Deep Learning and Applications

DSA 5204 • Lecture 3

Dr Low Yi Rui (Aaron)
Department of Mathematics

NUS
National University of Singapore

Project Groups and Homework

Please form your project groups by 31 Jan.

Please get started on the homework 2 with your group early, starting Monday. Instructions are on Luminus

Reminder: Homework 1 is due on 04 Feb.

Last Time

- Linear basis models vs adaptive basis models

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}) \quad \text{vs} \quad f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}; \boldsymbol{\theta})$$

- One-layer neural network is an example of the latter

$$f(\boldsymbol{x}) = f(\boldsymbol{x}; W, c, \boldsymbol{w}, \boldsymbol{b}) = \boldsymbol{w}^T g(W\boldsymbol{x} + \boldsymbol{b}) + c$$
$$= \begin{pmatrix} \boldsymbol{w} \\ c \end{pmatrix}^T \underbrace{\begin{pmatrix} g(W\boldsymbol{x} + \boldsymbol{b}) \\ 1 \end{pmatrix}}_{\boldsymbol{\phi}(\boldsymbol{x};\boldsymbol{\theta})}$$

  g is an activation function, e.g. ReLU, Sigmoid, etc…

- Advantage of adaptive basis: can adapt to data
- Disadvantage: harder to learn, no OLS formula

Last Time

- To learn a function f^*, we optimize the parameters $(\boldsymbol{w}, \boldsymbol{b}, W, c)$ so that

$$f^*(\boldsymbol{x}) \approx f(\boldsymbol{x}; \boldsymbol{w}, \boldsymbol{b}, W, c)$$

- This amounts to solving an optimization problem

$$\min_{\boldsymbol{\theta}} R(\boldsymbol{\theta})$$

- We can use **gradient descent** to solve this problem

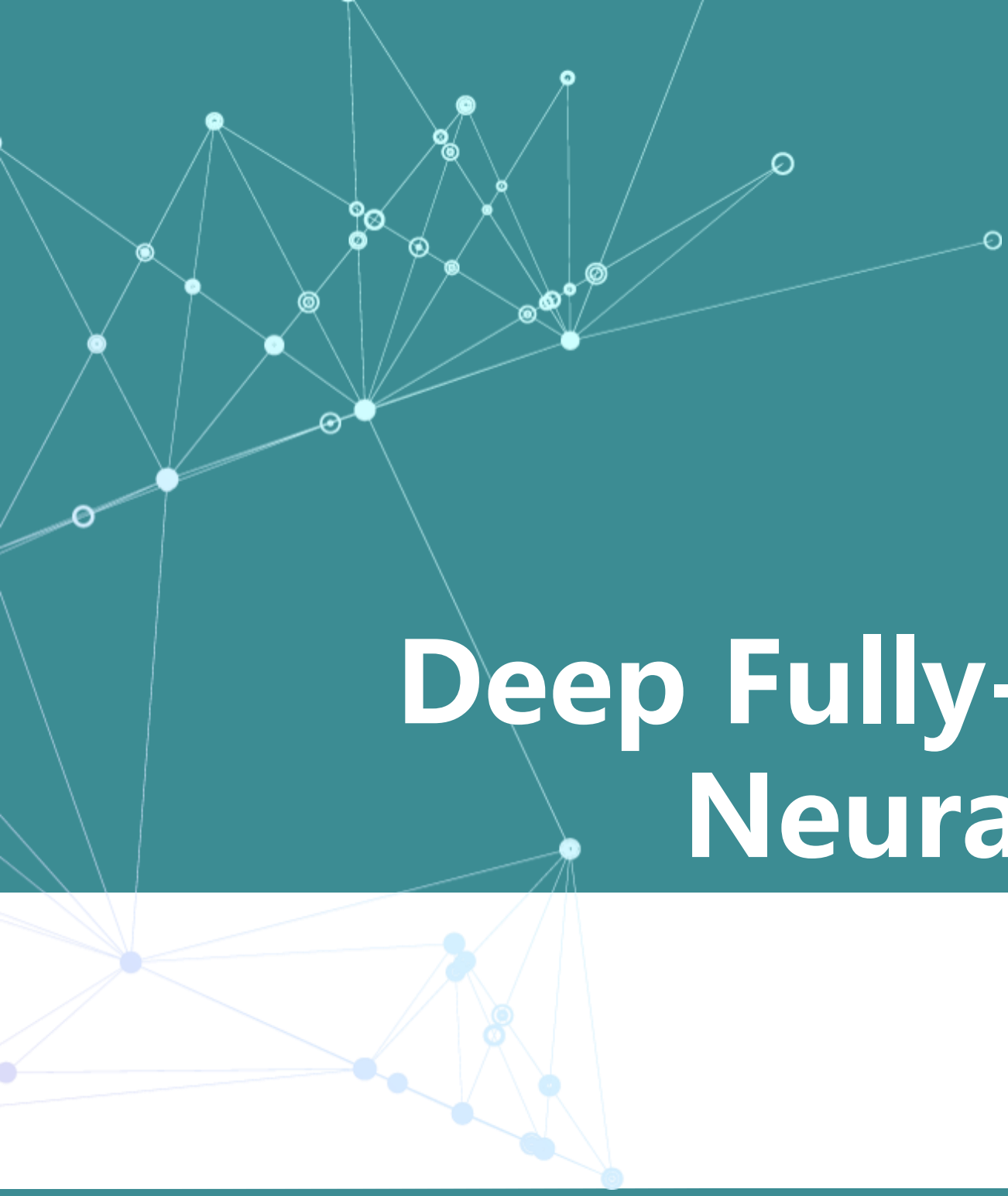
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \nabla R(\boldsymbol{\theta}_k)$$

- However, there are some issues
  - Gradient evaluation is expensive
  - How do we compute derivatives (i.e. how does Tensorflow work?)

In this class

We will develop these ideas in two directions

- We introduce the deep fully connected neural network (simplest DNN architecture)
- We introduce the extension of GD for large scale problems (such as DNNs)



Deep Fully-Connected Neural Networks

From Shallow to Deep Networks

Shallow (one-hidden-layer) neural network

$$\boldsymbol{h} = g(W\boldsymbol{x} + \boldsymbol{b})$$
$$\hat{y} = \boldsymbol{w}^T\boldsymbol{h} + c$$

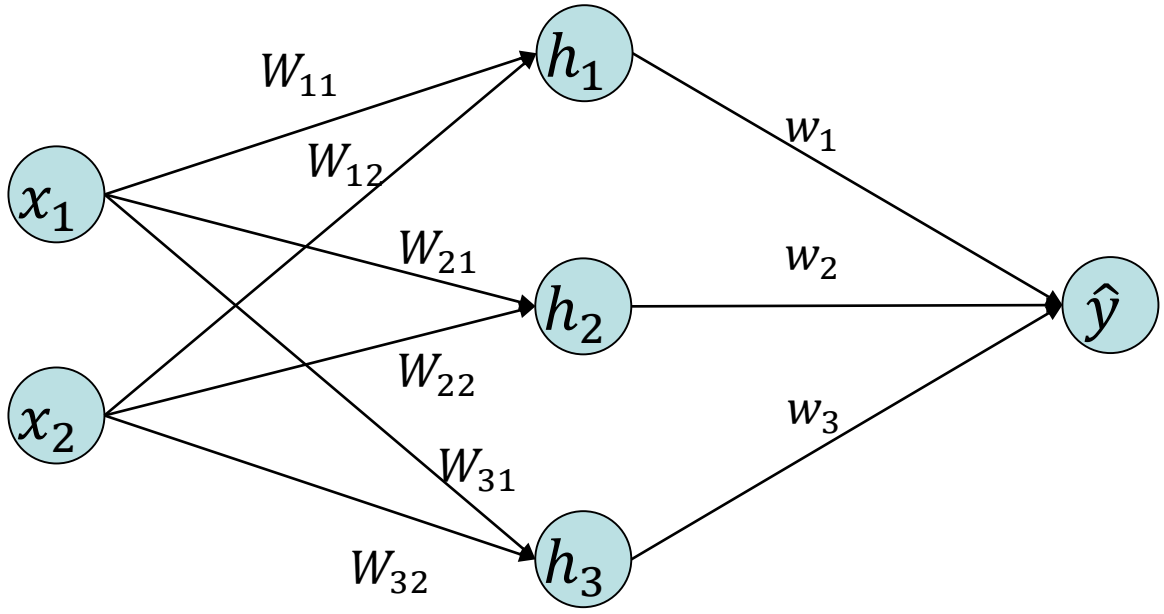
Deep networks are obtained by repeating the first step

