Optimization - optax & torch

Lecture 24

Dr. Colin Rundel

SGD Libraries

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

- Tensorflow / Keras
- Torch

JAX does not have built-in support for optimization beyond `jax.scipy.optimize.minimize()` (which only supports `BFGS`).

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.

```
1 import optax
2 optax.__version__
```

```
'0.2.8'
```

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)

lm = LinearRegression(fit_intercept=False).fit(X,y)
lm_loss = lr_loss(lm.coef_, X, y).item()
```

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 0x32a86e980>, update=<
```

- 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 the optimizer

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

```
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)
```

```
opt_state
```

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

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(30):
  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([ 3.0097,  0.0169,  0.0046,  0.0117, -0.008 ,  0.0029,  0.0275,
       -0.0024, -0.0021, 12.268 , 44.443 ,  3.6387,  0.0175, 61.3183,
        0.0043,  0.0054,  0.0125, -0.0145, -0.0015,  0.0005,  0.0318],      dtype=float64)
```

```
{ "lm_loss": lm_loss,
  "gd_loss": gd_loss[-1]}
```

```
{'lm_loss': 10105.859508383426, 'gd_loss': Array(10244.7649, dtype=float64)}
```

Optax and mini batches

While we called `sgd()`, the method is really just gradient descent - if we want to do mini-batch, we need to implement the batching ourselves.

```
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)
}
```

```
{'lm_loss': 10105.859508383426,
 'sgd (mb=10)': 10458.8010476452,
 'sgd (mb=100)': 10455.143628883325,
 'sgd (mb=1000)': 10419.084076347228,
 'sgd (mb=10000)': 10195.133454434466}
```

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)
}
```

```
{'lm_loss': 10105.859508383426,
 'sgd (mb=10)': 10963.709829142354,
 'sgd (mb=100)': 10365.713752947544,
 'sgd (mb=1000)': 10116.327374495744,
 'sgd (mb=10000)': 10195.133454434466}
```

Runtime per epoch

Implementation | Runtimes | Scaled

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

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 general comments

- Batch size determines both training time and computing resources
- Generally there should 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 (mb=10)': 24536.597439967307,
 'adam (mb=100)': 10385.231077560382,
 'adam (mb=25)': 12697.144058664155,
 'adam (mb=50)': 11484.064849550556,
 'lm_loss': 10105.859508383426}
```

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)
}
```

```
{'adam (mb=10)': 12462.56275387035,
 'adam (mb=100)': 217415.5995628308,
 'adam (mb=25)': 10545.409745819998,
 'adam (mb=50)': 10181.523600884477,
 'lm_loss': 10105.859508383426}
```

Learning rate schedules

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

Some of the methods make this issue worse (e.g. Adam)

Optax supports a large number of pre-built learning rate schedules which can be passed into any of its optimizers instead of a fixed 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
}
```

```
{'adam (mb=10)': 11179.679407559443,
 'adam (mb=100)': 10341.367689453024,
 'adam (mb=25)': 11414.747137047098,
 'adam (mb=50)': 10754.985906532085,
 'lm_loss': 10105.859508383426}
```

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!

Torch

PyTorch

> PyTorch is an open-source deep learning library, originally developed by Meta Platforms and currently developed with support from the Linux Foundation. The successor to Torch, PyTorch provides a high-level API that builds upon optimised, low-level implementations of deep learning algorithms and architectures, such as the Transformer, or SGD. Notably, this API simplifies model training and inference to a few lines of code. PyTorch allows for automatic parallelization of training and, internally, implements CUDA bindings that speed training further by leveraging GPU resources.
>
> PyTorch utilises the tensor as a fundamental data type, similarly to NumPy. Training is facilitated by a reversed automatic differentiation system, Autograd, that constructs a directed acyclic graph of the operations (and their arguments) executed by a model during its forward pass. With a loss, backpropagation is then undertaken.[4]

```
1 import torch
2 torch.__version__
```

```
'2.11.0'
```

Tensors

are the basic data abstraction in PyTorch and are implemented by the `torch.Tensor` class. They behave in much the same way as the other array libraries we've seen so far (`numpy`, `jax`, etc.) - including the same broadcasting rules.

