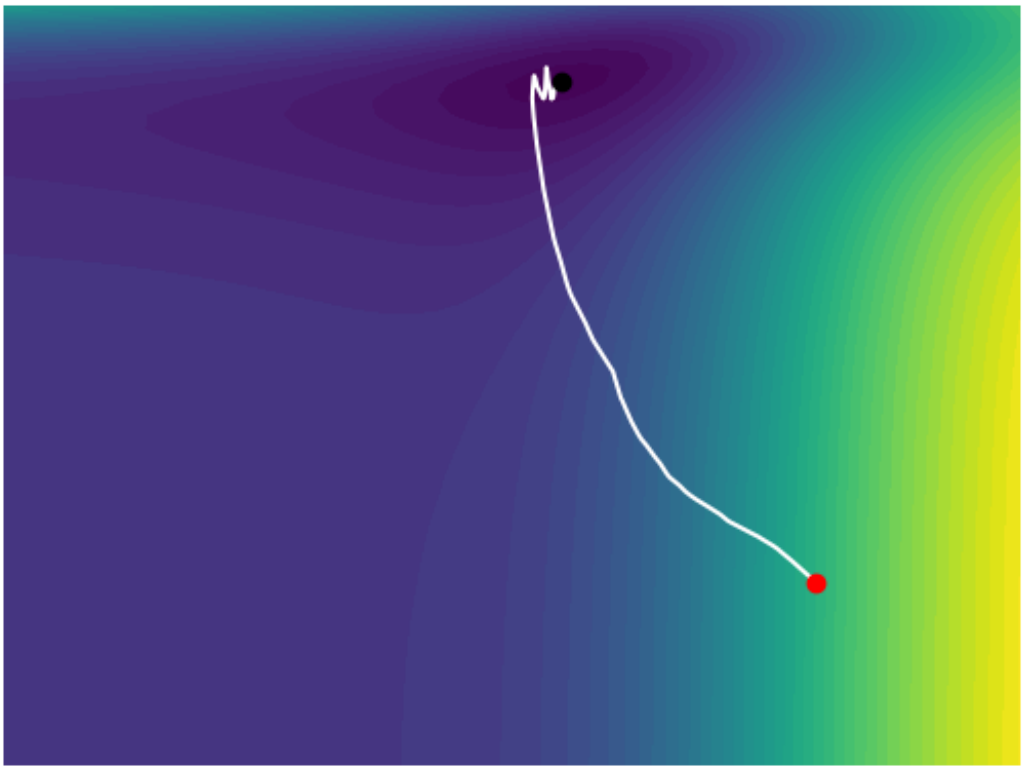
Visualizing optimization of weights in a neural network

Axel Orrhede

December 2024

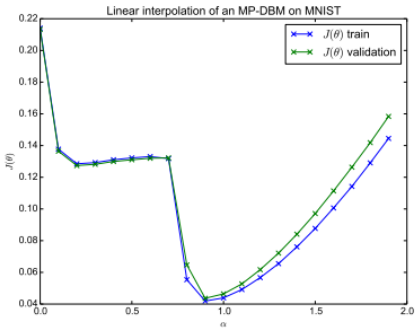


1 Introduction

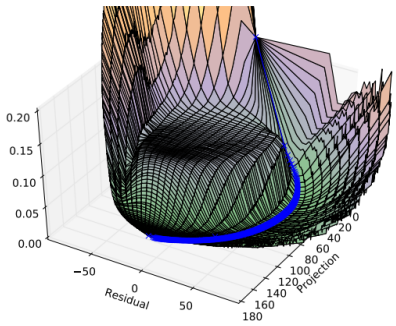
Visualization is the key to understanding, but in the context of Machine learning (ML), it is often hard to find insightful visual representations of the algorithms. Still, due to the complex training process and the large size of the most practically applicable models, there is often very limited intuition for them and their training process. In this paper, we will study the optimization of two of the parameters of a smaller neural network to find the best optimization algorithm. Honing in on just the two parameters will allow us to visualize the training process effectively. The hope is that some qualitative findings from this paper are generalizable to the ML field. The focus will be on SGD, RMSprop, Adam and AdamW, and our approach will visually compare the paths of these algorithms in the loss landscape and allow us to make some claims about their applicability and greatness.

