# Quantization-aware Training

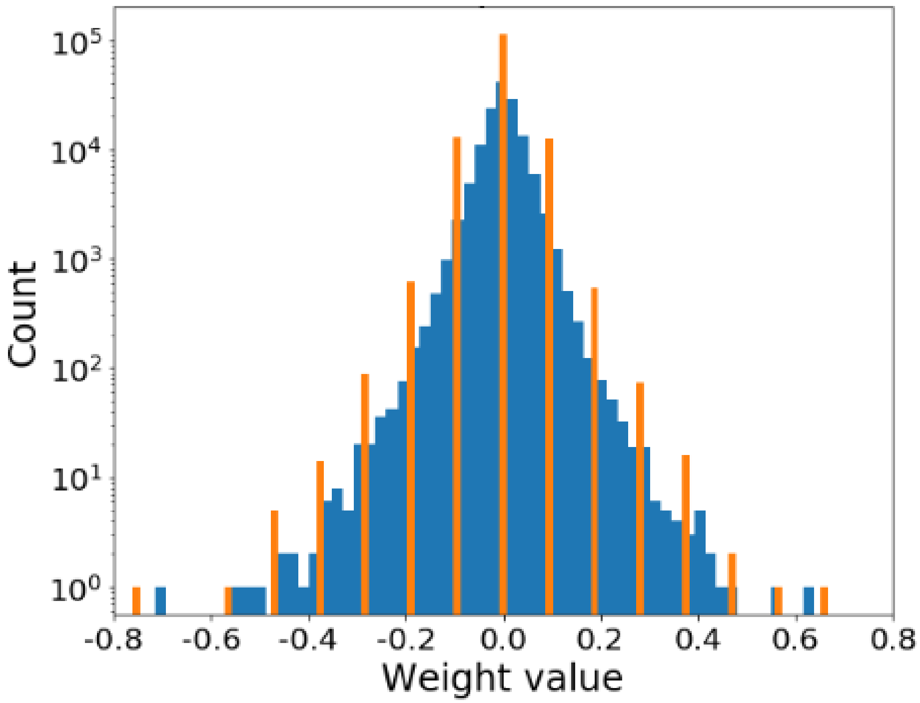
Hardware System Design

Spring, 2023

# What is Quantization?

- Process of reducing the precision of a neural network
  - weights / activations
  - FP32 to lower-precision integer (INT8, INT4, …)

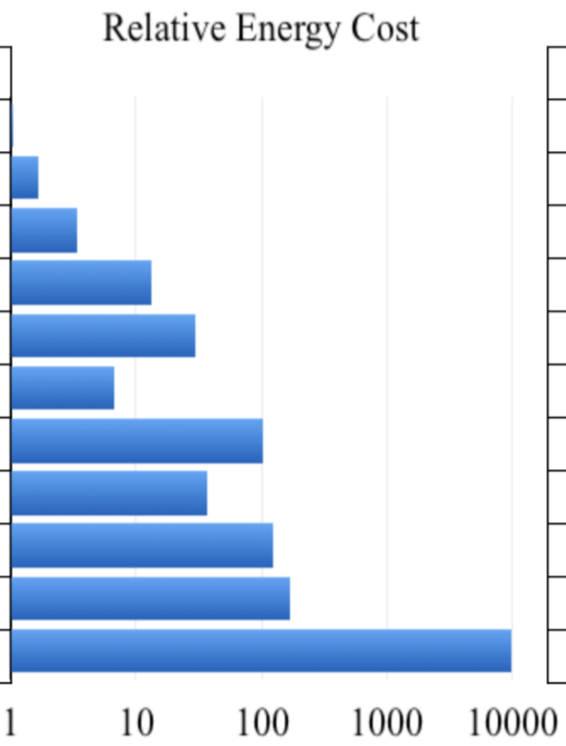
- Divides the range of floating-point values into a fixed number of discrete levels.
  - In 4-bit quantization, the range of values is divided into 16 levels.

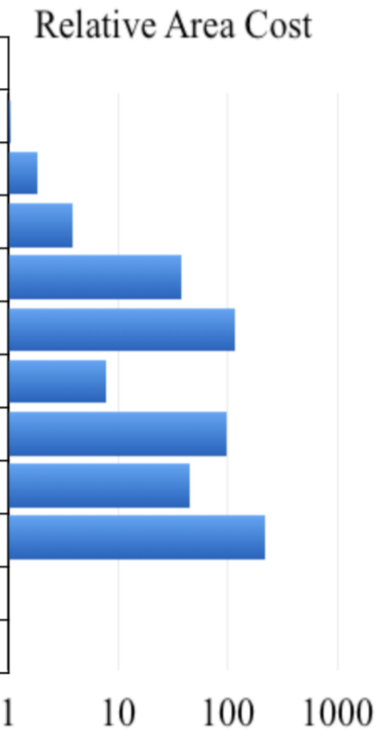


# What is Quantization?

- Reduces the memory footprint and computational complexity.
  - Makes it possible to run the model on low-power devices with limited resources.

| Operation: | Energy (pJ) | Area ($\mu m^2$) |
|---|---|---|
| 8b Add | 0.03 | 36 |
| 16b Add | 0.05 | 67 |
| 32b Add | 0.1 | 137 |
| 16b FP Add | 0.4 | 1360 |
| 32b FP Add | 0.9 | 4184 |
| 8b Mult | 0.2 | 282 |
| 32b Mult | 3.1 | 3495 |
| 16b FP Mult | 1.1 | 1640 |
| 32b FP Mult | 3.7 | 7700 |
| 32b SRAM Read (8KB) | 5 | N/A |
| 32b DRAM Read | 640 | N/A |




Energy numbers are from Mark Horowitz "Computing's Energy Problem (and what we can do about it)", ISSCC 2014
Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

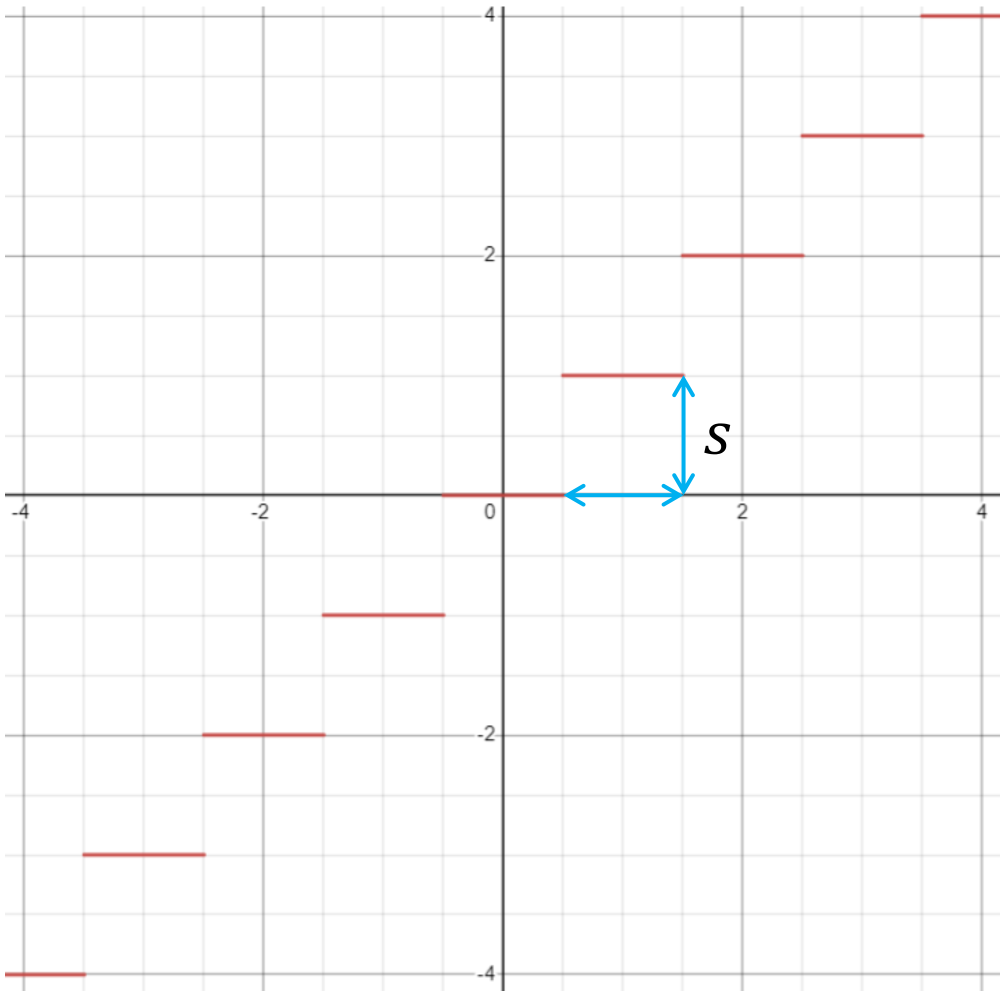
# What is Quantization?

- Can introduce quantization errors
  - Leads to a loss of accuracy.
  - Various techniques have been developed to mitigate this.

