Exercises week 41 (exploration)

FYS-STK4155

Even Sletteng Garvang

```python
# import libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
```

Here I implement various methods of gradient descent (GD) for a simple synthetic dataset generated by a polynomial function.

Generate data

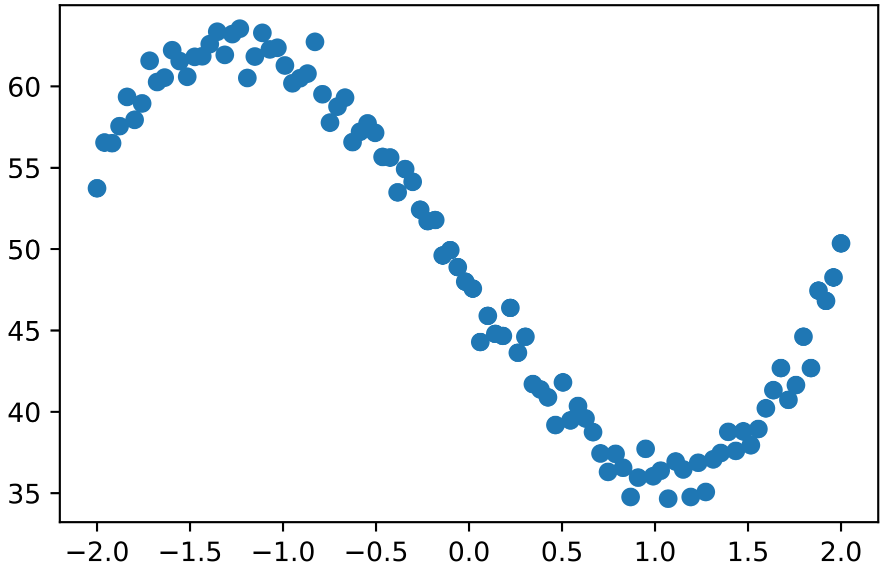
```python
def data_function(x):
    return 4*x**3 + x**2 - 17*x + 48

np.random.seed(8923)

# generate data
n = 100
x = np.linspace(-2, 2, n)
f_x = data_function(x)
y = f_x + np.random.normal(0, 1, n)
y = y.reshape(-1,1)

# create design matrix of polynomials
# for now 3rd order reflecting data function
X = PolynomialFeatures(3).fit_transform(x.reshape(-1, 1))

plt.plot(x, y, "o")
plt.show()
```



Plain gradient descent

```
def gradient_OLS(X, y, theta):
    n = y.shape[0]
    return -(2.0/n) * X.T @ (y - X @ theta)

def gradient_descent(eta, X, y, n_iter, gradient_fun=gradient_OLS, check_converge=False, thre
    theta = np.random.randn(X.shape[1], 1)
    for i in range(n_iter):
        gradient = gradient_fun(X, y, theta)
        if check_converge:
            if np.all(np.abs(gradient) < thresh):
                print(f"Converged for eta={round(eta, 3)} after {i+1} iterations.")
                return theta
        theta -= eta * gradient
    if check_converge:
        print(f"Did not converge for eta={round(eta, 3)} and {n_iter} iterations.")
    return theta


np.random.seed(50)
etas = np.linspace(0.01, 0.1, 10)
n_iter = 1000
```

```python
for eta in etas:
    theta = gradient_descent(eta, X, y, n_iter, check_converge=True)
```

```
Did not converge for eta=0.01 and 1000 iterations.
Did not converge for eta=0.02 and 1000 iterations.
Converged for eta=0.03 after 739 iterations.
Converged for eta=0.04 after 556 iterations.
Converged for eta=0.05 after 449 iterations.
Converged for eta=0.06 after 373 iterations.
Converged for eta=0.07 after 318 iterations.
Converged for eta=0.08 after 279 iterations.
Converged for eta=0.09 after 254 iterations.
Did not converge for eta=0.1 and 1000 iterations.
```