2 Visualizing weights

Machine learning, and especially neural networks, is an immensely popular field, and many before us have tried to explain the training process visually. This is difficult, simply because there are so many of parameters, and our imagination struggles with visualizing anything past the 3 dimensions we live in. Goodfellow et al. [1] solves this problem by creating lines $\theta(\alpha) = (1 - \alpha)\theta + \alpha\theta'$ through the parameter space and plotting the loss function along it. This leaves us with plots as figure 1a. They then proceeded to create a plane through the weights in a similar fashion and plot the 2D landscape of the loss, which is seen in figure 1b.



(a) Loss function along a line in the parameter space



(b) Loss function along a 2d slice in the parameter space. The blue line is the route of an SGD optimizer.

Figure 1: Both figures taken from [1], and show loss functions for the same neural network.

Although having a lot of potential to give insight, having to pick a line or a 2d slice

when there are more than 2 weights implies that our analysis misses out on most of the parameter space. We have to project our optimizer's path onto the chosen visualization. In this paper, we are instead going to show the complete optimization path, but for only two of the weights. This way, we get to explore optimizations that are complete in the sense that there is no projection, but all other weights will have to be frozen so that the visualization is meaningful.

3 The setup

The entire project used the following dataset and model architecture. The model was reinitialized every time the activation function was changed.

3.1 The toy dataset

For practicality, a toy dataset was created with 1000 points following equation 1 where $x_1, x_2, x_3 \in [-2, 2]$ evenly spaced.

$$y = x_1 x_2 - x_3^2 - x_2 + 0.1 * \mathcal{N}(0, 1) \tag{1}$$

The gaussian is term meant to simulate noise in the data and was added to allow us to meaningfully use stochastic methods for optimization.

3.2 The neural network

A small neural network f_w with 3 hidden layers of 3 nodes each was initialized. It has 30 weights and 10 biases, activation functions were applied to all nodes in the hidden layers.

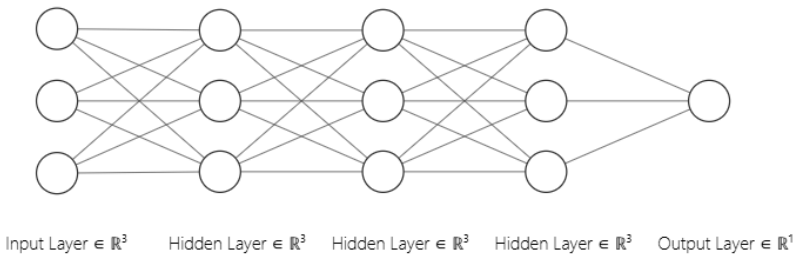


Figure 2: The neural network architecture.

The function we are trying to model is fairly simple, with only first and second-degree non-linearities. However, because of its smaller size, the network will still struggle with learning the behavior of this toy dataset, and this is the optimization we are going to examine further.

4 Optimizers

4.1 Stochastic gradient descent (SGD)

When randomly sampling a batch size n points randomly from the 1000 points generated, one can calculate a mean squared error according to.

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_n - f_w(x_1^n, x_2^n, x_3^n))^2, \tag{2}$$

If we then calculate the gradient for our weights w and subtract that times a learning rate α from the current weights, we can update them to improve the model.

$$w^{(t+1)} := w^{(t)} - \alpha\nabla_w L^{(t)} \tag{3}$$

Doing this iteratively to minimize the loss function is the algorithm called Stochastic gradient descent.

4.2 Root Mean Square Propagation (RMSprop)

One might think that deciding on a learning rate in advance is difficult and inflexible. You can therefore implement an adaptive learning rate v for each parameter. This way, the update length will depend on the squared gradient and also the squared gradients of previous steps.

$$v_w^{(t+1)} := \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2 \qquad [3] \tag{4}$$

Where β_2 is the forgetting factor with value 0.99 and $v_w^{(0)}$ is initialized as 0. Then, we can update the weights according to the following:

$$w^{(t+1)} := w^{(t)} - \frac{\alpha}{\sqrt{v_w^{(t)}} + \epsilon}\nabla_w L^{(t)} \qquad [3] \tag{5}$$

Where ϵ is used to prevent division by 0.

