Extending Differentiable Programming to include Non-differentiable Modules using Differentiable Bypass for Combining Convolutional Neural Networks and Dynamic Programming into an End-to-end Trainable Framework

by

Nhat Minh Nguyen

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Differentiable Programming is the paradigm where different functions or modules are combined into a unified pipeline with the purpose of applying end-to-end learning or optimization. A natural impediment is the non-differentiability characteristic of many modules. This thesis proposes a new way to overcome this obstacle by using a concept called Differentiable Bypass (DiffBypass). DiffBypass exploits the Universal Function Approximation property of neural networks to mimic the output of non-differentiable functions or modules in the pipeline, rerouting the gradient path to bypass these components entirely.

Further, as a significant application, we demonstrate the use of DiffBypass to combine Convolutional Neural Networks (CNN) and Dynamic Programming (DP) in end-to-end learning for segmenting left ventricle from short axis view of heart MRI. Our experiments show that end-to-end combination of CNN and DP requires fewer labeled images to achieve a significantly better segmentation accuracy than using only CNN by allowing the incorporation of strong prior knowledge into the pipeline to cope with lack of training data. Comparison between DiffBypass and Evolution Strategy (ES), another method that can be used to train non-differentiable modules, shows that DiffBypass is more robust and has better performance for high-dimension problems.

Finally, as a technical contribution, we provide a set of recommendations for training non-differentiable modules using DiffBypass. Furthermore, we also provide a code base for reproducibility. We think DiffBypass has the potential to become a blueprint to expand differentiable programming to include non-

differentiable modules.

Preface

Parts of this thesis have been submitted to The 26th biennial international conference on Information Processing in Medical Imaging (IPMI 2019).

To my mother for raising me and teaching me right from wrong.

My main interest is in trying to find radically different kinds of neural nets.

– Geoffrey Hinton.

Acknowledgements

I would like to express my gratitude my supervisor Nilanjan Ray for his support and guidance. Also, thank to the Mitacs Canada organization and the Canada government for giving me this opportunity as well as providing me an entrance scholarship.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Recently, many challenging problems in Artificial Intelligence have been conquered by Deep Learning, which is a rebranding of Neural Networks. It turns out that learning statistical patterns from a large amount of data is much more effective at solving problems than crafting fine-tuned features or rules. Deep Learning have achieved super-human performance in many areas, from detecting objects in images [37] to playing games [43]. At the core of Deep Learning is a decades-old timeless classic, the backpropagation algorithm, which is an elegant and clever way of applying the chain rule to compute the gradients of the parameters in neural networks effectively. The backpropagation algorithm lends its magic to the nature of neural networks architectures: they can usually be expressed as directed acyclic graph. This kind of structure allow us to go in the backward direction of the graph, accumulating and updating the gradients along the way. The gradients are then used by optimization techniques, most popular one being Stochastic Gradient Descent, to directly find a better solution to the problem.

However, backpropagation is not the key to successes of modern Deep Learning as traditional neural networks use the same algorithm at their core. Take Computer Vision (CV) for example, CV borrows most of its recent success from Convolutional Neural Networks, which is a special type of neural networks that share their weights across all spatial location in images. Many modern Natural Language Processing (NLP) algorithm use Recurrent Neural

Networks, a type of neural networks that share their weight across the time dimension. These success stories have one thing in common, that is the idea of using the same components of the neural networks in multiple places. This type of weight sharing force the neural network components to adapt to multiple type of input at different situation, not just a single point place or time, thereby making the model more robust and generalize better.

The idea that one component or module should be reused across places is not new, as it a common theme in computer science, for example, programming language. In fact, there is many common similarities between neural networks architecture and functions in programming that it is not a stretch to think that designing neural network architecture as a special, different form of "programming" where users create and put together blocks into a system that can be differentiated so that users can train the system end-to-end using some kind of gradient optimization. Consequently, it is natural to bring concepts from programming into neural network architecture design. One of the most interesting and important one is that the structure of the program should change accordingly to each situation. That means the input into the program should affect which functions are used and how many time they get called. This translates to allowing users to have conditionals and loops in their neural network architecture design. This is expected to be the next step in Differentiable Programming. Dynamical neural networks architecture that change depending on their input have become more and more popular in the past few years, especially after the inception of deep learning frameworks like Pytorch [32] and Chainer [47]. An increasingly large number of deep learning practitioners have picked up this dynamical design paradigm and there are active works on compilers for imperative differentiable programming languages [51]. In fact, Pytorch, a deep learning framework that use Automatic Differentiation (AD), a concept that Differentiable Programming relies on, has become one of the main contenders for being the most dominant framework used in Deep Learning.

Despite the successes and wide-spread use of Differentiable Programming, one of the paradigm's main weakness currently is that it doesn't allow the use of non-differentiable modules within the pipeline. Everything has to be differentiable from beginning to end for automatic differentiation to work. This limit the classes of blocks or components that can be used in a pipeline to ones that has clearly defined close form expression for their derivatives such as linear matrix operations as well as activation function like ReLu, Sigmoid... These kinds of differentiable modules work very well for most modern problems in artificial intelligence. However, they are not without downsides. First, the modules are not usually interpretable and this impedes the use of neural networks in safety-critical domains like health care or control system of important machinery. Second, it is not trivial to incorporate expert knowledge, which can be used to improve sample efficiency as well as final performance, into these modules. Moreover, combining modern neural networks architecture with domain-specific algorithms is hard because these algorithm usually are not differentiable therefore cannot be put into an end-to-end training system.

The aforementioned downside of current Differentiable Programming paradigm calls for its extension into non-differentiable modules that includes traditional domain-specific algorithms as well as algorithms that allow the incoporation of prior knowledge into the end-to-end pipeline. This thesis presents a blueprint for such an extension. The main idea presented is to create something called "Differentiable Bypass" for the non-differentiable components. A Differentiable Bypass is a module that is differentiable and approximates the output of a non-differentiable one. During training, instead of having broken gradient backward pass for the non-differentiable components, the gradient flow are rerouted to their own Differentiable Bypasses, creating a new, non-broken gradient flow that can be used for gradient optimization algorithms like backpropagation. A requirement of differentiable bypasses is that they need to be able to approximate their corresponding non-differentiable component well enough for the gradient to be correct. To sastify this requirement, we resort to using neural networks as our *de facto* implementation of the differentiable bypasses as neural networks are known for their Universal Approximation property.

1.2 Contributions

The contributions of this thesis is as follow:

- We present a novel way to combine *non-differentiable* functions with Deep Learning within an end-to-end framework. This is an extension of Differentiable Programming, which has been applied only to differentiable functions so far. In this context, we demonstrate the use of the Universal Approximation property of neural networks by something call Differentiable Bypass (DiffBypass) for a *non-differentiable* function. A similar technique, called Synthetic Gradient, has been used before for fast and asynchronous training of differentiable functions [22].

- As a demonstration of the aformentioned differentiable bypass strategy, we propose a combination of deep learning and a traditional method, namely Dynamic Programming (DP), using with strong prior knowledge that can compensate for the inadequate amount of training data significantly. Our combined method is applied to the context of medical imaging domain, specially for the problem of segmenting left ventricle of the heart in MRI images. [6]

1.3 Thesis Outline

The next sections of the thesis is partitioned as follow: Chapter 2 introduces the background of Differentiable Programming and Neural Networks. Chapter 3 present a number of method to combine differentiable and non-differentiable module, including the concept of Differentiable Bypass, which is the main contribution of this thesis. Chapter 4 provides some background on combining and incoporating prior knowledge into medical imagining and why this is an important research topic, then presents our method of combining Convolutional Neural Networks and Dynamic Programming into an end-to-end trainable pipeline along with the experiments to show its effectiveness compared to using Convolutional Neural Networks alone. Chapter 5 contains the experiments result of our method mentioned in the previous chapter. Chapter 6 provides discussion and some future directions for our research.

Chapter 2

Background

2.1 Differentiable Programming

Differentiable Programming is a term coined by Yann Lecun [51] as a re-branding of Deep Learning. Yet, Differentible Programming is more than just Deep Learning. The main idea behind Differentiable Programming is to extend the neural network layers to any type of function blocks that are parameterized and can be trained end-to-end using some form of gradient-based optimization. The paradigm includes building neural networks in an adaptive, data-dependent way with loops and conditionals. This allow users to specify neural network architectures that change dynamically as a function of the input data fed into them. The concept expand traditional neural network layers to programs that can be parameterized and optimized.

Under the hood, Differentible Programming relies on techniques belong to the family of general-purpose Automatic Differentiation. The benefits of applying Automatic Differentiation in machine learning are plentiful. Long gone the days researchers have to manually derive analytical derivatives of new machine learning models to plug them into optmization algorithms. Automatic Differentiation framework like Pytorch [32], Tensorflow [30] allows user to define and train neural network architectures in a convenient way. Automatic Differentiation have become ubiquitous in the Machine Learning field.

The methods for computing the automatically derivatives in programs can be classified into four categories as follow [6]: (1) no automation at all, manual differentiation; (2) numerical differentiation using finite difference approximations; (3) symbolic differentiation and (4) automatic differentiation. Automatic Differentiation (4) is the natural progression of the other categories. The next four subsections described what each methods is and the strength and/or weaknesses of each methods.

2.1.1 Manual differentiation

Manual differentiation happens when users provide both the code to compute the output of a code block as well as the code to compute the derivative of it with respect to the input. Some example of work that do manual differentiation include [16], [36]. Because manual differentiation requires human in the loop, it is prone to errors. Moreover, the derivative might not have a closed form that can be derived manually or analytically. In some cases, the derivative might have a closed form but is impossible to for users to write code to compute because of its complexity.

Manual differentiation is not without an upside though. As the derivatives are manually derived by humans, they can be analyzed and optimized for faster computation in some cases.

2.1.2 Numerical Differentiation

Numerical differentiation involves using the finite difference approximation of derivatives using values of the function evaluated at some sample points [6]. For example, let f be a multivariate function $f : \mathbb{R} \to \mathbb{R}$, the approximate of the gradient $\nabla f = (\frac{\delta f}{\delta x_1}, ..., \frac{\delta f}{\delta x_n})$ can be computed using the limit definition of a derivative as follow:

$$\frac{\delta f(x)}{\delta x_i} \approx \frac{f(x + he_i) - f(x)}{h} \tag{2.1}$$

where e_i is the $i - th$ unit vector and h is a small positive step size.

Numerical differentiation is simple to implement but its main disadvantage is that the computation requires $O(n)$ evaluation of f for a gradient of a n dimension input as well as a good step size h.

Numerical approximation of derivatives are ill-conditioned and unstable due to the limited precision of computer systems, such as truncation and round-off errors, especially when $h \to 0$ [6]. Various techniques have been introduced to alleviate the approximation errors in numerical differentiation. For example, center difference approximation can be used to reduce the truncation errors down to the second order in h:

$$\frac{\delta f(x)}{\delta x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h} + O(h^2) \tag{2.2}$$

However, these methods are often not as effective when h is very small and usually increase computational complexity by a large margin.

The main road block to allow the use of numerical differentiation in Machine Learning is still the $O(n)$ complexity. For modern Deep Learning architectures, the number of parameters n can be in the range millions, making numerical differentiation infeasible to use. On the other hand, approximation errors can be tolerated because neural networks can still function almost as good in low-precision system [17].

2.1.3 Symbolic differentiation

Symbolic differentiation is another paradigm where expressions of derivative are obtained by expanding and manipulating expressions progressively using differentiation rules [6]. For example:

$$\begin{aligned} \frac{d}{dx}(f(x) + g(x)) &\to \frac{d}{dx}f(x) + \frac{d}{dx}g(x) \\ \frac{d}{dx}(f(x)g(x)) &\to (\frac{d}{dx}f(x))g(x) + f(x)(\frac{d}{dx}g(x)) \end{aligned} \tag{2.3}$$

Symbolic differentiation of a formula's expression tree is a purely mechanical process [6] that can be implemented effectively. Symbolic differentiation address the weak points of manual differentiation as well as numerical differentiation and it have found wide-spread successes in the past. Example of symbolic differentiation can be found in software system like Mathametica, Maple and some deep learning frameworks like Theano [46].

One strength of symbolic differentiation is that symbolic derivatives can give valuable insight into the structure of the problem and in some cases can produce analytical solution that are trivial compute. However, in general, symbolic derivatives are not known for being efficient for runtime calculation of derivative values, as they can get exponentially larger than the expression whole derivative they represent [6].

As an example, consider a function $h(x) = f(x)g(x)$. Using the multiplicative rule for differentiation we get $\frac{d}{dx}h(x) \rightarrow (\frac{d}{dx}f(x))g(x) + f(x)(\frac{d}{dx}g(x))$. If we just blindly proceed to symbolically differentiate $f(x)$ and plug its derivatives into the previous expression, we could create a lot of duplications. The reason is that $f(x)$ and $\frac{d}{dx}f(x)$ usually have a lot of common component to compute. If we don't take this into account when expanding the expression, we could have nested duplications of any computation that appears in common between $f(x)$ and $\frac{d}{dx}f(x)$. This can lead to exponentially large symbolic expression that take very long time to evaluate during execution time. This is the reason why Deep Learning frameworks that use symbolic differentiation like Theano have an additional, optional step of optimize the computational graph of the symbolic derivative before execution. However, these kind of optimization are not perfect and we usually still end up with large cryptic symbolic expressions with many redundancies.