A learning rate of 0.09 works best for this data, converging after 254 iterations. This was with convergence criterion of all gradients being smaller than 0.001.

```python
theta_analytical = np.linalg.inv(X.T @ X) @ X.T @ y
np.random.seed(50)
eta = 0.09
n_iter = 254
theta = gradient_descent(eta, X, y, n_iter)

print("Analytical:")
print(theta_analytical)
print("Gradient descent (plain):")
print(theta)
```

```
Analytical:
[[ 47.89078067]
 [-16.65248688]
 [  1.06213488]
 [  3.87029735]]
Gradient descent (plain):
[[ 47.89077985]
 [-16.65052384]
 [  1.06213526]
 [  3.86961629]]
```

Gradient descent with momentum

```
# function for momentum
def momentum_change(eta, gradient, gamma, change):
    return eta * gradient + gamma * change

# modify for momentum function
def gradient_descent_momentum(eta, X, y, n_iter, gamma, gradient_fun=gradient_OLS, check_conv
    theta = np.random.randn(X.shape[1], 1)
    change = 0
    for i in range(n_iter):
        gradient = gradient_fun(X, y, theta)
        if check_converge:
            if np.all(np.abs(gradient) < thresh):
                print(f"Converged for eta={round(eta, 3)} after {i+1} iterations.")
                return theta
        new_change = momentum_change(eta, gradient, gamma, change)
        change = new_change
        theta -= new_change

    if check_converge:
        print(f"Did not converge for eta={round(eta, 3)} and {n_iter} iterations.")
    return theta

np.random.seed(50)
gamma = 0.7
n_iter = 200
np.random.seed(50)
etas = np.linspace(0.01, 0.1, 10)
n_iter = 1000
for eta in etas:
    theta = gradient_descent_momentum(eta, X, y, n_iter, gamma, check_converge=True)
```

```
Converged for eta=0.01 after 655 iterations.
Converged for eta=0.02 after 313 iterations.
Converged for eta=0.03 after 199 iterations.
Converged for eta=0.04 after 143 iterations.
Converged for eta=0.05 after 108 iterations.
Converged for eta=0.06 after 83 iterations.
Converged for eta=0.07 after 64 iterations.
Converged for eta=0.08 after 63 iterations.
```

```
Converged for eta=0.09 after 66 iterations.
Converged for eta=0.1 after 59 iterations.
```

For $\gamma = 0.7$, the algorithm converges much faster than without momentum, and converges for the whole range of learning rates.

Stochastic gradient descent

We implement stochastic gradient descent, which helps with avoiding getting stuck in local minima.

```
def SGD(X, y, eta, gradient_fun, n_epochs, M):
    n = y.shape[0]
    m = int(n/M)
    xy = np.column_stack([X,y]) # for shuffling x and y together
    theta = np.random.randn(X.shape[1], 1)

    for i in range(n_epochs):
        np.random.shuffle(xy)
        for j in range(m):
            random_index = M * np.random.randint(m)
            xi = xy[random_index:random_index+5, :-1]
            yi = xy[random_index:random_index+5, -1:]
            gradient = (1/M)*gradient_fun(xi, yi, theta)
            theta = theta - eta*gradient
    return theta

np.random.seed(39)
eta = 0.01
n_epochs = 200
M = 5        # minibatch size


theta = SGD(X, y, eta, gradient_OLS, n_epochs, M)

print(theta)
print(theta_analytical)
```

```
[[ 47.77879581]
 [-15.97183846]
 [  1.11092537]
```

```
 [  3.63334777]]
[[ 47.89078067]
 [-16.65248688]
 [  1.06213488]
 [  3.87029735]]
```

SGD with momentum

```
def SGD_momentum(X, y, eta, gradient_fun, n_epochs, M, gamma):
    n = y.shape[0]
    m = int(n/M)
    xy = np.column_stack([X,y]) # for shuffling x and y together
    theta = np.random.randn(X.shape[1], 1)

    change = 0
    for i in range(n_epochs):
        np.random.shuffle(xy)
        for j in range(m):
            random_index = M * np.random.randint(m)
            xi = xy[random_index:random_index+5, :-1]
            yi = xy[random_index:random_index+5, -1:]
            gradient = (1/M)*gradient_fun(xi, yi, theta)
            new_change = momentum_change(eta, gradient, gamma, change)
            theta = theta - new_change
            change = new_change
    return theta

np.random.seed(60)
eta = 0.01
gamma = 0.3
n_epochs = 100
M = 5         # minibatch size

theta = SGD_momentum(X, y, eta, gradient_OLS, n_epochs, M, gamma)

print(theta)
print(theta_analytical)
```

```
[[ 47.4171236 ]
 [-14.98964006]
```

```
 [  1.30777058]
 [  3.19582651]]
[[ 47.89078067]
 [-16.65248688]
 [  1.06213488]
 [  3.87029735]]
```

