Layers and Blocks

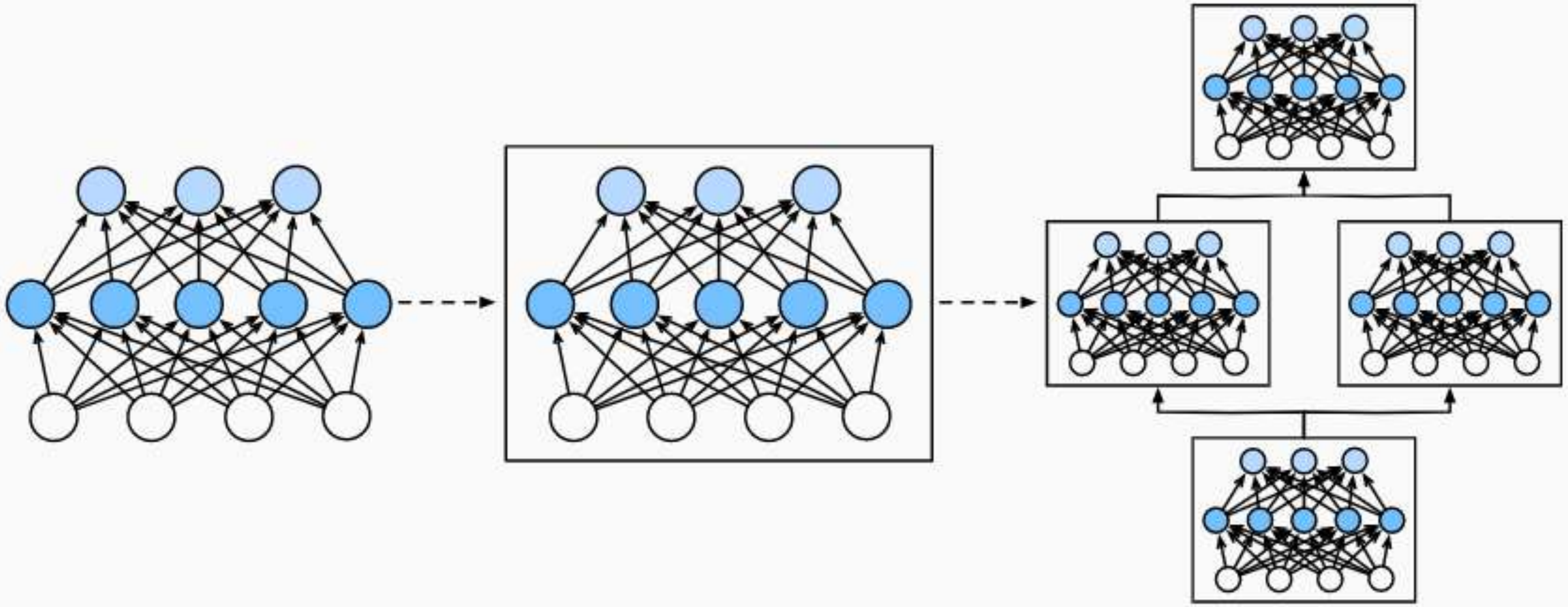

Fig. 5.1.1 Multiple layers are combined into blocks, forming repeating patterns of larger models.

- Take a set of **inputs**
- Generate corresponding **outputs**
- Described by a set of **tunable parameters**

Layers and Blocks

PyTorch

```
import torch
from torch import nn
from torch.nn import functional as F

net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)
net(X)
```

TensorFlow

```
import tensorflow as tf

net = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dense(10),
])

X = tf.random.uniform((2, 20))
net(X)
```

