CASE STUDY ON OPTIMIZER FUNCTIONS
OF
DEEP LEARNING

BACHELOR OF TECHNOLOGY

IN

INFORMATION TECHNOLOGY

SUBMITTED BY

19070124005 – ALI MAZHAR LUQMANI

19070124024 – HARSH VARDHAN GULERIA

19070124050 – YOGESHWAR PAWADE

19070124074 – SWANAND WAGH

Under the guidance of

Mr. EVAM KAUSHIK

SYMBIOSIS INSTITUTE OF TECHNOLOGY

(A CONSTITUENT OF SYMBIOSIS INTERNATIONAL UNIVERSITY)

Pune – 412115

2021-22

MINI-BATCH GRADIENT DESCENT

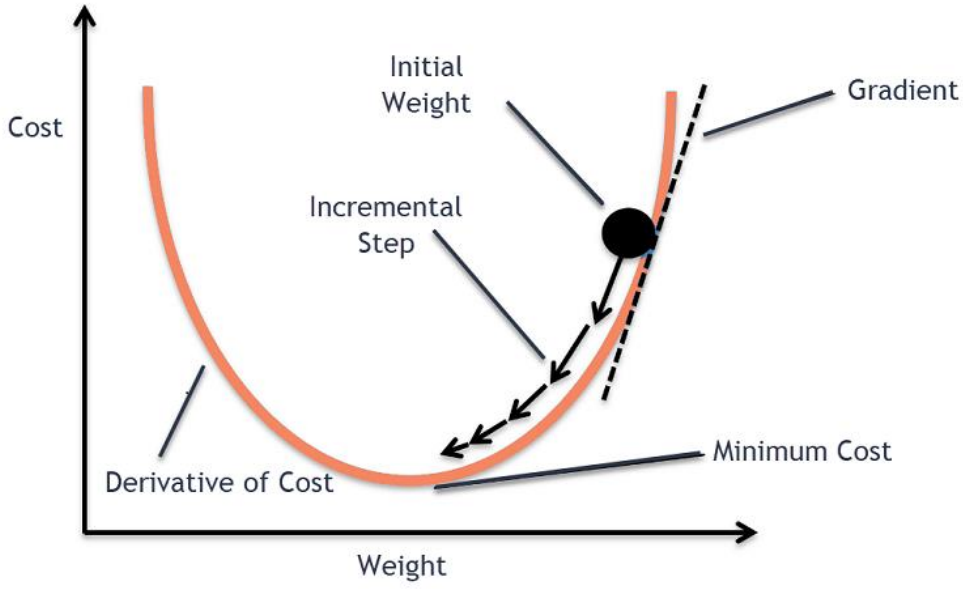
Introduction –

In machine learning, gradient descent is an optimization technique used for computing the model parameters for algorithms like linear regression, logistic regression, neural network, etc. In this technique, we repeatedly iterate through the training set and update the model parameters in accordance with the gradient of error with respect to the training set.

Neural networks are trained using gradient descent where the estimate of the error used to update the weights is calculated based on a subset of the training dataset.

Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function. To achieve this goal, it performs two steps iteratively.

1. Compute the slope (gradient) that is the first-order derivative of the function at the current point.
2. Move-in the opposite direction of the slope increase from the current point by the computed amount.



So, the idea is to pass the training set through the hidden layers of the neural network and then update the parameters of the layers by computing the gradients using the training samples from the training dataset.

