Imperial College London
Department of Computing

Machine Learning For Filtering Monte Carlo Noise In Ray Traced Images

Author: Zichen Liu

Supervisor: Dr Su-Lin Lee

June 21, 2017

Abstract

Ray tracing is a computer graphics method for creating photorealistic images and animation for games and film. Problems arise when the scene to be generated is highly detailed with many objects - these jobs can be hugely computationally expensive to render with this method. We will attempt to employ a machine learning algorithm to reduce the amount of time it requires to render an image. By cleaning up artefacts in fast, low quality renders to produce the end result, we will reduce the total pipeline time by 97%.

Our work results in a machine learning method that takes rough renders as input and performs artefact removal and detail preservation. We will document the training and evaluation of our evolving models during the research phase - the Single Layer Perceptron*, the* Convolutional Neural Network *and the* Deconvolutional Neural Network*; as well as scrutinizing our best architecture - the* Inception Module Network*.*

We then deploy our most successful model to the internet as a web application that allows access to the neural network via the internet, allowing clients to execute the network on their own examples.

Acknowledgements

I would like to thank

Dr Su-Lin Lee,

for her consistent support of the project and truly insightful comments and suggestions throughout,

Lloyd Karmara of the Computing Support Group(CSG),

for his help in allocating the resources I needed to complete this project efficiently,

my friend Zukang Liao of the Intelligent Behaviour Understanding Group (iBug),

for providing me the knowledge and assistance with understanding neural networks even before this project had started,

my friend Peng Peng,

for his friendship over the past 3 years and our deep and meaningful exchanges on the future of Artificial Intelligence,

my friend Winnie Wong,

for a valuable second opinion in the qualitative evaluation stage,

and My Parents,

for the world.

Contents

1 Introduction

The year is 1995 and Toy Story, which is the world's first feature length computer animated film, had just been released[1]. Over the next few weeks, the film went on to receive universal critical acclaim, creating a lasting impression with reviewers and filmgoers alike. For many people, this was the first time they had been sold on the impact and relevance of computer graphics as a platform for entertainment. At the centre of it all, PIXAR, the creator, had proven the worth of computer generated graphics among entrepreneurs, paving the way for a new industry.

Figure 1: Toy Story[2].

Computer graphics has a rich and diverse set of applications - architectural illustrations, simulations, computer aided design and entertainment. The 3d content creator community spans a wide range of professions including engineers, architects, entertainment artists and technical artists. As the love for artificial graphics permeate the masses, growing demand for ever more realistic and detailed scenes have surfaced. It is now all too common for us to mistake computer generated images for real ones. Over the years, computer hardware for processing graphics have evolved too - what used to be a subsidiary of the motherboard or CPU is now available as a large power hungry dedicated graphics processing unit with its own memory and power supply. Faced with ever growing demand for more vibrant and lifelike graphics, we must take advantage of the latest in technology to advance our graphics software. This will be the setting for our project.

Ray tracing emerged as one of the two main ways of creating computer graphics. It is often favoured over the traditional rendering pipeline for its potential to achieve superior photorealism through unbiased simulations of light behaviour in the real world, producing virtually photorealistic images. However these benefits come at the expense of significantly more computation time or resources. It is not unusual for one frame to take around 2 hours to render on a top of the range graphics card, and considering that films are rendered at 24 frames per second, there is a huge potential for the computation time to blow up. Nevertheless, this technique has entered many production environments such as Quake Wars: Ray Traced[3], a ray traced 3d shooter, and many of PIXAR's more recent films such as Cars[4].

1.1 Objectives

Most ray tracing methods use progressive refinement, that is, when we instruct a scene to be rendered from a simulated camera, we get an output right from the start. As time progresses, the renderer samples the scene more finely and as a result we receive progressively more refined outputs that slowly tends to the final output. We use the number of samples as a quantitative measure of how much refinement a scene has received. The higher this number, the more accurate

the output as a representation of the scene. With this unique ability to gain information about the entire scene right from the beginning we can choose when to stop the rendering process after enough detail and clarity has been obtained in the render.

Our goal would be to create a machine learning system that can take a low sample image as input, and output a result that is very similar to a high sample image. We would like this system to be fast, be general, be able to filter renders of all types and lighting scenarios, and be easily deployed to inexpensive hardware. This essentially means that we would slash the computational time by some defined proportion. Given that a low sample image with most of the details preserved only takes around 8-10% of the time it takes a high sample image to render, we would be saving a huge amount of time, especially if we are previewing an animation. Not only would this save the creator time and electricity, we also save the opportunity cost associated with high render times causing humans to idle. For example, artists frequently need to re-render an image to review the modifications they have made in a scene, the shorter the wait time, the more times he can adjust the scene to his artistic desires. Finally, it would also be great if this system could be presented to and used by the people who are actively engaged in this field. For this we certainly want to encapsulate all the machine learning details away so that the user has a simple interface to submit a job and retrieve the result.

1.2 Challenges

A neural network is an inherently hard construct to understand. The input layers deliver your data to an emporium of generic structures - the hidden layers are riddled with endless nodes and weights whose meaning seems to be concealed within. Intuition on the efficiency of particular architectures for various tasks comes only with repeated experimentation. Throughout the project, I will work hard to understand and document my journey to rationalizing the results from these different architectures - which will help me to better pick new architectures.

I want to keep my filtering application as general as possible, this means catering for images with different magnitudes of noise. So, how could we train a neural network to filter images with heavy noise aggressively, whilst being efficient at resolving images with light noise i.e. being gentle with the intricate detail that is already present in the scene? Perhaps we need to train a number of neural structures, and a differentiator that would feed heavily noisy images into some structures whilst inhibiting other structures for images with low noise?

Another challenge is the difficulty of choosing an efficient error function, something I will go into detail in section 4.1.

1.3 Contributions

- A trained machine learning architecture for removing noise from a Monte Carlo low sample ray traced render, thereby reducing the amount of samples and by extension, the amount time, required to create the same end product. Chapter 4.
- A load balanced, extensible web app architecture for deploying this machine learning algorithm to the internet, allowing users with no knowledge of the machine learning framework to filter their images. Chapter 5.
- An implementation of the about web app in PHP, Node.js and frontend javascript. Access at www.airenderer.com. Chapter 5.

2 Background

We will present the core of the relevant topics and ideas used in this project. An insight into computer graphics and ray tracing will be useful for analysing noisy images and formulating methods to target specific problems. We will also explain some neural network constructs that will be used to model and train our network. Finally, we will survey the current state of the art to update ourselves with the most successful techniques now.

2.1 Creating our 3D World

In computer graphics, the world is commonly represented as a list of geometries, a list of lights and an active camera. These comprise the basic ingredients we need to define a mechanism to produce a render - a 2D representation of the world as seen by the camera.

Camera

A camera is represented by a set of parameters usually comprising of a 3d location, a direction and a focal length. With these parameters, we can calculate which portion of the 3d world is visible and evaluate the rendering equation for this portion in the appropriate direction. The camera will not affect the physical world with respect to lighting so will not appear in any render. There are many effects that are added to a camera such as depth of field calculations and clipping during pre-rendering, and vignettes and chromatic aberration in post production compositing.

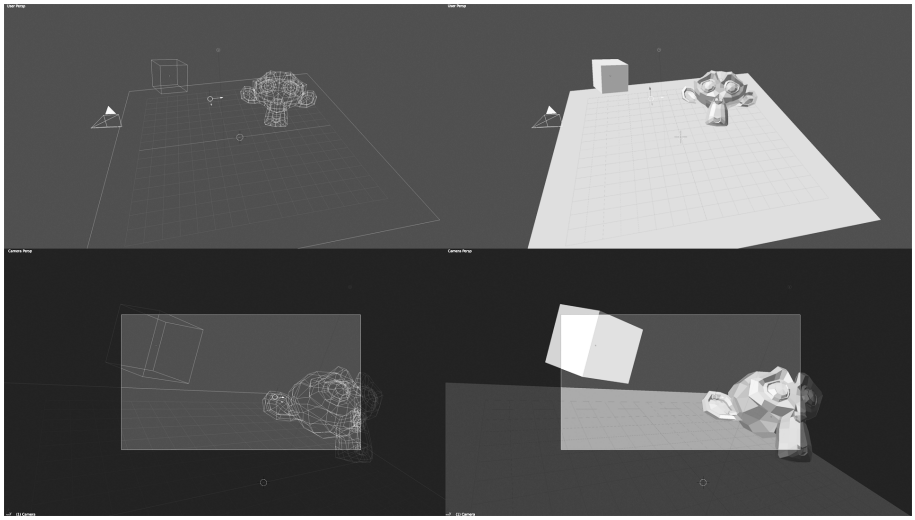

Figure 2: Example 3D Scene Specification[5].

Geometry

These are what the camera will see during the rendering process. Each piece of geometry is comprised of a list of vertices and method of connecting these into a face. Geometry fragments can also contain material data and other mapped data that may change its appearance. Large geometries are sometimes broken into triangles and quadrilaterals to simplify storage or manipulation. In figure 2, our example shows a monkey head made completely of triangles.

Lights

These objects emits light into the scene and are the only sources of light. For scenes where we do not consider secondary light bounces, the brightness at a point can be found by calculating the sum of the exposures of each light with respect to the point. There are two types of light - point and area sources. The occlusion of a point with respect to a point light source is easy, simply calculate whether there is an unobstructed vector from a point of occlusion to the source. These are usually added to the scene to increase the artificial brightness and not to be the main illuminators since point light sources are not realistic. Area sources are far more difficult to sample - for each point we wish to know the occlusion with respect to the source, we must calculate what percentage of the area light is occluded with respect to that point. The higher the percentage, the darker that point will appear.

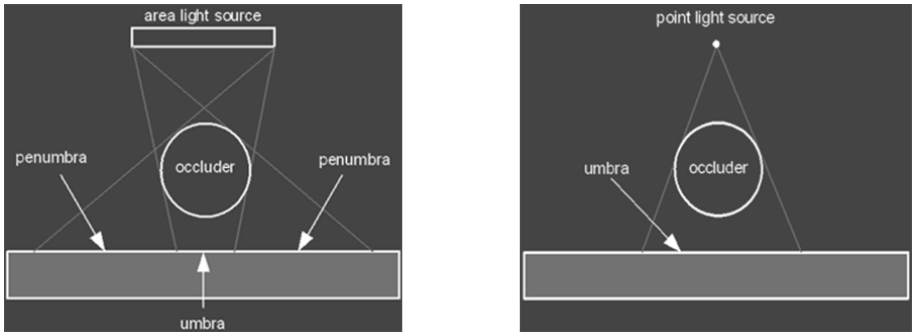

Figure 3: Area light vs Point light[6].

The Rendering Equation

So with the world set up, we just need a model of how light behaves in our sandbox. This model must accurately approximate how light behaves in the real world. It turns out that this may be represented as a single equation for the intensity of light at a given point and direction, known as the rendering equation[7][8][9].

$$L_0(x, \omega_0) = L_e(x, \omega_0) + \int_\Omega f_r(x, \omega_i, \omega_0)\, L_i(x, \omega_i)\, (\omega_i \cdot n)\, d\omega_i$$

where $\omega_i \in \Omega$.

All occurrences of L are triples of values representing red, green and blue.
$L_0(x, \omega_0)$ represents the total light leaving position x on a surface of the world in the direction of ω_0.
$L_e(x, \omega_0)$ represents the light created and emitted from position x on a surface of the world in the direction of ω_0. This is zero if the surface is not a light source.
$L_i(x, \omega_0)$ represents the light entering position x on a surface of the world from the direction of ω_0.
$f_r(x, \omega_i, \omega_0)$ represents the bidirectional reflectance relationship between the direction ω_i and ω_0) at the position x.
n represents the normal vector to the surface at position x.

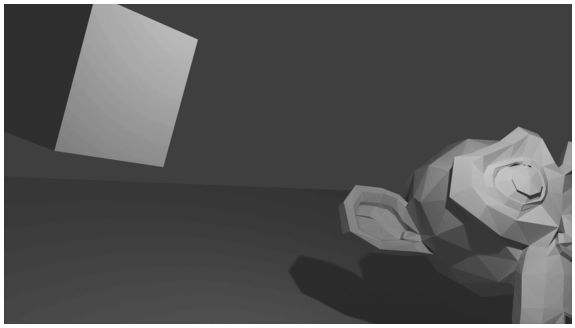

Figure 4: Example 3D Scene Rendered through the camera[5].

2.2 Monte Carlo Ray Tracing

Monte Carlo Ray Tracing is a technique to approximate the rendering equation without performing the explicit integral as well as to extend the ideas used in the rendering equation to transmission geometry. It simulates the behaviour of light in the real world to produce realistic images given a 3d scene. This technique is currently very popular and is supplied as the default production renderer for many 3d packages. It also progressively refines the render result - producing a clearer image the more samples it processes as shown in Figure 5 - a useful property that we will take advantage of later.

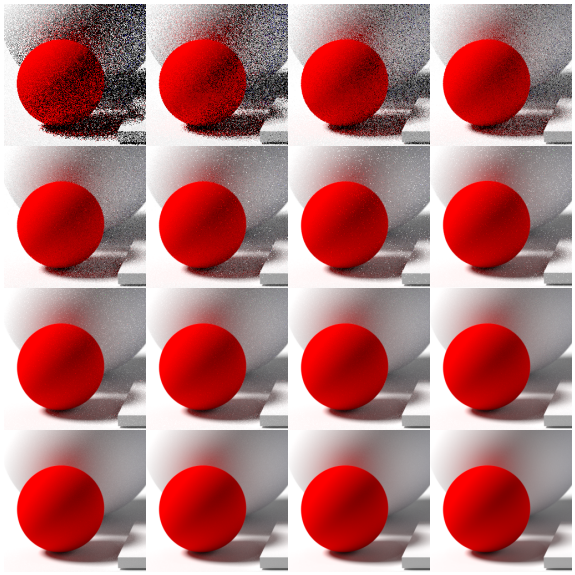

Figure 5: Progressive refinement of the path tracing technique [10].

2.3 The Tracing Algorithm

First of all we account for the pixels in the final image by projecting a vector ray into our scene for every pixel on our screen. For each ray, if it intersects a mesh in our scene then save that intersection, if not, we clip the ray and paint that pixel the background colour.

For each saved intersection point, we are interested in the true colour there. To approximate our integral in the rendering equation for our intersection, we shoot random rays of light from that particular point into the world and recursively calculate the colour intensities. Now we sum the colour intensities leaving the point of intersection in the direction of the camera for the final result. The recursion depth here is called the depth of the ray tracing algorithm.

Definition **Fireflies:** A spike in intensity often 1 pixel large that appears randomly in a low sample Monte Carlo ray trace of a scene. These appear because such as small number of rays have been shot from that intersection point, some of them just happened to hit something bright, causing a discontinuity of colour intensity. we can remove them by clamping the scene intensity but at the cost of realism.

Definition **Caustics:** Calculation of the interaction of light through glass type materials - effects such as reflection, refraction, total internal reflection and separation of light into its colour components must all be taken into account.

Definition **Subsurface Scattering:** Calculation of the interaction of light as it passes underneath of a surface of a diffuse material. This technique is used to create realistic skin, milk, etc.

Artefacts such as fireflies appear when the rendering equation is only partially evaluated and can be a prominent problem in a scene where only certain areas have a complex rendering equation. Caustics and Subsurface Scattering are the most common causes of a rendering equation that becomes difficult to evaluate. For example a diffuse scene with a large number of glass pieces in one corner would take very long to converge.

2.4 Artificial Neural Networks

2.4.1 Architecture

A neural network is an artificial construct that is designed to simulate real neurons in a human brain, approximating some function we want to apply to the input, to receive the desired output. The input data is serialized and inserted into the input layer nodes. The exact size of the input and output layer nodes should be suitably chosen for the task. Then for each of the nodes of the hidden layers and the output layer, we apply a weighted summation function over all the previous nodes, with the weights characteristic of each inter-node connection. It is these weights that we aim to find during the training process. Before the value leaves a node, we apply an activation function to the output to introduce a non-linearity to our multi-layer network. Our result will appear on the output layer.

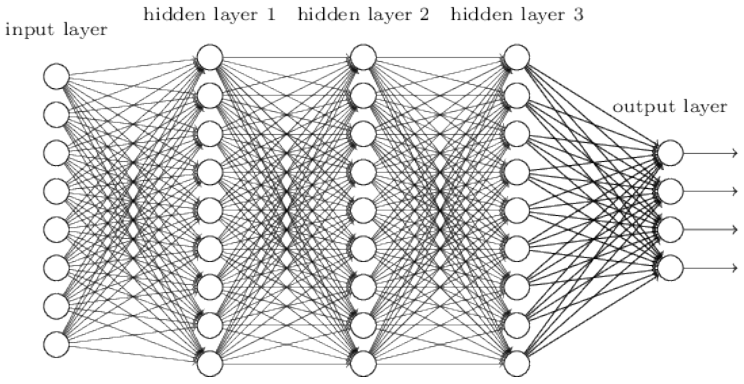

Figure 6: A Standard Multi-Layer Perceptron "Feed Forward" Neural Network[11].

Definition **Training Set:** A set of data comprising of data that should be fed into a neural network and the expected output of those data. The expected output will give us the error value used in backpropagation.

Definition **Ground Truth:** The expected output of the optimal neural network i.e. the actual result.

In supervised learning, when we train a neural network, we insert our training example at the

input nodes and retrieve the output. We then use our error function, which is a given measure of the difference between the output node and the ground truth, to evaluate the result. Our error function should be suitably picked to emphasise particular discrepancies that we want to minimize. Popular error functions are Euclidean distance and cross entropy. This error is then used to adjust weights to minimize the error.

The learning rate is the amount by which the weights of a neural network are adjusted with respect to the training examples. This is a hyperparameter of the neural network that must be tuned to ensure correct operation. We can also use a learning rate update schedule to decrease the learning rate as time goes on so that the corrective behaviour of backpropagation does not over-correct our finely tuned neural network in the final stages of training.

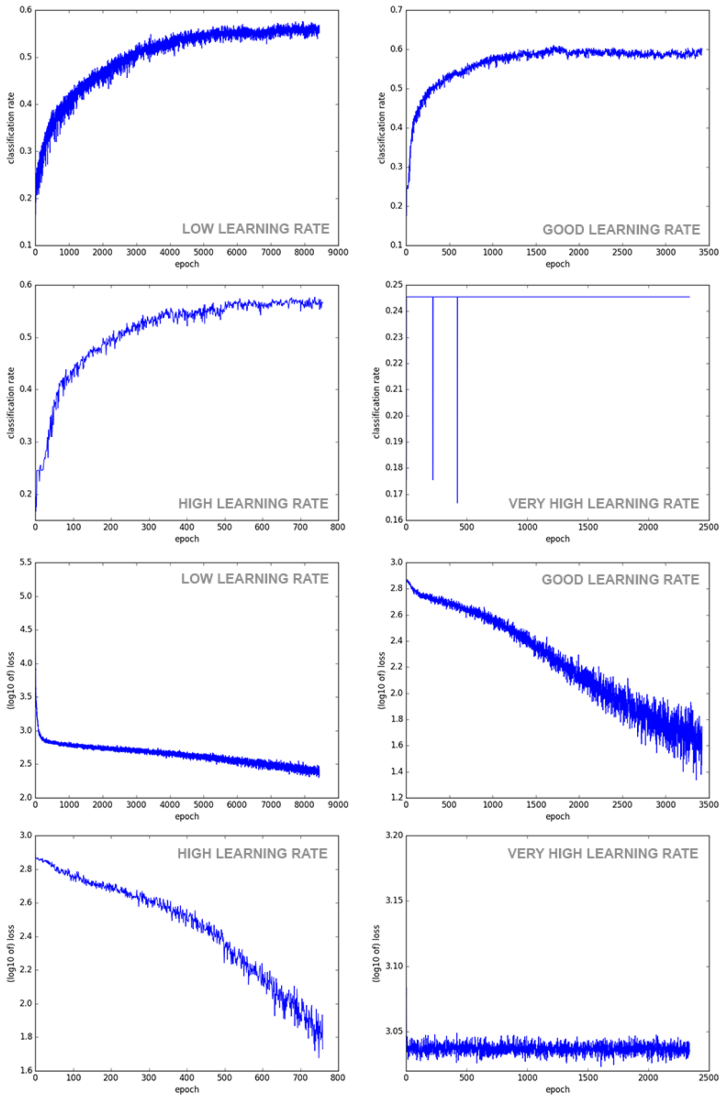

Figure 7: Effect of learning rate on convergence ability of a neural network[12].

***Definition* Epoch:** The step number of the current iteration.

***Definition* Training Loss:** The sum total of the amount each weight was adjusted by during the last training epoch.

Overfitting is a common phenomenon where the trained network is overly attached to your training set and fails to adapt to more general problems. One way of telling whether overfitting has occurred is by looking at whether training loss increases over a period of time. However, it has to be noted that the most definitive way to tell is to ensure that the quality of renders attained

at later epochs is better than the ones attained previously, figure 8 shows a training session with no overfitting from the loss-epoch curve. Dropout is a common technique to combat overfitting, particularly if you have a smaller training set. This dropout metric occludes a certain percentage of random selected elements of the training set from each epoch of training, and is usually made larger with more epochs.

