Stochastic Gradient Descent

Computational Statistics

Adam Gorm Hoffmann

Setup for stochastic gradient descent

- Estimate parameter in Θ based on a large number of observations from $\mathcal{X}$.
- With some loss function $L : \mathcal{X} \times \Theta \to \mathbb{R}$ the parameter can be estimated by minizing the risk (expected loss) $H(\theta) = E(L(X, \theta))$ over $\theta \in \Theta$.

Gradient Descent

Suppose that we have N iid. observations $X_1, \ldots, X_N$.

A gradient descent using all the data, would be

$$\theta_{n+1} = \theta_n - \gamma_n \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta L(X_i, \theta_n)$$

with some decay schedule (γ_n).

Problem: If the number of observations, N, is very large, or if the gradient $\nabla_\theta L(X_i, \theta_n)$ is very expensive to compute, then this will be a very time consuming algorithm.

Stochastic Gradient Descent (mini-batch)

Idea: Only use some of the data in each update.

Let M be the desired number of data points used for each update, and let I_n be a random sample of M indices from $\{1, \ldots, N\}$. Then we compute the n'th update using the M observations given by I_n.

$$\theta_{n+1} = \theta_n - \gamma_n \frac{1}{M} \sum_{i \in I_n} \nabla_\theta L(X_i, \theta_n).$$

Epochs

We further split the computations into *epochs* where (almost) all data is seen once in each epoch. In epoch n we sample $I_{n,j}$ such that

$$\bigcup_{j=1}^{\mathrm{floor}(N/M)} I_{n,j} \approx \{1, \ldots, N\}.$$

(If $\mathrm{floor}(N/M) < N/M$ then we leave out the remaining data points for simplicity.)

The learning rate γ_n stays the same within epoch n.

Implementation: highest level

```r
sgd <- function(par, n_obs, decay_schedule, epoch = batch,
                n_iter = 100, sampler = sample, cb = NULL, ...)
{
    learning_rates <- decay_schedule(1:n_iter)
    for(k in 1:n_iter) {
        if(!is.null(cb))
            cb()
        epoch_sample <- sampler(n_obs)
        par <- epoch(par, epoch_sample, learning_rates[k], grad, ...)
    }
    if (!is.null(cb))
        cb()
    par
}
```

Implementation: mini-batches

```r
batch <- function(par, epoch_sample, learning_rate, grad,
                  batch_size = 50, ...)
{
    n_batches <- floor(length(epoch_sample) / batch_size)
    for(j in 0:(n_batches - 1)) {
        i <- epoch_sample[(j * batch_size + 1):
                          (j * batch_size + batch_size)]
        par <- par - learning_rate * grad(par, i, ...)
    }
    par
}
```

Other update schemes:

- Momentum: remember last direction.
- Adam: rescale gradient components to avoid narrow valley.

Specific problem: logistic regression model

$p_i(\beta) = P(Y_i = 1 \mid X_i = x_i)$ with log-odds

$$\log \frac{p_i(\beta)}{1 - p_i(\beta)} = f\left(x_i \mid \beta\right) = \left(\varphi_1\left(x_i\right), \ldots, \varphi_p\left(x_i\right)\right)^T \beta$$

where $\beta \in \mathbb{R}^p$ and with fixed basis functions $\varphi_1, \ldots, \varphi_p : \mathbb{R} \to \mathbb{R}$.

We want to optimize the penalized negative log likelihood

$$H(\beta) = -\frac{1}{N} \sum_{i=1}^{N} \left(y_i \log p_i(\beta) + (1 - y_i) \log\left(1 - p_i(\beta)\right)\right) + \lambda \left\| f_\beta'' \right\|_2^2$$

over $\beta \in \mathbb{R}^p$.

Deriving $D_\beta H$

Let $\Phi_{ij} = \varphi_j(x_i)$ and

$$\mathbf{\Omega}_{ij} = \left\langle \varphi_i'', \varphi_j'' \right\rangle = \int \varphi_i''(z)\varphi_j''(z)dz$$

and $h(z) = -\log(1 + \exp(z))$.

Deriving $D_\beta H$

Then

$$\log p(\beta) = \Phi\beta + h(\Phi\beta)$$
$$\log(\mathbf{1} - p(\beta)) = h(\Phi\beta).$$

so

$$\begin{aligned} H(\beta) &= -\frac{1}{N}\mathbf{y}^T(\Phi\beta + h(\Phi\beta)) - \frac{1}{N}(\mathbf{1} - \mathbf{y})^T h(\Phi\beta) + \lambda\beta^T\Omega\beta \\ &= \frac{1}{N}\left(-\mathbf{y}^T\Phi\beta - \mathbf{1}^T h(\Phi\beta)\right) + \lambda\beta^T\Omega\beta \end{aligned}$$

with derivatives

$$D_\beta H(\beta) = -\frac{1}{N}\Phi^T(\mathbf{y} - p(\beta)) + 2\lambda\boldsymbol{\Omega}\beta$$
$$D_\beta^2 H(\beta) = \frac{1}{N}\Phi^T W\Phi + 2\lambda\Omega$$

where $W(\beta) = \mathrm{diag}(p(\beta))\,\mathrm{diag}(1 - p(\beta))$.

Implementation of H

$$H(\beta) = -\frac{1}{N}(\mathbf{y}^T \log p(\beta) + (\mathbf{1} - \mathbf{y})^T \log(\mathbf{1} - p(\beta)) + \lambda \beta^T \Omega \beta$$

```r
get_H <- function(lambda, x_vec, y_vec, knots, inner_knots)
{
    force_all_args()
    N <- length(x_vec)
    Omega_mat <- Omega(inner_knots)
    Phi_mat <- Phi(x_vec, knots)
    function(beta) {
        p_beta <- p(beta, Phi_mat[i, ])
        -1/N * (crossprod(y_vec, log(p_beta)) +
                crossprod(1 - y_vec, log(1 - p_beta))) +
            lambda * crossprod(beta, Omega_mat %*% beta)
    }
}
```

Forcing evaluation of arguments to avoid lazy evaluation problems

```r
force_all_args <- function()
    as.list(parent.frame())
```

Implementation of $D_\beta H$

$$D_\beta H(\beta) = -\frac{1}{N}\Phi^T(\mathbf{y} - p(\beta)) + 2\lambda\boldsymbol{\Omega}\beta$$

```r
get_grad_H <- function(lambda, x_vec, y_vec, knots, inner_knots)
{
    force_all_args()
    Phi_mat <- Phi(x_vec, knots)
    unpen_grad <- function(beta, i) {
        n_i <- length(i)
        Phi_i <- matrix(Phi_mat[i, ], nrow = n_i)
        -crossprod(Phi_i, y_vec[i] - p(beta, Phi_i)) / n_i
    }
    if (lambda > 0) {
        two_lambda_Omega <- 2 * lambda * Omega(inner_knots)
        function(beta, i)
            unpen_grad(beta, i) + two_lambda_Omega %*% beta
    } else
        unpen_grad
}
```

- Precompute as much as possible.
- Leave out penalization term if $\lambda = 0$ to avoid wasting time on matrix multiplication for a 0-matrix.

Implementation: components for H and $D_\beta \ell$

```r
inv_logit <- function(x)
    exp(x) / (1 + exp(x))

get_knots <- function(inner_knots)
    sort(c(rep(range(inner_knots), 3), inner_knots))

Phi <- function(x_vec, knots)
    splineDesign(knots, x_vec)

p <- function(beta, phi_mat)
    inv_logit(phi_mat %*% beta)

Omega <- function(inner_knots)
{
    knots <- sort(c(rep(range(inner_knots), 3), inner_knots))
    d <- diff(inner_knots)  # The vector of knot differences; b - a
    g_ab <- splineDesign(knots, inner_knots, derivs = 2)
    knots_mid <- inner_knots[-length(inner_knots)] + d / 2
    g_ab_mid <- splineDesign(knots, knots_mid, derivs = 2)
    g_a <- g_ab[-nrow(g_ab), ]
    g_b <- g_ab[-1, ]
    (crossprod(d * g_a,  g_a) +
     4 * crossprod(d * g_ab_mid, g_ab_mid) +
     crossprod(d * g_b, g_b)) / 6
}

d2f_two_norm_squared <- function(beta, inner_knots)
    crossprod(beta, Omega(inner_knots) %*% beta)
```

Decay schedule

```r
decay_scheduler <- function(gamma0 = 1, a = 1, K = 1, gamma1, n1)
{
    force_all_args()
    if (!missing(gamma1) && !missing(n1))
        K <- n1^a * gamma1 / (gamma0 - gamma1)
    b <- gamma0 * K
    function(n)
        b / (K + n^a)
}
```

Newton's method: highest level

```r
newton <- function(init_guess, x, y, max_iter = 100, epsilon = 1e-5,
                   lambda = 0, cb = NULL)
{
    n_param <- length(init_guess)
    par_new <- init_guess
    newton_update <- get_newton_update(n_param, x, y, lambda)
    for (i in 1:max_iter) {
        if (!is.null(cb))
            cb()
        par_old <- par_new
        par_new <- newton_update(par_old)
        if (sum((par_new - par_old)^2) <=
            epsilon * (sum(par_new^2) + epsilon))
            break
    }
    if (!is.null(cb))
        cb()
    par_new
}
```

Newton's method: update step

The solution to

$$D_\beta H(\beta^0) + (\beta - \beta^0)^T D_\beta^2 H(\beta^0) = 0$$

is given by

$$\beta = (\Phi^T W \Phi + N\lambda\Omega)^{-1} \Phi^T W z_0$$

where

$$z_0 = \Phi\beta^0 + W(\beta^0)^{-1}(y - p(\beta^0))$$

and

$$W(\beta) = \mathrm{diag}(p(\beta))\,\mathrm{diag}(1 - p(\beta)).$$

Newton's method: update step

```r
get_newton_update <- function(n_param, x, y, lambda)
{
    force_all_args()
    n_sim <- length(x)
    inner_knots <- seq(min(x), max(x), length.out = n_param - 2)
    knots <- get_knots(inner_knots)
    Phi_mat <- Phi(x, knots)
    tPhi <- t(Phi_mat)
    Omega_mat <- Omega(inner_knots)

    function(beta)
    {
        p_vec <- as.vector(p(beta, Phi_mat))
        p_prod <- p_vec * (1 - p_vec)
        W <- diag(p_prod)
        W_inv <- diag(1 / p_prod)
        z <- Phi_mat %*% beta + W_inv %*% (y - p(beta, Phi_mat))
        solve(crossprod(Phi_mat, W %*% Phi_mat) +
              n_sim * lambda * Omega_mat) %*%
            crossprod(Phi_mat, W %*% z)
    }
}
```

Simulating data for initial tests

```r
set.seed(2021)
n_beta <- 5
n_sim <- 1000
x <- seq(0, 1, length.out = n_sim)
inner_knots <- seq(min(x), max(x), length.out = n_beta - 2)
knots <- get_knots(inner_knots)
beta <- c(-1, 1, 3, -7, 1)
y <- as.numeric(runif(n_sim) < p(beta, x, knots))
beta_init <- rep(1, n_beta)
```

Getting H and $D_\beta H$ with $\lambda = 0$

- We set $\lambda = 0$, corresponding to usual unpenalized logistic regression, so we can compare results with `glm`.

```
grad_H <- get_grad_H(lambda = 0, x, y, knots, inner_knots)
H <- get_H(lambda = 0, x, y, knots, inner_knots)
```

- We confirm that our gradient implementation works by comparing to a numerical approximation.

```
all.equal(numDeriv::grad(H, beta),
          as.vector(grad_H(beta, 1:length(x))), tolerance = 1e-8)

## TRUE
```

Run methods

```r
set.seed(2021)

## Stochastic gradient descent
sgd_batch_tracer <- tracer(c("objective", "par"),
                           expr = quote(objective <- H(par)),
                           N = 0)
ds <- decay_scheduler(gamma0 = 0.8, gamma1 = 0.008, n1 = 90)
beta_fit_1 <- sgd(par = beta_init,
                  grad = grad_H,
                  n_obs = n_sim,
                  decay_schedule = ds,
                  epoch = batch,
                  n_iter = 200,
                  batch_size = 1,
                  cb = sgd_batch_tracer$tracer)

## Newton
newton_tracer <- tracer(c("objective", "par_new"),
                        expr = quote(objective <- H(par_new)))
newton_fit <- newton(init_guess = beta_init,
                     x = x,
                     y = y,
                     max_iter = 100,
                     cb = newton_tracer$tracer)
```

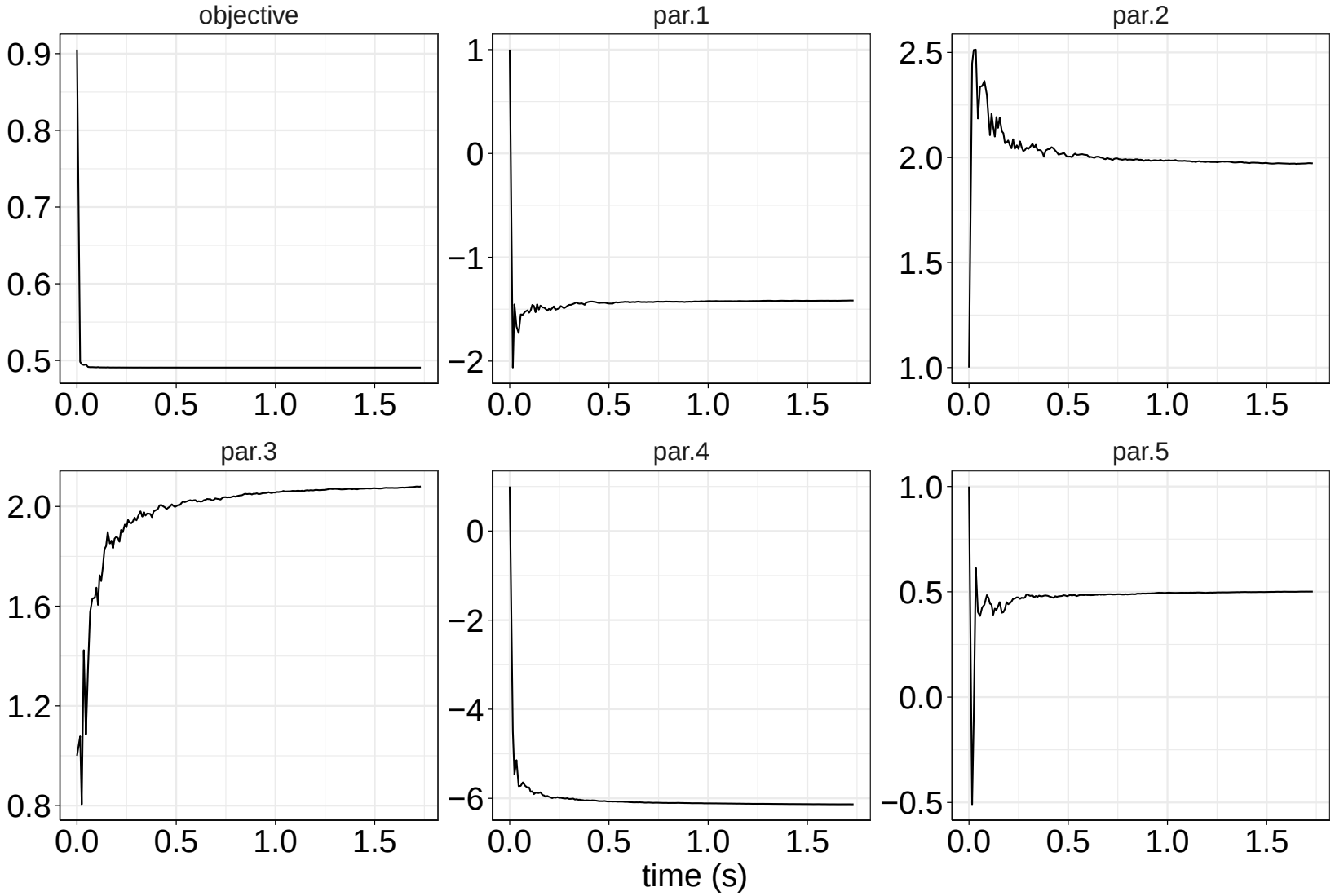
Comparing results to `glm` (IWLS)

```r
glm(y ~ splineDesign(knots, x) - 1, family = binomial)$coefficients
```

