Decoupling the Method of Auxiliary Coordinates to Distributively Train Deep Neural Networks

Paul Martin

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2024

Abstract

As deep neural networks grow larger, efficient and distributed training approaches become increasingly relevant. While existing distributive training methods like data parallelism and pipeline parallelism can accelerate training, the former assumes that the full model can fit on a single GPU, and the latter, depending on the exact approach used, either underutilises the GPUs or introduces training inconsistencies. This thesis investigates the Method of Auxiliary Coordinates (MAC) as an alternative optimisation framework designed for distributed training by splitting up a model into its constituent layers and training them in parallel on auxiliary tasks, regularly synchronising these tasks.

The thesis in particular proposes extensions to MAC to enable its application to modern deep learning architectures and allow it to be trained on various tasks. One key contribution is the explicit formulation of two approaches to decoupling the internal workings of MAC. This decoupling improves its distributability and reduces the data that is communicated between machines during training. Support for custom loss functions beyond mean squared error is also added.

A theoretical analysis quantifies the potential speedup offered by distributed MAC over standard techniques like stochastic gradient descent or Adam. Empirical evaluations on phone classification tasks demonstrate MAC's ability to train multi-layer perceptrons as well as Transformer-based models to lower losses than SGD in less time. The results also provide insights into how to select appropriate hyperparameters for MAC. The impact of choosing where to split a model is also evaluated.

While further research is needed into principled hyperparameter tuning and reduction of data transfer during training, this work finds MAC to be a promising optimisation framework for distributively training deep learning models.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Paul Martin*)

Acknowledgements

Thank you, Hao! Not only for the incredible support that you have shown me over the two years and for setting a high bar for what a great supervisor should be, but also for the insights you have shared that I could learn so much from. Our meetings, whether in a group or one-on-one were always valuable, even when some were more productive than others. I also enjoyed our occasional chats and laughs, which made this journey more enjoyable. Let's keep in touch!

Thank you also to you, Julia, for being by my side these years! And more recently, thank you for encouraging/pestering me to get over the cluster issues and lost data, re-evaluate my experiments and thesis structure, and get this write-up done without relying on special circumstances. It made a difference. I love you!

Table of Contents

Chapter 1

Introduction

Deep learning has rapidly become the dominant approach across various domains, with larger models pushing the boundaries of computer vision, natural language processing and speech recognition (Chai et al., 2021; Roger et al., 2022; Torfi et al., 2020). However, as models grow larger, training becomes slower and more costly, computationally and environmentally (Dhar, 2020; Lacoste et al., 2019). Hence, the need for efficient and scalable training methods becomes increasingly important.

Over the years several approaches have been proposed and are being used to distribute models across multiple machines to speed up their training (Nichols et al., 2021). However, most of these approaches rely on standard optimisation techniques such as stochastic gradient descent (SGD), which are inherently sequential and require each datapoint to be passed through the full network (and propagated back) before the model is updated and the next iteration can begin. This restriction limits the parallelism that can be achieved.

1.1 Motivation

The Method of Auxiliary Coordinates (MAC), proposed by Carreira-Perpiñán and Wang in 2014, presents an alternative optimisation framework that is designed to be distributable. By introducing auxiliary variables ('coordinates') after each layer, MAC effectively splits up the model, allowing each layer's parameters to be updated in parallel on separate machines.

MAC's fundamentally different approach to parallelism makes it an interesting optimisation method to investigate. Although this thesis merely investigates it out of curiosity, if the results are promising, MAC could even be a genuine alternative to some of the existing parallelism approaches.

1.2 Previous work (MInf 1)

In the pursuit of efficiency and modularity, last year's MInf 1 project (Martin, 2023) investigated the use of cross-architecture knowledge distillation, where a student network

gets trained to copy the output of a larger teacher network of a different architecture, to speed up the inference time of end-to-end speech recognition models. To this end, I first developed a guideline on how best to initialise the student models, and further proposed an algorithm to enable the distillation between two models with mismatched output dimensions. Both, the initialisation guideline and especially the distillation algorithm are useful to research outside of the MInf 1 project to enable and improve cross-architecture knowledge for speech recognition.

This thesis is a continuation of this pursuit, investigating how we can speed up training by modularising a network – splitting it up, training each layer on a different machine, and reassembling for inference. Similarly to the previous project, I will be using a speech processing task, specifically phone classification, to evaluate the method's effectiveness.

1.3 Contributions

The primary contributions of this thesis are:

- Extensions to the Method of Auxiliary Coordinates to enable its application to modern deep learning architectures and tasks beyond autoencoding. This is demonstrated by training an MLP and a Transformer-based model on phone classification.
- Proposal of two approaches for decoupling the auxiliary coordinate updates in the Z-Step to significantly improve distributability.
- A theoretical analysis of the potential speedup offered by distributed MAC over standard techniques like SGD or Adam.
- An empirical evaluation of MAC's performance on phone classification, again comparing it to SGD and Adam.
- Insights into effectively leveraging MAC's performance by splitting models into appropriate groups and selecting good hyperparameters.

1.4 Structure

Following this introduction, Chapter 2 begins by providing the relevant background on optimisation techniques for deep learning and details on the original formulation of the Method of Auxiliary Coordinates by Carreira-Perpiñán and Wang. It also highlights key limitations of their work, thereby motivating the extensions proposed in this thesis. Related attempts at accelerating the training of large models as well as an introduction to speech processing and phone classification are also covered in brief. Following this, Chapter 3 presents the core extensions, in particular the formulations for partially and fully decoupling the auxiliary coordinate updates to enable a more efficient distribution of MAC. It also discusses how MAC can be applied to modern architectures, custom loss functions, and dropout. Importantly, the chapter also provides a theoretical analysis of the potential speedup offered by distributed MAC. Chapter 4

then details the experimental evaluation, comparing the convergence and computational performance of MAC against SGD and Adam on phone classification tasks. It also provides insights into setting hyperparameters and grouping layers to effectively utilise MAC for optimisation. Finally, Chapter 5 summarises the key contributions, limitations, and directions for future work.

Chapter 2

Background

In this chapter, I will introduce the relevant background to understand the work presented in this thesis. I will begin by introducing two common optimisers used in deep learning, Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (Adam). Following this, I will introduce the original formulation of the Method of Auxiliary Coordinates (MAC) as proposed by Carreira-Perpiñán and Wang (2014), which forms the basis of my work and discuss its limitations, thus highlighting the importance of my further research into this method. I will also briefly cover other methods that have been proposed to speed up training, as well as a primer on speech processing and specifically the phone classification task used in Chapter 4.

2.1 Optimisation in Deep Learning

In deep learning, optimisation is the process by which we determine the parameters for a model such that it produces the desired output. This (supervised) training is typically done on a dataset of known input-output pairs $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}$. The model with parameters $\boldsymbol{\theta}_t$ first predicts an output $\hat{\mathbf{y}}^{(i)} = \mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta}_t)$, which are then compared to the ground truth $\mathbf{y}^{(i)}$ to compute the error. The errors are gathered across the full dataset to compute the loss function $\mathcal{L}(\boldsymbol{\theta}_t) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)})$, where N is the number of samples in the dataset. The goal is to find the parameters $\boldsymbol{\theta}^*$ that minimise this loss, which is typically done using gradient descent. Since the gradient of a function at a given point indicates the direction and magnitude of the steepest ascent, the negative gradient points in the direction of the steepest descent. Thus, we update the parameters iteratively by moving through the parameter space (a vector space) in the direction of the negative gradient until the loss converges to a minimum. Formally, the optimisation problem is

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\mathbf{y}^{(i)}, \mathbf{f}(\mathbf{x}^{(i)}; \boldsymbol{\theta})). \tag{2.1}$$

As a simple example, let us consider a one-layer perceptron used to classify a sound into one of 60 phones (the smallest distinct unit of speech) or silence. Assuming that

our sound recording is represented by a 40-dimensional vector $\mathbf{x}^{(i)}$ (for more details see Section 4.1), the classification involves first multiplying the input vector by a weight matrix $W \in \mathbb{R}^{61\times 40}$ and then passing the result through a softmax function $\boldsymbol{h}(\cdot)$ to obtain the predicted probability vector $\hat{\mathbf{y}} = \boldsymbol{h}(W\mathbf{x})$, where each element represents the predicted probability of the sound being the respective phone or silence. We can use cross-entropy to compute the loss as $\mathcal{L}(W) = -\sum_{i=1}^{N} \mathbf{y}^{(i)} \log(\hat{\mathbf{y}}^{(i)})$, where $\mathbf{y}^{(i)}$ is the one-hot encoded vector of the ground truth phone. Next, the gradient of this loss $\nabla_W \mathcal{L}(W)$ with respect to the parameters W is computed via backpropagation (Rumelhart et al., 1986), and the parameters are iteratively updated as

$$W \leftarrow W - \eta \nabla_W \mathcal{L}(W), \tag{2.2}$$

where η is the learning rate. The matrix $\nabla_W \mathcal{L}(W)$ is also known as the Jacobian matrix.

As the model's number of layers (its 'depth') increases, the model's functional representation becomes increasingly nested. For instance, in a two-layer perceptron, the prediction would be $\hat{\mathbf{y}} = \boldsymbol{h}(W_1\, \boldsymbol{g}(W_0\, \mathbf{x}))$, where $\boldsymbol{g}$ is a nonlinearity function, for which this thesis uses the Rectified Liner Unit (ReLU; Agarap, 2018).

2.1.1 Stochastic Gradient Descent (SGD)

Two drawbacks of regular gradient descent are that it assumes the loss function to be strongly convex (virtually never the case in practice) and that it requires loss to be computed across the full dataset for each iteration, which can become very expensive and slow for large datasets.

Stochastic Gradient Descent (SGD) addresses this issue by computing the gradient based on a single training example at each iteration. Instead of summing the errors across the full dataset, SGD randomly selects one data point $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ and computes the gradient only from that point:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) \tag{2.3}$$

for model parameters $\boldsymbol{\theta}$.

This significantly reduces the computational cost per iteration compared to gradient descent as described above, where we compute the loss over the full dataset.

However, using a single data point also introduces significant noise into the parameter updates due to randomness in sampling. This noisy gradient means that SGD does not follow a direct path towards the minimum like gradient descent, but instead exhibits a "zigzagging" behavior in the parameter space. While this erratic movement can help SGD escape shallow local minima, it can also lead to overshooting the global minimum and failing to converge.

To balance the tradeoff between the fast computations of SGD and the smooth convergence of gradient descent, a common compromise is to use 'mini-batch' SGD. Here, instead of a single data point, a small subset or 'mini-batch' of b data points $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{b}$ is sampled from the dataset at each iteration:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \left(\frac{1}{b} \sum_{i=1}^{b} \mathcal{L}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) \right) \tag{2.4}$$

Using a mini-batch reduces the noise from the parameter updates compared to true SGD, while still being more computationally efficient than gradient descent across the full dataset for large datasets (Bilmes et al., 1997).

2.1.2 Adaptive Moment Estimation (Adam)

Adaptive Moment Estimation (Adam), proposed by Kingma and Ba (2015) is meant to improve upon SGD by utilising a per-parameter learning rate. Each parameter's learning rate is adapted over time based on the first and second moments of the gradients (its mean and variance), although the exact equations are not relevant to this thesis. Adam has been shown to work well in practice and often converges faster than SGD (Kingma and Ba, 2015; See Section 4.2).

Both SGD and Adam will serve as important baselines for my work, and I will be specifically using Adam as part of my implementations.

Both of these optimisers are iterative first-order optimisers, as they use the first partial derivative of the loss with respect to each parameter, arranged in a Jacobian matrix. In contrast, second-order optimisers, such as Newton's method or its approximation, the Gauss-Newton method (Nocedal & Wright, 2006), use the Hessian matrix, consisting of the second derivatives with respect to each parameter. While second-order optimisers usually require fewer iterations to converge than first-order ones, the required Hessian is also significantly more expensive to compute. I am mentioning these, as they are used by Carreira-Perpiñán and Wang in their work on the Method of Auxiliary Coordinates, which I introduce in Section 2.2.

2.2 The Method of Auxiliary Coordinates (MAC)

Following the typical training procedure using SGD or Adam, we compute a full forward-pass (to compute the loss) but also a full backward-pass (to compute the updates to each parameter). Not only is this computationally expensive, but it also doesn't lend itself well to distributed training, as we need to wait for the full forward pass to be completed to start the backwards pass, which in turn needs to be completed for the next iteration to start. While some approaches exist to mitigate this, such as pipeline parallelism (Harlap et al., 2018; Y. Huang et al., 2019), they have their own problems including high communication costs and inconsistencies in the model's state across the different machines (Nichols et al., 2021; see Section 2.4).

Carreira-Perpiñán and Wang's proposed 'Method of Auxiliary Coordinates' (MAC; 2014) appears to be a promising alternative to the default training procedure, allowing for a more distributed training procedure. In their experiments, they find a speedup of approximately $5\times$ over SGD, although I explain in Section 2.3 why these results may not be entirely accurate for modern use cases of MAC.

Given a model, defined as a series of layers $f_0, \ldots, f_L$ and their corresponding parameters $\boldsymbol{\theta}_0, \ldots, \boldsymbol{\theta}_L$,

$$\hat{\mathbf{y}} = f(\mathbf{x};\, \boldsymbol{\theta}) = f_L\Big(f_{L-1}\big(\, \ldots\, f_0(\boldsymbol{x};\, \boldsymbol{\theta}_0)\, \ldots\, ;\boldsymbol{\theta}_{L-1}\big);\, \boldsymbol{\theta}_L\Big), \tag{2.5}$$

we can split up the notation by introducing auxiliary coordinates:

$$\hat{\mathbf{y}} = f(\mathbf{x};\, \boldsymbol{\theta}) = f_L(\mathbf{z}_L), \quad \mathbf{z}_{\ell+1} = f_\ell(\mathbf{z}_\ell;\, \boldsymbol{\theta}_\ell), \quad \mathbf{z}_0 = \mathbf{x}. \tag{2.6}$$

2.2.1 Mathematical Formulation

Given a dataset $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^N$, and using the mean squared error as the loss function, we can define the original objective function from Equation 2.1 as

$$\begin{aligned} \boldsymbol{\theta}^* &= \operatorname*{argmin}_{\boldsymbol{\theta}}\, E_1(\boldsymbol{\theta}) \\ \text{for} \quad E_1(\boldsymbol{\theta}) &= \frac{1}{N}\sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f(\mathbf{x}^{(i)};\, \boldsymbol{\theta}) \right\|^2. \end{aligned} \tag{2.7}$$

Using the auxiliary coordinates introduced in Equation 2.6, we can reformulate the optimisation problem as a constrained optimisation problem, where we optimise over both the model parameters $\boldsymbol{\theta}$ and the auxiliary coordinates $\mathbf{z}$.

For a model split into two parts, $\hat{\mathbf{y}} = f_1(\mathbf{z};\, \boldsymbol{\theta}_1)$, $\mathbf{z} = f_0(\mathbf{x};\, \boldsymbol{\theta}_0)$, the reformulated Equation 2.7 is

$$E(\boldsymbol{\theta}, Z) = \frac{1}{N}\sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f_1(\mathbf{z}^{(i)};\, \boldsymbol{\theta}_1) \right\|^2, \quad \text{s.t.} \quad \mathbf{z}^{(i)} = f_0(\mathbf{x}^{(i)};\, \boldsymbol{\theta}_0). \tag{2.8}$$

for $Z = \{\mathbf{z}^{(i)}\}_{i=1}^N$ being the set of all auxiliary coordinates.