4.3 Adam

Trying to improve even further, we define an additional variable for each parameter to introduce momentum.

$$m_w^{(t+1)} := \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)} \qquad [3] \tag{6}$$

We set $\beta_1 = 0.9$, while v is calculated the same as for RMSprop but with $\beta_2 = 0.999$. We also initialize $m_w^{(0)} = 0$, and do the following bias-correction, allowing better steps even though m and v have yet to reach their potential. [3]

$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-\beta_1^t} \qquad \hat{v}_w = \frac{v_w^{(t+1)}}{1-\beta_2^t} \tag{7}$$

In the end, you update the parameters according to the following formula.

$$w^{(t+1)} := w^{(t)} - \alpha \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon} \qquad [3] \tag{8}$$

4.4 AdamW

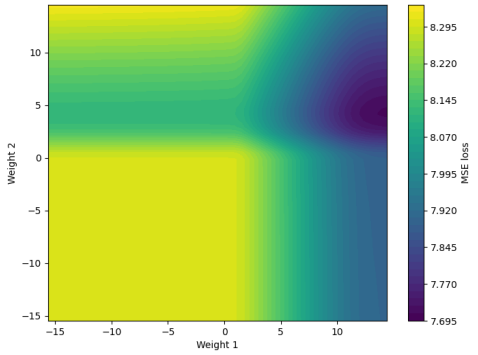
This algorithm is the same as Adam but with the addition of a weight decay step $w := w - \lambda w$ after the gradients are calculated but before the weights are updated. λ is usually a small positive number, in our case e^{-2}. The idea is that this counteracts weights slowly becoming really large since this is often a sign of overfitting.

5 Training and visualizing

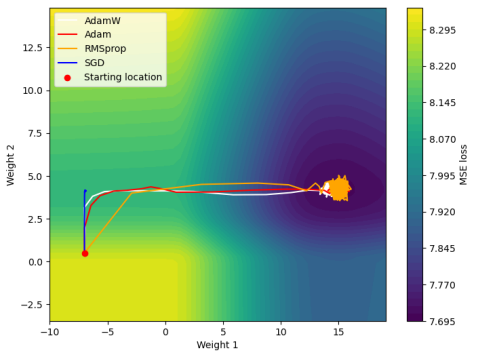
Activation functions are used to bring non-linearities into neural networks, and the choice of activation function greatly impacts model performance, which is why we are going to handle them separately. In this segment, we are studying two randomly chosen weights out of the 9 in between the first and second hidden layer (see figure 2) of some trained and untrained models and see how some arbitrarily initialized optimizers with $\alpha = 0.1$ behave. Loss landscapes are calculated by equation 2 with a batch containing the entire dataset when the weights are set to the corresponding coordinates in the plot.

5.1 Rectified linear unit (ReLU)

ReLU ($max(0, x)$) is the simplest of the popular activation functions because it just clips the negative inputs to 0. Looking at two arbitrary weights inbetween the first and second hidden layer allows us to plot the loss landscape for the untrained network and get figure 3a.



(a) Loss landscape for two randomly chosen weights in an untrained network with the ReLU activation function.



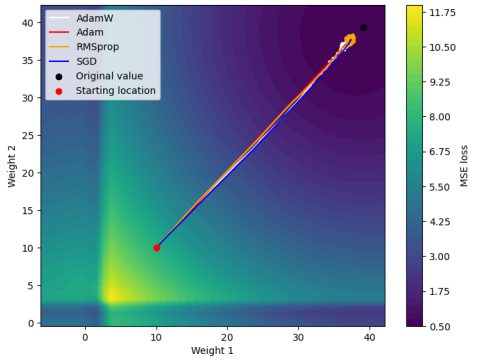
(b) Optimization in the loss landscape from figure 3a with different optimizers, optimization paths have one datapoint for every epoch.