```
1 torch.zeros(3)
```

```
tensor([0., 0., 0.])
```

```
1 torch.ones(3,2)
```

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
```

```
1 torch.empty(2,2,2)
```

```
tensor([[[0., 0.],
         [0., 0.]],

        [[0., 0.],
         [0., 0.]]])
```

```
1 torch.manual_seed(1234)
```

```
<torch._C.Generator object at 0x309fe7e30>
```

```
1 torch.rand(2,2,2,2)
```

```
tensor([[[[0.02898, 0.40190],
          [0.25984, 0.36664]],

         [[0.05830, 0.70064],
          [0.05180, 0.46814]]],


        [[[0.67381, 0.33146],
          [0.78371, 0.56306]],

         [[0.77485, 0.82080],
          [0.27928, 0.68171]]]])
```

NumPy conversion

It is possible to easily move between NumPy arrays and Tensors via the `from_numpy()` function and `numpy()` method.

```
1 a = np.eye(3,3)
2 torch.from_numpy(a)
```

```
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]], dtype=torch.float64)
```

```
1 b = np.array([1,2,3])
2 torch.from_numpy(b)
```

```
tensor([1, 2, 3])
```

```
1 c = torch.rand(2,3)
2 c.numpy()
```

```
array([[0.2837, 0.6567, 0.2388],
       [0.7313, 0.6012, 0.3043]], dtype=float32)
```

```
1 d = torch.ones(2,2, dtype=torch.int64)
2 d.numpy()
```

```
array([[1, 1],
       [1, 1]])
```

Inplace modification

Many functions have an inplace variant (indicated by a _ suffix) that modifies the tensor rather than creating a new one. This includes both math functions and arithmetic operators.

```
1 a = torch.rand(2,2)
2 print(torch.exp(a))
```

```
tensor([[1.29014, 1.87641],
        [2.62876, 2.09583]])
```

```
1 print(a)
```

```
tensor([[0.25475, 0.62936],
        [0.96651, 0.73995]])
```

```
1 print(torch.exp_(a))
```

```
tensor([[1.29014, 1.87641],
        [2.62876, 2.09583]])
```

```
1 print(a)
```

```
tensor([[1.29014, 1.87641],
        [2.62876, 2.09583]])
```

```
1 a = torch.ones(2, 2)
2 b = torch.rand(2, 2)
3 a+b
```

```
tensor([[1.45172, 1.47573],
        [1.78419, 1.15250]])
```

```
1 print(a)
```

```
tensor([[1., 1.],
        [1., 1.]])
```

```
1 a.add_(b)
```

```
tensor([[1.45172, 1.47573],
        [1.78419, 1.15250]])
```

```
1 print(a)
```

```
tensor([[1.45172, 1.47573],
        [1.78419, 1.15250]])
```

For functions without a _ variant, check if they have an `out` argument which can be used instead - e.g. see

Changing tensor shapes

The `shape` of a tensor can be changed using the `view()` or `reshape()` methods. The former guarantees that the result shares data with the original object (but requires contiguity), the latter may or may not copy the data.

```
1 x = torch.zeros(3, 2)
2 y = x.view(2, 3)
```

```
1 y
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

```
1 x.fill_(1)
```

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
```

```
1 y
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

```
1 x = torch.zeros(3, 2)
2 y = x.t()
```

```
1 x.view(6)
```

```
tensor([0., 0., 0., 0., 0., 0.])
```

```
1 y.view(6)
```

```
RuntimeError: view size is not c
```

```
1 z = y.reshape(6)
2 x.fill_(1)
```

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
```

```
1 y
```

