ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

ARTIFICIAL NEURAL NETWORKS

Reference materials for implementing the assignments

Course: Data Structures and Algorithms (CO2003)

HO CHI MINH CITY, September 2024

Neural Networks

Lê Thành Sách

Mục lục

1 Data and Data Analysis

1.1 Data and Digital Data

Data is a form of information representation. In practice, data can exist or be stored in different physical forms such as paper and film. For example, paper and films contain written or printed documents, images, and charts. This type of data is *not yet ready* to be processed and analyzed by computers to provide useful services to users. Therefore, one of the tasks of **digital transformation**, which many countries are focusing on, is the conversion to digital form [1] for traditional storage forms, as well as applying new technologies and processes so that the data generated is already in digital form.

The data of interest in the fields of Artificial Intelligence and Data Science is digital data, such as image, audio, and video files (in all formats); text files and files in CSV and Excel formats, etc. From now on, when referring to data, it implies digital data.

1.2 Data-based Applications

Based on the types of digital data provided, the field of Artificial Intelligence (AI) aims to analyze this data to **understand** it and make corresponding decisions to *replace* or *assist* humans. Some typical applications are as follows:

1. Applications running on iPhone:
    - Setting an alarm: To set an alarm, users can give a voice command to Siri, such as "set an alarm for 4PM." Siri will recognize, understand the command, and confirm the alarm time. If Siri does not understand, it will ask for clarification.
    - Similarly, users can turn the iPhone flashlight on/off with the commands "Lumos" and "Nox," respectively.
2. With ChatGPT and related versions: users can interact with ChatGPT via Q&A, asking it to perform various tasks, such as:
    - Revising a draft to make it clearer.
    - Translating between languages.
    - Answering a wide range of other queries.

[1] Digital data: data represented by sequences of **bits** of 0 and 1; stored in files on computers

3. Facial recognition and its applications: Currently, facial authentication is highly accurate, and based on that, many applications have emerged, such as:
    - Unlocking screens, opening/closing physical or digital gates (i.e., account authentication).
    - Transferring funds in banking.
    - Identifying and tracking criminals.

The application or implementation of AI technologies in practice is diverse; however, all such applications must rely on **core** problems within the field of Artificial Intelligence.

Traditionally, Artificial Intelligence has specialized research directions aimed at **replicating** human capabilities. The human abilities of interest include:

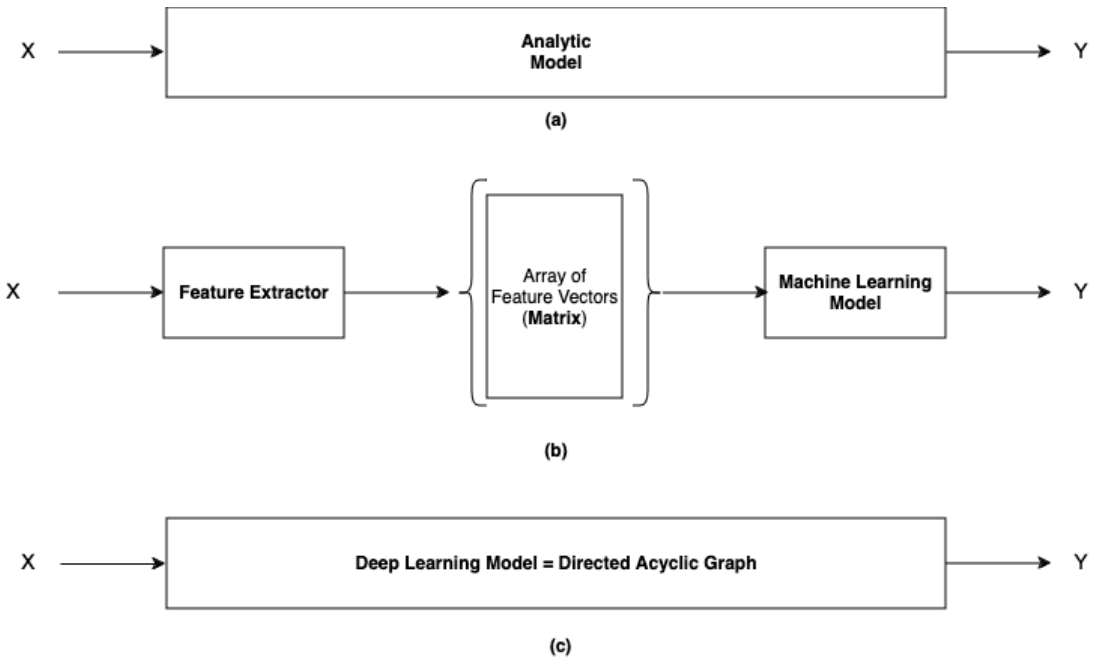
1. Vision capabilities (**Computer Vision**): the ability to understand images and videos;
2. Auditory capabilities (**Voice recognition and Synthesis**): including, voice recognition and speech synthesis.
3. Language capabilities (**Natural Language Processing**): the ability to understand and use symbols like humans.
4. Motor capabilities (**Robotics**): the ability to move like humans; typical applications include humanoid robots, self-driving cars, etc.
5. Reasoning capabilities (**Reasoning**): the ability to perform reasoning like humans.
6. Learning capabilities (**Learning**): the ability to learn from data. This is the most important capability that distinguishes humans from other animals. Typical of this direction are machine learning techniques (machine learning and deep learning).

1.3 Overview of Data Analysis

At a high level, data analysis is illustrated in Figure 1 (a). Here, X represents the input data, which could be image files, videos, audio, or text, while Y represents the output of the analysis. Let's consider a few examples to better understand X and Y.

- If X is a product review text, and the analysis model is **classification** to determine which of the following categories the review belongs to: "neutral", "negative", or "positive", then Y will be one of the three listed values.
- If X is an image of an animal, and the analysis model aims to classify which animal in the image X is (**classification**): "Dog", "Cat", or "Chicken", then Y will be one of the three listed values.

- If X is an image containing one or more animals and humans, with animals being "Dog", "Cat", and "Chicken", and the analysis model is **detection** to determine where the objects (humans and animals) are located in the frame, then Y will consist of:
  1. **For location:** A list of bounding boxes (rectangles) indicating where each object is located in the image. Each box is described by four values: c_x, c_y, w, h, corresponding to the center coordinates of the box in the image, and the width and height of the box.
  2. **For classification:** Each bounding box is associated with a value (from the set: "Human", "Dog", "Cat", and "Chicken") indicating whether the box contains a human or an animal.
- If X is a text passage in English and the model is for language translation, specifically translating the text X into Vietnamese, then Y will be the corresponding passage in Vietnamese.



Hình 1: Data analysis model

1.3.1 Data, Labels, Data Samples

In practice, to develop machine learning models, we need to collect data and assign labels to it. The data and its label form pairs $< X, Y >$ as presented earlier. Specifically, X is called the data, and Y is called the label.

For example, we need to classify a text into one of the categories: "neutral", "positive", and "negative". Let X be a text passage that has been collected. We must use humans as annotators to **label X as one of the three categories above**. The value obtained from the annotator [2] is called the label of X, denoted as Y (or t).

The process of using a computer to learn from data and labels can be understood as follows: we use humans to annotate a sufficiently large dataset and then train the computer to "copy" the human's labeling process. In other words, mathematically speaking, we aim to build a function that maps from X to Y.

Consider the following examples:

- In the text classification problem mentioned earlier, we need to collect a dataset containing many comments and assign labels ("neutral", "negative", and "positive") to each comment. Suppose we have $N = 1000$ comments, each being a passage in Vietnamese. We say that we have N data samples (**samples**). Since each data sample is assigned a label, after labeling, we obtain a list of N pairs, where each pair consists of a comment and its corresponding label.
- In the object detection problem mentioned earlier, suppose we collect $N = 10^6$ images, or 10^6 data samples. The labeling process involves drawing bounding boxes for each animal and human in the image, as well as assigning class labels to the bounding boxes. Therefore, the label of each image is a list of bounding boxes and the class labels for those boxes.
- In the language translation problem mentioned earlier, suppose we collect $N = 10^3$ text passages in English. The labeling process involves translating them into Vietnamese. Thus, we say we have a dataset of N pairs, each consisting of an English text passage and its corresponding meaning in Vietnamese.

1.3.2 Traditional Methods

In reality, the types of data for X (Figure 1 (a)) are very diverse, including text, images, videos, audio, Excel, and CSV files. Therefore, the main idea of data analysis consists of two basic steps:

1. **Feature extraction:** Represent each data sample of X by a descriptive vector for the sample. This vector is called a feature vector. Specifically, the feature vector is simply a sequence of real numbers [3].

[2] Annotator: a person who assigns labels
[3] The following libraries are commonly used to represent vectors: **Numpy**, **Pytorch**, **Tensorflow**, **xtensor**

The feature vectors of N data samples are combined into a feature matrix. Each row of this matrix is the feature vector of the corresponding data sample. The labels of the data are also represented as a vector with N elements or a matrix with N rows. The label of the k-th data sample is the k-th row in the feature matrix, and its label is the k-th element in the label vector or the k-th row of the label matrix.

2. **Using machine learning techniques:** Machine learning techniques are then applied to perform data analysis, such as classification, regression, clustering, etc. Some notable techniques include Multi-Layer Perceptron (MLP), Support Vector Machines (SVM), Decision Trees and Random Forest, Naive Bayes, etc. The course "Machine Learning" will provide students with the knowledge and skills to apply such techniques.

To obtain a vector representation for various types of data, over 12 years ago, researchers or developers would either create new algorithms or use existing ones for feature extraction and representation, as shown in Figure 1 (b). Courses such as "Image Processing and Computer Vision", "Digital Signal Processing", and "Natural Language Processing" used to teach students different techniques for extracting and representing features from image/video, audio, and text data.

Regardless of the type of input data, the feature extractor (Figure 1 (b)) outputs a common representation—a feature matrix, where each row is the feature vector of a data sample. This is why traditional machine learning libraries like **sklearn**[4] require the input for techniques to be in the form of a matrix, as shown in Figure 1 (b).

1.3.3 Modern Methods

Since 2012, beginning with the model named AlexNet[5], machine learning has shifted towards an **easier** and **more powerful** approach to data analysis. Instead of using specially designed functions to extract features, the new trend is to "**learn**" the feature vectors directly from raw data. This technique is called **deep learning**. Therefore, deep learning has the ability to learn from input data to output results, as shown in Figure 1(c). In fact, if we analyze modern deep learning models, they can be divided into two sequential parts: (a) the part that learns feature vectors for data points and (b) the part that performs core machine learning tasks (also known as assignments).

From a computational perspective, a deep learning model is simply a **Directed Acyclic Graph (DAG)**. In this graph, the nodes can represent **input** data nodes (in-degree = 0),

[4] **sklearn**: A library for traditional machine learning techniques.
[5] **AlexNet**: ImageNet Classification with Deep Convolutional Neural Networks

output nodes (out-degree = 0), or **computation nodes**, while the edges direct the flow of data through the graph.

Therefore, if we are given data at the input nodes, we can follow the directed edges of the graph to compute intermediate results and ultimately the final output of the model. For complex DAGs, we will certainly need to use the **TopologicalSort** algorithm to linearize the nodes and follow the order given by **TopologicalSort** to compute the final result.

The Multi-Layer Perceptron (MLP) is a special and simple case of a deep learning model; in it, the computation nodes are designed as a sequence with a clear order. Therefore, calculating the output of an MLP is very simple, by sequentially performing computations for each node in the sequence. Section 5 will delve deeper into topics related to MLP.

1.4 Core Tasks in Machine Learning and Deep Learning

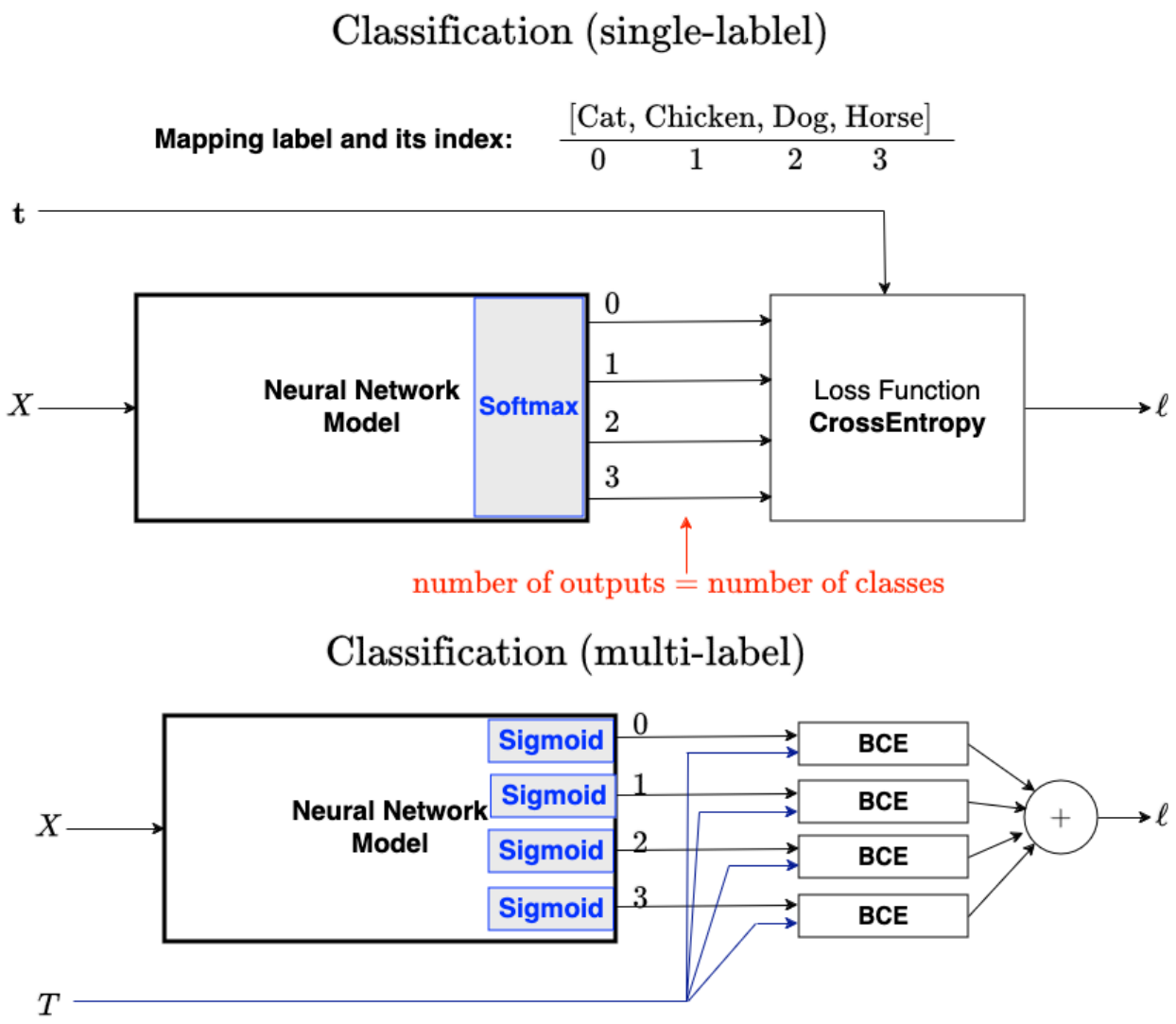
When analyzing data in Artificial Intelligence, we decompose the problem into **core tasks**[6] within the field of machine learning and deep learning.