Figure 3: Untrained ReLU-activated network

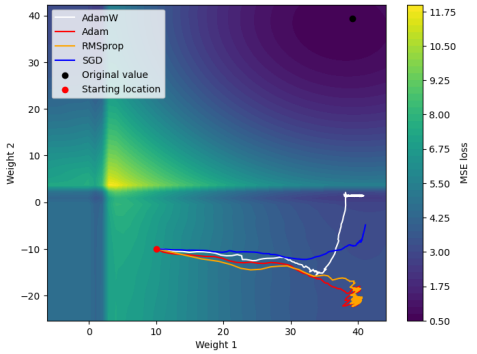
The figure shows a typical behavior of loss landscapes for weights in ReLU-activated networks. The clipping of negative entries means that the gradient will forever be 0 if the weight becomes too negative since the node will in practice always have a negative number before the activation. This results in the large flat patches seen above and is known as the Dying ReLU problem [2]. One could try to work around this by reinitializing the weights to "dead" nodes during training. However, it could also be advantageous to have sparser models, as it could make interpreting the network easier.

Trying to optimize the model with all but the two weights frozen is performed in figure 3b, and we note that 3 out of 4 optimizers find the global minima. However, the minimum for these two weights has an MSE of 7.695. To improve the model further one would have to iteratively move on to other weights, optimize them, and then continue until convergence (or not). This process, even though very visualizable, has horrible efficiency. Instead, we are going to train the model with no weights frozen for 1000 epochs of batch size 64 with the Adam optimizer with a learning rate of 0.01.

Looking back at the loss landscape for the same weights as before and initializing optimizers in two places results in figure 4.



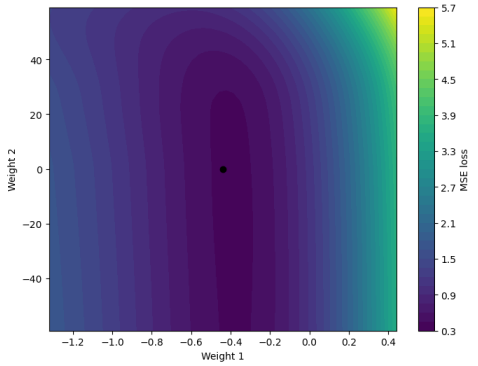
(a) Optimization with good initialization and a batch size of 64, optimization paths have one datapoint for every epoch.



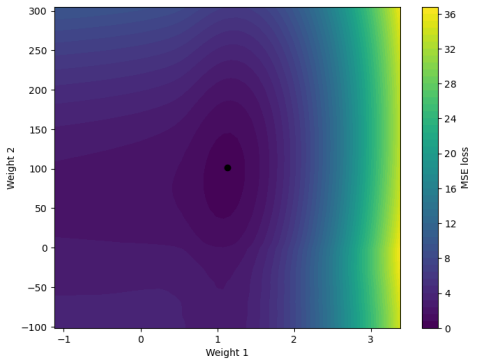
(b) Optimization with poor initialization and a batch size of 64, optimization paths have one datapoint for every epoch.

Figure 4: Trained ReLU-activated network

The landscape has now changed, but the characteristics of the ReLU still remain. The loss landscape is now fairly easy to optimize in, unless the initialization is too poor, as seen in 4b. Some more loss landscapes from other weights of the same model are shown in figure 5.



(a) Loss landscape for two randomly chosen weights.

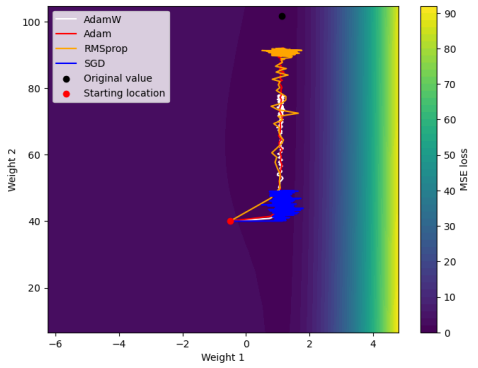


(b) Loss landscape for two more randomly chosen weights.