2.1.4 Automatic Differentiation

In modern Deep Learning, for training neural networks, we usually care less about the symbolic form of derivative and more about the numerical accuracy. It is much better to store only the values of intermediate sub-expression in memory. We can futher improve the effeciency by interleaving as much as possible the differentiation and simplification step. This is the basic idea of Automatic Differentiation: *apply symbolic differentiation at the elmentary operation level and keep intermediate numerical results, in lockstep with the evaluation of the main function* [6].

Automatic Differentiation can be thought as performing additional computation of various derivative on top of executing standard computer program. In a system where all component are differentiable, such as neural networks, all numerical computation of derivatives can be reduced into composition of finite set of elementary operations whose derivative are known [50]. An important point that need to be made here is that, unlike Symbolic Differentiation, Automatic Differentiation can handle control flow such as branching, loop, recursion and procedure call. This is due to the fact that any numeric code will eventually result in a numeric evaluation trace that can be used to compute the derivatives, regardless of any control flow path that was taken during execution. In another way, Automatic Differentiation is blind with respect to any operation, including control flow statements because they do not directly alter numeric values that are used for computing the derivative [6]. Compare this to Symbolic Differentiation, which require having the symbolic expression of the derivative before execution for computation efficiency. One can also rebuild the symbolic expression every time there are change in control flow. However, this is prohibitively expensive.

There are two step in Automatic Differentiation: forward mode and reverse mode. Forward mode in Automatic Differentiation is analogous to the forward pass in Deep Learning. In forward mode, Automatic Differentiation execute the program and build up all the numerical value required for computing the derivative in reverse mode in a book-keeping procedure. The stored information includes the Jacobian of each function blocks. Additionally, the input and output of each function blocks are also usually stored. Reverse mode in Automatic Differentiation is the step where we go backward and propagate the derivative from a given output. Reverse mode in Automatic Differentiation is related to the backpropagation algorithm. In fact, it is a generalization of the latter. The details on how to implement the forward and reverse mode of Automatic Differentiation is not in the scope of this thesis. However, a very good explanation can be found in the survey paper by Baydin *et al.* [6].

Automatic Differentiation and consequently Differentiable Programming are the newest development in the toolbox of computing derivative for gradient optimization in Machine Learning. It addressed the issue with previous paradigms that have been used by the Machine Learning community over the years. The end result of Automatic Differentiation in a Machine Learning model is to obtain the dervative of all the parameters model with respect to the loss function. However, Automatic Differentiation does not work when some of the components in the end-to-end neural network architecture are non-differentiable. For this reason, we propose a mechanism, called "Differentiable Pass", to overcome this challenge in the next chapter. Differentiable Pass uses use the Universal Approximation property of Neural Networks. The next sections in this chapter provide an overview of Neural Networks as well as their Universal Approximation property.

2.2 Neural Network overview

2.2.1 Multilayer Perceptron

Neural Networks are very powerful machine learning model that are loosely inspired by the biological structure of the brain. One of the simplest form of Neural Networks is a Multilayer Perceptron. The mathematical formulation of a Multilayer Perceptron is shown below.

A Multilayer Perceptron consists of N layers f_i for $i : 1 \to N$ connected together sequentially. Each layer receive an input x_i, which is a column vector of l_i dimension. x_i is then put through a linear transformation using a weight matrix W_i and a bias vector b_i. This linear transformation is followed immediately by a non-constant, continuous almost everywhere activation function $\sigma_i : \mathbb{R} \to \mathbb{R}$ that works on the output of the previous transformation in an element-wise fashion. In summary, the formulation of a layer is:

$$f_i(x_i) = \sigma_i(W_i x_i + b_i) \tag{2.4}$$

The activation function σ_i can be chosen arbitrary as long as it satisfy the requirement in the Universal Approximation Theorem. However, the most common activation functions are ReLU, Sigmoid and Tanh or Identity (*i.e.* no activation function). This formulation of a Neural Network layer is also called a Fully Connected layer or a Dense layer. As the layers in a Multilayer Percentron are connected together sequentially, the output of a layer is the input of the layer immediately right after it. In other word, we have:

$$x_{i+1} = f_i(x_i) \tag{2.5}$$

As the dimension of x_{i+1} is l_{i+1}, we can derive that the weight matrix W_i must be of size $l_{i+1} \times l_i$ while b_i is a l_{i+1} dimension vectors. W_i and b_i for $i : 1 \rightarrow N$ are the parameters of the Multilayer Perceptron and the targets of optimization to find the best configuration that fit the training data.

The first input x_1 to a Multilayer Perceptron are provided externally and the last output $y = f(x_N)$ are considered the output of the whole Neural Networks. The output y are then compared against a ground truth y_{gt} using a loss function $L(y, y_{gt})$ that measures the difference between the two. Training of Neural Networks then become minimizing the loss function and usually Stochastic Gradient Descent [8] is used for the optimization. As every layer in Multilayer Perceptrons are differentiable, the gradient/derivative used for Stochastic Gradient Descent can be computed automatically with Automatic Differentiation, in particular, using the famous backprogation algorithm.

2.2.2 Convolutional Neural Networks

For visual recognition related problem, such as working with images and videos, Multilayer Perceptron does not work so well because these kind of data are usually very high dimensional. Moreover, the inherent spatial structures of those data call for Neural Networks architecture that can exploit these structures efficiently. Convolutional Neural Networks are natural solution to those problems. In Convolutional Neural Networks, instead of having a weight matrix W_i do a linear transformation on the input x_i to project it to the output in each layer, a bunch of filters are convoluted with the input volume across all spatial dimensions. Contrary to popular thinking, Convolutional Neural Networks work on volumes to transform them into other volumes (images are special case of volumes where the depth dimension is one). The depth dimension of a volume effects the size of the 3D filters that would be applied to that volume as well as the number of filters that would be used to generate that volume. Selecting the depth of the volume in a Convolutional Neural

Networks usually depend on the problem one is working on, as the number of filters usually correlate to the amount of distinct information that we want to extract from the input of from the Convolutional layer.

Convolutional Neural Networks is related to Template Matching in tradition Computer Vision. However, instead of having the convolutional filters being designed by human, Convolutional Neural Networks treat the filters as learnable parameters and optimize the filters using Stochastic Gradient Descent so that the output of the whole Neural Networks best fit training data. In this weight, the filters in Convolutional Neural Networks are analogous to the weight matrix W_i in Multilayer Perceptron. In fact, implementation-wise, the convolution between the filters in Convolutional Neural Networks and the input volume use the same matrix multiplication as in Multilayer Perceptron. To implement convolution as matrix multiplication, first, one has to convert both the original input volume and the convolution filters into appropriate form. The algorithm to do this is called im2col [2]. Moreover, conceptually, one can think a fully connected layer applied to an volume that is flattened into a vector as a convolutional layer with the filter size big enough to cover the whole spatial dimensions of the volume.

Convolutional Neural Networks have found wide successes in solving many real life problems, from solving many computer vision tasks such object detection [37], image generation [14], image super resolution [26] to achieving superhuman performance in Atari games [27] and board games [43]. The effectiveness of Convolutional Neural Networks is hypothesized to comes from their inherent computation architectures, such as their translation invariant property, for example. Therefore, the act of designing Convolutional Neural Networks architectures can be thought as finding a good prior for the problem one is trying to solve. Recently, works such as [48] has found that the architecture of a generator Convolutional Network is sufficient to capture a great deal of low-level image statistics prior to any learning and randomly-initialized Neural Network can be used as a handcrafted prior with excellent results in standard inverse problems such as denoising, superresolution, and inpainting.

2.2.3 Universal Approximation Property of Neural Networks

Neural Networks has been studied extensively in the past few decades. One of the most iconic property of Neural Networks is, arguably, their Universal Approximation Property. Let F be a Multilayer Perceptron with at least two fully connected layers where the first layer has N output dimension and $\sigma : \mathbb{R} \to \mathbb{R}$ be a non-constant, bounded and continuous function. The Universal Approximation states that for any function f and any positive infinitesimal term ϵ, there always exists N large enough so that: $|f(x) - f(X)| < \epsilon$ for all x in the domain.

The theorem presents the powerful learning capacity of Neural Networks. It shows that a simple two-layered Multilayer Perceptron with enough parameters can match any function. The theorem doesn't propose how to find such set of Neural Networks parameters, but it guarantees they exists. Universal Approximation Property of Neural Networks was proved first for sigmoidal activation function in [10] and KurtHornik *et al.* proved the same theorem for classes of all non-constant, bounded, continuous function in [20]. The Universal Approximation Property of Neural Networks is the reason why we choose this class of function as implementation for our Differentiable Bypass concept that will be presented in the next chapter.

Figure 2.1 show an example of using Mutilayer Perceptron to mimics the function $f(x) = x \sin 4x - \cos \sqrt{15|x|}$ in the range $[-1, 1]$ with the number hidden neurons in the set $\{10, 50, 200, 1000, 5000, 20000\}$. As the number of hidden neurons increase, the Neural Networks fit the function better and better and with enough training one can fit this function perfectly.

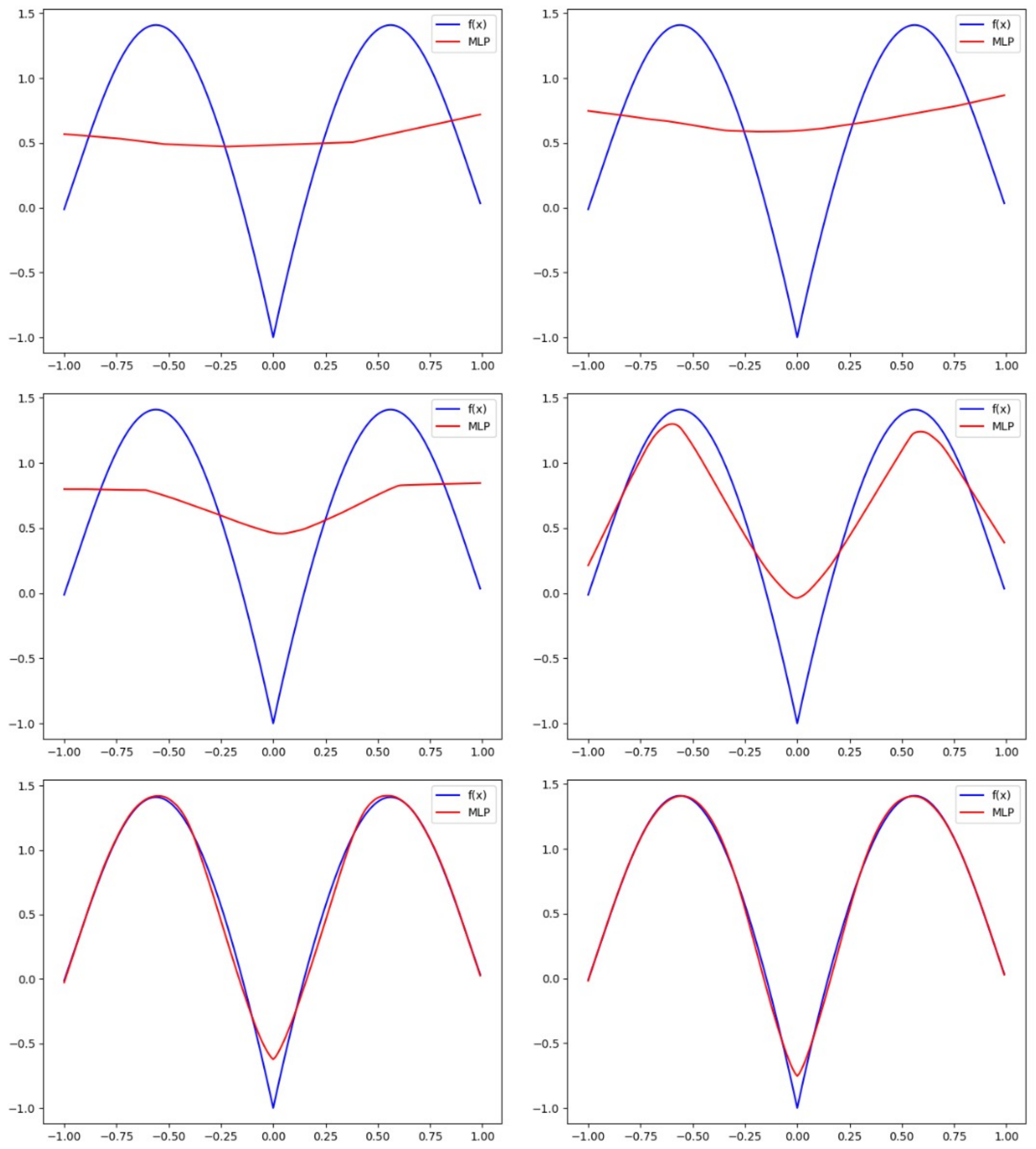

Figure 2.1: Example of using a two-layered Multilayer Perceptron to mimics a function. The directions from top to bottom, left to right indicates increasing number of parameters from 20 hidden layer unit to 20000.

