OUTLINE

OUTLINE

OUTLINE

Recap: the *model fitting* approach to Machine Learning

- $f_\theta =$ *a jean*
- $\theta =$ *its (width, length)*
- $\mathcal{F} = \{f_\theta\}_{\theta\in\Theta}$ *the shelves*

Recap: the *model fitting* approach to Machine Learning

- $f_\theta =$ *a jean*
- $\theta =$ *its (width, length)*
- $\mathcal{F} = \{f_\theta\}_{\theta\in\Theta}$ *the shelves*

- Given a **parameterized family** $\mathcal{F}$ of models == functions

Recap: the *model fitting* approach to Machine Learning

- $f_\theta =$ *a jean*
- $\theta =$ *its (width, length)*
- $\mathcal{F} = \{f_\theta\}_{\theta\in\Theta}$ *the shelves*

•Given a **parameterized family** $\mathcal{F}$ of models == functions

Ex: *a DNN with* $f_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^{N^{(\text{out})}}$

Recap: the *model fitting* approach to Machine Learning

- $f_\theta =$ *a jean*
- $\theta =$ *its (width, length)*
- $\mathcal{F} = \{f_\theta\}_{\theta\in\Theta}$ *the shelves*

•Given a **parameterized family** $\mathcal{F}$ of models == functions

Ex: *a DNN with* $f_\theta : \mathbb{R}^D \to \mathbb{R}^{N^{(\text{out})}}$

•Given a **training dataset** $\mathcal{T}$ *(your legs!),*

Recap: the *model fitting* approach to Machine Learning

- $f_\theta =$ *a jean*
- $\theta =$ *its (width, length)*
- $\mathcal{F} = \{f_\theta\}_{\theta \in \Theta}$ *the shelves*

•Given a **parameterized family** $\mathcal{F}$ of models == functions

Ex: *a DNN with* $f_\theta : \mathbb{R}^D \to \mathbb{R}^{N^{(\mathrm{out})}}$

•Given a **training dataset** $\mathcal{T}$ *(your legs!),*

•Given a ***total loss function*** $L(f_\theta, \mathcal{T})$ that measures the ***fit*** of a given model f_θ to the **full dataset**, for the given task (the smaller the better),

Recap: the *model fitting* approach to Machine Learning

- $f_\theta =$ *a jean*
- $\theta =$ *its (width, length)*
- $\mathcal{F} = \{f_\theta\}_{\theta \in \Theta}$ *the shelves*

•Given a **parameterized family** $\mathcal{F}$ of models == functions

Ex: *a DNN with* $f_\theta : \mathbb{R}^D \to \mathbb{R}^{N^{(\mathrm{out})}}$

•Given a **training dataset** $\mathcal{T}$ *(your legs!),*

•Given a ***total loss function*** $L(f_\theta, \mathcal{T})$ that measures the ***fit*** of a given model f_θ to the **full dataset**, for the given task (the smaller the better),

$\rightarrow$ We want to **minimize** the loss with respect to the **parameters** $\theta \in \Theta$ **:**

$$\hat{f} = f_{\hat{\theta}} \quad \text{where} \quad \hat{\theta} = \operatorname*{argmin}_{\theta \in \Theta} L(f_\theta, \mathcal{T})$$

Recap: the *model fitting* approach to Machine Learning

- $f_\theta =$ *a jean*
- $\theta =$ *its (width, length)*
- $\mathcal{F} = \{f_\theta\}_{\theta\in\Theta}$ *the shelves*

•Given a **parameterized family** $\mathcal{F}$ of models == functions

Ex: *a DNN with* $f_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^{N^{(\mathrm{out})}}$

•Given a **training dataset** $\mathcal{T}$ *(your legs!),*

•Given a ***total loss function*** $L(f_\theta, \mathcal{T})$ that measures the ***fit*** of a given model f_θ to the **full dataset**, for the given task (the smaller the better),

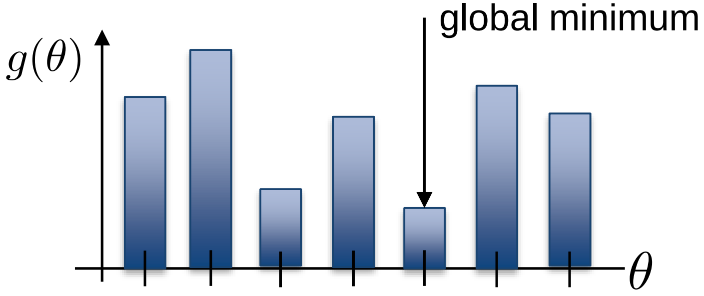
$\rightarrow$ We want to **minimize** the loss with respect to the **parameters** $\theta \in \Theta$ **:**

$$\hat{f} = f_{\hat{\theta}} \quad \text{where} \quad \hat{\theta} = \underset{\theta\in\Theta}{\operatorname{argmin}}\, L(f_\theta, \mathcal{T})$$

For conciseness we will use $g(\theta) \overset{\text{def}}{=} L(f_\theta, \mathcal{T}) \quad (g : \Theta \rightarrow \mathbb{R})$ in the next slides.

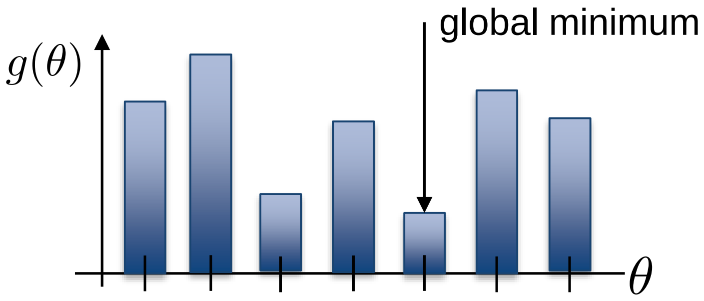
Domain of the function

Discrete: $\theta \in \{\theta_1, \dots, \theta_C\}$

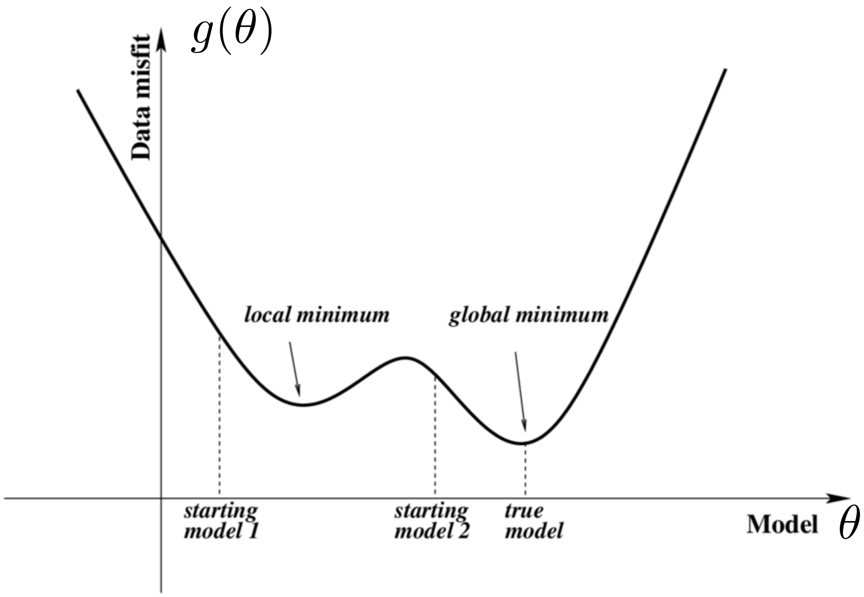


Domain of the function

Discrete: $\theta \in \{\theta_1, \ldots, \theta_C\}$

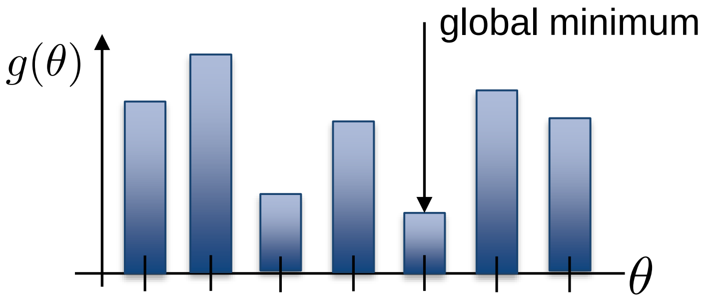


1D continuous: $\theta \in \mathbb{R}$

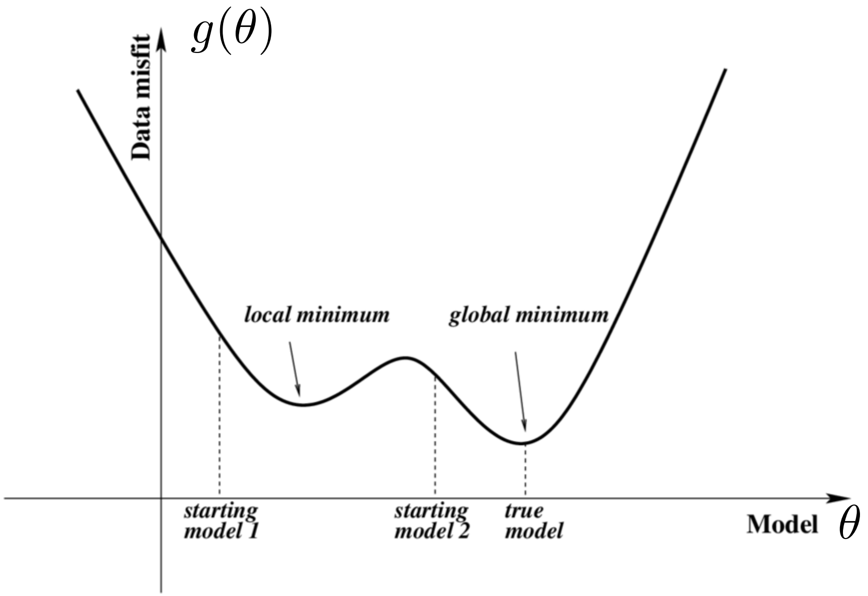


Domain of the function

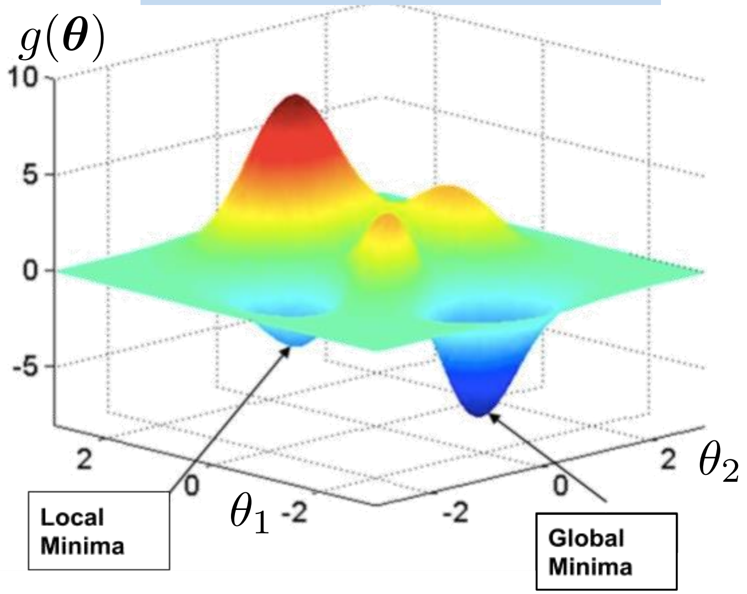
Discrete: $\theta \in \{\theta_1, \ldots, \theta_C\}$



1D continuous: $\theta \in \mathbb{R}$

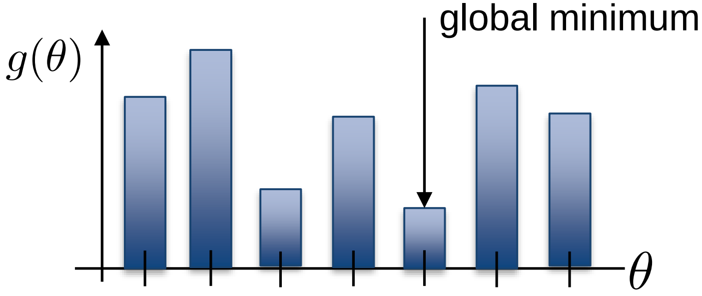


2D continuous: $\theta \in \mathbb{R}^2$

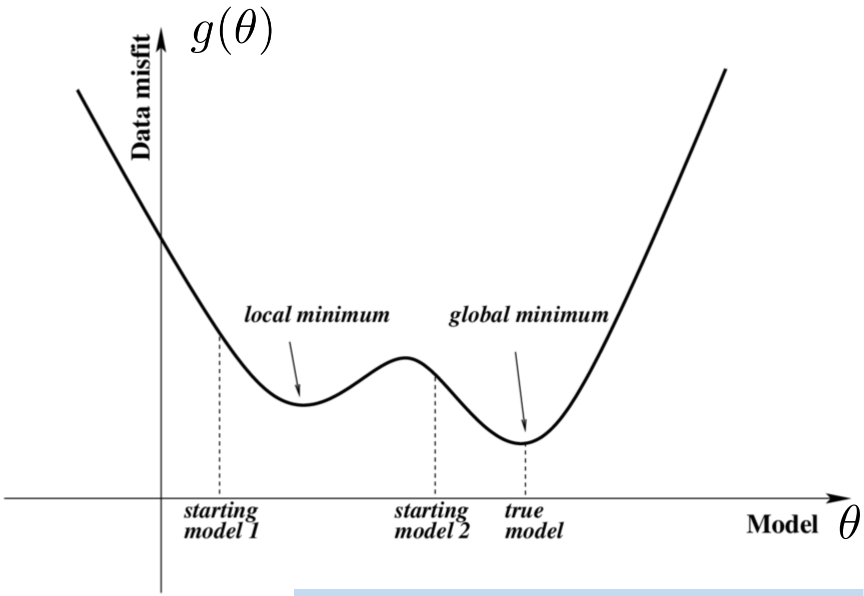


Domain of the function

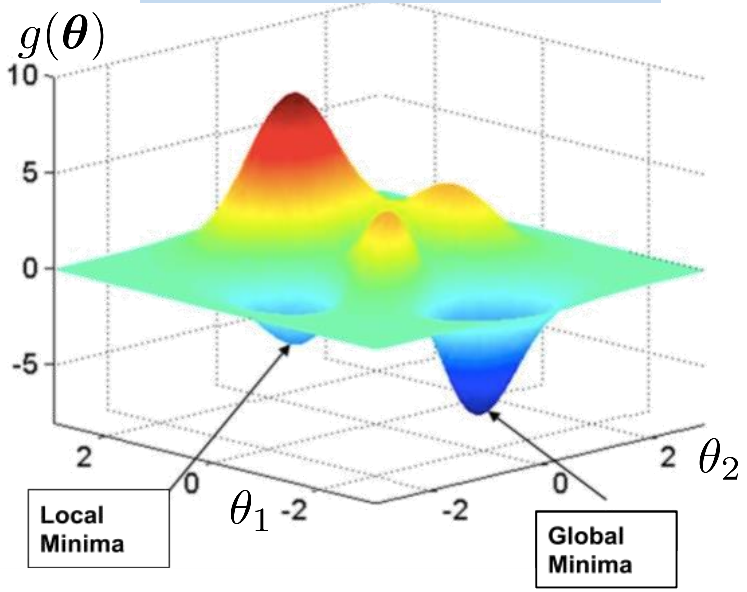
Discrete: $\theta \in \{\theta_1, \ldots, \theta_C\}$



1D continuous: $\theta \in \mathbb{R}$



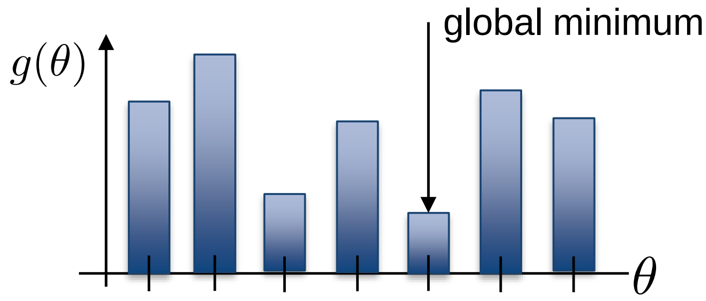
2D continuous: $\theta \in \mathbb{R}^2$



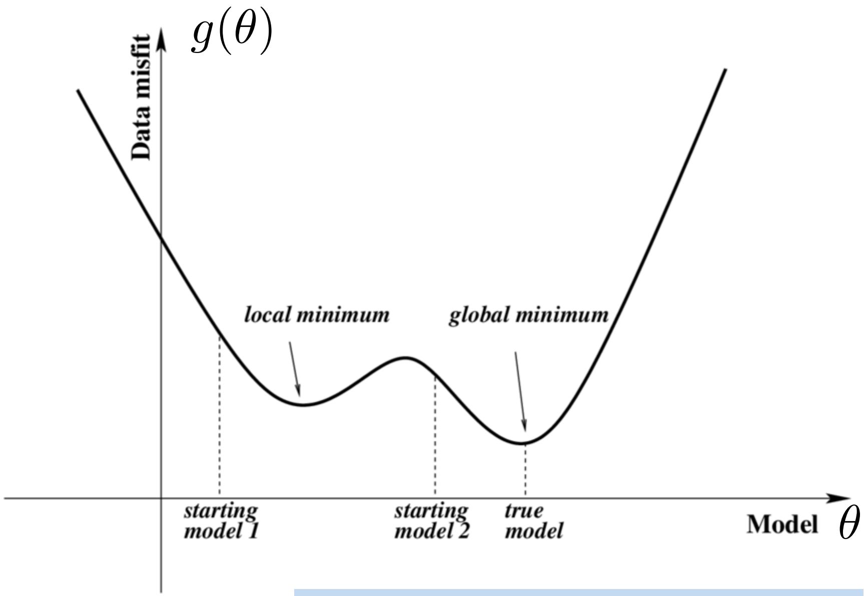
D-D continuous: $\theta \in \mathbb{R}^D$

Domain of the function

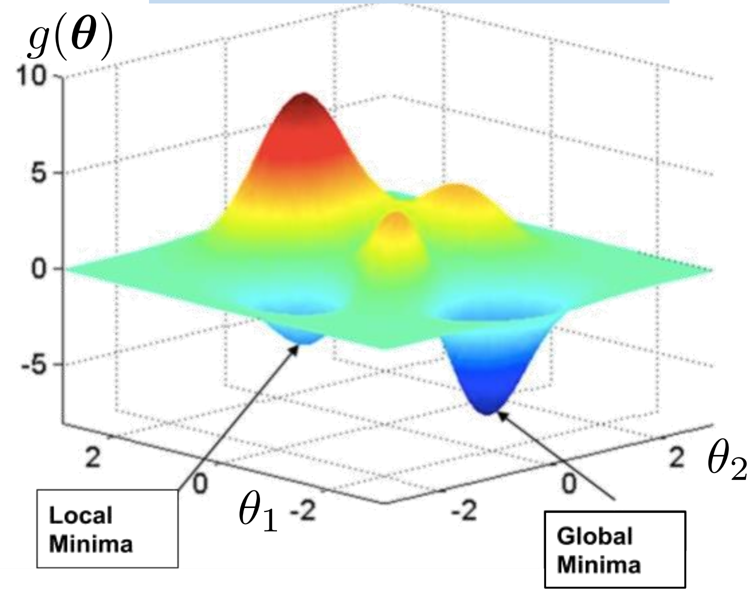
Discrete: $\theta \in \{\theta_1, \ldots, \theta_C\}$



1D continuous: $\theta \in \mathbb{R}$



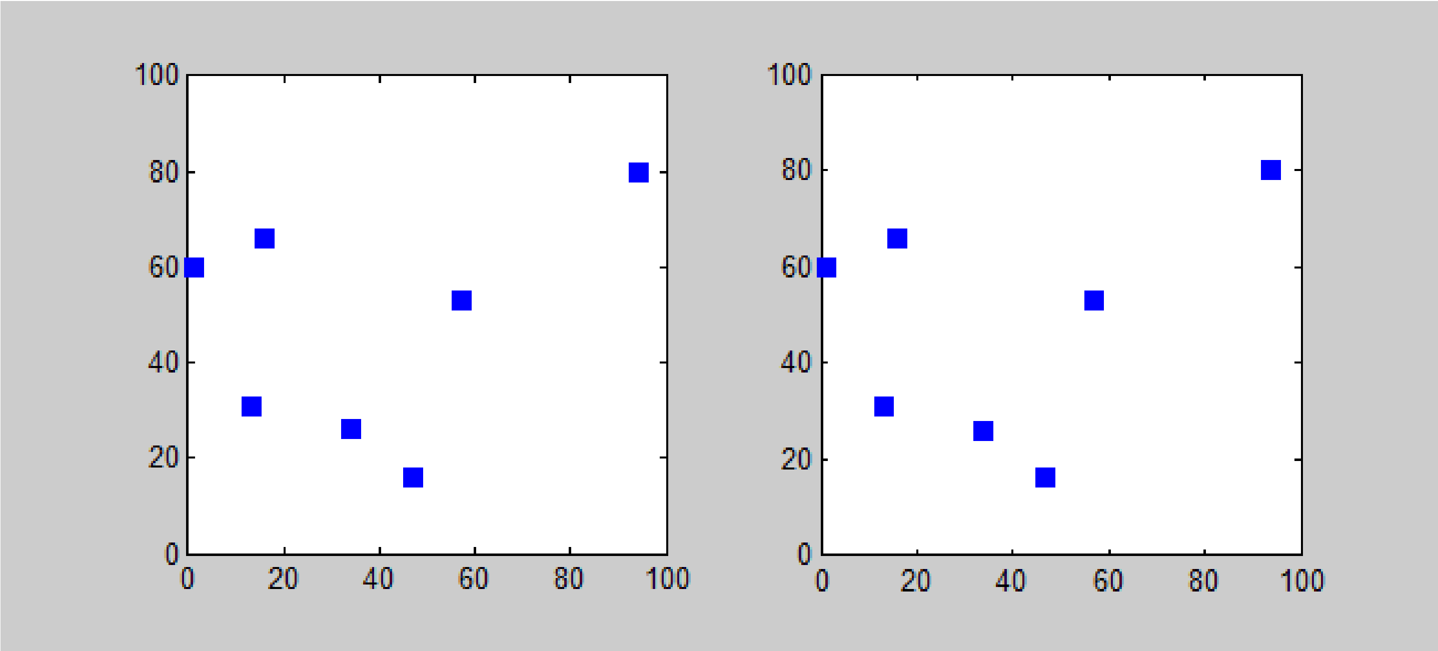
2D continuous: $\theta \in \mathbb{R}^2$



D-D continuous: $\theta \in \mathbb{R}^D$

Mixed: $\theta \in \{0,1\} \times \mathbb{R}^D \times [0,1] \times \mathbb{R}^+$ $\cdots$

1. Brute Force / Random / Grid Search

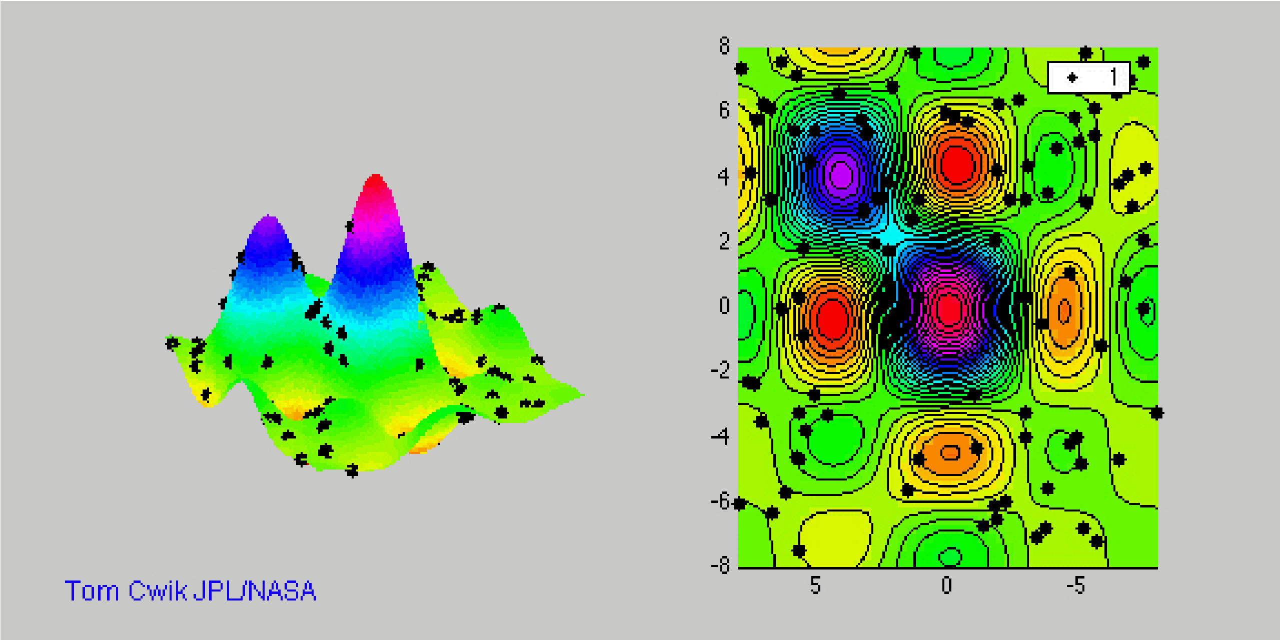


What is g *? What is* θ *? What is* Θ *?*

- Sometimes best when optimizing on a **small discrete set** of parameters
- **Ex:** DNN **architectures** or ***hyperparameters***

2. “Population-Based” Algorithms

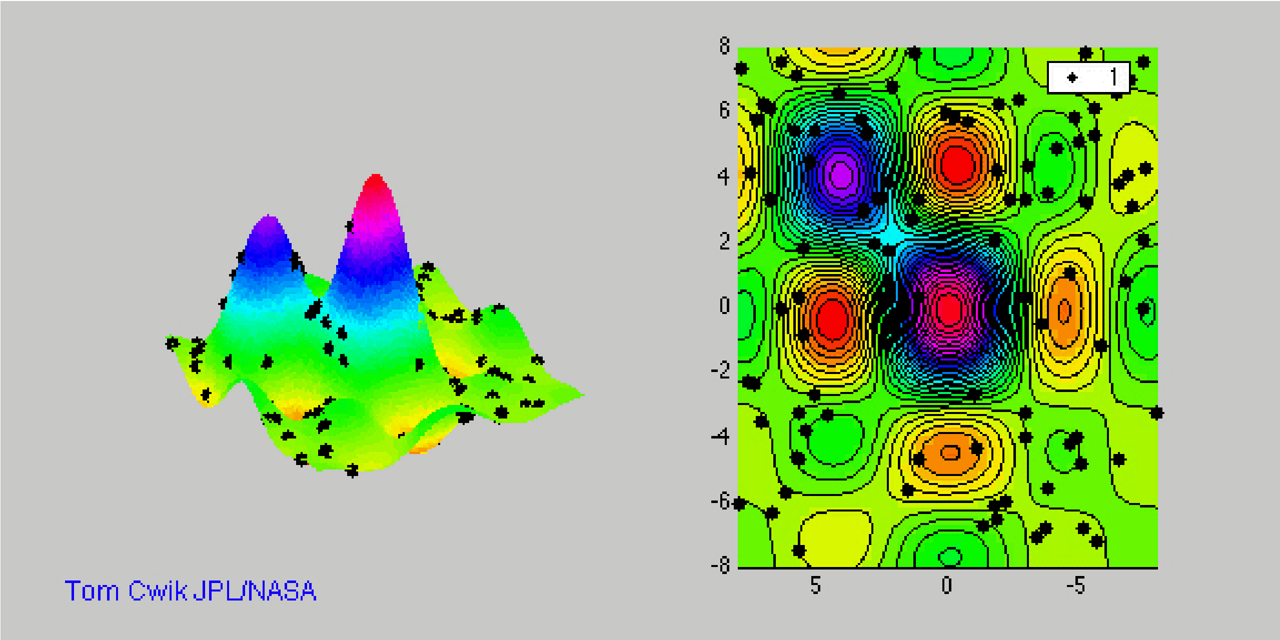
- Evolutionary/Genetic algorithms
- Particle Swarms
- Ant Colonies



What is g? What is θ? What is Θ?

2. "Population-Based" Algorithms

- Evolutionary/Genetic algorithms
- Particle Swarms
- Ant Colonies

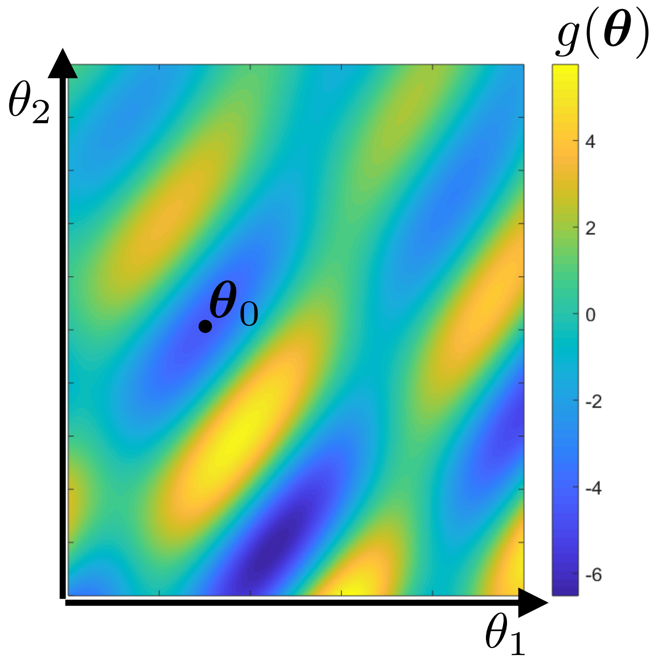


What is g? What is θ? What is Θ?

- Principle = Evolve a population.
- Strongly inspired by **nature** or **physics**
- Can be powerful and work on very general functions, but ***heuristic***

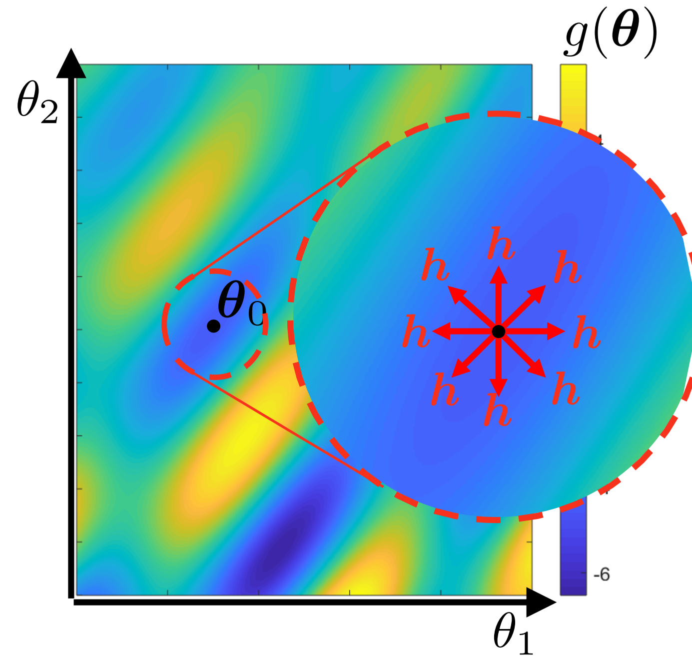
3. Calculating "*zeroes*" of the gradient

- We call ***zero*** of the gradient a point $\boldsymbol{\theta}_0 \in \mathbb{R}^P$ such that $\nabla_{\boldsymbol{x}} g(\boldsymbol{\theta}_0) = \mathbf{0}_P$.



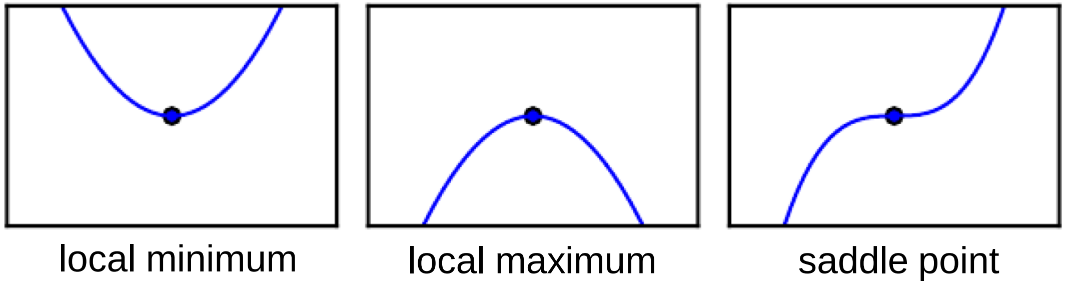
3. Calculating "*zeroes*" of the gradient

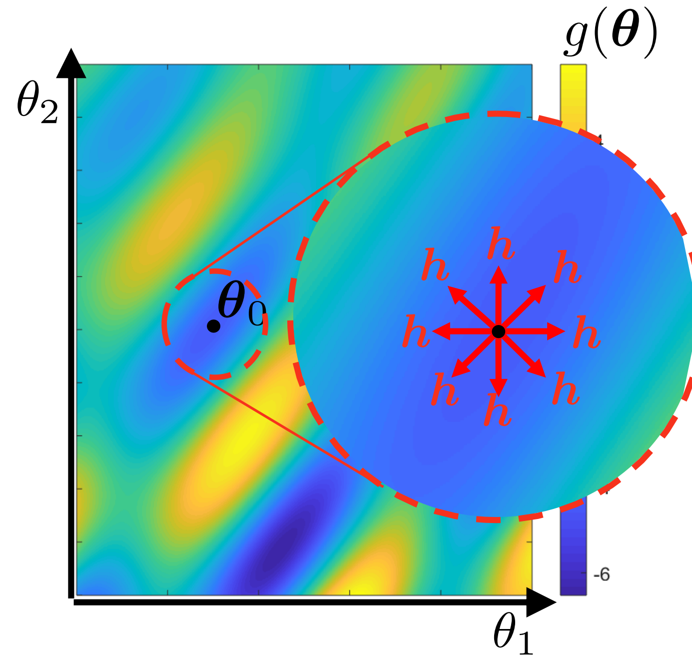
- We call ***zero*** of the gradient a point $\boldsymbol{\theta}_0 \in \mathbb{R}^P$ such that $\nabla_{\boldsymbol{x}} g(\boldsymbol{\theta}_0) = \mathbf{0}_P$.
- Also called ***stationary points*** of g : the points where g is ***locally constant***, i.e., *"flat"*.



