Question 1

(a) $R(w) = 13w_1^2 - 10w_1w_2 + 4w_1 + 2w_2^2 - 2w_2 + 1$

$$\frac{\partial R}{\partial w_1} = 26w_1 - 10w_2 + 4 = 0 \quad (1)$$

$$\frac{\partial R}{\partial w_2} = -10w_1 + 4w_2 - 2 = 0 \quad (2)$$

(optimality)

$$(2) \Rightarrow -25w_1 + 10w_2 - 5 = 0 \quad (3)$$

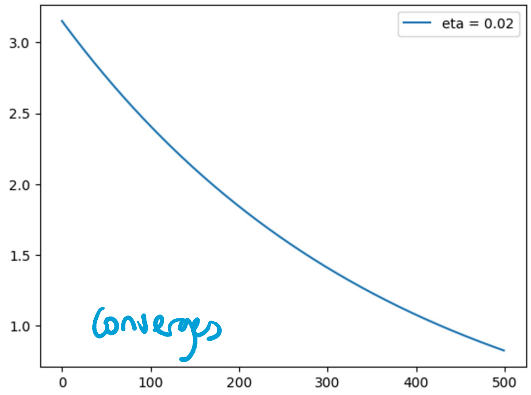
$$(1) + (3) \Rightarrow w_1 - 1 = 0 \Rightarrow \boxed{w_1 = 1} \quad (4)$$

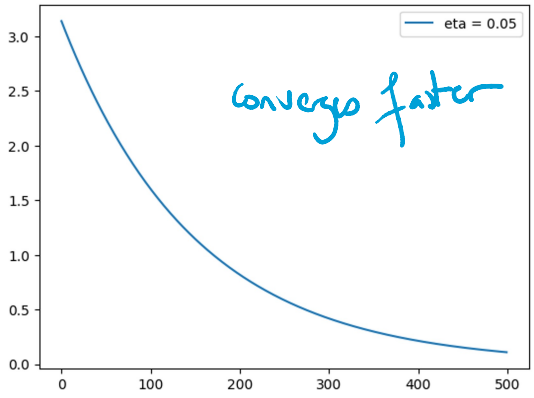
$$(2) + (4) \Rightarrow -10 + 4w_2 - 2 = 0 \Rightarrow \boxed{w_2 = 3}$$

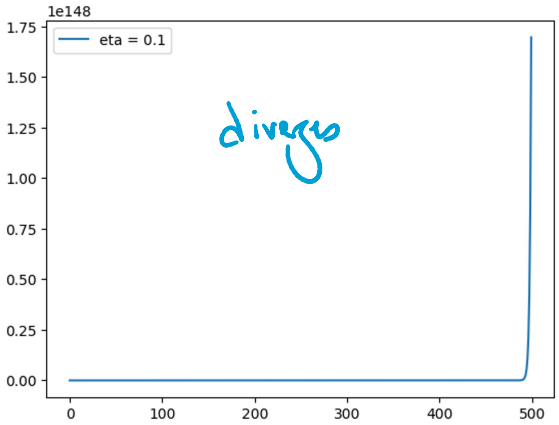
(b)

```
import numpy as np
import matplotlib.pyplot as plt
w_opt=[1,3]
number_of_epochs = 500
for eta in [0.02, 0.05, 0.1]:
  dist=np.zeros(number_of_epochs)
  w=np.zeros((2,))
  for i in range(number_of_epochs):
    grad = np.array([4+26*w[0]-10*w[1],-2-10*w[0]+4*w[1]])
    w = w - eta*grad
    dist[i]=np.linalg.norm(w-w_opt)
```

```
plt.plot(dist,label="eta = "+ str(eta))
plt.legend()
plt.show()
```







(c) Larger η is not always beneficial, because it would lead to instability (even if the gradients are exact, like here). This is because the 1st-order Taylor approximation is no longer very accurate.

Question 2

(a) $\varphi(v) = \frac{1}{1+e^{-5v}}$

$\varphi'(v) = 5\varphi(v)(1-\varphi(v))$

(b)

W	is	3×2	$(\dim(z) \times \dim(x))$
b	is	3	$(\dim(z))$
U	is	1×3	$(\dim(f) \times \dim(z))$
c	is	1	$(\dim(f))$

Forward equations!