Chapter 3

Extending Differentiable Programming to include non-differentiable components with Differentiable Bypass

3.1 Combining differentiable and non-differetiable modules for end-to-end training using the backpropagation algorithm

Consider the problem of combining N differentiable or non-differentiable modules $\{f_1, f_2, ..., f_N\}$ sequentially ($i.e.$ output of f_{i-1} is input into f_i for $i > 1$) into an end-to-end pipeline that can be trained end-to-end with the backpropagation algorithm. In this chapter, we only consider the case where the non-differentiable modules doesn't have any parameters that we want to train. If we would like to do so, we can treat these parameters or projections of these parameters as part of the input into the non-differentiable modules and proceed like normal. Let ϕ_i be the parameters of each f_i that is a differentiable module. Further, let θ_i for $i : 1 \to n$ be the input to each module f_i. By definition, θ_1 is the input provided by user to the whole pipeline and θ_i for each $i : 2 \to n$ are also the output of the previous module f_{i-1}. Additionally, define θ_{N+1} as the output of f_N ($i.e.$ the output of the whole pipeline).

Let L be a differentiable loss function that compare the output θ_{N+1} of the pipeline against some ground truth output that we want to minimize. To train the parameters of all differentiable modules, the backpropagation algorithm assume the gradients $\frac{\partial L}{\partial \theta_i}$ for all i where f_i is differentiable are provided. Then, the original backpropagation algorithm compute the gradient $\frac{\partial L}{\partial \phi_i}$ using the chain rule as follow:

$$\frac{\partial L}{\partial \phi_i} = \frac{\partial L}{\partial \theta_{i+1}} \times \frac{\partial \theta_{i+1}}{\partial \phi_i} \tag{3.1}$$

$\frac{\partial \theta_{i+1}}{\partial \phi_i} = \frac{\partial f_i(\theta_i)}{\partial \phi_i}$ is well-defined because the assumption that f_i are differentiable with respect to their own parameters. The problem is how to calculate $\frac{\partial L}{\partial \theta_{i+1}}$. In the case when there is no non-differentiable modules in the pipeline, $\frac{\partial L}{\partial \theta_i}$ can be computed recursively from the last to the first differentiable module by using chain rule:

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial L}{\partial \theta_{i+1}} \times \frac{\partial \theta_{i+1}}{\partial \theta_i} \tag{3.2}$$

Again, we have $\frac{\partial \theta_{i+1}}{\partial \theta_i} = \frac{\partial f_i(\theta_i)}{\partial \theta_i}$ to be well-defined for $i : 1 \rightarrow N$ because of the assumption that f_i are differentiable with respect to their own input. Addittionally, as L is differentiable with respect to θ_{N+1}, we can compute the gradient $\frac{\partial L}{\partial \theta_{N+1}}$ easily. Going from N down to 1 and applying the equation (3.2) recursively, we can compute all the $\frac{\partial L}{\partial \theta_i}$.

In the case where the sequential chain of f_i for $i : 1 \rightarrow N$ is broken, *i.e.* there is at some $i \in 1, ..., N$ where f_i is non-differentiable, the previously mentioned recursive strategy does not work anymore. The reason is that $\frac{\partial \theta_{i+1}}{\partial \theta_i} = \frac{\partial f_i(\theta_i)}{\partial \theta_i}$ is no longer well-defined for some i where f_i is non-differentiable. To continue using the backpropagation algorithm for gradient optimization, we need to estimate the these gradient somehow. The problem become estimating the derivative of a non-differentiable function with respect to its input, *i.e.* estimating $\frac{\partial f_i}{\partial \theta_i}$ where f_i is non-differentiable.

The problems of estimating the derivative of non-differentiable functions with respect to its input or parameters is well-studied with in the context of Reinforcement Learning, Control and Operation Research. The next sections describe two of the most popular methods for doing so, which are the score function estimator and evolution strategy and how they are related to our method described in this thesis. Then, we present our concept of Differentiable Bypass, which is the main contribution of the work. For the sake simplifying notations, we drop the index i from f, θ in the these sections.

3.2 Score function estimator

Consider the problem of computing the derivative of an expectation of a function $f(x)$ with respect to θ where f is non-differentiable and the a random variable x follow a distribution $p(x;\theta)$ parameterized by θ:

$$\nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)] = \nabla_\theta \int p(x;\theta) f(x) dx \tag{3.3}$$

This is a recurring task in machine learning, for example, to compute the policy gradient in reinforcement learning or computing the posterior in variational inference. The derivative is hard to compute because the integral over the whole distribution $p(x;\theta)$ are usually unknown. Using something called the *log derivative trick*, we can rewrite the integral into this form:

$$\nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)] = \mathbb{E}_{p(x;\theta)}[f(x) \nabla_\theta \log p(x;\theta)] \tag{3.4}$$

The formula on the right hand side has nicer property than the integral as we can compute the expectation by sampling. The exact derivation of the previous formula is shown below.

Start from equation (3.3), under some mild assumptions about the smoothness of f we can move the derivative inside the integral:

$$\begin{aligned} \nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)] &= \int \nabla_\theta p(x;\theta) f(x) dx \\ &= \int \frac{p(x;\theta)}{p(x;\theta)} \nabla_\theta p(x;\theta) f(x) dx \end{aligned} \tag{3.5}$$

Now, apply the derivative rule of a logarithm $\nabla_\theta \log p(x;\theta) = \frac{\nabla_\theta p(x;\theta)}{p(x;\theta)}$ we have:

$$\begin{aligned} \nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)] &= \int p(x;\theta) \nabla_\theta \log p(x;\theta) f(x) dx \\ &= \mathbb{E}_{p(x;\theta)}[f(x) \nabla_\theta \log p(x;\theta)] \\ &= \frac{1}{S} \sum_{s=1}^{S} f(x^{(s)}) \nabla_\theta \log p(x^{(s)};\theta), x^{(s)} \sim p(x^{(s)}, \theta) \end{aligned} \tag{3.6}$$

Note that the last line means we can can approximate the derivative $\nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)]$ by doing Monte Carlo sampling of $\nabla_\theta \log p(x;\theta)$ weighted by $f(x)$. This way of estimating the derivative is called the *score function estimator* [19]. In this thesis, we usually have $p(x;\theta) = \mathcal{N}(\theta, \sigma)$.

The score function estimator has the advantages is that it only requires the derivative of $\log p(x;\theta)$ to exists and it is *unbiased*. However, one downside is that this derivative estimator has high variance. Reducing the variance of this estimator as low as possible is crucial for effective learning. In practice, we subtract a control baseline b from the estimator:

$$\nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)] = \mathbb{E}_{p(x;\theta)}[(f(x) - b)\nabla_\theta \log p(x;\theta)] \tag{3.7}$$

Subtraction of this term does not change the expectation of the estimator while effecting the variance. A constant control variate b can be used. However, this is usually not useful and a careful, problem-dependent strategy of choosing b can lead to much lower variance for learning.

We can apply the score function estimator to compute the gradient of a non-differentiable function with respect to its input by treating the input θ into a non-differentiable f as the parameters of the distribution $p(x;\theta)$. This can be thought as applying a smoothing operation on the input space of the non-differentiable function $f(\theta)$. Futhermore, θ can be a parameters of f itself as well. That means we can also optimize the parameters of non-differentiable function using the score function estimator.

3.3 Evolution Strategy

Another way to estimate the gradient of the integral is to use evolution strategy [41]:

$$\nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)] = \frac{1}{\sigma}\mathbb{E}_{p(\epsilon)}[f(\theta + \sigma\epsilon)\epsilon] \tag{3.8}$$

Start from the score function estimator:

$$\nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)] = \mathbb{E}_{p(x;\theta)}[f(x)\nabla_\theta \log p(x;\theta)] \tag{3.9}$$

Note that: $p(x;\theta) = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(x-\theta)^2}{2\sigma^2}}$.

Let us reparameterize the random variable x as $x = \theta + \sigma\epsilon$, where ϵ is sampled from a normal distribution $p(\epsilon)$: $\epsilon \sim \mathcal{N}(0, \mathcal{I})$. Then,

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{p(x;\theta)}[f(x)] &= \nabla_\theta \mathbb{E}_{p(\epsilon)}[f(\theta + \sigma\epsilon)] \\
&= \mathbb{E}_{p(\epsilon)}[f(\theta + \sigma\epsilon)\nabla_\theta\{-\frac{(x-\theta)^2}{2\sigma^2} - \log\sqrt{2\pi}\sigma\}] \\
&= \mathbb{E}_{p(\epsilon)}[f(\theta + \sigma\epsilon)\{\frac{(x-\theta)}{2\sigma^2}\}] \qquad (3.10) \\
&= \mathbb{E}_{p(\epsilon)}[f(\theta + \sigma\epsilon)\{\frac{\epsilon}{\sigma}\}] \\
&= \frac{1}{\sigma}\mathbb{E}_{p(\epsilon)}[f(\theta + \sigma\epsilon)\epsilon]
\end{aligned}$$

Evolution strategy has the advantages that it is more easily parallelized with less communication overhead between processes and has lower variance than the score function estimator ([41]). Our preliminary results show that for our particular problem in this thesis, Evolution strategy gives much lower variance and faster learning during training than the score function estimator without having to select the control variate b. We hypothesize that is because of the reparameterization of x as this has been shown in previous literature ([39]). However, for our setting in this thesis, empirically, we show that Evolution strategy doesn't work well when the problem is high dimensional and our method, backpropagation via Differentiable Bypass, is preferred.

3.4 Estimating the gradient of non-differentiable modules using Differentiable Bypass

In this section, we present a novel way to compute the gradient of the non-differentiable function f using a concept called a Differentiable Bypass. The idea presented here is simple and can be applied to most non-differentiable f as long as we can find a function to approximate it. Let g be a differentiable function parameterized by ϕ_g that has the capacity to approximate f well enough. g can be any class of function. However, in this thesis, g is chosen to be neural networks because of their Universal Approximation property. Figure 3.1 shows a conceptual illustration of a Differentiable Bypass.

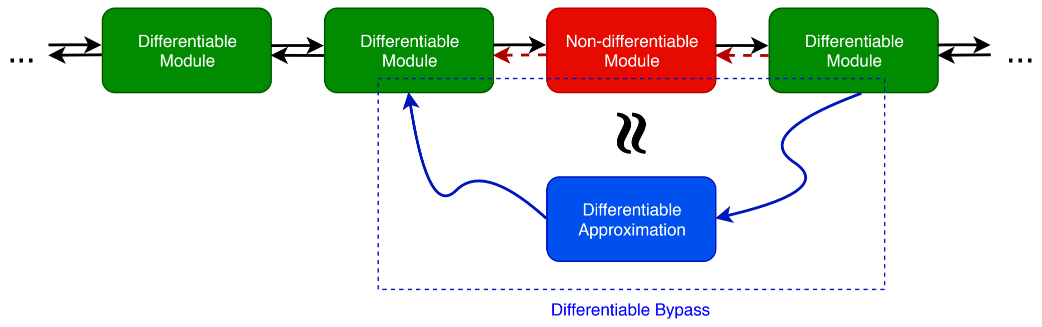

Figure 3.1: Illustration of a Differentiable Bypass. The broken red backward arrows means the red module is non-differentiable and Automatic Differentiation cannot work. The blue rounded rectangle is a differentiable function (for example, a neural network) that is trained to approximate the output of the non-differentiable red module. Blue arrow show the rerouting of gradient to avoid non-differentiable modules.

Let $y = f(\theta)$ and $y' = g(\theta)$ be the output of f and g for the input θ and $L_{approx}(y, y')$ be a differentiable loss function that compare these two output. The idea of Differentiable Bypass is to do the optimization:

$$\min_{\phi_g} L_{approx}(y, y') \tag{3.11}$$

a bunch of time so that g fit the surface f well enough. Then, instead of using the gradient of the function f with respect to θ, which is not available, we can *use the gradient of g with respect to θ instead*, *i.e.* use $\frac{\partial g}{\partial \theta}$ in place of $\frac{\partial f}{\partial \theta}$ in the normal backpropagation algorithm. In that case, we are assuming that:

$$\frac{\partial g}{\partial \theta} \approx \frac{\partial f}{\partial \theta} \tag{3.12}$$

if g fits f well enough.

In practice, during training of the whole $\{f_1, f_2, ..., f_N\}$ sequence, we need g to fit a specific f for the area around the input θ that we is computing the gradient for. Because of this reason, some kind of exploration in the area around the current θ need to be employed otherwise the function g will overfit to the value of θ and give us incorrect gradient. For an example of this exploration mechanism, please see the end-to-end learnabel Convolutional Neural Networks and Dynamic Programming algorithm that we propose in the next chapter.

The idea to estimate the gradient of a function using the gradient of another function that approximates it has been explored in the literature. Our method is related to Deep Deterministic Policy Gradient (DDPG) [27]. In DDPG, the authors train a neural network $Q(s, a)$, called the Q-network, to approximate the discounted return of an action a given a state s then use the gradient of $Q(s, a)$ with respect to a for gradient ascent on another network that output the action a. DDPG has found wide successes in training Reinforcement Learning agent to play Atari game and is still one of the algorithms of choice for solving continuous action environment. There are difference between our concept of Differentiable Bypass and DDPG, however. Our Differentiable Bypass employs supervised signal that come directly from the module right after it in the sequence while in DDPG, there is no such concept and the Q-network learns from the signal that come directly from the rewards at the end of the action execution. There is only one non-differentiable step in DDPG that come from the act of receiving a reward from the environment compared to multiple non-differentiable modules that can exist in our pipeline. The differences between our Differentiable Bypass and DDPG come from the setting of the problems even though the idea of using a Neural Networks to approximate the gradient of a non-differentiable step might be the same. Another work that is related to our is the Synthetic Gradient paper by Max Jaderberg *et al.* [22] where the authors use a Neural Network to directly output the gradient for training another Neural Network for decoupling the training of multiple layers in a Recurrent Neural Networks. The authors provided justification for training using Synthetic Gradient and proved the convergence of the training. Overall, we think the idea of approximating the gradient for training Neural Networks using other Neural Networks are very interesting and worth pursuing.