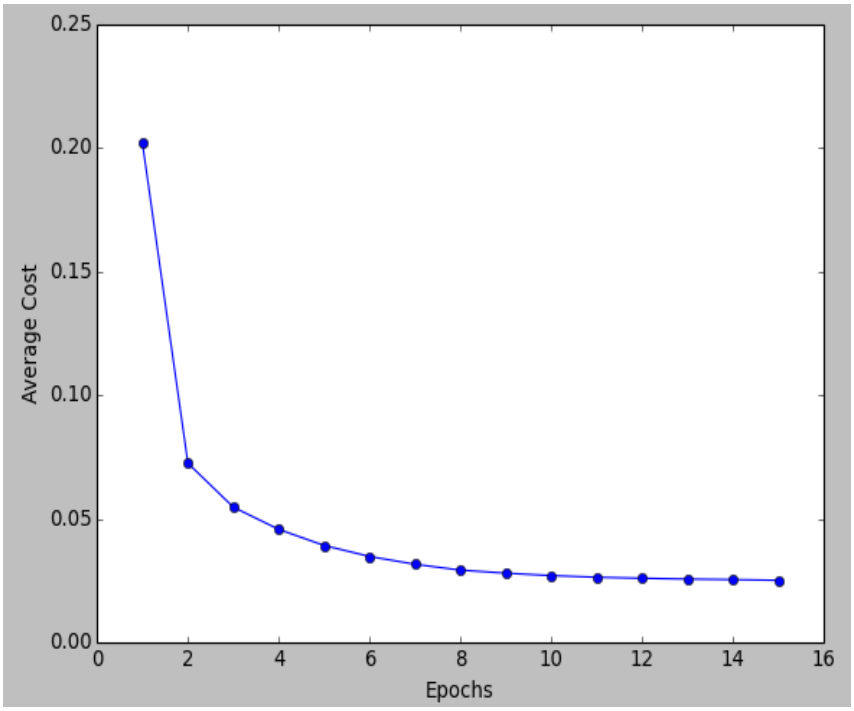
The number of examples from the training dataset used in the estimate of the error gradient is called the **batch size** and is an important hyperparameter that influences the dynamics of the learning algorithm.

Depending on the number of training examples considered in updating the model parameters, we have 3-types of gradient descents:

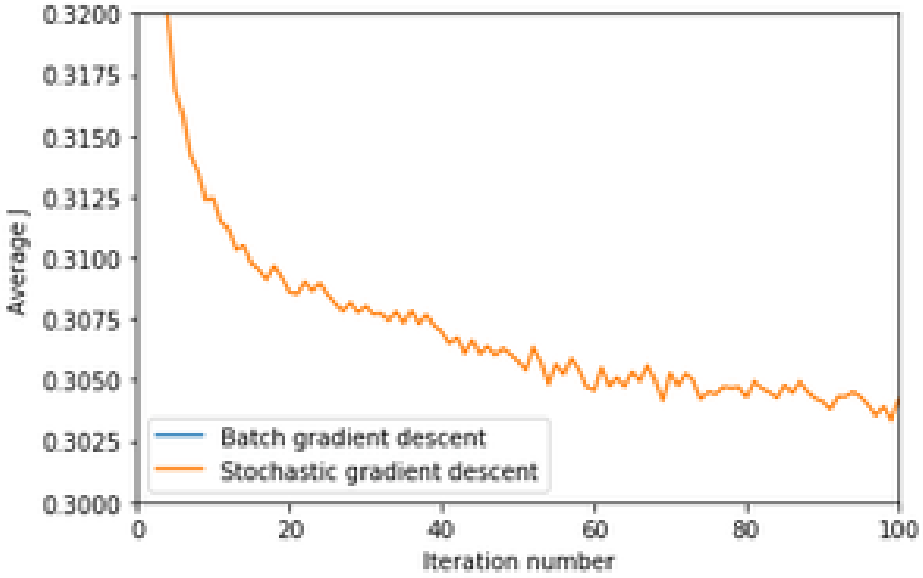
Batch Gradient Descent	Stochastic Gradient Descent	Mini-Batch Gradient Descent
Since entire training data is considered before taking a step in the direction of gradient, therefore it takes a lot of time for making a single update.	Since only a single training example is considered before taking a step in the direction of gradient, we are forced to loop over the training set and thus cannot exploit the speed associated with vectorizing the code.	Since a subset of training examples is considered, it can make quick updates in the model parameters and can also exploit the speed associated with vectorizing the code.
It makes smooth updates in the model parameters.	It makes very noisy updates in the parameters.	Depending upon the batch size, the updates can be made less noisy-greater the batch size less noisy is the update.

