OUTLINE

I. Introduction

II. Background

  • Multi-valued Multi-variate Functions
  • Tensors
  • Differential Calculus
  • The Chain Rule

III. Fitting a Model

IV. Supervised Learning

V. Unsupervised Learning

VI. Fantastic DNNs: How to choose them, how to train them
OUTLINE

I. Introduction
II. Background
III. Fitting a Model
IV. Supervised Learning
V. Unsupervised Learning
VI. Fantastic DNNs: How to choose them, how to train them
OUTLINE

I. Introduction
II. Background
III. Fitting a Model
   • How to minimize a function?
   • Backpropagation
   • Improved Gradient Descent
   • The PyTorch Framework
IV. Supervised Learning
V. Unsupervised Learning
VI. Fantastic DNNs: How to choose them, how to train them
Recap: the \textit{model fitting} approach to Machine Learning

- $f_\theta = \text{a jean}$
- $\theta = \text{its (width, length)}$
- $\mathcal{F} = \{f_\theta\}_{\theta \in \Theta} \text{ the shelves}$
Recap: the model fitting approach to Machine Learning

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

• Given a parameterized family $\mathcal{F}$ of models $==$ functions
III. Fitting a Model

Recap: the *model fitting* approach to Machine Learning

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

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

- Given a **parameterized family** \( \mathcal{F} \) of models == functions
  - **Ex:** a DNN with \( f_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^{N_{\text{out}}} \)

- Given a **training dataset** \( \mathcal{T} \) *(your legs!)*
Recap: the *model fitting* approach to Machine Learning

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

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

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

- 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 \( f_\theta \) to the **full dataset**, for the given task (the smaller the better),

- **How to minimize a function?**

  - a jean
  - its (width, length)
  - the shelves
III. Fitting a Model

Recap: the *model fitting* approach to Machine Learning

- Given a *parameterized family* $\mathcal{F}$ of models $\Rightarrow$ functions
  - Ex: a DNN with $f_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^{N_{\text{out}}}$

- 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 $f_\theta$ to the *full dataset*, for the given task (the smaller the better),

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

$$\hat{f} = f_{\hat{\theta}} \quad \text{where} \quad \hat{\theta} = \arg\min_{\theta \in \Theta} L(f_\theta, \mathcal{T})$$
III. Fitting a Model

How to minimize a function?

Recap: the *model fitting* approach to Machine Learning

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

- Given a **parameterized family** \( \mathcal{F} \) of models == functions
  
  **Ex:** a DNN with \( f_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^{N_{\text{out}}} \)

- 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 \( f_\theta \) to the **full dataset**, for the given task (the smaller the better),

→ We want to **minimize** the loss with respect to the **parameters** \( \theta \in \Theta \):

\[
\hat{f} = f_{\hat{\theta}} \quad \text{where} \quad \hat{\theta} = \arg\min_{\theta \in \Theta} L(f_\theta, \mathcal{T})
\]

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

Discrete: \( \theta \in \{ \theta_1, \ldots, \theta_C \} \)
III. Fitting a Model

Domain of the function

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

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

How to minimize a function?
III. Fitting a Model

How to minimize a function?

Domain of the function

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

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

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

Data misfit

$g(\theta)$

Local minimum

Global minimum

starting model 1

starting model 2

true model

Model $\theta$
III. Fitting a Model

How to minimize a function?

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: \( \theta \in \mathbb{R}^D \)
III. Fitting a Model

How to minimize a function?

Domain of the function

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

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

$g(\theta)$

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

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

2D continuous: $\theta \in \mathbb{R}^2$
1. Brute Force / Random / Grid Search

What is $g$? What is $\theta$? What is $\Theta$?

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

2. “Population-Based” Algorithms

- Evolutionary/Genetic algorithms
- Particle Swarms
- Ant Colonies

How to minimize a function?

What is $g$? What is $\theta$? What is $\Theta$?
III. Fitting a Model

2. “Population-Based” Algorithms

- Evolutionary/Genetic algorithms
- Particle Swarms
- Ant Colonies

What is \( g \) ? What is \( \theta \) ? What is \( \Theta \) ?

- 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 \( \theta_0 \in \mathbb{R}^P \) such that \( \nabla_x g(\theta_0) = 0_P \).
3. Calculating “zeroes” of the gradient

- We call **zero** of the gradient a point $\theta_0 \in \mathbb{R}^P$ such that $\nabla_x g(\theta_0) = 0_P$.

- Also called **stationary points** of $g$: the points where $g$ is **locally constant**, i.e., “flat”.
III. Fitting a Model

3. Calculating “ zeroes” of the gradient

- We call **zero** of the gradient a point \( \theta_0 \in \mathbb{R}^P \) such that \( \nabla_{\theta} g(\theta_0) = 0_P \).

- Also called **stationary points** of \( g \) : the points where \( g \) is **locally constant**, i.e., “flat”.

- They may correspond to:

  - [Diagram showing local minimum, local maximum, saddle point]
III. Fitting a Model

How to minimize a function?

3. Calculating “zeroes” of the gradient

- We call zero of the gradient a point $\theta_0 \in \mathbb{R}^P$ such that $\nabla_x g(\theta_0) = 0_P$.

- 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 $H_{\theta}[g](\theta_0) \in \mathbb{R}^{P \times P}$ of $g$ at $\theta_0$:

$$H_{\theta}[g](\theta_0) \overset{\text{def}}{=} J_{\theta}[\nabla_\theta g](\theta_0)$$

“Second order derivative of $g$”
III. Fitting a Model

3. Calculating “zeroes” of the gradient

• We call zero of the gradient a point $\theta_0 \in \mathbb{R}^P$ such that $\nabla x g(\theta_0) = 0_P$.

• Also called stationary points of $g$ : the points where $g$ is locally constant, i.e., “flat”.

• They may correspond to:

- local minimum: $H_\theta[g](\theta_0) > 0$
- local maximum: $H_\theta[g](\theta_0) < 0$
- saddle point: otherwise

• In case of doubt, it is possible to distinguish between the 3 by looking at the Hessian $H_\theta[g](\theta_0) \in \mathbb{R}^{P \times P}$ of $g$ at $\theta_0$:

$$H_\theta[g](\theta_0) \overset{\text{def}}{=} J_\theta[\nabla_\theta g](\theta_0)$$

“Second order derivative of $g$”
III. Fitting a Model

How to minimize a function?

3. Calculating “zeroes” of the gradient

- We call **zero** of the gradient a point \( \theta_0 \in \mathbb{R}^P \) such that \( \nabla_x g(\theta_0) = 0_P \).

- 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** \( H_\theta[g](\theta_0) \in \mathbb{R}^{P \times P} \) of \( g \) at \( \theta_0 \):
  \[
  H_\theta[g](\theta_0) \overset{\text{def}}{=} J_\theta[\nabla_\theta g](\theta_0)
  \]
  “Second order derivative of \( g \)”

  Only works if
  \[
  \text{Det } H_\theta[g](\theta_0) \neq 0
  \]
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: \( \mathcal{T} = \{(x_t, y_t)\}_{t=1}^T \)
III. Fitting a Model

► How to minimize a function?

