YaleNUSCollege

Improved Adaptive Optimization for Language Modeling

Andrew SIOW

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

Supervised by: Prof. Michael CHOI

AY 2023/2024

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

[✓ o Unrestricted access]
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from _____________ (mm/yyyy) to _______________ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
____.

After this period, the Thesis will be made available for worldwide access.

o Other restrictions: (please specify if any part of your thesis should be restricted)
____
____

Andrew Siow Wen Jie - Elm College
Name & Residential College of Student

[illegible]
Signature of Student

04-04-2024
Date

[illegible] Michael Choi
Name & Signature of Supervisor

5th April 2024
Date

Acknowledgements

I wish to thank my supervisor, Professor Michael Choi, for providing the novel idea for this paper and suggesting different ways I could expand on existing research, for his untiring and engaging guidance on academic research throughout this entire project, and for granting me the freedom to pursue interesting questions.

In this project, I also built on important empirical groundwork laid by AdaBelief authors Zhuang et al. and previous research on landscape modification in deep learning by Todea and Roata. I thank them all for their work that granted me more time to deeply consider and research this topic.

I also wish to thank Nicholas and Christine for proofreading my paper. All remaining errors are mine.

I thank God for lovingly being with me in all things, and providing me with all these important people in my life, whom I am deeply thankful for:

My parents, for loving me, providing abundantly for me, and instilling in me a joy in life and a love of learning.

My sister Christine, for entertaining my antics and always having my best interests at heart.

Nicole, for giving me so much encouragement throughout this research and being someone whom I appreciate having in life.

David, Makarios, and my many other friends in church, for loving me, showing me true friendship, and reminding me of who I am.

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Improved Adaptive Optimization for Language Modeling

by Andrew SIOW

Researchers aim to develop neural network optimization algorithms that converge quickly and train models which perform well on the validation set [34]. Empirically, while adaptive optimizers converge quickly, simpler optimizers tend to train models that generalize better [30]. We motivate a new hypothesis that landscape modification in deep learning biases convergence to flatter minima, which generalize better. Thus, we aim to improve adaptive optimization by implementing loss landscape modification into an adaptive optimizer, AdaBelief, to develop AdaBelief-LM. Empirical results are consistent with the new hypothesis, and show that AdaBelief-LM generally achieves lower validation loss than AdaBelief for language modeling. This paper extends research by offering a novel hypothesis about landscape modification's impact in deep learning and demonstrating its usefulness in language modeling.

Keywords: Landscape Modification, Optimization, Language Modeling, Deep Learning, AdaBelief, AdaBelief-LM

Contents

List of Tables

List of Figures

Chapter 1

Optimization for Neural Networks

1.1 Introduction

In recent years, new models have been introduced that can perform remarkably well over a range of tasks like language processing and image classification [13]. Many of these new advancements have been supported by a novel framework: neural networks [13, 32]. Training a neural network involves iteratively tweaking its many parameters based on training data, using an optimization algorithm [34]. A specialized field within machine learning focuses on developing optimization algorithms for neural networks. Critically, while the goal of pure optimization is to minimize training loss, the ultimate goal in neural network optimization is to minimize loss from data the model has not been trained on, also known as validation loss [34].

1.2 The Function of Optimization Algorithms

Although the ultimate goal of neural network optimization is minimizing validation loss, optimization algorithms still aim to minimize the training loss function so that the neural network makes predictions which fit the training data as closely as possible [34]. At the beginning of training, neural networks are usually initialized with random weights [32]. A batch of training data is then passed to the model, and its predictions are obtained [32]. We compare the model's predictions against ground truth from the training data and obtain a measure of how accurate the model's predictions were using a loss function [32]. To illustrate, suppose we have a set of training data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ and a vector of model weights $w \in \mathbb{R}^d$ [3]. For a specific observation (x_i, y_i), the model's prediction can be expressed as $h(x_i; w)$ where h is a prediction function that depends on the weights w [3]. We then apply a loss function $\mathcal{L}$ to both the prediction $h(x_i; w)$ and the desired output y_i [3]. We then average the loss over each observation in the set of training data [3]. Mathematically, our goal is to find

$$\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(h(x_i; w), y_i). \tag{1.1}$$

The loss function, $\mathcal{L}$, defines how we want to calculate the difference between a model's predictions and the truth. Typically, the choice of loss function depends on the task the model is performing. For example, for classification tasks, cross entropy loss is often used to compare a model's predictions to the desired output [18]. Essentially, given the training data, the loss function allows us to both calculate a loss and observe a gradient for each of the model's weights. Optimization algorithms typically use these gradients to adjust each of the model weights to better fit the sample of training data for the next iteration of training [34]. The optimization process can be intuitively thought of as the algorithm taking steps to descend a dark mountain, where the dark mountain is the graph drawn by the loss function.

1.3 AdaBelief Optimization Algorithm

The AdaBelief optimizer, developed by Zhuang et al., is a high-performing optimization algorithm that can be used in neural network training [37]. AdaBelief is an adaptive optimizer, which means that it adjusts the learning rate and update step during training based on historic gradient information, not only the last observed gradient [34]. Other optimizers, such as RMSProp, use an exponential moving average (EMA) of the square of historic gradients to modify the learning rate set by the user [34]. AdaBelief differs from other optimizers in that it tracks the EMA of the squared difference between the observed gradient and expected gradient to modify the learning rate [37]. The following subsections will introduce AdaBelief and give some context as to its performance in the domain of language modeling.

1.3.1 Algorithm

Concisely, AdaBelief begins by randomly initializing a vector of weights θ_0, and setting $m_0 = 0$, $s_0 = 0$, and $t = 0$ [37]. Then, for each subsequent time step t, given a gradient vector g_t already computed through backpropagation, AdaBelief makes an update by calculating θ_t as follows [37]:

1. Calculate $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$.
2. Calculate $s_t = \beta_2 s_{t-1} + (1 - \beta_2)(g_t - m_t)^2 + \epsilon$.
3. Calculate the unbiased estimates $\widehat{m_t} = \frac{m_t}{1-\beta_1}$, $\widehat{s_t} = \frac{s_t}{1-\beta_2}$.
4. Update the model's weights, $\theta_t = \theta_{t-1} - \frac{\alpha \widehat{m_t}}{\sqrt{\widehat{s_t}} + \epsilon}$.

1.3.2 Algorithm Walkthrough

Firstly, let $g_t \in \mathbb{R}^d$ be a vector of the observed gradients for each of the d weights. This vector is calculated at time step t. We then calculate a momentum parameter $m_t \in \mathbb{R}^d$ given by

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

where $\beta_1 \in [0, 1]$ is a hyperparameter, adjustable by the user [37]. In simple terms, m_t is an exponential moving average (EMA) of past gradients.

Secondly, we compute $s_t \in \mathbb{R}^d$, also known as the vector containing the EMA of squared differences between observed and expected gradients for each weight at time t. Mathematically,

$$s_t = \beta_2 s_{t-1} + (1 - \beta_2)(g_t - m_t)^2 + \epsilon$$

where $\beta_2 \in [0, 1]$ is a hyperparameter and ϵ is a small value [37]. All operations are element-wise, so s_t is still a vector of dimension $\mathbb{R}^d$. Here, m_t is used as as the expectation of the observed gradient g_t. Intuitively, since m_t captures both the size and direction of historic gradients, it is reasonable to use m_t as an expectation of g_t [37]. $(g_t - m_t)^2$ represents the "surprise" in g_t, and it increases the more g_t deviates from m_t. Logically, s_t can therefore be used to encourage smaller, cautious steps to be made when gradients keep changing.

For both m_t and s_t, a bias correction is applied as follows:

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1} \qquad \widehat{s_t} = \frac{s_t}{1 - \beta_2}.$$

In the original paper, the authors express the bias correction using β_1^t and β_2^t instead of β_1 and β_2, as the user can adjust these hyperparameters during training [37] (adjusting hyperparameters such as the base learning rate during training is also known as scheduling [34]). However, β_1 and β_2 are usually fixed during training.

Finally, let $\theta_t \in \mathbb{R}^d$ denote the vector of weights and biases for the neural network at time t. Let $i \in \{1, 2, \ldots, d\}$ index a unique weight or bias w_i in the neural network. Then at time t, each w_i has value $\theta_t^{(i)}$ with $\theta_t = [\theta_t^{(1)}, \theta_t^{(2)}, \ldots, \theta_t^{(d)}]$. Under AdaBelief, θ_t is computed as follows:

$$\theta_t = \theta_{t-1} - \frac{\alpha \widehat{m_t}}{\sqrt{\widehat{s_t}} + \epsilon}$$

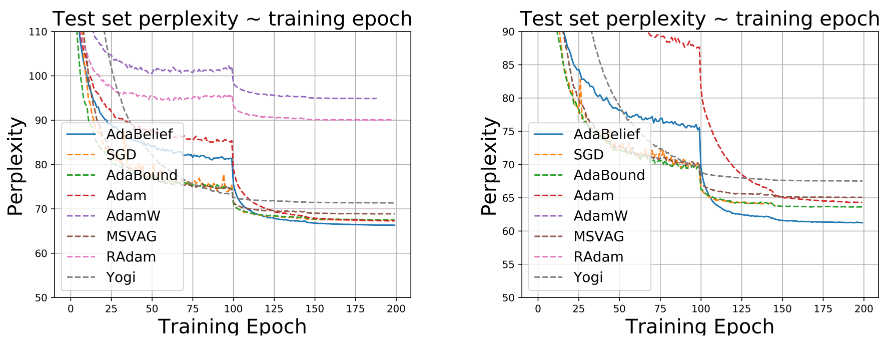
where α is a learning rate hyperparameter, and ϵ is a small value [37]. All operations here are element-wise [37]. Hence, at each iteration the weight

w_i is updated with this update equation:

$$\theta_t^{(i)} = \theta_{t-1}^{(i)} - \frac{\alpha \widehat{m_t}^{(i)}}{\sqrt{\widehat{s_t}^{(i)}} + \epsilon}.$$

1.3.3 AdaBelief Performance in Language Modeling

Beyond simply being an adaptive optimizer, AdaBelief has shown impressive performance when used to train benchmark language model architectures and datasets [37].