Thus, mini-batch gradient descent makes a compromise between the speedy convergence and the noise associated with gradient update which makes it a more flexible and robust algorithm.

Batch Gradient Descent can be used for smoother curves. SGD can be used when the dataset is large. Batch Gradient Descent converges directly to minima.



SGD converges faster for larger datasets. But, since in SGD we use only one example at a time, we cannot implement the vectorized implementation on it. This can slow down the computations.

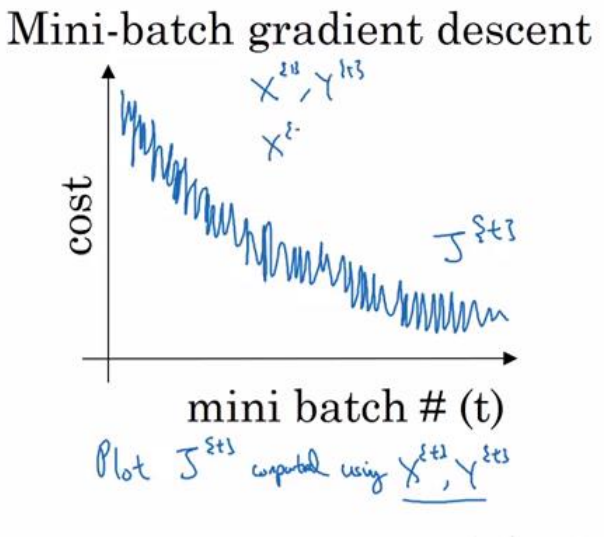


To tackle this problem, a mixture of Batch Gradient Descent and SGD is used.

Neither we use all the dataset all at once nor we use the single example at a time. We use a batch of a fixed number of training examples which is less than the actual dataset and call it a mini-batch. Doing this helps us achieve the advantages of both the former variants we saw. So, after creating the mini-batches of fixed size, we do the following steps in **one epoch:**

1. Pick a mini-batch.
2. Feed it to Neural Network.
3. Calculate the mean gradient of the mini-batch.
4. Use the mean gradient we calculated in step 3 to update the weights.
5. Repeat steps 1–4 for the mini-batches we created.

Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.



We have generated 8000 data examples, each having 2 attributes/features. These data examples are further divided into training set (X_train, y_train) and testing set (X_test, y_test) having 7200 and 800 examples respectively.

```
# linear regression using "mini-batch" gradient descent
# function to compute hypothesis / predictions
def hypothesis(X, theta):
    return np.dot(X, theta)

# function to compute gradient of error function w.r.t. theta
def gradient(X, y, theta):
```

```
    h = hypothesis(X, theta)
    grad = np.dot(X.transpose(), (h - y))
    return grad

# function to compute the error for current values of theta
def cost(X, y, theta):
    h = hypothesis(X, theta)
    J = np.dot((h - y).transpose(), (h - y))
    J /= 2
    return J[0]

# function to create a list containing mini-batches
def create_mini_batches(X, y, batch_size):
    mini_batches = []
    data = np.hstack((X, y))
    np.random.shuffle(data)
    n_minibatches = data.shape[0] // batch_size
    i = 0

    for i in range(n_minibatches + 1):
        mini_batch = data[i * batch_size:(i + 1)*batch_size, :]
        X_mini = mini_batch[:, :-1]
        Y_mini = mini_batch[:, -1].reshape((-1, 1))
        mini_batches.append((X_mini, Y_mini))
    if data.shape[0] % batch_size != 0:
        mini_batch = data[i * batch_size:data.shape[0]]
        X_mini = mini_batch[:, :-1]
        Y_mini = mini_batch[:, -1].reshape((-1, 1))
        mini_batches.append((X_mini, Y_mini))
    return mini_batches

# function to perform mini-batch gradient descent
def gradientDescent(X, y, learning_rate = 0.001, batch_size = 32):
    theta = np.zeros((X.shape[1], 1))
    error_list = []
    max_iters = 3
    for itr in range(max_iters):
        mini_batches = create_mini_batches(X, y, batch_size)
        for mini_batch in mini_batches:
            X_mini, y_mini = mini_batch
            theta = theta - learning_rate * gradient(X_mini, y_mini, theta)
            error_list.append(cost(X_mini, y_mini, theta))
```

```
    return theta, error_list
```

Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. It is the most common implementation of gradient descent used in the field of deep learning.