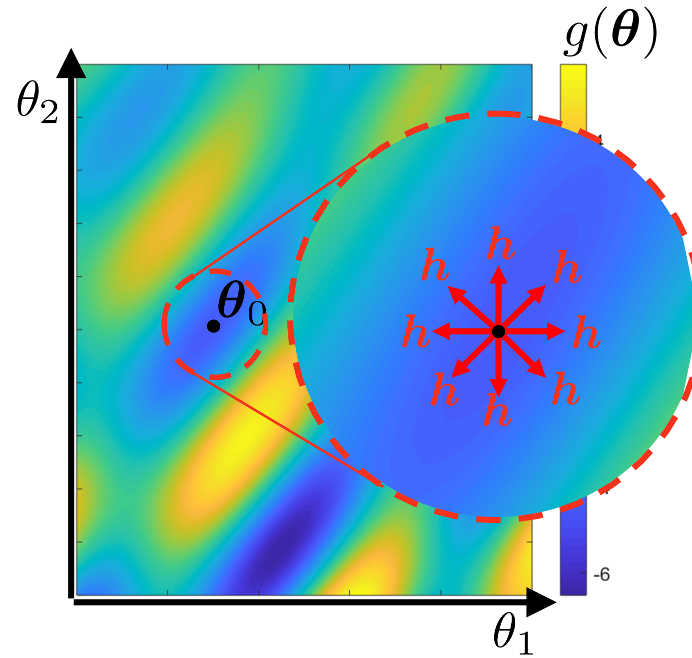
3. Calculating "*zeroes*" of the gradient

- We call ***zero*** of the gradient a point $\boldsymbol{\theta}_0 \in \mathbb{R}^P$ such that $\nabla_{\boldsymbol{x}} g(\boldsymbol{\theta}_0) = \mathbf{0}_P$.
- Also called ***stationary points*** of g : the points where g is ***locally constant***, i.e., *"flat"*.
- They may correspond to:

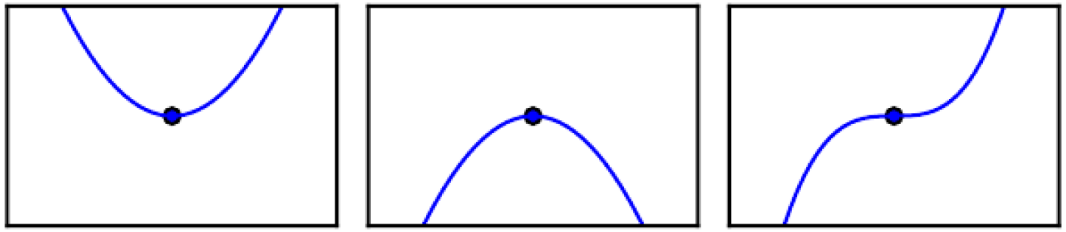




3. Calculating "*zeroes*" of the gradient



- We call ***zero*** of the gradient a point $\boldsymbol{\theta}_0 \in \mathbb{R}^P$ such that $\nabla_{\boldsymbol{x}} g(\boldsymbol{\theta}_0) = \mathbf{0}_P$.
- Also called ***stationary points*** of g : the points where g is ***locally constant***, i.e., *"flat"*.
- They may correspond to:



- In case of doubt, it is possible to distinguish between the 3 by looking at the **Hessian** $\mathbf{H}_{\boldsymbol{\theta}}[g](\boldsymbol{\theta}_0) \in \mathbb{R}^{P \times P}$ of g at $\boldsymbol{\theta}_0$:

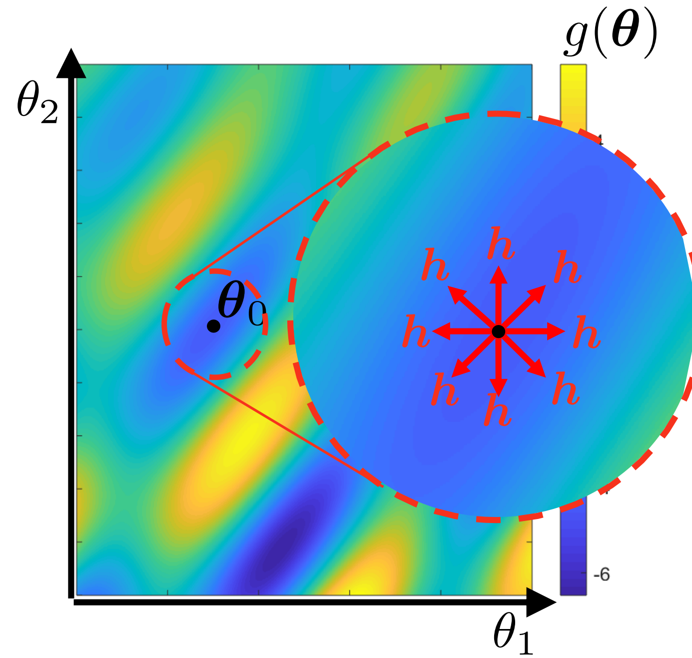
$$\mathbf{H}_{\boldsymbol{\theta}}[g](\boldsymbol{\theta}_0) \stackrel{\text{def}}{=} \mathbf{J}_{\boldsymbol{\theta}}[\nabla_{\boldsymbol{\theta}} g](\boldsymbol{\theta}_0)$$

"Second order derivative of g"

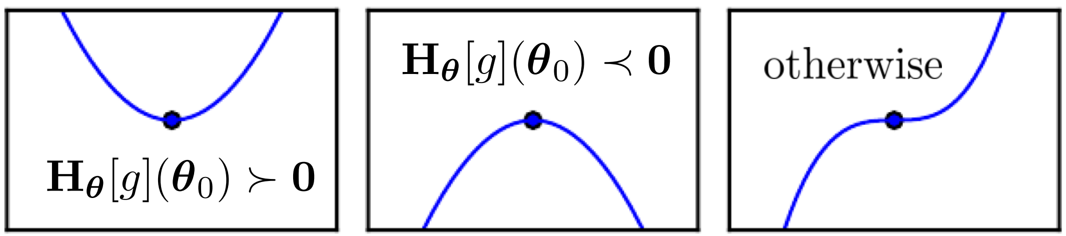
3. Calculating "*zeroes*" of the gradient

- We call ***zero*** of the gradient a point $\boldsymbol{\theta}_0 \in \mathbb{R}^P$ such that $\nabla_{\boldsymbol{x}} g(\boldsymbol{\theta}_0) = \mathbf{0}_P$.

- Also called ***stationary points*** of g : the points where g is ***locally constant***, i.e., "*flat*".



- They may correspond to:



local minimum | local maximum | saddle point

- In case of doubt, it is possible to distinguish between the 3 by looking at the **Hessian** $\mathbf{H}_{\boldsymbol{\theta}}[g](\boldsymbol{\theta}_0) \in \mathbb{R}^{P \times P}$ of g at $\boldsymbol{\theta}_0$:

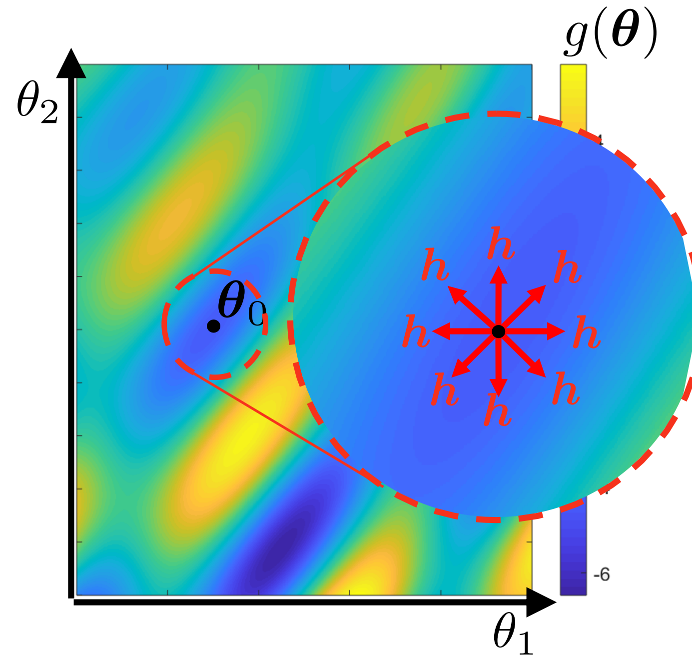
$$\mathbf{H}_{\boldsymbol{\theta}}[g](\boldsymbol{\theta}_0) \stackrel{\text{def}}{=} \mathbf{J}_{\boldsymbol{\theta}}[\nabla_{\boldsymbol{\theta}} g](\boldsymbol{\theta}_0)$$

"Second order derivative of g"

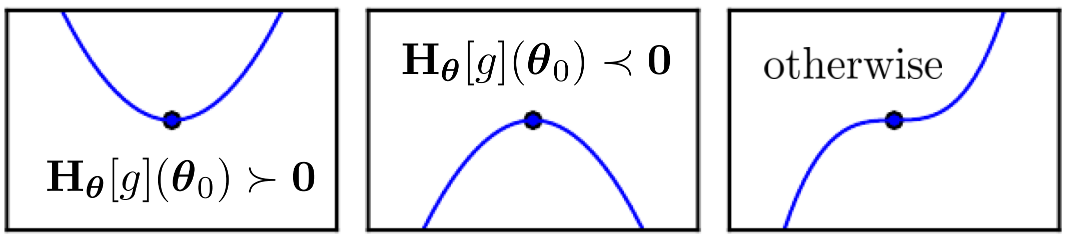
3. Calculating "*zeroes*" of the gradient

- We call ***zero*** of the gradient a point $\boldsymbol{\theta}_0 \in \mathbb{R}^P$ such that $\nabla_{\boldsymbol{x}} g(\boldsymbol{\theta}_0) = \mathbf{0}_P$.

- Also called ***stationary points*** of g : the points where g is ***locally constant***, i.e., *"flat"*.



- They may correspond to:



local minimum | local maximum | saddle point

- In case of doubt, it is possible to distinguish between the 3 by looking at the **Hessian** $\mathbf{H}_{\boldsymbol{\theta}}[g](\boldsymbol{\theta}_0) \in \mathbb{R}^{P \times P}$ of g at $\boldsymbol{\theta}_0$:

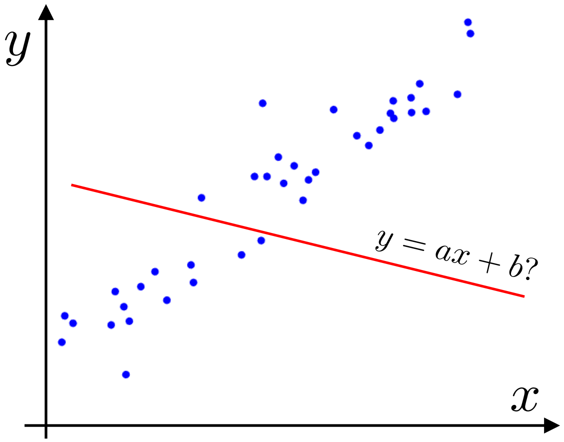
$$\mathbf{H}_{\boldsymbol{\theta}}[g](\boldsymbol{\theta}_0) \stackrel{\text{def}}{=} \mathbf{J}_{\boldsymbol{\theta}}[\nabla_{\boldsymbol{\theta}} g](\boldsymbol{\theta}_0)$$

"Second order derivative of g"

⚠ Only works if $\text{Det}\, \mathbf{H}_{\boldsymbol{\theta}}[g](\boldsymbol{\theta}_0) \neq 0$

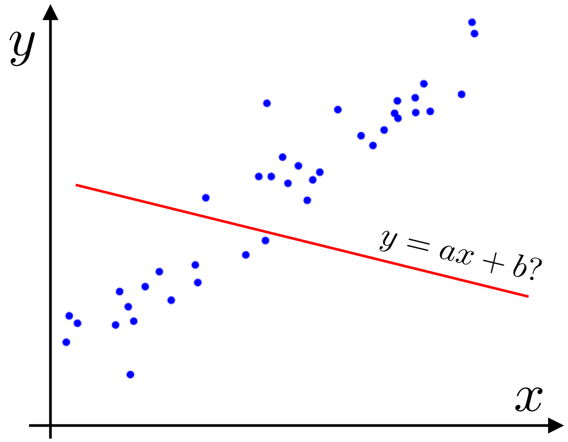
3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



3. Calculating "*zeroes*" of the gradient

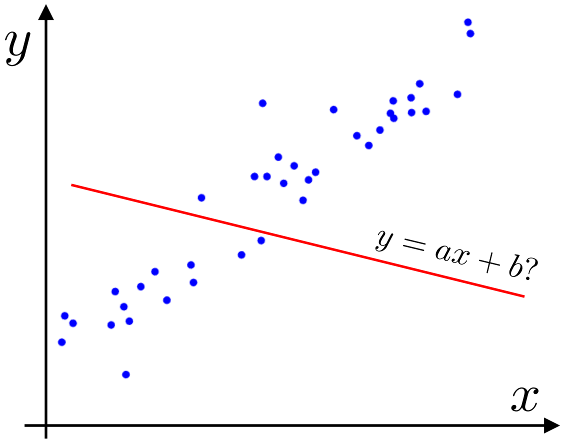
Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$

3. Calculating "*zeroes*" of the gradient

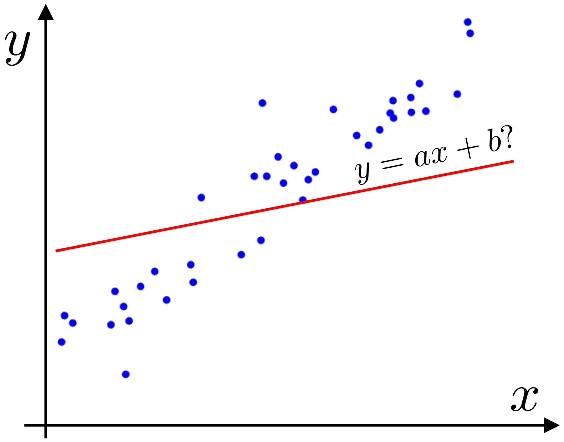
Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$

3. Calculating "*zeroes*" of the gradient

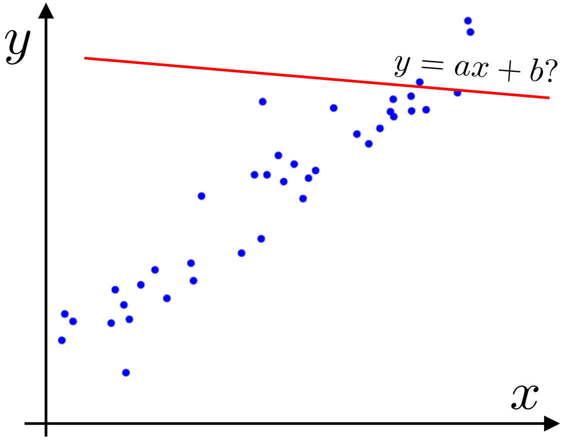
Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$

3. Calculating "*zeroes*" of the gradient

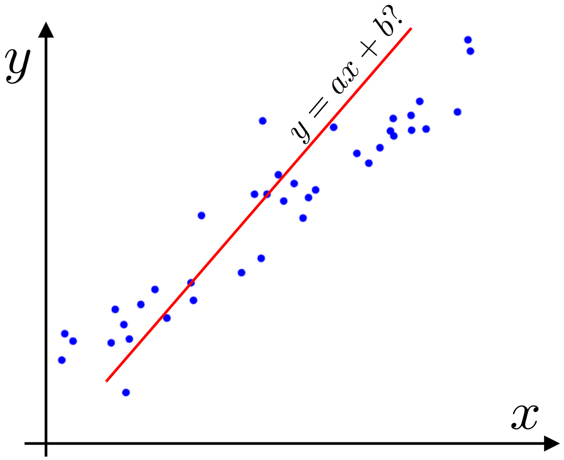
Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$

3. Calculating "*zeroes*" of the gradient

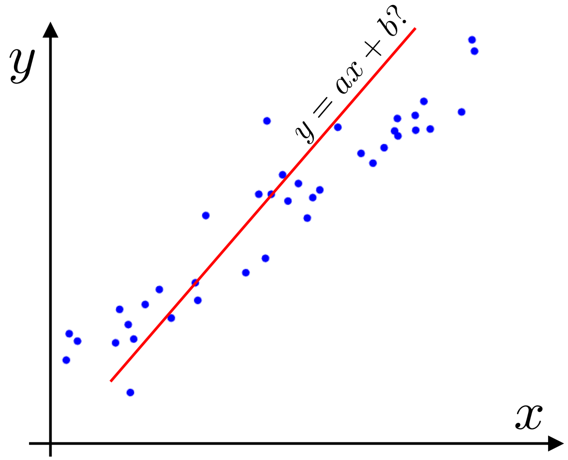
Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$

3. Calculating "*zeroes*" of the gradient

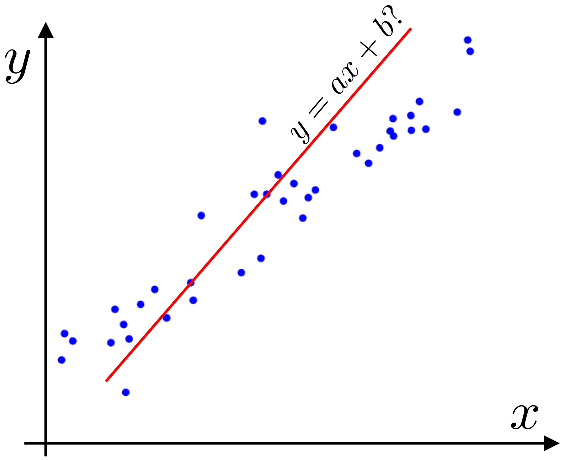
Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

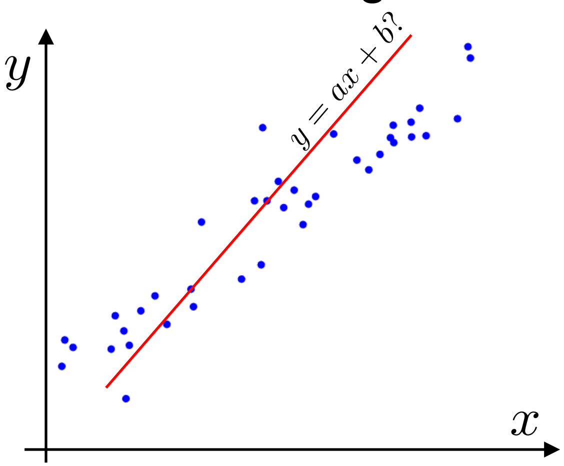
- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$

$g(\boldsymbol{\theta}) = ?$

$\nabla_\theta g(\boldsymbol{\theta}_0) = ?$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

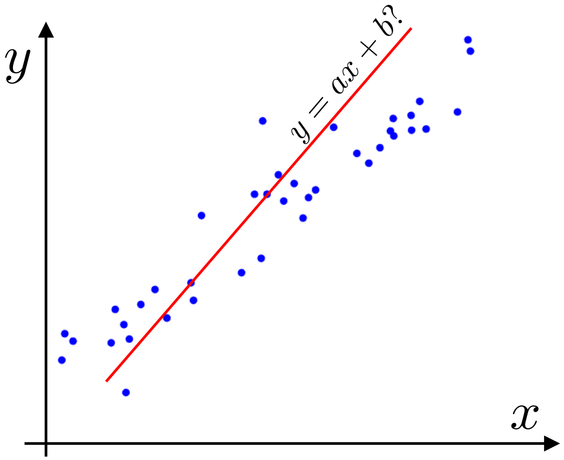
- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

$g(\boldsymbol{\theta}) = ?$

$\nabla_\theta g(\boldsymbol{\theta}_0) = ?$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

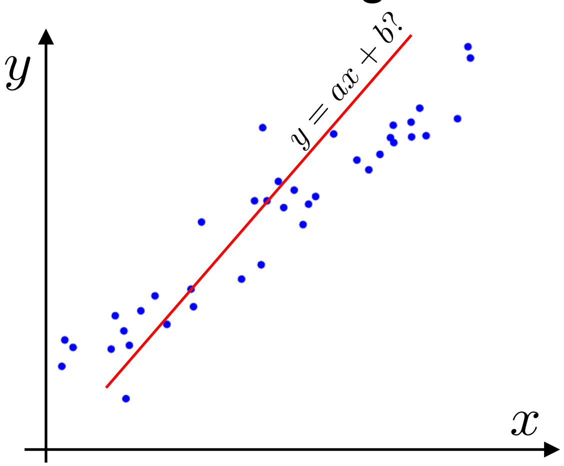
- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

$$g(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}(ax_t + b - y_t)^2$$

$$\nabla_\theta g(\boldsymbol{\theta}_0) = ?$$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

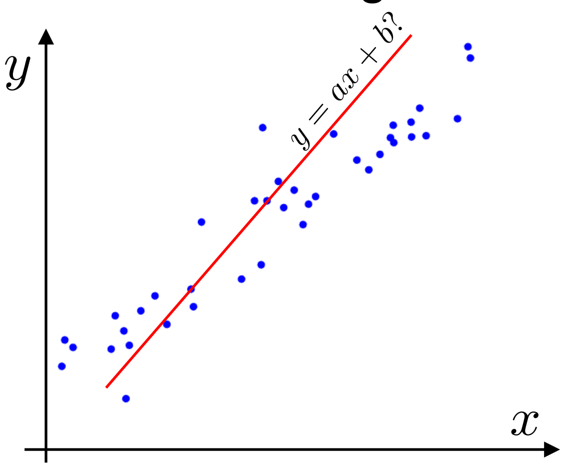
- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

$$g(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}(ax_t + b - y_t)^2 = \frac{1}{T}\sum_{t=1}^{T}\left([x_t, 1]^\top \begin{bmatrix} a \\ b \end{bmatrix} - y_t\right)^2$$

$$\nabla_\theta g(\boldsymbol{\theta}_0) = ?$$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^T$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^T (f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

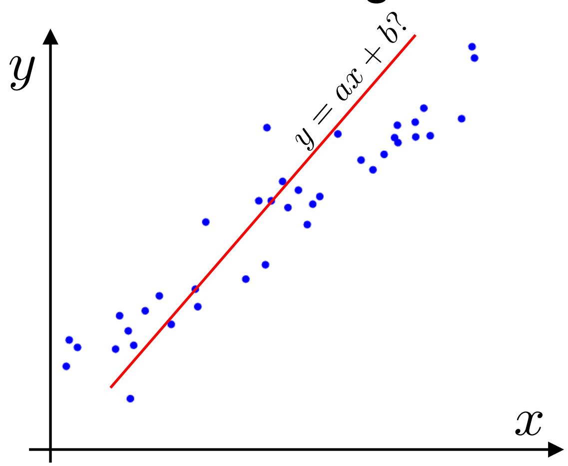
- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

$$g(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^T (ax_t + b - y_t)^2 = \frac{1}{T}\sum_{t=1}^T \Bigg(\underbrace{[x_t, 1]^\top}_{\boldsymbol{w}_t \in \mathbb{R}^2} \begin{bmatrix} a \\ b \end{bmatrix} - y_t\Bigg)^2$$

$$\nabla_\theta g(\boldsymbol{\theta}_0) = ?$$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

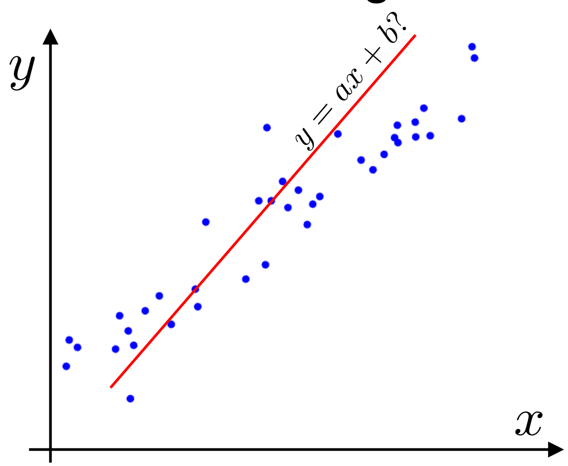
- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

$$g(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}(ax_t + b - y_t)^2 = \frac{1}{T}\sum_{t=1}^{T}\Bigg(\underbrace{[x_t, 1]}_{\boldsymbol{w}_t \in \mathbb{R}^2}{}^\top \begin{bmatrix} a \\ b \end{bmatrix} - y_t\Bigg)^2 = \frac{1}{T}\sum_{t=1}^{T}\left(\boldsymbol{w}_t^\top \boldsymbol{\theta} - y_t\right)^2$$

$$\nabla_\theta g(\boldsymbol{\theta}_0) = ?$$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^T$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^T (f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

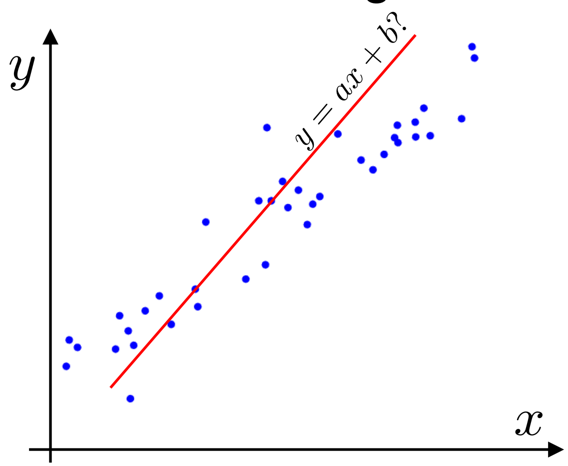
$$g(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^T (ax_t + b - y_t)^2 = \frac{1}{T}\sum_{t=1}^T \Bigg(\underbrace{[x_t, 1]^\top}_{\boldsymbol{w}_t \in \mathbb{R}^2} \begin{bmatrix} a \\ b \end{bmatrix} - y_t\Bigg)^2 = \frac{1}{T}\sum_{t=1}^T (\boldsymbol{w}_t^\top \boldsymbol{\theta} - y_t)^2$$

$$= \frac{1}{T}\|\mathbf{W}\boldsymbol{\theta} - \boldsymbol{y}\|_2^2, \text{ where } \mathbf{W} = [\boldsymbol{w}_1^\top, \dots, \boldsymbol{w}_T^\top]^\top \in \mathbb{R}^{T\times 2}, \quad \boldsymbol{y} = [y_1, \dots, y_T]^\top \in \mathbb{R}^T$$

$$\nabla_\theta g(\boldsymbol{\theta}_0) = ?$$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^T$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

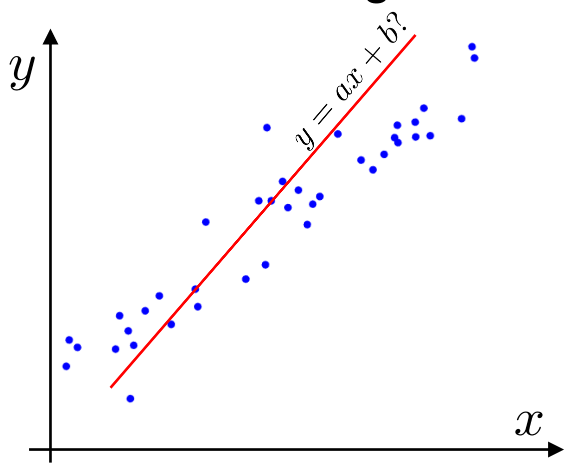
$$g(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}(ax_t + b - y_t)^2 = \frac{1}{T}\sum_{t=1}^{T}\Big(\underbrace{[x_t, 1]^\top}_{\boldsymbol{w}_t \in \mathbb{R}^2}\begin{bmatrix}a\\b\end{bmatrix} - y_t\Big)^2 = \frac{1}{T}\sum_{t=1}^{T}(\boldsymbol{w}_t^\top\boldsymbol{\theta} - y_t)^2$$

$$= \frac{1}{T}\|\mathbf{W}\boldsymbol{\theta} - \boldsymbol{y}\|_2^2, \text{ where } \mathbf{W} = [\boldsymbol{w}_1^\top, \dots, \boldsymbol{w}_T^\top]^\top \in \mathbb{R}^{T\times 2}, \quad \boldsymbol{y} = [y_1, \dots, y_T]^\top \in \mathbb{R}^T$$

$$\nabla_\theta g(\boldsymbol{\theta}_0) = 2\mathbf{W}^\top(\mathbf{W}\boldsymbol{\theta}_0 - \boldsymbol{y})$$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

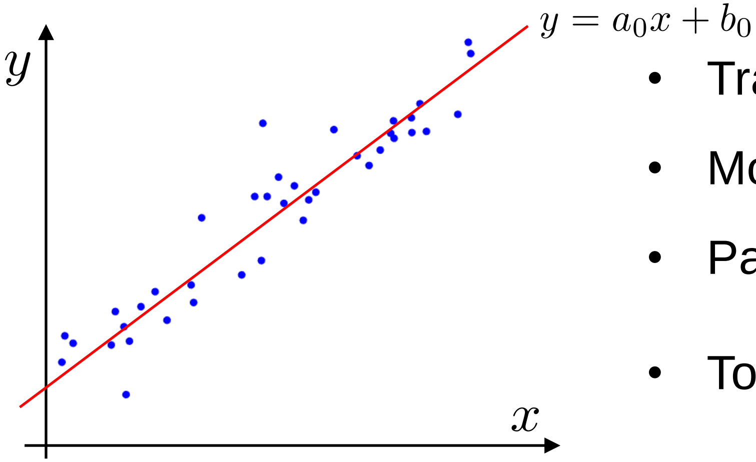
$$g(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}(ax_t + b - y_t)^2 = \frac{1}{T}\sum_{t=1}^{T}\Bigg(\underbrace{[x_t, 1]^\top}_{\boldsymbol{w}_t \in \mathbb{R}^2}\begin{bmatrix} a \\ b \end{bmatrix} - y_t\Bigg)^2 = \frac{1}{T}\sum_{t=1}^{T}\left(\boldsymbol{w}_t^\top\boldsymbol{\theta} - y_t\right)^2$$

$$= \frac{1}{T}\|\mathbf{W}\boldsymbol{\theta} - \boldsymbol{y}\|_2^2, \quad \text{where } \mathbf{W} = [\boldsymbol{w}_1^\top, \dots, \boldsymbol{w}_T^\top]^\top \in \mathbb{R}^{T\times 2}, \quad \boldsymbol{y} = [y_1, \dots, y_T]^\top \in \mathbb{R}^T$$

$$\nabla_\theta g(\boldsymbol{\theta}_0) = 2\mathbf{W}^\top(\mathbf{W}\boldsymbol{\theta}_0 - \boldsymbol{y}) = \mathbf{0}_2 \quad \Rightarrow \quad \boldsymbol{\theta}_0 = (\mathbf{W}^\top\mathbf{W})^{-1}\mathbf{W}^\top\boldsymbol{y} = \mathbf{W}^\dagger\boldsymbol{y}$$

3. Calculating "*zeroes*" of the gradient

Exercise: Fitting an **affine model** via ***least squares***



- Training set: $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T}$
- Models: $f_{\boldsymbol{\theta}}(x) = ax + b$
- Parameters: $\boldsymbol{\theta} = [a, b]^\top \in \mathbb{R}^2$
- Total Loss: $g(\boldsymbol{\theta}) = L(f_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}(f_{\boldsymbol{\theta}}(x_t) - y_t)^2$

- Find $\boldsymbol{\theta}_0$ such that $\nabla_\theta g(\boldsymbol{\theta}_0) = \mathbf{0}_2$ **Hint:** we *already* calculated $\nabla_\theta g(\boldsymbol{\theta}_0)$!

$$g(\boldsymbol{\theta}) = \frac{1}{T}\sum_{t=1}^{T}(ax_t + b - y_t)^2 = \frac{1}{T}\sum_{t=1}^{T}\Bigg(\underbrace{[x_t, 1]^\top}_{\boldsymbol{w}_t \in \mathbb{R}^2}\begin{bmatrix} a \\ b \end{bmatrix} - y_t\Bigg)^2 = \frac{1}{T}\sum_{t=1}^{T}\left(\boldsymbol{w}_t^\top \boldsymbol{\theta} - y_t\right)^2$$

$$= \frac{1}{T}\|\mathbf{W}\boldsymbol{\theta} - \boldsymbol{y}\|_2^2, \text{ where } \mathbf{W} = [\boldsymbol{w}_1^\top, \dots, \boldsymbol{w}_T^\top]^\top \in \mathbb{R}^{T\times 2}, \quad \boldsymbol{y} = [y_1, \dots, y_T]^\top \in \mathbb{R}^T$$

$$\nabla_\theta g(\boldsymbol{\theta}_0) = 2\mathbf{W}^\top(\mathbf{W}\boldsymbol{\theta}_0 - \boldsymbol{y}) = \mathbf{0}_2 \quad \Rightarrow \quad \boldsymbol{\theta}_0 = (\mathbf{W}^\top\mathbf{W})^{-1}\mathbf{W}^\top\boldsymbol{y} = \mathbf{W}^\dagger\boldsymbol{y}$$

4. Alternated Minimization

4. Alternated Minimization

$$\underset{\theta_1,\ldots,\theta_P}{\operatorname{argmin}}\ g(\theta_1,\ldots,\theta_P)?$$

4. Alternated Minimization

$$\underset{\theta_1,\ldots,\theta_P}{\operatorname{argmin}} \; g(\theta_1,\ldots,\theta_P)?$$

$$\left\{\begin{array}{l}
\theta_1^{(i+1)} = \underset{\theta_1}{\operatorname{argmin}} \; g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\
\theta_2^{(i+1)} = \underset{\theta_2}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\
\quad \vdots \qquad\qquad\qquad \vdots \\
\theta_P^{(i+1)} = \underset{\theta_P}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P)
\end{array}\right.$$

4. Alternated Minimization

$$\underset{\theta_1,\dots,\theta_P}{\operatorname{argmin}} \; g(\theta_1,\dots,\theta_P)?$$

Converges, but not necessarily to the global minimum

$$\left\{\begin{array}{l}
\theta_1^{(i+1)} = \underset{\theta_1}{\operatorname{argmin}} \; g(\theta_1, \theta_2^{(i)}, \dots, \theta_P^{(i)}) \\
\theta_2^{(i+1)} = \underset{\theta_2}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2, \dots, \theta_P^{(i)}) \\
\quad\vdots \qquad\qquad\qquad \vdots \\
\theta_P^{(i+1)} = \underset{\theta_P}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \dots, \theta_P)
\end{array}\right.$$

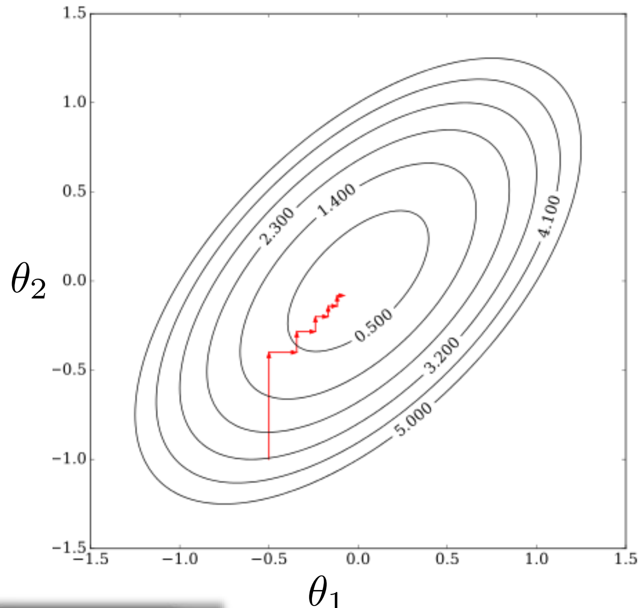
4. Alternated Minimization

$$\underset{\theta_1,\dots,\theta_P}{\operatorname{argmin}} \; g(\theta_1,\dots,\theta_P)?$$

Converges, but not necessarily to the global minimum

$$\begin{cases} \theta_1^{(i+1)} = \underset{\theta_1}{\operatorname{argmin}} \; g(\theta_1, \theta_2^{(i)}, \dots, \theta_P^{(i)}) \\ \theta_2^{(i+1)} = \underset{\theta_2}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2, \dots, \theta_P^{(i)}) \\ \vdots \\ \theta_P^{(i+1)} = \underset{\theta_P}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \dots, \theta_P) \end{cases}$$

