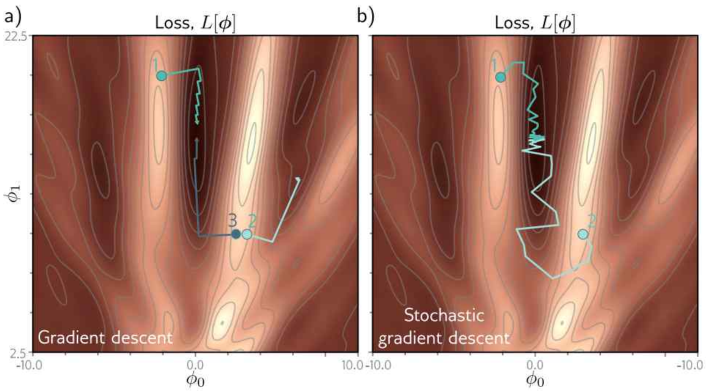
Notebook 6.3 Stochastic_Gradient_Descent

Figure 6.5

* 경사하강법 및 확률적 경사하강법 구현

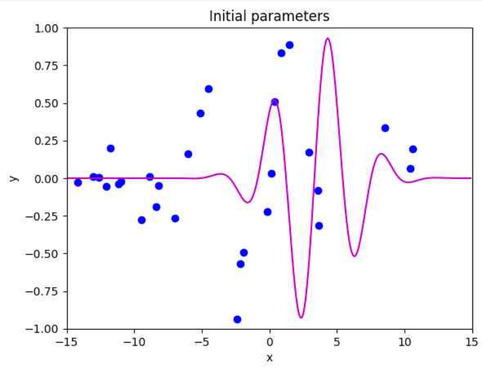


1. 데이터 및 Gabor model 정의

```
# 30쌍의 {x_i, y_i} 훈련 데이터 Gabor model에 적합시키기
data = np.array([[-1.920e+00,-1.422e+01,1.490e+00,-1.940e+00,-2.389e+00,-5.090e+00,
                                          ⋮
                  -1.119e+01,2.902e+00,-8.220e+00,-1.179e+01,-8.391e+00,-4.505e+00],
                 [-1.051e+00,-2.482e-02,8.896e-01,-4.943e-01,-9.371e-01,4.306e-01,
                                          ⋮
                  -3.666e-02,1.709e-01,-4.805e-02,2.008e-01,-1.904e-01,5.952e-01]])
```

$$\mathrm{f}[x, \boldsymbol{\phi}] = \sin[\phi_0 + 0.06 \cdot \phi_1 x] \cdot \exp\left(-\frac{(\phi_0 + 0.06 \cdot \phi_1 x)^2}{32.0}\right)$$

```
# 모델 정의
def model(phi,x):
  sin_component = np.sin(phi[0] + 0.06* phi[1] * x)
  gauss_component = np.exp(-(phi[0] + 0.06* phi[1] * x) * (phi[0] + 0.06* phi[1] * x) / 32)
  y_pred= sin_component * gauss_component
  return y_pred
```



- **파라미터 초기 설정**

```
phi = np.zeros((2,1)) # 열벡터
phi[0] = -5     # Horizontal offset
phi[1] = 25     # Frequency
draw_model(data,model,phi, "Initial parameters")
```

- **훈련 데이터에 대한 제곱합 손실 계산**

```
def compute_loss(data_x, data_y, model, phi):
   # 모델 예측값 계산
   y_pred = model(phi, data_x)
   # 제곱 오차 계산
   squared_diff = (y_pred - data_y) ** 2
   # 제곱 오차의 합 계산
   loss = np.sum(squared_diff)
```

```
    return loss

# 제곱합 손실 함수의 표현식 및 phi0과 phi1의 미분
def gabor_deriv_phi0(data_x,data_y,phi0, phi1):
 x = 0.06* phi1 * data_x + phi0
 y = data_y
 cos_component = np.cos(x)
 sin_component = np.sin(x)
 gauss_component = np.exp(-0.5* x *x / 16)
 deriv = cos_component * gauss_component - sin_component * gauss_component * x / 16
 deriv = 2* deriv * (sin_component * gauss_component - y)
    returnnp.sum(deriv)

def gabor_deriv_phi1(data_x, data_y,phi0, phi1):
x = 0.06* phi1 * data_x + phi0
y = data_y
cos_component = np.cos(x)
sin_component = np.sin(x)
gauss_component = np.exp(-0.5* x *x / 16)
deriv = 0.06* data_x * cos_component * gauss_component - 0.06* data_x*sin_component *
gauss_component * x / 16
deriv = 2*deriv * (sin_component * gauss_component - y)
    returnnp.sum(deriv)

def compute_gradient(data_x, data_y, phi):
dl_dphi0 = gabor_deriv_phi0(data_x, data_y, phi[0],phi[1])
dl_dphi1 = gabor_deriv_phi1(data_x, data_y, phi[0],phi[1])
    # gradient 반환
    returnnp.array([[dl_dphi0],[dl_dphi1]])
```