The core tasks include: **classification**, **regression**, and **identification/discrimination**.

Consider the following examples:

- English-Vietnamese Translation Task: This task can be reduced to a word prediction problem; specifically, after translating the first k words in the output Vietnamese sentence and with the English input sentence available, the key question is: what is the $(k+1)$-th word in Vietnamese? To solve this, the current technique estimates a probability distribution over a vocabulary **D** of Vietnamese words[7]. Mathematically, this is expressed as $P(\mathbf{D} \mid \textbf{translated segment with } k \textbf{ words and the English sentence})$. This is a classification task, and the probability distribution is the output of the softmax function, as explained in Section 5.6.
- Object Detection Task in Images: This is reduced to two tasks.
  - Regression: Responsible for predicting the coordinates of bounding boxes;
  - Classification: Predicting the type of object inside the bounding box.

[6] **Core tasks**: also known as core problems.
[7] **Vocabulary**: A list of all words in a particular language.



Hình 2: Classification Model

1.4.1 Classification

The task of a classification model is to select **one or more** labels from a set of labels[8] to assign to the input data sample. If the model selects only one label from the set, we call it **single-label classification**. On the other hand, if the model can select from 0 (none selected) to more than 1 label, we call it **multi-label classification**. Single-label classification is the most common problem when discussing classification, so when classification is mentioned without further details, it usually refers to single-label classification.

To perform the classification task, the **output of the neural network** must be designed

[8]Also known as class labels or categories. For example: [Cat, Chicken, Dog, Horse]

as shown in Figure 2. More specifically, consider the following explanations.

- **For single-label classification**: Let the number of labels in the set be K. The model has the following characteristics:
  1. The model has K outputs. These K values represent the probabilities of each label in the set.
  2. To output probability values, the model has a **Softmax** layer at the output (see Section 5.6), which converts the input scores (values) into a probability distribution[9].
  3. The outputs of the model are **always** indexed as $0, 1, \cdots, (K-1)$. The mapping of these indices to actual labels in the label set is the responsibility of the model developer and trainer. The most common way is to sort the list of labels alphabetically to get an ordered list. In that case, the index of each label corresponds to the index of the model's output.
     - In Figure 2, the ordered list of labels is `[Cat, Chicken, Dog, Horse]`. Suppose the model outputs the following probability distribution: $[0.2, 0.5, 0.1, 0.2]$. This means the model believes there is a 20% chance that the label of the input data is either "Cat" or "Horse". There is a 50% chance it is "Chicken" and only a 10% chance it is "Dog".
     - In cases where we need a specific prediction, we apply the **argmax** function[10] to the sequence. **argmax**$([0.2, 0.5, 0.1, 0.2]) = 1$; the index 1 corresponds to "Chicken", so the model predicts "Chicken".
  4. Training diagram: To train a neural network for classification, we typically connect the model's output to a **Cross-Entropy** loss function (see Section 6.1). This function takes two inputs: (a) the predicted probability distribution and (b) the true label distribution. It calculates and returns a non-negative real number. The optimizer then uses this value to perform the training steps. The following sections will explain the remaining steps in more detail.
- **For multi-label classification**: Instead of applying a **Softmax** function to compute a probability distribution over all labels, multi-label classification models apply a **Sigmoid** function (see Section 5.4) to each label. The **Sigmoid** function takes a real number as input and returns a value in the range $(0, 1)$. With the label set in Figure 2, the model contains four **Sigmoid** functions internally to output the probabilities of each label's presence. Suppose the model outputs $[0.7, 0.3, 0.1, 0.8]$; we can interpret these as follows:

[9] Probability distribution over the labels (**categorical distribution**)
[10] **argmax**: Instead of returning the maximum value, this function returns the index of the maximum value.

- Note: The sum of the values in the sequence does not equal 1 because these are not the outputs of a **Softmax** function.
- The first value is 0.7, meaning there is a 70% chance that the input data **contains the label** "Cat". Also, $1 - 0.7 = 0.3$, meaning there is a 30% chance that the input data **does not contain** "Cat"[11]. The other values are interpreted similarly.
- When we need the model to make a concrete prediction, we compare the predicted probabilities to a threshold value T, for example, $T = 0.5$. If the probabilities are greater than or equal to T, we consider the model to have predicted the corresponding label. For example, comparing the predicted values to T, only the 0.7 for "Cat" and the 0.8 for "Horse" satisfy $>= T$. Thus, the model predicts "Cat" and "Horse"[12].

Training diagram: To train a multi-label classification model, we typically connect the predicted outputs to a **Binary Cross-Entropy** loss function, as shown in Figure 2. The individual losses for each label are summed (possibly with weighting) to form the total loss for the data sample.

- **For binary classification (Two-Class classification)**: Binary classification is a classification task where the label set contains only two classes, such as [Cat, Not-a-Cat], [Normal, Abnormal], [Cancer, Not-Cancer], etc. There are two approaches to solving this problem:
  1. Using **Softmax**: We design the model like single-label classification in Figure 2. In this case, $K = 2$, so the model has two outputs.
  2. Using **Sigmoid**: We design the model with only one output, which represents the probability of one label. When training, we use the **Binary Cross-Entropy** loss function (see Section 6.2).

1.4.2 Regression

A regression model is one that predicts real numbers. Some practical applications include:

- Forecasting rainfall amounts, road flooding levels, temperature, pressure, etc.
- Predicting stock prices, gold prices, etc.
- Estimating age, heart rate, etc.

Neural networks designed for regression tasks also have specific configurations. In particular, the

[11] If the data is an image, this means there is a 70% chance that the image contains a cat and a 30% chance that it does not.

[12] If the application is classifying animal sounds, this prediction would mean that the input audio contains sounds of both a cat and a horse.

number of outputs corresponds to the number of variables to be predicted. The loss functions commonly used to train regression models are the **Mean Squared Error (MSE, L2)** and **Mean Absolute Error (MAE, L1)**.

1.4.3 Identification

To be updated!

2 Evaluation Methods for Core Tasks

To be updated!

2.1 For Classification Tasks

2.1.1 Confusion Matrix

2.1.2 Accuracy

2.1.3 Precision

2.1.4 Recall

2.1.5 F1-Score

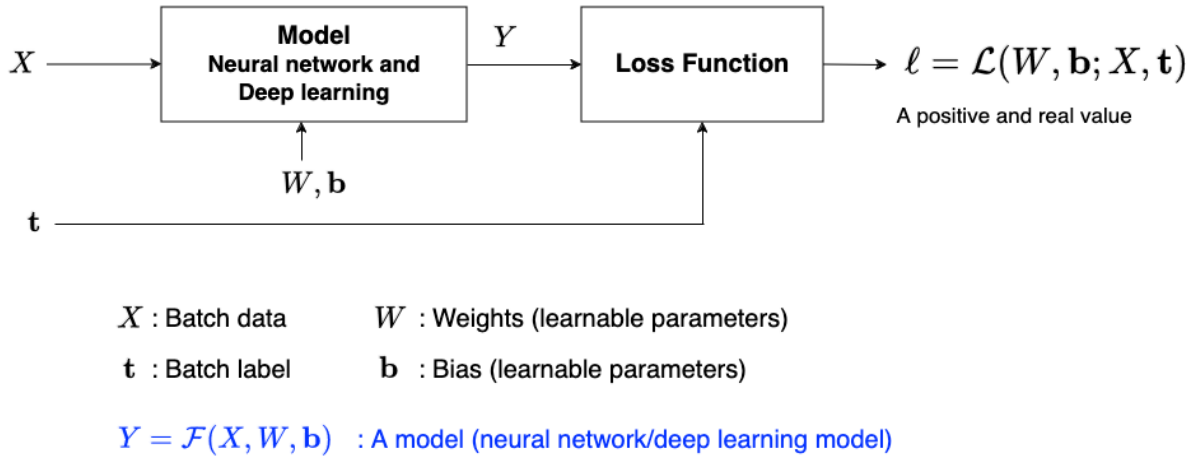
2.2 For Regression Tasks

2.2.1 Mean Squared Error (MSE)

2.2.2 Mean Absolute Error (MAE)

3 Mathematical Model for Deep Learning Networks

A deep learning model can be viewed as a mathematical function $Y = \mathcal{F}(X, W, \mathbf{b})$, as shown in Figure 3. A deep learning model has several learnable parameters denoted as W and $\mathbf{b}$; W is referred to as the **weights**, and $\mathbf{b}$ is referred to as the **bias**. Both W and $\mathbf{b}$ are learned from



Hình 3: Neural Networks and Deep Learning Training Diagram

the data; however, during the learning process, they are treated differently, which is why they are named separately.

3.1 Inference Process

With a deep learning model, after determining W and $\mathbf{b}$ from the data, meaning W and $\mathbf{b}$ are ready, when provided with X, it computes $Y = \mathcal{F}(X, W, \mathbf{b})$; this step is called **inference**.

3.2 Mathematical Basis of Training

To determine W and $\mathbf{b}$ of the model, we need data X and the labels of the data samples in X, denoted as $\mathbf{t}$.

The model predicts Y from the data X, and the learning process requires comparing this prediction with the labels $\mathbf{t}$ to find the difference, which is referred to as loss. Loss is a positive real number. $\ell = \mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$ [13]. The four parameters W, $\mathbf{b}$; X, $\mathbf{t}$ are understood as follows: To the left of the ";" are W, $\mathbf{b}$; this means that W, $\mathbf{b}$ are variables of the function $\mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$. To the right of the ";" are X, $\mathbf{t}$; meaning that X and $\mathbf{t}$ are provided when calculating the value of the loss.

Thus, the training problem for the deep learning model becomes a classic mathematical problem; that is, **finding the values of W, $\mathbf{b}$ so that the function $\mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$ achieves its minimum**. When $\mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$ reaches its minimum value over the entire space of W and $\mathbf{b}$, we say the function achieves a **global minimum**; conversely, when $\mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$ reaches

[13] $\ell = \mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$: also written as $\ell = \mathcal{L}(W, \mathbf{b}|X, \mathbf{t})$

its minimum value in a narrow region of the space of W and $\mathbf{b}$, we say the function achieves a **local minimum**. Until now, no algorithm currently in use can guarantee finding a global minimum point. The training algorithms used aim to find a local minimum. Note that we are genuinely interested in the values W^* and $\mathbf{b}^*$ at which the function $\mathcal{L}(W^*, \mathbf{b}^*; X, \mathbf{t})$ achieves its minimum rather than the minimum value of the function $\mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$.

The problem of finding minima is not unfamiliar to high school students. Given the function $f(w)$, the minimum w^* satisfies two conditions:

1. $f^{'}(w^*) = 0$
2. $f^{''}(w^*) > 0$

For simple functions that consist of a single variable, it is feasible to calculate the first and second derivatives using paper and pencil. However, with deep learning models, the number of variables (in W and $\mathbf{b}$) can reach billions. Therefore, it is impossible to calculate the partial derivatives in the traditional paper-and-pencil manner and solve the equations. Instead, we utilize **numerical methods** to compute the partial derivatives concerning the variables. This method will be presented in the following sections.

