ADAM optimizer

Mor and Yona Coscas

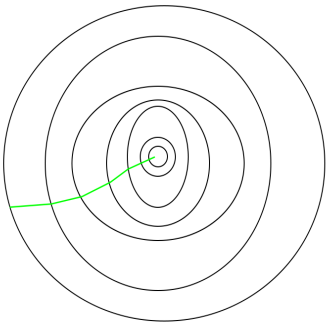
February 11, 2022

1 Introduction

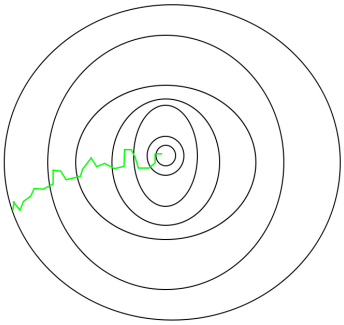
ADAM stands for ADAptive Moment estimation, which is commonly used to solve the optimization problem of updating deep neural network weights and biases, it solves some withdraws of the more common Stochastic Gradient Descent optimization algorithm.
The setting for a simple deep neural network optimization setup, is firstly a training dataset D which are samples from a unknown function $f(x)$, secondly a model $\hat{f}_\theta(x)$ with parameters θ that can be updated and lastly an objective function $L(\theta)$.
The goal of the optimization is to minimize the objective function w.r.t the model parameters, for example, we can look at the L_2 objective function, and the objective function would be $\min_\theta ||f(x) - \hat{f}_\theta(x)||_2^2$. $L(\theta)$ and a tuning parameter η which is the learning rate, its role is to determines the step size at each iteration while moving toward a minimum w.r.t the gradients.
The stochastic optimization process with vanilla stochastic gradient descent, D, $f(x)$, $\hat{f}_\theta(x)$ and η goes as follows:

1. Randomly sample a batch b of samples from the dataset $b \in D$.
2. Get the estimated function $\hat{f}_\theta(b)$, by running b through the model.
3. Evaluate the objective function $L(\theta) = \frac{1}{n}\sum_{i=1}^{n} L_i(\theta)$ using $f(\theta)$ and $\hat{f}_\theta(b)$.
4. Update the parameters *theta* as follows: $\theta := \theta - \frac{\eta}{n}\sum_{i=1}^{n} \nabla L_i(\theta)$

ADAM is an algorithm for first-order, gradient-based optimization for stochastic differentiable objective functions w.r.t their parameters. By first order, we mean that we need only to calculate the first-order partial derivatives of the function w.r.t its parameters. This is the desired property, since calculating first-order derivatives is as efficient as evaluating the function itself, but second-order methods such as Newton's method require second-order partial derivatives which require more computation. It is a gradient-based algorithm so it calculates the gradient direction of the objective function w.r.t its parameters, and updates them in the opposite direction of the gradients (we will elaborate on this point later on). Lastly, it is a stochastic algorithm, i.e., it does not require calculating the gradients for the whole dataset, but instead, it samples randomly from this dataset and updates the parameters based on this sample. The stochastic approach has been the main solution for deep learning models, over batch gradient descent although calculating the derivatives of the objective function over the whole dataset and updating the parameters results in a less noisy update rule, as can be seen in this illustration.



A stochastic approach will result in noisy update rules, and will look something like that



The reason for choosing SGD over BGD is the fact that calculating the exact gradients of the objective function of each and every sample over and over until the convergence of the optimization process is extremely time-consuming. Considering working with large datasets (millions of data samples), using this method renders the optimization process almost unfeasible, thus the use of a stochastic approach is desired.

2 Problem definition

The most general formalism for optimization in supervised learning is as follows, let us denote $L(\theta)$ to be a noisy objective function since it is a stochastic scalar function, and let it be differentiable w.r.t the parameters θ. We want to find a mapping function $\hat{f}_\theta(x)$ that minimizes the objective function over all training samples: $\min_\theta \frac{1}{N}\sum_{i=1}^{N} L(y^i, \hat{f}_\theta(x^i))$. $L(\theta)$, since it is a procedural optimization process that continues until convergence, we want to minimize the expected value of this function $\mathbf{E}[L_i(\theta)]$, with $L_1(\theta), ...L_T(\theta)$ being the realization of the function at subsequent timestamps 1,...,T.
Let $g = \nabla_\theta L_t(\theta)$ be the vector of partial derivatives of f_t, w.r.t θ evaluated at timestamp t. For the algorithm we will use

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

which is the exponential moving average of the first moment of the gradient (mean) weighted with the parameter $\beta_1 \in [0, 1)$ and

