Methods for Using the Deep Image Prior

Oliver Newcombe

Supervised by Rihuan Ke
Level 6
10 Credit Points
I confirm I have complied with guidelines as outlined in the project handbook.

December 14, 2022

Acknowledgement of Sources

For all ideas taken from other sources (books, articles, internet), the source of the ideas is mentioned in the main text and fully referenced at the end of the report.

All material which is quoted essentially word-for-word from other sources is given in quotation marks and referenced.

Pictures and diagrams copied from the internet or other sources are labelled with a reference to the web page or book, article etc.

Signed 

Date 14/12/2022

Contents

1 Introduction

In the image processing field it is a very common task to recover an image that has been transformed or degraded somehow, in what is known as an inverse problem. This class of problems comes with a range of issues to account for, particularly ill-posedness, where there is not a unique solution, potentially due to information being lost in the degradation process [3]. As such, a common method is to provide some prior information about our problem to assist in finding an accurate solution. This could be in any of a large number of forms, such as an explicit formula or manual changes to our process to find a solution. However, in general, these 'hand-crafted' methods are poor at generalising, so many in the field have turned to a more adaptable approach, which uses Artificial Intelligence (AI) to find a solution. These methods attempt to *learn* how to solve our problem by giving a Deep Neural Network (DNN) model prior knowledge in the form of a training dataset. However, a recent advance does away with this training, utilising DNNs in a completely different manner. This Deep Image Prior (DIP) method claims we can provide prior knowledge simply via the DNN itself [1]. This shall be the focus of the report, and in this introductory section we will give a high level overview of the general problem setting, explore other approaches for context, and then address the content and structure of the report.

1.1 Brief Problem Overview

Given an objective original input x^*, that we want to recover, we can model our noisy measurement y as being of the form

$$y = A(x^*) + \epsilon, \tag{1}$$

where A is a linear transformation/degradation operator, often written as a matrix, and ϵ is a random noise term, of any distribution [2]. We cannot simply solve this system of equations as our

noise term is random. In this situation, we need to find an x_0 to approximate x^*. If we assume that we know this operator A as a matrix, we could attempt the following,

$$x_0 = A^{-1}(y). \tag{2}$$

However, for the majority of cases, our problem is ill-posed, so we cannot find a unique solution to this. For example, if our image has lost pixels via some physical process, we would need to recover their values, but we now have more unknown variables than known. This leads to our system having infinite solutions, meaning we cannot recover this estimate of our unique original image [3]. In any case, even if our system was not ill-posed, inverting a large matrix is in and of itself an exceedingly inefficient task. Thus, we search for a numerical method. This leads to the formulation of the following optimisation problem,

$$x_0 = \underset{x}{\operatorname{argmin}}\, E(x; y), \tag{3}$$

where E is some loss/data term, dependent on the data we are working with. An illustrative, but naïve and ineffective, example would be to use the ℓ_2 loss, that is,

$$x_0 = \underset{x}{\operatorname{argmin}}\, \|A(x) - y\|_2^2. \tag{4}$$

This formulation, however, leads to overfitting, meaning that we would fit our model too closely to the data, so that it is unable to generalise to unseen examples well. As an example, in regression we could fit a model exactly to our n data points, by simply fitting a n-th order polynomial that goes exactly through each point. Our loss would be minimised or 0, but any new datapoint would not be predicted with any level of accuracy. In our case, the minimising value of 0 is achieved at $x = y$ for most standard loss terms, which, in this problem, is not desirable; we already have our noisy image y so this provides little information. We will return to this later though, as there is some information to be gained in the optimisation process itself. There are numerous ways we can combat this issue with the loss function, including noise modelling, regularisation and a subset of it, early stopping [4]. We will be focusing on the latter two.

Regularisation techniques aim to adjust our optimisation, such that desirable qualities of an estimate remain, whilst other, less desirable ones, are left out. For example, if an image has been corrupted, such that there are a range of coloured pixels that seem out of character for the original, we may wish to penalise estimates that show large deviations in pixel values. We will return to this formally later. A standard regularisation approach is to add a regularisation term to our optimisation problem above, giving us,

$$x_0 = \underset{x}{\operatorname{argmin}}\, E(x; y) + R(x), \tag{5}$$

where R is what we refer to as an regulariser, or for our purposes later, an (image) prior [5]. This could be an explicit function, referred to as a hand-crafted prior, for detecting specific attributes of an image e.g. vertical lines.

If we were to write our prior as $R(x) = \lambda g(x)$, then we can see this is a general form, which many popular estimators take, e.g. Ridge Regression [29] and Lasso [30], where g is the ℓ_2 and ℓ_1 norm respectively. This is an example of sparsity regularisation, where we wish to reduce overfitting by penalising complicated solutions, so that our solutions are in some sense 'sparse' [10].

However, creating a useful prior expression by hand for more complicated tasks, other than regression, is a very difficult exercise and requires more thought. For example, we may want to extend the ideas of the ℓ_1 and ℓ_2 norms to better suit our problem, with a more detailed sparsity regulariser. In our setting of image processing, we are trying to preserve patterns within our image, e.g. consistent colours and edges, whilst doing away with inconsistencies e.g. noise. As such, if

we can define a term that can qualify this consistency, we could use it to regularise our method to avoid overfitting. One such method that is regularly used in this field is Total Variation (TV), which, in the context of a 2D image x, can be defined as follows [24],

$$R(x) = d_{TV}(x) = \sum_{i,j} \sqrt{|x_{i+1,j} - x_{i,j}|^2 + |x_{i,j+1} - x_{i,j}|^2}, \tag{6}$$

with i, j horizontal and vertical indices of pixels in our image. This is known as the isotropic total variation, and is the original one introduced in 1992 [24]. However, some more recent papers use the anisotropic TV [25],

$$R(x) = d_{TV}(x) = \sum_{i,j} (|x_{i+1,j} - x_{i,j}| + |x_{i,j+1} - x_{i,j}|), \tag{7}$$

which is, in some cases, easier to optimise. This combats the corrupted pixel example mentioned earlier, by penalising high variance in nearby pixels and sharp colour gradients, which accurately describes noise for many of our problems [6]. The method is clearly gradient based, and can be interpreted as taking the gradient operator of our image x. So, we will have higher values when there are steeper gradients in pixel values. This means our optimisation problem gives us an output that is smooth in some sense, a desirable quality to counteract noise. This method is effective, however it does have issues, namely in reproducing more complicated or piecewise measurements. An explicit example would be an inpainting problem, where portions of the corrupted image have lost large continuous sections of content, where the actual measurement would be penalised under this TV approach. Thus, we would like to move away from these explicit methods, in favour of more flexible and general approaches.

A more modern approach to this optimisation is to move to a supervised machine learning setting. This means we have data to train on, so can train a CNN on noisy/noiseless image pairs, and aim to impose our prior through this training dataset. The task is to train our neural network f with parameters θ such that for noisy/noiseless images y_i, x_i^*, we find parameters θ^* that minimise some measure of how well our model is performing, that we denote as J. We call this the loss of our network and is defined much like our error term E, for example, again, the ℓ_2 loss. This turns our problem into the following:

$$\min_\theta J(\theta) = \min_\theta (A(f_\theta(y_i)) - x_i^*), \tag{8}$$

over all i in our training set, where we now want to find the parameters θ, rather than the image directly. The hope is that the network learns how to map a noisy image to its original by distinguishing the 'important' parts of an image from the noise, and so can generalise to unseen images. There are many variations of this approach [7, 8], where there are different architectures and decompositions of the images before training. This is incredibly effective, with most state of the art models following this methodology. However, it comes with the assumption we have well-labeled data that can be used to train our model. This is a strong assumption; in many, if not most, applications we do not have enough data to train a network sufficiently. So, it is desirable to find an unsupervised/self-supervised method, where we do not need any labelled data, even if there is a slight trade-off in performance.

The paper 'Deep Image Prior' [1] proposed a method that provided surprisingly competitive results in an unsupervised setting. Rather than imposing our prior knowledge in the explicit regularisation form $R(x)$, as we have seen in previous approaches that do not use a training dataset, we instead regularise our inverse problem using the structure of the DNN itself and perform an iterative approach based upon this. That is, we only observe the noisy image and instead claim that a DNN can impose a strong enough prior on our problem by preferring certain aspects in the optimisation process, due to its intrinsic structure.

1.2 Report Overview

We delve into the details of the method in section 2, describing the necessary background to understand what makes DIP work. Although it is a big breakthrough in the world of image processing, it does have its flaws, particularly in that because it is still an iterative approach, it overfits if left to optimise with standard methods for too long. However, if we can find a way to mitigate this, we achieve very good results. This report will, after introducing the necessary background, explore a selection of approaches that combat this overfitting problem in a range of different ways, which allow the use of a DIP to be feasible for real world use in inverse imaging problems. Section 3 covers a simple approach created as part of this project, whilst sections 4 and 5 explore more well established methods and prove their main results. Finally, we compare and conclude in section 6, mentioning where each method would be used and how they differ.

2 Deep Image Prior

First, we cover some prerequisites about neural networks, probability, statistics, and optimisation. We begin our discussion by briefly describing the components that create a neural network in the supervised setting. In general, a neural network f_θ is a composite function that takes an input, e.g an image or vector of measured values, which is then passed through a series of layers represented by a corresponding function and parameters/weights. Each layer/function feeds into the next, so the output of one is the input of the next, which continues through our network to create a final output. As we are in a supervised setting, that is, we know what our input should provide as an output, we can compare the two and then optimise our network's parameters, denoted θ. This is known as the training process, after which we hope to use our trained model on unseen examples [9]. This description is deliberately vague, as there are a great many different network architectures and layers. We will however just focus on one, the Convolutional Neural Network (CNN).

2.1 Convolutional Neural Networks

A CNN is most commonly used for processing images, that is in our case, inputs of the form $x \in \mathbb{R}^{H\times W\times D}$, with H, W, D representing Height, Width and Depth of the image respectively. If the image is in greyscale, $D = 1$, whereas for colour, there are 3 channels: Red, Green and Blue (RGB), so $D = 3$. An image is then passed into the first layer, called the input layer, which simply takes in the input, with no other operations. Then, there are a number of different layers that could follow, we shall cover a few relevant ones relevant to a CNN [20]. The overall structure consists of convolutional layers (which extract features from our image), followed by activation layers (which introduce non-linearity to our output, allowing for more complicated patterns to be learned), then pooling layers (which aim to more effectively transfer this data). The first layer we describe in detail is a convolutional layer, which, as the name suggests, is the main component of a CNN.