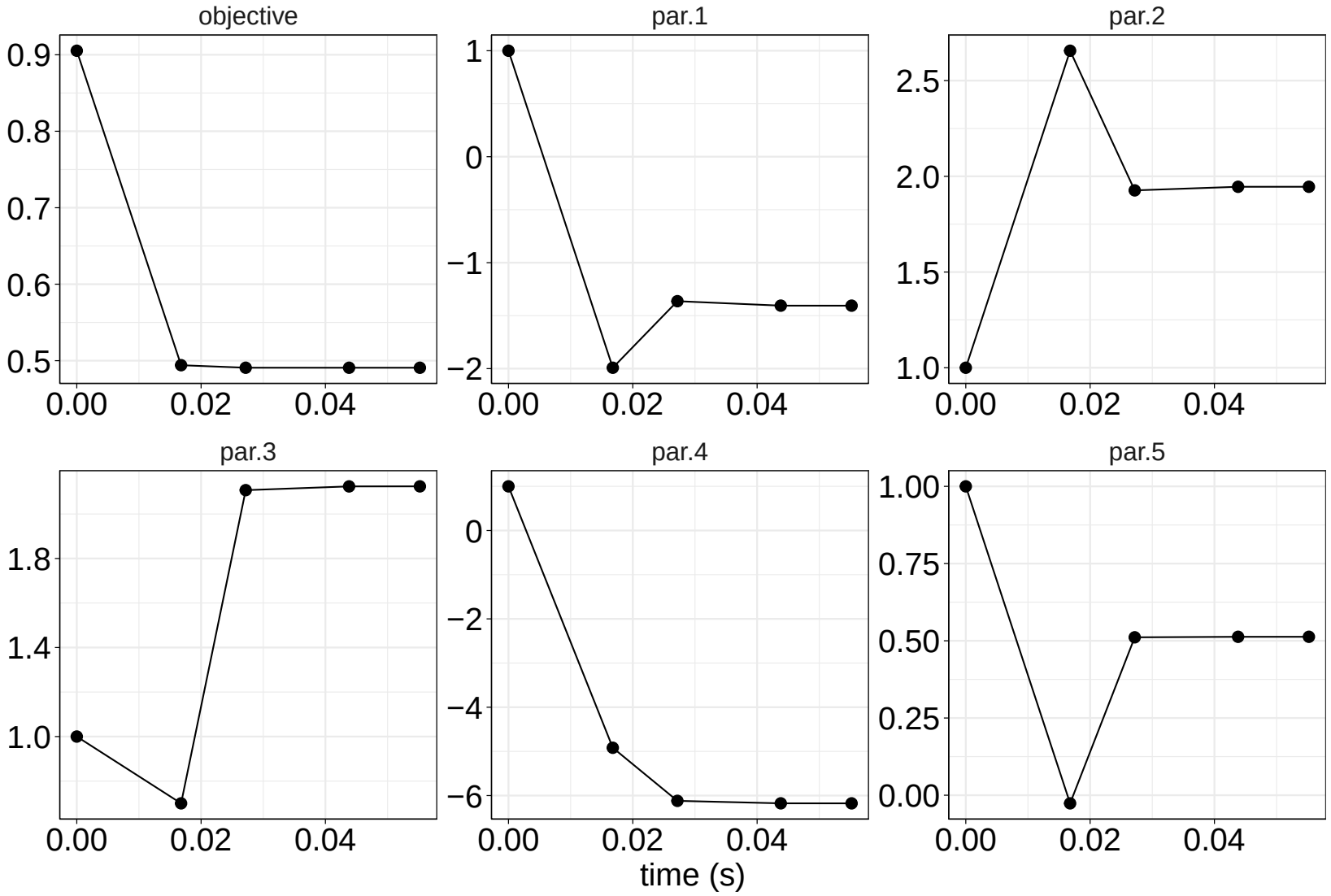
sgd	newton	glm
-1.4172427	-1.4058146	-1.4058146
1.9725305	1.9454677	1.9454677
2.0802703	2.1241283	2.1241283
-6.1350949	-6.1761810	-6.1761811
0.5009762	0.5129334	0.5129334

The methods yield approximately the same results, indicating that our implementations work with $\lambda = 0$.

SGD convergence (200 iterations)

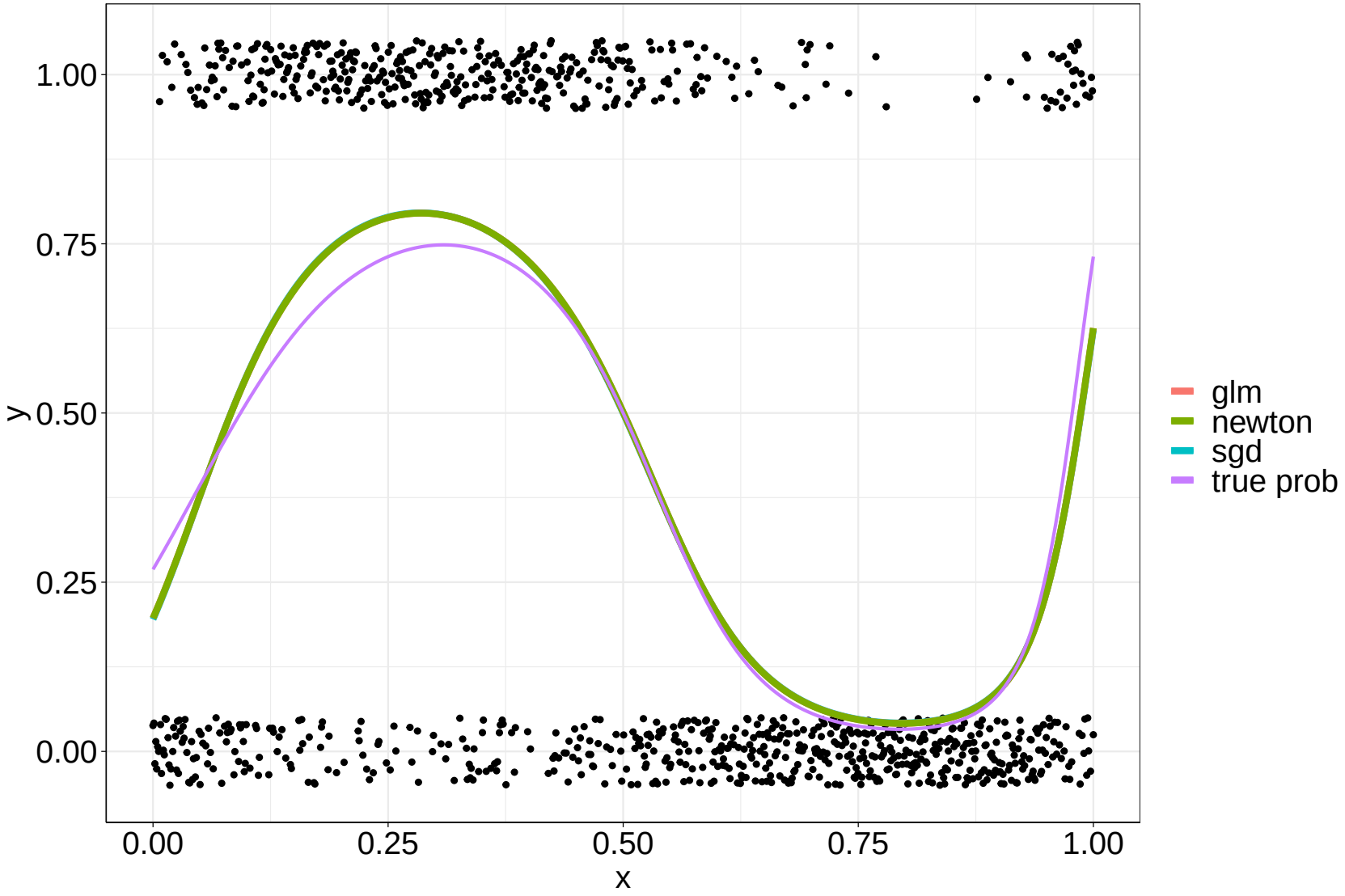


Newton convergence (4 iterations)



- Okay convergence two steps from initial guess!

Predictions (the same for all three methods)



$\lambda > 0$: Run methods

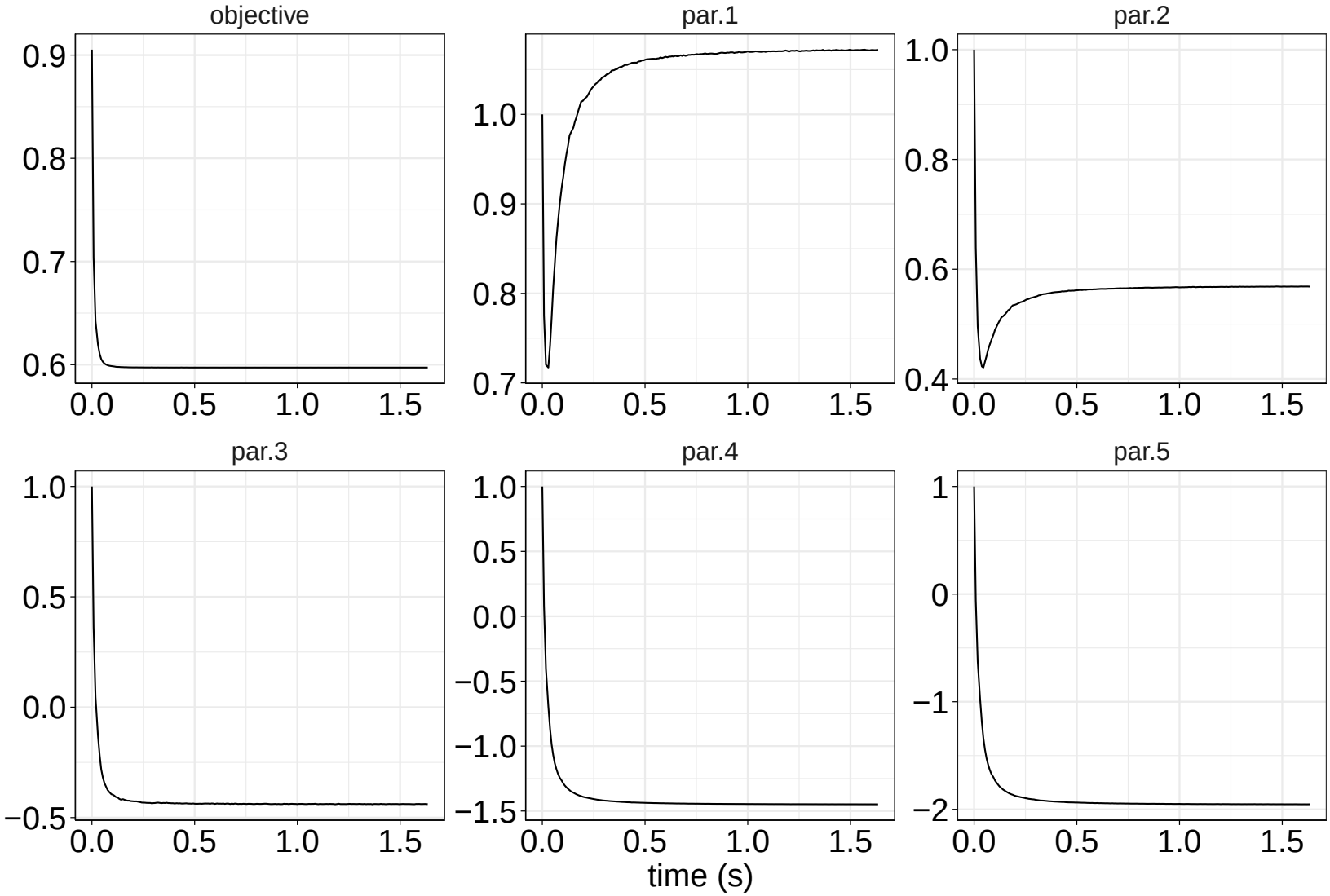
```r
grad_H <- get_grad_H(lambda = 0.2, x, y, knots, inner_knots)
H <- get_H(lambda = 0.2, x, y, knots, inner_knots)
opt_fit <- optim(par = beta_init, fn = H, method = "BFGS",
                 control = list(reltol = 1e-16))$par
## Stochastic gradient descent
sgd_batch_tracer <- tracer(c("objective", "par"),
                           expr = quote(objective <- H(par)),
                           N = 0)
beta_fit_lam <- sgd(par = beta_init, grad = grad_H, n_obs = n_sim,
                    decay_schedule = decay_scheduler(gamma0 = 0.015,
                                                     gamma1 = 0.001,
                                                     n1 = 90),
                    epoch = batch, n_iter = 200, batch_size = 1,
                    cb = sgd_batch_tracer$tracer)
## Newton
newton_tracer <- tracer(c("objective", "par_new"),
                        expr = quote(objective <- H(par_new)))
newton_fit_lam <- newton(init_guess = beta_init, x = x, y = y,
                         max_iter = 100, lambda = 0.2,
                         cb = newton_tracer$tracer)
```

$\lambda > 0$: Comparing β estimates

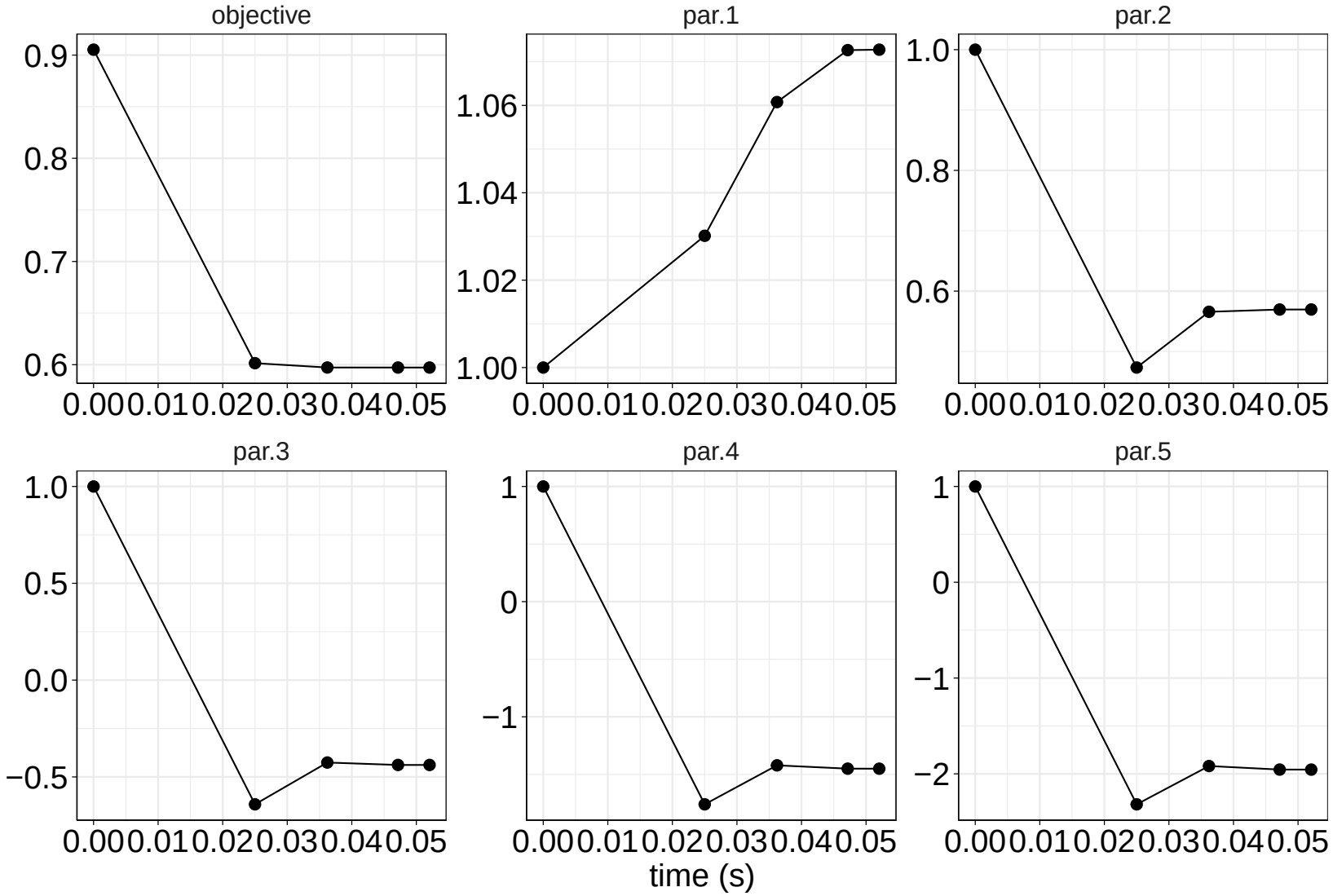
sgd	newton	optim
1.0719206	1.0727550	1.0735588
0.5686376	0.5695332	0.5695413
-0.4386992	-0.4381971	-0.4391370
-1.4484768	-1.4505377	-1.4501226
-1.9531086	-1.9558413	-1.9551820

- Approximately the same results!
- Strongly indicates that our implementations work.

