Gradient descent exploration

Here I test the gradient descent algorithm on a simple polynomial, before going on to using it on the Franke function and then real terrain data.

```
import autograd.numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.model_selection import train_test_split
from imageio import imread

from ptwo.utils import calculate_polynomial, franke_function
from ptwo.models import GradientDescent
from ptwo.optimizers import Momentum, ADAM, AdaGrad, RMSProp
from ptwo.gradients import grad_OLS, grad_ridge
from ptwo.costfuns import mse
```

Polynomial data

Start by generating data for a polynomial expression

```
np.random.seed(8923)

n = 100
x = np.linspace(-3, 2, n)
poly_list = np.array([3, 3, 1, 4])
y = calculate_polynomial(x, *poly_list)
y = y.reshape(-1,1)

# choose polynomial degree that matches design
```

```
X = PolynomialFeatures(len(poly_list) - 1).fit_transform(x.reshape(-1, 1))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)

scalerx = StandardScaler()
X_train_scaled = scalerx.fit_transform(X_train)
X_test_scaled = scalerx.transform(X_test)

scalery = StandardScaler()
y_train_scaled = scalerx.fit_transform(y_train)
y_test_scaled = scalerx.transform(y_test)
```

Then we test a standard gradient descent without optimizers. For this data generated by polynomials, we just compare the coefficients with the true values used to generate the data.

```
learning_rate = 0.01
n_iter = 200
grad = grad_OLS()

gd = GradientDescent(learning_rate, grad)

gd.descend(X, y, epochs = n_iter)
print("Regular", gd.theta, sep = "\n")
```

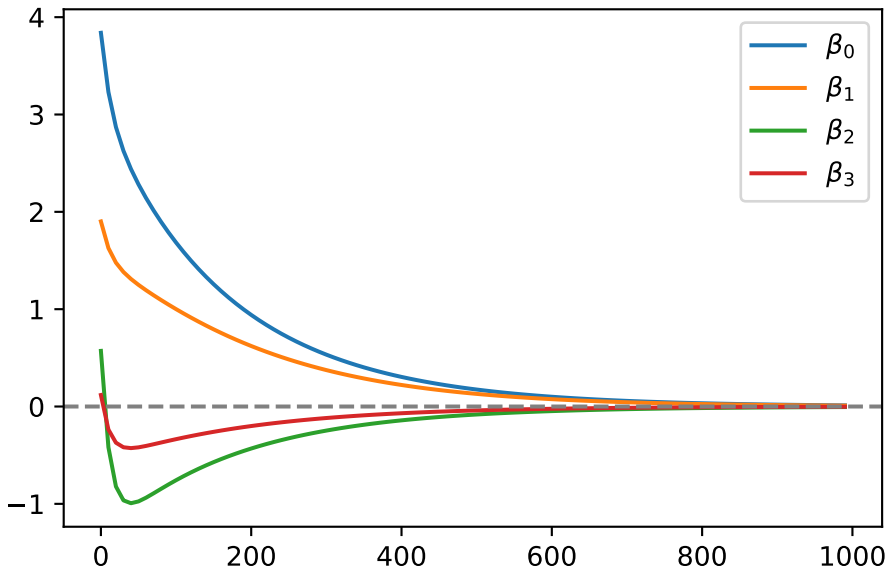
```
Regular
[[2.55269716]
 [2.52486346]
 [1.22872974]
 [4.13276221]]
```

It does not seem to converge completely after 200 iterations, let's look at convergence.

```
learning_rate = 0.01
n_iter = 1000
grad = grad_OLS()

gd = GradientDescent(learning_rate, grad)

step_size = 10
n_steps = int(n_iter / step_size)
convergence = np.zeros((n_steps,X.shape[1]))
```

```
print(convergence.shape)
iters = np.zeros(n_steps)
for i in range(n_steps):
    gd.descend(X, y, epochs = step_size)
    new_coef = gd.theta
    convergence[i, :] = (poly_list.reshape(-1, 1) - new_coef).flatten()
    iters[i] = i*step_size

print(gd.theta)

for i in range(X.shape[1]):
    plt.plot(iters, convergence[:,i], label=fr"$\beta_{i}$")

plt.legend()
plt.axhline(0, linestyle = "--", color="grey")
plt.show()
```

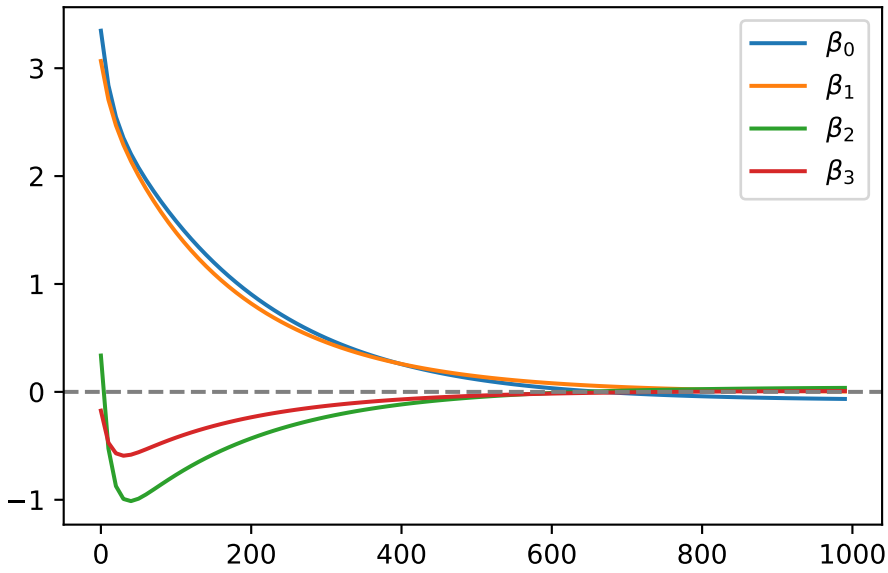
```
(100, 4)
[[2.98826017]
 [2.99102301]
 [1.00554423]
 [4.00275645]]
```