Table 1: Top-1 / Top-5 accuracy [%] of the quantized networks on ImageNet.

| | MobileNet-v1 | MobileNet-v2 | MobileNet-v3 | MNasNet-A1 |
|---|---|---|---|---|
| **Full** | 68.848 / 88.740 | 71.328 / 90.016 | 74.728 / 92.136 | 73.130 / 91.276 |
| **Full+** | 69.552 / 89.138 | 71.944 / 90.470 | 75.296 / 92.446 | 73.396 / 91.464 |
| **8-bit** | 70.164 / 89.370 | 72.352 / 90.636 | 75.166 / 92.498 | 73.742 / 91.756 |
| **5-bit** | 69.866 / 89.058 | 72.192 / 90.498 | 74.690 / 92.092 | 73.378 / 91.244 |
| **4-bit** | 69.056 / 88.412 | 71.564 / 90.398 | 73.812 / 91.588 | 72.244 / 90.584 |

# What is Quantization?




$y = Q(x)$ when $s = 1$

- Values are mapped to the nearest quantized values

$$\boxed{Q(x) = \left[\frac{x}{s}\right] \cdot s}$$

- $s$ = step size

# Types of Quantization

- QAT (Quantization-Aware Training)
  - Adds a quantization step to the training process
  - Model learns to become more robust to the effects of quantization early on.
  - More precise & accurate results
  - Computationally expensive

- PTQ (Post-Training Quantization)
  - Quantizes the model after the training process is complete.
  - Faster
  - Less computationally expensive.
  - May lead to a loss of accuracy

# Types of Quantization

- QAT (Quantization-Aware Training)
  - Adds a quantization step to the training process
  - Model learns to become more robust to the effects of quantization early on.
  - More precise & accurate results
  - Computationally expensive

For today's lab & final project

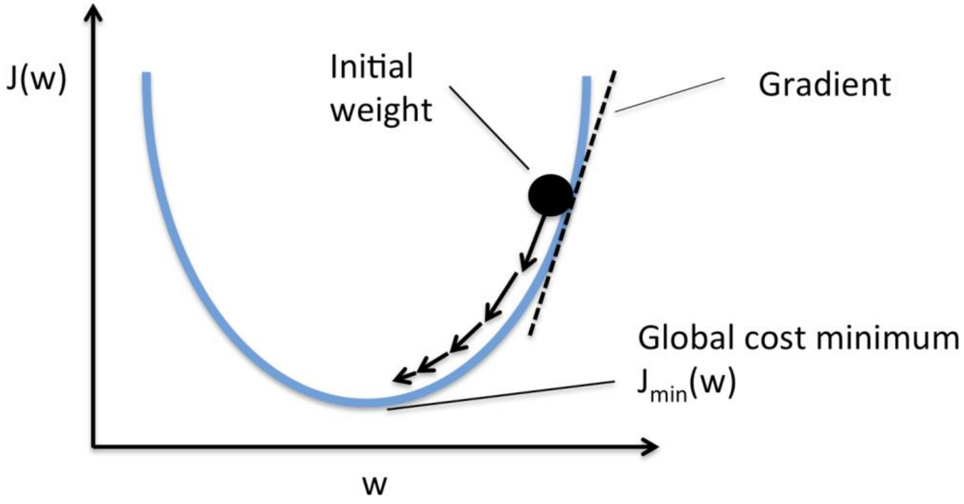
- PTQ (Post-Training Quantization)
  - Quantizes the model after the training process is complete.
  - Faster
  - Less computationally expensive.
  - May lead to a loss of accuracy

# Backpropagation

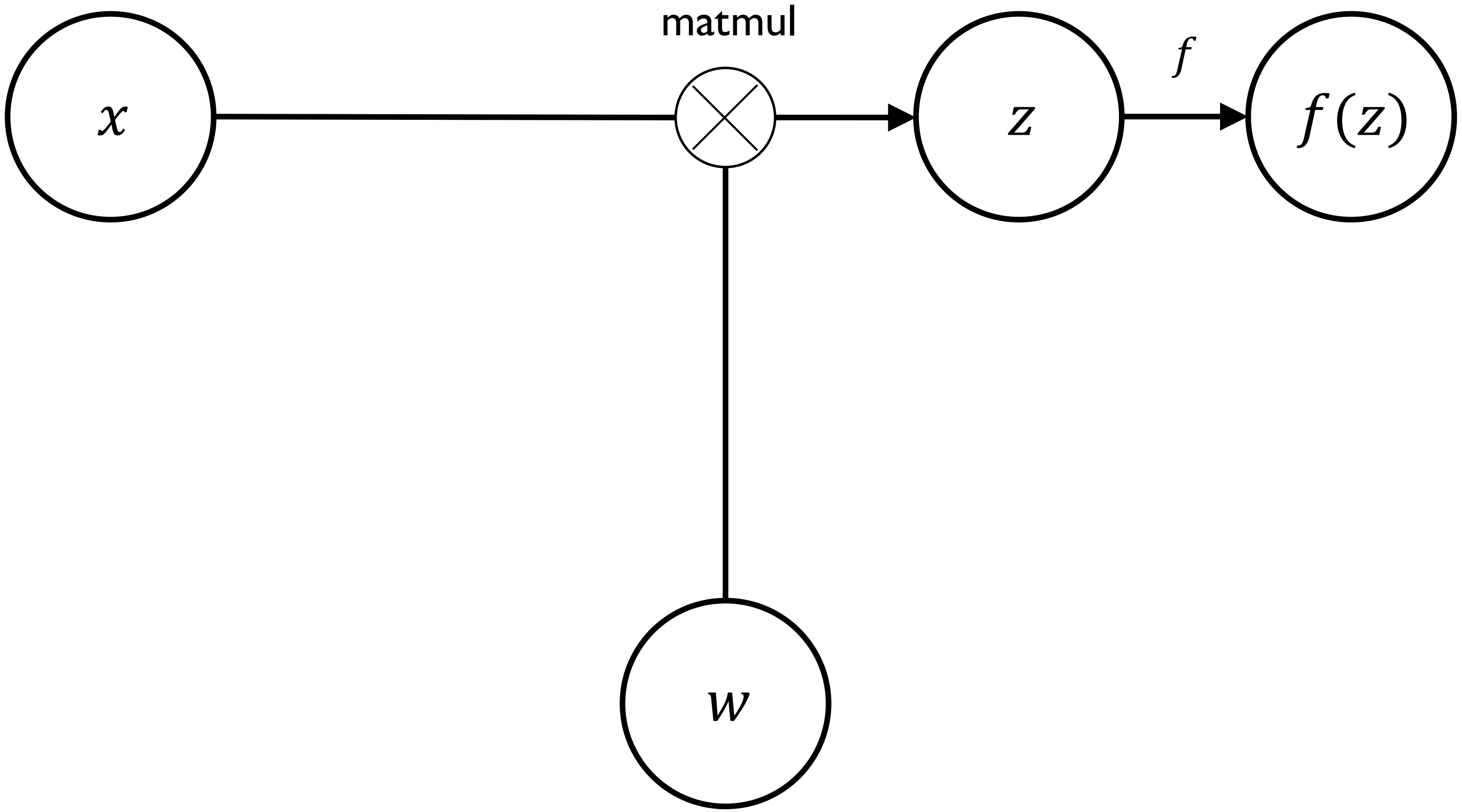
- Backpropagation
  - Computes gradients for all the weights.
  - From the last to first layers.

- All operations should be differentiable.
  - Matmul is differentiable.
  - ReLU is differentiable.
  - What about quantization function?

