Master's Paper of the Department of Statistics, the University of Chicago

Email: kaig@uchicago.edu)

Comparison of Optimization Algorithms and Potential Improvement of ADAM Based on Time-Series

Kai Gong

Supervisor: Weibiao Wu

Approved (Mei Wang)

Date (September-12, 2023)

September-12, 2023

Abstract

In this study, we provide a comprehensive comparison of various optimization methods across multiple model architectures and introduce a novel Adam-derived optimization technique that aims to enhance the performance observed in the conventional Adam method. Initially, we conduct an exhaustive evaluation of prominent optimization techniques, including SGD, ASGD, and ADAM. We assess their performance in terms of convergence and stability, especially when complemented with learning rate decay strategies. This comparative study spans two primary models: the Convolutional Neural Network (CNN) and the Deep Q-Network (DQN) – a model pivotal for game-based machine learning.

Building on the foundation of the Adam optimization method, we present "TAdam", a sophisticated variant that incorporates a time series function to achieve exponential decay. Our modification to the time series function ensures that TAdam is not just an iteration of Adam but a more evolved version. Preliminary results indicate that TAdam consistently outperforms the standard Adam optimizer in a variety of scenarios, suggesting its potential as a robust optimization choice for future machine learning endeavors.

Contents

1 Introduction **3**

2 Comparison of Optimization Methods **3**

3 Retrospect on traditional ADAM **5**

4 Influence on Converging Speed **7**

5 TADAM algorithm **8**
5.1 Initialization Bias Correlation 10
5.2 Observation on Performance Trend 10
5.3 Parameter Adjustment 12

6 Statistical Evidence of TAdam's Superior Performance **13**
6.1 Expectile Setting 13
6.1.1 Verification of Faster Convergence 13
6.1.2 Verification of Higher Accuracy 15
6.2 Logistic Regression Setting 17
6.2.1 Convergence Speed Comparison 17
6.2.2 Enhancements in Discrete Model 19

7 Conclusion **19**

A Appendix **20**

1 Introduction

Stochastic gradient optimization holds significant relevance across numerous scientific and engineering domains. Many challenges in these areas involve optimizing a scalar function based on its parameters. When this function is differentiable concerning its parameters, gradient descent emerges as a notably effective optimization technique. This is because computing first-order partial derivatives for all parameters is as computationally intensive as just evaluating the function itself. Many such functions display stochastic characteristics. For instance, they may comprise a series of subfunctions assessed on varying data subsets. Here, optimization can be enhanced using gradient steps related to these individual subfunctions, known as stochastic gradient descent (SGD) or ascent. SGD has been at the heart of many breakthroughs in machine learning, particularly in the realm of deep learning. Aside from data subsampling, other factors, like dropout regularization, can introduce variability in objectives. Efficient stochastic optimization strategies are essential for these volatile objectives. This paper mainly concentrates on optimizing stochastic objectives within vast parameter spaces. For such scenarios, advanced optimization techniques may not be appropriate, and our discussion will primarily revolve around first-order methods.

The rapid evolution of deep learning models necessitates optimization algorithms that can keep pace. Comparison between each optimization algorithm is necessary. Inspired by the learning rate decay, a potential improvement on the Adam algorithm rises. Enter TAdam - an optimization algorithm designed to address the shortcomings of the Adam algorithm by incorporating time-series dynamics. While the Adam algorithm has been the staple choice for many in the field, its limitations become evident in scenarios requiring faster convergence and greater accuracy. This paper introduces TAdam, rigorously evaluating its performance using empirical and statistical analyses across varied neural network settings.

2 Comparison of Optimization Methods

Optimization methods play a crucial role in training complex models, especially deep neural networks. There are various optimization algorithms, and selecting the right one can often improve convergence speed and model performance. In this section, we'll discuss and compare three popular optimization methods and the new method that will be introduced later: Stochastic Gradient Descent (SGD)[1], Averaged Stochastic Gradient Descent (ASGD)[2], Adam, and TAdam.

To give a clear view of how these optimization methods work on industry-level machine learning models, the CNN model and DQN gaming learning AI are used for training. Each method separately discusses the cases with or without introducing learning rate decay.

DQN (Deep Q-Network) integrates deep neural networks with Q-learning to tackle reinforcement learning tasks.[3] At its core, DQN uses a neural network to approximate the Q-function, estimating the expected cumulative reward for taking an action in a given state. To stabilize learning, two key strategies are employed: experience replay and a target network. With experience replay, the agent saves its interactions in a buffer, and during training, it samples randomly from this buffer, breaking the

correlation between sequential experiences. The target network, a periodically updated replica of the main Q-network, aids in computing the expected rewards, ensuring that learning targets remain consistent over a set of updates. By combining these techniques, DQN efficiently learns policies in high-dimensional environments by predicting optimal actions based on input states.[4] In the DQN gaming learning AI model, the AI performance score is compared after each iteration epoch (the score is known as the Q value or reward of gaming), and the record is shown in Figure 1.

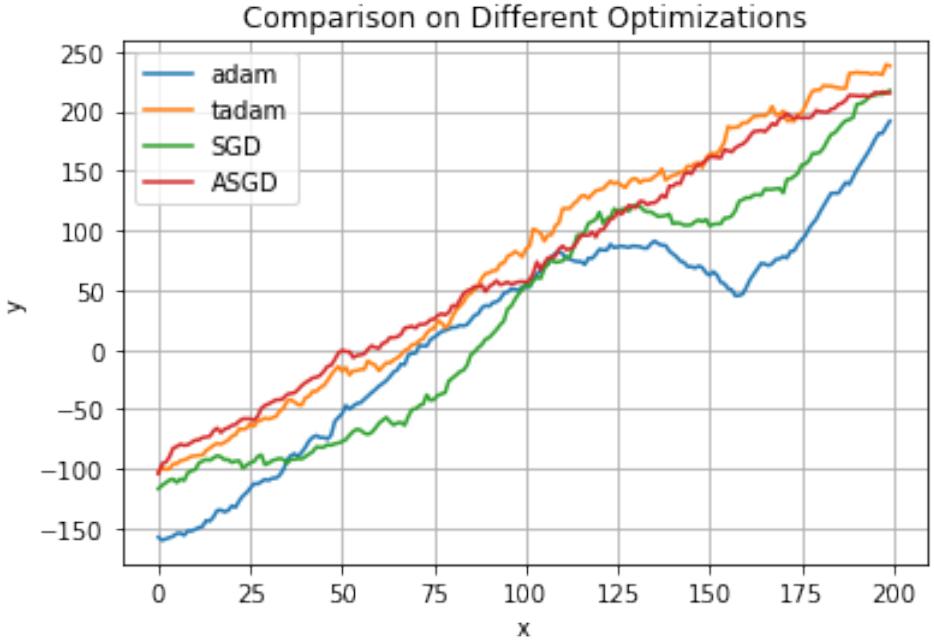

Figure 1

In the sphere learning case, ASGD and Tadam seem to perform slightly better than other learning methods. Under this setting, learning rate decay is not applicable, as the AI learning process is not continuous. There are two gaps in the learning process. If the learning rate decays during the training epochs, the model would tend to stay in local optima as the learning rate decreases, and would not overcome the gaps. However, if we consider the CNN model shown below, there would be no problem in using the learning rate decay.

A Convolutional Neural Network (CNN) is a class of deep learning models designed primarily for processing structured grid data, such as images.[5] Central to CNNs are convolutional layers, which employ a series of learnable filters to scan input data (like an image) in small, overlapping regions, called receptive fields. By doing so, these layers detect local patterns like edges, textures, and shapes. As the network deepens, it can recognize more complex patterns. Following the convolutional layers, pooling or subsampling layers are often used to reduce the spatial dimensions, thereby reducing the computation and emphasizing dominant features. The network usually concludes with fully connected layers to aggregate spatial information and produce the final output. Due to their architecture, CNNs are invariant to local translations and can learn hierarchical features, making them exceptionally powerful for tasks like image recognition and classification.[6] In CNN models, loss values are directly compared to illustrate the performance of convolution, Curves dawn as Figure 2:

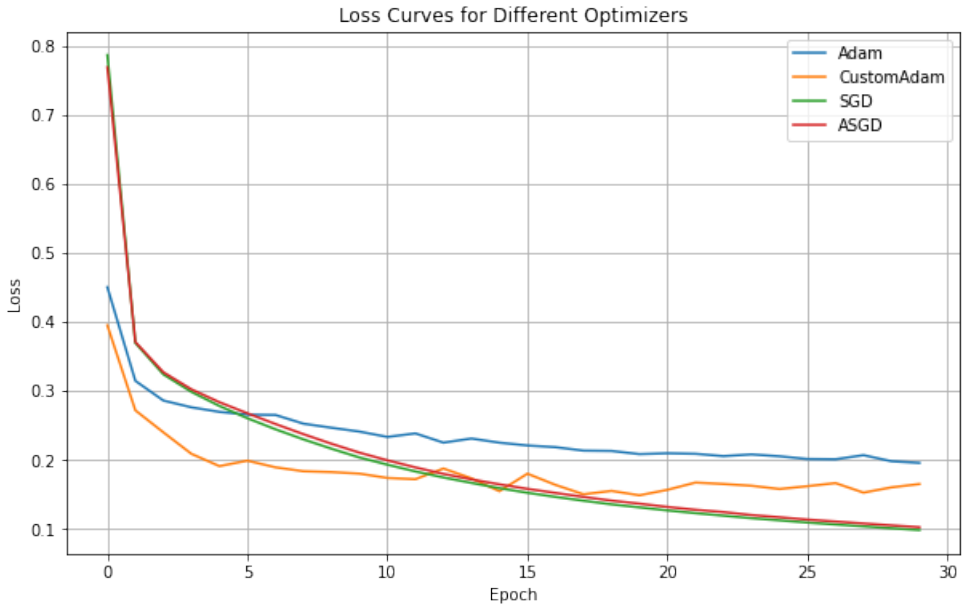

Figure 2

And Consider adding a learning rate decay as $lr = \frac{0.1}{step^{0.7}}$:

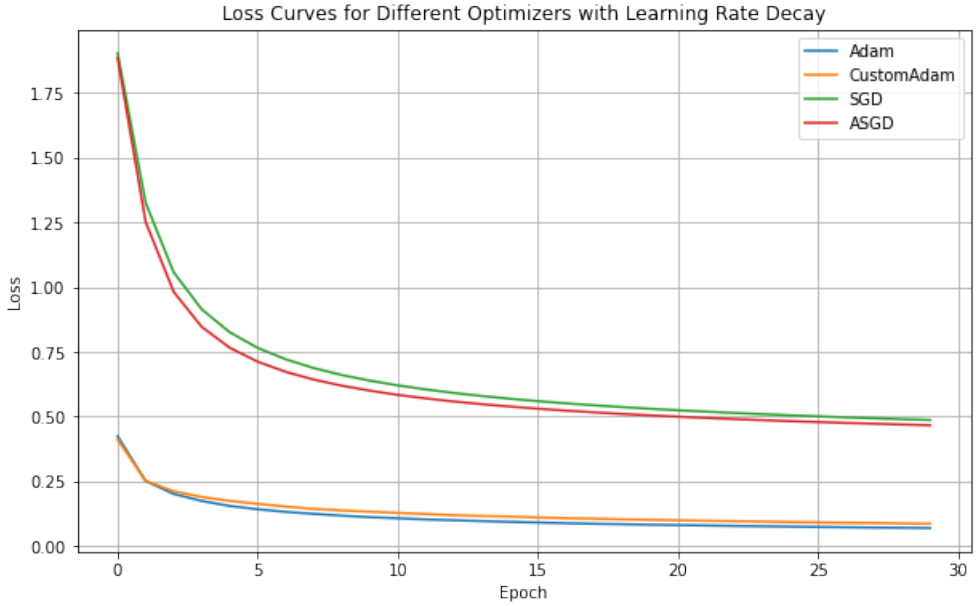

Figure 3

It seems that learning rate decay would benefit more from Adam-based algorithms compared to SGD-based algorithms. Notice that the curve with learning rate decay would smooth the curve, which indicates less variance and a higher convergence level.

