MSc Thesis

The Parameter Efficiency of Neural Ordinary Differential Equations

Investigation of a Novel Neural Network Framework

Academic Supervisor:

J. Hulshof

External supervisors:

V. Muhonen

D. Timmers

R. Price

Author:

L.F.D. Bijsterbosch

Date:

March 2020

The Parameter Efficiency of Neural Ordinary Differential Equations

Investigation of a Novel Neural Network framework

Lotte Floor Dirkje Bijsterbosch

Master Thesis

March 2020

Vrije Universiteit Amsterdam
Faculty of Science
Business Analytics
De Boelelaan 1081a
1081 HV Amsterdam

Mobiquity Inc.
Data Analytics Team
Tommaso Albinonistraat 9
1083 HM Amsterdam

Preface

Reading the award winning paper by Chen *et al.* [6] really sparked my interest in the subject of neural differential equations, and has been the inspiration for the topic of this project. It has shown to be a great thesis subject. It has given me the opportunity to combine both theoretical research and practical models, and the chance to really dive deep into the new technique. Altogether it has been very rewarding project and challenging at times.

This research project is conducted as part of the MSc Business Analytics. It consists of a half year of full-time research combined with an internship. The project has been carried out at *Mobiquity*, and my colleagues there have have been of great help during the project. Not only did they give me all the freedom to shape the project to my interest, they also have provided me with all the necessary equipment, from access to cloud computing services to proper hardware. Even more importantly, a great team of data scientists who are always open for discussions.

At last I would like to thank the supervisors who have been involved during the project. Joost for many interesting discussions, his support with regards to the mathematics and his eagerness to learn about neural networks. And the supervisors from *Mobiquity*; Dennis, Vesa and Richard. They have provided great feedback and support, and their enthusiasm for the project has been a great motivator.

Lotte Bijsterbosch,
March 2020, Amsterdam

Summary

Recently a new framework, the ODEnet, a neural network that incorporates an ODEsolver has been introduced by Chen *et al.* [6]. The new technique raises many questions, such as the advantages over existing techniques, and how to implement the model in practice. This research project will address a couple of aspect of ODEnets, both by diving into the theory and by experimental research. The objective is to review the claim by Chen *et al.* that the parameters of an ODEnet are more efficient, indicating that the ODEnet with less parameters has the same predictive performance as a ResNet. This claim is reviewed specifically for image classification. To answer this question we also need to get more insight into the workings of the model, and how to construct a succesful ODEnet architecture.

First, the theory of ODEnets is investigated. We derive the continuous backpropagation technique proposed by Chen *et al.* with an alternative method using Lagrange optimization. This creates a clear connection to traditional backpropagation methods. Investigation into forward propagation leads to insights in the expected advantages of ODEnets, and why the more efficient parameters are not found in experimental results. Additionally, it provides areas for future research, such as incorporating a time dependency in the weight parameters.

The experimental section of the research consists of two parts. Due to the novelty of ODEnets, there is very little known about how such a network can be constructed successfully. This project addresses this problem, and looks into different types of layers and activation functions and their effect on the training time and whether they are likely to cause errors. This information is put into practice while designing models to research whether the ODEnet has more efficient parameters. Comparing ODEnets with an equivalent residual network (ResNet) has shows that both have a very similar predictive performance. This indicates that the parameters of an ODEnet are not more efficient. Additionally, the ODEnet takes much longer to train than a ResNet. Therefore, the ODEnet should not be used to replace a ResNet. It does, however, have more potential in time-dependent applications, such as time-series.

List of Abbreviations

ODE	Ordinary differential equation
IVP	Initial value problem
NFE	Number of function evaluations
FNFE	Number of function evaluations of the forward pass
BNFE	Number of function evaluations of backpropagation
ResNet	Residual Network
CNN	Convolutional neural network
HPO	Hyperparameter optimization
groupnorm	Group normalization
batchnorm	Batch normalization
autodiff	Automatic differentiation

Contents

CHAPTER 1

Introduction

The field of machine learning is booming. Every year there are new models and improvements of already existing techniques. From neural networks that have more than 97% accuracy in face detection tasks [48, 50] to computer programs beating humans in the game of Go [42]. Another new type of neural network can be added to this list of notable developments. The ODEnet is an innovative new idea, inspired by the resemblance between residual networks (ResNet), a type of neural network, and the Euler method, a way to solve ordinary differential equations (ODE). This similarity has been noted for a while, but in 2018 Chen *et al.* [6] have designed the first working model.

There are a lot of questions that remain unanswered now that the ODEnet can be used for practical problems. In this research we will focus on the question whether the parameters of an ODEnet are more efficient than that of a ResNet for image classification. In other words, does an ODEnet with less parameters have the same predictive performance as a ResNet. Additionally, the model itself is studied. What happens during forward and backpropagation when a black-box ODEsolver is incorporated into the network? Another important aspect that is addressed is how to construct a successful ODEnet.

The great resemblance between ResNet and the Euler method has been observed by Weinan [51] and others in the past years [4, 15, 35]. At first this lead to analyzing ResNets with the knowledge from the field of ODEs. Understanding of what leads to stable ODEs, and how to properly solve them can be applied to ResNets. This helps in creating stable models that train well. The Euler method is not the only discretization scheme that can be recognized in a neural network. Other methods, such as Runge-Kutta or the backward Euler method have also been observed in existing neural network architectures. This has lead to the development of neural networks with structures based on ODE methods. This idea is further extended by Chen *et al.*, by creating a ResNet that incorporates a black-box ODEsolver. When using the Euler method as an ODEsolver, this network is equivalent to a plain ResNet. But, with the ODEsolver incorporated in the architecture it is possible to use different, better methods. The adaptive step size solvers actually lead to a neural network with a continuous hidden dynamics, a big change from a regular neural network. This dynamic depth brings many different advantages with it. First, the number of steps that are computed within the ODEsolver, equivalent to the network depth, are based on an error tolerance. The ODEsolver will take extra steps if necessary to stay bellow the threshold. This creates a trade-off between computational cost of the network, and the desired error tolerance. Second, the hidden state dynamics are treated as a continuous function. This could entail that the parameters of the different layers that are passed to the ODEsolver are more 'together', and work more efficiently. Additionally, the continuous nature of the internal dynamics is a good fit for time-dependent problems. Time dependent variables do not have to be forced into discrete time brackets, as the continuous nature of the ODEnet is a much better fit.

The main focus of this project is whether ODEnets are more efficient than a ResNet, meaning that the same accuracy can be achieved with less parameters. Chen *et al.* [6] showed that ODEnets can be used as a drop-in replacement for ResNets. With the claimed advantages of dynamic depth and more efficient parameters, the ODEnet is expected to be an improvement. However, these claims are not properly supported yet. The statements are mostly based on the theoretical knowledge about ODEnets, but due to the novelty of the network have not been observed in practice. In order to fill this gap, we will create an ODEnet to perform image classification. This ODEnet is compared to a ResNet with a similar structure, so the difference of adding an ODEsolver can be observed.

A second point of interest is how forward and backpropagation is influenced by incorporating a black-box ODEsolver inside a neural network. We will investigate what computations take place inside this solver, and how it compares to the usual computations of the forward pass. In order to correctly analyze the network, it is helpful to dive into the mathematical formulation. This also provides a clear overview of why certain advantages and disadvantages are expected.

Chen *et al.* [6] derive an alternative, continuous form of backpropagation for ODEnets. The main advantage of this technique is that it has constant memory cost, so the amount of computer memory required is not dependent on the number of computations. In order to clarify what this new method entails we derive it via an alternative proof using Lagrange optimization. We start from the regular discrete backpropagation and connect this to the continuous alternative approach proposed by Chen *et al.*

Lastly, it is important to know what limitations incorporating an ODEsolver puts on the architecture of an ODEnet. Theoretically it is possible to incorporate all different types of layers and activation functions, but does it work in practice? This is currently a big gap in the knowledge about ODEnets. A few models have been designed, but little is known as to what works and what does not. This makes the creation of an ODEnet relying heavily on trial and error, and fixing things when you face problems. We will create multiple ODEnet architectures, and investigate the influence on computational cost, accuracy and potential errors. These results lead to a couple of recommendations for designing ODEnets that prevent encountering problems during training.

1.1 Structure of Thesis

The structure of the report can be divided in different sections. The first three chapters cover the background theory. Starting with Chapter 2 about neural networks, which will lay the foundation for the different model types that will be used throughout the project. This is extended in Chapter 3 to an ODEnet. The basics about ODEs are covered, and how to construct an ODEnet. It will further elaborate on the forward pass through an ODEsolver and the whole ODEnet. Then the possible advantages and applications will be discussed. The next chapter is devoted to backpropagation. Due to the continuous nature of the ODEnet there are multiple possible techniques for calculating the gradient, starting from normal automatic differentiation to rewriting the problem completely in terms of ODEs. These methods will be discussed, as well as which technique is best in the case of ODEnets.

When all the background theory has been covered it is time to go into the experiments. First the data is discussed, followed by the research method. This chapter includes the procedure and the limitations of the research. Then Chapter 7 explores the design of an ODEnet, and what architectures are possible. This is concluded with the models used in the experiments.

The results are reported in Chapter 8 and interpreted. This leads to the discussion in Chapter 9, and a final chapter with the conclusion. The topics that will be covered include an analysis of the found results and a future direction for research.

CHAPTER 2

Neural Networks

This chapter will focus on three types of neural networks, starting with a traditional model and expanding this to convolutional and residual networks. The mathematical framework and notation of neural networks is introduced, which will form the groundwork for creating an ODEnet in the Chapter 3. A more detailed description of the different types of neural networks and layers can be found in in the *Deep Learning* textbook by Goodfellow [12].

2.1 Architecture of a Classical Neural Network

A neural network is a model $f(x, \theta)$ that maps an input x to an output or category y. The goal is to find the optimal weight parameters θ such that the mapping function comes as close to the actual value of y as possible. The structure of the model consists of multiple layers f_ℓ that map $z^{(\ell-1)} \to z^{(\ell)}$. An example of a network with fully connected layers is in Figure 2.1. This example has an input layer, multiple hidden layers ℓ with n_ℓ hidden nodes and one output layer $z^{(N)}$. There is a weight for every connection between two nodes, resulting in a weight matrix θ_ℓ for every layer.

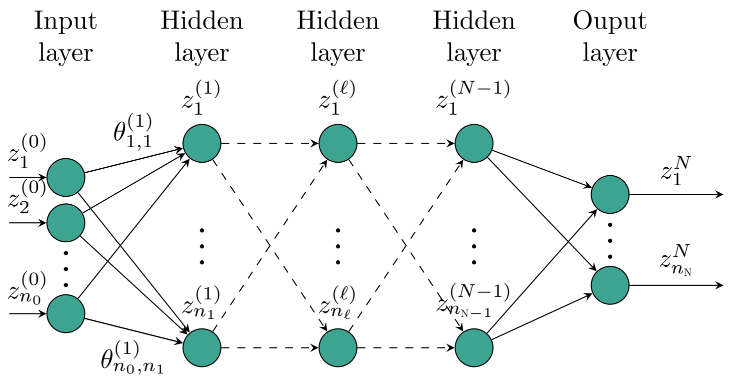


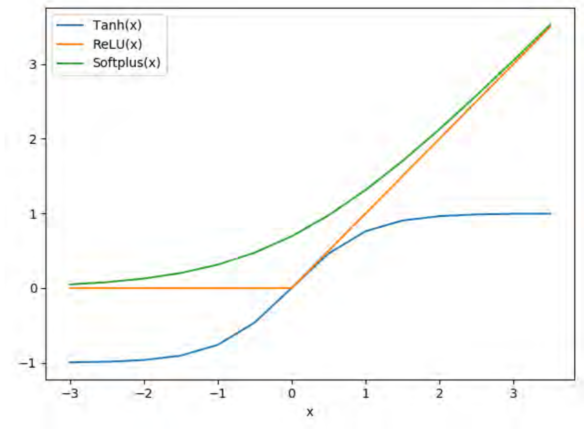
Figure 2.1: Neural network architecture with multiple hidden layers

The number of hidden layers and amount of nodes can vary, and define the neural network architecture of $f(x, \theta)$. In order to calculate the value of the hidden states $z^{(\ell)}$ of fully connected layer ℓ, one performs matrix multiplication of the input $z^{(\ell-1)}$ by the corresponding weights $\theta^{(\ell)}$ ($g_\ell = \theta_\ell^\top z^{(\ell-1)}$). This is followed by a non-linear activation function $h(x) = \sigma(x)$. Every layer consists of such a matrix multiplication and non-linearity, until one reaches the output layer. The mapping of a single layer ℓ can be described by the general function $f_\ell(z^{(\ell-1)}, \theta_\ell) = h(g_\ell) = \sigma(\theta_\ell^\top z^{(\ell-1)})$. A whole neural network model is described by multiple of these layers. The complete mathematical formulation of mapping input $x = z^{(0)}$ to the output $y = z^N$ though the network is described in Equation 2.1.

$$\begin{aligned}
z^{(0)} &= x \\
z^{(1)} &= f_1(z^{(0)}, \theta_1) \\
&\vdots \\
z^{(\ell)} &= f_\ell(z^{(\ell-1)}, \theta_\ell) \\
&\vdots \\
y = z^{(N)} &= f_N(z^{(N-1)}, \theta_N)
\end{aligned} \tag{2.1}$$

2.1.1 Activation Functions

Activation functions are used to introduce non-linearity in the model. This is an important part of the neural network, as it is a requirement for the universal approximation theorem to hold [20, 21, 32]. In a classic neural network there is an activation function $\sigma(x)$ after the weight multiplication. In this paper three different functions are used: tanh, ReLU and Softplus (Equation (2.2)). All these functions can be used in between hidden layers, but the use of ReLU is generally recommended [10, 12]. While this function is not differentiable at 0, empirical research shows this does not cause problems in practice [10].



$$\begin{aligned}
\text{Tanh}(x) &= \tanh(x) \\
\text{ReLU}(x) &= \max(0, x) \\
\text{Softplus}(x) &= \ln(1 + e^x)
\end{aligned} \tag{2.2}$$

Figure 2.2: Three activation functions.

2.2 Convolutional Network

A convolutional neural network (CNN) is a type of neural network that is often used for image classification. The CNN can take multidimensional input, which is ideal when processing an image. An image has a height and a width dimension, and a third dimension with every pixel described in colour, most commonly in red, green and blue (RGB). The theory of the CNN has been around since the 90's [31], but has gained most of its popularity after the introduction of AlexNet [28], a very successful implementation of a CNN. AlexNet had such a big increase in performance compared to other models that it sparked the field of image recognition to mainly develop CNNs for the task. Ever since, models such as GoogleNet [47] and VGG [43] have built on the same ideas. The main difference between a CNN and a classic neural network is that while the neural network only has fully connected layers, the CNN has many different layer types. Usually the CNN has multiple repetitions of a block consisting of a convolutional layer, pooling, activation function and normalization. The following paragraphs will provide a description of these layers.

2.2.1 Convolutional Layer

The main feature of the CNN is the convolutional layer. The input to a convolutional layer usually has several channels C, such as the three colours of an image. Then a 3x3xC feature map, often referred to as a filter, is passed over the input channels. The filter starts in the topleft corner, and calculates a new output value by multiplying the input and the filter, and adding everything together. Next, the filter moves one step to the right and again calculates the output value. The output channel is filled by passing the filter over the input image. The stride is the number of rows/columns the filter moves per step, which can be increased in order to reduce the output dimension. An example of such a layer is in Figure 2.3.

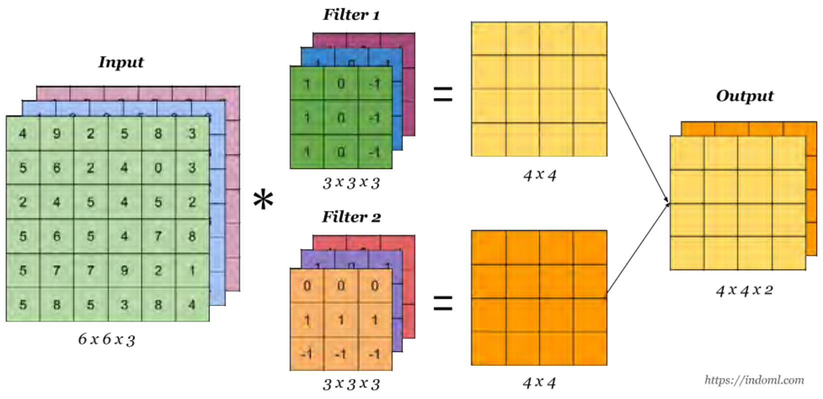


Figure 2.3: Example of a Convolutional layer. There are two filters of 3x3x3 and a stride of 1. This results in an output layer with two channels of 4x4 [22].

Every convolutional layer in the CNN has multiple filters, with each filter mapping a certain feature to its own output channel. Most commonly each of these output channels has information on a specific feature, and whether it is present in any location of the input. An example of this is edge detection. A filter is passed over the image to find whether there is a horizontal edge in the picture. This information is then passed on to another convolutional layer, where different feature maps gather different information. All these layers together ensure that in the last output layers of the model a final prediction can be made.

2.2.2 Pooling Layer

Pooling layers are used as a way to reduce the computational cost of the CNN, without adding extra parameters. By reducing the dimensions of the data, the next layer does not require as much parameters and computation. Another important aspect is that pooling makes the model more invariant to small changes in the input [12]. For example, one can see from Figure 2.4 that shifting the location of all the input values up by one only changes half of the output points. A small location change in the input data does not have a big effect on the output of the pooling layer.

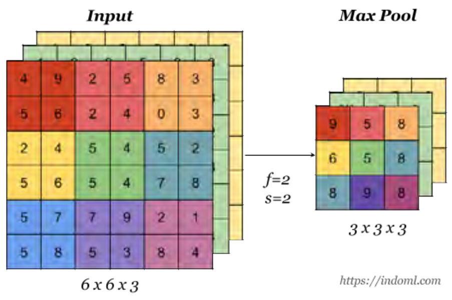


Figure 2.4: Example of max pooling. The result is a maximum value of the input group, where every colour in the output corresponds to the colour in the input [22].

There are a few types of pooling, such as maximum or average (Figure 2.4), in which a group of numbers is reduced by their maximum or average. This ensures that most information is retained, while greatly decreasing the dimensions. The most commonly used type of pooling is the max pooling.

2.2.3 Normalization

A difficulty in finding the optimal parameters is that the parameters of previous layers also change during training. Adding a normalization layer eases the training of a model [23, 52]. The input of a convolutional layer is first normalized, similar as during the preprocessing of the data. This reduces the dependency of parameters on the previous layers.

There are several ways to do normalization, such as batch or group normalization. These terms are abbreviated to batchnorm and groupnorm. In batchnorm the normalization is applied to the whole batch, meaning the whole batch is used to calculate the mean and standard deviation. The batch is then transformed to have zero mean and unit variance. Groupnorm is a little more precise, as it applies normalization to only a couple of input channels, a group. Every group of input channels has its own calculated mean and standard deviation.

2.2.4 Fully Connected Layer

The final layers of the model are one or more fully connected layers. The 3D output from the convolutional layers is flattened into a vector, so it can be fed to the fully connected layer. The final layer of the model will make the prediction, and has the dimension of the output classes.

2.3 Residual Network

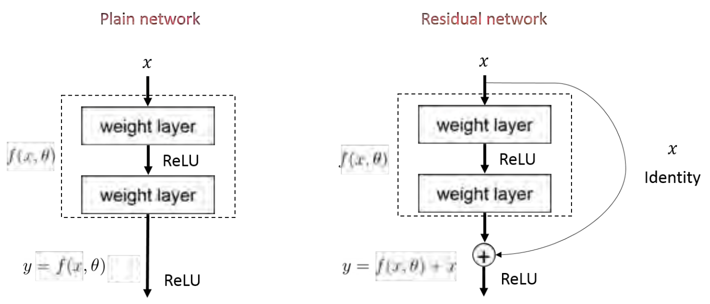
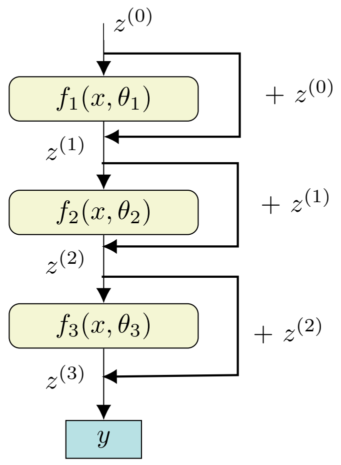


Figure 2.5: A normal and a residual connection [17] [33].