For deeper networks with L layers $f_0, \ldots, f_L$, this trivially extends to

$$E(\boldsymbol{\theta}, Z) = \frac{1}{N}\sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)};\, \boldsymbol{\theta}_L) \right\|^2, \quad \text{s.t.} \quad \mathbf{z}_{\ell+1}^{(i)} = f_\ell(\mathbf{z}_\ell^{(i)};\, \boldsymbol{\theta}_\ell), \quad \mathbf{z}_0^{(i)} = \mathbf{x}^{(i)}. \tag{2.9}$$

The constrained problem stated above can be solved using penalty-based methods that alternate between updating the parameters $\boldsymbol{\theta}$ and the auxiliary coordinates $\mathbf{Z}$. Importantly, each update step involves only a single layer rather than the entire network, enabling distributed optimisation.

2.2.2 Optimisation using Quadratic Penalties

While the constrained optimisation problem in Equation 2.9, can be solved using various constrained numerical optimisation methods, Carreira-Perpiñán and Wang specifically suggest the Quadratic Pentalty approach (Nocedal & Wright, 2006) in their

paper. This approach works similarly to how the analytical optimisation method of Lagrange Multipliers (*Encyclopaedia of Mathematics*, 2002) enforces the constraints by multiplying each constraint by a Lagrange Multiplier μ and adding them to the original optimisation problem:

$$\begin{aligned} \boldsymbol{\theta}^*, Z^* &= \underset{\boldsymbol{\theta}, Z, \mu}{\operatorname{argmin}} \; E_Q(\boldsymbol{\theta}, Z; \mu) \\ \text{for} \quad E_Q(\boldsymbol{\theta}, Z; \mu) &= \frac{1}{2N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}) \right\|^2 \\ &\quad + \frac{\mu}{2N} \sum_{\ell=0}^{L-1} \sum_{i=1}^{N} \left\| \mathbf{z}_{\ell+1}^{(i)} - f_\ell(\mathbf{z}_\ell^{(i)}; \boldsymbol{\theta}_\ell) \right\|^2 \end{aligned} \tag{2.10}$$

The first term is the original objective function from Equation 2.7, while the second term penalizes violations of the constraints, weighted by μ. As $\mu \to \infty$, minimizing E_Q forces the constraints to be satisfied exactly, recovering the original constrained problem.

2.2.3 Alternating Optimisation Steps

To minimise the quadratic penalty function (2.10), we alternatingly optimise over the parameters $\boldsymbol{\theta}$ and auxiliary coordinates Z [1] in what Carreira-Perpiñán and Wang call the W-step and Z-step, respectively:

W-step: Given the auxiliary coordinates Z, each layer's parameters $\boldsymbol{\theta}_\ell$ are optimised (trained) separately while keeping the other layers and the auxiliary coordinates fixed:

$$\boldsymbol{\theta}_\ell \leftarrow \underset{\boldsymbol{\theta}_\ell}{\operatorname{argmin}} \; E_Q(\boldsymbol{\theta}, Z; \mu). \tag{2.11}$$

Z-step: Since all the datapoints $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ are independent, their corresponding auxiliary coordinates $\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)}$ (jointly denoted as $\mathbf{z}^{(i)}$) are also independent from any other $\mathbf{z}^{(j)}$ for $j \in \{1, \ldots, N\} \setminus \{i\}$. Hence, the set of auxiliary coordinates $\mathbf{z}^{(i)}$ of each datapoint can be updated separately while keeping the other $\mathbf{z}$s and model parameters fixed:

$$\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)} \leftarrow \underset{\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)}}{\operatorname{argmin}} \; E_Q(\boldsymbol{\theta}, Z; \mu). \tag{2.12}$$

[1] This alternating optimisation of θ and Z resembles the Method of Coordinate Descent, where the parameters are updated one at a time while keeping the others fixed, which is sometimes used when 'a calculation of the derivatives involves a large amount of computation' (*Encyclopaedia of Mathematics*, 2002).

It is important to note that while the auxiliary coordinates are independent across datapoints, the coordinates $\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)}$ originating from the same input $\mathbf{x}^{(i)}$ are still coupled.

More details on the W and Z-Step update rules are provided in Chapter 3, as well as a thorough discussion of how the $\mathbf{z}$s are coupled across layers, and how they can be decoupled effectively.

2.3 Limitations of Carreira-Perpiñán and Wang's Work

While Carreira-Perpiñán and Wang's work appears promising, with their experiments showing a speedup of approximately $5\times$ over SGD, it has some notable limitations. Some of these can be attributed to the rapid advancement of deep learning since their first publication on this method in 2012, while other limitations are inherent to their specific approaches. In the following, I will discuss some of the key limitations. More meticulous discussion on the specific limitations of their Z-Step formulation and implementation are provided in Section 3.1, where I will also address these limitations through my own work.

2.3.1 Limitations due to Age

Carreira-Perpiñán and Wang first published a preprint of their work in 2012 and later published the journal article in 2014. Since then, various new network architectures, optimisers, training paradigms, as well as computer architectures have been developed to aid the training of more complex models on larger datasets.

Network Architecture The experiments by Carreira-Perpiñán and Wang focus primarily on Radial Basis Function (RBF) networks and feedforward neural networks with sigmoid activations. However, in the years since their publication, other architectures have become popular. In particular, Transformer networks (Vaswani et al., 2017) have seen widespread adoption in natural language processing tasks for handling text and speech data, and have even been adapted for vision tasks (Chai et al., 2021; Dosovitskiy et al., 2021; Torfi et al., 2020). Although the Method of Auxiliary Coordinates can likely be adapted to these architectures, it is not clear how well it would perform, and how fast it would converge, given the increased complexity of these architectures. Specifically, the use of architectural features such as skip connections and layer normalisation, or dropout regularisation (Ba et al., 2016; He et al., 2016; Hinton et al., 2012), in Transformers could impact the suitability of the Method of Auxiliary Coordinates.

Processors The authors report speedups of approximately 5x over stochastic gradient descent (SGD). However, these comparisons were performed using a 16-core CPU, whereas GPUs have since become the standard for training deep neural networks. Their parallel architecture and focus on single instruction, multiple data (SIMD) operations make GPUs well-suited for the expensive matrix multiplications common in the forward and backward passes of neural networks. Similar to how my MInf 1 project showed that different network architectures (CNNs vs Transformers) are better suited for CPUs

and GPUs, respectively, the Method of Auxiliary Coordinates may benefit more or less than optimisers like SGD or Adam from the parallelism of GPUs.

2.3.2 Limitations in their Approaches

Not all limitations of Carreira-Perpiñán and Wang's work can be attributed to the age of their work. Some limitations are inherent to their specific approaches and experiments.

Optimisation Algorithms The experiments in their work use the second-order Gauss-Newton method to solve both the W and Z-steps. While this method may be suitable for small networks, the computational cost of computing the Hessian for each layer in the W-Step and across all datapoints in the Z-Step may become prohibitive for large models and large datasets.

Dataset Size While the authors' use of comparatively small datasets is justified for fast implementation, iteration and verification of their method, their used USPS dataset (Hull, 1994) with 5 thousand 16×16 images and the COIL-20 dataset (Nene et al., 1996) with 1368 images (although in a higher resolution), were already small at the time of their publication. For instance, the MNIST (LeCun et al., 1998) dataset with 60 thousand 28×28 images, which is these days considered more of a 'toy' dataset, was already published for 14 years at the time of their publication.

However, even MNIST is very small and easy to train (a 2-layer multi-layer perceptron with 200k parameters can achieve a cross-entropy loss of near 0 within seconds on a Laptop CPU), compared to more common datasets such as ImageNet (Deng et al., 2009) with over 3 million images for classification, or speech recognition datasets such as the 80-hour Wall Street Journal (Paul & Baker, 1992), the up to 960-hour LibriSpeech (Panayotov et al., 2015) or the 60k-hour Libri-Light dataset (Kahn et al., 2020) common for self-supervised training.

MAC's scalability to larger datasets requires further evaluation.

Applications Since Carreira-Perpiñán and Wang motivate their work as being an extension of an autoencoding technique which they first published a few years prior (Carreira-Perpiñán & Lu, 2008), the experiments for MAC are also focused on autoencoding tasks for images. Specifically, they also restrict themselves to using the reconstruction mean squared error (MSE) as the objective function. Investigating the effectiveness of MAC for more diverse applications as well as other losses would provide insight into its more general applicability.

In summary, while Carreira-Perpiñán and Wang's work on the Method of Auxiliary Coordinates (MAC) presents a promising approach for distributed optimisation of deep neural networks, it has several limitations. Some of these limitations arise from the rapid advancements in deep learning since the original publication, such as the development of new architectures, processors, and optimisation algorithms. Other limitations are inherent to the specific approaches used in their work, such as their focus on autoencoding tasks for two comparatively small image datasets. Although

their follow-up publication (Carreira-Perpiñán & Alizadeh, 2019) does use larger datasets, all other mentioned limitations including in particular the use of CPUs, despite GPUs already being commonplace, remain in place. Addressing these limitations and extending MAC to modern deep learning settings could help provide insight into its suitability and possibly even help establish it as a promising framework for the distributed training of deep neural networks.

2.4 Related work on accelerating training

Besides MAC, other methods for accelerating the training of deep neural networks have been proposed. In this section, I introduce three different types of parallelism that have been and are currently used to distribute training, and also briefly mention two other approaches for speeding up model training that do not rely on parallelism.

Data Parallelism In data parallelism, each machine possesses an identical copy of the full model and a subset of the data. Each machine does a full forward and backward pass for its data to compute the gradients. When all machines are done, all of the gradients are accumulated, the model parameters are updated centrally and the model is sent to all machines. Ignoring communication costs, this leads to an approximately linear speedup with the number of GPUs (Nichols et al., 2021), however, it assumes that the entire model can fit on each GPU.

Model Parallelism When a model is too large to fit on a single GPU, we can instead use model parallelism, where each machine possesses only a subset of the model layers. For a single forward pass, the first machine with the first group of layers passes the input through its layers. The output is sent to the next machine with the next group of layers, which passes the last machine's output through its layers and sends its own output to the next machine, etc. until the data has passed through the whole model. The gradients are passed through the machines in reverse. However, as a model update requires a forward and backward pass through all layers, the machines are idle for most of the time. No speedup over regular SGD is achieved.

Pipeline Parallelism This can be seen as a combination of model and data parallelism. As with model parallelism, different machines handle different layers, but instead of waiting for the first batch's full gradient to be computed, the machines pass multiple batches forward before the first batch is done. Examples of this were proposed by Y. Huang et al. (2019) and Harlap et al. (2018). Generally, this converges faster than simple model parallelism, but slower than data parallelism (Harlap et al., 2018; Nichols et al., 2021).

Two other approaches that claim to accelerate training while not relying on parallelism were proposed by G. Huang et al. (2016) and Larsson et al. (2017). These approaches both rely on shortening the network during training, either by stochastically skipping layers or by randomly replacing a pair of layers with a single layer during training, respectively.

2.5 Speech Recognition and Phone Classification

In continuation of my MInf 1 project (Martin, 2023), this thesis trains deep neural networks for speech processing. Specifically, the task of phone classification is chosen.

In speech recognition, audio signals are digitally represented as waveforms, capturing air pressure fluctuations over time. The sample rate, often set at 16kHz for speech data, determines the number of discrete samples recorded per second.

Another common representation is the log-mel-spectrogram. It is derived from the spectrogram, which visualises the frequencies of a waveform as time progresses, by applying a Discrete Fourier Transform on sliding windows of a certain length and stride. The mel-scale, proposed by Stevens et al. (1937), is then applied to the spectrogram to better match human auditory perception, which is more sensitive to lower frequencies. This is achieved by averaging overlapping groups of frequencies, with smaller groups in the lower range and larger groups in the higher range, resulting in the mel-spectrogram. Finally, the amplitudes (y-axis) of the Mel-spectrogram are logarithmically scaled to obtain the log-mel-spectrogram.

The models in this thesis take log-mel-spectrograms with a 25 ms window, 10 ms stride and 40 mel-filters as input and are trained to predict a phone, the smallest identifiable unit of speech, for each time frame. This task forms a sub-problem of automatic speech recognition, where the goal is to transcribe spoken language into text.

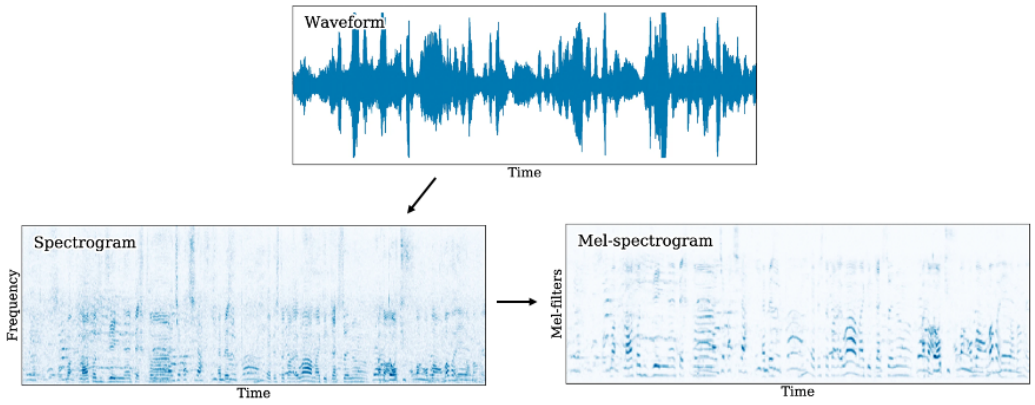


Figure 2.1: A waveform is converted into a spectrogram, displaying amplitude across frequencies over time. The spectrogram is then transformed into a mel-spectrogram by averaging frequencies according to the mel-scale. Amplitude is represented by pixel darkness in both the spectrogram and mel-spectrogram. The arrows indicate the sequence of transformations from the original waveform to the final mel-spectrogram representation. Diagram taken from Martin (2023), orginally adapted from de Benito et al. (2019).

Chapter 3

Extensions to MAC

This chapter discusses various extensions to the Method of Auxiliary Coordinates. I will begin by providing a short overview of the MAC update equations for the W- and, importantly, the Z-Step. As the Z-Step is MAC's main innovation over other training methods such as those mentioned in the background (Section 2.4), it will also be the main focus of the following two sections. In particular, I will discuss how the Z update rule is coupled across layers and what this means for distributing the training across different machines. Following this, I'll discuss two techniques of how to partially and fully decouple the Z-Step update equations to enable better distributivity. I will follow this up by proposing a series of smaller extensions to MAC, which allow the method to be applied to more modern architectures and tasks other than autoencoding. These extensions are proposed in this chapter and will be evaluated in the Experiments chapter (Chapter 4). This chapter will conclude by introducing a set of equations to describe the theoretical speedup of distributed, decoupled MAC when compared to a typical training procedure using SGD or Adam.

3.1 On the W and Z-Steps

In continuation of the previous chapter's description of MAC, this section provides a more detailed look at the updated equations for the W- and Z-Steps, and in particular, motivates the need for decoupling the Z-Step.

