Exercices

- Consider a random array with shape (100,2) representing coordinates, find point by point distances
- Considering a 2-dimensional array (matrix), how to get sum over the columns at once?
- Consider an arbitrary random array: compute its mean, median, and the standard deviation
- Compute a matrix rank (hint: `U, S, V = np.linalg.svd(Mat) # Singular Value Decomposition`)

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

Matplotlib

- Matplotlib is a plotting library
- In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB

```
[108] import matplotlib.pyplot as plt
```

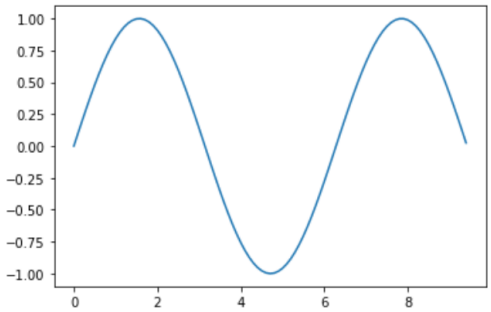
- By running this special iPython command, we will be displaying plots inline:

```
[109] %matplotlib inline
```

Plotting

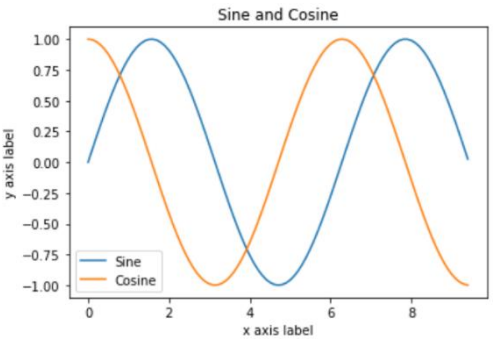
- The most important function in `matplotlib` is plot, which allows you to plot 2D data

```
[110] # Compute the x and y coordinates for points on a sine
      x = np.arange(0, 3 * np.pi, 0.1)
      y = np.sin(x)

      # Plot the points using matplotlib
      plt.plot(x, y)
      plt.show()
```



- With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

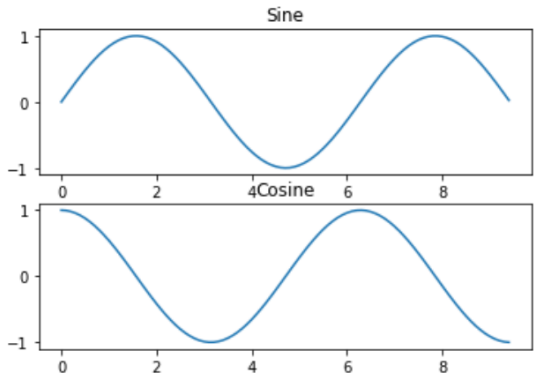
```
[111] y_sin = np.sin(x)
      y_cos = np.cos(x)

      # Plot the points using matplotlib
      plt.plot(x, y_sin)
      plt.plot(x, y_cos)
      plt.xlabel('x axis label')
      plt.ylabel('y axis label')
      plt.title('Sine and Cosine')
      plt.legend(['Sine', 'Cosine'])
      plt.show()
```



Subplots

- You can plot different things in the same figure using the subplot function

```
[112] # Compute the x and y coordinates for points on sine and cosine curves
      x = np.arange(0, 3 * np.pi, 0.1)
      y_sin = np.sin(x)
      y_cos = np.cos(x)

      # Set up a subplot grid that has height 2 and width 1,
      # and set the first such subplot as active.
      plt.subplot(2, 1, 1)

      # Make the first plot
      plt.plot(x, y_sin)
      plt.title('Sine')

      # Set the second subplot as active, and make the second plot.
      plt.subplot(2, 1, 2)
      plt.plot(x, y_cos)
      plt.title('Cosine')

      # Show the figure.
      plt.show()
```



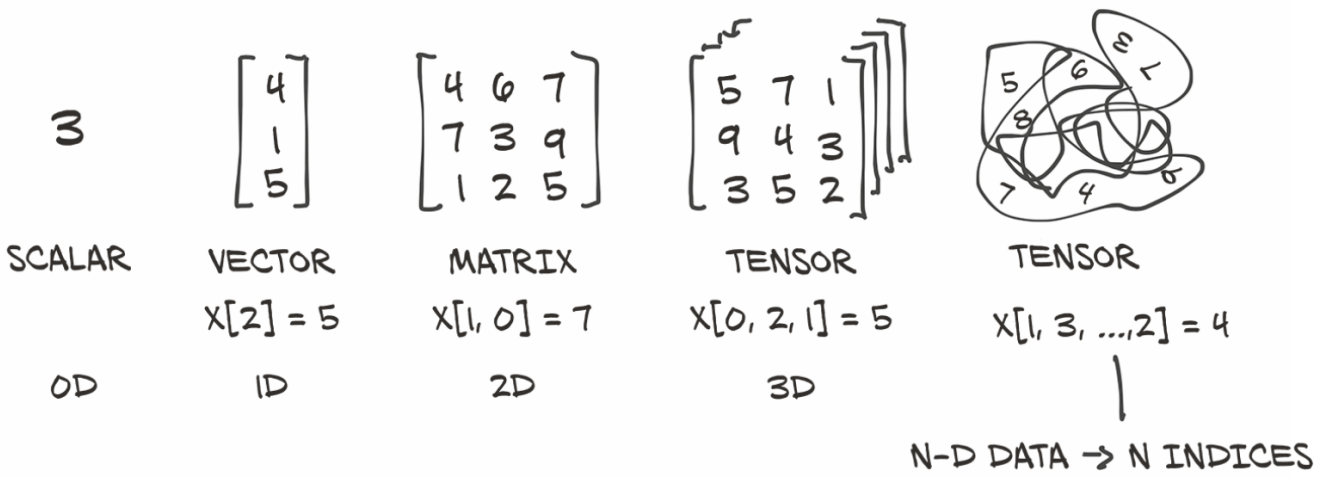
You can read much more about the `subplot` function in the documentation.

PyTorch

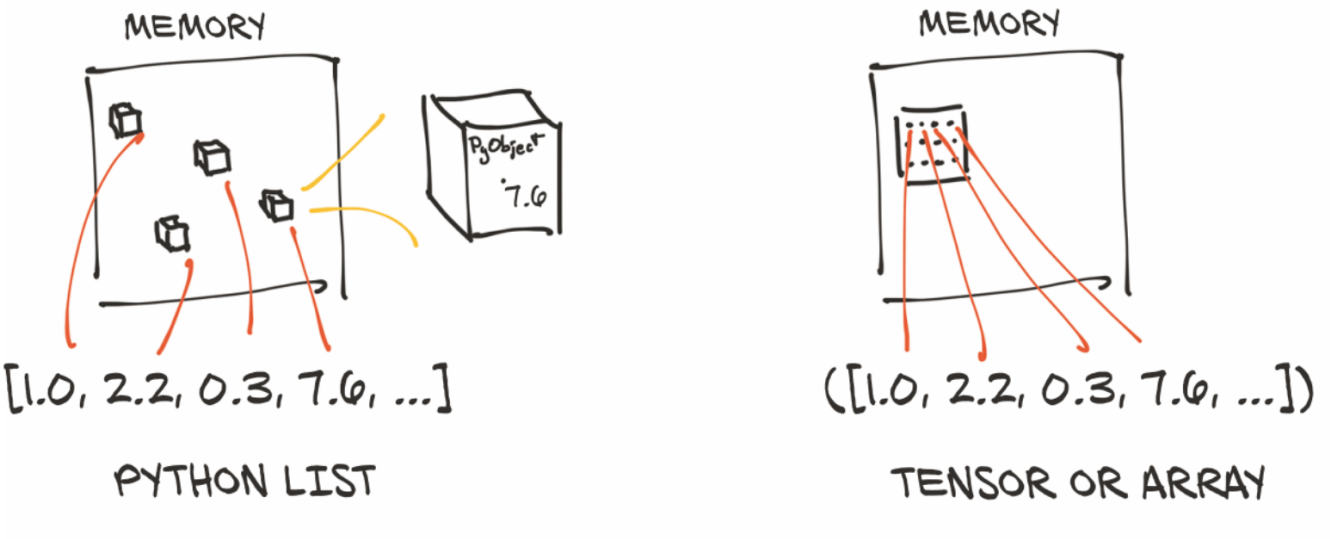
- Integration with the rest of the scientific libraries in Python, such as SciPy, Scikit-learn, and Pandas
- Compared to NumPy arrays, PyTorch tensors perform very fast operations on Graphical Processing Units (GPUs)
    - distribute operations on multiple devices or machines
    - keep track of the graph of computations that created them
    - important features when implementing a modern deep learning library.

Tensors

- Tensors are a specialized **data structure** that are very similar to **arrays** and **matrices**
- In **PyTorch**, we use tensors to encode the **inputs** and **outputs** of a model, as well as the model's **parameters**
- Tensors are similar to **NumPy** ndarrays, except that tensors can run on **GPUs**
- Tensors are also optimized for **automatic differentiation**
- The **dimensionality** of a tensor coincides with the number of **indices** or **axes** used to refer to scalar values (single elements) within the tensor



- Python **lists** or **tuples** of numbers are collections of Python objects that are **separately allocated** in memory in general
- **PyTorch tensors** or **NumPy arrays** are views over (typically) **contiguous memory cells** containing unboxed (i.e. not encapsulated into other container/object) numeric data types of C language rather than fully boxed Python objects
- Every element is the same data type among: 32 bit (4 byte) or 64 bit (8 byte) **float**, **int**, etc.



We need it to allocate and find faster the things. The elements of the tensor contains different data types.

Initializing a Tensor

- Tensors can be initialized in various ways, e.g. they can be created directly from other data (whose data type is automatically inferred for properly transforming to a tensor)

```
[2] import torch # initialize pythorch
    import numpy as np
```

```
[3] data = [[1,2],[3,4],[5,6]] # list of lists (items mutable)
    x_data = torch.tensor(data)
    x_data
```

```
    tensor([[1, 2],
            [3, 4],
            [5, 6]])
```

From a NumPy array

- Tensors can be created from NumPy arrays

```
[4] np_array = np.array(data)
    x_np = torch.from_numpy(np_array) # from numpy array
    x_np
```

```
    tensor([[1, 2],
            [3, 4],
            [5, 6]])
```

From another tensor

- The new tensor retains the properties (shape, datatype) of the argument tensor, unless explicitly overridden

```
[5] x_ones = torch.ones_like(x_data) # retains the properties of x_data
    print(f"Ones Tensor: \n {x_ones} \n")

    # rand_like() creates i.i.d. Uniform([0,1)) (pseudo)r.v.'s with properties of x_data.
    #But dtype argument overrides the datatype.
    x_rand = torch.rand_like(x_data, dtype=torch.float)
    print(f"Random Tensor: \n {x_rand} \n")
```

```
    Ones Tensor:
     tensor([[1, 1],
            [1, 1],
            [1, 1]])

    Random Tensor:
     tensor([[0.9582, 0.3861],
            [0.4511, 0.9384],
            [0.8318, 0.6496]])
```

With (pseudo)random or constant values

- Unpredictable numbers can be generated in several ways, for example:
  - from dedicated hardware sampling observations from physical phenomena, such as quantum mechanics or statistical mechanics (truly **random**). Related profound question: what is truly random in nature?
  - deterministically through an algebraic & code theory based algorithm starting from a *seed* value (**pseudorandom**): observed properties look like random. This is Python/NumPy/PyTorch
- Typically initialization with: zeros, ones, some other constants, or numbers randomly sampled from a specific probability distribution. Eg.:
  - `zeros` allows to initialize the tensors explicitly to 0
  - `ones` allows to initialize the tensors explicitly to 1
  - `randn` initializes the matrix randomly sampling the entries i.i.d. $N(0,1)$
- `shape` is a tuple of tensor's dimensions

```
[6] shape=(3,4)
    zero_tensor = torch.zeros(shape)
    zero_tensor
```

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

```
[7] one_tensor = torch.ones(shape)
    one_tensor
```