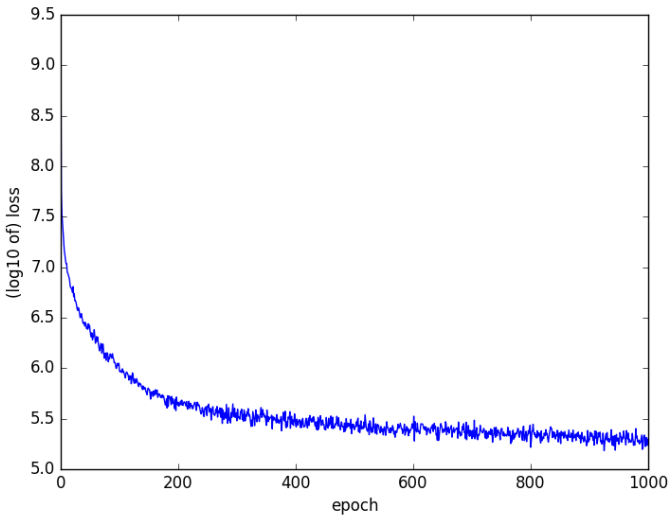

Figure 8: No overfitting in this session.

2.4.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are very similar to conventional Multi-Layer Perceptrons (MLPs), but differ in that the first few layers may not be fully connected. This allows CNNs to cope with much larger input values than MLPs at the expense of not being able to approximate as many functions. For this reason, CNNs are most commonly applied to images, where the input space is large.

***Definition* Same and Valid Padding:** When we take convolutions of a patch, we need to define how to handle the edge cases where the convolution matrix passes over a region outside a patch. Same padding pads the outside with the default value, often zero. Valid padding says the matrix is not allowed to travel that far out, hence reducing the output space.

A high level benefit of a CNN is that it is more likely to be sensitive to spatial patterns in images making it more suitable for computer vision tasks, that is, the neural network will not have to learn about apples in the top left and bottom right of the image separately - it will be able to learn it once and recognize apples everywhere. MLPs will almost always require some pre-processing in the input data to normalise the input to a format that it can work with more efficiently.

2.4.3 Inception Module

The Inception Module is a method of arranging convolutions in a CNN to achieve a more computationally efficient design and has enabled many implementations to achieve the current state of the art, notably Google LeNet which is more accurate and 12 times smaller than the previous state of the art published only 2 years ago[13]. The idea is to take a number of differently sized convolutions and use the output from each of them in the next layer. This strategy combines the strengths of using a large convolution mask with the efficiency of doing small convolutions.

2.5 Existing Work

"A Novel Monte Carlo Noise Reduction Operator"

Xu et al. provides an excellent demonstration of a noise reduction method for Monte Carlo noise. Their research applies directly to filtering noise generated from computing global illumination results, making specific use of the rendering loop. For example, they note that much of the noise arises from calculating diffuse rays recursively and that specular and glossy rays create little noise. It has to be said that a major drawback is that this is an algorithm that has to be run on the image each time, which would require expensive hardware and computation time for every image that is to be denoised. However a neural network only requires heavy computation at train time and applying the network afterwards is very cheap[14].

"Removing the Noise in Monte Carlo Rendering with General Image Denoising Algorithms"

For more general scenes with varying depths and camera effects, Kalantari et al. has proposed an algorithm that is effective at removing noise from these scenes. This is one of the first papers to address the problem of spatially dependent noise which arises all the time in Monte Carlo rendering. Again one drawback of this method is that we cannot run the algorithm on a new image very cheaply once the model has been learnt[15].

"Image Denoising: Can plain Neural Networks compete with BM3D?"

Burger et al. makes a challenge to using traditional algorithmic heuristics for denoising and makes a move to letting a supervised neural network learn this denoising function from a noisy and clean dataset. Two very large hidden layers are used with a hyperbolic tangent activation function in a standard feed forward neural network trained on noisy and clean images. This paper makes no assumptions about the source of noise and have been proven robust on a variety of noise types. This is excellent. Their results show an improvement over traditional algorithms for some types of images - in particular the neural network finds striped and jpeg noise particularly difficult. They also seem to be keen to emphasise the importance of a large network and using large patches which may negatively impact speeds and limit the hardware we can run this on in future[16].

"Natural Image Denoising with Convolutional Networks"

A good demonstration of the application of a Convolutional Neural Network to the image denoise problem is provided by Jain et al.. The great benefit of this method was that they could afford to use full size images e.g. 512x512 to train the network and hence could take the whole input image during a final denoise application. However this group noted that the performance was poorer when it came to introduced noise of an unknown variance in strength - which is the case in many rendered images. It should be noted that only greyscale images were studied in this paper[17].

"A Machine Learning Approach for Filtering Monte Carlo Noise"

Kalantari et al. uses a standard Multi Layer Perceptron as the neural network architecture. In this paper, they carefully describe the architecture of their network and presents their results to contrast with existing methods. This approach seems currently to outperform many of the existing methods. One drawback that the authors noted was that owing to the fact that this is a neural network, some effects such as specular reflections show up well only if there exist a large number of specular examples in the training set. Additionally, seven different, often domain specific data images are used to generate the final output, decreasing the usability and the range of 3d software

packages that it can be used on. So although this particular implementation seems to be producing some of the best results, the way in which so much information is required in the system is rather cumbersome[18].

3 The Dataset

I am fortunate in that there is an abundance of Public Domain and Creative Commons 3d artwork on the internet available for me to use as my training and test sets. I used the 3d software Blender[19] to render these scenes into images that will form the majority of my dataset.

The methodology for obtaining the datasets is simple: render just enough samples for an image so that there is no visible noise - this becomes the ground truth image, then divide the total number of samples used for the ground truth by around 20 and re-render - producing a noisy image - this becomes the input data.

The most successful technique for noise filtration also required an additional input. This is an OpenGL render of the basic 3d scene from which the renders are taken. For this pass an additional clay render is produced and saved as input data. This is always fast to render for our purposes, with most of them taking under 1 second.

Throughout the training process, training examples used were selected at random at each epoch. We were careful to avoid overfitting when using 64 by 64 patches from our dataset as this meant we had only 12,000 training examples. However, due to the small size of the networks we could not see any visible overfitting from our error graphs. Interestingly, using dropout actually increased our error rate and decreased the ability of the network to converge for many of the models tried - including most of the successful ones, most likely due to the fact that the output is hugely reliant on each input and training a reduced set of neurons would update the weights incorrectly with respect to our image equilibrium. This was not a problem in our final product as we used pixels as individual training examples so our dataset size was around 50 million.

We were careful to include images of varying magnitudes of noise, ranging from thin noise in images that have been rendered to half the number of samples of the clean version, to coarse noise which may only have 2 or 3 samples rendered out of 80 to 100 for a clean render. This was to address one of the challenges we mentioned on the introduction to diversify the noise patterns we can handle.

We have included a selection of training and test example images below. In order of appearance, clockwise, Vault by Cephei(CC-BY), Train by dailerob(CC-BY), Caustic Fair by Zichen Liu(CC-0), Coffee by b2przemo(CC-BY), Workshop by oldtimer(CC-BY) and Fishy Cat by Manu Jarvinen(CC-0).


4-8-8-4
4-8-8-4
ESPRESSO
cafe

4 Machine Learning Models

4.1 Preliminaries

Throughout the experimentation process, it is very important to have one uniform error metric. Our choice was the sum of the squares between each individual pixel of the output and the clean render. When this is minimized during the backpropagation phase, the output image should tend towards the clear render. This is a highly advantageous property for us. However, a shortfall of this is that if the output image is quite different from the clean render, then the error returned is liable to be far too large to fit into a standard double. Under these circumstances, a NaN is returned and backpropagation cannot begin. We will use normalization to circumvent this problem - mapping each pixel value from [0, 255] to [0, 1] reduces the frequency this occurs. Another shortcoming of this error function is that there are many ways in which a certain error value can be achieved. For example an image with a few very incorrect pixels could yield the same error value as an image with every pixel slightly off. We will see later on that models occasionally produce both these situations and under the occurrence of such contention, we will judge by using additional test images and by eye.

Correct parameter initialization is crucial for good training. On fully connected layers, if the matrix is initialized with random values, then the output is likely to be solid noise. This is because the neural network tries to factor in the surrounding pixels far too much resulting in unintended noise. To avoid this issue, each fully connected layer is initialized to the identity matrix with some noise added, ensuring that on the first iteration, the output is similar to the input - a good starting place for the network to start adjusting weights. A similar idea is used for the convolution layers, but this time the convolution mask matrix is all zero with a 1 in the central entry. The initial values for biases were set to the zero vector with some minor noise (mean 0.04, standard deviation 0.4), this technique as was introduced by Krizhevsky[20] is to prevent neuron death at activation nodes. This, coupled with an initially low learning rate gives us good convergence to the final result.

The image model we used for filtering is a typical RGB system with 3 channels. Any partially alpha pixel taken as input is assumed to have full alpha. During the first stages of modelling (Models 1 and 2), a monochromatic filter was trained on the basis that it may be applied separately onto the different channels. However, we realised that other channels may assist the filtering of a given channel, we switched to filters that act on 3 channels for later Models. Please note that the error function used for Models 3, 4, 5 and 6 do not correct for the additional colour channels i.e. it displays the error for all 3 channels. Since $log_{10}(3x) = log_{10}(3) + log_{10}(x)$ and $log_{10}(3) \approx 0.477$ we will take this quantity away from the error on the graphs when making a direct comparison to models 1 and 2.

Inspired by the range of exciting ideas in the literature review section - convolutional neural networks, hyperbolic tangent activations, using additional outputs from 3d software packages as the neural network input - I was very keen to try the broad spectrum of ideas that I had seen, as well as adding some of the latest state of the art in machine learning. The total number of different models I tried during the experimentation phase is roughly one hundred and fifty, however, I will outline below the five most successful, instructive or surprising models that helped me to arrive at the final product, which is model six. I will use the notation gt to denote the ground truth image usually the clean render, ny to denote the partially rendered image, gl to denote an OpenGL material render, Roman Numerals to denote the neural network filtered image and psd to denote a noisy render with post processing in Photoshop using template filters with parameters judged by a human trying their best to reduce noise.

This page is left intentionally blank.

4.2 Model I: Patch to Patch Single Layer Perceptron

The first successful model that achieved appreciable results was a very shallow Artificial Neural Network that processed image patches of 64 by 64 and returned a filtered 64 by 64 output. The shallow depth meant that the amount of interference from neighbouring pixels is small and that it converged very quickly. Once trained, the network can be re-evaluated cheaply. A critical shortfall of this scheme is that there simply isn't enough parameters here to store all the information that is being learnt, as we will see.

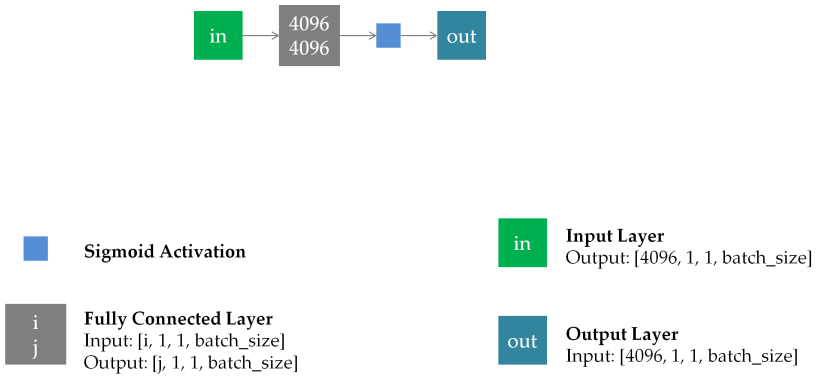

Figure 9: Model 1 Architecture.

This neural network was particularly easy to train due to its size and shallowness. With just a pancake of parameters, the process easily converges within a few epochs. The fact that over 50 epochs there was little change in the error since epoch 10 was really encouraging - the number of training examples that the network could have seen by epoch 10 is a small fraction of the whole dataset - this means the dataset is rather homogeneous in what it taught the network.

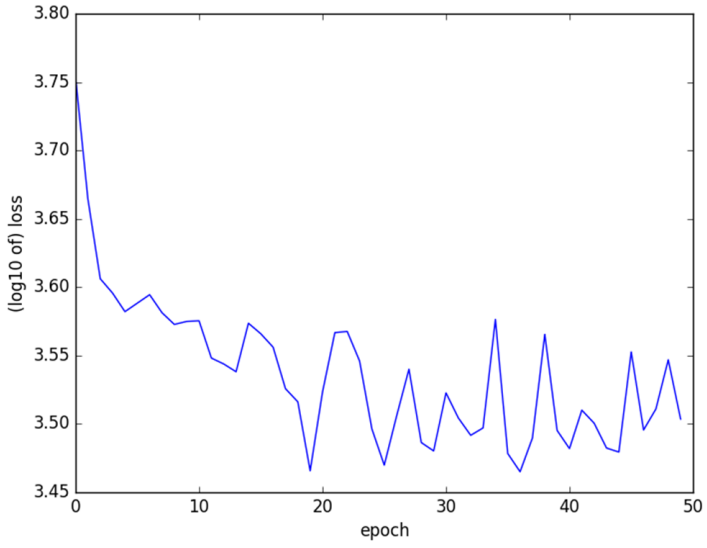

Figure 10: Training Error for Model 1.

The results from testing shows that the neural network has learnt the right idea of what needs to be done. It can be seen in the examples Contrast Edge and Haze that the result of the neural network is superior to the noisy output, albeit at the expense of sharpness. The loss of detail problem becomes very prominent in Stripes, Lilypad and Cave Wall where important feature details almost blends into the background. In the end, this turned out to be the most effective non-convolutional neural network for patch to patch filtering.

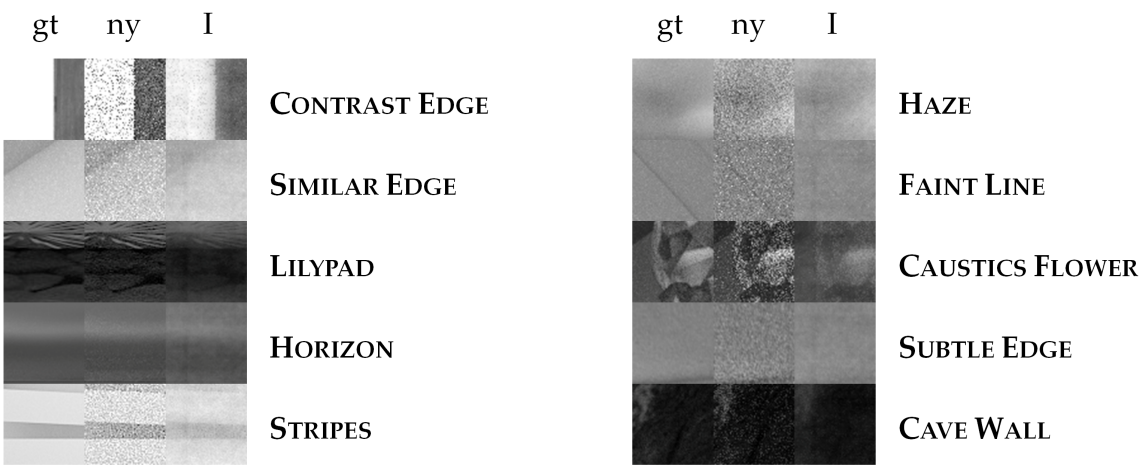

Figure 11: Model 1 Results.

4.3 Model II: Path to Patch Convolutional Neural Network

There simply wasn't enough parameters present in a small neural network for this kind of task, as seen in the last model, and so we decided to go deeper. Adding an additional fully connected layer and having 2 convolutional layers in parallel certainly increases the parameter count and is reflected in the output where there is a visible reduction of noise. However it should be noted that the error rate between models 1 and 2 are similar.

Training this model required more care as it is bigger and more prone to overfitting. We noticed that the majority of the learning occurred in the first 100 epochs, after which no significant progress was made. Our instinct to decrease the learning rate resulted in the same epoch-loss profile so the learning rate was maintained.

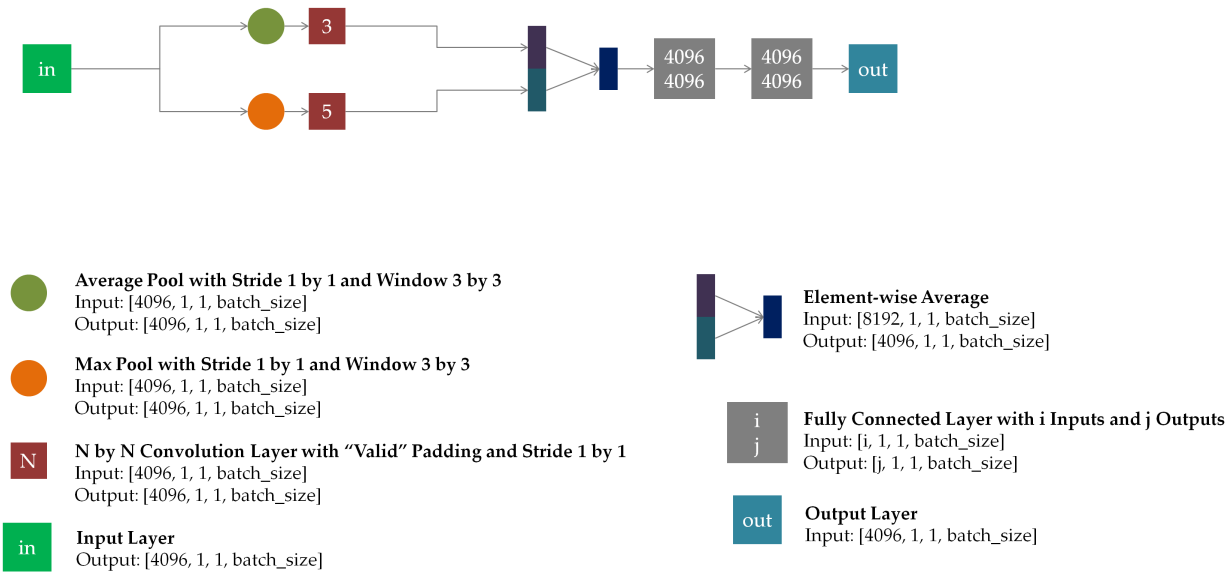

Figure 12: Model 2 Architecture.

From the result set we can see that this network is far better at preserving detail. Examples such as Subtle Edge and Stripes produced far more amicable results than the previous model. Cave Wall and Horizon produced excellent quality results.

However, we noticed that for certain examples such as Contrast Edge and Similar Edge there was an increase in the amount of blurry noise. We realised that this was because it is difficult to ascertain whether the noise in an image is part of the original image. The noise in Contrast Edge was certainly noise, but it could equally well be salt and pepper on a table edge. Additionally, we realised that if we could feed in additional colour channels, the neural network may perform better due to the additional information for each output pixel. This led us to rethink our strategy for modelling the neural network - perhaps we should allow a greater range of pixels affect the outcome of an output pixel? We will explore this next, resulting in Model III.

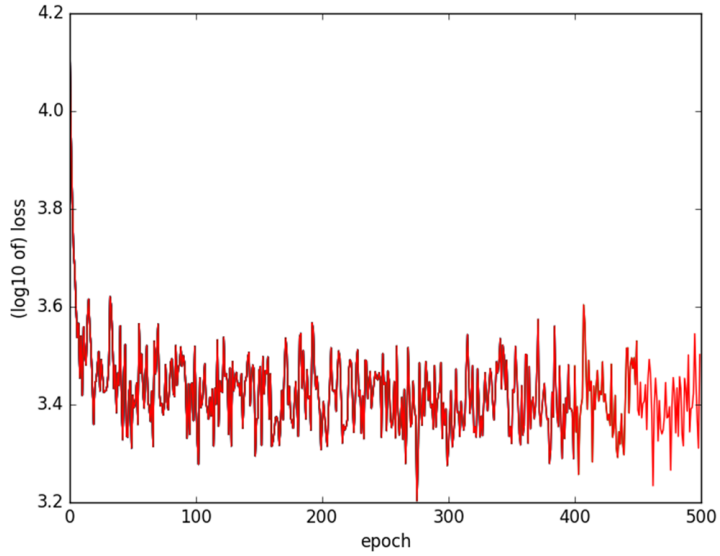

Figure 13: Training Error for Model 2.

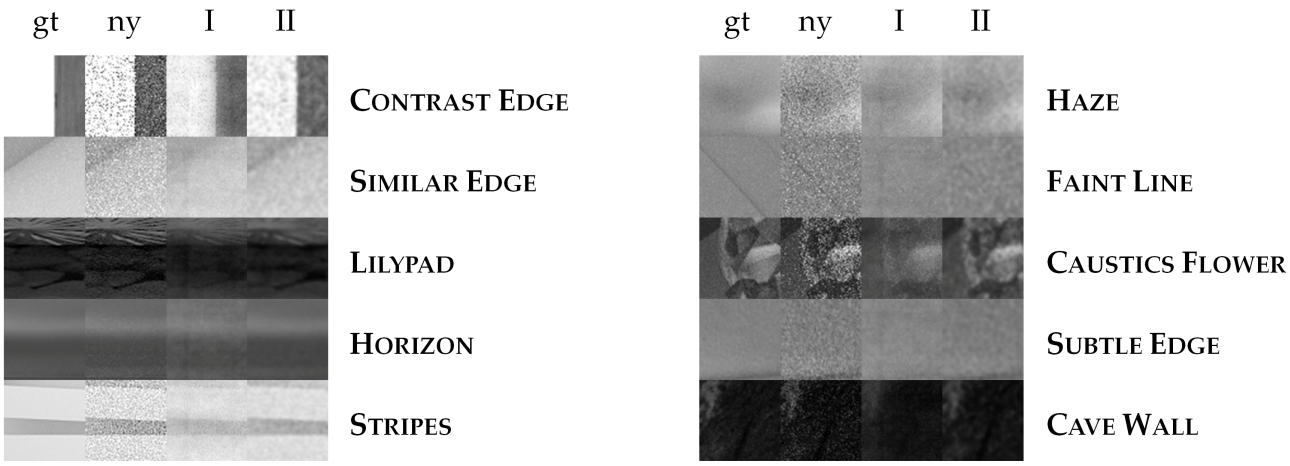

Figure 14: Model 2 Results.

