EFFICIENT CONVOLUTIONAL NEURAL NETWORKS IN IMAGE PROCESSING APPLICATIONS

Nicholas Chiapputo

Thesis Prepared for the Degree of

Masters of Science

UNIVERSITY OF NORTH TEXAS

August 2022

APPROVED:

Colleen P Bailey, Major Professor
Shengli Fu, Chair of the Department
of Electrical Engineering
Parthasartathy Guturu, Committee Member
Kamesh Namuduri, Committee Member
Hanchen Huang, Dean
of the College of Engineering
Victor Prybutok, Dean
of the Toulouse Graduate School

ACKNOWLEDGEMENTS

Thank you to my family – Mom, Doug, and Abby – for all of your support and guidance. Thank you to the members of the OSCAR Lab who have pushed me to be a better researcher and giving me the confidence and strength to persevere. The guidance and instruction of the members of my committee – Prof. Colleen Bailey, Dr. Kamesh Namuduri, and Dr. Parthasarathy Guturu – has been instrumental in improving myself both personally and academically. Thank you to the members and officers of the IEEE UNT Student Branch and the HKN Lambda Zeta chapter. Finally, thank you to my amazing wife, Rachel, for the sacrifices you have made and the unending support you have given to me over the years we've been together.

Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

Machine learning has shaken up the world of optimization and provided new and very successful methods of solving a plethora of problems from speech recognition to recommendation engines to computer vision. By simplifying these problems to statistical models and simple equations, machine learning models have outperformed humans and improved upon the previously hand-designed methods of optimization. The emergence of big data recently has accelerated this progress even further by providing models with more data than a human could go through in a decade. However, with this increase in accuracy and data size comes large computational costs. While hardware is accelerating at a rapid pace, so too are the memory and computational costs of neural networks. To meet this problem, a growing number of researchers look for optimized solutions and models that focus on keeping the memory footprint to a minimum and reducing the number of operations while still maintaining good prediction accuracy. This work is under the umbrella of Tiny Machine Learning (TinyML).

This thesis explores how convolutional neural networks can be accelerated while maintaining or improving performance. Chapter 2 dives through the background of machine learning to form a basis for discussions in the rest of the thesis. Chapter 3 explores the single-image super-resolution problem with a literature review and a proposal for the efficient super-resolution model TinyPSSR. Chapter 4 then discusses the results of TinyPSSR. Chapter 5 proposes a deep image compression model with performance comparisons to JPEG. This thesis is concluded in Chapter 6.

CHAPTER 2

BACKGROUND

Historically, optimization problems are solved through extremely tedious work often involving designing and solving (or approximating) complex mathematical models. In the absence of the ability to mathematically model a problem, brute force is commonly used by repeatedly making small adjustments to inputs in order to push towards a desired output. Final selected solutions are then hard-coded and remain constant until a newer, improved standard solution is accepted. Examples of this can be seen in image compression standards such as JPEG, JPEG2000, and PNG.

Rather than design solutions from the exceedingly tedious brute force methods, we seek to automate the discovery of optimal solutions by providing a computer with a set of representational data and an abstract model (or equation), and allowing the computer to repeatedly test various combinations of parameters in the model to find an optimal solution. This process is known as machine learning. While the idea of machine learning appears to be relatively recent, it is typically agreed that the modern definition of machine learning began with work in the 1940s and has been known by various names over the years (e.g., "cybernetics" and "connectionism").

This chapter introduces the background information that forms the basis for the research works presented in this thesis. Basic neural networks are introduced, followed by the techniques to train the networks, and finally methods of reducing the computational complexity while maintaining high accuracy are discussed.

2.1. Artificial Neural Networks

The earliest machine learning models took direct inspiration from biological functions. Figure 2.1 shows the model of an artificial neuron, believed to be the first neural network designed by Warren McCulloch and Walter Pitts in 1943 [1]. The model attempts to model a single neuron from the brain with input signals (dendrites), a summing function Σ (soma), an activation function $g(\cdot)$ (axon), and output signal(s) (synapses).

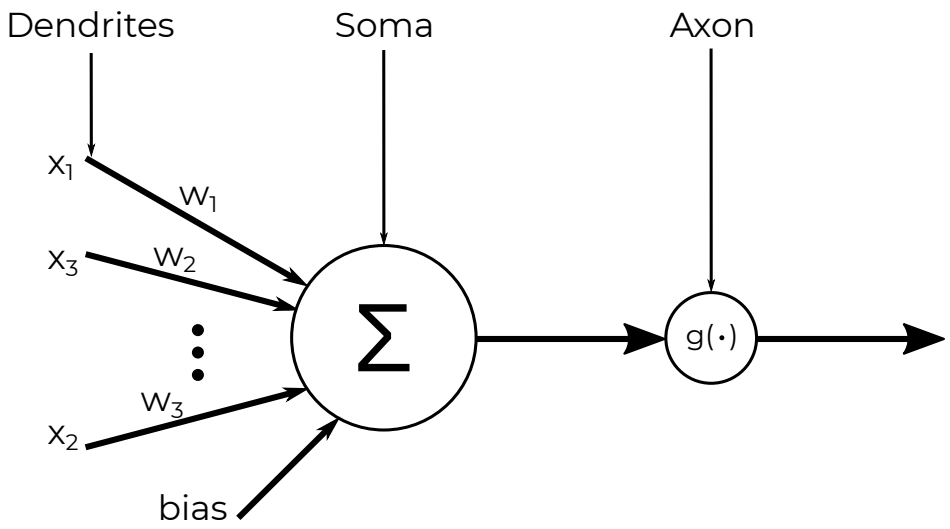


FIGURE 2.1. Basic model of an artificial neuron.

Dendrites, branches connected to the soma, are the information receivers of a neuron. After receiving an input signal, the dendrites perform a multiplication operation that applies a weight w_n to the input signal x_n and passes the weighted signal into the soma. A bias value b is also added in alongside the weighted inputs and can allow for fine-tuning of a node's output. These weights and biases are the learned parameters that provide the intelligence of a neuron and are adjusted during the training process.

The soma, the cell body of a neuron, aggregates the weighted signals from the dendrites. These signals can be positive or negative or, in a biological sense, exciting or inhibiting. The mathematical representation of the soma is a simple summation function that combines each of the weighted input signals into a single value. This operation effectively determines how important each input signal is to the overall system. Low-weighted signals will be overwhelmed by higher weighted signals.

The summed weighted input signals are passed from the soma to the axon. In a biological neuron, the axon has a basic threshold that can be activated when the signal from the soma reaches a certain electrical potential threshold. That is, if the summed and weighted inputs are large enough, the axon will "turn on," or activate, and transmit a signal through the axon to the synapses at the end. From a mathematical point of view, we can represent the axon using an "activation function" $g(\cdot)$ that is designed to fire only when the

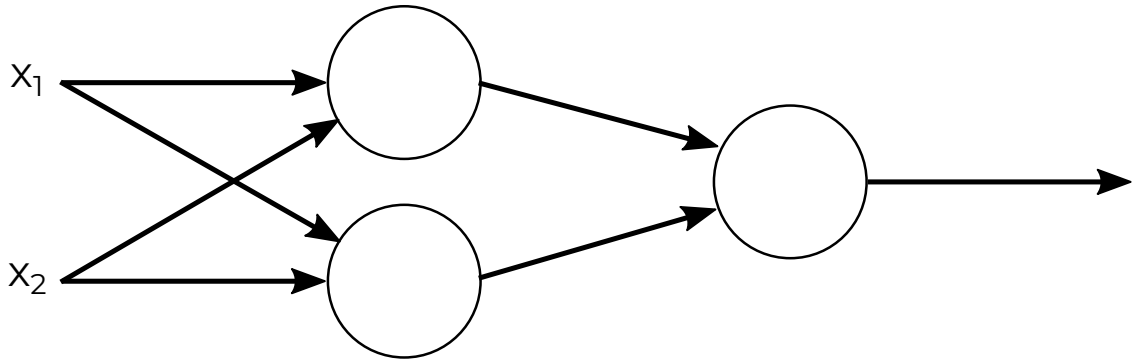


FIGURE 2.2. A two-layer artificial neural network.

input reaches a certain threshold. This operation can be modeled by several functions such as step, sign, or sigmoid (detailed further in section 2.4). Finally, the output reaches the synapses at the end of the neuron. These synapses are the connections from one neuron to the next and are what allow more complicated decisions to be made or actions to be taken.

Artificial neurons are the building blocks of larger models, called artificial neural networks (ANNs). Groups of these neurons, or nodes, who share the same input values and calculate output values in parallel are referred to as layers. ANNs can be designed by adding more neurons (nodes) in the same layer and/or adding more layers where subsequent layers take their input from the output of the previous layers. An example ANN with two layers having two and one nodes, respectively, is shown in Figure 2.2. Artificial neurons that output only binary values (high or low, 1 or 0) are known as perceptrons. An ANN designed exclusively using these perceptrons and organized with at least three layers are called multilayer perceptrons (MLPs).

2.2. Deep Learning

With Moore's Law in effect, computational power has exploded over the past few decades, providing resources for researchers to design more complex models. These complex models are built on the basic ideas from ANNs and form the basis of Deep Learning (DL). DL is a sub-field of machine learning where models use multiple layers within the network. Layers between the first and last (input and output layers, respectively), are referred to as "hidden" layers. This work deviates solidly away from the original biological designs

of artificial neurons and focuses on layers from the very simple, such as dense (ANNs) or convolutional, to the more complex, such as transformers, long short-term memory, and residual layers.

While artificial neurons were good at approximating simple mathematical equations, logical functions, or actions that have a strict set of rules or possible moves (e.g., AND/OR gates, part-of-speech tagging, or even playing chess), DL models tend to focus on very complex problems that are typically harder for computers. Such problems include computer vision or object classification that are easy for humans, but much more difficult for computers as there is a large amount of data that plays into the final outcome. These problems generally require that data be represented in different forms or features extracted from the raw data.

2.3. Layers

With the rise of DL, more complex layers have evolved. Originally, artificial neurons were arranged into dense layers. These dense layers have a single weight for each input value and can be connected in multiple topologies including fully connected where each neuron in layer l has an input dendrite to each output in layer $l-1$ and an output synapse to each input in layer $l+1$. While dense layers are good at simple weighting, there is a lot of contextual information that is lost.

For spatial context, convolutional layers are commonly used [2, 3, 4]. These layers are very good in image processing applications, as outputs can be weighted based on context around a single pixel. Naturally, they perform the same operation as classical image processing convolution (e.g., the Sobel operator for edge detection) and are effectively just a learned implementation rather than using pre-determined values. Convolution, both learned and non-learned, is used primarily as a feature detector for the input data. Neural networks whose learned layers (those with trainable weights) that are all convolutional layers are referred to as Convolutional Neural Networks (CNNs).

In convolutional layers, a kernel ω (a matrix, also referred to as a filter) is convolved (element-wise multiplication) with feature maps from the previous layer $l-1$ (or input in the case that $l = 1$). A single convolutional layer can have multiple kernels, creating multiple

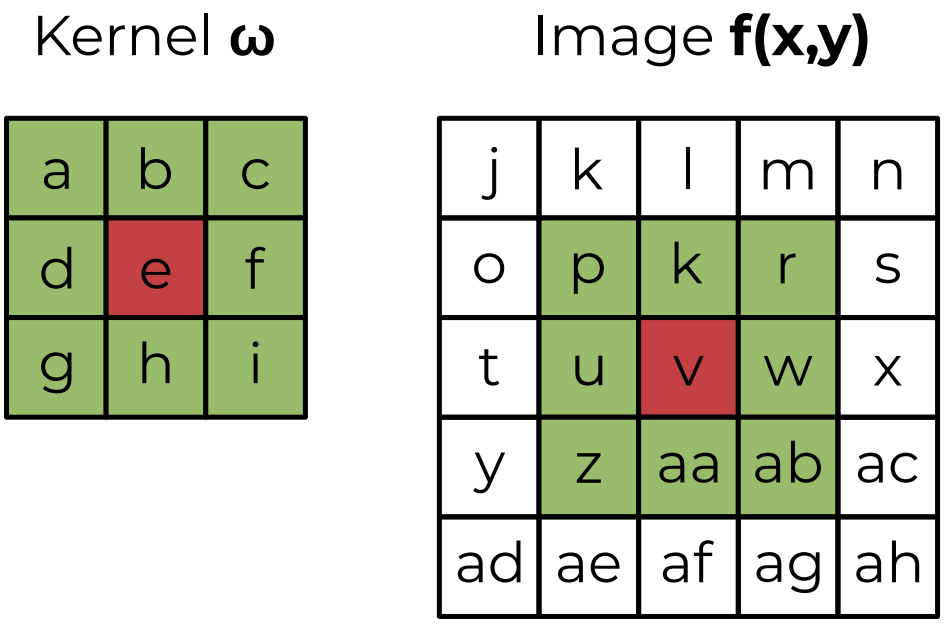


FIGURE 2.3. Convolutional kernel and image input.

output feature maps. When defining these layers, the syntax (l, a) can be used where l is the number of output filters and a is one of the dimensions of the kernels. Typically, kernels are chosen such that they are odd-sized squares (axb where $a = b$ and a is odd). The odd size allows for the kernel to center on a single pixel and the square shape gives equal importance to the neighboring pixels vertically and horizontally of the center. The equation for convolution of a kernel w of size axb and an input image $f(x, y)$ centered around the pixel at position (x, y) is

$$g(x,y) = \omega \circledast f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) \cdot f(x+dx, y+dy). \tag{1}$$

Figure 2.3 shows an example kernel of size 3x3 and an input image of 5x5. The red coloring shows the pixel where the convolution is centered when calculating $g(2, 2)$ and the green shows surrounding pixels whose values take part in the final calculation. The full

equation for $g(2,2)$ is given as

$$\begin{aligned} g(2,2) = (a \cdot p) + (b \cdot k) + (c \cdot r) + \\ (d \cdot u) + (e \cdot v) + (f \cdot w) + \\ (g \cdot z) + (h \cdot aa) + (i \cdot ab). \end{aligned} \tag{2}$$

From this example, it can be seen that there may be problems when we go near the edges of an input image. This is known as a boundary problem and can cause ringing effects (harsh aliasing) around the edges of the output feature maps. Typically, the size of the output feature map will be reduced by one less than the kernel size. That is, an input image of 8x8 with a kernel size of 3x3 will be reduced to a $(8-3-1)$x$(8-3-1) = 6$x6 feature map as the convolution will be begin by aligning the top and left edges of the kernel and input image. This can lead to other boundary problems such as reduced feature strength near the edge of an image. One solution to this problem is by padding the input image.

By default, no padding is added to the input image and the feature map is reduced in size. To retain the original input size, padding can be added around the edge of the input. The value of the added pixels can either be zero (more common), or the values of the pixels at the original border can be extended. By padding, each pixel is given the opportunity to be at the center of the kernel, allowing its features to be detected more strongly.

While adding padding can retain the input image size, the output size can be reduced by adjust the stride of the kernel. The stride determines how far the kernel moves along the input image, both vertically and horizontally, after each calculation. The example given by Equation 2 and Figure 2.3 uses a stride of 1 in both directions. A stride of 2 in either direction reduces the output size in that dimension to half. This technique is commonly used to reduce the complexity of a model by compressing feature maps and can also be helpful in reducing the number of parameters used if a dense layer is used later in the model. Combining this technique with padding and the kernel size, we can formulate the output

feature map size using the following equations:

$$r = \left\lfloor \frac{(y - a + 2p)}{s_y} \right\rfloor + 1$$
$$c = \left\lfloor \frac{(x - b + 2p)}{s_x} \right\rfloor + 1 \tag{3}$$

where r and c are the output height and width, y and x are the input height and width, a and b are the kernel height and width, s_y and s_x are the vertical and horizontal stride, p is the amount of padding added around the image, and $\lfloor \cdot \rfloor$ is floor division (removing fractional parts of the resulting division).

2.4. Activation Functions

As in the artificial neuron, deep learning layers also use an activation function $g(\cdot)$ after the main operation of the layer [5]. The purpose is to introduce non-linearities into the model. Without these, neural networks could only ever approximate linear functions ($y = Wx + b$) and hidden layers would not be useful in approximating higher order functions. By introducing non-linearities, neural networks can approximate more complex actions and decisions. This is useful for problems where classes of objects or actions cannot be linearly separated and is significantly useful in regression models where the output is not just a class of an object or a integer number, but is instead a real-valued output (e.g., 24-bit pixel values in an image).

Traditional activation functions include binary step and sigmoid. The former outputs a fully high value (typically 1) if the input is non-negative and a low value (typically 0 or -1) if the input is negative, while the latter outputs the function $\frac{1}{1+e^{-x}}$ where x is the input from the layer's output. Figure 2.4 shows the responses of these two activations. These functions are useful for linear or binary classification problems, however, they run into what is known as the vanishing gradient problem.

At each weight update instance during training, after a pre-determined number of inputs have been tested (i.e., a single batch), the trainable weights of a model are updated. The weight update process is known as backpropagation. To update the weights, the loss over

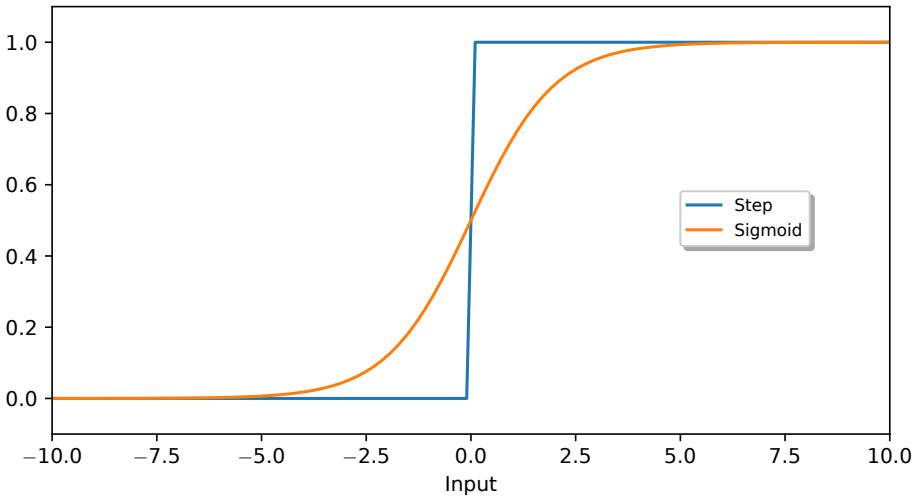


FIGURE 2.4. Step and sigmoid activation function responses.

the current batch is summed together to calculate the total error. To update an individual weight w_j, its contribution to the total error is found by calculating the partial gradient of the loss $\mathcal{L}$ with respect to w_j.

