Optimization - optax

Lecture 16

Dr. Colin Rundel

SGD Libraries

Most often you will be using the optimizer methods that come with your library of choice, all of the following have their own implementations:

- Tensorflow / Keras
- Torch

Interestingly, JAX does not have builtin support for optimization beyond `jax.scipy.optimize.minimize()` which only supports the `BFGS` method.

Google previously released jaxopt to provide SGD and other optimization methods but this project is now deprecated with the code being merged into DeepMind's Optax.

Optax

> Optax is a gradient processing and optimization library for JAX.
>
> Optax is designed to facilitate research by providing building blocks that can be easily recombined in custom ways.
>
> Our goals are to
>
> - Provide simple, well-tested, efficient implementations of core components.
> - Improve research productivity by enabling to easily combine low-level ingredients into custom optimizers (or other gradient processing components).
> - Accelerate adoption of new ideas by making it easy for anyone to contribute.
>
> We favor focusing on small composable building blocks that can be effectively combined into custom solutions. Others may build upon these basic components in more complicated abstractions. Whenever reasonable, implementations prioritize readability and structuring code to match standard equations, over code reuse.x

Same regression 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
)

X = jnp.c_[jnp.ones(len(y)), X]
n, k = X.shape

def lr_loss(beta, X, y):
  return jnp.sum((y - X @ beta)**2)
```

Optax process

- Construct a `GradientTransformation` object, set optimizer settings

```
optimizer = optax.sgd(learning_rate=0.0001); optimizer
```

```
GradientTransformationExtraArgs(init=<function chain.<locals>.init_fn at 0x169936ac0>, update=<function chain.<locals>.update_fn at
0x169936ca0>)
```

- Initialize the optimizer with the initial parameter values

```
beta = jnp.zeros(k)
opt_state = optimizer.init(beta); opt_state
```

```
(EmptyState(), EmptyState())
```

- Perform iterations
  - Calculate the current gradient and update for the optimizer

```
f, grad = jax.value_and_grad(lr_loss)(beta, X, y)
updates, opt_state = optimizer.update(grad, opt_state); updates, opt_state
```

```
(Array([  7.1983,   1.8515,   1.1396,   1.7858,  -2.8407,  -0.1266,
         0.1514,  -0.4875,  -0.2072,  25.7022,  90.4929,   7.5036,
         0.2313, 123.5414,   1.3136,   2.567 ,  -0.4262,  -1.2996,
         0.5124,  -0.2265,   2.3771], dtype=float64), (EmptyState(), EmptyState()))
```

  - Apply the update to the parameter

```
beta = optax.apply_updates(beta, updates); beta
```

```
Array([  7.1983,   1.8515,   1.1396,   1.7858,  -2.8407,  -0.1266,
         0.1514,  -0.4875,  -0.2072,  25.7022,  90.4929,   7.5036,
         0.2313, 123.5414,   1.3136,   2.567 ,  -0.4262,  -1.2996,
         0.5124,  -0.2265,   2.3771], dtype=float64)
```

Basic Example - GD

Implementation | Results

```
optimizer = optax.sgd(learning_rate=0.00001)

beta = jnp.zeros(k)
opt_state = optimizer.init(beta)

gd_loss = []
for iter in range(50):
  f, grad = jax.value_and_grad(lr_loss)(beta, X, y)
  updates, opt_state = optimizer.update(grad, opt_state)
  beta = optax.apply_updates(beta, updates)
  gd_loss.append(f)

beta
```

```
Array([ 3.0082,  0.009 ,  0.0003,  0.0023,  0.0035,  0.0033,  0.0261,
       -0.0006,  0.0005, 12.2771, 44.4933,  3.6423,  0.0168, 61.3929,
       -0.0011, -0.0054,  0.0139, -0.0094, -0.0054,  0.0023,  0.0219],      dtype=float64)
```

Basic Optax Example - Adam

Implementation | Results