4 Training Algorithms

4.1 Overview

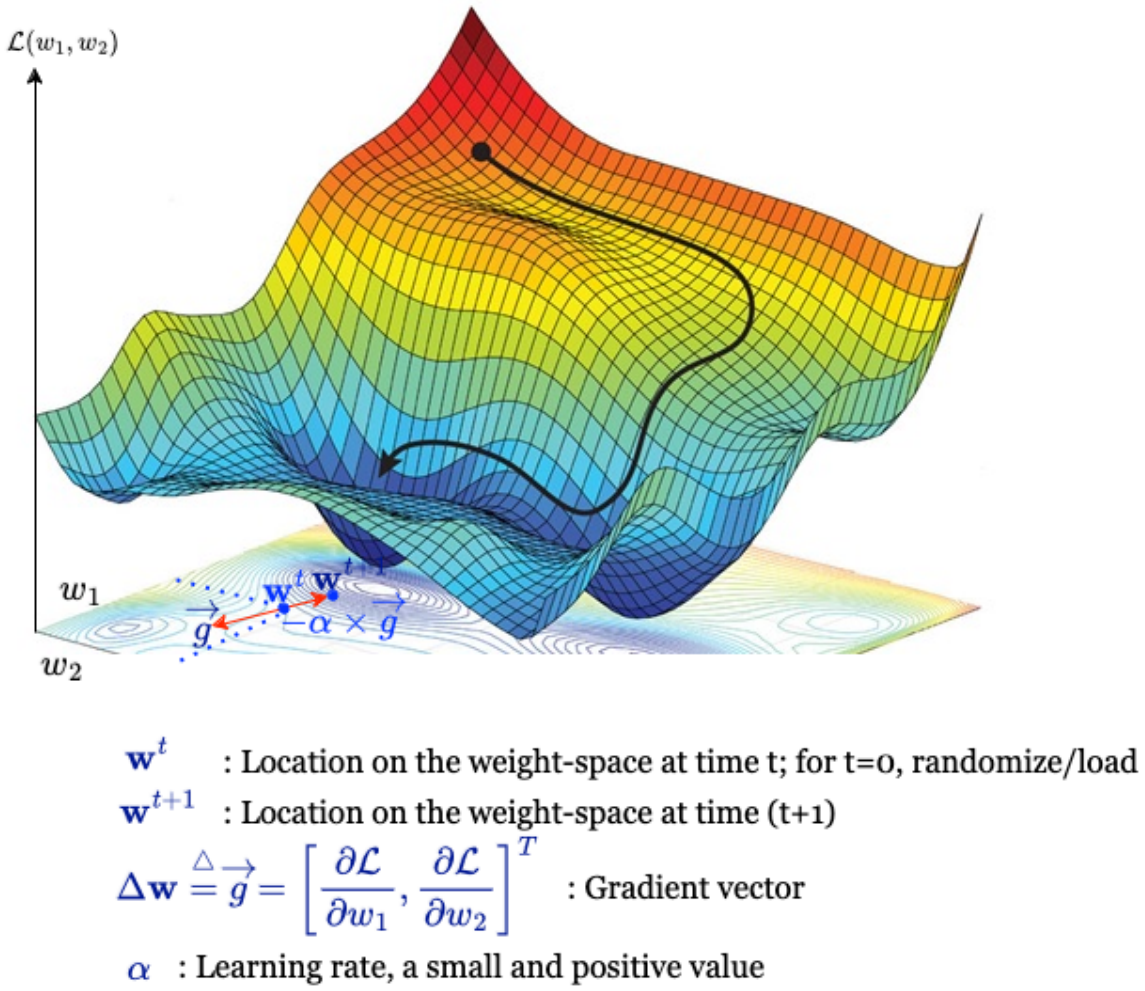
The goal of the deep learning training process is to find the parameters W^* and $\mathbf{b}^*$ such that the loss function $\mathcal{L}(W^*, \mathbf{b}^*; X, \mathbf{t})$ is sufficiently small. Please note the following points:

- $\mathcal{L}(W^*, \mathbf{b}^*; X, \mathbf{t})$ is sufficiently small rather than being precisely a local or global minimum.
- It is important that the found W^* and $\mathbf{b}^*$ enable the model to work correctly with unseen data, and its inference results are acceptable in practice.

The idea of the algorithm can be summarized as follows:

1. Initialize the **weights** and **bias** values in W and $\mathbf{b}$ with some initial values [14]. Figure 4 illustrates the loss function dependent on the two learned parameters w_1 and w_2. In the case of initialization, $\mathbf{w}^t$ corresponds to $\mathbf{w}^0$.

[14] There are two common methods: (1) generate random small numbers, often using a Gaussian distribution; (2) load from a pre-existing set of numbers



Hình 4: Illustration of the Loss Function. Source: https://culturemachine.net/

2. Iterate multiple times; during each iteration, adjust $\mathbf{w}^t$ to position $\mathbf{w}^{t+1}$ so that the loss function achieves a lower value at the point $\mathbf{w}^t$ and is minimized in a local neighborhood around point $\mathbf{w}^t$; that is:
   - $\mathcal{L}(w_1^{t+1}, w_2^{t+1}) \leq \mathcal{L}(w_1^t, w_2^t)$
   - $\mathcal{L}(w_1^{t+1}, w_2^{t+1})$: is the minimum in the local neighborhood around $\mathbf{w}^t$.

To perform step 2 above, we need to use the **gradient vector** ($\vec{\mathbf{g}}$). The **gradient vector** is a vector containing the partial derivatives of the loss function $\mathcal{L}$ with respect to each parameter that needs to be learned, which are the values in W and $\mathbf{b}$.

The gradient vector has the property that if we move in the parameter space (that is, the plane Ow_1w_2 in Figure 4) in the direction of $\vec{g}$, the loss function will **increase the fastest**; that is, its value will grow with each move. Our task is to find weights and biases such that the loss function decreases after each update. Therefore, we **must move** in the opposite direction

of $\overrightarrow{g}$, that is, in the direction of $-\overrightarrow{g}$. However, *how far should we move from* $\boldsymbol{w}^t$ *in the direction of* $-\overrightarrow{g}$*?* We have no way of knowing! Thus, we use a hyperparameter called the **learning rate** to calculate $\mathbf{w}^t$. The learning rate is a predefined positive number that is sufficiently small, for example, 10^{-4}. We calculate $\mathbf{w}^{t+1}$ from $\mathbf{w}^t$ using the formula (1).

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha \times \overrightarrow{g} \tag{1}$$

4.2 Some Concepts

4.2.1 Datasets

The collected data, including its labels, need to be represented in the form of a **tensor (multidimensional matrix)**[15] before the training process can begin. The entire dataset is divided into three subsets: (a) Training set, (b) Validation set, and (c) Testing set. The purposes of these datasets are explained as follows:

1. **Training Set**: This is the dataset used to compute the gradient vector and update the model parameters. In other words, this set is used to **select the parameters** of the model, which means the weights and biases.
2. **Validation Set**: This set is also referred to by many authors as the testing set. However, here, "validation" is used because this set is used to **test** how effective the model being trained is when working with data not in the training set. This set is used for **hyperparameter selection**.
3. **Testing Set**: This set is used to evaluate the model's effectiveness and to **report** and **compare** different techniques, architectures, etc.

The easiest way to create these three datasets from a common dataset is:

1. Randomly shuffle the data samples uniformly[16].
2. Split the samples into three sets according to the ratios $\alpha\%$ (for the training set), $\beta\%$ (for the validation set), and $(1-(\alpha+\beta))\%$ (for the testing set). Depending on the size of the dataset, we can use appropriate percentages. For reference: $\alpha = [70, 80]; \beta = [10, 15]$.

[15] Multidimensional matrix: In Python, there are libraries like **Numpy**, **Pytorch**, **Tensorflow**. In C++: xtensor

[16] also known as **shuffle**

4.2.2 Batch, Epoch, and Shuffle

We use the training sets to update W and $\mathbf{b}$. The common approach is:

1. Organize the data samples in the training set in some order (**shuffle**).
2. Divide the entire data in the set into batches[17] of size **batch-size**, which is also a hyper-parameter.
3. Use the data batches to compute the loss function, calculate the gradient vector, and update the weights and biases. Once all the batches in the dataset have been processed, the model has "**seen**" each sample once. At this point, we say that one **epoch** has been completed.

Similar to humans, very few people can fully understand and remember everything after just skimming through a book once. Training a neural network is the same; we need to perform multiple epochs by repeating the three steps above. During training, to effectively update the parameters, we need to simulate the data in the training set to arrive in any order. To do this, we need to randomly arrange the samples in the training set in Step 1 above. This is achieved by setting `shuffle=true` in the data loader.

When working with the validation and testing datasets, we do not need to shuffle the data.

4.3 Algorithm

Minibatch Stochastic Gradient Descent (SGD) is an important algorithm currently used for training neural network models and deep learning models. This algorithm can be summarized as shown in Algorithm 1.

- **Input**:
  - `trainloader` and `validloader`: are data loaders for the training and validation sets.
  - `lossLayer`: The loss function is mandatory during training; see Figure 3.
  - `optimizer`: The most basic formula for updating model parameters is provided in Formula (1). However, several effective variants of this exist that are easier to use; `optimizer` refers to whichever variant is being used.
  - `metricLayer`: During training, we need to record loss values and compute the model's performance; this is done by the `metricLayer`.

[17] batch: bó, lô

Data: (Input)
1. `trainloader`, `validloader`: DataLoaders for training and validation sets
2. `optimizer`, `lossLayer`, `metricLayer`: Optimizer, loss function, and evaluation layer
3. Hyperparameters: nepochs, learning-rate (α)
Result: (Output) $W^*, \mathbf{b}^*$
1. Initialize model parameters
2. for *epoch in range(nepochs)* **do**
  2.1. Set `train_mode=true`
  2.2. **for** *batch in trainloader* **do**
    2.2.1. X, t = batch.data, batch.label
    2.2.2. Forward pass (through `lossLayer`)
    2.2.3. Backward pass (from the output of `lossLayer`)
    2.2.4. Update model parameters (using `optimizer`, α)
    2.2.5. Record loss and model performance (using `metricLayer`)
  end
  2.3. Evaluate the model on the validation set (validloader)
  2.4. Print information
end

Algorithm 1: Minibatch Stochastic Gradient Descent (SGD)

  - `nepoches`, α (learning rate): The number of epochs for training and the magnitude of the learning rate must be set beforehand.

- **Output**: The ultimate goal of the training process is to find the values W^* and $\mathbf{b}^*$ that enable the model to perform well. These are also the model's outputs.
- **Steps in the algorithm**:
  - **1. Initialize model parameters**: Before training begins, we must initialize the parameters, meaning assigning values to them. There are two common methods:
    * (1) Initialize from a pre-trained model. In this case, training means fine-tuning the model to improve it compared to the previous model.
    * (2) Initialize from scratch. There are various initialization methods; a simple way is to start with small random numbers drawn from a standard distribution for weights, and biases can be set to 0.
    * **Note**: In specific implementations, the initialization step may occur outside the training algorithm; just ensure that the weights and biases are initialized properly.
  - **2. Loop over each epoch**: An epoch is one complete pass through all the data points in the training set. Thus, an epoch consists of many data batches. The tasks within an epoch are summarized as follows:
    * **Step 2.1**: This step sets all computation layers to training mode. The reason for this is that the forward pass operates differently between train mode and

eval mode. In the entire algorithm, only Step 2.3 needs to switch to eval mode. Therefore, if the implementation of Step 2.3 starts with switching to eval mode and ends with restoring the original mode, Step 2.1 is not strictly necessary.

* **Step 2.2**: Process the data batches within the epoch. In this, Steps **2.2.2**, **2.2.3**, and **2.2.4** are crucial and are explained separately in the following sections. Step **2.2.5** records the loss and performance for the batch and accumulates these across batches for each epoch.

4.4 Forward Pass

As introduced in previous sections, if we view deep learning models and loss functions as mathematical functions, $\mathcal{F}(X, W, \mathbf{b})$ and $\mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$ (see Figure 3), the forward pass involves calculating the loss value (ℓ) from a batch of data and labels $(X, \mathbf{t})$ along with the current values of weights and bias $(W, \mathbf{b})$.