A convolutional layer consists of a convolution kernel $w \in \mathbb{R}^{H'\times W'\times D'}$, where the dimensions are user specified but usually a low odd number with height and width equal to each other, for example, for a greyscale image $x \in \mathbb{R}^{H\times W\times 1}$ take $w_0 \in \mathbb{R}^{3\times 3\times 1}$. Here, depth represents how many kernel filters we wish to apply, a concept we will return to. This kernel is trainable, with the objective of highlighting patterns in our image, for example, detecting edges or facial features. This layer is why CNNs are so good at pattern detection, and hence why they are used in DIP. The kernel acts as a filter passing over our image, operating on the area under it. To produce an output, we pass over the image, with each element of our example 3×3 kernel w_0 having weights as such:

$$\begin{bmatrix} w_{-1,-1} & w_{-1,0} & w_{-1,1} \\ w_{0,-1} & w_{0,0} & w_{0,1} \\ w_{1,-1} & w_{1,0} & w_{1,1} \end{bmatrix}. \tag{9}$$

It is clear how this can generalise to higher dimensions, and to more filters, by increasing the range and adding another index d for the depth of our kernel, which we did not include in this case as it is trivial if only using one. However, in a $3 \times 3 \times 3$ case, this would look like three of the above, stacked on top of each other. This is then applied to our example greyscale image x in the following way:

$$x_{i,j}^{\text{out}} = \sum_{h,w=-1}^{1} x_{i+h,j+w} w_{h,w}. \tag{10}$$

In the colour image case, as stated before we add in an extra index d, to give the following for a $3 \times 3 \times 3$ kernel:

$$x_{i,j,k}^{\text{out}} = \sum_{h,w,d=-1}^{1} x_{i+h,j+w,k+d} w_{h,w,d}. \tag{11}$$

We shall generally stick to the greyscale case for simplicity in examples, but all results are very easily generalised. We can increase i, j by a user specified amount each time, this is called the stride. However, we will just assume the stride is 1 for this example, that is, we compute $x_{i,j}\ \forall i = 1, ..., H, j = 1, ..., W$. Notice how we would have to have our indices $i = 2, ..., (H-1), j = 2, ..., (W-1)$, so that we do not have indices that are outside the bounds of our image due to the negative values that h, w can take in the summation. However, this shrinks the dimension of the output to $x^{\text{out}} \in \mathbb{R}^{(H-2)\times(W-2)\times 1}$, which may not be desirable. A simple solution is to define a single pixel buffer around our image, that takes a user defined value, often all zeroes or the average values of the actual image pixels within its $H^{'} \times W^{'} \times D^{'} = 3 \times 3 \times 1$ area. This ensures there is no dimension loss and is called padding.

We can then add an additional bias term $b \in \mathbb{R}$ to each term, to give our output as $x_{i,j}^{\text{out}} + b$. We would do this to bring our values into a certain range and give another layer of flexibility to our network. Note, there is only one bias term for each filter w_i we have, and it is a trainable parameter. This gives rise to something else that is often done, using multiple filters to increase the depth of our array. So, if we were to use 16 3×3 filters on a $4 \times 4 \times 1$ greyscale image with padding, we would have an output array of $4 \times 4 \times 16$. So, we can transfer some level of information into the depth of our image by adding more channels. This is useful, as we shall soon see, because it means we can reduce the dimensions H and W, but still retain information.

The next layer usually follows a convolutional layer, adding some non-linearity. We call this an activation layer, sometimes this is combined within the definition of other layers where it would just be called an activation function. There are a wide range of these used for different purposes and in our case where we represent it as a layer, this activation function is applied to each input value, with the output being of the same size. Some examples include the sigmoid function,

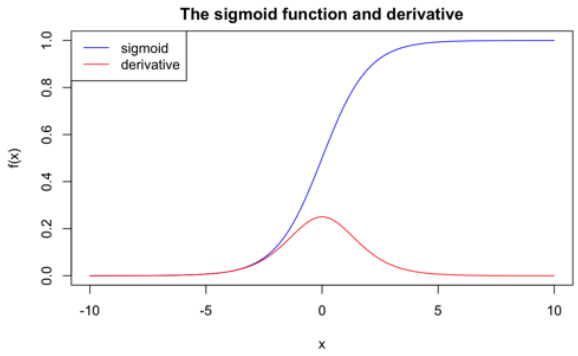

Figure 1: The sigmoid function plotted with its derivative in R, which is very simple and symmetric.

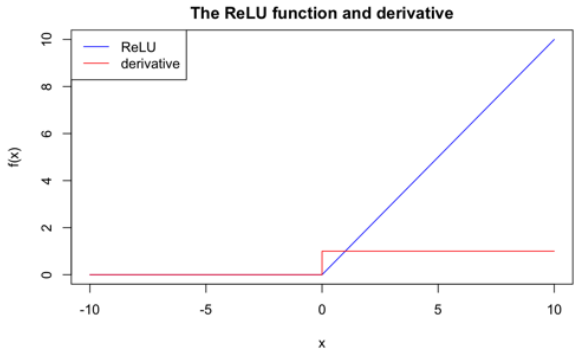

Figure 2: The ReLU function plotted with its derivative in R, notice the constant derivative.

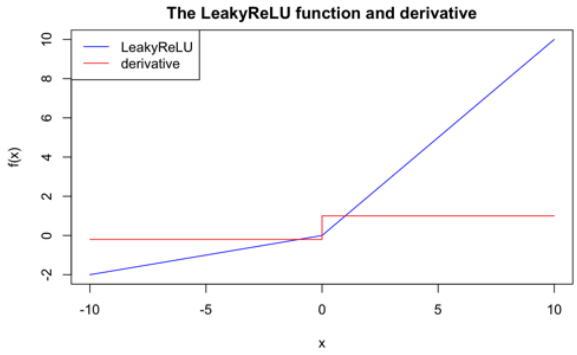

Figure 3: The LeakyReLU function with $\alpha = \frac{1}{5}$ plotted with its derivative in R, simliar to ReLU, but with non-zero derivative below zero.

$$\sigma(x) = \frac{1}{1+e^{-x}}, \ \frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x)), \tag{12}$$

ReLU function,

$$\text{ReLU}(x) = \max\{0, x\}, \ \frac{d}{dx}\text{ReLU}(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0, \end{cases} \tag{13}$$

and the LeakyReLU [22] for an $\alpha > 0$,

$$\text{LeakyReLU}(x) = \begin{cases} \alpha x & x < 0 \\ x & x \geq 0, \end{cases} \ \frac{d}{dx}\text{LeakyReLU}(x) = \begin{cases} -\alpha & x < 0 \\ 1 & x \geq 0. \end{cases} \tag{14}$$

Sigmoid was the traditional activation function used, however, in recent times, it has given way to ReLU and LeakyReLU, as their derivatives are significantly easier to compute due to being constant, which, as we shall see, is very important in the optimisation process. So, in our CNN, we would apply one of these functions to each value in our array, then pass it on to the next layer. In DIP, LeakyReLU is used.

Generally, after an activation layer, we may wish to reduce the size of our image. This is primarily to reduce computation but by creating a lower dimensional representation, this also removes irrelevant information to combat overfitting [21]. We do this in a pooling layer, which, like a convolutional layer, has a kernel that performs some sort of operation by passing over our array. This too has a stride with height and width generally equal, but the key difference is that

this kernel is not learnable, that is, it is a fixed layer that is applied in our network. There are many popular operations that a pooling kernel could perform, the simplest two being average and max pooling.

In average pooling, for a pooling kernel of size $H \times W$, such as 2×2, we simply compute the average of our array's elements that are in the 2×2 square. Similarly, in max pooling, the maximum value is taken. We illustrate this with an example. Consider the below $4 \times 4 \times 1$ greyscale image with simple pixel values,

$$\begin{bmatrix} 0.2 & 0.8 & 0.6 & 0.6 \\ 0.7 & 0.3 & 0.1 & 0.8 \\ 0.5 & 0.4 & 0.3 & 0.9 \\ 0.5 & 0.1 & 0.2 & 0.7 \end{bmatrix} \tag{15}$$

and a pooling layer with a 2×2 kernel and stride 2. For our two examples, we first apply pooling to the top left. Thus, our first calculation for average pooling is,

$$x_{1,1}^{\text{avg}} = \frac{0.2 + 0.8 + 0.7 + 0.3}{4} = 0.25, \tag{16}$$

and for max pooling,

$$x_{1,1}^{\text{max}} = \max\{0.2 + 0.8 + 0.7 + 0.3\} = 0.8. \tag{17}$$

We repeat this not by moving over by 1, but by 2, as our stride is 2, to give a final 2×2 output. If the stride was 1, this would be a 3×3 output. Our examples in the stride $= 2$ case look like the following,

$$x^{\text{avg}} = \begin{bmatrix} 0.25 & 0.525 \\ 0.375 & 0.525 \end{bmatrix} \tag{18}$$

and,

$$x^{\text{max}} = \begin{bmatrix} 0.8 & 0.8 \\ 0.5 & 0.9 \end{bmatrix}. \tag{19}$$

These three layers make up the bulk of general use CNNs, their general use being classification tasks. What follows for these networks is to convert the array into a vector, by flattening it, then producing probabilities by normalising the vector $z = (z_1, ..., z_N)$ using the softmax function,

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{N} e^{z_i}}, \tag{20}$$

to achieve this. However, we do not want prediction probabilities, we want an image of the same size as our original. This is because in our inverse problem setting, we wish to output an estimate of our original image before transformations, with our input that same image, of the same dimensions, with transformations applied. Therefore, we aim to use an architecture that can achieve this consistency of dimensionality. One that has showed very good results in the DIP setting is known as a U-net [23].

This U-net is so called because it, essentially, goes down, in terms of H, W dimensionality, then back up, to provide an image of the same dimension we had originally. It begins with the process described above, that is convolution $\rightarrow$ activation $\rightarrow$ pooling, then repeats until we have a lower dimensional representation of our image. However, as we do this, we use increasing amounts of filters in our convolutions, so the depth of our array increases. For example, in the original U-net [23], the input image is of dimension $572 \times 572 \times 1$, and the lowest level representation has dimension $32 \times 32 \times 1024$, that is, we have 1024 32×32 representations of our original image, each hopefully capturing some notion of pattern of our original image e.g. vertical edges. Note, these low level representations, although essential to the network's performance, are almost completely incomprehensible to humans. Although we can use simple examples to explain the intuition, these representations are far more abstract than, for example, detecting vertical edges. Despite this, the

network can learn from these and they are, in fact, the main reason for the model's effectiveness, along with why they are used for DIP . Then, we go back up with upsampling methods, that is, we increase the size of our input array, until we are back to the size of the original image. A simple example of this would be to repeat each single value to form a 2×2 array and create a new image by putting these back together, that is,

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix}. \tag{21}$$

This is not what is done in the U-net architecture, we do not cover the details, but it suffices to think of this process, named up-convolution, as a clever way of inverting the original (down) convolution. It is not an exact inversion, as information has been lost by reducing dimension, but as the process has convolutional filters, which are learnable, we can hope to recover certain attributes we want, whilst removing ones we do not. In our problem, this amounts to keeping the original image to recover, and ignoring the noise. This process continues until we reach the end of the network, where we produce an output image with the same dimensions as our original.