3.1.1 W-Step

We recall from the previous chapter that the W-Step involves updating each layer's parameters $\boldsymbol{\theta}_\ell$ separately while keeping the other layers and the auxiliary coordinates fixed. This is described generally by the update rule in Equation 2.11. Assuming that we train a model on a dataset $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{N}$ using a mean squared error (MSE) loss, the

W-Step update rule for layer ℓ is given by

$$\begin{aligned}\boldsymbol{\theta}_\ell &\leftarrow \operatorname*{argmin}_{\boldsymbol{\theta}_\ell} E_Q(\boldsymbol{\theta}, Z; \mu) \\ &= \operatorname*{argmin}_{\boldsymbol{\theta}_\ell} \frac{1}{2N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L) \right\|^2 \\ &\quad + \frac{\mu}{2N} \sum_{\ell'=0}^{L-1} \sum_{i=1}^{N} \left\| \mathbf{z}_{\ell'+1}^{(i)} - f_\ell(\mathbf{z}_{\ell'}^{(i)}; \boldsymbol{\theta}_{\ell'}) \right\|^2.\end{aligned} \tag{3.1}$$

To simplify the equation, we can drop all terms that do not depend on $\boldsymbol{\theta}_\ell$:

$$\boldsymbol{\theta}_\ell \leftarrow \begin{cases} \operatorname{argmin}_{\boldsymbol{\theta}_\ell} \frac{1}{2N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L) \right\|^2 & \text{if } \ell = L, \\ \operatorname{argmin}_{\boldsymbol{\theta}_\ell} \frac{\mu}{2N} \sum_{i=1}^{N} \left\| \mathbf{z}_{\ell+1}^{(i)} - f_\ell(\mathbf{z}_\ell^{(i)}; \boldsymbol{\theta}_\ell) \right\|^2 & \text{if } 0 \le \ell < L. \end{cases} \tag{3.2}$$

for $\mathbf{z}_0^{(i)} = \mathbf{x}^{(i)}$.

Since the constant factors $\frac{1}{2}$ and $\frac{\mu}{2}$ do not affect the minimisation, we can drop them as well, leaving us with the simplified W-Step update rule:

$$\boldsymbol{\theta}_\ell \leftarrow \begin{cases} \operatorname{argmin}_{\boldsymbol{\theta}_\ell} \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L) \right\|^2 & \text{if } \ell = L, \\ \operatorname{argmin}_{\boldsymbol{\theta}_\ell} \frac{1}{N} \sum_{i=1}^{N} \left\| \mathbf{z}_{\ell+1}^{(i)} - f_\ell(\mathbf{z}_\ell^{(i)}; \boldsymbol{\theta}_\ell) \right\|^2 & \text{if } 0 \le \ell < L. \end{cases} \tag{3.3}$$

Thus, given that all $\mathbf{z}$s are known and fixed, We have reduced the W-Step update rule for layer ℓ from its general form in terms of the quadratic penalty (Eq. 2.10) to training the layer by minimising the MSE loss between its output and the next layer's input across all datapoints. This is a standard minimisation problem that can be approached using various gradient-based optimisation algorithms. For example, Carreira-Perpiñán and Wang use the second-order Gauss-Newton method in their experiments, although I will use the first-order Adam optimiser in my experiments (see Section 4.1).

3.1.2 A coupled Z-Step

Akin to the W-Step, we can expand the Z-Step's update rule from the general form in Equation 2.12 to the specific form for each datapoint's auxiliary coordinates $\mathbf{z}^{(i)} = \{\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)}\}$:

$$\begin{aligned}\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)} &\leftarrow \operatorname*{argmin}_{\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)}} E_Q(\boldsymbol{\theta}, Z; \mu) \\ &= \operatorname*{argmin}_{\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)}} \frac{1}{2N} \sum_{j=1}^{N} \left\| \mathbf{y}^{(j)} - f_L(\mathbf{z}_L^{(j)}; \boldsymbol{\theta}_L) \right\|^2 \\ &\quad + \frac{\mu}{2N} \sum_{\ell=1}^{L} \sum_{j=1}^{N} \left\| \mathbf{z}_\ell^{(j)} - f_{\ell-1}(\mathbf{z}_{\ell-1}^{(j)}; \boldsymbol{\theta}_{\ell-1}) \right\|^2.\end{aligned} \tag{3.4}$$

Note the change in the third row, where the sum over ℓ starts at 1 instead of 0, to reflect the fact that $\mathbf{z}_0^{(i)} = \mathbf{x}^{(i)}$ is fixed.

Simplifying this equation by dropping all terms that do not depend on $\mathbf{z}^{(i)}$ yields

$$\begin{aligned}\mathbf{z}_1^{(i)}, \dots, \mathbf{z}_L^{(i)} \leftarrow \underset{\mathbf{z}_1^{(i)}, \dots, \mathbf{z}_L^{(i)}}{\operatorname{argmin}} \; & \frac{1}{2} \left\| \mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L) \right\|^2 \\ & + \frac{\mu}{2} \sum_{\ell=1}^{L} \left\| \mathbf{z}_\ell^{(i)} - f_{\ell-1}(\mathbf{z}_{\ell-1}^{(i)}; \boldsymbol{\theta}_{\ell-1}) \right\|^2 .\end{aligned} \tag{3.5}$$

While this equation is somewhat similar to the expanded W-Step update rule in (3.1), the W-Step's update rule only optimises over a single variable, while the Z-Step must optimise over all $\mathbf{z}_\ell^{(i)}$ jointly. This is necessary, as the $\mathbf{z}_\ell^{(i)}$s are coupled across layers, meaning that minimising one term by changing the value of $\mathbf{z}_\ell^{(i)}$ will affect the value of another term which also depends on the same $\mathbf{z}_\ell^{(i)}$. Hence, to optimise $\mathbf{z}_L^{(i)}$, we must also optimise $\mathbf{z}_{L-1}^{(i)}$. However, to optimise $\mathbf{z}_{L-1}^{(i)}$, we must also optimise $\mathbf{z}_{L-2}^{(i)}$, etc., until we reach $\mathbf{z}_1^{(i)}$. This not only complicates the optimisation problem but also complicates the distributivity of the Z-Step.

3.1.3 Issues with a coupled Z-Step

Given the coupling requirement, the only ways of parallelising the Z-Step are by either distributing the $\mathbf{z}$s by layer, synchronising the coordinates between each Z-Step iteration, or by distributing the Z-Step by datapoint, such that each machine is responsible for all auxiliary coordinates of a subset of the datapoints. The former approach has a huge communication overhead, while the latter approach is inconsistent with the W-Step, which is distributed by layer. With this inconsistency, we would have to communicate the updated θs to all machines after each W-Step iteration, and then communicate the updated $\mathbf{z}$s to all machines after each Z-Step iteration. This communication overhead would result in a significant bottleneck in MAC's distributivity.

Although Carreira-Perpiñán and Wang will have likely been aware of the complication of having the Z-Step be coupled across layers, they circumvent this by only 'running the Z-step with 1 Gauss-Newton iteration' (Carreira-Perpiñán & Wang, 2012). In this special case, the update of the coupled $\mathbf{z}^{(i)}$s is equivalent to the update of the partially coupled variant (see Section 3.2.1) which can be distributed more easily. However, as we increase the number of iterations, the coupling across layers becomes a more significant issue. To avoid using the expensive Gauss-Newton method (see the following subsection) in favour of a first-order optimiser, we must decouple the Z-Step. This will be the focus of the subsequent sections.

3.1.4 Cost of the Gauss-Newton method

As mentioned previously, while the Gauss-Newton method used by Carreira-Perpiñán and Wang requires very few iterations (the authors use 1 iteration), each iteration is very expensive to compute.

Given a concatenated auxiliary coordinate vector $\mathbf{z} = \begin{bmatrix} \mathbf{z}_1^{(i)} & \cdots & \mathbf{z}_L^{(i)} \end{bmatrix}^\top \in \mathbb{R}^{n \times 1}$ for datapoint i, where L is the number of layers, the Gauss-Newton method's update rule is given by

$$\mathbf{z} \leftarrow \mathbf{z} - (J^\top J)^{-1} J r, \tag{3.6}$$

where $r \in \mathbb{R}^{1 \times 1}$ is the sum of residuals in Equation 3.5 and $J \in \mathbb{R}^{n \times 1}$ is the Jacobian of r with respect to $\mathbf{z}$, s.t. $J_j = \frac{\partial r}{\partial \mathbf{z}_j}$ for element $j \in \{1, \ldots, n\}$.

Assuming that the residuals and Jacobian are given, and using basic schoolbook matrix multiplication (see Appendix A) to compute the update, we get the following complexities:

- Multiplying $J^\top J$ requires $O(n^2)$ operations.
- Inverting $J^\top J$ has the same time complexity as J^2, thus requiring $O(n^3)$ operations (Strassen, 1969).
- Multiplying $((J^\top J)^{-1})J$ requires $O(n^2)$ operations.
- Updating $\mathbf{z}$ is an element-wise subtraction, requiring $O(n)$ operations.

Hence, we have a total computational complexity of $O(n^3)$ for each Gauss-Newton iteration, assuming that the residuals and Jacobian are already known.

Comparing this to the cost of a first-order optimiser such as SGD:

$$\mathbf{z} \leftarrow \mathbf{z} - \eta J, \tag{3.7}$$

where η is the learning rate, we see that the cost of a single iteration is $O(n)$ and thereby significantly lower than the Gauss-Newton method. This is still true for cheaper matrix multiplication algorithms than the schoolbook method used above for simplicity (Williams et al., 2023).

Given the high cost of the Gauss-Newton method, it is clear that we should aim to decouple the Z-Step and use a first-order optimiser instead.

3.2 Decoupling the Z-Step

As discussed in the previous section, the coupling of the auxiliary coordinates $\mathbf{z}_1^{(i)}, \ldots, \mathbf{z}_L^{(i)}$ across layers in the Z-Step update rule (Equation 3.5) poses challenges for distributing the computation. The two main approaches for parallelising the coupled Z-Step – distributing by layer or by datapoint – both have significant drawbacks in terms of communication overhead and inconsistency with the layer-wise distribution of the W-Step.

To enable more efficient distributed optimization, it is desirable to decouple the Z-Step update rule such that the auxiliary coordinates for each layer can be updated

independently. This would allow the Z-Step to be parallelized across layers, consistent with the distribution scheme of the W-Step, thereby reducing the communication overhead.

In the following subsections, I will present two techniques for decoupling the Z-Step: partial decoupling and full decoupling. The partially decoupled variant approximates the coupled variant described in Section 3.1.2 by assuming independence between the auxiliary coordinates of different layers and optimising each $\mathbf{z}_\ell^{(i)}$ separately while keeping the other coordinates fixed. This variant will also be referred to as '2-term decoupling', as the optimisation equation can be reduced to two terms. I further show that performing one iteration of the coupled Z-Step update rule is equivalent to one iteration of the partially decoupled rule. The fully decoupled variant, called the '1-term' update rule, further simplifies the update equations to reduce the amount of data transferred between machines after each W and Z-Step.

By decoupling the Z-Step, I aim to develop a more computationally efficient and easily distributable version of the Method of Auxiliary Coordinates. The benefits and potential drawbacks of these decoupled variants will be discussed in detail, providing a foundation for the experimental evaluation in Chapter 4.

3.2.1 Partially decoupled training

Arguably the simplest way to decouple the Z-Step is to simply assume that the auxiliary coordinates $\mathbf{z}_1^{(i)}, \dots, \mathbf{z}_L^{(i)}$ are independent across layers. Under this assumption, we can minimise the Z-Step update rule from Equation 3.5 separately for each $\mathbf{z}_\ell^{(i)}$ while keeping the other coordinates fixed:

$$\mathbf{z}_\ell^{(i)} \leftarrow \underset{\mathbf{z}_\ell^{(i)}}{\operatorname{argmin}} \; \frac{1}{2} \left\| \mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L) \right\|^2 + \frac{\mu}{2} \sum_{\ell'=1}^{L} \left\| \mathbf{z}_{\ell'}^{(i)} - f_{\ell'-1}(\mathbf{z}_{\ell'-1}^{(i)}; \boldsymbol{\theta}_{\ell'-1}) \right\|^2. \quad (3.8)$$

By removing the constant terms that do not depend on $\mathbf{z}_\ell^{(i)}$ and simplifying, we obtain the partially decoupled Z-Step update rule:

$$\mathbf{z}_\ell^{(i)} \leftarrow \begin{cases} \operatorname{argmin}_{\mathbf{z}_\ell^{(i)}} \; \frac{1}{2} \left\| \mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L) \right\|^2 + \frac{\mu}{2} \left\| \mathbf{z}_L^{(i)} - f_{L-1}(\mathbf{z}_{L-1}^{(i)}; \boldsymbol{\theta}_{L-1}) \right\|^2 & \text{if } \ell = L, \\ \operatorname{argmin}_{\mathbf{z}_\ell^{(i)}} \; \frac{\mu}{2} \left\| \mathbf{z}_{\ell+1}^{(i)} - f_\ell(\mathbf{z}_\ell^{(i)}; \boldsymbol{\theta}_\ell) \right\|^2 + \frac{\mu}{2} \left\| \mathbf{z}_\ell^{(i)} - f_{\ell-1}(\mathbf{z}_{\ell-1}^{(i)}; \boldsymbol{\theta}_{\ell-1}) \right\|^2 & \text{if } 0 < \ell < L. \end{cases} \quad (3.9)$$

where $\mathbf{z}_0^{(i)} = \mathbf{x}^{(i)}$.

Interestingly, performing one iteration of the coupled Z-Step update rule (Equation 3.5) is equivalent to performing one iteration of the partially decoupled update rule. This can be shown by comparing the partial derivative of the coupled update rule with the partial derivative of the partially decoupled update rule for each layer ℓ in both equations.

Coupled (Eq. 3.5):

$$\begin{aligned}\frac{\partial E_Q}{\partial \mathbf{z}_\ell^{(i)}} &= \frac{\partial}{\partial \mathbf{z}_\ell^{(i)}}\left(\frac{1}{2}\left\|\mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)};\,\boldsymbol{\theta}_L)\right\|^2 + \frac{\mu}{2}\sum_{\ell'=1}^{L}\left\|\mathbf{z}_{\ell'}^{(i)} - f_{\ell'-1}(\mathbf{z}_{\ell'-1}^{(i)};\,\boldsymbol{\theta}_{\ell'-1})\right\|^2\right) \\ &= \frac{\partial}{\partial \mathbf{z}_\ell^{(i)}}\left(\frac{\mu}{2}\left\|\mathbf{z}_{\ell+1}^{(i)} - f_\ell(\mathbf{z}_\ell^{(i)};\,\boldsymbol{\theta}_\ell)\right\|^2 + \frac{\mu}{2}\left\|\mathbf{z}_\ell^{(i)} - f_{\ell-1}(\mathbf{z}_{\ell-1}^{(i)};\,\boldsymbol{\theta}_{\ell-1})\right\|^2\right)\end{aligned} \tag{3.10}$$

Partially decoupled (Eq. 3.9):

$$\frac{\partial}{\partial \mathbf{z}_\ell^{(i)}}\left(\frac{\mu}{2}\left\|\mathbf{z}_{\ell+1}^{(i)} - f_\ell(\mathbf{z}_\ell^{(i)};\,\boldsymbol{\theta}_\ell)\right\|^2 + \frac{\mu}{2}\left\|\mathbf{z}_\ell^{(i)} - f_{\ell-1}(\mathbf{z}_{\ell-1}^{(i)};\,\boldsymbol{\theta}_{\ell-1})\right\|^2\right) \tag{3.11}$$

As these equations are sufficient to show the equivalence of the two partial derivatives, I will not provide the full derivations here. However, the interested reader is encouraged to view them in Appendix B.