```
optimizer = optax.adam(learning_rate=1, b1=0.9, b2=0.999, eps=1e-8)

beta = jnp.zeros(k)
opt_state = optimizer.init(beta)

adam_loss = []
for iter in range(50):
  f, grad = jax.value_and_grad(lr_loss)(beta, X, y)
  updates, opt_state = optimizer.update(grad, opt_state)
  beta = optax.apply_updates(beta, updates)
  adam_loss.append(f)

beta
```

```
Array([ 3.3313,  0.229 ,  0.0878,  0.1995, -0.2172, -0.0805,  0.0505,
       -0.033 , -0.2049, 11.6465, 39.4043,  3.8843,  0.1239, 43.1531,
        0.2196,  0.3237, -0.0915, -0.2611,  0.0874,  0.1323,  0.3156],      dtype=float64)
```

A bit more on learning rate and batch size

Optax and mini batches

```python
def optax_optimize(params, X, y, loss_fn, optimizer, steps=50, batch_size=1, seed=1234):
  n, k = X.shape
  res = {"loss": [], "epoch": np.linspace(0, steps, int(steps*(n/batch_size) + 1))}

  opt_state = optimizer.init(params)
  grad_fn = jax.grad(loss_fn)

  rng = np.random.default_rng(seed)
  batches = np.array(range(n))
  rng.shuffle(batches)

  for iter in range(steps):
    for batch in batches.reshape(-1, batch_size):
      res["loss"].append(loss_fn(params, X, y).item())
      grad = grad_fn(params, X[batch,:], y[batch])
      updates, opt_state = optimizer.update(grad, opt_state)
      params = optax.apply_updates(params, updates)

  res["params"] = params
  res["loss"].append(loss_fn(params, X, y).item())

  return(res)
```

Fitting - SGD - Fixed LR (small)

Implementation | Results

```
batch_sizes = [10, 100, 1000, 10000]
lrs = [0.00001] * 4

sgd = {
  batch_size: optax_optimize(
    params=jnp.zeros(k), X=X, y=y, loss_fn=lr_loss,
    optimizer=optax.sgd(learning_rate=lr),
    steps=30, batch_size=batch_size, seed=1234
  )
  for batch_size, lr in zip(batch_sizes, lrs)
}
```

Fitting - SGD - Adjusted LR

Implementation | Full | Zoom

```
batch_sizes = [10, 100, 1000, 10000]
lrs = [0.005, 0.001, 0.0001, 0.00001]

sgd = {
  batch_size: optax_optimize(
    params=jnp.zeros(k), X=X, y=y, loss_fn=lr_loss,
    optimizer=optax.sgd(learning_rate=lr),
    steps=30, batch_size=batch_size, seed=1234
  )
  for batch_size, lr in zip(batch_sizes, lrs)
}
```

Fitting - SGD - Fixed LR, Small batch size

Implementation | Full | Zoom

```
batch_sizes = [10, 25, 50, 100]
lrs = [0.001] * 4

sgd = {
  batch_size: optax_optimize(
    params=jnp.zeros(k), X=X, y=y, loss_fn=lr_loss,
    optimizer=optax.sgd(learning_rate=lr),
    steps=2, batch_size=batch_size, seed=1234
  )
  for batch_size, lr in zip(batch_sizes, lrs)
}
```

Runtime per epoch

Implementation | Runtimes | Scaled

```
batch_sizes = [10, 50, 100, 10000]
lrs = [0.001] * 4

sgd_runtime = {
  batch_size: timeit.Timer( lambda:
    optax_optimize(
      params=jnp.zeros(k), X=X, y=y, loss_fn=lr_loss,
      optimizer=optax.sgd(learning_rate=lr),
      steps=1, batch_size=batch_size, seed=1234
    )
  ).repeat(5,1)
  for batch_size, lr in zip(batch_sizes, lrs)
}
```

Some lessons / comments

- Batch size determines both training time and computing resources
- Generally there will be an inverse relationship between learning rate and batch size
- Most optimizer hyperparameters are sensitive to batch size
- For really large models batches are a necessity and sizing is often determined by resource / memory constraints

Adam

Adam - Fixed LR

Implementation | Results