The values converge at the true values over time! With (quite a bit of) noise they converge at slightly different values:

```
# add noise
yn = y + np.random.normal(0, 1, (n,1))

learning_rate = 0.01
n_iter = 1000
grad = grad_OLS()

gd = GradientDescent(learning_rate, grad)

step_size = 10
n_steps = int(n_iter / step_size)
convergence = np.zeros((n_steps,X.shape[1]))
print(convergence.shape)
iters = np.zeros(n_steps)
for i in range(n_steps):
    gd.descend(X, yn, epochs = step_size)
    new_coef = gd.theta
    convergence[i, :] = (poly_list.reshape(-1, 1) - new_coef).flatten()
    iters[i] = i*step_size

print(gd.theta)

for i in range(X.shape[1]):
    plt.plot(iters, convergence[:,i], label=fr"$\beta_{i}$")

plt.legend()
plt.axhline(0, linestyle = "--", color="grey")
plt.show()
```

```
(100, 4)
[[3.0654166 ]
 [2.9969612 ]
 [0.96297431]
 [3.99210798]]
```

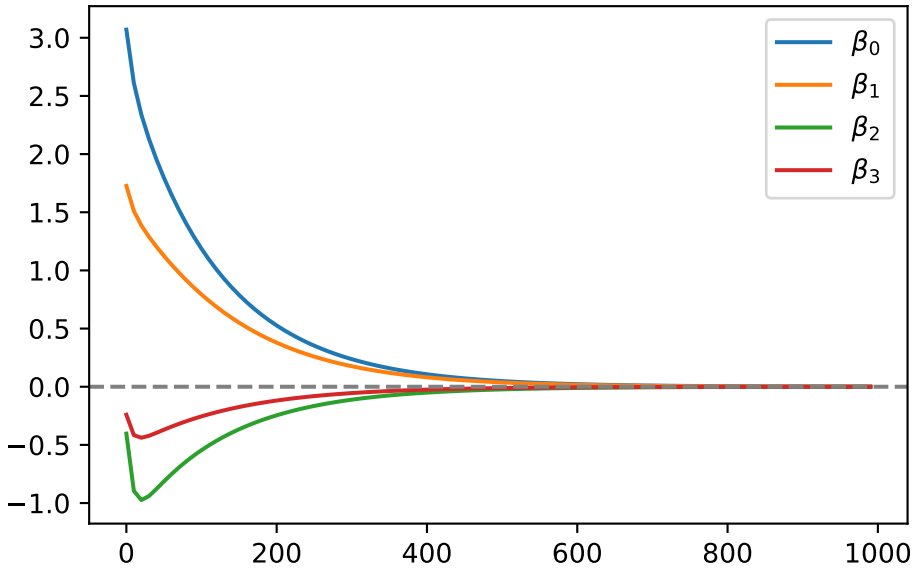


We stay with the no-noise data, and check how momentum affect convergence times.

```
learning_rate = 0.01
n_iter = 1000
grad = grad_OLS()

gd = GradientDescent(learning_rate, grad, optimizer=Momentum(0.3))

step_size = 10
n_steps = int(n_iter / step_size)
convergence = np.zeros((n_steps,X.shape[1]))
print(convergence.shape)
iters = np.zeros(n_steps)
for i in range(n_steps):
    gd.descend(X, y, epochs = step_size)
    new_coef = gd.theta
    convergence[i, :] = (poly_list.reshape(-1, 1) - new_coef).flatten()
    iters[i] = i*step_size

print(gd.theta)

for i in range(X.shape[1]):
    plt.plot(iters, convergence[:,i], label=fr"$\beta_{i}$")

plt.legend()
```

```
plt.axhline(0, linestyle = "--", color="grey")
plt.show()
```

```
(100, 4)
[[2.99897529]
 [2.99921472]
 [1.00048415]
 [4.00024096]]
```

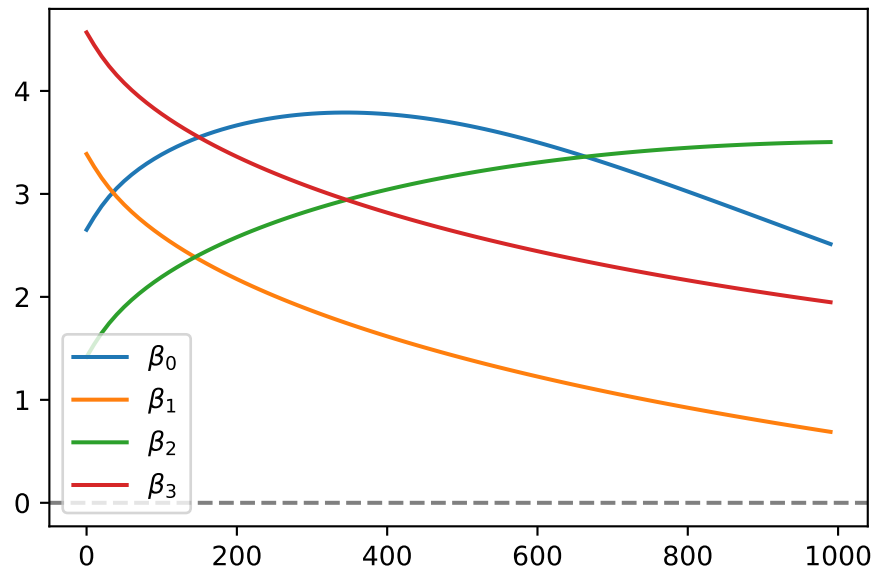


The parameters converge quicker with momentum. Try another optimizer:

```
learning_rate = 0.01
n_iter = 1000
grad = grad_OLS()

gd = GradientDescent(learning_rate, grad, optimizer = ADAM())

step_size = 10
n_steps = int(n_iter / step_size)
convergence = np.zeros((n_steps,X.shape[1]))
print(convergence.shape)
iters = np.zeros(n_steps)
for i in range(n_steps):
    gd.descend(X, y, epochs = step_size)
```

```
    new_coef = gd.theta
    convergence[i, :] = (poly_list.reshape(-1, 1) - new_coef).flatten()
    iters[i] = i*step_size

print(gd.theta)

for i in range(X.shape[1]):
    plt.plot(iters, convergence[:,i], label=fr"$\beta_{i}$")

plt.legend()
plt.axhline(0, linestyle = "--", color="grey")
plt.show()
```

```
(100, 4)
[[ 0.48723525]
 [ 2.3109661 ]
 [-2.50350787]
 [ 2.05200032]]
```