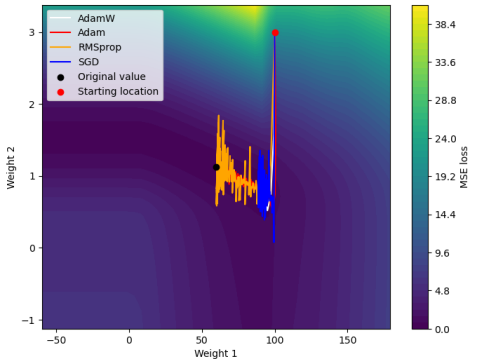
Figure 5: Trained ReLU-activated network

Many weights have convex or near-convex landscapes, but notice the scales in the figure. In 5a, we see that "weight 2" does not have an impact on the loss until it gets large enough to ruin the model. This is likely because the node "weight 2" leads to is already clipped to 0, and this weight needs to become very large to overcome this.

Optimization in landscapes with varying scales is difficult.



(a) Optimization paths over the weights from 5b with a batch size of 1, optimization paths have one datapoint for every epoch



(b) Optimization paths over two other weights with a batch size of 1, optimization paths have one datapoint for every epoch

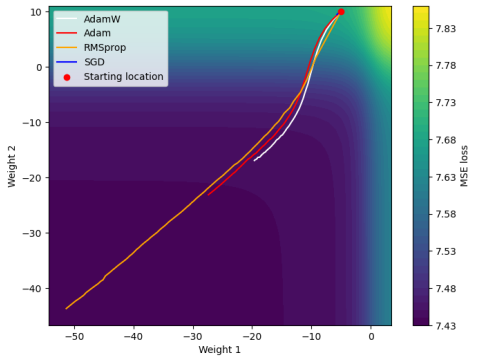
Figure 6: Trained ReLU-activated network

In figure 6 it is apparent that the adaptive learning rate from the Adam optimizers and RMSprop helps out a lot. Once the optimizers hit the optimum in the high-gradient direction, they are bumped around by the gradient noise and only slowly pushed towards the optimum in the large-scale direction. This slowly speeds up the Adams and RMSprop, but not SGD, which is why my patience ran out before SGD could make it to the optimum.

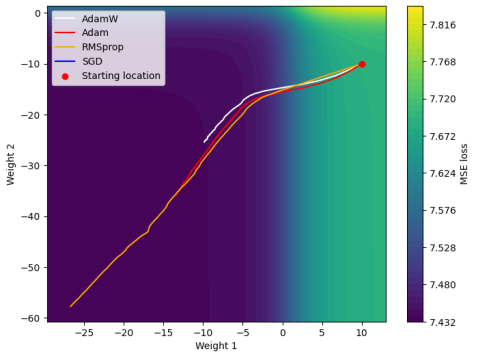
5.2 The Sigmoid activation function

The sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$ smoothly maps all real numbers $x \in \mathbb{R}$ to the interval $(0, 1)$.

The untrained sigmoid-activated networks lack the typical ReLU plateaus, and we can see the minimization of four arbitrary weights in figure 7.



(a) Optimization paths over two randomly chosen weights with a batch size of 64, optimization paths have one datapoint for every epoch.

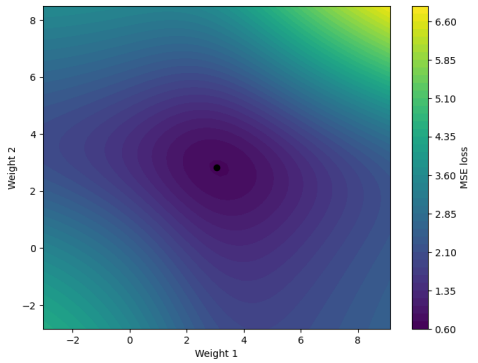


(b) Optimization paths over two other randomly chosen weights with a batch size of 4, optimization paths have one datapoint for every epoch.

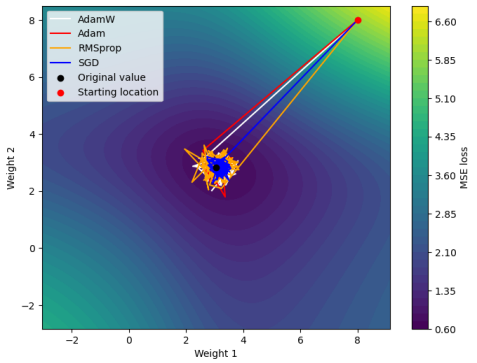
Figure 7: Untrained sigmoid-activated network

This figure illustrates the "speed" of the optimizers when gradients are small. RMSprop is quickly accelerated by its adaptive learning rate, while the adams are slightly held back by the bias correction. AdamW is even slower because of its weight decay. SGD is the slowest since its propagation is proportional to the small gradients.

