In [1]:

```
import numpy as np
import pandas as pd
import seaborn as sns

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib_inline.backend_inline

import plotly.graph_objs as go
```

In [2]:

```
matplotlib_inline.backend_inline.set_matplotlib_formats("png2x")
mpl.style.use("default")
mpl.rcParams.update({"figure.constrained_layout.use": True})

sns.set_context("paper")
sns.set_palette("Set2")
sns.set_style("whitegrid")

plt.rc("font", family = "Malgun Gothic")
plt.rcParams["axes.unicode_minus"] = False
```

1. 미분

- **딥러닝의 이론을 공부할 때 미분을 빼놓고 이야기 할 수 없다!**
- 미분: 어떤 점에서 함수의 기울기
- 대학에 가기 전에는 1변수 함수의 미분을 배운다.(교육강사의 교육과정에서... 요즘은...?)

다음 함수 $f(x)$의 x에 대한 도함수 $\frac{df}{dx}(= f'(x))$ 계산해 보자.

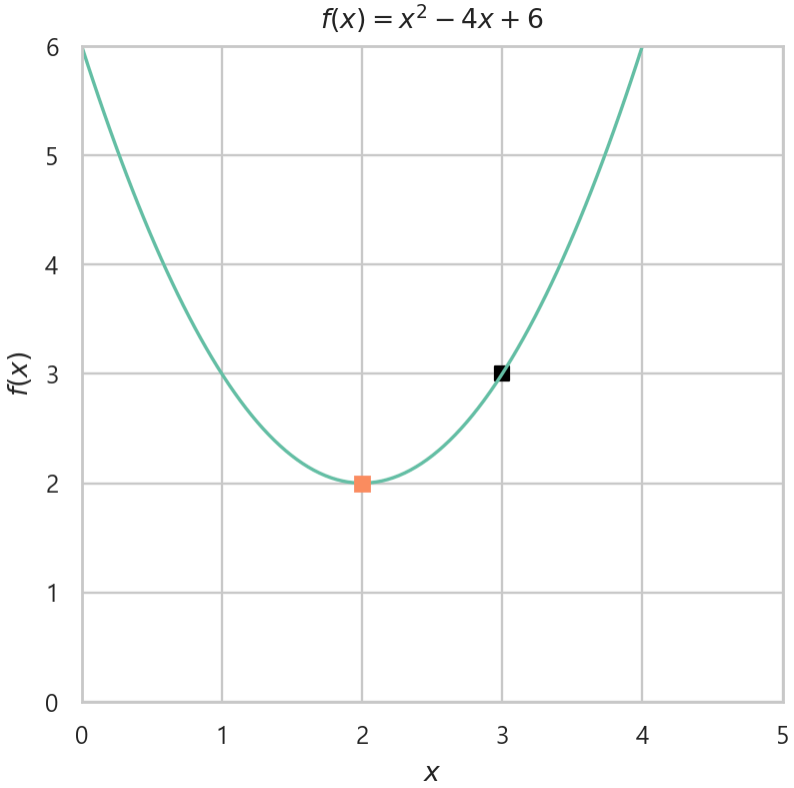
$$f(x) = 3x^3 - 2x + 1$$

In [3]:

```
x = np.linspace(0, 4, 100)
f = lambda x: x**2 - 4*x + 6
```

In [4]:

```
fig, ax = plt.subplots(figsize=(4,4))

ax.plot(x, f(x))
ax.plot(2, 2, marker = "s" )
ax.set(xlim = (0,5), ylim = (0,6))
ax.scatter(3, f(3), marker = "s", color = "k")
ax.set(xlabel = R'$x$', ylabel = R'$f(x)$', title = "$f(x) = x^2 - 4x +
```



- **딥러닝을 이해하려면 다변수 함수($f(x_1, x_2, \dots, x_n)$)의 미분에 대한 이해가 필요하다!**
  → 편미분

편미분

다음 이변수 함수 $f(x, y)$에 대한 $\frac{\partial f}{\partial x}(= f_x)$와 $\frac{\partial f}{\partial y}(= f_y)$를 계산해 보자.

$$f(x, y) = 3xy^2 + 4x^3 - 5y^2 + 1$$

$$\frac{\partial f}{\partial x} = f_x = 3y^2 + 12x^2$$

$$\frac{\partial f}{\partial y} = f_y = 6xy - 10y$$

Chain rule(연쇄법칙), 합성함수의 미분

- t에 대한 함수 $y = f(t)$가 있다고 하자.
- 만약 t가 x에 대한 함수 $t = g(x)$라고 한다면
  $$y = f(t) = f(g(x))$$
  의 형태로 합성해서 쓰일 수 있다.

- 여기서 y는 x에 대한 함수이므로

$$\begin{aligned}\frac{dy}{dx} &= \frac{dy}{dt}\cdot\frac{dt}{dx}\\ &= f'(t)\cdot g'(x)\\ &= f'(g(x))\cdot g'(x).\end{aligned}$$

- 식의 형태를 보면 바깥쪽의 함수 f에 대한 미분을 하고, 안쪽에 있는 함수 g의 미분이 곱해지는 형태이기 때문에 chain rule이라고 부른다.

함수 $f(x) = (2x-1)^3$에 대한 $\frac{df}{dx}$를 계산해 보자.

