SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Gradientless Optimization for Language Model Finetuning

Cosku Baris Coslu

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Gradientless Optimization for Language Model Finetuning

Gradientenfreie Optimierung für Language Model Finetuning

Author:	Cosku Baris Coslu
Supervisor:	Prof. Dr. Georg Groh
Advisor:	Jeremias Bohn
Submission Date:	15.07.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.07.2024 Cosku Baris Coslu

Abstract

Transformer models pre-trained on large amounts of data have become the standard method of building powerful language models in Natural Language Processing (NLP) research. These models incorporate comprehensive information about language, allowing them to be fine-tuned for a wide variety of tasks using a smaller, task-specific dataset without substantial modifications on the architecture. This work aims to explore gradientless optimization methods that can be utilized in the context of language model fine-tuning. In particular, it investigates the feasibility of using direct search methods based on random perturbations of parameter tensors as an alternative to state-of-the-art first-order optimizers for the fine-tuning of pre-trained language models. We introduce a direct search method based on an adaptation of the Gradientless Descent (GLD) algorithm. Our method can fine-tune a DistilBERT model on the SST-2 dataset using less memory than an Adam optimizer in exchange for a small reduction in validation accuracy.

Contents

1 Introduction

Neural Networks are powerful machine learning models that have been shown to yield state-of-the-art results in the field of Natural Language Processing (NLP) (Goldberg 2016). Recently, the Transformer (Vaswani, Shazeer, Parmar, et al. 2017) has been adopted as a conventional architecture for creating language models. Transformer models such as BERT (Devlin, Chang, Lee, and Toutanova 2019) have been shown to learn comprehensive information about language when trained on unsupervised language modeling tasks with large amounts of data. Representations learned by these models are useful for a wide variety of tasks and the same pre-trained model can be fine-tuned for different tasks by adding an appropriate output layer and training it on a smaller, task-specific dataset.

A neural network model is trained by iteratively updating its parameters in order to minimize a loss function measuring the error between observed data and the model output. The standard method of training relies on the backpropagation algorithm (Rumelhart, G. E. Hinton, and Williams 1986) to calculate the gradient of the loss function w.r.t the parameters of the model. The gradient is used to find the descent direction and parameters are moved in that direction to minimize the loss.

In general, optimization algorithms that make use of the gradient are called first-order methods referring to the order of the used derivatives. There are also zeroth-order methods that only need to evaluate a function at different points in order to optimize it. Zeroth-order methods have proven to be useful in generating adversarial examples for machine learning systems (P.-Y. Chen, H. Zhang, Sharma, et al. 2017) as well as in automated machine learning research (Liu, Ram, Vijaykeerthy, et al. 2020). They can also be used when gradient calculation is infeasible or impractical (Liu, P.-Y. Chen, Kailkhura, et al. 2020).

Many zeroth-order methods such as ZO-SGD (Spall 1992), ZO-AdaMM (X. Chen, Liu, Xu, et al. 2019) and MeZO (Malladi, Gao, Nichani, et al. 2023) calculate an estimation of the gradient using finite difference approximations, which can then be used in the place of the gradient. As such, these algorithms are easily implemented by only slightly modifying common first-order methods (Liu, P.-Y. Chen, Kailkhura, et al. 2020). Other zeroth-order algorithms, such as Stochastic Three Points (STP) (Bergou, Gorbunov, and Richtárik 2020) and Gradientless Descent (GLD) (Golovin, Karro, Kochanski, et al. 2020), perform a search in a small radius around a parameter and update it to the best

candidate. Algorithms from this class are called direct search methods. Direct search methods are not often used for the purpose of training neural networks as they do not scale well to large-size problems (Kolda, Lewis, and Torczon 2003).

Training large neural networks can be costly in terms of memory. Reducing the memory usage can be beneficial to make research more accessible by making neural network training possible on systems with limited computational resources. Recent approaches reduce the model size by pruning neurons from the network (Cheng, M. Zhang, and Shi 2023), train smaller models with the help of larger pre-trained models using knowledge distillation (G. Hinton, Vinyals, and Dean 2015) or reduce the memory usage of fine-tuning by training only a small subset of parameters (Lialin, Deshpande, and Rumshisky 2023).

This work aims to reduce the memory consumption of fine-tuning by using a gradientless optimization method, eliminating the need for backpropagation. In doing so, we also want to investigate the potential of direct search methods to train neural networks. We introduce an algorithm based on GLD that can effectively bypass the limitations of direct search methods on problem size. Using our algorithm, we fine-tune a feed-forward network attached to a pre-trained DistilBERT model (Sanh, Debut, Chaumond, and Wolf 2020) on the SST-2 dataset (Socher, Perelygin, Wu, et al. 2013) and obtain a validation accuracy comparable to that of an Adam optimizer (Kingma and J. Ba 2014) while using less memory.

2 Background

The field of NLP comprises various computational tasks involving human language, such as language modeling, machine translation, sentiment classification and question answering. Current state-of-the-art results in the field are predominantly produced by neural network models (Goldberg 2016). This chapter provides the basic background for neural networks, introduces several neural network architectures with a focus on their utilization in NLP, surveys various gradient-based and gradient-free approaches to optimization and examines methods to rank network parameters on importance.

2.1 Neural Networks

Neural networks are a collection of powerful machine learning tools that see persistent use in NLP and yield many of the recent state-of-the-art results in the field. The concept of the neural network takes inspiration from the human nervous system: It models a network of basic computation units, called *neurons*, which are connected with each other in specific layouts similarly to the nerve cells in the brain (Du and Swamy 2019, Chapter 1.2). In particular, the concept originates from the McCulloch–Pitts Neuron (McCulloch and Pitts 1943), which, slightly modified, is commonly used up to present (Bielecki 2019).

2.1.1 The Neuron

A neuron in a neural network is a node that processes inputs from all incoming connections into a single output. Each of a neuron's inputs has an associated weight. The output is computed by taking the sum of all inputs multiplied by their weights, adding a bias term and applying a nonlinear *activation function* on the sum. Precisely, a neuron with n incoming connections (n inputs, which can be expressed as a row vector $x \in \mathbb{R}^{1\times n}$), computes

$$y = \sigma(\mathbf{x}\mathbf{w} + b) \tag{2.1}$$

where $\mathbf{w} \in \mathbb{R}^n$ is the vector of weights, $b \in \mathbb{R}$ is the bias term and $\sigma : \mathbb{R} \to \mathbb{R}$ is the nonlinear activation function. Figure 2.1 visualizes a neuron in the form of a computation graph.

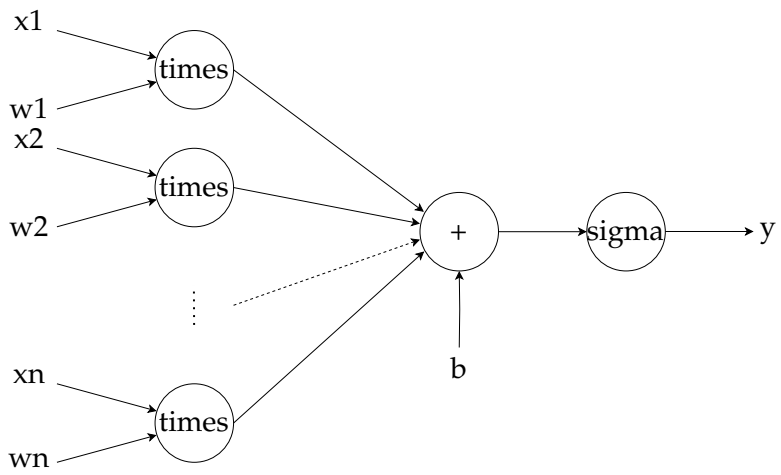

Figure 2.1: Computation graph of a neuron

The nonlinearity from the activation function is crucial for the computational power of the neural network, since without it, the network can only represent linear transformations of the input. Common activation functions are the sigmoid function,

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

the hyperbolic tangent (tanh) function

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

and the Rectified Linear Unit (ReLU) function

$$\text{ReLU}(x) = max(0, x)$$

.

The choice of activation function is usually made empirically and it has been established that in general, the ReLU function works better in neural networks than the presented alternatives (Goldberg 2016).

In the following, we will discuss key neural network architectures and their use-cases in NLP.

2.1.2 Feed-Forward Neural Networks

The simplest kind of neural network is the feed-forward network, where nodes are connected with no cycles (Jurafsky and Martin 2024). A feed-forward network is

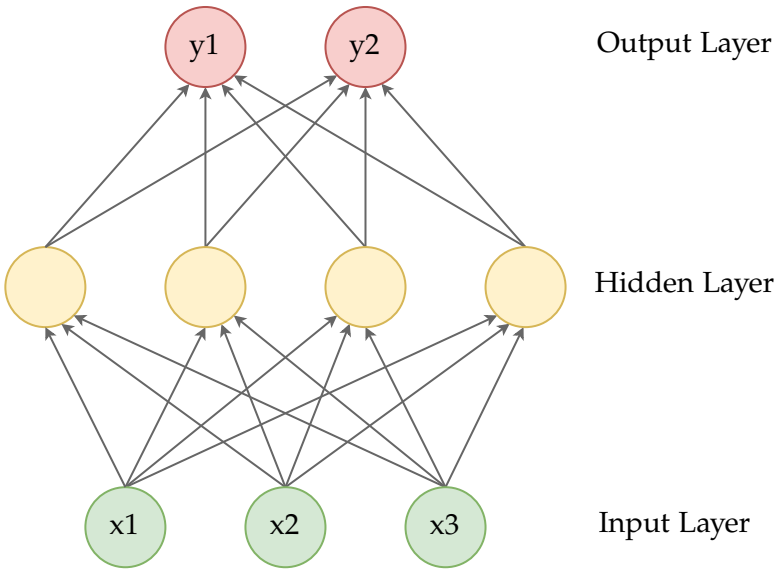

Figure 2.2: An example of a fully-connected feed-forward network with a single hidden layer. The network takes a 3 dimensional input and produces a 2 dimensional output. Nodes in the hidden layer apply a nonlinear activation function to their outputs.

usually arranged in layers: A layer for the inputs, a layer for the outputs and one or more *hidden layers* of neurons in between. Connections are in one direction, where the outputs from units in each layer are passed to units in the next higher layer and there are no connections between units of the same layer. A special case is the *fully-connected* feed-forward network, where each node is connected to all nodes of the next layer. Figure 2.2 shows an example of a fully-connected network with a single hidden layer. The visualized network performs two linear transformations: First, the input is transformed from 3 dimensions to 4 dimensions, then the result is transformed from 4 dimensions to 2 dimensions. The calculation performed by the network can be expressed concisely in the form of matrix-vector operations:

$$y = \sigma(\mathbf{x}\mathbf{W_1} + \mathbf{b_1})\mathbf{W_2} + \mathbf{b_2} \tag{2.2}$$

where $\mathbf{x} \in \mathbb{R}^{1\times 3}$ is the input, $\mathbf{W_1} \in \mathbb{R}^{3\times 4}; \mathbf{b_1} \in \mathbb{R}^{1\times 4}; \mathbf{W_2} \in \mathbb{R}^{4\times 2}; \mathbf{b_2} \in \mathbb{R}^{1\times 2}$ are the weights and biases for the first and the second linear transformations respectively, and $\sigma : \mathbb{R}^{1\times 4} \to \mathbb{R}^{1\times 4}$ is the nonlinearity of the hidden layer.

2.1.3 Neural Network Training

In order to fulfill a given task, a neural network needs to be trained on a dataset. The purpose of training is to find the ideal set of parameters (weights and biases) that best align the model with the observed data. We are interested in the *supervised learning* setting, where the dataset contains pairs (x_i, y_i) of input and output and the model is supposed to learn an underlying function of the data such that it produces output y_i given input x_i. This involves solving an optimization problem

$$\min_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) \tag{2.3}$$

where $\boldsymbol{\theta}$ are the model parameters, $\mathbf{x}$ and $\mathbf{y}$ are inputs and outputs from the dataset, and $\mathcal{J}$ is called the *objective function*. The goal of training is to find the parameters $\boldsymbol{\theta}$ that minimize the objective function. Different approaches to solve this optimization problem will be examined in Section 2.2.

Intuitively, $\mathcal{J}$ is a function that calculates the output $\hat{\mathbf{y}}$ of the model parameterized by $\boldsymbol{\theta}$ for the given inputs $\mathbf{x}$ and measures the error between the model output and the observed data. This error is also called *loss* and can be calculated using a loss function. Given true output (or *label*) $\mathbf{y}$ and model output $\hat{\mathbf{y}}$, a loss function $\mathcal{L}$ returns a scalar value for error. The objective of training is then minimizing the loss function

$$\mathcal{J}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) \tag{2.4}$$

The choice of loss function depends mainly on the task and the outputs expected from the model. For example, usually, the *cross-entropy* loss is used for classification tasks where the output is supposed to be interpreted as a probability distribution over k classes:

$$\mathcal{L}_{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{k} y_i \log(\hat{y}_i) \tag{2.5}$$

For so-called hard classification tasks where the label vector $\mathbf{y}$ contains a 1 for the correct class assignment t and 0 for every other class assignment, Equation 2.5 simplifies to

$$\mathcal{L}_{cross-entropy}(\hat{\mathbf{y}}, \mathbf{y}) = -\log(\hat{y}_t) \tag{2.6}$$

Typically, the model is trained for a certain number of iterations, where in each step loss is calculated and parameters are updated according to a certain optimization algorithm.