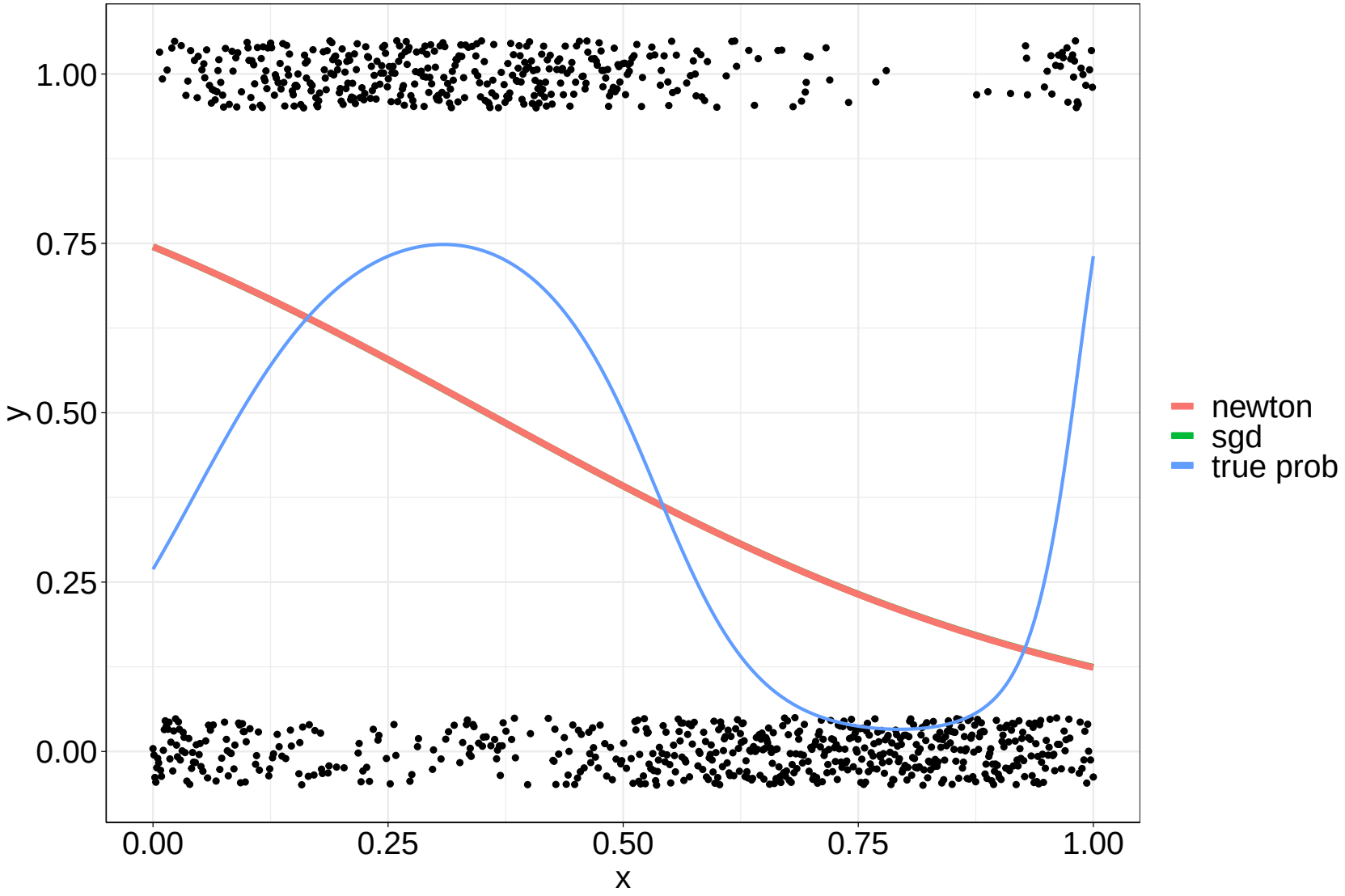
$\lambda > 0$: SGD convergence (200 iter.)



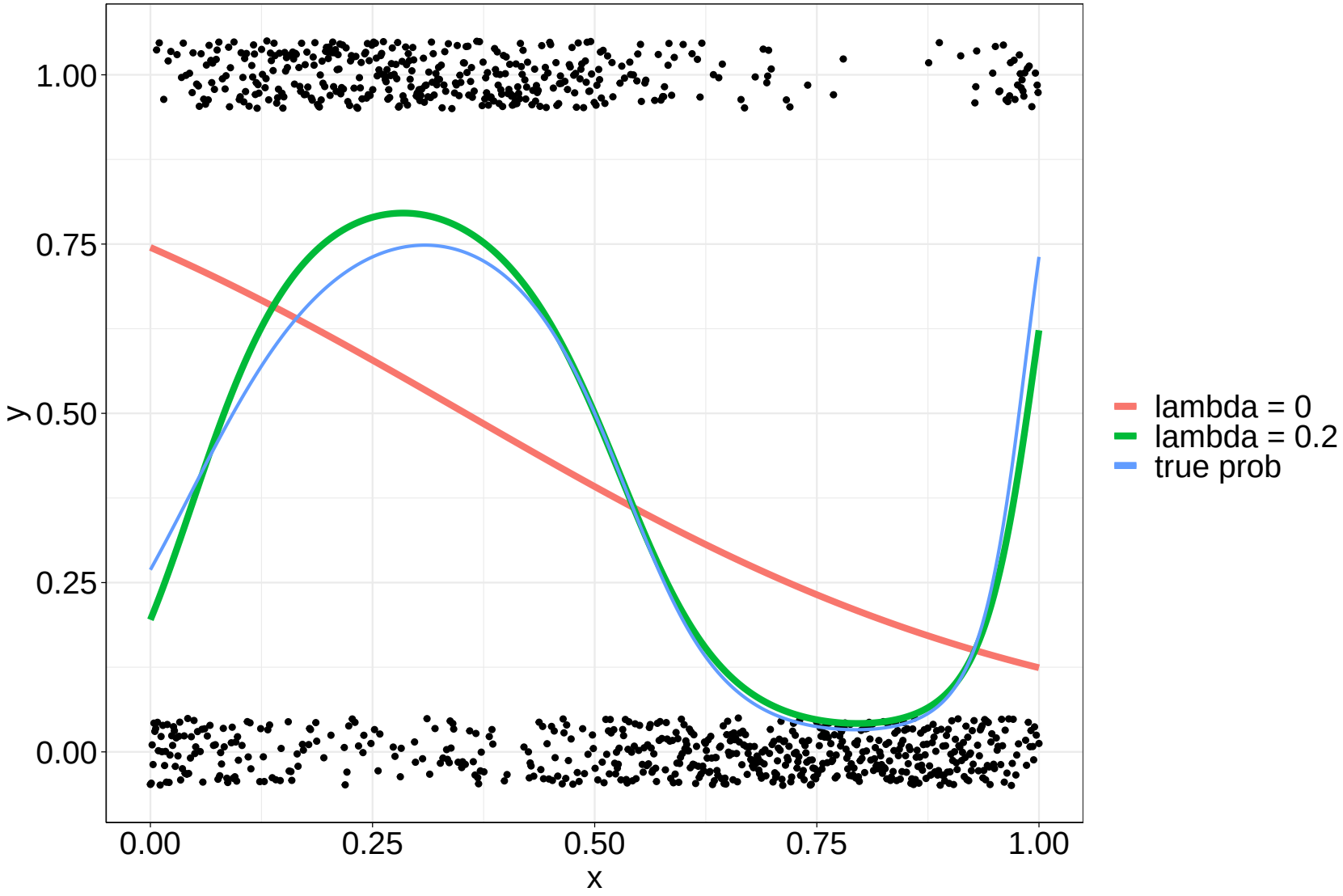
$\lambda > 0$: Newton convergence (4 iter.)



$\lambda > 0$: Fitted probabilities



Comparing prob. fits with $\lambda = 0$ and $\lambda = 0.2$



- As expected, $\lambda > 0$ gives a less volatile fit.
- But, in this case, $\lambda = 0.2$ oversmoothes.

Conclusion: Our `sgd` and `newton` implementations work for both $\lambda = 0$ and $\lambda > 0$.

Can we improve the speed of the `sgd` implementation?

Profiling: `sgd` and `batch`

```
sgd <- function(par, grad, n_obs, decay_schedule, epoch = batch,
                n_iter = 100, sampler = sample, cb = NULL, ...)
{
    learning_rates <- decay_schedule(1:n_iter)
    for(k in 1:n_iter) {
        if(!is.null(cb))                                                    380
            cb()
        epoch_sample <- sampler(n_obs)                                      340
        par <- epoch(par, epoch_sample, learning_rates[k], grad, ...)     47830
    }
    if (!is.null(cb))
        cb()
    par
}
```

```
batch <- function(par, epoch_sample, learning_rate, grad,
                  batch_size = 50, ...)
{
    n_batches <- floor(length(epoch_sample) / batch_size)                    20
    for(j in 0:(n_batches - 1)) {                                           100
        i <- epoch_sample[(j * batch_size + 1):                            1280
                          (j * batch_size + batch_size)]
        par <- par - learning_rate * grad(par, i, ...)                    46380
    }
    par
}
```

- `grad` is the bottleneck.

Profiling: `grad_H`

```
get_grad_H <- function(lambda, x_vec, y_vec, knots, inner_knots)
{
    force_all_args()
    Phi_mat <- Phi(x_vec, knots)
    unpen_grad <- function(beta, i) {                                  220
        n_i <- length(i)                                              1360
        Phi_i <- matrix(Phi_mat[i, ], nrow = n_i)                    12870
        y_p_diff <- y_vec[i] - p(beta, Phi_i)                        13770
        cp <- crossprod(Phi_i, y_p_diff)                              5260
        -cp / n_i                                                     2010
    }
    if (lambda > 0) {
        two_lambda_Omega <- 2 * lambda * Omega(inner_knots)
        function(beta, i) {                                            150
            two_lambda_Omega_beta <- two_lambda_Omega %*% beta        3100
            unpen_grad(beta, i) + two_lambda_Omega_beta              39790
        }
    } else
        unpen_grad
}
```

- The indexing `Phi_mat[i, ]` is slow in R. Would be faster in C++.
- The rest are all vectorized operations in R and can't be improved much in C++.

Profiling: `p` (not much to do about this one)

```
p <- function(beta, phi_mat)                                  200
{
    phi_beta_prod <- phi_mat %*% beta                        2980
    inv_logit(phi_beta_prod)                                 7130
}
```

What can we do?

- Reimplement the part of the gradient doing slicing in Rcpp, and as little else as possible, since everything is already vectorized.
- (Spoiler: It is actually a very good idea to implement everything in C++ so we can manage memory manually.)

How to do it in an easy way?

- Use RcppArmadillo: Armadillo is a C++ interface to the BLAS and LAPACK linear algebra libraries also used by R.

```
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
```

```cpp
arma::mat
crossprod(const arma::mat& A, const arma::mat& B)
{
  return A.t() * B;
}

arma::colvec
inv_logit(const arma::colvec& x)
{
  return exp(x) / (1 + exp(x));
}

arma::colvec
p(const arma::colvec& beta, const arma::mat& Phi_i)
{
  return inv_logit(Phi_i * beta);
}

// [[Rcpp::export]]
arma::colvec
unpen_grad_cpp(const arma::colvec& beta, arma::uvec i,
               double lambda, const arma::colvec& y,
               const arma::mat& Phi)
{
  i -= 1;                          // 0-indexing instead of 1-indexing.
  return crossprod(Phi.rows(i), p(beta, Phi.rows(i)) - y.elem(i)) / i.n_elem;
}
```

