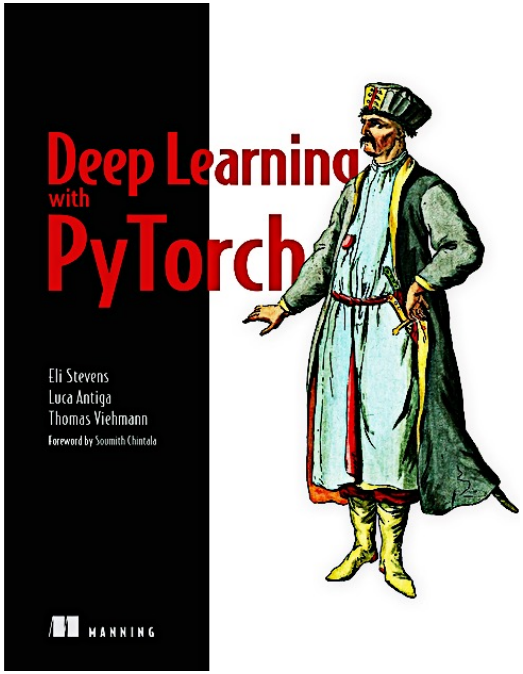
Additional Material: PyTorch





Suggested Literature



PyTorch: The basics (Hello World)

```
import torch
device = torch.device(
    "cuda" if torch.cuda.is_available()
    else "mps" if torch.backends.mps.is_available() # macOS metal
    else "cpu"
)
print(device) # Docker on macOS does not allow metal (or CUDA)
```

```
cpu
```

```
a = torch.ones(3,3) # Create a 3x3 tensor and set all elements to 1
b = torch.ones(3,3)
a + b
```

```
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])
```

```
a = a.to(device)
b = b.to(device)
a + b
```

```
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])
```

ndarray is the backbone of PyTorch

They called it a tensor (I am not a fan of the terminology), but this is not a proper tensor in the Physics sense

PyTorch: Why tensors?

```
// Schematically what is a Python list?
// PyListObject for a
//   ├── ob_size   = 3
//   ├── allocated = maybe >= 3
//   └── ob_item   = pointer to storage buffer
//                     |
//                     v
//              +-------------+-------------+-------------+
//              | PyObject*   | PyObject*   | PyObject*   |
//              +-------------+-------------+-------------+
//                    |             |             |
//                 PyLong(1)     PyLong(2)     PyLong(3)

typedef struct { // In CPython naming, the prefix ob means object
  PyVarObject ob_base;  // the base object header ---------------
  PyObject**  ob_item;  // The object-item storage array --------
                        // Why double pointer?
                        // ob_item -> to 1st slot in the array
                        // Each slot contains a PyObject*
  Py_ssize_t allocated; // capacity ----------------------------
} PyListObject;
```

Python Lists are not contiguous in memory.

Think about it: a list can have heterogeneous types.

How to accomplish this magic?

PyTorch: Why tensors?

```
// A Python object is a runtime value with:
// 1. identity: where this object lives in memory
// 2. type: what operations it supports
// 3. value: the data/content it represents
// Python list object
//   ├── identity: memory address of this list object
//   ├── type:     list
//   └── value:    references to 1, 2, 3
// Examples:
// PyListObject: Python list object
// PyLongObject: Python int object
// PyUnicodeObject: Python str object
// PyFloatObject: Python float object
typedef struct {
  Py_ssize_t ob_refcnt;   // reference count
  PyTypeObject *ob_type;  // points to &PyFloat_Type
  double ob_fval;
} PyFloatObject;
```

Python Lists are not contiguous in memory.

Think about it: a list can have heterogeneous types.

How to accomplish this magic?

PyTorch: Why tensors?

```
typedef struct { // Extension of PyObject that adds ob_size
    Py_ssize_t ob_refcnt;   // reference count
    PyTypeObject* ob_type;  // What is an object?
                            // Python's abstraction for data.
    Py_ssize_t ob_size;     // number of variable-size items
} PyVarObject;

// PyTypeObject: C struct that describes type of an object
// Examples:
// PyList_Type describes Python list
// PyLong_Type describes Python int
// PyUnicode_Type describes Python str
// PyFloat_Type describes Python float

// Summary: Python list has a lot more indirection than C arrays
//          That is bad for performance.
//          That us why Pytorch has "tensors"
```

Python Lists are not contiguous in memory.

Think about it: a list can have heterogeneous types.

How to accomplish this magic?

PyTorch: Why tensors?

```
# tensor with only one number
# Use .item() to access the value
pt = torch.tensor([1])
pt, pt.item()
```

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

```
points = torch.tensor([[4.0,1.0], [5.,3.], [2.,1.]])
points
```

```
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

```
points.shape
```

```
torch.Size([3, 2])
```

```
points[0,1]
```

```
tensor(1.)
```

```
float(points[0,1]) # Converts:
                   # From: Pytorch scalar (torch.float32 - 32 bit)
                   # to Python Float (which is a PyFloatObject in C)
                   # PyFloatObject holds a C double (64 bit)
                   # So user must be a bit careful
```

```
1.0
```

Torch tensor can accept a Python list in the constructor

Need to be a bit careful with type conversion

torch.tensor() always deep copy the data (e.g., the list)

PyTorch: Why tensors?

```
points[1:]
```

```
tensor([[5., 3.],
        [2., 1.]])
```

```
points[1:,:]
```

```
tensor([[5., 3.],
        [2., 1.]])
```

```
points[1:,0]
```

```
tensor([5., 2.])
```

```
points[0:3:2,:] # from zero to 3 (exclusive) in steps of 2
```

```
tensor([[4., 1.],
        [2., 1.]])
```

```
points[None] # adds a dimension – note the triple [[[
```

```
tensor([[[4., 1.],
         [5., 3.],
         [2., 1.]]])
```

```
out = points.pow(2).sum() # these operations produce a new Tensor
                          # they don't affect the original points
out.item()
```

```
56.0
```

```
points.unsqueeze(dim=0) # same as points[None]
```

```
tensor([[[4., 1.],
         [5., 3.],
         [2., 1.]]])
```

Torch tensors are contiguous in memory

Torch Tensors allow slicing similar to Numpy

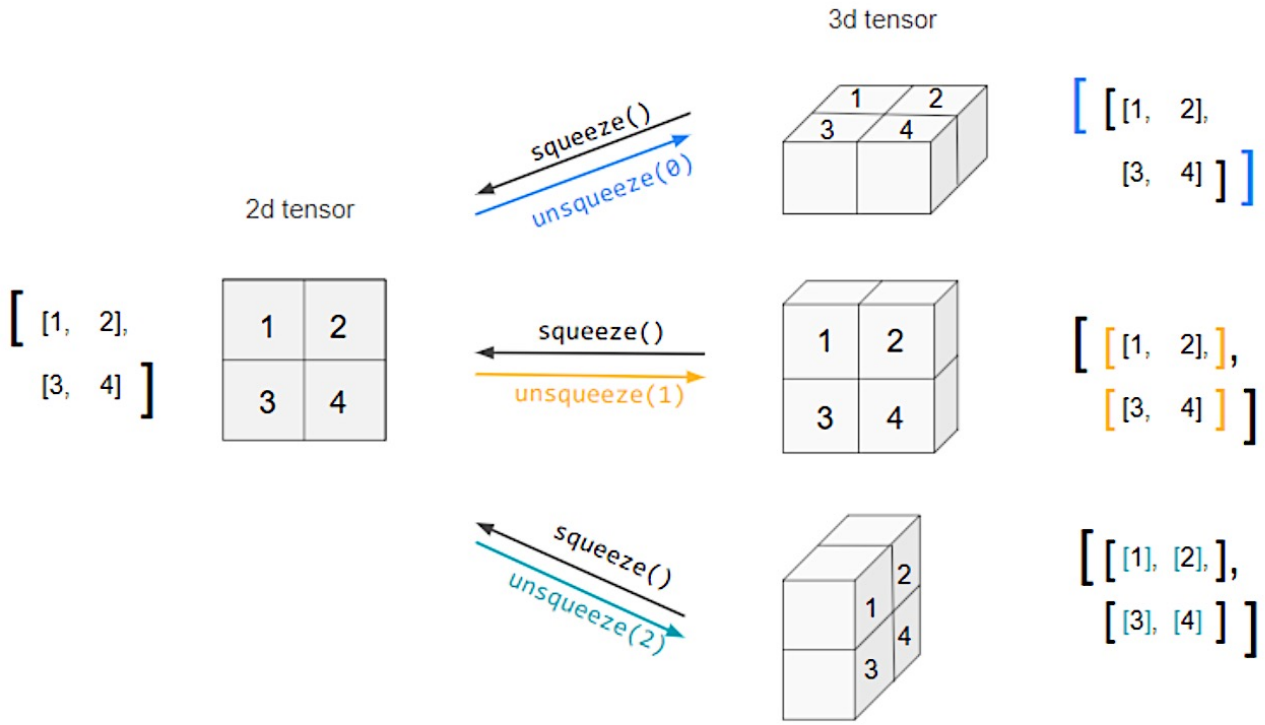
Many numerical manipulations are allowed by the API; they copy the tensor unless it has a “_”)

PyTorch: **unqueeze** Operation

210

`unsqueeze` turns an n.d. tensor into an (n+1).d. one by adding an extra dimension of depth 1. However, since it is ambiguous which axis the new dimension should lie across (i.e. in which direction it should be "unsqueezed"), this needs to be specified by the `dim` argument.

e.g. `unsqueeze` can be applied to a 2d tensor three different ways:



The resulting unsqueezed tensors have the same information, but the *indices* used to access them are different.

Common operation we need before learning broadcasting

PyTorch: **unqueeze** Operation

```
w = torch.tensor([0.2126, 0.7152, 0.0722])
unszdw = w.unsqueeze(-1).unsqueeze(-1)
unszdw.shape, unszdw
```

```
(torch.Size([3, 1, 1]),
 tensor([[[0.2126]],

         [[0.7152]],

         [[0.0722]]]))
```

```
So that is unszdw basically (so 3 1x1 matrices
[
  [
    [0.2126]
  ],
  [
    [0.7152]
  ],
  [
    [0.0722]
  ]
]
```

Common operation we need before learning broadcasting

PyTorch: Broadcast

Broadcast is an **element-wise** operation that should not be thought of as matrix addition or multiplication

```
#Broadcasting
tensor_a = torch.tensor([[1, 2, 3]])  # shape (1, 3)
result = tensor_a + 10
result
```

```
tensor([[11, 12, 13]])
```

```
Explanation:

tensor_a.shape = (1, 3) and tensor_b.shape = (1).

Broadcasting compares shapes from the right, so
tensor_a:   1   3
tensor_b:       1 => Pytorch pretends that tensor_b.shape is:  1  1 (to match tensor_a dimension)

Now Compare dimension by dimension
dimension 0: 1 vs 1  -> same
dimension 1: 1 vs 3  -> broadcast 1 to 3

The operation is equivalent to pretending that tensor_b became [[10, 10, 10]]

so we have [[1, 2, 3]] + [[10, 10, 10]] = [[11, 12, 13]]
```

Notice the data duplication when dim size increases

```python
tensor_a = torch.tensor([[1, 2, 3],
                         [4, 5, 6]])      # shape: (2, 3)
tensor_b = torch.tensor([[[10, 20, 30]],
                         [[40, 50, 60]]]) # shape: (2, 1, 3)
tensor_a + tensor_b
```

```
tensor([[[11, 22, 33],
         [14, 25, 36]],

        [[41, 52, 63],
         [44, 55, 66]]])
```

```
Explanation: tensor_a.shape = (2, 3) and tensor_b.shape = (2, 1, 3).

Broadcasting compares shapes from the right, so
tensor_a:       2   3, Pytorch pretends tensor_a.shape is:  1   2   3
tensor_b:    2   1   3

Now Compare dimension by dimension
dimension 0: 1 vs 2  -> broadcast 1 to 2
dimension 1: 2 vs 1  -> broadcast 1 to 2
dimension 2: 3 vs 3  -> same

Operation: pretending that tensor_a became
[
  [
    [1, 2, 3],
    [4, 5, 6]
  ],
  [
    [1, 2, 3],
    [4, 5, 6]
  ]
]
and tensor b became
[
  [
    [10, 20, 30]
    [10, 20, 30]
  ],
  [
    [40, 50, 60]
    [40, 50, 60]
  ]
]

Then add element-wise
```

PyTorch: Broadcast

Broadcast is an **element-wise** operation that should not be thought of as matrix addition or multiplication

Dim comparison goes from right to left

When increasing the dimension size, there is a duplication of data

PyTorch: Broadcast

```
w = torch.tensor([0.2126, 0.7152, 0.0722])
unszdw = w.unsqueeze(-1).unsqueeze(-1)
unszdw.shape, unszdw
```

```
(torch.Size([3, 1, 1]),
 tensor([[[0.2126]],

         [[0.7152]],

         [[0.0722]]]))
```

```
So that is unszdw basically (so 3 1x1 matrices
[
  [
    [0.2126]
  ],
  [
    [0.7152]
  ],
  [
    [0.0722]
  ]
]

So if you broadcast against img_t which has shape (3, 5, 5)
Broadcasting compares shapes from the right, so
unszdw:   3   1   1
img_t:    3   5   5

dimension 0: 3 vs 3  -> same
dimension 1: 1 vs 5  -> broadcast 5
dimension 2: 1 vs 5  -> broadcast 5

so the unszdw will become
[
  [
    [0.2126, 0.2126, 0.2126, 0.2126, 0.2126],
    [0.2126, 0.2126, 0.2126, 0.2126, 0.2126]
    [0.2126, 0.2126, 0.2126, 0.2126, 0.2126]
    [0.2126, 0.2126, 0.2126, 0.2126, 0.2126]
    [0.2126, 0.2126, 0.2126, 0.2126, 0.2126]
  ],
  [
    [0.7152, 0.7152, 0.7152, 0.7152, 0.7152]
    [0.7152, 0.7152, 0.7152, 0.7152, 0.7152]
    [0.7152, 0.7152, 0.7152, 0.7152, 0.7152]
    [0.7152, 0.7152, 0.7152, 0.7152, 0.7152]
    [0.7152, 0.7152, 0.7152, 0.7152, 0.7152]
  ],
  [
    [0.0722, 0.0722, 0.0722, 0.0722, 0.0722]
    [0.0722, 0.0722, 0.0722, 0.0722, 0.0722]
    [0.0722, 0.0722, 0.0722, 0.0722, 0.0722]
    [0.0722, 0.0722, 0.0722, 0.0722, 0.0722]
    [0.0722, 0.0722, 0.0722, 0.0722, 0.0722]
  ]
]
So every pixel of the image is multiplied by the weight.
Each RGB color has its own weight
```

Broadcast is an **element-wise** operation. So new gray scale

```
img_w = img_t * unszdw
img_gray_w = img_w.sum(-3)
img_gray_w.shape
```

```
torch.Size([5, 5])
```

What happens on batch image with shape (2, 3, 5, 5) broadcast to unszdw with shape (3,1,1)?

PyTorch: Broadcast

```
bath_w = batch_t * unszdw
batch_gray_w = bath_w.sum(-3)
batch_gray_w.shape
```

```
torch.Size([2, 5, 5])
```

```
So if you broadcast unszdw against batch_t,
which has shape (2, 3, 5, 5). Broadcasting
compares shapes from the right, so

unszdw:   1  3   1   1
batch_t:  2  3   5   5

dimension 0: 1 vs 2  -> broadcast 2
dimension 1: 3 vs 3  -> same
dimension 2: 1 vs 5  -> broadcast 5
dimension 3: 1 vs 5  -> broadcast 5

So the final vector will have shape 2, 3, 5, 5
```

What happens on batch image with shape (2, 3, 5, 5) broadcast to unszdw with shape (3,1,1)?

Remember that unszdw will become (1,3,1,1) and that broadcast will copy the first dim to apply the same weighting scheme to all images in the batch

PyTorch: dtypes

```
Tensor types (not comprehensive)

torch.float32 = 32-bit float (equiv of C float). Equiv name torch.float
torch.float64 = 64-bit float (equiv of C double). Equiv name torch.double
torch.float16 = 16-bit float
torch.int16   = signed 16-bit integer
torch.int32   = signed 32-bit integer (equiv of C int)
torch.int64   = signed 64-bit integer (def for torch idxs)
torch.bool    = boolean
```

```
x = torch.rand(5, dtype=torch.float64)
y = torch.rand(5, dtype=torch.float32)
x.dtype, y.dtype
```

```
(torch.float64, torch.float32)
```

```
y.double().dtype # casting
```

```
torch.float64
```

```
z = x*y
z.dtype # automatic casting upward
```

```
torch.float64
```

While in scientific computation we need 64-bit double precision, it is unclear if we need more than 32-bit (maybe even 16-bit) on ML (even on ML emulators that emulate a scientific data vector, as we can have an output layer to cast them upward).

PyTorch: Tensor operations

Type of tensor operations (funcs = functions)

- Creation ops: funcs for constructing a tensor
- Indexing, slicing, mutation ops: funcs for changing the shape, stride, or content (e.g., **transpose**)
- Reduction ops: funcs for computing aggregate values (e.g., **mean**, **std**, **norm**)
- Comparison ops: numerical predicates (e.g., **equal**, **max**)
- Spectral ops: funcs that operate in the frequency domain
- BLAS/LAPAC ops: linear algebra
- Random Sampling: gen values by drawing random numbers
- Serialization: funcs for saving/loading tensors
- Parallelism: funcs for controlling num of threads

PyTorch: In-place ops

```
a = torch.rand(3, 2, dtype=torch.float32)
a
```

```
tensor([[0.6922, 0.2869],
        [0.0850, 0.9297],
        [0.0998, 0.9049]])
```

```
a.add_(2)
a
```

```
tensor([[2.6922, 2.2869],
        [2.0850, 2.9297],
        [2.0998, 2.9049]])
```

```
a.mul_(3)
```

```
tensor([[8.0765, 6.8606],
        [6.2550, 8.7891],
        [6.2993, 8.7147]])
```

```
a.sqrt_()
```

```
tensor([[2.8419, 2.6193],
        [2.5010, 2.9646],
        [2.5098, 2.9521]])
```

```
a.t_()
```

```
tensor([[2.8419, 2.5010, 2.5098],
        [2.6193, 2.9646, 2.9521]])
```

```
a.zero_()
a
```

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

In-place functions (i.e., functions that alter the tensor itself w/o copying it) always have the "_" suffix.

With the suffix, output involves a deep copy (can save significant memory)

PyTorch: In-place ops

`Tensor.lerp`	See `torch.lerp()`
`Tensor.lerp_`	In-place version of `lerp()`
`Tensor.lgamma`	See `torch.lgamma()`
`Tensor.lgamma_`	In-place version of `lgamma()`
`Tensor.log`	See `torch.log()`
`Tensor.log_`	In-place version of `log()`
`Tensor.logdet`	See `torch.logdet()`
`Tensor.log10`	See `torch.log10()`
`Tensor.log10_`	In-place version of `log10()`
`Tensor.log1p`	See `torch.log1p()`
`Tensor.log1p_`	In-place version of `log1p()`
`Tensor.log2`	See `torch.log2()`
`Tensor.log2_`	In-place version of `log2()`

In-place functions (i.e., functions that alter the tensor itself w/o copying it) always have the "_" suffix.

With the suffix, output involves a deep copy (can save significant memory)

PyTorch: In-place ops

`Tensor.asin`	See `torch.asin()`
`Tensor.asin_`	In-place version of `asin()`
`Tensor.arcsin`	See `torch.arcsin()`
`Tensor.arcsin_`	In-place version of `arcsin()`
`Tensor.as_strided`	See `torch.as_strided()`
`Tensor.atan`	See `torch.atan()`
`Tensor.atan_`	In-place version of `atan()`
`Tensor.arctan`	See `torch.arctan()`
`Tensor.arctan_`	In-place version of `arctan()`
`Tensor.atan2`	See `torch.atan2()`
`Tensor.atan2_`	In-place version of `atan2()`

In-place functions (i.e., functions that alter the tensor itself w/o copying it) always have the "_" suffix.

With the suffix, output involves a deep copy (can save significant memory)

PyTorch: In-place ops

In-place functions (i.e., functions that alter the tensor itself w/o copying it) always have the "_" suffix.

