Intro to Deep Learning

Abhinav Bhatele, Daniel Nichols

UNIVERSITY OF MARYLAND

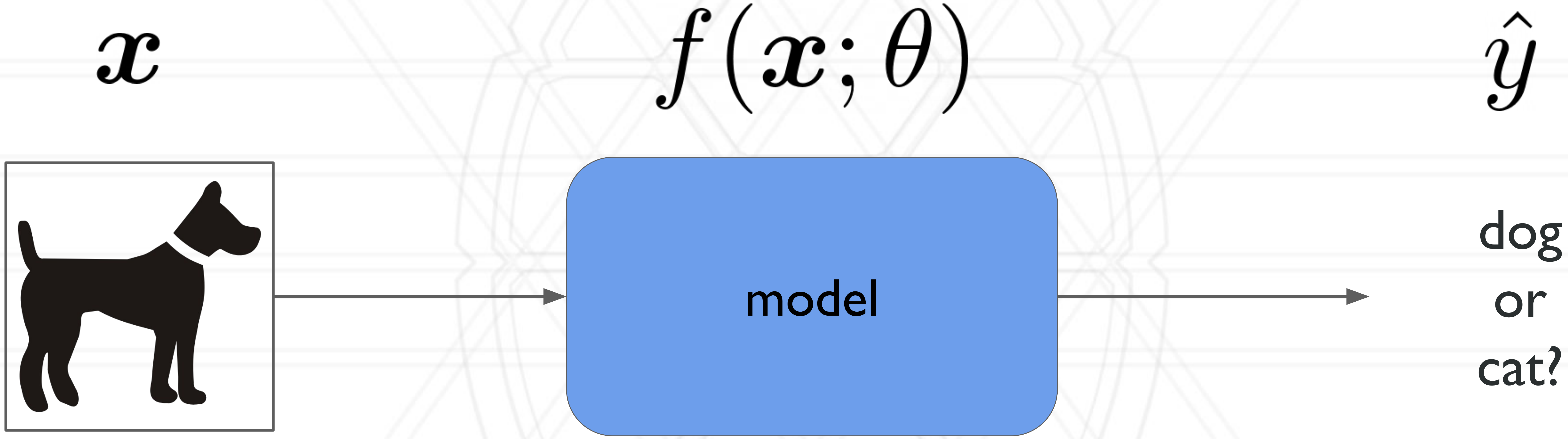
Announcements

- Assignment 1 is out
  - Due Feb. 25th at midnight

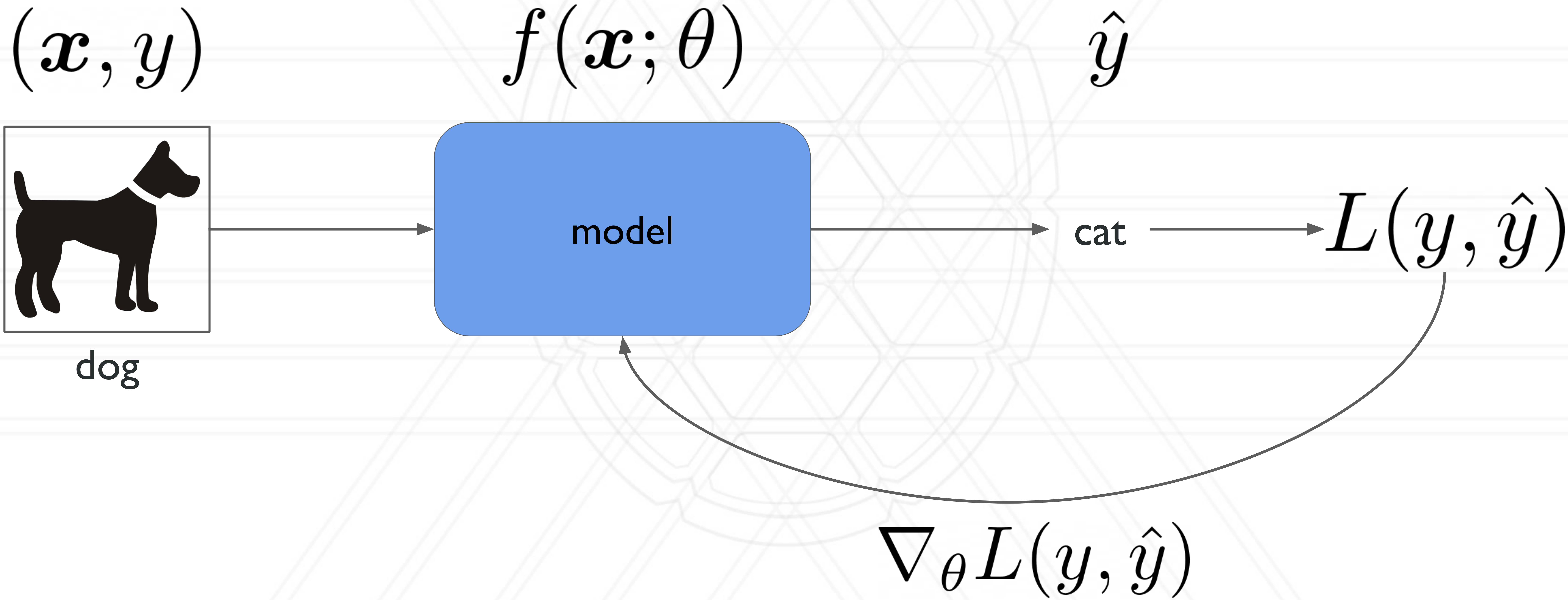
Triton

- Calling functions from Triton kernels
- Shape specific hyperparameters
- Triton functions
- https://triton-lang.org/main/index.html