While training optimizes network parameters to align model outputs to observed data, a model is only good if it can also perform well on previously unobserved data. The ability of a model to perform well on unseen data is called *generalization* and is a central challenge in machine learning (Goodfellow, Bengio, and Courville 2016,

Chapter 5). A model's ability to generalize is measured by calculating loss on a subset of the available data called the *validation dataset*. This loss is not used to optimize the parameters, but as an estimate on how well the model generalizes. The subset used to train the model parameters is then called the *training dataset*.

A common issue when training a neural network model is *overfitting*, where the model performs well on the training dataset but does not generalize. One can discern that the model is overfitting when validation loss is high with a large difference to training loss. Typically, validation loss decreases together with training loss at the beginning of training. While training loss keeps decreasing as it approaches the minimum possible loss value, validation loss begins rising after a certain number of iterations as the model begins to fit the noise in the data rather than the underlying function, hurting its generalization (Du and Swamy 2019, Chapter 2.2). Thus, a simple way to ensure generalization is to stop training when validation loss begins to rise and take the model parameters from the iteration with the lowest validation loss.

The occurrence of overfitting is tightly connected to *capacity*, a model's ability to fit a wide variety of functions, i.e. how much a model can learn. For neural networks, capacity depends largely on the number of parameters and the number of output components (Widrow and Lehr 1990). That means capacity can be increased by adding more layers or neurons to the network architecture. When capacity is too large, the model learns the noise in the data and overfitting occurs. Conversely, when capacity is too small, *underfitting* may occur, which means the model cannot perform well on the training dataset. Ideally, model capacity is adapted such that neither overfitting nor underfitting occur.

One method to control a model's capacity without changing its architecture is to implement *regularization*. Regularization is the common name of strategies that modify the training algorithm with the intention of reducing validation loss, possibly at the expense of increased training loss (Goodfellow, Bengio, and Courville 2016, Chapter 5). Usually, a penalty term on the parameters called the regularizer is added to the objective function, meaning the training algorithm optimizes the (weighted) sum of loss and regularizer:

$$\mathcal{J}(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}) = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \cdot \Omega(\boldsymbol{\theta}) \tag{2.7}$$

where $\Omega : \mathbb{R}^n \to \mathbb{R}$ is the regularizer term and $\lambda \in \mathbb{R}$ is its weight. The regularizer can stem from specific kinds of prior knowledge on the data, or it can express a generic preference for one type of model over others.

A common regularization method is the L_2-regularization, often also called weight decay, where the regularizer term is the (squared) L_2-norm of the parameters:

$$\Omega(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_2^2 = \boldsymbol{\theta}^T \boldsymbol{\theta} \tag{2.8}$$

L_2-regularization encodes a preference for smaller weights. This helps the model's generalization under the assumption that the underlying function the model is supposed to learn is *smooth*, meaning small changes in the input do not cause large changes in the output. The weight λ can then be understood as balancing the trade-off between smoothing and minimizing the error (Reed, Marks, and Oh 1995).

2.1.4 Neural Networks in Natural Language Processing

In NLP, tasks often involve analyzing *documents*, collections of sentences and words. A document needs to be turned into a vector of real numbers before being given as input to a neural network. For that, it needs to be segmented into core components, which can be words, sub-words or some specific meaningful morphemes like *-est* or *-er* (Jurafsky and Martin 2024, Chapter 2). Each of these components are then assigned a number called a *token*. This process is called tokenization and is performed by applying a tokenizer function.

Given tokens as input, the neural network has to extract *features* of the document relevant for the task, such as words and part-of-speech tags (classification of words as noun, verb, etc.). Each such feature is represented as a vector called the *embedding* of the feature. While embeddings can be retrieved externally and fed into the network, they are often modeled as part of the network, resulting from an embedding layer, meaning representations of features are also learned during training and training will cause similar features to have similar vectors (Goldberg 2016).

The output is generated by applying transformations on the extracted combination of features as in Equation 2.2. Depending on the task, the network may also perform an output transformation. For example, for classification tasks, the softmax function is applied, which transforms the k-dimensional output into a probability distribution over k classes:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}} \tag{2.9}$$

2.1.5 The Transformer Architecture

The transformer architecture (Vaswani, Shazeer, Parmar, et al. 2017) has established itself as a core component of state-of-the-art deep NLP models. Language representation models based on transformers such as BERT (Devlin, Chang, Lee, and Toutanova 2019) have been shown to learn rich information about language which can be used to create state-of-the-art models for a wide range of tasks without task-specific modifications on the core architecture. In fact, a BERT model pre-trained on large amounts of data can

be fine-tuned for a given task by only adding a feed-forward layer on top and training it with a smaller dataset.

The main component of the transformer is the *attention* mechanism, a method to model dependencies between sequences over arbitrarily long distances. It has been used mainly in encoder-decoder architectures where through the two respectively-called parts, an input is first encoded into a fixed-length vector and that vector is then decoded into one output at a time. Such an architecture is often used for machine translation tasks (Bahdanau, Cho, and Bengio 2015) and the role of attention is to provide a context for the decoder such that it can focus on, or *attend to*, different parts of the source text relevant to the output it is currently producing. Essentially, the idea of attention is to evaluate the relevance of a focus of attention with each item of a set of other items by performing a series of comparisons (Jurafsky and Martin 2024, Chapter 9). We will give the exact computation of attention on the example of a self-attention layer, a core component of a transformer block.

Self-attention is a type of attention relating different positions of a single sequence. Given a sequence of input vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$, a self-attention layer produces output $\mathbf{y}_1, \ldots, \mathbf{y}_n$ with the same dimensionality where each $\mathbf{y}_i$ is the attention of the corresponding $\mathbf{x}_i$. The relevance of two items of the sequence is measured by some score function. The simplest score function is the dot product, which implements relevance as similarity:

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j \tag{2.10}$$

The more similar the two vectors being compared, the higher the score. With $\mathbf{x}_i$ being the focus of attention, we can compute the proportional relevance to each other item in the sequence using the softmax function:

$$\alpha_{i,j} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) = \frac{\exp\big(\text{score}(\mathbf{x}_i, \mathbf{x}_j)\big)}{\sum_{k=1}^{n} \exp(\text{score}(\mathbf{x}_i, \mathbf{x}_k))} \tag{2.11}$$

Attention is then computed through a weighted sum of all items with their proportional relevance:

$$\mathbf{y}_i = \sum_{j=1}^{n} \alpha_{i,j} \mathbf{x}_j \tag{2.12}$$

Each item of the sequence plays three different roles in the above described calculation of attention: The current focus of attention is called the *query*. Each item is used as a *key* when being compared with the query to calculate their proportional relevance. Finally, each item multiplied by its relevance in the weighted sum plays the role of a *value*. The self-attention layer of a transformer block uses different vectors for each of the three roles. The vectors are obtained by multiplying the input with three different

weight matrices for query, key and value:

$$\mathbf{q}_i = \mathbf{W}^Q \mathbf{x}_i; \quad \mathbf{k}_i = \mathbf{W}^K \mathbf{x}_i; \quad \mathbf{v}_i = \mathbf{W}^V \mathbf{x}_i \tag{2.13}$$

where $\mathbf{q}_i$, $\mathbf{k}_i$ and $\mathbf{v}_i$ are query, key and value vectors respectively and $\mathbf{W}^Q$, $\mathbf{W}^K$ and $\mathbf{W}^V$ are trainable weight matrices of the layer.

The self-attention layer also divides the dot product attention by $\sqrt{d}$ where d is the dimensionality of the query and key vectors. Vaswani, Shazeer, Parmar, et al. (2017) introduce this scaling factor to counteract the issue of extremely small gradients of the softmax function, which occurs when the dot product becomes too large as dimensionality grows. With that, attention is computed as:

$$\mathbf{y}_i = \sum_{j=1}^{n} \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}}\right) \mathbf{v}_j \tag{2.14}$$

In practice, attention is computed simultaneously for a set of queries, keys and values packed together into matrices:

$$\mathbf{Y} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right) \mathbf{V} \tag{2.15}$$

A transformer block employs a so-called multi-head attention, which is a set of self-attention layers that compute their outputs in parallel, each of them using their own weight matrices. Each parallel computing unit is called a head and can capture different types of relation between the inputs. The output of each head is concatenated and projected into the original output dimensions through a final linear transformation.

The multi-head self-attention mechanism is one of the two main components of a transformer block. The other is a simple fully-connected feed-forward network. Figure 2.3 shows the structure of a transformer block. There are *residual connections* around each of the two layers, which are shortcuts that skip a layer entirely. The input of the layer is thus added to its output before being passed forward in the network. Residual connections give higher level layers direct access to information from lower layers, which has been shown to improve the learning of deep networks (He, X. Zhang, Ren, and Sun 2016). Finally, the outputs are normalized, which involves subtracting the mean and dividing by the standard deviation, resulting in a vector of 0 mean and a standard deviation of 1. The normalized output vector is then further transformed with a learnable weight and bias. This process is called *layer normalization* and has been shown to reduce the training time of deep networks by keeping the values of layer outputs in a reasonable range that gives useful gradients (J. L. Ba, Kiros, and G. E. Hinton 2016).

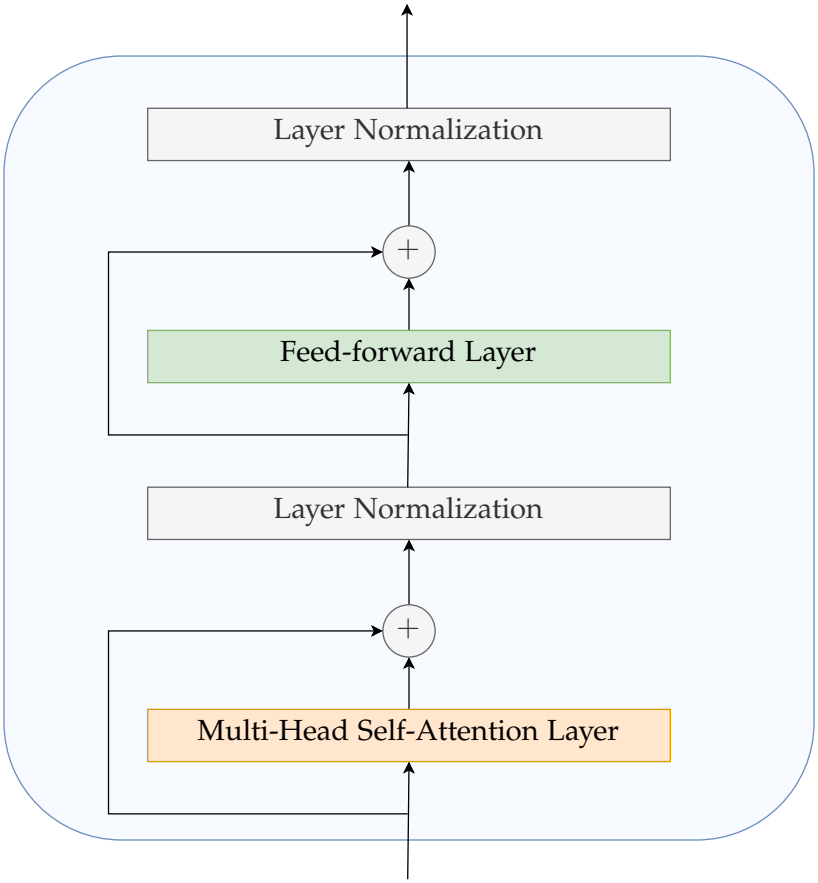

Figure 2.3: The structure of a transformer block. The key components are a multi-head self-attention layer and a feed-forward layer. There are residual connections around each layer and layer normalization is applied on the outputs.

2.2 Optimization

In Subsection 2.1.3 we have established that training a neural network involves minimizing an objective function. This section is dedicated to algorithms that can solve the optimization problem in the context of neural network training. In particular, we are interested in solving the following unconstrained minimization problem:

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^n} \mathcal{J}(\boldsymbol{\theta}) \tag{2.16}$$

where $\mathcal{J} : \mathbb{R}^n \to \mathbb{R}$ is the objective function and $\boldsymbol{\theta} \in \mathbb{R}^n$ contains the trainable parameters of the model. The optimization algorithms we will introduce define suitable parameter updates, meaning for a given parameter configuration $\boldsymbol{\theta}_t$ at iteration t, they define $\boldsymbol{\theta}_{t+1}$ with the intention of minimizing $\mathcal{J}$.

We will present first-order and zeroth-order methods of optimization, where *order* refers to the order of the associated derivatives. Zeroth-order methods use only function values $\mathcal{J}(\boldsymbol{\theta})$ while first-order methods also make use of the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$ (Kolda, Lewis, and Torczon 2003).

