Enhancing Optimization Algorithms

Momentum-Accelerated Gradient Descent

Momentum-Accelerated Gradient Descent

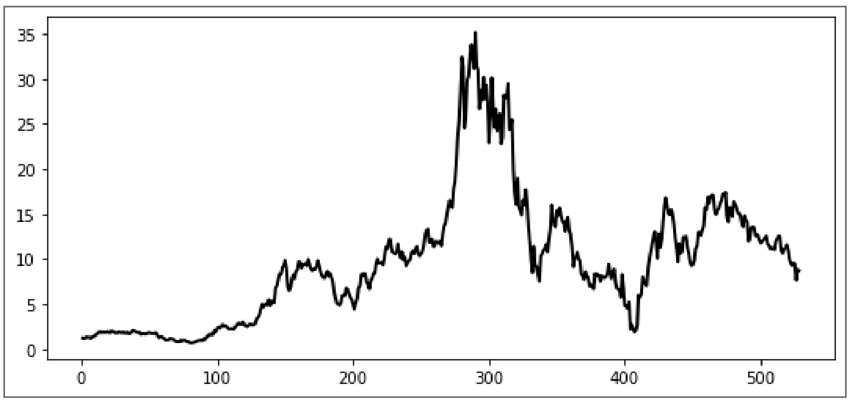
A fundamental issue with the direction of the negative gradient: depending on the function being minimized, it can oscillate rapidly, leading to zig-zagging gradient descent steps that slow down minimization.

- A popular enhancement address this zig-zagging issue of the standard gradient descent step is **momentum acceleration**.
- The core idea comes from a tool for *smoothing time series data* known as the **exponential average**.

The exponential average

- A general time series data consists of a sequence of K ordered points $w^1, w^2, \ldots, w^K$.
- For example, a history of the price of a stock over $K = 528$ periods of time.

In [6]:
```
fig = plt.figure(figsize = (9,4))
plt.plot(np.arange(1,x.size + 1),x,alpha = 1,color = 'k',linewidth = 2,zorder = 2);
```



- Or, we run a local optimization method with steps $\mathbf{w}^k = \mathbf{w}^{k-1} + \alpha \mathbf{d}^{k-1}$ which produces the time series sequence of ordered points $\mathbf{w}^1, \mathbf{w}^2, \ldots, \mathbf{w}^K$ that are multi-dimensional.

- The raw values of a time series often zig-zag up and down, it is common to **smooth** them for better visualization or prior to further analysis.

- First, we consider how to compute a **cumulative average** of K input points w^1, w^2, $\ldots, w^K$, that is the average of the first two points, the average of the first three points, and so forth.

$$
\begin{array}{rc}
\text{average of the first 1 elements:} & h^1 = w^1 \\
\text{average of the first 2 elements:} & h^2 = \frac{w^1+w^2}{2} \\
\text{average of the first 3 elements:} & h^3 = \frac{w^1+w^2+w^3}{3} \\
\text{average of the first 4 elements:} & h^4 = \frac{w^1+w^2+w^3+w^4}{4} \\
\vdots & \vdots \\
\text{average of the first } k \text{ elements:} & h^k = \frac{w^1+w^2+w^3+w^4+\cdots+w^k}{k} \\
\vdots & \vdots
\end{array}
$$

- At each step here the average computation h^k *summarizes* the input points w^1 through w^k via a simple summary statistic: their sample mean.
- We *need every raw point* w^1 *through* w^k in order to compute the running average h^k.

- We can write the cumulative average by expressing h^k for $k > 1$ in a recursive manner involving only its prceding cumulative average h^{k-1} and the current time series value w^k as:

$$h^k = \frac{k-1}{k} h^{k-1} + \frac{1}{k} w^k.$$

- This is more efficient because we only need to store two values.

- In the above running average formula, the two coefficients of the update always sum to 1, i.e., $\frac{k-1}{k} + \frac{1}{k} = 1$ for all k. As k grows larger, the coefficient on h^{k-1} gets closer to 1, while the one one w^k gets closer to 0.

- To create the **exponential average**, we freeze these coefficients: i.e., the coefficient on h^{k-1} is set to a constant $\beta \in [0, 1]$, and the coefficient on w^k is set to $1 - \beta$.

$$h^k = \beta\, h^{k-1} + (1 - \beta)\, w^k.$$

- β controls a **tradeoff**: the smaller β the more our exponential average approximates the raw (zig-zagging) time series itself, while the larger β the more each subsequent average looks like its predecessor.

- By using the exponential averaging formula and substituting in the value of each preceeding value h^{k-1}, all the way back to h^1, we can 'roll back' the exponential average at each step so that h^k is expressed entirely in terms of the input values w^1 through w^k preceeding it.

$$h^k = \beta\, h^{k-1} + (1-\beta)\, w^k.$$

substituting in the same formula for $h^{k-1} = \beta\, h^{k-2} + (1-\beta)\, w^{k-1}$ into the right hand side above for h^k, we have:

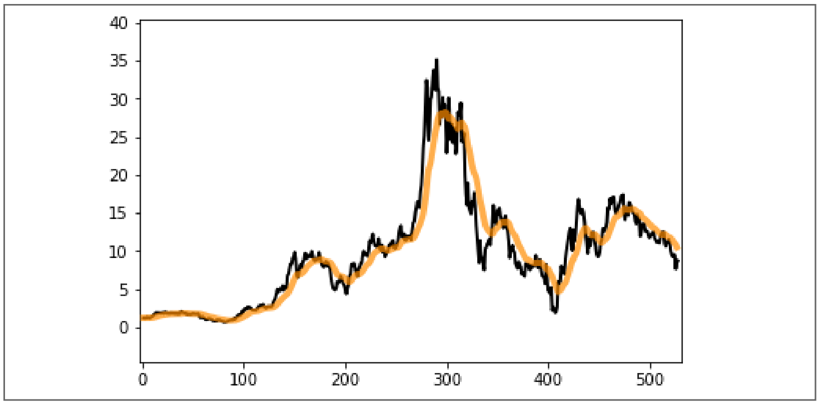
$$\begin{aligned} h^k &= \beta\, h^{k-1} + (1-\beta)\, w^k \\ &= \beta\left(\beta\, h^{k-2} + (1-\beta)\, w^{k-1}\right) + (1-\beta)\, w^k \\ &= (\beta)^2\, h^{k-2} + \beta\,(1-\beta)\, w^{k-1} + (1-\beta)\, w^k \\ &= \ldots \\ &= (\beta)^{\,k}\, w^1 + (\beta)^{\,k-1}\,(1-\beta)\, w^2 + (\beta)^{\,k-2}\,(1-\beta)\, w^3 + \cdots + \beta\,(1-\beta)\, w^{k-1} + (1-\beta)\, w^k \end{aligned}$$

- Similarly, the exponential average of a time series of general N dimensional points $\mathbf{w}^1, \mathbf{w}^2, \ldots, \mathbf{w}^K$ can be computed by initializing $\mathbf{h}^1 = \mathbf{w}^1$ and then for $k > 1$ building $\mathbf{h}^k$ as

$$\mathbf{h}^k = \beta\,\mathbf{h}^{k-1} + (1-\beta)\,\mathbf{w}^k.$$

In [7]:

```
def exponential_average(x,alpha):
    h = [x[0]]
    for p in range(len(x) - 1):
        # get next element of input series
        x_p = x[p]
        # make next hidden state
        h_p = alpha*h[-1] + (1 - alpha)*x_p
        h.append(h_p)
    return np.array(h)
# produce moving average time series
alpha = 0.9
h = exponential_average(x,alpha)
demo_1.animate_exponential_ave(x,h,savepath=video_path_1)
```



In [8]:

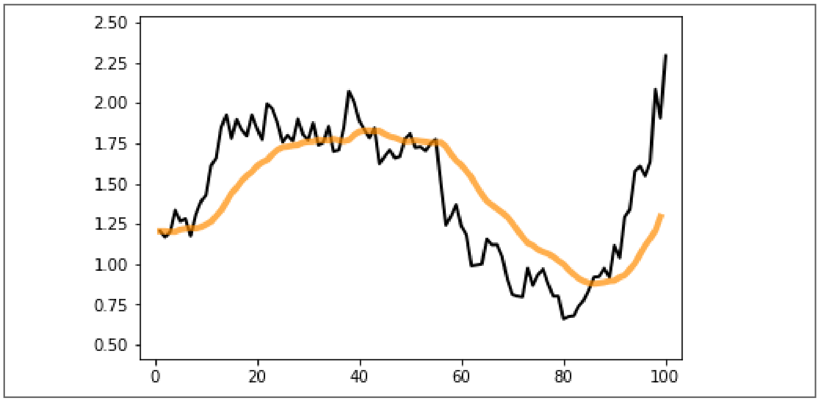
```
show_video(video_path_1, width=800)
```

Out[8]:

We zoom-in for the first 100 points.

In [9]:

```
demo_2.animate_exponential_ave(x[:100],h[:100],savepath=video_path_2)
```



In [10]:

```
show_video(video_path_2, width=800)
```

Out[10]:

Ameliorating the zig-zag behavior of gradient descent

The gradient descent update rule

$$\mathbf{w}^{\,k} = \mathbf{w}^{\,k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$$

suffers from zig-zagging behavior that slows progress of minimization.

- Both our sequence of gradient descent steps *and* the negative gradient directions themselves are both *time series*.

- If we take K steps of gradient descent using the form above we do create an time series of ordered *gradient descent steps* $\mathbf{w}^1, \mathbf{w}^2, \ldots, \mathbf{w}^K$ and descent directinos $-\nabla g\left(\mathbf{w}^0\right), -\nabla g\left(\mathbf{w}^1\right), \ldots, -\nabla g\left(\mathbf{w}^{K-1}\right)$.

- To ameliorate some of the zig-zagging behavior of our gradient descent steps $\mathbf{w}^1, \mathbf{w}^1, \ldots, \mathbf{w}^K$ we could compute their *exponential average*.

- However we do not want to smooth the gradient descent steps *after* they have been created - the 'damage is already done' in the sense that the zig-zagging has already slowed the progress of a gradient descent run.

- Instead what we want is to smooth the steps *as they are created*, so that our algorithm makes more progress in minimization.

- The root cause of zig-zagging gradient descent steps zig-zag is the oscillating nature of the (negative) gradient directions themselves.

- If the descent directions $-\nabla g\left(\mathbf{w}^0\right), -\nabla g\left(\mathbf{w}^1\right), \ldots, -\nabla g\left(\mathbf{w}^{K-1}\right)$ zig-zag, so to will the gradient descent steps.

- Using the exponential average, we will to create our smoothed descent directions as they are created.