3 Retrospect on traditional ADAM

Deep learning models, given their complexity and vast potential, require robust optimization strategies to realize their full potential. The Adam algorithm, a popular

choice among researchers and practitioners, exhibits commendable performance due to its adaptive learning rate. However, there is always room for enhancement. TAdam steps in here, bringing the promise of superior convergence speed and heightened accuracy by weaving in time-series dynamics.

Optimization algorithms serve as the backbone for deep learning. The Adam algorithm, christened after its "Adaptive Moment Estimation" mechanism, has garnered widespread attention due to its efficiency in training deep learning models. Originally conceptualized by Diederik Kingma and Jimmy Ba in 2015, Adam integrated the best of AdaGrad and RMSProp.[7]

Algorithm 1 Adam: Our proposed algorithm for stochastic optimization. g_t^2 indicates the elementwise square $g_t \bigodot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^8$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

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

The Adam algorithm has been pivotal in shaping the landscape of gradient-based optimization methods. Its ability to compute adaptive learning rates for different parameters sets it apart. However, its Achilles' heel lies in its static hyperparameters which, under certain conditions like non-convex settings or in the face of noisy gradient challenges, can lead to inconsistent performance. Recognizing these inconsistencies, our research sought to explore an evolved version of Adam, leading to the birth of TAdam. Integrating time-decaying aspects, TAdam promises to address the very challenges that sometimes plague Adam.

In Adam, the selection of two exponential decay rates was based on the recommendations of the authors in the paper "ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION". Yet, the arbitrary selection of two exponential decay rates in the original algorithm sparked intrigue. This research evaluates the influence of these decay rates on convergence speed.

Section 4 explores the impact of different exponential decay rates on convergence speed. In section 5, we elaborate on the refined algorithm that introduces time-series

dynamics to the exponential decay rates. Lastly, section 6 provides an empirical analysis of TAdam's convergence rate in defined expectile settings and logistic settings.

4 Influence on Converging Speed

Neural networks, with their intricate architectures, present a challenge when it comes to distinguishing the statistical superiority of one algorithm over another. The convergence point, especially in non-convex systems, remains elusive. To address this and offer a tangible comparative analysis, we ventured into two gradient settings - the expectile function (continuous targets) and logistic regression (discrete targets).

For this analysis, all neural networks are structured as 3-layer networks with dimensions 784 x 256, 256 x 64, and 64 x 10. We utilize the MNIST dataset comprising images of handwritten digits (0-9) with each image having 784 pixels.[8] Our chosen loss function is the negative log-likelihood, a suitable metric for multi-class classification problems.

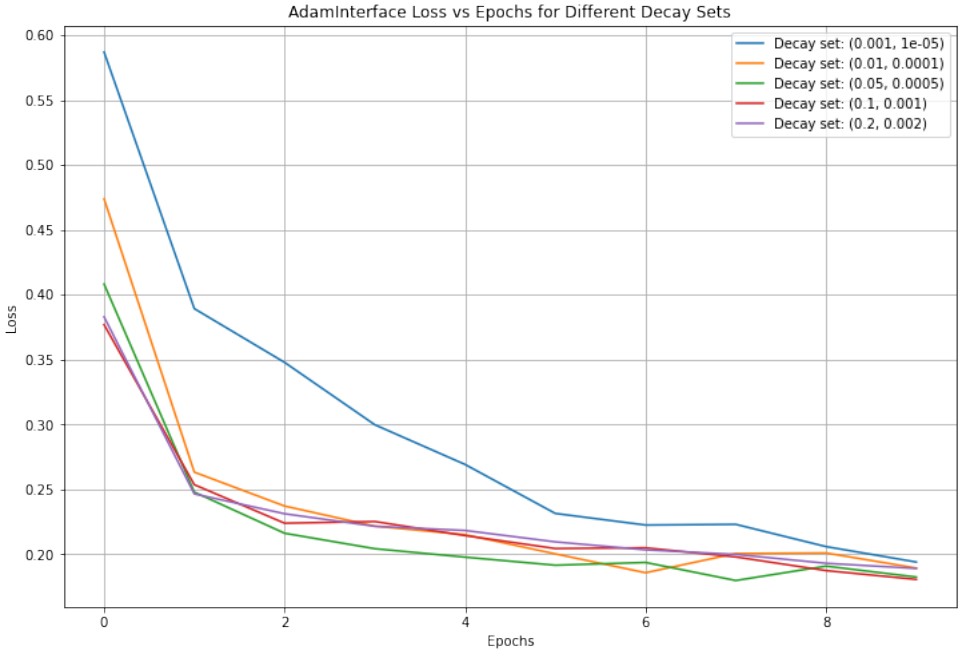

Figure 4

As observed in Figure 4, the decay rates (0.1, 0.001), as originally proposed by the Adam paper, do offer some advantages over other rates. However, the distinction is not stark. For instance, the rates (0.05, 0.0005) also display commendable performance, especially between epochs 2-7.

More specifically, there is some evidence showing that low decay rate values will perform better during late training, both (0.05,0.0005) and (0.01,0.0001) perform better than (0.1,0.001) at epochs 5-7.

Further investigation into decay rates around (0.1,0.001) revealed potential for time-series-based decay rates to optimize both early and late training phases.

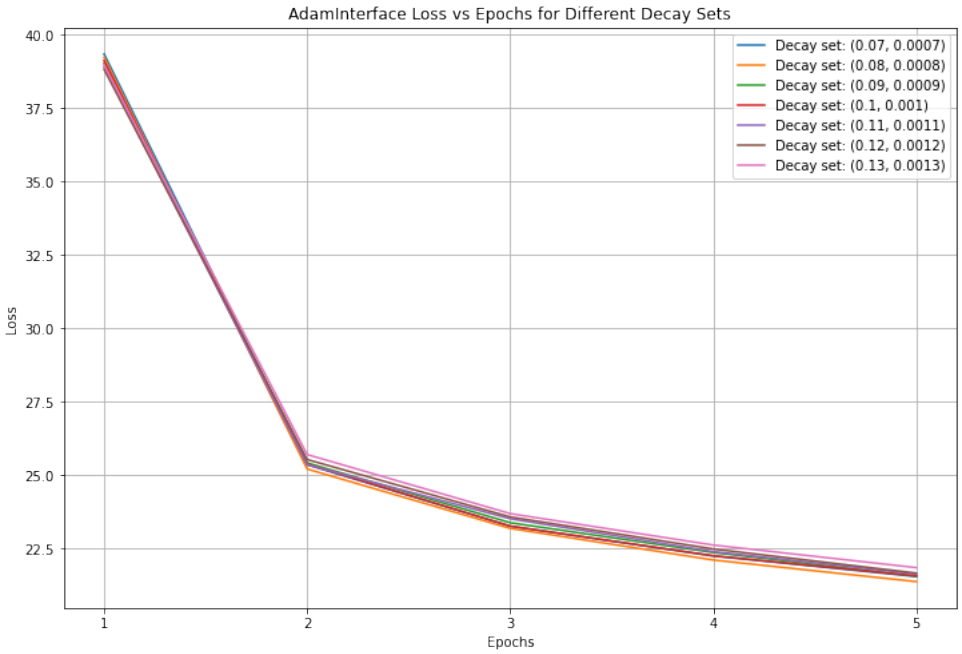

Figure 5

A closer examination of decay rates in the vicinity of (0.1,0.001) unveils that these rates might not be the best choice for both early and late training phases. This observation led us to contemplate the potential of time-series-based decay rates.

5 TADAM algorithm

The conventional Adam algorithm does not guarantee definitive convergence. This means that theoretically, it may oscillate around the true convergence point without ever reaching it.[9] This challenge can potentially be mitigated by incorporating a time-series-based exponential decay rate.[10]

Algorithm 2 TAdam: a variant of Adam based on the time-series exponential decay rate. Good default settings for the tested machine-learning problems are $\beta_{1,t} = 1 - \frac{1}{(10t+10)^{0.5}+10^{-3}}$, $\beta_{2,t} = 1 - \frac{1}{(100t+100)^{1.0}+10^{-5}}$. The function of the exponential decay rates can be varied.

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
$n_1 \leftarrow 0$ ▷ Initialize 1st normalizing factor
$n_2 \leftarrow 0$ ▷ Initialize 2nd normalizing factor
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
    $m_t \leftarrow \beta_{1,t} \cdot m_{t-1} + (1 - \beta_{1,t}) \cdot g_t$ ▷ Update biased first moment estimate
    $v_t \leftarrow \beta_{2,t} \cdot v_{t-1} + (1 - \beta_{2,t}) \cdot g_t^2$ ▷ Update biased second raw moment estimate
    $n_1 \leftarrow n_1 \cdot \beta_{1,t} + (1 - \beta_{1,t})$ ▷ Update 1st normalizing factor
    $n_2 \leftarrow n_2 \cdot \beta_{2,t} + (1 - \beta_{2,t})$ ▷ Update 2nd normalizing factor
    $\widehat{m_t} \leftarrow m_t / n_1$ ▷ Compute bias-corrected first moment estimate
    $\widehat{v_t} \leftarrow v_t / n_2$ ▷ 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

5.1 Initialization Bias Correlation

Compared to traditional Adam, TAdam also utilizes initialization bias correction terms.[11] We will derive the term for the second-moment estimate; the derivation for the first-moment estimate is completely analogous. Let g be the gradient of the stochastic objective f, and we wish to estimate its second raw moment (uncentered variance) using an exponential moving average of the squared gradient, with decay rate $\beta_{2,t}$. Let g_1, ..., g_T be the gradients at subsequent timesteps, each a draw from an underlying gradient distribution $g_t \sim p(g_t)$. Let us initialize the exponential moving average as $v_0 = 0$ (a vector of zeros). First note that the update at the timestep t of the exponential moving average $v_t = \beta_{2,t} v_{t-1} + (1 - \beta_{2,t}) g_t^2$ can be written as a function of the gradients at all previous timesteps:

$$v_t = g_t^2 \sum_{i=1}^{t} \prod_{j=i+1}^{t} \beta_{2,j}(1 - \beta_{2,i})$$

Notice we can write the expression in a recursive way as $n_{2,t} = n_{2,t-1} * \beta_{2,t} + (1 - \beta_{2,t})$.