Advantages :

- The model update frequency is higher than batch gradient descent which allows for a more robust convergence, avoiding local minima.
- The batched updates provided a computationally more efficient process than stochastic gradient descent.
- The batching allows both the efficiency of not having all training data in memory and algorithm implementations.

Disadvantages :

- Mini-batch requires the configuration of an additional “mini-batch size” hyperparameter for the learning algorithm.
- Error information must be accumulated across mini-batches of training examples like batch gradient descent.

RMSProp OPTIMIZER

AdaGrad works well when the objective is convex and it can shrink the learning rate in a fixed decay procedure. The RMSProp update adjusts the AdaGrad method in a very simple way in an attempt to reduce its aggressive, monotonically decreasing learning rate.

In particular, it uses a moving average of squared gradients instead, and we can also manipulate the accumulation of an exponentially decaying average of the gradient. This is what RMSProp performed in many neural networks.

RMSProp combines the idea that keeping a moving average of the squared gradient for each weight, which only uses the sign of the gradient. This strategy effectively avoid the earlier stop of the learning procedure caused by accumulated second order momentum. Its update can be expressed as:

$$V_n = \rho \cdot V_{n-1} + (1-\rho) \cdot \left(\nabla(\theta_i)|_{J(\theta)} \right)^2$$

$$\theta := \theta - \eta \cdot m_n / (\sqrt{V_n} + \varepsilon)$$

Where ρ is decay parameter. When ρ = 0.5, the RMSProp becomes to AdaGrad. In terms of some complex non-convex function in deep learning, the learning trajectory of parameters will pass through sorts of areas to reach a local convex area, and find a local minimum in this area. AdaGrad used in such networks may reduce the learning step too fast before it reaches this local convex region, causing it to fail to converge.

But RMSProp overcame this drawback, and it can forget the previous information using the exponential decay so that the weight of the recent gradient information can be augmented. After the optimizer stepping into the local convex region, it will perform as AdaGrad does.

Advantages :

(1) Overcome the shortcoming of AdaGrad method, which avoid the aggressive decay of the learning rate.

(2) Good for training big and redundant datasets.

Disadvantages : It still depends on a global learning rate.

Analysis :

RMSProp has experienced a lot of developments it has collaborated with standard momentum but the performance had not been significantly improved. Also RMSProp has combined with Nesterov momentum method, but it only performed better if the root-mean-square of the recent gradients is used to divide the correction rather than the jump in the direction of accumulated corrections.

GRADIENT DESCENT WITH MOMENTUM

Gradient descent :

Gradient Descent is an optimization algorithm. It is technically referred to as a first-order optimization algorithm as it explicitly makes use of the first-order derivative of the target objective function. It refers to a minimization optimization algorithm that follows the negative of the gradient downhill of the target function to locate the minimum of the function.

The gradient descent algorithm requires a target function that is being optimized and the derivative function for the objective function. The target function f() returns a score for a given set of inputs, and the derivative function f'() gives the derivative of the target function for a given set of inputs.

The gradient descent algorithm requires a starting point (x) in the problem, such as a randomly selected point in the input space.
The derivative is then calculated and a step is taken in the input space that is expected to result in a downhill movement in the target function, assuming we are minimizing the target function.

