FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Solving Partial Differential Equations with Machine Learning

Lukas Sammler

Bachelorarbeit

Solving Partial Differential Equations with Machine Learning

Lukas Sammler

Bachelorarbeit

Aufgabensteller:	Prof. Dr. H. Köstler
Betreuer:	M. Sc. J. Schmitt
Bearbeitungszeitraum:	8.07.2019 – 9.12.2019

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelorarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 8. Dezember 2019 .....

1 Introduction

One of the common core problems in developing software is the fact that it oftentimes requires not only an understanding of the problem and what steps need to be performed to solve it, but also how one can use the available tools to "tell" the computer how to perform those steps. This is where machine learning enters the picture. Machine learning deals with models and algorithms that allow computers to perform tasks by inferring how to execute them from data they were trained with rather than being told directly. Its history can be described as troublesome at best, since it was also affected by multiple so called "AI winters", when large amounts of funding were withdrawn because the results of the funded projects fell severely short of the promises the researchers had made or the hopes that were associated with the idea of machine learning and artificial intelligence in general. Despite this, development has continued, resulting in machine learning nowadays finding multiple common applications like spam filters, search engine optimisation and especially visual recognition tasks. More regarding that last example can be found in [KSH12] and [GDDM13], but the most publicly noticeable example in recent years has to be the defeat of Lee Sedol, a professional Go player, by AlphaGo, which was developed in [SHM+16].

One of the models used for machine learning are artificial neural networks, which consist of "neurons" that were inspired by biological neurons as found in the human brain. These neurons take an input, combine it with their internal state and emit that as their output. That output can then be used as the input of another set of neurons and so forth, creating very long or "deep" networks in the process, which is why the subclass of artificial neural networks that feature such chains is called deep networks. These networks can then be further classified as a convolutional neural network (CNN) in case at least one of the layers uses the mathematical operation of convolution. One noteworthy CNN is the so called "UNet", described in [RFB15], that was originally designed for biomedical image segmentation and features nothing that could immediately be recognised as a neuron anymore.

But one of the results of the previously mentioned shortages in funding for machine learning is the fact that there have been few to no attempts to solve certain problems with its help. Claiming that partial differential equations (PDEs) were one of the problems that were not attempted to be solved would be a gross miss-characterisation of the research performed in that direction (e.g. [SFG+18]). But something that to our knowledge does not exist yet is an analysis of what a CNN trained to solve PDEs actually learns. Whether it learns how to execute already known methods for solving PDEs, invents brand new ones or whether it just manages to find a set of operations that can solve the problems used during training, which will fail once used on different inputs. However, this extended analysis is beyond the scope of this thesis, which will instead focus on building, training and comparing different networks with TensorFlow (see [Teab]), aiming to lay the groundwork for such an extended analysis. Our reason for using CNNs instead of other types of machine learning is that we will be looking at a relatively large problem that can be described with a stencil code, which would require a significantly larger amount of trainable parameters in other types of machine learning. The amount of trainable parameters in a CNN is completely independent of the size of the original input, whereas fully connected neural networks, for example, require the same amount of parameters as the input size in their first hidden layer alone.

2 Related Work

This thesis will only briefly cover partial differential equations. More extensive descriptions and ways to solve them can be found in [Saa03] and [LeV07], where the former provides a broad overview and the latter focuses on Finite Difference methods.

Since PDEs are commonly used to model physical problems several similar pieces of work can be found that look at specific physical problems instead. An approach to solving steady state heat conduction and incompressible fluid flow problems directly using another class of neural network, called conditional generative adversarial networks, can be found in [FGP17], while [SFG+18] uses the paradigm of weak supervision to integrate domain specific knowledge in order to produce the steady-state solution to the 2D heat equation.

A way in which neural networks can help to solve PDEs without attempting to generate the solution directly can be found in [HZE+19], which describes how an existing iterative solver could

be modified by a deep neural network while preserving correctness and convergence guarantees. This thesis however will attempt to create a network that can provide a direct solution.

[OLT+19] is very similar to this work in that it describes a network intended for solving Poisson's equation directly using a convolutional neural network, but focuses more on different versions of the PDE with different boundary conditions, variable grid spacing and variable mesh sizes, while not considering the exploration of different network architectures and training methods as much, which is the focus of this thesis.

Most works on this subject, this thesis included, restrict themselves to 2D, which makes [STD+17] noteworthy for exploring the possibility of a neural network to act as a solver for Poisson's equation in 3D. It is also especially interested in the real world application of predicting electric potential.

When it comes to the architecture of neural networks the U-net described in [RFB15] has proven to be very inspirational for the creation of the architectures described here, while a completely different approach can be found in [SDBR14], which aims replace certain parts of CNNs like pooling layers with certain configurations of convolutional layers. This is in some ways the opposite of what we aim to achieve here, since that would increase the amount of trainable parameters, while we will aim to decrease it.

3 Partial Differential Equations

Partial differential equations (or PDEs for short) are differential equations that contain unknown functions depending on multiple variables and the partial derivatives of said functions. They can be used to model a wide range of different problems commonly occurring in physics, such as the propagation of sound waves, how heat will spread across an area and what that area's temperature will look like once it has come to rest and the way fluids will spread to name but a few.

Depending on the exact problem they can occasionally be solved analytically. For example, a subclass of PDEs called separable partial differential equations can be written as products of functions depending on just one variable each. This allows a reduction of the problem to solving several ordinary differential equations via a separation of variables. Such methods offer a direct and exact solution to the PDE but are simply not applicable to a wide variety of them.

PDEs being continuous also makes them harder to solve, which is why often times a discretisation of them is solved instead. There are several different types of methods to generate this discretisation, with the three most prolific being the Finite Difference Methods, the Finite Element Method and the Finite volume method, which are further described in [Saa03].

This process results in a problem that can be written as the linear matrix equation 1, where A is a linear operator, u is the unknown solution and f is the known right hand side.

$$Au = f \tag{1}$$

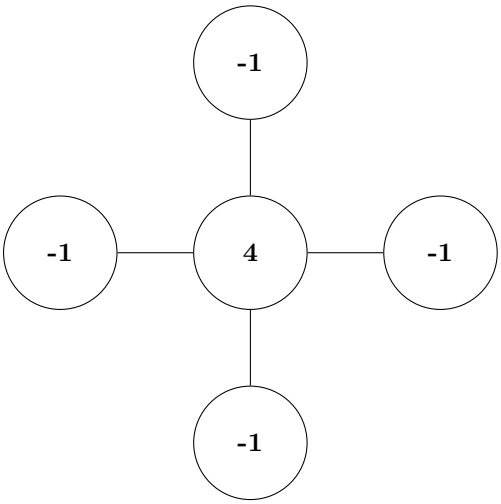
The matrix A that results from these discretisations is often sparse and banded and can therefore also be represented by a so called "stencil", an example of which can be seen in Figure 1. There are two different classes of methods to solve such a problem, which will be briefly described in the following.

3.1 Single Step Methods

The methods that are referred to here as "single step" methods generate an approximate solution in a deterministic amount of steps, without making their run time dependent on intermediate solution proposals. One such approach would be generating an approximate, or even exact, inverse of the matrix A, allowing the computation of Equation 2 not only for one right hand side f, but all combinations of right hand side f and solution u that are linked through A.

$$A^{-1}_{approx} f = u_{approx} \tag{2}$$

Another common example of this is the LU-Decomposition, which splits A into a lower triangular matrix L and an upper triangular matrix U, while using the permutation matrix P, which allows for the computation of $Ly = P^{-1}f = P^T f$, followed by $Uu = y$, using simple forward and backward substitution. The LU-Decomposition itself is the result of gaussian elimination with a complexity



$$4 \cdot u_{i,j} - u_{i+1,j} - u_{i-1,j} - u_{i,j+1} - u_{i,j-1} = f_{i,j}$$

Figure 1: Stencil representing the linear operator A, the equation describes the calculation this stencil implies

of $O(n^3)$. Once that is completed however, both forward and backwards substitution have a complexity of $O(n^2)$, which results in the calculation of further solutions from right hand sides having a complexity of $O(n^2)$, which is the same complexity class the multiplication with the approximate inverse A^{-1}_{approx} resides in.

3.2 Iterative Methods

A more complete explanation of the methods described here can be found in [Saa03], which was used as a reference for large parts of this chapter.

Another class of methods for solving linear matrix equations is composed of so called "iterative" methods that start of with an initial approximate solution u_0 and update it step by step to eventually converge to the exact solution. An example of this is the Jacobi method, which generates a new approximate solution u_{k+1} from the previous approximate solution u_k to fulfil Equation 3, where $(Au_{k+1})_i$ and f_i denote the i-th component of the vectors (Au_{k+1}) and f respectively.

$$(Au_{k+1})_i = f_i \tag{3}$$

When writing the matrix A as in Equation 4, where D indicates the diagonal of A, $-E$ denotes the values below the diagonal and $-F$ the values above it, this allows us to formulate the update rule for u the way seen in Equation 5.

$$A = D - F - E \tag{4}$$

$$u_{k+1} = D^{-1}(E + F)u_k + D^{-1}f \tag{5}$$

Very similar to this is the Gauss-Seidel method, which instead computes updates as described by Equation 6.

$$u_{k+1} = (D - E)^{-1}Fu_k + (D - E)^{-1}f \tag{6}$$

As proven in [Saa03], these methods will converge on the exact result if A is a square matrix that fulfils Equation 7 where $||.||$. is any matrix norm.

$$||A|| < 1 \tag{7}$$

All the methods described up to now are applicable to all problems that can be described by Equation 1 and as such fail to take advantage of the special properties and opportunities that

the discretisations of PDEs provide. The so called multigrid methods on the other hand take full advantage of these opportunities by utilising discretisations of the PDE with different levels of coarseness. Certain parts of the error in typical iterative methods can be reduced quickly. These are referred to as high-frequency errors and are responsible for the initially swift progress of iterative methods. Standard relaxation however has trouble reducing the low-frequency parts of the error. Luckily, about half of these low-frequency errors transform into high-frequency errors on coarser discretisations, whichs allows multigrid methods to quickly remove the high-frequency errors on one level of coarseness before moving on to the next level and removing the high-frequency errors there and so on. A detailed description of how this is executed can again be found in both [BM$^+$00] and [TOS00].

3.3 Our Problem

What we will be working with is Poisson's equation, which can be described by Equation 8, where Δ is the Laplace Operator, u is the unknown solution and f is the known right hand side.

$$\Delta u = f \tag{8}$$

More precisely, we will be working with a discretisation of Poisson's equation obtained through the finite volume method, which splits the domain into a mesh of so called "control volumes" around each node, which are responsible for the name of the method. These control volumes are then integrated over and discretised.

In our case, this results in something described by Equation 1, with right hand side f and solution u being 16×16 grids, which can also be interpreted as vectors of size 256, and the linear operator A behaving exactly like the stencil in Figure 1, which can also be represented by a 256×256 matrix. What we will be attempting to construct is a CNN using TensorFlow that will be able to calculate the corresponding solution u when given a right hand side f, effectively functioning as an inverse of the linear operator A, which puts our network into the category of direct solvers. To achieve this we will be using 6000 different pairs of u and f, which were created using ExaStencils (see [LAB$^+$14]), a specialised code framework that allows for highly parallel equation solving, to train networks with several different architectures. Our primary goal will be to minimise Equation 9, which describes the maximum deviation in one point that our networks result has from the correct solution, acting as an upper bound for all deviations from the correct solution.

$$||u - network(f)||_{max} \tag{9}$$

For this purpose we will take a look at several different possible architectures for such a network, determine the one best suited for this task and then analyse several different versions of that architecture while pursuing our secondary goal of minimising the amount of trainable parameters in the network, which is necessary, because the exact inverse of A could be dense and therefore have up to $256^2 = 65,536$ parameters, rendering any network with more trainable parameters superfluous, since computing A^{-1} would almost certainly be quicker than the lengthy training process CNNs require and generate exact results.

This will result in a network that achieves a balance between our two goals, whose functionality we will then test by applying it to pairs of u and f that still follow Equation 1, but are otherwise completely different form the us and fs used during training.

4 Layers

As mentioned in the previous chapter, we will be constructing convolutional neural networks, which consist of layers. To be able to understand how the resulting networks work requires an understanding of how the layers work on their own and how they are connected. This chapter will deal with the functionality of the layers most commonly used in CNNs on their own, while the chapter about networks will deal with ways they can be connected.

It should be noted that everything described in this thesis was written to describe TensorFlow as accessed through Keras, TensorFlow's high level API for building and training deep learning models. However, transferring the ideas discussed here to another machine learning framework like,

for example, PyTorch should require relatively little work. The following chapters will also stick closely to the terminology used by Keras, which should make it possible to recreate every network described in this thesis with only basic knowledge of Keras.