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

```
[8] randn_tensor = torch.randn(shape)
    randn_tensor
```

```
tensor([[ 9.8892e-01, -5.6992e-01, -9.7276e-03,  3.3119e-01],
        [ 1.8917e-01,  6.3395e-01, -7.9268e-02,  5.3893e-01],
        [ 1.8060e+00,  1.1255e+00,  1.5434e+00,  1.1552e-03]])
```

Attributes of a Tensor

- Tensor attributes describe its **shape** (`shape`), **datatype** (`dtype`), and the **device** (`device`) hosting the tensor. We will see their meaning.
    - **CPU**: main memory operated with computer's CPU
    - **GPU**: Graphical Processing Unit equipped with its own memory

```
[9] tensor = torch.rand(3,4)

    print(f"Shape of tensor: {tensor.shape}")
    print(f"Datatype of tensor: {tensor.dtype}")
    print(f"Device tensor is stored on: {tensor.device}")
    tensor.device
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
device(type='cpu')
```

OPERATIONS ON TENSORS

- Over **100 tensor operations**, including arithmetic, linear algebra, matrix manipulation (transposing, indexing, slicing), sampling and more are comprehensively described here

Element-wise lifting

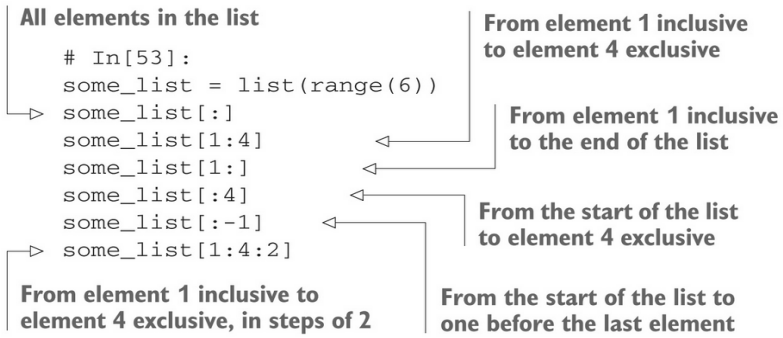
- Some of the simplest and most useful operations are the *element-wise* or *entry-wise* operations. Eg. Hadamard product between (compatible) matrices A and B:
  $C = A \odot B$
  with entries $C_{ij} = A_{ij}B_{ij}$. It just extends the product between scalars to matrices, in a particular and natural way.
- Likewise, we could take a scalar-valued function of one or several scalar variables, or an operation between scalars, and **extend** it to a function, or respectively operation, involving vectors/matrices/tensors in **element-wise** manner. This is sometimes called **lifting** the function/operation.
    - The common standard **arithmetic operators** (`+`, `-`, `*`, `/`, and `**`) have all been **lifted** to elementwise operations for any Pytorch tensors having *same shape*.

```
[10] twos = 2 * torch.ones((2,3,4)) # scalar 2 multiplication is broadcast
     base = torch.randn_like(twos)
     base ** twos # each element of the base will be raised ???
```

```
tensor([[[1.2295e-02, 6.2881e-02, 4.7376e-01, 9.9867e-01],
         [3.5083e-02, 1.1370e-01, 1.4839e-01, 7.8085e-02],
         [6.4754e-02, 5.4232e-01, 1.2453e-01, 1.0313e+00]],

        [[2.4843e-02, 8.9167e-02, 3.9468e+00, 4.6183e-01],
         [1.7501e-05, 1.7438e-02, 2.1516e+00, 9.0589e-01],
         [1.7926e+00, 5.3962e-01, 2.5991e-01, 2.4097e+00]]])
```

Standard numpy-like indexing and slicing

- Tensors use an indexing notation analogous to standard Python lists and NumPy, for accessing and assigning




```
[11] some_list = list(range(6))
     print(some_list[:])
     print(some_list[1:4])
     print(some_list[1:])
     print(some_list[:4])
     print(some_list[:-1])
     print(some_list[1:4:2])
```

```
     [0, 1, 2, 3, 4, 5]
     [1, 2, 3]
     [1, 2, 3, 4, 5]
     [0, 1, 2, 3]
     [0, 1, 2, 3, 4]
     [1, 3]
```

```
[12] tensor = torch.ones(4, 4)
     print('First row: ',tensor[0])
     print('First column: ', tensor[:, 0])
     print('Last column:', tensor[..., -1])
     print('Last column:', tensor[Ellipsis, -1]) # We can replace ... with Ellipsis
     tensor[:,1] = 0
     print(tensor)
```

```
     First row:  tensor([1., 1., 1., 1.])
     First column:  tensor([1., 1., 1., 1.])
     Last column: tensor([1., 1., 1., 1.])
     Last column: tensor([1., 1., 1., 1.])
     tensor([[1., 0., 1., 1.],
             [1., 0., 1., 1.],
             [1., 0., 1., 1.],
             [1., 0., 1., 1.]])
```

- `Ellipsis` and `...` in indexing both mean: *all remaining dimensions of the tensor are fully selected*

Joining and reshaping tensors

- You can use `torch.cat` to **concatenate** a sequence of tensors *along* a given **dim**ension

```
[13] t1 = torch.cat([tensor, tensor, tensor, tensor], dim=1)
     print(t1)
```

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

- Or build a tensor by changing the shape (entries are copied and rearranged)

```
[14] a = torch.arange(12.) # arange()'s argument can be int or float type
     a.reshape((2, 3, 2)) # a new tensor object is created; a is untouched
```

```
     tensor([[[ 0.,  1.],
              [ 2.,  3.],
              [ 4.,  5.]],

             [[ 6.,  7.],
              [ 8.,  9.],
              [10., 11.]]])
```

- ... even letting it *guess* one unspecified dimension, by writing `-1`:

```
[15] b = torch.tensor([[0,1,2],[3,4,5]])
     print(torch.reshape(b,(-1,)))
     print(torch.reshape(b,(3,-1)))
```

```
tensor([0, 1, 2, 3, 4, 5])
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

- Warning: What is the new ordering of the entries in the reshaped tensor?
  - the tensor entries are scanned and rearranged progressively following the innermost axes (corresponding to rightmost indices)

Arithmetic operations

```
[16] # This computes the matrix multiplication (row-by-column). y1, y2, y3, y4, y5 will have the SAME values
     y1 = tensor @ tensor.T # T attribute or t() method make a transposed copy
     y2 = tensor.matmul(tensor.T)
     y3 = torch.rand_like(tensor)
     torch.matmul(tensor, tensor.T, out=y3)
     y4 = torch.matmul(tensor, tensor.T)
     y5 = torch.mm(tensor, tensor.t())
     y4==y5
```

```
tensor([[True, True, True, True],
        [True, True, True, True],
        [True, True, True, True],
        [True, True, True, True]])
```

```
[17] # This computes the element-wise product. z1, z2, z3, z4 will have the same value
     z1 = tensor * tensor
     z2 = tensor.mul(tensor)
     z3 = torch.rand_like(tensor)
     torch.mul(tensor, tensor, out=z3)
     z4 = torch.mul(tensor, tensor)
     z3==z4
```

```
tensor([[True, True, True, True],
        [True, True, True, True],
        [True, True, True, True],
        [True, True, True, True]])
```

Single-element tensors

- If you have a one-element tensor, for example by aggregating all values of a tensor into one value, you can convert it to a Python numerical value using `item()`

```
[27] agg = tensor.sum()
     agg_item = agg.item()
     print(agg_item, type(agg_item))
```

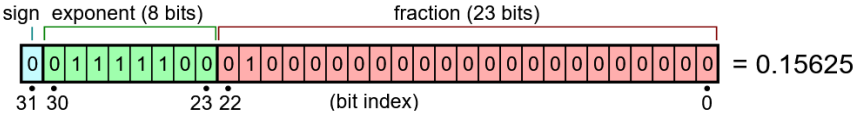
```
3.0 <class 'float'>
```

Tensor element types

- Many data science libraries rely on NumPy or introduce **dedicated data structures** like PyTorch tensors, which provide efficient **low-level implementations** of numerical data structures and **related operations** on them, wrapped in a convenient high-level API (Application Programming Interface: the set of functions, constants and functionalities offered to the programmer by a library)

PyTorch data types

- The `dtype` argument to tensor constructors specifies the **numerical data type** of the tensor's entries
- Some data types and corresponding values for the **dtype argument**:
  - `torch.float32` or `torch.float`: 32-bit floating-point

> Example of float32 encoding of 0.15625:
>
> 
>
>
> to be understood as:
>
> $(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$
>
> i.e. $(-1)^0 \times 2^{(1111100)_2 - 127} \times 1.01_2 = (+1) \times 2^{-3} \times 1.25 = 0.15625$
>
> In general, denoting *E* = exponent, we have:
>
> $$\text{value} = (-1)^{\text{sign}} \times 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right).$$

  - `torch.float64` or `torch.double`: 64-bit, **double-precision** floating-point
  - `torch.float16` or `torch.half`: 16-bit, half-precision floating-point
  - `torch.int8`: **signed** 8-bit integers (-128 to 127)
  - `torch.uint8`: **unsigned** 8-bit integers (0 to 255)
  - `torch.int16` or `torch.short`: signed 16-bit integers
  - `torch.int32` or `torch.int`: signed 32-bit integers
  - `torch.int64` or `torch.long`: signed 64-bit integers
  - `torch.bool`: Boolean
- Floating-point arithmetic above: complying with IEEE 754 technical standard
  - several kinds of numerical errors, leading to approximations acceptable for most scientific applications
  - characteristics common to Python, NumPy and other programming languages adhering to IEEE 754

Managing a tensor's dtype

- In order to allocate a tensor of the right numeric type, we can **specify** the **proper** `dtype` as an argument to the constructor

```
[19] double_points = torch.ones(10, 2, dtype=torch.double)
     short_points = torch.tensor([[1,2],[3,4]], dtype=torch.short)
     print(double_points)
     print(short_points)
```

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.],
        [1., 1.]], dtype=torch.float64)
tensor([[1, 2],
        [3, 4]], dtype=torch.int16)
```

```
print(short_points.dtype)
print(double_points.dtype)
```

```
torch.int16
torch.float64
```

- Type **casting** methods, such as: `half()`, `long()`, `double()`, `type(...)`, and more ...

```
[20] short_points = torch.ones(10, 2).short()
     double_points = torch.zeros(10, 2).double()
     long_points = torch.zeros(4,3).type(torch.long)
     print(short_points.dtype)
     print(double_points.dtype)
     print(long_points.dtype)
```

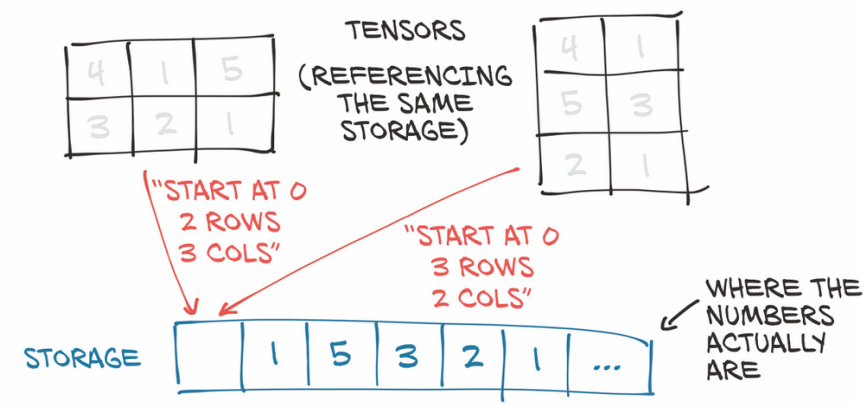
```
torch.int16
torch.float64
torch.int64
```

```
[21] # even the more complex to() method:
     double_points = torch.zeros(10, 2).to(torch.double)
     short_points = torch.ones(10, 2).to(dtype=torch.short)
     print(short_points.dtype)
     print(double_points.dtype)
```

```
torch.int16
torch.float64
```