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

being the weighted average of the second moment of the gradient (the uncentered variance) with $\beta_2 \in [0, 1)$. β_1 and β_2 control the exponential decay rates of these moving averages. Both m_t and v_t are vectors initialized to zeros, which make the moments estimates to be biased towards zeros especially when the βs become closer to 1, so we need to fix them to be bias corrected, the resulting estimates are

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

then we can use those moment estimates to update update the parameter θ:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

η being the learning rate and ϵ is required to prevent numerical errors with very small standard deviations. A pseudo-code of ADAM can be seen as follows:

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

3 Comparison to other optimizers and ADAM properties

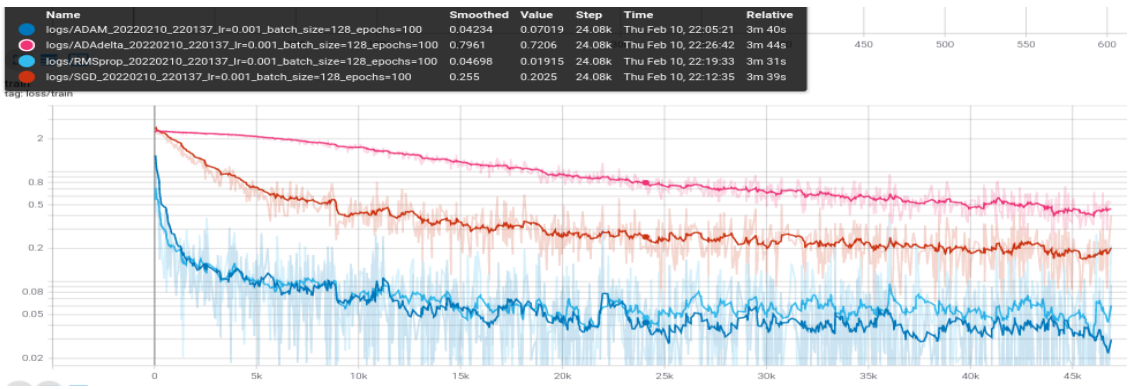
Batch Gradient Descent As mentioned in section 1, BGD passes through the whole dataset in batches, and store all the gradients in its path, lastly it performs the update step with fixed learning rate $\theta := \theta - \eta \nabla_\theta L(\theta)$.
Stochastic Gradient Descent As we mentioned above vanilla SGD, is just applying to each parameter the gradient of the error, with some learning rate, $\theta := \theta - \frac{\eta}{n} \sum_{i=1}^{n} \nabla_\theta L_i(\theta; x_i, y_i)$. It is worth mentioning that almost all stochastic approach (including ADAM) have a notable quality that makes them perform better on non-convex problems, the stochasticity of the formulation makes it easy for them to stay away from local minima, this also makes them possibly miss the global minima or not converge exactly to it, especially in the case of SGD with a fixed learning rate. In the vanilla setup, the error of convergence to a global minima in a convex setting of SGD is in the order of the learning rate, but if we add to vanilla SGD a learning rate schedule as suggested in [2] we can achieve the same optimal solution as in BGD.
Mini batch Gradient Descent is the combination of BGD and SGD, we do iterate over all dataset in random batches, but update the parameters each batch, thus achieving convergence much faster and still having the property of stochasticity. The key withdraw of all variants of gradient descent comparing to ADAM (or other adaptive methods) is they use a learning rate which the user needs to wisely choose and is the same for all parameters, another key challenge for those algorithms is saddle points, they do not perform well in smooth plateaus, which are common in deep learning [3].
As we mentioned earlier ADAM stands for adaptive moment estimation, the term adaptive refers to the property that ADAM calculates adaptive learning rates for each parameter, in contrast to SGD, in which the learning rate is fixed to all the parameters. This property is similar to other adaptive methods, such as Adadelta, Adagrad and RMSprop.
Adagrad is a gradient-based optimization algorithm that adapts the learning rate to the parameters, performing smaller updates for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features.
Let's denote $g_{t,i} = \nabla_\theta J(\theta_{t,i})$, which is the partial derivative of the objective function w.r.t its parameters θ_i, we calculate a diagonal matrix G_t, where each element is $G = \sum_{i=1}^{N} g_i g_i^T$, where N is the number of itterations, thus the update rule is $\theta_{t+1,i} = \theta_{t,i} - \dfrac{\eta}{\sqrt{G_{t,ii}} + \epsilon} \cdot g_{t,i}$ This method is well suited for sparse data because of its adaptivity to occurrence of features.
Adadelta Adadelta extens Adagrad, the main problem with Adagrad is the fact that it stores all past gradients in the G matrix, resulting in a monotonic decrease of the learning rate, which is not a suitable result, in non-convex problems. Adadelta, looks only on a running window of the accumulated gradients, i.e. a delta of the gradients. In this condiguration G_t stores in its diagonal elements the current gradients, and a decaying average of past gradients (instead of the sum of all past gradients), i.e. $E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$, where γ is a usually around 0.9, now we get an update rule: $\Delta\theta_t = -\dfrac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$. Adadelta is a momentum based optimizer
RMSprop was also introduced due to the shortcomings of Adagrad in regard to its quick decreasing of learning rate, it calculates the running average of past squared gradients $E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$, and the update rule is $\theta_{t+1} = \theta_t - \dfrac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$, we can see it is similar to the variance term in ADAM except the running average in RMSprop is only over the previous