Common to all layers described here is that they expect a 4D input, with shape (w, x, y, z), where w denotes the amount of incoming samples, x and y denote the shape of the arrays being worked on and z denotes different "channels" of those arrays, which is visualised in Figure 2. w will be mostly disregarded in further explanations of the layers, since it exists to allow layers to receive and evaluate batches of data simultaneously, so there is no cross contamination between values that differ in their w-Position and no difference in computation based on w-position. As such it is only necessary to describe the behaviour for a random, but fixed w-position, since the process will be identical for all other positions.

In image processing the channels denoted in z often start out as different colour values, such as RGB, but for our purposes it is more intuitive to think of the input as having shape (w, z, x, y). This is due to the fact that our initial inputs will be (x, y) arrays with only one channel. All of the proposed networks will split that input into different channels, but those will be more like versions of the same original array that subtly different computations have been executed on. As such, it may be more intuitive to think of z different (x, y) arrays than of one (x, y) array containing z different values at each position. It should also be noted that exchange of information can, will and in our case has to, happen between arrays with the same w-position, but different z-position. It is strictly necessary in our case, since we also expect our output to have only one channel. So all channels have to be reunited once a split has occurred to avoid wasting the computation time spent to calculate the final version of the channels that do not become the output. The informed reader may have noticed that all layers described here can also be passed a "data format" keyword argument, which, if set to "channels_first", would make the layer expect an input of shape (w, z, x, y). But this option has no use beyond that change and since describing every option of every layer used would exceed the scope of this thesis, options will be assumed to remain at their default value unless explicitly stated otherwise. The full list of available options and their effects can be found in the official Keras documentation for the layers [Teaa].

For the sake of brevity z will be used in this thesis to refer to the amount of different channels for a certain input, while z_i will refer to the (x, y) array that is the ith channel of the input. As mentioned before w will be disregarded here and zs with an index will always be in the same w-position. This means that $\sum_{i=0}^{z-1} z_i$ would calculate the sum of all channels for a certain w-position and $z_{i,j,k}$ would refer to the specific value stored at position (j, k) of the (x, y) array that is z_i.

The output of each layer will also be a 4D array with shape (w, x_{new}, y_{new}, o), where every dimension still corresponds to the same thing as in the input and w is even guaranteed to be identical. Whether or not the other values change is down to the specific layer in question. The same things stated for the input also apply to the output and specifically o with indices will be used analogous to z with indices.

4.1 Conv2D

As the name implies, this is an important layer in all convolutional neural networks and the most common layer in all of the networks described here. This is because this layer, occasionally along with Conv2DTranspose, will be the only layer performing significant computations, while the other layers will mainly serve to modify the data in a way that allow this layer to work with it more efficiently.

For each output channel a Conv2D layer by default contains several, one for each input channel z to be precise, two dimensional arrays called kernels and one bias value, which will all be adjusted during training to achieve better results. This means that the amount of learned parameters in this layer follows Equation 10, where k_h and k_w are the kernels height and width respectively.

$$param\_count = o \cdot z \cdot k_h \cdot k_w + o \tag{10}$$

The two parameters of such a layer that do not posses a default value are the size of the kernel and the amount of channels the layer should produce. For the purposes of this thesis we will assume that all Conv2D kernels will have the shape $(3, 3)$, while the amount of output channels o will vary depending on the specific context the layer will be used in.

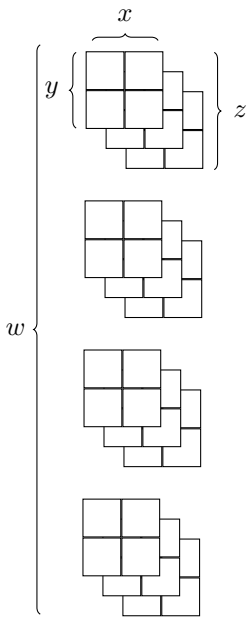


Figure 2: Visual representation of the input for layers with $w = 4$, $x = 2$, $y = 2$, $z = 3$

The output at position $o_{l,m,n}$ can then be calculated in the manner shown in Equation 11.

$$o_{l,m,n} = \sum_{i=0}^{z-1} \sum_{j=-1}^{1} \sum_{k=-1}^{1} z_{i,m+j,n+k} \cdot kernel_{l,i,j+1,k+1} + bias_l \tag{11}$$

It is important to notice that the only difference between two different output channels are the values in their kernels and their biases. In addition the inner two sums can be demonstrated more elegantly using the visuals seen in Figure 3. So the calculation can also be thought of as "moving" the relevant kernel over an input channel, calculating the sum of the product of the overlapping values and accumulating the results in the output repeated for all channels in the input before adding the bias.

It should also be mentioned here that this description and Equation 11 is somewhat incomplete, since it neglects the activation function, which would be applied to $o_{l,m,n}$ after the bias has been added. This thesis will not consider activation functions further, since the solution u and the right hand side f of our problem are connected through the matrix A, which means that a network should be able to calculate u from f while remaining linear. It should not be ignored, however, that activation functions can improve the training efficiency and the accuracy of the results of networks that use them, which is why papers like [RZL17] exist, which compare and contrast the effect of different activation functions on a network.

Already visible in Equation 11 and even more obvious in Figure 3 is the fact that this process creates an output that is missing a "ring" of cells along the edges as compared to the input. This is slightly problematic for our application, since our final output is required to have the same shape as our initial input. This can be avoided by instructing the layer to pad the input, resulting in a ring of zeros to be added before starting the convolution process. However, this means that the values along the edges of the output are the result of less "true" data, which could be avoided by combining this layer without padding with other layers, described later, that increase the x- and y-size of the input they receive. This is why some layers in the networks described later will use padding and some will not.

On the other hand, a reduction in input size is occasionally completely intentional and even something we later introduce separate layers for. This is due to the fact that a 3×3 kernel can only take adjacent cells into account. So it takes many more convolutions to spread information through a big input than through a small one. One way that allows decreasing the size of the input quicker

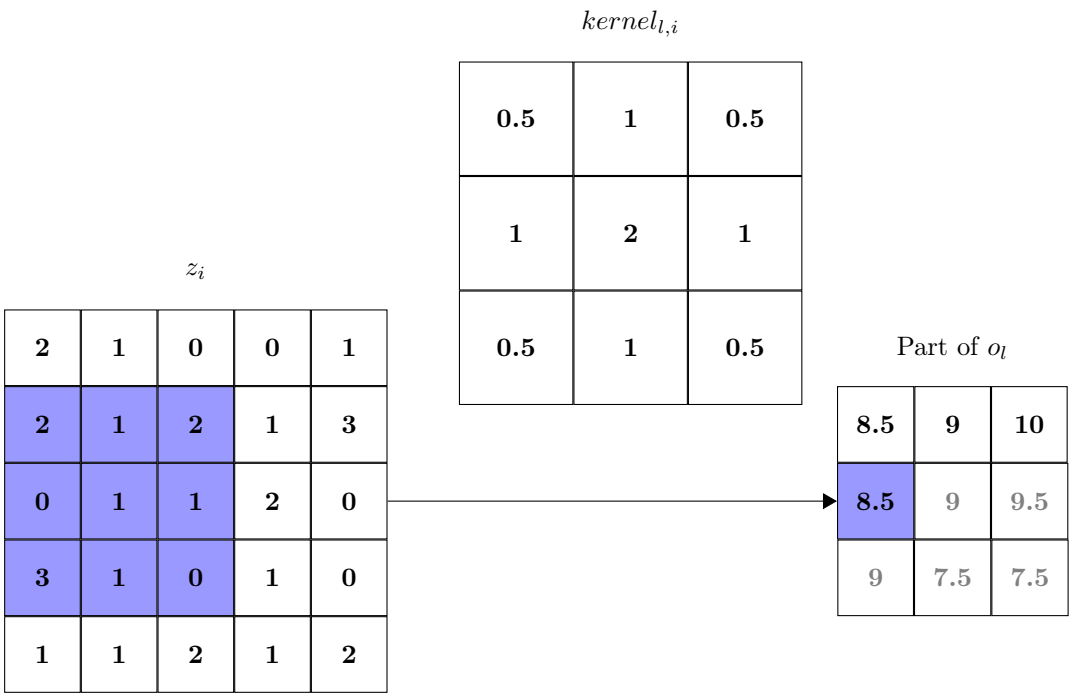


Figure 3: Part of the calculation of the output of a Conv2D layer
Example computation in blue, values that "have not been calculated yet" in grey

is by using strides, which change how far the kernel moves after each completed calculation, which is depicted in Figure 4, which shows the location of centres of 5×5 kernels moving over a 9×9 input with strides of two with and without padding. Special care has to be taken with strides and without padding. This can be seen in the left half of Figure 4, where the cells along the right and lower border are discarded because the previous kernel didn't use it and the next kernel position is skipped because it would require values beyond the right and/or lower border. Strides also interact possibly unintuitively with padding, since it can change which cells the kernel will be centred on, as seen in the same figure. Furthermore, it is possible to define different strides in x and y direction. However, this will not be done in this thesis, as there is no inherent difference between x and y so we have no reason to treat them differently.

4.2 Max- and AvgPooling2D

As mentioned in the description of the previous layer, reducing the size of the input can be advantageous. This is partially because it increases the speed at which Conv2D layers can propagate information through the arrays they are working on and partially because reducing the amount of values in the output reduces the amount of times Equation 11 has to be evaluated, thus speeding up the network. In addition to the ways described in the section on Conv2D, this can also be achieved through the use of the pooling layers MaxPooling2D and AvgPooling2D, which create their output through Equation 12, where op is max or avg depending on the layer.

$$o_{l,m,n} = op([z_{l,2m,2n}, z_{l,2m+1,2n}, z_{l,2m,2n+1}, z_{l,2m+1,2n+1}]) \tag{12}$$

As can be seen in that equation this layer keeps the different channels separate from one another and does not alter the amount of channels, but reduces x and y to $\lfloor x/2 \rfloor$ and $\lfloor y/2 \rfloor$ respectively. This is also depicted in Figure 5 for MaxPooling2D. It should be noted here that a Conv2D layer with a 2×2 kernel and strides of 2 can perfectly emulate the behaviour of the AvgPooling2D layer by following Equations 13.

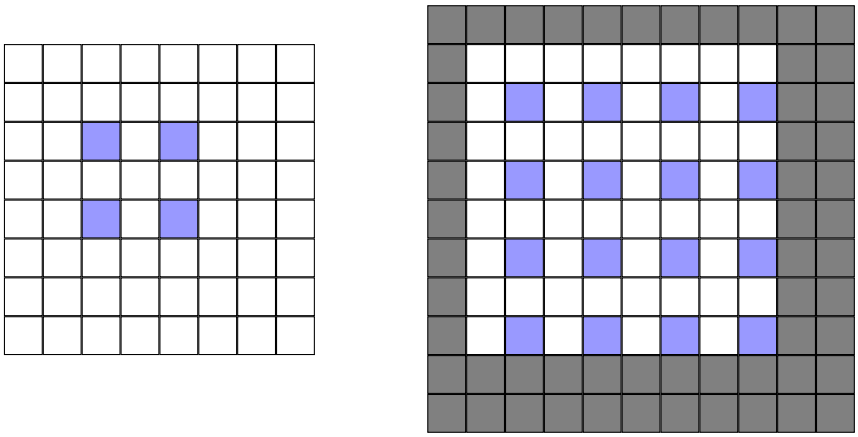


Figure 4: Representation of the kernel centres in the input of Conv2D with strides of 2 with and without padding and a kernel size of 5×5
Added padding in grey, blue cells are the centres of kernels, Outputs would have a size of 2×2 and 4×4 respectively

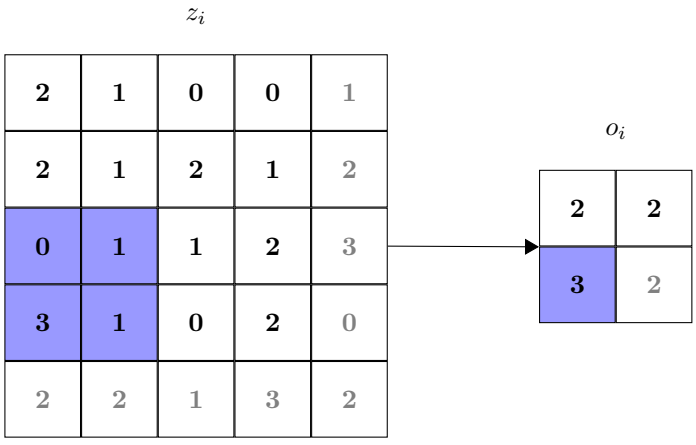


Figure 5: MaxPool, Sample calculation in blue, unused/"not yet calculated" values in grey

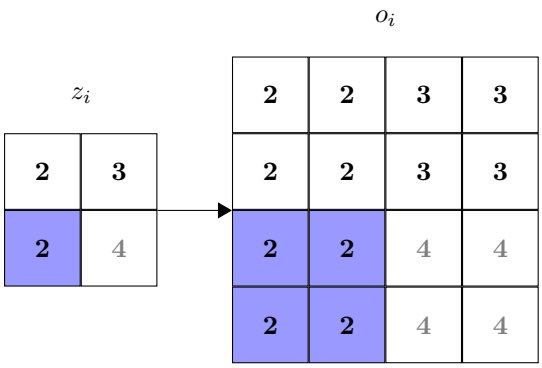


Figure 6: UpSampling, Sample calculation in blue, "not yet calculated" values in grey

$$\begin{aligned} kernel_{l,i,m,n} &= 0.25 \text{ if } l = i \\ kernel_{l,i,m,n} &= 0 \quad \text{if } l \neq i \\ bias_l &= 0 \end{aligned} \tag{13}$$

So replacing AvgPooling2D layers with such layers is completely reasonable, since even in the worst case scenario the resulting network would be at least as good as the one using AvgPooling2D. This does not apply to MaxPooling2D, but could still be attempted, which is exactly what is described in [SDBR14], creating an "all convolutional network" in the process. We will be mostly ignoring this possibility however, since we are also attempting to minimise the amount of learned parameters in our network and the pooling layers require no learned parameters, while the proposed replacement still follows Equation 10.

