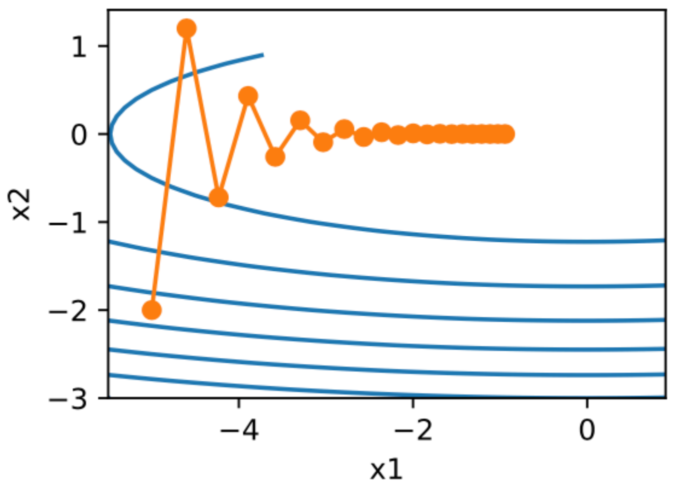
Problems with SGD

- A single learning rate η is used to update all variables
  - for the case of deep learning all optimization parameters
- Ideally, we want η_1 for x_1, η_2 for x_2, ..., η_d for x_d
  - Impossible to fix all of these by hand!
- Why's this a problem? Consider the following function:

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2.$$

  - f has its minimum at $(0, 0)$.
  - This function is *very* flat in the direction of x_1.
- Let us see what happens when we perform GD with learning rate of 0.4:

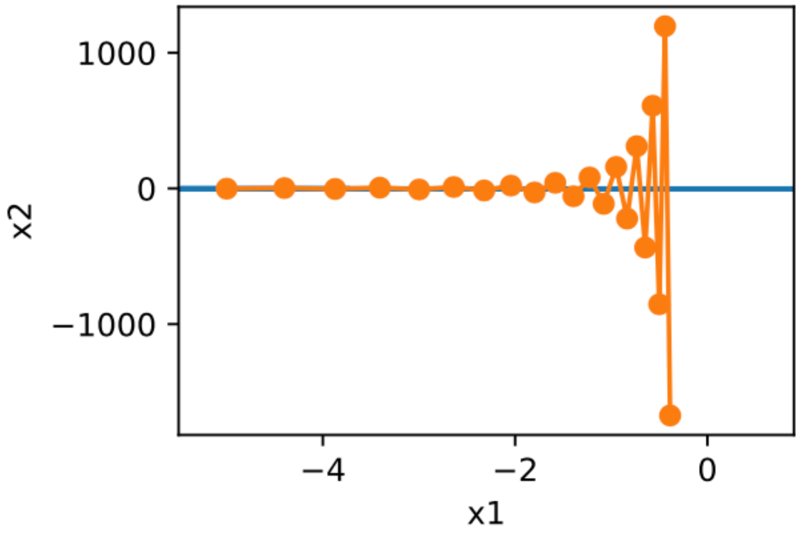
In [16]:
```
eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

mu.show_trace_2d(f_2d, mu.train_2d(gd_2d))
```



- The gradient in the x_2 direction is *much* higher and changes much more rapidly than in the horizontal x_1 direction.
- Thus we are stuck between two undesirable choices:
    - With a small learning rate we ensure that the solution does not diverge in the x_2 direction but we make poor progress in the x_1 direction.
    - With a large learning rate we progress rapidly in the x_1 direction but diverge in x_2.

- Let's increase learning rate from 0.4 to 0.6.

In [18]:
```
eta = 0.6
mu.show_trace_2d(f_2d, mu.train_2d(gd_2d))
```



- Convergence in the x_1 direction improves but the overall solution quality is much worse.

The Momentum Method

- The momentum method allows us to solve the gradient descent problem described above.

- Looking at the optimization trace above we might think that **averaging gradients over the past** would work well.

    - In the x_1 direction this will aggregate well-aligned gradients, thus increasing the distance we cover with every step.
    - In the x_2 direction where gradients oscillate, an aggregate gradient will reduce step size due to oscillations that cancel each other out.