```
# how scatter_ works?

dest = torch.zeros(3, 5)

# Define the target indices (class labels for each row)
index = torch.tensor([[1],
                      [4],
                      [0]])
# source value to inject (can be a scalar or matching tensor)
val = 1.0

# Perform the scatter along columns (dim=1)
dest.scatter_(dim=1, index=index, value=val)

print(dest)
```

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

.scatter_ useful, e.g., to create one-hot encoding

PyTorch: In-place ops

In-place funcs create problems with derivatives auto-diff: Pytorch needs to keep track of tensor modifications incompatible with in-place ops

```
x = torch.tensor([2.0], requires_grad=True)

y = x * x      # backward needs the original x = 2
x.add_(1.0)    # in-place modification: x is now 3

y.backward()
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[29], line 4
      1 x = torch.tensor([2.0], requires_grad=True)
      2
      3 y = x * x      # backward needs the original x = 2
----> 4 x.add_(1.0)    # in-place modification: x is now 3
      5
      6 y.backward()

RuntimeError: a leaf Variable that requires grad is being used in an in-place operation.
```

Forward pass: y = x^2; backward needs: dy/dx = 2x

But which x? original x = 2? modified x = 3?

PyTorch: Forward vs Backward pass

What is forward vs backward pass?

1. Forward pass: compute the value. So forward pass answers: What is the prediction? What is the loss?

2. Backward pass: compute how that value changes when inputs/parameters change. Backward passes are based on a computation graph (then walk it in reverse and apply the chain rule)

PyTorch: Forward pass

```
x = torch.tensor(2.0)
w = torch.tensor(3.0, requires_grad=True)
b = torch.tensor(1.0, requires_grad=True)

z = w * x + b
loss = z**2
```

```
The forward pass is just ordinary computation:
x = 2
w = 3
b = 1

z = w*x + b = 3*2 + 1 = 7
loss = z^2 = 49

Computational graph
w ----\
       multiply ----\
x ----/              add ---- z ---- square ---- loss
b ------------------/
```

Forward pass: The ordinary computation

forward pass answers: What is the prediction? What is the loss?

```
Backward pass:
  Walk graph backward and compute gradients
    at values from the forward pass

Autograd graph
w ---\
       multiply ---\
x ---/              add --- z --- square --- loss
b ----------------/

Start:
  ∂loss/∂loss = 1

Step 1: through square loss = z^2

in gradient: ∂loss/∂loss = 1

local derivative: ∂loss/∂z = 2z = 2*7 = 14

out gradient: ∂loss/∂z = 1*14 = 14

Step 2: through add: z = q + b, where q = w*x

in gradient: ∂loss/∂z = 14

local derivatives: ∂z/∂q = 1; ∂z/∂b = 1

out gradients: ∂loss/∂q = 14*1 = 14; ∂loss/∂b = 14*1 = 14

Step 3: through multiply q = w*x

ingradient: ∂loss/∂q = 14

local derivatives: ∂q/∂w = x = 2;  ∂q/∂x = w = 3

out gradients: ∂loss/∂w = 14*2 = 28; ∂loss/∂x = 14*3 = 42

Gradients stored by PyTorch: w.grad = 28; b.grad = 14
x.grad = 42   if x.requires_grad = True
```

PyTorch: Backward pass

Backward pass: Based on a computation graph, walk it in reverse & apply the chain rule

During the forward pass, PyTorch saves values needed later. If we modify x **after the forward pass but before backward**, then some derivatives need to be computed using the updated value. Error

PyTorch: In-place ops

With fwd and backward understanding: in-place operators AFTER computing backward pass: ok

```
After loss.backward(), PyTorch usually frees the graph.
So changing tensors after backward does not damage that backward pass.

This is exactly what optimizers do:

  loss.backward()        # compute gradients

  with torch.no_grad():
    w -= lr * w.grad     # in-place parameter update

  w.grad.zero_()

The optimizer updates parameters after gradients are computed.

two caveats:

1. If you want to call backward again on the same graph,
   in-place changes can still break things.

2. If the tensor is a leaf tensor with requires_grad=True,
   PyTorch may forbid in-place modification unless you use torch.no_grad().
```

Tensor metadata: size, offset, stride

```
a = torch.tensor([[4,1],[3,7],[5,5]],dtype=torch.int8)
astr = a.storage()
astr
```

```
 4
 1
 3
 7
 5
 5
[torch.storage.TypedStorage(dtype=torch.int8, device=cpu) of size 6]
```

Tensor stores all data in 1D vectors

```
a.stride() # memory step needed to move along each dimension
           # lay out in memory. Memory is contiguous row-major
```

```
(2, 1)
```

```
a.storage_offset()
```

```
0
```

```
a.size() # shape of the tensor
```

```
torch.Size([3, 2])
```

```
a.numel() #total number of elements
```

```
6
```

```
a.t_()  # now column-major
a.stride()
```

```
(1, 2)
```

```
# When stride would be different on each dim from size-1?
# Just think of a 3d-tensor
```

```
a = torch.arange(24).reshape(2, 3, 4)
a.size()
```

```
torch.Size([2, 3, 4])
```

```
a.stride()
```

```
(12, 4, 1)
```

Metadata to 1D vector: size, offset, stride

Size is not the total number of elements: that is returned by **numel()**

Tensor metadata: size, offset, stride

```
a = torch.rand(3, 4, 4, dtype=torch.float32)
a.stride()
```

```
(16, 4, 1)
```

```
b = a[1]
b.storage_offset()
```

```
a
```

```
tensor([[[0.9927, 0.3058, 0.3027, 0.1754],
         [0.5305, 0.8424, 0.8091, 0.6382],
         [0.5076, 0.1136, 0.3794, 0.0979],
         [0.2849, 0.9692, 0.8741, 0.8262]],

        [[0.3362, 0.2258, 0.3527, 0.3306],
         [0.7182, 0.1836, 0.1331, 0.8953],
         [0.7201, 0.7025, 0.7633, 0.1668],
         [0.6413, 0.3140, 0.6210, 0.0681]],

        [[0.2827, 0.1635, 0.3122, 0.4171],
         [0.2874, 0.7295, 0.2105, 0.6035],
         [0.4842, 0.3917, 0.0233, 0.2759],
         [0.2742, 0.0654, 0.7237, 0.2280]]])
```

```
b[0]=1
```

```
a
```

```
tensor([[[0.9927, 0.3058, 0.3027, 0.1754],
         [0.5305, 0.8424, 0.8091, 0.6382],
         [0.5076, 0.1136, 0.3794, 0.0979],
         [0.2849, 0.9692, 0.8741, 0.8262]],

        [[1.0000, 1.0000, 1.0000, 1.0000],
         [0.7182, 0.1836, 0.1331, 0.8953],
         [0.7201, 0.7025, 0.7633, 0.1668],
         [0.6413, 0.3140, 0.6210, 0.0681]],

        [[0.2827, 0.1635, 0.3122, 0.4171],
         [0.2874, 0.7295, 0.2105, 0.6035],
         [0.4842, 0.3917, 0.0233, 0.2759],
         [0.2742, 0.0654, 0.7237, 0.2280]]])
```

offset appears on subtensors

Why? No deep copy (that can be catastrophic)

How to force a deep copy? **clone()**

Tensor metadata: size, offset, stride

```
a
```

```
tensor([[[0.9927, 0.3058, 0.3027, 0.1754],
         [0.5305, 0.8424, 0.8091, 0.6382],
         [0.5076, 0.1136, 0.3794, 0.0979],
         [0.2849, 0.9692, 0.8741, 0.8262]],

        [[1.0000, 1.0000, 1.0000, 1.0000],
         [0.7182, 0.1836, 0.1331, 0.8953],
         [0.7201, 0.7025, 0.7633, 0.1668],
         [0.6413, 0.3140, 0.6210, 0.0681]],

        [[0.2827, 0.1635, 0.3122, 0.4171],
         [0.2874, 0.7295, 0.2105, 0.6035],
         [0.4842, 0.3917, 0.0233, 0.2759],
         [0.2742, 0.0654, 0.7237, 0.2280]]])
```

```
c = a[1].clone()
c[0] = 2
a
```

```
tensor([[[0.9927, 0.3058, 0.3027, 0.1754],
         [0.5305, 0.8424, 0.8091, 0.6382],
         [0.5076, 0.1136, 0.3794, 0.0979],
         [0.2849, 0.9692, 0.8741, 0.8262]],

        [[1.0000, 1.0000, 1.0000, 1.0000],
         [0.7182, 0.1836, 0.1331, 0.8953],
         [0.7201, 0.7025, 0.7633, 0.1668],
         [0.6413, 0.3140, 0.6210, 0.0681]],

        [[0.2827, 0.1635, 0.3122, 0.4171],
         [0.2874, 0.7295, 0.2105, 0.6035],
         [0.4842, 0.3917, 0.0233, 0.2759],
         [0.2742, 0.0654, 0.7237, 0.2280]]])
```

offset appears on subtensors

Why? No deep copy (that can be catastrophic)

How to force a deep copy? **clone()**

Tensor metadata: size, offset, stride

```
# t() transpose without deep copy (be careful)
a = torch.tensor([[4,1],[3,7],[5,5]],dtype=torch.int8)
a
```

```
tensor([[4, 1],
        [3, 7],
        [5, 5]], dtype=torch.int8)
```

```
b = a.t()
b
```

```
tensor([[4, 3, 5],
        [1, 7, 5]], dtype=torch.int8)
```

```
print(a.untyped_storage().data_ptr() == b.untyped_storage().data_ptr())
```

```
True
```

```
a.stride(), b.stride()
```

```
((2, 1), (1, 2))
```

```
c = a.t().clone()
print(a.untyped_storage().data_ptr() == c.untyped_storage().data_ptr())
```

```
False
```

```
a = torch.ones(3,4,5)
b = a.transpose(0,2)
a.shape, b.shape
```

```
(torch.Size([3, 4, 5]), torch.Size([5, 4, 3]))
```

```
print(a.untyped_storage().data_ptr() == b.untyped_storage().data_ptr())
```

```
True
```

How to force a deep copy?

clone()

Even on **t()** no deep copy

transpose(d1, d2) works for nd-tensors

```
a = torch.tensor([[4,1],[3,7],[5,5]],dtype=torch.int8)
b = a.t()
a.is_contiguous(), b.is_contiguous()
```

```
(True, False)
```

```
c = b.contiguous()
c.is_contiguous()
```

```
True
```

```
b
```

```
tensor([[4, 3, 5],
        [1, 7, 5]], dtype=torch.int8)
```

```
c
```

```
tensor([[4, 3, 5],
        [1, 7, 5]], dtype=torch.int8)
```

```
a.storage()
```

```
 4
 1
 3
 7
 5
 5
[torch.storage.TypedStorage(dtype=torch.int8, device=cpu) of size 6]
```

```
c.storage()
```

```
 4
 3
 5
 1
 7
 5
[torch.storage.TypedStorage(dtype=torch.int8, device=cpu) of size 6]
```

```
print("a:", a.size(), a.stride(), a.is_contiguous())
print("b:", b.size(), b.stride(), b.is_contiguous())
print("c:", c.size(), c.stride(), c.is_contiguous())
```

```
a: torch.Size([3, 2]) (2, 1) True
b: torch.Size([2, 3]) (1, 2) False
c: torch.Size([2, 3]) (3, 1) True
```

```
b is contiguous in a column-major sense,
but PyTorch's default meaning of "contiguous" is row-major / C-order contiguous.
```

Tensor metadata: size, offset, stride

How to ensure contiguous memory?

contiguous()

Contiguous how? C-order contiguous

Numpy Interoperability

```
import numpy as np
a =  torch.ones(3,4)
b = a.numpy() # this is a vew, so no deep copy
b
```

```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]], dtype=float32)
```

```
# How to do deep copy?
b = a.clone().numpy() # CPU tensor, no grad:

b = a.detach().clone().numpy() # Tensor may require grad:

b = a.detach().cpu().numpy().copy() # Tensor may require grad:
```

```
# Going from numpy - also NOT deep copy
a = np.ones((2, 3), dtype=np.float32) # default on np is 64-bit
b = torch.from_numpy(a)
b[0, 0] = 99
print(a)
```

```
[[99.  1.  1.]
 [ 1.  1.  1.]]
```

```
# deep copy
a = np.ones((2, 3), dtype=np.float32)
b = torch.from_numpy(a.copy())
b[0, 0] = 99
print(a)
```

```
[[1. 1. 1.]
 [1. 1. 1.]]
```

.from_numpy() and **.numpy()** are views, so no deep copy

To perform a deep copy, **.detach()** may be required

Saving to file

```
import h5py
```

```
a = torch.rand(3, 4, 4, dtype=torch.float32)
```

```
f = h5py.File('./test.hdf5', 'w')
```

```
dset = f.create_dataset('coords', data=a.numpy())
```

```
f.close()
```

```
f = h5py.File('./test.hdf5', 'r')
dset = f['coords']  # data still in the disk
                    # will be copied as needed
```

```
print(dset[-2:])
```

```
[[[0.09907109 0.07819158 0.6647874  0.6607735 ]
  [0.56342804 0.60521364 0.72460216 0.69622976]
  [0.32347876 0.4826182  0.54997635 0.03906649]
  [0.0950678  0.6122959  0.5100029  0.9745502 ]]

 [[0.27474242 0.67766637 0.40931314 0.18984431]
  [0.7295402  0.77288747 0.15452778 0.25056142]
  [0.93401253 0.7814844  0.08699363 0.9308709 ]
  [0.5461497  0.3643952  0.87187034 0.20747787]]]
```

```
a = torch.from_numpy(dset[-2:])
```

```
print(a)
```

```
tensor([[[0.0991, 0.0782, 0.6648, 0.6608],
         [0.5634, 0.6052, 0.7246, 0.6962],
         [0.3235, 0.4826, 0.5500, 0.0391],
         [0.0951, 0.6123, 0.5100, 0.9746]],

        [[0.2747, 0.6777, 0.4093, 0.1898],
         [0.7295, 0.7729, 0.1545, 0.2506],
         [0.9340, 0.7815, 0.0870, 0.9309],
         [0.5461, 0.3644, 0.8719, 0.2075]]])
```

```
f.close()
```

We adopt the HDF5 file format

Copy as needed (instead of the entire tensor all at once)

Loading from file

```
import csv
import numpy as np
import torch
```

```
wnp = np.loadtxt('winequality-white.csv', dtype=np.float32, delimiter=';',skiprows=1)
wts = torch.from_numpy(wnp)
wlist = next(csv.reader(open('winequality-white.csv'),delimiter=';'))
```

```
wlist
```

```
['fixed acidity',
 'volatile acidity',
 'citric acid',
 'residual sugar',
 'chlorides',
 'free sulfur dioxide',
 'total sulfur dioxide',
 'density',
 'pH',
 'sulphates',
 'alcohol',
 'quality']
```

```
wts.shape, wts.dtype
```

```
(torch.Size([4898, 12]), torch.float32)
```

```
data = wts[:,:-1]
data.shape
```

```
torch.Size([4898, 11])
```

```
target = wts[:,-1].long()          # 'quality' (casted to 64-bit integer)
target, target.shape, target.dtype # categorical values
```

Here we read from **CSV** (common format): read numpy, then convert to **torch.tensor**

Wine-tasting example

```
target = wts[:,-1].long()          # 'quality' (casted to 64-bit integer)
target, target.shape, target.dtype # categorical values
```

```
(tensor([6, 6, 6,  ..., 6, 7, 6]), torch.Size([4898]), torch.int64)
```

```
# One-Hot Encoding Fits Categorical Scores
# from 1 to 10 (but 2 is not 2x better than one)
# 1 -> (1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
#
```

```
toh = torch.zeros( target.shape[0], 10) #to hold target one hot encoding
toh
```

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

```
target.unsqueeze(1)
```

```
tensor([[6],
        [6],
        [6],
        ...,
        [6],
        [7],
        [6]])
```

```
toh.scatter_(dim=1, index=target.unsqueeze(1), value=1.0)
```

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

One-hot Encoding

Quality is a categorical value. Value 4 is not 2x better than 2.

Needs one-hot encoding

Data Normalization

Machine Learning need data to be normalize.

How to do that? Example

```
# Normalizing data
dmean = torch.mean(data, dim=0)
dvar  = torch.var(data, dim=0)
dnorm = (data - dmean)/torch.sqrt(dvar)
dnorm
```

```
tensor([[ 1.7208e-01, -8.1761e-02,  2.1326e-01,  ..., -1.2468e+00,
         -3.4915e-01, -1.3930e+00],
        [-6.5743e-01,  2.1587e-01,  4.7996e-02,  ...,  7.3995e-01,
          1.3422e-03, -8.2419e-01],
        [ 1.4756e+00,  1.7450e-02,  5.4378e-01,  ...,  4.7505e-01,
         -4.3677e-01, -3.3663e-01],
        ...,
        [-4.2043e-01, -3.7940e-01, -1.1915e+00,  ..., -1.3130e+00,
         -2.6153e-01, -9.0545e-01],
        [-1.6054e+00,  1.1666e-01, -2.8253e-01,  ...,  1.0049e+00,
         -9.6251e-01,  1.8574e+00],
        [-1.0129e+00, -6.7703e-01,  3.7852e-01,  ...,  4.7505e-01,
         -1.4882e+00,  1.0448e+00]])
```

```
torch.allclose(torch.sqrt(torch.var(data, dim=0)),
               torch.std(data, dim=0),
               rtol= 1e-05,
               atol= 1e-08) # just to check we could have used std
```

```
True
```

```
# Finding Threshold (for example, wines with bad taste)
bidx = torch.le(target, 3) # also possible to say torch <=3
bidx.shape, bidx.dtype, bidx.sum()
```

```
(torch.Size([4898]), torch.bool, tensor(20))
```

```
bidx # values are indeed boolean
```

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

```
bdata = data[bidx] # similar to what we would have done in Numpy
bdata.shape
```

```
torch.Size([20, 11])
```

```
#other ways to accomplish the same (or similar)
bdata = data[target <= 3]      # bad
mdata = data[(3 < target) & (target < 7)]  # mid
gdata = data[7 <= target]      # good
```

```
bmean = torch.mean(bdata, dim=0)
mmean = torch.mean(mdata, dim=0)
gmean = torch.mean(gdata, dim=0)
bmean.shape, mmean.shape, gmean.shape
```

```
(torch.Size([11]), torch.Size([11]), torch.Size([11]))
```

```
for i, args in enumerate(zip(wlist, bmean, mmean, gmean)):
    print('{:2} {:20} {:6.2f} {:6.2f} {:6.2f}'.format(i, *args))
```

```
 0 fixed acidity         7.60   6.89   6.73
 1 volatile acidity      0.33   0.28   0.27
 2 citric acid           0.34   0.34   0.33
 3 residual sugar        6.39   6.71   5.26
 4 chlorides             0.05   0.05   0.04
 5 free sulfur dioxide  53.33  35.42  34.55
 6 total sulfur dioxide 170.60 141.83 125.25
 7 density               0.99   0.99   0.99
 8 pH                    3.19   3.18   3.22
 9 sulphates             0.47   0.49   0.50
10 alcohol              10.34  10.26  11.42
```

Data Manipulation

Manipulating tensors can be quite similar to manipulating Numpy arrays (e.g., finding a threshold)

Representing text

How to encode text? (letter vs words – trivial e.g.)

