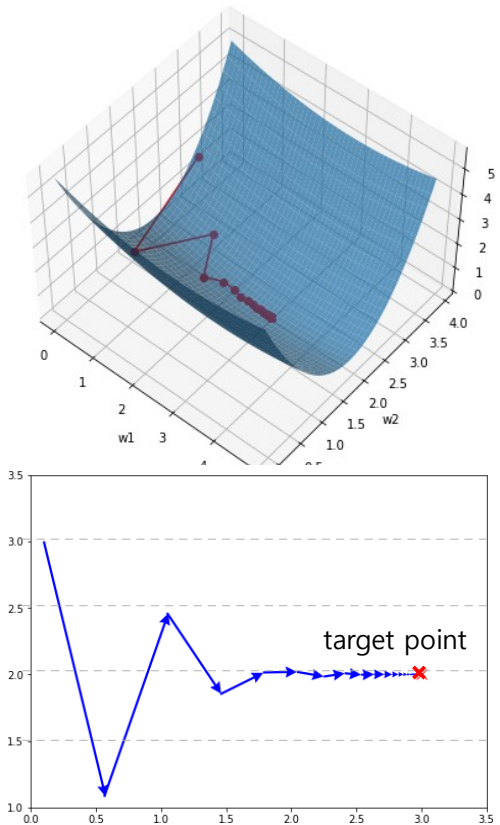


2. Optimizers

Part 1: Gradient Descent and Momentum

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

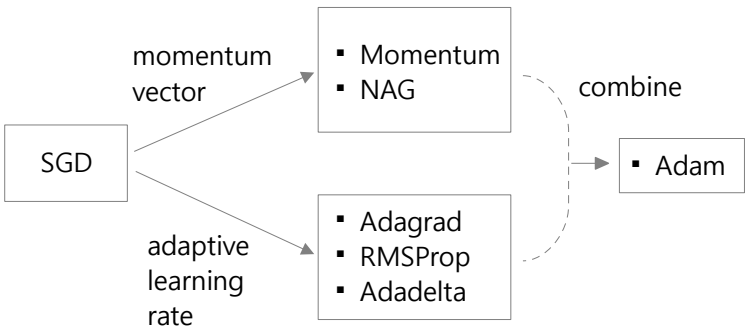
www.youtube.com/@meanxai

MX-AI

[MXDL-2-01]
1. Gradient Descent optimizer
2. Momentum optimizer

[MXDL-2-02]
3. Nesterov Accelerated Gradient (NAG)
4. Adaptive Gradient (Adagrad)
5. Root Mean Square Propagation (RMSprop)

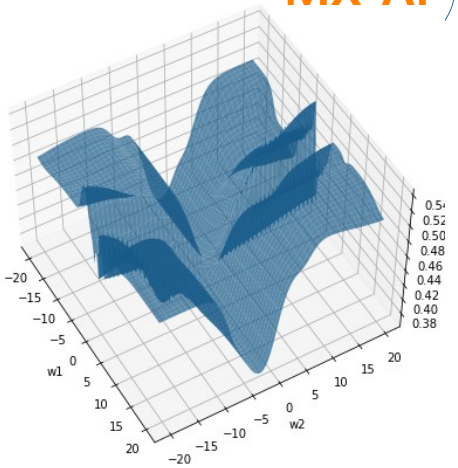
[MXDL-2-03]
6. Adaptive Delta (Adadelta)
7. Adaptive Moment Estimation (Adam)
8. Compare the paths to reach the target point.



MX-AI

■ Gradient descent optimizer

- There are multiple paths to reach the goal, which is a minimum point on the loss surface in a high-dimensional space. We need algorithms that find the optimal path to reach the goal as quickly and reliably as possible.
- These algorithms are called optimizers, and many different optimizers have been developed.
- Gradient descent is the most basic optimizer and has the advantage of being easy to understand and implement. However, this has some disadvantages, such as being easily trapped in local minima or having an oscillating path to the target point.



▪ Gradient Descent

$$w_{i,t+1} = w_{i,t} - \alpha \nabla_{w_i} L(w_1, w_2)$$

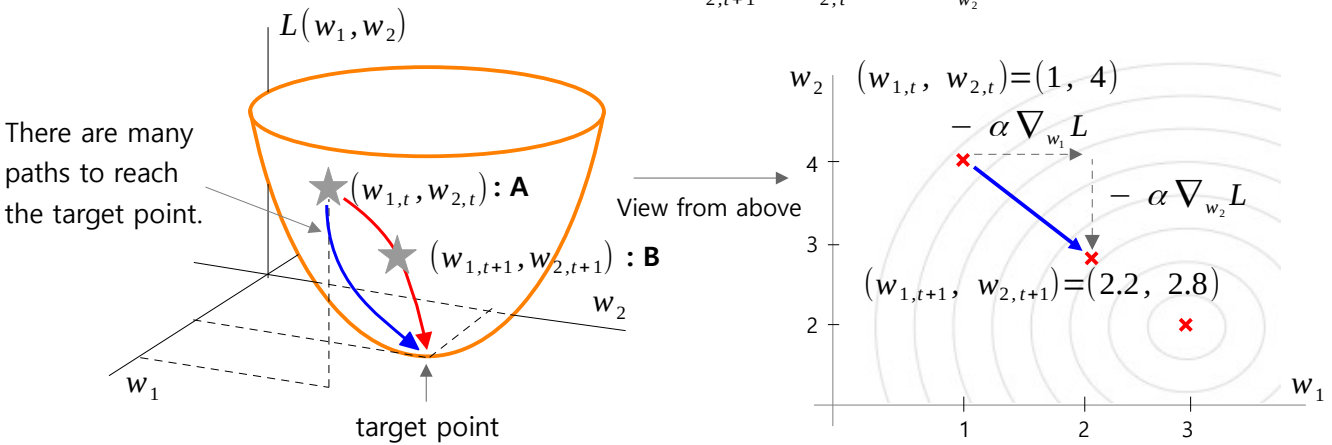
$$\nabla L(w_1, w_2) = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}]$$

ex: $L(w_1, w_2) = (w_1 - 3)^2 + (w_2 - 2)^2 \quad (w_{1,t}, w_{2,t}) = (1, 4)$

$$\nabla L(w_1, w_2) = [2w_1 - 6, \ 2w_2 - 4] \quad \nabla_t L(w_1, w_2) = (-4, 4)$$

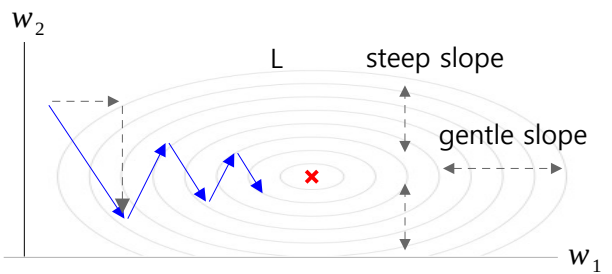
$$w_{1,t+1} = w_{1,t} - \alpha \nabla_{w_1} L = 1 - 0.3 \cdot (-4) = 2.2$$

$$w_{2,t+1} = w_{2,t} - \alpha \nabla_{w_2} L = 4 - 0.3 \cdot 4 = 2.8$$



▪ Zigzag oscillation

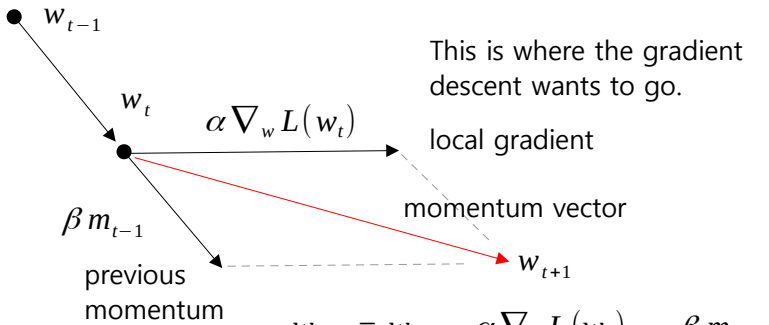
If the surface of the L has a gentle horizontal slope and a steep vertical slope, the gradient descent moves more toward the steep side and takes a zigzag path to find the target point.



