Optimization - SGD

Lecture 23

Dr. Colin Rundel

From last time
Maximum Likelihood example

Normal MLE

```
1 from scipy.stats import norm
2
3 n = norm(-3.2, 1.25)
4 x = n.rvs(size=100, random_state=1234)
5 {'μ': x.mean(), 'σ': x.std()}
```

```
{'μ': np.float64(-3.156109646093205), 'σ': np.fl
```

```
1 mle_norm = lambda θ: -np.sum(
2   norm.logpdf(x, loc=θ[0], scale=θ[1])
3 )
```

```
1 mle_norm([0,1])
```

```
np.float64(667.3974708213642)
```

```
1 mle_norm([-3, 1])
```

```
np.float64(170.56457699340282)
```

```
1 mle_norm([-3.2, 1.25])
```

```
np.float64(163.83926813257395)
```

```
1 mle_norm([-3.3, 1.25])
```

```
np.float64(164.44016639757749)
```

Minimizing

```
1 optimize.minimize(mle_norm, x0=[0,1], method="bfgs")
```

```
  message: Desired error not necessarily achieved due to precision loss.
  success: False
   status: 2
      fun: nan
        x: [-1.436e+04 -3.533e+03]
      nit: 2
      jac: [       nan        nan]
hess_inv: [[ 9.443e-01  2.340e-01]
           [ 2.340e-01  5.905e-02]]
     nfev: 339
     njev: 113
```

Adding constraints

A crude way of introducing constraints is to return `np.inf` (or another large value) for any parameter values that violate the constraint - here we require the standard deviation to be positive.

```
1  def mle_norm2(θ):
2    if θ[1] <= 0:
3      return np.inf
4    else:
5      return -np.sum(
6        norm.logpdf(x, loc=θ[0], scale=θ[1])
7      )
```

```
1  optimize.minimize(
2    mle_norm2, x0=[0,1], method="bfgs"
3  )
```

```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 163.77575977255518
       x: [-3.156e+00  1.245e+00]
     nit: 9
     jac: [ 0.000e+00  0.000e+00]
hess_inv: [[ 1.475e-02 -1.179e-04]
           [-1.179e-04  7.723e-03]]
    nfev: 43
    njev: 14
```

```
1  {'μ': x.mean(), 'σ': x.std()}
```

```
{'μ': np.float64(-3.156109646093205), 'σ': np.float64(1.2446060629192537)}
```

Reparameterization

An alternative approach is to transform the parameter so it is unconstrained - here we optimize over `log(σ)` instead of σ, using `exp()` to map back to the positive reals.

```python
def mle_norm3(θ):
  return -np.sum(
    norm.logpdf(
      x, loc=θ[0], scale=np.exp(θ[1])
    )
  )
```

```python
res = optimize.minimize(
  mle_norm3, x0=[0, np.log(1)], method="bfgs
)
res
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 163.77575977255518
        x: [-3.156e+00  2.188e-01]
      nit: 9
      jac: [ 3.815e-06 -1.907e-06]
hess_inv: [[ 1.558e-02 -8.347e-05]
           [-8.347e-05  4.799e-03]]
     nfev: 45
     njev: 15
```

```python
{'μ': res.x[0], 'σ': np.exp(res.x[1])}
```

```
{'μ': np.float64(-3.1561096795318764), 'σ': np.float64(1.2446060443133926)}
```

Specifying Bounds

It is also possible to specify bounds via `bounds` but this is only available for certain optimization methods (i.e. Nelder-Mead & L-BFGS-B).

```
optimize.minimize(
  mle_norm, x0=[0,1], method="l-bfgs-b",
  bounds = [(-1e16, 1e16), (1e-16, 1e16)]
)
```

```
  message: CONVERGENCE: RELATIVE REDUCTION OF F <= FACTR*EPSMCH
  success: True
   status: 0
      fun: 163.77575977287245
        x: [-3.156e+00  1.245e+00]
      nit: 10
      jac: [ 2.046e-04  0.000e+00]
     nfev: 69
     njev: 23
 hess_inv: <2x2 LbfgsInvHessProduct with dtype=float64>
```

Exercise 1

Using `optimize.minimize()` recover the shape and scale parameters for these data using MLE.

```
1 from scipy.stats import gamma
2
3 g = gamma(a=2.0, scale=2.0)
4 x = g.rvs(size=100, random_state=1234)
5 x.round(2)
```

```
array([ 4.7 ,  1.11,  1.8 ,  6.19,  3.37,  0.25,  6.45,  0.36,  4.49,
        4.14,  2.84,  1.91,  8.03,  2.26,  2.88,  6.88,  6.84,  6.83,
        6.1 ,  3.03,  3.67,  2.57,  3.53,  2.07,  4.01,  1.51,  5.69,
        3.92,  6.01,  0.82,  2.11,  2.97,  5.02,  9.13,  4.19,  2.82,
       11.81,  1.17,  1.69,  4.67,  1.47, 11.67,  5.25,  3.44,  8.04,
        3.74,  5.73,  6.58,  3.54,  2.4 ,  1.32,  2.04,  2.52,  4.89,
        4.14,  5.02,  4.75,  8.24,  7.6 ,  1.  ,  6.14,  0.58,  2.83,
        2.88,  5.42,  0.5 ,  3.46,  4.46,  1.86,  4.59,  2.24,  2.62,
        3.99,  3.74,  5.27,  1.42,  0.56,  7.54,  5.5 ,  1.58,  5.49,
        6.57,  4.79,  5.84,  8.21,  1.66,  1.53,  4.27,  2.57,  1.48,
        5.23,  3.84,  3.15,  2.1 ,  3.71,  2.79,  0.86,  8.52,  4.36,
        3.3 ])
```