R wrapper for the C++ implementation

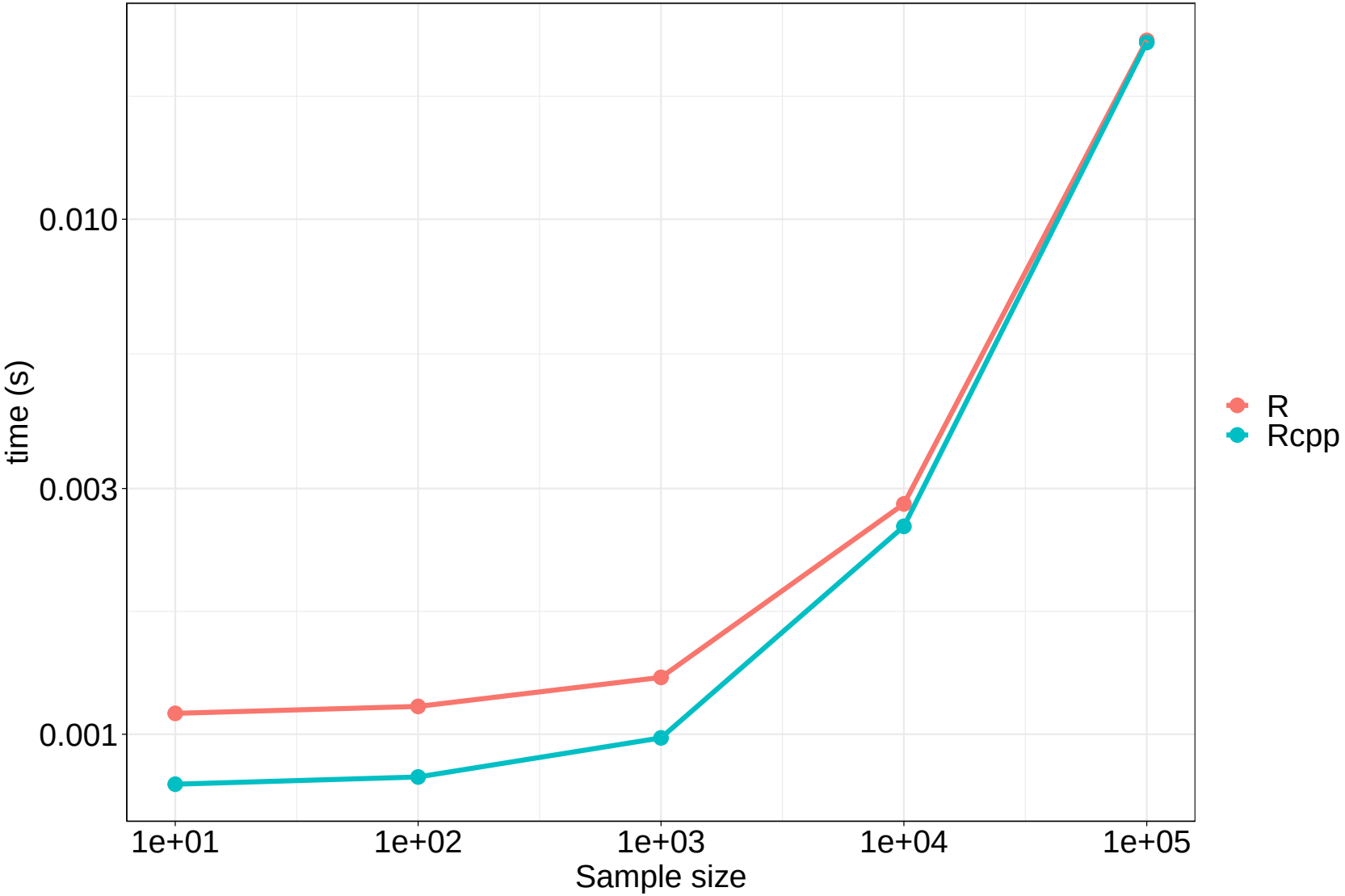
```r
get_grad_H_cpp_wrap <- function(lambda, x_vec, y_vec, knots, inner_knots)
{
    force_all_args()
    Phi_mat <- Phi(x_vec, knots)
    if (lambda > 0) {
        two_lambda_Omega <- 2 * lambda * Omega(inner_knots)
        function(beta, i)
            unpen_grad_cpp(beta, i, lambda, y_vec, Phi_mat) +
                two_lambda_Omega %*% beta
    } else
        function(beta, i)
            unpen_grad_cpp(beta, i, lambda, y_vec, Phi_mat)
}
```

Checking that the C++ gradient works

```r
grad_H <- get_grad_H(lambda = 0.2, x, y, knots, inner_knots)
grad_H_cpp <- get_grad_H_cpp_wrap(lambda = 0.2, x, y, knots, inner_knots)
H <- get_H(lambda = 0.2, x, y, knots, inner_knots)

set.seed(2058)
beta_random <- rnorm(5, sd = 3)

all.equal(grad_H(beta_random, seq_along(x)),
          grad_H_cpp(beta_random, seq_along(x)))

## TRUE

max(abs(grad_H_cpp(beta_random, seq_along(x)) -
        numDeriv::grad(H, beta_random)))

## Approx. 4 * 10^(-9)
```

numDeriv	R	C++
52.05423	52.05423	52.05423
-84.25675	-84.25675	-84.25675
-72.11850	-72.11850	-72.11850
324.56076	324.56076	324.56076
-220.28061	-220.28061	-220.28061

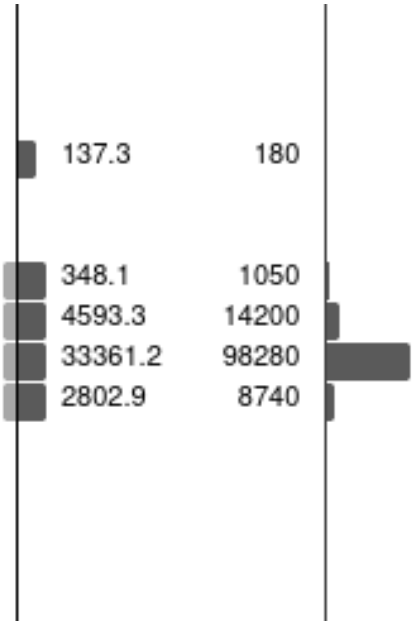
Benchmarking the gradient implementations



- The Rcpp gradient is much faster for small sample sizes, but doesn't improve speed for sample sizes over 10^6.

Profiling `sgd` with Rcpp gradient for 10^6 obs.

```
get_grad_H_cpp_wrap <- function(lambda, x_vec, y_vec, knots, innerknots)
{
    force_all_args()
    Phi_mat <- Phi(x_vec, knots)                                          -7.6    137.3    180
    if (lambda > 0) {
        two_lambda_Omega <- 2 * lambda * Omega(inner_knots)
        function(beta, i) {                                               -279.4   348.1    1050
            two_lambda_Omega_beta <- two_lambda_Omega %*% beta            -4620.5  4593.3   14200
            unpen_grad <- unpen_grad_cpp(beta, i, lambda, y, Phi_mat)     -32651.2 33361.2  98280
            unpen_grad + two_lambda_Omega_beta                            -2732.4  2802.9    8740
        }
    } else
        function(beta, i)
            unpen_grad_cpp(beta, i, lambda, y, Phi_mat)
}
```



- Gradient is still bottle neck.
- Very large memory allocation and deallocation for gradient!
- We must be able to do better with manual memory management in C++.

C++: highest level

```cpp
arma::colvec
sgd_cpp(arma::colvec par, const arma::colvec& learning_rates,
        int n_iter, int batch_size, double lambda, const arma::colvec& y,
        const arma::mat& Phi, const arma::mat& Omega)
{
  int n_obs = y.n_elem;
  arma::uvec epoch_order(n_obs);
  for (int k = 0; k < n_iter; k++) {
    epoch_order = arma::randperm(n_obs);
    batch(par, learning_rates[k], batch_size, lambda,
          n_obs, epoch_order, y, Phi, Omega);
  }
  return par;
}
```

- We only allocate memory for `Phi`, `Omega`, `y`, and the `learning_rates` once, right when they are copied from R to C++.
- We pass them around by reference inside C++.

C++: batch epoch

```
void
batch(arma::colvec& par, double learning_rate, int batch_size,
      double lambda, int n_obs, arma::uvec& epoch_order,
      const arma::colvec& y, const arma::mat& Phi, const arma::mat& Omega)
{
  int lower_i = 0;
  int upper_i = batch_size - 1;
  int n_batches = floor(n_obs / batch_size);
  for (int j = 0; j < n_batches; j++) {
    par = par - learning_rate * grad_cpp(par,
                                         epoch_order.subvec(lower_i, upper_i),
                                         lambda, y, Phi, Omega);
    lower_i += batch_size;
    upper_i += batch_size;
  }
}
```

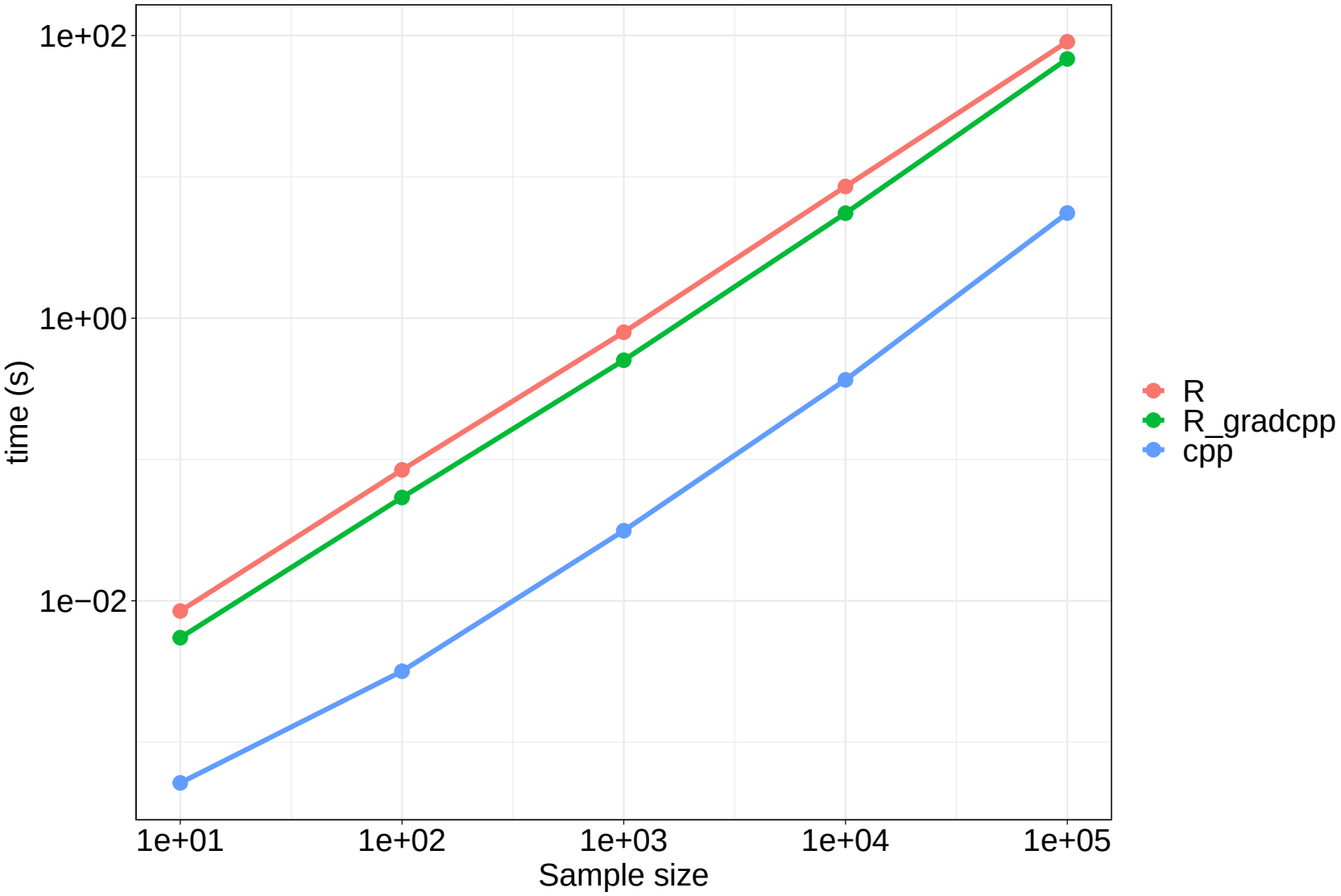
C++: gradient

```cpp
/* Version with 0-indexing for use in C++ */
arma::colvec
unpen_grad(const arma::colvec& beta, arma::uvec i,
           double lambda, const arma::colvec& y,
           const arma::mat& Phi)
{
  return -crossprod(Phi.rows(i), p(beta, Phi.rows(i)) - y.elem(i)) / i.n_elem;
}

arma::colvec
grad_cpp(const arma::colvec& beta, arma::uvec i,
         double lambda, const arma::colvec& y,
         const arma::mat& Phi, const arma::mat& Omega)
{
  if (lambda > 0)
    return unpen_grad(beta, i, lambda, y, Phi) + 2 * lambda * Omega;
  else
    return unpen_grad(beta, i, lambda, y, Phi);
}
```

C++: R wrapper

```r
sgd_cpp_wrap <- function(par, x, y, decay_schedule, inner_knots,
                         lambda = 0, n_iter = 100, batch_size = 1)
{
    learning_rates <- decay_schedule(1:n_iter)
    Om <- Omega(inner_knots)
    Phi_mat <- Phi(x, get_knots(inner_knots))
    sgd_cpp(par, learning_rates, n_iter, batch_size, lambda, y, Phi_mat, Om)
}
```

Benchmarking full implementations



- Staying in C++ pays off!

Should you use stochastic gradient descent?

Pros

- Can converge faster when the sample size is very large (by taking many small fast steps, instead of large expensive steps).
- Can be used in online learning settings, where the estimate is updated as more data becomes available.

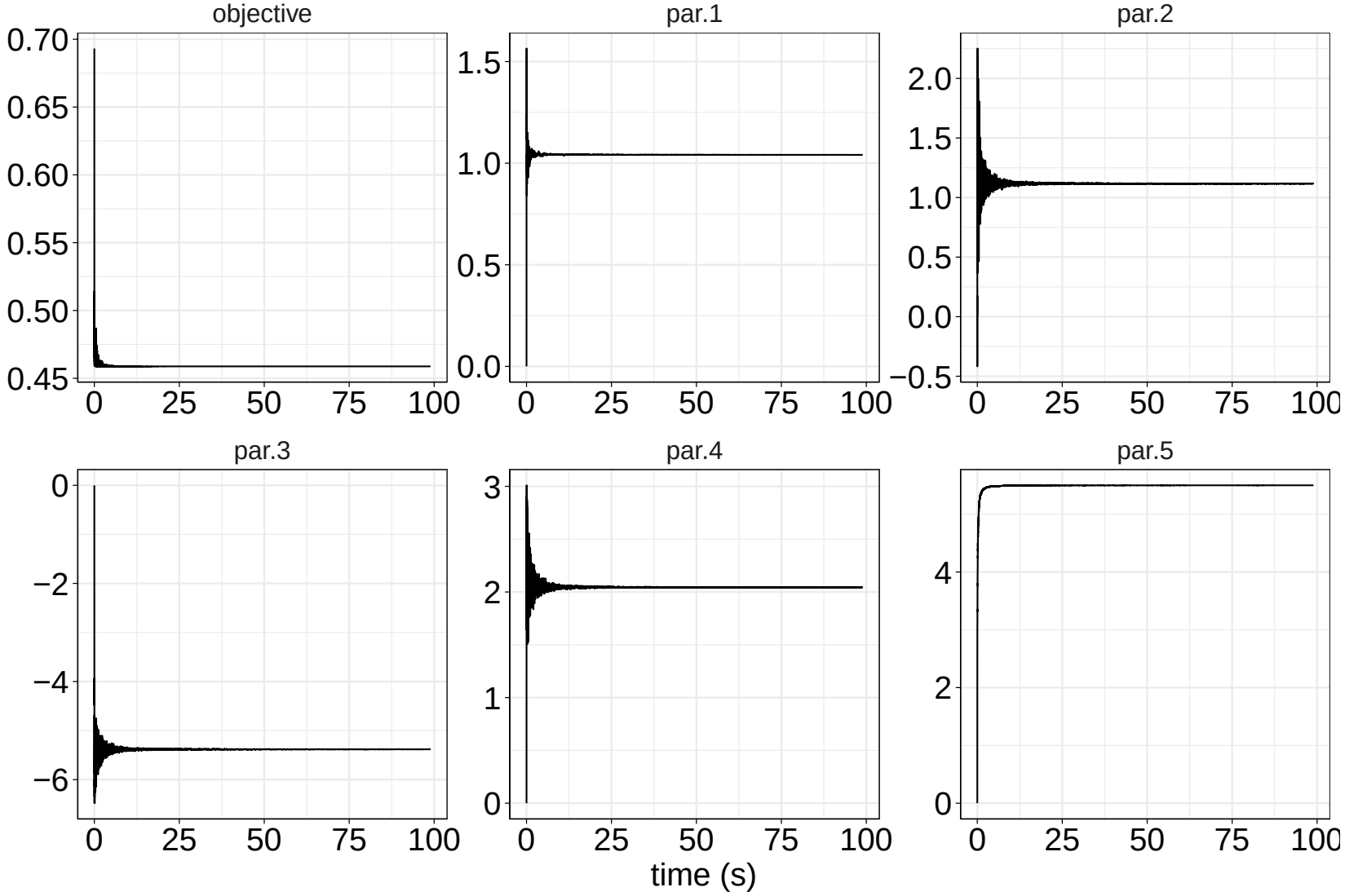
Cons

- Time consuming to tune decay schedule parameters.
- Can still get trapped in local optima (alternative for functions with many local optima: simulated annealing, *e.g.*, in the R package `GenSA`).