We now know how an input of an image can be transformed through our network to produce another of the same size, but this is currently useless; we have no idea what the weights $w \in \theta$ should be to produce our output, so there is no meaning to our output. This is where we can now train our network to find the optimal parameters. We do this using a process called gradient descent.

2.2 Gradient Descent

In gradient descent, in the context of a neural network, we have network $f_\theta(\cdot)$ with parameters θ and a training set of $(y_i, x_i)\, i = 1, ..., n$ true value/input sample pairs. Our end objective is to find some θ such that $f_\theta(x) \approx y$ for unseen y, x outside our training set. To do this, we want to minimise a loss function $J(\theta)$, as before, with respect to θ. So we aim to find θ that minimises J. A common analogy for this process describes a hiker at the top of a mountain, and asks the question, "which way should I go to get to the bottom quickest?". This motivates the use of the grad operator, to find the direction of steepest gradient for our loss. In standard gradient descent, note that this J will be a function of all our training samples, such as the sum or average of the difference of our model's output and the real value. We can perform this optimisation by initialising random weights θ_0, and updating these with the rule,

$$\theta_{t+1} = \theta_t - \Delta\theta_t = \theta_t - \alpha \nabla J(\theta_t), \tag{22}$$

where α is a user defined parameter called the learning rate [9]. In standard, or batch, gradient descent this is constant, and dictates how large our optimisation steps are. It is standard to call $\Delta\theta_t$ the update term. This method is widely used for its effectiveness in most problems, but it does have a drawback. Since we have to calculate the gradient of our samples over the entire training set, this can be very computationally expensive, even if we use ReLU or LeakyReLU. A more practical method relies on the fact that generally a small subset of our training dataset will be representative of the whole. An extreme case would be Stochastic Gradient Descent (SGD), where we randomise the order of our (y_i, x_i) pairs, and update one term at a time [9],

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t; (y_t, x_t)). \tag{23}$$

Here, the inclusion of (y_t, x_t) indicates that the output of J is dependent on this one pair of values. This will significantly decrease computation time, and will also allow for some stochasticity to escape some local minima in place of a better one. One problem of non-convergence, where because

we are picking random terms our optimisation trajectory bounces around our desired minima, is solved easily. We produce a decreasing sequence of $\{\alpha_t\}_{t\geq 0}$ such that the updates become smaller until convergence.

Another approach, mini-batch gradient descent attempts to incorporate the confidence in convergence of the batch approach and the randomness and speed of the stochastic approach. It does this by taking random subsets of our data; in each iteration we pick a random subset of n out of N data points, denoted by a subscript t, to give,

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t; (y_{t,1:N}, x_{t,1:N})), \tag{24}$$

where $1:N$ signifies we use all values $i = 1, ..., N$. Now, we move onto describing the method of DIP itself. As we shall see, because we have no training set of pairs, just one sample, we use batch gradient descent, but we will return to the other methods mentioned in this section later.

2.3 Problem Statement

In short, the process of training a neural network amounts to calculating gradients of our loss function. For DIP, recall (5). In our attempt to solve this problem so far, we have focused on finding an explicit representation of $R(x)$, our prior, or training a supervised CNN to map unseen noisy images back to their original. However, DIP attempts to mitigate the need to find this expression R or train an expensive network, by arguing it is possible to encode prior knowledge *implicitly* in the learning process. Despite using similar styles of machinery, these methods use DNNs in completely different ways. In the supervised setting, $f_\theta(\cdot)$ acts as a map that takes deformed images to cleaned images, and thus, after training, can be thought of as a function of y, our noisy image. In the DIP setting, there is no such training process, we instead leverage the fact that the structure of a DNN, specifically CNNs, represent images very well; it is the algorithm we use rather than the DNN that maps our input to the cleaned image. As such, we begin by initialising a CNN with a random input z and random weights θ, to give us our object $f_\theta(z)$, that provides a structure for our image. Note, that the output of this is currently meaningless as it is completely random, as seen in figure 6. We do not care about the input z as, again, we want the network purely for structure, so we can think of this network as a function of θ. As such, in order to slot this into our problem, we will need to shift the variables we optimise over. Note, that with a surjective $g : \theta \mapsto x$, (5) is equivalent to,

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, E(g(\theta); y) + R(g(\theta)), \tag{25}$$

for our desired parameters θ^* because θ and x are related via g. Now, the crucial step here is to consider how we can apply prior regularisation with this formulation. We could design our R term as before to regularise in this new setting, but we would run into the same issues previously mentioned. The observation made in the original paper is that DNNs can represent certain aspects of images remarkably well, e.g. convolutional layers encoding patterns in our image. This motivates the line of thought that we can use the *structure* of a DNN to regularise our inverse problem, imposing a prior that captures the statistics we desire. Thus, we can start to consider how different architectures of DNNs affect our optimisation problem with the type of prior knowledge they encode. For example, in a CNN, patterns in our image are easily detected, whilst random terms, such as our noise ϵ, cannot be represented as easily. We can now alter our problem to leverage this, and do away with the *explicit* prior R, in place of our *implicit* prior imposed by using a CNN. As such, we let our prior term be identically 0, and as stated before, because we can think of $f_\theta(z)$ as a function of θ, we can substitute in $g(\theta) = f_\theta(z)$ as our approximation to the original image, assuming surjectivity, and have our final formulation,

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, E(f_\theta(z); y), \tag{26}$$

in the hopes finding θ^* such that $f_{\theta^*}(z) \approx x^*$, an expression which we can optimise over using standard gradient descent, starting with our random parameters θ_0 and image output of our neural network $f_{\theta_0}(z)$. This is an iterative process, so eventually, just like (3), this will overfit and our CNN will find the global minimum when it copies our noisy image. However, the key observation and difference is that there is a different optimisation trajectory, one that passes close by a local minima that is close to our original image. See this illustrated for the case of $A = I$ in fig 4. A similar representation can be created for the general problem, but as A can change the dimensionality of our output, it is harder to represent, so we shall stick with this. It is also important to mention that this is a general method framework, rather than the only way to perform this optimisation. The term E can be chosen for the problem at hand, and variations of the DIP method have been visited, for example, one adds an additional TV regularisation term to the above problem to further regularise [6].

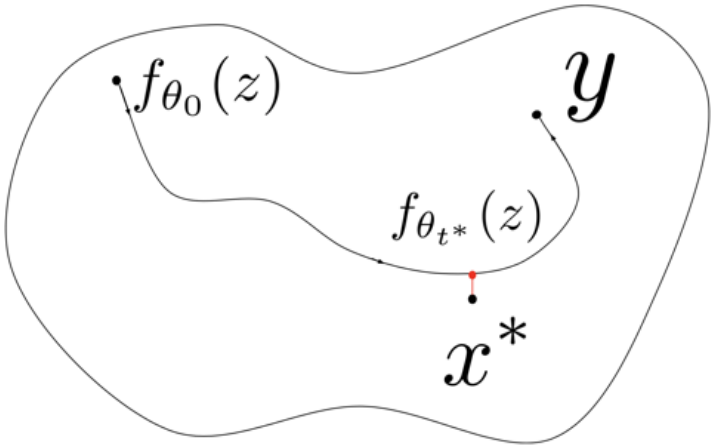

Figure 4: We can see that our optimisation trajectory from initial random image $f_{\theta_0}(z)$ to noisy image y passes near x^*, with the ideal stopping point being at time t^*, that is, our recovered image is approximated by $f_{\theta_{t^*}}(z)$. Here, we are assuming $A = I$, as in other more general cases, our data points may not lie in the same space. Similar ideas apply, but it is far harder to represent in an image.

Figure 5: The results of an experiment using DIP. From left to right we have the estimate x_0, the original image x^* and the noisy measurement y. Here we added Gaussian noise to our original x^* with $H = 400$, $W = 400$, $D = 3$, $\sigma = \frac{20}{255}$ and $\alpha = 0.05$. Code can be found with [1].

Figure 6: In this graphic we can see an example of the iterative process. We begin with our random initialisation, before starting to generate recognisable components of the original image. The larger middle image is what we deemed a good approximation of the original image, and so would be a good time to enforce early stopping. After this, we begin to overfit, culminating in the bottom right image, which is in fact just our noisy measurement. In this example we displayed every 100th image, and selected a representative subset to show here, adding in the final noisy image for clarity.

In fig 6 we can see a typical optimisation process for a small image. Notice the random initialisation, and how after a certain point, the noise begins to creep back in. This is fantastically subtle, and the fact it works is remarkable; the prior regularisation enforced by using a CNN's structure itself is enough to guide our optimisation trajectory past the minima, our desired image. Intuitively, it is more difficult for our CNN to learn the ϵ noise term, or any deformation A, as there is no discernible pattern for our convolutional layers to capture, so it learns to approximate the original image's low level statistics before the noise. This means we can stop the iterative process before it starts overfitting and learning the noise, leaving us a good approximation of our original image. However, this relies on a strong assumption; that we know when to stop.

2.4 Early Stopping

We come back to the early stopping approach mentioned in section 1.1. Due to the over-parameterisation of CNNs, our network has the capacity to learn an immense amount of data, and will overfit eventually, learning the noise term [1]. Again, the purpose of regularisation is to reduce the space of potential solutions to a subset with more desirable qualities, given some constraint. Previous methods we have seen do this explicitly in the optimisation criteria or implicitly within our network, however we can also regularise by stopping our optimisation process early. This is known as early stopping, which does indeed reduce our solution space by preventing our output straying too far from the initialisation and learning unnecessarily intricate details i.e. noise.

Now we focus on various methods for this, as it seems to be the key step to implement the DIP method practically, ranging from simple to complex, and from no assumptions to heavy.

3 Naïve Approaches

3.1 Original

In the original DIP paper, the main aim was simply to show a CNN has the capacity to function as an image prior. Thus, the authors simply examined the output of each iteration, and then picked the 'best of the bunch' for their paper. This is not a desirable solution as this is very much prone to human error and perpetuates the replication crisis that is an increasingly prevalent issue in the machine learning field [11].

A similar approach mentioned in the paper is to simply pick n such that we terminate our process at iteration n, giving us our result as the output of $f_{\theta_n}(z)$. This has some level of success if we know the size of the image and can estimate some notion of complexity, that is, how long the process will take. This is somewhat circular, in practice requiring prior testing, something we are trying to avoid. You could attempt to create some general heuristic based upon a measure of variation/complexity of the image, but this is very prone to counter-examples and will not contain the same level of accuracy as other approaches.