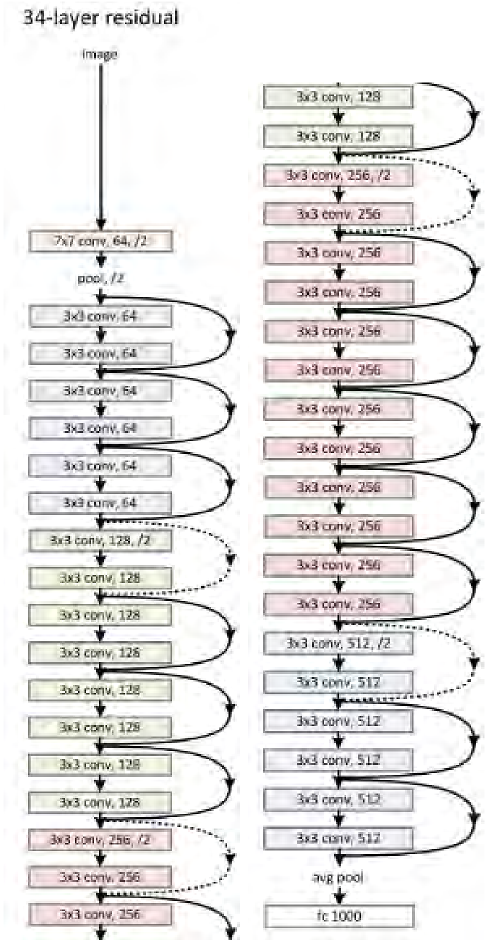
Residual networks (ResNet) are an important extension of CNN models. The most important part of a ResNet is the residual, or the *skip connection*. This connection can be added to most neural network structures, and is often used in combination with a CNN. Traditionally, a neural network tries to find the correct mapping $f(x,\theta)$ to output y by passing input x through several layers. The residual connection is slightly different. After passing the input x through a layer, the original value of x is added to the output of the layer (Figure 2.5). The output of the layer is now given by $y = f(x,\theta) + x$. Here, the function $f(x,\theta)$ only estimates the *change* from x to y. Work by He *et al.* [17] shows that this prevents the drop in accuracy that usually comes with very deep models. Therefore, the ResNet can have many extra hidden layers without loss of performance. The big advantage of this method is that it adds very little computational costs, while removing restrictions on the depth of a neural network.

The mathematical formulation of a ResNet is very similar to that of a regular neural network. The biggest difference is that at every layer, the input x is also added. Figure 2.6 shows a small example with three residual blocks. The forward pass through such a network is given by Equations (2.3). In practice ResNets are often combined with a CNN, such as in Figure 2.7, and can have many layers. The depth of a ResNet can go up to a 100 or even 1000 layers.



$$\begin{aligned} z^{(1)} &= x + f_1(z^{(0)}, \theta_1) \\ z^{(2)} &= z^{(1)} + f_2(z^{(1)}, \theta_2) \\ z^{(3)} &= z^{(2)} + f_3(z^{(2)}, \theta_3) \\ y &= z^{(3)} \end{aligned} \tag{2.3}$$

Figure 2.6: ResNet with three residual blocks.



(a) top half (b) bottom half

Figure 2.7: A full residual network [17].

CHAPTER 3

The ODEnet

In this chapter the theory of ODEnets will be introduced. First, ordinary differential equations (ODE) and how to solve related problems will be investigated in Section 3.1. This will follow the book by D. Zill [58], to which the reader can refer if they wish to read more about ODEs. The main focus is the Euler method, which is at the base of the idea to create an ODEnet. The connection between the Euler method and ResNets has been noted by several works in the literature. Section 3.2.1 dives into previous research, and reviews the literature on ODEnets.

Next is the structure of an ODEnet. The main difference with a regular neural network is that a black-box ODEsolver is incorporated into the network. This is a big change, and also alters the forward pass of data through the network. The exact differences, and what happens within the ODEsolver, will become clear in Section 3.2.2.

After the introduction of the ODEnet the details of the network will be explored. This includes the applications and expected advantages, a small extension so the network becomes a universal approximator and how to design the architecture of an ODEnet is covered. This last topic will be covered more in depth in Chapter 7 where we experiment with different ODEnet architectures.

3.1 Ordinary Differential Equations

An ordinary differential equation is an equation that involves an (unknown) function y and its derivatives [58]. A common question in the field of ODEs is the initial value problem (IVP). Given the derivative and an initial point y_0 (Equation (3.1)), can a solution $y(t)$ be found? Sometimes this question can be answered analytically and an exact solution can be found, but more often then not numerical methods are required. Via numerical methods, often referred to as *solvers*, a solution $y(t)$ can be estimated by extrapolating with small steps from initial point y_0.

$$\begin{aligned} y(t_0) &= y_0 \\ \frac{dy}{dt} &= f(t, y(t)) \end{aligned} \tag{3.1}$$

3.1.1 Solving Initial Value Problems

The most straightforward way to solve an IVP numerically is with the Euler's method. The solution is approximated by taking discrete steps starting from the initial value, calculating the next point as in Equation (3.2). This process is repeated to extrapolate the whole solution.

$$y_{n+1} = y_n + hf(y_n) \tag{3.2}$$

Take, for example, the following IVP problem in Equation (3.3). Take stepsize $h = 0.1$, then the first point is calculated by $y_1 = y_0 + hf(y_0) = 1 + 0.1 \cdot 1 = 1.1$. Further values can be found in Table 3.1.

$$\begin{aligned} y_0 &= 1 \\ \frac{dy}{dt} &= y \end{aligned} \tag{3.3}$$

y$_{step}$	t	Euler result	Analytical result
y_0	0	1.0	1.0
y_1	$y(0.1)$	1.1	1.105
y_2	$y(0.2)$	1.21	1.2214
y_3	$y(0.3)$	1.331	1.3498
$\vdots$	$\vdots$		
y_{40}	$y(4)$	45.259	54.6

Table 3.1: Euler method with stepsize h=0.1.

The function used in this example is the exponential growth function, with $y = e^t$ as the analytical solution. Figure 3.1 shows how the error of the numerical solution accumulates as the distance from y_0 increases.

Euler's method is a very simple method to solve IVP problems. There are many more sophisticated ways to solve an IVP. An intuitive one is to not update every step with the derivative from the current point y_n, but from the average of y_n and y_{n+1}. This is called the *midpoint* method. The family of *Runge-Kutta* methods extend this idea even further. In these methods the derivative is calculated using the weighted average of k points during one timestep. Higher order Runge-Kutta methods are

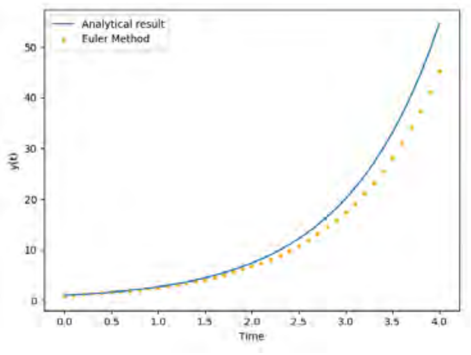


Figure 3.1: Graph of numerical and the analytical solution.

more precise by using a higher number of points, and therefore have a smaller error.

The techniques mentioned in the last paragraph all have a fixed stepsize, which is set beforehand. An alternative to this are adaptive stepsize methods, such as *Dopri5*. A certain error tolerance is set, and whenever the estimate of the error of a timestep is too high, the step is repeated with a smaller stepsize. Additionally, if the error is much smaller than the tolerance the step size is increased to save computation time. The idea is that this solver will be precise with small stepsizes on difficult parts of the function, and whenever the function is less complex it will use larger steps. This ensures the solver will remain within the given error tolerance, without making unnecessary calculations.

3.2 The ODEnet

3.2.1 Literature Review ODEnet

The mathematical formulation of a ResNet and the forward Euler method for solving ODEs are closely connected to one another. The function for a hidden layer in a ResNet is $z^{(\ell+1)} = z^{\ell} + f(z^{\ell}, \theta)$, which bears great resemblance with the Euler method from the previous section (Equation (3.4)). This leads to the intuition that multiple residual blocks can be seen as multiple updates with the Euler method [4]. This is illustrated by Figure 3.2, where every timestep is one residual block.

$$\begin{array}{ll} \text{ResNet layer} & \text{Euler Method} \\ z^{(\ell+1)} = z^{\ell} + f(z^{\ell}, \theta) & y(t+1) = y_t + f(y_t) \end{array} \tag{3.4}$$

There are several neural network architectures that can be described in terms of ODEs. Lu *et al.* [35] show how different existing architectures, such as FractalNet [29], RevNet [11] and PolyNet [57] are consistent with different discretization methods for ODEs. They correspond, respectively, to a Runge-Kutta method, a forward Euler method and an approximation of a backward Euler method. This connection is used to create a new model, also based on an existing discretization scheme. Furthermore,

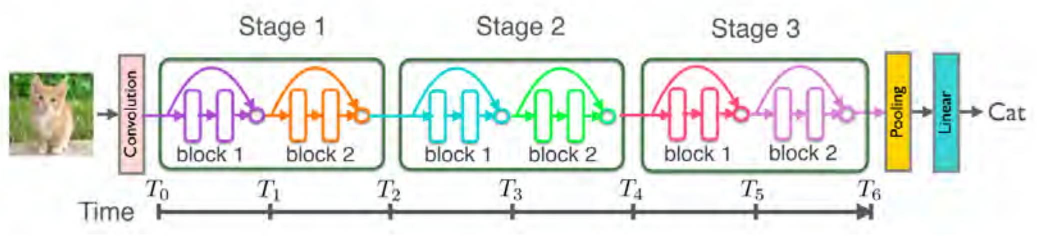


Figure 3.2: Dynamical systems view of ResNets as six Euler steps by Chang [4]. "ResNets equally discretize the time interval $[0, T]$ using time points $T_0, T_1, \ldots, T_d$, where $T_0 = 0$, $T_d = T$ and d is the total number of blocks."

Lu *et al.* already pitch the idea that ODEs are the continuum limit of a neural network, in other words, a neural network with infinite layers. This insight can be used to analyze neural networks with mathematical theory.

The theory of ODEs can be applied to design stable, memory efficient neural networks [4, 41]. An important notion for a neural network is whether it is stable; ensuring that small changes in the input have small effects on the output, and that training does not become very difficult through exploding or vanishing gradients. Since ResNets correspond to ODEs, the knowledge of stable differential equations can be applied to neural networks. This is done by Ruthotto and Haber [15, 41] and Chang [4, 5], who show how to create stable neural network architectures based on the criteria for a stable ODE. Where Ruthotto focuses on stability, Chang also extends his work to reversible architectures. This is when the network can be constructed backwards from the last activations, and allows for more memory-efficient implementations. They are not able to create a fully reversible architecture, and will need to save a few checkpoints, but this does create less need for memory.

The work by Chen *et al.* [6] takes this idea one step further, by replacing parts of a neural network with an ODEsolver. Instead of having a neural network and viewing it as an ODE, they create a neural network which incorporates an actual ODEsolver (Figure 3.3). A black-box ODEsolver becomes part of the network architecture, and depending on the chosen discretization scheme can correspond with one of the networks mentioned earlier. The ResNet is transformed into an IVP (Equation (3.5)), where $x = z_0$ is the initial value, and $z(t)$ the value at time t. This value $z(t)$ can be described by Equation (3.6)[1]. At the final time T, $z(T)$ is equal to the output. Even though the calculations for forward and backpropagation are different for an ODEnet, the general structure for training a neural network stays the same. The training phase still consists of iterations of a forward pass and backpropagation followed by a weight update.

$$\begin{aligned} z_0 &= x \\ \frac{dz}{dt} &= f(z(t), t, \theta) \end{aligned} \tag{3.5}$$

$$z(t) = z_0 + \int_0^t f(z(t), t, \theta)\, dt \qquad \forall t \in [0, T] \tag{3.6}$$

[1]It is important to note that this formula is not a direct solution

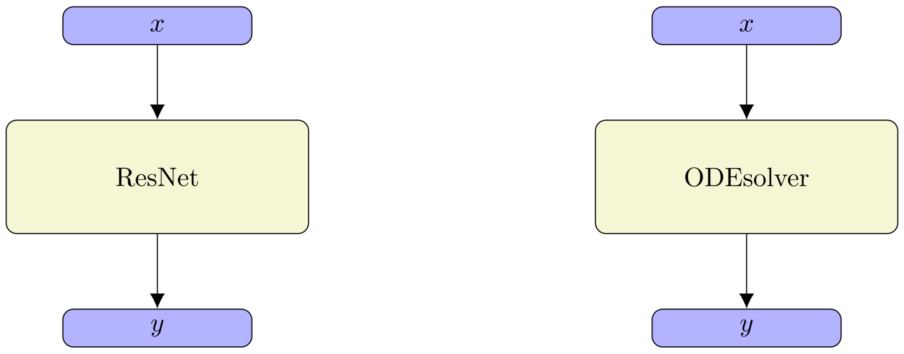


Figure 3.3: Replacing a Resblock with an ODEsolver within a Neural Network.

3.2.2 Forward Pass

We take a look at an ODEnet with a single ODEblock followed by a fully connected layer (Figure 3.4) to illustrate the forward pass. The ODEblock consists of one or more layers f_{ode}, which are passed through the ODEsolver. Similarly as before, the $f_{ode}(x)$ is a function for the network architecture, such as in Equation (3.7). This function can be extended to include convolutions, pooling and other types of layers. This way, the ODEblock can model all neural network architectures, such as a regular neural network or CNN.

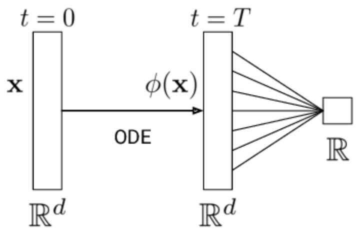


Figure 3.4: ODEnet with one ODEblock followed by a fully connected layer [7].

$$\begin{aligned}
\text{Fully connected layer} \qquad & f_1(x) = \theta^\top x \\
\text{Activation function} \qquad & f_2(x) = \sigma(x) \\
\text{Fully connected + activation} \qquad & f(x) = f_2(f_1(x)) = \sigma(\theta^\top x)
\end{aligned} \tag{3.7}$$

The value of $z(T)$ is calculated during the forward pass by passing the initial value z_0 and the network function $f_{ode}(x)$ to a black-box ODEsolver, such as the Euler, Runge-Kutte or Dopri5 method. Taking a look at the simplest ODEsolver, the Euler updates, will illustrate what happens within a black-box ODEsolver. Take timesteps $\Delta t = \frac{1}{3}$ and $T = 1$. The outcome $z(T)$ is evaluated in three steps, where in general the error is lower when the stepsize is smaller (Equation (3.8)). These three steps are the forward propagation through the ODEsolver, where input z_0 is mapped to output $y = z(T)$. An interesting aspect is that since $z(t)$ is a function of t, its dimensions do not change. Therefore, the dimension of $z(T)$ is the same as z_0. In order to match the desired output dimension a linear layer is added after the ODEblock.

Solved with Euler method (3 steps) :

$$\begin{aligned} z(\tfrac{1}{3}) &= z_0 + \tfrac{1}{3}f(z_0) \\ z(\tfrac{2}{3}) &= z(\tfrac{1}{3}) + \tfrac{1}{3}f(z(\tfrac{1}{3})) \\ z(1) &= z(\tfrac{2}{3}) + \tfrac{1}{3}f(z(\tfrac{2}{3})) \end{aligned} \tag{3.8}$$

The previous example can be extended by adding multiple layers within one ODEblock. One can imagine a convolutional network where every block exists of a convolution f_1, a pooling f_2 and an activation function f_3. In this case, $f_{ode}(x)$ is the function composition of those three layers. The ODEblock, and $f_{ode}(x)$, can also consists of more than one trainable layer with each their own parameters θ. Similar to the example in Equation (3.8), the function $f_{ode}(x)$ will be evaluated multiple times dependent on the ODEsolver.

The computational cost of the ODEnet is not only dependent on the layers within $f_{ode}(x)$, but also on how often this function is evaluated. In case of the Euler method this is once per step, but other methods use more function evaluations during a single forward or backward pass. The number of function evaluations (NFE) is a good way of estimating the computational cost of an ODEnet, which is useful for the architecture design. The NFE is also comparable to the number of hidden layers in a ResNet [6], and forms a good way of comparing the computational cost of the two networks. This is important for interpreting the results in Chapters 7 and 8.

3.2.3 Theoretical Advantages ODEnet

Replacement for ResNet architectures

There are several applications for ODEnets. The main focus of this project is on replacing residual blocks by ODEblocks. This ODEnet is used for image classification. There are three potential advantages of using an ODEnet instead of a ResNet: adaptive step size, less parameters and a trade-off between accuracy and evaluation time.

The first advantage of an ODEnet is that it is easy to implement different discretization schemes, since it incoprorates a general ODEsolver in which different methods can be selected. The area of ODEs and how to solve them has been extensively researched, and more efficient ODEsolvers than the Euler method have been developed. Using an adaptive step size solver, such as *Dopri5*, allows us to monitor the error during the forward pass, and stay below a desired error tolerance. As the model gets more complex, the solver will take more evaluations to prevent an increase in error. This means that the cost scales with the complexity of the problem [6]. Contrary to a ResNet, where the complexity of the model is set beforehand by the number of layers, regardless what is required for solving the problem.

The second advantage of an ODEnet lies within the parameters. Since the layers within an ODEblock are continuous instead of discrete, the parameters within this ODEblock are more 'tied together', and thus expected to be more efficient [6]. If this is the case it would mean that an ODEnet needs less parameters altogether. Chen *et al.* test this hypothesis by creating a ResNet with multiple ResBlocks of 3 layers, and comparing this to an ODEnet with one ODEblock of 3 layers. One ODEblock has approximately the same parameters as one ResBlock. They claim that such an ODEnet would achieve the same accuracy as the ResNet, while greatly decreasing the number of parameters required. However, the experiments in their paper did not have a proper benchmark and thus are not a good proof of the claim. Therefore, the claim remains unsupported.

Another important aspect related to the parameters is the difference between a step in an ODEsolver and the ResNet. Taking a close look at the equations of a forward pass (Equation (3.8)), the ODEnet uses the same $f(x, \theta)$ for every update, where a ResNet uses a different $f_i(x, \theta_i)$ for every layer (Equation (2.3) from Section 2.3). In other words, a ResNet has different parameters in every update step.

The third advantage is that the ODEsolver and the tolerance used in the ODEblock can be changed when implementing the model. The time it takes to run the model can be decreased by setting a higher error tolerance, or a different ODEsolver. This allows the user of the model to make a trade-off between accuracy and time, depending on the prioritization in the current situation.

ODEnet for Time-Series

Other applications of ODEnets are related to time-dependent problems. Neural networks have a discrete set of layers, which is an unfortunate fit to the continuous nature of time. Therefore, a promising application of ODEnets is for time-series, especially if they are irregularly sampled.

An example is the electronic medical data of a patient. Many different variables such as blood values and heart rate are measured. All these variables are measured at different time frequencies and with many missing observations. Usually, to deal with these problems the data is discretized to enforce discrete time bins. Not only is this impractical, it also causes relevant information about the time gaps in between measurements to get lost. For ODEnets the discretization is not necessary, and all important information is retained. Additionally, interpolation between measurements is possible with ODEnets. Research by Rubanova *et al.* [40] shows that a model can be created that correctly learns the latent state of the measurements, and outperform neural networks on inter- and extrapolation tasks.

Several authors have worked on extensions of an ODEnet, and a logical improvement is to extend the ODEnet to Stochastic Differential Equations (SDE). This idea can be found in work by several authors [13, 24, 34, 37, 49]. Liu *et al.* [34] and Peluchetti and Favaro [37] both use the connection with SDE to make ODEnets more robust and well-behaved. Liu *et al.* shows that extending to SDE has similar effects as incorporating some regularization techniques. Furthermore, Peluchetti and Favaro use the connection between infinite depth ResNet and the solution to SDE, and show that the resulting process is well-behaved. Where usually adding more layers to a neural network leads to undesirable properties, such as the dependency on the input decreasing with depth, this does not happen in the

SDE solution.

The idea of SDE is extended by Jia and Benson [24]. Where ODEs are good for continuous processes over time, they cannot incorporate stochastic events. An example of a continuous process with stocastic events is the political view of twitter users. This usually changes slowly over time, but can also change abruptly because of a single tweet. By adding 'jumps' to the network, a hybrid model is created. The new model is a general framework for temporal event sequences, that still has the memory efficiency of ODEnets.

3.2.4 Augmented ODE

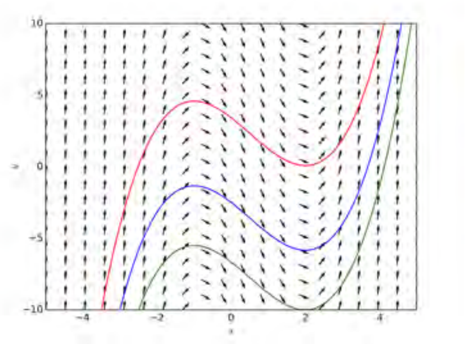


Figure 3.5: A vector field with examples of solutions to an IVP. Every line has a different initial value, resulting in a different solution [36].

