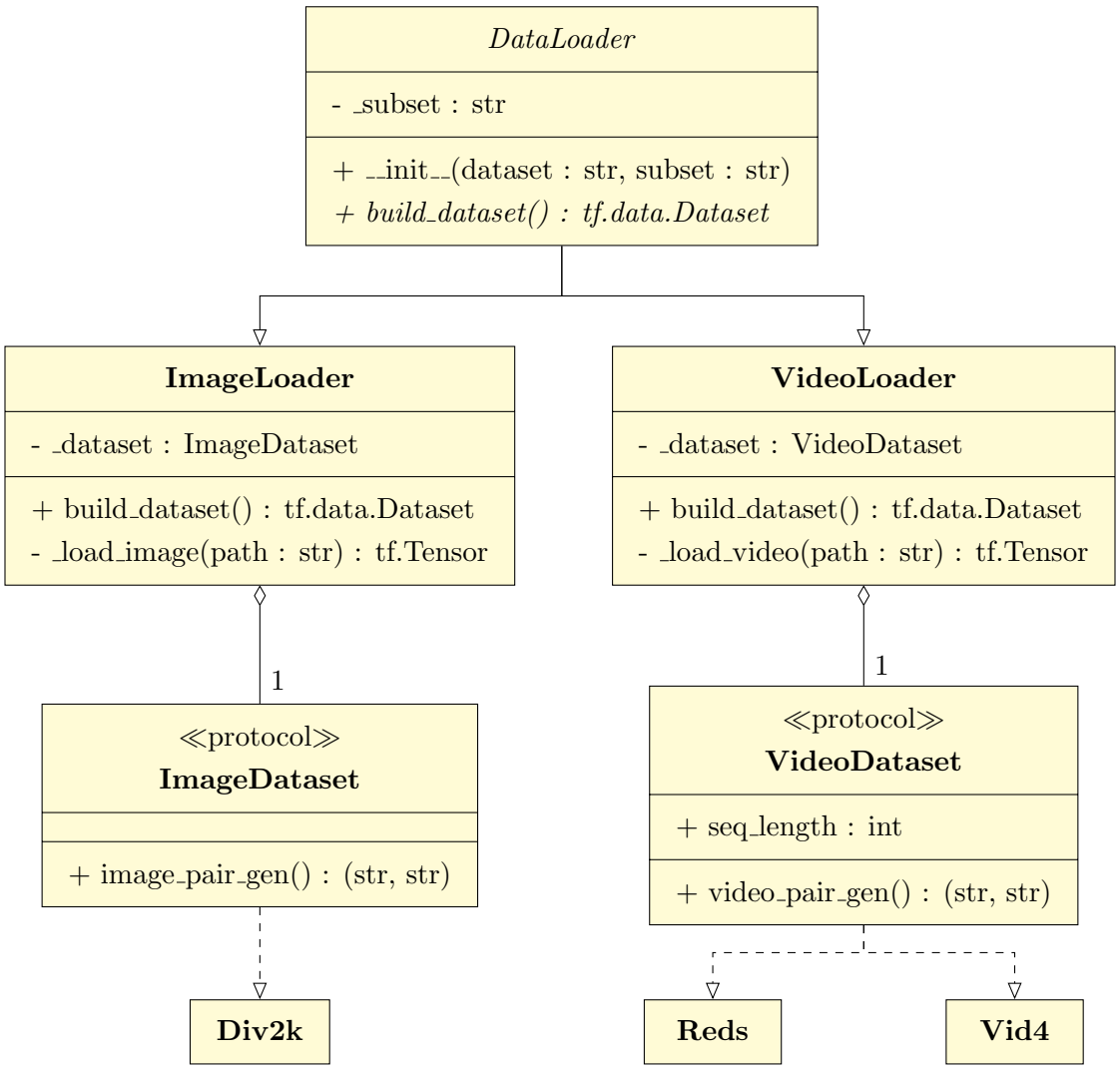


Figure 3.1: Dataset loading pipeline.

LSTM, the convolutional LSTM (Conv-LSTM) [40]. This replaces all fully-connected operations with convolution, making it suitable for image data. Therefore, it was selected as the recurrent unit in the core model.

3.2.1.2 Core model family

As described in §2.5.1, the success criteria stated that a core model was to be designed and implemented, so that the impact of various changes to the design could be evaluated. The design of the core model was a simple CNN architecture, inspired by the EDSR model [9]. However, in order to measure the trade-off between the SR and computational performance of the model, a collection of performance scores over different levels of the two performance types was required. To achieve this, the model had three size configurations (Table 3.2), hereby referred to as Core-S, Core and Core-L, or collectively as the Core model family. These exponentially varied in the number of convolutional blocks and filters per convolution layer, such that larger models would significantly sacrifice computational performance, allowing a resulting relationship with SR performance to be found.

In order to measure the impact of any technique applied to the models' architectures, the

Table 3.2: Size parameters of the Core model family.

Model	Blocks	Filters
Core-S	1	16
Core	4	32
Core-L	16	64

Core model family was first trained in a default configuration (Table 3.3). The technique being measured was then individually applied and a new family of models trained. By comparing the pair-wise performance of the two families, the impact of the technique on the SR-computational performance trade-off could therefore be assessed.

Table 3.3: Core model Parameters. Default values are given in bold text.

Parameter	Values	Details
Size	Small Medium Large	Specifies Core-S, Core or Core-L model, see Table 3.2 for associated size parameters.
Upsample	**Sub-pixel** Deconvolution	Upsampling operation performed by final layer.
Residual	**None** Local Global	No skip connections between blocks. Skip connection from the previous block. Concatenated skip connections from all previous blocks.
Activation	**ReLU** PReLU	Activation function used in blocks.
Activation removal	True **False**	Remove the first activation function from each block.
Recurrent	True **False**	Replace the first and last convolutional layers with Conv-LSTM layers.

3.2.2 Training pipeline

Given a Keras model and TensorFlow `tf.data.Dataset` object, the models were now able to be trained. This section details two significant components of the training pipeline: model checkpointing (§3.2.2.1) and optimisation with variable learning rates (§3.2.2.2).

3.2.2.1 Checkpointing

Models were loaded into the training pipeline by passing model parameters to the `model_loader` module, which would return the corresponding Keras model with randomly initialized weights.

Once the model was loaded, it would be trained on the `tf.data.Dataset` returned by the corresponding `DataLoader` object, requiring a long-running computation before convergence was reached. Therefore, if the session terminated prematurely due to a crash or time-out, a large amount of progress would be lost.

As a solution to this, the model's state — known as a checkpoint — was saved continuously throughout the training process, allowing training to be restarted from the most recent checkpoint before termination. The model's architecture was saved in a JSON format when the model was initialized, then its weights were exported to the HDF5 format and saved every n-th epoch (pass through the entire training dataset), with n defined at the start of the session. The `model_loader` module was then extended to load models from a specified checkpoint, at a file path defined by the model's configuration.

In order to simply interact with the training pipeline, a command line interface (Figure 3.2) was built using the *Argparse* library. This allowed the specification of model parameters and training details (e.g. number of epochs, checkpoint frequency, learning rate) when executing `train.py`.

```
!python training/train.py --epochs 100 --ckpt_epochs 5 --rebuild_freq 50 --batch_size 8 --scale 4 \\
--model core --size m --upscale sp --residual l --activation p --learn_rate 1e-3

Using TensorFlow backend.
Initializing new model
Building datasets
Model arch saved to ./checkpoints/core_MED_SUB_PIXEL_LOCAL_PRELU/arch.json
Inference model arch saved to ./checkpoints/core_MED_SUB_PIXEL_LOCAL_PRELU/arch_infer.json
91/91 [==============================] - 454s 5s/step - loss: 0.0575
Training epoch complete, calculating validation loss...
Train loss: 0.05747    Validation loss:  0.02005
91/91 [==============================] - 114s 1s/step - loss: 0.0206
Training epoch complete, calculating validation loss...
Train loss: 0.02055    Validation loss:  0.01891
91/91 [==============================] - 92s 1s/step - loss: 0.0199
Training epoch complete, calculating validation loss...
Train loss: 0.01992    Validation loss:  0.02022
91/91 [==============================] - 90s 990ms/step - loss: 0.0197
Training epoch complete, calculating validation loss...
Train loss: 0.01965    Validation loss:  0.01684
91/91 [==============================] - 86s 943ms/step - loss: 0.0184
Training epoch complete, calculating validation loss...
Train loss: 0.01843    Validation loss:  0.01658
Model weights saved to ./checkpoints/core_MED_SUB_PIXEL_LOCAL_PRELU/5.h5
```

Figure 3.2: Training pipeline command line interface in Google Colaboratory.

3.2.2.2 Variable learning rate

In the original implementation of the training pipeline, a constant learning rate η was used by the optimiser for all NN layers, as is typical for NN training. However, whilst training different model configurations, the performance of the deconvolution upsampling operation on the training dataset was found to be anomalously worse than sub-pixel convolution. In order to investigate this, the original paper [3] which used deconvolution for upscaling was reexamined. In this, Dong et al. suggest the use of a lower η for the deconvolution layer, 0.0001 compared to 0.001 for the other layers.