MX-AI

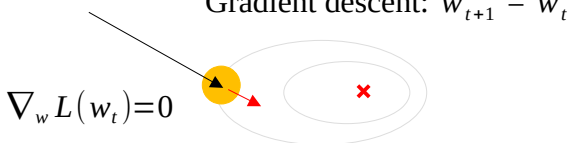
Momentum optimizer

- The momentum optimizer adds momentum to gradient descent.
- It adds the current gradient to the previous momentum. This is the momentum vector. This vector remembers past gradients.
- The parameter w is updated in the direction where the local gradient and previous momentum are combined.
- This will accelerate the gradient descent towards the target point and dampen the oscillations.
- Beta is a hyper-parameter and is specified as a value between 0 and 1. The inertia of the distant past gradually weakens, and the inertia of the recent past takes on greater weight.
- Due to inertia, the momentum optimizer may pass through a shallow local minimum or may take a different route.
- However, near the target point, we need to approach the target point slowly, but we can pass the target point due to acceleration. Of course, in the next iteration we will return towards the target point.

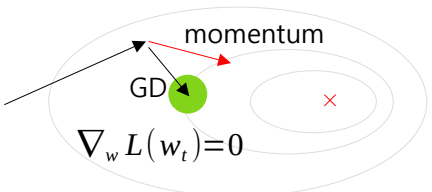


$$w_{t+1} = w_t - \alpha \nabla_w L(w_t) - \beta m_{t-1}$$

Gradient descent: $w_{t+1} = w_t$



Momentum: $w_{t+1} = w_t - \beta m_{t-1}$



Gradient descent cannot update w at the point where ∇L=0, but the momentum optimizer adds the previous momentum to w, so it can update w to move beyond this point.

If there is a shallow local minimum nearby, GD will move towards it, but the momentum optimizer may bypass it.

$$m_t = \beta m_{t-1} + \alpha \nabla_w L(w_t), \quad (m_0=0)$$

$$w_{t+1} = w_t - m_t$$

$$m_t = \beta(\beta m_{t-2} + \alpha \nabla_w L(w_{t-1})) + \alpha \nabla_w L(w_t)$$

$$m_t = \alpha \nabla_w L(w_t) + \beta \alpha \nabla_w L(w_{t-1}) + \beta^2 m_{t-2} + \ldots$$

← All momentum from the distant past is included, but the weight decreases rapidly as you go further into the past. This reflects a lot of recent momentum.

MX-AI

■ Zig-zag oscillation

```
# [MXDL-2-01] 1.oscillating.py
# We visually observe the zigzag oscillation phenomenon
# of gradient descent and verify that the momentum optimizer
# alleviates this phenomenon.
import numpy as np
import tensorflow as tf
from tensorflow.keras import optimizers
import matplotlib.pyplot as plt

# initial point and the target point
w0 = np.array([0.1, 3], dtype='float32') # initial point
wt = np.array([3, 2], dtype='float32')   # target point
w = tf.Variable(w0)                      # (w1, w2)

opt = optimizers.SGD(learning_rate = 0.8, momentum=0.0)
# opt = optimizers.SGD(learning_rate = 0.8, momentum=0.2)

def loss(w):
    return tf.reduce_sum(tf.square(w - wt) * [0.1, 1.2])

path = [w.numpy()]
for i in range(80):
    # perform automatic differentiation
    with tf.GradientTape() as tape:
        # the gradient of w1-axis is small,
        # and the gradient of w2-axis is large.
        dw = tape.gradient(loss(w), [w])

    # update w by gradient descent
    opt.apply_gradients(zip(dw, [w]))
    path.append(w.numpy())
```

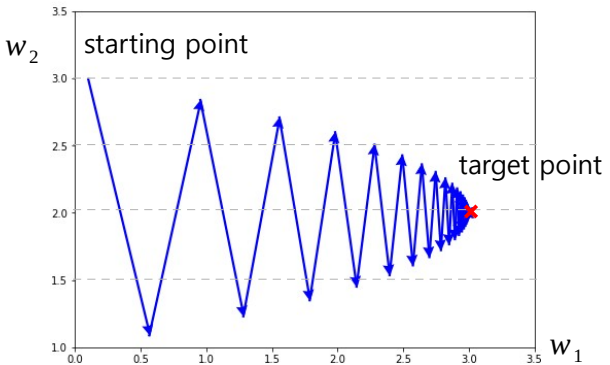
$$L(w_1, w_2) = 0.1(w_1-3)^2 + 1.2(w_2-2)^2$$

```
# visually check the path to the optimal point.
path = np.array(path)
x, y = path[:, 0], path[:, 1]
plt.figure(figsize=(8,6))
plt.quiver(x[:-1], y[:-1], x[1:]-x[:-1], y[1:]-y[:-1],
           color='blue', scale_units='xy',
           scale=1, width=0.005, headwidth=5)
plt.plot(wt[0], wt[1], marker='x', markersize=10, color='red')
plt.xlim(0, 3.5)
plt.ylim(1.0, 3.5)
plt.show()

# Draw the loss surface and the path to the optimal point.
w1, w2 = np.meshgrid(np.arange(0, 5, 0.1), np.arange(0, 4, 0.1))
zs = np.array([loss([a,b]).numpy() for [a, b] in zip(np.ravel(w1),
                np.ravel(w2))])
z = zs.reshape(w1.shape)

fig = plt.figure(figsize=(7, 7))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(w1, w2, z, alpha=0.7)

# Draw the path to the optimal point.
L = np.array([loss([a, b]).numpy() for [a, b] in zip(x, y)])
ax.plot(x, y, L, marker='o', color="r")
ax.set_xlabel('w1')
ax.set_ylabel('w2')
ax.set_zlabel('loss')
ax.azim = -50
ax.elev = 50
plt.show()
```