The vanishing gradient problem, typically a result of a poor activation function choice, occurs during this process of backpropagation and weight updates, resulting in the partial gradients for successively deeper layers within the network becoming so small as to barely have any effect. This is primarily because the derivative of the activation functions approaches zero as the input of the activation function x tends towards $\pm\infty$ (i.e., the partial gradients saturate in both directions).

Since many deep learning models obtain their better results through deeper layers, the most popular solution to the vanishing gradient problem is changing the activation function. Rectifiers, functions who fully pass through non-negative inputs and rectify, or turn off, negative inputs, are some of the most popular activations in modern neural networks. The main functions in the recitfier class are the Recitifed Linear Unit (ReLU) and Leaky ReLU, shown graphically in Figure 2.5. Additionally, there is the parametric ReLU (PReLU)

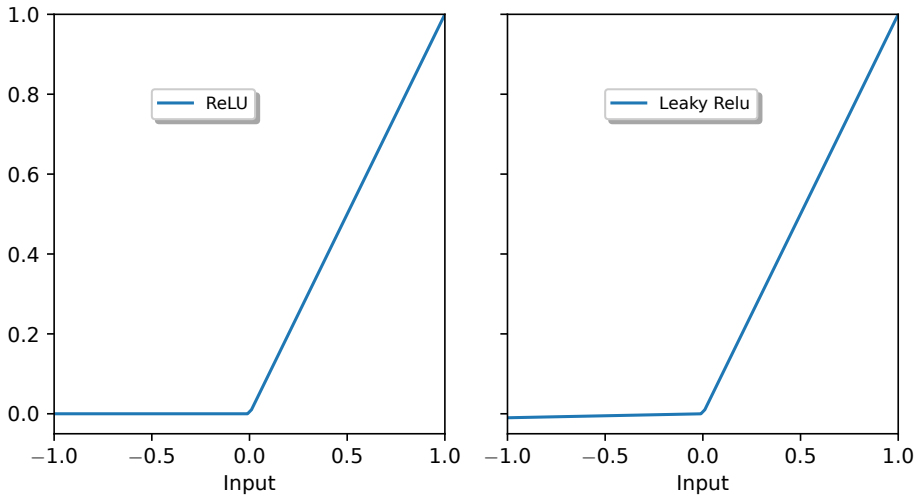


FIGURE 2.5. Rectified Linear Unit (ReLU), and Leaky ReLU ($\alpha = 0.1$) activation functions.

activation function. These three activations are defined as

$$\text{ReLU: } f(x) = \begin{cases} x, & x > 0 \\ 0, & x < 0 \end{cases} \tag{4}$$

$$\text{Leaky ReLU: } f(x) = \begin{cases} x, & x > 0 \\ 0.01x, & x < 0 \end{cases} \tag{5}$$

$$\text{PReLU: } f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x < 0. \end{cases} \tag{6}$$

The leaky and parametric ReLU functions arose from the problem of dead neurons caused by the complete rectification of negative inputs. This would result in models getting "stuck" in local minima during the weight update process of training and unable to optimize the results since the neurons would never activate. This is especially noticeable in CNN. Instead, leaky and parametric ReLU activations apply a small slope to pass through negative inputs at a lower weight. While the definition of Leaky ReLU shows a constant 0.01 slope for $x < 0$, most machine learning implementations allow this value to be set as a hyperparameter (that is, a constant set prior to training). PReLU is a significant improvement over this as

the slope α for $x < 0$ is a learned parameter that is updated with the weights and biases during training. This has been shown to have measureable improvements in accuracy while coming at a very small parameter and computational cost as there is only one parameter per layer, or filter in the case of convolutional layers.

A more recent development is the Growing Cosine Unit (GCU) [6] activation function defined as

$$g(x) = x \cdot cos(x). \tag{7}$$

In biological neurons, some neurons have been found that have oscillatory activation functions. The GCU function replicates this and has the advantage that the gradient is not bounded in either direction unlike ReLU. There is not a significant amount of work using this activation as it has been introduced very recently, but preliminary work shows moderate metric improvements in classification and some computational efficiency over more complex activation functions, though slightly higher cost (depending on processor architecture implementation) than the more simple rectification ReLU class of functions.

2.5. Training Data

The familiar phrase "practice makes perfect" rings just as true in machine learning. In order to train a neural network, it must repeatedly practice on training data and update its weights based on the difference between the outputs and the ground truth values (the desired outputs). The selection of data for this training is one of the most important steps in machine learning. It is important to select data that is representative of the population so that the model is generalizable and can be put into practice with little to no loss in accuracy when faced with real world data. Since it is typically cost-prohibitive to test a model in the real world at every iteration, data is typically split into three groups: training, validation, and testing.

The ratios of the three splits' size is set so that the validation and testing sets are large enough to still contain a representational dataset to generalize to the population data while keeping the training set as large as possible to give the model enough data to learn

from so as to be able to generalize in practice. Typically, the validation and testing sets are 5-15% each of the total dataset size with training using the remaining data. It is important that the training set remain as large as possible since the loss and metric results from the validation and testing sets are not used when updating the model weights. These results are used to determine when the model has converged and for the user to track the progress of the model training. The approximate ratios for data splits are good practice when only one dataset is available. However, in many problems there are standard testing sets that are used instead of arbitrary partitions. This allows for a greater amount of data in the training set.

2.6. Quality Metrics

To determine how well the model estimates the output, error metrics are defined to calculate the loss between the ground truth and the estimated outputs. When training, the model weights are updated in order to minimize the loss function $\mathcal{L}$. The loss for each iteration of training is calculated using

$$\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(o_i,\hat{o}_i) \tag{8}$$

where n is the number of data points in one training instance (also known as the batch size), o_i is the ground truth for input i, and $\hat{o}_i$ is the respective estimated output from the model for the same input. In regression problems where the estimated output is real-valued, the two most common loss functions are mean absolute error (MAE, L_1) and mean squared error (MSE, L_2). These loss functions are defined as:

$$\mathrm{MAE}(o,\hat{o}) = \frac{1}{k}\sum_{i=1}^{k}|o_i-\hat{o}_i| \tag{9}$$

$$\mathrm{MSE}(o,\hat{o}) = \frac{1}{k}\sum_{i=1}^{k}|o_i-\hat{o}_i|^2. \tag{10}$$

Both equations are calculated as the arithmetic mean of the absolute or squared difference between each element of the output. For example, if the estimated output is an

N-dimensional array, the error is the absolute or squared element-wise difference between the estimated and the ground truth output. MSE is more sensitive to outliers due to the squared difference while MAE tends to ignore outliers as they are weighted the same as all other points.

While MSE and MAE are good for measuring element-wise loss, no spatial information is retained. To improve on this, Wang et al, designed the Structural Similarity Index Measure (SSIM) to measure perceptual image quality by calculating structural loss in an image [7]. SSIM is calculated using the equation

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{11}$$

where μ is the average, σ_{xy} is the covariance, and σ^2 is the variance of the pixels in the two areas denoted by x and y. C_1 and C_2 are stabilization factors defined as $(k_1 L)^2$ and $(k_2 L)^2$, respectively, where L is the dynamic range of the pixel values, $k_1 = 0.01$, and $k_2 = 0.03$.

SSIM values are computed on a window of an image since perceptual information is more important locally as human eyes tend to focus on small regions at a time instead of perceiving the quality of an image as a whole. The windows x and y are typically of a small, square size around 11x11. Additionally, SSIM is defined to only operate on one channel. While traditionally the image would be grayscaled before calculating SSIM, recent work converts RGB images to the YCbCr color space and computes the SSIM over the Y (luminance) channel. This has a greater effect on the perceptual quality as human eyes are much more sensitive to luminance than other image metrics such as contrast or specific colors. For images that only have a grayscale channel and are not RGB, the single grayscale channel can be considered the Y channel.

Another standard metric used in image quality comparisons is Peak Signal-to-Noise Ratio (PSNR) and is calculated with

$$PSNR(x, y) = 20 \cdot \log_{10}(MAX_I) - 10 \cdot \log_{10}(MSE(x, y)) \tag{12}$$

where MAX_I is the maximum possible pixel value (255 for 8-bit unsigned integer image,

or 1.0 for floating-point images) and $MSE(x, y)$ is the mean squared error as defined in Equation 10. This metric focuses on pixel-wise reconstruction quality and is measured in units of dB where 30-50 dB is considered high quality for 8-bit images. While PSNR is one of the most common and popular metrics for image quality, it is not considered to accurately reflect a human's perceived image quality – images with lower PSNR can sometimes seem to be of higher quality. Therefore, it is typically used in conjunction with SSIM to measure both perceptual and mathematical quality.

2.7. Training a Model

Once a topology is determined for a neural network, data has been aggregated, and a loss function has been selected, the model can be trained. Parameters that are not learned during training, but instead are statically assigned prior to training, are referred to as hyperparameters. In training, the important hyperparameters are epochs and batch size. The number of epochs is the number of times the training iterates through the entire training set. During each epoch, each data point is tested on the model exactly one time. At the end of each epoch, the model is tested on the validation and test sets (whose loss and metric results are not considered in the weight update process). The batch size is the number of input data points passed through the model for each weight update. After each batch, the summed loss is set back to zero.

2.8. Tiny Machine Learning and Techniques

With the constant improvement in computing hardware, easy-to-use libraries for machine learning, and the move towards GPU-accelerated learning, models have exploded in size. State-of-the-art-models are typically in the tens or even hundreds of millions of parameters. With this massive model size comes significant performance penalties. Typically, in one forward pass through the network (i.e., generation of one estimated output), a model requires at least one multiplication operation per parameter. This then requires significant computational resources as one prediction can take on the order of minutes to complete. Additionally, large layers have even larger memory storage requirements to store the inputs,

the intermediate calculations, and the outputs. For example, convolutional layers need to store an uncompressed image for each filter. For a layer with 256 filters (a very common value), a 1920x1080 input image would require 530,841,600 32-bit floating point numbers – or 2.1 GB – just to store the output. Additionally, a full implementation also must store all the parameters in a model, the input to the layer, and the intermediate calculations (which are all 32-bit floating point numbers). From this, it can be easily seen that practical implementation of many machine learning models is not possible due to both computational and memory requirements.

With the move towards more expensive models, there has been growing interest in just the opposite – tiny models, or tiny machine learning (TinyML) [8, 9]. This approach to machine learning focuses on creating efficient models with a small number of parameters while still meeting the performance of somewhat larger models (or making appropriate trade-offs). This is particularly useful for Internet of Things (IoT) or embedded applications where memory capabilities are in the hundreds of kilobytes or tens of megabytes instead of tens of gigabytes as on desktop computers or dedicated machine learning hardware. This allows for models to run efficiently on mobile and embedded devices and be practically implemented for end-user use without rapidly draining battery or suffering in responsiveness.

There are a number of techniques for improving the efficiency of models. One of the most commonly used is pruning whereby certain nodes are turned off (either removed, or weights are set to zero) if their weights or contribution to the result is below a certain threshold. By removing nodes of less importance, little accuracy is lost while improving overall performance. After pruning, extra training can be performed to fine tune the model.

Quantization is the process of mapping one set of values to another. This is commonly done to convert 32-bit floating point calculations to either 16-bit floating point or 8-bit integer (signed or unsigned). Operations with lower bit counts are more efficient in both computation and memory, though some accuracy can be lost. For embedded applications on devices that do not have floating point units (FPUs), quantization to integer representations is required.

While many of these techniques focus on actions taken post-training, there can be significant improvements made during the model design phase. In fully connected dense layers (e.g., ANNs), a single weight is needed for each pair of input and output nodes. This is extremely computationally expensive. Instead, replacing these dense layers with convolutional layers can significantly reduce the cost while improving performance. Further improvements in CNNs can be made by adding in depthwise convolutional layers. These layers use a kernel size of 1x1, effectively element-wise multiplication, and play a large role in parameter reduction. For example, a 16-filter 3x3 kernel convolutional layer followed by an 8-filter 3x3-kernel layer results in 1,320 parameters while inserting an 8-filter 1x1-kernel layer in between results in a total of 880 parameters for a 33%, or 440, parameter reduction. These depthwise convolutional layers have been increasingly used in recent models and can also improve feature detection by improving on cross-channel knowledge.

CHAPTER 3

SINGLE-IMAGE SUPER-RESOLUTION AND AN EFFICIENT IMPLEMENTATION

Single-image super-resolution (SISR) aims to reconstruct a super-resolution (SR) image from a low-resolution (LR) base. Traditional algorithms, such as bicubic or linear interpolation, are among the most used in practice due to their ease of computation. However, these methods generally produce low-quality SR images. This is due to the ill-posed nature of the SISR problem, in which there are infinitely many solutions to the upscaling of an LR image. As these traditional methods do not employ prior knowledge in reconstruction, deep learning has been increasingly applied to the SISR problem.

In this chapter, I first provide a summary of the SISR problem and a literature review of model architectures, upsampling methods, and datasets. Then, I explore the limits of TinyML and propose **Tiny P**ixel **S**huffling **S**uper-**R**esolution (TinyPSSR), a fully convolutional neural network using only 1,356 parameters. The model is designed with a focus on reducing the memory requirements of the network by reducing the number of filters at each convolutional layer, resulting in a lower total required memory allocation for implementation. After introducing TinyPSSR, I discuss its implementation, including the data used and the training setup.

3.1. Literature Review

3.1.1. Model Architectures

Current state-of-the-art models employ edge-based[10], residual[11], attention block[12], and transformer[13] techniques to tackle the SISR problem [14]. These models perform extremely well on various SISR scales, though they come at varying computational and memory costs.

Early deep learning works and more modern efficient networks make use of simple linear feed-forward networks [15, 16, 17]. These models are also the most efficient, as a single forward path reduces the computational complexity. Additionally, to avoid a vanishing gradient problem as a result of too-deep CNNs, these models tend to be smaller with fewer

parameters. One of the earliest deep learning super-resolution models, SRCNN [15], uses a very simple three-layer CNN with 57,184 parameters. FSRCNN [16] adapts the SRCNN model and proposes one of the smallest published models, FSRCNN-s, at 3,937 parameters. ESPCN [17] introduces the pixel shuffling layer (referred to as a sub-pixel convolutional layer) at around 23,000 parameters. One of the main differences in feed-forward super-resolution networks is when the upsampling occurs. Earlier works upsampled the low-resolution input at the beginning of the model [15]. However, this lead to many noticeable artifacts [18], so later works moved the upscaling to the end [16, 17].

Recently, machine learning models (in many problem spaces) have begun to employ residual networks where skip connections are used [19, 11]. These skip connections occur when a layer is connected both to the immediately following layer and a layer further down in the forward path of the network. This skip connection further down the model is usually implemented as either a concatenation of the filters, or an element-wise addition. These types of networks are very strong at learning high-frequency residue information between the LR and HR images.

Attention blocks are added to models to drive focus onto certain areas, typically those with more dense features [12, 20]. This allows a model to shift focus towards areas of more importance. These blocks have been used in many applications, including object recognition and super-resolution. The two common types of attention blocks are channel-spatial and layer. The former focuses on inter-dependencies between layers, while the latter finds important regions within a single layer or image. There are a number of different specific attention block designs. Computationally-intensive designs use matrix multiplications between the input and the transpose of the input before passing the result through a softmax filter to normalize it and a final matrix multiplication to restructure the result into the original shape. More efficient designs use a residual structure where the input is sent through a smaller convolutional network (though some also include dense layers) before being either convolved, added with, or subtracted from the original input to the attention block. The result of each of the designs is that regions of importance that typically have the more dense

features (such as those with rapid color or feature changes) are weighted higher in the output feature maps than less dense areas (such as large smooth regions).

One of the hallmarks of a high-resolution image is the sharpness of the edges. A few state-of-the-art models have designed networks that take canny edge detectors that create grayscaled images following the hard edges of an image and combine that information with the original low-resolution image [10]. Similar to attention blocks, this technique allows a model to focus on an area of importance, but instead pre-defines the important areas as those near hard edges. It has shown promising results, however, it does add extra computational complexity as the edge map must be calculated and super-resolved along with the original image.

3.1.2. Upsampling Methods

The most widely used and implemented upsampling methods are interpolation based. These methods are not learned and thus, are much faster and easier to implement. Typically, this is nearest-neighbor interpolation, bilinear, or bicubic. Nearest-neighbor has the fastest execution time as there are no calculations. Instead, a single pixel is expanded to an NxN region, where N is the super-resolution scale, and each pixel in the region is set to the same value as the original pixel. Since this method creates very blocky results, bilinear or bicubic interpolation can be used instead to create a more smooth image. These two methods perform linear and cubic interpolation, respectively, in the x and y directions. This results in a very smooth image (bicubic even moreso than bilinear), however, this smoothness tends to reduce the visual clarity of the image so that high-frequency features, such as edges, are lessened in intensity and the image is not clear.

While some models use this interpolation followed by layers mostly as noise reduction, most modern models make use of transposed convolutional layers. These layers first add padding around the input equal to one less than the size of the kernel before applying the regular convolution operation. Further, strides can be used to increase the size more rapidly than simply using padding (e.g., a stride of 2 results in a 2x upscale). This method of upsampling allows the model to more intelligently upsample than using the interpolation

since weights for the kernels can be learned.

Similar to the deconvolution, a sub-pixel convolutional layer can also be used [17, 21, 22]. This method first uses a normal convolution with padding so that the output is the same size as the input. The number of output filters is equal to the square of the upscale amount (i.e., $2^2 = 4$ filters for 2x upscale). The total number of pixels in all output filters combined is equal to the number of pixels in the desired upscaled image. Thus, the pixels can be rearranged into one filter by interlacing the pixels in the same location of each filter into squares in the upscaled filter. This method shows better efficiency than the deconvolution with stride as the convolution filters are able to focus on each pixel in the input filters whereas adding the stride to the deconvolution causes the kernel to skip over some, resulting in feature loss and less accurate reconstructions.

3.1.3. Datasets

The classical testing datasets for super-resolution are Set5 [23] and Set14 [24], which contain 5 and 14 images, respectively, of various sizes. All but two (the bridge and man images from Set14, which are grayscale) are RGB color images. These sets provide a range of topics, textures, and colors, however, it is not a very robust set. To supplement these, 100 images from the Berkeley Segmentation Dataset (BSD100) [25] and the 100 images of the Urban100 [26] dataset are commonly used. BSD100 contains many images from nature with different objects such as plants, animals, and food. Urban100 contains mostly images with sharp edges such as buildings, though there are three images with people as the main object, two of which are on railroads in nature.