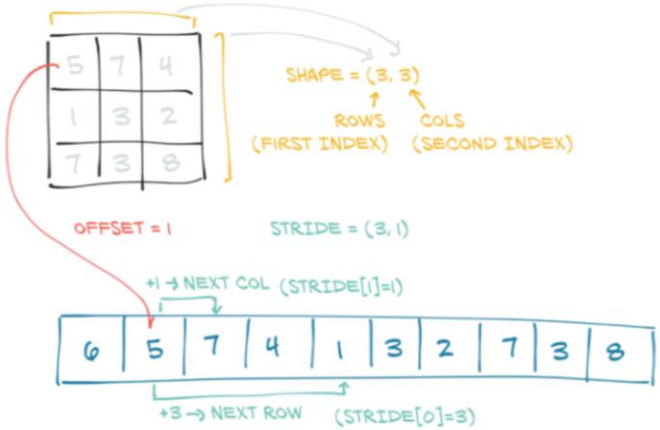
Tensors: Scenic views of storage *

- Allocation and storage of PyTorch tensor values is managed by `torch.Storage` instances
    - a **contiguous one-dimensional (array) block** of memory containing data of the same type



Tensor metadata

- In order to index into a storage instance, tensors rely on some "meta" information (additional auxiliary information) that, together with their storage, unequivocally locate the content:
    - **shape** attribute or **size()** method: gives the usual tuple describing the dimensions
    - **offset**: is the index in the storage corresponding to the **first element** in the tensor.
    - **stride**: gives the number of elements in the storage that need to be **skipped over** to obtain the **next element** along each dimension



In-place operations

- Operations that store the result into the operand are called **in-place**. They are **denoted** by a **_ suffix**
  - For example: `x.copy_(y)`, `x.t_()`, will change `x`

```
[30] tensor = torch.tensor(-2.)
     print(tensor)
     print(id(tensor))
     tensor.add_(5)
     print(tensor)
     print(id(tensor))

     aaa = torch.tensor([1])
     bbb = aaa + 0
     print(id(aaa))
     print(id(bbb))
```

```
tensor(-2.)
140610322788016
tensor(3.)
140610322788016
140610398635344
140610322674864
```

- **Warning**:
  In-place operations **save memory**, but can be problematic when computing **derivatives** because of an immediate **loss of history**.
  - hence, they must be used carefully.

Memory sharing between PyTorch tensor and NumPy array

- PyTorch tensors on CPU and NumPy arrays can **share** their underlying **memory** cell locations: can be thought of as two "views" of the same content
  ⇒ *Warning*: in-place changing one's content will reflect in other one's

- Example: from PyTorch tensor to NumPy array

```
[31] t = torch.ones(3)
     print(f"t: {t}")
     n = t.numpy()
     print(f"n: {n}")

     t.add_(1)
     print(f"t: {t}")
     print(f"n: {n}")
```

Now let's change the tensor

```
t: tensor([1., 1., 1.])
n: [1. 1. 1.]
t: tensor([2., 2., 2.])
n: [2. 2. 2.]
```

- ... the same occurs viceversa: eg. when we used `tensor = torch.from_numpy(np_array)`. Example:

```
[23] n = np.ones(2)
     t = torch.from_numpy(n)

     # Let's change the array
     np.add(n, 1, out=n)
     print(f"t: {t}")
     print(f"n: {n}")
```

```
t: tensor([2., 2.], dtype=torch.float64)
n: [2. 2.]
```

Device storing a Tensor

A tensor is stored in one device:

- **CPU**: main memory (RAM) operated with computer's CPU
- **GPU**: Graphical Processing Unit equipped with its own memory

By **default**: created on CPU. We **explicitly move tensors to the GPU** when needed

- *Warning*: moving **large tensors** across devices can be **expensive** in terms of time and memory (and energy)!

PyTorch tensors stored on GPU: for performing massively parallel, fast computations

- all **operations** will be carried out using **GPU hardware specific routines** implemented in the CUDA tensor type (see `torch.cuda` package)

Check availability of GPU through PyTorch's CUDA support

- CUDA drivers for the GPU need to be installed in the OS
- Google Colab's environment already has these requisites: first **Change runtime type** to **GPU** accelerated

```
[32] # Check if GPU is available
     torch.cuda.is_available()
```

```
     True
```

Managing a tensor's device

- Device type is also named by a string: `cpu` or `cuda`
  - `torch.device()` method: accepts this string as argument, returns an object representing the device
- When creating a tensor, **device** keyword argument tells where to place it:
  - it can be the string or the object representing the device

```
[ ]  if torch.cuda.is_available():
         gpu_dev = torch.device('cuda')
         points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device=gpu_dev) # same as ... devic
         points_gpu
```

- Or we could **copy a tensor** created on the CPU onto the GPU (or viceversa) using `to()` method:

```
[ ]  points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]]) # on default (CPU)
     points_gpu = points.to(device='cuda') # same as ... device=gpu_dev)
     points_gpu
```

```
     tensor([[4., 1.],
             [5., 3.],
             [2., 1.]], device='cuda:0')
```

Multiple GPUs

- If the machine has **more** than one **GPU**: we can decide for a specific GPU
  - by suffixing with `:` and a **zero-based integer** identifying the GPU on the machine

```
[ ]  points_gpu = points.to(device='cuda:0')
     points_gpu
```

```
     tensor([[4., 1.],
             [5., 3.],
             [2., 1.]], device='cuda:0')
```

- Any operation performed on the tensor, eg. multiplying by a constant, is carried out on the GPU

```
[ ]  # classical multiplication on the CPU
     points = 2 * points

     # This multiplication is performed on the GPU
     points_gpu = 3 * points_gpu
     points_gpu
```

```
tensor([[12.,  3.],
        [15.,  9.],
        [ 6.,  3.]], device='cuda:0')
```

- Note: tensor (`points_gpu`) stays on the device after the computation
- In order to copy the **tensor back** to the CPU: use the `to()` method with `device='cpu'` argument

```
[ ]  points_cpu = points_gpu.to(device='cpu')
     points_cpu.device
```

```
device(type='cpu')
```

- alternatively: **shorthand methods** `cpu()` and `cuda()` instead of `to()` to achieve the same goal

```
[ ]  points_gpu = points.cuda() # default to GPU index 0
     points_gpu = points.cuda(0)
     print(points_gpu)
     points_cpu = points_gpu.cpu()
```

```
tensor([[ 8.,  2.],
        [10.,  6.],
        [ 4.,  2.]], device='cuda:0')
```

Automatic Differentiation

Deep learning frameworks expedite the training of the networks by automatically calculating derivatives, i.e., *automatic differentiation*

- building a *computational graph* tracking data and operations
- *backpropagating*, i.e. tracing in backward direction through the computational graph, using the chain rule to fill in the partial derivatives with respect to each parameter (variable) represented in the graph.

A Simple Example

Differentiate the function

$$y = 2\mathbf{x}^\top \mathbf{x}$$

with respect to the column vector $\mathbf{x}$

1. Initialize the tensor

```
[7] import torch

    x = torch.arange(4.0)
    x
```

```
tensor([0., 1., 2., 3.])
```

... and declare that such variable is one for which we want the gradient to be calculated w.r.t.

- so, PyTorch proviveds a place to store the gradient information (`.grad` attribute)
- note: a gradient of a scalar-valued function with respect to a vector $\mathbf{x}$ is itself vector-valued and has the same shape as $\mathbf{x}$.

```
[8] x.requires_grad_(True)  # Same as `x = torch.arange(4.0, requires_grad=True)`
    print(x.grad)  # The default value is None
```

```
None
```

2. Calculate any intermediate and final results

```
[9] y = 2 * torch.dot(x, x) # dot product between vectors
    y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

3. Tell PyTorch to perform the backpropagation on the computational graph

```
[10] y.backward()
```

4. Access the gradient of y w.r.t x:

```
[11] x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

- Gradient of $y = 2\mathbf{x}^\top \mathbf{x}$ with respect to $\mathbf{x}$ should be $4\mathbf{x}$.
  Let us quickly verify that our desired gradient was calculated correctly.

```
[14] x.grad == 4 * x
```

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

Example: let us follow with another example: we want the gradient of $y = \sum_i x_i$, which is the vector of 1s.

- By default, PyTorch *accumulates* the gradient when calling `.backward()`, for instance after some other computations to get `y` from `x`

```
[15] #x.grad.zero_()
     y = x.sum() # because of the accumulation, the vector 1 1 1 ... sums and we obtain this
     y.backward()
     x.grad
```

```
tensor([ 2.,  6., 10., 14.])
```

- therefore... we need to **reset** the gradient (with in-place method `zero_()`) before other computations and call to `.backward()`:

```
[16] x.grad.zero_()
     y = x.sum()
     y.backward()
     x.grad
```

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

Backward for Non-Scalar Variables

- Differentiating a vector y w.r.t. a vector x: we get a matrix.
- If x or y are higher-dimensional: we even get higher-dimensional tensor

But most times, we need to differentiate the **loss** functions for each constituent of a *batch* (a lot) of training examples.

- that is, we usually want the *sum of the partial derivatives* computed individually for each example in the batch.

```
[ ]  # Invoking `backward` on a non-scalar requires passing in a `gradient` argument
     # which specifies the gradient of the differentiated function w.r.t `self`.
     # In our case, we simply want to sum the partial derivatives, so passing
     # in a gradient of ones is appropriate
     x.grad.zero_()
     y = x * x
     # y.backward(torch.ones(len(x))) equivalent to the below
     y.sum().backward()
     x.grad
```

```
tensor([0., 2., 4., 6.])
```

Detaching Computation

Sometimes, we need to differentiate a "complicately" composite function w.r.t. some arguments only, treating the others as constant.

The composite function corresponds, as usual, to some piece of code

- telling PyTorch to regard some variables as constant consists in making it **detach** the variables from the computational graph

Example: given $y = x^2, z = f(x, y) := yx$ we want to compute

$$\frac{\partial f}{\partial x}(x, y)$$

After writing the proper code, if we simply backpropagated from `z`, we would obttain the total derivative (stored in `x`)

$$\frac{\partial f}{\partial x}(x, y) + \frac{\partial f}{\partial y}(x, y)\frac{\partial y}{\partial x}(x)$$

- Instead, we need to detach `y` from the computation graph

Below, we can detach `y` to return a new variable `u` that has the same value as `y` but discards any information about how `y` was computed in the computational graph. In other words, the gradient will not flow backwards through `u` to `x`.

```
[17] x.grad.zero_()
     y = x * x
     u = y.detach()
     z = u * x

     z.sum().backward()
     x.grad == u
```

```
tensor([True, True, True, True])
```

We can check which tensors are in the computational graph, and which ones were detached or were initialized with no gradient information required, by means of `requires_grad` attribute

```
[18] print(x.requires_grad)
     print(y.requires_grad)
     print(u.requires_grad)
     print(z.requires_grad)
```

```
True
True
False
True
```

Anyway, we can invoke backpropagation on `y` to get the derivative of `y = x * x` with respect to `x`, which is `2 * x`.

```
[19] x.grad.zero_()
     y.sum().backward()
     x.grad == 2 * x
```

```
tensor([True, True, True, True])
```

Computing the Gradient of Python Control Flow

Useful capability of automatic differentiation:

- PyTorch is able to build the computational graph of complicate maze of code for computing some result from some data, for example even through the control flows: conditional statements, loops, arbitrary function calls, ...

Eg.: in the function `f` both the while-loop and the if-statement depend on `a`

```
[20] def f(a):
         b = a * 2
         while b.norm() < 1000:
             b = b * 2
         if b.sum() > 0:
             c = b
         else:
             c = 100 * b
         return c
```

Let us compute the gradient.

```
[ ] # torch.randn() return a random number following the Gauss distribution with
    # mean=0 and stdev=1
    # with size=(), return a 'float'
    # with size=(1,), return a '[float]'
    # with size=(1), ERROR
    a = torch.randn(size=(), requires_grad=True)
    print(a)
    d = f(a)
    d.backward()
```

```
tensor(-0.0632, requires_grad=True)
```

We can now analyze the `f` function defined above. Note that it is piecewise linear in its input `a`.

```
[ ] a.grad
```

```
tensor(1638400.)
```

Exercises

1. After running the function for backpropagation, immediately run it again and see what happens.
2. Redesign an example of finding the gradient of the control flow. Run and analyze the result.
3. Let $f(x) = \sin(x)$. Plot $f(x)$ and $\frac{df(x)}{dx}$, where the latter is computed without exploiting that $f'(x) = \cos(x)$.