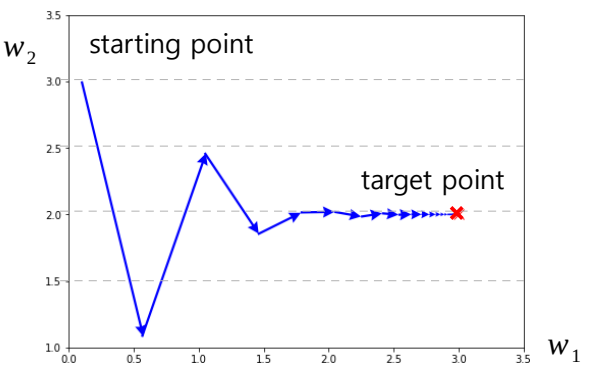
MX-AI

- Zig-zag oscillation

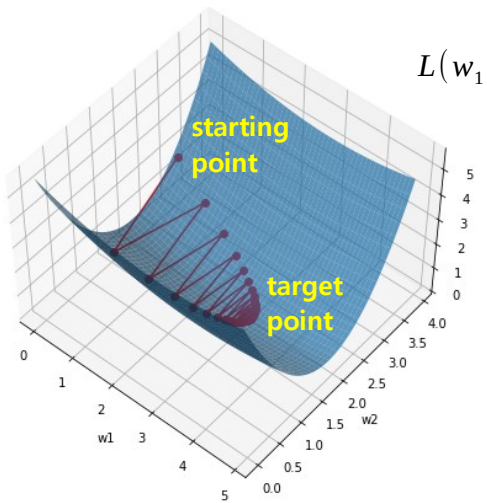
Gradient Descent optimizer

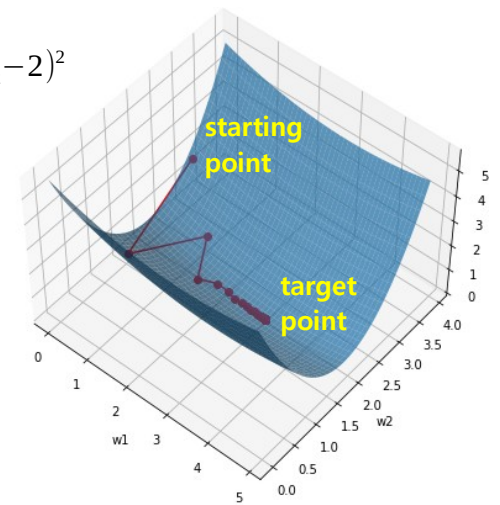


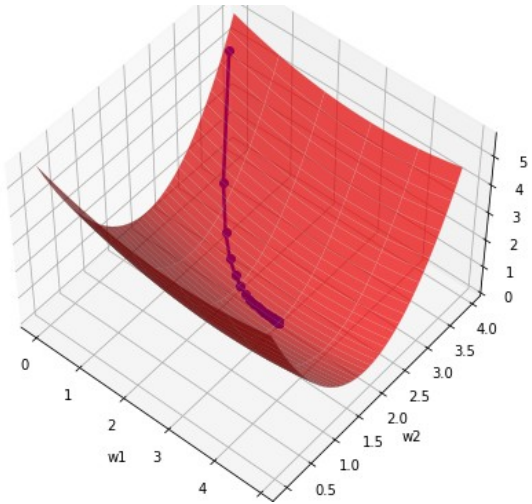
Momentum optimizer



$$L(w_1, w_2) = 0.1(w_1 - 3)^2 + 1.2(w_2 - 2)^2$$







2. Optimizers

Part 2: NAG, Adagrad, RMSprop

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

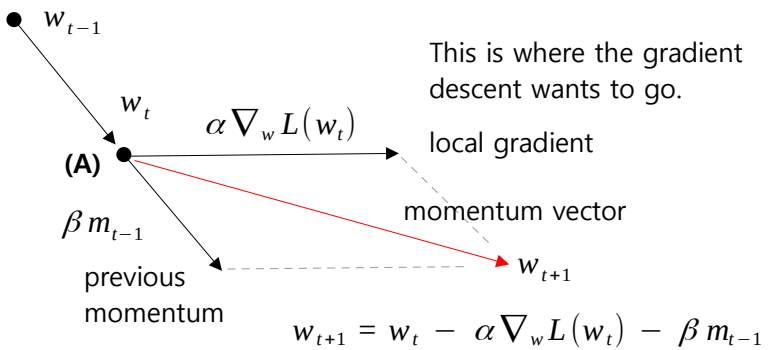
www.youtube.com/@meanxai

MX-AI

■ Nesterov Accelerated Gradient (NAG)

- The momentum optimizer has a disadvantage near the target point that it may pass the target point due to inertia. NAG is an improvement on this.
- When determining w_{t+1}, the momentum optimizer calculates the local gradient at the current point (A), while the NAG optimizer calculates the local gradient at the point where momentum was applied (B).

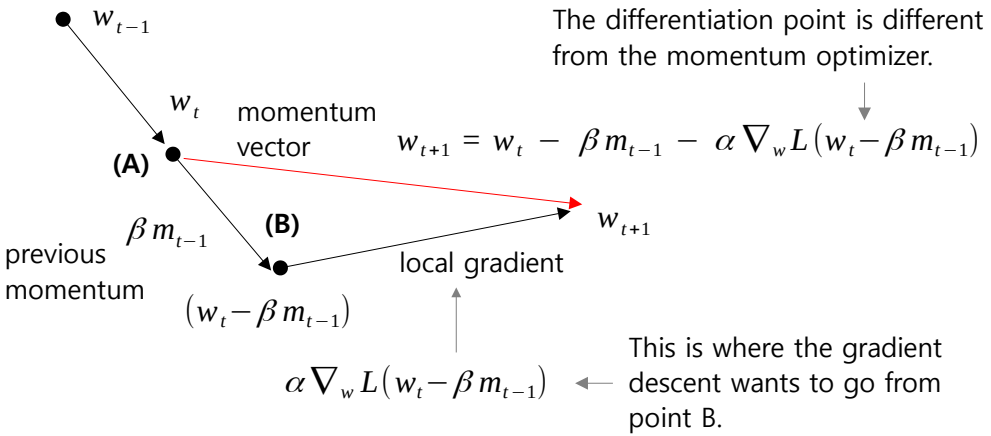
▪ Momentum optimizer



$$m_t = \beta m_{t-1} + \alpha \nabla_w L(w_t), \quad (m_0 = 0)$$

$$w_{t+1} = w_t - m_t$$

▪ NAG optimizer



$$m_t = \beta m_{t-1} + \alpha \nabla_w L(w_t - \beta m_{t-1}), \quad (m_0 = 0)$$

$$w_{t+1} = w_t - m_t$$

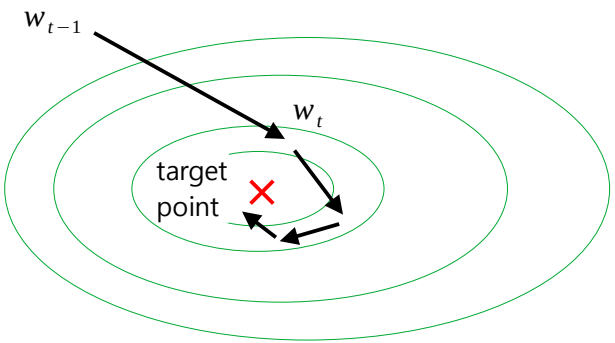
MX-AI

■ Nesterov Accelerated Gradient (NAG)

- NAG improves the disadvantage of the momentum optimizer, which may pass the target near the target point.
- When w is far away from the target point, it can quickly approach the target point due to the momentum effect. And when w approaches the target point, it can stably reach the target point without deviating from the target point due to the NAG effect.

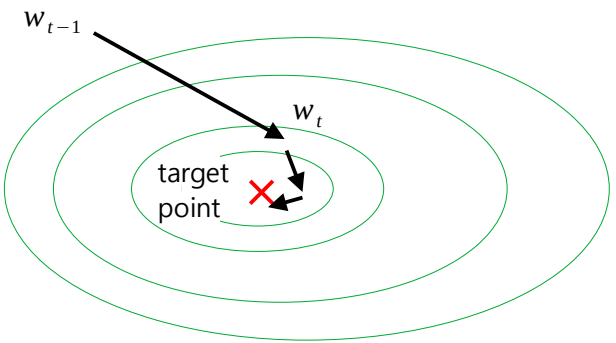
- Momentum optimizer

Example of a path of the momentum optimizer near the target point.