```
import requests
import io
import numpy as np
```

```
# pride and prejudice
url = ("https://raw.githubusercontent.com/deep-learning-with-pytorch/"
       "dlwpt-code-2e/refs/heads/main/data/p1ch4/jane-austen/1342-0.txt")
response = requests.get(url)

# Convert bytes to an in-memory file-like object
file_like_object = io.BytesIO(response.content)
# 2. Reset the file pointer (cursor) to the beginning
file_like_object.seek(0)
# 3. Read the contents of the file-like object
file_data = file_like_object.read().decode("utf-8")
```

```
lines = [line for line in file_data.splitlines() if line.strip()]
```

```
line = lines[126].lower().strip()
line
```

```
'“impossible, mr. bennet, impossible, when i am not acquainted with him'
```

Download and break into lines the Pride & Prejudice book (**io.BytesIO** saves file into RAM memory; still need to specify an encoder to convert bytes to str)

one-hot encoding of letters (ASCII - 128 chars)

```
letter_t = torch.zeros(len(line), 128)
letter_t.shape
```

```
torch.Size([70, 128])
```

```
for i, letter in enumerate(line):
    # ord: Return the Unicode code point for a one-character string
    tmp = ord(letter)
    idx = tmp if tmp < 128 else 0 # eliminate quotation mark
    letter_t[i][idx] = 1
```

one-hot encoding of words

```
# return text lower case and strip of punctuation
def clean_words(istr):
    punct = '.,;:"!?”“_-'
    wlist = istr.lower().replace('\n', ' ').split()
    wlist = [ word.strip(punct) for word in wlist ]
    return wlist
clean_words(line)
```

```
['impossible',
 'mr',
 'bennet',
 'impossible',
 'when',
 'i',
 'am',
 'not',
 'acquainted',
 'with',
 'him']
```

```
# set = an unordered collection of unique elements
# w2idx = word to idx
w2idx = {x: i for (i, x) in enumerate(sorted(set(clean_words(file_data))))}
len(w2idx), w2idx['impossible']
```

```
(7261, 3394)
```

```
wt = torch.zeros(len(clean_words(line)), len(w2idx))
for i, word in enumerate(clean_words(line)):
    idx = w2idx['impossible']
    wt[i][idx] = 1
```

Representing text

Not that many letters (dimension is under control).

Many more words (a significant problem).

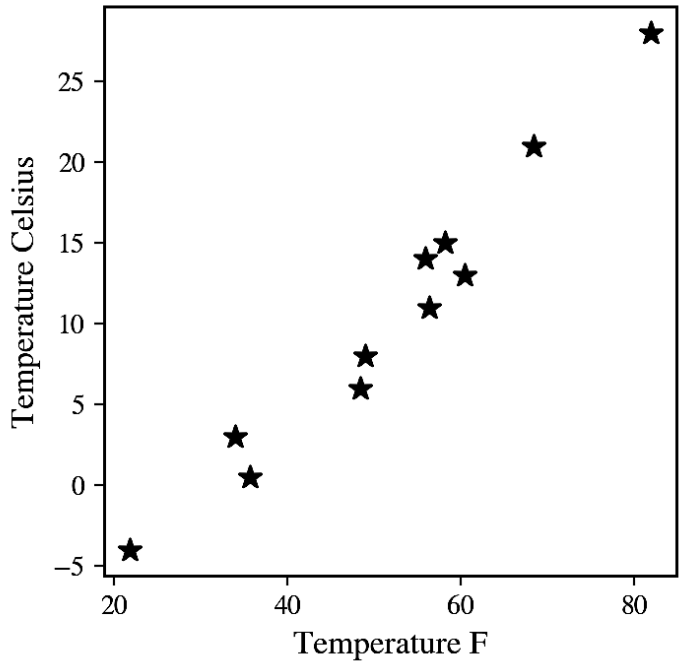
Professional encodings try a middle ground (with pieces of words)

Parameter Estimation 101

Simple case: conversion btw Fahrenheit and Celsius

```
import torch
tC = torch.tensor([0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0])
tF = torch.tensor([35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4])
scale = 0.15
```

```
plt.figure(figsize=(3.75, 3.75))
plt.scatter(tF, tC , marker = '*', c='k', s=70)
fontsize = 12
plt.ylabel('Temperature Celsius',fontsize = fontsize)
plt.xlabel('Temperature F',fontsize = fontsize)
plt.show()
```



Parameter Estimation 101

```python
def model(x, w, b):
    # linear relation
    return w * x + b
def loss_fn(y, t_c):
    squared_diffs = (y - t_c)**2
    return squared_diffs.mean()
```

```python
# Normalize data
tC_mean, tC_std = tC.mean(), tC.std()
tF_mean, tF_std = tF.mean(), tF.std()

tC_norm = (tC - tC_mean) / tC_std
tF_norm = (tF - tF_mean) / tF_std
```

```python
def training_loop1(n_epochs, learning_rate, params):
    for epoch in range(1, n_epochs+1):
        if params.grad is not None:  # <1>
            params.grad.zero_()

        y = model(tF_norm, *params)
        loss = loss_fn(y, tC_norm) # end of the forward pass
        loss.backward() # backward pass (computes grads w/ respect to params)

        with torch.no_grad():
            params -= learning_rate * params.grad

        if epoch % 25 == 0:
            print('Epoch %d, Loss %f' % (epoch, loss.item()))
    return params
```

Simple case: conversion between Fahrenheit and Celsius

Model: linear

Param update following backward pass

```python
import torch.optim as optim
def training_loop2(n_epochs, learning_rate, params, opt=0):
    if opt == 0:
        """Adam adapts learning rates per parameter:
            good for complex problems but can be slower for simple ones
        """
        optimizer = optim.Adam([params], lr=learning_rate)
    elif opt == 1:
        " just follows the gradient - wins on simple problems "
        # mathematical identical to training_loop1
        optimizer = optim.SGD([params], lr=learning_rate)
    for epoch in range(1, n_epochs+1):
        if params.grad is not None:  # init
            params.grad.zero_()

        y = model(tF_norm, *params)
        loss = loss_fn(y, tC_norm) # end of the forward pass
        loss.backward() # backward pass (computes grads w/ respect to params)

        optimizer.step()

        if epoch % 25 == 0:
            print('Epoch %d, Loss %f' % (epoch, loss.item()))
    return params
```

```
The API of optimizers:
-   PyTorch optimizers expect an iterable of leaf tensors (tensors created
    directly with requires_grad=True, whose .grad gets populated by backward()).

    That is why we wrap our single tensor in a list:
      optimizer = optim.Adam([params], lr=learning_rate)
    We could have also written
      optimizer = optim.Adam((params,), lr=learning_rate)       # tuple
      optimizer = optim.Adam(iter([params]), lr=learning_rate)  # any iterable

    We can also send a list of dictionaries to adjust learning parameters
    Example:
      optimizer = optim.Adam([
        {'params': [w], 'weight_decay': 1e-4},         # inherits lr=0.9
        {'params': [b], 'lr': 0.01, 'weight_decay': 0} # overrides to 0.01
      ], lr=0.9)

    But in this case we need to change the API of the function to be
      def training_loop2(n_epochs, learning_rate, w, b, opt=0)
    and called with (so both w and b are leaf tensors)
      training_loop2(n_epochs = 150,
                     learning_rate = 0.9,
                     w = torch.tensor([0.3], requires_grad=True),
                     b = torch.tensor([10.0], requires_grad=True),
                     opt=0)
```

Parameter Estimation 101

PyTorch provides many types of Optimizers

The API of optimizers worth paying attention to (requires an **iterable of leaf tensors**)

Parameter Estimation 101

```
# by default, all tensors have requires_grad=False
params1 = training_loop1(n_epochs = 150,
                         learning_rate = 0.9,
                         params = torch.tensor([0.3, 10.0], requires_grad=True))
```

```
Epoch 25, Loss 0.036734
Epoch 50, Loss 0.034504
Epoch 75, Loss 0.034504
Epoch 100, Loss 0.034504
Epoch 125, Loss 0.034504
Epoch 150, Loss 0.034504
```

```
params2 = training_loop2(n_epochs = 150, learning_rate = 0.9,
                         params = torch.tensor([0.3, 10.0], requires_grad=True),
                         opt=0)
```

```
Epoch 25, Loss 5.268936
Epoch 50, Loss 0.225251
Epoch 75, Loss 0.043877
Epoch 100, Loss 0.036062
Epoch 125, Loss 0.034676
Epoch 150, Loss 0.034506
```

```
params3 = training_loop2(n_epochs = 150, learning_rate = 0.9,
                         params = torch.tensor([0.3, 10.0], requires_grad=True),
                         opt=1)
```

```
Epoch 25, Loss 0.036734
Epoch 50, Loss 0.034504
Epoch 75, Loss 0.034504
Epoch 100, Loss 0.034504
Epoch 125, Loss 0.034504
Epoch 150, Loss 0.034504
```

Simple case: conversion between Fahrenheit and Celsius

Model: linear

Param update following **backward pass**

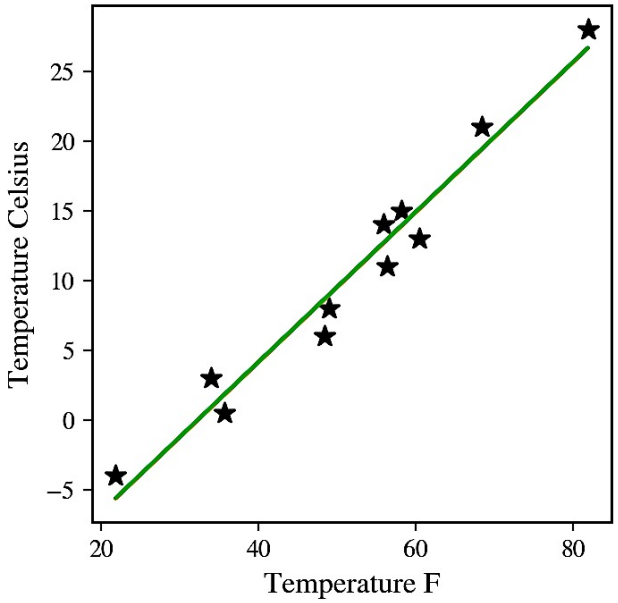
Parameter Estimation 101

```
with torch.no_grad():
    tM_norm = model(tF_norm, *params1)
tM = tM_norm * tC_std + tC_mean

with torch.no_grad():
    tM_norm = model(tF_norm, *params2)
tM2 = tM_norm * tC_std + tC_mean

with torch.no_grad():
    tM_norm = model(tF_norm, *params3)
tM3 = tM_norm * tC_std + tC_mean

plt.figure(figsize=(3.75, 3.75))
plt.scatter(tF, tC , marker = '*', c='k',s=70,label = 'Linear Relation')
plt.plot(tF, tM, label = 'Linear Relation')
plt.plot(tF, tM2, label = 'Linear Relation')
plt.plot(tF, tM3, label = 'Linear Relation')
fontsize = 12
plt.ylabel('Temperature Celsius',fontsize = fontsize)
plt.xlabel('Temperature F',fontsize = fontsize)
plt.show()
```



Simple case: conversion between Fahrenheit and Celsius

Model: linear

Param update following **backward pass**

```
Understanding common language

- A "feature" is an input variable, one piece of information per sample that
  the model uses to make its prediction.

  The number of features: the dimensionality of each sample's input.

  E.g., temp conversion. Per sample, you feed 1 var (T in F):  nfeatures=1

- PyTorch layers like nn.Linear expect.

  Why this is important? A 1D tensor of shape [11] is ambiguous.

  Why? it could mean 11 samples with 1 feature, or 1 sample with 11 features.
```

```
tC0 = torch.tensor([0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0])
tF0 = torch.tensor([35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4])
tC = (tC0 - tC0.mean()) / tC0.std()
tF = (tF0 - tF0.mean()) / tF0.std()

tC = tC.unsqueeze(1) # now this is [nsample, n_features=1]
tF = tF.unsqueeze(1) # now this is [nsample, n_features=1]

nsamples = tC.shape[0]
```

```
# Train - validation split
nval = int(0.2 * nsamples)
shuffled_indices = torch.randperm(nsamples)

train_indices = shuffled_indices[:-nval] # explain the minus sign below

val_indices = shuffled_indices[-nval:]

train_indices, val_indices
```

```
(tensor([ 8, 10,  5,  2,  7,  4,  6,  1,  0]), tensor([3, 9]))
```

```
Explanation of the minus sign in
shuffled_indices[:-n_val] & shuffled_indices[-n_val:]

positions:      0  1  2  3  4  5  6  7  8  9
neg positions: -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

shuffled_indices[:-2]  →  positions 0..7  (train, first 8)
shuffled_indices[-2:]  →  positions 8..9  (val,   last 2)
```

```
tF_train = tF[train_indices]
tC_train = tC[train_indices]
tF_val = tF[val_indices]
tC_val = tC[val_indices]
```

Parameter Estimation 102

API closer to ML **nn.Linear** (linear layer)

First step: train/validation split

Input shaped **[batch_size, nfeatures]**

```
Understanding common language

-   A "feature" is an input variable, one piece of information per
    sample that the model uses to make its prediction.

    The number of features: the dimensionality of each sample's input

    E.g., temp conversion.
    Per sample, you feed 1 var (T in F):  nfeatures=1
```

```
import torch.nn as nn
# one T in F to one T in Celsius. So (1,1)
lmodel = nn.Linear(in_features=1, out_features=1)
# random initialized the weights
lmodel.weight, lmodel.bias
```

```
(Parameter containing:
 tensor([[0.2672]], requires_grad=True),
 Parameter containing:
 tensor([0.2326], requires_grad=True))
```

```
-  lmodel is callable object (similar to functor in C++).

-   What was a functor in C++? A Class that had the operator()
    (operator parenthesis). This allowed an instantiation of the
    class to be used as a function.

    Here will be similar - see below
```

```
# example of forward evaluation
lmodel(torch.randn(5, 1)) #rand = random following Guassian
```

```
tensor([[-0.2449],
        [ 0.1181],
        [ 0.1623],
        [ 0.1479],
        [ 0.5510]], grad_fn=<AddmmBackward0>)
```

Parameter Estimation 102

API closer to ML **nn.Linear** (linear layer)

Constructor takes the number of input/output features (here, 1 temperature in, 1 temperature out)

```
mse = nn.MSELoss()  # Constructor
```

```
def training_loop(n_epochs, optimizer):
    for epoch in range(1, n_epochs + 1):
        # Forward Pass Training set
        tp_train   = lmodel(tF_train)
        loss_train = mse(tp_train, tC_train)

        # Forward Pass Validation set
        tp_val = lmodel(tF_val)
        loss_val = mse(tp_val, tC_val)

        # zero the parameter gradients
        optimizer.zero_grad(set_to_none=True)
        # backward pass (only training)
        loss_train.backward()

        optimizer.step() # update trainable params

        if epoch == 1 or epoch%25 == 0:
            print(f"Epoch {epoch},"
                  f"Training loss  {loss_train.item():.4f},"
                  f" Validation loss {loss_val.item():.4f}")
```

```
- MSE = Mean Squared Error. Equivalent to
    def loss_fn(y, t_c):
      squared_diffs = (y - t_c)**2
      return squared_diffs.mean()

- One subtlety worth knowing: optimizer.zero_grad() takes a set_to_none
  argument that, when true, sets .grad to None instead of writing zeros
  into the tensor.

  Why this is better? (set to None)?
    (1) This is faster (no memory write)
    (2) Fixes a subtle interaction with momentum-based optimizers
        A literal zero gradient still nudges the momentum buffer,
        while a None gradient correctly skips that parameter's
        update entirely.

set_to_none=True is the default in recent PyTorch (≥ 2.0);
in older versions you'd pass it explicitly:

- optimizer.zero_grad(set_to_none=True) is equivalent to

  for p in lmodel.parameters():
    if p.grad is not None:
      p.grad.zero_()

  This connects this example with previous code where we use grad.zero_()
```

Parameter Estimation 102

lmodel (**nn.Linear**) is a callable object

Why do we need to call zero_grad() in PyTorch?

Remaining Question. What is a momentum-based optimizer?

```
What does momentum-based optimizers mean? Example:
  Plain SGD': at each step, take the current ∇ and subtract a learning-
  rate-scaled chunk of it (each step depends only on the ∇ right now):

    θ ← θ − lr ∗ ∇L(θ)

  Mo memory of where you came from. Momentum adds a velocity vector v
  that accumulates past gradients with exponential decay:

    v ← μ ∗ v + ∇L(θ)
    θ ← θ − lr ∗ v

  The coefficient μ (~ 0.9) controls how much past matters: at each step
  the existing v is multiplied by 0.9, then the current gradient is added.

  Interesting limiting cases
  (1) Gradient always point to a same direction (let's assume constant)

    v_1 = 0.9 ∗ 0     + g = 1.00 g
    v_2 = 0.9 ∗ 1.00g + g = 1.90 g
    ...
    v_∞                   = 10   g  -> (1-\mu) ∗ v
    multiplies effective step size by ~10× along consistent directions

  (2) |g| = 1 at angle ±60° from x. So g_x = cos 60° = 0.5 (consistent)
      and g_y = ±sin 60° = ±0.866 (flipping each step).

    v_x_1 = 0.9 ∗  0     + 0.5 = 0.500
    v_x_2 = 0.9 ∗  0.500 + 0.5 = 0.950
    ...
    v_x_∞                      = 5.000     = 0.5 / (1 − 0.9)

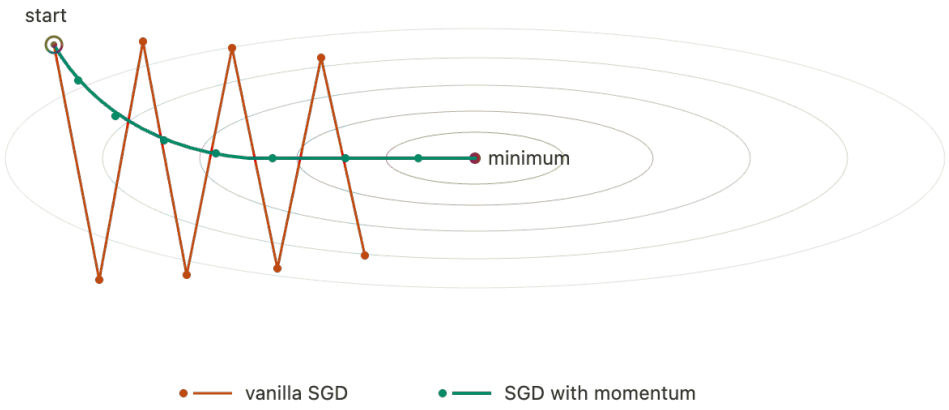
    v_y_1 = 0.9 ∗  0      + 0.866 = +0.866
    v_y_2 = 0.9 ∗  0.866  − 0.866 = −0.087
    v_y_3 = 0.9 ∗ −0.087  + 0.866 = +0.788
    v_y_4 = 0.9 ∗  0.788  − 0.866 = −0.157
    ...
    v_y_∞                         = ±0.456  = ±0.866 / (1 + 0.9)

    Now stack them as a 2D velocity vector and look at its direction:
    step | v_x    v_y     | |v|    angle from x-axis
      1  | 0.500  +0.866  | 1.000  +60.0°    ← same as gradient
      2  | 0.950  −0.087  | 0.954   −5.2°
     ... |                |
      ∞  | 5.000  ±0.456  |   −    ±5.2°

    tan(angle_v) = (1 − μ) / (1 + μ)  ·  tan(angle_g). So momentum is
    rotating the update direction toward whichever component is
    consistent across time.
```

Parameter Estimation 102

Remaining Question. What is a momentum-based optimizer?

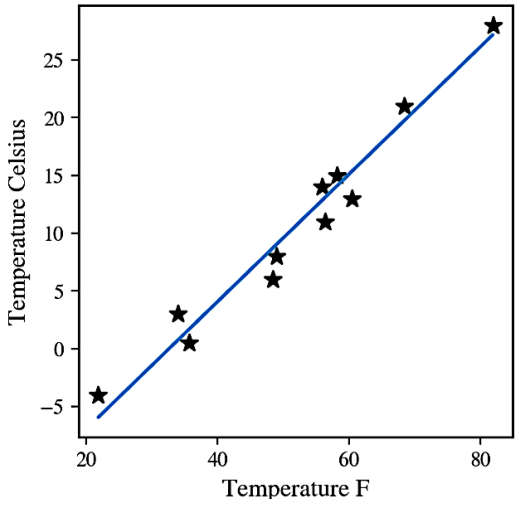