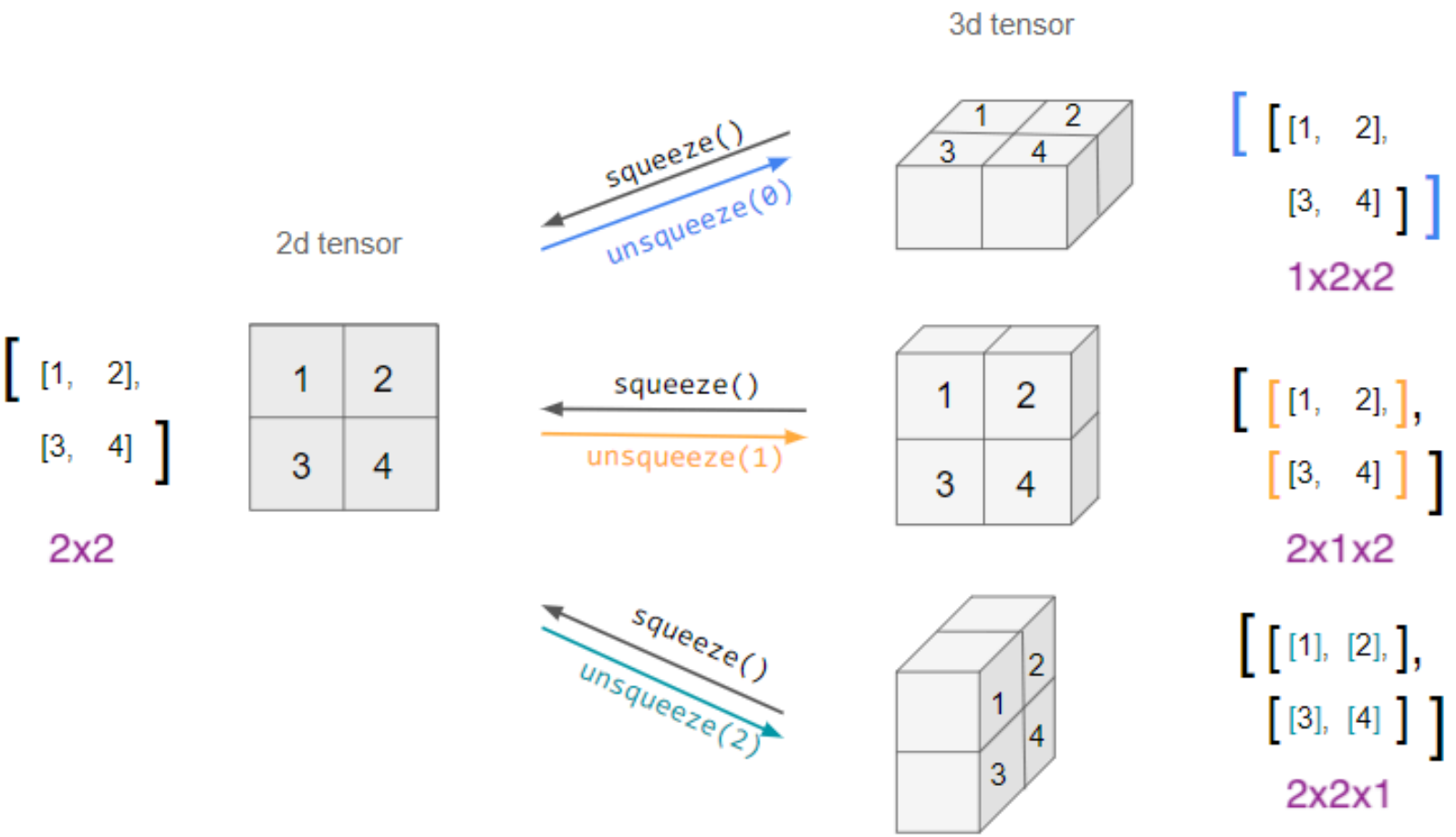
```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