이변수 함수 $f(x,y)$에 대한 $\frac{\partial}{\partial x}f(x,y)$와 $\frac{\partial}{\partial y}f(x,y)$를 계산해 보자.

$$f(x,y) = 3(x-2)^2 + (y-2)^2$$

2. 신경망의 학습

- **최적화 문제**: 주어진 상황에서 어떤 수치가 가장 알맞은 것인지 찾는 것
- 신경망 모델은 연산을 통해 출력 값을 내고, 문제(분류인지 회귀인지)에 따라 손실함수($L(\mathbf{w},\mathbf{b})$)를 통해 정답과 출력 사이의 거리를 계산한다.
- 정답과 출력 사이를 가깝게 하도록 가중치와 바이어스를 갱신하여 최적의 해를 찾는 것이 신경망의 학습 목표이다.
- 정답과 출력 사이를 가깝게 한다는 것은 손실함수의 최솟값을 찾는다는 의미이고 같은 말로는 손실함수의 최적화 문제를 푸는 것이라고 할 수 있다.

$$\min_{\mathbf{x},\mathbf{b}} L(\mathbf{w},\mathbf{b})$$

다음과 같은 이차 함수 문제가 있다고 가정해 보자.

$$\min_x f(x) = x^2 - 4x + 6$$

이 문제를 푸는 방법은 4가지가 있다!

1. 완전제곱식으로 만드는 방법

$$\begin{aligned}f(x) &= x^2 - 4x + 6\\ &= (x-2)^2 + 2\end{aligned}$$

2. 미분을 이용하는 방법

$$f'(x) = 2x - 4$$

3. 모든 x값을 입력하여 최솟값을 찾는 방법 → 현실적으로 불가능
4. 수치 최적화 알고리즘 사용

딥러닝 모델의 학습(손실함수의 최솟값을 계산하는 것)은 수치 최적화 알고리즘 방법으로 해결(1, 2, 3번으로는 풀 수가 없기 때문에)

→ 수치 최적화 방법은 반복법(정해진 방법을 되풀이 하여 값을 업데이트하는 방식)을 사용하여 문제를 푼다.

→ 즉, 초기값이 있고, 어느방향으로 업데이트할 것인지 방향을 찾고, 학습률 만큼 업데이트한다.

- 딥러닝에 사용되는 수치 최적화 알고리즘(=손실함수의 최솟값을 찾아가는 방법들, 옵티마이저)
  - Stochastic Gradient Descent
  - Momentum
  - Nesterov
  - Adagrad
  - RMSProp
  - Adam

알고리즘	연도	학습률	탐색방향	탐색방향/학습률 기반 알고리즘
Stochastic Gradient Descent	1945	상수	그래디언트	탐색방향
Momentum/Nesterov	1964/1983	상수	단기 누적 그래디언트	탐색방향
Adagrad	2011	장기 파라미터 변화량과 반비례	그래디언트	학습률
RMSProp	2012	단기 파라미터 변화량과 반비례	그래디언트	학습률
Adam	2014	단기 파라미터 변화량과 반비례	단기 누적 그래디언트	학습률

2.1 Gradient Descent를 통해 최적화 문제를 이해해 보자

- **그래디언트**: 어떤 점에서 함수가 가장 가파르게 상승하는 방향
- **손실함수의 그래디언트 방향**: 손실함수가 가장 가파르게 증가하는 방향
- 우리는 손실함수가 가장 가파르게 감소하길 바람 → 손실함수의 음의 그래디언트로 이동하면 최솟값으로 가게 됨

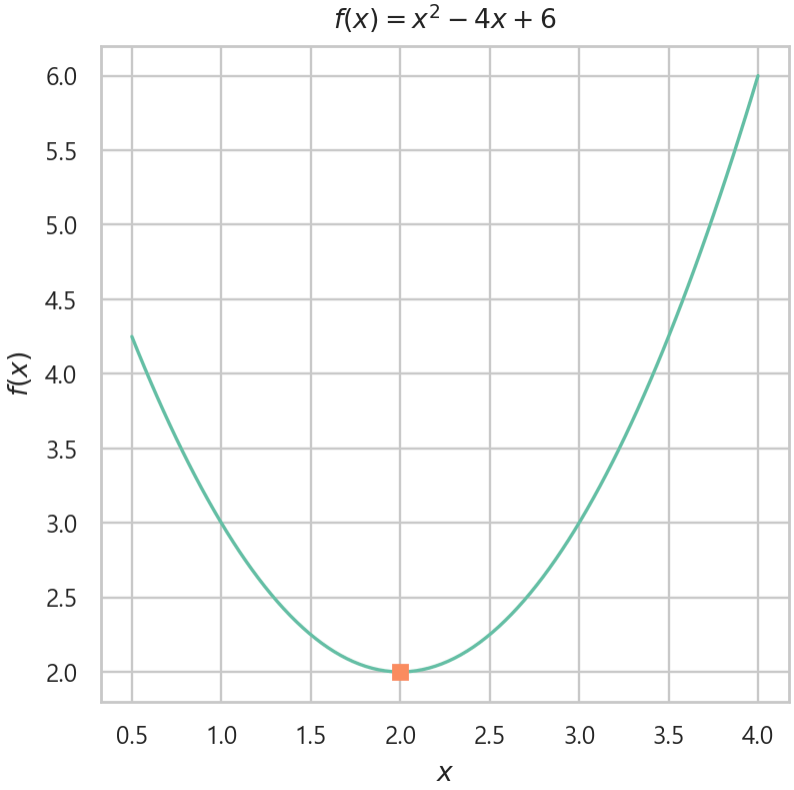
$$w^{(k+1)} \leftarrow w^{(k)} - \epsilon \nabla f(w^{(k)})$$