For testing, larger datasets with a more varied set of features is needed. The most popular training set is DIV2K [27], which contains images who have at least one dimension of size 2040 while the other dimension is no more than 2040. The dataset contains 800 training, 100 validation, and 100 test images of high-quality and contains many different scenes. Other common high quality training sets or supplemental datasets include ImageNet [28] and Flickr2K [29].

3.2. Model Design

The design of the proposed TinyPSSR model follows the goal of reducing the memory and computational footprint. Thus, focus is placed on reducing the overall number of layers and feature maps while still maintaining a reasonable level of reconstruction quality that improves upon baseline bicubic interpolation results. Further, to allow the model to accept arbitrary size input, I follow the standard of designing a fully convolutional neural network.

Many of the state-of-the-art models in the super-resolution domain focus on using residual and attention blocks to improve on feature detection and noise reduction, which greatly improve high-frequency feature reconstruction. However, these networks result in multiple parallel paths for data to take within the model, requiring much larger concurrent memory requirements than a single-path network.

To improve memory requirements, I forego multi-path networks and focus instead only on single-path convolutional networks. This allows for much lower concurrent memory usage, requiring only the largest of two consecutive layers and intermediate calculations to be allocated. As discussed previously, making use of depthwise convolutions (i.e., 1x1 convolution kernels) is a good tool for improving the efficiency of CNNs. These layers play a large role in reducing both the memory and computational cost by reducing the number of weights and filters.

TinyPSSR is a fully convolutional network and each convolutional layer is given a small number of filters. For ease of reference, I refer to the convolutional layers in the form (n, k) where n and k represent the number of filters and the square kernel (i.e. kxk) size, respectively. Convolutional neural networks can experience the vanishing gradient problem when the number of layers grows large. Since I am avoiding the use of parallel paths to keep the total concurrent memory usage to a minimum, I forego common solutions to the vanishing gradient problem, such as residual and long skip connections. Instead, I focus on keeping the total number of convolutional layers small. Additionally, the model is designed to super resolve one channel at a time. This not only allows the TinyPSSR model to easily handle grayscale input, but also reduces the concurrent memory requirements at the potential cost

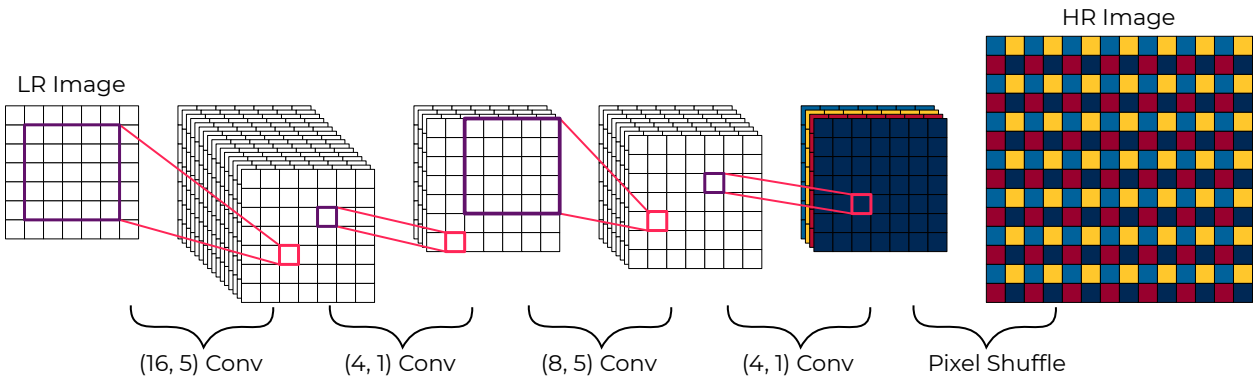


FIGURE 3.1. Model architecture of TinyPSSR.

of some execution time. The final TinyPSSR model is illustrated in Figure 3.1.

Since the proposed model takes a single-channel input, I can be more generous with the number of filters in the first layer without exploding the number of parameters, so I choose a (16, 5) first convolutional layer with 416 parameters. The 5x5 kernel allows for the filters to learn more structural information as opposed to local information of 2x2 and 3x3 kernels. Improving structural information, including high-frequency features, leads to much better perceived quality as focusing instead on local pixel-wise accuracy leads to blurrier images [30]. The first layer is then followed with a (4, 1) depthwise convolutional layer (68 parameters) to learn cross-filter features and reduce the number of parameters. This set of two convolutional layers is the largest consecutive number of filters in our model with 20 total filters. Thus, the total required memory allocation is calculated by HxWx20, where H and W are the height and width, respectively, of the input LR image. Next, an (8, 5) convolutional layer (808 parameters) is added to learn final deep features.

In the super-resolution literature, three of the most common methods of upsampling in deep models are bicubic interpolation, deconvolutional layers, and pixel shuffling. Increasingly, pixel shuffling (also referred to as sub-pixel convolution) techniques are being used [17, 21, 31, 22] and have been shown to have better efficiency. Therefore, pixel shuffling is utilized for the final block. The pixel shuffling block is composed of two steps. The first utilizes a convolution with a number of filters equal to r^2, where r is the upscaling factor. Thus, for the 2x super-resolution model, 4 filters are used for the pixel shuffling layer. After

the convolutional layer, the r^2 filters are rearranged into the output super-resolution image of size HrxWr.

After each convolutional layer in the model, a non-linear activation function is applied. During testing of the proposed model, using PReLU activation functions increased the testing SSIM when compared to ReLU and LeakyReLU activation functions. This is a result of the learnable parameters (one per convolutional layer filter) of the activation function. In effect, PReLU is a learnable version of the LeakyReLU function, allowing some negative feature coefficients to pass through at a rate learned for each feature map. This significantly improves the efficiency of the network with a small parameter (28 in total for the proposed model) and computational cost. After the pixel shuffling block, a clipping ReLU activation is applied to ensure the output image is within the [0,1] range.

3.3. Data

The DIV2K dataset [27] is widely used as a training set for supervised super-resolution models. The dataset is split into 800 training, 100 validation, and 100 test RGB images ranging in size from 648x2040 to 2040x2040. These images have a large diversity of contents, providing a wide variety of features for the model to learn. The TinyPSSR model is trained on the 800 DIV2K training images. A validation set is constructed by taking 1,000 images of size 500x500 from the ILSVRC [28] validation split. This dataset contains a large number of high-resolution images, helping to prevent overfitting during training.

Before passing the images to the model, the training and validation sets are passed through a number of pre-processing steps. Since the DIV2K training images are not all a consistent size, they are split into non-overlapping 64x64 patches. Splitting large training images into smaller patches allows the model to learn more precise features, whereas large images can sometimes be dominated by smooth regions with low-frequency features. I then downscale the validation and patched training images by two times in width and height and convert from uint8 to float32 in the range [0,1] to produce the low-resolution input images to the model. The final training dataset consists of 522,939 images of size 32x32 with ground truth images of size 64x64.

Performance of the TinyPSSR model is tested using the standard Set5 [23], Set14 [24], BSD100 [25], and Urban100 [26] datasets. The first two are standard datasets containing 5 and 14 images, respectively, with some overlap, while the last two are larger datasets of 100 images that are higher resolution and represent more real-world images. Each dataset is tested on both 2x and 4x downscaled images as provided by the respective authors.

Figure 3.2 shows example images from the DIV2K, ILSVRC, Set5, Set14, BSD100, and Urban100 datasets.

3.4. Training

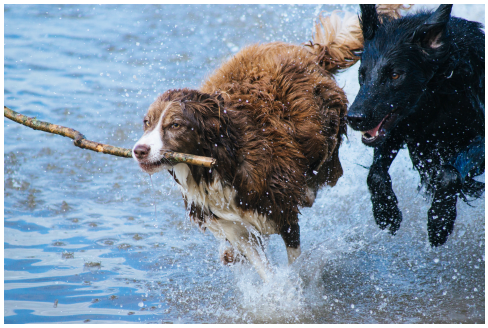
The proposed model is implemented in TensorFlow and uses the built-in Adam optimizer [32] setting parameters β_1=0, β_2=0.9, and learning rate to 2×10^{-4}. The model is trained until it reaches 50 epochs with no improvement in the loss metric on the validation split in order to ensure the model weights have converged. The SSIM loss function is selected as the loss metric for the optimizer.

3.5. Metrics

Following previous literature, PSNR and SSIM metrics are calculated on the Y-channel (luminance) of the YCbCr color space. Since the proposed TinyPSSR model super resolves each channel in an image separately, the RGB channels are combined and converted to the YCbCr color space before calculating metrics. In the case of grayscale images, the grayscale channel is considered as the Y-channel. Conversions are performed using TensorFlow's `rgb_to_ycbcr` implementation and SSIM and PSNR are calculated using TensorFlow's methods for each.

When training the proposed model, a number of different loss functions were tested including mean squared error, mean absolute error, and SSIM. Since neural network optimizers aim to minimize the loss, the SSIM loss is defined as

$$\text{SSIM}_{\text{loss}}(x) = 1 - \text{SSIM}(x). \tag{13}$$



(A) DIV2K

(B) ILSVRC

(C) Set5

(D) Set14

(E) BSD100

(F) Urban100

FIGURE 3.2. Example images from DIV2K, ILSVRC, Set5, Set14, BSD100, and Urban100.

3.6. Implementation

TinyPSSR is trained and tested using TensorFlow in Python. However, TensorFlow incurs significant overhead while setting up the model, so it is difficult to properly examine the inference speed of a model. While TensorFlow does perform a significant amount of optimization as far down as the CPU instruction level, an implementation in C is a more accurate representation of the true execution speed in practice. For that, a simple C program is written to implement the convolutional and PReLU layers of TinyPSSR alongside the Python implementation.

3.6.1. Python

Before the model can be trained and tested, the data must be pre-processed. For this, a `data_processing.py` script is written. In it, aggregation functions are written for a number of different image sets that were tested during the process of the project. These sets are:

- Set5 [23]
- Set14 [24]
- BSD100 [25]
- Urban100 [26]
- DIV2K [27]
- ILSVRC [28]
- FLIR [33]
- OSU infrared pedestrian [34]

Since the source for each of the sets is different, the aggregation functions are slightly different. Other than the four test sets (BSD100, Urban100, Set5, Set14), most sets have a training and validation split. Each split is read in, stored as a NumPy array, and saved locally as '.npy' files. This allows for more rapid testing so that conversion from JPEG or PNG images is not necessary every time. For sets that provide high- and low-resolution splits, these are saved separately. The test sets Set5, Set14, BSD100, and Urban100 are collated by the authors in [35] and provided online with 2x, 3x, and 4x downscaled versions at [36].

After aggregating, splits without provided low-resolution images can be run again through the script (with a different flag) to be pre-processed. Multiple pre-processing flags

are provided to allow for grayscaling, patching, and downscaling of the images. Downscaling and patching have further properties to determine specific attributes (i.e., patch size, downscale size, downscale method). For all training and testing, the downscale method is done using linear interpolation with the 'area' option through OpenCV. Grayscaling is also done using the OpenCV library. Patching is done by sub-indexing the image arrays and splitting into a larger array after optional grayscaling and prior to optional downscaling. At this point, all data is retained as 8-bit unsigned integers to avoid any potential floating point losses when saving the data.

To improve the ease of use, a utility script `util.py` is written to go along with all training and testing scripts. This script provides interfaces for loading training and testing data (`load_downscale_dataset()`, `get_test_set()`), upscaling images through a provided model or using the bicubic method (`upscale_images()`, `bicubic_upscale()`), calculating SSIM and PSNR metric results (`calculate_metrics()`), and plotting comparison images (`plot_image_reconstructions()`).

For modularization of the code, the custom loss and metric functions are included separately in the `custom_losses.py` and `custom_metrics.py` scripts. The custom function wrappers for SSIM and PSNR calculations are included in the `custom_functions.py` script. The TinyPSSR model is included in the `models.py` script. This model is defined as a subclass of the TensorFlow Keras model object, so that the testing functionality can be overwritten. Along with the custom parameters added in the constructor, this allows for extra validation sets to be added during the training. This is used to test on any of the testing sets so that the progress may be tracked more closely while training a model instead of waiting (potentially a significant amount of time) for the model to finish training before testing it.

The model and simulation parameters are set in the main script, `main.py`. This script is the one called to initiate the model training. A number of commented parameters are included in here to easily adjust any part of the model, training, and/or testing. Additionally, a connection to the online service Weights & Biases (WandB) is added. WandB is extremely useful for cataloging and visualizing machine learning models, their parameters, and results.

For post-training testing (after the `main.py` script), the `reconstruction.py` script is used. This allows any previous model to be called using the files stored with WandB and any testing set may be read in, upscaled, compared to bicubic, have the metrics calculated, display comparisons upscaled images, and save the metrics to a csv file. During the uspcaling process, the execution time for each image being upscaled is calculated to get an idea of the performance of the model.

3.6.2. C

The C implementation of TinyPSSR is a generic convolutional neural network implementation. In fact, it is written abstractly to allow any CNN model to be run instead of just TinyPSSR. Because of this, there are likely some optimizations that could be made to improve its performance. The implementation consists of a main file along with helpers for the activation and layer implementations. Additionally, the implementation consists of a simple library to read and parse the MNIST dataset.

Before running the model in C, the model weights must be converted to a simpler representation that can be read from C (since TensorFlow stores models in their own binary file format). For this, another python script `weights_export.py` is written. This script generates a binary file for each convolutional layer and PReLU activation that contains the weights and bias values stored in IEEE.754 32-bit float format. This allows the C program to directly read the binary data from the file into the address location stored by the pointer to the weights and biases for each layer without any conversion (and allows for the smallest possible uncompressed weights files).

Since writing a C image format parser is outside the scope of this project, another python script, `export_binary_image_data.py`, is written to export the unsigned 8-bit integer image data for the Set5 and Set14 test sets into binary files. As with the weights, this allows for the C program to easily read in each image.

In the C program, simple constructors are written to create a basic API for the creation of the convolutional and PReLU layers. For this, the basic hyperparameter information and the weights file paths are passed to create structs for each layer. A single allocation is

made for input and output layers equal to the largest required concurrent memory usage. This is after the first layer, which requires 16 output filters. This is more than is strictly necessary as instead, rotating buffers could be used with one allocating 16 filters and the other allocating 4 as would likely be done in a true application with strict memory requirements.

At this point, the entire model is allocated and set up with each layer having a forward function implemented in the respective libraries and can be called easily using the `forward` property of their respective struct objects (this is all set up in the layer constructor). Finally, to measure the execution time of the model, the `clock()` function and `clock_t` datatypes from `time.h` are used.

CHAPTER 4

TINYPSSR RESULTS

The **Tiny P**ixel **S**huffling **S**uper-**R**esolution (TinyPSSR) model as proposed in the previous chapter has the main focus of maintaining the quality of upscaled images while keeping a low memory footprint and computational cost. This chapter first explores various combinations of hyperparameters to analyze how they affect the metric quality results. Then, the model is compared with other works and perceptual quality of the results is discussed. Finally, TinyPSSR is extended to the infra-red thermal super-resolution problem.

4.1. Hyperparameter Evaluation

To identify the best model, different hyperparameters and training parameters must be changed. Those that affect the metric results the most are the activation functions after the convolutional layers, the loss metric, the training data, and the scale at which the model is trained. This section covers the testing of various values for each of these. Unless otherwise specified, each model in this section is trained using the SSIM loss, PReLU activation, 700,000 images from the RGB channel-split DIV2K 64x64 patched training data, and trained at 2x downscale.

4.1.1. Activation Function

Most SISR (and most machine learning models in general) tend towards the ReLU activation function. Due to local minima in the error surface, some fully convolutional neural networks will instead use Leaky ReLU with a small (typically less than 0.1) α value. Figure 4.1 shows the results of tests using the Growing Cosine Unit (GCU), GCU and PReLU, Leaky ReLU ($\alpha = 0.1$, 0.2, 0.5, and 0.75), ReLU, and no activation function. Table 4.1 and Table 4.2 show the testing results for SSIM and PSNR, respectively.

The results show that the combination GCU+PReLU outperforms all other activations by a significant margin in SSIM while PReLU wins out in PSNR. GCU+PReLU performs similarly to both GCU and PReLU individually as well. Since GCU requires both

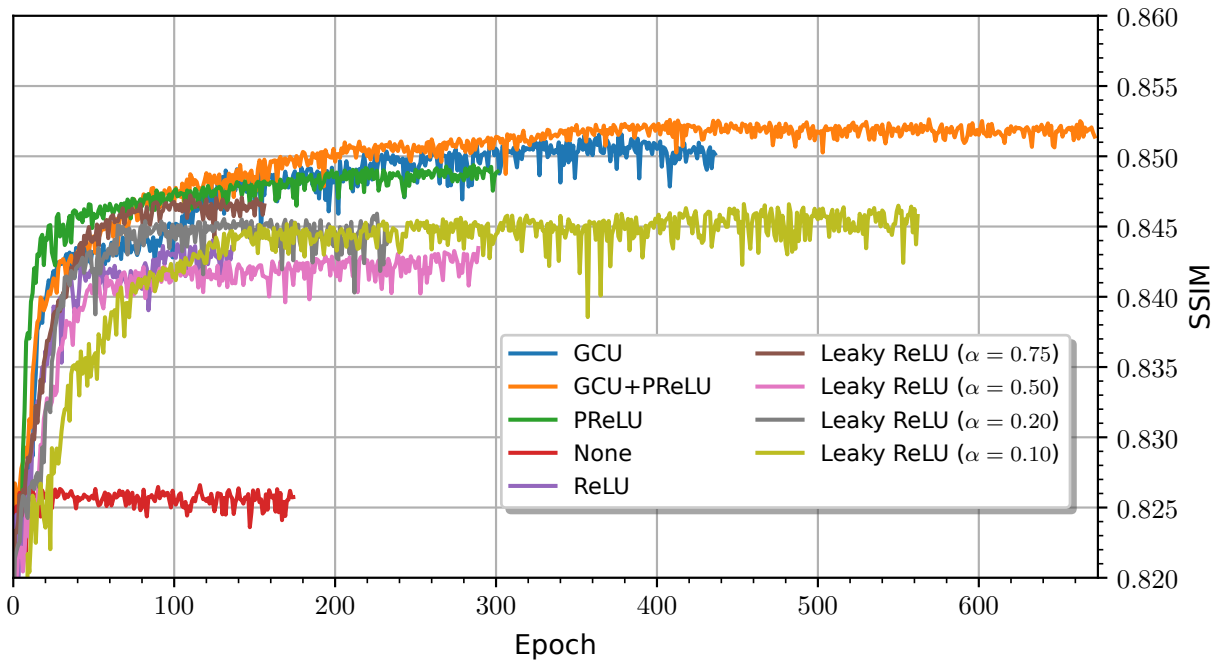


FIGURE 4.1. Training history SSIM comparison on Set5 at 4x upscale between various activation functions.