Layers and Blocks

Basic functionality that each block must provide

- **Input data** as arguments to its forward propagation function
- **Generate an output** by having the forward propagation function return a value
- **Calculate the gradient** of its output with respect to its input (Typically this happens automatically)
- **Store and provide access** to those parameters necessary to execute
- **Initialize model parameters** as needed

```python
class MLP(tf.keras.Model):
    # Declare a layer with model parameters. Here, we declare two fully
    # connected layers
    def __init__(self):
        # Call the constructor of the `MLP` parent class `Block` to perform
        # the necessary initialization. In this way, other function arguments
        # can also be specified during class instantiation, such as the model
        # parameters, `params` (to be described later)
        super().__init__()
        # Hidden layer
        self.hidden = tf.keras.layers.Dense(units=256, activation=tf.nn.relu)
        self.out = tf.keras.layers.Dense(units=10)  # Output layer

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input `X`
    def call(self, X):
        return self.out(self.hidden((X)))
```

Layers and Blocks

Sequential Block

```python
class MySequential(tf.keras.Model):
    def __init__(self, *args):
        super().__init__()
        self.modules = []
        for block in args:
            # Here, `block` is an instance of a `tf.keras.layers.Layer`
            # subclass
            self.modules.append(block)

    def call(self, X):
        for module in self.modules:
            X = module(X)
        return X
```

```python
net = MySequential(
    tf.keras.layers.Dense(units=256, activation=tf.nn.relu),
    tf.keras.layers.Dense(10))
net(X)
```

- **Function to append** blocks one by one to a list
- Forward propagation to **pass an input through the chain of blocks**

Layers and Blocks

Executing Code in the Forward Propagation Function

```
class FixedHiddenMLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.flatten = tf.keras.layers.Flatten()
        # Random weight parameters created with `tf.constant` are not updated
        # during training (i.e., constant parameters)
        self.rand_weight = tf.constant(tf.random.uniform((20, 20)))
        self.dense = tf.keras.layers.Dense(20, activation=tf.nn.relu)

    def call(self, inputs):
        X = self.flatten(inputs)
        # Use the created constant parameters, as well as the `relu` and
        # `matmul` functions
        X = tf.nn.relu(tf.matmul(X, self.rand_weight) + 1)
        # Reuse the fully-connected layer. This is equivalent to sharing
        # parameters with two fully-connected layers
        X = self.dense(X)
        # Control flow
        while tf.reduce_sum(tf.math.abs(X)) > 1:
            X /= 2
        return tf.reduce_sum(X)
```

- **rand_weight** (initialized randomly and **constant**)
- **Arbitrary** mathematical ops

Layers and Blocks

```
class NestMLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.net = tf.keras.Sequential()
        self.net.add(tf.keras.layers.Dense(64, activation=tf.nn.relu))
        self.net.add(tf.keras.layers.Dense(32, activation=tf.nn.relu))
        self.dense = tf.keras.layers.Dense(16, activation=tf.nn.relu)

    def call(self, inputs):
        return self.dense(self.net(inputs))

chimera = tf.keras.Sequential()
chimera.add(NestMLP())
chimera.add(tf.keras.layers.Dense(20))
chimera.add(FixedHiddenMLP())
chimera(X)
```

Parameter Management

Find **parameter values** that minimize our loss function !!

- After choosing an **architecture** and set our **hyperparameters**
- Almost nitty-gritty details about parameters are done by deep learning frameworks
- Focusing on
  - Accessing parameters for **debugging, diagnostics and visualization**
  - Parameter **initialization**
  - **Sharing** parameters across different model components

```
import tensorflow as tf
import numpy as np

net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4, activation=tf.nn.relu),
    tf.keras.layers.Dense(1),
])

X = tf.random.uniform((2, 4))
net(X)
```

Parameter Management

Parameter Access

```
print(net.layers[2].weights)
```

```
[<tf.Variable 'dense_1/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[ 0.611843 ],
       [-1.0367968],
       [-0.626264 ],
       [-1.0634434]], dtype=float32)>, <tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32,
```

```
print(type(net.layers[2].weights[1]))
print(net.layers[2].weights[1])
print(tf.convert_to_tensor(net.layers[2].weights[1]))
```

```
<class 'tensorflow.python.ops.resource_variable_ops.ResourceVariable'>
<tf.Variable 'dense_1/bias:0' shape=(1,) dtype=float32, numpy=array([0.], dtype=float32)>
tf.Tensor([0.], shape=(1,), dtype=float32)
```