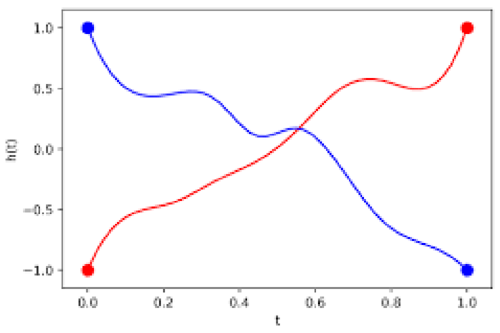


Figure 3.6: Example of a mapping that an ODEnet cannot represent. For the mapping of $f(x) = -x$ the solutions must intersect, indicating this is not possible [7].

Unfortunately, there are functions that ODEnets cannot represent. Recall that an ODEsolver finds a solution $y(t)$ by extrapolating from starting point y_0 with small steps, based on the derivative. An example of such a system is in Figure 3.5, with different solutions for different starting points.

An indication that a function cannot be represented by ODE flows is that the solutions would have to intersect. There is no vector field that corresponds to such a function. Intuitively this means that the derivative at the intersection gives two possible directions, after which it is impossible to know which option the correct directory is. An example of such a function is $f(x) = -x$. Then $f(1) = -1$ and $f(-1) = 1$. This mapping is shown in Figure 3.6. This is a very simple example of a function that cannot be represented by an ODEnet, and therefore a clear example that ODEnet is not an universal approximator.

Fortunately there is a simple solution to this problem. By augmenting the dimensions of the ODEnet, the problem of intersecting solutions can be solved [7]. Recall that an ODEblock is a mapping from $\mathbb{R}^p$ to $\mathbb{R}^p$. This system is augmented by adding an extra dimension, so that $x \in \mathbb{R}^{p+1}$. This is done by concatenating a vector of zeros to x. With this increased dimensionality the ODEnet is not only able to represent the function $f(x) = -x$, but also all other mappings. Work by Zhang *et al.* [54]

shows that with just one augmented dimension the ODEnet becomes a universal approximator.

Adding one or more extra dimensions does not only ensure the ODEnet is a universal approximator, it also eases learning and improves generalization. Experiments by Dupont *et al.* [7] show that an augmented ODEnet needs less function evaluations than a regular ODEnet with the same number of parameters. Additionally, the augmented ODE generalizes better on image datasets such as MNIST and Cifar10.

3.3 Architecture Design

In theory the network architecture can consist of all type of layers, as long as the input and output dimension of the ODEblock are the same. In practice there are layers and functions that lead to complex solutions within the ODEblock, which can cause a large increase in the number of function evaluations required during training. Even worse, an *underflow error* can be encountered. Chapter 7 will provide more insights in the ODEnet architecture by performing several experiments, and elaborate more on what layers are possible, and which combinations are likely to cause problems.

CHAPTER 4

Backpropagation and the Adjoint Method

This chapter will expand on backpropagation methods for neural networks, and how backpropagation can be implemented in ODEnets. First, the traditional backpropagation method, reverse automatic differentiation, will be described. This is then linked to the use of adjoint states. Even though adjoint methods use different terminology than reverse automatic differentiation, they are essentially the same technique for calculating gradients.

In Section 4.3 a neural network is translated into a Lagrange optimization problem. The objective is to minimize the loss, and the network architecture is described by the constraints. Through this optimization problem all the formulas used in training a neural network, the training equations, can be derived [30]. This is an alternative way to derive the formulas for forward propagation, backpropagation and the weight update. Additionally, the equations for backpropagation are defined in terms of adjoint states.

The ideas from the previous sections are tied together, and used to derive the equations for continuous backpropagation in an ODEnet. Again, the ODEnet is constructed as a Lagrange optimization problem, but this time with continuous constraints. This results in training equations and adjoint states that are the continuous counterparts to the traditional discrete methods. The final step will show that backpropagation, with help of adjoint states, can be done with a single call to an ODEsolver. This is equivalent to the method described by Chen *et al.* [6], but derived in an alternative way.

All this leads us to two different backpropagation techniques for ODEnets, the traditional discrete autodiff or the continuous variation. The final section elaborates on the differences between the two, and why the use of the continuous version is preferred. However, also the continuous version is not without disadvantages. The possible stability issue are also discussed, and their implication on ODEnet models.

4.1 Automatic Differentiation

In order to update the weights the gradients of the loss function, the prediction error of the neural network, need to be calculated. Finding the gradient of a function can be very costly, depending on the dimensions of the parameters and the output. The conventional method is to use *automatic differentiation* (autodiff) [25], which consists of three main steps:

1. Trace the forward pass and construct the computational graph (Figure 4.1)
2. Calculate the derivatives for every node
3. Construct the gradient with backwards accumulation

There are two techniques for calculating the gradient, forward and backward accumulation. The difference between these two methods is the way in which the Jacobian is constructed. For example, take a neural network with two hidden layers, f_1 and f_2, and an output layer f_3. The formulation of y and its derivative are in Equation (4.1)[1]. Breaking the derivative down with the chain rule leaves us with two ways to calculate this derivative, shown in Equation (4.2). The difference is the order in which the derivatives are multiplied. In the forward accumulation the Jacobian is built one column (parameter) at a time, and with backwards accumulation per row (output). The most efficient method will depend on the specific problem at hand. When the output dimension is large, forward accumulation is the best method. On the other hand, when there are many parameters it is better to use backward accumulation. In a neural network the number of parameters exceeds the output dimension by a large amount, and backwards accumulation will be much more efficient.

$$\begin{aligned} y &= f_3(f_2(f_1(x))) \\ \frac{\partial y}{\partial x} &= \frac{\partial y}{\partial f_3}\frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial f_1}\frac{\partial f_1}{\partial x} \end{aligned} \tag{4.1}$$

$$\begin{aligned} &\text{Forward accumulation} && \frac{\partial y}{\partial x} = \frac{\partial y}{\partial f_3}\left(\frac{\partial f_3}{\partial f_2}\left(\frac{\partial f_2}{\partial f_1}\frac{\partial f_1}{\partial x}\right)\right) \\ &\text{Backward accumulation} && \frac{\partial y}{\partial x} = \left(\left(\frac{\partial y}{\partial f_3}\frac{\partial f_3}{\partial f_2}\right)\frac{\partial f_2}{\partial f_1}\right)\frac{\partial f_1}{\partial x} \end{aligned} \tag{4.2}$$

Before constructing the Jacobian with backward accumulation, the partial derivatives have to be calculated. For neural networks this is done by tracing the composition of y, and creating a computational graph (Figure 4.1). The derivative is calculated for every node, after which the complete derivative of y is constructed by going backwards over the graph.

[1]We take the derivative with respect to x as an example. The same technique applies for other variables, such as the weights θ.

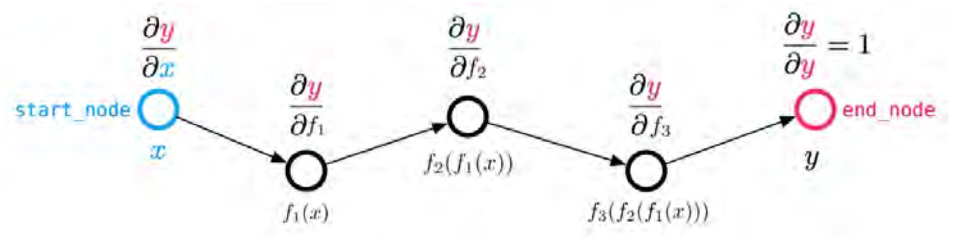

$\frac{\partial y}{\partial x}$
start_node
x
$\frac{\partial y}{\partial f_1}$
$f_1(x)$
$\frac{\partial y}{\partial f_2}$
$f_2(f_1(x))$
$\frac{\partial y}{\partial f_3}$
$f_3(f_2(f_1(x)))$
$\frac{\partial y}{\partial y} = 1$
end_node
y

Figure 4.1: Computational graph for automatic differentiation [25].

4.2 The Adjoint Method

The adjoint method is a different notation for doing reverse autodiff [19], but with the intermediate states defined as *adjoints*. The cost of solving for the adjoint state is independent of the number of parameters [26], and are similar to the initial forward solve [3]. The example from the previous section is extended by defining adjoint states λ (Equation (4.3)). These are the partial derivatives with respect to a node/function from the computational graph. Calculating $\frac{dy}{dx}$ in terms of the adjoint states is equivalent to backwards accumulation (Equation (4.4)), the only difference is that the intermediate states are specifically defined.

$$\lambda_x = \frac{\partial y}{\partial x}, \qquad \lambda_1 = \frac{\partial y}{\partial f_1}, \qquad \lambda_2 = \frac{\partial y}{\partial f_2}, \qquad \lambda_3 = \frac{\partial y}{\partial f_3} \tag{4.3}$$

$$\begin{aligned} \lambda_x &= \lambda_1 \frac{\partial f_1}{\partial x} = \left(\lambda_2 \frac{\partial f_2}{\partial f_1}\right)\frac{\partial f_1}{\partial x} = \left(\left(\lambda_3 \frac{\partial f_3}{\partial f_2}\right)\frac{\partial f_2}{\partial f_1}\right)\frac{\partial f_1}{\partial x} \\ \lambda_x &= \frac{\partial y}{\partial x} = \left(\left(\frac{\partial y}{\partial f_3}\frac{\partial f_3}{\partial f_2}\right)\frac{\partial f_2}{\partial f_1}\right)\frac{\partial f_1}{\partial x} \end{aligned} \tag{4.4}$$

4.3 Lagrange Optimization

The training process of a neural network can be described by an optimization problem with non-linear constraints, as explained in the paper by LeCun [30]. The objective is to minimize the loss, while adhering to the architecture of the network. This architecture is captured by a set of constraints, where every constraint describes the dependency of a layer on the previous one. Altogether this can be formulated as the following optimization problem:

$$\begin{aligned} \min_\theta \quad & Loss = L(z^{(N)}, y) \\ \text{subject to} \quad & z^{(0)} = x \\ & z^{(1)} = f_1(z^{(0)}, \theta_1) \\ & \quad \vdots \\ & z^{(N-1)} = f_{N-1}(z^{(N-2)}, \theta_{N-1}) \\ & z^{(N)} = f_N(z^{(N-1)}, \theta_N) \end{aligned} \tag{4.5}$$

This corresponds to the Lagrangian[2]:

$$\mathcal{L}(z, \theta, \lambda) = Loss(z^{(N)}, y) - \sum_{\ell=1}^{N} \lambda_\ell \left(z^{(\ell)} - f_\ell(z^{(\ell-1)}, \theta_\ell)\right) \tag{4.6}$$

The stationary point of the Lagrangian, which indicates a local minimum, is obtained by $\nabla \mathcal{L}(z, \theta, \lambda) = 0$.

$$\nabla_{z_\ell} \mathcal{L} = -\lambda_\ell + \lambda_{\ell+1} \cdot \nabla_{z_\ell} f_{\ell+1}(z^{(\ell)}, \theta_{\ell+1}) \quad = 0 \tag{4.7}$$

$$\nabla_{z_N} \mathcal{L} = -\lambda_N + \nabla_{z_N} Loss(z_N, y) \quad = 0 \tag{4.8}$$

$$\nabla_{\lambda_\ell} \mathcal{L} = f_\ell(z^{(\ell-1)}, \theta_\ell) - z^{(\ell)} \quad = 0 \tag{4.9}$$

This leads to the following equations. The derivatives with respect to z gives us all λ's defined in terms of z, and the derivative with respect to λ results in the constraints.

$$\lambda_\ell = \lambda_{\ell+1} \cdot \nabla_{z_\ell} f_{\ell+1}(z^{(\ell)}, \theta_{\ell+1}) \tag{4.10}$$

$$\lambda_N = \nabla_{z_N} Loss(z_N, y) \tag{4.11}$$

$$z^{(\ell)} = f_\ell(z^{(\ell-1)}, \theta_\ell) \tag{4.12}$$

Next we calculate $\nabla_{\theta_\ell} \mathcal{L}(z, \theta, \lambda)$. Finding θ that minimizes the objective, the loss, is equivalent to finding θ that minimizes $\nabla_{\theta_\ell} \mathcal{L}(z, \theta, \lambda)$ while satisfying the constraints. Unfortunately, setting $\nabla_{\theta_\ell} \mathcal{L}(z, \theta, \lambda) = 0$ does not directly give the optimal value for θ. However, it does give a condition that θ should satisfy. Therefore, steepest descent is used to find the optimal value for θ. This

[2] λ is the Lagrange multiplier

results in the weight update with learning rate α.

$$\nabla_{\theta_\ell}\mathcal{L} = \nabla_{\theta_\ell} Loss = \lambda_\ell \cdot \nabla_{\theta_\ell} f_\ell(z^{(\ell-1)}, \theta_\ell) \tag{4.13}$$

$$\theta \leftarrow \theta + \alpha\lambda_\ell \cdot \nabla_{\theta_\ell} f_\ell(z^{(\ell-1)}, \theta_\ell) \tag{4.14}$$

From the resulting equations the backward pass (4.10) and (4.11), forward pass (4.12) and weight update (4.14) can be recognized. Similar to the backward pass using reverse autodiff, it is described by intermediate states λ. In other words, the Lagrange multipliers function as adjoint states.

These results show that by formulating the neural network as an optimization problem, the conventional formulas for training are derived (Equation (4.10) - (4.14)). The optimization problem follows the network architecture, and allow us to obtain the the weight update rule and adjoint states. The advantage of this framework is that it allows variations on conventional methods, as it provides a way to derive backpropagation for these variations.

4.4 Backpropagation for ODEnet

Similarly to a traditional neural network, the gradients of the ODEnet are calculated by backpropagation with the use of adjoint states. There are two options for choosing the adjoint state. First is to treat the ODEnet as the discrete steps the ODEsolver takes. This discretization requires one to trace all the computations and store them in memory. Since the number of computations can grow much larger than in a regular neural network, this is not an efficient way of backpropagating. The second option is to define the adjoint states in a continuous manner, and solve for this system.

For a traditional neural network the network architecture in the Lagrange optimization is described by a set of discrete constraints. Given the continuous nature of the ODEnet, the network architecture can be described by continuous constraints instead. Similarly to the previous section the formulas for the forward pass, backpropagation and the weight update can be derived by solving the optimization problem. We describe the constraints in a continuous fashion, and expect that the derived functions are also continuous. This is supported by the knowledge that the complexity of solving a problem for its adjoint state is comparable to a regular forward calculation [26]. In the case of an ODEnet, this means that because the forward pass goes through an ODE, with the help of an adjoint state the backpropagation should also go through an ODE. The goal is to find this adjoint state. In this section we will derive these states by a Lagrange optimization problem, similarly to the last section. This is an alternative derivation to the one given by Chen *et al.* [6], and we will show that our derivation leads to the same adjoint states.

Lemma 4.1 *Formulating the ODEnet as a Lagrange optimization problem*

$$\begin{aligned} min_\theta \quad & Loss = L(z(t_1)) \\ Subject\ to \quad & \dot{z}(t) = \frac{dz}{dt} = f(z(t), \theta, t) \end{aligned} \tag{4.15}$$

Corresponds to the Lagrangian:

$$\mathcal{L}(z(t), \theta, \lambda) = L\left(z(t_1)\right) - \int_{t_0}^{t_1} \lambda(t)(\dot{z}(t) - f(z(t), \theta, t))\, dt \tag{4.16}$$

Optimizing the Lagrangian leads to the following gradients of the Loss:

$$\nabla_\theta L = -\int_{t_1}^{t_0} \lambda(t) \frac{\partial f(z(t), \theta, t)}{\partial \theta}\, dt \tag{4.17}$$

$$\nabla_t L = -\int_{t_1}^{t_0} \lambda(t) \frac{\partial f(z(t), \theta, t)}{\partial t}\, dt + \lambda(t_1) \frac{\partial z(t_1)}{\partial t} \tag{4.18}$$

Both functions derived in Lemma 4.1[3] are equivalent to the equations by Chen *et al.* [6]. Equation (4.17) is equivalent to Equation 51 in their paper, and Equation (4.18) is equivalent to Equation 52.

Corollary 4.2 *From Lemma 4.1, it follow that the adjoint state $\lambda(t)$ is:*

$$\dot{\lambda}(t) = \lambda(t) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} \tag{4.19}$$

Lemma 4.3 *The gradients of L with respect to θ, t and z can be calculated by solving one single ODE system backwards in time. Equation (4.20) is the initial value of this system, with $\frac{\partial L}{\partial \theta} = a_\theta$ and $\frac{\partial L}{\partial t} = a_t$, and Equation (4.21) the derivative with respect to time.*

[3]Proof in Appendix A.1

$$\textit{Initial Value at } (t_1)\text{:} \qquad \begin{bmatrix} \lambda(t_1) \\ a_\theta(t_1) \\ a_t(t_1) \end{bmatrix} = \begin{bmatrix} \nabla_z L(t_1) \\ 0 \\ \nabla_z L(t_1)\frac{\partial z(t_1)}{\partial t} \end{bmatrix} \tag{4.20}$$

$$\textit{Time derivative:} \qquad \begin{bmatrix} \dot{\lambda}(t) \\ \dot{a_\theta}(t) \\ \dot{a_t}(t) \end{bmatrix} = \begin{bmatrix} -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} \\ -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial \theta} \\ -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial t} \end{bmatrix} \tag{4.21}$$

By solving the Lagrange optimization problem for the continuous ODEnet, a continuous equivalent of backpropagation can be derived. All the adjoint states, and therefore backpropagation, can be calculated by a single call to an ODEsolver (Lemma 4.3[4]). A big advantage is that contrary to regular autodiff, there is no need to store a trace of the forward propagation in memory.

4.5 Continuous versus Discrete Backpropagation

As mentioned before, there are two ways of calculating the reverse adjoint during backpropagation for an ODEnet. First is to discretize the system, and then optimize (DTO), or the other way around, first optimize and then discretize (OTD). DTO is the equivalent of regular (reverse) autodiff, performed on the discrete steps within the ODEsolver, and OTD corresponds with the continuous backpropagation technique. The decision for which method to use is important, as they often lead to different computational results [44, 55]. The implementation of DTO for an ODEnet is relatively easy and can be done with information from the forward solve. The downside to this method is that it requires to trace all computations. In the case of an ODEnet, storing the trace can cost a lot of memory. Therefore, when using ODE models the OTD method is used [6]. In this technique the intermediate values do not have to be stored, but can be calculated during the backpropagation.

The paper by Sirkes and Tziperman [44] gives a good overview between the two approaches. In general they advise to use DTO, as the error is smaller and it is preferable when you only want gradients at the initial time. However, they also acknowledge that none of the two methods is inherently better. There are cases where the continuous might be preferred, even though the error cannot be guaranteed. In our case we do not want to impose high memory from the discrete method, so we choose to go for the continuous OTD approach.

[4]Proof in Appendix A.2

4.5.1 Discrete Automatic Differentiation in Practice

In this section the memory costs of using autodiff for ODEnet will be explored. This section provides a clear example of why DTO is not recommended for ODEnets. The memory cost of backpropagation is based on the number of steps from the computational graph that needs to be stored in memory. From Table 4.1 one can see that even using the relatively simple fixed step method RK4, the steps of the trace, and thus the memory requirements, are almost 10 times higher than with a ResNet.

The number of function evaluations during the forward pass (FNFE) for adaptive step size solvers such as Dopri5 is much larger than the examples in Table 4.1. For instance, the smallest FNFE encountered during our experiments was 16 (Figure D.9), but have also gone up to almost 30 (Figure 8.3a). Even though the exact FNFE when using Dopri5 is not known beforehand, it likely much higher than the fixed step methods. Therefore, the trace of Dopri5 will be even larger compared to that of a ResNet. All these steps need to be saved in memory when using autodiff, which is highly inefficient.

ODEsolver	Trace steps	FNFE
ResNet	16	-
Euler	21	1
Midpoint	34	2
RK4	190	4

Table 4.1: The number of steps saved in the computational graph, and the FNFE per fixed step size ODEmethod.

To put this into an even more concrete example, the same ODEnet is trained twice under the same circumstances, but with different backpropagation techniques. Once training with discrete autodiff, and another training with the continuous adjoint method. For a small network, the memory required by discrete autodiff is twice as much, and takes up almost all of the GPU memory[5]. Increasing the network size leads to memory errors. Using DTO to backpropagate is not only inefficient, but also impossible without extremely powerful hardware.

4.5.2 Stability of Continuous Adjoint Method

A big point of discussion is whether the adjoint method used by Chen *et al.* [6] is stable. Several mathematicians agree that the method cannot be guaranteed to be stable, meaning that there might be a large error in the gradients. There are examples of systems that are correct with the forward pass, but have a result that diverges in backpropagation [39].

The value of $z(t)$ is necessary to calculate the gradients $\nabla_\theta L$ and $\nabla_t L$ during backpropagation. Some of these values are calculated during the forward pass, but when using an adaptive step size solver the times t at which $z(t)$ is evaluated during the forward pass are likely different from the values $z(t)$ needed during the backwards pass. Therefore, the values of $z(t)$ need to be calculated separately for the

[5] 12GiB

specific time t during the backpropagation step. This can be done by calculating $z(t)$ forwards in time for every function evaluation, greatly increasing either computational or memory costs. Alternatively, in Chen's method this is done by calculating the value of $z(t)$ backwards in time consecutively with the ODEs to find the gradients, so this value $z(t)$ does not need to be saved or calculated. However, solving $z(t)$ backwards in time can lead to large errors, since there are no reversible adaptive integrators [39]. This is reason for Rackaukas and Gholami *et al.* [9, 56] to question this method and its accuracy.