$v_z = Wx + b \qquad z = \varphi(v_z)$

$v_f = Uz + c \qquad f = \varphi(v_f)$

```
v_z = np.matmul(W,x)+b
z = phi(v_z)
v_f = np.matmul(U,z)+c
f = phi(v_f)
```

(c) Backward equations for $\ell(y,f) = (y-f)^2$

$\nabla_f \ell = \frac{\partial \ell}{\partial f} = -2(y-f) \qquad \delta_f = \frac{\partial f}{\partial \ell} \cdot \varphi'(v_f)$

$$\nabla_f \ell = \frac{\partial \ell}{\partial f} = -2(y-f) \qquad \delta_f = \frac{\partial f}{\partial e} \cdot \varphi'(v_f)$$

$$\nabla_z \ell = U^T \delta_f \qquad \delta_z = \nabla_z \ell \cdot \varphi'(v_z)$$

$$\nabla_W \ell = \delta_z x^T \qquad \nabla_b = \delta_z$$

$$\nabla_U \ell = \delta_f z^T \qquad \nabla_c = \delta_f$$

```
dloss_df = -2*(y-f)
delta_f = dloss_df * phi_prime(v_f)
dloss_dz = np.matmul(np.transpose(U),delta_f)
delta_z = dloss_dz * phi_prime(v_z)
grad_W = np.matmul(delta_z, np.transpose(x))
grad_b = delta_z * 1
grad_U = np.matmul(delta_f, np.transpose(z))
grad_c = delta_f * 1
```

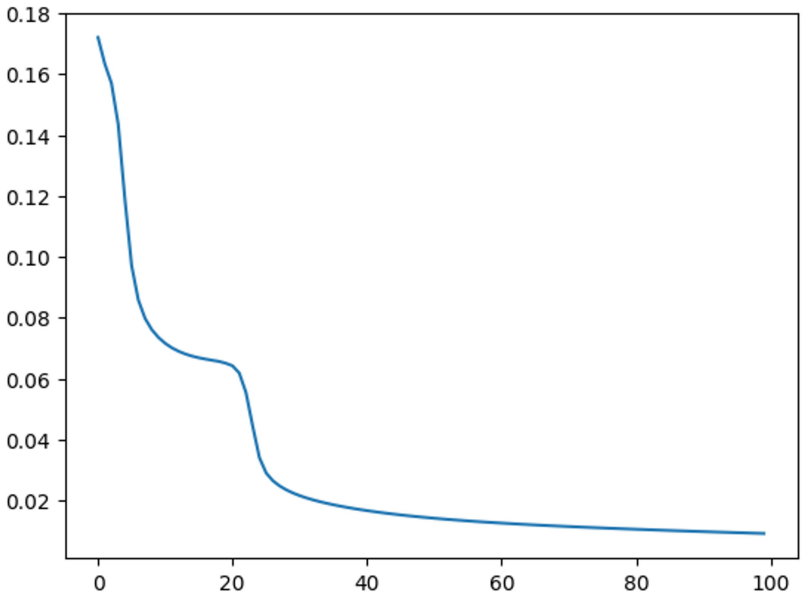
(d) Initialization:

```
sigma=0.1
W = np.random.randn(k,X.shape[0])*sigma
b = np.random.randn(k,1)*sigma
U = np.random.randn(1,k)*sigma
c = np.random.randn(1,1)*sigma
```

SGD:

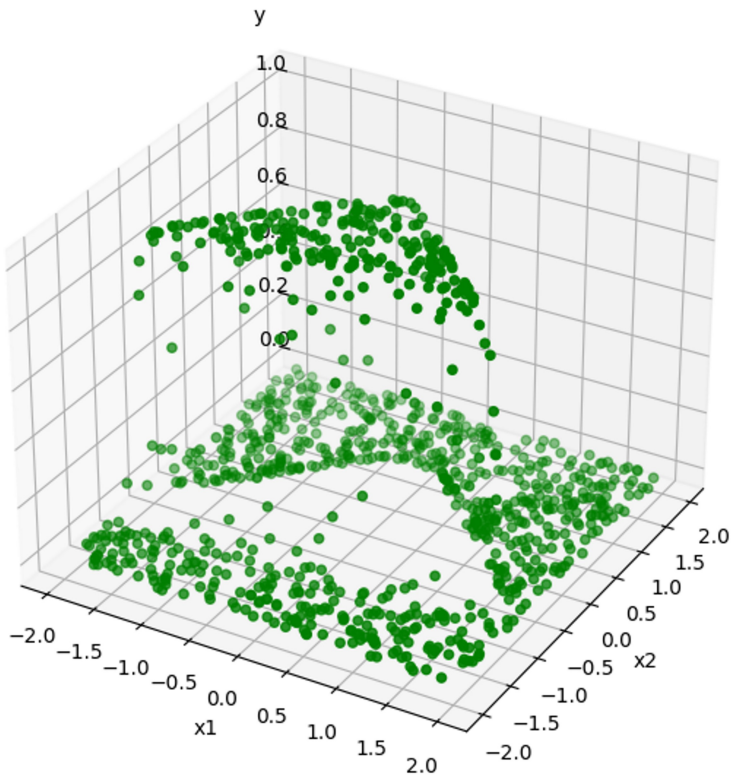
```
eta = 0.01
epochs = 100
risk = np.zeros(epochs)
```

```
for epoch in range(epochs):
  for i in range(N):
    x = X[:,i].reshape(x.shape)
    y = Y[0,i].reshape(f.shape)
    # Forward - Question 2(b)
    v_z = np.matmul(W,x)+b
    z = phi(v_z)
    v_f = np.matmul(U,z)+c
    f = phi(v_f)
    # Backward - Question 2(c)
    dloss_df = -2*(y-f)
    delta_f = dloss_df * phi_prime(v_f)
    dloss_dz = np.matmul(np.transpose(U),delta_f)
    delta_z = dloss_dz * phi_prime(v_z)
    grad_W = np.matmul(delta_z, np.transpose(x))
    grad_b = delta_z * 1
    grad_U = np.matmul(delta_f, np.transpose(z))
    grad_c = delta_f * 1
    # SGD
    W = W - eta * grad_W
    b = b - eta * grad_b
    U = U - eta * grad_U
    c = c - eta * grad_c
  Y_predicted = phi(np.matmul(U,phi(np.matmul(W,X)+b))+c)
  risk[epoch] = np.sum((Y-Y_predicted)**2)/N
plt.plot(risk)
plt.show()
```



(e) Visualization

```
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, Y_predicted, color = "green")
ax.set_xlabel('x1'); ax.set_ylabel('x2'); ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```



(e) This question is a bit open-ended.

- Changing η has its usual pitfalls. An adaptive η works remarkably well (many methods like Adam and Adagrad do this by monitoring the

Adam and Adagrad do this by monitoring the gradients themselves.)

- Minibatch is helpful to smooth out the gradients of SGD. However, we shouldn't simply accumulate the gradients, since then ther variance increases. Instead, we need to average across the minibatch.

- Initializing weights at 0 slows the convergence considerably, even halting it (by getting stuck in local minima.) Local minima are also an issue with very large σ for the random initialization.

- Making a big in φ makes it "less differentiable" do more unstable, because of both large (at the transition) and very small (at saturation) gradients. However,

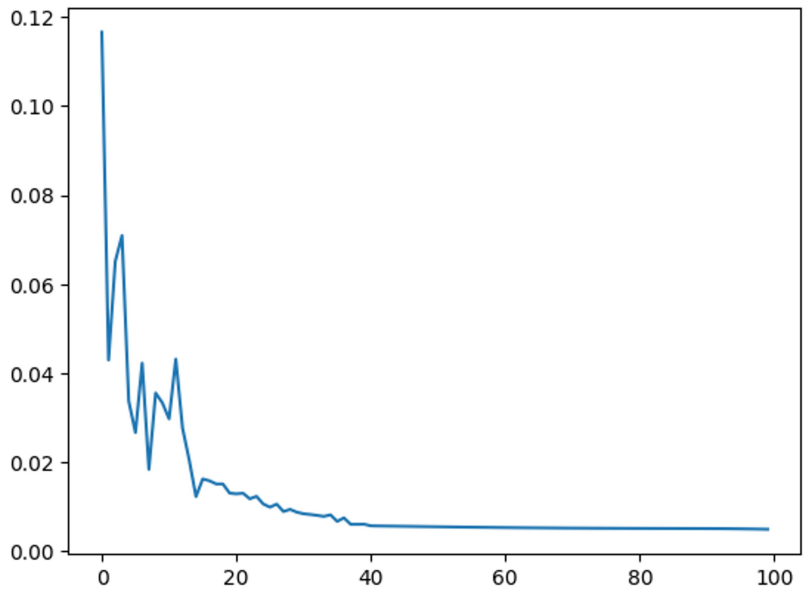
small (at saturation) gradients. However, the end results are crisper, as long as we can stabilize (adaptive η + minibatch help.)

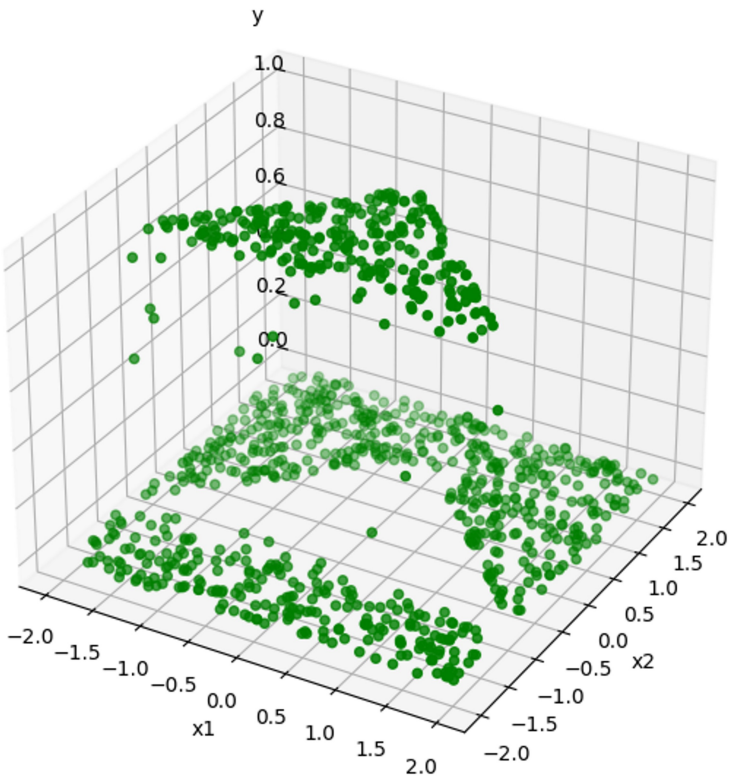
The following code does:

- Sets $\alpha = 10$ in φ.
- Adaptive η, starting at $\eta = 0.1$
- Minibatch with batch size 5, and with averaged gradients.

