Optimization - SGD

Lecture 15

Dr. Colin Rundel

Stochastic Gradient Descent

A regression examples

```
from sklearn.datasets import make_regression
X, y, coef = make_regression(
  n_samples=200, n_features=20, n_informative=4,
  bias=3, noise=1, random_state=1234, coef=True
)
```

```
y
```

```
array([
-36.2252,
9.6357,
66.4583,
48.9574,
24.1885,
-13.2444,

18.1455,
-135.047 ,
116.5772,
60.2524,
30.9319,
107.148 ,

21.6209,
66.2401,
-132.8878,
58.636 ,
```

```
X
```

```
array([[-0.6465,  2.0803,  0.1412, -0.8419, -0.1595,  1.3321, -0.4262,
        -0.0351, -0.1938, -0.6093, -0.3433,  0.6126,  0.3777, -1.2062,
        -0.2277, -0.8896, -0.4674, -1.3566,  1.4989, -0.7468],
       [-0.3834, -0.3631, -1.2196,  0.6   ,  0.3315,  1.1056,  0.2662,
        -0.7239,  0.0259, -0.2172, -0.6841,  0.0991,  0.2794, -1.208 ,
        -0.7818, -1.7348, -1.3397, -0.5723, -0.5882,  0.2717],
       [-0.1637, -0.8118,  0.9551,  0.5711,  0.8719, -0.9619,  1.9846,
        -1.1806, -1.1261,  0.297 ,  1.2499,  0.7109, -0.1183,  0.6708,
         0.6895,  1.4705,  0.0634, -0.3079, -2.2512, -0.0216],
       [-0.9292, -0.4897, -2.1196, -1.142 ,  1.266 , -0.2988,  1.0016,
        -2.1969, -1.0739, -0.1149,  0.5122,  0.302 , -0.0974,  1.3461,
         0.1909,  1.1223,  0.6268,  2.2035, -0.5135,  2.0118],
       [ 0.1645, -0.5847,  0.2708, -3.5635,  0.1526,  0.5283,  0.7674,
         1.392 , -0.0819,  1.3211,  0.4644, -1.0279,  0.9849, -1.069 ,
        -0.4301,  0.0798, -0.5119, -0.3448,  0.8166, -0.4   ],
       [ 0.4134,  1.9511, -0.5013, -1.4894,  0.4191, -1.4104,  0.2617,
        -0.6981,  0.0368, -1.151 ,  2.0752,  0.5001, -0.2428,  0.45  ,
         0.7176,  1.3846,  0.5155,  0.4459, -0.2784, -0.2864],
       [-0.0628, -1.424 , -1.1023,  0.1445, -0.4836,  1.4795, -0.5921,
```

```
coef
```

```
array([ 0.
,  0.    ,  0.
,  9.6106,
43.4239,  0.
,  0.    ,
        0.
,  0.    ,
34.453 ,
9.2929,  0.
,  0.    ,  0.
,
        0.
,  0.    ,  0.
,  0.    ,  0.
,  0.    ])
```

Minimalistic GD for LR

```python
def grad_desc_lm(X, y, beta, step, max_step=50):
  X = jnp.c_[jnp.ones(X.shape[0]), X]
  f = lambda beta: jnp.sum((y - X @ beta)**2)
  grad = jax.grad(f)

  res = {"x": [beta], "loss": [f(beta).item()], "iter": [0]}

  for i in range(max_step):
    beta = beta - grad(beta) * step
    res["x"].append(beta)
    res["loss"].append(f(beta).item())
    res["iter"].append(res["iter"][-1]+1)

  return res
```

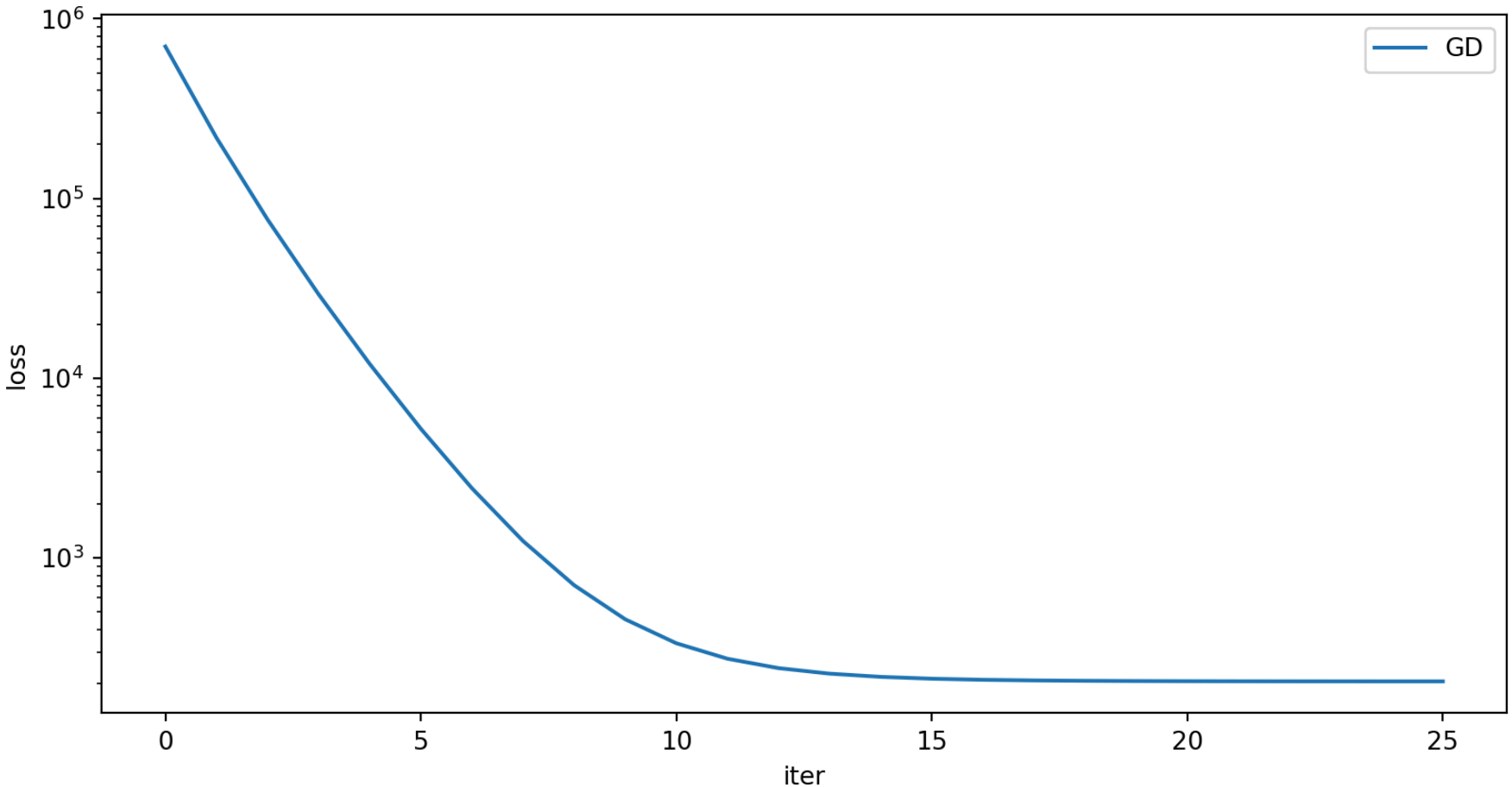
Linear regression

```
lm = LinearRegression().fit(X,y)
np.r_[lm.intercept_, lm.coef_]
```

```
array([ 3.0616, -0.0121, -0.0096,  0.096 ,  9.6955, 43.406 ,  0.0253,
        0.0284,  0.0962,  0.1069, 34.4884,  9.3445, -0.0165, -0.0147,
       -0.0396,  0.0969, -0.1057, -0.0943,  0.11  , -0.0096, -0.0875])
```

```
gd_lm = grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.001, max_step=25
)
gd_lm["x"][-1]
```

```
Array([ 3.0634, -0.0118, -0.0136,  0.097 ,  9.7003, 43.4074,  0.0301,
        0.0292,  0.0976,  0.1007, 34.479 ,  9.3418, -0.0197, -0.0123,
       -0.0427,  0.0878, -0.106 , -0.087 ,  0.1106, -0.018 , -0.0955],      dtype=float64)
```


