AI Applications Lecture 3
Functions Represented by Neural Networks

SUZUKI, Atsushi
Jing WANG

2025-09-01

Contents

1 Introduction **2**
1.1 Review of the Previous Lecture 2
1.2 Learning Outcomes of This Lecture 2
1.3 Policy of This Lecture 2

2 Preparation: Mathematical Notations **3**

3 What is a Neural Network? **5**

4 Directed Acyclic Graph (DAG) **5**

5 The Function Represented by a Neural Network **7**
5.1 Specific Computation Examples 10

6 Architecture and Checkpoints **25**

7 Summary and Future Outlook **26**
7.1 Today's Summary 26
7.2 Preview of the Next Lecture 27

A Constraints on Activation Functions **28**

1 Introduction

1.1 Review of the Previous Lecture

In the last lecture, we saw that many machine learning models, including neural networks, can be understood within a unified framework.

- Many machine learning models can be formulated as a **parametric function** $\{f_\theta\}$, where the specific computation is determined by the values of parameters θ.
- The process of solving a task is divided into two phases.
  - **Training:** Using data to find the parameter θ^* that best solves the task.
  - **Inference:** Using the trained parameter θ^* to provide a new input to the function f_{θ^*} and obtain an output.

1.2 Learning Outcomes of This Lecture

Through this lecture, students will aim to be able to perform the following tasks:

- Describe in mathematical terms the function represented by a neural network defined by a general **Directed Acyclic Graph (DAG)**, not limited to specific structures like Multilayer Perceptrons, CNNs, or Transformers.
- State the definitions of frequently used terms in modern neural network applications, such as **architecture**, **checkpoint**, and **model**.
- Explain why the distinction between architecture and checkpoint is important for utilizing AI in real-world applications.

1.3 Policy of This Lecture

Although many students may have already studied neural networks, this lecture will deliberately start from their mathematical definition. The goal is to **define neural networks in the most general form possible**.

The field of AI and machine learning is developing very rapidly, and the specific models widely known today (e.g., famous architectures like ResNet or Transformer) may well be outdated by the time you graduate. Therefore, this lecture emphasizes understanding more general and universal concepts that will be applicable in the future and will not become obsolete, rather than learning individual technologies. To this end, we will introduce a mathematical framework for viewing neural networks as a broad class of functions, not limited to specific structures.

On the other hand, one might ask, ”Is a fundamental understanding of neural networks necessary if we are to use them as black boxes?” The answer to this question is ”Yes.” In

particular, the **distinction between architecture and checkpoint** that we will learn in this lecture is extremely important for using AI in the real world. Without understanding this distinction, you will not be able to correctly understand the issue of AI **licenses**, which we will cover in the next lecture. Using AI without regard for its license can lead to legal risks such as copyright infringement, making this understanding essential.

2 Preparation: Mathematical Notations

Here is a review of the basic mathematical notations used in this lecture.

- **Definition:**
  - $(\mathrm{LHS}) \coloneqq (\mathrm{RHS})$: Indicates that the left-hand side is defined by the right-hand side. For example, $a \coloneqq b$ indicates that a is defined as b.
- **Set:**
  - Sets are often denoted by uppercase calligraphic letters. E.g., $\mathcal{A}$.
  - $x \in \mathcal{A}$: Indicates that the element x belongs to the set $\mathcal{A}$.
  - $\{\}$: The empty set.
  - $\{a, b, c\}$: The set consisting of elements a, b, c (set-builder notation).
  - $\{x \in \mathcal{A} | P(x)\}$: The set of elements in $\mathcal{A}$ for which the proposition $P(x)$ is true (set-builder notation).
  - $\mathbb{R}$: The set of all real numbers.
  - $\mathbb{R}_{>0}$: The set of all positive real numbers.
  - $\mathbb{R}_{\geq 0}$: The set of all non-negative real numbers.
  - $\mathbb{Z}$: The set of all integers.
  - $\mathbb{Z}_{>0}$: The set of all positive integers.
  - $\mathbb{Z}_{\geq 0}$: The set of all non-negative integers.
  - $[1, k]_{\mathbb{Z}} \coloneqq \{1, 2, \ldots, k\}$: For a positive integer k, the set of integers from 1 to k.
- **Function:**
  - $f : X \to Y$: Indicates that the function f is a map that takes an element of set X as input and outputs an element of set Y.
  - $y = f(x)$: Indicates that the output of the function f for an input $x \in X$ is $y \in Y$.
- **Vector:**
  - In this course, a vector refers to a column of numbers arranged vertically.

- Vectors are denoted by bold italic lowercase letters. E.g., $\boldsymbol{v}$.
- $\boldsymbol{v} \in \mathbb{R}^n$: Indicates that the vector $\boldsymbol{v}$ is an n-dimensional real vector.
- The i-th element of a vector $\boldsymbol{v}$ is denoted as v_i.

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}. \tag{1}$$

- **Matrix:**
  - Matrices are denoted by bold italic uppercase letters. E.g., $\boldsymbol{A}$.
  - $\boldsymbol{A} \in \mathbb{R}^{m,n}$: Indicates that the matrix $\boldsymbol{A}$ is an $m \times n$ real matrix.
  - The element in the i-th row and j-th column of a matrix $\boldsymbol{A}$ is denoted as $a_{i,j}$.

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}. \tag{2}$$

  - The transpose of a matrix $\boldsymbol{A}$ is denoted as $\boldsymbol{A}^\top$. If $\boldsymbol{A} \in \mathbb{R}^{m,n}$, then $\boldsymbol{A}^\top \in \mathbb{R}^{n,m}$, and

$$\boldsymbol{A}^\top = \begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{m,n} \end{bmatrix} \tag{3}$$

    is true.
  - A vector is a matrix with 1 column, and its transpose can also be defined.

$$\boldsymbol{v}^\top = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} \in \mathbb{R}^{1,n} \tag{4}$$

    is true.

- **Tensor:**
  - In this lecture, the word tensor simply refers to a multi-dimensional array. A vector can be considered a 1st-order tensor, and a matrix a 2nd-order tensor. Tensors of 3rd order or higher are denoted by underlined bold italic uppercase letters, like $\underline{\boldsymbol{A}}$.
  - Students who have already learned about abstract tensors in mathematics or physics may feel uncomfortable calling a mere multi-dimensional array a tensor.

If we consider the basis to be always fixed to the standard basis and identify the mathematical meaning of a tensor with its component representation (which becomes a multi-dimensional array), then the terminology is (at least) consistent.

3 What is a Neural Network?

An **Artificial Neural Network (ANN)** is a type of parametric function. We will now define, in a sequential and general manner, what kind of correspondence it has between parameters and functions.

Remark 3.1 (On Terminology)**.** Originally, "neural network" refers to the neural circuitry of living organisms, and engineering models should be distinguished by being called "artificial neural networks." Historically, the latter was inspired by imitating the former. However, in the modern field of AI, the similarity between the two is not essential, and it is easier to understand them as independent concepts. In this lecture, when we say "neural network" without qualification, it shall refer to an artificial neural network.

Remark 3.2 (Scope of this Lecture)**.** All neural networks covered in this lecture belong to a class called **feedforward neural networks**. This includes Multilayer Perceptrons (MLP), Convolutional Neural Networks (CNN), and Transformers. Examples of non-feedforward networks include Boltzmann machines and Hopfield networks, but since their applications are limited in the 2020s, they will not be covered in this lecture.

A feedforward neural network, in a nutshell, is a **parametric function defined by a directed acyclic graph (DAG) where each node has a fixed function, and many edges are associated with parameters (real values)**. To understand this definition, let's first look at its component, the directed acyclic graph.

4 Directed Acyclic Graph (DAG)

To define the structure of a neural network, we first define its backbone, a directed graph. In particular, we will consider a **multigraph**, which allows multiple edges between the same nodes.

Definition 4.1 (Directed Multigraph)**.** A **directed multigraph** G is defined by a tuple $G = (\mathcal{V}, \mathcal{E}, \mathrm{Tail}, \mathrm{Head})$, which consists of a set of **nodes** or **vertices** $\mathcal{V}$, a set of **edges** or **directed arcs** $\mathcal{E}$, and two maps $\mathrm{Tail} : \mathcal{E} \to \mathcal{V}$ and $\mathrm{Head} : \mathcal{E} \to \mathcal{V}$. For an edge $e \in \mathcal{E}$, $\mathrm{Tail}(e)$ is called its **tail node**, and $\mathrm{Head}(e)$ is called its **head node**.

Using these terms, we define the relationships between nodes.

- For a given node $v \in \mathcal{V}$, the set of edges e such that $\mathrm{Head}(e) = v$ is called the **incoming edges** of v, and is written as $\mathcal{E}_{\mathrm{in}}^{(v)}$.

$$\mathcal{E}_{\mathrm{in}}^{(v)} = \mathrm{Head}^{-1}(\{v\}) = \{e \in \mathcal{E} \mid \mathrm{Head}(e) = v\} \tag{5}$$

- Conversely, the set of edges e such that $\mathrm{Tail}(e) = v$ is called the **outgoing edges** of v, and is written as $\mathcal{E}_{\mathrm{out}}^{(v)}$.

$$\mathcal{E}_{\mathrm{out}}^{(v)} = \mathrm{Tail}^{-1}(\{v\}) = \{e \in \mathcal{E} \mid \mathrm{Tail}(e) = v\} \tag{6}$$

- For a node v, if there exists an edge $e \in \mathcal{E}_{\mathrm{in}}^{(v)}$, its tail node $\mathrm{Tail}(e)$ is called a **parent node** of v.
- For a node v, if there exists an edge $e \in \mathcal{E}_{\mathrm{out}}^{(v)}$, its head node $\mathrm{Head}(e)$ is called a **child node** of v.