(A) 2 Layer LSTM (B) 3 Layer LSTM

FIGURE 1.1: Validation Performance of different LSTMs trained over Penn TreeBank using different optimizers. Lower perplexity is better. [36]

Figure 1.1 depicts the performance of various optimizers, including AdaBelief, when used to train a certain model architecture (LSTM networks) as a language model. AdaBelief's convergence speed remains similar to other optimizers, and it outperforms stochastic gradient descent on validation set performance. This is significant considering that stochastic gradient descent tends to outperform adaptive optimizers in validation performance [30]. In this project, we will see if AdaBelief can be further improved by implementing landscape modification.

Chapter 2

Landscape Modification

2.1 Introduction

In deep learning, the loss landscapes drawn by loss functions are usually non-convex with many local minima [34]. Adaptive optimization algorithms typically have features that help them escape minima, such as relying on exponential moving averages to determine step direction, thus implementing a form of momentum. In addition, researchers have also developed certain methods, such as learning rate warm-up and gradient clipping, to help parameters escape minima [21]. Landscape modification is novel in that it helps parameters escape minima by modifying the loss landscape itself. In fact, landscape modification was first proposed as a means to improve simulated annealing with Langevin dynamics, another optimization algorithm [9]. I will proceed by outlining the origins of landscape modification and explaining how we can apply it to machine learning optimization. At the end of this chapter, I propose a new hypothesis by which to understand the impact of landscape modification and present AdaBelief-LM, an augmentation of AdaBelief that implements landscape modification.

2.2 Simulated Annealing with Langevin Dynamics

Landscape modification has its origins in literature regarding a group of optimization algorithms related to simulated annealing with Langevin dynamics. Simulated annealing borrows its name from annealing in metallurgy, where a metal is heated to a high temperature before a gradual cooling process allows particles to settle into a better position, enhancing the metal's physical properties [14]. Langevin dynamics model the stochastic behaviour of particles at varying temperatures [2]. 2.1 is the classic overdamped Langevin equation that models Brownian motion.

$$\mathrm{d}Z_t = -\nabla U(Z_t)\,\mathrm{d}t + \sqrt{2\epsilon_t}\,\mathrm{d}B_t \tag{2.1}$$

Here, Z_t represents the position of a particle at time t, under the influence of some force ∇U and some random noise provided by Brownian motion $(B_t)_{t\geq 0}$ [5]. To incorporate simulated annealing into this Langevin equation, we can define a sequence $(\epsilon_t)_{t\geq 0}$ to schedule cooling, which reduces the noise encountered by the particle [5]. Provided that $(\epsilon_t)_{t\geq 0}$ is appropriately defined, and we treat $U : \mathbb{R}^d \to \mathbb{R}$ as a target function to minimize, 2.1 characterizes an optimization algorithm [9]. In fact, simulated annealing with Langevin dynamics (SA) is a widely known algorithm to find the global minimum of any target function U, even if U is non-convex. Impressively, researchers have also proven that this algorithm indeed converges to the global minimum of U under a logarithmic cooling schedule of the form $\epsilon_t = \frac{E}{\ln t}$ where E represents the energy level of the particle [5]. Given that this algorithm provably converges to the global minimum, a natural question is why other optimizers like AdaBelief are used for neural network optimization. Firstly, the algorithm is not guaranteed to converge to the global minimum unless the logarithmic cooling schedule is strictly followed, which means SA converges very slowly [2]. Secondly, the ultimate goal of neural network optimization is to minimize validation loss, which is not equivalent to minimizing training loss [34].

2.3 Improved Simulated Annealing

To mitigate the issue of slow convergence, Fang et al. proposed a modified version of simulated annealing, called improved simulated annealing (ISA) [9]. While still slower than optimizers used in the mainstream of deep learning today, ISA enjoys provably faster convergence to the global minimum than regular simulated annealing [9]. The authors achieved this effect by modifying 2.1 as follows:

$$\mathrm{d}Z_t = -\nabla U(Z_t)\,\mathrm{d}t + \sqrt{2(f((U(Z_t)-c)_+) + \epsilon_t)}\,\mathrm{d}B_t \tag{2.2}$$

where $f : \mathbb{R} \to \mathbb{R}^+$ is twice differentiable with $f(0) = f'(0) = f''(0) = 0$, $c \geq U_{\min}$ and the rest of the terms are as defined in 2.1 [5, 9]. The modification proposed by the paper is the addition of $2 \cdot f((U(Z_t) - c)_+)$ to the term representing random noise. Note that if $U(Z_t) - c \leq 0$, the term $f(U(Z_t) - c)_+ = 0$ and we have 2.1 exactly. Essentially, $f((U(Z_t) - c)_+)$ injects state-dependent noise into the optimization algorithm. f is referred to as the noise injection function. Suppose that for a loss function U at state Z_t, $U(Z_t) > c$. Then f is applied to the difference between

$U(Z_t)$ and c to determine an additional amount of noise to inject into the system. The additional state-dependent noise in ISA helps it to escape local minima or saddle points [5].

2.4 Improved Kinetic Simulated Annealing

State-dependent noise can be incorporated into other optimizers by modifying the loss function U. This implementation, termed landscape modification, was proposed by Choi in a research paper that sought to incorporate the state-dependent noise proposed by Fang et al. to improve kinetic simulated annealing with Langevin dynamics (KSA) [5].

2.4.1 Kinetic Simulated Annealing

Just as how simulated annealing is applied to the overdamped Langevin equation 2.1, kinetic simulated annealing (KSA) can be applied to the underdamped Langevin setting. Unlike in overdamped Langevin dynamics, underdamped Langevin dynamics model particle inertia, meaning that the particle's velocity does not immediately respond to the force ∇U. Given the following underdamped Langevin equations:

$$\begin{aligned} \mathrm{d}Z_t &= Y_t\,\mathrm{d}t \\ \mathrm{d}Y_t &= -\frac{1}{\epsilon_t}Y_t\,\mathrm{d}t - \nabla U(Z_t)\,\mathrm{d}t + \sqrt{2}\,\mathrm{d}B_t \end{aligned} \tag{2.3}$$

$(Z_t)_{t\geq 0}$ is a sequence representing the position of a particle, $(Y_t)_{t\geq 0}$ is a sequence representing the velocity of the particle and the rest of the variables are as defined in 2.1 [5]. Similarly to in 2.1, we can apply KSA as an optimization algorithm by appropriately defining a cooling schedule

$(\epsilon_t)_{t\geq 0}$ and treating U as a loss function. Moreover, KSA also converges to $U_{\min}$ in the long run [24].

2.4.2 Improved Kinetic Simulated Annealing

To recap, in 2.2, Fang et al. added state-dependent noise to the SA optimization algorithm. Inspired by this approach, Choi embedded this state-dependent noise to the KSA optimization algorithm and proved its improved convergence speed compared to KSA, terming this new algorithm improved kinetic simulated annealing (IKSA) [5]. IKSA is defined as follows:

$$\begin{aligned} \mathrm{d}Z_t &= Y_t \,\mathrm{d}t \\ \mathrm{d}Y_t &= -\frac{1}{\epsilon_t} Y_t \,\mathrm{d}t - \epsilon_t \nabla_z H_{\epsilon_t,c}(Z_t)\,\mathrm{d}t + \sqrt{2}\,\mathrm{d}B_t \end{aligned} \tag{2.4}$$

where each variable is defined in the same way as in 2.3 except for H, which is a transformation of the original loss function U [5]. In IKSA, the state-dependent noise is embedded into the loss function itself [5]. This is equivalent to modifying the loss landscape. $H_{\epsilon,c}$ is defined as follows:

$$H_{\epsilon,c}(z) := \int_{U_{\min}}^{U(z)} \frac{1}{f((u-c)_+)+\epsilon}\,\mathrm{d}u + \ln(f((U(z)-c)_+)+\epsilon) \tag{2.5}$$

where ϵ is a hyperparameter set by the user and f must follow the constraints set out in 2.2 [5]. The gradient of $H_{\epsilon,c}$ can be expressed as follows:

$$\nabla H_{\epsilon,c}(z) = \frac{1+f'((U(z)-c)_+)}{f((U(z)-c)_+)+\epsilon} \nabla U(z) \tag{2.6}$$

where ∇U is the gradient of U at z [5].

2.5 Landscape Modification in Deep Learning

Landscape modification injects state-dependent noise that helps an optimizer escape local minima [5]. Therefore, when we apply 2.6 to other optimizers, we can ignore $f'((U(z)-c)_+)$ as it is only necessary for the proof of improved convergence speed for IKSA in [5]. Thus, we can still inject state-dependent noise while expressing $\nabla H_{\epsilon,c}$ in a simpler manner, as follows:

$$\nabla H_{\epsilon,c}(z) = \frac{1}{f((U(z)-c)_+)+\epsilon}\nabla U(z) \tag{2.7}$$

This means that we can apply landscape modification to other optimizers by adding a step to modify the observed gradient before it is used for any computations. 2.7 gives us $H_{\epsilon,c}(z)$ as defined as follows:

$$H_{\epsilon,c}(z) := \int_{U_{\min}}^{U(z)} \frac{1}{f((u-c)_+)+\epsilon}\,\mathrm{d}u \tag{2.8}$$

Note that $H_{\epsilon,c}(z)$ now omits the logarithmic term from 2.5. We do so because the logarithmic term itself is only relevant to the proof of IKSA's convergence [5].

