Multilayer Perceptrons

- Last week: we introduced and implemented softmax regression to recognize 10 categories of clothing from images.
    - This model mapped inputs directly to outputs via a single linear transformation, followed by a softmax operation.
- Now that we have mastered the mechanics of simple linear models, we can launch our exploration of deep neural networks

Linear Models May Go Wrong

- If labels truly related to our input data by a linear transformation, then a linear model would be sufficient.
- But linearity is a *strong* assumption.

Examples

- **Example 1:** Predict whether an individual will repay a loan.
    - Use income as input feature
    - An individual with a higher income would be more likely to repay (than one with a lower income).
    - However, this relationship is not linearly associated with the probability of repayment.
        - An increase in income from 0 to 50 thousand likely corresponds to a bigger increase in likelihood of repayment than an increase from 1 million to 1.05 million.

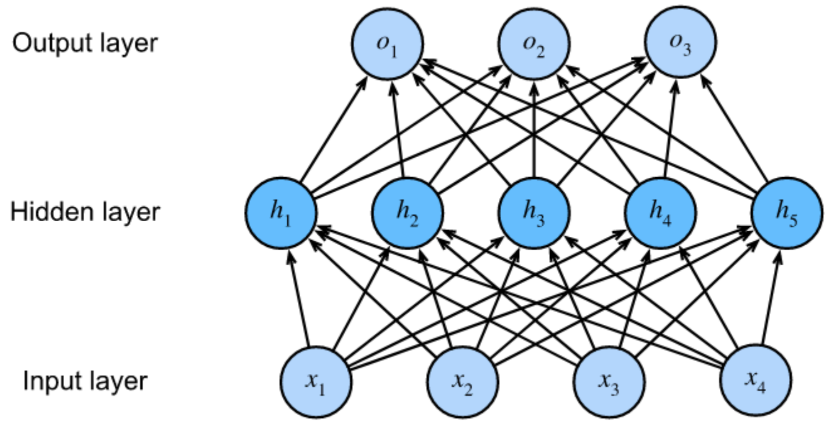
Feature Extraction

- One way to handle this might be to **preprocess** input data such that linearity is plausible; e.g. by using the logarithm of income as our feature.
    - Preprocess input data $\triangleq$ extract features from input data

- **Example 2:** Classify images of cats and dogs.
    - Relying on a linear model implicit assumes that for differentiating cats vs. dogs can be done using only the brightness of individual pixels.
    - Doesn't always work well
    - Less obvious that we could address the problem with a simple preprocessing fix.
- We can use deep neural networks, to jointly learn (from input data) both to extract features from the data and a classifier that acts upon the features.

Incorporating Hidden Layers

- We overcome the limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers.
- The easiest way to do this is to stack many fully-connected (linear) layers on top of each other.
- Each layer feeds into the layer above it, until we generate outputs.
- This architecture is commonly called a **Multi-Layer Perceptron** (**MLP**).



- The above MLP has 4 inputs, 3 outputs, and its hidden layer contains 5 hidden units.
- The number of layers in this MLP is 2.
  - The last (second) layer is the classifier.
  - The input to the last (second) layer are the **features**.
    - So the first layer calculates features from the input data.

The Maths of the Model

- The outputs $\mathbf{O} \in \mathbb{R}^{n \times q}$ of the previous one-hidden-layer MLP are calculated from:

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)},$$
$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)},$$

  - where the matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, denotes a minibatch of n examples with d inputs (features).
  - $\mathbf{X}$ is processed by the first hidden linear layer having weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$
  - The output of the hidden layer (i.e. the extracted features) is denoted by $\mathbf{H} \in \mathbb{R}^{n \times h}$.
  - $\mathbf{H}$ is processed by the second linear layer having weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$.