average of square gradients and ADAM is exponentially moving. The other property of ADAM is moment estimation, ADAM calculates past gradient averages, similarly to momentum methods and in fact can be seen as a combination of a momentum method that does not calculate the variance and RMSprop that do not calculate the average, and lacks the term regarding the first moment averaging.

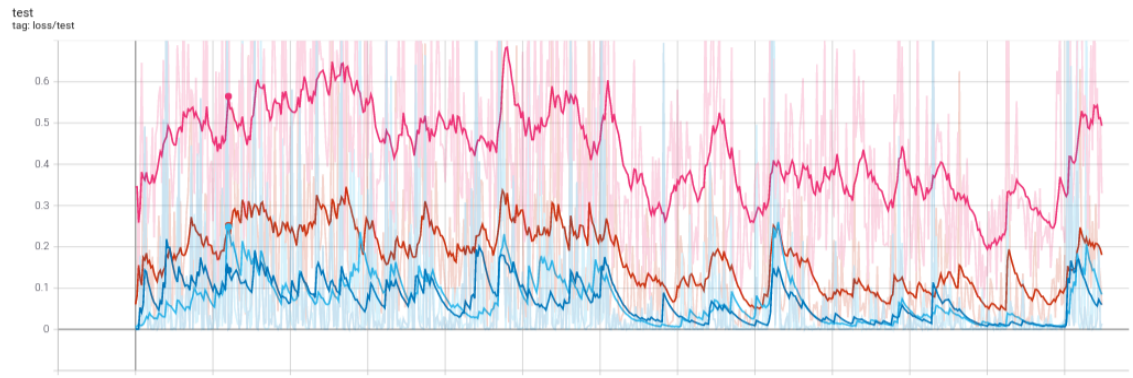
AMSGrad In [5] the authors argue that ADAM do not converge well to an optimal solution for convex problems, due to the exponential moving avergae of the gradients, which result in forgettness of long-term memory of past gradients. Therefore suggest an alternative AMSGrad which is a corrected version of ADAM, which replaces $\hat{v}_t$ to be $\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$, all in all the authors get a similar algorithm to ADAM, but found that the de-biasing can be avoided, thus getting:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{v}_t &= \max(\hat{v}_{t-1}, v_t) \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t \end{aligned} \tag{1}$$

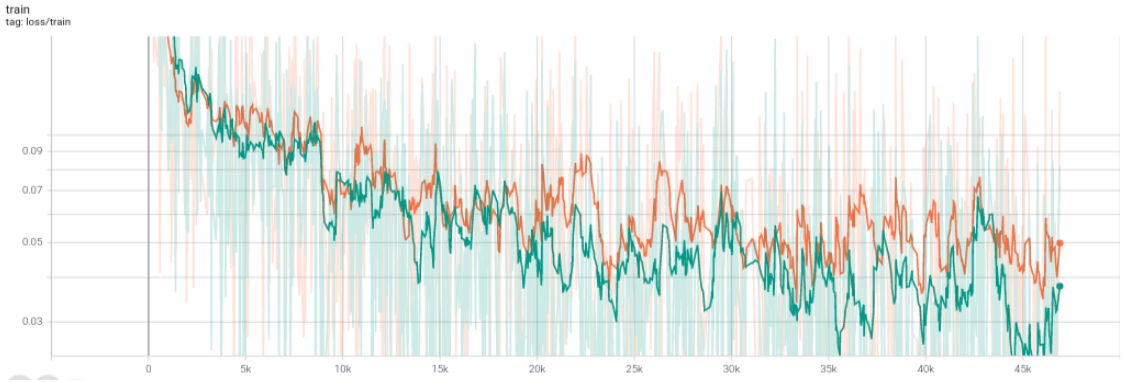
where v_t is the regular moving average for the variance this fixes the forgetness of ADAM and was shown to perform better on convex problems than ADAM, and we get an update rule similar to ADAM but never get an increasing step-size (which is to blame for its inability to converge).
Where ADAM shines is actually in non-convex error surface such as deep learning models (where it is widely used), and in [1], the authors even provided an upper bound for convergence rate under some mild conditions, but the authors showed that in some cases of deep learning problems AMSgrad outperforms ADAM. We evaluated 4 optimizers in different settings using our code, and a sample problem. We designed a convolutional neural network with 2 convolution layers and two fully connected layers with relu as the non-linear function, and trained it to recognize hand written digits from 0 to 9 (known as the MNIST dataset). We compared ADAM, SGD, RMSprop and Adadelta, with different hyperparameters.
In this figure we can see the results with batch size of 128 and learning rate of 0.001 (all lines were smoothed for better visual evaluation):