```
1 z
```

```
tensor([0., 0., 0., 0., 0., 0.])
```

Adding or removing dimensions

The `squeeze()` and `unsqueeze()` methods can be used to remove or add length 1 dimension(s) to a tensor.



Autograd

Autograd vs JAX

PyTorch's autograd takes a fundamentally different approach from JAX:

- JAX (functional) - `jax.grad(f)` returns a *new function* that computes the gradient. No state is mutated.
- PyTorch (stateful) - tensors record operations into a computational graph, then `.backward()` populates `.grad` attributes on leaf tensors.

JAX

```
1 def f(x):
2   return jnp.sum(3*x + 2)
3
4 x = jnp.linspace(0, 2, 21)
5 jax.grad(f)(x)
```

```
Array([3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
       3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.], dtype=float64)
```

PyTorch

```
1 x = torch.linspace(
2   0, 2, steps=21, requires_grad=True
3 )
4 y = (3*x + 2).sum()
5
6 y.backward()
7 x.grad
```

```
tensor([3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
```

Tensor expressions

Gradient tracking can be enabled using the `requires_grad` argument at initialization, alternatively the `requires_grad` flag can be set on the tensor or the `enable_grad()` context manager used (via `with`).

```
1 x = torch.linspace(2, 1, steps=11, requires_grad=True)
2 x
```

```
tensor([2.00000, 1.90000, 1.80000, 1.70000, 1.60000, 1.50000, 1.40000, 1.30000, 1.2000
```

```
1 y = torch.linspace(1, 2, steps=11, requires_grad=True)
2 y
```

```
tensor([1.00000, 1.10000, 1.20000, 1.30000, 1.40000, 1.50000, 1.60000, 1.70000, 1.8000
```

```
1 z = torch.log(x*y)
2 z
```

```
tensor([0.69315, 0.73716, 0.77011, 0.79299, 0.80648, 0.81093, 0.80648, 0.79299, 0.7701
```

Computational graph

Basics of the computation graph can be explored via the `next_functions` attribute

```
1 z.grad_fn
```

```
<LogBackward0 object at 0x3adb276a0>
```

```
1 z.grad_fn.next_functions
```

```
((<MulBackward0 object at 0x3adb24790>, 0),)
```

```
1 z.grad_fn.next_functions[0][0].next_functions
```

```
((<AccumulateGrad object at 0x3adb29f90>, 0), (<AccumulateGrad object at 0x3adb26e30>,
```

Autogradient

In order to calculate the gradients we use the `backward()` method on the *output* tensor (must be a scalar), this then makes the grad attribute available for the input (leaf) tensors.