We wish to know how E[vt], the expected value of the exponential moving average at timestep t, relates to the true second moment E[g_t^2], so we can correct for the discrepancy between the two. Taking expectations of the left-hand and right-hand sides of the above equation:

$$\begin{aligned} E(v_t) &= E[g_t^2 \sum_{i=1}^{t} \prod_{j=i+1}^{t} \beta_{2,j}(1 - \beta_{2,i})] \\ &= E[g_t^2] * n_{2,t} + \zeta \end{aligned}$$

where $\zeta = 0$ if the true second moment $E[g_i^2]$ is stationary; otherwise ζ can be kept small since the exponential decay rate function β can (and should) be chosen such that the exponential moving average assigns small weights to gradients too far in the past. What is left is the term n_t which is caused by initializing the running average with zeros. In algorithm 2 we therefore divide by this term to correct the initialization bias. In case of sparse gradients, for a reliable estimate of the second moment, one needs to average over many gradients by choosing a small starting value of β_2; however, it is exactly this case of small β_2 where a lack of initialization bias correction would lead to initial steps that are much larger.[12]

5.2 Observation on Performance Trend

This modification stems from the findings highlighted in the previous section. The proposed algorithm dynamically adjusts decay rates, ensuring rapid and precise convergence. The idea of this algorithm comes from the observation from the previous section. Notice that comparative high decay rates lead to low loss value at early training in high frequency; while the low rates lead to low loss value at late training process. The key challenge with the traditional Adam algorithm is its potential to oscillate without ever reaching true convergence. To counteract this, TAdam introduces time-series-based exponential decay rates.

Derived from observations in Section 2, TAdam dynamically adjusts decay rates, achieving swift and precise convergence. Comparative analysis using the same neural network parameters as before, showcased TAdam's superior performance.

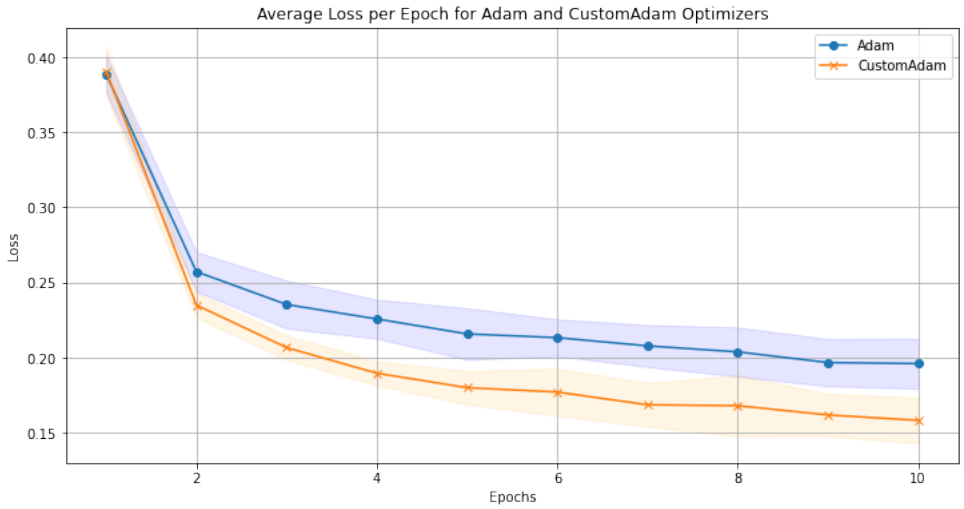

Figure 6

In Figure 6, there are two trends that require notice. The first obvious trend is that TAdam has considerably smaller loss values than Adam, which indicates a more precise convergence to the true function. The second trend is that TAdam seems to converge faster than the normal Adam. TAdam usually reaches convergence at epoch 4, while Adam usually reaches convergence at epoch 8 or 9. The above graph makes it somehow hard to observe this trend, so a closer look is necessary at each single comparative set:

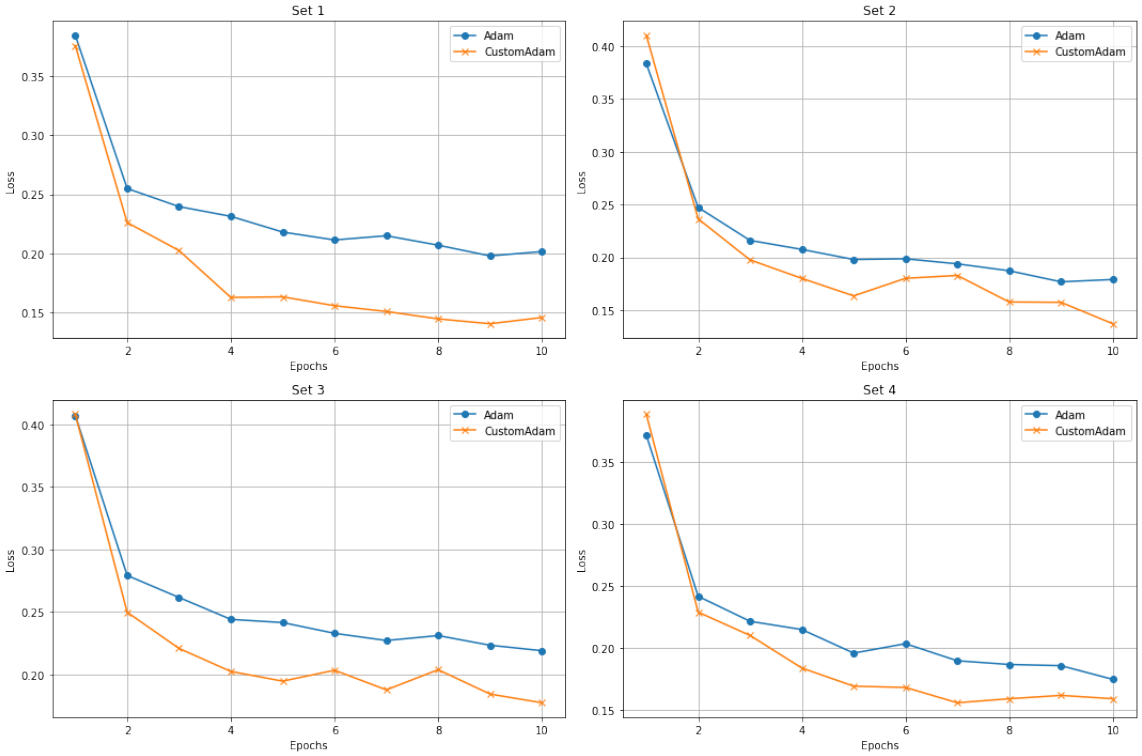

Figure 7

The single paired sets in Figure 7 show clearly that TAdam typically achieved convergence by epoch 4, whereas Adam took until epoch 6. This might indicate an early

convergence of the TAdam algorithm. However, there is no statistical evidence showing the significance of the result. Thus we need a more precise model to make the conclusion.

5.3 Parameter Adjustment

Notice the exponential decay function should be adjusted by different problems to achieve the best performance.[13] In fact, the function is quite sensitive according to the problem setting. Consider several decay functions below.

function 1 (faster decay):

$$\beta_{1,t} = 1 - \frac{1}{(10t+10)^{0.7} + 10^{-3}}$$

$$\beta_{2,t} = 1 - \frac{1}{(10t+10)^{1.4} + 10^{-5}}$$

function 2 (slower decay):

$$\beta_{1,t} = 1 - \frac{1}{(10t+10)^{0.3} + 10^{-3}}$$

$$\beta_{2,t} = 1 - \frac{1}{(10t+10)^{0.6} + 10^{-5}}$$

function 3 (logit decay):

$$\beta_{1,t} = \frac{1}{(1+e^{\frac{-t-150}{100}})} - 10^{-3}$$

$$\beta_{2,t} = \frac{1}{(1+e^{\frac{-t-300}{100}})} - 10^{-5}$$

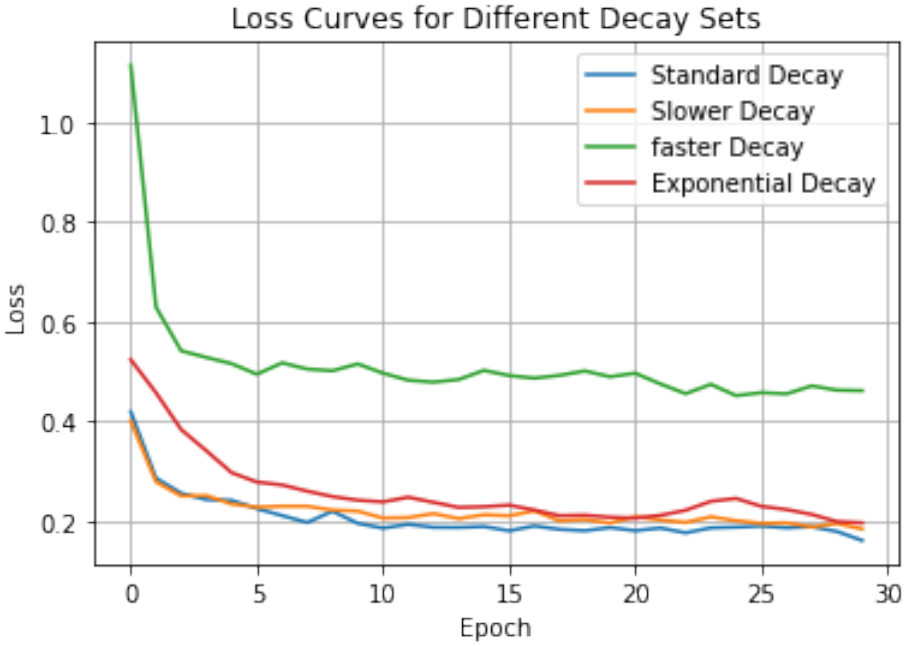

Figure 8

In Figure 8, the loss value is sensitive to the decay function. Hence the decay function should be customized for each individual problem. The design of the decay function is beyond the topic of this paper, further study may need to be done in the future.

6 Statistical Evidence of TAdam's Superior Performance

In the context of neural networks, it's evident that TAdam converges more rapidly and produces a reduced loss value than its predecessor. However, quantifying this difference becomes challenging due to the complexity of the classification model's accuracy. This complexity, combined with the unpredictability of determining a true convergence point in a non-linear, non-convex system, limits our capability for direct comparison.