GD
loss
iter
10^6
10^5
10^4
10^3
0
5
10
15
20
25

A quick analysis

Lets take a quick look at the linear regression loss function and gradient descent and think a bit about its cost(s), we can define the loss function and its gradient as follows:

$$f(\boldsymbol{\beta}) = (y - \boldsymbol{X}\boldsymbol{\beta})^T(y - \boldsymbol{X}\boldsymbol{\beta})$$

$$\nabla f(\boldsymbol{\beta}) = 2\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{y})$$

What are the costs of calculating the loss function and gradient respectively in terms of n and k?

- Calculating the loss costs $O(nk)$
- Calculating the gradient costs $O(n^2k)$

Stochastic Gradient Descent

This is a variant of gradient descent where rather than using all n data points we randomly sample one at a time and use that single point to make our gradient step.

- Sampling of observations can be done with or without replacement
- Will take more steps to converge but each step is now cheaper to compute
- SGD has slower asymptotic convergence than GD, but is often faster in practice
- Generally requires the learning rate to shrink as a function of iteration to guarantee convergence

SGD - Linear Regression

```python
def sto_grad_desc_lm(X, y, beta, step, max_step=50, seed=1234, replace=True):
  X = jnp.c_[jnp.ones(X.shape[0]), X]
  f = lambda beta: jnp.sum((y - X @ beta)**2)
  grad = lambda beta, i: 2*X[i,:] * (X[i,:]@beta - y[i])
  n, k = X.shape

  res = {"x": [beta], "loss": [f(beta).item()], "iter": [0]}
  rng = np.random.default_rng(seed)

  for i in range(max_step):
    if replace:
      js = rng.integers(0,n,n)
    else:
      js = np.array(range(n))
      rng.shuffle(js)

    for j in js:
      beta = beta - grad(beta, j) * step
      res["x"].append(beta)
      res["loss"].append(f(beta).item())
      res["iter"].append(res["iter"][-1]+1)

  return res
```

Fitting

```
1 np.r_[lm.intercept_, lm.coef_]
```

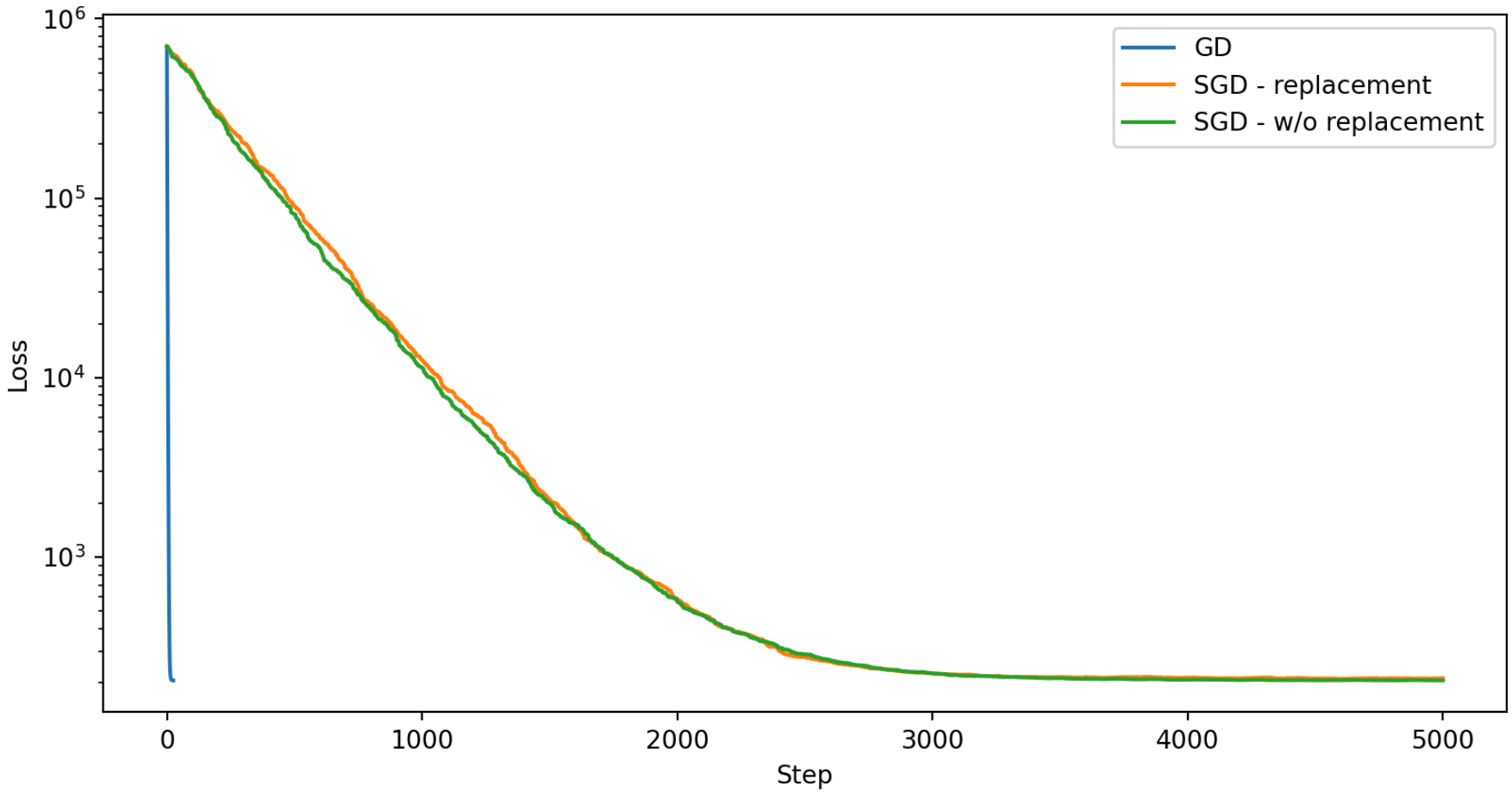
```
array([ 3.0616, -0.0121, -0.0096,  0.096 ,  9.6955, 43.406 ,  0.0253,
        0.0284,  0.0962,  0.1069, 34.4884,  9.3445, -0.0165, -0.0147,
       -0.0396,  0.0969, -0.1057, -0.0943,  0.11  , -0.0096, -0.0875])
```

```
1 sgd_lm_rep = sto_grad_desc_lm(
2   X, y, np.zeros(X.shape[1]+1),
3   step = 0.001, max_step=25, replace=True
4 )
5 sgd_lm_rep["x"][-1]
```

```
Array([ 3.0342, -0.0896, -0.0556,  0.045 ,  9.718 , 43.423 ,  0.0448,
        0.0839,  0.0512,  0.1082, 34.4395,  9.311 ,  0.0179,  0.035 ,
        0.0028,  0.1128, -0.0999, -0.123 ,  0.1248,  0.0134, -0.1609],      dtype=float64)
```