Equivalence to a Linear Model

- The hidden units above are given by a linear function of the inputs, and the outputs (pre-softmax) are just a linear function of the hidden units.
- Overall this is still a linear model with 2 layers
- It is equivalent to a single-layer linear model with parameters

$$\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)},$$

$$\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.$$

- Proof:

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}.$$

Adding Non-linearities

- To realize the potential of multilayer architectures, we need one more key ingredient: a **nonlinear activation function** σ to be applied to each hidden unit after the linear transformation.

- With activation functions in place, it is no longer possible to collapse an MLP into a linear model:

$$\begin{aligned} \mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned}$$

- Typically, the activation functions are applied elementwise.

- The outputs of activation functions are called **activations** or **features**.
- To build more general MLPs, we can continue stacking such hidden layers, e.g., $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ and $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, one on top of the other, yielding ever more expressive models.

Universal Approximators

- MLPs can capture complex interactions among their inputs via their hidden neurons.
- For example, we can easily design hidden nodes to perform basic logic operations on a pair of inputs.
- Moreover, for certain choices of the activation function, it is widely known that MLPs are universal approximators.
    - Even with a single-hidden-layer network, given enough nodes, and the right set of weights, we can model any function
    - However, actually finding the weights is the hard part.
- We can learn functions a lot more easily from data by using deep networks.

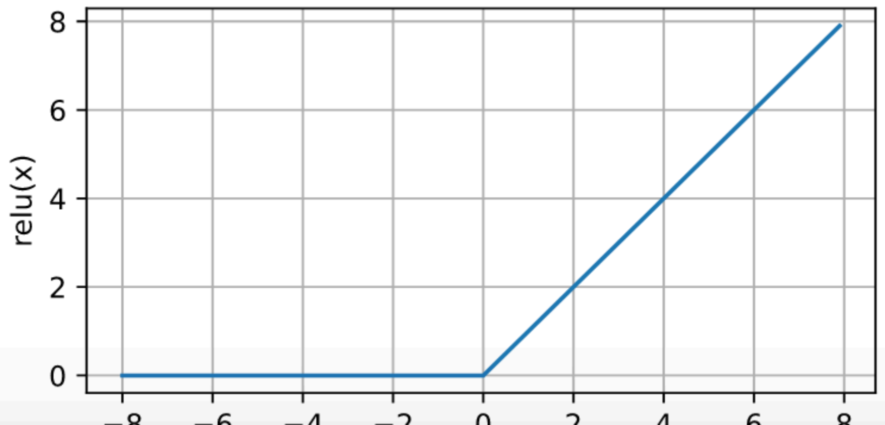
Activation Functions

- Activation functions decide whether a neuron should be activated.
- They are differentiable operators.

ReLU Function

- The most popular choice due to its great performance on a variety of predictive tasks, is the **Rectified Linear Unit (ReLU)**.
- ReLU provides a very simple nonlinear transformation. It is defined as:
$$\operatorname{ReLU}(x) = \max(x, 0).$$
- ReLU retains only positive elements and `discards' all negative elements.

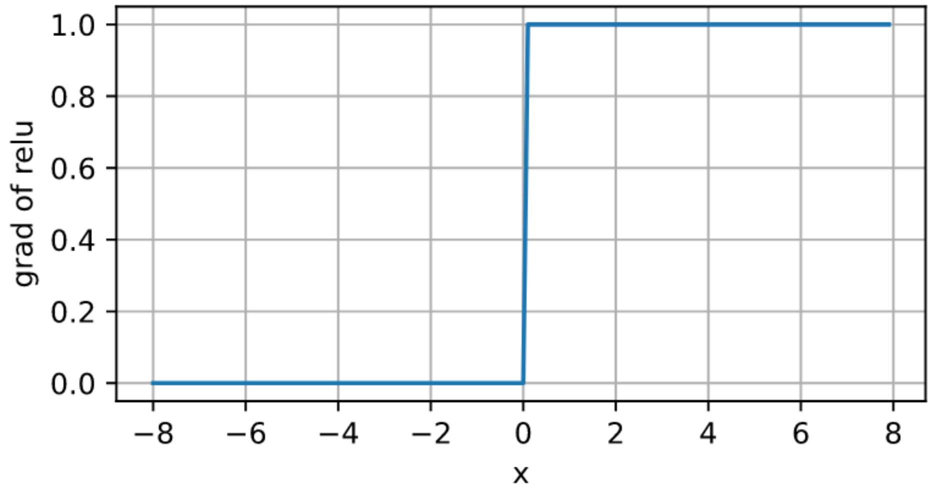
In [2]:
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
mu.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