To address this, our section delves into two distinct convex and deterministic gradient settings, catering to both continuous and discrete objectives. This enables a more quantitative evaluation of the performance disparity between the models. Initially, we employ the expectile function as our loss determinant in a continuous target setting. Here, the target is continuous, enabling a clear computation of both the precision of convergence and the model's accuracy. Subsequently, we use logistic regression for cases with discrete targets. Given that the target, y, adopts a Bernoulli distribution (either 0 or 1), the model's accuracy isn't as transparent as in continuous scenarios. Nonetheless, assessing the rate of convergence serves as a reliable indicator of the model's efficacy.

6.1 Expectile Setting

$$Loss = \alpha[(x-m)^+]^2 + (1-\alpha)[(x-m)^-]^2$$

$$Gradient = 2\alpha(m-x)\mathbb{1}_{x-m>0} + 2(1-\alpha)(m-x)\mathbb{1}_{x-m<0}$$

Diving into the expectile function, we utilized it as a loss measure. This continuous target setting allowed for a tangible comparison, revealing TAdam's consistent edge over Adam. Through graphical representations, it became evident that TAdam's convergence was not just faster but also more accurate. Over 10,000 iterations, TAdam exhibited remarkable stability, with its loss curve showcasing lower variance. This superior accuracy, further augmented by learning rate decay considerations, emphasizes TAdam's prowess in the continuous setting.[14]

6.1.1 Verification of Faster Convergence

We set $\alpha = 0.5$ and consider that if the difference between calculated m and true m is less than 10^{-3}, the model reaches convergence.

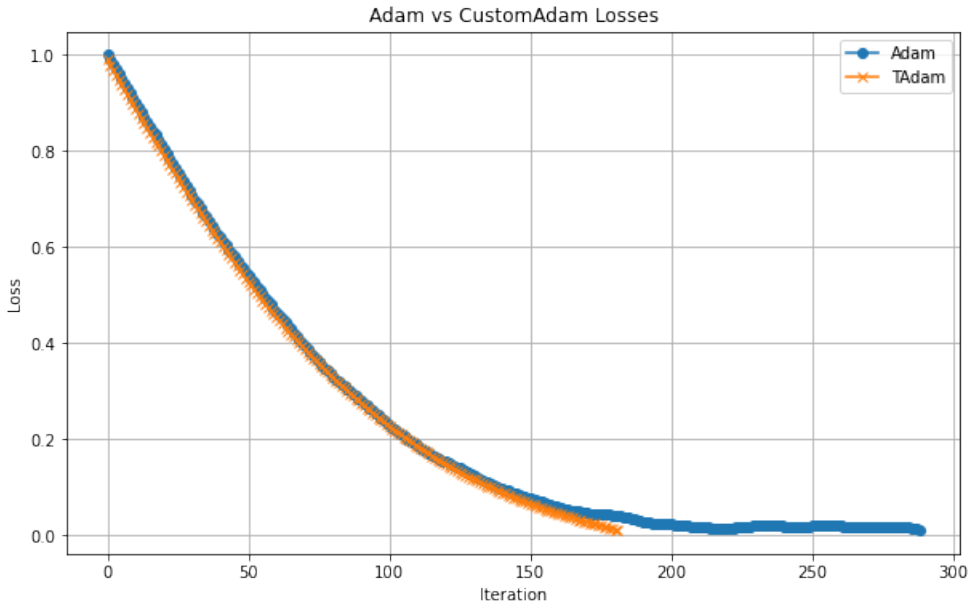

Figure 9

From Figure 9, we can clearly see that TAdam will reach convergence much faster than Adam. We run such paired model training 1000 times and calculate the difference in iteration times. Here is the result:

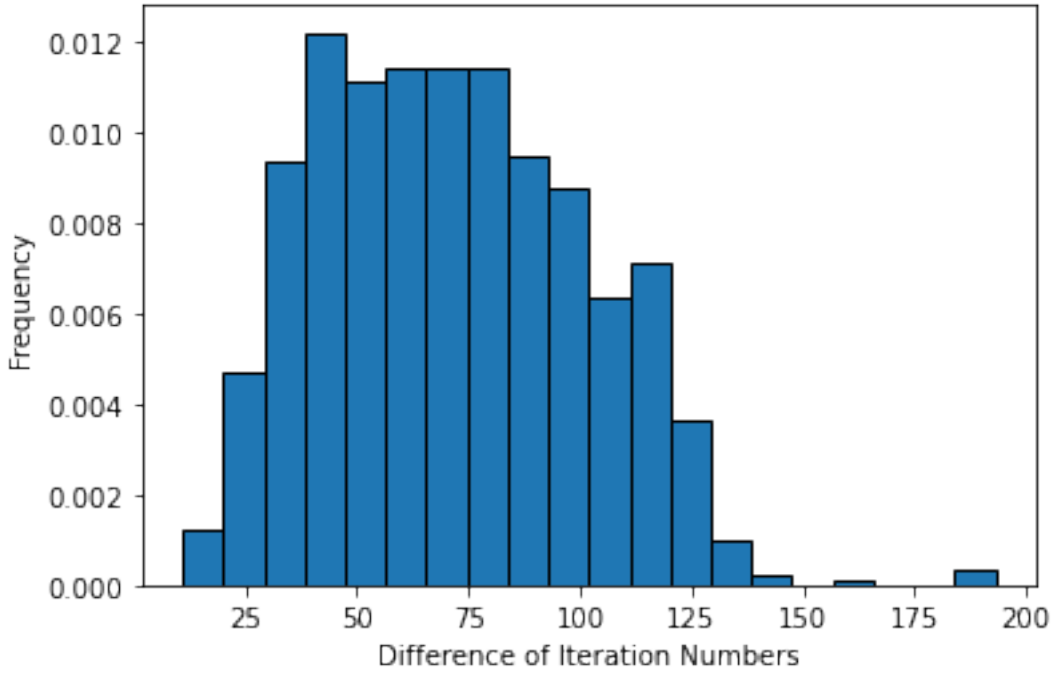

Figure 10

Figure 10 shows the difference distribution under a bell shape with a heavy tail skew to the right. The data have a mean of 71.23 and a standard deviation of 29.13. By paired T-test, the t statistics = 78.50, p-value = 0.00, indicating an almost 100 percent significant result.

If taking learning rate decay into consideration, setting $lr = 0.99^{epoch} * lr_0$, we would observe a more standard normal-shaped histogram as follow:

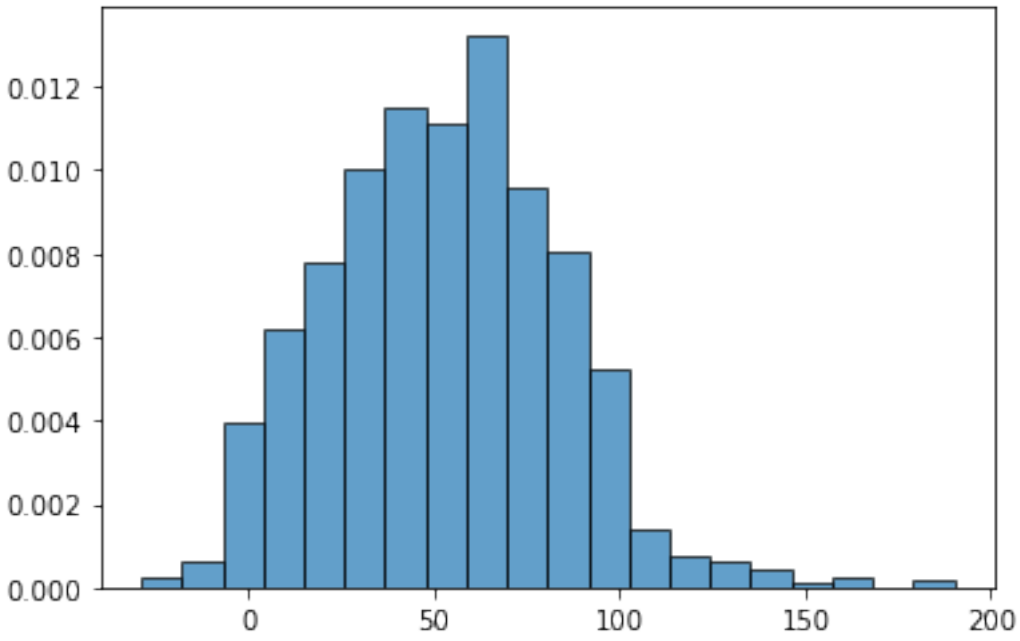

Figure 11

The result seems to be less centered, indicating that a learning rate decay implementation would lead to a high variance. This might be caused by a double time-based decay on both the learning rate and the exponential rates, causing a co-linearity problem in the algorithm.

6.1.2 Verification of Higher Accuracy

Convergence speed is just one facet of optimization. Another crucial aspect is the accuracy at which the algorithm converges to the optimal value. We set $\alpha = 0.5$ and let both algorithms run 10000 iterations, then we record the fluctuation of the loss curve. It is evident that TAdam consistently outperforms the original Adam.

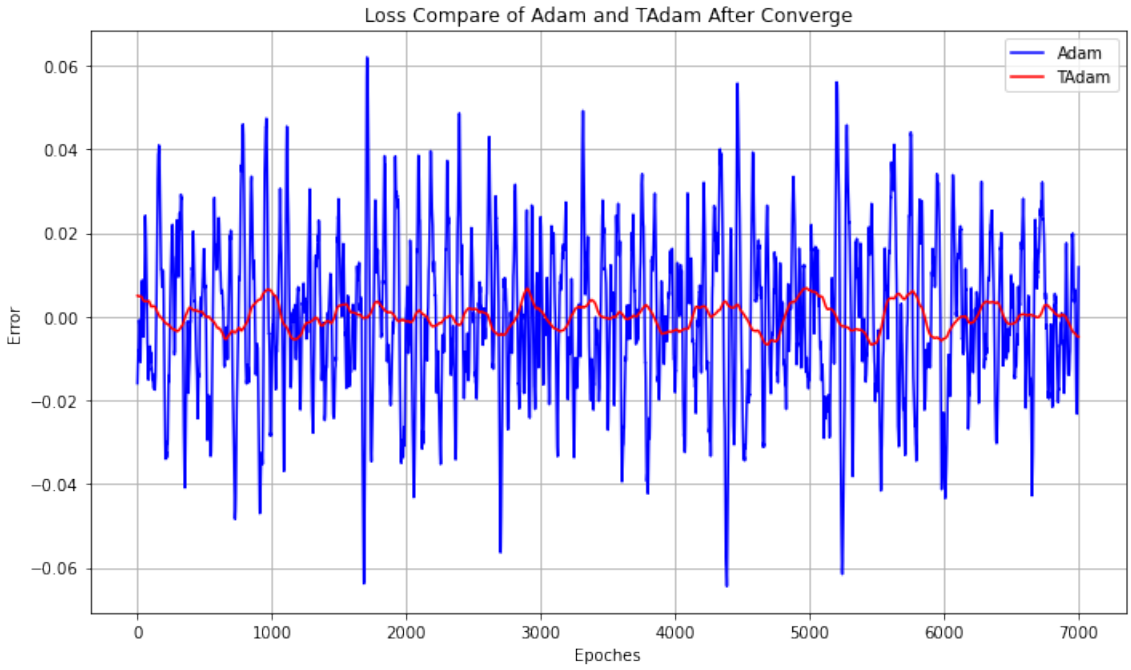

Figure 12

Figure 12 shows that the Tadam shows lower variance during the epochs after the model converges. In fact, during the training process, the lowest Adam loss is $1.16{*}10^{-4}$, while the TAdam can reach $2.28 * 10^{-8}$. TAdam demonstrates superior accuracy, with mean loss values consistently lower than those yielded by Adam.

