CS 471/571(Fall 2023): Introduction to Artificial Intelligence

Lecture 25: Neural Nets

Thanh H. Nguyen

Source: http://ai.berkeley.edu/home.html

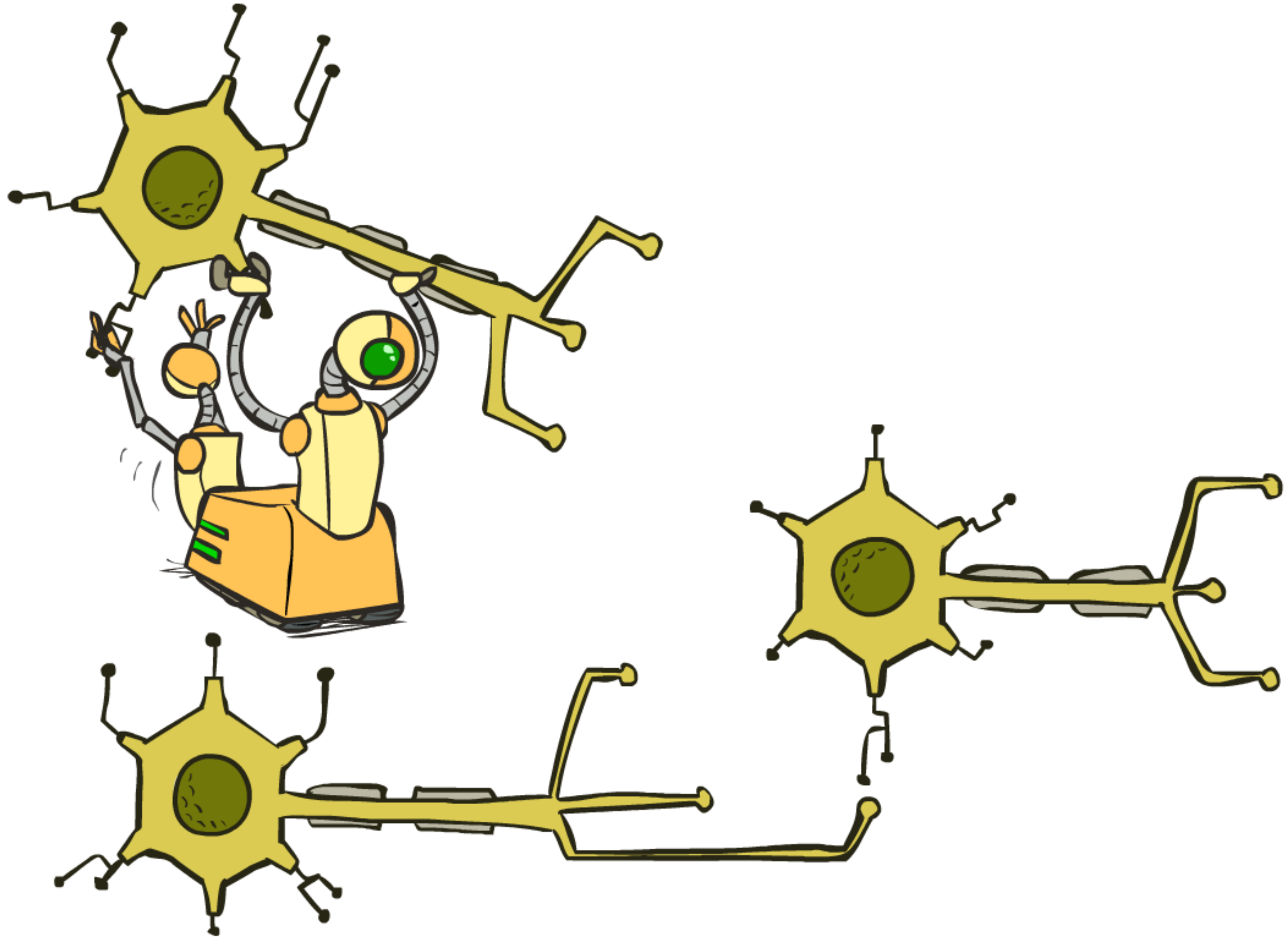
Announcement & Reminder

- Exam Review
  - Friday, Dec 01st, 2023

- Written assignment 4
  - Deadline: Wednesday, November 29th, 2023.

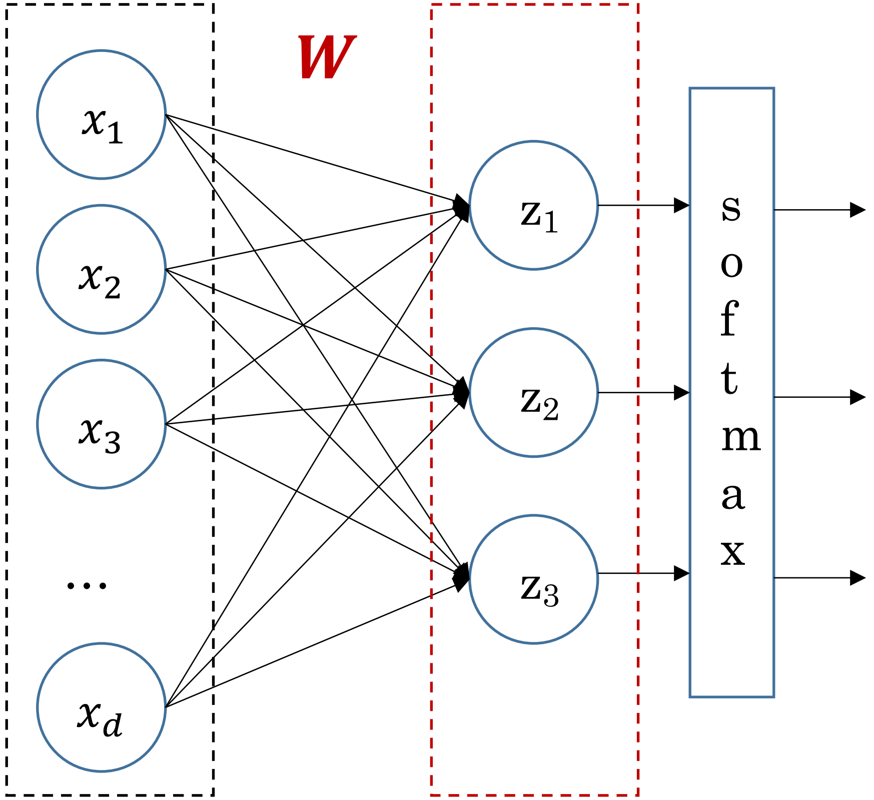
- Student experience survey
  - Deadline: 06:00 AM on Monday, Dec 4th, 2023
  - If >= 80% of students complete the survey, everyone will get an extra <u>2% credit</u> for your final grade

Neural Networks



Multi-class Logistic Regression

- = special case of neural network



$$P(y_1) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

$$P(y_2) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

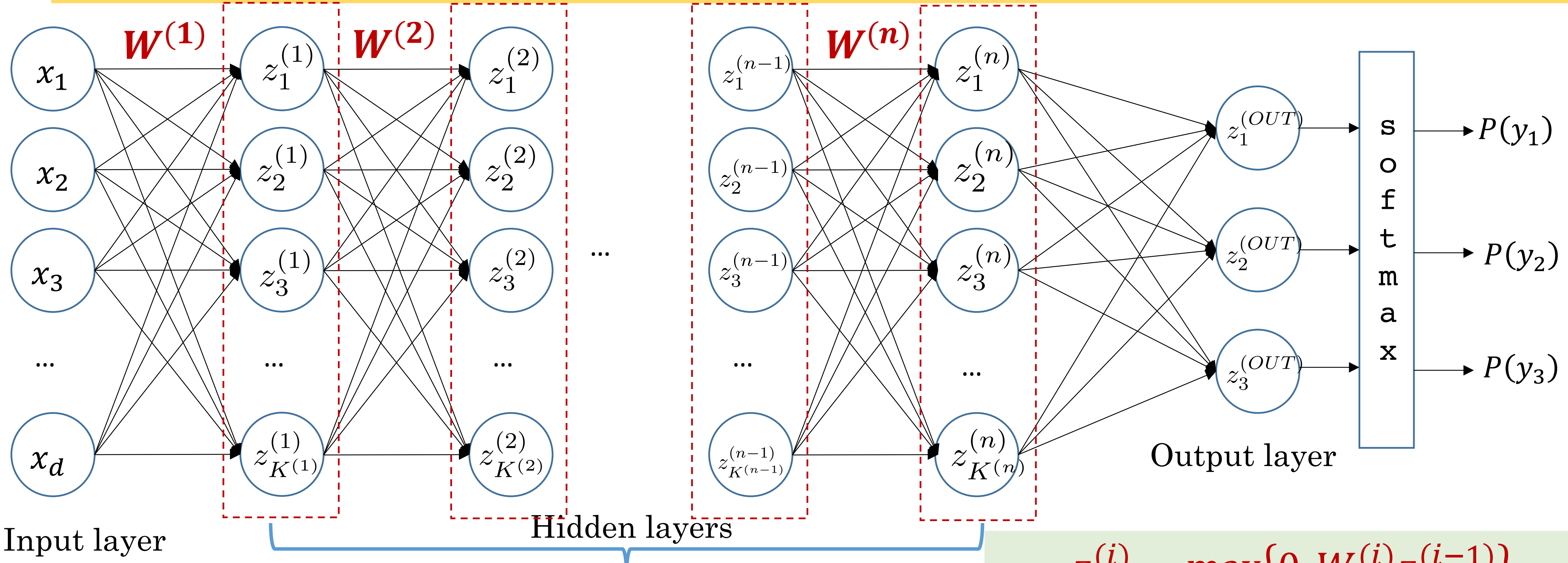
$$P(y_3) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

Input layer: $x = [x_1, x_2, \cdots, x_d]^T$

Intermediate (hidden) layer: $z = [z_1, z_2, z_3] = Wx$

Weight matrix: $W = \{w_{ij}\}$
Capture how much each input feature x_i influence each intermediate value z_j

Deep Neural Network = Learn the Features!