```
training_loop(n_epochs = 250,
              optimizer = optim.SGD(lmodel.parameters(), lr=1e-2))
print()
(lmodel.weight, lmodel.bias)
```

```
Epoch 1,Training loss  1.4711, Validation loss 2.1275
Epoch 25,Training loss  0.7029, Validation loss 0.9690
Epoch 50,Training loss  0.3436, Validation loss 0.4098
Epoch 75,Training loss  0.1797, Validation loss 0.1637
Epoch 100,Training loss  0.1028, Validation loss 0.0634
Epoch 125,Training loss  0.0659, Validation loss 0.0293
Epoch 150,Training loss  0.0478, Validation loss 0.0237
Epoch 175,Training loss  0.0389, Validation loss 0.0293
Epoch 200,Training loss  0.0344, Validation loss 0.0381
Epoch 225,Training loss  0.0321, Validation loss 0.0468
Epoch 250,Training loss  0.0310, Validation loss 0.0542

(Parameter containing:
 tensor([[1.0076]], requires_grad=True),
 Parameter containing:
 tensor([0.0101], requires_grad=True))
```

```
with torch.no_grad():
    # tM comes out as shape [N, 1] because lmodel keeps
    # the feature dim. So we use squeeze()
    tM = lmodel(tF).squeeze() * tC0.std() + tC0.mean()
plt.figure(figsize=(3.75, 3.75))
idx = torch.argsort(tF0) # sort tF0
plt.scatter(tF0, tC0 , marker = '*', c='k',s=70,label = 'Data')
plt.plot(tF0, tM, label = 'Linear Relation')
plt.ylabel('Temperature Celsius',fontsize = 12)
plt.xlabel('Temperature F',fontsize = 12)
plt.show()
```



Parameter Estimation 102

Finally: training

Careful: if you code this in a Jupyter notebook, do not run this cell twice w/o redefining the tensor (weights are saved, so the model will just keep “improving”)

```python
nh = 13     # dim hidden
ndense = 1  # number of dense layers

def dense_layer(n):
  block = nn.Sequential(
    nn.Linear(n, n),
    nn.Tanh(),
  )
  # defensive coding
  #   everything inside the dense block must be trainable
  assert all(p.requires_grad for p in block.parameters())
  return block

lmodel2 = nn.Sequential(
  nn.Linear(1, nh),
  *[dense_layer(nh) for _ in range(ndense)],
  nn.Linear(nh, 1)
)

# Get the overall structure of the Network
for name, param in lmodel2.named_parameters():
    print(name, param.shape)
```

```
0.weight torch.Size([13, 1])
0.bias torch.Size([13])
1.0.weight torch.Size([13, 13])
1.0.bias torch.Size([13])
2.weight torch.Size([1, 13])
2.bias torch.Size([1])
```

Parameter Estimation103

Although not appropriate for our simple example, let's use MLP (it is going to overfit, but we want to practice the coding)

Parameter Estimation103

```
nh = 13     # dim hidden
ndense = 1  # number of dense layers

def dense_layer(n):
  block = nn.Sequential(
    nn.Linear(n, n),
    nn.Tanh(),
  )
  # defensive coding
  #   everything inside the dense block must be trainable
  assert all(p.requires_grad for p in block.parameters())
  return block

lmodel2 = nn.Sequential(
  nn.Linear(1, nh),
  *[dense_layer(nh) for _ in range(ndense)],
  nn.Linear(nh, 1)
)

# Get the overall structure of the Network
for name, param in lmodel2.named_parameters():
    print(name, param.shape)
```

```
0.weight torch.Size([13, 1])
0.bias torch.Size([13])
1.0.weight torch.Size([13, 13])
1.0.bias torch.Size([13])
2.weight torch.Size([1, 13])
2.bias torch.Size([1])
```

In our research/class, a dense layer never changes the dimension of the data vector

```
# Two questions:
  (1) How to count the total number of trainable parameter?
  (2) How to count the number of trainable parameter per block
      Here considering the dense layer one block
```

```
# Question 1
x = sum(p.numel() for p in lmodel2.parameters() if p.requires_grad)
print(f"total trainable: {x}")
```

```
total trainable: 222
```

```
# Question 2: per top-level layer
for name, child in lmodel2.named_children():
  count = sum(p.numel() for p in child.parameters() if p.requires_grad)
  print(f"  layer {name} ({type(child).__name__}): {count}")
```

```
  layer 0 (Linear): 26
  layer 1 (Sequential): 182
  layer 2 (Linear): 14
```

```
# What named child returns?
for name, child in lmodel2.named_children():
  print(name)
print("----------------------")
for name, child in lmodel2.named_children():
  print(child)
```

```
0
1
2
----------------------
Linear(in_features=1, out_features=13, bias=True)
Sequential(
  (0): Linear(in_features=13, out_features=13, bias=True)
  (1): Tanh()
)
Linear(in_features=13, out_features=1, bias=True)
```

Parameter Estimation103

How to count the number of trainable parameters (total and per top layer)

Python question: why the use of **type()** function?

Parameter Estimation103

```
# Question 1
x = sum(p.numel() for p in lmodel2.parameters() if p.requires_grad)
print(f"total trainable: {x}")
```

```
total trainable: 222
```

```
# What named_children returns?
for name, child in lmodel2.named_children():
  print(name)
print("----------------------")
for name, child in lmodel2.named_children():
  print(child)
```

```
0
1
2
----------------------
Linear(in_features=1, out_features=13, bias=True)
Sequential(
  (0): Linear(in_features=13, out_features=13, bias=True
  (1): Tanh()
)
Linear(in_features=13, out_features=1, bias=True)
```

```
# Question 2: per top-level layer
for name, child in lmodel2.named_children():
  # only if required_grad = True, the parameters are trainable
  count = sum(p.numel() for p in child.parameters() if p.requires_grad)
  print(f"  layer {name} ({type(child).__name__}): {count}")
```

```
  layer 0 (Linear): 26
  layer 1 (Sequential): 182
  layer 2 (Linear): 14
```

How to count the number of trainable parameters (total and per top layer)

Python question: why the use of **type()** function?

Parameter Estimation 103

```
# Question 2: per top-level layer
for name, child in lmodel2.named_children():
  # only if required_grad = True, the parameters are trainable
  count = sum(p.numel() for p in child.parameters() if p.requires_grad)
  print(f"  layer {name} ({type(child).__name__}): {count}")
```

```
  layer 0 (Linear): 26
  layer 1 (Sequential): 182
  layer 2 (Linear): 14
```

```
-  Why to access __name__ we needed to call type()?

Short Answer: Because __name__ lives on the class,
              not on instances of it.

Long Answer: (from C++ knowledge) child is an instance, i.e.,
              a particular Linear object, a particular Sequential object.

              Instances generally don't carry their class name as an
              attribute. To get from instance to class you need type(child)

linear = nn.Linear(1, 1)
linear.__name__           # AttributeError - instances don't have this
type(linear)              # torch.nn.modules.linear.Linear
type(linear).__name__     # 'Linear'
linear.__class__.__name__ # 'Linear' (same thing)
```

```
lt = nn.Linear(1, nh)
type(lt), type(lt).__name__, lt.__class__.__name__
```

```
(torch.nn.modules.linear.Linear, 'Linear', 'Linear')
```

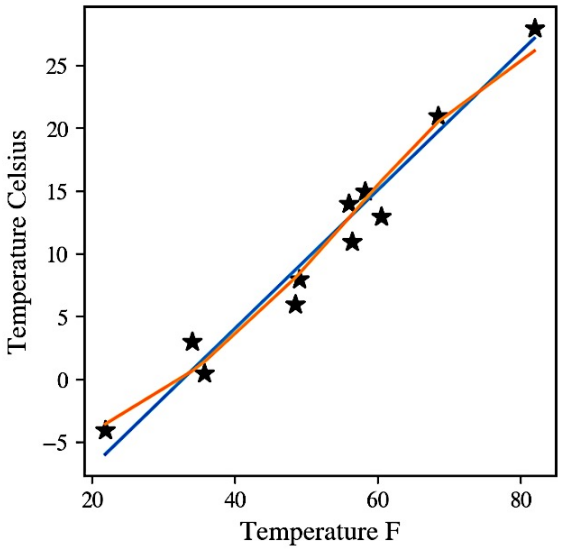
How to count the number of trainable parameters (total and per top layer)

Python question: why the use of **type()** function?

```
import torch.optim as optim
training_loop2(nepochs = 250,
               optimizer = optim.SGD(lmodel2.parameters(), lr=5e-3))
print()
```

```
Epoch 1,Training loss  1.0868, Validation loss 1.9830
Epoch 25,Training loss  0.7700, Validation loss 1.4018
Epoch 50,Training loss  0.5443, Validation loss 0.9517
Epoch 75,Training loss  0.3609, Validation loss 0.5876
Epoch 100,Training loss  0.2195, Validation loss 0.3228
Epoch 125,Training loss  0.1257, Validation loss 0.1639
Epoch 150,Training loss  0.0727, Validation loss 0.0863
Epoch 175,Training loss  0.0466, Validation loss 0.0557
Epoch 200,Training loss  0.0347, Validation loss 0.0467
Epoch 225,Training loss  0.0295, Validation loss 0.0455
Epoch 250,Training loss  0.0271, Validation loss 0.0464
```

```
with torch.no_grad():
    tM1 = lmodel(tF).squeeze() * tC0.std() + tC0.mean()
    tM2 = lmodel2(tF).squeeze() * tC0.std() + tC0.mean()
plt.figure(figsize=(3.75, 3.75))
plt.scatter(tF0, tC0 , marker = '*', c='k',s=70,label = 'data')
idx = torch.argsort(tF0) # sort tF0
plt.plot(tF0[idx], tM1[idx], label='Linear Relation')
plt.plot(tF0[idx], tM2[idx], label='MLP')
plt.ylabel('Temperature Celsius',fontsize = 12)
plt.xlabel('Temperature F',fontsize = 12)
plt.show()
```



Parameter Estimation 103

Finally: training

Of course, we have a bit of overfitting (too many parameters)

Careful: if you code this in a Jupyter notebook, do not run this cell twice w/o redefining the tensor. Weights are saved, so the model will just keep “improving” (or overfitting even more)

```
Lingering Question on

  plt.plot(tF0[idx], tM1[idx], label='Linear Relation')
  plt.plot(tF0[idx], tM2[idx], label='MLP')

- How can we plot tF0 vs tM[1-2] tensors without converting
  them explicitly to NumPy arrays?

  Matplotlib calls np.asarray() on its inputs. np.asarray doesn't
  know what a torch tensor is, but it follows a protocol: if the
  object has an __array__() method, call it and use what comes back.

  PyTorch tensors implement that method. Roughly:

    def __array__(self, dtype=None): # Simplified

      if dtype is None:
        return self.numpy()

      return self.numpy().astype(dtype, copy=False)

  .numpy() is where the actual guard lives:

    def numpy(self): # Simplified
      if self.requires_grad:
        raise RuntimeError(
          "Can't call numpy() on Tensor that requires grad. "
          "Use tensor.detach().numpy() instead."
        )

      if self.device.type != 'cpu':
        raise TypeError("Can't convert cuda tensor to numpy...")

      # otherwise: return a numpy array sharing memory
                   with the tensor's storage

Summary

plt.plot(t)
  → np.asarray(t)           # matplotlib calls this internally
  → t.__array__()           # numpy invokes the protocol method
  → t.numpy()               # PyTorch's implementation
  → checks requires_grad, device, dtype, then returns shared-memory array
```

Parameter Estimation 103

Once more on the PyTorch to Numpy conversion

tensor.numpy()

returns an array that

shares memory.

Modify one; the other reflects the change.

Parameter Estimation 104

```
We will explore the concept of batch size

-  Up to know we had batch_size = nsample
   Here lets assume ntrain >> 1 (something large)

    When batch_size == n_train, we update once per epoch with
    the full-dataset gradient (lowest noise possible — it's the
    true gradient over your training set).

- What if, let's say batch_size == 10 and ntrain == 2000?

    In this case, we update 200 times per epoch (2000 / 10 = 200),
    each using a 10-sample gradient with much higher noise.

So a simplified table would be

    batch_size   updates/epoch   gradient noise per update
    ----------   -------------   -------------------------
    2000 (full)  1               minimal (full gradient)
    200          10              low
    20           100             medium
    10           200             high
    1 (pure SGD) 2000            maximum
```

Batch size: don’t use the entire training set to compute the gradient

Use a subset, get a noisier estimation, do multiple updates

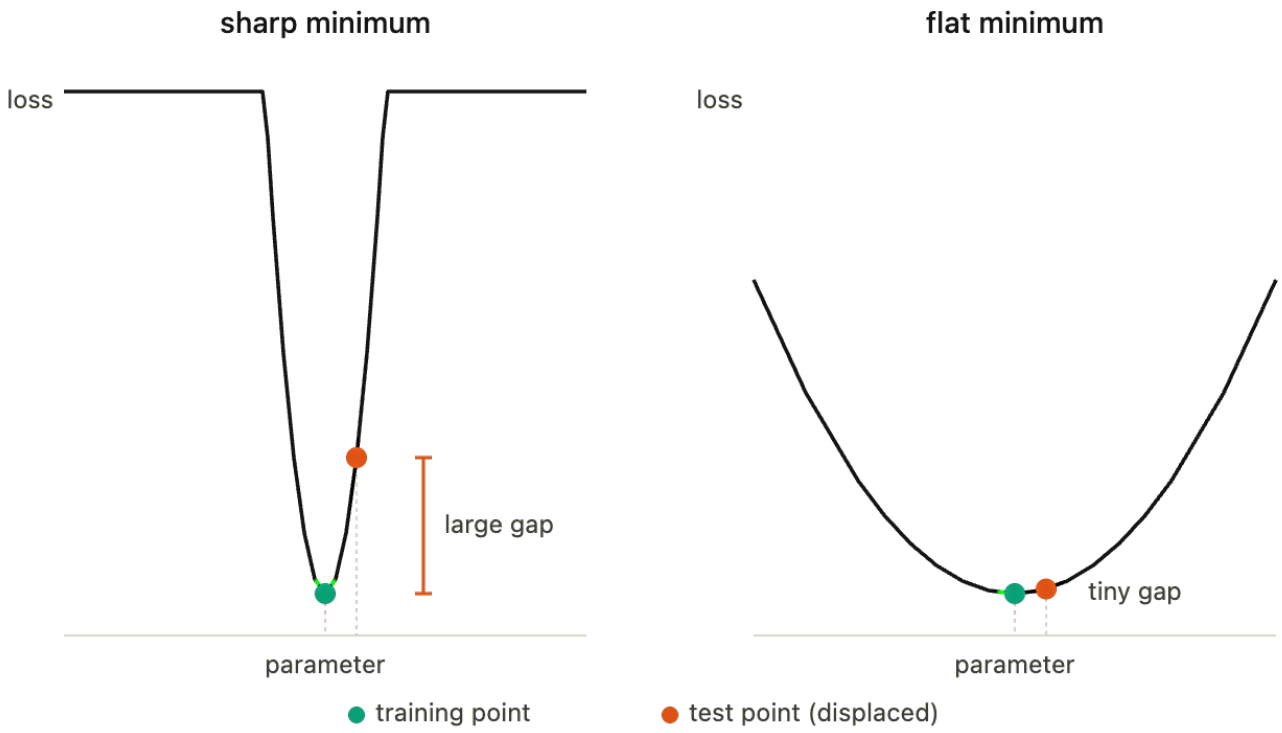
Parameter Estimation 104

- Why does it matter to use noisy estimation of the gradient?

  Curvature of the minimum: for higher generalization, we want to convergeto a shallower minimum. Too much curvature k in the minimum and there is a greater chance of overfitting.

- Why does generalization depend on the minimum curvature?

  Because real data is never exactly the same as training. Minimum with large curvature implies that if the data is a bit different from the training, the loss quickly increases.



```
- So, how to converge to a high curvature minimum?
  (which we don't wan't)

  There are two effects (the first one not related to ∇ noise)

(1) The lr ceiling (there is a maximum lr for convergence).

    Let's consider a quadratic loss
      Loss:     L = (1/2) k x^2
      Gradient: dL/dx = k x

    SGD update:
      x_{n+1} = x_n - lr * k * x_n = (1 - lr*k) * x_n

    So each step multiplies x by r = 1 - lr*k,
    giving x_n = r^n * x_0. Converges to 0 iff |r| < 1:

      |1 - lr*k| < 1 => 0 < lr*k < 2

    Regimes:
      lr*k = 0.5   r = +0.5   slow decay, no overshoot
      lr*k = 1.5   r = -0.5   damped oscillation
      lr*k = 2.0   r = -1.0   perpetual oscillation
      lr*k = 2.5   r = -1.5   diverges (amplitude grows)

    Geometric picture: step from x is -lr*k*x.
      lr*k = 2   step = -2x   flips to mirror point -x (same distance)
      lr*k > 2   step beyond -x, lands farther out than start

    In d dimensions with Hessian H, replace k with the
    largest eigenvalue: lr * lambda_max(H) < 2
```

Parameter Estimation 104

Flat vs sharp minimum

```
- So, how to converge to a high curvature minimum?
  (which we don't wan't)

  There are two effects

(2) Basin Volume. Before settling, SGD wanders through
    parameter space with a typical per-step displacement
    (a "kick size") of roughly

      lr * sigma / sqrt(batch_size)

    where sigma is the per-sample ∇ std: how much one sample's
    ∇ disagrees with the true (full-dataset) ∇ on average.

    Mini-batch noise comes from averaging B noisy samples, so the
    batch-gradient std is sigma / sqrt(B).

    This is also where the sqrt-batch-size scaling rule comes
    from (Hoffer et al. 2017): the kick size lr * sigma / sqrt(B)
    is what controls which basins SGD can reach.

    To preserve small-batch exploration behavior at larger B,
    the kick size must stay constant, which means
    lr must scale as sqrt(B).

    Implication for basin selection:
      - Sharp basin: geometrically narrow. Small target in param space
                     SGD's wandering trajectory rarely lands in it.