4.4 Model III: Patch to Patch Mixed Neural Network

A common issue with de-noising algorithms mentioned in the previous works or in the filters that are commonly available on Photoshop is that the end result is blurry - causing detail to be irrecoverably lost. The training set for noise removal frequently have inputs with lots of sharp noise that disappear in the ground truth. So it is no surprise that the neural network will learn to blur the input as well as removing noise. We will now attempt to address this.

4.4.1 Producing Sharp Images

First we have our 3 by 3 sharpen convolution mask that is applied to one branch. This is a standard sharpen kernel that is used in lieu of the identity kernel in the hope of emphasising detail after some noise removal convolutions. This is likely to be modified during training as the kernel values change. The final unsharpen mask[21], however, is an untrainable module that applies an unsharpen convolution to the final image. In this case we are favouring a sharper final image over something that minimizes the l2 norm error. We think this is justified as there is inherently less information provided by a noisy image, so we would like to sharpen the image up after having performed noise removal. Since the unsharpen convolution kernel is likely to increase the error, we will make this convolution untrainable such that at backpropagation will not modify the entries of the unsharp kernel.

Aside from kernels, we also have a branch that takes takes the average of the largest in a 7 by 7 window, the average in a 5 by 5 window, the average in a 3 by 3 window and the smallest in a 7 by 7 window. We hope that this will dim fireflies and also flesh out boundaries of major regions. Since in the max and min pool, we only take the brightest or darkest pixels, this decreases the chance of noise interfering with our edge detection.

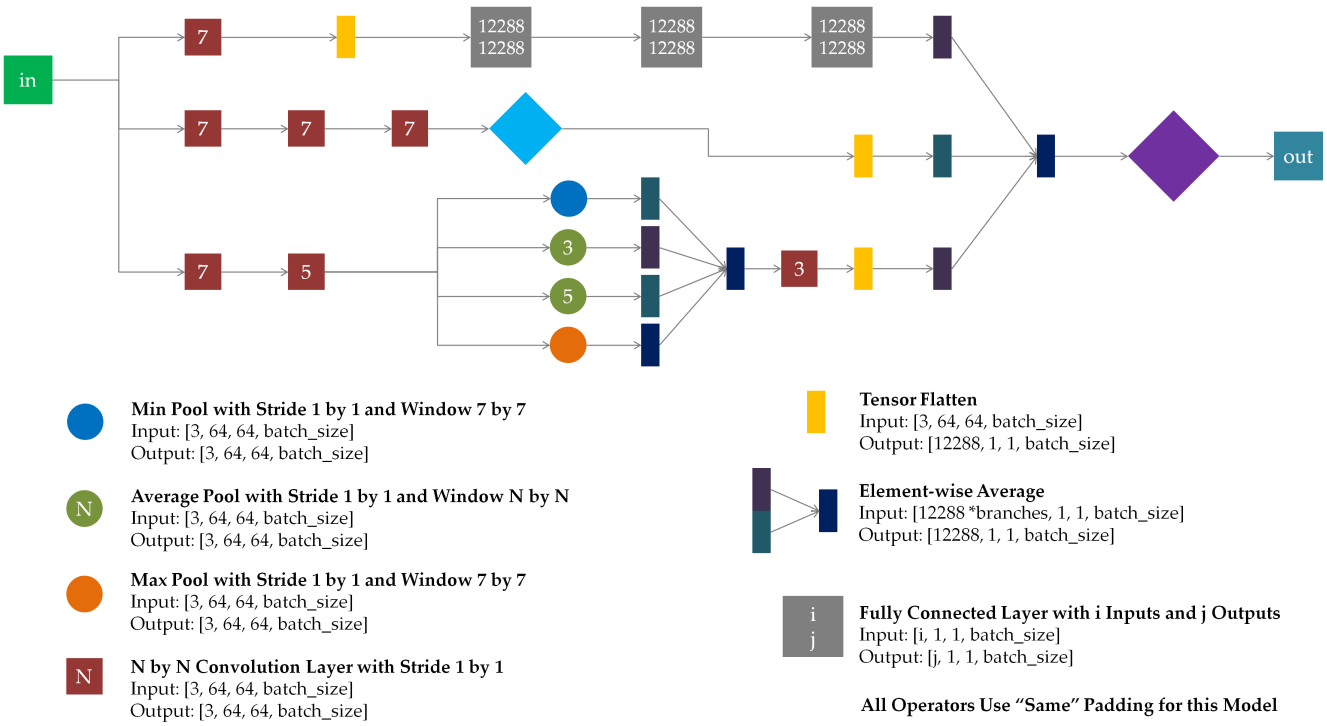

Figure 15: Model 3 Architecture.

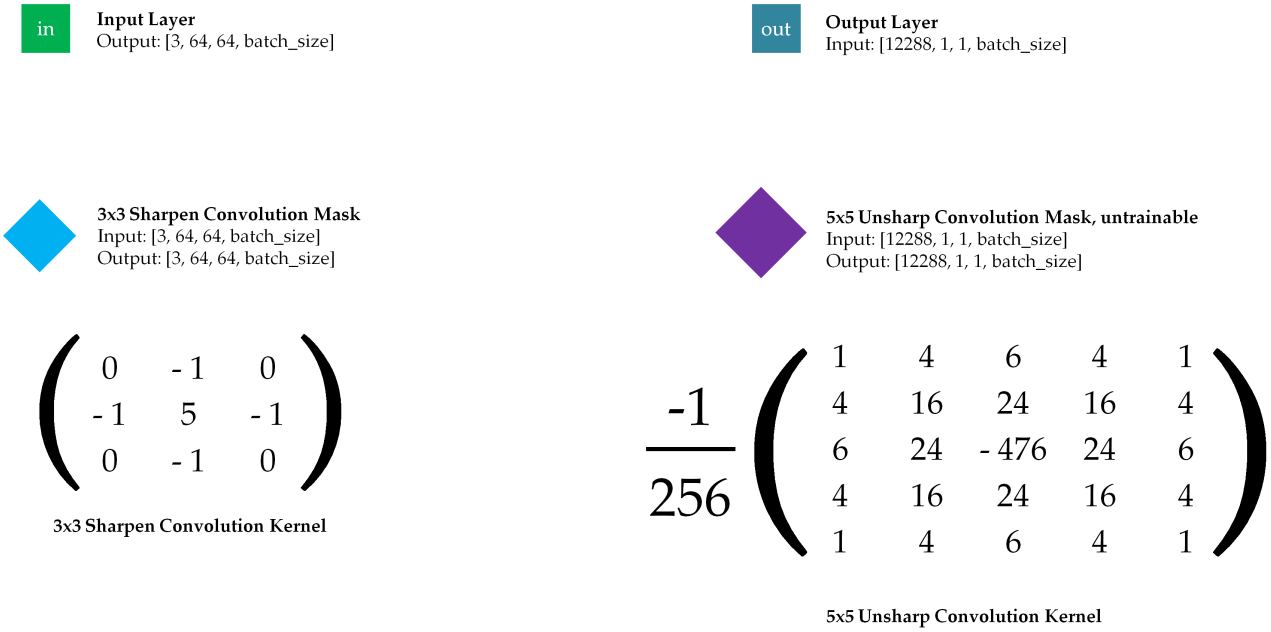

Figure 16: Model 3 Architecture.

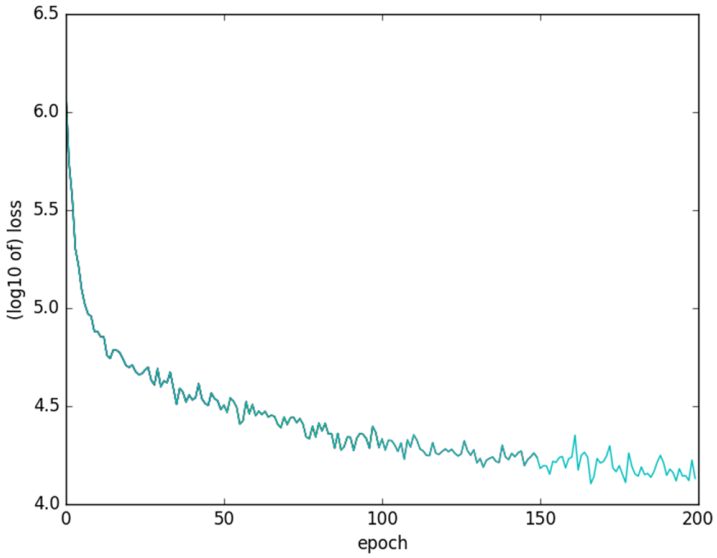

Figure 17: Training Error for Model 3.

4.4.2 Results and Evaluation

In a number of the examples, such as Bright Edge, Universe I and Caustic Chaos, you can clearly see small squares around fireflies or bright detail. This was a direct result of our max/avg/min pool module. Although this seems detrimental, it actually helps to preserve detail in scenes where there is a lot of edges such as Caustic Flower. Detail preservation was really good here with Similar Edge and Faint Line retaining the most out of the models so far. This was the first model that I used all 3 colour channels for filtering and the colour preservation was excellent with even minor variations in the original image preserved, for example the dark region in Contrast Edge.

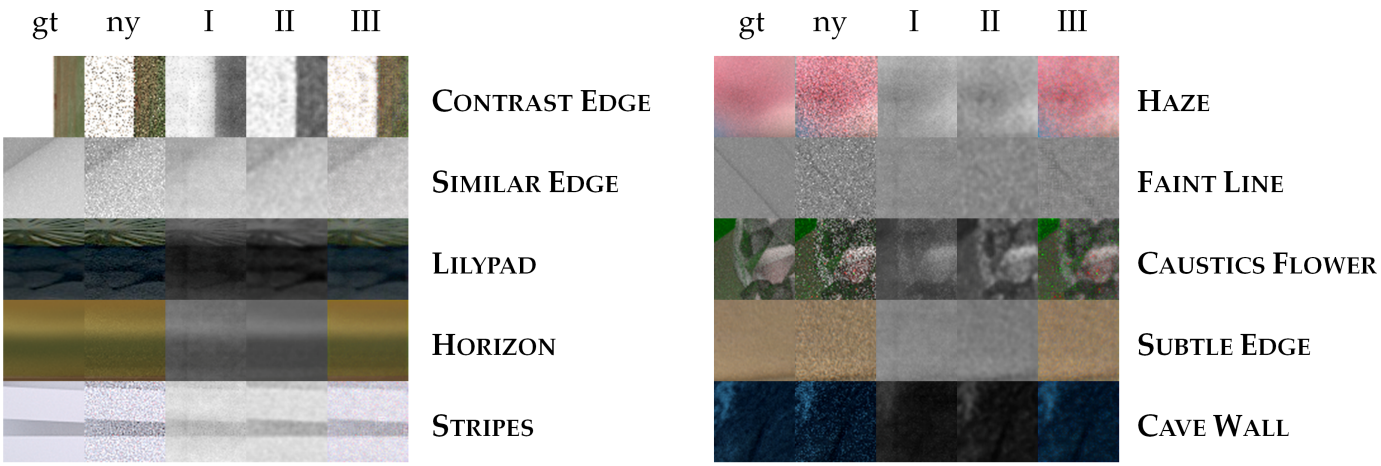

Figure 18: Model 3 Results.

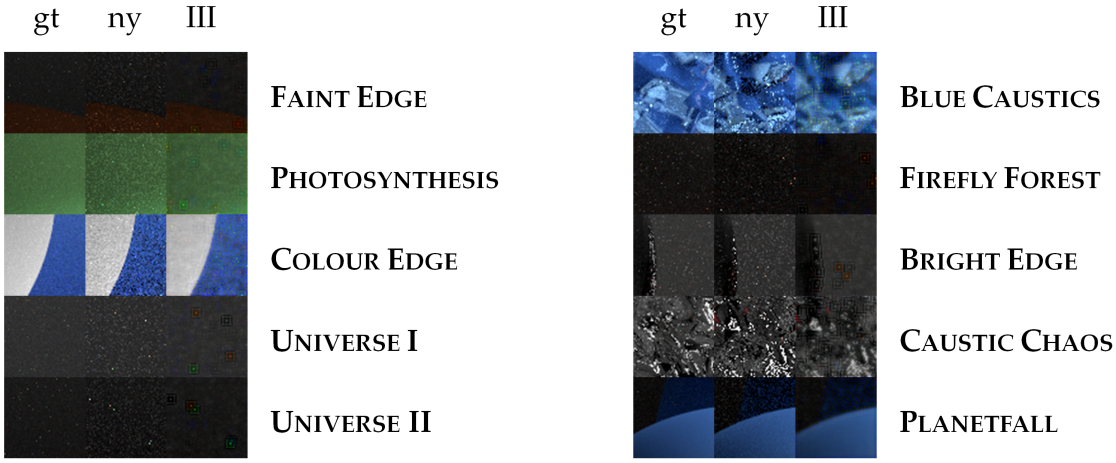

Figure 19: Model 3 Results.

The biggest drawback of this method was the enormity of the network and intermediates. With around 8 GiBs of parameters and intermediate values, this network was very slow to converge and very slow to evaluate. This is highly undesirable. The huge number of epochs this network has been through meant that each training example would have been seen an average of 10 times. Combined with its size, we are also at risk of overfitting.

This page is left intentionally blank.

4.5 Model IV: Patch to Patch Deconvolutional Neural Network

4.5.1 Introducing Deconvolutions

An issue with using a convolutional neural network is that its strength arises from the fact that the dimension of the input is reduced over time. Since we are maintaining the dimension of our patch, we cannot do this. A new technique for using a convolution whilst maintaining dimension was introduced by Noh et al. called deconvolutional neural networks[22].

Repeated experiments have shown that for this task, it is most effective when used without a fully connected layer. I will also begin with deconvolution layers and finish with convolution layers in order to prevent the original image data being destroyed by convolutions. Although this deconvolutions have most commonly been used after convolutions such as for image segmentation[23], it is most appropriate for us to use them first since we want to increase the space of data for each pixel and let the convolutions convolute this data back into a single pixel.

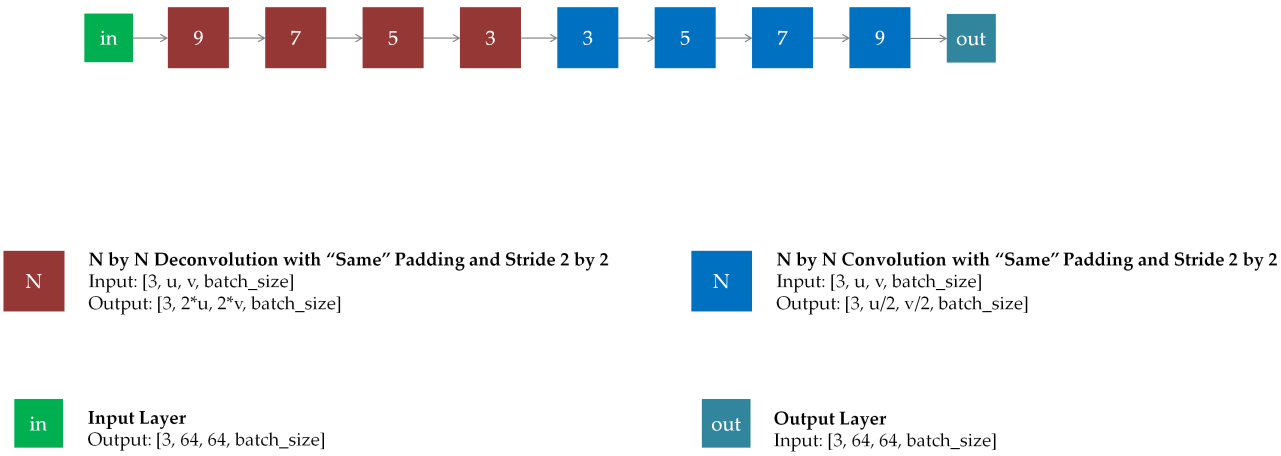

Figure 20: Model 4 Architecture.

4.5.2 Training

This network was particularly expensive to train due to its size. We see there is some good learning in the first 1200 epochs, coming to resting at a local minima. After this there is some spurious further learning which is highly likely to be overfitting. If we needed to decide on a final model, it seems like removing a few deconvolutions and putting a fully connected layer in its place would be a wise decision - we would get faster training times and, from Model 1 and 2, a good chance of getting a better render. However networks of this type uniformly failed to train on experimentation so we are left to use deconvolutions without Perceptron layers. We will now evaluate its effectiveness before deciding to use only deconvolutions or abandoning it entirely.

4.5.3 Results and Evaluation

This method at first seemed like a disappointment - the log error rate was around 3.9 compared with around 3.5 previously. However upon closer inspection of the test set, the overall result was far nicer than I and situationally better than II and III. One strength of this method was that fireflies were removed very effectively, especially in Faint Edge and Firefly Forest mitigating the need for a separate layer of post-processing. There is some welcome additional definition in Contrast Edge and Stripes.

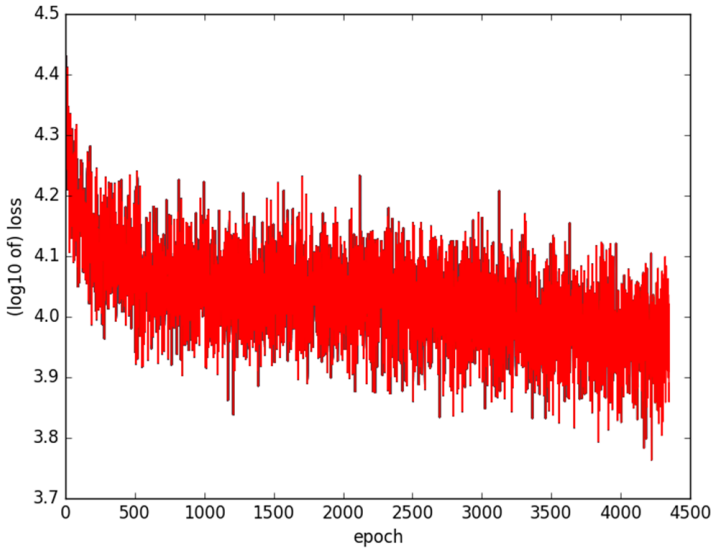

Figure 21: Training Error for Model 4.

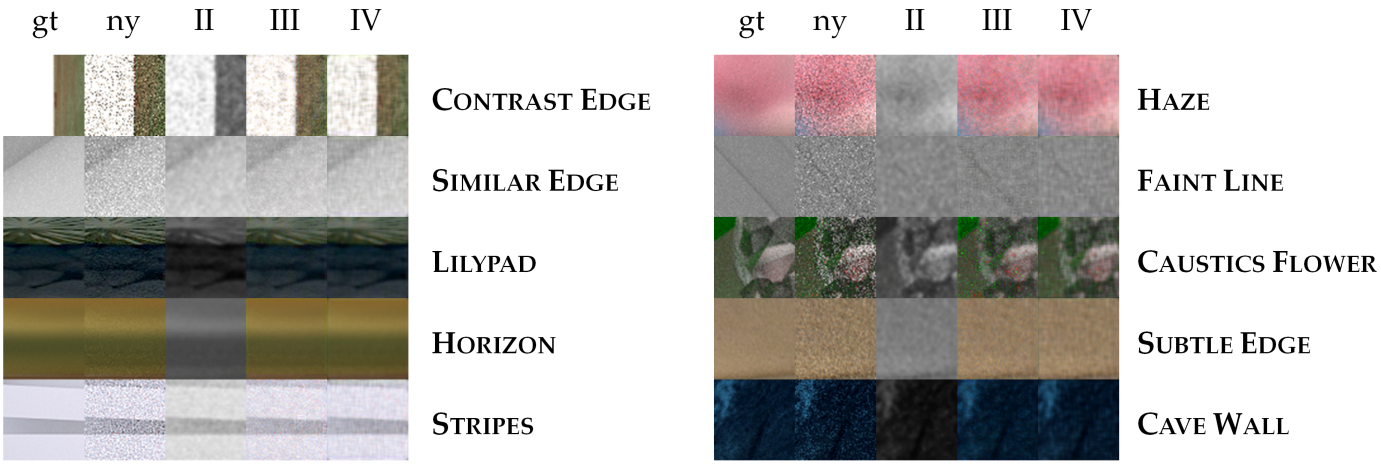

Figure 22: Model 4 Results.

In the discussion for the previous model, we talked about increasing the window of pixels that could affect an output pixel in the convolutional layers. One major fear of mine was that this would lead to entire regions being deformed or otherwise affected. However, thankfully, this is not the case, COLOUR EDGE and PHOTOSYNTHESIS demonstrated that whilst lines may be slightly blurred, entire regions stayed true.