Horse data: Fitting

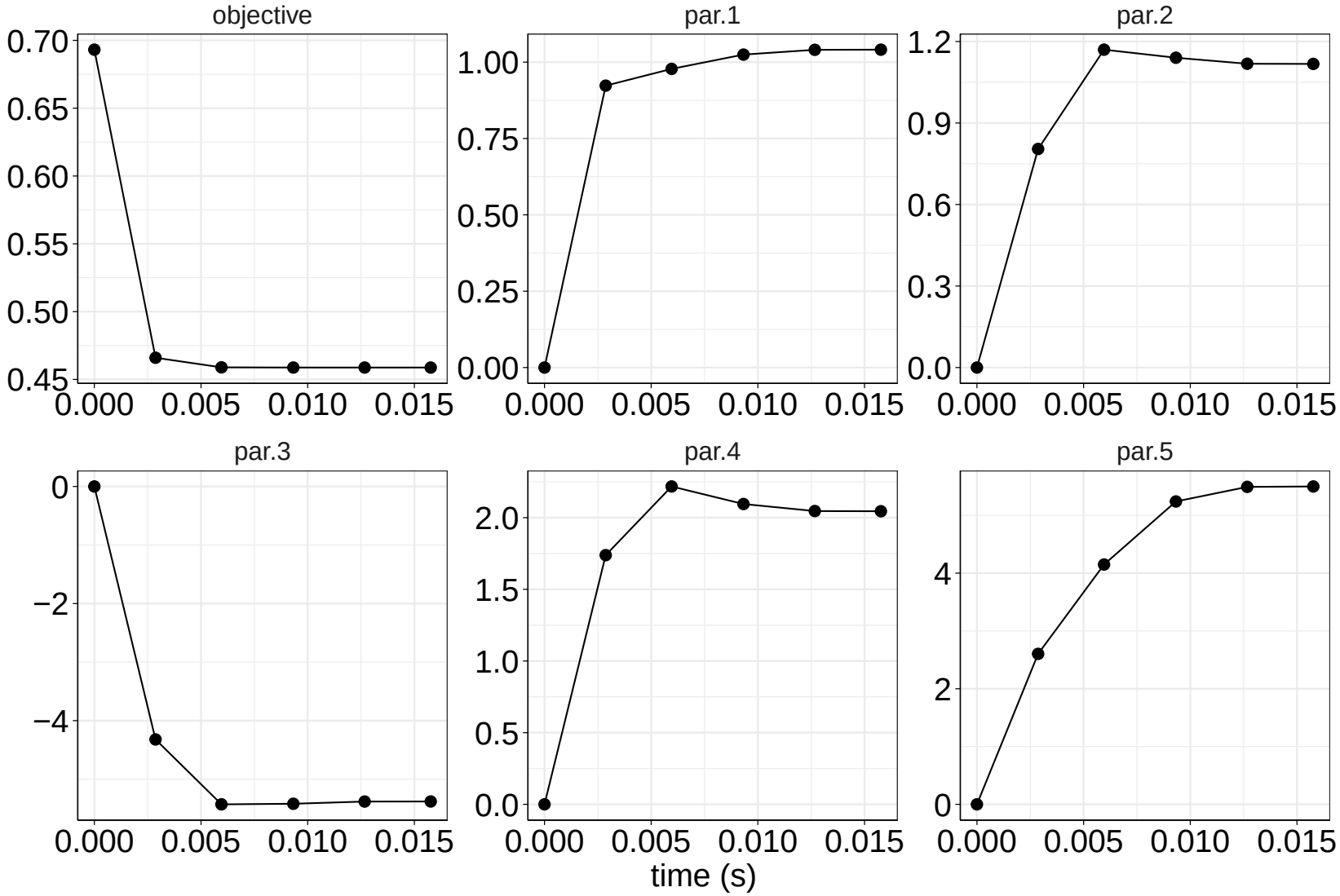
```r
## Load data and scale and center x for faster convergence
x <- scale(horses$Temperature)
y <- horses$dead
n_beta <- 5
beta_init <- rep(0, n_beta)
inner_knots <- seq(min(x), max(x), length.out = n_beta - 2)
knots <- get_knots(inner_knots)

## SGD
grad_H <- get_grad_H(lambda = 0, x, y, knots, inner_knots)
H <- get_H(lambda = 0, x, y, knots, inner_knots)
sgd_batch_tracer <- tracer(c("objective", "par"),
                           expr = quote(objective <- H(par)),
                           N = 0)
beta_fit_horse <- sgd(par = beta_init,
                      grad = grad_H,
                      n_obs = length(x),
                      decay_schedule = decay_scheduler(gamma0 = 1,
                                                       gamma1 = 0.3,
                                                       n1 = 125),
                      epoch = batch,
                      n_iter = 20000,
                      batch_size = 1,
                      cb = sgd_batch_tracer$tracer)

## Newton
newton_tracer <- tracer(c("objective", "par_new"),
                        expr = quote(objective <- H(par_new)))

newton_fit_horse <- newton(init_guess = beta_init,
                           x = x,
                           y = y,
                           max_iter = 100,
                           cb = newton_tracer$tracer)
```

Horse data: SGD convergence (20000 iter.)



- Hard to find a better decay schedule that takes large enough steps, but without fluctuating up and down.

Horse data: Newton convergence (5 iter.)

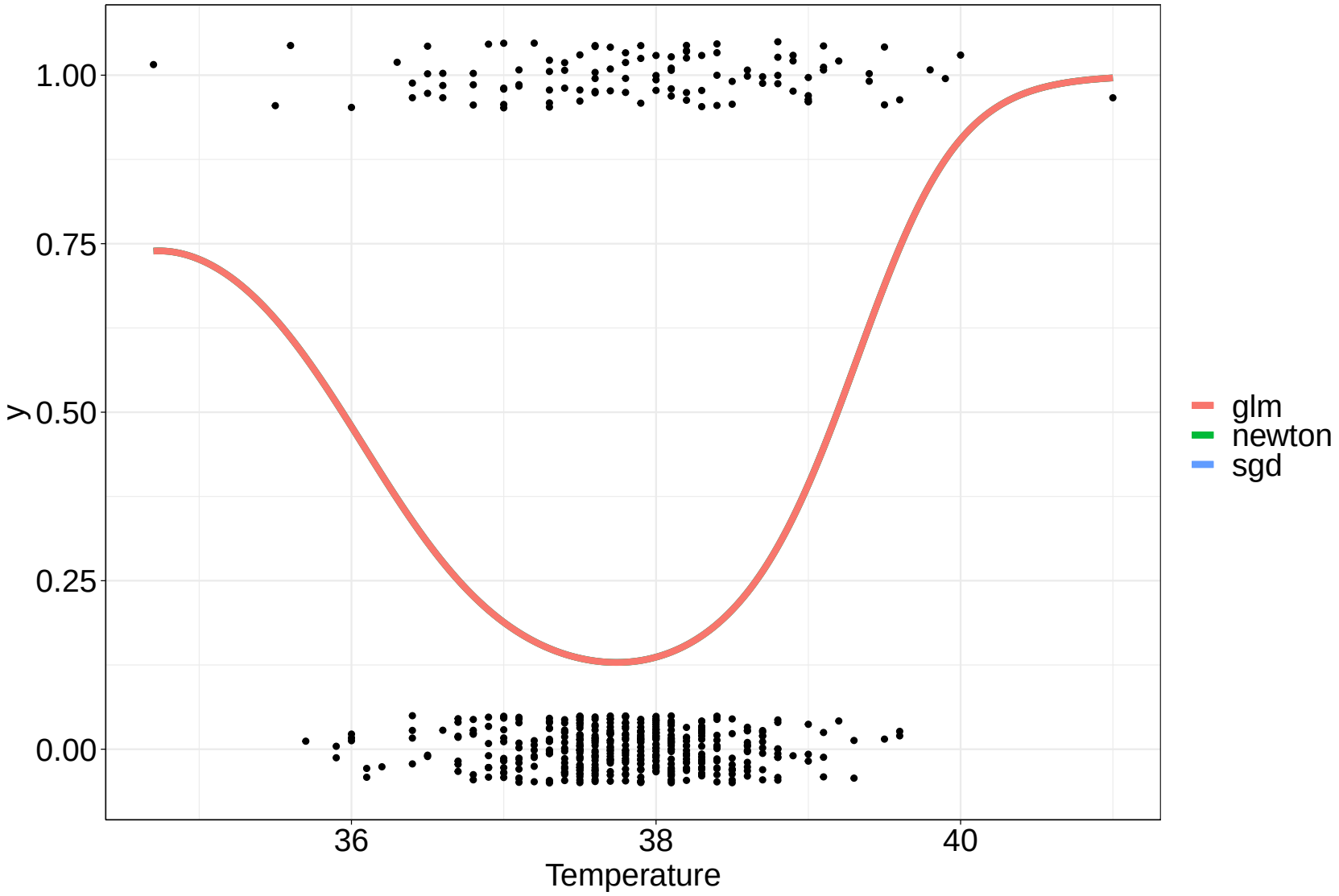


Horse data: Estimates

sgd	newton	glm
1.040949	1.040815	1.040816
1.117849	1.117413	1.117412
-5.380647	-5.381459	-5.381458
2.045161	2.044704	2.044703
5.499648	5.499755	5.499760

- Approximately the same.

Horse data: Fitted probabilities



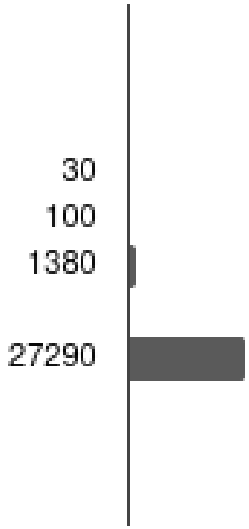
- Approximately the same.

Extra slides

- Gradient derivation.
- Comparing profiling of R and Rcpp.
- Momentum.
- Adam.
- Old slides for convergence and fits.
- `profvis` on `sgd_cpp`

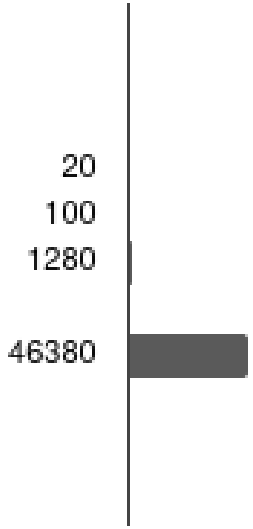
Rcpp implementation

```r
batch <- function(par, epoch_sample, learning_rate, grad,
                  batch_size = 50, ...)
{
    n_batches <- floor(length(epoch_sample) / batch_size)
    for(j in 0:(n_batches - 1)) {
        i <- epoch_sample[(j * batch_size + 1):
                          (j * batch_size + batch_size)]
        par <- par - learning_rate * grad(par, i, ...)
    }
    par
}
```



R implementation

```r
batch <- function(par, epoch_sample, learning_rate, grad,
                  batch_size = 50, ...)
{
    n_batches <- floor(length(epoch_sample) / batch_size)
    for(j in 0:(n_batches - 1)) {
        i <- epoch_sample[(j * batch_size + 1):
                          (j * batch_size + batch_size)]
        par <- par - learning_rate * grad(par, i, ...)
    }
    par
}
```



Rcpp implementation

```
get_grad_H_cpp_wrap <- function(lambda, x_vec, y_vec, knots, innerknots)
{
    force_all_args()
    Phi_mat <- Phi(x_vec, knots)
    if (lambda > 0) {
        two_lambda_Omega <- 2 * lambda * Omega(inner_knots)
        function(beta, i) {                                          410
            two_lambda_Omega_beta <- two_lambda_Omega %*% beta      3240
            unpen_grad <- unpen_grad_cpp(beta, i, lambda, y, Phi_mat)  19330
            unpen_grad + two_lambda_Omega_beta                      2620
        }
    } else
        function(beta, i)
            unpen_grad_cpp(beta, i, lambda, y, Phi_mat)
}
```

R implementation

```
get_grad_H <- function(lambda, x_vec, y_vec, knots, inner_knots)
{
    force_all_args()
    Phi_mat <- Phi(x_vec, knots)
    unpen_grad <- function(beta, i) {                                220
        n_i <- length(i)                                            1360
        Phi_i <- matrix(Phi_mat[i, ], nrow = n_i)                  12870
        y_p_diff <- y_vec[i] - p(beta, Phi_i)                      13770
        cp <- crossprod(Phi_i, y_p_diff)                            5260
        -cp / n_i                                                   2010
    }
    if (lambda > 0) {
        two_lambda_Omega <- 2 * lambda * Omega(inner_knots)
        function(beta, i) {                                          150
            two_lambda_Omega_beta <- two_lambda_Omega %*% beta      3100
            unpen_grad(beta, i) + two_lambda_Omega_beta            39790
        }
    } else
        unpen_grad
}
```

Momentum: keep memory of last direction

With memory $\beta \in [0, 1)$, use update direction

$$\rho_n = \beta \rho_{n-1} + (1 - \beta) \frac{1}{M} \sum_{i \in I_n} \nabla_\theta L(x_i, \theta_n)$$

to get

$$\theta_n = \theta_{n-1} - \gamma_n \rho_n.$$

```r
momentum <- function() {
    rho <- 0
    function(par, epoch_sample, learning_rate, grad,
             batch_size = 50, mom_memory = 0.95, ...)
    {
        M <- floor(length(epoch_sample) / batch_size)
        for(j in 0:(M - 1)) {
            i <- epoch_sample[(j * batch_size + 1):
                      (j * batch_size + batch_size)]
            rho <<- mom_memory * rho + (1 - mom_memory) * grad(par, i, ...)
            par <- par - learning_rate * rho
        }
        par
    }
}
```

Adam: rescale gradient components to avoid narrow valley.

- Keeps memory of second moment for rescaling.
- *"invariant to diagonal rescaling of the gradients"* (Kingma and Ba, 2015).

```r
adam <- function() {
    rho <- v <- 0
    function(par, epoch_sample, learning_rate, grad,
             batch_size = 50, mom_memory = 0.9, mom2_memory = 0.9, ...)
    {
        M <- floor(length(epoch_sample) / batch_size)

        for(j in 0:(M - 1)) {
            i <- epoch_sample[(j * m + 1):(j * m + m)]
            gr <- grad(par, i, ...)
            rho <<- mom_memory * rho + (1 - mom_memory) * gr
            v <<- mom2_memory * v + (1 - mom2_memory) * gr^2
            par <- par - learning_rate * (rho / (sqrt(v) + 1e-8))
        }
        par
    }
}
```

Rcpp implementation

```
sgd <- function(par, grad, n_obs, decay_schedule, epoch = batch,
                n_iter = 100, sampler = sample, cb = NULL, ...)
{
    learning_rates <- decay_schedule(1:n_iter)
    for(k in 1:n_iter) {
        if(!is.null(cb))                                                    10
            cb()
        epoch_sample <- sampler(n_obs)                                     290
        par <- epoch(par, epoch_sample, learning_rates[k], grad, ...)    28840
    }
    if (!is.null(cb))
        cb()
    par
}
```

R implementation

```
sgd <- function(par, grad, n_obs, decay_schedule, epoch = batch,
                n_iter = 100, sampler = sample, cb = NULL, ...)
{
    learning_rates <- decay_schedule(1:n_iter)
    for(k in 1:n_iter) {
        if(!is.null(cb))                                                   380
            cb()
        epoch_sample <- sampler(n_obs)                                     340
        par <- epoch(par, epoch_sample, learning_rates[k], grad, ...)    47830
    }
    if (!is.null(cb))
        cb()
    par
}
```

Old slides

Slides using old version of gradient.