Given that the partial derivatives of the coupled and partially decoupled update rules are shown to be equal, their respective Jacobians are also equal. Hence, assuming that all variables $\mathbf{z}_\ell$, $\mathbf{z}_{\ell-1}$, $\boldsymbol{\theta}_\ell$ and $\boldsymbol{\theta}_{\ell-1}$ are known and the same for both Z-Step variants (which they are), any update using the Gauss-Newton, SGD, Adam or any other optimiser that is solely dependent on the Jacobian will be equivalent for both the coupled and partially decoupled Z-Step update rules. This equivalence is likely what Carreira-Perpiñán and Wang rely on to distribute the Z-Step in their experiments.

While the coupled Z-Step updates all $\mathbf{z}$s in each iteration and uses the updated coordinates for the subsequent iteration, the 2-term Z-Step updates each $\mathbf{z}_\ell$ separately from $\mathbf{z}_{\ell-1}$, thereby using the old value of $\mathbf{z}_{\ell-1}$ for each iteration of $\mathbf{z}_\ell$. Thus, as the number of iterations increases, the 2-term rule becomes an approximation for the coupled update rule and the equivalence between the two update rules becomes less accurate. However, as the number of iterations is typically small, the approximation is likely to be sufficient for most practical purposes and its inaccuracies are outweighed by the benefits of distributing the Z-Step by layer.

Now that we can distribute both, the W- and 2-term Z-Step by layer, the computation flow across different machines, where each machine is responsible for a single layer, is illustrated in Figure 3.1. The procedure is as follows:

1. Machine 0 initialises the model parameters and the auxiliary coordinates by computing a forward pass through all layers for each datapoint $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$.
2. The relevant initial parameters and coordinates are sent to each machine. Machine ℓ (for $\ell < L$) receives $\boldsymbol{\theta}_\ell$, $\mathbf{z}_\ell$ and $\mathbf{z}_{\ell+1}$. Machine L receives $\boldsymbol{\theta}_L$, $\mathbf{z}_L$ and $\mathbf{y}$.
3. Each machine trains its respective layer ℓ using the W-Step update rule from Equation 3.3 for a predefined number of iterations.
4. After the W-Step, each machine communicates the updated $\boldsymbol{\theta}_\ell$ as well as its coordinates $\mathbf{z}_\ell$ to the next machine (e.g. machine ℓ sends $(\boldsymbol{\theta}_\ell, \mathbf{z}_\ell)$ to machine $\ell+1$). As machine L has no successor, it does not send any data.

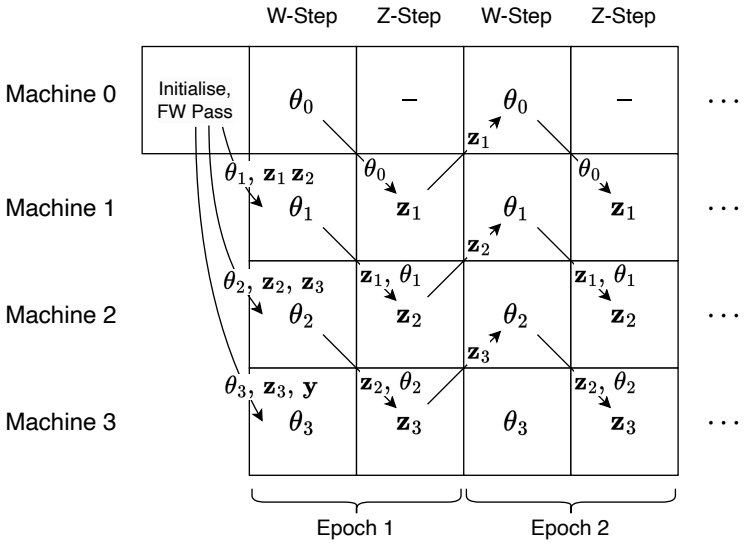


Figure 3.1: Computation and communication flow of partially decoupled MAC. Each square indicates a machine performing a computation. The label at the centre of the square describes which values are being updated by the computation. The arrows indicate the transfer of data from one machine to another. Each arrow's label indicates what data is being sent. Data transfers occur after each step. Each machine is assumed to have persistent storage between each computation. For example, machine 0 first initialises the model parameters and auxiliary coordinates. Machine 0 then sends the relevant parameters and coordinates to each other machine and proceeds to optimise θ_0 during the W-Step. Next, θ_0 is sent to machine 1. While the other machines compute the Z-Step, machine 0 remains idle. Etc.

5. Each machine optimises its respective auxiliary coordinates $\mathbf{z}_\ell$ using the 2-term Z-Step update rule from Equation 3.9 for a predefined number of iterations. Since $\mathbf{z}_0 = \mathbf{x}$ is fixed, machine 0 does not perform a Z-Step. Note that because the machines only communicate before the Z-Step and not during it, machine ℓ can only use the old values of $\mathbf{z}_{\ell-1}$ and $\mathbf{z}_{\ell+1}$ from before the Z-Step.
6. After the Z-Step, each machine communicates the updated auxiliary coordinates $\mathbf{z}_\ell$ to the preceding machine (e.g. machine ℓ sends $\mathbf{z}_\ell$ to machine $\ell - 1$). As machine 0 has not performed a Z-Step, it does not send any data.
7. The process is repeated from step 3 for a predefined number of epochs.

This communication diagram shows that the 2-term Z-Step can be distributed by layer in a similar manner to the W-Step. The communication overhead is limited to the transfer of the updated parameters and coordinates between machines after each W and Z-Step. This is a significant improvement over the coupled Z-Step, which would require synchronising all auxiliary coordinates between machines after each Z-Step iteration, which can quickly become very expensive, especially with large datasets. The 2-term Z-Step is therefore more easily distributable and can be parallelised across layers, making it a more efficient and scalable training method for deep neural networks.

While the partially decoupled variant requires less communication than the coupled

Z-Step, there may be room for further reduction in communication overhead. This leads us to the fully decoupled variant, which will be discussed in the next subsection.

3.2.2 Fully decoupled training

While the partially decoupled 2-term Z-Step update rule reduces the communication overhead compared to the coupled variant, it may be possible to further decrease the amount of data transferred between machines. This leads us to propose a fully decoupled 1-term Z-Step update rule.

The key idea is to further simplify the 2-term update rule in Equation 3.9 by removing one of the two terms for each layer ℓ. Since our intent is to minimise the MSE loss $\|\mathbf{y} - f_L(\mathbf{z}_L; \boldsymbol{\theta}_L)\|^2$ under the constraint that $\mathbf{z}_{\ell+1} = f_\ell(\mathbf{z}_\ell; \boldsymbol{\theta}_\ell)$, we must keep $\mathbf{y}$ as part of our update rule. If we then disregard the second term from the 2-term update rule for each $\mathbf{z}_\ell$, we get

$$\mathbf{z}_\ell^{(i)} \leftarrow \begin{cases} \operatorname{argmin}_{\mathbf{z}_\ell^{(i)}} \ \frac{1}{2} \left\| \mathbf{y}^{(i)} - f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L) \right\|^2 & \text{if } \ell = L, \\ \operatorname{argmin}_{\mathbf{z}_\ell^{(i)}} \ \frac{\mu}{2} \left\| \mathbf{z}_{\ell+1}^{(i)} - f_\ell(\mathbf{z}_\ell^{(i)}; \boldsymbol{\theta}_\ell) \right\|^2 & \text{if } 0 < \ell < L. \end{cases} \tag{3.12}$$

as the fully decoupled Z-Step update rule. Each $\mathbf{z}_\ell$ is now optimised to a value that, when it is input into the layer f_ℓ, the output is as close to the target as possible. $\mathbf{z}_L$ is optimised such that the output $f_L(\mathbf{z}_L; \boldsymbol{\theta}_L)$ is as close to the ground truth as possible. Next, $\mathbf{z}_{L-1}$ is optimised such that the output of $f_{L-1}(\mathbf{z}_{L-1}; \boldsymbol{\theta}_{L-1})$ is as close to $\mathbf{z}_L$ as possible, etc. until $\mathbf{z}_1$.

By further decoupling the update rule and reducing it to a single term, the optimisation problem for each $\mathbf{z}_\ell$ is no longer dependent on $\mathbf{z}_{\ell-1}$ or $\boldsymbol{\theta}_{\ell-1}$. Hence, the machines don't need to request $\mathbf{z}_{\ell-1}$ or $\boldsymbol{\theta}_{\ell-1}$ from the previous machine before the Z-Step, thereby more than halving the communication required between machines. Figure 3.2 illustrates the updated computations across machines for the fully decoupled Z-Step. Note that the auxiliary coordinates are only communicated once from one machine to another after each epoch. The model weights are never communicated.

Drawbacks While this 1-term update rule seems very effective in terms of reducing communication overhead, it also has some potential drawbacks that need to be considered:

1. Since we are only optimising each $\mathbf{z}_\ell$ to minimise the MSE loss between the output of layer f_ℓ and either the next layer's input or the target, we may end up 'over-optimising' a coordinate. For instance, if we minimise $\|\mathbf{y} - f_L(\mathbf{z}_L; \boldsymbol{\theta}_L)\|^2$ for $\mathbf{z}_L$, we could end up choosing values for $\mathbf{z}_L$ that the L-th layer can use to almost perfectly predict $\mathbf{y}$, but now the preceding layers need to be trained to predict $\mathbf{z}_L$. If we then optimise $\|\mathbf{z}_L - f_{L-1}(\mathbf{z}_{L-1}; \boldsymbol{\theta}_{L-1})\|^2$ for $\mathbf{z}_{L-1}$, we will end up in a similar situation where layers $L-1$ and L can together almost perfectly transform $\mathbf{z}_{L-1}$ into $\mathbf{y}$, but now the preceding layers need to be trained to predict $\mathbf{z}_{L-1}$, etc. In the end, we have effectively reduced the full network to only the bottom layer.

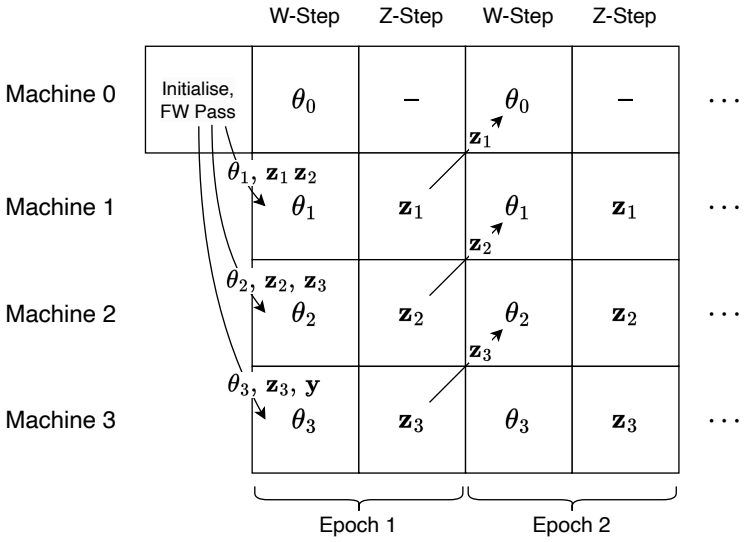


Figure 3.2: Computation and communication flow of fully decoupled MAC. The distribution of computations across machines through time is identical to partially decoupled MAC in Figure 3.1. The data transfers between machines are reduced in fully decoupled MAC when compared to Figure 3.1.

To avoid this, we must carefully choose the Z-Step learning rate and number of iterations to only slightly update $\mathbf{z}_\ell$ during each epoch.

2. Although the 1-term rule requires less communication, the time spent on computation likely far outweighs the time spent on communication, which, despite the reduced number of terms, is not significantly faster than the 2-term rule. This is because in the 2-term case, the output of the previous layer $f_{\ell-1}(\mathbf{z}_{\ell-1}; \boldsymbol{\theta}_{\ell-1})$ can be precomputed and reused for each iteration of the Z-Step.

In summary, the proposed fully decoupled 1-term Z-Step update rule reduces the communication overhead of the partially decoupled 2-term rule even further by simplifying the update equations. However, this may also cause it to be more sensitive to the hyperparameter choice. Further, as it is only an approximation of an approximation of the original coupled Z-Step, it may even lead to reduced performance of the trained model. Additionally, the 1-term approach provides little to no computational speedup over the 2-term rule. The trade-offs between the 1-term and 2-term variants will be investigated experimentally in Chapter 4.

3.3 Further extensions

In addition to the decoupled Z-Step variants proposed in the previous sections, several other extensions can be made to the Method of Auxiliary Coordinates to improve its applicability outside of the image autoencoding use case presented by Carreira-Perpiñán and Wang (2014). In particular, this section will cover how we can use MAC with custom non-MSE loss functions, modern sequential network architectures, and dropout regularisation.

3.3.1 Custom loss functions

The update rules introduced by Carreira-Perpiñán and Wang (2014), which were extended and decoupled in this chapter assume a mean squared error (MSE) loss function. However, just like other training techniques, the Method of Auxiliary Coordinates can be extended to work with any differentiable loss function $\mathcal{L}(\mathbf{y}, f(\mathbf{x};\boldsymbol{\theta}))$. This is useful for tasks where the MSE loss is not appropriate, such as classification tasks, where the cross-entropy loss is commonly used, speech recognition using a Connectionist Temporal Classification loss, or self-supervised learning tasks using autoregressive or contrastive losses (Graves et al., 2006; van den Oord et al., 2018; Yang et al., 2022). The W-Step update rule for the output layer (Eq. 3.3) simply becomes:

$$\boldsymbol{\theta}_L \leftarrow \operatorname*{argmin}_{\boldsymbol{\theta}_L} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\mathbf{y}^{(i)}, f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L)) \tag{3.13}$$

Similarly, the output layer's Z-Step update rules (Eq. 3.5, 3.9 and 3.12) become

$$\text{Coupled:} \quad \mathbf{z}_L^{(i)} \leftarrow \operatorname*{argmin}_{\mathbf{z}_1^{(i)},\ldots,\mathbf{z}_L^{(i)}} \frac{1}{2} \mathcal{L}(\mathbf{y}^{(i)}, f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L)) + \frac{\mu}{2} \sum_{\ell=1}^{L} \left\| \mathbf{z}_\ell^{(i)} - f_{\ell-1}(\mathbf{z}_{\ell-1}^{(i)};\, \boldsymbol{\theta}_{\ell-1}) \right\|^2 \tag{3.14}$$

$$\text{Partially decoupled:} \quad \mathbf{z}_L^{(i)} \leftarrow \operatorname*{argmin}_{\mathbf{z}_L^{(i)}} \frac{1}{2} \mathcal{L}(\mathbf{y}^{(i)}, f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L)) + \frac{\mu}{2} \left\| \mathbf{z}_L^{(i)} - f_{L-1}(\mathbf{z}_{L-1}^{(i)};\, \boldsymbol{\theta}_{L-1}) \right\|^2 \tag{3.15}$$

$$\text{Fully decoupled:} \quad \mathbf{z}_L^{(i)} \leftarrow \operatorname*{argmin}_{\mathbf{z}_L^{(i)}} \frac{1}{2} \mathcal{L}(\mathbf{y}^{(i)}, f_L(\mathbf{z}_L^{(i)}; \boldsymbol{\theta}_L)) \tag{3.16}$$

The update rules for the hidden layers remain unchanged. This extension allows MAC to be applied to a wider range of tasks.

3.3.2 Arbitrary network architectures

The architectures used by Carreira-Perpiñán and Wang in their 2014 publication on MAC and their 2019 follow-up are a sigmoidal autoencoder, as well as autoencoders based on Radial basis functions (RBF) and a linear Support vector machines (SVM). While they were sufficient for the authors to demonstrate the effectiveness of MAC, they are not commonly used for modern applications or even representative of today's deep learning architectures. However, MAC can handle more complex, modern architectures as well.