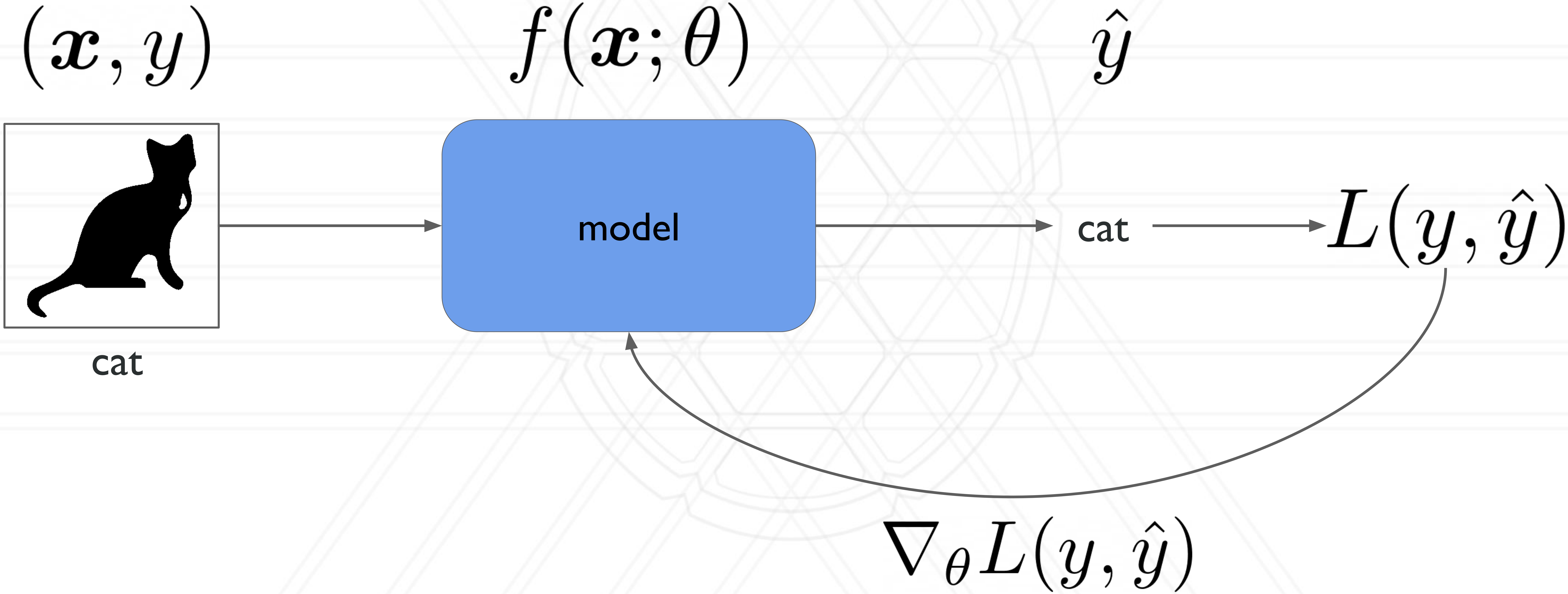
DL Models High Level Overview



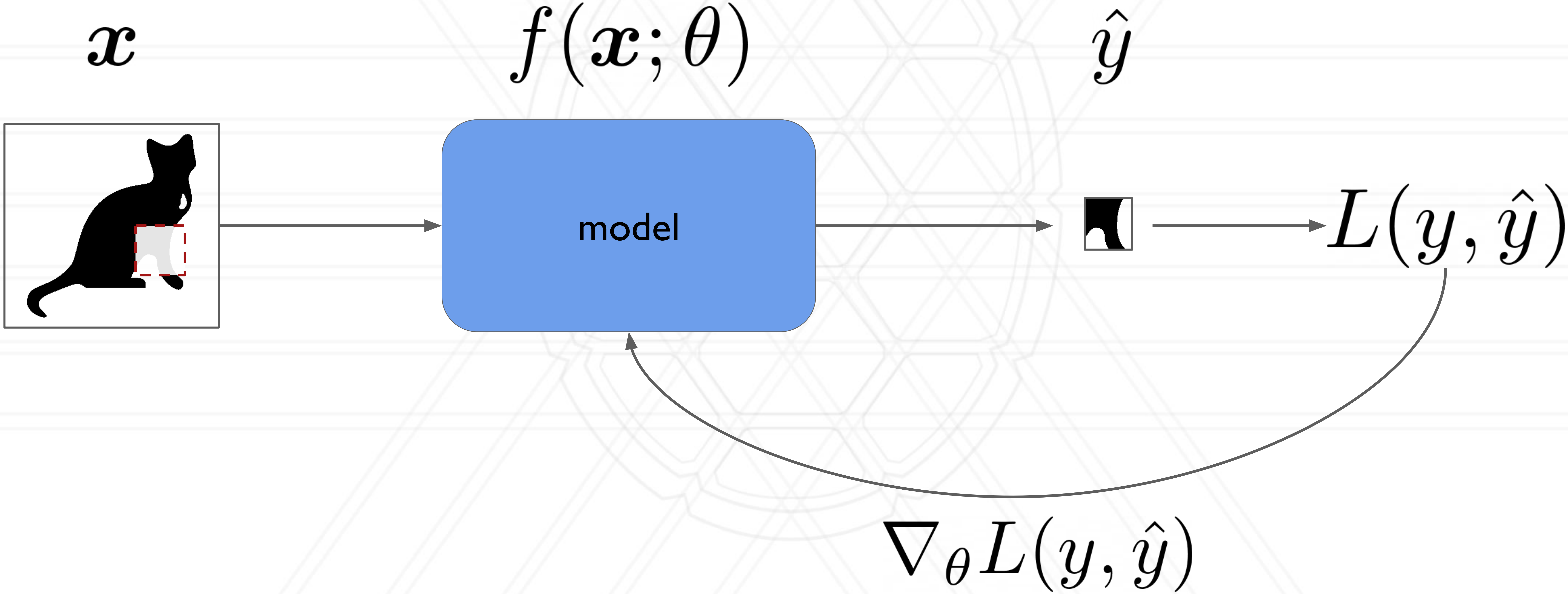
Supervised Training Overview



Supervised Training Overview



Self-Supervised Training Overview



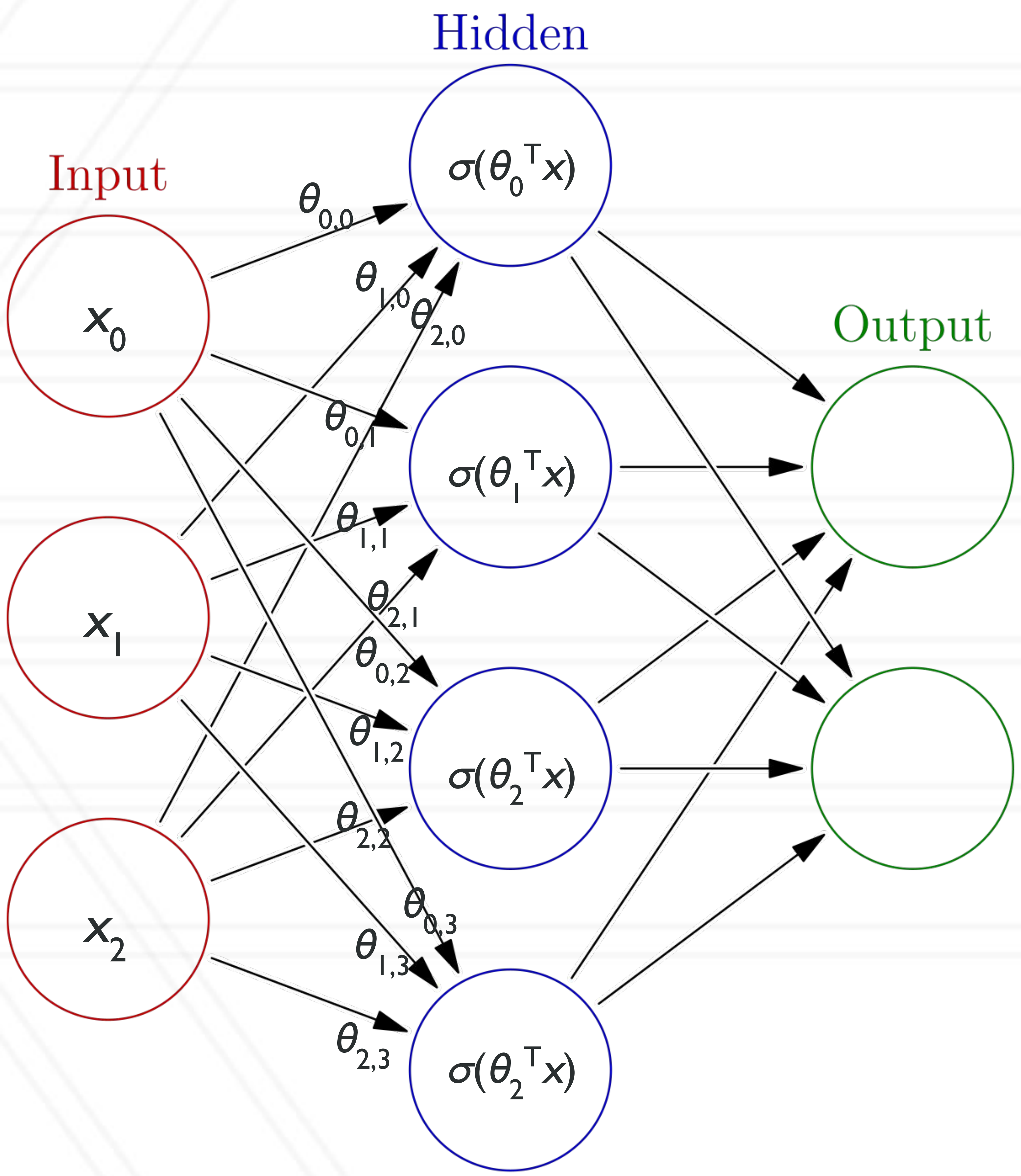
Dense Neural Networks

- Linear models are not always enough