A common theme here is to attempt to leverage some specific information about the image, e.g. total variation, to inform our stopping criteria. If we could make an assumption about the data that provides us more information, we could make a stronger inference.

3.2 Known Variance Case Study

Consider the problem of de-noising, that is where $A = I$ in (1), so we only have to consider the noise. This noise can follow any distribution, unknown to us. One of the most common of these is where $\epsilon_i \sim_{\text{iid}} \mathcal{N}(0, \sigma^2)$, for $i \in 1, ..., m = n$ and $\sigma^2 \in \mathbb{R}^+$, or equivalently, $\epsilon \sim \mathcal{N}_n(0, \sigma^2 \mathbb{I})$. That is, for each pixel value in our image, we add to it a noise term of zero mean and finite variance. Clearly, the larger the variance, the noisier our image is, and the harder it is to recover. What we will assume for this approach is that our noise is uniform in the above sense, with our variance known and $A = I$.

Provided the above, we can approach the problem as such. Recalling terms from section 2), at any given iteration t, we have our noisy image $y = Ax^* + \epsilon = x^* + \epsilon$, and our current output $f_{\theta_t}(z)$, given the newly updated parameters θ_t. Note, $\lim_{t\to\infty} f_{\theta_t}(z) = y$, but $\exists t^* < \infty$ such that $f_{\theta_t}(z) \approx x^*$, our original image. The idea is that at this t^*, we can calculate

$$(y - f_{\theta_{t^*}}(z)) \approx x^* + \epsilon - x^* = \epsilon. \tag{27}$$

This seems very promising, however, as ϵ is normally distributed, not constant, we cannot simply detect this manually. Instead, we can take the squared ℓ_2 norm,

$$\|y - f_{\theta_{t^*}}(z)\|_2^2 \approx \|\epsilon\|_2^2, \tag{28}$$

and find a criteria with this. Now we can compute the expectation here to find an approximate value,

$$\mathbb{E}[\|\epsilon\|_2^2] = \mathbb{E}[\sum_{i=1}^{n} \epsilon_i^2] = \sum_{i=1}^{n} \mathbb{E}[\epsilon_i^2], \tag{29}$$

with n the number of pixels in our image. Then, note we have that, for a random variable X with finite mean and variance,

$$\text{var}(X) = \mathbb{E}(X^2) - [\mathbb{E}(X)]^2. \tag{30}$$

This leads to,

$$\mathbb{E}[\|\epsilon\|_2^2] = \sum_{i=1}^{n} [\text{var}(\epsilon_i) + \mathbb{E}[\epsilon_i]^2] = \sum_{i=1}^{n} [\sigma^2 + 0] = n\sigma^2, \tag{31}$$

as $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. Consider how the values in (28) change as our optimisation runs, illustrated in figure 7. In the initial step, $\|y - f_{\theta_0}(z)\|_2^2$ will be large as there is not necessarily any similarity; z and θ_0 are random. We know this will then decrease as $f_{\theta_t}(z) \to y$. Then we have the stage where $f_{\theta_{t^*}}(z) \approx x^*$, meaning $\|y - f_{\theta_{t^*}}(z)\|_2^2 \approx n\sigma^2$. Then, we begin to overfit, clearly,

$$f_{\theta_t}(z) \to y \implies \|y - f_{\theta_t}(z)\|_2^2 \to 0. \tag{32}$$

This gives rise to a natural algorithm:

Algorithm 1 Case study for known variance

```
t = 0
initialise z
initialise θ_0
while True do
    γ ← ‖y − f_{θ_t}(z)‖_2^2
    if γ < nσ^2 then
        t* ← t
        x_0 ← f_{θ_{t*}}(z)
        return x_0
    else
        θ_{t+1} = θ_t − Δθ_t
        t ← t + 1
    end if
end while
```

In our case, this results in subtracting the pixel values of our network output and noisy image, resulting in a increasingly better representation of the noise term, until iteration t^*, approximately at which point the noise representation gets absorbed into the network, as we overfit. We see this illustrated in figure 7.

In many applications these assumptions are rather strong; it is unlikely you will know the exact variance of the noise. However, it is not completely unfeasible, for example, the ISO setting on a camera can provide an estimation of the noise in an image. This is a very simple method for computing our estimate of the reconstructed image, with many flaws. Ideally you would also combine this with a spacial metric, to check the noise is distributed as in the assumptions, rather than all concentrated in one small area, for example. We will now go onto more sophisticated methods, that come with the drawbacks of higher compute required, but deliver on the results.

Figure 7: These are a selection of results of the DIP experiment seen in section 2. Here we have our output image $f_\theta(z)$ in the top row and our difference image $(y - f_{\theta_t}(z))$ in the bottom row, at varying iterations. From left to right we have iterations 100, 4200 and 9000, with difference image norms, that is values of $\|y - f_{\theta_t}(z)\|_2^2$, of 12,535.04, 2940.89 and 2163.18 respectively. Then note that our value of $n\sigma^2 = 3 \cdot 400^2 \cdot (\frac{20}{255})^2 = 2952.71$. That is, the 4200th iteration, which was the first observed sample with difference norm below $n\sigma^2$, should be a good approximation of x_0, which seems to be the case. Note, with regards to the error in using the expectation of ϵ, the actual noise term $\|\epsilon\|_2^2 = 2948.76 \approx 2952.71 = n\sigma^2$.

4 Stochastic Gradient Langevin Dynamics

A follow-up paper to DIP [12] suggested a Bayesian approach to the problem, adapting our gradient descent methods to account for the random noise. The method utilises Stochastic Gradient Langevin Dynamics (SGLD), an idea inspired by the random movement of particles [19]. This adds in extra stochasticity that prevents the network from overfitting. To be able to understand the mechanics around why it works, we have to cover a few essential concepts.

4.1 MLE, MAP and Bayes

In essence, the DIP task is one of estimation - we need to find the parameters θ that best parameterise our image. There are two well-known estimators that we need to cover before delving in.

The first estimator is the Maximum Likelihood Estimator (MLE), which forms the basis for latter estimators. For parameters θ and a random variable X that we want to estimate the distribution of, we define $p(X|\theta)$ as the likelihood. Then, for a vector of observations $X = (x_1, ..., x_n)$ our MLE is

$$\theta_{\text{MLE}} = \underset{\theta}{\text{argmax}}\, L(\theta|X) = \underset{\theta}{\text{argmax}}\, p(X|\theta) = \underset{\theta}{\text{argmax}} \prod_{i=1}^{n} p(x_i|\theta). \tag{33}$$

However, this can be hard to calculate directly, so we can take logarithms in order to write this in terms of the log-likelihood, as follows,

$$\theta_{\text{MLE}} = \underset{\theta}{\text{argmax}} \log(L(\theta|X)) = \underset{\theta}{\text{argmax}}\, \ell(\theta|X) = \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} \log(p(x_i|\theta)), \tag{34}$$

where log is the natural logarithm. This is equivalent because log is monotone increasing, so the optimisation will find the same maxima.

Note how we can interpret (3) as an MLE estimation problem. In our context, as noted with regards to (3), this estimator leads to overfitting in our inverse problem context. Hence, we could consider utilising some prior knowledge to better inform our estimate and regularise, so as to avoid overfitting. To do this we can leverage Bayes rule to find what is called the posterior distribution,

$$p(\theta|X) = \frac{p(X|\theta)p(\theta)}{p(X)}. \tag{35}$$

In the type of optimisation problems we are interested in, as we want to find the parameters and do not care about the exact probabilities past proportionality, we can discard $p(X)$ and rewrite (35) as,

$$p(\theta|X) \propto p(X|\theta)p(\theta). \tag{36}$$

This incorporates our likelihood equation as before, but also includes $p(\theta)$, referred to as a prior in Bayesian statistics. The next section motivates the previous naming in (5), as following the same procedure as before gives,

$$\begin{aligned}
\underset{\theta}{\text{argmax}}\, p(X|\theta)p(\theta) &= \underset{\theta}{\text{argmax}} \log(p(X|\theta)p(\theta)) \\
&= \underset{\theta}{\text{argmax}} \log(p(X|\theta)) + \log(p(\theta)) \\
&= \underset{\theta}{\text{argmax}} \log(\prod_{i=1}^{n} p(x_i|\theta)) + \log(p(\theta)) \\
&= \underset{\theta}{\text{argmax}} \sum_{i=1}^{n} \log(p(x_i|\theta)) + \log(p(\theta)) \qquad (37) \\
&= \theta_{\text{MAP}}. \qquad (38)
\end{aligned}$$

We can see that $\theta_{\text{MAP}} = \theta_{\text{MLE}} + \log(p(\theta))$ under argmax. So, if we can interpret (3) as an MLE estimate, we can interpret our MAP estimate as our full regularised problem, as in (5), with $R(x) = \log(p(x))$, the logarithm of our prior. This motivates the name, and leads us to the question of why we do not simply use this, a regularised version of our simple MLE. That is, we could try to acquire an estimate via $x_0 \approx f_{\theta_{\text{MLE}}}(z)$ or $x_0 \approx f_{\theta_{\text{MAP}}}(z)$. This prior is not as flexible as others we could use, and fails to capture uncertainty, so we could still overfit the data [12]. Although this does not lead to the best results, the concept of including a Bayesian prior is promising. This leads us to use an estimate based upon Bayes rule (35). However, when we are working with the posterior explicitly, rather than with respect to an iterative optimisation method, we would need to calculate our data term $p(X)$. This is because it acts as a normalising constant, so we need it to calculate exact probabilities; proportion is not helpful. This is a larger issue than it seems, as we have [14],

$$p(X) = \int p(Z, X) dZ. \tag{39}$$

In many, if not nearly all, cases this is not possible to be calculated, that is, it is intractable. This leads us to attempt an alternative estimate, using quantities that can be estimated more easily. One example is known as the Minimum Mean Squared Error (MMSE) estimate [13] and suggests an alternative approach as follows,

$$x_0 = \underset{x}{\operatorname{argmin}}\, \mathbb{E}_{(x^*|y)} \|x - x^*\|_2^2 = \mathbb{E}_{(x^*|y)}(x^*|y), \tag{40}$$

where y is our measured noisy image and x^* is our desired recovery image. As before, we can replace x with a randomly initialised CNN $f_\theta(z)$, so, expanding the expectation, we can rewrite this as a problem in terms of θ,

$$x_0 = \int x^* p(x^*|y) dx^* = \int f_\theta(z) p(\theta|y) d\theta. \tag{41}$$

Our posterior here is still intractable, but these roadblocks motivate us to explore methods to approximate our posterior and calculate estimates despite not having the explicit form. The two most common approaches are Markov Chain Monte Carlo methods (MCMC) in section 4.2, and Variational Inference (VI) in section 5.1. We discuss MCMC first as SGLD uses its principles.