2.2.1 Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) algorithm computes the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$ at each step to find the right direction for $\boldsymbol{\theta}$ to move (the descent direction). With $\boldsymbol{\theta} \in \mathbb{R}^n$ being the vector of parameters, the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$ is defined as the vector of partial derivatives of $\mathcal{J}$ w.r.t each parameter:

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}_n} \end{bmatrix} \tag{2.17}$$

That means finding the descent direction requires the partial derivative of $\mathcal{J}$ w.r.t each weight and bias of the network in each layer. The partial derivatives are calculated using the chain rule of differentiation, which specifies the derivative of function compositions. The derivative of a composite function $f(x) = u(v(x))$ w.r.t x is calculated using the chain rule as:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}u}{\mathrm{d}v} \cdot \frac{\mathrm{d}v}{\mathrm{d}x} \tag{2.18}$$

At the core of gradient calculation for SGD and other gradient-based optimization methods lies the backpropagation algorithm (Rumelhart, G. E. Hinton, and Williams 1986), which does a backward pass on the network after calculating the objective $\mathcal{J}$ to propagate the partial derivatives backwards beginning from the last layer. Each layer

then takes the gradients passed in from the next higher layer, locally computes the partial derivatives of $\mathcal{J}$ w.r.t each of its parameters according to the chain rule, and finally passes the gradients necessary for further calculations to the next lower layer. As shown in Section 2.1, each layer in a neural network performs a chain of operations on its input to generate its output and the entire network can be represented as a computation graph where each node contains the intermediary result of one operation. Figure 2.4 shows an example of such computation graph expanded at one hidden layer. A forward pass through the computation graph calculates the model output and loss, thus the objective. A backward pass (shown in blue) calculates partial derivatives at each node, where the partial derivatives at the weights and biases (orange nodes) are used for the parameter update of SGD.

Since the gradient gives the ascent direction of the objective function, SGD moves $\boldsymbol{\theta}$ in the opposite direction by a certain step size η, which is called the *learning rate*. Equation 2.19 gives the parameter update of SGD:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) \tag{2.19}$$

The learning rate η is a *hyperparameter* of the algorithm, meaning the choice for its value needs to be made before applying the algorithm and plays a crucial role in its effectiveness: If the learning rate is too low, convergence to the minimum will be too slow, whereas if it is too high, the algorithm might oscillate around the minimum and fail to converge. It can be beneficial to change the learning rate during training such that we begin training with a larger learning rate to rapidly approach the minimum and gradually decrease the learning rate as training goes on to get as close to the minimum as possible with smaller adjustments. This can be achieved by using a learning rate scheduler, a function that returns a multiplier for η at each step. The learning rate η in Equation 2.19 can then be replaced by η_t, the effective learning rate at step t.

SGD is an online algorithm, which means it does not need to process the entire input outright but can go through the input example by example (Jurafsky and Martin 2024, Chapter 5). Usually, inputs are given in small groups called mini-batches: In each iteration, b randomly-picked samples are given as input to the model to calculate b outputs, and loss is averaged over the b samples:

$$\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) = \frac{1}{b} \sum_{i=1}^{b} \nabla_{\boldsymbol{\theta}} \mathcal{J}_i(\boldsymbol{\theta}) \; ; \quad \mathcal{J}_i(\boldsymbol{\theta}) = \mathcal{J}(\boldsymbol{\theta}; \mathbf{x}_i, \mathbf{y}_i) \tag{2.20}$$

The batch size b is also a hyperparameter. Smaller batch sizes make noisy updates because of a greater variance in the gradients (in the extreme case of $b = 1$, parameters are updated to fit a single sample in each iteration). Larger batch sizes allow the use of larger learning rates while achieving the same optimal error bounds (Lin and

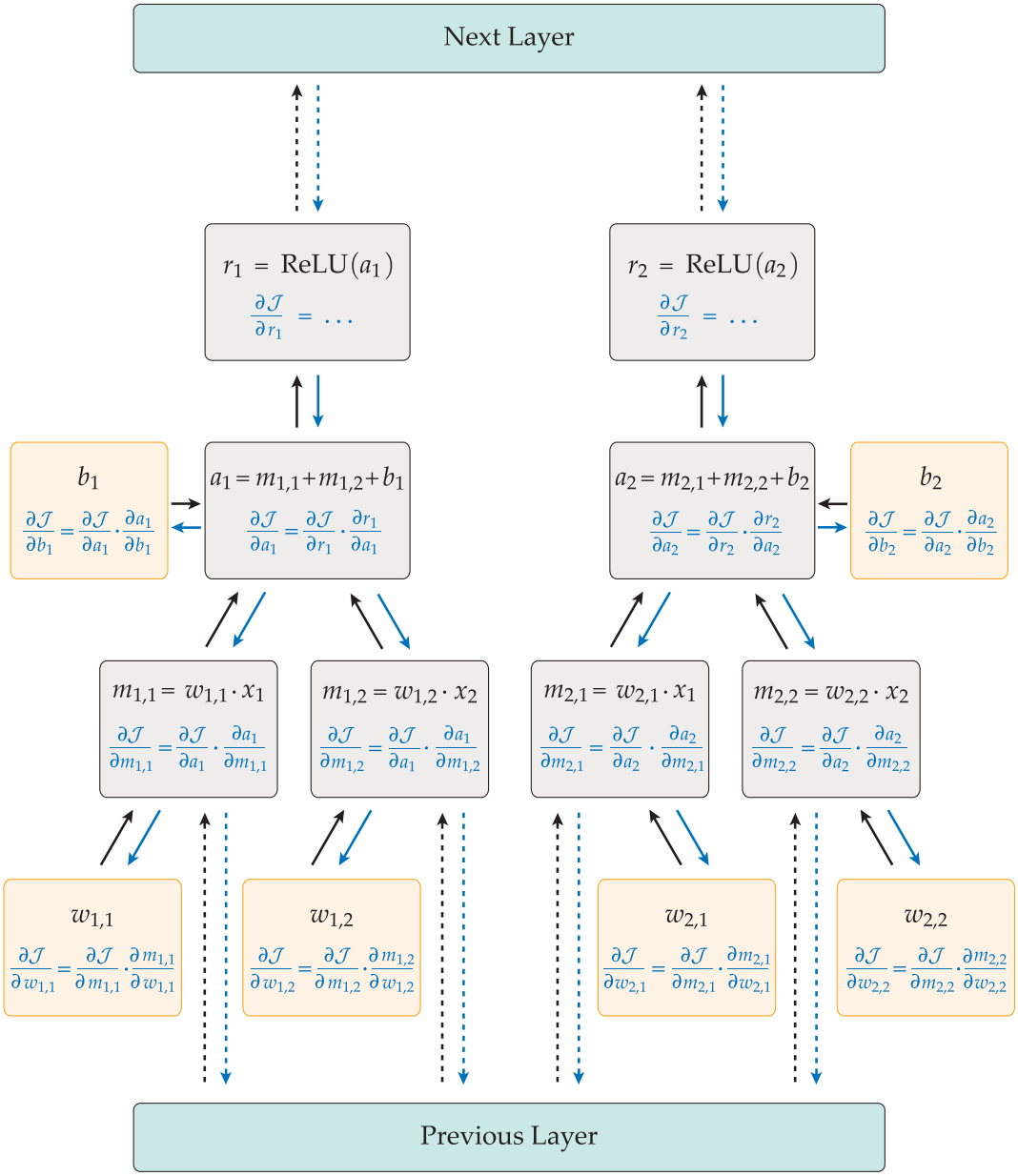

Figure 2.4: Backward Pass on the computation graph of a feed-forward network. The graph shows a hidden layer with two neurons. The direction of the backward pass and the partial derivatives computed by the backpropagation algorithm are shown in blue. Orange nodes are the trainable parameters of the layer.

Rosasco 2017) and improve parallelism as the increased amount of computation per iteration can be effectively distributed. On the other hand, they increase the memory requirement of the algorithm and when batch size is too high, it can impair the model's ability to generalize: It has been observed that when using larger batch sizes, algorithms tend to converge to *sharp* minimizers where the function increases rapidly in a small neighborhood of the minimum, which negatively impacts generalization (Keskar, Mudigere, Nocedal, et al. 2017).

2.2.2 Adaptive Gradient Methods

For efficient optimization, one needs to overcome a central issue of the SGD algorithm, which is its slow convergence rate. As discussed in Subsection 2.2.1, the convergence rate is directly bound to the learning rate η, but the algorithm fails to converge when η is too high. Thus, η needs to be picked conservatively to avoid divergence, resulting in excessively slow convergence rates. Adaptive gradient methods try to solve this issue by taking the parameter updates from previous iterations into account when calculating the update for the current iteration.

Momentum

Ideally, we want to take larger steps in a direction when a sequence of previous gradients all pointed in that direction, effectively accumulating gradients over time. This is the main idea behind the momentum method (Polyak 1964). The momentum method calculates a velocity vector $\mathbf{v}$ that accumulates in directions that persistently reduce the objective function across iterations:

$$\mathbf{v}_{t+1} = \beta \cdot \mathbf{v}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}_t) \tag{2.21}$$

where $\beta \in \mathbb{R}$ is called the momentum coefficient and determines the rate of velocity accumulation. The parameter update is then given using the velocity vector:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1} \tag{2.22}$$

Nesterov Momentum

Nesterov's Accelerated Gradient, also called Nesterov momentum (Nesterov 1983), is a simple modification of the classical momentum method that has been observed to increase stability and improve performance (Sutskever, Martens, Dahl, and G. Hinton 2013). Instead of computing the velocity using the gradient from the current position, it first performs a partial update and uses the gradient at the position $\boldsymbol{\theta}_t + \beta \cdot \mathbf{v}_t$:

$$\mathbf{v}_{t+1} = \beta \cdot \mathbf{v}_t - \eta \cdot \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}_t + \beta \cdot \mathbf{v}_t) \tag{2.23}$$

Adam

The last adaptive gradient method we want to share is Adam (Kingma and J. Ba 2014). Adam computes two moving averages similar to momentum:

$$\mathbf{m}_{t+1} = \beta_1 \cdot \mathbf{m}_t + (1 - \beta_1) \cdot \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}_t) \tag{2.24}$$

$$\mathbf{v}_{t+1} = \beta_2 \cdot \mathbf{v}_t + (1 - \beta_2) \cdot (\nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}_t))^2 \tag{2.25}$$

where $\mathbf{m}$ and $\mathbf{v}$ operate as estimates of the mean and the uncentered variance of the gradient respectively and $\beta_1, \beta_2 \in [0, 1)$ control their decay rates. $\mathbf{m}_0$ and $\mathbf{v}_0$ are initialized as 0's, which cause a bias towards 0, especially in the early steps and when β_1 and β_2 are close to 1. Thus, a bias correction is applied:

$$\hat{\mathbf{m}}_{t+1} = \frac{1}{1 - \beta_1^{t+1}} \mathbf{m}_{t+1} \tag{2.26}$$

$$\hat{\mathbf{v}}_{t+1} = \frac{1}{1 - \beta_2^{t+1}} \mathbf{v}_{t+1} \tag{2.27}$$

Finally, parameters are updated using the bias-corrected estimates:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \frac{\hat{\mathbf{m}}_{t+1}}{\sqrt{\hat{\mathbf{v}}_{t+1}} + \epsilon} \tag{2.28}$$

where $\epsilon > 0$ is a very small number. The authors give $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ as good default settings with $\eta = 0.001$.

2.2.3 Finite Difference Methods

Finite difference methods are zeroth-order methods that solve an optimization problem using the same recipe as first-order gradient methods, only without explicit computation of a gradient. Instead, gradients are estimated from a sequence of function evaluations, effectively requiring only forward passes to train a neural network.

The Kiefer-Wolfowitz finite-difference algorithm (Kiefer and Wolfowitz 1952) approximates the derivative of a function $f(x)$ as

$$f'(x) \approx \frac{f(x + h) - f(x - h)}{2h} \tag{2.29}$$

The quantity h is called the finite difference interval and using the Taylor series expansion, the following relation can be shown for sufficiently small intervals (Gill, Murray, and Wright 1981, Page 54):

$$\frac{f(x + h) - f(x - h)}{2h} = f'(x) + O(h^2) \tag{2.30}$$

Using the Kiefer-Wolfowitz finite difference, we can obtain a gradient approximation $\hat{\nabla}\mathcal{J}(\boldsymbol{\theta})$:

$$\hat{\nabla}_i \mathcal{J}(\boldsymbol{\theta}) = \frac{\mathcal{J}(\boldsymbol{\theta} + h \cdot \mathbf{e}_i) - \mathcal{J}(\boldsymbol{\theta} - h \cdot \mathbf{e}_i)}{2h} \tag{2.31}$$

$$\hat{\nabla}\mathcal{J}(\boldsymbol{\theta}) = (\hat{\nabla}_1 \mathcal{J}(\boldsymbol{\theta}), \hat{\nabla}_2 \mathcal{J}(\boldsymbol{\theta}), \ldots, \hat{\nabla}_n \mathcal{J}(\boldsymbol{\theta}))^T \tag{2.32}$$

where $\mathbf{e}_i$ is the i-th unit vector. The disadvantage of this method is the large number of function evaluations it requires ($2n$ evaluations for n-dimensional $\boldsymbol{\theta}$) (Pachal, Bhatnagar, and Prashanth 2023).