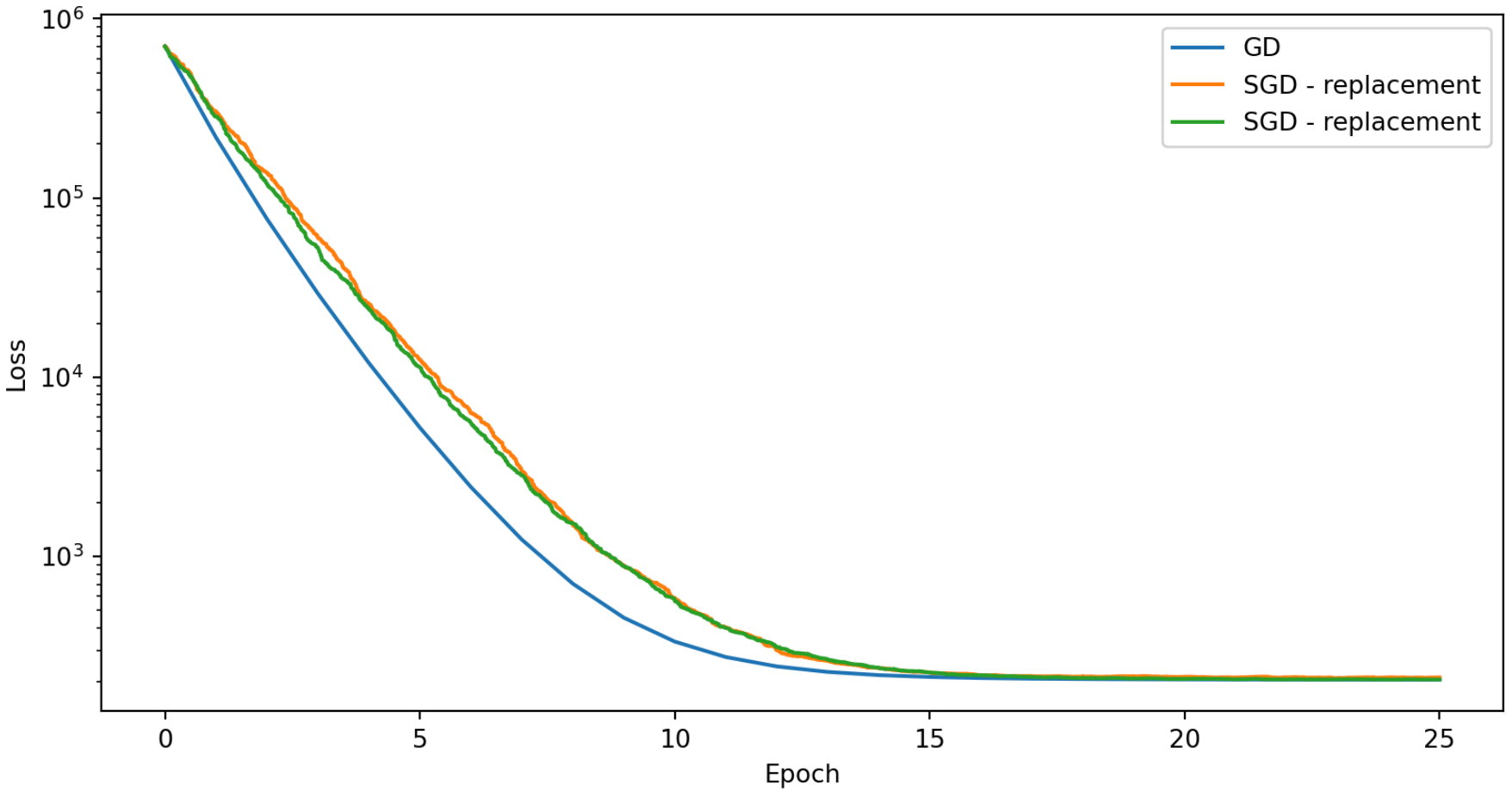
```
1 sgd_lm_worep = sto_grad_desc_lm(
2   X, y, np.zeros(X.shape[1]+1),
3   step = 0.001, max_step=25, replace=False
4 )
5 sgd_lm_worep["x"][-1]
```

```
Array([ 3.065 , -0.0102, -0.0065,  0.0996,  9.7028, 43.4078,  0.0298,
        0.0349,  0.0952,  0.0933, 34.4662,  9.336 , -0.0252, -0.0115,
       -0.0462,  0.0774, -0.1103, -0.0746,  0.1096, -0.0269, -0.0995],      dtype=float64)
```


GD
SGD - replacement
SGD - w/o replacement
Loss
Step
10^6
10^5
10^4
10^3
0
1000
2000
3000
4000
5000

Using Epochs

Generally, rather than thinking in steps we use epochs instead - an epoch is one complete pass through the data.



A bigger example

```
from sklearn.datasets import make_regression
X, y, coef = make_regression(
  n_samples=10000, n_features=20, n_informative=4,
  bias=3, noise=1, random_state=1234, coef=True
)
```

```
lm = LinearRegression().fit(X,y)
np.r_[lm.intercept_, lm.coef_]
```

```
array([ 3.0081,  0.0088,  0.0002,  0.0021,  0.0037,  0.0033,  0.026 ,
       -0.0006,  0.0005, 12.2771, 44.4939,  3.6423,  0.0168, 61.3938,
       -0.0012, -0.0056,  0.014 , -0.0093, -0.0056,  0.0024,  0.0217])
```

Fitting

```
gd_lm = grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.00005, max_step=3
)
gd_lm["x"][-1]
```