Stochastic Gradient Descent

A regression example

```
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
)
```

```
1 y
```

```
array([ -36.225
         18.145
         21.620
        -85.038
        -29.561
          3.567
        122.616
         35.006
         28.068
         54.144
         -5.532
         14.795
         11.174
         39.417
       -129.227
         88.835
         13.839
         79.478
         77.644
          9.778
        -36.573
         51.364
         20.426
        -31.104
```

```
1 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,
         1.6423, -0.5013,  0.4435,  2.0044,  0.6221,  0.0747, -1.4117,
        -0.202 , -1.3071, -0.8656, -1.311 ,  0.0424,  0.7255],
       [-0.6642,  1.4317, -0.0658, -0.7379, -0.9153,  0.8653,  0.7143,
         1.0912, -1.3773, -2.6022, -0.2955, -0.3985,  0.0918,  0.3851,
         0.502 , -0.4665,  1.6432, -0.2438, -0.4943,  1.4753],
       [ 1.5247,  1.3419,  0.4453,  0.6141,  2.0632,  1.0742,  1.4419,
```

```
1 coef.reshap
```

```
array([[ 0.
         [ 0.
         [ 0.
         [ 9.6106
         [43.4239
         [ 0.
         [ 0.
         [ 0.
         [ 0.
         [34.453
         [ 9.2929
         [ 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]
  n, k = X.shape

  f = lambda beta: jnp.sum((y - X @ beta)**2)
  grad = lambda beta: 2*X.T @ (X@beta - y)

  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
```

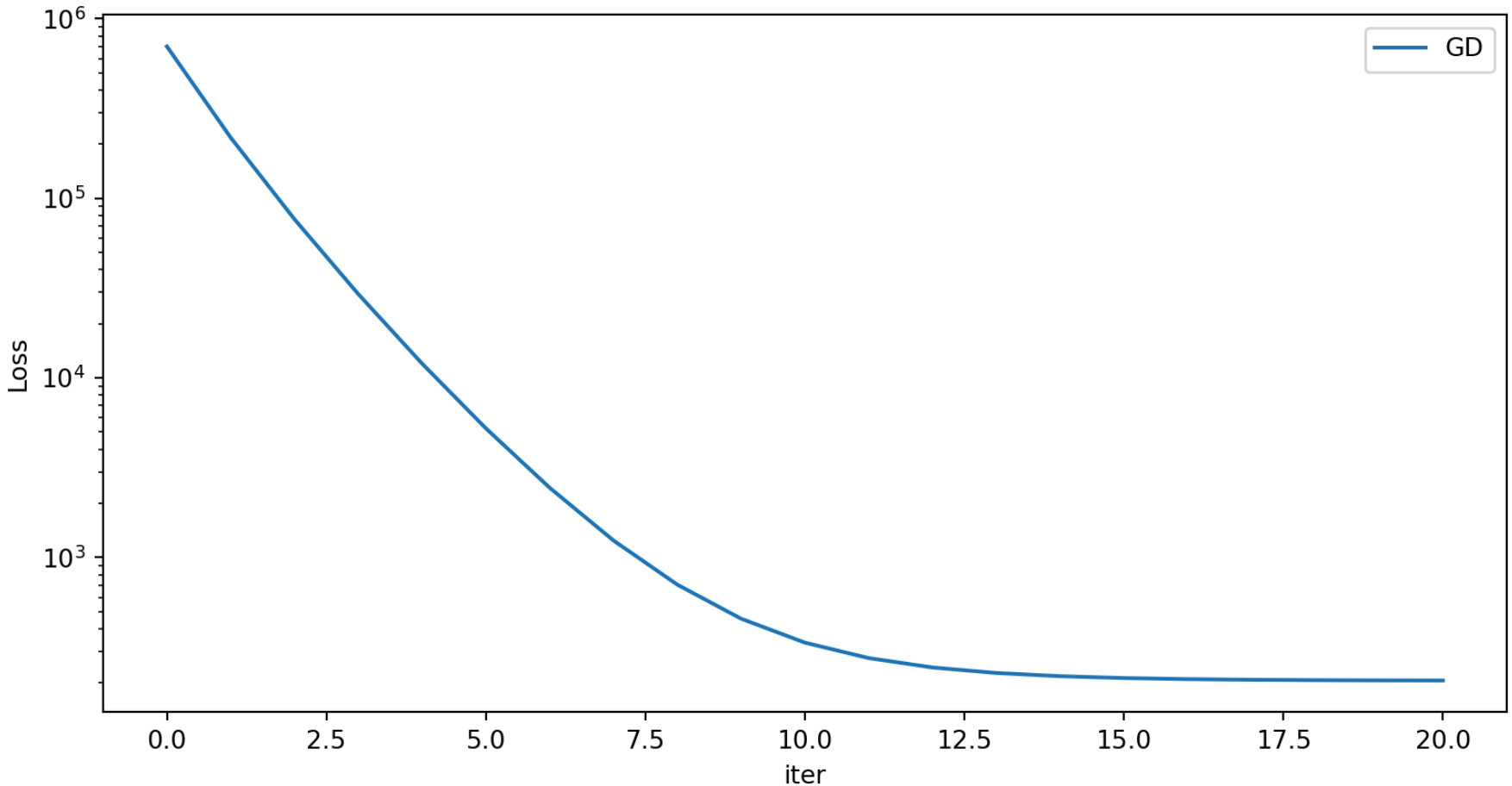
Linear regression

```
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])
```

```
t0 = time.perf_counter()
gd_lm = grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.001, max_step=20
)
gd_time = time.perf_counter() - t0
gd_lm["x"][-1]
```

```
Array([ 3.0672, -0.01  , -0.0219,  0.098 ,  9.7102, 43.4062,  0.0421,
        0.0315,  0.0994,  0.0893, 34.4578,  9.3326, -0.0302, -0.0057,
       -0.0506,  0.0686, -0.1084, -0.0687,  0.1097, -0.0393, -0.1107],      dtype=float64)
```