Simultaneous Perturbation Stochastic Approximation (SPSA) (Spall 1992) reduces the number of function evaluations to 2 by perturbing all directions at the same time:

$$\hat{\nabla}\mathcal{J}(\boldsymbol{\theta}) = \frac{\mathcal{J}(\boldsymbol{\theta} + h \cdot \mathbf{z}) - \mathcal{J}(\boldsymbol{\theta} - h \cdot \mathbf{z})}{2h} \cdot \mathbf{z} \tag{2.33}$$

where $\mathbf{z} \in \mathbb{R}^n$ is a vector of small random perturbations and can be sampled, e.g. from a standard gaussian distribution with $z_i \sim \mathcal{N}(0, 1)$. Using the gradient estimate from Equation 2.33 as a replacement for the gradient in Equation 2.19 (SGD), we obtain the **ZO-SGD** algorithm (Spall 1992).

The **ZO-signSGD** algorithm (Liu, P.-Y. Chen, X. Chen, and Hong 2019) instead uses the sign of the estimated gradient as direction for the update. It has been shown that the first-order signSGD algorithm (Bernstein, Wang, Azizzadenesheli, and Anandkumar 2018) yields an empirically faster convergence speed than standard SGD ZO-signSGD shows analogous improvements in convergence rate over ZO-SGD. However, it has worse convergence accuracy than ZO-SGD as it only converges to a neighborhood of the minimum (X. Chen, Liu, Xu, et al. 2019).

MeZO (Malladi, Gao, Nichani, et al. 2023) proposes a memory-efficient implementation of ZO-SGD where the algorithm does not store the random perturbation vector $\mathbf{z}$ of Equation 2.33. Instead, the random seed is stored and the random number generator is reset by this seed to resample $\mathbf{z}$ for each of its uses, giving a memory footprint equivalent to the inference memory cost.

The zeroth-order adaptive momentum method (**ZO-AdaMM**) (X. Chen, Liu, Xu, et al. 2019) implements an adaptive gradient method using the zeroth-order gradient estimate. It fuses the gradient estimate with AMSGrad (Reddi, Kale, and Kumar 2018), which is a modified version of Adam. Compared to other zeroth-order methods, the authors show superior performance and convergence rate for the task of designing adversarial examples.

2.2.4 Direct Search Methods

Unlike the previously discussed finite difference methods, direct search methods do not rely on a gradient approximation. Instead, they conduct a search by identifying possible candidates $\hat{\boldsymbol{\theta}}_1, \dots \hat{\boldsymbol{\theta}}_c$, for the parameter vector $\boldsymbol{\theta}$, evaluating the objective function $\mathcal{J}$ on those points and updating the parameters to the best candidate:

$$\boldsymbol{\theta}_{t+1} = \arg\min_{\hat{\boldsymbol{\theta}}} \{\mathcal{J}(\hat{\boldsymbol{\theta}}) | \hat{\boldsymbol{\theta}} \in \{\hat{\boldsymbol{\theta}}_1, \dots, \hat{\boldsymbol{\theta}}_c\}\} \tag{2.34}$$

Typically, a candidate is determined by adding a random perturbation vector to the current parameter:

$$\hat{\boldsymbol{\theta}}_k = \boldsymbol{\theta}_t + \mathbf{z}_k \; ; \quad \mathbf{z}_k \sim \mathcal{D}_k \tag{2.35}$$

where $\mathcal{D}_k$ is some sampling distribution.

Kolda, Lewis, and Torczon (2003) give a detailed review of the strengths and weaknesses of direct search methods: Direct search algorithms are usually straightforward to implement but have slower convergence rates. Since they only need to identify optimal parameters in a small search space, direct search algorithms can be used to optimize noisy or non-smooth functions where a finite difference approximation of the gradient cannot be found reliably, as well as nonnumerical functions where gradient calculation or estimation is impossible. Even though asymptotic convergence rates are slow, direct search methods can still be applied in practice when reaching a sufficiently good solution is more important than convergence, particularly when the function to optimize is inaccurate. A major limitation of direct search methods is that their performance deteriorates significantly as the number of variables increases. Seeking for ways to overcome this limitation is a key part of our work.

Stochastic Three Points (**STP**) (Bergou, Gorbunov, and Richtárik 2020) samples a single perturbation vector at each iteration with $\mathbf{z}_t \sim r_t \mathcal{D}$ where r_t is the step size at iteration t. Then, it considers three points for the parameter update:

$$\boldsymbol{\theta}_{t+1} = \arg\min\{\mathcal{J}(\boldsymbol{\theta}_t), \mathcal{J}(\boldsymbol{\theta}_t + \mathbf{z}_t), \mathcal{J}(\boldsymbol{\theta}_t - \mathbf{z}_t)\} \tag{2.36}$$

Thus, STP only requires two additional function evaluations per iteration.

Stochastic Momentum Three Points (**SMTP**) (Gorbunov, Bibi, Sener, et al. 2020) is a modification of the STP algorithm. Motivated by the efficiency of momentum methods in first-order optimization, the authors introduce momentum to STP and demonstrate that momentum could be beneficial for stochastic zeroth-order methods as well.

Gradientless Descent (**GLD**) (Golovin, Karro, Kochanski, et al. 2020) is a simple direct search algorithm that samples random perturbation vectors from a fixed distribution $\mathcal{D}$ but with different choices of radius r_k. Given a maximal radius R and a minimal radius

r, the algorithm performs a binary search in the interval $[r, R]$ where $r_0 = R$ and each consecutive candidate halves the search radius until the minimal radius is reached:

$$\mathbf{z}_k \sim r_k \mathcal{D} \; ; \quad r_k = 2^{-k} R \tag{2.37}$$

The parameter is updated to the best candidate if at least one of them improves the objective:

$$\boldsymbol{\theta}_{t+1} = \arg\min_{\boldsymbol{\theta}} \{ \mathcal{J}(\boldsymbol{\theta} + \mathbf{c}) \mid \mathbf{c} = \mathbf{0}, \mathbf{z}_0, \dots \mathbf{z}_n \} \tag{2.38}$$

This binary search arrangement allows the authors to give convergence guarantees depending on the search interval $[r, R]$ when minimizing arbitrary functions.

Although the update rule is similar to STP, for GLD, the number of function evaluations per step depends on the size of the search interval with $K = \log_2(R/r)$, i.e. it can be adjusted by changing the search interval where larger intervals perform more function evaluations per iteration but give better convergence guarantees. Particularly, a smaller minimal radius guarantees convergence to a closer neighborhood of the minimum (Golovin, Karro, Kochanski, et al. 2020).

GLD provides the base algorithm of our work. We add our own modifications on top of the base GLD algorithm for the purpose of using it as an optimizer for a neural network model. Above all, this involves an effort to overcome the limitations on the number of parameters. Our methods will be detailed in Chapter 3.

2.3 Importance of Network Parameters

Neural Networks can fit a wide variety of functions thanks to their strong parameterization. However, not all parameters are equally important for the performance of a network. Having information about the importance of parameters can be useful as unimportant parameters can be removed to prune the network, reducing its size.

The magnitude of a parameter is a good indicator of its importance since parameters with low absolute values will have little effect on the output of their respective layers. This importance measure allows connections with weights below a certain threshold to be removed from the network without losing performance (Han, Pool, Tran, and Dally 2015).

The Fisher Importance Matrix (FIM) is a useful quantity in statistics that provides an estimate of the amount of information a random variable carries about a parameter of its distribution. For a neural network with parameters $\boldsymbol{\theta}$, the FIM is given as

$$F_{i,j}(\boldsymbol{\theta}) = E\left[\frac{\partial \log p(\mathbf{x}; \boldsymbol{\theta})}{\partial \theta_i} \frac{\partial \log p(\mathbf{x}; \boldsymbol{\theta})}{\partial \theta_j} \right] \tag{2.39}$$

where p is a probability distribution of model outputs with the given parameters. The FIM can be used as an importance measure for model parameters. Tu, Berisha, Woolf, et al. (2016) show that network complexity can be reduced without a noticeable effect on performance by removing parameters with low entries on the diagonal of the FIM.

Computing the FIM as given in Equation 2.39 requires estimating the probability distribution function p, which is mostly unknown. Therefore, the authors define a method to estimate only the diagonal entries of the FIM: The algorithm iterates over each parameter in $\boldsymbol{\theta}$. Each iteration calculates a perturbation $\tilde{\boldsymbol{\theta}}$ of the parameter vector where a small random perturbation z is added to only the current iterate:

$$\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta}$$

$$\tilde{\boldsymbol{\theta}}(i) = \tilde{\boldsymbol{\theta}}(i) + z \tag{2.40}$$

The algorithm then computes the D_α-divergence of 2 different model outputs: One with the original parameters $\boldsymbol{\theta}$, the other with the perturbed parameters $\tilde{\boldsymbol{\theta}}$. The divergence is a measure of distance between the 2 distributions described by the 2 model outputs and is therefore used as an estimation for the importance of the iterate.

The D_α-divergence is given by

$$D_\alpha(p, q) = \frac{1}{4\alpha(1-\alpha)} \times \left[\int \frac{(\alpha p(\mathbf{x}) - (1-\alpha) q(\mathbf{x}))^2}{\alpha p(\mathbf{x}) + (1-\alpha) q(\mathbf{x})} d\mathbf{x} - (2\alpha - 1)^2 \right] \tag{2.41}$$

Berisha and Hero (2015) provide a method to estimate the D_α-divergence, which is crucial for the importance estimation algorithm. The method involves constructing an Euclidean minimal spanning tree of from all points $\mathbf{X}_p, \mathbf{X}_q$ sampled from the 2 distributions p and q. The number of edges connecting a point from p to a point from q gives the Friedman-Rafsky multi-variate two sample test statistic $\mathcal{C}(\mathbf{X}_p, \mathbf{X}_q)$ (Friedman and Rafsky 1979). An estimation for the D_α-divergence can then be computed as

$$D_\alpha(p, q) \approx 1 - \mathcal{C}(\mathbf{X}_p, \mathbf{X}_q) \frac{N_p + N_q}{2 N_p N_q} \tag{2.42}$$

where N_p and N_q are the number of samples from p and q. Finally, taking p and q as distributions modeled by 2 neural networks with parameters $\boldsymbol{\theta}$ and $\tilde{\boldsymbol{\theta}}$ with $\mathbf{X}_p$ and $\mathbf{X}_q$ being the sampled model outputs, one can estimate the D_α-divergence, which gives an estimation for the importance of the perturbed parameter.

3 Methods

We start with the GLD algorithm (Golovin, Karro, Kochanski, et al. 2020) and through constant empirical testing propose modifications to overcome its limitations. Our goal is to develop a memory-efficient method that can perform comparably to state-of-the-art first-order gradient-based optimization techniques when fine-tuning a Large Language Model (LLM). We understand that such a method might come at the cost of increased time complexity, particularly considering the already established slow convergence rates of direct search methods (Kolda, Lewis, and Torczon 2003). Nevertheless, we try to find a balance such that convergence rates remain reasonable. This chapter introduces the modifications we developed.

3.1 Parameter Partitioning

The biggest hurdle to fine-tuning a LLM using direct search methods is the problem size. LLMs have a large number of parameters, which means training them all at the same time is impractical with our base algorithm as it would have to use very small radii. Instead, we divide the parameters into subsets and we let the GLD algorithm optimize one subset at a time.

Initially, we experimented with considering each individual weight matrix and bias vector of the network as one subset. However, this partitioning scheme proved to be too coarse since some weight matrices were still too large on their own even after the partitioning. For those weight matrices we develop different methods to select s individual weights to train at each iteration, where s is a number of variables optimizable by GLD. The simplest method is to select s weights at random. We mainly use this method for our experiments. We also experiment with picking weights based on importance where we rank the weights based on some importance metric and select the s elements with the highest ranking. We consider two different importance metrics: the absolute value and the Fisher importance.

The absolute value method comes from the intuition that higher absolute weight values have a more noticeable effect on the output of a layer (Conversely, weights close to 0 have little effect).

The Fisher method requires estimating the as described FIM in Section 2.3. This is a costly process, making it infeasible to run in each iteration. Instead, we run it only a few

times throughout the entire training. See Section 4.2 for the details of our experiments.