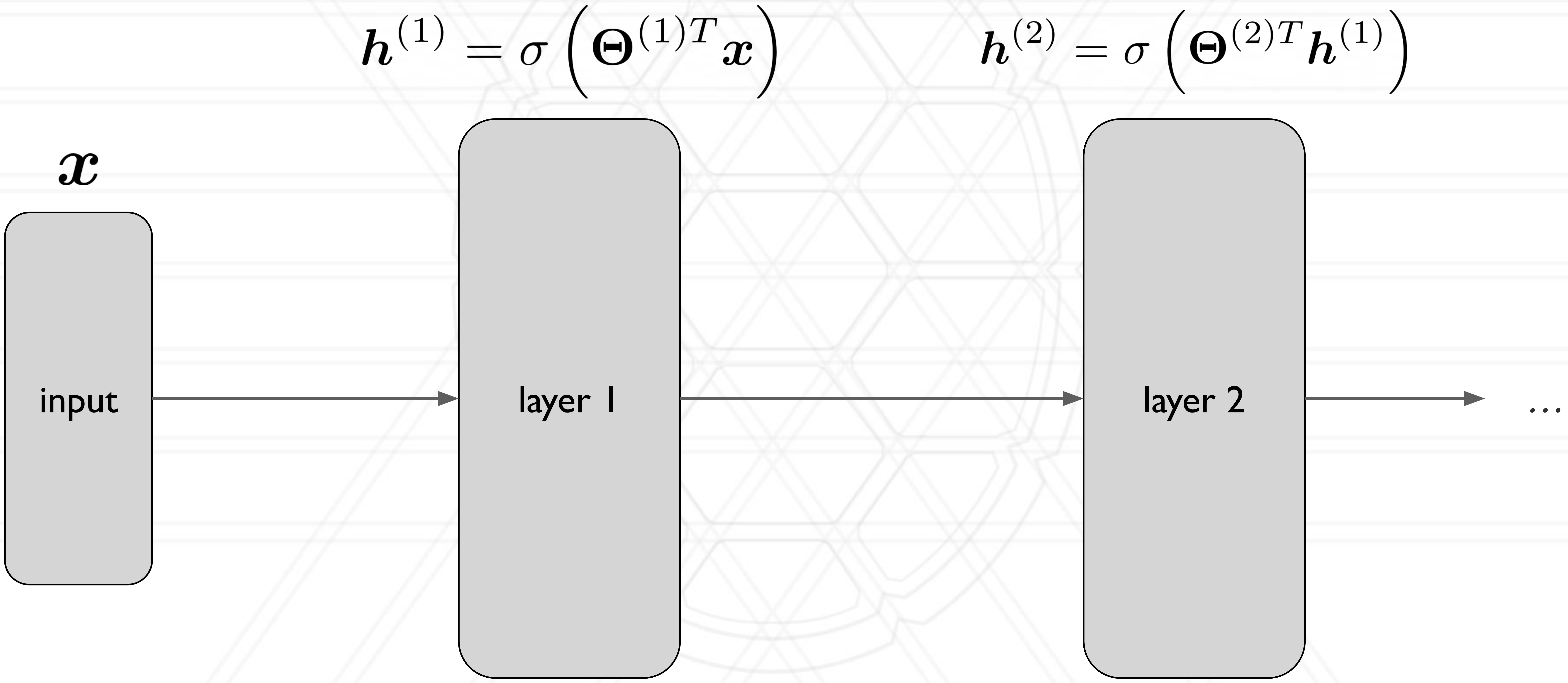
$$f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{\theta}^T \boldsymbol{x}$$

- Most real world problems are hierarchical and non-linear
- Neural networks add levels of non-linearity
- Each unit is an activation of a linear transformation