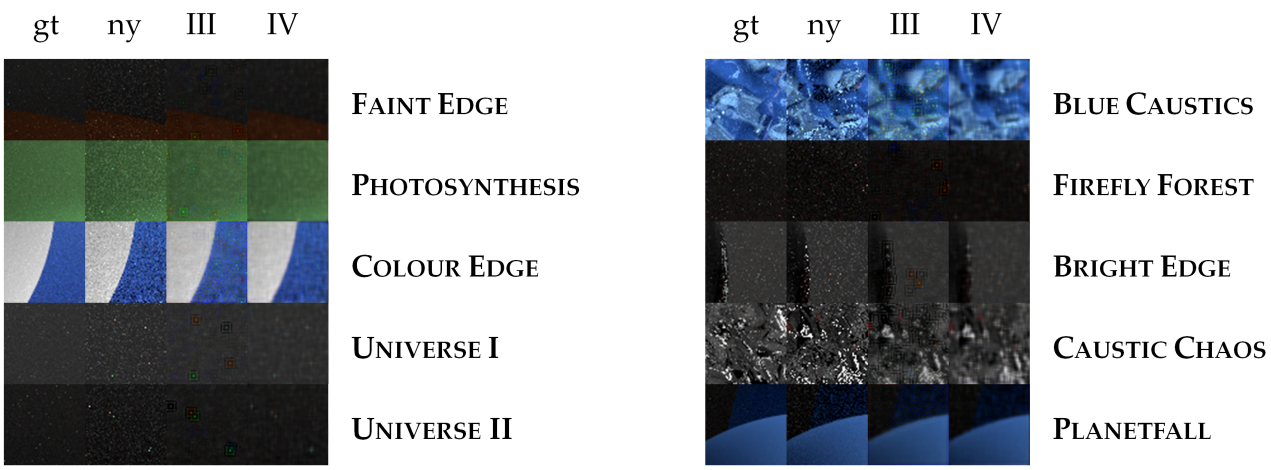

Figure 23: Model 4 Results.

4.6 Model V: Patch to Pixel Convolutional Neural Network

4.6.1 Moving to Patch to Pixel Networks

To further increase the number of parameters in our neural network per pixel output, we will switch the input output architecture to Patch to Pixel. This means that the input to the neural network is a 31 by 31 patch and the central pixel of the patch is the output.

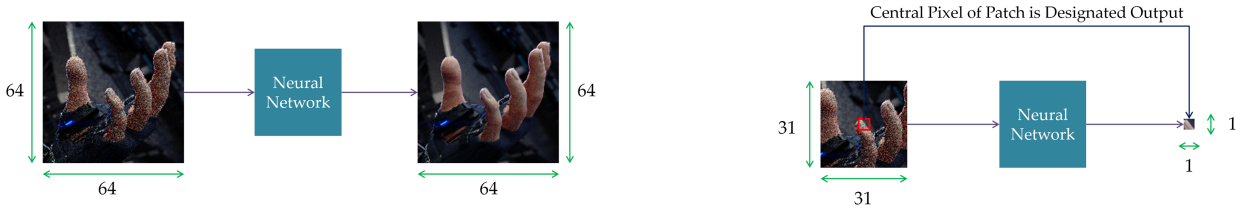

Figure 24: Left: Patch to Patch networks. Right: Patch to Pixel networks.

From a quality standpoint Patch to Pixel filtering is superior to Patch to Patch filtering for a number of reasons. First, we have automatically increased the number of training examples by 4096 times, i.e. each pixel is now an example with its own input. This means our training set size is around 50 million with another 30 million for the test set. One consideration is that during the entire training process, the neural network may not see certain training examples, by chance - however since we value each data example equally, we are happy to allow this.

Additionally, assume we are taking patches from a sufficiently large image, for Patch to Patch, the edges of each patch does not receive information from its surroundings from the other patches as in same padding. However we cannot use valid padding as this would reduce our output space. With Patch to Pixel, we can access surrounding pixel information for every pixel apart from the edges of the original image, greatly increasing the filtering accuracy.

One major problem of Patch to Pixel processing is that the amount of computational time increases by at least 3 orders of magnitude for the same architecture. We offset against this increase by using memory more efficiently and loading all of the data into memory before training. Despite this, our training time went from minutes to days.

4.6.2 Use of Additional Data

It is often a good idea to adorn a neural network with additional, appropriate information to produce a better render. For example, if there is a region of the scene where we have a texture material, then it would be highly advantageous to be able to use the information from the texture to help us in the denoising process. This idea is used in many papers including Kalantari's MLP[18] with spectacularly good results. Seven different images are used in the input, drastically increasing the training and evaluation time. Furthermore, some of the additional data can only be generated by certain software packages making the system domain specific. We wanted to maintain our network as a piece of general purpose post production software that may be appended to any rendering pipeline, hence we will introduce a general "OpenGL material render" that is to be included in the training and evaluation process.

The OpenGL material render time is very small compared to producing even a noisy render and can be produced on every 3d software package. In figure 25, we see that the noisy image has patches where the texture is occluded by fireflies and other inconsistencies. The gl render on the

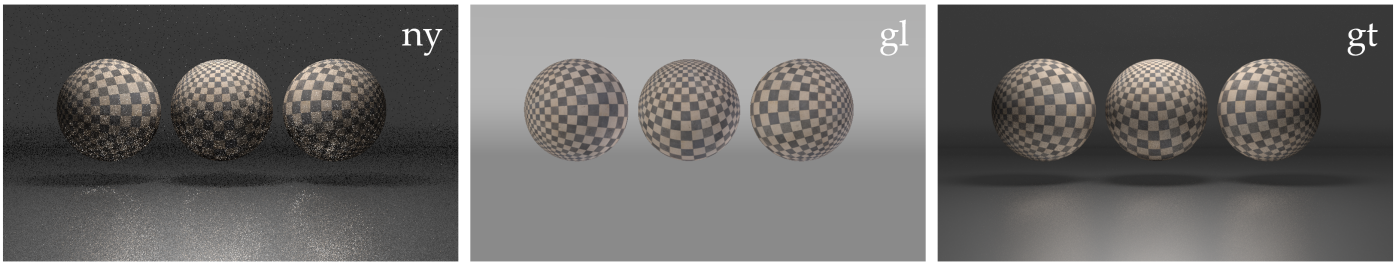

Figure 25: Benefits of having an OpenGL Render.

other hand produces a clear reference to what the texture really looks like underneath. This assists the neural network in discarding fireflies.

4.6.3 Architecture, Training and Evaluation

The input to the network is a 31 by 31, 6 colour tensor with 3 of the colours arising from the noisy render and 3 colours arising from the OpenGL input. The network has 2 convolution layers that maps trainable convolution matrices of varying sizes across the tensor. The dimensionality reduction comes from the valid padding of the convolutions. The output of the convolutional part is a 6 colour 7 by 7 patch which is flattened into a vector and fed through some fully connected layers until it is a single RGB pixel. The design of the neural network appears to be of a very traditional and conservative shape much like one used for image classification.

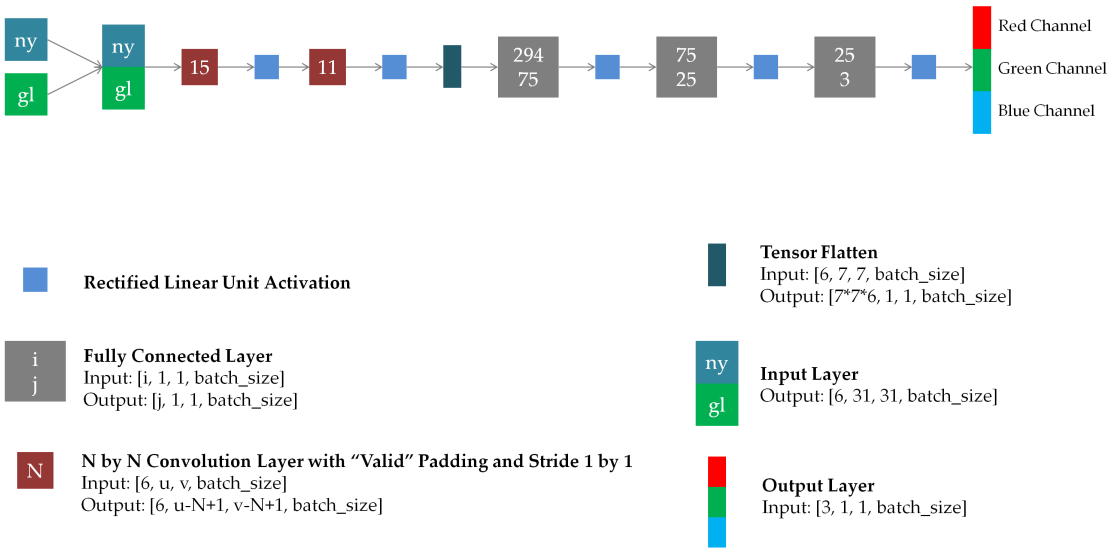

Figure 26: Model 5 Architecture.

This was a model that hugely benefited from sudden decreases in learning rate. We divided the learning rate by 10 at epoch 1000 and 1800, and from the graph, these were where we had huge decreases in the error rate. The final error was disappointingly high, however, surprisingly the final result was clearer than previous models. A full discussion is given in the next model as well as extensive testing.

The results show that one of the great strengths of Patch to Pixel is preservation of detail. In

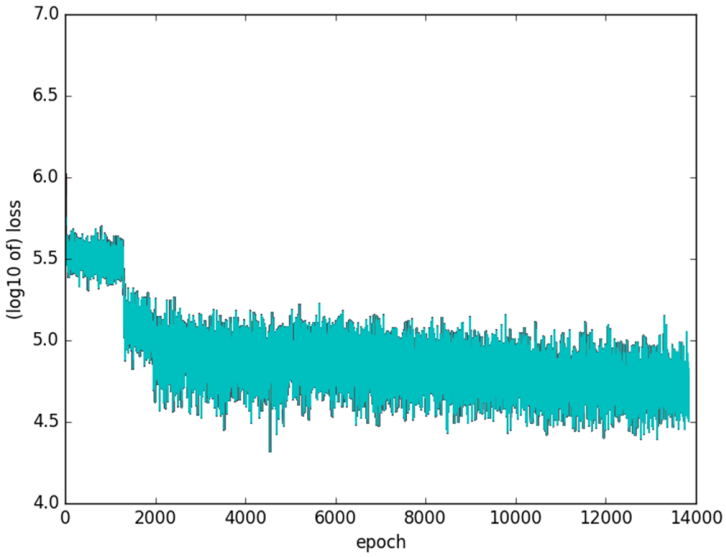

Figure 27: Training Error for Model 5.

MONKEY SMILE, the lines are sharp and crisp, almost as much so as the ground truth. The results panel shows the output at each 100 epochs of training - this was one of the methods we used to prevent models from overfitting. If there exists a few patches which is less well filtered than a few other patches which have smaller epoch numbers, then we say overfitting has occurred and we can perform early-termination.

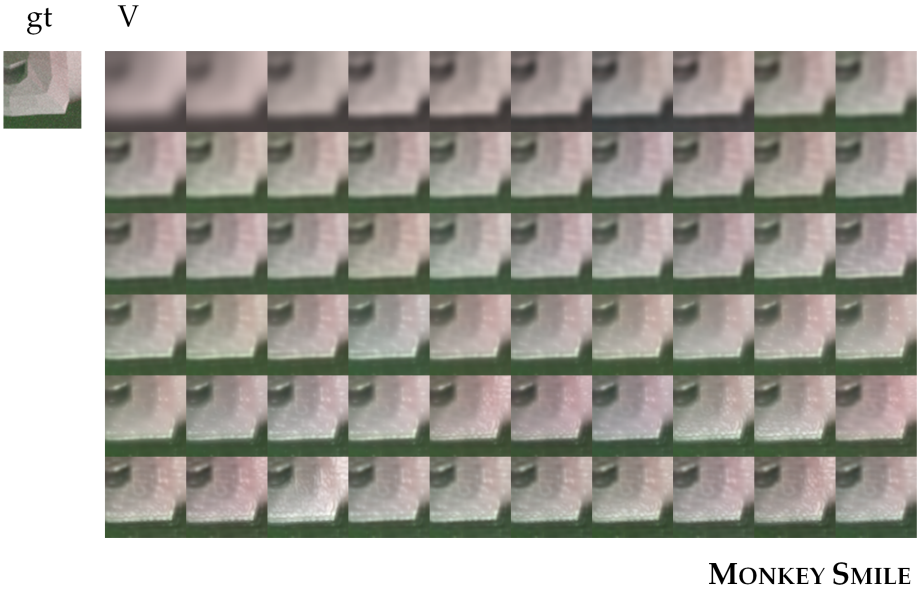

Figure 28: Model 5 Results.

4.7 Model VI: Patch to Pixel Inception Neural Network

Having now experimented with a number of models, it was time to pool together our experience from each thus far to create our final product. Our experience tells us to go for a deep network with lots of parameters, to use additional information from an OpenGL material render, to concatenate and not merge tensors post inception and to be creative with window and stride sizes when performing a convolution.

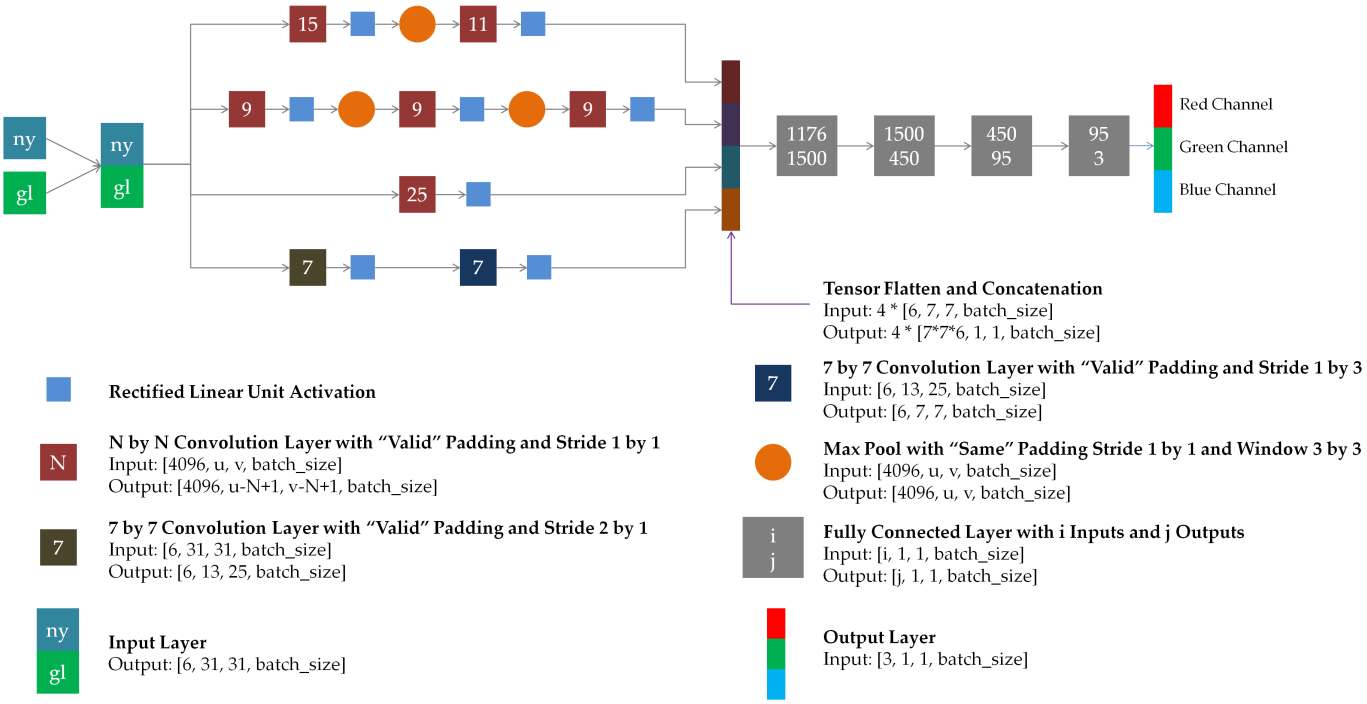

Figure 29: Model 6 Architecture.

We will use an Inception Module as the prelude for our fully connected layers. Each prong of the module has a different way of reducing the image. Notice the topmost prong is the same as Model 5 with an added max pool. As we discovered in section 4.4.1, max/min pools are invaluable to preservation of quality, so this was a welcome inspiration. We omit unsharp kernels here as they may be applied efficiently after coming out of a neural network. The bottommost prong is intended to remove horizontal or vertical noise by using non square strides. The features extracted from these prongs are then concatenated into one large tensor of data and flattened to fit into a fully connected layer.

4.7.1 Training

We ran this model many times with small variations to optimize the parameters - adjusting the learning rate, the learning rate update schedule, dropout rate and the architectural hyperparameters. We report our optimal architectural hyperparameters in figure 29. Since this was the model we used in production, we left the training to run for 6 days, cumulating in around 44000 epochs. Worryingly, although the training error decreased nicely as training progressed - showing good learning, our final error rate was much higher than the final error rate for many of the other models. This may be explained by slight distortions in colour in some scenes. Or in other cases, the neural network was simply being too assertive about its beliefs, but this is not always a bad thing. For example although Caustic Flower has consistently been the most difficult to filter correctly, Model 5 has done an admirable job - even though the output is not hugely similar to the ground

truth - we would liken the result of VI to the ground truth, more so than the result of the other renders - which would have had yield a lower error.

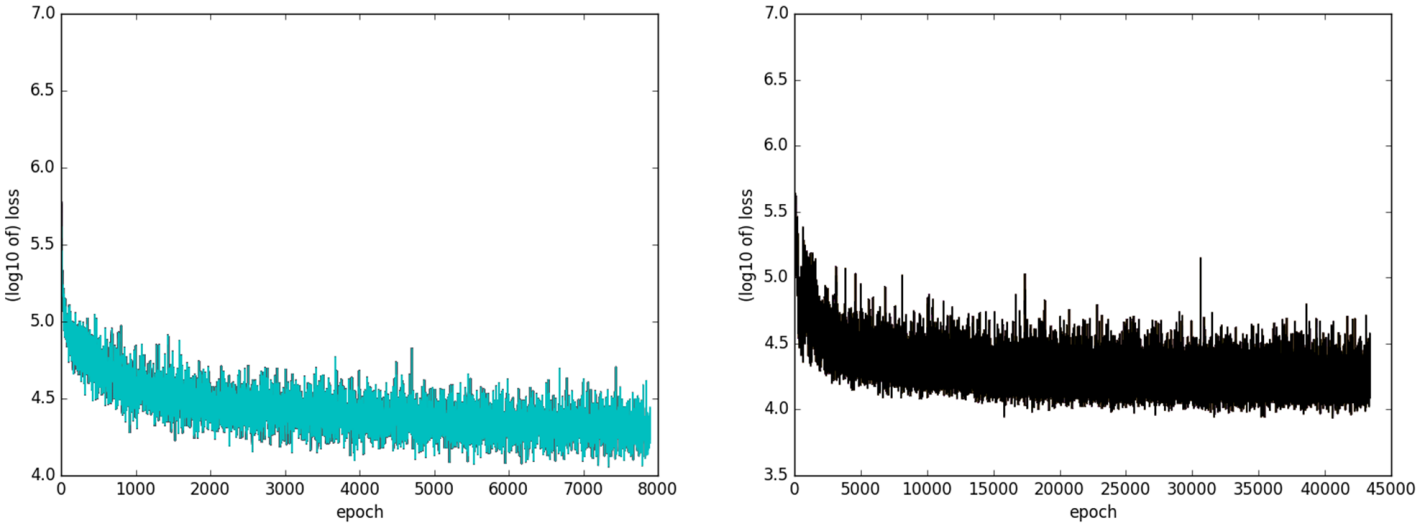

Figure 30: Training Error for Model 6. Left - first 7900 epochs, Right - 44000 epochs.

4.7.2 Results and Evaluation

As this is our final and best result, we will leave our detailed analysis of how well it meets our objectives for section 6. For now, we will consider the output as a comparison to our previous attempts. First thing to note is that there is very little blurring compared to before - individual lines and boundaries in the test images stayed sharp and clear, as in Cave Wall and Stripes. Noise removal from a blurry background has also improved as in Haze. Notice how there is this undesirable square of haze that floats near the four edges of the output. This is due to the small dimensions of the test input - 64 by 64 - this effect far more subdued in images of size over 100 by 100. For images over 500 by 500, other sources of error renders this effect insignificant.

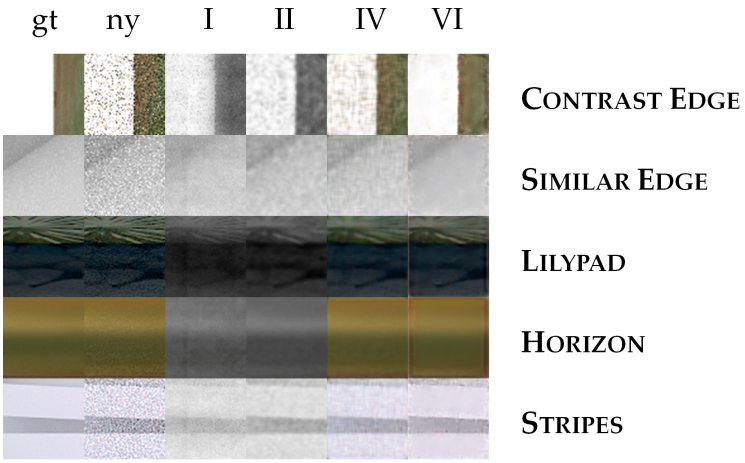

Figure 31: Model 6 Results.

One of the places where the OpenGL addition helped significantly is Contrast Edge where almost all of the noise in the image has been removed, producing an output very similar to the elusive ground truth. The majority of the noise is gone from Similar Edge, with some loss of detail at the edge. From figure 30 the final error was around 4.0 to 4.4. When we take 0.477 from this figure, our error is quite low, but certainly not the lowest.