Getting H and $D_\beta H$ with $\lambda = 0$

- We set $\lambda = 0$, corresponding to usual unpenalized logistic regression, so we can compare results with `glm`.

```
grad_H <- get_grad_H(lambda = 0, x, y, knots, inner_knots)
H <- get_H(lambda = 0, x, y, knots, inner_knots)
```

- We confirm that our gradient implementation works by comparing to a numerical approximation.

```
all.equal(numDeriv::grad(H, beta),
          as.vector(grad_H(beta, 1:length(x))), tolerance = 1e-8)

## TRUE
```

Run methods

```r
set.seed(2021)

## Stochastic gradient descent
sgd_batch_tracer <- tracer(c("objective", "par"),
                           expr = quote(objective <- H(par)),
                           N = 0)
ds <- decay_scheduler(gamma0 = 0.8, gamma1 = 0.008, n1 = 90)
beta_fit_1 <- sgd(par = beta_init,
                  grad = grad_H,
                  n_obs = n_sim,
                  decay_schedule = ds,
                  epoch = batch,
                  n_iter = 200,
                  batch_size = 1,
                  cb = sgd_batch_tracer$tracer)

## Newton
newton_tracer <- tracer(c("objective", "par_new"),
                        expr = quote(objective <- H(par_new)))
newton_fit <- newton(init_guess = beta_init,
                     x = x,
                     y = y,
                     max_iter = 100,
                     cb = newton_tracer$tracer)
```

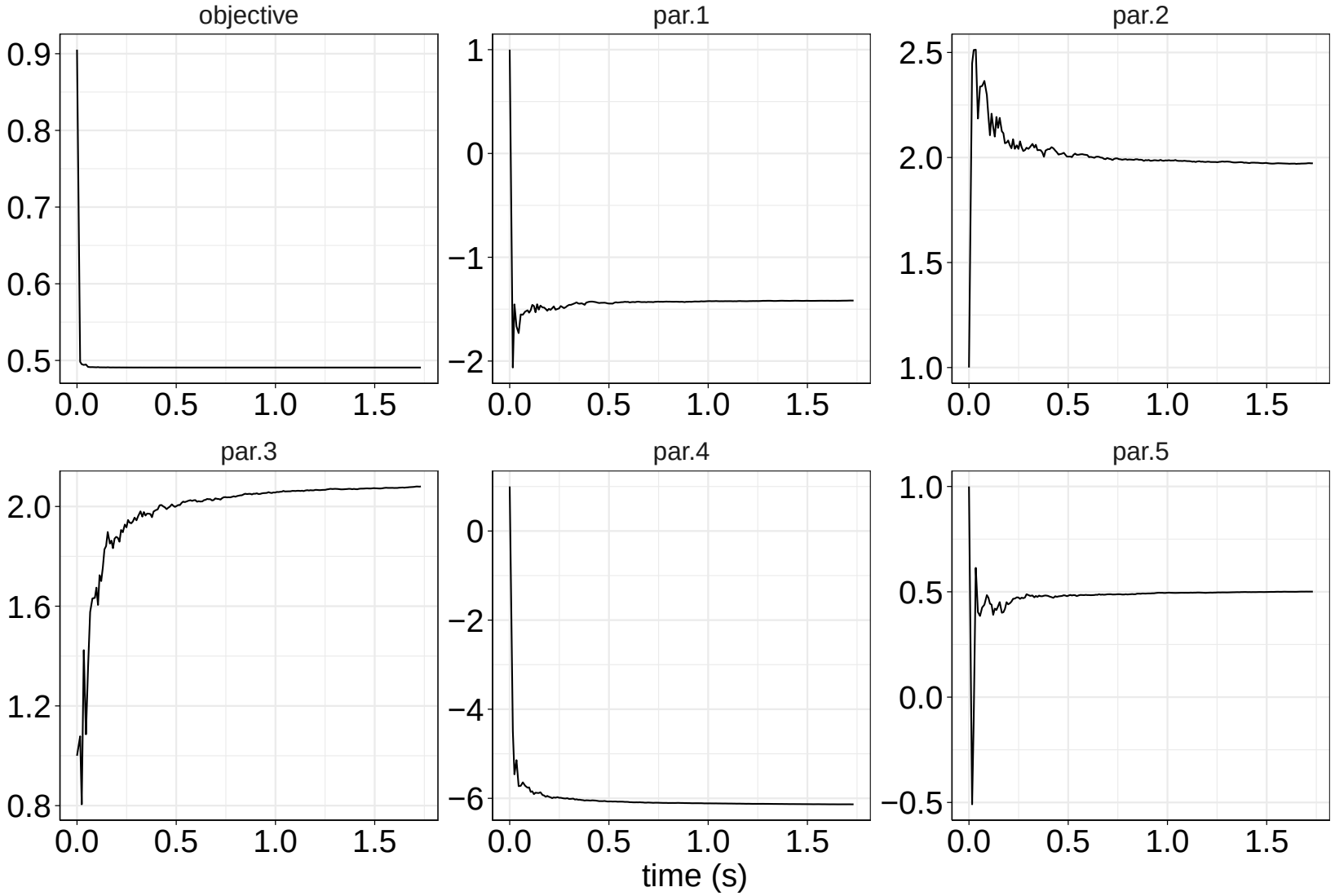
Comparing results to `glm` (IWLS)

```r
glm(y ~ splineDesign(knots, x) - 1, family = binomial)$coefficients
```

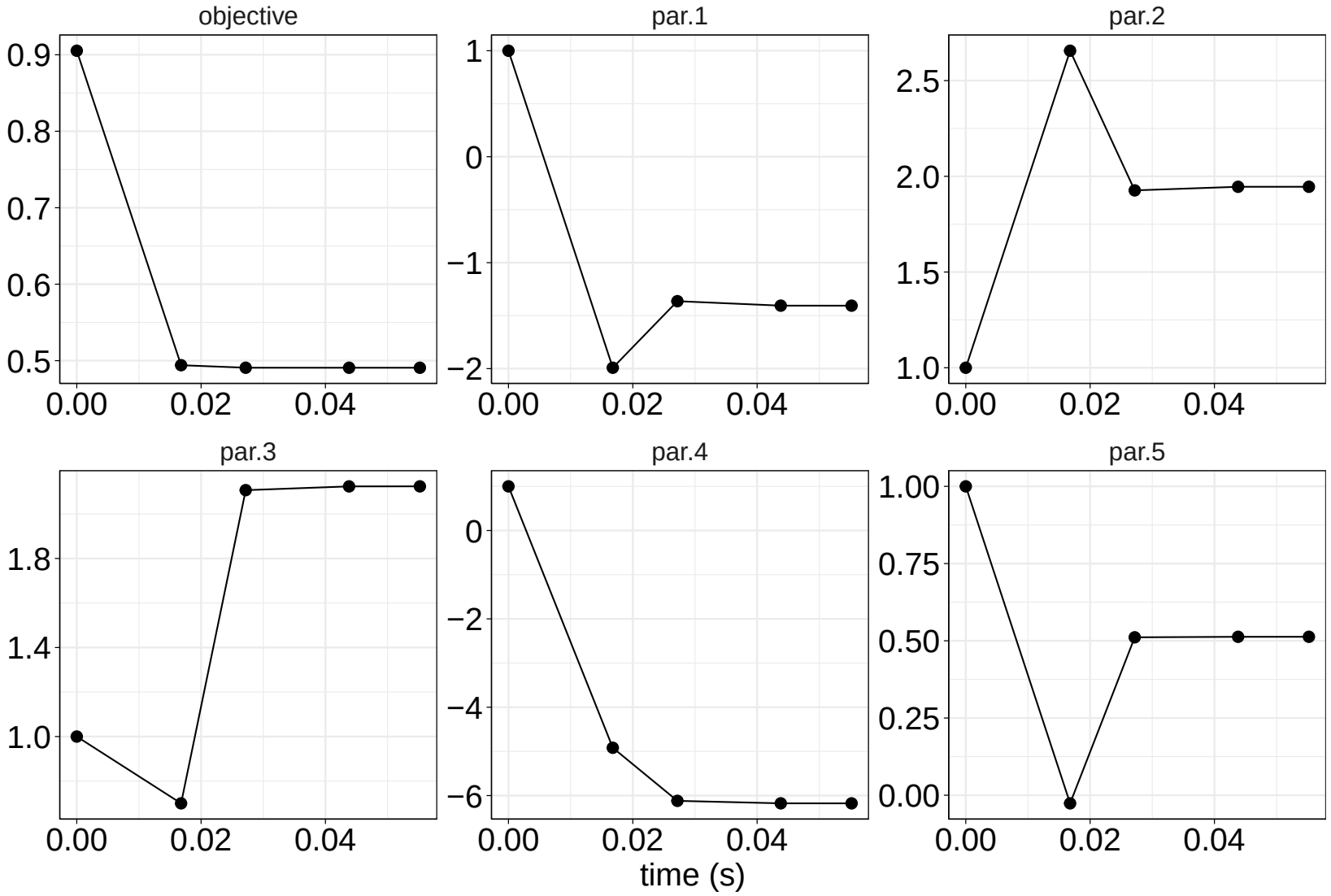
sgd	newton	glm
-1.4172427	-1.4058146	-1.4058146
1.9725305	1.9454677	1.9454677
2.0802703	2.1241283	2.1241283
-6.1350949	-6.1761810	-6.1761811
0.5009762	0.5129334	0.5129334

The methods yield approximately the same results, indicating that our implementations work with $\lambda = 0$.

SGD convergence (200 iterations)

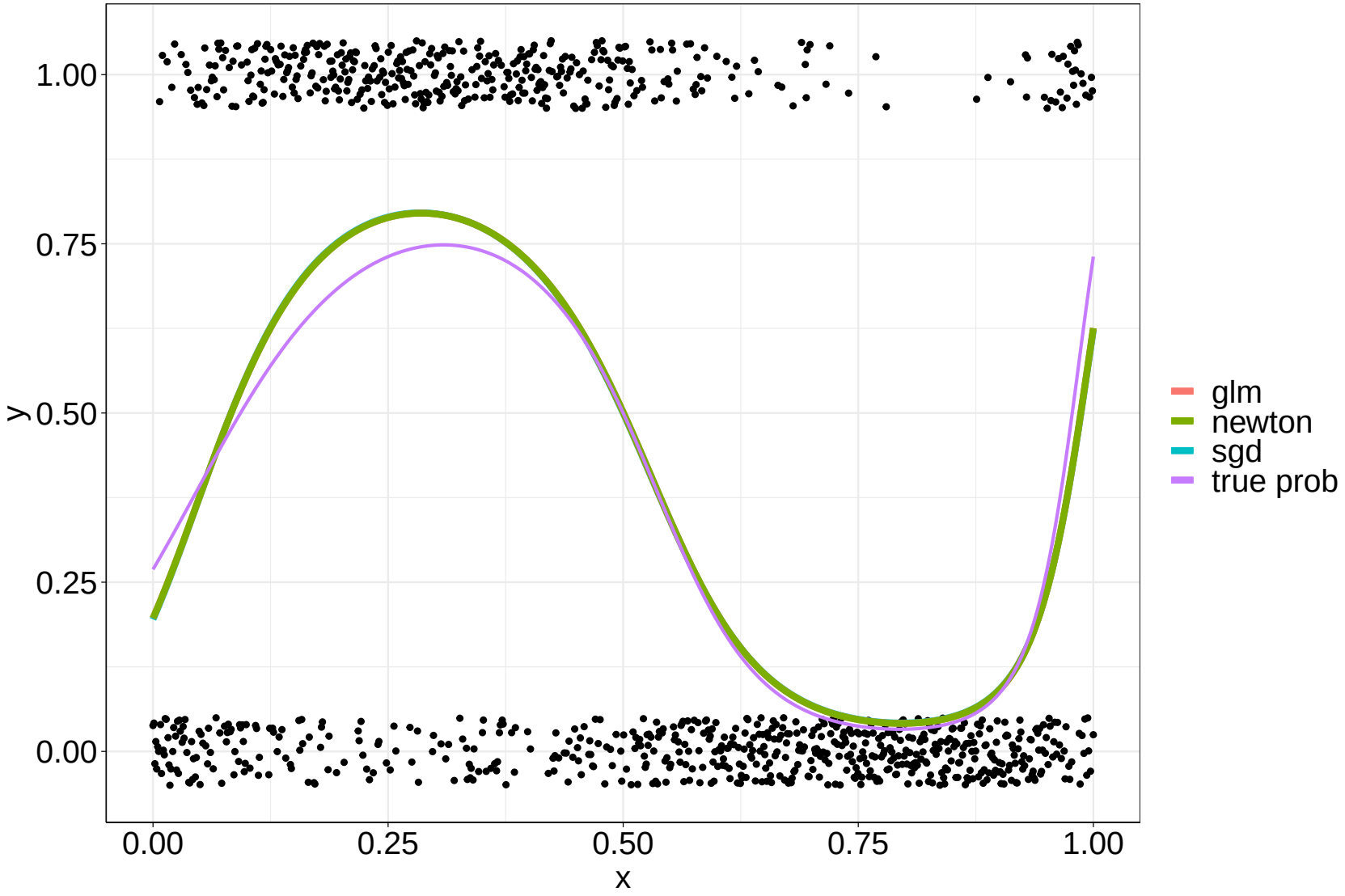


Newton convergence (4 iterations)



- Okay convergence two steps from initial guess!

Predictions (the same for all three methods)



Profiling `sgd` and `batch`

```
sgd <- function(par, grad, n_obs, decay_schedule, epoch = batch,
                n_iter = 100, sampler = sample, cb = NULL, ...)
{
    learning_rates <- decay_schedule(1:n_iter)
    for(k in 1:n_iter) {
        if(!is.null(cb))                                                  330
            cb()
        epoch_sample <- sampler(n_obs)                                    330
        par <- epoch(par, epoch_sample, learning_rates[k], grad, ...)   43240
    }
    if (!is.null(cb))
        cb()
    par
}
```

```
batch <- function(par, epoch_sample, learning_rate, grad,
                  batch_size = 50, ...)
{
    n_batches <- floor(length(epoch_sample) / batch_size)                  10
    for(j in 0:(n_batches - 1)) {                                         100
        i <- epoch_sample[(j * batch_size + 1):                          1420
                          (j * batch_size + batch_size)]
        par <- par - learning_rate * grad(par, i, ...)                  41680
    }
    par
}
```

- `grad` is the bottleneck.

Profiling `grad_H` and `p`

```
p <- function(beta, phi_mat)                                          220
{
    phi_beta_prod <- phi_mat %*% beta                                 2720
    inv_logit(phi_beta_prod)                                          6330
}

get_grad_H <- function(lambda, x_vec, y_vec, knots, inner_knots)
{
    force_all_args()
    Phi_mat <- Phi(x_vec, knots)
    Omega_mat <- Omega(inner_knots)
    function(beta, i) {                                               160
        Phi_i <- matrix(Phi_mat[i,], nrow = length(i))                13090
        Omega_beta <- Omega_mat %*% beta                              3220
        cp <- crossprod(Phi_i, y_vec[i] - p(beta, Phi_i))             16630
        -cp / length(x_vec[i]) + 2 * lambda * Omega_beta              6180
    }
}
```

- The indexing `Phi_mat[i, ]` is slow in R. Would be faster in C++.
- The rest are all vectorized operations in R, so they cannot be improved much in C++.