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$
$$\boldsymbol{h}^{(2)} = g^{(2)}\left(W^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$
$$\boldsymbol{h}^{(3)} = g^{(3)}\left(W^{(3)}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}\right)$$
$$\vdots$$
$$\boldsymbol{h}^{(\ell)} = g^{(\ell)}\left(W^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}\right)$$
$$\hat{y} = \boldsymbol{w}^T\boldsymbol{h}^{(\ell)} + c$$

Graphical Representation (Shallow)

Ignoring bias:

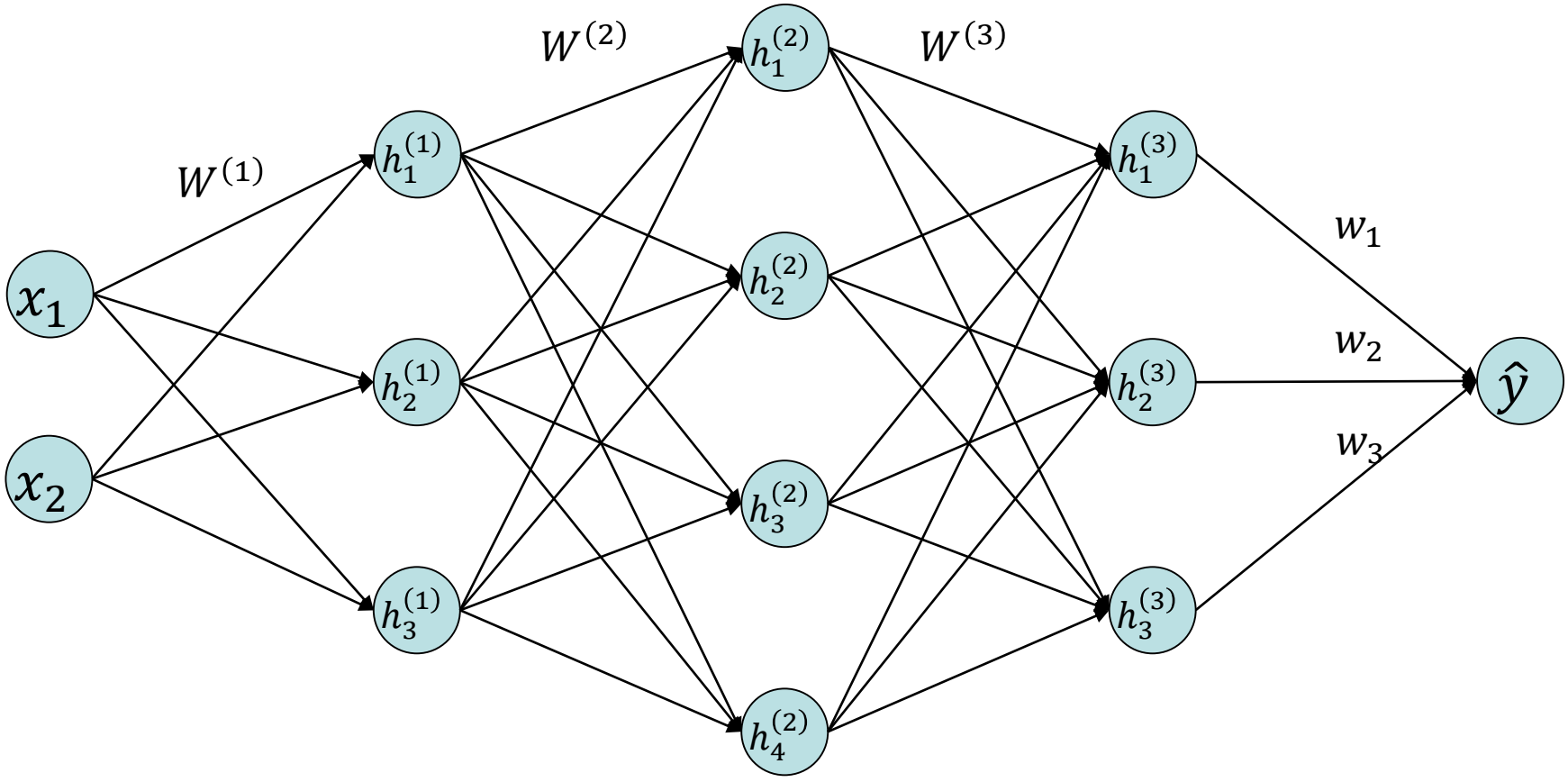
$$\boldsymbol{h} = g(W\boldsymbol{x})$$
$$\hat{y} = \boldsymbol{w}^T\boldsymbol{h}$$



Graphical Representation (Deep)

Ignoring bias again:

$$\boldsymbol{h}^{(i)} = g\left(W^{(i)}\boldsymbol{h}^{(i-1)}\right) \quad \boldsymbol{h}^{(0)} = \boldsymbol{x} \quad \hat{y} = \boldsymbol{w}^T\boldsymbol{h}^{(\ell)}$$



Why deep NNs?

We do not have a complete understanding of why deep NNs are good…

Several possible explanations

- Better approximation properties for certain functions, e.g. highly oscillatory ones
- Better generalization properties, given the right initialization, training or regularization strategies
- Sequential feature extraction is a good “prior” for some problems

But, deep is not always better!

Issues with deep neural networks

- Optimization problem becomes harder, due to high non-convexity of the problem
- Prone to overfitting if inappropriately initialized or trained
- More computational resources needed

Demo: Shallow vs Deep Networks

Stochastic Gradient Descent

The Structure of the Empirical Risk

The empirical risk has a specific form

$$R(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} R_i(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} L\left(y^{(i)}, f\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right)\right)$$

Example:

- MSE for regression:

$$R(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} \left(y^{(i)} - \boldsymbol{w}^T \boldsymbol{\phi}\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right)\right)^2$$

Computational Complexity of Gradient Descent

Consider applying GD to minimize

$$R(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N} R_i(\boldsymbol{\theta})$$

GD updates:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \frac{1}{N}\sum_{i=1}^{N} \nabla R_i(\boldsymbol{\theta})$$

The cost of evaluation of the gradient per iteration is $\mathcal{O}(N)$, typical data size: $N \sim 10^4 - 10^6$!

We want an algorithm that is $\mathcal{O}(1)$ with respect to data size

The Stochastic Gradient Algorithm (Robbins & Monro, 1951)

Idea: we replace the actual gradient with an **unbiased estimate**

Gradient Descent (GD):

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \frac{1}{N} \sum_{i=1}^{N} \nabla R_i(\boldsymbol{\theta}_k)$$

Stochastic Gradient Descent (SGD):

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \nabla R_{\gamma_k}(\boldsymbol{\theta}_k)$$

$$\gamma_k \sim \text{Uniform}\{1,2,\ldots,N\}$$

Why unbiased estimate?

$$\mathbb{E}(\boldsymbol{\theta}_{k+1}|\boldsymbol{\theta}_k) = \boldsymbol{\theta}_k - \epsilon \frac{1}{N} \sum_{i=1}^{N} \nabla R_i(\boldsymbol{\theta}_k)$$

$\mathbb{E}(\nabla R_{\gamma_k}(\theta_k))$

Robbins, Herbert, and Sutton Monro. "A Stochastic Approximation Method." The Annals of Mathematical Statistics, 1951, 400–407.

Mini-batch Stochastic Gradient Descent

Instead of just taking one random sample ∇R_{γ_i}, we can also take a few samples per iteration.

