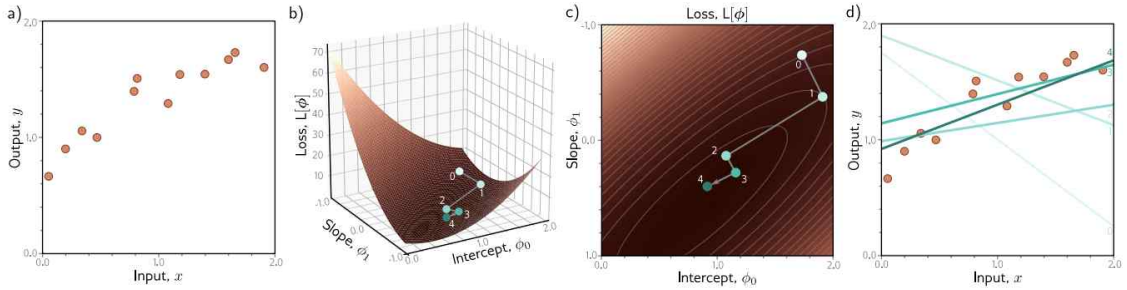
Notebook 6.2 Gradient descent

Figure 6.1

* 경사하강 알고리즘 구현



1. 데이터 및 모델 정의

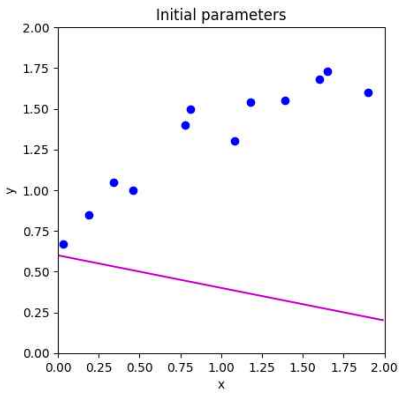
```
# training data 12 pairs {x_i, y_i}

data = np.array([[0.03,0.19,0.34,0.46,0.78,0.81,1.08,1.18,1.39,1.60,1.65,1.90],
                 [0.67,0.85,1.05,1.00,1.40,1.50,1.30,1.54,1.55,1.68,1.73,1.60]])
```

```
def model(phi,x):
  y_pred = phi[0] + phi[1] * x
  return y_pred
```

$$\hat{y} = \phi_0 + \phi_1 x$$

2 파라미터 초기화 및 초기 모델 시각화

```
phi = np.zeros((2,1))
phi[0] = 0.6
phi[1] = -0.2
draw_model(data,model,phi, "Initial parameters")
```



3 손실함수(오차제곱합) 계산

```
# 오차제곱합
def compute_loss(data_x, data_y, model, phi):
pred_y = model(phi, data_x)
loss = np.sum((pred_y-data_y)**2)
  return loss
```

4. 경사도 계산

```
def compute_gradient(data_x, data_y, phi):
    # Number of data points
    num_data = len(data_x)
    pred_y = model(phi, data_x)
    # gradients 계산
    dl_dphi0 = -2* np.sum(data_y - pred_y) # 절편
    dl_dphi1 = -2* np.sum(data_x * (data_y - pred_y)) # 기울기
    # gradient를 열 벡터로 반환
    return np.array([[dl_dphi0], [dl_dphi1]])
```