    Even if a sharp basin would be locally stable once reached,
    SGD rarely reaches it just because it's a small target.
```

Parameter Estimation 104

Flat vs sharp minimum

```python
def make_dataset(nsample, err, low=20.0, high=80.0):
  # sample F uniformly in [low, high)
  tF0 = torch.rand(nsample) * (high - low) + low
  # true F→C with Gaussian noise on C
  tC0 = (tF0 - 32) * 5/9 + torch.randn(nsample) * err
  # normalize and add feature dim
  tF = ((tF0 - tF0.mean()) / tF0.std()).unsqueeze(1)
  tC = ((tC0 - tC0.mean()) / tC0.std()).unsqueeze(1)

  return tF0, tC0, tF, tC
tF0, tC0, tF, tC = make_dataset(nsample=2000, err=2.5)
```

```python
# Train - validation split
nsamples = tC.shape[0]
nval = int(0.2*nsamples)
shuffled_indices = torch.randperm(nsamples)

train_indices = shuffled_indices[:-nval]
val_indices = shuffled_indices[-nval:]

tF_train = tF[train_indices]
tC_train = tC[train_indices]
tF_val = tF[val_indices]
tC_val = tC[val_indices]
```

```python
import torch.nn as nn
import torch.optim as optim

nh = 13     # dim hidden
ndense = 1  # number of dense layers

def dense_layer(n):
  block = nn.Sequential(
    nn.Linear(n, n),
    nn.Tanh(),
  )
  assert all(p.requires_grad for p in block.parameters())
  return block

lmodel2 = nn.Sequential(
  nn.Linear(1, nh),
  *[dense_layer(nh) for _ in range(ndense)],
  nn.Linear(nh, 1)
)

mse = nn.MSELoss()  # Constructor
```

Parameter Estimation 104

Define the temperature conversion data again (but now make a bit more general function).

Increased the number of samples to 2000

Same MLP model

Same Loss function

Parameter Estimation 104

Define the new training procedure, now computing the gradient in batches

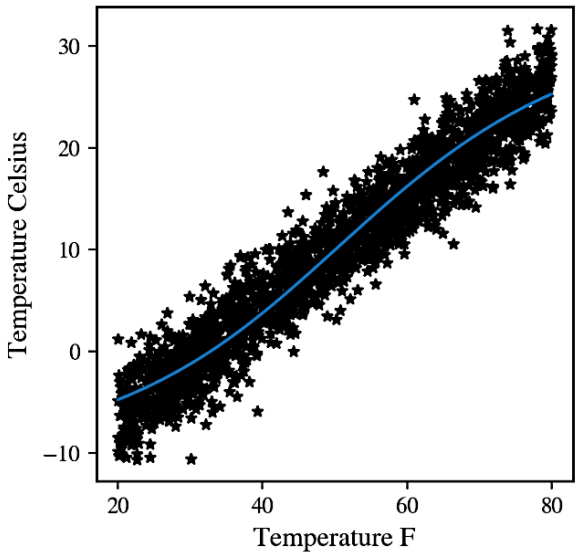
```
def training_loop_batched(nepochs, optimizer, bs):
  ntrain = tF_train.shape[0]
  for epoch in range(1, nepochs+1):
    perm = torch.randperm(ntrain)  # reshuffle each epoch
    for start in range(0, ntrain, bs):
      idx = perm[start:start+bs]   # bs = batch size
      tp = lmodel2(tF_train[idx])
      loss = mse(tp, tC_train[idx])

      optimizer.zero_grad(set_to_none=True)
      loss.backward()
      optimizer.step()
    if epoch == 1 or epoch % 2 == 0:
      with torch.no_grad():
        loss_val = mse(lmodel2(tF_val), tC_val).item()
      print(f"Epoch {epoch}, val loss {loss_val:.4f}")
```

```
training_loop_batched(nepochs = 25, bs=int(tF_train.shape[0]/10),
                      optimizer = optim.SGD(lmodel2.parameters(), lr=5e-3))
print()
```

```
Epoch 1, val loss 0.5719
Epoch 2, val loss 0.4252
Epoch 4, val loss 0.2261
Epoch 6, val loss 0.1272
Epoch 8, val loss 0.0876
Epoch 10, val loss 0.0744
Epoch 12, val loss 0.0708
Epoch 14, val loss 0.0700
Epoch 16, val loss 0.0700
Epoch 18, val loss 0.0702
Epoch 20, val loss 0.0703
Epoch 22, val loss 0.0702
Epoch 24, val loss 0.0703
```

```
with torch.no_grad():
    tM1 = lmodel2(tF).squeeze() * tC0.std() + tC0.mean()
plt.figure(figsize=(3.75, 3.75))
plt.scatter(tF0, tC0 , marker = '*', c='k',s=20,label = 'Data')
idx = torch.argsort(tF0) # sort tF0
plt.plot(tF0[idx], tM1[idx], label='MLP')
plt.ylabel('Temperature Celsius',fontsize = 12)
plt.xlabel('Temperature F',fontsize = 12)
plt.show()
```



Parameter Estimation 104

Train….

Noticed the reduced number of epochs

Careful: if you code this in a Jupyter notebook, do not run this cell twice w/o redefining the tensor. Weights are saved, so the model will just keep “improving” (or overfitting even more)

```
import torch.nn as nn
import torch.optim as optim
import copy
```

```
def make_dataset(nsample, err, low=20.0, high=80.0):
  tF0 = torch.rand(nsample) * (high - low) + low
  tC0 = (tF0 - 32) * 5/9 + torch.randn(nsample) * err
  tF = ((tF0 - tF0.mean()) / tF0.std()).unsqueeze(1)
  tC = ((tC0 - tC0.mean()) / tC0.std()).unsqueeze(1)
  return tF0, tC0, tF, tC
tF0, tC0, tF, tC = make_dataset(nsample=2000, err=2.5)
# Train - validation split
nsamples = tC.shape[0]
nval = int(0.2*nsamples);
shuffled_indices = torch.randperm(nsamples)
train_indices = shuffled_indices[:-nval]
val_indices = shuffled_indices[-nval:]
tF_train = tF[train_indices]; tC_train = tC[train_indices]
tF_val = tF[val_indices]; tC_val = tC[val_indices]
ntrain = tF_train.shape[0]
```

```
nh = 13; ndense = 1

def dense_layer(n):
  block = nn.Sequential(
    nn.Linear(n, n),
    nn.Tanh(),
  )
  assert all(p.requires_grad for p in block.parameters())
  return block

lmodel2 = nn.Sequential(nn.Linear(1, nh),
  *[dense_layer(nh) for _ in range(ndense)],
  nn.Linear(nh, 1))

mse = nn.MSELoss()  # Constructor
```

Parameter Estimation 105

How to loop over a hyperparameter (like batch size)?

Key is to reset the weights

Starting again from definitions

```
# ------------------------------------------------------------
# capture fresh weights ONCE, before any training has happened
# ------------------------------------------------------------
init_state = copy.deepcopy(lmodel2.state_dict())
```

```
def training_loop_batched(nepochs, optimizer, bs):
  ntrain = tF_train.shape[0]
  for epoch in range(1, nepochs+1):
    perm = torch.randperm(ntrain)  # reshuffle each epoch
    for start in range(0, ntrain, bs):
      idx = perm[start:start+bs]   # bs = batch size
      tp = lmodel2(tF_train[idx])
      loss = mse(tp, tC_train[idx])
      optimizer.zero_grad(set_to_none=True)
      loss.backward()
      optimizer.step()
    if epoch == 1 or epoch % 2 == 0:
      with torch.no_grad():
        loss_val = mse(lmodel2(tF_val), tC_val).item()
```

```
- Requirement: adjust the nepochs depending on batch size

    bs = ntrain / x, so x counts how many mini-batches make up
    ne epoch. Small x = large batches, large x = small batches.

    With fixed nepochs, large batches get far fewer gradient
    updates than small ones. Large batches can then look bad
    simply because they are undertrained.

    To prevent undertraining, we hold total updates fixed.

      total updates ~ nepochs * (ntrain/bs) = nepochs * x

    Holding this constant at

      nepochs_base * x_base = 40 * 10 = 400

    gives nepochs = 400/x.
```

Parameter Estimation 105

How to loop over a hyperparameter (like batch size)?

We must reset the weights btw runs

Fair comparison requires adjusting the number of epochs

```
nepochs_base = 30
x_base = 10

idx_sort  = torch.argsort(tF0)
plt.figure(figsize=(5, 5))
plt.scatter(tF0, tC0, marker='*', c='k', s=20, label='data')

for x in [1, 2, 5, 10, 50, 100, 200]:
  bs = max(1, ntrain // x)
  lmodel2.load_state_dict(init_state)           # reset weights
  opt = optim.SGD(lmodel2.parameters(), lr=5e-3) # fresh optimizer

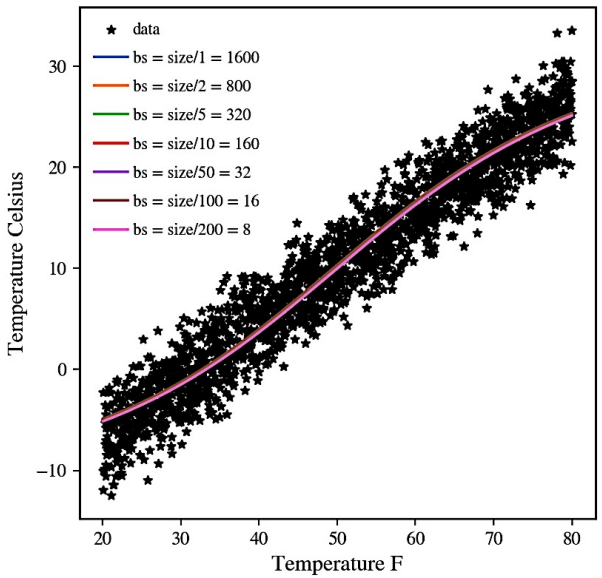
  # need to adjust the number of epochs
  nepochs = max(1, round(nepochs_base * x_base / x))

  training_loop_batched(nepochs=nepochs, optimizer=opt, bs=bs)

  with torch.no_grad():
    tM = lmodel2(tF).squeeze() * tC0.std() + tC0.mean()
    loss_val = mse(lmodel2(tF_val), tC_val).item()
    print(f"Batch Size {bs}, val loss {loss_val:.4f}")
  plt.plot(tF0[idx_sort], tM[idx_sort], label=f'bs = size/{x} = {bs}')


plt.xlabel('Temperature F', fontsize=12)
plt.ylabel('Temperature Celsius', fontsize=12)
plt.legend(handletextpad=0.3, fontsize=9, frameon=False)
plt.show()
```

```
Batch Size 1600, val loss 0.0722
Batch Size 800, val loss 0.0722
Batch Size 320, val loss 0.0722
Batch Size 160, val loss 0.0722
Batch Size 32, val loss 0.0724
Batch Size 16, val loss 0.0743
Batch Size 8, val loss 0.0712
```



Parameter Estimation 105

How to loop over a hyperparameter (like batch size)?

We must reset the weights btw runs

Fair comparison requires adjusting the number of epochs

Parameter Estimation 105

How to loop over a hyperparameter (like batch size)?

What happens when nepochs is fixed (and too low for high batch size)?

High batch size is undertrained

- What happens when nepochs is fixed (and too low for high batch size)?

```
idx_sort  = torch.argsort(tF0)
plt.figure(figsize=(5, 5))
plt.scatter(tF0, tC0, marker='*', c='k', s=20, label='data')

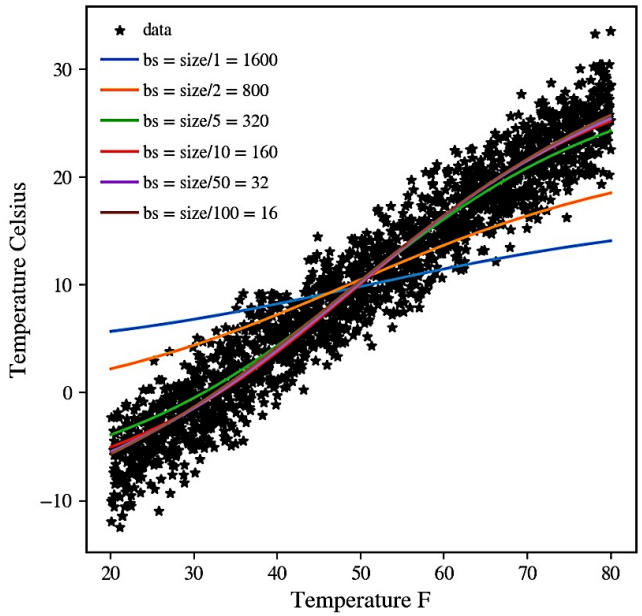
for x in [1, 2, 5, 10, 50, 100]:
  bs = max(1, ntrain // x)
  lmodel2.load_state_dict(init_state)            # reset weights
  opt = optim.SGD(lmodel2.parameters(), lr=5e-3) # fresh optimizer

  nepochs = 30

  training_loop_batched(nepochs=nepochs, optimizer=opt, bs=bs)
  with torch.no_grad():
    tM = lmodel2(tF).squeeze() * tC0.std() + tC0.mean()
    loss_val = mse(lmodel2(tF_val), tC_val).item()
    print(f"Batch Size {bs}, val loss {loss_val:.4f}")
  plt.plot(tF0[idx_sort], tM[idx_sort], label=f'bs = size/{x} = {bs}')

plt.xlabel('Temperature F', fontsize=12)
plt.ylabel('Temperature Celsius', fontsize=12)
plt.legend(handletextpad=0.3, fontsize=9, frameon=False)
plt.show()
```

```
Batch Size 1600, val loss 0.5568
Batch Size 800, val loss 0.2724
Batch Size 320, val loss 0.0762
Batch Size 160, val loss 0.0722
Batch Size 32, val loss 0.0717
Batch Size 16, val loss 0.0726
```



```
lr_base       = 5e-3
nepochs_base  = 50 # increased the number of epochs
                   # if nepoch is too low, higher lr will
                   # not converge
x_base        = 10
bs_base       = ntrain // x_base

idx_sort = torch.argsort(tF0)
plt.figure(figsize=(5, 5))
plt.scatter(tF0, tC0, marker='*', c='k', s=20, label='data')

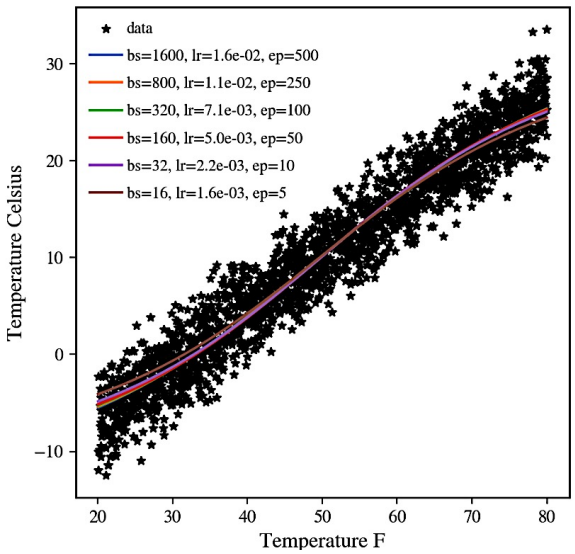
for x in [1, 2, 5, 10, 50, 100]:
  bs  = max(1, ntrain // x)
  nepochs = max(1, round(nepochs_base * x_base / x))
  lr  = lr_base * (bs / bs_base) ** 0.5 # sqrt-batch-size rule

  lmodel2.load_state_dict(init_state)          # reset weights
  opt = optim.SGD(lmodel2.parameters(), lr=lr) # fresh optimizer
  training_loop_batched(nepochs=nepochs, optimizer=opt, bs=bs)

  with torch.no_grad():
    tM = lmodel2(tF).squeeze() * tC0.std() + tC0.mean()
    loss_val = mse(lmodel2(tF_val), tC_val).item()
    print(f"Batch Size {bs}, val loss {loss_val:.4f}")
  plt.plot(tF0[idx_sort], tM[idx_sort],
           label=f'bs={bs}, lr={lr:.1e}, ep={nepochs}')

plt.legend(handletextpad=0.3, fontsize=9,
           frameon=False, loc='upper left')
plt.xlabel('Temperature F', fontsize=12)
plt.ylabel('Temperature Celsius', fontsize=12)
plt.show()
```

```
Batch Size 1600, val loss 0.0711
Batch Size 800, val loss 0.0715
Batch Size 320, val loss 0.0717
Batch Size 160, val loss 0.0719
Batch Size 32, val loss 0.0722
Batch Size 16, val loss 0.0748
```



Parameter Estimation 105

How to loop over a hyperparameter (like batch size)?

Here, we adjust both the number of epochs and the learning rate

```
-   Challenge: loop over batch size (adjusting the
    number  of epochs) and the dense layer width

    For each dense layer width (nh), plot only the
    best batch size case (what is a good metric to
    to determine best batch size? average over
    last 5 epochs? min over all epochs?)

    We are going to drastically reduce training size,
    increase errors, and increase the ML internal dim.
    Why? So the task is overfittable

- One error we had before is data leakage (fixed now)

    Normalization of the data should involve
    ONLY training so there is no validation data leakage

- Why averaging over SEEDS (which we will do)

    What varies across seeds: model init + mini-batch order.

    The dataset and train/val split are computed once with
    fixed SEED = 0. This isolates the training-noise effects

- Why training noise matters

    The loss surface of a neural network is non-convex:
    many local minima of comparable depth coexist:
    training stochastic even on a fixed dataset.

    Init lottery: different starting points run
    downhill into different basins.

    Mini-batch lottery: wach mini-batch gradient is a
    noisy estimate of the true gradient. The order in
    which batches are seen determines the exact path
    through parameter space.

    A single run is one realization of these two lotteries.
    The seed band measures how much val loss varies just
    from re-rolling them, with no change in data or
    hyperparameters.

    If small-batch effects were smaller than the seed
    band, we couldn't claim it was real.
```

Parameter Estimation 106

Loop over batch size and the dense layer width

For each layer width, plot only the best batch size case (best to which metric?)

```
import torch.nn as nn
import torch.optim as optim
import copy
```

```
def make_dataset(nsample, err, low=20.0, high=80.0):
  tF0 = torch.rand(nsample) * (high - low) + low
  tC0 = (tF0 - 32) * 5/9 + torch.randn(nsample) * err
  return tF0, tC0   # raw only
tF0, tC0 = make_dataset(nsample=2000, err=5)

# Split FIRST
nsamples = tC0.shape[0]
nval = int(0.97 * nsamples)
shuffled_indices = torch.randperm(nsamples)
train_indices = shuffled_indices[:-nval]
val_indices   = shuffled_indices[-nval:]

# Normalization stats from TRAIN ONLY
tF_mean = tF0[train_indices].mean()
tF_std  = tF0[train_indices].std()
tC_mean = tC0[train_indices].mean()
tC_std  = tC0[train_indices].std()

# Apply the SAME train-derived stats to both splits
tF = ((tF0 - tF_mean) / tF_std).unsqueeze(1)
tC = ((tC0 - tC_mean) / tC_std).unsqueeze(1)

tF_train = tF[train_indices];
tC_train = tC[train_indices]
tF_val   = tF[val_indices];
tC_val   = tC[val_indices]
ntrain   = tF_train.shape[0]
```

Parameter Estimation 106

Fix a problem on the previous examples (dataset leak)

```
def dense_layer(n):
  block = nn.Sequential(nn.Linear(n, n), nn.Tanh())
  return block

mse = nn.MSELoss()  # Constructor

def training_loop_batched(nepochs, optimizer, model, bs):
  ntrain = tF_train.shape[0]
  val_losses = []
  for epoch in range(1, nepochs+1):
    perm = torch.randperm(ntrain)
    for start in range(0, ntrain, bs):
      idx  = perm[start:start+bs]
      tp   = model(tF_train[idx])
      loss = mse(tp, tC_train[idx])
      optimizer.zero_grad(set_to_none=True)
      loss.backward()
      optimizer.step()
    with torch.no_grad():
      val_losses.append(mse(model(tF_val), tC_val).item())
  return val_losses

nepochs_base = 60
x_base  = 10
bs_base = ntrain // x_base
lr_base  = 5e-3
nh_base = 30
```

Parameter Estimation 106

Loop over batch size (adjusting number of epochs) and the dense layer width

Here we keep the losses from all epochs

```
ndense = 1
n_seeds = 10
idx_sort = torch.argsort(tF0)

# Accumulator: all_losses[nh][bs] = {'mean': [seed0, seed1, ...],
#                                    'min': [...], 'gap': [...]}
all_losses = {}

for seed in range(n_seeds):
  torch.manual_seed(seed)
  for nh in range(nh_base, nh_base+12+1, 3):
    # fresh model + fresh init for this width (varies with seed)
    lmodel2 = nn.Sequential(nn.Linear(1, nh),
      *[dense_layer(nh) for _ in range(ndense)], nn.Linear(nh, 1),
    )
    init_state = copy.deepcopy(lmodel2.state_dict())

    if seed == 0:
      all_losses[nh] = {}

    for x in [2, 5, 10, 50]:
      bs      = max(1, ntrain // x)
      nepochs = max(1, round(nepochs_base * x_base / x))
      lr      = lr_base

      lmodel2.load_state_dict(init_state)
      opt = optim.SGD(lmodel2.parameters(), lr=lr)
      val_losses = training_loop_batched(nepochs=nepochs,
                                         optimizer=opt,
                                         model=lmodel2,
                                         bs=bs)
      tail    = val_losses[-5:]
      avg_val = sum(tail) / len(tail)
      min_val = min(val_losses)
      gap     = avg_val - min_val