Injecting state-dependent noise can be thought of as smoothing, which intuitively makes the loss landscape easier to navigate. Figure 2.1 depicts this effect on a toy example, where the original loss landscape is $U(z)$, but the optimizer encounters the smoothed landscape $H(z)$. In regions where $U(z) > c$, $H(z)$ is noticeably smoother as a result of injecting state-dependent noise. In the context of deep learning, it has been traditionally held that landscape modification biases convergence towards lower training loss by making it easier for parameters to escape suboptimal local minima [26, 28]. Note that it is not guaranteed that lower training

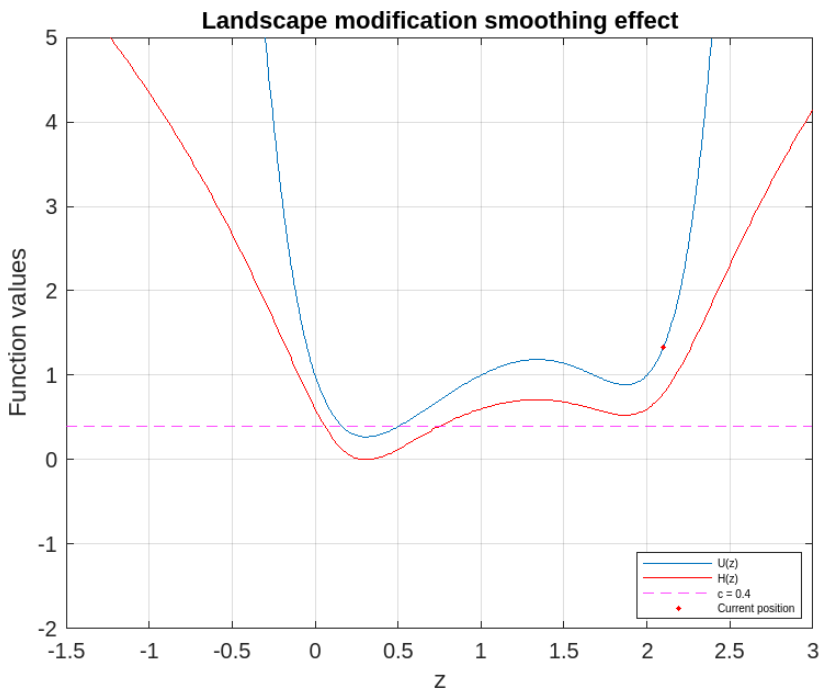


FIGURE 2.1: Effect of landscape modification on $U(z) = (z-1)^6 + 2z^2 - z^3$ with $c = 0.4$, $f(x) = x$ and $\epsilon = 1$.

loss corresponds to lower validation loss. Intuitively, consider the global minimum where a model has memorized the entire training set, and note that validation loss will certainly be high as the model has grossly overfit the training data. A challenge with accepting the traditional hypothesis is that landscape modification can also cause model parameters to escape favourable minima of low loss, leading them instead towards local minima with higher loss. It is true that by setting c as the running minimum loss and choosing f as a monotonic increasing function, one can tune landscape modification to smooth gradients more drastically at higher losses and minimally within low losses, thereby making it harder for parameters to ultimately converge to minima at higher losses. However, recent theoretical arguments suggest that the number of high-loss local minima diminishes exponentially with network size, indicating that suboptimal local minima in the large networks typically used in deep learning are rare [6]. Corroborating these results are recent empirical results indicating the rarity of encountering local minima during training [10]. The academic debate around this topic is ongoing, but if local minima with high losses are rare, then landscape modification will affect most minima similarly even with clever choice of hyperparameters. This would complicate the argument that landscape modification significantly biases convergence to minima with lower loss.

2.5.1 Hypothesis: Landscape Modification Biases Convergence Towards Flat Minima

In response to the problems with the traditional hypothesis, I propose an alternative hypothesis by which to understand the impact of landscape modification in deep learning. I argue that with appropriate hyperparameter selection, landscape modification biases parameter convergence towards flatter minima, which do not always correspond to lower training loss but generally correspond to better validation loss. I will proceed by outlining some definitions before providing justification for this hypothesis.

Definition 1 *The weight vector x_0 is a local minimizer if all eigenvalues of the Hessian matrix $\nabla^2 f(x_0)$ are positive [7].*

Intuitively, this means that a local minimum exists if the loss function around a minimizer is convex. Formally, sharpness is characterized by the magnitude of the eigenvalues of $\nabla^2 f(x_0)$ [17]. As sharpness is comparative, there is no universally agreed-upon threshold that defines what constitutes sharp or flat minima. In this paper, I adopt looser definitions of sharp and flat minima. These definitions are as follows:

Definition 2 *A flat minimum, denoted by $\overline{M}$, refers to a relatively large neighborhood in the weight space where each weight vector within this region yields similar losses on the loss function [17].*

Definition 3 *A sharp minimum, denoted by $\widehat{M}$, refers to a relatively small neighborhood in the weight space where small differences in weight vectors tend to yield relatively large differences in losses on the loss function [17].*

Now suppose that from 2.7 we set c to be the running minimum loss and $\epsilon = 1$. This yields the specific gradient update equation

$$\nabla H_{1,c}(z) = \frac{1}{f((U(z) - c)_+) + 1} \nabla U(z)$$

where $U(z)$ is the current loss and $\nabla U(z)$ is a vector of observed gradients. Now, suppose a set of parameters with the same initial weight vector are experiencing an identical update vector v_k within a flat and sharp minimum respectively. Further suppose the resulting loss is higher in both cases (i.e. the parameters are in the process of escaping the minima). Then, we can infer that $U_{flat}(z) < U_{sharp}(z)$ as the increase in loss for a flatter minimum must be lower in all directions for all points in the minimum (based on our informal definitions 2 and 3). Assuming that the running minimum loss c in both cases are equal, then for any continuous and monotonically increasing f with $f(0) = 0$,

$$\begin{aligned} f(U_{flat}(z) - c) &< f(U_{sharp}(z) - c) \\ \Rightarrow \frac{1}{f(U_{flat}(z) - c) + 1} &> \frac{1}{f(U_{sharp}(z) - c) + 1}. \end{aligned}$$

Given the conditions above, the observed gradient vector $\nabla U_{sharp}(z)$ will be reduced by a larger factor than $\nabla U_{flat}(z)$. With respect to a single dimension, the magnitude of the gradient $|g_t|$ encountered by a parameter escaping a sharp minimum is reduced by a larger factor than if it were escaping a flat minimum. Note also that as we progressively use steeper functions for f, the increasing disparity in the factor of gradient modification will mean that modified gradients for sharper minima converge to the modified gradients for flatter minima. For most optimization algorithms, this will mean that step sizes for both flat and sharp minima become similar as gradient smoothing increases. Concretely, in the AdaBelief algorithm outlined in 1.3.1, a smaller $(g_t - m_t)^2$ leads to a larger step size, where g_t is the current gradient and m_t is an exponential moving average of past gradients. Also note that the step is taken in the direction of $\widehat{m_t}$, a scalar multiple of m_t. Suppose that k_t is the modified gradient that will be used by the algorithm to make step updates instead of g_t. Now, if a parameter within a minimum is moving in the direction of escape and yielding higher loss, then the momentum m_t must be in the opposite direction to g_t and consequently k_t. This means that as stronger gradient smoothing is applied (i.e. $|k_t| \to 0$), step sizes across minima become larger and similar. Larger step sizes make it easier to escape any minimum, but particularly make it easier to escape sharp minima, which are generally narrower. Intuitively speaking, if step sizes across all minima progressively become similar, then the minima which parameters fail to escape and therefore converge to will be progressively flatter.

The Significance of Flatter Minima

Though some researchers present arguments that generalization ability is unrelated to the flatness of minima [8], the common view is that flatter minima correspond to better generalization [12, 17, 4]. In other words, parameters which converge to flatter minima tend to yield models which perform comparatively better on the validation set. The arguments supporting the widely-accepted belief are based on minimum description length (MDL) theory and Bayesian probability. The MDL principle is a theory of model selection which asserts that the best model is the one that describes the data using the least information [11]. Therefore, we would prefer models which require the least specification of parameters to generate their outputs. Flatter minima are regions which correspond to simpler models because the model's output requires less information from the parameters to perform well [17, 12]. Therefore, according to MDL theory, parameters converging to flat minima should yield better generalization [17, 12]. The Bayesian method of model selection is to choose the model M with maximum evidence, denoted as $P(D|M) = \int_\theta P(D|\theta)P(\theta|M)\,d\theta$ where θ are the possible parameter values for the model [23]. Given 2 models, one of simpler complexity M_{flat} and one of higher complexity M_{sharp}, $P(D|\theta)$ (intuitively understood as the fit of the data to the set of parameters) for a flat minima will be higher for almost all θ as all losses are similar. $P(\theta|M)$ is derived from our prior belief about the distribution of the parameters given the model. If we take a prior concentrated at the minimum expected loss and reasonably assume that the minimum loss of M_{flat} is not extremely high, then $P(D|M_{flat}) > P(D|M_{sharp})$ and the simpler model corresponding to selecting parameters in flatter minima is preferred [4]. Undergirding both model selection criteria, however, is the paradigm that simpler models best encapsulate reality [12, 23]. Thus, to achieve better generalization, we aim to find a simpler model that performs reasonably well on the training data. In deep learning, this corresponds to helping parameters converge to flatter minima. Given the positive association between flatness and generalization, my hypothesis would imply both that landscape modification lowers validation loss, and that increased gradient smoothing up to a reasonable level further lowers validation loss. I use 'reasonable level' because implementing excessively aggressive gradient smoothing will hinder convergence. For example, pick f such that all gradients tend to 0. Then step sizes will be large enough and step directions inconsistent enough that the optimizer will simply fail to converge to any minimum.

Caveats and Considerations

A consideration that the reader should note is that this argument is reliant on the assumption that the loss landscape is static. However, in almost all realistic training scenarios, the loss landscape is dynamic. To illustrate, a crucial training method known as dropout causes the loss landscape to change slightly in between training batches by removing a percentage of the parameters from the model for each training batch [31]. This means that a slightly different model is being trained over each batch of data [31], generating a different loss over a different set of parameters, implying that the loss landscape itself changes between iterations.

Another important consideration is that this argument has been derived from naïve assumptions about flat and sharp minima. In realistic loss landscapes, certain assumptions used in this argument may be overly simplistic. For example, the justification of $U_{flat}(z) < U_{sharp}(z)$ relies on a flatter minimum having lower increases in loss with respect to all parameters than a sharper minimum. In deep learning, minima are high-dimensional, and it is impractical to only define a minimum as flatter if it yields lower loss increases with respect to every dimension compared to a sharper minimum.

2.6 AdaBelief-LM Optimization Algorithm

Given the purported benefits of landscape modification, we seek to apply landscape modification to an existing adaptive optimization algorithm and test whether it improves this optimizer's performance. This section details the implementation of landscape modification within AdaBelief to create AdaBelief-LM. AdaBelief-LM was created by implementing 2.7 into AdaBelief's algorithm. For the reader's convenience, I restate 2.7 here:

$$\nabla H_{\epsilon,c}(z) = \frac{1}{f((U(z)-c)_+)+\epsilon}\nabla U(z) \tag{2.9}$$

To implement landscape modification into any optimizer, one takes the gradient the optimizer uses to make each update step, $\nabla U(z)$, and modifies it based on the current loss $U(z)$ along with some parameters ϵ and c. The rest of the algorithm then proceeds as normal. The Python code for my implementation of AdaBelief-LM can be perused in Appendix A.

