

Derivative-free Optimization for Sequence-to-Sequence Models

BACHELOR THESIS

Lyubomira Dimitrova
dimitrova@cl.uni-heidelberg.de

Supervisor	Prof. Dr. Stefan Riezler
1 st Reviewer	Prof. Dr. Stefan Riezler
2 nd Reviewer	Prof. Dr. Artem Sokolov

February 15, 2020
Heidelberg

Contents

1	Introduction	1
2	Related work	4
3	Derivative-free optimization	5
3.1	Problem Statement	5
3.2	Gradient Estimators	6
4	Implementation	8
5	Application	13
5.1	Tasks and Data Generation	13
5.2	Experiments	14
5.2.1	Choice of smoothing parameter	14
5.2.2	Gradient estimators	16
5.2.3	Augmented Random Search (ARS)	17
5.2.4	Step size	19
5.2.5	Using pre-trained embeddings	20
5.2.6	Scaling to multiple workers	21
6	Discussion	22
7	Conclusion	24
References		

Abstract

Derivative-free optimization (DFO) is an alternative to traditional, gradient-based, approaches and to more recent reinforcement learning (RL) methods. Especially suitable for black-box scenarios or when optimizing a non-differentiable function, it is interesting to measure the usefulness of DFO for problems habitually tackled with backpropagation.

One such class of problems, called sequence-to-sequence (**seq2seq**), is a staple in the natural language processing (NLP) field, unifying tasks like machine translation, text summarization and speech recognition.

In this bachelor's thesis, I implement DFO from scratch and attempt to solve three simple sequence-to-sequence tasks represented by JoeyNMT models. I find that the simplicity of DFO allows highly configurable implementations, and describe my approach to implementing DFO for **seq2seq** in detail. In the application phase I experiment with hyperparamers, gradient estimation strategies and optimizers in order to study their effect on the training process.

My results show that estimating the gradient with symmetric function evaluations leads to stable improvement in validation set rewards, contrary to other gradient estimation techniques. Furthermore, I find that the choice of smoothing parameter is highly relevant to the success of training. Although two promising algorithmic enhancements led to a decrease in performance for the three tasks, using the ADAM optimizer to compute the update step yielded a small improvement in validation rewards over the simple SGD optimizer. Using pre-trained embeddings under ADAM proved to be beneficial as well. Unfortunately, my attempt to efficiently parallelize the algorithm proved unsuccessful, greatly limiting this method's applicability in practice. Finally, I propose some interesting directions for further research and experiments.

Резюме

Много от приложенията на естествената езикова обработка предполагат работа с редици - редици от думи (текст), редици от звуци (реч) и дори редици от изображения (видео). Проблеми, които преобразуват редици в други редици се наричат *sequence-to-sequence* проблеми. В днешно време повечето проблеми от този клас биват разрешени по следния начин. Първо, системата, или моделът, прочита входната редица и на базата на нея 'предрича' изходната редица. Тя от своя страна бива сравнена с истинската изходна редица, и грешката бива измерена. По методът за обратно разпространение на грешката, основан на частични производни, параметрите на моделът биват обновени, така че при следващо извикване на моделът със същата входна редица измерената грешка да е по-малка.

В бакалавърската си работа проучих дали *sequence-to-sequence* проблемите могат да бъдат разрешени чрез алгоритъм за оптимизация, който не използва производни (*derivative-free*, *DF*). Такъв метод е от полза когато грешката (или наградата) не е диференцируема, или когато моделът е черна кутия - тоест, когато нямаме информация за това как е пресметната грешката. Оценката на изходните редици при машинния превод е пример за това. Най-често се използват BLEU точките (начин за оценка на качеството на превода чрез измерване на припокриването между предречената и истинската изходни редици), които не са диференцируеми.

Имплементирах DF алгоритъм за оптимизация от нула, и проведох експерименти, за да реша чрез него три прости *sequence-to-sequence* проблема. Въпреки смесените резултати открих, че симетричният естиматор на градиента е най-подходящ за задачата, и че правилен избор на стъпката на алгоритъма могат да бъдат постигнати още по-добри резултати. Също използването на предварително оптимизирани репрезентации донесе подобрение във финалния резултат. За съжаление опитът ми да приложа ефективна паралелизация върху процеса беше неуспешен, което попречи на методът да се конкурира с традиционните методи за оптимизация чрез градиенти. Накрая, обобщавам откритията си и предлагам няколко възможности за развитие на метода за разрешаването на този тип проблеми.

1 Introduction

Many problems in the field of natural language processing (NLP) can be reduced to the transformation of a sequence to another sequence. These problems are called *sequence-to-sequence* problems, where the sequence can be any temporally dependent set of objects, for example a text, a video, a time series, an audio file. Examples of sequence-to-sequence problems include speech recognition, machine translation, question answering, image captioning.

Deep feed-forward neural networks can solve such problems only if the input and output sequences are of fixed lengths, which greatly limits their use in practice. Sequence-to-sequence (`seq2seq`) models (Sutskever et al., 2014; Cho et al., 2014) were developed to overcome this limitation. In their most basic form, `seq2seq` models are deep learning models based on recurrent neural networks (RNNs). The simplest model architecture consists of an encoder-RNN which learns a single vector representation of the input sequence (see vector \mathbf{c} , Figure 1), and a decoder-RNN which transforms that representation to an output sequence, generating one token at a time. Training such models usually consists in maximizing the likelihood of the input, also called maximum likelihood estimation (MLE). In practice, `seq2seq` models are largely optimized by calculating the gradient of a differentiable error function called a *loss* function, and using that gradient to update the model parameters, for instance with gradient descent.

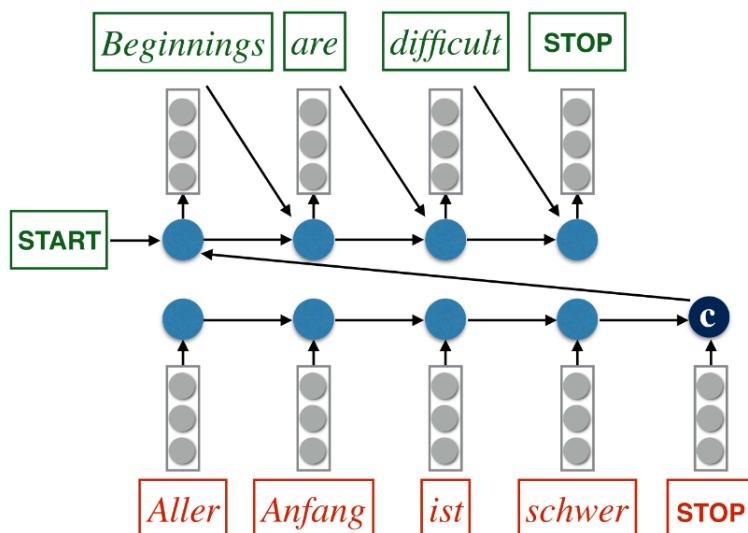


Figure 1: Encoder-Decoder model for Machine Translation. Source: Chris Dyer, "Modeling Sequential Data with Recurrent Networks" <http://1xmls.it.pt/2016/1xmls-d12.pdf>

Quite often, the cross-entropy loss is used, which measures the error between prediction and desired output one token at a time, and averages these errors to obtain the full sequence error.