```
batch_sizes = [10, 25, 50, 100]
lrs = [1]*4

adam = {
  batch_size: optax_optimize(
    params=jnp.zeros(k), X=X, y=y, loss_fn=lr_loss,
    optimizer=optax.adam(learning_rate=lr, b1=0.9, b2=0.999, eps=1e-8),
    steps=2, batch_size=batch_size, seed=1234
  )
  for batch_size, lr in zip(batch_sizes, lrs)
}
```

Adam - Smaller Fixed LR

Implementation | Results

```
batch_sizes = [10, 25, 50, 100]
lrs = [0.1]*4

adam = {
  batch_size: optax_optimize(
    params=jnp.zeros(k), X=X, y=y, loss_fn=lr_loss,
    optimizer=optax.adam(learning_rate=lr, b1=0.9, b2=0.999, eps=1e-8),
    steps=10, batch_size=batch_size, seed=1234
  )
  for batch_size, lr in zip(batch_sizes, lrs)
}
```

Learning rate schedules

As mentioned last time, most gradient based methods are not guaranteed to converge unless their learning rates decay as a function of step number.

Optax supports a large number of pre-built learning rate schedules which can be passed into any of its optimizers instead of a fixed floating point value.

```
schedule = optax.linear_schedule(
    init_value=1., end_value=0., transition_steps=5
)

[schedule(step).item() for step in range(6)]
```

```
[1.0, 0.8, 0.6, 0.4, 0.19999999999999996, 0.0]
```

Adam w/ Exp Decay

Implementation | Results

```
batch_sizes = [10, 25, 50, 100]

adam = {
  batch_size: optax_optimize(
    params=jnp.zeros(k), X=X, y=y, loss_fn=lr_loss,
    optimizer=optax.adam(
      learning_rate=optax.schedules.exponential_decay(
        init_value=1,
        transition_steps=100,
        decay_rate=0.9
      ),
      b1=0.9, b2=0.999, eps=1e-8
    ),
    steps=2, batch_size=batch_size, seed=1234
  )
  for batch_size in batch_sizes
}
```

Runtime per epoch

Implementation | Runtimes | Scaled

```
batch_sizes = [10, 25, 50, 100]

adam_runtime = {
  batch_size: timeit.Timer( lambda:
    optax_optimize(
      params=jnp.zeros(k), X=X, y=y, loss_fn=lr_loss,
      optimizer=optax.adam(
        learning_rate=optax.schedules.exponential_decay(
          init_value=1,
          transition_steps=100,
          decay_rate=0.9
        ),
        b1=0.9, b2=0.999, eps=1e-8
      ),
      steps=1, batch_size=batch_size, seed=1234
    )
  ).repeat(5,1)
  for batch_size in batch_sizes
}
```

Some advice …

The following is from Google Research's Tuning Playbook:

- No optimizer is the "best" across all types of machine learning problems and model architectures. Even just comparing the performance of optimizers is a difficult task. 🤖
- We recommend sticking with well-established, popular optimizers, especially when starting a new project.
  - Ideally, choose the most popular optimizer used for the same type of problem.
- Be prepared to give attention to *all* hyperparameters of the chosen optimizer.
  - Optimizers with more hyperparameters may require more tuning effort to find the best configuration.
  - This is particularly relevant in the beginning stages of a project when we are trying to find the best values of various other hyperparameters (e.g. architecture hyperparameters) while treating optimizer hyperparameters as nuisance parameters.
  - It may be preferable to start with a simpler optimizer (e.g. SGD with fixed momentum or Adam with fixed ϵ, β_1, and β_2) in the initial stages of the project and switch to a more general optimizer later.
- Well-established optimizers that we like include (but are not limited to):
  - SGD with momentum (we like the Nesterov variant)
  - Adam and NAdam, which are more general than SGD with momentum. Note that Adam has 4 tunable hyperparameters and they can all matter!

Optimization in R

Basic optimization

The equivalent of `scipy`'s `optimize.minimize()` for unconstrained continuous optimization problems in R is `stats::optim()` - there is nearly a 1-to-1 correspondence between the two functions and the available optimizers.