This is known as **mini-batch SGD**

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \frac{1}{M} \sum_{j \in B_k} \nabla R_j(\boldsymbol{\theta}_k)$$

B_k is a randomly chosen subset of $\{1,2, \dots, N\}$ of size M

Computational complexity:

$$\text{GD} = \mathcal{O}(N) \quad \text{SGD} = \mathcal{O}(1) \quad \text{Mini} - \text{batch SGD} = \mathcal{O}(M)$$

Disadvantages of SGD?

We know from the previous class:

$$\epsilon \text{ small enough} \Rightarrow \text{GD converges to a stationary point}$$

If $R(\boldsymbol{\theta})$ is convex*, then the only stationary points are global minima, hence GD converges.

What about SGD?

Example

Recall that we discussed the GD dynamics for

$$R(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{\theta}\|^2$$

Which gives

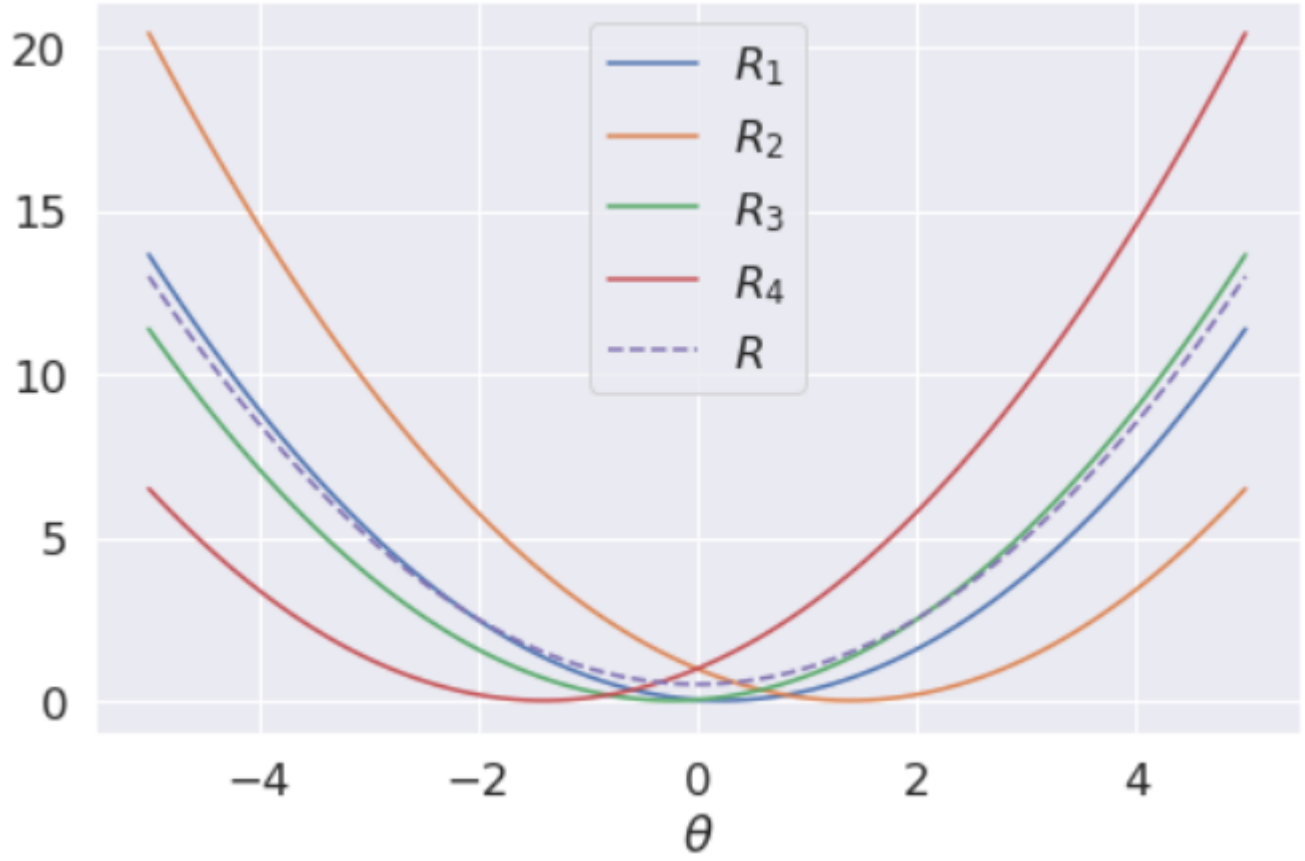
$$\boldsymbol{\theta}_k = (1-\epsilon)^k \boldsymbol{\theta}_0 \to 0 \quad \text{if} \quad 0 < \epsilon < 2$$

Consider a 1D variant of this example

$$R(\theta) = \frac{1}{N}\sum_{i=1}^{N} R_i(\theta) \qquad R_i(\theta) = \frac{1}{2}\left(\theta - \theta^{(i)}\right)^2$$

Where $\frac{1}{N}\sum_i \theta^{(i)} = 0$ and $\frac{1}{N}\sum_i \left(\theta^{(i)}\right)^2 = 1$

Total loss and sample losses



Gradient for sample i: $\nabla R_i(\theta) = \theta - \theta^{(i)}$

SGD iterations:

$$\theta_{k+1} = \theta_k - \epsilon \nabla R_{\gamma_k}(\theta_k) = (1-\epsilon)\theta_k + \epsilon \theta^{(\gamma_k)}$$

$$\gamma_k \sim \text{Uniform}\{1,2,\dots,N\} \text{ (i. i. d.)}$$

Solving, we get

$$\theta_k = \underbrace{(1-\epsilon)^k \theta_0}_{\text{deterministic}} + \underbrace{\epsilon \sum_{j=1}^{k} (1-\epsilon)^{j-1} \theta^{(\gamma_{k-j})}}_{\text{random}}$$

On average, we have the same dynamics as GD

$$\mathbb{E}\theta_k = (1-\epsilon)^k \theta_0$$

But, we should also look at the variance!

We have

$$\mathbb{E}(\theta_k)^2$$

$$= (1-\epsilon)^{2k}\theta_0^2 + \epsilon(1-\epsilon)^k\theta_0 \sum_{j=1}^{k}(1-\epsilon)^{j-1}\,\mathbb{E}\theta^{(\gamma_{k-j})}$$

$$+ \epsilon^2 \sum_{j,l=1}^{k}(1-\epsilon)^{j-1}(1-\epsilon)^{l-1}\mathbb{E}\left[\theta^{(\gamma_{k-j})}\theta^{(\gamma_{k-l})}\right]$$

Using the i.i.d. assumption, we know that

$$\mathbb{E}\theta^{(\gamma_{k-j})} = 0 \text{ and } \mathbb{E}\left[\theta^{(\gamma_{k-j})}\theta^{(\gamma_{k-l})}\right] = \delta_{jl}$$