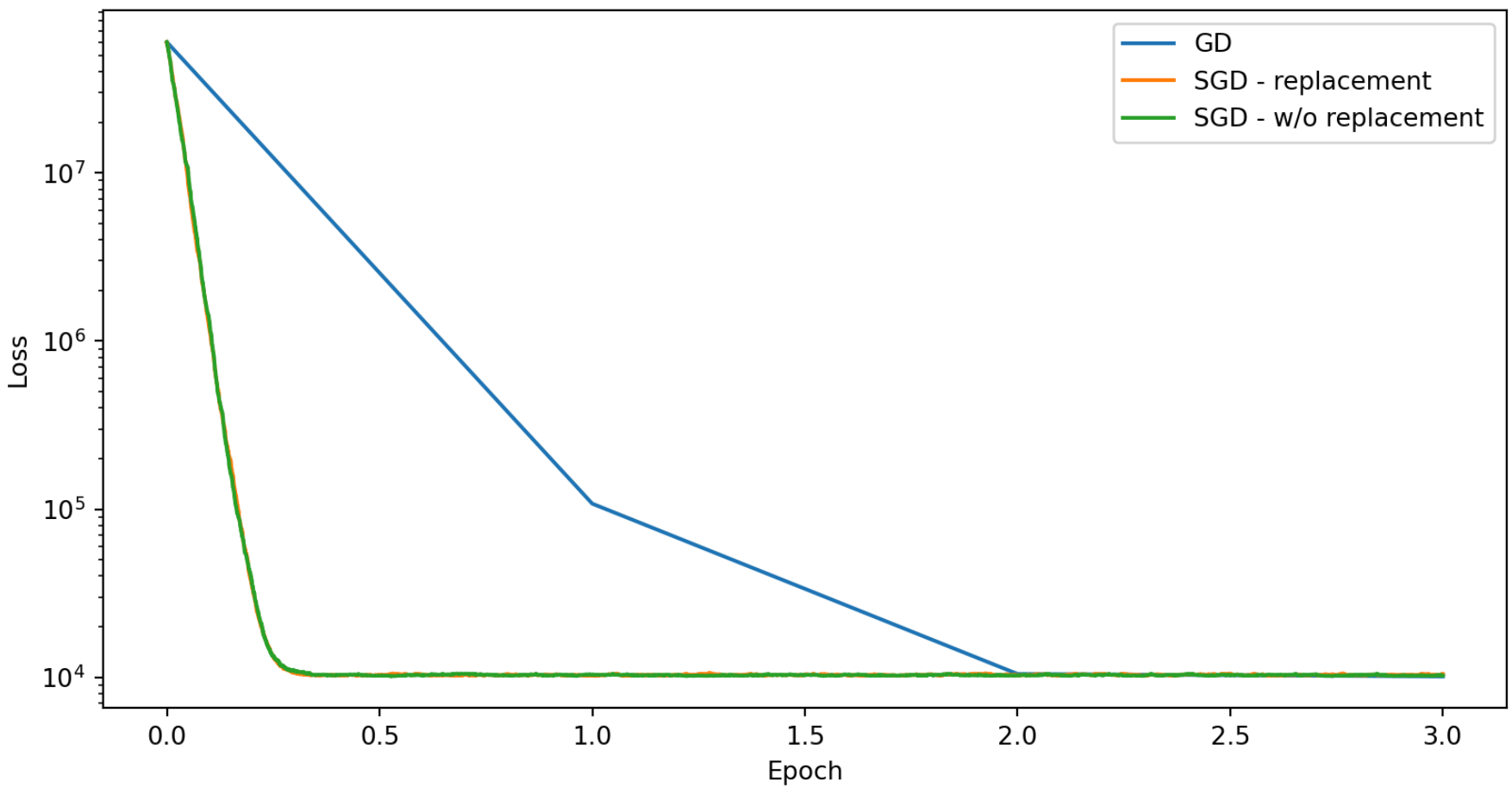
```
Array([ 3.01  ,  0.0118,  0.0029,  0.0033, -0.0014,  0.0028,  0.0252,
       -0.0005,  0.0009, 12.2793, 44.4961,  3.6409,  0.0165, 61.3964,
        0.0011,  0.0005,  0.011 , -0.0134, -0.0045,  0.0028,  0.0244],      dtype=float64)
```

```
sgd_lm_rep = sto_grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.001, max_step=3, replace=True
)
sgd_lm_rep["x"][-1]
```

```
Array([ 2.9968,  0.0452,  0.0346,  0.0205, -0.0286, -0.0518, -0.0534,
       -0.0142,  0.0371, 12.2115, 44.5628,  3.6736,  0.0249, 61.4223,
       -0.0094, -0.0877,  0.0134, -0.0324, -0.0178,  0.0222,  0.0089],      dtype=float64)
```

```
sgd_lm_worep = sto_grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.001, max_step=3, replace=False
)
sgd_lm_worep["x"][-1]
```

```
Array([ 2.9756,  0.0149, -0.0011, -0.0047,  0.011 , -0.0176,  0.0251,
       -0.0244, -0.0328, 12.2937, 44.5281,  3.6333, -0.0158, 61.4162,
        0.0541,  0.0064, -0.0231, -0.0014, -0.0144, -0.0299,  0.0141],      dtype=float64)
```


GD
SGD - replacement
SGD - w/o replacement
Loss
10^7
10^6
10^5
10^4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Epoch

Mini batch gradient descent

This is a further variant of stochastic gradient descent where a mini batch of m data points is selected for each gradient update,

- The idea is to find a balance between the cost of increasing the data size vs the speed-up of vectorized calculations.
- More updates per epoch than GD, but less than SGD

MBGD - Linear Regression

```
def mb_grad_desc_lm(X, y, beta, step, batch_size = 10, max_step=50, seed=1234, replace=True):
  X = jnp.c_[jnp.ones(X.shape[0]), X]
  f = lambda beta: jnp.sum((y - X @ beta)**2)
  grad = lambda beta, i: 2*X[i,:].T @ (X[i,:]@beta - y[i])
  n, k = X.shape

  res = {"x": [beta], "loss": [f(beta).item()], "iter": [0]}
  rng = np.random.default_rng(seed)

  for i in range(max_step):
    if replace:
      js = rng.integers(0,n,n)
    else:
      js = np.array(range(n))
      rng.shuffle(js)

    for j in js.reshape(-1, batch_size):
      beta = beta - grad(beta, j) * step
      res["x"].append(beta)
      res["loss"].append(f(beta).item())
      res["iter"].append(res["iter"][-1]+1)

  return res
```

Fitting

```
lm = LinearRegression().fit(X,y)
np.r_[lm.intercept_, lm.coef_]
```

```
array([ 3.0081,  0.0088,  0.0002,  0.0021,  0.0037,  0.0033,  0.026 ,
       -0.0006,  0.0005, 12.2771, 44.4939,  3.6423,  0.0168, 61.3938,
       -0.0012, -0.0056,  0.014 , -0.0093, -0.0056,  0.0024,  0.0217])
```

```
sizes = [10,50,100]
mbgd = { size: mb_grad_desc_lm(
            X, y, np.zeros(X.shape[1]+1), batch_size=size,
            step = 0.001, max_step=3, replace=False
          )
          for size in sizes }
```