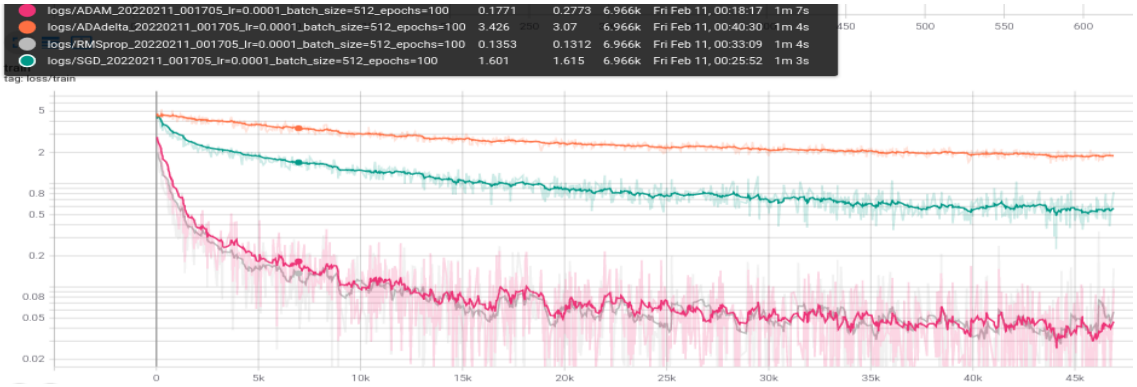
the pink line is Adadelta, and has a poor convergance, as do SGD in orange, while RMSprop (in bright blu) provide a very good convergance and is very close to the results from ADAM (in darker blue), but still ADAM outperform them all.
It is also seen in the results on the test set, where ADAM outperform all other optimizer in average, here we see results over all the test set with the model (without calculating the optimizer, since we do not train the model):



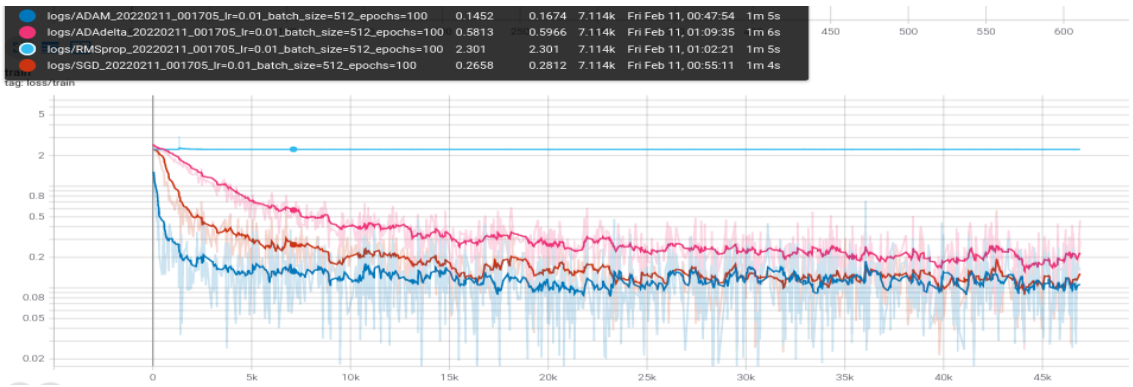
If we look at the convergence of ADAM (green) vs. RMSprop (orange) in a logarithmic scale we can see that ADAM is slightly better as expected.