- For θ_p scalar: *coordinate descent*



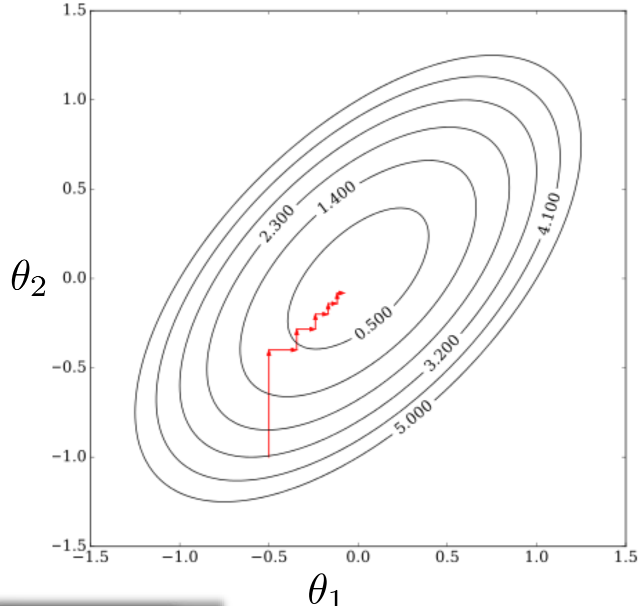
4. Alternated Minimization

$$\underset{\theta_1,\ldots,\theta_P}{\operatorname{argmin}} \; g(\theta_1,\ldots,\theta_P)?$$

Converges, but not necessarily to the global minimum

$$\begin{cases} \theta_1^{(i+1)} = \underset{\theta_1}{\operatorname{argmin}} \; g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\ \theta_2^{(i+1)} = \underset{\theta_2}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\ \vdots \qquad\qquad \vdots \\ \theta_P^{(i+1)} = \underset{\theta_P}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P) \end{cases}$$

- For θ_p scalar: *coordinate descent*



- Convenient when:
  - Variables are mixed discrete / continuous
  - There are direct solutions wrt. each variable

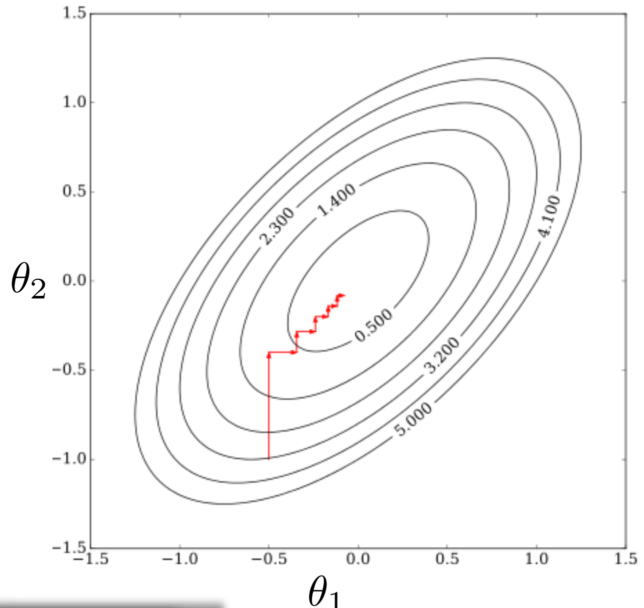
4. Alternated Minimization

$$\underset{\theta_1,\ldots,\theta_P}{\operatorname{argmin}} \; g(\theta_1,\ldots,\theta_P)?$$

$$\begin{cases} \theta_1^{(i+1)} = \underset{\theta_1}{\operatorname{argmin}} \; g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\ \theta_2^{(i+1)} = \underset{\theta_2}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\ \vdots \qquad\qquad\qquad \vdots \\ \theta_P^{(i+1)} = \underset{\theta_P}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P) \end{cases}$$

Converges, but not necessarily to the global minimum

- For θ_p scalar: *coordinate descent*



- Convenient when:
  - Variables are mixed discrete / continuous
  - There are direct solutions wrt. each variable
- Sometimes, introducing **new variables** and **then** performing AM yields efficient algorithms, e.g., *Expectation-Maximization (EM)* or *ADMM*

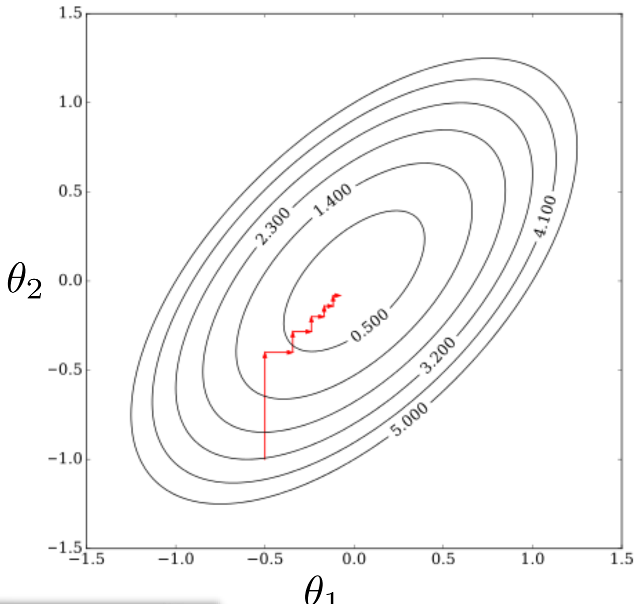
4. Alternated Minimization

$$\underset{\theta_1,\ldots,\theta_P}{\operatorname{argmin}} \; g(\theta_1,\ldots,\theta_P)?$$

Converges, but not necessarily to the global minimum

$$\left\{\begin{array}{l}
\theta_1^{(i+1)} = \underset{\theta_1}{\operatorname{argmin}} \; g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\
\theta_2^{(i+1)} = \underset{\theta_2}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\
\quad \vdots \qquad\qquad\qquad \vdots \\
\theta_P^{(i+1)} = \underset{\theta_P}{\operatorname{argmin}} \; g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P)
\end{array}\right.$$

- For θ_p scalar: *coordinate descent*



- Convenient when:
  - Variables are mixed discrete / continuous
  - There are direct solutions wrt. each variable
- Sometimes, introducing **new variables** and **then** performing AM yields efficient algorithms, e.g., *Expectation-Maximization (EM)* or *ADMM*
- Variant: Alternate between **minimization** and **projection** onto *constraints* (e.g.: $\theta \geq 0$)

5. Gradient Descent

Intuition:

- Start from an initial **parameter vector** $\boldsymbol{\theta}^{(0)} \in \mathbb{R}^P$
- From here, follow the ***direction of steepest descent***
- Stop when things look ***flat***

5. Gradient Descent

Intuition:

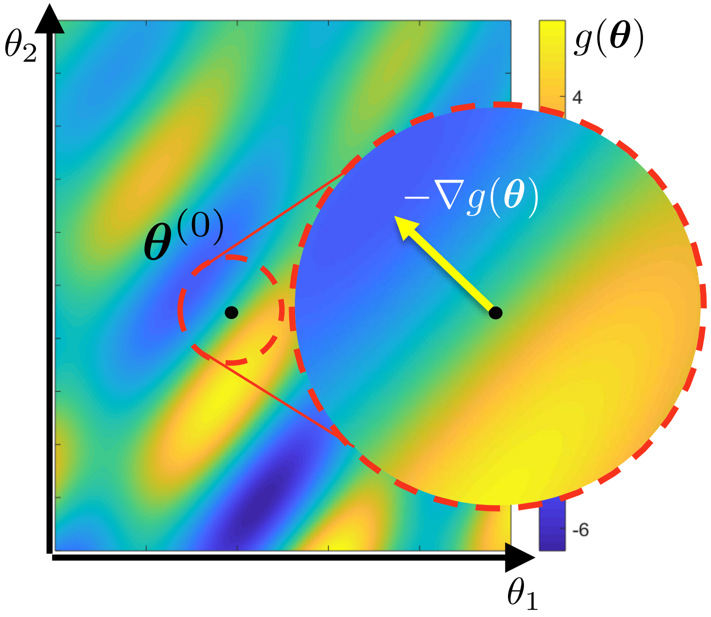
- Start from an initial **parameter vector** $\boldsymbol{\theta}^{(0)} \in \mathbb{R}^P$
- From here, follow the ***direction of steepest descent***
- Stop when things look ***flat***

$$-\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(0)})$$

5. Gradient Descent

Intuition:
- Start from an initial **parameter vector** $\boldsymbol{\theta}^{(0)} \in \mathbb{R}^P$
- From here, follow the ***direction of steepest descent***
- Stop when things look ***flat***

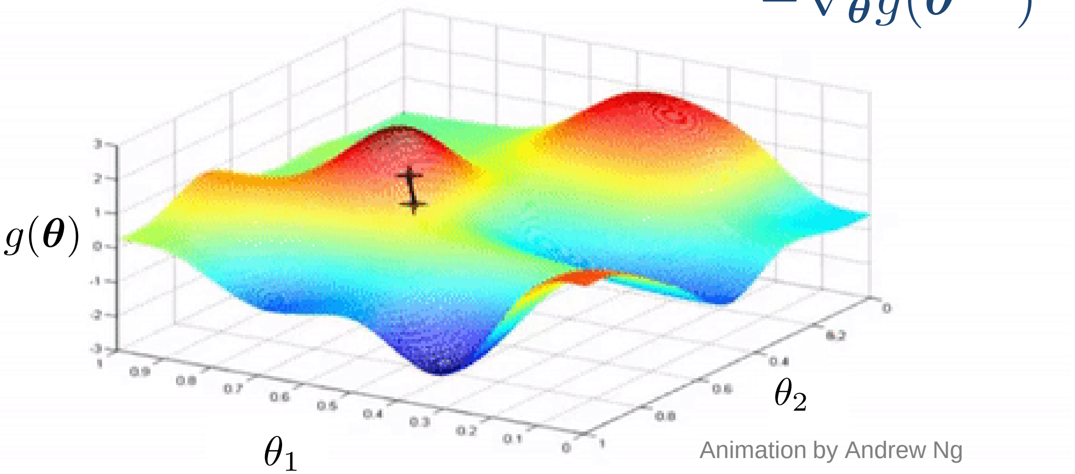
$$-\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(0)})$$



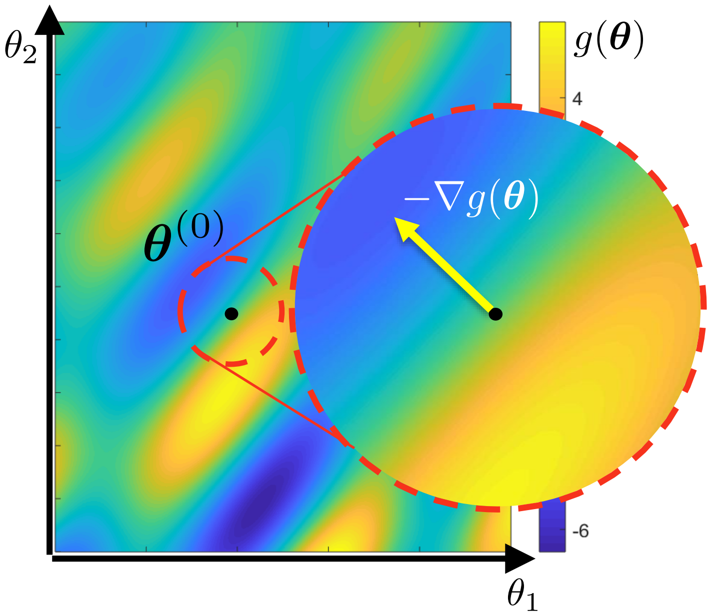
5. Gradient Descent

Intuition:
- Start from an initial **parameter vector** $\boldsymbol{\theta}^{(0)} \in \mathbb{R}^P$
- From here, follow the ***direction of steepest descent***
- Stop when things look ***flat***

$$-\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(0)})$$



Animation by Andrew Ng

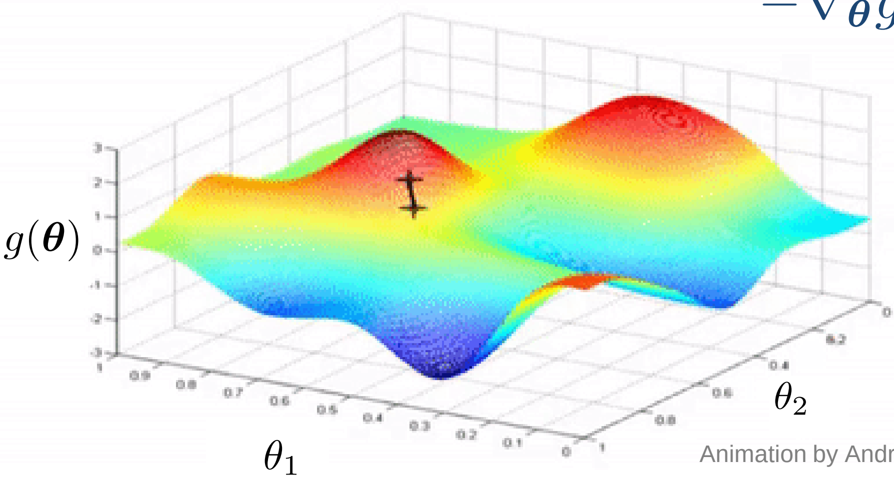


5. Gradient Descent

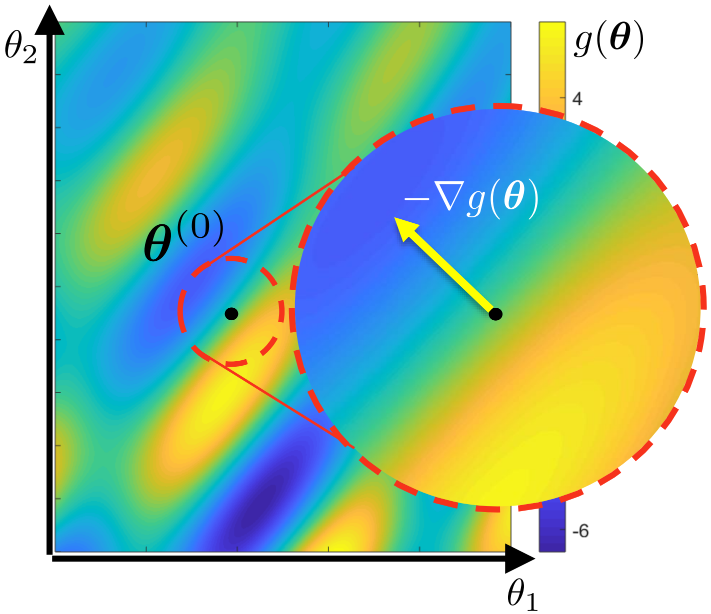
Intuition:

- Start from an initial **parameter vector** $\boldsymbol{\theta}^{(0)} \in \mathbb{R}^P$
- From here, follow the ***direction of steepest descent***
- Stop when things look ***flat***

$-\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(0)})$



Animation by Andrew Ng



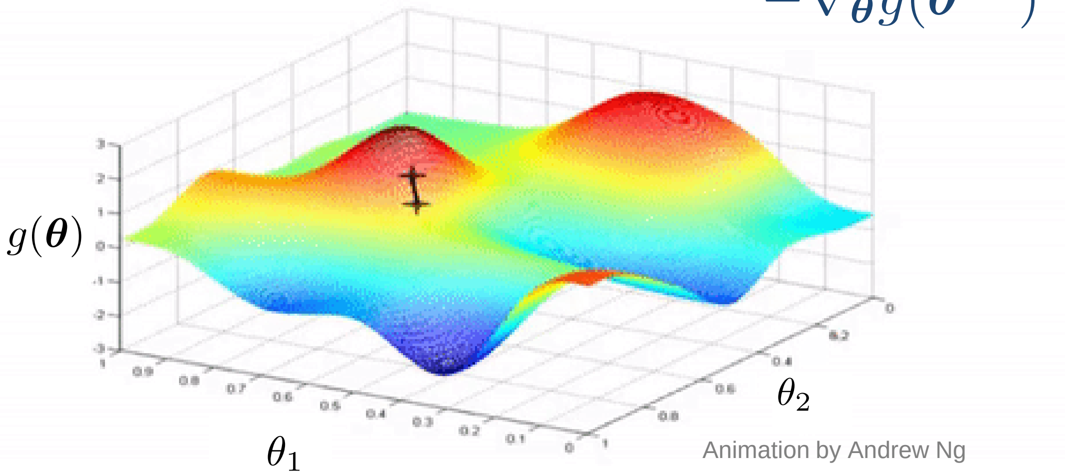
- The updates are: $\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$

5. Gradient Descent

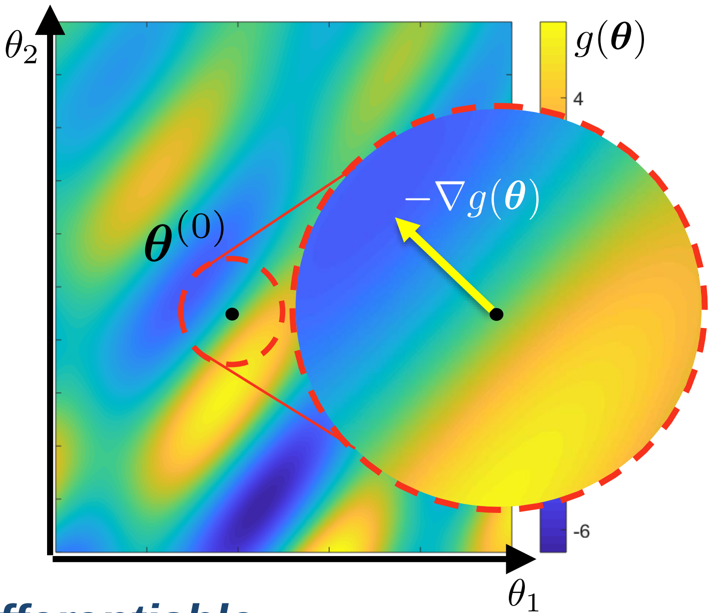
Intuition:

- Start from an initial **parameter vector** $\boldsymbol{\theta}^{(0)} \in \mathbb{R}^P$
- From here, follow the ***direction of steepest descent***
- Stop when things look ***flat***

$$-\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(0)})$$



Animation by Andrew Ng

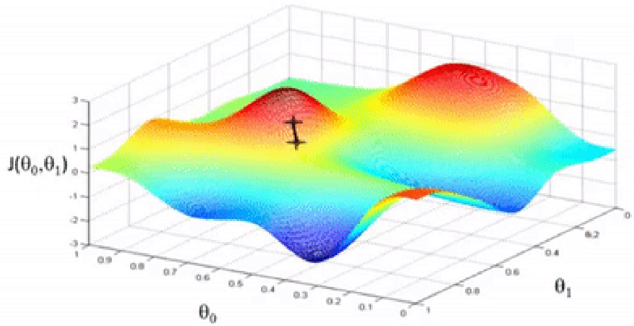


- The updates are: $$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- Requires the function to be (almost everywhere) ***differentiable***

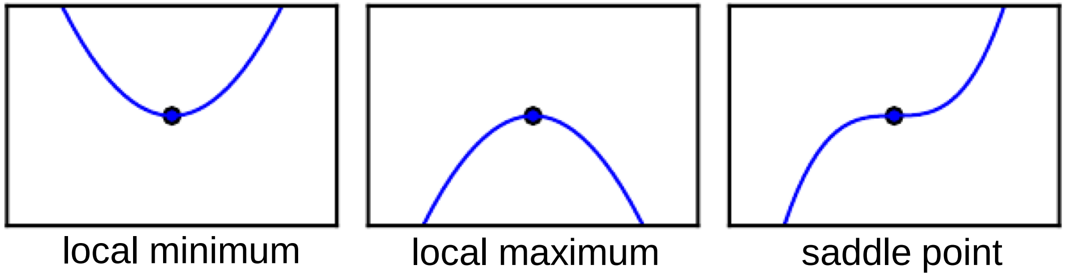
5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$



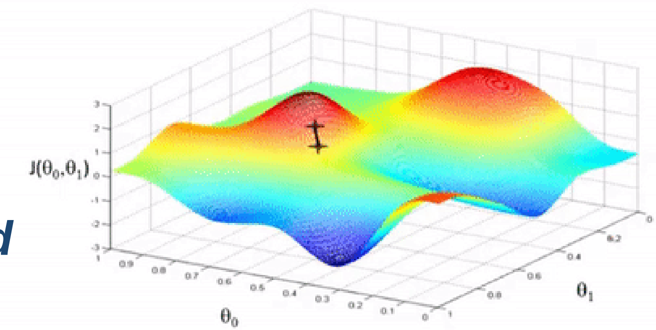
Animation by Andrew Ng

- Local maxima and saddle points are ***unstable fixed points***, while local minima are ***stable fixed points***



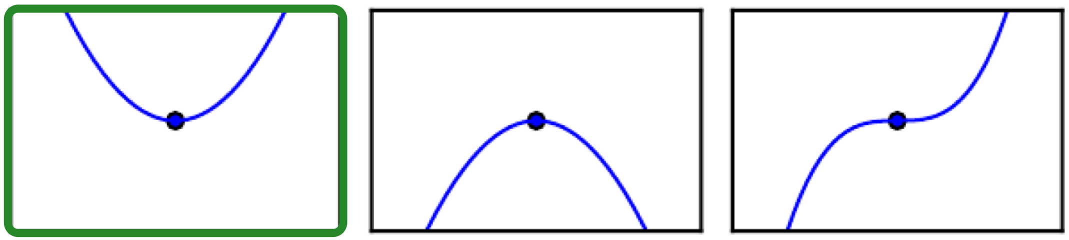
5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$



Animation by Andrew Ng

- Local maxima and saddle points are ***unstable fixed points***, while local minima are ***stable fixed points***

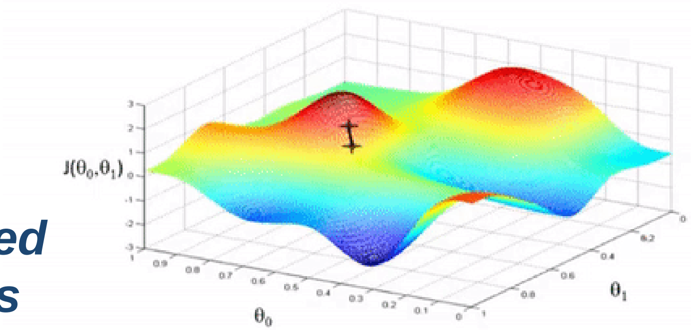


local minimum | local maximum | saddle point

→ ***The algorithm converges to local minima under mild assumptions*** ☺

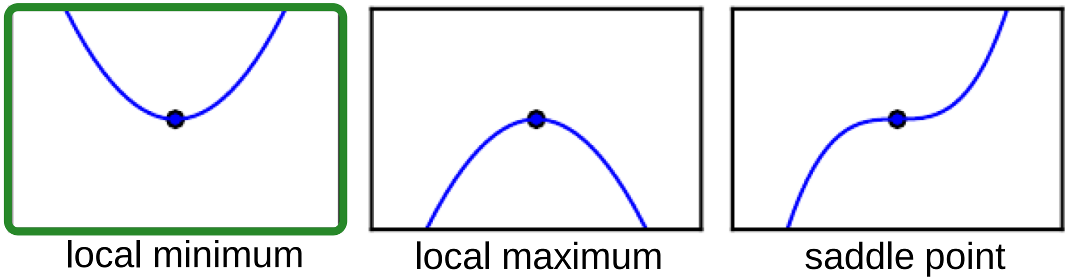
5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

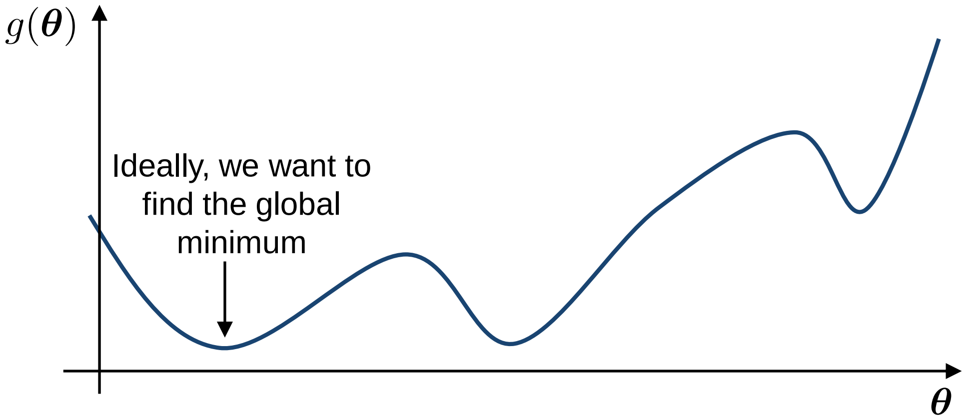


Animation by Andrew Ng

- Local maxima and saddle points are ***unstable fixed points***, while local minima are ***stable fixed points***

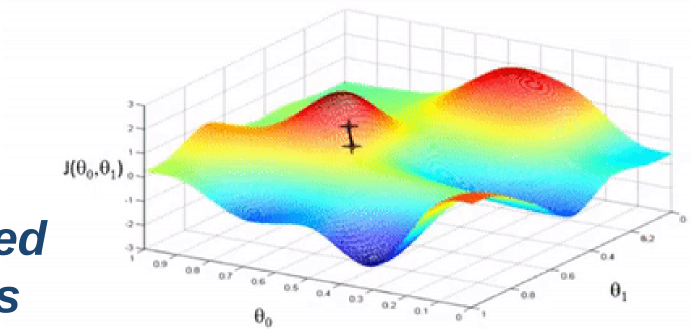


→ ***The algorithm converges to local minima under mild assumptions*** ☺



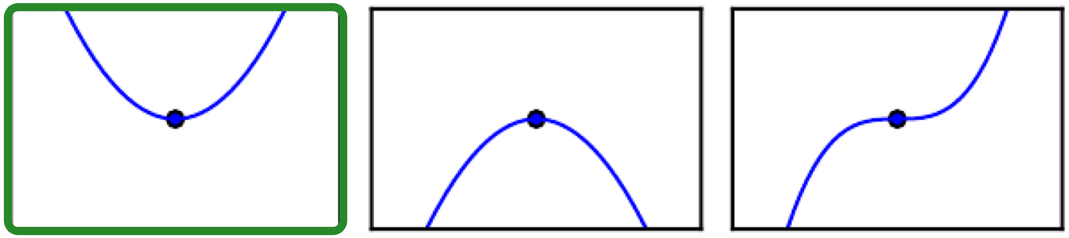
5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$



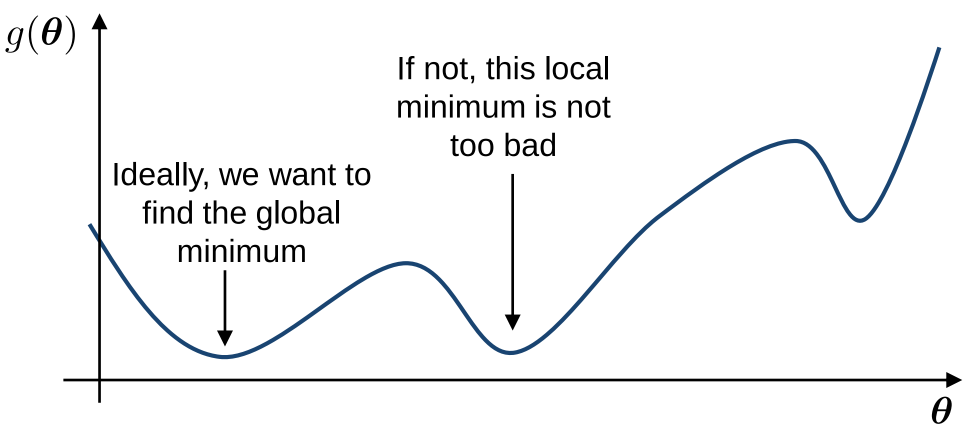
Animation by Andrew Ng

- Local maxima and saddle points are ***unstable fixed points***, while local minima are ***stable fixed points***



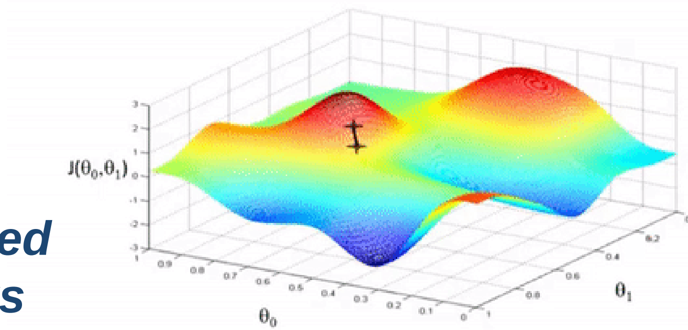
local minimum | local maximum | saddle point

→ ***The algorithm converges to local minima under mild assumptions*** ☺



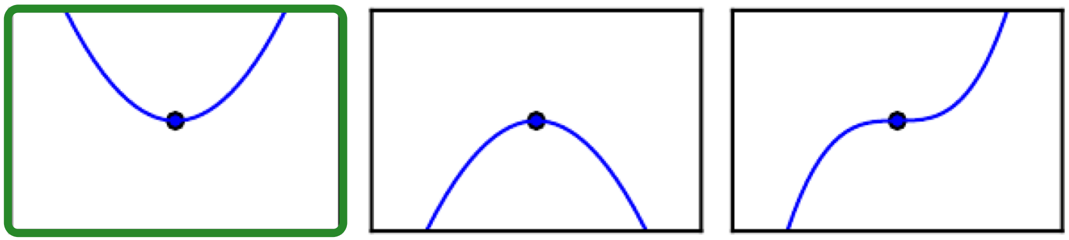
5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

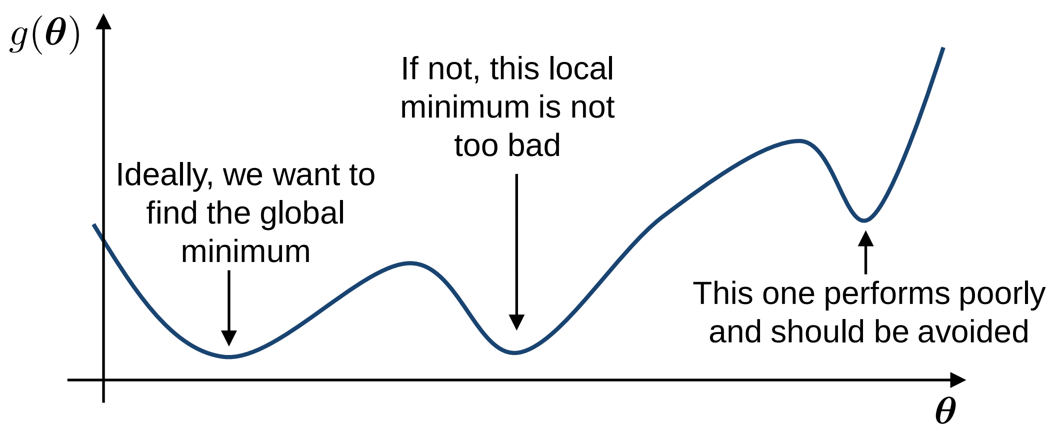


Animation by Andrew Ng

- Local maxima and saddle points are ***unstable fixed points***, while local minima are ***stable fixed points***

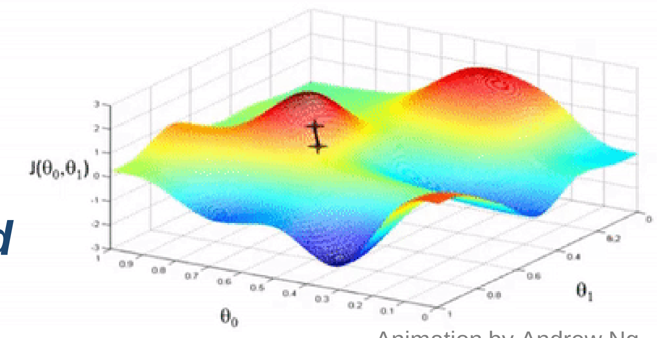


→ ***The algorithm converges to local minima under mild assumptions*** ☺

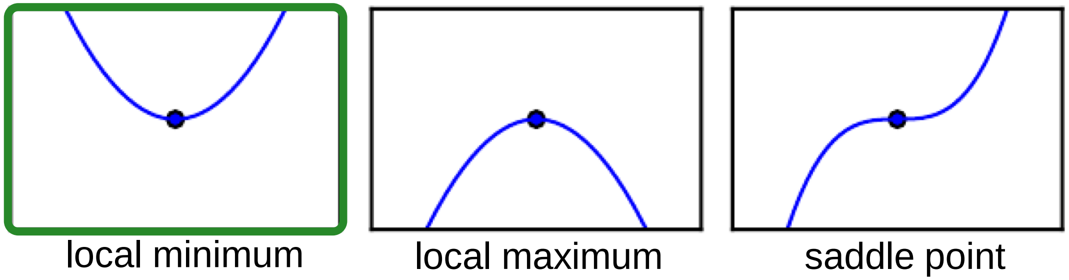


5. Gradient Descent

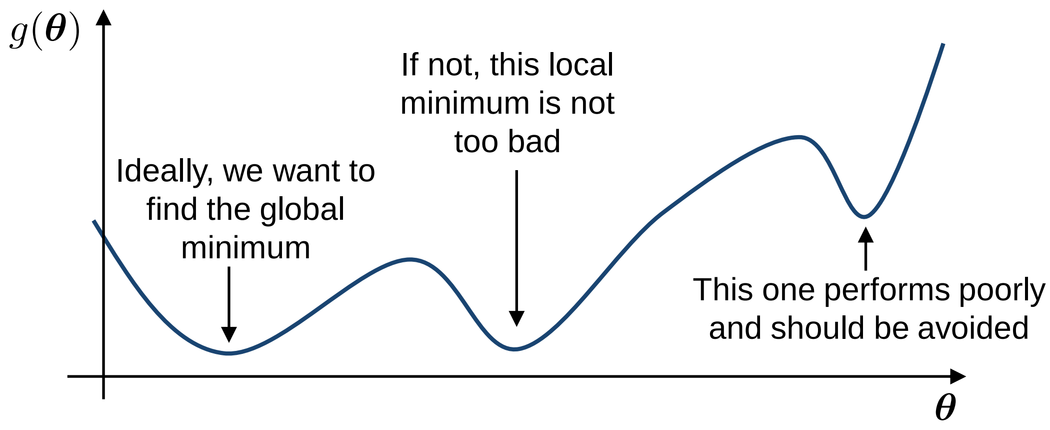
$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$



- Local maxima and saddle points are ***unstable fixed points***, while local minima are ***stable fixed points***



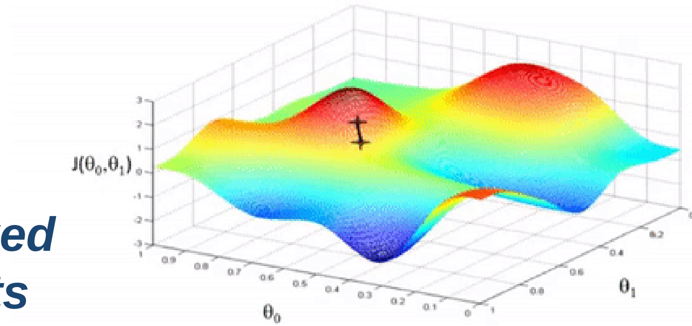
→ ***The algorithm converges to local minima under mild assumptions*** ☺



- ***Spurious local minima*** cannot always be avoided

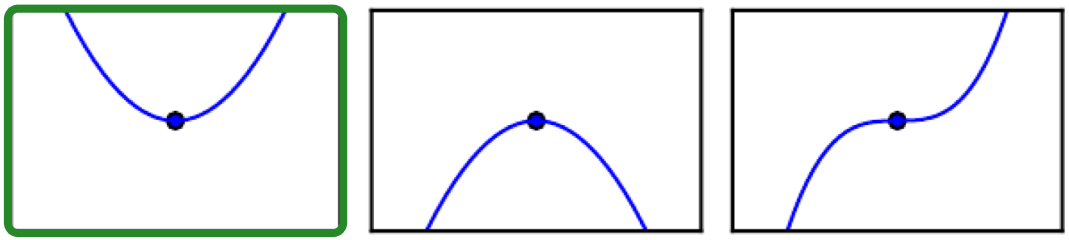
5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

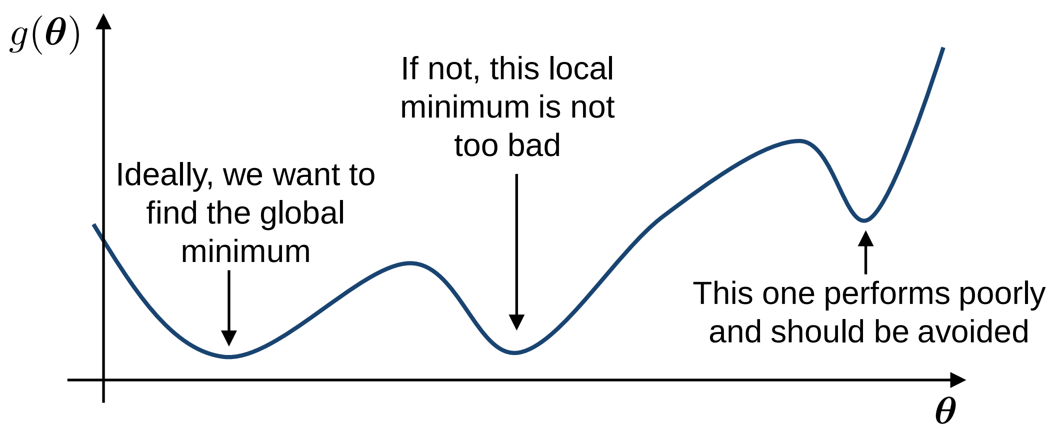


Animation by Andrew Ng

- Local maxima and saddle points are ***unstable fixed points***, while local minima are ***stable fixed points***


local minimum   local maximum   saddle point

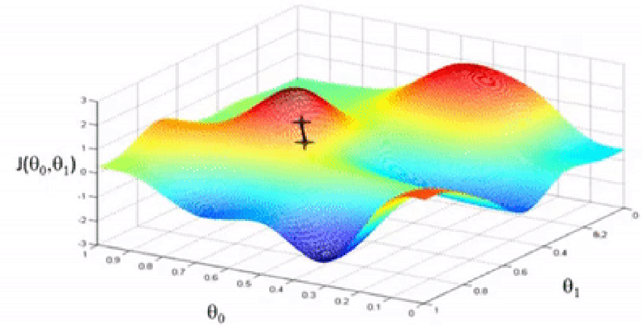
→ ***The algorithm converges to local minima under mild assumptions*** ☺



- ***Spurious local minima*** cannot always be avoided
- Many variants have been derived to limit them, and to speed up convergence

5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

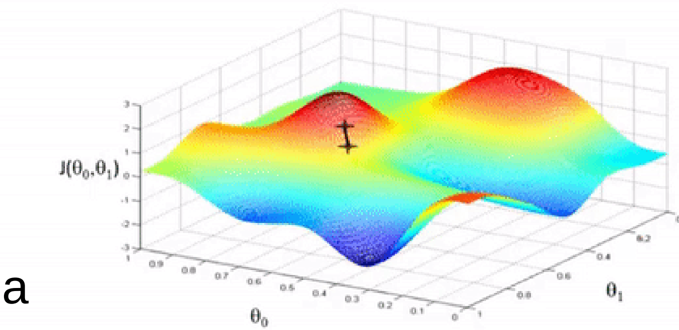


Animation by Andrew Ng

- The ***gradient-step*** ϵ, also called ***learning rate***, is a critical ***hyper-parameter*** of the algorithm.