Keras does not support variable learning rates within models, so a third-party Python package, *Keras LR Multiplier*, was used to implement this solution. After decreasing η to 0.0001 for the deconvolution layer only, the model's performance on the training dataset improved

significantly, so the variable learning rate was kept for further evaluation of the deconvolution layer (see §4.4.2).

3.2.3 Model tuning

Once the models and the training pipeline were implemented, the models were trained. In order to maximise the performance of the models, a tuning process was applied. In this, the model's hyper-parameters (§3.2.3.1), followed by its weights (§3.2.3.2) were tuned.

3.2.3.1 Hyper-parameter tuning

When training NN models, their hyper-parameters — that is, the model parameters that are determined before training — can have a significant effect on the performance of the trained model. When training the previously described models, two hyper-parameters were therefore tuned to optimise performance: learning rate η and optimisation algorithm (optimiser). Since η is continuous, a set of exponentially varying samples $\eta \in \{0.1, 0.01, 0.001, 0.0001, 0.00001\}$ were selected as candidates. In order to define this range, the most commonly used η from reference implementations of existing SR models, 0.001, was found and used as the range's median. The candidate optimisers were similarly chosen by examining optimisers used in reference implementations: stochastic gradient descent [35], RMSProp [41] and Adam [42].

The optimal hyper-parameter values were selected from a grid search over all combinations of candidate η and optimiser. However, since training of these models requires a significant amount of time (approximately 3 hours) it was not feasible to tune the hyper-parameters for every model configuration. Therefore, the Core model with default parameters was selected to perform the grid search.

In order to evaluate the performance of each configuration in the grid search, a subset of the data not used for training was required, otherwise whichever configuration maximally overfit the training data would be selected. The test dataset could be used for this, however tuning the model's hyper-parameters on the same dataset it was to later be evaluated on would lead to overfitting of the test dataset. For this reason, a subset of the training dataset, known as the *validation set*, was excluded from model training and used for continuous evaluation of models during the training process. The training dataset was split so that approximately 20% of the frames were used for validation, with the remainder being used for training.

As a result of the grid search, the optimal hyper-parameters were found to be the **Adam optimiser** with $\boldsymbol{\eta = 0.001}$.

3.2.3.2 Fine-tuning

Model fine-tuning refers to the process of using the weights from a pre-trained model to initialise a new model for further training. While this technique can be used to adapt a model from a similar domain to a specific dataset, it can also be applied when training a single model in order to improve the performance and speed of convergence. This is accomplished with *learning rate decay*, in which the η decreases over the duration of training. This allows large weight updates early in training, followed by fine-tuning of the weights towards the end of training.

This technique was used for the training of SR models, by halving η every 75 epochs, with the number of epochs determined by the average required for convergence of the Core model when hyper-parameter tuning. In addition to this, the batch size was doubled with the same frequency, so as to generate progressively more useful weight updates over the learning process. When the model's performance no longer improved after an iteration of this process, training was halted.

3.3 Inference package

Using the set of trained models described in §3.2, the inference package was implemented. This was composed of an inference pipeline for upsampling input videos (§3.3.2), as well as a framework for evaluating the performance of trained models (§3.3.3). This section first presents an overview of the package's implementation (§3.3.1), then gives further details regarding each of these two components.

3.3.1 Package overview

An overview of the package is presented in Figure 3.3. The package is designed to allow developers with no knowledge of SR to upsample input videos without specifying a model, using the `Vsr` class. A suitable model is first selected with the `configure` method, then it is executed on input videos by calling the `execute_video` method. All loading of and inference by the corresponding `tf.keras.Model` object is handled in the `Upsampler` class, which acts as an interface for the execution of a single model.

The package is also designed such that SR researchers can evaluate the performance of a trained model, or bicubic interpolation, by passing the corresponding `ModelArgs` object to the `Evaluation` module.

3.3.2 Inference pipeline

As discussed in §1.1, many use cases of VSR require execution in real-time, at or above a certain frame rate. For a user with no technical knowledge of SR, an ideal system would be able to select whichever model achieves the highest SR performance, whilst meeting their computational performance requirements.

In order to achieve this, the results of the technique evaluation (see §4.4) were used to define a set of model configurations, which were ordered by their SR performance. The `configure` method iterates over this set, evaluating each model with the `Evaluation` module (see §3.3.3). The first model to achieve a sufficient execution speed is selected and loaded through an `Upsampler` instance.

In subsequent calls to the `execute_video` method, the loaded model is executed on the given input video. After execution, the model's output is saturated, clipping each value into the valid range, and exported to the source file format.

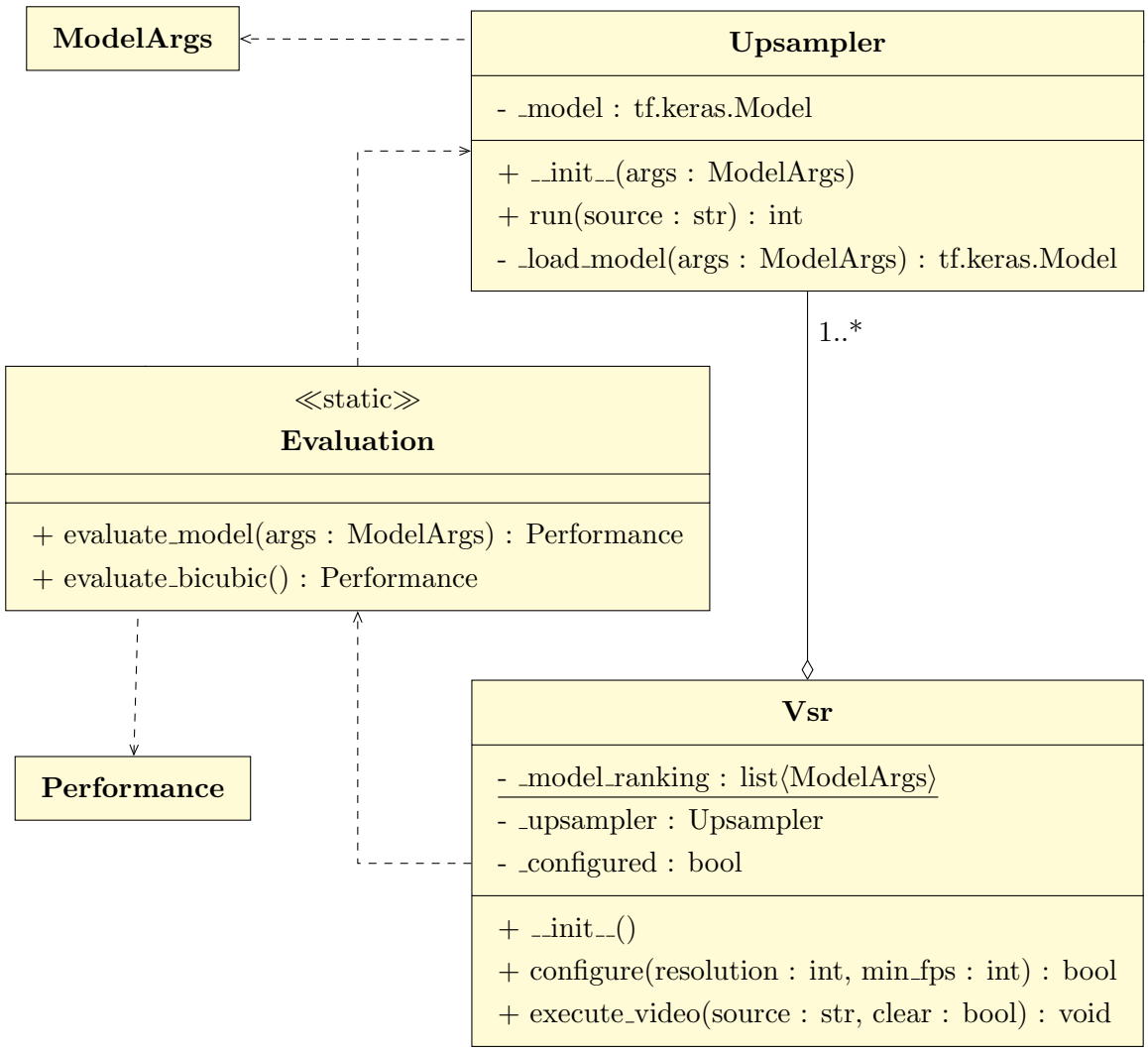


Figure 3.3: Overview of key inference package components.

3.3.3 Evaluation framework

In order to evaluate the output of network inference, an evaluation framework was constructed. This used instances of the `Upsampler` object to load and execute pre-trained models on the test dataset (Vid4), then computed the SR and computational performances of the output, based on the metrics defined in §4.1. As part of this, the time taken to complete inference on the dataset was required. Ideally, this would return only the time taken for the model to execute the dataset; however, the time taken to build the model and load the dataset into memory added noise to this measurement, making it less useful for comparison between models. In theory, these processes would only occur on the first execution of the model, meaning subsequent inferences on the same dataset would not suffer from the same noise. After performing consecutive inferences with the Core model, it was observed that the time taken to complete the first inference was anomalously higher than the following inferences, thereby confirming this hypothesis. Therefore, a 'warm-up' run over the entire test dataset was executed by the `Evaluation` module, prior to the evaluation being performed.