손실함수가 $f(x) = x^2 - 4x + 6$라고 가정하고 Gradient Descent로 최솟값을 찾아보자

- 아래로 볼록하기 때문에 그래디언트($f'(x) = 2x - 4$)의 반대방향으로 충분히 작은 학습률을 주면 최솟값으로 수렴한다

In [5]:

```
x = np.linspace(0.5, 4, 100)
f = lambda x: x**2 - 4*x + 6
grad_f = lambda x: 2*x - 4

fig, ax = plt.subplots(figsize=(4,4))

ax.plot(x, f(x))
ax.plot(2, 2, marker = "s" )
ax.set(xlabel = R'$x$', ylabel = R'$f(x)$', title = "$f(x) = x^2 - 4x +
```



In [6]:

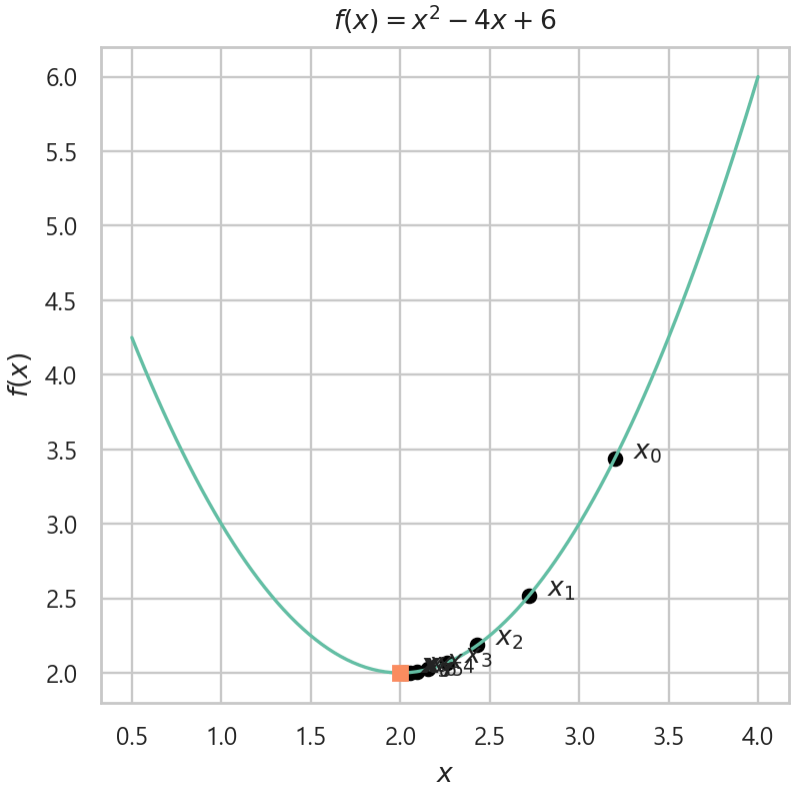
```
def gradient_descent(f, grad_f, x_0, learning_rate, max_iter):
    paths = []
    for i in range(max_iter):
        x_1 = x_0 - learning_rate * grad_f(x_0)
        paths.append(x_1)
        x_0 = x_1

    return np.array(paths)
```

In [7]:

```
##
paths = gradient_descent(f, grad_f, 4.0, learning_rate=0.2, max_iter=10)
```

In [8]:

```
##
fig, ax = plt.subplots(figsize=(4,4))

ax.plot(x, f(x))
ax.plot(2, 2, marker = "s" )
ax.set(xlabel = R'$x$', ylabel = R'$f(x)$', title = "$f(x) = x^2 - 4x +
ax.scatter(paths, f(paths), color = "k")

for k, point in enumerate(paths):
    ax.text(point+0.1, f(point), f'$x_{k}$')
```

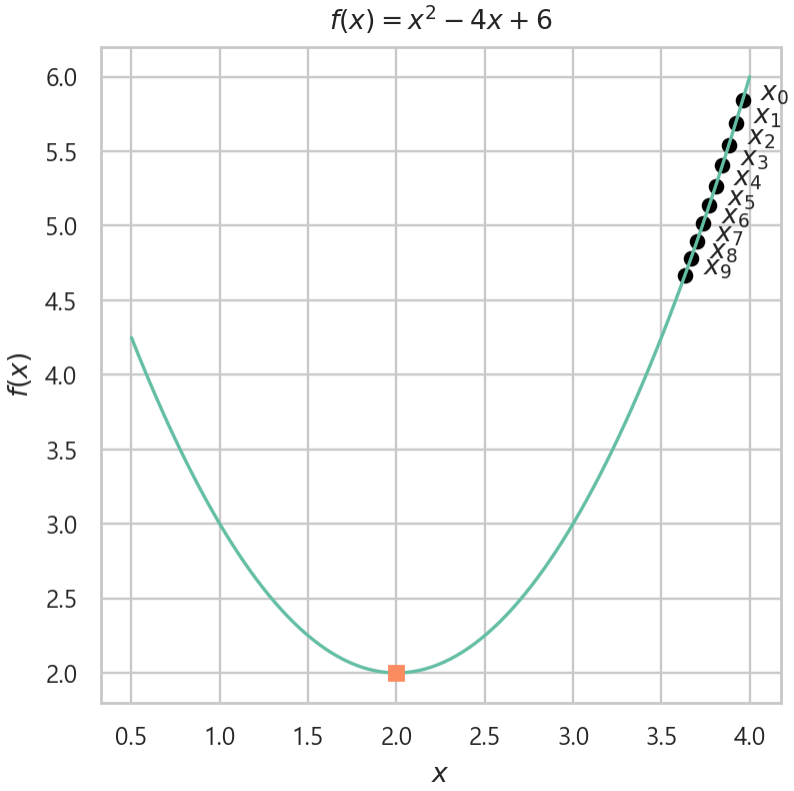


In [9]:

```
## learning_rate를 변경해 보자

paths = gradient_descent(f, grad_f, 4, learning_rate=0.01, max_iter=10)
```