```
optim(par, fn, gr = NULL, …,
      method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",
                 "Brent"),
      lower = -Inf, upper = Inf,
      control = list(), hessian = FALSE)
```

The only missing method from `scipy` is `Newton-CG` and there is the addition of the `SANN` method which is a variant of simulated annealing and does not require gradient information. However, it is slow and very sensitive to its control parameters and is not considered a general-purpose method.

All other tuning knobs are hidden in `control` - see the documentation for details. Most important options include: `maxit`, `abstol`, and `reltol`.

Return values

`optim()` returns a list of results, most of which are expected: `par` the minimizer, `value` objective function at `par`, `counts` the number of function and gradient evaluations.

"Success" of the optimization is reported by `convergence` which is a little bit weird (think unix exit codes):

- `0` - indicates successful convergence based on the criteria specified by `control`
- `1` - indicates failure due to reaching the `maxit` limit
- Any other number indicates a special case depending on the method, check `message`

Usage

```r
## Rosenbrock Banana function
f = function(x) {
  100 * (x[2] - x[1] * x[1]) ^ 2 + (1 - x[1]) ^ 2
}
grad = function(x) {
  c(-400 * x[1] * (x[2] - x[1] * x[1]) - 2 * (1 - x[1]),
    200 * (x[2] - x[1] * x[1]))
}
x0 = c(-1.2, 1)
```

```r
optim(x0, f, grad, method = "BFGS")
```

```
$par
[1] 1 1

$value
[1] 9.594956e-18

$counts
function gradient
     110       43

$convergence
[1] 0

$message
NULL
```

```r
optim(x0, f, grad, method = "CG")
```

```
$par
[1] -0.7648373  0.5927588

$value
[1] 3.106579

$counts
function gradient
     402      101

$convergence
[1] 1

$message
NULL
```

SGD related methods

For any of these algorithms you will generally be depending on the underlying modeling library to make them available to you, for example:

- Keras optimizers implemented
- Torch optimizers

Details are library dependent.

optimx

> optimx is an R package that extends and enhances the optim() function of base R, in particular by unifying the call to many solvers.

Makes a variety of solvers from different packages available with a unified calling framework.

Packages include: `pracma`, `minqa`, `dfoptim`, `lbfgs`, `lbfgsb3c`, `marqLevAlg`, `nloptr`, `dfoptim`, `BB`, `subplex`, and `ucminf`

nloptr

Wrapper around the NLopt library (which also has a Python interface).

- Provides a large number of global and local solvers (including everything available in optim)
- Provides more robust support for constrained optimization problems

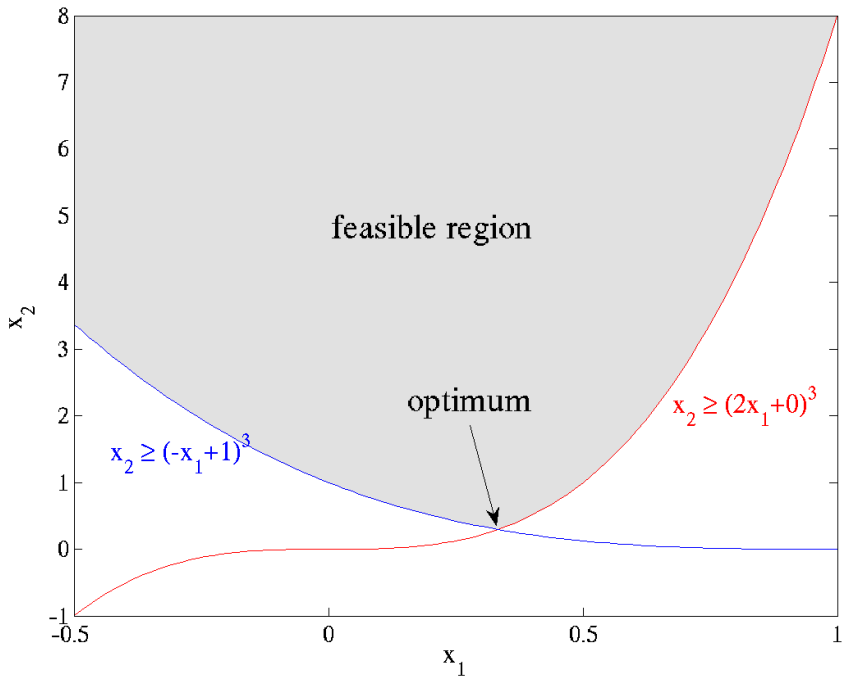
Usage