ReLU Function

- When $x < 0$, the derivative of the ReLU is 0, and when $x > 0$, it is equal to 1.
- ReLU is not differentiable when $x = 0$: we just set the derivative to 0.

In [3]:
```
y.backward(torch.ones_like(x), retain_graph=True)
mu.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



- ReLU works well because its derivatives are well-behaved: either they vanish or they let the argument through.
- This makes optimization better behaved and mitigates the known problem of vanishing gradients
- *Parameterized ReLU* (*pReLU*) allows information to flow, even when $x < 0$:

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

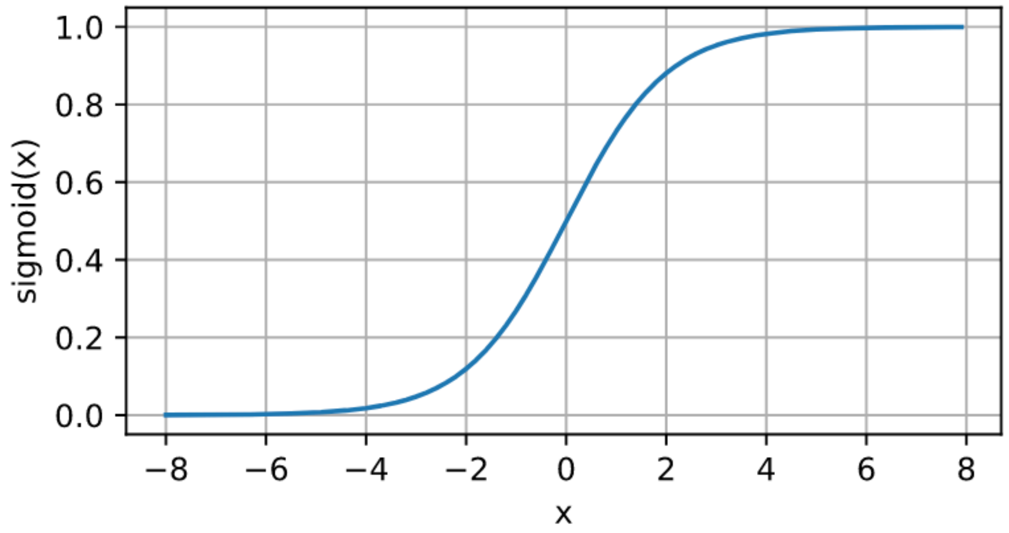
Sigmoid Function

- The **sigmoid function** transforms its input to output that lies on the interval (0, 1):
    - Also called a *squashing function*

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

- An approximation to step function used to model biological neurons which either *fire* or *do not fire.*
- In constrast to step function, the sigmoid is smooth, differentiable

In [4]:
```
y = torch.sigmoid(x)
mu.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