In short, **parent node/child node** expresses the relationship from a node-centric perspective, while **tail node/head node** are terms that refer to the endpoints from an edge-centric perspective.

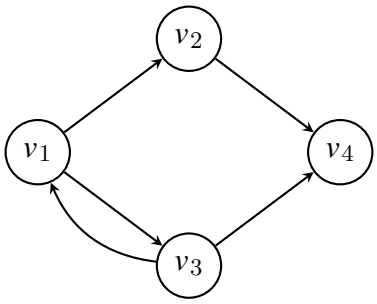

Figure 1: Example of a directed graph. $\mathcal{V} = \{v_1, v_2, v_3, v_4\}$, $\mathcal{E} = \{e_1, e_2, e_3, e_4, e_5\}$, and for example, $\mathrm{Tail}(e_1) = v_1, \mathrm{Head}(e_1) = v_2$.

Definition 4.2 (Path and Cycle)**.** In a directed multigraph G,

- A **path** is a pair of a sequence of nodes $v_0, v_1, \ldots, v_k$ and a sequence of edges $e_1, \ldots, e_k$ such that for all $i \in \{1, \ldots, k\}$, $\mathrm{Tail}(e_i) = v_{i-1}$ and $\mathrm{Head}(e_i) = v_i$. The length of this path is k.
- A **cycle** is a path whose start and end points are the same, i.e., a path where $v_0 = v_k$.

Definition 4.3 (Directed Acyclic Graph)**.** When a directed multigraph G has **no cycles**, it is called a **Directed Acyclic Graph (DAG)**.

Intuitively, a DAG is a graph structure where only ”one-way” flows exist. Starting from a node and following the edges, you can never return to the same place. For this reason, in a DAG, the direction of edges can be seen as a ”dependency” (the head node depends on the tail node). This property allows computations on a DAG to be defined and executed recursively, starting from the ends of the dependencies (nodes without parents). This is equivalent to the graph being **topologically sortable**. In other words, we can assign numbers to all nodes $v_1, v_2, \ldots, v_N$ such that for any edge $e \in \mathcal{E}$, if $\mathrm{Tail}(e) = v_i$ and $\mathrm{Head}(e) = v_j$, then $i < j$ always holds.

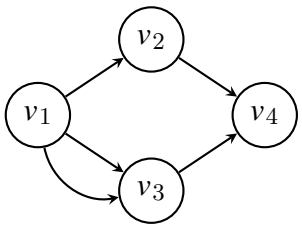

Figure 2: Example of a DAG (including multi-edges)

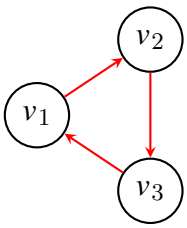

Figure 3: Example that is not a DAG (a cycle $v_1 \to v_2 \to v_3 \to v_1$ exists)

In a DAG, there is always at least one node that has no incoming edges ($\mathcal{E}_{\text{in}}^{(v)} = \emptyset$). These nodes become the **input nodes** that receive the input when the neural network is viewed as a function.

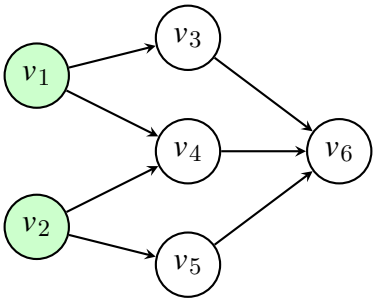

Figure 4: Example of a DAG with complex dependencies. v_1, v_2 are the input nodes.

5 The Function Represented by a Neural Network

Now, we will define the function represented by a neural network using a DAG. The rigorous definition may seem complex, but the basic idea is simple.

A neural network can be seen as a collection of many simple computational units (nodes and edges) connected in a DAG structure. Each unit has a simple function: ”receive values from sources, apply its own processing, and pass the values to destinations.” Mathematically, each unit is a function, and the entire network can be interpreted as a massive composite function obtained by applying these functions in topological sort order.

- **Role of Nodes:**
  - **Source:** Its parent nodes (more precisely, the incoming edges for which it is the head node), or a part of the neural network’s own **input**.
  - **Own Processing:** Each node has its own function called an **activation function** (generally multi-variable input, multi-variable output).
  - **Processing Content:** Applies the activation function to the values received from its sources (**arguments**) and calculates a new value (**return value**).

  - **Destination:** Its child nodes (more precisely, the outgoing edges for which it is the tail node), or a part of the neural network's own **output**.

- **Role of Edges:**
  - **Source:** Its tail node.
  - **Own Processing:** Each edge has a specific value assigned to it from the neural network's parameter vector as a weight.
  - **Processing Content:** Multiplies the value received from its source by this weight.
  - **Destination:** Its head node.

In the rigorous definition to follow, we will formally define the source and own processing for each unit. The destination is implicitly determined by other units specifying it as their source, so it is not explicitly defined.

Remark 5.1 (Terminology specific to this lecture)**.** The terms "source," "destination," "argument," and "return value" used here have been introduced for explanatory convenience to distinguish them from the neural network's own "input" and "output." Please note that they are not necessarily standard terms generally used in the field of neural networks.

Now, based on this idea, we will first rigorously define the "architecture," which is the computational structure independent of parameters, and then define the function that is determined when parameters are given.

Definition 5.1 (Neural Network Architecture)**.** A neural network **architecture** is rigorously defined as a tuple of the following elements.

1. **Computation Graph:** A directed acyclic multigraph $G = (\mathcal{V}, \mathcal{E}, \mathrm{Tail}, \mathrm{Head})$. Here, the node set $\mathcal{V}$ is a finite subset of positive integers $\mathcal{V} \subset \mathbb{Z}_{>0}$, and this numbering is assumed to represent a topological sort order. That is, for any $e \in \mathcal{E}$, $\mathrm{Tail}(e) < \mathrm{Head}(e)$ holds.

2. **Input/Output Dimensions:** The overall input dimension $d_{\mathrm{input}} \in \mathbb{Z}_{>0}$ and output dimension $d_{\mathrm{output}} \in \mathbb{Z}_{>0}$ of the neural network.

3. **Node Specifications:** Each node $v \in \mathcal{V}$ has the following information.
   - An **activation function** $\mathrm{g}^{(v)} : \mathbb{R}^{d_{\mathrm{arg}}^{(v)}} \to \mathbb{R}^{d_{\mathrm{ret}}^{(v)}}$. Here, $d_{\mathrm{arg}}^{(v)}, d_{\mathrm{ret}}^{(v)} \in \mathbb{Z}_{\geq 0}$ are the dimensions of the argument and return value of node v, respectively.
   - An **argument source tuple** $\mathrm{NodeSrc}^{(v)} \in ([1, d_{\mathrm{input}}]_{\mathbb{Z}} \cup \mathcal{E}_{\mathrm{in}}^{(v)})^{[1, d_{\mathrm{arg}}^{(v)}]_{\mathbb{Z}}}$. This tuple specifies for each argument of the activation function $\mathrm{g}^{(v)}$ whether it comes from an element of the overall network input or from an incoming edge.

4. **Edge Specifications:** Each edge $e \in \mathcal{E}$ has an **edge source index** $\mathrm{EdgeSrc}(e) \in [1, d_{\mathrm{ret}}^{(u)}]_{\mathbb{Z}}$, which indicates which component of the return value vector of its tail node

$u = \mathrm{Tail}(e)$ to retrieve.

5. **Parameter Specifications:**
   - The total number of parameters $d_{\mathrm{param}} \in \mathbb{Z}_{>0}$ is defined.
   - A **weight map**: A surjective map $\mathrm{Weight} : \mathcal{E} \to [1, d_{\mathrm{param}}]_{\mathbb{Z}}$ that specifies which component of the parameter vector is assigned to each edge. This map allows multiple edges to share the same parameter (**parameter sharing**).
   - **Variable parameter set**: The set of indices of parameters to be trained, $\mathcal{I}_{var} \subseteq [1, d_{\mathrm{param}}]_{\mathbb{Z}}$.
   - **Fixed parameter set**: The set of indices of parameters not to be trained, $\mathcal{I}_{fix} = [1, d_{\mathrm{param}}]_{\mathbb{Z}} \setminus \mathcal{I}_{var}$.
   - Based on the above, the set of **variable edges** $\mathcal{E}_{var} = \mathrm{Weight}^{-1}(\mathcal{I}_{var})$ and the set of **fixed edges** $\mathcal{E}_{fix} = \mathrm{Weight}^{-1}(\mathcal{I}_{fix})$ are determined.
6. **Output Specifications:**
   - An **output source map** $\mathrm{OutSrc} : [1, d_{\mathrm{output}}]_{\mathbb{Z}} \to \bigcup_{v \in \mathcal{V}} (\{v\} \times [1, d_{\mathrm{ret}}^{(v)}]_{\mathbb{Z}})$. This map specifies which component of which node's return value vector corresponds to each component of the overall network output vector.

Definition 5.2 (The Function Represented by a Neural Network)**.** Given the above architecture and specific parameter values (a **checkpoint**) $\boldsymbol{\theta} \in \mathbb{R}^{d_{\mathrm{param}}}$, the computation of the parametric function $f_{\boldsymbol{\theta}} : \mathbb{R}^{d_{\mathrm{input}}} \to \mathbb{R}^{d_{\mathrm{output}}}$ represented by the neural network is defined as follows.