As an initialization step, we assign each tensor that needs further reduction in size one of the defined methods (random, absolute value or Fisher) to select the parameters that are to be trained in an iteration. Each iteration t selects a list of perturbation indices $\mathbf{i}$ for each tensor $\boldsymbol{\theta}$. The tensor is then perturbed by vector $\mathbf{v}$ with $\boldsymbol{\theta}_t(\mathbf{i}) = \boldsymbol{\theta}_t(\mathbf{i}) + \mathbf{v}$ where $\boldsymbol{\theta}_t(\mathbf{i})$ refers to parameters of the tensor at indices $\mathbf{i}$. Matrices, which comprise $\leq s$ parameters receive updates for all their parameters. Similarly, we can assign each tensor a maximal radius R for the binary search of GLD, allowing the flexibility to search in different intervals for different parameters. In the following, $R(\boldsymbol{\theta})$ denotes the maximal radius assigned to a tensor $\boldsymbol{\theta}$. In this case, we consider the number of used radii K as a tunable hyperparameter of the algorithm instead of providing the minimal radius r. The search interval can then be described as $[2^{1-K}R, R]$.

Algorithm 1 Gradientless Descent Training

for $t = 1, \ldots, T$ **do**
  for each tensor $\boldsymbol{\theta}_t$ **do**
    $\mathbf{i} \leftarrow$ SelectPerturbationIndices($\boldsymbol{\theta}_t$, s) ▷ *Either selects all indices, or selects $s < size(\theta_t)$ indices using one of our random, absolute value or Fisher methods*
    for $k = 1, \ldots, K$ **do**
      $r_k \leftarrow 2^{-k} R(\boldsymbol{\theta}_t)$
      $\mathbf{v}_k \sim r_k \mathcal{D}$
      $\boldsymbol{\theta}_{t,k} \leftarrow \boldsymbol{\theta}_t$
      $\boldsymbol{\theta}_{t,k}(\mathbf{i}) \leftarrow \boldsymbol{\theta}_{t,k}(\mathbf{i}) + \mathbf{v}_k$
    $\boldsymbol{\theta}_{t+1} \leftarrow \arg\min_{\boldsymbol{\theta}} \{\mathcal{J}(\boldsymbol{\theta}_t), \mathcal{J}(\boldsymbol{\theta}_{t,k})\}$

Algorithm 1 defines our main method for training neural networks using GLD. Since it is crucial to reduce the size of the problem to successfully apply GLD, all further methods we define also partition parameters as in Algorithm 1.

3.2 Radius scheduling

The radius r_k in Algorithm 1 serves a similar purpose as the step size when the perturbation vector $\mathbf{v}_k$ gives the best loss, as together with the distribution $\mathcal{D}$, it determines how big the values are in $\mathbf{v}_k$. It is therefore comparable to the learning rate of first-order optimizers.

As discussed in Subsection 2.2.4, the binary search performed by GLD across the interval $[r, R]$ makes it possible to give convergence guarantees depending on the values of r and R. Better guarantees can be given for bigger search intervals as the algorithm

considers both large radii to escape saddle points and small radii to converge when in a close neighborhood of the minimum. However, having a large search interval also increases the number of forward passes required in each iteration, increasing the time complexity of the algorithm. Ideally, we want to keep K small without sacrificing the ability to converge.

The resemblance between search radius and learning rate brings about the idea to implement a radius scheduler similar to a learning rate scheduler for first-order optimizers. We already assigned each parameter matrix a maximal radius R. We extend our implementation to assign each tensor a radius scheduler as well. A scheduler is hereby a function $\mathbb{R} \to \mathbb{R}$ taking the current step count and returning a multiplier for R. With constant K, this is equivalent to shifting the interval $[r, R]$ during training. Starting with larger radii and gradually shifting the interval towards smaller radii, we expect to keep the base algorithm's convergence with a fewer number of forward passes each iteration.

3.3 Options

A key weakness of Algorithm 1 is that updates through random perturbation of parameters end up being noisy. Often, a perturbation that improves the objective for one mini-batch makes it worse for others. A stabler update policy is desired as it could allow faster progress and better convergence. The simplest way to solve this issue is to increase the batch size, making it more likely for perturbations picked by the algorithm to be good in the long run. In this section we introduce a different approach. We store so-called *options*, potential updates that the algorithm has considered for each tensor, and update to an option only after exploring a lookahead of future updates it can lead to.

We define an option p as a tuple $(\mathbf{i}, \mathbf{v})$ of indices and perturbation vector that stores a perturbation applied by the algorithm. With $\boldsymbol{\theta}_p$, we denote a tensor resulting from perturbing according to p. Similarly $\boldsymbol{\theta}_{t,p}$ denotes the tensor at time step t being perturbed according to p. For each tensor $\boldsymbol{\theta}$ we define a set of options $\psi(\boldsymbol{\theta})$. After the binary search, the algorithm does not update the parameter with the best perturbation. Instead the top m candidates are stored in $\psi(\boldsymbol{\theta})$. In each iteration t, the algorithm then applies each option $p \in \psi(\boldsymbol{\theta}_t)$ and does a binary search for perturbations on each $\boldsymbol{\theta}_{t,p}$. We say $p^* \in \psi(\boldsymbol{\theta}_t)$ is the best option if one of the perturbations on $\boldsymbol{\theta}_{t,p^*}$ gives the minimum objective of that step. The parameter is updated to $\boldsymbol{\theta}_{t,p^*}$ and $\psi(\boldsymbol{\theta}_{t+1})$ is produced from the top m perturbations on $\boldsymbol{\theta}_{t,p^*}$. We note that for $m = 1$ the method is equivalent to Algorithm 1: Since there is only one option, p^* is always the best perturbation found in an iteration and the next iteration will update the parameter

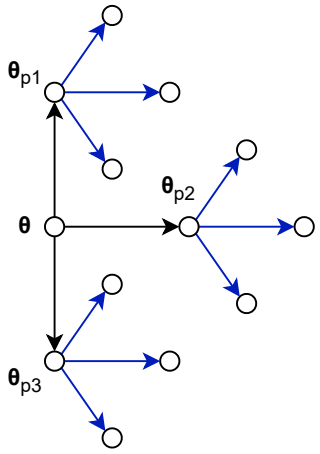

Figure 3.1: Visualization of a potential setup of two levels of options. Black and blue arrows show first-level and second-level options respectively. K perturbations are applied on each option on the second level and $\boldsymbol{\theta}$ is updated to either $\boldsymbol{\theta}_{p1}$, $\boldsymbol{\theta}_{p2}$ or $\boldsymbol{\theta}_{p3}$ depending on which first-level option is the parent of the best perturbation.

with p^*.

The method can be extended to include multiple layers of options. In that case, each option p also stores a set of options $\psi(p)$. We refer to the number of option levels as ℓ. Figure 3.1 visualizes two levels of options on a tensor. This setup can be understood as a tree structure with a height of ℓ where each level has a width of m and each node holds an option. At the root of the tree is the parameter $\boldsymbol{\theta}$, which can be described as an "empty" option where the perturbation vector is 0. $\psi(p)$ describes the children of each node and we have $\psi(p) = \emptyset$ for leaf nodes, which are options at the ℓ-th level.

Algorithm 2 gives the formal definition of the method. Intuitively, the proposed method describes a lookahead into potential future updates from each point to select parameter updates which provide a continuous loss decrease over multiple steps. An update is selected if one of its updates at the ℓ-th level gives the best objective. We hypothesize this will reduce the noise of updates.

We remark that the type of lookahead we develop is in essence only slightly different to applying Algorithm 1 with larger batch sizes: When performing a single parameter update, Algorithm 2 considers not only multiple mini-batches but also multiple perturbations on the original parameter. We can say with batch size b, $(\ell + 1) \cdot b$ samples and $\sum_{i=0}^{\ell} m^i \cdot K$ perturbations influence a parameter update, while each perturbation considers b samples and from each point we have K perturbations. We can look ahead to points up to $(\ell + 1) \cdot R$ optimization steps away from the original parameter but

each perturbation is still in the interval $[r, R]$. Using options significantly increases the number of forward passes performed in each iteration with $m^\ell \cdot K$ forward passes per tensor instead of only K. However, unlike using a larger batch size, any improvement in performance comes at no additional memory cost.

Algorithm 2 Gradientless Descent Training with Options

for $t = 1, \ldots, T$ **do**
  for each tensor $\boldsymbol{\theta}_t$ **do**
    $\mathbf{i} \leftarrow$ SelectPerturbationIndices($\boldsymbol{\theta}_t$, s)
    $p^* \leftarrow \arg\min_{p \in \psi(\boldsymbol{\theta}_t)} \{$ComputeOptionLoss($\boldsymbol{\theta}_t$, p, $\mathbf{i}$)$\}$ ▷ *Best first-level option*
    $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_{t,p^*}$
    $\psi(\boldsymbol{\theta}_{t+1}) \leftarrow \psi(p^*)$
procedure ComputeOptionLoss($\boldsymbol{\theta}_t$, p, $\mathbf{i}$)
  if $\psi(p) = \emptyset$ **then**
    ▷ *If this is the last level of options, apply perturbations to set the next level and return the minimum loss* ◁
    for $k = 1, \ldots, K$ **do**
      $r_k \leftarrow 2^{-k} R(\boldsymbol{\theta}_t)$
      $\mathbf{v}_k \sim r_k \mathcal{D}$
      $\boldsymbol{\theta}_{t,p,k} \leftarrow \boldsymbol{\theta}_{t,p}$
      $\boldsymbol{\theta}_{t,p,k}(\mathbf{i}) \leftarrow \boldsymbol{\theta}_{t,p,k}(\mathbf{i}) + \mathbf{v}_k$
    $\psi(p) \leftarrow \arg\min^{(m)}_{p'} \{\mathcal{J}(\boldsymbol{\theta}_{t,p,p'}) \mid p' \in \{(\mathbf{i}, \mathbf{0}), (\mathbf{i}, \mathbf{v}_k) \mid k = 1, \ldots K\}\}$ ▷ *The top m candidates*
    return $\min_k \{\mathcal{J}(\boldsymbol{\theta}_{t,p}), \mathcal{J}(\boldsymbol{\theta}_{t,p,k})\}$
  else
    ▷ *If this is not the last level of options, recursively return the minimum loss across all children* ◁
    return $\min_{p' \in \psi(p)} \{$ComputeOptionLoss($\boldsymbol{\theta}_{t,p}$, p', $\mathbf{i}$)$\}$

4 Experiments & Results

We test our methods on a binary classification task provided by the SST-2 dataset (Socher, Perelygin, Wu, et al. 2013). The dataset consists of sentences and smaller groups of words extracted from movie reviews and human annotations of their sentiment as either positive (1) or negative (0). Table 4.1 shows some example input-output pairs.

As our base model we use DistilBERT (Sanh, Debut, Chaumond, and Wolf 2020), which is a transformer model that uses knowledge distillation to reduce the size of a BERT model. The model consists of an embedding layer and 6 transformer layers with a hidden size of 768. For the binary classification task, a classification head is attached to the base model, which is a fully-connected feed-forward network consisting of a hidden layer of size 768 with ReLU activation and an output layer of size 2. For our experiments we train only this classification head, which means we optimize 4 tensors:

$$\mathbf{W}_1 \in \mathbb{R}^{768\times768}; \quad \mathbf{b}_1 \in \mathbb{R}^{1\times768}; \quad \mathbf{W}_2 \in \mathbb{R}^{768\times2}; \quad \mathbf{b}_2 \in \mathbb{R}^{1\times2}$$

Unless explicitly stated otherwise, we train all parameters of $\mathbf{b}_1, \mathbf{W}_2$ and $\mathbf{b}_2$, and randomly select $768 \times 2 = 1536$ parameters from $\mathbf{W}_1$ to perturb per iteration.

Since we train only the classification head and not the base model, we configure forward passes to calculate the output of the base model only once per iteration. This output is stored and further forward passes required by the algorithm in the same iteration calculate only the pass through the classification head, improving the speed of computation.

We use Algorithm 2 for all experiments. When an experiment does not involve options, we set both the number of option levels ℓ and the number of options per level m to 1, making it the equivalent of Algorithm 1. To ensure reproducibility, experiments set all random seeds to 42.

4.1 Hyperparameters

There are several hyperparameters that can influence the performance of Algorithm 1. As discussed in Section 3.1, we define the search interval through a maximal radius R and the number of searches per iteration K. Both R and K are hyperparameters as well as the batch size b. Testing our main algorithm, we are interested in seeing how each of those hyperparameters affect performance.

Table 4.1: Example input-output pairs from the SST-2 dataset

Sentence	**Label**
contains no wit , only labored gags	0
with his usual intelligence and subtlety	1
we never feel anything for these characters	0
equals the original and in some ways even betters it	1

Figure 4.1 shows validation accuracy for 3 of our experiments with different values for the maximal radius R where we use the same R for each of the 4 tensors. As expected, the maximal radius R is a deciding factor for the algorithm's performance. Generally, when R is too low improvement is slow. When it is too high, updates are noisy and the algorithm fails to converge at a good minimum. We decide that $R = 5 \cdot 10^{-3}$ is an adequate default value.