A downhill movement is made by first calculating how far to move in the input space, calculated as the step size (called alpha or the learning rate) multiplied by the gradient. This is then subtracted from the current point, ensuring we move against the gradient, or down the target function.

x = x – step_size * f'(x)

The steeper the objective function at a given point, the larger the magnitude of the gradient and, in turn, the larger the step taken in the search space. The size of the step taken is scaled using a step size hyperparameter.
Step Size (alpha): Hyperparameter that controls how far to move in the search space against the gradient each iteration of the algorithm, also called the learning rate.
If the step size is too small, the movement in the search space will be small and the search will take a long time. If the step size is too large, the search may bounce around the search space and skip over the optima.

Momentum :

Momentum is an extension to the gradient descent optimization algorithm, often referred to as gradient descent with momentum.

It is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result.

A problem with the gradient descent algorithm is that the progression of the search can bounce around the search space based on the gradient. For example, the search may progress downhill towards the minima, but during this progression, it may move in another direction, even uphill, depending on the gradient of specific points (sets of parameters) encountered during the search.

This can slow down the progress of the search, especially for those optimization problems where the broader trend or shape of the search space is more useful than specific gradients along the way.

One approach to this problem is to add history to the parameter update equation based on the gradient encountered in the previous updates.

Momentum involves adding an additional hyperparameter that controls the amount of history (momentum) to include in the update equation, i.e. the step to a new point in the search space. The value for the hyperparameter is defined in the range 0.0 to 1.0 and often has a value close to 1.0, such as 0.8, 0.9, or 0.99. A momentum of 0.0 is the same as gradient descent without momentum.

The change in the parameters is calculated as the gradient for the point scaled by the step size.

change_x = step_size * f'(x)

The new position is calculated by simply subtracting the change from the current point

x = x – change_x

Momentum involves maintaining the change in the position and using it in the subsequent calculation of the change in position.
If we think of updates over time, then the update at the current iteration or time (t) will add the change used at the previous time (t-1) weighted by the momentum hyperparameter, as follows:

change_x(t) = step_size * f'(x(t-1)) + momentum * change_x(t-1)

The update to the position is then performed as before.

x(t) = x(t-1) – change_x(t)

The change in the position accumulates magnitude and direction of changes over the iterations of the search, proportional to the size of the momentum hyperparameter.
For example, a large momentum (e.g. 0.9) will mean that the update is strongly influenced by the previous update, whereas a modest momentum (0.2) will mean very little influence.

Advantages:

1. Momentum method suppressed the problem that the gradient descent methods converge not stable when using batches, and accelerated the learning procedures, furthermore, it will have faster convergence.

2. It has the ability to get rid of the local optimum, because it will keep updating a few steps after the current gradient is zero and the momentum term is still non-zero.

Disadvantages :

1. It has one more hyper parameter, (momentum factor and learning rate), and their initialization will heavily influence the performance of optimization algorithm.

2. Sometimes, with the accumulated momentum and increasing update speed, this optimizer have the probability to jump out of the optimum region.

ADAM OPTIMIZER

Theory :

Some explorations has been made on the RMSProp combined with the momentum, but the effects are not that remarkable. Adam is like RMSProp with momentum, but involved with the bias correction terms for the first and second momentum.

Adam uses the first and second order moment estimations of the gradient to adaptively adjust the learning rate of each parameter. After the bias correction, the learning rate can be dynamically constrained into a certain slope, hence the parameters are updated stably.

The parameter update rule can be expressed as:

$$\hat{m}_n = \frac{m_n}{1-\beta_1^n}$$

$$\hat{V}_n = \frac{V_n}{1-\beta_2^n}$$

$$\theta := \theta - \frac{\eta}{\sqrt{\hat{V}_n} + \varepsilon}\hat{m}_n$$

Where β1 and β2 are exponential decay rates for moment estimates, and it suggested defaults by β1 = 0.9, β2 = 0.999. Again note that it used the first and second order moment estimation of the loss function to constrain the global learning rate.

How Adam Work's :

1. It calculates an exponentially weighted average of past gradients, and stores it in variables v (before bias correction) and $v^{corrected}$ (with bias correction).
2. It calculates an exponentially weighted average of the squares of the past gradients, and stores it in variables s (before bias correction) and $s^{corrected}$ (with bias correction).
3. It updates parameters in a direction based on combining information from "1" and "2".