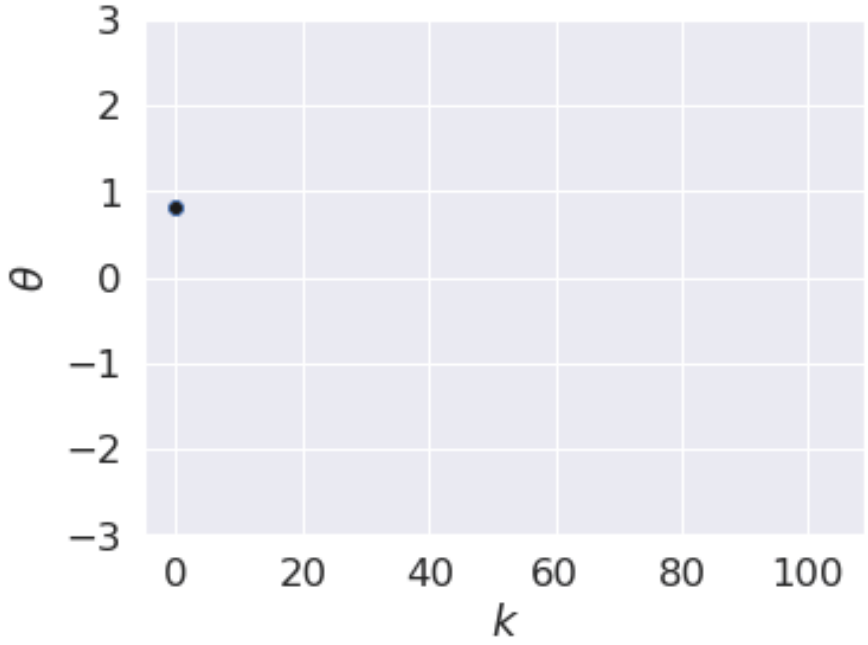
This gives

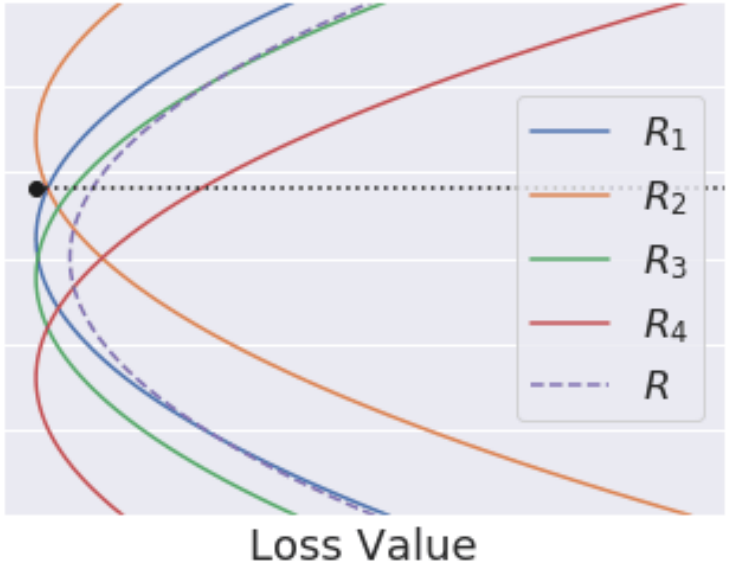
$$\mathbb{E}(\theta_k)^2 = (1-\epsilon)^{2k}\theta_0^2 + \frac{\epsilon}{2-\epsilon}[1-(1-\epsilon)^{2k}]$$

As $k \to \infty$, we have

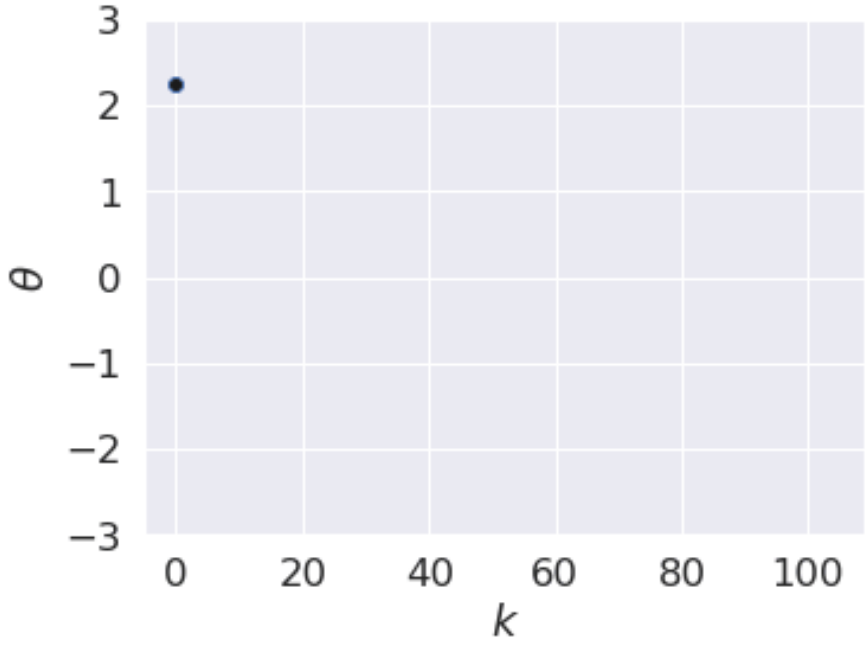
$$\mathbb{E}\theta_k \to 0 \quad \text{but} \quad \mathbb{E}(\theta_k)^2 \to \frac{\epsilon}{2-\epsilon}$$

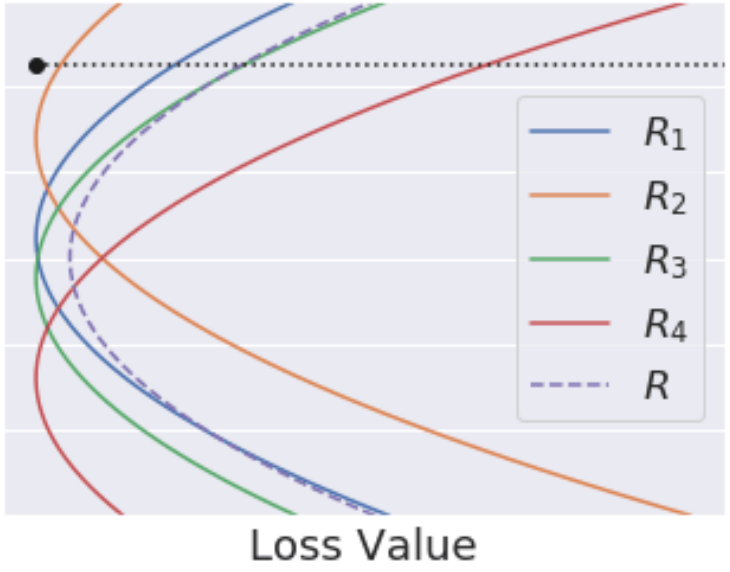
SGD Dynamics ($\epsilon = 0.5$)





SGD Dynamics ($\epsilon = 0.1$)





SGD with Varying Learning Rate

For fixed learning rate, SGD may not converge, due to fluctuations (Is this always the case?)

Idea: **decay** the learning rate.

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon_k \nabla R_{\gamma_k}(\boldsymbol{\theta}_k)$$

It turns out that if ϵ_k decays not too fast and not too slow, then we indeed have convergence. A sufficient condition is

$$\sum_{k=0}^{\infty} \epsilon_k = \infty \qquad \sum_{k=0}^{\infty} \epsilon_k^2 < \infty$$

e.g. $\epsilon_k = \frac{1}{k+1}$

Kushner, Harold, and G. George Yin. Stochastic Approximation and Recursive Algorithms and Applications. Vol. 35. Springer 2003.

Variants of SGD

There are many variants of SGD used for large-scale machine learning

- Momentum
- Adaptive learning rates (Adagrad, Adadelta)
- Combination of the above (Adam, RMSprop)
- Variance reduction (SVRG)

Useful survey:

Bottou, Léon, Frank E. Curtis, and Jorge Nocedal. "Optimization Methods for Large-Scale Machine Learning." *SIAM Review*, 2018. https://doi.org/10.1137/16m1080173.

Momentum (Polyak, 1964)

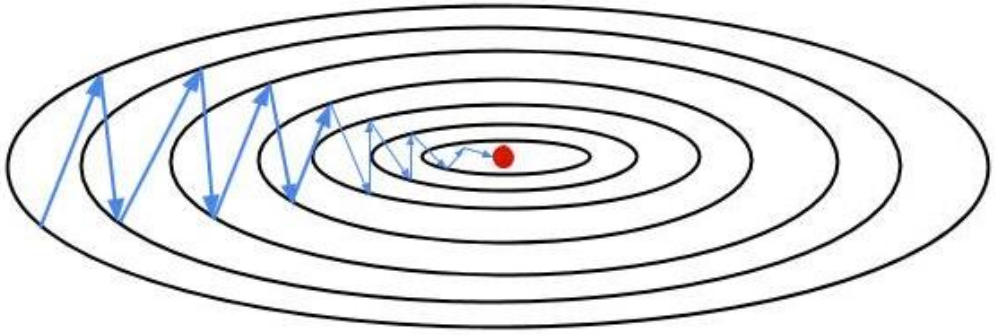
Momentum is a popular way to speed up GD/SGD.