Alternatively, if we consider the deep learning model as a computational graph (DAG), the forward pass proceeds with two steps:

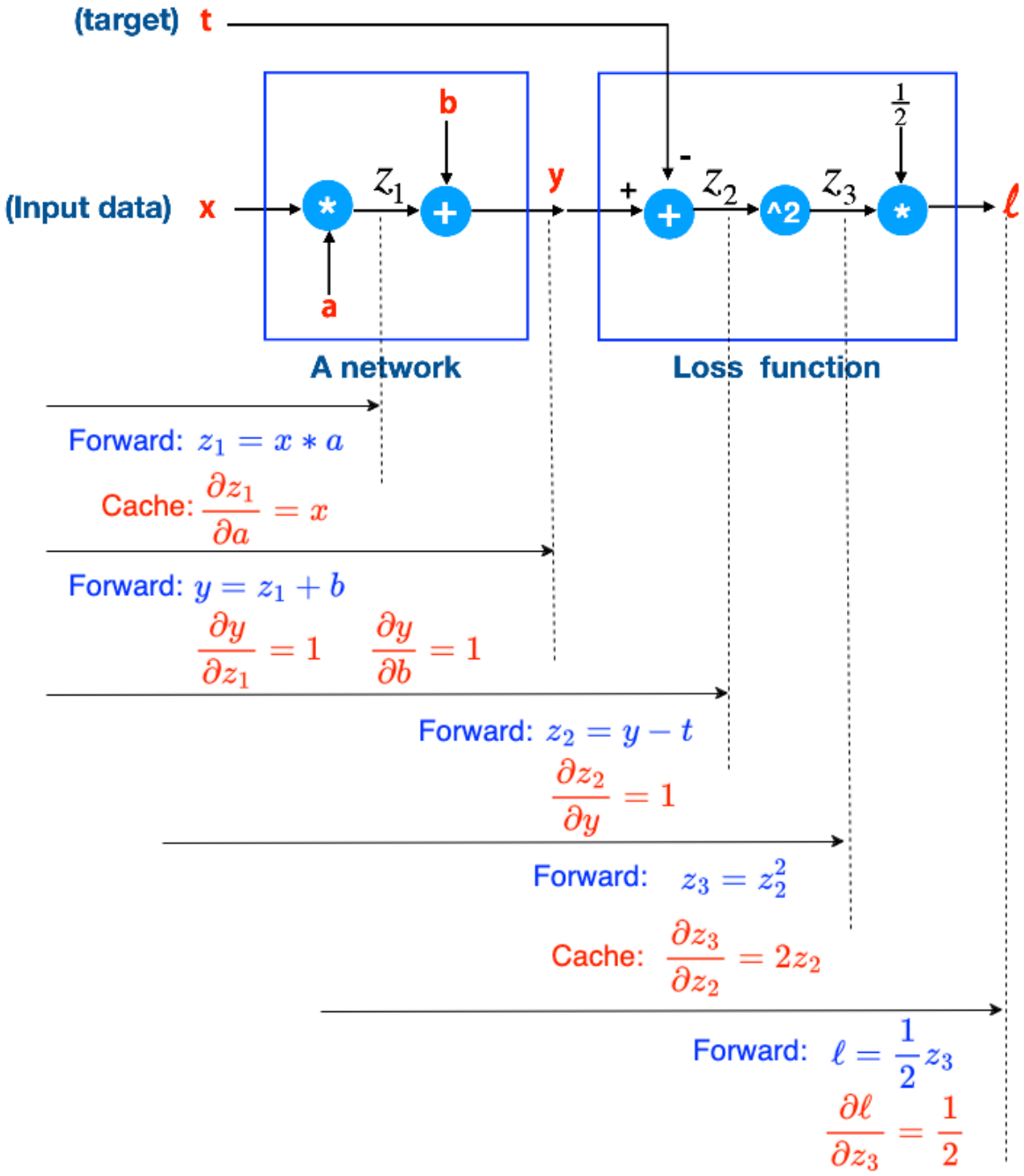
1. Linearize the sequence of computation layers in the graph using the **TopologicalSort** algorithm. Since the deep learning model is a DAG, such a sequence always exists.
2. Compute the output for each layer in the sequence obtained from the previous step.

The forward pass is implemented simply as described above. However, a crucial point to note is the need to clearly differentiate between the two modes: `train_mode` and `eval_mode`.

1. `train_mode`: This mode indicates that the model is currently being trained. The forward pass in this mode not only performs the calculations mentioned above but also needs to store data to assist in computing the derivatives during the backward pass (see the backward pass section).
2. `eval_mode`: This mode indicates that the model is not in the training phase and is operating in **inference** mode. The forward pass in this mode only requires the computations as discussed above. Typically, when the model is not in training mode, inference is used instead of the forward pass.

4.4.1 Example 1: Operations on Real Numbers

Figure 5 illustrates the structure of a simple model that consists solely of basic operations on real numbers. This model can predict the value y that has a linear relationship with the input



Hình 5: Simple model, computations only on real numbers

x, i.e., $y = a \times x + b$.

4.4.1.a Inference Mode (eval-mode)

In this mode, the model's parameters are predefined; specifically, here they are a and b. Thus, the forward pass in this case follows the arrows to sequentially calculate the values: z_1 and y;

then return y, as shown in Figure 5. In this mode, the forward pass does not need to cache any values.

4.4.1.b Training Mode (training-mode)

In this mode, the forward pass not only follows the arrows to calculate the output values from the computations in the model but also needs to cache some values to support the computation of gradients during the backward pass.

Considering a data-label pair $< x, t >$, based on this value pair, the forward pass performs the following computations sequentially from left to right in Figure 5; specifically:

1. Sequentially calculate the quantities z_1, y, z_2, z_3, and ℓ.
2. As it traverses each computation node, the forward pass also calculates the local derivatives[18]. If these local derivatives are not constants, they must also be stored for use during the subsequent backward pass.

4.5 Backward Pass

The goal of the backward pass is to compute the gradient vector mentioned in previous sections to assist with the update operations based on Formula (1).

4.5.1 Simple Backward Pass

Consider the computational graph shown in Figure 5. The goal of the backward pass in this example is to compute $\overrightarrow{g} = [\frac{\partial \ell}{\partial a}, \frac{\partial \ell}{\partial b}]^T$. Using the **chain rule**, we can calculate $\frac{\partial \ell}{\partial a}$ and $\frac{\partial \ell}{\partial b}$ according to Formulas (2) and (3). However, how do we know the sequence of variables to differentiate in these formulas? In fact, we do not decompose them mathematically; instead, we **backtrack along the arrows** in Figure 5 to multiply the local derivatives sequentially. These local derivatives have been computed and cached during the forward pass. In this way, we can compute $\frac{\partial \ell}{\partial a}$ and $\frac{\partial \ell}{\partial b}$ according to Formulas (4) and (5). Note that the computer uses the values of variables such as z_2 and x in the formulas (4) and (5) instead of using the variable names.

[18] **Local derivatives**: are the partial derivatives of the output with respect to the input of the computation node

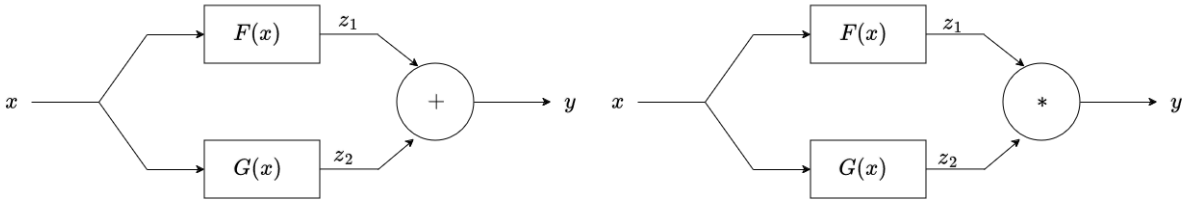
$$\frac{\partial \ell}{\partial a} = \frac{\partial \ell}{\partial z_3} \times \frac{\partial z_3}{\partial z_2} \times \frac{\partial z_2}{\partial y} \times \frac{\partial y}{\partial z_1} \times \frac{\partial z_1}{\partial a} \tag{2}$$

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial z_3} \times \frac{\partial z_3}{\partial z_2} \times \frac{\partial z_2}{\partial y} \times \frac{\partial y}{\partial b} \tag{3}$$

$$\frac{\partial \ell}{\partial a} = \frac{1}{2} \times 2 \times z_2 \times 1 \times 1 \times x \tag{4}$$

$$\frac{\partial \ell}{\partial b} = \frac{1}{2} \times 2 \times z_2 \times 1 \times 1 \tag{5}$$

4.5.2 Backward Pass through Split-Merge Nodes



$y = F(x) + G(x) => y^{'} = F^{'}(x) + G^{'}(x)$

(a) Aggregation: sum

$y = F(x) * G(x) => y^{'} = F^{'}(x)G(x) + F(x)G^{'}(x)$

(b) Aggregation: multiply

Hình 6: Split-Merge and Backward Pass

Complex neural network architectures often involve distributing data through branches and performing calculations by separate modules, then aggregating them together, as illustrated by the two diagrams in Figure 6. In the case of Figure 6 (a), x is distributed to two functions $F(x)$ and $G(x)$, which are then aggregated by addition. Figure 6 (b) is similar to (a) in terms of distribution but differs in the aggregation operation. In both cases, we see that during the backward pass, **when distributing a variable to different functions during the forward pass, we sum the derivatives together in the backward pass**.

In another case, when the forward pass merges multiple branches at a summation node; for example, the nodes $+$ and $*$ in Figure 6, **in the backward pass we need to compute the derivatives for each input branch based on the aggregation mechanism**. For instance, in Figure 6 (a), we have $y = z_1 + z_2$. Therefore, $\partial y/\partial z_1 = \partial y/\partial z_2 = 1$; whereas in Figure 6 (b), $y = z_1 * z_2$. Therefore, $\partial y/\partial z_1 = z_2 = G(x)$ and $\partial y/\partial z_2 = z_1 = F(x)$.

4.6 Parameter Update Strategies

The parameters of the model, or learnable parameters, include weights (W) and bias ($\mathbf{b}$). The update formula for weights and bias is the same. To simplify the presentation, we introduce the variable $\mathbf{x}$; $\mathbf{x}$ can be W, $\mathbf{b}$, or any other learnable parameter.

The parameter update strategies in this section require at least the following quantities as input:

1. $\mathbf{x}$: the parameter to be learned;
2. $\Delta\mathbf{x}$: the derivative of the loss function with respect to $\mathbf{x}$.
3. α: the learning rate, a sufficiently small positive real number, e.g., 10^{-4}.
4. Other hyperparameters, introduced specifically for each strategy.

4.6.1 SGD

SGD is the most basic strategy, updating the parameter $\mathbf{x}$ according to Formula (6).

$$\Delta\mathbf{x}^{(t+1)} = \Delta\mathbf{x}^{t} - \alpha \times \Delta\mathbf{x} \tag{6}$$

4.6.2 Momentum

The Momentum strategy introduces an additional variable $\mathbf{v}$ (velocity), which accumulates the derivatives over time[19]. The variable $\mathbf{v}$ is initialized to 0 and is updated according to Formula (7). Here, ρ is a hyperparameter used to control the accumulation of derivatives in $\mathbf{v}$. Typical values for ρ are 0.9 or 0.99. If $\rho = 0$, the Momentum strategy behaves like the original SGD.

Based on the updated $\mathbf{v}$, Momentum recalculates $\Delta\mathbf{x}$ using Formula (8).

$$\mathbf{v}^{(t+1)} = \rho\mathbf{v}^{t} + \Delta\mathbf{x} \tag{7}$$

$$\Delta\mathbf{x}^{(t+1)} = \Delta\mathbf{x}^{t} - \alpha \times \mathbf{v}^{(t+1)} \tag{8}$$

[19]That is, across updates.

4.6.3 Adagrad

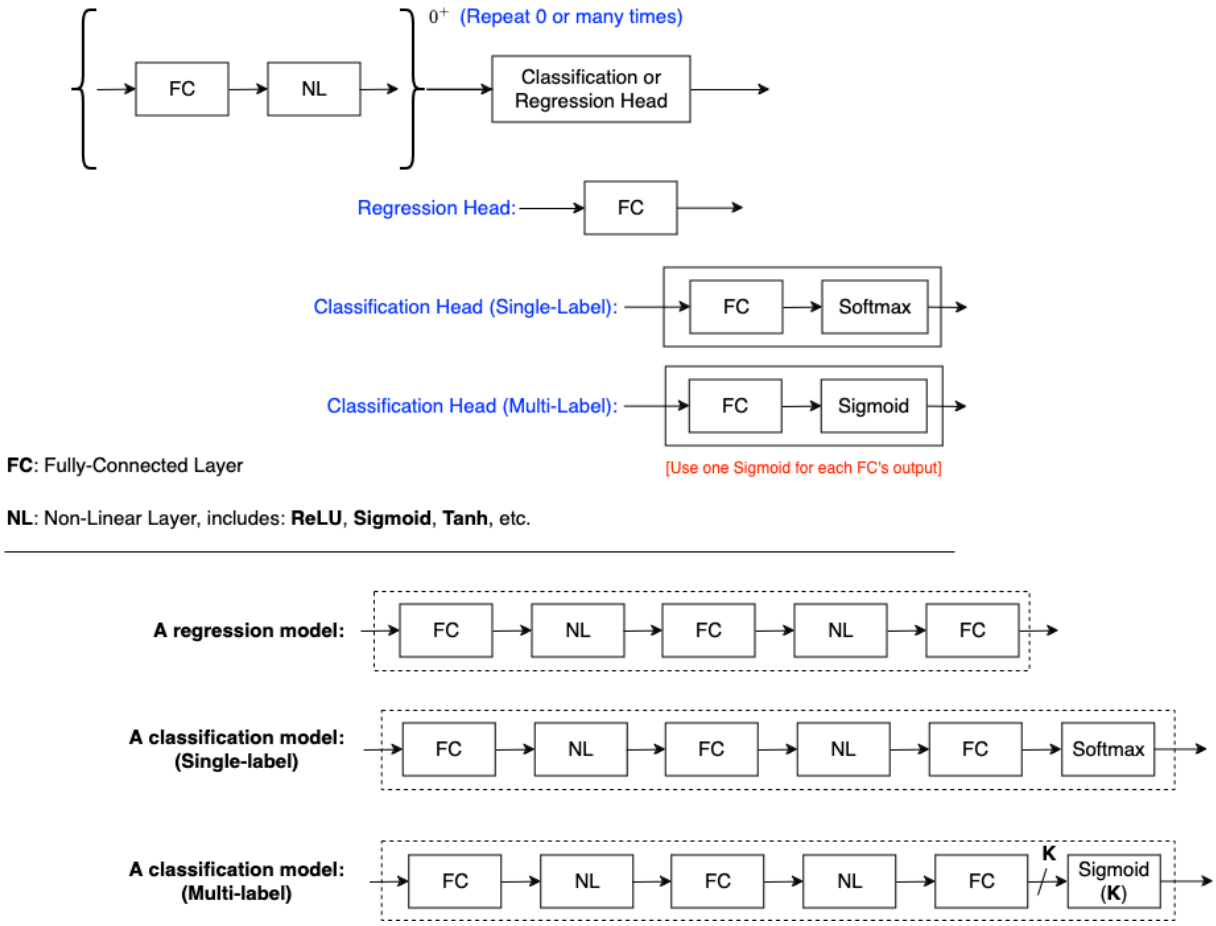
To be updated!

