DEEP LEARNING AND NEURAL NET 2

MINH QUANG LE

3.4 Automatic Differentiation and Its Main Modes

Automatic Differentiation (AD) is a technique used in computational mathematics and computer science for efficiently and accurately evaluating derivatives of functions [71-74]. It plays a crucial role in machine learning, optimization, and scientific computing. AD is also known as algorithmic differentiation or autodiff. BP is really a special case of automatic differentiation. It may be somewhat easier to understand the basic idea of BP by seeing the more general algorithm first. For that reason, we will first talk about autodiff.

The core Idea [71]**:** Automatic differentiation exploits the fact that all numerical computations, no matter how complicated, are ultimately compositions of a finite set of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exponential, logarithmic, trigonometric, etc.), for which derivatives are known, and combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition. By doing so, AD can automatically and accurately calculate gradients, even for functions with intricate compositions and nested operations.

For example. Consider the scalar function

$$f(x) = \exp(\exp(x) + \exp(x)^2) + \sin(\exp(x) + \exp(x)^2). \tag{3.43}$$

By introducing intermediate variables, you can break down complex expressions into simpler steps, making it easier to apply the chain rule and compute derivatives. Say

$$a = \exp(x), \quad b = a^2, \quad c = a + b, \quad d = \exp(c), \quad e = \sin(c), \tag{3.44.1}$$

and finally, we have

$$f = d + e. \tag{3.44.2}$$

Fundamental to automatic differentiation is the decomposition of differentials provided by the chain rule of partial derivatives of composite functions. For the simple composition

$$\begin{aligned} y &= g_3\left(g_2(g_1(x))\right) \\ &= g_3\left(g_2(g_1(v_0))\right) \\ &= g_3(g_2(v_1)) \\ &= g_3(v_2) \\ &= v_3, \end{aligned} \tag{3.45}$$

where

$$v_0 = x, \qquad v_1 = g_1(v_0), \qquad v_2 = g_2(v_1), \qquad v_3 = g_3(v_2) = y, \tag{3.46}$$

the chain rule gives

$$\begin{aligned} \frac{\partial y}{\partial x} &= \frac{\partial y}{\partial v_2}\left(\frac{\partial v_2}{\partial v_1}\frac{\partial v_1}{\partial x}\right) \\ &= \left(\frac{\partial y}{\partial v_2}\frac{\partial v_2}{\partial v_1}\right)\frac{\partial v_1}{\partial x}. \end{aligned} \tag{3.47}$$

Usually, two distinct modes of automatic differentiation are presented.

Forward Mode:

- Forward accumulation (also called bottom-up, forward mode, or tangent mode)
- In forward mode, the computation proceeds from the input variables to the output variables.
- Forward accumulation specifies that one traverses the chain rule from inside to outside (that is, first compute $\partial v_1/\partial x$ and then $\partial v_2/\partial v_1$ and at last $\partial y/\partial v_2$.

- More succinctly, forward accumulation computes the recursive relation: $\frac{\partial v_i}{\partial x} = \frac{\partial v_i}{\partial v_{i-1}} \frac{\partial v_{i-1}}{\partial x}$ with $v_3 = y$.
- In forward accumulation AD, one first fixes the independent variable with respect to which differentiation is performed and computes the derivative of each sub-expression recursively. This involves repeatedly substituting the derivative of the inner functions in the chain rule:

$$\begin{aligned} \frac{\partial y}{\partial x} &= \frac{\partial y}{\partial v_{n-1}} \frac{\partial v_{n-1}}{\partial x} \\ &= \frac{\partial y}{\partial v_{n-1}} \left(\frac{\partial v_{n-1}}{\partial v_{n-2}} \frac{\partial v_{n-2}}{\partial x} \right) \\ &= \frac{\partial y}{\partial v_{n-1}} \left(\frac{\partial v_{n-1}}{\partial v_{n-2}} \left(\frac{\partial v_{n-2}}{\partial v_{n-3}} \frac{\partial v_{n-3}}{\partial x} \right) \right) \\ &= \cdots. \end{aligned} \tag{3.48}$$

- In forward accumulation, the quantity of interest is the tangent, denoted with a dot $\dot{v}_i$; it is a derivative of a subexpression with respect to a chosen independent variable

$$\dot{v}_i = \frac{\partial v_i}{\partial x}. \tag{3.49}$$

- Forward accumulation is particularly useful when the number of input variables is much smaller than the number of output variables (for functions $f: \mathbb{R}^n \to \mathbb{R}^m$ with $n \ll m$).

Reverse Mode (or Backward Mode):

- Reverse accumulation (also called top-down, reverse mode, or adjoint mode)
- In reverse mode, the computation proceeds backward from the output variables to the input variables.
- Reverse accumulation has the traversal from outside to inside (first compute $\partial y/\partial v_2$ and then $\partial v_2/\partial v_1$ and at last $\partial v_1/\partial x$.