4.2 Markov Chain Monte Carlo

The objective here is to be able to sample from a distribution we would not otherwise be able to, such as the ones in our intractable integrals. The use of this is, in our case, to inform our update criteria for optimisation. MCMC methods aim to do this by constructing a Markov Chain that converges to a stationary distribution we set to be our posterior $p(\theta|X)$. Once we have our Markov Chain stationary after a burn-in period, we can sample from it such that our samples are approximately independent from our target posterior distribution. Once enough have been collected, we can produce an estimate of our posterior and use this for our problem [14, 15]. We will discuss one popular MCMC algorithm, Metropolis-Hastings (MH) [16].

We start by assuming we have some distribution that is proportional to our target posterior $p(\theta|x)$ for a single parameter θ, so by (36) we do, that is $p(x|\theta)p(\theta) = f(\theta)$, with $p(\theta|x) = \frac{f(\theta)}{c}$ for c a normalising constant. Now, we design a proposal distribution, that is, a distribution that is easy to sample from and that we can iteratively update based upon our samples. As such, we write this as $q(\theta^*|\theta_i)$. As an example, this could be such that $q(\theta^*|\theta_i) \sim \mathcal{N}(\theta_i, \sigma^2)$ for a given σ^2. Next, we initialise θ_0 randomly in the support of both p and q.

We begin our first iteration by sampling $u \sim \mathcal{U}[0,1]$, and a proposed update term $\theta^* \sim q(\theta^*|\theta_0)$. This update term is what we propose to update our θ_{i+1} to, but we must accept it first. This occurs when $u < \mathcal{A}(\theta_i, \theta^*) = \min\{1, \frac{p(\theta^*|x)q(\theta_i|\theta^*)}{p(\theta_i|x)q(\theta^*|\theta_i)}\}$. This is however in terms of our posterior $p(\theta|x)$, which we do not know, but, we know it up to proportionality. So, our update probability becomes,

$$\mathcal{A}(\theta_i, \theta^*) = \min\{1, \frac{\frac{f(\theta^*)}{c} q(\theta_i|\theta^*)}{\frac{f(\theta_i)}{c} q(\theta^*|\theta_i)}\} = \min\{1, \frac{f(\theta^*)q(\theta_i|\theta^*)}{f(\theta_i)q(\theta^*|\theta_i)}\}, \tag{42}$$

where we now know how to calculate all of the terms due to this clever cancelling of the constant, which happened to be our intractible $p(x)$. If the condition holds, we accept with $\theta_{i+1} = \theta^*$, and if not, we keep $\theta_{i+1} = \theta_i$. We repeat this until convergence, where our samples θ_i can then be considered to be independent from p. Once this step, which we call the 'burn-in' phase, has been completed, that is, we now have a stationary Markov Chain that we can sample from as if we were sampling from p, we can perform Monte Carlo Integration to solve our intractable integrals. Once our chain is stationary, we can sample $\{\theta_i\}_{i=1}^n$ from it, and then compute the approximation [17],

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} f(\theta_i) = \int_\chi f(\theta) p(\theta)\, d\theta \tag{43}$$

for χ the support of p and any f we choose.

We can now use this method to solve our problem, but this is a *very* compute heavy exercise, to the point where it is unfeasible in many use cases for DIP. As such, we must find a more efficient way of utilising this idea.

4.3 Stochastic Gradient Langevin Dynamics

We want to find a regularised estimation of our parameters θ_i that is computationally efficient. We know that MCMC methods allow us to compute the posterior we desire, and that stochastic gradient descent can optimise in a more efficient manner. It turns out there is a physical process captured by the Langevin Equation, that naturally links these ideas, and allows us to perform early stopping as we go.

We will not cover the derivation as it is not relevant to the statistical nature of this project, but the Langevin Equation was used to model pollen grains in water with respect to time, and has since been discretised into time steps and used as a method for MCMC, providing a proposal distribution and update terms for our chain [19]. This discretised version of it can be written as an update term for parameters θ and as follows [18]:

$$\Delta\theta_t = \frac{\epsilon}{2}(\sum_{i=1}^{n} \nabla \log p(x_i|\theta_t) + \nabla \log p(\theta_t)) + \eta, \tag{44}$$

where $\eta \sim \mathcal{N}(0, \epsilon)$. This can be interpreted as a proposal distribution for our Metropolis Hastings algorithm, where we let these updates burn-in, and it turns out this will make θ converge to the posterior we wish to sample from. This seems very similar to two things we have covered. Firstly, our term inside the brackets looks very much like our MAP estimator (37). This is good as it is one of our objective functions with regularisation, but we must take care not to have our solution converge to it exactly, so that we do not overfit. This is the effect of the random term η_t, so if we can consider some way to perform this optimisation, without the massive computational cost of MCMC, we will have a method. The second observation is that, because log is a monotone increasing function, from (22) we can define our loss J as our MAP estimator, to give us a SGD update term using mini-batches (24) of,

$$\Delta\theta_t = \frac{\epsilon_t}{2}(\frac{N}{n}\sum_{i=1}^{n} \nabla \log p(x_{t,i}|\theta_t) + \nabla \log p(\theta_t)), \tag{45}$$

for $x_{t,i}$ $i = 1, ..., N$, defined as in (24) and a decreasing sequence of step sizes $\{\epsilon_t\}_{t\geq 0}$. This is remarkably similar to (44), but without the random update term, and has the computational advantage of mini-batches. It is only natural then to consider combining the two, to create our update term,

$$\Delta\theta_t = \frac{\epsilon_t}{2}(\frac{N}{n}\sum_{i=1}^{n} \nabla \log p(x_{t,i}|\theta_t) + \nabla \log p(\theta_t)) + \eta_t, \tag{46}$$

with $\eta_t \sim \mathcal{N}(0, \epsilon_t)$, the t subscript representing random and the following assumptions,

$$\sum_t \epsilon_t = \infty, \sum_t \epsilon_t^2 < \infty. \tag{47}$$

In our specific example, the reason why this method was chosen is that this random noise term has the effect of cancelling out the noise in our image, that is, it prevents overfitting. This is because the network cannot converge to the global minima of our loss (26), y, due to the stochasticity in our update term, so it converges to the minima minus the noise, approximately our desired recovery image [12]. This is the practical interpretation, we will see this convergence behavior

explicitly in section 6, but the machinery is more subtle.

Clearly the key here lies in the marriage of MCMC and gradient based methods, so why does this work? Intuitively, we are leveraging the fact that MCMC methods guarantee convergence to the posterior distribution, then taking advantage of this without explicitly sampling, instead using more efficient gradient methods. This is desirable, as sampling traditionally is a significant time sink. We are still technically sampling, but, rather than approximating the posterior to sample from, we converge close to it (which is why the solution collapses to the MAP without noise), but inject noise so that we fully explore the space near our posterior, rather than just a point estimate, which would cause overfitting.

Due to this being derived from a continuous time physical process, turning this into a discrete iterative process comes at the cost of some discretization error. As mentioned before, we can interpret this update term as a proposal distribution for our chain in the Metropolis-Hastings sense, and the accept/reject step accounts suitably for this error. Writing our update term as a wholly random variable,

$$\Delta\theta_t \sim \mathcal{N}(\frac{\epsilon_t}{2}(\frac{N}{n}\sum_{i=1}^{n}\nabla \log p(x_{t,i}|\theta_t) + \nabla \log p(\theta_t)), \epsilon_t), \tag{48}$$

so in terms of section 4.1, our proposal distribution is,

$$q(\theta^*|\theta_i) \sim \mathcal{N}(\theta_i + \frac{\epsilon_t}{2}(\frac{N}{n}\sum_{i=1}^{n}\nabla \log p(x_{t,i}|\theta_t) + \nabla \log p(\theta_t)), \epsilon_t). \tag{49}$$

Note that, as we reduce ϵ_t as the process goes with updates getting smaller alongside, the gradient term will become increasingly less significant, to the point where the noise term/variance dominates. This at a high level means we smoothly transition from gradient methods at the beginning of our optimisation, to a random sampling via MCMC at the end. This smooth transition is aided by the MH acceptance probability tending to 1 as t increases [18], meaning we can effectively ignore this step past a certain point; this acts as our burn-in phase. We will explicitly show this, but to understand this transition intuitively, consider the optimisation process.

At the beginning, we will move towards our MAP solution with large gradient steps, that is, large values of $\frac{\epsilon_t}{2}(\frac{N}{n}\sum_{i=1}^{n}\nabla \log p(x_{t,i}|\theta_t) + \nabla \log p(\theta_t))$, with a standard accept/reject criteria to ensure we are not straying too far. Here, our zero-mean noise term is negligible due to the large step sizes. However, as we get closer to the MAP solution, our update term gets smaller, so the random noise starts to contribute more, giving us a method to somewhat overcome overfitting. When we start getting very close to the MAP solution, our update term becomes negligible, so our MH acceptance probability tends to 1, that is, we begin accepting every update. At this point, we can simply ignore the MH steps and assume that we have had our 'burn-in' phase, and begin to sample from our chain. The noise ensures we explore the area near our MAP estimate fully, getting novel samples each time, and the proximity to it prevents the gradient term becoming too large.

Finally then, we can then apply this to our DIP problem. In this setting there is only one sample pair of our network output and noisy image, so we can drop the summation and substitute in y to give the following,

$$\Delta\theta_t = \frac{\epsilon_t}{2}(\nabla \log p(y|\theta_t) + \nabla \log p(\theta_t)) + \eta_t. \tag{50}$$

In practice we simply swap our standard gradient descent term for this above and we can begin to optimise, but behind the scenes there is an awful lot more going on. The guarantee of convergence comes from the fact this acts as a MCMC method with MH updates.