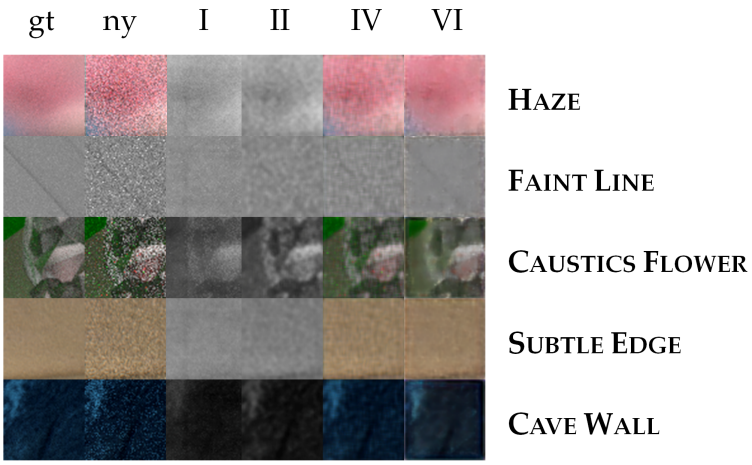

Figure 32: Model 6 Results.

5 The Web Application

Our overall architecture comprises of a modular web architecture, to allow easier separation of concerns and distributed operation. Additionally, this design allows render nodes to be placed behind strict firewalls such as the Department of Computing firewall. I made use of code snippets from tutorial websites and adapted it to my needs. Snippets that assisted me were from W3School's php[24], NodeJS' filesystem[25] and npm package manager examples for restler, multer, static-serve, mkdirp, mysql and http[26]. The website template is html5up[27]. The full detailed acknowledgements is in the appendix.

5.1 Individual Components

User App

Front end web application written in Javascript/jQuery that allows users to submit a noisy and OpenGL image along with their name and email. The client app shortly receives a unique key that is issued to it by the Web Host and waits until there is a result on the Amazon host that accepts the key, before retrieving it and displaying it to the user.

Web Host

A back end application written in PHP that logs any jobs that are received from the user to a database and hosts the job for the render node to retrieve. It also logs the reviews that the users might submit after having retrieved their result. This is where a large number of sanity checks occur for the input data. Besides safeguarding our servers from injection attacks, we also restrict the size of the image to sane dimensions and ensure the noisy image has the same dimension as the OpenGL.

Database

A MySQL database that stores the TASKS and the RATINGS received. This database is hosted which means the render node queries its rows directly in the first instance, without adding load to the web server.

Amazon Host

A simple Node.js server that listens for incoming requests of the filtering results of jobs from render nodes, the communication from the render nodes occurs via a key and the results are stored on the file system. It then hosts the images received, again under a key, for the user to access.

Render Node

A heavy Node.js server that queries the database and the web host for jobs and stores it in the file system. Any outstanding jobs will cause it to spawn Python child processes that invokes Tensorflow framework. The locally stored neural network weights are loaded into memory and the reconstructed neural network then begins to evaluate the outstanding jobs. When it is processing a job, the server will not synchronize with the database and no new jobs are fetched. After completion, it sends the result to the Amazon host and continues to listen for more jobs.

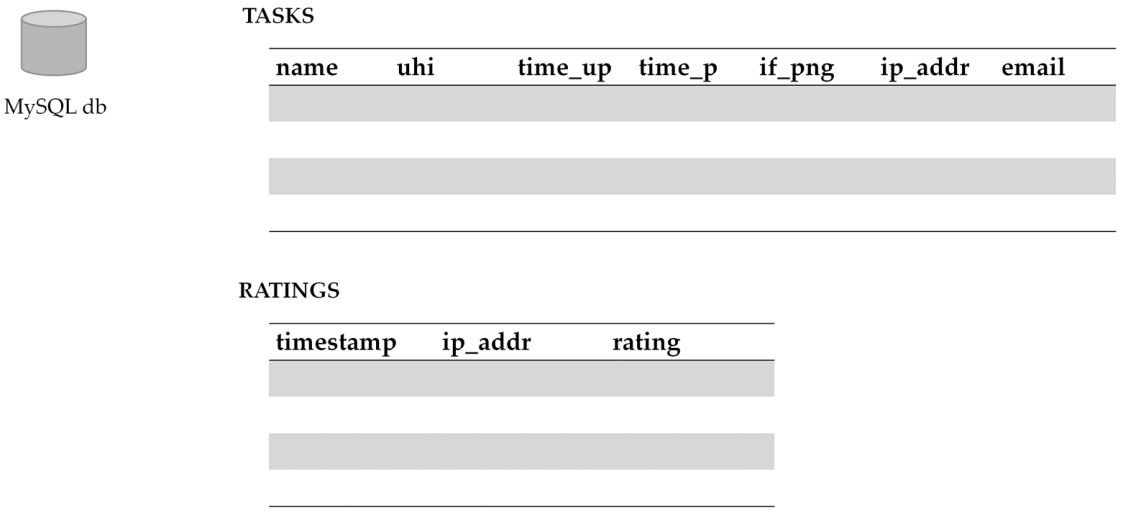

Figure 33: Database Schema.

Normal Operation

There are two actors in the system - the user who initiates the original request and the request for the end result, and the render node which initiates the request for a job and sends the result to the Amazon host.

The following actions are initiated by the user:
1 - The user visits *airenderer.com/app* and uploads a payload to be processed.
2 - The web host generates an unique hash identifier (uhi) and is associated with the payload on the file system. The uhi is a hash of the current UNIX timestamp and the name of the user.
3 - The client uses the uhi generated to check for the result of the render that is hosted on a separate box.
4 - As soon as the render result is received from Graphic01, it is hosted and therefore any request for this resource by uhi is fulfilled.

The following actions are initiated by the render nodes:
a - Graphic01 requests the uhi of any outstanding jobs from the database and updates that row as retrieved.
b - The uhi is returned to Graphic01.
c - Graphic01 uses the uhi to retrieve the payload from the web host.
d - The web host returns the user uploaded payload to Graphic01, which is then processed.
e - The processed image and its uhi is then sent to the Amazon server and hosted immediately.

5.2 Distributed Operation

One of the strengths of the modular design is that it allows the workload to be distributed during surges in demand. The trained neural network and web communication system is packaged in a git repository called airenderer_heavy. This may be deployed to any number of additional render nodes and can be used in lieu of or with Graphic01.

The fact that no part of the system has knowledge of the number of the number of render nodes there are means that we may launch additional render nodes without modifying or informing

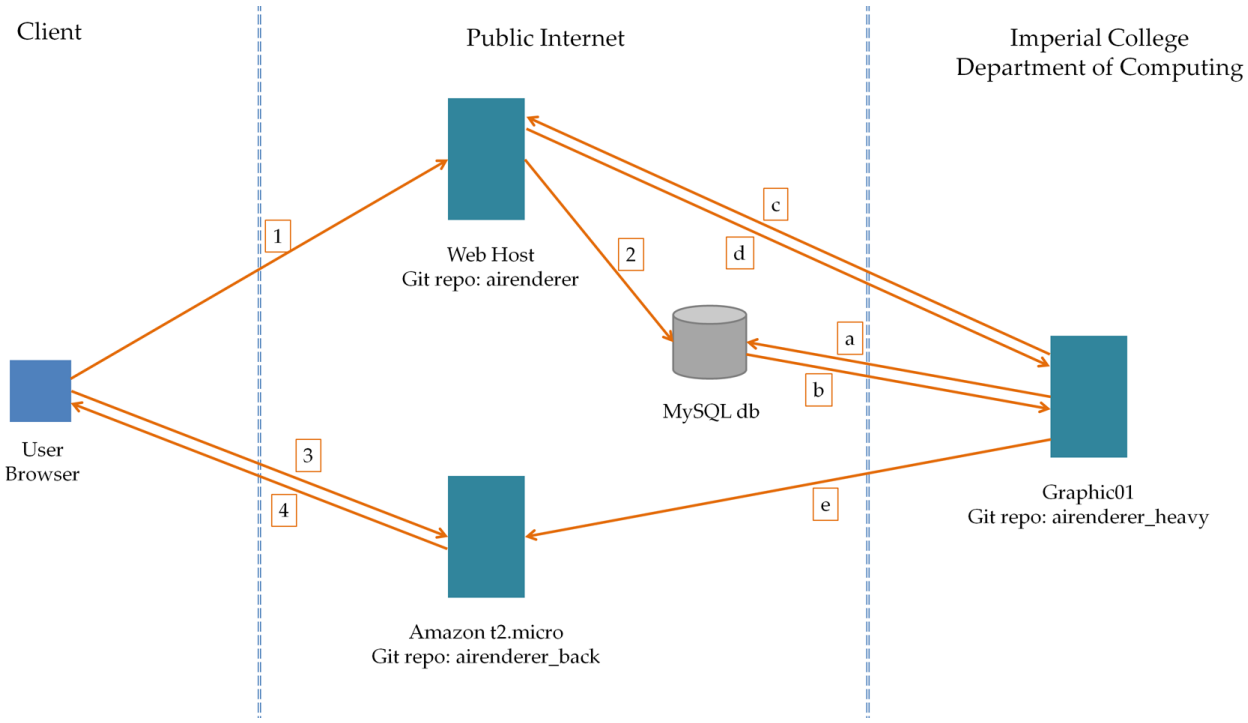

Figure 34: Web Application Architecture.

the current system. This operation has been tested on one additional public (Amazon) node and one additional Graphic node.

5.3 Neural Network Adaptation

For each render node, the logic is simple. If there is an outstanding job, retrieve the noisy and OpenGL images and cache them. We then invoke the neural network and feed in the patch corresponding to each pixel of the input image and the output we cache as the result. This is immediately sent to the results host server.

Our scheme for handling the edge cases, which arises when a patch is taken of a boundary pixel, is to pad the original image with an all black background, hence any undefined pixel is taken in as black, suppressing the nodes in the neural network. The padding causes some minor distortion in smaller images but is trumped by other filtering inaccuracies for larger images.

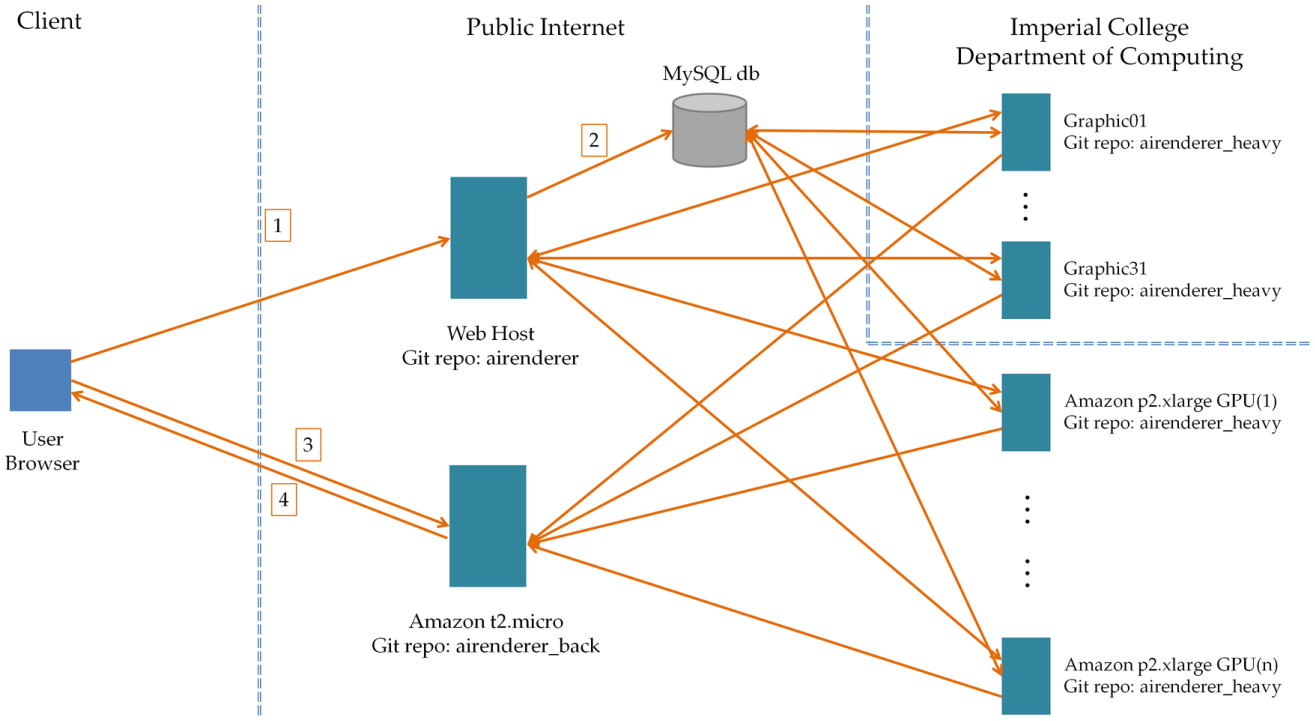

Figure 35: A Distributed Operation Example.

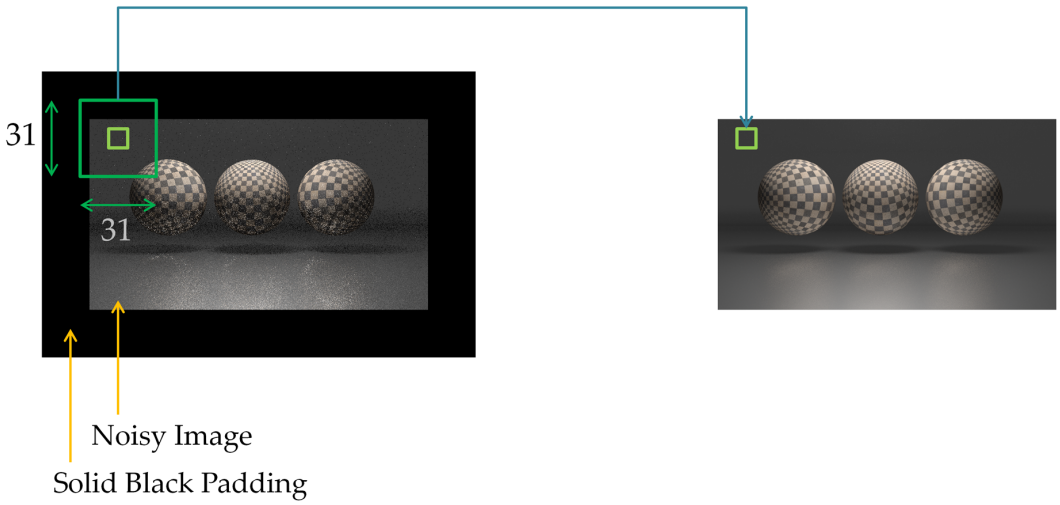

Figure 36: Padding the Input Image.

6 Evaluation and Conclusion

With our objectives from the introduction in mind, we will first try to show that this method indeed speeds up the rendering of real world jobs. Once we are convinced that this has been satisfied, we will then compare the quality of the output for a number of methods when given a very large amount of time, to show that not only we are faster, we are also better than current methods used to clear up noisy renders. The range of test images we will present here includes a number of situations where our filter has performed very good and very bad to provide for an interesting discussion, to see a list of really excellent filter output examples please check out https://youtu.be/UOFa6Jd7NB8. Finally we will present the ratings obtained from user feedback on our site on the public internet.

One new tool we would like to use to evaluate quality is Signal to Noise Ratio (SNR). This is very commonly used to evaluate the strength of the true image against a noisy backdrop. In general the signal to noise ratio[28] is expressed as

$$snr = \frac{\mu_{signal}}{\sigma_{noise}}$$

which we interpreted as a function of the test image

$$snr(test\_image) = \frac{\mu_{gt}}{\sigma_{||gt-test\_image||}}$$

for our particular situation. This means that a noisy image will have a lower SNR than a clearer image. The SNR of the ground truth is infinity as the denominator is zero. One problem with this method of evaluation is that if our ground truth has not truly been rendered to completion, and that there are still some remaining artefacts, then the absence of these artefacts in a filtered image will cause a decrease in the SNR - not what we want. Before we embarked on our evaluation, we ensured that all the noisy images were to our satisfaction, inspecting individual patches of pixels where necessary. The only one gt image that was a problem is from Abandoned Lab where small patches of noise were still stubborn after 2 hours of rendering, so, the snr statistic for this particular test example should be treated with less weight.

6.1 Speed

First we will show that an OpenGL material render is consistently fast to generate across a range of scenes and the amount of time is negligible compared to other renders. For every image in our test set, the time taken to produce has been less than 1 second, hence we will use 1 second as a constant for this purpose.

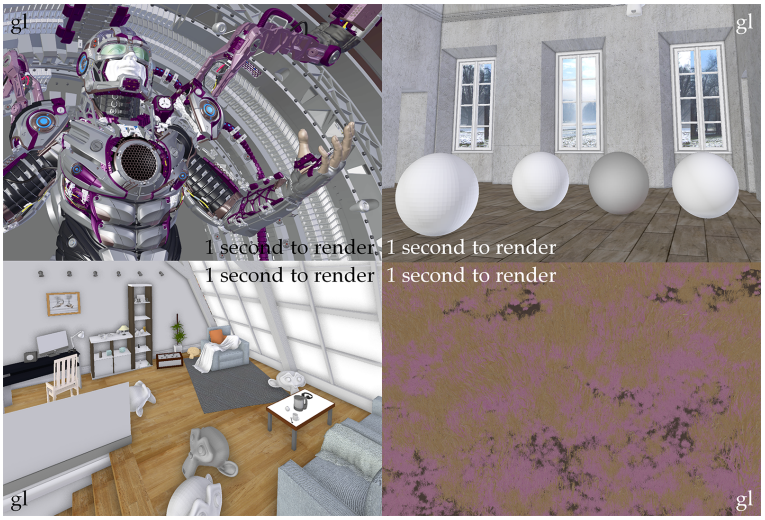

Figure 37: OpenGL material renders are fast to produce.

In order to evaluate the speed, we first take a noisy image and filter it, noting down the amount of time to filter. Then we let the renderer continue rendering from the noisy image for the same amount of time as it took to filter algorithm to run, we will call this the equal time ray traced (ETRT) result. We should note that the timings for the ETRT result has an error bound of ± 10 seconds due to our method - we will place a star next to the statistic for these. If our filtered result is equally or more desirable than the equal time ray traced image in all our tests, then we will be satisfied that we are faster.

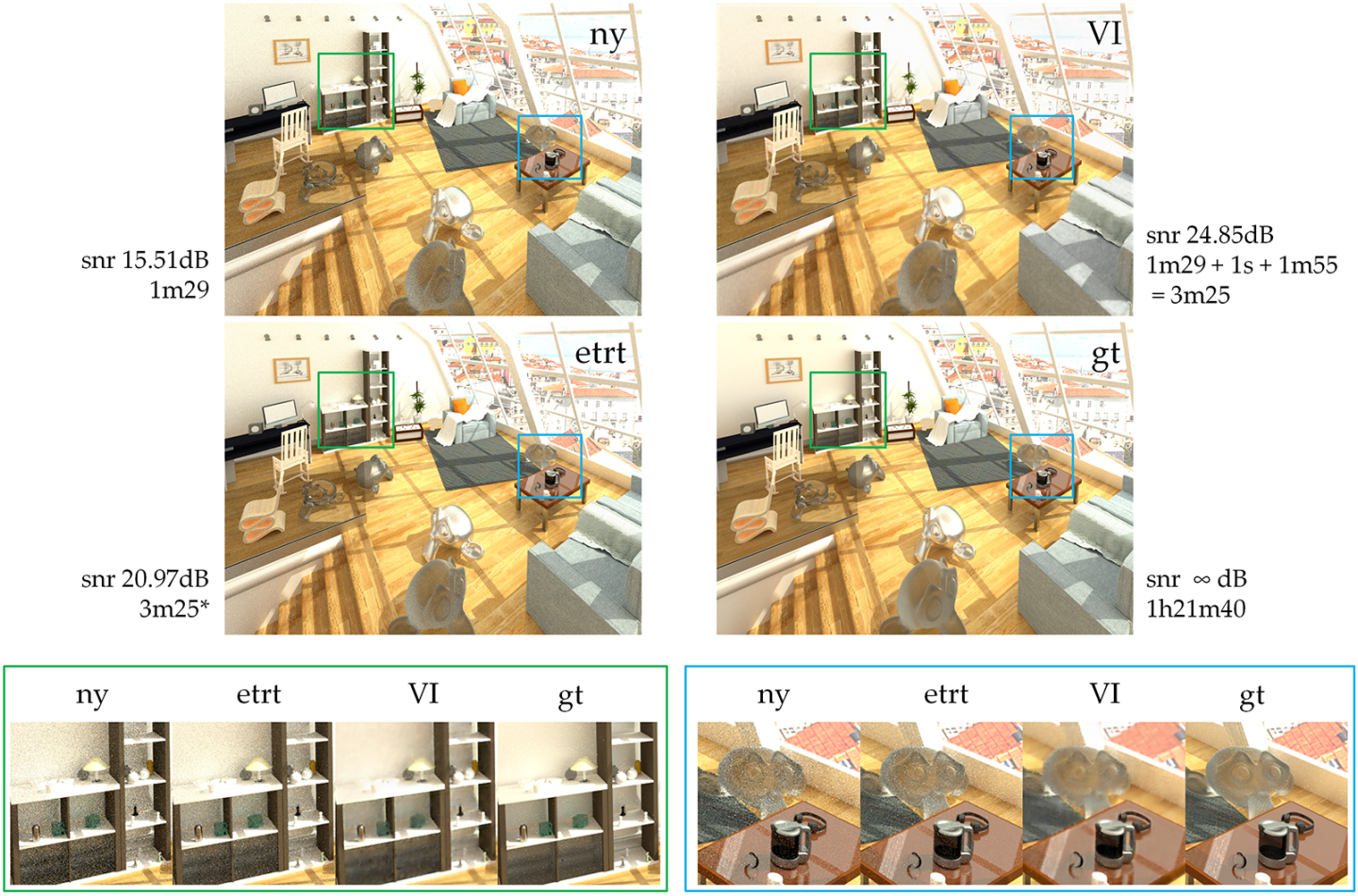

Figure 38: Sunny Room by ermmus (CC-BY).