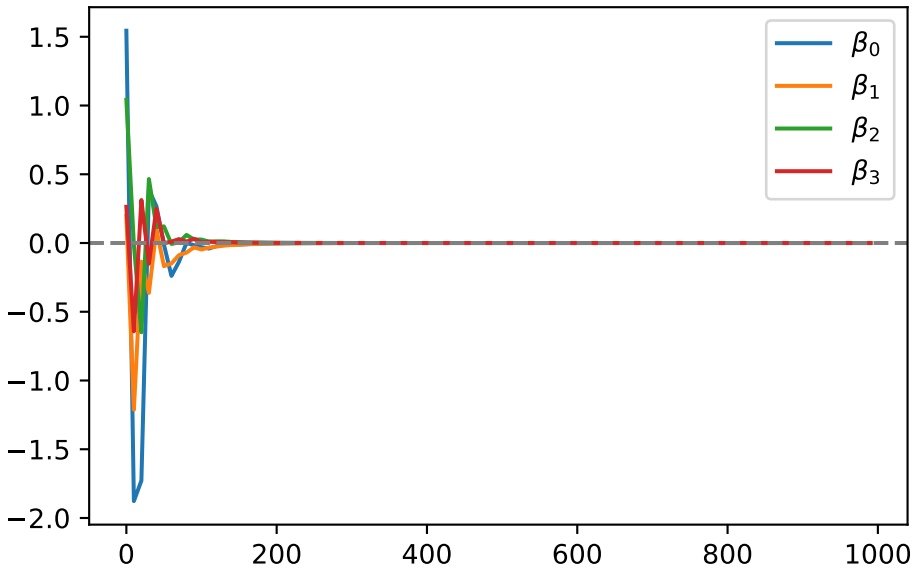


It looks like it's heading there slowly, but ADAM performs poorly for this particular data and learning rate. Try another learning rate:

```
learning_rate = 1
n_iter = 1000
```

```
grad = grad_OLS()

gd = GradientDescent(learning_rate, grad, optimizer = ADAM())

step_size = 10
n_steps = int(n_iter / step_size)
convergence = np.zeros((n_steps,X.shape[1]))
print(convergence.shape)
iters = np.zeros(n_steps)
for i in range(n_steps):
    gd.descend(X, y, epochs = step_size)
    new_coef = gd.theta
    convergence[i, :] = (poly_list.reshape(-1, 1) - new_coef).flatten()
    iters[i] = i*step_size

print(gd.theta)

for i in range(X.shape[1]):
    plt.plot(iters, convergence[:,i], label=fr"$\beta_{i}$")

plt.legend()
plt.axhline(0, linestyle = "--", color="grey")
plt.show()
```

```
(100, 4)
[[3.]
 [3.]
 [1.]
 [4.]]
```



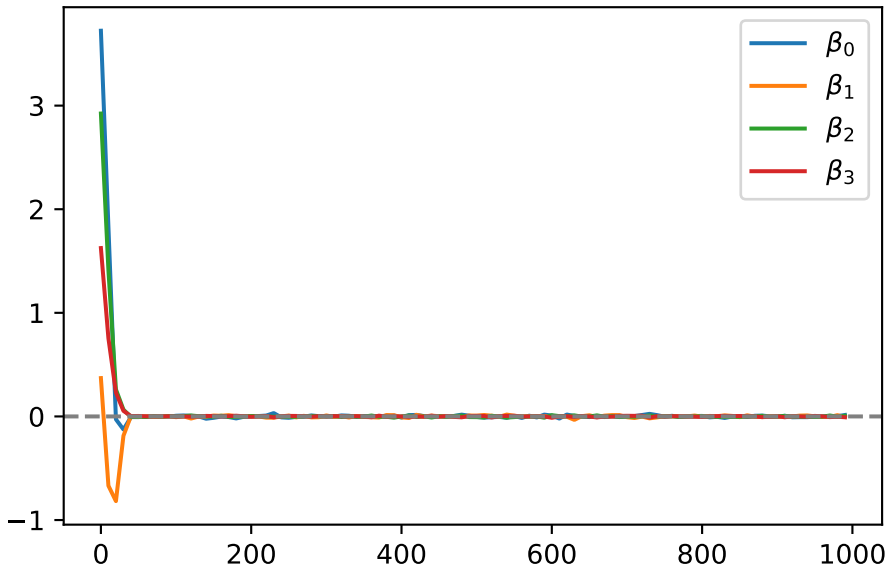
ADAM is considerably faster than just plain momentum when changing the learning rate (which is a bit unfair, since we didn't tweak any momentum parameters ...).

Finally we do the same but for SGD, so we know that that works too (but it's not likely to get stuck in local minima for this simple function).

```
learning_rate = 0.01
n_iter = 1000
grad = grad_OLS()

gd = GradientDescent(learning_rate, grad, optimizer = ADAM())

step_size = 10
n_steps = int(n_iter / step_size)
convergence = np.zeros((n_steps,X.shape[1]))
print(convergence.shape)
iters = np.zeros(n_steps)
for i in range(n_steps):
    gd.descend(X, y, epochs = step_size, batch_size = 5)
    new_coef = gd.theta
    convergence[i, :] = (poly_list.reshape(-1, 1) - new_coef).flatten()
    iters[i] = i*step_size

print(gd.theta)

for i in range(X.shape[1]):
```

```
    plt.plot(iters, convergence[:,i], label=fr"$\beta_{i}$")

plt.legend()
plt.axhline(0, linestyle = "--", color="grey")
plt.show()
```

```
(100, 4)
[[2.98601664]
 [3.00451476]
 [0.99398022]
 [4.00786939]]
```



This also converges really quickly (but keeps moving due to the stochasticity), but again we have tweaked the parameters a little bit. I will tweak more systematically when using more complex data.