- More succinctly, reverse accumulation computes the recursive relation: $\frac{\partial y}{\partial v_i} = \frac{\partial y}{\partial v_{i+1}} \frac{\partial v_{i+1}}{\partial v_i}$ with $v_0 = x$.
- In reverse accumulation AD, the dependent variable to be differentiated is fixed and the derivative is computed with respect to each sub-expression recursively. The derivative of the outer functions is repeatedly substituted in the chain rule:

$$\begin{aligned} \frac{\partial y}{\partial x} &= \frac{\partial y}{\partial v_1} \frac{\partial v_1}{\partial x} \\ &= \left(\frac{\partial y}{\partial v_2} \frac{\partial v_2}{\partial v_1} \right) \frac{\partial v_1}{\partial x} \\ &= \left(\left(\frac{\partial y}{\partial v_3} \frac{\partial v_3}{\partial v_2} \right) \frac{\partial v_2}{\partial v_1} \right) \frac{\partial v_1}{\partial x} \\ &= \cdots. \end{aligned} \tag{3.50}$$

- In reverse accumulation, the quantity of interest is the adjoint, denoted with a bar $\bar{v}_i$; it is a derivative of a dependent variable with respect to a subexpression

$$\bar{v}_i = \frac{\partial y}{\partial v_i}. \tag{3.51}$$

- Reverse accumulation is especially efficient when the number of output variables is much smaller than the number of input variables (for functions $f: \mathbb{R}^n \to \mathbb{R}^m$ with $n \gg m$).

Example of Forward Mode

Step by step forward mode

- Express your function as a composition of elementary operations (addition, multiplication, etc.).
- Choose the variables with respect to which you want to compute derivatives (seeds) (usually input variables).
- Execute the computation graph in a forward direction, evaluating both the function values and the derivatives of the elementary operations with respect to the seed.
- Store intermediate values (function values and derivatives) at each node in the computation graph.
- Use the chain rule to propagate the derivatives through the computation graph.
- Combine the stored intermediate values to compute the derivative of the overall function with respect to each input variable.
- Extract the derivatives of interest. These derivatives represent the gradient of the function with respect to the chosen input variables.

On the left hand side of Table 3.1 we see the representation of the computation $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ as an evaluation trace of elementary operations—also called a Wengert list and graphically represented in Figure 3.13. For computing the derivative of f with respect to x_1, we start by associating with each intermediate variable v_i a derivative $\dot{v}_i = \frac{\partial v_i}{\partial x_1}$. Applying the chain rule to each elementary operation in the forward primal trace, we generate the corresponding tangent (derivative) trace, given on the right-hand side in Table 3.1. Evaluating the