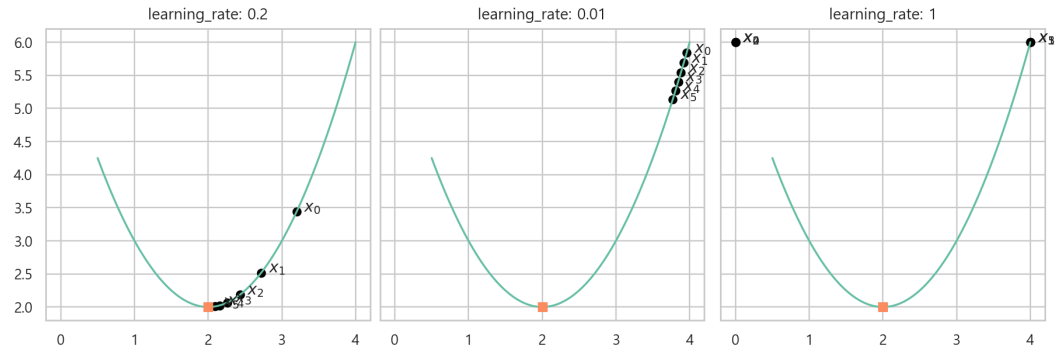
In [10]:

```
##
fig, ax = plt.subplots(figsize=(4,4))

ax.plot(x, f(x))
ax.plot(2, 2, marker = "s" )
ax.set(xlabel = R'$x$', ylabel = R'$f(x)$', title = "$f(x) = x^2 - 4x +
ax.scatter(paths, f(paths), color = "k")

for k, point in enumerate(paths):
    ax.text(point+0.1, f(point), f'$x_{k}$')
```



In [11]:

```
fig, axes = plt.subplots(1, 3, figsize=(9, 3), sharex = True, sharey = T

for lr, ax in zip([0.2, 0.01, 1], axes.flat):
    paths = gradient_descent(f, grad_f, 4, learning_rate=lr, max_iter=6)

    ax.plot(x, f(x))
    ax.plot(2, 2, marker = "s" )
    ax.set(title = f"learning_rate: {lr}")
    ax.scatter(paths, f(paths), color = "k")

    for k, point in enumerate(paths):
        ax.text(point+0.1, f(point), f'$x_{k}$')
```



그래디언트 디센트의 한계점

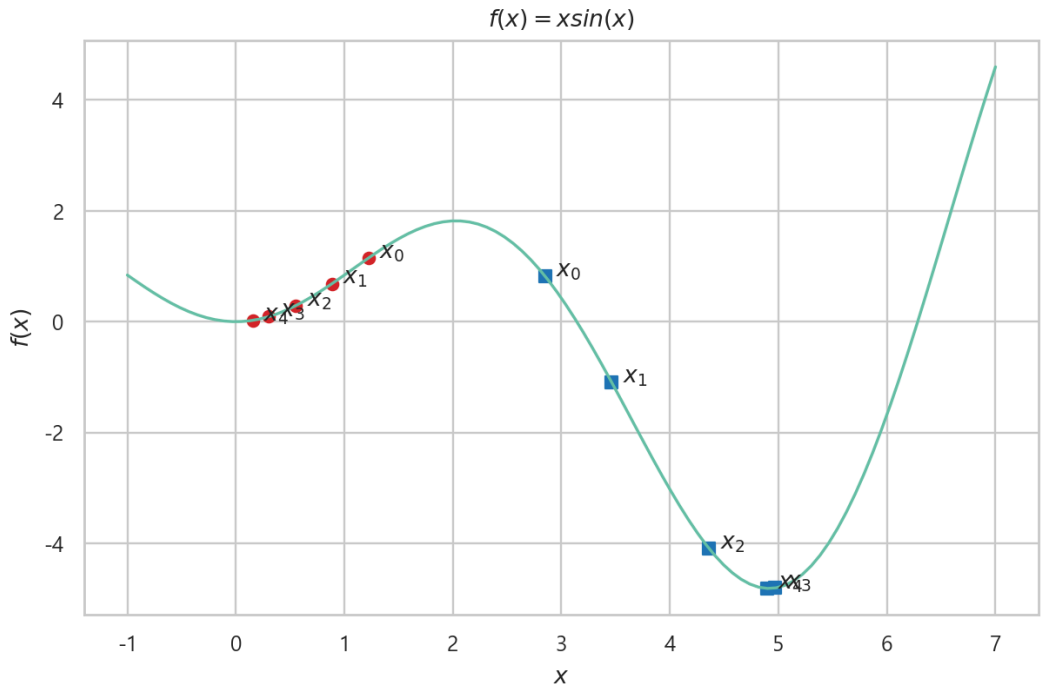
- 초깃값 민감성: 어디에서 시작했느냐에 따라 전역 최솟값으로 갈 수도 있고, 국소 최솟값으로 갈 수도 있다.
- 학습률 민감성: 학습률이 크면 발산해 버릴 수 있으므로 보통 작게 잡지만 최솟값까지 가는데 오래 걸릴 수 있다.