After training the network, we find that the loss landscapes often look like 8a.



(a) Loss landscape for randomly chosen weights with sigmoid function.

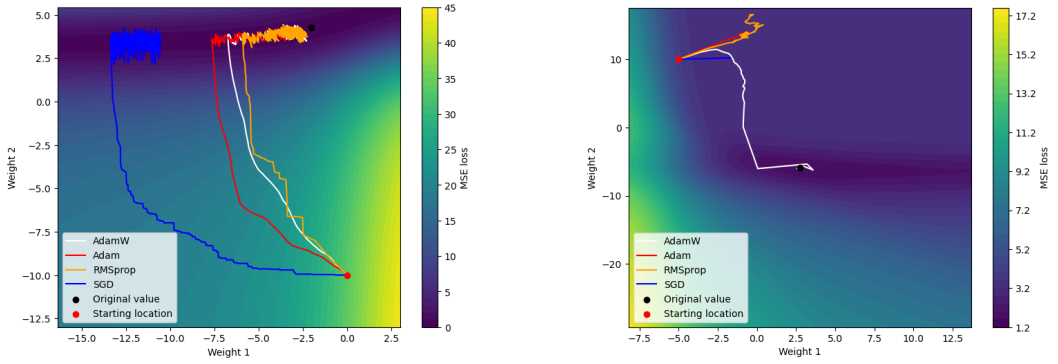


(b) Optimization paths over the two weights with a batch size of 1, optimization paths have one datapoint for every epoch.

Figure 8: Trained sigmoid-activated network

Many landscapes look like an almost convex function, and near-optimum minimiza-

tion of the two weights can easily be done by all of the studied optimizers. Looking at even more weights, we can find some interesting landscapes.



(a) Loss landscape of two randomly chosen weights with batch size of 1, optimization paths have one datapoint per batch.

(b) Loss landscape of two randomly chosen weights with batch size of 64, optimization paths have one datapoint per batch.

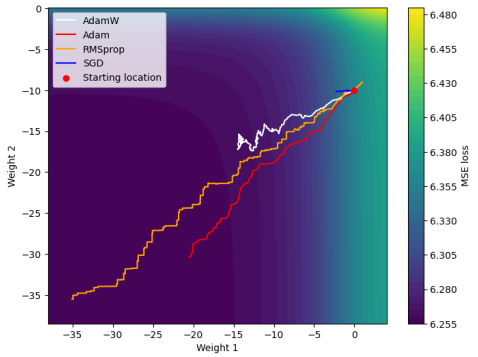
Figure 9: Trained sigmoid-activated network

Figure 9a has a higher sample rate of the optimization path, which allows us to notice how the momentum terms from the Adam optimizers make the process "calmer" since these often want to continue in the same direction. They are not bumped around by the gradient noise in the same way SGD and RMSprop are. This makes the Adam based optimizer's behavior in the loss landscape similar to how one would image a ball would roll down a hill. Also, we can again note how well all the adaptive learning rate-based optimizers handle the difference in gradient sizes and take a "shortcut" to the optimum. In the 9b we also see a case where the weight decay of AdamW helps out! It turned out that the global minima was close to origo, which is where AdamW is drawn when gradients are too small or in random directions. This made it converge while the other optimizers were bumped around by the noise. One must still note that with "normal" weight decay applied to the gradients, the other optimizers might have found this minima too.

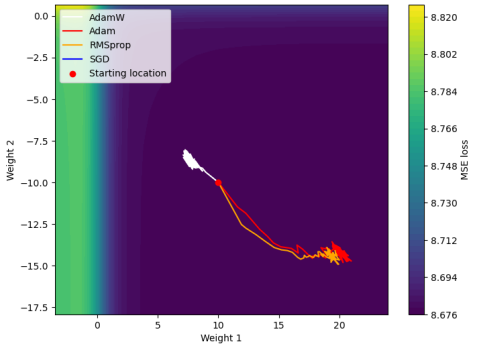
5.3 Tanh

The last activation function we are going to study is the hyperbolic tangent function $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. It maps real numbers $x \in \mathbb{R}$ to the interval $(-1, 1)$.