4.6.4 Adam

To be updated!

5 Multi-Layer Feedforward Neural Networks

5.1 Overview of the Architecture



Hình 7: Multi-Layer Feedforward Neural Network (MLP)

A Multi-Layer Feedforward Neural Network is a computational structure consisting of a sequence of nodes (also called layers), as illustrated by the three models at the end of Figure

7. In other words, we can use a list structure to store the sequence of computational nodes in an MLP.

5.1.1 Overview of Computational Layers

The computational nodes in an MLP are divided into two groups:

1. **Linear**: As of now, there are only two layers in this group; these are the Fully-Connected layer (FC) and the Convolutional layer (Conv). However, in the context of MLP, only the FC layer is used. Conv layers are widely used in deep learning models.
2. **Non-linear**: In Figure 7, this group is abbreviated as NL. Currently, this group includes many layers, the most common in MLP are layers like ReLU, Sigmoid, and Tanh. Softmax is also a non-linear layer; however, it is only used as the final layer of a single-label classification model.

5.1.2 Architecture

The sequence of computational nodes can be divided into two parts, as illustrated in the upper part of Figure 7.

1. **Classification or Regression Head**: This part directly addresses the classification or regression task. They are referred to as the "classification head" or "regression head". The architecture of these heads is also simple.
    - **Regression Head**: Regression is the task of predicting a real-valued output for a specific quantity of interest. Therefore, this head consists of only one linear layer (FC).
    - **Single-label Classification Head**: Single-label classification is a very common task where the model intends to predict a single label for each input data point. Thus, the computational structure of this head includes an FC layer to transform features into scores and a Softmax layer to convert scores into a categorical distribution. To make the decision on which label to output, the inference step needs to perform **Argmax** on this distribution to select the label with the highest confidence (probability). In the literature, this head is also referred to as **Softmax Regression**.
    - **Multi-label Classification Head**: Multi-label classification, also known as "tagging", is the task where the number of labels that the model can predict for each data point can range from 0 to the total number of labels for the task. Thus, the

computational structure of this head includes an FC layer to transform features into scores and applies a Sigmoid function to each output of the FC layer. The output of the Sigmoid is a value in the range $(0, 1)$, which signifies the probability of the corresponding label. The inference step involves comparing the Sigmoid outputs against a threshold value (a hyperparameter, predetermined, e.g., 0.5) to determine whether any labels can be assigned to the input data.

2. **Feature Transformation Part**: Feature transformation is a crucial task that enhances the model's capability. Specifically,
   - The regression head is merely a linear transformation. If the model consists solely of the layers in this head, one cannot expect it to uncover complex (non-linear) relationships between outputs and inputs.
   - Similarly, the classification head has a linear decision boundary. Thus, if only this head is used directly without a feature transformation block, one cannot expect the model to have a non-linear decision boundary.
   - To address these two issues and enhance the model's capacity, a feature transformation block is necessary. In an MLP, as illustrated in Figure 7, the feature transformation block is formed by stacking one or more sub-blocks, each being a combination of an FC layer and a layer from the Non-linear group. The deep neural network and, specifically, the architecture of popular versions of ChatGPT [20] today, features a renowned transformation block known as the Transformer[21], serving the role of the feature transformation blocks discussed here.

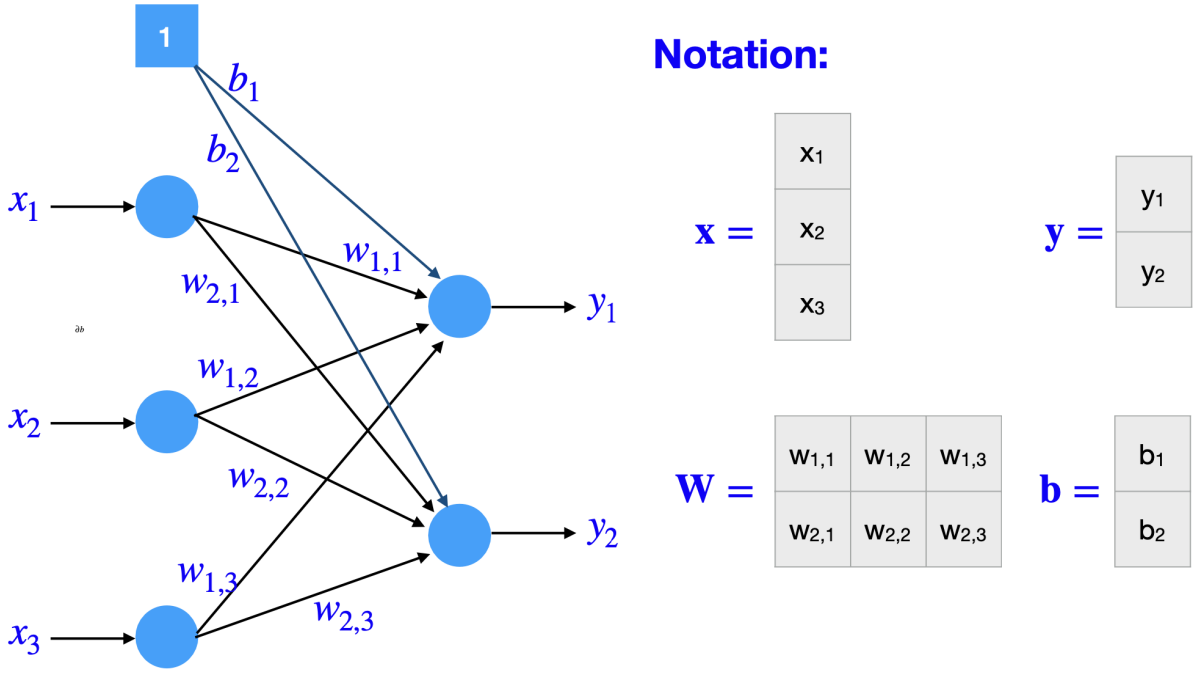
5.2 Fully Connected Layer

The fully connected layer is a linear transformation, as illustrated in Figure 8; where the output values y_1 and y_2 are calculated using Formulas (9) and (10), respectively. If we let $\mathbf{x} = [x_1, x_2, x_3]^T$, $\mathbf{w}_1 = [w_{1,1}, w_{1,2}, w_{1,3}]^T$, and $\mathbf{w}_2 = [w_{2,1}, w_{2,2}, w_{2,3}]^T$, we can derive a more concise form, which is $y_1 = \mathbf{w}_1^T\mathbf{x} + b_1$ and $y_2 = \mathbf{w}_2^T\mathbf{x} + b_2$. For the FC layer, $\mathbf{w}_1$, $\mathbf{w}_2$, b_1, and b_2 are collectively referred to as the parameters of the layer or learnable parameters; where $\mathbf{w}_1$ and $\mathbf{w}_2$ are the "weights", while b_1 and b_2 are referred to as the "biases".

A few notes:

1. The number of "weights" is equal to the number of edges connecting the input to the output. Since the FC layer is fully connected, the number of edges is the product $N_{in} \times N_{out}$;

[20] **ChatGPT**
[21] **Transformer**: Attention Is All You Need



Hình 8: Illustration of a Fully Connected (FC) layer with three inputs and two outputs.

where N_{in} and N_{out} correspond to the number of inputs and outputs of the FC layer, respectively. In the case of the FC layer shown in Figure 8, the number of weights is $3 \times 2 = 6$.

2. The number of "biases" equals the number of outputs of the layer, which is N_{out}. In the case of the FC layer in Figure 8, the number of biases is 2. In some cases, users may choose not to use biases, meaning there are no b_1 and b_2 as shown in Figure 8.
3. The total number of learnable parameters in the model is $(N_{in} \times N_{out})$ if biases are not used, and $(N_{in} \times N_{out} + N_{out})$ when biases are included.

$$y_1 = w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3}x_3 + b_1 \tag{9}$$

$$y_2 = w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3 + b_2 \tag{10}$$

5.2.1 Weight Matrix and Bias Vector

The weights and biases of the FC layer are organized in the form of a matrix and a vector, respectively. For example, the matrix W and vector $\mathbf{b}$ are illustrated in Figure 8; where W is a matrix of size 2×3 (which is $N_{out} \times N_{in}$ in general[22]); while $\mathbf{b}$ is a vector (column) containing the two biases b_1 and b_2 (which is N_{out} biases in general when biases are used).

[22]The organization of weights as $N_{out} \times N_{in}$ or $N_{in} \times N_{out}$ depends on the library used.

5.2.2 Forward Propagation

5.2.2.a For a Single Data Sample x

We also represent the input data for a single data sample as the vector $\mathbf{x}$ as shown in Figure 8. In this case, the output $\mathbf{y}$ is obtained using Formula (11); the vector $\mathbf{y}$ contains the two values y_1 and y_2 calculated by Formulas (9) and (10). Formula (11) requires the matrix multiplication of W with the vector $\mathbf{x}$ and the addition of the intermediate result (also a vector) with the vector $\mathbf{b}$.

$$\mathbf{y} = W\mathbf{x} + \mathbf{b} \tag{11}$$

5.2.2.b For a Batch of Data X

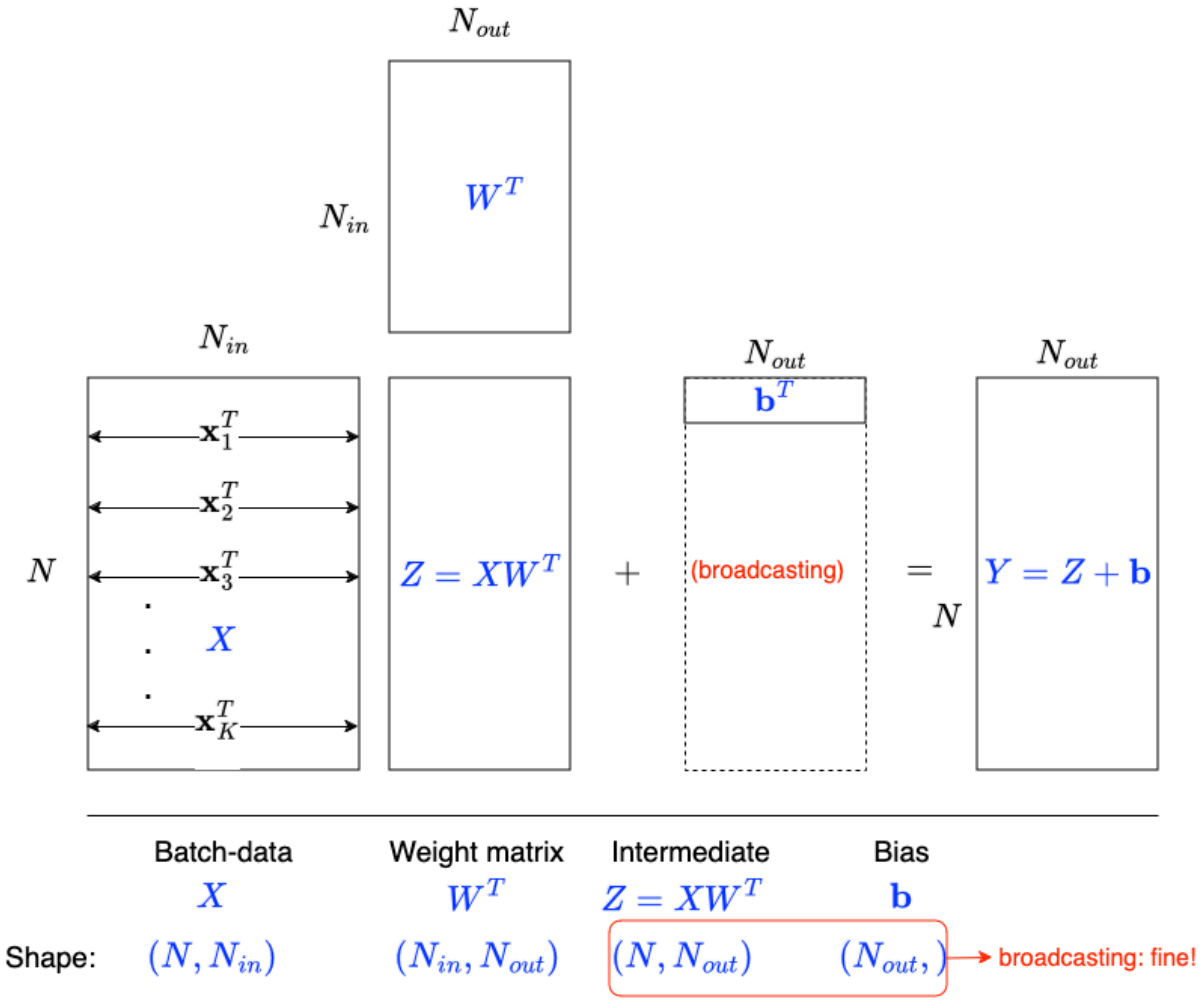
In neural networks and deep learning models, we often process a batch of data through the network rather than just a single data sample. As illustrated in Figure 9, we can transform the entire batch of data X to the output Y using Formula (12). This formula requires the multiplication of two matrices X and W^T, followed by the addition of the vector $\mathbf{b}$. Letting $Z = XW^T$ be the intermediate result, Z is a matrix of shape (N, N_{out}). Therefore, we need to add the matrix Z to the vector $\mathbf{b}$ (which has the shape $(N_{out},)$). This addition requires a concept known as "broadcasting". Fortunately, libraries for multi-dimensional matrices support this, including the **xtensor** library used in the large assignment.