Since all the equations have thus far used the generic notation $f_\ell(\cdot;\, \boldsymbol{\theta}_\ell)$ to denote the ℓ-th layer, accepting an input of a pre-specified dimension and returning an output of the same or different dimension. Thus, instead of following Carreira-Perpiñán and Wang, we can define the function to be any self-contained network layer, such as a feedforward layer (linearity + non-linearity), a Transformer block (Vaswani et al., 2017), an LSTM

layer, or any other current or future function that accepts an input and provides an output. This allows us to incorporate architectural features such as skip connections or layer normalisation within a layer (Ba et al., 2016; He et al., 2016).

3.3.3 Dropout

Although we can theoretically use any layer type as the f_ℓ function, one restriction is that the chosen layer should be deterministic (i.e. given the same input with the same parameters, always produce the same output). While stochasticity can for instance be useful in the case of dropout (Hinton et al., 2012), it would either average out over multiple iterations of the Z-Step or, if the Z-Step is only run for very few iterations (as I have shown to be desirable in Chapter 4), the stochasticity would throw off the gradients in the Z update step.

Therefore, to be able to take advantage of the benefits of dropout while still using MAC, we can simply apply dropout only during the W-Step. Thus, the forward pass through the network during the Z-Step is deterministic, while the W-Step can use dropout to regularise the model.

3.4 Theoretical speedup

In the previous sections, I proposed several extensions to the Method of Auxiliary Coordinates to enable more efficient distributed training of deep neural networks. Specifically, I introduced partially and fully decoupled variants of the Z-Step update rule to reduce the communication overhead between machines and allow the Z-Step to be parallelized across layers, consistent with the distribution scheme of the W-Step.

These extensions aim to improve the computational efficiency and scalability of MAC. Hence, in this section, I will introduce a set of equations to describe the theoretical speedup of distributed, decoupled MAC when compared to batched stochastic gradient descent or Adam. In particular, I will highlight under which circumstances a distributed MAC can be expected to provide a speedup.

3.4.1 Training time of SGD

Let us begin by deconstructing the time T_{SGD} that SGD (or by extension Adam) take to achieve the desired loss on the given training set:

$$\begin{aligned} T_{\text{SGD}} &= E_{\text{SGD}} \cdot t_{\text{SGD}} \\ &= E_{\text{SGD}} \cdot \left(\frac{t_{\text{F}} + t_{\text{B}} + t_{\text{other}}}{B_{\text{SGD}}} \right) \\ &\approx E_{\text{SGD}} \cdot \left(\frac{t_{\text{F}} + t_{\text{B}}}{B_{\text{SGD}}} \right) \end{aligned} \tag{3.17}$$

where E_{SGD} is the number of epochs required to reach the desired loss, t_{SGD} is the time taken per epoch, and t_{F} and t_{B} are the times taken for a single forward pass and

backward pass through the entire network for all datapoints with batch size 1. t_{other} is a placeholder for the time taken by other computations (e.g. the paramter update), although this tends to take significantly less time than the forward and backward passes, I will disregard t_{other} for this analysis. To account for batching, I introduce B_{SGD} as the batch size.

The times taken for a single forward pass t_{F} and backward pass t_{B} are given by

$$t_{\text{F}} = \sum_{\ell=0}^{L} t_{\text{F},\ell}, \qquad t_{\text{B}} = \sum_{\ell=0}^{L} t_{\text{B},\ell}, \tag{3.18}$$

where $t_{\text{F},\ell}$ and $t_{\text{B},\ell}$ are the times taken for the forward and backward passes through the ℓ-th layer, respectively.

3.4.2 Training time of MAC

The training time T_{MAC} of MAC (when run synchronously on one machine) can first be similarly deconstructed into the time taken for one epoch t_{MAC} and the number of epochs E_{MAC} required to reach the desired loss, and then further into the time taken for the W and the Z-Step:

$$\begin{aligned} T_{\text{MAC}} &= E_{\text{MAC}} \cdot t_{\text{MAC}} \\ &= E_{\text{MAC}} \cdot (N_{\text{W}} \cdot t_{\text{W}} + N_{\text{Z}} \cdot t_{\text{W}} + t_{\text{other}}) \\ &\approx E_{\text{MAC}} \cdot (N_{\text{W}} \cdot t_{\text{W}} + N_{\text{Z}} \cdot t_{\text{W}}) \\ &= E_{\text{MAC}} \cdot \left(\sum_{\ell=0}^{L} (N_{\text{W},\ell} \cdot t_{\text{W},\ell}) + \sum_{\ell=0}^{L} (N_{\text{Z},\ell} \cdot t_{\text{Z},\ell}) \right) \end{aligned} \tag{3.19}$$

where N_{W} and N_{Z} are the number of iterations of the W and Z-Step, and t_{W} and t_{Z} are the times taken for one iteration of the respective steps across the full network and dataset with batch size 1. $N_{\text{W},\ell}$, $N_{\text{Z},\ell}$, $t_{\text{W},\ell}$ and $t_{\text{Z},\ell}$ are the iterations and times taken for each step on an individual layer ℓ. As with SGD, I disregard t_{other} for this analysis.

If we now take advantage of the distributability of MAC and assign the training of each layer ℓ's parameters and respective auxiliary coordinates to a different machine as shown in Figures 3.2 and 3.1, we can reduce the time to

$$T_{\text{MAC}} = E_{\text{MAC}} \cdot \left(\max_{\ell} [N_{\text{W},\ell} \cdot t_{\text{W},\ell}] + \max_{\ell} [N_{\text{Z},\ell} \cdot t_{\text{Z},\ell}] \right). \tag{3.20}$$

Each W-Step iteration using SGD or Adam as the optimiser consists of a forward and backward pass to update the respective layer (Eq. 3.3). Each Z-Step also consists of a forward and backward pass to update the coordinates (Eq. 3.9, 3.12). Note that for the partially decoupled Z-Step, we can precompute one of the two forward passes as it stays constant for all iterations. Hence, we can further deconstruct the time as

$$T_{\text{MAC}} = E_{\text{MAC}} \cdot \left(\max_{\ell} \left[N_{\text{W},\ell} \cdot \frac{t_{\text{F},\ell} + t_{\text{B},\ell}}{B_{\text{W}}} \right] + \max_{\ell} \left[N_{\text{Z},\ell} \cdot \frac{t_{\text{F},\ell} + t_{\text{B},\ell}}{B_{\text{Z}}} \right] \right). \tag{3.21}$$

Assuming that the layers are distributed equally, such that the forward and backward passes take the same time for each layer on each machine: $\forall \ell, \ell' \in \{0, ..., L\}\ t_{\mathrm{F},\ell} \approx t_{\mathrm{F},\ell'}$ and $t_{\mathrm{B},\ell} \approx t_{\mathrm{B},\ell'}$, then we can further simplify the equation to

$$\begin{aligned} T_{\mathrm{MAC}} &= E_{\mathrm{MAC}} \cdot \left(\max_{\ell}[N_{\mathrm{W},\ell}] \frac{t_{\mathrm{F}} + t_{\mathrm{B}}}{B_{\mathrm{W}} \cdot (L+1)} + \max_{\ell}[N_{\mathrm{Z},\ell}] \frac{t_{\mathrm{F}} + t_{\mathrm{B}}}{B_{\mathrm{Z}} \cdot (L+1)} \right) \\ &= E_{\mathrm{MAC}} \cdot \left(\frac{t_{\mathrm{F}} + t_{\mathrm{B}}}{L+1} \right) \left(\frac{\max_{\ell}[N_{\mathrm{W},\ell}]}{B_{\mathrm{W}}} + \frac{\max_{\ell}[N_{\mathrm{Z},\ell}]}{B_{\mathrm{Z}}} \right). \end{aligned} \tag{3.22}$$

where L is the index of the last layer, and $L+1$ is the total number of model layers[1].

In practice, because each machine is only training a single layer, $B_{\mathrm{SGD}} \leq B_{\mathrm{W}} \leq B_{\mathrm{Z}}$. For a conservative estimation of T_{MAC}, let us assume $B_{\mathrm{SGD}} = B_{\mathrm{W}} = B_{\mathrm{Z}}$. Hence, by rearranging the equation and substituting $N_{\mathrm{W,\,max}} \triangleq \max_{\ell}[N_{\mathrm{W},\ell}]$ and $N_{\mathrm{Z,\,max}} \triangleq \max_{\ell}[N_{\mathrm{Z},\ell}]$ for readability, we get

$$\begin{aligned} T_{\mathrm{MAC}} &= E_{\mathrm{MAC}} \cdot \left(\frac{1}{L+1} \right) \left(\frac{t_{\mathrm{F}} + t_{\mathrm{B}}}{B_{\mathrm{SGD}}} \right) (N_{\mathrm{W,\,max}} + N_{\mathrm{Z,\,max}}) \\ &= E_{\mathrm{MAC}} \cdot t_{\mathrm{SGD}} \cdot \left(\frac{N_{\mathrm{W,\,max}} + N_{\mathrm{Z,\,max}}}{L+1} \right) \end{aligned} \tag{3.23}$$

as an approximation of MAC's distributed runtime. It is stated in terms of the number of MAC epochs E_{MAC}, the time for one epoch of SGD (or Adam) t_{SGD} through the full network and all datapoints, the total number of layers $L+1$ that we can split the model into, and the maximum number of W and Z iterations $N_{\mathrm{W,\,max}}$, $N_{\mathrm{Z,\,max}}$ that any machine performs.

3.4.3 Speedup of MAC over SGD

Using Equations 3.17 and 3.23 as relative estimates for the runtimes of batched SGD (and by extension Adam) and distributed MAC, we can approximate that MAC is expected to be faster than SGD or Adam when

$$\frac{E_{\mathrm{MAC}} \cdot (N_{\mathrm{W,\,max}} + N_{\mathrm{Z,\,max}})}{L+1} < E_{\mathrm{SGD}} \tag{3.24}$$

with a speedup of

$$\begin{aligned} \text{Speedup} &= \frac{T_{\mathrm{SGD}}}{T_{\mathrm{MAC}}} \\ &= \frac{E_{\mathrm{SGD}} \cdot t_{\mathrm{SGD}}}{E_{\mathrm{MAC}} \cdot t_{\mathrm{SGD}} \cdot \left(\frac{N_{\mathrm{W,\,max}} + N_{\mathrm{Z,\,max}}}{L+1} \right)} \\ &= \frac{E_{\mathrm{SGD}} \cdot (L+1)}{E_{\mathrm{MAC}} \cdot (N_{\mathrm{W,\,max}} + N_{\mathrm{Z,\,max}})}. \end{aligned} \tag{3.25}$$

Thus, MAC is expected to provide a speedup over SGD proportional to

[1] Although this definition of L is slightly awkward here, it is chosen to ensure consistency with the rest of the thesis, where it is more convenient.

- the number of layers ($L+1$) that the model is split into, and

and inversely proportional to

- the maximum number of W and Z-Step iterations ($N_{\text{W, max}}$ and $N_{\text{Z, max}}$) that any machine performs, and
- the number of MAC epochs E_{MAC} required to reach the desired loss with MAC.

It is important to note, however, that these equations make some simplifying assumptions, such as ignoring time spent on computations other than the forward and backward passes or the communication costs of MAC. It is also assumed that the distribution of layers is approximately equal across machines. Nonetheless, this can be a reasonable way to estimate when distributed MAC may provide a speedup over synchronous SGD.

This estimation also applies to training pipelines using model parallelism for SGD, as the runtime is expected to be proportional to the runtime of SGD on a single machine (see Section 2.4).

3.5 Summary

In this chapter, I proposed several extensions to the Method of Auxiliary Coordinates (MAC) to improve its computational efficiency, scalability, and applicability to modern deep learning architectures and tasks.

I began by providing a detailed analysis of the W-Step and Z-Step update rules, highlighting the coupling of the auxiliary coordinates across layers in the Z-Step. To address this issue, I introduced two techniques for decoupling the Z-Step: partial decoupling (2-term) and full decoupling (1-term). The partially decoupled variant approximates the coupled Z-Step by assuming independence between the auxiliary coordinates of different layers, while the fully decoupled variant further simplifies the update equations to reduce communication overhead. I demonstrated that one iteration of the coupled Z-Step is equivalent to one iteration of the partially decoupled rule, and discussed the potential drawbacks of the fully decoupled variant.

Furthermore, I extended MAC to support custom differentiable loss functions, enabling its application to a wider range of tasks beyond image autoencoding. I also showed how MAC can handle modern, complex network architectures by treating each layer as a self-contained function, allowing for the incorporation of architectural features such as skip connections and layer normalisation. Additionally, I discussed the use of dropout regularization in MAC, suggesting its application only during the W-Step to maintain deterministic forward passes during the Z-Step.

Finally, I introduced a set of equations to describe the theoretical speedup of distributed, decoupled MAC compared to batched SGD or Adam. The speedup is expected to be proportional to the number of layers the model is split into and inversely proportional to the maximum number of W and Z-Step iterations performed by any machine as well as the number of MAC epochs required to reach the desired loss, although the latter should come at no surprise.

These extensions aim to enhance the efficiency and applicability of MAC, making it a more viable alternative to traditional training methods for modern deep learning tasks. These extensions will be applied to MAC and experimentally evaluated in the next chapter, providing insights into their effectiveness and potential trade-offs. It will also provide empirical insight into some of MAC's peculiarities.

Chapter 4

Experiments

This chapter details the experiments that were run to evaluate the decoupled Method of Auxiliary Coordinates in various scenarios to provide insight into three fundamental questions, enumerated by their sections:

(4.2) Whether MAC converges, as demonstrated for the given models on the given task, and how it compares to the commonly used SGD and Adam optimisers (Kingma & Ba, 2015; Rumelhart et al., 1986).

(4.3) How we can choose good hyperparameters for MAC and how sensitive the training outcome is to these hyperparameters.

(4.4) How we might split up a model such that training it using MAC achieves the best final performance.

Finally, based on the results in this chapter, I will provide an intermediate conclusion on the practicality of MAC.

4.1 Experimental setup

Dataset Following on from last year's MInf 1 project, my experiments are designed around speech processing, specifically phone classification on the TIMIT (Garofolo, John S. et al., 1993) dataset. Although the dataset is commonly regarded as a small 'toy' dataset with its 6300 English utterances and approximately 5 hours of speech, compared to the up to 960 hours contained in the widely used LibriSpeech dataset (Panayotov et al., 2015), I am using TIMIT specifically for its smaller size. Its size allows for faster iteration over implementations, faster training, easier caching of the auxiliary coordinates and importantly, as I will elaborate on in Section 4.3, easier hyperparameter searches.

For the model input, the utterances' waveforms were converted to 40-dimensional log-mel features with a 25 ms window and 10 ms stride (see Section 2.5. The features are normalised using the mean and variance of the full training set.

Models The optimisation algorithms were used to train a 6-layer multi-layer perceptron (MLP; See Table 4.1) as well as a model with 4 stacked Transformer encoders and linear layers to scale the input and output to the appropriate dimensions (Vaswani et al., 2017; See Table 4.2).

Groupings As described in the previous chapter, the Method of Auxiliary Coordinates works by splitting up a model into layers, where each layer is represented functionally as $f_\ell(\cdot\,; \boldsymbol{\theta}_\ell)$. For simplicity, instead of splitting up the MLP and Transformer models into six layers each, I split them into two groups consisting of one or more layers each. The groupings are indicated in the tables by dashed lines. Grouping (1,5) has the input layer in the first group, and all four hidden layers as well as the output layer in the second group. Grouping (3,3) is an approximately even split in terms of parameter counts, where Group 1 contains the input layer and two hidden layers, and Group 2 contains the other two hidden layers as well as the output layer. Grouping (5,1) has all layers except for the output layer in the first group.