Idea: imagine going down a valley which is wide in one direction and narrow in the other

$$R(\boldsymbol{\theta}) = \frac{1}{2}\theta_1^2 + \frac{\lambda}{2}\theta_2^2, \qquad \lambda \gg 1$$

Negative gradient direction

$$\nabla R(\boldsymbol{\theta}) = -\begin{pmatrix} \theta_1 \\ \lambda\theta_2 \end{pmatrix}$$



Polyak, Boris T. "Some Methods of Speeding up the Convergence of Iteration Methods." USSR Computational Mathematics and Mathematical Physics 4, no. 5 (1964): 1–17.

Idea: we want to remember a little about past descent directions, so as to reduce the zig-zag behavior

GD with momentum

$$\boldsymbol{v}_{k+1} = \alpha \boldsymbol{v}_k - \epsilon \nabla R(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \boldsymbol{v}_{k+1}$$

The vector $\boldsymbol{v}$ is known as **momentum**, and $\alpha \in (0,1)$ is the **momentum parameter**.

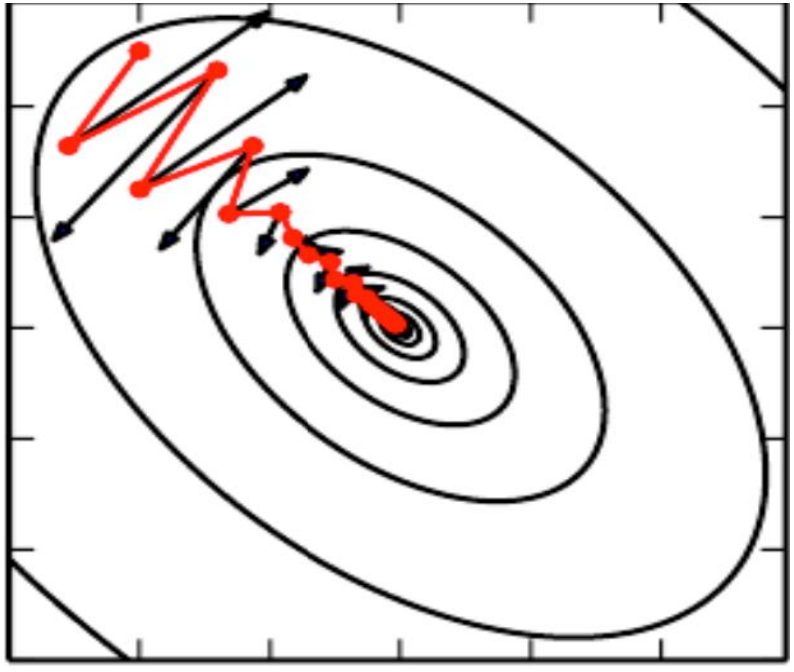
- When $\alpha = 0$, this is just GD
- When $\alpha > 0$, updates to $\boldsymbol{\theta}_k$ remembers previous directions

Inspiration from Physics

Recall Newton's second law

$$\text{rate of change of momentum} = \text{external force}$$

Ball rolling down hill



Black arrows = force

Red path = actual path followed

The bigger the mass, the less the velocity changes with external force.

Assume mass is constant, so momentum is proportional to velocity

$$\boldsymbol{p} = m\boldsymbol{v}$$

Then, the second law is $F = m\frac{d\boldsymbol{v}}{dt}$.

Assume that $F = F_{\text{ext}}(\boldsymbol{\theta}) + F_{\text{friction}}(\boldsymbol{v})$ with friction proportional to $\boldsymbol{v}$, then

$$\frac{d\boldsymbol{v}}{dt} = -\frac{\gamma}{m}\boldsymbol{v} + \frac{1}{m}F_{ext}(\boldsymbol{\theta})$$
$$\frac{d\boldsymbol{\theta}}{dt} = \boldsymbol{v}$$

$\longleftrightarrow$

$$\boldsymbol{v}_{k+1} = \alpha\boldsymbol{v}_k - \epsilon\nabla R(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \boldsymbol{v}_{k+1}$$

Exercise

Derive (and/or code up) the GD and GD with momentum updates for

$$R(\boldsymbol{\theta}) = \frac{1}{2}\theta_1^2 + \frac{\lambda}{2}\theta_2^2, \qquad \lambda \geq 1$$

How does the convergence rate depend on λ in each case? (Hint: it has something to do with the condition number)

SGD in Practice

In practice, SGD is almost never used with a single batch. We typically use a mini-batch version.

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \frac{1}{M} \sum_{j \in B_k} \nabla R_j(\boldsymbol{\theta}_k)$$

B_k is a subset of $\{1,2,\dots,N\}$ of size M chosen at random

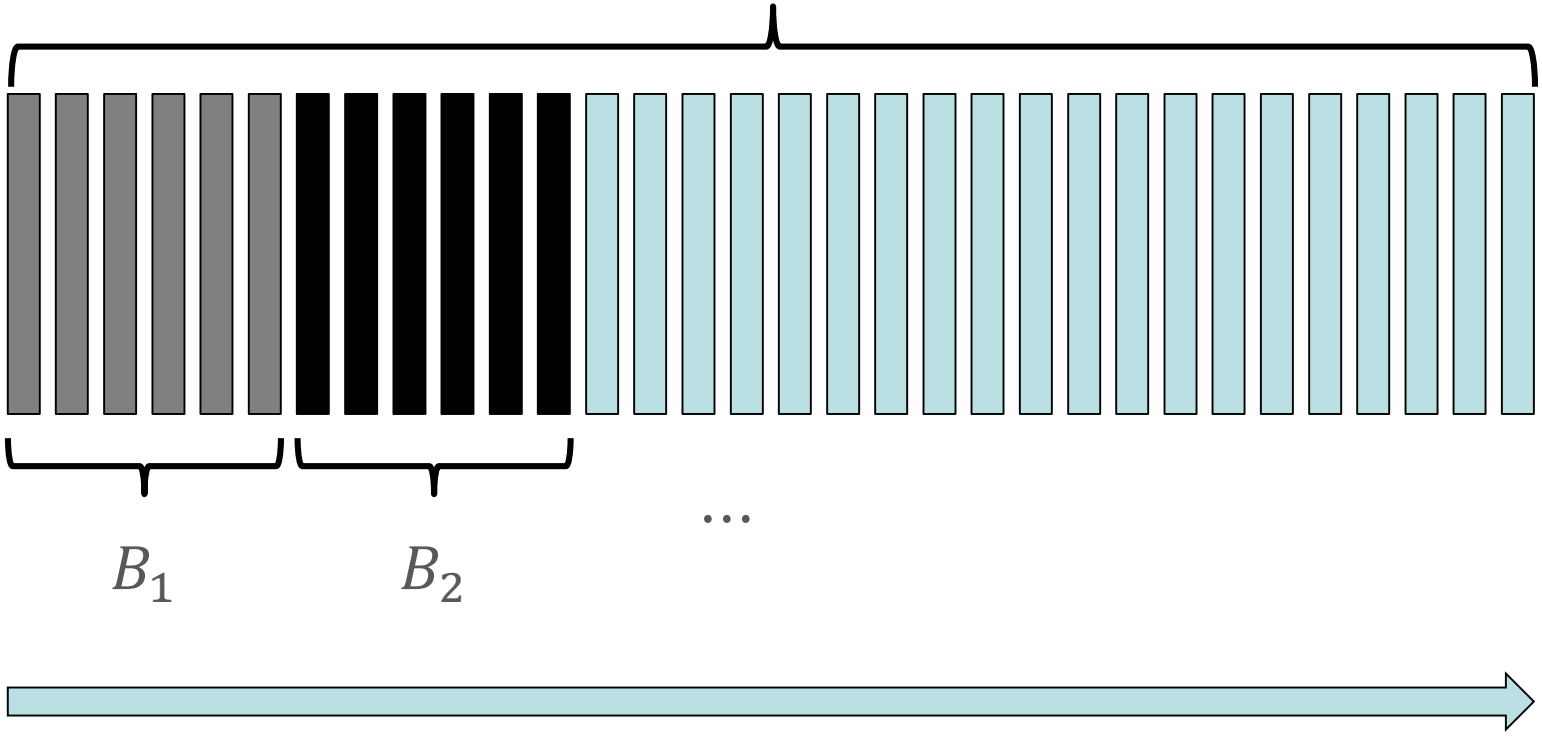
A further modification is typical:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \epsilon \frac{1}{M} \sum_{j \in B_k} \nabla R_j(\boldsymbol{\theta}_k)$$

