UNC CHARLOTTE

The WILLIAM STATES LEE COLLEGE *of* ENGINEERING

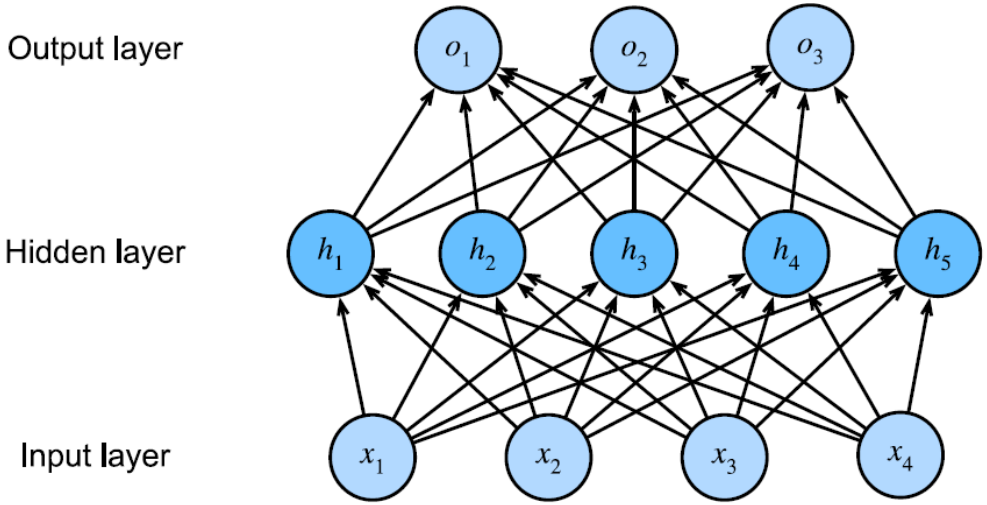
Introduction to ML
Lecture 3: Multi Layer Perceptron and GPUs

Hamed Tabkhi
Department of Electrical and Computer Engineering,
University of North Carolina Charlotte (UNCC)
htabkhiv@uncc.edu

UNC CHARLOTTE

Hidden Layers

- We can overcome the limitations of linear models by incorporating one or more hidden layers.
- The easiest way to do this is to stack many fully connected layers on top of each other.
- Each layer feeds into the layer above it, until we generate outputs.
- This architecture is commonly called a *multilayer perceptron*, often abbreviated as *MLP*



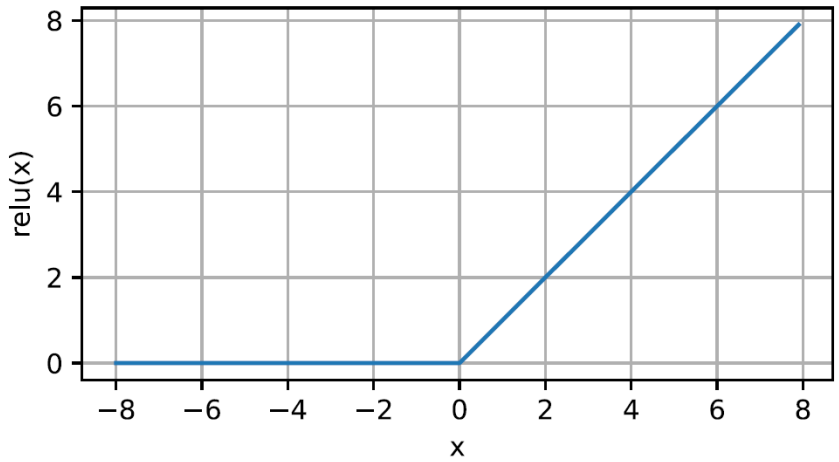
UNC CHARLOTTE

Activation Functions: ReLU Function

- The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the *rectified linear unit* (*ReLU*) (Nair and Hinton, 2010).
- ReLU provides a very simple nonlinear transformation. Given an element *x*, the function is defined as the maximum of that element and 0:

$$\mathrm{ReLU}(x) = \max(x, 0).$$

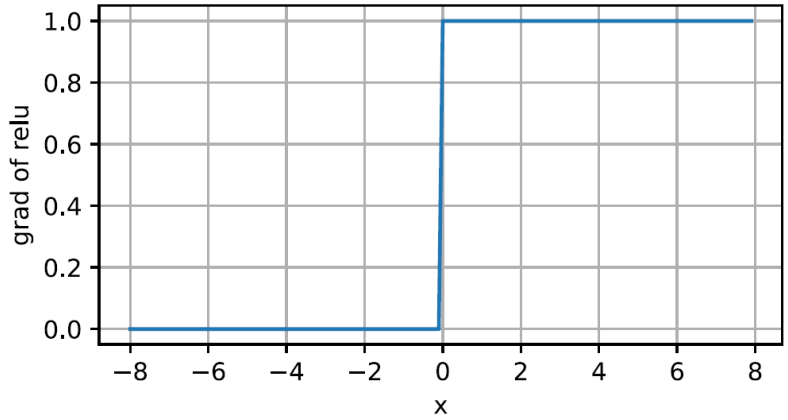
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