- NAG optimizer

Example of a path of the NAG optimizer near the target point.



MX-AI

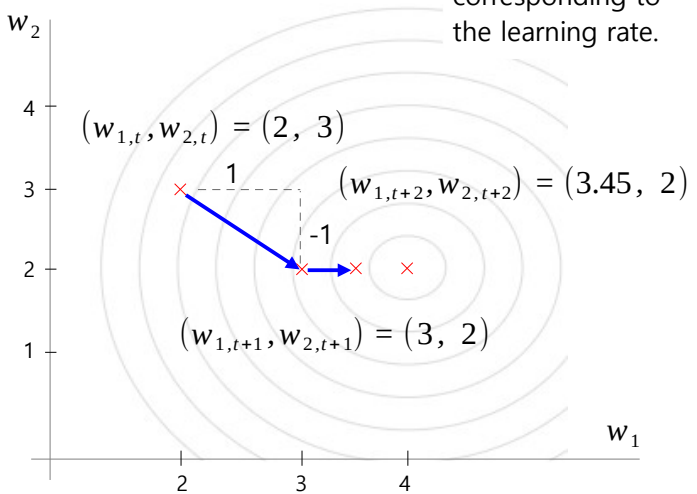
■ Adaptive Gradient (Adagrad)

- Gradient descent and momentum optimizers use a constant learning rate, and apply the same learning rate for all w. It is also a good idea to gradually reduce the learning rate while iterating. At the beginning of the iteration, w is likely to be far away from the target point, so w needs to be updated a lot by applying a large learning rate. And as w approaches the target point, w needs to be updated little by little by applying a smaller learning rate.
- Adagrad not only does this, but also applies a different learning rate to each w depending on the magnitude of its gradient.
- ϵ is to prevent the denominator from being zero. [G_0=0, ϵ=small value (ex: 10^{-6})]
- As iterations progress, Gt increases exponentially, which has the disadvantage of decreasing the learning rate too quickly.

$$G_t = G_{t-1} + [\nabla L_t(w)]^2, \quad (G_0=0)$$

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{G_t+\epsilon}} \nabla L_t(w)$$

corresponding to the learning rate.



$$L(w) = (w_1-4)^2 + (w_2-2)^2 \qquad \nabla L(w) = [2w_1-8,\ 2w_2-4]$$

$$(w_{1,t}, w_{2,t}) = (2,\ 3) \qquad \nabla L_t(w) = [-4,\ 2]$$

$$G_t = 0 + [\nabla L_t(w)]^2 = [16,\ 4]$$

$$\begin{cases} w_{1,t+1} = w_{1,t} - \frac{\alpha}{\sqrt{16}} \nabla_{w_1} L_t = 2 - \frac{1}{4}\cdot(-4) = 3, & (\alpha=1,\ \epsilon=0) \\ w_{2,t+1} = w_{2,t} - \frac{\alpha}{\sqrt{4}} \nabla_{w_1} L_t = 3 - \frac{1}{2}\cdot 2 = 2 \end{cases}$$

α can be set high because the learning rate drops significantly.

$$(w_{1,t+1}, w_{2,t+1}) = (3,\ 2) \qquad \nabla L_{t+1}(w) = (-2,\ 0)$$

$$G_{t+1} = [16,\ 4] + [\nabla L_{t+1}(w)]^2 = [20,\ 4]$$

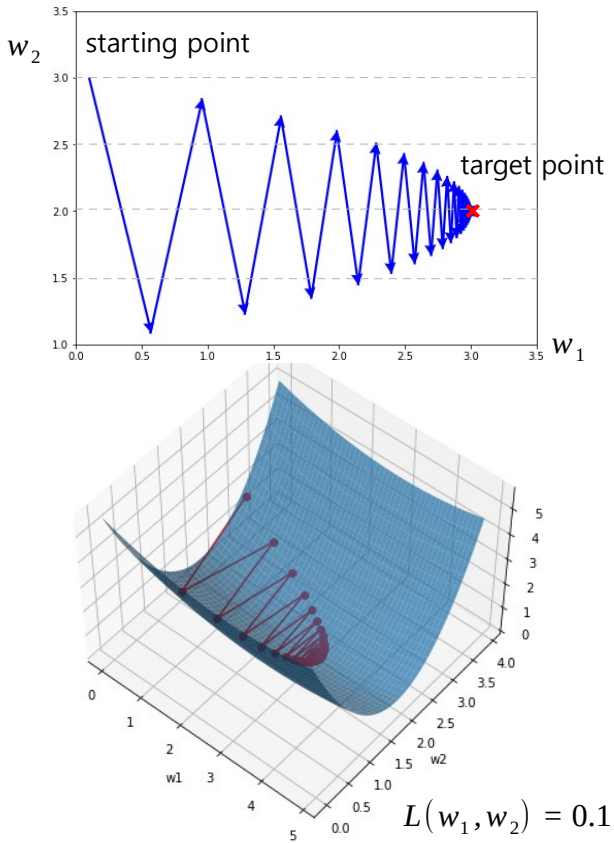
$$\begin{cases} w_{1,t+2} = w_{1,t+1} - \frac{\alpha}{\sqrt{20}} \nabla_{w_1} L_{t+1} = 3 - \frac{1}{\sqrt{20}}\cdot(-2) = 3.45 \\ w_{2,t+2} = w_{2,t+1} - \frac{\alpha}{\sqrt{4}} \nabla_{w_2} L_{t+1} = 2 - \frac{1}{\sqrt{4}}\cdot 0 = 2 \end{cases}$$

MX-AI

■ Adaptive Gradient (Adagrad)

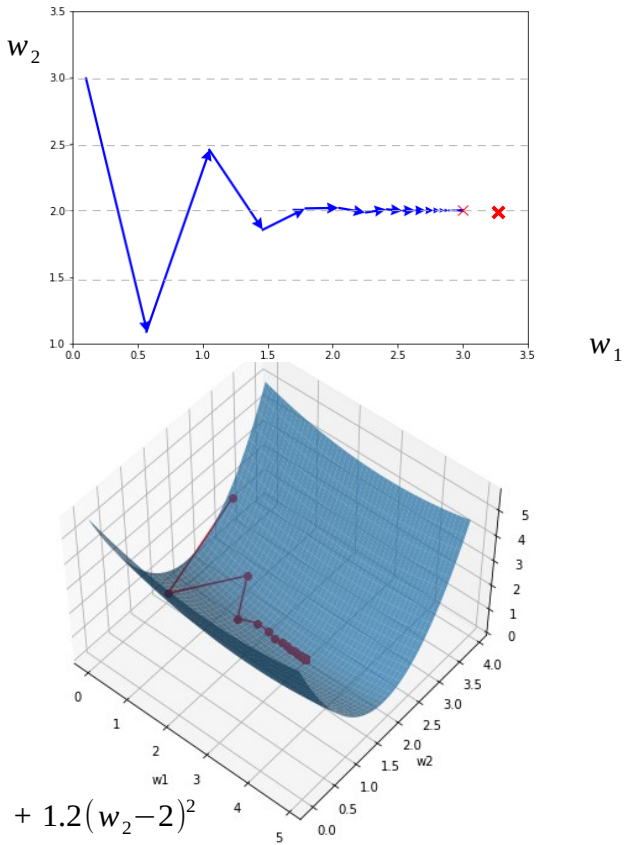
Gradient Descent optimizer

optimizers.SGD(learning_rate = 0.8, momentum=0.0)