Parameter Management

Parameter Access

```
import tensorflow as tf
import numpy as np

net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(4, activation=tf.nn.relu),
    tf.keras.layers.Dense(1),
])

X = tf.random.uniform((2, 4))
net(X)
```

```
print(net.layers[1].weights)
print(net.get_weights())
```

```
[<tf.Variable 'dense/kernel:0' shape=(4, 4) dtype=float32, numpy=
array([[-0.4394562 , -0.03386152, -0.51694846, -0.37519595],
       [-0.31819874,  0.80430835, -0.6736374 , -0.525049  ],
       [-0.11170632,  0.72245175,  0.22587043, -0.11595535],
       [-0.7310393 , -0.64875305,  0.12634093, -0.7527868 ]],
      dtype=float32)>, <tf.Variable 'dense/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0.
[array([[-0.4394562 , -0.03386152, -0.51694846, -0.37519595],
       [-0.31819874,  0.80430835, -0.6736374 , -0.525049  ],
       [-0.11170632,  0.72245175,  0.22587043, -0.11595535],
       [-0.7310393 , -0.64875305,  0.12634093, -0.7527868 ]],
      dtype=float32), array([0., 0., 0., 0.], dtype=float32), array([[ 0.611843 ],
       [-1.0367968],
       [-0.626264 ],
       [-1.0634434]], dtype=float32), array([0.], dtype=float32)]
```

Parameter Management

Parameter Access

```python
def block1():
    return nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                         nn.Linear(8, 4), nn.ReLU())

def block2():
    net = nn.Sequential()
    for i in range(4):
        # Nested here
        net.add_module(f'block {i}', block1())
    return net

rgnet = nn.Sequential(block2(), nn.Linear(4, 1))
rgnet(X)
```

```
Sequential(
  (0): Sequential(
    (block 0): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 1): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 2): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 3): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
  )
  (1): Linear(in_features=4, out_features=1, bias=True)
)
```

Parameter Management

Parameter Initialization

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4, activation=tf.nn.relu,
        kernel_initializer=tf.random_normal_initializer(mean=0, stddev=0.01),
        bias_initializer=tf.zeros_initializer()),
    tf.keras.layers.Dense(1)])

net(X)
net.weights[0], net.weights[1]
```

```
(<tf.Variable 'dense_7/kernel:0' shape=(4, 4) dtype=float32, numpy=
 array([[-0.01232721,  0.00203559,  0.01416426,  0.01324802],
        [-0.00774537, -0.00615086,  0.00605043,  0.00262026],
        [ 0.00496995,  0.01149419, -0.0010287 ,  0.00156263],
        [ 0.00589002,  0.01721195, -0.01172042,  0.00626536]],
       dtype=float32)>,
 <tf.Variable 'dense_7/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0., 0., 0.],
```

- Normal distribution (mean = 0, stddev = 0.01) for weight
- Zero for bias

Parameter Management

Parameter Initialization

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4, activation=tf.nn.relu,
        kernel_initializer=tf.keras.initializers.Constant(1),
        bias_initializer=tf.zeros_initializer()),
    tf.keras.layers.Dense(1),
])

net(X)
net.weights[0], net.weights[1]
```

```
(<tf.Variable 'dense_9/kernel:0' shape=(4, 4) dtype=float32, numpy=
 array([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]], dtype=float32)>,
 <tf.Variable 'dense_9/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0., 0., 0.],
```

- One for weight
- Zero for bias

Parameter Management

```
net = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(
        4,
        activation=tf.nn.relu,
        kernel_initializer=tf.keras.initializers.GlorotUniform()),
    tf.keras.layers.Dense(
        1, kernel_initializer=tf.keras.initializers.Constant(1)),
])

net(X)
print(net.layers[1].weights[0])
print(net.layers[2].weights[0])
```

```
<tf.Variable 'dense_11/kernel:0' shape=(4, 4) dtype=float32, numpy=
array([[-5.9017664e-01,  4.9841267e-01, -2.2482872e-04,  6.0929531e-01],
       [-6.9594216e-01, -2.5807732e-01,  5.5503267e-01, -2.5179982e-03],
       [ 7.6451021e-01,  6.9580036e-01,  6.8952948e-01,  8.6293107e-01],
       [-6.5245885e-01, -7.7867943e-01,  1.4007348e-01,  3.8748115e-01]],
      dtype=float32)>
<tf.Variable 'dense_12/kernel:0' shape=(4, 1) dtype=float32, numpy=
array([[1.],
       [1.],
       [1.],
       [1.]], dtype=float32)>
```