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: $f_\theta(x) = ax + b$
- Parameters: $\theta = [a, b]^\top \in \mathbb{R}^2$
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: $f_\theta(x) = ax + b$
- Parameters: $\theta = [a, b]^\top \in \mathbb{R}^2$
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: $f_{\theta}(x) = ax + b$
- Parameters: $\theta = [a, b]^T \in \mathbb{R}^2$
III. Fitting a Model

艾滋 Intelligence & Deep Learning

3. Calculating “zeros” of the gradient

Exercise: Fitting an affine model via least squares

- Training set: \( \mathcal{T} = \{(x_t, y_t)\}_{t=1}^{T} \)
- Models: \( f_{\theta}(x) = ax + b \)
- Parameters: \( \theta = [a, b]^T \in \mathbb{R}^2 \)
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: $f_\theta(x) = ax + b$
- Parameters: $\theta = [a, b]^T \in \mathbb{R}^2$
- Total Loss: $g(\theta) = L(f_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^T (f_\theta(x_t) - y_t)^2$
3. Calculating “zeroes” of the gradient

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

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

Find $\theta_0$ such that $\nabla_\theta g(\theta_0) = 0_2$

$g(\theta) = ?$

$\nabla_\theta g(\theta_0) = ?$
III. Fitting a Model

3. Calculating “zeroes” of the gradient

Exercise: Fitting an affine model via least squares

\[ y = ax + b \]

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

- Find \( \theta_0 \) such that \( \nabla_{\theta} g(\theta_0) = 0_2 \)  
  Hint: we already calculated \( \nabla_{\theta} g(\theta_0) \) !

\[ g(\theta) = ? \]

\[ \nabla_{\theta} g(\theta_0) = ? \]
III. Fitting a Model

3. Calculating “zeroes” of the gradient

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

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

- Find \( \theta_0 \) such that \( \nabla_\theta g(\theta_0) = 0_2 \)  
  **Hint:** we already calculated \( \nabla_\theta g(\theta_0) \)!

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

\[
\nabla_\theta g(\theta_0) = ?
\]
III. Fitting a Model

How to minimize a function?

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

Find $\theta_0$ such that $\nabla_\theta g(\theta_0) = 0_2$  

**Hint:** we already calculated $\nabla_\theta g(\theta_0)$!

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

$$\nabla_\theta g(\theta_0) = ?$$


III. Fitting a Model

How to minimize a function?

3. Calculating “zeroes” of the gradient

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

\[ y = ax + b \]

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

- Find \( \theta_0 \) such that \( \nabla_\theta g(\theta_0) = 0_2 \)

**Hint:** we already calculated \( \nabla_\theta g(\theta_0) \)!

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

\( \nabla_\theta g(\theta_0) = ? \)
III. Fitting a Model

▷ How to minimize a function?

3. Calculating “zeroes” of the gradient

Exercise: Fitting an affine model via least squares

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

- Find \( \theta_0 \) such that \( \nabla_{\theta} g(\theta_0) = 0_2 \)

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

\( \nabla_{\theta} g(\theta_0) = ? \)
### III. Fitting a Model

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

• **Find** \( \theta_0 \) such that \( \nabla_\theta g(\theta_0) = 0_2 \)

**Hint:** we already calculated \( \nabla_\theta g(\theta_0) \) !

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

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

\[
\nabla_\theta g(\theta_0) = ?
\]
### 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:** \( f_{\theta}(x) = ax + b \)
- **Parameters:** \( \theta = [a, b]^T \in \mathbb{R}^2 \)
- **Total Loss:** \( g(\theta) = L(f_{\theta}, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} (f_{\theta}(x_t) - y_t)^2 \)

**Find** \( \theta_0 \) such that \( \nabla_\theta g(\theta_0) = 0_2 \quad \text{Hint: we already calculated } \nabla_\theta g(\theta_0) ! \)

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

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

\[
\nabla_\theta g(\theta_0) = 2\mathbf{W}^T (\mathbf{W} \theta_0 - \mathbf{y})
\]
III. Fitting a Model

▷ How to minimize a function?

3. Calculating “zeroes” of the gradient

Exercise: Fitting an affine model via least squares

\[
y = ax + b
\]

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

- Find \( \theta_0 \) such that \( \nabla_\theta g(\theta_0) = 0_2 \)

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

\[
= \frac{1}{T} \| W\theta - y \|_2^2, \quad \text{where} \quad W = [w_1^T, \ldots, w_T^T]^T \in \mathbb{R}^{T \times 2}, \quad y = [y_1, \ldots, y_T]^T \in \mathbb{R}^T
\]

\[
\nabla_\theta g(\theta_0) = 2W^T(W\theta_0 - y) = 0_2 \quad \Rightarrow \quad \theta_0 = (W^T W)^{-1} W^T y = W^+ y
\]
3. Calculating “zeros” of the gradient

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

\[ y = a_0 x + b_0 \]

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

**Hint:** we already calculated \( \nabla_\theta g(\theta_0) \)!

\[
\begin{align*}
g(\theta) &= \frac{1}{T} \sum_{t=1}^T (ax_t + b - y_t)^2 \\
&= \frac{1}{T} \sum_{t=1}^T \left( \begin{bmatrix} x_t, 1 \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix} - y_t \right)^2 \\
&= \frac{1}{T} \sum_{t=1}^T \left( \begin{bmatrix} a \\ b \end{bmatrix}^T \begin{bmatrix} x_t \\ 1 \end{bmatrix} - y_t \right)^2 \\
&= \frac{1}{T} \left\| \begin{bmatrix} w_t \end{bmatrix}^T \theta - y_t \right\|_2^2, \quad \text{where} \quad w_t \in \mathbb{R}^2
\end{align*}
\]

\[
\begin{align*}
\nabla_\theta g(\theta_0) &= 2W^T (W\theta_0 - y) = 0_2 \\
\Rightarrow \quad \theta_0 &= (W^T W)^{-1} W^T y = W^\dagger y
\end{align*}
\]
III. Fitting a Model

► How to minimize a function?