4.3 UpSampling2D

As mentioned in the section on the Conv2D layer, all the processes to reduce the size of the input just described pose a problem for us, since our output has to have the same shape as the input. One way to fix this can again be found in image processing, where the simplest way to increase the size of an image is to replace every pixel with a square of several pixels set to the value of the original pixel. The UpSampling2D layer does just that with a size of 2×2 for the resulting square, which is also depicted in Figure 6. One of the downsides of this layer is the fact that can only change the image size by an integer factor. So applying this to an input whose size is not a fraction of the target size would "overshoot", requiring the usage of the previous two layers to reduce the size again. This problem, however, can be avoided by using the Conv2DTranspose layer.

4.4 Conv2DTranspose

This chapter is largely based on [DV16] and [Lan], the latter of which was originally written for ApacheMXNet but also applies to Keras/TensorFlow. Padding in Keras, however, does not follow the rules described in any of those two texts.

An analogue of the Conv2D layer, but functioning in the other direction is the so called Conv2DTranspose layer. It is important to note that the "Transpose" in the name references the fact that this layer will revert the reduction and x and y that will occur in the previously described Conv2D layer without padding if no cells were ignored due to strides. This does however not mean that this layer acts as a transposition or even inverse of the Conv2D layer in a mathematical sense. Instead it produces an output that follows equation 14 with nonexistent values assumed to be zero.

$$o_{l,m,n} = \sum_{i=0}^{z-1} \sum_{j=-1}^{1} \sum_{k=-1}^{1} z_{i,m+j,n+k} \cdot kernel_{l,i,1-j,1-k} + bias_l \tag{14}$$

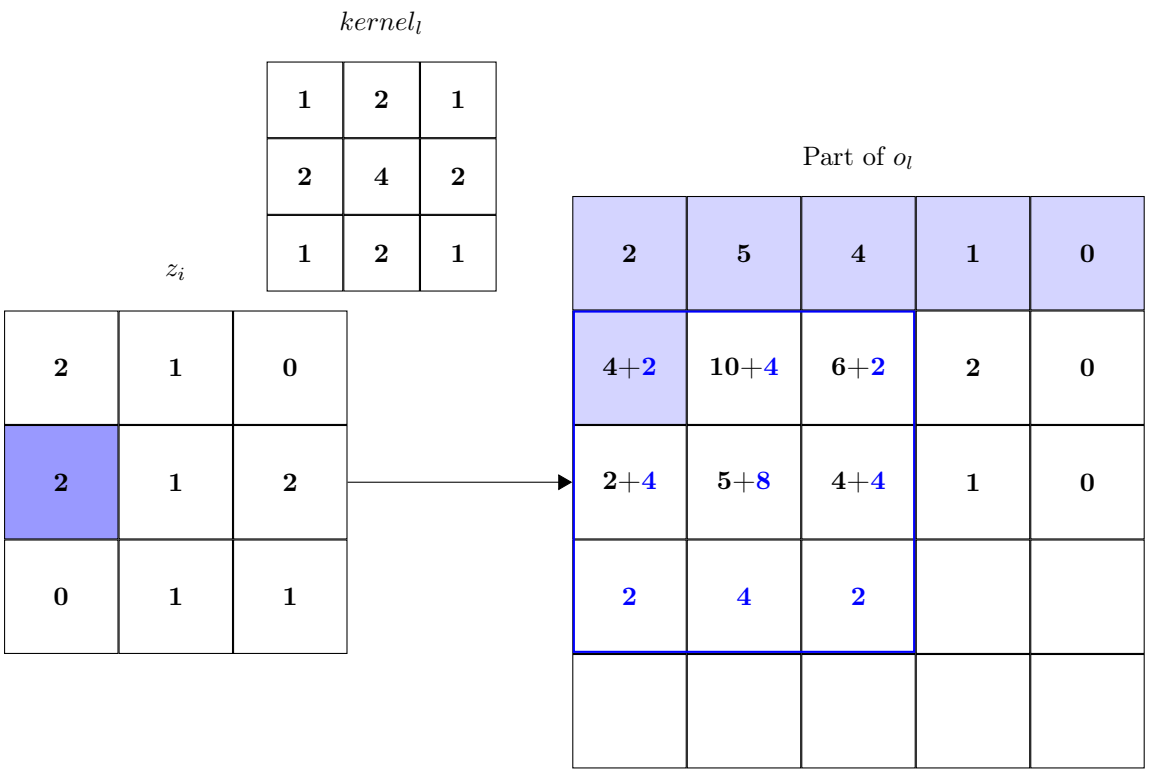


Figure 7: Part of the calculation of a Conv2DTranspose layer from the "distributing the input" point of view
Example computation in dark blue, cells that will not be altered further in this part in light blue

While this could easily mistaken for Equation 11 at first glance it should be noted that the indices for the kernel have changed, resulting in it being flipped along both x and y axis, which additionally means that the amount of trainable parameters in this layer also follow Equation 10.

Another way of thinking of it can be seen in Figure 7. With that approach every cell in the input is spread out using the kernel and accumulated in the output. Regardless of how one chooses to think about that process, the output on a certain channel is still the sum of this process for all channels with a bias added, while different channels are still calculated in the same manner, but with different kernels and biases.

How padding and strides work is also the opposite of what would be expected from Conv2D. Strides increase the size of the output as seen in Figure 8. They can however be relatively easily understood when thinking about this layer as spreading out the input since strides in that framework just adjust where the results are written to in the way you would expect from strides. Padding on the other hand reduces the size of the output instead of increasing it by fixing it to the size of the input multiplied by the amount of strides. How exactly that is achieved is beyond the scope of this thesis, since none of the networks described in this thesis utilise padding in Conv2DTranspose layers.

One problem with this layer can already be seen in Figure 7: It can very easily create so called "checkerboard artefacts", which is also referred to as the "checkerboard effect" and visualised in Figure 8. This occurs because cells along the edges have fewer neighbours and are therefore reached by fewer kernels than cells that are surrounded on all sides. While it would be possible for the network to learn kernels that have lower values in areas where overlapping occurs to mitigate this effect, there is no guarantee that this will actually happen.

It is, however, possible to circumvent this problem by choosing a kernel size that is identical to, or lower than, the amount of strides. This prevents the kernels from overlapping and makes this layer behave like the UpSampling2D layer, except with learned weights. A more extensive description of checkerboard artifacts and ways to avoid them can be found in [ODO16].

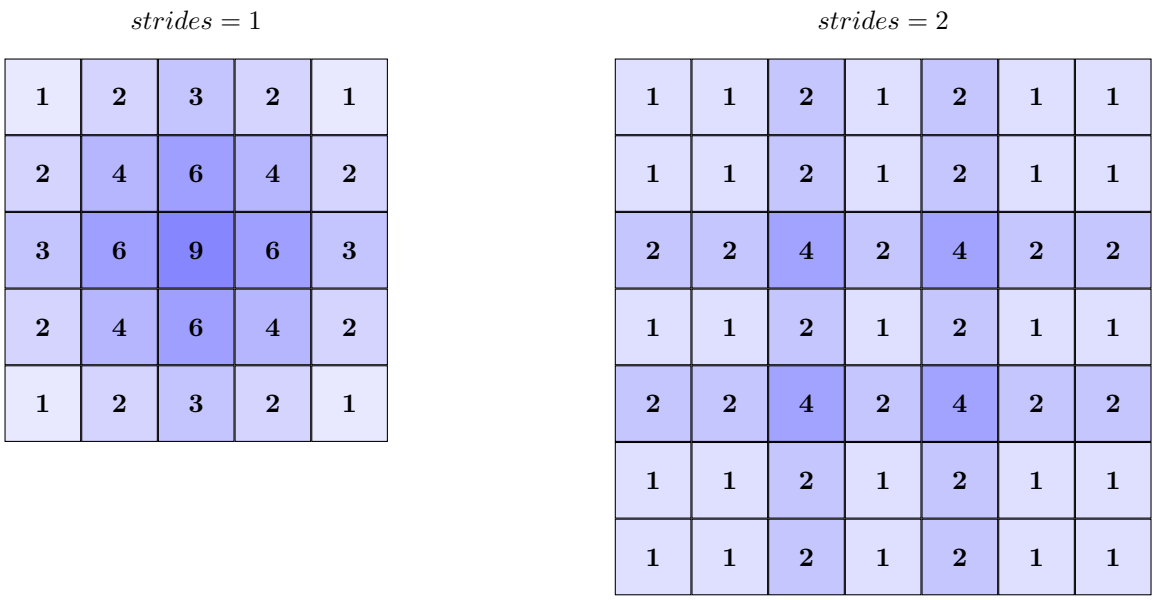


Figure 8: Depiction of the checkerboard effect that can be caused by Conv2DTranspose layers Input and kernel are both assumed to be 3×3 arrays filled with ones, colour for emphasis

4.5 Concatenate

Being more of a tool than an actual layer, this layer is noteworthy for having both multiple inputs and not really performing any calculations. As the name implies the Concatenate layer simply combines all of its inputs resulting in an output that is identical to the inputs with concatenated channels.

Since this would be impossible for differently shaped inputs this layer is the only layer described here that requires more of its input than being a 4D array. It should be noted here that while care has to be taken to ensure that x and y are identical for all inputs the same does not apply to the batch size w and the amount of incoming channels z. Since none of the layers described here interact with w it is already guaranteed to be identical for all inputs. z however is allowed to differ between inputs, since computing the concatenation of lists of different lengths is not a problem.

One of the problems of reducing the size of the input is the loss of information that comes with it. As such, increasing the size again will inevitably cause a loss of detail. This layer allows us to create so called "skip connections", linking later layers that would normally receive a heavily processed version of the input back to the original input, which will hopefully allow those layers to take advantage of the detail that was present in the beginning. Since this explanation of why this layer is used at all has already drifted into an explanation of how networks will utilise it the next chapter will cover just that.

5 Networks

This chapter will focus on the different networks that can be created from the parts described in the last chapter. While it would be possible to describe the following networks with just words, for example with the code used to create them, each subchapter will instead feature a visual depiction of the described network. Each layer will be represented by a box describing the layer used, with connections between the layers represented by arrows. The only exception to this rule is the Concatenate layer, which will be represented by any other layer simply having multiple inputs. The y-axis will be used to represent the input size of the respective layers in both x and y, so a layer twice as far down will have an input with one quarter the size.

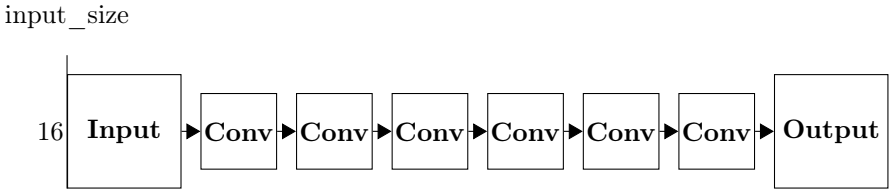
As mentioned previously, one of the factors that interest us in our networks is the amount of trainable parameters they posses, which is the sum of Equation 10 for all Conv2D and Conv2DTranspose layers in the network, since all other layers used in this thesis have no trainable parameters.

This can be calculated if the amount of input and output channels and the kernel size for each layer are known, but will result in a polynomial of degree two in the amount of channels, even if the kernel size is assumed to be constant and the amount of input and output channels are assumed to be identical for all but the first and last layers in such a network. This makes it difficult to draw conclusive results from the comparison of the amount of parameters in different networks, since changes in those parameters will affect them differently, which is why the following descriptions will assume set kernel sizes and channel counts. Each layer depicted will be assumed to operate with 3×3 kernels, since the linear operator A of our problem could also be represented by such a kernel, and 32 output channels, which has proven to be an appropriate amount of channels during experimentation. Additionally, each layer will be assumed to otherwise use its default arguments. Exceptions to this are the padding of all Conv2D layers, which will be assumed to be on even though it defaults to off, the last Conv2D layer in each network, which will only feature one output channel, since our expected result also consists of only one channel and the pooling layers including UpSampling2D, which will use their default pool size of 2×2. Further deviations for specific networks will be noted in the box belonging to the altered layer. These assumptions also describe the networks that will be used at the start of the chapter dedicated to the results of our experiments.