If taking learning rate decay into consideration, setting $lr = 0.9^{epoch} * lr_0$, this trend would be more obvious:

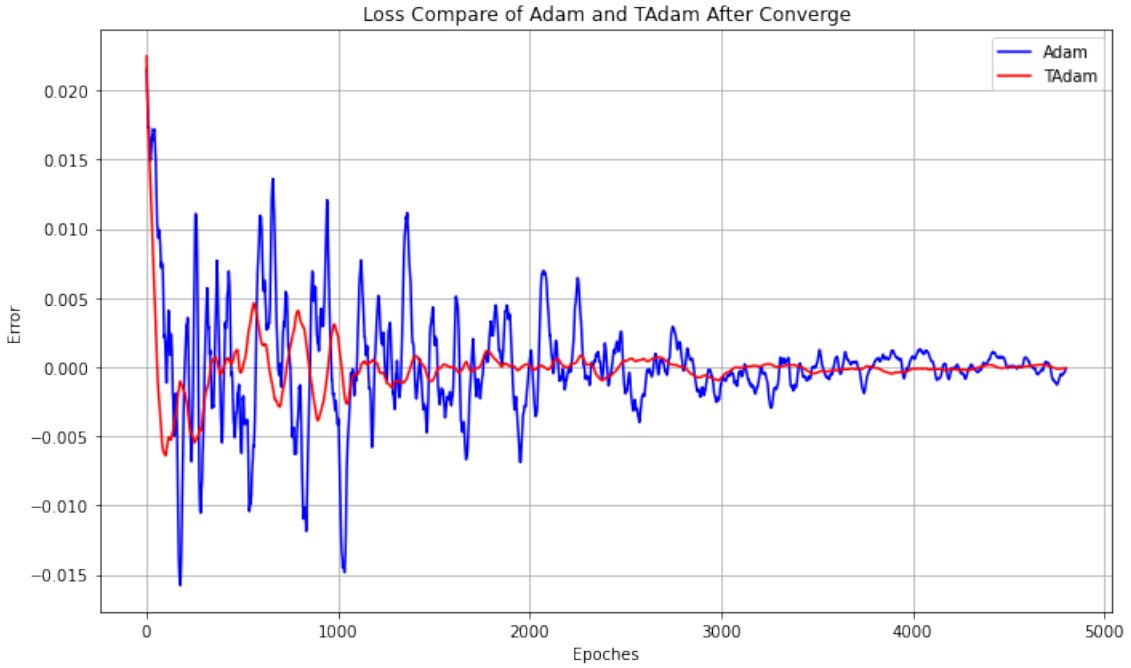

Figure 13

The difference between the true target and the estimated target will be more stable under the model trained by the TAdam, with lower variance and higher accuracy.

The learning rate time series decay would lead to a lower variance positively correlated with time, while the exponential time series decay would lead to a lower variance at each time step.

6.2 Logistic Regression Setting

$$y_i = bernoulli(p_i)$$
$$ln(\frac{p_i}{1-p_i}) = \theta_0 + \theta x_i$$

Generate every x_i as a vector with p elements each i.i.d. Norm(0,1), and calculate p_i and then y_i as the generated result. In this setting, y_i is the target, and Bernoulli entropy function is used as the loss function:

$$Loss = \sum_i -y_i log(h_\theta(x)) - (1-y_i) log(1-h_\theta(x))$$

where $h_\theta(x) = \frac{1}{1+e^{\theta_0+\theta x}}$. Taking derivatives, the gradient becomes:

$$gradient = (h_\theta(x) - y)x$$

The accuracy of the model is measured as:

$$\frac{1}{m}\sum_{i=1}^{m} \mathbb{1}_{y_i = y_i'}$$

And it is considered to converge if the model loss stays the same for two consecutive epochs.

In the logistic regression scenario, TAdam exhibited faster convergence, especially in the early stages. While both models converged after 10 epochs, TAdam's early iterations generally resulted in lower loss values. However, due to the fixed nature of discrete models, TAdam's incremental updates during late iterations were less impactful than in the expectile case.

6.2.1 Convergence Speed Comparison

For convenience, we set each x_i to be a length of 2, and default $\theta = [3, 1, 2]$. First, run both Adam and TAdam under this setting, and compare their converging epoch index as follows:

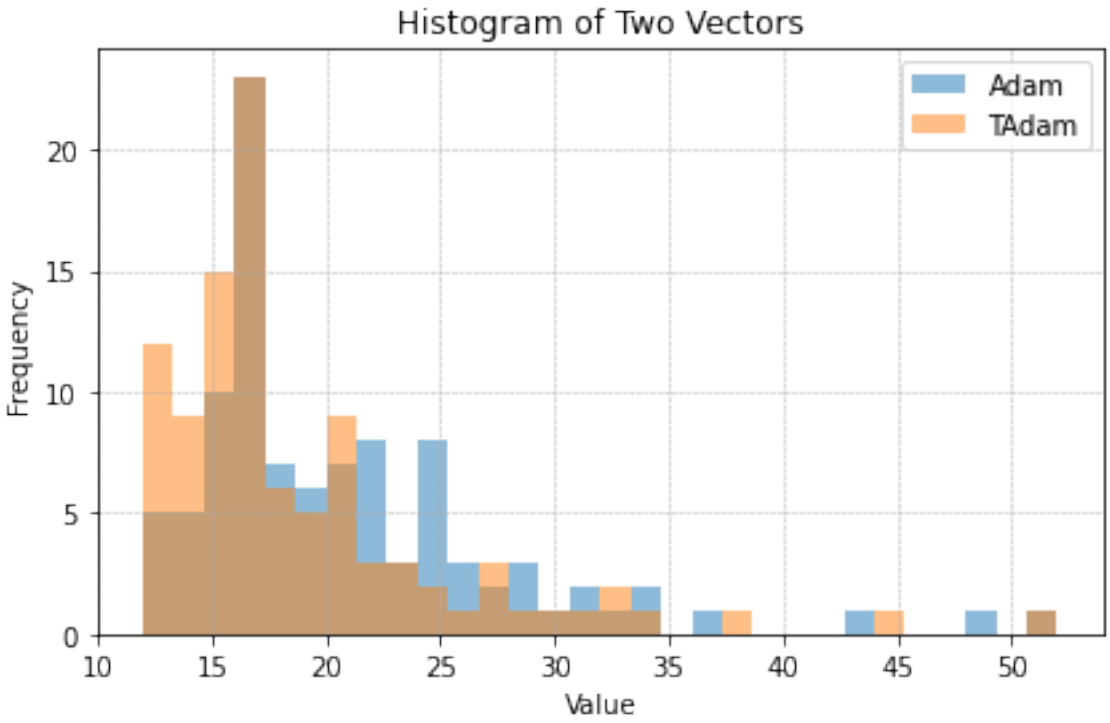

Figure 14

From Figure 14, TAdam has a higher frequency distributed at lower epoch levels. This might indicate that the TAdam's convergence would reach slightly faster than the Adam's. However, this trend is not clearly viewed in the above graph. To demonstrate the converge rate more precisely, we record the change of the loss value in 30 epochs as shown in the line chart below:

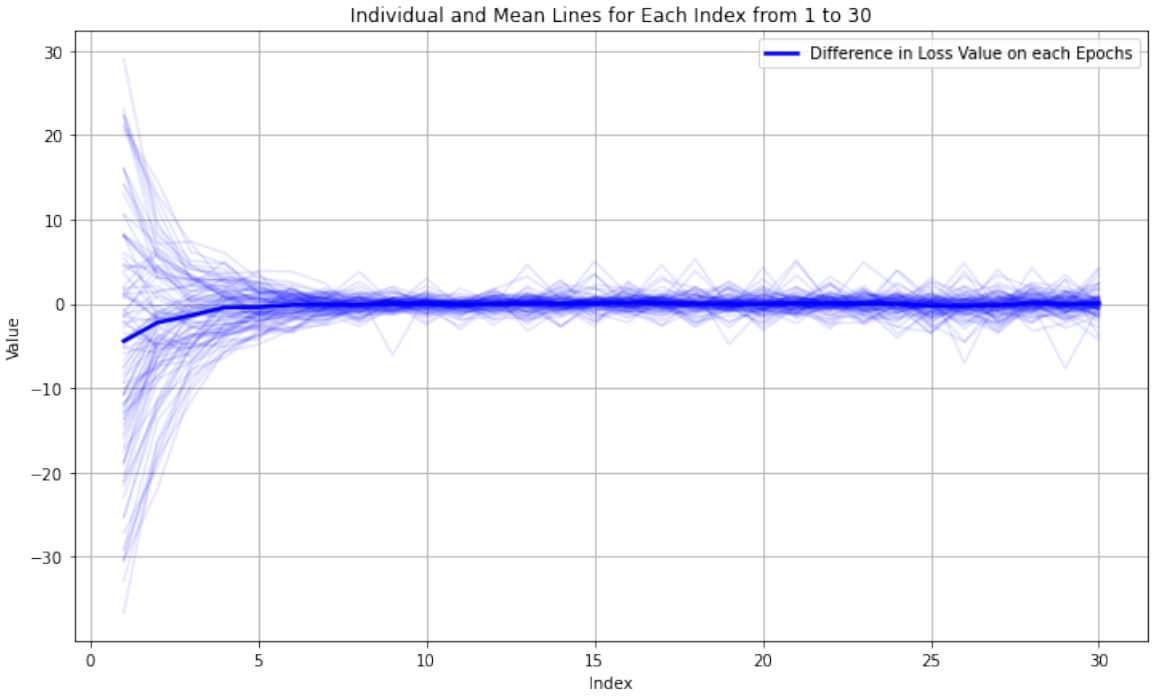

Figure 15

Figure 15 shows the difference in loss values on each epoch from 1 to 30 between Tadam and Adam. We can see that in the early stages, TAdam would generally lead to a lower loss value. With epochs running after 10, the difference is eliminated and the loss value difference fluctuates around 0. This is because both models reach convergence after 10 epochs of iterations. TAdam will have a lower loss value in early iterations, indicating that the model will converge faster than Adam.