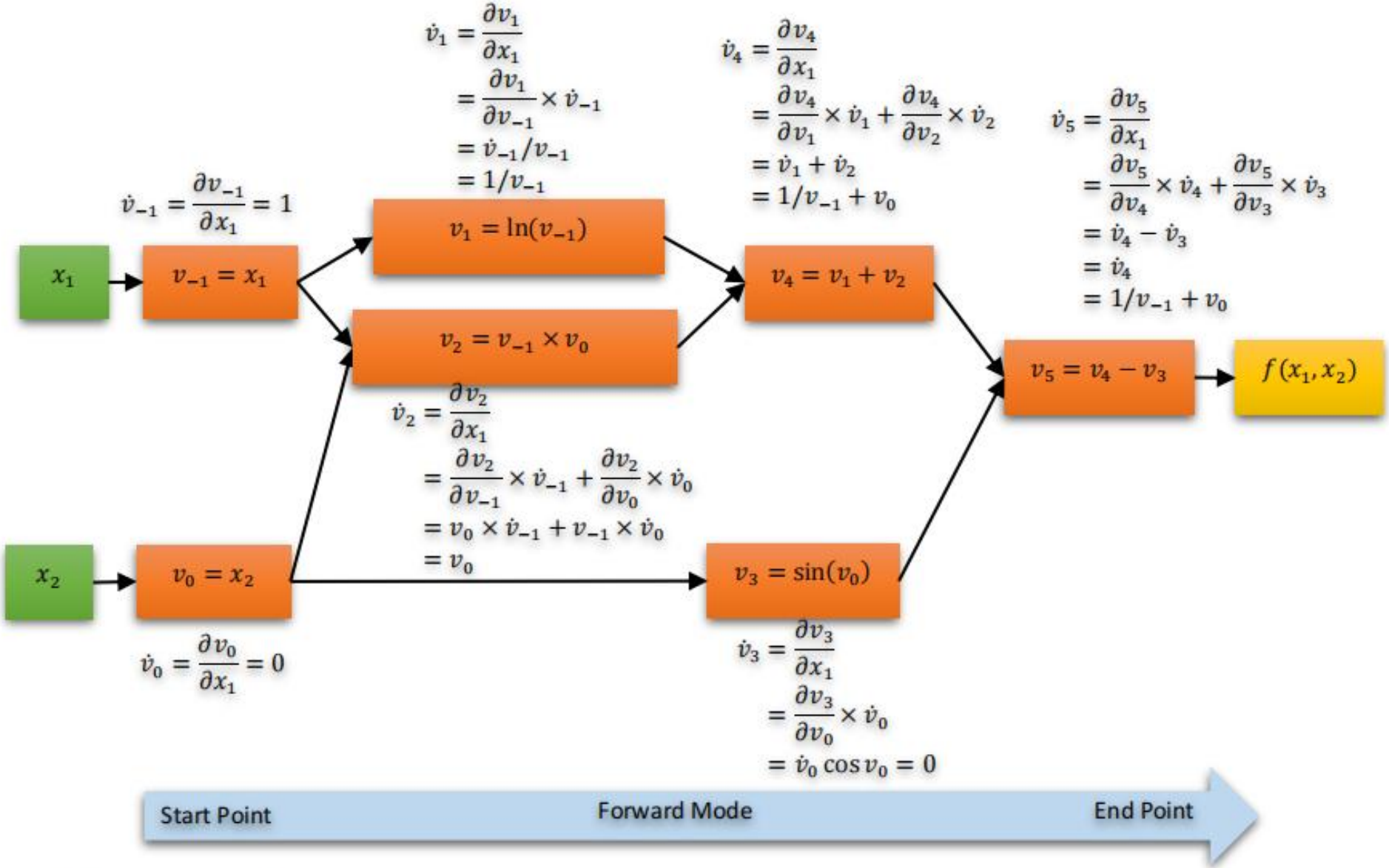

Figure 3.13. Example of forward accumulation with computational graph of $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. The choice of the independent variable to which differentiation is performed affects the seed values $\dot{v}_{-1}$ and $\dot{v}_0$. Given interest in the derivative of this function with respect to x_1, the seed values should be set to: $\dot{v}_{-1} = \dot{x}_1 = 1$, $\dot{v}_0 = \dot{x}_2 = 0$.

Table 3.1. Forward mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2,5)$ and setting $\dot{x}_1 = 1$ to compute $\partial y / \partial x_1$, see Figure 3.13. The original forward evaluation of the primals on the left is augmented by the tangent operations on the right, where each line complements the original directly to its left.

Forward Primal Trace		Forward Tangent (Derivative) Trace	
$v_{-1} = x_1$	$= 2$	$\dot{v}_{-1} = \dot{x}_1$	$= 1$
$v_0 = x_2$	$= 5$	$\dot{v}_0 = \dot{x}_2$	$= 0$
$v_1 = \ln v_{-1}$	$= \ln 2$	$\dot{v}_1 = \dot{v}_{-1} / v_{-1}$	$= 1/2$
$v_2 = v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2 = \dot{v}_{-1} \times v_0 + v_{-1} \times \dot{v}_0$	$= 1 \times 5 + 0 \times 2$
$v_3 = \sin v_0$	$= \sin 5$	$\dot{v}_3 = \dot{v}_0 \cos v_0$	$= 0 \times \cos 5$
$v_4 = v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4 = \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
$v_5 = v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5 = \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
$y = v_5$	$= 11.652$	$\dot{y} = \dot{v}_5$	$= 5.5$

Example of Reverse Mode

Returning to the example $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$, see Figure 3.14. In Table 3.2, we see the adjoint statements on the right-hand side, and original elementary operation on the left-hand side. In simple terms, we are interested in computing the contribution $\bar{v}_i = \frac{\partial y}{\partial v_i}$ of the change in each variable v_i to the change in the output y. After the forward pass on the left-hand side, we run the reverse pass of the adjoints on the right-hand side, starting with $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$. In the end we get the derivatives $\frac{\partial y}{\partial x_1} = \bar{x}_1$ and $\frac{\partial y}{\partial x_2} = \bar{x}_2$ in just one reverse pass.