This means that each Conv2D and Conv2DTranspose layer will use $32 \cdot 32 = 1024$ kernels, with the exception of the first and last layer in each network, which will use 32 kernels and all layers that receive a skip connection, which will use $(32 + 32) \cdot 32 = 2048$ kernels.

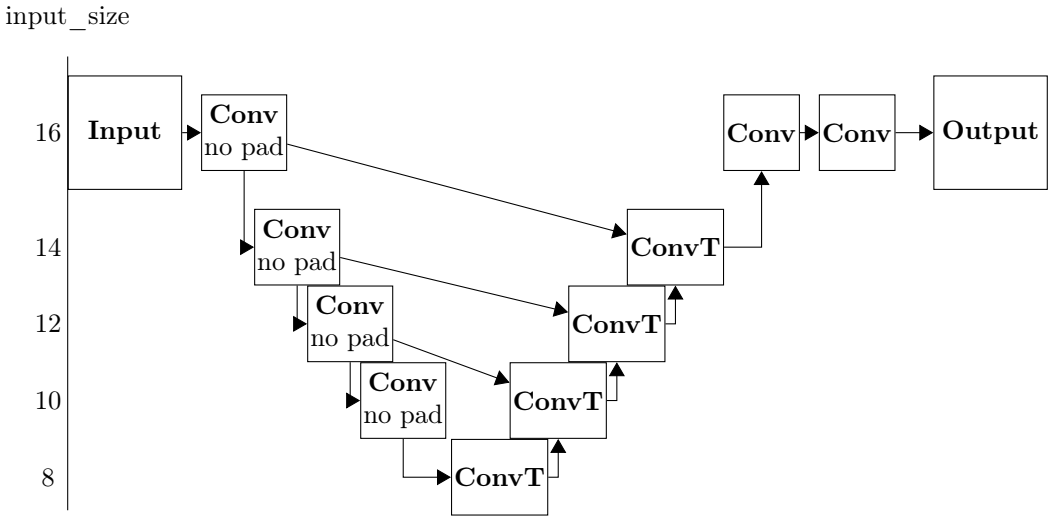
5.1 Sequential Convolution Network (scnet)

This network simply consists of sequential convolutional layers, hence the name. One of the easiest constructions to come up with, this network features 37,601 trainable parameters. For reasons briefly touched upon in the chapter on layers this network is expected to perform fairly badly. However, this is entirely intended in this case, since it allows us to have a sort of "upper boundary" for the performance of our networks, which in turn allows us to take a very critical look at the changes made, should a more complicated network ever perform worse than this one.



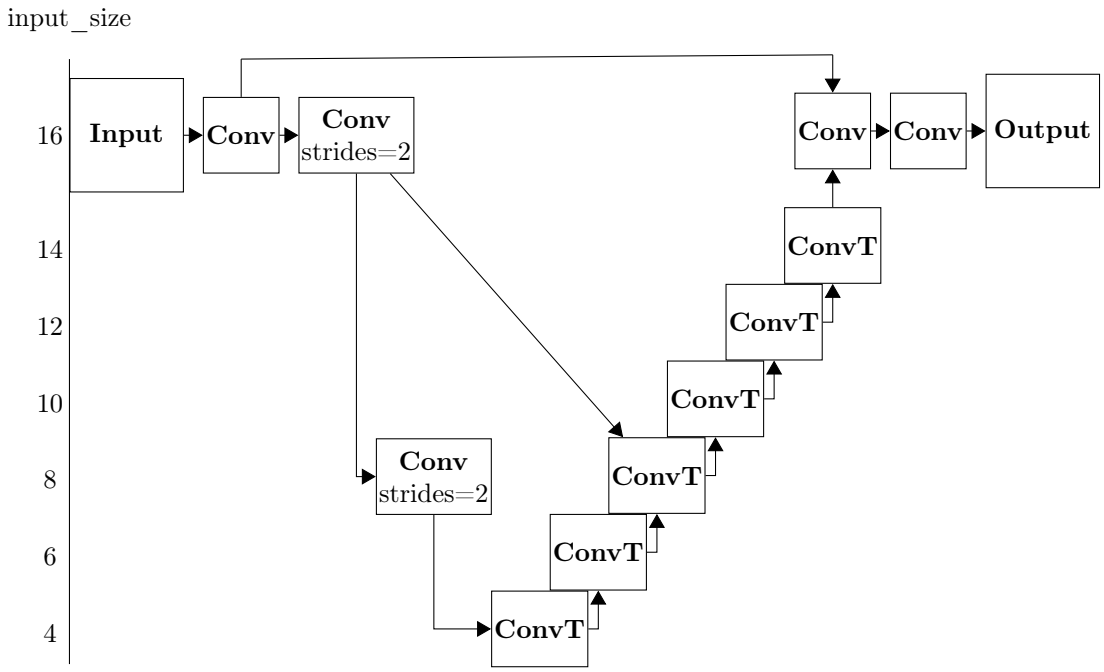
5.2 Slow Descent Network (sdnet)

An attempt at reducing the size of the input by simply not using padding, resulting in a comparatively slow descent, which is what this network was named for. This allows for many skip connections, but also requires many layers, resulting in 92,993 trainable parameters for this network. The fact that this network has about 2.5 times the amount of trainable parameters as the previous network while only using slightly more than 1.6 layers is due to the fact that in our networks skip connections double the amount of incoming channels for the receiving layer and therefore increase the amount of trainable parameters that layer has significantly. It should be noted that this architecture can only change the shape of the input by two in each direction in each layer.



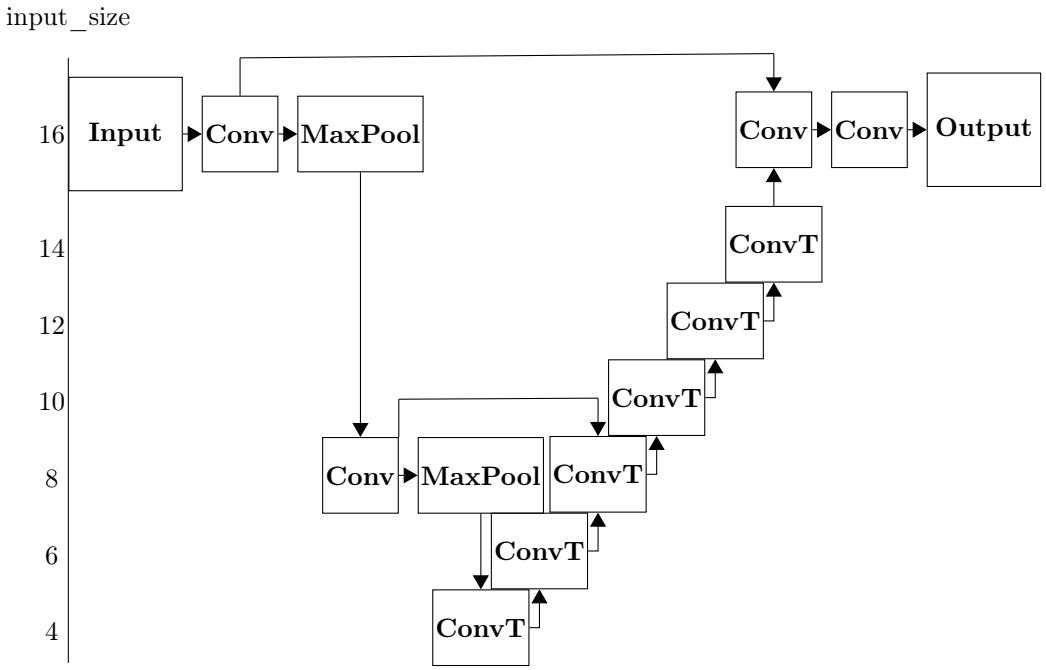
5.3 Stride Network (snet)

The next step to reduce the amount of layers in the network while reducing the size of the input. The chain of Conv2DTranspose layers to get back to the original size still exists, but the chain downwards has been replaced by Conv2D layers with strides, resulting in this networks name. This network features 102,273 trainable parameters. 9280 more than the previous network, which is only slightly more than the 9248 parameters one typical Conv2D or Conv2DTranspose layer has in these architectures. This is because the minimum size this network achieves is 4×4, while the previous network only went down to 8×8. While this should result in a better performance, there is a possible problem in the specific way this is achieved here. The usage of padding results in all cells in the output an especially the ones in the corners to be calculated with data that was not part of the original input. While this is true for all other networks too, this specific network uses strides to reduce the size of the input to 4×4, which causes 10 of the 16 values at the lowest level to be the result of calculations with "made up" data. The next network will perform a slight modification to mitigate this.



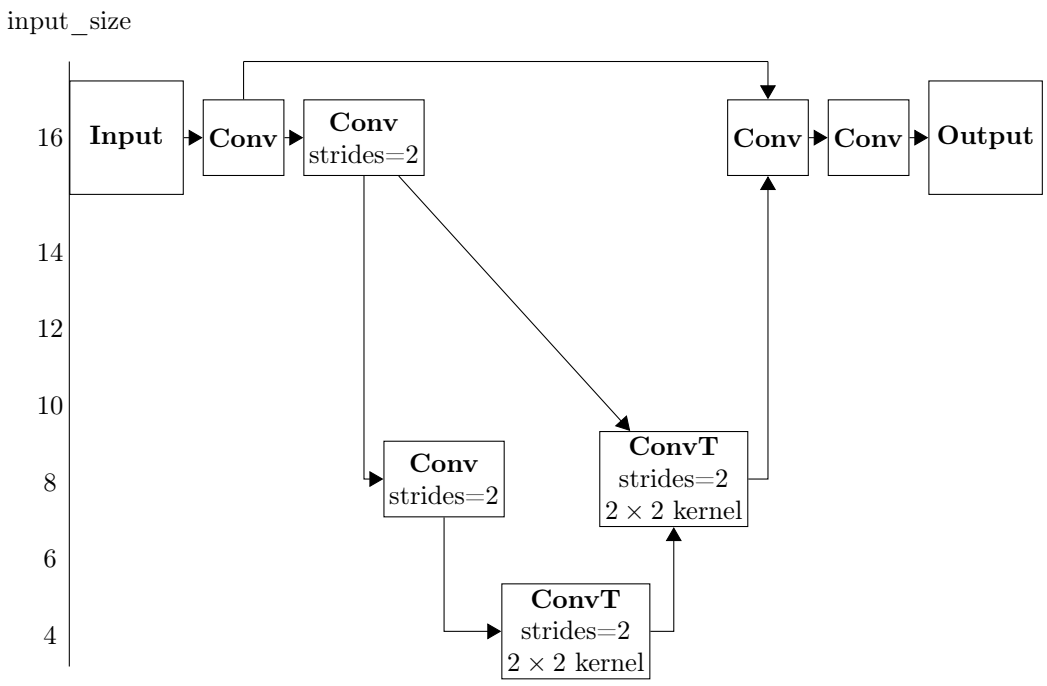
5.4 Max Pooling Network (mpnet)

A modification of the previous network, using MaxPooling2D, which is responsible for the name of this network, and one fewer Conv2D layer resulting in 93,025 trainable parameters. This mitigates the problem described in the section of the previous layer because the values used for padding are zeros, which means that values calculated using the padding are very likely to be lower than other values, which will cause the MaxPooling2D layer to pick those "pure" values instead. The next step in reducing the amount of trainable parameters will be rectifying the chain of Conv2DTranspose layers currently used to get back to the original size.



5.5 Advanced Convolutional Network (acnet)

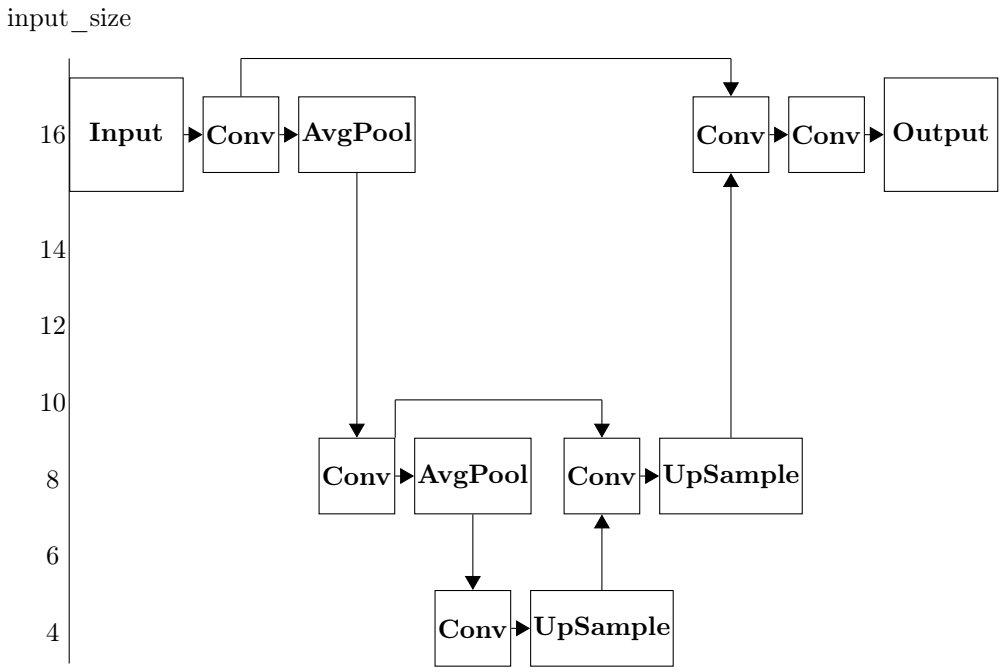
The most obvious way to address the chain of Conv2DTranspose layers present in the last three networks is to replace them with Conv2DTranspose with bigger strides, drastically reducing the amount of used layers and the amount of trainable parameters to 49,921. Additionally, care has been taken to avoid the checkerboard effect described previously by setting the strides and the kernel height and width of those layers to the same values. It should be noted that the change in the last network that added MaxPooling2D layers has been reverted to create a network consisting purely of Conv2D and Conv2DTranspose layers, which will allow for a more direct comparison between this network and the first three networks described in this thesis, since they also only contain Conv2D and Conv2DTranspose layers. This is also the reason why this network is named "Advanced Convolutional Network".