We compute the output $\boldsymbol{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$ for an input $\boldsymbol{x} \in \mathbb{R}^{d_{\mathrm{input}}}$. The computation is performed in ascending order of the node numbers $v \in \mathcal{V}$. Let the return value vector of each node v be $\boldsymbol{r}^{(v)} \in \mathbb{R}^{d_{\mathrm{ret}}^{(v)}}$.

1. **Per-Node Definition (Computation):** Process $v \in \mathcal{V}$ in ascending order.
   (a) First, construct the **argument vector** $\boldsymbol{a}^{(v)} \in \mathbb{R}^{d_{\mathrm{arg}}^{(v)}}$ for the activation function $\mathrm{g}^{(v)}$ of node v.
   (b) Compute each component $a_m^{(v)}$ ($m \in [1, d_{\mathrm{arg}}^{(v)}]_{\mathbb{Z}}$) of the argument vector.
      - Let the source of the argument be $s = \mathrm{NodeSrc}_m^{(v)}$.
      - If $s = j \in [1, d_{\mathrm{input}}]_{\mathbb{Z}}$ (the source is the overall network input), then let $a_m^{(v)} \coloneqq x_j$.
      - If $s = e \in \mathcal{E}_{\mathrm{in}}^{(v)}$ (the source is an incoming edge), compute as follows.
        – Tail node: $u = \mathrm{Tail}(e)$
        – Reference index in the tail node's return value: $k = \mathrm{EdgeSrc}(e)$
        – Index of the parameter assigned to the edge: $p = \mathrm{Weight}(e)$

– Formula: $a_m^{(v)} \coloneqq \theta_p \times r_k^{(u)}$

(Since $u < v$, by the assumption that the node set is topologically sorted, $r_k^{(u)}$ has already been computed at this point.)

(c) The **return value vector** $\boldsymbol{r}^{(v)}$ of node v is computed as $\boldsymbol{r}^{(v)} \coloneqq \mathrm{g}^{(v)}(\boldsymbol{a}^{(v)})$.

2. **Overall Network Output:** After the computation for all nodes is finished, construct the output vector $\boldsymbol{y} \in \mathbb{R}^{d_{\text{output}}}$.

   - Compute each output component y_j ($j \in [1, d_{\text{output}}]_{\mathbb{Z}}$).
   - Let the source of output j be $(v, k) = \mathrm{OutSrc}(j)$.
   - Let $y_j \coloneqq r_k^{(v)}$.

Remark 5.2 (On Generality)**.** This definition is very general. For example, the node computation $\mathrm{g}(\sum_j w_j x_j + b)$ in a typical multilayer perceptron can be represented in this framework by interpreting the activation function $\mathrm{g}^{(v)}$ as encompassing the series of operations: "take the sum of all elements of the argument vector, add a bias term, and apply a non-linear function." (The bias term can be modeled as a fixed edge from a special input node whose value is always 1.) Also, complex operations like convolutional layers can be represented by defining an appropriate DAG and node functions. By generalizing to this extent, it becomes possible to handle practically used neural networks such as Convolutional Neural Networks (CNN) and Transformers. And even with this level of generalization, by imposing appropriate constraints on the differentiability of the activation functions (e.g., local Lipschitz continuity), the parameters can be determined by a unified method (specifically, a combination of gradient calculation by backpropagation and optimization by stochastic gradient methods). In other words, there is no need to understand the parameter determination method for each architecture. Some students may have learned the parameter determination method by backpropagation for limited neural networks like fully connected neural networks (multilayer perceptrons), but in this lecture, we will later see that practically used neural networks such as CNNs and Transformers can be handled similarly based on the generalized definition of neural networks provided above.

5.1 Specific Computation Examples

Let's look at a few examples to see how the formal definition of a neural network defined in this lecture can represent various functions. You might be overwhelmed by the detailed definitions, but what I want you to understand through this section is the following:

- By properly defining its architecture, a neural network can be equivalent to linear regression or logistic regression as a parametric function. In other words, a neural network is a broad concept that includes these.

- The architecture of a neural network can be irregular. Nodes can have activation functions with multi-variable outputs. The design freedom for neural network architectures is that high, and new, practically useful architectures may be created in the future.

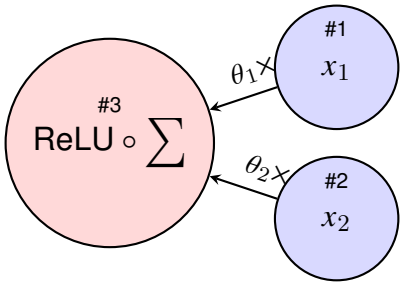

Figure 5: Legend for DAG representation of a neural network. **Nodes:** Circles represent computational units. Light blue indicates network-wide **input nodes**, light red indicates **output nodes**, and white indicates intermediate nodes. The top part inside a node is the node number (#), and the middle part shows the **activation function**. Since the activation function for input nodes is the trivial identity map (id), the input variable name (x_1, etc.) is shown instead. **Activation Function Shorthand:** If a node receives multiple inputs and a single-variable function (e.g., ReLU, σ) is written as its activation function, it is a shorthand for the operation "sum the inputs, then apply the function" (ReLU $\circ \sum$). **Edges:** Arrows indicate the flow of data. A label on an edge (e.g., $\theta_1\times$) means that the output from the tail node is multiplied by that parameter value. Note that in many cases, the activation function of a node has a single-variable output, so EdgeSrc is trivial and not explicitly written. Also, in many cases, the activation function is a symmetric function with respect to the order of its inputs (e.g., when the summation operation $\sum$ is applied first), so it does not depend on the choice of NodeSrc and is not explicitly written.

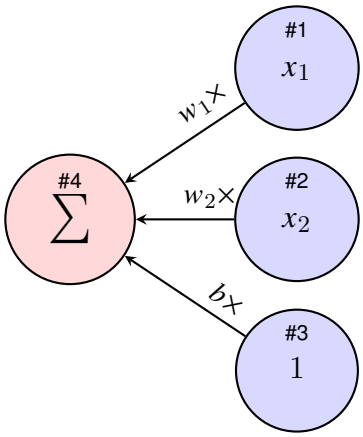

Figure 6: DAG representation of a linear regression model

Example 5.1 (Linear Regression)**.** Consider the linear regression model $f(\boldsymbol{x}) = w_1x_1 + w_2x_2 + b$ with parameters $\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, b$, which we discussed in the last lecture. Let's express this using our formal definition.

1. Architecture Definition

- **Input/Output Dimensions:** $d_{\text{input}} = 2$, $d_{\text{output}} = 1$.
- **Computation Graph:**
  - Nodes: $\mathcal{V} = \{1, 2, 3, 4\}$. (1,2 for inputs x_1, x_2, 3 for bias, 4 for output)
  - Edges: $\mathcal{E} = \{e_{(1,4)}, e_{(2,4)}, e_{(3,4)}\}$. Tail$(e_{(1,4)}) = 1$, Head$(e_{(1,4)}) = 4$, etc.
- **Parameters:** $d_{\text{param}} = 3$. $\mathcal{I}_{\text{var}} = \{1, 2, 3\}$. The weight map is Weight$(e_{(1,4)}) = 1(w_1)$, Weight$(e_{(2,4)}) = 2(w_2)$, Weight$(e_{(3,4)}) = 3(b)$.
- **Node Specifications:**
  - **Nodes 1, 2**: Represent inputs x_1, x_2. $\mathrm{g}^{(1)}() = \left[x_1\right]$, $\mathrm{g}^{(2)}() = \left[x_2\right]$.
  - **Node 3**: For bias. Always returns 1. $\mathrm{g}^{(3)}() = \left[1\right]$.
  - **Node 4**: Output node. $d_{\text{arg}}^{(4)} = 3, d_{\text{ret}}^{(4)} = 1$. Activation function $\mathrm{g}^{(4)}(\boldsymbol{a}) = \left[a_1 + a_2 + a_3\right]$. Argument source tuple NodeSrc$^{(4)} = (e_{(1,4)}, e_{(2,4)}, e_{(3,4)})$.
- **Edge Specifications:** EdgeSrc$(e_{(1,4)}) = 1$, EdgeSrc$(e_{(2,4)}) = 1$, EdgeSrc$(e_{(3,4)}) = 1$.
- **Output Specification:** OutSrc$(1) = (4, 1)$.

2. Execution of Computation

As in the previous lecture, we use the checkpoint $\boldsymbol{\theta} = \begin{bmatrix} 1.0 \\ -2.0 \\ 0.5 \end{bmatrix}$ (i.e., $w_1 = 1.0, w_2 = -2.0, b = 0.5$) and the input $\boldsymbol{x} = \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix}$.

1. **Node 1**: $\boldsymbol{r}^{(1)} = \left[x_1\right] = \left[3.0\right]$.
2. **Node 2**: $\boldsymbol{r}^{(2)} = \left[x_2\right] = \left[4.0\right]$.
3. **Node 3**: $\boldsymbol{r}^{(3)} = \left[1.0\right]$.
4. **Node 4**:
   - Compute the argument vector $\boldsymbol{a}^{(4)} = \begin{bmatrix} a_1^{(4)} \\ a_2^{(4)} \\ a_3^{(4)} \end{bmatrix}$.
   - $m = 1$: $a_1^{(4)} = \theta_1 \times r_1^{(1)} = 1.0 \times 3.0 = 3.0$.
   - $m = 2$: $a_2^{(4)} = \theta_2 \times r_1^{(2)} = -2.0 \times 4.0 = -8.0$.
   - $m = 3$: $a_3^{(4)} = \theta_3 \times r_1^{(3)} = 0.5 \times 1.0 = 0.5$.