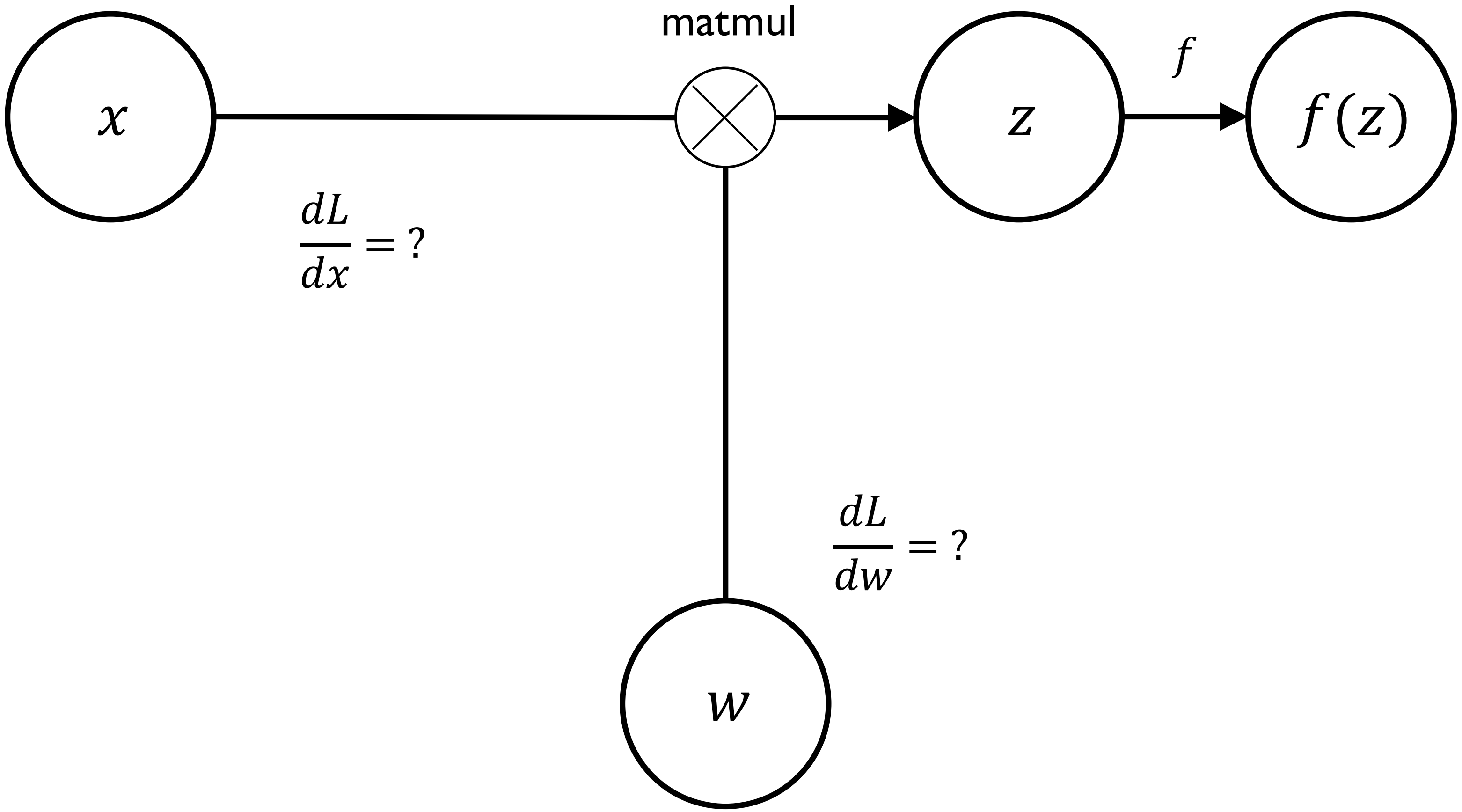
$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, \qquad \Delta w_j = -\eta \frac{\partial J}{\partial w_j},$$



# Backpropagation

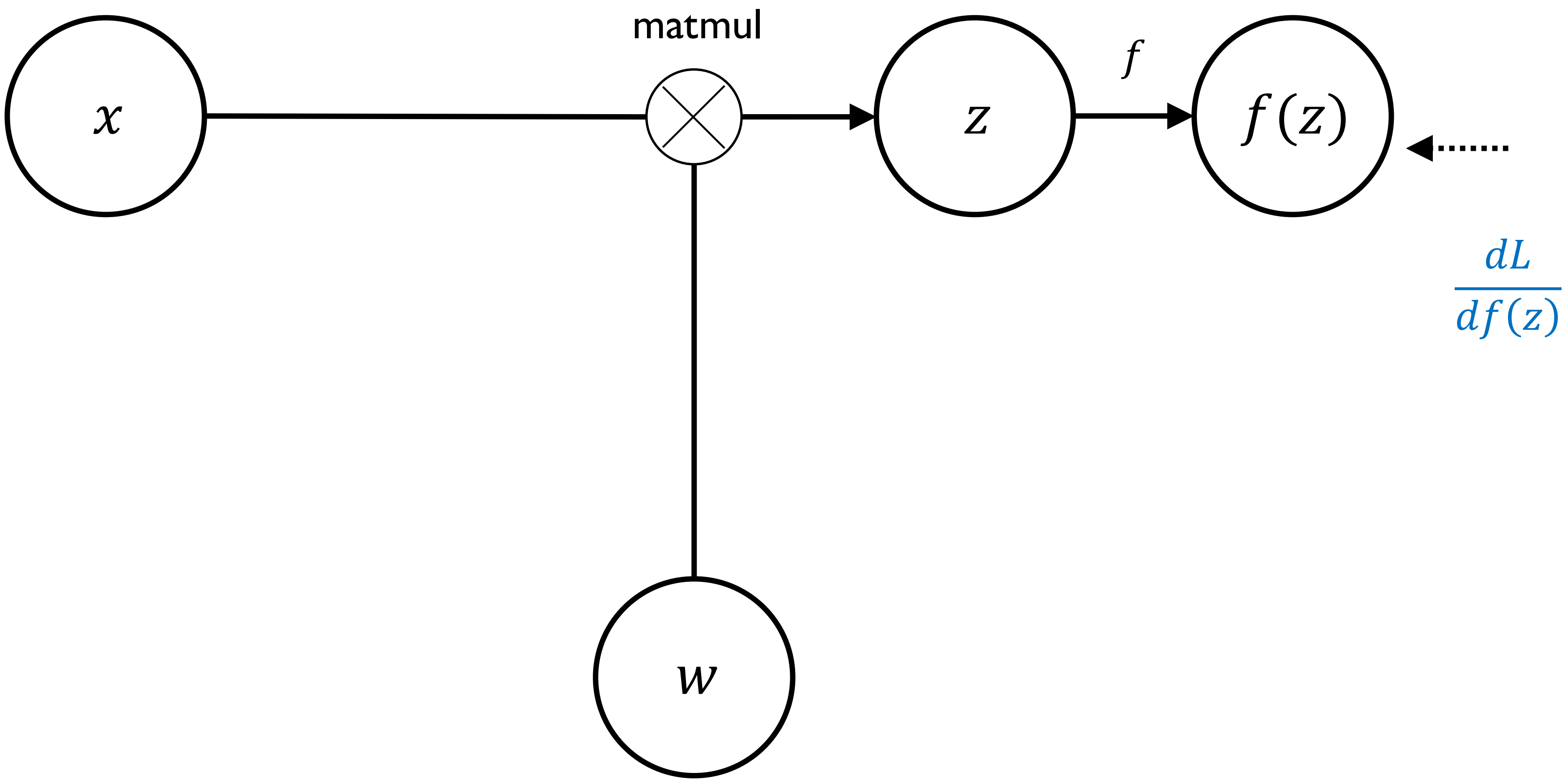


# Backpropagation

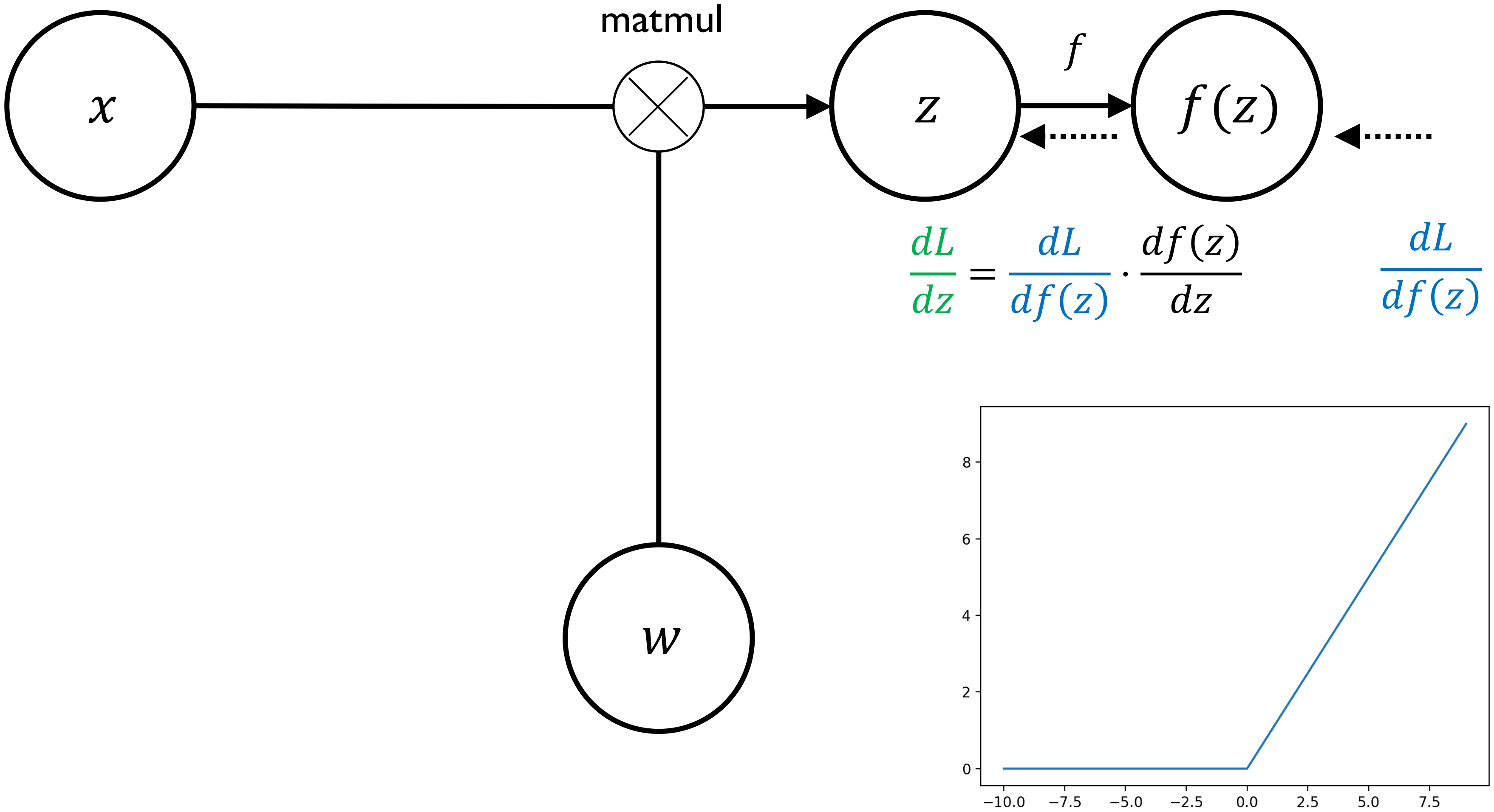



- To optimize w, we should find $\frac{dL}{dx}$ and $\frac{dL}{dw}$.