- We initialize $\mathbf{d}^0 = -\nabla g\left(\mathbf{w}^0\right)$ and then for $k-1>0$ the $(k-1)^{th}$ exponentially averaged descent direction $\mathbf{d}^{k-1}$ takes the form:

$$\mathbf{d}^{k-1} = \beta\,\mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right)$$

- The update in our gradient descent now becomes:

$$\begin{aligned} \mathbf{d}^{k-1} &= \beta\,\mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right) \\ \mathbf{w}^{\,k} &= \mathbf{w}^{\,k-1} + \alpha\,\mathbf{d}^{k-1}. \end{aligned}$$

- This adjustment to gradient descent is often called **momentum accelerated gradient descent**. The term ''momentum'' refers to the exponentially averaged descent direction $\mathbf{d}^{k-1}$.

$$\mathbf{d}^{k-1} = \beta\,\mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right)$$

The choice of $\beta \in [0,1]$ provides a trade-off:

- The smaller β is chosen the *more* the exponential average resembles the actual sequence of negative descent directions since *more* of each negative gradient direction is used in the update.
- The larger β is chosen the *less* these exponentially averaged descent steps resemble the negative gradient directions, since each update will use *less* of each subsequent negative gradient direction.
- Often in practice larger values of β are used, in the range $[0.7, 1]$.

- In practice this step is also written slightly differently: instead of averaging the *negative* gradient directions, the gradient itself is exponentially averaged, and then the *step* is taken in their *negative* direction.

- This means that we initialize our exponential average at the first *negative* descent direction $\mathbf{d}^0 = -\nabla g\left(\mathbf{w}^0\right)$ and for $k-1>0$ the general descent direction and corresponding step is computed as

$$
\begin{aligned}
\mathbf{d}^{k-1} &= \beta\,\mathbf{d}^{k-2} + (1-\beta)\,\nabla g\left(\mathbf{w}^{k-1}\right) \\
\mathbf{w}^{\,k} &= \mathbf{w}^{\,k-1} - \alpha\,\mathbf{d}^{k-1}.
\end{aligned}
$$

Example: Accelerating gradient descent on a simple quadratic

Using a quadratic function of the form

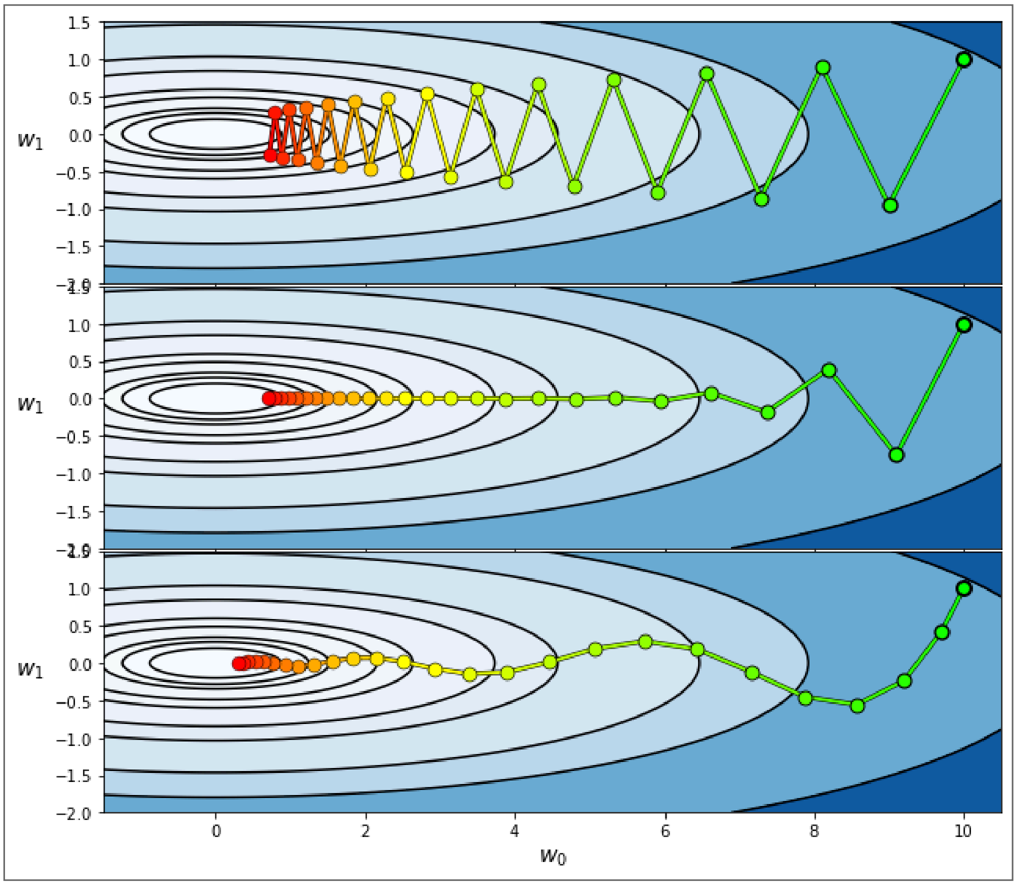
$$g(\mathbf{w}) = a + \mathbf{b}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}.$$

where $a = 0$, $\mathbf{b} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathbf{C} = \begin{bmatrix} 0.5 & 0 \\ 0 & 9.75 \end{bmatrix}$.

- We run 25 gradient descent steps, and also compare two run of *momentum accelerated graient descent* with two choices for $\beta \in \{0.2, 0.7\}$.
- All three runs are initialized at the same point $\mathbf{w}^0 = \begin{bmatrix} 10 \\ 1 \end{bmatrix}$ and use the same learning rate $\alpha = 10^{-1}$.