5.6 Grid Network (grid)

All of the networks described so far were quite general and inspired by considerations about what should work in general and what configurations would be interesting to look at. Another source of inspiration for network architectures that should be considered are methods for solving PDEs that are already known to work, which is why this network is loosely inspired by the multigrid methods briefly described in the chapter on PDEs.

Coincidentally it is also a sort of synthesis of the last two networks described. Taking the usage of pooling layers and their natural reverse the UpSampling2D layer from the first one and the quick descent to small inputs and swift rise back to the original size from the last one. Perhaps unintuitively this network has 56,033 trainable parameters. More than the last network even though it has one fewer Conv2D layer, since the last network contained two layers using 2×2 kernels, which reduced the amount of parameters in that network noticeably.



6 Training

Having covered the architecture of several different networks and the amount of trainable parameters they posses, this section will cover how these networks were actually trained. How well, how long and on what data a network was trained is very important to a network, since the same architecture described here could for example be used for image processing if trained differently. This of course also means that it is possible to train a network incorrectly, resulting in bad performance even if the architecture could theoretically achieve much better results.

To be able to train a network to provide accurate results it is necessary to provide large amounts of data it can learn from. This data usually comes in pairs of the input to the network and the output the network should produce with that input. Exceptions to this exist however. For example [HZE+19] describes the attempt to train a network to compute a linear operator that could be used to improve an already known iterative process, but that is not a thing this thesis will attempt, instead using 6000 pairs of 16×16 grids corresponding to right hand side f and solution u obtained from the discretisation of the Poisson Equation using the finite volume method, assuming f to be a combination of trigonometric functions and solving for u.

It should be noted here that there are two common groups of problems that artificial neural networks are trained to solve. Classification problems deal with sorting the input into distinct groups e.g. whether the image given to the network contains a face or not. This would require success to be measured by how often the network is correct, in other words by how accurate the network is.

We are trying to solve a regression problem however, where what we are trying to predict is a continuous quality and we can measure success by how far the network deviates from the correct solution. As such the training loop, which will be examined in more detail later, can be described in the following way: The majority of the provided data, referred to as “training data”, which makes up 90% of the provided data in our case, is separated into smaller batches and fed to the network batch by batch, which calculates its output and a single number referred to as loss that somehow corresponds to how far the networks output deviated from the correct solution. This will shortly be explained in more detail. This loss is then used by the optimiser to adjust the weights in the network before the process is repeated for the remaining batches. After all batches have been processed the loss is also calculated for the rest of the data, which is referred to as “validation data", but without the network being updated based on it. This updateless loss is called “validation loss” and one iteration over the entire data set is commonly referred to as an epoch. This loop is often repeated

for a set amount of epochs, but can also be ended after a certain amount of time has passed, if the loss doesn't show significant improvements anymore or if the loss and the validation loss start deviating significantly from one another.

That last one is also called "overfitting" and is a sign that the network has "memorized" the answers for the data used to calculate the loss, resulting in great results for that data, but does not know how to solve the problem in general, since it would also perform well on the data used for validation if that was the case. If the data came in pairs of input and correct solution such a network is completely useless, since it can only calculate an accurate approximation of the solution for those inputs, where the exact solution is already known. The validation loss exists solely to detect and avoid this. Either by ending the training before loss and validation loss start to deviate, or by changing the entire training process to make it less susceptible to overfitting.

Most of our results will be obtained from the best configuration the networks achieved during 10,000 epochs of training, where "best" is measured by the minimum value achieved by Equation 9, while ensuring that no overfitting has occurred. This limit of epochs has been chosen more or less arbitrarily to provide an end point that allows for reasonably fast changes to the training process. It should be noted that this is not necessarily the point at which the network has finished training. In general it is not easily possible to discern if and when a network will achieve a better configuration than the current one during training, since the loss is not guaranteed to fall monotonously, which is why the previous list of end criteria did not contain the option to end training once no improvement will be achieved anymore. Since our first set of experiments exist to compare networks against each other, risking to end their training early is reasonable, since it can be assumed that all the networks described here will show similar improvements if trained for longer. Once a good network and training configuration has been found training will continue until it is reasonably certain that further training will not create better results.

Another part of the previous description that requires further explanation is the optimiser, which attempts to minimise the loss of the network during training. This thesis will not cover the exact functionality of optimisers, nor were any networks trained with an optimiser other than Adam with its default parameters aside from the learning rate, whose documentation can be found in [Teaa]. The learning rate is a parameter common to all Keras optimisers and determines how quickly the learned values are changed. A high learning rate will result in bigger updates and therefore shorter training times, but bears the risk of the network being optimised for the most recent batch while "forgetting" everything it learned from other batches. A lower learning rate will make sure that things that have been learned aren't thrown out without good reason, but will lengthen the time required to reach good results in the first place. It should be noted that the experiments performed for this thesis only tested a few different learning rates.

Also something that has not been experimented with extensively, but is important to understand nonetheless, is the size of the batches the data gets split into. Updates to the network can only occur after a complete batch has been processed, which would favour separating the input into as many batches as possible, so batches with size one. The problem with that approach is that it will cause the loss reported for each batch to fluctuate wildly, more or less independently of the values the network has learned so far, which can cause the optimizer to "correct" values that worked well in general, but not for a specific batch. As such, it is important for the batches to be big enough to be representative of the entire data set. This variable batch size is the reason for the first dimension of the input w mentioned in the chapter on layers and also the reason there is no interaction between different w's which allowed us to mostly ignore it. This also makes losses slightly more difficult, as the single number that will be used by the optimiser has to represent all of the data points in a batch. For an assumed batch size of 30 and our 16×16 grids that's one value representing 7,680 values. Should the batch size for some reason be set to 5400 it would result in the loss having to represent 1,382,400 values. Another reason to keep the batch size small.

One important factor of this that still hasn't been explained however is how said loss is calculated, which is also vital since it will influence how well the optimiser will be able to perform, thus affecting how well the network can be trained. During the creation of the network a so called loss function has to be specified. An example of such a function can be seen in Equation 15, where y_{pred} is the networks predicted solution and y_{true} is the correct solution, which shows the loss function mse, for mean squared error, one of several commonly used loss functions already implemented in Keras.

$$mean((y_{pred} - y_{true})^2) \tag{15}$$

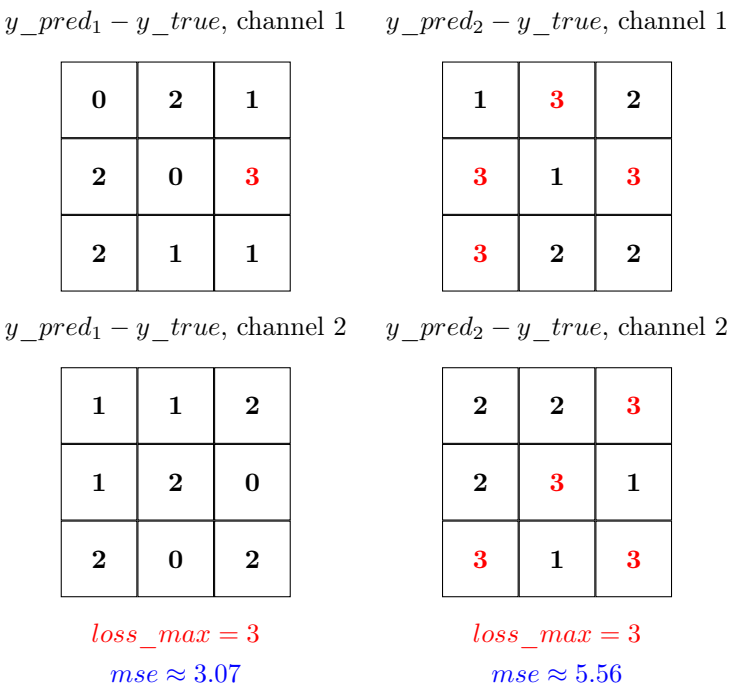


Figure 9: Depiction of the problems with $loss\_max$
Maximum deviations in red, results of the two loss functions below the predictions, assuming only two channels

As stated in the chapter on PDEs, our goal is to minimise Equation 9 for our entire dataset. To achieve this we set up the network to use it as its loss function, from now on referred to as $loss\_max$ with can be calculated with Equation 16, where y_{pred} is the networks predicted solution and y_{true} is the correct solution, and one batch set to contain all 5400 datapoints intended for training the network.

$$max(|y_{pred} - y_{true}|) \tag{16}$$

As mentioned previously however, using a sample size that large is not efficient and that is not the only problem with this loss function. Since $loss\_max$ returns the maximum values present in any of the 5400 16×16 grids its result is solely determined by one of 1,382,400 values. This is a problem, since it results in the optimiser considering many different results to be as good as one another even if some would be strictly better than others using any other metric. An example of this can be seen in Figure 9. This will also be confirmed experimentally, which will result in mse being used as the loss function with a batch size of 30 instead.

This is also the reason why we will use a test data set for comparing the different networks that is simply the combination of the training and validation data sets. Usually comparisons between different networks are done through a separate test data set that represents the entire breadth of the problem. This is done because comparing networks based on their performance on the training data would not notice overfitting and because the split between the training and validation data sets are often random, which could result in some networks achieving a better validation loss simply because they ended up with simpler problems in their validation data set. But using mse instead of $loss\_max$ leaves us in the somewhat unique position of grading the performance of our networks based on something they were not explicitly trained for, which allows us to reuse the combination of the training and the validation data set as our test data set.

It should be noted here that this change in loss function is not guaranteed to work. It is only possible in the first place because the most efficient way to reduce mse is to reduce the maximum deviation, thus reducing $loss\_max$. We only know that it works in our case because we trained the networks both with $loss\_max$ and mse and compared the results.

7 Results

One of the problems the analysis and comparison of convolutional neural networks faces is the reproducability of the training process. Python itself as well as TensorFlow have sources of randomness that can be seeded, but training such a network on a CPU is significantly slower than training it on a GPU, which is why the training for this thesis was performed on a GPU using the Nvidia Cuda deep neural network library cuDNN. The problem with that is that cuDNN, as stated in the official documentation [NVI], does not guarantee reproducibility for all of its operations. As such, training the same network with the same data will in all likelihood result in very similar, but not the exact same results. This also applies to different networks, making it slightly more difficult to compare them, since the network with the better end result might just have gotten "lucky". This is thankfully mitigated somewhat because the results for each network that we will be comparing against each other is their best performance during all of the 10,000 epochs of training. This should, statistically speaking, cause all of the "positive luck" and "negative luck" to cancel out, resulting in comparable values.

This still doesn't remove this randomness completely, but the following results will show a strong difference in performance of different groups of networks that is very unlikely to be caused by said randomness. It should be noted here that "performance" will be used in this chapter to refer to the maximum deviation, the result of Equation 9, our network achieves on out test data set, which, as stated in the previous chapter, consists of all 6000 pairs of solution u and right hand side f that were created for the purposes of this thesis. As such "maximum deviation" will also be used as a shorthand for "maximum deviation on the test dataset".

All of the following results were achieved using a Intel® Xeon® E5-1650 v4 processor and a NVIDIA GeForce GTX 1080 graphics card using Python version 3.7.3, Keras version 2.2.4, TensorFlow version 1.14.0 and cuDNN version 7.6.0. With that configuration training one network for 10,000 epochs took roughly 6 hours, while the continued training for 100,000 epochs usually took around 2.5 days, but once finished the evaluation of the entire test data set can be computed in a few seconds. These times were measured on GPU and only very brief tests were run on CPU, which resulted in 1.5 to 2 times longer computation times. This is not surprising, since it has been shown time and time again, e.g. in [LMR+17], that running networks on GPU is significantly faster.