The number of searches per iteration K decides the size of the search interval. We expect better performance with bigger intervals. The results shown in Figure 4.2 confirm this for small K. However, there are diminishing returns as K grows. Conveniently, the run with only one search radius per iteration is able to get reasonable accuracy on our dataset. We remark that this might not be representative in general. We note that for $K = 1$, parameter updates behave like a stochastic two points update, where the algorithm decides whether the parameter should be updated with the given perturbation of radius R or not. In particular, there is no binary search.

The batch size b has the most significant effect on performance. Figure 4.3 shows validation accuracy with 3 different batch sizes. We find that larger batch sizes perform consistently better. The drawback is a higher memory and time consumption with constant step count. We find the results with $b = 64$ to be adequate and therefore conduct most of our experiments with a batch size of 64.

Since they provide a suitable balance between performance, memory and time consumption, we decide to take $R = 5 \cdot 10^{-3}$, $K = 5$ and $b = 64$ as the set of hyperparameters we use when testing our other approaches and modifications over the base algorithm. In the following, we refer to this configuration as "default hyperparameters".

4.2 Parameter Partitioning Methods

In Section 3.1, we introduced 3 methods to partition tensors that are too large to be trained efficiently by GLD: We either sample random indices or rank parameters

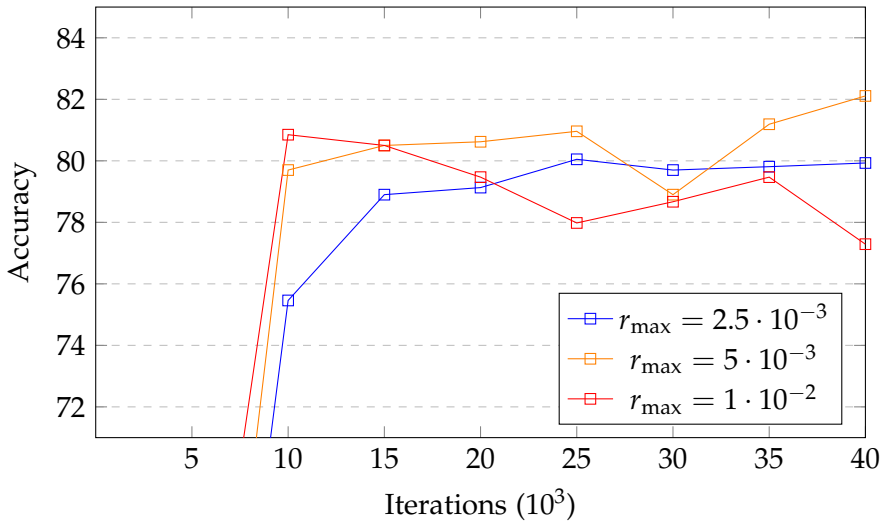

Figure 4.1: Validation Accuracy for three different values of maximal search radius. K is set to 5, batch size is 64.

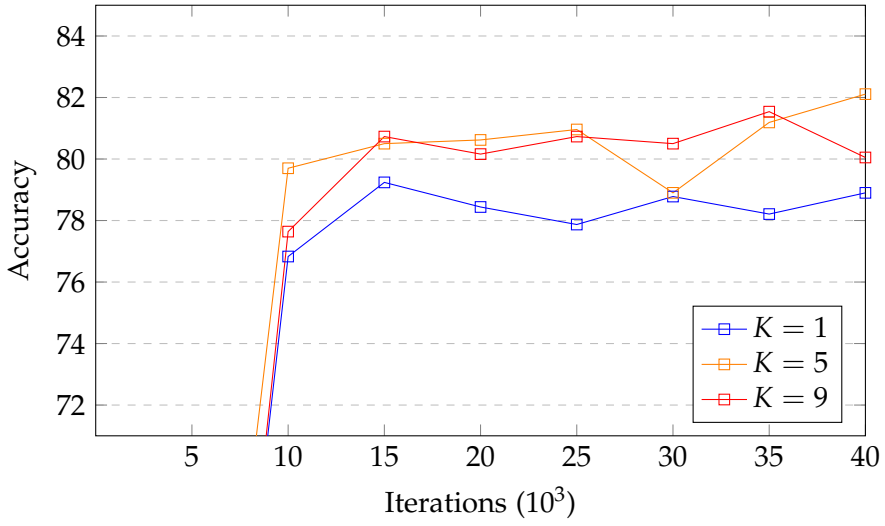

Figure 4.2: Validation Accuracy when performing 1, 5 and 9 searches per iteration with maximal radius $R = 5 \cdot 10^{-3}$. Batch size is 64.

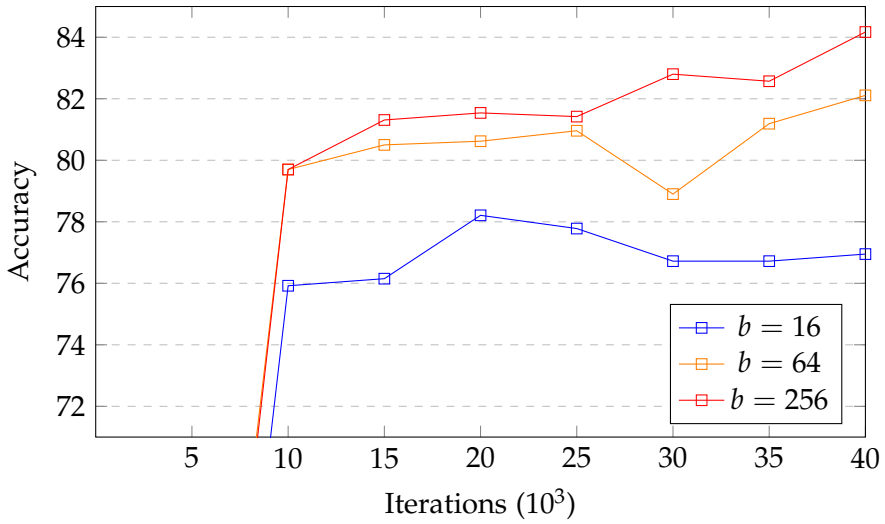

Figure 4.3: Validation Accuracy with 3 different batch sizes. K is set to 5, R is set to $5 \cdot 10^{-3}$.

based on absolute value or Fisher information estimate. The absolute value and Fisher methods aim to select the most important parameters to train, improving the convergence rate as the algorithm does not spend time on training unimportant parameters with little effect on the outcome. From the four tensors we are training, $\mathbf{W}_1$ is the largest with size 768×768, followed by $\mathbf{W}_1$ with size 768×2. We find that our algorithm can reliably train 768×2 parameters at a time, but 768×768 is too large. Therefore, we choose to select and train only 768×2 parameters from $\mathbf{W}_1$ per iteration. Figure 4.4 shows validation loss when training all parameters of $\mathbf{W}_1$ and when selecting 768×2 parameters randomly, through absolute value and through Fisher information estimate. $\mathbf{b}_1, \mathbf{W}_2$ and $\mathbf{b}_2$ each train all of their parameters. For the Fisher method we get a new Fisher information estimate every 10,000 steps. The algorithm diverges when training all 768×768 parameters of $\mathbf{W}_1$ using the default maximal radius. We find that contrary to our expectations, the importance methods do not bring an improvement on the rate of convergence. The performance of the absolute value method is comparable to that of the random method while the Fisher method performs worse.

4.3 Radius Scheduling

We introduce a radius scheduler to each of the trainable tensors that reduces the maximum search radius as training goes on. We use a linear schedule s that scales from

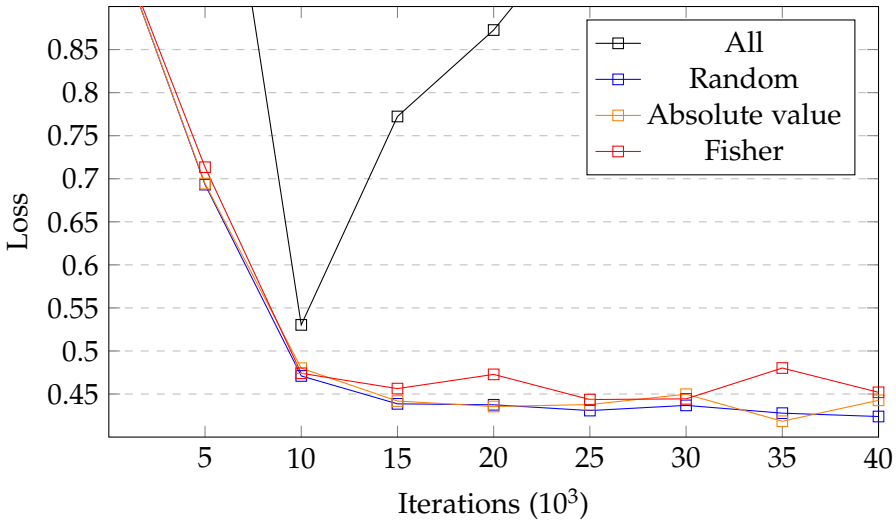

Figure 4.4: Validation loss of different parameter partitioning methods on $\mathbf{W}_1$ using default hyperparameters. Other tensors are not split.

1 to $h = 0.1$ over the course of training T steps:

$$s(t) = 1 - \frac{t(1-h)}{T} \tag{4.1}$$

Maximal radius at step t is calculated by multiplying the maximal radius of the tensor with the schedule:

$$R(\boldsymbol{\theta}_t) = R(\boldsymbol{\theta}) \cdot s(t) \tag{4.2}$$

We use the same schedule with all parameters as well as the same initial maximal radius. Figure 4.5 shows the validation loss for this experiment in comparison to the original result with no schedule. We find that without the radius schedule the model learns faster initially but overall loss is less stable and the radius scheduler outperforms at later steps. Still, a confident increase in performance is not observed with default hyperparameters. One of our hypotheses was that using a radius scheduler, the algorithm could retain its convergence with fewer searches. To test this, We redo the experiment with $K = 1$. The results are shown in Figure 4.6. We observe that the effect of the radius scheduler is amplified when doing only one search per iteration. In particular, it causes loss to more consistently decrease throughout the entire duration of training. While $K = 5$ still performs better, the gap is significantly smaller. We surmise that a smaller K could be used in conjunction with a radius scheduler when an increase in computation speed is desired.

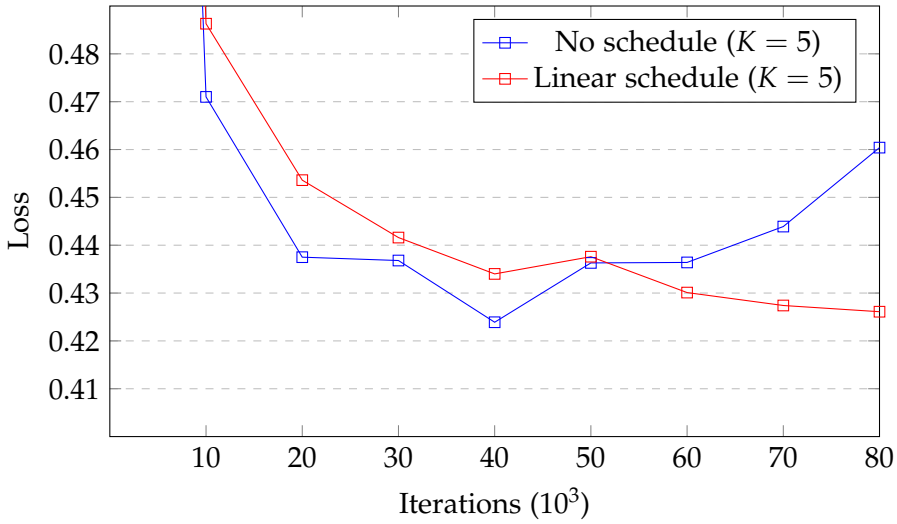

Figure 4.5: Validation loss with and without a linear radius scheduler using default hyperparameters

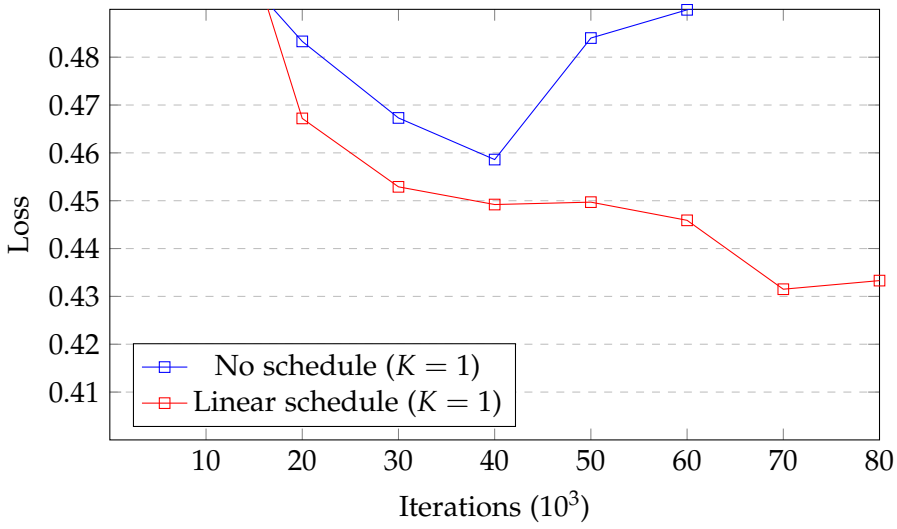

Figure 4.6: Validation loss with and without a linear radius scheduler when performing only one search per iteration

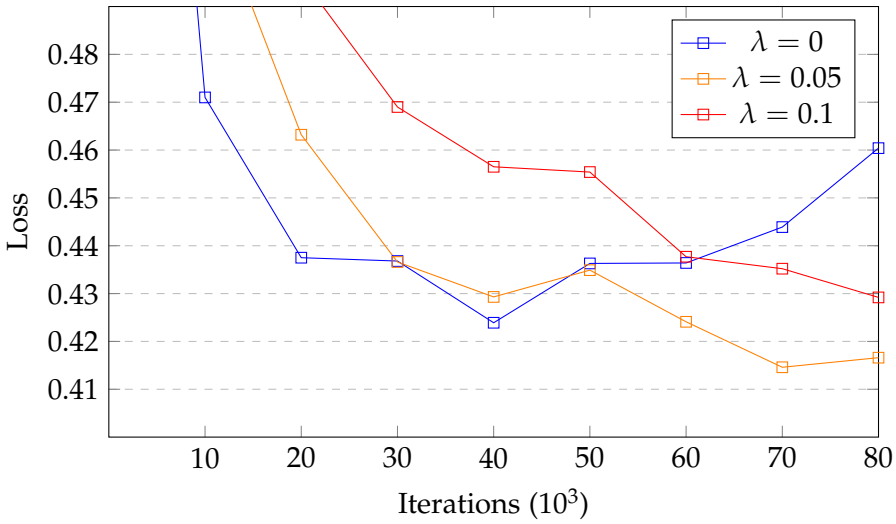

Figure 4.7: Validation loss using weight decay with different norm weights.