As with the training pipeline (see §3.2.2), a command line interface was designed for simple interaction with the evaluation framework. This was used to complete the evaluation described in Chapter 4, thereby demonstrating its functionality.

3.4 Repository overview

The project repository (Figure 3.4) is primarily composed of two packages: `training` and `inference`. The `training` package contains the source code for data handling (§3.1) and model training (§3.2), whilst the `inference` package contains the inference pipeline (§3.3.2) and evaluation framework (§3.3.3). Components common to both packages, such as the SR datasets (`data`) and model checkpoints (`checkpoints`), are located outside of these packages. This ensures no dependencies exist between the two packages, meaning they can be developed and imported in isolation.

- `video-super-resolution`
  - `checkpoints` — Training checkpoints.
  - `data` — SR datasets for model training and evaluation (§3.1.1).
  - `inference` — VSR inference package (Figure 3.3).
    - `evaluation.py` — Model evaluation framework (§3.3.3).
    - `upsampler.py` — VSR inference pipeline (§3.3.2).
    - `vsr.py` — Interface to the inference pipeline.
  - `notebooks` — Jupyter notebooks used in Google Colaboratory.
    - `colab_train.py` — Model training interface.
    - `sisr_demo.py` — Demonstration of SISR inference, for debugging.
  - `training` — Source code for model training.
    - `datasets` — Implementation of the dataset protocols (Figure 3.1) for each dataset.
    - `models` — Implementation of Core (§3.2.1.2), FSRCNN [3] and EDSR [9] models.
    - `data_loader.py` — `DataLoader` implementations (Figure 3.1), for building datasets.
    - `model_loader.py` — Model initialization and checkpointing.
    - `train.py` — Model training pipeline (§3.2.2).
    - `utils.py` — Common data processing scripts.
  - `LICENCE.txt` — Project's MIT Licence.
  - `README.md` — Brief overview of the project.
  - `requirements.txt` — List of project dependencies, for building the virtual environment.

Figure 3.4: Repository overview.

Chapter 4

Evaluation

In this chapter, the evaluation of the techniques presented in §2.4, using the implemented evaluation framework (§3.3.3), is given. Firstly, key components of the evaluation — performance metrics (§4.1) and statistical significance testing (§4.2) — are presented. The results of the evaluation against a baseline method (§4.3) and of the individual techniques (§4.4) are then given and used to construct and evaluate an improved model (§4.5). Finally, the success of the project is summarised by reviewing the success criteria (§4.6).

4.1 Performance metrics

The aim of this dissertation was to explore the impact of various advances in CNN-based SR (see §2.4) on *real-time performance*. This requires the trade-off of two measures: *computational* and *SR performance*. Computational performance regards the general execution speed of the model, while SR performance regards the quality of reconstruction of the HR image. In this section, multiple metrics are proposed in order to quantitatively evaluate these.

4.1.1 SR performance

4.1.1.1 Peak signal-to-noise ratio

Peak signal-to-noise ratio (PSNR) is a popular metric for the quantitative evaluation of image reconstruction algorithms. It measures the ratio of the maximum possible signal power to the power of the corrupting noise affecting the reconstruction. In the context of SR, it is therefore defined by the maximum pixel value P and the mean squared error between I^{HR} and I^{SR}.

In order to develop SR models for real-world applications, an ideal metric should measure the level of *perceptible difference*. By being based solely on pixel difference, PSNR provides a coarse approximation of this. Caballero et al. [24] proposed improving the effectiveness of PSNR in this regard by taking the *luminance* of each pixel, since this provides a more perceptually meaningful measure than each of the RGB colour channels. In order to implement this, each image was converted from RGB to the YCbCr[1] colour space and the Y-channel (luminance) was kept. This gave two-dimensional transformed images $I^{\mathrm{HR}(Y)}, I^{\mathrm{SR}(Y)} \in [0,1]^{H \times W}$, for which PSNR is defined as follows:

$$\mathrm{PSNR}(I^{\mathrm{HR}(Y)}, I^{\mathrm{SR}(Y)}) = 10 \cdot \log_{10}\left(\frac{P^2}{\frac{1}{HW}\sum_{h=1}^{H}\sum_{w=1}^{W}(I^{\mathrm{HR}(Y)}(h,w) - I^{\mathrm{SR}(Y)}(h,w))^2}\right). \quad (4.1)$$

[1] The ITU-R BT.601 YCbCr standard is used, as in the source paper [24].

4.1.1.2 Structural similarity

The structural similarity index (SSIM) [43] is a perception-based metric for image similarity, which measures the perceived change in structural information. This is achieved by estimating the change in luminance, contrast and structure between the images. Here, the luminance μ_I and contrast σ_I of an image I are approximated by the mean and standard deviation of the image's Y-channel,

$$\begin{aligned} \mu_I &= \frac{1}{HW}\sum_{h=1}^{H}\sum_{w=1}^{W} I^{(Y)}(h,w) \\ \sigma_I &= \sqrt{\frac{1}{HW-1}\sum_{h=1}^{H}\sum_{w=1}^{W}\left(I^{(Y)}(h,w)-\mu_I\right)^2}, \end{aligned} \tag{4.2}$$

while the structural similarity $\sigma_{I^{\mathrm{HR}}I^{\mathrm{SR}}}$ between the two images is approximated by their covariance,

$$\sigma_{I^{\mathrm{HR}}I^{\mathrm{SR}}} = \frac{1}{HW-1}\sum_{h=1}^{H}\sum_{w=1}^{W}\left(I^{\mathrm{HR}(Y)}(h,w)-\mu_{I^{\mathrm{HR}}}\right)\left(I^{\mathrm{SR}(Y)}(h,w)-\mu_{I^{\mathrm{SR}}}\right). \tag{4.3}$$

From these, independent comparison functions for luminance $\mathcal{C}_l$, contrast $\mathcal{C}_c$ and structure $\mathcal{C}_s$ are derived,

$$\begin{aligned} \mathcal{C}_l(I^{\mathrm{HR}},I^{\mathrm{SR}}) &= \frac{2\mu_{I^{\mathrm{HR}}}\mu_{I^{\mathrm{SR}}}+c_l}{\mu_{I^{\mathrm{HR}}}^2+\mu_{I^{\mathrm{SR}}}^2+c_l} \\ \mathcal{C}_c(I^{\mathrm{HR}},I^{\mathrm{SR}}) &= \frac{2\sigma_{I^{\mathrm{HR}}}\sigma_{I^{\mathrm{SR}}}+c_c}{\sigma_{I^{\mathrm{HR}}}^2+\sigma_{I^{\mathrm{SR}}}^2+c_c} \\ \mathcal{C}_s(I^{\mathrm{HR}},I^{\mathrm{SR}}) &= \frac{\sigma_{I^{\mathrm{HR}}I^{\mathrm{SR}}}+c_s}{\sigma_{I^{\mathrm{HR}}}\sigma_{I^{\mathrm{SR}}}+c_s} \end{aligned} \tag{4.4}$$

where c_l, c_c and c_s are constants for stability. SSIM is calculated by a weighted product of these functions,

$$\mathrm{SSIM}(I^{\mathrm{HR}},I^{\mathrm{SR}}) = \left[\mathcal{C}_l(I^{\mathrm{HR}},I^{\mathrm{SR}})\right]^{\alpha}\cdot\left[\mathcal{C}_c(I^{\mathrm{HR}},I^{\mathrm{SR}})\right]^{\beta}\cdot\left[\mathcal{C}_s(I^{\mathrm{HR}},I^{\mathrm{SR}})\right]^{\gamma} \tag{4.5}$$

where α, β and γ are control parameters. These can be tuned to determine the relative importance of each component, however they are set to $\alpha=\beta=\gamma=1$ in this report, as is standard in SR literature.

4.1.2 Computational performance

4.1.2.1 Execution speed

An intuitive metric for the computational performance of an SR model is the number of frames processed in a fixed time interval, otherwise referred to as frame rate or frames-per-second (FPS). This is useful for benchmarking the model, since most use cases with a requirement on

computational performance will specify it in FPS.[2] It also provides a useful indication of the real-world performance of a model.

However, the metric is dependent on many implementation-specific factors, namely the executing hardware and implementation of model operations, meaning FPS values are not generally comparable. To minimise the effect of this within this report, all models were evaluated on the same hardware (Tesla K80 GPU) using the same inference backend (TensorFlow), providing an estimation of performance in a real-world application.

4.1.2.2 Multi-Adds

An alternative metric for computational performance is the number of multi-add operations executed by the model, in order to perform inference on a single frame at a specific resolution [45]. This provides a theoretical and implementation-agnostic measure of performance, making the values consistent and generally comparable. However, this does not perfectly approximate real-world performance due to implementation-specific factors.

For example, the implementation of common operations, such as general matrix multiplication and convolution, are usually optimised to a greater extent than uncommon operations in standard NN libraries. This means that a model containing many uncommon operations is likely to achieve worse real-world performance than another model with an equal number of multi-adds but common operations. In addition to this, 'broad' models — that is, shallow models with a high number of convolutional filters per layer — are able to execute a large number of operations in parallel. Whereas, deep models with an equal number of multi-adds will be forced to execute more operations sequentially, which may not utilise the parallelisation capability fully.