In many cases, however, the loss function is distinct from the metric we use to evaluate how good a trained model is. For instance, in machine translation (MT) we might not necessarily need to generate the exact translation reference in order to produce a good translation of an input sequence. Instead, we want a certain amount of overlap between the model’s prediction and the reference translation(s). One such overlap measure is the BLEU-score (Papineni et al., 2002), which is even today the de facto standard for evaluating MT.

The question arises of whether we can use the actual evaluation metric (e.g. the BLEU-score) to train a model, without using another function as a proxy. However, since the BLEU-score and related metrics are non-differentiable, we cannot easily use gradient-based methods like backpropagation for training the model. In fact, we might need to view the entire model as a black box – as if only the final output score is visible.

Such approaches, called black-box, or derivative-free, optimization (DFO), have recently been used to solve continuous robotic control problems simulated in MuJoCo¹ (Todorov et al., 2012), for example learning bipedal walking. This difficult set of problems were previously only solvable with reinforcement learning (RL) methods, but in recent years derivative-free optimization techniques have managed to provide ‘a scalable alternative to reinforcement learning’ (Salimans et al., 2017), as well as ‘a competitive approach to reinforcement learning’ (Mania et al., 2018). However, the suitability of DFO for other types of problems, especially those easily solved with gradient information, is questionable. A well-known disadvantage of most DF methods is that their convergence speed scales with the (effective) dimensionality of the function they optimize (Nesterov and Spokoiny, 2017; Sokolov et al., 2018; Wang et al., 2016). This limitation is often a bottleneck when it comes to using such methods in practice. Nevertheless, interest in DFO has been rising in the last decade (Larson et al., 2019, Figure 1.1.), with numerous strategies for overcoming this limitation, owing to these methods’ simplicity, parallelizability, and suitability when only function evaluations are available.

In this thesis, I apply DFO to three simple `seq2seq` tasks to study the suitability of DF methods for solving this class of problems. My objectives are as follows:

¹<https://gym.openai.com/envs/#mujoco>

1. Implementation

- Implement derivative-free optimization from scratch, based on the pseudocode and insights of recent DFO papers (Salimans et al., 2017; Mania et al., 2018). Apart from the basic algorithm, optional algorithmic enhancements should be available and configurable, for example parallelization, custom model initialization and update strategies.
- Provide detailed documentation.

2. Application

- Using model architectures provided by JoeyNMT² (Kreutzer et al., 2019), a framework for neural machine translation, apply DFO to three simple seq2seq problems, namely `copy`, `reverse`, and `sort`. Focus on the training process and how different scenarios and hyperparameters affect it.

In the next section I summarize the findings of several papers on DFO, which serve as the core references for this thesis. In addition, while the training of seq2seq models with DF methods remains (to my knowledge) unexplored, there are similarities to reinforcement learning (RL) approaches. For this reason, the next section also contains a summary of applying RL to seq2seq learning tasks.

²<https://github.com/joeynmt/joeynmt>

2 Related work

I use two papers as the core papers for this thesis. The work of Salimans et al. (2017) proposes an evolutionary algorithm called Evolution Strategies (ES) as an alternative to RL on the MuJoCo tasks. Evolutionary algorithms model their methods on biological evolution, solving a given optimization problem by generating a ‘population’ of candidate solutions (‘mutations’), evaluating their ‘fitness’, and selecting the ‘fittest’ individuals for ‘reproduction’. In practice, Salimans et al.’s ES optimizes the parameters θ of a neural network by generating and evaluating a population of candidates $\hat{\theta}_i$, where the candidates are Gaussian noise perturbations of the original θ . ES is found to have several advantages to RL on the MuJoCo tasks, as well as being highly parallelizable, solving the Humanoid³ task in 10 minutes on 1440 CPUs. Mania et al. (2018) respond to ES with a simple random search algorithm that uses linear models to achieve results better than or comparable to ES on the MuJoCo tasks. The authors report 15 times higher computational efficiency than ES, necessitating a lower degree of parallelization.

In the literature, DF methods are often compared to RL benchmarks, which is why a comparison to seq2seq RL is important. An excellent summary of deep RL approaches and their application to improving seq2seq learning is given in Keneshloo et al. (2019). RL has been used for seq2seq problems as an attempt to bridge the discrepancy of model behaviour at train and test time (see *exposure bias*, Ranzato et al., 2015). Rather than training seq2seq models from scratch, however, RL approaches are used for fine-tuning the performance of traditionally trained models, owing to the fact that RL training is time consuming (Keneshloo et al., 2019). For example, RL can be applied to a pre-trained model, or RL rewards can be incorporated into the standard cross-entropy loss and scheduled to become more important in the course of training. Keneshloo et al. (2019) also give a summary of RL-based seq2seq applications in their Table IV.

Further related works deal with orthogonality of the exploration (Choromanski et al., 2018), exploiting sparsity patterns to reduce the number of perturbed dimensions (Sokolov et al., 2018), and DFO in high dimensions when the underlying problem dimensionality is low (Wang et al., 2016; Qian et al., 2016).

³<https://gym.openai.com/envs/Humanoid-v2/>

3 Derivative-free optimization

Derivative-free optimization has many names in the literature - gradient-free, zeroth-order, and black box optimization are some of them. The definition of DFO often depends on the problem one wants to solve and the particular DF method used. In the next pages, I detail the definition used in Choromanski et al. (2018), and explain it in the context of seq2seq models.

3.1 Problem Statement

We define the following maximization problem objective:

$$\max_{\theta} f(\theta), \text{ where } f(\theta) := \mathbb{E}_x [F(x, \theta)] \quad (1)$$

$\theta \in \mathbb{R}^d$ are the model parameters, e.g. the weights of a neural network, F is a reward function parametrized by θ and evaluated at point x , and \mathbb{E}_x corresponds to the expectation, or mean, of the rewards of all training instances $x \in \mathcal{X}$.⁴ As such, the objective $f : \mathbb{R}^d \rightarrow \mathbb{R}$ aims to find θ that maximizes the expected total reward of \mathcal{X} w.r.t. θ . One could then search for the optimal θ with gradient methods, for example stochastic gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \nabla f(\theta_k) \quad (2)$$

where α is the step size, or learning rate. However, in some cases the gradient of f is not available, either because of non-smoothness at point x , or because only function evaluations are available. In such cases, we can smooth the objective (1) using Gaussian smoothing, as suggested in Nesterov and Spokoiny (2017). The smoothed version f_σ of f , also called the Gaussian approximation, is defined as:

$$f_\sigma(\theta) = \mathbb{E}_\epsilon [f(\theta + \sigma \epsilon)] \quad (3)$$

$\sigma \in \mathbb{R}_+$ is a smoothing parameter, and $\epsilon \sim \mathcal{N}(0, I)$ is an n -dimensional Gaussian random vector, i.e. Gaussian noise. The approximation of f is obtained by evaluating perturbations $\theta + \sigma \epsilon_i$ at a given point x and then averaging the noisy evaluations. Nesterov and Spokoiny (2017) further prove that the distance

⁴In practice, $f(\theta)$ is calculated with m samples $x \in \mathcal{X}$, and corresponds for instance to the total reward of a minibatch.