7.1 Architecture Comparisons

Figure 10 shows the maximum deviation in one point for all the networks described in this thesis when trained with $loss\_max$. The Sequential Convolution Network's poor performance is entirely expected, as is the fact that the Grid Network achieves the best performance. The very similar performance of the Stride Network and the Max Pooling Network is also not surprising given their very similar architecture.

Interesting to note however is that the Slow Descent Network can perform about as well as the last two networks mentioned, which is significantly better than the performance of the Advanced Convolutional Network. This is possibly because the amount of trainable parameters in the three networks that perform very similarly doesn't differ by more than 10,000, while the Advanced Convolutional Network has about 50,000 fewer than them. In general, if one were to order the networks by performance and by trainable parameters there would only be two differences. The Max Pooling Network performs slightly better than the Stride Network despite boasting about 9,000 fewer trainable parameters with a very similar architecture, lending credence to the idea that the MaxPooling2D layers help to mitigate the problems of padding. This is not definitive evidence that this is actually the case however, since this slight difference in performance could also have been caused by the randomness mentioned at the start of this chapter. Experiments described later in this thesis resulted in a better performance for the Stride Network, which makes that seem even more likely.

What is very significant and noticeable however is the performance of the Grid Network, which is noticeably better than the rest even though it has about 43,000 fewer parameters than the next best network. This is exactly what we hoped for, since the design of the Grid Network drew inspiration from methods for solving PDEs that are already known to work. It should be noted here that further experiments will make it clear that increasing the number of trainable parameters will not automatically result in better performance and vice versa.

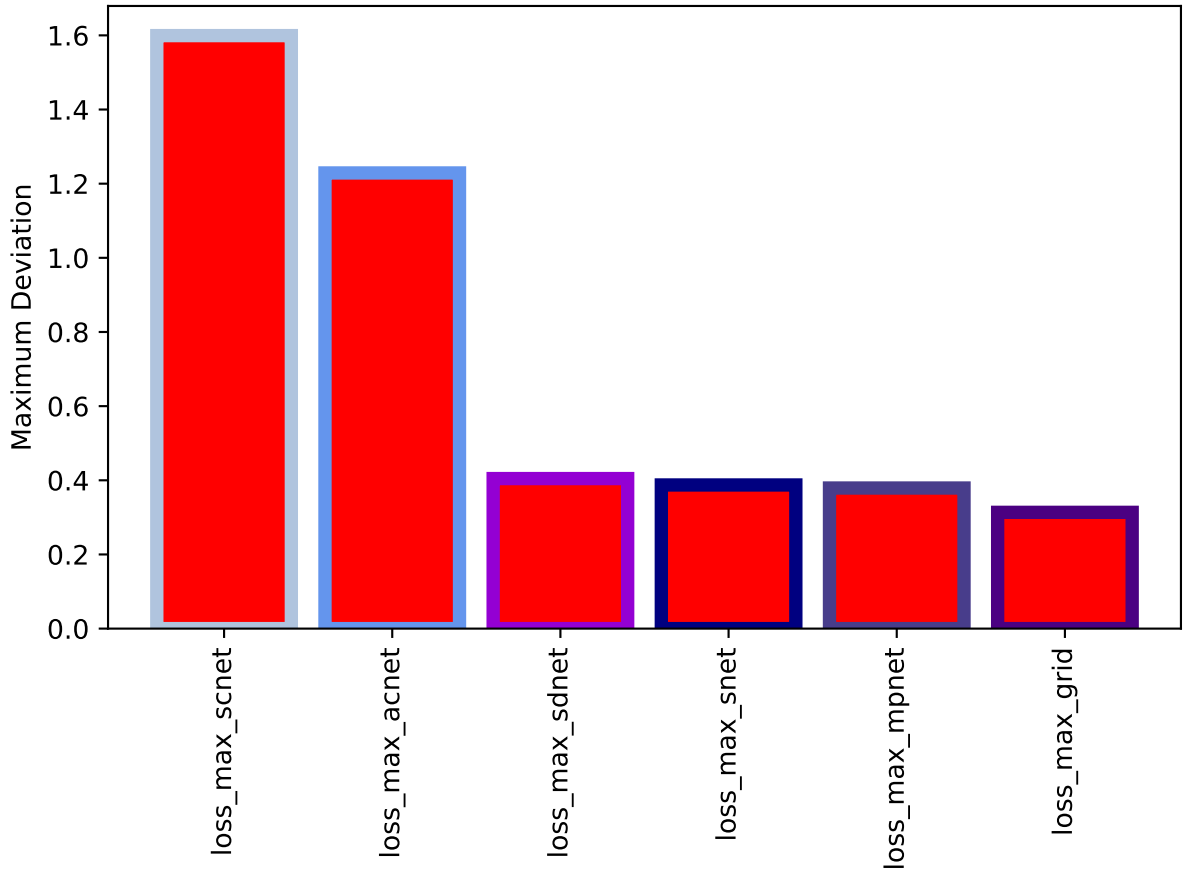


Figure 10: Training results using *loss_max*

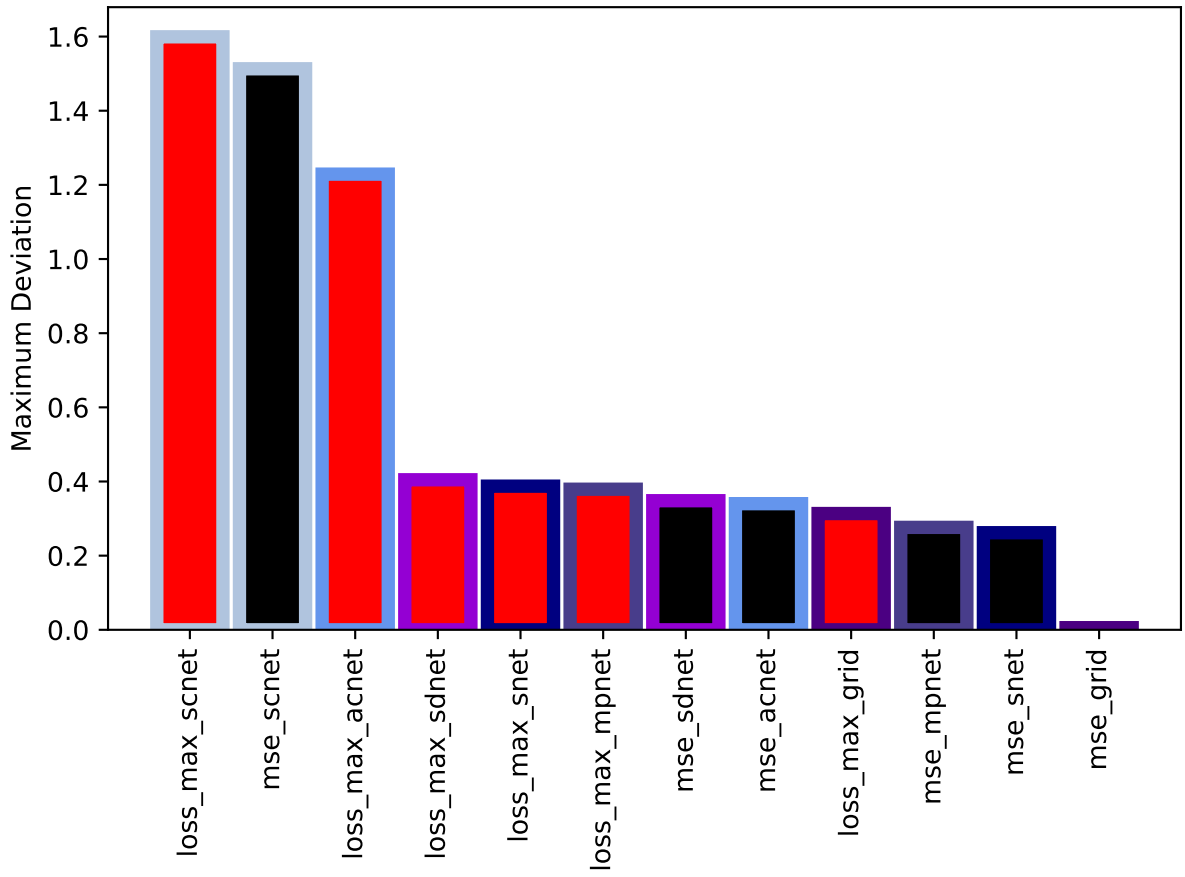


Figure 11: Training results using *loss_max* and *mse*
Networks using the same architecture have the same border colour, central colour corresponds to the loss funtion used

There are several theoretical flaws with the loss function *loss_max* and the fact that it requires the maximum possible batch size as already explained in the chapter on training. For that reason all of the networks described in this thesis were constructed again and trained in the same way as the ones that resulted in Figure 10 were, except using *mse* as loss and with a batch size of 30. The results of that, in addition to the results of the first set of networks, can be found in Figure 11.

As predicted, the networks trained with *mse* consistently perform better than the ones trained with *loss_max*. It is however interesting to note that the Max Pooling Network and the Stride Network changed ranks again, making it seen likely that they are about equal in terms of performance with one performing better than the other based on chance. Also noteworthy is the significant increase in performance of the Advanced Convolutional Network, allowing it to pull ahead of the Slow Descent Network. This is especially interesting since it shows that different networks might respond better to certain loss functions than others.

These last results still very clearly point to the Grid Network being the best architecture for this task, which conclusively ends our search for a baseline architecture.

7.2 Trainable Parameter Reduction

Having established the architecture to use as a baseline, more experiments were performed in order to minimise the amount of trainable parameters in the network while maintaining an acceptable maximum deviation. This was achieved by modifying the amount of channels the Conv2D layers produce, by altering the minimum size the network would reduce the input to and by changing whether or not skip connections were enabled. This was also taken as an opportunity to compare

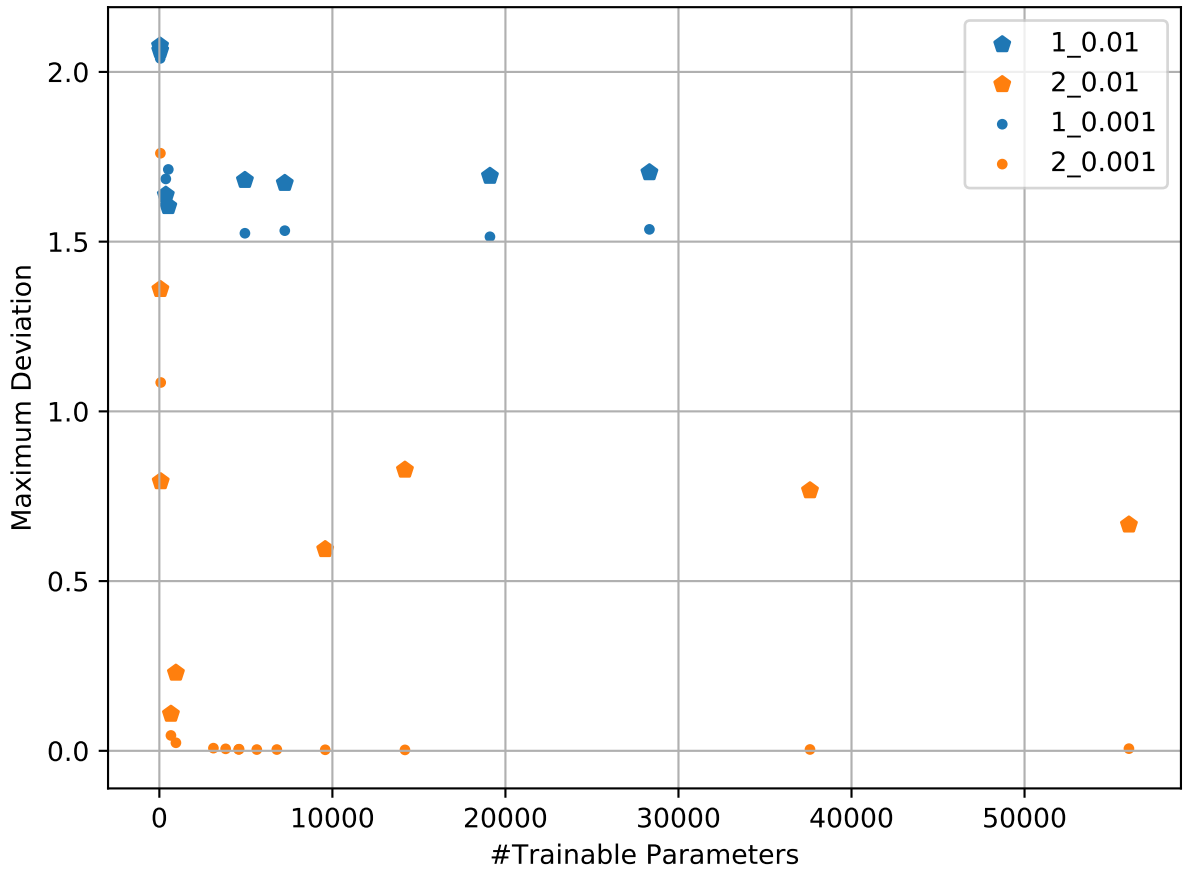


Figure 12: Results for different variations of the Grid Network
Colour and the first number in the legend denotes the amount of times the size of the input was reduced. Shape and the second number in the legend denotes the learning rate of the optimizer used during training

different learning rates, resulting in each architecture being trained with the default leaning rate of 0.001 and a modified rate of 0.01. The results of this training can be seen in Figure 12, where the numbers in the names stand for the minimum size that the input is reduced to, as measured by the amount of AvgPooling2D layers present in the network and the learning rate used by the optimiser. It should be mentioned here that removing a AvgPooling2D layer will also remove all the layers behind it without replacement until the next UpSampling2D layer is reached.