Due to this, FPS was used as the sole metric for the evaluation of computational performance, so as to give a useful indication of performance in the real-world deployment of the models.

4.2 Significance testing

When comparing the performance of models, we wish to determine which model will generalise better — that is, generally outperform the other model on unseen data — under a particular metric. The models' results on a particular dataset provides an approximation of this, however we must determine whether the difference in their results is a significant indication of improved performance or a result of noise. In order to do this, statistical significance tests are used.

As the null hypothesis, we assume that the results for each model are generated from the same probability distribution. The likelihood, known as the p-value, of obtaining the achieved results given the null hypothesis is then calculated. If the p-value is less than a predefined threshold, the null hypothesis is rejected and the results are deemed statistically significant. Otherwise, we conclude there is insufficient evidence to determine which model outperforms the other, thereby accepting the null hypothesis.

[2]The standard threshold for the perception of motion, rather than individual frames, from video is 24 FPS [44]. This is consequently used as the threshold for 'real-time' computational performance in this report.

In the context of this evaluation, the SR performance of the core model on the test dataset was to be compared to a variant of the core model with some technique applied. Since each model was evaluated on the same frames, matched pairs (x_i, y_i) of PSNR or SSIM scores from each model on the i-th frame could be generated. By using each pair's difference $d_i = y_i - x_i$ as the sample set, the *paired Student's t-test* was therefore used to calculate statistical significance. This computes the test statistic t from the set of differences d, as follows:

$$t = \frac{\mu_d}{\sigma_d / \sqrt{n}} \tag{4.6}$$

where μ_d and σ_d are respectively the mean and standard deviation of d, while n is the number of difference samples.

In order to calculate the p-value from this, we then compare t to the t-distribution with $n-1$ degrees of freedom. Since the alternative hypothesis was that the two models had unequal performance, rather than a specific model outperforming the other, a *two-tailed* test was used. Therefore, as the criterion for rejecting the null hypothesis in either direction, we require

$$p = 2 \cdot Pr(T > |t|) < \alpha \tag{4.7}$$

where T is a random variable distributed by the aforementioned t-distribution and α is the significance threshold. As is standard in SR literature, the significance threshold was set to $\alpha = 0.01$, such that the null hypothesis must be rejected with at least 99% confidence. Furthermore, results were deemed 'very significant' if they achieved $p < 0.001$.

4.3 Baseline comparison

In order to better understand the relative magnitude of the performance differences between models, it is helpful to have a reference value for each metric, which can act as a baseline for the comparison of performance statistics. Often, the optimal value of each metric is used for this, so as to demonstrate how close models are to a perfect fit of the data. However, as I^{SR} tends towards a perfect reconstruction of I^{HR}, the PSNR of the reconstruction approaches infinity. This is not a useful baseline for comparison, meaning another baseline must be set.

The simplicity of bicubic interpolation (§2.3.1) makes its performance a standard baseline for evaluating SR models, so it is used in this report. In addition to this, since any SR model is likely to be more complex than bicubic interpolation, it would be expected to achieve higher performance. Therefore, in order to ensure the Core model family was correctly learning a mapping from I^{LR} to I^{SR} which approximated I^{HR}, the performance of Core-S, Core and Core-L were evaluated against bicubic interpolation (Table 4.1).

The Core-S and Core models achieved significantly better PSNR and SSIM scores than bicubic interpolation, thereby confirming that the model's architecture was suitable for the task, in addition to that the training pipeline was functioning correctly. However, the Core-L model was outperformed by bicubic interpolation, although not significantly. As discussed in §2.4.2, the phenomenon of deep, non-residual models being outperformed by their shallower counterparts has previously been observed [20], making this result unsurprising given the depth of the large model. However, in order to determine the SR-computational performance trade-

off when the techniques under evaluation are applied to deep models, the convergence of the Core-L model needed to improve. Since RNNs are specifically designed to improve the training of deep networks, the residual learning methods were therefore evaluated first (see §2.4.2).

Table 4.1: Mean SR performances of bicubic interpolation and the Core model family on Vid4, ± the standard deviation. Significant results ($p < 0.01$) against bicubic interpolation are underlined.

Model	**PSNR**	**SSIM**
Bicubic	22.47 ± 1.03	0.614 ± 0.051
Core-S	$\underline{22.87 \pm 1.15}$	$\underline{0.642 \pm 0.049}$
Core	$\underline{23.23 \pm 1.16}$	$\underline{0.660 \pm 0.047}$
Core-L	22.38 ± 1.11	0.609 ± 0.049

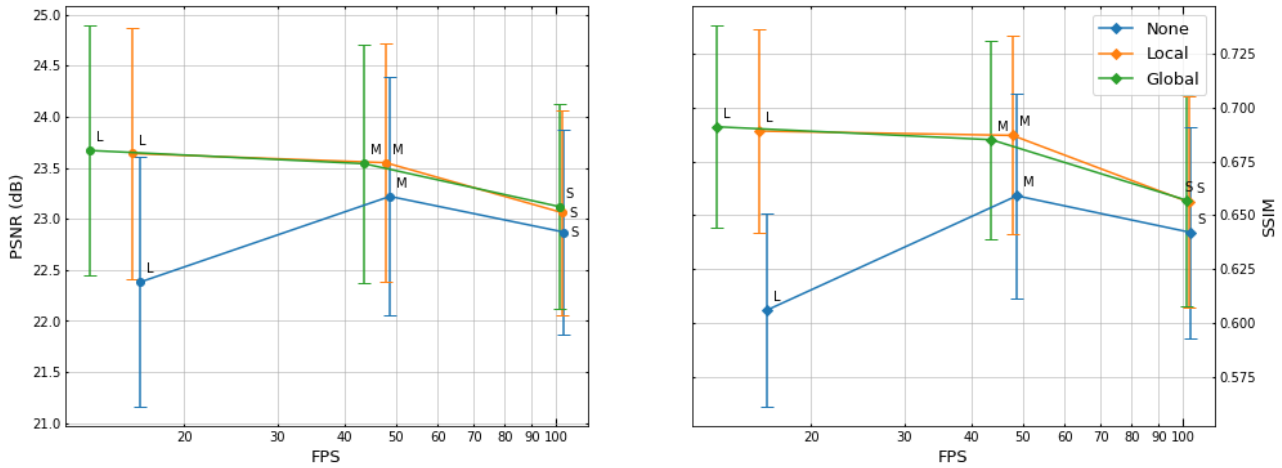
4.4 Technique evaluation

This section presents the results of the evaluation of each technique (see §2.4) on the Vid4 dataset [34], discussing their comparative SR-computational performance trade-offs. The figures referenced in §4.4.1–4.4.4 contain linear-log plots of PSNR and SSIM against FPS for the default configuration and the techniques under review, demonstrating their SR-computational performance trade-offs. The x-axis was made logarithmic due to the size, and therefore FPS, of the Core-S, Core and Core-L models varying exponentially. The referenced tables present the results of the paired t-test for the variants against the default configuration, with '=', '>' and '≫' denoting no, some and very significant differences respectively.

4.4.1 Residual learning

The Core model family was evaluated after local and global residual learning techniques (see §2.4.2) were applied to it. Figure 4.1 presents the SR-computational performance trade-offs of the default configuration (no residual learning) and the two variants, while Table 4.2 presents the significance of the results. As expected, the PSNR and SSIM scores of the large model improve very significantly with residual learning, supporting the hypotheses presented by Lim et al. [9] and Ahn et al. [4]. However, the SR performances of both the small and medium models are also significantly improved by residual learning, which was unexpected.

In comparing the two residual learning methods, we observe an insignificant difference in SR performance over each model size, although the FPS of the local residual models is higher than the global residual model in the large and medium variants. This is as expected, given globally residual networks will generate larger concatenated tensors as more residual blocks are added. Due to this, the real-time performance of the local residual models is marginally superior to the global residual models. Given this, in addition to local residual learning introducing less model complexity than global residual learning, the default configuration of the Core model family was updated to use local residual learning for all further analysis.



Local vs. none

Size	PSNR	SSIM
Small	$>$	$\gg$
Medium	$\gg$	$\gg$
Large	$\gg$	$\gg$

Global vs. none

Size	PSNR	SSIM
Small	$\gg$	$\gg$
Medium	$\gg$	$\gg$
Large	$\gg$	$\gg$

Figure 4.1 & Table 4.2: Results of using local and global residual learning against no residual connections, in terms of PSNR vs. FPS (**top left**) and SSIM vs. FPS (**top right**).