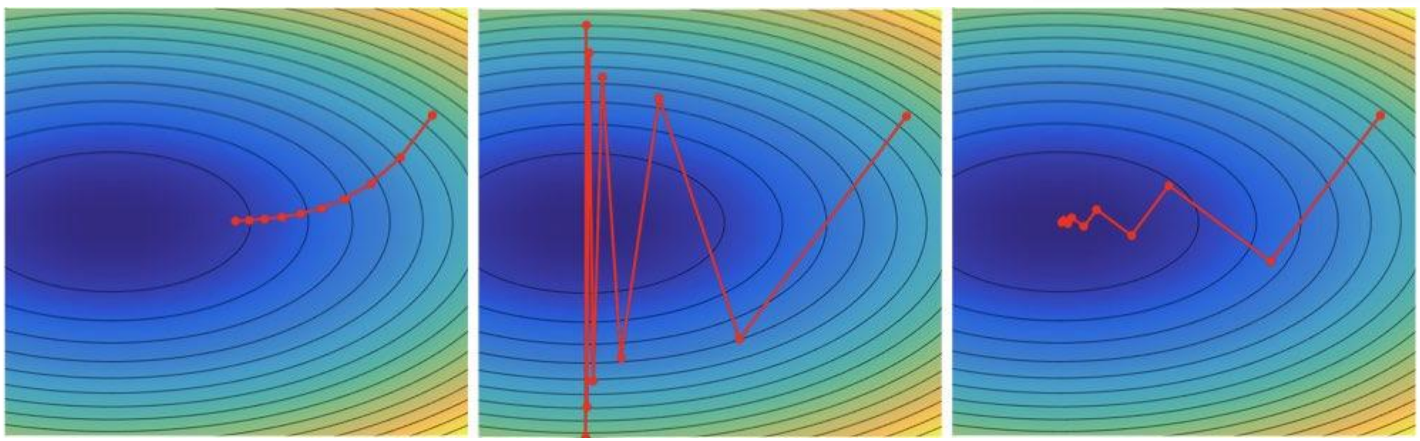
5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$



Animation by Andrew Ng

- The ***gradient-step*** ϵ, also called ***learning rate***, is a critical ***hyper-parameter*** of the algorithm.

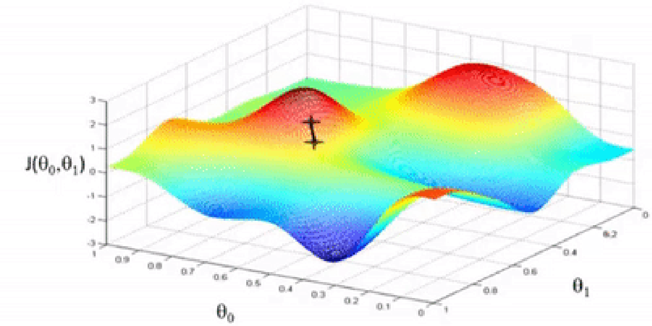


Small ϵ | Large ϵ | Optimal ϵ

Image by Gabriel Peyre

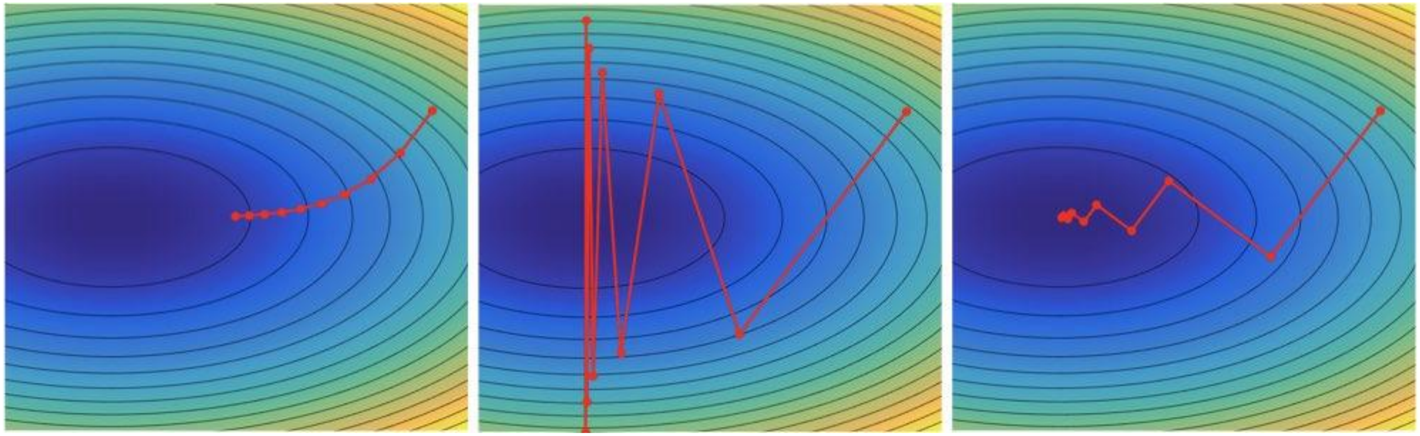
5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$



Animation by Andrew Ng

- The ***gradient-step*** ϵ, also called ***learning rate***, is a critical ***hyper-parameter*** of the algorithm.



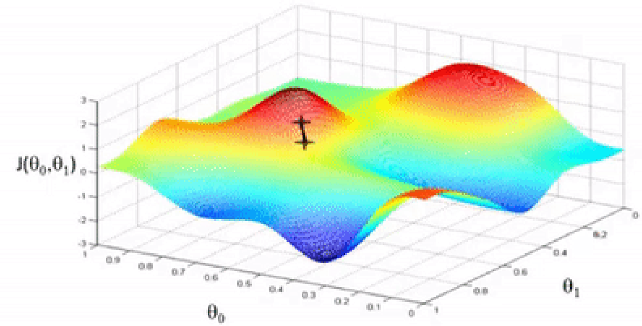
Small ϵ Large ϵ Optimal ϵ

Image by Gabriel Peyre

- Choosing a small ϵ is always the **safest**, but might result in **slow** convergence

5. Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$



Animation by Andrew Ng

- The ***gradient-step*** ϵ, also called ***learning rate***, is a critical ***hyper-parameter*** of the algorithm.

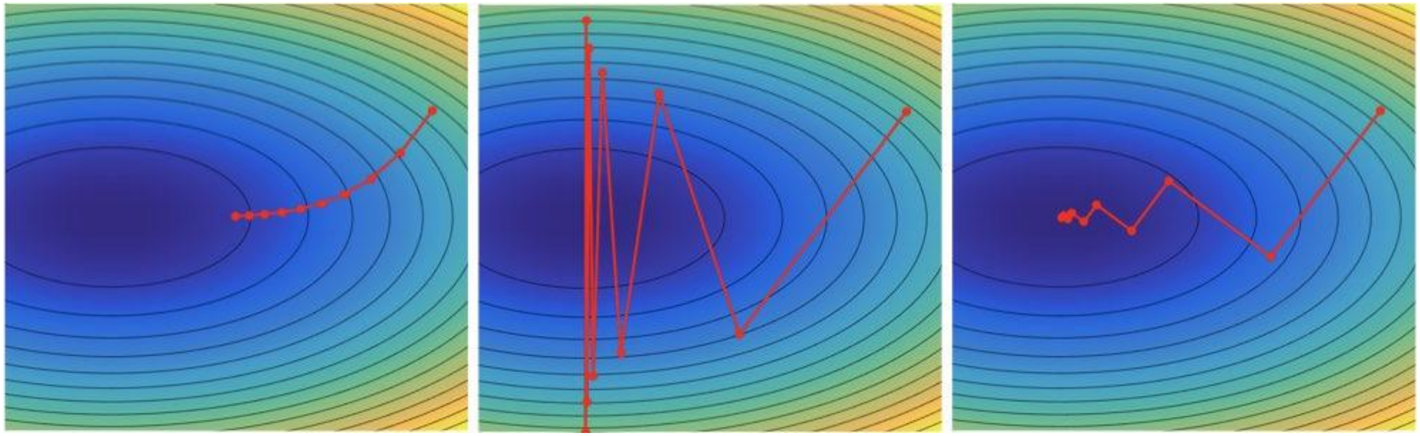

Small ϵ Large ϵ Optimal ϵ

Image by Gabriel Peyre

- Choosing a small ϵ is always the **safest**, but might result in **slow** convergence
- There exists many variations on gradient descent. We will cover some of them later in this chapter.

Summary of optimization techniques

Summary of optimization techniques

1. **Brute Force / Random / Grid Search** : Useful when searching among **discrete** parameters. Quickly **explodes** in complexity.

Summary of optimization techniques

1. **Brute Force / Random / Grid Search** : Useful when searching among **discrete** parameters. Quickly **explodes** in complexity.

2. **Population-Based algorithms** : Versatile but **heuristic**.

Summary of optimization techniques

1. **Brute Force / Random / Grid Search** : Useful when searching among **discrete** parameters. Quickly **explodes** in complexity.

2. **Population-Based algorithms** : Versatile but **heuristic**.

3. **Directly finding zeroes of the gradient** : Very efficient (**not** iterative) but only possible with a **limited** number of functions.

Summary of optimization techniques

1. **Brute Force / Random / Grid Search** : Useful when searching among **discrete** parameters. Quickly **explodes** in complexity.

2. **Population-Based algorithms** : Versatile but **heuristic**.

3. **Directly finding zeroes of the gradient** : Very efficient (**not** iterative) but only possible with a **limited** number of functions.

4. **Alternate minimization** : A **very general** family of principled methods. Allows **combining** multiple techniques. Often **no hyperparameters**. Needs to be designed on a **case-by-case** basis. Global convergence is generally **not guaranteed**.

Summary of optimization techniques

1. **Brute Force / Random / Grid Search** : Useful when searching among **discrete** parameters. Quickly **explodes** in complexity.

2. **Population-Based algorithms** : Versatile but **heuristic**.

3. **Directly finding zeroes of the gradient** : Very efficient (**not** iterative) but only possible with a **limited** number of functions.

4. **Alternate minimization** : A **very general** family of principled methods. Allows **combining** multiple techniques. Often **no hyperparameters**. Needs to be designed on a **case-by-case** basis. Global convergence is generally **not guaranteed**.

5. **Gradient descent** : Works on **any differentiable functions**. Convergence to **local minima**. The learning rate is a **critical hyperparameter**.

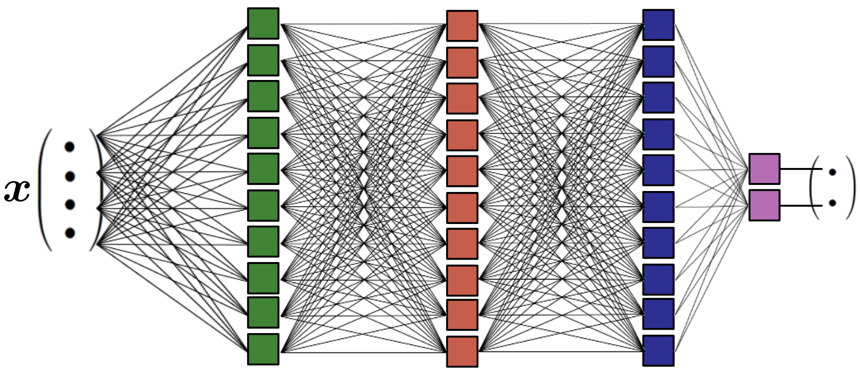
Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Remember: *A* ***deep feedforward neural network***

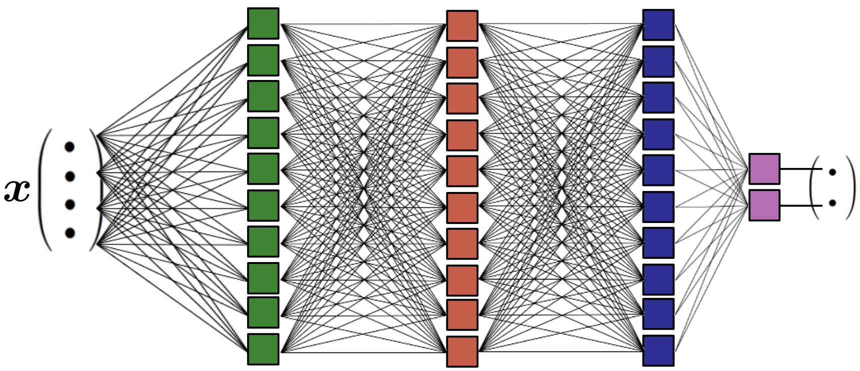


$\boldsymbol{y} = \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x})$

Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Remember: *A* ***deep feedforward neural network***



$$y = \mathrm{dnn}_{\theta}(x)$$

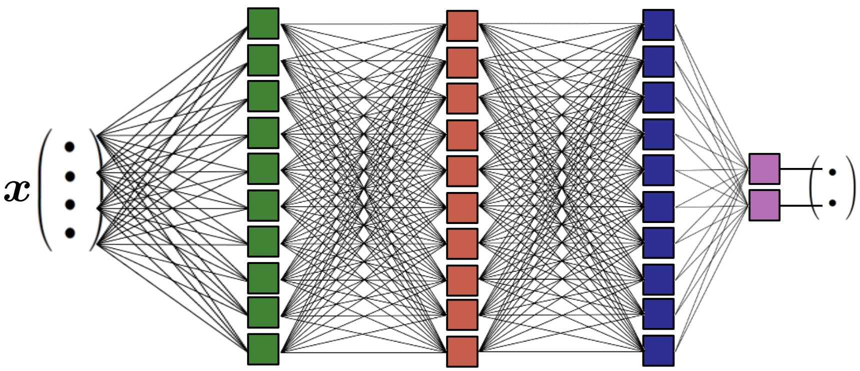
$$= \sigma\left(b^4 + \mathbf{W}^4\sigma\left(b^3 + \mathbf{W}^3\sigma\left(b^2 + \mathbf{W}^2\sigma\left(b^1 + \mathbf{W}^1 x\right)\right)\right)\right)$$

$$\theta = [\theta^i]_{i=1}^4 = [(b^i, \mathbf{W}^i)]_{i=1}^4$$

Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Remember: *A **deep feedforward neural network***



$$\boldsymbol{y} = \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x})$$

$$= \sigma\left(\boldsymbol{b}^4 + \mathbf{W}^4 \sigma\left(\boldsymbol{b}^3 + \mathbf{W}^3 \sigma\left(\boldsymbol{b}^2 + \mathbf{W}^2 \sigma\left(\boldsymbol{b}^1 + \mathbf{W}^1 \boldsymbol{x}\right)\right)\right)\right)$$

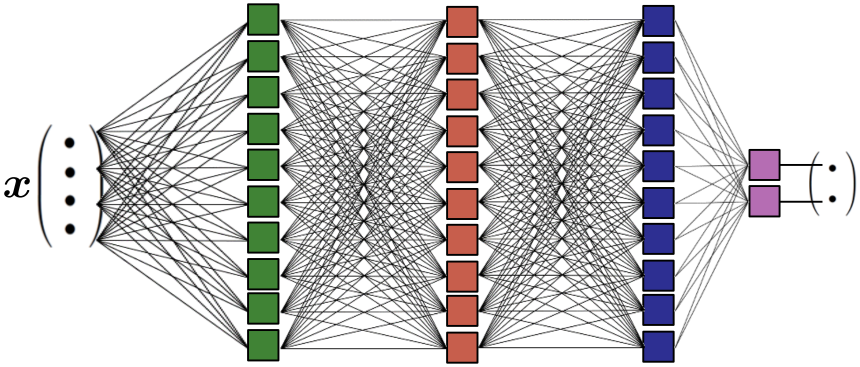
$$\boldsymbol{\theta} = [\boldsymbol{\theta}^i]_{i=1}^4 = [(\boldsymbol{b}^i, \mathbf{W}^i)]_{i=1}^4$$

- Given a **training dataset** of *input* ↔ *output* $\mathcal{T} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^T$, the general goal is to adjust $\boldsymbol{\theta}$ so that $\boldsymbol{y}_t \approx \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t)$.

Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Remember: *A **deep feedforward neural network***



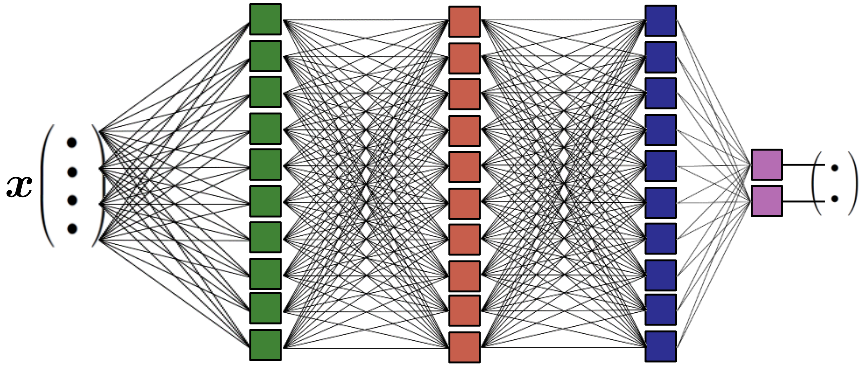
$$\boldsymbol{y} = \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x})$$
$$= \sigma\left(\boldsymbol{b}^4 + \mathbf{W}^4\sigma\left(\boldsymbol{b}^3 + \mathbf{W}^3\sigma\left(\boldsymbol{b}^2 + \mathbf{W}^2\sigma\left(\boldsymbol{b}^1 + \mathbf{W}^1\boldsymbol{x}\right)\right)\right)\right)$$
$$\boldsymbol{\theta} = [\boldsymbol{\theta}^i]_{i=1}^4 = [(\boldsymbol{b}^i, \mathbf{W}^i)]_{i=1}^4$$

- Given a **training dataset** of *input* ↔ *output* $\mathcal{T} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^T$, the general goal is to adjust $\boldsymbol{\theta}$ so that $\boldsymbol{y}_t \approx \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t)$.
- We use a **total loss** of this form: $L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \ell\left(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t\right)$, where ℓ is simply called ***the loss*** of the DNN.

Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Remember: *A* ***deep feedforward neural network***



$$\boldsymbol{y} = \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x})$$
$$= \sigma\left(\boldsymbol{b}^4 + \mathbf{W}^4\sigma\left(\boldsymbol{b}^3 + \mathbf{W}^3\sigma\left(\boldsymbol{b}^2 + \mathbf{W}^2\sigma\left(\boldsymbol{b}^1 + \mathbf{W}^1\boldsymbol{x}\right)\right)\right)\right)$$
$$\boldsymbol{\theta} = [\boldsymbol{\theta}^i]_{i=1}^4 = [(\boldsymbol{b}^i, \mathbf{W}^i)]_{i=1}^4$$

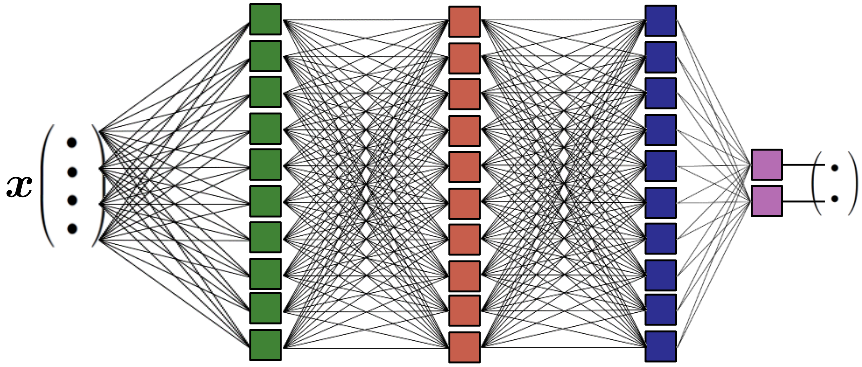
- Given a **training dataset** of *input* ↔ *output* $\mathcal{T} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^T$, the general goal is to adjust $\boldsymbol{\theta}$ so that $\boldsymbol{y}_t \approx \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t)$.
- We use a **total loss** of this form: $L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T}\ell\left(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t\right)$, where ℓ is simply called ***the loss*** of the DNN.

  For example: $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$ *, the so called "L2 loss" or "Euclidean loss".*

Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Remember: *A* ***deep feedforward neural network***



$$\boldsymbol{y} = \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x})$$
$$= \sigma\left(\boldsymbol{b}^4 + \mathbf{W}^4\sigma\left(\boldsymbol{b}^3 + \mathbf{W}^3\sigma\left(\boldsymbol{b}^2 + \mathbf{W}^2\sigma\left(\boldsymbol{b}^1 + \mathbf{W}^1\boldsymbol{x}\right)\right)\right)\right)$$
$$\boldsymbol{\theta} = [\boldsymbol{\theta}^i]_{i=1}^4 = [(\boldsymbol{b}^i, \mathbf{W}^i)]_{i=1}^4$$

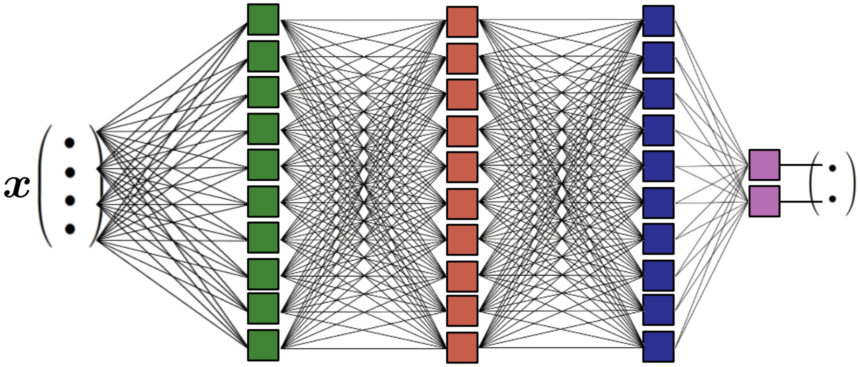
- Given a **training dataset** of *input* ↔ *output* $\mathcal{T} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^T$, the general goal is to adjust $\boldsymbol{\theta}$ so that $\boldsymbol{y}_t \approx \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t)$.
- We use a **total loss** of this form: $L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \ell\left(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t\right)$, where ℓ is simply called ***the loss*** of the DNN.

  For example: $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$ *, the so called "L2 loss" or "Euclidean loss".*
- Losses of the form L are called ***Empirical Risk***, where the ***Risk*** of the model is defined as $\mathcal{R}(\mathrm{dnn}_{\boldsymbol{\theta}}) \stackrel{\text{def}}{=} \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\{\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{Y})\} \approx L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T})$.

Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Remember: *A **deep feedforward neural network***



$$\boldsymbol{y} = \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x})$$
$$= \sigma\left(\boldsymbol{b}^4 + \mathbf{W}^4\sigma\left(\boldsymbol{b}^3 + \mathbf{W}^3\sigma\left(\boldsymbol{b}^2 + \mathbf{W}^2\sigma\left(\boldsymbol{b}^1 + \mathbf{W}^1\boldsymbol{x}\right)\right)\right)\right)$$
$$\boldsymbol{\theta} = [\boldsymbol{\theta}^i]_{i=1}^4 = [(\boldsymbol{b}^i, \mathbf{W}^i)]_{i=1}^4$$

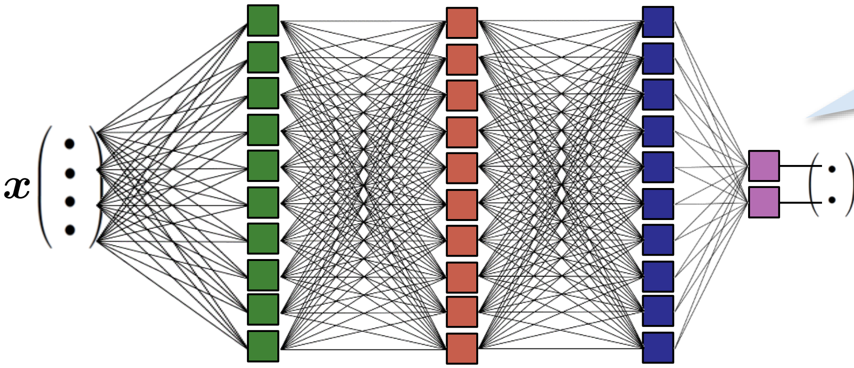
- Given a **training dataset** of *input* ↔ *output* $\mathcal{T} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^T$, the general goal is to adjust $\boldsymbol{\theta}$ so that $\boldsymbol{y}_t \approx \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t)$.
- We use a **total loss** of this form: $L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \ell\left(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t\right)$, where ℓ is simply called ***the loss*** of the DNN.

  For example: $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$*, the so called "L2 loss" or "Euclidean loss".*
- Losses of the form L are called ***Empirical Risk***, where the ***Risk*** of the model is defined as $\mathcal{R}(\mathrm{dnn}_{\boldsymbol{\theta}}) \stackrel{\mathrm{def}}{=} \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\{\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{Y})\} \approx L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T})$.
- We will focus next on **supervised learning**, but the approach is more general.

Back to Neural Networks

- Neural network models are **fitted** using variants of **gradient descent**.

Remember: *A **deep feedforward neural network***



How to compute $\nabla_{\boldsymbol{\theta}} L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T})$ *?!*

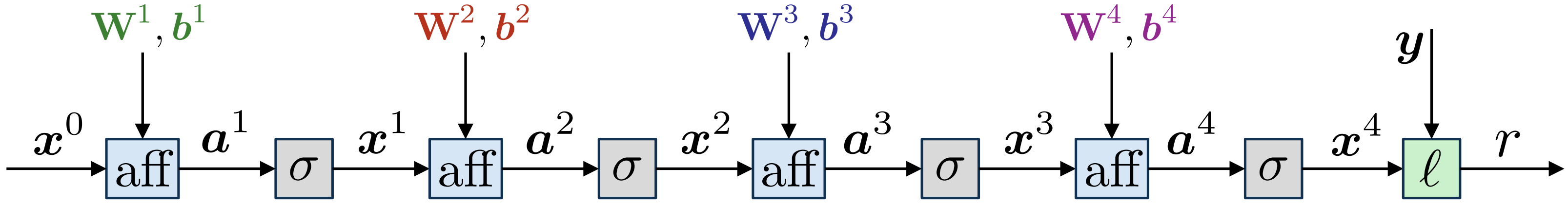
$$\boldsymbol{y} = \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x})$$
$$= \sigma\left(\boldsymbol{b}^4 + \mathbf{W}^4\sigma\left(\boldsymbol{b}^3 + \mathbf{W}^3\sigma\left(\boldsymbol{b}^2 + \mathbf{W}^2\sigma\left(\boldsymbol{b}^1 + \mathbf{W}^1\boldsymbol{x}\right)\right)\right)\right)$$
$$\boldsymbol{\theta} = [\boldsymbol{\theta}^i]_{i=1}^4 = [(\boldsymbol{b}^i, \mathbf{W}^i)]_{i=1}^4$$

- Given a **training dataset** of *input* ↔ *output* $\mathcal{T} = \{(\boldsymbol{x}_t, \boldsymbol{y}_t)\}_{t=1}^T$, the general goal is to adjust $\boldsymbol{\theta}$ so that $\boldsymbol{y}_t \approx \mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t)$.
- We use a **total loss** of this form: $L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \ell\left(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t\right)$, where ℓ is simply called ***the loss*** of the DNN.

  For example: $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$ *, the so called "L2 loss" or "Euclidean loss".*
- Losses of the form L are called ***Empirical Risk***, where the ***Risk*** of the model is defined as $\mathcal{R}(\mathrm{dnn}_{\boldsymbol{\theta}}) \stackrel{\text{def}}{=} \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\{\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{Y})\} \approx L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T})$.
- We will focus next on **supervised learning**, but the approach is more general.