In [15]:

```
static_plotter.two_input_contour_vert_plots(gs,histories,num_contours = 25,xmin = -1.5,xmax = 10.5,ymin = -2.0,ymax = 1.5)
```



Normalized Gradient Descent

Normalizing Gradient Descent

- A fundamental issue of gradient descent is that the magnitude of the (negative) gradient vanishes near stationary points.
- Gradient descent crawls slowly near stationary points, and it can halt near saddle points.
- An idea to overcome this issue is simply ignoring the magnitude at each step by normalizing it.

Normalizing out the full gradient magnitude

- The length of a standard gradient descent step is *proportional to the magnitude of the gradient*:

$$\text{length of standard gradient descent step:} \quad \alpha \left\| \nabla g(\mathbf{w}^{k-1}) \right\|_2.$$

- This explains why gradient descent crawls slowly near stationary points because near such points the gradient vanishes, i.e, $\nabla g(\mathbf{w}^{k-1}) \approx \mathbf{0}$. What if we ignore the magnitude of the gradient, and just travel in the direction of negative gradient.

- We can normalize out the full magnitude of the gradient in our standard gradient descent step, giving a **normalized gradient descent step** of the form:

$$\mathbf{w}^{k} = \mathbf{w}^{k-1} - \alpha \frac{\nabla g(\mathbf{w}^{k-1})}{\left\| \nabla g(\mathbf{w}^{k-1}) \right\|_2}$$

- We indeed ignore the magnitude of the gradient, since the length of such step is:

$$\left\| \mathbf{w}^{k} - \mathbf{w}^{k-1} \right\|_2 = \left\| \left(\mathbf{w}^{k-1} - \alpha \frac{\nabla g(\mathbf{w}^{k-1})}{\left\| \nabla g(\mathbf{w}^{k-1}) \right\|_2} \right) - \mathbf{w}^{k-1} \right\|_2 = \left\| -\alpha \frac{\nabla g(\mathbf{w}^{k-1})}{\left\| \nabla g(\mathbf{w}^{k-1}) \right\|_2} \right\|_2 = \alpha.$$

- The lenght of fully-normalized gradient descent step: α.

Normalizing out the full gradient magnitude

- We slightly re-write the fully-normalized step above as

$$\mathbf{w}^{k} = \mathbf{w}^{k-1} - \frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2}\nabla g(\mathbf{w}^{k-1})$$

- We can interpret our fully normalized step as a standard gradient descent step with a steplength / learning rate value $\frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2}$ that *adjusts itself at each step based on the magnitude of the gradient to ensure that the length of each step is precisely* α.

- In practice, it is often useful to add a small constant ϵ (e.g., 10^{-7} or smaller) to the gradient magnitude to avoid potential division by zero (where the magnitude completely vanishes)

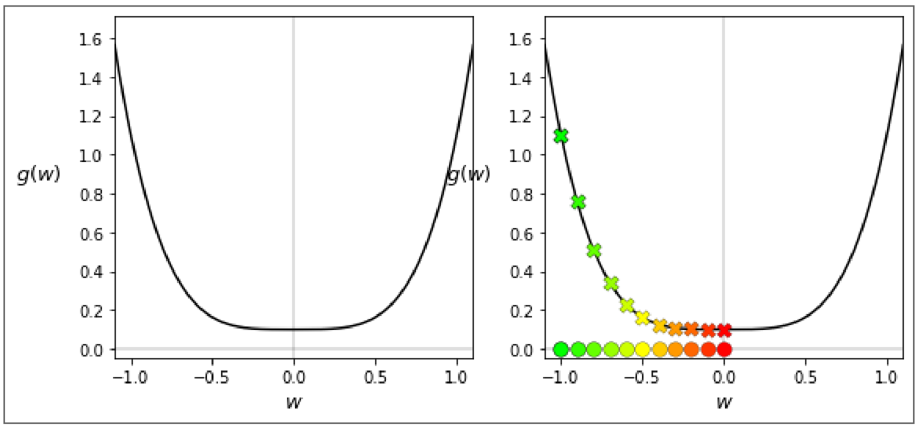
$$\mathbf{w}^{k} = \mathbf{w}^{k-1} - \frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2 + \epsilon}\nabla g(\mathbf{w}^{k-1}) \quad (1)$$

Example: Ameliorating the slow-crawling behavior of gradient descent near the minimum of a function

$$g(w) = w^4 + 0.1$$

In [20]:

```
                # what function should we play with?  Defined in the next line.
g = lambda w: w**4 + 0.1
# run gradient descent
w = -1.0; max_its = 10; alpha_choice = 0.1;
version = 'full'
weight_history,cost_history = gradient_descent(g,alpha_choice,max_its,w,version)
# make static plot showcasing each step of this run
static_plotter.single_input_plot(g,[weight_history],[cost_history],wmin = -1.1,wmax = 1.1)
```