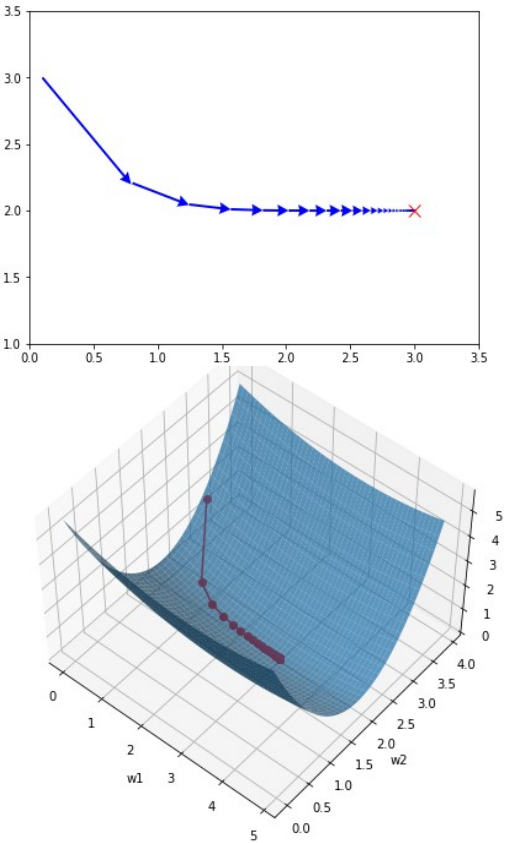
Momentum optimizer

optimizers.SGD(learning_rate = 0.8, momentum=0.2)



Adagrad optimizer

optimizers.Adagrad(learning_rate = 0.8)



$$L(w_1, w_2) = 0.1(w_1 - 3)^2 + 1.2(w_2 - 2)^2$$

MX-AI

■ Root Mean Square Propagation (RMSprop)

- RMSprop improves the disadvantage of Adagrad that the learning rate decays too quickly. When calculating Gt, we take the exponentially weighted average to the previous G and the most recent squared gradient, allowing the learning rate to decay smoothly. The larger ρ, the more past G is reflected, so the learning rate decreases more smoothly.

$$G_t = \rho G_{t-1} + (1-\rho)[\nabla L_t(w)]^2, \ (G_0=0)$$

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{G_t+\epsilon}} \nabla L_t(w)$$

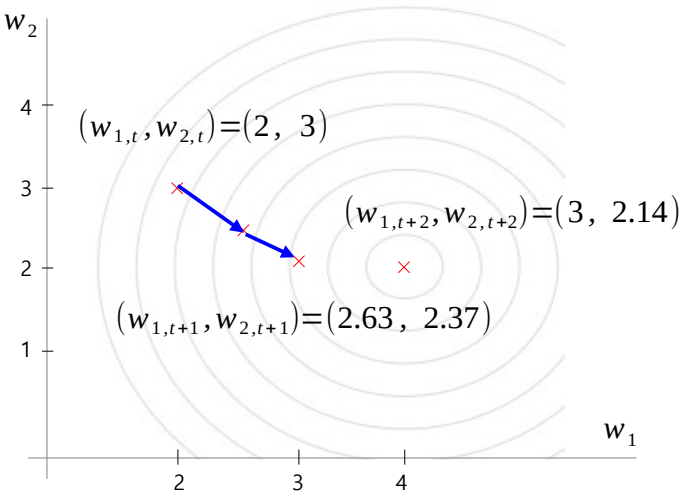
same expression

$$g_t = \nabla L_t \quad G_t = E_{ema}[g_t^2] \quad RMS(g_t) = \sqrt{G_t+\epsilon}$$

$$w_{t+1} = w_t - \frac{\alpha}{RMS(g_t)} \nabla L_t(w)$$

Because the learning rate drops less than Adagrad, setting alpha as large as Adagrad moves w too much.

$$L(w) = (w_1-4)^2 + (w_2-2)^2 \quad \nabla L(w) = [2w_1-8, \ 2w_2-4]$$



$$(w_{1,t}, w_{2,t})=(2, 3) \quad \nabla L_t(w) = [-4, 2]$$

$$G_t = 0.9\times 0 + 0.1\cdot[\nabla L_t(w)]^2 = [1.6, 0.4], \ (\rho = 0.9)$$

$$w_{1,t+1} = w_{1,t} - \frac{\alpha}{\sqrt{1.6}} \nabla_{w_1} L_t = 2 - 0.158\cdot(-4) = 2.63, \ (\alpha=0.2, \ \epsilon=0)$$

$$w_{2,t+1} = w_{2,t} - \frac{\alpha}{\sqrt{0.4}} \nabla_{w_1} L_t = 3 - 0.316\times 2 = 2.37$$

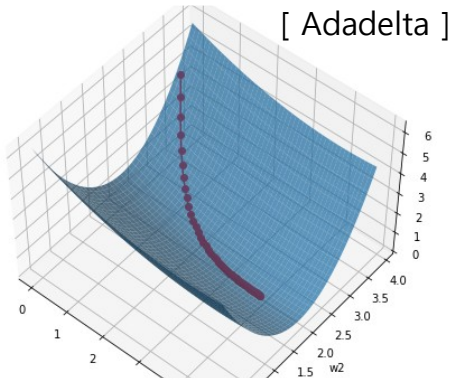
$$(w_{1,t+1}, w_{2,t+1})=(2.63, 2.37) \quad \nabla L_{t+1}(w) = (-2.74, 0.74)$$

$$G_{t+1} = 0.9\times[1.6, 0.4] + 0.1\times[7.5, 0.548] = [2.19, 0.415]$$

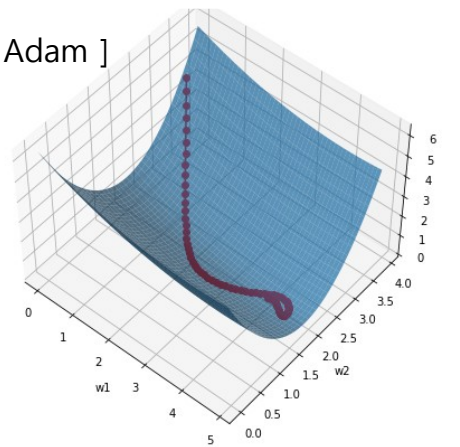
$$w_{1,t+2} = w_{1,t+1} - \frac{\alpha}{\sqrt{2.19}} \nabla_{w_1} L_{t+1} = 2.63 - \frac{0.2}{\sqrt{2.19}}\cdot(-2.74) = 3.00$$

$$w_{2,t+2} = w_{2,t+1} - \frac{\alpha}{\sqrt{0.415}} \nabla_{w_2} L_{t+1} = 2.37 - \frac{0.2}{\sqrt{0.415}}\times 0.74 = 2.14$$

[Adadelta]



[Adam]



2. Optimizers

Part 3: Adadelta, Adam

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

3. ADADELTA METHOD

The idea presented in this paper was derived from ADAGRAD [3] in order to improve upon the two main drawbacks of the method: 1) the continual decay of learning rates throughout training, and 2) the need for a manually selected global learning rate. After deriving our method we noticed several similarities to Schaul *et al.* [6], which will be compared to below.