# Backpropagation



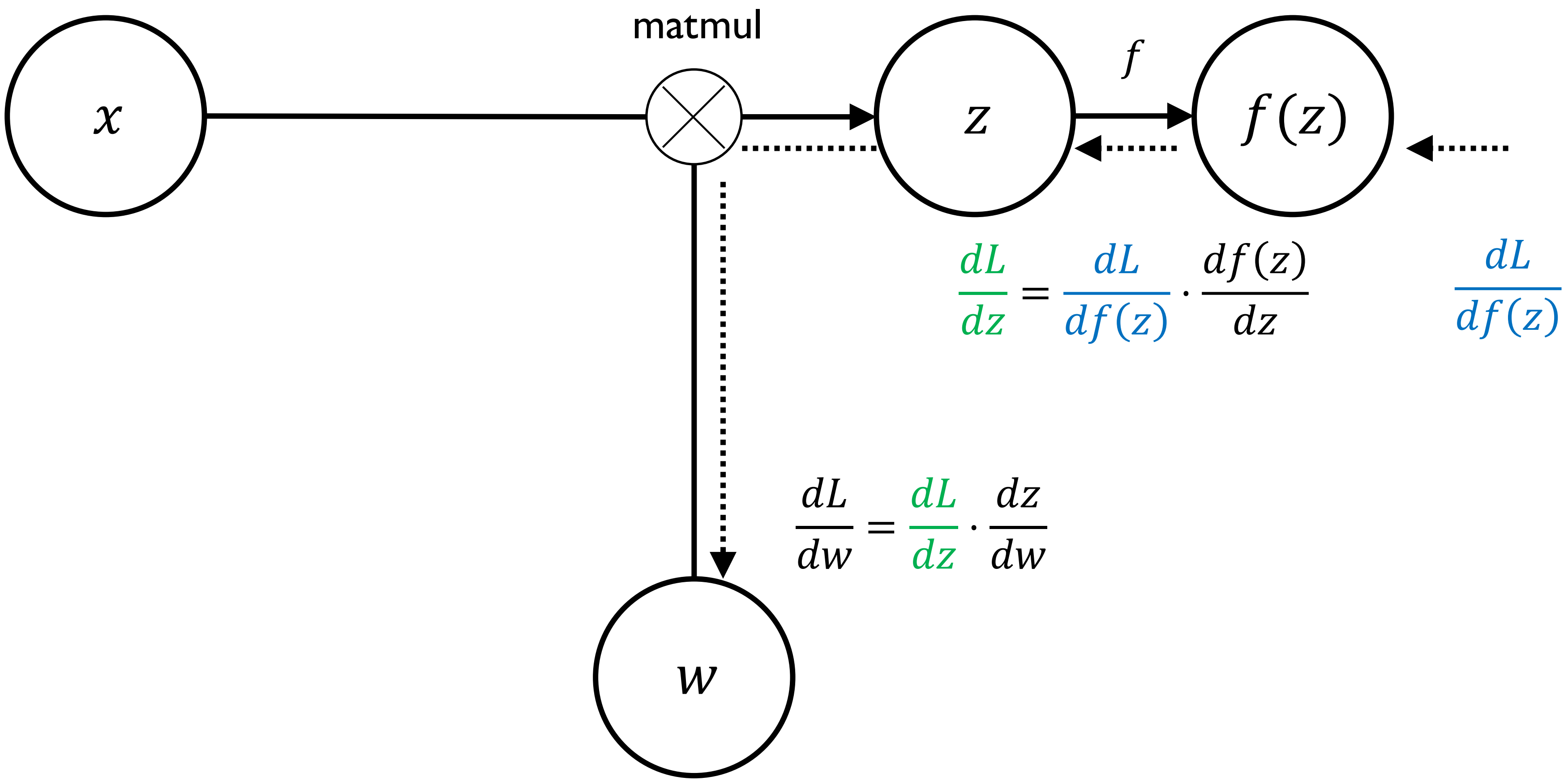
# Backpropagation



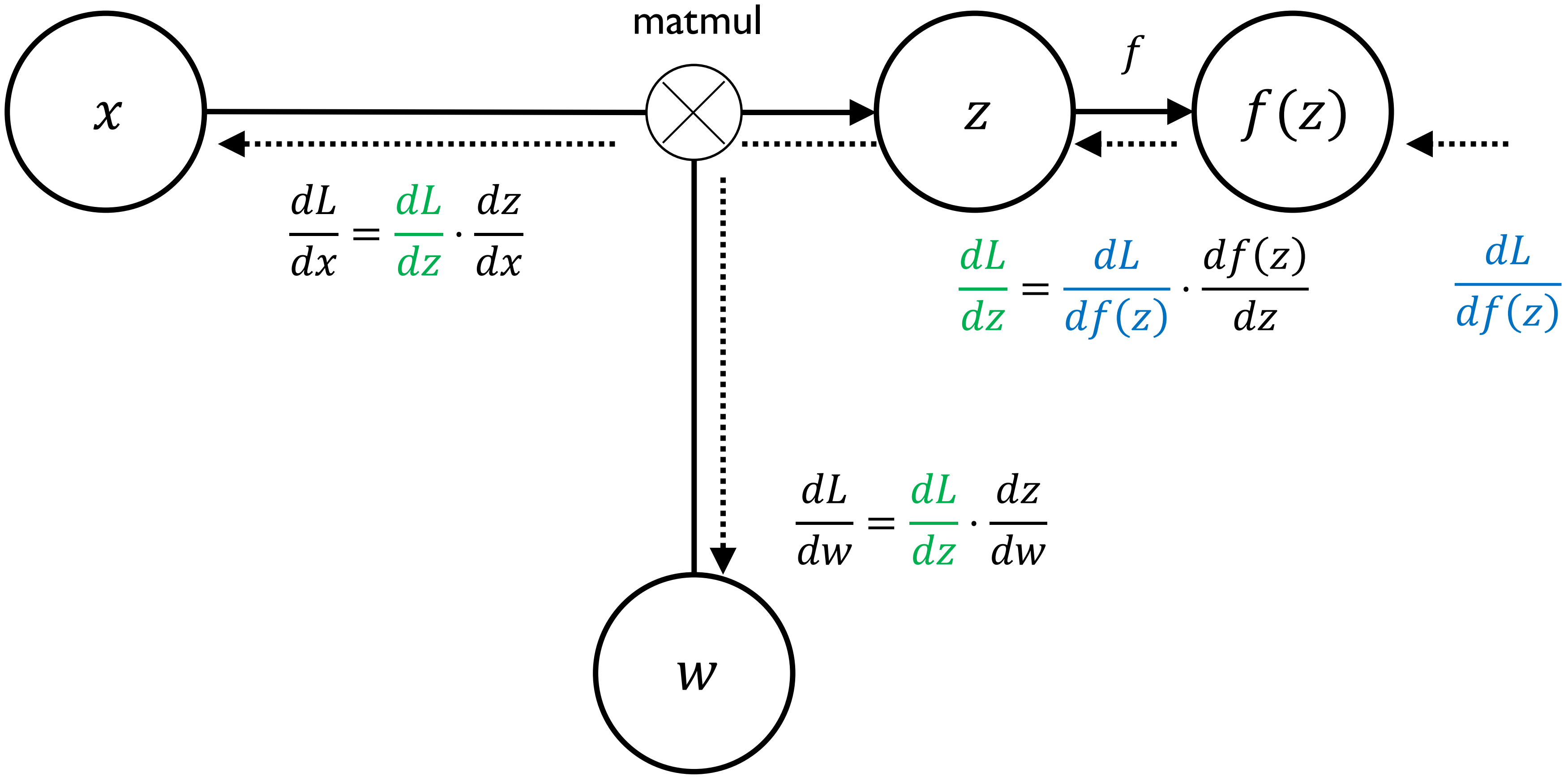

$$\frac{dL}{dz} = \frac{dL}{df(z)} \cdot \frac{df(z)}{dz}$$

$$\frac{dL}{df(z)}$$

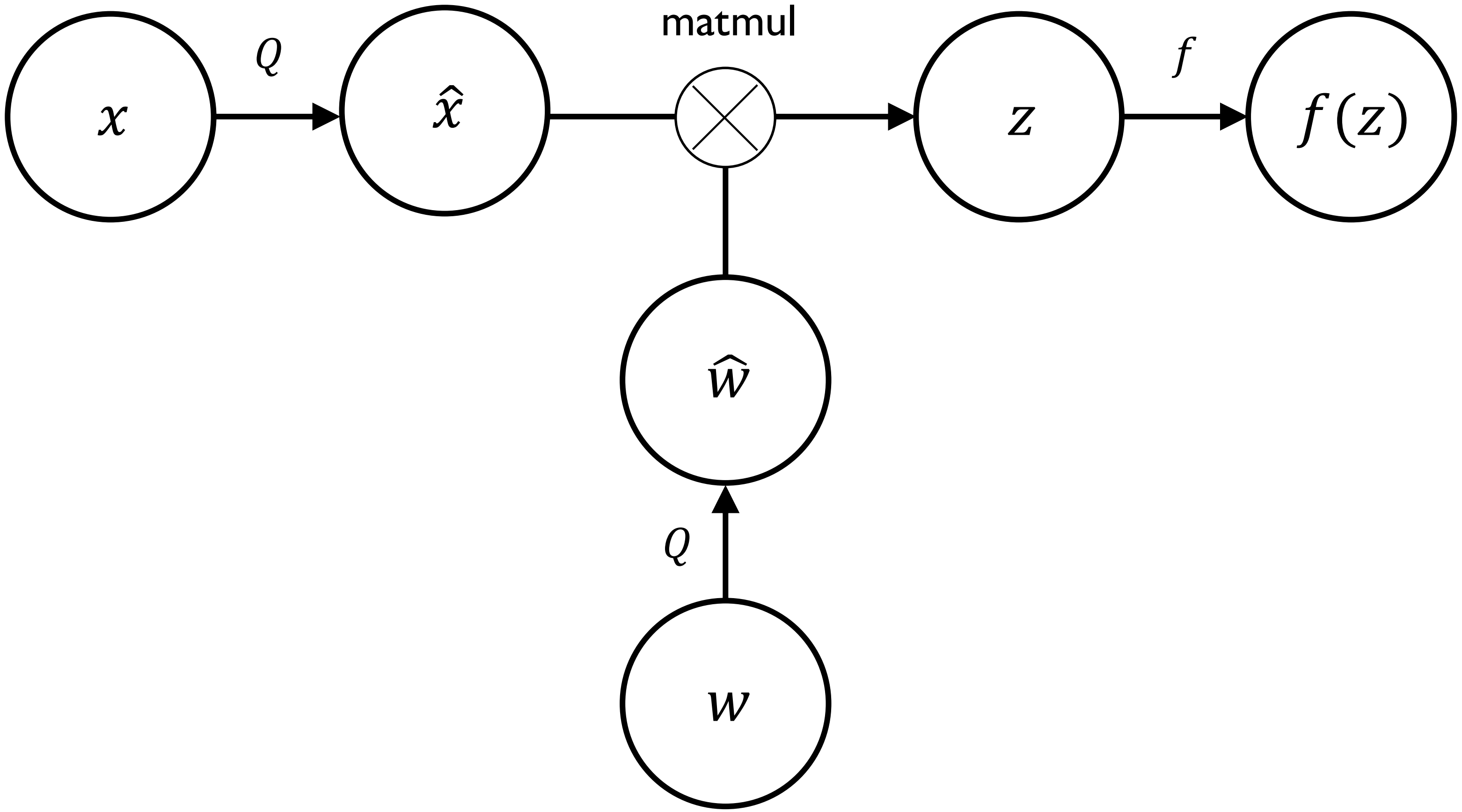
# Backpropagation



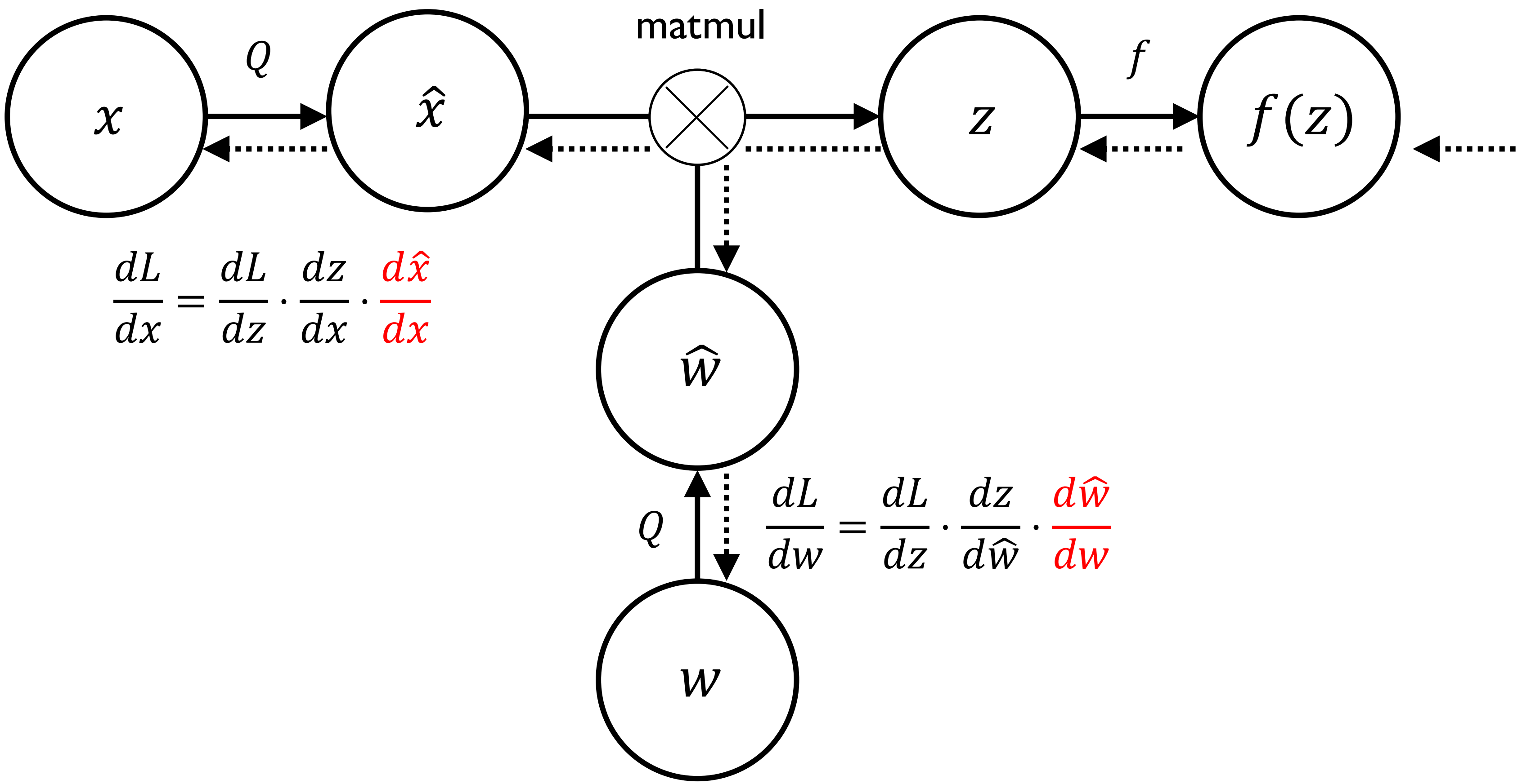
# Backpropagation



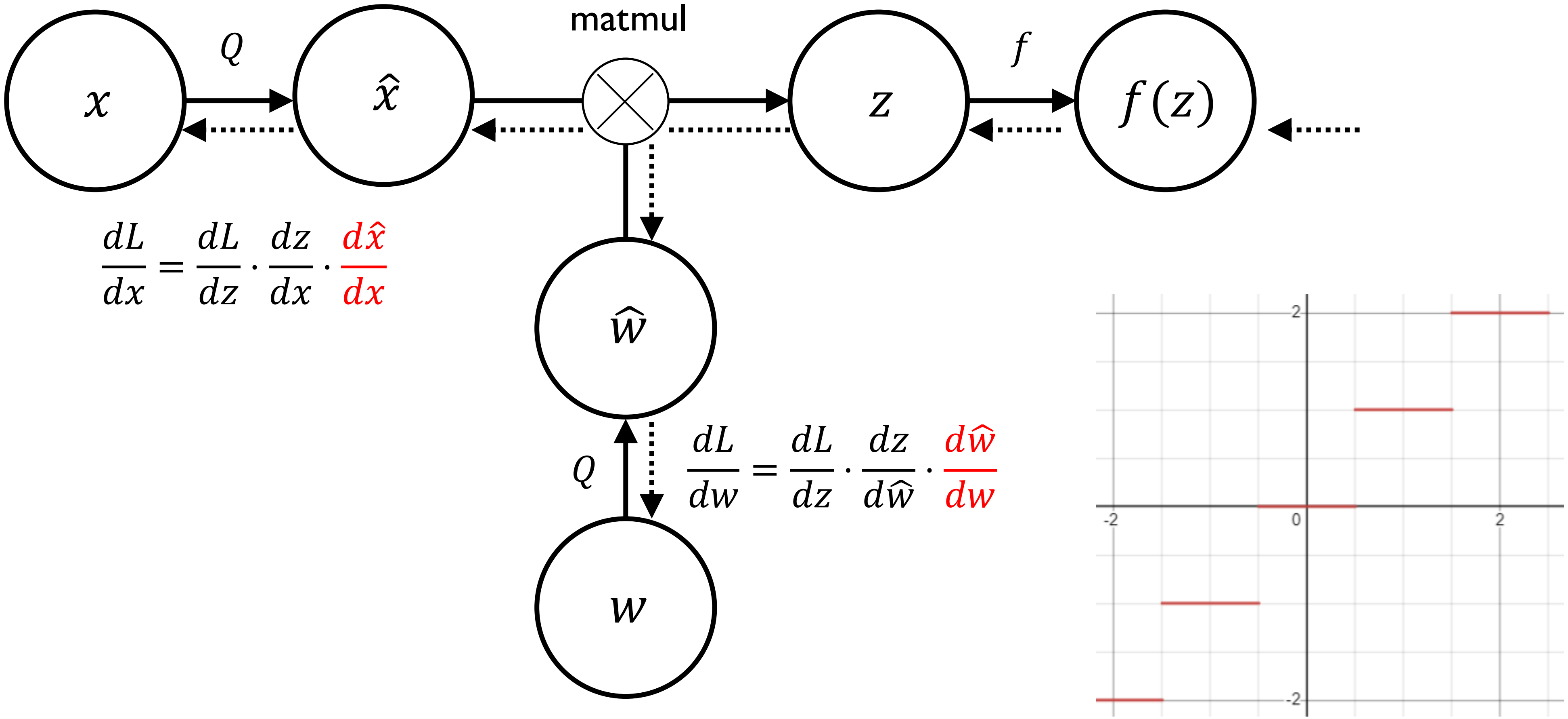
# Backpropagation



# Backpropagation



# Backpropagation




$$\frac{dL}{dx} = \frac{dL}{dz} \cdot \frac{dz}{dx} \cdot \frac{d\hat{x}}{dx}$$

$$\frac{dL}{dw} = \frac{dL}{dz} \cdot \frac{dz}{d\widehat{w}} \cdot \frac{d\widehat{w}}{dw}$$

- Since Q is not differentiable, $\frac{d\widehat{w}}{dw}$ does not exist.

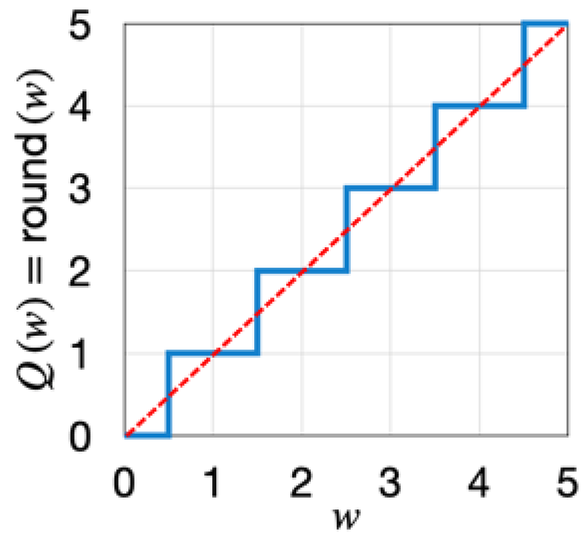
# STE (Straight-Through Estimator)

- How to backpropagate through quantization function?
  - Not differentiable -> true gradient is undefined.

- STE allows gradients to be backpropagated through a non-differentiable function.
  - Uses a surrogate gradient that approximates the true gradient.

- Replaces the non-differentiable function with a differentiable function.