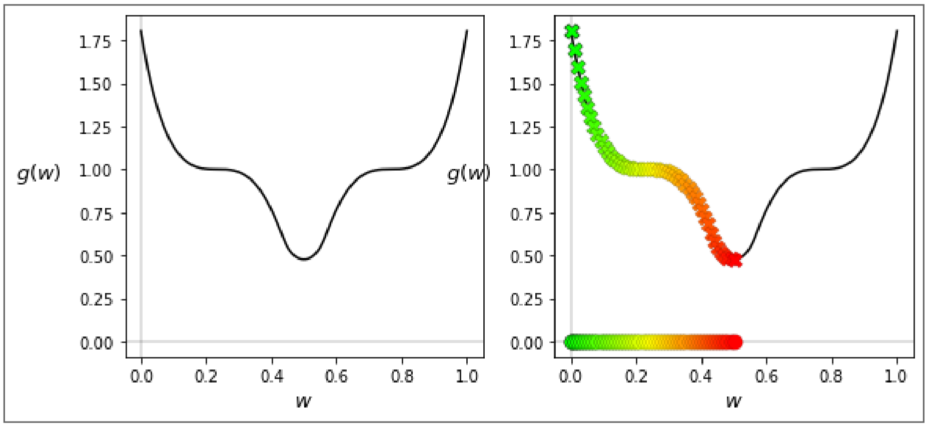


Example: Ameliorating the slow-crawling behavior of gradient descent near saddle points

$$g(w) = \text{maximum}(0, (3w - 2.3)^3 + 1)^2 + \text{maximum}(0, (-3w + 0.7)^3 + 1)^2$$

In [21]:

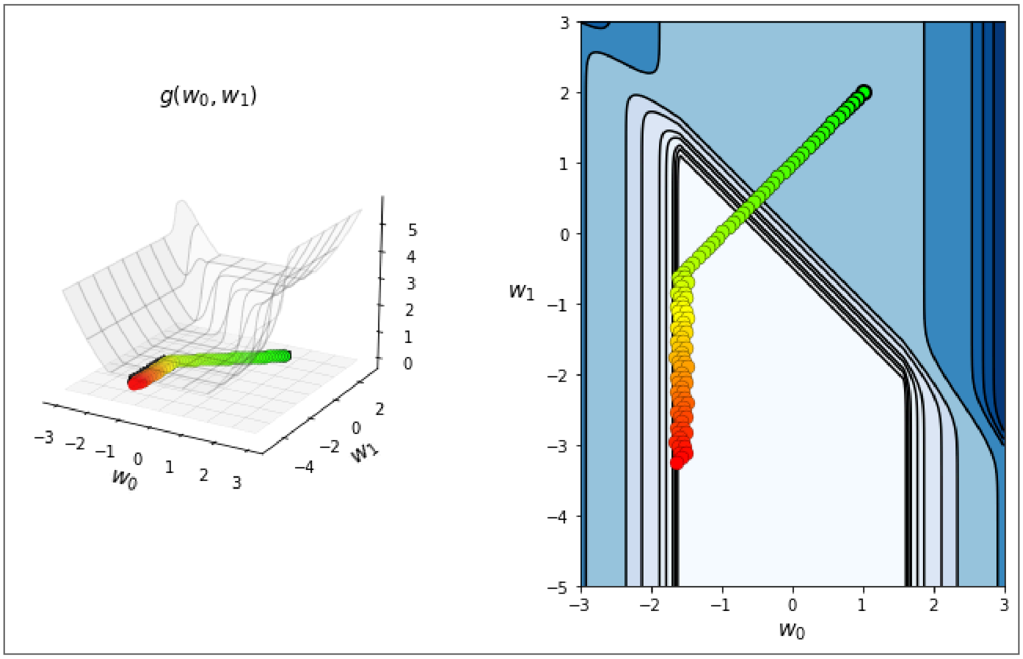
```
g = lambda w: np.maximum(0,(3*w - 2.3)**3 + 1)**2 + np.maximum(0, (-3*w + 0.7)**3 + 1)**2
demo_1.draw_2d(g=g, w_inits = [0],steplength = 0.01,max_its = 50,version = 'normalized',wmin = 0,wmax = 1.0)
```



Example: Slow-crawling behavior of gradient descent in large flat regions of a function

In [22]:

```
g = lambda w: np.tanh(4*w[0] + 4*w[1]) + max(0.4*w[0]**2,1) + 1
w = np.array([1.0,2.0]); max_its = 100; alpha_choice = 10**(-1);
version = 'full'
weight_history_1,cost_history_1 = gradient_descent(g,alpha_choice,max_its,w,version)
static_plotter.two_input_surface_contour_plot(g,weight_history_1,view = [20,300],num_contours = 20,xmin = -3,xmax = 3,ymin = -5,ymax = 3)
```



Normalizing out the magnitude component-wise

The **gradient** is a vector of N *partial derivatives*

$$\nabla g(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial w_1} g\left(\mathbf{w}\right) \\ \vdots \\ \frac{\partial}{\partial w_N} g\left(\mathbf{w}\right) \end{bmatrix} \tag{2}$$

with the j^{th} partial derivative $\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)$ defining how the gradient behaves along the j^{th} coordinate axis.

- Look at the j^{th} partial derivative when we normalize off the *full magnitude*:

$$\frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\left\Vert \nabla g\left(\mathbf{w}\right)\right\Vert_2} = \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\sqrt{\sum_{n=1}^{N}\left(\frac{\partial}{\partial w_n} g\left(\mathbf{w}\right)\right)^2}}$$

we can see that *the j^{th} partial derivative is normalized using a sum of the magnitudes of every partial derivative*.

$$\frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\left\|\nabla g\left(\mathbf{w}\right)\right\|_2} = \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\sqrt{\sum_{n=1}^{N}\left(\frac{\partial}{\partial w_n} g\left(\mathbf{w}\right)\right)^2}}$$

- If the j^{th} partial derivative is already small, this normalization will erase all of its contribution to the descent step.
- This is problematic when dealing with functions containing regions that are flat with respect to only some of partial derivative directions.