- Possible to apply different initializers for certain blocks

Parameter Management

Tied Parameters

```
# We need to give the shared layer a name so that we can refer to its
# parameters
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.Linear(8, 1))
net(X)
# Check whether the parameters are the same
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[2].weight.data[0] == net[4].weight.data[0])
```

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

- Not just equal, they are represented by the same exact tensor
- Since the model parameters contain gradients, the gradients of the second hidden layer and the third layer are added together during backpropagation

Parameter Management

Tied Parameters

```
# We need to give the shared layer a name so that we
# parameters
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.Linear(8, 1))
net(X)
# Check whether the parameters are the same
print(net[2].weight.data[0] == net[4].weight.data[0]
net[2].weight.data[0, 0] = 100
# Make sure that they are actually the same object r
# same value
print(net[2].weight.data[0] == net[4].weight.data[0]
```

```
tensor([True, True, True, True, True, True, True, Tr
tensor([True, True, True, True, True, True, True, Tr
```

ganeshk — 11 Oct

For tied parameters (link), why is the gradient the sum of the gradients of the two layers? I was thinking it would be the product of the gradients of the two layers. Reasoning:

y = f(f(x))
dy/dx = f'(f(x))*f'(x) where x is a vector denoting the shared parameters.

(Cross posting from the D2L pytorch forum, since it does not really have anything to do with pytorch).

1 reply

goldpiggy — 15 Oct

> ganeshk:
>
> For tied parameters (link), why is the gradient the sum of the gradients of the two layers?

Hi @ganeshk, fantastic question! Even though it is not intuitively obvious, we design the operator by using "sum" rather than "product". That aligns with the idea how we learn a convolution kernel. Check this tutorial for more details.

1 reply

ganeshk — ▸ goldpiggy — 16 Oct

This is helpful. Thanks. I suppose having a product is more likely to lead to problems like vanishing gradients. The sum should be more stable to that.

1 reply

goldpiggy — ▸ ganeshk — 16 Oct

Great @ganeshk ! As you may understand now, theoretical intuition needs more practical experiments 😉 . Good luck!

Deferred Initialization

We did the following unintuitive things, but it runs!

- Defined the network architecture without specifying the **input dimensionality**
- Added layers **without specifying the output dimension** of the previous layer
- Even '**initialized' these parameters** before providing enough information to determine how many parameters our model should contain

Waiting until the first time we pass data through the model

- Defers initialization, infer the sizes of each layer on the fly
- More convenient when working with convolutional neural networks
  - Input dimensionality (i.e., the resolution of an image) will affect the dim of each sub layer

Deferred Initialization

We did the following unintuitive things, but it runs!

```
import tensorflow as tf

net = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation=tf.nn.relu),
    tf.keras.layers.Dense(10),
])
```

```
[net.layers[i].get_weights() for i in range(len(net.layers))]
```

```
[[], []]
```

```
X = tf.random.uniform((2, 20))
net(X)
[w.shape for w in net.get_weights()]
```

```
[(20, 256), (256,), (256, 10), (10,)]
```

Custom Layers

Layers without parameters

```python
import tensorflow as tf

class CenteredLayer(tf.keras.Model):
    def __init__(self):
        super().__init__()

    def call(self, inputs):
        return inputs - tf.reduce_mean(inputs)
```

```python
layer = CenteredLayer()
layer(tf.constant([1, 2, 3, 4, 5]))
```

```
<tf.Tensor: shape=(5,), dtype=int32, numpy=array([-2, -1,  0,  1,  2], dtype=int32)>
```

```python
net = tf.keras.Sequential([tf.keras.layers.Dense(128), CenteredLayer()])
```

```python
Y = net(tf.random.uniform((4, 8)))
tf.reduce_mean(Y)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=4.1909516e-09>
```