i^{th} hidden layer: $z^{(i)} = \left(z_1^{(i)}, z_2^{(i)}, \cdots, z_{K^{(i)}}^{(i)}\right)$

has hidden size $K^{(i)}$

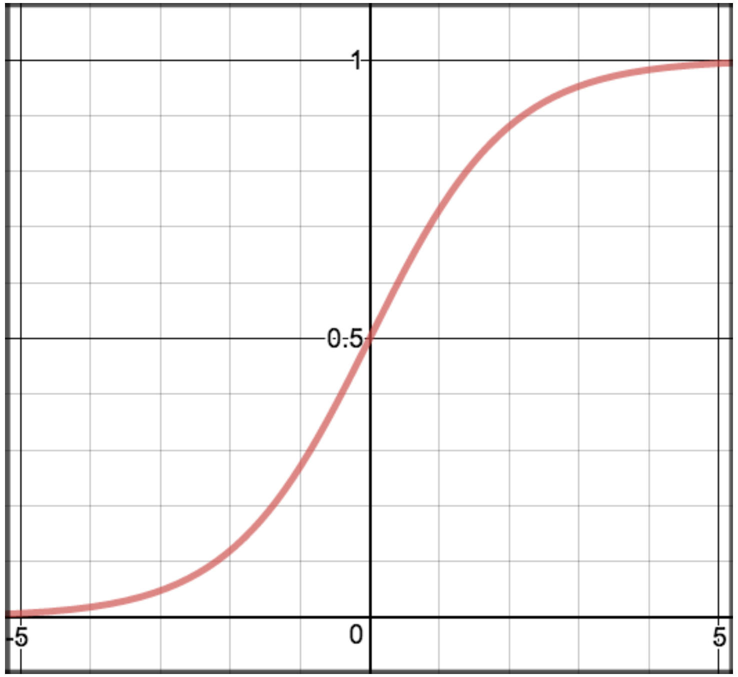
$$z^{(i)} = \max\{0, W^{(i)} z^{(i-1)}\}$$

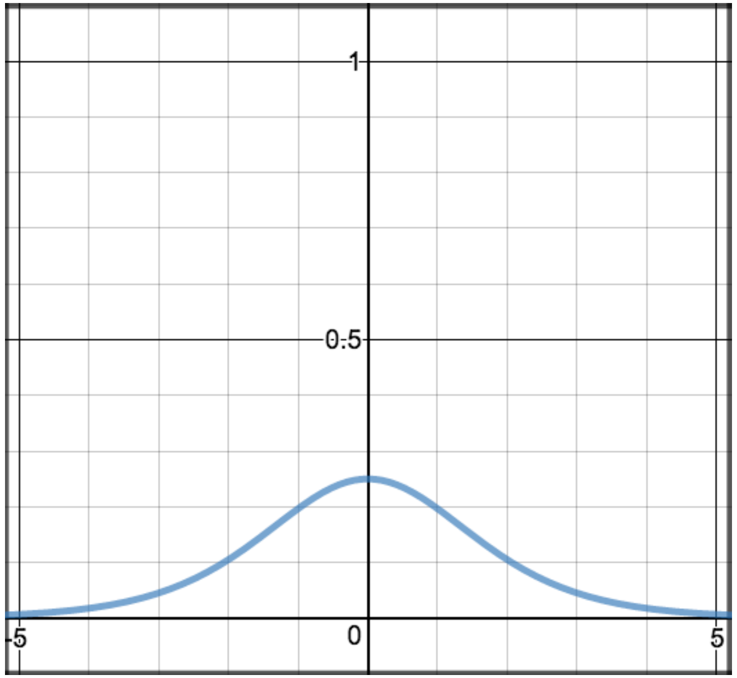
Element-wise max operator, called ReLu activation function

Common Activation Functions

Source: https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

Sigmoid



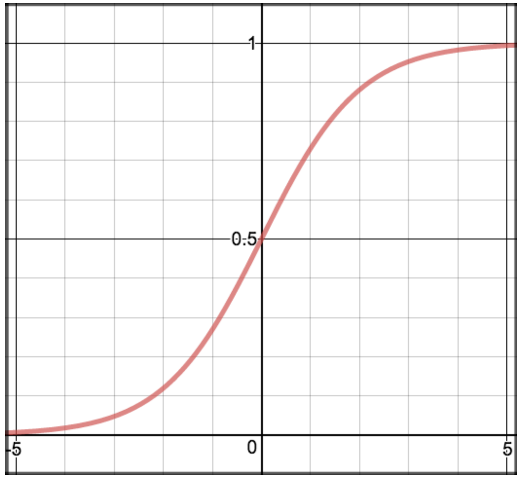


Function: $z = \dfrac{1}{1 + e^{-x}}$

Derivative: $\dfrac{dz}{dx} = z \cdot (1 - z)$

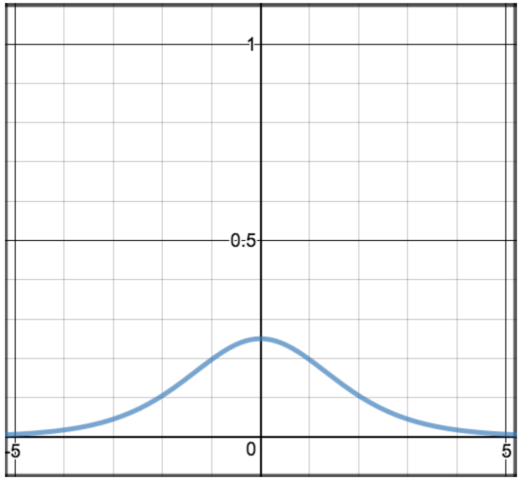
Sigmoid

- Pros
  - Is nonlinear.
  - Has a smooth gradient.
  - Good for a classifier.
  - The output is bounded within (0,1)
- Cons
  - Towards either end of the sigmoid function, the z values tend to respond very less to changes in x.
  - Gives rise to a problem of "vanishing gradients".



Function:

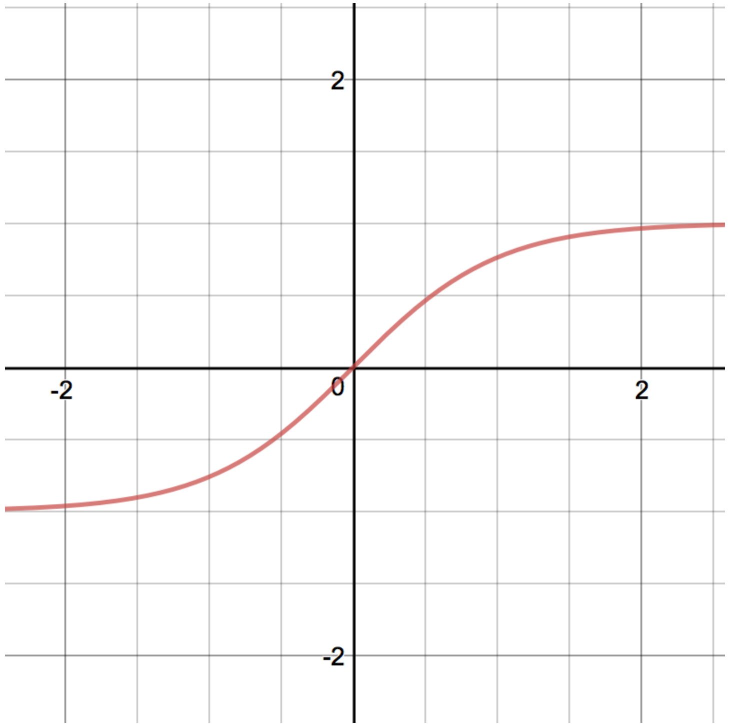
$$z = \frac{1}{1 + e^{-x}}$$

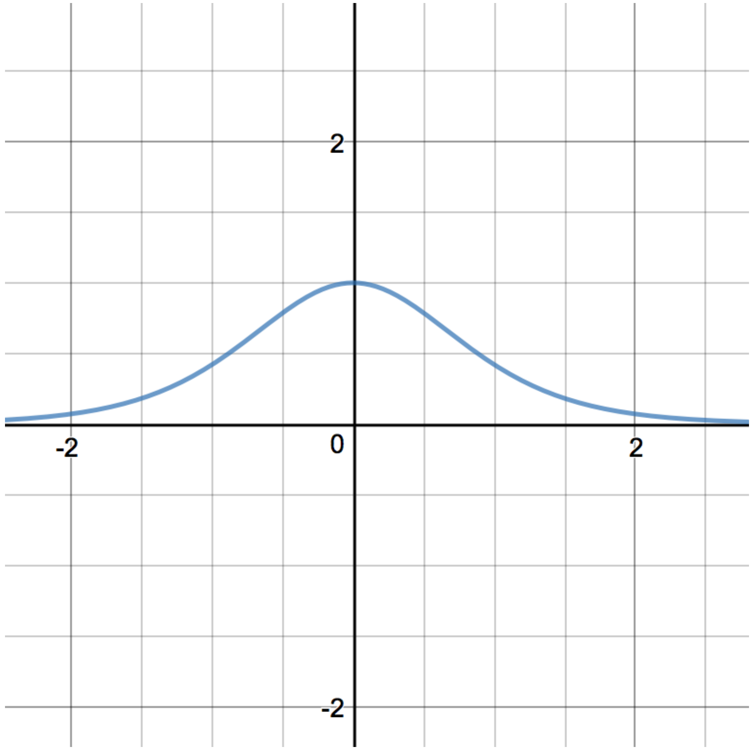


Derivative:

$$\frac{dz}{dx} = z \cdot (1 - z)$$

Tanh



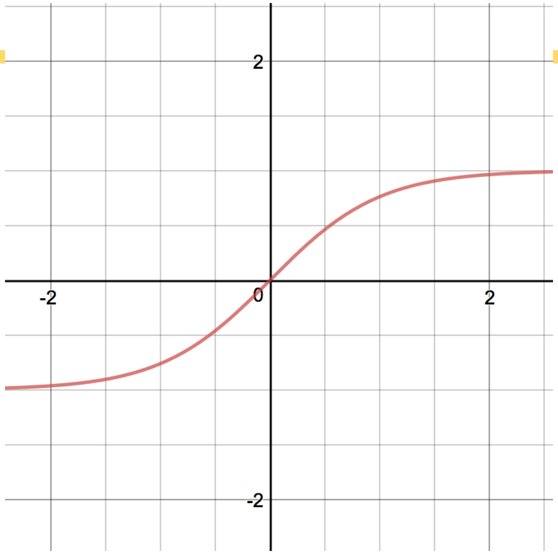


Function: $z = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

Derivative: $\dfrac{dz}{dx} = 1 - z^2$

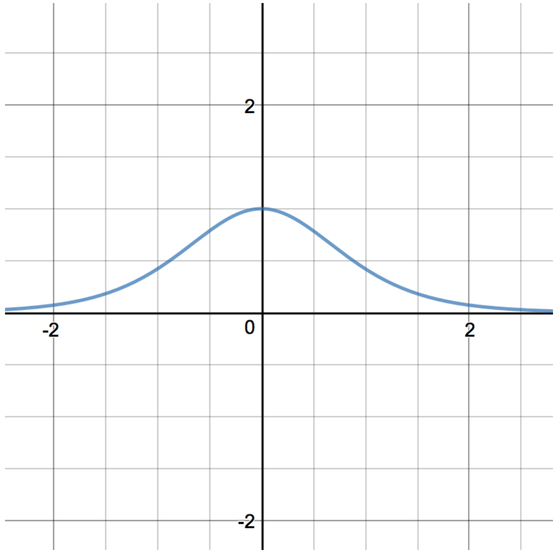
Tanh

- Pros
  - The gradient is stronger for tanh than sigmoid (derivatives are steeper).
- Cons
  - Tanh also has the vanishing gradient problem.