$\|\nabla f_\sigma(\theta) - \nabla f(\theta)\|$ can be bounded for every θ (see their Lemma 3). It is therefore feasible that any solution to the maximization problem

$$\max_{\theta} f_\sigma(\theta) \quad (4)$$

would be a solution of similar quality to the original problem (1). Since f_σ is smooth everywhere, we can define its gradient with respect to θ :

$$\nabla_{\theta} f_\sigma(\theta) = \frac{1}{\sigma} \mathbb{E}_{\epsilon} [f(\theta + \sigma\epsilon)\epsilon] \quad (5)$$

3.2 Gradient Estimators

A direct computation of the gradient (5) is intractible. Choromanski et al. (2018) introduce three gradient estimators for $\nabla_{\theta} f_\sigma(\theta)$, with ES in mind.

Vanilla gradient estimator

$$\hat{\nabla} f_\sigma(\theta) = \frac{1}{N\sigma} \sum_{i=1}^N f(\theta + \sigma\epsilon_i)\epsilon_i \quad (6)$$

In ES terms, N is the *population size*, and $(\epsilon)_i \sim \mathcal{N}(0, I)$ the perturbations used to create the candidates $\theta + \sigma\epsilon_i$ in the population. I prefer the term exploration directions (e.g. ϵ_i is one exploration direction), because it contributes to the understanding of the approach - the algorithm explores in different directions from the current θ in search of better return.

Forward finite difference (FD) gradient estimator

$$\hat{\nabla} f_\sigma(\theta) = \frac{1}{N\sigma} \sum_{i=1}^N (f(\theta + \sigma\epsilon_i) - f(\theta))\epsilon_i \quad (7)$$

Here, $f(\theta)$ acts as a baseline and is subtracted from $f(\theta + \sigma\epsilon)$. As we want to maximize $f_\sigma(\theta)$, it stands to reason that $f(\theta)$ grows during training, so the norm of the gradients produced by the vanilla gradient estimator will automatically grow as well. Instead, the forward FD estimator focuses only on improvement over the current parameters.

Antithetic gradient estimator

$$\hat{\nabla} f_\sigma(\theta) = \frac{1}{2N\sigma} \sum_{i=1}^N (f(\theta + \sigma\epsilon_i) - f(\theta - \sigma\epsilon_i))\epsilon_i \quad (8)$$

This estimator evaluates both ϵ_i and $-\epsilon_i$ in order to promote or discard the exploration direction ϵ_i . If the antithetical directions lead to a similar reward, their contribution to the gradient is close to 0. Conversely, if for instance $f(\theta + \sigma\epsilon) \gg f(\theta - \sigma\epsilon)$, the negative perturbation acts as a baseline similar to $f(\theta)$ in the forward FD estimator. Sehnke et al. (2010) state that employing symmetric exploration directions in the estimation is much more robust than simply using a baseline. Nevertheless, it is important to note that this gradient estimator incurs twice the number of function evaluations as the other two, which presents a problem if function evaluations are expensive.

Although these are the definitions provided by Choromanski et al. (2018), many modifications are possible when information about the problem itself is available. For instance, Mania et al. (2018) scale the antithetic gradient estimator by the standard deviation of the collected rewards, instead of by the smoothing parameter σ .

4 Implementation

While several different relatively recent Python implementations of DFO exist (e.g.^{5,6}), applying them directly to the task of seq2seq learning would have necessitated a significant amount of modifications to the code itself. For example, there are several tensorflow implementations, but the underlying deep learning framework of JoeyNMT is PyTorch. Another problem is that these implementations are tailored specifically to MuJoCo environments and, perhaps most importantly, usually sparsely documented.

For these reasons, I decided to implement DFO from scratch, drawing inspiration from different papers and implementations. My main goal was producing readable, well-documented, easily configurable code to tackle the problem of DFO for seq2seq problems. Both code and documentation are publicly available.⁷

In its core, my implementation follows closely the pseudocode of Augmented Random Search (ARS) (Mania et al., 2018). Despite the fact that Mania et al. use ARS to train only linear models, ARS has a few important advantages compared to Salimans et al.’s ES.

First, it does not rely only on efficient parallelization, which is strongly dependent on available hardware, and often difficult to implement. Secondly, ARS is less complex and computationally expensive than ES: it does not require virtual batch normalization or fitness shaping (Salimans et al., 2017, Section 2). A very basic pseudocode is given in Algorithm 1, with optional steps in square brackets. Relevant implementation details are given for each line of the algorithm:

- **Training configuration.** JoeyNMT models are built on the basis of YAML configuration files. They describe the datasets used for training, testing and development, the model configuration, and the (backpropagation) training process. While some of the training settings are relevant for DFO (e.g. batch size, learning rate, evaluation metric), many are redundant. I extend the configuration file to include some DFO-specific hyperparameters like the smoothing parameter σ , the number of exploration directions to evaluate (i.e. the population size), as well as the maximum number of workers to use. An example configuration file is available in the repository as well.

⁵<https://github.com/openai/evolution-strategies-starter/blob/master/>

⁶<https://github.com/sourcecode369/Augmented-Random-Search->

⁷<https://gitlab.cl.uni-heidelberg.de/dimitrova/dfoseq2seq>

Algorithm 1 Basic DFO Algorithm

Input: Model M , smoothing parameter σ , (starting) learning rate α , num. exploration directions N

- 1: Initialize M with parameters θ
 - 2: **for** $k := 0 \dots K$ **do** $\triangleright K$ iterations
 - 3: Sample minibatch x
 - 4: Generate exploration directions $(\epsilon)_{j=1}^N$
 - 5: **for** $j := 0 \dots N$ **do**
 - 6: Collect rewards $f(x, \theta + \sigma \epsilon_j)$ [and $f(x, \theta - \sigma \epsilon_j)$]
 - 7: Estimate $\hat{\nabla} f(x, \theta)$ using the collected rewards
 - 8: Update $\theta = \theta + s(\alpha) \hat{\nabla} f(x, \theta)$
-

- **Line 1: Model initialization.** The model can be (1) randomly initialized with parameters drawn from a distribution given in the configuration file, (2) initialized from a checkpoint, the path to which is given in the configuration file. The checkpoint can be either a backpropagation checkpoint, or a DFO checkpoint. An additional checkpoint can be given for initializing the embeddings only, i.e. with pre-trained embeddings, which is a common and useful practice in real NLP applications.
- **Line 3: Minibatching.** Mania et al. suggest using minibatches to reduce the variance of the gradient estimate. Of course, an optimal case would be to 'see' the entire training set during each iteration. However, evaluating the entire training set multiple times (as many as the number of exploration directions) is impractical. Instead, one DFO training iteration updates the parameters on the basis of a single minibatch. Iterating over minibatches is implemented with `torchtext`, the default batch size set to 256.
- **Line 4: Noise generation.** Both Choromanski et al. (2018) and Salimans et al. (2017) solve the problem of random exploration directions by generating one large noise matrix, with size $n \times n$ where n is the number of exploration directions. In order to reduce the amount of memory necessary, I generate sets of random perturbations on the go. Furthermore, Choromanski et al. (2018) propose Gaussian orthogonal exploration, i.e. ensuring that all the exploration directions are orthogonal. Practically, this involves a transformation of the original Gaussian matrix, which proved to be intractable in my experiments. Again, Choromanski et al. (2018) train linear models with much

fewer parameters. Even so, the orthogonalization of the random explorations directions is also possible with my implementation.