- Therefore $\boldsymbol{a}^{(4)} = \begin{bmatrix} 3.0 \\ -8.0 \\ 0.5 \end{bmatrix}$.
- Return value vector $\boldsymbol{r}^{(4)} = \mathrm{g}^{(4)}(\boldsymbol{a}^{(4)}) = \begin{bmatrix} 3.0 - 8.0 + 0.5 \end{bmatrix} = \begin{bmatrix} -4.5 \end{bmatrix}$.

5. **Network Output**: $y_1 = r_1^{(4)} = -4.5$.

The final output is $\boldsymbol{y} = \begin{bmatrix} -4.5 \end{bmatrix}$.

Exercise 5.1. For parameters $\boldsymbol{w} = \begin{bmatrix} -0.5 \\ 3.0 \end{bmatrix}, b = -1.0$, find the output for the same input $\boldsymbol{x} = \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix}$.

Solution. The architecture is the same as in the example above. The checkpoint is $\boldsymbol{\theta} = \begin{bmatrix} -0.5 \\ 3.0 \\ -1.0 \end{bmatrix}$.

1. **Nodes 1, 2, 3**: $\boldsymbol{r}^{(1)} = \begin{bmatrix} 3.0 \end{bmatrix}, \boldsymbol{r}^{(2)} = \begin{bmatrix} 4.0 \end{bmatrix}, \boldsymbol{r}^{(3)} = \begin{bmatrix} 1.0 \end{bmatrix}$.
2. **Node 4**:
   - Compute the argument vector $\boldsymbol{a}^{(4)}$.
   - $a_1^{(4)} = \theta_1 \times r_1^{(1)} = -0.5 \times 3.0 = -1.5$.
   - $a_2^{(4)} = \theta_2 \times r_1^{(2)} = 3.0 \times 4.0 = 12.0$.
   - $a_3^{(4)} = \theta_3 \times r_1^{(3)} = -1.0 \times 1.0 = -1.0$.
   - Therefore $\boldsymbol{a}^{(4)} = \begin{bmatrix} -1.5 \\ 12.0 \\ -1.0 \end{bmatrix}$.
   - Return value vector $\boldsymbol{r}^{(4)} = \mathrm{g}^{(4)}(\boldsymbol{a}^{(4)}) = \begin{bmatrix} -1.5 + 12.0 - 1.0 \end{bmatrix} = \begin{bmatrix} 9.5 \end{bmatrix}$.
3. **Network Output**: $y_1 = r_1^{(4)} = 9.5$.

The final output is $\boldsymbol{y} = \begin{bmatrix} 9.5 \end{bmatrix}$.

Example 5.2 (Logistic Regression)**.** Next, consider the logistic regression model $f(\boldsymbol{x}) = \sigma(w_1 x_1 + w_2 x_2 + b)$, where $\sigma(z) = 1/(1 + \exp(-z))$ is the sigmoid function. The architecture is almost the same as for linear regression, but the activation function of node 4 is different.

1. Architecture Definition

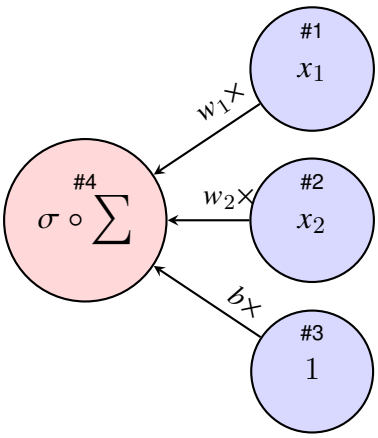

Figure 7: DAG representation of a logistic regression model

- **Node 4 Specification (Change):** The activation function is $\mathrm{g}^{(4)}(\boldsymbol{a}) = \left[\sigma(a_1 + a_2 + a_3)\right]$.
- Other specifications are the same as in the linear regression example.

2. Execution of Computation As in the previous lecture, we use the checkpoint $\boldsymbol{\theta} = \begin{bmatrix} 1.0 \\ -2.0 \\ 0.5 \end{bmatrix}$ and input $\boldsymbol{x} = \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix}$.

1. **Nodes 1, 2, 3**: $\boldsymbol{r}^{(1)} = \left[3.0\right], \boldsymbol{r}^{(2)} = \left[4.0\right], \boldsymbol{r}^{(3)} = \left[1.0\right]$.
2. **Node 4**:
   - The computation of the argument vector $\boldsymbol{a}^{(4)} = \begin{bmatrix} 3.0 \\ -8.0 \\ 0.5 \end{bmatrix}$ is exactly the same as in the linear regression example.
   - Return value vector $\boldsymbol{r}^{(4)} = \mathrm{g}^{(4)}(\boldsymbol{a}^{(4)}) = \left[\sigma(3.0 - 8.0 + 0.5)\right] = \left[\sigma(-4.5)\right]$.
   - $\sigma(-4.5) = 1/(1 + e^{4.5}) \approx 1/(1 + 90.017) \approx 0.01098$.
   - Thus $\boldsymbol{r}^{(4)} \approx \left[0.01098\right]$.
3. **Network Output**: $y_1 = r_1^{(4)} \approx 0.01098$.

The final output is $\boldsymbol{y} \approx \left[0.01098\right]$.

Exercise 5.2. For parameters $\boldsymbol{w} = \begin{bmatrix} -0.5 \\ 3.0 \end{bmatrix}, b = -1.0$, find the output of the logistic regression model for the same input $\boldsymbol{x} = \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix}$.

Solution. The checkpoint is $\boldsymbol{\theta} = \begin{bmatrix} -0.5 \\ 3.0 \\ -1.0 \end{bmatrix}$.

1. **Nodes 1, 2, 3**: $\boldsymbol{r}^{(1)} = \begin{bmatrix} 3.0 \end{bmatrix}, \boldsymbol{r}^{(2)} = \begin{bmatrix} 4.0 \end{bmatrix}, \boldsymbol{r}^{(3)} = \begin{bmatrix} 1.0 \end{bmatrix}$.
2. **Node 4**:
   - The computation of the argument vector $\boldsymbol{a}^{(4)} = \begin{bmatrix} -1.5 \\ 12.0 \\ -1.0 \end{bmatrix}$ is the same as in the linear regression exercise.
   - Return value vector $\boldsymbol{r}^{(4)} = \begin{bmatrix} \sigma(-1.5 + 12.0 - 1.0) \end{bmatrix} = \begin{bmatrix} \sigma(9.5) \end{bmatrix}$.
   - $\sigma(9.5) = 1/(1 + e^{-9.5}) \approx 1/(1 + 0.0000748) \approx 0.999925$.
   - Thus $\boldsymbol{r}^{(4)} \approx \begin{bmatrix} 0.999925 \end{bmatrix}$.
3. **Network Output**: $y_1 = r_1^{(4)} \approx 0.999925$.

The final output is $\boldsymbol{y} \approx \begin{bmatrix} 0.999925 \end{bmatrix}$.

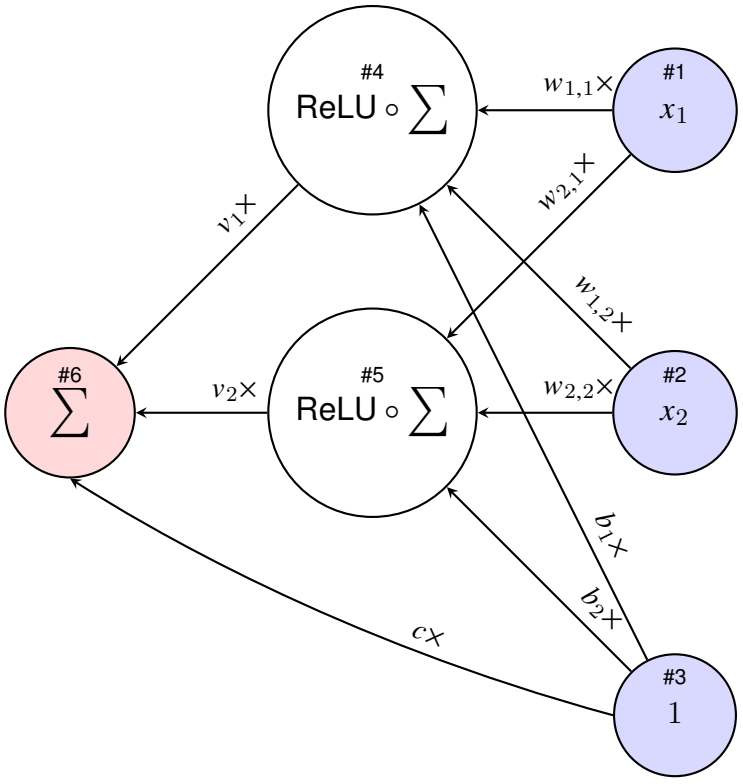

Figure 8: DAG representation of a Multilayer Perceptron (MLP)

Example 5.3 (Multilayer Perceptron (MLP))**.** Consider an MLP with a 2D input, a 2D hidden layer (with ReLU activation), and a 1D output. $h_1 = \mathsf{ReLU}(w_{1,1}x_1 + w_{1,2}x_2 + b_1)$, $h_2 = \mathsf{ReLU}(w_{2,1}x_1 + w_{2,2}x_2 + b_2)$, $y = v_1h_1 + v_2h_2 + c$.