Activation	Set5		Set14		Urban100		BSD100	
	2x	4x	2x	4x	2x	4x	2x	4x
GCU+PReLU	**0.9492**	**0.8517**	**0.9015**	**0.7502**	**0.8897**	**0.7146**	**0.8814**	**0.7115**
GCU	0.9486	0.8505	0.9002	0.7479	0.8875	0.7116	0.8798	0.7093
PReLU	0.9483	0.8485	0.9004	0.7474	0.8865	0.7091	0.8802	0.7092
LReLU $\alpha = 0.50$	0.9482	0.8468	0.8997	0.7460	0.8858	0.7077	0.8792	0.7078
ReLU	0.9479	0.8462	0.8993	0.7455	0.8853	0.7078	0.8794	0.7087
LReLU $\alpha = 0.10$	0.9478	0.8451	0.8997	0.7455	0.8856	0.7074	0.8796	0.7082
LReLU $\alpha = 0.20$	0.9472	0.8426	0.8995	0.7440	0.8849	0.7056	0.8790	0.7071
LReLU $\alpha = 0.75$	0.9472	0.8414	0.8995	0.7439	0.8839	0.7046	0.8797	0.7079
None	0.9411	0.8254	0.8938	0.7321	0.8723	0.6873	0.8739	0.6988

TABLE 4.1. SSIM results for various activation functions ordered by descending Set5 4x results. The highest result in each column is in bold, second highest is underlined.

a multiplication and a cosine call for each pixel value in each filter, it is much more expensive than a simple PReLU activation that only requires a multiplication when the input pixel value is less than zero. Trigonometric functions are also not as efficient and are not possible on embedded devices without FPU units. The most efficient edition of TinyPSSR would select PReLU as the activation function after all but the last layer (whose activation

Activation	Set5		Set14		Urban100		BSD100	
	2x	4x	2x	4x	2x	4x	2x	4x
PReLU	**35.136**	**29.197**	31.337	26.501	28.045	**23.826**	30.609	26.397
ReLU	35.045	29.107	31.274	26.462	27.992	23.782	**30.628**	**26.418**
LReLU $\alpha = 0.20$	34.988	29.088	31.227	26.431	27.958	23.766	30.573	26.385
LReLU $\alpha = 0.10$	34.952	29.032	**31.352**	**26.520**	27.998	23.798	30.602	26.377
LReLU $\alpha = 0.50$	34.991	29.030	31.304	26.443	28.024	23.769	30.603	26.356
GCU	34.873	28.988	31.268	26.411	28.065	23.791	30.526	26.269
LReLU $\alpha = 0.75$	34.703	28.725	31.083	26.189	27.819	23.596	30.468	26.210
GCU+PReLU	34.768	28.720	31.159	26.192	**28.143**	23.740	30.539	26.150
None	34.327	28.672	30.729	26.047	27.377	23.427	30.232	26.177

TABLE 4.2. PSNR results for various activation functions ordered by descending Set5 4x results. The highest result in each column is in bold, second highest is underlined.

function would be a [0,1] clipping ReLU). Fortunately, PReLU does not lose much quality w.r.t the SSIM metric with an average 0.25% drop across the four test sets and obtains the best PSNR results.

4.1.2. Loss Metric

Activation	Set5		Set14		Urban100		BSD100	
	2x	4x	2x	4x	2x	4x	2x	4x
MAE	0.9476	**0.8488**	0.8953	0.7402	0.8813	0.7032	0.8730	0.7006
SSIM	**0.9486**	0.8487	**0.9002**	**0.7478**	**0.8873**	**0.7109**	**0.8800**	**0.7099**
MSE	0.9468	0.8457	0.8949	0.7392	0.8809	0.7012	0.8725	0.6994

TABLE 4.3. SSIM results for various loss functions ordered by descending Set5 4x results. The highest result in each column is in bold, second highest is underlined.

Activation	Set5		Set14		Urban100		BSD100	
	2x	4x	2x	4x	2x	4x	2x	4x
MAE	**35.568**	**29.813**	**31.449**	26.753	28.148	23.982	30.664	**26.618**
SSIM	35.163	29.199	31.412	26.538	28.118	23.859	**30.686**	26.457
MSE	35.534	29.804	31.424	**26.781**	**28.202**	**24.006**	30.651	26.600

TABLE 4.4. PSNR results for various loss functions ordered by descending Set5 4x results. The highest result in each column is in bold, second highest is underlined.

When training a model, the loss function will help it learn what features to focus on. Naturally, mean absolute error (MAE) and mean squared error (MSE) that focus on single

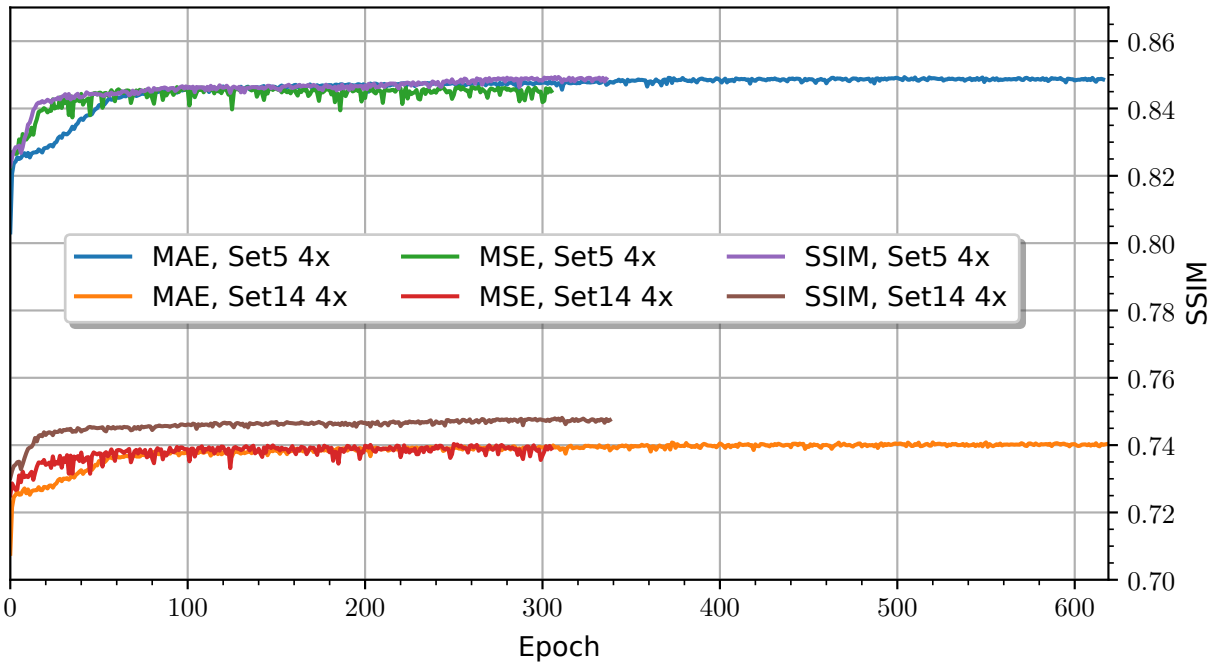


FIGURE 4.2. Training history SSIM comparison between various loss metrics.

pixel average loss values will improve PSNR results while an SSIM loss metric will improve the SSIM metric. However, it is still important to look at the metric results and compare the loss functions to determine if a loss in one metric is worth a gain in the other. Table 4.3 and Table 4.4 show the metric results for SSIM and PSNR, respectively, of the three loss functions. Figure 4.2 shows the SSIM results at each epoch during the training process on both Set5 and Set14. There is a significant difference in the results between these two models primarily due to the mandrill image from Set14. This image contains a large number of high-frequency features (namely, whiskers and fur on the face) that is very difficult to super-resolve, resulting in very low SSIM and PSNR results. Since there are so few images in each set, the average results are heavily impacted by a few outliers.

One very interesting, and somewhat unexpected, result from these results is that MAE performs somewhat close to the SSIM-trained models in the SSIM measurement. A significant portion of state-of-the-art models train on MSE loss and most image quality metrics (particularly in the compression field) tend to focus on MSE and PSNR (which is derived from MSE). This would lead one to believe that MSE would be a driving force for

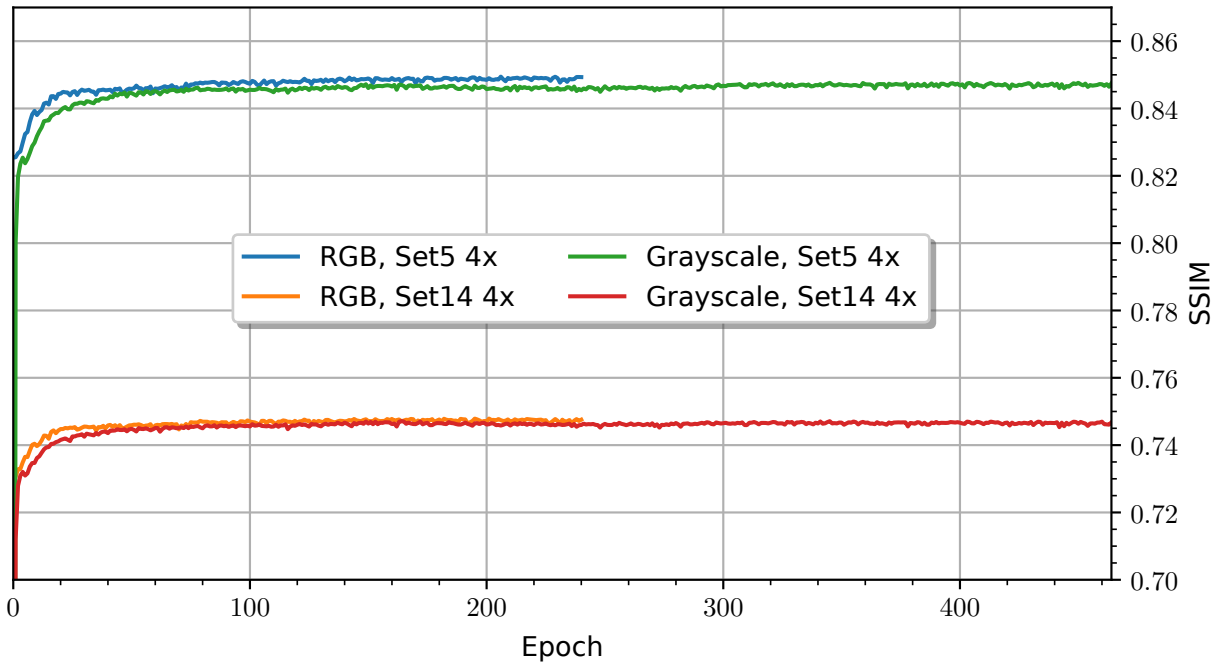


FIGURE 4.3. Training history SSIM comparison between training on RGB and grayscale images.

higher fidelity images while the results here show that TinyPSSR obtains better quality when putting less focus on outlier error pixels.

4.1.3. Training Data

Many SISR models train on full-color RGB images. However, in order to maintain a small memory footprint as well as be adaptable to grayscale images, TinyPSSR is designed to take one channel at a time. It naturally then makes sense to test on grayscaled images. However, this likely removes some color channel information that the model could use to learn cross-channel features and improve the structural or luminance quality. Thus, the model can be trained on both grayscale and RGB images. Since the model only takes one channel at a time, RGB training images are split into the three channels before being fed through the network. Figure 4.3 compares the SSIM metric for each epoch during grayscale and RGB training. Table 4.5 and Table 4.6 compare the SSIM and PSNR testing results, respectively.

As expected, the RGB training does somewhat improve on the metric results over

Color	Set5		Set14		Urban100		BSD100	
	2x	4x	2x	4x	2x	4x	2x	4x
RGB	**0.9489**	**0.8487**	**0.9006**	**0.7477**	**0.8876**	**0.7111**	**0.8807**	**0.7106**
Gray	0.9484	0.8473	0.9001	0.7468	**0.8876**	0.7106	0.8800	0.7095

TABLE 4.5. SSIM results compared between training on RGB and grayscale images. The highest result in each column is in bold.

Color	Set5		Set14		Urban100		BSD100	
	2x	4x	2x	4x	2x	4x	2x	4x
RGB	**35.124**	**29.125**	**31.336**	**26.472**	28.087	23.824	**30.687**	26.425
Gray	35.056	29.105	31.327	26.470	**28.118**	**23.837**	30.649	**26.434**

TABLE 4.6. PSNR results compared between training on RGB and grayscale images. The highest result in each column is in bold.

the grayscale training – moreso in PSNR than in SSIM. SSIM naturally does not improve as much as the grayscale images still provide the structure of an image. Adding color adds very little to how defined edges and shapes are. Instead, providing full-color information allows the model to more accurately reconstruct the individual pixel values for each channel. Since MSE measures pixel-wise error, and is a component of the PSNR calculation, this directly results in an increase PSNR value. A little more improvement may be possible, however, there are memory constrains when training on RGB data. Since it uses three times the amount of information per image, a third of the number of images can be used during training. The original grayscaled DIV2K training set after being patched to 64x64 (and downscaled to 32x32), contains 522,939 image patches. Empirically, I found that the training system could withstand up to 700,000 training patches before running into memory overflow errors. With RGB splitting, that moves to around 233,000 full-color image patches.

4.1.4. Model Scale

The vast majority of super-resolution models are designed and trained specifically for one scale (e.g., 2x, 3x, 4x, etc.) with parts of the model replaced and/or re-trained for different scales. With the goal of a small, performant, and adaptable model, TinyPSSR is designed to be used at multiple scales, so it is normally trained at 2x so that it may be used at other multiples of two (namely, 4x and 8x) by iterating through the model another time.

At first glance, this would seem to increase computational cost, however, moving TinyPSSR to 4x scale doubles the parameters from 1,356 to 2,716, so the computational cost is about the same. The total required size to store the model is naturally reduced by utilizing the iterative approach, though there is a slight real-time memory cost as a larger image must be sent through the second time.

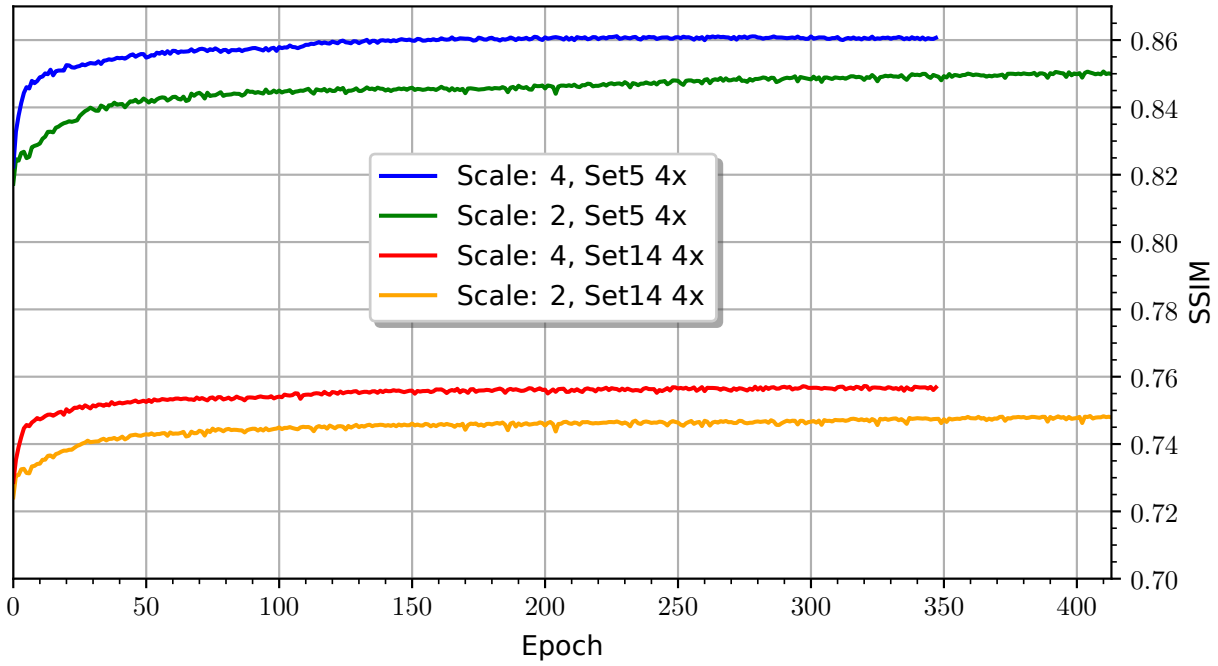


FIGURE 4.4. Training history SSIM comparison on 4x super-resolution between TinyPSSR trained on 2x scale (using two iterations) and 4x scale (using one iteration).

Scale	Set5	Set14	Urban100	BSD100
2	0.8497	0.7481	0.7123	0.7106
4	**0.8606**	**0.7572**	**0.7254**	**0.7178**

TABLE 4.7. SSIM results at 4x upscaling comparing for models trained at 2x and 4x scales. The highest result in each column is in bold.

Scale	Set5	Set14	Urban100	BSD100
2	28.933	26.338	23.789	26.346
4	**29.297**	**26.549**	**23.983**	**26.395**

TABLE 4.8. PSNR results at 4x upscaling comparing for models trained at 2x and 4x scales. The highest result in each column is in bold.

To compare appropriately with other models from literature that are trained at 4x upscaling, the larger model, also trained at 4x upscale, is tested on the four test sets (at only 4x upscaling). Figure 4.4 shows the training SSIM results at each epoch for both models using the test sets at 4x upscaling. Table 4.7 and Table 4.8 show the final testing results for SSIM and PSNR, respectively.

As expected, the model trained explicitly for 4x upscaling performs noticeably better. These results are extremely promising as it beats FSRCNN-s with 32% fewer parameters and approaches full-size FSRCNN, ESPCN, and SRCNN with 20%, 87%, and 95% fewer parameters, respectively, when considering Set 5 4x SSIM. Additionally, it beats SRCNN, ESPCN, and full-size FSRCNN at Set14 4x.