2.6.1 Algorithm

AdaBelief-LM starts by randomly initializing a vector of weights θ_0 and setting $m_0 = 0$, $s_0 = 0$, $t = 0$. Then, for each subsequent time step t, given a gradient vector g_t already computed through backpropagation, the value c_t, the current loss u_t and a function f, AdaBelief-LM updates θ_t as follows:

1. Compute the modified gradient $k_t = \frac{g_t}{f((u_t - c_t)_+) + \epsilon}$.
2. Calculate $m_t = \beta_1 m_{t-1} + (1 - \beta_1) k_t$.
3. Calculate $s_t = \beta_2 s_{t-1} + (1 - \beta_2)(k_t - m_t)^2 + \epsilon$.
4. Calculate the unbiased estimates $\widehat{m_t} = \frac{m_t}{1-\beta_1}$, $\widehat{s_t} = \frac{s_t}{1-\beta_2}$.
5. Update the model's weights, $\theta_t = \theta_{t-1} - \frac{\alpha \widehat{m_t}}{\sqrt{\widehat{s_t}} + \epsilon}$.

Except for step 1, AdaBelief-LM proceeds exactly as AdaBelief.

Chapter 3

Language Modeling

3.1 The Task of Language Modeling

Language modeling is best understood as one of several tasks within the broad domain of language processing. In language modeling, one trains a model to predict the next word in a sequence given previous context [16]. Suppose we have the sequence of words $x_1, x_2, \ldots, x_n$. The task of language modeling is to train models that maximize

$$\mathcal{P}(x_{n+1} \mid x_1, x_2, \ldots, x_n)$$

where x_{n+1} is the true next word in the sequence [35]. Language models will output this probability within a vector of conditional probabilities, where each element of the vector corresponds to the conditional probability for a word in the model's vocabulary [16].

3.2 Language Model Evaluation Metrics

Based on our discussion above, one can think of language modeling as a classification task. For classification tasks, cross entropy loss is usually used a loss function [33]. In the context of language modeling, suppose we have an output vector of probabilities $\hat{v} \in \mathbb{R}^j$. Denote the model's predicted probability of the next word being the word indexed j as $\hat{y}_j$, and the actual probability of the next word being the word indexed j as y_j. y_j is either 0 or 1. Then, cross entropy loss is computed as follows [16]:

$$\mathbf{CE} = -\sum_j y_j \log \hat{y}_j \tag{3.1}$$

Indeed, for language model training, 3.1 is the loss function $\mathcal{L}$ described in 1.1 [16]. However, for historical reasons, researchers in language processing prefer to measure language model performance using perplexity instead of cross entropy loss [35]. Perplexity is closely related to cross entropy loss. Using 3.1, perplexity is computed as follows [15]:

$$\mathbf{PPL} = 2^{\mathbf{CE}} \tag{3.2}$$

where **CE** is the cross entropy loss. Lower perplexity is better [15].

3.3 LSTMs

Long short-term memory networks (LSTMs) are a type of recurrent neural network (RNN) particularly adept at modeling natural language sequences. To help the reader appreciate the reason why LSTMs are effective language models, I first introduce RNNs.

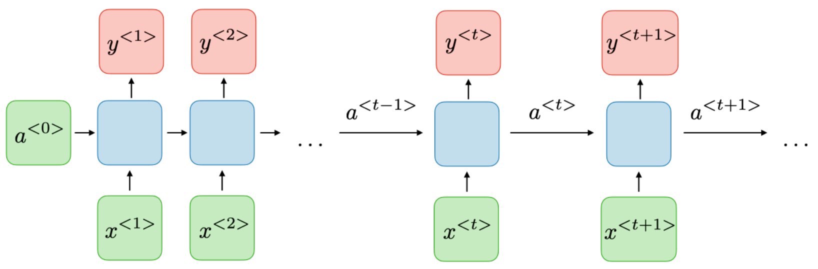


FIGURE 3.1: Simple RNN unit processing sequential inputs. The RNN unit is depicted as a blue square. [1]

3.3.1 Introduction to Recurrent Neural Networks

RNNs are a class of neural network architectures containing layers of RNN units [16]. RNN units are computational elements that process an input and a hidden state, best understood as the current value of the sequence and the sequence context respectively, to both generate an output and update the hidden state [16]. This process is depicted in Figure 3.1, which illustrates an RNN unit processing a sequence of inputs $x^{<t>}$ over several time steps. At each time step, the RNN unit considers both the value of the sequence at the current time step $x^{<t>}$ and the hidden state carried over from the previous time step $a^{<t-1>}$ to output $y^{<t>}$ and a new hidden state $a^{<t>}$. In formal terminology, $a^{<t>}$ is known as the hidden state for the RNN unit at time step t. The hidden state functions as a means to carry information forward to future time steps. At each time step, the hidden state $a^{<t>}$ is updated based on the input, allowing the RNN to adjust the information carried forward based on new information learned about the sequence from the input at the current time step.

In language modeling, one uses sequences of words as training data and feeds in one word at each time step [16]. At each time step, one compares the model's output, usually a vector of probabilities for each word in the model's vocabulary, against the true next word. The model is trained to minimize the error in predicting the next word in the training sequence with cross-entropy loss as the loss function [16]. Unfortunately, simple RNNs struggle to retain information much earlier in the sequence [16], which is critical for effective language modeling. For example, consider the following sequence:

Flowers in national parks around the country began to bloom.

At the final time step, our target word is "bloom". We want the model to consider both recent information ("began to") to output a verb, and distant information ("Flowers") to properly understand the context of the sequence. Unfortunately, however, the information carried forward in hidden states tends to be local [16]. Distant information is often lost by the time the model needs to rely on it to generate an accurate output [16]. The reason for this is that the architecture of simple RNN units is overly simplistic, and as such weights are being asked to perform the impractical dual role of both generating an accurate output for the current time step and determining relevant information to carry forward [16].

3.3.2 How LSTMs are Effective Language Models

LSTM units are able to generate relevant information for the current time step's output while still carrying forward relevant information for time steps far in the future. Figure 3.2 illustrates the structure of an LSTM unit. Unlike a simple RNN unit, an LSTM unit also processes a cell state $c^{<t>}$ that functions as long-term memory by carrying relevant information forward through many time steps [16]. The first part of an LSTM unit is known as the forget gate, where element-wise multiplication is used to determine which pieces of information in the cell-state are no longer

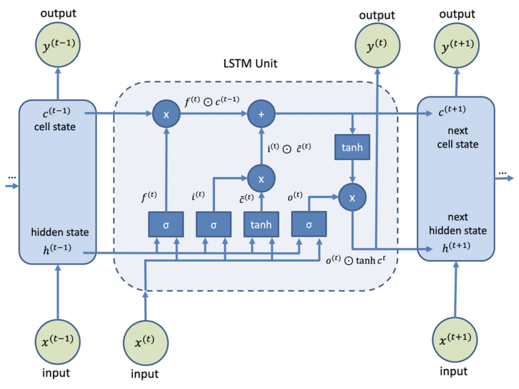


FIGURE 3.2: Basic structure of a LSTM unit connected to other LSTM units. [19]

required for future time steps [16]. The second part of the LSTM unit, the input gate, allows information about the previous hidden state and current input to be added to the cell state to be preserved as long-term memory [16]. Finally, the output gate extracts relevant information from the cell state to influence the current time step's output for the LSTM unit [16]. These architectural nuances make LSTMs highly effective for language modeling.

3.4 Penn TreeBank Dataset

The Penn Treebank dataset is an annotated English corpus comprised of almost 50000 sentences [25]. It is widely used to evaluate models trained to label words with tags that convey their grammatical functions or semantic roles [25]. This corpus contains language sequences from news articles, bulletins, research abstracts, and books [22]. Conveniently, there is a standard dataset split for training, validation and testing sets [25], making Penn TreeBank a common benchmark for researchers evaluating novel language model architectures and optimization algorithms.

Chapter 4

Empirical Results

4.1 Experimental Setup

The main aim of this project has been to investigate whether landscape modification can improve adaptive optimization for language modeling by yielding better validation set performance. A secondary aim has been to check whether the data contradict or align with my hypothesis in 2.5.1, which claims that increased gradient smoothing up to a reasonable level should result in further improved validation set performance. The following subsections detail the experimental setup used to address these two questions.

4.1.1 Models and Dataset

The model architectures used in this project were 1,2, and 3-layer LSTMs, with the same specific architecture (embedding size, number of LSTM units per layer, etc.) as in the AdaBelief paper. Penn TreeBank was used as the dataset for all experiments, the same dataset used by the AdaBelief authors to obtain their LSTM results.

4.1.2 AdaBelief-LM Optimizer Hyperparameter Setup

Specific Gradient Update

For all experiments, c was set to the running minimum training loss and ϵ was set to 1. We also always picked f such that $f(0) = 0$. This yielded the specific gradient update equation:

$$\nabla H_{1,c}(z) = \frac{1}{f((U(z) - c)_+) + 1} \nabla U(z). \tag{4.1}$$

At any point in the training process, if the current batch loss $U(z) < c$, then the input to f would be 0, and no landscape modification would occur as the gradient would be left unchanged.

Noise injection functions

Continuing our discussion from 2.5.1, f affects the level of gradient magnitude reduction, thus adjusting the difficulty of escaping minima. According to my hypothesis, as the disparity in gradient smoothing for flat and sharp minima increases by choosing a steeper f, parameters should converge to flatter minima, corresponding to the model having increasingly improved validation set performance. Unfortunately, as the parameter space is extremely high-dimensional, it is very difficult to visualize the specific minimum which parameters converge to to check if they are flat. However, we can check if validation performance indeed improves when we apply increased levels of gradient smoothing. To test the hypothesis, each LSTM architecture was trained with AdaBelief-LM 4 times, each time varying f using the following functions:

1. $f(x) = x^2$

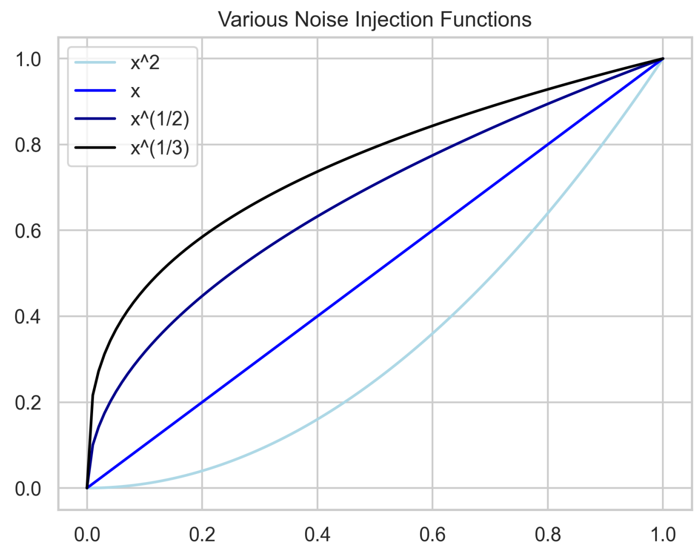


FIGURE 4.1: Varying f causes different levels of gradient smoothing. Gradients are adjusted by a factor of $\frac{1}{f(x)+1}$.