Franke function

We generate data using the Franke function, and polynomial features. We found polynomial degree 10 to be the best fit in project 1, so we use that as a starting point, and look at combinations of learning rates and the regularization parameter λ

```
# Make data.
x1 = np.arange(0, 1, 0.05)
x2 = np.arange(0, 1, 0.05)
x1, x2 = np.meshgrid(x1,x2)

z = franke_function(x1, x2) + np.random.normal(0,0.1,x1.shape)

x1x2 = np.column_stack((x1.flatten(), x2.flatten()))
max_poly = 10
X_feat = PolynomialFeatures(max_poly).fit_transform(x1x2)
X = X_feat[:, 1:] # remove intercept
y = z.flatten()

np.random.seed(42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
# reshape y to 2D array
y_test = y_test[:,np.newaxis]
y_train = y_train[:,np.newaxis]

# scale data
scalerX = StandardScaler(with_std=True)
X_train_scaled = scalerX.fit_transform(X_train)
X_test_scaled = scalerX.transform(X_test)

scalery = StandardScaler(with_std=True)
y_train_scaled = scalery.fit_transform(y_train)
y_test_scaled = scalery.transform(y_test)
```

We work with standardized data without intercept, and try to minimize MSE of the test data for different combinations of λ and learning rates.

Learning rates and optimizers

First we try changing learning rate with a λ of 10^{-4}, which we found to be the optimal value in project 1. Trying also with different optimizers.

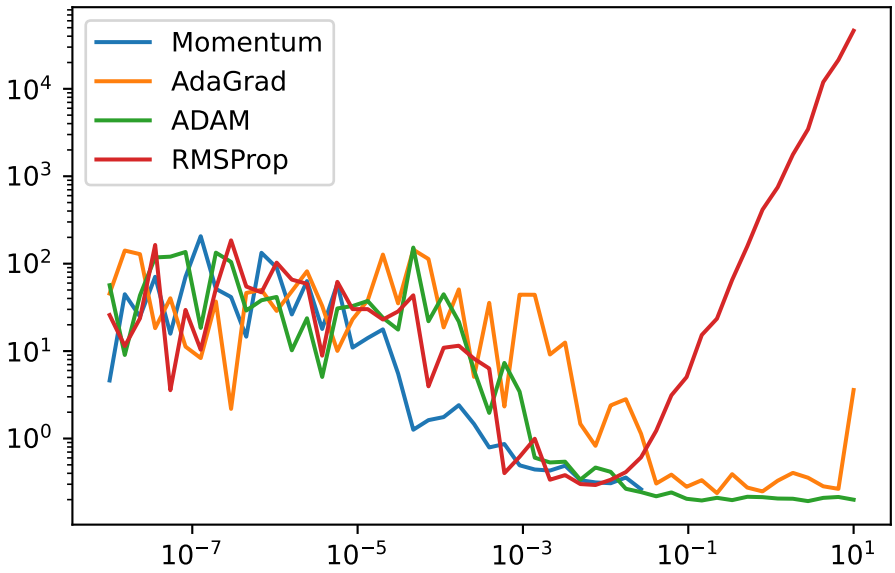
```
learning_rate=0.01
n_iter = 200
lmb=1e-4
grad = grad_ridge(lmb)
grad=grad_OLS()
```

```
gd = GradientDescent(learning_rate, grad)
gd.descend(X_train_scaled, y_train_scaled, n_iter)
```

```
np.random.seed(3489)
learning_rates = np.logspace(-8,1, 50)
optimizers = [Momentum, AdaGrad, ADAM, RMSProp]
n_iter = 1000
lmb=1e-4
grad = grad_ridge(lmb)

mses = np.zeros( (len(learning_rates), len(optimizers)) )
for j, optimizer in enumerate(optimizers):
    for i, learning_rate in enumerate(learning_rates):
        gd = GradientDescent(learning_rate, grad, optimizer=optimizer())
        gd.descend(X_train_scaled, y_train_scaled, n_iter)
        y_pred = X_test_scaled @ gd.theta
        mses[i,j] = mse(y_pred, y_test_scaled)
```

```
/home/fleskelapp/Documents/MachineLearningProjects/FYS-STK4155-Project-2/src/ptwo/gradients.p
  return (2.0/n) * X.T @ (X @ theta - y) + 2*lmb*theta
/home/fleskelapp/Documents/MachineLearningProjects/FYS-STK4155-Project-2/src/ptwo/gradients.p
  return (2.0/n) * X.T @ (X @ theta - y) + 2*lmb*theta
/home/fleskelapp/Documents/MachineLearningProjects/FYS-STK4155-Project-2/src/ptwo/optimizers
  update = learning_rate * grad + update_term
```

```
optimizer_names = ["Momentum", "AdaGrad", "ADAM", "RMSProp"]
for j in range(len(optimizers)):
    plt.plot(learning_rates, mses[:,j], label=optimizer_names[j])
plt.legend()
plt.xscale("log")
plt.yscale("log")
plt.show()
```



Learning rates above 10^{-1} seems to give the best results for AdaGrad and ADAM. Overall, ADAM seems to produce the best results.

