Suryanarayan.B

CB.EN.U4CSE19056

Neural Network and Deep Learning

Reading Assignment

**What are Optimizers?**

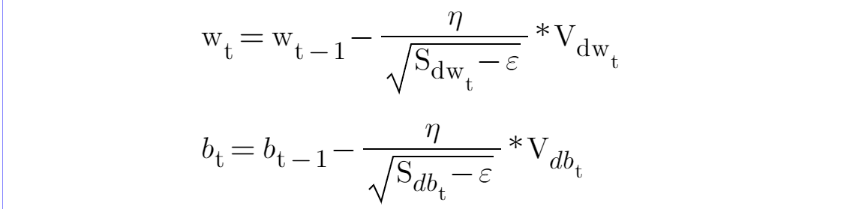
Optimizers are the expanded class, which includes the method to train your machine/deep learning model. Right optimizers are necessary for your model as they improve training speed and performance

**ADAM :**

Adam optimization is an extension to Stochastic gradient decent and can be used in place of classical stochastic gradient descent to update network weights more efficiently.Adam uses Momentum and Root Mean square Propogation or RMSProp to converge faster. Momentum algorithm, accelerates stochastic gradient descent in the relevant direction, as well as dampening oscillations.Adaptive learning rates can be thought of as adjustments to the learning rate in the training phase by reducing the learning rate to a pre-defined schedule.

**The Equation of Adam:**

Using the equation for Exponential Weighted Averages for past gradients and Exponential Weighted Averages for past squared gradients we get the following final equation for ADAM

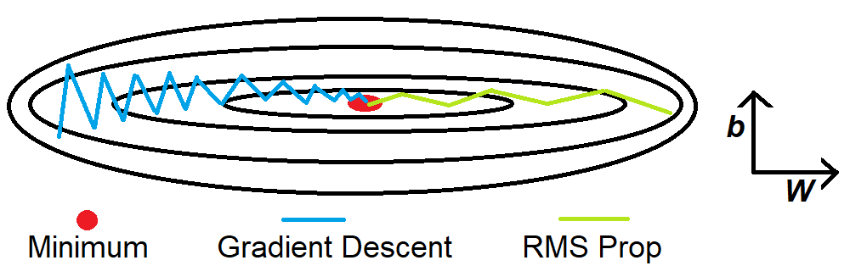


Advantages of Adam:

* Straightforward to implement
* Computationally efficient
* Little memory requirements
* Appropriate for problems with very noisy/or sparse gradients
* Hyper-parameters have intuitive interpretation and typically require a little tuning

**Root Mean Square Propagation RMS Prop**

The Root Mean Square Propagation **RMS Prop** is similar to **Momentum**, it is a technique to dampen out the motion in the y-axis and speed up gradient descent. For better understanding, let us denote the Y-axis as the bias **b** and the X-axis as the weight **W**. It is called Root Mean Square because we square the derivatioves of both w and b parameters.



The intuition is that when we divide a large number by another number, the result becomes small. In our case, the first large number is db and the second large number that we use is the weighted average of db². We introduce two new variables Sdb and SdW, to keep track of the weighted average of db² and dW². The division of db and Sdb results in a smaller value which dampens out the movement in the y-axis. The Ⲉ is introduced to avoid the division by 0 error.

SdW=βSdW+(1−β)dW2

Sdb=βSdb+(1−β)db2

W=W−α⋅dW√SdW+ϵ

b=b−α⋅db√Sdb+ϵ

The idea is to slow down the learning on the y-axis direction and speed up the learning on the x-axis direction. On each interatioin **t** the derivatives **dw** and **db** are computed on the current **mini-batch**. It is also perfomed the **exponentially weighted average** called **SdW** and **Sdb**. Now, the exponentially weighted average parameter **SdW** is relatively small and so we are deviding by a small number to obtain the weight. The **alpha** parameter is the learning rate and the **Ⲉ** is introduced to avoid the division by 0 error. On the contrary, **Sdb** is relatively large and it helps to slow down the updates vertical dimention b. In fact looking at the figure above, the derivative in the orizontal dimenstion is aways small and the derivative on the vertical dimention is large. The net effect is to speed up the veritical learing and at the same time slow down the vertical learning. The result is the green line of th figure above.

**Adaptive Gradient Algorithm (AdaGrad)**

Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the updates.

**Arguments**

* **learning\_rate**: Initial value for the learning rate: either a floating point value, or a [tf.keras.optimizers.schedules.LearningRateSchedule](https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/LearningRateSchedule) instance. Defaults to 0.001. Note that Adagrad tends to benefit from higher initial learning rate values compared to other optimizers. To match the exact form in the original paper, use 1.0.
* **initial\_accumulator\_value**: Floating point value. Starting value for the accumulators (per-parameter momentum values). Must be non-negative.
* **epsilon**: Small floating point value used to maintain numerical stability.
* **name**: Optional name prefix for the operations created when applying gradients. Defaults to "Adagrad".
* **\*\*kwargs**: Keyword arguments. Allowed to be one of "clipnorm" or "clipvalue". "clipnorm" (float) clips gradients by norm and represents the maximum L2 norm of each weight variable; "clipvalue" (float) clips gradient by value and represents the maximum absolute value of each weight variable.

AdaGrad is designed to specifically explore the idea of automatically tailoring the step size for each dimension in the search space.

This is achieved by first calculating a step size for a given dimension, then using the calculated step size to make a movement in that dimension using the partial derivative. This process is then repeated for each dimension in the search space.

*Adagrad dulls the influence of parameters with consistently high gradients, thereby increasing the influence of parameters with infrequent updates.*

AdaGrad is suited to objective functions where the curvature of the search space is different in different dimensions, allowing a more effective optimization given the customization of the step size in each dimension.

The algorithm requires that you set an initial step size for all input variables as per normal, such as 0.1 or 0.001, or similar. Although, the benefit of the algorithm is that it is not as sensitive to the initial learning rate as the gradient descent algorithm.

*Adagrad is far less sensitive to the learning rate parameter alpha. The learning rate parameter is typically set to a default value of 0.01.*

An internal variable is then maintained for each input variable that is the sum of the squared partial derivatives for the input variable observed during the search.

This sum of the squared partial derivatives is then used to calculate the step size for the variable by dividing the initial step size value (e.g. hyperparameter value specified at the start of the run) divided by the square root of the sum of the squared partial derivatives.

* cust\_step\_size = step\_size / sqrt(s)

It is possible for the square root of the sum of squared partial derivatives to result in a value of 0.0, resulting in a divide by zero error. Therefore, a tiny value can be added to the denominator to avoid this possibility, such as 1e-8.

* cust\_step\_size = step\_size / (1e-8 + sqrt(s))

Where *cust\_step\_size* is the calculated step size for an input variable for a given point during the search, *step\_size* is the initial step size, *sqrt()* is the square root operation, and *s* is the sum of the squared partial derivatives for the input variable seen during the search so far.

The custom step size is then used to calculate the value for the variable in the next point or solution in the search.

* x(t+1) = x(t) – cust\_step\_size \* f'(x(t))

This process is then repeated for each input variable until a new point in the search space is created and can be evaluated.

Importantly, the partial derivative for the current solution (iteration of the search) is included in the sum of the square root of partial derivatives.

We could maintain an array of partial derivatives or squared partial derivatives for each input variable, but this is not necessary. Instead, we simply maintain the sum of the squared partial derivatives and add new values to this sum along the way.