  - Identity function ($y = x$) for quantization function.
  - Forward : $y = \left\lfloor \frac{x}{s} \right\rceil \cdot s$
  - Backward : $y = x \Longrightarrow \frac{dy}{dx} = 1$

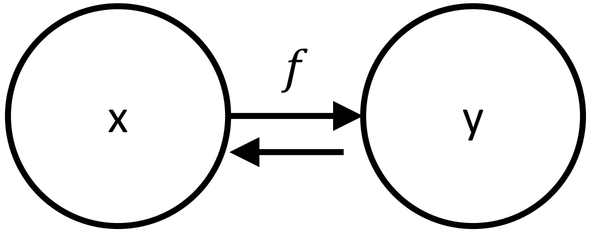
blue line : forward
red dots : backward



# STE – custom autograd

- **torch.autograd.Function** allows us to
  - customize autograd operations.
  - implement the forward and backward passes for their own tensor operations.

- Without STE

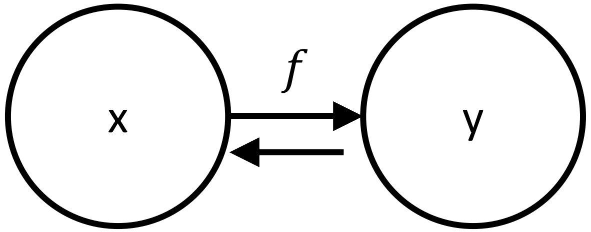



$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$$

# STE – custom autograd

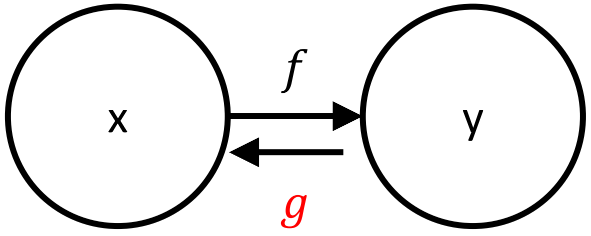
- **torch.autograd.Function** allows us to
  - customize autograd operations.
  - implement the forward and backward passes for their own tensor operations.

- Without STE




$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$$

- With STE



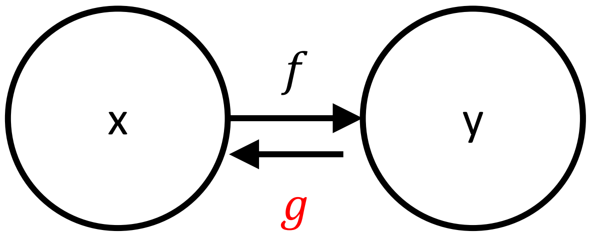

$$\frac{dL}{dx} = g(\frac{dL}{dy})$$

# STE – custom autograd

- **torch.autograd.Function** allows us to
  - customize autograd operations.
  - implement the forward and backward passes for their own tensor operations.

```