Optimization over the untrained network is similar to the sigmoidal case.



(a) Optimization over two randomly chosen weights with batch size of 1, optimization paths have one datapoint per batch.

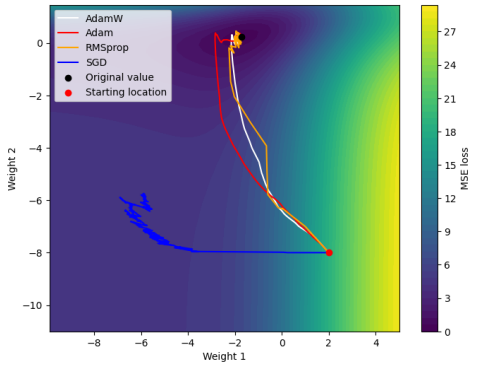


(b) Optimization over two randomly chosen weights with batch size of 4, optimization paths have one datapoint per epoch.

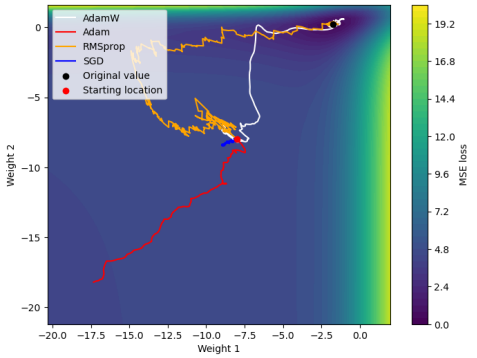
Figure 10: Untrained tanh-activated network

Again, we see the same speed comparison in 10a, but if we initialize a bit further away, the weight decay of AdamW actually slows it down enough to travel in the other direction. Which shows the regularizing effect of AdamW.

Again, training the network gives us more interesting landscapes.



(a) Optimization over two randomly chosen weights with batch size of 1, optimization paths have one datapoint per batch.



(b) Optimization over two randomly chosen weights with batch size of 4, optimization paths have one datapoint per batch.

Figure 11: Trained tanh-activated network

11a shows us the advantage of the adaptive learning rate once more, and 11b is another case where the weight decay draws AdamW in the correct direction when the gradients are small. All other optimizers begin traveling in the wrong direction, and only RMSprop is lucky and fast enough to make it to the minima.

6 Discussion

Plotting only the two weights gave an interesting insight into the training process, and readers of this paper are hopefully left with an intuitive understanding of the training process. However, in practice, one would never freeze all other weights, and the landscapes would therefore always change. This is why this study can only make an argument based on a qualitative analysis of the optimizers.

Firstly, one must note that the author has done his best to choose representative figures, and any "hand-picking" has at least not been deliberate. Nonetheless, AdamW gets close to the identified minima in 9 out of the 10 cases with the identified minima above. Getting even closer is likely just a question of reducing the learning rate and letting it iterate further. Interestingly, all the intended characteristics of AdamW(adaptive learning rate, momentum and weight decay) were identified to help the model. Adaptive learning rate helped it manage the problem with different scales in the loss landscape, momentum made it calmer and seemingly more methodical than RMSprop and the weight decay helped it find 2 global minima.

7 Conclusion

This paper visualized the training of two weights at a time for a model and identified some properties of their loss landscapes. Some of these properties turned out to be what AdamW was designed to solve, and most of the landscapes studied during the paper were successfully optimized by AdamW. The success of AdamW in navigating these landscapes offers further empirical support for its effectiveness, but future work should involve testing AdamW's performance on more complex models, or comparing its behavior across different datasets with varying characteristics.

References

[1] I. J. Goodfellow, O. Vinyals, and A. M. Saxe. Qualitatively characterizing neural network optimization problems, 2015.

[2] T. Szandała. *Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks*, pages 203–224. Springer Singapore, Singapore, 2021.

[3] Wikipedia contributors. Stochastic gradient descent — Wikipedia, the free encyclopedia, 2024. [Online; accessed 8-November-2024].