Chapter 4

End-to-end learning of Convolutional Neural Networks and Dynamic Programming for Left Ventricle Segmentation

Recent progress in medical image analysis is undoubtedly boosted by deep learning [15], [23]. Progress is observed in several medical image analysis tasks, such as segmentation [9], [33], registration [13], tracking [18] and detection [11]. One of the significant challenges in applying deep learning to medical image analysis is limited amount of labeled data [15].

Our contribution in this chapter is twofold. First, we demonstrate a combination of deep learning and a traditional method with strong prior knowledge can compensate for the inadequate amount of training data significantly. The method in this chapter is referred to as End-to-end learning of Convolutional Neural Networks and Dynamic Programming (EDPCNN). We use differentiable programming [6] (i.e., end-to-end learning) for combining different methods.

Our second contribution is a recommendation for combining a *non-differentiable* function with deep learning using the concept of Differentiable Bypass (DiffBypass) within an end-to-end learning framework. This is an extension of Differentiable Programming, which has been applied only to differentiable functions so far. In this context, we demonstrate the use of the universal function approximation property of neural networks to approximate a *non-differentiable* function.

4.1 Left Ventricle Segmentation and Combining Traditional Techinque with Deep Learning for Medical Imaging

The left ventricle appears as a “blob” object in short axis MRI. Traditionally active contours and level set based methods were used for blob object segmentation [1]. While these methods offer object shape constraints, they typically look for strong edges or statistical modeling for successful segmentation. These techniques lack a way to work with labeled images in a supervised machine learning framework. For complex segmentation tasks, such as cardiac MRI segmentation [5] these methods are inadequate. Deep learning (DL) has invigorated interest for these classic techniques in the recent years, including our present work, because starting from raw pixels DL can be trained end-to-end with labeled images. With the exception of limited literature, such as shape prior Convolutional Neural Networks (CNN) [52], DL lacks any inherent mechanism to incorporate prior knowledge about object shapes; instead, DL relies on the volume of labeled images to implicitly learn about object shapes or constraints. Hence, there is a need to combine CNN with these traditional methods so that the latter can provide adequate prior knowledge.

Hu *et al.* [21] proposed to use CNN to learn a level set function (signed distance transform) for salient object detection. Tang *et al.* [45] used level set in conjunction with deep learning to segment liver CT data and left ventricle from MRI. However, their method does not use end-to-end training for this combination. Deep active contours [40] combined CNN and active contours; the work, however, fell short of an end-to-end training process.

Literature on combined end-to-end learning is not yet abundant. End-to-end learning employing level set and deep learning-based object detector has been utilized in Le *et al.*’s work [25], where the authors modeled level set computation as a recurrent neural network. Marcos *et al.* [29] have combined CNN and active contours in end-to-end training with a structured loss function.

Proposed EDPCNN is another addition to the growing repertoire combining CNN and active contours with a noteworthy novelty. While all the aforementioned literature on segmentation combines differentiable components, in EDPCNN we demonstrate how to combine a DP-based active contour with CNN in an end-to-end fashion, where DP is non-differentibale.

Medical image analysis often has to deal with limited amount of labeled / annotated images. DL has been most successful where plenty of data was annotated, e.g., diabetic retinopathy [49]. Transfer learning is the dominant approach to deal with limited labeled data in medical image analysis. In transfer learning, a deep network is first trained on an unrelated, but large dataset, such as Imagenet; then the trained model is fine-tuned on smaller data set specific to the task. Transfer learning has been applied for lymph node detection and classification [42], localization of kidney [34] and many other tasks [44]. Data augmentation is also applied to deal with limited labeled data [38].

In this work, we present a complementary approach to work with limited amount of labeled images. Our guiding principle is to inject the learning system with prior knowledge about the solution. A similar argument was made by Ngo, Lu, and Carneiro [31] for combining level set and CNN to work with limited labeled data for left ventricle segmentation. For this segmentation task, the prior knowledge is a smooth shape, which can be modeled as a closed contour drawn through a star pattern. To inject such knowledge into the learning system, we resort to the principle of differentiable programming, where more than one differentiable algorithms are stitched together. However, the added difficulty in our case is the non-differentiable nature of DP that we overcome using the DiffBypass strategy mentioned in the previous Chapter.

4.1.1 U-Net for Segmenting Medical Image

U-Net is a Convolutional Neural Networks architecture used for solving the semantic segmentation problems. We include an introduction about U-Net here because the architecture will be used as the base Convolutional Neural Networks architecture for EDPCNN as well as the baseline we compare to.

Figure 4.1 show the architecture of U-Net.

U-Net is an autoencoder-like architecture. Conceptually, there are two part in the U-Net: the encoder and the decoder. The decoder takes the input image, progressively process and extract high level contextual features by using convolutional layers and max pooling. The spatial dimensions of the feature map reduce gradually while the depth dimension increases at the same time. After the encoder layers, we obtain a feature map that has 1024 depth channel. The decoder part of U-Net uses up-convolution layers, which consists of bilinear interpolation operation follow by convolution layers, to build up the segmentation mask, again, progressively from the extracted high level feature from the encoder. Additionally, U-Net employs skip connections from each layer in the encoder to its corresponding layer in the decoder for improving dense prediction accuracy. We choose U-Net as our base architecture because U-Net has found wide successes in segmenting medical images with low amount of data. However, other segmentation architectures, FCN [28] and SegNet [3] for examples, can be used as the baseline. For more information about the implementation detail of U-Net, please refer to the original paper [38].

4.2 Dynamic Programming

Use of Dynamic Programming (DP) in computer vision is wide ranging, including interactive object segmentation [12]. Here, we use the DP setup described in [35] to delineate star-shaped/blob objects that perfectly describe left ventricles in the short axis view.

Let the star pattern have N radial lines with M points on each line. Dynamic Programming minimizes the following cost function:

$$\min_{v_1,\ldots,v_N} E(N, v_N, v_1) + \sum_{n=1}^{N-1} E(n, v_n, v_{n+1}), \tag{4.1}$$

where each variable v_n is descrete and $v_n \in \{1, \ldots, M\}$. Cost component for the radial line n is $E(n, i, j)$ and it is defined as follows:

$$E(n,i,j) = \begin{cases} g(n,i) - g(n,i-1) + g(n \oplus 1, j) - g(n \oplus 1, j-1), & |i-j| \leq \delta \\ \infty, & \text{otherwise}, \end{cases} \tag{4.2}$$

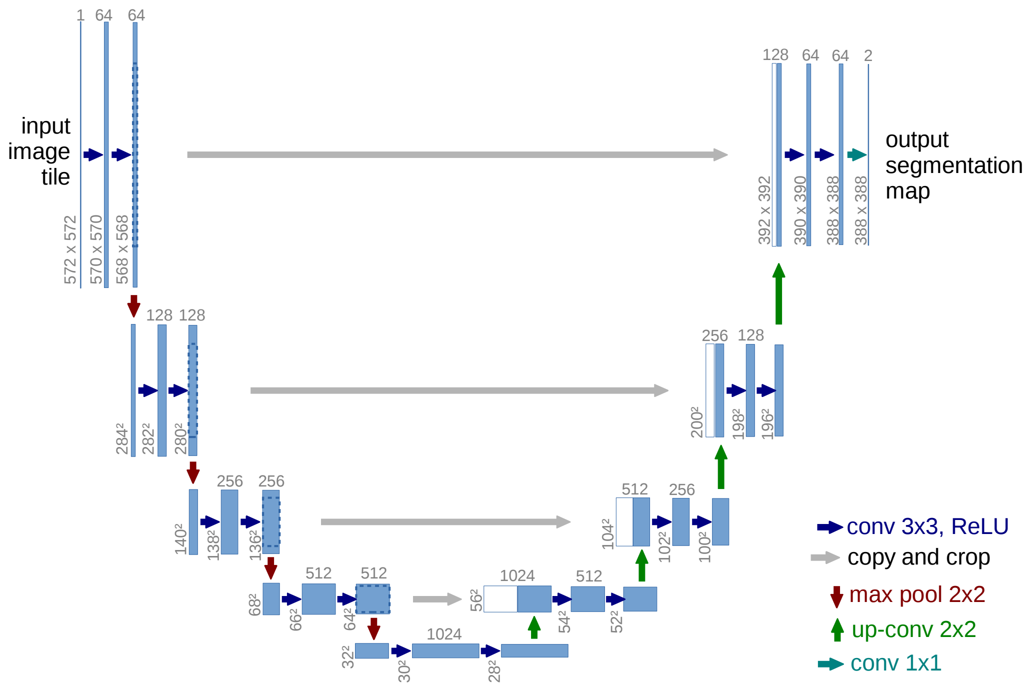

Figure 4.1: U-Net architecture. Image belongs to the original authors at [38]
.

where g is the Warped Map in the EDPCNN pipeline (Fig. 4.2), with $g(n, i)$ representing the value of Warped Map on the i^{th} point of radial line n. The symbol $\oplus$ denotes a modulo N addition, so that $N \oplus 1 = 1$ and $n \oplus 1 = n+1$ for $n < N$. The discrete variable $v_n \in \{1, \ldots, M\}$ represents the index of a point on radial line n. DP selects exactly one point on each radial line to minimize the directional derivatives of g along the radial lines. The collection of indices $\{v(1), \ldots, v(N), v(1)\}$ chosen by DP forms a closed contour representing a delineated left ventricle. To maintain the continuity of the closed contour, (4.2) imposes a constraint: chosen points on two consecutive radial lines have to be within a distance δ. In this fashion, DP acts as a blob object boundary detector maximizing edge contrast, while maintaining a continuity constraint.

Algorithm 1 implements DP, where a number of calls to the *argmin* function are responsible for the non-differentiable nature of it. So, during end-to-end learning we cannot rely on the automatic differentiation of software packages.

Naive implementation of the nested **for** loops in algorithm 1 is prohibitively inefficient for training of our model as its time complexity is $O(N^2M)$ for a

```
/* Construct value function $VF$ and index function $IF$ */
for $n = 1, \dots, N-1$ do
    for $i, k = 1, \dots, M$ do
        if $n == 1$ then
            $VF(n,i,k) = \min_{1 \le j \le M}[E(n,i,j) + E(n+1,j,k)]$ ;
            $IF(n,i,k) = \text{argmin}_{1 \le j \le M}[E(n,i,j) + E(n+1,j,k)]$ ;
        else
            $VF(n,i,k) = \min_{1 \le j \le M}[VF(n-1,i,j) + E(n+1,j,k)]$ ;
            $IF(n,i,k) = \text{argmin}_{1 \le j \le M}[VF(n-1,i,j) + E(n+1,j,k)]$ ;
        end
    end
end
/* Backtrack and output $v(1), \dots, v(N)$ */
$v(1) = \text{argmin}_{1 \le j \le M}[VF(N-1,j,j)]$;
$v(N) = IF(N-1, v(1), v(1))$;
for $n = N-1, \dots, 2$ do
    $v(n) = IF(n-1, v(1), v(n+1))$;
end
```

Algorithm 1: Dynamic programming

single contour and the number of contours processed for each training batch can be in order of thousands. Therefore, we need vectorize these **for** loops 1 so that they can be run efficiently on GPU. Moreover, pre-allocating of GPU memory at the beginning and for reuse later during execution time is required as memory allocation take a long time. To utilize GPU parallel computation capability even better, algorithm 1 is extended to the batch dimension and vectorized. Algorithm 2 shows the pseudocode of our modified version of algorithm 1 that can run up to thousands of time faster than the original on GPU. For a detailed implementation of algorithm 2, see our Github repository.