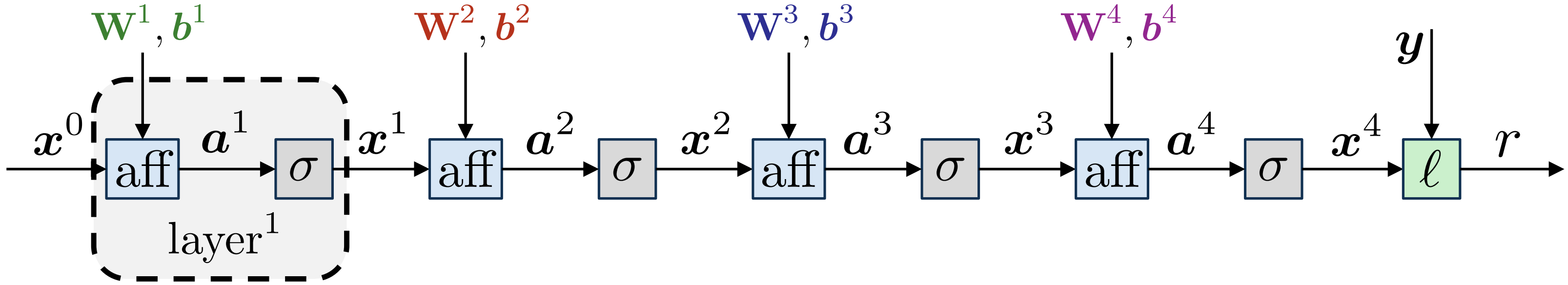
The Backpropagation Algorithm

Let's start by **cleaning** a bit the picture:



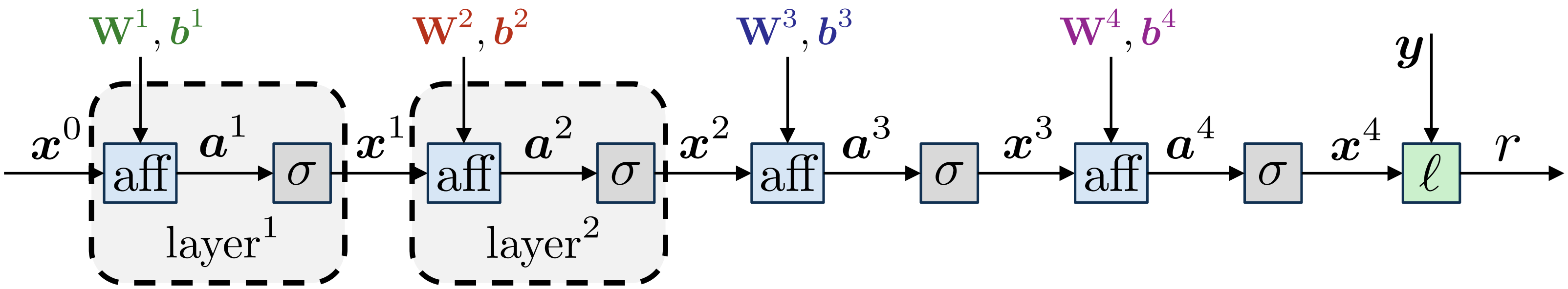
The Backpropagation Algorithm

Let's start by **cleaning** a bit the picture:



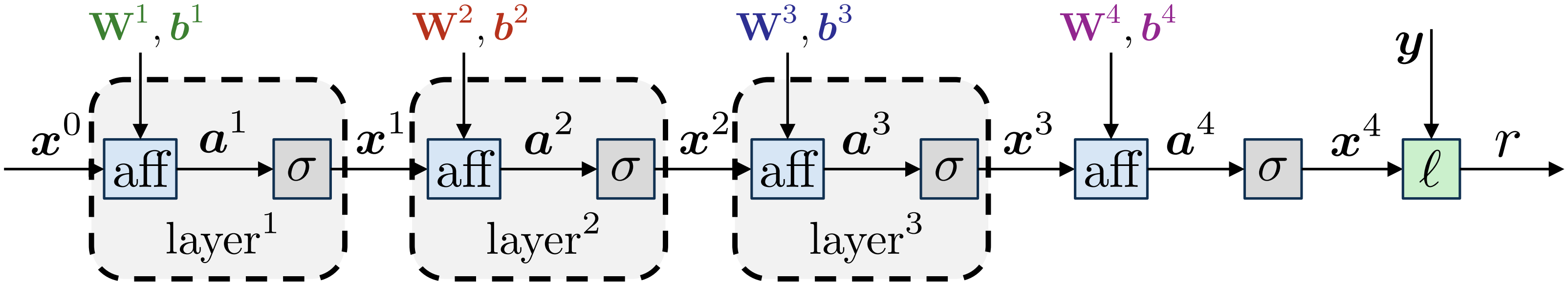
The Backpropagation Algorithm

Let's start by **cleaning** a bit the picture:



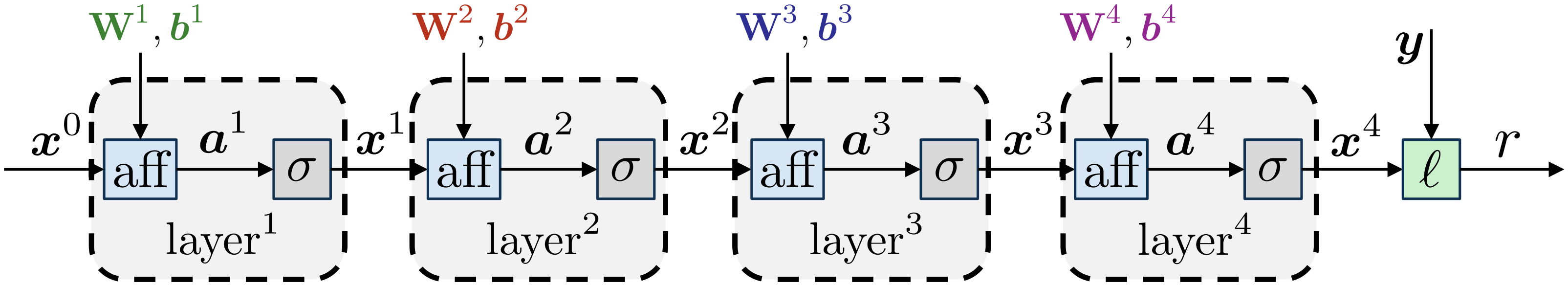
The Backpropagation Algorithm

Let's start by **cleaning** a bit the picture:



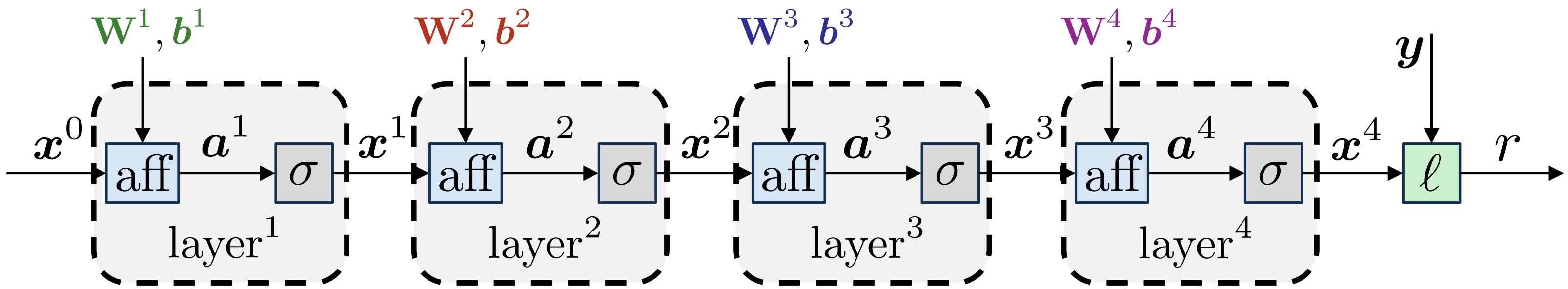
The Backpropagation Algorithm

Let's start by **cleaning** a bit the picture:



The Backpropagation Algorithm

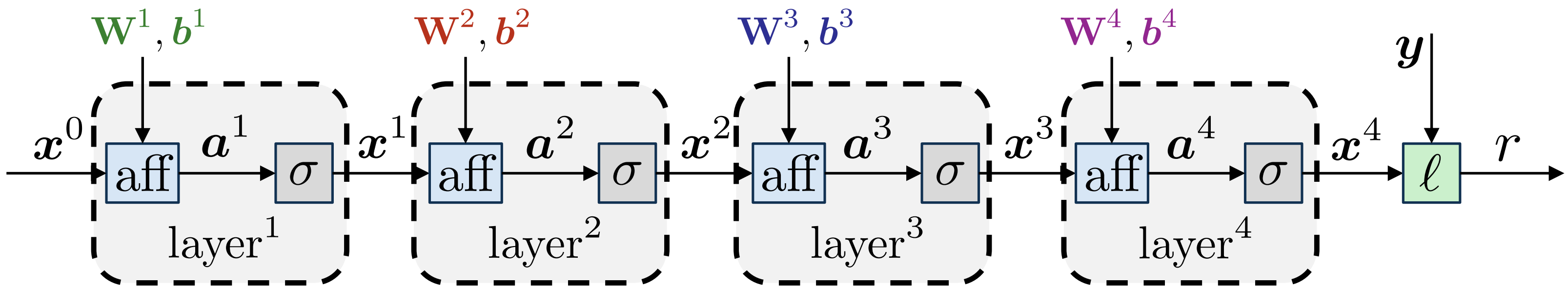
Let's start by **cleaning** a bit the picture:



- $\boldsymbol{a}^i = \mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1}) = \mathbf{W}^i\boldsymbol{x}^{i-1} + \boldsymbol{b}^i$ are the **pre-activations**.

The Backpropagation Algorithm

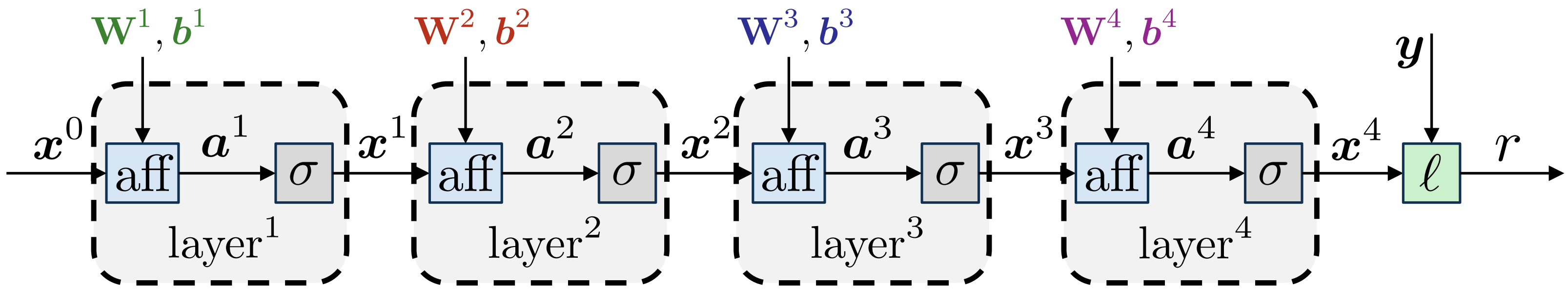
Let's start by **cleaning** a bit the picture:



- $\boldsymbol{a}^i = \text{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1}) = \mathbf{W}^i \boldsymbol{x}^{i-1} + \boldsymbol{b}^i$ are the **pre-activations**.
- $\boldsymbol{x}^i = \sigma(\boldsymbol{a}^i) = \sigma(\text{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1})) = \text{layer}^i(\boldsymbol{x}^{i-1})$ are the **activations**.

The Backpropagation Algorithm

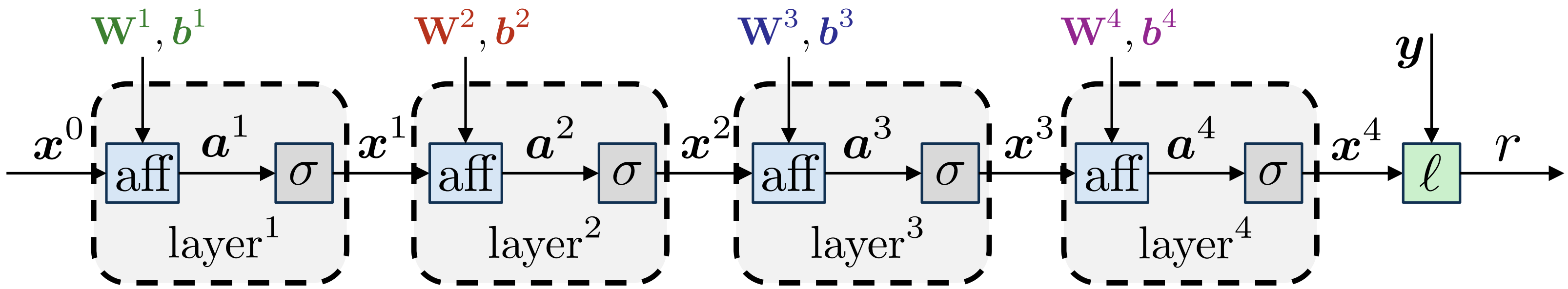
Let's start by **cleaning** a bit the picture:



- $\boldsymbol{a}^i = \mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1}) = \mathbf{W}^i\boldsymbol{x}^{i-1} + \boldsymbol{b}^i$ are the **pre-activations**.
- $\boldsymbol{x}^i = \sigma(\boldsymbol{a}^i) = \sigma(\mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1})) = \mathrm{layer}^i(\boldsymbol{x}^{i-1})$ are the **activations**.
- The **loss** ℓ can be viewed as another layer, with **real output** r (the *"residual"*).

The Backpropagation Algorithm

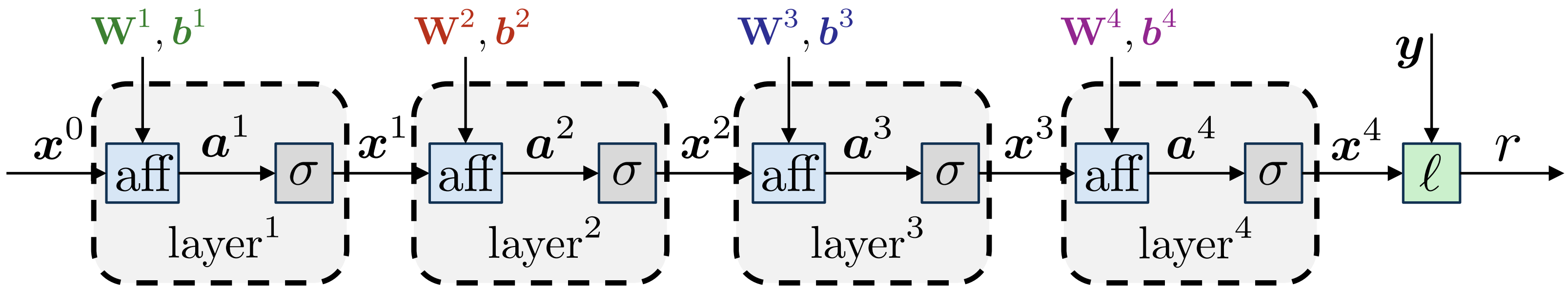
Let's start by **cleaning** a bit the picture:



- $\boldsymbol{a}^i = \mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1}) = \mathbf{W}^i\boldsymbol{x}^{i-1} + \boldsymbol{b}^i$ are the **pre-activations**.
- $\boldsymbol{x}^i = \sigma(\boldsymbol{a}^i) = \sigma(\mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1})) = \mathrm{layer}^i(\boldsymbol{x}^{i-1})$ are the **activations**.
- The **loss** ℓ can be viewed as another layer, with **real output** r (the *"residual"*).
- By **linearity** of the gradient, we have: $\nabla_{\boldsymbol{\theta}} L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$.

The Backpropagation Algorithm

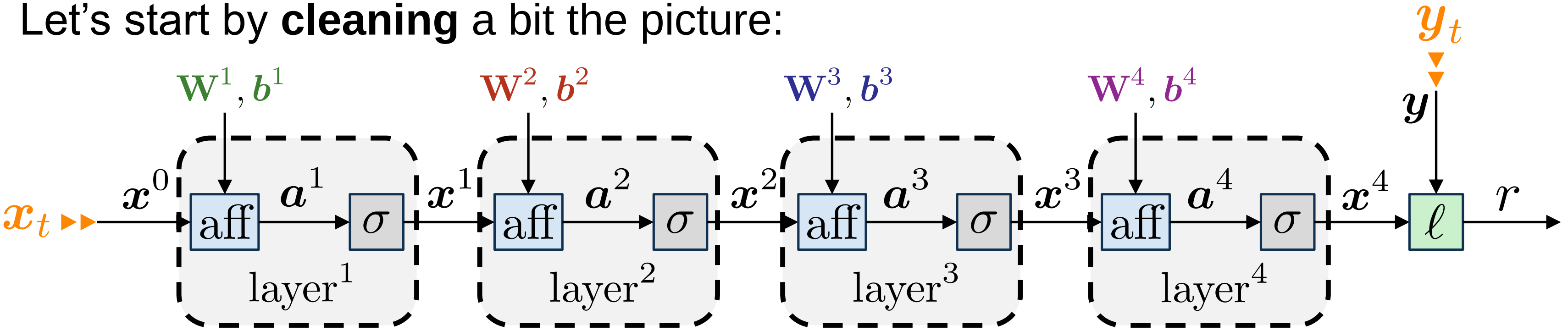
Let's start by **cleaning** a bit the picture:



- $\boldsymbol{a}^i = \mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1}) = \mathbf{W}^i\boldsymbol{x}^{i-1} + \boldsymbol{b}^i$ are the **pre-activations**.
- $\boldsymbol{x}^i = \sigma(\boldsymbol{a}^i) = \sigma(\mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1})) = \mathrm{layer}^i(\boldsymbol{x}^{i-1})$ are the **activations**.
- The **loss** ℓ can be viewed as another layer, with **real output** r (the *"residual"*).
- By **linearity** of the gradient, we have: $\nabla_{\boldsymbol{\theta}} L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}}\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$.
- Hence, it is enough to calculate the gradient of the loss for **one sample** $(\boldsymbol{x}_t, \boldsymbol{y}_t)$, i.e., $\boldsymbol{G}_{\boldsymbol{\theta}} \stackrel{\mathrm{def}}{=} \nabla_{\boldsymbol{\theta}}\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$.

The Backpropagation Algorithm

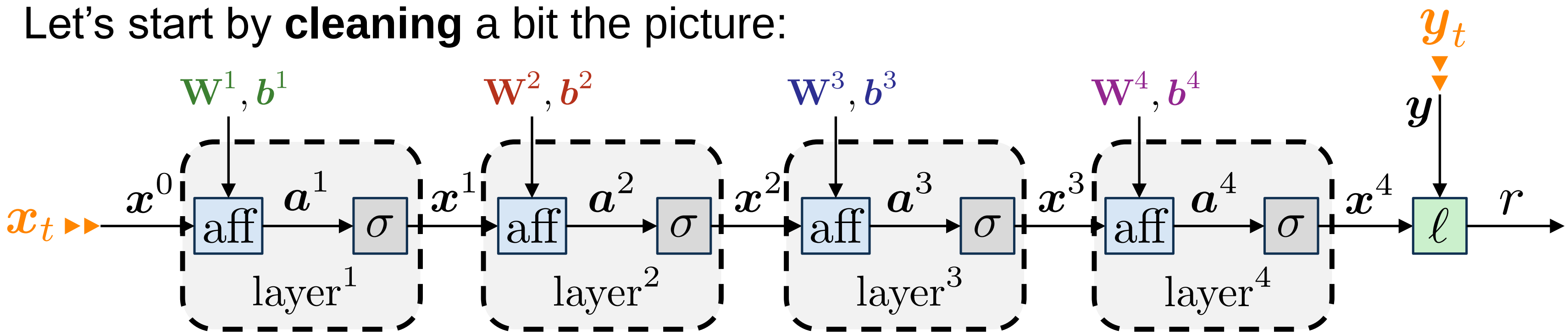
Let's start by **cleaning** a bit the picture:



- $\boldsymbol{a}^i = \mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1}) = \mathbf{W}^i\boldsymbol{x}^{i-1} + \boldsymbol{b}^i$ are the **pre-activations**.
- $\boldsymbol{x}^i = \sigma(\boldsymbol{a}^i) = \sigma(\mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1})) = \mathrm{layer}^i(\boldsymbol{x}^{i-1})$ are the **activations**.
- The **loss** ℓ can be viewed as another layer, with **real output** r (the *"residual"*).
- By **linearity** of the gradient, we have: $\nabla_{\boldsymbol{\theta}} L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$.
- Hence, it is enough to calculate the gradient of the loss for **one sample** $(\boldsymbol{x}_t, \boldsymbol{y}_t)$, i.e., $\boldsymbol{G}_{\boldsymbol{\theta}} \overset{\mathrm{def}}{=} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$.

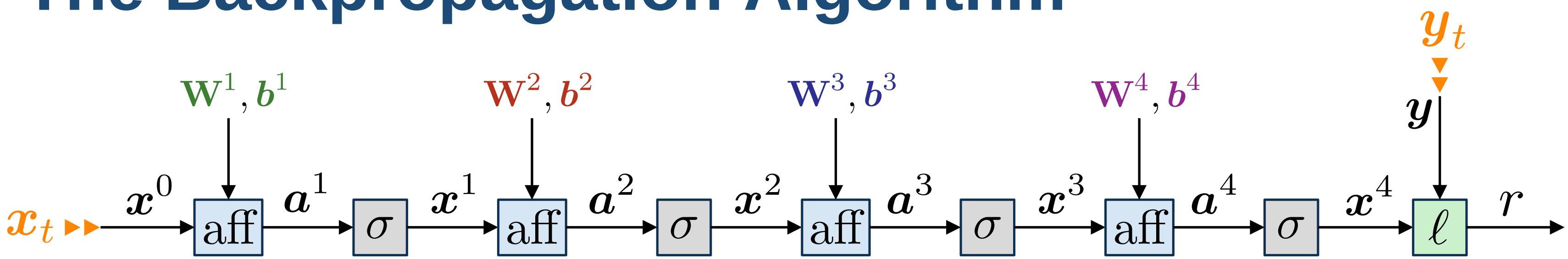
The Backpropagation Algorithm

Let's start by **cleaning** a bit the picture:



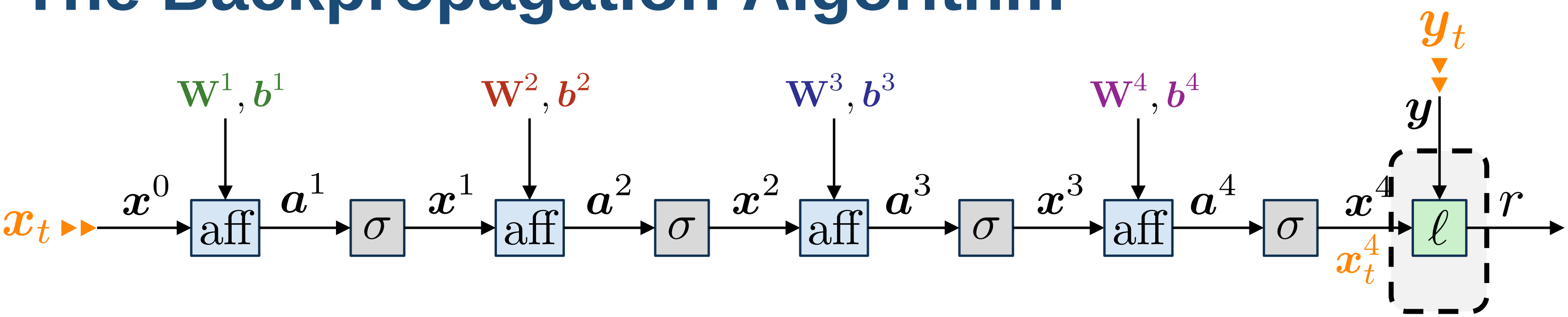
- $\boldsymbol{a}^i = \mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1}) = \mathbf{W}^i \boldsymbol{x}^{i-1} + \boldsymbol{b}^i$ are the **pre-activations**.
- $\boldsymbol{x}^i = \sigma(\boldsymbol{a}^i) = \sigma(\mathrm{aff}_{\boldsymbol{\theta}^i}(\boldsymbol{x}^{i-1})) = \mathrm{layer}^i(\boldsymbol{x}^{i-1})$ are the **activations**.
- The **loss** ℓ can be viewed as another layer, with **real output** r (the *"residual"*).
- By **linearity** of the gradient, we have: $\nabla_{\boldsymbol{\theta}} L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$.
- Hence, it is enough to calculate the gradient of the loss for **one sample** $(\boldsymbol{x}_t, \boldsymbol{y}_t)$, i.e., $\boldsymbol{G}_{\boldsymbol{\theta}} \stackrel{\mathrm{def}}{=} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$.
- The **Backpropagation Algorithm** (*"Backprop"*) is an efficient way to do this.

The Backpropagation Algorithm



*The trick is to **recursively calculate** the gradient of the **loss** with respect to both the **parameters** and **activations**, going **backwards** from the end, using the **chain rule**.*

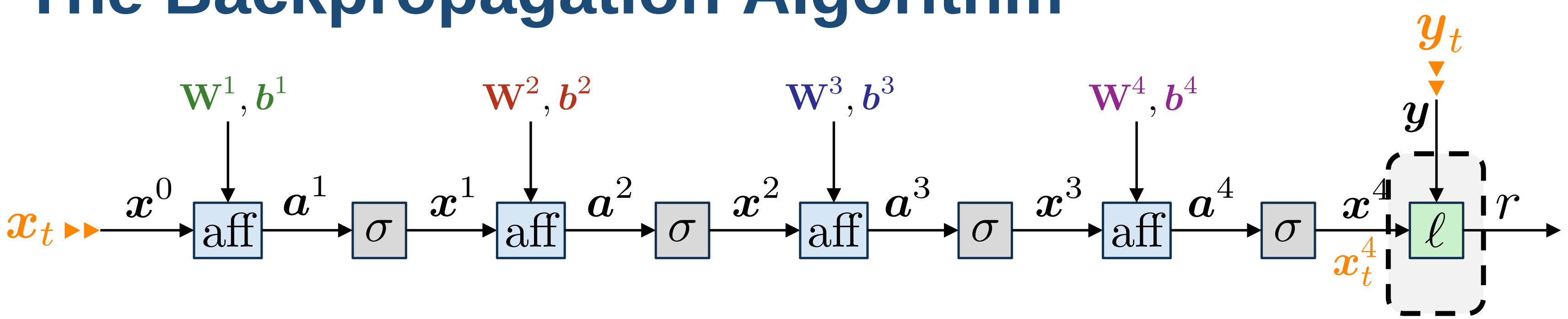
The Backpropagation Algorithm



The trick is to ***recursively calculate*** *the gradient of the* ***loss*** *with respect to both the* ***parameters*** *and* ***activations****, going* ***backwards*** *from the end, using the* ***chain rule****.*

0) We start by $\boldsymbol{G}_{\boldsymbol{x}^4} = \left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4}^\top$

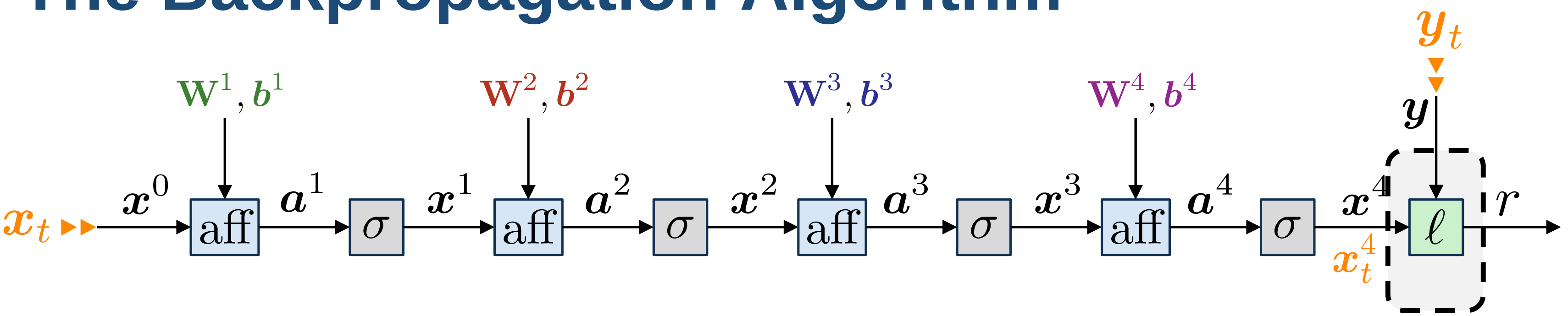
The Backpropagation Algorithm



The trick is to ***recursively calculate*** *the gradient of the* ***loss*** *with respect to both the* ***parameters*** *and* ***activations****, going* ***backwards*** *from the end, using the* ***chain rule****.*

0) We start by $\boldsymbol{G}_{\boldsymbol{x}^4} = \left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4}^\top = \nabla_{\boldsymbol{x}^4} \ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t)$.

The Backpropagation Algorithm

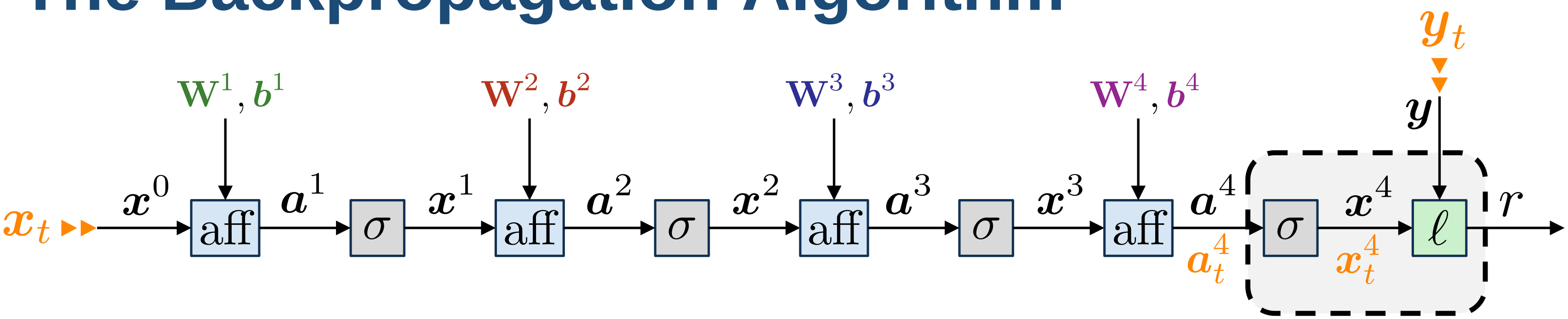


The trick is to ***recursively calculate*** *the gradient of the* ***loss*** *with respect to both the* ***parameters*** *and* ***activations****, going* ***backwards*** *from the end, using the* ***chain rule****.*

0) We start by $\boldsymbol{G}_{\boldsymbol{x}^4} = \left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4}^{\top} = \nabla_{\boldsymbol{x}^4} \ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t)$.

For example, for the L2 loss $\ell(\boldsymbol{x}^4, \boldsymbol{y}_t) = \|\boldsymbol{x}^4 - \boldsymbol{y}_t\|_2^2$, we have $\boldsymbol{G}_{\boldsymbol{x}^4} = 2(\boldsymbol{x}_t^4 - \boldsymbol{y}_t)$, the **difference** between the network **prediction** and the **target**.

The Backpropagation Algorithm



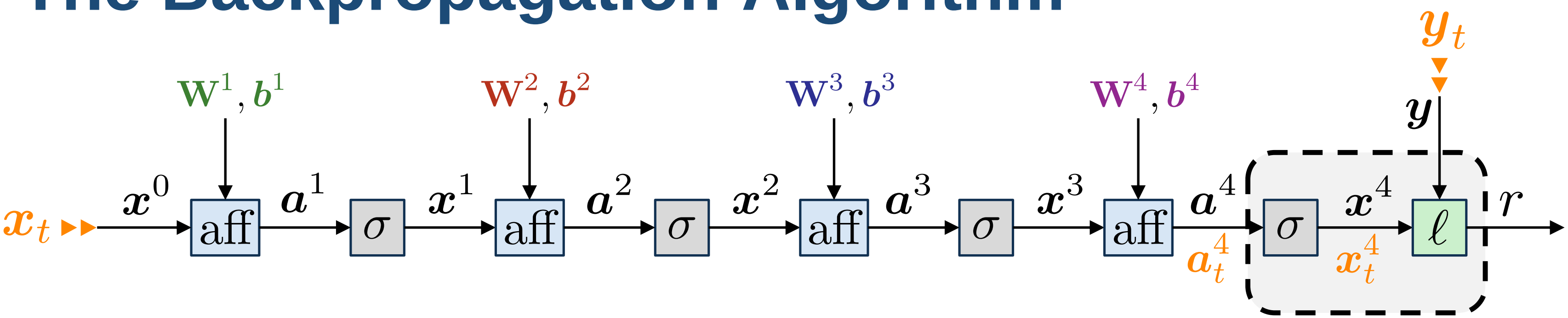
*The trick is to **recursively calculate** the gradient of the **loss** with respect to both the **parameters** and **activations**, going **backwards** from the end, using the **chain rule**.*

0) We start by $\boldsymbol{G}_{\boldsymbol{x}^4} = \left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4}^\top = \nabla_{\boldsymbol{x}^4}\ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t)$.

For example, for the L2 loss $\ell(\boldsymbol{x}^4, \boldsymbol{y}_t) = \|\boldsymbol{x}^4 - \boldsymbol{y}_t\|_2^2$, we have $\boldsymbol{G}_{\boldsymbol{x}^4} = 2(\boldsymbol{x}_t^4 - \boldsymbol{y}_t)$, the **difference** between the network **prediction** and the **target**.

1) Then, $\boldsymbol{G}_{\boldsymbol{a}^4} = \left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}^\top$

The Backpropagation Algorithm



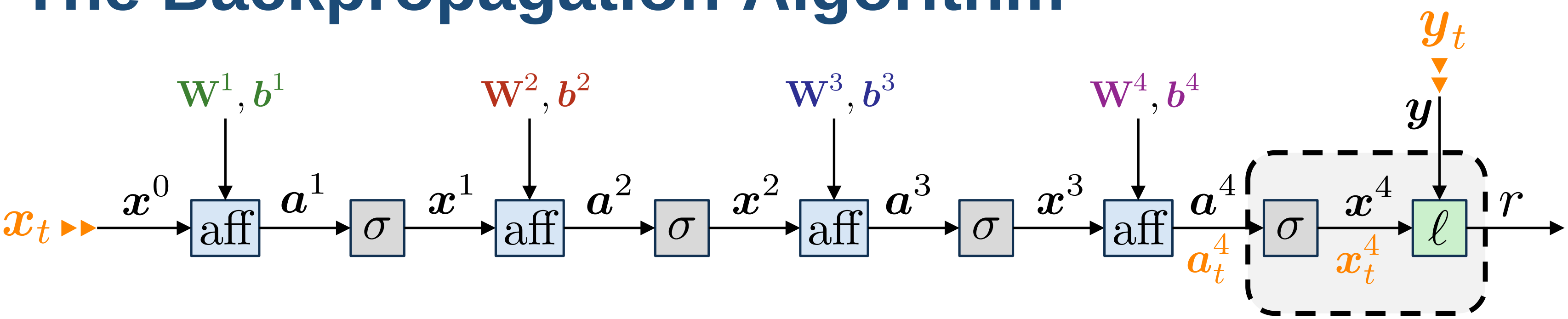
*The trick is to **recursively calculate** the gradient of the **loss** with respect to both the **parameters** and **activations**, going **backwards** from the end, using the **chain rule**.*

0) We start by $\boldsymbol{G}_{\boldsymbol{x}^4} = \left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4}^\top = \nabla_{\boldsymbol{x}^4} \ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t)$.

For example, for the L2 loss $\ell(\boldsymbol{x}^4, \boldsymbol{y}_t) = \|\boldsymbol{x}^4 - \boldsymbol{y}_t\|_2^2$, we have $\boldsymbol{G}_{\boldsymbol{x}^4} = 2(\boldsymbol{x}_t^4 - \boldsymbol{y}_t)$, the **difference** between the network **prediction** and the **target**.

1) Then, $\boldsymbol{G}_{\boldsymbol{a}^4} = \left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4} \times \left.\frac{\partial \boldsymbol{x}^4}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \right)^\top$

The Backpropagation Algorithm



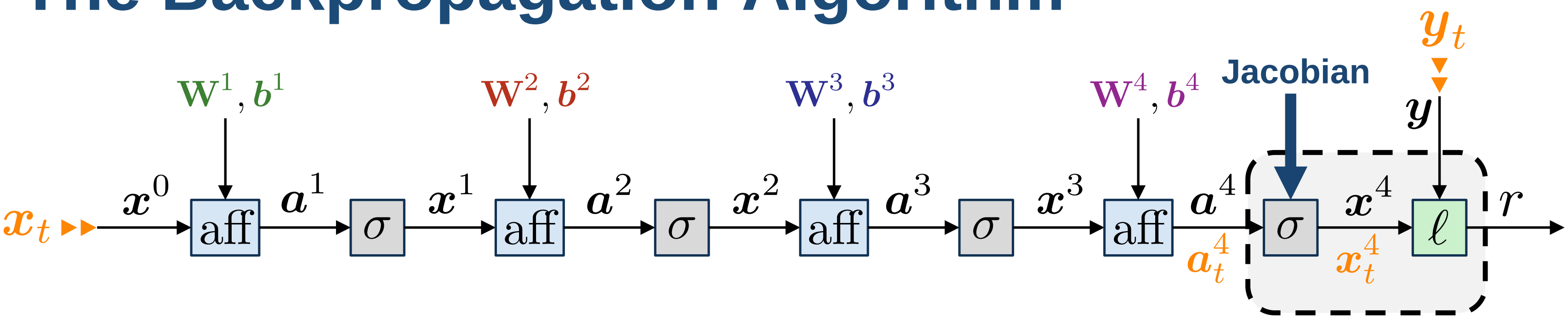
The trick is to ***recursively calculate*** *the gradient of the* ***loss*** *with respect to both the* ***parameters*** *and* ***activations****, going* ***backwards*** *from the end, using the* ***chain rule****.*

0) We start by $\boldsymbol{G}_{\boldsymbol{x}^4} = \left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4}^\top = \nabla_{\boldsymbol{x}^4} \ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t)$.

For example, for the L2 loss $\ell(\boldsymbol{x}^4, \boldsymbol{y}_t) = \|\boldsymbol{x}^4 - \boldsymbol{y}_t\|_2^2$, we have $\boldsymbol{G}_{\boldsymbol{x}^4} = 2(\boldsymbol{x}_t^4 - \boldsymbol{y}_t)$, the **difference** between the network **prediction** and the **target**.