- **Line 6: Reward functions.** There are four `seq2seq` evaluation metrics shipped with JoeyNMT:

- BLEU (Papineni et al., 2002): originally developed for evaluation of machine translation, BLEU is based on modified n-gram precision; de facto standard in the field.
- chrF (Popović, 2015): character n-gram F-score.
- Token accuracy: Percentage of tokens in the hypothesis that are in the same position in the reference.
- Sequence accuracy: Percentage of correct sequences (e.g. sentences). Correct means equal to the corresponding reference.

All of these metrics are available as reward functions for DFO, although in experiments the BLEU reward was comparatively easier to learn from. BLEU has been further been used as a reward function in RL approaches to `seq2seq` learning (e.g. Ranzato et al., 2015). Implementation-wise, the reward function is separate from the DF training process, and bound to it after initialization. This design decision makes it possible to re-write or change the reward function without worrying about how the DFO is implemented.

- **Lines 5-6: Parallelization.** I implemented two types of optional parallelization, omitting them from Algorithm 1 for the sake of readability. The first approach is similar to the parallelization technique employed by Salimans et al. (2017). In their paper, every worker evaluates N different perturbed parameters, resulting in an ‘inflated’ number of exploration directions ϵ_i : Nw , where w is the number of workers. Consequently, the number of iterations is reduced to K/w . The second parallelization technique retains the original number of exploration directions N and iterations K , but parallelizes the reward function evaluations in Line 6. In early experiments, both strategies proved difficult for optimizing a JoeyNMT `seq2seq` model, mostly due to hardware limitations, e.g. available memory, number of cores. I explore whether parallelization is a viable alternative to the single-worker case in the next section.

- **Line 7: Gradient Estimation.** The choice of gradient estimator is left to the user. Default is the central differences (antithetic) estimator, but gradient estimation via the vanilla estimator or via forward finite differences is

available. In practice the weighted sums in formulas (6), (7), (8) can be computed with a matrix-vector product. For instance, for the vanilla gradient estimator, we can collect all rewards in a single vector $r \in \mathbb{R}^N$ with entries $r(\epsilon_i) = f(\theta + \sigma\epsilon_i)$, and view the explorations directions as a matrix $E \in \mathbb{R}^{N \times d}$, where every row vector corresponds to a single exploration direction ϵ_i :

$$r_{all} = [r(\epsilon_1) \ r(\epsilon_2) \ \dots \ r(\epsilon_N)] \quad \text{and} \quad E = \begin{bmatrix} \text{---} & \epsilon_1 & \text{---} \\ \text{---} & \epsilon_2 & \text{---} \\ \dots & & \dots \\ \text{---} & \epsilon_N & \text{---} \end{bmatrix}$$

We can then calculate the product $r_{all} \cdot E \in \mathbb{R}^d$, which equals the weighted sum used in the vanilla gradient estimator. I use $\epsilon_i[d]$ to access the d^{th} entry of exploration direction ϵ_i .

$$\begin{aligned} r_{all} \cdot E &= \left[\begin{pmatrix} r(\epsilon_1)\epsilon_1[1] \\ + r(\epsilon_2)\epsilon_2[1] \\ + \dots \\ + r(\epsilon_N)\epsilon_N[1] \end{pmatrix} \quad \begin{pmatrix} r(\epsilon_1)\epsilon_1[2] \\ + r(\epsilon_2)\epsilon_2[2] \\ + \dots \\ + r(\epsilon_N)\epsilon_N[2] \end{pmatrix} \quad \dots \quad \begin{pmatrix} r(\epsilon_1)\epsilon_1[d] \\ + r(\epsilon_2)\epsilon_2[d] \\ + \dots \\ + r(\epsilon_N)\epsilon_N[d] \end{pmatrix} \right] \\ &= + \left\{ \begin{bmatrix} r(\epsilon_1)\epsilon_1[1] & r(\epsilon_1)\epsilon_1[2] & \dots & r(\epsilon_1)\epsilon_1[d] \\ r(\epsilon_2)\epsilon_2[1] & r(\epsilon_2)\epsilon_2[2] & \dots & r(\epsilon_2)\epsilon_2[d] \\ \dots & & & \\ r(\epsilon_N)\epsilon_N[1] & r(\epsilon_N)\epsilon_N[2] & \dots & r(\epsilon_N)\epsilon_N[d] \end{bmatrix} \right\} = + \left\{ \begin{array}{l} r(\epsilon_1)\epsilon_1 \\ r(\epsilon_2)\epsilon_2 \\ \dots \\ r(\epsilon_N)\epsilon_N \end{array} \right\} \\ &= \sum_{i=1}^N r(\epsilon_i)\epsilon_i = \sum_{i=1}^N f(\theta + \sigma\epsilon_i)\epsilon_i \end{aligned} \quad \square$$

Following the approach of Mania et al. (2018), my implementation also allows for using only the top performing exploration directions for the gradient estimate, as well as optionally scaling by the standard deviation of the rewards instead of by σ .

- **Line 8: Step size $s(\alpha)$.** An initial learning rate must be given in the configuration file. The update step size (i.e. how much of the estimated gradient is used to update θ) can be computed by one of three optimizers,

chosen in the configuration file. The current possibilities are simple SGD with a fixed decay rate ≤ 1 , SGD with momentum, and Adam. I implement Momentum SGD and Adam similarly to the ES implementation⁸, but adapted for PyTorch, and extended with the simple SGD optimizer with decay. Since all of these optimizers move θ in the opposite direction of the gradient (i.e. minimization), they receive as input the negative of the estimated gradient.

- **Model outputs and output files.** Logging the training process is essential if we want to analyse it. Every model saves its output files to a directory specified in the configuration file. Another logging parameter is the validation frequency. How often a model checkpoint is saved is dictated by the validation frequency and a variable keeping track of the current highest validation set reward. A checkpoint is saved only if a higher validation set reward was achieved, and takes note of the model parameters and the current state of the optimizer so continuing training from a DFO checkpoint is possible. All console outputs are saved in a .txt file in the output directory, which apart from measured rewards, includes a copy of the configuration file for reference purposes. The output files of training include a list of all parameters θ over time, the gradients computed and update steps taken at every iteration, as well as a list of the validation rewards.

⁸https://github.com/openai/evolution-strategies-starter/blob/master/es_distributed/optimizers.py

5 Application

In this section, I apply my DFO implementation to three simple `seq2seq` tasks, namely `copy`, `sort` and `reverse`. I study the training process and note which scenarios and hyperparameter settings result in improvement in validation reward, training times and stability. All plots were drawn with `matplotlib` (Hunter, 2007).

5.1 Tasks and Data Generation

Algorithmic tasks like `copy`, `sort` and `reverse` have been used for evaluating novel (gradient-based) approaches to `seq2seq` problems (Dehghani et al., 2018; Kaiser and Sutskever, 2015). Despite their simplicity, I focus on these tasks for two reasons. First, solving e.g. `reverse` with backpropagation requires an encoder-decoder model with attention with more than 30000 parameters. This dimensionality is more than enough to present a challenge for DFO and to test the algorithmic enhancements proposed in the literature. Secondly, a comprehensive study of DFO for `copy`, `sort` and `reverse` can serve as a stepping stone to applying DF methods to difficult problems like machine translation, either because of successful experiments or due to lessons learned.