$\lambda > 0$: Run methods

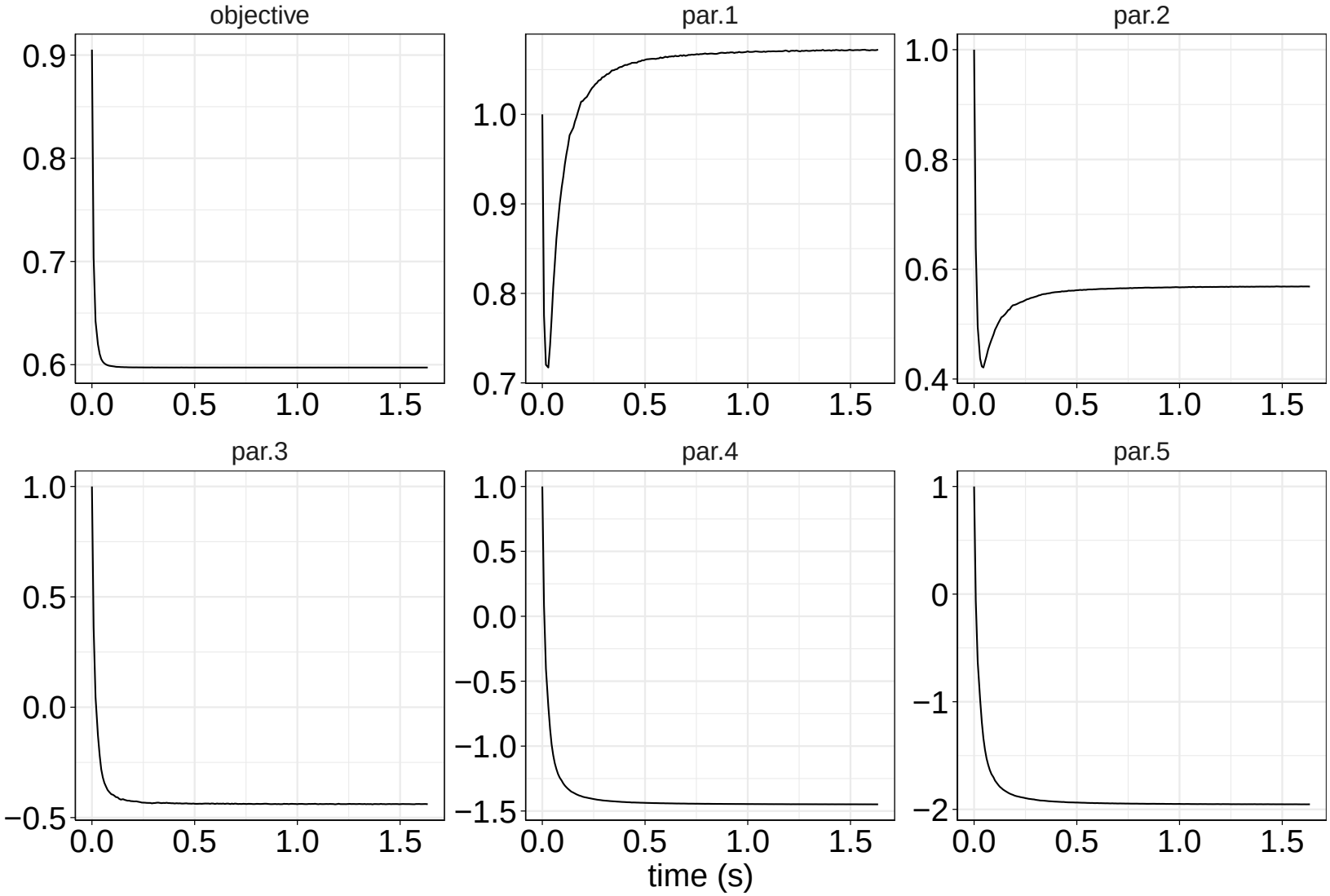
```r
## Stochastic gradient descent
grad_H <- get_grad_H(lambda = 0.2, x, y, knots, inner_knots)
H <- get_H(lambda = 0.2, x, y, knots, inner_knots)
sgd_batch_tracer <- tracer(c("objective", "par"),
                           expr = quote(objective <- H(par)),
                           N = 0)
beta_fit_lam <- sgd(par = beta_init,
                    grad = grad_H,
                    n_obs = n_sim,
                    decay_schedule = decay_scheduler(gamma0 = 0.015,
                                                     gamma1 = 0.001,
                                                     n1 = 90),
                    epoch = batch,
                    n_iter = 200,
                    batch_size = 1,
                    cb = sgd_batch_tracer$tracer)

## Newton
newton_tracer <- tracer(c("objective", "par_new"),
                        expr = quote(objective <- H(par_new)))
newton_fit_lam <- newton(init_guess = beta_init, x = x, y = y,
                         max_iter = 100,
                         lambda = 0.2,
                         cb = newton_tracer$tracer)
```

$\lambda > 0$: Comparing β estimates

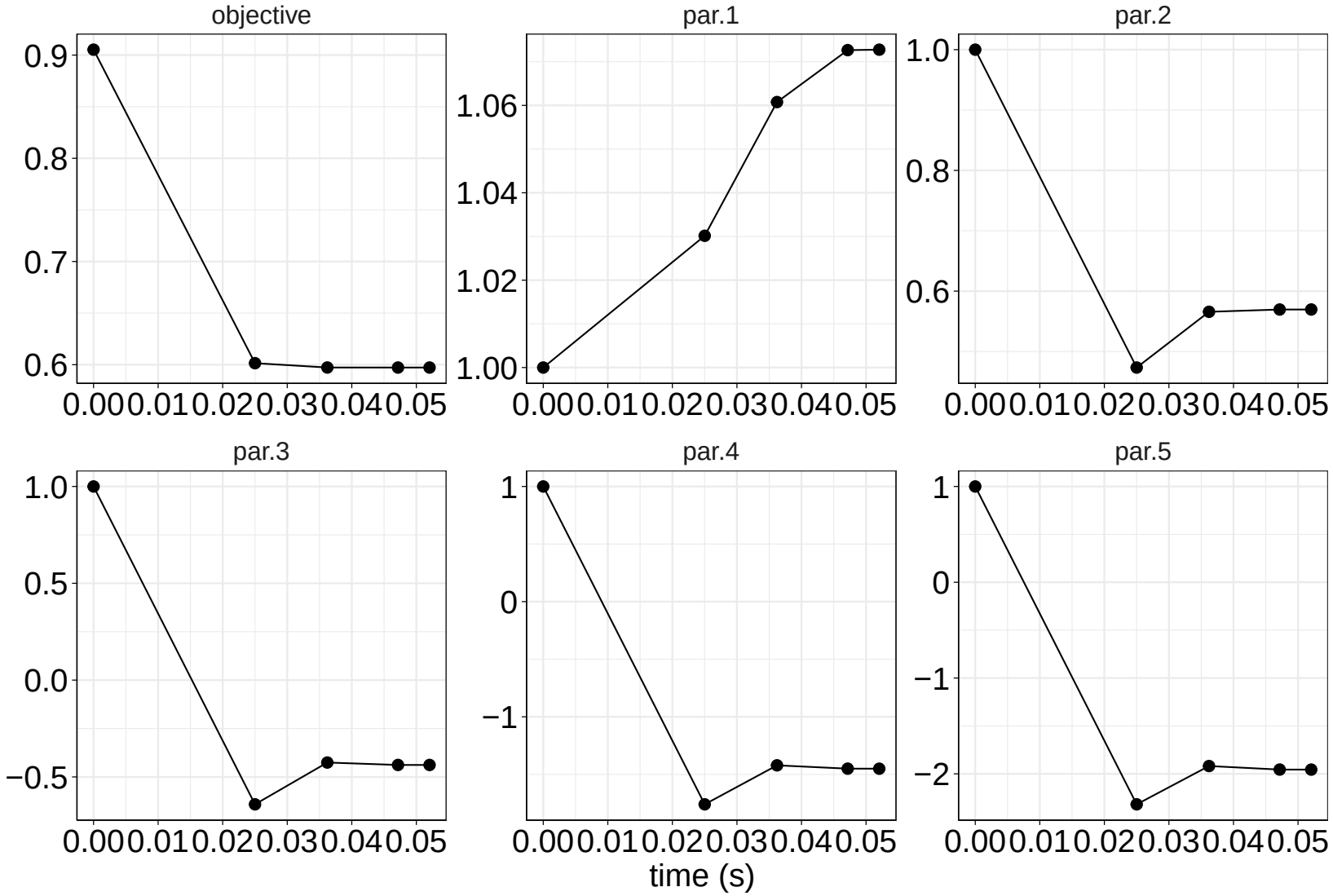
sgd	newton
1.0717191	1.0727550
0.5687178	0.5695332
-0.4380832	-0.4381971
-1.4485311	-1.4505377
-1.9531791	-1.9558413

- Approximately the same results!
- Indicates that there are no bugs in the implementations.

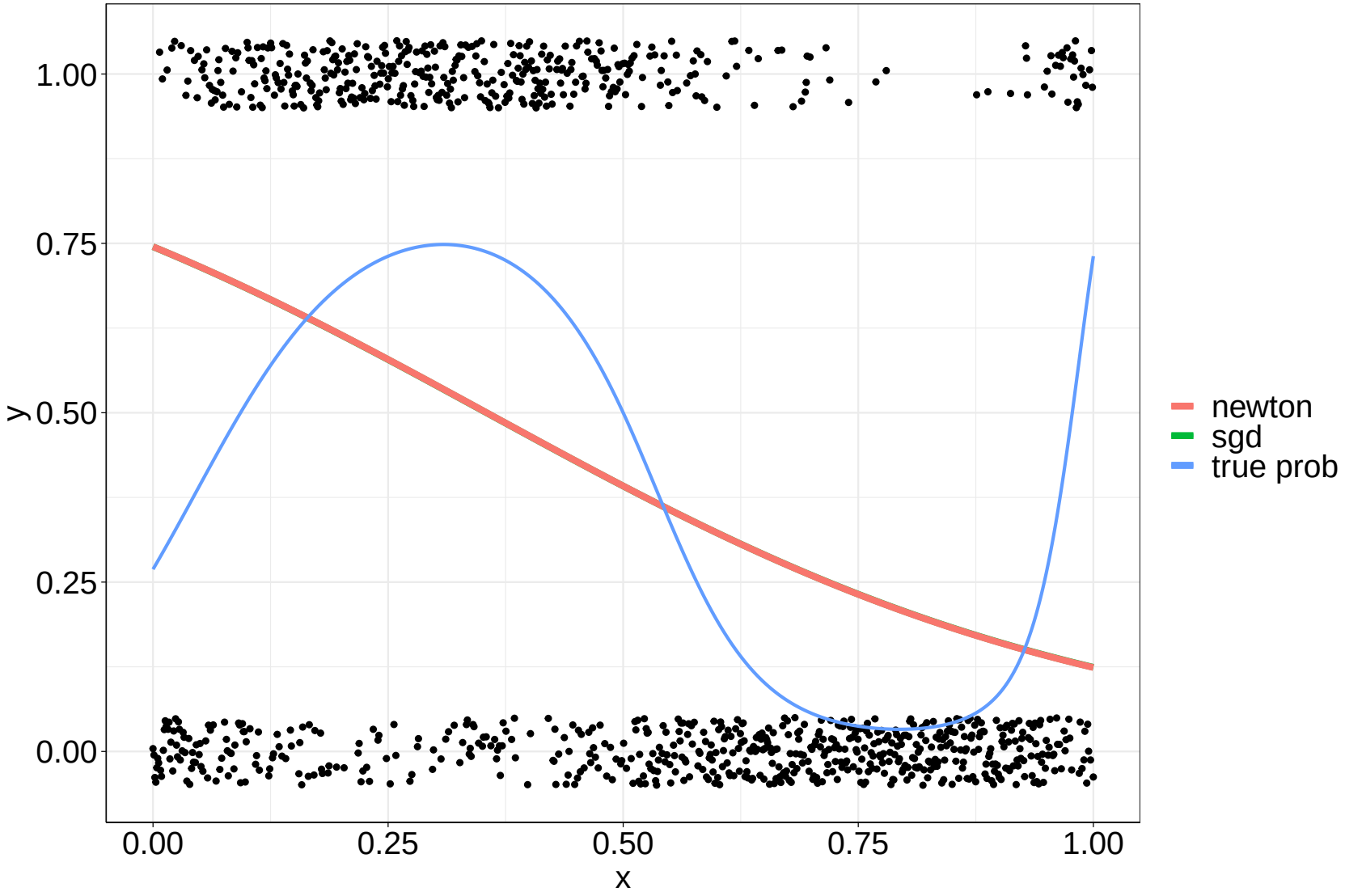
$\lambda > 0$: SGD convergence (200 iter.)



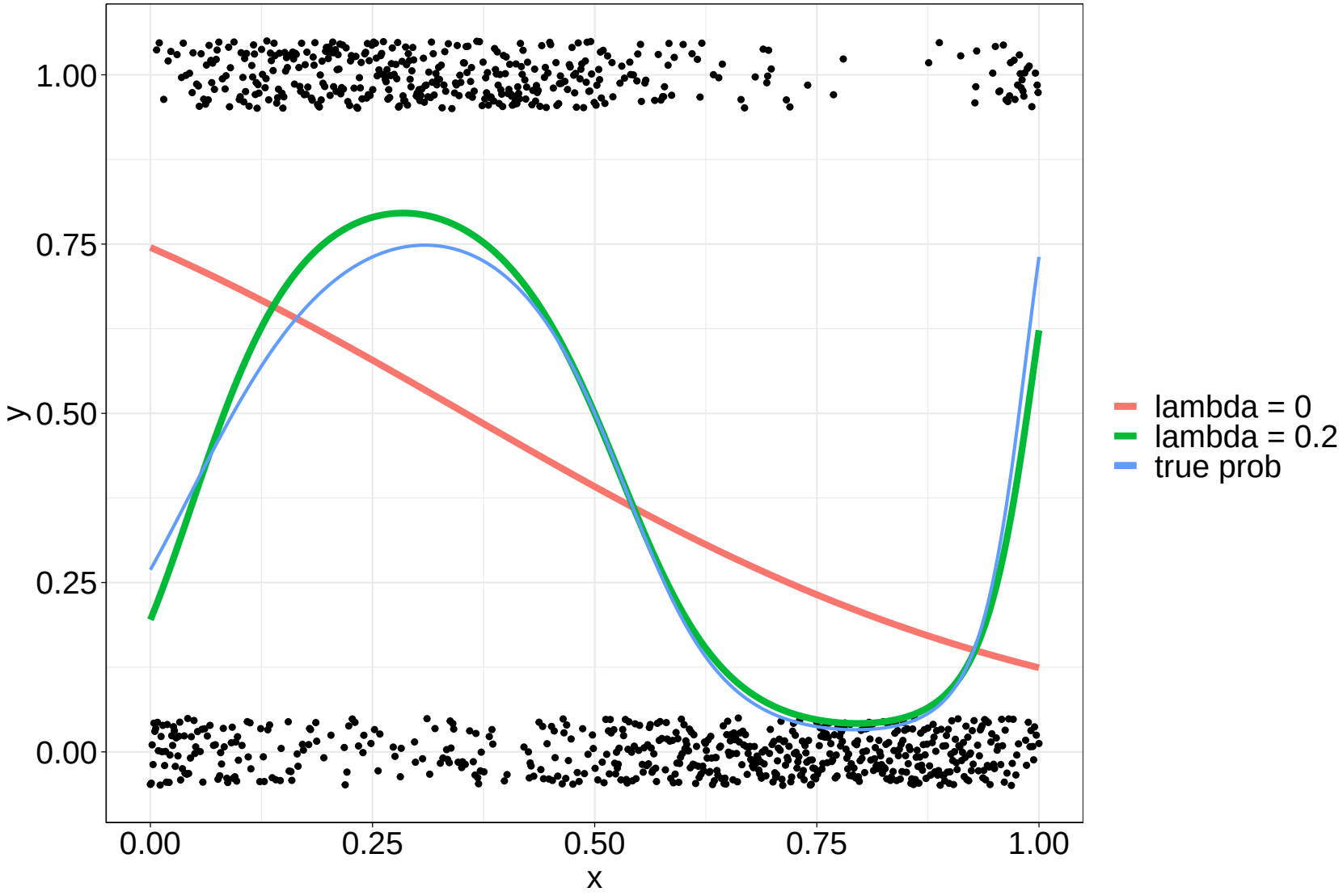
$\lambda > 0$: Newton convergence (4 iter.)