Custom Layers

Layers with parameters

```
class MyDense(tf.keras.Model):
    def __init__(self, units):
        super().__init__()
        self.units = units

    def build(self, X_shape):
        self.weight = self.add_weight(name='weight',
            shape=[X_shape[-1], self.units],
            initializer=tf.random_normal_initializer())
        self.bias = self.add_weight(
            name='bias', shape=[self.units],
            initializer=tf.zeros_initializer())

    def call(self, X):
        linear = tf.matmul(X, self.weight) + self.bias
        return tf.nn.relu(linear)
```

```
dense = MyDense(3)
dense(tf.random.uniform((2, 5)))
dense.get_weights()
```

```
[array([[ 8.5037880e-02, -6.5666504e-02,  4.8301648e-02],
        [ 7.5907312e-02, -6.5662928e-02, -1.7469548e-02],
        [ 9.7988052e-03, -3.8920205e-02,  2.5238220e-02],
        [ 5.8334973e-02,  5.7492969e-05,  1.8756824e-03],
        [ 1.0372555e-01,  5.6811761e-02,  3.4672324e-02]], dtype=float32),
 array([0., 0., 0.], dtype=float32)]
```

File I/O

Loading and Saving Tensors

- Want to save the results for later use (e.g. make predictions in deployment)
- When running a long training process, the best practice is to periodically save intermediate results
- Time to learn how to load and store both individual weights vectors and entire models

```python
import tensorflow as tf
import numpy as np

x = tf.range(4)
np.save("x-file.npy", x)
```

```python
x2 = np.load('x-file.npy', allow_pickle=True)
x2
```

```
array([0, 1, 2, 3], dtype=int32)
```

```python
y = tf.zeros(4)
np.save('xy-files.npy', [x, y])
x2, y2 = np.load('xy-files.npy', allow_pickle=True)
(x2, y2)
```

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

```python
mydict = {'x': x, 'y': y}
np.save('mydict.npy', mydict)
mydict2 = np.load('mydict.npy', allow_pickle=True)
mydict2
```

```
array({'x': <tf.Tensor: shape=(4,), dtype=int32, numpy=array([0, 1, 2, 3], dtype=int32)>, 'y':
      dtype=object)
```

File I/O

Loading and Saving Tensors

- Want to save the results for later use (e.g. make predictions in deployment)
- When running a long training process, the best practice is to periodically save intermediate results
- Time to learn how to load and store both individual weights vectors and entire models

```python
import torch
from torch import nn
from torch.nn import functional as F

x = torch.arange(4)
torch.save(x, 'x-file')
```

```python
x2 = torch.load("x-file")
x2
```

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

```python
y = torch.zeros(4)
torch.save([x, y],'x-files')
x2, y2 = torch.load('x-files')
(x2, y2)
```

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

```python
mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2
```

```
{'x': tensor([0, 1, 2, 3]), 'y': tensor([0., 0., 0., 0.])}
```

File I/O

Loading and Saving Model Parameters

- We might have hundreds of parameter groups and framework provides built-in functions
- This saves model **parameters** and not the entire model
- We need to **generate the architecture in code first** and then **load the parameters** from disk

```
class MLP(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.flatten = tf.keras.layers.Flatten()
        self.hidden = tf.keras.layers.Dense(units=256, activation=tf.nn.relu)
        self.out = tf.keras.layers.Dense(units=10)

    def call(self, inputs):
        x = self.flatten(inputs)
        x = self.hidden(x)
        return self.out(x)

net = MLP()
X = tf.random.uniform((2, 20))
Y = net(X)
```

```
net.save_weights('mlp.params')
```

```
clone = MLP()
clone.load_weights("mlp.params")
```

```
Y_clone = clone(X)
Y_clone == Y
```

```
<tf.Tensor: shape=(2, 10), dtype=bool, numpy=
array([[ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,
         True]])>
```