Solution 1)

```
[ ] print(a.grad)
    d.backward()
    print(a.grad)
```

```
    tensor(1638400.)
    tensor(1638400.)
```

Solution 2)

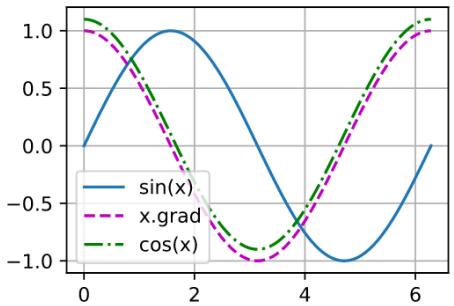
```
[ ] def f(x):
        if torch.all(x > 0):
            return x+f(x-1)
        else:
            return x*x

    x=torch.arange(2.,10., requires_grad=True)
    y=f(x)
    #for i in range(10):
    #    y[i]=f(x[i])
    print(x)
    print(y)
    y.backward(torch.ones(len(x)))
    #for i in range(10):
    #    y[i].backward(torch.ones(len(x)), retain_graph = True)
    print(x.grad)
```

```
    tensor([2., 3., 4., 5., 6., 7., 8., 9.], requires_grad=True)
    tensor([ 3.,  6., 11., 18., 27., 38., 51., 66.], grad_fn=<AddBackward0>)
    tensor([ 2.,  4.,  6.,  8., 10., 12., 14., 16.])
```

Solution 3)

```
[ ] !pip install d2l
```

```
[ ] import numpy as np
    from d2l import torch as d2l
    n=100
    x=np.linspace(0,2*np.pi,n)
    x=torch.tensor(x, requires_grad=True)
    y=torch.sin(x)
    z=torch.cos(x) + 0.1 # little 'shift' for visibility
    #print(x)
    #print(y)
    for i in range(n):
        y[i].backward(retain_graph = True)
    d2l.plot(x.detach(),(y.detach(),x.grad,z.detach()),legend = (('sin(x)','x.grad','cos(x)')))
```



Concise Implementation of Linear Regression

- Goal: implement the linear regression model concisely by using high-level APIs of deep learning frameworks
- Modern libraries implement useful "routine" components: data iterators, loss functions, optimizers, and neural network layers

Preliminaries:

```
pip install d2l
```

```
Collecting d2l
  Downloading d2l-0.17.0-py3-none-any.whl (83 kB)
     |████████████████████████████████| 83 kB 1.3 MB/s
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (from d2l) (1.1.5
```

```
[2] import numpy as np
    import torch
    import torchvision
    from PIL import Image
    from torch import nn
    from torch.nn import functional as F
    from torch.utils import data
    from torchvision import transforms
```

```
[3] import collections
    import hashlib
    import math
    import os
    import random
    import re
    import shutil
    import sys
    import tarfile
    import time
    import zipfile
    from collections import defaultdict
    import pandas as pd
    import requests
    from IPython import display
    from matplotlib import pyplot as plt
    d2l = sys.modules[__name__]

    from d2l import torch as d2l
```

Generating the Dataset

To generate the linear regression dataset, `d2l` has this service function:

```
[4] def synthetic_data(w, b, num_examples):
        """Generate y = Xw + b + noise."""
        X = torch.normal(0, 1, (num_examples, len(w)))
        y = torch.matmul(X, w) + b
        y += torch.normal(0, 0.01, y.shape)
        return X, y.reshape((-1, 1))
```

- Start generating a dataset for linear regression:

```
[5] true_w = torch.tensor([2, -3.4])
    true_b = 4.2
    features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

Reading the Dataset

`d2l` has a function based on `data` module that: takes the dataset and batch size β, and returns an iterable for passing through the dataset

- `is_train` boolean argument indicates whether the iterator must shuffle the data on every epoch (unneeded for test phase)

```
[6] def load_array(data_arrays, batch_size, is_train=True):
        """Construct a PyTorch data iterable."""
        dataset = data.TensorDataset(*data_arrays)
        return data.DataLoader(dataset, batch_size, shuffle=is_train)
```

```
[7] batch_size = 10
    data_iter = d2l.load_array((features, labels), batch_size)
```

- Let us construct the iterator and check out the first data example

```
[8] next(iter(data_iter))
```

```
    [tensor([[-1.3431,  0.3771],
             [ 0.3778, -0.4004],
             [ 2.3479,  0.6928],
             [-0.8975,  0.1257],
             [ 1.4191, -1.5062],
             [-0.4659, -0.2816],
             [-1.3680,  1.5065],
             [ 0.7027, -1.3651],
             [-0.3450,  0.5969],
             [-2.4439, -0.4236]]), tensor([[ 0.2259],
             [ 6.3303],
             [ 6.5373],
             [ 1.9789],
             [12.1595],
             [ 4.2368],
             [-3.6516],
             [10.2341],
             [ 1.4610],
             [ 0.7586]])]
```

Defining the Model

- A model is a sequence of layers.
    - `Sequential` is a class that defines a **container** for several layers, chained together: outputs are wired to inputs of successive layer
        - just list the layers as arguments
    - `Linear` class is used in PyTorch to define a linear or fully-connected **layer**
        - 2 arguments: input feature dimension, output feature dimension

Continuing previous example, let us define a network for the linear model, i.e. a 1 layer NN

- the `Sequential` container is not really needed, but will always be used later on

```
[9] net = nn.Sequential(nn.Linear(2, 1))
```

- the assigned object represents a NN: can be indexed to get the layers
    - `net[0]` locate the layer of interest (zero-based index): first layer in this case

Initializing Model Parameters

- `weight` and `bias` attributes represent the parameters
  - both have `.data` attribute to access the value(s)

Deep learning frameworks often have a predefined way to initialize the parameters.

Here, we initialize the parameters

- $w_i \sim N(0, 0.01^2)$ for all i independently
- $b = 0$

```
print(net[0].weight.data.normal_(0, 0.01))  # each layer have a parameter
print(net[0].bias.data.fill_(0)) # fills each element of the bayes paramter with 0
```

```
tensor([[ 0.0104, -0.0107]])
tensor([0.])
```

Defining the Loss Function

The `MSELoss` class computes the mean squared error, much related to squared L_2 norm.

- it refers to formula with the average loss over examples by default: behaviour can be changed

```
[11] loss = nn.MSELoss()
```

Defining the Optimization Algorithm

PyTorch has several algorithm variations for optimizing NNs, in the `optim` module

- an `SGD` class instance represents the SGD optimizer algorithm: it must know the network parameters and any required hyperparameter's value
  - `net.parameters()`
  - learning rate for SGD is required

```
[12] trainer = torch.optim.SGD(net.parameters(), lr=0.03)
```

Training

- For some number of epochs:
  - make a complete pass over the dataset (`train_data`), iteratively grabbing one minibatch of inputs and labels.
  - For each minibatch
    - Generate predictions by calling `net(X)` and calculate the loss `l` (the forward computation).
    - Calculate gradients by running the backpropagation.
    - Update the model parameters by invoking our optimizer.

```
[13] num_epochs = 3
     for epoch in range(num_epochs):
         for X, y in data_iter:
             l = loss(net(X) ,y)
             trainer.zero_grad() # To avoid accumulating the gradients on subsequent backward passes
             l.backward()
             trainer.step() # explicitly perform one iteration
         l = loss(net(features), labels)
         print(f'epoch {epoch + 1}, loss {l:f}')
```

```
epoch 1, loss 0.000355
epoch 2, loss 0.000104
epoch 3, loss 0.000104
```

Comparison between learnt and actual parameters

For post-hoc check, we compare the model parameters learned by training on finite data and the actual (ground-truth) parameters that generated our synthetic dataset.

- first access the layer by indexing `net`, then access layer's parameters

```
[14] w = net[0].weight.data
     print('error in estimating w:', true_w - w.reshape(true_w.shape))
     b = net[0].bias.data
     print('error in estimating b:', true_b - b)
```

```
     error in estimating w: tensor([-0.0007,  0.0001])
     error in estimating b: tensor([-0.0007])
```

Summary

- Using PyTorch's high-level APIs, we can implement models much more concisely.
- In PyTorch, the `data` module provides tools for data processing, the `nn` module defines a large number of neural network layers and common loss functions.
- We can initialize the parameters by replacing their values with methods ending with `_`.

Exercises

1. Review the PyTorch documentation to see what loss functions and initialization methods are provided. Replace the loss by Huber's loss.
2. How do you access the gradient of `net[0].weight`?

Solution

1. Search PyTorch documentation you can find the HuberLoss function. So, replace:

```
[15] loss = nn.HuberLoss()
```

2. By the code snippet:

```
[16] net[0].weight.grad
```

```
     tensor([[-0.0001, -0.0059]])
```

The Image Classification Dataset

Today, well-known **MNIST** dataset serves as more of sanity checks than as a benchmark.

Qualitatively similar, but comparatively more complex **Fashion-MNIST** dataset, released in 2017.

```
[ ] pip install d2l
```

```
     |████████████████████████████████| 85 kB 1.1 MB/s
    Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (from d2l) (1.1.5)
    Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from d2l) (3.2.2
    Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from d2l) (1.19.5)
    Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-packages (from d2l) (1.0.0)
    Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from d2l) (2.23.0)
```

```
[ ] %matplotlib inline
    from d2l import torch as d2l
    import torch
    import torchvision
    from torchvision import transforms
    from torch.utils import data

    d2l.use_svg_display()
```

Reading the Dataset

We can download and read the Fashion-MNIST dataset **into memory** via the build-in functions in the framework.

```
[ ] # `ToTensor` converts the image data from PIL (Python Image Library) type to 32-bit floating point
    # tensors. It divides all numbers by 255 so that all pixel values are between
    # 0 and 1
    trans = transforms.ToTensor()
    mnist_train = torchvision.datasets.FashionMNIST(
        root="../data", train=True, transform=trans, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=True)
```

```
    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ../
```

Fashion-MNIST consists of:

- 10 categories
- 6000 images in training dataset for each category
- 1000 images in test dataset for each category

A **test dataset** (or **test set**) is used for evaluating model performance and not for training.

Consequently: 60000 training and 10000 test images, respectively.

