Stochastic Gradient Descent

Lecture 26

Dr. Colin Rundel

Gradient Descent for Regression

A regression example

```r
set.seed(1234)
n = 200; p = 20
X = matrix(rnorm(n * p), n, p)
true_beta = rep(0, p)
true_beta[c(3, 7, 12, 18)] = c(5.2, -3.1, 8.7, -1.5)
y = 3 + X %*% true_beta + rnorm(n)
```

```r
head(y, 20)
```

```
            [,1]
 [1,]  4.54630945
 [2,] -0.27563573
 [3,] -8.92201672
 [4,]  3.81019348
 [5,] -8.01772803
 [6,] -5.65360256
 [7,] -9.45140764
 [8,]  4.92135279
 [9,] -4.13507430
[10,]  6.77822004
[11,] -1.16028786
[12,] 20.83784033
[13,]  4.07084356
[14,]  9.58649951
[15,]  1.41834659
[16,] 13.08886089
[17,]  8.68524259
[18,] -1.72856652
[19,]  0.97934767
[20,]  0.03342649
```

```r
true_beta
```

```
 [1]  0.0  0.0  5.2  0.0  0.0  0.0 -3.1  0.0  0.0  0.0
[14]  0.0  0.0  0.0  0.0 -1.5  0.0  0.0
```

Minimalistic GD for linear regression

```r
grad_desc_lm = function(X, y, beta, step, max_step = 50) {
  X = cbind(1, X)
  n = nrow(X)
  k = ncol(X)

  f = \(beta) sum((y - X %*% beta)^2)
  grad = \(beta) 2 * t(X) %*% (X %*% beta - y)

  res = list(x = list(beta), loss = f(beta), iter = 0)

  for (i in seq_len(max_step)) {
    beta = beta - grad(beta) * step
    res$x = c(res$x, list(beta))
    res$loss = c(res$loss, f(beta))
    res$iter = c(res$iter, i)
  }

  res
}
```

Linear regression

```
(lm_fit = lm(y ~ X)) |> coef() |> round(2) |> unname()
```

```
 [1]  3.12 -0.07  0.01  5.34  0.05  0.07  0.02 -3.07 -0.04  0.10  0.04
[12]  0.10  8.66 -0.03  0.10 -0.01  0.11  0.00 -1.44  0.07 -0.06
```

```
deviance(lm_fit) |> round(2)
```

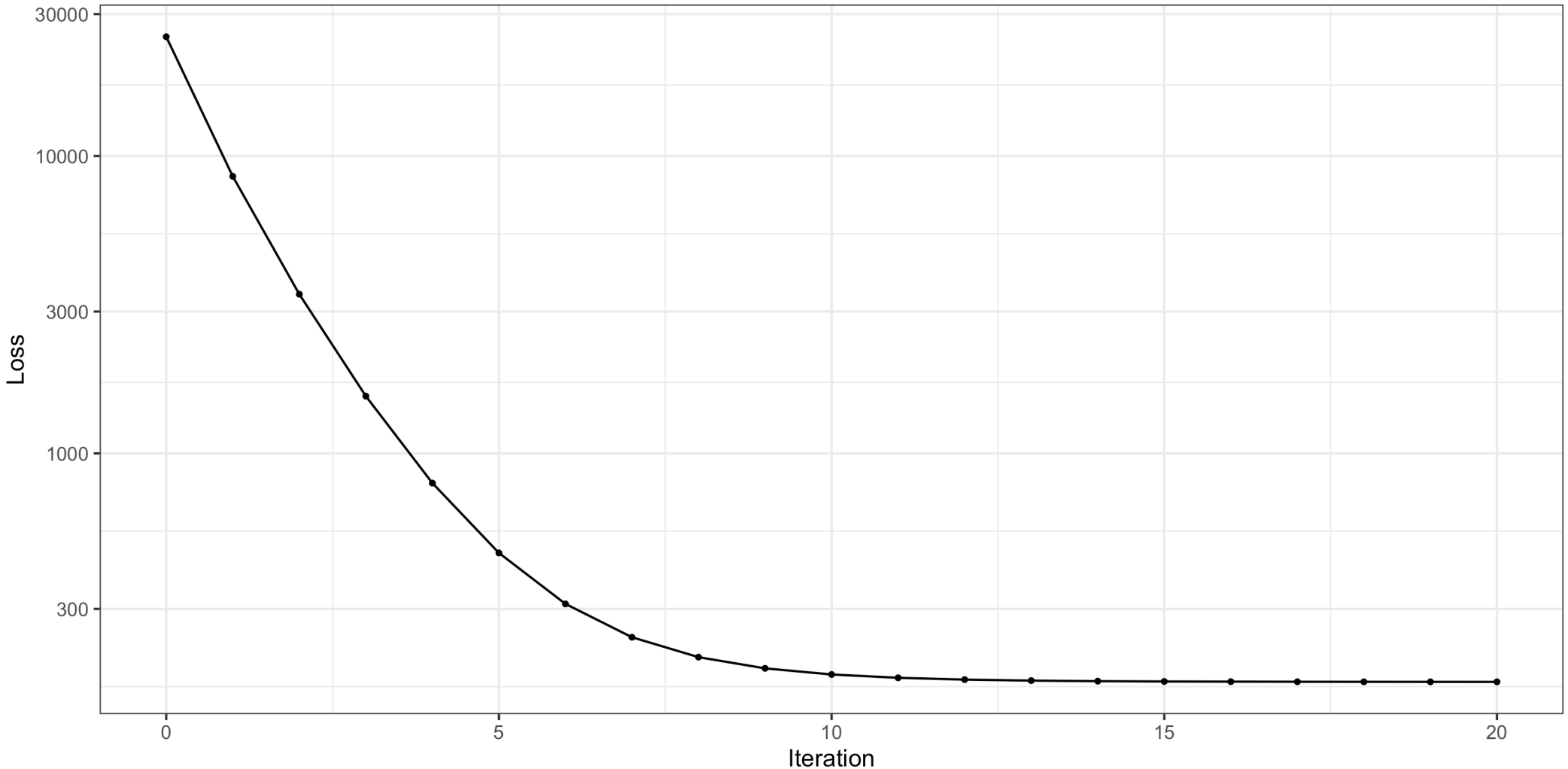
```
[1] 170.49
```

```
gd_lm = grad_desc_lm(X, y, rep(0, p + 1), step = 0.001, max_step = 20)
gd_lm$x |> tail(1) |> unlist() |> round(2)
```

```
 [1]  3.11 -0.08  0.01  5.34  0.05  0.07  0.02 -3.06 -0.04  0.10  0.04
[12]  0.10  8.65 -0.02  0.09 -0.01  0.11  0.00 -1.44  0.08 -0.07
```

```
gd_lm$loss |> tail(1) |> round(2)
```

```
[1] 170.53
```