In [12]:

```
x = np.linspace(-1, 7, 100)
f = lambda x: x*np.sin(x)
grad_f = lambda x: np.sin(x) + x * np.cos(x)

## 초깃값의 민감성
paths_s = gradient_descent(f, grad_f, 2.5, learning_rate=0.25, max_iter=
paths_o = gradient_descent(f, grad_f, 1.5, learning_rate=0.25, max_iter=
```

In [13]:

```
fig, ax = plt.subplots(figsize=(6,4))

ax.plot(x, f(x))
ax.scatter(paths_s, f(paths_s), marker = "s", color = "tab:blue")
ax.scatter(paths_o, f(paths_o), marker = "o", color = "tab:red")

for k, point in enumerate(paths_s):
        ax.text(point+0.1, f(point), f'$x_{k}$')

for k, point in enumerate(paths_o):
        ax.text(point+0.1, f(point), f'$x_{k}$')

ax.set(xlabel = R'$x$', ylabel = R'$f(x)$', title = "$f(x) = xsin(x)$");
```

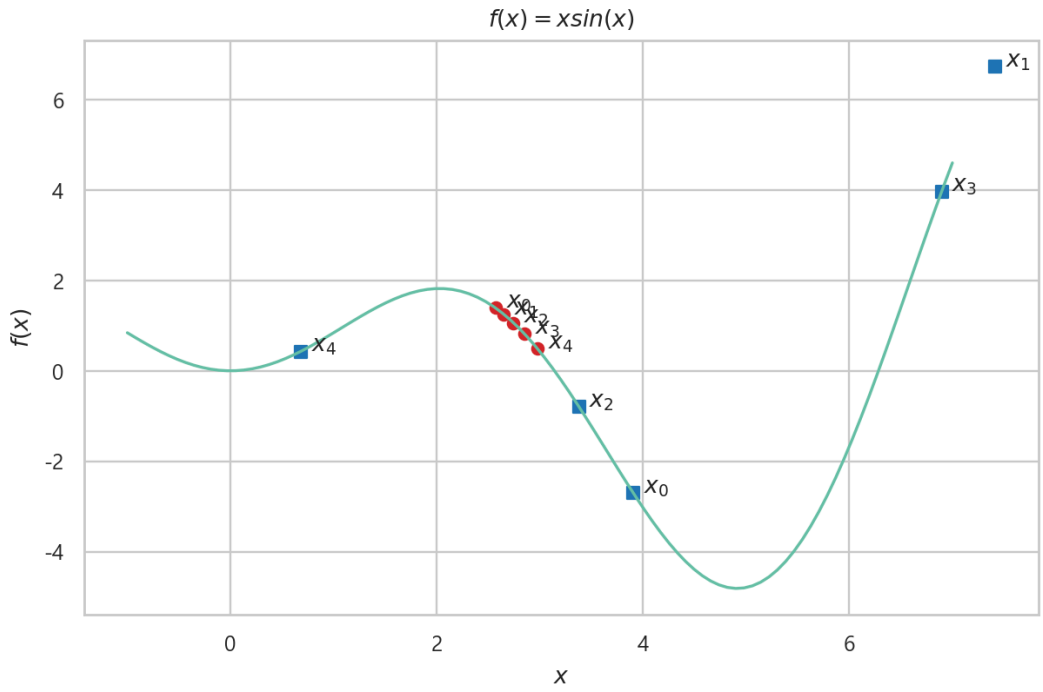


In [14]:

```
## 학습률의 민감성
paths_s = gradient_descent(f, grad_f, 2.5, learning_rate=1.0, max_iter=5
paths_o = gradient_descent(f, grad_f, 2.5, learning_rate=0.05, max_iter=
```

In [15]:

```
fig, ax = plt.subplots(figsize=(6,4))

ax.plot(x, f(x))
ax.scatter(paths_s, f(paths_s), marker = "s", color = "tab:blue")
ax.scatter(paths_o, f(paths_o), marker = "o", color = "tab:red")

for k, point in enumerate(paths_s):
        ax.text(point+0.1, f(point), f'$x_{k}$')

for k, point in enumerate(paths_o):
        ax.text(point+0.1, f(point), f'$x_{k}$')

ax.set(xlabel = R'$x$', ylabel = R'$f(x)$', title = "$f(x) = xsin(x)$");
```



2.2 수치 최적화 알고리즘의 한계점

- 손실함수가 convex function이라면 국소 최솟값(local minimum)은 전역 최솟값(global minimun)이다.
- 그러나 실무에서 손실함수가 convex function은 거의 없다.(심지어 시각화도 못한다.)
- 따라서 모든 수치 최적 알고리즘은 국소 최솟값은 찾아주지만 항상 전역 최솟값을 찾을 수는 없다.
- 최근에 고안된 수치 최적화 알고리즘은 한계점을 완화시킬 수는 있지만 그렇다고 항상 전역 최솟값을 찾아 주는 것은 아니다.ㅜㅜ

