MSc Thesis Topic Proposal: An Empirical Study of *Relaxed Activation Identity*'s Effects on First-Order Optimization of Neural Networks

Dennis Alexander Mertens Velasquez (student i.d.: i6206990)

1 Introduction

Suppose we have n training examples $Z_0 = [z_0^0, \ldots, z_0^{n-1}]$ denoting inputs and $Z_T = [z_T^0, \ldots, z_T^{n-1}]$ targets, and a feed-forward differentiable circuit of depth T as in figure 1. The circuit is composed of T layers; each a non-linear differentiable[1] function—denoted f_t, where $0 \leq t \leq T-1$. The input and output of each layer are denoted z_t and z_{t+1}, respectively. Additionally, each layer has a corresponding set of parameters θ_t. Including the fact that we have multiple training examples, without any loss of generality, the relation between input and output is described by

$$z_{t+1}^i = f_t(z_t^i, \theta_t), \tag{1}$$

for the ith training example.

When applying backpropagation (BP) and gradient descent (GD), we optimize parameters $\Theta = [\theta_0, \ldots, \theta_{T-1}]$ by nudging them according to their gradients.[2] In such a setting, the gradients of θ_t depend on the gradients and activations of layers $t+1, \ldots, T-1$. As we will see in section 2, the nonlinearities through which gradients have to flow through, can be destructive. The objective behind this proposal is to transform the sequential dependencies such that gradients flow through less destructive nonlinearities. I think we can achieve this by "relaxing" equation 1.

[1] We understand by *differentiable* a function whose first-order gradient is defined everywhere.

[2] To be clear: BP is responsible for propagating the errors through the circuit, and GD is responsible for adjusting the parameters accordingly.

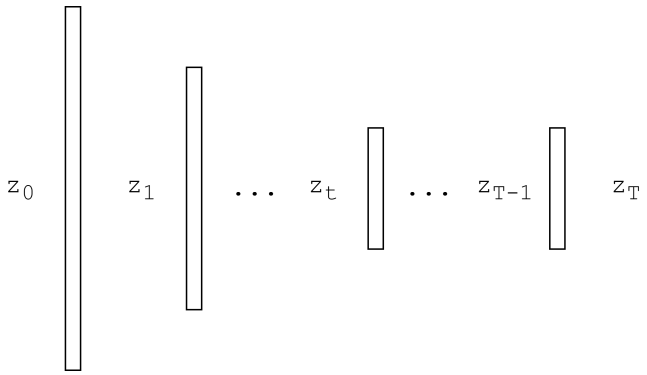

Figure 1: A feed-forward differentiable circuit of depth T; z_0 denotes the input, $z_1, \ldots, z_{T-1}$ denote the hidden activations, and z_T denotes the output. Each layer $t \in \{0, \ldots, T-1\}$ has parameters θ_t and activation function f_t.

2 Motivation

The vanishing and exploding gradients problem, as described by Sepp Hochreiter, is a phenomenon that affects gradient norms as a result of the nonlinearities in a differentiable circuit applied in sequence along the same gradient's path, leading to exceedingly small weight updates in earlier layers (i.e., vanishing gradients) or exceedingly large weight updates in earlier layers (i.e., exploding gradients) [4]. A direct consequence is that the difficulty of training a circuit with BP and GD increases as a function of its depth[3].

There exist ways to mitigate the adverse effects of depth, such as the gating mechanism in highway nets [11], skip connections in ResNet [3], and dense connections in dense neural nets [6]. All of them provide alternative paths for the gradients to flow backward whilst circumventing nonlinearities. However, tricks like these are not general enough, as they cannot be applied to every situation. For instance, circuits with backward connections, such as recurrent neural nets, require a separate set of tricks. In particular, we have two popular examples: long-short term memory [5] and gated recurrent units [2].

[3]We understand by *depth* the number of elementary operations mapping from input to output. For example, in the case of feed-forward neural nets, this translates to the number of layers.