1. Architecture Definition

- **Input/Output Dimensions:** $d_{\text{input}} = 2$, $d_{\text{output}} = 1$.
- **Computation Graph:** $\mathcal{V} = \{1, 2, 3, 4, 5, 6\}$, $\mathcal{E} = \{e_{(1,4)}, e_{(2,4)}, e_{(3,4)}, e_{(1,5)}, e_{(2,5)}, e_{(3,5)}, e_{(4,6)}, e_{(5,6)}, e_{(3,6)}\}$.
- **Parameters:** $d_{\text{param}} = 9$.
- **Node Specifications:**
  - **Nodes 1, 2, 3**: Input x_1, x_2 and bias. $\mathrm{g}^{(1)} = \left[x_1\right], \mathrm{g}^{(2)} = \left[x_2\right], \mathrm{g}^{(3)} = \left[1\right]$.
  - **Node 4**: First hidden layer node. $\mathrm{g}^{(4)}(a_1, a_2, a_3) = \left[\mathsf{ReLU}(a_1 + a_2 + a_3)\right]$. $\mathsf{NodeSrc}^{(4)} = (e_{(1,4)}, e_{(2,4)}, e_{(3,4)})$.
  - **Node 5**: Second hidden layer node. $\mathrm{g}^{(5)}(a_1, a_2, a_3) = \left[\mathsf{ReLU}(a_1 + a_2 + a_3)\right]$. $\mathsf{NodeSrc}^{(5)} = (e_{(1,5)}, e_{(2,5)}, e_{(3,5)})$.
  - **Node 6**: Output layer. $\mathrm{g}^{(6)}(a_1, a_2, a_3) = \left[a_1 + a_2 + a_3\right]$. $\mathsf{NodeSrc}^{(6)} = (e_{(4,6)}, e_{(5,6)}, e_{(3,6)})$.
- **Weight Map:** $\mathsf{Weight}(e_{(1,4)}) = \theta_1(w_{1,1}), \mathsf{Weight}(e_{(2,4)}) = \theta_2(w_{1,2}), \mathsf{Weight}(e_{(1,5)}) = \theta_3(w_{2,1}), \mathsf{Weight}(e_{(2,5)}) = \theta_4(w_{2,2}), \mathsf{Weight}(e_{(3,4)}) = \theta_5(b_1), \mathsf{Weight}(e_{(3,5)}) = \theta_6(b_2), \mathsf{Weight}(e_{(4,6)}) = \theta_7(v_1), \mathsf{Weight}(e_{(5,6)}) = \theta_8(v_2), \mathsf{Weight}(e_{(3,6)}) = \theta_9(c)$.
- **Output Specification:** $\mathsf{OutSrc}(1) = (6, 1)$.

2. Execution of Computation Use the checkpoint $\boldsymbol{\theta} = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \\ 0 \\ 1 \\ 2 \\ 3 \\ -1 \end{bmatrix}$ and input $\boldsymbol{x} = \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix}$. ($w_{1,1} = 1, w_{1,2} = 1, w_{2,1} = -1, w_{2,2} = -1, b_1 = 0, b_2 = 1, v_1 = 2, v_2 = 3, c = -1$)

1. **Nodes 1, 2, 3**: $\boldsymbol{r}^{(1)} = \left[3.0\right], \boldsymbol{r}^{(2)} = \left[4.0\right], \boldsymbol{r}^{(3)} = \left[1.0\right]$.
2. **Node 4 (Hidden 1)**:
   - $a_1^{(4)} = \theta_1 r_1^{(1)} = 1 \times 3.0 = 3.0$.

- $a_2^{(4)} = \theta_2 r_1^{(2)} = 1 \times 4.0 = 4.0$.
- $a_3^{(4)} = \theta_5 r_1^{(3)} = 0 \times 1.0 = 0$.
- $\boldsymbol{r}^{(4)} = \left[\text{ReLU}(3.0 + 4.0 + 0)\right] = \left[7.0\right]$.

3. **Node 5 (Hidden 2)**:

- $a_1^{(5)} = \theta_3 r_1^{(1)} = -1 \times 3.0 = -3.0$.
- $a_2^{(5)} = \theta_4 r_1^{(2)} = -1 \times 4.0 = -4.0$.
- $a_3^{(5)} = \theta_6 r_1^{(3)} = 1 \times 1.0 = 1.0$.
- $\boldsymbol{r}^{(5)} = \left[\text{ReLU}(-3.0 - 4.0 + 1.0)\right] = \left[\text{ReLU}(-6.0)\right] = \left[0.0\right]$.

4. **Node 6 (Output)**:

- $a_1^{(6)} = \theta_7 r_1^{(4)} = 2 \times 7.0 = 14.0$.
- $a_2^{(6)} = \theta_8 r_1^{(5)} = 3 \times 0.0 = 0.0$.
- $a_3^{(6)} = \theta_9 r_1^{(3)} = -1 \times 1.0 = -1.0$.
- $\boldsymbol{r}^{(6)} = \left[14.0 + 0.0 - 1.0\right] = \left[13.0\right]$.

5. **Network Output**: $y_1 = r_1^{(6)} = 13.0$.

Remark 5.3 (Vector Notation for MLP)**.** The function represented by the MLP in this example can be expressed more concisely using vectors and matrices. Let the input vector be $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, the hidden layer weights be the matrix $\boldsymbol{W} = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$, the hidden layer bias be the vector $\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$, and the output layer weights be the vector $\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$. The MLP computation can be written as:

$$y = c + \boldsymbol{v}^\top \text{ReLU}\left(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}\right) \tag{7}$$

Here, when a scalar function like ReLU takes a vector as an argument, it means applying ReLU to each element of the vector (**element-wise application**). That is, for $\boldsymbol{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$, $\text{ReLU}(\boldsymbol{z}) = \begin{bmatrix} \text{ReLU}(z_1) \\ \text{ReLU}(z_2) \end{bmatrix}$.

Exercise 5.3. In the MLP example, find the output for the same input $\boldsymbol{x} = \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix}$ with param-

eters $\boldsymbol{\theta} = \begin{bmatrix} 0.5 \\ -0.5 \\ 0.5 \\ -0.5 \\ 1 \\ -1 \\ -2 \\ 1 \\ 0.5 \end{bmatrix}$.

Solution. 1. **Nodes 1, 2, 3**: $\boldsymbol{r}^{(1)} = \begin{bmatrix} 3.0 \end{bmatrix}, \boldsymbol{r}^{(2)} = \begin{bmatrix} 4.0 \end{bmatrix}, \boldsymbol{r}^{(3)} = \begin{bmatrix} 1.0 \end{bmatrix}$.

2. **Node 4 (Hidden 1)**:
   - $a_1^{(4)} = 0.5 \times 3.0 = 1.5$.
   - $a_2^{(4)} = -0.5 \times 4.0 = -2.0$.
   - $a_3^{(4)} = 1 \times 1.0 = 1.0$.
   - $\boldsymbol{r}^{(4)} = \begin{bmatrix} \mathsf{ReLU}(1.5 - 2.0 + 1.0) \end{bmatrix} = \begin{bmatrix} \mathsf{ReLU}(0.5) \end{bmatrix} = \begin{bmatrix} 0.5 \end{bmatrix}$.
3. **Node 5 (Hidden 2)**:
   - $a_1^{(5)} = 0.5 \times 3.0 = 1.5$.
   - $a_2^{(5)} = -0.5 \times 4.0 = -2.0$.
   - $a_3^{(5)} = -1 \times 1.0 = -1.0$.
   - $\boldsymbol{r}^{(5)} = \begin{bmatrix} \mathsf{ReLU}(1.5 - 2.0 - 1.0) \end{bmatrix} = \begin{bmatrix} \mathsf{ReLU}(-1.5) \end{bmatrix} = \begin{bmatrix} 0.0 \end{bmatrix}$.
4. **Node 6 (Output)**:
   - $a_1^{(6)} = -2 \times 0.5 = -1.0$.
   - $a_2^{(6)} = 1 \times 0.0 = 0.0$.
   - $a_3^{(6)} = 0.5 \times 1.0 = 0.5$.
   - $\boldsymbol{r}^{(6)} = \begin{bmatrix} -1.0 + 0.0 + 0.5 \end{bmatrix} = \begin{bmatrix} -0.5 \end{bmatrix}$.
5. **Network Output**: $y_1 = r_1^{(6)} = -0.5$.

Example 5.4 (1D Convolution (Parameter Sharing))**.** As an example where the map from edges to parameters, Weight, is not injective, i.e., an example of parameter sharing, let's

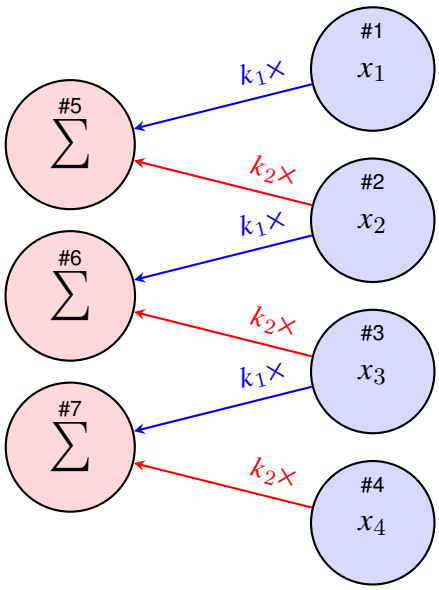

Figure 9: DAG representation of a 1D convolutional layer (kernel size 2, stride 1). Edges of the same color share the same parameter.

consider a 1D convolutional layer. For an input $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$, we compute the output using a kernel of size 2, $\boldsymbol{k} = \begin{bmatrix} k_1 \\ k_2 \end{bmatrix}$.