UNC CHARLOTTE

Activation Functions: ReLU Function

- The reason for using ReLU is that its derivatives are particularly well behaved: either they vanish, or they just let the argument through.
- This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradients that plagued previous versions of neural networks



- Note that there are many variants to the ReLU function, including the *parameterized ReLU* (*pReLU*) function (He *et al.*, 2015).
- This variation adds a linear term to ReLU, so some information still gets through, even when the argument is negative (Leaky ReLU):

$$\mathrm{pReLU}(x) = \max(0, x) + \alpha \min(0, x).$$

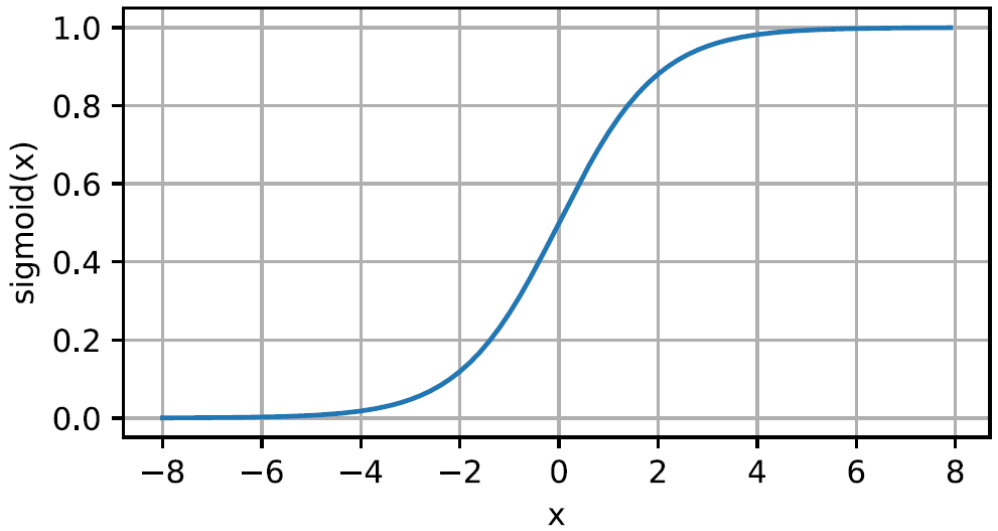
Activation Functions: Sigmoid Function

- The *sigmoid function* transforms its inputs, for which values lie in the domain R, to outputs that lie on the interval (0, 1).

- For that reason, the sigmoid is often called a *squashing function*:
  - it squashes any input in the range (-inf, inf) to some value in the range (0, 1):

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

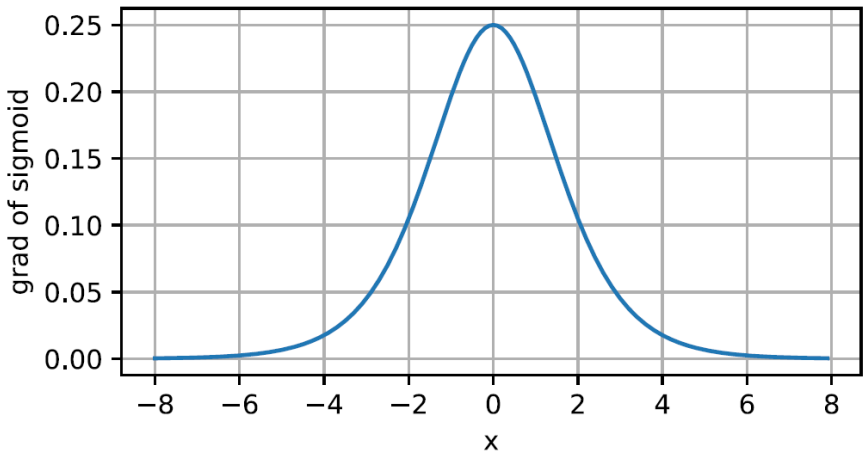
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