- An alternative is to normalize out the magnitude component-wise:

$$\frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right)^2}} = \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)}{\left|\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right|} = \text{sign}\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right)$$

- So in the j^{th} direction, the component-normalized gradent descent step is:

$$w_j^k = w_j^{k-1} - \alpha \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}} = w_j^{k-1} - \alpha\, \text{sign}\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right).$$

- We can write the entire component-wise normalized step as:

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\,\text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right)$$

- The length of a single step of this component-normalized gradient descent step is:

$$\left\|\mathbf{w}^{\,k} - \mathbf{w}^{\,k-1}\right\|_2 = \left\|-\alpha\,\text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right)\right\|_2 = \sqrt{N}\,\alpha$$

- If we slightly rewrite the j^{th} component-normalized step as:

$$w_j^k = w_j^{k-1} - \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}}\,\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right).$$

  then we can interpret the component-normalized step as a standard gradient descent step with an individual steplength value $\frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2}}$ per component that all *adjusts themselves individually at each step based on component-wise magnitude of the gradient to ensure that the length of each step is precisely* $\sqrt{N}\,\alpha$

- We write

$$\mathbf{a}^{k-1} = \begin{bmatrix} \frac{\alpha}{\sqrt{\left(\frac{\partial}{(\partial w_1}g(\mathbf{w}^{k-1})\right)^2}} \\ \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_2}g(\mathbf{w}^{k-1})\right)^2}} \\ \vdots \\ \frac{\alpha}{\sqrt{\left(\frac{\partial}{\partial w_N}g(\mathbf{w}^{k-1})\right)^2}} \end{bmatrix}$$

- The full component-normalized descent step can be written as:

$$\mathbf{w}^{\,k} = \mathbf{w}^{\,k-1} - \mathbf{a}^{k-1} \odot \nabla g(\mathbf{w}^{\,k-1})$$

where the $\odot$ symbol denotes component-wise multiplication. In practice, a small $\epsilon > 0$ is added to the denominator of each value of each entry of $\mathbf{a}^{k-1}$ to avoid division by zero.

- The above formula is equivalent to

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha \, \text{sign}\left(\nabla g\left(\mathbf{w}^{k-1}\right)\right)$$

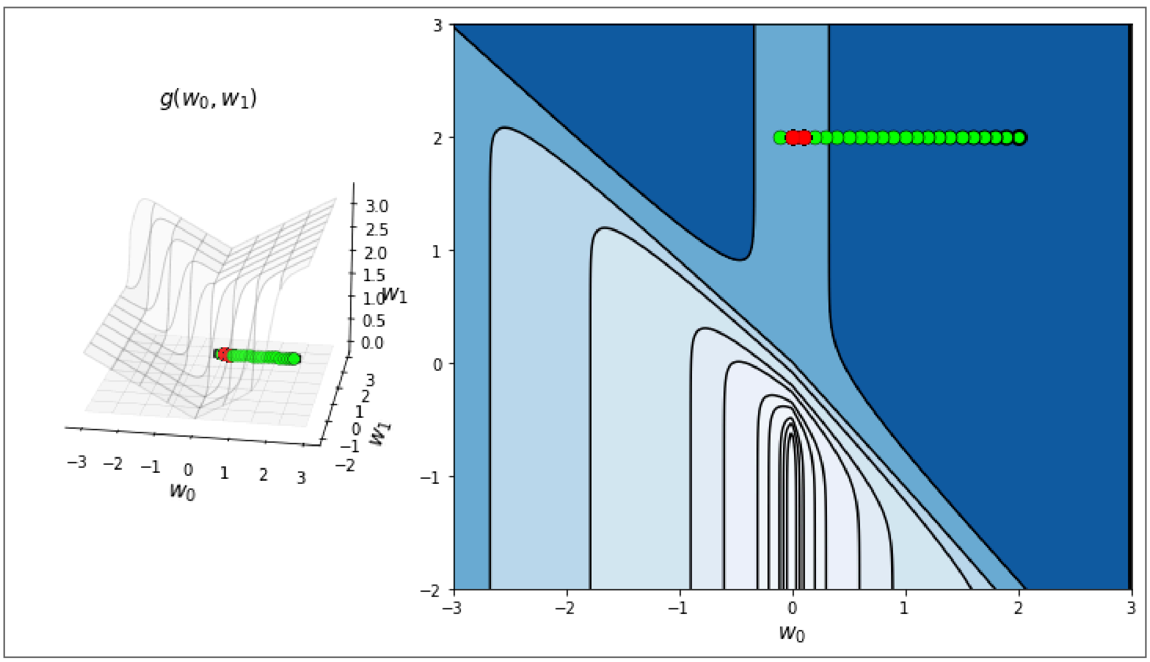
Example: Full versus component-normalized gradient descent

$$g(w_0, w_1) = \max\left(0, \tanh(4w_0 + 4w_1)\right) + \max(0, \text{abs}\left(0.4w_0\right)) + 1$$