```
1 out = z.sum()
2 out.backward()
3 out
```

```
tensor(8.41071, grad_fn=<SumBackward0>)
```

```
1 y.grad
```

```
tensor([1.00000, 0.90909, 0.83333, 0.76923, 0.71429, 0.66667, 0.62500, 0.58824, 0.5555
```

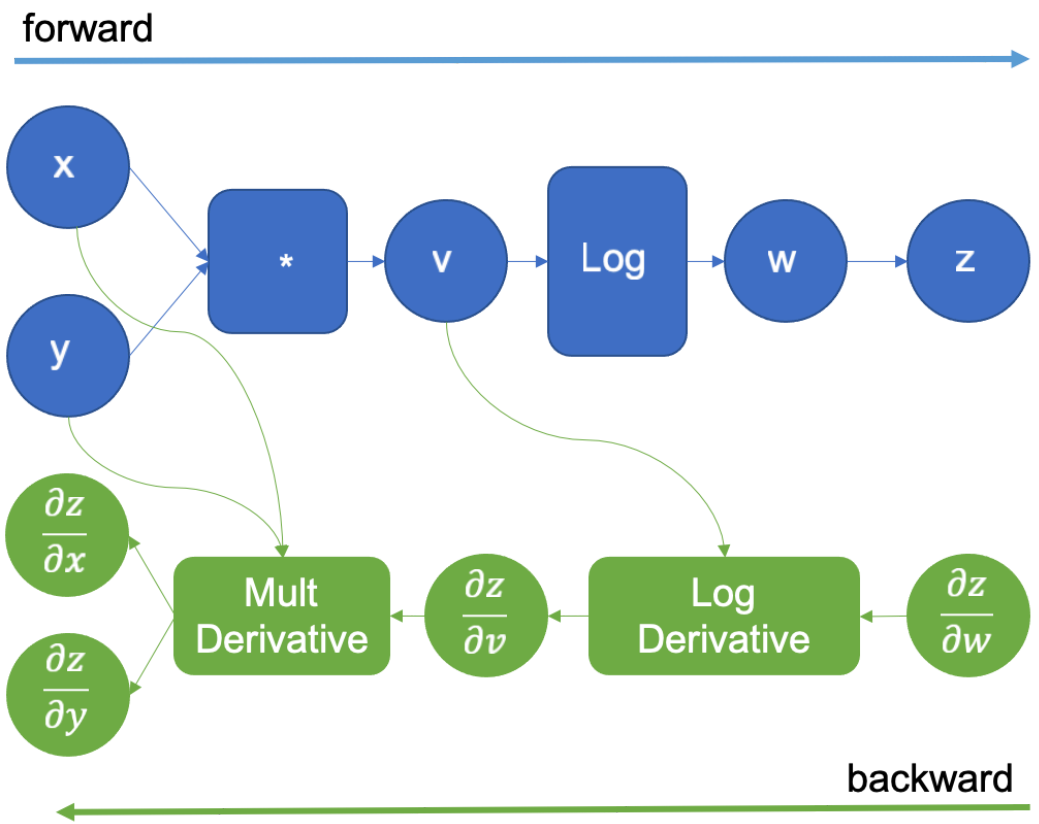
```
1 x.grad
```

```
tensor([0.50000, 0.52632, 0.55556, 0.58824, 0.62500, 0.66667, 0.71429, 0.76923, 0.8333
```

```
1 z.grad
```

Forwards and Backwards passes

Consider the torch tensor expression, $z = \log(x * y)$



A bit more complex

```
1 n = 21
2 x = torch.linspace(0, 2, steps=n, requires_grad=True)
3 m = torch.rand(n, requires_grad=True)
4
5 y = m*x + 2
6
7 y.backward(torch.ones(n))
```

```
1 x.grad
```

```
tensor([0.66622, 0.33432, 0.78929, 0.32164, 0.52472, 0.66884, 0.84361, 0.42651, 0.9561
```

```
1 m.grad
```

```
tensor([0.00000, 0.10000, 0.20000, 0.30000, 0.40000, 0.50000, 0.60000, 0.70000, 0.8000
```

In context you can interpret `x.grad` and `m.grad` as the gradient of `y` with respect to `x` or `m` respectively.

High-level autograd API

Provides for a JAX like functional calculation and evaluation of the jacobian and hessian using torch tensors.

```
def f(x, y):
  return 3*x + 1 + 2*y**2 + x*y
```

```
for x in [0.,1.]:
  for y in [0.,1.]:
    print("x =",x, "y = ",y)
    inputs = (torch.tensor([x]), torch.tensor([y]))
    print(torch.autograd.functional.jacobian(f, inputs),"\n")
```

```
x = 0.0 y =  0.0
(tensor([[3.]]), tensor([[0.]]))

x = 0.0 y =  1.0
(tensor([[4.]]), tensor([[4.]]))

x = 1.0 y =  0.0
(tensor([[3.]]), tensor([[1.]]))

x = 1.0 y =  1.0
(tensor([[4.]]), tensor([[5.]]))
```

```
inputs = (torch.tensor([0.]), torch.tensor([0.]))
torch.autograd.functional.hessian(f, inputs)
```