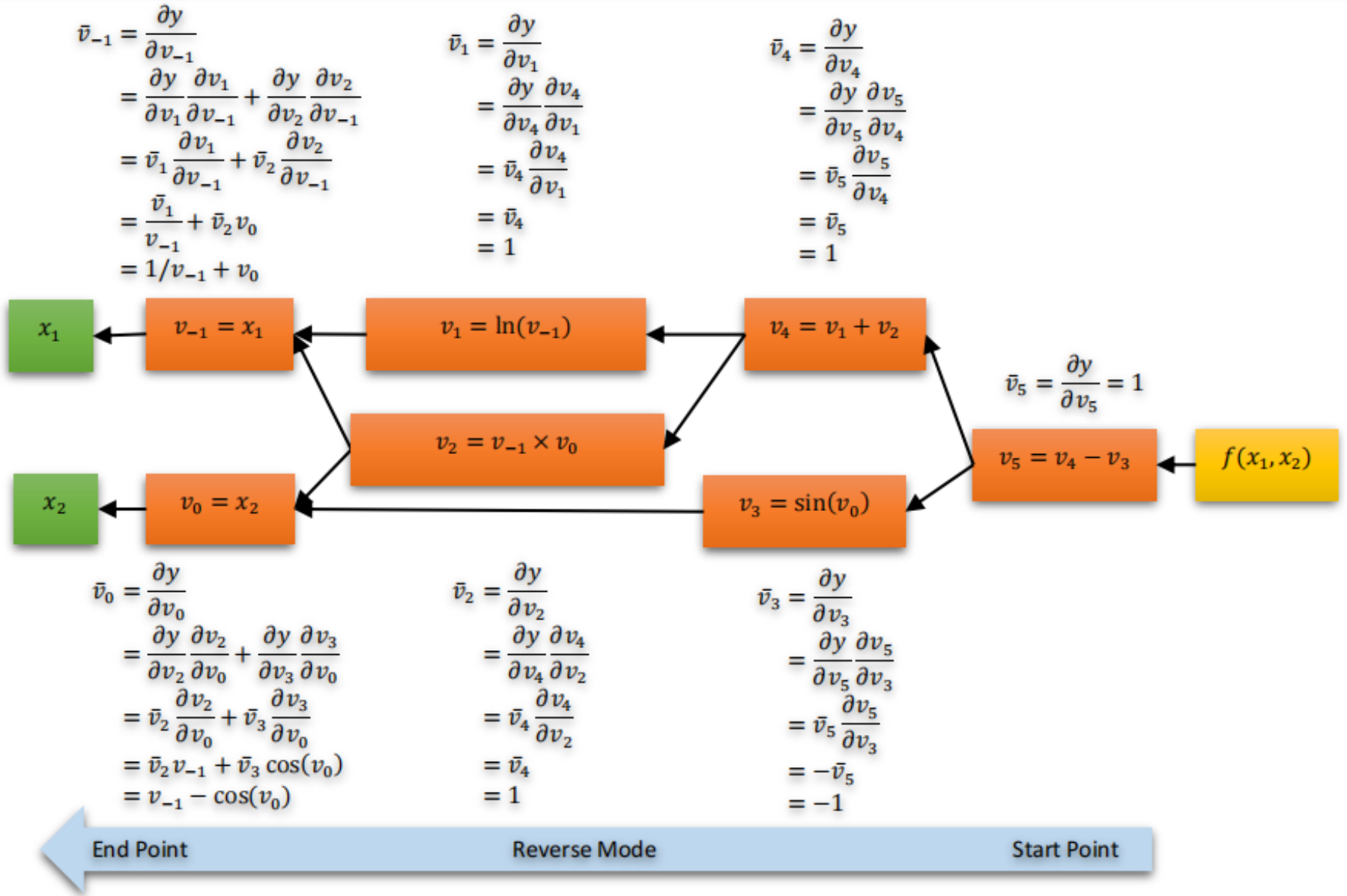

Figure 3.14. Example of reverse accumulation with computational graph of $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$. Reverse accumulation evaluates the function first and calculates the derivatives with respect to all independent variables in an additional pass. The adjoint, $\bar{v}_i$, is a derivative of a dependent variable with respect to a subexpression $\bar{v}_i = \frac{\partial y}{\partial v_i}$.

Table 3.2. Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2,5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated (see Figure 3.14). Note that both $\partial y/\partial x_1$ and $\partial y/\partial x_2$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \partial y/\partial y = 1$.

Forward Primal Trace	Reverse Adjoint (Derivative) Trace
$v_{-1} = x_1 = 2$ $v_0 = x_2 = 5$	$\bar{v}_5 = \bar{y} = 1$ $\bar{v}_4 = \bar{v}_5 \dfrac{\partial v_5}{\partial v_4} = \bar{v}_5 \times (1) = 1$
$v_1 = \ln v_{-1} = \ln 2$	$\bar{v}_3 = \bar{v}_5 \dfrac{\partial v_5}{\partial v_3} = \bar{v}_5 \times (-1) = -1$
$v_2 = v_{-1} \times v_0 = 2 \times 5$	$\bar{v}_1 = \bar{v}_4 \dfrac{\partial v_4}{\partial v_1} = \bar{v}_4 \times 1 = 1$
$v_3 = \sin v_0 = \sin 5$	$\bar{v}_2 = \bar{v}_4 \dfrac{\partial v_4}{\partial v_2} = \bar{v}_4 \times 1 = 1$
$v_4 = v_1 + v_2 = 0.693 + 10$	$\bar{v}_0 = \bar{v}_3 \dfrac{\partial v_3}{\partial v_0} + \bar{v}_2 \dfrac{\partial v_2}{\partial v_0} = \bar{v}_3 \cos v_0 + \bar{v}_2 v_{-1} = 1.716$
$v_5 = v_4 - v_3 = 10.693 + 0.959$	$\bar{v}_{-1} = \bar{v}_2 \dfrac{\partial v_2}{\partial v_{-1}} + \bar{v}_1 \dfrac{\partial v_1}{\partial v_{-1}} = \bar{v}_2 v_0 + \dfrac{\bar{v}_1}{v_{-1}} = 5.5$
$y = v_5 = 11.652$	$\bar{x}_2 = \bar{v}_0 = 1.716$ $\bar{x}_1 = \bar{v}_{-1} = 5.5$

Remarks:

- Reverse accumulation traverses the chain rule from outside to inside. The example function is scalar-valued, and thus there is only one seed for the derivative computation, and only one sweep of the computational graph is needed to calculate the (two-component) gradient. This is only half the work when compared to forward accumulation, but reverse accumulation requires the storage of the intermediate variables v_i as well as the instructions that produced them in a data structure known as a "tape" or a Wengert list, which may consume significant memory if the computational graph is large.
- Forward mode requires a new graph for each input x_i, to compute the partial derivative $\frac{\partial y}{\partial x_i}$.
- The reverse mode starts with the output y. It computes the derivatives with respect to both inputs. The computations go backward through the graph. That means it does not follow the empty line that started with $\partial x_2/\partial x_1 = 0$ in the forward graph for x_1-derivatives. And it would not follow the empty line $\partial x_1/\partial x_2 = 0$ in the forward graph for x_2-derivatives.
- A larger and more realistic problem with N inputs will have N forward graphs, each with $N-1$ empty lines (because the N inputs are independent). The derivative of x_i with respect to every other input x_j is $\partial x_i/\partial x_j = 0$. Instead of N forward graphs from N inputs, we will have one backward graph from one output. This is the success of reverse mode.
- Because machine learning practice principally involves the gradient of a scalar-valued objective with respect to a large number of parameters, this establishes the reverse mode, as opposed to the forward mode, as the mainstay technique in the form of the BP algorithm.

for the purpose of gradient-based optimization, such as GD. Closed-form solutions, which would provide an explicit expression for these derivatives, are often elusive due to the intricate nature of the composition of functions in the graph. The reverse mode of automatic differentiation or BP addresses this issue by recursively applying the chain rule. In a computational graph representing a NN, the chain rule needs to be applied repeatedly. This is because the output is typically a complex function of many intermediate variables, and the chain rule helps in breaking down the derivatives with respect to each variable. Instead of explicitly writing out the entire function and differentiating it with respect to each parameter, BP allows the algorithm to traverse the computational graph backward, computing the derivatives at each step. This way, the complexity of the computation is managed automatically, and you can efficiently calculate the gradients needed for optimization without having to deal with the explicit closed form. In essence, BP leverages the structure of the computational graph to compute gradients efficiently, making it feasible to train deep and wide NNs without needing to write out and differentiate the entire function manually. The BP algorithm is a cornerstone of modern deep learning.

3.5 Training Process and Loss/Cost Functions

Loss functions, also known as cost functions or objective functions, play a crucial role in training NNs. These functions measure how well the model is performing by quantifying the difference between the predicted values and the actual ground truth values. The loss is essentially a measure of the model's error. The choice of a specific loss function depends on the nature of the problem you are trying to solve. Different tasks, such as regression or classification, may require different loss functions.

Mean Squared Error

The quadratic loss function, also known as squared error loss, or mean squared error (MSE) is indeed commonly used, especially in regression problems and when employing least squares techniques. The mathematical formula is:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{j=1}^{n}(y_j - a_j)^2, \tag{3.52.1}$$

or

$$\mathcal{L}_{\text{MSE}} = c\sum_{j=1}^{n}(y_j - a_j)^2, \tag{3.52.2}$$

where n is the number of instances in the dataset, y_j is the actual value and a_j is the predicted value, for the j-th instance and c is a constant (the value of the constant makes no difference to a decision and can be ignored by setting it equal to 1). The derivative with respect to a_i is (in the case $c = 1/2$):

$$\begin{aligned}\frac{\partial \mathcal{L}_{\text{MSE}}}{\partial a_i} &= \frac{1}{2}\frac{\partial}{\partial a_i}\sum_{j=1}^{n}(y_j - a_j)^2 \\ &= \frac{1}{2}\frac{\partial}{\partial a_i}(y_i - a_i)^2 \\ &= a_i - y_i. \end{aligned} \tag{3.53}$$

The summation term over $j = 1$ to n is dropped because the derivatives $\frac{\partial}{\partial a_i}(y_j - a_j)^2$ will be zero for all j except for the case when $j = i$. The quadratic loss has several desirable properties:

- The quadratic form simplifies mathematical operations, making it easier to find analytical solutions and derivatives. This is particularly advantageous in optimization algorithms.
- The loss is symmetric with respect to errors above and below the target. An error of $y_i - a_i$ and $a_i - y_i$ both contribute the same amount to the loss.
- The use of quadratic loss is well-suited for problems where the goal is to minimize the average squared difference between predicted and actual values. However, it is sensitive to outliers, and if your data contains outliers, other loss functions like Huber loss or Tukey's bisquare loss might be more robust.