The update rule is, for $l = 1, ..., L$:

$$\begin{cases} v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1) \frac{\partial \mathcal{J}}{\partial W^{[l]}} \\ v_{dW^{[l]}}^{corrected} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t} \\ s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2)(\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2 \\ s_{dW^{[l]}}^{corrected} = \frac{s_{dW^{[l]}}}{1 - (\beta_2)^t} \\ W^{[l]} = W^{[l]} - \alpha \frac{v_{dW^{[l]}}^{corrected}}{\sqrt{s_{dW^{[l]}}^{corrected}} + \varepsilon} \end{cases}$$

where:

- t counts the number of steps taken of Adam
- L is the number of layers
- β_1 and β_2 are hyperparameters that control the two exponentially weighted averages.

- α is the learning rate
- ε is a very small number to avoid dividing by zero.

Advantages :

1. Combines the advantages of processing sparse gradient (AdaGrad), and non-stable objects (RMSProp).
2. Adam converges faster since it is not required to store all scaled gradient, hence it is suitable for large-scale datasets and high-dimensional space (many non-convex optimization).
3. Setting different adaptive learning rate for different parameters.

Analysis :

For most of the optimization algorithms, they used different strategy, take different path, but reached a same goal. If no specific requirement for a precise optimization, Adam method can be a pretty good choice. But Adam cannot be always satisfied with all the situations. We can also make a better use of an optimization algorithm by fine-tuning their control hyperparamters as long as we have a better understand about the data. For my best knowledge, even though Adam works well in lots of models.

Adam may not converge:

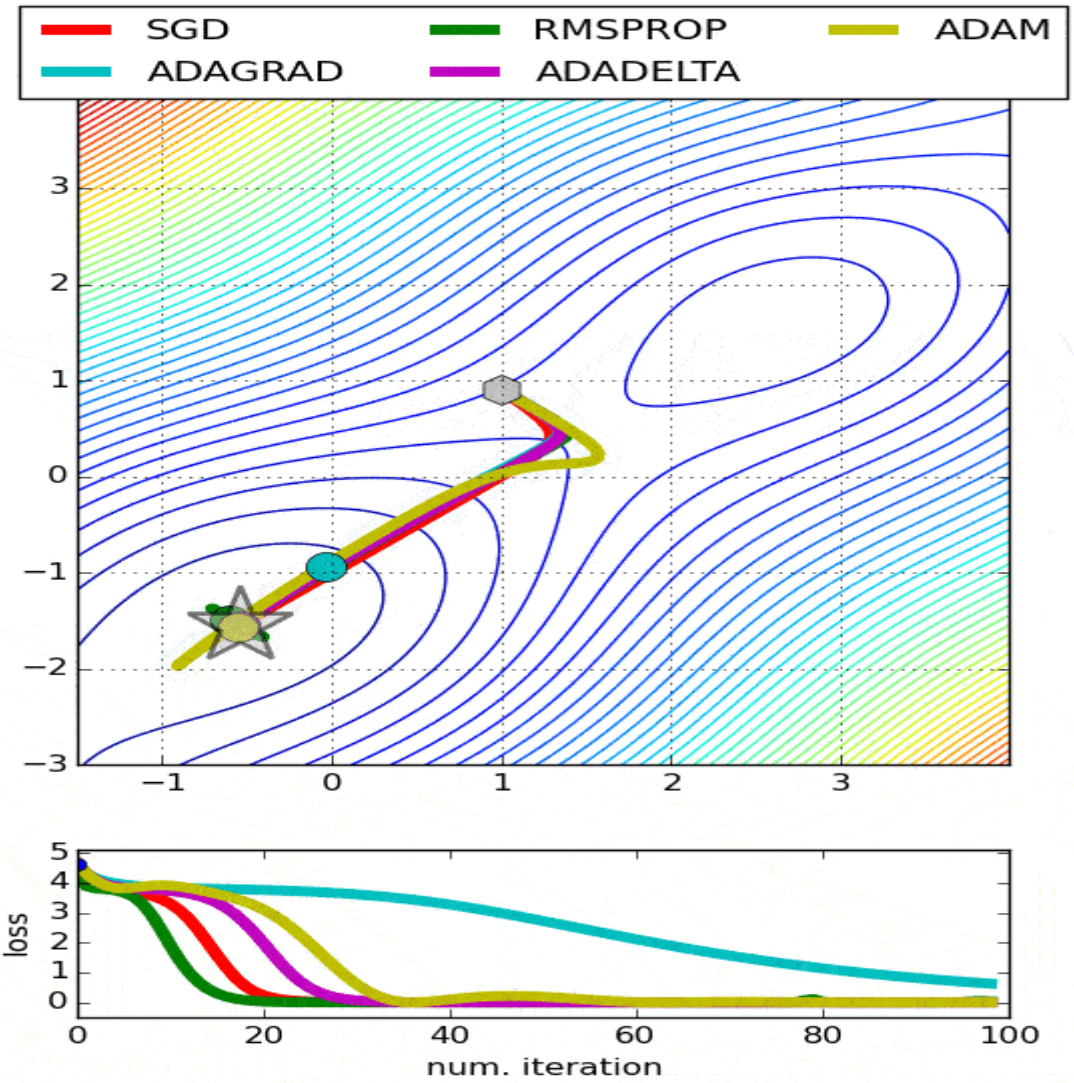
A conference paper in ICLR discussed the convergence of Adam, and they pointed out Adam cannot converge by listing some counter-examples. Stochastic gradient descent doesn't Report Optimization Methods for Deep Learning X. Z.Wen 17 use second order momentum, so the learning rate can be fixed. But stochastic gradient descent always comes with learning rate decay strategies, hence the learning rate can be decreasing. However, the Adam accumulate the momentum with some certain time slots. If the data are varying within a period of time, the moment can be vibrating instead of monotonically decreasing. This situation can make learning rate unstable so that the algorithm can be not converged. This paper gives the solutions for this case, and it can manipulate the second order momentum to avoid learning rate oscillations [5]: □ □ 2 V V g V n n t n □ □ □ □ □ max (1) , □ □ 2 1 2 1 □ □ (28) After the