```
((tensor([[0.]]), tensor([[1.]])), (tensor([[1.]]), tensor([[4.]])))
```

```
inputs = (torch.tensor([1.]), torch.tensor([1.]))
torch.autograd.functional.hessian(f, inputs)
```

```
((tensor([[0.]]), tensor([[1.]])), (tensor([[1.]]), tensor([[4.]])))
```

Linear Regression w/ PyTorch

Same regression example (again)

```
Xt = torch.from_numpy(np.array(X))
yt = torch.from_numpy(np.array(y))
n, k = Xt.shape

bt = torch.zeros(k, dtype=torch.float64, requires_grad=True)
```

```
Xt.shape
```

```
torch.Size([10000, 21])
```

```
yt.shape
```

```
torch.Size([10000])
```

```
bt.shape
```

```
torch.Size([21])
```

```
yt_pred = Xt @ bt
```

```
loss = (yt_pred - yt).pow(2).sum()
loss.item()
```

```
59888326.630047254
```

Gradient descent

```
learning_rate = 1e-5

loss.backward() # Compute the backward pass

with torch.no_grad():
  bt -= learning_rate * bt.grad # Make the step

  bt.grad = None # Reset the gradients
```

```
yt_pred = Xt @ bt
loss = (yt_pred - yt).pow(2).sum()
loss.item()
```

```
38094962.507031634
```

`torch.no_grad()` disables gradient tracking within the context — necessary here because in-place updates to `bt` would

Putting it together

```
bt = torch.zeros(k, dtype=torch.float64, requires_grad=True)

learning_rate = 1e-5
for i in range(31):

  yt_pred = Xt @ bt

  loss = (yt_pred - yt).pow(2).sum()
  if i % 5 == 0:
    print(f"Step: {i},\tloss: {loss.item():.4f}")

  loss.backward()

  with torch.no_grad():
    bt -= learning_rate * bt.grad
    bt.grad = None
```

Putting it together

```
Step: 0,    loss: 59888326.6300
Step: 5,    loss: 6266083.4579
Step: 10,   loss: 669796.6821
Step: 15,   loss: 80312.2264
Step: 20,   loss: 17645.4153
Step: 25,   loss: 10922.7506
Step: 30,   loss: 10195.1335
```

Comparing results

```
1 lm.coef_
```

```
array([ 3.0081,  0.0088,  0.0002,  0.0021,  0.00
       -0.0006,  0.0005, 12.2771, 44.4939,  3.64
       -0.0012, -0.0056,  0.014 , -0.0093, -0.00
```

```
1 bt.detach().numpy()
```

```
array([ 3.0095,  0.0155,  0.0038,  0.0101, -0.00
       -0.0021, -0.0017, 12.27  , 44.4532,  3.63
        0.0033,  0.0034,  0.0128, -0.0136, -0.00
```

```
1 pprint(
2   { "lm_loss": lm_loss,
3   "torch_loss": (Xt @ bt - yt).pow(2).sum().item() }
4 )
```

```
{'lm_loss': 10105.859508383426, 'torch_loss': 10163.254649280698}
```

Note - `bt.detach()` is needed here to obtain a tensor detached from the computation graph (e.g. for converting to

Using a torch model

A simple model

```
class Model(torch.nn.Module):
    def __init__(self, beta):
        super().__init__()
        beta.requires_grad = True
        self.beta = torch.nn.Parameter(beta)

    def forward(self, X):
        return X @ self.beta

def training_loop(model, X, y, optimizer, n=100):
    losses = []
    for i in range(n):
        y_pred = model(X)

        loss = (y_pred - y).pow(2).sum()
        loss.backward()

        optimizer.step()
        optimizer.zero_grad()

        losses.append(loss.item())

```

Fitting

To fit the model we need to initialize the model object, create an optimizer object (which we attach to our model's parameters), and then we call the training loop.