GD
Loss
iter
10^6
10^5
10^4
10^3
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

A quick analysis

Let's 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:

$$\underset{k\times 1}{f(\boldsymbol{\beta})} = \underset{n\times 1}{(y} - \underset{n\times k}{\boldsymbol{X}}\ \underset{k\times 1}{\boldsymbol{\beta}})^T \underset{n\times 1}{(y} - \underset{n\times k}{\boldsymbol{X}}\ \underset{k\times 1}{\boldsymbol{\beta}})$$

$$\underset{k\times 1}{\nabla f(\boldsymbol{\beta})} = 2\underset{k\times n}{\boldsymbol{X}^T}(\underset{n\times k}{\boldsymbol{X}}\ \underset{k\times 1}{\boldsymbol{\beta}} - \underset{n\times 1}{\boldsymbol{y}})$$

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

- Calculating the loss function costs $O(nk)$
- Calculating the gradient costs $O(nk)$

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 in terms of runtime
- 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)

```

Fitting

```
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])
```

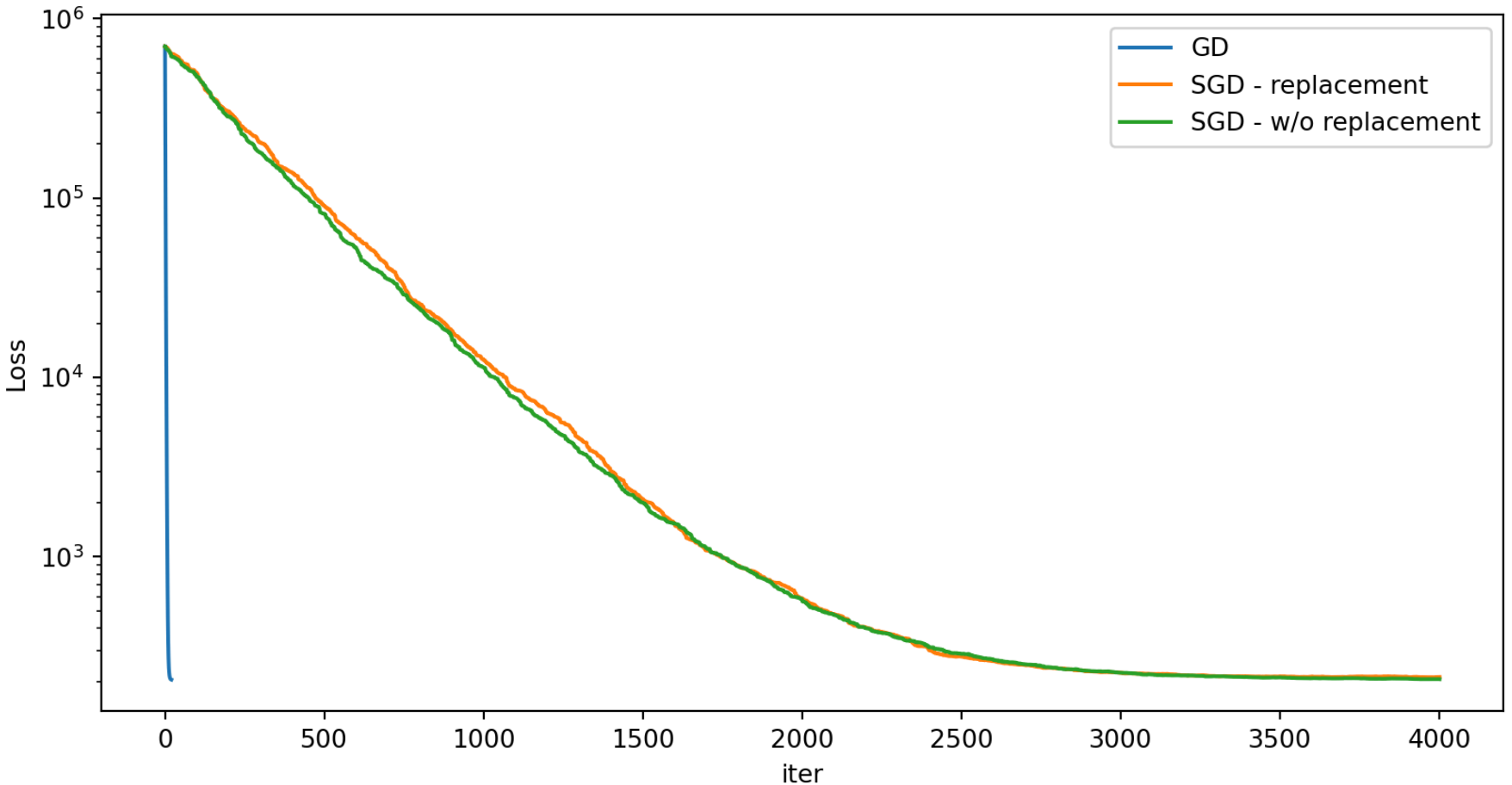
```
t0 = time.perf_counter()
sgd_lm_rep = sto_grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.001, max_step=20, replace=True
)
sgd_rep_time = time.perf_counter() - t0
sgd_lm_rep["x"][-1]
```