In [16]:

```
x = np.outer(np.linspace(-5, 5, 5), np.ones((10,)))
x
```

Out[16]:

```
array([[-5. , -5. , -5. , -5. , -5. , -5. , -5. , -5. , -5. , -5. ],
       [-2.5, -2.5, -2.5, -2.5, -2.5, -2.5, -2.5, -2.5, -2.5, -2.5],
       [ 0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ,  0. ],
       [ 2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5],
       [ 5. ,  5. ,  5. ,  5. ,  5. ,  5. ,  5. ,  5. ,  5. ,  5. ]])
```

In [17]:

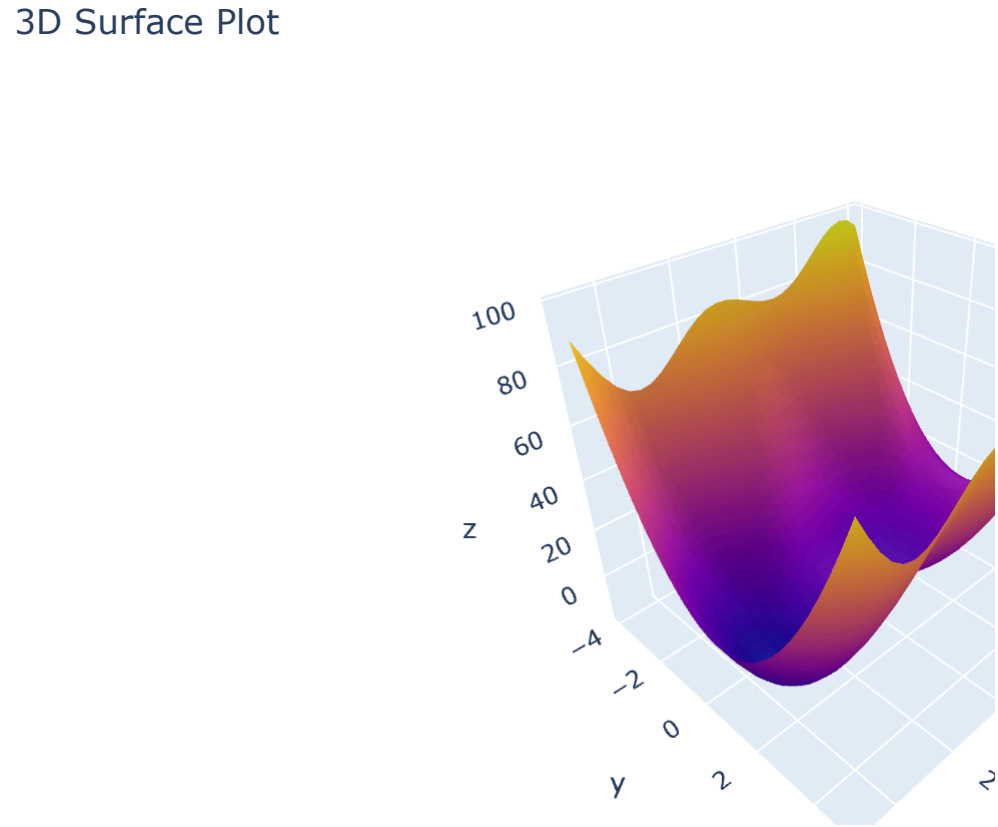
```
y =  x.copy().T
y
```

Out[17]:

```
array([[-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ],
       [-5. , -2.5,  0. ,  2.5,  5. ]])
```

In [18]:

```
x = np.outer(np.linspace(-4, 5, 30), np.ones(30))
y = x.copy().T # transpose
z = 5*(x*np.cos(x)) + (2*y-1)**2

surface = go.Surface(x = x, y = y, z = z)
data = [surface]

layout = go.Layout(title = '3D Surface Plot')

fig = go.Figure(data = data, layout=layout)
fig.show()
```



3. 탐색방향/학습률 기반 알고리즘

알고리즘	연도	학습률	탐색방향	탐색방향/학습률 기반 알고리즘
Stochastic Gradient Descent	1945	상수	그래디언트	탐색방향
Momentum/Nesterov	1964/1983	상수	단기 누적 그래디언트	탐색방향
Adagrad	2011	장기 파라미터 변화량과 반비례	그래디언트	학습률

알고리즘	연도	학습률	탐색방향	탐색방향/학습률 기반 알고리즘
RMSProp	2012	단기 파라미터 변화량과 반비례	그래디언트	학습률

2.1 탐색방향 기반 알고리즘

Stochastic Gradient Descent

- Gradient Descent의 방법으로 계산하면 데이터가 많을 때 계산량이 너무 많다.
- 이 점을 보완하기 위해 Stochastic Gradient Descent는 랜덤하게 데이터를 추출(mini batch)하여 그래디언트를 계산한다.