4. Alternated Minimization
III. Fitting a Model

4. Alternated Minimization

$$\arg\min_{\theta_1, \ldots, \theta_P} g(\theta_1, \ldots, \theta_P)?$$
III. Fitting a Model

4. Alternated Minimization

argmin_{\theta_1, \ldots, \theta_P} g(\theta_1, \ldots, \theta_P)?

\begin{align*}
\theta_1^{(i+1)} &= \arg\min_{\theta_1} g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\
\theta_2^{(i+1)} &= \arg\min_{\theta_2} g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\
& \quad \vdots \\
\theta_P^{(i+1)} &= \arg\min_{\theta_P} g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P)
\end{align*}
III. Fitting a Model

4. Alternated Minimization

\[ \text{argmin}_{\theta_1, \ldots, \theta_P} \ g(\theta_1, \ldots, \theta_P) ? \]

\[ \begin{align*}
\theta_1^{(i+1)} &= \underset{\theta_1}{\text{argmin}} \ g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\
\theta_2^{(i+1)} &= \underset{\theta_2}{\text{argmin}} \ g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\
& \vdots \\
\theta_P^{(i+1)} &= \underset{\theta_P}{\text{argmin}} \ g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P) 
\end{align*} \]

Converges, but not necessarily to the global minimum
III. Fitting a Model

4. Alternated Minimization

\[
\text{argmin}_{\theta_1, \ldots, \theta_P} g(\theta_1, \ldots, \theta_P) ?
\]

Converges, but not necessarily to the global minimum

\[
\begin{align*}
\theta_1^{(i+1)} &= \text{argmin}_{\theta_1} g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\
\theta_2^{(i+1)} &= \text{argmin}_{\theta_2} g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\
& \vdots \\
\theta_P^{(i+1)} &= \text{argmin}_{\theta_P} g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P)
\end{align*}
\]

- For \( \theta_p \) scalar: coordinate descent
III. Fitting a Model

4. Alternated Minimization

\[ \text{argmin}_{\theta_1, \ldots, \theta_P} g(\theta_1, \ldots, \theta_P) ? \]

\[ \begin{align*}
\theta_1^{(i+1)} &= \arg\min_{\theta_1} g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\
\theta_2^{(i+1)} &= \arg\min_{\theta_2} g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\
& \vdots \\
\theta_P^{(i+1)} &= \arg\min_{\theta_P} g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P) 
\end{align*} \]

Converges, but not necessarily to the global minimum

- For \( \theta_p \) scalar: coordinate descent

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

![Diagram showing the alternated minimization process](image)
III. Fitting a Model

4. Alternated Minimization

argmin $g(\theta_1, \ldots, \theta_P)$?

$\theta_1, \ldots, \theta_P$

Converges, but not necessarily to the global minimum

For $\theta_P$ 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

$\theta_1^{(i+1)} = \arg\min_{\theta_1} g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)})$

$\theta_2^{(i+1)} = \arg\min_{\theta_2} g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)})$

$\vdots$

$\vdots$

$\theta_P^{(i+1)} = \arg\min_{\theta_P} g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P)$
III. Fitting a Model

4. Alternated Minimization

argmin \( g(\theta_1, \ldots, \theta_P) \)?

\[
\begin{align*}
\theta_1^{(i+1)} &= \arg\min_{\theta_1} g(\theta_1, \theta_2^{(i)}, \ldots, \theta_P^{(i)}) \\
\theta_2^{(i+1)} &= \arg\min_{\theta_2} g(\theta_1^{(i+1)}, \theta_2, \ldots, \theta_P^{(i)}) \\
& \quad \vdots \\
\theta_P^{(i+1)} &= \arg\min_{\theta_P} g(\theta_1^{(i+1)}, \theta_2^{(i+1)}, \ldots, \theta_P)
\end{align*}
\]

• For \( \theta_P \) 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 $\theta^{(0)} \in \mathbb{R}^P$
• From here, follow the direction of steepest descent
• Stop when things look flat
III. Fitting a Model

5. Gradient Descent

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

\[-\nabla_{\theta} g(\theta^{(0)})\]
5. Gradient Descent

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

$-\nabla \theta g(\theta^{(0)})$
5. Gradient Descent

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

Animation by Andrew Ng
III. Fitting a Model

5. Gradient Descent

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

The updates are:
\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_\theta g(\theta^{(i)})
\]
III. Fitting a Model

5. Gradient Descent

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

\[ -\nabla_{\theta} g(\theta^{(0)}) \]

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

- Requires the function to be (almost everywhere) differentiable

► How to minimize a function?

Animation by Andrew Ng
5. Gradient Descent

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)}) \]

- Local maxima and saddle points are **unstable fixed points**, while local minima are **stable fixed points**.
III. Fitting a Model

5. Gradient Descent

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)}) \]

- 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** 😊
III. Fitting a Model

5. Gradient Descent

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

- 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** 😊

Ideally, we want to find the global minimum

Animation by Andrew Ng
III. Fitting a Model

5. Gradient Descent

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)}) \]

- 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 😊

Ideally, we want to find the global minimum. If not, this local minimum is not too bad.

Animation by Andrew Ng
III. Fitting a Model

5. Gradient Descent

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)}) \]

- 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** 😊

Ideally, we want to find the global minimum. If not, this local minimum is not too bad. This one performs poorly and should be avoided.

Animation by Andrew Ng
III. Fitting a Model

5. Gradient Descent

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} f(\theta^{(i)}) \]

- 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

Ideally, we want to find the global minimum

If not, this local minimum is not too bad

This one performs poorly and should be avoided
III. Fitting a Model

5. Gradient Descent

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

- 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
- Many variants have been derived to limit them, and to speed up convergence

Ideally, we want to find the global minimum

If not, this local minimum is not too bad

This one performs poorly and should be avoided

Animation by Andrew Ng
5. Gradient Descent

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla \theta g(\theta^{(i)}) \]

- The *gradient-step* \( \epsilon \), also called *learning rate*, is a critical *hyper-parameter* of the algorithm.
III. Fitting a Model

5. Gradient Descent

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)})
\]

- The \textit{gradient-step} \( \epsilon \), also called \textit{learning rate}, is a critical \textit{hyper-parameter} of the algorithm.

![Animation by Andrew Ng](image.png)

Image by Gabriel Peyre

Small \( \epsilon \)  
Large \( \epsilon \)  
Optimal \( \epsilon \)
5. Gradient Descent

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)}) \]

- The gradient-step \( \epsilon \), also called \textit{learning rate}, is a critical \textit{hyper-parameter} of the algorithm.

- Choosing a small \( \epsilon \) is always the \textit{safest}, but might result in \textit{slow} convergence.
III. Fitting a Model

5. Gradient Descent

How to minimize a function?

The gradient-step $\epsilon$, also called learning rate, is a critical hyper-parameter of the algorithm.

Choosing a small $\epsilon$ 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.
III. Fitting a Model

How to minimize a function?

Summary of optimization techniques
Summary of optimization techniques

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

How to minimize a function?

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*.
III. Fitting a Model

How to minimize a function?