In the ADAGRAD method the denominator accumulates the squared gradients from each iteration starting at the beginning of training. Since each term is positive, this accumulated sum continues to grow throughout training, effectively shrinking the learning rate on each dimension. After many iterations, this learning rate will become infinitesimally small.

3.1. Idea 1: Accumulate Over Window

Instead of accumulating the sum of squared gradients over all time, we restricted the window of past gradients that are accumulated to be some fixed size w (instead of size t where t is the current iteration as in ADAGRAD). With this windowed accumulation the denominator of ADAGRAD cannot accumulate to infinity and instead becomes a local estimate using recent gradients. This ensures that learning continues to make progress even after many iterations of updates have been done.

Since storing w previous squared gradients is inefficient, our methods implements this accumulation as an exponentially decaying average of the squared gradients. Assume at time t this running average is $E[g^2]_t$ then we compute:

$$E[g^2]_t = \rho\, E[g^2]_{t-1} + (1-\rho)\, g_t^2 \tag{8}$$

where ρ is a decay constant similar to that used in the momentum method. Since we require the square root of this quantity in the parameter updates, this effectively becomes the RMS of previous squared gradients up to time t:

$$\text{RMS}[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{9}$$

where a constant ϵ is added to better condition the denominator as in [5]. The resulting parameter update is then:

$$\Delta x_t = -\frac{\eta}{\text{RMS}[g]_t}\, g_t \tag{10}$$

3.2. Idea 2: Correct Units with Hessian Approximation

When considering the parameter updates, Δx, being applied to x, the units should match. That is, if the parameter had some hypothetical units, the changes to the parameter should be changes in those units as well. When considering SGD, Momentum, or ADAGRAD, we can see that this is not the case. The units in SGD and Momentum relate to the gradient, not the parameter:

$$\text{units of } \Delta x \propto \text{units of } g \propto \frac{\partial f}{\partial x} \propto \frac{1}{\text{units of } x} \tag{11}$$

assuming the cost function, f, is unitless. ADAGRAD also does not have correct units since the update involves ratios of gradient quantities, hence the update is unitless.

In contrast, second order methods such as Newton's method that use Hessian information or an approximation to the Hessian do have the correct units for the parameter updates:

$$\Delta x \propto H^{-1} g \propto \frac{\frac{\partial f}{\partial x}}{\frac{\partial^2 f}{\partial x^2}} \propto \text{units of } x \tag{12}$$

Noticing this mismatch of units we considered terms to add to Eqn. 10 in order for the units of the update to match the units of the parameters. Since second order methods are correct, we rearrange Newton's method (assuming a diagonal Hessian) for the inverse of the second derivative to determine the quantities involved:

$$\Delta x = \frac{\frac{\partial f}{\partial x}}{\frac{\partial^2 f}{\partial x^2}} \Rightarrow \frac{1}{\frac{\partial^2 f}{\partial x^2}} = \frac{\Delta x}{\frac{\partial f}{\partial x}} \tag{13}$$

Since the RMS of the previous gradients is already represented in the denominator in Eqn. 10 we considered a measure of the Δx quantity in the numerator. Δx_t for the current time step is not known, so we assume the curvature is locally smooth and approximate Δx_t by compute the exponentially decaying RMS over a window of size w of previous Δx to give the ADADELTA method:

$$\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t}\, g_t \tag{14}$$

where the same constant ϵ is added to the numerator RMS as well. This constant serves the purpose both to start off the first

MX-AI

Optimizer: Adaptive Delta (Adadelta)

- Matthew D. Zeiler, 2012, "Adadelta : An adaptive learning rate method"

3. ADADELTA METHOD

In the ADAGRAD method the denominator accumulates the squared gradients from each iteration starting at the beginning of training. Since each term is positive, this accumulated sum continues to grow throughout training, effectively shrinking the learning rate on each dimension. After many iterations, this learning rate will become infinitesimally small.

3.1. Idea 1: Accumulate over Window

Instead of accumulating the sum of squared gradients over all time, we restricted the window of past gradients that are accumulated to be some fixed size T (instead of size t where t is the current iteration as in ADAGRAD). With this windowed accumulation the denominator of ADAGRAD cannot accumulate to infinity and instead becomes a local estimate using recent gradients. This ensures that learning continues to make progress even after many iterations of updates have been done. Since storing T previous squared gradients is inefficient, our methods implements this accumulation as an exponentially decaying average of the squared gradients.

- **Adagrad**

$$G_t = G_{t-1} + [\nabla L_t(w)]^2, \ (G_0=0)$$

$$w_{t+1} = w_t + \Delta w_t \qquad \Delta w_t = -\frac{\alpha}{\sqrt{G_t+\epsilon}} \nabla L_t$$

- **Idea 1:**

$$G_t = \rho G_{t-1} + (1-\rho)[\nabla L_t(w)]^2, \ (G_0=0)$$

$$g_t = \nabla L_t \qquad G_t = E_{ema}[g_t^2] \qquad RMS(g_t) = \sqrt{G_t+\epsilon}$$

$$w_{t+1} = w_t + \Delta w_t \qquad \Delta w_t = -\frac{\alpha}{RMS(g_t)} g_t$$

MX-AI

Optimizer: Adaptive Delta (Adadelta)

- **Adagrad**

$$G_t = G_{t-1} + [\nabla L_t(w)]^2, \ (G_0=0)$$

$$w_{t+1} = w_t + \Delta w_t \qquad \Delta w_t = -\frac{\alpha}{\sqrt{G_t+\epsilon}} \nabla L_t \quad \text{--- (10)}$$

3.2. Idea 2: Correct Units with Hessian Approximation

When considering the parameter updates, Δw, being applied to w, the units should match. That is, if the parameter had some hypothetical units, the changes to the parameter should be changes in those units as well. When considering SGD, Momentum, or ADAGRAD, we can see that this is not the case. The units in SGD and Momentum relate to the gradient, not the parameter:

$$\text{units of } \Delta w \propto \text{units of g} \propto \frac{\partial L}{\partial w} \propto \frac{1}{\text{units of w}} \quad \text{--- (11)}$$

assuming the cost function, L, is unitless. ADAGRAD also does not have correct units since the update involves ratios of gradient quantities, hence the update is unitless. In contrast, second order methods such as Newton's method that use Hessian information or an approximation to the Hessian do have the correct units for the parameter updates:

$$\Delta w \propto H^{-1} g \propto \frac{\frac{\partial L}{\partial w}}{\frac{\partial^2 L}{\partial w^2}} \propto \text{units of w} \quad \text{--- (12)}$$

Noticing this mismatch of units we considered terms to add to Eqn. 10 in order for the units of the update to match the units of the parameters. Since second order methods are correct, we rearrange Newton's method (assuming a diagonal Hessian) for the inverse of the second derivative to determine the quantities involved:

$$\Delta w = \frac{\frac{\partial L}{\partial w}}{\frac{\partial^2 L}{\partial w^2}} \rightarrow \frac{\partial L}{\partial w} = \frac{\partial^2 L}{\partial w^2}\Delta w \rightarrow \frac{1}{(\frac{\partial^2 L}{\partial w^2})} = \frac{\Delta w}{(\frac{\partial L}{\partial w})} \quad \text{--- (13)}$$