class STE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x):
        return f(x)

    @staticmethod
    def backward(ctx, dLdy):
        return g(dLdy)
```

- With STE

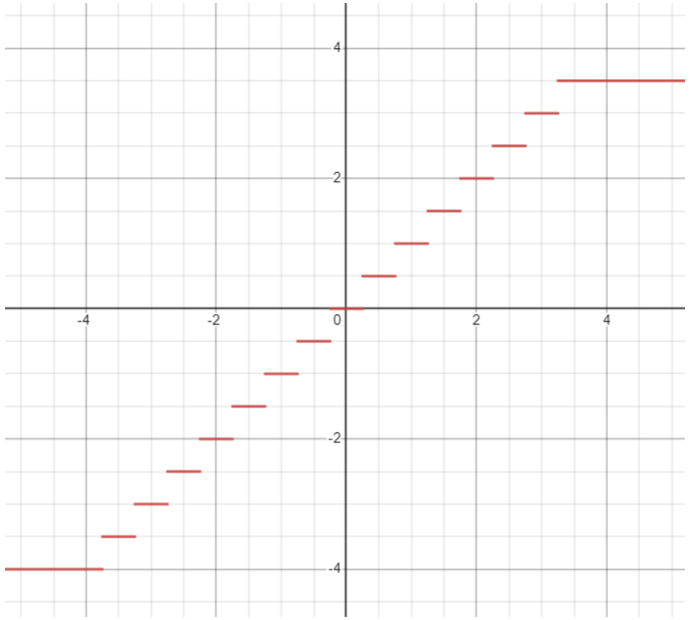



$$\frac{dL}{dx} = g\left(\frac{dL}{dy}\right)$$

# Quantization Function

$$q_x = clip\left(\left\lceil \frac{x - off}{s} \right\rfloor, Q_n, Q_p\right) \cdot s + off$$

- $clip$ : a function for truncation
- $off$ : offset value
- $s$ : step size
- $Q_n$ : smallest bit value (-128 for INT8, 0 for UINT8)
- $Q_p$ : largest bit value (127 for INT8, 255 for UINT8)




- INT4 Quantization

$(s = 0.5, off = 0, Q_n = -8, Q_p = 7)$

# Quantization Function – Fake quantization

integer

$$q_x = \boxed{clip\left(\left\lfloor\frac{x - off}{s}\right\rceil, Q_n, Q_p\right)} \cdot s + off$$

rescaled back to the original range

- Quantization operation is "*simulated*" <u>during training</u>.
  - without actually quantizing the weights or activations.
- The actual quantization operation is applied <u>during inference</u>.
  - when the network is deployed on a hardware platform with limited bit precision.