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \end{bmatrix}$$

```
def loss_function_1D(dist_prop, data, model, phi_start, search_direction):
  # 주어진 거리만큼 이동한 후의 손실 반환
  return compute_loss(data[0,:], data[1,:], model, phi_start+ search_direction * dist_prop)

def line_search(data, model, phi, gradient, thresh=.00001, max_dist 0.1, max_iter15):
    # 검색할 범위 내에서 네 개의 초기 점 설정
a = 0
```

```
b = 0.33* max_dist
c = 0.66* max_dist
d = 1.0* max_dist
n_iter = 0
    while np.abs(b-c) > thresh and n_iter < max_iter:
      n_iter = n_iter+1
      lossa = loss_function_1D(a, data, model, phi,gradient)
      lossb = loss_function_1D(b, data, model, phi,gradient)
      lossc = loss_function_1D(c, data, model, phi,gradient)
      lossd = loss_function_1D(d, data, model, phi,gradient)

       # A가 B, C, D보다 작을 경우
       if np.argmin((lossa,lossb,lossc,lossd))==0:
         b = b/2
         c = c/2
         d = d/2
         continue;
       # B가 C보다 작을 경우
       if lossb < lossc:
          d = c
          b = a + (d-a)/3
          c = a+ 2*(d-a)/3
         continue
       # C가 B보다 작을 경우
        a = b
        b = a+ (d-a)/3
        2*(d-a)/3
    # 중간 두 점의 평균을 반환
    return (b+c)/2.0
```

6. 경사 하강법 단계수행

Step 1. Compute the derivatives of the loss with respect to the parameters:

$$\frac{\partial L}{\partial \boldsymbol{\phi}} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}. \tag{6.2}$$

Step 2. Update the parameters according to the rule:

$$\boldsymbol{\phi} \longleftarrow \boldsymbol{\phi} - \alpha \cdot \frac{\partial L}{\partial \boldsymbol{\phi}}, \tag{6.3}$$

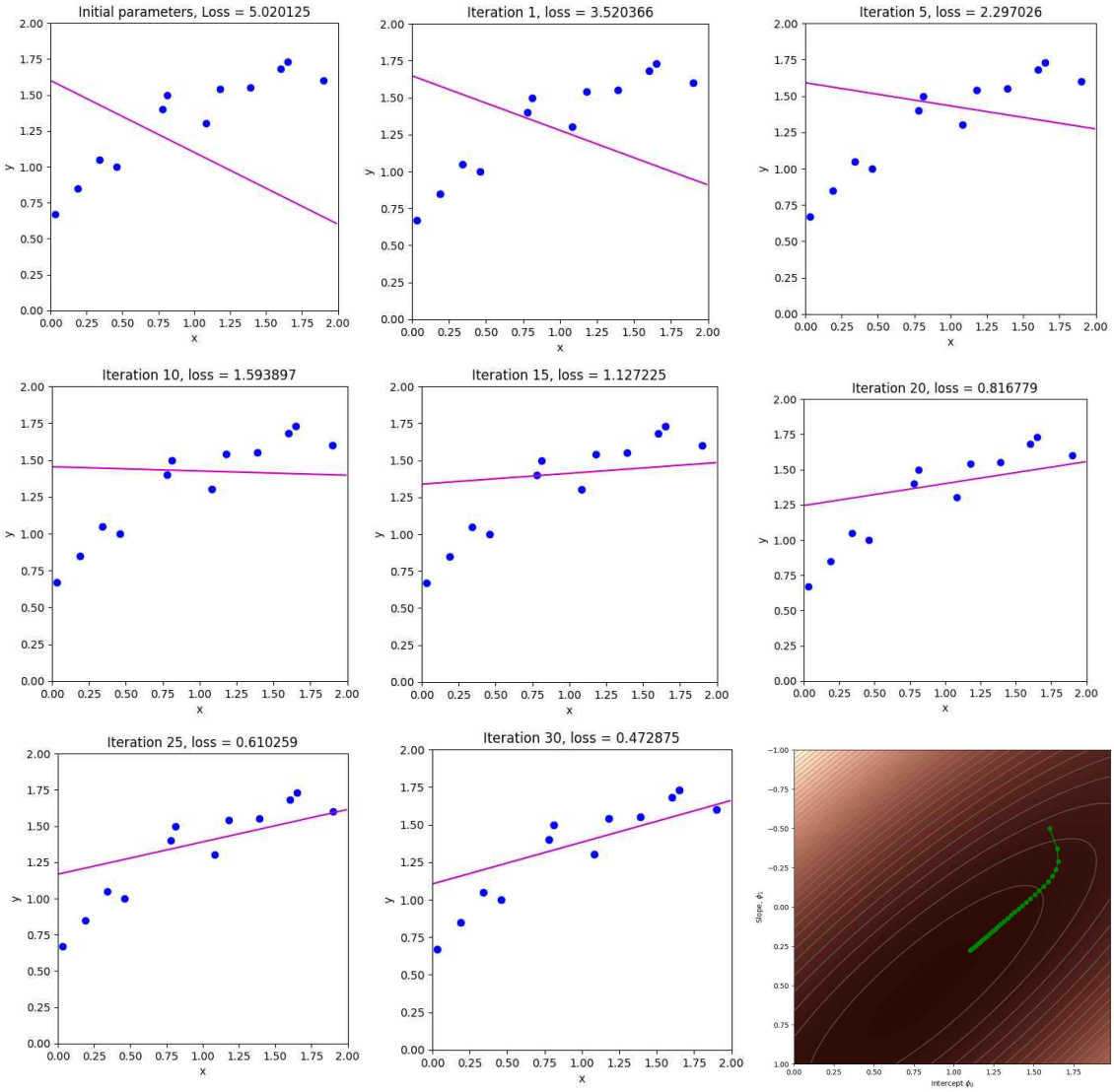
where the positive scalar α determines the magnitude of the change.

```
def gradient_descent_step(phi, data, model, alpha=0.01):
    # gradient 계산
gradient = compute_gradient(data[0, :], data[1, :], phi)
    # parameters 업데이트
phi = phi - alpha * gradient
    return phi
```