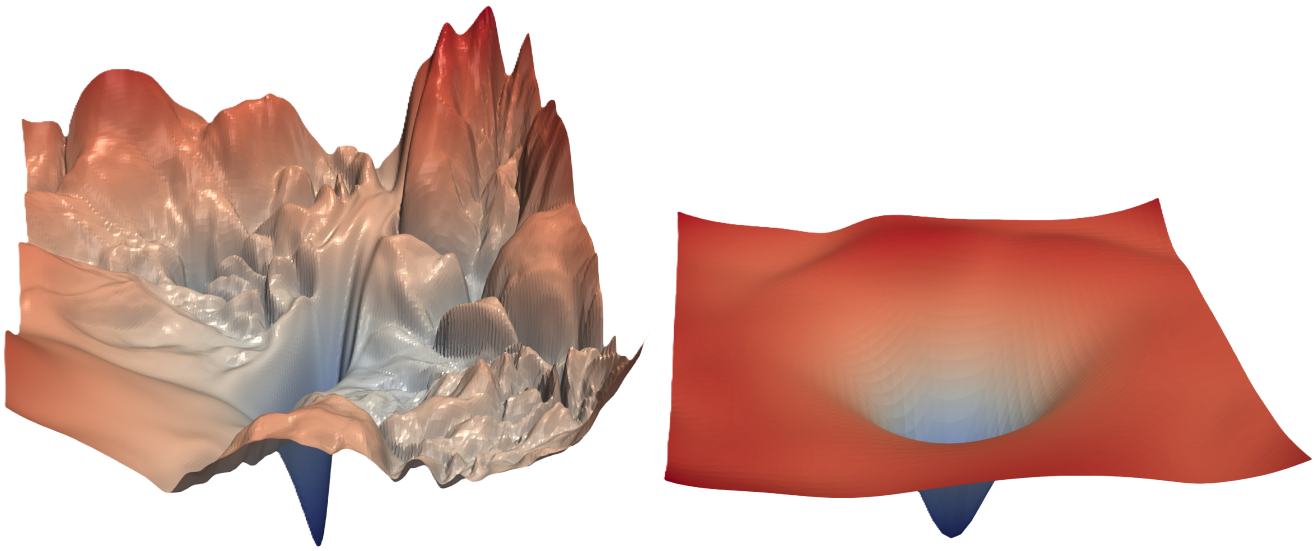

Figure 2: **Left:** Low-dimensional projection of the loss landscape of a ResNet *without* skip connections. **Right:** Low-dimensional projection of the loss landscape of a ResNet *with* skip connections. Both images were taken from [7].

In mixed-integer linear programming, there is a family of methods often known as *lifting*. When a problem cannot be directly modeled with a convex boundary and/or a linear objective, we can introduce auxiliary decision variables that map the problem to a higher-dimensional space. Oftentimes, this moves the problem to a space that can be modeled with a convex boundary and/or a linear objective. For example, when we have a constraint that can become active or inactive in response to a choice, we can introduce a decision variable to represent said choice and formulate a constraint that tightens or loosens in response to the added decision variable. I suspect we can also "lift" the problem of optimizing a differentiable circuit—though admittedly less elegant.

If we can reformulate a circuit such that gradients flow through a different path backward than forward, we could prevent the vanishing and exploding gradients problem altogether. In section 3, an approach is discussed, and in section 4, preliminary results are provided as evidence that there is merit to researching it. Furthermore, if the approach works, there is a real chance that we can implement a compiler atop existing auto-differentiation frameworks such as PyTorch [9], Jax [1], or TensorFlow [8]—automatically enabling pre-existing architectures to work with the proposed approach.

3 Approach

Extending the formula introduced in section 1, equation 1, let z_t^i denote the target activation of the tth layer given the ith training example, and $\hat{z}_t^i$ its estimate. For $t \in \{0, T\}$, the targets are given. For $t \in \{1, \ldots, T-1\}$, assume auxiliary trainable parameters $Z_t = [z_t^0, \ldots, z_t^{n-1}]$ and define the hidden activations as

$$\hat{z}_{t+1}^i = f_t(z_t^i, \theta_t). \tag{2}$$

Usually, any activation $\hat{z}_{t+1}^i$ is defined in terms of the preceding $\hat{z}_t^i$, hence the sequential dependence. In contrast, the dependence is broken by introducing an auxiliary z_t^i. Suppose we train such a

system by only minimizing the output layer's error. In that case, it will not generalize because only the last layer will learn to map z_{T-1}^i to z_T^i whilst every other part of the circuit will settle for arbitrary configurations. To fix this, we must enforce that

$$\hat{z}_t^i \approx z_t^i, \forall t \in \{1, \dots, T-1\}, \forall i \in \{0, \dots, n-1\}. \tag{3}$$

During training, some loss function $\mathcal{L}(Z_T, \hat{Z}_T) = \sum_{i=0}^{n-1} L_y(z_T^i, \hat{z}_T^i)$ is always required, where L_y measures the output error.[4] By extending the loss to include terms that regulate the hidden activations, the condition in equation 3 is enforced. That is,

$$\mathcal{L}(Z_T, \hat{Z}_T) = \sum_{i=0}^{n-1} L_y(z_T^i, \hat{z}_T^i) + \sum_{t=1}^{T-1} \sum_{i=0}^{n-1} L_h(z_t^i, \hat{z}_t^i), \tag{4}$$

where L_h measures the hidden error.