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

\[
\begin{align*}
\hat{y} &= \text{dnn}_\theta(x)
\end{align*}
\]
Back to Neural Networks

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

*Remember:* A deep feedforward neural network

\[
x = \text{dnn}_\theta(x)
\]

\[
y = \sigma (b^4 + W^4 \sigma (b^3 + W^3 \sigma (b^2 + W^2 \sigma (b^1 + W^1 x))))
\]

\[
\theta = [\theta^i]_{i=1}^4 = [(b^i, W^i)]_{i=1}^4
\]
III. Fitting a Model

Back to Neural Networks

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

**Remember:** A deep feedforward neural network

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

$$ = \sigma(b^4 + W^4 \sigma(b^3 + W^3 \sigma(b^2 + W^2 \sigma(b^1 + W^1 x)))) $$

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

- Given a **training dataset** of input ↔ output $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^T$, the general goal is to adjust $\theta$ so that $y_t \approx \text{dnn}_\theta(x_t)$. 
III. Fitting a Model

Back to Neural Networks

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

*Remember:* A *deep feedforward neural network*

\[
x (\vdots) \mapsto y = \text{dnn}_\theta(x)
\]

\[
= \sigma (b^4 + W^4 \sigma (b^3 + W^3 \sigma (b^2 + W^2 \sigma (b^1 + W^1 x))))
\]

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

- Given a training dataset of input \(\leftrightarrow\) output \(\mathcal{T} = \{(x_t, y_t)\}_{t=1}^T\), the general goal is to adjust \(\theta\) so that \(y_t \approx \text{dnn}_\theta(x_t)\).

- We use a total loss of this form: 
  \[
  L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^T \ell (\text{dnn}_\theta(x_t) , y_t),
  \]
  where \(\ell\) 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

\[
x \xrightarrow{\cdot} y = \text{dnn}_\theta(x) = \sigma(b^4 + W^4 \sigma(b^3 + W^3 \sigma(b^2 + W^2 \sigma(b^1 + W^1 x))))
\]

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

- Given a training dataset of input ↔ output \( \mathcal{T} = \{(x_t, y_t)\}_{t=1}^T \), the general goal is to adjust \( \theta \) so that \( y_t \approx \text{dnn}_\theta(x_t) \).

- We use a total loss of this form: \( L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^T \ell(\text{dnn}_\theta(x_t); y_t) \), where \( \ell \) is simply called the loss of the DNN.

For example: \( \ell(\hat{y}, y) = \|\hat{y} - y\|_2^2 \), 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

Given a **training dataset** of input ↔ output $\mathcal{T} = \{(x_t, y_t)\}_{t=1}^T$, the general goal is to adjust $\theta$ so that $y_t \approx \text{dnn}_\theta(x_t)$.

We use a **total loss** of this form: $L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^T \ell(\text{dnn}_\theta(x_t), y_t)$, where $\ell$ is simply called the **loss** of the DNN.

*For example:* $\ell(\hat{y}, y) = \|\hat{y} - y\|_2^2$, 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 $\mathcal{R}(\text{dnn}_\theta) \overset{\text{def}}{=} \mathbb{E}_{X,Y} \{\ell(\text{dnn}_\theta(X), Y)\} \approx L(\text{dnn}_\theta, \mathcal{T})$. 

$\xrightarrow{}$ Backpropagation
Back to Neural Networks

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

**Remember:** A deep feedforward neural network

\[ y = \text{dnn}_\theta(x) \]

\[ = \sigma (b^4 + W^4 \sigma (b^3 + W^3 \sigma (b^2 + W^2 \sigma (b^1 + W^1 x)))) \]

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

- Given a training dataset of input ↔ output \( \mathcal{T} = \{(x_t, y_t)\}_{t=1}^T \), the general goal is to adjust \( \theta \) so that \( y_t \approx \text{dnn}_\theta(x_t) \).

- We use a total loss of this form: \( L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^T \ell(\text{dnn}_\theta(x_t), y_t) \), where \( \ell \) is simply called the loss of the DNN.

  For example: \( \ell(\hat{y}, y) = \| \hat{y} - y \|_2^2 \), 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 \( \mathcal{R}(\text{dnn}_\theta) \overset{\text{def}}{=} \mathbb{E}_{X,Y} \{ \ell(\text{dnn}_\theta(X), Y) \} \approx L(\text{dnn}_\theta, \mathcal{T}) \).

- We will focus next on supervised learning, but the approach is more general.
III. Fitting a Model

Back to Neural Networks

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

*Remember: A deep feedforward neural network*

\[
\begin{align*}
    \mathbf{y} &= \text{dnn}_\theta(\mathbf{x}) \\
    &= \sigma(\mathbf{b}^4 + \mathbf{W}^4 \sigma(\mathbf{b}^3 + \mathbf{W}^3 \sigma(\mathbf{b}^2 + \mathbf{W}^2 \sigma(\mathbf{b}^1 + \mathbf{W}^1 \mathbf{x})))) \\
    \theta &= [\theta^i]_{i=1}^4 = [(\mathbf{b}^i, \mathbf{W}^i)]_{i=1}^4
\end{align*}
\]

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

- We use a **total loss** of this form: \( L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \ell(\text{dnn}_\theta(\mathbf{x}_t), \mathbf{y}_t) \), where \( \ell \) is simply called the **loss** of the DNN.

  *For example: \( \ell(\mathbf{\hat{y}}, \mathbf{y}) = \|\mathbf{\hat{y}} - \mathbf{y}\|_2^2 \), 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 \( \mathcal{R}(\text{dnn}_\theta) \overset{\text{def}}{=} \mathbb{E}_{\mathbf{X}, \mathbf{Y}} \{\ell(\text{dnn}_\theta(\mathbf{X}), \mathbf{Y})\} \approx L(\text{dnn}_\theta, \mathcal{T}) \).

- 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:

\[ a^i = \text{aff}_{\theta^i}(x^{i-1}) = W^i x^{i-1} + b^i \] are the **pre-activations**.
The Backpropagation Algorithm

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

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

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

- \( a^i = \text{aff}_{\theta_i}(x^{i-1}) = W^i x^{i-1} + b^i \) are the *pre-activations*.
- \( x^i = \sigma(a^i) = \sigma(\text{aff}_{\theta_i}(x^{i-1})) = \text{layer}^i(x^{i-1}) \) are the *activations*.
- The **loss** \( \ell \) can be viewed as another layer, with *real output* \( r \) (the “*residual*”).
The Backpropagation Algorithm

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

- \( a^i = \text{aff}_{\theta^i}(x^{i-1}) = W^i x^{i-1} + b^i \) are the **pre-activations**.
- \( x^i = \sigma(a^i) = \sigma(\text{aff}_{\theta^i}(x^{i-1})) = \text{layer}^i(x^{i-1}) \) are the **activations**.
- The **loss** \( \ell \) can be viewed as another layer, with **real output** \( r \) (the “residual”).
- By **linearity** of the gradient, we have: \( \nabla_\theta L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \nabla_\theta \ell(\text{dnn}_\theta(x_t), y_t) \).
III. Fitting a Model