Another reason for using the MH criteria is that because this is derived from a continuous time physical process, turning this into a discrete iterative process comes at the cost of some discretization error. This is accounted for in our update criteria, that naturally adjusts the trajectory. We will assume for the purpose of this project that Langevin Dynamics (44) do indeed converge to the posterior, so all that is needed to prove our samples θ_t converge to the posterior is to prove our update term (46) converges to the Langevin Dynamics update term (44). Note, we will prove this for the general case of SGLD, not specifically for our DIP modification, following [18] closely

4.3.1 Proof of SGLD Posterior Convergence

We start by defining some auxiliary functions g and h_t as follows: $g(\theta) = \sum_{i=1}^{N} \nabla \log p(x_i|\theta_t + \nabla \log p(\theta_t))$, that is, the standard full batch gradient, calculating over the whole dataset. We also have $h_t(\theta) = \frac{N}{n} \sum_{i=1}^{n} \nabla \log p(x_{t,i}|\theta_t) + \nabla \log p(\theta_i) - g(\theta)$, that is, our mini-batched update term, subtracting $g(\theta)$. As we are picking random batches of our data at each iteration t, h_t is a random variable with $h_t(\theta) \sim (0, V(\theta))$; mean 0 as we expect our mini-batches to be representative of our whole dataset, with our variance dependent on our parameters. It is clear to see that our gradient term plus noise can be written as $\Delta\theta_t = \frac{\epsilon_t}{2}(g(\theta_t) + h_t(\theta_t)) + \eta_t$, as defined before. Note now,

$$\text{var}[\Delta\theta_t] = (\frac{\epsilon_t}{2})^2 \text{var}[g(\theta_t) + h_t(\theta_t)] + \text{var}[\eta_t] = (\frac{\epsilon_t}{2})^2 V(\theta_t) + \epsilon_t. \tag{51}$$

Recalling that as $t \to \infty$, $\epsilon_t \to 0$ for $\epsilon_t << 1$, we can see that the squared ϵ_t in our gradient update term $(\frac{\epsilon_t}{2})^2 V(\theta_t)$ will diminish far faster than just ϵ_t in our injected noise, so our update term becomes dominated by the variance of our η_t term. It also, as mentioned earlier, means we can simply ignore our MH accept/reject steps as the acceptance probability $\to 1$. These combined, imply $\Delta\theta_t$ approximates Langevin Dynamics, so, for large enough t, our $\Delta\theta_t$ is a non-stationary Markov Chain with its equilibrium distribution, due to Langevin Dynamics, as our desired posterior. However, we still need to check if the θ_i will converge to our posterior as we update, because the chain is non-stationary.

For this, we will show that a subsequence $\theta_{t_1}, \theta_{t_2}, ...$ will converge to our posterior, so the whole sequence's limit must also be the posterior. This will be due to the injected noise of our update term being dominant, so that we have our update term converging to Langevin Dynamics and so also to the posterior. Notice that a key step will be extracting a term that has a constant ϵ, rather than the decreasing step sizes ϵ_t. We make use of subsequences for this argument, and consider the inter-subsequence sum of our chosen subsequence, that is, for subsequence $t_1 < t_2 < ...$ and full noise term sequence $\epsilon_1, \epsilon_2, ...$, between each term, we can consider the sum in our full sequence $\sum_{t=t_s+1}^{t_{s+1}} \epsilon_t$. By constructing a subsequence so that the sequence of these sums converges to a set value, we can compare the noise, to show what source of noise is dominant, as well as provide a constant step size for our final Langevin Dynamics.

First we choose a small positive $\epsilon_0 << 1$ which we wish our inter-subsequence sum to converge to. Then, due to (47), as the step sizes are decreasing with the sum tending to ∞, we can find some subsequence $t_1 < t_2 < ...$ such that $\lim_{s\to\infty} \sum_{t=t_s+1}^{t_{s+1}} \epsilon_t = \epsilon_0$. That is, we can, for a given small ϵ_0, choose our subsequence such that the sum of full sequence error terms between our s th and $(s+1)$ th subsequence terms, will converge to our small ϵ_0 in the limit. This implies that in the limit, our total injected noise $\| \sum_{t=t_s+1}^{t_{s+1}} \eta_t \|_2$ with $\eta_t \sim N(0, \epsilon_t)$, will be of the order $O(\|\epsilon_0\|_2) = O(\sqrt{\epsilon_0})$, between subsequence steps t_s and t_{s+1}. We will show that this dominates the other sources of noise in our update.

Recall, as we can write $\Delta\theta_t = \frac{\epsilon_t}{2}(g(\theta_t) + h_t(\theta_t)) + \eta_t$, we can interpret $\frac{\epsilon_t}{2}(g(\theta_t) + h_t(\theta_t))$ as the gradient portion of our update term, and calculate the order of this in our given interval t_s to t_{s+1}

and compare this to our noise. Under suitable continuity assumptions, this is given as:

$$\sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t}{2}(g(\theta_t)+h_t(\theta_t)) = \sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t}{2}g(\theta_t) + \sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t}{2}h_t(\theta_t) = \frac{\epsilon_0}{2}g(\theta_{t_s}) + O(\epsilon_0) + \sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t}{2}h_t(\theta_t). \tag{52}$$

We know $\frac{\epsilon_0}{2}g(\theta_{t_s})$ and the order term $O(\epsilon_0)$, so all that is left to determine is $O(\sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t}{2}h_t(\theta_t))$. Due to the small size of $\epsilon_0 << 1$, we can choose a t subject to $t_s < t < t_{s+1}$ such that $\|\theta_t - \theta_{t_s}\|_2 << 1$, so there is very little difference in our parameters. This means that the dominant level of stochasticity in h_t will come from the random selection of our mini-batches. Assuming these batches are chosen iid, we have that $h_t(\theta_t)$ are also iid, then recall (51) with $\text{var}[h_t(\theta_t)] = V(\theta_t)$, so unrelated to $O(\epsilon_t)$, to provide us with,

$$\text{var}[\sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t}{2}h_t(\theta_t)] = \sum_{t=t_s+1}^{t_{s+1}} (\frac{\epsilon_t}{2})^2 V(\theta_t) \in O(\sum_{t=t_s+1}^{t_{s+1}} (\frac{\epsilon_t}{2})^2) \implies \sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t}{2}h_t(\theta_t) \in O(\sqrt{\sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t^2}{4}}), \tag{53}$$

which we can do due to $V(\theta_t)$ being a bounded term, and taking square roots. Considering the square root and squared ϵ_t here, we can simply absorb this into our $O(\epsilon_t)$ term to give,

$$\sum_{t=t_s+1}^{t_{s+1}} \frac{\epsilon_t}{2}(g(\theta_t)+h_t(\theta_t)) = \frac{\epsilon_0}{2}g(\theta_{t_s}) + O(\epsilon_0). \tag{54}$$

Finally, we can consider the full update term, and because $0 < \epsilon_0 < 1$, we have that between steps t_s and t_{s+1} our total update is,

$$\sum_{t=t_s+1}^{t_{s+1}} \Delta\theta_t = \frac{\epsilon_0}{2}g(\theta_{t_s}) + O(\epsilon_0) + O(\sqrt{\epsilon_0}) = \frac{\epsilon_0}{2}g(\theta_{t_s}) + O(\sqrt{\epsilon_0}), \tag{55}$$

as we absorb the $O(\epsilon_0)$ into $O(\sqrt{\epsilon_0})$. This means our total update is approximately our exact full batch gradient at θ_{t_s}, with fixed step size ϵ_0 and noise dominated by our injected term. Hence, our subsequence $\theta_{t_1}, \theta_{t_2}, ...$ will converge to a sequence from Langevin Dynamics, due to the step size now being constant and the use of g, our exact gradient, noticing the similarity between this and (44). Hence, it will converge to our posterior, and, as this subsequence converges to the posterior, so too does our whole sequence, if it indeed has a defined limit, which in most cases it does. This means we can sample θ from our chain after burn in to approximate our solution $\square$.

Here, we have shown one of the two methods for side-stepping our intractable integrals, MCMC, and a method using it to solve the overfitting problem in DIP. Now, we move onto the other method mentioned, Variational Inference, which aims to do away with sampling entirely, taking a completely different approach to our problem. Note, that despite our Markov Chain converging to the posterior, we do not have an explicit form for it; we can only sample. VI aims instead to find some explicit form to estimate our parameters. This idea leads to another approach to prevent early stopping in the original problem statement, using Bayesian learning.

5 Bayesian Neural Networks

We could interpret our traditional neural network as a frequentist model, in that our end objective is to find a set of parameters θ, that are point estimates best representing our data. However, in a setting where we there is some notion of uncertainty, e.g. our random noise terms, it can be useful to capture this within our network. As such, we will cover the Bayesian neural network architecture (BNN), which models the *parameters* as random variables, that we can then perform inference with. This could include generating confidence intervals for our parameters, to

ascertain how sure we are that this is the correct parameter and to allow a user to adjust network hyperparameters with more information. Crucially however, having this weight uncertainty helps to prevent overfitting in a similar way to the random noise term in SGLD (REF), by correcting for bias caused by the network [13]. We could also collate multiple of these parameter predictions with what is known as an ensemble method, which again prevents overfitting by considering multiple optimisation problems and taking the 'best bits' of each, so to speak [26]. In our setting, we would take multiple realisations of our random weight distributions, to obtain a more accurate average result for each weight in our network. As each iteration would have a different initialisation, we would have different trajectories, helping to iron out any random errors that would be present in a deterministic model.

Here we will investigate an interpretation of the DIP method using these BNNs, which utilises the second way to tackle intractable integrals mentioned in section 4.1, Variational Inference.

5.1 Variational Inference

The setting is the same as in section 4.1, where we wish to obtain some representation of our posterior distribution $p(\theta|X)$, which allows us to compute a good approximation of our original image via integration as in (EQ). However, this involves an intractable integral, which we cannot solve analytically. Rather than attempt to sample from this distribution and build up inference on our posterior from these samples as in MCMC, we instead reframe our problem. Instead of seeing this as a problem of inference via sampling, we attempt to convert it into one of optimisation, so we can use our existing gradient based methods. This is not dissimilar to SGLD's method, but here, instead of optimising for our parameters θ directly, we are optimising for our posterior $p(\theta|y)$ where y is our noisy image [13]. Thus, we want to approximate our posterior distribution by optimising to find some distribution that is somehow similar, which we can then use for our problem.

They key component of VI is the metric we use to optimise over. As we are going to be comparing two distributions, that is, our posterior $p(\theta|y)$ and some candidate distribution $q(\theta)$, we cannot use the metrics we have used so far e.g. ℓ_2 norm. It is a harder question to measure distance between distributions than it is to measure between constants, however there are some standard measures. The one used in VI is called the Kullback-Leibler (KL) divergence, defined for two densities $p(x), q(x)$ as [27],

$$D_{\mathrm{KL}}(p\|q) = \sum_{x\in X} p(x)\log\frac{p(x)}{q(x)}, \tag{56}$$

for X the support of our two distributions, and identically 0 when $p = q$. This is fundamentally an information-theoretic concept, leading to an alternative name of the relative entropy. It is used as a measure of how 'different' the two distributions are, but this is slightly misleading. An important, perhaps unexpected, consequence of this definition is that it is not a metric: it is not symmetric nor does it satisfy the triangle inequality. So rather than simply 'how different' our two distributions are, we can interpret it as measuring how ineffective using distribution q is as a replacement for the true distribution p. This intuitively makes sense for our purposes, we want to find a posterior that approximates our actual distribution well.