AdaGrad

```python
def AdaGrad(gradient, Giter, eta, delta = 1e-8):
    Giter += gradient*gradient
    update = gradient * eta / (delta + np.sqrt(Giter))
    return Giter, update

def GD_AdaGrad(eta, X, y, n_iter, gradient_fun=gradient_OLS):
    theta = np.random.randn(X.shape[1], 1)
    Giter = 0
    for i in range(n_iter):
        gradient = gradient_fun(X, y, theta)
        Giter, update = AdaGrad(gradient, Giter, eta)
        theta -= update
    return theta

eta = 0.5
n_iter = 200

theta = GD_AdaGrad(eta, X, y, n_iter)
print(theta)
print(theta_analytical)
```

```
[[ 13.2019473 ]
 [-10.06549974]
 [ 10.91808174]
 [  1.57741944]]
[[ 47.89078067]
 [-16.65248688]
 [  1.06213488]
 [  3.87029735]]
```

RMSProp

```python
def RMSProp(gradient, Giter, eta, rho, delta=1e-8):
    Giter = rho*Giter + (1-rho)*gradient*gradient
    update = gradient*eta/(delta+np.sqrt(Giter))
    return Giter, update

def GD_RMSProp(eta, X, y, n_iter, gradient_fun=gradient_OLS):
    theta = np.random.randn(X.shape[1], 1)
    Giter = 0
    for i in range(n_iter):
        gradient = gradient_fun(X, y, theta)
        Giter, update = RMSProp(gradient, Giter, eta, rho)
        theta -= eta * gradient
    return theta
```