```
[ ] len(mnist_train), len(mnist_test)
```

```
    (60000, 10000)
```

A digital image is a rectangular regular grid whose points are **pixels**:

- (row, column) **position**
- for B/W images, namely **greyscale** images: a fractional value in $[0, 1]$ or integer value in $\{0, \dots, 255\}$ representing the **intensity of grey** (0=black). Other colours are not represented: 1 channel only

Fashion-MNIST's images: shape is $height \times width = 28 \times 28$
$h \times w : (h, w)$

```
[ ] mnist_train[0][0].shape
```

```
    torch.Size([1, 28, 28])
```

10 **Categories**: t-shirt, trousers, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot.

- Function for converting between numeric label **indices** and their **names** in text.

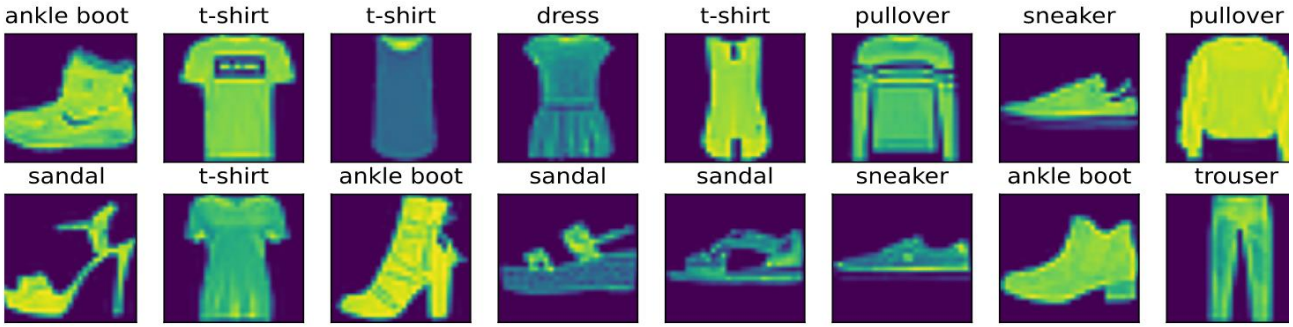
```
[ ]  def get_fashion_mnist_labels(labels):
         """Return text labels for the Fashion-MNIST dataset."""
         text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                        'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
         return [text_labels[int(i)] for i in labels]
```

- Function for visualizing these examples

```
[ ]  def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
         """Plot a list of images."""
         figsize = (num_cols * scale, num_rows * scale)
         _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
         axes = axes.flatten()
         for i, (ax, img) in enumerate(zip(axes, imgs)):
             if torch.is_tensor(img):
                 # Tensor Image
                 ax.imshow(img.numpy())
             else:
                 # PIL Image
                 ax.imshow(img)
             ax.axes.get_xaxis().set_visible(False)
             ax.axes.get_yaxis().set_visible(False)
             if titles:
                 ax.set_title(titles[i])
         return axes
```

Here are the images and their corresponding labels (in text) for the first few examples in the training dataset.

```
[ ]  X, y = next(iter(data.DataLoader(mnist_train, batch_size=18)))
     show_images(X.reshape(18, 28, 28), 2, 9, titles=get_fashion_mnist_labels(y));
```



Reading a Minibatch

We use the built-in data iterator.

- at each iteration, a data loader reads a **minibatch** of data with size `batch_size`
- we also randomly **shuffle** the examples for the training data iterator

```
[ ]  batch_size = 256

     def get_dataloader_workers():
         """Use 2 processes to read the data."""
         return 2

     train_iter = data.DataLoader(mnist_train, batch_size, shuffle=True,
                                  num_workers=get_dataloader_workers())
```

Let us look at the time it takes to read the training data.

```
[ ]  timer = d2l.Timer()
     for X, y in train_iter:
         continue
     f'{timer.stop():.2f} sec'
```

```
torch.Size([32, 1, 64, 64]) torch.float32 torch.Size([32]) torch.int64
```

Putting All Things Together

Define `load_data_fashion_mnist` function:

- obtains and reads the Fashion-MNIST dataset
- returns data iterators for both training set and validation set
- optional argument to resize images to another shape

```
[ ]  def load_data_fashion_mnist(batch_size, resize=None):
         """Download the Fashion-MNIST dataset and then load it into memory."""
         trans = [transforms.ToTensor()]
         if resize:
             trans.insert(0, transforms.Resize(resize))
         trans = transforms.Compose(trans)
         mnist_train = torchvision.datasets.FashionMNIST(
             root="../data", train=True, transform=trans, download=True)
         mnist_test = torchvision.datasets.FashionMNIST(
             root="../data", train=False, transform=trans, download=True)
         return (data.DataLoader(mnist_train, batch_size, shuffle=True,
                                 num_workers=get_dataloader_workers()),
                 data.DataLoader(mnist_test, batch_size, shuffle=False,
                                 num_workers=get_dataloader_workers()))
```

We want to test the image resizing feature

```
[ ]  train_iter, test_iter = load_data_fashion_mnist(32, resize=64)
     for X, y in train_iter:
         print(X.shape, X.dtype, y.shape, y.dtype)
         break
```

```
torch.Size([32, 1, 64, 64]) torch.float32 torch.Size([32]) torch.int64
```

We are now ready to work with the Fashion-MNIST dataset.

Summary

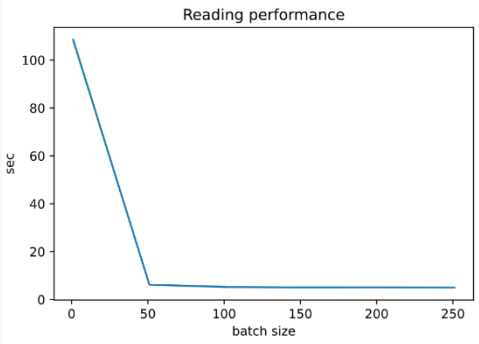
- Fashion-MNIST is an apparel classification dataset consisting of images representing 10 categories. We will use this dataset in subsequent sections and chapters to evaluate various classification algorithms.
- We store the shape of any image with height h width w pixels as $h \times w$ or (h, w).
- Data iterators are a key component for efficient performance. Rely on well-implemented data iterators that exploit high-performance computing to avoid slowing down your training loop.

Exercises

1. Does reducing the `batch_size` (for instance, to 1) affect the reading performance? Try in the range 1 - 256, with step 50, and plot the corresponding reading time

1. A possible solution using the `d2l.Timer` class:

```
[13] from matplotlib import pyplot as plt

     T = list()
     batch_size = range(1, 256, 50)

     for b in batch_size:
         train_iter = data.DataLoader(mnist_train, b, shuffle=True,
                                      num_workers=get_dataloader_workers())
         timer = d2l.Timer()
         for X, y in train_iter:
             continue
         T.append(timer.stop())

     plt.plot(batch_size, T)
     plt.xlabel('batch size')
     plt.ylabel('sec')
     plt.title('Reading performance')
     plt.show()
```



Concise Implementation of Softmax Regression

on Fashion_MNIST dataset

Choices

- batch size: 256
- softmax regression (one fully-connected layer with softmax activation function)
- evaluate by Cross Entropy loss

```
[1] pip install d2l
```

```
Collecting d2l
  Downloading d2l-0.17.0-py3-none-any.whl (83 kB)
     |████████████████████████████████| 83 kB 667 kB/s
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from d2l) (2.23.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from d2l) (1.19.5)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from d2l) (3.2.2)
```

```
[2] from d2l import torch as d2l
    import torch
    from torch import nn
    from torch.utils import data
```

```
[3] batch_size = 256
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz to ../data/
26422272/? [00:01<00:00, 19144061.88it/s]
```

Initializing Model

In this Fashion-MNIST case: softmax regression with one fully-connected layer

- 10 outputs
- again, `Sequential` is not really needed
- we initialize weights w_i i.i.d. $N(0, 0.01^2)$

Moreover:

- an image is a matrix
- NN multivariate input is a vector
  $\Rightarrow$ need for "flattening" the matrices to vectors

```
[4] # PyTorch does not implicitly reshape the inputs. Thus we define the "flatten"
    # layer to reshape the inputs before the linear layer in our network
    net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))

    def init_weights(m): # argument m shall be a layer
        if type(m) == nn.Linear:
            nn.init.normal_(m.weight, std=0.01)

    net.apply(init_weights); # apply() will call init_weights() providing it with each layer of net as actual
                             # argument m
```

Cross Entropy loss

WARNING: numerical stability

Exponentiation (and several other operations) can be a source of **numerical stability** issues: due to numbers stored in **finite** precision and operations performed with **approximations** by electronic calculators.

$\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$

- If some $\exp(o_k)$ is too large positive, it could exceed machine encoding's range (*overflow* problem): denominator (and/or numerator) blows up to `inf` (infinity) and we wind up encountering either 0, `inf`, or `nan` (not a number) for $\hat{y}_j$.
  - not a well-defined return value for cross-entropy
  - we could think of subtracting $\max(o_k)$: can cause next issue
- Analogously, $\exp(o_j)$ may be too small: rounded to 0 (*underflow* problem), giving `-inf` for $\log(\hat{y}_j)$
  - few steps down the road in backpropagation, we might face `nan` results.

We are saved somehow: we ultimately want logarithm (when calculating the cross-entropy loss).

Combining softmax and cross-entropy: we avoid calculating $\exp(o_j)$, use o_j directly due to the canceling in $\log(\exp(\cdot))$.

$$\begin{aligned}\log\left(\hat{y}_j\right) &= \log\left(\frac{\exp(o_j)}{\sum_k \exp(o_k)}\right) \\ &= \log\left(\exp(o_j)\right) - \log\left(\sum_k \exp(o_k)\right) \\ &= o_j - \log\left(\sum_k \exp(o_k)\right).\end{aligned}$$

We will want to keep the conventional softmax function handy in case we ever want to evaluate the output probabilities by our model. But instead of passing softmax probabilities into our new loss function, we will just pass the **logits** and compute the softmax and its log all at once inside the cross-entropy loss function, which does smart things like the "LogSumExp trick".

Optimization Algorithm

Minibatch SGD:

- learning rate 0.1

```
[6]  trainer = torch.optim.SGD(net.parameters(), lr=0.1)
```

Training

First, we define a function to **train** for **one epoch**.

Note that `updater` is a general function to update the model parameters, which accepts the batch size as an argument. It can be either a wrapper of the `d2l.sgd` function or a framework's built-in optimization function.

```
[7]  def train_epoch_ch3(net, train_iter, loss, updater):
         """The training loop defined in Chapter 3."""
         # Set the model to training mode
         if isinstance(net, torch.nn.Module):
             net.train()
         # Sum of training loss, sum of training accuracy, nb. of examples
         metric = d2l.Accumulator(3)
         for X, y in train_iter:
             # Compute gradients and update parameters
             y_hat = net(X)
             l = loss(y_hat, y)
             updater.zero_grad()
             l.backward()
             updater.step()
             metric.add(float(l) * len(y), d2l.accuracy(y_hat, y), y.size().numel())
         # Return training loss and training accuracy
         return metric[0] / metric[2], metric[1] / metric[2]
```

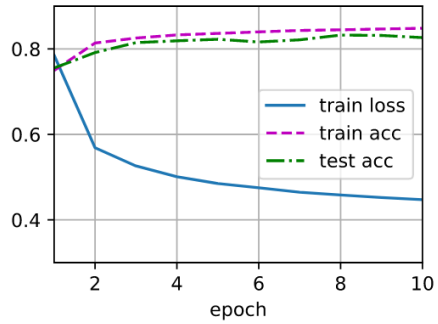
The following training function then trains a model `net` on a training dataset accessed via `train_iter` for **multiple epochs**, which is specified by `num_epochs`. At the end of each epoch, the model is evaluated on a testing dataset accessed via `test_iter`. We will leverage the `Animator` class to visualize the training progress.

```
[8]  def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):
         """Train a model (defined in Chapter 3)."""
         animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                                 legend=['train loss', 'train acc', 'test acc'])
         for epoch in range(num_epochs):
             train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
             test_acc = d2l.evaluate_accuracy(net, test_iter)
             animator.add(epoch + 1, train_metrics + (test_acc,))
         train_loss, train_acc = train_metrics
         return train_loss, train_acc
```

Next we call the training function to train the model.