Now we can start to define our problem. For a target posterior distribution $p(\theta|y)$ we can initialise a candidate distribution $q(\theta)$ and attempt to solve the following,

$$q^*(\theta) = \operatorname*{argmin}_{\theta} D_{\mathrm{KL}}(p(\theta|y)\|q(\theta)). \tag{57}$$

This seems reasonable, however, it turns out this is very hard to optimise for. To see this,

notice that we can write (56) as an expectation, in terms of p to give,

$$\sum_{x \in X} p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_p[\log \frac{p(X)}{q(X)}]. \tag{58}$$

However, we are taking the expectation with respect to this data term $p(x)$, which, as discussed in section 4.1, is hard to determine. Practically, continuing this method by estimating our expectations with respect to p leads to solutions using this optimisation criteria generally either being intractable, or too simplistic for our purposes [28]. This is fine in a simple example, but as we are working with a CNN with large amounts of parameters, this becomes unfeasible. Surprisingly, the asymmetry of the KL divergence helps us here, as we can switch the places of p and q, getting a different expression entirely, giving us a problem of the form,

$$q^*(\theta) = \underset{q(\theta)}{\operatorname{argmin}} D_{\mathrm{KL}}(q(\theta) \| p(\theta|y)). \tag{59}$$

This still measures some notion of similarity between the two, but because the expectation of (58) in this case is now with respect to q, our suggested candidate distribution which we know, it is far easier to calculate. However, we still cannot determine this exactly because it directly uses the intractable posterior $p(\theta|y)$, which is dependent on our data term. This is because we can write our posterior as such,

$$p(\theta|y) = \frac{p(\theta, y)}{p(y)}, \tag{60}$$

then combinine this with (58),

$$D_{\mathrm{KL}}(q(\theta) \| p(\theta|y)) = \sum_{\theta} q(\theta) \log \frac{q(\theta)}{p(\theta|y)} = \mathbb{E}_p[\log \frac{q(\theta)}{p(\theta|y)}], \tag{61}$$

and use logarithm rules so that our expression becomes,

$$D_{\mathrm{KL}}(q(\theta) \| p(\theta|y)) = \mathbb{E}_q[\log q(\theta)] - \mathbb{E}_q[\log p(\theta|y)] = \mathbb{E}_q[\log q(\theta)] - \mathbb{E}_q[\log \frac{p(\theta, y)}{p(y)}], \tag{62}$$

to finally give us

$$q^*(\theta) = \underset{q(\theta)}{\operatorname{argmin}} \mathbb{E}_q[\log q(\theta)] - \mathbb{E}_q[\log p(\theta, y)] + \log p(y). \tag{63}$$

This shows we still would need to compute our data term $p(y)$, so cannot be feasibly done. Note, the expectation has been removed as $p(y)$ is constant with respect to $q(\theta)$. From this, we now need to find a proportional expression to optimise over.

The fact that our term here is constant motivates us to define a value without it, called the Evidence Lower Bound (ELBO) [14],

$$\mathrm{ELBO}(q(\theta)) = \mathbb{E}_q[\log p(\theta, y)] - \mathbb{E}_q[\log q(\theta)]. \tag{64}$$

Notice,

$$D_{\mathrm{KL}}(q(\theta) \| p(\theta|y)) = \log p(y) - \mathrm{ELBO}(q(\theta)), \tag{65}$$

so there is some link, but there is a change of sign. We can overcome this with some simple manipulation. By recognising that the KL divergence is non-negative, that is, $D_{\mathrm{KL}}(q(\theta) \| p(\theta|y)) \geq 0$, we can produce the following bound,

$$0 \leq D_{\mathrm{KL}}(q(\theta) \| p(\theta|y)) = \log p(y) - \mathrm{ELBO}(q(\theta)) \implies \log p(y) \geq \mathrm{ELBO}(q(\theta)), \tag{66}$$

motivating the naming, as our data term $p(y)$ can also be called the evidence of our model, so, our ELBO is the lower bound for our evidence. From this and (65) we can see that minimising the

KL divergence is equivalent to maximising the ELBO. This provides a far more accessible criteria that we can optimise over. There are other types of these bounds, but we will stick with ELBO for this paper.

Now that we have our objective function, we can begin to optimise with it. This can be done with some tweaks to the gradient based methods we have already discussed. After this, we have our optimisation objective for approximating a posterior distribution. We can now apply this to our image processing context, by utilising a different type of neural network that makes use of the posterior.

5.2 Bayesian Neural Networks and the Approach

As mentioned in the introduction, the main difference when using a BNN as opposed to a standard frequentist DNN is that we calculate a distribution for our parameters θ, rather than just a point estimate. By doing this, we avoid overfitting by effectively creating an ensemble of different estimate values that we can compile at the end [31]. We will discuss a method for doing this explicitly in this section, following the method of a recent paper that takes the same general approach as DIP. That is, we are utilising the structure of our network to impose a prior on our problem, then optimising our parameters θ. We shall now derive the criteria to optimise over, and describe the method itself, following the proof in [13] closely, expanding on certain steps and rephrasing in light of the VI concepts covered in section 5.1.

5.2.1 Derivation of Method

First, recall the setting we find ourselves in. We wish to find an estimate x_0 for our desired image x^* without a training dataset, using our measurement $y = A(x^*) + \epsilon$, where we use a neural network with random input z and weights θ, such that $f_\theta(z) \approx x^*$. Here we will make some extra assumptions: we will assume A is known and can be written as a matrix, model our noise term ϵ as Gaussian white noise, and approximate our prior $p(\theta)$ by a product of i.i.d normal $\mathcal{N}(0, \bar{\sigma}^2)$ random variables. These mean that we have, ignoring constants,

$$\epsilon_i \sim \mathcal{N}(0, \tilde{\sigma}^2) \implies p(\epsilon) \sim \prod_i \exp\{\frac{-\epsilon_i^2}{2\tilde{\sigma}^2}\} \text{ and, } p(\theta) \sim \prod_i \exp\{\frac{-\theta_i^2}{2\bar{\sigma}^2}\} \tag{67}$$

We showed in (40, 41) that such an estimate x_0 is the MMSE, that can be written as,

$$x_0 = \int f_\theta(z) p(\theta|y) d\theta, \tag{68}$$

which, as we discussed, cannot be calculated due to the complexity of our neural network meaning the posterior is intractable. Thus, we must approximate this as well, either with MCMC methods, or, as we shall do now, with VI. First, we initialise a candidate distribution $q(\theta)$, which in the paper is such that $\theta_i \sim \mathcal{N}(\mu_i, \sigma_i)$, so, due to this dependence on mean and variance, we denote $q(\theta) = q(\theta|\mu, \sigma)$. Now, rather than trying to sample straight from the posterior, we aim to minimise the KL divergence between our posterior and candidate. Due to the dependence on μ and σ, we can write our problem like in (59)as the following,

$$q^*(\theta|\mu, \sigma) = \underset{q(\theta|\mu,\sigma)}{\operatorname{argmin}} D_{\mathrm{KL}}(q(\theta|\mu, \sigma) \| p(\theta|y)). \tag{69}$$

Again, this cannot be computed manually as we do not know our posterior. So, we find an alternative measure, using the ELBO bound from earlier,

$$D_{\mathrm{KL}}(q(\theta|\mu, \sigma) \| p(\theta|y)) = \log p(y) - \mathrm{ELBO}(q(\theta|\mu, \sigma)) = c - \mathbb{E}_q[\log p(\theta, y)] + \mathbb{E}_q[\log q(\theta|\mu, \sigma)], \tag{70}$$

where c is constant and hence unimportant for optimisation. Writing the marginal distribution in (60) in terms of $y|\theta$, we obtain the similar $p(y|\theta) = \frac{p(\theta,y)}{p(\theta)}$, which we can rearrange, substitute in and expand logs to give,

$$D_{\mathrm{KL}}(q(\theta|\mu,\sigma)\|p(\theta|y)) = c - \mathbb{E}_q[\log p(\theta)] - \mathbb{E}_q[\log p(y|\theta)] + \mathbb{E}_q[\log q(\theta|\mu,\sigma)]. \tag{71}$$

Now we can use the definition of KL divergence to give us the following,

$$D_{\mathrm{KL}}(q(\theta|\mu,\sigma)\|p(\theta|y)) = c+\mathbb{E}_q[\log \frac{q(\theta|\mu,\sigma)}{p(\theta)}]-\mathbb{E}_q[\log p(y|\theta)] = c+D_{\mathrm{KL}}(q(\theta|\mu,\sigma)\|p(\theta))-\mathbb{E}_q[\log p(y|\theta)]. \tag{72}$$

Then, recalling our assumptions and recognising that, as our candidate distribution is normal and independent, we have $q(\theta_i|\mu_i,\sigma_i) \sim \mathcal{N}(\mu_i,\sigma_i^2)$ and $p(\theta_i) \sim \mathcal{N}(0,\sigma_i^2)$. We then use the 1-dimensional result from [32] concerning the KL-divergence of two normally distributed variables, giving us the following,

$$D_{\mathrm{KL}}(q(\theta_i|\mu_i,\sigma_i)\|p(\theta_i)) = \frac{1}{2}(\log\frac{\bar{\sigma}}{\sigma_i} + \frac{\sigma_i^2}{\bar{\sigma}^2} + \frac{(\mu_i-0)^2}{\bar{\sigma}^2} - 1) = -\log\sigma_i + \frac{1}{2\bar{\sigma}^2}(\sigma_i^2+\mu_i^2) + c_1. \tag{73}$$

So, for our full distributions, abusing constant notation slightly,

$$D_{\mathrm{KL}}(q(\theta|\mu,\sigma)\|p(\theta)) = D_{\mathrm{KL}}(q(\theta_i|\mu_i,\sigma_i)\|p(\theta_i)) = \frac{1}{2\bar{\sigma}^2}(\|\sigma\|_2^2+\|\mu\|_2^2) - \sum_i \log\sigma_i + c_1. \tag{74}$$

The next component of our optimisation term is the expectation of $\log p(y|\theta)$. As we know that $\epsilon = y - Ax^*$ for our known matrix A, as well as the distribution of $p(\epsilon)$, we can substitute in for ϵ to give us the likelihood based on θ,

$$\log p(y|\theta) = -\frac{1}{2\tilde{\sigma}^2}\|y - Af_\theta(z)\|_2^2 + c_2. \tag{75}$$

Combining these together, ignoring constant terms and multiplying through by $2\tilde{\sigma}^2$ for neatness we obtain,

$$\underset{\mu,\sigma}{\operatorname{argmin}}\, D_{\mathrm{KL}}(q(\theta|\mu,\sigma)\|p(\theta|y)) = \underset{\mu,\sigma}{\operatorname{argmin}}\, \mathbb{E}_q\|y-Af_\theta(z)\|_2^2+(\frac{\tilde{\sigma}^2}{\bar{\sigma}^2}(\|\sigma\|_2^2+\|\mu\|_2^2)-2\tilde{\sigma}^2\sum_i\log\sigma_i). \tag{76}$$

In summary, we are aiming to find μ,σ that make our candidate distribution approximate our posterior the best, and now we have a method to optimise for this. We can use (76) to train our BNN model, and find the μ^*,σ^* that minimise that loss, using a variant of gradient descent appropriate for this problem. Then, we can return to (68) and substitute p for q to obtain our final approximation,

$$x_0 = \int f_\theta(z)q(\theta|\mu^*,\sigma^*)d\theta, \tag{77}$$

then use Monte Carlo integration to speed things up, giving us,

$$x_0 \approx \frac{1}{n}\sum_{i=1}^{n} f_{\theta_i}(z), \tag{78}$$

for $\{\theta_i\}_{i=1}^n$ realisations of parameters θ from our simple and easy to sample from distribution $q(\theta|\mu^*,\sigma^*)$. From here, we can simply inspect our resulting x_0 and we have our result $\square$.