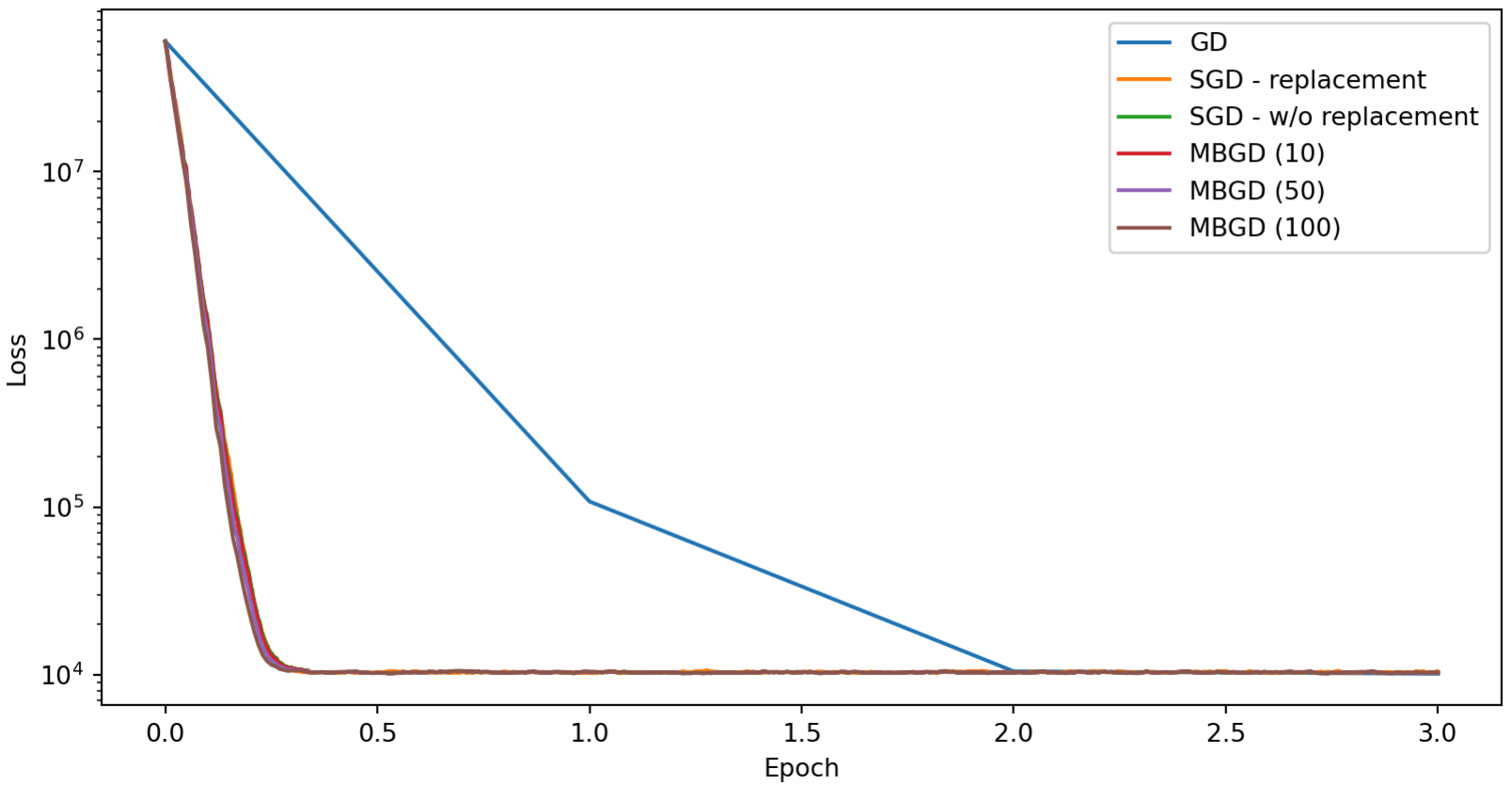
```
Batch size: 10
[ 2.9754  0.0154  0.0004 -0.0038  0.0118 -0.0171  0.0248 -0.0242 -0.0336
 12.2937 44.5285  3.6334 -0.0156 61.417   0.0546  0.0075 -0.023  -0.0004
 -0.0135 -0.031   0.0138]

Batch size: 50
[ 2.9761  0.0107 -0.001  -0.0029  0.0119 -0.0161  0.0238 -0.0243 -0.0374
 12.2943 44.5304  3.6337 -0.0172 61.4199  0.0557  0.0068 -0.0246 -0.0015
 -0.0134 -0.0326  0.0127]

Batch size: 100
[ 2.973   0.0094 -0.0001 -0.0038  0.0123 -0.0171  0.0223 -0.0263 -0.0416
 12.2923 44.5302  3.635  -0.0155 61.4176  0.0565  0.009  -0.0258 -0.002
 -0.014  -0.0353  0.0045]
```

Results

Full | Zoom



A bit of theory

We've talked a bit about the computational side of things, but why do these approaches work at all?

In statistics and machine learning many of our problems have a form that looks like,

$$\arg\min_{\theta} \ \ell(\boldsymbol{X}, \theta) = \arg\min_{\theta} \ \frac{1}{n} \sum_{i=1}^{n} \ell(\boldsymbol{X}_i, \theta)$$

which means that the gradient of the loss function is given by

$$\nabla \ell(\boldsymbol{X}, \theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla \ell(\boldsymbol{X}_i, \theta)$$

$$\nabla \ell(\boldsymbol{X}, \theta) \approx \frac{1}{|B|} \sum_{i \in B}^{n} \nabla \ell(\boldsymbol{X}_i, \theta)$$

SGD estimator

Because we are sampling B randomly, then our SGD and mini batch GD approximations are unbiased estimated of the full gradient,

$$E\left[\frac{1}{|B|}\sum_{i\in B}^{n}\nabla\ell(\boldsymbol{X}_i,\theta)\right]=\frac{1}{n}\sum_{i=1}^{n}\nabla\ell(\boldsymbol{X}_i,\theta)=\nabla\ell(\boldsymbol{X},\theta)$$

Each update can be viewed as a noisy gradient descent step (gradient + zero mean noise).

- The difference between mini batch and stochastic gradient descent is that by increasing the computation cost per step we are reducing the noise variance for that step

Limitations

As mentioned previously we need to be a bit careful with learning rates and convergence for both of these methods. So far, our approach has been naive and runs for a fixed number of epochs.

If we want to use a convergence criterion we need to keep the following in mind:

- Let θ^* be a global / local minimizer of our loss function $\ell(\boldsymbol{X}, \theta)$, then by definition $\nabla\ell(\boldsymbol{X}, \theta^*) = 0$
- The issue is that our gradient approximation,

$$\frac{1}{|B|}\sum_{i\in B}^{n}\nabla\ell(\boldsymbol{X}_i, \theta) \neq 0$$

  as B is a subset of the data, therefore our algorithm will keep taking steps / never converge.

Solution

The practical solution to this is to implement a learning rate schedule which generally shrink the learning rate / step size over time to ensure convergence.

The choice of the exact learning schedule is problem specific, and is usually about finding the balance of how quickly to shrink the step size.

Some common examples:

- Piecewise constant - $\eta_t = \eta_i \text{ if } t_i \leq t \leq t_{i+1}$
- Exponential decay - $\eta_t = \eta_0 e^{-\lambda t}$
- Polynomial decay - $\eta_t = \eta_0 (\beta t + 1)^{-\alpha}$

There are many more approaches including more exotic techniques that allow the learning rate to increase and decrease to help the optimizer better explore the objective function and in some cases escape local optima.

Adaptive updates & Momentum

AdaGrad

This approach was proposed in by Duchi, Hazan, & Singer in 2011 and is based on the idea of scaling the learning rates for the current step by the sum of the square gradients of previous steps - this has the effect of shrinking the step size of dimensions with large previous gradients.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \frac{1}{\sqrt{\boldsymbol{s}_t + \epsilon}} \odot \nabla \ell(\boldsymbol{X}, \boldsymbol{\theta}_t)$$

$$\boldsymbol{s}_t = \sum_{i=1}^{t} \left(\nabla \ell(\boldsymbol{X}, \boldsymbol{\theta}_i)\right)^2$$

where ϵ is a small constant (i.e. 10^{-8}) to avoid division by zero.

Implementation

```
def adagrad_lm(X, y, beta, step, batch_size = 10, max_step=50, seed=1234, replace=True, eps=1
  X = jnp.c_[jnp.ones(X.shape[0]), X]
  f = lambda beta: jnp.sum((y - X @ beta)**2)
  grad = lambda beta, i: 2*X[i,:].T @ (X[i,:]@beta - y[i])
  n, k = X.shape

  res = {"x": [beta], "loss": [f(beta).item()], "iter": [0]}
  rng = np.random.default_rng(seed)

  S = np.zeros(k)

  for i in range(max_step):
    if replace:
      js = rng.integers(0,n,n)
    else:
      js = np.array(range(n))
      rng.shuffle(js)

    for j in js.reshape(-1, batch_size):
      G = grad(beta, j)
      S += G**2

      beta = beta - step * (1/np.sqrt(S + eps)) * G

```

A medium example

```
from sklearn.datasets import make_regression
X, y, coef = make_regression(
  n_samples=1000, n_features=20, n_informative=4,
  bias=3, noise=1, random_state=1234, coef=True
)
```

```
lm = LinearRegression().fit(X,y)
np.r_[lm.intercept_, lm.coef_]
```

```
array([ 3.0034,  0.0144,  0.0304, -0.0306,  0.0334,  0.0292, -0.0214,
       30.9685,  0.0189, -0.005 ,  0.005 ,  0.0016, 40.5613, -0.0422,
       44.6158, -0.0495, 72.2522,  0.002 , -0.0336,  0.0148,  0.0516])
```