$$\boldsymbol{h} = \sigma\left(\boldsymbol{\Theta}^T \boldsymbol{x}\right)$$

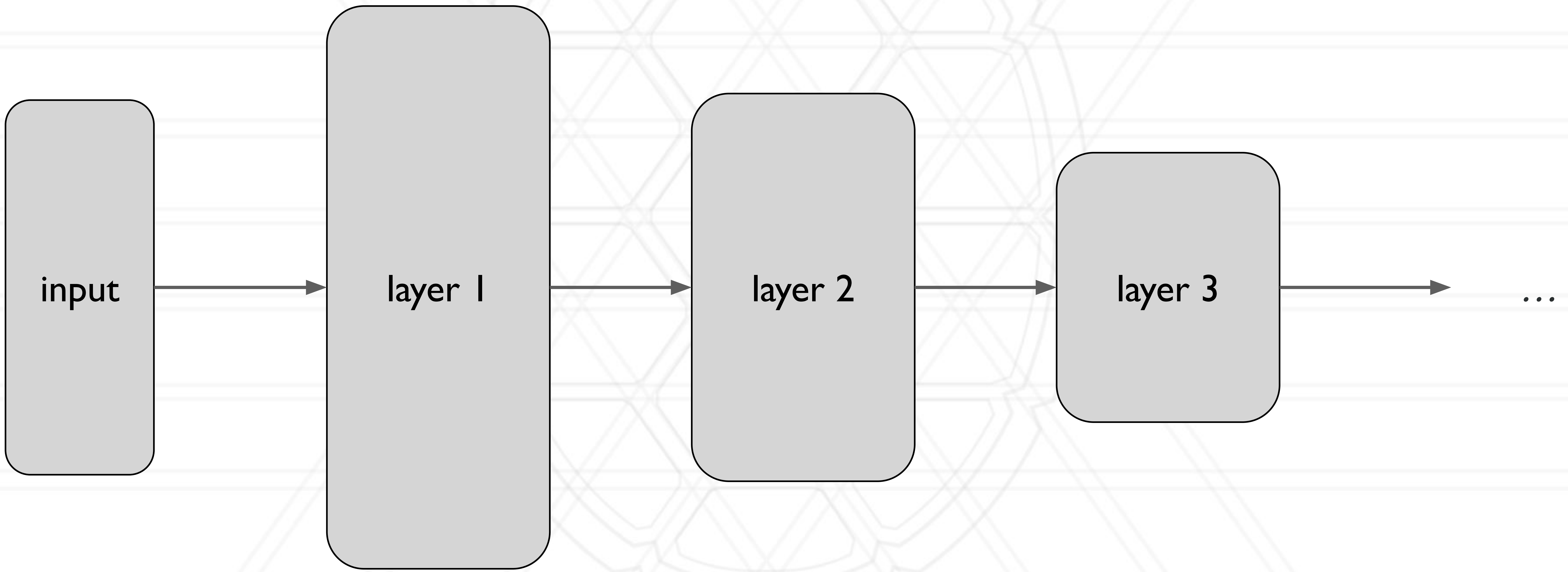


Layers



$\boldsymbol{h}^{(1)} = \sigma\left(\boldsymbol{\Theta}^{(1)T}\boldsymbol{x}\right)$
$\boldsymbol{h}^{(2)} = \sigma\left(\boldsymbol{\Theta}^{(2)T}\boldsymbol{h}^{(1)}\right)$
$\boldsymbol{x}$
input
layer 1
layer 2
...

Layers



Gradient Descent

- Optimization algorithm for convex functions
- Also works well for neural networks
- Iteratively step in opposite direction of gradient
- Used to minimize prediction loss or error
- To minimize *f(x)*:

$$x_{n+1} = x_n - \eta \nabla_x f(x_n)$$

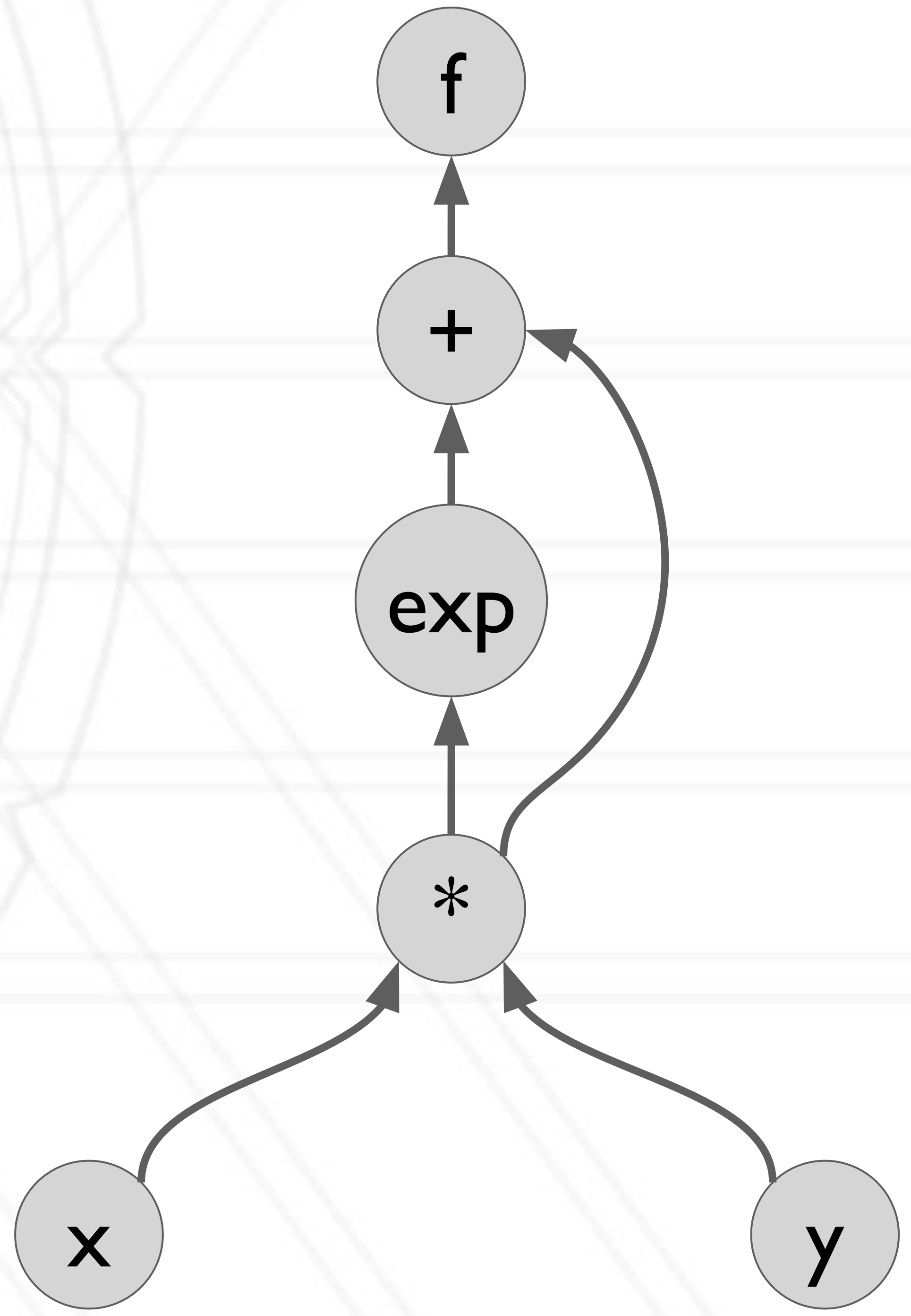
https://www.cs.umd.edu/class/spring2025/cmsc828g/gradient-descent.shtml