- 그래디언트 하강법 수행

```
def loss_function_1D(dist_prop, data, model, phi_start, gradient):
  # 거리 이동 후 손실 반환
  return compute_loss(data[0,:], data[1,:], model, phi_start+ gradient * dist_prop)
def line_search(data, model, phi, gradient, thresh=.00001, max_dist0.1, max_iter15):
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)

        # Rule #1 If point A is less than points B, C, and D then halve points B,C, and D
        if np.argmin((lossa,lossb,lossc,lossd))==0:
b = b/2
c = c/2
```

```
d = d/2
          continue;
        # Rule #2 If point b is less than point c then
        #                   point d becomes point c, and
        #                   point b becomes 1/3 between a and new d
        #                   point c becomes 2/3 between a and new d
        if lossb < lossc:
d = c
b = a+ (d-a)/3
c = a+ 2*(d-a)/3
          continue
        # Rule #2 If point c is less than point b then
        #                   point a becomes point b, and
        #                   point b becomes 1/3 between new a and d
        #                   point c becomes 2/3 between new a and d
a = b
b = a+ (d-a)/3
2*(d-a)/3
    # Return average of two middle points
    return(b+c)/2.0
```

- 동적 학습률

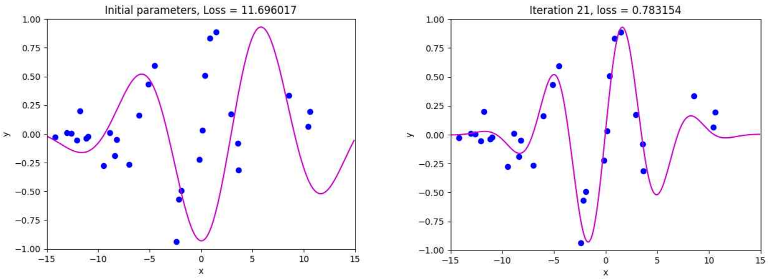
line_search(alpha)를 이용하여 학습률을 동적으로 설정

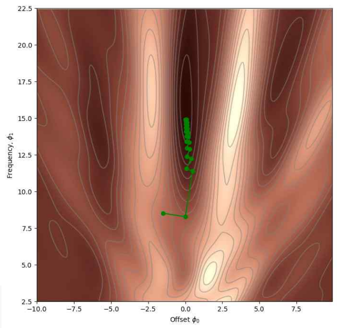
```
def gradient_descent_step(phi, data,  model):
  # gradient 계산
gradient = compute_gradient(data[0,:],data[1,:], phi)
  # parameters 업데이트 - 음의 방향으로 탐색
alpha = line_search(data, model, phi, gradient*-1, max_dist = 2.0)
phi = phi - alpha * gradient
  return phi
```

```
n_steps = 21
phi_all = np.zeros((2,n_steps+1))
phi_all[0,0] = -1.5
phi_all[1,0] = 8.5
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))
# gradient 하강법 스텝 수행
for c_step inrange (n_steps):
phi_all[:,c_step+1:c_step+2] = gradient_descent_step(phi_all[:,c_step:c_step+1],data, model)
  # 매 5번째 스텝마다 손실 측정 및 모델 시각화
  if c_step % 5== 0:
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)
```

- 고정된 학습률

```
def gradient_descent_step_fixed_learning_rate(phi, data, alpha):
```