In both scenarios, TAdam surpasses the Adam algorithm in performance. Yet, when comparing the degree of enhancement, TAdam's efficacy is marginally diminished in the logistic regression scenario compared to the expectile function. This is attributed to the fact that once the discrete model converges, its accuracy stabilizes at a fixed value. Owing to the inherent traits of the SGD-based algorithm, the eventual outcome oscillates around the convergence loss. Since this fixed accuracy value is less than one, the loss value can't zero out. Thus, TAdam's minute updates in the latter iteration epochs lose significance. On the other hand, in the expectile scenario, these minuscule updates help bring the estimated value incrementally closer to zero.

6.2.2 Enhancements in Discrete Model

A pragmatic enhancement involves refining the exponential decay function. Given the discrete model's propensity for quicker convergence, it necessitates a function exhibiting a stronger inverse correlation with the time step. This function's design should vary based on the given context, with the primary objective being to estimate the convergence iteration duration and design a function that descends to its minimum from the inception, in tandem with the time step.[15]

Another tangible improvement is integrating the decay learning rate, as demonstrated in the expectile scenario. Implementing the same parameters in the logistic model can yield outcomes with reduced loss fluctuations.

7 Conclusion

The realm of deep learning optimization is dynamic, demanding algorithms that can adapt and deliver. TAdam, with its integration of time-series dynamics, is poised to revolutionize this landscape. The empirical evidence and statistical analyses provided in this paper underscore its potential. Whether it's faster convergence speeds or superior accuracy, TAdam consistently outperforms its predecessor, the Adam algorithm. As researchers and practitioners continue to push the boundaries of deep learning models, TAdam stands ready to be the robust optimization backbone they can rely on.

A Appendix

All simulation code are included as follow.

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])


# In[147]:


import random
import sys
sys.path.append('c:/users/18383/appdata/local/packages/pythonsoftwarefoundation.python.3.9_qbz5n2kfra8p0/localcache/loca
import gym
import sys
import numpy as np
from collections import deque,namedtuple
import os
from copy import deepcopy

import torch
import torch.nn as nn
from torch.optim import Adam
from torch.optim import Optimizer


# In[148]:


env = gym.make("LunarLander-v2")


# In[149]:



s = env.reset()
for _ in range(1000):
    action = env.action_space.sample()  # Random action
    s_prime, reward,done, _ = env.step(action)
    s = s_prime
    if done:
        s = env.reset()
    env.render()


# In[150]:


env.action_space


# In[151]:


state_sz = env.observation_space.shape[0]
action_sz = env.action_space.n
print('State space: ',state_sz)
print('Action space: ',action_sz)


# In[152]:


env.action_space.sample()


# In[153]:


s = env.reset()
print(s)
s_prime,r,done,_ = env.step(0)
print(s_prime,r,done,s)


# In[ ]:




# In[154]:


n_episodes=250
batch_size=128
gamma = 0.995
lr = 0.0001
eps = 1.0
decay = 0.99
```

```
# In[155]:


class DQN(nn.Module):
    def __init__(self,hidden_sz):
        super().__init__()
        self.hidden_sz = hidden_sz

        self.fc1 = nn.Linear(state_sz,self.hidden_sz)
        self.fc2 = nn.Linear(self.hidden_sz,self.hidden_sz)
        self.fc3 = nn.Linear(self.hidden_sz,action_sz)
        self.relu = nn.ReLU()

    def forward(self,x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)

        return x


# In[156]:


replay_buffer = deque(maxlen=10000)


# In[157]:


transition = namedtuple('transition',['s_prime','reward','s','action','done'])


# In[158]:


def store(transition):
    replay_buffer.append(transition)


# In[159]:


dq_network = DQN(256)
target_network = deepcopy(dq_network)


# In[160]:


dq_network.state_dict()


# In[161]:


class CustomAdam(Optimizer):
    def __init__(self, params, lr=0.001, eps=1e-8):
        defaults = dict(lr=lr, eps=eps)
        super(CustomAdam, self).__init__(params, defaults)

    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('CustomAdam does not support sparse gradients')

                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    # Exponential moving average of gradient values
                    state['exp_avg'] = torch.zeros_like(p.data)
                    # Exponential moving average of squared gradient values
                    state['exp_avg_sq'] = torch.zeros_like(p.data)
                    # a_t term
                    state['exp_inf'] = torch.tensor(0.)
                    state['b_inf'] = torch.tensor(0.)

                exp_avg, exp_avg_sq, exp_inf,b_inf = state['exp_avg'], state['exp_avg_sq'], state['exp_inf'],state['b_in

                state['step'] += 1
                i = state['step']

                # Calculate the a_t term
                gap = 1 / ((10*i+10)**0.5)+1e-3
                gap2 = 1 / ((100*i+100)**1)+1e-5
                a_t = gap + (1 - gap)* exp_inf
                b_t = gap2 + (1 - gap2)* b_inf
```

```
                # Decay the first and second moment running average coefficient
                exp_avg.mul_(1-gap).add_(gap, grad)
                exp_avg_sq.mul_(1-gap2).addcmul_(gap2, grad, grad)

                bias_correction1 = exp_avg / a_t
                bias_correction2 = exp_avg_sq / b_t
                denom = bias_correction2.sqrt().add_(group['eps'])
                p.data.addcdiv_(-group['lr'], bias_correction1, denom)

                # Update a_t for next step
                state['exp_inf'] = a_t
                state['b_inf'] = b_t


# In[162]:


from torch.optim.lr_scheduler import StepLR
optimizer = torch.optim.SGD(dq_network.parameters(),lr=lr)
#scheduler = StepLR(optimizer, step_size=50, gamma=1)
loss_fn = nn.MSELoss()


# In[163]:


def update():

    if len(replay_buffer)<batch_size:
        return

    batch = random.sample(replay_buffer,batch_size)

    s = torch.tensor(np.array([t.s for t in batch]))
    r = torch.FloatTensor(np.array([t.reward for t in batch]))
    s_prime = torch.FloatTensor(np.array([t.s_prime for t in batch]))
    a = torch.LongTensor(np.array([t.action for t in batch])).unsqueeze(1)
    done = torch.FloatTensor(np.array([t.done for t in batch]))

    target = (r + gamma*target_network(s_prime).max(dim=1)[0]*(1-done))

    prediction = dq_network(s).gather(1,a)


    optimizer.zero_grad()

    loss = loss_fn(target.unsqueeze(1),prediction)

    loss.backward()

    optimizer.step()


# In[164]:


class Agent():
    def __init__(self,target_update_frequency=100,eps=1):

        self.eps = eps
        self.target_update_frequency = target_update_frequency
        self.target_update_counter = 0
        self.rewards = []

    def select_action(self,state,eps):

        t = np.random.random()
        if t < eps:
            a = np.random.choice(range(action_sz))
        else:
            q = dq_network(torch.FloatTensor(state))
            a = q.argmax().item()
        return a



    def run_episode(self,render):

        s = env.reset()
        done = False
        total_reward = 0.0
        self.eps = self.eps * decay
        transition_count = 0

        while not done:

            self.target_update_counter += 1
            if self.eps > 0.01:
                eps = self.eps
            else:
                eps = 0.01

            action = self.select_action(s,eps)

            s_prime,reward,done,_= env.step(action)
```

```
            store(transition(s_prime,reward,s,action,done))


            total_reward += reward

            s = s_prime

            #if render:
            #    env.render()

            update()

            done = done

            transition_count+=1

        print('Transition Count: ',transition_count)
        print('Episode Reward: ',total_reward)
        self.rewards.append(total_reward)

    def train(self):

        for k in range(n_episodes):

            render = False

            if k % 100 < 10:
                render=True

            print('Episode: ',k)
            self.run_episode(render)

            if self.target_update_counter >= self.target_update_frequency:

                self.target_update_counter = 0
                target_network.load_state_dict(dq_network.state_dict())


# In[165]:


agent = Agent()


# In[166]:


get_ipython().run_cell_magic('time', '', 'agent.train()')


# In[72]:


s = env.reset()
for _ in range(1000):
    action = dq_network(torch.tensor(s)).argmax().item()
    s_prime, reward,done, _ = env.step(action)
    s = s_prime
    if done:
        s = env.reset()
    env.render()


# In[67]:


print(np.mean(agent.rewards))
print(np.std(agent.rewards))


# In[1]:


plt.plot(adamgame, label='adam')
#plt.plot(tadamgame, label='tadam')
plt.plot(sgdgame, label='SGD')
plt.plot(asgdgame, label='ASGD')
plt.title("Comparison on Different Optimizations")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.show()


# In[73]:


asgdgame = [np.mean(agent.rewards[i-50:i]) for i in range(50,250)]


import math
```

```
import torchtext
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchtext.data.utils import get_tokenizer
from collections import Counter
from torchtext.vocab import vocab,Vocab
from torchtext.utils import download_from_url, extract_archive
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, TensorDataset
from torchtext.transforms import ToTensor
from torch import Tensor
from torch.nn import TransformerEncoder,TransformerDecoder,TransformerEncoderLayer,TransformerDecoderLayer
import io
import time


# In[2]:


url_base = 'https://raw.githubusercontent.com/multi30k/dataset/master/data/task1/raw/'
train_urls = ('train.fr.gz', 'train.en.gz')
val_urls = ('val.de.gz', 'val.en.gz')
test_urls = ('test_2016_flickr.de.gz', 'test_2016_flickr.en.gz')


# In[3]:


train_filepaths = [extract_archive(download_from_url(url_base + url))[0] for url in train_urls]
val_filepaths = [extract_archive(download_from_url(url_base + url))[0] for url in val_urls]
test_filepaths = [extract_archive(download_from_url(url_base + url))[0] for url in test_urls]


# In[4]:


import spacy


# In[5]:


import pandas as pd
df = pd.read_csv('questions_easy.csv')
fr_tokenizer = get_tokenizer('spacy', language='fr_core_news_sm')
en_tokenizer = get_tokenizer('spacy', language='en_core_web_sm')


# In[7]:


counter = Counter()
for string_ in df['en']:
    counter.update(en_tokenizer(string_))

en_vocab =  vocab(counter,specials=['<unk>','<pad>','<bos>','<eos>'])

counter = Counter()
for string_ in df['fr']:
    counter.update(fr_tokenizer(string_))

fr_vocab =  vocab(counter,specials=['<unk>','<pad>','<bos>','<eos>'])

fr_vocab.set_default_index(0)
en_vocab.set_default_index(0)