30000
10000
3000
1000
300
Loss
0
5
10
15
20
Iteration

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.

We've defined 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

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

```r
sgd_lm = function(X, y, beta, step, max_step = 50, seed = 1234, replace = TRUE) {
  X = cbind(1, X)
  n = nrow(X); k = ncol(X)
  f = \(beta) sum((y - X %*% beta)^2)
  grad_i = \(beta, i) 2 * X[i, ] * (X[i, ] %*% beta - y[i])

  res = list(x = list(beta), loss = f(beta), iter = 0)
  set.seed(seed)

  for (ep in seq_len(max_step)) {
    if (replace) {
      js = sample.int(n, n, replace = TRUE)
    } else {
      js = sample.int(n)
    }

    for (j in js) {
      beta = beta - grad_i(beta, j) * step
      res$x = c(res$x, list(beta))
      res$loss = c(res$loss, f(beta))
      res$iter = c(res$iter, tail(res$iter, 1) + 1)
    }
  }
```

Fitting

```
lm_fit |> coef() |> round(2) |> unname()
```

```
 [1]  3.12 -0.07  0.01  5.34  0.05  0.07  0.02 -3.07 -0.04  0.10  0.04
[12]  0.10  8.66 -0.03  0.10 -0.01  0.11  0.00 -1.44  0.07 -0.06
```

```
lm_fit |> deviance() |> round(2)
```

```
[1] 170.49
```

```
sgd_lm_rep = sgd_lm(
  X, y, rep(0, p + 1),
  step = 0.001, max_step = 20, replace = TRUE
)
sgd_lm_rep$x |> tail(1) |> unlist() |> round(2)
```

```
 [1]  3.11 -0.07  0.00  5.38  0.06  0.07  0.05 -3.04 -0.06  0.09  0.07
[12]  0.13  8.65  0.01  0.12  0.01  0.14 -0.06 -1.45  0.03 -0.08
```

```
sgd_lm_rep$loss |> tail(1) |> round(2)
```

```
[1] 173.31
```