```
Array([ 3.1032, -0.0554,  0.0073,  0.0165,  9.7246, 43.4316,  0.075 ,
        0.0785,  0.1037,  0.0195, 34.4093,  9.3187, -0.0244,  0.0348,
        0.0063,  0.0499, -0.0332, -0.0741,  0.1323, -0.091 , -0.1617],      dtype=float64)
```

```
t0 = time.perf_counter()
sgd_lm_worep = sto_grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.001, max_step=20, replace=False
)
sgd_worep_time = time.perf_counter() - t0
sgd_lm_worep["x"][-1]
```

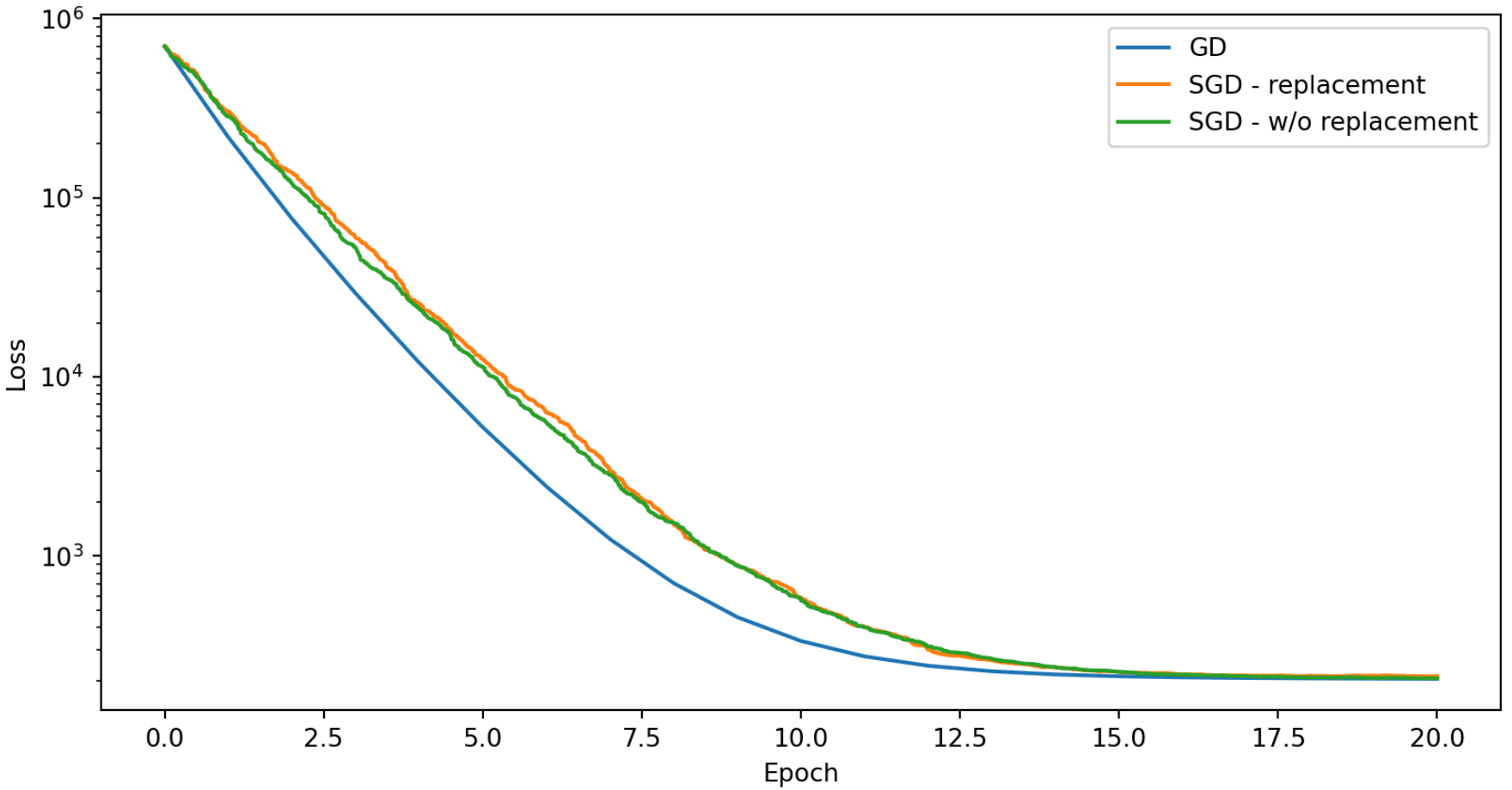
```
Array([ 3.0677, -0.0049, -0.0261,  0.0937,  9.7143, 43.3926,  0.0592,
        0.038 ,  0.1038,  0.0831, 34.4311,  9.3172, -0.0477,  0.0014,
       -0.0651,  0.06  , -0.1149, -0.0543,  0.0993, -0.0569, -0.1273],      dtype=float64)
```

Learning by iteration



Learning by Epochs

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



Run times

Method	Time / Epoch
GD	0.008s
SGD (replacement)	0.028s
SGD (w/o replacement)	0.026s

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

```
t0 = time.perf_counter()
gd_lm = grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.00005, max_step=3
)
gd_time = time.perf_counter() - t0
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)
```

```
t0 = time.perf_counter()
sgd_lm_rep = sto_grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.001, max_step=3, replace=True
)
sgd_rep_time = time.perf_counter() - t0
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)
```