Backpropagation

- Algorithm used to compute gradients
- Uses chain rule and dynamic programming to remove redundant computations

$$f(x, y) = xy + \exp(xy)$$

$$\frac{\partial f}{\partial x} = ? \qquad \frac{\partial f}{\partial y} = ?$$



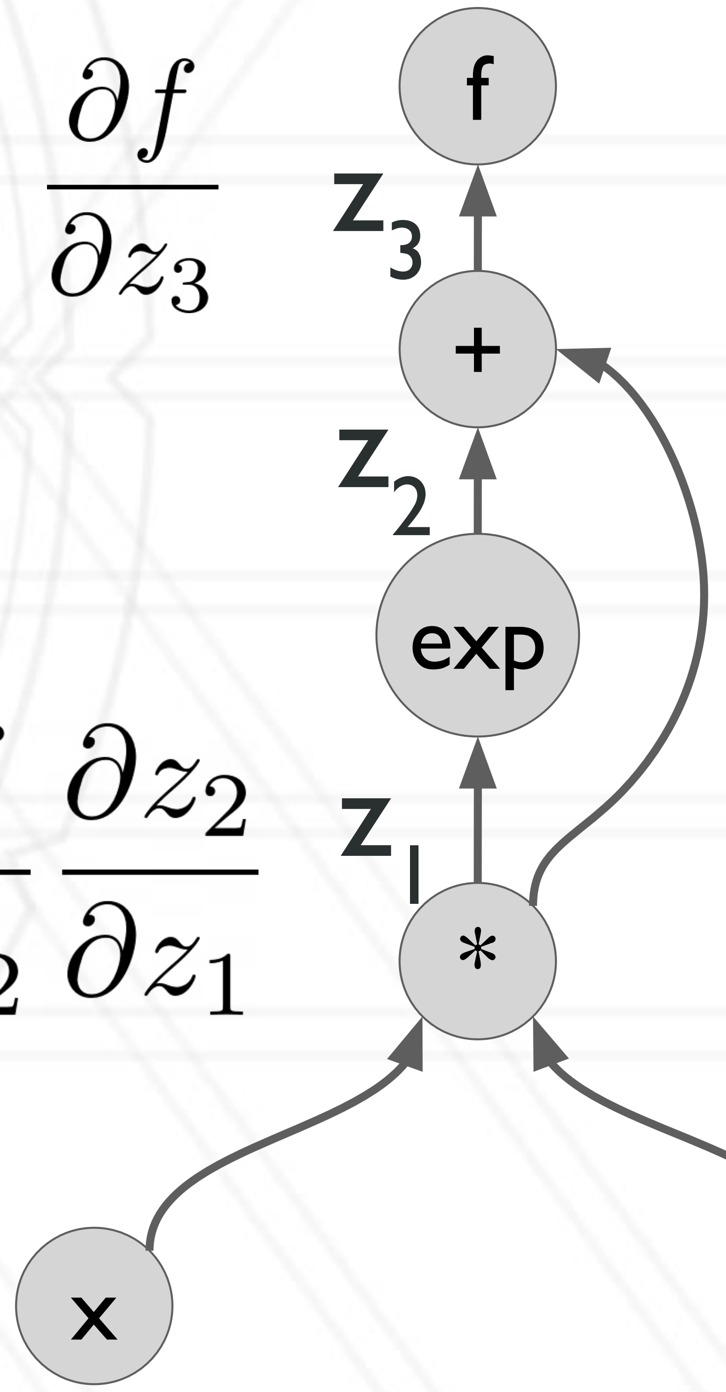
Backpropagation

- Algorithm used to compute gradients
- Uses chain rule and dynamic programming to remove redundant computations

$$f(x, y) = xy + \exp(xy)$$

$$\frac{\partial f}{\partial x} = ? \qquad \frac{\partial f}{\partial y} = ?$$

$$\frac{\partial f}{\partial z_3}$$



$$\frac{\partial f}{\partial z_2} = \frac{\partial f}{\partial z_3}\frac{\partial z_3}{\partial z_2}$$

$$\frac{\partial f}{\partial z_1} = \frac{\partial f}{\partial z_3}\frac{\partial z_3}{\partial z_1} + \frac{\partial f}{\partial z_2}\frac{\partial z_2}{\partial z_1}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x}$$

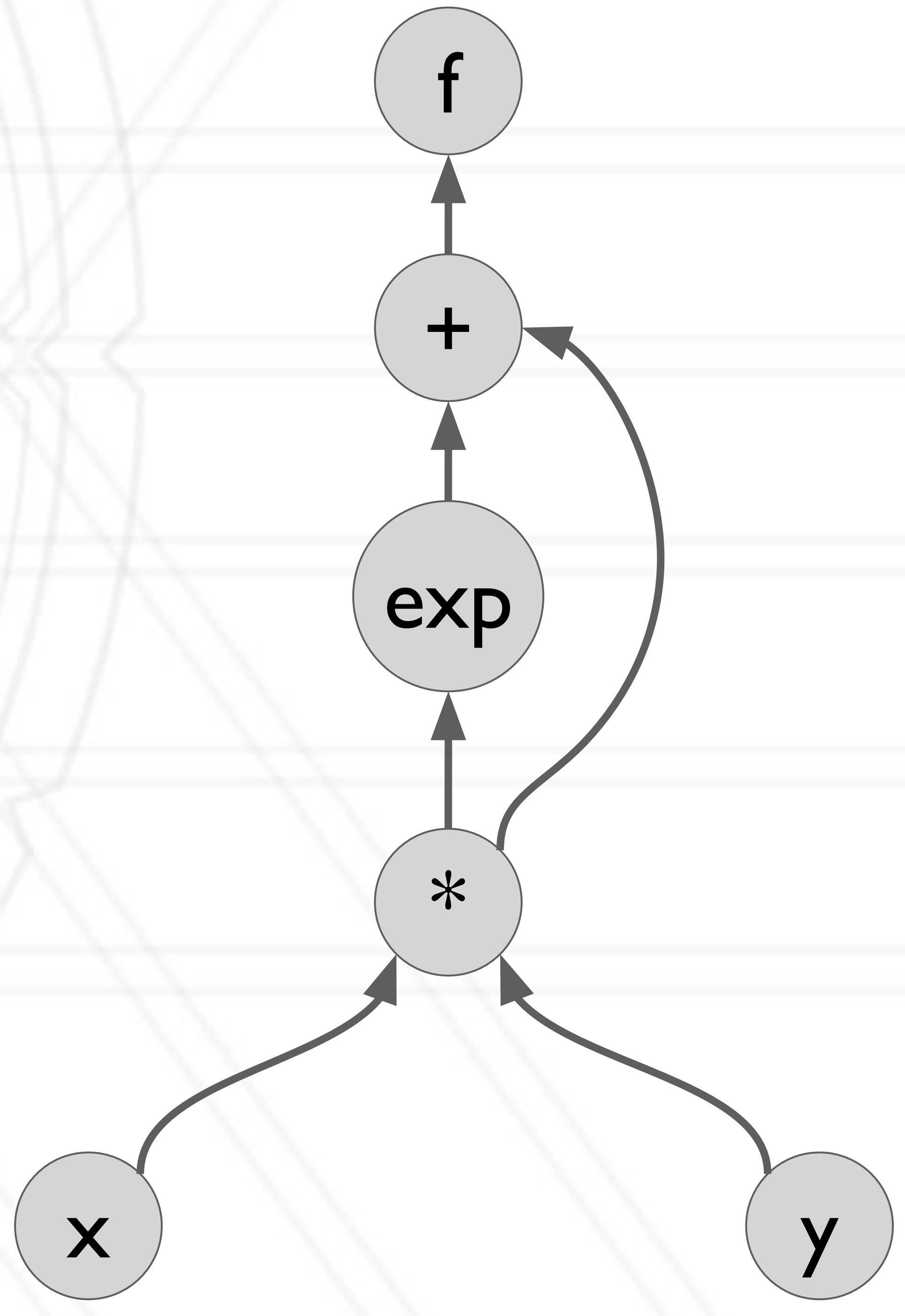
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial y}$$