1. 초기값 $w^{(0)}$ 설정

2. 데이터 셔플

3. mini batch만큼 데이터 부분집합들을 추출

4. 각 부분집합마다 다음을 반복

   a. 부분집합의 그래디언트 계산: $\nabla f(w^{(k)})$

$$\nabla f(w^{(k)}) = \begin{bmatrix} \frac{\partial f}{\partial w_{1,1}} & \cdots & \frac{\partial f}{\partial w_{1,N}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial w_{M,1}} & \cdots & \frac{\partial f}{\partial w_{M,N}} \end{bmatrix}$$

   b. 그래디언트 반대 방향으로 학습률 만큼 탐색 방향 설정: $-\epsilon \nabla f(w^{(k)})$
   c. 파라미터 업데이트: $w^{(k+1)} \leftarrow w^{(k)} - \epsilon \nabla f(w^{(k)})$

5. 만들어진 부분집합을 모두 사용한 후 4.를 epoch만큼 반복

Momentum

- 그래디언트 디센트 방법은 국소 최솟값에 빠지게 되면 빠져나갈 방법이 없다.

$$w^{(k+1)} \leftarrow w^{(k)} - \epsilon \nabla f(w^{(k)})$$

- Momentum는 **전 단계의 탐색 방향의 누적합**에 현재 단계의 그래디언트 디센트를 더해서 탐색 방향을 찾기 때문에 국소 최솟값을 벗어나 전역 최솟값으로 갈 수 있게 해준다.

$$v^{(k+1)} \leftarrow \alpha v^{(k)} - \epsilon \nabla f(w^{(k)})$$
$$w^{(k+1)} \leftarrow w^{(k)} + v^{(k+1)}$$

1. 초기값 $w^{(0)}$과 단기 누적속도 합 $v^{(0)} = 0$ 설정

2. 데이터 셔플

3. mini batch만큼 데이터 부분집합들을 추출

4. 각 부분집합마다 다음을 반복

   a. 부분집합의 그래디언트 계산: $\nabla f(w^{(k)})$

   b. 탐색 방향 설정: $v^{(k+1)} \leftarrow \alpha v^{(k)} - \epsilon \nabla f(w^{(k)})$

c. 파라미터 업데이트: $w^{(k+1)} \leftarrow w^{(k)} + v^{(k+1)}$

5. 만들어진 부분집합을 모두 사용한 후 4.를 epoch만큼 반복

Nesterov

- Momentum과 그래디언트를 계산하는 부분만 차이가 있음
- Nesterov가 Momentum보다 그래디언트 계산을 좀 더 정확하게 해주기 때문에 수렴 속도가 더 빠르다.

$$v^{(k+1)} \leftarrow \alpha v^{(k)} - \epsilon \nabla f(w^{(k)} + \alpha v^{(k)})$$
$$w^{(k+1)} \leftarrow w^{(k)} + v^{(k+1)}$$

1. 초기값 $w^{(0)}$과 단기 누적속도 합 $v^{(0)} = 0$ 설정

2. 데이터 셔플

3. mini batch만큼 데이터 부분집합들을 추출

4. 각 부분집합마다 다음을 반복

a. 부분집합의 그래디언트 계산: $\nabla f(w^{(k)} + \alpha v^{(k)})$

b. 탐색 방향 설정: $v^{(k+1)} \leftarrow \alpha v^{(k)} - \epsilon \nabla f(w^{(k)})$

c. 파라미터 업데이트: $w^{(k+1)} \leftarrow w^{(k)} + v^{(k+1)}$

5. 만들어진 부분집합을 모두 사용한 후 4.를 epoch만큼 반복

2.2 학습률 기반 알고리즘

- 학습률을 고정하지 않고 매번 적절한 학습률을 계산
- 학습률을 고정하는 방법은 학습률에 따라 최솟값으로 수렴이 너무 느리거나 발산할 수는 문제 발생
  - 즉, SDG, Momentum, Nesterov는 이 문제를 피해갈 수 없음
- 따라서 적응형 학습률을 쓰면 해결 할 수 있다!
  - 처음에는 큰 학습률로 시작
  - 업데이트된 가중치로 손실함수 값을 계산했을 때 값이 전 단계의 손실함수 값보다 커지면 학습률을 업데이트하지 않음
  - 업데이트된 가중치로 손실함수 값을 계산했을 때 값이 전 단계의 손실함수 값보다 작아지면 학습률 크기를 절반으로 줄임
- 학습률 기반 알고리즘은 매 단계 학습률이 업데이트 되기 때문에 학습률의 초깃값을 정할 때 신경을 조금 덜 써도 된다.

Adagard

- 학습률을 갱신하기 때문에 최솟값을 잘 찾아가지만 초기부터 그래디언트 제곱의 누적합을 가지고 가기 때문에 수렴속도가 느릴 수 있다.

1. 초기값 $w^{(0)}$과 장기 누적 그래디언트 크기 $r^{(0)} = 0$설정

2. 데이터 셔플

3. mini batch만큼 데이터 부분집합들을 추출

4. 각 부분집합마다 다음을 반복

   a. 부분집합의 그래디언트 계산: $\nabla f(w^{(k)})$

   b. 장기 누적 그래디언트 계산: $r^{(k+1)} \leftarrow r^{(k)} + \nabla f(w^{(k)}) \odot \nabla f(w^{(k)})$

$$\nabla f(w^{(k)}) \odot \nabla f(w^{(k)}) = \begin{bmatrix} \frac{\partial f}{\partial w_{1,1}} & \cdots & \frac{\partial f}{\partial w_{1,N}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial w_{M,1}} & \cdots & \frac{\partial f}{\partial w_{M,N}} \end{bmatrix} \odot \begin{bmatrix} \frac{\partial f}{\partial w_{1,1}} & \cdots & \frac{\partial f}{\partial w_{1,N}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial w_{M,1}} & \cdots & \frac{\partial f}{\partial w_{M,N}} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial f}{\partial w_{1,1}}\right)^2 & & \\ & & \\ & & \left(\frac{\partial f}{\partial w_{M,N}}\right)^2 \end{bmatrix}$$

   c. 파라미터 업데이트: $w^{(k+1)} \leftarrow w^{(k)} - \frac{\epsilon}{\delta + \sqrt{r^{(k+1)}}} \odot \nabla f(w^{(k)})$

5. 만들어진 부분집합을 모두 사용한 후 4.를 epoch만큼 반복

RMSProop(Root Mean Square Propagation)

- 너무 오래된 그래디언트는 반영을 적게 하겠다!