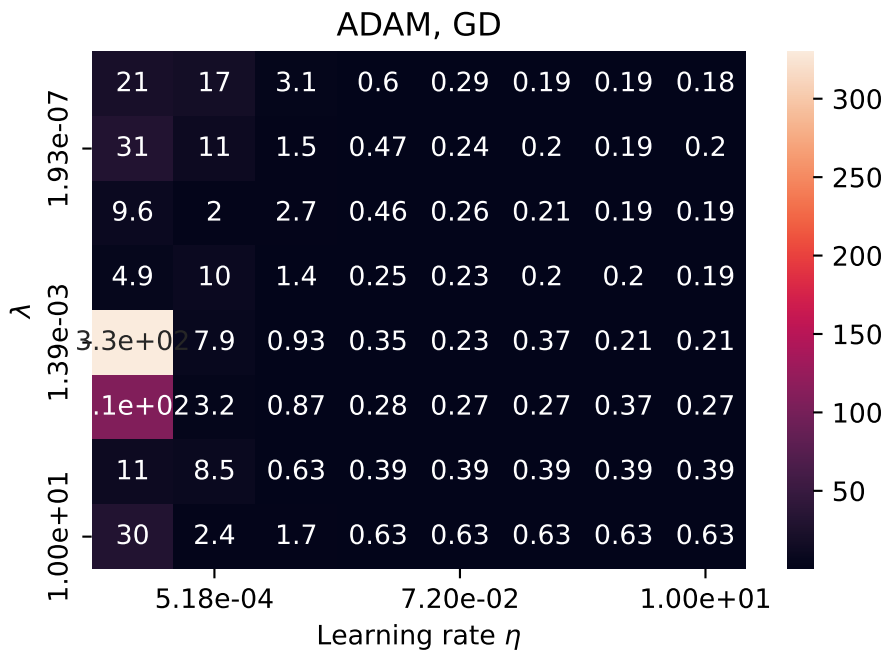
Learning rate and λ grid search

Start with ADAM

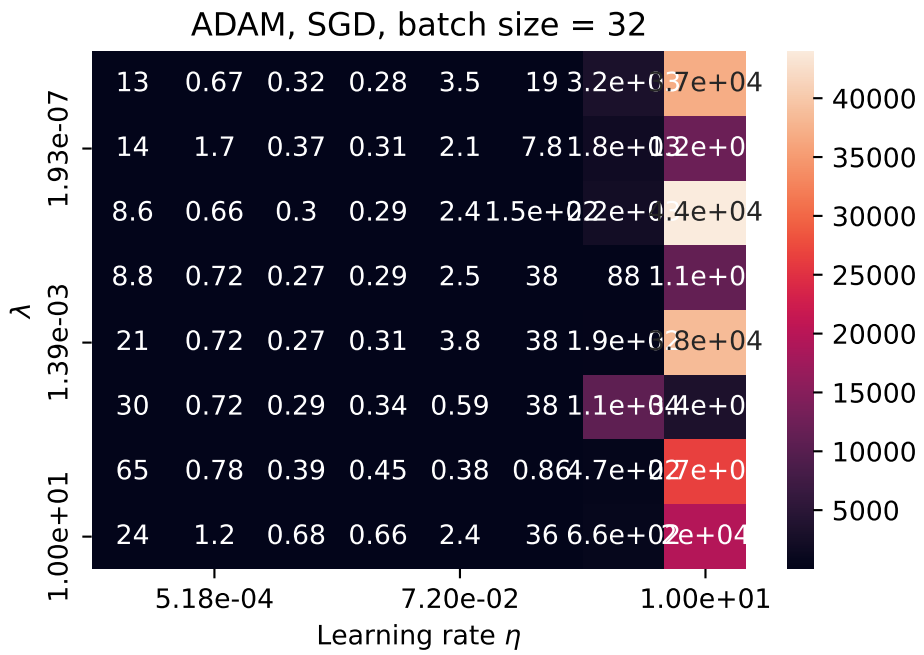
```
def GD_lambda_mse(
    X_train, X_test, y_train, y_test, learning_rate, lmbs, n_iter, batch_size=None,
    optimizer=None, gradient_fun=grad_ridge):
    mses=np.zeros(len(lmbs))
    for i in range(len(lmbs)):
        gd = GradientDescent(learning_rate, gradient_fun(lmbs[i]), optimizer=optimizer)
        gd.descend(X_train, y_train, n_iter, batch_size)
        y_pred = X_test @ gd.theta
        mses[i] = mse(y_test, y_pred)
    return mses

def eta_lambda_grid(
    X_train, X_test, y_train, y_test, learning_rates, lmbs, n_iter, batch_size=None,
    optimizer=None, gradient_fun=grad_ridge
    ):
    mses = np.zeros( (len(lmbs), len(learning_rates)) )
    for i in range(len(learning_rates)):
```

```
        mse_eta = GD_lambda_mse(
            X_train, X_test, y_train, y_test, learning_rate=learning_rates[i],
            lmbs=lmbs, n_iter=n_iter, batch_size=batch_size,
            optimizer=optimizer, gradient_fun=gradient_fun
            )
        mses[:,i] = mse_eta
    return mses

def lambda_lr_heatmap(mses, lmbs, learning_rates,
lmb_label_res=3, lr_label_res=3):
    lmb_lab = ["{0:.2e}".format(x) for x in lmbs]
    lr_lab = ["{0:.2e}".format(x) for x in learning_rates]

    sns.heatmap(mses, annot=True)
    plt.xticks(np.arange(
        len(learning_rates))[1::lmb_label_res] + 0.5, lr_lab[1::lmb_label_res]
        )
    plt.yticks(np.arange(
        len(lmbs))[1::lr_label_res] + 0.5, lmb_lab[1::lr_label_res]
        )
    plt.xlabel(r"Learning rate $\eta$")
    plt.ylabel(r"$\lambda$")
    plt.show()
```

```
np.random.seed(8654)
learning_rates = np.logspace(-4, 1, 8)
n_iter = 1000
lmbs=np.logspace(-8,1, 8)

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer=ADAM()
)
plt.title(fr"ADAM, GD")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```

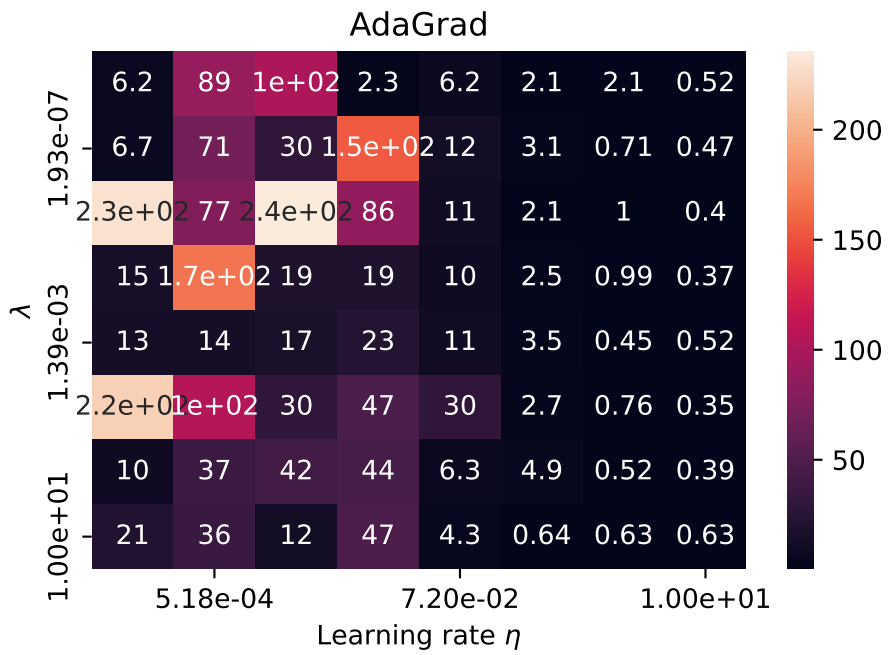