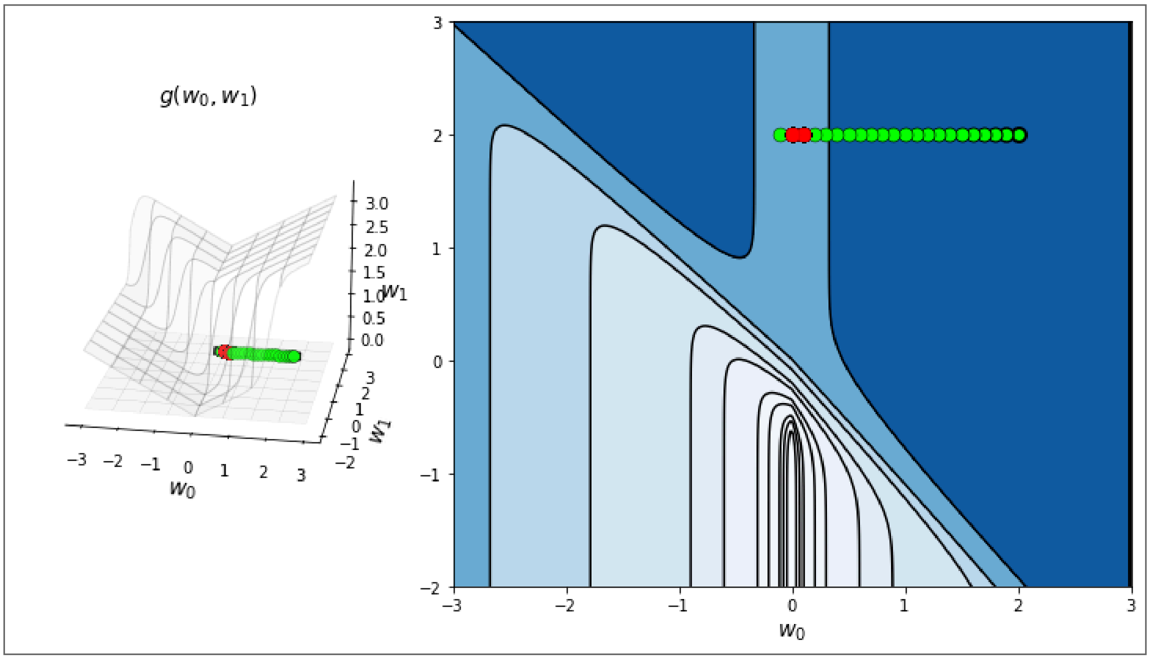
- This function is very flat along the w_1 direction for any fixed value of w_0.
- It has a very narrow valley leading toward its minima in the w_1 dimension where $w_0 = 0$.

In [24]:

```
static_plotter.two_input_surface_contour_plot(g,weight_history_1,view = [20,280],num_contours = 24,xmin = -3,xmax = 3,ymin = -2,ymax =
```

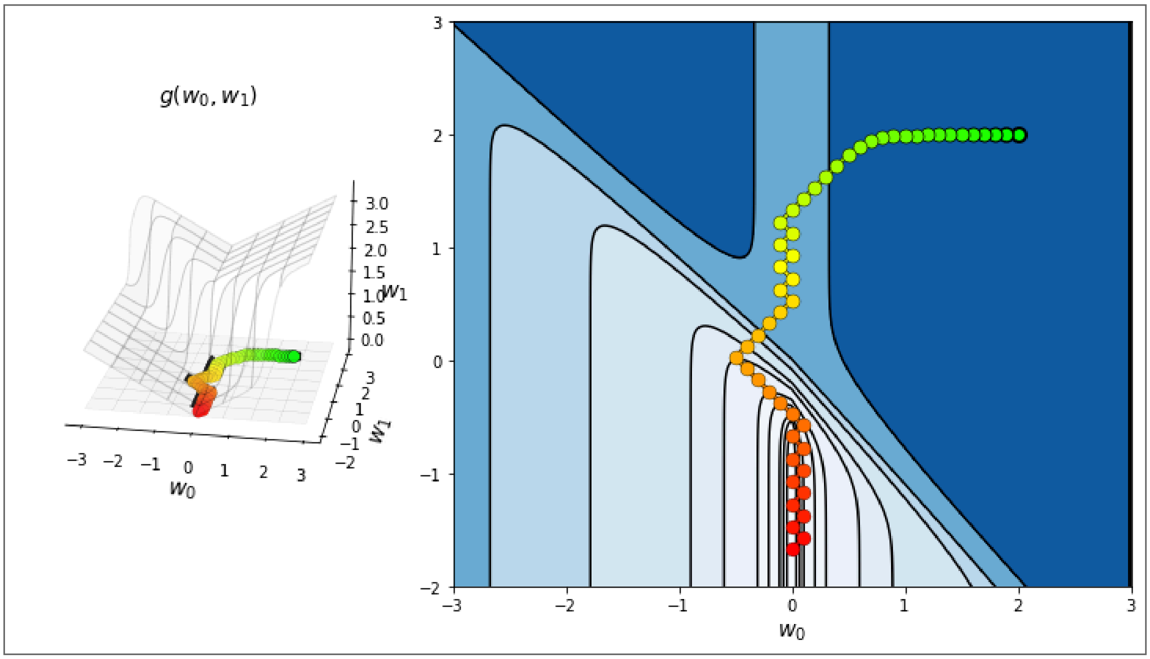


```
In [25]: static_plotter.two_input_surface_contour_plot(g,weight_history_1,view = [20,280],num_contours = 24,xmin = -3,xmax = 3,ymin = -2,ymax =
```



- When we use the fully normalized version, the magnitude of the partial derivative in w_1 is nearly zero everywhere, so fully-normalizing makes this contribution smaller and halts progress. The demo shows 1000 steps.

```
In [27]: static_plotter.two_input_surface_contour_plot(g,weight_history_1,view = [20,280],num_contours = 24,xmin = -3,xmax = 3,ymin = -2,ymax =
```



- To make progress, we need to enhance the partial derivative in the w_1 direction via the component-normalization scheme. Here we need only 50 steps.