2. $f(x) = x$

3. $f(x) = \sqrt{x}$

4. $f(x) = \sqrt[3]{x}$

where $x = U(z) - c$ if $U(z) \geq c$, $x = 0$ otherwise. Figure 4.1 plots these functions. At this point, it is helpful to note that in the experiments, cross entropy training losses between every 100 batches after the initial stages of training mostly varied by significantly less than 0.2. From Figure 4.1, it is therefore apparent that x^2 not only applied the least gradient smoothing but also the least gradient smoothing disparity between flat and sharp minima, followed by x, $\sqrt{x}$ and $\sqrt[3]{x}$ in ascending order. These choices of f allow us to check if increased gradient smoothing disparity indeed leads to improved validation performance.

Other Hyperparameters

Other hyperparameters such as learning rate, learning rate scheduling and weight decay were kept identical with AdaBelief. Hyperparameter values were chosen in line with the AdaBelief authors' recommendations [36]. The goal of doing this was to reduce the risk of inadvertently disadvantaging AdaBelief by selectively applying suboptimal hyperparameters. The specific values can be viewed in Appendix B.

4.1.3 Experimental Procedure

The experimental procedure was slightly different depending on whether we were training using AdaBelief or AdaBelief-LM.

AdaBelief

1. Set all random seeds to 141.
2. Train the model over Penn TreeBank using AdaBelief for 200 epochs.
3. At the end of each epoch, record the training and validation perplexity.
4. Repeat steps 2-3 for random seeds 6 and 42.

AdaBelief-LM

1. Set the noise injection function to $f(x) = x^2$.
2. Set all random seeds to 141.
3. Train the model over Penn TreeBank using AdaBelief-LM for 200 epochs.

4. At the end of each epoch, record the training and validation perplexity.

5. Repeat steps 3-4 for random seeds 6 and 42.

6. Repeat steps 2-5 with noise injection functions x, $\sqrt{x}$, and $\sqrt[3]{x}$.

4.1.4 Technical Implementation

Experiments were run using Python, with Pytorch implementations of LSTMs, AdaBelief, and AdaBelief-LM. To implement AdaBelief-LM, we modified the source code for AdaBelief by installing the Pytorch implementation of AdaBelief through the authors' GitHub repository [36]. The code to train the models was adapted into Jupyter Notebook from the AdaBelief authors' LSTM training code [36], recreating the training pipeline used in the original paper.

4.2 LSTM over Penn TreeBank Results

4.2.1 Perplexity Curves

We present the perplexity curves per epoch for AdaBelief and AdaBelief-LM with different noise injection functions. These visualizations contain data from experiments run with random seed 141. As a note for the reader, the steep descents which occur for all perplexity curves at epochs 100 and 145 occur because of learning rate scheduling.

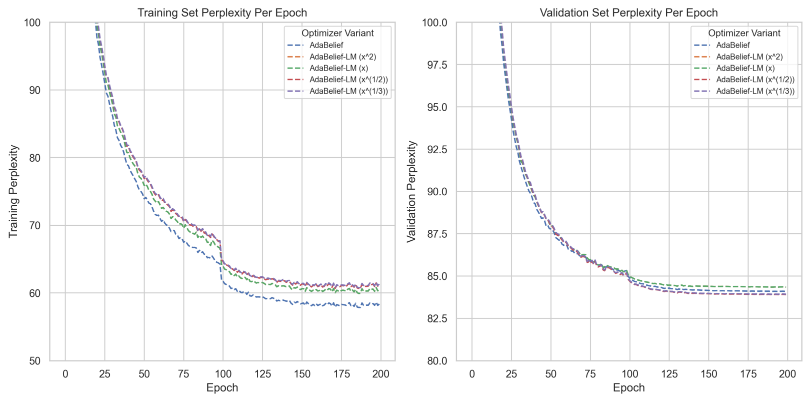


FIGURE 4.2: 1 Layer LSTM Perplexity Curves per Epoch for AdaBelief and AdaBelief-LM Variants

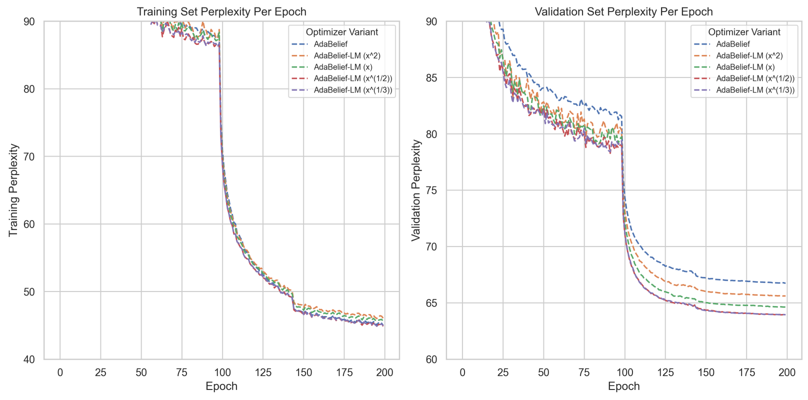


FIGURE 4.3: 2 Layer LSTM Perplexity Curves per Epoch for AdaBelief and AdaBelief-LM Variants

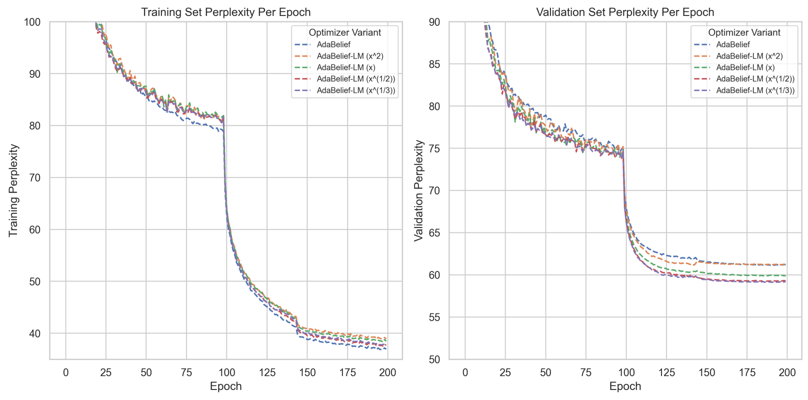


FIGURE 4.4: 3 Layer LSTM Perplexity Curves per Epoch for AdaBelief and AdaBelief-LM Variants

Across all models, AdaBelief-LM seems to generally achieve lower validation perplexity in spite of having higher training perplexity. This indicates both that AdaBelief-LM improves on AdaBelief and that landscape modification does not necessarily improve training loss, but discussion on this will be reserved for the next section where more rigorous results regarding minimum perplexities are presented.

From Figures 4.2 and 4.4, AdaBelief converges slightly faster with respect to training loss than AdaBelief-LM. This is an interesting result, possibly caused by how AdaBelief-LM explores the loss landscape by more easily climbing to areas of higher loss, slowing down its convergence. Figures 4.3 and 4.4 make this pattern clear, where the more stochastic looking perplexity curves indicate that AdaBelief-LM is allowing parameters to frequently explore areas of higher loss compared to AdaBelief. Hence, parameters optimized with AdaBelief-LM can explore the loss landscape more extensively. These observations are consistent with the proposition in 2.5 that landscape modification helps parameters to escape minima.

4.2.2 Minimum Perplexities

We investigate the average minimum perplexities over seeds 6, 42, and 141 as well as the variation in minimum perplexities to reduce the risk that the previous observation about AdaBelief generalizing poorly compared to AdaBelief-LM is a product of seed bias towards AdaBelief-LM.

1-Layer LSTMs

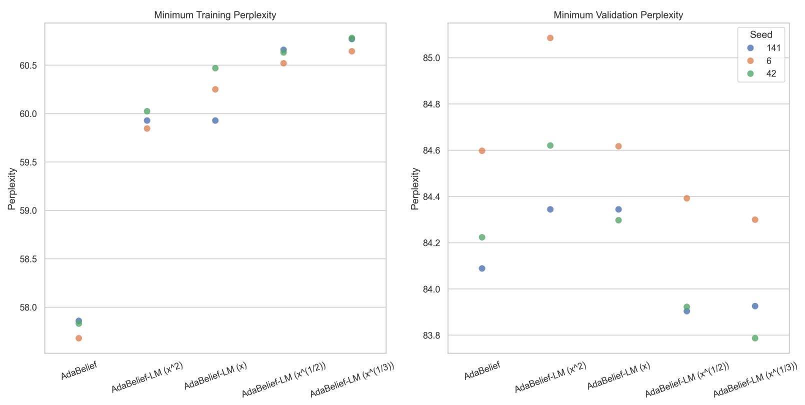


FIGURE 4.5: 1 Layer LSTM Minimum Perplexity Variation for AdaBelief and AdaBelief-LM Variants

Optimizer	f	**Min. Train PPL**	**Min. Valid PPL**
AdaBelief	NA	57.8	84.3
AdaBelief-LM	x^2	59.9	84.7
AdaBelief-LM	x	60.2	84.4
AdaBelief-LM	$\sqrt{x}$	60.6	84.1
AdaBelief-LM	$\sqrt[3]{x}$	60.7	84.0

TABLE 4.1: 1-Layer LSTM Average Minimum Perplexities for AdaBelief and AdaBelief-LM variations

Figure 4.5 shows AdaBelief-LM consistently converging to higher training perplexities as more aggressive gradient smoothing is implemented. This would be at odds with the traditional intuition that landscape modification in deep learning helps to attain lower training loss. However, it is still consistent with my hypothesis that landscape modification biases convergence to flatter minima, as flatter minima do not necessarily imply lower training loss.

