Training Neural Networks

CMPUT 366: Intelligent Systems

GBC §6.5

Lecture Outline

1. Recap
2. Gradient Descent for Neural Networks
3. Automatic Differentiation
4. Back-Propagation

Recap: Nonlinear Features

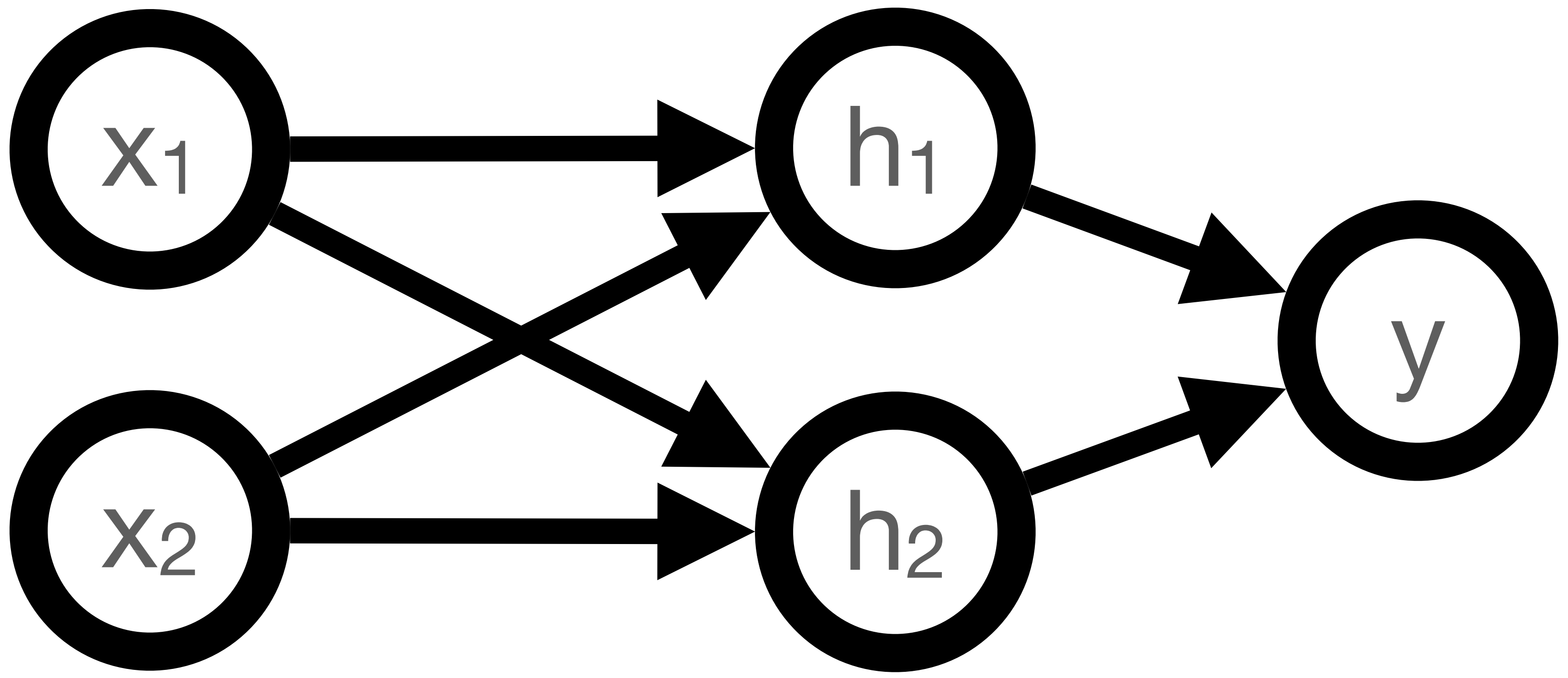
$$y = f(\mathbf{x}; \mathbf{w}) = g(\mathbf{w}^T \mathbf{x}) = g\left(\sum_{i=1}^{n} w_i x_i\right)$$