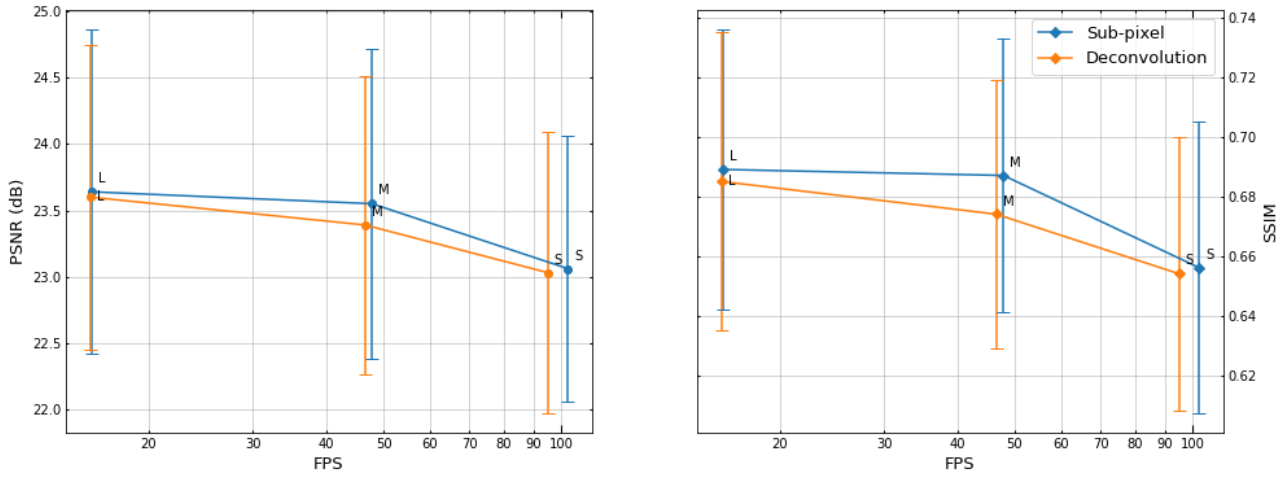
4.4.2 Post-upsampling

Using the updated Core model family with local residual learning (see §4.4.1), the models were retrained using deconvolution as the post-upsampling method, in contrast to sub-pixel convolution as used in the default configuration (see §2.4.1). Figure 4.2 presents the SR-computational performance trade-offs, while Table 4.3 presents the significance of the results. Sub-pixel convolution significantly outperforms deconvolution on both SR metrics in the medium-sized model, however there is no significant difference in the small and large models. The FPS of the deconvolution models is also worse than the sub-pixel models, particularly in the small model. Given the deconvolution operation performs convolution with a filter, while sub-pixel convolution only rearranges data in memory, the difference in computational performance is as expected. In addition to this, the computational cost of the operations are constant, so the difference would be expected to be magnified for smaller models.

From these results, sub-pixel convolution was concluded to achieve slightly better real-time performance than deconvolution, so the default configuration was not changed in further analysis. However, since the performance of the two techniques was insignificantly different over the small and large models, it was possible that further fine-tuning of the medium-sized deconvolution model might still improve its SR performance. This was attempted, but no improvement in SR performance was achieved.

4.4.3 Activation functions

The updated Core model family was retrained with two new techniques, both varying the network's activation functions. These were the use of the PReLU, rather than ReLU, activation



Deconvolution vs. sub-pixel

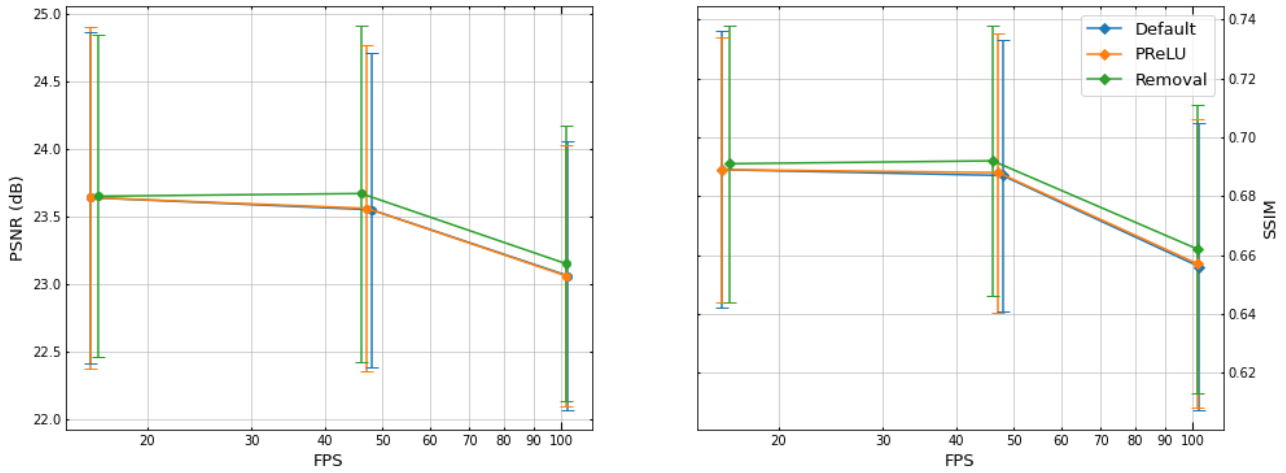
Size	PSNR	SSIM
Small	=	=
Medium	<	<
Large	=	=

Figure 4.2 & Table 4.3: Results of using deconvolution against sub-pixel convolution, in terms of PSNR vs. FPS (**top left**) and SSIM vs. FPS (**top right**).

function and the removal of half of the activation operations from the network (see §2.4.3). Again, Figure 4.3 presents the SR-computational performance trade-offs, with Table 4.4 containing the statistical significance of the differences.

As was discussed in §2.4.3.1, Dong et al. [3] propose the use of the PReLU activation function, claiming to have observed improved performance against ReLU in undisclosed experiments. Since, to the best of my knowledge, the PReLU activation function has not been used in any published SR models since the publication of this work, it is implicit that other researchers have not found similar results. From this experiment, we observe no significant performance difference between PReLU-activated networks and their ReLU-activated counterparts, at any size configuration. Therefore, these results oppose the hypothesis of Dong et al., that the PReLU activation function improves SR model performance.

Activation removal (see §2.4.3.2) was proposed by Li et al. [5], who presented the SR performance of a range of removal proportions over SISR datasets, concluding that 1/2 gave the optimal performance on their model, s-LWSR. Being a recent paper (Sept. 2019), this experiment has not, to the best of my knowledge, been recreated. The results given in Figure 4.3 demonstrate a significant improvement in the SR performance of the small- and medium-sized models after activation removal is applied (with the exception of the medium-sized model's SSIM). The large model showed no significant performance difference, however this is unsurprising given the technique was proposed for retention of information in small, lightweight models like s-LWSR. Since this report is primarily interested in models of this type, the Core model family was updated with activation removal for all further analysis.



PReLU vs. ReLU

Size	PSNR	SSIM
Small	=	=
Medium	=	=
Large	=	=

Removal vs. full

Size	PSNR	SSIM
Small	>	>
Medium	>	=
Large	=	=

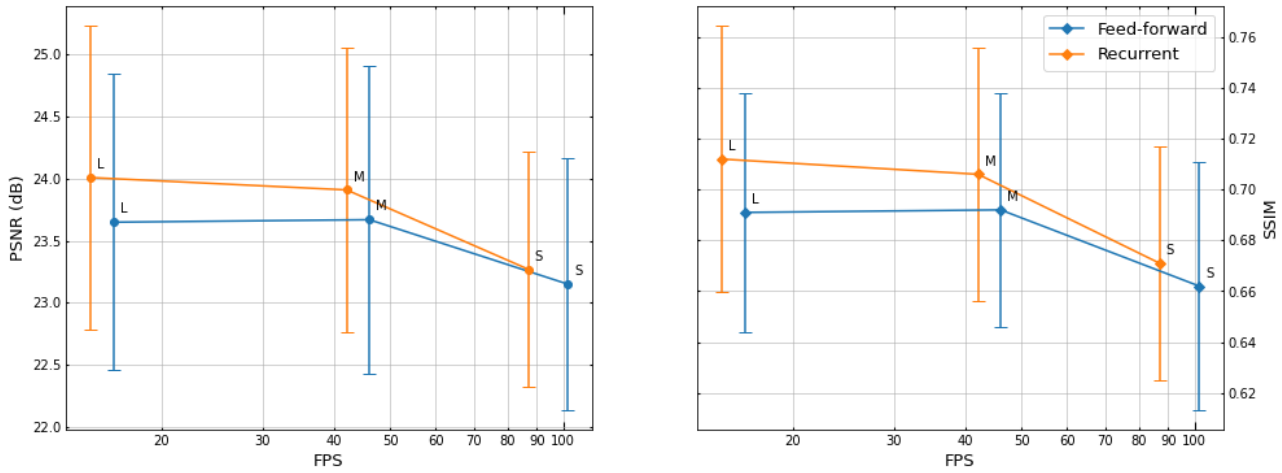
Figure 4.3 & Table 4.4: Results of using PReLU and activation removal against the default fully-activated ReLU network, in terms of PSNR vs. FPS (**top left**) and SSIM vs. FPS (**top right**).