Backpropagation

- Algorithm used to compute gradients
- Uses chain rule and dynamic programming to remove redundant computations
- Algorithm:

compute grad of V

1. if cached grad(V), return grad(V)
2. loop through consumers c of V
    - 2a. d = recursively compute grad of c
    - 2b. G_c = use backprop to compute grad of V wrt c
1. return sum of G_c



Training Loop

```
running_loss = 0.
last_loss = 0.

for i, data in enumerate(training_loader):
    inputs, labels = data

    optimizer.zero_grad()

    outputs = model(inputs)

    loss = loss_fn(outputs, labels)
    loss.backward()

    optimizer.step()

    running_loss += loss.item()
    if i % print_every == print_every-1:
        last_loss = running_loss / print_every
        print('  batch {} loss: {}'.format(i + 1, last_loss))
        running_loss = 0.
```

Training Loop

```
running_loss = 0.
last_loss = 0.

for i, data in enumerate(training_loader):
   inputs, labels = data

   optimizer.zero_grad()

   outputs = model(inputs)

   loss = loss_fn(outputs, labels)
   loss.backward()

   optimizer.step()

   running_loss += loss.item()
   if i % print_every == print_every-1:
       last_loss = running_loss / print_every
       print('  batch {} loss: {}'.format(i + 1, last_loss))
       running_loss = 0.
```

Each *epoch* loop through the entire dataset

Prepare for gradient computation

Forward pass

Loss Computation

Compute gradients

Update weights

Training Loop: Bottlenecks

```
running_loss = 0.
last_loss = 0.

for i, data in enumerate(training_loader):
   inputs, labels = data

   optimizer.zero_grad()

   outputs = model(inputs)

   loss = loss_fn(outputs, labels)
   loss.backward()

   optimizer.step()

   running_loss += loss.item()
   if i % print_every == print_every-1:
       last_loss = running_loss / print_every
       print('  batch {} loss: {}'.format(i + 1, last_loss))
       running_loss = 0.
```

Getting data from disk to GPU

Forward pass

Backward pass

Batching and Stochastic Gradient Descent

- Computing entire gradient is infeasible
  - Estimate with sample mean using samples
- Use matrices for fully connected layers
- Batching allows us to trade-off accuracy and efficiency
  - Larger batches provide more accurate gradient estimates
  - Diminishing returns for larger batches with increasing compute requirements

$$\boldsymbol{h}^{(l)} = \sigma\left(\boldsymbol{X}\boldsymbol{\Theta}\right)$$

Momentum and Adam

- SGD is inefficient
  - We can vary our step size using momentum

$$\boldsymbol{v}_{n+1} = \alpha \boldsymbol{v}_n - \eta \nabla f(\boldsymbol{x}_n)$$
$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \boldsymbol{v}_n$$

- Adam
  - Use 1st and 2nd moments to further decide step size

https://www.cs.umd.edu/class/spring2025/cmsc828g/gradient-descent.shtml

Optimizations: Activation Checkpointing

- Recompute values from forward pass to save memory

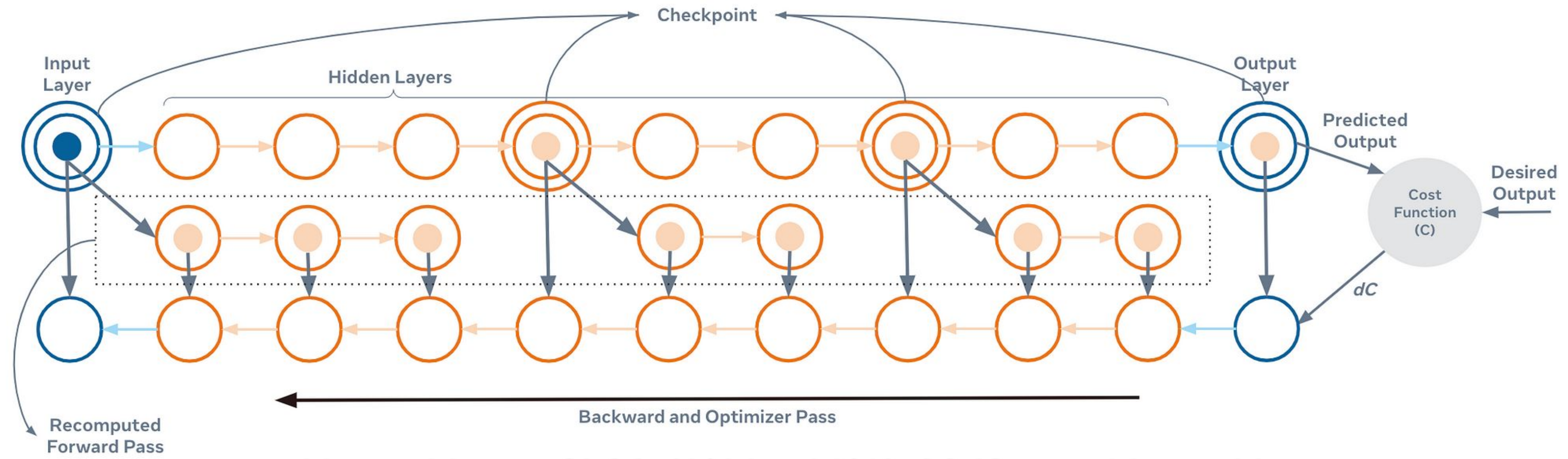
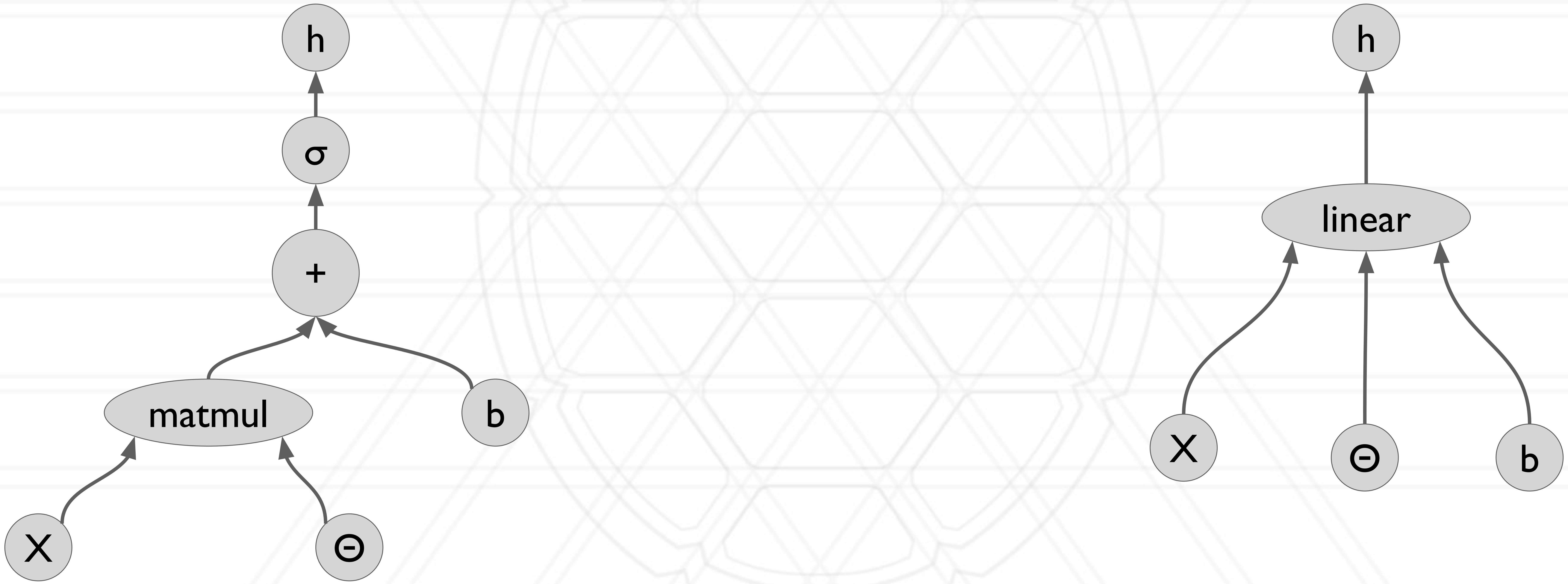