```
np.random.seed(8654)
learning_rates = np.logspace(-4, 1, 8)
n_iter = 200
lmbs=np.logspace(-8,1, 8)
batch_size = 32

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer=ADAM(), batch_size=batch_size
)
plt.title(f"ADAM, SGD, batch size = {batch_size}")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```

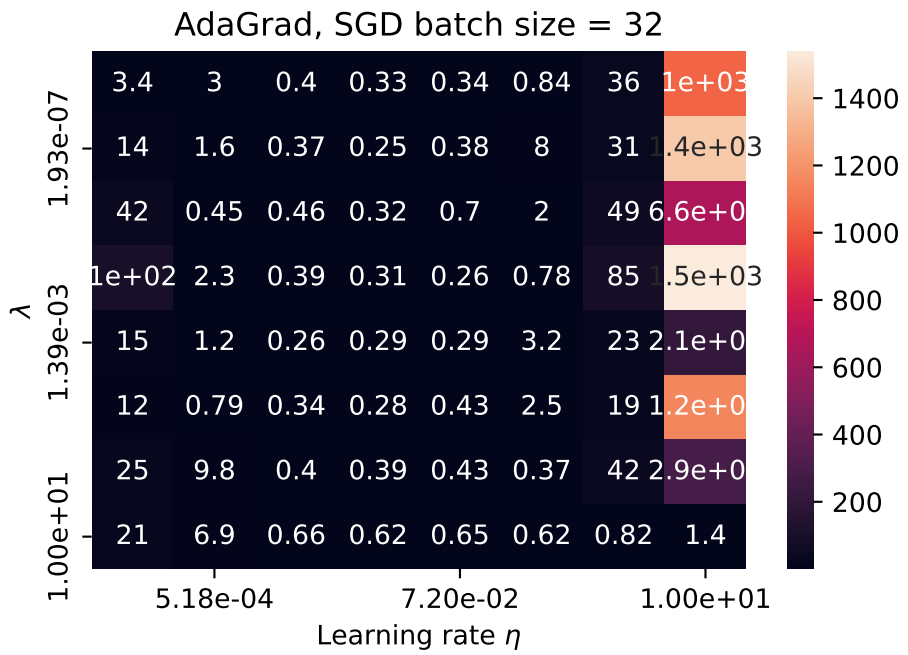


GD and SGD performs roughly the same, but with different optima (min MSE 0.12 vs. 0.13). UPDATE: increasing number of iterations in GD significantly improved performance, GD now outperforms SGD with almost an order of magnitude! UPDATE2: with noise the differences aren't that large anymore

```
np.random.seed(34)
learning_rates = np.logspace(-4,1, 8)
n_iter = 1000
lmbs=np.logspace(-8,1, 8)

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer=AdaGrad()
)

plt.title("AdaGrad")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```

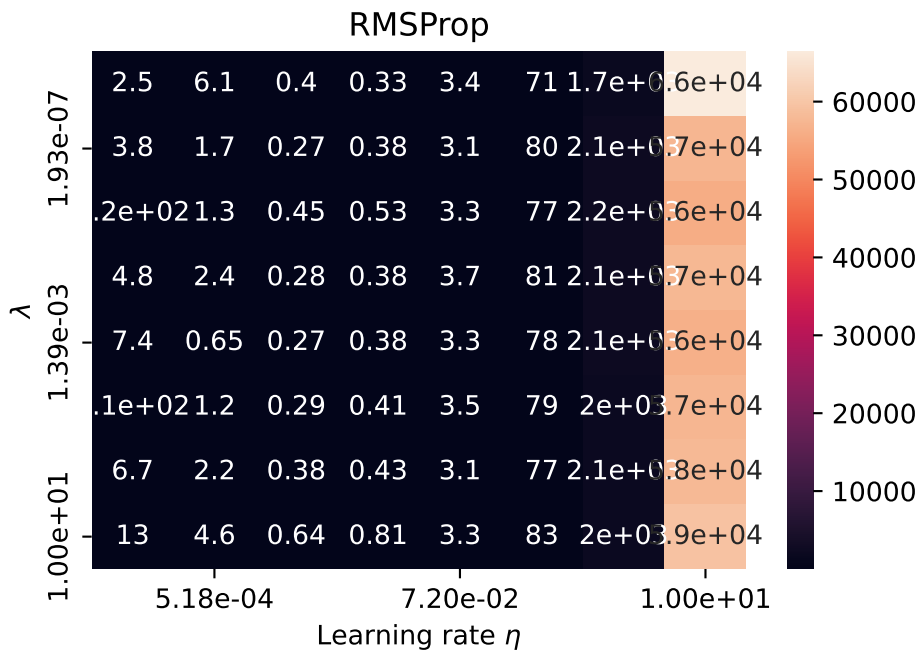



```
np.random.seed(34)
learning_rates = np.logspace(-4,1, 8)
n_iter = 200
lmbs=np.logspace(-8,1, 8)
batch_size = 32

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer=AdaGrad(), batch_size=batch_size
)

plt.title(f"AdaGrad, SGD batch size = {batch_size}")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```