B_k loops over $\{1,2,\dots,N\}$ in chunks of size M

Every loop over $\{1,2,\dots,N\}$ is called an **epoch**.

$$R = \frac{1}{N}\sum_{j} R_j$$



The Backpropagation Algorithm

Computing Gradients

In both GD and SGD (and their variants), it is necessary to compute the gradients

$$\nabla R(\boldsymbol{\theta}) \quad \text{or} \quad \nabla R_i(\boldsymbol{\theta})$$

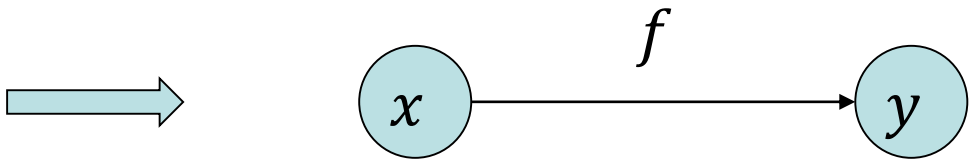
How do we compute gradients efficiently for DNNs?

Computational Graph

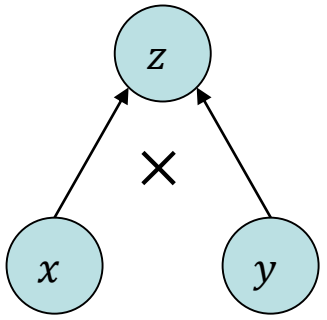
A computational graph is a graph whose

- **Nodes** represent **variables**
- **Directed edges** represent **operations**

$$y = f(x)$$

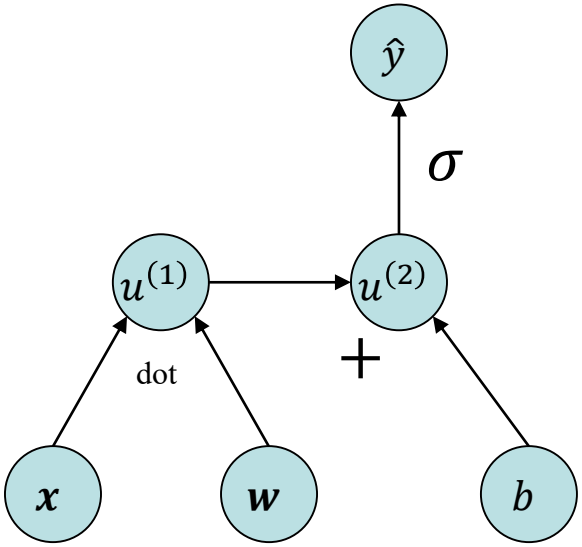


Some Examples of Computational Graphs



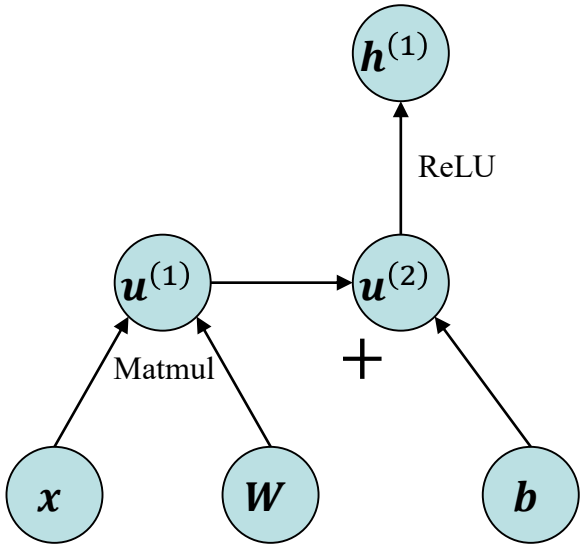
$$z = xy$$

Some Examples of Computational Graphs



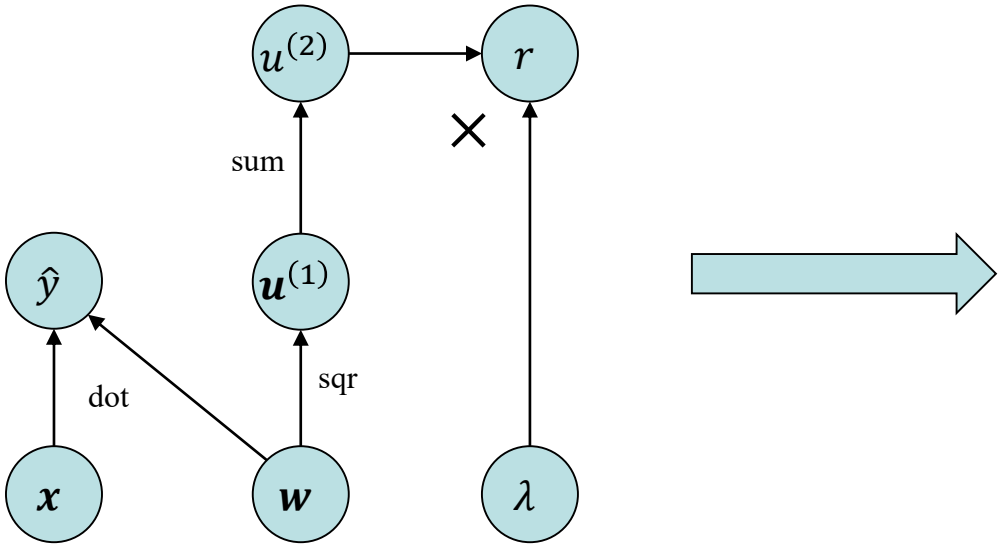
$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$$

Some Examples of Computational Graphs



$$h^{(1)} = \mathrm{ReLU}(Wx + b)$$

Some Examples of Computational Graphs



$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x}$$
$$r = \lambda \|\boldsymbol{w}\|^2$$

Chain Rule for Scalar Functions

Let x be a real number and f, g be scalar functions

Let

$$y = g(x) \quad \text{and} \quad z = f(y) = f\big(g(x)\big)$$

Then, we know

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \quad \text{and so} \quad \frac{dz}{dx}(x) = f'\big(g(x)\big)g'(x)$$

Chain Rule for Vector-Valued Functions

The general chain rule is similar. Let

$$f: \mathbb{R}^m \to \mathbb{R}^n \quad \text{and} \quad g: \mathbb{R}^n \to \mathbb{R}$$
$$\boldsymbol{y} = f(\boldsymbol{x}) \quad \text{and} \quad z = g(\boldsymbol{y})$$

Then,

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Vector notation:

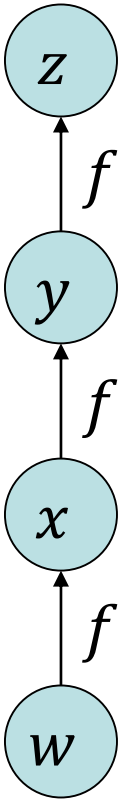
$$\nabla_x z = \underbrace{\left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T}_{\text{Jacobian of } f} \nabla_{\boldsymbol{y}} z$$

Why do we need an algorithm?

Chain rule already tell us how to compute gradients.
Why is there a need for an algorithm?

We want an **efficient** algorithm that, where possible, does not repeat computations unnecessarily!