$$Y = XW^T + \mathbf{b} \tag{12}$$

5.2.3 Backpropagation

5.2.3.a Notation

- In this document, the loss function is denoted as $\ell = \mathcal{L}(W, \mathbf{b}; X, \mathbf{t})$.
- To avoid cumbersome notation for partial derivatives, we use the symbols δ and Δ defined as follows:
  - If z is a scalar (real number), the partial derivative of the loss function with respect to z is denoted as δz; that is, $\delta z \stackrel{\text{def}}{=} \partial\ell/\partial z$.
  - The partial derivative of the loss function with respect to the variables in the vector $\mathbf{z}$ or the variables in the matrix and multi-dimensional array Z is denoted by $\Delta\mathbf{z}$



Hình 9: FC with a Batch of Data

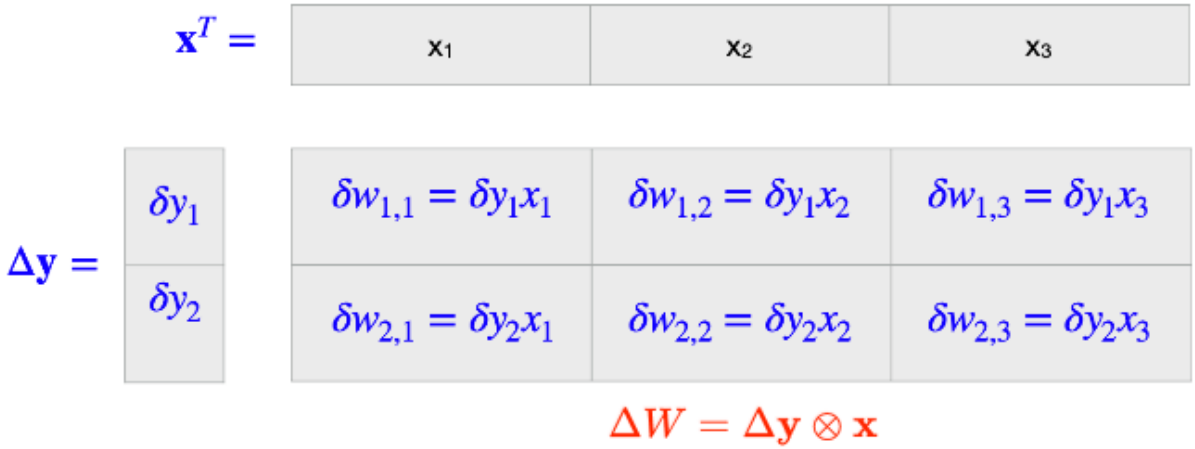
and ΔZ, respectively.

5.2.3.b Computing ΔW for a Single Data Sample

For a single data sample $\mathbf{x}$, the output $\mathbf{y}$ is calculated using Formula (11). Assuming that the backpropagation process has reached $\mathbf{y}$; that is, we have obtained $\Delta\mathbf{y}$. The value of ΔW can then be calculated using Formula (13), where $\otimes$ denotes the outer product between two vectors. Figure 10 illustrates this calculation for the case of the FC layer in Figure 8.

Some notes:

1. Formula (13) also shows that to compute ΔW, the data sample $\mathbf{x}$ needs to be cached during the forward propagation process.
2. The shape of ΔW is the same as the shape of W, which is $(N_{out} \times N_{in})$. Similarly, for any



Hình 10: How to compute ΔW for a single data sample.

1D or multi-dimensional array $\mathbf{Z}$, its shape will match the shape of $\Delta\mathbf{Z}$.

3. Students can refer to the derivation of Formula (13) at the end of the document. However, to implement the features of the FC layer, Formula (13) is sufficient.

$$\Delta W = \Delta\mathbf{y} \otimes \mathbf{x}^T \tag{13}$$

5.2.3.c Calculating ΔW for a Batch of Data

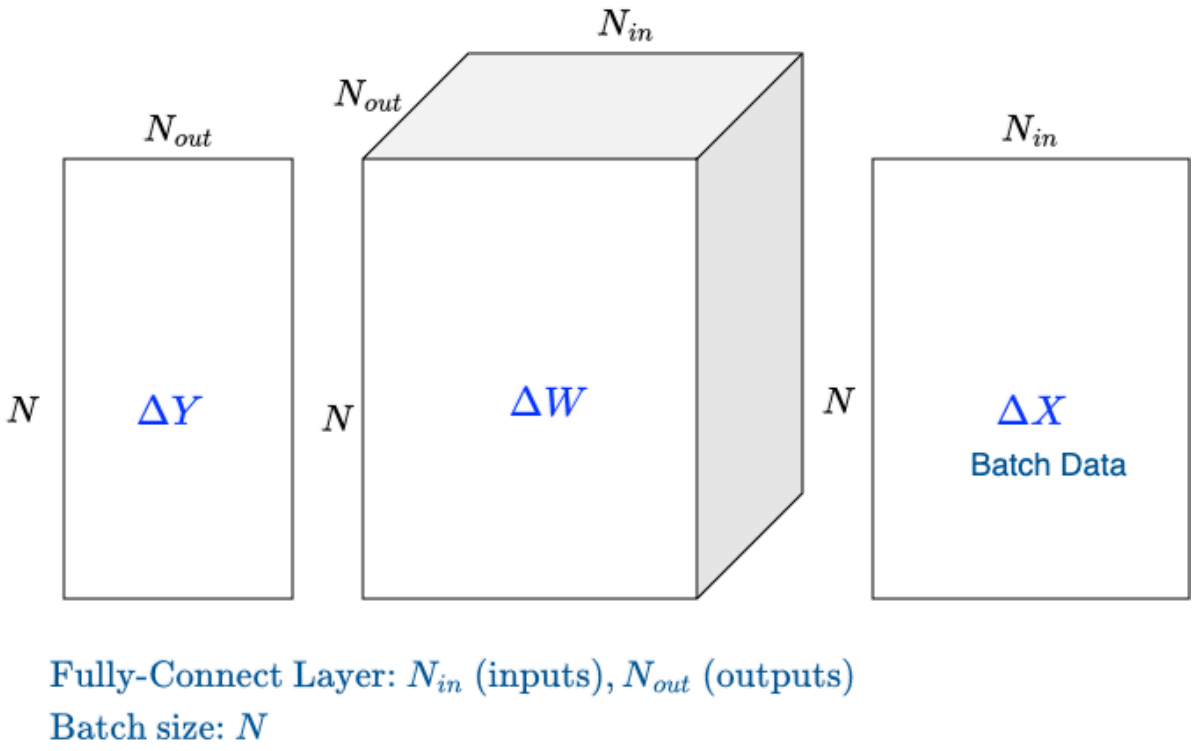
In the previous section, we saw that ΔW calculated for a single data sample is already a matrix. Thus, to store ΔW for a batch of data, we need a 3D array. The first dimension has a size of N, equal to the batch size. The other two dimensions match the dimensions of the weight matrix W. That is, ΔW has the shape: (N, N_{out}, N_{in}).

However, after calculating this block, we take the average along the first dimension and only store the average matrix ΔW. We should also keep track of the total number of samples in the batch, N, so that we can use it to compute the average ΔW across multiple batches of data.

5.2.3.d Calculating $\Delta\mathbf{b}$

Calculating $\Delta\mathbf{b}$ is much simpler compared to calculating ΔW. When a feature vector is passed through the layer, $\Delta\mathbf{b}$ is calculated using Formula (14).

When a batch of N feature vectors is passed through the layer, $\Delta\mathbf{b}$ is computed using



Hình 11: How to compute ΔW for a batch of data.

Formula (15). However, we then average over N vectors in $\Delta\mathbf{b}$, storing only the average vector $\Delta\mathbf{b}$; we also need to keep track of the number of samples in the batch to support accumulating gradients across multiple batches of data.

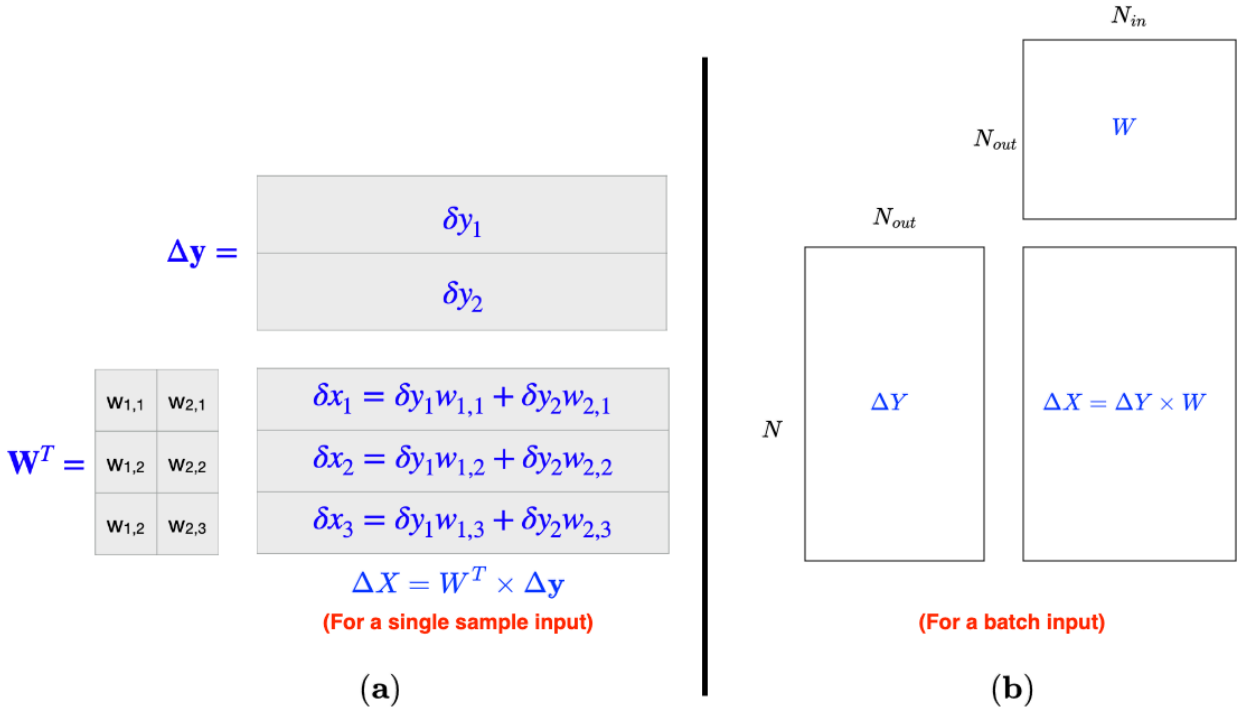
$$\Delta\mathbf{b} = \Delta\mathbf{y} \tag{14}$$

$$\Delta\mathbf{b} = \Delta Y \tag{15}$$

5.2.3.e Calculating ΔX

When passing a feature vector through the FC layer, the calculation of ΔX in the backpropagation step is illustrated in Figure 12 (a). Specifically, we take the transpose of the weight matrix W^T and multiply it with the vector $\Delta\mathbf{y}$ (which is provided during backpropagation to the FC layer).

In the case where a batch of data is passed through the FC layer, during backpropagation, ΔY is a matrix of size $N \times N_{out}$. Formula (16) is used to calculate ΔX from ΔY; specifically, we multiply the matrix ΔY by the weight matrix W. The resulting ΔX has a size of $N \times N_{in}$.



Hình 12: How to compute ΔX.

Some notes:

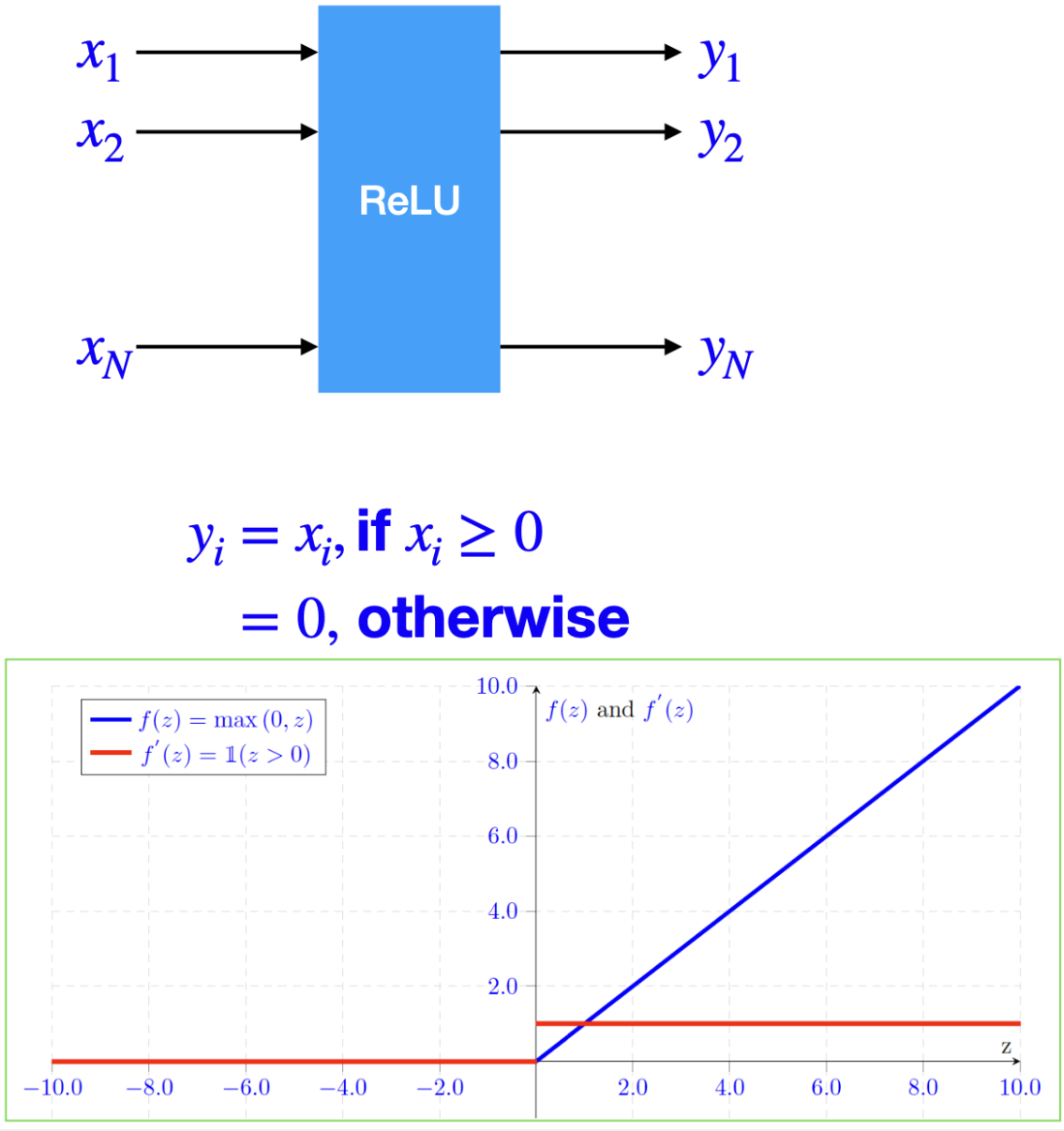
- For ΔW and $\Delta \mathbf{b}$, we need to compute and store the average ΔW and $\Delta \mathbf{b}$ for N samples in the batch. However, for ΔX, we should not average over the number of samples; we keep it as is and continue backpropagation.
- Formula (16) shows that ΔX only depends on ΔY received during backpropagation and the weight matrix W (which is always available in the FC layer). Thus, we do not need to store anything additional to support the computation of ΔX.