General GD function

```python
# function for momentum
def momentum_change(eta, gradient, gamma, change):
    return eta * gradient + gamma * change

def AdaGrad(update_term, gradient, Giter, delta = 1e-8):
    Giter += gradient*gradient
    update = update_term / (delta + np.sqrt(Giter))
    return Giter, update

def RMSProp(update_term, gradient, Giter, rho, delta=1e-8):
    Giter = rho*Giter + (1-rho)*gradient*gradient
    update = update_term / (delta+np.sqrt(Giter))
    return Giter, update

def ADAM(gradient, first_moment, second_moment, beta1, beta2, itr, delta=1e-8):
    first_moment = beta1*first_moment + (1-beta1)*gradient
    second_moment = beta2*second_moment + (1-beta2)*gradient*gradient

    first_term = first_moment/(1.0-beta1**itr)
    second_term = second_moment/(1.0-beta2**itr)
    update = eta*first_term/(np.sqrt(second_term)+delta)
```

```
    return first_moment, second_moment, update

def GD_inner(eta, theta, moments, gradient, momentum=False, gamma=None, adaptive_fun=None, ad
    adaptive = adaptive_fun is not None

    if adam:
        first_moment, second_moment = moments
        first_moment, second_moment, update = ADAM(gradient, first_moment, second_moment, **
        theta -= update
        return theta, first_moment, second_moment
    else:
        Giter, change = moments
        update = eta * gradient
        if momentum:
            update += gamma * change
            change = update
        if adaptive:
            Giter, update = adaptive_fun(update, gradient, Giter, **kwargs)

    theta -= update
    return theta, Giter, change

def GD(X, y, eta, n_iter, gradient_fun=gradient_OLS, momentum=False, gamma=None, adaptive_fun
    theta = np.random.randn(X.shape[1], 1)

    # moment 1 and 2 of ADAM
    # Giter and change if not ADAM
    moments = [0, 0]

    for i in range(n_iter):
        gradient = gradient_fun(X, y, theta)
        if adam:
            theta, moments[0], moments[1] = GD_inner(eta, theta, moments, gradient, momentum
        else:
            theta, moments[0], moments[1] = GD_inner(eta, theta, moments, gradient, momentum
    return theta

def SGD(X, y, eta, M, n_epochs, gradient_fun=gradient_OLS, momentum=False, gamma=None, adapti
    n = y.shape[0]
    m = int(n/M)
    xy = np.column_stack([X,y]) # for shuffling x and y together
    theta = np.random.randn(X.shape[1], 1)
```

```
    # moment 1 and 2 of ADAM
    # Giter and change if not ADAM
    moments = [0, 0]

    for i in range(n_epochs):
        Giter = 0.0
        np.random.shuffle(xy)
        for j in range(m):
            random_index = M * np.random.randint(m)
            xi = xy[random_index:random_index+5, :-1]
            yi = xy[random_index:random_index+5, -1:]
            gradient = (1/M)*gradient_fun(xi, yi, theta)
            if adam:
                theta, moments[0], moments[1] = GD_inner(eta, theta, moments, gradient, momen
            else:
                theta, moments[0], moments[1] = GD_inner(eta, theta, moments, gradient, momen
    return theta
```

```
eta = 0.5
n_iter = 1000
beta1 = 0.9
beta2 = 0.999
theta = GD(X, y, eta, n_iter, adam=True, beta1=beta1, beta2=beta2)
print(theta)

M = 5
n_epochs = 100
theta_sgd_adam = SGD(X, y, eta, M, n_epochs, adam=True, beta1=beta1, beta2=beta2)

print(theta_sgd_adam)
```

```
[[ 47.89078067]
 [-16.65248688]
 [  1.06213488]
 [  3.87029735]]
[[ 47.93196761]
 [-16.59502347]
 [  1.02168202]
 [  3.83096949]]
```

```
eta = 0.09
n_iter = 100


theta = GD(X, y, eta, n_iter)
np.random.seed(56)
theta_momentum = GD(X, y, eta, n_iter, momentum=True, gamma = 0.3)
np.random.seed(56)
theta_momentum_old = gradient_descent_momentum(eta, X, y, n_iter, gamma=0.3)

print(theta)
print(theta_momentum)
print(theta_momentum_old)


theta_adagrad = GD(X, y, eta=10, n_iter=n_iter, adaptive_fun=AdaGrad)
theta_adagrad_momentum = GD(X, y, eta=10, n_iter=n_iter, momentum=True, gamma = 0.3, adaptive




print("Adagrad:")
print(theta_adagrad)
print(theta_adagrad_momentum)

theta_rmsprop = GD(X, y, eta=10, n_iter=n_iter, adaptive_fun=RMSProp, rho=0.99)
print("RMSProp")
print(theta_rmsprop)
```

```
[[ 47.85354031]
 [-16.18668031]
 [  1.07927133]
 [  3.73293436]]
[[ 47.88976967]
 [-16.56208425]
 [  1.0626001 ]
 [  3.83864136]]
[[ 47.88976967]
 [-16.56208425]
 [  1.0626001 ]
 [  3.83864136]]
Adagrad:
```

```
[[ 47.75739966]
 [-16.43216276]
 [  1.12418121]
 [  3.79094167]]
[[ 47.88739958]
 [-16.63144066]
 [  1.06372826]
 [  3.86272304]]
RMSProp
[[ 47.89078066]
 [-16.49087085]
 [  1.06213489]
 [  3.81212571]]
```

```
eta = 0.09
M = 5
n_epochs = 50
# test SGD
theta = SGD(X, y, eta, M, n_epochs)
theta_momentum = SGD(X, y, eta, M, n_epochs, momentum=True, gamma = 0.3)

print(theta)
print(theta_momentum)

eta = 10
theta_adagrad = SGD(X, y, eta, M, n_epochs, adaptive_fun=AdaGrad)
theta_adagrad_momentum = SGD(X, y, eta, M, n_epochs, momentum=True, gamma = 0.3, adaptive_fu

print("Adagrad:")
print(theta_adagrad)
print(theta_adagrad_momentum)
```

```
[[ 47.76360679]
 [-16.6772413 ]
 [  1.04379253]
 [  3.80506926]]
[[ 47.93549226]
 [-16.69514403]
 [  1.13236295]
 [  3.89640275]]
Adagrad:
[[ 47.96023195]
```

```
 [-16.64580242]
 [  1.16326879]
 [  3.87458737]]
[[ 47.73724392]
 [-16.60879717]
 [  0.76667081]
 [  3.72090355]]
```