4.3 End-to-end training using DiffBypass

We apply DiffBypass to combine CNN with DP and refer to this method as end-to-end DP with CNN. As a significant test application, we use EDPCNN to segment left ventricle from short axis heart MRI [5]. Fig. 4.2 illustrates our processing pipeline. The input to the CNN (we use U-Net [38] in our experiments) is an MR image as shown in Fig. 4.3(a). Output from the CNN

```
/* Pre-allocate GPU arrays                                              */
Initialize energy array $E$ with size $(batch\_size, N, M + 2\delta, M + 2\delta)$ ;
Initialize value array $VF$ with size $(batch\_size, N - 1, M, M + 2\delta)$ ;
Initialize index array $IF$ with size $(batch\_size, N - 1, M, M)$ ;
/* Construct value function $VF$ and index function $IF$                */
for $n = 1, \ldots, N - 1$ do
    for $i, k = 1, \ldots, M$ do
        if $n == 1$ then
            $VS = E(:, 1, \delta+1 : M+\delta+1, :).toshape((-1, M, M+2\delta, 1))+$
              $E(:, 2, :, \delta + 1 : M + \delta + 1).toshape((-1, 1, M + 2\delta, M))$ ;
        else
            $VS = E(:, n - 1, \delta + 1 : M + \delta + 1, :$
              $).toshape((-1, M, M + 2\delta, 1)) + E(:, n + 1, :, \delta + 1 :$
              $M + \delta + 1).toshape((-1, 1, M + 2\delta, M))$ ;
        end
        $VF(:, n, :, \delta + 1 : \delta + M + 1) = \min_{1\leq j\leq M} VS(:, :, j, :)$ ;
        $IF(:, n, :, \delta + 1 : \delta + M + 1) = \text{argmin}_{1\leq j\leq M} VS(:, :, j, :)$ ;
    end
end
/* Backtrack and output $v(1), \ldots, v(N)$ like in the previous
   algorithm                                                            */
```

Algorithm 2: Efficient Dynamic programming

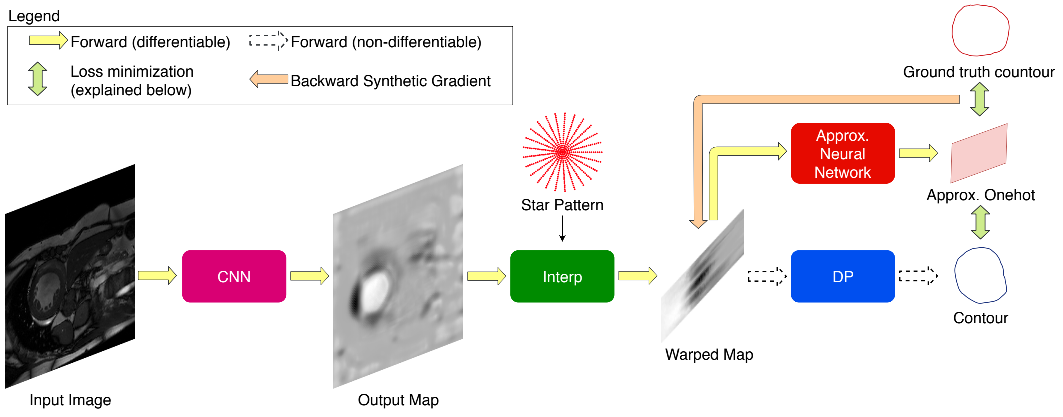

Figure 4.2: EDPCNN processing pipeline.

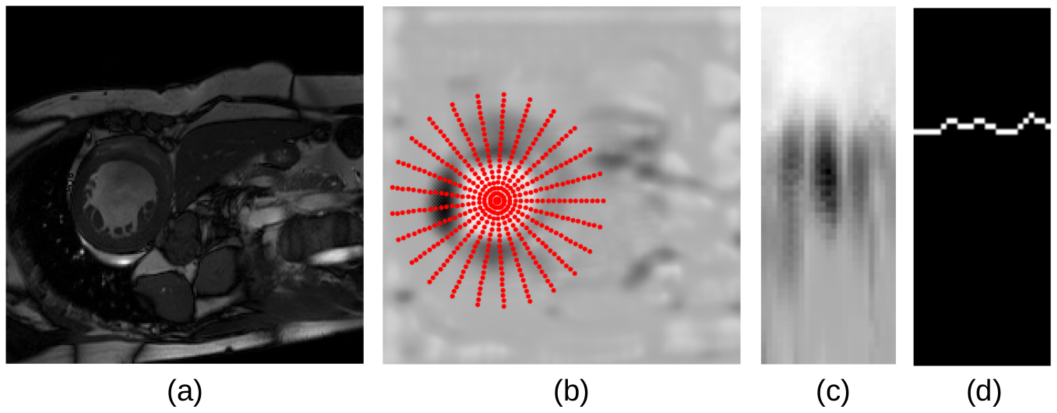
Figure 4.3: Illustrations of processing pipeline: (a) input image, (b) Output Map with an example star pattern, (c) Warped Map and (d) output indices indicating LV on the warped space

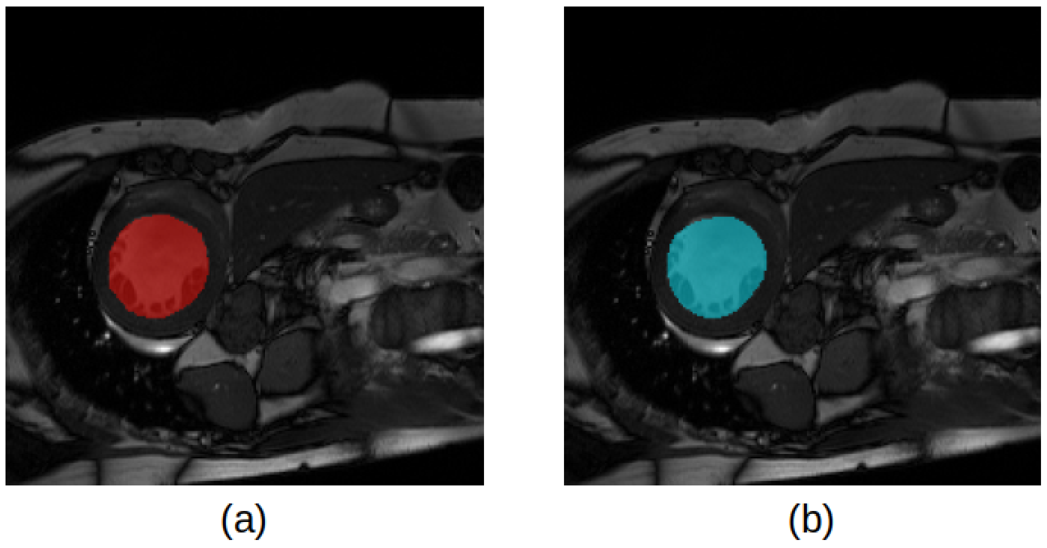
Figure 4.4: (a) segmentation obtained with EDPCNN (b) ground truth.

is a processed image, called output map, on which a pattern is overlaid in Fig. 4.3(b). The pattern consists of a few graduated radial lines. We refer to it as a “star pattern.” The interpolator (“Interp” in Fig. 4.2) interpolates output map on the points of the star pattern and warp the interpolated values in a matrix called “Warped Map” in Fig. 4.2. Fig. 4.3(c) illustrates a Warped Map. DP minimizes a cost function on the Warped Map and chooses exactly one point on each radial line in the star pattern to output a set of indices in the warped domain as shown in Fig. 4.3(d). Mapping the indices back to the image space gives us a closed contour as the final segmentation, as shown in Fig. 4.3(e). In comparison, ground truth segmentation, created by an expert, is shown in 4.3(f).

EDPCNN pipeline is differentiable except for *argmin* function calls inside the DP module that renders the entire pipeline unsuitable for end-to-end learning. For example, if there is a differentiable loss function that measures the error between output contour and ground truth contour, we would not be able to train the system end-to-end, because gradient would not reliably flow back across the *argmin* function using the standard mechanisms of automatic differentiation. In the past, soft assignment has been utilized to mitigate the issue of non-differentiability for the *argmin* function [4]. Here, we illustrate DiffBypass to approximate the gradient of the Warped Map, so that all the preceding differentiable layers (Interp and CNN) can apply standard backpropagation to learn trainable parameters. Fig. 4.2 illustrates that an approximating neural network (“Approx. Neural Network”) creates a *DiffBypass* for the non-differentiable DP module. This second neural network approximates the contour that the DP module outputs. Then a differentiable loss function is applied between the ground truth contour and the output of the approximating neural network, making backpropagation possible with automatic differentiation. This mechanism is known as synthetic gradients, because the gradients of the approximating neural network serves as a proxy for the gradients of the DP module.

The DiffBypass strategy uses the universal function approximation property of neural networks. This strategy is similar to using Synthetic Gradient

[22] to train deep neural networks asynchronously to yield faster training. In order to use the DiffBypass strategy in the EDPCNN processing pipeline, as before, let us first denote by g the Warped Map, which is input to the DP module. Let $L(p, p_{gt})$ denote a differentiable loss function which evaluates the collection of indices output from the DP module $p = DP(g) = \{v_1, ..., v_N\}$ against its ground truth $p_{gt} = \{v_1^*, ..., v_n^*\}$, which can be obtained by taking the intersection between the ground truth segmentation mask and the radial lines of the star pattern. Let us also denote by F a neural network, which takes g as input and outputs a *softmax* function to mimic the output of DP. In Fig. 4.2, F apperas as "Approx. Neural Network." Let ϕ and ψ denote the trainable parameters of F and U-Net, respectively. The inner minimization in the Dynamic Programming algorithm (Algorithm 3) trains the approximating neural network F, whereas the outer minimization trains U-Net. Both the networks being differentiable are trained by backpropagation using automatic differentiation. The general idea here is to train F to mimic the output indices of the DP module p as closely as possible, then use $\nabla_g L(F(g), p_{gt})$ to approximate $\nabla_g L(p, p_{gt})$, bypassing the non-differentiable *argmin* steps of DP entirely. Minimizing $L(p, p_{gt})$ then becomes minimizing $L(F(g), p_{gt})$ with this approximation.

The loss function L in this work is chosen to be the Cross Entropy between the output of F against the one-hot form of $\{v_1, ..., v_N\}$ or $\{v_1^*, ..., v_N^*\}$. In this case, $F(g)$ comprises of N vectors, each of size M, representing the *softmax* output of the classification problem for selecting an index on each radial line.

We have observed that introducing randomness as a way of exploration in the inner loop by adding $\sigma\varepsilon_s$ to g is important for the algorithm to succeed. Instead of minimizing $L(F(g), DP(g))$, we minimize $L(F(g+\sigma\varepsilon_s), DP(g+\sigma\varepsilon_s))$. In comparison, the use of Synthetic Gradient in asynchronous training [22] did not have to resort to any such exploration mechanism. The correctness of the gradient provided by the DiffBypass depends on how well F fits the surface of the DP algorithm around g. We hypothesize that without sufficient exploration added, F will overfit to a few points on the surface and lead to improper gradient signal. Hyperparameter σ can be set using cross validation, while the number of noise samples S controls trade off between gradient accuracy and training time. We found that $\sigma = 1$ and $S = 10$ works well for our experiments.

```
for $I, p_{gt} \in$ *Training {Image,Ground Truth} Batch* do
    $g = Interp(Unet(I))$;
    Initialize $s$ to 0;
    while $s < S$ do
        Sample $\varepsilon_s$ from $\mathcal{N}(0; I)$;
        $\min_\phi L(F(g + \sigma\varepsilon_s), DP(g + \sigma\varepsilon_s))$;
        $s = s + 1$;
    end
    $\min_\psi \ L(F(g), p_{gt})$;
end
```

Algorithm 3: Training EDPCNN using DiffBypass

Chapter 5

Experiment Setup and Results

In this chapter, we discuss the experiment setup to evaluate the effectiveness our combining Convolutional Neural Networks and Dynamic Programming into end-to-end training agains our baselines.

5.1 Dataset and Preprocessing

We evaluate the performance of EDPCNN against U-Net on a modified ACDC [7] datatset. As the test set is not publicly available, we split the original training set into a training set and a validation set according to [5]. Following the same work, the images are re-sampled to a resolution of 212×212. As the original U-Net model does not use padded convolution, each image in the dataset has to be padded to size 396×396 at the beginning, so that the final output has the same size as the original image. After these steps, we remove all images that does not have the left ventricle class from the two datasets, resulting in a training set of 1436 images and a validation set of 372 images.

We train U-Net and EDPCNN increasing training sample size from 10 training images to the full training set size, 1436. To avoid ordering bias, we randomly shuffle the entire training set once, then choose training images from the beginning of the shuffled set, so that each smaller training set is successively contained in the bigger sets, creating telescopic training sets, suitable for an ablation study.

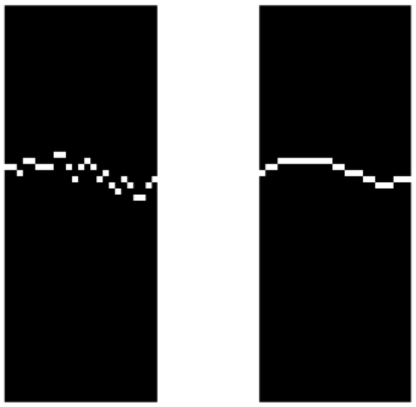

Figure 5.1: Left: Original indices. Right: Smoothed indices.

5.2 Postprocessing

As the output contour of DP may sometimes be jagged, we employ a post-processing step where the output indices are smoothed by a fixed 1D moving average convolution filter with circular padding. The size of the convolutional filter is set using a heuristic to be around one-fourth the number of radial lines on the star pattern. This post-processing also has the effects of pushing the contour to be closer to a circle, which is also a good prior for the left ventricle. This step improves our validation accuracy by around 0.5 to 0.8 percent. Since Differentiable Bypass mimics the post-processed output, postprocessing is a part of the end-to-end processing. Figure 5.1 shows an example of the selected indices before and after postprocessing step.

5.3 Evaluation Metric

For evaluation of a segmentation against its corresponding ground truth, we use Dice score [5], a widely accepted metric for medical image segmentation. EDPCNN requires the star pattern to be available so that the output of U-Net can be interpolated on the star pattern to produce Warped Map. The star pattern is fixed; but its center can be supplied by a user in the interactive segmentation. For all our experiments, the ground truth left ventricle center for an image serves as the center of the star pattern for the same image. While by design EPDCNN outputs a single connected component, U-Net can produce as many components without any control. Thus, to treat the evaluation of U-Net fairly against EDPCNN, in all the experiments we compute Dice scores within a square, which tightly fits the star pattern. So, any connected component produced by U-Net outside of this square is discarded during Dice score computation.