1. 초기값 $w^{(0)}$과 단기 누적 그래디언트 크기 $r^{(0)} = 0$설정

2. 데이터 셔플

3. mini batch만큼 데이터 부분집합들을 추출

4. 각 부분집합마다 다음을 반복

   a. 부분집합의 그래디언트 계산: $\nabla f(w^{(k)})$

   b. 단기 누적 그래디언트 계산: $r^{(k+1)} \leftarrow \rho r^{(k)} + (1-\rho)\nabla f(w^{(k)}) \odot \nabla f(w^{(k)})$

   c. 파라미터 업데이트: $w^{(k+1)} \leftarrow w^{(k)} - \frac{\epsilon}{\delta + \sqrt{r^{(k+1)}}} \odot \nabla f(w^{(k)})$

5. 만들어진 부분집합을 모두 사용한 후 4.를 epoch만큼 반복

Adam(Adaptive moments)

- Momentum과 RMSProp의 조합
- 즉, 탐색 방향도 보고, 학습률도 갱신하는 알고리즘

1. 초기값 $w^{(0)}$, 단기 누적 그래디언트 크기 $r^{(0)} = 0$, 단기 누적 그랜디언트 합 $s^{(0)} = 0$ 설정

2. 데이터 셔플

3. mini batch만큼 데이터 부분집합들을 추출

4. 각 부분집합마다 다음을 반복

   a. 부분집합의 그래디언트 계산: $\nabla f(w^{(k)})$

   b. 탐색 방향을 단기 누적 그래디언트 합으로 설정

$$s^{(k+1)} \leftarrow \rho_1 s^{(k)} + (1-\rho_1)\nabla f(w^{(k)})$$
$$\hat{s}^{(k+1)} \leftarrow \frac{s^{(k+1)}}{1-\rho_1^{k+1}}$$

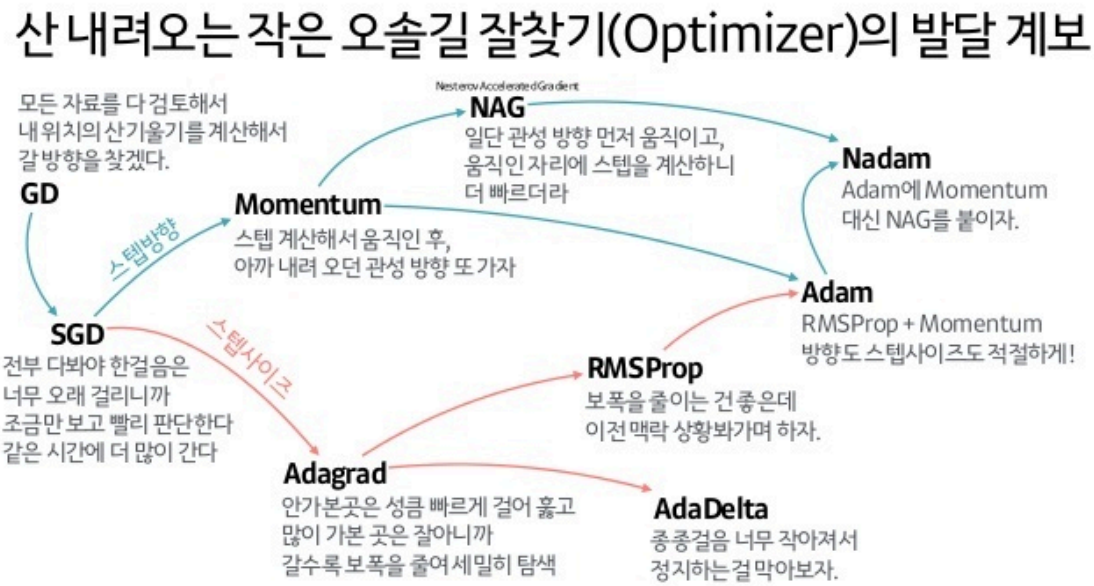
c. 학습률을 단기 누적 그래디언트 크기로 설정

$$r^{(k+1)} \leftarrow \rho_2 r^{(k)} + (1-\rho_2)\nabla f(w^{(k)}) \odot \nabla f(w^{(k)})$$
$$\hat{r}^{(k+1)} \leftarrow \frac{r^{(k+1)}}{1-\rho_2^{k+1}}$$

d. 파라미터 업데이트

$$w^{(k+1)} \leftarrow w^{(k)} - \frac{\epsilon}{\delta + \sqrt{\hat{r}^{(k+1)}}}\hat{s}^{(k+1)}$$

2.3 수치 최적화 알고리즘(Optimizer) 발달 계보



3. 어떤 Optimizer가 제일 좋은가?

Joshua Bengio(딥러닝 창립자 중 한 사람), 2016

- 어떤 알고리즘도 딥러닝 모델의 최적화 문제를 완벽히 풀 수 없음
- 선호도를 바탕으로 실험을 하면 노하우가 생기게 된다.
- 개인의 딥러닝 모델 구성 능력은 이론과 많은 실험의 결합으로 향상시켜야 한다!

In [ ]:

```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```

In [ ]:
```
1
```