Generalized Linear Model: Learn a linear model on **richer inputs**

1. Define a **feature mapping** $\phi(\mathbf{x})$ that returns **functions** of the original inputs
2. Learn a linear model of the **features** instead of the **inputs**

$$y = f(\mathbf{x}; \mathbf{w}) = g(\mathbf{w}^T \phi(\mathbf{x})) = g\left(\sum_{i=1}^{n} w_i [\phi(\mathbf{x})]_i\right)$$

Recap:
Feedforward Neural Network



$$h_1(\mathbf{x}; \mathbf{w}^{(1)}, b^{(1)}) = g\left(b^{(1)} + \sum_{i=1}^{n} w_i^{(1)} x_i\right)$$

$$\begin{aligned} y(\mathbf{x}; \mathbf{w}, \mathbf{b}) &= g\left(b^{(y)} + \sum_{i=1}^{n} w_i^{(y)} h_i(\mathbf{x}_i; \mathbf{w}^{(i)}, b^{(i)})\right) \\ &= g\left(b^{(y)} + \sum_{i=1}^{n} w_i^{(y)} g\left(b^{(i)} + \sum_{j=1}^{n} w_j^{(i)} x_j\right)\right) \end{aligned}$$

- A **neural network** is many **units** **composed** together
- **Feedforward neural network:** Units arranged into **layers**
  - Each layer takes outputs of **previous layer** as its **inputs**

Recap: Chain Rule of Calculus

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

i.e,

$$h(x) = f(g(x)) \implies h'(x) = f'(g(x))g'(x)$$

If we know formulas for the derivatives of **components** of a function, then we can build up the derivative of their composition mechanically

Neural Network Parameters

$$y = f(x; \theta)$$

A neural network is just a **supervised model**

- It is a function that takes **inputs** $\mathbf{x}$, and computes an output y based on parameters θ
- **Question:** What is θ in a feedforward neural network?

Training Neural Networks

- Specify a **loss** L and a set of **training examples:**

$$E = (\mathbf{x}^{(1)}, y^{(1)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})$$

- Training by **gradient descent**:

1. Compute **loss** on training data: $L(\mathbf{W}, \mathbf{b}) = \sum_i \ell \left(\underbrace{f(\mathbf{x}^{(i)}; \mathbf{W}, \mathbf{b})}_{\text{Prediction}}, \underbrace{y^{(i)}}_{\text{Target}} \right)$

2. Compute **gradient** of loss: $\boxed{\nabla L(\mathbf{W}, \mathbf{b})}$

3. **Update parameters** to make loss smaller:

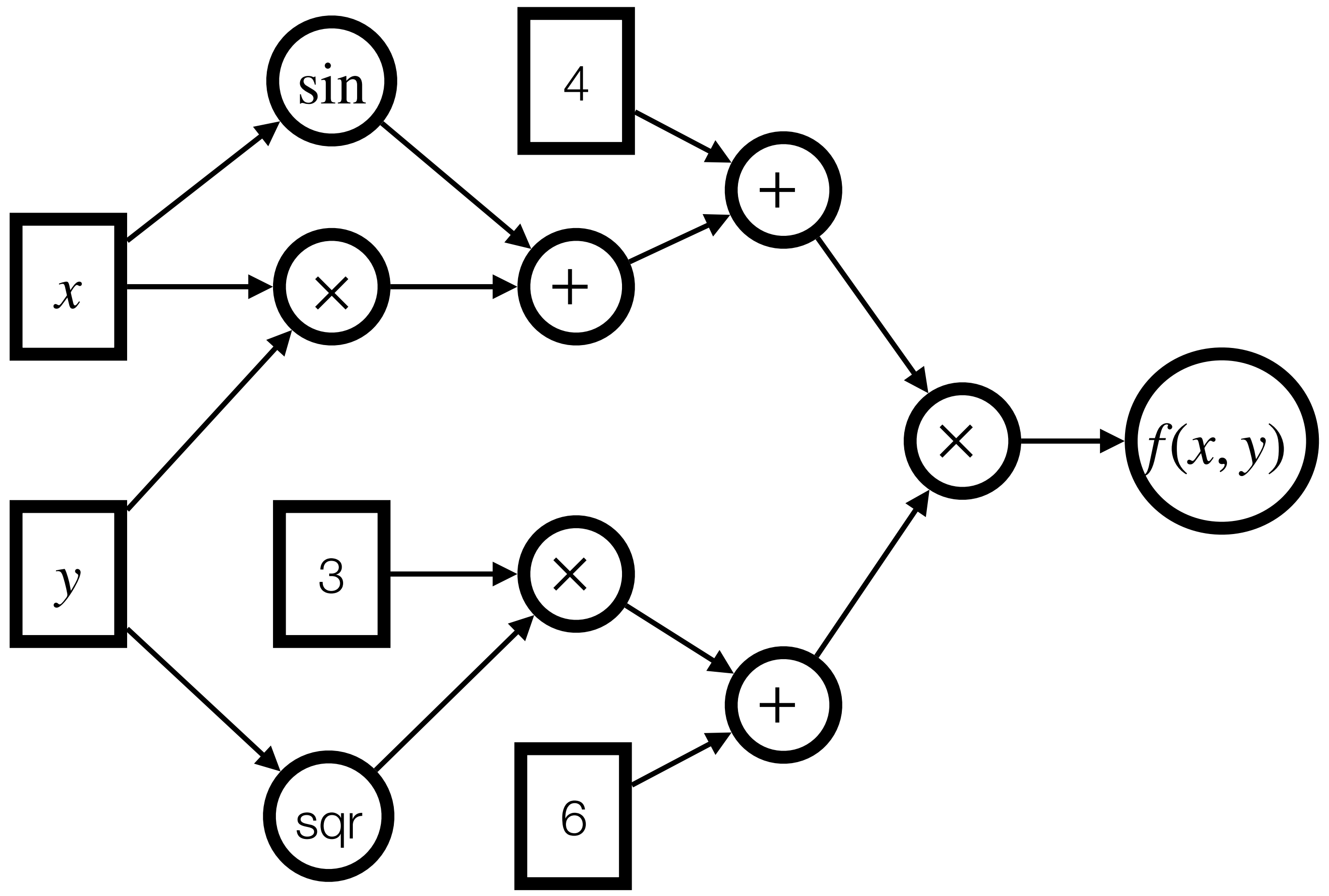
$$\begin{bmatrix} \mathbf{W}^{new} \\ \mathbf{b}^{new} \end{bmatrix} = \begin{bmatrix} \mathbf{W}^{old} \\ \mathbf{b}^{old} \end{bmatrix} - \eta \nabla L(\mathbf{W}^{old}, \mathbf{b}^{old})$$

Three Representations

A function $f(x, y)$ can be represented in multiple ways:

1. As a **formula:**
   $f(x, y) = (xy + \sin x + 4)(3y^2 + 6)$

2. As a **computational graph:**



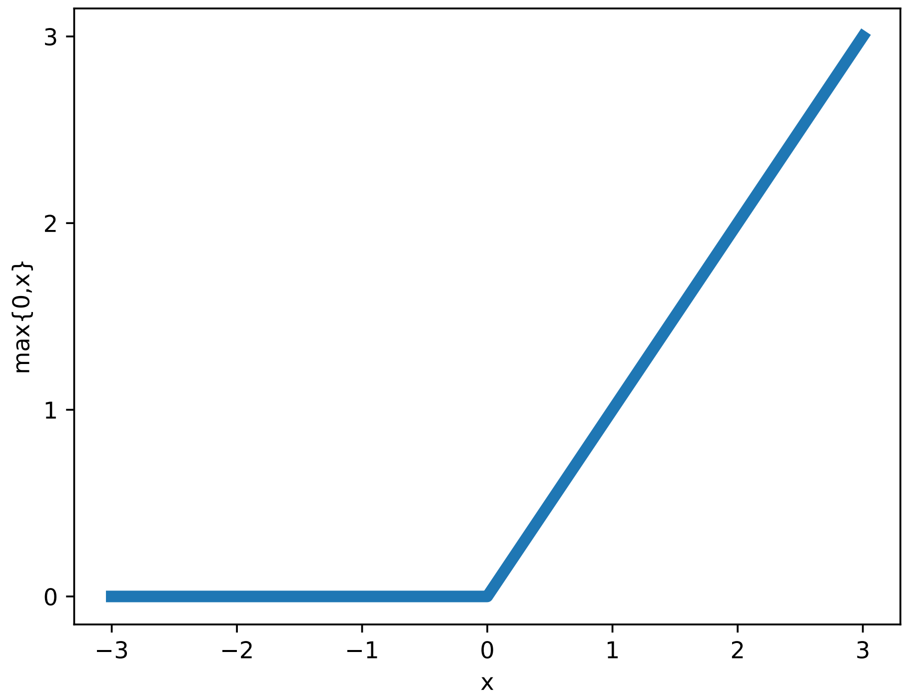
3. As a **finite numerical algorithm**

$$
\begin{aligned}
s_1 &= x \\
s_2 &= y \\
s_3 &= s_1 \times s_2 \\
s_4 &= \sin(s_1) \\
s_5 &= s_3 + s_4 \\
s_6 &= s_5 + 4 \\
s_7 &= \mathrm{sqr}(s_2) \\
s_8 &= 3 \times s_7 \\
s_9 &= s_8 + 6 \\
s_{10} &= s_6 \times s_9
\end{aligned}
$$

example from [Bücker et al., 2006]

Symbolic Differentiation

$$
\begin{aligned}
z &= f(y) \\
y &= f(x) \\
x &= f(w)
\end{aligned}
\qquad z = f(f(f(w)))
\qquad
\begin{aligned}
\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
&= f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$



$$f(w) = \begin{cases} w & \text{if } w > 0 \\ 0 & \text{otherwise.} \end{cases}$$

- We can differentiate a nested **formula** by recursively applying the **chain rule** to derive a **new formula** for the gradient
- **Problem:** This can result in a lot of repeated subexpressions
- **Question:** What happens if the nested function is defined **piecewise**?

Automatic Differentiation: Forward Mode

- The forward mode converts a finite numerical algorithm for computing a function into a finite numerical algorithm for computing the **function's derivative**

- For each step, a new step is constructed representing the derivative of the corresponding step in the original program:

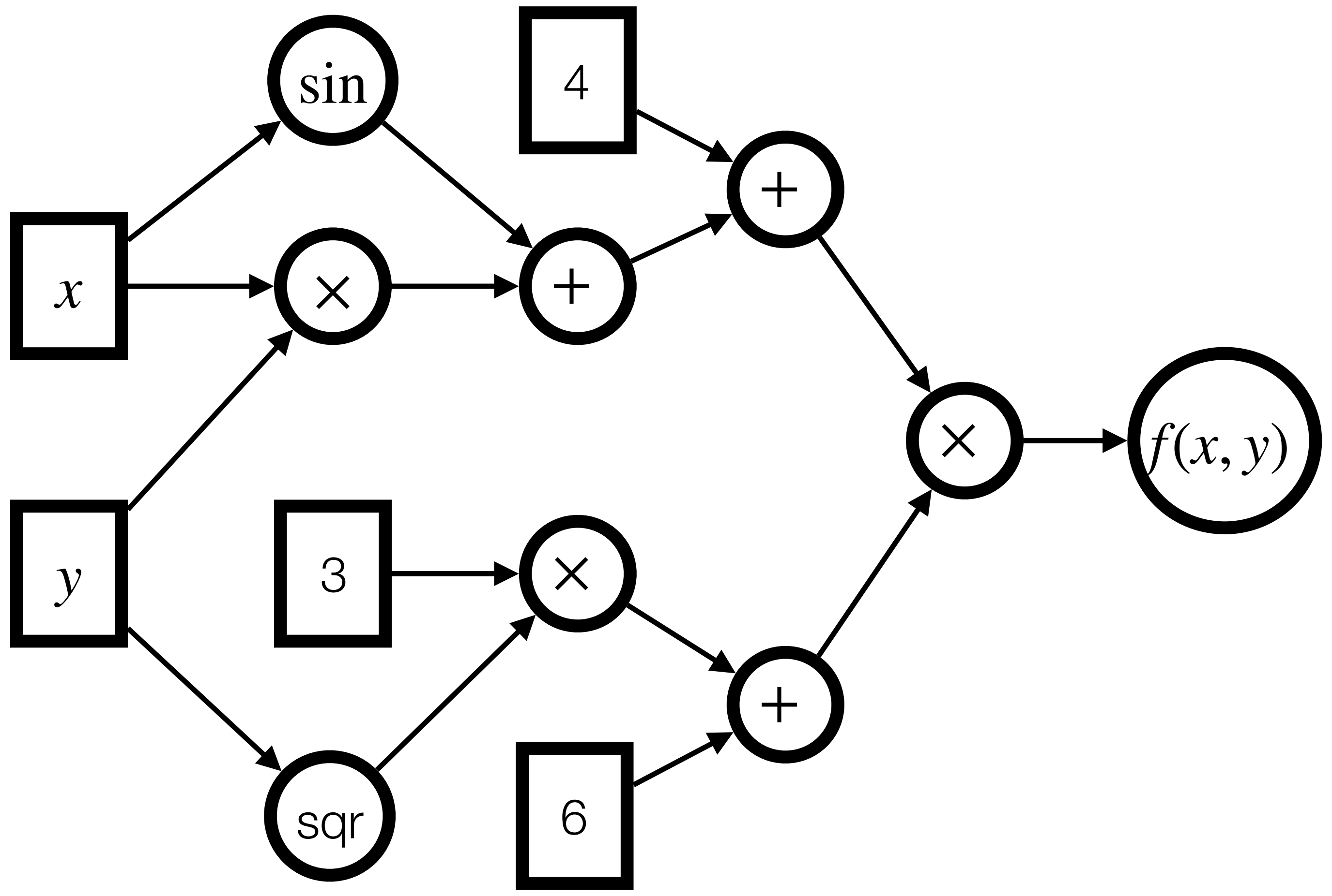
$$
\begin{aligned}
s_1 &= x \\
s_2 &= y \\
s_3 &= s_1 + s_2 \\
s_4 &= s_1 \times s_2 \\
&\vdots
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
s_1' &= 1 \\
s_2' &= 0 \\
s_3' &= s_1' + s_2' \\
s_4' &= s_1 \times s_2' + s_1' \times s_2 \\
&\vdots
\end{aligned}
$$

- To compute the partial derivative $\dfrac{\partial s_n}{\partial s_1}$, set $s_1' = 1$ and $s_2' = 0$ and run augmented algorithm

- This takes roughly twice as long to run as the original algorithm (**why?**)

Forward Mode Example

Let's compute $\left.\dfrac{\partial f}{\partial y}\right|_{x=2,y=8}$ using forward mode:



$$
\begin{aligned}
s_1 &= x & &= 2 \\
s_2 &= y & &= 8 \\
s_3 &= s_1 \times s_2 & &= 16 \\
s_4 &= \sin(s_1) & &\approx 0.034 \\
s_5 &= s_3 + s_4 & &= 16.034 \\
s_6 &= s_5 + 4 & &= 20.034 \\
s_7 &= \mathrm{sqr}(s_2) & &= 64 \\
s_8 &= 3 \times s_7 & &= 192 \\
s_9 &= s_8 + 6 & &= 198 \\
s_{10} &= s_6 \times s_9 & &= 3966.732
\end{aligned}
$$

$$
\begin{aligned}
s_1' &= 0 \\
s_2' &= 1 \\
s_3' &= s_1 \times s_2' + s_1' \times s_2 = 2 \\
s_4' &= \cos(s_1) \times s_1' = 0 \\
s_5' &= s_3' + s_4' = 2 \\
s_6' &= s_5' = 2 \\
s_7' &= s_2' \times 2 \times s_2 = 16 \\
s_8' &= 3 \times s_7' = 48 \\
s_9' &= s_8' = 48 \\
s_{10}' &= s_6 \times s_9' + s_6' \times s_9 = 1357.632
\end{aligned}
$$