Table 1 defines the input and output sequences of each of the tasks. Generating the train, validation, and test sets was done with JoeyNMT, using the default settings:

- **Size:** 50000 examples for training, 1000 for testing and validation each.
- **Maximum sequence length:** 25 for the training set, 30 for the validation and test sets.
- **Vocabulary:** The natural numbers in the range [0, 50].

Task	Input	Output
<code>copy</code>		The same sequence.
<code>sort</code>	Sequence of tokens (e.g. numbers).	The sorted input sequence.
<code>reverse</code>		The input sequence in reverse order.

Table 1: Input and output sequences for three algorithmic `seq2seq` tasks.

5.2 Experiments

One of the aims of this thesis was implementing a highly configurable DFO framework. Because of this, there are numerous possible configurations to choose from. I focus on the following experiments for several reasons. First, I avoid complexity without reason, and always test the simplest alternative first, for example pure SGD is tested before SGD with momentum. Secondly, I concentrate on configurations used in the core reference papers, since they performed well, though on a different type of task.

For the experiments, the main model architecture is fixed, and the same for all three tasks. The JoeyNMT model consists of a single-layer bidirectional LSTM encoder with 32 hidden units, Luong attention (Luong et al., 2015), and a single-layer LSTM decoder. Although the configuration file includes a small dropout, DFO training is based on function evaluations, so dropout is disabled before training. The number of model parameters (weights + biases) is 31904. References to ‘the model’ pertain to this model.

Pre-training with cross-entropy Applying DFO to a randomly initialized model predictably resulted in an inability to move past a zero reward. For this reason, I trained a model for each of the tasks using the configuration above. The model is optimized through backpropagation of the cross-entropy loss. During training, checkpoints of the models were saved every 200-500 iterations if there was an improvement in the validation set BLEU-score. A final score of 90-95 BLEU on the validation sets was reached for all tasks. Of the three, `sort` proved to be the most difficult to solve with backpropagation of the cross-entropy loss. On average, the same model needed twice as many iterations to reach a certain validation set BLEU on the `sort` task compared to `copy` and `reverse`. Still, training times were negligible.

5.2.1 Choice of smoothing parameter

First, I run the simplest possible set of experiments in order to judge how, and whether, the choice of σ affects learning. I consider the values [0.01, 0.05, 0.1]. In most implementations, the default value of σ is set to 0.1. For each task, I start DFO from a pre-trained model with validation set reward of 35.2 BLEU for the `copy` task, 29.6 for the `reverse` task, and 33.1 for the `sort` task. The models were trained for 500 iterations, so that every training example was seen 2-3 times.

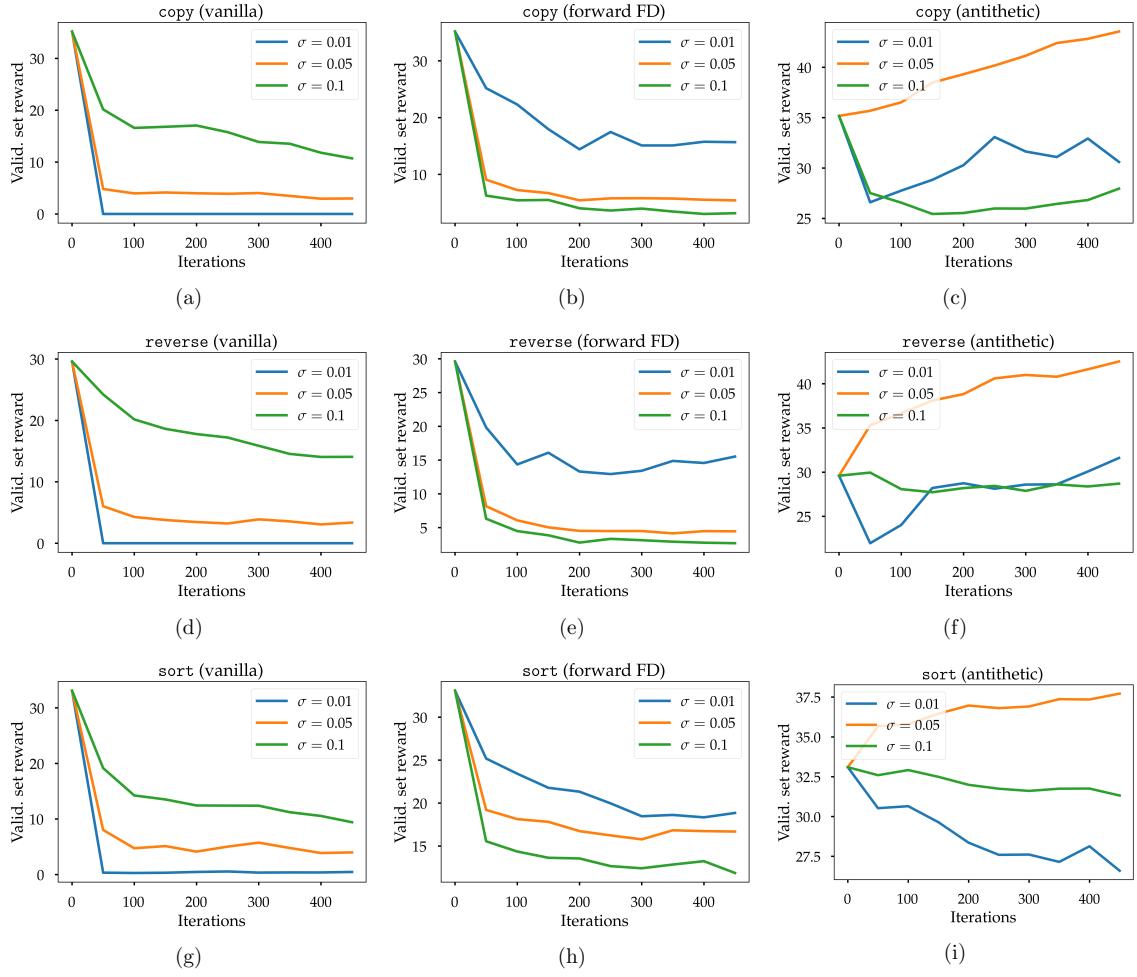


Figure 2: **copy**, **reverse** and **sort** with constant σ . 450 iterations, single worker, minibatch size 256, 50 exploration directions and constant learning rate $\alpha = 0.001$.

On average, the training process itself (without time needed for initialization and saving outputs) took 1.5 hours on GPU. While training longer achieved higher rewards, my aim is not obtaining perfect reward, but exploring what can be learned by a DFO model in a reasonable amount of time. I measure rewards on the validation set only once every 50 iterations. Training rewards are not available because the model evaluates only the perturbed parameters, not θ itself. Figure 2 shows the validation set reward at different iterations for the first set of experiments. The results were averaged over three runs with different random seeds for this and all other experiments.

Rather surprisingly, only the models with $\sigma = 0.05$ and antithetic gradient estimation achieved any improvement over the starting point. An important observation is that, with few exceptions, the first fifty iterations are the most important to the overall success of training. This is especially true for the vanilla gradient estimator

(Figures 2a, 2d, 2g), whose performance decreases the smaller σ is set.

The disappointing results of the forward FD gradient estimator (Figures 2b, 2e, 2h) are mostly likely due to the fact that candidates $\theta + \sigma\epsilon$ of better quality than the original θ were rarely produced. Without the stabilising antithetic exploration direction, $f(\theta + \sigma\epsilon) - f(\theta)$ was usually negative, causing the forward FD gradient estimator to consistently propose steps in the negative direction $-\epsilon$. Contrary to the vanilla approach, here small σ values were better, i.e., caused a less steep decrease of validation set reward.