Function:

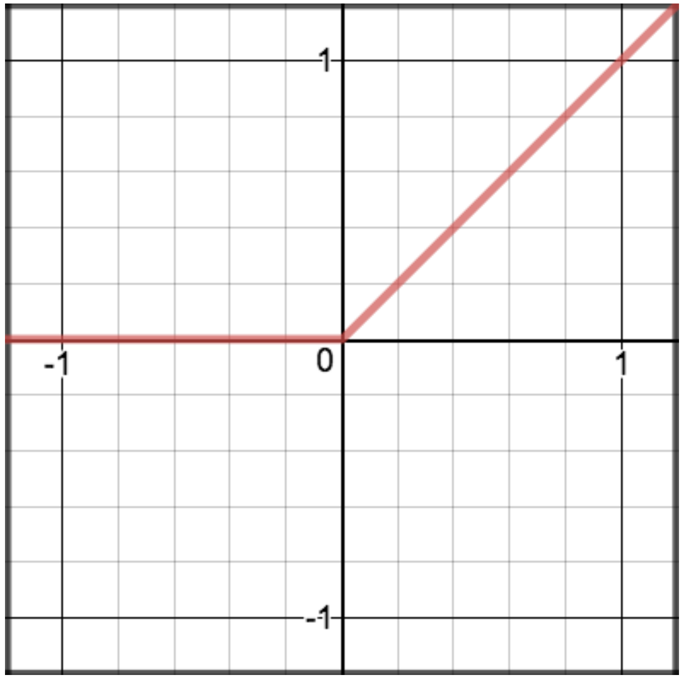
$$z = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



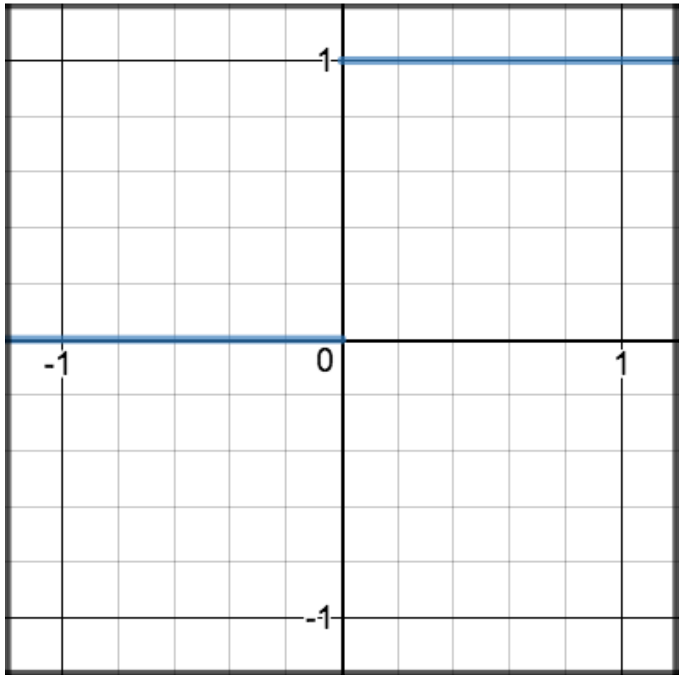
Derivative:

$$\frac{dz}{dx} = 1 - z^2$$

ReLU



Function: $z = \max\{0, x\}$



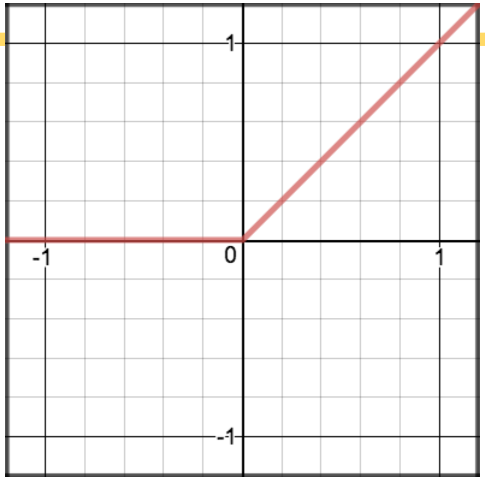
Derivative: $\dfrac{dz}{dx} = \begin{cases} 1, \text{if } x > 0 \\ 0, \text{if } x < 0 \end{cases}$

[source: MIT 6.S191 introtodeeplearning.com]

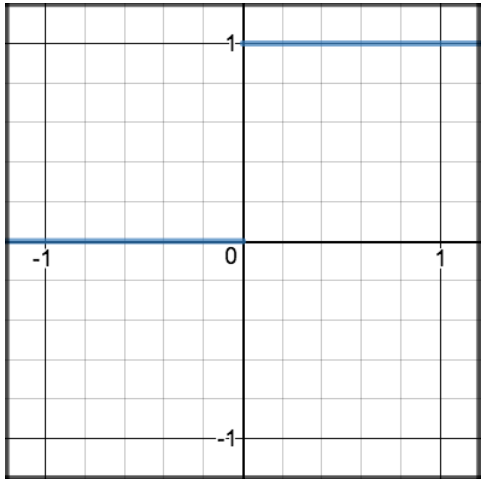
ReLU

- Pros
  - ReLu avoids and rectifies vanishing gradient problem.
  - ReLu is less computationally expensive

- Cons
  - ReLu should only be used within hidden layers.
  - Dying ReLu problem: When ($x < 0$), gradient of ReLu is 0, meaning the weights will not get adjusted during gradient descent.
  - The range of ReLu is [0,∞). This means it can blow up the activation.



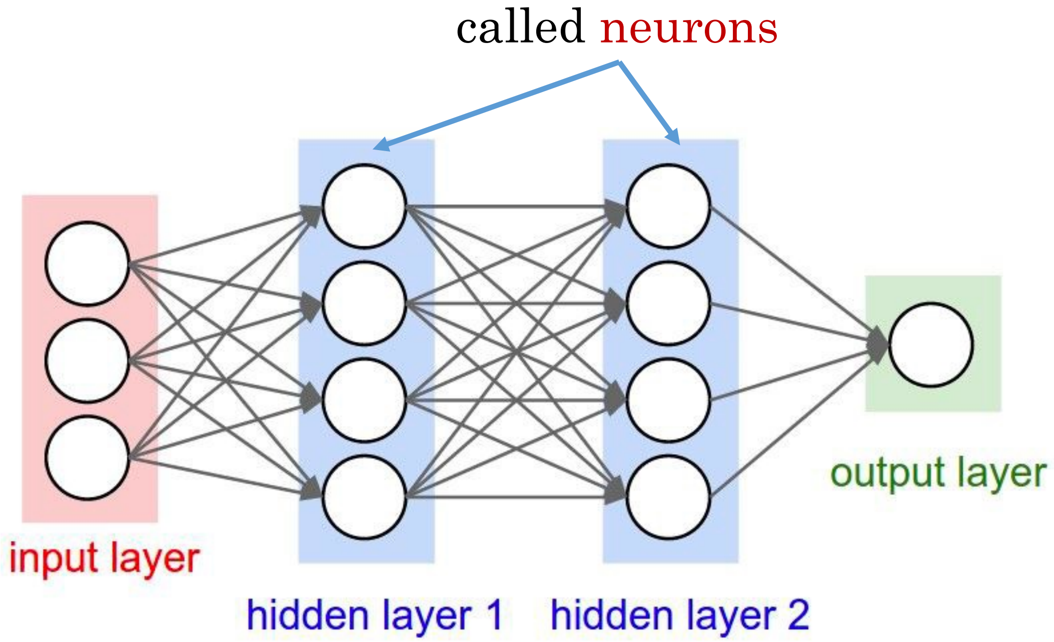
Function:
$z = \max\{0, x\}$



Derivative:
$$\frac{dz}{dx} = \begin{cases} 1, \text{if } x > 0 \\ 0, \text{if } x < 0 \end{cases}$$

Train a Neural Net

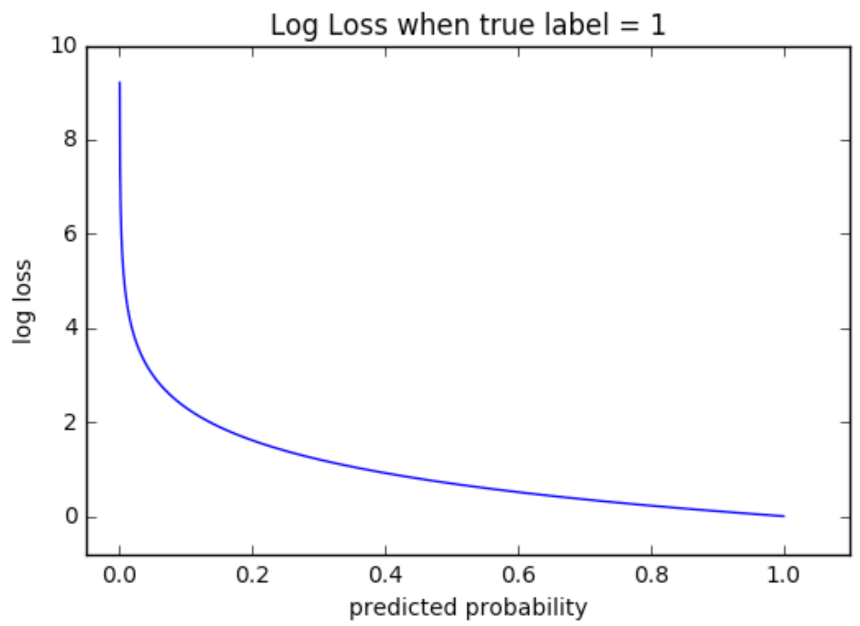
- Train a neural networks: gradient descent
  - Forward pass: input is passed forward through the network to produce a prediction output. A loss metric is computed based on the difference between prediction and target (true output).
  - Backward pass: derivatives of this loss metric are calculated and propagated back through the network using a technique called backpropagation.
- We will talk more about this!!!



Common Loss Functions

Cross Entropy

- Measures the performance of a classification model whose output is a probability value between 0 and 1
- Cross-entropy loss increases as the predicted probability diverges from the actual label.
- A perfect model would have a log loss of 0.