There was a lot of visible noise that remained on ETRT render. Shadow regions where Monte Carlo algorithms are less efficient were particularly bad. Our output managed to preserve details whilst dousing the visibility of the coarse noise. We were pleased with the amount of detail preservation especially between regions here. The frosted glass monkey head was a place where our output blurred too much - there were some loss of detail around the eyes and more generally between regions of similar colour.

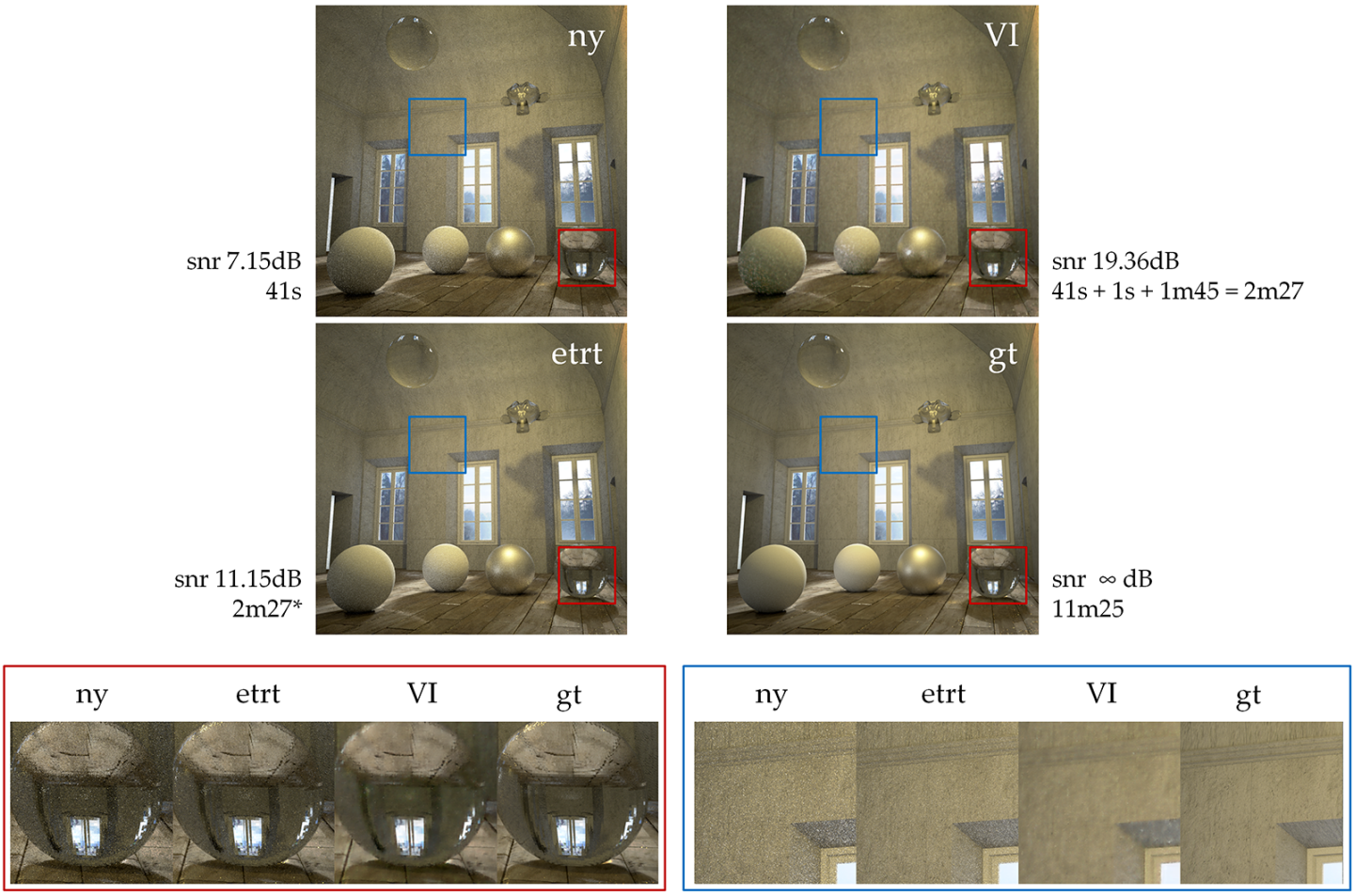

Figure 39: Old Interior by 2Theo (CC-0).

In Old Interior we had a good level of noise removal from the walls - with more of the raw and sharp noise removed than the ETRT render. However, some detail was lost from the walls - we suspect this is because the colour of the pattern on the walls were too faint and similar in colour - prompting an anti-noise response from the neural network. This is a shame. The detail preservation through the glass ball and flooring was excellent however, with all the subtle detail overtly preserved. The ETRT rendition suffered since tracing refractive caustics is computationally expensive. The output from our neural network proved to be much better than the noisy and ETRT at the signal to noise ratio metric.

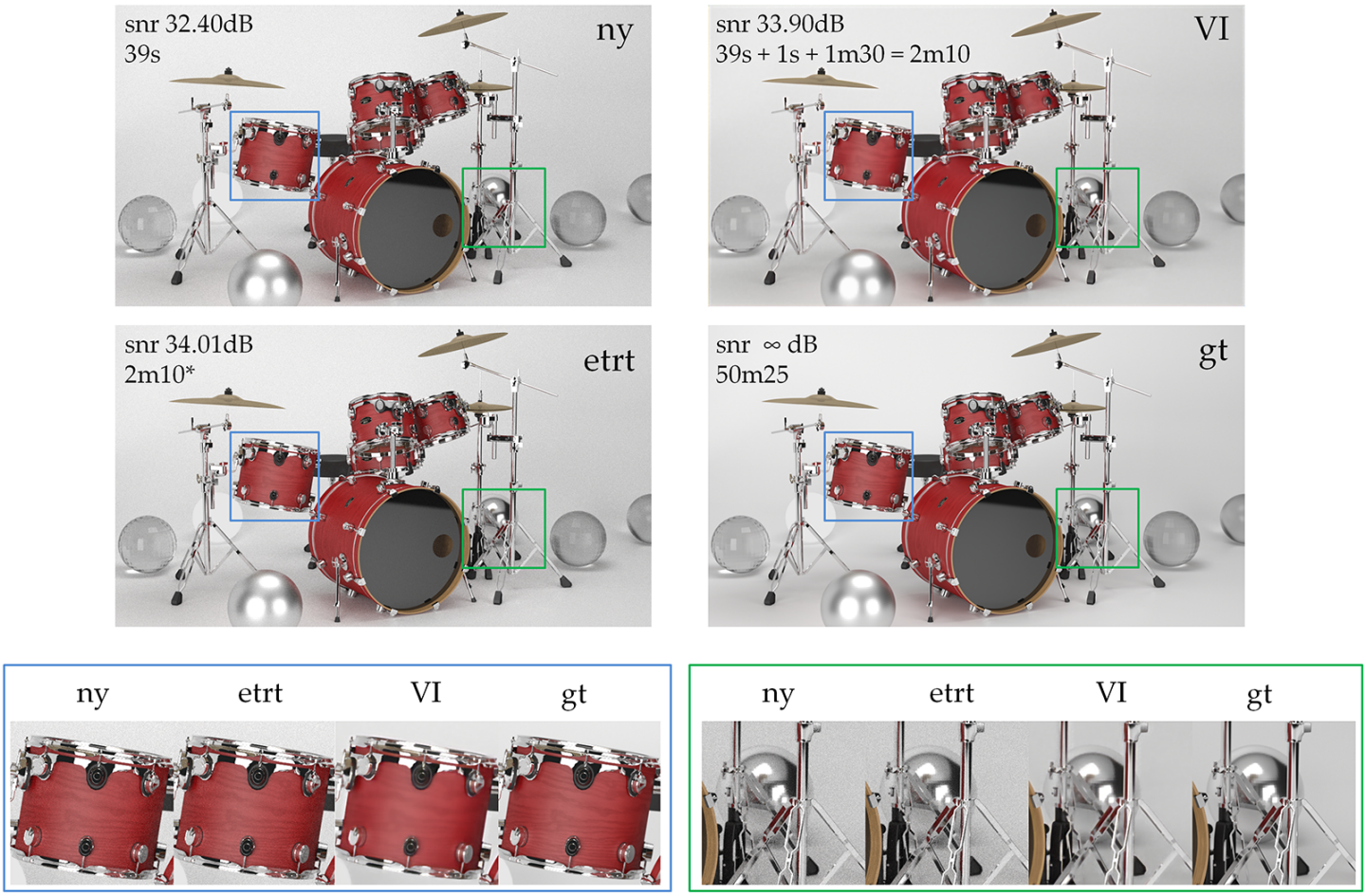

Figure 40: Drums by bryanajones (CC-BY).

One recurring theme so far has been that two regions of very similar colour would have its edges blurred rather badly, and the red drum texture here is another example - although the output is better than the noisy input, it has arguably lost too much detail to be recognized as a wood texture. In contrast, regions with different intensities or colours have had huge success - our output for the drum stand with a chrome ball in the background is practically indistinguishable from the ground truth whereas the ETRT render still has visible noise and distortion. Our output SNR was within margin of error to the ETRT render which themselves are very close to the noisy SNR - we believe that this is because our initial noisy input is of such a high quality, aggressive methods to augment it to remove noise will disturb this high SNR. Here we will say that the accuracy of the ray trace has not been improved by much, but the quality and aesthetics of the output has.

6.2 Quality

We will test the resolve of Model 6 against competing Photoshop filters and the ground truth. Photoshop filters are oftentimes quick to evaluate, sometimes being applied within seconds. However, this is not a contender if too much quality is lost or too little noise is removed. We also let the scenes render to completion for the ground truth image. These should be taken as the ground truth image.

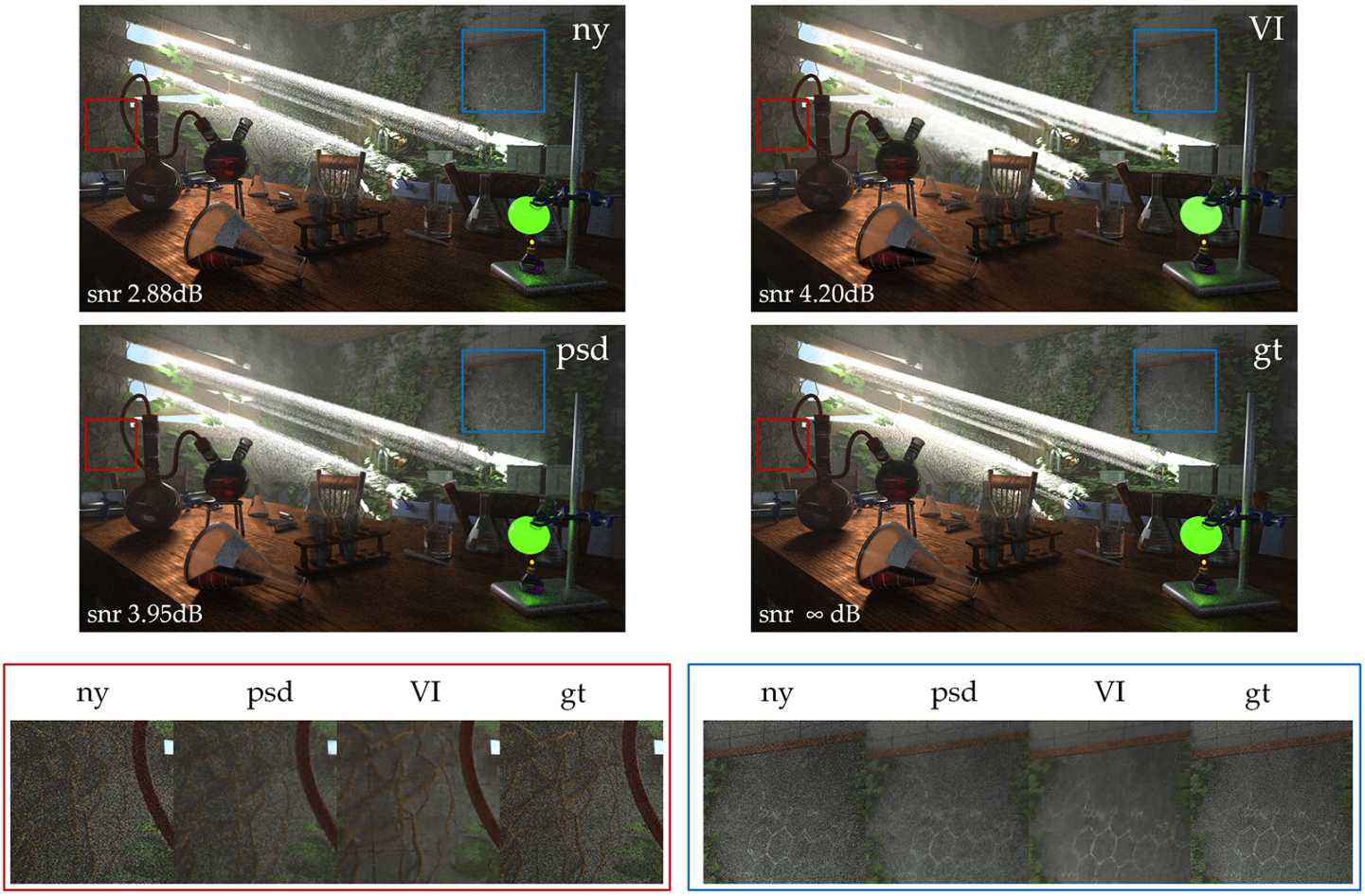

Figure 41: Abandoned Lab by Elysia (CC-BY).

In figure 43 the noisy tree branches patch was done well by Photoshop but the output from Model 6 is absolutely unmatched in terms of both colour quality and noise presence. It even trumps the ground truth result which had already taken considerable amount of time and computational power to produce. The Model 6 output for the blackboard blurred the details of the tiles above the blackboard that was clearly visible in the ground truth render. The writing on the blackboard was also somewhat less legible than the ground truth.

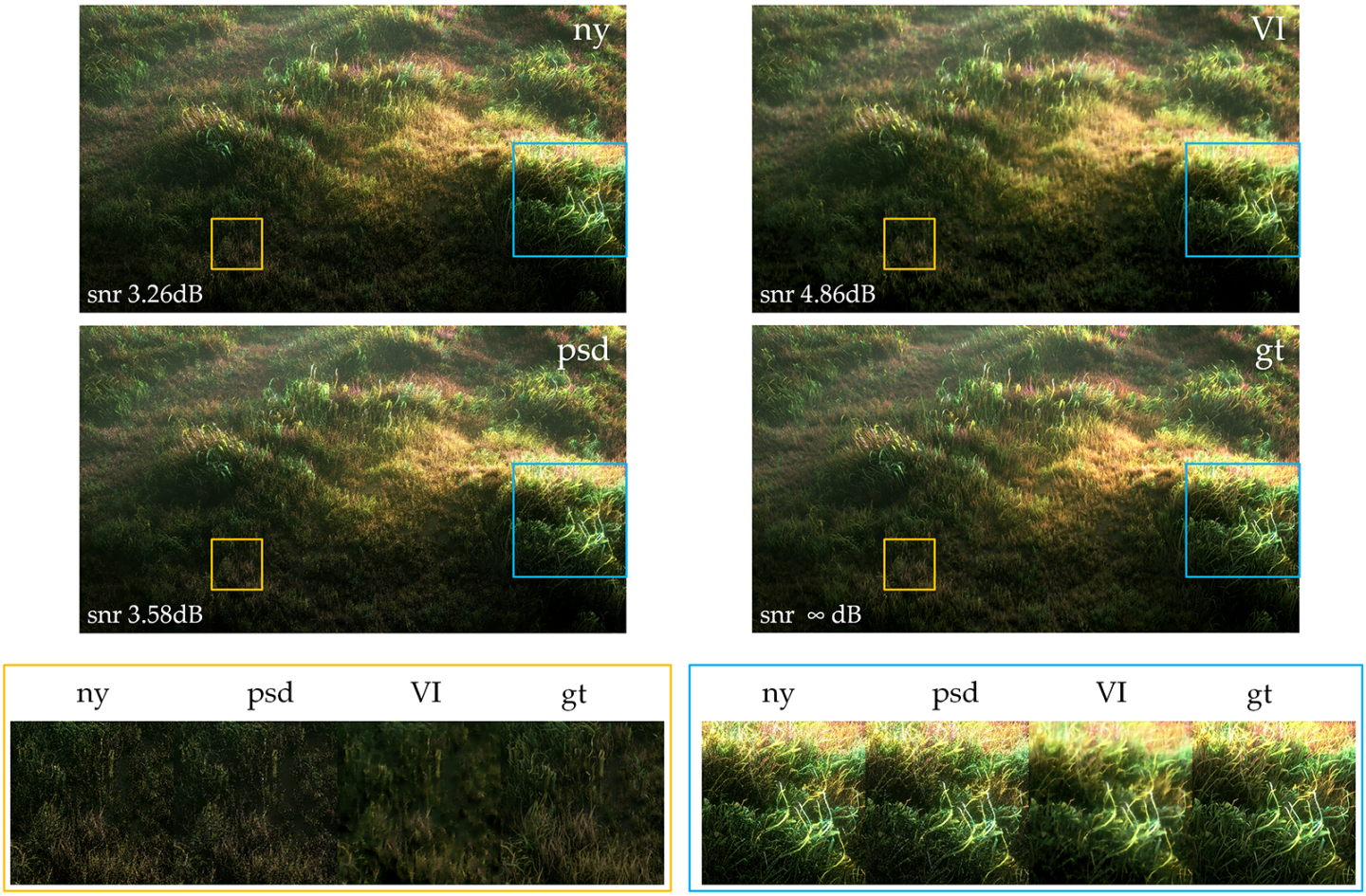

Figure 42: Grassy Field by Spelle (CC-BY).

The removal of noise from this image is impressive. The noisy image is practically carpeted with many types of fine and coarse grain noise and yet our output seems to triumph this to produce a virtually noise free image. However, a main artefact that is in fact produced by the neural network is a very dim haze around thin strands of bright regions. This is an artificial addition to the image that blurs some detail unfortunately. The signal to noise ratio sees a great improvement over the photoshopped version which can only use primitive noise reduction techniques as smart region blur is not applicable here because each strand is too small to be considered a region of uniform colour. In fact, this precludes a large number of scenes with similar detail from having smart region blur applied in Photoshop.

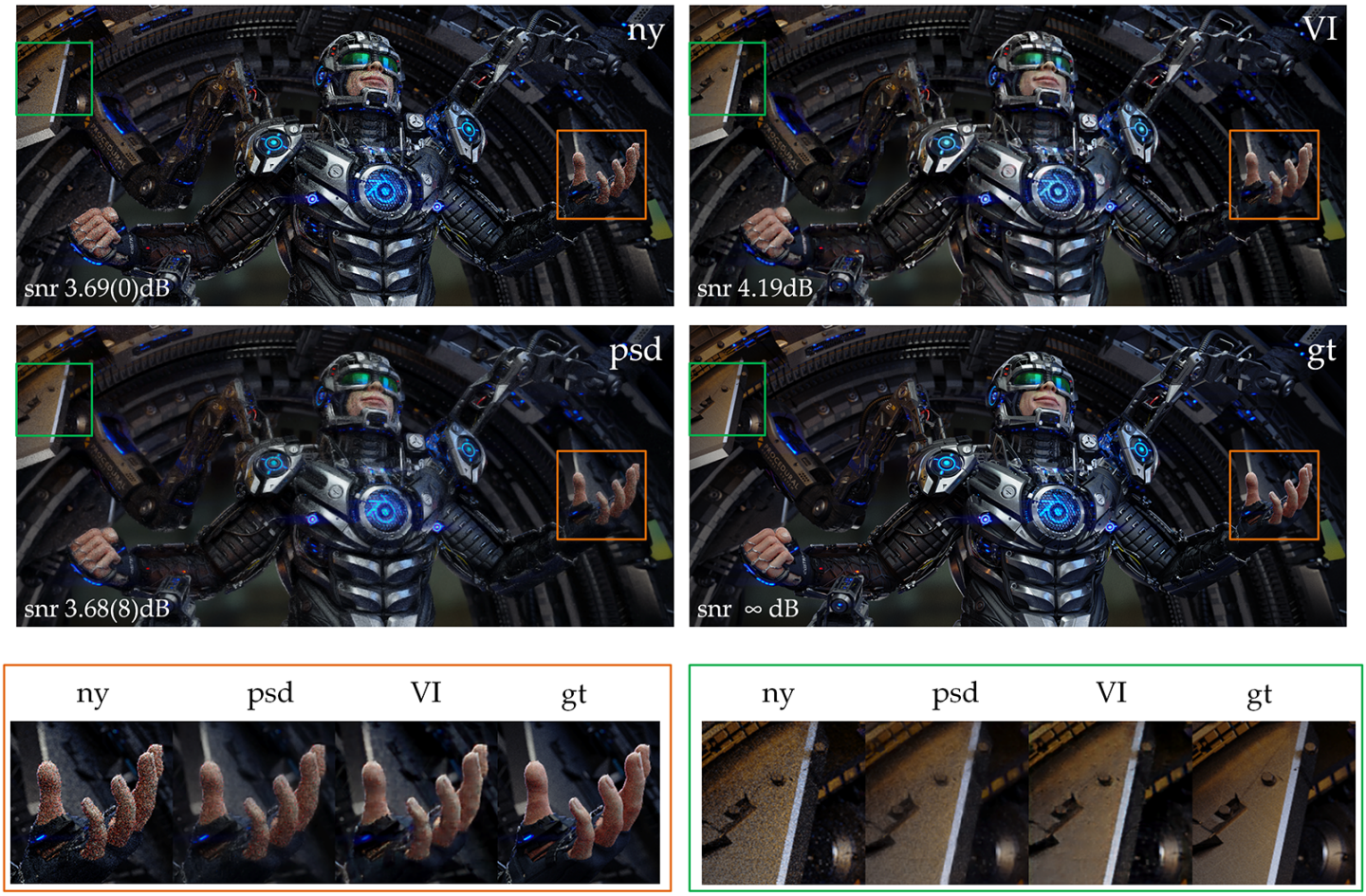

Figure 43: Blenderman by Juri Unt (CC-BY).