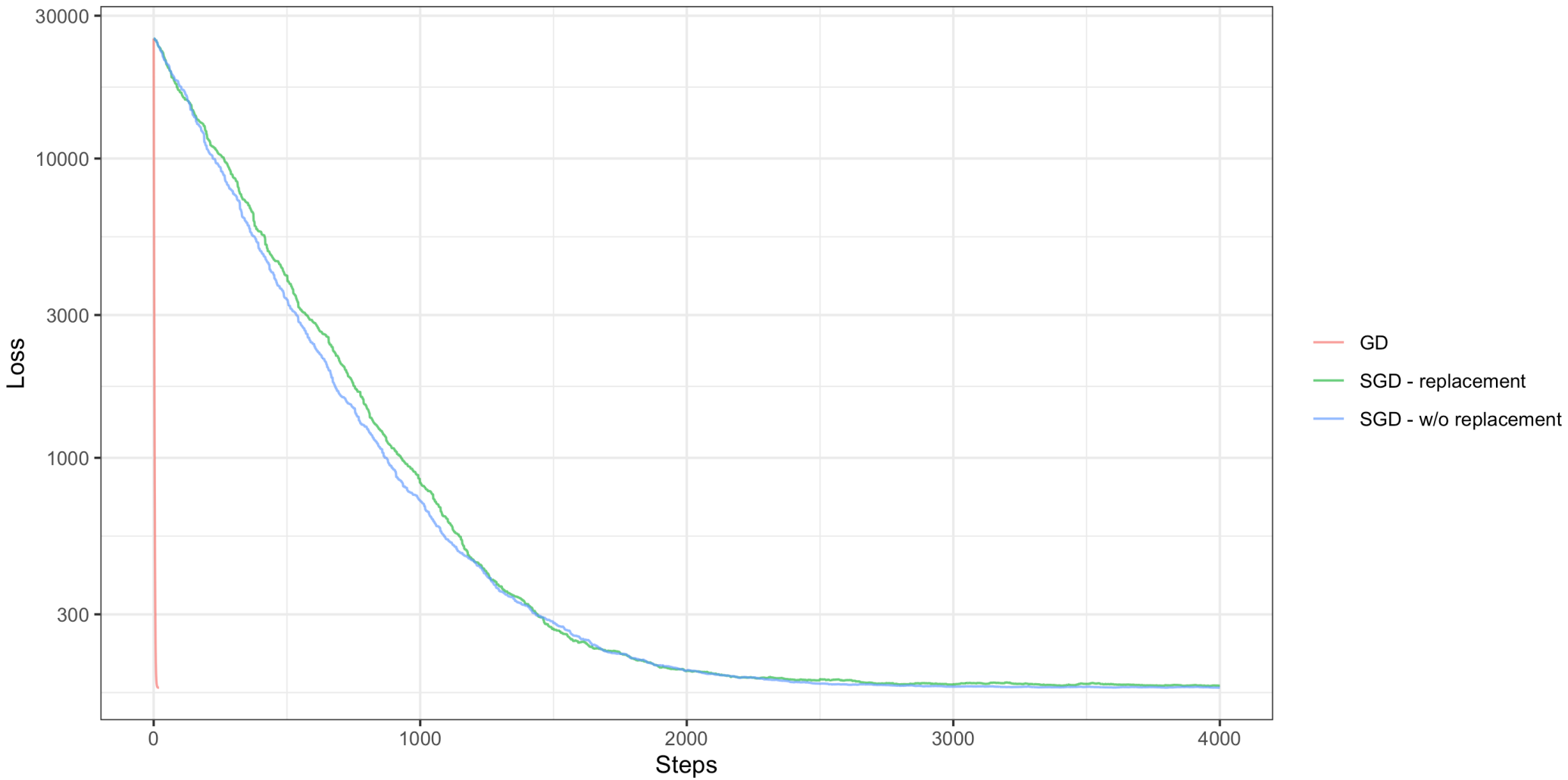
```
sgd_lm_worep = sgd_lm(
   X, y, rep(0, p + 1),
   step = 0.001, max_step = 20, replace = FALSE
)
sgd_lm_worep$x |> tail(1) |> unlist() |> round(2)
```

```
 [1]  3.11 -0.08  0.00  5.34  0.04  0.07  0.02 -3.06 -0.04  0.10  0.04
[12]  0.11  8.65 -0.01  0.08 -0.01  0.10 -0.01 -1.43  0.07 -0.07
```

```
sgd_lm_worep$loss |> tail(1) |> round(2)
```