```
m = Model(beta = torch.zeros(k, dtype=torch.float64))
opt = torch.optim.SGD(m.parameters(), lr=1e-5)

losses = training_loop(m, Xt, yt, opt, n=30)
```

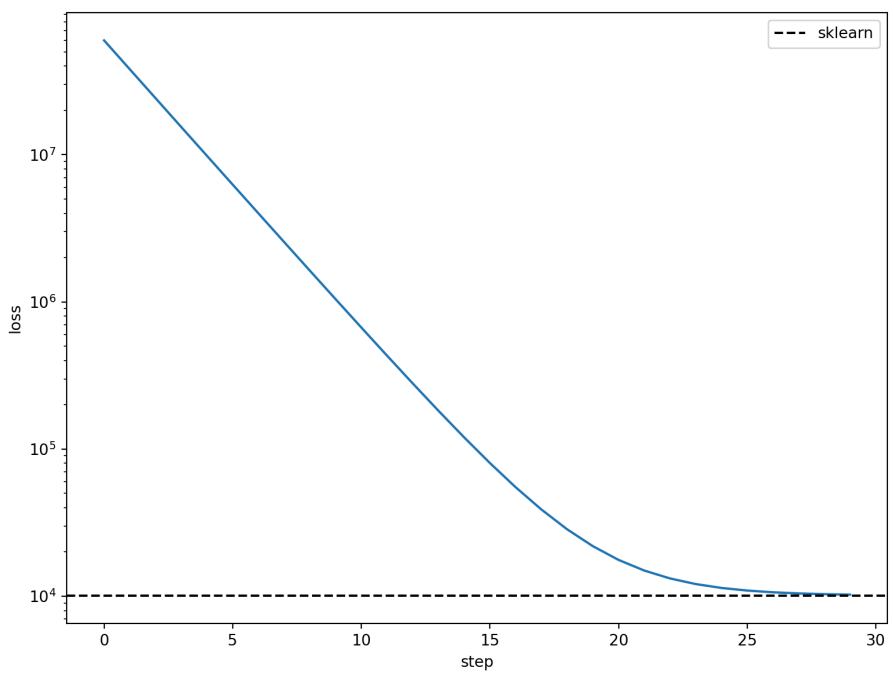
Results

```
1 m.beta
```

```
Parameter containing:
tensor([ 3.00966e+00,  1.68516e-02,  4.60588e-03,  1.17352e-02, -8.00591e-03,  2.92673e-03,  2.750
         6.13183e+01,  4.31299e-03,  5.35222e-03,  1.24837e-02, -1.45383e-02, -1.50216e-03,  5.148
```