The Backpropagation Algorithm

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

- \( a^i = \text{aff}_{\theta_i}(x^{i-1}) = W^i x^{i-1} + b^i \) are the **pre-activations**.
- \( x^i = \sigma(a^i) = \sigma(\text{aff}_{\theta_i}(x^{i-1})) = \text{layer}^i(x^{i-1}) \) are the **activations**.
- The **loss** \( \ell \) can be viewed as another layer, with **real output** \( r \) (the “residual”).
- By **linearity** of the gradient, we have: \( \nabla_{\theta} L(dnn_{\theta}, T) = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\theta} \ell(dnn_{\theta}(x_t), y_t) \).
- Hence, it is enough to calculate the gradient of the loss for **one sample** \((x_t, y_t)\), i.e., \( G_{\theta} \overset{\text{def}}{=} \nabla_{\theta} \ell(dnn_{\theta}(x_t), y_t) \).
The Backpropagation Algorithm

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

- \( a^i = \text{aff}_{\theta^i}(x^{i-1}) = W^i x^{i-1} + b^i \) are the pre-activations.
- \( x^i = \sigma(a^i) = \sigma(\text{aff}_{\theta^i}(x^{i-1})) = \text{layer}^i(x^{i-1}) \) are the activations.
- The loss \( \ell \) can be viewed as another layer, with **real output** \( r \) (the “residual”).
- By linearity of the gradient, we have: \( \nabla_{\theta}L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \nabla_{\theta} \ell(\text{dnn}_\theta(x_t), y_t) \).
- Hence, it is enough to calculate the gradient of the loss for **one sample** \((x_t, y_t)\), i.e., \( G_\theta \overset{\text{def}}{=} \nabla_{\theta} \ell(\text{dnn}_\theta(x_t), y_t) \).
The Backpropagation Algorithm

Let's start by cleaning a bit the picture:

- $a^i = \text{aff}_{\theta^i}(x^{i-1}) = W^i x^{i-1} + b^i$ are the pre-activations.
- $x^i = \sigma(a^i) = \sigma(\text{aff}_{\theta^i}(x^{i-1}))$ are the activations.
- The loss $\ell$ can be viewed as another layer, with real output $r$ (the “residual”).
- By linearity of the gradient, we have: $\nabla_\theta L(dnn_\theta, T) = \frac{1}{T} \sum_{t=1}^T \nabla_\theta \ell(dnn_\theta(x_t), y_t)$.
- Hence, it is enough to calculate the gradient of the loss for one sample $(x_t, y_t)$, i.e., $G_\theta \overset{\text{def}}{=} \nabla_\theta \ell(dnn_\theta(x_t), y_t)$.
- 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 \( G_{x^4} = \frac{\partial r}{\partial x^4} \bigg|_{x_t^4} \)
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 \( G_{x^4} = \frac{\partial r}{\partial x^4} \bigg|_{x^4_t}^T = \nabla x^4 \ell(x^4_t, y_t) \).
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

\[
G_{x^4} = \frac{\partial r}{\partial x^4} \big|_{x^4_t} = \nabla x^4 \ell(x^4_t, y_t).
\]

For example, for the L2 loss \( \ell(x^4, y_t) = \| x^4 - y_t \|^2 \), we have \( G_{x^4} = 2(x^4_t - y_t) \), the **difference** between the network **prediction** and the **target**.
III. Fitting a Model

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 \( \mathbf{G}_{x^4} = \left. \frac{\partial r}{\partial x^4} \right|_{x_t^4}^\top = \nabla x^4 \ell(x^4_t, y_t) \).

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

1) Then, \( \mathbf{G}_{a^4} = \left. \frac{\partial r}{\partial a^4} \right|_{a_t^4}^\top \)
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 \( G_{x^4} = \frac{\partial r}{\partial x^4} \bigg|_{x^4_t} = \nabla x^4 \ell(x^4_t, y_t). \)

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

1) Then, \( G_{a^4} = \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} = \left( \frac{\partial r}{\partial x^4} \bigg|_{x^4_t} \times \frac{\partial x^4}{\partial a^4} \bigg|_{a^4_t} \right)^\top \)
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 \( G_{x^4} = \frac{\partial r}{\partial x^4} \bigg|_{x^4_t} = \nabla x^4 \ell(x^4_t, y_t) \).

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

1) Then, \( G_{a^4} = \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} = \left( \frac{\partial r}{\partial x^4} \bigg|_{x^4_t} \times \frac{\partial x^4}{\partial a^4} \bigg|_{a^4_t} \right)^\top = \frac{\partial x^4}{\partial a^4} \bigg|_{a^4_t}^\top \times G_{x^4} \)
III. Fitting a Model

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 \( G_{x^4} = \frac{\partial r}{\partial x^4} \bigg|_{x^4_t} = \nabla_x^4 \ell(x^4_t, y_t) \).

For example, for the L2 loss \( \ell(x^4, y_t) = \|x^4 - y_t\|_2^2 \), we have \( G_{x^4} = 2(x^4_t - y_t) \), the difference between the network prediction and the target.

1) Then, \( G_{a^4} = \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} = \left( \frac{\partial r}{\partial x^4} \bigg|_{x^4_t} \times \frac{\partial x^4}{\partial a^4} \bigg|_{a^4_t} \right)^\top = \frac{\partial x^4}{\partial a^4} \bigg|_{a^4_t} \times G_{x^4} \) diag[\( \sigma'(a^4_t) \)]!
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 \( G_{x^4} = \frac{\partial r}{\partial x^4} \bigg|_{x^4_t}^{\top} = \nabla x^4 \ell(x^4_t, y_t) \).

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

1) Then,  \( G_{a^4} = \frac{\partial r}{\partial a^4} \bigg|_{a^4_t}^{\top} = \left( \frac{\partial r}{\partial x^4} \bigg|_{x^4_t} \times \frac{\partial x^4}{\partial a^4} \bigg|_{a^4_t} \right)^{\top} = \frac{\partial x^4}{\partial a^4} \bigg|_{a^4_t}^{\top} \times G_{x^4} \)

\( \sigma'(a^4_t) \odot G_{x^4} \).
2) Then:

\[ G_{x^3} = \frac{\partial r}{\partial x^3} \bigg|_{x^3_{t}}^{\top} \]
The Backpropagation Algorithm

III. Fitting a Model

2) Then:

- $G_{x^3} = \frac{\partial r}{\partial x^3} \bigg|_{x^3_t}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t} \right)^\top$
III. Fitting a Model

The Backpropagation Algorithm

2) Then:

- \[ G_{x^3} = \left. \frac{\partial r}{\partial x^3} \right|_{x_t^3}^{\top} = \left( \frac{\partial r}{\partial a^4} \bigg|_{a_t^4} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x_t^3} \right)^{\top} = \frac{\partial a^4}{\partial x^3} \bigg|_{x_t^3}^{\top} \times G_{a^4} \]
The Backpropagation Algorithm

III. Fitting a Model

► Back Propagation

2) Then:

- \( G_{x^3} = \frac{\partial r}{\partial x^3} \bigg|_{x^3_t}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t} \right)^\top = \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t}^\top \times G_{a^4} \)

\[ a^1 = W^1x^0 + b^1 \]
\[ a^2 = W^2a^1 + b^2 \]
\[ a^3 = W^3a^2 + b^3 \]
\[ a^4 = W^4a^3 + b^4 \]
\[ x^0 = x_t 
\[ x^1 = \sigma(a^1) \]
\[ x^2 = \sigma(a^2) \]
\[ x^3 = \sigma(a^3) \]
\[ x^4 = \ell(a^4) \]
\[ y = \sigma(x^4) \]
\[ y_t = y \]
III. Fitting a Model

The Backpropagation Algorithm

2) Then:
\[ G_{x^3} = \frac{\partial r}{\partial x^3} \bigg|_{x^3_t}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t} \right)^\top \times \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t} \times G_{a^4} = W^4^\top G_{a^4}. \]
III. Fitting a Model

The Backpropagation Algorithm

2) Then:

\[ G_{x^3} = \left( \frac{\partial r}{\partial a^4} \right)^\top \times \left( \frac{\partial a^4}{\partial x^3} \right)^\top = \frac{\partial r}{\partial a^4} \times \frac{\partial a^4}{\partial x^3} \times G_{a^4} = W^4^\top G_{a^4} \]

\[ G_{b^4} = \left( \frac{\partial r}{\partial b^4} \right)^\top \]
III. Fitting a Model

The Backpropagation Algorithm

2) Then:

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

- \( G_{b^4} = \frac{\partial r}{\partial b^4} \bigg|_{b^4}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial b^4} \bigg|_{b^4} \right)^\top \)
III. Fitting a Model  

The Backpropagation Algorithm

2) Then:

\[ G_{x^3} = \frac{\partial r}{\partial x^3} \bigg|_{x^3_t}^{\top} = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t} \right)^{\top} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t}^{\top} \times G_{a^4} = W^4^{\top} G_{a^4}. \]

\[ G_{b^4} = \frac{\partial r}{\partial b^4} \bigg|_{b^4}^{\top} = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial b^4} \bigg|_{b^4} \right)^{\top} \times \frac{\partial a^4}{\partial b^4} \bigg|_{b^4}^{\top} \times G_{a^4}. \]
The Backpropagation Algorithm

III. Fitting a Model

2) Then:

\[ G_{x^3} = \frac{\partial r}{\partial x^3} \bigg|_{x_t^3}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a_t^4} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x_t^3} \right)^\top = \frac{\partial a^4}{\partial x^3} \bigg|_{x_t^3}^\top \times G_{a^4} = W^{4\top} G_{a^4}. \]

\[ G_{b^4} = \frac{\partial r}{\partial b^4} \bigg|_{b^4}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a_t^4} \times \frac{\partial a^4}{\partial b^4} \bigg|_{b^4} \right)^\top = \frac{\partial a^4}{\partial b^4} \bigg|_{b^4}^\top \times G_{a^4} = I . \]
The Backpropagation Algorithm

III. Fitting a Model

2) Then:

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

- \( G_{b^4} = \frac{\partial r}{\partial b^4} \bigg|_{b^4}^{\top} = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial b^4} \bigg|_{b^4} \right)^\top = \frac{\partial a^4}{\partial b^4} \bigg|_{b^4}^{\top} \times G_{a^4} = G_{a^4}. \)
III. Fitting a Model

The Backpropagation Algorithm

2) Then:

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

- \( G_{b^4} = \left( \frac{\partial r}{\partial b^4} \right) \Bigg|_{b^4} = \left( \frac{\partial r}{\partial a^4} \right) \Bigg|_{a^4_t} \times \left( \frac{\partial a^4}{\partial b^4} \right) \Bigg|_{b^4} = \left( \frac{\partial r}{\partial a^4} \right) \Bigg|_{a^4_t} \times G_{a^4} = G_{a^4} \).

- \( G_{w^4_d} = \left( \frac{\partial r}{\partial w^4_d} \right) \Bigg|_{w^4_d} \)
III. Fitting a Model

The Backpropagation Algorithm

2) Then:

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

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

- \( G_{w^4_d} = \frac{\partial r}{\partial w^4_d} \bigg|_{w^4_d} ^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial w^4_d} \bigg|_{w^4_d} \right) ^\top \)
III. Fitting a Model

The Backpropagation Algorithm

2) Then:

- \( G_{x^3} = \frac{\partial r}{\partial x^3} \bigg|_{x_i^3} = \left( \frac{\partial r}{\partial a^4} \bigg|_{a_i^4} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x_i^3} \right) = \frac{\partial a^4}{\partial x^3} \bigg|_{x_i^3} \times G_{a^4} = W_{x^4}^T G_{a^4} \).

- \( G_{b^4} = \frac{\partial r}{\partial b^4} \bigg|_{b^4} = \left( \frac{\partial r}{\partial a^4} \bigg|_{a_i^4} \times \frac{\partial a^4}{\partial b^4} \bigg|_{b^4} \right) = \frac{\partial a^4}{\partial b^4} \bigg|_{b^4} \times G_{a^4} = G_{a^4} \).

- \( G_{w^4_d} = \frac{\partial r}{\partial w^4_d} \bigg|_{w^4_d} = \left( \frac{\partial r}{\partial a^4} \bigg|_{a_i^4} \times \frac{\partial a^4}{\partial w^4_d} \bigg|_{w^4_d} \right) = \frac{\partial a^4}{\partial w^4_d} \bigg|_{w^4_d} \times G_{a^4} \).
III. Fitting a Model

The Backpropagation Algorithm

2) Then:
   - \( G_{x^3} = \frac{\partial r}{\partial x^3} \bigg|_{x^3_t}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t} \right)^\top = \frac{\partial a^4}{\partial x^3} \bigg|_{x^3_t}^\top \times G_{a^4} = W^4^\top G_{a^4}. \)
   - \( G_{b^4} = \frac{\partial r}{\partial b^4} \bigg|_{b^4}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial b^4} \bigg|_{b^4} \right)^\top = \frac{\partial a^4}{\partial b^4} \bigg|_{b^4}^\top \times G_{a^4} = G_{a^4}. \)
   - \( G_{w^4_d} = \frac{\partial r}{\partial w^4_d} \bigg|_{w^4_d}^\top = \left( \frac{\partial r}{\partial a^4} \bigg|_{a^4_t} \times \frac{\partial a^4}{\partial w^4_d} \bigg|_{w^4_d} \right)^\top = \frac{\partial a^4}{\partial w^4_d} \bigg|_{w^4_d}^\top \times G_{a^4}. \)
III. Fitting a Model

The Backpropagation Algorithm

2) Then:

- \( G_{x^3} = \left. \frac{\partial r}{\partial x^3} \right|_{x^3_t} \) = \( \left( \frac{\partial r}{\partial a^4} \right|_{a^4_t} \times \frac{\partial a^4}{\partial x^3} \right|_{x^3_t} \) = \( \left. \frac{\partial a^4}{\partial x^3} \right|_{x^3_t} \times G_{a^4} = W^4^T G_{a^4} \) .