The software packages CVODES (sundials) [18], CasADi[8] and FATODE [55] implement a variation of the adjoint method by Chen *et al.* [6] that does not require backwards calculation of $z(t)$. The reverse continuous adjoint method is also implemented, but $z(t)$ is calculated differently to prevent the errors mentioned in the last paragraph. Instead of solving for $z(t)$ backwards in time, they create checkpoints for $z(t)$ during the forward solve (Figure 4.2). The $z(t)$ required during backpropagation can then be calculated by forward solving from k_t and backpropagating from k_{t+1} for a t that is in between the two checkpoints.

The memory and computational cost required by the checkpointing is higher then the method proposed by Chen *et al.* First, the checkpoints themselves need to be saved. The cost of calculating $z(t)$ is one forward and one backward pass, compared to $z(t)$ being computed in a consecutively with the gradients while backpropagating.

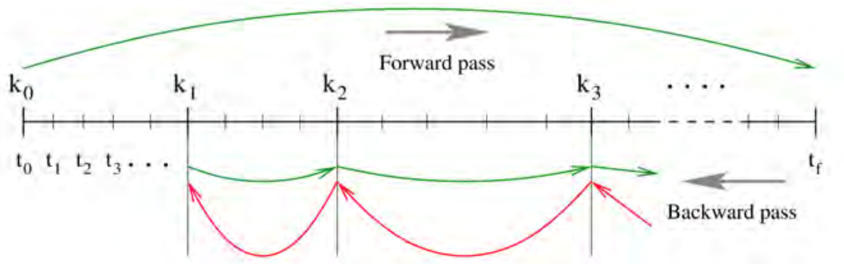


Figure 4.2: Checkpointing scheme from CVODES [18].

The stability remains an interesting question. Implementing the checkpointing scheme from CVODES can improve stability, at the cost of more memory and computation. It is important to note that even though the method by Chen *et al.* can not be guaranteed to be stable, it might be stable enough to use in practice. Results from Chen *et al.*, Dupont *et al.* [7] and this research show that the adjoint method works in experiment settings. Additionally, Rackauckas *et al.* [39] agree that in the case the ODEsolver does not cause large errors, it is the most efficient to use the continuous adjoint as implemented by Chen *et al.* Altogether, further research needs to be done into what the limitations are of the ODEnet, and if there are certain architectures that can prevent stability issues.

CHAPTER 5

Data

The data used in this project are two image classification datasets, FashionMNIST and Cifar10. Both of these datasets are common benchmarks in image classification problems, and provide a clear overview of whether ODEnets can compare to already existing models based on predictive performance.

5.1 FashionMNIST

FashionMNIST [53] is a variation of the well-known MNIST handwritten digits dataset. There has been a lot of progression in image classification since the introduction of the original MNIST dataset, with simple models already reaching accuracy of +95%. This makes it difficult to compare the performance of two models, as the difference between the accuracy is probably very small. Therefore, the FashionMNIST variation is used. This dataset is a little more complicated and makes it easier to see improvements in performance. The images in this dataset are in greyscale, and consist of different types of clothing. The goal is to classify the item on the picture into the correct clothing category.

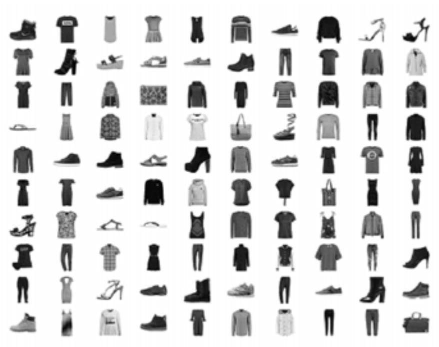
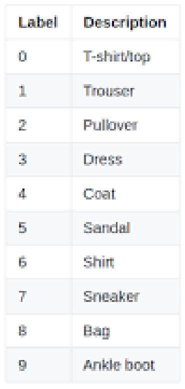


Figure 5.1: Images from the FashionMNIST dataset [53].

The size of the training set is 60.000, and there are 10.000 images in the test set. Every image has a dimension of 28x28, and is in greyscale so only has one input channel. It is a balanced dataset, with 10 labels for the different classes of clothing.



Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Figure 5.2: The labels of the FashionMNIST dataset [53].

5.2 Cifar10

Cifar10 [27] is also an image classification dataset, but quite a bit more complicated then FashionMNIST. The images are in colour, and include some background. Good scores on this classification problem require more sophisticated architectures, whereas FashionMNIST can even be done by the most simple neural networks. This dataset is used as an extension of the results for FashionMNIST. Because the dataset is more complicated, it might show differences between the different architectures more clearly than in the simple FashionMNIST dataset. The architectures are tested on both image classification datasets to strengthen the findings from the first experiments, and to draw a proper conclusion.

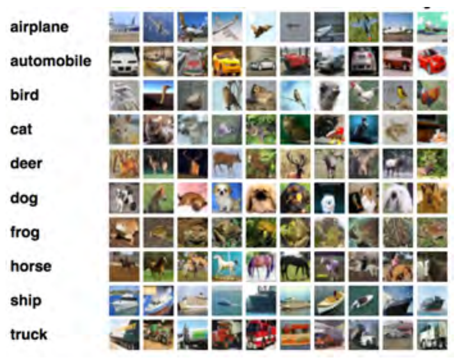


Figure 5.3: Images from the Cifar10 dataset [27].

The Cifar10 dataset is slightly smaller than FashionMNIST. It also has 10.000 images in the testset, but only 50.000 in the train set. Again, there are 10 class labels, with every category evenly represented (6000 per class). The dimensions of the image are 32x32, with 3 channels for the colours RGB.

5.3 Preprocessing

Only minimal preprocessing is done on both datasets. The data is already split in train and a test set, so only a validation set needs to be created. This dataset consists of 20% of the images in the training set. The only preprocessing of the images is normalization of the input. They are all normalized so that the input is between [-1, 1].

CHAPTER 6

Research Method

The goal of this research project is to find out whether ODEnet parameters are more efficient than ResNets for image classification. In order to test this hypothesis, several models are compared to each other. The architectures are designed in such a way that the structure of the ODEnet and the ResNet is equivalent, so the differences in the results can be attributed to the use of an ODEsolver within the model. There are two categories of models used, the simple and complex architectures. More details about these models are in Chapter 7.

Chen *et al.* [6] made the claim that ODEnets are more efficient, and have run experiments to prove this. However, their methods lack a proper benchmark. They compare a small ODEnet with a ResNet with more layers and parameters, but lack an experiment which includes a ResNet that is comparable to the ODEnet. It is known that the ODEnet can get the same accuracy as the bigger ResNet, but it is unsure whether this larger ResNet is an improvement over its smaller counterpart.

This chapter is divided into three main subjects. Section 6.1 describes the limitations inherent to the research, and is followed by Section 6.2 with training details. Section 6.3 provides an overview of the pipeline and logging incorporated in the experiments.

6.1 Limitations Inherent to Model Training

The largest limitation to this research project is time. The training of an ODEnet takes a couple of hours, even on an AWS instance with a GPU. Especially for the complex ODEnet one run takes approximately 10-20 hours. This greatly limits the number of experiments that can be run, and has influenced design choices for the network architectures. Increasing the number of channels and layers increases the training time even more. Therefore, this project is limited to smaller architectures and smaller images.

The long training time has also limited the ability to perform hyperparameter optimization (HPO). For the simple architectures with the FashionMNIST dataset this was still possible, but for the complex architectures it became infeasible due to the time constraint. When running HPO one needs to check multiple models, usually 20 or 50 different combinations. This would take multiple weeks to run!

To overcome the limitation of long training times a smaller data sample is used in some experiments. Instead of the whole dataset, only a subset of 10.000 images is used to train the model. This greatly reduces training time, and allows to train multiple ODEnets and compare the results. This helps the choice for network architecture, and is a way to test the performance of different hyperparameters without doing full HPO. Based on the information gained from the smaller sample runs, only a few experiments with ODEnets on the full training set are required.

Another limitation is that the ODEnet is a very novel technique. This means that not much prior research on the topic exists, especially concerning stability issues. There are activation functions and other network structures which cause errors during training, which we will elaborate on in Chapter 7. Detailed specifics about what those architectures are, however, remain unknown. This occasionally caused delays in the research procedure when creating models. It is practically impossible to predict beforehand when an underflow error will be encountered, what exactly causes the problem and how can it be prevented.

6.2 Training Details

6.2.1 Hyperparameter Optimization

HPO tries to find the most optimal hyperparameters to train a model. With *Hyperopt* [2] a search space of the hyperparameters is made, and the HPO algorithm searches through this space for the optimal parameters. It runs several experiments, and chooses its direction within the search space based on the results. In this project the hyperparameters to optimize are the learning rate, weight decay, the type of optimizer and the number of augmented dimensions. HPO is performed for all ResNets, but is only performed on the simple ODEnet architectures due to time constraints.

6.2.2 Early Stopping

During training of the models *early stopping* is implemented. This is a way to prevent overfitting. Instead of training a fixed set of epochs, training is stopped when the model does not improve anymore. In this project it is set up to stop training when the validation loss has not improved for the last 5 epochs.

6.3 Experiment Pipeline

We created an experiment pipeline to simplify running many experiments. By setting up a pipeline, the same framework can be used to train different models. The pipeline consists of 4 parts, and a main function that ties these parts together into one easy to run experiment. The main function takes all parameters as input, and then goes through the different processes. First, the data is loaded and preprocessed. Then, a model is initialized. This model is then trained and as last also evaluated. This results in a trained model as the output of the main function, together with all the information of the run. Figure 6.1 shows a schematic representation of the pipeline.

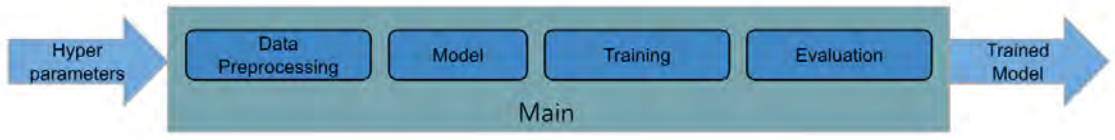


Figure 6.1: Experiment Pipeline

6.3.1 Experiment Logging

To improve the research procedure we have created a framework in which all experiments are automatically logged. Every experiment has a lot of settings, such as the model architecture, hyperparameters and the libraries that are used. Mistakes in the settings are easily made. For example, using the wrong version of a library can lead to the use of an outdated function, which changes the results. Mistakes like this can be prevented with a good experiment set-up where all details of an experiment are meticulously logged. This prevents losing important information and having to perform the same experiment again or even drawing wrong conclusions. Additionally, it improves reproducibility. The exact information that is needed to run an experiment is logged, and can be used to run exactly the same experiment again.

The experiment framework is created in *Python* with the use of *Sacred* [14], *MongoDB* and *Omniboard* [46]. All information of one experiment run is logged, stored and visualized[1]. This includes model configuration, parameters, output and all the source files.

[1]Screenshots of Omniboard are in Appendix B.3

CHAPTER 7

Design of ODEnet architecture

Because there is not much information about the possible architectures of an ODEnet, this needs to be researched before deciding on what models to use in the experiments. This chapter will elaborate on what types of layers, activation functions and other design choices lead to successful models. The biggest hurdle is encountering an *underflow error* during training, or an increase in the number of function evaluations. While the latter only increases the training time, the former causes the training to completely terminate. This section will experiment with different architectures, and investigate the choices that can be made to prevent these errors during training. These results form the foundation for the final model architecture. Their details are in Section 7.3.

7.1 Training/Underflow Error

The design of the neural network architecture in combination with an ODEsolver is an important step. Certain architectures correspond to complex ODEs, which can lead to complications during training of the model. This either leads to a higher error with a fixed solver, or to a smaller stepsize with an adaptive solver. The adaptive solver decreases the stepsize in order to stay bellow the desired error tolerance, which will require more function evaluations per forward pass or backpropagation. Not only does this increase the NFE and thereby the training time, it can also cause an *underflow error*. This fatal error is encountered when the stepsize goes to zero, and the model becomes practically unsolvable.

There are a few easy fixes that prevent a decrease in stepsize, such as increasing the error tolerance or getting an approximation of the solution with a fixed step solver. But neither option is very desirable. The better solution is to choose a better architecture for the ODEnet. Unfortunately this step is far from trivial. As the combination of neural network and ODEsolvers is relatively new, there is no extensive research to what layers and functions lead to a complex neural network. This section will explore different architectures, and their influence on the NFE during training.

7.1.1 Activation Functions

There are several activation functions to add non-linearity to the ODEnet, and ensure the network is a universal approximator [20, 32]. Commonly used functions for hidden layers are ReLU, tanh, or Softplus. Nowadays the general recommendation is to use ReLU, even though it is not differentiable everywhere [10, 12]. There is no consensus yet on whether all these activation functions work well within an ODEnet, since research shows conflicting evidence. On one hand, the networks used by Chen *et al.* and Dupont *et al.* [6, 7] incorporate ReLU layers succesfully. On the other hand, Chen *et al.* advises users of the *torchdiffeq* library[1] to avoid non-smooth functions such as ReLU. The intuition is that ReLU might cause underflow error, but this is not supported by any evidence.

In order to have a better understanding of what activation function to use, an ODEnet is created with ReLU, tanh and Softplus. To speed up training time only a subset of 10.000 samples of the FashionMNIST dataset is used. The results provide insight in the effect of the activation function on the number of function evaluations, accuracy and loss.

The first experiment looks into an activation layer right before the ODEblock (Table 7.1). While the accuracy and loss are mostly unaffected, there is a clear difference between the function evaluations during backpropagation (BNFE). In general the BNFE increases during the first few epochs, and then plateaus (Figure 8.2a). The ReLU layer clearly has the highest BNFE, where tanh and no activation functions only have 27. Since the gain in accuracy is so small, the best choice would be a tanh activation function, or to skip the layer.

The second question is what activation function to use within an ODEblock (Table 7.2 and Figure 8.2b). Here, the Softplus has the highest BNFE, and the difference between tanh and ReLU is smaller. Tanh is a little faster, but also a little less accurate. Both are valid options to use as activation functions within an ODEblock.

Altogether it can be noted that there is a difference between the activation functions, but also that there is no clear outlier. In general the tanh and ReLU are the better options. Interesting is that even within the ODEblock the ReLU did not cause any problems, which follows the experimental results from [6, 7]. This is in contrast with the statement made by Chen *et al.* Additionally, only the BNFE increased, but a fatal underflow was never encountered.

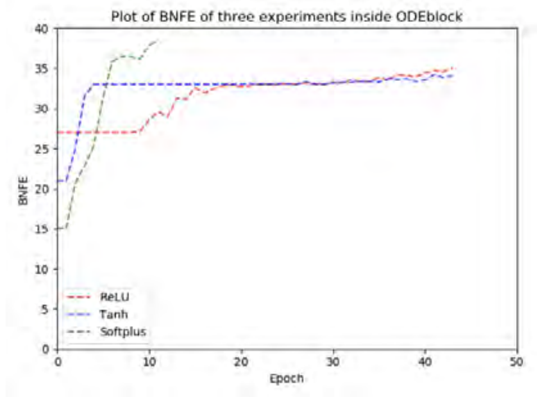
Activation function	Accuracy	Loss
Softplus	86.06	0.3520
ReLU	87.06	**0.2920**
Tanh	87.04	0.2932
no activation	86.90	0.2953

Table 7.1: Accuracy and Loss for activations functions before the ODEblock.

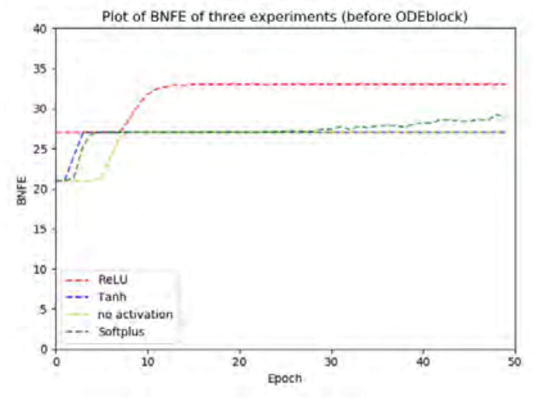
Activation	Accuracy	Loss
Softplus	81.54	0.5343
Tanh	87.1	0.3666
Relu	88.3	**0.3369**

Table 7.2: Accuracy and Loss for activation functions within the ODEblock.

[1]The library created by Chen *et al.* to implement ODEnets



(a) Three different activation functions placed inside the ODEblock.



(b) Four different activation functions placed before the ODEblock.

Figure 7.1: BNFE per epoch

7.1.2 Stride

An easy way to speed up training is adding stride to the convolutional layer. This decreases the width and height dimensions of an image, reducing the number of computations required. However, since the dimensions change this cannot be done within the ODEblock without adding padding. Therefore, stride should be implemented before entering the ODEblock.

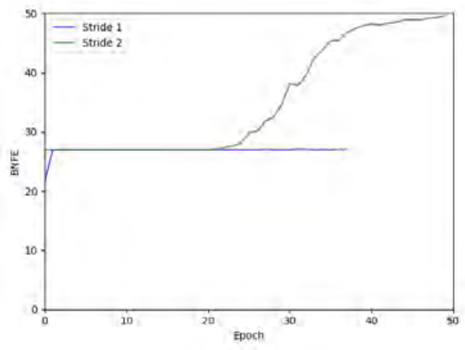


Figure 7.2: BNFE per Epoch between a stride of 1 and a stride of 2.

Adding a stride of 2 to the model decreases the training time drastically, now only half of the original time. This is a major improvement, but there is also a big drawback. With a stride of 1 the BNFE stays steady during training, but with a stride of 2 this increases (Figure 7.2). This means that the network structure is becoming more complex, and more difficult for the ODEsolver to solve. It also slightly decreases the accuracy of the network. However, the big decrease in training time is worth the small decrease in accuracy.

7.1.3 The Normalization Layer

An unexpected error occurs when adding a normalization layer to the architecture. While a batchnorm leads to an underflow error, this does not happen with groupnorm. This means the problem is not with the normalization layer in itself, but with the difference between batch- and groupnorm. Batchnorm is more stochastic and inaccurate then group normalization [52]. This difference could be vital to the ODEnet architecture, as groupnorm is a more robust alternative to batchnorm. Therefore, if an

ODEnet has underflow error and a batchnorm layer, exchanging this with a groupnorm is likely to be an easy fix.

7.1.4 Variations of the Final Time

The models used by Chen *et al.* all incorporate an ODEsolver with a timescale from [0,1]. This is equivalent to one layer of the ResNet, but taking multiple steps between 0 and 1. An interesting alternative that is not investigated by Chen *et al.* is setting the final time T_n to a different time. This corresponds more to the structure proposed by Chang [4] depicted in Figure 3.2 from Chapter 3, where every integer n corresponds to a ResBlock. In an ODEnet the parameters for these blocks will be the same, so it resembles a repetition of the first block. Experiment results (Table 7.3) show that just increasing T_n mostly increases the FNFE and backward pass. A higher T_n increases training time, but there is also a slight decrease in accuracy.

This does raise the question of whether the network could be improved upon with parameters that are also a function of time. This would result in different parameters $\theta(t)$ corresponding to different blocks at T_n. However, incorporating θ as a function of time is not straightforward, but an interesting angle for future research.

Tn	Accuracy	Loss	FNFE	BNFE
1	**88.02**	**0.3417**	**26**	**27**
2	87.86	0.3506	32	39
5	86.88	0.3850	38	63
10	85.36	0.4431	50	75-80[1]

Table 7.3: Accuracy and Runtime for different Tn's (trained on FashionMNIST).
[1]The BNFE increased from 75 to 80 during training.

7.2 Complexity ODEblock and Fully Connected Layers

It is not the case that every hidden layer with more parameters also has a higher complexity. The relation between the two is dependent on the type of layer. In general, fully connected layers have many parameters, but the forward and backward pass do not require many computations. The order of this layer is $O(in \cdot out)$ and is the same as the number of parameters in the layer. The number of parameters and complexity of the convolutional layer is dependent on the filter size f, the number of input channels C_{in}, the number of output channels C_{out} and the image dimensions height h and width w. The convolutional layer has only a couple of parameters ($f^2 \cdot C_{out} + 1$), but the order of complexity is much larger at $O(C_{in} \cdot C_{out} \cdot f^2 \cdot h \cdot w)$. As a rule of thumb, the computational time of layers such as the fully connected layer and pooling is 5-10% compared to convolutional layers [16].

This means that in the ODEnet and ResNet most of the parameters will be in the fully connected layer, while most of the complexity will come from the blocks of convolutions. Therefore, reducing the number of parameters within the ResNet and ODEnet blocks has a big impact on the overall complexity. This impact is much larger then reducing the size of the fully connected layer.