Fitting

```python
sizes = [1, 25, 50, 1000]
lrs = [10] * 4
algos = ["AdaGrad - SGD", "AdaGrad - MBGD (25)", "AdaGrad - MBGD (50)", "AdaGrad - GD"]

adagrad = { size: adagrad_lm(
                    X, y, np.zeros(X.shape[1]+1), batch_size=size,
                    step = lr, max_step=15, replace=False
                  )
            for size, lr in zip(sizes,lrs) }
```

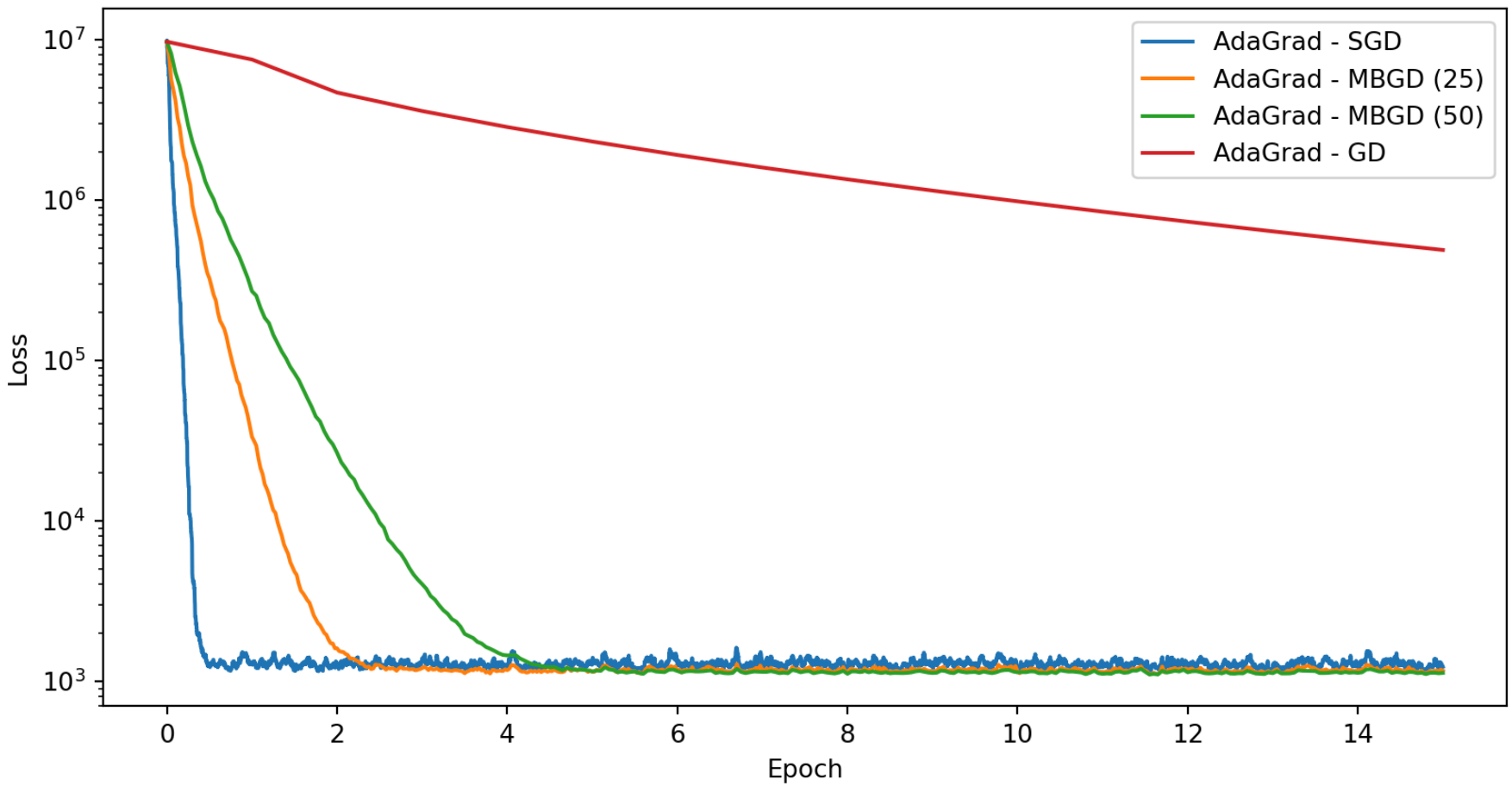
```
AdaGrad - SGD
[ 3.0218  0.0577  0.0304  0.0376 -0.0169 -0.0808 -0.0156 31.0294  0.0226
  0.0419 -0.0442  0.1085 40.7294  0.0879 44.675  -0.246  72.3294  0.0787
 -0.0684  0.0128  0.0556]

AdaGrad - MBGD (25)
[ 3.0464  0.1249  0.0086  0.0118 -0.0177 -0.0265 -0.0327 30.9548 -0.0094
  0.0146  0.0122  0.1039 40.6525  0.0591 44.65   -0.1758 72.27    0.0771
 -0.0531 -0.0165  0.0682]

AdaGrad - MBGD (50)
[ 3.0341  0.1253  0.0233 -0.0095  0.0094 -0.0057 -0.0136 30.9632 -0.0047
 -0.0028  0.0226  0.0697 40.6104  0.024  44.638  -0.1379 72.2525  0.0463
 -0.0467 -0.0267  0.0971]

AdaGrad - GD
[ 2.5491  0.1287 -1.0413  1.0849  0.2096  0.9691 -0.342  31.1497 -0.4613
  0.8328  0.0183  1.3411 38.2487 -0.4805 40.5215  0.522  49.9215 -0.6421
 -0.1804 -1.0691 -0.8411]
```

Results


AdaGrad - SGD
AdaGrad - MBGD (25)
AdaGrad - MBGD (50)
AdaGrad - GD
Loss
Epoch
10^7
10^6
10^5
10^4
10^3
0
2
4
6
8
10
12
14

RMSProp

With AdaGrad the denominator involving $\boldsymbol{s}_t$ gets larger as t increases, but in some cases it gets too large too fast to effectively explore the loss function. An alternative is to use a moving average of the past squared gradients instead.

RMSProp replaces AdaGrad's $\boldsymbol{s}_t$ with the following,

$$\boldsymbol{s}_t = \beta\, \boldsymbol{s}_{t-1} + (1-\beta)\left(\nabla \ell(\boldsymbol{X}, \boldsymbol{\theta}_t)\right)^2$$
$$\boldsymbol{s}_0 = \mathbf{0}$$

in practice a value of $\beta \approx 0.9$ is used.