Finally, the antithetic gradient estimator (Figures 2c, 2f, 2i) with $\sigma = 0.05$ achieved an improvement of the validation set reward of 5-10 BLEU for all three tasks.

For the rest of this thesis, I refer to these experiments as the base experiments.

5.2.2 Gradient estimators

The previous section compares gradient estimators on the basis of validation set reward. A further comparison of the gradient estimators can be found in Figure 3, which depicts the average norm of gradients produced by the three gradient estimators on the `reverse` task, where very different behaviors can be observed.

Generally speaking, the smaller the value of σ , the larger the gradient, since the normalizing part of the estimator grows, i.e.

$$\lim_{\sigma \rightarrow 0} \frac{1}{N\sigma} = \infty, \quad (9)$$

where the number of exploration directions N is constant. This is visible in the ranges of the y-axes and quite important considering that the models all start from the same point and follow the same data trajectory (minibatching with `torchtext` is deterministic when the same seed is used). It is obvious that the initial estimates of the forward FD and vanilla estimators have lasting consequences for the entirety of the training process, while the antithetic estimator proposes gradients of (comparatively) similar norm at every step.

A larger σ could make the vanilla estimator a viable alternative, while the same might be true for using the forward FD estimator with smaller σ values. However, as I aim to test a variety of scenarios in my experiments, obtaining positive results with all three gradient estimators is not a priority. Furthermore, the core reference papers of this thesis (Mania et al., 2018; Salimans et al., 2017) employ the antithetic

variant of gradient estimation, to which their variance reduction techniques are applied. Therefore, I leave the investigation of other gradient estimators for future work.

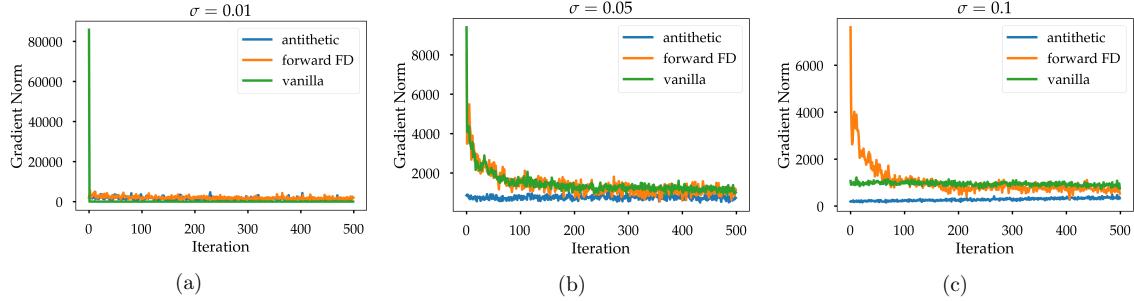


Figure 3: Gradient norms during training for the `reverse` task, with different σ values. See Figure 2 for training specifications.

5.2.3 Augmented Random Search (ARS)

In Section 3 I pointed out that the given definitions of the gradient estimators are often modified in practice in order to tailor them to a particular problem. Two examples of this are proposed in the ARS algorithm of (Mania et al., 2018):

1. Scaling the gradient by the standard deviation of the collected rewards, instead of by σ .
2. Incorporating only the top performing exploration directions ϵ_i into the weighted sum, i.e. the ones that achieve highest rewards $f(\theta \pm \sigma\epsilon)$.

A third technique used in ARS is normalization of the inputs. This makes sense for the MuJoCo tasks, where inputs are feature vectors where each feature has a

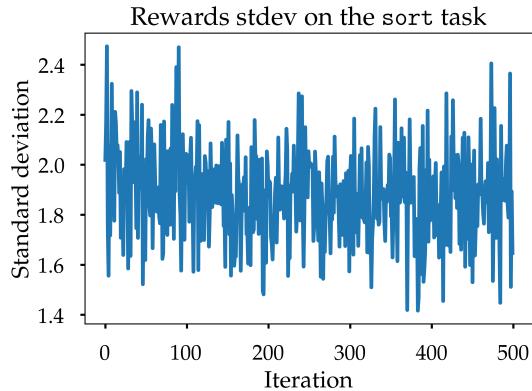


Figure 4: Standard deviation of the rewards collected at every iteration on the `sort` task.
 $\sigma = 0.05$, antithetic gradient estimator.

different value range⁹, but not in a seq2seq context where the input consists of one-hot vectors, and meaningful representations (i.e. embeddings) are learned in the course of training.

Scaling by the standard deviation was motivated by the rising standard deviation of rewards during training (Mania et al., 2018, see Figure 1). Following this example, I plot the reward standard deviation σ_R during learning the `sort` task (Figure 4). The behaviour of the standard deviation differs greatly from the values Mania et al. report for the Humanoid task. As such, I expect this modification to the gradient estimate to perform worse on the three seq2seq tasks. The second idea is a way of discarding badly performing exploration directions, which, intuitively, would lead to larger gradients. It is easy to see how the combination of these two strategies could lead to favourable results - the effect of each modification balances the other in the gradient estimation.

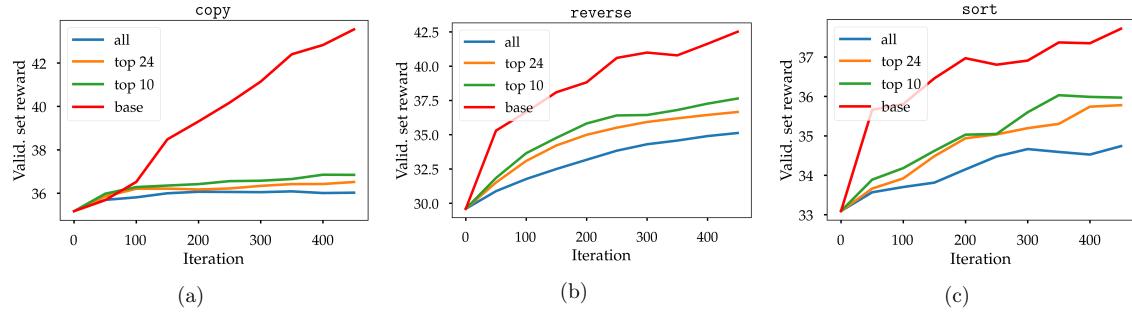


Figure 5: Comparison of three ARS scenarios (all explorations directions, top 24 and top 10 performing exploration directions; scaling by the standard deviation) with the base experiment. $\sigma = 0.05$, antithetic gradient estimator.

Finally, I train an ARS-based model for each of the three tasks and plot the achieved validation set rewards in Figure 5. Above all, I compare the performance to that of the base experiments. The results clearly show that the ARS modifications do not lead to a higher validation reward. Rather, they are pointedly worse, especially when applied to the `copy` task. Within the ARS experiments, the best model for all three tasks used only the ten exploration directions with best return $f(\theta \pm \sigma\epsilon)$ in the gradient estimate. Experiments with top performing directions without scaling by the standard deviation are not shown, but also yielded no improvement over the base experiment. Scaling by the standard deviation as strategy on its own predictably lead to the most significant decrease in performance.