$\lambda > 0$: Fitted probabilities



Comparing prob. fits with $\lambda = 0$ and $\lambda = 0.2$

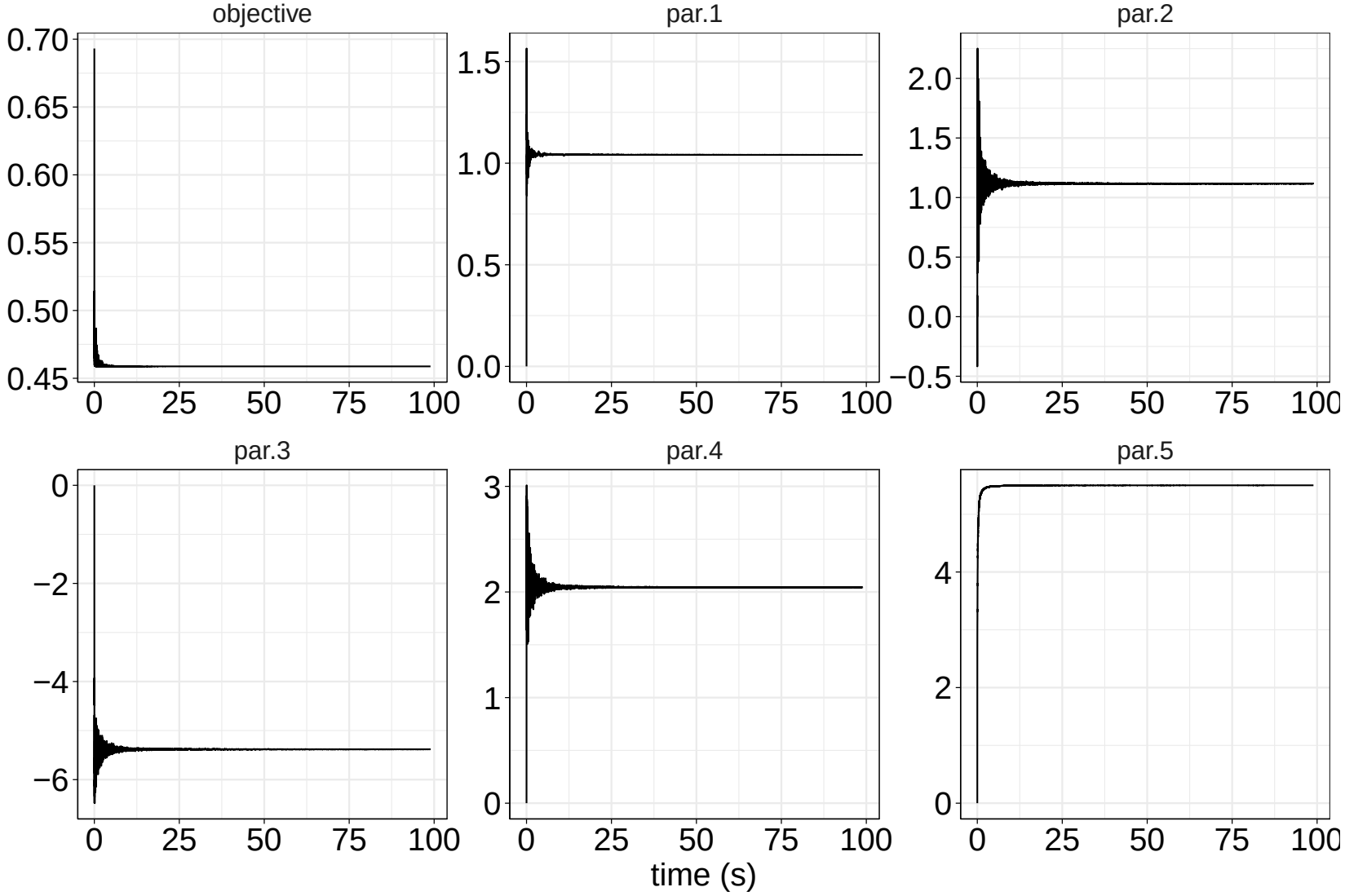


- As expected, $\lambda > 0$ gives a less volatile fit.
- But, in this case, $\lambda = 0.2$ oversmoothes.

Horse data: Fitting

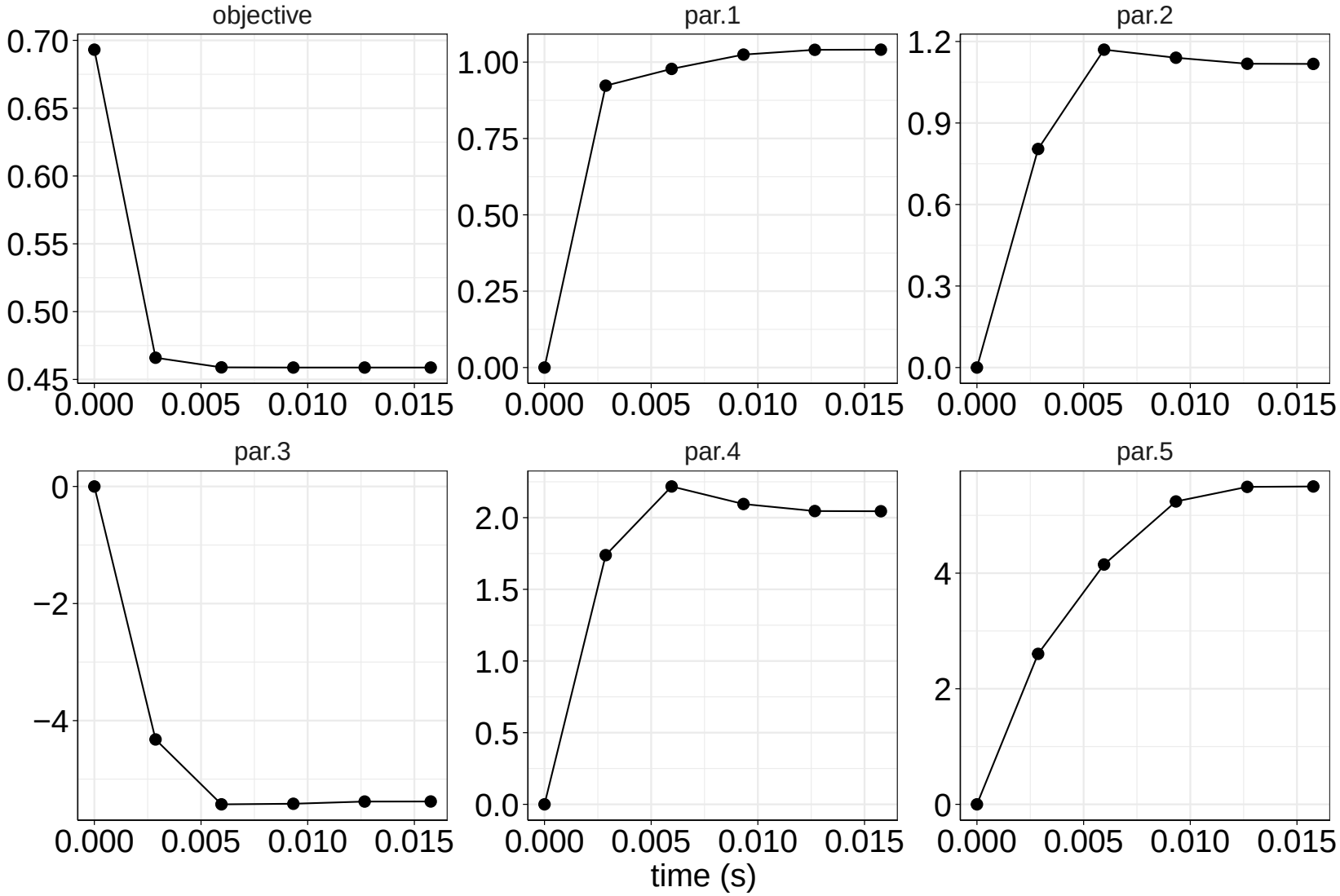
```r
## Load data and scale and center x for faster convergence
x <- scale(horses$Temperature)
y <- horses$dead
n_beta <- 5
beta_init <- rep(0, n_beta)
inner_knots <- seq(min(x), max(x), length.out = n_beta - 2)
knots <- get_knots(inner_knots)

## SGD
grad_H <- get_grad_H(lambda = 0, x, y, knots, inner_knots)
H <- get_H(lambda = 0, x, y, knots, inner_knots)
sgd_batch_tracer <- tracer(c("objective", "par"),
                           expr = quote(objective <- H(par)),
                           N = 0)
beta_fit_horse <- sgd(par = beta_init,
                      grad = grad_H,
                      n_obs = length(x),
                      decay_schedule = decay_scheduler(gamma0 = 1,
                                                       gamma1 = 0.3,
                                                       n1 = 125),
                      epoch = batch,
                      n_iter = 20000,
                      batch_size = 1,
                      cb = sgd_batch_tracer$tracer)

## Newton
newton_tracer <- tracer(c("objective", "par_new"),
                        expr = quote(objective <- H(par_new)))

newton_fit_horse <- newton(init_guess = beta_init,
                           x = x,
                           y = y,
                           max_iter = 100,
                           cb = newton_tracer$tracer)
```

Horse data: SGD convergence (20000 iter.)



- Hard to find a better decay schedule that takes large enough steps, but without fluctuating up and down.

Horse data: Newton convergence (5 iter.)

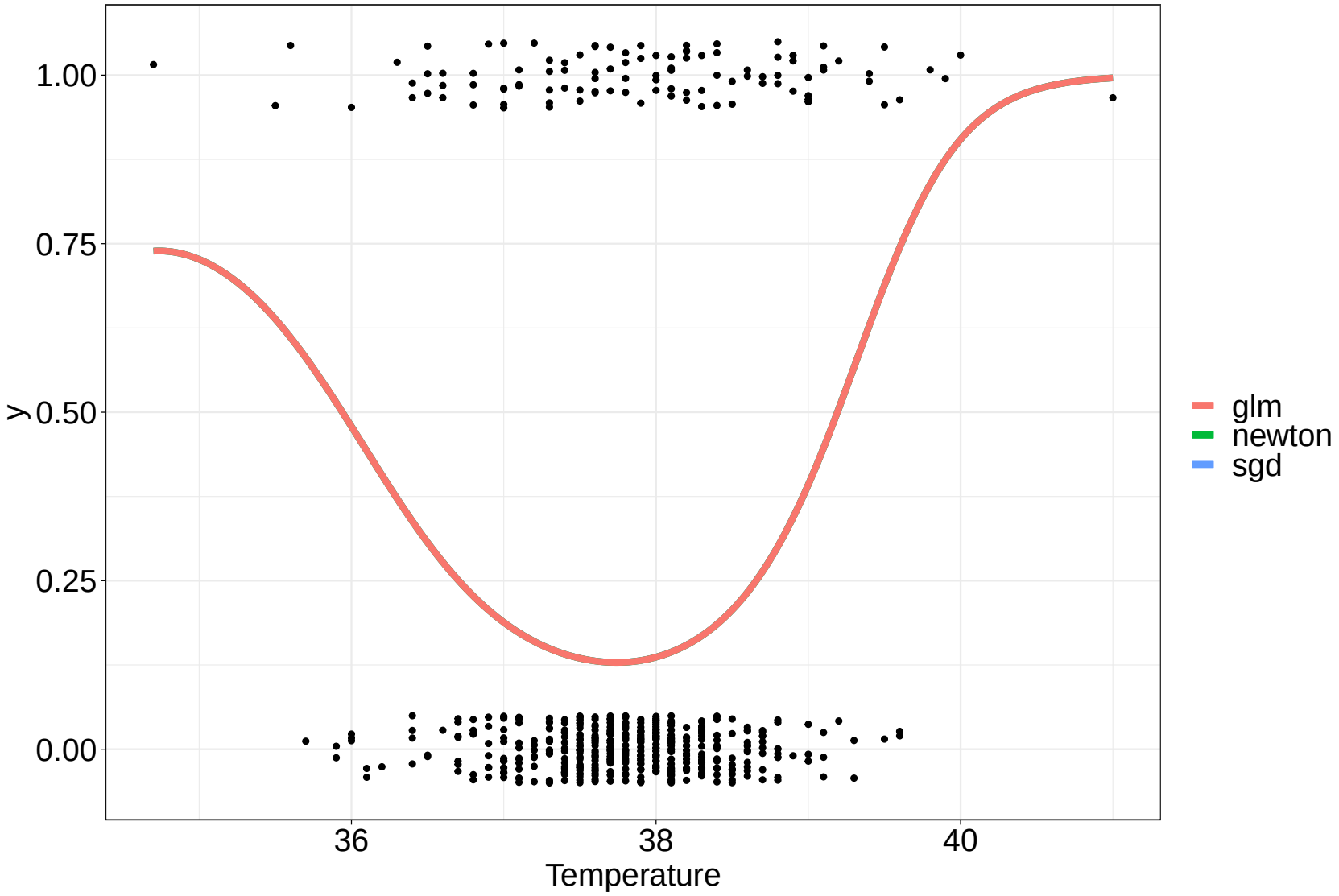


Horse data: Estimates

sgd	newton	glm
1.040949	1.040815	1.040816
1.117849	1.117413	1.117412
-5.380647	-5.381459	-5.381458
2.045161	2.044704	2.044703
5.499648	5.499755	5.499760

- Approximately the same.

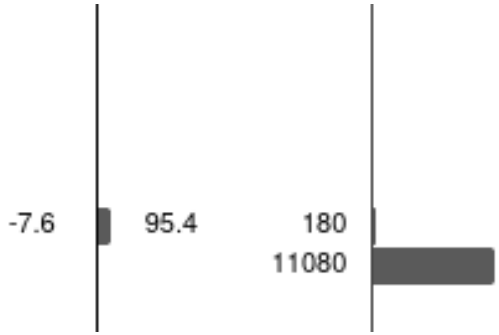
Horse data: Fitted probabilities



- Approximately the same.

C++ profiling

```
sgd_cpp_wrap <- function(par, x, y, decay_schedule, inner_knots,
                         lambda = 0, n_iter = 100, batch_size = 1)
{
    learning_rates <- decay_schedule(1:n_iter)
    Om <- Omega(inner_knots)
    Phi_mat <- Phi(x, get_knots(inner_knots))
    sgd_cpp(par, learning_rates, n_iter, batch_size, lambda, y, Phi_mat, Om)
}
```



- `profvis` cannot check memory allocation and deallocation done in C++.
- But we know that we only allocate memory for all large objects once, so we must do better.