$$y_1 = k_1 x_1 + k_2 x_2, \tag{8}$$

$$y_2 = k_1 x_2 + k_2 x_3, \tag{9}$$

$$y_3 = k_1 x_3 + k_2 x_4. \tag{10}$$

1. Architecture Definition

- **Input/Output Dimensions:** $d_{\text{input}} = 4$, $d_{\text{output}} = 3$.
- **Computation Graph:** $\mathcal{V} = \{1, 2, 3, 4, 5, 6, 7\}$, $\mathcal{E} = \{e_{(1,5)}, e_{(2,5)}, e_{(2,6)}, e_{(3,6)}, e_{(3,7)}, e_{(4,7)}\}$.
- **Parameters:** $d_{\text{param}} = 2$. The only parameters are the kernel weights $\theta_1 = k_1, \theta_2 = k_2$.
- **Node Specifications:**
  - **Nodes 1-4**: Input. $\mathrm{g}^{(1)} = \left[x_1\right], \mathrm{g}^{(2)} = \left[x_2\right], \mathrm{g}^{(3)} = \left[x_3\right], \mathrm{g}^{(4)} = \left[x_4\right]$.
  - **Node 5**: Output 1. $\mathrm{g}^{(5)}(a_1, a_2) = \left[a_1 + a_2\right]$. $\text{NodeSrc}^{(5)} = (e_{(1,5)}, e_{(2,5)})$.
  - **Node 6**: Output 2. $\mathrm{g}^{(6)}(a_1, a_2) = \left[a_1 + a_2\right]$. $\text{NodeSrc}^{(6)} = (e_{(2,6)}, e_{(3,6)})$.
  - **Node 7**: Output 3. $\mathrm{g}^{(7)}(a_1, a_2) = \left[a_1 + a_2\right]$. $\text{NodeSrc}^{(7)} = (e_{(3,7)}, e_{(4,7)})$.

- **Weight Map (Parameter Sharing):**
  - Sharing of $\theta_1(k_1)$: $\text{Weight}(e_{(1,5)}) = 1, \text{Weight}(e_{(2,6)}) = 1, \text{Weight}(e_{(3,7)}) = 1$.
  - Sharing of $\theta_2(k_2)$: $\text{Weight}(e_{(2,5)}) = 2, \text{Weight}(e_{(3,6)}) = 2, \text{Weight}(e_{(4,7)}) = 2$.
- **Output Specification:** $\text{OutSrc}(1) = (5, 1), \text{OutSrc}(2) = (6, 1), \text{OutSrc}(3) = (7, 1)$.

2. Execution of Computation Use the checkpoint $\boldsymbol{\theta} = \begin{bmatrix} 0.5 \\ -1.0 \end{bmatrix}$ (i.e., $k_1 = 0.5, k_2 = -1.0$) and the input $\boldsymbol{x} = \begin{bmatrix} 2.0 \\ -1.0 \\ 3.0 \\ 0.0 \end{bmatrix}$.

1. **Nodes 1-4**: $\boldsymbol{r}^{(1)} = \begin{bmatrix} 2.0 \end{bmatrix}, \boldsymbol{r}^{(2)} = \begin{bmatrix} -1.0 \end{bmatrix}, \boldsymbol{r}^{(3)} = \begin{bmatrix} 3.0 \end{bmatrix}, \boldsymbol{r}^{(4)} = \begin{bmatrix} 0.0 \end{bmatrix}$.
2. **Node 5 (Output 1)**:
   - $a_1^{(5)} = \theta_1 r_1^{(1)} = 0.5 \times 2.0 = 1.0$.
   - $a_2^{(5)} = \theta_2 r_1^{(2)} = -1.0 \times (-1.0) = 1.0$.
   - $\boldsymbol{r}^{(5)} = \begin{bmatrix} 1.0 + 1.0 \end{bmatrix} = \begin{bmatrix} 2.0 \end{bmatrix}$.
3. **Node 6 (Output 2)**:
   - $a_1^{(6)} = \theta_1 r_1^{(2)} = 0.5 \times (-1.0) = -0.5$.
   - $a_2^{(6)} = \theta_2 r_1^{(3)} = -1.0 \times 3.0 = -3.0$.
   - $\boldsymbol{r}^{(6)} = \begin{bmatrix} -0.5 - 3.0 \end{bmatrix} = \begin{bmatrix} -3.5 \end{bmatrix}$.
4. **Node 7 (Output 3)**:
   - $a_1^{(7)} = \theta_1 r_1^{(3)} = 0.5 \times 3.0 = 1.5$.
   - $a_2^{(7)} = \theta_2 r_1^{(4)} = -1.0 \times 0.0 = 0.0$.
   - $\boldsymbol{r}^{(7)} = \begin{bmatrix} 1.5 + 0.0 \end{bmatrix} = \begin{bmatrix} 1.5 \end{bmatrix}$.
5. **Network Output**: $\boldsymbol{y} = \begin{bmatrix} r_1^{(5)} \\ r_1^{(6)} \\ r_1^{(7)} \end{bmatrix} = \begin{bmatrix} 2.0 \\ -3.5 \\ 1.5 \end{bmatrix}$.

Remark 5.4 (Definition of Convolution)**.** The convolution operation is often denoted using an asterisk symbol as $\boldsymbol{y} = \boldsymbol{k} * \boldsymbol{x}$. More generally, it is defined by considering **padding** and **stride**.

Let the input vector be $\boldsymbol{x} \in \mathbb{R}^{d_{\text{in}}}$, the kernel be $\boldsymbol{k} \in \mathbb{R}^{d_{\text{k}}}$, the padding number be $p \in \mathbb{Z}_{\geq 0}$, and

the stride be $s \in \mathbb{Z}_{>0}$. First, create a padded vector $\boldsymbol{x}'$ of length $d_{\text{in}} + 2p$ by adding p zeros before and after $\boldsymbol{x}$.

$$x'_i = \begin{cases} x_{i-p} & \text{if } p+1 \le i \le p + d_{\text{in}} \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

Then, the i-th component ($i = 1, 2, \dots$) of the output vector $\boldsymbol{y}$ is calculated as:

$$y_i = \sum_{j=1}^{d_{\text{k}}} k_j x'_{s(i-1)+j} \tag{12}$$

The dimension of the output vector d_{out} is

$$d_{\text{out}} = \lfloor \frac{d_{\text{in}} + 2p - d_{\text{k}}}{s} \rfloor + 1 \tag{13}$$

The example above corresponds to the case where $d_{\text{in}} = 4, d_{\text{k}} = 2, p = 0, s = 1$, resulting in $d_{\text{out}} = \lfloor \frac{4+0-2}{1} \rfloor + 1 = 3$.

Exercise 5.4. In the 1D convolution example, find the output for the same input $\boldsymbol{x} = \begin{bmatrix} 2.0 \\ -1.0 \\ 3.0 \\ 0.0 \end{bmatrix}$ with parameters $\boldsymbol{\theta} = \begin{bmatrix} 1.5 \\ 1.0 \end{bmatrix}$.

Solution. 1. **Nodes 1-4**: $\boldsymbol{r}^{(1)} = \begin{bmatrix} 2.0 \end{bmatrix}, \boldsymbol{r}^{(2)} = \begin{bmatrix} -1.0 \end{bmatrix}, \boldsymbol{r}^{(3)} = \begin{bmatrix} 3.0 \end{bmatrix}, \boldsymbol{r}^{(4)} = \begin{bmatrix} 0.0 \end{bmatrix}$.

2. **Node 5 (Output 1)**:
   - $a_1^{(5)} = 1.5 \times 2.0 = 3.0$.
   - $a_2^{(5)} = 1.0 \times (-1.0) = -1.0$.
   - $\boldsymbol{r}^{(5)} = \begin{bmatrix} 3.0 - 1.0 \end{bmatrix} = \begin{bmatrix} 2.0 \end{bmatrix}$.

3. **Node 6 (Output 2)**:
   - $a_1^{(6)} = 1.5 \times (-1.0) = -1.5$.
   - $a_2^{(6)} = 1.0 \times 3.0 = 3.0$.
   - $\boldsymbol{r}^{(6)} = \begin{bmatrix} -1.5 + 3.0 \end{bmatrix} = \begin{bmatrix} 1.5 \end{bmatrix}$.

4. **Node 7 (Output 3)**:
   - $a_1^{(7)} = 1.5 \times 3.0 = 4.5$.
   - $a_2^{(7)} = 1.0 \times 0.0 = 0.0$.

- $\boldsymbol{r}^{(7)} = \begin{bmatrix} 4.5 + 0.0 \end{bmatrix} = \begin{bmatrix} 4.5 \end{bmatrix}$.

5. **Network Output**: $\boldsymbol{y} = \begin{bmatrix} 2.0 \\ 1.5 \\ 4.5 \end{bmatrix}$.

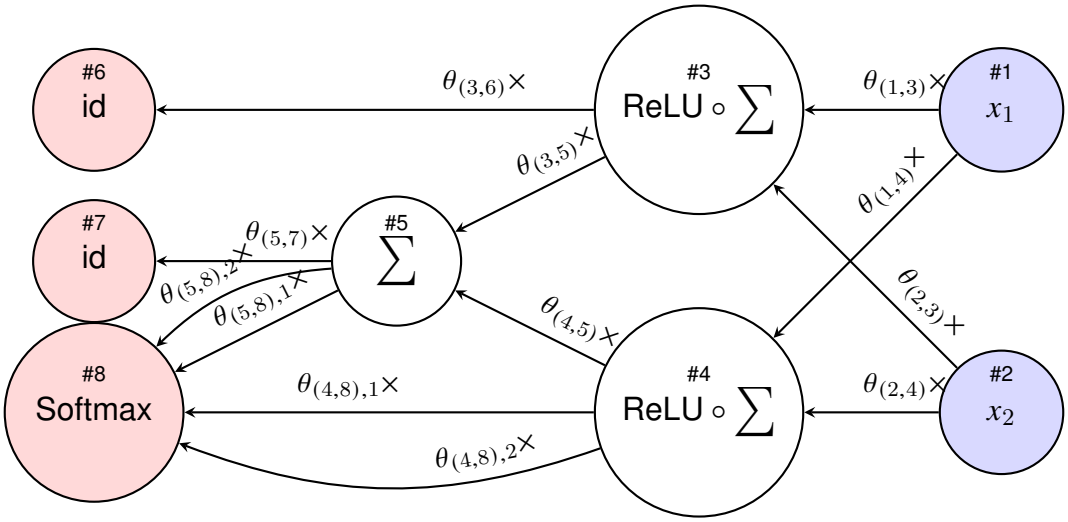

Figure 10: Example of a network with a complex structure

Example 5.5 (Network with a Complex Structure)**.** Consider the following 8-node neural network. (This is an example of the rigorous definition above). Let the node set be $\mathcal{V} = \{1, \ldots, 8\}$.