5.4 Training Details and Hyperparameters

We train U-Net and EDPCNN using Adam optimizer [24] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and a learning rate value of 0.0001 to make the training of U-Net stable. Training batch size is 10 for each iteration and the total number of iteration is 20000. No learning rate decay as well as weight decay are used because we have not found these helpful. We evaluate each method on the validation set after every 50 iterations and select the model with the highest validation Dice score.

For EDPCNN, we use nearest neighbor method to interpolate the output of U-Net on the star pattern to compute Warped Map g. We choose the center of the star pattern for each image to be the center of mass. To make the model more robust and have better generalization, during training, we randomly jitter the center of the star pattern with the requirement that the center will still stay inside the object. Define "object radius" as the distance between the center of mass of an object to its nearest points on the contour. We then randomly move the true center inside a 2D truncated normal distribution with mean equal to the coordinate of the center of mass and standard deviation equal to the object radius. We find that this kind of jittering can improve the dice score on smaller training sets by up to about 2%. We also randomly rotate the star pattern from -0.5 to 0.5 radian as an additional random exploration.

The radius of the star pattern is chosen to be 65 so that all objects in the training set can be covered by the pattern after taking into account the random placement of the center during training. The number of points on a radial line has also been chosen to be the radius of the star pattern: $M = 65$. For the number of radial lines N and the smoothness parameter δ, we run a

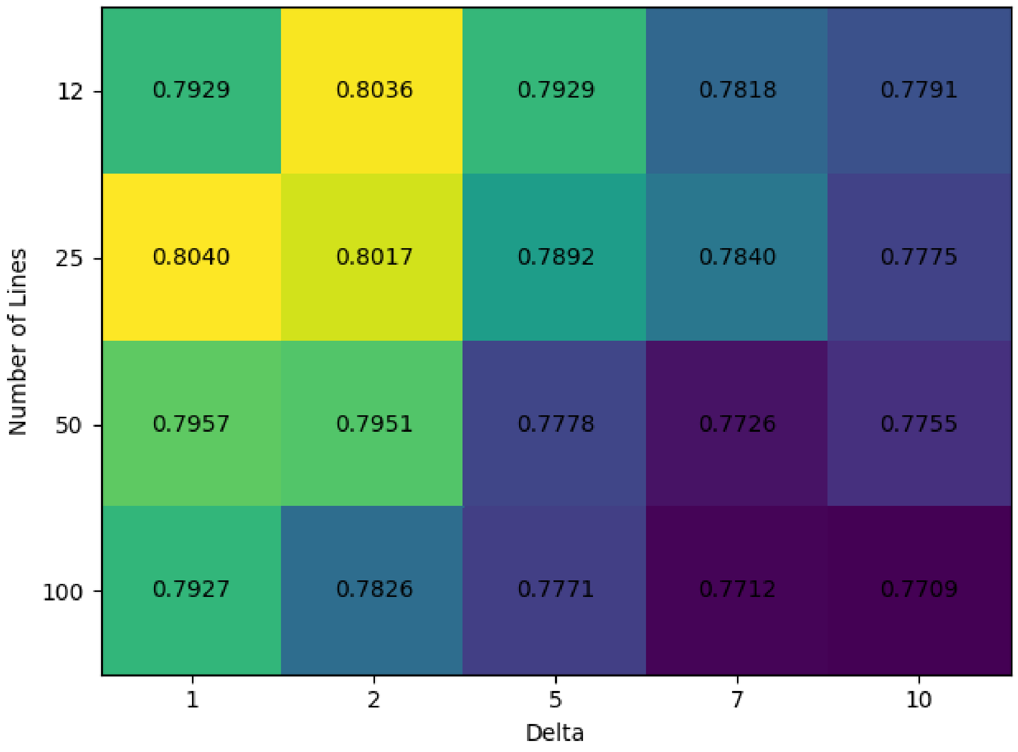

Figure 5.2: Hyper-parameters search for DiffBypass.

grid search over $N \in \{12, 25, 50, 100\}$, $\delta \in \{1, 2, 5, 7, 10\}$ and find $N = 25$, $\delta = 1$ to be good values. We also find that the performance of our algorithm is quite robust to the choices of these hyperparameters. The Dice score only drops around 3% when the values of N and δ are extreme (e.g. $N = 100$, $\delta = 10$). Figure **??** shows the result of doing a grid search for selecting N and δ on training set size of 200 samples over 200 epochs. Lastly, for the optimization of $\min_\phi L(F(g + \sigma\varepsilon_s), DP(g + \sigma\varepsilon_s))$ in Algorithm 3, to make $F(g)$ fit $DP(g)$ well enough, we do the minimization step repeatedly for 10 times.

The architecture of F used to approximate the output of DP is a U-Net-like architecture. As the size of g is smaller and the complexity of g is likely to be less than the original image, instead of having 4 encoder and 4 decoder blocks as in U-Net, F only has 3 encoder and 3 decoder blocks. Additionally, we use padding for convolutions/transposed convolutions in the encoder/decoder blocks so that those layers keep the size of the feature maps unchanged instead of doing a large padding at the beginning like in U-Net. This is purely for convenience. Note that these choices can be arbitrary as long as F can fit the surface of DP well enough. For the same reason, we find that the number of output channels in the first convolution of F, called *base_channels*, is an important hyperparameter because this value controls the capacity of F and affects how well F fits the surface of DP. We find that $base\_channels = 8$ works well for our algorithm (compared to 64 in U-Net).

5.5 Experiment Results and Discussions

Fig. 5.3 shows an ablation study to demonstrate the effectiveness of combining CNN and DP in an end-to-end learning pipeline. The numerical result for this figure can be found in 5.1. The horizontal axis shows the number of training images and the vertical axis shows the Dice score of LV segmentation on a fixed validation set of images. Note that when the number of training images is small, EDPCNN performs significantly better than U-Net. Eventually, as the training set grows, the gap between the Dice scores by U-Net and EDPCNN starts to close. However, we observe that EDPCNN throughout maintains its superior performance over U-Net. Results section (Table 5.2) shows that the performance gain of EDPCNN over U-Net comes only with a modest increase (16%) in the processing time.

Fig. 5.3 shows another experiment called "U-Net+DP". In the U-Net+DP processing pipeline, DP is applied on the output of a trained U-Net without end-to-end training. Once again, EDPCNN shows significantly better performance than U-Net+DP for small training sets, demonstrating the effectiveness of the end-to-end learning. We hypothesize that DP infuses strong prior knowledge in the training of U-Net within EDPCNN and this prior knowledge acts as a regularizer to overcome some of challenges associated with small training data.

Supply of the target object center to EDPCNN can be perceived as a significant advantage. We argue that this advantage cannot overshadow the contribution of end-to-end learning. To establish this claim, we refer readers

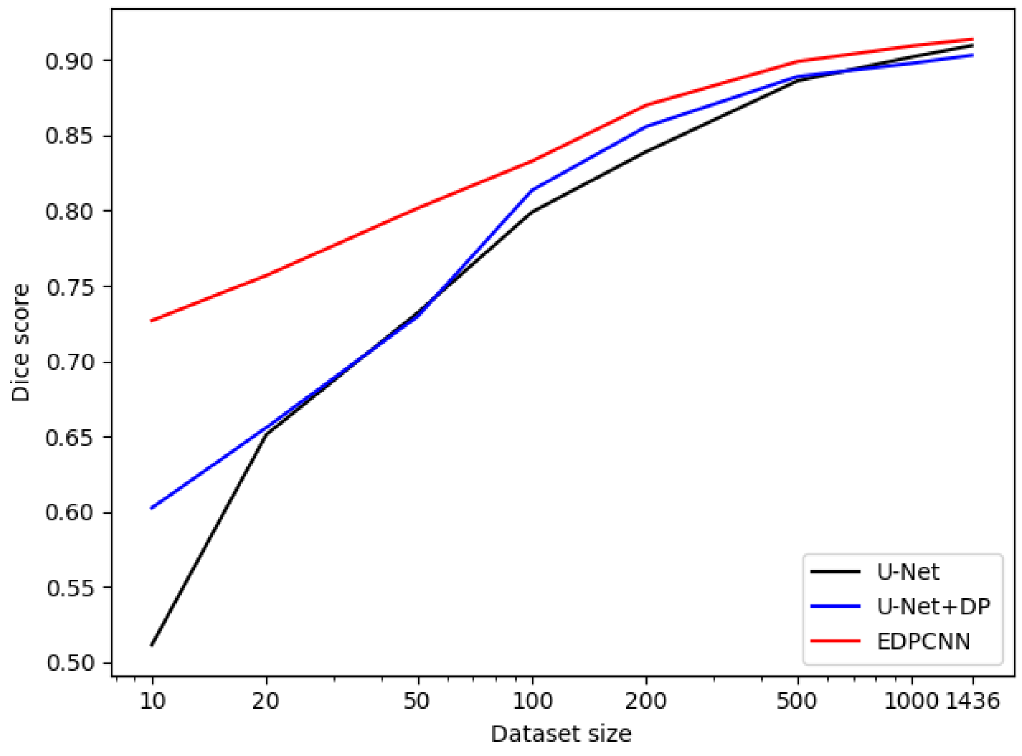

Figure 5.3: Ablation study: training set size *vs.* Dice score on validation set. EDPCNN uses the original U-Net as base network architecture.

Table 5.1: Dice score of ablation study for U-Net, U-Net+DP and EDPCNN at different training set sizes.

	10	20	50	100	200	500	1000	1436
U-Net	0.5115	0.6511	0.7321	0.7990	0.8392	0.8862	0.9020	0.9096
U-Net+DP	0.6025	0.6558	0.7296	0.8135	0.8558	0.8891	0.8978	0.9031
EDPCNN	0.7270	0.7569	0.8015	0.8328	0.8701	0.8990	0.9093	0.9137

to Fig. 5.3 and note that the Unet+DP model, despite having the same advantage, lags significantly behind EDPCNN. Therefore, end-to-end learning is the only attributable factor behind the success of EDPCNN.

Further, to test the robustness of EDPCNN with respect to the position of the star pattern center, we perform an experiment where the supplied center during testing is purposely jittered inside the object in the way it was done during training. Fig. 5.4 shows the effect of random jitter with the increase of jitter radius from no jitter to 0.5 of the object radius. We can see that there is no significant degradation in performance, especially for 0.2 jitter or below. Fig. 5.4 plots the average Dice scores for these experiments. In all the cases, the standard deviation of Dice scores remains small, below 0.01. Thus, the standard deviation has not been shown in Fig. 5.4.

Fig. 5.5 shows training iterations *vs.* Dice scores for training and validation sets. Two training sample sizes were shown: 10 and 1436 (full training set). For training sample size 10, the Dice scores on the validation set show significant variations and eventual overfitting for the U-Net model, while EDPCNN does not exhibit such a tendency. This overfitting behaviour is counter intuitive, because learnable parameters in EDPCNN form a superset for those in U-Net. Our hypothesis is that a strong object model and prior knowledge infused by DP into U-Net prevents overfitting.

Table 5.2 shows running time for U-Net and EDPCNN. We observe that computationally EDPCNN is about 64% more expensive during training. However, test time for EDPCNN is only about 16% more than that of U-Net.

To further evaluate the result of what the U-Net in EDPCNN has learned, we plot the predicted segmentation mask as well as the output of U-Net for some typical images that has the left ventricle of size from small to medium and

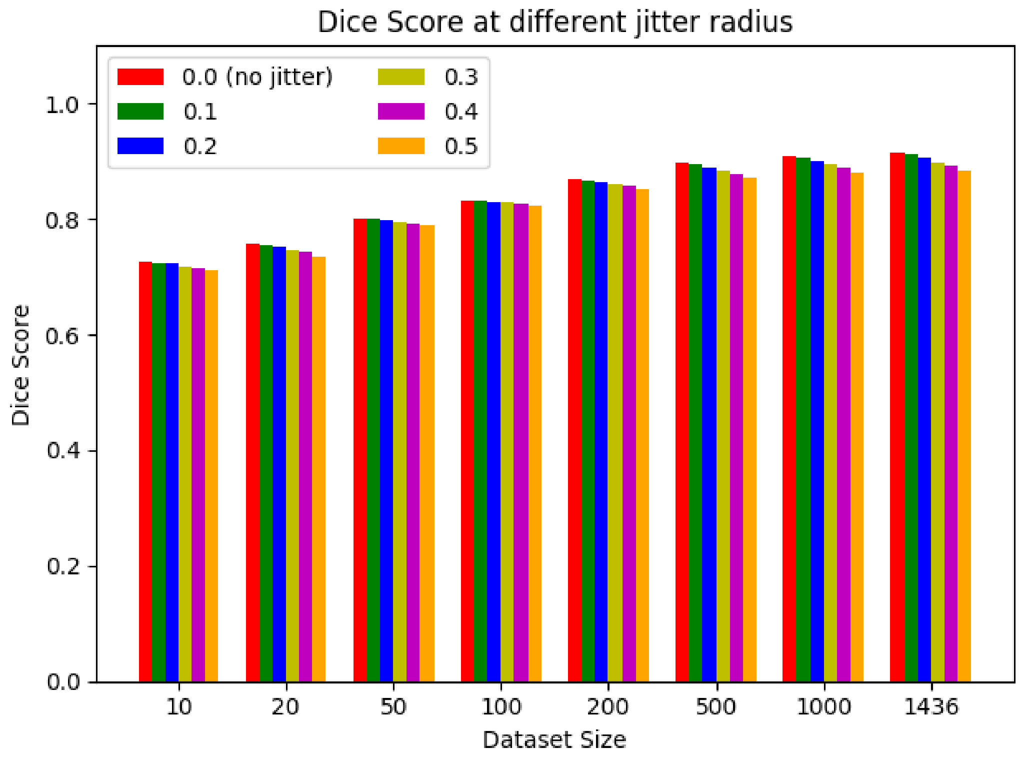

Figure 5.4: Robustness test: Dice Score at different dataset size and different jitter radius.