- Binary classification: $Loss = -(y \log p + (1 - y) \log(1 - p))$
- Multi-classification: $Loss = -\sum_l y_l \log p_l$
  - $y_l = 1$ if the true label is l, and $y_l = 0$, otherwise
  - p_l: predicted probability of having label l

Mean Square Error (MSE)

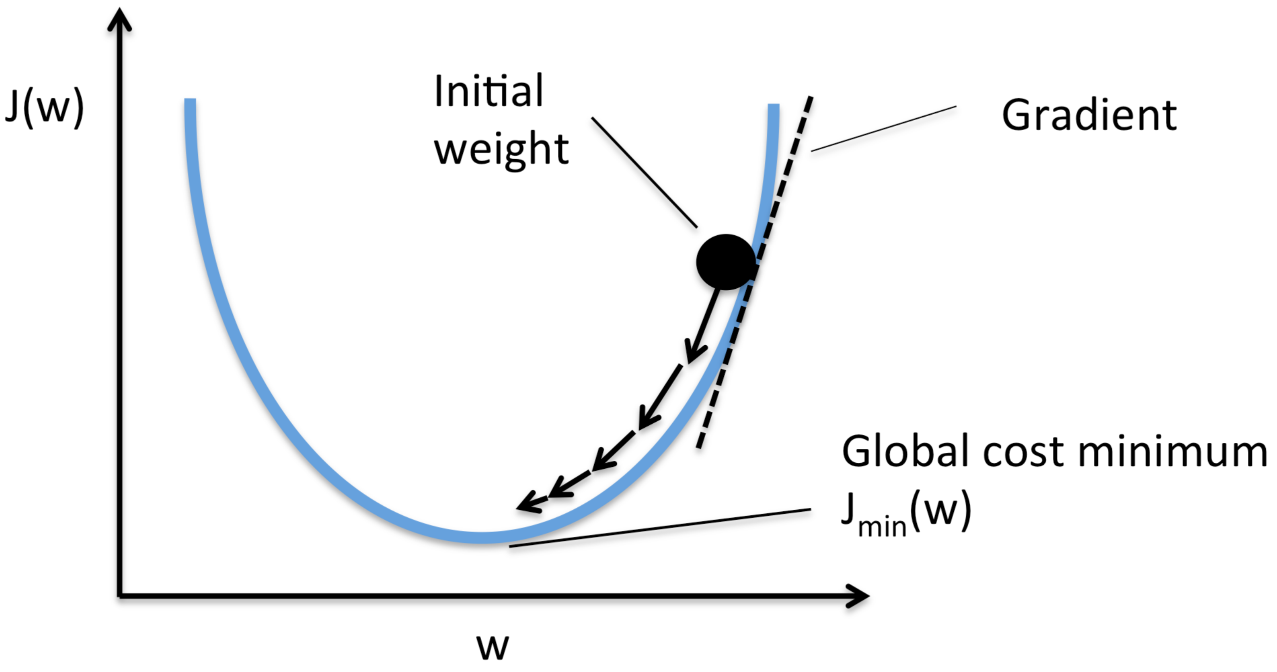
- Commonly used for regression loss
- Measure the average of squared difference between predictions and actual observations

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Training a Neural Network

Minimizing Error: Gradient Descent

- Main technique to minimize prediction error in neural network



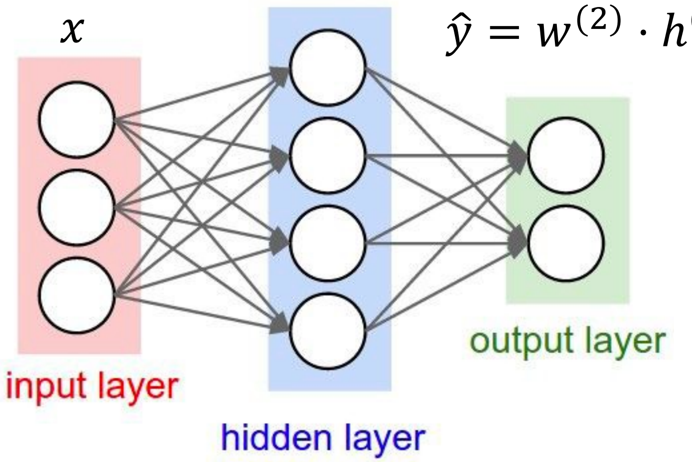
- Challenge: computing gradient is non-trivial!!!

Graph source: http://rasbt.github.io/mlxtend/user_guide/general_concepts/gradient-optimization/

(Bad Choice) Direct Computation

- Input:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{bmatrix}$$



Hing loss: $\max(0, 1 - \hat{y} \cdot y)$

- One hidden layer $h^{(1)} = \max(0, w^{(1)}x)$ with $K^{(1)}$ neurons
- Output layer $\hat{y} = w^{(2)} \cdot h^{(1)}$

$$w^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & \cdots & w_{1,K^{(1)}}^{(1)} \\ \vdots & \ddots & \vdots \\ w_{d,1}^{(1)} & \cdots & w_{d,K^{(1)}}^{(1)} \end{bmatrix}^T \qquad w^{(2)} = \begin{bmatrix} w_1^{(2)} \\ w_2^{(2)} \\ \dots \\ w_{K^{(1)}}^{(2)} \end{bmatrix}$$

(Bad Choice) Direct Computation

- Input data: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(n)}, y^{(n)})$
- Prediction (two-layer network): $\hat{y}^{(i)} = w^{(2)} \max(0, w^{(1)} x^{(i)})$
- Total loss = prediction loss + regularization

$$L = \sum_i \max(0, 1 - \hat{y}^{(i)} \cdot y^{(i)}) + \lambda R(w^{(1)}) + \lambda R(w^{(2)})$$
$$= \sum_i \max(0, 1 - \hat{y}^{(i)} w^{(2)} \max(0, w^{(1)} x^{(i)})) + \lambda R(w^{(1)}) + \lambda R(w^{(2)})$$

- Computing gradients: $\frac{dL}{dw^{(1)}}$ and $\frac{dL}{dw^{(2)}}$
  - So that we can learn $w^{(1)}$ and $w^{(2)}$

Direct computation of gradients is extremely inefficient!!!

Computing Gradients

Computational Graphs

+

Backpropagation

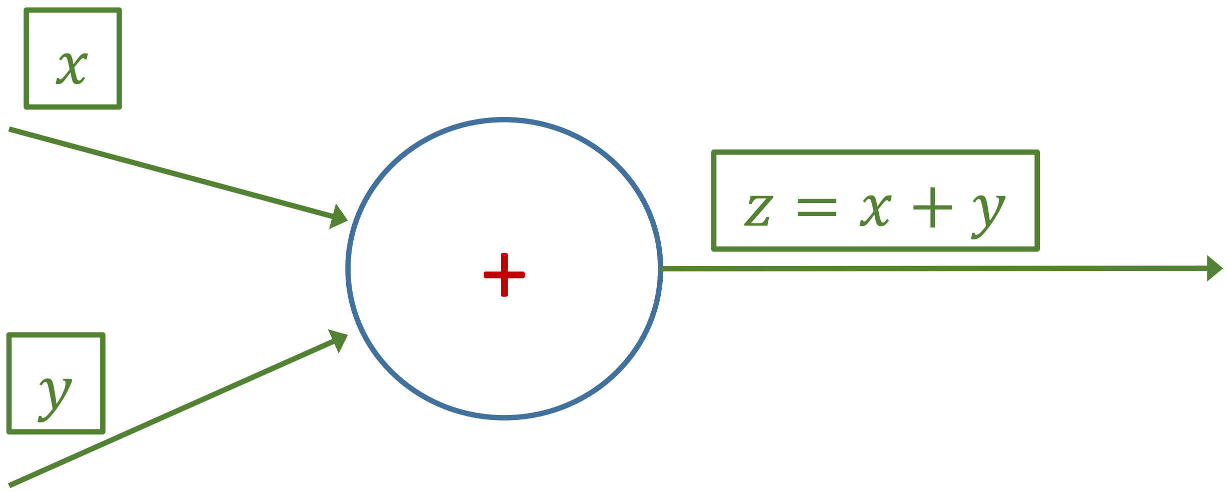
Examples are from http://cs231n.stanford.edu

Computational Graphs

- The descriptive language of deep learning models
- Functional description of the required computation
- Can be instantiated to do two types of computation:
  - Forward computation
  - Backward computation

Computational Graphs

- An acyclic directed graph
- Nodes: A node with an incoming edge is a function of that edge's tail node. This function can be basic arithmetic operations or any functions of which derivatives can be easily computed.
- Edges: An edge represents a function argument.



Computing Gradients: Computational Graphs + Backpropagation

- Key ideas: applying gradient chain rule to unroll gradients through hidden layers in a neural nets

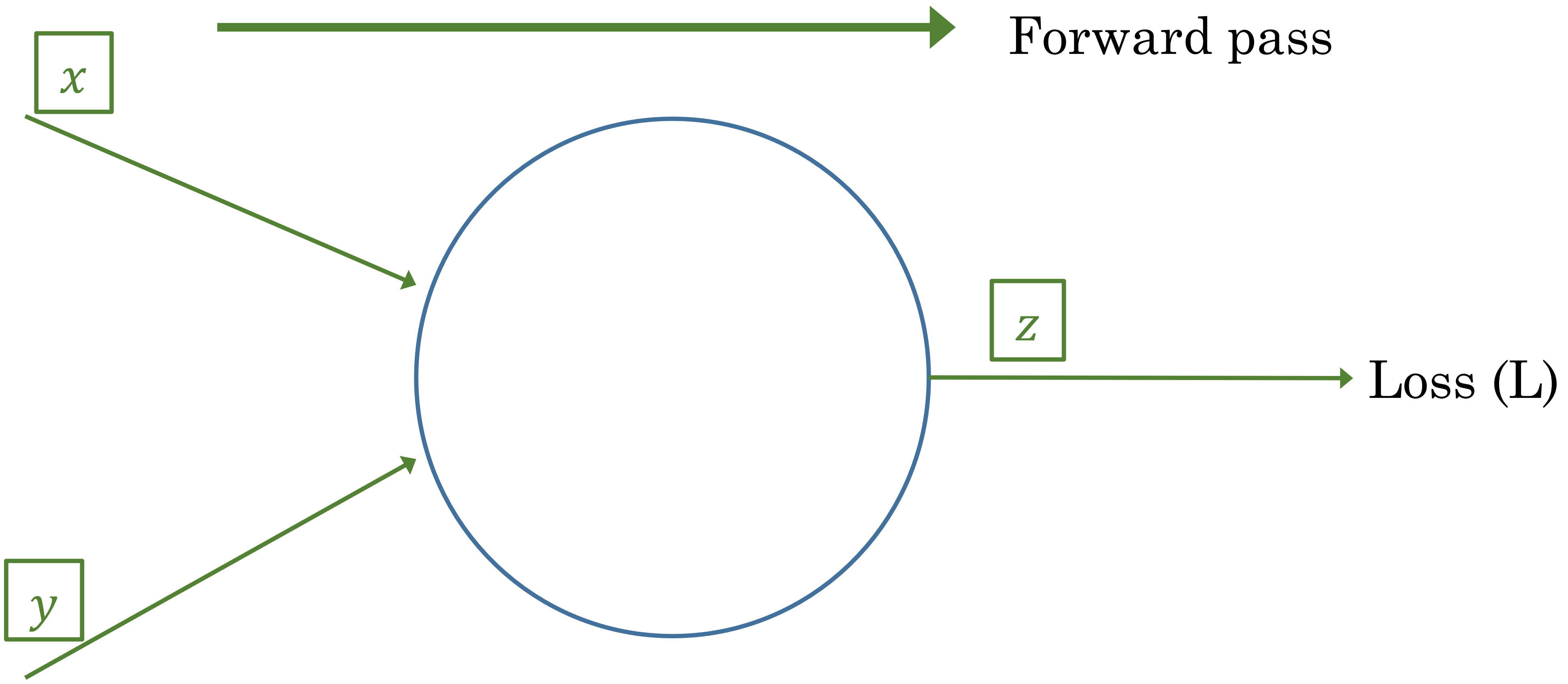
- Chain rule: $\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx}$