image: https://shivambharuka.medium.com/deep-learning-a-primer-on-distributed-training-part-1-d0ae0054bb1c

Optimizations: Fusion

- Fuse subgraphs in the compute graph into faster operations



PyTorch

- A machine learning Python framework
- Sophisticated autograd capabilities
- Supports many accelerator backends
- ML specific optimizations
  - compiler
  - kernels

Tensors

- N-D arrays
- Usually created with *torch.empty*, *torch.ones*, *torch.zeros*, *torch.rand*
- Support most math operations

```
ones = torch.zeros(2, 2) + 1
twos = torch.ones(2, 2) * 2
threes = (torch.ones(2, 2) * 7 - 1) / 2
fours = twos ** 2
sqrt2s = twos ** 0.5
```

https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html

Tensors

- N-D arrays
- Usually created with *torch.empty*, *torch.ones*, *torch.zeros*, *torch.rand*
- Support most math operations
- Support broadcasting

```
rand = torch.rand(2, 4)
doubled = rand * (torch.ones(1, 4) * 2)
```

https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html

Tensors

- N-D arrays
- Usually created with *torch.empty*, *torch.ones*, *torch.zeros*, *torch.rand*
- Support most math operations
- Support broadcasting
- Can be stored on CPU or GPU

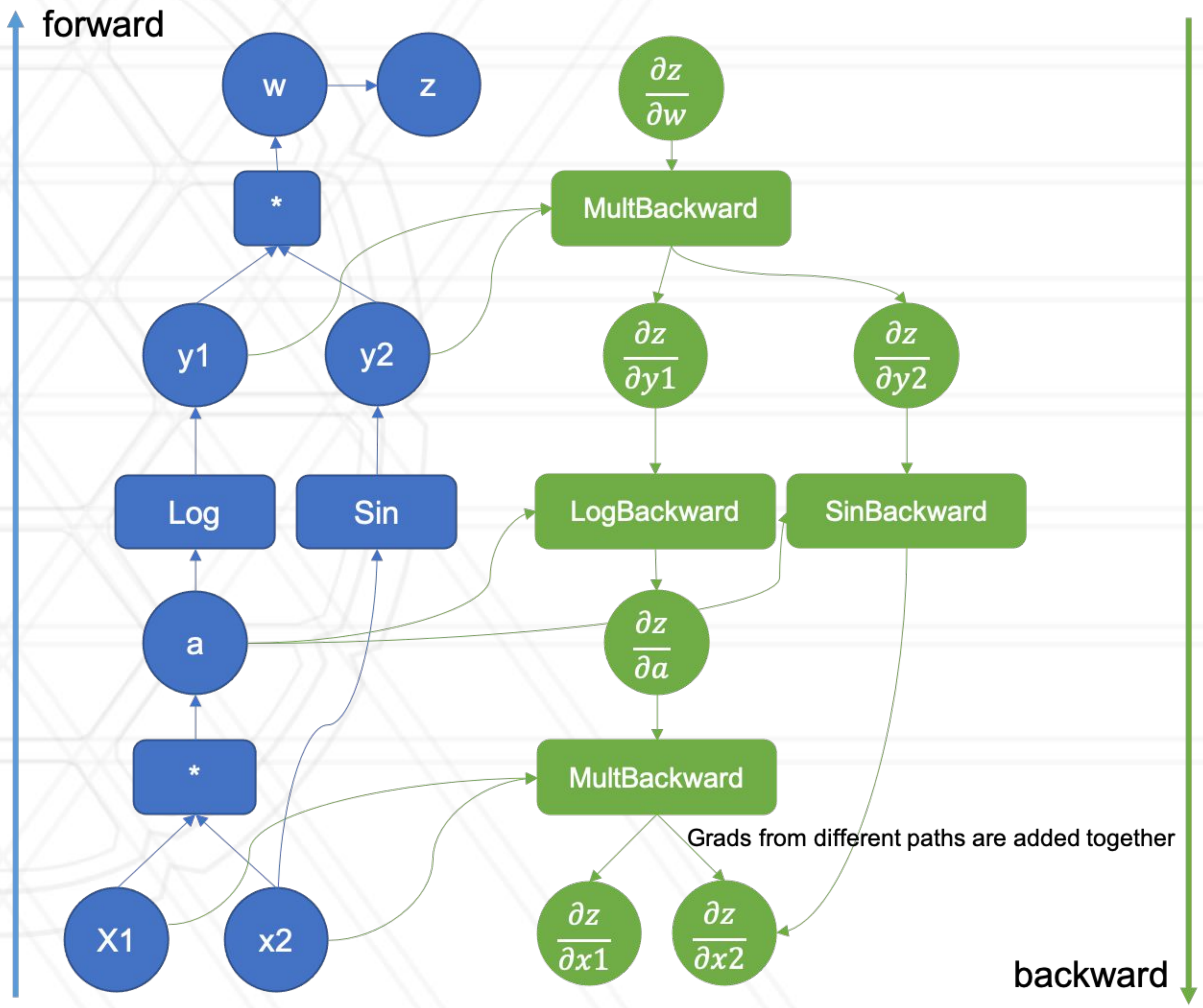
```
y = torch.rand(2, 2)
y = y.to(my_device)

y = torch.rand(2, 2, device='cuda')
```