Binary Cross-Entropy (Logistic Loss):

Binary Cross-Entropy (BCE), also known as logistic loss, is a loss function commonly used in binary classification problems. It is particularly associated with problems where the goal is to classify instances into one of two classes, often denoted as 0 or 1. The loss is calculated for each instance, and the overall loss is the average over all instances in the dataset. The formula for BCE loss is as follows:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{n}\sum_{j=1}^{n} y_j \log(a_j) + (1-y_j)\log(1-a_j), \tag{3.54}$$

where n is the number of instances in the dataset, y_j is the true label for the j-th instance (either 0 or 1) and a_j is the predicted probability that the j-th instance belongs to class 1. This loss function is commonly used in logistic regression and NN models with a Sigmoid AF in the output layer for binary classification. The derivative with respect to a_i is:

$$\begin{aligned}
\frac{\partial}{\partial a_i}\mathcal{L}_{\text{BCE}} &= \frac{\partial}{\partial a_i}\left\{\frac{1}{n}\sum_{j=1}^{n}\left(-y_j \log(a_j) - (1-y_j)\log(1-a_j)\right)\right\} \\
&= \frac{\partial}{\partial a_i}\left\{\frac{1}{n}\left(-y_i \log(a_i) - (1-y_i)\log(1-a_i)\right)\right\} \\
&= \frac{1}{n}\left(-y_i\frac{1}{a_i} - (1-y_i)\frac{(-1)}{(1-a_i)}\right) \\
&= -\frac{1}{n}\left(\frac{y_i}{a_i} - \frac{1-y_i}{1-a_i}\right) \\
&= \frac{1}{n}\frac{a_i - y_i}{a_i(1-a_i)}.
\end{aligned} \tag{3.55}$$

The power of NNs lies in their ability to learn and optimize the parameters jointly through a process called training. The basic training process for a NN using GD and a loss function can describes as follows:

- During training, the network is presented with input data along with the corresponding target outputs. Then, the computations are performed in the forward direction to determine the predicted values of the outputs. If these predicted values are different from the observed values in the training data, compute the loss value. The loss function measures the difference between the predicted output of the NN and the true output (ground truth) for a given input. Mathematically, it is often denoted as $\mathcal{L}_i$, where the subscript i indicates the specific training instance. The overall objective during training is to minimize the average or total loss over all training instances, $\mathcal{L}$. This is often expressed as the average of the individual losses across the entire training dataset. The objective function $\mathcal{L}$, which represents the goal of the optimization, is the sum or average of the individual losses:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_i, \tag{3.56}$$

  where n is the number of training instances.
- Symbolically, we can write the loss function as $\mathcal{L}(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is a vector of all the weights and biases in the network. Our goal is to move through the space that the loss function defines to find the minimum, the specific $\boldsymbol{\theta}$ leading to the smallest loss, $\mathcal{L}$. The gradient of the loss function with respect to the parameters $\boldsymbol{\theta}$ of the model tells us how the loss changes concerning each parameter. GD is an optimization algorithm used to minimize the objective function in NNs. It works by iteratively adjusting the weights of the NN in the direction opposite to the gradient of the objective function with respect to the weights (iteratively moving towards the minimum of the function). This adjustment is proportional to the learning rate, a hyperparameter that determines the step size in the weight update, review Chapter 2.
- Therefore, to train a NN via GD, we need to know how each weight and bias value contributes to the loss function; that is, we need to know $\partial\mathcal{L}/\partial w$ and $\partial\mathcal{L}/\partial b$, for some weight w and bias b.

- The weights of the NN are updated in the direction that reduces the loss. The general update rule for a weight w_{ij} is given by:

$$w_{ij\,\text{new}} = w_{ij\,\text{old}} - \alpha \frac{\partial \mathcal{L}}{\partial w_{ij}}, \qquad \Delta w_{ij} = -\alpha \frac{\partial \mathcal{L}}{\partial w_{ij}}, \tag{3.57}$$

where α is a free parameter (α is the "learning rate" that we set prior to training; it lets us scale our step size according to the problem at hand), and $\partial \mathcal{L}/\partial w_{ij}$ is the partial derivative of the objective function with respect to the weight w_{ij}. This iterative process of computing gradients, updating weights, and repeating the process is performed until the network's performance converges to an acceptable level or a predefined stopping criterion is met. This is the essence of supervised learning in NNs.

- This phase is referred to as the backwards phase. The rationale for calling it a "backwards phase" is that derivatives of the loss with respect to weights near the output (where the loss function is computed) are easier to compute and are computed first. The derivatives become increasingly complex as we move towards edge weights away from the output (in the backwards direction).
- The goal of BP is to compute the partial derivatives $\partial \mathcal{L}/\partial w$ and $\partial \mathcal{L}/\partial b$ of the cost function $\mathcal{L}$ with respect to any weight w or bias b in the network.
- Remember from Chapter 2, the univariate and multivariate chain rules allow you to find the derivative of a composite function. Let $f(u)$ be a function of u, and $u = g(x)$ be another function of x. The composite function is $h(x) = f(g(x))$. The univariate chain rule can be expressed as:

$$\frac{\partial h(x)}{\partial x} = \frac{\partial f(u)}{\partial u}\frac{\partial u}{\partial x}, \tag{3.58}$$

or

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)}\frac{\partial g(x)}{\partial x}. \tag{3.59}$$