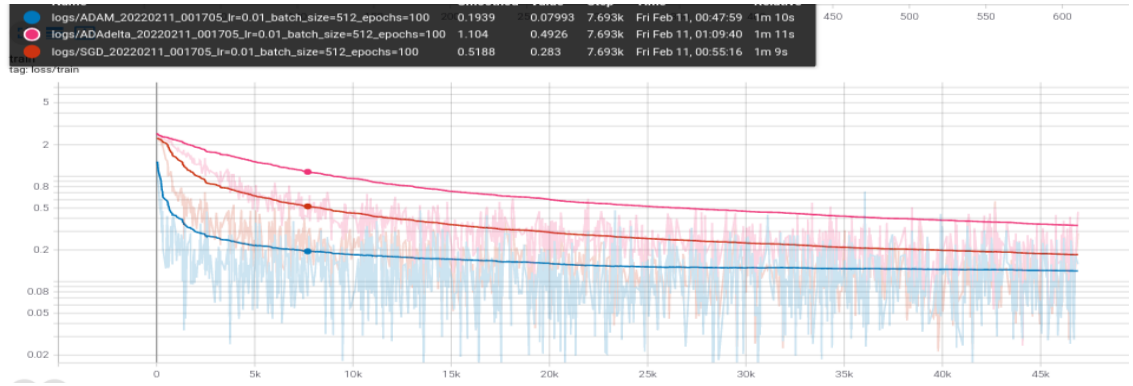
In this figure we can see the different behavior when choosing a smaller learning rate of 0.0001:



ADAM in pink, Adadelta in orange, SGD in green and RMSprop in grey.
We can see that we are able to converge with a slightly smaller error, but it took almost twice as long to arrive to do so. We also can see that SGD suffer greatly from alternating the learning rate, since it does not adapt the learning rate properly to the loss and has a bad cobvergance rate, if we continue to iterate we probably would get a small error but it would take days, Adadelta also suffer and provide the worst convergence rate.
We can see that the test results are as expected:
In the figure bellow we used a high learning rate of 0.01, and expected SGD and Adadelta to converga fast to a high error value, since the step size is high, but not being able to adapt the step size to a smaller one will cause a high error while training:



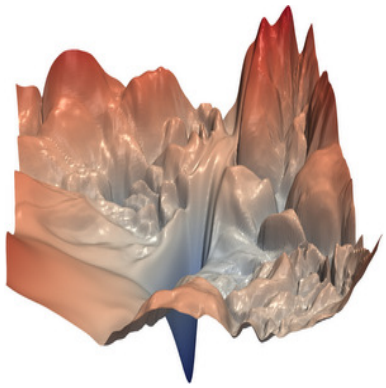
ADAM in blue, SGD in brown, Adadelta in pink and RMSprop in light blue.
This figure show us, that as expected the error of SGD and Adadelta got stuck after they converge, but their convergence rate was quick and they get almost the same error results as ADAM, but still ADAM provide a better convergence rate. We also can see that RMSprop does not learn and got stuck, this is due to the lack of momentum term in RMSprop, that got it stuck at a high minima or saddle point, in contrast to ADAM and Adadelta who have a momentum term. Here we rempved RMSprop to be able to better visualize the convergence of the optimizers



It visualizes better the better convergence rate of ADAM (after smoothing the results)

4 Miscellanea

In [6], the authors argue that although ADAM provides extremely good results on deep learning optimization i.e. trying to optimize a loss surface, it provides very poor results on other optimization problems. The key difference between deep learning model optimization is the sheer number of parameters to optimize (several million), and that derivatives of the error function are usually ill-conditioned [4], thus the error surface has many saddle points, resulting from one hand convex optimization is not feasible in this cases, and in the other hand second-order optimization is not feasible due to the high dimensionality of parameters, which will require $O(n^2)$ in memory. This is where ADAM shines, it doesn't require more than calculating the gradients themselves, like SGD, but its momentum allows it to overcome local minima and saddle points, with its ability to properly tune the learning rate to different parameters. in this figure we can look at one loss surface from ResNet architecture:



we can see that it is a non convex and hard optimization problem, consisting of many local minima, but it has a very distinct global minima.

References

[1] Xiangyi Chen, Sijia Liu, Ruoyu Sun, and Mingyi Hong. On the convergence of a class of adam-type algorithms for non-convex optimization. *arXiv preprint arXiv:1808.02941*, 2018.

[2] Christian Darken, Joseph Chang, John Moody, et al. Learning rate schedules for faster stochastic gradient search. In *Neural networks for signal processing*, volume 2. Citeseer, 1992.

[3] Yann Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *arXiv preprint arXiv:1406.2572*, 2014.

[4] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. *arXiv preprint arXiv:1712.09913*, 2017.

[5] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*, 2019.

[6] Sharan Vaswani, Aaron Mishkin, Issam Laradji, Mark Schmidt, Gauthier Gidel, and Simon Lacoste-Julien. Painless stochastic gradient: Interpolation, line-search, and convergence rates. *Advances in neural information processing systems*, 32:3732–3745, 2019.