Since the RMS of the previous gradients is already represented in the denominator in equation 10. we considered a measure of the Δw quantity in the numerator. Δwt for the current time step is not known, so we assume the curvature is locally smooth and approximate Δwt by compute the exponentially decaying RMS over a window of size T of previous Δw to give the ADADELTA method:

$$\Delta w_t = -H^{-1} g_t = -\frac{RMS(\Delta w_t)}{RMS(g_t)} g_t \quad \text{--- (14)} \qquad H^{-1} = \frac{\Delta w}{(\frac{\partial L}{\partial w})} \rightarrow -\frac{RMS(\Delta w_t)}{RMS(g_t)}$$

$$S_{t-1} = \rho S_{t-2} + (1-\rho)(\Delta w_{t-1})^2, \ (S_0=0, \ \Delta w_0=0)$$

$$RMS(\Delta w_{t-1}) = \sqrt{S_{t-1}+\epsilon}$$

$$\Delta w_t = -\frac{\sqrt{S_{t-1}+\epsilon}}{\sqrt{G_t+\epsilon}} g_t \qquad w_{t+1} = w_t + \Delta w_t$$

MX-AI

■ Adaptive Delta (Adadelta) – Example

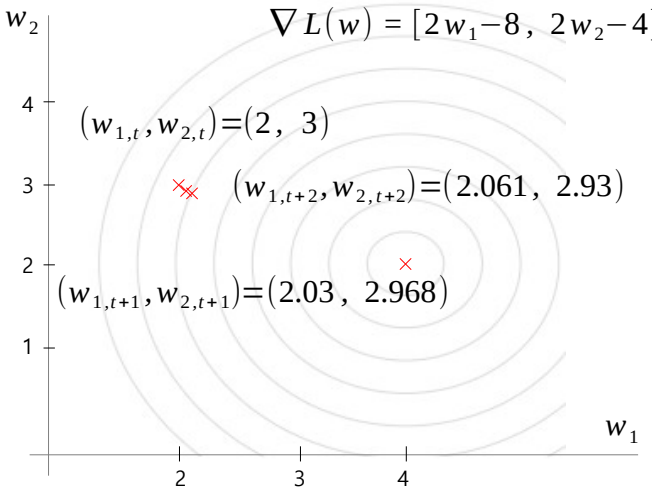
$$G_t = \rho G_{t-1} + (1-\rho)[\nabla L_t(w)]^2, \ (G_0=0)$$

$$S_{t-1}=\rho S_{t-2} + (1-\rho)(\Delta w_{t-1})^2, \ (S_0=0, \ \Delta w_0=0)$$

$$\Delta w_t = -\frac{\sqrt{S_{t-1}+\epsilon}}{\sqrt{G_t+\epsilon}} \nabla L_t(w) \qquad w_{t+1} = w_t + \Delta w_t$$

$$L(w) = (w_1-4)^2 + (w_2-2)^2$$

$$\nabla L(w) = [2w_1-8, \ 2w_2-4]$$



$(w_{1,t}, w_{2,t})=(2, \ 3) \quad \nabla L_t(w) = [-4, \ 2]$

$G_t = 0.9\times 0 + 0.1\cdot[\nabla L_t(w)]^2 = [1.6, \ 0.4], \ (\rho = 0.9)$

$S_{t-1} = 0.9\times 0 + 0.1\times 0 = [0, \ 0]$

Corresponds to the initial learning rate alpha of other optimizers. (α = 0.01).

$$w_{1,t+1} = w_{1,t} - \frac{\sqrt{0.0001}}{\sqrt{1.6001}} \nabla_{w_1} L_t = 2 - 0.0079\cdot(-4) = 2.03, \ (\epsilon=10^{-4})$$

$$w_{2,t+1} = w_{2,t} - \frac{\sqrt{0.0001}}{\sqrt{0.4001}} \nabla_{w_1} L_t = 3 - 0.0158\times 2 = 2.968$$

$(w_{1,t+1}, w_{2,t+1})=(2.03, \ 2.968) \quad \nabla L_{t+1}(w) = (-3.94, \ 1.936)$

$G_{t+1} = 0.9\times[1.6, \ 0.4] + 0.1\times[15.52, \ 3.75] = [2.99, \ 0.735]$

$S_t = 0.9\times 0 + 0.1\cdot[(2-2.03)^2, \ (3-2.968)^2] = [0.00009, \ 0.0001]$

$$w_{1,t+2} = w_{1,t+1} - \frac{\sqrt{0.00009+0.0001}}{\sqrt{2.99+0.0001}} \nabla_{w_1} L_{t+1} = 2.03 - 0.007968\times(-3.94) = 2.061$$

$$w_{2,t+2} = w_{2,t+1} - \frac{\sqrt{0.0001+0.0001}}{\sqrt{0.735+0.0001}} \nabla_{w_2} L_{t+1} = 2.968 - 0.0165\times 1.936 = 2.93$$

MX-AI

■ Adaptive Delta (Adadelta) – Example

$$L(w) = (w_1 - 4)^2 + (w_2 - 2)^2$$

```
# [MXDL-2-03] 2.adadelta(ex).py
import numpy as np
import tensorflow as tf
from tensorflow.keras import optimizers

# initial point and the target point
w0 = np.array([2, 3], dtype='float32')  # initial point
wt = np.array([4, 2], dtype='float32')  # target point
w = tf.Variable(w0)                     # (w1, w2)

# create Adadelta optimizer
# Note that Adadelta 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.
opt = optimizers.Adadelta(learning_rate=1.0, rho = 0.9, epsilon=1e-4)

path = [w.numpy()]
for i in range(10):
    with tf.GradientTape() as tape:
        loss = tf.reduce_sum(tf.square(w - wt))
        dw = tape.gradient(loss, [w])

    opt.apply_gradients(zip(dw, [w]))
    path.append(w.numpy())
path = np.array(path)

print(path)
```

```
Results:

[[2.        3.       ]
 [2.0316217 2.9683812]
 [2.063819  2.9364393]
 [2.0962355 2.9046328]
 [2.1286917 2.8731945]
 [2.161079  2.8422616]
 [2.193325  2.8119214]
 [2.2253795 2.7822316]
 [2.2572048 2.7532303]
 [2.2887726 2.7249427]
 [2.3200612 2.697384 ]]
```

These match well with the calculations by hand on the previous page.

Calculations by hand:

$$(w_{1,t}, w_{2,t}) = (2,\ 3)$$

$$(w_{1,t+1}, w_{2,t+1}) = (2.03,\ 2.968)$$

$$(w_{1,t+2}, w_{2,t+2}) = (2.061,\ 2.93)$$

MX-AI

■ Adaptive Moment Estimation (Adam)

- Adam was proposed by Kingma and Ba in 2014 and is a combination of adaptive optimizer and momentum optimizer. Adam takes advantage of the best of both optimizers. (Diederik P. Kingma and Jimmy Lei Ba, 2014, "Adam : A method for stochastic optimization")

$$g_t = \nabla L(w_t)$$

$$m_t = \beta m_{t-1} + (1-\beta) g_t, \quad (m_0 = 0)$$ ← Momentum factor

$$G_t = \rho G_{t-1} + (1-\rho) g_t^2, \quad (G_0 = 0)$$ ← Adaptive optimizer (RMSprop)

$$\hat{m}_t = \frac{m_t}{1-\beta^t} \qquad \hat{G}_t = \frac{G_t}{1-\rho^t}$$ ← bias corrected estimates (t: timesteps)

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{\hat{G}_t + \epsilon}} \cdot \hat{m}_t$$

m, G, the exponential moving averages are initialized as 0, leading to moment estimates that are biased towards zero, especially during the initial timesteps, and especially when the decay rates are small. At the beginning of the repetition, m-hat and G-hat become large, and as the repetition progresses, they become equal to the original m and G.