```
t0 = time.perf_counter()
sgd_lm_worep = sto_grad_desc_lm(
  X, y, np.zeros(X.shape[1]+1),
  step = 0.001, max_step=3, replace=False
)
```

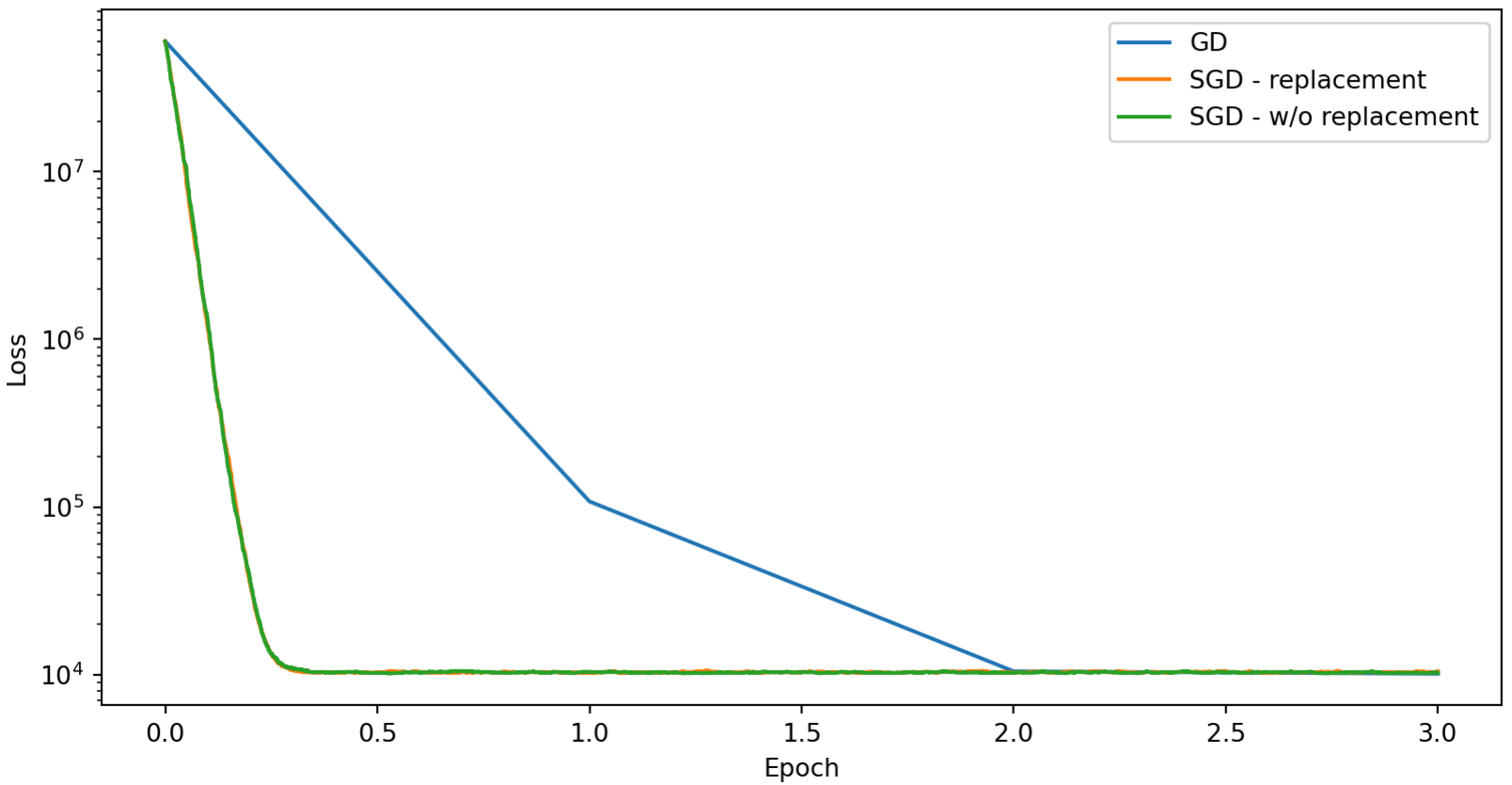
```
sgd_worep_time = time.perf_counter() - t0
sgd_lm_worep["x"][-1]
```

```
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)
```

Method	Time / Epoch
GD	0.035s
SGD (replacement)	1.558s
SGD (w/o replacement)	1.504s

Results

Full | Zoom | Timings



Mini batch gradient descent

This is a variant of stochastic gradient descent where a subset 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
- Mini batch composition can be constructed by sampling data with or without replacement

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)