7.3 Experiment Models

The insights from the last two sections have lead to two different sets of models to compare. First is the simple architecture, which only consists of a few layers. The second set of experiments have a more complex architecture with more and different types of layers. A small ResNet, large ResNet and an ODEnet are created for the simple architecture. For the complex architectures a ResNet and an ODEnet are created. The details of these networks are provided in Section 7.3.1 and Section 7.3.2

7.3.1 Simple Architecture

The first set of models have a simple architecture, with only one block of layers. The block consists of a convolutional layer, groupnorm and is followed by a ReLU activation function (Figure 7.3). Before entering the block there is a 1x1 convolutional layer to ensure the dimensions of the input and output of the ODEblock are the same. After the block there is a fully connected layer that makes the final classification of the 10 labels. The ODEnet has one ODEblock as described before, where the convolutional layer has 64 channels and is augmented with 2 extra dimensions. To match this, one small ResNet is constructed with one block and 66 channels. In order to compare both models with a network that has more parameters, an additional larger ResNet model is designed. This larger model has three blocks, all with 66 channels. This corresponds computationally with three updates within the ODEblock, but has 3x as many parameters (Table 7.4). A more detailed description of the architecture can be found in Appendix C.1, C.2 and C.3.

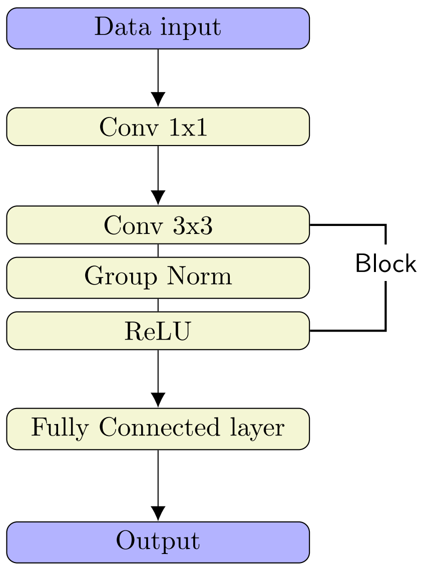


Figure 7.3: Simple Network Architecture

	Model Type	Number of Channels	Number of Parameters	Parameters not in fully connected layer
FashionMNIST	ResNet small	66	169 000	40 100
	ResNet large	66	248 000	119 000
	ODEnet	64+2	169 000	40 000
Cifar10	ResNet small	258	1 270 000	607 000
	ResNet large	258	2 470 000	1 810 000
	ODEnet	256+2	1 270 000	607 000

Table 7.4: Number of parameters per simple model.

7.3.2 Complex Architecture

The architectures used for the first set of experiments are very simple, and can very easily be extended with much improvement in accuracy. A more complex ResNet with different number of channels and pooling layers can achieve a 10% increase in accuracy on Cifar10, with less parameters. Even if the ODEnet is an improvement in the first round of experiments, it might not be of use if it is not tested against better performing neural network architectures.

Therefore, instead of fitting a ResNet to an intuitive ODE structure, the second architecture is designed the other way around. A more complex ResNet model is created, and then an ODEnet to match its structure. The models consist of three sets of a convolutional layer, max pooling, groupnorm and a ReLU activation. An overview of the model is in Figure 7.4, a more detailed version is in Appendix C.4 and C.5.

While it is uncommon to design ResNets with pooling layers, it is necessary in this project to limit the training time of the models. Pooling is difficult to incorporate a residual connection, since it reduces the dimensions of the output. Additionally, work by Springenberg and Dosovitskiy [45] show that pooling layers are not necessary. However, without pooling layers the training time of the ODEnet would be too large to run the experiments. By adding the pooling layers this time is reduced significantly.

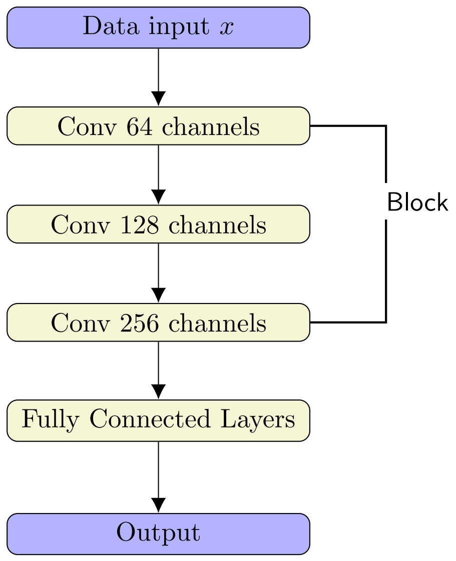


Figure 7.4: Complex architecture, where every convolution is followed by pooling, groupnorm and activation.

	Model Type	Number of Parameters	Parameters not in fully connected layer
FashionMNIST	Complex Architecture	394 000	353 000
Cifar10	Complex Architecture	413 000	372 000

Table 7.5: Number of parameters per complex model.

Getting Correct Dimensions

It is not a trivial task to fit any neural network architecture into an ODEnet. An important point is that the ODEblock needs to have the same input and output dimensions. This requirement limits the type of layers that can be used in a convolutional network, as many types of layers cause a change in dimensions. Techniques such as pooling, stride and a convolution to a different number of channels all have this property. Not only are these layers a way to increase the accuracy of a model, they are also ways of downsampling the data in order to prevent large architectures. There are two types of changes in the dimensions. First, the height and width of the input are reduced by the pooling layers, and second the number of channels increases with the convolutions.

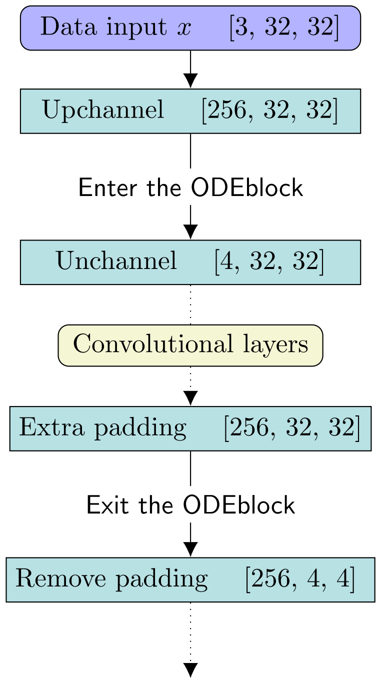


Figure 7.5: Framework for ensuring the correct dimensions. Including the output dimension of every layer with Cifar10 [channels, height, width].

The most difficult change in dimension, the decrease of image height and width, can be circumvented in the ResNet. By placing the residual connection before the pooling layer the reduction in dimension does not have to be incorporated.

Unfortunately, the reduction in dimension cannot be avoided in the ODEnet. The input and output of the whole ODEblock need to match, so a different technique is required. We add a zero padding to the output of the ODEblock, so it has the same dimensions as the original input when exiting the ODEblock. Then, before entering the fully connected layer, the padding is removed again. This ensures that both the restrictions of the ODEblock are met, while also incorporating the advantages of the pooling layers.

The second change in dimension is relatively simple to implement. There are two common techniques used in ResNets; padding the input with extra zero entries or by adding a 1x1 convolution. Since the latter introduces extra parameters the first method is implemented in both the ResNet and the ODEnet. This method is also very similar to incorporating the augmented dimensions. The technique is incorporated into the ODEnet before entering the ODEblock. Knowing the last layer will have 256 channels, a few zero channels are added to the input image. This augmented image, with the correct number of channels, is passed to the ODEblock. Here, the first action is to remove the zero channels. This prevents unnecessary parameters in the next convolution. Now, the dimensions are correct without changing what would happen to the image itself. As a minor detail, when removing the dimensions one zero channel is kept. This extra channel is enough to ensure the model is an universal approximator, as explained in Section 3.2.4.

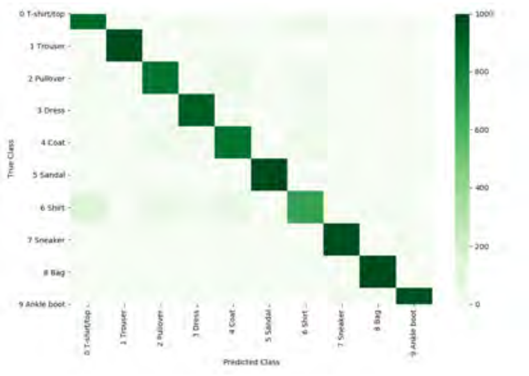
CHAPTER 8

Results

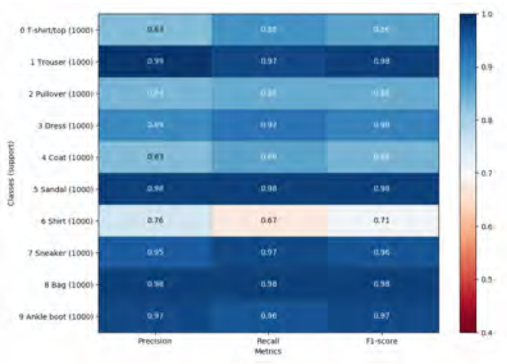
This chapter reports the results from all the different model experiments. Section 8.1 will elaborate on the results from the simple architectures, and Section 8.2 will focus on the complex architectures. The last section incorporates the results of setting different tolerances or ODEsolvers during evaluation.

8.1 Simple Architecture

8.1.1 FashionMNIST

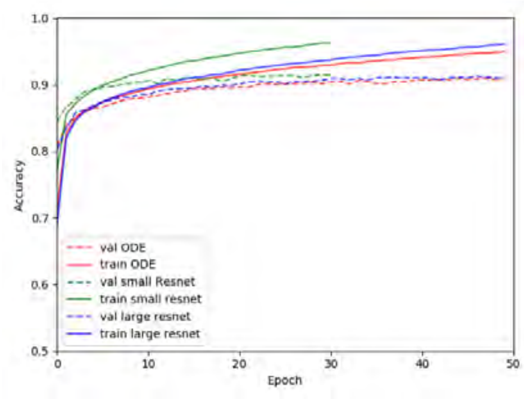


(a) Confusion Matrix

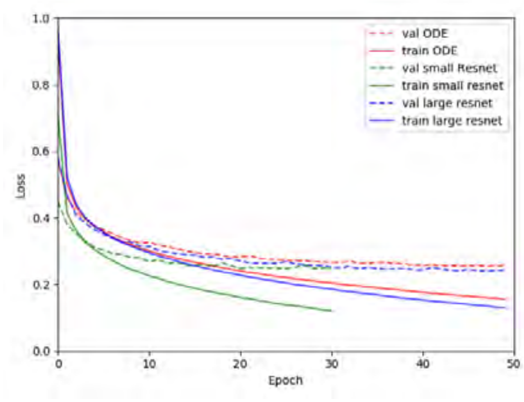


(b) Classification Report

Figure 8.1: Prediction scores on FashionMNIST for the simple ODEnet.

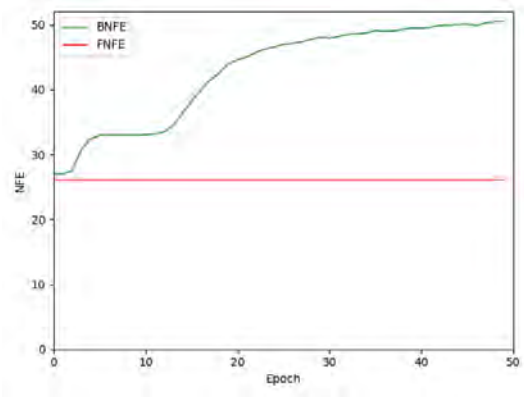


(a) Accuracy during training FashionMNIST with simple architectures.

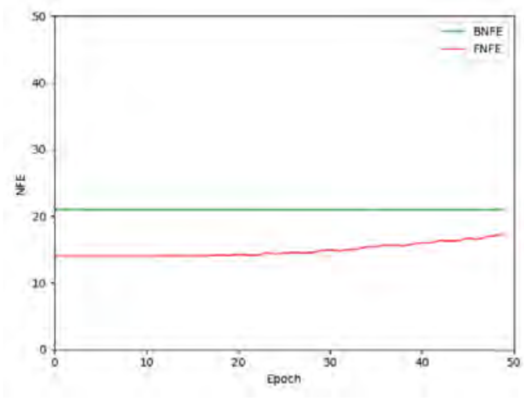


(b) Loss during Training FashionMNIST with simple architectures.

Figure 8.2: Accuracy and Loss for the ODEnet, small and large ResNet.



(a) Simple ODEnet



(b) Complex ODEnet

Figure 8.3: NFE per epoch on FashionMNIST.

The ODEnet is able to classify the FashionMNIST data well. From Figure 8.1 it is clear that most classes are correctly classified, even though it has difficulty with the *shirt* category. The overall accuracy is 90% (Table 8.1), and this performance is on par with other simple ResNet models. Additionally, the training and validation curves are relatively close together (Figure 8.2). This indicates that the model is not prone to overfitting, and has a smooth learning curve. One downside is that the BNFE increases halfway through training (Figure 8.3a). This is expected as explained in Section 7.1.2, and shows that the learned model gets more complex during training.

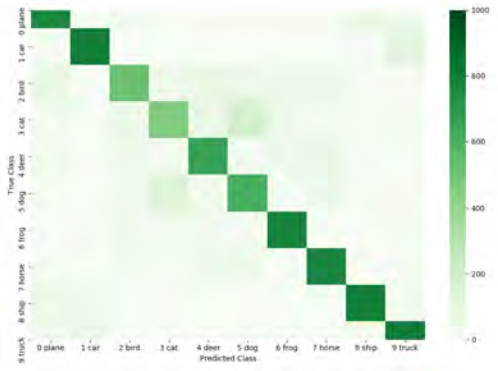
The predictive results for the three models, ODEnet, small ResNet and the large ResNet are very similar in all regards. The main difference is that the training loss and accuracy of the small ResNet is much lower, but the validation curves are comparable to the other networks. This could be an indication that the model is prone to overfitting.

A very important aspect is that the time to run one epoch for the ODEnet is on average 4 minutes when running on one GPU. Using the same hardware, the small ResNet only takes 17 seconds, and the large ResNet 24 seconds (Table 8.1).

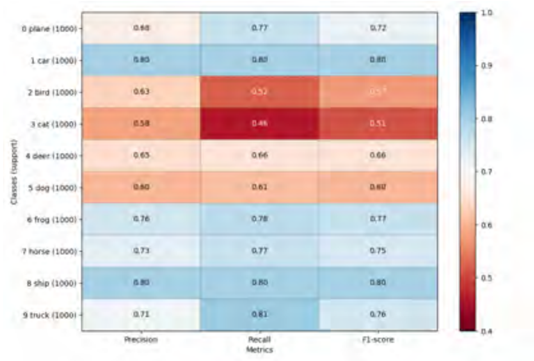
Model	Architecture	Dataset	Accuracy	Loss	Average epoch time
ODEnet	Simple	FashionMNIST	90.36	0.2706	4 min.
small ResNet	Simple	FashionMNIST	90.56	0.2686	17 sec.
large ResNet	Simple	FashionMNIST	90.30	0.2695	24 sec.
ODEnet	Complex	FashionMNIST	**90.80**	**0.2523**	18 min.
ResNet	Complex	FashionMNIST	90.73	0.2552	23 sec.
ODEnet	Simple	Cifar10	69.80	0.8751	18 min.
small ResNet	Simple	Cifar10	71.26	0.8254	28 sec.
large ResNet	Simple	Cifar10	70.95	0.8542	53 sec.
ODEnet	Complex	Cifar10	75.45	0.7168	20 min.
ResNet	Complex	Cifar10	**77.31**	**0.6736**	28 sec.

Table 8.1: Complete table of test performance for all architectures

8.1.2 Cifar10



(a) Confusion Matrix



(b) Classification Report

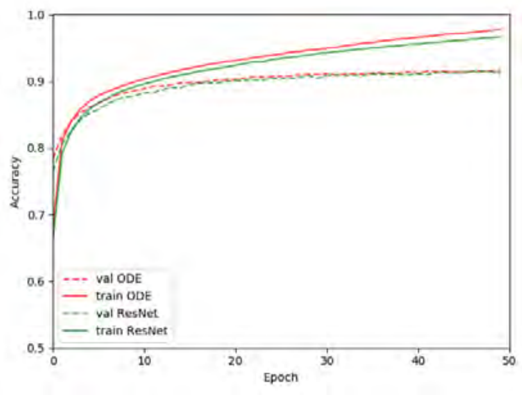
Figure 8.4: Prediction scores on Cifar10 for the simple ODEnet.

The classification report (Figure 8.4b) of the ODEnet model trained on the Cifar10 dataset shows that the model does not perform as good as on the FashionMNIST dataset. Since the Cifar10 dataset is more difficult, this was expected.

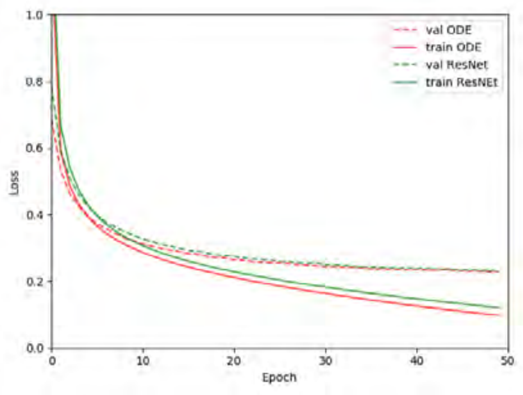
The performance of the other two models, small and large ResNet, is largely the same as the ODEnet. This corroborates the results found in the FashionMNIST experiments. Figures supporting this are in Appendix D.2. Again the training time of the ODEnet is much longer than that of either of the ResNets. In this case the ODEnet trains on average 18 minutes per epoch, while the small and large ResNets take half a minute and a minute respectively.

8.2 Complex Architecture

8.2.1 FashionMNIST

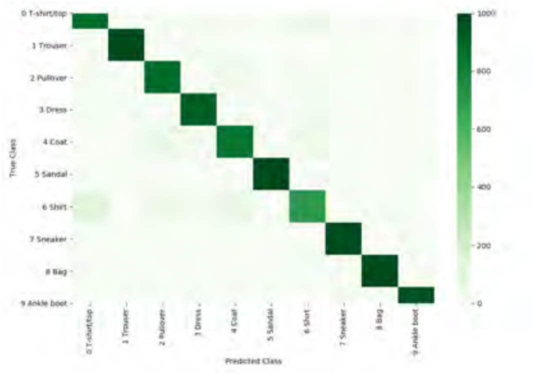


(a) Accuracy during training FashionMNIST with complex architectures.

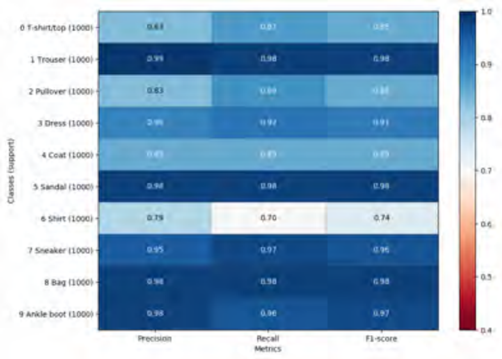


(b) Accuracy during training FashionMNIST with complex architectures.

Figure 8.5: Accuracy and Loss for the ODEnet and ResNet.



(a) Confusion Matrix ODEnet



Classes (support)	Precision	Recall	F1-score
0 T-shirt/top (1000)	0.83	0.87	0.85
1 Trouser (1000)	0.99	0.98	0.98
2 Pullover (1000)	0.83	0.89	0.86
3 Dress (1000)	0.90	0.92	0.91
4 Coat (1000)	0.85	0.85	0.85
5 Sandal (1000)	0.98	0.98	0.98
6 Shirt (1000)	0.79	0.70	0.74
7 Sneaker (1000)	0.95	0.97	0.96
8 Bag (1000)	0.98	0.98	0.98
9 Ankle boot (1000)	0.98	0.96	0.97

(b) Classification report ODEnet

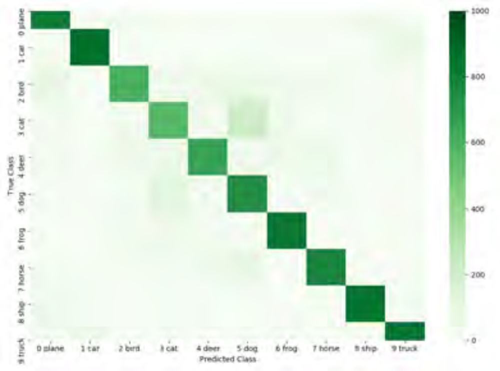
Figure 8.6: Prediction scores on FashionMNIST for the complex ODEnet.

Also for the complex architectures trained on FashionMNIST, the ResNet and ODEnet perform very similarly. In this case the learning rate and weight decay of the models are the same, which leads to practically identical learning curves in Figure 8.5. Table 8.1 does show that the ResNet generalizes slightly better, indicated by the performance on the test set.