4.4.4 Frame-recurrence

Finally, frame-recurrence using Conv-LSTM as the recurrent unit (see §2.4.4 and §3.2.1.1) was evaluated on the updated Core model family (Figure 4.4). The frame-recurrent models significantly outperformed the default feed-forward model over all model sizes (Table 4.5). However, the addition of two Conv-LSTM layers in place of convolutional layers decreased the FPS of all model sizes, in particular the small model. This is unsurprising given the increased complexity of the LSTM operation, in addition to the requirement that frames be executed sequentially. Due to this, individually comparing the SR performances of the models is not fair, since the feed-forward models could have increased in complexity in order to achieve equal computational performance. However, when analysing the SR-computational performance trade-off (characterised by the integral of the best fit curve over data points), the frame-recurrent models appear to outperform the feed-forward models. Since this determines the real-time performance of the model, we can conclude that the use of frame-recurrence improves real-time performance.

4.5 Improved model evaluation

Based on the evaluation of techniques presented in §4.4, an improved model architecture was designed. This extended the original Core model architecture with *local residual learning*, *activation removal* and *frame-recurrence*, using *sub-pixel convolution* to perform post-upsampling. As shown in Figure 4.4, in which the recurrent model implements this architecture, the large variant of this model did not meet the required FPS for real-time performance (24 FPS). Therefore, the medium-sized variant was used for evaluation in this section, as it gave the best SR



Recurrent vs. feed-forward

Size	PSNR	SSIM
Small	$>$	$>$
Medium	$>$	$\gg$
Large	$\gg$	$\gg$

Figure 4.4 & Table 4.5: Results of using frame-recurrence against the default feed-forward architecture, in terms of PSNR vs. FPS (**top left**) and SSIM vs. FPS (**top right**).

performance whilst being suitable for real-world deployment.

In order to determine the model's effectiveness on different image patterns, the performance of the model on each of Vid4's videos was examined (Figure 4.5). This demonstrates a large variance between mean video PSNR scores, with relatively little variance within videos. Since high-frequency information is lost when the target image I^{HR} is degraded to produce I^{LR}, it would be expected that target images with high entropy are more difficult to reconstruct. To qualitatively evaluate this, the entropy of 10×10 regions over the first HR frame of each Vid4 video were visualised (Figure 4.6).[3] From this, we observe that the target image with the lowest PSNR ('calendar') contains more regions with high entropy than the image with the highest PSNR ('walk'), thereby supporting the hypothesis. Based on this, an area for further research would be the construction of an entropy-based loss function, which increases the weighting of target image regions with higher entropy. This would encourage learning fine details within training images, rather than smooth regions with low entropy.

In summary, the SR performance of the improved model greatly exceeds the baseline method, whilst achieving real-time computational performance (24 FPS) on the Vid4 task. From this, the extension non-functional deliverable described in §2.5 has been fulfilled. Example predictions by the two methods on a validation image from the DIV2K dataset are presented in Figure 4.7, qualitatively demonstrating the greater level of detail restored by the improved model.

[3]The implementation of the entropy visualisation was inspired by an online guide: `https://www.hdm-stuttgart.de/~maucher/Python/MMCodecs/html/basicFunctions.html`.

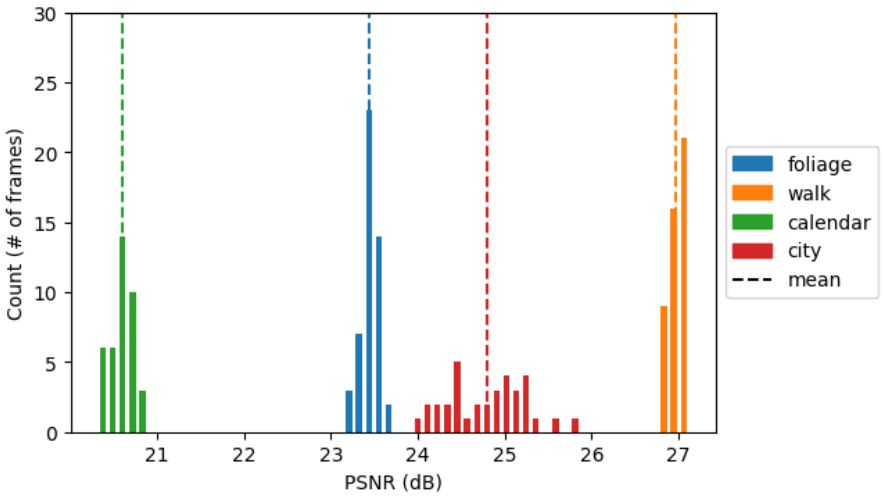


Figure 4.5: Vid4 performance breakdown.

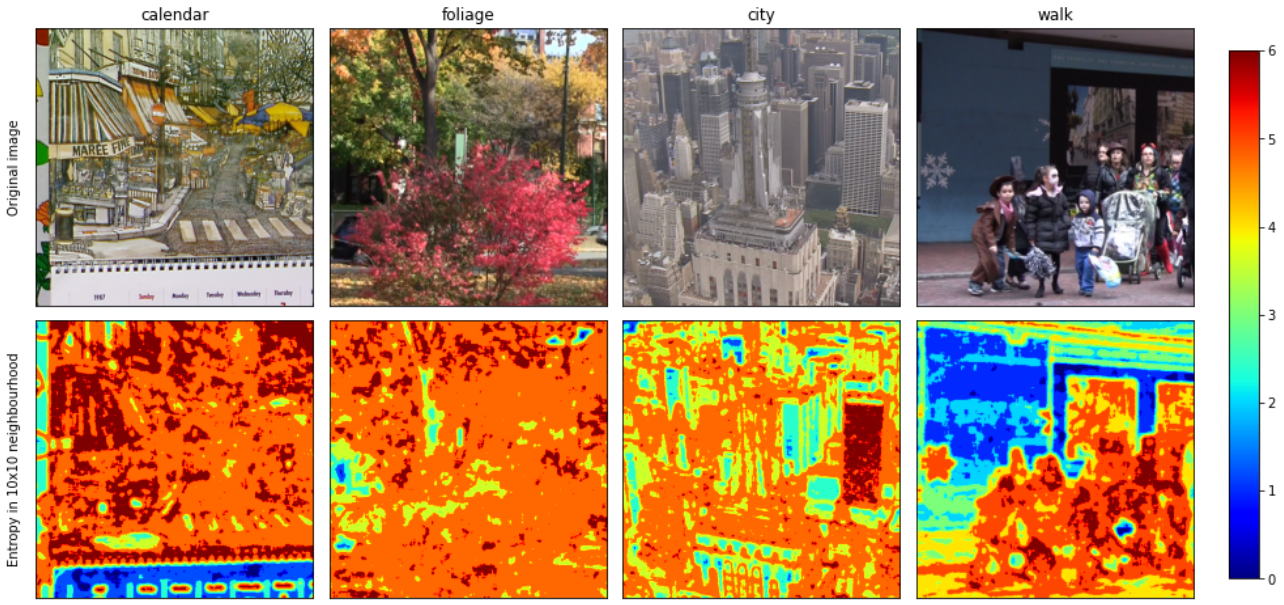


Figure 4.6: First frame entropy of Vid4 videos.

Figure 4.7: Improved model demonstration on '0846' from DIV2K, with ×4 upsampling.

4.6 Success criteria

The success criteria have all been satisfied. In addition to this, two extensions have been completed, with the remaining extension (**Extension 2**) being dismissed, as discussed below. This section details how this was performed, with reference to the relevant sections in this report.

Since refinements were made to the success criteria (see §2.5.1), the criteria given in this section are adapted from the original project proposal (Appendix A). **Criterion 2** is adapted from the 'model training' section of original criteria, to move from the implementation of specific models to the implementation of a core model, upon which various techniques are applied. **Criteria 3 and 4** are also modified in order to reference an inference pipeline, rather than a generic SR application. The successful completion of a criterion is denoted by (✓), while (✗) denotes the criterion not being accomplished. A new extension criterion was proposed, which is denoted by (✓†).

- **Criterion 1**: *Collect an image and/or video dataset to be used for training SR models.* (✓)

  Div2K [32, 33] and REDS [31] were selected for training of SISR and VSR models respectively, whilst Vid4 [34] was selected for evaluation of all models (§3.1.1). The training datasets were pre-processed (§3.1.2) and an efficient data loading pipeline was constructed to access the datasets during training (§3.1.3).

- **Criterion 2**: *Perform research into SISR techniques, then implement and train a core model with each technique applied.* (✓)

A literature review of SR models was performed to select candidate SR model techniques (§2.4), with seven techniques across four general categories (post-upsampling, residual learning, activation functions and frame-recurrence) being selected.

An extensible family of models was designed and implemented (§3.2.1) and a training pipeline was constructed (§3.2.2). The models were then trained and fine-tuned using the pipeline (§3.2.3).

- **Criterion 3**: *Evaluate the performance of the trained models, selecting which will be used in the inference pipeline.* (✓)

  Each candidate technique was evaluated over SR and computational metrics (§4.1) with significance testing (§4.2) against the default configuration of the model family (§4.4). The techniques providing improvements in real-time performance were consequently determined and an improved model was constructed and evaluated (§4.5).

- **Criterion 4**: *Implement an inference pipeline to apply the selected models to an input video.* (✓)

  An inference pipeline was designed and implemented (§3.3.2), performing the execution of pre-trained models on input videos. This was included in the VSR package, described in **Extension 3**.