```

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])
```

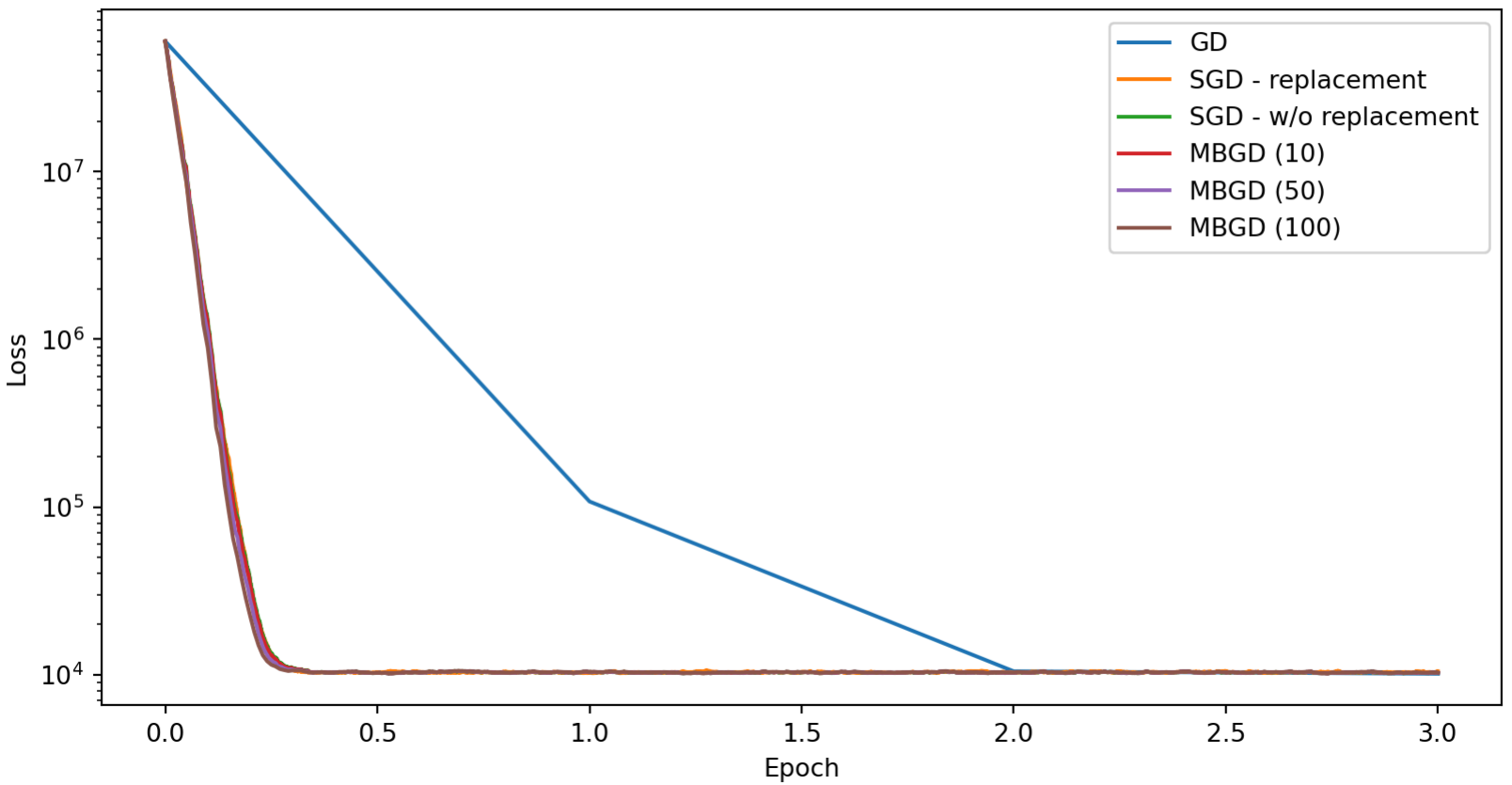
```
sizes = [10,50,100]
mbgd = {}
mbgd_time = {}
for size in sizes:
  t0 = time.perf_counter()
  mbgd[size] = mb_grad_desc_lm(
    X, y, np.zeros(X.shape[1]+1), batch_size=size,
    step = 0.001, max_step=3, replace=False
  )
  mbgd_time[size] = time.perf_counter() - t0
```

```
Batch size: 10 (0.401s / epoch)
[ 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 (0.117s / epoch)
[ 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 (0.074s / epoch)
[ 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 | Timings



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

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

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

SGD estimator

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

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

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,

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

  as B is a subset of the data, therefore our algorithm is likely to never converge.

Solution

The practical solution to this is to implement a learning rate schedule which generally shrinks 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 right balance.

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 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.

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

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

here ϵ is a small constant (i.e. 10^{-7}) 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

```

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_]
```