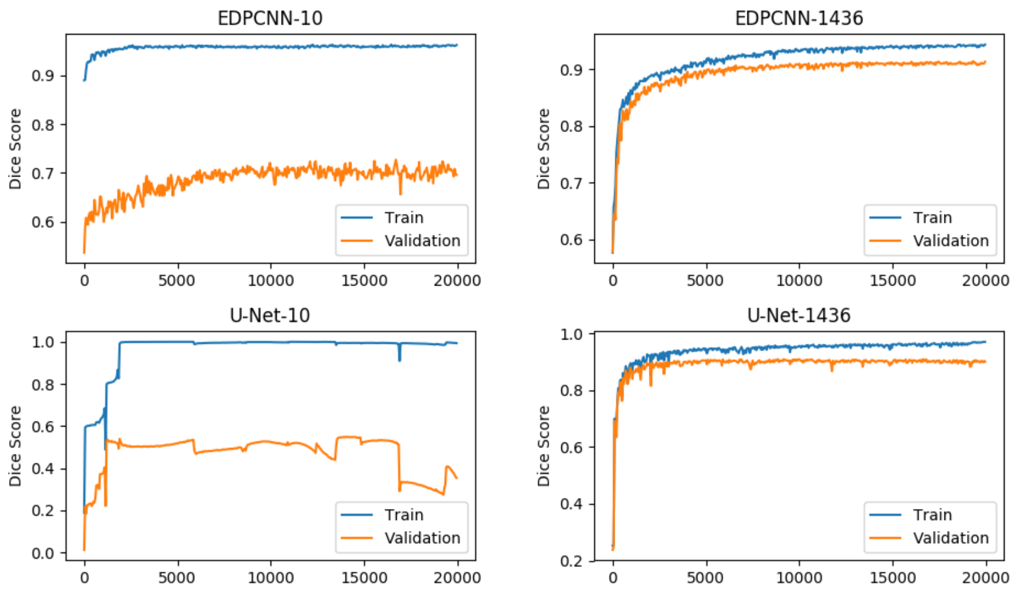

Figure 5.5: Dice Score on the training and validation sets during training for EDPCNN and UNet for dataset size of 10 and 1436.

Table 5.2: Computation time on an NVIDIA GTX 1080 TI

Method	Time / Training iteration	Total training time	Inference time / Image
U-Net	0.96s	5h 20m	0.01465s
EDPCNN	1.575s	8h 45m	0.01701s

large in figure 5.6. Overall, we find that U-Net learn to distinguish the edge of the left ventricle against the rest of the image very strongly. Additionally, figure 5.7 shows a failure case of our EDPCNN pipeline. In general, we find EDPCNN doesn't work as well on images with the left ventricle of smaller size.

Finally, we do another ablation study by training EDPCNN using Evolution Strategy (ES). We find that ES work well on a low dimensional version of U-Net where the input is not padded, resulting in a final feature map of lower dimension that will be upsampled as input into the Dynamic Programming step. In this low dimensional optimization problem, ES works well and has similar performance to DiffBypass (see figure 5.8. However, when we use ES on the original U-Net, the performance of ES is worse than using DiffBypass. In fact, ES on original U-Net is even worse than ES on the low dimensional U-Net (see figure 5.9. We hypothesize that this maybe because the number of random samples used to train ES does not scale well with the dimension of the input. This show that our DiffBypass migh be preferred over Evolution Strategy for estimating the gradient of non-differentiable module in high dimensional problems.

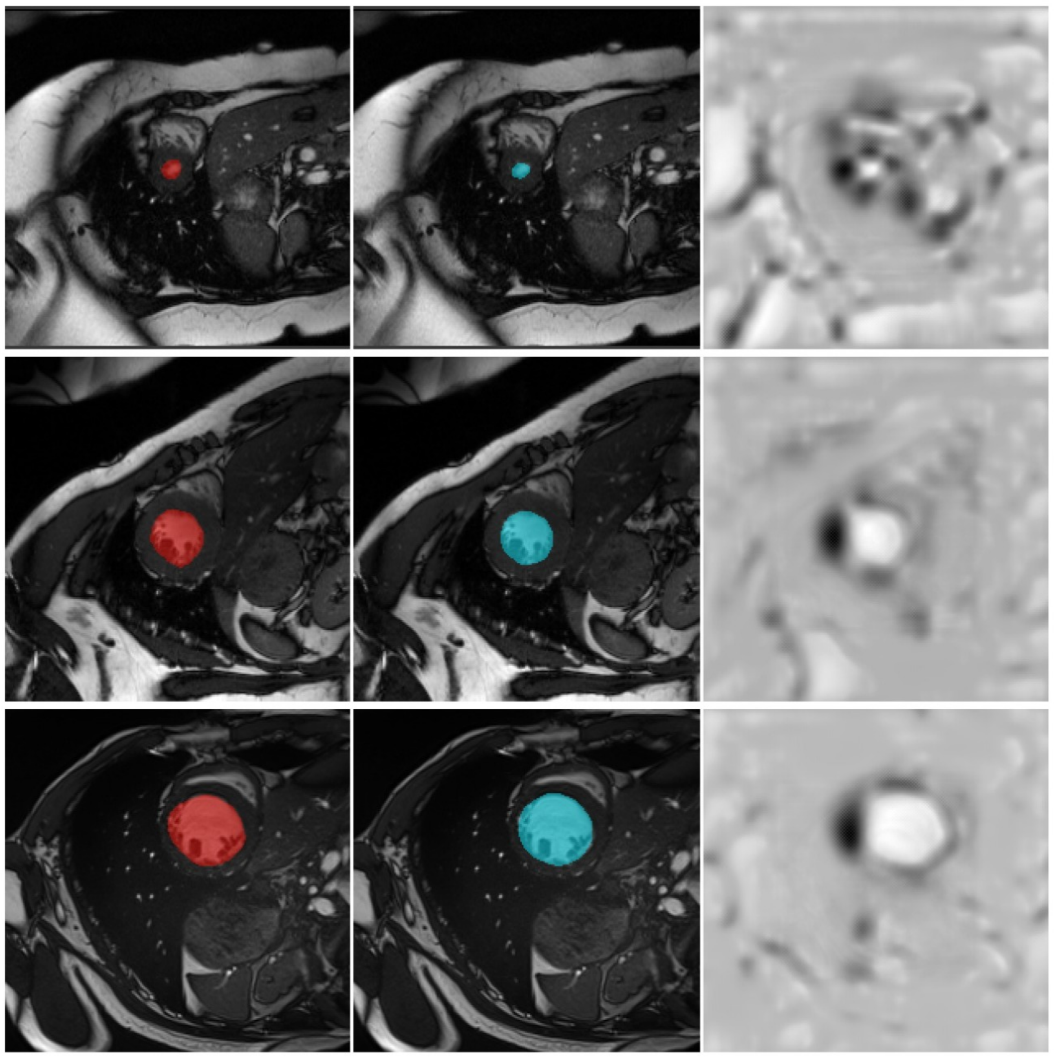

Figure 5.6: From top to bottom: segmentation mask from small to large. From left to right: ground truth mask, predicted mask, CNN output.

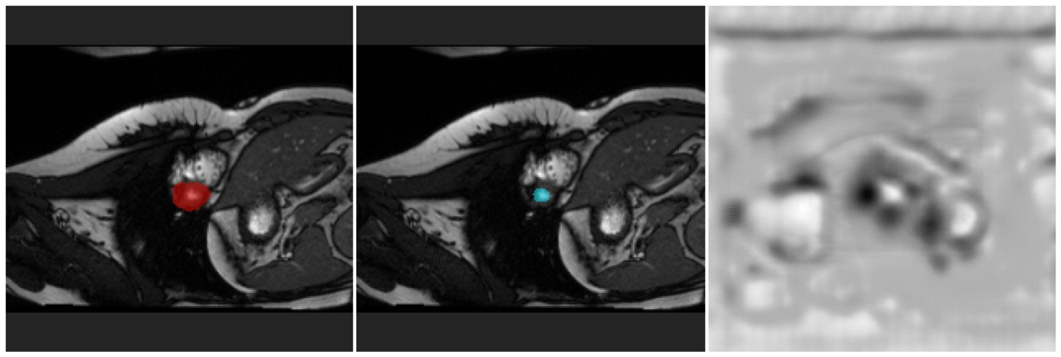

Figure 5.7: A failure case. From left to right: ground truth mask, predicted mask, CNN output.

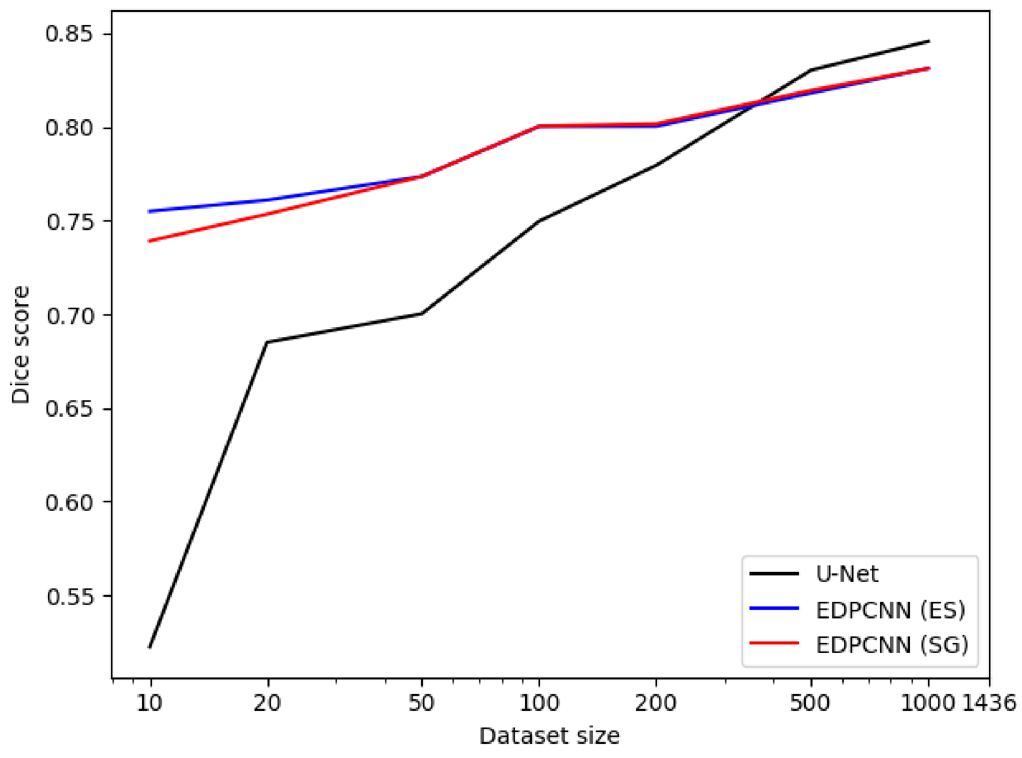

Figure 5.8: Training set size *vs.* Dice score on validation set for U-Net, EDPCNN trained using Evolution Strategy (ES) and DiffBypass (denoted as EDPCNN (SG)). The base U-Net used here is a modified version where no padding happen at the beginning.

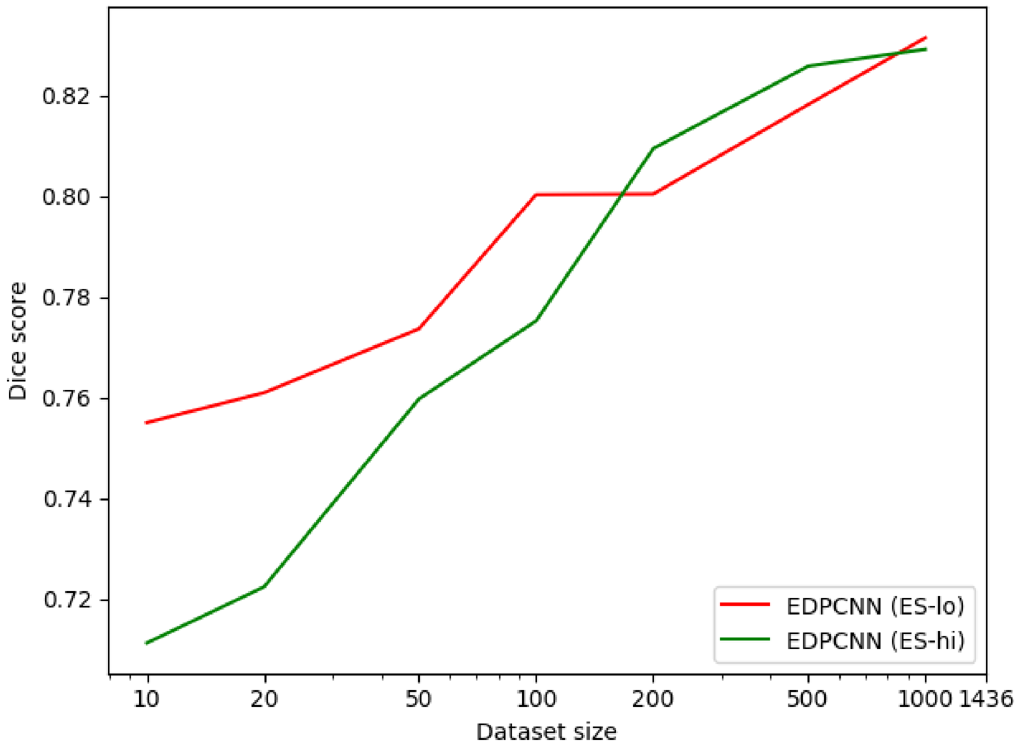

Figure 5.9: Ablation study: Performance of EDPCNN trained using Evolution Strategy. EDPCNN (ES-lo) denotes using the U-Net version without input padding as the base CNN while EDPCNN (ES-hi) denotes using the orignal U-Net as the base CNN.