# Quantization Function – Fake quantization

rounding function $\lfloor \cdot \rceil$ is not differentiable

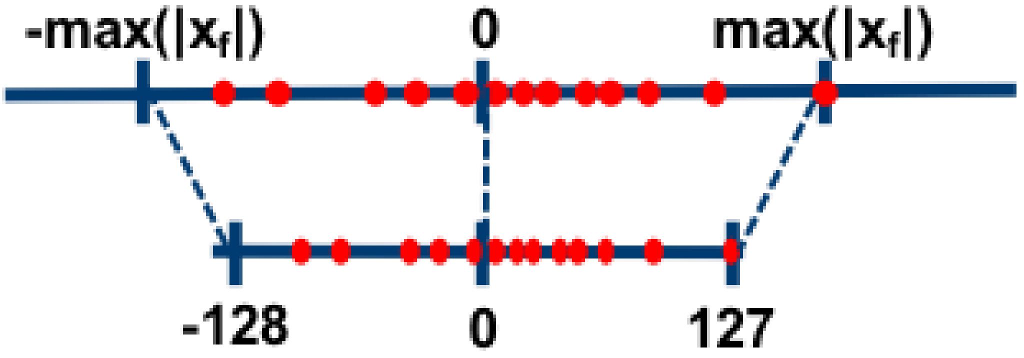
$$q_x = clip\left(\left\lfloor \frac{x - off}{s} \right\rceil, Q_n, Q_p\right) \cdot s + off$$

- The only part that requires the use of STE is the rounding function.

```
class roundpass(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        ## TODO ##
        ## Define output w.r.t. input
        return output

    @staticmethod
    def backward(ctx, grad_output):
        ## TODO ##
        ## Define grad_input w.r.t. grad_output
        return grad_input
```

# Quantization Function – Quantization range

$$q_x = clip\left(\left\lfloor\frac{x - off}{s}\right\rceil, Q_n, Q_p\right) \cdot s + off$$




- How to determine the step size s?
- A naïve approach
  - $\alpha = \max(|x|)$
  - $s = \alpha / 2^{bits-1}$

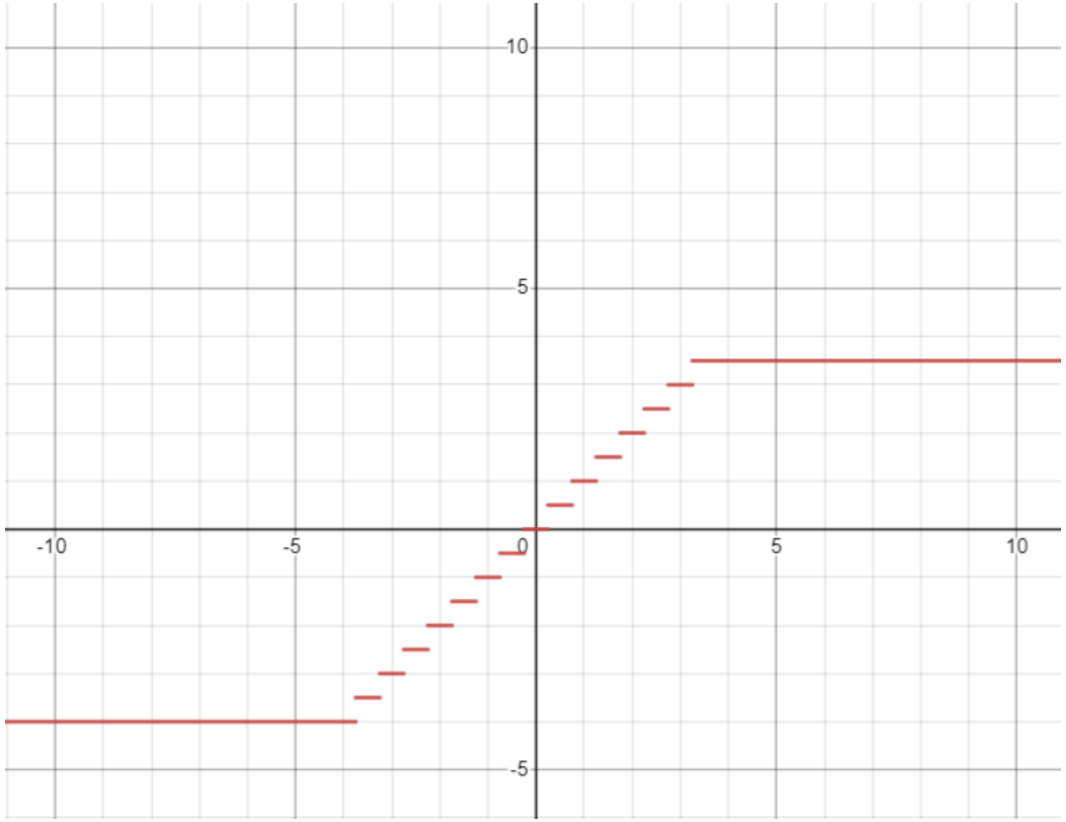
# Quantization Function – Quantization range

$$q_x = clip\left(\left[\frac{x - off}{s}\right], Q_n, Q_p\right) \cdot s + off$$

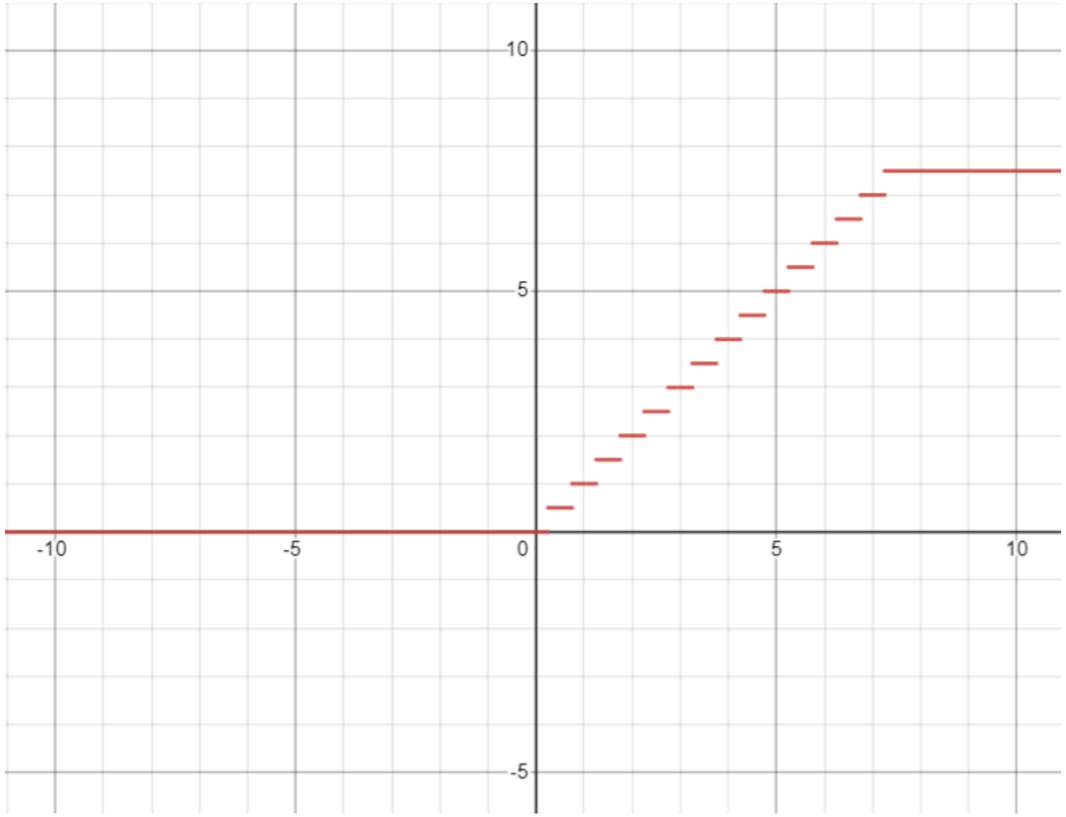
- How to determine the step size s?
- A naïve approach
  - $\alpha = \max(|x|)$
  - $s = \alpha / 2^{bits-1}$

```
class Quantizer(nn.Module):
    def __init__(self, bits=8, always_pos=False):
        super(Quantizer, self).__init__()

        self.first = True
        self.num_steps = 2 ** bits
        self.always_pos = always_pos

        self.Qp = 2**(bits-1) - 1
        self.Qn = - 2**(bits-1)

    def forward(self, x):
        if self.first:
          self.alpha = x.abs().max().detach()
          self.first = False

        step_size = 2 * self.alpha / self.num_steps
        if self.always_pos:
            off = self.alpha
        else:
            off = 0

        ## TODO ##
        ## define q_x given x and other components
above.
        return q_x
```