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

- \( G_{w^4_d} = \left. \frac{\partial r}{\partial w^4_d} \right|_{w^4_d} \) = \( \left( \frac{\partial r}{\partial a^4} \right|_{a^4_t} \times \frac{\partial a^4}{\partial w^4_d} \right|_{w^4_d} \) = \( \left. \frac{\partial a^4}{\partial w^4_d} \right|_{w^4_d} \times G_{a^4} = x^3_{t,d} G_{a^4} \) .
The Backpropagation Algorithm

And so on…
The Backpropagation Algorithm

III. Fitting a Model

► Back Propagation

And so on…

3) $G_{\alpha^3} = \sigma'(a^3_t) \odot G_{x^3}$. 
III. Fitting a Model

The Backpropagation Algorithm

And so on...

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

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

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

$G_{w^3} = x_{t,d}^2 G_{a^3}$.
III. Fitting a Model

The Backpropagation Algorithm

And so on…

3) \( G_{\alpha^3} = \sigma'(a_t^3) \odot G_{x^3} \).

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

4) \( G_{x^2} = W^3 \top G_{\alpha^3} \).

\( G_{b^3} = G_{\alpha^3} \).

\( G_{w_d^3} = x_{t,d}^2 G_{\alpha^3} \).
III. Fitting a Model

The Backpropagation Algorithm

And so on…

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

4) \( G_{x^2} = W^3^\top G_{a^3} \)

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

6) \( G_{x^1} = W^2^\top G_{a^2} \)

\( G_{b^3} = G_{a^3} \)

\( G_{w^3} = x_{t,d}^2 G_{a^3} \)

\( G_{b^2} = G_{a^2} \)

\( G_{w^2} = x_{t,d}^1 G_{a^2} \)
III. Fitting a Model

The Backpropagation Algorithm

\[ x_t \rightarrow x^0 \rightarrow \text{aff} \rightarrow a^1 \rightarrow \sigma \rightarrow x^1 \rightarrow \text{aff} \rightarrow a^2 \rightarrow \sigma \rightarrow x^2 \rightarrow \text{aff} \rightarrow a^3 \rightarrow \sigma \rightarrow x^3 \rightarrow \text{aff} \rightarrow a^4 \rightarrow \sigma \rightarrow x^4 \rightarrow \ell \quad r \]

\[ y_t \]

And so on…

3) \[ G_{a^3} = \sigma'(a_t^3) \odot G_{x^3} \]

5) \[ G_{a^2} = \sigma'(a_t^2) \odot G_{x^2} \]

7) \[ G_{a^1} = \sigma'(a_t^1) \odot G_{x^1} \]

4) \[ G_{x^2} = W^3\top G_{a^3} \]

6) \[ G_{x^1} = W^2\top G_{a^2} \]

\[ G_{b^3} = G_{a^3} \]

\[ G_{b^2} = G_{a^2} \]

\[ G_{w_3^d} = x^2_{t,d} G_{a^3} \]

\[ G_{w_2^d} = x^1_{t,d} G_{a^2} \]
The Backpropagation Algorithm

III. Fitting a Model

And so on…

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

4) \( G_{x^2} = W^3 \top G_{a^3} \)

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

6) \( G_{x^1} = W^2 \top G_{a^2} \)

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

8) \( G_{x^0} = W^1 \top G_{a^1} \)

\( G_{b^3} = G_{a^3} \)

\( G_{b^2} = G_{a^2} \)

\( G_{b^1} = G_{a^1} \)

\( G_{w_d^3} = x_{t,d}^2 G_{a^3} \)

\( G_{w_d^2} = x_{t,d}^1 G_{a^2} \)

\( G_{w_d^1} = x_{t,d}^0 G_{a^1} \)
The Backpropagation Algorithm

III. Fitting a Model

And so on…

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

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

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

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

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

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

In Fine:

$G_{x^0} = W^1 \top \sigma'(a_t^1) \odot W^2 \top \sigma'(a_t^2) \odot W^3 \top \sigma'(a_t^3) \odot W^4 \top \sigma'(a_t^4) \odot \nabla x^4 \ell(x_t^4, y_t)$
III. Fitting a Model

The Backpropagation Algorithm

And so on…

3) \( G_{a^3} = \sigma'(a_t^3) \circ G_{x^3} \)
5) \( G_{a^2} = \sigma'(a_t^2) \circ G_{x^2} \)
7) \( G_{a^1} = \sigma'(a_t^1) \circ G_{x^1} \)

4) \( G_{x^2} = W^{3\top} G_{a^3} \)
6) \( G_{x^1} = W^{2\top} G_{a^2} \)
8) \( G_{x^0} = W^{1\top} G_{a^1} \)

\( G_{b^3} = G_{a^3} \)
\( G_{w_d^3} = x_t^2 G_{a^3} \)
\( G_{b^2} = G_{a^2} \)
\( G_{w_d^2} = x_t^1 G_{a^2} \)
\( G_{b^1} = G_{a^1} \)
\( G_{w_d^1} = x_t^0 G_{a^1} \)

In Fine:

\( G_{x^0} = W^{1\top} \sigma'(a_t^1) \circ W^{2\top} \sigma'(a_t^2) \circ W^{3\top} \sigma'(a_t^3) \circ W^{4\top} \sigma'(a_t^4) \circ \nabla_x^4 \ell(x_t^4, y_t) \)

\( \Rightarrow \) Using the parameters’ gradients, we update them via gradient descent.
III. Fitting a Model

The Backpropagation Algorithm

Conclusions
The Backpropagation Algorithm

Conclusions

- The idea is very general and can be applied to any feedforward neural network architecture
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)
III. Fitting a Model

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

• Remember that we train DNNs using Empirical Risk Minimization:

$$L(dnn_{\theta}, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \ell(dnn_{\theta}(x_t), y_t) \approx \mathbb{E}_{X,Y} \{ \ell(dnn_{\theta}(X), Y) \}$$
Back to gradient descent

- Remember that we train DNNs using Empirical Risk Minimization:

$$L(dnn_\theta, T) = \frac{1}{T} \sum_{t=1}^{T} \ell(dnn_\theta(x_t), y_t) \approx \mathbb{E}_{X,Y} \{\ell(dnn_\theta(X), Y)\}$$

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

$$\nabla_\theta L(dnn_\theta, T) = \frac{1}{T} \sum_{t=1}^{T} \nabla_\theta \ell(dnn_\theta(x_t), y_t)$$

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_\theta g(\theta^{(i)})$$
III. Fitting a Model

Back to gradient descent

• Remember that we train DNNs using Empirical Risk Minimization:

\[ L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \ell(\text{dnn}_\theta(x_t), y_t) \approx \mathbb{E}_{X,Y} \{ \ell(\text{dnn}_\theta(X), Y) \} \]