Chapter 6

Conclusion

In this thesis, we present a method to combine a non-differentiable module into Differentiable Programming via a concept called Differentiable Bypass. We illustrate how to combine Convolutional Neural Networks and Dynamic Programming for end-to-end learning. Combination of Convolutional Neural Networks and traditional tools is not new; however, the novelty here is to handle a *non-differentiable* module, dynamic programming, within the end-to-end pipeline. We employ a neural network as our Differentiable Bypass to approximate the gradient of the non-differentiable module. We found that the approximating neural network should have an *exploration* mechanism to be successful.

As a significant application we choose left ventricle segmentation from short axis MRI. Our experiments show that end-to-end combination is beneficial when training data size is small. Our end-to-end model has very little computational overhead, making it a practical choice.

In the future, we plan to segment myocardium and right ventricle with automated placement of star patterns. For these and many other segmentation tasks in medical image analysis, strong object models given by traditional functional modules, such as dynamic programming, provide a way to cope with the lack of training data. Our presented method has the potential to become a blueprint to expand differentiable programming to include *non-differentiable* modules.

The concept of Differentiable Bypass we presented here works in the case where the non-differentiable modules have no parameters. A natural extension of Differentiable Bypass is to include the training of non-differentiable modules that have parameters into the pipeline as well. To do this, one possible solution could be to model the Differentiable Bypass as a function of *both the input into the non-differentiable module as well as its parameters.* We think that this extension is a very interesting concept and would like to pursuit this direction in the future.

Another very ambitious direction to extend this thesis would be automating the designing the neural network architecture of the Differentiable Bypass. To do this, Neural Architecture Search would be required. Additionally, we can also model the problem of choosing the Differentiable Bypass architecture as a Reinforcement Learning problem as well. Successfully solving the problem of automating the architecture search for the Differentiable Bypass would implies lots of breakthrough for Differentiable Programming. For one, it means that we can construct a black box programming language that allow users to write, combine and optimize modules and functions that are either differentiable or non-differentiable. To be able to achieve this step would require a lot of engineering as well as scientific efforts. However, we think that the step of being able to optimize both differentiable as well as non-differentiable modules at the same time will be an important step to help solving many problems and worth pursuing.

References

[1] S. T. Acton and N. Ray, "Biomedical image analysis: Segmentation," *Synthesis Lectures on Image, Video, and Multimedia Processing*, vol. 4, no. 1, pp. 1–108, 2009. DOI: 10.2200/S00133ED1V01Y200807IVM009. 24

[2] *Artificial intelligence gitbook*, http://archive.is/dqmXQ, Accessed: 2018-12-22. 13

[3] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *arXiv preprint arXiv:1511.00561*, 2015. 26

[4] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," *ArXiv e-prints*, Sep. 2014. arXiv: 1409.0473 [cs.CL]. 31

[5] C. F. Baumgartner, L. M. Koch, M. Pollefeys, and E. Konukoglu, "An exploration of 2d and 3d deep learning techniques for cardiac mr image segmentation," in *Statistical Atlases and Computational Models of the Heart. ACDC and MMWHS Challenges*, M. Pop *et al.*, Eds., Cham: Springer Int. Publishing, 2018, pp. 111–119, ISBN: 978-3-319-75541-0. 24, 28, 34, 35

[6] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: A survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018. [Online]. Available: http://jmlr.org/papers/v18/17-468.html. 4, 6–10, 23

[7] O. Bernard, A. Lalande, C. Zotti, F. Cervenansky, X. Yang, P.-A. Heng, I. Cetin, K. Lekadir, O. Camara, M. A. G. Ballester, *et al.*, "Deep learning techniques for automatic mri cardiac multi-structures segmentation and diagnosis: Is the problem solved?" *IEEE Transactions on Medical Imaging*, 2018. 34

[8] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, Springer, 2010, pp. 177–186. 12

[9] T. Brosch, L. Y. W. Tang, Y. Yoo, D. K. B. Li, A. Traboulsee, and R. Tam, "Deep 3d convolutional encoder networks with shortcuts for multiscale feature integration applied to multiple sclerosis lesion segmentation," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1229–1239, May 2016, ISSN: 0278-0062. DOI: 10.1109/TMI.2016.2528821. 23

[10] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989. 14

[11] Q. Dou, H. Chen, L. Yu, L. Zhao, J. Qin, D. Wang, V. C. Mok, L. Shi, and P. Heng, "Automatic detection of cerebral microbleeds from mr images via 3d convolutional neural networks," *IEEE Trans. Med. Im.*, vol. 35, no. 5, pp. 1182–1195, May 2016, ISSN: 0278-0062. 23

[12] P. F. Felzenszwalb and R. Zabih, "Dynamic programming and graph algorithms in computer vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 4, pp. 721–740, Apr. 2011, ISSN: 0162-8828. DOI: `10.1109/TPAMI.2010.135`. 26

[13] S. Ghosal and N. Ray, "Deep deformable registration: Enhancing accuracy by fully convolutional neural net," *Pattern Recognition Letters*, vol. 94, pp. 81–86, 2017, ISSN: 0167-8655. DOI: `https://doi.org/10.1016/j.patrec.2017.05.022`. 23

[14] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680. 13

[15] H. Greenspan, B. van Ginneken, and R. M. Summers, "Guest editorial deep learning in medical imaging: Overview and future promise of an exciting new technique," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1153–1159, May 2016, ISSN: 0278-0062. 23

[16] K. Gregor and Y. LeCun, "Learning fast approximations of sparse coding," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, Omnipress, 2010, pp. 399–406. 7

[17] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746. 8

[18] T. He, H. Mao, J. Guo, and Z. Yi, "Cell tracking using deep neural networks with multi-task learning," *Image and Vision Computing*, vol. 60, pp. 142–153, 2017, ISSN: 0262-8856. 23

[19] S. G. Henderson and B. L. Nelson, "Stochastic computer simulation," *Handbooks in Operations Research and Management Science*, vol. 13, pp. 1–18, 2006. 19

[20] K. Hornik, M. Stinchcombe, and H. White, "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks," *Neural networks*, vol. 3, no. 5, pp. 551–560, 1990. 14

[21] P. Hu, B. Shuai, J. Liu, and G. Wang, “Deep level sets for salient object detection,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 540–549. DOI: `10.1109/CVPR.2017.65`. 24

[22] M. Jaderberg, W. M. Czarnecki, S. Osindero, O. Vinyals, A. Graves, and K. Kavukcuoglu, “Decoupled neural interfaces using synthetic gradients,” *CoRR*, vol. abs/1608.05343, 2016. arXiv: `1608.05343`. [Online]. Available: `http://arxiv.org/abs/1608.05343`. 4, 22, 32

[23] J. Ker, L. Wang, J. Rao, and T. Lim, “Deep learning applications in medical image analysis,” *IEEE Access*, vol. 6, pp. 9375–9389, 2018, ISSN: 2169-3536. DOI: `10.1109/ACCESS.2017.2788044`. 23

[24] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014. 36

[25] T. H. N. Le, K. G. Quach, K. Luu, C. N. Duong, and M. Savvides, “Reformulating level sets as deep recurrent neural network approach to semantic segmentation,” *IEEE Transactions on Image Processing*, vol. 27, no. 5, pp. 2393–2407, May 2018, ISSN: 1057-7149. 24

[26] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. P. Aitken, A. Tejani, J. Totz, Z. Wang, *et al.*, “Photo-realistic single image super-resolution using a generative adversarial network.”. 13

[27] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *arXiv preprint arXiv:1509.02971*, 2015. 13, 22

[28] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440. 26

[29] D. Marcos, D. Tuia, B. Kellenberger, L. Zhang, M. Bai, R. Liao, and R. Urtasun, “Learning deep structured active contours end-to-end,” *ArXiv e-prints*, Mar. 2018. arXiv: `1803.06329 [cs.CV]`. 24

[30] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`. 6

[31] T. A. Ngo, Z. Lu, and G. Carneiro, "Combining deep learning and level set for the automated segmentation of the left ventricle of the heart from cardiac cine magnetic resonance," *Medical Image Analysis*, vol. 35, pp. 159–171, 2017, ISSN: 1361-8415. DOI: `https://doi.org/10.1016/j.media.2016.05.009`. 25

[32] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017. 2, 6

[33] S. Pereira, A. Pinto, V. Alves, and C. A. Silva, "Brain tumor segmentation using convolutional neural networks in mri images," *IEEE Transactions on Medical Imaging*, vol. 35, no. 5, pp. 1240–1251, May 2016, ISSN: 0278-0062. DOI: `10.1109/TMI.2016.2538465`. 23

[34] H. Ravishankar, P. Sudhakar, R. Venkataramani, S. Thiruvenkadam, P. Annangi, N. Babu, and V. Vaidya, "Understanding the Mechanisms of Deep Transfer Learning for Medical Images," *ArXiv e-prints*, Apr. 2017. arXiv: `1704.06040 [cs.CV]`. 25

[35] N. Ray, S. T. Acton, and H. Zhang, "Seeing through clutter: Snake computation with dynamic programming for particle segmentation," in *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, Nov. 2012, pp. 801–804. 26

[36] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788. 7

[37] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99. 1, 13

[38] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *ArXiv e-prints*, May 2015. arXiv: `1505.04597 [cs.CV]`. 25–28

[39] F. R. Ruiz, M. T. R. AUEB, and D. Blei, "The generalized reparameterization gradient," in *Advances in neural information processing systems*, 2016, pp. 460–468. 20

[40] C. Rupprecht, E. Huaroc, M. Baust, and N. Navab, "Deep active contours," *CoRR*, vol. abs/1607.05074, 2016. arXiv: `1607.05074`. [Online]. Available: `http://arxiv.org/abs/1607.05074`. 24

[41] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," *ArXiv e-prints*, Mar. 2017. arXiv: `1703.03864 [stat.ML]`. 19, 20

[42] H.-C. Shin *et al.*, “Deep Convolutional Neural Networks for Computer-Aided Detection: CNN Architectures, Dataset Characteristics and Transfer Learning,” *ArXiv e-prints*, Feb. 2016. arXiv: `1602.03409 [cs.CV]`. 25

[43] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *arXiv preprint arXiv:1712.01815*, 2017. 1, 13

[44] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. T. Hurst, C. B. Kendall, M. B. Gotway, and J. Liang, “Convolutional neural networks for medical image analysis: Full training or fine tuning?” *IEEE Trans. on Med. Im.*, vol. 35, no. 5, pp. 1299–1312, May 2016, ISSN: 0278-0062. 25

[45] M. Tang, S. Valipour, Z. V. Zhang, D. Cobzas, and M. Jägersand, “A deep level set method for image segmentation,” *CoRR*, vol. abs/1705.06260, 2017. arXiv: `1705.06260`. [Online]. Available: `http://arxiv.org/abs/1705.06260`. 24

[46] Theano Development Team, “Theano: A Python framework for fast computation of mathematical expressions,” *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: `http://arxiv.org/abs/1605.02688`. 8

[47] S. Tokui, K. Oono, S. Hido, and J. Clayton, “Chainer: A next-generation open source framework for deep learning,” in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. [Online]. Available: `http://learningsys.org/papers/LearningSys_2015_paper_33.pdf`. 2

[48] D. Ulyanov, A. Vedaldi, and V. Lempitsky, “Deep image prior,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9446–9454. 13

[49] G. V, P. L, C. M, and et al, “Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs,” *JAMA*, vol. 316, no. 22, pp. 2402–2410, 2016. DOI: `10.1001/jama.2016.17216`. eprint: `/data/journals/jama/935924/joi160132.pdf`. 25

[50] A. Verma, “An introduction to automatic differentiation,” *Current Science*, pp. 804–807, 2000. 10

[51] *Yann lecun post on differentiable programming*, `http://archive.is/esn03`, Accessed: 6 Jan 2018 00:41:57 UTC. 2, 6

[52] C. Zotti, Z. Luo, A. Lalande, and P. Jodoin, “Convolutional neural network with shape prior applied to cardiac mri segmentation,” *IEEE Journal of Biomedical and Health Informatics*, pp. 1–1, 2018, ISSN: 2168-2194. DOI: `10.1109/JBHI.2018.2865450`. 24