⁹See for instance the state description of the Humanoid task: <https://github.com/openai/gym/wiki/Humanoid-V1>

5.2.4 Step size

The estimation of the gradient is only one part of updating the model parameters. Another is the step size, computed by one of the three optimizers mentioned in Section 4. All the experiments until now used a constant learning rate. However, a dynamically computed step size tends to boost training flexibility in cross-entropy based optimization. One should then pose the question of whether such a strategy is beneficial for DFO as well.

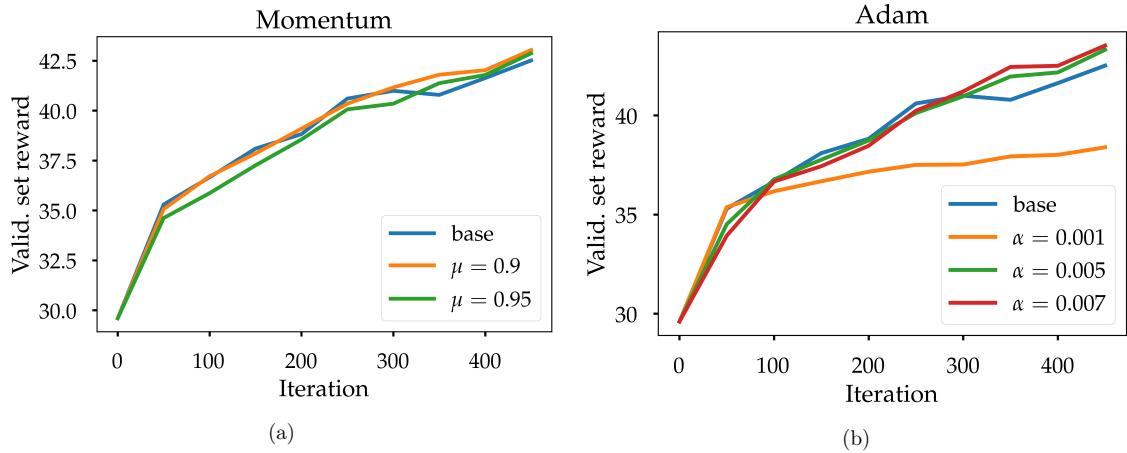


Figure 6: Using SGD with momentum and ADAM, validation set rewards compared to the base experiment. $\sigma = 0.05$, antithetic gradient estimator.

A larger step size could either speed up learning or drive it in an undesirable direction too quickly to recover. In Figure 6 I compare the achieved validation reward on the `reverse` task when using Momentum-SGD (6a) and ADAM (6b), compared to the base experiment with constant learning rate SGD. The results were similar across the other two tasks. For the momentum experiments, I try out two values used often in practice, and keep the starting learning rate $\alpha = 0.001$. For ADAM, I noticed that a higher starting learning rate performed better, since otherwise the updates were very small, slowing down training. The results are comparable to the base experiment, with a slight increase in validation reward. However, a difference can be seen in the update ratios. The update ratio denotes how large the update step is as a percentage of the weights. For an update step $\mathbf{s}_k \in \mathbb{R}^d$ computed at iteration k , it takes the form:

$$\text{ratio} = \frac{\|\mathbf{s}_k\|}{\|\theta_k\|}, \quad (10)$$

where $\|\mathbf{x}\|$ is the Euclidean norm of vector \mathbf{x} . In Figure 7 the update ratio of

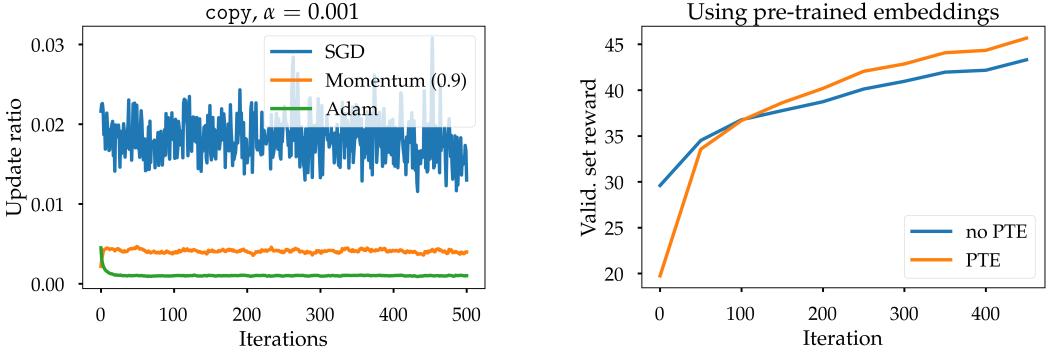


Figure 7: Update ratios at every iteration on the copy task. $\sigma = 0.05$, antithetic gradient estimator.

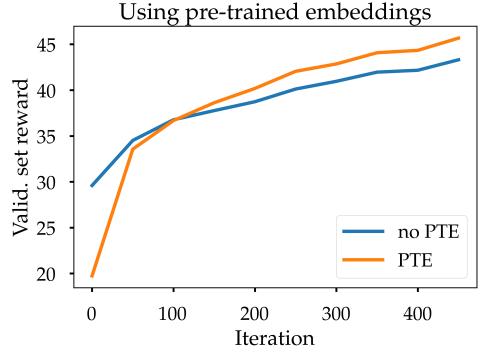


Figure 8: Validation set rewards when using pre-trained embeddings on the reverse task. ADAM with $\alpha = 0.005$.

the copy task under different optimizers is plotted. The “rough heuristic” for the update ratio when using backpropagation is that it should be around 0.001¹⁰. We can see that the ADAM optimizer quickly converges to this ratio, if a little lower. SGD with momentum proposes larger update steps, and standard SGD fluctuates strongly, which is to be expected since the update step size is proportional to the computed gradient. It is interesting to note that ADAM proposes 20 times smaller updates than SGD without momentum, but performs just as well. Furthermore, the update step was not more difficult to compute with ADAM, and training times were not affected negatively.

5.2.5 Using pre-trained embeddings

One of the model initialization methods in my implementation is loading pre-trained embeddings. In the model, $\sim 6\%$ of the parameter weights are part of an embedding matrix - either in the encoder or in the decoder. When using pre-trained embeddings, the entries of ϵ_i corresponding to embedding entries in θ are masked, i.e. set to zero. This ensures that the embeddings are not trained. Figure 8 shows that using pre-trained embeddings may result in a lower starting validation reward, but in the course of training manage to yield better performance. The embeddings for this experiment were loaded from the best backpropagation checkpoint of the reverse task (BLEU-score 94.4), while the rest of the weights were again initialized from the 29.6 BLEU checkpoint. Both PTE and no PTE models were optimized with ADAM ($\alpha = 0.005$).

¹⁰<http://cs231n.github.io/neural-networks-3/#ratio>

5.2.6 Scaling to multiple workers

As already mentioned, Salimans et al. (2017) and Mania et al. (2018) scale their approaches to multiple CPUs in order to solve several tasks in the MuJoCo simulator. In my experiments with parallelizing DF seq2seq learning, I encountered a serious limitation - in multithreaded scenarios the simultaneous evaluation of candidates $\theta \pm \sigma\epsilon$ often caused instability of evaluations or even program termination due to race conditions¹¹. A solution to this problem was cloning the JoeyNMT model for every candidate- θ evaluation (in the case of parallel function evaluations) or for every worker (in the case of parallel workers evaluating N candidates each). However, in the first case, this resulted in a certain time and memory overhead - for the antithetic gradient estimator, the model was cloned $2N$ times in every iteration, where N is the number of exploration directions. I observed a very slight speedup when using two workers compared to training sequentially with one worker, and increasing training times when using more than two workers. In the second case, the model was cloned once for every worker, but due to communicating larger tensors (passing N exploration directions to each worker, returning an array of computed rewards), training times were again similar to the single-worker case. It is unclear whether the behaviour reported by Salimans et al. could be achieved with better hardware, or efficient parallelization is beyond my implementation. On the subject of training times, without parallelization it was possible to use GPUs for training. The same model and DFO configuration needed about 13 times longer on CPU than on GPU.