1) Then, $\boldsymbol{G}_{\boldsymbol{a}^4} = \left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4} \times \left.\frac{\partial \boldsymbol{x}^4}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}\right)^\top = \left.\frac{\partial \boldsymbol{x}^4}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}^\top \times \boldsymbol{G}_{\boldsymbol{x}^4}$

The Backpropagation Algorithm



*The trick is to **recursively calculate** the gradient of the **loss** with respect to both the **parameters** and **activations**, going **backwards** from the end, using the **chain rule**.*

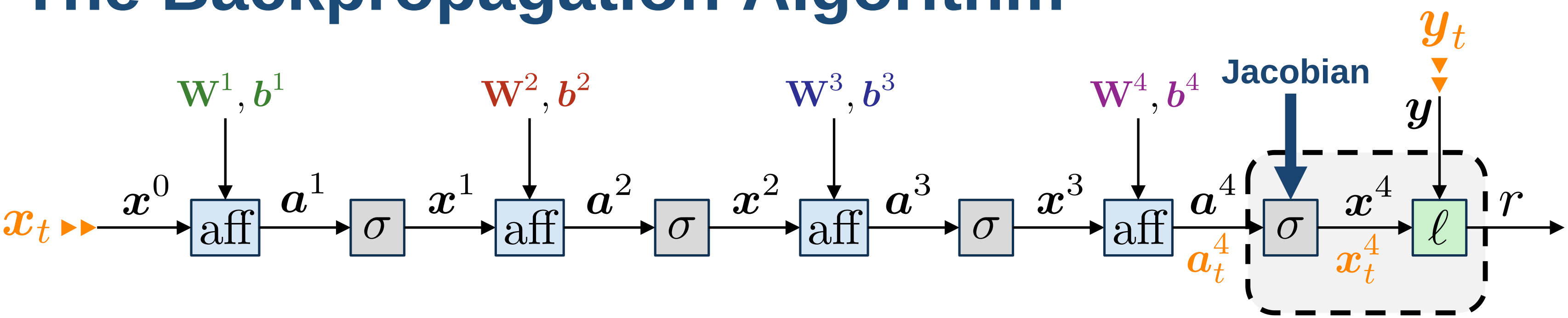
0) We start by $\boldsymbol{G}_{\boldsymbol{x}^4} = \left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4}^\top = \nabla_{\boldsymbol{x}^4}\ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t)$.

For example, for the L2 loss $\ell(\boldsymbol{x}^4, \boldsymbol{y}_t) = \|\boldsymbol{x}^4 - \boldsymbol{y}_t\|_2^2$, we have $\boldsymbol{G}_{\boldsymbol{x}^4} = 2(\boldsymbol{x}_t^4 - \boldsymbol{y}_t)$, the **difference** between the network **prediction** and the **target**.

1) Then, $\boldsymbol{G}_{\boldsymbol{a}^4} = \left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4} \times \left.\frac{\partial \boldsymbol{x}^4}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}\right)^\top = \left.\frac{\partial \boldsymbol{x}^4}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}^\top \times \boldsymbol{G}_{\boldsymbol{x}^4}$

$\rightarrow \mathrm{diag}[\sigma'(\boldsymbol{a}_t^4)]$!

The Backpropagation Algorithm



The trick is to ***recursively calculate*** *the gradient of the* ***loss*** *with respect to both the* ***parameters*** *and* ***activations****, going* ***backwards*** *from the end, using the* ***chain rule****.*

0) We start by $\boldsymbol{G}_{\boldsymbol{x}^4} = \left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4}^\top = \nabla_{\boldsymbol{x}^4} \ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t)$.

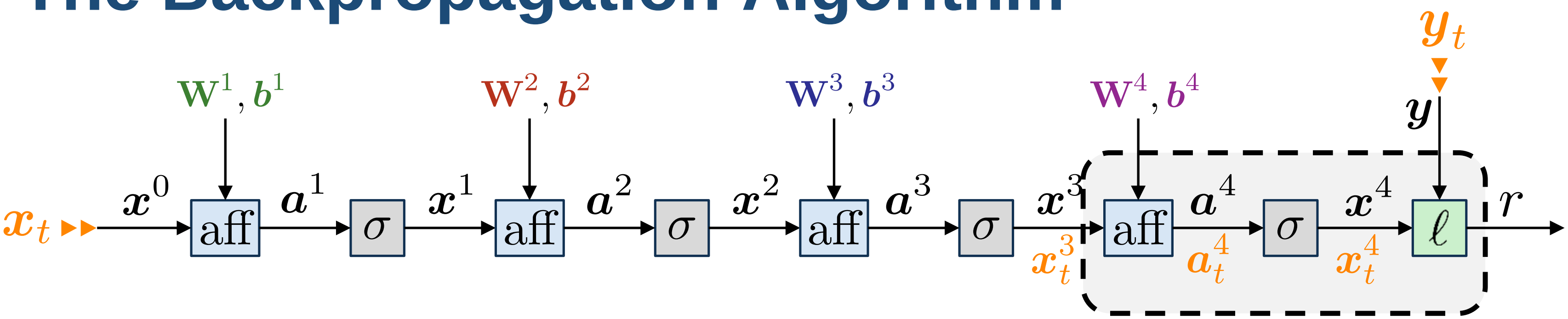
For example, for the L2 loss $\ell(\boldsymbol{x}^4, \boldsymbol{y}_t) = \|\boldsymbol{x}^4 - \boldsymbol{y}_t\|_2^2$, we have $\boldsymbol{G}_{\boldsymbol{x}^4} = 2(\boldsymbol{x}_t^4 - \boldsymbol{y}_t)$, the **difference** between the network **prediction** and the **target**.

1) Then, $\boldsymbol{G}_{\boldsymbol{a}^4} = \left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{x}^4}\right|_{\boldsymbol{x}_t^4} \times \left.\frac{\partial \boldsymbol{x}^4}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \right)^\top = \left.\frac{\partial \boldsymbol{x}^4}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4}^\top \times \boldsymbol{G}_{\boldsymbol{x}^4}$

$= \sigma'(\boldsymbol{a}_t^4) \odot \boldsymbol{G}_{\boldsymbol{x}^4}$.

$\mathrm{diag}[\sigma'(\boldsymbol{a}_t^4)]$!

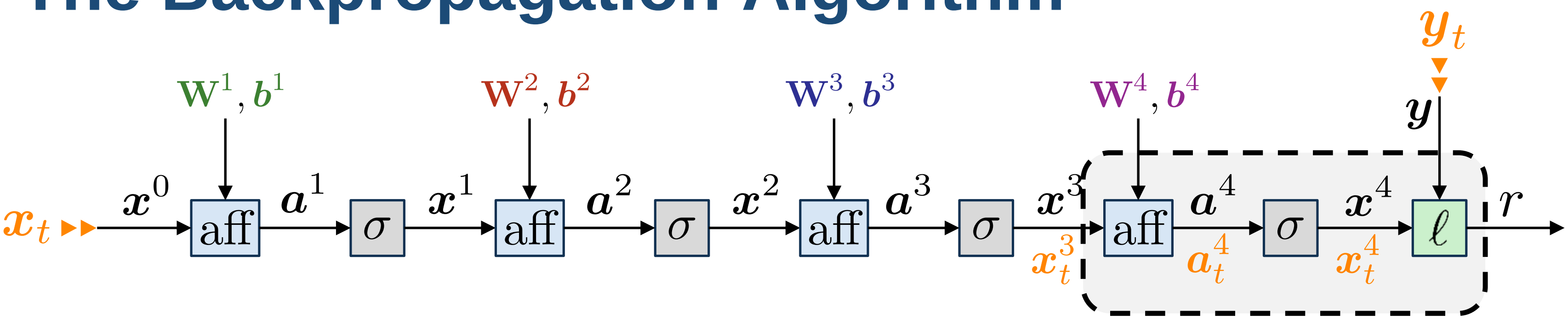
The Backpropagation Algorithm



2) Then:

- $\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top}$

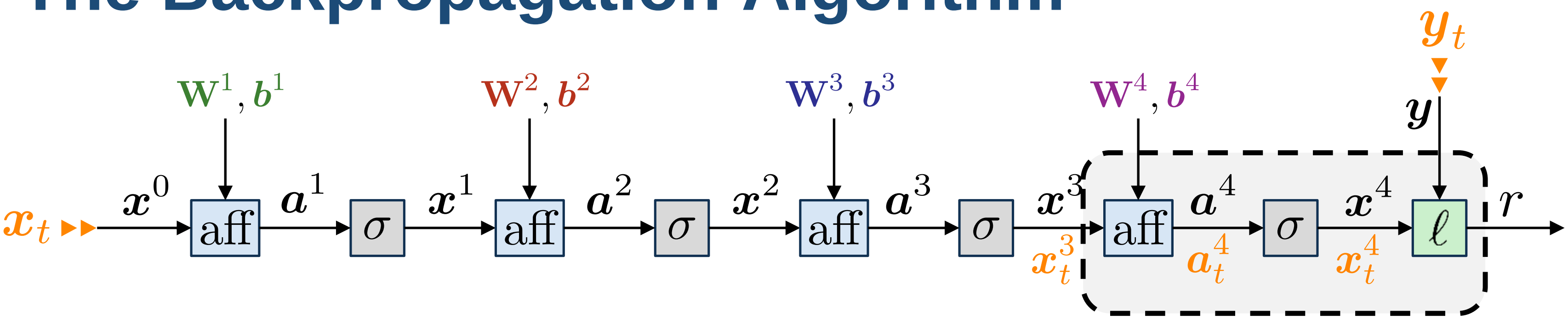
The Backpropagation Algorithm



2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3} \right)^{\top}$$

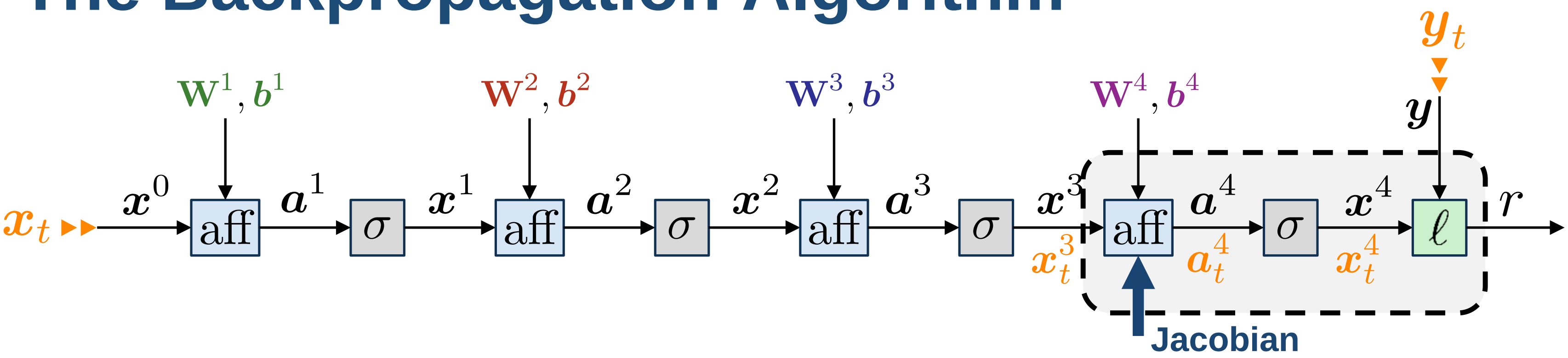
The Backpropagation Algorithm



2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3} \right)^{\top} = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4}$$

The Backpropagation Algorithm

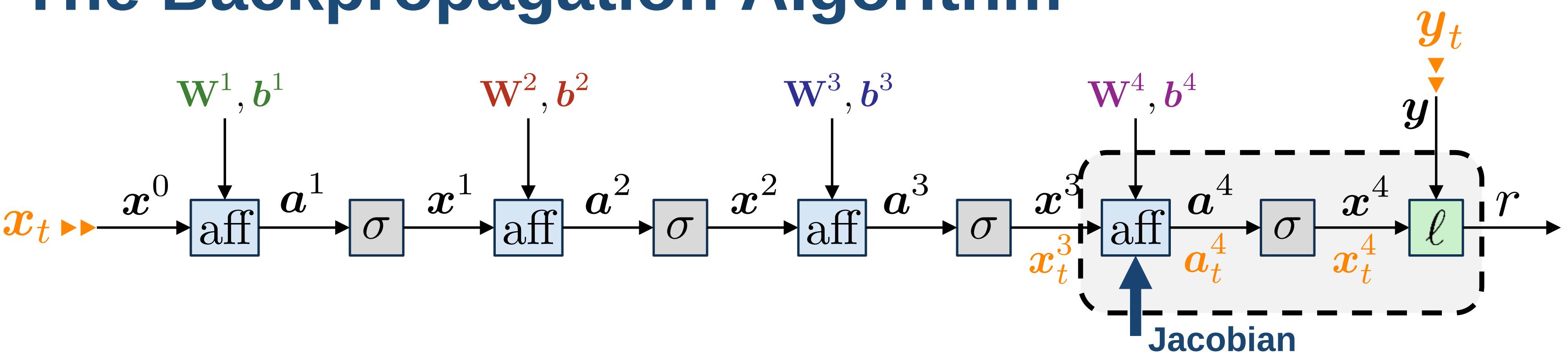


2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3} \right)^{\top} = \boxed{\left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top}} \times \boldsymbol{G}_{\boldsymbol{a}^4}$$

$\to \mathbf{W}^4$

The Backpropagation Algorithm

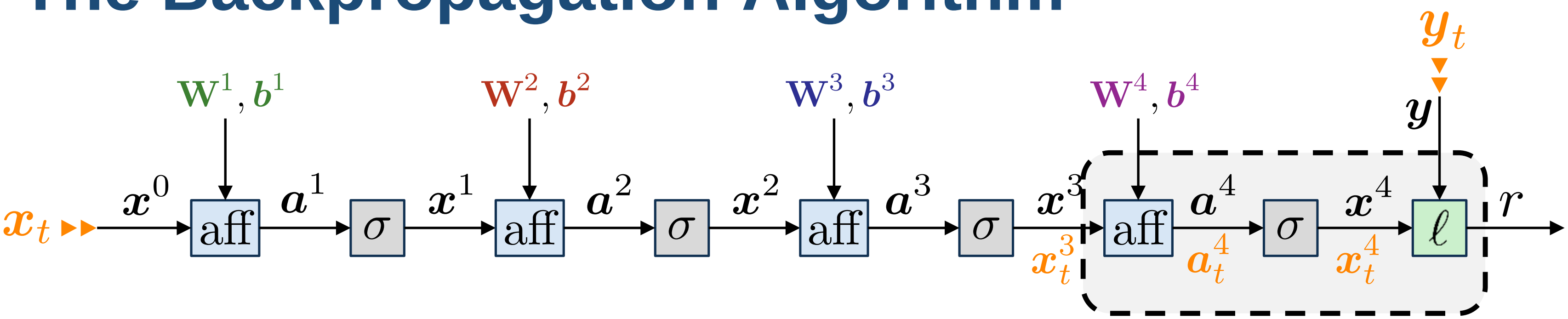


2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3} \right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4} = \mathbf{W}^{4\top} \boldsymbol{G}_{\boldsymbol{a}^4}.$$

$\mathbf{W}^4$

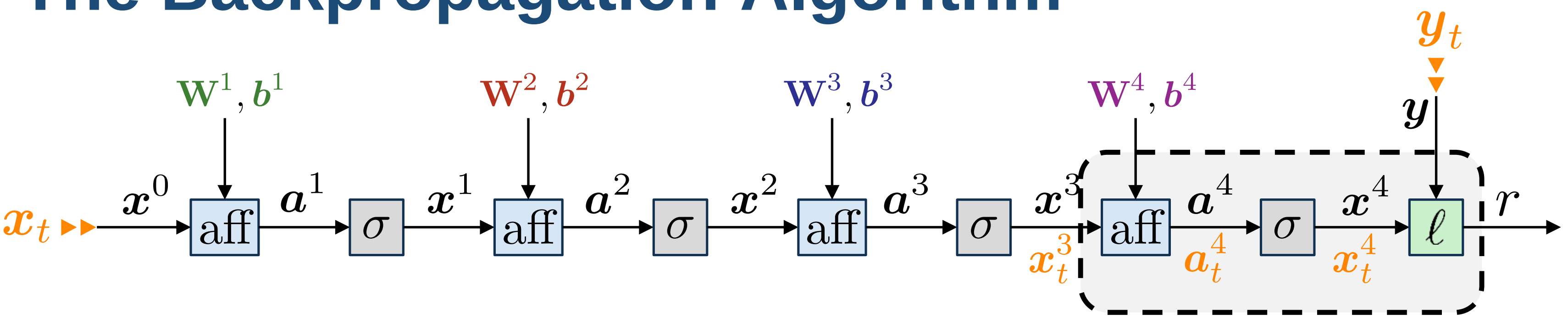
The Backpropagation Algorithm



2) Then:

- $\boldsymbol{G}_{\boldsymbol{x}^3} = \frac{\partial r}{\partial \boldsymbol{x}^3}\Big|_{\boldsymbol{x}_t^3}^{\top} = \left(\frac{\partial r}{\partial \boldsymbol{a}^4}\Big|_{\boldsymbol{a}_t^4} \times \frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\Big|_{\boldsymbol{x}_t^3}\right)^{\top} = \frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\Big|_{\boldsymbol{x}_t^3}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \mathbf{W}^{4\top}\boldsymbol{G}_{\boldsymbol{a}^4}$.
- $\boldsymbol{G}_{\boldsymbol{b}^4} = \frac{\partial r}{\partial \boldsymbol{b}^4}\Big|_{\boldsymbol{b}^4}^{\top}$

The Backpropagation Algorithm

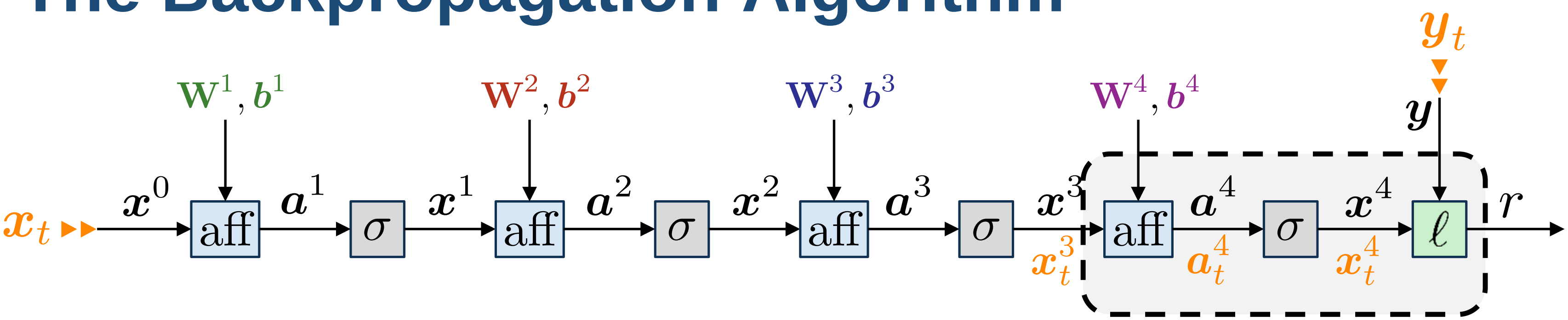


2) Then:

- $$\mathbf{G}_{\mathbf{x}^3} = \left.\frac{\partial r}{\partial \mathbf{x}^3}\right|_{\mathbf{x}_t^3}^{\top} = \left(\left.\frac{\partial r}{\partial \mathbf{a}^4}\right|_{\mathbf{a}_t^4} \times \left.\frac{\partial \mathbf{a}^4}{\partial \mathbf{x}^3}\right|_{\mathbf{x}_t^3} \right)^{\top} = \left.\frac{\partial \mathbf{a}^4}{\partial \mathbf{x}^3}\right|_{\mathbf{x}_t^3}^{\top} \times \mathbf{G}_{\mathbf{a}^4} = \boxed{\mathbf{W}^{4\top} \mathbf{G}_{\mathbf{a}^4}}\,.$$

- $$\mathbf{G}_{\mathbf{b}^4} = \left.\frac{\partial r}{\partial \mathbf{b}^4}\right|_{\mathbf{b}^4}^{\top} = \left(\left.\frac{\partial r}{\partial \mathbf{a}^4}\right|_{\mathbf{a}_t^4} \times \left.\frac{\partial \mathbf{a}^4}{\partial \mathbf{b}^4}\right|_{\mathbf{b}^4} \right)^{\top}$$

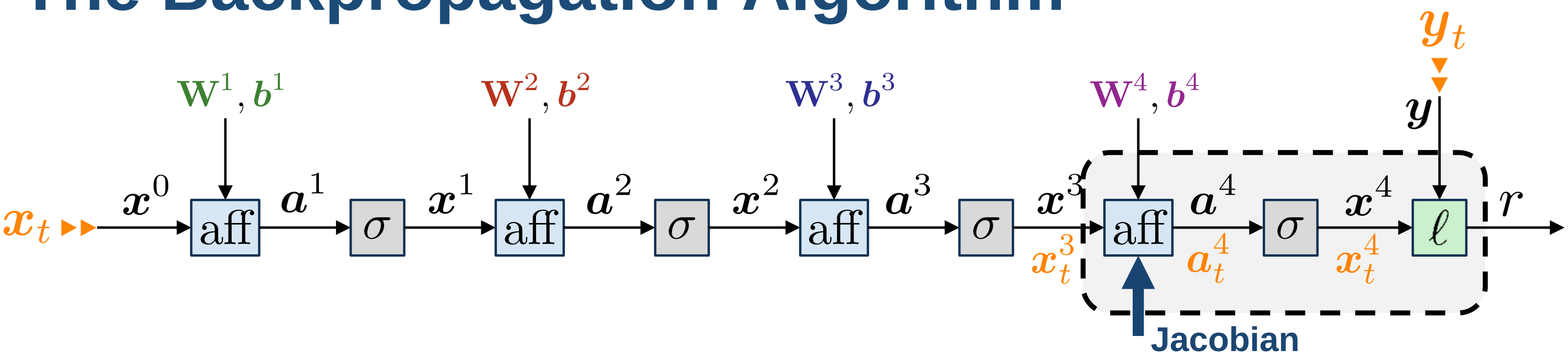
The Backpropagation Algorithm



2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}\right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boxed{\mathbf{W}^{4\top}\boldsymbol{G}_{\boldsymbol{a}^4}}.$$
- $$\boldsymbol{G}_{\boldsymbol{b}^4} = \left.\frac{\partial r}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}\right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4}$$

The Backpropagation Algorithm



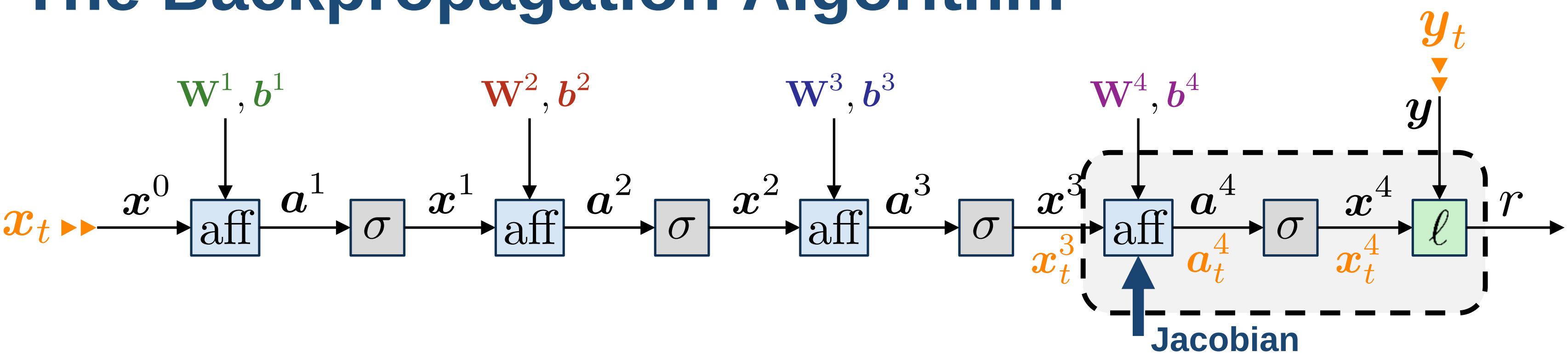
2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3} \right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4} = \mathbf{W}^{4\top} \boldsymbol{G}_{\boldsymbol{a}^4}.$$

- $$\boldsymbol{G}_{\boldsymbol{b}^4} = \left.\frac{\partial r}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4} \right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4}$$

  $\left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top \rightarrow \mathbf{I}$

The Backpropagation Algorithm

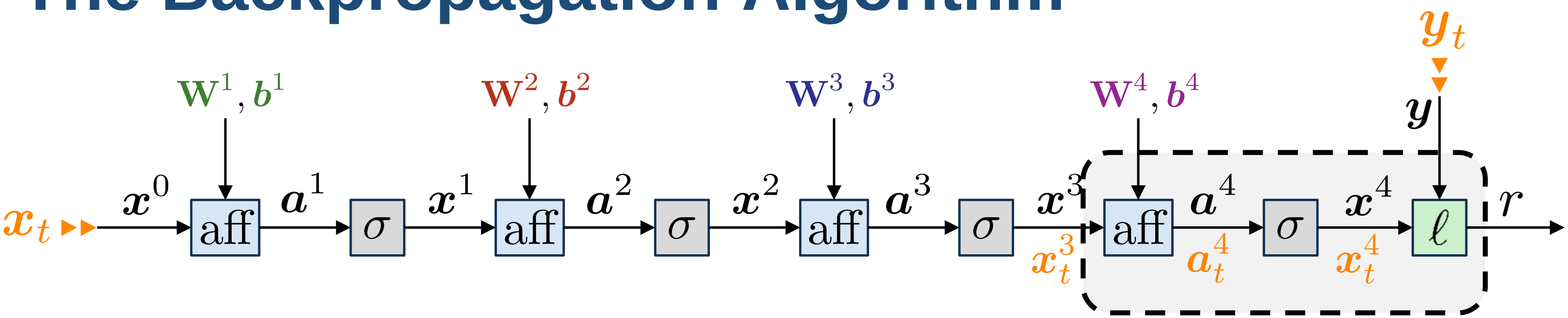


2) Then:

- $\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3} \right)^{\top} = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \mathbf{W}^{4\top} \boldsymbol{G}_{\boldsymbol{a}^4}\,.$
- $\boldsymbol{G}_{\boldsymbol{b}^4} = \left.\frac{\partial r}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4} \right)^{\top} = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boldsymbol{G}_{\boldsymbol{a}^4}\,.$

$\rightarrow \mathbf{I}$

The Backpropagation Algorithm



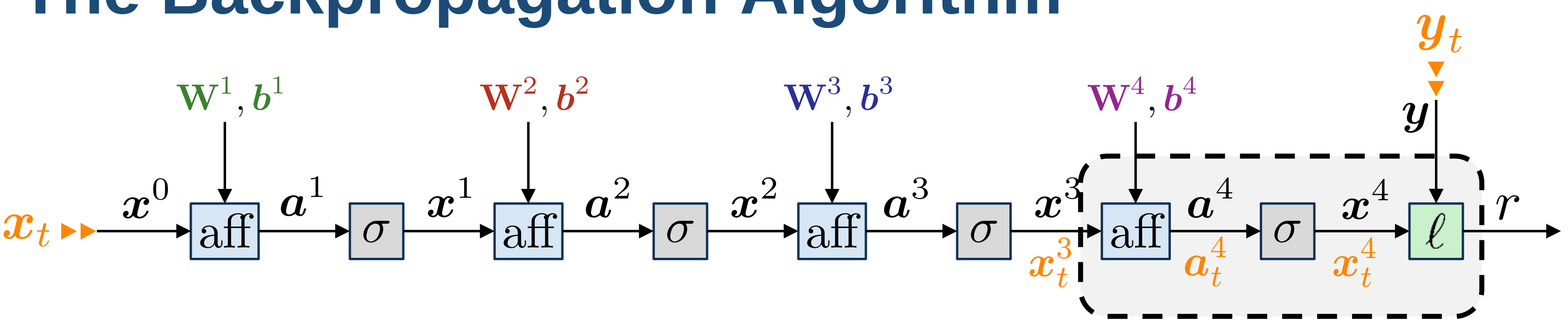
2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3} \right)^{\top} = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \mathbf{W}^{4\top} \boldsymbol{G}_{\boldsymbol{a}^4} .$$

- $$\boldsymbol{G}_{\boldsymbol{b}^4} = \left.\frac{\partial r}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4} \right)^{\top} = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boldsymbol{G}_{\boldsymbol{a}^4} .$$

- $$\boldsymbol{G}_{\boldsymbol{w}_d^4} = \left.\frac{\partial r}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}^{\top}$$

The Backpropagation Algorithm



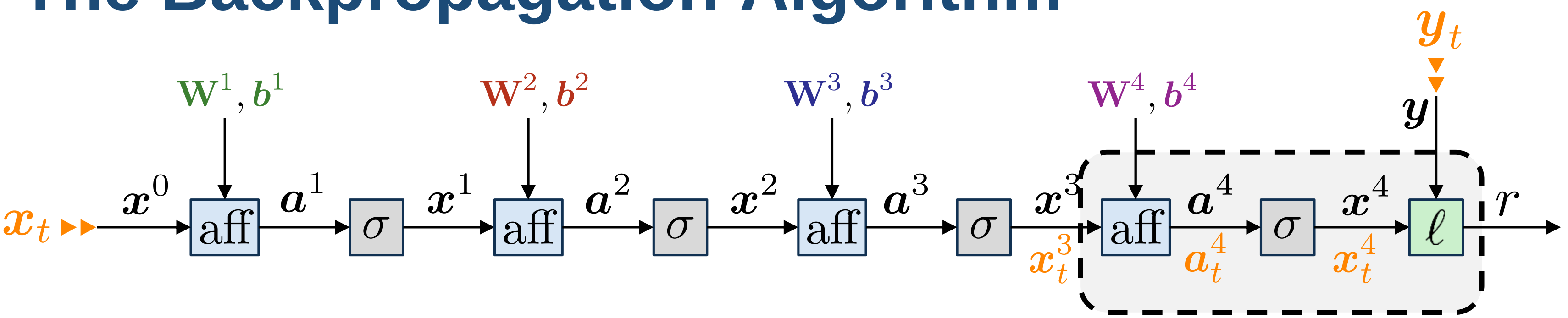
2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \frac{\partial r}{\partial \boldsymbol{x}^3}\bigg|_{\boldsymbol{x}_t^3}^{\top} = \left(\frac{\partial r}{\partial \boldsymbol{a}^4}\bigg|_{\boldsymbol{a}_t^4} \times \frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\bigg|_{\boldsymbol{x}_t^3} \right)^{\top} = \frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\bigg|_{\boldsymbol{x}_t^3}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boxed{\mathbf{W}^{4\top}\boldsymbol{G}_{\boldsymbol{a}^4}}.$$

- $$\boldsymbol{G}_{\boldsymbol{b}^4} = \frac{\partial r}{\partial \boldsymbol{b}^4}\bigg|_{\boldsymbol{b}^4}^{\top} = \left(\frac{\partial r}{\partial \boldsymbol{a}^4}\bigg|_{\boldsymbol{a}_t^4} \times \frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\bigg|_{\boldsymbol{b}^4} \right)^{\top} = \frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\bigg|_{\boldsymbol{b}^4}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boxed{\boldsymbol{G}_{\boldsymbol{a}^4}}.$$

- $$\boldsymbol{G}_{\boldsymbol{w}_d^4} = \frac{\partial r}{\partial \boldsymbol{w}_d^4}\bigg|_{\boldsymbol{w}_d^4}^{\top} = \left(\frac{\partial r}{\partial \boldsymbol{a}^4}\bigg|_{\boldsymbol{a}_t^4} \times \frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{w}_d^4}\bigg|_{\boldsymbol{w}_d^4} \right)^{\top}$$

The Backpropagation Algorithm



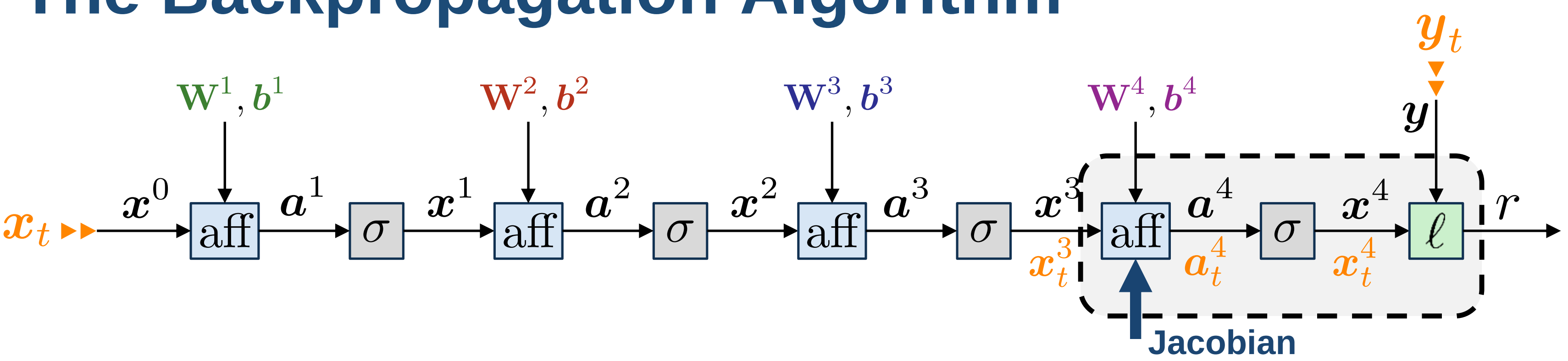
2) Then:

- $\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}\right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boxed{\mathbf{W}^{4\top}\boldsymbol{G}_{\boldsymbol{a}^4}}.$

- $\boldsymbol{G}_{\boldsymbol{b}^4} = \left.\frac{\partial r}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}\right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boxed{\boldsymbol{G}_{\boldsymbol{a}^4}}.$

- $\boldsymbol{G}_{\boldsymbol{w}_d^4} = \left.\frac{\partial r}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}\right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4}$

The Backpropagation Algorithm

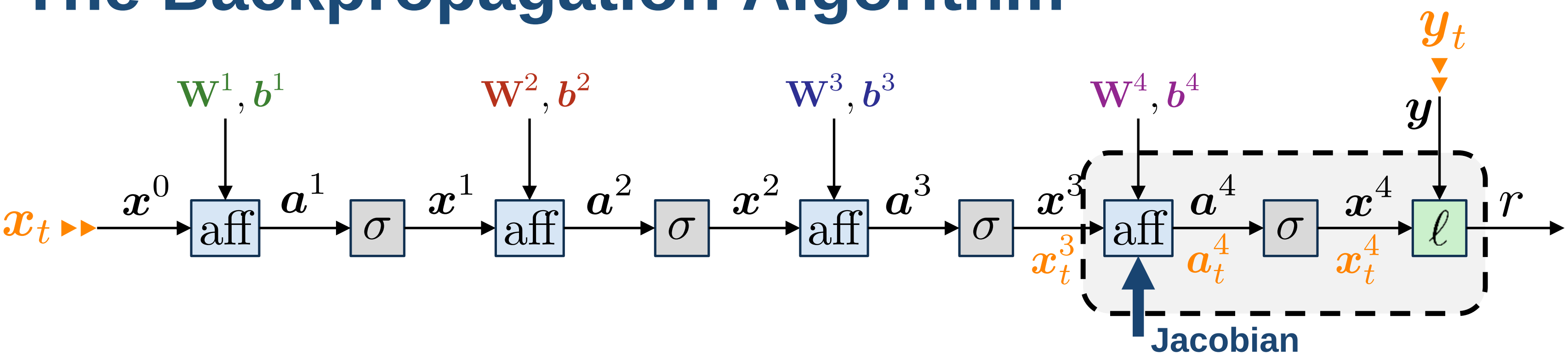


2) Then:

- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}\right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4} = \mathbf{W}^{4\top}\boldsymbol{G}_{\boldsymbol{a}^4}.$$
- $$\boldsymbol{G}_{\boldsymbol{b}^4} = \left.\frac{\partial r}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}\right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boldsymbol{G}_{\boldsymbol{a}^4}.$$
- $$\boldsymbol{G}_{\boldsymbol{w}_d^4} = \left.\frac{\partial r}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}^\top = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}\right)^\top = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}^\top \times \boldsymbol{G}_{\boldsymbol{a}^4}$$

  $\rightarrow x_{t,d}^3\mathbf{I}$

The Backpropagation Algorithm



2) Then:

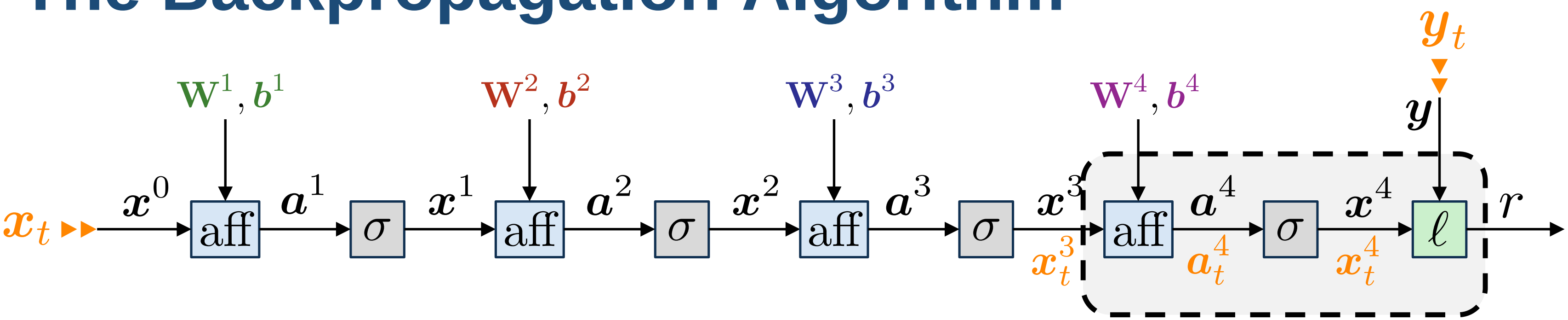
- $$\boldsymbol{G}_{\boldsymbol{x}^3} = \left.\frac{\partial r}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}\right)^{\top} = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{x}^3}\right|_{\boldsymbol{x}_t^3}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \mathbf{W}^{4\top}\boldsymbol{G}_{\boldsymbol{a}^4}.$$

- $$\boldsymbol{G}_{\boldsymbol{b}^4} = \left.\frac{\partial r}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}\right)^{\top} = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{b}^4}\right|_{\boldsymbol{b}^4}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = \boldsymbol{G}_{\boldsymbol{a}^4}.$$

- $$\boldsymbol{G}_{\boldsymbol{w}_d^4} = \left.\frac{\partial r}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}^{\top} = \left(\left.\frac{\partial r}{\partial \boldsymbol{a}^4}\right|_{\boldsymbol{a}_t^4} \times \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}\right)^{\top} = \left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}^{\top} \times \boldsymbol{G}_{\boldsymbol{a}^4} = x_{t,d}^3\boldsymbol{G}_{\boldsymbol{a}^4}.$$

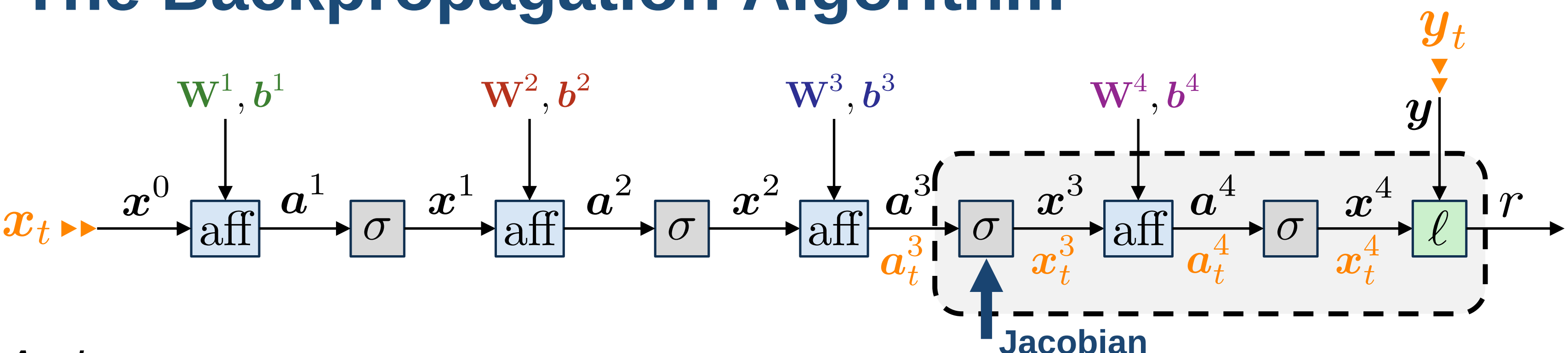
  $\left.\frac{\partial \boldsymbol{a}^4}{\partial \boldsymbol{w}_d^4}\right|_{\boldsymbol{w}_d^4}^{\top} \rightarrow x_{t,d}^3\mathbf{I}$

The Backpropagation Algorithm



And so on…

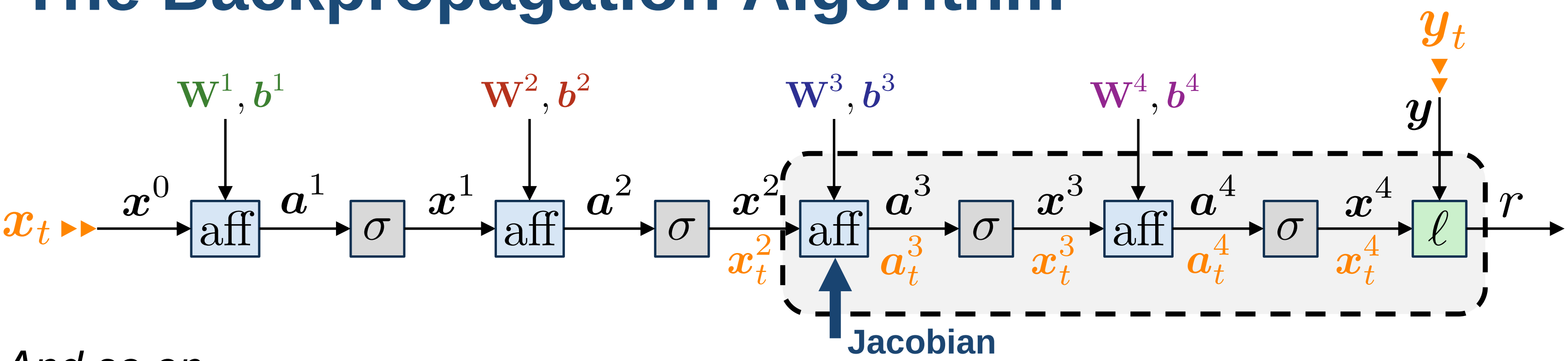
The Backpropagation Algorithm



And so on…

3) $\boldsymbol{G}_{\boldsymbol{a}^3} = \sigma'(\boldsymbol{a}_t^3) \odot \boldsymbol{G}_{\boldsymbol{x}^3}$.

The Backpropagation Algorithm



And so on…

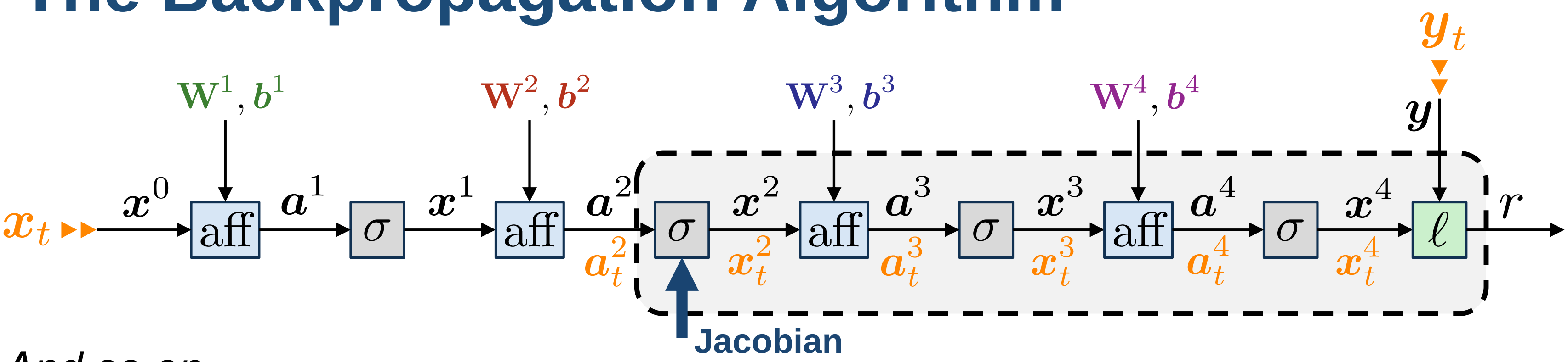
3) $\boldsymbol{G}_{\boldsymbol{a}^3} = \sigma'(\boldsymbol{a}_t^3) \odot \boldsymbol{G}_{\boldsymbol{x}^3}$.

4) $\boldsymbol{G}_{\boldsymbol{x}^2} = \mathbf{W}^{3\top}\boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{b}^3} = \boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{w}_d^3} = x_{t,d}^2 \boldsymbol{G}_{\boldsymbol{a}^3}$.

The Backpropagation Algorithm



And so on…

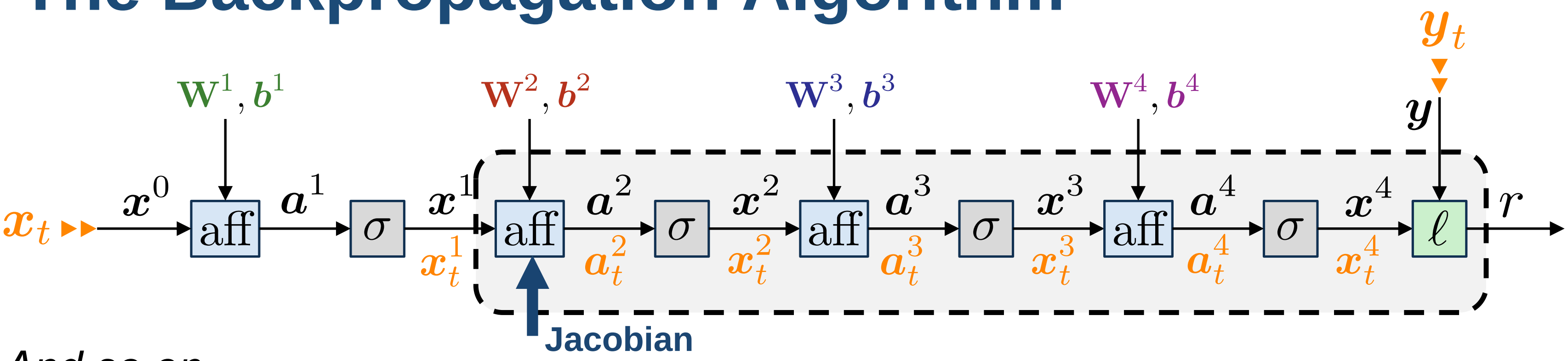
3) $\boldsymbol{G}_{\boldsymbol{a}^3} = \sigma'(\boldsymbol{a}_t^3) \odot \boldsymbol{G}_{\boldsymbol{x}^3}$. 5) $\boldsymbol{G}_{\boldsymbol{a}^2} = \sigma'(\boldsymbol{a}_t^2) \odot \boldsymbol{G}_{\boldsymbol{x}^2}$.

4) $\boldsymbol{G}_{\boldsymbol{x}^2} = \mathbf{W}^{3\top} \boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{b}^3} = \boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{w}_d^3} = x_{t,d}^2 \boldsymbol{G}_{\boldsymbol{a}^3}$.

The Backpropagation Algorithm



And so on…

3) $\boldsymbol{G}_{\boldsymbol{a}^3} = \sigma'(\boldsymbol{a}^3_t) \odot \boldsymbol{G}_{\boldsymbol{x}^3}$.

4) $\boldsymbol{G}_{\boldsymbol{x}^2} = \mathbf{W}^{3\top} \boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{b}^3} = \boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{w}^3_d} = x^2_{t,d} \boldsymbol{G}_{\boldsymbol{a}^3}$.

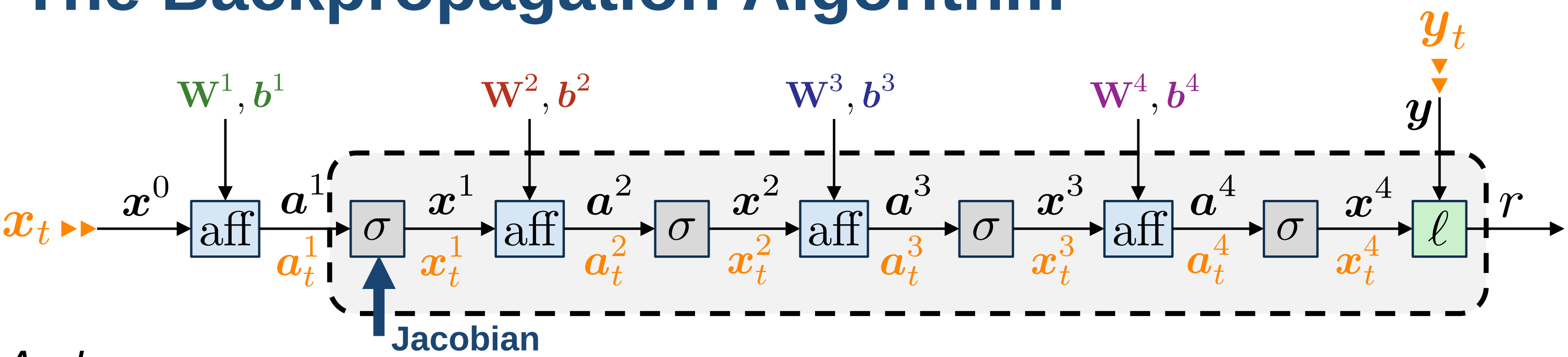
5) $\boldsymbol{G}_{\boldsymbol{a}^2} = \sigma'(\boldsymbol{a}^2_t) \odot \boldsymbol{G}_{\boldsymbol{x}^2}$.

6) $\boldsymbol{G}_{\boldsymbol{x}^1} = \mathbf{W}^{2\top} \boldsymbol{G}_{\boldsymbol{a}^2}$.

$\boldsymbol{G}_{\boldsymbol{b}^2} = \boldsymbol{G}_{\boldsymbol{a}^2}$.

$\boldsymbol{G}_{\boldsymbol{w}^2_d} = x^1_{t,d} \boldsymbol{G}_{\boldsymbol{a}^2}$.

The Backpropagation Algorithm



And so on…

3) $\boldsymbol{G}_{\boldsymbol{a}^3} = \sigma'(a_t^3) \odot \boldsymbol{G}_{\boldsymbol{x}^3}$.

5) $\boldsymbol{G}_{\boldsymbol{a}^2} = \sigma'(a_t^2) \odot \boldsymbol{G}_{\boldsymbol{x}^2}$.

7) $\boldsymbol{G}_{\boldsymbol{a}^1} = \sigma'(a_t^1) \odot \boldsymbol{G}_{\boldsymbol{x}^1}$.

4) $\boldsymbol{G}_{\boldsymbol{x}^2} = \mathbf{W}^{3\top} \boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{b}^3} = \boldsymbol{G}_{\boldsymbol{a}^3}$.

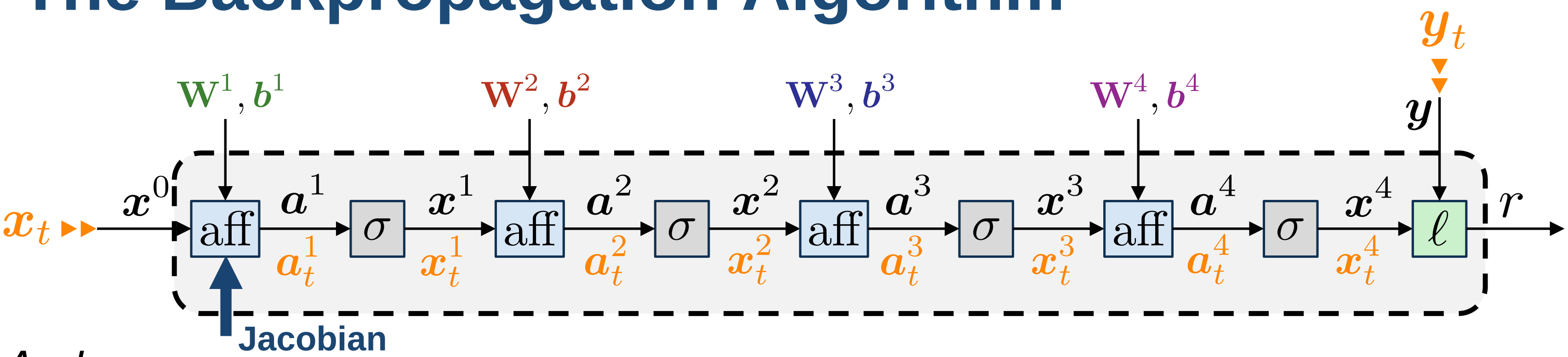
$\boldsymbol{G}_{\boldsymbol{w}_d^3} = x_{t,d}^2 \boldsymbol{G}_{\boldsymbol{a}^3}$.

6) $\boldsymbol{G}_{\boldsymbol{x}^1} = \mathbf{W}^{2\top} \boldsymbol{G}_{\boldsymbol{a}^2}$.

$\boldsymbol{G}_{\boldsymbol{b}^2} = \boldsymbol{G}_{\boldsymbol{a}^2}$.

$\boldsymbol{G}_{\boldsymbol{w}_d^2} = x_{t,d}^1 \boldsymbol{G}_{\boldsymbol{a}^2}$.

The Backpropagation Algorithm



And so on…

3) $G_{\boldsymbol{a}^3} = \sigma'(a_t^3) \odot G_{\boldsymbol{x}^3}$.

4) $G_{\boldsymbol{x}^2} = \mathbf{W}^{3\top} G_{\boldsymbol{a}^3}$.

$G_{\boldsymbol{b}^3} = G_{\boldsymbol{a}^3}$.

$G_{\boldsymbol{w}_d^3} = x_{t,d}^2 G_{\boldsymbol{a}^3}$.

5) $G_{\boldsymbol{a}^2} = \sigma'(a_t^2) \odot G_{\boldsymbol{x}^2}$.

6) $G_{\boldsymbol{x}^1} = \mathbf{W}^{2\top} G_{\boldsymbol{a}^2}$.

$G_{\boldsymbol{b}^2} = G_{\boldsymbol{a}^2}$.

$G_{\boldsymbol{w}_d^2} = x_{t,d}^1 G_{\boldsymbol{a}^2}$.

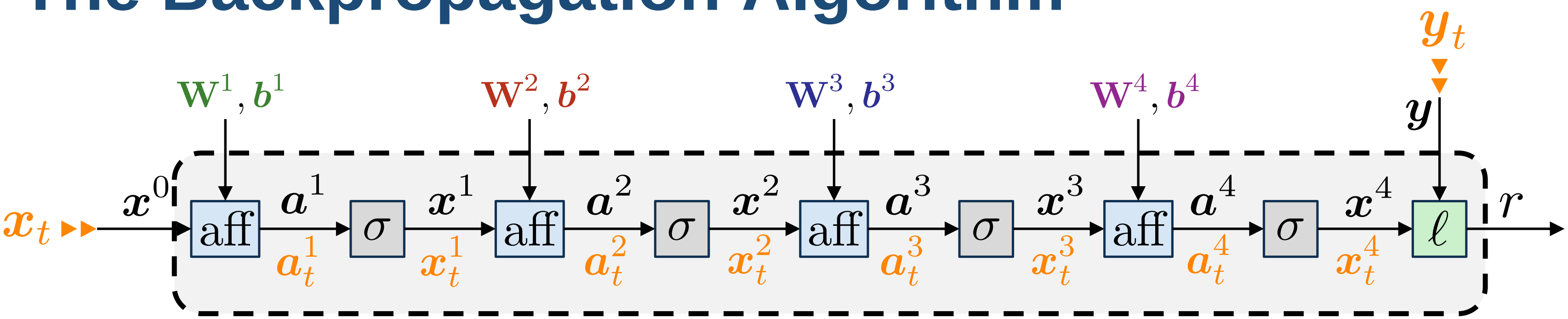
7) $G_{\boldsymbol{a}^1} = \sigma'(a_t^1) \odot G_{\boldsymbol{x}^1}$.

8) $G_{\boldsymbol{x}^0} = \mathbf{W}^{1\top} G_{\boldsymbol{a}^1}$.

$G_{\boldsymbol{b}^1} = G_{\boldsymbol{a}^1}$.

$G_{\boldsymbol{w}_d^1} = x_{t,d}^0 G_{\boldsymbol{a}^1}$.

The Backpropagation Algorithm



And so on…

3) $\boldsymbol{G}_{\boldsymbol{a}^3} = \sigma'(\boldsymbol{a}_t^3) \odot \boldsymbol{G}_{\boldsymbol{x}^3}$.

4) $\boldsymbol{G}_{\boldsymbol{x}^2} = \mathbf{W}^{3\top} \boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{b}^3} = \boldsymbol{G}_{\boldsymbol{a}^3}$.

$\boldsymbol{G}_{\boldsymbol{w}_d^3} = x_{t,d}^2 \boldsymbol{G}_{\boldsymbol{a}^3}$.

5) $\boldsymbol{G}_{\boldsymbol{a}^2} = \sigma'(\boldsymbol{a}_t^2) \odot \boldsymbol{G}_{\boldsymbol{x}^2}$.

6) $\boldsymbol{G}_{\boldsymbol{x}^1} = \mathbf{W}^{2\top} \boldsymbol{G}_{\boldsymbol{a}^2}$.

$\boldsymbol{G}_{\boldsymbol{b}^2} = \boldsymbol{G}_{\boldsymbol{a}^2}$.

$\boldsymbol{G}_{\boldsymbol{w}_d^2} = x_{t,d}^1 \boldsymbol{G}_{\boldsymbol{a}^2}$.

7) $\boldsymbol{G}_{\boldsymbol{a}^1} = \sigma'(\boldsymbol{a}_t^1) \odot \boldsymbol{G}_{\boldsymbol{x}^1}$.

8) $\boldsymbol{G}_{\boldsymbol{x}^0} = \mathbf{W}^{1\top} \boldsymbol{G}_{\boldsymbol{a}^1}$.

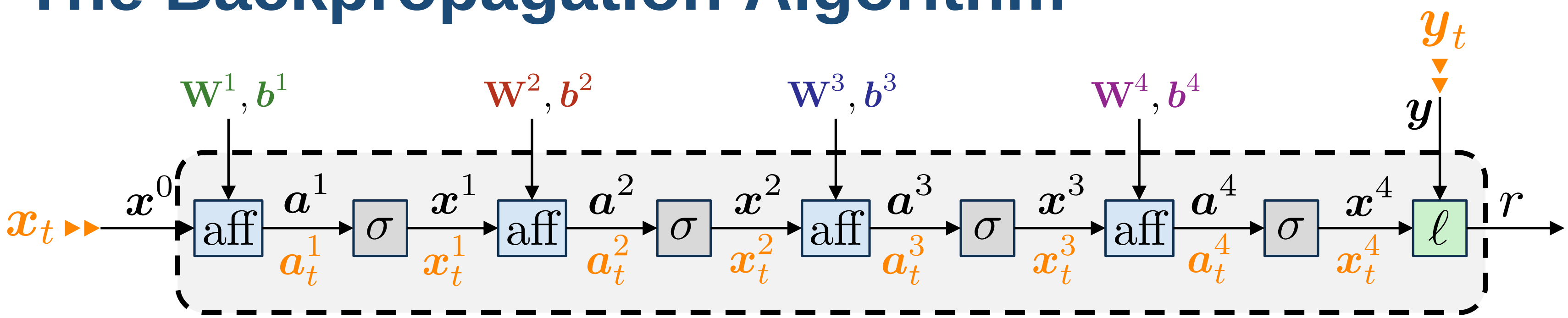
$\boldsymbol{G}_{\boldsymbol{b}^1} = \boldsymbol{G}_{\boldsymbol{a}^1}$.

$\boldsymbol{G}_{\boldsymbol{w}_d^1} = x_{t,d}^0 \boldsymbol{G}_{\boldsymbol{a}^1}$.

In Fine:

$$\boldsymbol{G}_{\boldsymbol{x}^0} = \mathbf{W}^{1\top}\sigma'(\boldsymbol{a}_t^1) \odot \mathbf{W}^{2\top}\sigma'(\boldsymbol{a}_t^2) \odot \mathbf{W}^{3\top}\sigma'(\boldsymbol{a}_t^3) \odot \mathbf{W}^{4\top}\sigma'(\boldsymbol{a}_t^4) \odot \nabla_{\boldsymbol{x}^4}\ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t).$$

The Backpropagation Algorithm



And so on…

3) $\boldsymbol{G}_{\boldsymbol{a}^3} = \sigma'(\boldsymbol{a}_t^3) \odot \boldsymbol{G}_{\boldsymbol{x}^3}$.
5) $\boldsymbol{G}_{\boldsymbol{a}^2} = \sigma'(\boldsymbol{a}_t^2) \odot \boldsymbol{G}_{\boldsymbol{x}^2}$.
7) $\boldsymbol{G}_{\boldsymbol{a}^1} = \sigma'(\boldsymbol{a}_t^1) \odot \boldsymbol{G}_{\boldsymbol{x}^1}$.

4) $\boldsymbol{G}_{\boldsymbol{x}^2} = \mathbf{W}^{3\top}\boldsymbol{G}_{\boldsymbol{a}^3}$.
$\boldsymbol{G}_{\boldsymbol{b}^3} = \boldsymbol{G}_{\boldsymbol{a}^3}$.
$\boldsymbol{G}_{\boldsymbol{w}_d^3} = x_{t,d}^2 \boldsymbol{G}_{\boldsymbol{a}^3}$.

6) $\boldsymbol{G}_{\boldsymbol{x}^1} = \mathbf{W}^{2\top}\boldsymbol{G}_{\boldsymbol{a}^2}$.
$\boldsymbol{G}_{\boldsymbol{b}^2} = \boldsymbol{G}_{\boldsymbol{a}^2}$.
$\boldsymbol{G}_{\boldsymbol{w}_d^2} = x_{t,d}^1 \boldsymbol{G}_{\boldsymbol{a}^2}$.

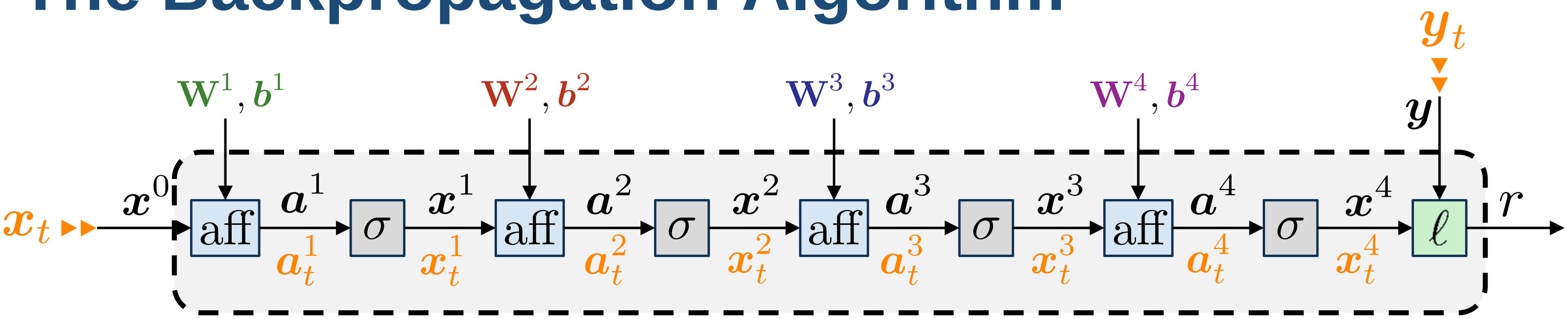
8) $\boldsymbol{G}_{\boldsymbol{x}^0} = \mathbf{W}^{1\top}\boldsymbol{G}_{\boldsymbol{a}^1}$.
$\boldsymbol{G}_{\boldsymbol{b}^1} = \boldsymbol{G}_{\boldsymbol{a}^1}$.
$\boldsymbol{G}_{\boldsymbol{w}_d^1} = x_{t,d}^0 \boldsymbol{G}_{\boldsymbol{a}^1}$.

In Fine:

$$\boldsymbol{G}_{\boldsymbol{x}^0} = \mathbf{W}^{1\top}\sigma'(\boldsymbol{a}_t^1) \odot \mathbf{W}^{2\top}\sigma'(\boldsymbol{a}_t^2) \odot \mathbf{W}^{3\top}\sigma'(\boldsymbol{a}_t^3) \odot \mathbf{W}^{4\top}\sigma'(\boldsymbol{a}_t^4) \odot \nabla_{\boldsymbol{x}^4}\ell(\boldsymbol{x}_t^4, \boldsymbol{y}_t).$$

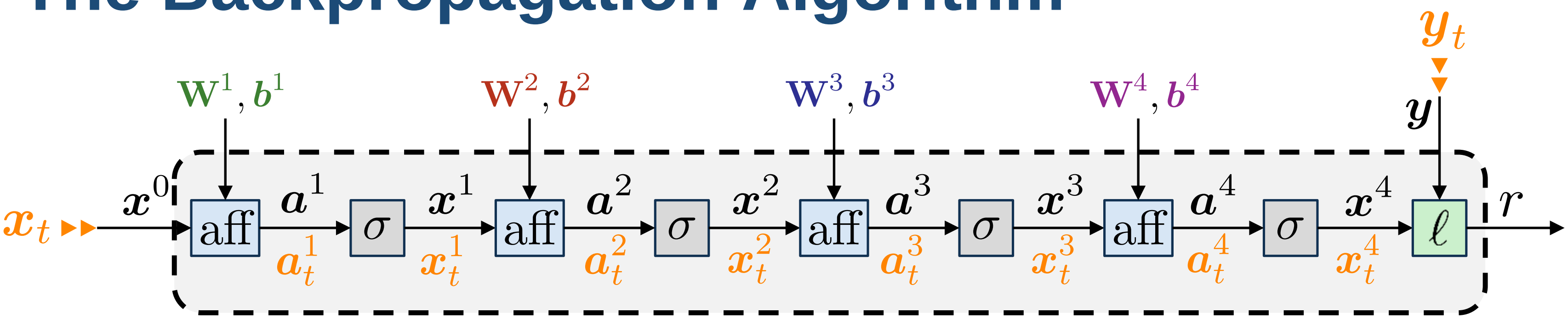
⇨ Using the parameters' gradients, we **update** them via **gradient descent**.

The Backpropagation Algorithm



Conclusions

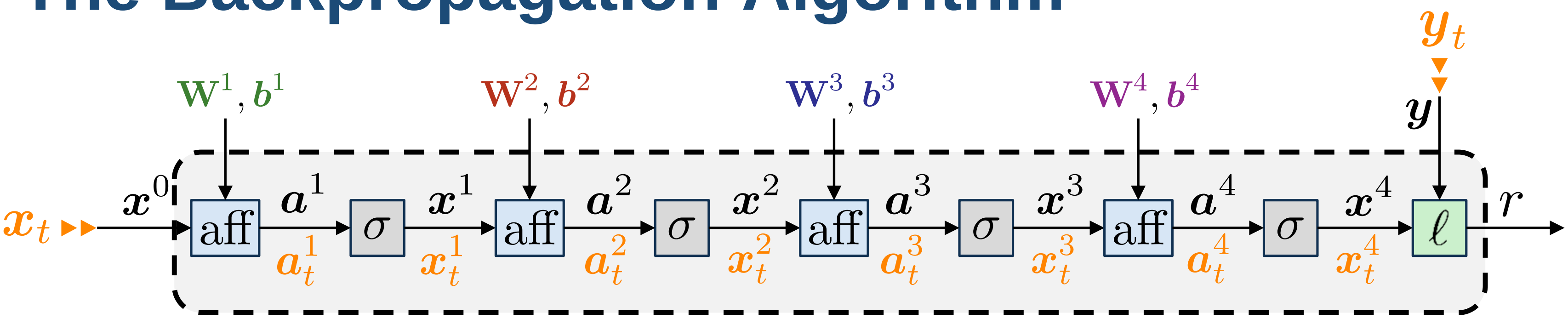
The Backpropagation Algorithm



Conclusions

- The idea is **very general** and can be applied to **any feedforward** neural network architecture

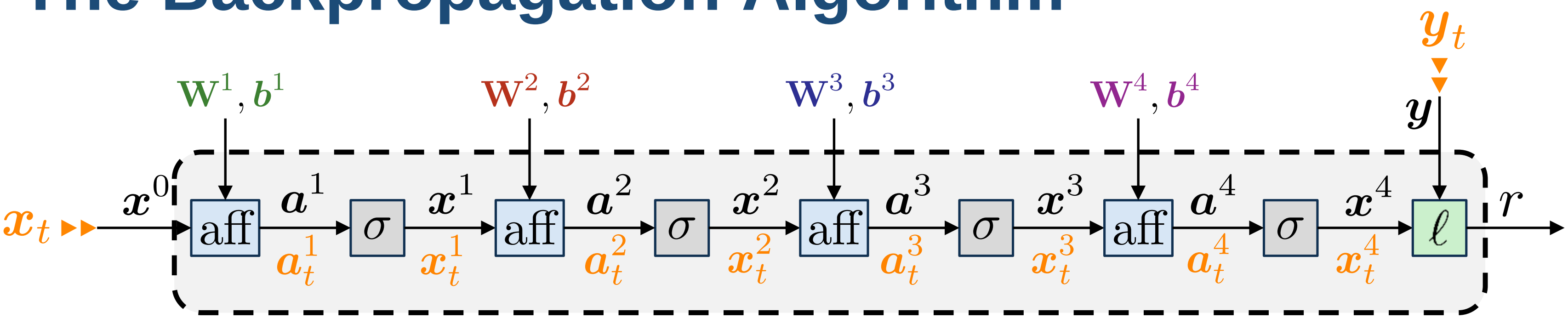
The Backpropagation Algorithm



Conclusions

- The idea is **very general** and can be applied to **any feedforward** neural network architecture
- **There are 4 key ingredients**:
  - the **data** (constants)
  - the **parameters** (*free* variables to optimize)
  - the **activations** / **layer outputs** (*dependent* variables)
  - the **functions** / **layers** (layers are generally compositions of functions)

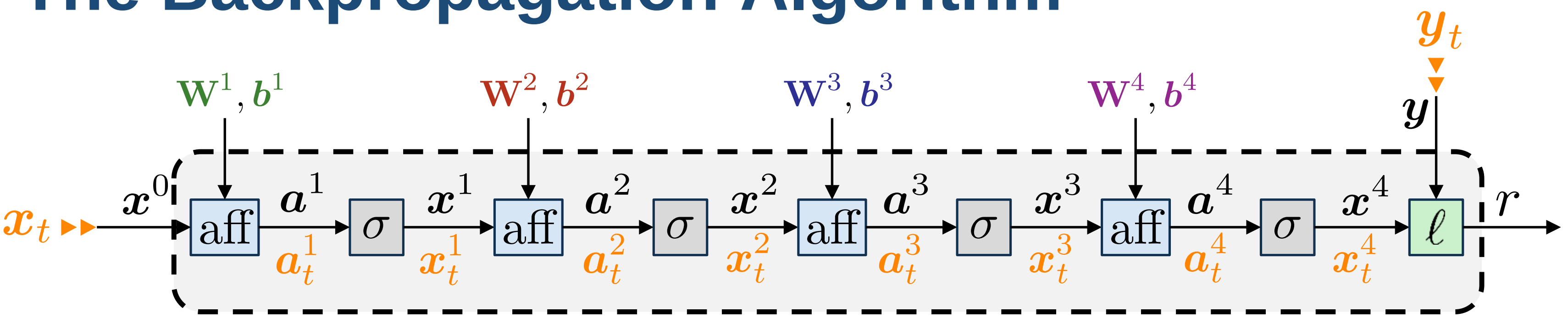
The Backpropagation Algorithm



Conclusions

- The idea is **very general** and can be applied to **any feedforward** neural network architecture
- **There are 4 key ingredients**:
  - the **data** (constants)
  - the **parameters** (*free* variables to optimize)
  - the **activations** / **layer outputs** (*dependent* variables)
  - the **functions** / **layers** (layers are generally compositions of functions)
- The data flows **forwards** while the gradient propagates **backwards**, a bit like another neural network, with only vector/matrix multiplications

The Backpropagation Algorithm



Conclusions

- The idea is **very general** and can be applied to **any feedforward** neural network architecture
- **There are 4 key ingredients**:
  - the **data** (constants)
  - the **parameters** (*free* variables to optimize)
  - the **activations** / **layer outputs** (*dependent* variables)
  - the **functions** / **layers** (layers are generally compositions of functions)
- The data flows **forwards** while the gradient propagates **backwards**, a bit like another neural network, with only vector/matrix multiplications
- All we need are the **forward operators** and **Jacobians** of each module

Back to gradient descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- Remember that we train DNNs using Empirical Risk Minimization:

$$L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \ell\left(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t\right) \approx \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\{\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{Y})\}$$

Back to gradient descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- Remember that we train DNNs using Empirical Risk Minimization:

$$L(\text{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \ell\left(\text{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t\right) \approx \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\{\ell(\text{dnn}_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{Y})\}$$

- We compute the gradient of the **total loss** by summing the gradients of the **loss** at individual data samples:

$$\nabla_{\boldsymbol{\theta}} L(\text{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T}\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \ell(\text{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

Back to gradient descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- Remember that we train DNNs using Empirical Risk Minimization:

$$L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \ell\left(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t\right) \approx \mathbb{E}_{\boldsymbol{X},\boldsymbol{Y}}\{\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{X}), \boldsymbol{Y})\}$$

- We compute the gradient of the **total loss** by summing the gradients of the **loss** at individual data samples:

$$\nabla_{\boldsymbol{\theta}} L(\mathrm{dnn}_{\boldsymbol{\theta}}, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

- But doing so across the **entire dataset** (e.g.: 1 million images) for **every gradient step** would be very expansive.