The distributions of validation perplexities based on random seed as shown in Figure 4.5 overlap for all variations of AdaBelief-LM and AdaBelief. Since we only use 3 random seeds, it is difficult to ascertain which optimizer yields significantly better validation performance. However, the validation performance of AdaBelief-LM appears to be discernibly better when using $\sqrt[3]{x}$ as the noise function compared to x^2. This is consistent with my hypothesis that increasing gradient smoothing up to a reasonable level leads to better generalization.

2 Layer LSTMs

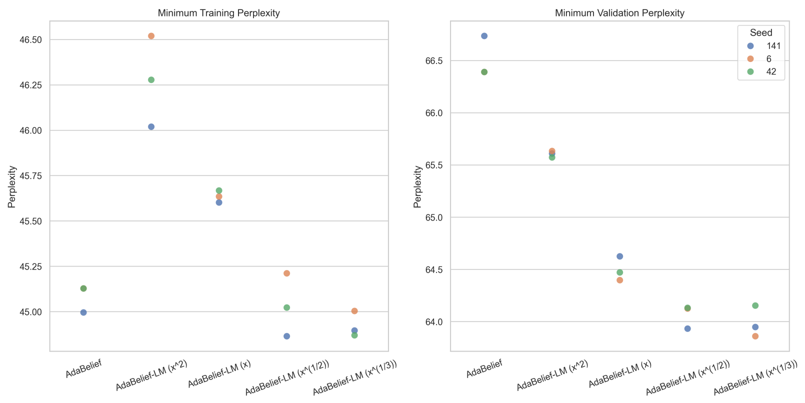


FIGURE 4.6: 2 Layer LSTM Minimum Perplexity Variation for AdaBelief and AdaBelief-LM Variants

Optimizer	f	Min. Train PPL	Min. Valid PPL
AdaBelief	NA	45.1	66.5
AdaBelief-LM	x^2	46.3	65.6
AdaBelief-LM	x	45.6	64.5
AdaBelief-LM	$\sqrt{x}$	45.0	64.1
AdaBelief-LM	$\sqrt[3]{x}$	44.9	64.0

TABLE 4.2: 2-Layer LSTM Average Minimum Perplexities for AdaBelief and AdaBelief-LM variations

Figure 4.6 shows that increased gradient smoothing does not necessarily lead to higher training loss, contrary to the 1-layer LSTM case in Figure 4.5. In fact, AdaBelief-LM is yields lower training loss on average than AdaBelief precisely when aggressive gradient smoothing functions $f(x) = \sqrt[3]{x}$ and $f(x) = \sqrt{x}$ are being used. A possible way to reconcile the differences in the training perplexity patterns for 1-layer and 2-layer LSTMs is to argue that flatter minima exist at higher levels of the loss landscape for 1-layer LSTMs, but exist at lower levels of the loss landscape for 2-layer LSTMs. Thus landscape modification converges to higher training loss in the 1-layer case, but lower loss in the 2-layer case. However, accepting this argument would rely on first accepting my hypothesis.

My hypothesis is further supported by the validation perplexities in 4.6, where we not only observe that all variations of AdaBelief-LM deliver lower validation perplexity but also that increasing gradient smoothing disparity results in progressively lower validation perplexities. The validation performance for $f(x) = \sqrt[3]{x}$ and $f(x) = \sqrt{x}$ is similar, indicating an upper limit for the effectiveness of aggressive gradient smoothing.

3 Layer LSTMs

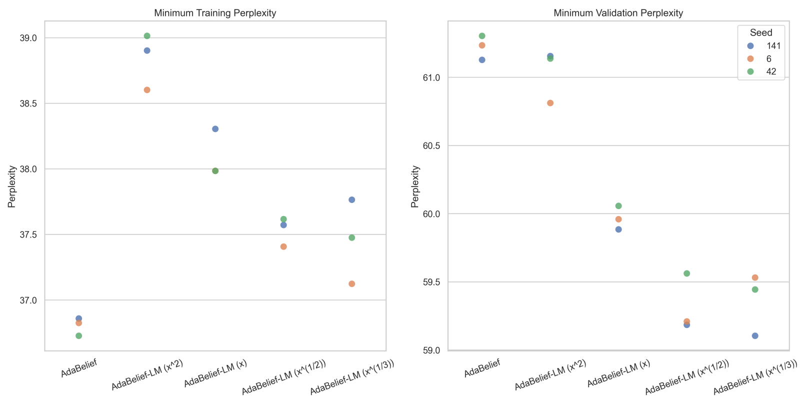


FIGURE 4.7: 3 Layer LSTM Minimum Perplexity Variation for AdaBelief and AdaBelief-LM Variants

Optimizer	f	Min. Train PPL	Min. Valid PPL
AdaBelief	NA	36.8	61.2
AdaBelief-LM	x^2	38.8	61.0
AdaBelief-LM	x	38.1	60.0
AdaBelief-LM	$\sqrt{x}$	37.5	59.3
AdaBelief-LM	$\sqrt[3]{x}$	37.5	59.4

TABLE 4.3: 3-Layer LSTM Average Minimum Perplexities for AdaBelief and AdaBelief-LM variations

The training perplexity results for 3-layer LSTMs further strengthen the assertion that AdaBelief-LM does not necessarily yield lower training loss than AdaBelief. Though increased gradient smoothing decreases training loss as in the 2-layer LSTM case, all models training with variants of AdaBelief-LM ultimately converge to higher training perplexity compared to any model trained with AdaBelief.

Once more, in line with my hypothesis, we observe that validation perplexity progressively decreases with the implementation of more aggressive gradient smoothing functions. However, we again notice an apparent upper limit to the effectiveness of increasing gradient smoothing. Using $f(x) = \sqrt[3]{x}$ with AdaBelief-LM does not appear to deliver better validation perplexity as opposed to $f(x) = \sqrt{x}$.

4.2.3 Summary

Sufficiently aggressive landscape modification allows AdaBelief-LM to outperform AdaBelief on the validation set. This highlights the effectiveness of landscape modification as a method to improve adaptive optimization for language modeling.

Furthermore, we observe a consistent pattern: As more aggressive gradient smoothing is implemented, validation loss generally decreases. This is consistent with my hypothesis in 2.5.1 that increased gradient smoothing should bias convergence to flatter minima that generalize better. However, it is difficult to show that these minima are indeed flatter. While it is commonly held that flat minima tend to generalize better, the converse is not necessarily true. The verification that these minima are flat would perhaps be the role of a formal proof regarding the convergence behaviour of adaptive optimizers with landscape modification, or empirical studies involving detailed loss landscape visualizations, both of which are complicated tasks.

The lack of a consistent pattern regarding training set performance makes it apparent that landscape modification does not always yield lower training loss. In fact, training losses for AdaBelief-LM are often higher than that of AdaBelief. These results cast doubt on the traditional intuition that landscape modification biases convergence to minima corresponding to lower loss.

Across the board, the validation perplexities attained when using $f(x) = \sqrt[3]{x}$ and $f(x) = \sqrt{x}$ are roughly similar, potentially indicating an upper limit for effective gradient smoothing disparity. As f is a hyperparameter, these results suggest that practical use of landscape modification should involve tuning f to implement progressively increased gradient smoothing disparity to the effective limit.

If the reader is interested in visualizing the performance difference between the trained models, Appendix C contains samples of generated text produced by different 2 and 3 layer LSTMs trained with AdaBelief and AdaBelief-LM.

Chapter 5

Conclusion

In this project, we investigated the effectiveness of landscape modification as a means to improve adaptive optimization for language modeling. To that end, we developed a new optimizer, AdaBelief-LM, as an extension of AdaBelief, and compared its effectiveness to AdaBelief in terms of training LSTMs that generalize well in the domain of language modeling. Empirical results show that AdaBelief-LM generally outperforms AdaBelief, especially for the more complex (2,3 layer) LSTM networks. This extends existing research about the effectiveness of landscape modification into the area of language modeling and a new class of neural network architectures, and shows that landscape modification can further improve adaptive optimizers which are already achieving impressive performance [36]. Furthermore, AdaBelief-LM may be of interest to machine learning practitioners due to its improved validation performance, and is now available to the broader community [27].

Moreover, research into deep learning optimization raises difficulties with applying the traditional argument that landscape modification biases convergence to lower training loss to the domain of deep learning. To that end, I provide some intuition in 2.5.1 for a new hypothesis that landscape modification in deep learning biases convergence to flatter minima, which results in better generalization. Though we are unable to visualize the flatness of individual minima in this project, we observe that increased gradient smoothing up to a certain level results in better generalization, which is in line with my hypothesis.

To bolster the current empirical results, future work in this area can investigate whether AdaBelief-LM also outperforms AdaBelief when applied to train different modern architectures also used for language modeling, such as GRUs or Transformers. Moreover, one can also apply the adaptive optimizers with landscape modification developed by Todea [29] to the same task of training LSTMs for language modeling to see if landscape modification improves validation performance in language modeling for more optimizers.

Moreover, though the empirical results are consistent with my hypothesis, we have not directly verified that the minima found by AdaBelief-LM are flatter. This is a challenging task, and would probably be the role of future work producing a proof regarding the convergence behaviour of adaptive optimizers with landscape modification, or an empirical study to visualize the loss landscape, following the filter normalization approach suggested by Li et al. in [20].

Bibliography

[1] Afshine Amidi and Shervine Amidi. *Recurrent Neural Networks Cheatsheet*. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks. Accessed: 2024-04-04. 2018.

[2] Oleksandr Borysenko and Maksym Byshkin. "CoolMomentum: a method for stochastic optimization by Langevin dynamics with simulated annealing". In: *Sci Rep* 11 (2021). DOI: 10.1038/s41598-021-90144-3. URL: https://doi.org/10.1038/s41598-021-90144-3.

[3] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. *Optimization Methods for Large-Scale Machine Learning*. 2018. arXiv: 1606.04838 [stat.ML].

[4] Pratik Chaudhari et al. *Entropy-SGD: Biasing Gradient Descent Into Wide Valleys*. 2017. arXiv: 1611.01838 [cs.LG].

[5] Michael C. H. Choi. *On the convergence of an improved and adaptive kinetic simulated annealing*. 2020. arXiv: 2009.00195 [math.PR].