I call this approach *relaxed activation identity* (RAI) because it is based on the idea of—in a sense—relaxing the identity of an activation. Without relaxation, we have $z_t^i = \hat{z}_t^i$, where the distinction between the prediction $\hat{z}_t^i$ and the target z_t^i is redundant. With relaxation, we eventually have $z_t^i \approx \hat{z}_t^i$. Yet, at times, these two may differ. Hence, the activation's identity is "relaxed".

4 Preliminary Evidence

In a shallow circuit, both the standard application of BP and GD, and RAI are expected to succeed. Figure 3 shows the results of the shallow case, where a shallow MLP with 4 layers and ReLu hidden activations was trained on the Moons dataset from Scikit Learn [10]. In a deep circuit, the standard application of BP and GD is expected to fail without tricks such as skip connections, whereas RAI is expected to succeed. Figure 4 shows the results of the deep case, where a deep MLP with 101 layers and Sigmoid activations was trained on the same dataset.[5]

[4] Usually, loss functions are augmented with a regularization term, but it is irrelevant in this context, and thus not included.

[5] To reproduce these experiments, go to the GitHub repository at https://github.com/damertensv/rai and find the notebooks `rai-demo.ipynb` and `rai-grads.ipynb`.

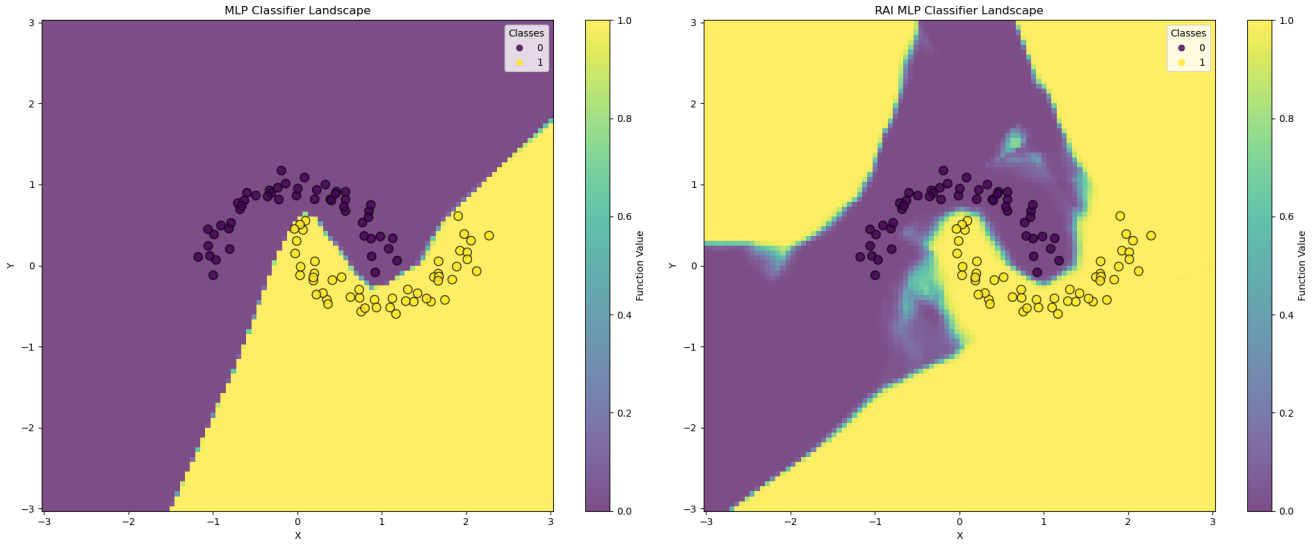

Figure 3: **Left:** Decision landscape of a shallow MLP with relu activations. **Right:** Decision landscape of a shallow RAI MLP with relu activations.