I tested various DF training scenarios on a total of 500 iterations. Despite considering different strategies for improving and speeding up learning, I achieved only limited improvement over the simplest model using the antithetic gradient estimator.

¹¹A race condition occurs when different processes of threads attempt to change the shared data simultaneously. See e.g. Netzer and Miller (1992).

6 Discussion

The experiments in Section 5 showed clearly that the problem of applying DFO to `seq2seq` poses a challenge. I tested out several training scenarios and varied hyperparameters with limited success.

My main findings are the following:

- Out of the three proposed methods for gradient estimation, the antithetic gradient estimator exhibits the most consistent behaviour, producing gradients of similar norm during the entire training process. Less volatile than the vanilla and forward FD estimators in the first 50 iterations, it achieves an improvement over the initial validation reward that other, more complicated gradients estimation techniques failed to reach.
- DFO algorithmic enhancements developed for the solving of a particular problem generally do not translate well to other problems as is. The ARS algorithm proposed by Mania et al. (2018) performed excellently on the MuJoCo tasks, as it was intended to do, but caused a significant decrease in performance in my experiments with `seq2seq` models.
- Standard optimizers like ADAM and momentum-SGD can be used successfully on gradient estimates. ADAM in particular stabilizes the update ratio and achieves a higher validation reward than SGD.
- Pre-training a part of θ , in this case the embeddings, proved beneficial to the training process, pushing the validation reward higher in all three tasks. This was true despite the fact that less than the embeddings constituted less than 6% of all model parameters.
- Without efficient parallelization, DFO is much slower than traditional back-propagation training, and not a viable alternative. In hindsight, deferring to an existing DFO implementation for the parallelization would have been sensible, as that was the part I struggled with most. I believe, however, that focusing on algorithmic enhancements in my experiments was the sensible choice, considering the hardware at my disposal.

Next, I propose and discuss some ideas outside of the scope of this bachelor’s thesis, that could potentially improve `seq2seq` DFO.

In most of the experiments in Section 5, I attempted to optimize while using all parameters - perturbing every entry in θ to obtain the candidates, as well as updating every weight, even when the updates were of magnitude 10^{-4} or less. Practically all of the DFO implementations I discovered in my research used this strategy. Nevertheless, in every iteration 15 – 30% of the entries of θ receive an update ≤ 0.001 . Discarding these entries from the computed update step could reduce update noise, especially when optimizing larger networks. The idea of discarding small weights bears some resemblance to the *lottery ticket hypothesis* (Frankle and Carbin, 2018), which claims that, in the same number of iterations, one can train only a subnetwork of the original neural network, achieving similar test accuracy. An in-depth application of the lottery ticket hypothesis to DFO is an exciting research avenue.

Another possible improvement in a similar direction is incorporating a learning-to-learn, or meta-learning, approach to the parameter perturbations. For instance, Chen et al. (2017) train optimizers based on RNNs that can optimize low-dimensional DF functions. For the random search case specifically, an optimizer that proposes which entries of θ should be perturbed to achieve an optimal improvement could be beneficial.

Lastly, applying DFO to optimize model architectures developed specifically for backpropagation may be constraining. For example, the JoeyNMT encoder-decoder models use soft attention instead of hard attention (Xu et al., 2015), due to the fact that hard attention is not differentiable and needs to be estimated with samples. Still, trained models using hard attention have been found to perform better (Shankar and Sarawagi, 2018), making it an attractive mechanism for DFO since gradients are not computed. Beyond designing new neural network architectures per hand, the possibility of *evolving* problem-specific architectures exists (Elsken et al., 2018; Liu et al., 2017).

7 Conclusion

I implemented DFO from scratch and applied it to three simple `seq2seq` tasks. Applying DFO to `seq2seq` models was a challenging task, and one interesting for many reasons. For one, the concept behind it was something entirely different from my experience with machine learning until now. Moreover, while the main idea behind DFO approaches is simple and intuitive, their execution opens a wide range of possibilities, allowing for creative, problem-specific solutions. Although the results to my experiments were far from overwhelming, I now have a much better understanding of the theory behind and the practical approaches to DFO, up to and including their implementation.

I look forward to furthering my knowledge on the subject and experimenting with the numerous new neural DFO approaches introduced every year.

References

- Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando De Freitas. Learning to learn without gradient descent by gradient descent. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 748–756. JMLR.org, 2017.
- Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.
- Krzysztof Choromanski, Mark Rowland, Vikas Sindhwani, Richard E Turner, and Adrian Weller. Structured evolution with compact architectures for scalable policy optimization. *arXiv preprint arXiv:1804.02395*, 2018.
- Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. *arXiv preprint arXiv:1807.03819*, 2018.
- Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.
- Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, abs/1803.03635, 2018. URL <http://arxiv.org/abs/1803.03635>.
- J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.
- Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.
- Yaser Keneshloo, Tian Shi, Naren Ramakrishnan, and Chandan K Reddy. Deep reinforcement learning for sequence-to-sequence models. *IEEE Transactions on Neural Networks and Learning Systems*, 2019.
- Julia Kreutzer, Joost Bastings, and Stefan Riezler. Joey NMT: A minimalist NMT toolkit for novices. *To Appear in EMNLP-IJCNLP 2019: System Demonstrations*, Nov 2019. URL <https://arxiv.org/abs/1907.12484>.
- Jeffrey Larson, Matt Menickelly, and Stefan M Wild. Derivative-free optimization methods. *Acta Numerica*, 28:287–404, 2019.
- Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *CoRR*, abs/1711.00436, 2017. URL <http://arxiv.org/abs/1711.00436>.
- Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015. URL <http://arxiv.org/abs/1508.04025>.
- Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

Yurii Nesterov and Vladimir Spokoiny. Random gradient-free minimization of convex functions. *Foundations of Computational Mathematics*, 17(2):527–566, 2017.

Robert HB Netzer and Barton P Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

Maja Popović. chrf: character n-gram f-score for automatic mt evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, 2015.

Hong Qian, Yi-Qi Hu, and Yang Yu. Derivative-free optimization of high-dimensional non-convex functions by sequential random embeddings. In *IJCAI*, pages 1946–1952, 2016.

Marc’Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. *arXiv preprint arXiv:1511.06732*, 2015.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Parameter-exploring policy gradients. *Neural Networks*, 23(4):551–559, 2010.

Shiv Shankar and Sunita Sarawagi. Labeled memory networks for online model adaptation. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

Artem Sokolov, Julian Hitschler, and Stefan Riezler. Sparse stochastic zeroth-order optimization with an application to bandit structured prediction. *arXiv preprint arXiv:1806.04458*, 2018.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando de Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044, 2015. URL <http://arxiv.org/abs/1502.03044>.