[6] Anna Choromanska et al. *The Loss Surfaces of Multilayer Networks*. 2015. arXiv: 1412.0233 [cs.LG].

[7] Department of Mathematics at UTSA. *Maxima and Minima Problems*. https://mathresearch.utsa.edu/wiki/index.php?title=Maxima_and_Minima_Problems. Accessed: 2024-04-04. 2021.

[8] Laurent Dinh et al. *Sharp Minima Can Generalize For Deep Nets*. 2017. arXiv: 1703.04933 [cs.LG].

[9] Haitao Fang, Minping Qian, and Guanglu Gong. "An improved annealing method and its large-time behavior". In: *Stochastic Processes and their Applications* 71.1 (1997), pp. 55–74. ISSN: 0304-4149. DOI: https://doi.org/10.1016/S0304-4149(97)00069-0. URL: https://www.sciencedirect.com/science/article/pii/S0304414997000690.

[10] Ian J. Goodfellow, Oriol Vinyals, and Andrew M. Saxe. *Qualitatively characterizing neural network optimization problems*. 2015. arXiv: 1412.6544 [cs.NE].

[11] Peter Grünwald and Teemu Roos. "Minimum description length revisited". In: *International Journal of Mathematics for Industry* 11.01 (Dec. 2019). ISSN: 2661-3344. DOI: 10.1142/s2661335219300018. URL: http://dx.doi.org/10.1142/S2661335219300018.

[12] Sepp Hochreiter and Jürgen Schmidhuber. "Flat Minima". In: *Neural Computation* 9.1 (Jan. 1997), pp. 1–42. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.1.1. eprint: https://direct.mit.edu/neco/article-pdf/9/1/1/813385/neco.1997.9.1.1.pdf. URL: https://doi.org/10.1162/neco.1997.9.1.1.

[13] IBM. *Deep Learning*. Accessed: 2023-11-12. 2023. URL: https://www.ibm.com/topics/deep-learning.

[14] Naveen James. *Simulated Annealing Algorithm Explained from Scratch*. Machine Learning Plus. Accessed: 2023-11-12. URL: https://www.

machinelearningplus.com/machine-learning/simulated-annealing-algorithm-explained-from-scratch-python/.

[15] Daniel Jurafsky and James H. Martin. *N-Gram Language Models*. https://web.stanford.edu/~jurafsky/slp3/3.pdf. Chapter 3 of the 3rd edition of "Speech and Language Processing", Accessed: 2024-04-04. In progress.

[16] Daniel Jurafsky and James H. Martin. *RNNs and LSTMs*. https://web.stanford.edu/~jurafsky/slp3/9.pdf. Chapter 9 of the 3rd edition of "Speech and Language Processing", Accessed: 2024-04-04. In progress.

[17] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. 2017. arXiv: 1609.04836 [cs.LG].

[18] Kiprono Elijah Koech. *Cross-Entropy Loss Function*. Towards Data Science. Accessed: 2023-11-12. 2020. URL: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e.

[19] Aggelos Lazaris and Viktor K. Prasanna. “An LSTM Framework for Software-Defined Measurement”. In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 855–869. DOI: 10.1109/TNSM.2020.3040157.

[20] Hao Li et al. *Visualizing the Loss Landscape of Neural Nets*. 2018. arXiv: 1712.09913 [cs.LG].

[21] Liyuan Liu et al. *On the Variance of the Adaptive Learning Rate and Beyond*. 2021. arXiv: 1908.03265 [cs.LG].

[22] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a large annotated corpus of English: the penn treebank". In: *Comput. Linguist.* 19.2 (June 1993), pp. 313–330. ISSN: 0891-2017.

[23] Tom Minka. *Bayesian Model Selection*. `https://alumni.media.mit.edu/~tpminka/statlearn/demo/`. Accessed: 2024-04-04. 2005.

[24] Pierre Monmarché. *Hypocoercivity in metastable settings and kinetic simulated annealing*. 2017. arXiv: `1502.07263 [math.PR]`.

[25] *Penn Treebank*. Papers with Code. Accessed: 2023-11-12. URL: `https://paperswithcode.com/dataset/penn-treebank`.

[26] Stefan-Cristian Roata. "Landscape Modification Meets Deep Learning: Visualizing the Loss Landscape of Neural Networks". Capstone Project, Yale-NUS College. 2023.

[27] Andrew Siow. *Improved Adaptive Optimization for Language Modeling*. `https://github.com/andrewswj/Improved-Adaptive-Optimization-Language-Modeling`. Accessed: 2024-04-04. 2024.

[28] Ioana Todea. "Landscape Modification in Machine Learning Optimization". Capstone Project, Yale-NUS College. 2022.

[29] Ioana Todea. *LandscapeModification*. `https://github.com/IoanaTodea22/LandscapeModification`. Accessed: 2024-04-04. 2024.

[30] Ashia C. Wilson et al. *The Marginal Value of Adaptive Gradient Methods in Machine Learning*. 2018. arXiv: `1705.08292 [stat.ML]`.

[31] Aston Zhang et al. *Dropout*. Dive into Deep Learning. Accessed: 2024-04-04. 2021. URL: `https://d2l.ai/chapter_multilayer-perceptrons/dropout.html`.

[32] Aston Zhang et al. *Introduction*. Dive into Deep Learning. Accessed: 2023-11-12. 2021. URL: https://d2l.ai/chapter_introduction/index.html.

[33] Aston Zhang et al. *Linear Neural Networks for Classification*. Dive into Deep Learning. Accessed: 2024-04-04. 2021. URL: https://d2l.ai/chapter_linear-classification/softmax-regression.html#classification.

[34] Aston Zhang et al. *Optimization and Deep Learning*. Dive into Deep Learning. Accessed: 2023-11-12. 2021. URL: https://d2l.ai/chapter_optimization/optimization-intro.html#goal-of-optimization.

[35] Aston Zhang et al. *Recurrent Neural Networks*. Dive into Deep Learning. Accessed: 2024-04-04. 2021. URL: https://d2l.ai/chapter_recurrent-neural-networks/language-model.html.

[36] Juntang Zhuang. *Adabelief-Optimizer*. Commit: 2855178. 2022. URL: https://github.com/juntang-zhuang/Adabelief-Optimizer.

[37] Juntang Zhuang et al. *AdaBelief Optimizer: Adapting Stepsizes by the Belief in Observed Gradients*. 2020. arXiv: 2010.07468 [cs.LG].

Appendix A

AdaBelief-LM

To create AdaBelief-LM, we modified the Pytorch implementation of AdaBelief and saved it as a .py file. As the source code is relatively lengthy, for convenience we have only shown below the function which was modified to implement landscape modification.

```
def step(self, running_loss, c, closure=None):
    """Performs a single optimization step.
    Arguments:
        closure (callable, optional): A closure that reevaluates the
             model
            and returns the loss.
    """
    loss = None
    if closure is not None:
        loss = closure()

    for group in self.param_groups:
        for p in group['params']:
            if p.grad is None:
                continue
```

```
grad = p.grad.data
new_grad = grad / (self.function(torch.maximum(torch.
    zeros_like(grad, device=p.data.device), (running_loss
     - c) * torch.ones_like(grad, device=p.data.device)))
     + self.eps_iksa)

p.grad.data = new_grad # Replace the original gradient
    with the modified one

# cast data type
half_precision = False
if p.data.dtype == torch.float16:
    half_precision = True
    p.data = p.data.float()
    p.grad = p.grad.float()

grad = p.grad.data
if grad.is_sparse:
    raise RuntimeError(
        'AdaBelief-LM does not support sparse gradients,
            please consider SparseAdam instead')
amsgrad = group['amsgrad']

state = self.state[p]

beta1, beta2 = group['betas']

# State initialization
```

```
if len(state) == 0:
    state['step'] = 0
    # Exponential moving average of gradient values
    state['exp_avg'] = torch.zeros_like(p.data,
        memory_format=torch.preserve_format) \
        if version_higher else torch.zeros_like(p.data)
    # Exponential moving average of squared gradient
        values
    state['exp_avg_var'] = torch.zeros_like(p.data,
        memory_format=torch.preserve_format) \
        if version_higher else torch.zeros_like(p.data)
    if amsgrad:
        # Maintains max of all exp. moving avg. of sq.
            grad. values
        state['max_exp_avg_var'] = torch.zeros_like(p.
            data,memory_format=torch.preserve_format) \
            if version_higher else torch.zeros_like(p.data
                )

# perform weight decay, check if decoupled weight decay
if self.weight_decouple:
    if not self.fixed_decay:
        p.data.mul_(1.0 - group['lr'] * group['
            weight_decay'])
    else:
        p.data.mul_(1.0 - group['weight_decay'])
else:
    if group['weight_decay'] != 0:
        grad.add_(p.data, alpha=group['weight_decay'])

# get current state variable
exp_avg, exp_avg_var = state['exp_avg'], state['
    exp_avg_var']

state['step'] += 1
bias_correction1 = 1 - beta1 ** state['step']
bias_correction2 = 1 - beta2 ** state['step']

# Update first and second moment running average
exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
grad_residual = grad - exp_avg
exp_avg_var.mul_(beta2).addcmul_( grad_residual,
    grad_residual, value=1 - beta2)

if amsgrad:
    max_exp_avg_var = state['max_exp_avg_var']
    # Maintains the maximum of all 2nd moment running avg
        . till now
    torch.max(max_exp_avg_var, exp_avg_var.add_(group['
        eps']), out=max_exp_avg_var)

    # Use the max. for normalizing running avg. of
        gradient
    denom = (max_exp_avg_var.sqrt() / math.sqrt(
        bias_correction2)).add_(group['eps'])
else:
    denom = (exp_avg_var.add_(group['eps']).sqrt() / math
        .sqrt(bias_correction2)).add_(group['eps'])

# update
if not self.rectify:
    # Default update
    step_size = group['lr'] / bias_correction1
    p.data.addcdiv_( exp_avg, denom, value=-step_size)

else: # Rectified update, forked from RAdam
    buffered = group['buffer'][int(state['step'] % 10)]
    if state['step'] == buffered[0]:
        N_sma, step_size = buffered[1], buffered[2]
    else:
        buffered[0] = state['step']
        beta2_t = beta2 ** state['step']
        N_sma_max = 2 / (1 - beta2) - 1
        N_sma = N_sma_max - 2 * state['step'] * beta2_t /
             (1 - beta2_t)
        buffered[1] = N_sma

        # more conservative since it's an approximated
            value
        if N_sma >= 5:
            step_size = math.sqrt(
                (1 - beta2_t) * (N_sma - 4) / (N_sma_max -
                     4) * (N_sma - 2) / N_sma * N_sma_max /
                     (
                        N_sma_max - 2)) / (1 - beta1 **
                          state['step'])
            elif self.degenerated_to_sgd:
                step_size = 1.0 / (1 - beta1 ** state['step'])
            else:
                step_size = -1
            buffered[2] = step_size

        if N_sma >= 5:
            denom = exp_avg_var.sqrt().add_(group['eps'])
            p.data.addcdiv_(exp_avg, denom, value=-step_size
                * group['lr'])
        elif step_size > 0:
            p.data.add_( exp_avg, alpha=-step_size * group['
                lr'])

    if half_precision:
        p.data = p.data.half()
        p.grad = p.grad.half()

return loss
```

Appendix B

Hyperparameters

The following hyperparameters were used for the experiments. Where possible, hyperparameter values were kept identical with the AdaBelief authors' recommendations and training pipeline in [36].