4.4 Weight Decay

We investigate the effects of adding an L_2-regularizer to the objective function:

$$\mathcal{J}(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \lambda \cdot \|\boldsymbol{\theta}\|_2 \tag{4.3}$$

where $\mathcal{L}$ is the cross-entropy loss and λ is the norm weight. Figure 4.7 compares three different values of λ. Our original experiment without weight decay has $\lambda = 0$. We observe a more stable decrease in loss with regularization. The hyperparameter λ plays a crucial role in the effectiveness of this method as it controls the balance between regularization and error minimization. We consider $\lambda = 0.05$ to be a good default value.

4.5 Options

Algorithm 2 introduces 2 additional hyperparameters: The number of option levels ℓ and the number of options per level m. Both hyperparameters have a significant effect on the time requirement of the algorithm: The number of forward passes each iteration is $n \cdot m^\ell \cdot K$ when training n tensors. Therefore, although we anticipate a reduction in update noise when using options, we have to pick ℓ and m conservatively in order to keep within a reasonable computation time. Figure 4.8 shows validation loss for four experiments with different values of m and ℓ, each of them named $m \times \ell$. We observe moderate improvements in loss as $m \times \ell$ increases with ℓ having a larger effect than m.

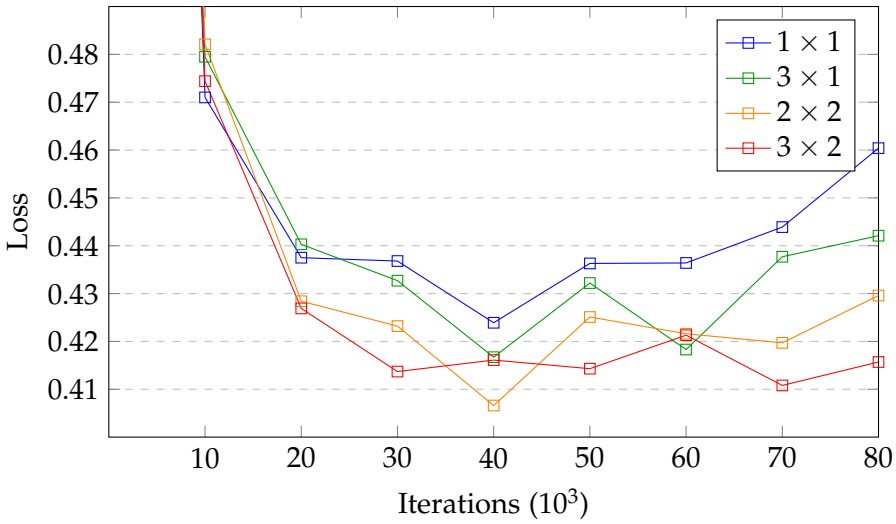

Figure 4.8: Validation loss for different values of option width and depth. We refer to m options per level with ℓ levels as $m \times \ell$.

4.6 Final Results & Preferred Method

To test the interoperability of our methods, we perform an experiment using both radius scheduling and weight decay at the same time. Figure 4.9 shows the validation loss for this experiment with and without using options. Figure 4.10 shows a repeat of this experiment with an increased maximal radius of $R = 7.5 \cdot 10^{-3}$. An experiment that uses this maximal radius without radius scheduling or weight decay is shown for comparison. We find that using both a linear radius schedule and weight decay allows the use of higher maximal search radii for faster convergence. This configuration yields our best results in terms of loss and thus becomes our preferred method. We find that the effects of using options are negligible with this configuration and hence opt not to use them. Table 4.2 compares our methods in terms of validation accuracy and memory consumption with a state-of-the-art Adam optimizer. We find that GLD uses slightly less memory than Adam. With a batch size of 64, GLD yields a slightly lower accuracy than Adam, but the accuracy gap closes with a batch size of 256.

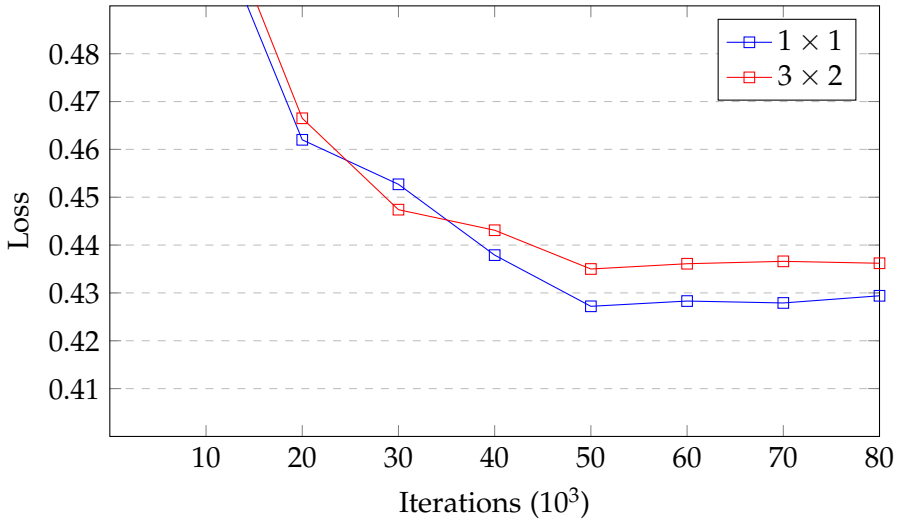

Figure 4.9: Validation loss for two experiments applying both a linear radius schedule and weight decay with $\lambda = 0.05$ using default hyperparameters.

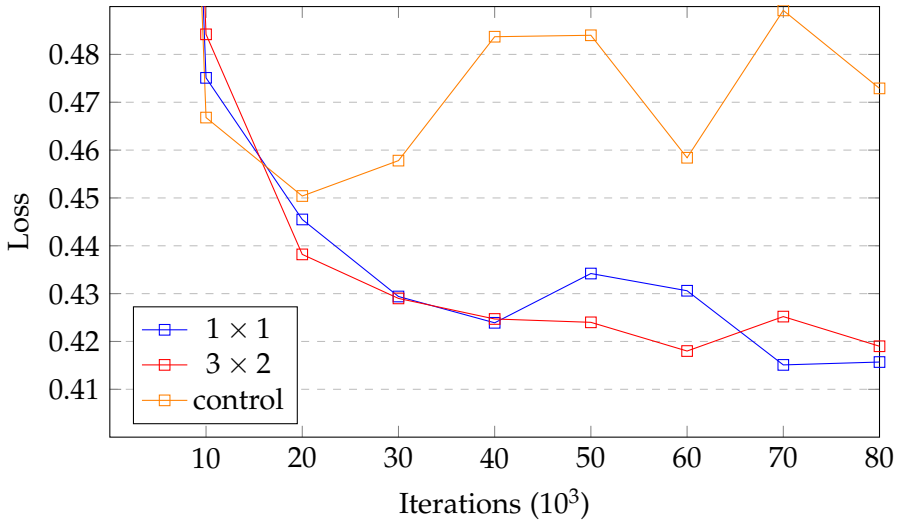

Figure 4.10: Validation loss for two experiments applying both a linear radius schedule and weight decay with $\lambda = 0.05$ using a maximal radius of $R = 7.5 \cdot 10^{-3}$. The third control experiment uses the same R but does not use a radius scheduler or weight decay.

Table 4.2: Comparison of GLD and Adam in terms of validation accuracy on SST-2 and memory consumption. The Adam optimizer uses a learning rate of $5 \cdot 10^{-5}$ and a linear learning rate schedule. GLD methods set $K = 5$ and $R = 5 \cdot 10^{-3}$. GLD* refers to our preferred method, setting $R = 7.5 \cdot 10^{-3}$ while applying a linear radius schedule and weight decay with $\lambda = 0.05$. Experiments with batch size 64 run for $80,000$ steps, those with batch size 256 run for $40,000$ steps. Validation accuracy is computed every $5,000$ steps.

Optimizer	**Batch Size**	**Validation Accuracy**	**Memory Consumption**
GLD	64	82.11%	409 MiB
GLD*	64	82.22%	409 MiB
GLD	256	84.17%	854 MiB
Adam	64	84.75%	421 MiB
Adam	256	84.40%	867 MiB

5 Discussion & Future Work

This chapter evaluates our approaches and suggests future research directions that arise from our work. We will discuss the strengths and shortcomings of our methods, as well as possible modifications that might improve the effectiveness and experiments that might provide useful results.

Our experiments show that the effectiveness of our methods depends largely on batch size where larger batch sizes consistently perform better. This outcome is hardly surprising as it has already been established for gradient methods that larger batch sizes result in better parameter updates by giving descent directions more representative of the entire dataset, reducing the probability of choosing parameters that give a worse overall representation. We suspect that this effect might be amplified for direct search methods since random parameter updates that do not make use of gradient information are also not necessarily in the optimal descent direction for a given batch. This might make it a necessity to pick a larger batch size than a first-order method in order to be able to compete with it. This of course seems counter-productive considering our main objective of reducing memory consumption. However, for larger networks, the memory gains from eliminating backpropagation can offset the additional memory usage due to the larger batch size. On the other hand, our experiments on the Adam optimizer did not yield a better accuracy when we increased the batch size, which might indicate that for larger batch sizes, the performance of our method could be on the same level as that of first-order methods using the same batch size. In any case, experimenting with larger batch sizes can provide valuable data.

Our method of partitioning parameters might not be optimal. Since we first consider individual tensors as a unit of training and only then further partition a unit if it is too large, our algorithms might spend excessive time on some parameters while seldom updating others. For example, our algorithm considers the same number of perturbations for the 1×2 bias of the output layer as the 768×768 weight matrix of the hidden layer, of which only a smaller partition of size $s = 768 \times 2$ is optimized in each step. A more clever way of splitting trainable parameters could gain efficiency in terms of convergence rate. One could, for example, consider the entire set of trainable parameters as a single large unit and select s indices at random from that large unit to train each step. While our original method stems from initial intuition and ease of implementation, it can also have the advantage of making sure all weights and biases

of the network are trained in each step.

The number of parameters s we select to train from from the large weight matrix is decided empirically. It copies the size of the second largest weight matrix that we train, because we have seen through experimentation that a matrix of that size could be trained by our algorithm. It is known that direct search methods lose performance with larger parameter dimensions and our experiments in Section 4.2 confirm that training the entire 768×768 matrix is not feasible with our algorithm. However, there could be a better value for s. We expect that with smaller s, the algorithm will still learn but with a slower convergence rate as fewer parameters are optimized each step. Conversely, larger values could provide faster convergence, but performance will suffer when s is too large. Experimenting with both smaller and larger values of s could lead to a better understanding of the effects of dimensionality on the algorithm.

We designed the method of using options as a way to direct the algorithm towards parameter updates that give a better representation of the entire dataset without increasing the batch size. Our experiments indicate that using options improves loss with no additional memory cost. However, there is a significant increase in time cost as the number of forward passes increases. Whether or not options should be used depends therefore on time constraints. Using deeper levels of options could bring further improvement, however becomes less practical as time usage will be excessive compared to a similarly performing first-order method. It could also be interesting to see how the performance gains are affected by batch size. Since larger batch sizes are already more representative of the entire dataset, improvements brought by options might become less impactful.