This rule essentially states that the derivative of the composite function $h(x)$ with respect to x is the product of the derivative of f with respect to its immediate variable u, and the derivative of $g(x)$ with respect to x. The derivative $df(u)/du$ represents the local gradient of f with respect to its immediate argument u, and du/dx represents the local gradient of g with respect to its immediate argument x. In the NNs, this univariate chain rule is applied iteratively during the backward pass in BP. Each layer of the network corresponds to a function, and the chain rule is used to compute the gradients with respect to the parameters of each layer by combining local gradients at each step. The multivariate chain rule is defined as follows:

$$\frac{\partial f(g_1(x), \ldots, g_k(x))}{\partial x} = \sum_{i=1}^{k} \frac{\partial f(g_1(x), \ldots, g_k(x))}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x}. \tag{3.60}$$

In the training of a NN, an epoch (or round) refers to one complete pass through the entire training dataset. During each epoch, the NN processes every training example in the dataset, calculates the loss, performs BP to compute gradients. For each epoch, the parameters are updated based on the accumulated gradients of the loss with respect to the weights, see Example 3.1 for clarification.

The number of epochs is a hyperparameter that determines how many times the learning algorithm will work through the entire training dataset. Choosing the right number of epochs is important. Too few epochs may result in the model not capturing the underlying patterns in the data, while too many epochs may lead to overfitting, where the model learns the training data too well but fails to generalize to new, unseen data.

By learning the optimal parameters, NNs can capture complex patterns and relationships in data, allowing them to generalize well to unseen examples. The joint optimization of parameters enables NNs to create highly nonlinear and expressive compositions of simple functions, making them effective for a wide range of tasks, such as image recognition, natural language processing, and more. This is what makes NNs more powerful than their individual building blocks or basic parametric models.

Example 3.1

Let us define a simple NN, one that accepts two input values, has two nodes in its hidden layer, and has a single output node, as shown in Figure 3.15. We will use Sigmoid AFs $\sigma(z) = \frac{1}{1+e^{-z}}$ in the hidden layer. Notice that output has identity AF $\sigma(z) = z$. To train the network, we will use a squared-error loss function, $\mathcal{L}_{\text{MSE}} = \frac{1}{2}\left(y - a_1^{(2)}\right)^2$, where y is the actual value and $a_1^{(2)}$ is the predicted value of the network for the input associated with y, namely $a_1^{(0)}$ and $a_2^{(0)}$.

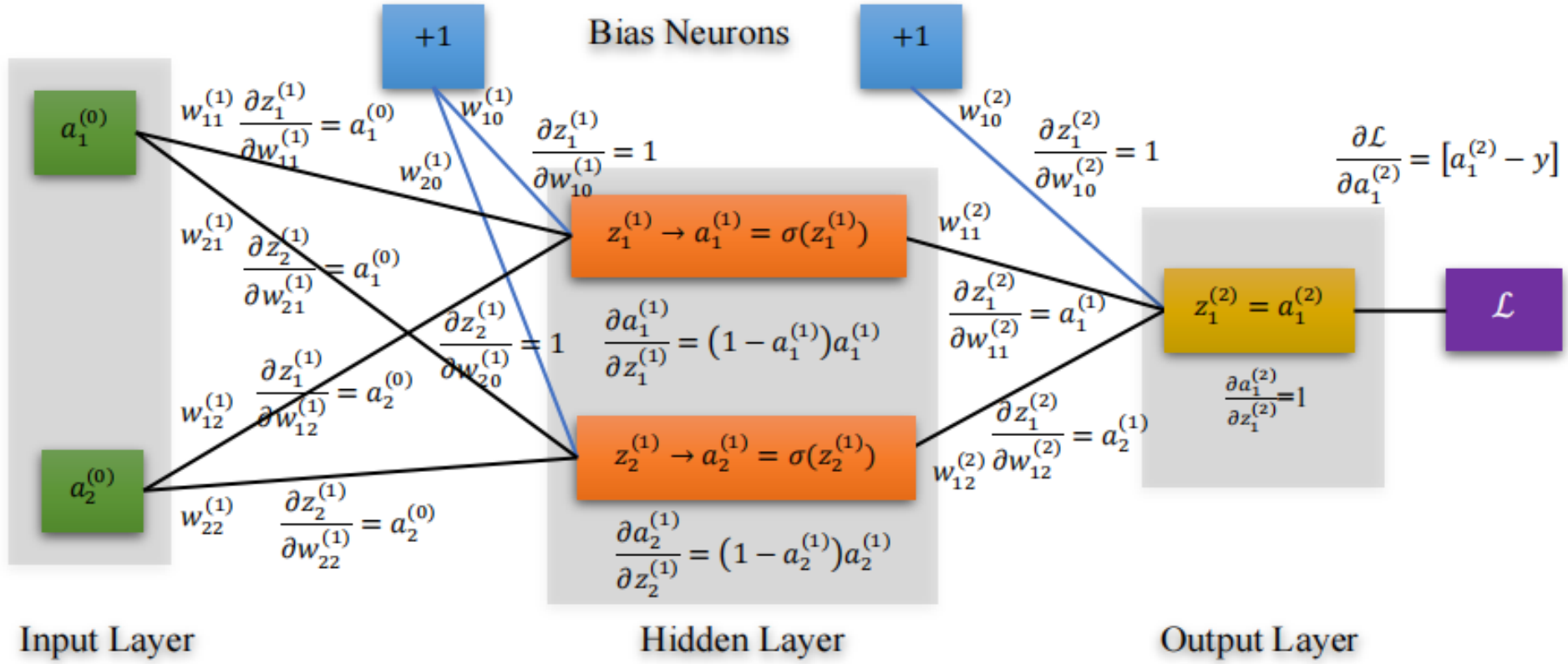

Figure 3.15. A simple NN.