Implementation

```
def rmsprop_lm(X, y, beta, step, batch_size = 10, max_step=50, seed=1234, replace=True, eps=1
  X = jnp.c_[jnp.ones(X.shape[0]), X]
  f = lambda beta: jnp.sum((y - X @ beta)**2)
  grad = lambda beta, i: 2*X[i,:].T @ (X[i,:]@beta - y[i])
  n, k = X.shape

  res = {"x": [beta], "loss": [f(beta).item()], "iter": [0]}
  rng = np.random.default_rng(seed)

  S = np.zeros(k)

  for i in range(max_step):
    if replace:
      js = rng.integers(0,n,n)
    else:
      js = np.array(range(n))
      rng.shuffle(js)

    for j in js.reshape(-1, batch_size):
      G = grad(beta, j)
      S = b*S + (1-b) * G**2

      beta = beta - step * (1/np.sqrt(S + eps)) * G

```

Fitting

```python
sizes = [1, 25, 50, 1000]
lrs = [0.01, 0.1, 0.25, 1]
algos = ["RMSProp - SGD", "RMSProp - MBGD (25)", "RMSProp - MBGD (50)", "RMSProp - GD"]

rmsprop = { size: rmsprop_lm(
                    X, y, np.zeros(X.shape[1]+1), batch_size=size,
                    step = lr, max_step=25, replace=False
                  )
            for size, lr in zip(sizes,lrs) }
```

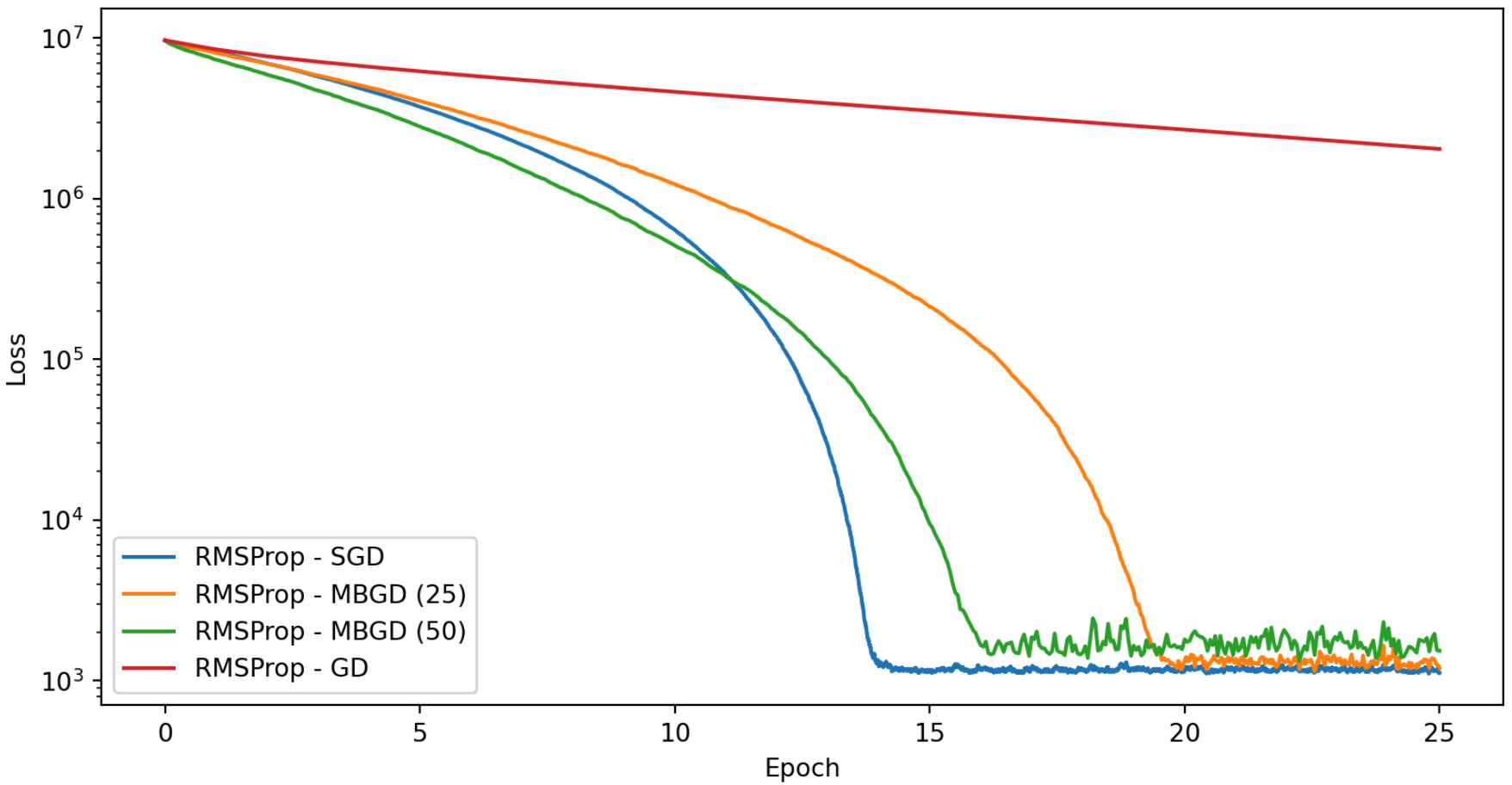
```
RMSProp - SGD
[ 3.0721  0.0241 -0.0229 -0.0109  0.0477  0.0879 -0.019  30.968   0.063
 -0.0005  0.0926 -0.0024 40.5439 -0.0316 44.6041 -0.0575 72.2169 -0.0054
 -0.0461 -0.0527  0.0905]

RMSProp - MBGD (25)
[ 3.148   0.0316 -0.1395 -0.0606  0.0544  0.0715 -0.0488 30.9947  0.0788
 -0.0545 -0.0162  0.0012 40.4057 -0.0185 44.6314 -0.1133 72.2111 -0.0824
 -0.0153 -0.0569  0.117 ]

RMSProp - MBGD (50)
[ 3.3168 -0.0474 -0.2298 -0.0102  0.0364 -0.0025 -0.0444 31.0121  0.152
 -0.0367 -0.0114 -0.0227 40.485  -0.0842 44.7648 -0.2086 72.221  -0.0022
 -0.0477 -0.2108  0.4098]

RMSProp - GD
[ 1.6267 -0.159  -1.8509  1.9847  0.9328  2.265  -0.7434 26.0131 -0.579
  1.2681 -0.6497  2.8729 28.3096 -0.5988 28.8862  0.5487 30.9147 -0.9718
 -0.5039 -2.0638 -1.3314]
```

Results


10^7
10^6
10^5
10^4
10^3
Loss
0
5
10
15
20
25
Epoch
RMSProp - SGD
RMSProp - MBGD (25)
RMSProp - MBGD (50)
RMSProp - GD

Momentum

Rather then just using the gradient information at our current location it may be benefitial to use information from our previous steps as well. A general setup for this type approach looks like,

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta\, \boldsymbol{m}_t \\ \boldsymbol{m}_t &= \beta\, \boldsymbol{m}_{t-1} + (1-\beta)\, \nabla \ell(\boldsymbol{X}, \boldsymbol{\theta}_t)\end{aligned}$$

where η is our step size and β determines the weighting of the current gradient and the previous gradients.

If you have taken a course on time series, this has a flavor that looks a lot like moving average models,

$$\boldsymbol{m}_t = (1-\beta)\, \nabla \ell(\boldsymbol{X}, \boldsymbol{\theta}_t) + \beta(1-\beta)\, \nabla \ell(\boldsymbol{X}, \boldsymbol{\theta}_{t-1}) + \beta^2(1-\beta)\, \nabla \ell(\boldsymbol{X}, \boldsymbol{\theta}_{t-2}) + \cdots$$

Adam

The "adaptive moment estimation" algorithm is a combination of momentum with RMSProp,

$$\begin{aligned}\theta_{t+1} &= \theta_t - \eta_t \frac{\boldsymbol{m}_t}{\sqrt{\boldsymbol{s}_t + \epsilon}} \\ \boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\nabla\ell(\boldsymbol{X}, \theta_t) \\ \boldsymbol{s}_t &= \beta_2 \boldsymbol{s}_{t-1} + (1-\beta_2)(\nabla\ell(\boldsymbol{X}, \boldsymbol{\theta}_t))^2\end{aligned}$$

Note that RMSProp is a special case of Adam when $\beta_1 = 0$.