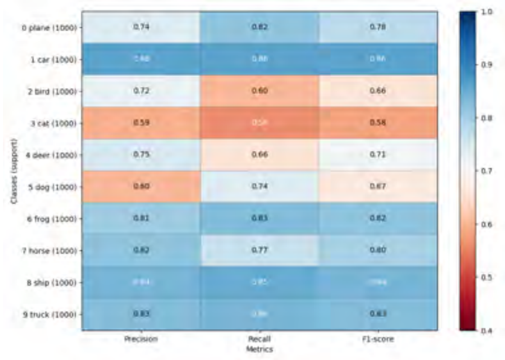
Since this is only a simple dataset, the predictive performance of this set of models is only marginaly better then the more simple architecture (Table 8.1). The training time of both models does increase compared to the simple architectures. The ODEnet takes 18 minutes per epoch compared to 4 minutes, and the training time from the ResNet has increased from 17 to 27 seconds.

An interesting aspect is that the NFE, both for the forward and backward pass, is consistently lower during training (Figure 8.3). Where in the simple architecture has a higher start values and a large increase in BNFE halfway through training, the values of the complex ODEnet remains low. There is only a slight increase in the FNFE, but the maximum is still lower than the FNFE of the simple architecture.

8.2.2 Cifar10



(a) Confusion Matrix ODEnet



(b) Classification report ODEnet

Figure 8.7: Prediction scores on Cifar10 for the complex ODEnet.

For the Cifar10 dataset there is a clear improvement when using the complex architecture. The accuracy increases with 5% and the loss drops (Table 8.1). This is also clear from the classification reports in (Figure 8.4b and 8.7b). The other aspects are similar to the results found earlier. Again, the ODEnet has a much higher training time than the ResNet, while the performance metrics are very similar.

A very interesting and unexpected aspect, that does not occur in the other experiments, is that the FNFE decreases during training (Figure 8.8). This suggests that the problem becomes less complex during training, and causes the epoch time decrease during training.

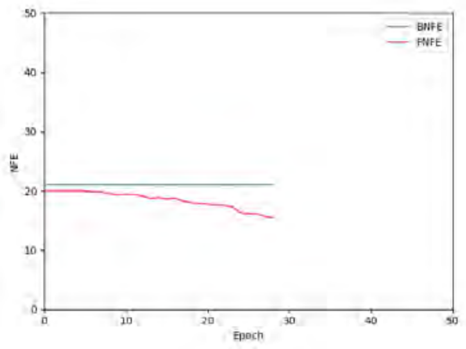


Figure 8.8: NFE per epoch on FashionMNIST with complex ODEnet architecture.

8.3 Different Evaluation Settings

One of the advantages of using an ODEblock claimed by the authors of Chen *et al.* [6] is that the the tolerance can be changed during the evaluation. This idea can be extended by also changing the ODEsolver. Having a lower tolerance or a simpler ODEsolver can decrease runtime, but also accuracy. This means there is a trade-off between the two, and dependent on the situation runtime or accuracy can be prioritized. In order to test this claim, a model is trained with Dopri5, but is evaluated with different ODEsolver settings. The results in Table 8.2 show that there is indeed a trade-off, and that the more simple solvers have less FNFE, and thus cost less time. Additionally, the results of these solvers are not as accurate as the adaptive solvers. However, no difference is observed between the different tolerances of the adaptive step size.

			Cifar10		FashionMNIST	
Architecture	ODEsolver	Tolerance	Average FNFE	Accuracy	Average FNFE	Accuracy
Simple	Euler		1	83.48	1	37.19
	RK4		4	88.90	4	61.79
	Dopri5	[0.001]	26	90.36	28.64	69.82
	Dopri5	[0.01]	26	90.36	28.64	69.82
	Dopri5	[0.1]	26	90.36	28.64	69.82
Complex	Euler		1	90.88	1	75.09
	RK4		4	90.80	4	75.45
	Dopri5	[0.001]	19.06	90.80	16.96	75.45
	Dopri5	[0.01]	19.06	90.80	16.96	75.45
	Dopri5	[0.1]	19.06	90.80	16.96	75.45

Table 8.2: FNFE and accuracy for different evaluation solvers and tolerances.

CHAPTER 9

Discussion

9.1 Interpretation of Results

There are three main reasons why the ODEnet was expected be a good replacement for a ResNet for image classification: more efficient parameters, cost scaling with complexity, and a trade-off between cost and accuracy. Unfortunately, these advantages are not found in our experimental setting. The results from the different experiments all follow the same pattern. There is no big difference between the predictive performance of the models, but the ODEnet requires a lot more computations, and thus more time, to get these results. Since the expected advantages cannot be found and the long training time of the model, the ODEnet is not a good replacement for ResNets when doing image classification. The following section will elaborate on the results, and provide reasoning as to why the initial hypothesis was not confirmed.

We extended the experiment by Chen *et al.* [6] with a proper benchmark in order to review the claim that the parameters of an ODEnet are more efficient. The results by Chen *et al.* show that the accuracy of the large ResNet and the ODEnet are the same. Since the ODEnet has much less parameters this would indicate that the ODEnet needs less parameters to get the same accuracy as a ResNet. However, our experiments show that all simple architectures, small-, large ResNet and ODEnet, have remarkably similar classification scores. The small ResNet with the same number of parameters as the ODEnet even slightly outperforms both the larger ResNet and the ODEnet. Additionally, the predictive performances of the complex ResNet and ODEnet are also similar. This demonstrates that the parameters of an ODEnet are not better than the parameters of a ResNet. Since the extra parameters in the large ResNet as used by Chen *et al.* are not necessary to get the performance, the claim of more efficient parameters is not supported.

A second point of discussion is that the cost of training an ODEnet are very high. This relates to the other two expected advantages. The observation that the NFE tends to increase during training is a good indication that the cost of the network does scale with the underlying complexity. Additionally, changing the ODEsolver during evaluation does affect the time it takes to classify the new images. However, the initial cost of the model are way larger than that of a ResNet. This makes the advantages of scaling and the trade-off obsolete. The ResNet is already much faster, so even if has more layers than necessary it is still much better than the ODEnet.

Even though the complex ODEnet was not an improvement over the ResNet, the experiments do show another important aspect. Namely, that padding the dimension and channels of the input and output of the ODEblock can be incorporated succesfully. This opens up the possibility for different network architectures, and eases the limitation of input and output dimensions having to be exactly the same. The risk of this extra padding was that it could result in a complex system that is difficult for the ODEsolver, and causes errors or high computational costs. However, the opposite is true. The experiments show that the complex architectures were able to train on the classification problem, and even had a lower NFE than the other networks.

9.1.1 Analysis of Results

An important question is why the results are not corresponding with the expected advantages. Not finding more efficient parameters can be related to a difference noted already in Section 3.2.3. In every step of the ODEsolver the same parameters are used again. Contrary to multiple ResBlocks where every ResBlock has its own parameters. Since the ODEnet did not improve on the ResNets, this points to the notion that repeating an identical block does not improve accuracy. The power of different blocks lies within the different parameters.

The cost of training an ODEnet are very high because the ODEsolver takes multiple function evaluations. The adaptive step size solvers have a minimum of six NFEs [6], which is equivalent to going through a ResNet six times. However, our results show that the NFE is usually much higher. For instance, the lowest FNFE encountered in the experiments was 14. When the structure gets more complex the NFE gets even higher. The large amount of evaluations is the reason the ODEnet is so much slower than the ResNet.

9.2 Future Research

An interesting direction for future research is to incorporate a time dependency in the weight parameters. This would mean that every step within the ODEblock has a different parameter value $\theta(t)$, as the t in every step is different. In the current networks the ODEblock is mostly extra computation, but with different parameters it is more likely to make an improvement in the performance. However, this is a big change in structure to regular NNs. It is not clear how to create such parameters, and how the forward pass, backpropagation and weight update would change.

That the ODEnet is not a good alternative for image classification, does not mean the model should not be used at all. As mentioned in Section 3.2.3, there are other applications for ODEnets. One should take into account that ODEnets are very slow, but when dealing with time dependent problems there is no good alternative that deals with the continuous nature of time properly. If hardware or time is less of a priority, the ODEnet can prove to be a better fit. Think for instance of medical data. Accuracy is of the utmost importance, and way more important than the cost involved in running models.

CHAPTER 10

Conclusion

The first aim of this research is to investigate whether ODEnets are have more efficient parameters than ResNets, in particular of image classification. The results clearly show that this is not the case. The classification scores of both models are very similar, but the training time of the ODEnet is 10-50 times as long. This makes the ODEnet a very impractical model, unless one has access to powerful hardware.

The second objective is to provide more details of the forward- and backpropagation when incorporating an ODEsolver. The first can be examined by looking at the discretization scheme the ODEsolver uses, which for the Euler method greatly resembles a regular forward pass for a ResNet. This gives a good overview, and can be extended to the computations that happen when using a more sophisticated ODEsolver. We use this overview to understand why the ODEnet does not perform better than a ResNet, and to find directions for future research.

The backpropagation method by Chen *et al.* [6] is the continuous equivalent of reverse autodiff, the traditional backpropagation technique. We show this connection by first deriving autodiff through Lagrange optimization, and applying the same derivations on the continuous equivalent. This also indicates that autodiff can be implemented into an ODEnet.

The stability of the new continuous backpropagation remains an important issue. The implementation by Chen *et al.* can encounter stability issues, but should not immediately be dismissed. In the cases it works, it does save a lot of memory compared to other checkpointing methods as used in *cvodes*. The method by Chen *et al.* has been succesfully implemented in their research, and by others such as Dupont *et al.* [7]. This is supported by the results from this project, as stability issues during backpropagation were not encountered. Therefore, the decision of which backpropagation to use is dependent on the application and the limitations imposed by the hardware.

In theory all neural network architectures can be incorporated in an ODEblock, as long as the input and output dimensions are equal. However, in practice some types of layer cause issues. When designing a model it is important to try out different architectures. Our results provide some guidelines. Try to avoid using batch normalization, and use groupnorm instead. Be aware of ReLU functions. They do not cause problems within an ODEblock, but should be avoided before the block. Furthermore, adding stride before the ODEblock is not ideal as it increases the NFE. However, it also reduces the dimension and therefore the training time. The trade-off between training time and a higher NFE is dependent on the use case.

Additionally, a method is proposed to incorporate different input and output dimensions. This is done by padding the image width and height, and padding the channels. This results in a succesful model, that even has less NFE than the simpler counterpart. By allowing different input and output dimensions, the ODEblock can incorporate layers such as pooling and convolutions between different number of channels.

Altogether, the ODEnet has been demystified over the course of this project. It provides much more insight into what happens, and what to expect when using an ODEnet. Unfortunately, it did not prove a good alternative for the application used in this research. Nevertheless, the information gained can be used to further investigate the new model, and expose its full potential.

Bibliography

[1] Alexe, Mihai and Sandu, Adrian. “On the discrete adjoints of adaptive time stepping algorithms”. In: *Journal of Computational and Applied Mathematics* 233.4 (2009), pp. 1005–1020. DOI: `10.1016/j.cam.2009.08.109`.

[2] Bergstra, James, Yamins, Daniel, and Cox, David Daniel. “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures”. In: *Proc. of the 30th International Conference on Machine Learning* (2013). URL: `http://proceedings.mlr.press/v28/bergstra13.pdf`.

[3] Bradley, Andrew M. *PDE-constrained optimization and the adjoint method.* (2010). URL: `https://cs.stanford.edu/~ambrad/adjoint_tutorial.pdf`.

[4] Chang, Bo et al. “Multi-level Residual Networks from Dynamical Systems View”. In: *International Conference on Learning Representations.* (2018). arXiv: `1710.10348`.

[5] Chang, Bo et al. “Reversible Architectures for Arbitrarily Deep Residual Neural Networks”. In: *AAAI Conference on Artificial Intelligence.* (2018), pp. 2811–2818. URL: `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16517`.

[6] Chen, Ricky T. Q. et al. “Neural Ordinary Differential Equations”. In: *Advances in Neural Information Processing Systems 31.* Ed. by S. Bengio et al. Curran Associates, Inc., (2018), pp. 6571–6583. URL: `http://papers.nips.cc/paper/7892-neural-ordinary-differential-equations.pdf`.

[7] Dupont, Emilien, Doucet, Arnaud, and Teh, Yee Whye. “Augmented Neural ODEs”. In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., (2019), pp. 3140–3150. URL: `http://papers.nips.cc/paper/8577-augmented-neural-odes.pdf`.

[8] Forth, Shaun et al. *Recent Advances in Algorithmic Differentiation.* Springer Publishing Company, Incorporated, (2012). ISBN: 3642300227.

[9] Gholami, Amir, Keutzer, Kurt, and Biros, George. “ANODE: Unconditionally Accurate Memory-Efficient Gradients for Neural ODEs”. In: *2019 International Joint Conference on Artificial Intelligence* (2019). arXiv: `1902.10298`.

[10] Glorot, Xavier, Bordes, Antoine, and Bengio, Yoshua. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, (2011), pp. 315–323. URL: `http://proceedings.mlr.press/v15/glorot11a/glorot11a.pdf`.

[11] Gomez, Aidan N et al. “The Reversible Residual Network: Backpropagation Without Storing Activations”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., (2017), pp. 2214–2224. URL: `http://papers.nips.cc/paper/6816-the-reversible-residual-network-backpropagation-without-storing-activations.pdf`.

[12] Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. *Deep Learning*. MIT Press, (2016). URL: `http://www.deeplearningbook.org`.

[13] Grathwohl, Will et al. “FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models”. In: *International Conference on Learning Representations*. (2019). arXiv: `1810.01367`.

[14] Greff, Klaus et al. “The Sacred Infrastructure for Computational Research”. In: *Proceedings of the 16th Python in Science Conference*. Ed. by Katy Huff et al. (2017), pp. 49 –56. DOI: `10.25080/shinma-7f4c6e7-008`.

[15] Haber, Eldad and Ruthotto, Lars. “Stable architectures for deep neural networks”. In: *Inverse Problems* 34.1 (2017), p. 014004. DOI: `10.1088/1361-6420/aa9a90`.

[16] He, Kaiming and Sun, Jian. “Convolutional Neural Networks at Constrained Time Cost”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (2015). URL: `https://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/He_Convolutional_Neural_Networks_2015_CVPR_paper.pdf`.

[17] He, Kaiming et al. “Deep Residual Learning for Image Recognition”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (2016). arXiv: `1512.03385`.

[18] Hindmarsh, Alan C. and Serban, Radu. *User Documentation for cvodes v5.0.0 (sundials v5.0.0)*. English. Version v5.0.0. Center for Applied Scientific Computing Lawrence Livermore National Laboratory. 2019.

[19] Homescu, Cristian. “Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance”. (2011). arXiv: `1107.1831`.

[20] Hornik, Kurt. “Approximation capabilities of multilayer feedforward networks”. In: *Neural networks* 4.2 (1991), pp. 251–257. DOI: `10.1016/0893-6080(91)90009-T`.

[21] Hornik, Kurt, Stinchcombe, Maxwell, White, Halbert, et al. "Multilayer feedforward networks are universal approximators." In: *Neural networks* 2.5 (1989), pp. 359–366. DOI: 10.1016/0893-6080(91)90009-T.

[22] indoML. *Student Notes: Convolutional Neural Networks (CNN) Introduction.* [Online]. (2018). URL: https://indoml.com/2018/03/07/student-notes-convolutional-neural-networks-cnn-introduction/.

[23] Ioffe, Sergey and Szegedy, Christian. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning.* Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, (2015), pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.pdf.

[24] Jia, Junteng and Benson, Austin R. "Neural Jump Stochastic Differential Equations". In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., (2019), pp. 9847–9858. URL: http://papers.nips.cc/paper/9177-neural-jump-stochastic-differential-equations.pdf.

[25] Johnson, Matthew James. "Automatic Differentiation". [Online]. (2017). URL: http://videolectures.net/deeplearning2017_johnson_automatic_differentiation/.

[26] Johnson, Steven G. *Notes on Adjoint Methods for 18.335.* Introduction to Numerical Methods. MIT, (2006). URL: https://math.mit.edu/~stevenj/18.336/adjoint.pdf.

[27] Krizhevsky, Alex. "Learning multiple layers of features from tiny images". (2009). URL: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[28] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25.* Ed. by F. Pereira et al. Curran Associates, Inc., (2012), pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[29] Larsson, Gustav, Maire, Michael, and Shakhnarovich, Gregory. "FractalNet: Ultra-Deep Neural Networks without Residuals". In: *CoRR* abs/1605.07648 (2016). arXiv: 1605.07648.

[30] LeCun, Yann. "A theoretical framework for back-propagation". In: *Proceedings of the 1988 connectionist models summer school.* Ed. by D. Touretzky, G. Hinton, and T. Sejnowski. Vol. 1. CMU, Pittsburgh, Pa: Morgan Kaufmann. (1988), pp. 21–28. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-88.pdf.

[31] LeCun, Yann et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE.* Vol. 86. 11. (1998), pp. 2278–2324. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

[32] Leshno, Moshe et al. “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function”. In: *Neural networks* 6.6 (1993), pp. 861–867. DOI: 10.1016/S0893-6080(05)80131-5.

[33] Li, Sunner. *An Interesting Idea toward CNN — Residual.* [Online]. (2017). URL: https://medium.com/@sunnerli/an-interesting-idea-toward-cnn-residual-4bb54040b9a.

[34] Liu, Xuanqing et al. “Neural SDE: Stabilizing Neural ODE Networks with Stochastic Noise”. (2019). arXiv: 1906.02355.

[35] Lu, Yiping et al. “Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations”. In: *Proceedings of the 35th International Conference on Machine Learning.* Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, (2018), pp. 3276–3285. URL: http://proceedings.mlr.press/v80/lu18d/lu18d.pdf.

[36] Machine Learning @ Berkeley. *Neural Ordinary Differential Equations and Dynamics Models.* [Online]. (2019). URL: https://medium.com/@ml.at.berkeley/neural-ordinary-differential-equations-and-dynamics-models-1a4277fbb80.

[37] Peluchetti, Stefano and Favaro, Stefano. “Infinitely deep neural networks as diffusion processes.” (2019). arXiv: 1905.11065.

[38] Rackauckas, Christopher et al. “A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions”. (2018). arXiv: 1812.01892.

[39] Rackauckas, Christopher et al. “DiffEqFlux.jl - A Julia Library for Neural Differential Equations”. (2019). arXiv: 1902.02376.

[40] Rubanova, Yulia, Chen, Tian Qi, and Duvenaud, David K. “Latent Ordinary Differential Equations for Irregularly-Sampled Time Series”. In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., (2019), pp. 5320–5330. URL: http://papers.nips.cc/paper/8773-latent-ordinary-differential-equations-for-irregularly-sampled-time-series.pdf.

[41] Ruthotto, Lars and Haber, Eldad. “Deep Neural Networks motivated by Partial Differential Equations”. In: *Journal of Mathematical Imaging and Vision* abs/1804.04272 (2019). DOI: 10.1007/s10851-019-00903-1.

[42] Silver, David et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.

[43] Simonyan, Karen and Zisserman, Andrew. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. (2014). arXiv: 1409.1556.

[44] Sirkes, Ziv and Tziperman, Eli. “Finite Difference of Adjoint or Adjoint of Finite Difference?” In: *Monthly Weather Review* 125.12 (1997), pp. 3373–3378. DOI: 10.1175/1520-0493(1997)125<3373:FDOAOA>2.0.CO;2.

[45] Springenberg, J.T. et al. “Striving for Simplicity: The All Convolutional Net”. In: *ICLR (workshop track).* (2015). arXiv: 1412.6806.

[46] Subramanian, Vivek Ratnavel. *Omniboard.* (2018). URL: https://github.com/vivekratnavel/omniboard.

[47] Szegedy, Christian et al. “Going Deeper With Convolutions”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* (2015). arXiv: 1409.4842.

[48] Taigman, Yaniv et al. “Deepface: Closing the gap to human-level performance in face verification”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* (2014), pp. 1701–1708. URL: https://research.fb.com/publications/deepface-closing-the-gap-to-human-level-performance-in-face-verification/.

[49] Tzen, Belinda and Raginsky, Maxim. “Neural Stochastic Differential Equations: Deep Latent Gaussian Models in the Diffusion Limit”. (2019). arXiv: 1905.09883.

[50] Wang, Mei and Deng, Weihong. “Deep Face Recognition: A Survey”. (2018). arXiv: 1804.06655.

[51] Weinan, E. “A proposal on machine learning via dynamical systems”. In: *Communications in Mathematics and Statistics* 5.1 (2017), pp. 1–11. DOI: 10.1007/s40304-017-0103-z.

[52] Wu, Yuxin and He, Kaiming. “Group Normalization”. In: *The European Conference on Computer Vision (ECCV).* (2018). arXiv: 1803.08494.

[53] Xiao, Han, Rasul, Kashif, and Vollgraf, Roland. “Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms”. (2017). arXiv: 1708.07747.

[54] Zhang, Han et al. “Approximation capabilities of neural ordinary differential equations”. (2019). arXiv: 1907.12998.