GPUs

Computing Devices

- Can specify devices CPUs or GPUs

```
import tensorflow as tf

tf.device('/CPU:0'), tf.device('/GPU:0'), tf.device('/GPU:1')
```

```
(<tensorflow.python.eager.context._EagerDeviceContext at 0x7fe135bdc
 <tensorflow.python.eager.context._EagerDeviceContext at 0x7fe138c21
 <tensorflow.python.eager.context._EagerDeviceContext at 0x7fe138c21
```

```
len(tf.config.experimental.list_physical_devices('GPU'))
```

```
2
```

```
def try_gpu(i=0):  #@save
    """Return gpu(i) if exists, otherwise return cpu()."""
    if len(tf.config.experimental.list_physical_devices('GPU')) >= i + 1:
        return tf.device(f'/GPU:{i}')
    return tf.device('/CPU:0')

def try_all_gpus():  #@save
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    num_gpus = len(tf.config.experimental.list_physical_devices('GPU'))
    devices = [tf.device(f'/GPU:{i}') for i in range(num_gpus)]
    return devices if devices else [tf.device('/CPU:0')]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(<tensorflow.python.eager.context._EagerDeviceContext at 0x7fe13536af50>,
 <tensorflow.python.eager.context._EagerDeviceContext at 0x7fe13536b150>,
 [<tensorflow.python.eager.context._EagerDeviceContext at 0x7fe13536b1d0>,
  <tensorflow.python.eager.context._EagerDeviceContext at 0x7fe13536b290>])
```

GPUs

Tensors and GPUs

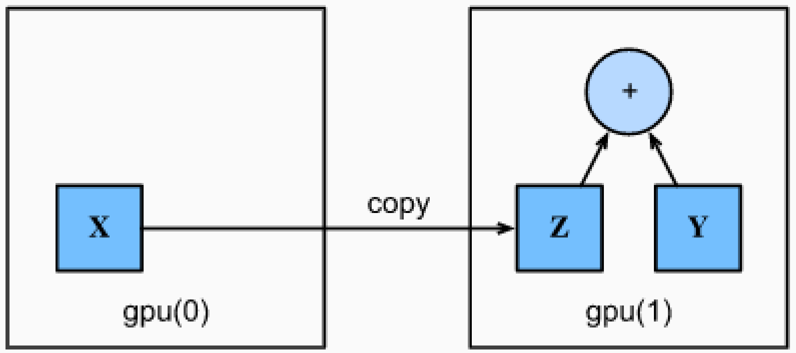

Fig. 5.6.1 Copy data to perform an operation on the same device.

- By default, tensors are created on the CPU
- When we want to operate on multiple terms, they need to be on the same device
- For example, if we sum two tensors, they should be on same device

```
x = tf.constant([1, 2, 3])
x.device
```

```
'/job:localhost/replica:0/task:0/device:CPU:0'
```

```
with try_gpu():
    X = tf.ones((2, 3))
X
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 1., 1.],
       [1., 1., 1.]], dtype=float32)>
```

```
with try_gpu(1):
    Z = X
print(X)
print(Z)
```

```
tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
tf.Tensor(
[[1. 1. 1.]
 [1. 1. 1.]], shape=(2, 3), dtype=float32)
```

GPUs

Neural Networks and GPUs

```
strategy = tf.distribute.MirroredStrategy()
with strategy.scope():
    net = tf.keras.models.Sequential([
        tf.keras.layers.Dense(1)])
```

```
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/device:GPU
```

INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/device:GPU:0', '/job:localhost/replica:0/task:0/device:GPU:1')

GPUs

Side Notes

Transferring data between (CPU, GPUs, and other machines)

- Transferring is much slower than computation
- Therefore copy operations should be taken with great care
- Many small operations are much worse than one big operation
- Such operations can block if one device has to wait for the other

When we print tensors or convert tensors to the NumPy format

- If the data is not in the main memory, framework will copy it to the main memory with overhead
- Also, it needs to be wait affected by Python GIL to complete

Q & A