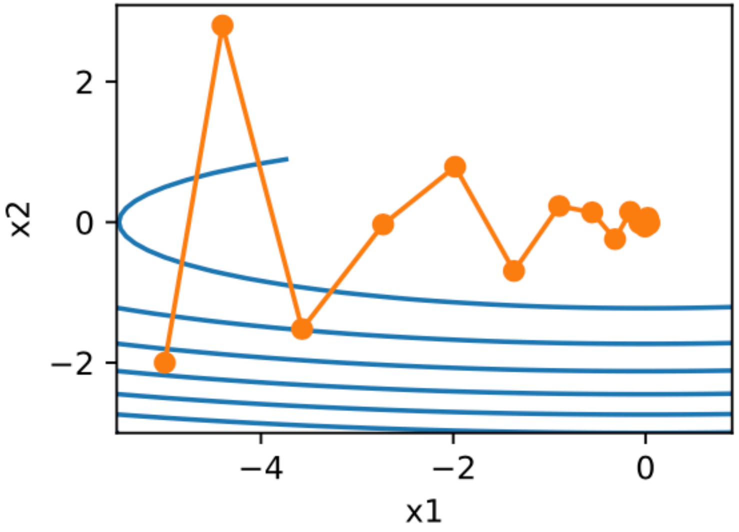
- Instead of the gradient $\mathbf{g}_t = \sum_{i=1}^{|\mathcal{B}|} \nabla f_i(\mathbf{x})$, use $\mathbf{v}_t$ to update:

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_t,$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.$$

- For $\beta = 0$ we recover regular GD descent.
- Let's consider optimizing the same function as above using GD with momemtum.

In [19]:

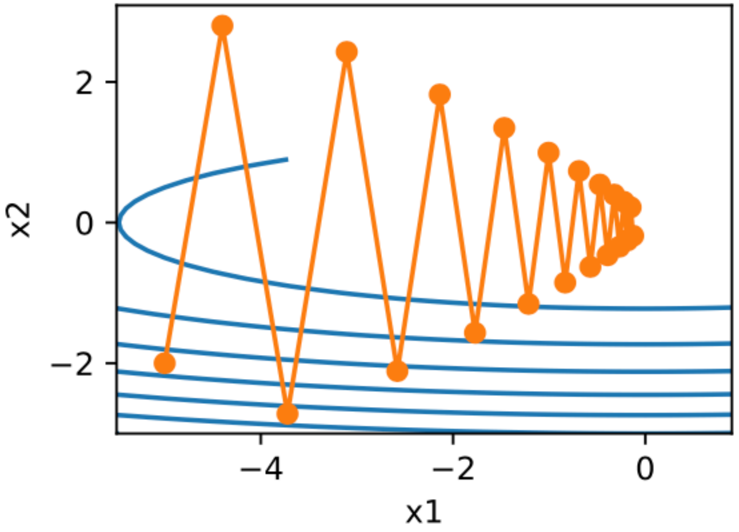
```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
mu.show_trace_2d(f_2d, mu.train_2d(momentum_2d))
```



- Even with the same learning rate that we used before, momentum still converges well!

- Let us see what happens when we decrease the momentum parameter.

In [20]:
```
eta, beta = 0.6, 0.25
mu.show_trace_2d(f_2d, mu.train_2d(momentum_2d))
```



- Halving it to $\beta = 0.25$ leads to a trajectory that barely converges at all.
    - Nonetheless, it is a lot better than without momentum (when the solution diverges).

RMS-Prop

- This is an adaptive method which produces essentially a different effective learning rate for each optimization variable $x_1, x_2, \ldots, x_d$.
- Main idea: if the anisitropic shape of the optimization function could become isotropic (like a radius), then we could use a single learning rate for all optimization variables.
- To achieve this RMS-PROP normalizes (divides) each derivative by its magnitute.
  - it actually keeps track of an average magnitude over past samples.

$$\mathbf{g}_t = \sum_{i=1}^{|\mathcal{B}|} \nabla f_i(\mathbf{x}),$$

$$\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2,$$

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta \mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}.$$

- The constant $\epsilon > 0$ is typically set to 10^{-6} to ensure that we do not suffer from division by zero.

ADAM

- Yes it's the guy from Dark (no he's not!)
- Adam combines momentum with RMS-Prop
- If you check RMS-Prop update equation above the gradient $\mathbf{g}_t$ is used.
- Instead we can use the same quantity $\mathbf{v}_t$ we used in momentum. Overall we have:

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1-\beta_2)\mathbf{g}_t^2.\end{aligned}$$

- Here β_1 and β_2 are nonnegative weighting parameters.
  - Common choices for them are $\beta_1 = 0.9$ and $\beta_2 = 0.999$.
- If we initialize $\mathbf{v}_0 = \mathbf{s}_0 = 0$ we have a significant amount of bias initially towards smaller values. To address this we use the fact that $\sum_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$ to re-normalize terms. Hence, the normalized state variables are given by:

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-\beta_2^t}.$$

- Now we have all the pieces in place to compute the updates:

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}.$$