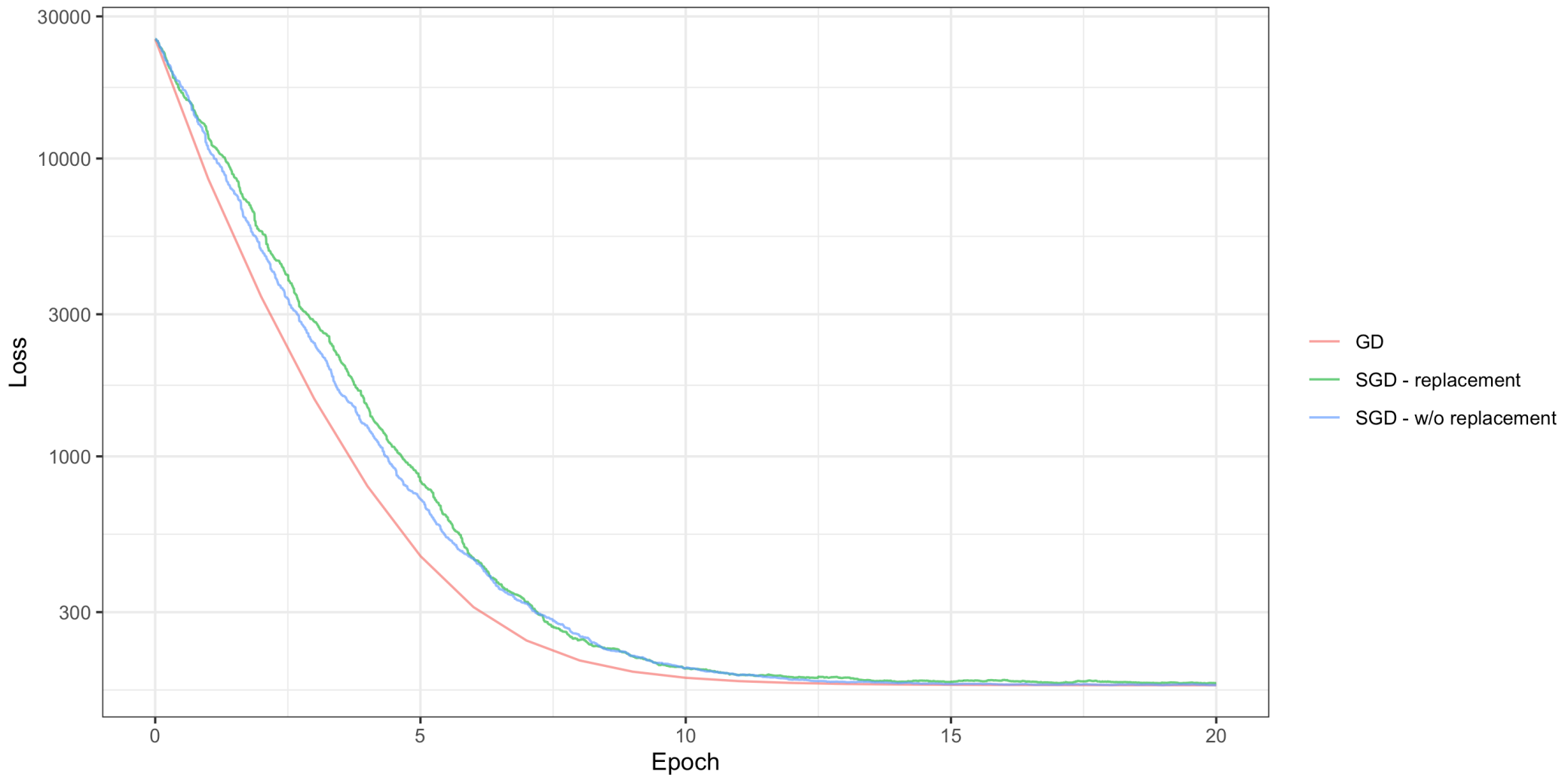
```
[1] 170.65
```

Learning by step



Learning by Epochs

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



Run times

```r
gd_time = system.time(
  grad_desc_lm(X, y, rep(0, p + 1), step = 0.001, max_step = 20)
)
sgd_rep_time = system.time(
  sgd_lm(X, y, rep(0, p + 1), step = 0.001, max_step = 20, replace = TRUE)
)
sgd_worep_time = system.time(
  sgd_lm(X, y, rep(0, p + 1), step = 0.001, max_step = 20, replace = FALSE)
)
```

Method	Run time	Time / Epoch
GD	0.000s	0.000s
SGD (replacement)	0.357s	0.018s
SGD (w/o replacement)	0.358s	0.018s

A bigger example

Remember that our costs scale as $O(nk)$, so let's see what happens when we increase the data size.

```
set.seed(1234)
n = 10000; p = 20
X = matrix(rnorm(n * p), n, p)
true_beta = rep(0, p)
true_beta[c(3, 7, 12, 18)] = c(5.2, -3.1, 8.7, -1.5)
y = 3 + X %*% true_beta + rnorm(n)
```

```
lm_fit = lm(y ~ X)
lm_fit |> coef() |> round(2) |> unname()
```

```
 [1]  3.03  0.01  0.00  5.22 -0.01  0.00  0.00 -3.10 -0.01  0.00 -0.01
[12] -0.01  8.69 -0.02 -0.01  0.02  0.02  0.01 -1.48  0.00 -0.01
```

Fitting

```
lm_fit |> coef() |> round(2) |> unname()
```

```
 [1]  3.03  0.01  0.00  5.22 -0.01  0.00  0.00 -3.10 -0.01  0.00 -0.01
[12] -0.01  8.69 -0.02 -0.01  0.02  0.02  0.01 -1.48  0.00 -0.01
```

```
lm_fit |> deviance() |> round(2)
```

```
[1] 9859.22
```

```
gd_lm = grad_desc_lm(
  X, y, rep(0, p + 1),
  step = 0.00005, max_step = 3
)
gd_lm$x |> tail(1) |> unlist() |> round(2)
```

```
 [1]  3.03  0.01  0.00  5.22 -0.01 -0.01  0.01 -3.10 -0.01  0.00 -0.01
[12] -0.01  8.70 -0.02 -0.01  0.01  0.02  0.01 -1.48  0.00 -0.01
```

```
gd_lm$loss |> tail(1) |> round(2)
```

```
[1] 9859.28
```

```r
sgd_lm_rep = sgd_lm(
  X, y, rep(0, p + 1),
  step = 0.001, max_step = 3, replace = TRUE
)
sgd_lm_rep$x |> tail(1) |> unlist() |> round(2)
```

```
 [1]  3.00  0.00  0.01  5.24 -0.01  0.00 -0.03 -3.14 -0.03 -0.02 -0.01
[12] -0.03  8.69 -0.01  0.01  0.06 -0.01 -0.04 -1.51  0.01 -0.02
```

```r
sgd_lm_rep$loss |> tail(1) |> round(2)
```

```
[1] 9980.55
```

```r
sgd_lm_worep = sgd_lm(
  X, y, rep(0, p + 1),
  step = 0.001, max_step = 3, replace = FALSE
)
sgd_lm_worep$x |> tail(1) |> unlist() |> round(2)
```