# Quantization Function – Offset

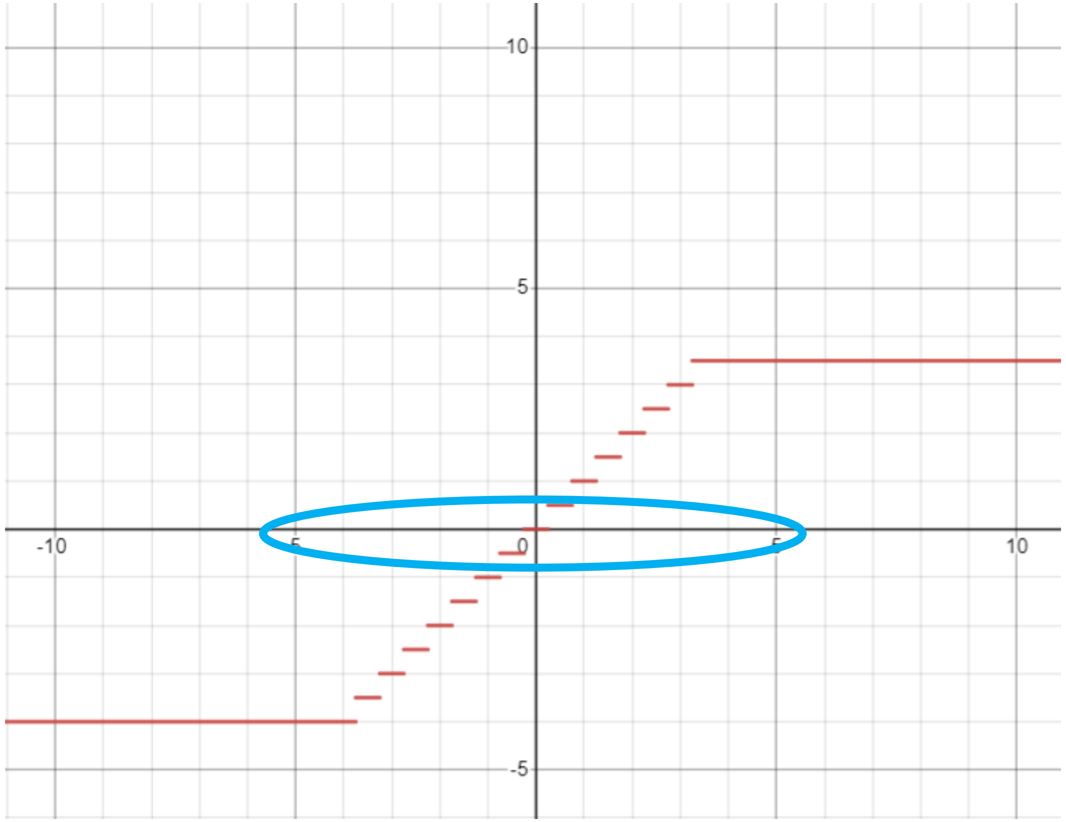



off=0



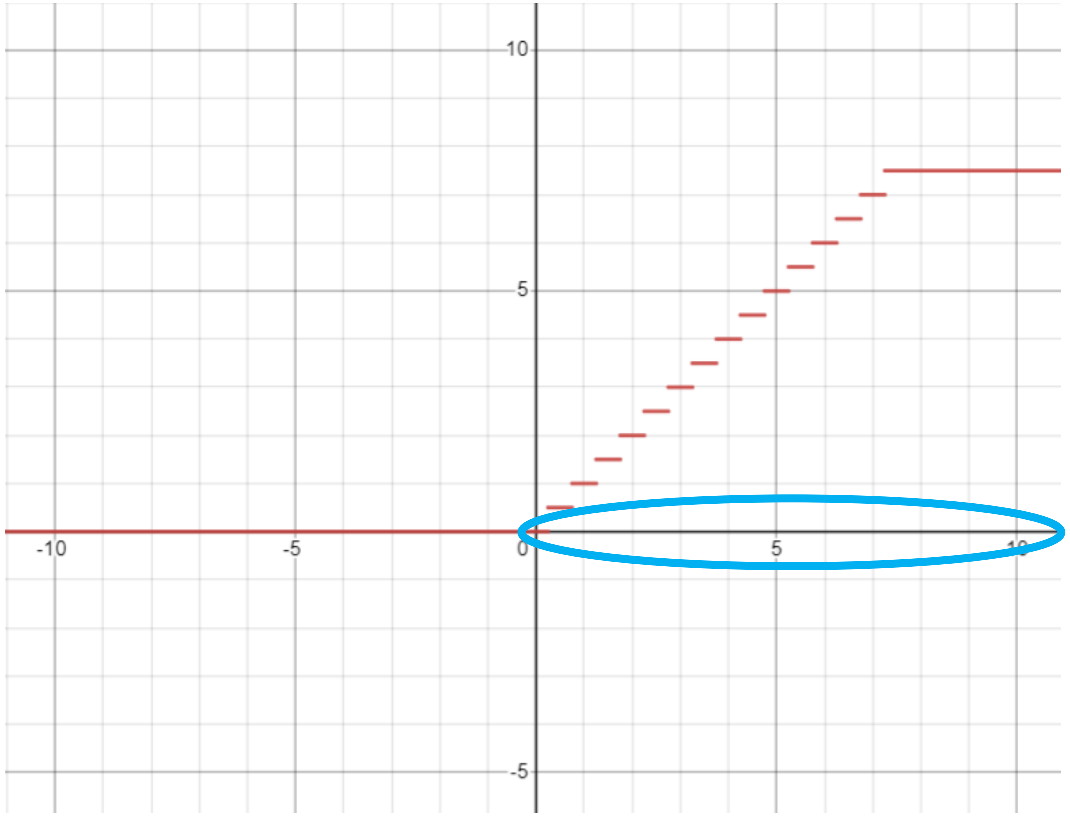

off=4

# Quantization Function – Offset




off=0

for weight




off=4

for activation (after ReLU)

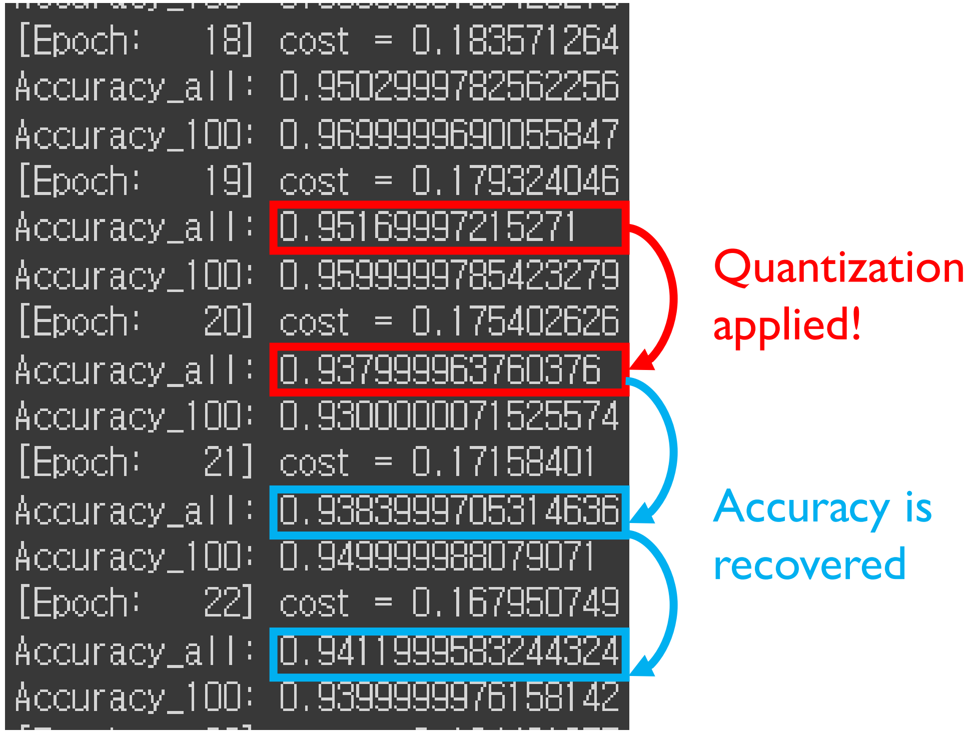
# Applying Quantization

- Now we are ready to apply quantization to activations/outputs.
- In the previous lab, we used nn.Linear class from Pytorch.
- To apply quantization, we should manually define a module that subclasses nn.Linear.

```
class CustomLinear(nn.Linear):
    def __init__(self, *args, **kwargs):
        super(CustomLinear, self).__init__(*args,
**kwargs)
        self.q_w = Quantizer()
        self.q_a = Quantizer(always_pos=True)
        self.is_quant = False


    def forward(self, x):
      if self.is_quant:
          ## TODO ##
          ## quantize the weights and inputs using the
``Quantize`` modules.

      else:
          weight = self.weight
          inputs = x

      return F.linear(inputs, weight, bias=self.bias)
```

# Applying Quantization – expected result

- Training will be done for
  - 20 epochs under FP32
  - 20 epochs under INT8.
- Accuracy may be dropped as quantization is applied, but will be recovered soon.



# Today's assignment

- Complete the codes for
  1) Defining STE
  2) Defining quantization function
  3) Applying quantization
- Only modifying ## TODO ## section is allowed.
- Don't hesitate to ask questions
  - It requires knowledge of (very basic) python and the previous lab.