Perhaps counter-intuitively these 38 variations of the same network scatter quite far apart, with some configurations even performing worse than the Sequential Convolutional Network. This is unsurprising however, when considering that the four networks that reached a maximum deviation above 2.0 all reduced the size of the input only once and also only used one channel throughout, resulting in a total of 40 or 49 trainable parameters depending on whether or not skip connections were used. This bad performance extends to all networks that reduced the size of the input only once, which again strengthens the claim that reducing the size of the input is a good measure to increase performance.

It is also noteworthy that the majority of networks trained with learning rate 0.01 perform worse than their counterparts that were trained with a learning rate of 0.001. This may not be visible immediately but becomes apparent when considering that the learning rate does not affect the amount of trainable parameters in a network. So whenever two data points only differ in their maximum deviation it is because they are the same architecture, but trained with a different learning rate. This negative effect the learning rate might have on performance is very important to notice, since the name might lead some to believe that a higher learning rate might simply lead

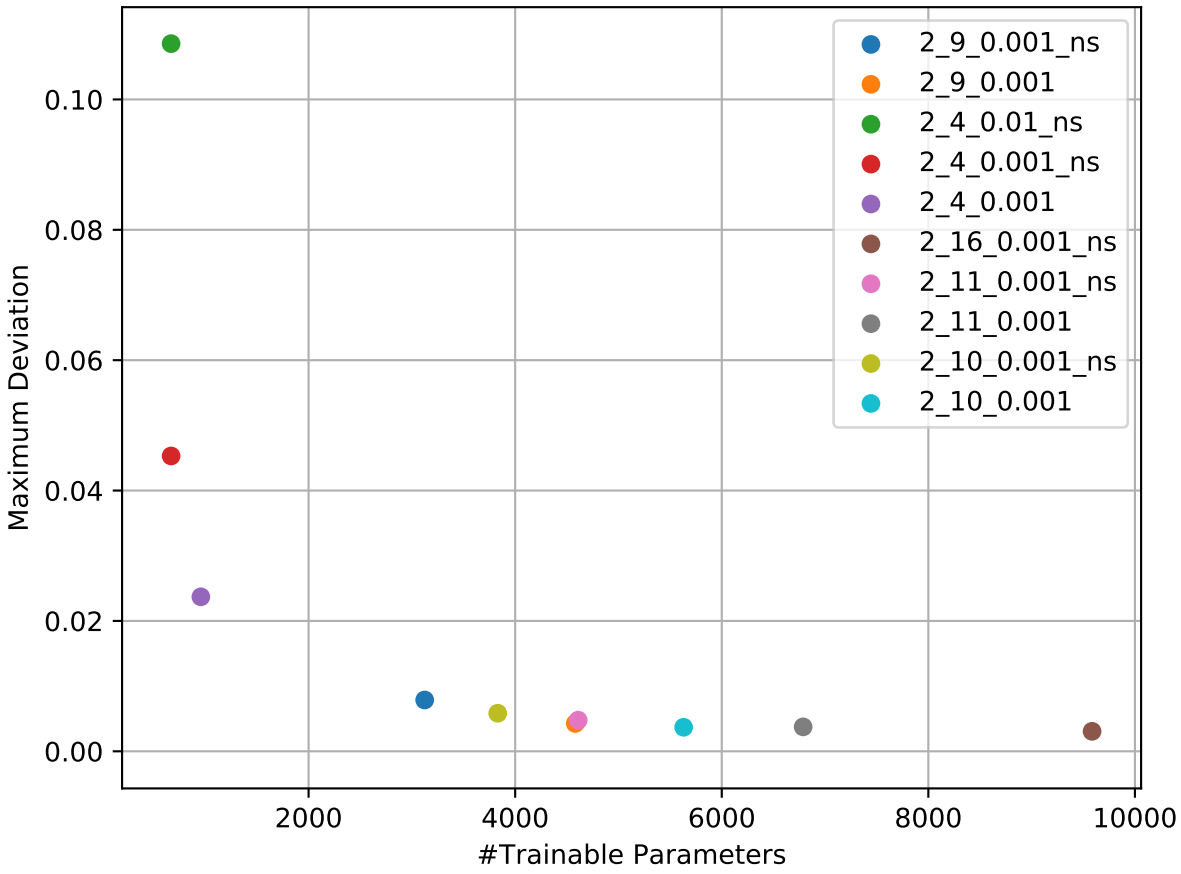


Figure 13: Part of Figure 12
The names correspond to the amount of times the input size was reduced, the amount of channels used and the learning rate in that order. The "_ns" suffix denotes configurations that did not use skip connections.

a network to learn the same things faster thus making a higher learning rate automatically better. This is not the case, as can be seen clearly from this data. A higher learning rate instead results in the network being adjusted more after each batch. Making it more accurate for that batch, but potentially causing a loss of accuracy for all other batches.

This sort of broad analysis is possible with Figure 12, but the data points are spread out too far to make out details and giving every data point a unique shape and colour would result in a visual mess. This is why Figure 13 was created, which shows only a small part of Figure 12 containing ten datapoints from the lower left corner to allow for better differentiation of the different results.

Something that could already be seen in the full image, but is even more apparent here, is the fact that a lot of the datapoints roughly fall onto a hyperbola with the coordinate axes as it's asymptotes. This is not surprising, since some sort of inverse relationship between a networks trainable parameters and it's maximum deviation makes complete logical sense. This means that there is no clear "best" configuration, allowing for different options depending on how important the amount of trainable parameters is relative to the maximum deviation. There are however a few things that can be noted about these ten results that feature a balance of the two priorities.

The following description will use the same designations as Figure 13. This means that the names for different networks will consist of three numbers, separated by _ and possibly followed by *_ns*. These numbers stand for, in order, the minimum size that the input is reduced to, as measured by the amount of AvgPooling2D layers present in the network, the amount of output channels each layer produces, again with the exception of the last Conv2D layer, and the learning rate used by the optimiser. It should be mentioned again that removing a AvgPooling2D layer will also remove

all the layers behind it without replacement until the next UpSampling2D layer is reached. The suffix $\_ns$ stands for "no skip" and is added to networks that do not use skip connections.

The only network trained with the modified learning rate of 0.01 that is part of this collection is 2_4_0.01_ns and it is also one of the three networks that is strictly worse than another network. It is strictly worse than 2_4_0.001_ns, since the latter performs better while having the same amount of trainable parameters, since that is unaffected by the learning rate. This is important because 16 of the 38 total variations were trained with the increased learning rate, but the only one that ended among the top 10 is strictly worse than its counterpart without a modified learning rate. This again proves that a higher learning rate will not cause a network to learn the same things faster, but rather cause it to learn different, and therefore possibly worse, things.

The other two networks that are strictly worse than another network are 2_11_0.001_ns and 2_11_0.001, which are worse than 2_9_0.001 and 2_10_0.001 respectively, but only by about $5{\cdot}10^{-5}$ and $1{\cdot}10^{-5}$ respectively. Especially the second case is probably an artefact of the randomness involved in training and can be safely disregarded, especially since 2_11_0.001 could be trained to produce the exact same results as 2_10_0.001 by setting all of the values related to that additional channel to zero. This makes it possible that this could also be caused by insufficient training time or an optimizer that is unable to train 2_11_0.001 to do that.

It is also noteworthy that none of the networks that only reduced the size of the input once are part of this list, despite making up 16 of the 38 samples. This would have removed two Conv2D layers from the network resulting in noticeably fewer parameters, but also resulted in significantly increased maximum deviation pushing all of them out of the area considered here.

A group of networks that were successful however, making up 6 of the 10 samples in this list while making up 50% of the variations that were considered in total, are the ones that didn't use skip connections. This is best exemplified by 2_11_0.001_ns and 2_11_0.001, where the former has 2178 fewer trainable parameters, while performing about equal with a deviation between the two of around $1 \cdot 10^{-4}$. This significant decrease in trainable parameters with comparatively little loss when it comes to the maximum deviation accounts for the disproportional representation of this group.

These skip connections are however an important part of the multigrid methods that inspired the Grid Network, which is why we will be using 2_9_0.001 with 4,582 trainable parameters and a maximum deviation of about 0.0042720, which was reduced to 0.0027423 after continued training, as the baseline for the rest of this thesis. This second maximum deviation was reached after 83,642 epochs and didn't improve for around 16,000 epochs after that, which is why that network can be considered fully trained. The values for loss and validation loss at that point were $6.6862508 \cdot 10^{-06}$ and $3.2827134 \cdot 10^{-07}$ respectively. These values refer to mse and make it safe to conclude that no overfitting has occurred, since the network is able to perform better on the validation data than on the training data. The 4,582 trainable parameters are also significantly lower than the 65,536 values the inverse of the matrix A could consist of.

7.3 Runtime Analysis

At this point it is also worthwhile to consider how many floating point operations (FLOPS) one evaluation of the fully trained network will take compared to the amount of FLOPS required by other common single step methods for solving Equation 1. For this section the linear operator A will be assumed to have shape (n, n), which results in both the right hand side f and the solution u to be scaled to size $(\sqrt{n}, \sqrt{n})$ for our network. This allows us to compare these methods for potentially very large input sizes. It should be noted here that increasing the original input size of a network is a simple change, that would nonetheless necessitate retraining and make it very likely that the network will perform worse as measured by Equation 9. The amount of FLOPS required by the network can then be calculated by considering the amount of operations each cell in the output of each layer will take, resulting in a total of $\frac{33803}{8} * n \approx 4225.38 * n$ operations for 2_9_0.001.

If f and u are considered as vectors again and the time spent to train the network was instead used to calculate the inverse of the linear operator A^{-1} the amount of FLOPS to calculate u from $A^{-1}f = u$ would be $2n^2 - n$. n^2 multiplications and $n^2 - n$ additions. That same amount of FLOPS is also required when solving that system with a given LU-Decomposition of A if no permutation matrix P is required, even if the types of operations differ.

input	samples	random	images total
maximum deviation	0.0019669	28.5216835	23.3618183
maximum value	35.60313	<1.0	0.5
minimum value	0.6256021	0.0	0.0

Figure 14: Maximum deviation and maximum and minimum value in any of the correct solutions for all of the respective baselines

input image	X	Checkerboard	List	Ring	Smiley
maximum deviation	6.1115670	23.3618183	16.0333786	3.9011507	6.6746039

Figure 15: Maximum deviation for specific images as baseline

Comparing these two results allows us to conclude that our network is faster than those two methods for $n > \frac{33811}{16} = 2113.1875$, though it should again be stated that the other two methods provide exact results, while our network will likely only ever be able to calculate approximations.

7.4 Further Analysis

Another thing that is interesting when considering neural networks is how broadly they work. Wanting the network to be able to solve only the problems used during training and validation is pointless, since the solutions to those problems are already known. It is very likely that the network will be able to perform well on some problem instances it was never specifically trained for, since the networks performance on the problem instances used during validation is still good. This leaves it unclear how similar to the training data a problem instance has to be to allow our network to solve it with a good degree of accuracy.

Furthermore it can be interesting to examine the results of the network on a visual level, since this might allow us to understand what the parameters the network has learned ultimately do to its input.

For this purpose and to check how broadly the network can work Figure 16, Figure 17 and Figure 18 were created using a baseline of samples of the test data, random data and predefined images respectively. In the first step, this baseline, depicted in the first column of the figures, was given to the network "as right hand side f", so to speak, to create a visual representation of what the network does to its input, depicted in the second column. No deviation from the correct solution was calculated for this, since the u that would fulfil Equation 1 is unknown for those fs for the random data and the images. As a second step, the baseline was used "as solution u", which allows for the calculation of the corresponding f using Equation 1 and the stencil seen in Figure 1. Said corresponding f was then given to the network to calculate $u_{network}$, depicted in the third column, which allows us to calculate the deviation of the networks solution from the correct solution and depict how big the difference between the two solutions is in each point in the last column of the figures. The specific network used here is still the fully trained version of 2_9_0.001 described previously.

Figure 16 as expected shows little difference between expected and produced output. It is also good to see that the positions the maximum deviation is located seem to be more or less random. That not being the case would imply some sort of systematic error that the network should have learned to avoid. There is also nothing obvious to be noticed in the random images of Figure 17.