Example: Naïve chain rule



Consider a simple computational graph on the left

We have

$$z = f(y) = f\big(f(x)\big) = f\Big(f\big(f(w)\big)\Big)$$

Using chain rule, we have

$$\frac{dz}{dw} = f'(y)f'(x)f'(w)$$
$$= f'\Big(f\big(f(w)\big)\Big)f'\big(f(w)\big)f'(w)$$

Here, it is more efficient to compute $x = f(w)$ and store it, rather than to compute it twice.

Example: Simplest backprop

Consider a 1D linear neural network with one data point

$$h^{(1)} = w^{(1)}x$$
$$h^{(2)} = w^{(2)}h^{(1)}$$
$$h^{(3)} = w^{(3)}h^{(2)}$$
$$\hat{y} = wh^{(3)}$$

Note: $x, w^{(j)}, h^{(j)}, \hat{y}$ are all scalars and the label corresponding to input x is y

Empirical risk is

$$R(\boldsymbol{\theta}) = R\left(w^{(1)}, w^{(2)}, w^{(3)}, w\right) = L(\hat{y}(x; \boldsymbol{\theta}), y)$$

Goal: compute $\frac{\partial R}{\partial w^{(j)}}$ for $j = 1,2,3$ and $\frac{\partial R}{\partial w}$ **efficiently**

Step 1: Forward propagation to compute hidden states

Given

- input x and label y
- current weights $w^{(1)}$, $w^{(2)}$, $w^{(3)}$, w

Compute in forward fashion

$$h^{(1)} = w^{(1)}x$$
$$h^{(2)} = w^{(2)}h^{(1)}$$
$$h^{(3)} = w^{(3)}h^{(2)}$$
$$\hat{y} = wh^{(3)}$$

Store $h^{(1)}, h^{(2)}, h^{(3)}, \hat{y}$

Step 2: Backward propagation of derivatives wrt hidden states

Last layer:

$$\frac{dR}{d\hat{y}} = \frac{\partial}{\partial \hat{y}} L(y, \hat{y}) \rightarrow \hat{p}$$

Next layers:

1. $\frac{dR}{dh^{(3)}} = \frac{dR}{d\hat{y}} \frac{d\hat{y}}{dh^{(3)}} = \hat{p}w \rightarrow p^{(3)}$
2. $\frac{dR}{dh^{(2)}} = \frac{dR}{dh^{(3)}} \frac{dh^{(3)}}{dh^{(2)}} = p^{(3)} w^{(3)} \rightarrow p^{(2)}$
3. $\frac{dR}{dh^{(1)}} = \frac{dR}{dh^{(2)}} \frac{dh^{(2)}}{dh^{(1)}} = p^{(2)} w^{(2)} \rightarrow p^{(1)}$

Step 3: Computing derivatives wrt parameters

By Chain Rule,

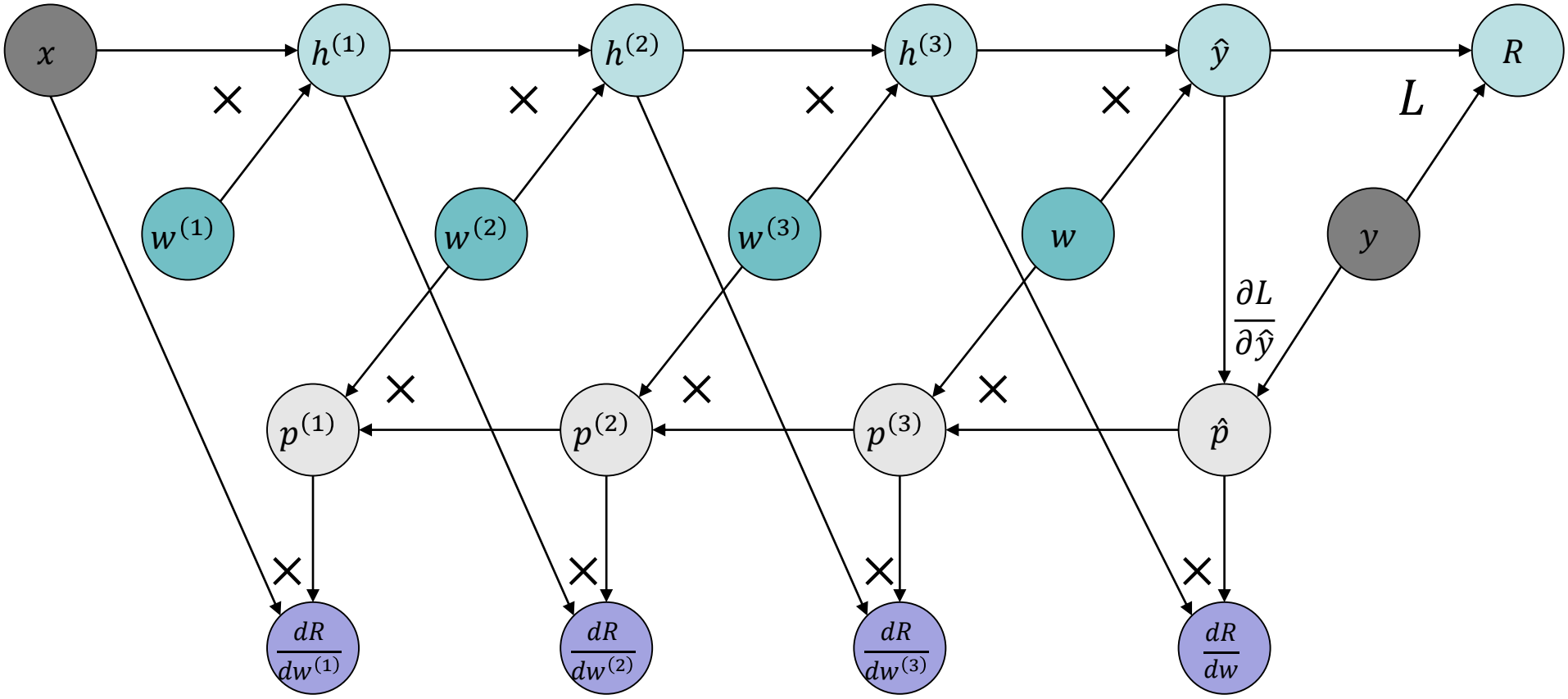
$$\frac{dR}{dw} = \frac{dR}{d\hat{y}}\frac{d\hat{y}}{dw} = \hat{p}h^{(3)}$$

And,

$$\frac{dR}{dw^{(j)}} = \frac{dR}{dh^{(j)}}\frac{dh^{(j)}}{dw^{(j)}} = p^{(j)}h^{(j-1)}$$

For $j = 3,2,1$, where $h^{(0)} := x$

Computational Graph Visualization of Back Propagation



Some notable properties

- Only two passes of the graph, one forward and one backward
- Total computational cost is $\mathcal{O}(\#\text{nodes})$. What about naïve chain rule application for all the derivatives?
- Only the forward states need to be stored