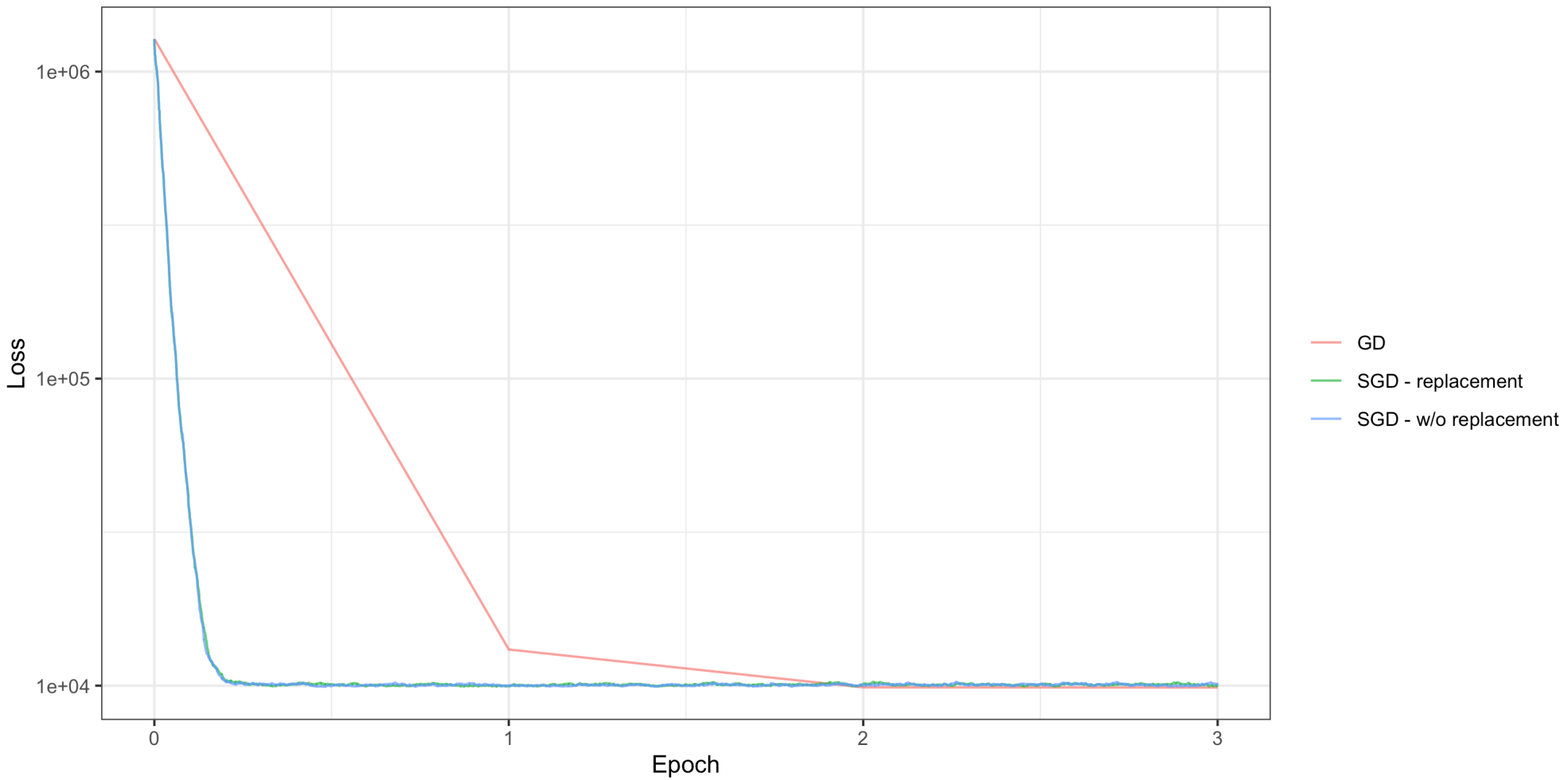
```
 [1]  2.97 -0.04  0.02  5.15  0.03 -0.02  0.04 -3.09  0.03 -0.03  0.00
[12] -0.05  8.72 -0.05 -0.01 -0.01  0.01  0.00 -1.47  0.03  0.04
```

```r
sgd_lm_worep$loss |> tail(1) |> round(2)
```

```
[1] 10085.99
```

Results

Full | Zoom | Timings



Mini-batch Gradient Descent

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

```
mb_grad_desc_lm = function(X, y, beta, step, batch_size = 10, max_step = 50, seed = 1234, rep
  X = cbind(1, X)
  n = nrow(X); k = ncol(X)
  f = \(beta) sum((y - X %*% beta)^2)
  grad_b = \(beta, i) 2 * t(X[i,]) %*% (X[i,] %*% beta - y[i])

  res = list(x = list(beta), loss = f(beta), iter = 0)
  set.seed(seed)

  for (ep in seq_len(max_step)) {
    if (replace) {
      js = sample.int(n, n, replace = TRUE)
    } else {
      js = sample.int(n)
    }

    batches = split(js, ceiling(seq_along(js) / batch_size))
    for (batch in batches) {
      beta = beta - grad_b(beta, batch) * step
      res$x = c(res$x, list(beta))
      res$loss = c(res$loss, f(beta))
      res$iter = c(res$iter, tail(res$iter, 1) + 1)
    }
```

Fitting

```r
lm_fit |> coef() |> round(2) |> unname()
```

```
 [1]  3.03  0.01  0.00  5.22 -0.01  0.00  0.00 -3.10 -0.01  0.00 -0.01
[12] -0.01  8.69 -0.02 -0.01  0.02  0.02  0.01 -1.48  0.00 -0.01
```

```r
lm_fit |> deviance() |> round(2)
```

```
[1] 9859.22
```

```r
sizes = c(10, 50, 100)
mbgd = list()
for (size in sizes) {
  mbgd[[as.character(size)]] = mb_grad_desc_lm(
    X, y, rep(0, p + 1), batch_size = size,
    step = 0.001, max_step = 3, replace = FALSE
  )
}
```

```r
mbgd[["10"]]$x |> tail(1) |> unlist() |> round(2)
```

```
 [1]  2.97 -0.04  0.02  5.15  0.03 -0.02  0.04 -3.09  0.03 -0.03  0.00
[12] -0.05  8.72 -0.05 -0.01 -0.01  0.01  0.00 -1.47  0.03  0.04
```

```r
mbgd[["10"]]$loss |> tail(1) |> round(2)
```

```
[1] 10088.01
```

```r
mbgd[["50"]]$x |> tail(1) |> unlist() |> round(2)
```

```
 [1]  2.97 -0.04  0.02  5.15  0.03 -0.02  0.04 -3.09  0.03 -0.03  0.00
[12] -0.05  8.72 -0.05 -0.01 -0.01  0.01  0.00 -1.47  0.03  0.05
```

```r
mbgd[["50"]]$loss |> tail(1) |> round(2)
```

```
[1] 10105.2
```

```r
mbgd[["100"]]$x |> tail(1) |> unlist() |> round(2)
```