Interestingly enough however, Figure 18 shows clear patterns in both the networks output and the deviation between said output and the corresponding correct solution. An especially interesting result is generated by the checkerboard image, which causes the network to generate a checkerboard pattern that, as can be seen in the last image, is offset by one cell. The corresponding maximum deviations can also be seen in Figure 15, which also shows a large deviation from the correct solution in the checkerboard image.

It is very apparent from the difference in deviations seen in Figure 14, which shows the maximum of all deviations and the maximum of the values in the correct solutions for the different baseline groups, that the network does not know how to handle random data and images nearly as well as the test data. This is probably mostly caused by commonalities present in the training data that

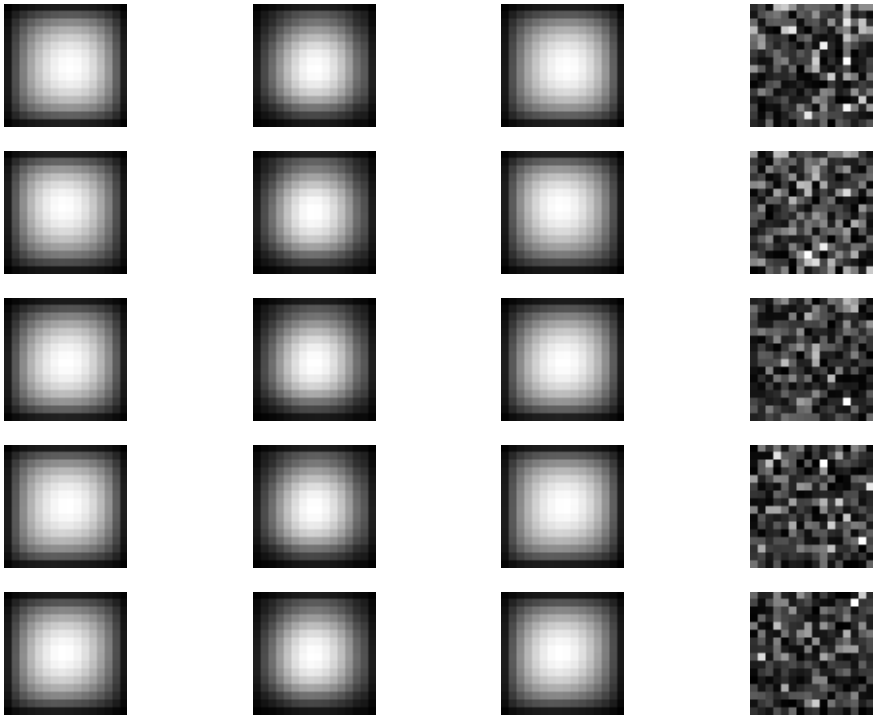


Figure 16: Depiction of the results of our final network on samples of the test data
The columns in order correspond to: Original input, result of original input, result of original input after application of A, absolute value of the difference between the first and the previous column
Note that images are not necessarily scaled to the same values

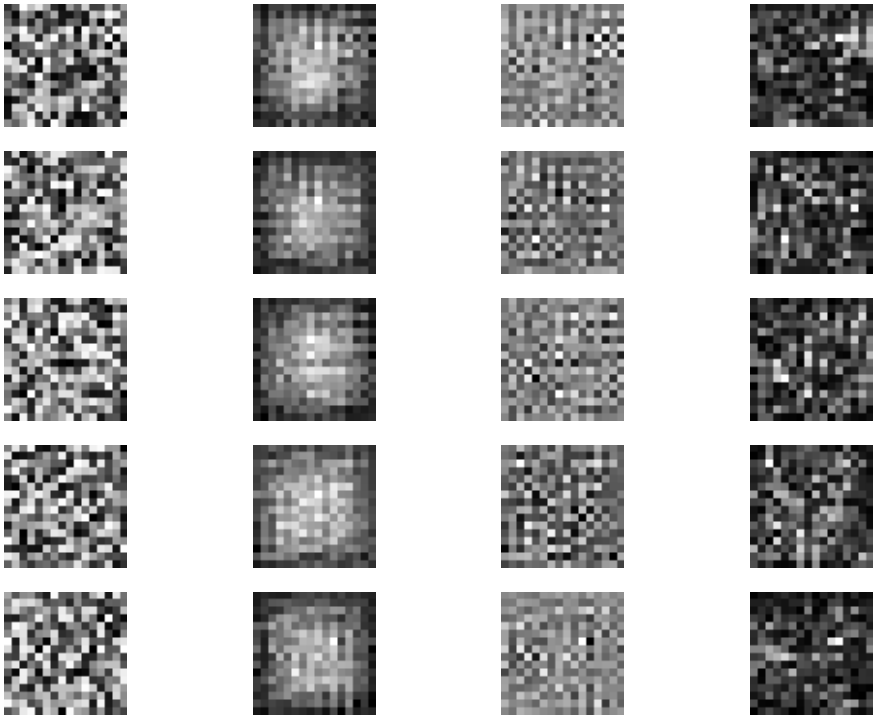


Figure 17: Depiction of the results of our final network on random inputs
The columns in order correspond to: Original input, result of original input, result of original input after application of A, absolute value of the difference between the first and the previous column
Note that images are not necessarily scaled to the same values

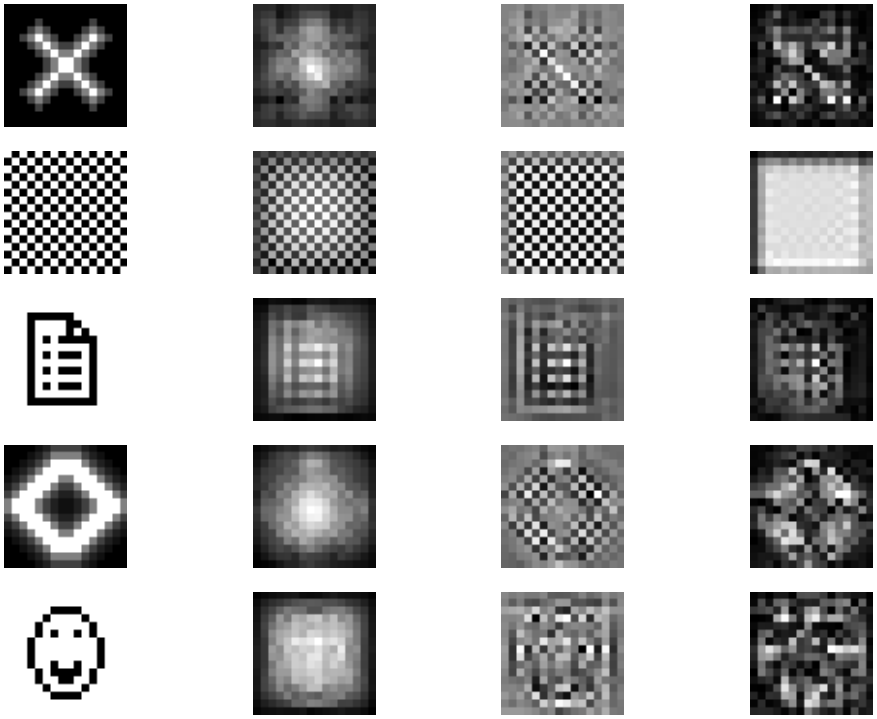


Figure 18: Depiction of the results of our final network when using hand crafted images as inputs The columns in order correspond to: Original input, result of original input, result of original input after application of A, absolute value of the difference between the first and the previous column Note that images are not necessarily scaled to the same values

are not present in the random data and the images. The most obvious commonality and the one that is most likely to be responsible for this difference is the fact that the samples transition rather smoothly from high values to low values, which is not necessarily the case for the random data and the images. This is all but confirmed by Figure 15, which shows the maximum deviations for specific images, with "Ring", which has the smoothest transition of all the images, having the lowest maximum deviation, while "Checkerboard" has the highest deviation.

This means that our network was unable to learn to function as an inverse of the matrix A that is generally applicable, but is able to function as a decently accurate approximation of A^{-1} for problems that are sufficiently similar to the ones it was trained with.

8 Conclusion

This thesis proposed several different architectures for convolutional neural networks which could be used to function as an approximation of the matrix A^{-1}_{approx} in Equation 2. We aimed to minimise Equation 9 and the amount of trainable parameters of the network. Through experimentation it was discovered that training our network to minimise Equation 9 directly was flawed, which allowed us to achieve significantly better performance by changing the goal of the training to minimise mse (see Equation 15) instead. Our experiments clearly pointed to the Grid Network being the best of the considered architectures for this task, which was inspired by the multigrid methods, a set of methods for solving partial differential equations. In an attempt to reduce the amount of trainable parameters, several different variations of the Grid Network were created and trained in the same way. This resulted in a network able to achieve a maximum deviation on our test data of 0.0027423, while only using 4,582 parameters, which fulfilled our goal of keeping the amount of trainable parameters below the 65,536 values the exact inverse A^{-1} could potentially consist of. To find out how broadly this network could be applied we then generated new inputs from random noise and images. Applying our network to these inputs resulted in Figure 17 and Figure 18 and a maximum deviation of 28.5216835, proving that our network was incapable of functioning as a reasonably accurate approximation of A^{-1} for inputs that deviated too much from the data used in training.

9 Future Work

Something that would be interesting to find out in regards to the final network this thesis resulted in is where the borders of similarity in regards to the input lie. This would require several problems with known solutions and decreasing similarity to the data used while training the network, but would generate an understanding of what types of problems this network can and can't solve. Furthermore it would be reasonable to try out even more different network architectures as there is no guarantee that the Grid Network is the absolute best baseline architecture for this task. When considering the significant increase in performance the Advanced Convolutional Network achieved when transitioning from $loss\_max$ to mse it may also be worthwhile to consider different loss functions. Activation functions were also ignored by this thesis, because we considered a linear problem, which should be possible to solve by a network that only uses linear elements. Studying the effects of different activation functions might also be worth it's time, should this not be a consideration anymore. If an abundance of time is available all of the modifications to networks and their training mentioned above could be performed on a CPU and with significantly longer training times. This would make it possible to remove all of the randomness produced by cuDNN and make sure that all networks were trained until they have reached their peak performance, which would allow for a clear and reproducible result.

Another worthwhile consideration would be if approximating the inverse A^{-1} through a convolutional neural network is even the correct approach. In a way similar to the weighted Jacobi method it may be possible to parameterise already existing methods for solving PDEs and train a neural network to supply those parameters in order to improve performance. Somewhat similarly to that approach, attempting to train a CNN to imitate single steps or parts of an already existing algorithm might also yield better results than attempting to train a network to reach the correct solution in just one step.

References

[BM+00] William L Briggs, Steve F McCormick, et al. *A multigrid tutorial*, volume 72. Siam, 2000.

[DV16] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, 2016.

[FGP17] Amir Barati Farimani, Joseph Gomes, and Vijay S. Pande. Deep learning the physics of transport phenomena. *CoRR*, abs/1709.02432, 2017.

[GDDM13] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[HZE+19] Jun-Ting Hsieh, Shengjia Zhao, Stephan Eismann, Lucia Mirabella, and Stefano Ermon. Learning neural PDE solvers with convergence guarantees. *CoRR*, abs/1906.01200, 2019.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[LAB+14] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, et al. Exastencils: Advanced stencil-code engineering. In *European Conference on Parallel Processing*, pages 553–564. Springer, 2014.

[Lan] Thom Lane. Transposed convolutions explained with... ms excel! - apachemxnet - medium. https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8. Accessed: 2019-11-17.

[LeV07] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial and Applied Mathematics, 2007.

[LMR+17] John Lawrence, Jonas Malmsten, Andrey Rybka, Daniel A Sabol, and Ken Triplin. Comparing tensorflow deep learning performance using cpus, gpus, local pcs and cloud. 2017.

[NVI] NVIDIA. cudnn developer guide :: Nvidia deep learning sdk documentation. https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html. Accessed: 2019-11-27.

[ODO16] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts. *Distill*, 2016.

[OLT+19] Ali Girayhan Özbay, Sylvain Laizet, Panagiotis Tzirakis, Georgios Rizos, and Björn Schuller. Poisson cnn: Convolutional neural networks for the solution of the poisson equation with varying meshes and dirichlet boundary conditions, 2019.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017.

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[SDBR14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net, 2014.

[SFG+18] Rishi Sharma, Amir Barati Farimani, Joe Gomes, Peter Eastman, and Vijay Pande. Weakly-supervised deep learning of heat transport via physics informed loss, 2018.

[SHM+16] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[STD+17] Tao Shan, Wei Tang, Xunwang Dang, Maokun Li, Fan Yang, Shenheng Xu, and Ji Wu. Study on a poisson's equation solver based on deep learning technique, 2017.

[Teaa] The Keras Team. Home - keras documentation. https://keras.io. Accessed: 2019-11-21.

[Teab] The TensorFlow Team. Tensorflow. https://www.tensorflow.org. Accessed: 2019-11-28.

[TOS00] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. *Multigrid*. Elsevier, 2000.