[55] Zhang, Hong and Sandu, Adrian. “FATODE: A Library for Forward, Adjoint, and Tangent Linear Integration of ODEs”. In: *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C504–C523. DOI: 10.1137/130912335.

[56] Zhang, Tianjun et al. “ANODEV2: A Coupled Neural ODE Framework”. In: *Advances in Neural Information Processing Systems 32.* Ed. by H. Wallach et al. Curran Associates, Inc., (2019), pp. 5151–5161. URL: http://papers.nips.cc/paper/8758-anodev2-a-coupled-neural-ode-framework.pdf.

[57] Zhang, Xingcheng et al. “PolyNet: A Pursuit of Structural Diversity in Very Deep Networks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* (2017). arXiv: 1611.05725.

[58] Zill, Dennis G. *A first Course in Differential Equations with modeling applications*. Cengage Learning. ISBN: 78-1-111-82705-2.

APPENDIX A

Proofs

A.1 Proof of Lemma 4.1

Lemma 4.1 *Formulating the ODEnet as a Lagrange optimization problem*

$$\begin{aligned} min_\theta \quad & Loss = L(z(t_1)) \\ Subject\ to \quad & \dot{z}(t) = \frac{dz}{dt} = f(z(t), \theta, t) \end{aligned} \tag{4.15}$$

Corresponds to the Lagrangian:

$$\mathcal{L}(z(t), \theta, \lambda) = L\left(z(t_1)\right) - \int_{t_0}^{t_1} \lambda(t)(\dot{z}(t) - f(z(t), \theta, t))\, dt \tag{4.16}$$

Optimizing the Lagrangian leads to the following gradients of the Loss:

$$\nabla_\theta L = -\int_{t_1}^{t_0} \lambda(t) \frac{\partial f(z(t), \theta, t)}{\partial \theta}\, dt \tag{4.17}$$

$$\nabla_t L = -\int_{t_1}^{t_0} \lambda(t) \frac{\partial f(z(t), \theta, t)}{\partial t}\, dt + \lambda(t_1) \frac{\partial z(t_1)}{\partial t} \tag{4.18}$$

Proof To optimize $\mathcal{L}(z(t), \theta, \lambda)$, the gradients $\nabla_\theta \mathcal{L}$ and $\nabla_t \mathcal{L}$ need to be calculated and set to zero.

First calculate $\nabla_\theta \mathcal{L}$.

The first term $L(z(t_1))$ in (4.16) is not directly dependent on θ, thus its derivative is zero.

$$\nabla_\theta \mathcal{L} = -\int_{t_0}^{t_1} \lambda(t) \left(\frac{\partial \dot{z}}{\partial \theta} - \left(\frac{\partial f}{\partial \theta} + \frac{\partial f}{\partial z} \frac{\partial z}{\partial \theta} \right) \right) dt \tag{A.1}$$

Solving integration by parts for $\frac{\partial \dot{z}}{\partial \theta}$:

$$\begin{aligned}\int_{t_0}^{t_1} \lambda(t) \frac{\partial \dot{z}}{\partial \theta}\, dt &= \lambda(t) \frac{\partial z}{\partial \theta}\bigg|_{t_0}^{t_1} - \int_{t_0}^{t_1} \dot{\lambda}(t) \frac{\partial z}{\partial \theta}\, dt \\ &= \lambda(t_1) \frac{\partial z(t_1)}{\partial \theta} - \lambda(t_0) \frac{\partial z(t_0)}{\partial \theta} - \int_{t_0}^{t_1} \dot{\lambda}(t) \frac{\partial z}{\partial \theta}\, dt \\ &= \lambda(t_1) \frac{\partial z(t_1)}{\partial \theta} - \int_{t_0}^{t_1} \dot{\lambda}(t) \frac{\partial z}{\partial \theta}\, dt \end{aligned} \tag{A.2}$$

Plugging (A.2) back for $\frac{\partial \dot{z}}{\partial \theta}$ in the Equation (A.1)

$$\nabla_\theta \mathcal{L} = \int_{t_0}^{t_1} \lambda(t) \frac{\partial f}{\partial \theta} + \lambda(t) \frac{\partial f}{\partial z} \frac{\partial z}{\partial \theta} + \dot{\lambda}(t) \frac{\partial z}{\partial \theta}\, dt + \lambda(t_1) \frac{\partial z(t_1)}{\partial \theta} \tag{A.3}$$

Collect all terms with $\frac{\partial z}{\partial \theta}$:

$$\nabla_\theta \mathcal{L} = \int_{t_0}^{t_1} \lambda(t) \frac{\partial f}{\partial \theta} + \left(\lambda(t) \frac{\partial f}{\partial z} + \dot{\lambda}(t) \right) \frac{\partial z}{\partial \theta}\, dt + \lambda(t_1) \frac{\partial z(t_1)}{\partial \theta} \tag{A.4}$$

The value of $\lambda(t)$ can be freely set since the Lagrangian is constructed such that the second term will be zero when the constraints are satisfied [3]. We choose $\lambda(t)$ such that the following equation is satisfied:

$$\dot{\lambda}(t) = -\lambda(t) \frac{\partial f}{\partial z} \tag{A.5}$$

Then (A.5) in (A.4) gives:

$$\nabla_\theta \mathcal{L} = \int_{t_0}^{t_1} \lambda(t) \frac{\partial f}{\partial \theta}\, dt + \lambda(t_1) \frac{\partial z(t_1)}{\partial \theta} \tag{A.6}$$

Because $z(t_1)$ is defined by the constraints, it does not depend on θ, and the last term $\lambda(t_1)\frac{\partial z(t_1)}{\partial \theta} = 0$. Additionally, the weight parameters that minimize $\mathcal{L}$ are the same that minimize the

Loss:

$$\nabla_\theta \mathcal{L} = \nabla_\theta Loss(z) = -\int_{t_1}^{t_0} \lambda(t) \frac{\partial f}{\partial \theta} \, dt \tag{A.7}$$

This Equation A.7 is equivalent to Equation 51 of Chen *et al.*

Now, calculate $\nabla_t \mathcal{L}$

$$\nabla_t \mathcal{L} = -\int_{t_0}^{t_1} \lambda(t) \left(\frac{\partial \dot{z}}{\partial t} - \left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial z} \frac{\partial z}{\partial t} \right) \right) dt \tag{A.8}$$

Solving integration by parts for $\frac{\partial \dot{z}}{\partial t}$:

$$\begin{aligned}
\int_{t_0}^{t_1} \lambda(t) \frac{\partial \dot{z}}{\partial t} \, dt &= \lambda(t) \frac{\partial z}{\partial t} \Big|_{t_0}^{t_1} - \int_{t_0}^{t_1} \dot{\lambda(t)} \frac{\partial z}{\partial t} \, dt \\
&= \lambda(t_1) \frac{\partial z(t_1)}{\partial t} - \lambda(t_0) \frac{\partial z(t_0)}{\partial t} - \int_{t_0}^{t_1} \dot{\lambda}(t) \frac{\partial z}{\partial t} \, dt \\
&= \lambda(t_1) \frac{\partial z(t_1)}{\partial t} - \int_{t_0}^{t_1} \dot{\lambda}(t) \frac{\partial z}{\partial t} \, dt
\end{aligned} \tag{A.9}$$

Plugging (A.9) and (A.5) back into Equation (A.8):

$$\nabla_t \mathcal{L} = \nabla_t Loss(z) = -\int_{t_1}^{t_0} \lambda(t) \frac{\partial f}{\partial t} \, dt + \lambda(t_1) \frac{\partial z(t_1)}{\partial t} \tag{A.10}$$

Again, it can be recognized that Equation A.10 is equivalent to Equation 52 by Chen *et al.*

A.2 Proof of Lemma 4.3

Lemma 4.3 *The gradients of L with respect to θ, t and z can be calculated by solving one single ODE system backwards in time. Equation* (4.20) *is the initial value of this system, with $\frac{\partial L}{\partial \theta} = a_\theta$ and $\frac{\partial L}{\partial t} = a_t$, and Equation* (4.21) *the derivative with respect to time.*

$$\textit{Initial Value at } (t_1)\textit{:} \qquad \begin{bmatrix} \lambda(t_1) \\ a_\theta(t_1) \\ a_t(t_1) \end{bmatrix} = \begin{bmatrix} \nabla_z L(t_1) \\ 0 \\ \nabla_z L(t_1)\frac{\partial z(t_1)}{\partial t} \end{bmatrix} \tag{4.20}$$

$$\textit{Time derivative:} \qquad \begin{bmatrix} \dot{\lambda}(t) \\ \dot{a_\theta}(t) \\ \dot{a_t}(t) \end{bmatrix} = \begin{bmatrix} -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} \\ -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial \theta} \\ -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial t} \end{bmatrix} \tag{4.21}$$

Proof Define the adjoint state $\lambda(t) = \nabla_{z(t)} Loss$, similar as to the discrete case (Equation (4.11)). Now λ_{t_1} is the initial state used to calculate the other adjoints by solving an ODE backwards in time (Equation (A.11)). This results in the initial value problem:

$$\begin{aligned} &\dot{\lambda}(t) = -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial z} \quad \text{(See Equation (A.5))} \\ &\lambda(t_1) = \nabla_z L(t_1) \end{aligned} \tag{A.11}$$

Additionally, Equation (A.6) can be recognized as the solution to an ODE that is solved backward in time. Using the same notation as Chen *et al.*, set $a_\theta = \frac{dL}{d\theta} = \lambda(t)\frac{dz}{d\theta}$. Reformulating the equation in terms of an ODE:

$$\nabla_\theta Loss(z) = a_\theta(t_0) = \underbrace{\lambda(t_1)\frac{\partial z(t_1)}{\partial \theta}}_{\text{Initial value } a_\theta(t_1)} - \int_{t_1}^{t_0} \underbrace{\lambda(t)\frac{\partial f}{\partial \theta}}_{\frac{da_\theta}{dt}} \, dt \tag{A.12}$$

$$\begin{aligned} &\frac{da_\theta}{dt} = \dot{a}_\theta(t) = -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial \theta} \\ &a_\theta(t_1) = 0 \end{aligned} \tag{A.13}$$

Similar as the previous case, we can recognize Equation (A.10) as the solution to an ODE. Set $a_t = \frac{dL}{dt} = \lambda(t)\frac{dz}{dt}$, then:

$$\nabla_t Loss(z) = a_t(t_0) = \underbrace{\lambda(t_1)\frac{\partial z(t_1)}{\partial t}}_{\text{Initial value } a_t(t_1)} - \int_{t_1}^{t_0} \underbrace{\lambda(t)\frac{\partial f}{\partial t}}_{\frac{da_t}{dt}} \, dt \tag{A.14}$$

$$\begin{aligned}\frac{da_t}{dt} &= \dot{a}_t(t) = -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial t} \\ a_t(t_1) &= \lambda(t_1)\frac{\partial z(t_1)}{\partial t}\end{aligned} \tag{A.15}$$

These results align with the results from Chen *et al.*, and all gradients can be calculated by solving an ODE system. The gradients with respect to t and θ are also dependent on the current state of $\lambda(t)$. The trajectory of λ can be computed simultaneously with a_θ and a_t, by extending the original ODE system with all gradients (Equation (A.16)) [1, 38, 55]. All the gradients are calculated by this system, which can now be solved with a single call to the ODEsolver.

$$\begin{aligned}\text{Initial Value at } (t_1)\text{:} \quad & \begin{bmatrix}\lambda(t_1) \\ a_\theta(t_1) \\ a_t(t_1)\end{bmatrix} = \begin{bmatrix}\nabla_z L(t_1) \\ 0 \\ \nabla_z L(t_1)\frac{\partial z(t_1)}{\partial t}\end{bmatrix} \\ \text{Time derivative:} \quad & \begin{bmatrix}\dot{\lambda}(t) \\ \dot{a_\theta}(t) \\ \dot{a_t}(t)\end{bmatrix} = \begin{bmatrix}-\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial z(t)} \\ -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial \theta} \\ -\lambda(t)\frac{\partial f(z(t),t,\theta)}{\partial t}\end{bmatrix}\end{aligned} \tag{A.16}$$

APPENDIX B

Implementation Details

B.1 AWS Hardware

All experiments are run on AWS EC2 instance, a virtual computing environment. This EC2 instance is linked with an S3 bucket, where all data and experiment logs are stored. The EC2 instance that was used is a p2.xlarge. This instance has a single GPU[1] with 12 GiB memory, 4 CPU's and a total memory of 61GiB.

B.2 Packages and Libraries

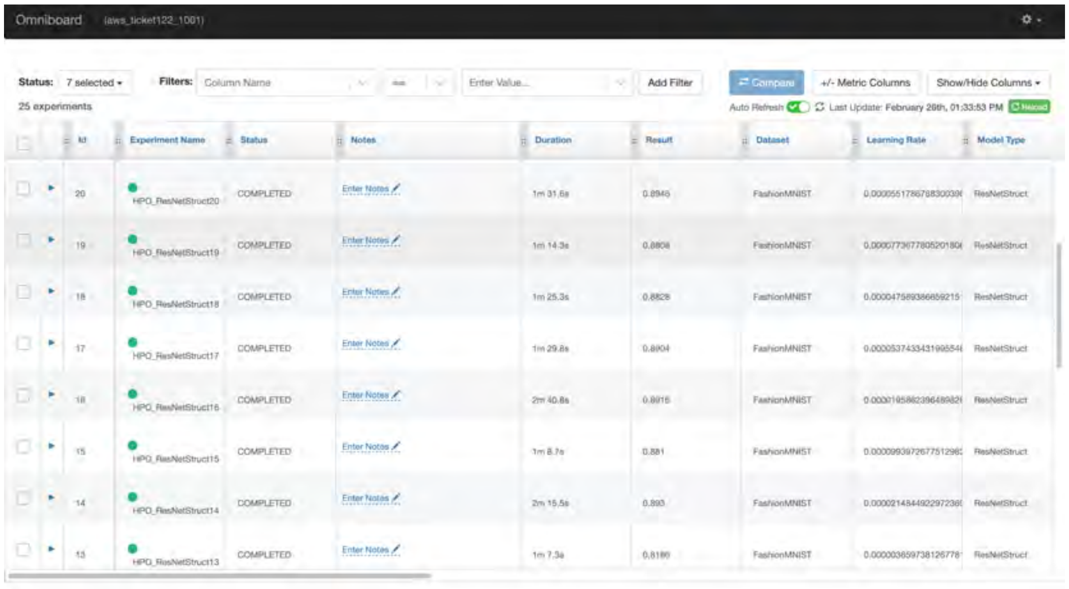
library	version
numpy	1.17.2
torch	0.4.0 (torchvision)
torchdiffeq	-
matplotlib	3.1.1
sacred	0.7.5
sklearn	0.21.2
scipy	1.3.1
seaborn	0.9.0
pandas	0.25.1
hyperopt	0.2.1

Table B.1: List of python libraries used an their version.

[1] High-performance NVIDIA K80 GPU

B.3 Experiment Logging

These are screenshots of the experiment logging created with *Sacred*, *Omniboard* and *MongoDB*.



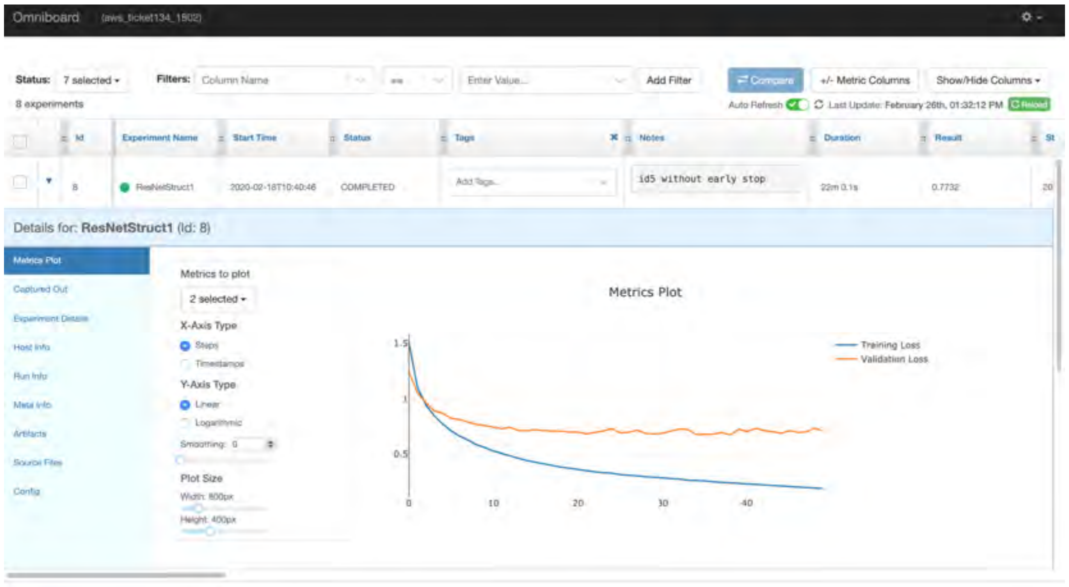
Status: 7 selected Filters: Column Name == Enter Value...

25 experiments

Auto Refresh Last Update: February 26th, 01:33:53 PM

Id	Experiment Name	Status	Notes	Duration	Result	Dataset	Learning Rate	Model Type
20	HPO_ResNetStruct20	COMPLETED	Enter Notes	1m 31.6s	0.8946	FashionMNIST	0.0000551786788300	ResNetStruct
19	HPO_ResNetStruct19	COMPLETED	Enter Notes	1m 14.3s	0.8808	FashionMNIST	0.000077367780520180	ResNetStruct
18	HPO_ResNetStruct18	COMPLETED	Enter Notes	1m 25.3s	0.8828	FashionMNIST	0.00004758938665921	ResNetStruct
17	HPO_ResNetStruct17	COMPLETED	Enter Notes	1m 29.8s	0.8904	FashionMNIST	0.000053743343199554	ResNetStruct
16	HPO_ResNetStruct16	COMPLETED	Enter Notes	2m 40.8s	0.8916	FashionMNIST	0.000195882396489826	ResNetStruct
15	HPO_ResNetStruct15	COMPLETED	Enter Notes	1m 8.7s	0.881	FashionMNIST	0.0000993972677512982	ResNetStruct
14	HPO_ResNetStruct14	COMPLETED	Enter Notes	2m 15.5s	0.893	FashionMNIST	0.0000214844922972366	ResNetStruct
13	HPO_ResNetStruct13	COMPLETED	Enter Notes	1m 7.3s	0.8186	FashionMNIST	0.00000365973812677	ResNetStruct

Figure B.1: An example of Omniboard with multiple experiment runs.



Status: 7 selected Filters: Column Name == Enter Value...

8 experiments

Auto Refresh Last Update: February 26th, 01:32:12 PM

Id	Experiment Name	Start Time	Status	Tags	Notes	Duration	Result	St
8	ResNetStruct1	2020-02-18T10:40:46	COMPLETED	Add Tags...	id5 without early stop	22m 0.1s	0.7732	20

Details for: **ResNetStruct1** (Id: 8)

Metrics to plot: 2 selected

X-Axis Type: [✓ Steps] Timestamps

Y-Axis Type: [✓ Linear] Logarithmic

Smoothing: 0

Plot Size

Width: 800px

Height: 400px

Figure B.2: Details of a single experiment run, and the visualization that is available during training.

APPENDIX C

Network Architecture

C.1 Simple ODE net

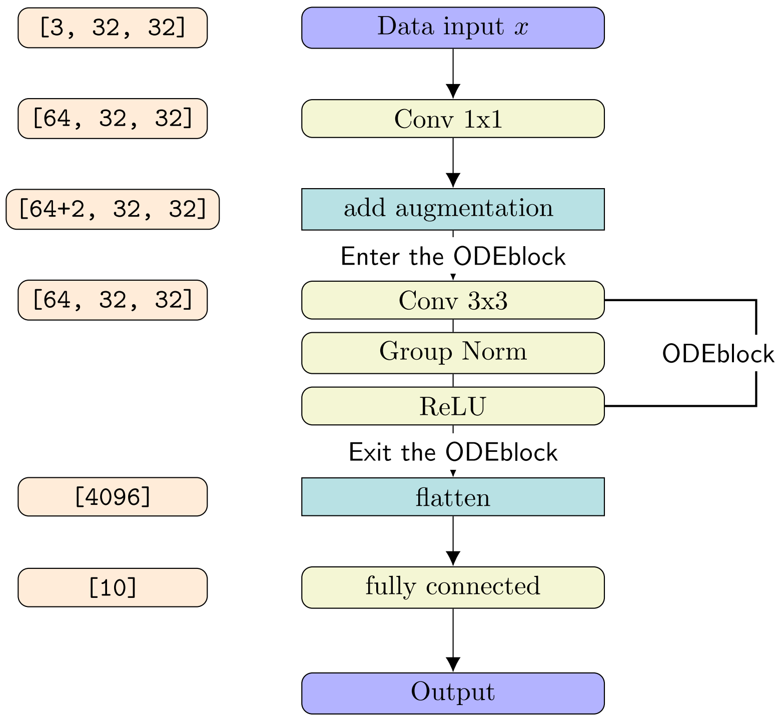


Figure C.1: Simple ODEnet structure from Chapter 7, including the output dimensions of every hidden layer.