We find that using the linear radius schedule we define in Section 4.3 as well as weight decay, we achieve better results by increasing the initial radius. One could increase the minimal multiplier h of the scheduler for a flatter schedule where maximal radius changes slower in a smaller interval throughout training. It could also be interesting to experiment with different types of schedule function such as cosine annealing (Loshchilov and Hutter 2017) or to apply a warmup strategy (Goyal, Dollár, Girshick, et al. 2018).

One weakness of getting parameter updates solely through random perturbations is that previous update steps are not considered. Intuitively, an update direction that has improved the model in one training iteration should be more likely to improve it further than, for example, the exact opposite direction. Our algorithm, however, cannot make use of this assumption and samples both directions with the same probability when searching for improving perturbations. A possible improvement is to implement a system similar to momentum that factors in previous update directions when calculating random perturbations. Update candidates could then be computed by adding a vector accumulating previous updates with some momentum coefficient to

the random perturbation sampled from the original distribution, effectively shifting the distribution towards the region of perturbation vectors that have previously improved the model. The amount of improvement an update brings could also be part of the formulation. We think that being able to select better candidates to consider is an important step to optimizing the algorithm for time and that the described approach or a derivative thereof can improve the convergence rate by making it more likely for a sampled candidate to be improving the model.

Introducing parallelism could be another way of optimizing the algorithm for time. The key factor making the algorithm slow is the large number of forward passes it requires. Each training step requires multiple forward passes with the same batch. Computing those in separate processing units or threads of execution can speed up the algorithm significantly.

6 Conclusion

We have introduced a recipe for training neural networks using an optimizer based on the Gradientless Descent (GLD) algorithm (Golovin, Karro, Kochanski, et al. 2020). Being a zeroth-order optimization algorithm, GLD does not make use of gradient information when updating parameters. This removes the need for a backward pass on the network, which can have a significant impact on memory consumption. GLD is a direct search method, which means that unlike many state-of-the-art zeroth-order optimizers that have been used with neural networks, our method does not rely on an estimation of the gradient and instead selects parameter updates from a set of randomly sampled perturbations of the parameter. We think that direct search methods have the potential to create memory-efficient optimizers of neural networks, which motivates our work.

A known limitation of direct search methods is that their performance deteriorates as the number of variables increases (Kolda, Lewis, and Torczon 2003). Our main contribution is introducing a parameter partitioning method to split the set of trainable parameters of the model into smaller subsets in order to bypass this limitation. The size of the subsets is chosen empirically to be sufficiently small, such that GLD is able to train an individual subset with a reasonable rate of convergence. We are then able to train the entire network by cycling through the subsets in each iteration, training one subset at a time. Our method takes each individual weight matrix and bias vector as a subset, which is further divided if the size is too large by selecting parameters either randomly or based on importance, using the absolute value or an estimation of the Fisher importance matrix as the importance measure.

We also propose additional modifications on the algorithm with the aim of further improving its performance. We introduce radius scheduling, the application of a learning rate schedule onto the maximal search radius of GLD, and weight decay, which adds an L_2-norm regularizer to the objective function. Both methods are motivated by their established success in improving first-order methods and our experiments show that they are applicable for direct search methods as well. We introduce options, which is a method that implements a lookahead into potential future updates that are then considered when choosing the current update. We apply options as a way to choose parameter updates that are more representative of the entire training dataset without increasing the batch size.

To test our methods, we fine-tune a classifier attached to a pre-trained DistilBERT model for the binary classification task given by the SST-2 dataset. We obtain our best results by using both radius scheduling and weight decay on top of our main parameter partitioning method where we pick parameters at random when the weight matrix is too large. When compared to Adam, our method gives slightly reduced accuracy while using less memory. The difference in accuracy seems to decrease as batch size increases. We identify the slow speed of our algorithm as a major drawback. We think that future work can both further close the accuracy gap to state-of-the-art first-order methods and speed up the algorithm, addressing this drawback.

Abbreviations

NLP Natural Language Processing

ReLU Rectified Linear Unit

SGD Stochastic Gradient Descent

SPSA Simultaneous Perturbation Stochastic Approximation

GLD Gradientless Descent

LLM Large Language Model

STP Stochastic Three Points

FIM Fisher Importance Matrix

List of Figures

List of Tables

Bibliography

Ba, J. L., J. R. Kiros, and G. E. Hinton (2016). *Layer Normalization*. arXiv: 1607.06450.

Bahdanau, D., K. Cho, and Y. Bengio (2015). "Neural Machine Translation by Jointly Learning to Align and Translate." In: *ICLR*. arXiv: 1409.0473.

Bergou, E. H., E. Gorbunov, and P. Richtárik (2020). "Stochastic three points method for unconstrained smooth minimization." In: *SIAM Journal on Optimization* 30.4, pp. 2726–2749.

Berisha, V. and A. O. Hero (2015). "Empirical Non-Parametric Estimation of the Fisher Information." In: *IEEE Signal Processing Letters* 22.7, pp. 988–992. DOI: 10.1109/LSP.2014.2378514.

Bernstein, J., Y.-X. Wang, K. Azizzadenesheli, and A. Anandkumar (2018). "signSGD: Compressed optimisation for non-convex problems." In: *International Conference on Machine Learning*. PMLR, pp. 560–569.

Bielecki, A. (2019). *Models of Neurons and Perceptrons: Selected Problems and Challenges*. Springer Cham. Chap. 3, p. 15. ISBN: 978-3-319-90140-4.

Chen, P.-Y., H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh (2017). "ZOO: Zeroth Order Optimization Based Black-box Attacks to Deep Neural Networks without Training Substitute Models." In: *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*. AISec '17. Dallas, Texas, USA: Association for Computing Machinery, pp. 15–26. ISBN: 9781450352024. DOI: 10.1145/3128572.3140448.

Chen, X., S. Liu, K. Xu, X. Li, X. Lin, M. Hong, and D. Cox (2019). "Zo-adamm: Zeroth-order adaptive momentum method for black-box optimization." In: *Advances in neural information processing systems* 32.

Cheng, H., M. Zhang, and J. Q. Shi (2023). *A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations*. arXiv: 2308.06767.

Devlin, J., M.-W. Chang, K. Lee, and K. Toutanova (June 2019). "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Ed. by J. Burstein, C. Doran, and T. Solorio. Minneapolis, Minnesota: Association for Computational Linguistics, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://aclanthology.org/N19-1423.

Du, K.-L. and M. N. S. Swamy (2019). *Neural Networks and Statistical Learning*. 2nd ed. Springer London. ISBN: 978-1-4471-7452-3.

Friedman, J. H. and L. C. Rafsky (1979). "Multivariate generalizations of the Wald-Wolfowitz and Smirnov two-sample tests." In: *The Annals of Statistics*, pp. 697–717.

Gill, P. E., W. Murray, and M. H. Wright (1981). *Practical Optimization*. Academic Press. ISBN: 0-12-283952-8.

Goldberg, Y. (2016). "A primer on neural network models for natural language processing." In: *Journal of Artificial Intelligence Research* 57, pp. 345–420.

Golovin, D., J. Karro, G. Kochanski, C. Lee, X. Song, and Q. R. Zhang (2020). "Gradientless Descent: High-Dimensional Zeroth-Order Optimization." In: *International Conference on Learning Representations*.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. MIT Press. URL: http://www.deeplearningbook.org.

Gorbunov, E., A. Bibi, O. Sener, E. H. Bergou, and P. Richtárik (2020). "A stochastic derivative free optimization method with momentum." In: *International Conference on Learning Representations*.

Goyal, P., P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He (2018). *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*. arXiv: 1706.02677.

Han, S., J. Pool, J. Tran, and W. Dally (2015). "Learning both Weights and Connections for Efficient Neural Network." In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc.

He, K., X. Zhang, S. Ren, and J. Sun (2016). "Deep Residual Learning for Image Recognition." In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

Hinton, G., O. Vinyals, and J. Dean (2015). *Distilling the Knowledge in a Neural Network*. arXiv: 1503.02531.

Jurafsky, D. and J. H. Martin (Feb. 2024). *Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Third Edition Draft. URL: https://web.stanford.edu/~jurafsky/slp3/ (visited on 05/30/2024).

Keskar, N. S., D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang (2017). "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Kiefer, J. and J. Wolfowitz (1952). "Stochastic Estimation of the Maximum of a Regression Function." In: *Annals of Mathematical Statistics* 23, pp. 462–466.

Kingma, D. and J. Ba (Dec. 2014). "Adam: A Method for Stochastic Optimization." In: *International Conference on Learning Representations*.

Kolda, T. G., R. M. Lewis, and V. Torczon (2003). "Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods." In: *SIAM Review* 45.3, pp. 385–482. DOI: 10.1137/S003614450242889.

Lialin, V., V. Deshpande, and A. Rumshisky (2023). *Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning*. arXiv: 2303.15647.

Lin, J. and L. Rosasco (2017). "Optimal Rates for Multi-pass Stochastic Gradient Methods." In: *Journal of Machine Learning Research* 18.97, pp. 1–47. URL: http://jmlr.org/papers/v18/17-176.html.

Liu, S., P.-Y. Chen, X. Chen, and M. Hong (2019). "signSGD via Zeroth-Order Oracle." In: *International Conference on Learning Representations*.

Liu, S., P.-Y. Chen, B. Kailkhura, G. Zhang, A. O. Hero III, and P. K. Varshney (2020). "A Primer on Zeroth-Order Optimization in Signal Processing and Machine Learning: Principals, Recent Advances, and Applications." In: *IEEE Signal Processing Magazine* 37.5, pp. 43–54. DOI: 10.1109/MSP.2020.3003837.

Liu, S., P. Ram, D. Vijaykeerthy, D. Bouneffouf, G. Bramble, H. Samulowitz, D. Wang, A. Conn, and A. Gray (Apr. 2020). "An ADMM Based Framework for AutoML Pipeline Configuration." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34, pp. 4892–4899. DOI: 10.1609/aaai.v34i04.5926.

Loshchilov, I. and F. Hutter (2017). "SGDR: Stochastic Gradient Descent with Warm Restarts." In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Malladi, S., T. Gao, E. Nichani, A. Damian, J. D. Lee, D. Chen, and S. Arora (2023). "Fine-Tuning Language Models with Just Forward Passes." In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., pp. 53038–53075.

McCulloch, W. S. and W. Pitts (1943). "A logical calculus of the ideas immanent in nervous activity." In: *Bulletin of Mathematical Biology* 5, pp. 115–133.

Nesterov, Y. (1983). "A method for solving the convex programming problem with convergence rate $O(\frac{1}{k^2})$." In: *Proceedings of the USSR Academy of Sciences* 269, pp. 543–547.

Pachal, S., S. Bhatnagar, and L. A. Prashanth (2023). *Generalized Simultaneous Perturbation-based Gradient Search with Reduced Estimator Bias*. arXiv: 2212.10477.

Polyak, B. (Dec. 1964). "Some methods of speeding up the convergence of iteration methods." In: *Ussr Computational Mathematics and Mathematical Physics* 4, pp. 1–17. DOI: 10.1016/0041-5553(64)90137-5.

Reddi, S. J., S. Kale, and S. Kumar (2018). "On the Convergence of Adam and Beyond." In: *International Conference on Learning Representations*.

Reed, R., R. Marks, and S. Oh (1995). "Similarities of error regularization, sigmoid gain scaling, target smoothing, and training with jitter." In: *IEEE Transactions on Neural Networks* 6.3, pp. 529–538. DOI: 10.1109/72.377960.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (July 1986). "Learning Internal Representations by Error Propagation." In: *Parallel Distributed Processing, Volume 1: Explorations in the Microstructure of Cognition: Foundations*. The MIT Press. ISBN: 9780262291408. DOI: 10.7551/mitpress/5236.003.0012.

Sanh, V., L. Debut, J. Chaumond, and T. Wolf (2020). *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. arXiv: 1910.01108.

Socher, R., A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts (Oct. 2013). "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank." In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, pp. 1631–1642. URL: https://www.aclweb.org/anthology/D13-1170.

Spall, J. (1992). "Multivariate stochastic approximation using a simultaneous perturbation gradient approximation." In: *IEEE Transactions on Automatic Control* 37.3, pp. 332–341. DOI: 10.1109/9.119632.

Sutskever, I., J. Martens, G. Dahl, and G. Hinton (June 2013). "On the importance of initialization and momentum in deep learning." In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by S. Dasgupta and D. McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, pp. 1139–1147. URL: https://proceedings.mlr.press/v28/sutskever13.html.

Tu, M., V. Berisha, M. Woolf, J.-s. Seo, and Y. Cao (2016). "Ranking the parameters of deep neural networks using the fisher information." In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2647–2651. DOI: 10.1109/ICASSP.2016.7472157.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017). "Attention is All you Need." In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc.

Widrow, B. and M. Lehr (1990). "30 years of adaptive neural networks: perceptron, Madaline, and backpropagation." In: *Proceedings of the IEEE* 78.9, pp. 1415–1442. DOI: 10.1109/5.58323.