      # accumulate this seed's result
      if bs not in all_losses[nh]:
        all_losses[nh][bs] = {'mean': [], 'min': [], 'gap': []}
      all_losses[nh][bs]['mean'].append(avg_val)
      all_losses[nh][bs]['min'].append(min_val)
      all_losses[nh][bs]['gap'].append(gap)
```

Parameter Estimation 106

With no adjustable LR.

What is the best model?

Average of the last 5 epochs?

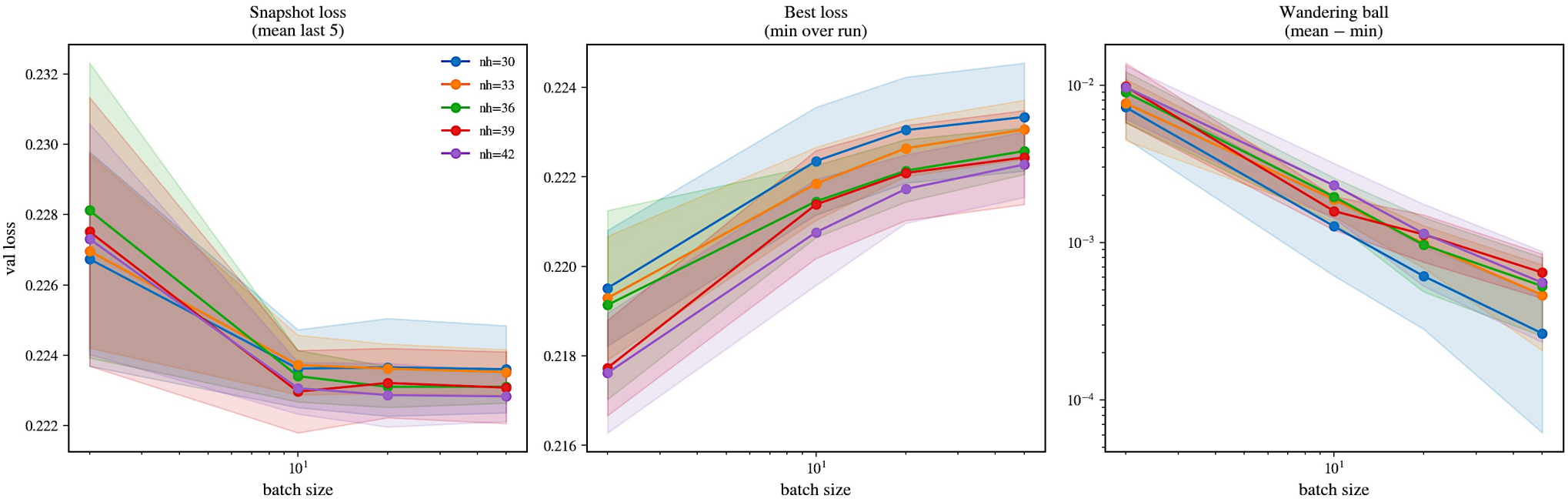
Wouldn’t that punish small batches?

To answer that, we average over 10 seeds

PE 106: What is the best model?

Average of the last five epochs? Val on last epoch?

That punishes small batches (noise)




```
- What the wandering ball plot measured?

  1. Taylor expand the loss at the minimum. Shift coordinates so x_min = 0:

    L(x) = L_min + L'(0) x + (1/2) L''(0) x² + O(x³)
         = L_min + (k/2) x²

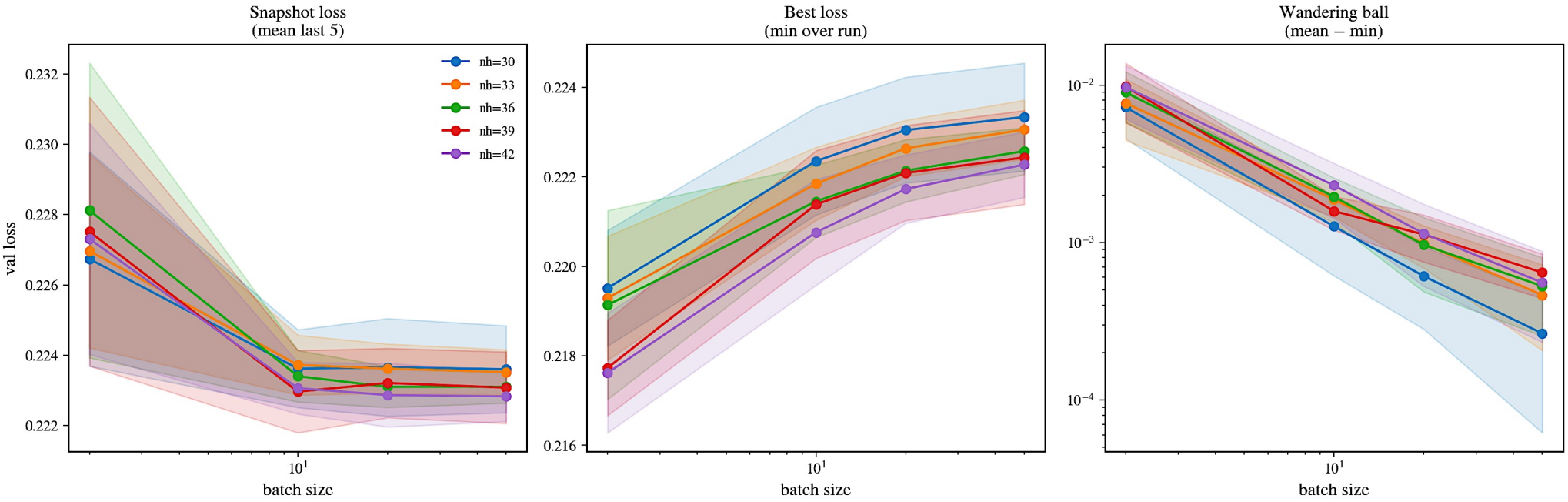
    E[L(x)] = L_min + (k/2) E[x²] (and E[x²] = Var[x] around the minimum)

    Therefore E[L - L_min] = (k/2) Var[x]
  Now
    Var[x]         = lr·σ² / (2 k B)
  So
    E[L - L_min] = lr·σ² / (4 B) (the third panel)
```

PE 106: What is the best model?

Average of the last five epochs? Val on last epoch?

That punishes small batches (noise)



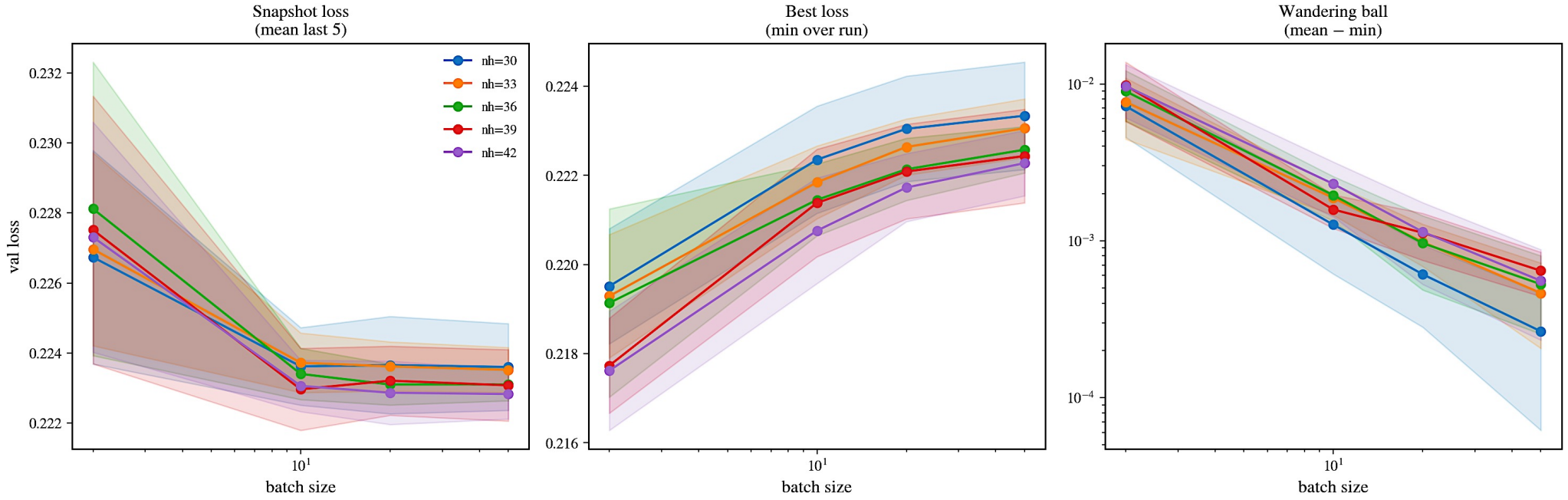

```
-  Lesson: don't just check val at the last epoch

  Panel 3 shows the gap between snapshot loss (mean of last 5)
  and basin floor (min over run). For small batch this gap reaches ~0.01.

  Once in a basin, the optimizer orbits the floor — it doesn't sit there.
  A last-epoch val is a random point on that orbit, not the basin depth.

  Evaluate val every epoch and keep the lowest (**best-on-val checkpointing**).
```

PE 106: What is the best model?




```
The 1/B slope on the wandering ball panel comes from holding lr fixed
If instead lr scales with B, the slope changes:

  scaling            gap proportional to    gap-panel slope
  ---------------    -------------------    ---------------
  lr = const         1/B                    -1
  lr ~ sqrt(B)       1/sqrt(B)              -0.5    (Hoffer)
  lr ~ B             constant                0      (Goyal)

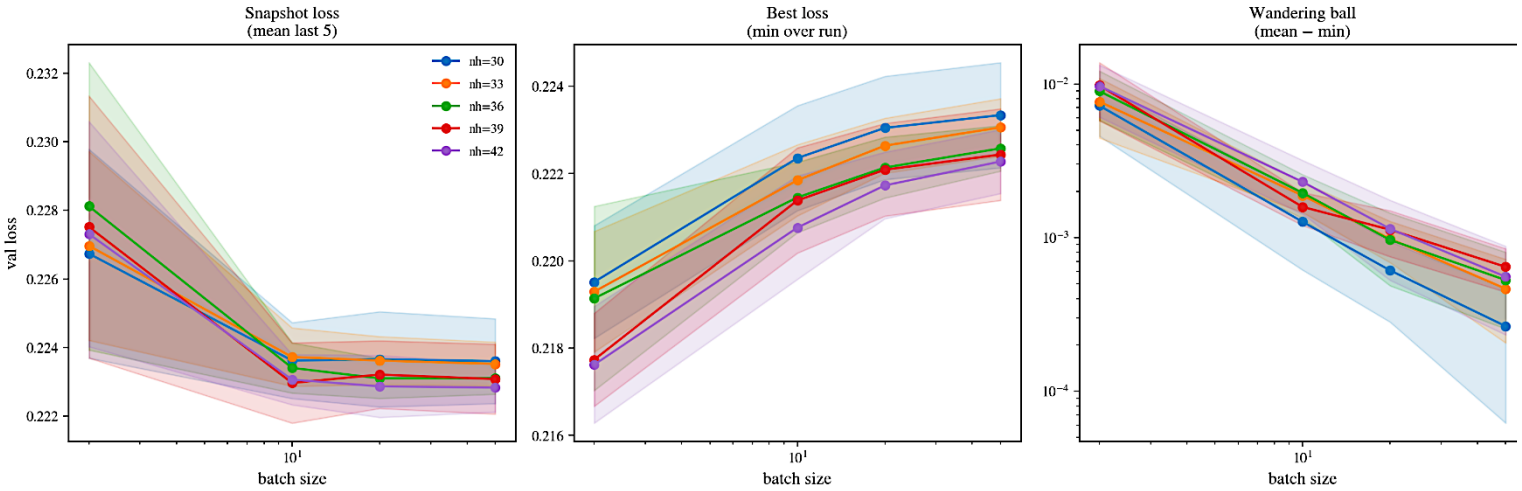
Linear scaling (lr ~ B) keeps the SDE temperature lr/B constant, so
the wandering ball is preserved by construction — a flat line on the
gap panel.

Caveat: linear scaling hits the lr ceiling (lr * lambda_max(H) < 2)
at modest B and diverges. In practice it needs warmup or gradient
clipping. Sqrt scaling trades some preservation for stability at
larger B.
```

PS: The python code to plot the shades

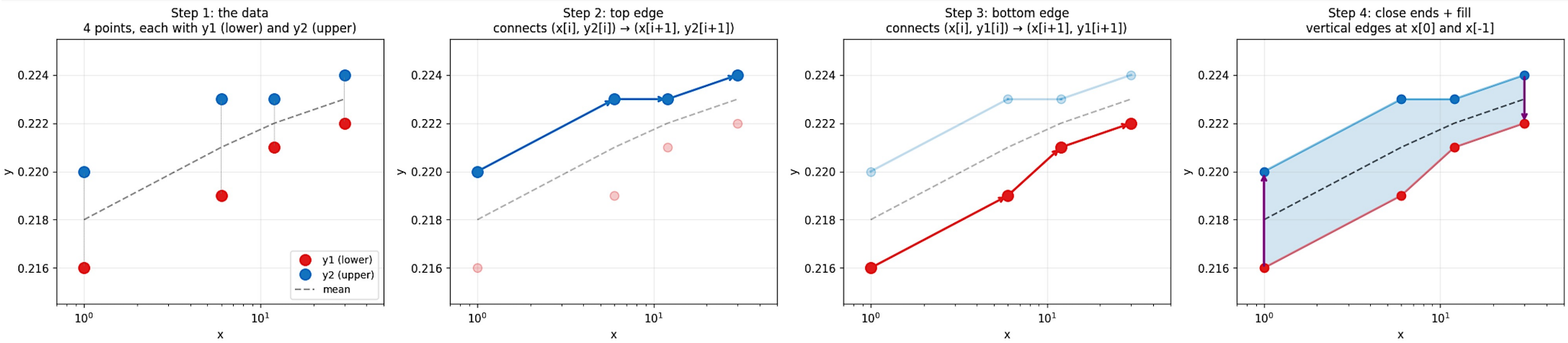
```python
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
metrics = ['mean', 'min', 'gap']
titles  = ['Snapshot loss\n(mean last 5)',
           'Best loss\n(min over run)',
           'Wandering ball\n(mean - min)']

for ax, metric, title in zip(axes, metrics, titles):
  for nh, bs_dict in all_losses.items():
    # (fill_between needs monotonic x).
    bss = sorted(bs_dict.keys())
    arr = np.array([bs_dict[b][metric] for b in bss])  # (n_bs, n_seeds)
    mu, sd = arr.mean(axis=1), arr.std(axis=1)
    line, = ax.plot(bss, mu, marker='o', label=f'nh={nh}')
    ax.fill_between(bss, mu - sd, mu + sd, alpha=0.2, color=line.get_color())
  ax.set_xscale('log')
  ax.set_xlabel('batch size', fontsize=12)
  ax.set_title(title, fontsize=12)
axes[0].set_ylabel('val loss', fontsize=12)
axes[2].set_yscale('log')
axes[0].legend(frameon=False, fontsize=9)
plt.tight_layout()
plt.show()
```



If you don't understand why we need to make the dictionary keys sorted, see the next slide

PS: The python code to plot the shades




```
PS: why the line bss = sorted(bs_dict.keys()) ?

How fill_between builds the polygon

-  Step 1. Input: four x-positions, each with a lower bound (y1 = mu - sigma)
and an upper bound (y2 = mu + sigma).

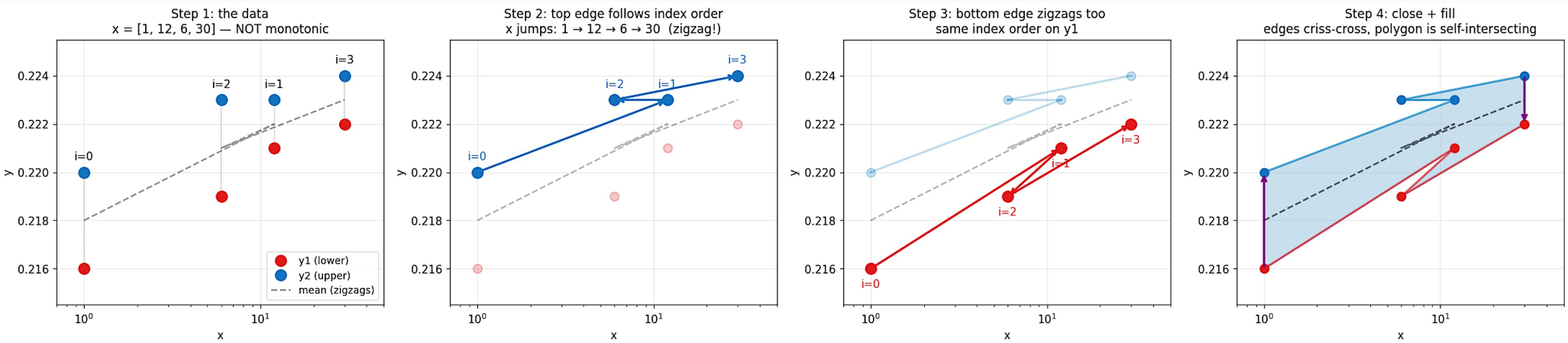
-  Step 2. Top edge: connect (x[i], y2[i]) to (x[i+1], y2[i+1]) for
i = 0, 1, 2. Three line segments running left-to-right across the top.

-  Step 3. Bottom edge: same on y1. Three segments, same left-to-right
order.

-  Step 4. Close the polygon with two vertical segments: one at the
leftmost x from y1[0] up to y2[0], one at the rightmost x from y2[-1]
down to y1[-1]. Now it's a closed shape — matplotlib fills the interior.

Why monotonic x matters. In Steps 2 and 3, consecutive segments share
an x-direction. As long as every step goes rightward (ascending) or
every step goes leftward (descending), the top edge sweeps cleanly
across without doubling back. The bottom does the same. Two clean
sweeps plus two vertical caps = a simple, non-self-intersecting polygon.
```

PS: The python code to plot the shades



PS: why the line bss = sorted(bs_dict.keys()) ?

How fill_between builds the polygon

- Step 1. Input: four x-positions, each with a lower bound (y1 = mu - sigma) and an upper bound (y2 = mu + sigma).

- Step 2. Top edge: connect (x[i], y2[i]) to (x[i+1], y2[i+1]) for i = 0, 1, 2. Three line segments running left-to-right across the top.

- Step 3. Bottom edge: same on y1. Three segments, same left-to-right order.

- Step 4. Close the polygon with two vertical segments: one at the leftmost x from y1[0] up to y2[0], one at the rightmost x from y2[-1] down to y1[-1]. Now it's a closed shape — matplotlib fills the interior.

Why monotonic x matters. In Steps 2 and 3, consecutive segments share an x-direction. As long as every step goes rightward (ascending) or every step goes leftward (descending), the top edge sweeps cleanly across without doubling back. The bottom does the same. Two clean sweeps plus two vertical caps = a simple, non-self-intersecting polygon.