- Example: $f(u) = u^2 + 2u,\ g(x) = 3x + 1$
  - Direct computation: $f(g(x)) = g(x)^2 + 2g(x) = (3x + 1)^2 + 2(3x + 1) = 9x^2 + 12x + 3 \rightarrow \frac{df(g(x))}{dx} = 18x + 12$ (Inefficient)
  - Chain rule:

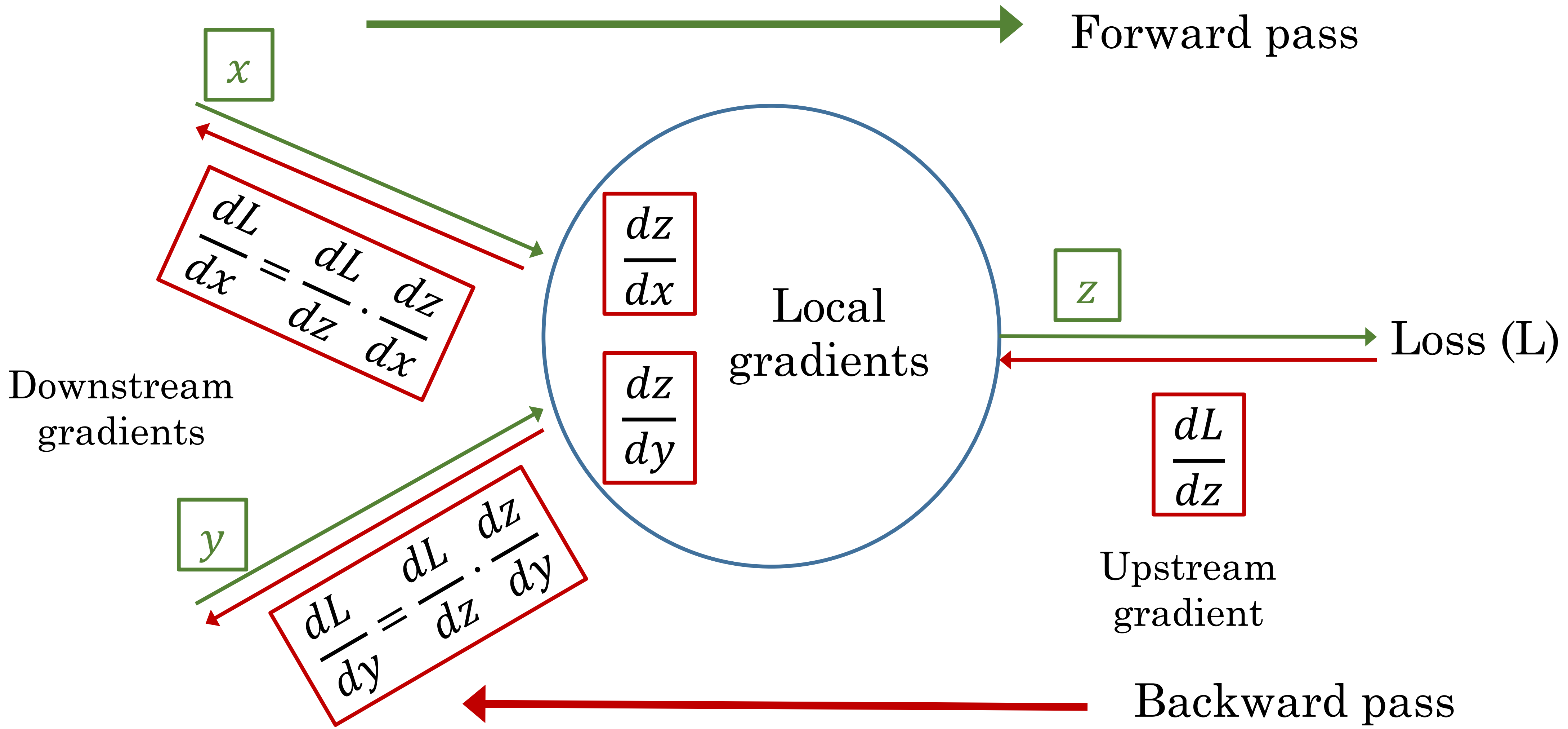
(Much simpler)

$$\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx} = (2g(x) + 2)\times 3 = (6x + 4)\times 3 = 18x + 12$$