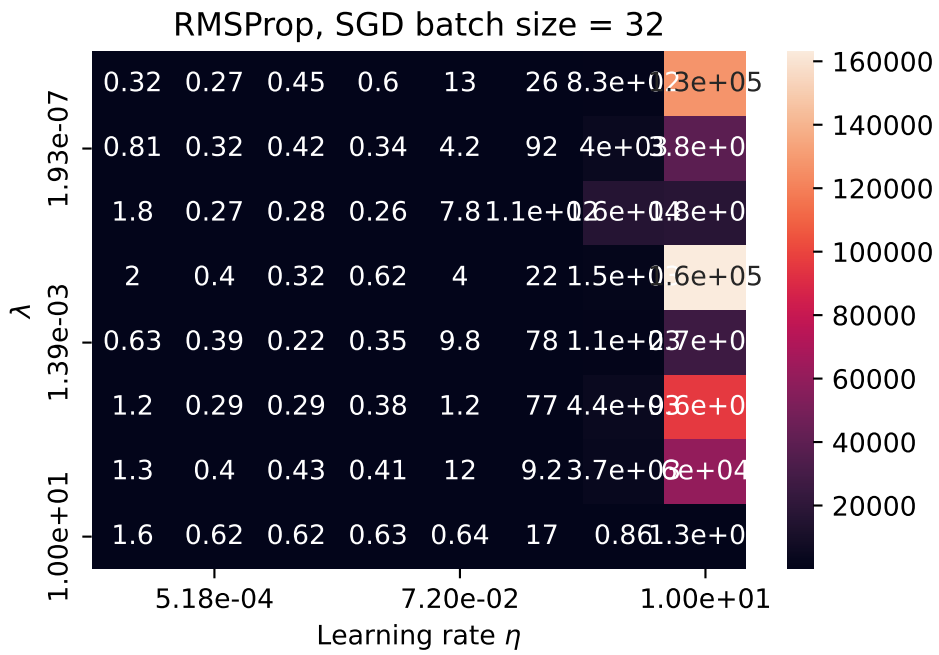


For AdaGrad, SGD performs better than GD, bringing it to around the same performance as ADAM with SGD.

```
np.random.seed(34)
learning_rates = np.logspace(-4,1, 8)
n_iter = 1000
lmbs=np.logspace(-8,1, 8)

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer=RMSProp()
)

plt.title("RMSProp")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```

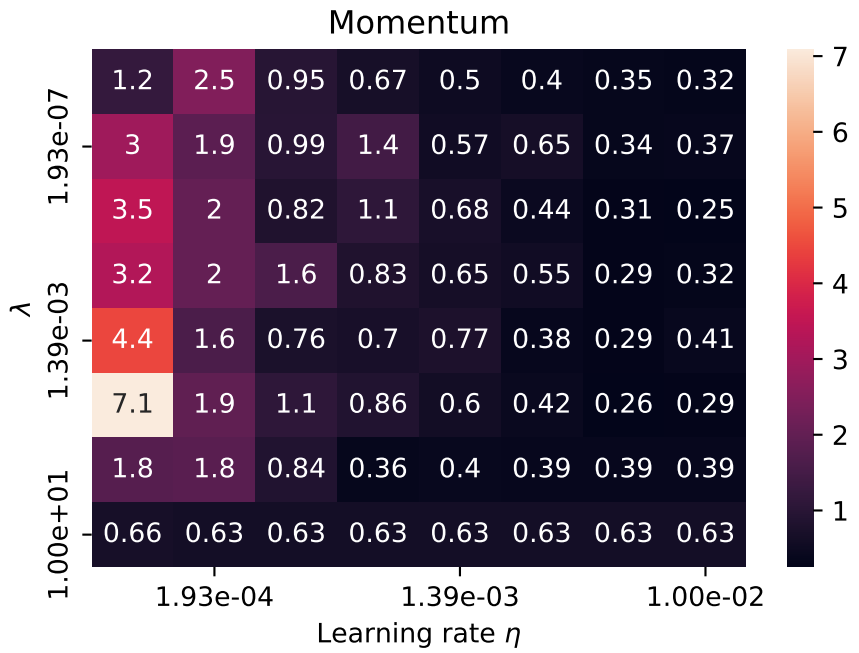



```
np.random.seed(34)
learning_rates = np.logspace(-4,1, 8)
n_iter = 200
lmbs=np.logspace(-8,1, 8)
batch_size = 32

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter,
    optimizer=RMSProp(), batch_size=batch_size
)

plt.title(f"RMSProp, SGD batch size = {batch_size}")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```

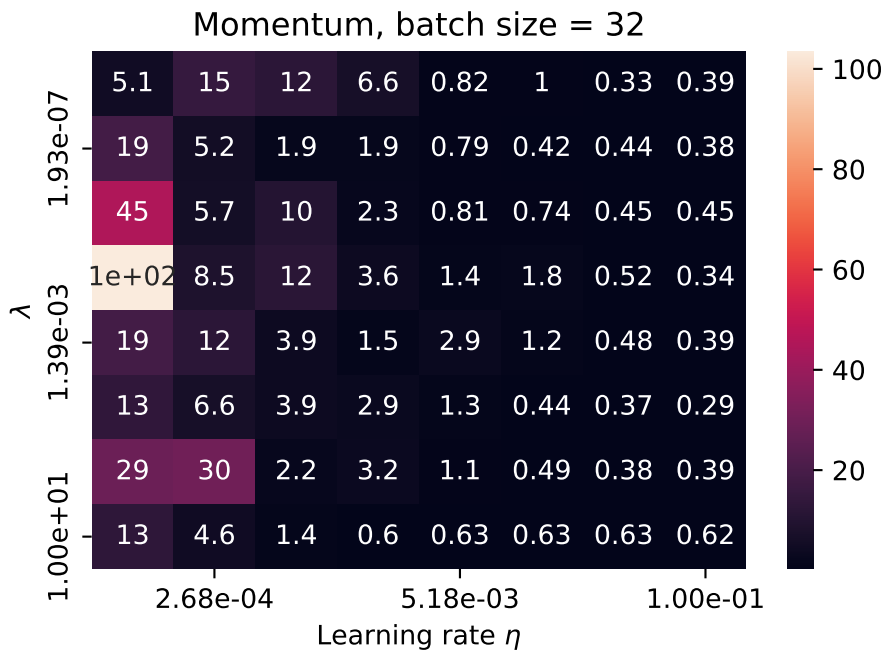


Same for RMSProp as for AdaGrad, SGD outperforms regular GD slightly, and matches ADAM with SGD

```
np.random.seed(34)
learning_rates = np.logspace(-4, -2, 8)
n_iter = 1000
lmbs=np.logspace(-8,1, 8)

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer=Momentum(),
)

plt.title("Momentum")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```




```
np.random.seed(34)
learning_rates = np.logspace(-4,-1, 8)
n_iter = 200
lmbs=np.logspace(-8,1, 8)
batch_size=32

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer=Momentum(), batch_size=batch_size
)

plt.title(f"Momentum, batch size = {batch_size}")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```



Momentum fails above learning rate of approx 10^{-2} for GD and 10^{-1} for SGD. Performance was slightly better for GD.