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

\[ \nabla_\theta L(\text{dnn}_\theta, \mathcal{T}) = \frac{1}{T} \sum_{t=1}^{T} \nabla_\theta \ell(\text{dnn}_\theta(x_t), y_t) \]

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

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(x_t, y_t) \in \mathcal{T}_i} \nabla_{\theta} \ell(d_{nn}(x_t), y_t)
\]
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:

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)})
\]

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

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)})
\]

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \cdot \frac{1}{|\mathcal{T}|} \sum_{(x_t, y_t) \in \mathcal{T}_i} \nabla_{\theta} \ell(d_{nn}(x_t), y_t)
\]

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

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \sum_{(x_t, y_t) \in \mathcal{T}_i} \nabla_{\theta} \ell(dnn_{\theta}(x_t), y_t)
\]

- 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
Stochastic Gradient Descent
(The SGD algorithm)

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

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)})
\]

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

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)})
\]

- 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
(The SGD algorithm)

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

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(x_t, y_t) \in T_i} \nabla \theta \ell(dnn_\theta(x_t), y_t)
\]

- At the next iteration, pick another (disjoint) random subset
- When the entire dataset has passed, start over again
- Each \(T^{(i)}\) 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.
Stochastic Gradient Descent
(The SGD algorithm)

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

\[
\theta^{(i+1)} = \theta^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(x_t, y_t) \in T_i} \nabla \ell(dnn_\theta(x_t), y_t)
\]

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

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)}) \]
Stochastic Gradient Descent
(The SGD algorithm)

- In practice the gradients $\ell(\text{dnn}_\theta(x_t), y_t)$ of all examples $(x_t, y_t)$ are computed in parallel using a graphical processing unit (GPU) and summed up within a minibatch.
Stochastic Gradient Descent (The SGD algorithm)

- In practice the gradients $\ell(\text{dnn}_\theta(x_t), y_t)$ of all examples $(x_t, y_t)$ 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.

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)})
\]
Stochastic Gradient Descent
(The SGD algorithm)

- In practice the gradients $\ell(\text{dnn}_\theta(x_t), y_t)$ of all examples $(x_t, y_t)$ 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.

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)})$$
III. Fitting a Model

Stochastic Gradient Descent (The SGD algorithm)

- In practice the gradients $\ell(\text{dnn}_\theta(x_t), y_t)$ of all examples $(x_t, y_t)$ 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.

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_\theta g(\theta^{(i)}) \]
III. Fitting a Model

SGD with Momentum

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla \theta g(\theta^{(i)})
\]
III. Fitting a Model

SGD with Momentum

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

\[ \theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)}) \]
III. Fitting a Model

SGD with Momentum

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

\[
\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla \theta g(\theta^{(i)})
\]
III. Fitting a Model

SGD with Momentum

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

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \epsilon \nabla_{\theta} g(\theta^{(i)})$$
III. Fitting a Model

SGD with Momentum

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

- **Momentum** = vector $\mathbf{v}$ 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 $\mathbf{v}^{(0)} = 0$, then replace each iteration of SGD by:

  \[
  \begin{align*}
  \mathbf{v}^{(i+1)} &\leftarrow \alpha \mathbf{v}^{(i)} - \epsilon \cdot \frac{1}{T} \sum_{(x_t, y_t) \in T_i} \nabla_\theta \ell(\text{dnn}_\theta(x_t), y_t) \\
  \theta^{(i+1)} &\leftarrow \theta^{(i)} + \mathbf{v}^{(i)}
  \end{align*}
  \]
### SGD with Momentum

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

- **Momentum** = vector $\mathbf{v}$ 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 $\mathbf{v}^{(0)} = 0$, then replace each iteration of SGD by:

  $$
  \begin{align*}
  \mathbf{v}^{(i+1)} &\leftarrow \alpha \mathbf{v}^{(i)} - \epsilon \frac{1}{T} \sum_{(x_t, y_t) \in T_i} \nabla \theta \ell (\text{dnn}_\theta (x_t), y_t) \\
  \theta^{(i+1)} &\leftarrow \theta^{(i)} + \mathbf{v}^{(i)}
  \end{align*}
  $$

  **Converges faster than SGD**
III. Fitting a Model

SGD with Momentum

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

- **Momentum** = vector $\mathbf{v}$ 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 $\mathbf{v}^{(0)} = 0$, then replace each iteration of SGD by:

$$
\begin{aligned}
\mathbf{v}^{(i+1)} &\leftarrow \alpha \mathbf{v}^{(i)} - \epsilon \frac{1}{T} \sum_{(x_t,y_t) \in \mathcal{T}_i} \nabla_\theta \ell(\text{dnn}_\theta(x_t), y_t) \\
\theta^{(i+1)} &\leftarrow \theta^{(i)} + \mathbf{v}^{(i)}
\end{aligned}
$$

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

Converges faster than SGD
III. Fitting a Model

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 not change the value of the loss):
III. Fitting a Model

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 \(\beta\) and its outgoing weights by \(1/\beta\).
III. Fitting a Model

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 not 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 $\beta$ 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
GOOD NEWS: You (probably) won’t ever need to implement backpropagation or SGD yourself!
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.
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…
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.
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.
III. Fitting a Model

The PyTorch framework
Overview of differential Programming

$\mathbf{x}_t \xrightarrow{W^1, b^1} \mathbf{a}_t^1 \xrightarrow{\sigma} \mathbf{x}_t^1 \xrightarrow{W^2, b^2} \mathbf{a}_t^2 \xrightarrow{\sigma} \mathbf{x}_t^2 \xrightarrow{W^3, b^3} \mathbf{a}_t^3 \xrightarrow{\sigma} \mathbf{x}_t^3 \xrightarrow{W^4, b^4} \mathbf{a}_t^4 \xrightarrow{\sigma} \mathbf{x}_t^4 \xrightarrow{\ell} r$

$\mathbf{y}_t$
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
III. Fitting a Model

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 \textit{acyclic computational flow graph}
- Associate each box with a \textit{forward} method, that computes the value of the box given its children
- Call the \textit{forward} method of each box in \textit{left->right} order

\textbf{Similarly for backpropagation:}
- Associate each box with a \textit{backward} method, that computes the gradient with respect to each child box
III. Fitting a Model

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

- Tensors (Data)

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

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

- Tensors (Data)

```python
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

```python
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

```python
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

```python
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_rijk = 2*a_rijk for a_11, a_12, a21, a22

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

https://cs230.stanford.edu/blog/pytorch/
III. Fitting a Model

The PyTorch framework

- Loss

```python
loss_fn = nn.CrossEntropyLoss()
loss = loss_fn(out, target)
```
The PyTorch framework

• Loss

```python
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

- Models / Neural Network Modules

```python
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

```python
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
The PyTorch framework

• Core Training Step

```python
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/