Computational Graph + Backpropagation

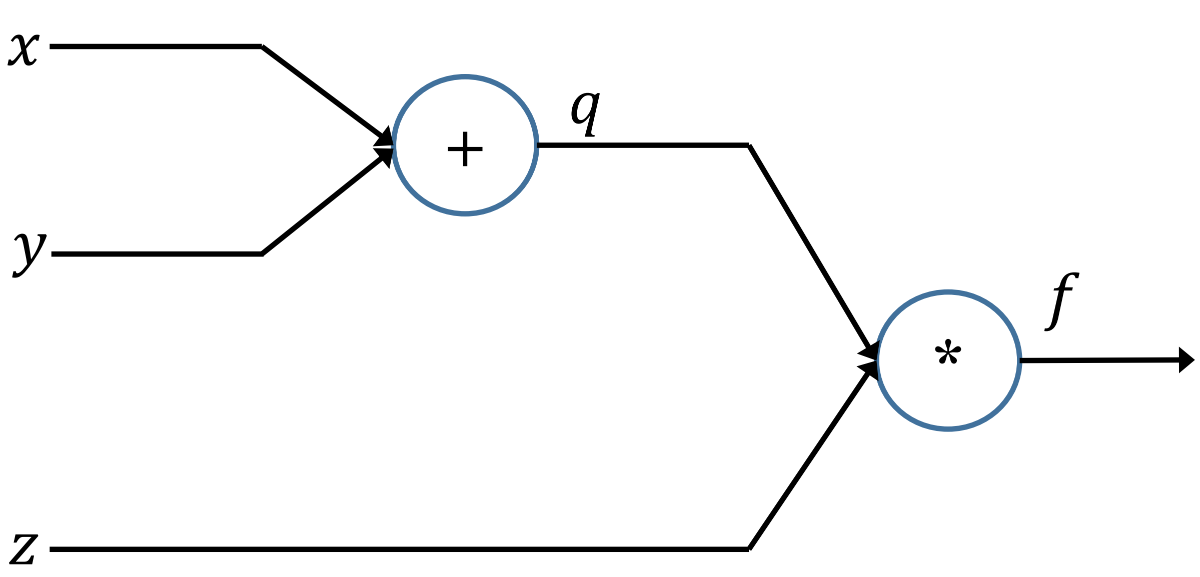


Computational Graph + Backpropagation



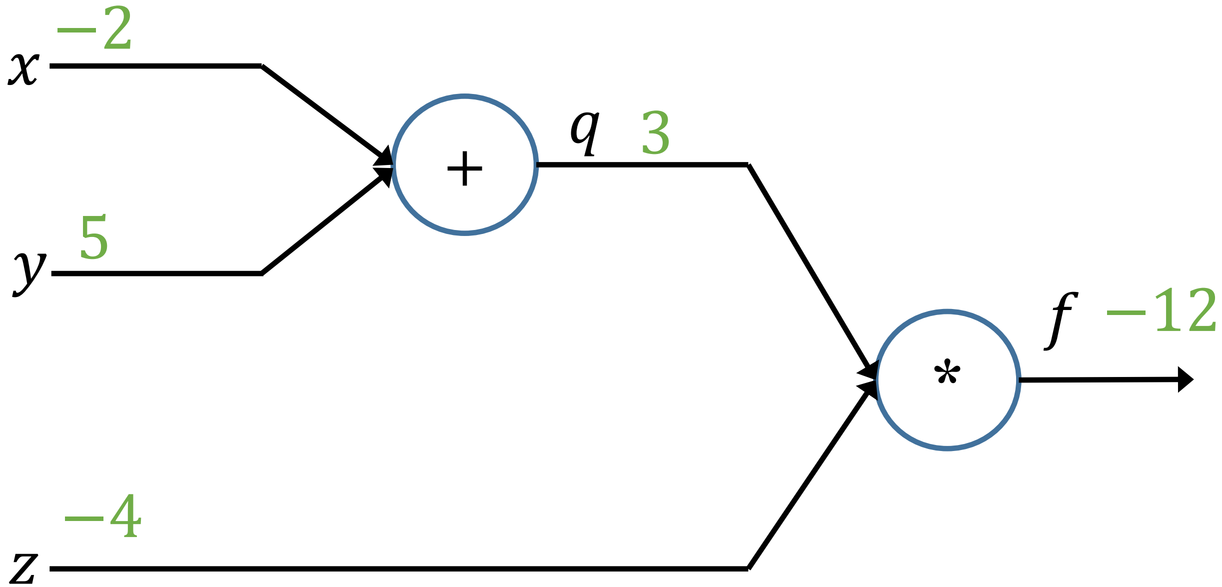
Example

- $q = x + y$
- $f = q * z$



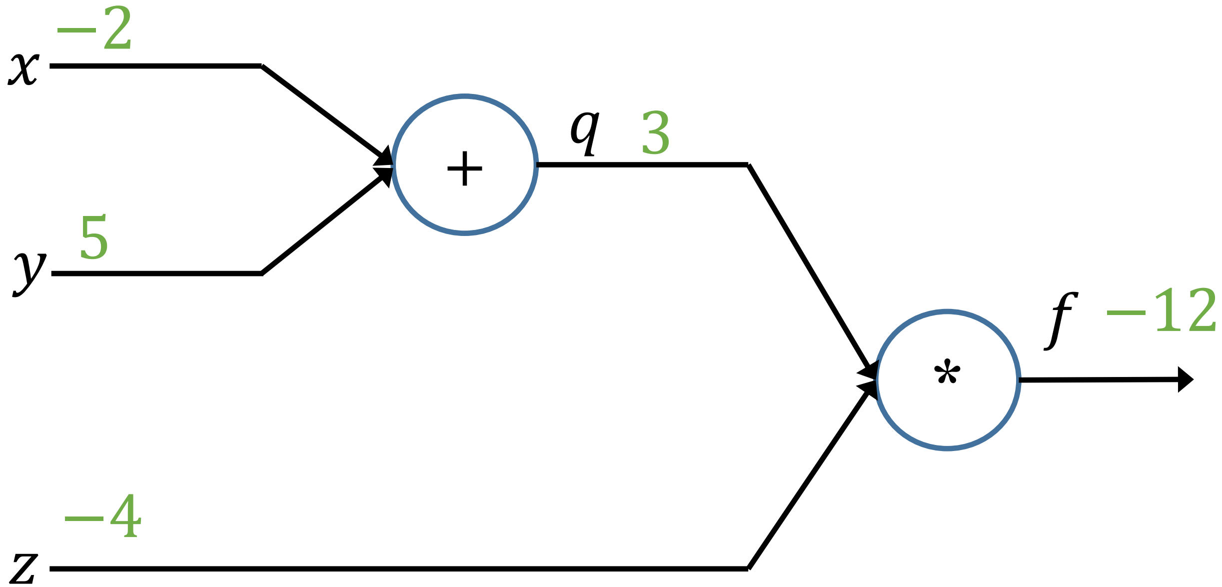
Example

- $q = x + y$
- $f = q * z$
- Input: $x = -2, y = 5, z = -4$
- Goal: $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$



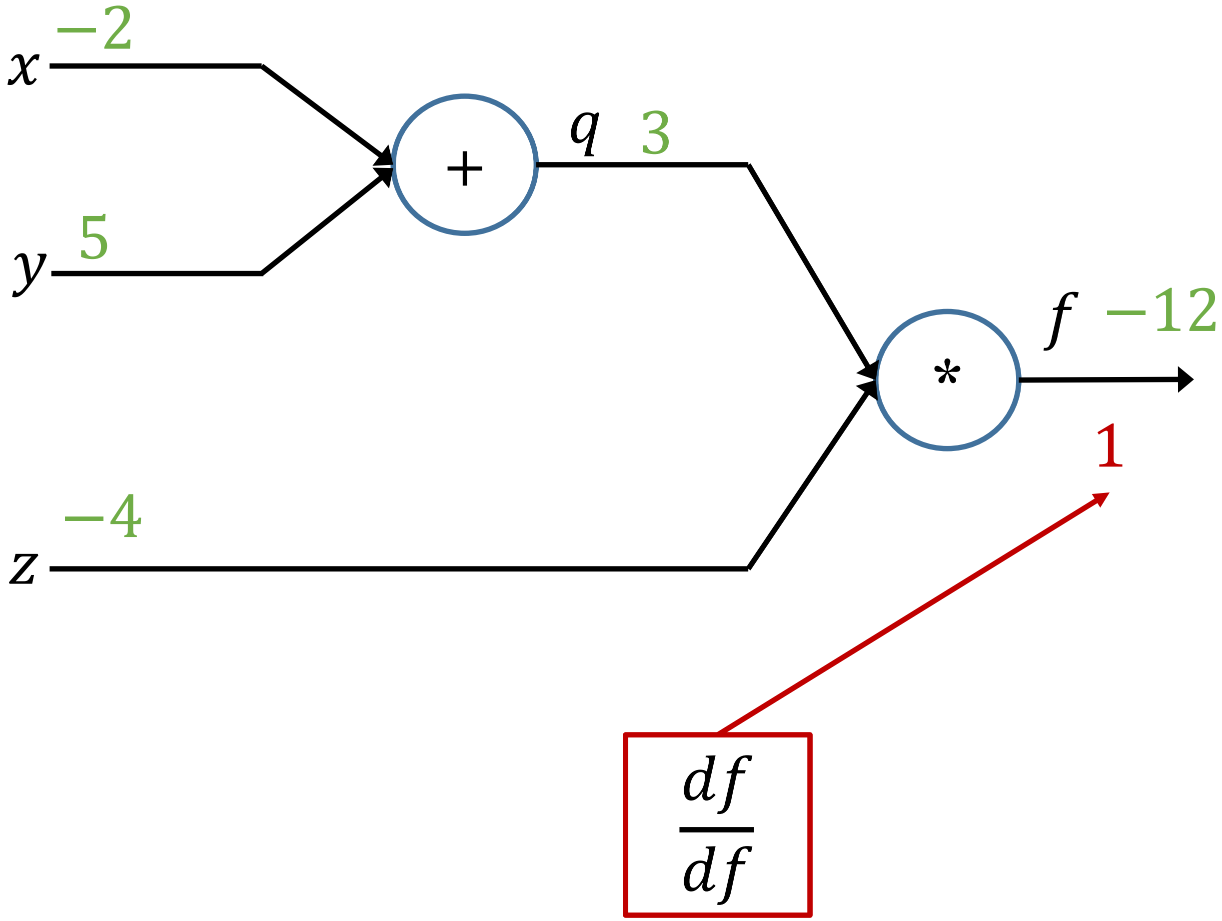
Example

- $q = x + y$
- $f = q * z$
- Input: $x = -2, y = 5, z = -4$
- Goal: $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$



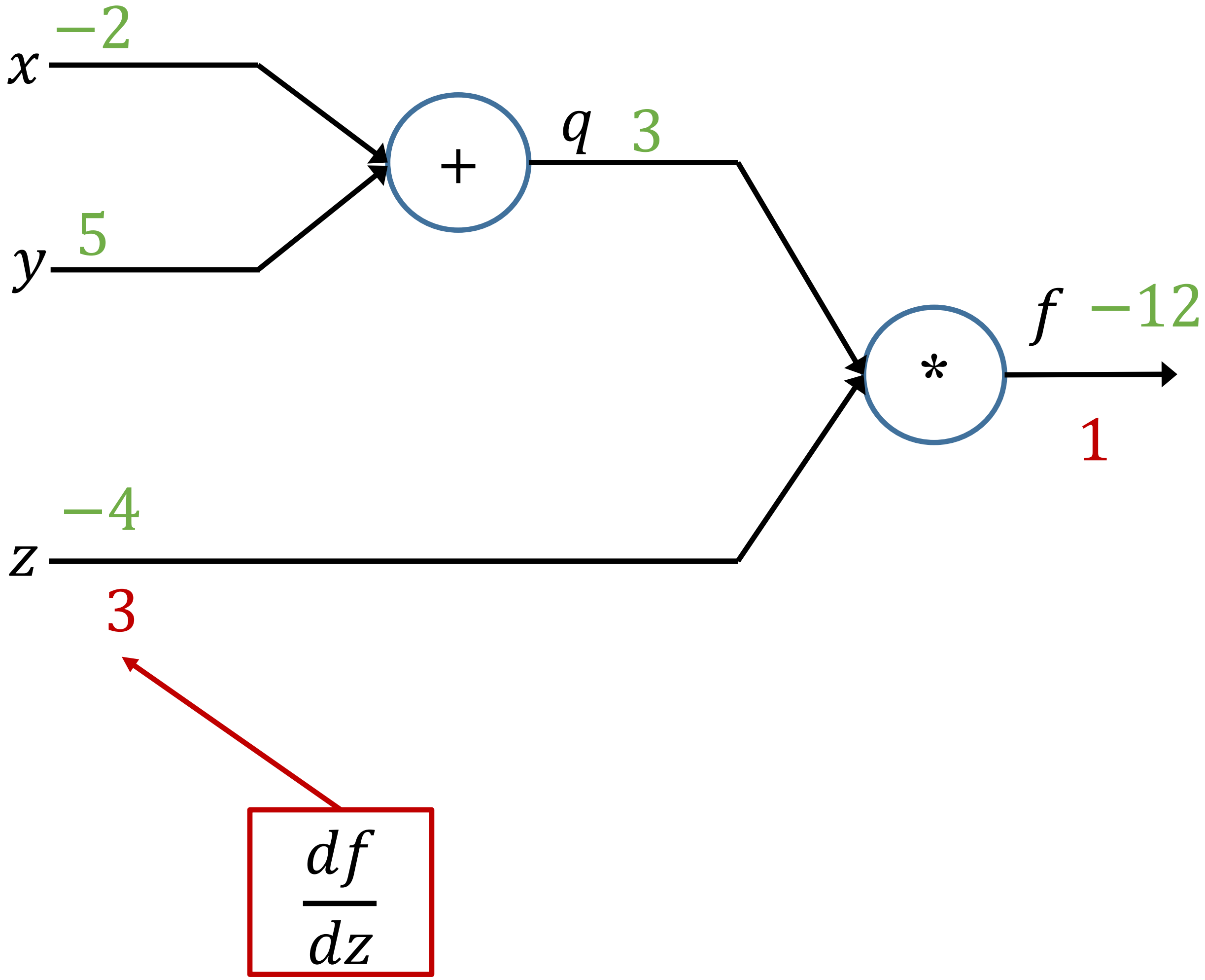
Example

- $q = x + y$
- $f = q * z$
- Input: $x = -2, y = 5, z = -4$
- Goal: $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$