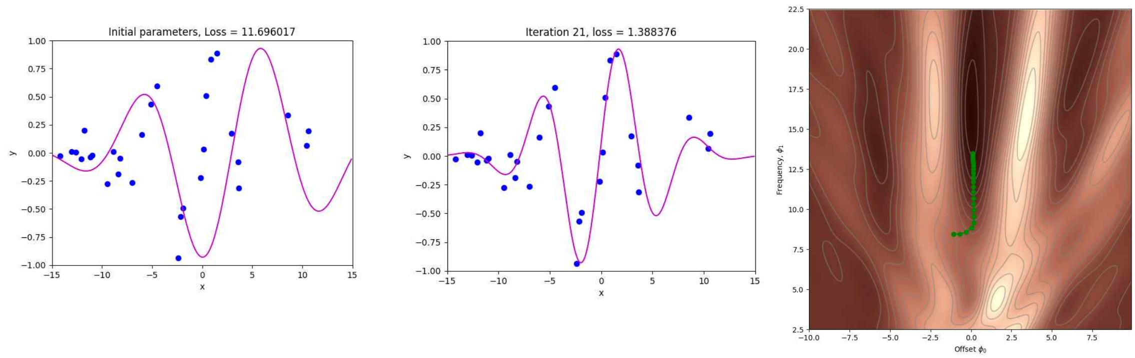





```
    # 기울기 계산
    grad = compute_gradient(data[0, :], data[1, :], phi)
    # 파라미터 업데이트
    phi = phi - alpha * grad
    return phi
```

```
n_steps = 21
phi_all = np.zeros((2,n_steps+1))
phi_all[0,0] = -1.5
phi_all[1,0] = 8.5
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))

for c_step in range (n_steps):
phi_all[:,c_step+1:c_step+2]  =  gradient_descent_step_fixed_learning_rate(phi_all[:,c_step:c_step+1],data,  alpha
=0.2)
  if c_step % 5== 0:
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]

```
# model시각화
def draw_model(data,model,phi,title=None):
x_model = np.arange(-15,15,0.1)
y_model = model(phi,x_model)
fix, ax = plt.subplots()
ax.plot(data[0,:],data[1,:],'bo')
ax.plot(x_model,y_model,'m-')
ax.set_xlim([-15,15]);ax.set_ylim([-1,1])
ax.set_xlabel('x'); ax.set_ylabel('y')
  iftitle is not None:
ax.set_title(title)
plt.show()
```

```
def draw_loss_function(compute_loss, data,  model, phi_itersNone):
  # my_colormap_vals_hex : 색상 맵을 정의하는데 사용
                         my_colormap_vals_hex =('2a0902', '2b0a03', '2c0b04', '2d0c05', '2e0c06', :
'fff8d9', 'fffada', 'fffbdc', 'fffcdd', 'fffedf', 'ffffe0')
  # 16진수 색상 값을 10진수로 변환하여 각 색상 구성 요소 (r, g, b)를 추출 후 my_colormap으로 결합하여 색상 맵 생성
my_colormap_vals_dec = np.array([int(element,base=16) forelement inmy_colormap_vals_hex])
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)
my_colormap = ListedColormap(np.vstack((r,g,b)).transpose()/255.0)
  # 시각화를 위한 offset/frequency values의 그리드 생성
offsets_mesh, freqs_mesh = np.meshgrid(np.arange(-10,10.0,0.1), np.arange(2.5,22.5,0.1))
loss_mesh = np.zeros_like(freqs_mesh)
  # 모든 파라미터 집합에 대한 손실 계산
  foridslope, slope innp.ndenumerate(freqs_mesh):
loss_mesh[idslope] = compute_loss(data[0,:], data[1,:], model, np.array([[offsets_mesh[idslope]], [slope]]))
fig,ax = plt.subplots()
fig.set_size_inches(8,8)
  # contourf는 채워진 등고선, contour는 선으로만 된 등고선
ax.contourf(offsets_mesh,freqs_mesh,loss_mesh,256,cmap=my_colormap)
ax.contour(offsets_mesh,freqs_mesh,loss_mesh,20,colors=['#80808080'])
  # parameter iteration path를 녹색 선과 원으로 출력
  ifphi_iters is not None:
ax.plot(phi_iters[0,:], phi_iters[1,:],'go-')
ax.set_ylim([2.5,22.5])
ax.set_xlabel('Offset $\phi_{0}$'); ax.set_ylabel('Frequency, $\phi_{1}$')
plt.show()

draw_loss_function(compute_loss, data, model)
```

```
 # 계산된 손실값과 정확한 손실값을 비교(계산된 손실값과 정확한 손실값이 일치 할 경우 올바른 코드 구현)
loss = compute_loss(data[0,:],data[1,:],model,np.array([[0.6],[-0.2]]))
print('Your loss = %3.3f, Correct loss = %3.3f'%(loss, 16.419))
```

```
Your loss = 16.419, Correct loss = 16.419
```

정확한 계산여부는 **finite differences.** 을 통해 알 수 있음. 함수를 평가한 후 파라미터 중 하나를 매우 작은 양만큼 변경하고 그 양으로 정규화하면 그래디언트에 대한 근사를 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 많은 경우 사용 불가능(parameters가 많을 경우 손실함수의 평가의 횟수도 증가하기 때문, gradients를 직접 계산하는 것이 더 효율적)

```
# finite differences 활용
delta = 0.0001
dl_dphi0_est = (compute_loss(data[0,:],data[1,:],model,phi+np.array([[delta],[0]])) - \
compute_loss(data[0,:],data[1,:],model,phi))/delta
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))
```