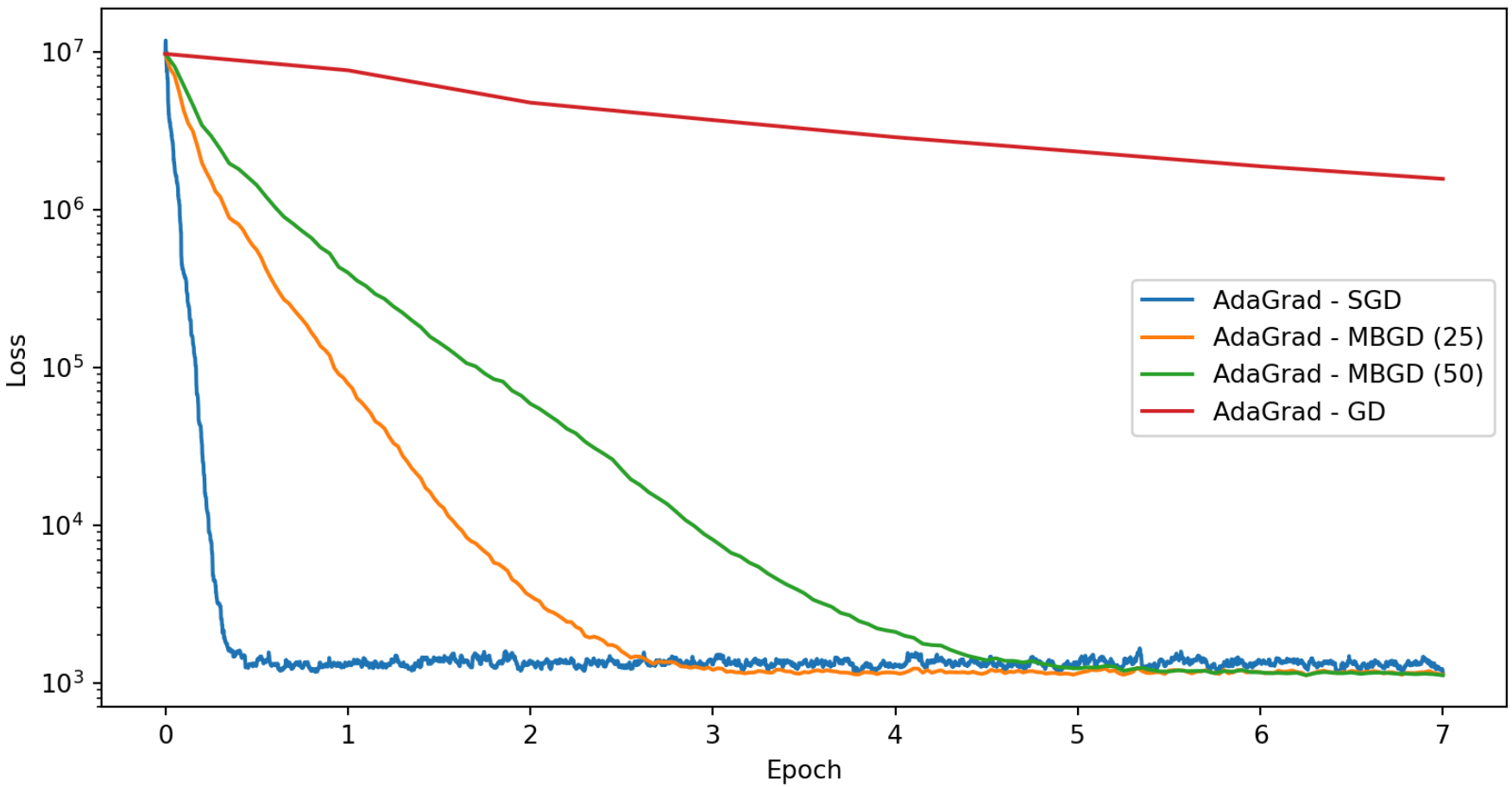
```
array([ 3.0034,  0.0144,  0.0304, -0.0306,  0.0334,  0.0292, -0.0214,
       30.9685,  0.0189, -0.005 ,  0.005 ,  0.0016, 40.5613, -0.0422,
       44.6158, -0.0495, 72.2522,  0.002 , -0.0336,  0.0148,  0.0516])
```

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=7, replace=True, eps=1e-8
                  )
            for size, lr in zip(sizes,lrs) }
```

```
AdaGrad - SGD
[ 3.1134  0.0657  0.1134 -0.0746 -0.0023 -0.0545 -0.0544 30.8796 -0.1221
  0.0449 -0.1069  0.0407 40.5307  0.0046 44.6237 -0.0401 72.2858  0.0184
 -0.0153  0.0663  0.0114]

AdaGrad - MBGD (25)
[ 3.0676  0.0346  0.0862 -0.0847  0.0706 -0.0527  0.0161 30.9695 -0.0555
  0.0156 -0.0922 -0.0061 40.5614 -0.0126 44.6231 -0.0221 72.2704 -0.0215
 -0.0182  0.0601  0.0787]

AdaGrad - MBGD (50)
[ 3.0519  0.0268  0.0942 -0.0765  0.0593 -0.0484  0.0193 30.9786 -0.0188
  0.0246 -0.0656 -0.0208 40.5611  0.0059 44.6133 -0.0297 72.2234 -0.0276
 -0.0198  0.0425  0.0926]