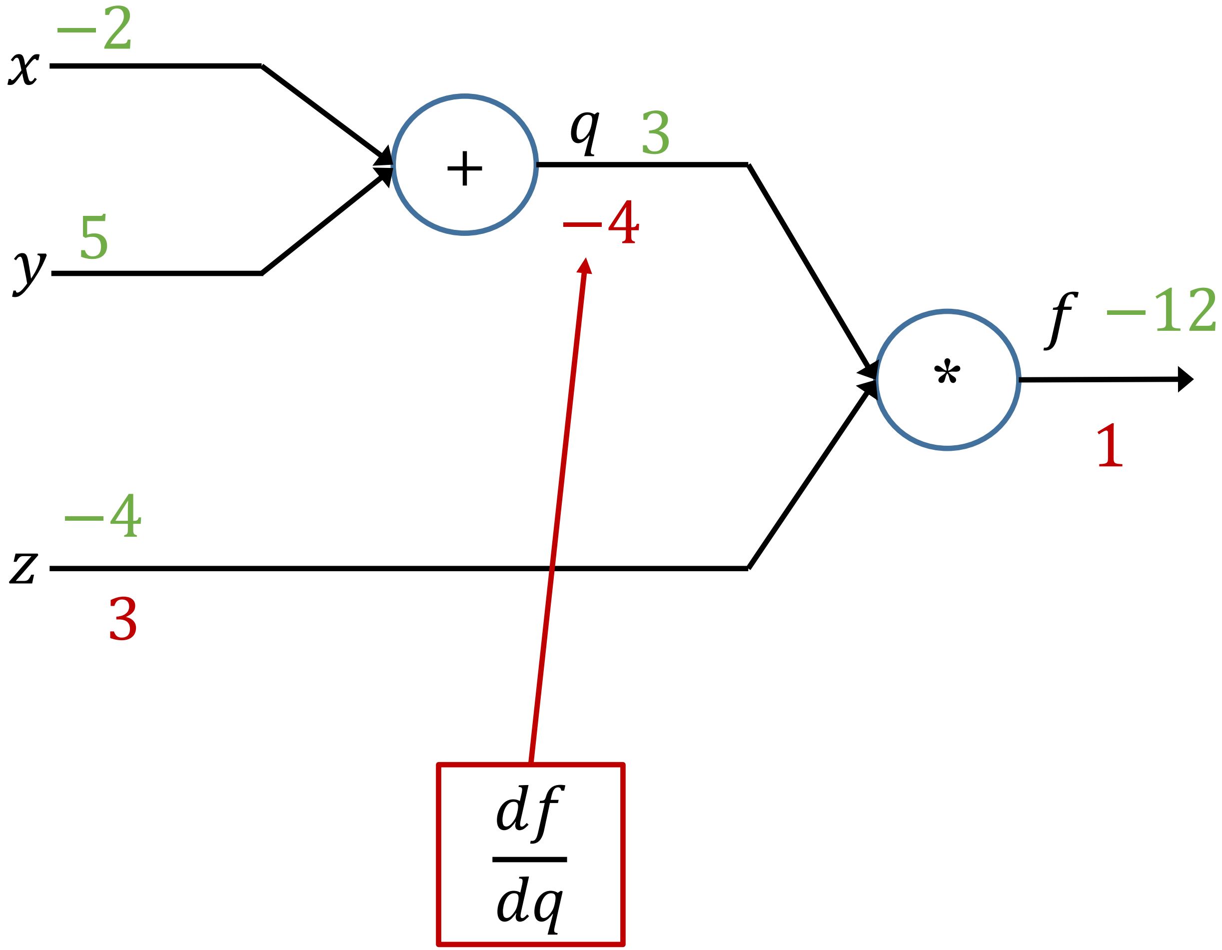
Example

- $q = x + y$
- $f = q * z$
- Input: $x = -2, y = 5, z = -4$
- Goal: $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$



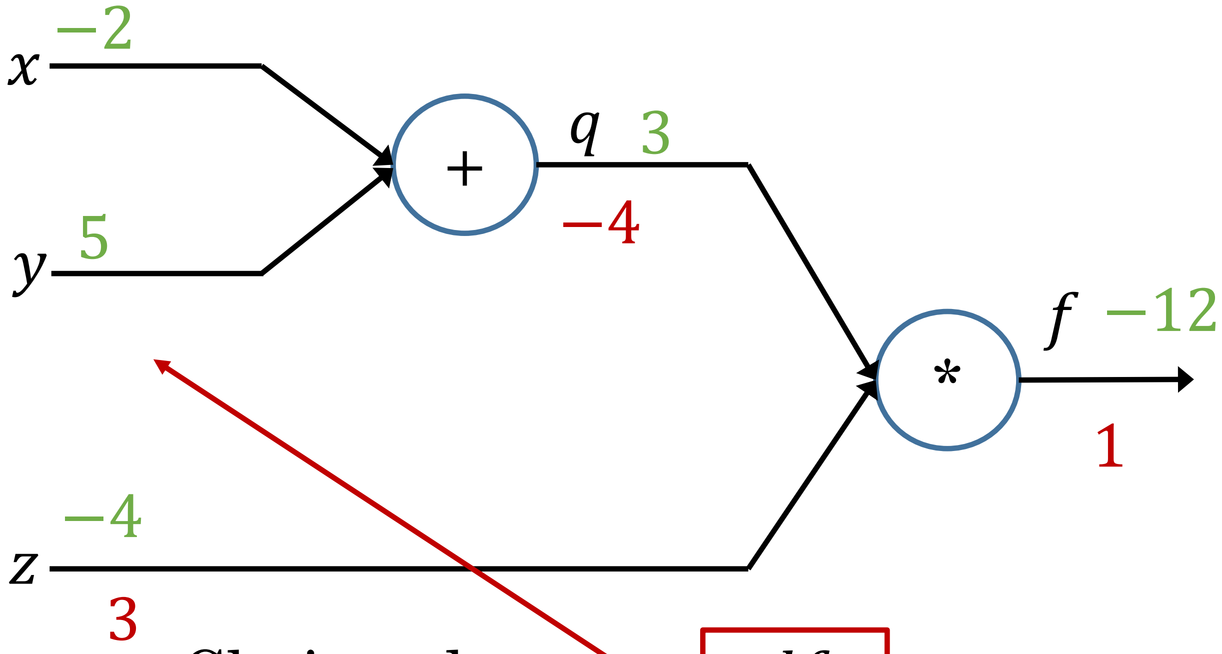
Example

- $q = x + y$
- $f = q * z$
- Input: $x = -2, y = 5, z = -4$
- Goal: $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$



Example

- $q = x + y$
- $f = q * z$
- Input: $x = -2, y = 5, z = -4$
- Goal: $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$



Chain rule

$$\boxed{\frac{df}{dy} = \frac{df}{dq} \cdot \frac{dq}{dy}}$$

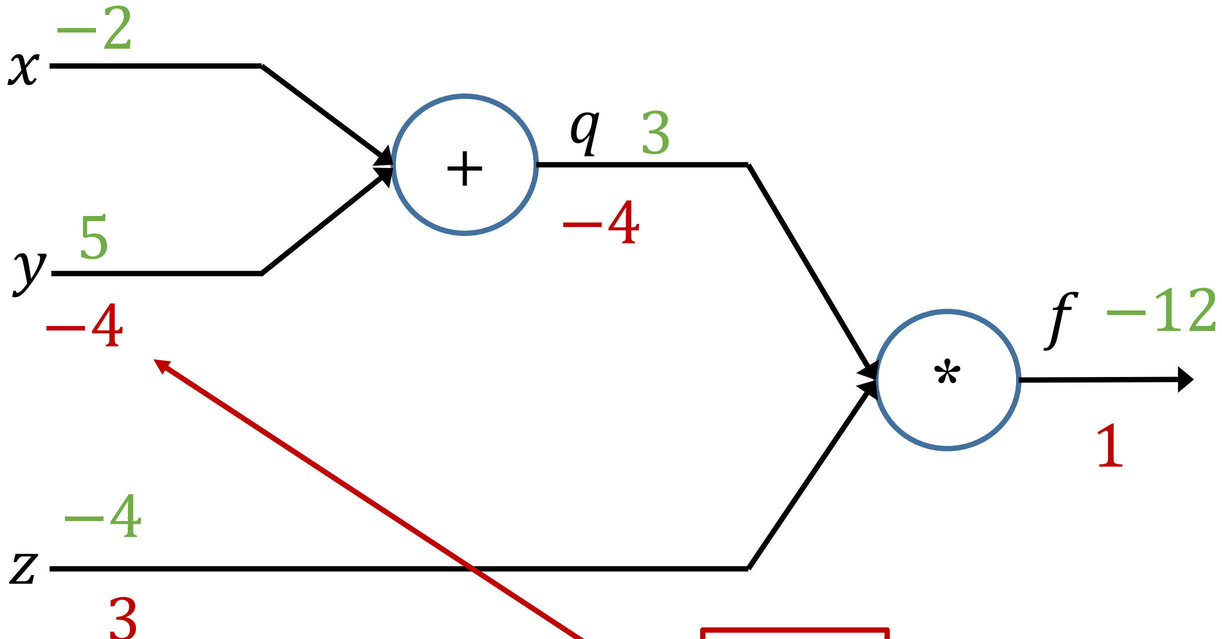
$$\boxed{\frac{df}{dy}}$$

Upstream gradient

Local gradient

Example

- $q = x + y$
- $f = q * z$
- Input: $x = -2, y = 5, z = -4$
- Goal: $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$