C.2 Simple Small ResNet

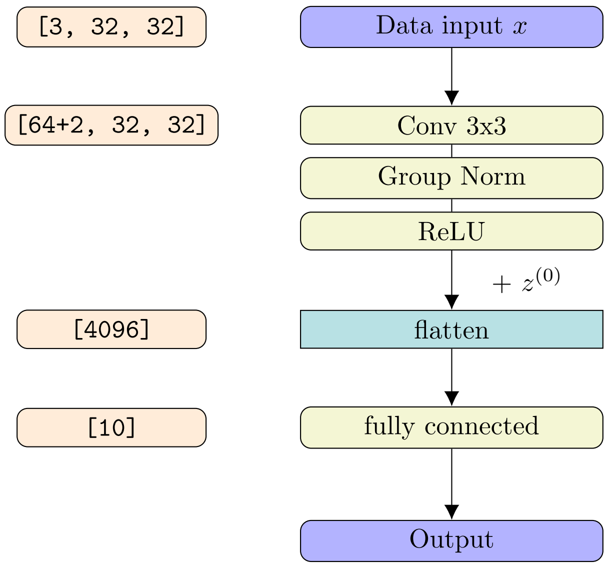


Figure C.2: Small Simple ResNet structure from Chapter 7, including the output dimensions of every hidden layer.

C.3 Simple Large ResNet

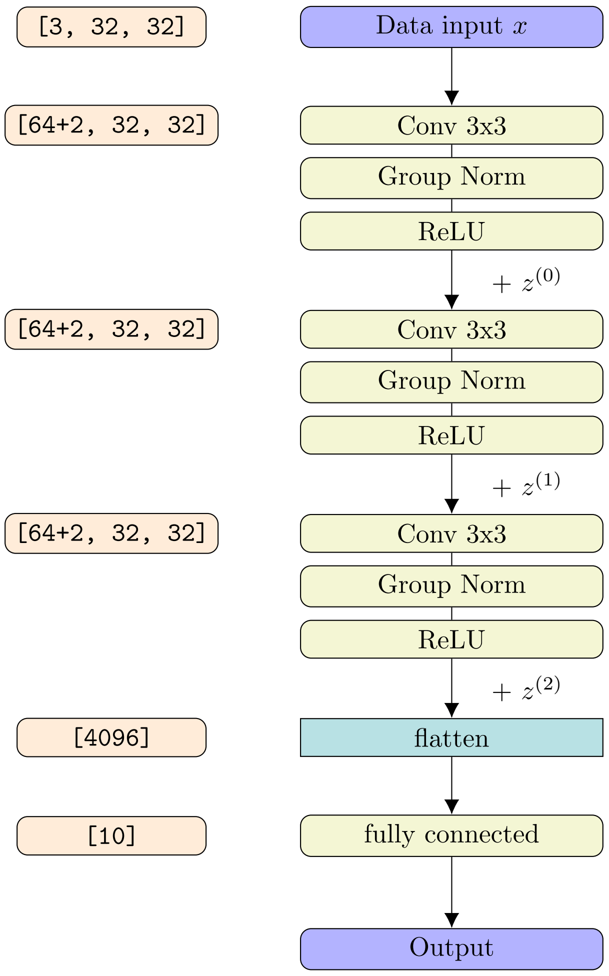


Figure C.3: Large Simple ResNet structure from Chapter 7, including the output dimensions of every hidden layer.

C.4 ODE Complex Network

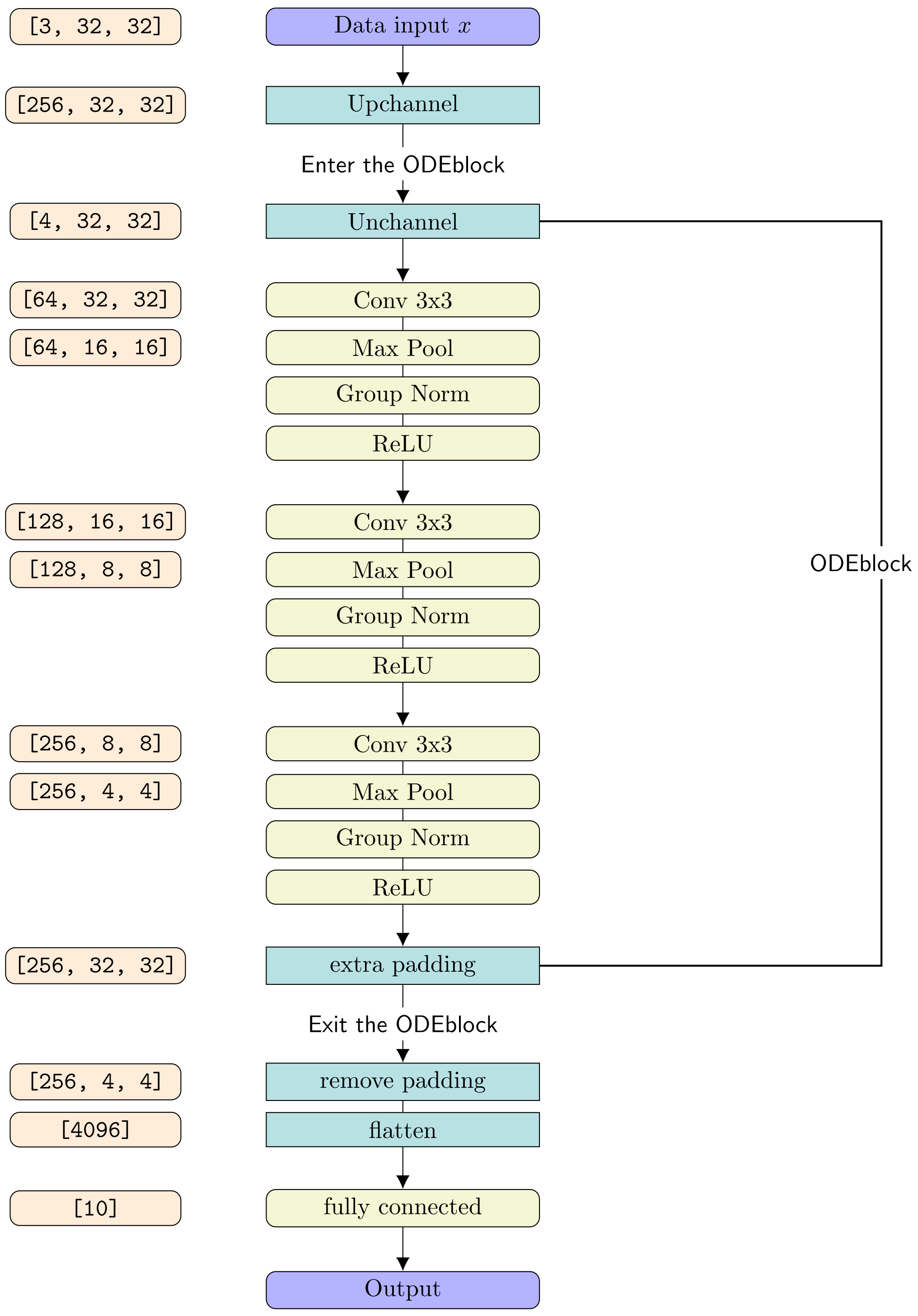


Figure C.4: Complex ODEnet structure from Chapter 7, including the output dimensions of every hidden layer.

C.5 ResNet Complex Network

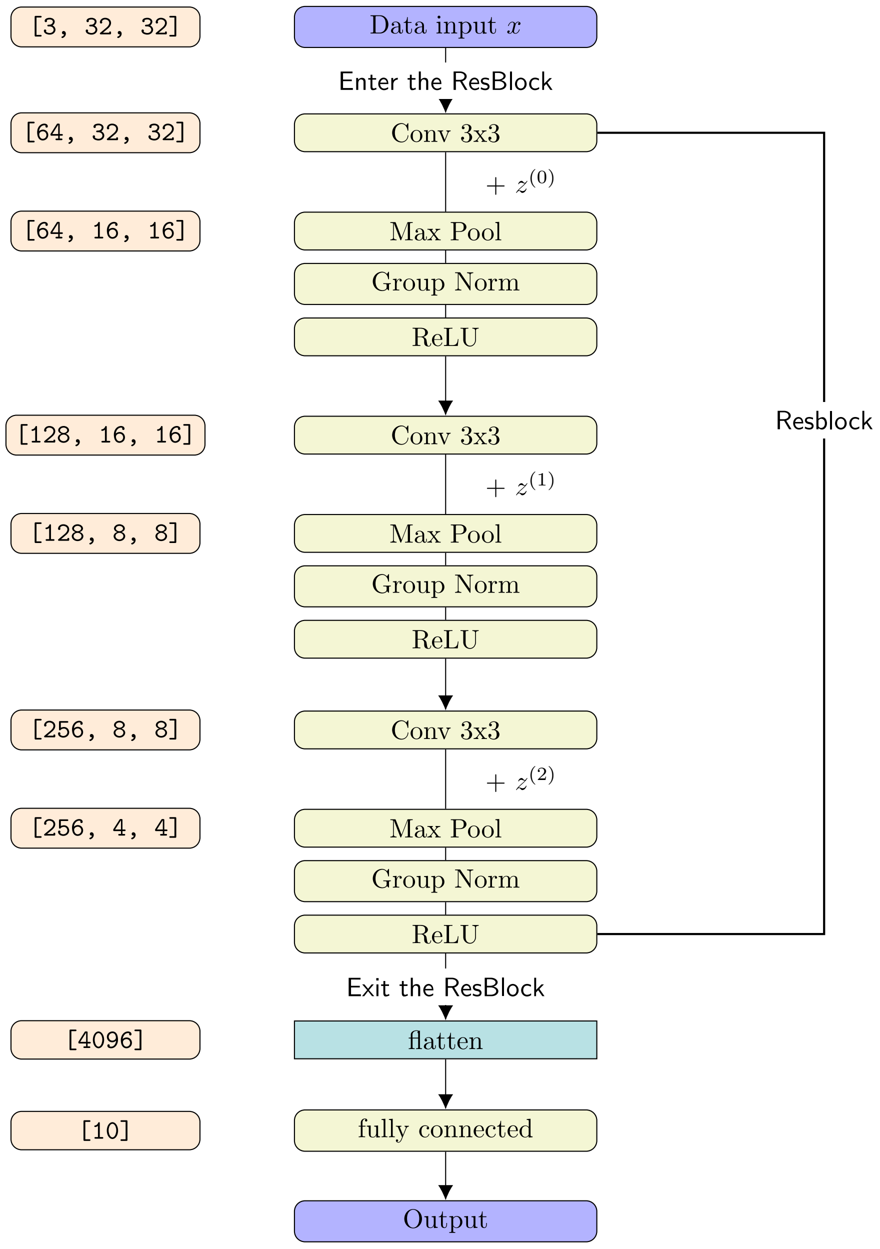


Figure C.5: Complex ResNet structure from Chapter 7, including the output dimensions of every hidden layer.

APPENDIX D

Results

D.1 FashionMNIST Simple Architectures

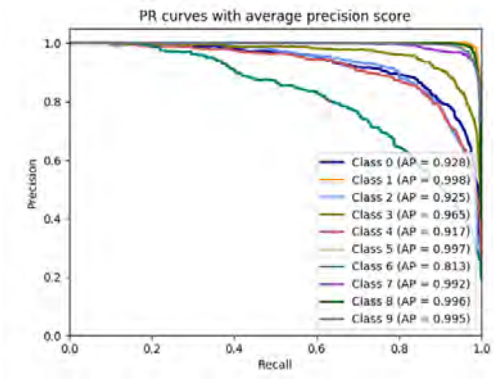


Figure D.1: Average Precision of Precision-Recall curves.

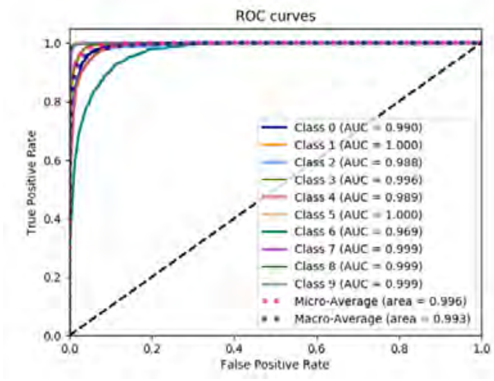
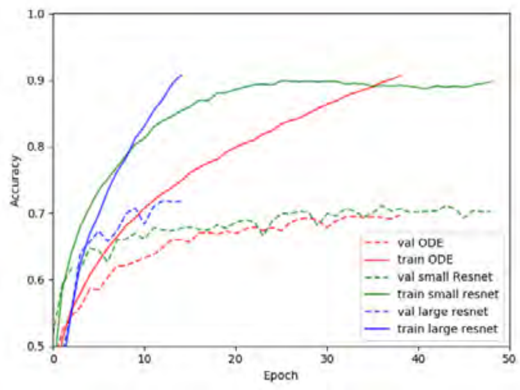
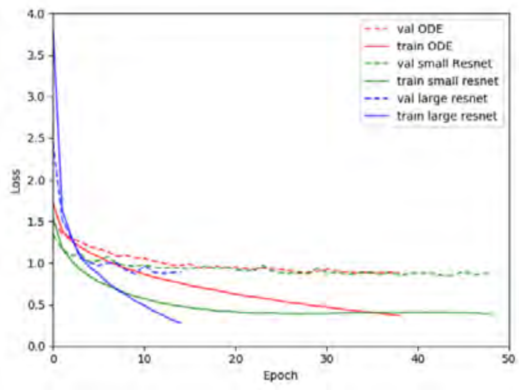


Figure D.2: Area under Curve of Receiver Operating Characteristics.

D.2 Cifar10 Simple Architectures



(a) Accuracy during training Cifar10 with simple architectures.



(b) Loss during Training Cifar10 with simple architectures.

Figure D.3: Accuracy and Loss for the ODEnet, small and large ResNet.

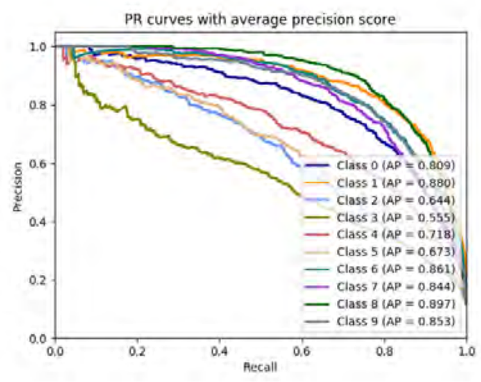


Figure D.4: Average Precision of Precision-Recall curves.

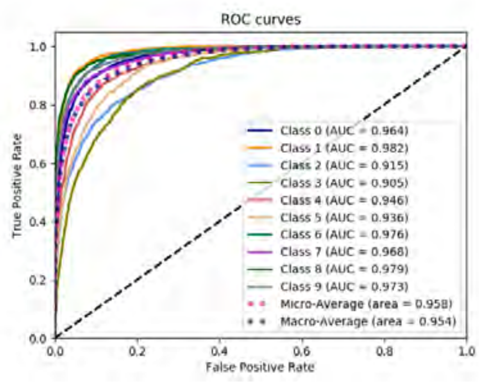


Figure D.5: Area under Curve of Receiver Operating Characteristics.

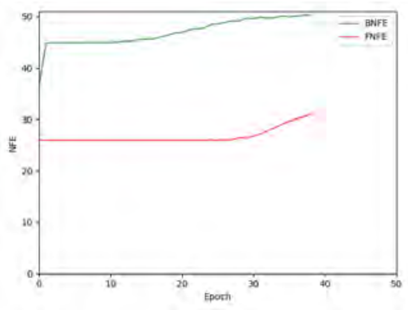


Figure D.6: NFE per epoch on Cifar10 with a simple ODEnet architecture.

D.3 FashionMNIST Complex Architectures

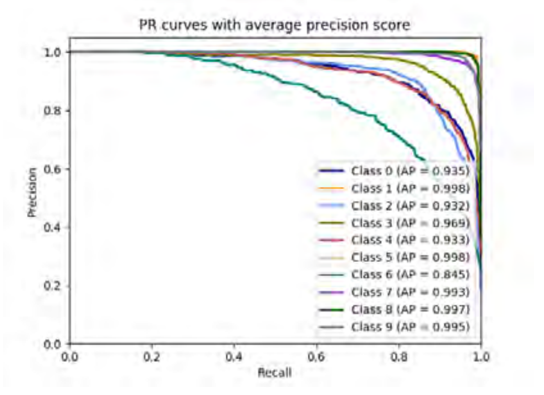


Figure D.7: Average Precision of Precision-Recall curves.

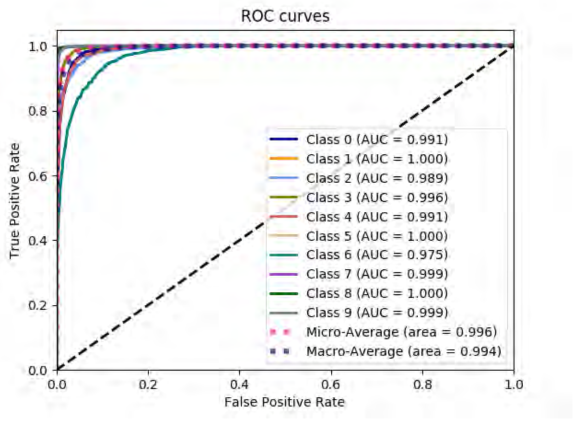


Figure D.8: Area under Curve of Receiver Operating Characteristics.

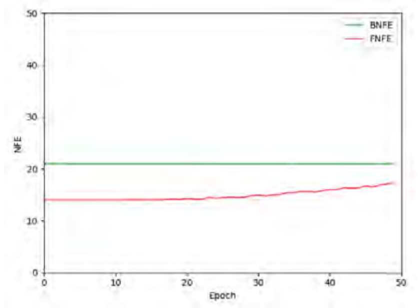
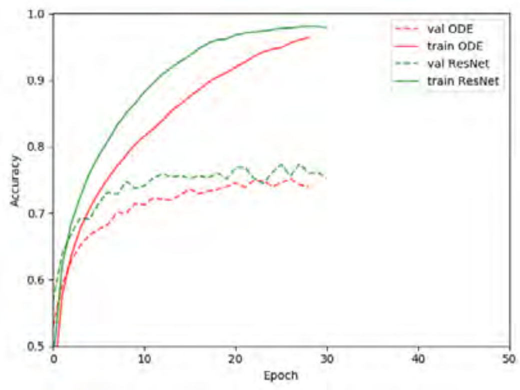
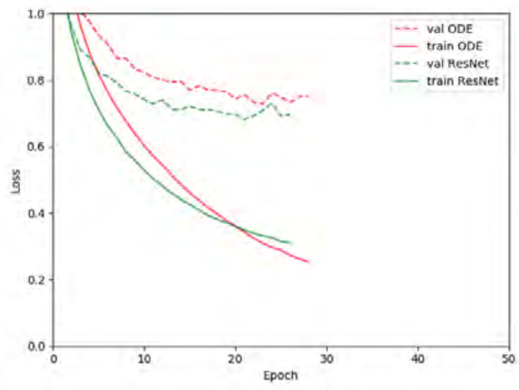


Figure D.9: NFE per epoch on FashionMNIST with a complex ODEnet architecture.

D.4 Cifar10 Complex Architectures



(a) Accuracy during training Cifar10 with complex architectures.



(b) Loss during training Cifar10 with complex architectures.

Figure D.10: Accuracy and Loss for the ODEnet, small and large ResNet.

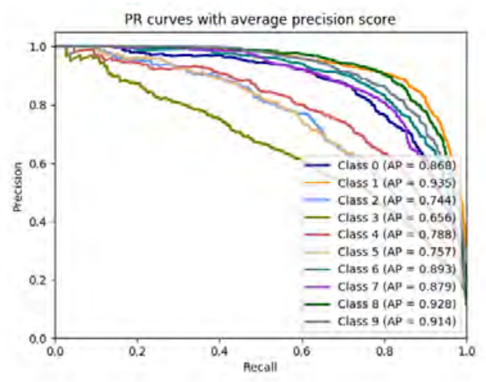


Figure D.11: Average Precision of Precision-Recall curves.

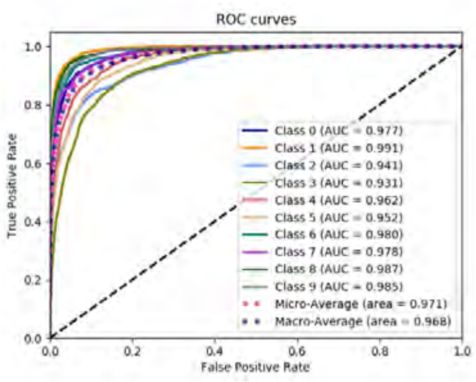


Figure D.12: Area under Curve of Receiver Operating Characteristics.