```
[9]  num_epochs = 10
     train_loss, train_acc = train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
     print("Loss: ", train_loss)
     print("Accuracy: ", train_acc)
```

```
Loss:  0.4473460349559784
Accuracy:  0.84825
```



This algorithm converges to a solution that achieves a decent accuracy:

- classification **accuracy**: fraction of all predictions that are correct

Summary

- Using high-level APIs, we can implement softmax regression much more concisely.
- From a computational perspective, implementing softmax regression has intricacies. Note that in many cases, a deep learning framework takes additional precautions beyond these most well-known tricks to ensure numerical stability, saving us from even more pitfalls that we would encounter if we tried to code all of our models from scratch in practice.

Exercises

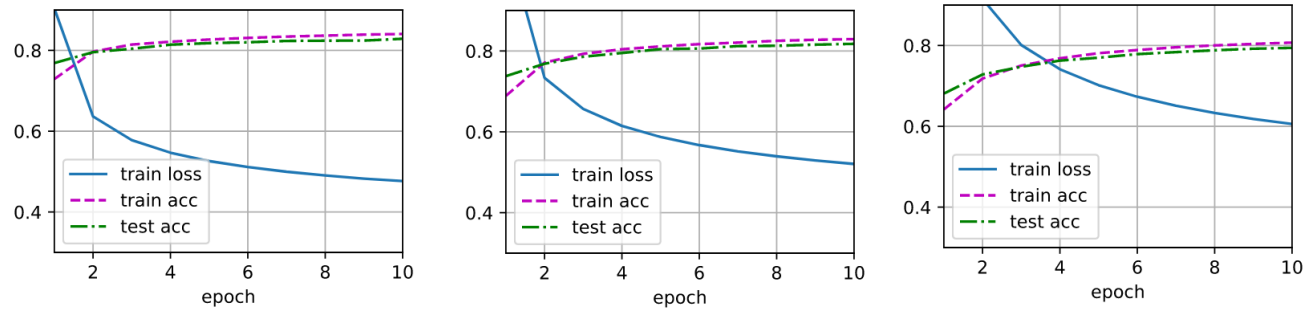
1. Try adjusting the hyperparameters, such as the batch size [50, 100, 200, 256 *], number of epochs [1, 10 * , 30, 50] , and learning rate [0.01 *, 0.1, 0.3], and plot the accuracy vs the hyperparameters (ome plot for each hyperparameter) to highlight the best. When exploring one parameter, fix the others setting the value with the *.
2. Why might the test accuracy decrease after a while, increasing the number of epochs for training? How could we fix this?

Solutions

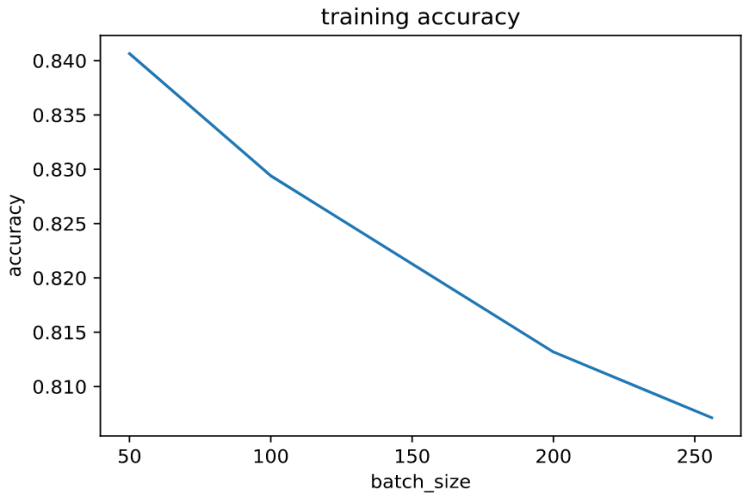
1. In this possible solution we can set the 3 hyperparameters to single value or a list of values depending on which one we want to explore, by simply changing the initial assignments

```
[10] %matplotlib inline

     batch_size = [50, 100, 200, 256]        # [50, 100, 200, 256]
     learning_rate = [0.01]           # [0.01 , 0.1, 0.3]
     epochs = [10]        # [1, 10, 30, 50]
     acc = list()

     loss = nn.CrossEntropyLoss()

     for b in batch_size:
         for l in learning_rate:
             for e in epochs:
                 print(f'batch_size: {b} - learning_rate: {l} - epochs {e}')
                 train_iter, test_iter = d2l.load_data_fashion_mnist(b)
                 net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))

                 net.apply(init_weights);
                 trainer = torch.optim.SGD(net.parameters(), lr=l)

                 train_loss, train_acc = train_ch3(net, train_iter, test_iter, loss, e, trainer)
                 print("Loss: ", train_loss)
                 print("Accuracy: ", train_acc)
                 acc.append(train_acc)
```

```
Loss:  0.6057078352292379
Accuracy:  0.8071333333333334
```




```
[11] from matplotlib import pyplot as plt
     plt.plot(batch_size, acc)
     plt.xlabel('batch_size')
     plt.ylabel('accuracy')
     plt.title('training accuracy')
     plt.show()
```



2. Training by more epochs might yield a model with very low training loss value, indicating a tight fit with the training set, while not being able to generalize to unseen data as in the test set. That is, we can incur overfitting when the model is too complex.

Concise Implementation of Multilayer Perceptrons

By relying on the high-level APIs of PyTorch, we can implement MLPs even more concisely.

```
[1] pip install d2l
```

```
[2] from d2l import torch as d2l
    import torch
    from torch import nn
```

Model

We will build a very simple NN with 1 hidden layer:

- inputs are 28 x 28 images, so: need to flatten
- ReLU as activation function (no softmax usually used on output layer)

```
[3] net = nn.Sequential(nn.Flatten(),
                        nn.Linear(784, 256),
                        nn.ReLU(),
                        nn.Linear(256, 10))

    def init_weights(m):
        if type(m) == nn.Linear:
            nn.init.normal_(m.weight, std=0.01)

    net.apply(init_weights);
```

Loss and optomization functions

The **training** loop: same as when we implemented softmax regression.

Code modularity: reuse

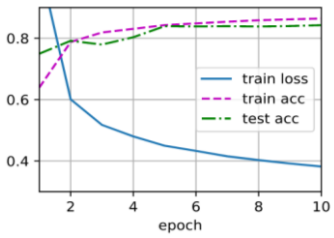
```
[4] batch_size, lr, num_epochs = 256, 0.1, 10
    loss = nn.CrossEntropyLoss()
    trainer = torch.optim.SGD(net.parameters(), lr=lr)
```

Training

```
[5] train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

```
    Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
```

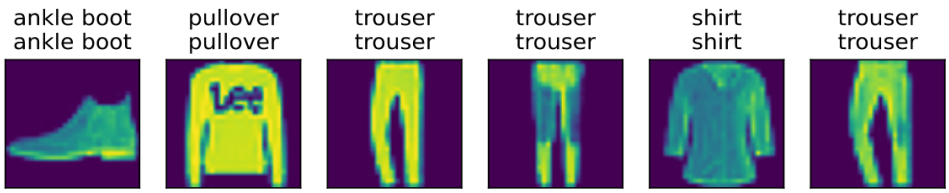
```
[6] d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Test phase

```
[7] d2l.predict_ch3(net, test_iter)
```

```
    /usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481: UserWarning: This DataLoader will
      cpuset_checked))
```



The function `d2l.predict_ch` has the following easily intelligible definition:

```
[8]  def predict_ch3(net, test_iter, n=6):
         """Predict labels (defined in Chapter 3)."""
         for X, y in test_iter:
             break
         trues = d2l.get_fashion_mnist_labels(y)
         preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
         titles = [true + '\n' + pred for true, pred in zip(trues, preds)]
         d2l.show_images(X[0:n].reshape((n, 28, 28)), 1, n, titles=titles[0:n])
```

Summary

- Using high-level APIs, we can implement MLPs much more concisely.
- For the same classification problem, the implementation of an MLP is the same as that of softmax regression except for additional hidden layers with activation functions.

Exercises

1. Try adding different numbers of hidden layers (you may also modify the learning rate). What setting works best?
2. Try out different activation functions. Which one works best?
3. Try different schemes for initializing the weights. What method works best?
4. Try differen optimization algoritms (https://pytorch.org/docs/stable/optim.html)

Solutions

1. For example adding one more hidden layer basically consists in the code snippet for the new network `net1`

```
[ ]  net1 = nn.Sequential(nn.Flatten(),
                          nn.Linear(784, 256),
                          nn.ReLU(),
                          nn.Linear(256, 128),
                          nn.ReLU(),
                          nn.Linear(128, 10))
```

2. We could change the activation function, choosing among: https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

```
[ ]  net2 = nn.Sequential(nn.Flatten(),
                          nn.Linear(784, 256),
                          nn.Sigmoid(),
                          nn.Linear(256, 10))
```

3. You can change the initialization step choosing from: https://pytorch.org/docs/stable/nn.init.html. Plausible examples:
    - `nn.init.uniform_(m.weight)`
    - `nn.init.constant_(m.weight, 0.3)`
    - `torch.nn.init.ones_(m.weight)`
    - `torch.nn.init.zeros_(tensor)`

4. This is a bit beyond our current topic coverage, but you can look ahead to some of those algorithms at https://pytorch.org/docs/stable/optim.html
   Another common optimization algorithm is Adagrad:

```
[ ]  trainer = torch.optim.Adagrad(net.parameters(), lr=0.01, lr_decay=0, weight_decay=0, initial_accumulator_value=
```

Dropout

MLP before and after dropout.

```
[1]  !pip install d2l torch
```

Overfitting Revisited

Traditional treatment: tension between generalizability and flexibility described as the *bias-variance tradeoff*

- linear models have high bias, low variance: they can represent a small class of functions, give similar results across different random samples of the data

Robustness through Perturbations

- smoothness as simplicity: i.e. model should not be sensitive to small changes to its inputs
  - e.g. image classification: expect that adding some random noise to image pixels should be mostly harmless
- from input noise to internal layer's noise:
  - Bishop 1995: training with input noise is equivalent to Tikhonov regularization
  - Srivastava, Hinton et al. 2014: inject **noise** into **internal layers** too

Droput: they realized that when training a deep network, injecting noise **enforces smoothness** just on the input-output mapping

- throughout training iterations: apply zeroing out some fraction of the nodes in each layer before calculating the subsequent layer
  - in an *unbiased* manner: preserving the expected value of each layer

Fixing *dropout probability* p: each intermediate activation x is replaced by a random variable x' as follows:

$$x' = \begin{cases} 0 & \text{with probability } p \\ \frac{x}{1-p} & \text{with probability } 1-p \end{cases}$$

$\Rightarrow E[x'] = x$.

Dropout in Practice

- performed at each iteration
- typically, no dropout at inference time: given a trained model and a new example, we do not drop out any nodes

Implementation of Dropout function from Scratch

Straight-forward:
if X is the vector of activations

- for every i sample a "mask" value i.i.d. from Bernoulli$(1-p)$, to be multiplied with X_i
  - equivalently: $U_i \sim$ Uniform$(0,1)$, then the mask value is $\mathbf{1}(U_i > 1-p)$
- rescale by $1/(1-p)$

Code for implementing dropout function:

```
[2]  from d2l import torch as d2l
     import torch
     from torch import nn

     def dropout_layer(X, dropout_prob):
         assert 0 <= dropout_prob <= 1  # "assert" raises an exception if the given condition is false
         # Shortcut case: all elements are dropped out
         if dropout_prob == 1:
             return torch.zeros_like(X)
         # Shortcut case: all elements are kept
         if dropout_prob == 0:
             return X
         mask = (torch.rand_like(X) > dropout_prob).float()
         return mask * X / (1.0 - dropout_prob)
```

Let us test it with dropout probabilities: 0, 0.5, and 1

```
[3] X= torch.arange(16, dtype = torch.float32).reshape((2, 8))
    print(X)
    print(dropout_layer(X, 0.))
    print(dropout_layer(X, 0.5))
    print(dropout_layer(X, 1.))
```

```
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  0.,  0.,  6.,  8.,  0., 12., 14.],
        [ 0.,  0., 20., 22.,  0., 26.,  0., 30.]])
tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

Using PyTorch's interface

In place of "handmade" dropout function, we can use the constructor in PyTorch high-level API: `Dropout(p)`, p being the dropout probability

Example: let us construct an MLP working on Fashion-MNIST dataset

- input: 28 x 28 pixels (to be "flattened" of course)
- 2 hidden layers: 256 units each, ReLU activation function
    - dropout with a dropout probability for each layer
- output: 10 (for 10 clothing categories), fully-connected
- classification as before: take argmax, so output can be interpreted just as a "score" indicating belongingness to each category

1. Defining Model Parameters:

```
[4] num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
    dropout_p1, dropout_p2 = 0.2, 0.5
```