```python
# full-batch loss on the resident Part VI train tensors
def full_loss(model):
  return mse(model(tF_train), tC_train)

torch.manual_seed(0)
hmodel = nn.Sequential(nn.Linear(1, nh_base),
  *[dense_layer(nh_base) for _ in range(ndense)],
  nn.Linear(nh_base, 1)).to(device)

# A minimum is where the gradient vanishes.
# Run full-batch GD until |grad| has fallen to grad_drop
# of its starting value
grad_drop = 1e-3   # stop when |grad| is 1/1000 of start
max_steps = 20000  # upper bound if grad_drop  never met

opt = optim.SGD(hmodel.parameters(), lr=5e-3)

g0 = None
for step in range(max_steps):
  loss = full_loss(hmodel)
  # backward adds into .grad, so clear it each step
  opt.zero_grad(set_to_none=True)
  loss.backward()

  # global gradient norm: stack every parameter's grad
  # into one long vec, then take its length. p.grad has
  # the same shape as p; reshape(-1) flattens each to 1D;
  # torch.cat joins them; .norm() is the L2 length.
  # This single number -> 0 as we reach the minimum.
  gn = torch.cat([p.grad.reshape(-1)
                  for p in hmodel.parameters()]).norm()
  if g0 is None: g0 = gn

  # apply the optimizer's update using the grads in
  # each p.grad (just filled by loss.backward()).
  # For SGD this is p <- p - lr * p.grad, which is
  # the actual step downhill.
  if gn < grad_drop * g0:
    break

  opt.step() # not converged -> take the step

minpar = [p.detach().clone() for p in hmodel.parameters()]
```

Parameter Estimation 107

Can we visualize how the maximum learning rate depends on the minimum curvature?

```
Measure minimum curvature
loop:
  g = ∇f(x)
  if ‖g‖ < ε:  stop      # we've arrived — x is the answer
  x ← x − lr·g           # otherwise, take a step
```

```python
# we can't afford to build that whole Hessian
# matrix, H, but we don't need it. We only want
# H applied to a vec (here a flat vector.
def hvp(model, vec):
  # Returns H@vec. vec is a flat vector (i.e.,
  # all params stacked into one 1D-tensor).
  params = list(model.parameters())

  # First backward -> the gradient g = dL/dtheta.
  # Keep its graph (create_graph=True) so we can
  # take a derivative OF g in step 3. grads comes
  # back shaped like the params, so we flatten it
  grads = torch.autograd.grad(full_loss(model), params,
                              create_graph=True)
  g = torch.cat([gi.reshape(-1) for gi in grads])

  # Step 2: dot product of g . vec = a scalar
  gv = (g * vec).sum()

  # Step 3: second backward. d(g . vec)/dtheta is H@vec,
  # Why? d/dtheta(dL/dtheta . vec) = (d2L/dtheta2)vec.
  Hg = torch.autograd.grad(gv, params)
  return torch.cat([hi.reshape(-1) for hi in Hg])
```

```
WHY torch.cat / reshape (it is NOT the matrix multiply)

autograd.grad already computed H @ vec. It just returns
it the way params are stored: a LIST of layer tensors
in their natural shapes, not a single vector:

  W0(30,1) b0(30) W1(30,30) b1(30) W2(1,30) b2(1)

But power iteration (norms, dot products) needs one
flat vector of length d = total scalar parameters.

reshape(-1) flattens each piece to 1D:

  30 30 900 30 30 1

torch.cat glues them end to end
```

Parameter Estimation 107

Can we visualize how the maximum learning rate depends on the minimum curvature?

```
Measure minimum curvature
loop:
  g = ∇f(x)
  if ‖g‖ < ε:  stop      # we've arrived — x is the answer
  x ← x − lr·g           # otherwise, take a step
```

Parameter Estimation 107

Can we visualize how the maximum learning rate depends on the minimum curvature?

```python
def lambda_max(model, iters=50):
  # power iteration: v <- (H v)/||H v|| tends to
  # the top eigenvector; Rayleigh quotient v.(H v)
  # is its eigenvalue
  d = sum(p.numel() for p in model.parameters())
  v = torch.randn(d, device=device); v /= v.norm()
  lam = 0.0
  for _ in range(iters):
    Hv = hvp(model, v)
    lam = torch.dot(v, Hv).item()
    v = Hv / (Hv.norm() + 1e-12)
  return lam

# restore minpar and measure curvature at the minimum.
# minpar is not flattened: it holds one clone per
# parameter, each in its natural shape (W0 (30,1),
# b0 (30,), ...). zip pairs each live param p with its
# saved clone ps in the same order, so their shapes
# already match; copy_ writes ps back into p in place,
# no reshape needed.
with torch.no_grad():
  for p, ps in zip(hmodel.parameters(), minpar):
    p.copy_(ps)
lmax = lambda_max(hmodel)
lr_crit = 2.0 / lmax
print(f"lr_crit = 2/lambda_max = {lr_crit:.4f}")
```

```
lr_crit = 2/lambda_max = 0.1302
```

```
Measure minimum curvature
loop:
  g = ∇f(x)
  if ‖g‖ < ε:  stop        # we've arrived — x is the answer
  x ← x − lr·g             # otherwise, take a step
```

```python
import numpy as np
import matplotlib.pyplot as plt

# ONE fixed perturbation off the minimum, reused for EVERY
# lr so the only variable is lr. Then run full-batch GD and
# see if the loss decays (stable) or explodes (diverges).
torch.manual_seed(0)
kick = [0.01 * torch.randn_like(ps) for ps in minpar]
tk = list(zip(minpar, kick))

def runaway(model, lr, steps=60):
  # Overwrite the weights in place with min + the fixed
  # kick. The parameters are leaf tensors requiring grad,
  # and copy_ is an in-place edit -- autograd forbids that
  # on such tensors and would error. torch.no_grad() turns
  # grad tracking off, so we can set the weights directly.
  with torch.no_grad():
    for p, (ps, k) in zip(model.parameters(), tk):
      p.copy_(ps + k) # reset to min + fixed kick

  # fresh optimizer at this lr (one lr per call,
  # no leftover momentum/state)
  opt = optim.SGD(model.parameters(), lr=lr)
  l0 = full_loss(model).item() # loss right after the kick
  # run full-batch GD at this lr
  for _ in range(steps):
    loss = full_loss(model)
    opt.zero_grad(set_to_none=True)
    loss.backward(); opt.step()
  # ratio of final to post-kick loss: >1 grew (diverged),
  # <1 decayed (stable)
  r = full_loss(model).item() / l0
  # an unstable lr overflows the loss to inf/nan; clamp
  # to a large finite value so the plot stays readable
  return min(r, 1e12) if np.isfinite(r) else 1e12
```

Parameter Estimation 107

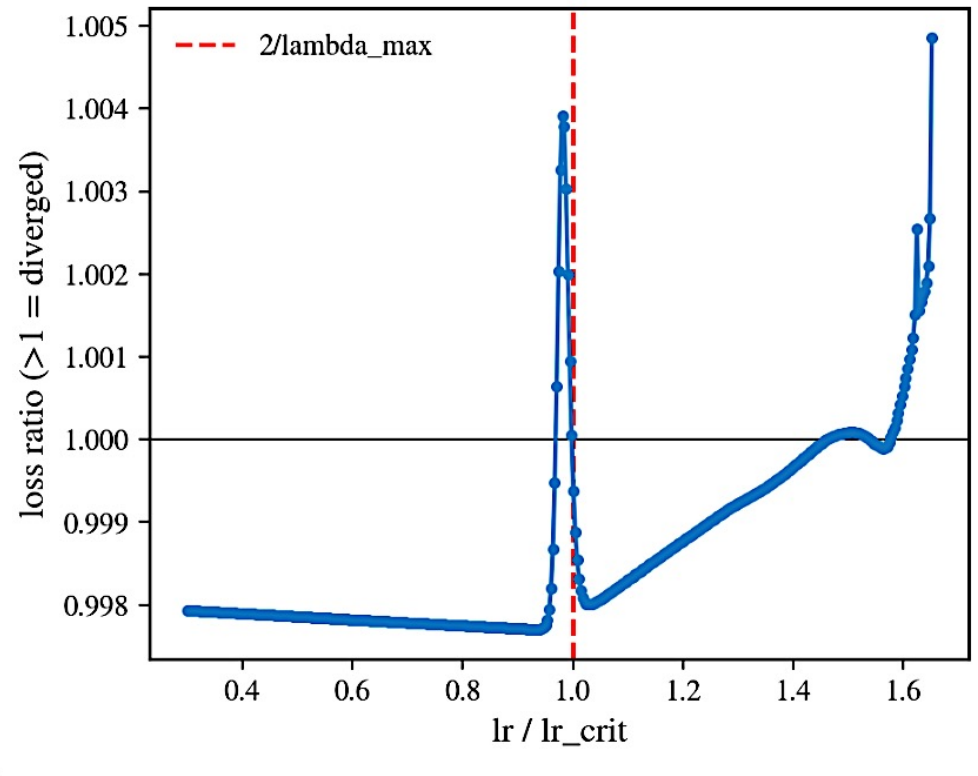
Can we visualize how the maximum learning rate depends on the minimum curvature?

```
Measure minimum curvature
loop:
  g = ∇f(x)
  if ‖g‖ < ε:  stop      # we've arrived — x is the answer
  x ← x − lr·g           # otherwise, take a step
```

Parameter Estimation 107

```
lrs = lr_crit * np.linspace(0.3, 1.6511, 400)
ratio = [runaway(hmodel, float(lr)) for lr in lrs]

plt.figure(figsize=(5, 4))
plt.axhline(1.0, color='k', lw=0.7)
plt.axvline(1.0,color='r',ls='--',label='2/lambda_max')
plt.plot(lrs / lr_crit, ratio, 'o-', ms=3)
plt.xlabel('lr / lr_crit')
plt.ylabel('loss ratio (>1 = diverged)')
plt.legend(frameon=False); plt.show()
```




```
lr_crit = 2/lambda_max is the linear stability limit:
above it the sharpest direction is unstable in the
quadratic picture. But a nonlinear loss does not just
blow up there. Four features:

x < 1: stable; the loss decays, fastest near x = 0.9
(the optimal lr, just under the edge).

x ~ 1: a sharp, narrow spike sits on the line -- the
sharpest mode going marginal. The loss briefly pokes
above 1, then settles. This pins the threshold at
2/lambda_max.

1 < x < 1.58: the mode is now linearly unstable, but as
the params move that direction flattens.

x > 1.58: even the flattened curvature cannot hold
lr*lambda < 2; the loss truly blows up.

other refs:
(1) arXiv:2103.00065
(2) eregis.github.io/blog/2025/09/08/edge-of-stability.html
(3) https://arxiv.org/abs/2209.15594
(4) https://arxiv.org/abs/2406.09405
(5) https://arxiv.org/pdf/2410.24206

Ref 5 seems quite amazing
```

```
  SWA (Izmailov et al. 2018, 1803.05407): average
  parameter states over the last half of training
  to land at the center of the wandering ball instead
  of a random orbit sample.
```

```python
def training_loop_batched(nepochs, optimizer, model, bs):
  ntrain = tF_train.shape[0]
  val_losses = []

  swa_start = (3 * nepochs) // 4
  swas = []

  for epoch in range(1, nepochs+1):
    if epoch == swa_start + 1:
      for g in optimizer.param_groups:
        g['lr'] *= 0.5   # contract the orbit during averaging
    perm = torch.randperm(ntrain)
    for start in range(0, ntrain, bs):
      idx  = perm[start:start+bs]
      tp   = model(tF_train[idx])
      loss = mse(tp, tC_train[idx])
      optimizer.zero_grad(set_to_none=True)
      loss.backward()
      optimizer.step()
    with torch.no_grad():
      val_losses.append(mse(model(tF_val), tC_val).item())
    if epoch > swa_start:
      # model.state_dict() #  {'0.weight': tensor, ...} =
      #                         reference to live tensors
      # .items()             # iterate (key, value) pairs
      # {k: v.clone() ...}   # deep copy frozen at this epoch
      snapshot = {k: v.clone() for k, v in model.state_dict().items()}
      swas.append(snapshot)

  # Average parameter states across the SWA window
  # swas = swa states
  n = len(swas)
  swa_state = {k: sum(s[k] for s in swas) / n for k in swas[0]}

  # Evaluate the averaged model
  model.load_state_dict(swa_state)
  with torch.no_grad():
    # .item() converts a 0-dim PyTorch tensor to a Python float
    swa_val = mse(model(tF_val), tC_val).item()
  return val_losses, swa_state, swa_val

# SWA NEEDS MORE EPOCHS (especially at low bs)
# Why? Because it is an average over nepochs
nepochs_base = 150
x_base  = 10
bs_base = ntrain // x_base
lr_base  = 5e-3
nh_base = 30
```

Parameter Estimation 108

Second point: generalization

This depends on the curvature of the minimum

SWA methods punish sharp minima (if orbits around minima are symmetric and a clean quadratic basin)

```python
    if epoch > swa_start:
        # model.state_dict() #  {'0.weight': tensor, ...} =
        #                         reference to live tensors
        # .items()              # iterate (key, value) pairs
        # {k: v.clone() ...}    # deep copy frozen at this epoch
        snapshot = {k: v.clone() for k, v in model.state_dict().items()}
        swas.append(snapshot)

# Average parameter states across the SWA window
# swas = swa states
n = len(swas)
swa_state = {k: sum(s[k] for s in swas) / n for k in swas[0]}
```

```
Explanation of the line

    n = len(swas)
    swas = {k: sum(s[k] for s in swas) / n for k in swas[0]}

  swas is a list of dicts, one per epoch:
    [{'0.weight': tensor, '0.bias': tensor, ...},   # epoch 31
     {'0.weight': tensor, '0.bias': tensor, ...},   # epoch 32
     ...]

  Outer comprehension {... for k in swas[0]}: iterate over
  parameter names (the keys), build one entry per name
  in the output dict.

  Inner sum(s[k] for s in swa_states): for parameter k,
  walk every snapshot and add its tensor. PyTorch's + does
  element-wise addition, so the result is one tensor of
  the same shape with per-element sums.

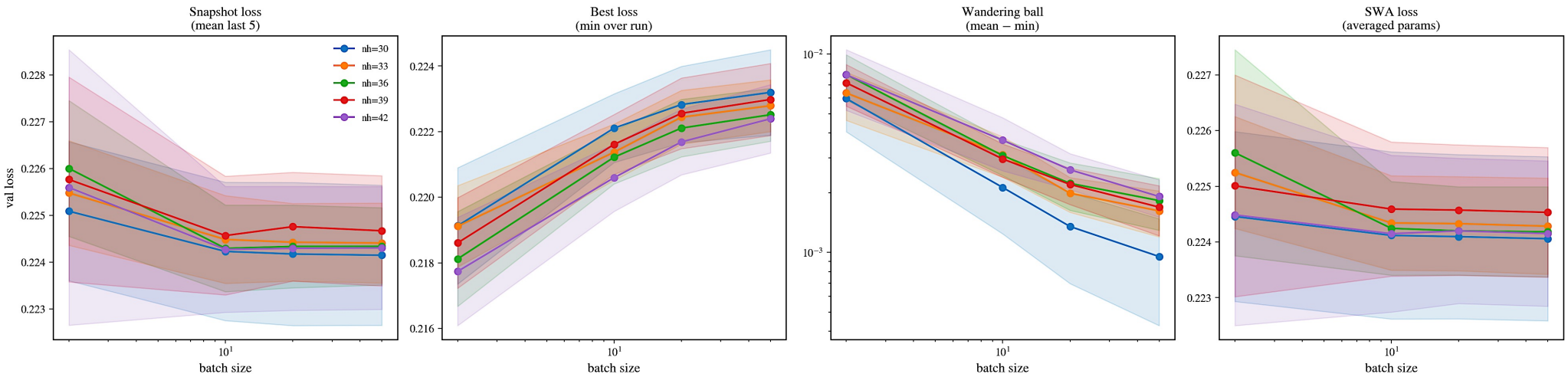

  Divide by len(swas): scalar division broadcasts
  over every element, giving the per-element mean.
```

Parameter Estimation 108

Code explanation

SWA takes averages over trainable parameters (the weights we are trying to optimize)

PE 107: What is the best model?



The SWA panel.

- SWA replaces the snapshot loss, a random orbit sample, with the loss of the parameter-averaged model.

  Panel 2 (best loss) vs panel 4 (SWA):
  - At bs=2: min ~0.218, SWA ~0.224
  - At bs=30: min ~0.223, SWA ~0.224

So SWA loses to min by ~0.005 at small batch and is roughly comparable at large batch. The parameter centroid sits inside the basin, not at the floor. Maybe the orbit is not perfectly symmetric.

Why we care about SWA at all: a flat basin tolerates parameter averaging (averaged params stay in the low-loss region), a sharp basin does not (averaging escapes the narrow well, loss climbs). So a low SWA loss is implicit evidence of low minimum curvature.

This is the generalization-friendly regime.

```
Exercise is to redo our temperature with the same code
as the one require to send the computations to the GPU

RULES

1. Tensors and modules must be on the same device. Mixing
   raises "RuntimeError: Expected all tensors to be on the
    same device."

2. Move once, not per-batch. Calling .to(device) inside the
   inner loop copies every step and dominates training time.

3. .item() moves the scalar back to CPU automatically, so
   val_losses is a CPU-side list of Python floats — ready for
   plotting.

Other device-aware bits:

- torch.randperm(n, device=device): RNG defaults to CPU.
  Without device=device you index a GPU tensor with a CPU
  index and pay a silent copy.

- torch.manual_seed(s) seeds the CPU RNG;
  torch.cuda.manual_seed_all(s) covers all GPUs.
  PyTorch >=1.8 handles both via torch.manual_seed when CUDA
  is available, but the explicit version is more robust.
```

```python
import torch.nn as nn
import torch.optim as optim
import copy
import torch, numpy as np, random
device = torch.device(
    "cuda" if torch.cuda.is_available()
    else "mps" if torch.backends.mps.is_available() # macOS metal
    else "cpu"
)
SEED = 0
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED) # GONNA FIX THE SEED FOR THE NEXT TESTS
print(device)
```

Parameter Estimation 109

Redo PE106 exercise, but sending data and tensors to a device (which can be a GPU)

```python
def make_dataset(nsample, err, low=20.0, high=80.0):
  tF0 = torch.rand(nsample) * (high - low) + low
  tC0 = (tF0 - 32) * 5/9 + torch.randn(nsample) * err
  return tF0, tC0   # raw only
tF0, tC0 = make_dataset(nsample=2000, err=5)

# Split FIRST
nsamples = tC0.shape[0]
nval = int(0.95 * nsamples)
shuffled_indices = torch.randperm(nsamples)
train_indices = shuffled_indices[:-nval]
val_indices   = shuffled_indices[-nval:]

# Normalization stats from TRAIN ONLY
tF_mean = tF0[train_indices].mean()
tF_std  = tF0[train_indices].std()
tC_mean = tC0[train_indices].mean()
tC_std  = tC0[train_indices].std()

# Apply the SAME train-derived stats to both splits
tF = ((tF0 - tF_mean) / tF_std).unsqueeze(1)
tC = ((tC0 - tC_mean) / tC_std).unsqueeze(1)

# Move data to device
tF_train = tF[train_indices].to(device)
tC_train = tC[train_indices].to(device)
tF_val   = tF[val_indices].to(device)
tC_val   = tC[val_indices].to(device)

# .shape returns Python ints, not tensors. So ntrain has
# no device. Shape metadata is always CPU-side
ntrain   = tF_train.shape[0]
```

Parameter Estimation 109

Redo PE106 exercise, but sending data and tensors to a device (which can be a GPU)

```python
def dense_layer(n):
  block = nn.Sequential(nn.Linear(n, n), nn.Tanh())
  return block

mse = nn.MSELoss()  # Constructor

def training_loop_batched(nepochs, optimizer, model, bs):
  ntrain = tF_train.shape[0]
  val_losses = []
  for epoch in range(1, nepochs+1):
    # build the shuffle index ON the data's device. tF_train
    # lives on `device`; a CPU index into it = a silent
    # host->device copy every epoch
    perm = torch.randperm(ntrain, device=device)
    for start in range(0, ntrain, bs):
      idx  = perm[start:start+bs]
      tp   = model(tF_train[idx])   # tF_train already on device
      loss = mse(tp, tC_train[idx]) # tC_train already on device
      optimizer.zero_grad(set_to_none=True)
      loss.backward()
      optimizer.step()
    with torch.no_grad():
      # .item() pulls the scalar back to CPU automatically
      # val_losses stays a CPU list of plain Python floats
      val_losses.append(mse(model(tF_val), tC_val).item())
  return val_losses

nepochs_base = 60
x_base  = 10
bs_base = ntrain // x_base
lr_base  = 5e-3
nh_base = 30
```

Parameter Estimation 109

Redo PE106 exercise, but sending data and tensors to a device (which can be a GPU)

```
n_seeds = 10
idx_sort = torch.argsort(tF0)  # tF0 is still CPU->CPU index,
# Accumulator: all_losses[nh][bs] = {'mean': [seed0, seed1, ...],
#                                    'min': [...], 'gap': [...]}
all_losses = {}

for seed in range(n_seeds):
  torch.manual_seed(seed)
  # manual_seed(seed) already covers CUDA on PyTorch>=1.8;
  # this explicit call (didactics)
  if torch.cuda.is_available():
    torch.cuda.manual_seed_all(seed)
  for nh in range(nh_base, nh_base+12+1, 3):
    # model is born on CPU -> move it to device.  Do this
    # before deepcopy/optimizer so init_state, params,
    # and grads all live on the same device as the data.
    lmodel2 = nn.Sequential(nn.Linear(1, nh),
      *[dense_layer(nh) for _ in range(ndense)], nn.Linear(nh, 1),
    ).to(device)

    # now a device-side snapshot
    init_state = copy.deepcopy(lmodel2.state_dict())
    if seed == 0:
      all_losses[nh] = {}

    for x in [2, 5, 10, 50]:
      bs      = max(1, ntrain // x)
      nepochs = max(1, round(nepochs_base * x_base / x))
      lr      = lr_base
      # in-place copy; params stay on device
      lmodel2.load_state_dict(init_state)
      opt = optim.SGD(lmodel2.parameters(), lr=lr)
      val_losses = training_loop_batched(nepochs=nepochs,
                                         optimizer=opt,
                                         model=lmodel2,
                                         bs=bs)
      tail    = val_losses[-5:]
      avg_val = sum(tail) / len(tail)
      min_val = min(val_losses)
      gap     = avg_val - min_val

      if bs not in all_losses[nh]:
        all_losses[nh][bs] = {'mean': [], 'min': [], 'gap': []}
      all_losses[nh][bs]['mean'].append(avg_val)
      all_losses[nh][bs]['min'].append(min_val)
      all_losses[nh][bs]['gap'].append(gap)
```

Parameter Estimation 109

Redo PE106 exercise, but sending data and tensors to a device (which can be a GPU)

Plot the same as PE106

```
-  PRETEND the data is too big for RAM and for the GPU
   (it is small, but we treat it as huge). Same
   temperature fit as Part VI, two new tools.