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. 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)

MX-AI

- Compare the paths to the target

```
# [MXDL-2-03] 3.optimizers(tf).py
import numpy as np
import tensorflow as tf
from tensorflow.keras import optimizers as op
import matplotlib.pyplot as plt

# initial point and the target point
w0 = np.array([0.1, 3.5], dtype='float32')  # initial point
wt = np.array([4, 2], dtype='float32')      # target point
w = tf.Variable(w0)                         # (w1, w2)

opt = []
opt_name = ["SGD", "Momentum", "NAG", "Adagrad",
            "RMSprop", "Adadelta", "Adam"]
opt.append(op.SGD(learning_rate = 0.8, momentum=0))
opt.append(op.SGD(learning_rate = 0.8, momentum=0.2))
opt.append(op.SGD(learning_rate = 0.1, momentum=0.2, nesterov=True))
opt.append(op.Adagrad(learning_rate = 0.8))
opt.append(op.RMSprop(learning_rate = 0.2, rho=0.9))
opt.append(op.Adadelta(learning_rate=5.0, rho=0.9, epsilon=1e-4))
opt.append(op.Adam(learning_rate = 0.1, beta_1=0.9, beta_2=0.9))

def loss(w):
    return tf.reduce_sum(tf.square(w - wt) * [0.1, 1.2])

for k in range(len(opt)):
    w.assign(w0)
    path = [w.numpy()]
    for i in range(50):
        # perform automatic differentiation
        with tf.GradientTape() as tape:
            dw = tape.gradient(loss(w), [w])

        # update w by gradient descent
        opt[k].apply_gradients(zip(dw, [w]))
        path.append(w.numpy())
    path = np.array(path)

    # Draw the loss surface and the path to the optimal point.
    x, y = path[:, 0], path[:, 1]
    w1, w2 = np.meshgrid(np.arange(0,5,0.1), np.arange(0,4,0.1))
    zs = np.array([loss([a,b]).numpy()
                   for [a, b] in zip(np.ravel(w1), np.ravel(w2))])
    z = zs.reshape(w1.shape)

    # Draw the surface of the loss function
    fig = plt.figure(figsize=(7, 7))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(w1, w2, z, alpha=0.7)

    # Dwaw the path to the optimal point.
    L = np.array([loss([a, b]).numpy() for [a, b] in zip(x, y)])
    ax.plot(x, y, L, marker='o', color="r")

    ax.set_xlabel('w1')
    ax.set_ylabel('w2')
    ax.set_zlabel('loss')
    ax.set_title(opt_name[k], fontsize= 20)
    ax.azim = -50
    ax.elev = 50
    plt.show()
```

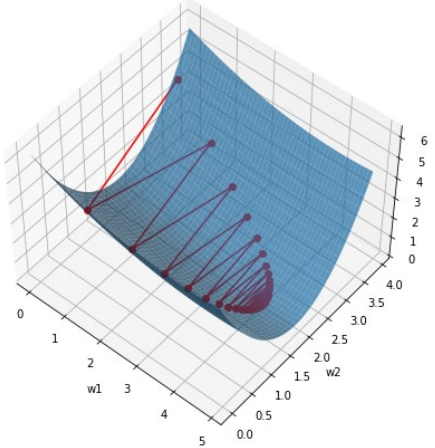
MX-AI

Compare the paths to the target

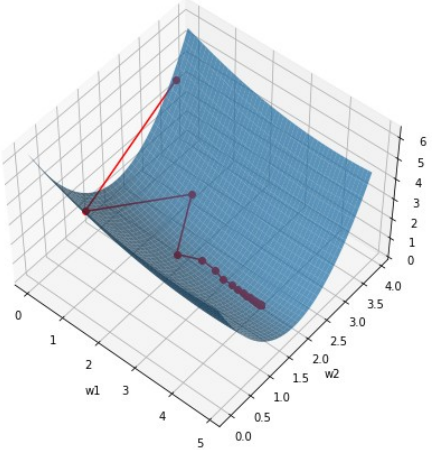
- To make the path more visible, we set the initial learning rate to be somewhat large.
- The path depends on the shape of the loss surface and the hyperparameter settings.

$$L(w_1, w_2) = 0.1(w_1 - 4)^2 + 1.2(w_2 - 2)^2$$

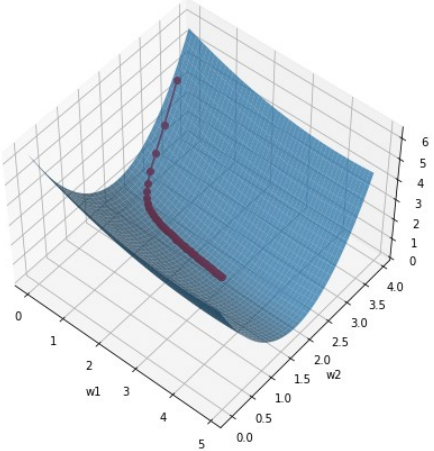
SGD



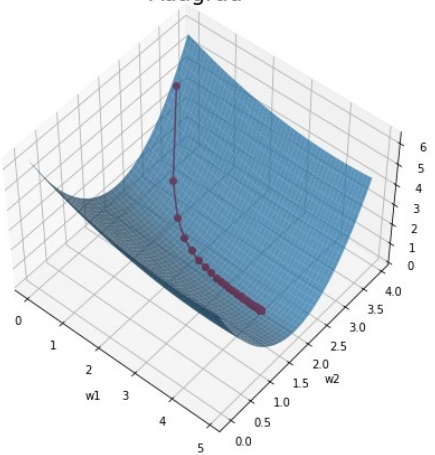
Momentum



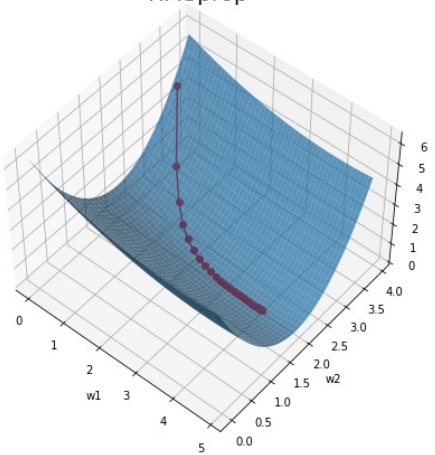
NAG



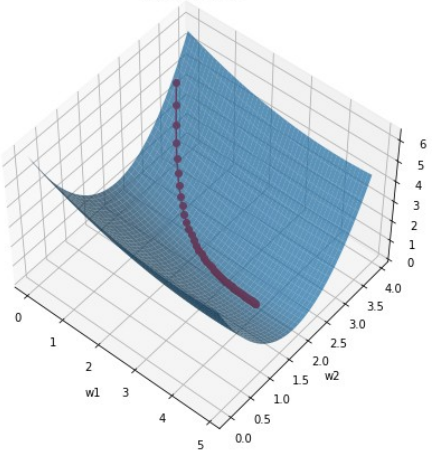
Adagrad



RMSprop



Adadelta



Adam