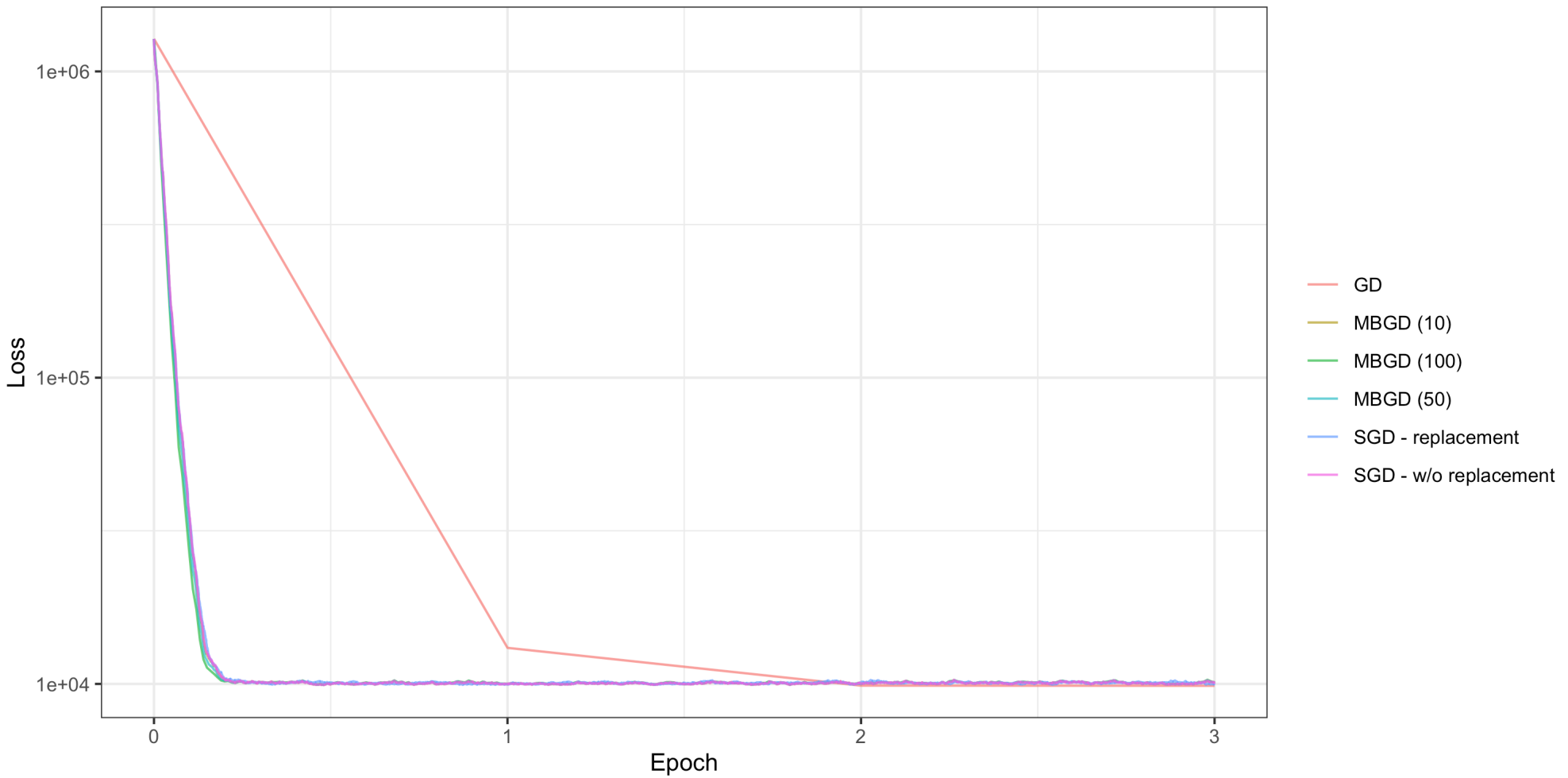
```
 [1]  2.96 -0.04  0.02  5.15  0.03 -0.02  0.04 -3.09  0.03 -0.03  0.00
[12] -0.05  8.73 -0.05 -0.01 -0.01  0.00  0.00 -1.47  0.03  0.05
```

```r
mbgd[["100"]]$loss |> tail(1) |> round(2)
```

```
[1] 10122.4
```

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.

torch

What is `torch`?

The `torch` package is a native R port of PyTorch, built directly on top of `libtorch` (the C++ backend that powers PyTorch). There is no Python dependency.