Chain rule

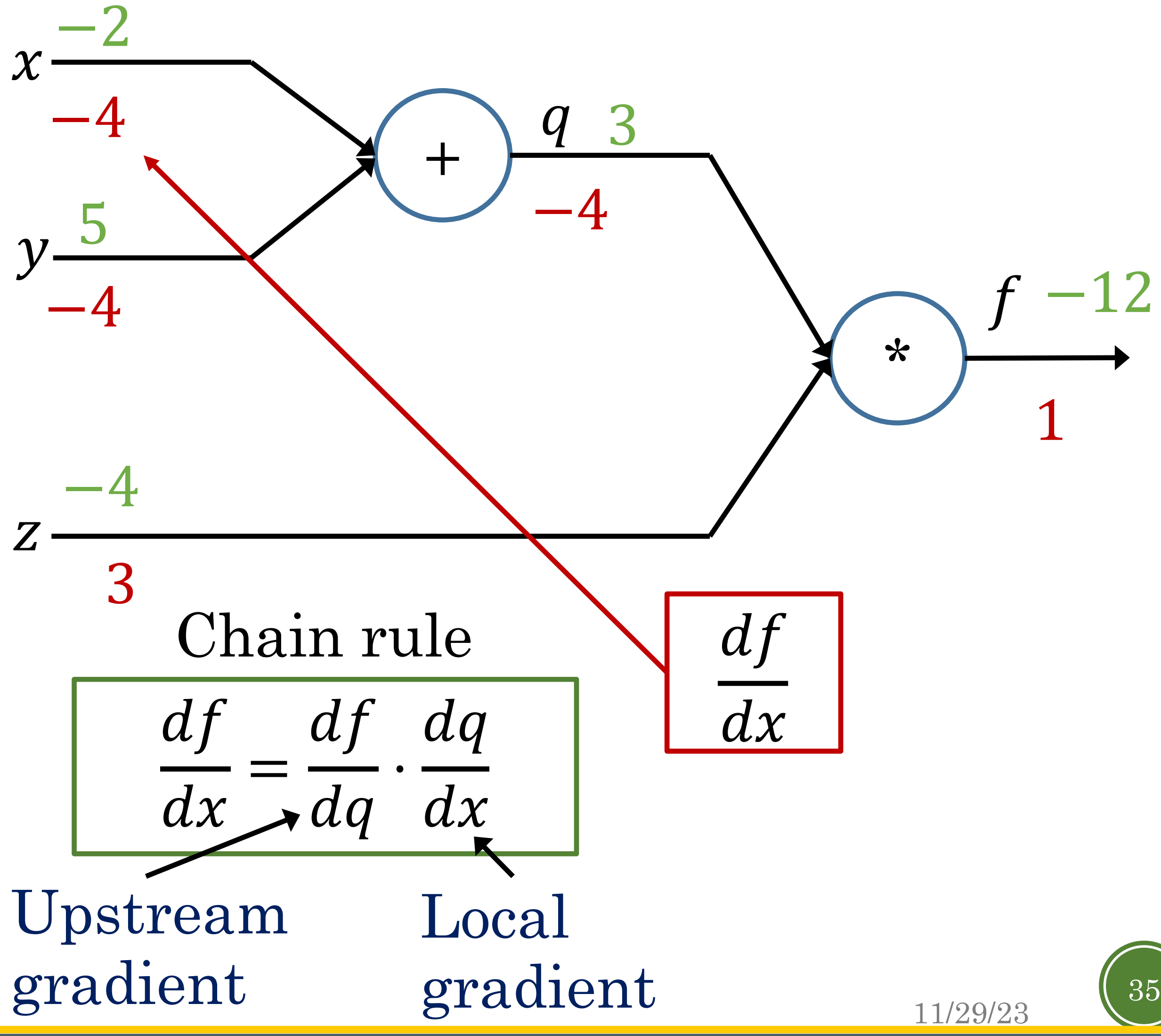
$$\boxed{\frac{df}{dy} = \frac{df}{dq} \cdot \frac{dq}{dy}}$$

Upstream gradient

Local gradient

Example

- $q = x + y$
- $f = q * z$
- Input: $x = -2, y = 5, z = -4$
- Goal: $\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz}$



Pytorch

- A Python-based scientific computing package of which goals are:
  - A replacement for NumPy to use the power of GPUs and other accelerators.
  - An automatic differentiation library that is useful to implement neural networks.

- GPU
  - A processor that has many smaller and more specialized cores
  - Has massive performance when a processing task can be divided up and processed across many cores.

Source: https://pytorch.org/

Pytorch: Fundamental Concepts

- torch.Tensor: Tensors are the central data abstraction in PyTorch. Tensors are specialized data structure that are similar to arrays and matrices.

- torch.autograd: a built-in differentiation engine that supports automatic computation of gradient for any computational graph.

- torch.nn.Module: base class for all neural network modules.

- Pytorch tutorial:
  - https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Pytorch: Example

Create data

```python
import torch
import math

dtype = torch.float
device = torch.device("cpu")

x = torch.linspace(-math.pi, math.pi, 2000, device = device, dtype = dtype)
y = torch.sin(x)

a = torch.randn((), device = device, dtype = dtype, requires_grad=True)
b = torch.randn((), device = device, dtype = dtype, requires_grad=True)
c = torch.randn((), device = device, dtype = dtype, requires_grad=True)
d = torch.randn((), device = device, dtype = dtype, requires_grad=True)

learning_rate = 1e-6

for t in range(2000):
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
    loss = (y_pred - y).pow(2).sum()

    if t % 100 == 99:
        print(t, loss.item())

    loss.backward()
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

    print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

Pytorch: Example

Create random weights

```python
import torch
import math

dtype = torch.float
device = torch.device("cpu")

x = torch.linspace(-math.pi, math.pi, 2000, device = device, dtype = dtype)
y = torch.sin(x)

a = torch.randn((), device = device, dtype = dtype, requires_grad=True)
b = torch.randn((), device = device, dtype = dtype, requires_grad=True)
c = torch.randn((), device = device, dtype = dtype, requires_grad=True)
d = torch.randn((), device = device, dtype = dtype, requires_grad=True)

learning_rate = 1e-6

for t in range(2000):
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
    loss = (y_pred - y).pow(2).sum()

    if t % 100 == 99:
        print(t, loss.item())

    loss.backward()
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

    print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

Pytorch: Example

Forward pass

```python
import torch
import math

dtype = torch.float
device = torch.device("cpu")

x = torch.linspace(-math.pi, math.pi, 2000, device = device, dtype = dtype)
y = torch.sin(x)

a = torch.randn((), device = device, dtype = dtype, requires_grad=True)
b = torch.randn((), device = device, dtype = dtype, requires_grad=True)
c = torch.randn((), device = device, dtype = dtype, requires_grad=True)
d = torch.randn((), device = device, dtype = dtype, requires_grad=True)

learning_rate = 1e-6

for t in range(2000):
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
    loss = (y_pred - y).pow(2).sum()

    if t % 100 == 99:
        print(t, loss.item())

    loss.backward()
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

Pytorch: Example

Backward pass

```python
import torch
import math

dtype = torch.float
device = torch.device("cpu")

x = torch.linspace(-math.pi, math.pi, 2000, device = device, dtype = dtype)
y = torch.sin(x)

a = torch.randn((), device = device, dtype = dtype, requires_grad=True)
b = torch.randn((), device = device, dtype = dtype, requires_grad=True)
c = torch.randn((), device = device, dtype = dtype, requires_grad=True)
d = torch.randn((), device = device, dtype = dtype, requires_grad=True)

learning_rate = 1e-6

for t in range(2000):
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
    loss = (y_pred - y).pow(2).sum()

    if t % 100 == 99:
        print(t, loss.item())

    loss.backward()
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

    print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

Pytorch: Example

```
import torch
import math

dtype = torch.float
device = torch.device("cpu")

x = torch.linspace(-math.pi, math.pi, 2000, device = device, dtype = dtype)
y = torch.sin(x)

a = torch.randn((), device = device, dtype = dtype, requires_grad=True)
b = torch.randn((), device = device, dtype = dtype, requires_grad=True)
c = torch.randn((), device = device, dtype = dtype, requires_grad=True)
d = torch.randn((), device = device, dtype = dtype, requires_grad=True)

learning_rate = 1e-6

for t in range(2000):
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
    loss = (y_pred - y).pow(2).sum()

    if t % 100 == 99:
        print(t, loss.item())

    loss.backward()
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

Gradient descent

Pytorch：Example

Reset gradient before next round of forward-backward pass

```
import torch
import math

dtype = torch.float
device = torch.device("cpu")

x = torch.linspace(-math.pi, math.pi, 2000, device = device, dtype = dtype)
y = torch.sin(x)

a = torch.randn((), device = device, dtype = dtype, requires_grad=True)
b = torch.randn((), device = device, dtype = dtype, requires_grad=True)
c = torch.randn((), device = device, dtype = dtype, requires_grad=True)
d = torch.randn((), device = device, dtype = dtype, requires_grad=True)

learning_rate = 1e-6

for t in range(2000):
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
    loss = (y_pred - y).pow(2).sum()

    if t % 100 == 99:
        print(t, loss.item())

    loss.backward()
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

    print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

Pytorch: Neural Network

Create data

```
import torch
import math

x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-6
for t in range(2000):
    y_pred = model(xx)

    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

linear_layer = model[0]
```

Pytorch: Neural Network

Create predictive model

```python
import torch
import math

x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-6
for t in range(2000):
    y_pred = model(xx)

    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

linear_layer = model[0]
```

Pytorch: Neural Network

Define loss function

```python
import torch
import math

x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-6
for t in range(2000):
    y_pred = model(xx)

    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

linear_layer = model[0]
```

Pytorch: Neural Network

Forward pass

```
import torch
import math

x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-6
for t in range(2000):
    y_pred = model(xx)

    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

linear_layer = model[0]
```

Pytorch: Neural Network

```python
import torch
import math

x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-6
for t in range(2000):
    y_pred = model(xx)

    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

linear_layer = model[0]
```

Reset gradient and
run backward pass

Pytorch: Neural Network

```python
import torch
import math

x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-6
for t in range(2000):
    y_pred = model(xx)

    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad

linear_layer = model[0]
```

Run gradient descent

Pytorch: Optimizer

Choose an optimizer

```python
import torch
import math

x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
for t in range(2000):

    y_pred = model(xx)
    loss = loss_fn(y_pred, y)

    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()

    loss.backward()

    optimizer.step()
```

Pytorch: Optimizer

```
import torch
import math

x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

loss_fn = torch.nn.MSELoss(reduction='sum')

learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
for t in range(2000):

    y_pred = model(xx)
    loss = loss_fn(y_pred, y)

    if t % 100 == 99:
        print(t, loss.item())
    model.zero_grad()

    loss.backward()

    optimizer.step()
```

Update weights

Pytorch: Define a New Module

Define a neural network model as a module

```python
import torch


class Net(torch.nn.Module):
    def __init__(self, d_in, d_hidden, d_out):
        super(Net, self).__init__()
        self.ln1 = torch.nn.Linear(d_in, d_hidden)
        self.ln2 = torch.nn.Linear(d_hidden, d_out)

    def forward(self, x):
        h_relu = self.ln1(x).clamp(min=0)
        y_pred = self.ln2(h_relu)
        return y_pred


train_size, d_in, d_hidden, d_out = 64, 1000, 100, 10
x = torch.randn(train_size, d_in)
y = torch.randn(train_size, d_out)

model = Net(d_in, d_hidden, d_out)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
loss_fn = torch.nn.MSELoss(reduction='sum')

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    if t % 50 == 49:
        print(t, loss.item())

    model.zero_grad()
    loss.backward()

    optimizer.step()

```

Pytorch: Define a New Module

Construct and train a model

```python
import torch


class Net(torch.nn.Module):
    def __init__(self, d_in, d_hidden, d_out):
        super(Net, self).__init__()
        self.ln1 = torch.nn.Linear(d_in, d_hidden)
        self.ln2 = torch.nn.Linear(d_hidden, d_out)

    def forward(self, x):
        h_relu = self.ln1(x).clamp(min=0)
        y_pred = self.ln2(h_relu)
        return y_pred


train_size, d_in, d_hidden, d_out = 64, 1000, 100, 10
x = torch.randn(train_size, d_in)
y = torch.randn(train_size, d_out)

model = Net(d_in, d_hidden, d_out)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
loss_fn = torch.nn.MSELoss(reduction='sum')

for t in range(500):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    if t % 50 == 49:
        print(t, loss.item())

    model.zero_grad()
    loss.backward()

    optimizer.step()
```