7. 반복적인 경사 하강법 수행 및 시각화

```
# 파라미터를 초기화 및 모델 시각화
n_steps = 30
phi_all = np.zeros((2,n_steps+1))
phi_all[0,0] = 1.6
phi_all[1,0] = -0.5
# 손실 측정 및 초기 모델 시각화
loss =  compute_loss(data[0,:], data[1,:], model, phi_all[:,0:1])

draw_model(data,model,phi_all[:,0:1], "Initial parameters, Loss = %f"%(loss))
```

```
# 반복적 경사 하강 단계 수행
for c_step in range (n_steps):
phi_all[:,c_step+1:c_step+2] = gradient_descent_step(phi_all[:,c_step:c_step+1],data, model)
loss = compute_loss(data[0,:], data[1,:], model, phi_all[:,c_step+1:c_step+2])
draw_model(data,model,phi_all[:,c_step+1], "Iteration %d, loss = %f"%(c_step+1,loss))
draw_loss_function(compute_loss, data, model,phi_all)
```



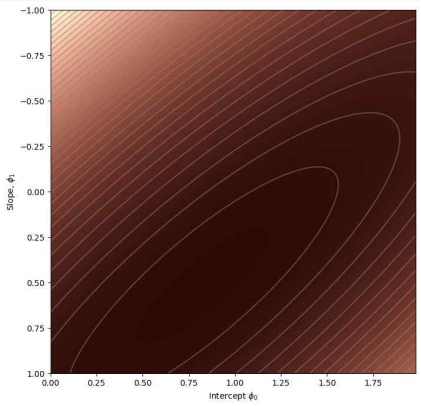
[참고. 시각화 코드 구현]