In figure 41, we have an indoor scene with glossy surfaces and additional light sources from the scene. There is Subsurface Scattering used for the flesh and full sample Monte Carlo ray tracing. Photoshop is able to remove the grunt of the noise from the hand, however some hazy noise still remain. Model 6 was superior than Photoshop at both noise removal and maintaining contrast, but there certainly is less vibrance than the ground truth. In the control panel patch, both methods were good at maintaining colour quality, however, Photoshop still suffered a slight overlay of grain on the bright parts of the image. Whilst Model V proved effective at de-noising large patches of consistent colour on the control panel, there is a small amount of visible noise in intersections between regions of colour.

6.3 Preservation of Detail

We also would like to see explicitly how well detail is preserved in our neural network. To do this, we will replace the ny input with the gt input and see explicitly just how much detail is lost. If our neural network has done a good job, then our prediction is that the noisy output should be the same as the ground truth input.

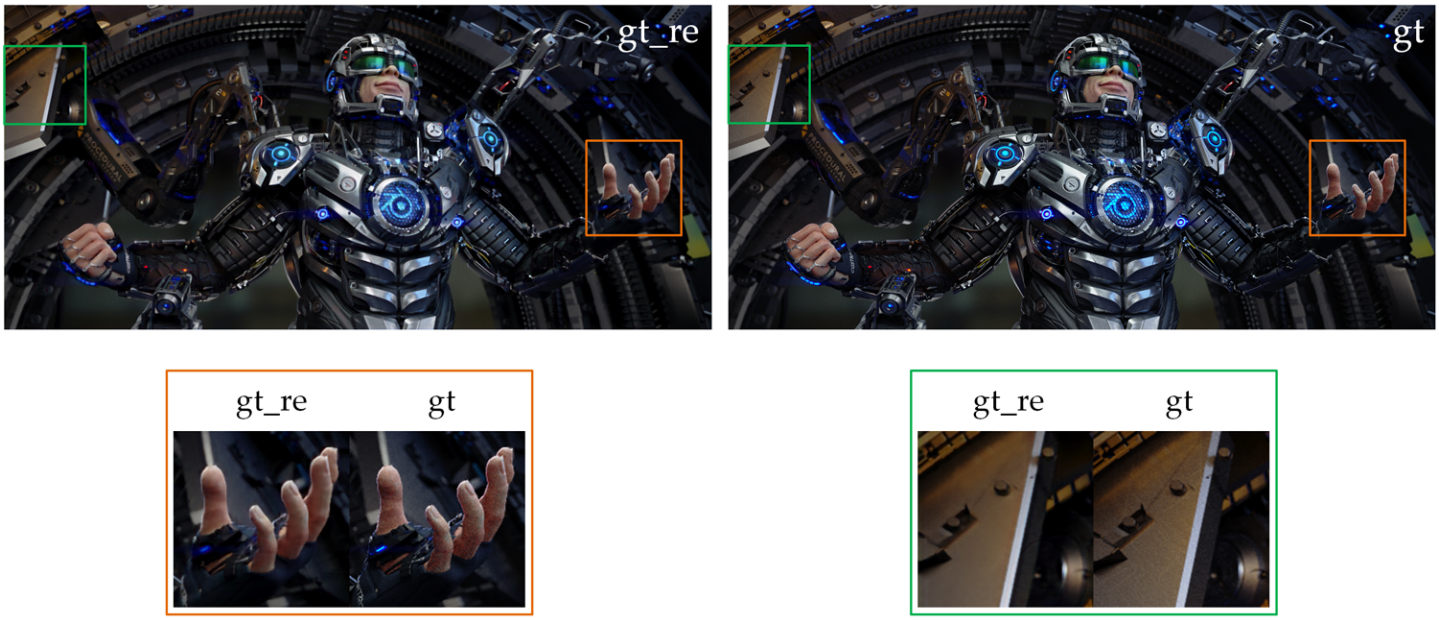

Figure 44: BLENDERMAN by Juri Unt (CC-BY). gt_re is the filtered ground truth output.

We were wrong. The neural network proceeded to filter the ground truth image, removing the final patches of noise from an already very clear ground truth render. The end output from the neural network is actually better than the ground truth input! There is some very minor loss of clarity near the honeycomb badge on the centre of his chest and the fan grille in the background. This is definitely not desirable, and we would certainly want to ensure this doesn't happen in a future iteration. But I must stress that the detail loss is very minor - with the two images side by side, one cannot notice the difference - it is only when you scroll between the images on a full resolution screen do you notice this.

6.4 Effectiveness of the OpenGL Render

We would like to see explicitly just how much of an effect the OpenGL pass has on the filtering ability. We will now compare two filtered images, one with its OpenGL render used as the OpenGL input, the other with the noisy image used as the OpenGL input.

We find that the image that has been supplemented with an OpenGL input is far more efficiently filtered. In figure 45, you can see that in the green patch, the numbers 2.78 is preserved well in the filter that used an OpenGL supplement and the numbers are almost illegible in the filter that did not use the supplement. In the orange patch, the mesh in the OpenGL supplemented output is far clearer and sharper than the other. However, there was still some denoising compared to the psd output which attained a snr of 3.70 dB, lower than the 3.94 here.

Considering the speed of production of OpenGL material renders as shown in figure 37 and

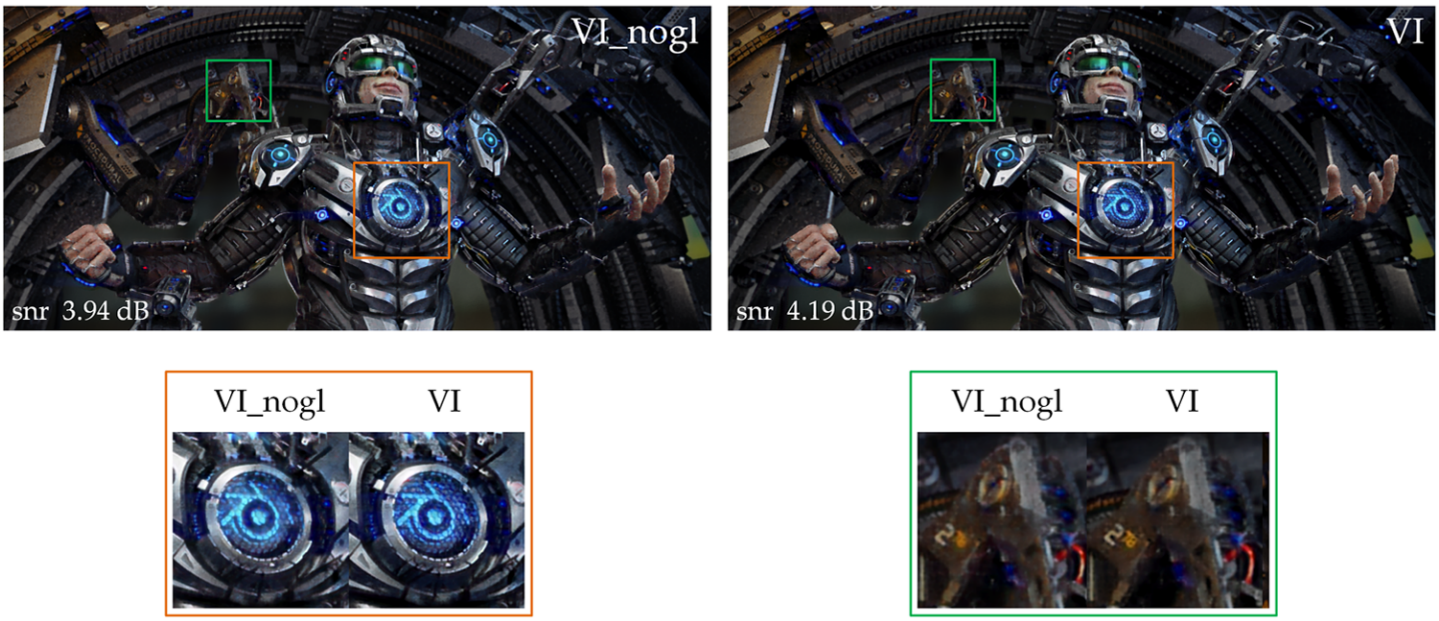

Figure 45: Blenderman by Juri Unt (CC-BY). VI_nogl is the no OpenGL input pass result.

the fact that every 3d renderer has the ability to produce this pass, it is fantastic that we decided to include this in our neural network. Other authors in our background section often opted to include many more, often rather ambiguous passes at the expense of speed or accessibility. We also note that on the areas where the OpenGL image did not provide much information, such as on the hands, we did not notice a discrepancy between the two outputs - so it seems like where the OpenGL is unable to provide information, there is no improvement in quality.

6.5 The Trial Period

We launched our app on the public internet on 26th of April 2017. Seven days later we implemented an update - a feedback system for people to rate their results. The scale we used was out of 5 where 5 is "Excellent", 4 "Good", 3 "Acceptable", 2 "Bad" and 1 "Joke". Over this period of time until the 1st of June 2017, we had over 1000 visits to view our website, 108 uses of the app and 26 ratings left.

The ratings were initially rather disappointing, however under close inspection, it turns out that some of the jobs uploaded were users using our app improperly. A lot of these were people uploading the same image for the noisy and OpenGL inputs. Although this was suggested to novices in the quick start guide who did not know how to produce an OpenGL render in their own software packages, it was not meant to be used in a production environment as professional users of any 3d software package should be able to produce this OpenGL pass blindfolded. Please refer to section 6.4 for the importance of having an OpenGL render. We will remove the ratings given by users incorrectly using our app (shown in grey in figure 46) as we would like to simulate usage by a professional who was proficient in his rendering workflow. Overall, we have an average rating of 3.533/5.

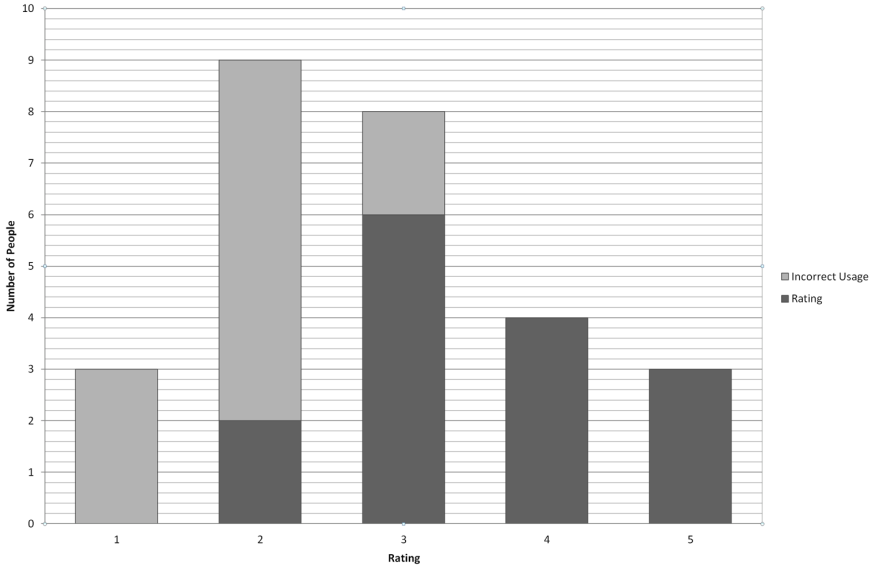

Figure 46: Accumulated Ratings.

6.6 Addressing Challenges

I want to address the challenges I opened with in the Introduction. Neural networks are difficult to understand, and having to architect a new shape for a new task was daunting. However, this turned out to be one of the most enjoyable parts of the project. Early on, I developed a methodical approach for me to follow when evaluating architectures, which in the end would teach me the subtle intuition associated with architectural design. The method was this. First, always peek at what the output looks like in the early epochs. This gives you invaluable information of how well you have tuned your initiation parameters. Most of the time we wanted the identity mapping with some noise and this step tells you the success of the amplitude and variance of your noise filter. An example of this is figure 28. Second, look for subtle hints of overfitting - if an image of the test set degrades in quality later than a previous point in time, then we are likely to have some form of overfitting. Overfitting is confirmation that our network size is large enough and that we to simply vary our method. Finally, when you have branches in inception modules, it can be difficult to ascertain how each branch might be affecting the output. In this case, it is perfectly good to suppress the other branches and again, peek at the output. For example, in 29, we have a branch with 2 convolutions of rectangular stride sizes. If we wanted to know what each convolution did, then we simply replace the whole module with just 1 rectangular convolution.

Figure 47: A Vertical Convolution Applied to a Test Image.

The result of this replacement is shown in figure 47, when a vertically orientated convolution layer is applied to image. The output shows good preservation of detail along the side of the pipe, but the noise pattern turns into a nasty vertical looking disarray. This is what prompted me to

follow this up with a horizontally rectangular convolution. These techniques allowed me to peek at the concealed inner workings of neural network modules and to better understand how to handle the machinery.

Keeping the system general with respect to the initial amount of noise in the image turned out to be manageable. Two ingredients were needed - a large enough neural network with many parameters and a training set that features coarse grain noise and gentle noise. An example of coarse grain noise being aggressively filtered is Caustic Flower in figure 32, and an example of detail preservation through gentle noise is the whiteboard patch of Abandoned Lab in figure 41. We showed in section 6.3 and the drums example that if we feed in a image with very little noise, then almost all the detail is preserved and in some cases, the final noise is removed giving us a result superior to the ground truth. For more noisy images, we realise that there is inherently less information in a noisy image, and since being able to create fine detail is difficult and biased, this would not be our intention. However, occasionally we may salvage some of this detail from the OpenGL input, in which case, the extra information is certainly made use of by the neural network.

One of the fears that I had before the projects - which did become a problem - was the choice of error function for this task. Sure, the error function served us well in training the neural network and evaluating the efficiency of training from loss-epoch graphs. But the final model that we were most satisfied with had quite a high error! We were lucky that in this scenario we could use our qualitative judgement to evaluate the best filter by eye but it certainly would have been nice to find an error metric that is effective at training and evaluating a neural network for this task.

6.7 Final Thoughts

To finish, I just want to conclude by saying that this project has gone reasonably well in my opinion. I had achieved much of what I set out to do in the objectives section and some of the test set images have been filtered very very well. However, it is not a complete or universal solution. Given a bad scene with bad lighting and and a bad combination of mesh locations, the filtered output can be around the same or even slightly less clear than the equal time ray traced result. This can also occur if the scene is very simple with only a few geometries. Sometimes some detail is simply not preserved for no good reason, for example in figure 40. So we have a few proposals for a more universal approach to the problem, in the next chapter.

7 Future Work

Many Neural Networks

One of the challenges we had was to filter noise in different scenes. For example given a noisy image of a uniformly coloured table, we would like all the detail of the noise to be removed - this is best achieved by some form of smoothing or blurring filter. On the other hand, say we had a scene of a dense forest bursting with leaves of every colour. In this scenario, using the same blurring filter would have disastrous consequences. Throughout the project, there was a subtle power play of whether we should go for more blur - for Horizon, Photosynthesis and Firefly Forest - or more emphasis on the preservation of detail in Similar Edge, Faint Line and Colour Edge.

One possible solution for this problem is to train a neural network to determine whether a pixel is in a region of roughly uniform colour or in a region of fragile edges that must be filtered with care. We can then use the output of this neural network to determine which neural network to use in order to filter the image. This is great because we can then train many neural networks that are more specialized in filtering blurry pixels versus high contrast pixels versus firefly pixels. We may find that the filtering resolve of the neural network that filters high contrast pixels is the same as this neural network, but at least, if the other neural networks perform better, then we will end up with a better overall result.

If this were successful, then we could even take it a step further by training our initial classifier to recognize environments and then eventually specific objects. Imagine, the first neural network pass could be to determine the type of scene, for example, nature versus architecture versus food versus humans etc. Then if say nature were to picked, we could have another classifier neural network to determine whether it is a tree or an animal or mostly sky... So we would have more specializations until we can describe that pixel well, and then we would put it into the relevant neural network to be filtered. Needless to say, this would require us to train hundreds of neural networks for denoising specific objects, making the creation of training sets rather long.

Adding Artificial Detail

We must admit, that despite our best efforts, our final image will never be as clear or as accurate as the original image. This is because for any two images A and B, if A has more noise than B, then B will contain more information than A. We say A has a higher entropy. Thus the discrepancy of the amount of information contained in A and B sets the upper bound for how close we are able to reach the clarity of B. This is the reason we used material outputs from the OpenGL pipeline to decrease the overall entropy of the noisy system. If we do not have information beyond this, then one option may be to perform in-painting in regions where noise removal has left the image bare.

Neural networks have had much recent success at in-painting[29]. The usual test setup is to remove a small rectangular portion of an image and let the neural network deduce what the colour intensities in the removed portion should be. This is what we need! We simply need to mark patches of the filtered image for regions where we have a lot of uniformity and run an in-painting neural network on these patches. It may be that the uniformity was intended in the original scene, in which case, the in-painting network should just leave it alone. This technique would certainly add detail to the final output, at the expense of adding bias.

It would be interesting to see what would happen if we concatenated the denoising neural network to the in-painting neural network, initialized to their respective trained weights and then train the whole neural network.

References

[1] Wikipedia, "Timeline of computer animation in film and television," *https://en.wikipedia.org/wiki/Timeline_ of_ computer_ animation_ in_ film_ and_ television.*

[2] T. S. Disney, "Available at https://lumiere-a.akamaihd.net/v1/images/open-uri20150422-20810-m8zzyx_5670999f.jpeg?region=0,0,300,450," *Accessed 2017*, 2017.

[3] Daniel Pohl, "Quake wars gets ray traced," *https://software.intel.com/en-us/articles/quake-wars-gets-ray-traced/.*

[4] J. F. Per H. Christensen *et al.*, "Ray tracing for the movie cars," *IEEE Symposium on Interactive Ray Tracing*, no. Available at https://graphics.pixar.com/library/RayTracingCars/paper.pdf, pp. 1–6, 2006.

[5] Zichen Liu p. Available on request., 2017.

[6] Zack Waters, "Area light vs point light," *http://web.cs.wpi.edu/ emmanuel/courses/cs563/write_ ups/zackw/realistic_ raytracing.html.*

[7] James Kajiya et al., "The rendering equation," *http://cs.brown.edu/courses/cs224/papers/*, vol. Brown University, 2011.

[8] Wikipedia, "The rendering equation," *https://en.wikipedia.org/wiki/Rendering_ equation.*

[9] J. T. Kajiya, "The rendering equation," *http://dl.acm.org/citation.cfm?doid=15922.15902*, vol. SIGGRAPH, 1986.

[10] Wikipedia, "Path tracing progressive refinement," *https://en.wikipedia.org/wiki/Path_ tracing.*

[11] Michael Nielsen, "A standard multi-layer perceptron neural network," *http://neuralnetworksanddeeplearning.com.*

[12] P. P. Zichen Liu *et al.*, "Facial emotion recognition by convolutional neural networks," 2017.

[13] Christian Szegedy et al., "Going deeper with convolutions," *2015 CVPR*, vol. Google Inc. and elsewhere, 2015.

[14] Ruifeng Xu et al., "A novel monte carlo noise reduction operator," *IEEE Computer Graphics and Applications*, vol. University of Central Florida, 2005.

[15] Nima Khademi Kalantari et al., "Removing the noise in monte carlo rendering with general image denoising algorithms," *EUROGRAPHICS 2013*, vol. University of California at Santa Barbara, 2013.

[16] Harold C. Burger et al., "Image denoising: Can plain neural networks compete with bm3d?," *http://people.tuebingen.mpg.de/burger/neural_ denoising/*, vol. Max Planck Institute for Intelligent Systems, Tubingen, 2012.

[17] Viren Jain et al., "Natural image denoising with convolutional networks," *Advances in Neural Information Processing Systems 21*, vol. Massachusetts Institute of Technology, 2008.

[18] Nima Khademi Kalantari et al., "A machine learning approach for filtering monte carlo noise," *SIGGRAPH, http://nkhademi.com/Papers/SIGGRAPH15_ LBF_ LoRes.pdf*, vol. University of California at Santa Barbara, 2015.

[19] The Blender Software Foundation, "Blender 3d," v2.78a.

[20] Alex Krizhevsky, "Learning multiple layers of features from tiny images," *http://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf*, 2009.