Activation Functions: Sigmoid Function

- The derivative of the sigmoid function is plotted below.
- Note that when the input is 0, the derivative of the sigmoid function reaches a maximum of 0.25.
- As the input diverges from 0 in either direction, the derivative approaches 0.

```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



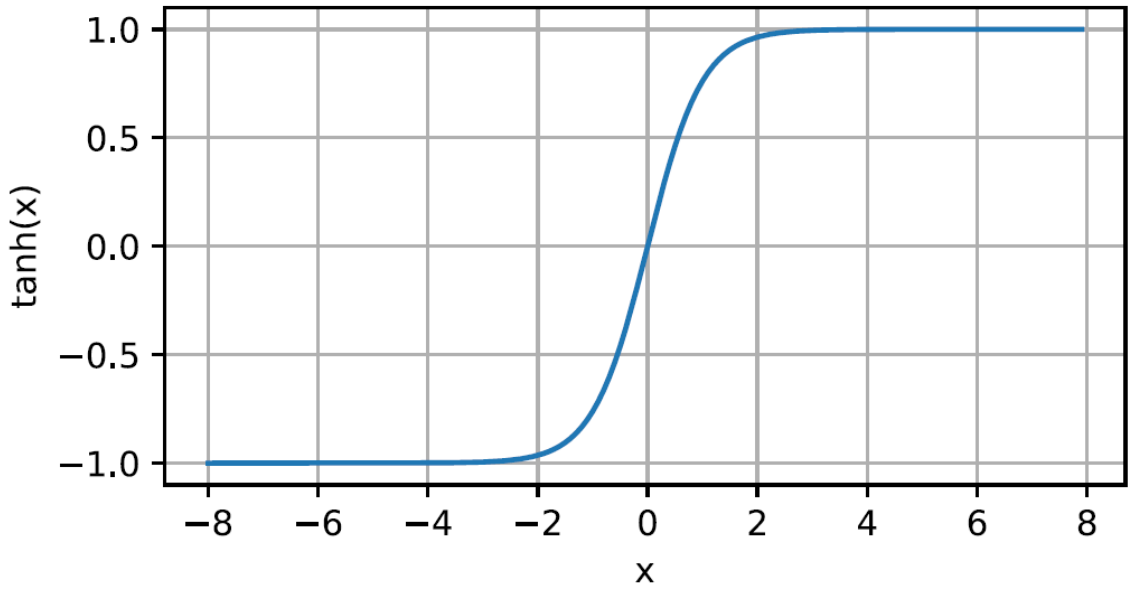
UNC CHARLOTTE

Activation Functions: Tanh Function

Like the sigmoid function, the tanh (hyperbolic tangent) function also squashes its inputs, transforming them into elements on the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



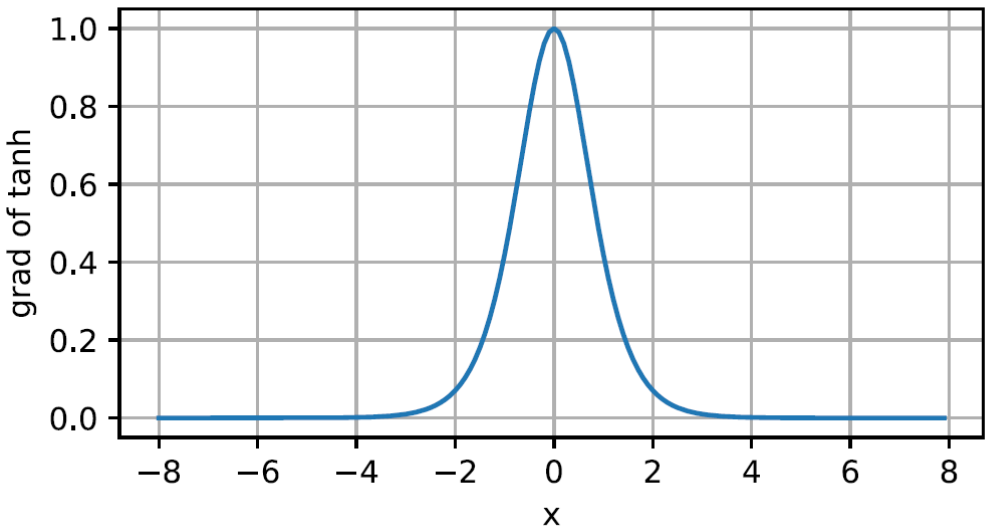
UNC CHARLOTTE

Activation Functions: Tanh Function

- As the input nears 0, the derivative of the tanh function approaches a maximum of 1.
- And as we saw with the sigmoid function, as input moves away from 0 in either direction, the derivative of the tanh function approaches 0.

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

```
# Clear out previous gradients.
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