- **Extension 1**: *Implement a video-based SR model.* (✓)

  The frame-recurrence technique was researched (§2.4.4) and implemented using ConvLSTM recurrent units (§3.2.1.1). A video training dataset was selected for the training of recurrent models, as described in **Criterion 1**, and the models were subsequently trained and evaluated, as in **Criteria 2 and 3**.

- **Extension 2**: *Execute SR on subsections of the frame, determined by some heuristic (localised SR).* (✗)

  Localised SR was hypothesised during the planning phase of this project, as a possible technique for accelerating SR applications. However, after achieving initial results from SR models, it was decided that this technique would likely introduce noticeable artefacts at the border of the subsection, lowering the perceptual quality of the reconstruction. In addition to this, after the focus of the project was changed to the evaluation of SR techniques, rather than the implementation and acceleration of a generic SR application, localised SR became less relevant to the project's primary goal.

- **Extension 3**: *Implement a general-purpose VSR package, providing an API for out-of-box upscaling of videos.* (✓†)

  A general-purpose VSR inference package was designed (§3.3.1), using the inference pipeline from **Criterion 4** in order to perform VSR on input videos. This package also provided an evaluation framework (§3.3.3), allowing pre-trained models to be benchmarked in the executing environment. Using the models selected from the evaluation described in **Criterion 3**, the model with highest SR performance, which met the user's computational performance requirements in the executing environment, would be selected by the package. This evaluation framework was used to perform the previously described evaluation, demonstrating its functionality.

Chapter 5

Conclusion

This dissertation has presented the research, implementation and evaluation of seven techniques for CNN-based SR models, determining their impact on real-time performance. Those providing significant performance improvements were implemented in an improved model, which was evaluated further. In addition to this, a package was constructed for out-of-box VSR and the evaluation of VSR models.

In this chapter, the results of this dissertation are summarised. Following this, personal lessons learnt from the development of this project are discussed, followed by concerns and recommendations regarding the future of VSR research.

5.1 Results

In Chapter 1, I explained the need for VSR models with high real-time performance, if they are to be deployed in real-world applications. By evaluating the impact of CNN techniques on the real-time performance of CNN-based VSR models, this dissertation has provided an effective guide as to how such a model might be developed. In doing so, the success criteria have been met and exceeded, with the achievement of all core and two extension criteria.

Furthermore, the improved model greatly exceeds both the SR performance of the baseline method and the minimum computational performance requirement. Therefore, this model could feasibly be deployed in a real-time VSR application, providing insight into the near-term nature of commercial VSR. The VSR inference package, implemented in fulfillment of an extension criterion, allows for simple integration of VSR into existing video processing pipelines, with no technical knowledge of the field. In addition to this, by benchmarking the collection of pre-trained models on the executing hardware, the package maximises SR performance whilst meeting the user's computational performance requirements.

For these reasons, I hope this dissertation is able to assist VSR researchers, in addition to developers wishing to integrate VSR into video processing applications, thereby making a meaningful contribution to the field.

5.2 Lessons learnt

This dissertation has been the most daunting project I have undertaken, greatly improving both my confidence and competence in software development and research. In tackling a project of this scope, I have come to understand the importance of decomposing large tasks into smaller components, which can be effectively implemented in isolation. As such, I have found enormous utility in modular development, which enabled the concurrent implementation of the data loading, model training and evaluation pipelines.

With regards to project planning, I now appreciate the importance of deciding and maintaining a consistent direction for a project of this scope. Making the decision to change the focus of the project from the acceleration of a VSR application to an investigation into real-time VSR model performance was difficult and could have been avoided with further thought in the project planning phase. In addition to this, when preoccupied with the development of an individual component, it is easy to lose sight of its purpose in the context of the project and thereby diverge from the intended aim. Therefore, when undertaking similar projects in future, I will be sure to maintain a continual focus on what I am attempting to achieve through the project and how my current work assists in this pursuit.

5.3 Ethical concerns

The concept that "seeing is believing" is embedded in our psychology, due to the rarity with which we perceive failures in the human visual system. As the reconstruction quality of SR moves closer to human imperceptibility, we must therefore be careful not to place absolute trust in the output of SR models. As with other applications of machine learning, these models are vulnerable to dataset bias and suffer from a lack of interpretability. Consequently, when applying SR models to critical applications, such as medical imaging [46] or evidence for criminal prosecution [47], we must be wary of their flaws. In essence, we must exercise caution when treating or presenting super-resolved images as anything more than a hallucination.

5.4 Further work

Whilst undertaking this project, I identified multiple areas for further exploration, which were unfortunately outside of the project's scope. Two of these, which appeared particularly promising, are presented below:

- **Temporal stability** — CNNs are well known to exhibit strong sensitivity to their inputs, for which small changes may result in dramatically different outputs. In the context of VSR, this can lead to perceptual differences when an input image is minimally transformed, as occurs between video frames. This lack of stability between temporally-adjacent frames results in flickering artefacts, which decrease the perceptual quality of the output. Regularisation techniques have been proposed for this [48], although only minimally in the context of SR [10]. Utilising this regularisation in the cost function of recurrent VSR models may therefore improve the perceptual performance of these models.
- **Inference package improvements** — While the inference package designed and released as part of this project provides a simple interface for the upsampling of videos, further work could be done to increase the scope and performance of the package. Execution on video streams, rather than fixed-length videos, is a clear next step for the package, due to the wide range of use cases this would introduce. If this is to be completed with high performance, multi-threading should be implemented such that the upsampling of videos occurs concurrently to the remainder of the processing pipeline.

Bibliography

[1] A. Cuthbertson. (2020, March) Coronavirus: Netflix to lower streaming quality to stop internet from collapsing. Accessed on: Apr. 26, 2020. [Online]. Available: https://www.independent.co.uk/life-style/gadgets-and-tech/news/coronavirus-netflix-streaming-quality-internet-down-a9412771.html

[2] P. Kafka. (2018, March) You can watch netflix on any screen you want, but you're probably watching it on a tv. Accessed on: Apr. 26, 2020. [Online]. Available: https://www.vox.com/2018/3/7/17094610/netflix-70-percent-tv-viewing-statistics

[3] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," in *Computer Vision – ECCV 2016*. Cham: Springer International Publishing, 2016, pp. 391–407.

[4] N. Ahn, B. Kang, and K.-A. Sohn, "Fast, accurate, and lightweight super-resolution with cascading residual network," in *Computer Vision – ECCV 2018*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Cham: Springer International Publishing, 2018, pp. 256–272.

[5] B. Li, J. Liu, B. Wang, Z. Qi, and Y. Shi, "s-lwsr: Super lightweight super-resolution network," 2019.

[6] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 1874–1883.

[7] P. S. Sane and A. B. Gavade, "Super resolution image reconstruction by using bicubic interpolation," in *National Conference on Advanced Technologies in Electrical and Electronic Systems*, Belgaum, India, Oct. 2014, pp. 290–294.

[8] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 2, pp. 295–307, 2016.

[9] B. Lim, S. Son, H. Kim, S. Nah, and K. M. Lee, "Enhanced deep residual networks for single image super-resolution," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017, pp. 1132–1140.

[10] M. S. M. Sajjadi, R. Vemulapalli, and M. Brown, "Frame-Recurrent Video Super-Resolution," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[11] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural networks*, vol. 6, no. 6, pp. 861–867, 1993.

[12] B. Hanin and M. Sellke, "Approximating continuous functions by relu nets of minimal width," *arXiv preprint arXiv:1710.11278*, 2017.

[13] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. Madison, WI, USA: Omnipress, 2010, p. 807–814.

[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[15] M. D. Zeiler, D. Krishnan, G. W. Taylor, and R. Fergus, "Deconvolutional networks," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010, pp. 2528–2535.

[16] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[17] Tensorflow core v2.1.0 documentation - depth_to_space. Accessed on: Mar. 30, 2020. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/nn/depth_to_space

[18] W. Shi, J. Caballero, L. Theis, F. Huszar, A. Aitken, C. Ledig, and Z. Wang, "Is the deconvolution layer the same as a convolutional layer?" *arXiv preprint arXiv:1609.07009*, 2016.

[19] Z. Wang, J. Chen, and S. C. Hoi, "Deep learning for image super-resolution: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[21] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1, 2013, p. 3.

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[23] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[24] J. Caballero, C. Ledig, A. Aitken, A. Acosta, J. Totz, Z. Wang, and W. Shi, "Real-time video super-resolution with spatio-temporal networks and motion compensation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 4778–4787.

[25] A. Kappeler, S. Yoo, Q. Dai, and A. K. Katsaggelos, "Video super-resolution with convolutional neural networks," *IEEE Transactions on Computational Imaging*, vol. 2, no. 2, pp. 109–122, 2016.

[26] B. Lim and K. M. Lee, "Deep recurrent resnet for video super-resolution," in *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE, 2017, pp. 1452–1455.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[28] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[29] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[30] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[31] S. Nah, S. Baik, S. Hong, G. Moon, S. Son, R. Timofte, and K. M. Lee, "Ntire 2019 challenge on video deblurring and super-resolution: Dataset and study," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[32] E. Agustsson and R. Timofte, "Ntire 2017 challenge on single image super-resolution: Dataset and study," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[33] R. Timofte, E. Agustsson, L. Van Gool, M.-H. Yang, L. Zhang, B. Lim *et al.*, "Ntire 2017 challenge on single image super-resolution: Methods and results," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.