[21] Wikipedia, "Kernel," *https://en.wikipedia.org/wiki/Kernel_ (image_ processing).*

[22] Matthew Zeiler et al., "Deconvolutional networks," *https://pdfs.semanticscholar.org/10b8/3e269c1dcb171ae678e28322997f48e2ee6c.pdf.*

[23] Hyeonwoo Noh et al., "Learning deconvolution network for semantic segmentation," *http://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Noh_Learning_Deconvolution_Network_ICCV_2015_paper.pdf*, 2015.

[24] W3Schools, "Php tutorial," *https://www.w3schools.com/php/*.

[25] NodeJS, "fs tutorial," *https://nodejs.org/api/fs.html*.

[26] NPM, "Package sources," *https://www.npmjs.com/*.

[27] HTML5UP, "Spectral, stellar," *http://www.html5up.net*.

[28] Wikipedia, "Signal to noise ratio," *https://en.wikipedia.org/wiki/Signal-to-noise_ratio_(imaging)*.

[29] Alhussein Fawzi et al., "Image inpainting through neural networks hallucinations," *http://www.cs.toronto.edu/ horst/cogrobo/papers/ivmsp2016.pdf*, vol. IVMSP, 2016.

Appendix

Machines Used to Obtain Results

We used Tensorflow as our deep learning framework for operating tensors. We trained our small models on a Nvidia GTX 960m and larger models on a Nvidia GTX 1080 (Graphic01). Evaluation of images for the web app happened on various nodes including a Nvidia GTX 1080 (Graphic01) and Nvidia Tesla K80 (Amazon p2.xlarge). Creation of the dataset occurred on various i7 CPU machines that were free in labs.

Acknowledgement of Source Code Used

Repository - deep_filter

- In files *tfrf.py* and *dataProvider.py* we used the ideas related to setting up the training and test datasets and the logging and graph plotting capabilities. No code was directly copied. The original files came from a paper by Zichen Liu for emotion classification of faces, the source code can be made available on request by the author.
- Note that all test patches used throughout this report are unreferenced here, but acknowledged in the code base submission repository.

Repository - airenderer

- All files in *assets/* and *help/assets/* are library files taken directly from their respective authors.
- The files *elements.html*, *generic.html*, *help/elements.html* and *help/generic.html* are taken directly from the source code by the author Html5up at html5up.net.
- The files *index.html* and *help/index.html* are from Html5up but adapted with my content.
- The file *app/test_input.php* is from W3schools entirely.
- In the file *app/index.php*, lines 87-96 are from Google to provide analytics tracking.
- In the file *app/welcome.php*, lines 82-168 are adapted from W3schools to purpose.

Repository - airenderer_back

- In *index.js*, the code between lines 27-32 was adapted from the npm http site to purpose.
- In *index.js*, the code between lines 52-54 was adapted from the npm mv site to purpose.

Repository - airenderer_heavy

- In *index.js*, the code between lines 19 and 23 were adapted from the npm request site to purpose.
- In *index.js*, the code between lines 151-156 was adapted from the npm restler site to purpose.

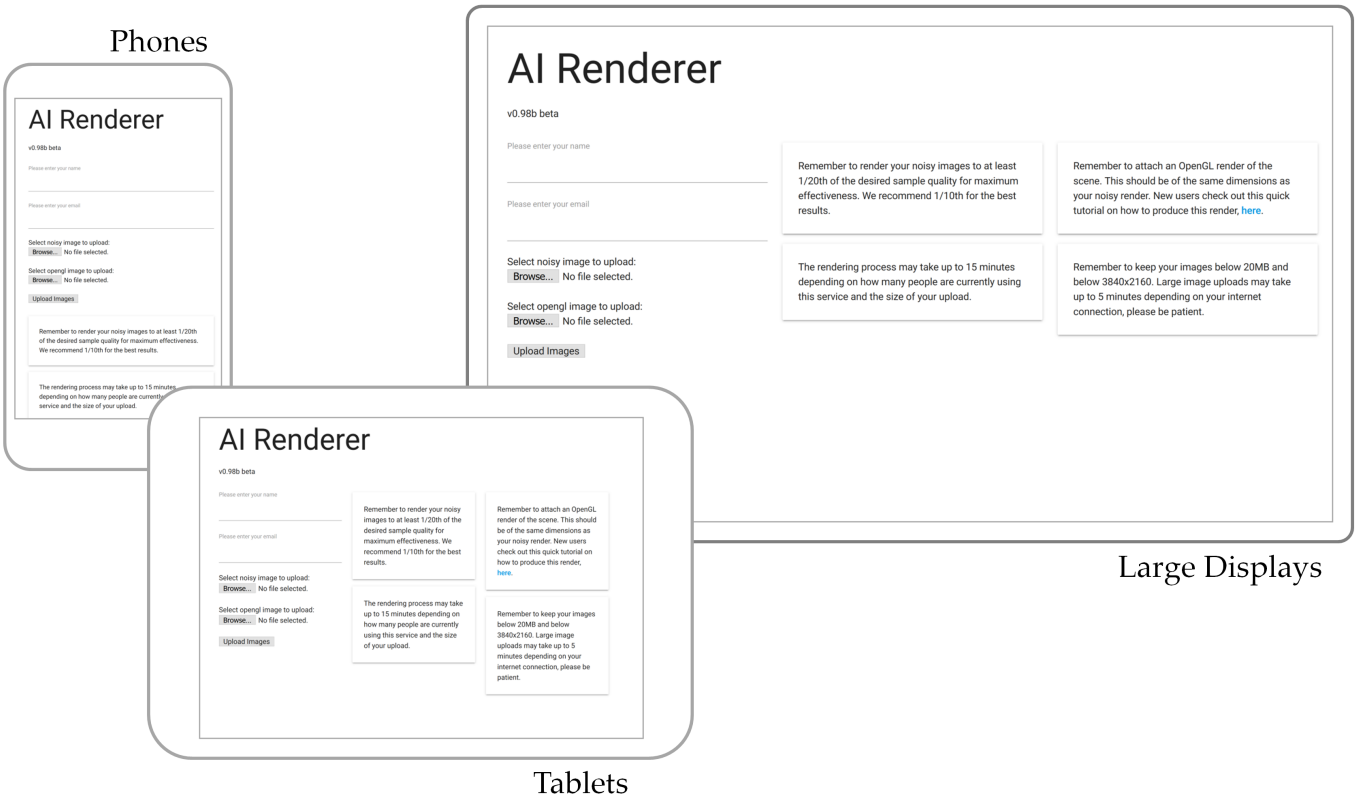

Figure 48: Screenshot of the app page.

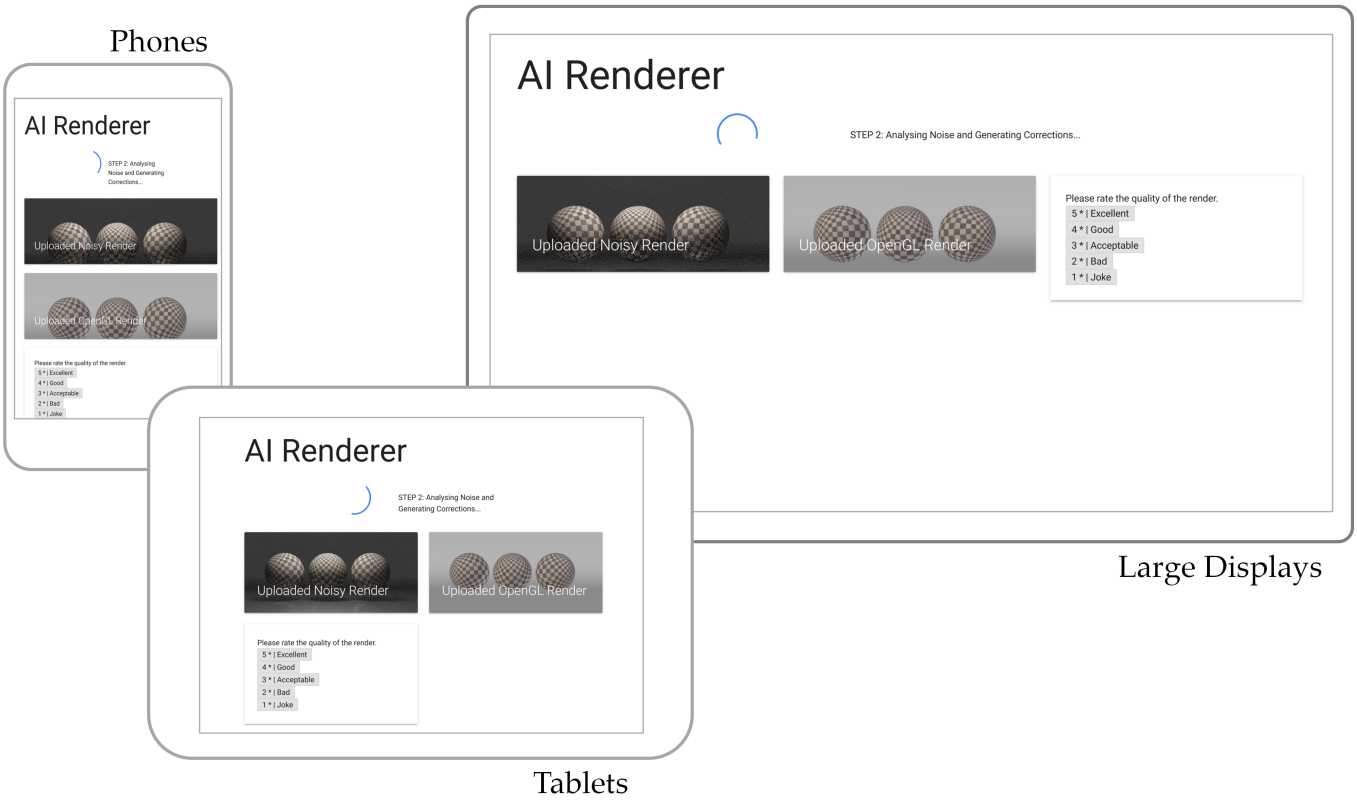

Figure 49: Screenshot of the app performing filtering operations.

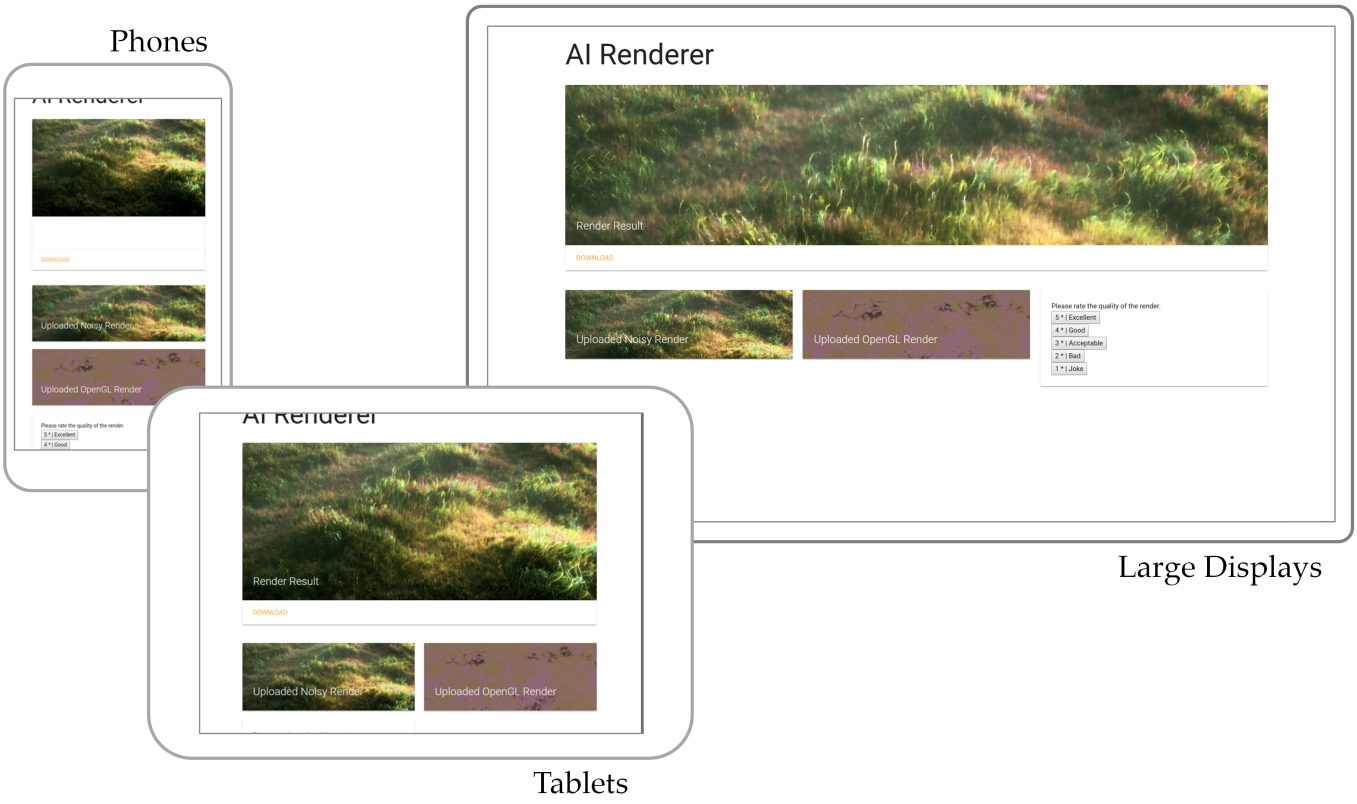

Figure 50: Screenshot of when filtering operations are complete.

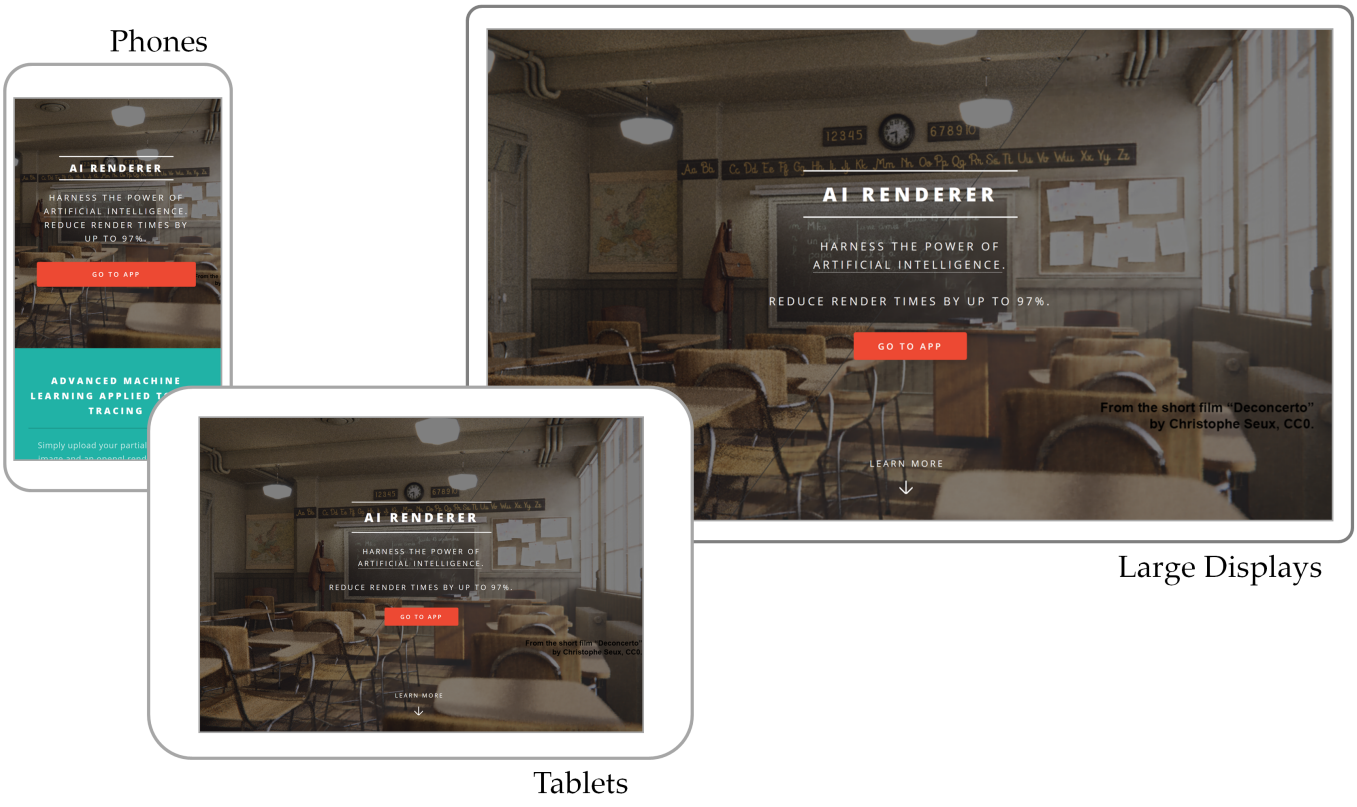

Figure 51: Screenshot of the landing page.

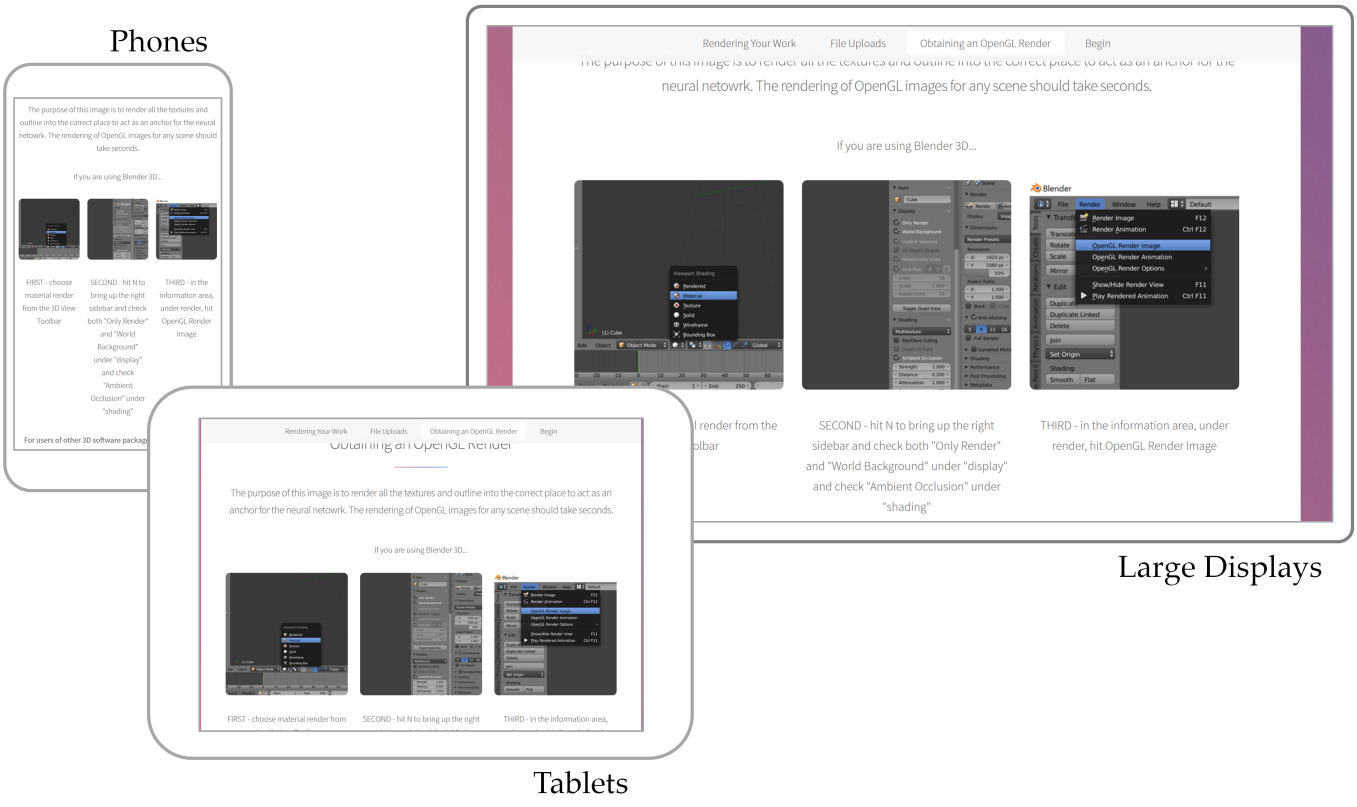

Figure 52: Screenshot of the tutorial guide.

Quickstart Guide – Reproduced from airenderer.com/help

Rendering Your Work

We recommend that you render to around 1/10th your desired sample size,
so if you want a render of 2000 samples, render your noisy image to 200 samples.

Uploading Your Work

Two files will be required. A noisy render and an opengl render.
The two files must be both of PNG or JPG type.
The two files must each be below 20MB.
The dimensions of both files must be at least 100x100 and at most 3840x2160 (4K Resolution).

Obtaining an OpenGL Render

The purpose of this image is to render all the textures and outline into the correct place to act as an
anchor for the neural netowrk. The rendering of OpenGL images for any scene should take seconds.

If you are using Blender 3D...

FIRST - choose material render from the 3D View Toolbar
SECOND - hit N to bring up the right sidebar and check both "Only Render" and "World Background"
under "display" and check "Ambient Occlusion" under "shading"
THIRD - in the information area, under render, hit OpenGL Render Image

For users of other 3D software packages, please generate a suitable OpenGL materials render with Ambient Occlusion.

In the circumstance that an OpenGL image is not representative of the actual output, for example volumetrics or strand renders
for certain software packages, then it is actually more beneficial to upload the noisy image again, in lieu of the OpenGL render.

Figure 53: Reproduction of the quickstart guide from airenderer.com/help.