AdaGrad - GD
[ 2.6765  1.1926 -0.7393  1.2202 -0.0051  1.5508  0.3784 27.3744 -1.4125
  3.0303 -0.2343  2.4958 30.5815  2.4433 32.5644  1.4608 35.3517 -0.7034
 -0.5997 -0.6608 -1.1752]
```

Results


10^7
10^6
10^5
10^4
10^3
Loss
0
1
2
3
4
5
6
7
Epoch
AdaGrad - SGD
AdaGrad - MBGD (25)
AdaGrad - MBGD (50)
AdaGrad - GD

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 often used.

Implementation

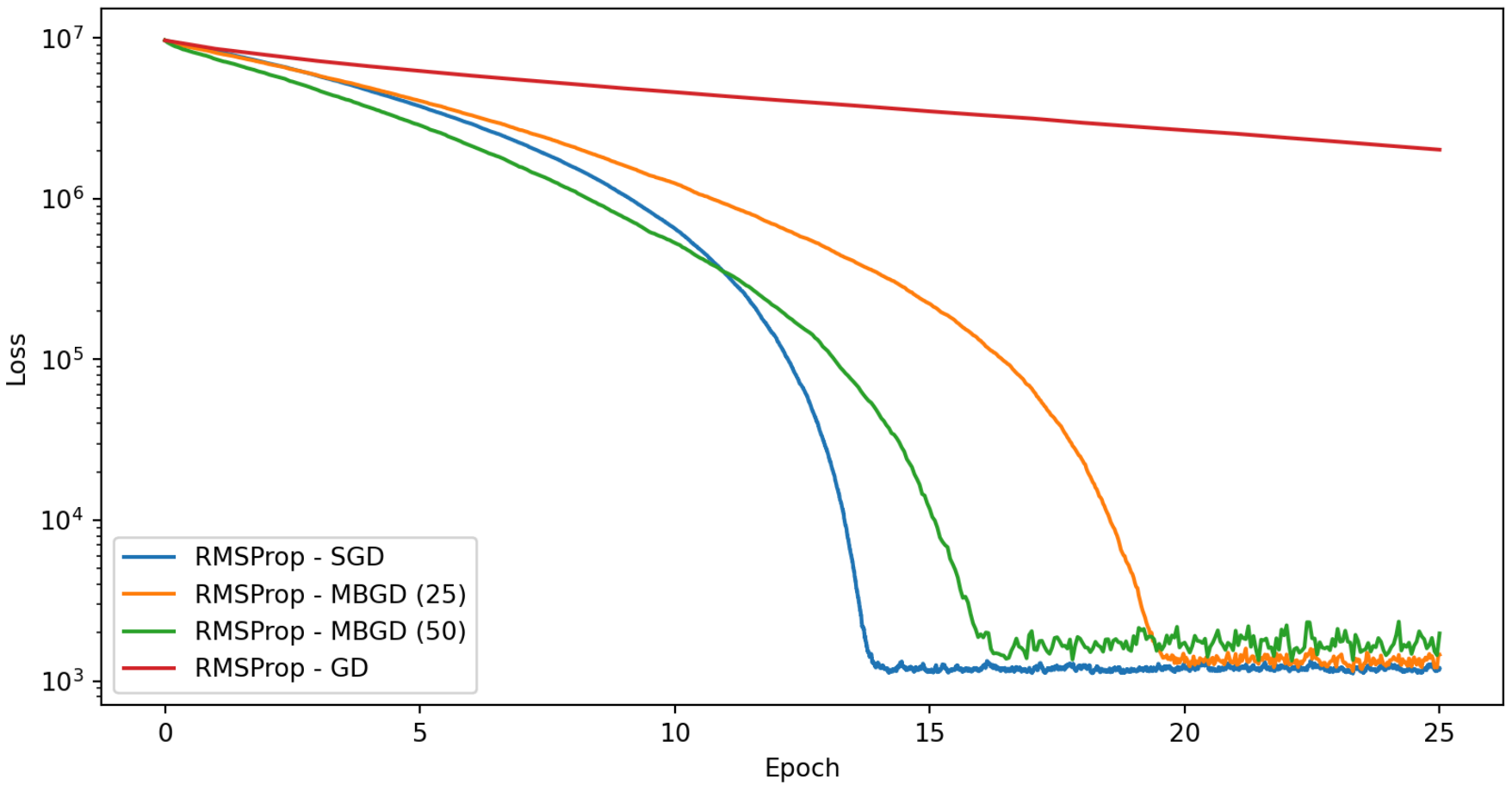
```
def rmsprop_lm(X, y, beta, step, batch_size = 10, max_step=50, seed=1234, replace=True, eps=1
  X = jnp.c_[jnp.ones(X.shape[0]), X]
  f = lambda beta: jnp.sum((y - X @ beta)**2)
  grad = lambda beta, i: 2*X[i,:].T @ (X[i,:]@beta - y[i])
  n, k = X.shape

  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

```

Fitting

```
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=True
                  )
            for size, lr in zip(sizes,lrs) }
```

```
RMSProp - SGD
[ 2.9718 -0.0486  0.1521  0.0179  0.1236 -0.0449 -0.0614 30.8114  0.0474
  0.074  -0.0083 -0.0569 40.4583 -0.0975 44.679  -0.1505 72.2751 -0.0066
 -0.0618 -0.1117 -0.0201]

RMSProp - MBGD (25)
[ 2.9707 -0.2066  0.2688  0.0507  0.1513  0.0343 -0.0799 30.7114  0.0451
  0.1576 -0.0103 -0.0894 40.338  -0.1608 44.6958 -0.2083 72.2931 -0.143
 -0.0646 -0.0814 -0.0219]

RMSProp - MBGD (50)
[ 3.0954 -0.181   0.4018  0.2723  0.2013 -0.0766 -0.0231 30.7481  0.1931
  0.4516 -0.0279 -0.1569 40.3096 -0.2517 44.6712 -0.349  72.3357 -0.1481
 -0.0044 -0.1344  0.0849]

RMSProp - GD
[ 2.2162 -0.6382 -2.1805  2.4839  0.9998  0.9919  0.0923 25.8598 -1.7242
  1.7694  0.0141  3.112  28.4576 -1.6735 29.3372  0.484  31.0173  0.0241
  1.0046 -1.2184 -2.8247]
```

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

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta\, \boldsymbol{m}_t$$

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

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 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 authors 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,

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

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
```

Fitting

```python
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=True
                )
          for size, lr in zip(sizes,lrs) }
```

```
Adam - SGD
[ 2.9752 -0.0641  0.1981  0.0515  0.0914 -0.053  -0.0403 30.8483  0.0677
  0.1179  0.0005 -0.0296 40.4603 -0.0836 44.6719 -0.1786 72.2796 -0.0657
 -0.0142 -0.1159 -0.0386]

Adam - MBGD (25)
[ 2.9886  0.0628  0.0822 -0.0225  0.0747 -0.1306 -0.0269 30.9399  0.1265
  0.0126  0.0624  0.0686 40.5911 -0.0319 44.6542 -0.0513 72.2608  0.0044
  0.0609 -0.0892  0.0563]

Adam - MBGD (50)
[ 2.9901  0.0812  0.0418 -0.0795  0.039  -0.0832  0.0236 30.9293  0.1255
 -0.0052  0.0245  0.014  40.6152 -0.0599 44.6557 -0.0594 72.2556 -0.0312
  0.0611 -0.0655  0.1096]

Adam - GD
[ 2.7747 -1.783  -3.6167  4.6093  2.4556  0.8484  0.0964 22.7301 -0.3594
  1.9632  0.5982  2.0708 23.4988 -0.3894 23.818   0.4661 24.1671 -0.9899
 -0.1128 -1.777  -4.0147]
```

Results