[34] C. Liu and D. Sun, "A bayesian approach to adaptive video super resolution," in *CVPR 2011*, 2011, pp. 209–216.

[35] A. Shapiro and Y. Wardi, "Convergence analysis of gradient descent stochastic algorithms," *Journal of optimization theory and applications*, vol. 91, no. 2, pp. 439–454, 1996.

[36] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[37] A. Giachetti and N. Asuni, "Fast artifacts-free image interpolation," in *Proceedings of the British Machine Vision Conference*. BMVA Press, 2008, pp. 13.1–13.10, doi:10.5244/C.22.13.

[38] I. Levkivskyi, J. Lehtosalo, and L. Langa, "Protocols: Structural subtyping (static duck typing)," PEP 544, 2017. [Online]. Available: https://www.python.org/dev/peps/pep-0544/

[39] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[40] S. Xingjian, Z. Chen, H. Wang, D.-Y. Yeung, W.-K. Wong, and W.-c. Woo, "Convolutional lstm network: A machine learning approach for precipitation nowcasting," in *Advances in neural information processing systems*, 2015, pp. 802–810.

[41] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent."

[42] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[43] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[44] P. Read and M. Meyer, *Restoration of Motion Picture Film*, ser. Butterworth-Heinemann Series in Conservation and Museology. Elsevier Science, 2000.

[45] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[46] J. Zhu, G. Yang, and P. Lio, "How can we make gan perform better in single medical image super-resolution? a lesion focused multi-scale approach," in *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*. IEEE, 2019, pp. 1669–1673.

[47] Y. Yamada and D. Sasagawa, "Super-resolution processing of the partial pictorial image of the single pictorial image which eliminated artificiality," in *2012 IEEE International Carnahan Conference on Security Technology (ICCST)*, 2012, pp. 338–344.

[48] G. Eilertsen, R. K. Mantiuk, and J. Unger, "Single-frame regularization for temporally stable cnns," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 176–11 185.

Appendix A

Project Proposal

Computer Science Tripos Part II Project Proposal

Accelerated video super-resolution

2538G
Originator: 2538G
23/10/2019

Project Supervisors: Prof Pietro Lio & Jin Zhu
Director of Studies: Dr Graham Titmus
Project overseers: Prof Glynn Winskel & Dr Richard Mortier
Third-party resource provider: Peter Horsman

Introduction

Super-resolution (SR) refers to a class of techniques which aim to enhance images by both increasing spatial resolution and removing distortions. Increasing image resolution 'correctly' is a fundamentally ill-posed problem due to there being multiple valid high-resolution source images for any low-resolution image, hence solving the problem from the low-resolution image alone leads to information gain. However, recent advances from deep neural network based approaches have greatly improved both accuracy and execution speed when performing this task, allowing for many practical applications of SR technology to emerge.

Most of the literature on SR has been focused on single-image methods, which aim to enhance a single frame rather than multiple consecutive frames, as found in video. Video-based SR demands increased computational performance due to the high number of frames, while providing the opportunity for increased accuracy through temporal information from adjacent frames.

In this project, I will research, train and evaluate multiple SR models, with a focus on maximising execution speed while maintaining sufficient accuracy. This will allow the implementation of an application capable of performing SR on a given input video. As an extension, I may experiment with methods for accelerating this application, such as localised SR and training video-based, rather than single-image, models. If the application can be accelerated to a point where the execution speed is sufficient for live video enhancement, it can be extended to execute in real-time on webcam or streamed video inputs.

This project is being undertaken in partnership with Arm, due to their recent efforts to accelerate machine learning workloads on low-power platforms. Achieving acceptable SR performance on low-power systems presents an additional challenge and represents the resources available in many of its potential use cases. By executing the application on a HiKey 960 development board loaned from Arm, I hope to create an environment which mimics the restrictions faced in real-world use cases.

Starting Point

I have no prior experience with SR specifically, however I have completed courses in Machine Learning and Real-world Data as well as Artificial Intelligence. Outside of university, I have completed deep learning projects, in addition to undertaking an internship with Arm where I contributed to ArmNN, the neural network inference engine I intend to use in this project.

Success Criteria

The core success criterion for this project is the completion of a SR application capable of enhancing video with acceptable accuracy (as defined in the Evaluation section). This will involve the following stages:

1. **Data collection** — Obtain an image and/or video dataset to be used for training the SR models. This will likely be a pre-existing dataset as there already exist many for single-image and video SR, however it may be manually gathered if required.
2. **Model training** — Perform research into video and single-image SR, then train multiple SR models, aiming to reproduce the results achieved in their respective papers.
3. **Model evaluation** — Compare the performance of the trained models, selecting which will be used in the application.
4. **Application implementation** — Implement an application to apply the selected SR model to an input video.
5. **Application acceleration (extension)** — Experiment with acceleration techniques to decrease the overall video processing time while maintaining acceptable accuracy. This is discussed further in the Possible Extensions section.

Evaluation

In addition to judging whether the success criteria have been met, evaluation will be performed by quantitatively analysing the performance of the application on a test video dataset. Performance will be based on both accuracy and execution speed — accuracy being measured by peak signal-to-noise ratio (PSNR) and Structural Similarity Index (SSIM).

Bicubic interpolation, an extension of bilinear interpolation with superior accuracy and low computational cost, will be used as a lower bound for accuracy. For a model's accuracy on a dataset to be deemed 'acceptable', it must surpass the performance of bicubic interpolation in both PSNR and SSIM. In addition to this, the results gathered from a survey of papers will be used to benchmark model performance. For each trained model, I will also aim to recreate the results achieved by the model in its respective paper.

Possible Extensions

While accelerating video SR is the core goal of the project, the extent to and the methods by which it is accelerated form the possible extensions for the project.

As mentioned in the introduction, the application of SR to video rather than single-image data introduces opportunities for optimisation. Namely, the frames adjacent to the frame being targeted in a video provide temporal information which can be used to accelerate or increase the accuracy of predictions. Recurrent neural networks can make use of this information since they hold internal state from previous inputs, allowing video-based SR models to be developed.

In addition to this, computer vision techniques may be used in conjunction with the trained model to improve performance. One possibility is performing localised SR, in which the SR model is only executed on subsections of the frame as determined by some heuristic, such pixel variance or an object detector.

Depending on the performance achieved by the application, new extensions may become possible. If acceptable accuracy (as defined in the Evaluation section) can be achieved with high execution speed, a real-time application may be able to be developed. This could apply SR to, for example, live webcam or streamed internet video.

Timetable and Milestones

I plan to start work on 24/10/19.

- **Michaelmas weeks 3–4** (24/10/19 – 06/11/19)

  Research SR models and obtain the image/video dataset for model training.

- **Michaelmas weeks 5–6** (07/11/19 – 20/11/19)

  Set up the HiKey development board with the Arm inference engine, ArmNN. Implement selected models in TensorFlow with dummy weights and test on the engine to verify support.

- **Michaelmas weeks 7–8** (21/11/19 – 04/11/19)

  Begin training of supported models on the obtained dataset.

- **Michaelmas vacation** (05/12/19 – 15/01/20)

  Finish model training and complete model evaluation. Implement the application using the best performing model. Begin extension work (application acceleration) if time allows.

- **Lent weeks 1–2** (16/01/20 – 29/01/20)

  Write up the progress report and presentation, continuing any extension work.

- **Lent weeks 3–4** (30/01/20 – 12/02/20)

  Progress report submission deadline: 31/01/20

  Finish extension work and/or any core criteria not yet completed. Begin work on the draft dissertation, writing the Introduction and Preparation chapters.

- **Lent weeks 5–6** (13/02/20 – 26/02/20)

  Continue work on the draft dissertation, writing the Implementation chapter.

- **Lent weeks 7–8** (27/02/20 – 11/03/20)

  Finish work on the draft dissertation, writing the Evaluation and Conclusion chapters. Submit draft to supervisors for comments.

- **Lent vacation & Easter weeks 1–2** (12/03/20 – 06/05/20)

  Make amendments to dissertation based on supervisor comments and submit the formal dissertation.

  Dissertation submission deadline: 08/05/20

Resource Declaration

I plan to use my personal laptop (2016 Macbook Pro, 2GHz Intel Core i5 processor with 8GB LPDDR3 RAM and an Intel Iris Graphics 540 GPU running MacOS) for development and writing the dissertation. I have automatic backups to a cloud provider, and will periodically save all work to an external disk. An Ubuntu dual boot is installed on the machine, should it be needed for development. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. If this machine fails, I will switch to using the MCS machines.

In addition to this, I will be using a GPU from Google CoLab and/or the Computational Biology research group to perform model training. I am also signing a Board Loan Agreement for a HiKey 960 development board with Arm, which will extend from the point of signing until the end of June 2020. This will be solely for executing the application. If the board malfunctions, a new board will be provided by Arm. In the case of no replacement being provided or the replacement taking too long, I will revert to executing the application on my personal laptop or an MCS machine. The point of contact for this loan is Peter Horsman, who can be reached at Peter.Horsman@arm.com.