Adam is widely used in practice is and is commonly available within tools like Torch for fitting NN models.

In typical use $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-6}$, and $\eta_t = 0.001$ are used. As the learning rate is not guaranteed to decrease over time, the algorithm is not guaranteed to converge.

Bias corrections

One small alteration that was suggested by the original others and is commonly used is to correct for the bias towards small values in the initial estimates of $\boldsymbol{m}_t$ and $\boldsymbol{s}_t$. In which case they are replaced with,

$$\hat{\boldsymbol{m}_t} = \boldsymbol{m}/(1-\beta_1{}^t)$$
$$\hat{\boldsymbol{s}_t} = \boldsymbol{s}_t/(1-\beta_2{}^t)$$

Implementation

```
def adam_lm(X, y, beta, step=0.001, batch_size = 10, max_step=50, seed=1234, replace=True, ep
  X = jnp.c_[jnp.ones(X.shape[0]), X]
  f = lambda beta: jnp.sum((y - X @ beta)**2)
  grad = lambda beta, i: 2*X[i,:].T @ (X[i,:]@beta - y[i])
  n, k = X.shape

  res = {"x": [beta], "loss": [f(beta).item()], "iter": [0]}
  rng = np.random.default_rng(seed)

  S = np.zeros(k)
  M = np.zeros(k)
  t = 0

  for i in range(max_step):
    if replace:
      js = rng.integers(0,n,n)
    else:
      js = np.array(range(n))
      rng.shuffle(js)

    for j in js.reshape(-1, batch_size):
      t += 1
      G = grad(beta, j)
      S = b2*S + (1-b2) * G**2
```

Fitting

```
sizes = [1, 25, 50, 1000]
lrs = [0.01, 0.5, 0.75, 1]
algos = ["Adam - SGD", "Adam - MBGD (25)", "Adam - MBGD (50)", "Adam - GD"]

adam = { size: adam_lm(
                  X, y, np.zeros(X.shape[1]+1), batch_size=size,
                  step=lr, max_step=25, replace=False
                )
          for size, lr in zip(sizes,lrs) }
```

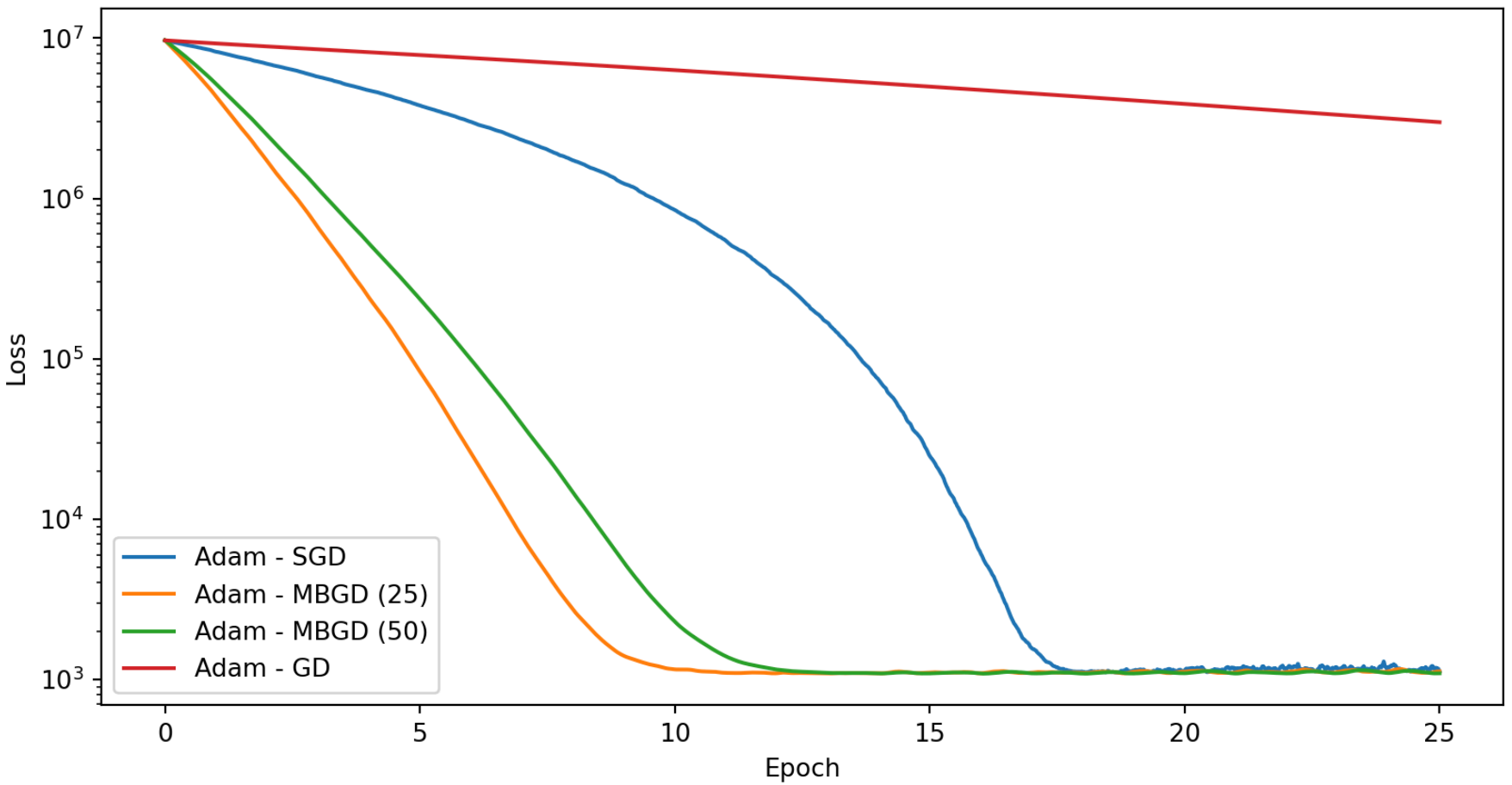
```
Adam - SGD
[ 3.0729  0.0375 -0.0207 -0.0162  0.1077  0.0935  0.0036 31.0195  0.0789
  0.0502  0.0577  0.0243 40.5269 -0.0207 44.5955 -0.0656 72.2082 -0.0416
 -0.0112 -0.0108  0.0732]

Adam - MBGD (25)
[ 2.9533 -0.0291  0.0937 -0.1201  0.0327  0.0753 -0.0276 30.9863  0.0151
  0.1048  0.046   0.0183 40.5607  0.012  44.5998 -0.0339 72.2525  0.0056
  0.0034 -0.0059  0.0159]

Adam - MBGD (50)
[ 2.9813 -0.0251  0.0356 -0.0671  0.0195  0.0527 -0.0149 30.9626  0.0383
  0.0202  0.0046 -0.0108 40.5586 -0.0162 44.6244 -0.0177 72.25    0.0053
 -0.0439 -0.0183  0.038 ]

Adam - GD
[ 1.6529 -0.045  -1.7455  1.8308  0.7478  2.5328 -0.3475 22.8207 -1.9976
  1.59   -0.1611  2.5457 23.488  -1.2908 23.6385  0.9278 24.1926 -0.3689
 -0.3436 -1.4462 -2.3327]
```

Results


10^7
10^6
10^5
10^4
10^3
Loss
0
5
10
15
20
25
Epoch
Adam - SGD
Adam - MBGD (25)
Adam - MBGD (50)
Adam - GD