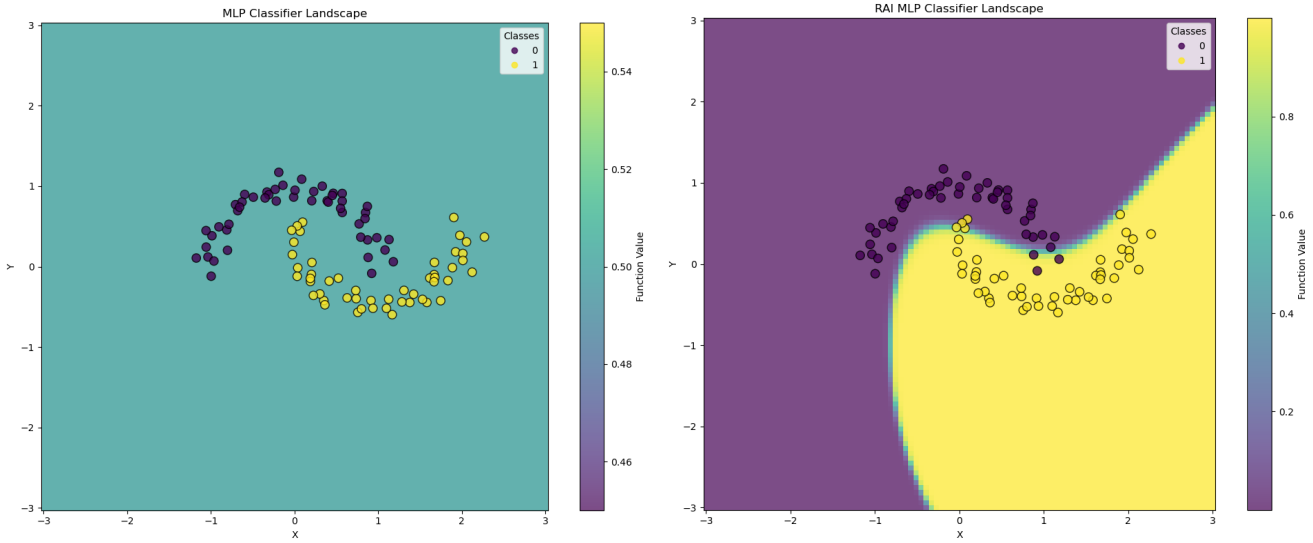

Figure 4: **Left:** Decision landscape of a deep MLP with sigmoid activations. **Right:** Decision landscape of a deep RAI MLP with sigmoid activations.

5 Research Questions

We see from figures 3 and 4 that—at first glance—the RAI MLP can generalize like the standard MLP whilst retaining the ability to train in contexts where the standard MLP cannot. However, we also see qualitative differences between the two models' decision boundaries in the shallow context. This could be an indication that the RAI MLP needs additional regularization.

Furthermore, while not visible in the plots, re-running the experiments multiple times revealed that the standard MLP tends to settle for similar decision boundaries across runs, whereas the RAI MLP does not. Hence, it is my opinion that the RAI MLP has weaker biases during optimization.

Since this *could* imply the RAI MLP will struggle to generalize in higher dimensions, it prompts the following two research questions:

- Does RAI generalize as well as standard BP and GD across regression and classification benchmarks?
- What regularization methods exist that could aid RAI to retain standard BP and GD's ability to generalize across regression and classification benchmarks?

Besides regularization during training, we should study RAI's sensitivity to the choice of initial parameters in a model. If RAI is more sensitive than standard BP and GD, that could explain the variance in the shape of the decision boundary across runs. Hence, the research question:

- Is RAI more sensitive than standard BP and GD to the choice of initial model parameters?

Finally, almost by definition, RAI is incompatible with batched, mini-batched, and stochastic gradient descent methods. While unimportant in toy datasets, the aforementioned methods are necessary in real-world applications due to the otherwise unrealistic requirement of loading all the data into working memory at once and going through all training examples for every iteration. Hence, the research question:

- How can we enable RAI to be compatible with batched, mini-batched, and stochastic gradient descent methods?

References

[1] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/jax-ml/jax.

[2] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL]. URL: https://arxiv.org/abs/1406.1078.

[3] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[4] Sepp Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen*. German. Unpublished diploma thesis. Analyzes vanishing and exploding gradients in recurrent neural networks. 1991.

[5] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[6] Gao Huang et al. *Densely Connected Convolutional Networks*. 2018. arXiv: 1608.06993 [cs.CV]. URL: https://arxiv.org/abs/1608.06993.

[7] Hao Li et al. *Visualizing the Loss Landscape of Neural Nets*. 2018. arXiv: 1712.09913 [cs.LG]. URL: https://arxiv.org/abs/1712.09913.

[8] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[9] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[10] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[11] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. *Highway Networks*. 2015. arXiv: 1505.00387 [cs.LG]. URL: https://arxiv.org/abs/1505.00387.