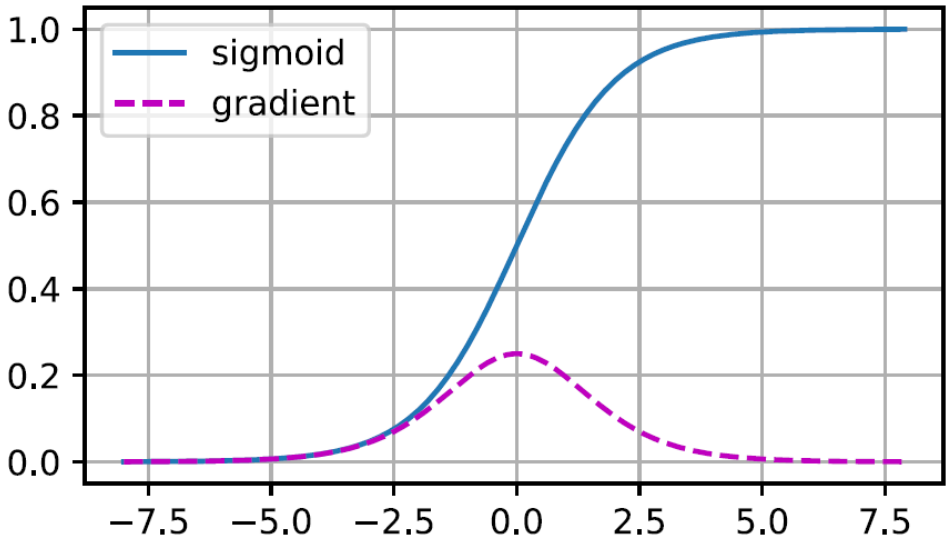
UNC CHARLOTTE

Vanishing Gradients

- One frequent culprit causing the vanishing gradient problem is the choice of the activation function that is appended following each layer's linear operations.
- Historically, the sigmoid function was popular because it resembles a
- thresholding function.
- Let's take a closer look at the sigmoid to see why it can cause vanishing gradients.

Vanishing Gradients

- As you can see, the sigmoid's gradient vanishes both when its inputs are large and when
- They are small. Moreover, when backpropagating through many layers, unless we are in the Goldilocks zone, where the inputs to many of the sigmoids are close to zero, the gradients of the overall product may vanish.
- When our network boasts many layers, unless we are careful, the gradient will likely be cut off at some layer.
- Indeed, this problem used to plague deep network training. Consequently, ReLUs, which are more stable (but less neurally plausible), have emerged as the default choice for practitioners.



UNC CHARLOTTE

Summary of Activation Function

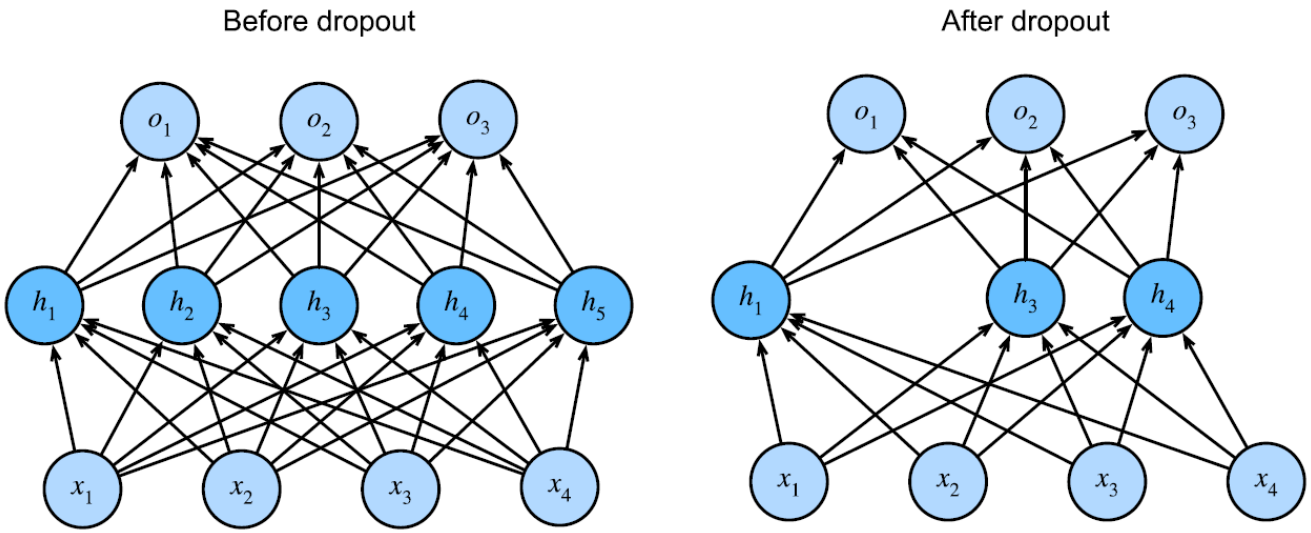
- We now know how to incorporate nonlinearities to build expressive multilayer neural network architectures
- ReLU is significantly more amenable to optimization than the sigmoid or the tanh function.
- One could argue that this was one of the key innovations that helped the resurgence of deep learning over the past decade.
- Note, though, that research in activation functions has not stopped!
  - For instance, the Swish activation function $(x) = x \text{ sigmoid}(x)$ as proposed in (Ramachandran *et al.*, 2017) can yield better accuracy in many cases.