-  RAM: a numpy memmap keeps the data on disk and we touch
   only CHUNK rows at a time, so the split, the stats, and
   the normalization are all streamed.

-  GPU: we load the data in chunks that fit device memory
   (weights + data + a safety margin). We are CPU-only here
   (Docker), so we EMULATE a GPU of GPU_MEM bytes.

-  KEY IDEA: reading one batch at a time from disk is
   communication-bound and dominates the runtime. Instead,
   each epoch we read a whole chunk once, send it once, and
   run all of its batches on-device.

-  Unchanged: the model, the loss, and the experiment driver.
```

```python
import torch.nn as nn
import torch.optim as optim
import copy
import torch, numpy as np, random
device = torch.device(
    "cuda" if torch.cuda.is_available()
    else "mps" if torch.backends.mps.is_available() # macOS metal
    else "cpu"
)
SEED = 0
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED) # GONNA FIX THE SEED FOR THE NEXT TESTS
print(device)
```

Parameter Estimation 110

Redo PE106, but emulating data that is too large to fit on CPU and GPU RAM

Step 1: how to compute data vectors that do not fit into RAM and save them

```python
import os
import numpy as np
from numpy.lib.format import open_memmap

# In real life this dataset is ALREADY on disk, written
# once by some upstream pipeline (think TB-scale). We fake
# that one-time step here; the exercise proper starts in
# the NEXT cell by reading from disk -- never all in RAM.
CHUNK = 2500                    # rows per write to disk

def make_dataset(nsample, err, low=20.0, high=80.0):
  # Write to *.tmp, then os.replace() onto the real name
  # only when COMPLETE. These arrays can take HOURS to
  # build in research; a mid-run crash must not clobber a
  # good file. os.replace is atomic -> always whole.
  Ftmp, Ctmp = "tF.tmp.npy", "tC.tmp.npy"
  Fmm = open_memmap(Ftmp, mode="w+",
                    shape=(nsample, 1), dtype="float32")
  Cmm = open_memmap(Ctmp, mode="w+",
                    shape=(nsample, 1), dtype="float32")
  # Walk the rows in windows of CHUNK: s = window start,
  # e = its end (clamped to nsample). Each pass builds and
  # writes only those e-s rows, so RAM holds CHUNK, not N.
  for s in range(0, nsample, CHUNK):
    e = min(nsample, s + CHUNK)
    f = torch.rand(e - s) * (high - low) + low
    c = (f - 32) * 5/9 + torch.randn(e - s) * err
    Fmm[s:e, 0] = f.numpy()      # window -> disk
    Cmm[s:e, 0] = c.numpy()
  Fmm.flush(); del Fmm
  Cmm.flush(); del Cmm
  os.replace(Ftmp, "tF.npy")   # atomic publish to disk
  os.replace(Ctmp, "tC.npy")

make_dataset(nsample=2000, err=5)
```

Parameter Estimation 110

Redo PE106, but emulating data that is too large to fit on CPU and GPU RAM

Step 1: How to compute data vectors that do not fit into RAM and save them

```python
# Open it as memmaps; we never load the whole thing in RAM.
tF = np.load("tF.npy", mmap_mode="r+", allow_pickle=False)
tC = np.load("tC.npy", mmap_mode="r+", allow_pickle=False)
N  = tF.shape[0]

# A shuffle is just a permutation of N integers (~8 B each):
# 50M rows (the max in ML emulators) is 400 MB, fits in RAM
# Only the DATA is too big (data vector is long), so rows
# stay on disk and we gather them by shuffled index.
nval = int(0.95 * N)
# .numpy() so these index the numpy memmaps on disk,
# not torch tensors (zero-copy: shares the CPU buffer).
order = torch.randperm(N).numpy()
train_idx = order[:-nval]; val_idx = order[-nval:]
ntrain = train_idx.shape[0]

def stream_chunks(idx, chunk):
  # walk a row-index array in blocks of chunk rows, each
  # block SORTED so the scattered disk reads run sequential.
  for a in range(0, len(idx), chunk):
    # yield (not return) makes this a GENERATOR. Lazy ->
    # we never build all blocks at once, just the current one.
    yield np.sort(idx[a:a+chunk])

def stream_stats(mm, idx):
  # compute mean/std over rows without loading them all
  # gather CHUNK scattered rows at a time. Even though the
  # data is float32, accumulators are float64 to keep the
  # running totals (total sum) accurate. Why?  64-bit float
  # has more range, so no overflow
  n = len(idx); s1 = 0.0; s2 = 0.0
  for rows in stream_chunks(idx, CHUNK):
    x = np.asarray(mm[rows], dtype="float64")
    s1 += x.sum(); s2 += (x * x).sum()
  mean = s1 / n
  std  = np.sqrt((s2 - s1*s1/n) / (n - 1))   # unbiased
  return np.float32(mean), np.float32(std)

# Normalization stats from TRAIN ROWS ONLY (streamed)
tF_mean, tF_std = stream_stats(tF, train_idx)
tC_mean, tC_std = stream_stats(tC, train_idx)

def load_rows(rows):
  # gather RAW rows, normalize on the fly with TRAIN stats,
  # return device tensors. Raw files stay pristine, so no
  # double-normalize and no 2nd on-disk copy of big data.
  f = torch.from_numpy(tF[rows])
  c = torch.from_numpy(tC[rows])
  Fc = (f - float(tF_mean)) / float(tF_std)
  Cc = (c - float(tC_mean)) / float(tC_std)
  return Fc.to(device), Cc.to(device)
```

Parameter Estimation 110

Redo PE106, but emulating data too large to fit on CPU and GPU RAM

Step 2: how to read data vectors that do not fit into RAM. Use **memmaps** (the **mmap_mode** option on **np.load**)

```python
def compute_batch_size_bytes(model, bs, sample_dims):
  # sample_dims = the shape of ONE sample (no batch axis).
  # Research example (emulating cosmic shear) each row
  # ROW is one shear data vector (contiguous 1D vector),
  # ordered by ordered by z-bin PAIR, then angle

  # model.parameters() is an ITERATOR over the weight
  # tensors; next() grabs the FIRST one. .device is where
  # that tensor lives. We put x there too so they match.
  dev = next(model.parameters()).device
  # dummy batch of shape (bs, *sample_dims); only the
  # SHAPES matter for memory, not the values -> zeros
  x = torch.zeros(bs, *sample_dims, device=dev)

  total = 0 # running byte count of saved tensors
  def pack(t):
    # pack: the FIRST callback. autograd calls it the
    # moment a tensor is saved during forward, and stores
    # whatever pack RETURNS. We use it to record the size
    # and return t unchanged. (+= would make total a new
    # local -> nonlocal points it at the outer total.)
    nonlocal total
    total += t.numel() * t.element_size()
    return t
  def unpack(stored):
    # unpack: the 2nf callback. autograd call it in BACKWARD
    # to rebuild the tensor from what pack stored. We never
    # call backward, but it is needed by saved_tensors_hooks,
    # so we just hand the stored tensor straight back.
    return stored

  # saved_tensors_hooks lets you define a custom way to STORE
  # activations between forward/backward (to save memory):
  # pack transforms each tensor on the way IN (e.g. compress),
  # unpack reverses it on the way OUT. The pair must round-trip:
  # unpack(pack(t)) == t. We use a no-op pair to spy on the sizes.
  hooks = torch.autograd.graph.saved_tensors_hooks  # alias
  with hooks(pack, unpack):
    # saving happens DURING forward, so the instant model(x)
    # returns, total is complete -- no backward needed.
    out = model(x)
  # batch's own input + output buffers. element_size() =
  # THIS tensor's bytes per element (float32->4, float64->8)
  io = (x.numel() * x.element_size()
        + out.numel() * out.element_size())
  return total + io
```

Parameter Estimation 110

Step 3: Loading directly from disk to GPU per batch is horribly inefficient. Need to do per group of batches

Q1: What is the size of one batch?

Parameter Estimation 110

Q3: Given the size of one batch (and model size, that is Q2), how many batches fit in the GPU RAM?

```
GPU_MEM = 128 * 1024          # emulated GPU RAM (bytes)

def compute_model_size_bytes(model):
  # trainable params + grads: resident the whole run
  p = sum(t.numel() for t in model.parameters())
  return p * 4 * 2            # float32 weights + grads

def batches_per_load(model, bs, sample_shape):
  # GPU room LEFT once the model is resident. On a real GPU the
  # model would be already on the card, then read the free
  # bytes with torch.cuda.mem_get_info(). Keep 20% spare.
  free = 0.8 * GPU_MEM - compute_model_size_bytes(model)
  #free, _ = torch.cuda.mem_get_info() # real world scenario
  per  = compute_batch_size_bytes(model, bs, sample_shape)
  return max(1, int(free // per))
```

```python
mse_sum = nn.MSELoss(reduction="sum")  # sum (pooled val)

def eval_val(model, load):
  # Validation data is on DISK too, so we STREAM it: read a
  # chunk of rows, send it to the device, run the model,
  # never holding the whole val set at once.
  #
  # We can use PyTorch's MSE here, but with reduction="sum"
  # not the default "mean". Why? We need the per-chunk SUM
  # so we can pool across chunks and divide by the true total
  # once -> exact MSE. (Averaging per-chunk means would be
  # wrong: the last chunk is smaller.)
  tot, cnt = 0.0, 0
  for rows in stream_chunks(val_idx, load):
    Fc, Cc = load_rows(rows)   # gather + normalize
    tot += mse_sum(model(Fc), Cc).item()
    # reduction="sum" summed the squared error over every
    # output number. In the research cosmic-shear case L
    # would NOT be 1 but n_pairs*n_theta. That is why we
    # use .numel() and not len(rows)
    cnt += Cc.numel()        # ELEMENTS, not rows
  return tot / cnt
```

```
Extra explanation on MSELoss used on validation

MODULE vs FUNCTIONAL: two ways to call the same op.

MODULE: built once, settings baked in, then reused.
mse_sum = nn.MSELoss(reduction="sum") # sum, val pool
loss = mse_sum(pred, y)

FUNCTIONAL: the raw op; pass every setting each call.
F.mse_loss(pred, y, reduction="sum")
(needs: import torch.nn.functional as F)
```

Parameter Estimation 110

Q4: Given validation may also not fit into RAM, how do we compute validation loss?

```
def dense_layer(n):
  block = nn.Sequential(nn.Linear(n, n), nn.Tanh())
  return block

mse = nn.MSELoss()  # Constructor

def training_loop_batched(nepochs, optimizer, model, bs):
  ntrain = len(train_idx)
  # Data is on DISK, so one read per batch would be I/O bound
  # Instead we pre-load a chunk of many batches at once.
  nb = batches_per_load(model, bs, tF.shape[1:])
  load = nb * bs   # rows fetched per disk read
  val_losses = []
  for epoch in range(1, nepochs+1):
    # shuffle which disk rows we visit (indices are cheap;
    # only the long data vectors are too big to shuffle).
    epoch_rows = train_idx[torch.randperm(ntrain).numpy()]
    for cs in range(0, ntrain, load):
      # ONE disk read + ONE host->device copy per chunk,
      # covering nb batches. Sort the rows so the disk
      # read is sequential (faster), not scattered.
      rows = np.sort(epoch_rows[cs:cs+load])
      Fc, Cc = load_rows(rows)      # gather + normalize
      # now run this chunk's batches entirely on-device,
      # reshuffled, with no more disk or transfer cost.
      bp = torch.randperm(Fc.shape[0], device=device)
      for s in range(0, Fc.shape[0], bs):
        b    = bp[s:s+bs]
        loss = mse(model(Fc[b]), Cc[b])
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()
    with torch.no_grad():
      # .item() inside eval_val pulls scalars to CPU, so
      # val_losses stays a CPU list of plain Python floats
      val_losses.append(eval_val(model, load))
  return val_losses
```

Parameter Estimation 110

Step 4: How to go on each epoch, loading not one batch at a time, but chunks (how many? whatever fits in GPU memory).

```
nepochs_base = 60; x_base  = 10
bs_base = ntrain // x_base
lr_base  = 5e-3; nh_base = 30
ndense = 1; n_seeds = 10
# Accumulator: all_losses[nh][bs] = {'mean': [seed0, seed1, ...],
#                                    'min': [...], 'gap': [...]}
all_losses = {}

for seed in range(n_seeds):
  torch.manual_seed(seed) # manual_seed already covers CUDA
  for nh in range(nh_base, nh_base+12+1, 3):
    # model is born on CPU -> move it to device. Do this
    # before deepcopy/optimizer so init_state, params,
    # and grads all live on the same device as the data.
    lmodel2 = nn.Sequential(nn.Linear(1, nh),
      *[dense_layer(nh) for _ in range(ndense)], nn.Linear(nh, 1),
    ).to(device)

    # now a device-side snapshot
    init_state = copy.deepcopy(lmodel2.state_dict())
    if seed == 0:
      all_losses[nh] = {}

    for x in [2, 5, 10, 50]:
      bs      = max(1, ntrain // x)
      nepochs = max(1, round(nepochs_base * x_base / x))
      lr      = lr_base
      # in-place copy; params stay on device
      lmodel2.load_state_dict(init_state)
      opt = optim.SGD(lmodel2.parameters(), lr=lr)
      val_losses = training_loop_batched(nepochs=nepochs,
                                         optimizer=opt,
                                         model=lmodel2,
                                         bs=bs)

      tail    = val_losses[-5:]
      avg_val = sum(tail) / len(tail)
      min_val = min(val_losses)
      gap     = avg_val - min_val

      if bs not in all_losses[nh]:
        all_losses[nh][bs] = {'mean': [], 'min': [], 'gap': []}
      all_losses[nh][bs]['mean'].append(avg_val)
      all_losses[nh][bs]['min'].append(min_val)
      all_losses[nh][bs]['gap'].append(gap)
```

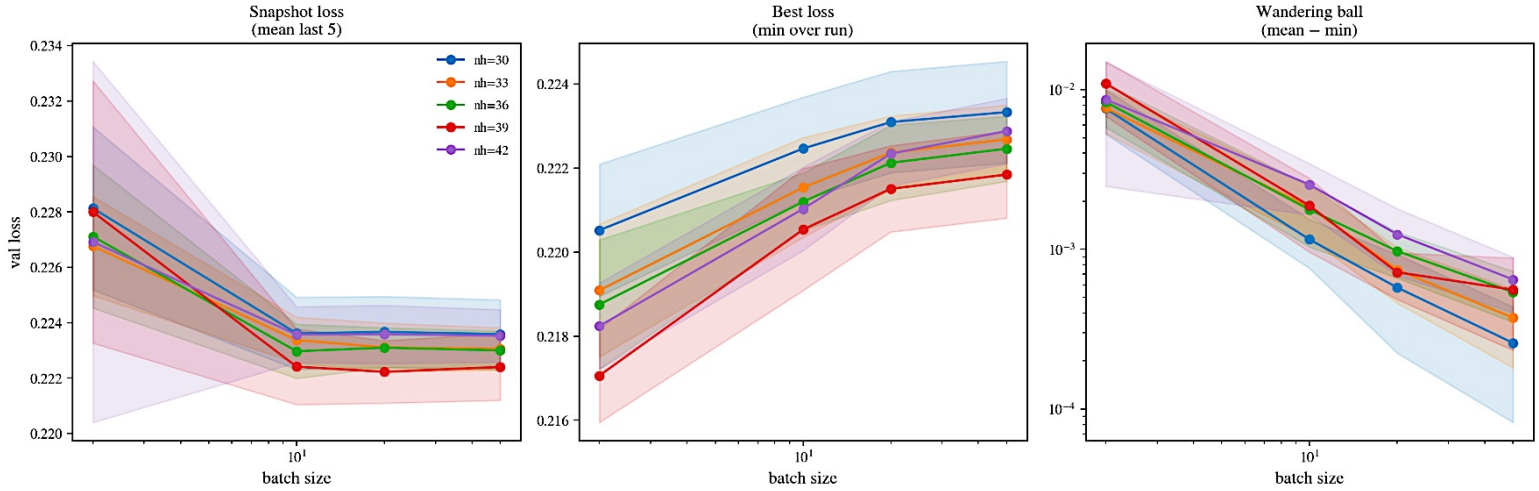
Parameter Estimation 110

Step 5: training loops over seed values and hyperparameters are the same as PE106 (w/o split of data and batches)

Parameter Estimation 110

```
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
metrics = ['mean', 'min', 'gap']
titles  = ['Snapshot loss\n(mean last 5)',
           'Best loss\n(min over run)',
           'Wandering ball\n(mean − min)']

for ax, metric, title in zip(axes, metrics, titles):
  for nh, bs_dict in all_losses.items():
    # (fill_between needs monotonic x).
    bss = sorted(bs_dict.keys())
    arr = np.array([bs_dict[b][metric] for b in bss])  # (n_bs, n_seeds)
    mu, sd = arr.mean(axis=1), arr.std(axis=1)
    line, = ax.plot(bss, mu, marker='o', label=f'nh={nh}')
    ax.fill_between(bss, mu - sd, mu + sd, alpha=0.2, color=line.get_color())
  ax.set_xscale('log')
  ax.set_xlabel('batch size', fontsize=12)
  ax.set_title(title, fontsize=12)
axes[0].set_ylabel('val loss', fontsize=12)
axes[2].set_yscale('log')
axes[0].legend(frameon=False, fontsize=9)
plt.tight_layout()
plt.show()
```



Step 7: same as PE106 (w/o split of data and batches)