https://pytorch.org/tutorials/beginner/introyt/tensors_deeper_tutorial.html

Operations and Compute Graph

- The graph is automatically managed in PyTorch
- Most typical numpy and math operations are supported
- https://pytorch.org/docs/stable/torch.html



Computing Gradients

- Tensors must have *.requires_grad = True*
- *.backward()* computes gradients

```
x = torch.ones(5)
y = torch.zeros(3)
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)

z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)

loss.backward()
print(w.grad)
print(b.grad)
```

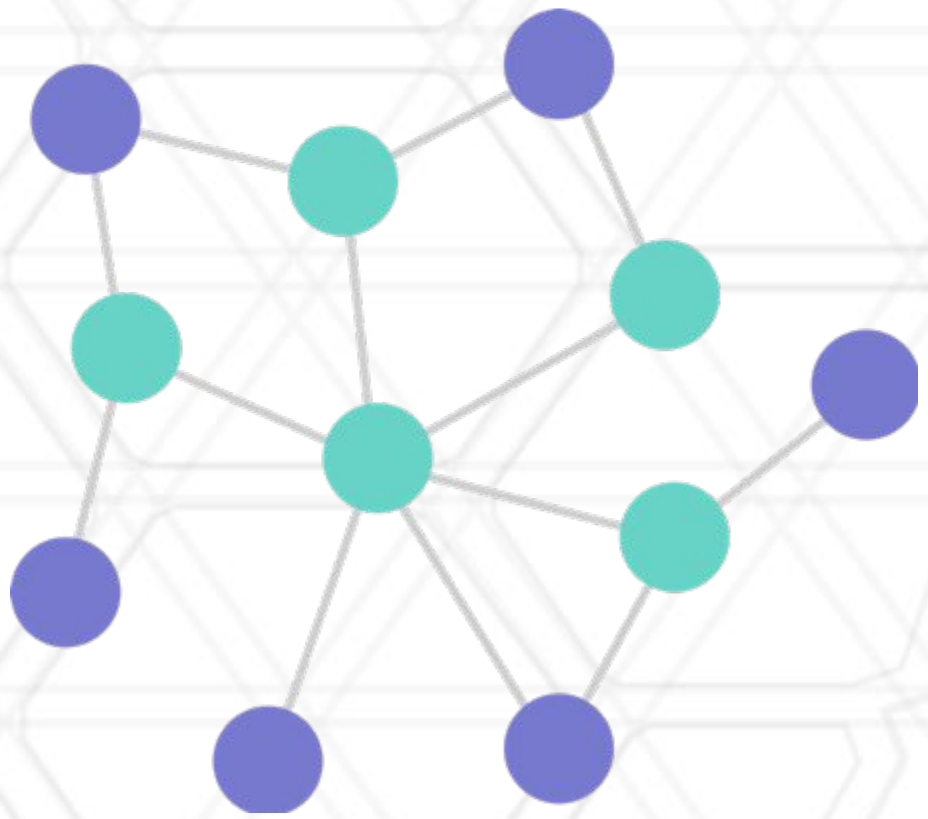
Tell torch we need gradients for these tensors

Compute the gradients

Graph Neural Networks

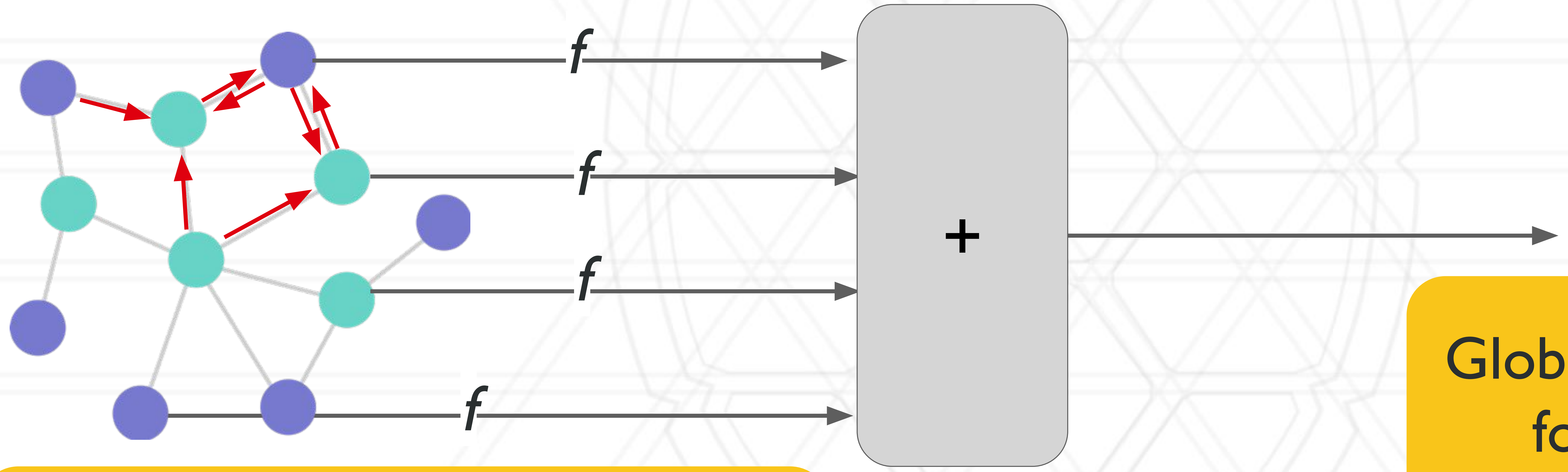
3 pieces of data: node values, edge values, adjacency information

Several learning tasks: node-level, edge-level, graph-level



Graph Neural Networks

We can use a neural network to simply model node features



Global pooling can be used for graph level tasks

Message passing is used to learn from graph relational structure

UNIVERSITY OF
MARYLAND