Forward Mode Performance

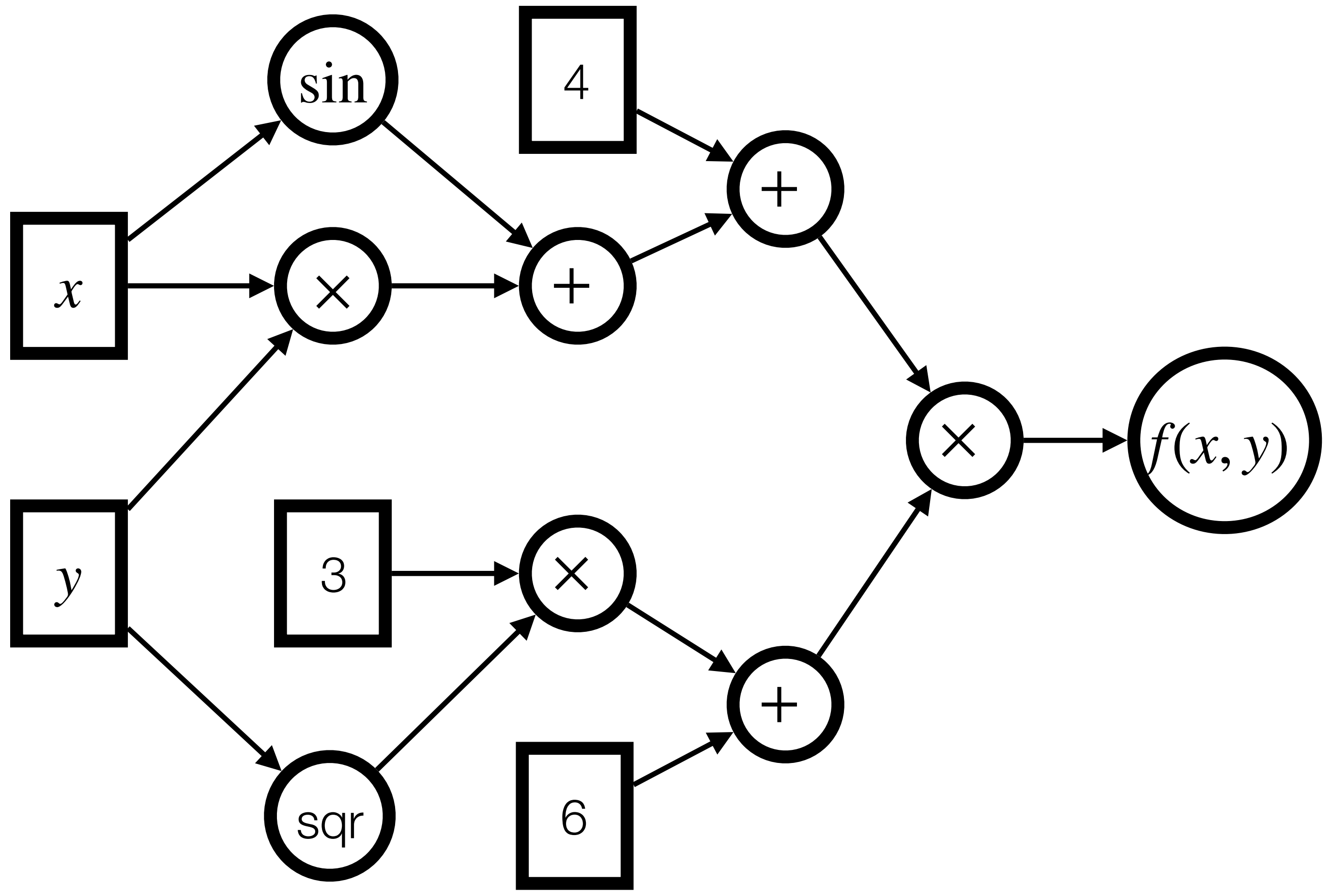
- To compute the full gradient of a function m variables requires computing m partial derivatives
- In forward mode, this requires m forward passes
- For our toy examples, that means running the forward pass twice
- Neural networks can easily have **thousands** of parameters
- We don't want to run the network thousands of times for each gradient update!