Generalization to Nonlinear Case

Previously we have

$$h^{(i)} = w^{(i)} h^{(i-1)}$$

Consider now instead

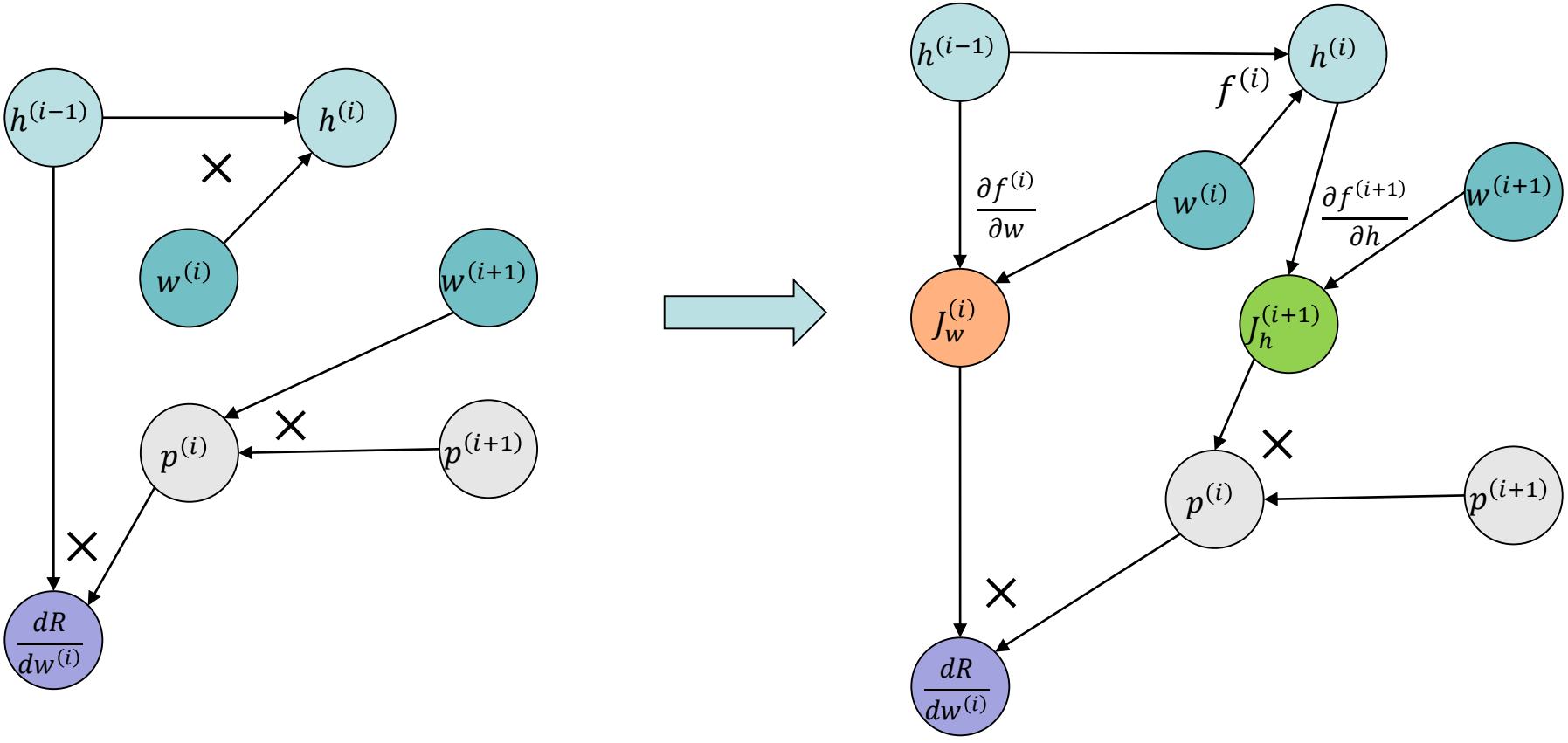
$$h^{(i)} = f^{(i)}\left(h^{(i-1)}, w^{(i)}\right)$$

Then we can do entirely as before, except:

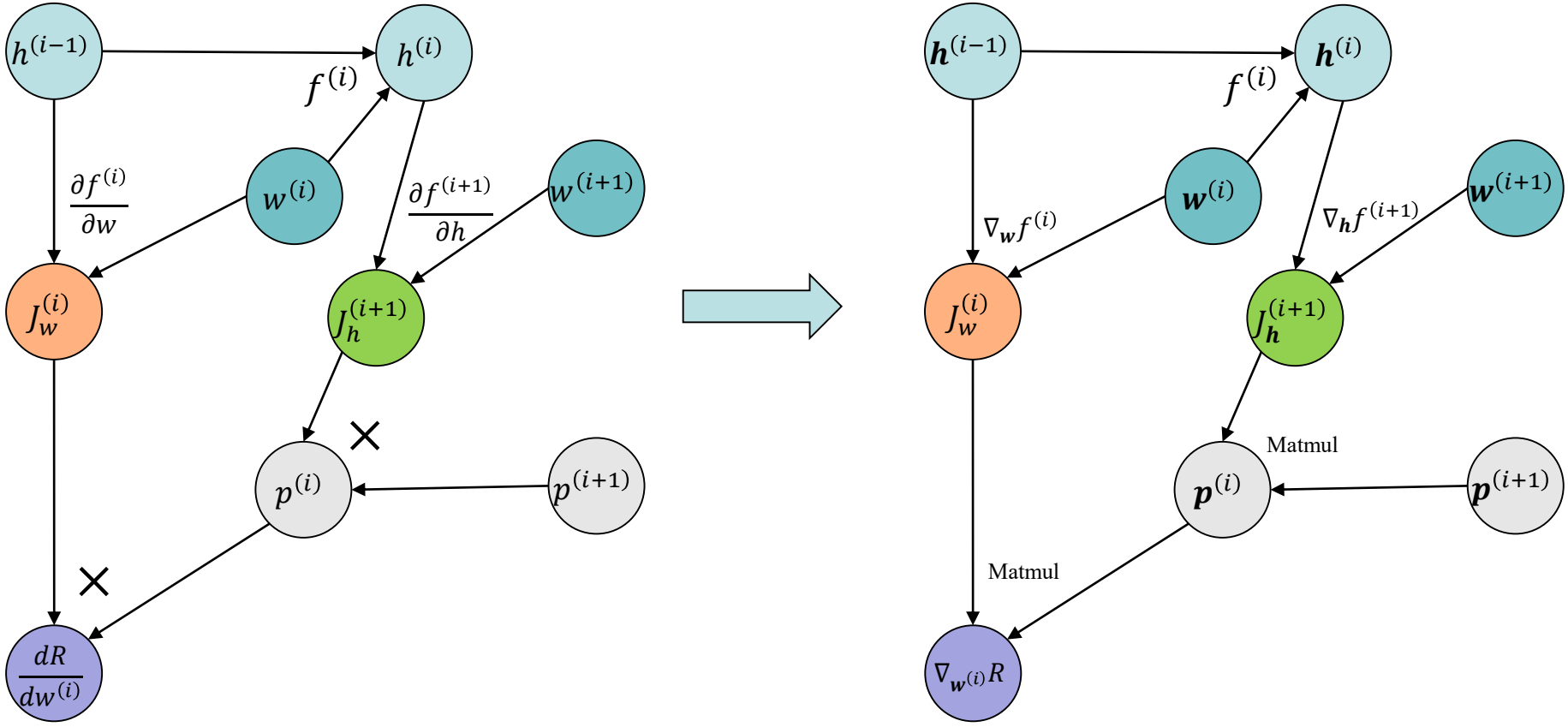
$$p^{(i)} = \frac{dR}{dh^{(i)}} = \frac{dR}{dh^{(i+1)}} \frac{dh^{(i+1)}}{dh(i)} = p^{(i+1)} \frac{\partial f^{(i+1)}}{\partial h}\left(h^{(i)}, w^{(i+1)}\right)$$

$$\frac{dR}{dw^{(i)}} = \frac{dR}{dh^{(i)}} \frac{dh^{(i)}}{dw^{(i)}} = p^{(i)} \frac{\partial f^{(i)}}{\partial w}\left(h^{(i-1)}, w^{(i)}\right)$$

Computational Graph for Nonlinear Case



Computational Graph for Multi-dimensional Case



Other Generalizations

- Tensor inputs and outputs
- Multiple inputs and outputs, or general acyclic directed graphs
- More than one sample (do backprop for each R_i then sum them)
- More computational/memory efficient implementations
- For all of these and more, see Chapter 6.5 of the deep learning textbook on automatic differentiation and back propagation.

Tensorflow (Or AD) in a Nutshell

For each operation op representing some function f, two methods are implemented

- op.forward(**x**): computes $f(\boldsymbol{x})$
- op.backward(**x**): computes $\nabla f(\boldsymbol{x})$

As long as both of these methods are available for all operations used in a computational graph, arbitrary derivatives can be computed via back-propagation.

Summary

In this lecture, we introduced

- Deep neural networks, which are compositions of shallow layers
- Stochastic gradient descent and its variants for handling large datasets and large models
- Backpropagation algorithm as an efficient means to calculate gradients