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

(The SGD algorithm)

- At each iteration (i), compute the gradient over a **random subset** $\mathcal{T}^{(i)} \subseteq \mathcal{T}$ and perform one step of gradient descent:

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

Stochastic Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

(The SGD algorithm)

- At each iteration (i), compute the gradient over a **random subset** $\mathcal{T}^{(i)} \subseteq \mathcal{T}$ and perform one step of gradient descent:

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

- At the next iteration, pick another (disjoint) random subset

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

(The SGD algorithm)

- At each iteration (i), compute the gradient over a **random subset** $\mathcal{T}^{(i)} \subseteq \mathcal{T}$ and perform one step of gradient descent:

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

- At the next iteration, pick another (disjoint) random subset
- When the entire dataset has passed, start over again

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

(The SGD algorithm)

- At each iteration (i), compute the gradient over a **random subset** $\mathcal{T}^{(i)} \subseteq \mathcal{T}$ and perform one step of gradient descent:

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

- At the next iteration, pick another (disjoint) random subset
- When the entire dataset has passed, start over again
- Each $\mathcal{T}^{(i)}$ is called a **minibatch**

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

(The SGD algorithm)

- At each iteration (i), compute the gradient over a **random subset** $\mathcal{T}^{(i)} \subseteq \mathcal{T}$ and perform one step of gradient descent:

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

- At the next iteration, pick another (disjoint) random subset
- When the entire dataset has passed, start over again
- Each $\mathcal{T}^{(i)}$ is called a **minibatch**
- Each pass over the entire dataset is called an **epoch**

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

(The SGD algorithm)

- At each iteration (i), compute the gradient over a **random subset** $\mathcal{T}^{(i)} \subseteq \mathcal{T}$ and perform one step of gradient descent:

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

- At the next iteration, pick another (disjoint) random subset
- When the entire dataset has passed, start over again
- Each $\mathcal{T}^{(i)}$ is called a **minibatch**
- Each pass over the entire dataset is called an **epoch**

Splitting the training set into B minibatches:

- Reduces the computation cost of one gradient by a factor of B

Stochastic Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

(The SGD algorithm)

- At each iteration (i), compute the gradient over a **random subset** $\mathcal{T}^{(i)} \subseteq \mathcal{T}$ and perform one step of gradient descent:

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

- At the next iteration, pick another (disjoint) random subset
- When the entire dataset has passed, start over again
- Each $\mathcal{T}^{(i)}$ is called a **minibatch**
- Each pass over the entire dataset is called an **epoch**

Splitting the training set into B minibatches:

- Reduces the computation cost of one gradient by a factor of B
- Increases the **standard deviation** on the gradient estimate by a factor of $\sqrt{B}$ only.

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

(The SGD algorithm)

- At each iteration (i), compute the gradient over a **random subset** $\mathcal{T}^{(i)} \subseteq \mathcal{T}$ and perform one step of gradient descent:

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$$

- At the next iteration, pick another (disjoint) random subset
- When the entire dataset has passed, start over again
- Each $\mathcal{T}^{(i)}$ is called a **minibatch**
- Each pass over the entire dataset is called an **epoch**

Splitting the training set into B minibatches:

- Reduces the computation cost of one gradient by a factor of B
- Increases the **standard deviation** on the gradient estimate by a factor of $\sqrt{B}$ only.

More iterations but fewer epochs
= less total computation

Stochastic Gradient Descent

(The SGD algorithm)

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

(The SGD algorithm)

- In practice the gradients $\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$ of all examples $(\boldsymbol{x}_t, \boldsymbol{y}_t)$ are computed in **parallel** using a **graphical processing unit** (GPU) and summed up within a minibatch

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

(The SGD algorithm)

- In practice the gradients $\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$ of all examples $(\boldsymbol{x}_t, \boldsymbol{y}_t)$ are computed in **parallel** using a **graphical processing unit** (GPU) and summed up within a minibatch

- The choice of the minibatch size is governed by these considerations:
  - The minibatch data and computations must fit in **GPU memory**
  - Too small minibatches do **not exploit well** GPU capabilities
  - Some kinds of hardware perform better with **power-of-2** sizes

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

Stochastic Gradient Descent

(The SGD algorithm)

- In practice the gradients $\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$ of all examples $(\boldsymbol{x}_t, \boldsymbol{y}_t)$ are computed in **parallel** using a **graphical processing unit** (GPU) and summed up within a minibatch

- The choice of the minibatch size is governed by these considerations:
  - The minibatch data and computations must fit in **GPU memory**
  - Too small minibatches do **not exploit well** GPU capabilities
  - Some kinds of hardware perform better with **power-of-2** sizes

- Typical minibatch sizes: from 32 to 256.

Stochastic Gradient Descent

(The SGD algorithm)

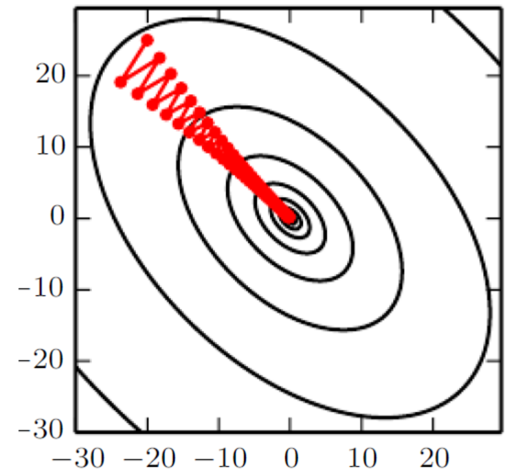
$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- In practice the gradients $\ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t)$ of all examples $(\boldsymbol{x}_t, \boldsymbol{y}_t)$ are computed in **parallel** using a **graphical processing unit** (GPU) and summed up within a minibatch

- The choice of the minibatch size is governed by these considerations:
  - The minibatch data and computations must fit in **GPU memory**
  - Too small minibatches do **not exploit well** GPU capabilities
  - Some kinds of hardware perform better with **power-of-2** sizes

- Typical minibatch sizes: from 32 to 256.

- **Limit of SGD:** Tends to "zigzag" when descending a "canyon", which increases the number of iterations



SGD with Momentum

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

SGD with Momentum

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- **Solution:** “smooth” the gradient estimates across several iterations.

SGD with Momentum

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- **Solution:** “smooth” the gradient estimates across several iterations.
- **Momentum** = vector $\boldsymbol{v}$ representing the direction and speed at which the parameters move through parameter space.

SGD with Momentum

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- **Solution:** "smooth" the gradient estimates across several iterations.
- **Momentum** = vector $\boldsymbol{v}$ representing the direction and speed at which the parameters move through parameter space.
- Defined as an ***exponentially decaying average*** of the negative gradient.

SGD with Momentum

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- **Solution:** "smooth" the gradient estimates across several iterations.
- **Momentum** = vector $\boldsymbol{v}$ representing the direction and speed at which the parameters move through parameter space.
- Defined as an ***exponentially decaying average*** of the negative gradient.
- **SGD with momentum**: initialize $\boldsymbol{v}^{(0)} = \mathbf{0}$, then replace each iteration of SGD by:

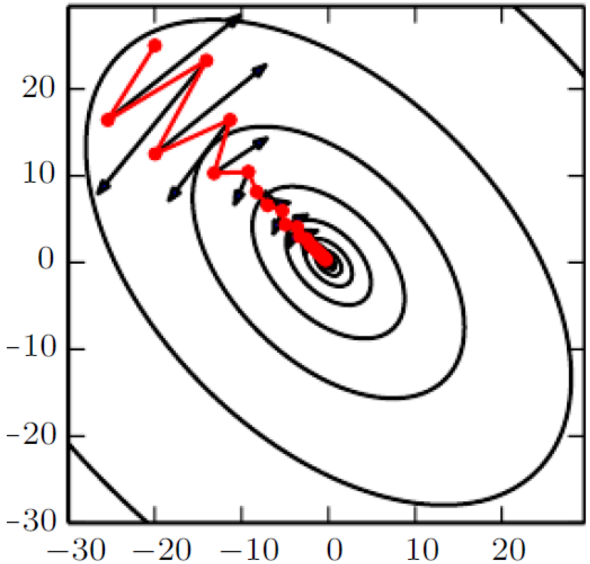
$$\begin{cases} \boldsymbol{v}^{(i+1)} \leftarrow \alpha \boldsymbol{v}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t) \\ \boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} + \boldsymbol{v}^{(i)} \end{cases}$$

SGD with Momentum

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- **Solution:** "smooth" the gradient estimates across several iterations.
- **Momentum** = vector $\boldsymbol{v}$ representing the direction and speed at which the parameters move through parameter space.
- Defined as an ***exponentially decaying average*** of the negative gradient.
- **SGD with momentum**: initialize $\boldsymbol{v}^{(0)} = \mathbf{0}$, then replace each iteration of SGD by:

$$\begin{cases} \boldsymbol{v}^{(i+1)} \leftarrow \alpha \boldsymbol{v}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t) \\ \boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} + \boldsymbol{v}^{(i)} \end{cases}$$



Converges faster than SGD

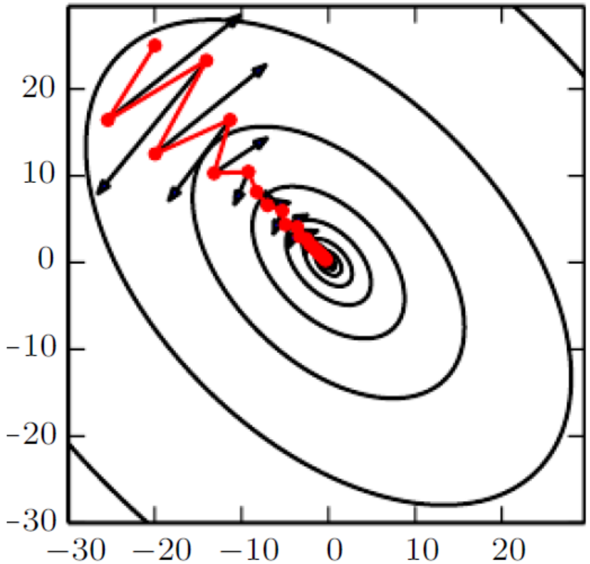
SGD with Momentum

$$\boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} - \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}^{(i)})$$

- **Solution:** "smooth" the gradient estimates across several iterations.
- **Momentum** = vector $\boldsymbol{v}$ representing the direction and speed at which the parameters move through parameter space.
- Defined as an ***exponentially decaying average*** of the negative gradient.
- **SGD with momentum**: initialize $\boldsymbol{v}^{(0)} = \mathbf{0}$, then replace each iteration of SGD by:

$$\begin{cases} \boldsymbol{v}^{(i+1)} \leftarrow \alpha \boldsymbol{v}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(\boldsymbol{x}_t, \boldsymbol{y}_t) \in \mathcal{T}_i} \nabla_{\boldsymbol{\theta}} \ell(\mathrm{dnn}_{\boldsymbol{\theta}}(\boldsymbol{x}_t), \boldsymbol{y}_t) \\ \boldsymbol{\theta}^{(i+1)} \leftarrow \boldsymbol{\theta}^{(i)} + \boldsymbol{v}^{(i)} \end{cases}$$

- The very popular **ADAM optimizer** (**140k** citations since 2014!) extends this idea by also averaging **squared** gradients.



Converges faster than SGD

Local Minima

Local Minima

When properly tuned (learning rate not too large nor too small), SGD converges to a **local minimum**.

Local Minima

When properly tuned (learning rate not too large nor too small), SGD converges to a **local minimum**.

How many local minima are they? Are they good or bad?

Local Minima

When properly tuned (learning rate not too large nor too small), SGD converges to a **local minimum**.

How many local minima are they? Are they good or bad?

Neural networks always have multiple local minima because of **model identifiability** issues (things that do no change the value of the loss):

Local Minima

When properly tuned (learning rate not too large nor too small), SGD converges to a **local minimum**.

How many local minima are they? Are they good or bad?

Neural networks always have multiple local minima because of **model identifiability** issues (things that do no change the value of the loss):

- **Reordering** the neurons in each layer ($N!^K$ possible orderings for K layers with N neurons each!)

Local Minima

When properly tuned (learning rate not too large nor too small), SGD converges to a **local minimum**.

How many local minima are they? Are they good or bad?

Neural networks always have multiple local minima because of **model identifiability** issues (things that do no change the value of the loss):

- **Reordering** the neurons in each layer ($N!^K$ possible orderings for K layers with N neurons each!)
- **Scaling** the incoming weights and biases of a ReLU neuron by β and its outgoing weights by $1/\beta$.

Local Minima

When properly tuned (learning rate not too large nor too small), SGD converges to a **local minimum**.

How many local minima are they? Are they good or bad?

Neural networks always have multiple local minima because of **model identifiability** issues (things that do no change the value of the loss):

- **Reordering** the neurons in each layer ($N!^K$ possible orderings for K layers with N neurons each!)
- **Scaling** the incoming weights and biases of a ReLU neuron by β and its outgoing weights by $1/\beta$.

→ This creates a large or infinite number of local minima, but they are **all equivalent** to each other (not a problem).

Local Minima

For many years, people believed that large neural networks failed because of poor local minima.

Local Minima

For many years, people believed that large neural networks failed because of poor local minima.

Recent theoretical and experimental results suggest that, for **sufficiently large** neural networks:

- Most stationary points are **saddle points** corresponding to a **high value** of the loss function

Local Minima

For many years, people believed that large neural networks failed because of poor local minima.

Recent theoretical and experimental results suggest that, for **sufficiently large** neural networks:

- Most stationary points are **saddle points** corresponding to a **high value** of the loss function
- SGD manages to avoid them in practice

Local Minima

For many years, people believed that large neural networks failed because of poor local minima.

Recent theoretical and experimental results suggest that, for **sufficiently large** neural networks:

- Most stationary points are **saddle points** corresponding to a **high value** of the loss function
- SGD manages to avoid them in practice
- Most local minima correspond to a **low value** of the cost function

The PyTorch framework

The PyTorch framework

PyTorch

GOOD NEWS: **You** (probably) **won't ever need to implement backpropagation or SGD yourself!** ☺

The PyTorch framework

PyTorch

GOOD NEWS: You (probably) **won't ever need to implement backpropagation or SGD yourself!** ☺

- PyTorch is an opensource Python library designed to easily **design**, **train** and **test** neural networks, initially developed by Facebook (Meta), based on Torch. **Constantly evolving** thanks to a broad community

The PyTorch framework

PyTorch

GOOD NEWS: You (probably) **won't ever need to implement backpropagation or SGD yourself!** ☺

- PyTorch is an opensource Python library designed to easily **design**, **train** and **test** neural networks, initially developed by Facebook (Meta), based on Torch. **Constantly evolving** thanks to a broad community
- It uses the **abstractions** allowed by Python, and in particular **object oriented programming**, in order to seamlessly manipulate all the objects we have seen: **data/constants**, **variables/parameters**, **functions**, **optimizers**, **loss…**

The PyTorch framework

GOOD NEWS: **You** (probably) **won't ever need to implement backpropagation or SGD yourself!** ☺

- PyTorch is an opensource Python library designed to easily **design**, **train** and **test** neural networks, initially developed by Facebook (Meta), based on Torch. **Constantly evolving** thanks to a broad community

- It uses the **abstractions** allowed by Python, and in particular **object oriented programming**, in order to seamlessly manipulate all the objects we have seen: **data/constants**, **variables/parameters**, **functions**, **optimizers**, **loss…**

- It uses **differential programming**, a concept first introduced in ***Theano***. A module called ***AutoGrad*** automatically records every operations done on variables, so that the gradient of complex functions (such as DNN) can be automatically calculated using **backprop** and **elementary gradients**.

The PyTorch framework

GOOD NEWS: You (probably) **won't ever need to implement backpropagation or SGD yourself!** ☺

- PyTorch is an opensource Python library designed to easily **design**, **train** and **test** neural networks, initially developed by Facebook (Meta), based on Torch. **Constantly evolving** thanks to a broad community

- It uses the **abstractions** allowed by Python, and in particular **object oriented programming**, in order to seamlessly manipulate all the objects we have seen: **data/constants**, **variables/parameters**, **functions**, **optimizers**, **loss…**

- It uses **differential programming**, a concept first introduced in ***Theano***. A module called ***AutoGrad*** automatically records every operations done on variables, so that the gradient of complex functions (such as DNN) can be automatically calculated using **backprop** and **elementary gradients**.

- Includes support for **GPU** and a C++ interface.

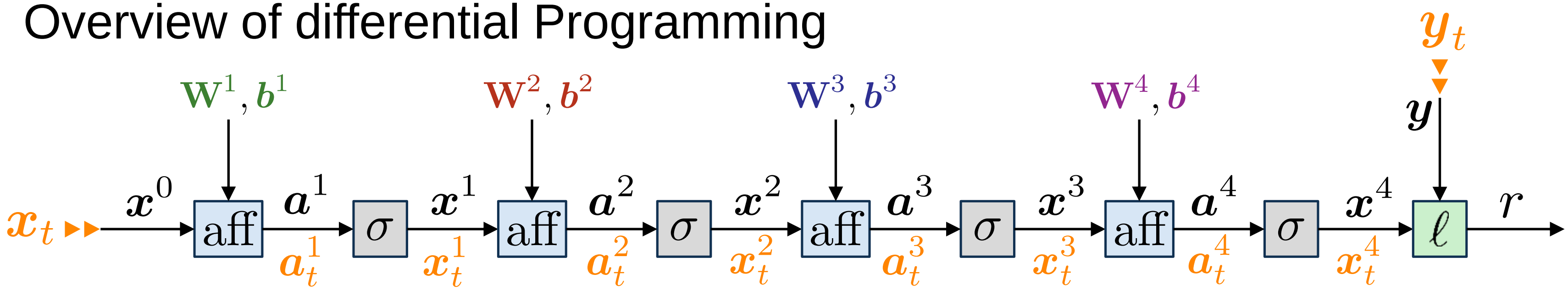
The PyTorch framework

GOOD NEWS: You (probably) **won't ever need to implement backpropagation or SGD yourself!** ☺

- PyTorch is an opensource Python library designed to easily **design**, **train** and **test** neural networks, initially developed by Facebook (Meta), based on Torch. **Constantly evolving** thanks to a broad community
- It uses the **abstractions** allowed by Python, and in particular **object oriented programming**, in order to seamlessly manipulate all the objects we have seen: **data/constants**, **variables/parameters**, **functions**, **optimizers**, **loss…**
- It uses **differential programming**, a concept first introduced in ***Theano***. A module called ***AutoGrad*** automatically records every operations done on variables, so that the gradient of complex functions (such as DNN) can be automatically calculated using **backprop** and **elementary gradients**.
- Includes support for **GPU** and a C++ interface.
- Competing framework: **TensorFlow**, initially developed by Google Brain.
  ≈ *TensorFlow → Production / PyTorch → R&D.*

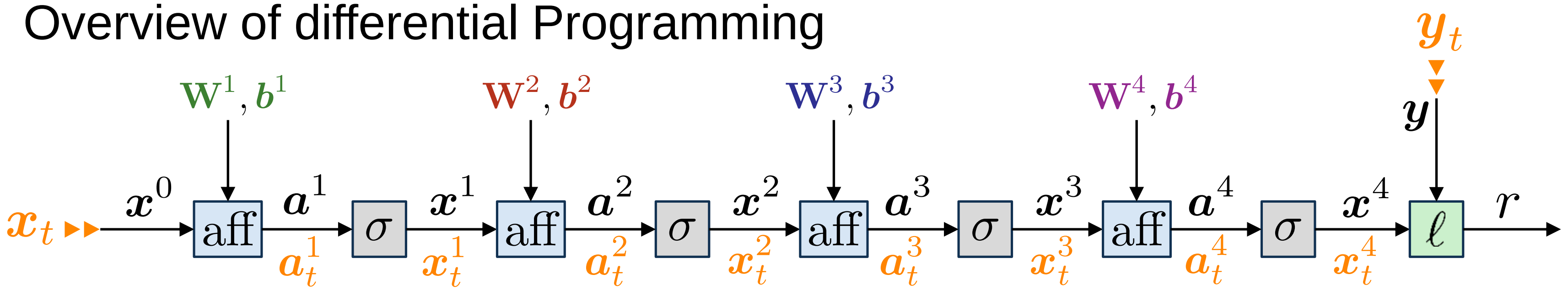
The PyTorch framework

Overview of differential Programming



The PyTorch framework

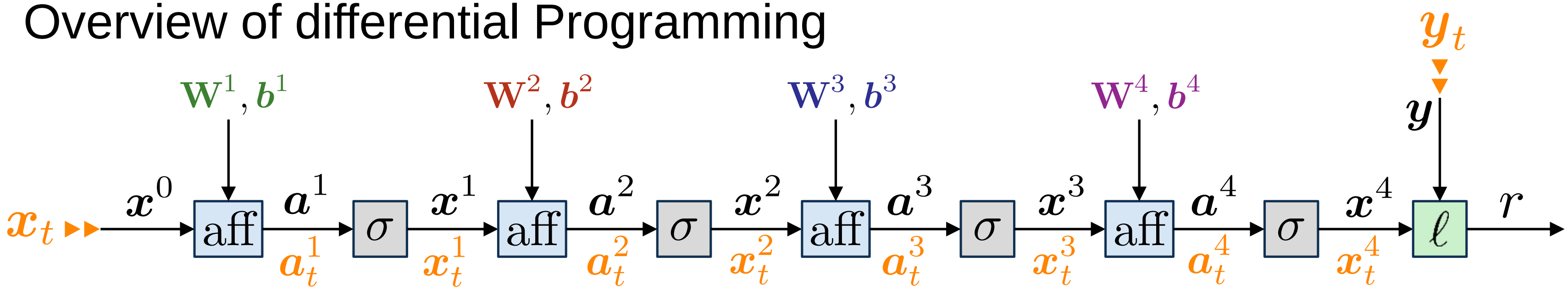
Overview of differential Programming



- Model the network as an ***acyclic computational flow graph***

The PyTorch framework

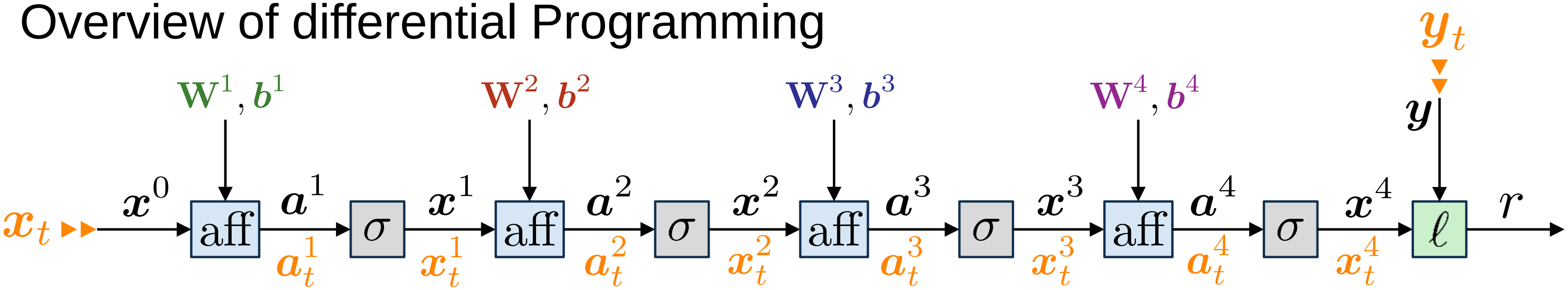
Overview of differential Programming



- Model the network as an ***acyclic computational flow graph***
- Associate each box with a `forward` method, that computes the value of the box given its children

The PyTorch framework

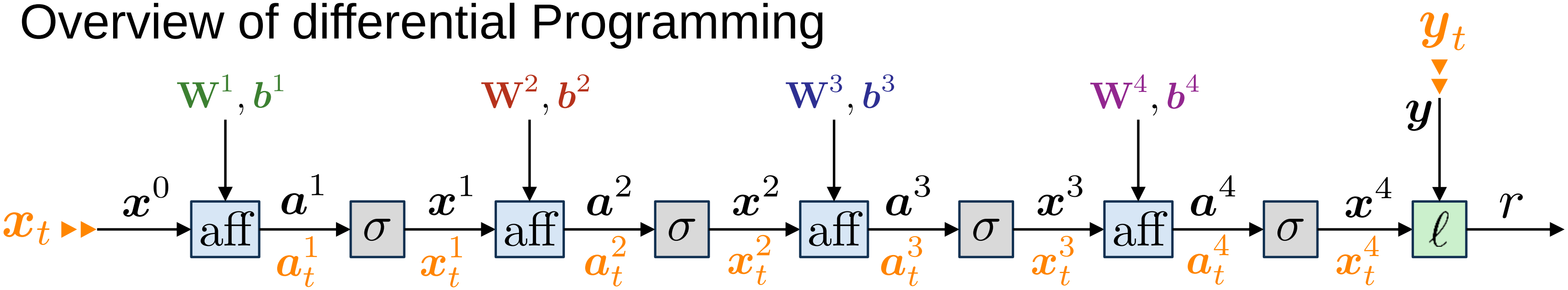
Overview of differential Programming



- Model the network as an ***acyclic computational flow graph***
- Associate each box with a `forward` method, that computes the value of the box given its children
- Call the `forward` method of each box in **left->right** order

The PyTorch framework

Overview of differential Programming

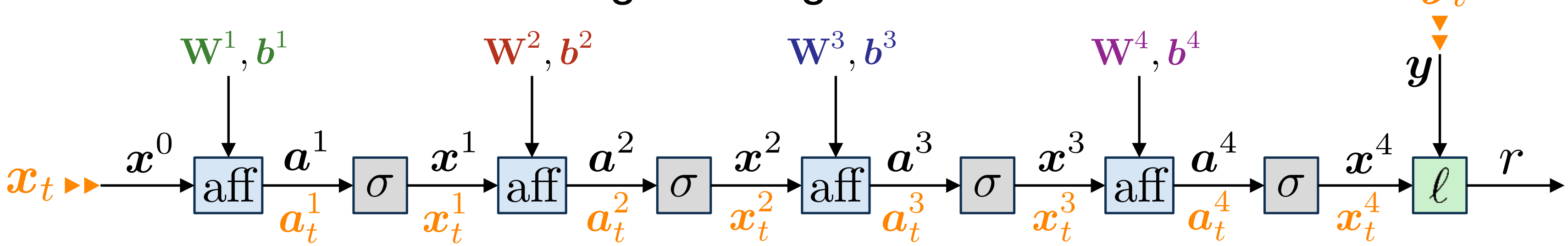


- Model the network as an ***acyclic computational flow graph***
- Associate each box with a `forward` method, that computes the value of the box given its children
- Call the `forward` method of each box in **left->right** order

Similarly for backpropagation:

The PyTorch framework

Overview of differential Programming



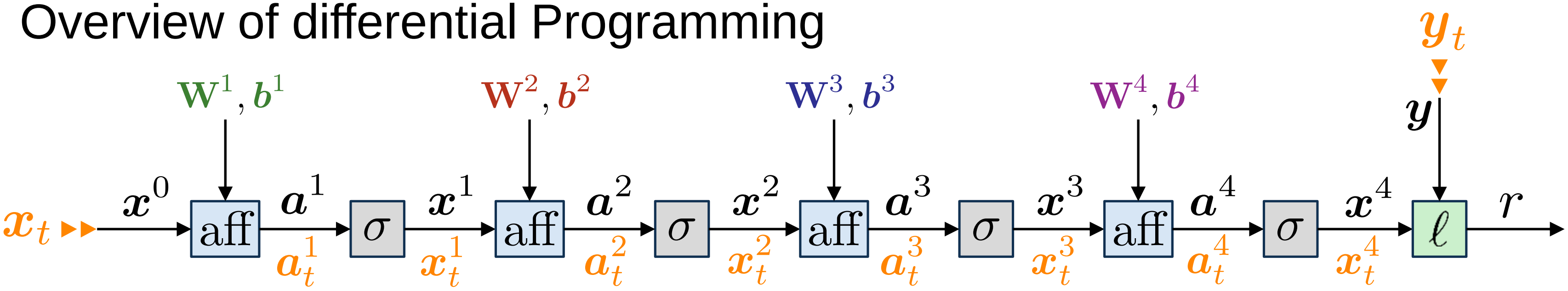
- Model the network as an ***acyclic computational flow graph***
- Associate each box with a `forward` method, that computes the value of the box given its children
- Call the `forward` method of each box in **left->right** order

Similarly for backpropagation:

- Associate each box with a backward method, that computes the gradient with respect to each child box

The PyTorch framework

Overview of differential Programming



- Model the network as an ***acyclic computational flow graph***
- Associate each box with a `forward` method, that computes the value of the box given its children
- Call the `forward` method of each box in **left->right** order

Similarly for backpropagation:

- Associate each box with a backward method, that computes the gradient with respect to each child box
- Call the backward method of each box in reverse, **right->left** order

The PyTorch framework

PyTorch

- Tensors (Data)

```
import torch
a = torch.Tensor([[1,2],[3,4]])
print(a)
```

```
1 2
3 4
[torch.FloatTensor of size 2x2]
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

PyTorch

- Tensors (Data)

```
import torch
a = torch.Tensor([[1,2],[3,4]])
print(a)
```

```
1 2
3 4
[torch.FloatTensor of size 2x2]
```

```
print(a**2)
```

```
1 4
9 16
[torch.FloatTensor of size 2x2]
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

- Variables, Functions and Autograd

```
from torch.autograd import Variable
a = Variable(torch.Tensor([[1,2],[3,4]]), requires_grad=True)
print(a)
```

```
Variable containing:
1 2
3 4
[torch.FloatTensor of size 2x2]
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

- Variables, Functions and Autograd

```
from torch.autograd import Variable
a = Variable(torch.Tensor([[1,2],[3,4]]), requires_grad=True)
print(a)
```

```
Variable containing:
1 2
3 4
[torch.FloatTensor of size 2x2]
```

```
y = torch.sum(a**2) # 1 + 4 + 9 + 16
print(y)
```

```
Variable containing:
30
[torch.FloatTensor of size 1]
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

- Variables, Functions and Autograd

```
from torch.autograd import Variable
a = Variable(torch.Tensor([[1,2],[3,4]]), requires_grad=True)
print(a)
```

```
Variable containing:
1 2
3 4
[torch.FloatTensor of size 2x2]
```

```
y = torch.sum(a**2) # 1 + 4 + 9 + 16
print(y)
```

```
Variable containing:
30
[torch.FloatTensor of size 1]
```

```
y.backward() # compute gradients of y wrt a
print(a.grad) # print dy/da_ij = 2*a_ij for a_11, a_12, a21, a22
```

```
Variable containing:
2 4
6 8
[torch.FloatTensor of size 2x2]
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

PyTorch

- Loss

```
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(out, target)
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

- Loss

```
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(out, target)

def myCrossEntropyLoss(outputs, labels):
    batch_size = outputs.size()[0] # batch_size
    outputs = F.log_softmax(outputs, dim=1) # compute the log of softmax values
    outputs = outputs[(batch_size), labels] # pick the values corresponding to the labels
    return -torch.sum(outputs)/num_examples
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

PyTorch

- Models / Neural Network Modules

```
import torch.nn as nn
import torch.nn.functional as F
class TwoLayerNet(nn.Module):
    def __init__(self, D_in, H, D_out):
        """ Constructor. Instantiate two nn.Linear modules and assign them as member variables.
        D_in: input dimension,  H: dimension of hidden layer,   D_out: output dimension
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = nn.Linear(D_in, H)
        self.linear2 = nn.Linear(H, D_out)
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

- Models / Neural Network Modules

```
import torch.nn as nn
import torch.nn.functional as F
class TwoLayerNet(nn.Module):
    def __init__(self, D_in, H, D_out):
        """ Constructor. Instantiate two nn.Linear modules and assign them as member variables.
        D_in: input dimension,  H: dimension of hidden layer,   D_out: output dimension
        """
        super(TwoLayerNet, self).__init__()
        self.linear1 = nn.Linear(D_in, H)
        self.linear2 = nn.Linear(H, D_out)

 def forward(self, x):
        """ In the forward function we accept a Variable of input data and we must return a
        Variable of output data. We can use Modules defined in the constructor as well as arbitrary
        operators on Variables.
        """
        h_relu = F.relu(self.linear1(x))
        y_pred = self.linear2(h_relu)
        return y_pred
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

- Using Models / Neural Network Modules

```
#N is batch size; D_in is input dimension;
#H is the dimension of the hidden layer; D_out is output dimension.
N, D_in, H, D_out = 32, 100, 50, 10

#Create random Tensors to hold inputs and outputs, and wrap them in Variables
x = Variable(torch.randn(N, D_in)) # dim: 32 x 100

#Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

#Forward pass: Compute predicted y by passing x to the model
y_pred = model(x) # dim: 32 x 10
```

https://cs230.stanford.edu/blog/pytorch/

The PyTorch framework

- Core Training Step

```
output_batch = model(train_batch) # compute model output
loss = loss_fn(output_batch, labels_batch) # calculate loss

#pick an SGD optimizer
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)

#or pick ADAM
optimizer = torch.optim.Adam(model.parameters(), lr = 0.0001)

optimizer.zero_grad() # clear previous gradients
loss.backward() # compute gradients of all variables wrt loss
optimizer.step() # perform updates using calculated gradients
```

https://cs230.stanford.edu/blog/pytorch/