Automatic Differentiation: Backward Mode

- **Forward mode** sweeps through the graph, computing $s_i' = \dfrac{\partial s_i}{\partial s_1}$ for each s_i
    - The **numerator varies**, and the **denominator is fixed**
- **Backward mode** does the opposite:
    - For each s_i, computes the **local gradient** $\overline{s_i} = \dfrac{\partial s_n}{\partial s_i}$
    - The **numerator is fixed**, and the **denominator varies**
- At the end, we have computed $\overline{x_i} = \dfrac{\partial s_n}{\partial x_i}$ for each input x_i

Backward Mode Example

Let's compute $\left.\frac{\partial f}{\partial x}\right|_{x=2,y=8}$ and $\left.\frac{\partial f}{\partial y}\right|_{x=2,y=8}$ using backward mode:



$$
\begin{aligned}
s_1 &= x && = 2\\
s_2 &= y && = 8\\
s_3 &= s_1 \times s_2 && = 16\\
s_4 &= \sin(s_1) && \approx 0.034\\
s_5 &= s_3 + s_4 && = 16.034\\
s_6 &= s_5 + 4 && = 20.034\\
s_7 &= \text{sqr}(s_2) && = 64\\
s_8 &= 3 \times s_7 && = 192\\
s_9 &= s_8 + 6 && = 198\\
s_{10} &= s_6 \times s_9 && = 3966.732
\end{aligned}
$$

$$\vdots$$

$$\overline{s_6} = \frac{\partial s_{10}}{\partial s_6} = \frac{\partial s_{10}}{\partial s_7}\frac{\partial s_7}{\partial s_6} = \overline{s_7}0 = 0$$

$$\overline{s_7} = \frac{\partial s_{10}}{\partial s_7} = \frac{\partial s_{10}}{\partial s_8}\frac{\partial s_8}{\partial s_7} = \overline{s_8}3 = 60.102$$

$$\overline{s_8} = \frac{\partial s_{10}}{\partial s_8} = \frac{\partial s_{10}}{\partial s_9}\frac{\partial s_9}{\partial s_8} = \overline{s_9}1 = 20.034$$

$$\overline{s_9} = \frac{\partial s_{10}}{\partial s_9} = \frac{\partial s_{10}}{\partial s_{10}}\frac{\partial s_{10}}{\partial s_9} = \overline{s_{10}}s_6 = 20.034$$

$$\overline{s_{10}} = \frac{\partial s_{10}}{\partial s_{10}} = 1$$

Back-Propagation

$$L(\mathbf{W}, \mathbf{b}) = \sum_i \ell\left(f(\mathbf{x}^{(i)}; \mathbf{W}, \mathbf{b}), y^{(i)}\right)$$

Back-propagation is simply automatic differentiation in **backward mode**, used to compute the gradient $\nabla_{\mathbf{W},\mathbf{b}} L$ of the **loss function** with respect to its **parameters** $\mathbf{W}, \mathbf{b}$:

1. At each layer, compute the **local gradients** of the layer's computations
2. These local gradients will be used as inputs to the **next layer's** local gradient computations
3. At the end, we have a partial derivative for each of the parameters, which we can use to take a **gradient step**

Summary

- The loss function of a **deep feedforward networks** is simply a very nested function of the **parameters** of the model
- **Automatic differentiation** can compute these gradients more efficiently than symbolic differentiation or finite-differences numeric computations
    - Symbolic differentiation is **interleaved** with numeric computation
    - In **forward mode**, m sweeps are required for a function of m parameters
    - In **backward mode**, only a single sweep is required
- **Back-propagation** is simply automatic differentiation **applied to neural networks** in backward mode