correction, it can be guaranteed that the learning rate is monotonically decreasing.

Adam can miss the global optimum:

Deep learning involves with a large number of parameters, which is in a pretty highdimensional spaces. The non-convex objective function tends to rise and fall, with numerous uplands and depressions. There are lots of peaks, which can easily be crossed by the momentum terms. For some plateaus, algorithm cannot found after many steps and they just stop training. Different algorithms gives a large variety of optimum solutions. The adaptive learning rate method can over-fit the former shown-up features, but for the features observed later after, it can be too hard to completely eliminate the over-fit. Researchers tested these algorithms on CIFAR-10 dataset, they found that Adam converges very fast, but the optimum derived from Adam can be worse than that from stochastic gradient descent. This is because the learning rate can be really small at end which slowed the convergence speed. This problem can be solved by control the lower bound of the learning rate [6, 7]. After review these papers, they all used some counter-examples to demonstrate the effectiveness of Adam, which really remind me that the algorithms have all sorts of advantages, but it is always better to get fully understand the data formation. As seen from the optimization history, all optimization algorithms are based on an initial assumption of the data. The performances can be determined if these assumptions are satisfied or not.

CONCLUSION



Adam is the best optimizers. If one wants to train the neural network in less time and more efficiently than Adam is the optimizer.

For sparse data use the optimizers with dynamic learning rate.

If, want to use gradient descent algorithm than min-batch gradient descent is the best option.

To view the source code Click Here.

REFERENCES

Towards data science - https://towardsdatascience.com/

Machine Learning – https://machinelearningmastery.com/

Geeks for Geeks - https://www.geeksforgeeks.org/

Optimization methods in Deep Learning - Xunzhe Wen