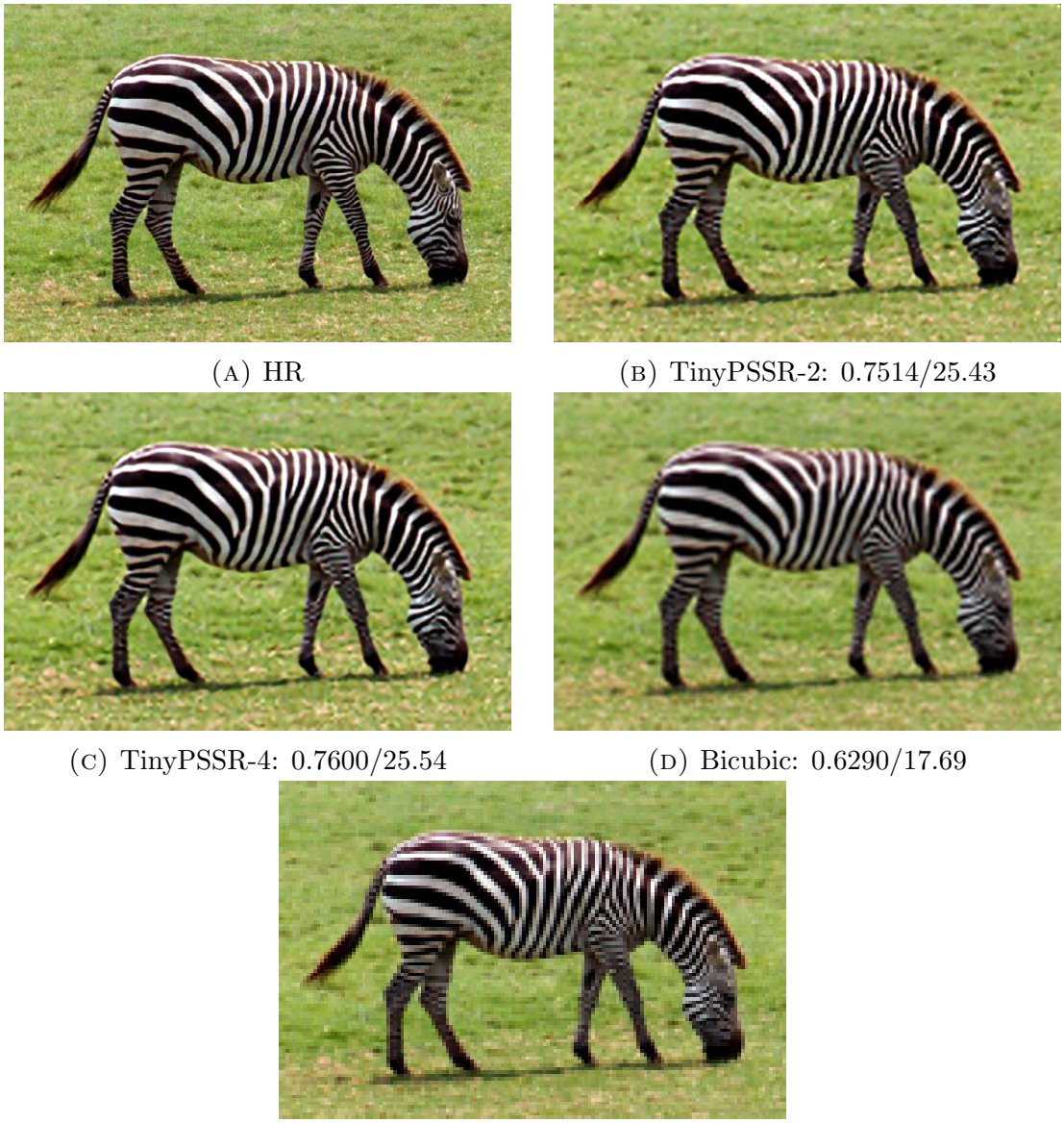
4.2. Results

As discussed in the previous chapter, the TinyPSSR model was trained on both 2x super-resolution (using an iterative process to get 4x) and 4x native. The results showed a fairly significant increase in metric quality. Thus, comparisons in this section will use both models. For ease of reference, the 2x trained model is referred to as TinyPSSR-2 and the 4x trained model is referred to as TinyPSSR-4. Both models are trained using the SSIM loss function and PReLU activation functions.

TABLE 4.9. SSIM/PSNR results for bicubic interpolation, FSRCNN-s (results are the maximum from their paper), TinyPSSR-2, and TinyPSSR-4 for various test datasets with 2x and 4x super-resolution.

Dataset	Scale	Bicubic	FSRCNN-s[16]	TinyPSSR-2	TinyPSSR-4
Set5	2	0.9226/31.964	0.9532/36.58	0.9488/34.95	–
Set14	2	0.8341/25.78	0.9052/32.28	0.9004/31.28	–
Urban100	2	0.7933/22.91	–	0.8879/28.10	–
BSD100	2	0.8287/27.233	–	0.8805/30.63	–
Set5	4	0.7890/26.07	0.8499/30.11	0.8497/28.93	0.8606/29.30
Set14	4	0.6774/23.13	0.7423/27.19	0.7481/26.34	0.7572/26.55
Urban100	4	0.6282/20.85	–	0.7123/23.79	0.7254/23.98
BSD100	4	0.6599/24.59	–	0.7106/26.35	0.7178/26.39

The average SSIM and PSNR results for all experiments on the test sets are presented



(A) HR (B) TinyPSSR-2: 0.7514/25.43

(C) TinyPSSR-4: 0.7600/25.54 (D) Bicubic: 0.6290/17.69

(E) LR

FIGURE 4.5. The "zebra" image from Set14 with 4x scale reconstructions. Each subfigure provides SSIM/PSNR metric values.

in Table 4.9 with comparisons to bicubic and the 3,937 parameter FSRCNN-s* [16]. The FSRCNN-s results are taken from the highest values provided in the paper as the original source is no longer available. Since the TinyPSSR-2 model is only trained for 2x super-resolution, 4x reconstructions are generated by iterating through the model twice. This table

*The authors of FSRCNN report 3,937 parameters not including the learned parameters of the PReLU activation function and bias values for each convolutional layer filter, which would bring the model up to 4,086.

shows that despite the small size of the model, it performs similar to the larger FSRCNN-s model. In the FSRCNN-s model, the deconvolution (i.e., transpose convolution) upscaling layer is trained separately for different upscaling factors. Thus, the TinyPSSR-4 model is included for a more fair comparison. As can be seen from the table, TinyPSSR-4 outperforms FSRCNN-s with 34% fewer parameters (2,716 vs 4,086).

The 4x super-resolution reconstruction of the "zebra" image from Set14 is shown in Figure 4.5 with the high-resolution, TinyPSSR-2, TinyPSSR-4, bicubic, and low-resolution images. In this figure, both TinyPSSR models show a remarkable improvement over the bicubic result. In particular, this result showcases TinyPSSR's ability to sharpen edges. On the smaller of the zebra's stripes closer to the head and front shoulders, the black stripes noticeably bleed over the white stripes on the bicubic result. This causes the white areas to appear a darker gray than the ground truth. In the TinyPSSR results, this bleeding hardly occurs, leaving the colors a much more true white. The TinyPSSR-4 model improves on the edge sharpness, particularly noticeable on the neck where TinyPSSR-2 is slightly jaggy and bicubic nearly blurs together.

4.3. IR Super-Resolution

While RGB images are certainly the most popular SISR task, infra-red (IR) imaging also has some interest. One of the most popular datasets is the FLIR thermal imaging dataset [33]. This dataset contains a number of corresponding RGB and thermal frame sequences (i.e., frames from videos). While the landing page for the dataset claims 9,711 thermal training and validation images, the downloaded data actually contains 10,742 training and 1,144 validation images. All of these images are of size 512x640 and contain one channel (as IR imaging is in grayscale). The original dataset also includes annotations for object detection in the still images and videos, however, that is discarded for this project. Figure 4.6 shows an example IR image from the FLIR validation set.

To train TinyPSSR on the thermal data, the provided 10,742 training images are used. Following the method for regular image training, the training images are patched into 64x64 blocks and downsampled to 2x scale. This creates a total of 859,360 training images.

FIGURE 4.6. An example image from the FLIR IR validation set.

However, this is slightly too large for the 24 GB of memory available on the training device (an NVIDIA 3090 GPU). Instead, 700,000 images from this set, a number found through experimental search to not cause memory overflow, are kept. Since the 1,144 validation set (full-size images, not patched) in addition to the training data is also quite large and would cause memory issues during training, 10% of the training set (70,000 patches) are reserved for validation, leaving the remaining 630,000 to train on. An additional test set is taken from the OSU thermal pedestrian database [34]. This set contains ten sequences of recorded pedestrians walking through a location through a university campus intersection. The length of the sequences varies from 18 to 73 frames for a total of 284 frames each of size 360x240. The first eight sequences, composed of 187 frames, is used for training, while the remaining 97 frames are used for testing.

One difficult aspect in IR super-resolution is that IR images are inherently noisy, causing a poor signal-to-noise ratio, due to characteristics of IR sensors and internal reflections within the lenses. Due to this, super-resolution in IR must also act somewhat as a denoiser. While bicubic reconstruction smooths an image out and thus loses structural information, it does perform similarly in some cases in retaining pixel quality (i.e., MSE and therefore PSNR) compared to the TinyPSSR model trained using SSIM. This is also why the

bicubic reconstruction images sometimes appear smoother than the TinyPSSR reconstructions when trained on SSIM which typically have more sharp, aliased edges. Though, this smoothness in the bicubic method comes at the cost of the stronger edges that TinyPSSR typically provides.

While SSIM as a loss function is very good at reconstructing structural information, it can fall behind in noise reduction resulting in lower PSNR metrics. For regular RGB or grayscale images, this is okay, as structural information is generally more important towards perceptual quality (how the human eye sees an image). However, due to the discussed noisy quality of IR images, an SSIM loss function generally doesn't perform as well. Instead, we can use mean absolute error (MAE) as a loss function to train the model to focus more on pixel value reconstruction than structural information. We found earlier from the results in subsection 4.1.2 that MAE as a loss function performs nearly as good as SSIM in structural information, and better in PSNR than SSIM as well as MSE on most test sets. Thus, MAE offers a good compromise between structural and pixel-wise quality. In this section, I refer to TinyPSSR with the regular SSIM loss function and trained on IR data as TinyPSSR-IR while the model with an MAE loss function as TinyPSSR-IR-MAE.

TABLE 4.10. SSIM/PSNR results for bicubic and TinyPSSR-IR on the FLIR validation and OSU thermal pedestrian test datasets.

Dataset	Scale	Bicubic	TinyPSSR-IR	TinyPSSR-IR-MAE
FLIR	2	0.8472/33.33	0.8657/35.26	0.8634/35.52
OSU	2	0.7210/32.69	0.7477/32.78	0.7405/32.90
FLIR	4	0.7409/31.08	0.7584/31.26	0.7566/31.68
OSU	4	0.6195/31.21	0.6338/31.08	0.6291/31.31

The metric results comparison between TinyPSSR-IR, TinyPSSR-IR-MAE, and bicubic interpolation on the 1,144 images of the FLIR validation set and the 97 images of the OSU thermal pedestrian test set is shown in Table 4.10. Following previous literature, the SSIM and PSNR metrics are calculated on the luminance channel of a center-cropped (i.e., removal of a 4-pixel border) image. These results show a moderate increase over the baseline bicubic super-resolution for TinyPSSR-IR and a slightly more significant improvement in

(A) HR

(B) TinyPSSR-IR: 0.6374/27.67

(C) TinyPSSR-IR-MAE: 0.6667/29.30

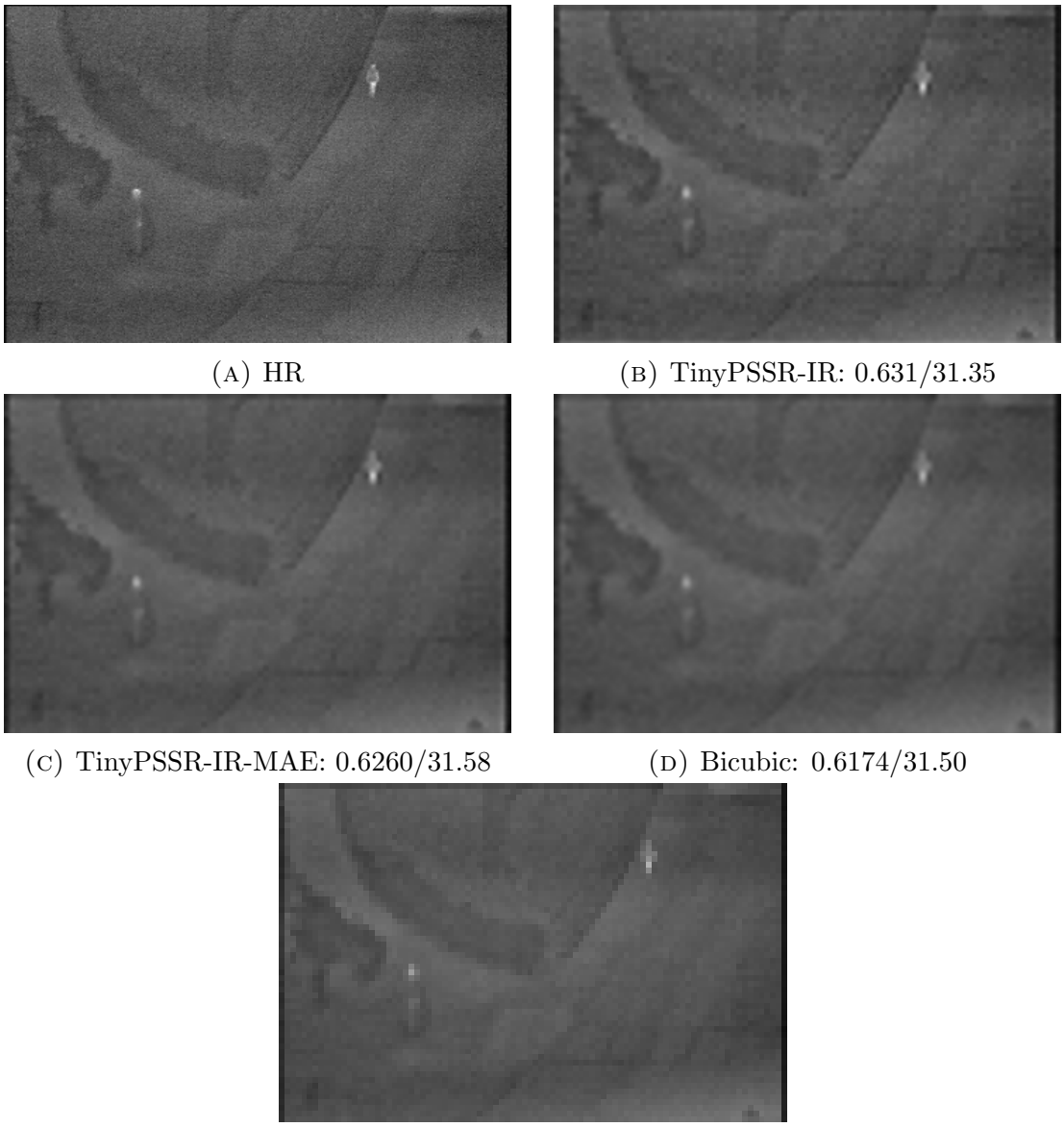
(D) Bicubic: 0.6416/29.00

(E) LR

FIGURE 4.7. Comparison 4x scale reconstruction from the FLIR validation set. Each subfigure provides SSIM/PSNR metric values.

PSNR at the cost of some SSIM percentage for TinyPSSR-IR-MAE.

One major use case for IR imaging is in object detection. For this to work, there must be a noticeable border around the object (i.e., a difference in color or structure to



(A) HR

(B) TinyPSSR-IR: 0.631/31.35

(C) TinyPSSR-IR-MAE: 0.6260/31.58

(D) Bicubic: 0.6174/31.50

(E) LR

FIGURE 4.8. Comparison 4x scale reconstruction from the OSU thermal pedestrian test set. Each subfigure provides SSIM/PSNR metric values.

differentiate it from the background). Figure 4.7 and Figure 4.8 show the results of a 4x super-resolution of an image from the FLIR validation set and the OSU thermal pedestrian test set, respectively. In the bicubic results, the structure of the image is significantly smoothed out. Thus, the important border differentiation is lost. In the TinyPSSR-IR reconstruction, a slight boundary region is created by reproducing some of the lower intensity thermal readings from the background around an object (more specifically, the pedestrians

in either image). TinyPSSR-IR-MAE combines the best of both and produces images with good separation of objects while also reducing some of the noticeable noise apparent in TinyPSSR-IR resulting in high-quality reconstructions.

CHAPTER 5

DEEP IMAGE COMPRESSION

A raw color (RGB) image typically stores one byte (an unsigned integer between 0 and 255) per channel per pixel. For a full HD image of size 1920x1080, this requires 6.2 MB of storage space. For modern storage capabilities, this isn't overly expensive, as a common 32 GB phone could store around 5,100 of these images – assuming we have no operating system, apps, music, or other files, which can easily take 10-20 GB. However, new phones capture images of even higher resolutions, commonly above 12 megapixels which requires 36 MB of space to store raw, reducing the number of images our 32 GB phone can store down to less than 1,000. While one can simply pay more and get the 128 GB version of our phone, we will still have some trouble downloading a large number of these images over the internet.

In order to store more selfies, we need to use image compression standards to efficiently reduce the file size to a more reasonable and manageable size. Image compression comes in both lossless and lossy techniques. In lossless techniques, the full raw image can be recovered from the compressed file and typically uses various entropy encoding methods (such as arithmetic, Huffman, or run-length encoding) implemented in standards including PNG and JPEG2000. Images compressed using lossy techniques however, cannot be fully recovered and, instead, are merely approximated with some (usually, acceptable) quality loss. Standards that implement lossy compression include JPEG, WebP, and BPG.

While traditional compression techniques do perform some dynamic compression based on statistical qualities and local information, it is notoriously hard for humans to find deep patterns in digital information that can be used to further compress information without significant quality loss. This is where deep learning has been increasingly applied to produce compression techniques that perform at or above the level of traditional compression schemes. As with many machine learning applications, often models achieve good performance at the cost of memory and computational performance which, in turn, reduces the applicability and deployability of such models.