It provides:

- Multi-dimensional arrays (tensors) with GPU support
- Automatic differentiation (autograd)
- Neural network building blocks (`nn_module`, `nn_linear`, `nn_relu`, ...)
- A growing ecosystem - `luz` (high-level training), `torchvision`, `torchaudio`

Tensors

A `torch_tensor` is the basic data type - a multi-dimensional array with a specific dtype and device.

```
1  torch_zeros(3)
```

```
torch_tensor
 0
 0
 0
[ CPUFloatType{3} ]
```

```
1  torch_ones(3, 2)
```

```
torch_tensor
 1  1
 1  1
 1  1
[ CPUFloatType{3,2} ]
```

```
1  torch_manual_seed(1234)
2  torch_rand(2, 2, 2)
```

```
torch_tensor
(1,.,.) =
 0.0290  0.4019
  0.2598  0.3666

(2,.,.) =
 0.0583  0.7006
  0.0518  0.4681
[ CPUFloatType{2,2,2} ]
```

From R to torch

Converting R objects to tensors uses `torch_tensor()`:

```
1 torch_tensor(diag(3))
```

```
torch_tensor
 1  0  0
 0  1  0
 0  0  1
[ CPUFloatType{3,3} ]
```

```
1 torch_tensor(matrix(1:6, 2, 3))
```

```
torch_tensor
 1  3  5
 2  4  6
[ CPULongType{2,3} ]
```

Note that R integer vectors and matrices become `Long` tensors. Most `nn` layers expect `Float` (float32) so you usually want to convert explicitly:

```
1 torch_tensor(matrix(1:6, 2, 3), dtype = torch_float())
```

```
torch_tensor
 1  3  5
 2  4  6
[ CPUFloatType{2,3} ]
```

From torch to R

Going the other direction uses `as_array()` (or `as.matrix()` / `as.numeric()` for 2d / 1d tensors):

```
t = torch_tensor(matrix(1:6, 2, 3), dtype = torch_float())
as_array(t)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
as.matrix(t)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

If a tensor is tracking gradients you must `$detach()` it first:

```
t = torch_tensor(c(1, 2, 3), requires_grad = TRUE)
as_array(t$detach())
```

```
[1] 1 2 3
```

Reshaping tensors

Tensors can be reshaped with `$view()` or `$reshape()`, transposed with `$t()`:

```
1  a = torch_arange(1, 12)
2  a$view(c(3, 4))
```

```
torch_tensor
  1   2   3   4
  5   6   7   8
  9  10  11  12
[ CPUFloatType{3,4} ]
```

```
1  a$reshape(c(2, 6))
```

```
torch_tensor
  1   2   3   4   5   6
  7   8   9  10  11  12
[ CPUFloatType{2,6} ]
```

```
1  a$view(c(3, 4))$t()
```

```
torch_tensor
  1   5   9
  2   6  10
  3   7  11
  4   8  12
[ CPUFloatType{4,3} ]
```

Adding and removing dimensions

`$squeeze()` drops dimensions of size 1, `$unsqueeze(dim)` inserts a new dimension of size 1:

```
1 b = torch_zeros(2, 1, 3)
2 b$shape
```

```
[1] 2 1 3
```

```
1 b$squeeze()$shape
```

```
[1] 2 3
```

```
1 c = torch_zeros(2, 3)
2 c$unsqueeze(1)$shape
```

```
[1] 1 2 3
```

```
1 c$unsqueeze(2)$shape
```

```
[1] 2 1 3
```

Adding and removing dimensions is typically done to conform tensor shapes so that they can be combined via

Autograd

This is where torch starts to do real work for us - any tensor created with `requires_grad = TRUE` will have its operations recorded so that gradients can be computed automatically:

```
a = torch_tensor(2, requires_grad = TRUE)
f = a^3 + 2*a
f
```

```
torch_tensor
 12
[ CPUFloatType{1} ][ grad_fn = <AddBackward0> ]
```

```
f$backward()
a$grad
```

```
torch_tensor
 14
[ CPUFloatType{1} ]
```

This is the same gradient we would get by hand from $\frac{df}{da} = 3a^2 + 2 = 14$ at $a = 2$.

Importantly, we did not have to write the gradient ourselves - we wrote the `grad` function by hand earlier in `grad_desc_lm()`, but autograd will do that for us.

Computational graph

Behind the scenes, every operation on a tensor with `requires_grad = TRUE` adds a node to a *dynamic* computational graph. The `$grad_fn` attribute points at the operation that produced the tensor:

```
1 a = torch_tensor(2, requires_grad = TRUE)
2 f = a^3 + 2*a
3 f$grad_fn
```

```
AddBackward0
```

When we call `f$backward()`, torch walks this graph from `f` back to `a`, applies the chain rule, and accumulates the result into `a$grad`.

The graph is rebuilt on every forward pass - this is what makes it "dynamic" and is one of the things people like about

Linear regression with `torch`

Tensors from our regression data

We will reuse the regression problem from earlier in this lecture (`X` is $n \times p$ with $n = 10000$, $p = 20$, `y` is the response). We need to convert these to tensors and prepend a column of 1s for the intercept:

```
Xt = torch_tensor(cbind(1, X), dtype = torch_float())
yt = torch_tensor(as.numeric(y), dtype = torch_float())

beta = torch_zeros(p + 1, dtype = torch_float(), requires_grad = TRUE)
```

```
Xt$shape
```

```
[1] 10000    21
```

```
yt$shape
```

```
[1] 10000
```

```
beta$shape
```

```
[1] 21
```

One step by hand

```
loss = ((Xt %*% beta - yt)^2)$sum()
loss
```

```
torch_tensor
1277300.5
[ CPUFloatType{} ][ grad_fn = <SumBackward0> ]
```

```
loss$backward()
beta$grad
```

```
torch_tensor
100000 *
-0.6027
 0.0121
-0.0090
-1.0898
-0.0245
 0.0274
-0.0559
 0.6441
 0.0035
-0.0066
-0.0275
-0.0117
-1.7697
 0.0351
 0.0053
 0.0174
```