# In[8]:


def data_process(filepaths):
    raw_de_iter = iter(io.open(filepaths[0],encoding='utf8'))
    raw_en_iter = iter(io.open(filepaths[1],encoding='utf8'))
    fr_data = []
    en_data = []
    for (raw_de,raw_en) in zip(raw_de_iter,raw_en_iter):
            de_data.append([2]+de_vocab(de_tokenizer(raw_de.rstrip('n')))+[3])
            en_data.append([2]+en_vocab(en_tokenizer(raw_en.rstrip('n')))+[3])
    return  fr_data,en_data


# In[9]:


raw_fr_iter = iter(df['fr'])
raw_en_iter = iter(df['en'])
fr_data = []
en_data = []
for (raw_fr,raw_en) in zip(raw_fr_iter,raw_en_iter):
        fr_data.append([2]+fr_vocab(fr_tokenizer(raw_fr.rstrip('n')))+[3])
        en_data.append([2]+en_vocab(en_tokenizer(raw_en.rstrip('n')))+[3])
train_data = fr_data, en_data
val_data = data_process(val_filepaths)
test_data = data_process(test_filepaths)
```

```
# In[10]:


to_tensor = ToTensor(padding_value=1)


# In[11]:


fr_train_data = to_tensor(train_data[0])
en_train_data = to_tensor(train_data[1])
fr_val_data = to_tensor(val_data[0])
en_val_data = to_tensor(val_data[1])
fr_test_data = to_tensor(test_data[0])
en_test_data = to_tensor(test_data[1])


# In[12]:


fr_train_data.size()


# In[13]:


en_train_data.size()


# In[14]:


train_ds = TensorDataset(fr_train_data,en_train_data)
val_ds = TensorDataset(fr_val_data,en_val_data)
test_ds = TensorDataset(fr_test_data,en_test_data)


# In[15]:


bs = 128


# In[16]:


train_dl = DataLoader(train_ds,batch_size=bs,shuffle=True)
val_dl = DataLoader(val_ds,batch_size=bs,shuffle=True)
test_dl = DataLoader(test_ds,batch_size=bs,shuffle=True)


# In[17]:


class Seq2SeqTransformer(nn.Module):
    def __init__(self, num_encoder_layers: int, num_decoder_layers: int,
                 emb_size: int, src_vocab_size: int, tgt_vocab_size: int,
                 dim_feedforward:int = 512, dropout:float = 0.1):
        super().__init__()
        encoder_layer = TransformerEncoderLayer(d_model=emb_size, nhead=NHEAD,
                                                dim_feedforward=dim_feedforward,batch_first=True)
        self.transformer_encoder = TransformerEncoder(encoder_layer, num_layers=num_encoder_layers)
        decoder_layer = TransformerDecoderLayer(d_model=emb_size, nhead=NHEAD,
                                                dim_feedforward=dim_feedforward,batch_first=True)
        self.transformer_decoder = TransformerDecoder(decoder_layer, num_layers=num_decoder_layers)

        self.generator = nn.Linear(emb_size, tgt_vocab_size)
        self.src_tok_emb = TokenEmbedding(src_vocab_size, emb_size)
        self.tgt_tok_emb = TokenEmbedding(tgt_vocab_size, emb_size)
        self.positional_encoding = PositionalEncoding(emb_size, dropout=dropout)

    def forward(self, src: Tensor, trg: Tensor, src_mask: Tensor,
                tgt_mask: Tensor, src_padding_mask: Tensor,
                tgt_padding_mask: Tensor, memory_key_padding_mask: Tensor):
        src_emb = self.positional_encoding(self.src_tok_emb(src))
        tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg))
        memory = self.transformer_encoder(src_emb, src_mask, src_padding_mask)
        outs = self.transformer_decoder(tgt_emb, memory,tgt_mask, None,
                                        tgt_padding_mask, memory_key_padding_mask)
        return self.generator(outs)

    def encode(self, src: Tensor, src_mask: Tensor):
        return self.transformer_encoder(self.positional_encoding(
                            self.src_tok_emb(src)), src_mask)

    def decode(self, tgt: Tensor, memory: Tensor, tgt_mask: Tensor):
        return self.transformer_decoder(self.positional_encoding(
                          self.tgt_tok_emb(tgt)), memory,
                          tgt_mask)

class PositionalEncoding(nn.Module):

    def __init__(self,emb_size,dropout,max_len=5000):
        super().__init__()
        den = torch.exp(-torch.arange(0,emb_size,2) * math.log(10000)/emb_size)
```

```
        pos = torch.arange(0,max_len).reshape(max_len,1)
        pos_embedding = torch.zeros((max_len,emb_size))
        pos_embedding[:,0::2] = torch.sin(pos * den)
        pos_embedding[:,1::2] = torch.cos(pos * den)
        pos_embedding = pos_embedding.unsqueeze(-2)

        self.dropout = nn.Dropout(dropout)
        self.register_buffer('pos_embedding',pos_embedding)

    def forward(self,token_embedding):
        return self.dropout(token_embedding + self.pos_embedding[:token_embedding.size(0),:])




# In[18]:


class TokenEmbedding(nn.Module):
    def __init__(self, vocab_size: int, emb_size):
        super(TokenEmbedding, self).__init__()
        self.embedding = nn.Embedding(vocab_size, emb_size)
        self.emb_size = emb_size
    def forward(self, tokens: Tensor):
        return self.embedding(tokens.long()) * math.sqrt(self.emb_size)


# In[19]:


def generate_square_subsequent_mask(sz):
    mask = (torch.triu(torch.ones((sz,sz),device='cuda')) == 1).transpose(0,1)

    mask = mask.float().masked_fill(mask==0,float('-inf')).masked_fill(mask==1,float(0.0))
    return mask


# In[20]:


def create_mask(src,tgt):
    src_seq_len = src.shape[1]
    tgt_seq_len = tgt.shape[1]

    tgt_mask = generate_square_subsequent_mask(tgt_seq_len).cuda()
    src_mask = torch.zeros((src_seq_len,src_seq_len),device='cuda').type(torch.bool)

    src_padding_mask = (src == 1).cuda()
    tgt_padding_mask = (tgt == 1).cuda()

    return src_mask, tgt_mask, src_padding_mask,tgt_padding_mask


# In[21]:


SRC_VOCAB_SIZE = len(fr_vocab)
TGT_VOCAB_SIZE = len(en_vocab)
EMB_SIZE = 64
NHEAD = 8
FFN_HID_DIM = 512
BATCH_SIZE = 128
NUM_ENCODER_LAYERS = 3
NUM_DECODER_LAYERS = 3
NUM_EPOCHS = 50
DEVICE = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')


# In[37]:


transformer = Seq2SeqTransformer(NUM_ENCODER_LAYERS, NUM_DECODER_LAYERS,
                                 EMB_SIZE, SRC_VOCAB_SIZE, TGT_VOCAB_SIZE,
                                 FFN_HID_DIM)


# In[38]:


for p in transformer.parameters():
    if p.dim() > 1:
        nn.init.xavier_uniform_(p)


# In[39]:


transformer = transformer.cuda()


# In[40]:


loss_fn = torch.nn.CrossEntropyLoss(ignore_index=1)
```

```
optimizer = torch.optim.Adam(transformer.parameters(),lr=0.0001,betas=(0.9,0.98),eps=1e-9)


# In[41]:


def train_epoch(model, train_dl, optimizer):
    model.train()
    losses = 0
    for idx, (src, tgt) in enumerate(train_dl):
        src = src.cuda()
        tgt = tgt.cuda()

        tgt_input = tgt[:,:-1]

        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tgt_input)

        logits = model(src, tgt_input, src_mask,
                tgt_mask, src_padding_mask,
                tgt_padding_mask, src_padding_mask)

        optimizer.zero_grad()

        tgt_out = tgt[:, 1:]
        loss = loss_fn(logits.reshape(-1, logits.shape[-1]), tgt_out.reshape(-1))
        loss.backward()

        optimizer.step()
        losses += loss.item()

    return losses / len(train_dl)



# In[42]:


def evaluate(model,val_dl):
    model.eval()
    losses = 0

    for idx, (src,tgt) in enumerate(val_dl):
        src = src.cuda()
        tgt = tgt.cuda()

        tgt_input =tgt[:,:-1]

        src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tgt_input)

        logits = model(src, tgt_input, src_mask,
                tgt_mask, src_padding_mask,
                tgt_padding_mask, src_padding_mask)

        optimizer.zero_grad()

        tgt_out = tgt[:,1:]

        loss = loss_fn(logits.reshape(-1, logits.shape[-1]), tgt_out.reshape(-1))

        losses += loss.item()

    return losses/len(val_dl)

# In[43]:


for epoch in range(19):
    start_time = time.time()
    train_loss = train_epoch(transformer,train_dl,optimizer)

    end_time = time.time()

    val_loss = evaluate(transformer,val_dl)

    print(f'Epoch: {epoch}, Train loss: {train_loss:.3f}, Val loss: {val_loss:.3f}, '
          f'Epoch time = {(end_time - start_time):.3f}s ')


# In[29]:


def greedy_decode(model, src, src_mask, max_len, start_symbol):
    src = src.cuda()
    src_mask = src_mask.cuda()
    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(2).type(torch.long).cuda()
    for i in range(max_len-1):

        memory = memory.cuda()
        memory_mask = torch.zeros(ys.shape[1], memory.shape[1]).cuda().type(torch.bool)
        tgt_mask = (generate_square_subsequent_mask(ys.size(1))
                                    .type(torch.bool)).cuda()
        out = model.decode(ys, memory, tgt_mask).squeeze(0)[-1]
```

```
        prob = model.generator(out)
        next_word = prob.argmax()
        next_word = next_word.item()

        ys = torch.cat([ys,
                        torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
        if next_word == 3:
            break
    return ys


def translate(model, src, src_vocab, tgt_vocab, src_tokenizer):
    model.eval()
    tokens = [2]+de_vocab(de_tokenizer(src))+[3]
    num_tokens = len(tokens)
    src = (torch.LongTensor(tokens).reshape(1,num_tokens))
    src_mask = (torch.zeros(num_tokens, num_tokens)).type(torch.bool)
    tgt_tokens = greedy_decode(model, src, src_mask, max_len=num_tokens + 5, start_symbol=2).flatten()
    return " ".join([tgt_vocab.get_itos()[tok] for tok in tgt_tokens]).replace("<bos>", "").replace("<eos>", "")


# In[45]:


output = translate(transformer, "Eine Gruppe von Menschen steht vor einem Flughafen .", de_vocab, en_vocab, de_tokenizer
print(output)

import torch
import math
import torchvision
from torchvision import datasets, transforms
from torch import nn, optim
from tqdm import tqdm
from torch.optim.lr_scheduler import ExponentialLR


# In[19]:


import torch
from torch.optim import Optimizer

class AdaMax(Optimizer):
    def __init__(self, params, lr=0.01, betas=(0.9, 0.999), eps=1e-10):
        defaults = dict(lr=lr, betas=betas, eps=eps)
        super(AdaMax, self).__init__(params, defaults)