In this chapter, I propose an efficient convolutional deep image compression model with similar computational costs and reconstruction quality to JPEG. The proposed model also makes use of bit-plane quantization and adaptive arithmetic coding to further reduce the bits required per pixel of an input image. The chapter is organized as follows. First, the model architecture is introduced, then the techniques for quantization and encoding are discussed. Finally, the experiments and simulations are designed and the results and conclusions are presented.

5.1. Model Design

In the deep compression research space, a number of different model architectures have been proposed including autoencoders, residual networks (RNNs), and generative adversarial networks (GANs) [37]. The most basic is the autoencoder (AE), such as in [38], which follows the natural process of compression. AEs are divided into an encoder and decoder. A full-sized input is provided to the encoder and encoded into a smaller size (known as the latent dimension). This latent information is the compressed input. To reconstruct, the latent information is sent through the decoder and the output is the reconstructed signal. A general model of an autoencoder is shown in Figure 5.1. In image processing, many times the input size is arbitrary, so only convolutional layers are used resulting in convolutional AEs (CAEs).

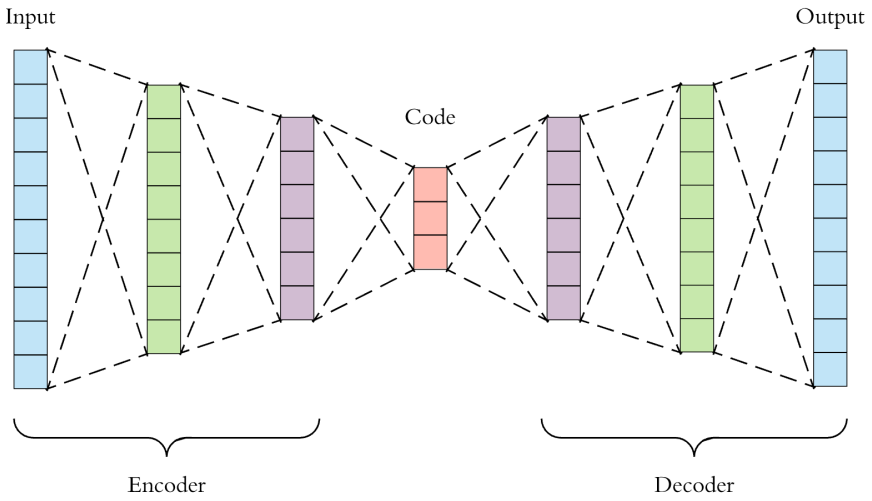


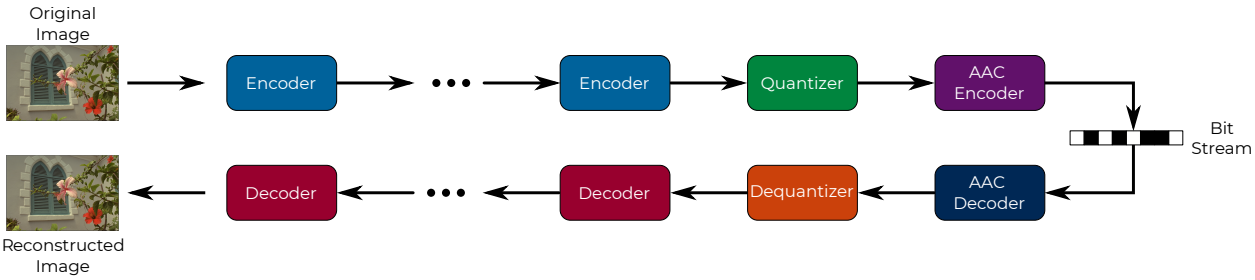
FIGURE 5.1. General model of an autoencoder.

Given that the design goal of this model is high performance and low computational and memory cost, the model should be designed with an emphasis on relatively low-complexity with a single forward path, similar to the design of TinyPSSR in Chapter 3. Thus, an autoencoder architecture is chosen. For the upscaling operations of the decoder model, the pixel shuffling layers as used in TinyPSSR are used again as they are more efficient than simple scaling or de-convolutional layers with stride greater than one. For downscaling operations, the equivalent pixel shuffling operation is used to convert spatial information into channel depth (as opposed to channel depth to spatial for the upsampling operation). For example, an input downscaled in 2x2 blocks using this method from size HxWxC produces an output of size $\frac{H}{2}$x$\frac{W}{2}$x4C. To improve the computational efficiency and reduce the number of parameters (and thus memory requirements), depthwise convolutional layers are again used. Particularly, depthwise convolutions are important after downscale pixel shuffling operations to reduce the much deeper feature maps.

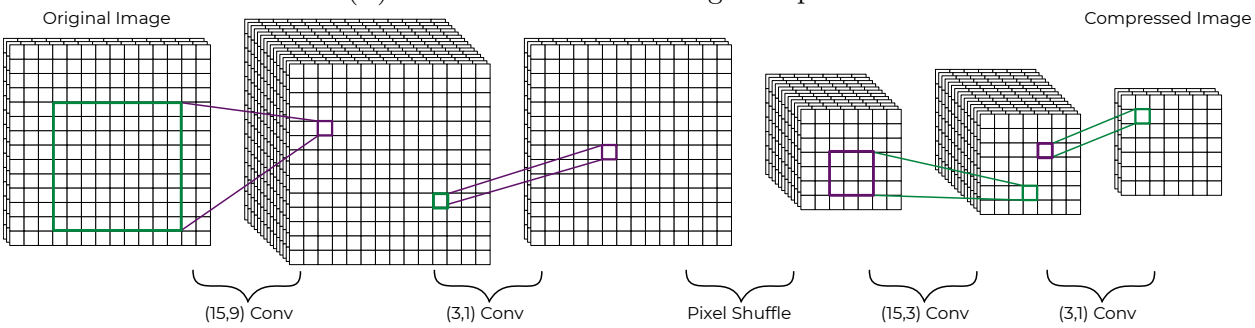
Initially, I developed three separate models for 2x, 4x, and 8x downscaling (compression ratios of 75%, 93.75%, and 98.44%, respectively). However, the larger downscaling models produced very poor results while the 75% compression model produced very good results. Thus, I combined all three into an iterative approach, similar to TinyPSSR, and trained on all three compression ratios together. This allows the larger compressions, a much harder task, to learn features from smaller compressions. This showed a significant improvement, particularly in color accuracy, which is a very difficult problem to solve. For ease of reference, from here on I will refer to the models as TinyCompress-4, TinyCompress-16, and TinyCompress-64.

This iterative approach produces a scheme as shown with the encoder and decoder blocks in Figure 5.2. The encoder and decoder blocks perform 2x up and downscaling (75% compression). Performing N iterations (that is, first N passes through the encoder, then N passes through the decoder) results in 2^N downscaling, or a compression scale of $(\frac{1}{2^N})^2$. For example, $N = 3$ passes performs 8x scaling for a compression scale of $\frac{1}{64} = 0.016$ (98.44%).

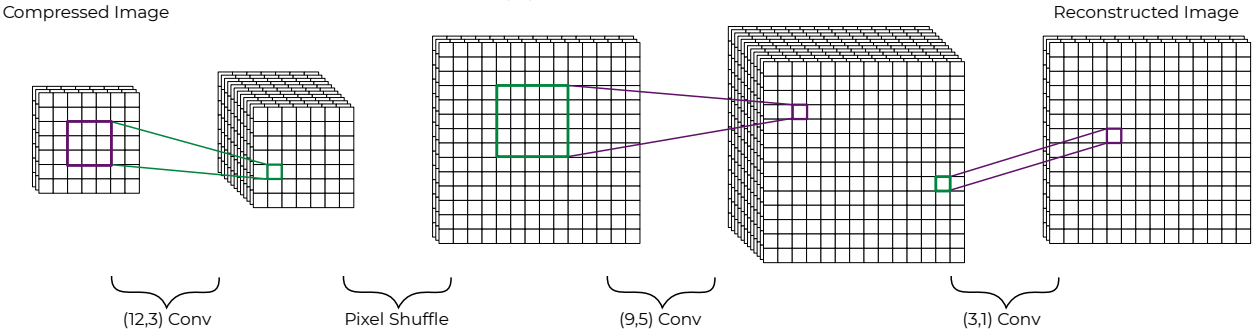
The input and outputs for both the encoding and decoding blocks is three chan-



(A) Iterative scheme for image compression.



(B) Encoding block.



(C) Decoding block.

FIGURE 5.2. The iterative deep compression scheme with encoding and decoding blocks.

nels with arbitrary height and width. For grayscale inputs, the intensity values are duplicated three times and concatenated so that all RGB channels have the same value. While TinyPSSR performed well on single-channel information (and actually did not see any improvement on RGB inputs), image compression requires that all three channels are passed as inputs concurrently. This is because in image compression cross-channel information is very important as it provides another vector of data that can be calculated as redundant or of little use (or of important use if it is harder to compress).

Following the activation function results found in testing TinyPSSR, a PReLU activation function is used after each convolutional layer, except the final layer in the decoders.

For the final layer, a sigmoid activation function is used. This proved to slightly improve the reconstruction results over a [0,1] clipping ReLU as it is effectively normalizing the results into the [0,1] range instead of clipping only those out of bounds.

5.2. Quantization

5.2.1. Floating Point to Integer

This model is trained using 32-bit (4 byte) floating-point calculations. The inputs are expected to be normalized into the [0,1] range as are the outputs of the decoder. However, if the compressed information were to be transmitted as floating point information, the compression rate would be quadrupled since it is four times as large as an unsigned 8-bit integer that images are normally stored in. Thus, we need a method of quantizing the compressed information to realize the expected compression rates.

The floating point to integer quantization process simply rescales the compressed image into the [0,255] range and casts the values into unsigned eight-bit integers (uint8). To do this, we first normalize the compressed image over the range [0,1] by finding the minimum and maximum values from the compressed image across all channels. These values can be easily found during the final activation layer, so there are no extra computational costs for scanning. The minimum value is then subtracted from each value in the compressed image and divided by the difference between the maximum and minimum. After rescaling into [0,1], each pixel is multiplied by 255 and cast to uint8. The computational cost for this entire process is relatively low, requiring only one extra scan and one multiplication (of $\frac{255}{X_{max}-X_{min}}$) per value in the compressed image

5.2.2. Bit Planes

Color quantization is a common technique in many image compression techniques. Typically, a color palette is generated or decide prior to the compression where the colors in the image are mapped to some palette (e.g., a set of 256 unique colors). Since each pixel is represented by three bytes (one each for the red, green, and blue channels), an image can be decomposed into 24 bit planes $\boldsymbol{b} \in 0, 1^{HW}$, where H and W are the height and width of the

compressed image, with eight for each channel (i.e., the first eight are for the red channel, next eight for green, and final eight for blue). The bitplanes are ordered so that in each set of eight, the first contains the most significant bit (MSB) and the last contains the least significant bit (LSB).

To quantize compressed signal using bit planes, we can simply remove a number of bit planes starting with from the LSB (since the MSB planes will contain the most important, i.e. *significant*, data). Quantizing in this manner reduces the overall number of possible values to 2^B where B is the number of bit planes remaining. For example, quantizing to 6 bit planes reduces the total number of values to $2^6 = 64$, where each value is a multiple of four (4, 8, ..., 256) since the two LSBs are zeroed. By doing this, we improve bitrate during the encoding process as the number of possibilities is reduced.

5.3. Entropy Encoding

An encoding scheme, a lossless technique, is used after compression and quantization in order to further reduce the bitrate. Typically, the quantized results are stored at a constant rate (i.e., eight bits per uint8 value). However, we can take advantage of the statistical properties of an image and represent more common pixel values with fewer bits and less common values with more. This way, the average number of bits per value is reduced overall.

Arithmetic coding encodes an entire input into a single number over the floating point range [0.0,1.0). Typically, a probability distribution is provided for each character in the input alphabet (for uint8 data, this is the numbers 0 through 255). However, since the decoder needs to perform the exact same operations in order as the encoder, this means that the probability distribution needs to be communicated without any error. Thus, no quantization or lossy compression can be used to reduce the size of this probability table. This makes it quite expensive to perform this method.

Adaptive arithmetic coding fixes these problems by updating the probability of each symbol while reading in the symbols in sequential order. The decoder can simply initialize the frequency table in the same manner as the encoder and decode sequentially without

Algorithm 1: Adaptive Arithmetic Coding

```
1  total ← 256
2  underflow ← 0
3  freqs ← 1^256
4  cumfreqs ← [0, ..., 0]
5  for i = 0 to 255 do
6      cumfreqs[ i + 1 ] ← cumfreqs[ i ] + freqs[ i ]
7  for sym ∈ sequence do
8      range ← high - low + 1
9      symlow ← cumfreqs[ sym ]
10     symlow ← cumfreqs[ sym + 1 ]
11     newlow ← low + ⌊ symlow * range / total ⌋
12     while (low ⊕ high) & (1 << 32) do
13         bit ← low / (1<<31)
14         write bit to output
15         while underflow > 0 do
16             write (bit ⊕ 1) to output
17             underflow ← underflow - 1
18         low ← (low * 2) & ((1 <<32) - 1)
19         high ← ((high * 2) & ) — 1
20     while low & ∼high & (1 << 30) do
21         underflow ← underflow + 1
22         low ← (low * 2) ⊕ (1 <<31)
23         high ← ((high ⊕ (1 <<31)) <<1) — (1 <<31) — 1
24     total ← total + 1
25     freqs[ sym ] ← freqs[ sym ] + 1
26     for i = sym to 255 do
27         cumfreqs[ i + 1 ] ← cumfreqs[ i ] + freqs[ i ]
```

needing to transmit a frequency table for the image. The pseudocode for this algorithm is shown in Algorithm 1. With this technique combined with the quantization and the autoencoder model, images can be fully compressed, transmitted (or saved to a file system) as a bitstream, de-compressed and viewed.

5.4. Experiments

To test the deep compression model, it is implemented in Python using TensorFlow. The training setup remains the same for each of the three models and mirrors that of TinyPSSR. For weight updating, the Adam optimizer is used with a learning rate of $5\text{x}10^{-4}$ and the parameters β_1 and β_2 are kept at the default values of 0.9 and 0.999, respectively. The model is trained until it has gone 20 epochs without an improvement in the validation loss metric.

The model uses the RGB images from the DIV2K dataset for training. Each image is split into non-overlapping 64x64 patches. Due to memory limitations, only 100,000 of the patches (out of 522,939) are retained for training. For validation and testing, the Kodak PhotoCD dataset [39] containing 24 images of size 768x512 (some are rotated to ensure consistent dimensions) is used. The Kodak dataset is one of the most widely used sets for image compression validation in the literature. Additionally, the Set5 [23] and Set14 [24] datasets are used during validation.

The selected loss metric is mean squared error (MSE). A number of other loss metrics were tested including SSIM (on grayscale), MAE, inverted PSNR, the 1976 and 1994 editions of the ΔE color difference metric, MS-SSIM, and weighted combinations of SSIM, MS-SSIM, MSE, and MAE. However, of these, MSE produced results with the highest MS-SSIM and PSNR results. During training, loss is computed on 75%, 93.75%, 98.44%, and 99.61% compression results. The total loss value is calculated as the sum of the losses for each compression result.

When measuring the compression performance, the metric bits per pixel (bpp) is used. This is calculated by dividing the size of the compressed image in bits by the number of pixels in the original image. Raw RGB images are 24 bpp since they use one byte (eight bits) per color channel per pixel and thus 24 bits (or three bytes) per pixel.

In the image compression literature, the common metric of comparison is multi-scale SSIM (MS-SSIM)[40]. MS-SSIM is designed as an improvement over SSIM by taking into account the human visual perspective at various distances. The standard SSIM algorithm

assumes a constant viewing size relative to the distance of the viewer (i.e., it is assumed the image being compared is always the same size). However, in common applications this is not true as images can sometimes be small on a screen or a viewer can be further away. MS-SSIM takes this into account by calculating SSIM results at differing scales (typically 2, 4, 8, and 16 times downscaled), weighting them individually, and summing together to produce a single result. I use this MS-SSIM metric alongisde PSNR on the Y-channel of the YCbCr color space to evaluate the results of the model.

5.5. Results

The static compression ratios of the autoencoder on 24-bit inputs are 6, 1.5, and 0.375 bpp. This is further reduced using the bit-plane quantization and adaptive arithmetic coding (AAC). Table 5.1 shows the effect of AAC on reducing the bpp. The percentage in reduction remains fairly linear across all three models with an average of 14.8% reduction in bpp after using AAC.

TABLE 5.1. Bits per pixel results on the Kodak set after implementing adaptive arithmetic coding.

Model	Bits per Pixel
TinyCompress-4	5.1043
TinyCompress-16	1.2719
TinyCompress-64	0.3216

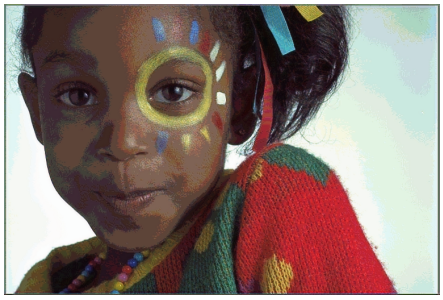
Figure 5.3 shows the visual effects of using bit-plane quantization from the full eight-bits down to one on the results from TinyCompress-4. It is important here to note that, since the quantization is being done at the compressed image level, this isn't quantizing the number of colors at the reconstruction level. This is why, in the one bit plane case, there are more than eight colors (two per channel for a total of eight unique combinations). There is very little noticeable quality loss until the quantization reaches five bits when some color banding begins to appear. However, this result is dependent on the image as some more complex images begin to show degradation after removing only one bit plane and should be used as an extra knob to turn for compression on the user input side. The degradation also increases with higher compression ratios using TinyCompress-16 and -64. When using

(A) 8 bit planes.
0.9913/39.6596/5.2862

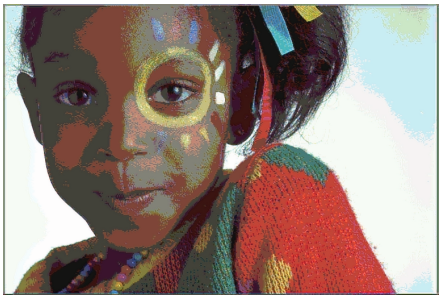
(B) 7 bit planes.
0.9957/39.0109/4.5383

(C) 6 bit planes.
0.9941/36.8899/3.7905

(D) 5 bit planes.
0.9877/32.6625/3.0464



(E) 4 bit planes.
0.9594/27.0897/2.3132



(F) 3 bit planes.
0.8666/20.9543/1.5867

(G) 2 bit planes.
0.7101/15.0303/1.0825

(H) 1 bit plane.
0.6353/11.2548/0.5684

FIGURE 5.3. Effects of bit-plane quantization on MS-SSIM PSNR, and bits per pixel using TinyCompress-4.