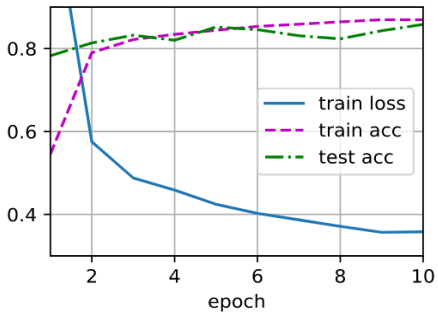
- add a `Dropout` "layer" after each fully-connected layer,
- in training mode the `Dropout` layer will randomly drop out its inputs, namely outputs of the previous layer
    - in evaluation mode: the `Dropout` layer simply passes all the values through
    - `.train(True)` or `.train(False)` method for switching mode (see `d2l.train_epoch_ch3()` implementation)

```
[5] net = nn.Sequential(nn.Flatten(),
            nn.Linear(784, 256),
            nn.ReLU(),
            # Add a dropout layer after the first fully connected layer
            nn.Dropout(dropout_p1),
            nn.Linear(256, 256),
            nn.ReLU(),
            # Add a dropout layer after the second fully connected layer
            nn.Dropout(dropout_p2),
            nn.Linear(256, 10))

    def init_weights(m):
        if type(m) == nn.Linear:
            nn.init.normal_(m.weight, std=0.01)

    net.apply(init_weights);
```

2. Training and Testing

Let us train the model with SGD and progressively evalute on the test

```
[6]  num_epochs, lr, batch_size = 10, 0.5, 256
     loss = nn.CrossEntropyLoss()
     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
     trainer = torch.optim.SGD(net.parameters(), lr=lr)
     d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Summary

- Beyond controlling the number of dimensions and the size of the weight vector, dropout is yet another tool to avoid overfitting. Often they are used jointly.
- Dropout replaces an activation h with a random variable with expected value h.
- Dropout is only used during training.

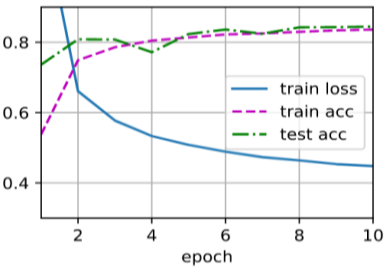
Exercises

1. What happens if you set dropout_p1, dropout_p2 = 0.7, 0.6? Summarize the qualitative takeaways.
2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.
3. Why is dropout not typically used at test time?

Solutions

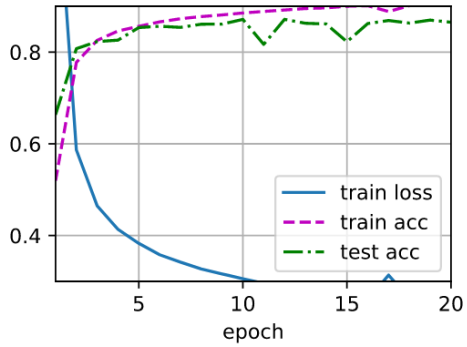
1. The generalization ability increases, showing a slightly better test accuracy than train accuracy

```
[ ]  num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
     dropout_p1, dropout_p2 = 0.7, 0.6
     net = nn.Sequential(nn.Flatten(),
             nn.Linear(784, 256),
             nn.ReLU(),
             # Add a dropout layer after the first fully connected layer
             nn.Dropout(dropout_p1),
             nn.Linear(256, 256),
             nn.ReLU(),
             # Add a dropout layer after the second fully connected layer
             nn.Dropout(dropout_p2),
             nn.Linear(256, 10))

     net.apply(init_weights);

     num_epochs, lr, batch_size = 10, 0.5, 256
     loss = nn.CrossEntropyLoss()
     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
     trainer = torch.optim.SGD(net.parameters(), lr=lr)
     d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



2. Set dropout probabilities to 0

```
[ ]  num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
     dropout_p1, dropout_p2 = 0, 0
     net = nn.Sequential(nn.Flatten(),
             nn.Linear(784, 256),
             nn.ReLU(),
             # Add a dropout layer after the first fully connected layer
             nn.Dropout(dropout_p1),
             nn.Linear(256, 256),
             nn.ReLU(),
             # Add a dropout layer after the second fully connected layer
             nn.Dropout(dropout_p2),
             nn.Linear(256, 10))

     net.apply(init_weights);

     num_epochs, lr, batch_size = 20, 0.5, 256
     loss = nn.CrossEntropyLoss()
     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
     trainer = torch.optim.SGD(net.parameters(), lr=lr)
     d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



3. Typically we want the output to be deterministic w.r.t. input at test time, except for very particular situations where researchers want to carry out certain experimental analysis.

Convolutions for Images

```
[ ]  !pip install torch d2l
```

Now that we understand how convolutional layers work in theory, we are ready to see how they work in practice. Building on our motivation of convolutional neural networks as efficient architectures for exploring structure in image data, we stick with **images** as our running example.

Two-dimensional Cross-correlation Operation

- Though the proper name is *cross-correlation*, it is commonly called *convolution*

We want to write a `corr2d` function implementing this operation.
Needed elements:

- input tensor `X` (a greyscale image) with shape: $n_h \times n_w$
- kernel tensor `K` for the convolution, with window: $k_h \times k_w$
    - of course its shape must fit within the image
- $\Rightarrow$ output tensor `Y` with shape: $(n_h - k_h + 1) \times (n_w - k_w + 1)$
- The formula representing the cross-correlation between two matrices X and K is easily deducible from the class explanation:

$$Y[i,j] = \sum_s \sum_t X[s,t] \cdot K[s+i, t+j]$$

where the multi-index (s,t) runs over the admissible domain, namely the shape of `Y`.

```
[ ]  from d2l import torch as d2l
     import torch
     from torch import nn
```

```
[ ]  def corr2d(X, K):
         """Compute 2D cross-correlation."""
         h, w = K.shape
         Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
         for i in range(Y.shape[0]):
             for j in range(Y.shape[1]):
                 Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
         return Y
```

Example: let us construct an input tensor `X` and a kernel tensor `K` to validate the cross-correlation operation

```
[ ]  X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
     K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
     corr2d(X, K)
```

```
     tensor([[19., 25.],
             [37., 43.]])
```

Modules

We learned that `nn.Sequential` is a container of layers that are **chained** in cascade.

There is a more general and flexible container: a *PyTorch module*, namely `nn.Module` class (not to be confused with the concept of module=library in Python)

- PyTorch module can contain other layers and modules as well, i.e. modules can be **nested** in a complicate manner
    - a `nn.Module` can be just one layer
- `nn.Sequential` is a subclass of `nn.Module`
    - it can be a sequence of modules
- price to pay for flexibility of `nn.Module`: writing more code
    - we are required to define a sublass implementing (the constructor `__init__` and) the `foward()` method
    - when a network is used in *forward computation*, the `forward()` method of the class is called
        - for `nn.Sequential` PyTorch suitably wires outputs/inputs of the chained layers automatically
        - while for a `nn.Module` instance the `forward()` method must be **implemented manually** by the programmer
- PyTorch module can be used to implement a *block*, which loosely stands for common group/structure of layers that occurs several times to build up the whole architecture, like in certain modern Deep NN architectures

Convolutional Layers

We have the `corr2d` function implementing the 2D convolution.

This is enough for constructing a **convolutional layer**

- need to suitably define a subclass of PyTorch module
- `nn.Parameter` allows to define the intrinsic parameters of the layer, eg. kernel's weights

```
[ ]  class Conv2D(nn.Module): # Conv2D inherits from nn.Module class
         # constructor: we need to initialize Conv2D object using the superclass' constructor first
         def __init__(self, kernel_size):
             super().__init__()
             self.weight = nn.Parameter(torch.rand(kernel_size))
             self.bias = nn.Parameter(torch.zeros(1)) # not much useful in this specific example, but convolut

         # forward() method needed in order to implement network's forward computation
         def forward(self, x):
             return corr2d(x, self.weight) + self.bias
```

In $h \times w$ convolution or a $h \times w$ convolution kernel, the height and width of the convolution kernel are h and w, respectively. We also refer to a convolutional layer with a $h \times w$ convolution kernel simply as a $h \times w$ convolutional layer.

Elementary convolutional layer implementation: Object Edge Detection in Images

Let us detect the edge of an object in a "synthetic image" by finding the **location of the pixel change**.

1. construct a 6×8 pixel image: middle 4 columns' values are 0=black and the rest are 1=white

```
[ ]  X = torch.ones((6, 8))
     X[:, 2:6] = 0
     X
```

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

2. construct a 1×2 kernel `K`

```
[ ]  K = torch.tensor([[1.0, -1.0]])
```

It behaves as elementary vertical **edge detector**: when performing the cross-correlation operation with the input...

- if the **horizontally adjacent elements** are the **same**, the output is 0
- otherwise, the output is $\neq 0$

3. perform cross-correlation

- edge from white to black will yield 1
- edge from black to white will yield -1

```
[ ]  Y = corr2d(X, K)
     Y
```

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

Does this particular convolutional kernel detect **only vertical** edges in an image?

- it should be clear from the evident directionality
- by transposing the image: surely the edge turns to horizontal
    - then try applying the same kernel

```
[ ]  corr2d(X.t(), K)
```

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

This kernel simply calculated the difference between two adjacent entries: in numerical mathematics we would say that it calculates the *finite difference* (not far from the numerical approximation of a first-order directional derivative)

Using built-in Convolutional Layer: learn a kernel

We had the synthetic image `X`, the given kernel `K`, and then calculated the output `Y`

Goal: let us try to learn the kernel for vertical edge detection, based on examples: (`X`, `Y`)

1. construct a convolutional layer
2. initialize the kernel randomly
    - PyTorch has a default method for randomly initializing the parameters in convolutional (and linear) layers. You can read the details here: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html
3. compute the error wrt the known edges
4. update the kernel using the gradient

Instead of manually implementing a **2D convolutional layer**

- PyTorch's built-in class `nn.Conv2d`
    - arguments: # input channels, # output channels, kernel size, stride (default: 1), padding (default: 0), bias (default: True), ...
      https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html
    - images as input to such 2D convolutional layer: 4-dimensional tensor with shape (batch size, # channels, height, width)

```
[ ]  # Construct a 2D convolutional layer: 1 input and output channel, 1x2 kernel
     # For the sake of simplicity, we cancel the bias here
     conv2d = nn.Conv2d(1,1, kernel_size=(1, 2), bias=False)

     # 1 example per batch
     # 1 channel in the images
     # 6x8 images
     X = X.reshape((1, 1, 6, 8))
     Y = Y.reshape((1, 1, 6, 7))

     # This loop manually iterates in the gradient descent method, step size = 0.03
     for i in range(10):
         Y_hat = conv2d(X)
         l = (Y_hat - Y) ** 2
         conv2d.zero_grad()
         l.sum().backward() # This calls the backpropagation, calculating the gradient(s) only
         # Then update the kernel weights
         conv2d.weight.data[:] -= 3e-2 * conv2d.weight.grad
         if (i + 1) % 2 == 0:
             print(f'batch {i + 1}, loss {l.sum():.3f}')
```

```
batch 2, loss 1.586
batch 4, loss 0.269
batch 6, loss 0.046
batch 8, loss 0.008
batch 10, loss 0.002
```

Note that the error has dropped to a small value after 10 iterations. Now we will take a look at the kernel tensor we learned.

- Learnt kernel weights:

```
[ ]  conv2d.weight.data.reshape((1, 2))

     tensor([[ 0.9915, -0.9954]])
```

Indeed, the learned kernel tensor is remarkably close to the kernel tensor `K` we defined earlier.

Feature Map and Receptive Field

Feature map of the 2D convolution: is the result of the 2D convolution of an image with a kernel

Receptive field of **one** entry in the feature map: the patch in the input of the convolutional layer that is used for calculating that entry

- clearly: shape of a receptive field = shape of the kernel

Summary

- The core computation of a two-dimensional convolutional layer is a two-dimensional cross-correlation operation. In its simplest form, this performs a cross-correlation operation on the two-dimensional input data and the kernel, and then adds a bias.
- We can design a kernel to detect edges in images.
- We can learn the kernel's parameters from data.
- With kernels learned from data, the outputs of convolutional layers remain unaffected regardless of such layers' performed operations (either strict convolution or cross-correlation).
- When any element in a feature map needs a larger receptive field to detect broader features on the input, a deeper network can be considered.

Exercises

1. Construct an image `X` with diagonal edges.
    1. What happens if you apply the kernel `K` in this section to it?
    2. What happens if you transpose `X`?
    3. What happens if you transpose `K`?
2. How do you represent a cross-correlation operation as a matrix multiplication by changing the input and kernel tensors?
3. Design some kernels manually.
    1. What is the form of a kernel for the ("discretize" version of) second derivative?
    2. What is the kernel for (the "discrete" version of) an integral?
    3. What is the minimum size of a kernel to obtain a ("discrete" version of the) derivative of degree *d*?

Deep Convolutional Neural Network example architecture: AlexNet

AlexNet, novelties w.r.t. LeNet

- it's an 8-layer CNN (deeper than LeNet)
- use of ReLU
- start with a larger convolution window size (11x11)
- adopt the MaxPooling
- Itwon the ImageNet Large Scale Visual Recognition Challenge 2012

```
[ ] !pip install torch d2l
```

AlexNet model implemetation

```
[ ] from d2l import torch as d2l
    import torch
    from torch import nn

    net = nn.Sequential(
        # Here, we use a larger 11 x 11 window to capture objects. At the same
        # time, we use a stride of 4 to greatly reduce the height and width of the
        # output. Here, the number of output channels is much larger than that in
        # LeNet
        nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2),
        # Make the convolution window smaller, set padding to 2 for consistent
        # height and width across the input and output, and increase the number of
        # output channels
        nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2),
        # Use three successive convolutional layers and a smaller convolution
        # window. Except for the final convolutional layer, the number of output
        # channels is further increased. Pooling layers are not used to reduce the
        # height and width of input after the first two convolutional layers
        nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),
        nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU(),
        nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2),
        nn.Flatten(),
        # Here, the number of outputs of the fully-connected layer is several
        # times larger than that in LeNet. Use the dropout layer to mitigate
        # overfitting
        nn.Linear(6400, 4096), nn.ReLU(),
        nn.Dropout(p=0.5),
        nn.Linear(4096, 4096), nn.ReLU(),
        nn.Dropout(p=0.5),
        # Output layer. Since we are using Fashion-MNIST, the number of classes is
        # 10, instead of 1000 as in the paper
        nn.Linear(4096, 10))
```

- We try on a single-channel 224 x 224 data example to observe the output shape of each layer

```
[ ] X = torch.randn(1, 1, 224, 224)
    for layer in net:
        X=layer(X)
        print(layer.__class__.__name__,'Output shape:\t',X.shape)
```

```
Conv2d Output shape:     torch.Size([1, 96, 54, 54])
ReLU Output shape:       torch.Size([1, 96, 54, 54])
MaxPool2d Output shape:  torch.Size([1, 96, 26, 26])
Conv2d Output shape:     torch.Size([1, 256, 26, 26])
ReLU Output shape:       torch.Size([1, 256, 26, 26])
MaxPool2d Output shape:  torch.Size([1, 256, 12, 12])
Conv2d Output shape:     torch.Size([1, 384, 12, 12])
ReLU Output shape:       torch.Size([1, 384, 12, 12])
Conv2d Output shape:     torch.Size([1, 384, 12, 12])
ReLU Output shape:       torch.Size([1, 384, 12, 12])
Conv2d Output shape:     torch.Size([1, 256, 12, 12])
ReLU Output shape:       torch.Size([1, 256, 12, 12])
```

Reading the Dataset

Let us use the Fashion-MNIST dataset for reducing the computational time:

- need to **upsample** from 28 x 28 pixels to 224 x 224: using `resize` argument

```
[4] batch_size = 128
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-idx3-ubyte.gz
```

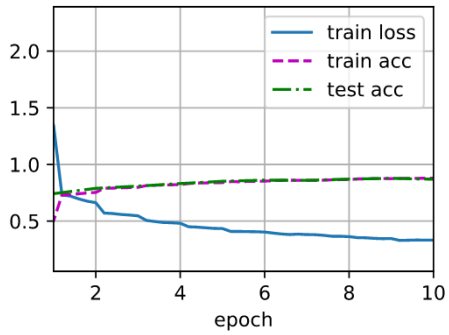
Training

Main changes (wrt LeNet):

- use of a smaller learning rate
- much slower training due to
    - the deeper and wider network,
    - the higher image resolution,
    - the more costly convolutions.

```
[5] lr, num_epochs = 0.01, 10
    d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, device=d2l.try_gpu())
```

```
loss 0.332, train acc 0.878, test acc 0.870
415.3 examples/sec on cuda:0
```



Summary

- AlexNet has a similar structure to that of LeNet, but uses more convolutional layers and a larger parameter space to fit the large-scale ImageNet dataset.
- Today AlexNet has been surpassed by much more effective architectures but it is a key step from shallow to deep networks that are used nowadays.
- Although it seems that there are only a few more lines in AlexNet's implementation than in LeNet, it took the academic community many years to embrace this conceptual change and take advantage of its excellent experimental results. This was also due to the lack of efficient computational tools.
- Dropout, ReLU, and preprocessing were the other key steps in achieving excellent performance in computer vision tasks.

Exercises

1. Try increasing the number of epochs. Compared with LeNet, how are the results different? Why?
2. AlexNet may be too complex for the Fashion-MNIST dataset. Try simplifying the model to make the training faster, while ensuring that the accuracy does not drop significantly.
3. Modify the batch size, and observe the changes in accuracy and GPU memory.

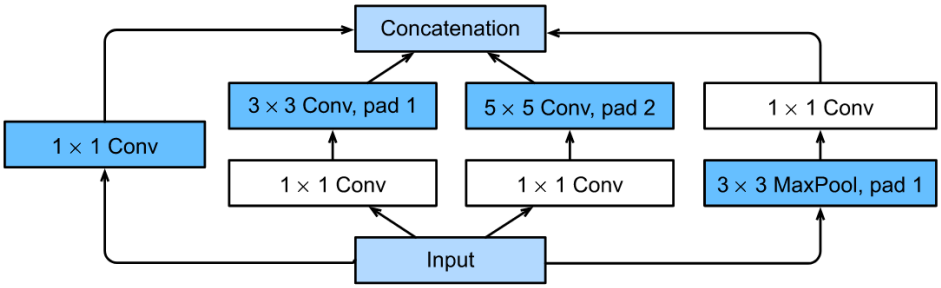
Networks with Parallel Concatenations (GoogLeNet)

Idea: employ a combination of variously-sized kernels working in parallel, capturing details at different extents at the same network depth

```
[ ] !pip install torch d2l
```

Inception Blocks

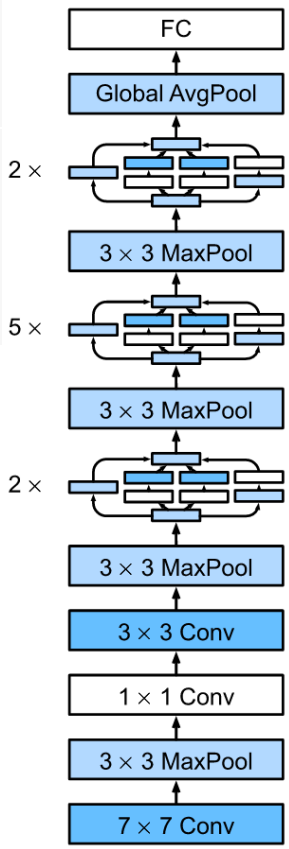
Inception block: four parallel paths, concatenated to obtain the block output




```
[ ] from d2l import torch as d2l
    import torch
    from torch import nn
    from torch.nn import functional as F

    class Inception(nn.Module):
        # `c1`--`c4` are the number of output channels for each path
        def __init__(self, in_channels, c1, c2, c3, c4, **kwargs): #192, 64, (96, 128), (16, 32), 32)
            super(Inception, self).__init__(**kwargs)
            # Path 1 is a single 1 x 1 convolutional layer
            self.p1_1 = nn.Conv2d(in_channels, c1, kernel_size=1)
            # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
            # convolutional layer
            self.p2_1 = nn.Conv2d(in_channels, c2[0], kernel_size=1)
            self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
            # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
            # convolutional layer
            self.p3_1 = nn.Conv2d(in_channels, c3[0], kernel_size=1)
            self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
            # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
            # convolutional layer
            self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
            self.p4_2 = nn.Conv2d(in_channels, c4, kernel_size=1)

        def forward(self, x):
            p1 = F.relu(self.p1_1(x))
            p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
            p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
            p4 = F.relu(self.p4_2(self.p4_1(x)))
            # Concatenate the outputs on the channel dimension
            return torch.cat((p1, p2, p3, p4), dim=1)  # cat concatenation
```

GoogLeNet Architecture

GoogLeNet uses a stack of a total of 9 inception blocks and global average pooling to generate its estimates.



First module uses a 64-channel 7×7 convolutional layer.

```
[ ]  b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3), # b1 first state
                        nn.ReLU(),
                        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

And so on ... other modules

```
[ ]  b2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=1),  # b2 second state
                        nn.ReLU(),
                        nn.Conv2d(64, 192, kernel_size=3, padding=1),
                        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

Now comes the **Inception blocks**!

- we use the blocks as defined above deriving a subclass, `Inception`, of Module

```
[ ]  b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),
                        Inception(256, 128, (128, 192), (32, 96), 64),
                        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

```
[ ]  b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                        Inception(512, 160, (112, 224), (24, 64), 64),
                        Inception(512, 128, (128, 256), (24, 64), 64),
                        Inception(512, 112, (144, 288), (32, 64), 64),
                        Inception(528, 256, (160, 320), (32, 128), 128),
                        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

```
[ ]  b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),
                        Inception(832, 384, (192, 384), (48, 128), 128),
                        nn.AdaptiveAvgPool2d((1,1)),
                        nn.Flatten())

     net = nn.Sequential(b1, b2, b3, b4, b5, nn.Linear(1024, 10))
```

```
[ ]  X = torch.rand(size=(1, 1, 96, 96))
     for layer in net:
         X = layer(X)
         print(layer.__class__.__name__,'output shape:\t', X.shape)
```

```
     /usr/local/lib/python3.7/dist-packages/torch/nn/functional.py:718: UserWarning: Named tensors and all thei
```

Training

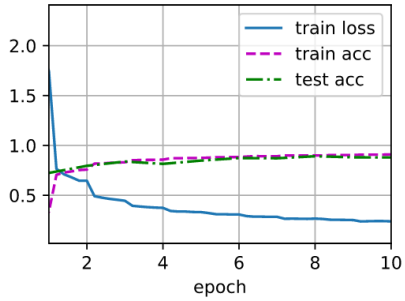
As before, we train our model using the Fashion-MNIST dataset. We transform it to 96×96 pixel resolution before invoking the training procedure.

Notice again: `train_ch6` function in textbook's library `d2l` can **move data** and the **network** to the **GPU device** for carrying out the training.

- check out that function definition for understanding how to suitably use the `.to()` method

```
[ ]  lr, num_epochs, batch_size = 0.1, 10, 128
     train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
     d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, device=d2l.try_gpu())
```

```
loss 0.239, train acc 0.909, test acc 0.879
391.1 examples/sec on cuda:0
```



Summary

- The Inception block is equivalent to a subnetwork with four paths. It extracts information in parallel through convolutional layers of different window shapes and maximum pooling layers. 1×1 convolutions reduce channel dimensionality on a per-pixel level. Maximum pooling reduces the resolution.
- GoogLeNet connects multiple well-designed Inception blocks with other layers in series. The ratio of the number of channels assigned in the Inception block is obtained through a large number of experiments on the ImageNet dataset.
- GoogLeNet, as well as its succeeding versions, was one of the most efficient models on ImageNet, providing similar test accuracy with lower computational complexity.

Exercises

1. What is the minimum image size for GoogLeNet to work?
2. Compare the model parameter sizes of AlexNet, VGG, and NiN with GoogLeNet. How do the latter two network architectures significantly reduce the model parameter size?

LIBRO

1- 21
2- 33
3- 25
4- 27