    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('AdaMax does not support sparse gradients')

                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    # Exponential moving average of gradient values
                    state['exp_avg'] = torch.zeros_like(p.data)
                    # Exponentially weighted infinity norm
                    state['exp_infinity'] = torch.zeros_like(p.data)

                exp_avg, exp_infinity = state['exp_avg'], state['exp_infinity']
                beta1, beta2 = group['betas']

                # Add op to update the steps
                state['step'] += 1

                # Decay the first and second moment running average coefficient
                exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
                torch.max(exp_infinity.mul_(beta2), grad.abs(), out=exp_infinity)

                lr = group['lr'] / (1 - beta1 ** state['step'])

                p.data.addcdiv_(-lr, exp_avg, exp_infinity.add_(group['eps']))


# In[20]:


# Transform to normalize the data
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])

# Download and load the training data
trainset = datasets.MNIST('~/.pytorch/MNIST_data/', download=True, train=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)


# In[76]:
```

```
class CustomAdam(Optimizer):
    def __init__(self, params, lr=0.001, eps=1e-8):
        defaults = dict(lr=lr, eps=eps)
        super(CustomAdam, self).__init__(params, defaults)

    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('CustomAdam does not support sparse gradients')

                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    # Exponential moving average of gradient values
                    state['exp_avg'] = torch.zeros_like(p.data)
                    # Exponential moving average of squared gradient values
                    state['exp_avg_sq'] = torch.zeros_like(p.data)
                    # a_t term
                    state['exp_inf'] = torch.tensor(0.)
                    state['b_inf'] = torch.tensor(0.)

                exp_avg, exp_avg_sq, exp_inf,b_inf = state['exp_avg'], state['exp_avg_sq'], state['exp_inf'],state['b_in

                state['step'] += 1
                i = state['step']

                # Calculate the a_t term
                gap = 1-1 / (1+math.exp((-i-150)/100))+1e-3
                gap2 = 1-1 / (1+math.exp((-i-300)/100))+1e-5
                a_t = gap + (1 - gap)* exp_inf
                b_t = gap2 + (1 - gap2)* b_inf

                # Decay the first and second moment running average coefficient
                exp_avg.mul_(1-gap).add_(gap, grad)
                exp_avg_sq.mul_(1-gap2).addcmul_(gap2, grad, grad)

                bias_correction1 = exp_avg / a_t
                bias_correction2 = exp_avg_sq / b_t
                denom = bias_correction2.sqrt().add_(group['eps'])
                p.data.addcdiv_(-group['lr'], bias_correction1, denom)

                # Update a_t for next step
                state['exp_inf'] = a_t
                state['b_inf'] = b_t


# In[35]:


# Define a function that returns a new model
def create_model():
    model = nn.Sequential(nn.Linear(784, 64),
                          nn.ReLU(),
                          nn.Linear(64, 10),
                          nn.LogSoftmax(dim=1))
    return model

# Define a loss function
criterion = nn.NLLLoss()



# In[43]:


# Dictionaries to store the optimizers and losses for each optimizer
#optimizers = {'SGD': optim.SGD, 'Adam': optim.Adam, 'ASGD': optim.ASGD,"AdaMax":AdaMax}
#losses = {'SGD': [], 'Adam': [], 'ASGD': [],"AdaMax":[]}
from torch.optim.lr_scheduler import LambdaLR
optimizers = {'Adam': optim.Adam,"CustomAdam":CustomAdam,'SGD': optim.SGD, 'ASGD': optim.ASGD}
losses = { 'Adam': [],"CustomAdam":[],'SGD': [], 'ASGD': []}

# Loop over all optimizers
for name, optimizer_class in optimizers.items():
    model = create_model()
    optimizer = optimizer_class(model.parameters(), lr=0.01)
    lr_lambda = lambda step: 0.1 / ((step+1) ** 0.7)
    scheduler = LambdaLR(optimizer, lr_lambda=lr_lambda)

    for epoch in tqdm(range(30)):  # loop over the dataset multiple times
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            # flatten inputs
            inputs = inputs.view(inputs.shape[0], -1)

            # zero the parameter gradients
            optimizer.zero_grad()
```

```
            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()

        losses[name].append(running_loss / len(trainloader))
        scheduler.step()

    print(f"{name} loss: {losses[name]}")

print('Finished Training')


# In[44]:


plt.figure(figsize=(10, 6))
for optimizer_name, loss_values in losses.items():
    plt.plot(loss_values, label=optimizer_name)

plt.title('Loss Curves for Different Optimizers with Learning Rate Decay')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()



# In[115]:


class AdamInterface(Optimizer):
    def __init__(self, params, lr=0.001, eps=1e-8,gap = 0.1,gap2=0.001):
        defaults = dict(lr=lr, eps=eps)
        self.gap = gap
        self.gap2 = gap2
        super(AdamInterface, self).__init__(params, defaults)

    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('CustomAdam does not support sparse gradients')

                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    # Exponential moving average of gradient values
                    state['exp_avg'] = torch.zeros_like(p.data)
                    # Exponential moving average of squared gradient values
                    state['exp_avg_sq'] = torch.zeros_like(p.data)
                    # a_t term
                    state['exp_inf'] = torch.tensor(0.)
                    state['b_inf'] = torch.tensor(0.)

                exp_avg, exp_avg_sq, exp_inf,b_inf = state['exp_avg'], state['exp_avg_sq'], state['exp_inf'],state['b_in

                state['step'] += 1
                i = state['step']

                # Calculate the a_t term
                gap = self.gap
                gap2 = self.gap2
                a_t = gap + (1 - gap)* exp_inf
                b_t = gap2 + (1 - gap2)* b_inf

                # Decay the first and second moment running average coefficient
                exp_avg.mul_(1-gap).add_(gap, grad)
                exp_avg_sq.mul_(1-gap2).addcmul_(gap2, grad, grad)

                bias_correction1 = exp_avg / a_t
                bias_correction2 = exp_avg_sq / b_t
                denom = bias_correction2.sqrt().add_(group['eps'])
                p.data.addcdiv_(-group['lr'], bias_correction1, denom)

                # Update a_t for next step
                state['exp_inf'] = a_t
                state['b_inf'] = b_t


# In[124]:


optimizers = {'AdamInterface': AdamInterface}
```

```
# Loop over all optimizers
g1s = [0.001,0.01,0.05,0.1,0.2]
g2s = [0.00001,0.0001,0.0005,0.001,0.002]
for j in range(5):
    losses = {'AdamInterface': []}
    for name, optimizer_class in optimizers.items():
        model = create_model()
        optimizer = optimizer_class(model.parameters(), lr=0.01,gap = g1s[j],gap2 = g2s[j])

        for epoch in tqdm(range(10)):  # loop over the dataset multiple times
            running_loss = 0.0
            for i, data in enumerate(trainloader, 0):
                # get the inputs; data is a list of [inputs, labels]
                inputs, labels = data

                # flatten inputs
                inputs = inputs.view(inputs.shape[0], -1)

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward + backward + optimize
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()

                # print statistics
                running_loss += loss.item()

            losses[name].append(running_loss / len(trainloader))

        print(f"Decay set:({g1s[j]}, {g2s[j]}) {name} loss: {losses[name]}")

print('Finished Training')
```

References

[1] Amari, Shun-Ichi. *Natural gradient works efficiently in learning*. Neural computation, 10(2):251–276, 1998

[2] Robert Mansel Gower, Nicolas Loizou, Xun Qian, Alibek Sailanbayev, Egor Shulgin, Peter Richtárik. *SGD: General Analysis and Improved Rates*. PMLR 97:5200-5209, 2019. URL: http://proceedings.mlr.press/v97/qian19b

[3] J. Fernando Hernandez-Garcia, Richard S. Sutton. *Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target*. Feb 2019. URL: https://arxiv.org/abs/1901.07510

[4] Zaipeng Xie; Yufeng Zhang; Pengfei Shao; Weiyi Zhao. *QDN: An Efficient Value Decomposition Method for Cooperative Multi-agent Deep Reinforcement Learning*. 2022 IEEE 34th International Conference on Tools with Artificial Intelligence (ICTAI), Macao, China, 2022, pp. 1204-1211, doi: 10.1109/ICTAI56018.2022.00183.

[5] L.O. Chua, T. Roska. *The CNN paradigm*. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, March 1993. URL: https://ieeexplore.ieee.org/document/222795

[6] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. *Imagenet classification with deep convolutional neural networks*. Biometrika, 77(4):669–687, 1990. ISSN 0006-3444. doi: 10.2307/ 2337091. URL https://www.jstor.org/stable/2337091. Publisher: [Oxford University Press, Biometrika Trust].

[7] Diederik P. Kingma, Jimmy Lei Ba. *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*. ICLR, 2015. URL: https://arxiv.org/pdf/1412.6980.pdf

[8] Gregory Cohen; Saeed Afshar; Jonathan Tapson; André van Schaik. *EMNIST: Extending MNIST to handwritten letters*. 2017 International Joint Conference on Neural Networks (IJCNN), 2017. URL: https://ieeexplore.ieee.org/abstract/document/7966217/authorsauthors

[9] Wanrong Zhua, Xi Chenb, and Wei Biao Wu. *Online Covariance Matrix Estimation in Stochastic Gradient Descent*. JOURNAL OF THE AMERICAN STATISTICAL ASSOCIATION, 2023. URL: https://www.tandfonline.com/doi/epdf/10.1080/01621459.2021.1933498

[10] Xinyi Wang, Mei-jen Lee, Qing Zhao, Lang Tong. *Non-parametric Probabilistic Time Series Forecasting via Innovations Representation*. June, 2023. URL: https://arxiv.org/pdf/2306.03782.pdf

[11] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. https://doi.org/10.48550/arXiv.1212.5701 Focus to learn more

[12] Wangpeng An; Haoqian Wang; Yulun Zhang; Qionghai Dai. *Exponential decay sine wave learning rate for fast deep neural network training*. 2017 IEEE Visual Communications and Image Processing (VCIP), St. Petersburg, FL, USA, 2017, pp. 1-4, doi: 10.1109/VCIP.2017.8305126.

[13] Duchi, John, Hazan, Elad, and Singer, Yoram. *Adaptive subgradient methods for online learning and stochastic optimization*. The Journal of Machine Learning Research, 12:2121–2159, 2011.

[14] Fabian Sobotka, Thomas Kneib. *Geoadditive expectile regression*. December 2010. URL: https://doi.org/10.1016/j.csda.2010.11.015

[15] Xi Chen, Jason D. Lee, Xin T. Tong, Yichen Zhang. *Statistical inference for model parameters in stochastic gradient descent*. Ann. Statist. 48 (1) 251 - 273, February 2020. URL: https://projecteuclid.org/journals/annals-of-statistics/volume-48/issue-1/Statistical-inference-for-model-parameters-in-stochastic-gradient-descent/10.1214/18-AOS1801.full