*** 모델 시각화**

```
def draw_model(data,model,phi,title=None):
x_model = np.arange(0,2,0.01)
y_model = model(phi,x_model)
fix, ax = plt.subplots()
  # bo : 파란색 원
ax.plot(data[0,:],data[1,:],'bo')
  # m- : 마젠타 색의 직선
ax.plot(x_model,y_model,'m-')
ax.set_xlim([0,2]);ax.set_ylim([0,2])
ax.set_xlabel('x'); ax.set_ylabel('y')
  # x축과 y축의 비율을 동일하게 설정
ax.set_aspect('equal')
  if title is not None:
ax.set_title(title)
plt.show()
```

```
def draw_loss_function(compute_loss, data,  model, phi_itersNone):
  # 컬러맵 정의
   my_colormap_vals_hex =('2a0902', '2b0a03', '2c0b04', '2d0c05', '2e0c06', '2f0d07', '300d08',
                                                    ⋮
                                'fff8d9', 'fffada', 'fffbdc', 'fffcdd', 'fffedf', 'ffffe0')
  # # 16진수 색상 값을 10진수로 변환
   my_colormap_vals_dec = np.array([int(element,base=16) forelement inmy_colormap_vals_hex])
  # R, G, B값 추출
   r = np.floor(my_colormap_vals_dec/(256*256))
   g = np.floor((my_colormap_vals_dec - r *256*256)/256)
   b = np.floor(my_colormap_vals_dec - r * 256*256- g * 256)
  # 컬러맵 생성
  # np.vstack : 수직 결합
   my_colormap = ListedColormap(np.vstack((r,g,b)).transpose()/255.0)
  # 절편/기울기 값의 그리드 생성
   intercepts_mesh, slopes_mesh = np.meshgrid(np.arange(0.0,2.0,0.02), np.arange(-1.0,1.0,0.002))
   loss_mesh = np.zeros_like(slopes_mesh)
  # 각 parameters set에 대한 loss 계산
  # np.ndenumerate : 반복 가능한 객체에 대해 인덱스 정보를 가져오고 싶은 경우
  for idslope, slope in np.ndenumerate(slopes_mesh):
     loss_mesh[idslope] = compute_loss(data[0,:], data[1,:], model, np.array([[intercepts_mesh[idslope]],
                                [slope]]))
     fig,ax = plt.subplots()
     fig.set_size_inches(8,8)
  # contourf는 채워진 등고선, contour는 선으로만 된 등고선
   ax.contourf(intercepts_mesh,slopes_mesh,loss_mesh,256,cmap=my_colormap)
   ax.contour(intercepts_mesh,slopes_mesh,loss_mesh,40,colors=['#80808080'])
  if phi_iters is not None:
    ax.plot(phi_iters[0,:], phi_iters[1,:],'go-')
    ax.set_ylim([1,-1])
    ax.set_xlabel('Intercept $\phi_{0}$'); ax.set_ylabel('Slope, $\phi_{1}$')
   plt.show()
```

```
draw_loss_function(compute_loss, data, model)
```



*** 오차제곱합 구현 테스트**

```
loss = compute_loss(data[0,:],data[1,:],model,np.array([[0.6],[-0.2]]))
print('Your loss = %3.3f, Correct loss = %3.3f'%(loss, 12.367))
```

```
Your loss = 12.367, Correct loss = 12.367
```

*** 경사도 계산 테스트**

finite differences : 올바른 계산 확인 방법

함수 값을 평가 후 parameters 중 하나를 아주 작은 양만큼 변경하고 그 양으로 정규화할 경우 gradient의 근사값을 얻을 수 있음

$$\frac{\partial L}{\partial \phi_0} \approx \frac{L[\phi_0+\delta,\phi_1]-L[\phi_0,\phi_1]}{\delta}$$
$$\frac{\partial L}{\partial \phi_1} \approx \frac{L[\phi_0,\phi_1+\delta]-L[\phi_0,\phi_1]}{\delta}$$

parameters가 많을 경우 사용 불가능(gradients를 직접 계산하는 것이 효율적)

```
# gradient 계산
gradient = compute_gradient(data[0,:],data[1,:], phi)
print("Your gradients: (%3.3f,%3.3f)"%(gradient[0],gradient[1]))
delta = 0.0001  # finite differences을 활용
# phi0에 대한 finite differences 근사 gradient 계산
dl_dphi0_est = (compute_loss(data[0,:],data[1,:],model,phi+np.array([[delta],[0]])) - \
                compute_loss(data[0,:],data[1,:],model,phi))/delta
# phi1에 대한 finite differences 근사 gradient 계산
dl_dphi1_est = (compute_loss(data[0,:],data[1,:],model,phi+np.array([[0],[delta]])) - \
                compute_loss(data[0,:],data[1,:],model,phi))/delta
print("Approx gradients: (%3.3f,%3.3f)"%(dl_dphi0_est,dl_dphi1_est))
```