$$\Delta X = \Delta Y \times W \tag{16}$$

5.3 ReLU Layer

ReLU is a nonlinear function with no parameters to learn. When this function receives a feature vector or a batch of data of any shape, it performs the following transformation, illustrated in Figure 13:

- For each number in the batch, if the number is greater than or equal to zero, it is retained



Hình 13: The working principle of the ReLU function.

in the output; otherwise, the corresponding output is set to 0.

- Note: ReLU preserves the shape of the input data for the output.

ReLU is quite simple to implement and can be described as follows:

1. **Forward Propagation**: Consists of two main steps,
   (a) Create a mask M using Formula (17). This mask consists of `true (1)` and `false (0)` values corresponding to non-negative and negative values in X, respectively. This mask has the same shape as X. If the neural network is in the training phase,

it also needs to be cached to support the calculation of DY.

(b) The output Y of ReLU is calculated using Formula (18), where $\odot$ represents element-wise multiplication. This operation corresponds to the $*$ operator in many multidimensional array libraries.

2. **Backward Propagation**: Using the cached mask M, DX is calculated using Formula (19). The meaning of this multiplication is that the gradient only flows through the activated neurons (those with values greater than or equal to zero during forward propagation).

$$M = X \geq 0 \qquad \text{(M: a mask, cached)} \tag{17}$$

$$Y = M \odot X \qquad \text{(forward)} \tag{18}$$

$$DX = M \odot DY \qquad \text{(backward)} \tag{19}$$

5.4 Sigmoid Layer

The Sigmoid function is a nonlinear function with no parameters to learn. Similar to many other functions in this category, it processes input numbers independently and retains the shape of X in the input as Y in the output.

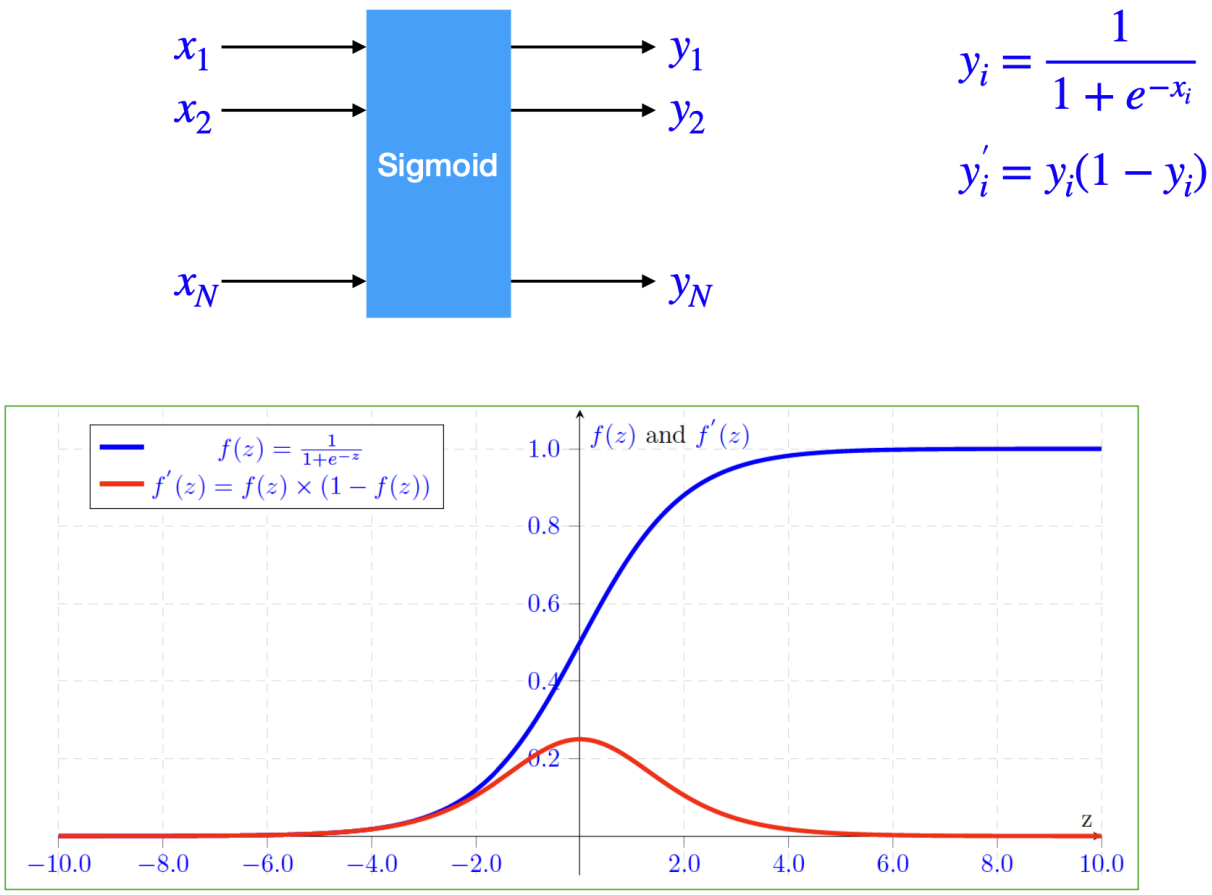
When a batch of data X is inputted, the forward and backward propagation steps are calculated using Formulas (20) and (21), respectively.

$$Y = \frac{1}{1 + \exp(-X)} \qquad \text{(forward)} \tag{20}$$

$$DX = DY \odot Y \odot (1 - Y) \qquad \text{(backward)} \tag{21}$$

As shown in Figure 14, the derivative of the Sigmoid function is relatively small, with a maximum value of less than 0.3 (at $z = 0$); it approaches zero when the input lies outside the range $(-6.0, +6.0)$. Consequently, this small derivative must be multiplied with other local derivatives (during backpropagation) to compute the final gradient. The small values can also lead to the vanishing gradient problem, making it challenging to train neural networks.

The Sigmoid function transforms values in the input range $(0, 1)$ at the output, so besides serving as a nonlinear layer in the middle, it is also used as the output of multi-label classification



Hình 14: The working principle of the Sigmoid function.

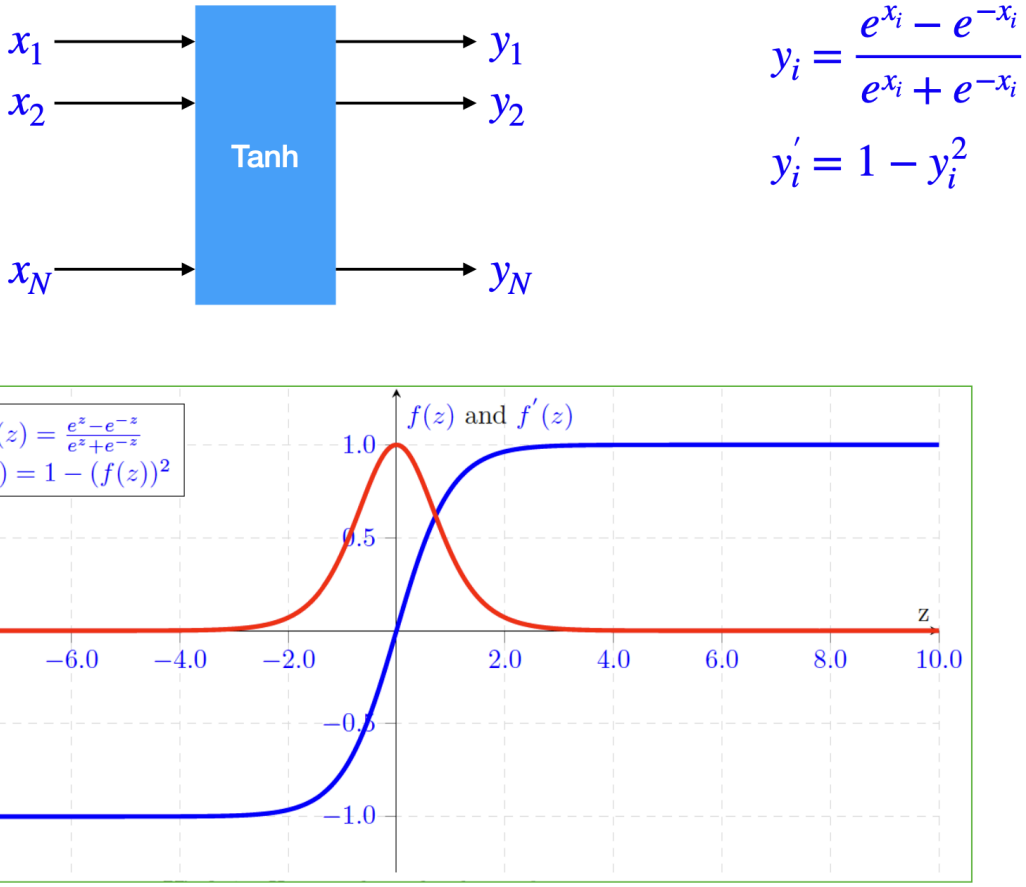
models. Specifically, it predicts the probability of the presence (or absence) of certain labels for the input data sample.

5.5 Tanh Layer

Figures 15 and 14 show that the graph of the Tanh function is quite similar to that of the Sigmoid function, with the distinction that Tanh maps input data to the range $(-1, 1)$ instead of $(0, 1)$ as with the Sigmoid function. In terms of computation, when a batch of data X is input, forward and backward propagation are calculated using Formulas (22) and (23).

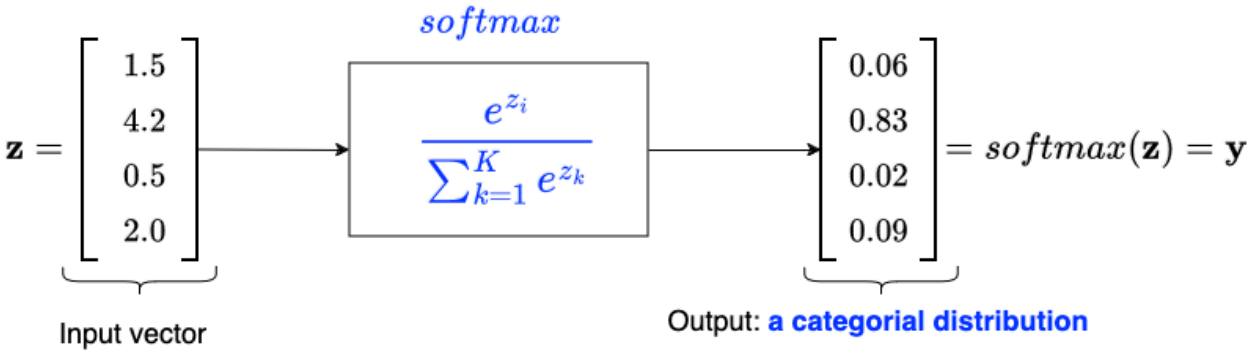
$$Y = \frac{\exp(X) - \exp(-X)}{\exp(X) + \exp(-X)} \quad \text{(forward)} \tag{22}$$

$$DX = DY \odot (1 - Y \odot Y) \quad \text{(backward)} \tag{23}$$



Hình 15: The working principle of the Tanh function.

5.6 Softmax Layer



Hình 16: The Softmax function.