Computational resources The computational resources used for the experiments in this chapter are on the so-called 'MLP' cluster kindly provided by the University of Edinburgh School of Informatics. The compute nodes used each had an 'NVIDIA GeForce RTX 2080 Ti' GPU with 11 GiB of memory. The distributed training using MAC was executed on two compute nodes in parallel, one group per node. Because of a significant outage of the cluster up until a little over one week before submission, and corruption of the data stored on it, the experiments detailed in this chapter are not as extensive as initially intended. This has in particular affected the more time-consuming experiments on Transformers. However, the limited results should still suffice in demonstrating the intended results and patterns.

Table 4.1: The MLP's architecture trained for this chapter's experiments. The experiments on MAC split the model into two groups. Depending on the experiment, different groupings are chosen. Each group must have at least one layer. The points in the network where the model is split are marked by dashed lines and are labeled by the grouping which they represent. E.g. the first grouping (1,5) has only the one layer in the first group and five layers in the second group.

Grouping	Layer	Operation	Output Shape	# Parameters
	Input	Linear(40, 256)	(batch, frames, 256)	10,496
		ReLU	(batch, frames, 256)	–
(1,5)				
	Hidden	Linear(256, 256)	(batch, frames, 256)	65,792
		ReLU	(batch, frames, 256)	–
	Hidden	Linear(256, 256)	(batch, frames, 256)	65,792
		ReLU	(batch, frames, 256)	–
(3,3)				
	Hidden	Linear(256, 256)	(batch, frames, 256)	65,792
		ReLU	(batch, frames, 256)	–
	Hidden	Linear(256, 256)	(batch, frames, 256)	65,792
		ReLU	(batch, frames, 256)	–
(5,1)				
	Output	Linear(256, 62)	(batch, frames, 62)	15,934

Table 4.2: The architecture of the Transformer-based model trained for this chapter's experiments. The experiments on MAC split the model into two groups. Each grouping is marked with a dashed line and labeled with the number of layers in each group. E.g. the first grouping (1,5) has only the one layer in the first group and five layers in the second group. Each 'TransformerEncoder' operation used in the hidden layers has a dimension of 256 and 8 attention heads.

Grouping	Layer	Operation	Output Shape	# Parameters
	Input	Linear(40, 256)	(batch, frames, 256)	10,496
(1,5)				
	Hidden	TransformerEncoder	(batch, frames, 256)	1.3M
	Hidden	TransformerEncoder	(batch, frames, 256)	1.3M
(3,3)				
	Hidden	TransformerEncoder	(batch, frames, 256)	1.3M
	Hidden	TransformerEncoder	(batch, frames, 256)	1.3M
(5,1)				
	Output	Linear(256, 62)	(batch, frames, 62)	15,934

4.1.1 Implementation

The implementation is done using PyTorch (Paszke et al., 2019) in two variants: a synchronous implementation where first the W-Steps for each group are run in succession and then the Z-Steps in succession, although each step's layers or zs can be optimised in any order or even in parallel. This simplifies certain measurements shown throughout

the following sections. The other variant is a distributed, communication-based implementation to take advantage of the decoupled nature of MAC. Here, multiple machines (cluster nodes) optimise each group's W-Step in parallel, then communicate the updated parameters to the other machines, and subsequently optimise the zs in parallel. As a proof-of-concept, the communication is done by writing to and reading from the files, although more sophisticated Interprocess Communication methods (or IPC) will be significantly faster in practice.

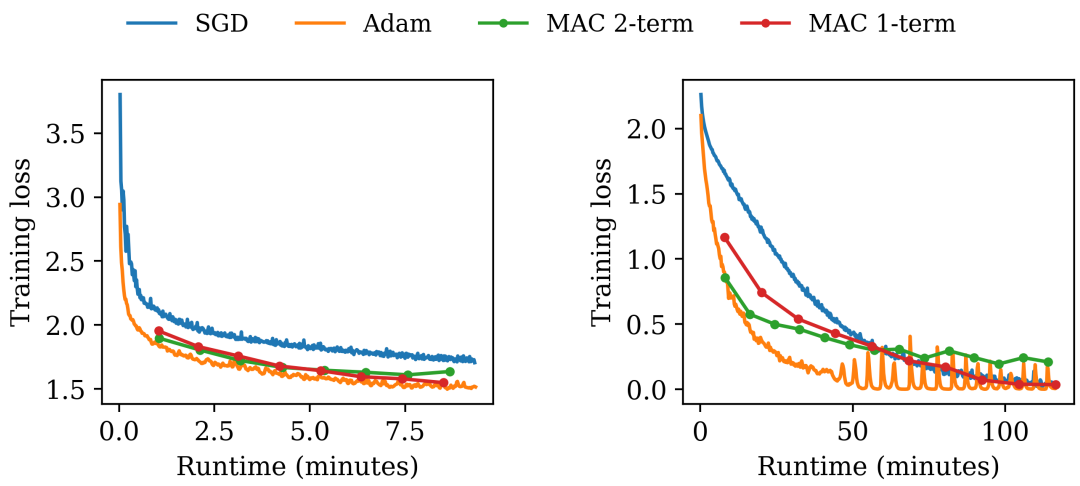
Optimiser for W- and Z-Steps Carreira-Perpiñán and Wang's description of MAC (2014) uses the Gauss-Newton method (Nocedal & Wright, 2006) to optimise the groups in the W-Step and the auxiliary coordinates in the Z-Step. However, because Gauss-Newton is an iterative second-order optimisation algorithm which relies on computing the Hessian as a product of the Jacobian $J^\top J$, the algorithm quickly becomes very expensive and is intractable even at the small scale of my experiments. Therefore, I adapted the algorithm to use Adam as a first-order optimisation algorithm that was specifically developed for training model parameters and requires only the Jacobian.

Hyperparameters With the use of Adam to optimise each W-Step and Z-Step, we also need to specify the learning rate and number of iterations for each group's W- and Z-Steps. To avoid further complexity, I keep the chosen learning rate and iterations constant throughout the training run. Since each auxiliary coordinate $z_\ell^{(i)}$ is optimised independently of the other coordinates, increasing the batch size of the Z-Step does not impact the gradients that are backpropagated to the zs and we can hence set the Z-Step batch size to the maximum that the GPU's memory permits. As suggested by Carreira-Perpiñán and Wang, the Lagrange multiplier μ is initialised to 1 and increased by $10\times$ whenever the training loss plateaus or increases.

Finally, since I am investigating the suitability of MAC as an optimiser, I will be primarily discussing its use in minimising training loss. The generalisation of the resulting models has been mentioned by Carreira-Perpiñán and Wang and any remaining analysis is left for future work.

4.2 How does MAC compare to SGD and Adam?

To provide an initial demonstration of the Method of Auxiliary Coordinates' ability as an optimisation technique to train deep neural networks, Figure 4.1 shows the training loss achieved by the MLP and Transformer models (Tables 4.1 and 4.2) when trained using SGD, Adam and MAC on the TIMIT dataset. The SGD and Adam baselines were each run for 500 epochs, where they converged. Their learning rates were determined by a random search as shown in Appendix C. The MAC results are shown for both, the partially decoupled (2-term) and fully decoupled (1-term) approaches. For the MLP, the (1,5) grouping was used while the (3,3) grouping was used for the Transformer. These groupings were chosen as they resulted in the lowest loss (see Section 4.4). The hyperparameters for MAC were chosen based on the results in Section 4.3. Due to the cluster outage, no hyperparameter search was run for the Transformer model, so the hyperparameters determined for the MLP in Table 4.4 were used across both models.



(a) 6-layer MLP

(b) 4-layer Transformer

Figure 4.1: Training loss of a 6-layer MLP (a) and 4-layer Transformer model (b) over time when trained using different optimisers. The MLP's grouping for both MAC variants is (1,5) and the Transformer's grouping is (3,3). MAC's runtime was measured only for the computational steps and not the communication between machines. All experiments were executed on an NVIDIA GeForce RTX 2080 Ti GPU. The batch sizes were consistent across optimisers: MLP=512, Transformer=40.

MLP We can see from the training plots of the 6-layer MLP (Fig. 4.1a) that although each MAC epoch takes significantly longer than the epochs of SGD and Adam, each epoch also results in a significantly larger decrease in the training loss. In effect, both, the 1-term and 2-term variants of MAC achieve a lower training loss than the batched stochastic gradient descent baseline within the same runtime. While MAC does not outperform Adam, the achieved loss for any runtime is comparable. Notably, the partially decoupled variant (2-term) appears to worsen slightly around minute 9 while the fully decoupled variant continues to improve. Although this appears counter-intuitive given the discussion about the fully decoupled variant being less stable, at the end of Section 3.2.2, the difference is minute and could be due to randomness.

Transformer Although the MLP's hyperparameters (Table 4.4) were used to train the 4-layer Transformer in Figure 4.1b, instead of tuning them specifically for this model, the pattern is similar. Both variants of MAC initially outperform SGD by achieving a lower training loss within a shorter runtime. However, after some epochs the partially decoupled variant begins to plateau above the loss achieved by SGD, while the fully decoupled variant continues to decrease the loss, outperforming SGD. Adam again converges faster, although the loss appears very unstable towards the end with the spikes being caused by exploding gradients which I am clipping. If the hyperparameters are tuned appropriately, the loss curves of MAC might edge closer to Adam, although they are unlikely to beat it, given the MLP's results.

These results show that with the extensions from Chapter 3, MAC can effectively generalise to training more modern deep learning architectures on phone classification

tasks with a non-MSE loss. The experiments further demonstrate that by distributively training models, we can achieve a lower training loss than SGD in less time on TIMIT. While the models' layers were split into only 2 groups, the theoretical analysis of MAC's speedup in Section 3.4.3 suggests that MAC's performance may further improve, likely beating Adam, if the models are divided into more groups.

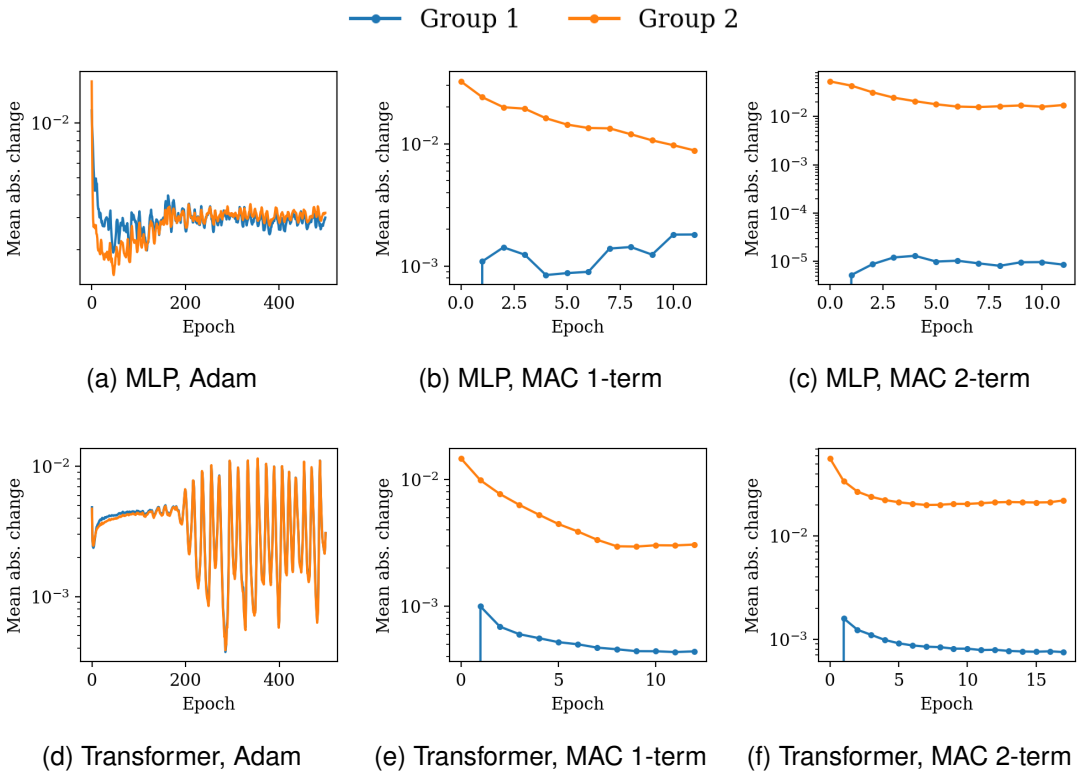


Figure 4.2: Parameter changes per epoch for each group, measured as the mean absolute difference per epoch. The subfigures show different combinations of model type and optimiser, where the top row is the MLP and the bottom row is the Transformer model. The training runs are identical to those in Figure 4.1. MLP: grouping=(1,5) batch size=512, Transformer: grouping=(1,5) batch size=40.

Parameter updates To visualise 'how much' each group gets trained during MAC, I plotted the mean absolute parameter changes in Figure 4.2. The top row compares Adam against MAC's 1-term and 2-term variants when training the MLP. The bottom row compares the same optimisers when training the Transformer. It is interesting to observe that while Adam trains the model by adjusting both groups with magnitude, MAC changes the parameters in the second group significantly more (10 to 1000×). Nonetheless, both optimisers (especially Adam and 1-term MAC) achieve comparable losses. This difference in magnitudes is even present in Figure 4.2e where both groups have almost the same learning rate and number of iterations for both groups' W-Steps. We can further deduce from the previous plots in Figure 4.1a that for the MLP one MAC epoch corresponds to approximately 50 epochs of Adam, which might cause us to expect the parameter changes in MAC to be of a greater magnitude. However, this is not the case either.

It is also worth noting that the first MAC epoch does not change the parameters in Group 1. This is because the auxiliary coordinates are initialised by a forward pass through the model and since the first W-Step is executed before the first Z-Step (following Carreira-Perpiñán and Wang, 2014), the MSE loss for Group 1 is initially 0 until the first Z-Step.

The spikes in Figure 4.2d are the same ones as seen in Figure 4.1b and are again due to exploding gradients, which are being clipped.

4.3 Choosing hyperparameters

Besides knowing that MAC is a feasible method to distributively train deep neural networks, its fundamental differences to stochastic gradient descent and Adam makes it difficult to form a first intuition about how to set the hyperparameters. This section is intended to shine a light on this issue and provide an initial guideline.

Hyperparameters For a model which is split into n groups, each MAC epoch consists of n W-Steps and $n-1$ Z-Steps (see Figures 3.1 and 3.2). Each W and Z-Step is a separate minimisation problem that is optimised using Adam. This makes $2n-1$ hyperparameters for n groups. Besides those, we also need to set the batch size for each W and Z-Step, the Lagrange Multiplier μ and decide on the grouping. However, for this thesis, all batch sizes will be fixed at 512 for the MLP and 40 for the Transformer. The Lagrange Multiplier scheduling follows Carreira-Perpiñán and Wang (2014), and the grouping will be discussed in Section 4.4.

While the number of hyperparameters scales linearly with the number of groups, the cost of searching the hyperparameter space scales exponentially with its size and thus exponentially with the number of groups. Therefore, to get an idea of how to set the hyperparameters without having to search an exponentially large space, I limit the number of groups to two. Because of the cluster outage, I also present only the results for the MLP, although they should sufficiently generalise as demonstrated in Section 4.2.