Advanced Gradient-Based Methods

Combining momentum with normalized gradient descent

- We know that **momentum-accelerated gradient descent** can ameliorate the zig-zagging problem of standard gradient descent algorithm. The momentum-accelerated descent direction $\mathbf{d}^{k-1}$ is simply an *exponential average* of gradient descent directions taking the form

$$\begin{aligned} \mathbf{d}^{k-1} &= \beta\,\mathbf{d}^{k-2} - (1-\beta)\,\nabla g\left(\mathbf{w}^{k-1}\right) \\ \mathbf{w}^{\,k} &= \mathbf{w}^{\,k-1} + \alpha\,\mathbf{d}^{k-1} \end{aligned} \tag{3}$$

where $\beta \in [0,1]$ is typically set at a value of $\beta = 0.7$ or higher.

- We also know that **normalizing the gradient descent direction component-wise** helps to deal with the problem of standard gradient descent has when traversing **flat regions** of a function. A component-normalized gradient descent step take the form:

$$w_j^k = w_j^{k-1} - \alpha\,\frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right)^2}}$$

where in practice of course a small $\epsilon > 0$ (like e.g., $\epsilon = 10^{-8}$) is added to the denominator to avoid division by zero.

- How about combine momentum with normalizing gradient descent direction?
- For example, we can component-normalize the exponential average descent direction computed in momentum-accelerated gradient descent.
- The update for the j^{th} component of the resulting step:

$$d_j^{k-1} = \beta\, d_j^{k-2} - (1-\beta)\,\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)$$
$$d_j^{k-1} \longleftarrow \frac{d_j^{k-1}}{\sqrt{\left(d_j^{k-1}\right)^2}}$$

- With a full direction $\mathbf{d}^{k-1}$ commputed like above, we can take a descent step:

$$\mathbf{w}^{\,k} = \mathbf{w}^{\,k-1} + \alpha\, \mathbf{d}^{k-1}.$$

- There are many different ways for combining these two enhancements.

Adaptive Moment Estimation (Adam)

- Adam has component-wised normalized gradient steps that calculates exponential averages for both the descent direction and magnitude.

- We compute the j^{th} coordinate of the updated descent direction by:
    1. Computing the exponential average of the gradient descent direction d_j^k
    2. Computing the exponential average of the squared magnitude h_j^k.

$$\begin{aligned} d_j^{k-1} &= \beta_1\, d_j^{k-2} + (1-\beta_1)\, \frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right) \\ h_j^{k-1} &= \beta_2\, h_j^{k-2} + (1-\beta_2) \left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2 \end{aligned} \tag{4}$$

where β_1 and β_2 lie in the range $[0,1]$. Popular values the parameters of this update step are $\beta_1 = 0.9$, $\beta_2 = 0.999$.

- These two updates apply when $k > 1$ and should be initialized as $d_j^0 = \frac{\partial}{\partial w_j} g\left(\mathbf{w}^0\right)$ and its squared magnitude $h_j^0 = \left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^0\right)\right)^2$.

- The original Adam's publication used a slightly different initialization with bias-correction.

Adaptive Moment Estimation (Adam)

- The Adam update step is a component-normalized descent step using the exponentially average descent direction and magnitude.

- A step in the j^{th} coordinate then takes the form

$$w_j^k = w_j^{k-1} - \alpha \frac{d_j^{k-1}}{\sqrt{h_j^{k-1}}}. \tag{5}$$

where in practice of course a small $\epsilon > 0$ (like e.g., $\epsilon = 10^{-8}$) is added to the denominator to avoid division by zero.

- If we slightly re-write above as

$$w_j^k = w_j^{k-1} - \frac{\alpha}{\sqrt{h_j^{k-1}}}\, d_j^{k-1}.$$

we can interpret the Adam step as a momentum-accelerated gradient descent step with an individual steplength $\frac{\alpha}{\sqrt{h_j^{k-1}}}$ per component that all *adjusts themselves individually at each step based on component-wise exponentially normalized magnitude of the gradient*.

Root Mean Squared Propagation (RMSprop)

- In component-wise normalized gradient descent, each component of the gradient is normalized by its magnitude.

- We can normalize each component of the gradient by the **exponential average of the component-wise magnitudes** of previous gradient directions.
- The exponential average of the squared magnitude of the j^{th} partial derivative at step k as:

$$h_j^k = \gamma\, h_j^{k-1} + (1-\gamma)\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)\right)^2$$

- The **RMSprop** step is a component-wise normalized descent step using the above exponential average:

$$w_j^k = w_j^{k-1} - \alpha \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{h_j^{k-1}}}$$

where in practice of course a small $\epsilon > 0$ (like e.g., $\epsilon = 10^{-8}$) is added to the denominator to avoid division by zero.

Root Mean Squared Propagation (RMSprop)

$$w_j^k = w_j^{k-1} - \alpha \frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{h_j^{k-1}}}$$

We can re-write the above update step as:

$$w_j^k = w_j^{k-1} - \frac{\alpha}{\sqrt{h_j^{k-1}}} \frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right).$$

We can interpret the RMSprop step as a standard gradient descent step with an individual steplength value $\frac{\alpha}{\sqrt{h_j^{k-1}}}$ per component that all *adjusts themselves individually at each step based on component-wise magnitude of the gradient*.