```
## Rosenbrock Banana function
f = function(x) {
  100 * (x[2] - x[1] * x[1]) ^ 2 + (1 - x[1
}

grad = function(x) {
  c(-400 * x[1] * (x[2] - x[1] * x[1]) - 2 *
    200 * (x[2] - x[1] * x[1]))
}

x0 = c(-1.2, 1)
```

```
nloptr::nloptr(
  x0 = x0,
  eval_f = f, eval_grad_f = grad,
  opts = list(
    "algorithm" = "NLOPT_LD_LBFGS",
    "xtol_rel" = 1.0e-8
  )
)
```

```
Call:

nloptr::nloptr(x0 = x0, eval_f = f, eval_grad_f
= grad, opts = list(algorithm =
"NLOPT_LD_LBFGS",
    xtol_rel = 1e-08))


Minimization using NLopt version 2.7.1

NLopt solver status: 1 ( NLOPT_SUCCESS: Generic
success return value. )

Number of Iterations....: 56
Termination conditions:  xtol_rel: 1e-08
Number of inequality constraints:  0
Number of equality constraints:    0
```

Constrained Example



$$\min_{x \in R^n} \sqrt{x_2}$$

$$\text{s.t.} \quad x_2 \geq 0$$

$$(a_1 x_1 + b_1)^3 - x_2 \leq 0$$

$$(a_2 x_1 + b_2)^3 - x_2 \leq 0$$

where $a_1 = 2, b_1 = 0, a_2 = -1$, and $b_2 = 1$.

Implementation

```r
# Objective function & gradient
f = function(x, a, b) {
  sqrt(x[2])
}
grad_f = function(x, a, b)  {
  c(0, 0.5 / sqrt(x[2]))
}

# Constraint function
g = function(x, a, b) {
  (a * x[1] + b) ^ 3 - x[2]
}

# Jacobian of constraint
jac_g = function(x, a, b) {
  rbind(
    c(3 * a[1] * (a[1] * x[1] + b[1]) ^ 2, -
    c(3 * a[2] * (a[2] * x[1] + b[2]) ^ 2, -
  )
}

a = c(2, -1)
b = c(0, 1)

```

```
Call:
nloptr::nloptr(x0 = c(1.234, 5.678), eval_f =
f, eval_grad_f = grad_f,
    lb = c(-Inf, 0), ub = c(Inf, Inf),
eval_g_ineq = g, eval_jac_g_ineq = jac_g,
    opts = list(algorithm = "NLOPT_LD_MMA",
xtol_rel = 1e-08),     a = a, b = b)



Minimization using NLopt version 2.7.1

NLopt solver status: 4 ( NLOPT_XTOL_REACHED:
Optimization stopped because
xtol_rel or xtol_abs (above) was reached. )

Number of Iterations    : 18
```