```
 0.0002
 0.0205
 0.3062
-0.0344
```

Notice that we never wrote down the gradient ourselves - autograd derived it from the forward pass.

A torch training loop

```
beta = torch_zeros(p + 1, dtype = torch_float(), requires_grad = TRUE)
for (i in 1:200) {
  loss = ((Xt %*% beta - yt)^2)$sum()
  loss$backward()
  with_no_grad({
    beta$sub_(1e-5 * beta$grad)
    beta$grad$zero_()
  })
}
```

```
loss$item()
```

```
[1] 9859.225
```

```
deviance(lm_fit)
```

```
[1] 9859.224
```

```
beta$detach() |> as_array() |> round(2)
```

```
 [1]  3.03  0.01  0.00  5.22 -0.01  0.00  0.00 -3.10 -0.01  0.00 -0.01
[12] -0.01  8.69 -0.02 -0.01  0.02  0.02  0.01 -1.48  0.00 -0.01
```

```
lm_fit |> coef() |> round(2) |> unname()
```

```
 [1]  3.03  0.01  0.00  5.22 -0.01  0.00  0.00 -3.10 -0.01  0.00 -0.01
[12] -0.01  8.69 -0.02 -0.01  0.02  0.02  0.01 -1.48  0.00 -0.01
```

`with_no_grad({...})` tells autograd "don't track these operations" - we need it because the parameter update is *not*

nn_module

What is `nn_module`?

`nn_module()` creates a model class with two key methods:

- `initialize()` - builds the parameters and submodules
- `forward()` - defines how an input is transformed into an output

Any tensor wrapped in `nn_parameter()` (or any `nn_*` submodule like `nn_linear`) is automatically registered as a parameter of the model and will show up in `model$parameters`.

A first linear regression model

```
linear_model = nn_module(
  initialize = function(k) {
    self$beta = nn_parameter(torch_zeros(k))
  },
  forward = function(X) {
    X %*% self$beta
  }
)
```

```
(m = linear_model(p + 1))
```

```
An `nn_module` containing 21 parameters.

── Parameters ──────────────────────────────────────────────────────────────────
• beta: Float [1:21]
```

```
m$parameters
```

```
$beta
torch_tensor
 0
 0
 0
 0
 0
 0
 0
 0
 0
```

```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```

Training loop with an optimizer

Rather than updating `beta` by hand, we can hand the model's parameters to a torch optimizer and let it do the work:

```
m = linear_model(p + 1)
opt = optim_sgd(m$parameters, lr = 1e-5)

for (i in 1:200) {
  opt$zero_grad()
  loss = ((m(Xt) - yt)^2)$sum()
  loss$backward()
  opt$step()
}
```

```
loss$item()
```

```
[1] 9859.225
```

```
round(as_array(m$beta$detach()), 2)
```

```
 [1]  3.03  0.01  0.00  5.22 -0.01  0.00  0.00 -3.10 -0.01  0.00 -0.01
[12] -0.01  8.69 -0.02 -0.01  0.02  0.02  0.01 -1.48  0.00 -0.01
```

Using `nn_linear`

In practice we rarely build a `nn_parameter` by hand - torch ships with layer modules that handle parameter creation, weight initialization, and the bias term for us. `nn_linear(in_features, out_features)` implements $y = XW^T + b$:

```
linear_model = nn_module(
  initialize = function(in_features) {
    self$linear = nn_linear(in_features, 1)
  },
  forward = function(X) {
    self$linear(X)
  }
)
```

Because `nn_linear` already includes a bias term, we drop the column of 1s and feed in `X` directly. We do need to reshape `y` to `(n, 1)` so it matches the shape of the model's output:

```
Xt = torch_tensor(X, dtype = torch_float())
yt = torch_tensor(matrix(y, ncol = 1), dtype = torch_float())
```

Linear parameters

Constructing the model registers a `weight` matrix (shape `(out_features, in_features)`) and a `bias` vector (shape `(out_features)`) as parameters:

```
1 (m = linear_model(p))
```

```
An `nn_module` containing 21 parameters.

── Modules ─────────────────────────────────────────────────────────────────────
• linear: <nn_linear> #21 parameters
```

```
1 m$parameters
```

```
$linear.weight
torch_tensor
Columns 1 to 10 0.0777 -0.0754  0.1269  0.0282  0.1229  0.1435 -0.0987  0.0813 -0.0967  0.0701

Columns 11 to 20-0.1168  0.1034  0.0453 -0.0875 -0.1097  0.0579  0.2086  0.1073 -0.0216 -0.0109
[ CPUFloatType{1,20} ][ requires_grad = TRUE ]

$linear.bias
torch_tensor
 0.1271
[ CPUFloatType{1} ][ requires_grad = TRUE ]
```

Fitting

```
m = linear_model(p)
opt = optim_sgd(m$parameters, lr = 0.05)

losses = numeric(100)
for (i in 1:100) {
  opt$zero_grad()
  loss = nnf_mse_loss(m(Xt), yt)
  loss$backward()
  opt$step()
  losses[i] = loss$item()
}
```

```
c(torch = loss$item(), lm = deviance(lm_fit) / n)
```

```
    torch        lm
0.9859226 0.9859224
```

```
c(torch_intercept = as.numeric(m$linear$bias), lm_intercept = unname(coef(lm_fit)[1]))
```

```
torch_intercept    lm_intercept
       3.025901        3.025989
```

`nnf_mse_loss()` defaults to mean reduction (versus the manual sum we used before), which is why the learning rate is

Loss curve