Search configuration Based on my experience with implementing and testing MAC, I chose the hyperparameter ranges detailed in Table 4.3 for a random search. The number of iterations is restricted to a grid of three values for the W-Step and four values for the Z-Step to keep the search space sufficiently small. The search tried 1500 hyperparameter combinations where each training run lasted for 10 epochs. To reduce the runtime, I applied pruning, although very carefully. Given two sets of hyperparameters, where the first set only trains Group 2 to convergence, while hardly training Group 1, while the second set trains both groups, the first set will achieve a much lower loss in the first few epochs, but the second set will achieve a lower loss overall. Because of this, I can't simply prune a training run when it starts with a higher loss than previous runs. Instead, I decided to let every run continue at least until the loss stops decreasing and then prune if the loss at the current epoch is higher than the median

Table 4.3: Hyperparameter ranges for the random search. The learning rates were log-distributed such that $[10^{-4}, 10]$ is more accurately represented as 10^i where i is uniformly distributed under $[-4, 1]$. The table's notation was used for simplicity. The iterations can only take on the values in their respective sets, similar to a grid search. The ranges were applied to training both the fully and partially decoupled MAC variants.

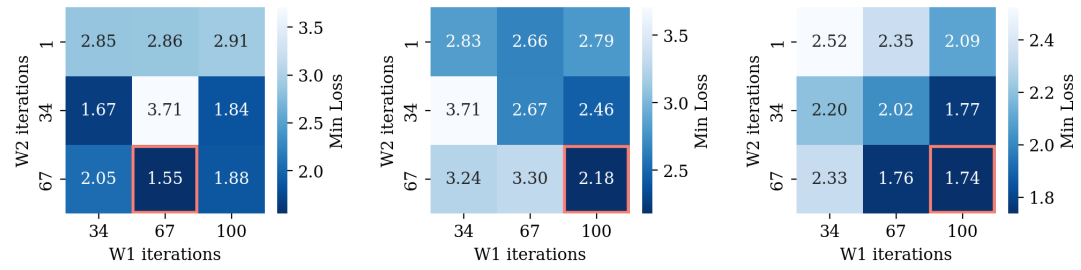
Step	Hyperparameter	Range
W-Step: Group 1	Learning Rate	$[10^{-4}, 10]$
	Iterations	$\{34, 67, 100\}$
W-Step: Group 2	Learning Rate	$[10^{-4}, 10]$
	Iterations	$\{1, 34, 67\}$
Z-Step	Learning Rate	$[10^{-5}, 1]$
	Iterations	$\{1, 15, 30, 45\}$

loss of previous runs at that epoch. While this keeps many unpromising candidates, it importantly avoids discarding good hyperparameter candidates.

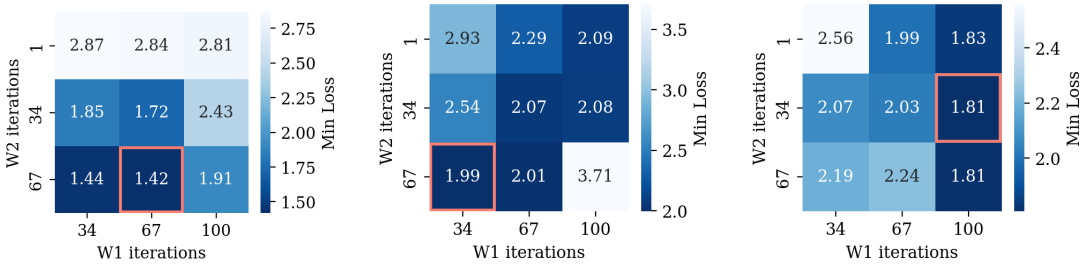
W-Step iterations Recalling the condition under which MAC provides a speedup over SGD (Eq. 3.24), the number of W-Step and Z-Step iterations has a large impact on the speedup. As the search has found the number of Z-Step iterations to be negligible in comparison to the W-Step (Table 4.4), I focus on how the final loss relates to the number of W-Step iterations within each group. Each Subfigure in 4.3 shows the minimum loss achieved for each combination of W-Step iterations in Group 1 (W1) and Group 2 (W2) for a different model and grouping. For instance, Figure 4.3a's bottom left field shows the lowest loss that the MLP was trained to during the search, given that we use fully decoupled MAC, a (1,5) grouping, 34 iterations in Group 1's W-Step and 67 iterations in Group 2's W-Step. The fields highlighted in orange show the best combination of W-Step iterations for each MAC variant and grouping.

Looking at the shading of the heatmaps, especially in the bottom row for the 2-term variant, we can observe that for grouping (1,5) where most layers are in Group 2, changing Group 2's W-Step iterations (W2) has the most impact on the loss, while for grouping (5,1) where most layers are in the first group, changing W1 has the greatest impact. In particular, we want the number of iterations to be high for the group with the most layers. Oddly, in Figure 4.3e with grouping (3,3) where both groups have the same number of layers, using the maximum number of iterations for both groups leads to the worst performance. However interestingly, a W1-W2 combination of 67-67 or 34-67 appears to be reasonably good. The overall best combination of iterations, grouping and decoupling variant for training a 6-layer MLP on TIMIT appears to be to use 2-term MAC with a (1,5) grouping and W1=W2=67.

W-Step learning rates The best hyperparameters for each MAC variant and grouping are compiled in Table 4.4. Looking at the learning rates for groups 1 and 2, we can again see a pattern: The group with more layers tends to have a lower learning rate. If the layers are approximately evenly split between the groups, the learning rates are also



(a) 1-term, Grouping (1,5) (b) 1-term, Grouping (3,3) (c) 1-term, Grouping (5,1)



(d) 2-term, Grouping (1,5) (e) 2-term, Grouping (3,3) (f) 2-term, Grouping (5,1)

Figure 4.3: Training losses achieved on the MLP with the best hyperparameters given the number of W-Step iterations for each group. The subplots show different combinations of decoupling variant and grouping. The best W-Step iterations are highlighted in orange within each subplot.

approximately equal.

Z-Step Looking at the same Table's (4.4) Z-Step column, we can make out that groupings with a larger Group 1 require a larger Z-Step by either increasing the number of iterations as for 1-term (5,1) or increasing the learning rate as for 2-term (5,1). This makes intuitive sense since the objective is passed down to Group 1 via the Z-Step. The larger the Z-Step, the more of the task gets passed on to Group 1.

Table 4.4: Best Hyperparameters for each MAC variant and grouping of the MLP. The hyperparameters were found by a random search over the ranges in Table 4.3, over 1,500 trials. Each trial trained for 10 epochs and the final training loss was recorded.

	Grouping	W-Step: Group 1		W-Step: Group 2		Z-Step		Loss
		Learning Rate	Iterations	Learning Rate	Iterations	Learning Rate	Iterations	
1-term	(1,5)	2.8×10^{-2}	67	2.5×10^{-3}	67	1.4×10^{-5}	1	1.6
	(3,3)	1.2×10^{-4}	100	1.5×10^{-4}	67	1.0×10^{-4}	1	2.2
	(5,1)	2.1×10^{-3}	100	1.3×10^{-2}	67	8.7×10^{-4}	15	1.7
2-term	(1,5)	3.4×10^{-4}	67	6.7×10^{-3}	67	3.0×10^{-5}	1	**1.4**
	(3,3)	2.8×10^{-4}	34	9.2×10^{-4}	67	1.4×10^{-5}	1	2.0
	(5,1)	1.2×10^{-3}	100	3.1×10^{-3}	34	1.0×10^{-1}	1	1.8

Summary This section explored how to set the hyperparameters when using MAC to distributively train deep neural networks. A random search was conducted over a limited

hyperparameter space, focusing on a 6-layer MLP split into two groups to keep the search tractable. The results provide initial guidelines for setting MAC hyperparameters:

- The number of W-Step iterations should be higher for the group containing more layers. When both groups are evenly split, use roughly the same number of iterations.
- The learning rate for the W-Step tends to be lower for the group with more layers. When layers are approximately evenly divided, the learning rates are also roughly equal between groups.
- Groupings with a larger first group require a larger Z-Step, either by increasing the number of iterations or the learning rate. This makes the first group learn a bigger part of the task.

The best overall hyperparameters found for training the 6-layer MLP on TIMIT were to use 2-term MAC with a (1,5) layer grouping, 67 iterations for both W-Steps and a single iteration Z-Step. While these findings provide helpful intuition, the optimal hyperparameter settings will likely vary based on the specific model architecture, dataset, and number of groups. Further research is needed to develop more generalisable hyperparameter recommendations for MAC.

4.4 Grouping layers for MAC

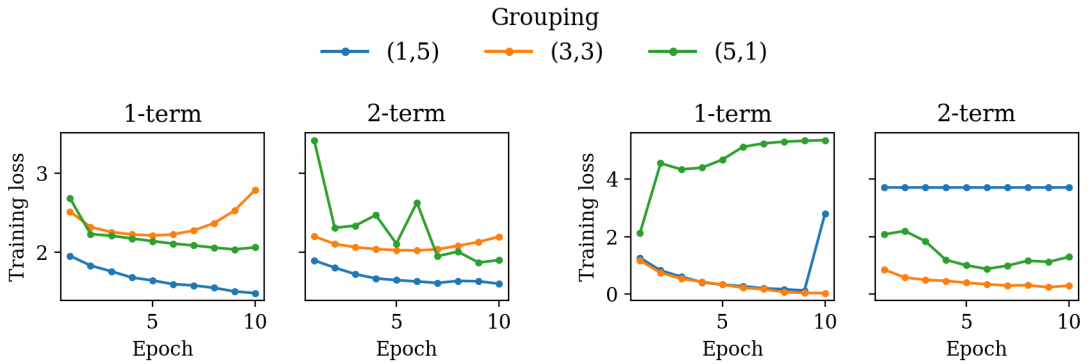
In the previous sections, I already mentioned some differences between the groupings, particularly in choosing hyperparameters. Figure 4.4 further visualises the varying training patterns of the groupings for both decoupling variants and both models.

Looking only at Figure 4.4a on training the MLP, both the fully (1-term) and partially (2-term) decoupled variants of MAC show a similar trend where the (1,5) achieves the best results, next the (5,1) and although the even grouping starts off reasonably well, it ends up performing the worst. The plots for the Transformer model in the bottom Figure 4.4b show a very different pattern. Here, the (3,3) grouping does best, again for both variants. However, with the chosen Hyperparameters, there is no clear winner when comparing the (1,5) and (5,1) groupings. I cannot make out any pattern for which grouping to choose for which architecture, and given a previously untested model, all three groupings would need to be tried (or possibly even others).

Besides demonstrating MAC's sensitivity to the grouping of layers, the fact that the (5,1) grouping does comparatively well is also evidence that MAC not only trains the top group (Group 2) but also effectively trains the bottom group which is further removed from the output and thus the objective.

4.5 Conclusion on decoupled MAC

The experiments in this chapter have demonstrated that the decoupled Method of Auxiliary Coordinates can effectively train deep neural networks such as MLPs, Transformers and likely others on speech tasks like phone classification on the TIMIT dataset. By



(a) MLP (b) Transformer

Figure 4.4: Training losses for each grouping of the 6-layer MLP (a) and 4-layer Transformer (b) when trained with partially or fully decoupled MAC. All hyperparameters except for the batch size were the same for both models and are found in Table 4.4. Batch sizes: MLP=512, Transformer=40.

optimising the model in two groups, distributing the groups as visualised in Figures 3.1 and 3.2, MAC can achieve a lower training loss than SGD in less time. While MAC did not outperform Adam in the experiments, the theoretical speedup analysis in Section 3.4.3 suggests that MAC's performance may further improve, likely beating Adam, if the models are divided into more than two groups.

As the theoretical analysis in Section 3.2.2 shows that 1-term MAC requires less communication between machines, and the above results demonstrate that it consistently outperforms the 2-term variant, fully decoupling should be recommended.

The hyperparameter search provided initial guidelines for setting the learning rates and number of iterations for the W-Steps and Z-Steps based on the relative sizes of the layer groups. However, the optimal hyperparameter settings will likely vary based on the specific model architecture, dataset, and number of groups. Further research is needed to develop more generalisable hyperparameter recommendations.

The sensitivity of MAC to how the model layers are grouped was also shown. Although different groupings are optimal for the MLP and Transformer models, with no apparent pattern, the best grouping (when found) performs well. Furthermore, MAC's reasonable performance even with a grouping of (5,1) that is heavily imbalanced towards the input demonstrates its ability to train both the last and earlier layers of the model.

In summary, the decoupled Method of Auxiliary Coordinates is a promising optimisation technique for distributively training deep neural networks. Despite requiring more computations per layer than SGD or Adam, MAC's parallelisation can offset the computational overhead with as little as two groups. While MAC is unlikely to be used by individuals or academics training small to medium-sized models, it has potential for companies training very deep models with access to large data centres.

To fully harness MAC's potential in these use cases, however, future work should focus on developing principled methods for choosing hyperparameters and layer groupings.

Chapter 5

Conclusion

This thesis investigated the Method of Auxiliary Coordinates (MAC) as an alternative optimisation framework for the distributed training of deep neural networks. Several extensions were proposed to enable MAC's application to modern architectures and tasks beyond the autoencoding use case originally explored by Carreira-Perpiñán and Wang (2014).

5.1 Contributions

The primary contributions of this thesis were:

- Decoupling the Z-Step update rules by proposing a partially decoupled and fully decoupled MAC variant. The partially decoupled variant makes the implicit assumption of Carreira-Perpiñán and Wang (2014) explicit, while the fully decoupled variant further improves upon it by reducing the amount of data that needs to be transferred between machines after each epoch. The explicit introduction of these variants ultimately improves the distributability and effectiveness of MAC.
- Adapting MAC to use the first-order Adam optimiser in its W and Z-Step, instead of the significantly more expensive Gauss-Newton method used by Carreira-Perpiñán and Wang (2014), thereby making the implementation faster.
- Adjusting MAC's update rules to support custom differentiable loss functions and modern architectures such as Transformers.
- A description of the theoretical speedup offered by distributed MAC over standard techniques like SGD and Adam. Specifically, this speedup was found to be proportional to the number of groups into which the model is split for the optimisation, and inversely proportional to the number of iterations required for each W and Z-Step.
- Experiments on how the runtime of MAC, using the determined optimal hyperparameters and layer grouping, compares to SGD and Adam baselines.
- An evaluation of the different possible ways in which we can split the model and

its effect on MAC's performance. Further, analyses of how to set the hyperparameters and the W and Z-Step iterations to take the most advantage of MAC's distributability.

5.2 Limitations

While the proposed extensions enabled MAC to train modern architectures on speech tasks, there were some notable limitations:

- Although the 6-layer MLP and 4-layer Transformer models and the TIMT dataset used in my experiments were larger and more complex than those used by Carreira-Perpiñán and Wang (2014), they were nonetheless small in comparison to more commonly used models and benchmarks (Hsu et al., 2021; Panayotov et al., 2015).
- While the hyperparameter selection guidelines provided in this thesis should give the reader an initial idea of the ranges and patterns, they are limited to models split into two groups and were only determined using the multi-layer perceptron. However, the hyperparameters have been shown to also provide reasonable performance when training a Transformer-based model. They may generalise further.
- Despite fully decoupled MAC showing improved performance when compared to both the coupled and partially decoupled variants, the number of auxiliary coordinates introduced by MAC scale linearly with the dataset size. As all of a layer's coordinates need to be transferred from one machine to another after each epoch, the communication cost rises significantly for larger datasets.