B.1 Model Hyperparameters

LSTM Layers	**Emb. Size**	**Hid. Unit-s/Layer**	**Input Dropout**	**Emb. Dropout**	**Hid. Dropout**	**Gen. Dropout**	**Wt. Dropout**
1	400	1150	0.4	0.1	0.25	0.4	0.5
2	400	1150	0.4	0.1	0.25	0.4	0.5
3	400	1150	0.4	0.1	0.25	0.4	0.5

TABLE B.1: LSTM model hyperparameters used in this study's experiments.

B.2 AdaBelief Hyperparameters

Learning rate scheduling was implemented at the 100th and 145th epochs, shrinking the current learning rate by a factor of 10 at each time.

Model	Learning Rate	Betas	Epsilon	Wt. Decay
1-layer LSTM	0.001	(0.9, 0.999)	1e-16	1.2e-6
2-layer LSTM	0.01	(0.9, 0.999)	1e-12	1.2e-6
3-layer LSTM	0.01	(0.9, 0.999)	1e-12	1.2e-6

TABLE B.2: AdaBelief hyperparameters used in this study's experiments.

B.3 AdaBelief-LM Hyperparameters

Model	LR	Betas	Epsilon	Wt. Decay	ϵ	c	f
1-layer LSTM	0.001	(0.9, 0.999)	1e-16	1.2e-6	1	Run. min.	See 4.1.2
2-layer LSTM	0.01	(0.9, 0.999)	1e-12	1.2e-6	1	Run. min.	See 4.1.2
3-layer LSTM	0.01	(0.9, 0.999)	1e-12	1.2e-6	1	Run. min.	See 4.1.2

TABLE B.3: AdaBelief-LM hyperparameters used in this study's experiments.

Appendix C

2,3 Layer LSTM Generated Sequences

This section contains a visualization of the relative performance between selected models trained in this paper on a text generation task. Selected models were fed a starting sequence of words as a seed text, and asked to generate up to 30 tokens (words) to complete the sequence. At each time step, the token with the highest probability was added to the sequence, and the new sequence was passed to the model to generate the next token. As it is difficult to measure the relative performance of generated text, the decision as to which model performs better is left to the reader.

For conciseness, the results below display only the coherent section of each model's generated text. Most of the time, after a certain number of time steps the generated text would devolve into incoherent repetitions of the same token. Note that the <unk> token refers to a word outside the PennTreeBank dataset vocabulary, the N token refers to a number, and the <eos> token refers to a full stop. The Base model corresponds to a model trained with AdaBelief. LM-$\sqrt{x}$ and LM-$\sqrt[3]{x}$ correspond to a model trained with AdaBelief-LM with $\sqrt{x}$ and $\sqrt[3]{x}$ as noise injection functions respectively.

TABLE C.1: 2-Layer LSTM: 30 token sequence generation

Seed Text	**Base**	**LM-$\sqrt{x}$**	**LM-$\sqrt[3]{x}$**
makes some executives nervous <eos> last year the research	makes some executives nervous <eos> last year the research team was <unk> by the <unk> of the <unk>	makes some executives nervous <eos> last year the research department said it was <unk> by the <unk> of the <unk>	makes some executives nervous <eos> last year the research firm had a $ N million loss in the third quarter <eos> the
those days are now over he believes <eos> competition from	those days are over now he believes <eos> competition from the	those days are over now he believes <eos> competition from the <unk> of the <unk>	those days are over now he believes <eos> competition from the <unk> of the <unk>
why would he ever	why would he ever be <unk> <eos> the <unk> of the <unk>	why would he ever have to be a <unk> <eos> the <unk> of the <unk>	why would he ever be able to <unk> the <unk> of the <unk> <eos> the <unk> of the <unk>
neither the environmental protection agency nor the	neither the environmental protection agency nor the <unk> of the <unk> <eos> the <unk> of the <unk>	neither the environmental protection agency nor the <unk> of the <unk> <eos> the	neither the environmental protection agency nor the <unk> of the <unk> <eos> the

TABLE C.1: 2-Layer LSTM: 30 token sequence generation

Seed Text	**Base**	**LM-$\sqrt{x}$**	**LM-$\sqrt[3]{x}$**
futures prices had been headed up on expectations that world oil demand will continue to be strong <eos> the organization of petroleum	futures prices had been headed up on expectations that world oil demand will continue to be strong <eos> the organization of petroleum exporting countries is <unk> by the <unk> of the <unk> of the <unk> <unk> <eos> the <unk> of the <unk>	futures prices had been headed up on expectations that world oil demand will continue to be strong <eos> the organization of petroleum exporting countries is expected to be <unk> by the end of the year <eos> the <unk> of the u.s. is expected to be <unk> by the end of the year	futures prices had been headed up on expectations that world oil demand will continue to be strong <eos> the organization of petroleum exporting countries is expected to be <unk> by the end of the year <eos> the <unk> of the <unk>
the group had an operating profit	the group had an operating profit of $ N million <eos> the company said it will take a charge of $ N million to $ N million in the first quarter of N <eos> the company	the group had an operating profit of $ N million <eos> the	the group had an operating profit of $ N million <eos> the company said it will take a $ N million charge against the sale of its <unk> unit <eos> the company said it will sell
that fund was put together by blackstone group a new york investment bank <eos> the latest two funds were assembled jointly by	that fund was put together by blackstone group a new york investment bank <eos> the latest two funds were assembled jointly by the <unk> of the <unk> <unk> and <unk> of the <unk> <unk> <eos> the	that fund was put together by blackstone group a new york investment bank <eos> the latest two funds were assembled jointly by the <unk> group <eos> the	that fund was put together by blackstone group a new york investment bank <eos> the latest two funds were assembled jointly by the <unk> group <eos> the

TABLE C.2: 3-Layer LSTM: 30 token sequence generation

Seed Text	**Base**	**LM-$\sqrt{x}$**	**LM-$\sqrt[3]{x}$**
makes some executives nervous <eos> last year the research	makes some executives nervous <eos> last year the research team had been <unk> by the <unk> of the <unk>	makes some executives nervous <eos> last year the research and development company said the <unk> <unk> was <unk> by the <unk> of the <unk> <unk> <eos> the	makes some executives nervous <eos> last year the research and development concern said it will take a $ N million charge against the <unk> of the <unk>
those days are over now he believes <eos> competition from	those days are over now he believes <eos> competition from the <unk> of the <unk>	those days are over now he believes <eos> competition from the <unk> of the <unk> is <unk> <eos> the <unk> of the <unk>	those days are over now he believes <eos> competition from the <unk> of the <unk>
why would he ever	why would he ever have to be <unk> <eos> the <unk> of the <unk>	why would he ever be able to <unk> the <unk> of the <unk> <eos> the	why would he ever <unk> the <unk> of the <unk> <eos> the
neither the environmental protection agency nor the	neither the environmental protection agency nor the <unk> of the <unk> <unk> <unk> <eos> the	neither the environmental protection agency nor the <unk> of the <unk> <eos> the	neither the environmental protection agency nor the state of new york is the <unk> of the <unk> <eos> the <unk> of the

TABLE C.2: 3-Layer LSTM: 30 token sequence generation

Seed Text	**Base**	**LM-$\sqrt{x}$**	**LM-$\sqrt[3]{x}$**
futures prices had been headed up on expectations that world oil demand will continue to be strong <eos> the organization of petroleum	futures prices had been headed up on expectations that world oil demand will continue to be strong <eos> the organization of petroleum exporting countries is a <unk> of the <unk>	futures prices had been headed up on expectations that world oil demand will continue to be strong <eos> the organization of petroleum exporting countries is expected to report a loss of N million barrels a day for the first six months of N <eos> the	futures prices had been headed up on expectations that world oil demand will continue to be strong <eos> the organization of petroleum exporting countries is <unk> the <unk> of the <unk> of the <unk> <unk> <eos> the <unk> of the <unk> <unk> is a <unk>
the group had an operating profit	the group had an operating profit of $ N million <eos> the company said it had a loss of $ N million or N cents a share for the third quarter <eos> the company said it	the group had an operating profit of $ N million <eos> the company said it was n't aware of the <unk> of the company 's <unk> <unk> <eos> the company said it is n't aware of	the group had an operating profit of $ N million <eos> the
blue arrow was able to pull off the $	blue arrow was able to pull off the $ N million in <unk>	blue arrow was able to pull off the $ N million in <unk> <unk> <eos> the	blue arrow was able to pull off the $ N million in cash and stock <eos> the company said it would n't disclose the value of the transaction <eos> the company said it would n't disclose the terms of

TABLE C.2: 3-Layer LSTM: 30 token sequence generation

Seed Text	**Base**	**LM-$\sqrt{x}$**	**LM-$\sqrt[3]{x}$**
that fund was put together by blackstone group a new york investment bank <eos> the latest two funds were assembled jointly by	that fund was put together by blackstone group a new york investment bank <eos> the latest two funds were assembled jointly by the new york investment bank <eos> the fund 's portfolio is a <unk> <unk> and <unk> <unk> <eos> the	that fund was put together by blackstone group a new york investment bank <eos> the latest two funds were assembled jointly by the securities and exchange commission <eos> the	that fund was put together by blackstone group a new york investment bank <eos> the latest two funds were assembled jointly by the <unk> group of <unk> <unk> <unk> & <unk> a <unk> firm <eos> the