- **Input:** The network input is $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, with $x_1, x_2 \in \mathbb{R}$. $d_{\text{input}} = 2$.
- **Output:** The return values of nodes $6, 7, 8$ will be the overall output of the network. $d_{\text{output}} = 4$.
- **Activation Functions and Computations:**
  - Nodes $1, 2$: Return the input as is. $\mathrm{g}^{(1)}() = \begin{bmatrix} x_1 \end{bmatrix}, \mathrm{g}^{(2)}() = \begin{bmatrix} x_2 \end{bmatrix}$.
  - Node 3: Passes the sum of two arguments through ReLU. $\mathrm{g}^{(3)}(\boldsymbol{a}) = \mathrm{ReLU}(a_1 + a_2)$.
  - Node 4: Passes the sum of two arguments through ReLU. $\mathrm{g}^{(4)}(\boldsymbol{a}) = \mathrm{ReLU}(a_1 + a_2)$.
  - Node 5: Computes the sum of two arguments. $\mathrm{g}^{(5)}(\boldsymbol{a}) = a_1 + a_2$.
  - Node 6: Identity map. $\mathrm{g}^{(6)}(a) = a$.
  - Node 7: Identity map. $\mathrm{g}^{(7)}(a) = a$.
  - Node 8: Takes four arguments, divides them into two pairs, sums them, and applies Softmax. $\mathrm{g}^{(8)}(\boldsymbol{a}) = \mathrm{Softmax}\left(\begin{bmatrix} a_1 + a_3 \\ a_2 + a_4 \end{bmatrix}\right)$.

- **ReLU and Softmax:**
  - $\text{ReLU}(z) = \max(0, z)$.
  - $\text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$.

Specific Computation: Suppose the input is $\boldsymbol{x} = \begin{bmatrix} 1.0 \\ -2.0 \end{bmatrix}$, and the parameters are given as follows. (For simplicity, we denote the edge from node u to v as $e_{(u,v)}$, and its weight as $\theta_{(u,v)}$. If there are multiple edges between the same pair of nodes, like from node 4 to 8, or from node 5 to 8, we distinguish them as $e_{(u,v),1}, e_{(u,v),2}$, etc.) $\theta_{(1,3)} = 1, \theta_{(2,3)} = 1, \theta_{(1,4)} = 1, \theta_{(2,4)} = -1, \theta_{(3,5)} = 0.5, \theta_{(4,5)} = -0.5, \theta_{(3,6)} = 1, \theta_{(5,7)} = 2, \theta_{(4,8),1} = 1, \theta_{(4,8),2} = -1, \theta_{(5,8),1} = 1, \theta_{(5,8),2} = 1$.

1. **Node 1**: $\boldsymbol{r}^{(1)} = \Big[x_1\Big] = \Big[1.0\Big]$.
2. **Node 2**: $\boldsymbol{r}^{(2)} = \Big[x_2\Big] = \Big[-2.0\Big]$.
3. **Node 3**:
   - Arguments: $a_1^{(3)} = \theta_{(1,3)} r_1^{(1)} = 1 \times 1.0 = 1.0$, $a_2^{(3)} = \theta_{(2,3)} r_1^{(2)} = 1 \times (-2.0) = -2.0$.
   - Return value: $\boldsymbol{r}^{(3)} = \Big[\mathrm{g}^{(3)}(1.0, -2.0)\Big] = \Big[\text{ReLU}(1.0 - 2.0)\Big] = \Big[\text{ReLU}(-1.0)\Big] = \Big[0.0\Big]$.
4. **Node 4**:
   - Arguments: $a_1^{(4)} = \theta_{(1,4)} r_1^{(1)} = 1 \times 1.0 = 1.0$, $a_2^{(4)} = \theta_{(2,4)} r_1^{(2)} = (-1) \times (-2.0) = 2.0$.
   - Return value: $\boldsymbol{r}^{(4)} = \Big[\mathrm{g}^{(4)}(1.0, 2.0)\Big] = \Big[\text{ReLU}(1.0 + 2.0)\Big] = \Big[\text{ReLU}(3.0)\Big] = \Big[3.0\Big]$.
5. **Node 5**:
   - Arguments: $a_1^{(5)} = \theta_{(3,5)} r_1^{(3)} = 0.5 \times 0.0 = 0.0$, $a_2^{(5)} = \theta_{(4,5)} r_1^{(4)} = -0.5 \times 3.0 = -1.5$.
   - Return value: $\boldsymbol{r}^{(5)} = \Big[\mathrm{g}^{(5)}(0.0, -1.5)\Big] = \Big[0.0 - 1.5\Big] = \Big[-1.5\Big]$.
6. **Node 6**:
   - Argument: $a_1^{(6)} = \theta_{(3,6)} r_1^{(3)} = 1 \times 0.0 = 0.0$.
   - Return value: $\boldsymbol{r}^{(6)} = \Big[\mathrm{g}^{(6)}(0.0)\Big] = \Big[0.0\Big]$.
7. **Node 7**:
   - Argument: $a_1^{(7)} = \theta_{(5,7)} r_1^{(5)} = 2 \times (-1.5) = -3.0$.
   - Return value: $\boldsymbol{r}^{(7)} = \Big[\mathrm{g}^{(7)}(-3.0)\Big] = \Big[-3.0\Big]$.
8. **Node 8**:
   - Arguments: $a_1^{(8)} = \theta_{(4,8),1} r_1^{(4)} = 1 \times 3.0 = 3.0$. $a_2^{(8)} = \theta_{(4,8),2} r_1^{(4)} = -1 \times 3.0 = -3.0$. $a_3^{(8)} = \theta_{(5,8),1} r_1^{(5)} = 1 \times (-1.5) = -1.5$. $a_4^{(8)} = \theta_{(5,8),2} r_1^{(5)} = 1 \times (-1.5) = -1.5$.

- Inside activation function: $z_1 = a_1^{(8)} + a_3^{(8)} = 3.0 - 1.5 = 1.5$. $z_2 = a_2^{(8)} + a_4^{(8)} = -3.0 - 1.5 = -4.5$.
- Return value:

$$\begin{aligned} \boldsymbol{r}^{(8)} = \mathrm{g}^{(8)}(\boldsymbol{a}^{(8)}) = \text{Softmax}\left(\begin{bmatrix} 1.5 \\ -4.5 \end{bmatrix}\right) = \frac{1}{e^{1.5} + e^{-4.5}} \begin{bmatrix} e^{1.5} \\ e^{-4.5} \end{bmatrix} \\ \approx \frac{1}{4.482 + 0.011} \begin{bmatrix} 4.482 \\ 0.011 \end{bmatrix} \approx \begin{bmatrix} 0.9975 \\ 0.0025 \end{bmatrix}. \end{aligned} \tag{14}$$

Network Output: The output is a combination of the elements of $(\boldsymbol{r}^{(6)}, \boldsymbol{r}^{(7)}, \boldsymbol{r}^{(8)})$. Depending on the definition of OutSrc, for example, if $\boldsymbol{y} = \begin{bmatrix} r_1^{(6)} \\ r_1^{(7)} \\ r_1^{(8)} \\ r_2^{(8)} \end{bmatrix}$, then $\boldsymbol{y} \approx \begin{bmatrix} 0.0 \\ -3.0 \\ 0.9975 \\ 0.0025 \end{bmatrix}$.

Exercise 5.5. For the same architecture and input $\boldsymbol{x} = \begin{bmatrix} 1.0 \\ -2.0 \end{bmatrix}$, calculate the output with the following parameters: $\theta_{(1,3)} = 0.5, \theta_{(2,3)} = -0.5, \theta_{(1,4)} = -1, \theta_{(2,4)} = -1, \theta_{(3,5)} = 1, \theta_{(4,5)} = 1, \theta_{(3,6)} = -2, \theta_{(5,7)} = 1, \theta_{(4,8),1} = 0.5, \theta_{(4,8),2} = 0.5, \theta_{(5,8),1} = -1, \theta_{(5,8),2} = 1$.

Solution.