5.3 Future work

Based on the findings and limitations of this thesis, several directions for future research can be identified:

- Develop approaches, approximations, or simplifications that reduce the number of hyperparameters required by MAC. Following this improvement, it is useful to develop more detailed guidelines for selecting hyperparameters and layer groupings based on the model architecture and dataset.
- Explore alternative approaches to distributing the computations that avoid transferring all $\mathbf{z}$s after each epoch and instead transfer the model's parameters which tend to be smaller and thus faster to send between machines.

While the current work demonstrates MAC's potential as a distributed optimisation method, addressing these remaining limitations would transform the Method of Auxiliary Coordinates into a viable technique for efficiently training large-scale deep learning models.

Bibliography

Agarap, A. F. (2018). Deep learning using rectified linear units (relu). *arXiv preprint*, *abs/1803.08375*.

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization.

Bilmes, J. A., Asanovic, K., Chin, C., & Demmel, J. (1997). Using phipac to speed error back-propagation learning. *ICASSP*.

Carreira-Perpiñán, M. Á., & Alizadeh, M. (2019). Parmac: Distributed optimisation of nested functions, with application to learning binary autoencoders. *MLSys*.

Carreira-Perpiñán, M. Á., & Lu, Z. (2008). Dimensionality reduction by unsupervised regression. *CVPR*.

Carreira-Perpiñán, M. Á., & Wang, W. (2012). Distributed optimization of deeply nested systems. *arXiv preprint*, *abs/1212.5921*.

Carreira-Perpiñán, M. Á., & Wang, W. (2014). Distributed optimization of deeply nested systems. *AISTATS*.

Chai, J., Zeng, H., Li, A., & Ngai, E. W. (2021). Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *MLWA*, *6*.

de Benito, D., Lozano-Diez, A., Toledano, D., & Gonzalez-Rodriguez, J. (2019). Exploring convolutional, recurrent, and hybrid deep neural networks for speech and music detection in a large audio dataset. *EURASIP JASM*, *2019*.

Deng, J., Dong, W., Socher, R., Li, L., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. *CVPR*.

Dhar, P. (2020). The carbon impact of artificial intelligence. *Nature Machine Intelligence*, *2*.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. *ICLR*.

Encyclopaedia of mathematics. (2002).

Garofolo, John S., Lamel, Lori F., Fisher, William M., Pallett, David S., Dahlgren, Nancy L., Zue, Victor, & Fiscus, Jonathan G. (1993). Timit acoustic-phonetic continuous speech corpus.

Graves, A., Fernández, S., Gomez, F. J., & Schmidhuber, J. (2006). Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. *ICML*, *148*.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., & Gibbons, P. B. (2018). Pipedream: Fast and efficient pipeline parallel DNN training. *arXiv preprint*, *abs/1806.03377*.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *CVPR*.

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint*, *abs/1207.0580*.

Hsu, W., Bolte, B., Tsai, Y. H., Lakhotia, K., Salakhutdinov, R., & Mohamed, A. (2021). Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *ACM*, *29*.

Huang, G., Sun, Y., Liu, Z., Sedra, D., & Weinberger, K. Q. (2016). Deep networks with stochastic depth. *ECCV*, *9908*.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M. X., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., & Chen, Z. (2019). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS*.

Hull, J. J. (1994). A database for handwritten text recognition research. *IEEE*, *16*.

Kahn, J., Rivière, M., Zheng, W., Kharitonov, E., Xu, Q., Mazaré, P., Karadayi, J., Liptchinsky, V., Collobert, R., Fuegen, C., Likhomanenko, T., Synnaeve, G., Joulin, A., Mohamed, A., & Dupoux, E. (2020). Libri-light: A benchmark for ASR with limited or no supervision. *ICASSP*.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. *ICLR*.

Lacoste, A., Luccioni, A., Schmidt, V., & Dandres, T. (2019). Quantifying the carbon emissions of machine learning. *arXiv preprint*, *abs/1910.09700*.

Larsson, G., Maire, M., & Shakhnarovich, G. (2017). Fractalnet: Ultra-deep neural networks without residuals. *ICLR*.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *IEEE*, *86*.

Martin, P. (2023). *Knowledge distillation for end-to-end asr in resource-constrained environments* [Bachelor's Thesis].

Nene, S. A., Nayar, S. K., Murase, H., et al. (1996). Columbia object image library (coil-20).

Nichols, D., Singh, S., Lin, S.-H., & Bhatele, A. (2021). A survey and empirical evaluation of parallel deep learning frameworks.

Nocedal, J., & Wright, S. (2006). *Numerical optimization*.

Panayotov, V., Chen, G., Povey, D., & Khudanpur, S. (2015). Librispeech: An ASR corpus based on public domain audio books. *ICASSP*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *NeurIPS*.

Paul, D. B., & Baker, J. M. (1992). The design for the wall street journal-based CSR corpus. *Proceedings of the workshop on Speech and Natural Language - HLT '91*.

Roger, V., Farinas, J., & Pinquier, J. (2022). Deep neural networks for automatic speech processing: A survey from large corpora to limited data. *EURASIP JASM*, *2022*.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*.

Stevens, S. S., Volkmann, J., & Newman, E. B. (1937). A Scale for the Measurement of the Psychological Magnitude Pitch. *JASA*, *8*.

Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, *13*.

Torfi, A., Shirvani, R. A., Keneshloo, Y., Tavaf, N., & Fox, E. A. (2020). Natural language processing advancements by deep learning: A survey. *arXiv preprint*, *abs/2003.01200*.

van den Oord, A., Li, Y., & Vinyals, O. (2018). Representation learning with contrastive predictive coding. *arXiv preprint*, *abs/1807.03748*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *NeurIPS*, *30*.

Williams, V. V., Xu, Y., Xu, Z., & Zhou, R. (2023). New bounds for matrix multiplication: From alpha to omega. *arXiv preprint*, *abs/2307.07970*.

Yang, G., Yeh, S., Chung, Y., Glass, J. R., & Tang, H. (2022). Autoregressive predictive coding: A comprehensive study. *IEEE J-STSP*, *16*.

Appendix A

Schoolbook Matrix Multiplication

The schoolbook algorithm for matrix multiplication is the algorithm typically taught in schools and is the way most people multiply matrices by hand, although it is not the most efficient way to multiply matrices. The algorithm is as follows:

Algorithm A.1 Schoolbook Matrix Multiplication

Input: $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$
Output: $C = AB \in \mathbb{R}^{m \times p}$
Initialize $C \in \mathbb{R}^{m \times p}$ to be the zero matrix
for $i = 1$ to m **do**
    for $j = 1$ to p **do**
        for $k = 1$ to n **do**
            $C_{ij} \leftarrow C_{ij} + A_{ik} \cdot B_{kj}$
        end for
    end for
end for

It should be easy to see that the time complexity of this algorithm is $O(mnp)$. For square matrices of size $n \times n$, the time complexity is $O(n^3)$.

Appendix B

Partial derivatives of the coupled and decoupled Z-Step update rules

The following are the partial derivatives for the coupled, partially decoupled and fully decoupled Z-Step updates for a single auxiliary coordinate $\mathbf{z}_\ell^{(i)}$. For simplicity, let $\mathbf{z}_\ell \triangleq \mathbf{z}_\ell^{(i)}$ and $f_\ell(\cdot) \triangleq f_\ell(\cdot\,;\,\boldsymbol{\theta}_\ell)$.

Coupled (Eq. 3.5):

$$\begin{aligned}
\frac{\partial E_Q}{\partial \mathbf{z}_\ell} &= \frac{\partial}{\partial \mathbf{z}_\ell}\left(\frac{1}{2}\left\|\mathbf{y}-f_L(\mathbf{z}_L)\right\|^2 + \frac{\mu}{2}\sum_{\ell'=1}^{L}\left\|\mathbf{z}_{\ell'}-f_{\ell'-1}(\mathbf{z}_{\ell'-1})\right\|^2\right) \\
&= \frac{\partial}{\partial \mathbf{z}_\ell}\left(\frac{\mu}{2}\left\|\mathbf{z}_{\ell+1}-f_\ell(\mathbf{z}_\ell)\right\|^2 + \frac{\mu}{2}\left\|\mathbf{z}_\ell-f_{\ell-1}(\mathbf{z}_{\ell-1})\right\|^2\right) \\
&= \left(\frac{\mu}{2}\right) 2\left\|\mathbf{z}_{\ell+1}-f_\ell(\mathbf{z}_\ell)\right\|^2 \frac{f_\ell(\mathbf{z}_\ell)-\mathbf{z}_{\ell+1}}{\left\|\mathbf{z}_{\ell+1}-f_\ell(\mathbf{z}_\ell)\right\|}\left(\frac{\partial f_\ell(\mathbf{z}_\ell)}{\partial \mathbf{z}_\ell}\right) \\
&\qquad + \left(\frac{\mu}{2}\right) 2\left\|\mathbf{z}_\ell-f_{\ell-1}(\mathbf{z}_{\ell-1})\right\|^2 \frac{f_{\ell-1}(\mathbf{z}_{\ell-1})-\mathbf{z}_\ell}{\left\|\mathbf{z}_\ell-f_{\ell-1}(\mathbf{z}_{\ell-1})\right\|} \\
&= \mu\left((f_\ell(\mathbf{z}_\ell)-\mathbf{z}_{\ell+1})\frac{\partial f_\ell(\mathbf{z}_\ell)}{\partial \mathbf{z}_\ell} + \mathbf{z}_\ell - f_{\ell-1}(\mathbf{z}_{\ell-1})\right).
\end{aligned} \tag{B.1}$$

Partially decoupled (Eq. 3.9):

$$\begin{aligned}
&\frac{\partial}{\partial \mathbf{z}_\ell}\left(\frac{\mu}{2}\left\|\mathbf{z}_{\ell+1}-f_\ell(\mathbf{z}_\ell)\right\|^2 + \frac{\mu}{2}\left\|\mathbf{z}_\ell-f_{\ell-1}(\mathbf{z}_{\ell-1})\right\|^2\right) \\
&= \cdots \text{ (same as coupled) } \cdots \\
&= \mu\left((f_\ell(\mathbf{z}_\ell)-\mathbf{z}_{\ell+1})\frac{\partial f_\ell(\mathbf{z}_\ell)}{\partial \mathbf{z}_\ell} + \mathbf{z}_\ell - f_{\ell-1}(\mathbf{z}_{\ell-1})\right).
\end{aligned} \tag{B.2}$$

Fully decoupled (Eq. 3.12):

$$\begin{aligned}
&\frac{\partial}{\partial \mathbf{z}_\ell}\left(\frac{\mu}{2}\left\|\mathbf{z}_{\ell+1}-f_\ell(\mathbf{z}_\ell)\right\|^2\right)\\
&=\left(\frac{\mu}{2}\right)2\left\|\mathbf{z}_{\ell+1}-f_\ell(\mathbf{z}_\ell)\right\|^2\frac{f_\ell(\mathbf{z}_\ell)-\mathbf{z}_{\ell+1}}{\left\|\mathbf{z}_{\ell+1}-f_\ell(\mathbf{z}_\ell)\right\|}\left(\frac{\partial f_\ell(\mathbf{z}_\ell)}{\partial \mathbf{z}_\ell}\right)\\
&=\mu\left(\left(f_\ell(\mathbf{z}_\ell)-\mathbf{z}_{\ell+1}\right)\frac{\partial f_\ell(\mathbf{z}_\ell)}{\partial \mathbf{z}_\ell}\right).
\end{aligned} \tag{B.3}$$

As the above formulations are independent of the network architecture choice, the partial derivative $\frac{\partial f_\ell(\mathbf{z}_\ell)}{\partial \mathbf{z}_\ell}$ is to be computed accordingly.

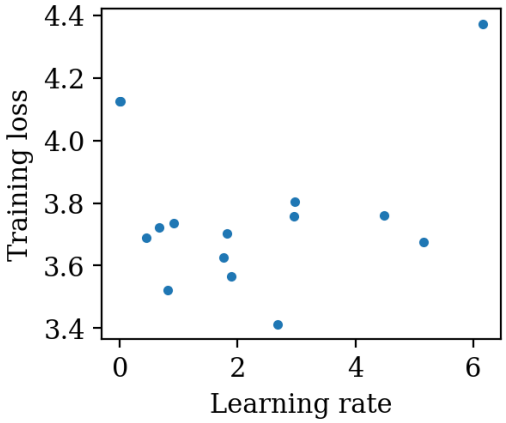
Appendix C

Learning rate searches for SGD and Adam

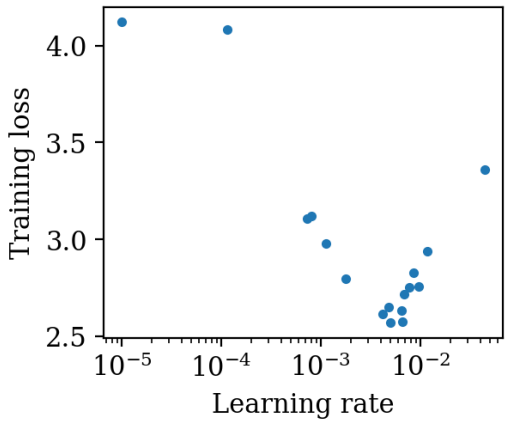
Figure C.1 shows the results of the random hyperparameter searches for the learning rate, performed for Stochastic Gradient Descent (SGD) and Adam on a 6-layer MLP (see Table 4.1, $N = 4$) and a 4-layer Transformer network (see Table 4.2, $N = 6$). All four searches were done over the range $[10^{-5}, 10^{2}]$ and for 20 epochs. Runs that appeared not promising were discarded before completion. The best learning rates are found in Table C.1.

Table C.1: Best learning rates according to the searches from Figure C.1.

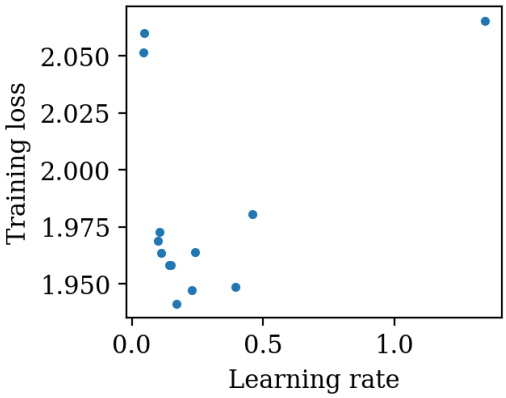
	SGD	Adam
6-layer MLP	2.7×10^{0}	5.0×10^{-3}
4-layer Tranformer	1.7×10^{-1}	7.9×10^{-4}



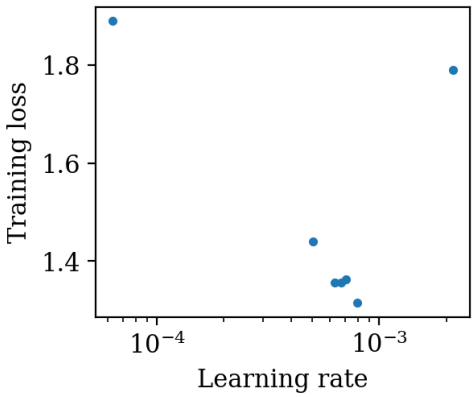
(a) SGD for a 6-layer MLP



(b) Adam for a 6-layer MLP



(c) SGD for a 4-layer Transformer



(d) Adam for a 4-layer Transformer

Figure C.1: Random search of the learning rate for SGD and Adam for two architectures used in Chapter 4. Unpromising runs were pruned before completion.