```
def phi(v):
  return 1./(1.+np.exp(-10*v))
def phi_prime(v):
  return 10*phi(v)*(1-phi(v))
sigma=0.1
# Initialization
W = np.random.randn(k,X.shape[0])*sigma
b = np.random.randn(k,1)*sigma
U = np.random.randn(1,k)*sigma
c = np.random.randn(1,1)*sigma
# W=W*0 # Zero initialization
# b=b*0
# U=U*0
# c=c*0
eta = 0.1
epochs = 100
batch_size = 5
risk = np.zeros(epochs)
grad_W = 0*W
grad_b = 0*b
grad_U = 0*U
grad_c = 0*c
for epoch in range(epochs):
  for i in range(N):
    x = X[:,i].reshape(x.shape)
    y = Y[0,i].reshape(f.shape)
    # Forward
    v_z = np.matmul(W,x)+b
    z = phi(v_z)
```

```
    v_f = np.matmul(U,z)+c
    f = phi(v_f)
    # Backward
    dloss_df = -2*(y-f)
    delta_f = dloss_df * phi_prime(v_f)
    dloss_dz = np.matmul(np.transpose(U),delta_f)
    delta_z = dloss_dz * phi_prime(v_z)
    grad_W += np.matmul(delta_z, np.transpose(x))
    grad_b += delta_z * 1
    grad_U += np.matmul(delta_f, np.transpose(z))
    grad_c += delta_f * 1
    # Minibatch SGD
    if (i+1)%batch_size==0:
      W = W - eta * grad_W/batch_size # Averaged
      b = b - eta * grad_b/batch_size # gradients
      U = U - eta * grad_U/batch_size
      c = c - eta * grad_c/batch_size
      grad_W = 0*W # Zeroing out the gradients
      grad_b = 0*b
      grad_U = 0*U
      grad_c = 0*c
  Y_predicted = phi(np.matmul(U,phi(np.matmul(W,X)+b))+c)
  risk[epoch] = np.sum((Y-Y_predicted)**2)/N
  if epoch > 1:
    if risk[epoch]>risk[epoch-1]:
      eta = eta*0.9
plt.plot(risk)
plt.show()
# Visualization
fig = plt.figure(figsize = (7, 10))
ax = plt.axes(projection ="3d")
ax.scatter3D(x1, x2, Y_predicted, color = "green")
ax.set_xlabel('x1'); ax.set_ylabel('x2'); ax.set_zlabel('y')
ax.zaxis._axinfo['juggled'] = (2, 2, 1)
plt.show()
```





Note the faster convergence and the crisper output. The convergence is noisier due to the large step sizes, but it's controlled thanks to the adaptation of η and the averaging inside the mini-batch.

(set batch_size=20 to see it get smoother but a bit slower.)