1. **Nodes 1, 2**: $\boldsymbol{r}^{(1)} = \begin{bmatrix} 1.0 \end{bmatrix}, \boldsymbol{r}^{(2)} = \begin{bmatrix} -2.0 \end{bmatrix}$.
2. **Node 3**: $a_1^{(3)} = 0.5 \times 1.0 = 0.5$, $a_2^{(3)} = -0.5 \times (-2.0) = 1.0$. $\boldsymbol{r}^{(3)} = \begin{bmatrix} \text{ReLU}(0.5 + 1.0) \end{bmatrix} = \begin{bmatrix} 1.5 \end{bmatrix}$.
3. **Node 4**: $a_1^{(4)} = -1 \times 1.0 = -1.0$, $a_2^{(4)} = -1 \times (-2.0) = 2.0$. $\boldsymbol{r}^{(4)} = \begin{bmatrix} \text{ReLU}(-1.0 + 2.0) \end{bmatrix} = \begin{bmatrix} 1.0 \end{bmatrix}$.
4. **Node 5**: $a_1^{(5)} = 1 \times 1.5 = 1.5$, $a_2^{(5)} = 1 \times 1.0 = 1.0$. $\boldsymbol{r}^{(5)} = \begin{bmatrix} 1.5 + 1.0 \end{bmatrix} = \begin{bmatrix} 2.5 \end{bmatrix}$.
5. **Node 6**: $a_1^{(6)} = -2 \times 1.5 = -3.0$. $\boldsymbol{r}^{(6)} = \begin{bmatrix} -3.0 \end{bmatrix}$.
6. **Node 7**: $a_1^{(7)} = 1 \times 2.5 = 2.5$. $\boldsymbol{r}^{(7)} = \begin{bmatrix} 2.5 \end{bmatrix}$.
7. **Node 8**: $a_1^{(8)} = 0.5 \times 1.0 = 0.5$. $a_2^{(8)} = 0.5 \times 1.0 = 0.5$. $a_3^{(8)} = -1 \times 2.5 = -2.5$. $a_4^{(8)} = 1 \times 2.5 = 2.5$. $z_1 = 0.5 - 2.5 = -2.0$. $z_2 = 0.5 + 2.5 = 3.0$.

$$\begin{aligned} \boldsymbol{r}^{(8)} = \text{Softmax}\left(\begin{bmatrix} -2.0 \\ 3.0 \end{bmatrix}\right) = \frac{1}{e^{-2.0} + e^{3.0}} \begin{bmatrix} e^{-2.0} \\ e^{3.0} \end{bmatrix} \\ \approx \frac{1}{0.135 + 20.086} \begin{bmatrix} 0.135 \\ 20.086 \end{bmatrix} \approx \begin{bmatrix} 0.0067 \\ 0.9933 \end{bmatrix}. \end{aligned} \tag{15}$$

Network Output: $y \approx \begin{bmatrix} -3.0 \\ 2.5 \\ 0.0067 \\ 0.9933 \end{bmatrix}$.

6 Architecture and Checkpoints

Let's summarize the discussion so far.

1. The goal of AI is to determine a function that solves a task.
2. When using a parametric function like a neural network, determining the function is replaced by determining its parameters.
3. To do this, a human first designs and fixes the DAG structure, the activation function for each node, and which edge corresponds to which parameter.
4. Then, the specific values of the parameters suitable for solving the task are found. This "finding" process is commonly called **training**.

In this context, let's define important terms used in the modern field of AI.

Definition 6.1 (Architecture and Checkpoint)**.**

- **Architecture:** The design information of a neural network, including the DAG structure, the type of activation function for each node, the correspondence between edges and parameters, and **everything other than the specific values of the parameters**. (As rigorously defined in the previous section.)
- **Checkpoint:** A set of **specific parameter values** (usually a huge list of floating-point numbers) that have been learned for a specific architecture.

By combining an architecture and a checkpoint, a specific, computable function is finally determined.

Remark 6.1 (The Ambiguity of the Word "Model")**.** The term "**model** of a neural network" can refer to different things depending on the context, so caution is required.

- In theoretical contexts or research papers, it often refers **only to the architecture**, as in "ResNet model."
- In applied contexts or software libraries, it often refers to the **pair of an architecture and a checkpoint**, as in "pre-trained model."

This distinction between architecture and checkpoint corresponds directly to the training and inference schemes we saw earlier.

- **Training:** The process of providing a human-designed **architecture** (corresponding to a family of parametric functions) and training data $\mathcal{D}$ to a training algorithm $\mathfrak{A}$ to obtain the optimal parameter values for that architecture, which is the **checkpoint** θ^*.
- **Inference:** By loading the trained **checkpoint** θ^* into the **architecture**, a specific, computable **trained model** is completed. A new input x is given to this model to obtain an output y.

This relationship is shown in Figure 11.

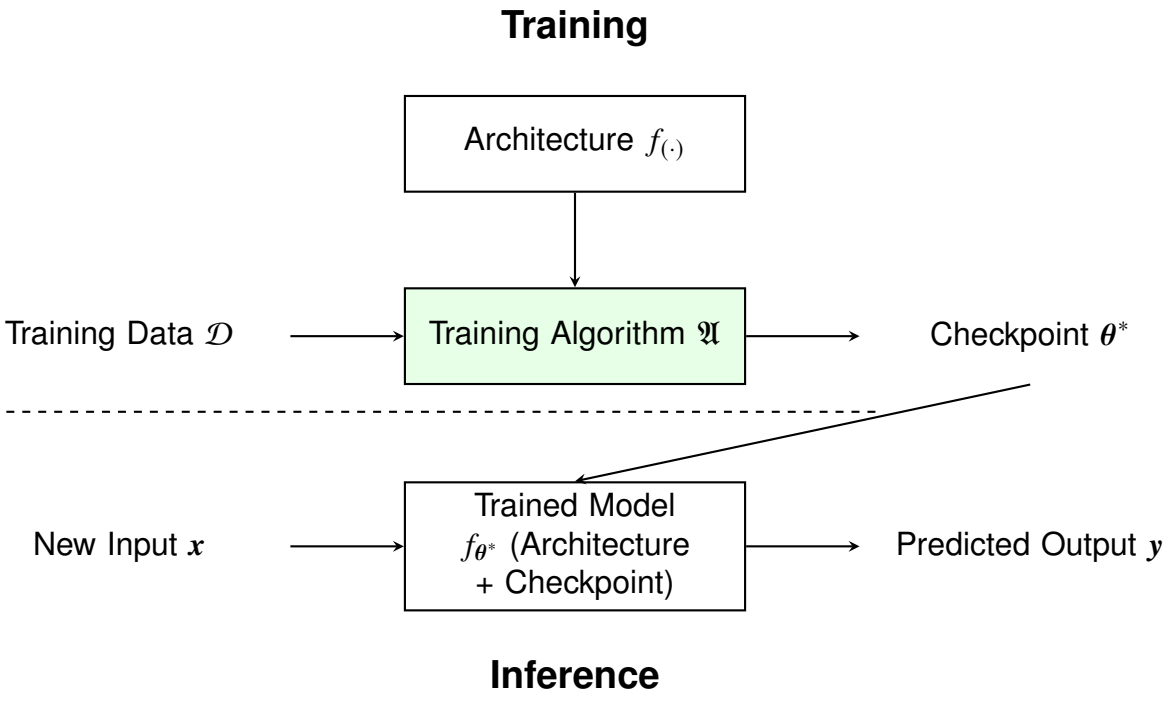

Figure 11: Training and inference scheme in neural networks.

7 Summary and Future Outlook

7.1 Today's Summary

- A neural network is a type of **parametric function** whose specific computation is determined by the values of its parameters.
- Its computational structure is rigorously defined by a **Directed Acyclic Graph (DAG)**.
- The specification of a neural network is clearly separated into two components: the **architecture**, which is the "blueprint" of the computation, and the **checkpoint**, which is the set of specific parameter values obtained through training.
- A specific, computable "trained model" is completed only when an architecture and a checkpoint are combined.

7.2 Preview of the Next Lecture

This time, we have seen that a neural network is a parametric function composed of two elements: the architecture and the checkpoint. With an understanding of this distinction, you are, for now, ready to use neural networks as black boxes.

Next time, we will discuss the inescapable issue of **licenses** when using publicly available neural network models. In fact, the distinction between architecture and checkpoint, and between ”training” and ”inference,” which we learned today, is critically important for understanding the scope of license application.

A Constraints on Activation Functions

In theory, any function can be assigned to each node of a DAG. However, in real-world applications, activation functions with specific properties are chosen. There are two main reasons for this.

1. **Possibility of Training:** To determine parameters efficiently (to train), a method called the **gradient method** is widely used. This requires the ability to calculate how a small change in a parameter affects the output (the gradient). If the activation function is discontinuous or has poor properties, the gradient cannot be defined, making training difficult.

2. **Reusability of Parameters:** The technique of **fine-tuning**, which reuses a checkpoint trained on one task as the initial value for another similar task, is extremely important. For this to be possible, a stability is required such that a small change in parameters results in only a small change in the output.

The mathematical property that satisfies these requirements is **local Lipschitz continuity**.

Definition A.1 (Local Lipschitz Continuity)**.** A function $\mathrm{g} : \mathbb{R}^n \to \mathbb{R}^m$ is locally Lipschitz continuous if, for any point $\boldsymbol{x}_0$ in its domain, there exists a neighborhood $B(\boldsymbol{x}_0, \delta)$ and a constant $K \geq 0$ such that for any two points $\boldsymbol{x}', \boldsymbol{x}''$ within that neighborhood, the following inequality holds:

$$\|\mathrm{g}(\boldsymbol{x}') - \mathrm{g}(\boldsymbol{x}'')\| \leq K\|\boldsymbol{x}' - \boldsymbol{x}''\| \tag{16}$$

Intuitively, this means that the rate of change of the function's output is locally bounded. Differentiable functions, and piecewise differentiable functions like ReLU, satisfy this property.

Local Lipschitz continuity has the following important consequences:

- It guarantees the existence of a generalized concept of the gradient (e.g., Clarke's generalized derivative), mathematically justifying the application of gradient methods.

- It ensures that the entire neural network, as a parametric function, is locally Lipschitz continuous with respect to the parameters $\boldsymbol{\theta}$. This provides stability, preventing small changes in parameters from causing explosive changes in the output, which enables fine-tuning.