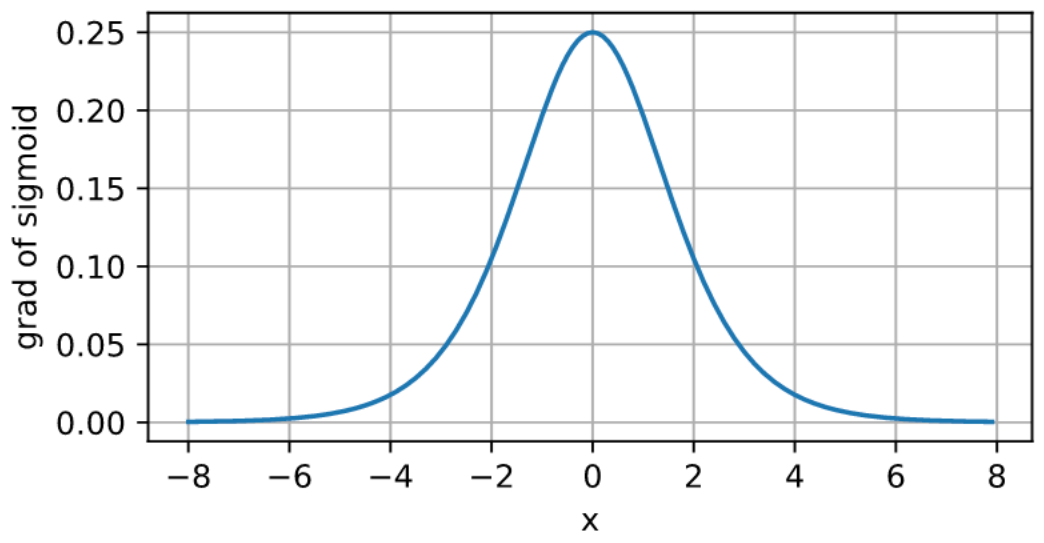


Sigmoid Function

- The derivative of the sigmoid function is given by :
$$\frac{d}{dx}\operatorname{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \operatorname{sigmoid}(x)\left(1 - \operatorname{sigmoid}(x)\right).$$

- As the input diverges from 0 in either direction, the derivative approaches 0.
    - Compare this to ReLU!

In [5]:
```
# Sigmoid derivative
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
mu.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

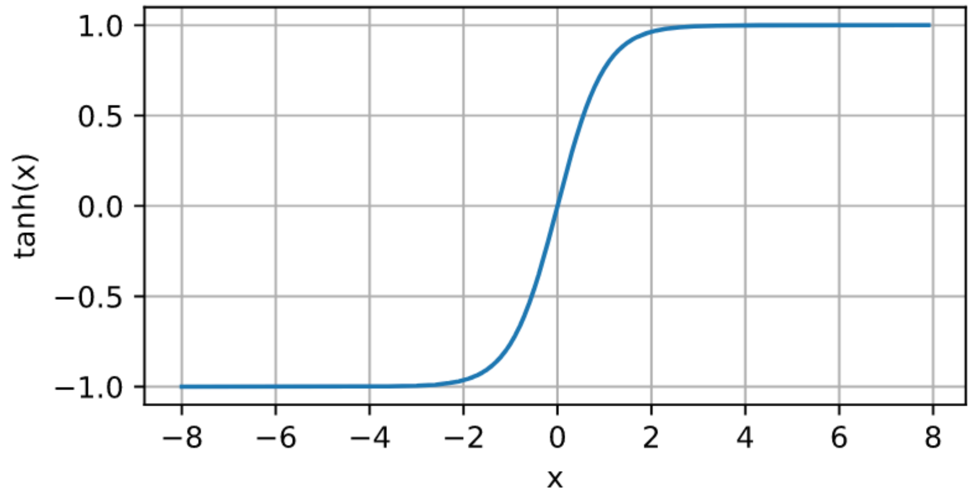


Tanh Function

- Similar to sigmoid. Also squashes its inputs to $[-1, 1]$:
$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

- The tanh function exhibits point symmetry about the origin of the coordinate system.

In [6]:
```python
y = torch.tanh(x)
mu.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



- The derivative of the tanh function is: $\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$.

Summary

- MLP adds one or multiple fully-connected hidden layers between the output and input layers and transforms the output of the hidden layer via an activation function.
- Commonly-used activation functions include the ReLU function, the sigmoid function, and the tanh function.