Softmax is a mathematical function defined in Formula (24). The meaning and computation for an input vector are illustrated in Figure 16. An important property of the Softmax function is that it transforms an input vector (with arbitrary values) into a probability distribution at the

output. Consequently, the numbers in the returned vector satisfy the following two constraints:

1. Each value is in the range $(0, 1)$.
2. The sum of the values is 1.

The distribution returned by Softmax is called a **categorical distribution**, meaning that each number represents the probability of a particular class in a single-label classification task. For example, consider the classification task for the following four animals in order: `Cat`, `Chicken`, `Dog`, and `Horse`. The probability distribution shown in Figure 16 indicates that the model believes there is a 6% chance that the data sample is labeled `Cat`, an 83% chance that it is `Chicken`, a 2% chance that it is `Dog`, and a 9% chance that it is `Horse`.

$$\mathbf{y} = softmax(\mathbf{z}) \quad | \quad \mathbf{z} = [z_1, z_2, \cdots, z_K]^T, \quad y_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}, \quad \mathbf{y} = [y_1, y_2, \cdots, y_K]^T \tag{24}$$

$$\Delta\mathbf{z} = (\text{DIAG}(\mathbf{y}) - \mathbf{y} \otimes \mathbf{y}^T) \times \Delta\mathbf{y} \tag{25}$$

During backward propagation, $\Delta\mathbf{z}$ (where $\mathbf{z}$ is the input vector of softmax) is calculated using Formula (25), where $\text{DIAG}(\mathbf{y})$ generates a square matrix with zeros everywhere except for its main diagonal, which contains the vector $\mathbf{y}$; $\otimes$ denotes the outer product, and $\times$ indicates the matrix multiplication between $(\text{DIAG}(\mathbf{y}) - \mathbf{y} \otimes \mathbf{y}^T)$ and vector $\mathbf{y}$. In the case where the input is a batch of data (for example, a matrix), we compute Formulas (24) and (25) for each row independently.

6 Loss Function

6.1 Cross-Entropy Loss

Cross-Entropy is a loss function used in training neural networks and deep learning for single-label classification tasks. As defined in Formula (26), this function takes two matrices: Y and T; where Y is the model's prediction for a batch of input data, and T is the labels for that batch of data. More specifically,

- Y:
  - Y is a matrix with N rows and K columns; with N being the number of samples in the batch and K being the number of classes in the classification task. We often

represent matrix Y as a sequence of vectors $\mathbf{y}_i$; each $\mathbf{y}_i^T$ is a row in matrix Y; here, $i = 1, 2, \cdots, N$.

- The Cross-Entropy function assumes that the model has a final layer that is Softmax, producing a probability distribution across the classes of the task. Therefore, the rows of matrix Y contain numbers in the range $(0, 1)$ that sum to 1.

- T:
  - T is also a matrix with the same dimensions as Y, that is, $N \times K$. Each row is called vector $\mathbf{t}_i^T$, where $i = 1, 2, \cdots, N$; it is also a probability distribution.
  - $\mathbf{t}_i$ encodes the labels in probabilistic form. For example, if $K = 4$, and the list of labels is ["Cat", "Chicken", "Dog", "Horse"] (in that order), the label encoding can be understood as follows:
    * If the i^{th} data sample has a unique label of "Dog", then $\mathbf{t}_i = [0, 0, 1, 0]^T$; meaning that 100% of labelers believe that the label must be "Dog". This case is called "hard-label". If all samples in the dataset are "hard-label", then Formula (27) can also be used to calculate cross-entropy.
    * If the labelers are uncertain about assigning a label to the i^{th} data sample; for example, 80% they believe it is "Cat" and 20% it could be "Dog", then $\mathbf{t}_i = [0.8, 0, 0.2, 0]^T$. This case is called "soft-label", and Formula (26) should be used to calculate cross-entropy.

Formulas (26) and (27) return a real value, representing the computable loss for a batch of data with N samples; where N_{norm} can be 1 if we want the total loss across samples (also known as "reduce=sum"). If we want to take the average ("reduce=mean"), then $N_{norm} = N$. In Formula (26), we require the dot product operation $\cdot$, which is the scalar multiplication of two vectors. In Formula (27), the component y_{i,t_i} is the element at row i and column t_i in matrix Y; where t_i is the label for the i^{th} data sample. Note that Formula (27) is used for the "hard-label" case, so the label vector $\mathbf{t}$ only stores the **index** of the label in the label list (in order); meaning that t_i is an integer in $\{0, 1, \cdots, (K-1)\}$.

The backward propagation step calculates $\Delta\mathbf{y}$ for each data sample as shown in Formula (28). Here, the division between two vectors is element-wise division, and EPSILON is a small value to avoid division by 0, e.g.,

$$\text{CE}(Y,T) = -\frac{1}{N_{norm}}\sum_{i=1}^{N}\mathbf{t}_i \cdot \log\left(\mathbf{y}_i\right) \quad | \quad Y = [\mathbf{y}_1^T, \mathbf{y}_2^T, \cdots, \mathbf{y}_N^T], T = [\mathbf{t}_1^T, \mathbf{t}_2^T, \cdots, \mathbf{t}_N^T], \tag{26}$$

$$\text{CE}(Y,\mathbf{t}) = -\frac{1}{N_{norm}}\sum_{i=1}^{N}\log\left(y_{i,t_i}\right) \quad | \quad Y = [\mathbf{y}_1^T, \mathbf{y}_2^T, \cdots, \mathbf{y}_N^T], \mathbf{t} = [t_1, \cdots, t_i, \cdots t_N], \tag{27}$$

$$\Delta y = -\frac{1}{N_{norm}} \times \frac{\mathbf{t}}{\mathbf{y} + \text{EPSILON}} \tag{28}$$

6.2 Binary Cross-Entropy (BCE)

Binary Cross-Entropy is used in training neural network models and for performing two-class classification or multi-label classification tasks. This function takes two vectors, $\mathbf{y}$ and $\mathbf{t}$, representing predictions and labels, respectively. The vector $\mathbf{y}$ contains probability values within the range $(0, 1)$; thus, BCE assumes that the model has used the Sigmoid function to convert the outputs into probabilities. The values in $\mathbf{t}$ represent probability distributions for the labels. If it is a "hard-label", then the values in $\mathbf{t}$ are either 0 (probability = 0) or 1 (probability = 1).

Binary Cross-Entropy is calculated according to Formula (29); where N_{norm} equals 1 if calculating the total loss and equals N if calculating the average loss.

In the backward propagation step, BCE is calculated according to Formula (30).

$$\text{BCE}(\mathbf{y},\mathbf{t}) = -\frac{1}{N_{norm}} \times \sum_{i=1}^{N}\left[t_i \times \log y_i + (1-t_i) \times \log\left(1-y_i\right)\right] \tag{29}$$

$$\Delta\mathbf{y} = [\delta y_1, \cdots, \delta y_i, \cdots, \delta y_N]^T \quad | \quad \delta y_i = -\frac{1}{N_{norm}} \times \left[\frac{t_i}{y_i} - \frac{1-t_i}{1-y_i}\right] \tag{30}$$

6.3 Mean Squared Error

To be updated!

7 Implementation Guide

7.1 Computational Layers

7.1.1 FCLayer Class

```cpp
#include "ann/Layer.h"

class FCLayer: public Layer {
public:
    FCLayer(int Nin=2, int Nout=10, bool use_bias=true);
    FCLayer(const FCLayer& orig);
    virtual ~FCLayer();

    xt::xarray<double> forward(xt::xarray<double> X);
    xt::xarray<double> backward(xt::xarray<double> DY);
    int register_params(IParamGroup* ptr_group);
    bool save(string file);
    bool load(string filename, bool use_bias);


protected:
    virtual void init_weights();

private:
    int m_nNin, m_nNout;
    bool m_bUse_Bias;

    xt::xarray<double> m_aWeights; //N_out x N_in
    xt::xarray<double> m_aBias;

    xt::xarray<double> m_aGrad_W;
    xt::xarray<double> m_aGrad_b;
    xt::xarray<double> m_aCached_X;
    unsigned long long m_unSample_Counter;
};
```

Hình 17: **FCLayer**: Fully Connected Layer

The `FCLayer` class is defined in Figure 17. The member variables are explained as follows:

- `m_nNin` and `m_nNout`: these two variables store the number of inputs and outputs of the fully connected layer, respectively. They correspond to N_{in} and N_{out} in Section 5.2.1. These variables must be initialized in the constructor using the corresponding parameters.
- `m_bUse_Bias`: this `bool` variable indicates whether the fully connected layer uses a bias; it must be assigned in the constructor from the corresponding variable.

- `m_aWeights` and `m_aBias`: these two variables represent the weights and bias of the fully connected layer. They must be created and initialized in the constructor.
  - `m_aWeights`: It must be created to have a shape of (N_{out}, N_{in}), and initialized with random numbers according to a standard normal distribution[23].
  - `m_aBias`: **If the variable `m_bUse_Bias` is true**, it must be created to have a shape of $(N_{out},)$, and initialized with zeros[24].
- `m_aGrad_W`: this is ΔW in Section 5.2.3.c. This variable must be created and initialized to 0 in the constructor[25].
- `m_aGrad_b`: this is $\Delta \mathbf{b}$ in Section 5.2.3.d. **If the variable `m_bUse_Bias` is true**, it must be created and initialized to 0 in the constructor[26].
- `m_aCached_X`: **If the variable `is_training` is true**, the `forward` function must store X at the input in this variable to be used when calculating ΔW in the `backward` function.
- `m_unSample_Counter`: This variable keeps track of the total number of samples that have been processed **backward**. **Note**: it only accumulates the number of data samples across batches when **backward** is performed on a batch. This means that `m_unSample_Counter` must be updated in the **backward** function instead of **forward**.

The methods of the `FCLayer` class are explained as follows:

- `forward`:
  - Input: `xt::xarray<double> X`
  - Performs:
    - Assigns `X` to `m_aCached_X` if in training mode.
    - Computes the response according to Formula 12. Note that do not add **b** if bias is not used. It is recommended to use the function **xt::linalg::tensordot** to calculate XW^T in Formula 12.
- `backward`:
  - Input: `xt::xarray<double> DY`. `DY` is the derivative of the loss function with respect to the output of `FCLayer`.
  - Performs:
    - Updates the variable `m_unSample_Counter`
    - Updates the variable `m_aGrad_W` according to Formula 13.

[23] using the function `xt::random::randn<double>(.)`
[24] using the function `xt::zeros<double>(.)`
[25] using the function `xt::random::zeros<double>(.)`
[26] using the function `xt::random::zeros<double>(.)`

```cpp
#include "ann/Layer.h"

class ReLU: public Layer {
public:
    ReLU();
    ReLU(const ReLU& orig);
    virtual ~ReLU();

    xt::xarray<double> forward(xt::xarray<double> X);
    xt::xarray<double> backward(xt::xarray<double> DY);
private:
    xt::xarray<bool> m_aMask;
};
```

Hình 18: **ReLU**: Non-linear Layer

```cpp
#include "ann/Layer.h"

class Sigmoid: public Layer {
public:
    Sigmoid();
    Sigmoid(const Sigmoid& orig);
    virtual ~Sigmoid();

    xt::xarray<double> forward(xt::xarray<double> X);
    xt::xarray<double> backward(xt::xarray<double> DY);
private:
    xt::xarray<double> m_aCached_Y;
};
```

Hình 19: **Sigmoid**: Non-linear Layer

* Updates the variable `m_aGrad_b` (if the variable `m_bUse_Bias` is true) according to Formula 15
* Computes and returns ΔX, according to Formula 16.

7.1.2 ReLU Layer

To be updated!

7.1.3 Sigmoid Layer

7.1.4 Tanh Layer

```cpp
#include "ann/Layer.h"

class Tanh: public Layer  {
public:
    Tanh();
    Tanh(const Tanh& orig);
    virtual ~Tanh();

    xt::xarray<double> forward(xt::xarray<double> X);
    xt::xarray<double> backward(xt::xarray<double> DY);
private:
    xt::xarray<double> m_aCached_Y;
};
```

Hình 20: **Tanh**: Non-linear Layer

```cpp
#include "ann/Layer.h"

class Softmax: public Layer {
public:
    Softmax(int axis=-1);
    Softmax(const Softmax& orig);
    virtual ~Softmax();

    virtual xt::xarray<double> forward(xt::xarray<double> X);
    virtual xt::xarray<double> backward(xt::xarray<double> DY);

private:
    int m_nAxis;
    xt::xarray<double> m_aCached_Y;
};
```

Hình 21: **Softmax**: Output of the classifier

7.1.5 Softmax Layer

7.2 Loss Layers

7.2.1 LossLayer

7.3 CrossEntropy Layer

```cpp
#include "ann/xtensor_lib.h"
enum LossReduction{
    REDUCE_NONE = 0,
    REDUCE_SUM,
    REDUCE_MEAN
};

class LossLayer {
public:
    LossLayer(LossReduction reduction=REDUCE_MEAN);
    LossLayer(const LossLayer& orig);
    virtual ~LossLayer();

    virtual double forward(xt::xarray<double> X, xt::xarray<double> t)=0;
    virtual xt::xarray<double> backward()=0;
private:
    LossReduction reduction;
};
```

Hình 22: **LossLayer**: Base class for all loss layers

```cpp
#include "tensor/xtensor_lib.h"

enum LossReduction{
    REDUCE_NONE = 0,
    REDUCE_SUM,
    REDUCE_MEAN
};

class LossLayer {
public:
    LossLayer(LossReduction reduction=REDUCE_MEAN);
    LossLayer(const LossLayer& orig);
    virtual ~LossLayer();

    virtual double forward(xt::xarray<double> X, xt::xarray<double> t)=0;
    virtual xt::xarray<double> backward()=0;
private:
    LossReduction m_eReduction;
};
```

Hình 23: **CrossEntropy**

7.4 Model

7.5 Optimizer

——————————HẾT——————————