This concludes the discussion on the methods in this paper. In the final section, we summarise the content covered and provide some intuition about each method.

6 Comparisons and Conclusions

Throughout the second half of this report we have explained three methods to combat the overfitting issue of DIP, in ways that do not require manual early stopping or selection. We started with a novel but naïve approach based on the extra assumption of knowing the variance of our noise, then derived a criterion for early stopping based upon that. Following this, we observed how we can interpret the original regularisation problem using a Bayesian prior, before encountering an issue with intractable Bayesian integrals and posteriors. The previous two sections detailed two methods to combat these: Markov Chain Monte Carlo and Variational Inference, and methods that were derived from these. Stochastic Gradient Langevin Dynamics make use of asymptotic guarantees from MCMC to prevent the optimisation process from overfitting to the noisy image, instead converging to approximately our original image, thus, removing the need for early stopping. We can see this difference in convergence in figure 8. The Bayesian Neural Network approach on the other hand aims to approximate the Minimum Mean Square Error estimate of our image, which we can do by training a BNN, that is an NN with parameters as random variables, by utilising VI techniques. We therefore estimate these random variables via sampling, and this process's extra stochasticity and ensemble methodology prevent overfitting. These are three different methods, which clearly have different qualities, that we shall conclude the report with.

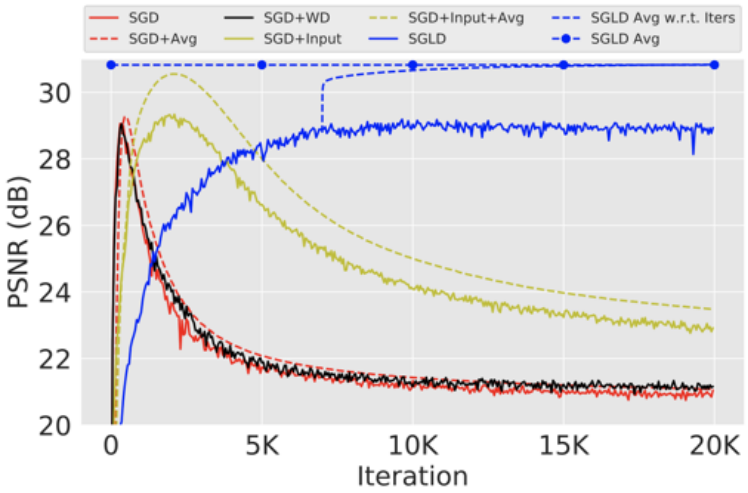

Figure 8: This graph from [12] shows how the Peak Signal to Noise Ratio (PSNR) varies with iterations for a range of different methods of optimisation for DIP. We will not fully define PSNR, it suffices to think of it as a measure of quality in our estimation, so we want to return an iteration with the highest value. Specific numbers do not matter too much here, the main thing to notice is the shape of our trajectories. Whilst the variations of SGD and others show a peak before then overfitting as we learn the noise, SGLD (blue) exhibits a more traditional curve, where it does not overfit, instead, converging. However, note that certain methods have a higher peak, the advantage of SGLD is that it is 'hands-off' with no need to detect this peak. Due to this, by inspecting the iterations, we can see that the process takes far longer.

In the naïve case, firstly note that the assumption of known variance is quite strong in most scenarios, so cannot be used accurately in the majority of cases. Secondly, through testing, larger variance values tend to provide very poor results. With this in mind, the other two methods are far more reasonable for practical use. Both SGLD and use of BNNs are effective and it is not the purpose of this paper to claim which method is better. Instead, what we can do is compare when you may want to use each, stemming from the difference between MCMC and VI.

In the MCMC/MH method, the guarantee of convergence is asymptotic and based upon a large amount of sampling from potentially complicated distributions. This means that although we are guaranteed to be able to sample directly from the posterior distribution we want, it comes at the cost of computational resources. SGLD does combat this with faster gradient methods, but, going back to figure 8, the convergence is still considerably slower due to the burn-in and sampling phase. Thus, MCMC methods, and by extension SGLD, are perhaps more useful when one has time and resources and wants to guarantee an accurate result. In the BNN setting, we utilise VI which does not come with the same guarantees of convergence that MCMC does. As such, the process is somewhat quicker; VI converts inference into optimisation which is faster in general [33]. Perhaps then this BNN stategy is better suited for quick prototyping where time and resources are a concern, and assume that the ensemble nature of BNNs are enough to prevent the overfitting.

In summary, we have described and explored the Deep Image Prior method, inspecting the mathematics behind it, along with the motivation for its creation. We then outlined the overfitting issue that would prevent DIP from being used in practice, before introducing three methods to combat this, with varying levels of sophistication and effectiveness. We have compared these, along with the core concepts that uphold them, to form a cohesive introduction to prior knowledge for use in inverse problems.

References

[1] D. Ulyanov, A. Vedaldi and V. Lempitsky *Deep Image Prior*, 2017, arXiv:1711.10925.

[2] D. Van Veen, A. Jalal, M. Soltanolkotabi, E. Price, S, Vishwanath and A. Dimakis, *Compressed Sensing with Deep Image Prior and Learned Regularization*, 2020, arXiv:1806.06438.

[3] P. Hansen, *Discrete Inverse Problems*, 2010, Society for Industrial and Applied Mathematics.

[4] H. Wang, T. Li, Z. Zhuang, T. Chen, H. Liang and J. Sun, *Early Stopping for Deep Image Prior*, 2022, arXiv:2112.06074.

[5] A. Mahendran and A. Vedaldi, *Understanding deep image representations by inverting them*, 2015, CVPR.

[6] J. Liu, Y. Sun, X. Xu and U. Kamilov, *Image Restoration using Total Variation Regularized Deep Image Prior*, 2018, arXiv:1810.12864.

[7] S. Lefkimmiatis, *Non-local Color Image Denoising with Convolutional Neural Networks*, 2017, arXiv:1611.06757.

[8] H. Burger, C. Schuler, and S. Harmeling, *Image denoising: Can plain Neural Networks compete with BM3D?*, 2012, CVPR.

[9] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach 3rd edition*, 2010, Pearson Education Inc.

[10] K. Jin, M. McCann, E. Froustey and M. Unser, *Deep Convolutional Neural Network for Inverse Problems in Imaging*, 2016, arXiv:1611.03679.

[11] S. Kapoor and A. Narayanan, *Leakage and the Reproducibility Crisis in ML-based Science*, 2022, arXiv:2207.07048.

[12] Z. Cheng, M. Gadelha, S. Maji and D. Sheldon, *A Bayesian Perspective on the Deep Image Prior*, 2019, CVPR.

[13] T. Pang, Y. Quan and H. Ji, *Self-supervised Bayesian Deep Learning for Image Recovery with Applications to Compressive Sensing*, 2020, ECCV.

[14] D. Blei, A. Kucukelbir and J. McAuliffe, *Variational Inference: A Review for Statisticians* 2018, arXiv:1601.00670.

[15] D. MacKay, *Information Theory, Inference, and Learning Algorithms*, 2003, Cambridge University Press.

[16] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller and E. Teller, *Equations of State Calculations by Fast Computing Machines*, 1953, Journal of Chemical Physics.

[17] C. Andrieu, N. De Freitas, A. Doucet and M. Jordan, *An Introduction to MCMC for Machine Learning*, 2003, Kluwer Academic Publishers.

[18] M. Welling and Y. Teh, *Bayesian Learning via Stochastic Gradient Langevin Dynamics*, 2011, Proceedings of the 28th International Conference on Machine Learning.

[19] R. Neal *MCMC using Hamiltonian dynamics*, 2011, CRC Press.

[20] F. Huang and Y. LeCun, *Large-scale Learning with SVM and Convolutional for Generic Object Categorization*, 2006, CVPR.

[21] H. Gholamalinezhad and H. Khosravi, *Pooling Methods in Deep Neural Networks, a Review*, 2020, arXiv:2009.07485.

[22] K. He, X. Zhang, S. Ren and J. Sun, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, 2015, arXiv:1502.01852.

[23] O. Ronneberger, P. Fischer and T. Brox, *U-Net: Convolutional Networks for Biomedical Image Segmentation*, 2015, arXiv:1505.04597.

[24] L. Rudin, S. Osher and E. Fatemi, *Nonlinear total variation based noise removal algorithms*, 2002, Elsevier Science Publisher.

[25] Y. Lou, T. Zeng, S. Osher and J. Xin, *A Weighted Difference of Anisotropic and Isotropic Total Variation Model for Image Processing*, Society for Industrial and Applied Mathematics.

[26] B. Lakshminarayanan, A. Pritzel and C. Blundell, *Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles*, 2017, arXiv:1612.01474.

[27] T. Cover and J. Thomas, *Elements of Information Theory 2nd edition*, 2006, John Wiley Sons, Inc.

[28] K. Murphy *Machine Learning: A Probabilistic Perspective*, 2012, MIT Press.

[29] A. Hoerl and R. Kennard, *Ridge Regression: Biased Estimation for Nonorthogonal Problems*, 1970, Technometrics.

[30] R. Tibshirani, *Regression Shrinkage and Selection via the Lasso*, 1996, Journal of the Royal Statistical Society.

[31] F. Wenzel, K. Roth, B. Veeling, J. Swiątkowski, L. Tran, S. Mandt, J. Snoek, T. Salimans, R. Jenatton and S. Nowozin, *How Good is the Bayes Posterior in Deep Neural Networks Really?*, 2020, arXiv:2002.02405.

[32] Y. Zhang, W. Liu, Z. Chen, J. Wang and K. Li, *On the Properties of Kullback-Leibler Divergence Between Multivariate Gaussian Distributions*, 2022, arXiv:2102.05485.

[33] G. Gunapati1, A. Jain, P. Srijith and S. Desai, *Variational Inference as an alternative to MCMC for parameter estimation and model selection*, 2021, arXiv:1803.06473.