Exploding Gradients

- The opposite problem, when gradients explode, can be similarly vexing.
- To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix.
- For the scale that we picked (the choice of the variance $_2$ = 1), the matrix product explodes.
- When this happens due to the initialization of a deep network, we have no chance of getting a gradient descent optimizer to converge.

DropOut

In standard dropout regularization, one zeros out some fraction of the nodes in each layer and then *debiases* each layer by normalizing by the fraction of nodes that were retained (not dropped out).



- Typically, we disable dropout at test time. Given a trained model and a new example, we do not drop out any nodes and thus do not need to normalize.
- However, there are some exceptions: some researchers use dropout at test time as a heuristic for estimating the *uncertainty*
- of neural network predictions: if the predictions agree across many different dropout masks, then we might say that the network is more confident.

Concise Implementation of Dropout

- With high-level APIs, all we need to do is add a Dropout layer after each fully connected layer, passing in the dropout probability as the only argument to its constructor.
- During training, the Dropout layer will randomly drop out outputs of the previous layer (or equivalently, the inputs to the subsequent layer) according to the specified dropout probability.
- When not in training mode, the Dropout layer simply passes the data through during testing.

GPUs

- We discussed the rapid growth of computation over the past two decades.
- In a nutshell, GPU performance has increased by a factor of 1000 every decade since 2000.
- This offers great opportunities but it also suggests a significant need to provide such performance.
- In this section, we begin to discuss how to harness this computational performance.
- First by using single GPUs and at a later point, how to use multiple GPUs and multiple servers (with multiple GPUs).
    - Specifically, we will discuss how to use a single NVIDIA GPU for calculations.
    - First, make sure you have at least one NVIDIA GPU installed.
    - Then, download the NVIDIA driver and CUDA and follow the prompts to set the appropriate path.

GPUs

- Once these preparations are complete, the nvidia-smi command can be used to view the graphics card information.

```
!nvidia-smi
```

```
Wed Dec 14 05:37:30 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.106.00   Driver Version: 460.106.00   CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1B.0 Off |                    0 |
| N/A   23C    P0    48W / 300W |   1224MiB / 16160MiB |     10%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   23C    P0    49W / 300W |   1436MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   20C    P0    48W / 300W |      0MiB / 16160MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A     22466      C   ...l-en-release-1/bin/python     1221MiB |
+-----------------------------------------------------------------------------+
```


UNC CHARLOTTE

GPUs

- In PyTorch, the CPU and GPU can be indicated by torch.device('cpu') and torch.device('cuda’).
- It should be noted that the cpu device means all physical CPUs and memory.
- This means that PyTorch’s calculations will try to use all CPU cores.
- However, a gpu device only represents one card and the corresponding memory.
- If there are multiple GPUs, we use torch.device(f'cuda:{i}') to represent the i_{th} GPU (*i* starts from 0). Also, gpu:0 and gpu are equivalent.

Tensors and GPUs

- By default, tensors are created on the CPU.
- We can query the device where the tensor is located.

```
x = torch.tensor([1, 2, 3])
x.device
```

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

- There are several ways to store a tensor on the GPU:
- We can specify a storage device when creating a tensor.
- Next, we create the tensor variable X on the first gpu.
- Note: The tensor created on a GPU only consumes the memory of this GPU.
- We can use the nvidia-smi command to view GPU memory usage.
- We need to make sure that we do not create data that exceeds the GPU memory limit.

```
X = torch.ones(2, 3, device=try_gpu())
X
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

Neural Networks and GPUs

- Similarly, a neural network model can specify devices. The following code puts the model parameters on the GPU.

```
net = nn.Sequential(nn.LazyLinear(1))
net = net.to(device=try_gpu())
```

- We will see many more examples of how to run models on GPUs in future, simply since they will become somewhat more computationally intensive.
- When the input is a tensor on the GPU, the model will calculate the result on the same GPU.