Terrain data

Import the data from project 1. But scaling with std this time.

```
terrain_full = imread('../../data/SRTM_data_Norway_2.tif')
# subset to a manageable size
terrain1 = terrain_full[1050:1250, 500:700]

x_1d = np.arange(terrain1.shape[1])
y_1d = np.arange(terrain1.shape[0])
# create grid
x_2d, y_2d = np.meshgrid(x_1d,y_1d)

# flatten the data and features
xy = np.column_stack((x_2d.flatten(), y_2d.flatten()))
max_poly = 10
X_feat = PolynomialFeatures(max_poly).fit_transform(xy)
X = X_feat[:, 1:]
```

```
y = terrain1.flatten()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
y_test = y_test[:,np.newaxis]
y_train = y_train[:,np.newaxis]

scalerX = StandardScaler(with_std = True)
X_train_scaled = scalerX.fit_transform(X_train)
X_test_scaled = scalerX.transform(X_test)

scalery = StandardScaler(with_std = True)
y_train_scaled = scalery.fit_transform(y_train)
y_test_scaled = scalery.transform(y_test)
```

```
/tmp/ipykernel_294782/3535801973.py:1: DeprecationWarning: Starting with ImageIO v3 the beha
  terrain_full = imread('../../data/SRTM_data_Norway_2.tif')
```

We start with the optimal parameters we found in project 1, which was polynomial degree 10 and $\lambda = 10^{-4}$

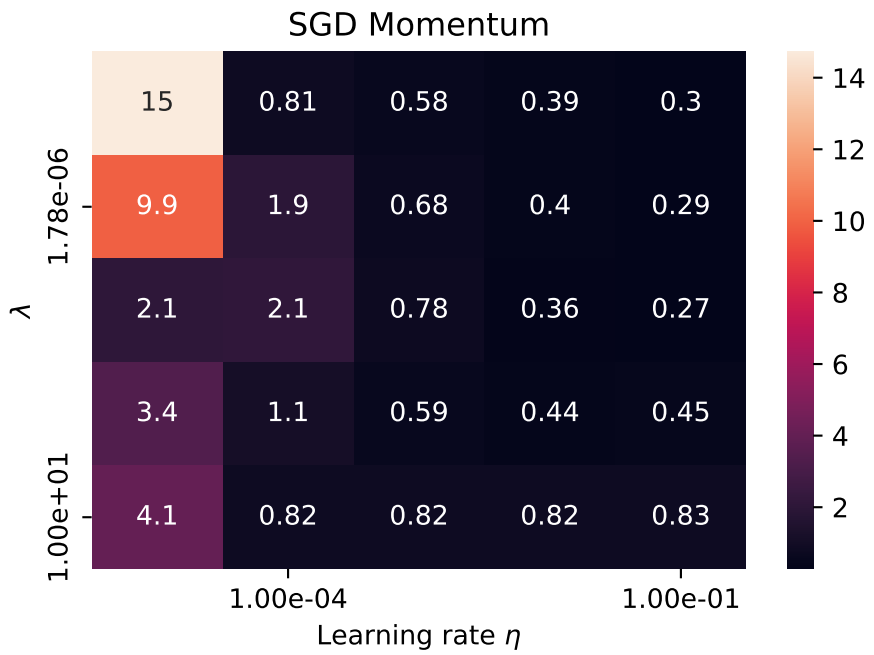
```
learning_rate=0.01
n_iter = 1000
lmb=1e-4
grad = grad_ridge(lmb)
gd = GradientDescent(learning_rate, grad,  optimizer = Momentum())
gd.descend(X_train_scaled, y_train_scaled, n_iter)

pred = X_test_scaled@gd.theta
print(mse(pred, y_test_scaled))
```

```
0.4314283265483981
```

```
np.random.seed(8654)
learning_rates = np.logspace(-4, 1, 5)
n_iter = 50
lmbs=np.logspace(-8,1, 5)

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer = ADAM(), batch_size = 32
```

```
)
plt.title("ADAM")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```

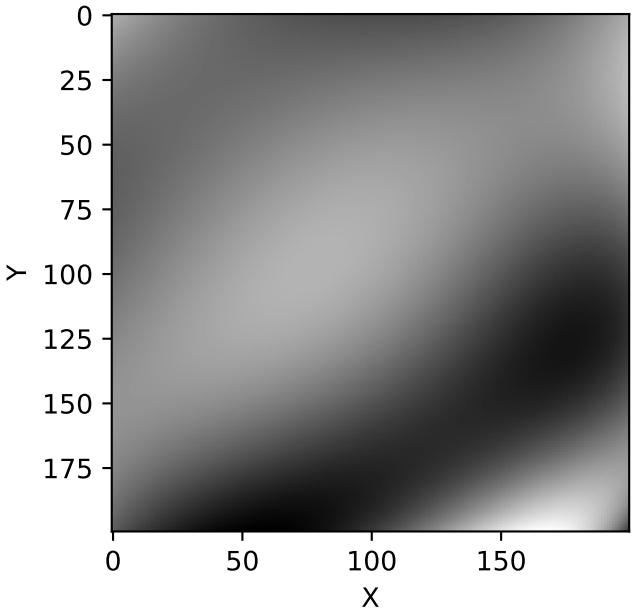

```
np.random.seed(8654)
learning_rates = np.logspace(-5, -1, 5)
n_iter = 50
lmbs=np.logspace(-8,1, 5)

mses = eta_lambda_grid(
    X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled,
    learning_rates, lmbs, n_iter, optimizer = Momentum(0.5), batch_size=32
)
plt.title("SGD Momentum")
lambda_lr_heatmap(mses, lmbs, learning_rates)
```




```
np.random.seed(2309148230)
learning_rate=1.78e-3
n_iter = 1000
lmb=1e-8
grad = grad_ridge(lmb)
#grad = grad_OLS()
gd = GradientDescent(learning_rate, grad,  optimizer = ADAM())
gd.descend(X_train_scaled, y_train_scaled, n_iter, batch_size = 32)
pred = X_test_scaled@gd.theta
print(mse(pred, y_test_scaled))
```

```
0.20814416476497555
```

```
X_scaled = scalerX.fit_transform(X)
pred = X_scaled@gd.theta
y_predict = pred.reshape(200,200)
plt.imshow(y_predict, cmap='gray')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```


0
25
50
75
100
125
150
175
Y
0
50
100
150
X