$$\eta = 0.0001 \quad \eta = 0.0001 \quad \eta = 0.1 \quad \eta = 0.1 \quad \eta = 0.01 \quad \eta = 0.01 \quad \eta = 0.0001 \quad \eta = 0.0001$$

Optimization - Final Project

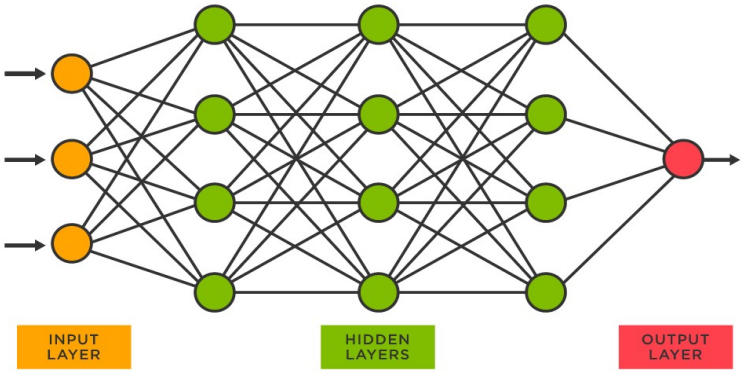
Mor Hale and Yona Coscas

February 12, 2022

1 Introduction

The Problem

An Artificial Neural Networks (ANNs or NNs) is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. An artificial neuron receives a signal then processes it and can signal neurons connected to it. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. Typically, neurons are aggregated into layers. Signals travel from the first layer (the input layer),through the hidden layers, to the last layer (the output layer).



In supervised learning, neural network learn (or are trained) by processing examples, each of which contains a known "input" and "result," forming probability-weighted associations between the two, which are stored within the data structure of the net itself. The training of a neural network from a given example is usually conducted by determining the difference between the processed output of the network (often a prediction) and a target output. This difference is the error or loss. The network then optimize its weighted associations according to a learning rule and using this error value. Successive adjustments will cause the neural network to produce output which is increasingly similar to the target output. After a sufficient number of these adjustments the training can be terminated based upon certain criteria.

The setting for a simple neural network optimization setup, is firstly a training dataset D, secondly a model $\hat{f}_\theta(x)$ with parameters θ that can be updated and lastly an loss function $L(\theta)$ that represents the accuracy of the model. The goal of the optimization process is to find the parameters that minimize the loss function on the training set.

The Optimizer

ADAM, stands for ADAptive Moment estimation, is an gradient based optimization algorithm for stochastic differentiable functions. In this algorithm we use only first order derivitives so it is better then second-order methods such as Newton's. It is also a stochastic algorithm, i.e., it does not require calculating the gradients for the whole dataset. Instead it samples randomly from the dataset and optimize the parameters based on this sample, which reduce the computation time.

The stochastic approach has been the main solution for deep learning models.

[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 $L_t(\theta)$, 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, and set to be 0.9, 0.999 respectively according to the author [*****add reference*****]. 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:

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

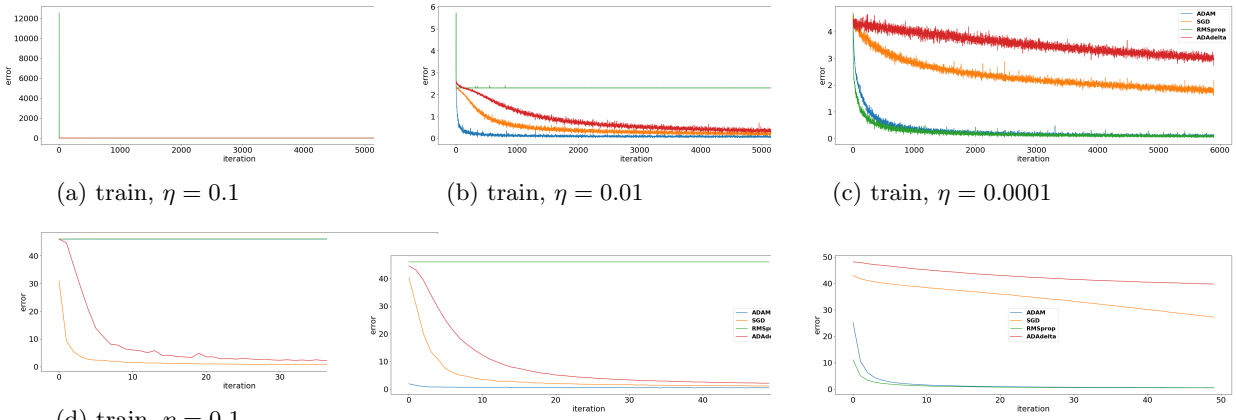
then we can use those moment estimates to update the parameter θ:

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

While η being the learning rate and ϵ is required to prevent numerical errors with very small standard deviations. The learning rate is a tuning parameter that determines the step size at each iteration while moving toward the minimum of the loss function.