(A) Original

(B) TinyCompress-4:
0.9913/35.4868/5.2514

(C) TinyCompress-16:
0.9749/30.2987/1.4000

(D) TinyCompress-64:
0.9387/27.4564/0.3516

FIGURE 5.4. MS-SSIM, PSNR, and bits per pixel results for an image from kodak using TinyCompress-4, -16, and -64.

TABLE 5.2. MS-SSIM and PSNR values for TinyCompress-4, -16, and -64 on Kodak.

Model	MS-SSIM	PSNR
TinyCompress-4	0.9929	37.195
TinyCompress-16	0.9641	29.620
TinyCompress-64	0.8902	26.457

TinyCompress-64, almost all images will show some form of degradation after quantizing only one bit-plane since it has already compressed the information significantly.

Table 5.2 shows the MS-SSIM and PSNR average results on the Kodak set for each of TinyCompress-4, -16, and -64 without any bit-plane quantization. As expected, the drop-off in results follows closely with the compression increase. Figure 5.4 and Figure 5.5 show example image results from the Kodak set with each of the three models compared with the original image. These images show that in general the color accuracy of the images is maintained and the general structure of the images is not degraded significantly. There is a

(A) Original

(B) TinyCompress-4:
0.9961/39.6596/5.2862

(C) TinyCompress-16:
0.9819/31.9334/1.2758

(D) TinyCompress-64:
0.9387/28.5576/0.3350

FIGURE 5.5. MS-SSIM, PSNR, and bits per pixel results for an image from Kodak using TinyCompress-4, -16, and -64.

very noticeable decline in image sharpness when going to TinyCompress-64. However, one may notice that when viewing images at a distance without zooming in, the images do look relatively similar without much noticeable loss. This is where the MS-SSIM metric is useful. The somewhat high MS-SSIM results in the table show that there is not much loss at a

TABLE 5.3. Execution time in milliseconds for encoding and decoding only using GPU and CPU.

Model	Encoding-GPU	Encoding-CPU	Decoding-GPU	Decoding-CPU
TinyCompress-4	2.7	29.7	1.5	18.4
TinyCompress-16	4.4	39.2	2.7	23.5
TinyCompress-64	6.0	42.4	3.6	26.3

further viewing distance and that significantly quality loss is only apparent when moving closer, or zooming, in.

Since this model is designed to be efficient, it is also important to look at the execution time performance. Table 5.3 shows the execution time results for the deep encoding and decoding models on both GPU and CPU. These times omit the AAC performance that is very unoptimized to focus on the models themselves. Additionally, the times are calculated using Python outside of the TensorFlow calls and thus there is likely some significant overhead (including data transmission to and from the GPU) incurred that is unrelated to the actual encoding and decoding processes. The GPU times compare well with the results from [41], one of the fastest in literature, whose GPU results for encoding and decoding are at minimum 8.6ms and 9.9ms.

Figure 5.6 shows a comparison between the three models and the JPEG results at similar bpp levels. The JPEG images were generated using the Python PIL package and by setting the JPEG quality attribute to adjust the bpp levels. While the JPEG images show higher metric values, the first two images show practically no differences in quality. For the highest compression, the JPEG results show overall better structure with stronger edges, however the color accuracy is noticeably worse compared with TinyCompress.

(A) 0.9959/40.0778/4.8578

(B) JPEG: 0.9975/51.8739/5.1224

(C) 0.9759/31.6855/0.9909

(D) JPEG: 0.9881/39.8930/1.0562

(E) 0.9228/27.6528/0.3075

(F) JPEG: 0.9277/32.6283/0.3145

FIGURE 5.6. TinyCompress-4, -16, and -64 (left) comparisons with JPEG (right) showing MS-SSIM, PSNR, and bpp metrics.

CHAPTER 6

CONCLUSION

In this work, the single-image super-resolution problem domain is explored and a **Tiny P**ixel **S**huffling **S**uper-**R**esolution (TinyPSSR) model is proposed. To allow for edge device application, focus is placed on reducing the total memory requirements and the computational complexity. This is implemented by keeping the number of filters at each convolutional layer low and making use of depthwise convolution layers to reduce the number of parameters. Finally, a pixel shuffling layer at the end of the model is used to super resolve the low-resolution image. With only 1,356 parameters, the perceptual quality of the images significantly outperforms traditional bicubic upsampling and is similar in quality to the 3,937 parameter FSRCNN-s model with a 3x speedup when testing on the traditional Set5, Set14, Urban100, and BSD100 testing sets at 2x and 4x super-resolution. The traditional model upscales inputs by 2x and uses iterations to perform successive upscaling operations. A direct 4x upscaling model is also designed with twice the parameters (2,716) that significantly outperforms FSRCNN-s and performs nearly on par with other models nearly 6x times the size. Finally, TinyPSSR is expanded to super-resolve infrared thermal images and its efficacy is tested through simulations on the FLIR and OSU pedestrian thermal datasets to show improvements over the baseline bicubic operation.

Efficient convolutional neural network techniques is also applied to the deep image compression problem and an iterative model, TinyCompress, is proposed to perform efficient image compression with 6,498 parameters per 2x, or 75%, compression. The model is tested at three static compression rates – 75%, 93.75%, and 98.44% – and combined with bit-plane quantization and adaptive arithmetic coding to further reduce the bitrates to as low as 0.3 bpp while maintaining acceptable quality. Quality metrics and visual comparisons show that the results compare closely with JPEG and performance comparisons are on par with state-of-the-art models.

Appendix A

TINYPSSR - RESULTS

A.1. SET5

A.1.1. Upscale 2x

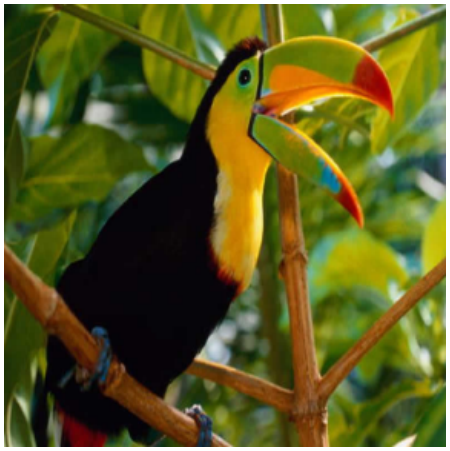
(A) HR

(B) TinyPSSR-2: 0.9621/36.99

(C) Bicubic: 0.9491/35.86

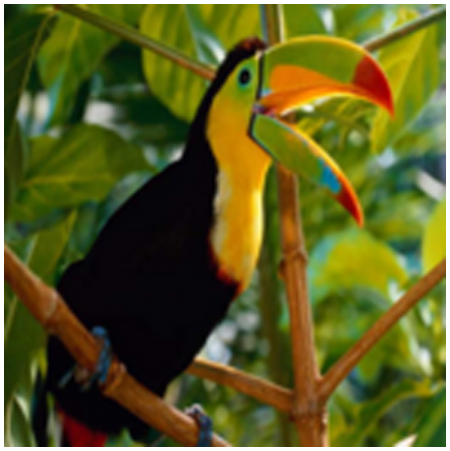
(D) LR

FIGURE A.1. The "baby" image from Set5 with 2x scale reconstructions and SSIM/PSNR metric values.

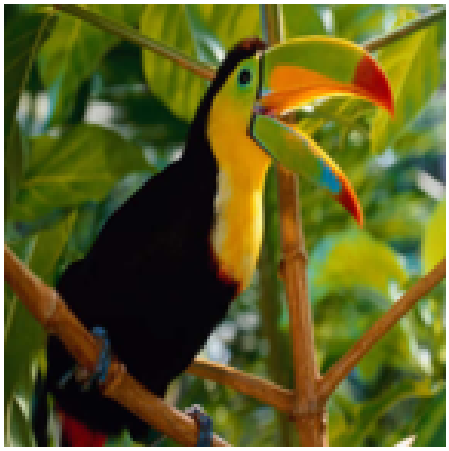


(A) HR

(B) TinyPSSR-2: 0.9824/38.47



(C) Bicubic: 0.9454/31.51



(D) LR

FIGURE A.2. The "bird" image from Set5 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.9530/30.30

(C) Bicubic: 0.9158/27.46

(D) LR

FIGURE A.3. The "butterfly" image from Set5 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8829/35.36

(C) Bicubic: 0.8639/34.45

(D) LR

FIGURE A.4. The "head" image from Set5 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.9636/33.64

(C) Bicubic: 0.9390/30.54

(D) LR

FIGURE A.5. The "woman" image from Set5 with 2x scale reconstructions and SSIM/PSNR metric values.

A.1.2. Upscale 4x

(A) HR

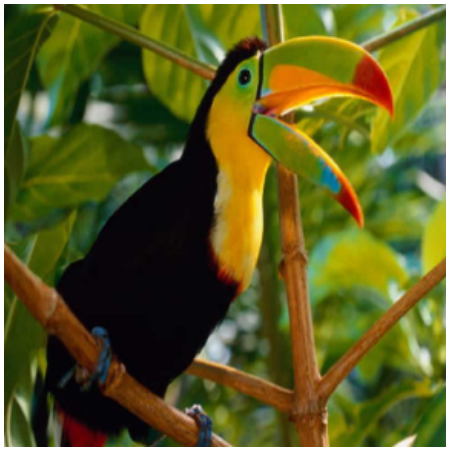
(B) TinyPSSR-2: 0.8825/31.34

(C) TinyPSSR-4: 0.8856/31.80

(D) Bicubic: 0.8439/27.92

(E) LR

FIGURE A.6. The "baby" image from Set5 with 4x scale reconstructions and SSIM/PSNR metric values.

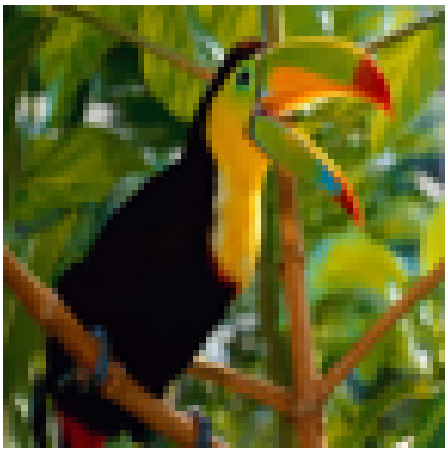


(A) HR

(B) TinyPSSR-2: 0.9060/30.82

(C) TinyPSSR-4: 0.9107/30.93

(D) Bicubic: 0.8043/24.53



(E) LR

FIGURE A.7. The "bird" image from Set5 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8099/23.52

(C) TinyPSSR-4: 0.8381/24.04

(D) Bicubic: 0.7329/22.20

(E) LR

FIGURE A.8. The "butterfly" image from Set5 with 4x scale reconstructions and SSIM/PSNR metric values.

A.2. SET14

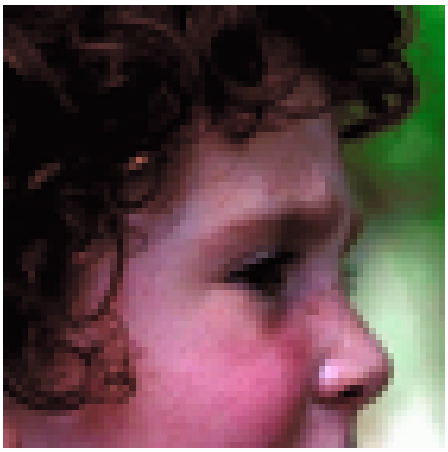
A.2.1. Upscale 2x

(A) HR

(B) TinyPSSR-2: 0.7820/31.79

(C) TinyPSSR-4: 0.7853/32.01

(D) Bicubic: 0.7550/31.07



(E) LR

FIGURE A.9. The "head" image from Set5 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8678/27.19

(C) TinyPSSR-4: 0.8833/27.71

(D) Bicubic: 0.8091/24.61

(E) LR

FIGURE A.10. The "woman" image from Set5 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.7619/25.36

(C) Bicubic: 0.7069/24.74

(D) LR

FIGURE A.11. The "mandrill" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8706/28.48

(C) Bicubic: 0.8450/27.93

(D) LR

FIGURE A.12. The "barbara" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8433/27.27

(C) Bicubic: 0.8019/26.66

(D) LR

FIGURE A.13. The "bridge" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8414/30.22

(C) Bicubic: 0.7871/22.87

(D) LR

FIGURE A.14. The "coast guard" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.9057/27.70

(C) Bicubic: 0.7956/22.43

(D) LR

FIGURE A.15. The "comic" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8828/35.33

(C) Bicubic: 0.8638/34.46

(D) LR

FIGURE A.16. The "face" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

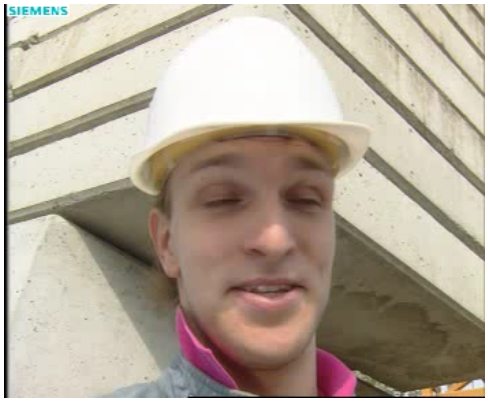
(A) HR

(B) TinyPSSR-2: 0.9297/32.24

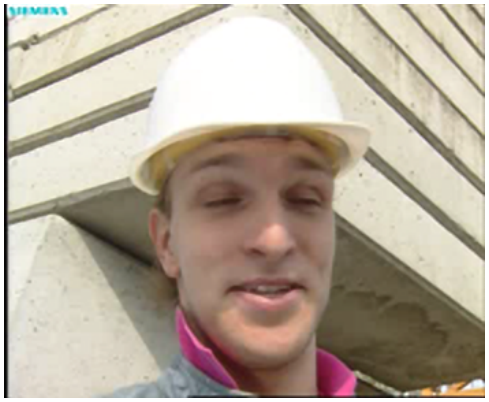
(C) Bicubic: 0.8839/26.31

(D) LR

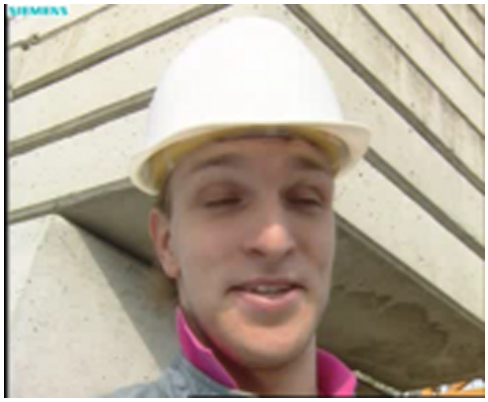
FIGURE A.17. The "flowers" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.



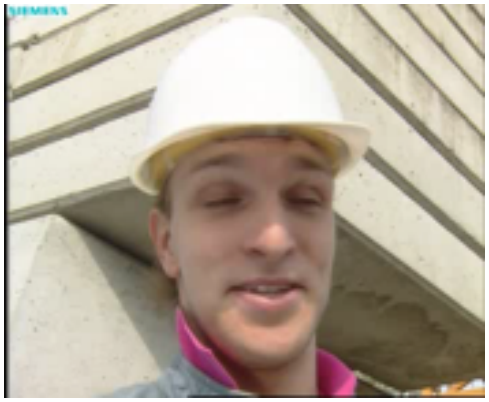
(A) HR



(B) TinyPSSR-2: 0.9637/33.79



(C) Bicubic: 0.9430/30.26

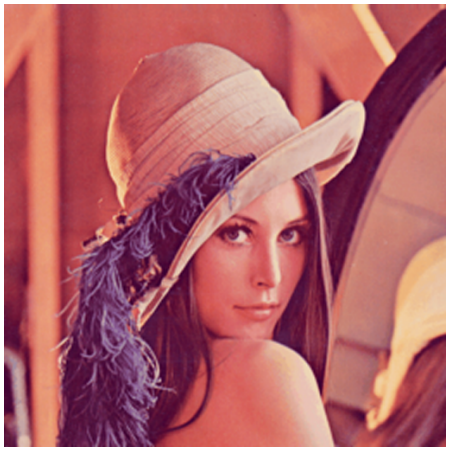


(D) LR

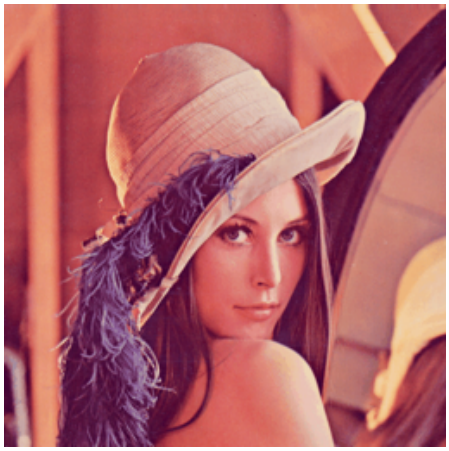
FIGURE A.18. The "foreman" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.9258/35.62



(C) Bicubic: 0.9078/33.11



(D) LR

FIGURE A.19. The "lenna" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8804/30.35

(C) Bicubic: 0.7842/21.61

(D) LR

FIGURE A.20. The "man" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.9721/35.45

(C) Bicubic: 0.9603/33.00

(D) LR

FIGURE A.21. The "monarch" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

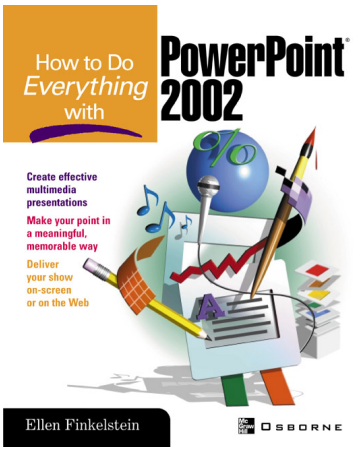
(A) HR

(B) TinyPSSR-2: 0.9180/34.12

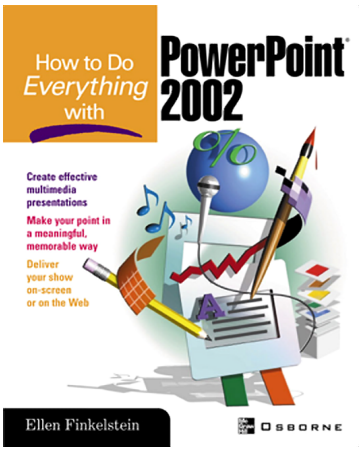
(C) Bicubic: 0.8542/24.62

(D) LR

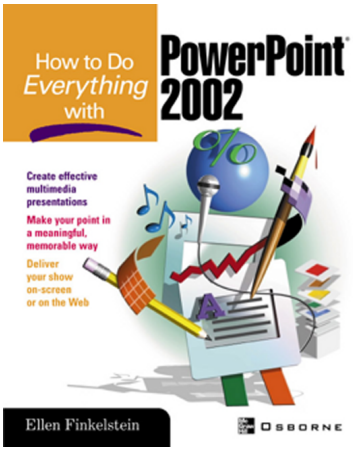
FIGURE A.22. The "pepper" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.