```
1 pprint(
2   { "lm_loss": lm_loss,
3     "torch_loss": losses[-1] }
4 )
```

```
{'lm_loss': 10105.859508383426, 'torch_loss': 10244.764879521514}
```



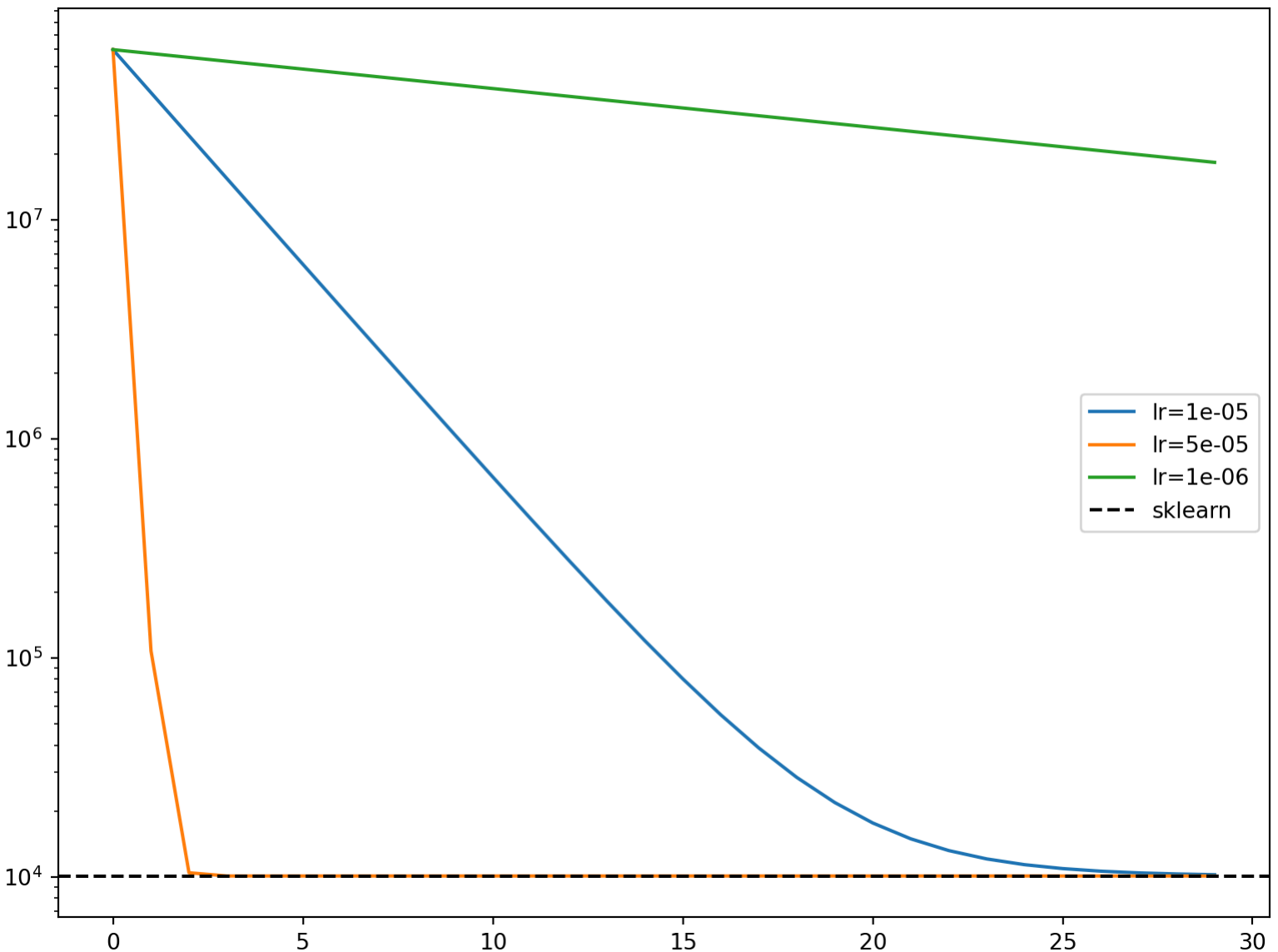
An all-in-one model

```
class Model(torch.nn.Module):
    def __init__(self, k, beta=None):
        super().__init__()
        if beta is None:
          beta = torch.zeros(k, dtype=torch.float64)
        beta.requires_grad = True
        self.beta = torch.nn.Parameter(beta)

    def forward(self, X):
        return X @ self.beta

    def fit(self, X, y, opt, n=30):
      losses = []
      for i in range(n):
          loss = (self.forward(X) - y).pow(2).sum()
          loss.backward()
          opt.step()
          opt.zero_grad()
          losses.append(loss.item())
```

Learning rate and convergence



What about mini-batches & LR?

All of the torch examples so far have used “full-batch” gradient descent. In the next lecture we will cover:

- Mini-batch training via `DataLoader` and `Dataset` classes
- Learning rate schedulers via `lr_scheduler`
- Other optimizers via `torch.optim` (e.g. Adam, RMSProp, etc.)