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)



(a) train, $\eta = 0.1$ (b) train, $\eta = 0.01$ (c) train, $\eta = 0.0001$

(d) train, $\eta = 0.1$ (e) test, $\eta = 0.01$ (f) test, $\eta = 0.0001$

Figure 1: Three simple graphs

2 Results

We evaluated four optimizers in different settings on a neural network with two 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 the following figure we can see the results of 3 experiments with learning rates of 0.1, 0.01, 0.0001. In all of them the batch size is 512 and all lines were smoothed for better visual evaluation.

In figure 1(a) We can see that it converge to a slightly smaller loss, but it took almost twice as long to get this value. We also can see that SGD suffer greatly from decreasing the learning rate since has a bad convergence rate. Adadelta also suffer and provide the worst convergence rate. If we continue to iterate we probably would get a smaller loss value but it would take days.

In figure 1(b) We can see Adadelta and SGD has a poor convergance, while RMSprop provide a very good convergance and is very close to the results from ADAM, but still ADAM outperform them all.

In figure 1(c), with the lowest learning rate, We can see SGD and Adadelta converge to a similar value as ADAM, but still ADAM provide a better convergence rate. We also can see that RMSprop does not learn properly and got stuck on high loss value, 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.

* * * add explanations for test graphs * * *

Add here the results where Adam is not good

3 Discussions

In [6], the authors argue that althogh ADAM provide extremely good results on deep learning optimization i.e. trying to optimize a loss surface, it provide extremey poor results on other optimization problems. The key difference between deep learning model optimization, is the shear number of parameters to optimize (several millions), and that derivatives of the error function are ususally ill conditioned [4], thus the error surface has many saddle points, thus from one hand convex optimization is not feasible in this cases, and in the other hand second order optimization is not feasible dus 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.

4 More about Other Algorithms

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 apllying 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 make them perform better on non-convex problems, the stochasity of the formulation makes it easy for them to stay away from local minimas, this also make 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 covex 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.
The stochastic optimization process with Stochastic Gradient Descent (SGD), 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: $\theta := \theta - \frac{\eta}{n} \sum_{i=1}^{n} \nabla L_i(\theta)$

Mini batch Gradient Descent is the combination of BGD and SGD, we do itterate over the all dataset in random batches, but update the parameters each batch, thus acheiving convergence much faster and still have the property of stochasity. The key withdraw of all variants of gradient descent comparimg 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 chalenge 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 occurence of features.

Adadelta

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. Therefor 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 widley 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.

5 Miscellanea

In [6], the authors argue that althogh ADAM provide extremely good results on deep learning optimization i.e. trying to optimize a loss surface, it provide extremey poor results on other optimization problems. The key difference between deep learning model optimization, is the shear number of parameters to optimize (several millions), and that derivatives of the error function are ususally ill conditioned [4], thus the error surface has many saddle points, thus from one hand convex optimization is not feasible in this cases, and in the other hand second order optimization is not feasible dus 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.

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.

6 The Code

```
def loss_function(x):
    return x ** 3 - 3 * x ** 2 + 4 * x


def grad_function(x):
    return 3 * x ** 2 - 6 * x


def check_convergence(theta0, theta1):
    return all(theta0 == theta1)


theta = np.random.rand(5)
num_iter = 10
converged = False
adam = my_adam.adam_optimizer(theta)
d_theta = np.zeros_like(theta)
old_theta = np.zeros_like(theta)
loss_values = []

while not converged:
    for i, theta_i in enumerate(theta):
        d_theta[i] = grad_function(theta_i)
        old_theta = theta_i
    theta = adam.step(num_iter, theta, d_theta)
    loss_values.append(loss_function(theta_i))
    if check_convergence(theta, old_theta):
        print('number of iterations until ADAM converged: ' + str(num_iter))
        break
    else:
        print('iteration number ' + str(num_iter) + ': weight = ' + str(theta))
        num_iter += 1
```

```
import numpy as np


class adam_optimizer:
    def __init__(self, theta, step_size=0.01, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.mean_d_theta = np.zeros_like(theta)
        self.u_variance_d_theta = np.zeros_like(theta)
        self.beta1 = beta1  # beta1 is the exponential decay of the rate for
                                the first moment estimates
        self.beta2 = beta2  # beta2 is the exponential decay rate for
                                the second-moment estimates
        self.epsilon = epsilon  # prevent zero-division
        self.step_size = step_size

    def step(self, num_iter, theta, d_theta):
        mean_d_theta_corr = np.zeros_like(theta)
        u_variance_d_theta_corr = np.zeros_like(theta)
        for i, theta_i in enumerate(theta):
            self.mean_d_theta[i] = self.beta1 * self.mean_d_theta[i]
                    + (1 - self.beta1) * d_theta[i]
            self.u_variance_d_theta[i] = self.beta2 * self.u_variance_d_theta[i]
                    + (1 - self.beta2) * (d_theta[i] ** 2)

            # the bias correction step
            mean_d_theta_corr[i] = self.mean_d_theta[i]
                    / (1 - self.beta1 ** num_iter)
```

```
        u_variance_d_theta_corr[i] = self.u_variance_d_theta[i]
                / (1 - self.beta2 ** num_iter)

        # update weights and biases
        theta[i] = theta[i] - self.step_size * (mean_d_theta_corr[i]
                / (np.sqrt(u_variance_d_theta_corr[i])+ self.epsilon))
    return theta
```