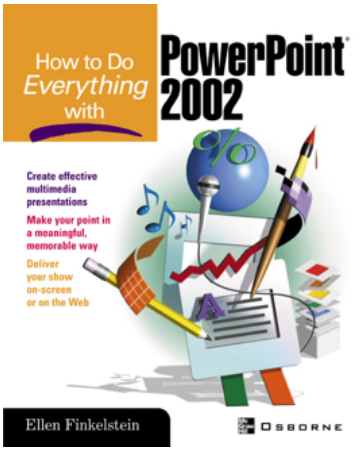
(A) HR



(B) TinyPSSR-2: 0.9726/29.34

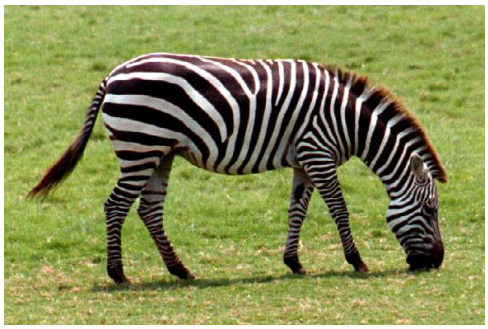


(C) Bicubic: 0.7486/14.45



(D) LR

FIGURE A.23. The "ppt3" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.



(A) HR

(B) TinyPSSR-2: 0.9369/32.57

(C) Bicubic: 0.7953/18.41

(D) LR

FIGURE A.24. The "zebra" image from Set14 with 2x scale reconstructions and SSIM/PSNR metric values.

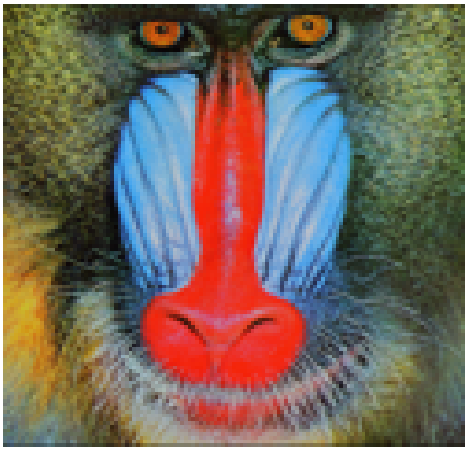
A.2.2. Upscale 4x

(A) HR

(B) TinyPSSR-2: 0.5103/22.44

(C) TinyPSSR-4: 0.5254/22.36

(D) Bicubic: 0.4572/22.28



(E) LR

FIGURE A.25. The "mandrill" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.7294/25.46

(C) TinyPSSR-4: 0.7361/25.37

(D) Bicubic: 0.6899/25.10

(E) LR

FIGURE A.26. The "barbara" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.6007/23.30

(C) TinyPSSR-4: 0.6108/23.12

(D) Bicubic: 0.5452/23.15

(E) LR

FIGURE A.27. The "bridge" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.5643/25.68

(C) TinyPSSR-4: 0.5701/25.77

(D) Bicubic: 0.5246/21.83

(E) LR

FIGURE A.28. The "coast guard" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.6625/22.24

(C) TinyPSSR-4: 0.6793/22.22

(D) Bicubic: 0.5460/20.21

(E) LR

FIGURE A.29. The "comic" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.7799/31.74

(C) TinyPSSR-4: 0.7843/31.99

(D) Bicubic: 0.7504/30.46

(E) LR

FIGURE A.30. The "face" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

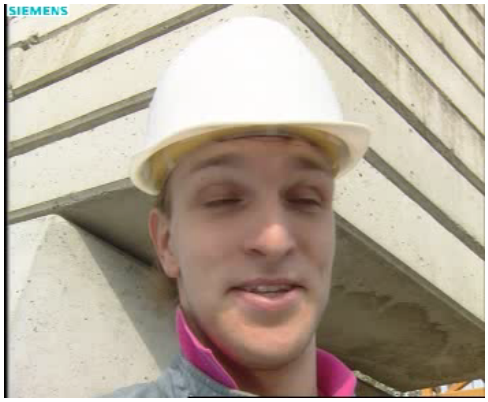
(B) TinyPSSR-2: 0.7764/26.52

(C) TinyPSSR-4: 0.7833/26.49

(D) Bicubic: 0.7134/24.62

(E) LR

FIGURE A.31. The "flowers" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

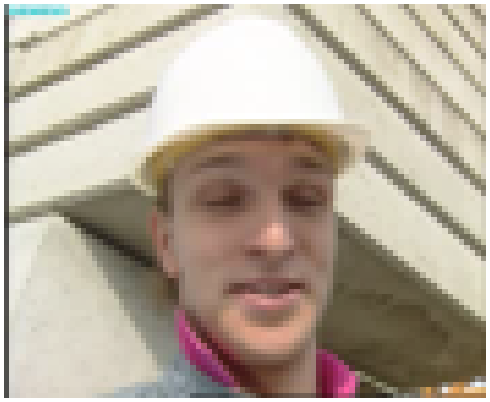


(A) HR

(B) TinyPSSR-2: 0.8857/27.72

(C) TinyPSSR-4: 0.8956/28.50

(D) Bicubic: 0.8464/26.1



(E) LR

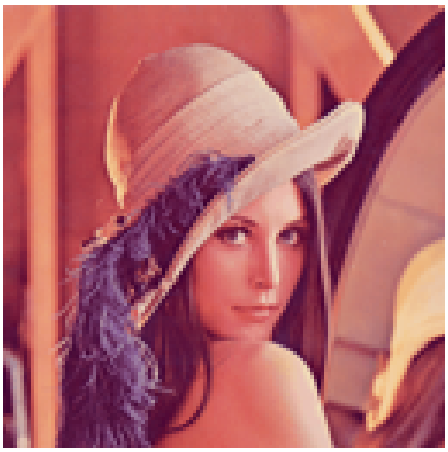
FIGURE A.32. The "foreman" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.8425/30.20

(C) TinyPSSR-4: 0.8494/30.72

(D) Bicubic: 0.8135/29.26



(E) LR

FIGURE A.33. The "lenna" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.7258/26.27

(C) TinyPSSR-4: 0.7349/26.38

(D) Bicubic: 0.6362/20.02

(E) LR

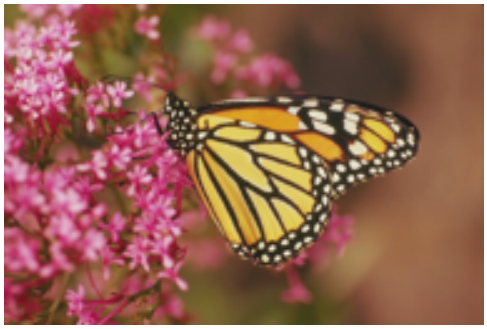
FIGURE A.34. The "man" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

(B) TinyPSSR-2: 0.9075/28.65

(C) TinyPSSR-4: 0.9141/28.93

(D) Bicubic: 0.8831/27.59



(E) LR

FIGURE A.35. The "monarch" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

(A) HR

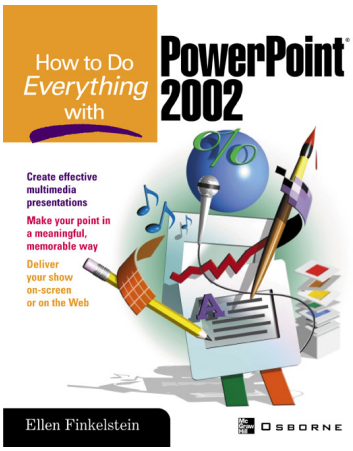
(B) TinyPSSR-2: 0.8569/29.65

(C) TinyPSSR-4: 0.8646/30.41

(D) Bicubic: 0.7852/22.19

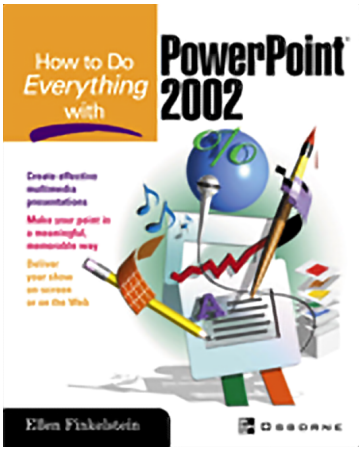
(E) LR

FIGURE A.36. The "pepper" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.



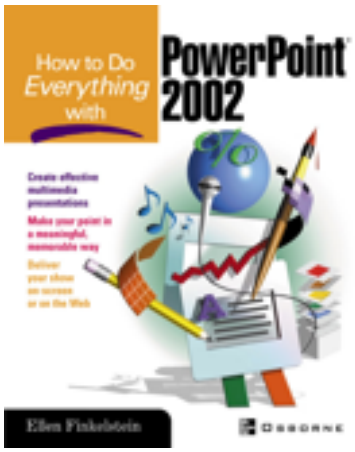
(A) HR

(B) TinyPSSR-2: 0.8794/23.41



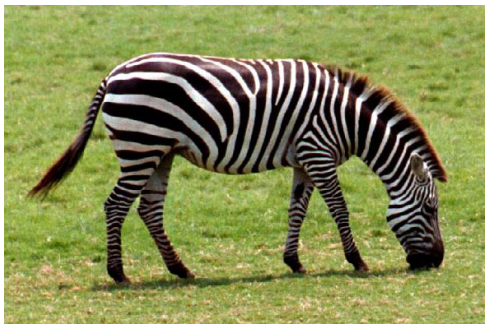
(C) TinyPSSR-4: 0.8926/23.90

(D) Bicubic: 0.6637/13.36



(E) LR

FIGURE A.37. The "ppt3" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

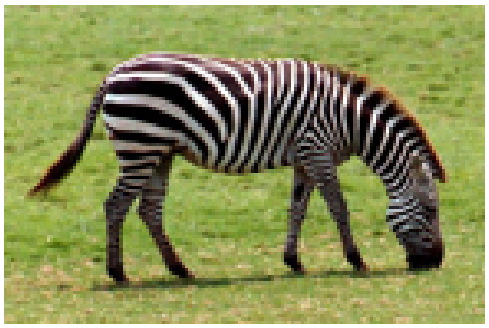


(A) HR

(B) TinyPSSR-2: 0.7514/25.43

(C) TinyPSSR-4: 0.7600/25.54

(D) Bicubic: 0.6290/17.69



(E) LR

FIGURE A.38. The "zebra" image from Set14 with 4x scale reconstructions and SSIM/PSNR metric values.

REFERENCES

[1] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, pp. 115–133, 12 1943.

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

[3] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[4] "Convolutional neural networks (cnns / convnets)." [Online]. Available: https://cs231n.github.io/convolutional-networks/

[5] M. Gustineli, "A survey on recently proposed activation functions for deep learning," 2022. [Online]. Available: https://arxiv.org/abs/2204.02921

[6] M. M. Noel, A. L, A. Trivedi, and P. Dutta, "Growing cosine unit: A novel oscillatory activation function that can speedup training and reduce parameters in convolutional neural networks," 2021. [Online]. Available: https://arxiv.org/abs/2108.12943

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

[8] P. Warden and D. Situnayake, *TinyML*. O'Reilly Media, Incorporated, 2019.

[9] P. P. Ray, "A review on tinyml: State-of-the-art and prospects," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 4, pp. 1595–1623, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157821003335

[10] K. Nazeri, H. Thasarathan, and M. Ebrahimi, "Edge-informed single image super-resolution," in *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, 2019, pp. 3275–3284.

[11] S. Anwar and N. Barnes, "Densely residual laplacian super-resolution," *IEEE Trans-*

actions on Pattern Analysis and Machine Intelligence, vol. 44, no. 3, pp. 1192–1204, 2022.

[12] B. Niu, W. Wen, W. Ren, X. Zhang, L. Yang, S. Wang, K. Zhang, X. Cao, and H. Shen, "Single image super-resolution via a holistic attention network," in *Computer Vision – ECCV 2020*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds. Cham: Springer International Publishing, 2020, pp. 191–207.

[13] J. Liang, J. Cao, G. Sun, K. Zhang, L. Van Gool, and R. Timofte, "Swinir: Image restoration using swin transformer," in *2021 IEEE/CVF International Conference on Computer Vision Workshops (ICCVW)*, 2021, pp. 1833–1844.

[14] J. Li, Z. Pei, and T. Zeng, "From beginner to master: A survey for deep learning-based single-image super-resolution," *ArXiv*, vol. abs/2109.14335, 2021.

[15] C. Dong, C. C. Loy, K. He, and X. Tang, "Learning a deep convolutional network for image super-resolution," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 184–199.

[16] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," *CoRR*, vol. abs/1608.00367, 2016. [Online]. Available: http://arxiv.org/abs/1608.00367

[17] 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 subpixel convolutional neural network," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 1874–1883.

[18] S. Anwar, S. Khan, and N. Barnes, "A deep journey into super-resolution: A survey," 2019. [Online]. Available: https://arxiv.org/abs/1904.07523

[19] J. Kim, J. K. Lee, and K. M. Lee, "Accurate image super-resolution using very deep convolutional networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 1646–1654.

[20] S. Hwang, G. Yu, C. Jung, and J. Kim, "Attention-aware linear depthwise

convolution for single image super-resolution," 2019. [Online]. Available: https://arxiv.org/abs/1908.02648

[21] W. Shi, J. Caballero, L. Theis, F. Huszar, A. P. Aitken, C. Ledig, and Z. Wang, "Is the deconvolution layer the same as a convolutional layer?" *CoRR*, vol. abs/1609.07009, 2016. [Online]. Available: http://arxiv.org/abs/1609.07009

[22] S. Zhang, G. Liang, S. Pan, and L. Zheng, "A fast medical image super resolution method based on deep learning network," *IEEE Access*, vol. 7, pp. 12 319–12 327, 2019.

[23] M. Bevilacqua, A. Roumy, C. Guillemot, and M.-L. Alberi-Morel, "Low-complexity single image super-resolution based on nonnegative neighbor embedding," 09 2012.

[24] R. Zeyde, M. Elad, and M. Protter, "On single image scale-up using sparse-representations," in *Curves and Surfaces*, J.-D. Boissonnat, P. Chenin, A. Cohen, C. Gout, T. Lyche, M.-L. Mazure, and L. Schumaker, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 711–730.

[25] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, vol. 2, 2001, pp. 416–423 vol.2.

[26] J.-B. Huang, A. Singh, and N. Ahuja, "Single image super-resolution from transformed self-exemplars," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 5197–5206.

[27] 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.

[28] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[29] R. Timofte, E. Agustsson, L. V. Gool, M.-H. Yang, L. Zhang, B. Lim, S. Son, H. Kim,

S. Nah, K. M. Lee, X. Wang, Y. Tian, K. Yu, Y. Zhang, S. Wu, C. Dong, L. Lin, Y. Qiao, C. C. Loy, W. Bae, J. Yoo, Y. Han, J. C. Ye, J.-S. Choi, M. Kim, Y. Fan, J. Yu, W. Han, D. Liu, H. Yu, Z. Wang, H. Shi, X. Wang, T. S. Huang, Y. Chen, K. Zhang, W. Zuo, Z. Tang, L. Luo, S. Li, M. Fu, L. Cao, W. Heng, G. Bui, T. Le, Y. Duan, D. Tao, R. Wang, X. Lin, J. Pang, J. Xu, Y. Zhao, X. Xu, J. Pan, D. Sun, Y. Zhang, X. Song, Y. Dai, X. Qin, X.-P. Huynh, T. Guo, H. S. Mousavi, T. H. Vu, V. Monga, C. Cruz, K. Egiazarian, V. Katkovnik, R. Mehta, A. K. Jain, A. Agarwalla, C. V. S. Praveen, R. Zhou, H. Wen, C. Zhu, Z. Xia, Z. Wang, and Q. Guo, "Ntire 2017 challenge on single image super-resolution: Methods and results," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2017, pp. 1110–1121.

[30] M. S. M. Sajjadi, B. Schölkopf, and M. Hirsch, "Enhancenet: Single image super-resolution through automated texture synthesis," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 4501–4510.

[31] J. Gu, H. Lu, W. Zuo, and C. Dong, "Blind super-resolution with iterative kernel correction," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 1604–1613.

[32] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[33] "Free teledyne flir thermal dataset for algorithm training," *FREE - FLIR Thermal Dataset for Algorithm Training — Teledyne FLIR*. [Online]. Available: https://www.flir.in/oem/adas/adas-dataset-form/

[34] J. W. Davis, "Otcbvs benchmark dataset collection," *OTCBVS*. [Online]. Available: https://vcipl-okstate.org/pbvs/bench/

[35] J.-B. Huang, A. Singh, and N. Ahuja, "Single image super-resolution from transformed self-exemplars," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5197–5206.

[36] J.-B. Huang, "Single image super-resolution from transformed self-exemplars," *GitHub*. [Online]. Available: https://github.com/jbhuang0604/SelfExSR

[37] S. Jamil, M. J. Piran, and MuhibUrRahman, "Learning-driven lossy image compression; a comprehensive survey," 2022. [Online]. Available: https://arxiv.org/abs/2201.09240

[38] L. Theis, W. Shi, A. Cunningham, and F. Huszár, "Lossy image compression with compressive autoencoders," 2017. [Online]. Available: https://arxiv.org/abs/1703.00395

[39] R. W. Franzen, "Kodak lossless true color image suite," *True Color Kodak Images*. [Online]. Available: https://r0k.us/graphics/kodak/

[40] Z. Wang, E. Simoncelli, and A. Bovik, "Multiscale structural similarity for image quality assessment," in *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, vol. 2, 2003, pp. 1398–1402 Vol.2.

[41] O. Rippel and L. Bourdev, "Real-time adaptive image compression," 2017. [Online]. Available: https://arxiv.org/abs/1705.05823