Solution

The forward pass involves calculating the activations at each layer. The equations for the forward pass are as follows:

$$
\begin{aligned}
z_1^{(1)} &= w_{11}^{(1)} a_1^{(0)} + w_{12}^{(1)} a_2^{(0)} + w_{10}^{(1)}, \\
a_1^{(1)} &= \sigma\left(z_1^{(1)}\right) \\
&= \frac{1}{1 + e^{-\left(w_{11}^{(1)} a_1^{(0)} + w_{12}^{(1)} a_2^{(0)} + w_{10}^{(1)}\right)}},
\end{aligned}
$$

$$
\begin{aligned}
z_2^{(1)} &= w_{21}^{(1)} a_1^{(0)} + w_{22}^{(1)} a_2^{(0)} + w_{20}^{(1)}, \\
a_2^{(1)} &= \sigma\left(z_2^{(1)}\right) \\
&= \frac{1}{1 + e^{-\left(w_{21}^{(1)} a_1^{(0)} + w_{22}^{(1)} a_2^{(0)} + w_{20}^{(1)}\right)}},
\end{aligned}
$$

$$
\begin{aligned}
z_1^{(2)} &= w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + w_{10}^{(2)}, \\
a_1^{(2)} &= \sigma\left(z_1^{(2)}\right) \\
&= w_{11}^{(2)} a_1^{(1)} + w_{12}^{(2)} a_2^{(1)} + w_{10}^{(2)}.
\end{aligned}
$$

If the label associated with the training example $\mathbf{a}^{(0)} = \left(a_1^{(0)}, a_2^{(0)}\right)^T$ is y, and the output of the network for this input is $a_1^{(2)}$, then we can define a loss function that measures the difference between the predicted output and the actual label, $\mathcal{L}_{\text{MSE}} = \frac{1}{2}\left(y - a_1^{(2)}\right)^2$. The argument to the loss function is $a_1^{(2)}$; y is a fixed constant.

The loss function can be considered a function of the weights and biases, denoted as $\boldsymbol{\theta}$ (which includes $w_{11}^{(1)}$, $w_{12}^{(1)}$, $w_{21}^{(1)}$, $w_{22}^{(1)}$, $w_{11}^{(2)}$, $w_{12}^{(2)}$, $w_{10}^{(1)}$, $w_{20}^{(1)}$ and $w_{10}^{(2)}$), along with the constant input vector $\mathbf{a}^{(0)} = \left(a_1^{(0)}, a_2^{(0)}\right)^T$ and the associated label y. Mathematically, this can be expressed as:

$$\mathcal{L} = \mathcal{L}_{\text{MSE}}\left(\boldsymbol{\theta}; \mathbf{a}^{(0)}, y\right).$$

During the training process, the goal is to minimize this loss function with respect to the weights and biases ($\boldsymbol{\theta}$). This involves the forward pass to compute the predicted output $a_1^{(2)}$ given the current set of weights and biases, followed by the backward pass, BP, to calculate the gradients of the loss with respect to $\boldsymbol{\theta}$. These gradients are then used in an optimization algorithm to update the weights and biases iteratively. To perform gradient-based optimization, you need to compute the partial derivatives of the loss function with respect to each weight and bias in the network.

We also need an expression for the derivative of our AF, the Sigmoid. The derivative of the sigmoid is

$$\begin{aligned}
\frac{d}{dz}\sigma(z) &= \frac{d}{dz}\frac{1}{1+e^{-z}} \\
&= \frac{d}{dz}\frac{e^z}{1+e^z} \\
&= \frac{e^z(1+e^z)-e^z e^z}{(1+e^z)^2} \\
&= \frac{e^z\big((1+e^z)-e^z\big)}{(1+e^z)^2} \\
&= \frac{1}{(1+e^z)}\frac{e^z}{(1+e^z)} \\
&= \left(1-\frac{e^z}{(1+e^z)}\right)\frac{e^z}{(1+e^z)} \\
&= \big(1-\sigma(x)\big)\sigma(x).
\end{aligned}$$

The derivative of the Sigmoid function can be written in terms of the Sigmoid itself: This is a convenient property when performing BP in NNs because during the forward pass, you have already calculated $\sigma(z)$ as the activation

of the neuron. So, you can reuse this value during the backward pass to efficiently compute the derivative without recalculating the Sigmoid.

BP involves working backward from the output layer to the input layer, applying the chain rule at each step to compute the partial derivatives. The expression for the partial derivative for the specified parameter $a_1^{(2)}$ is:

$$\frac{\partial \mathcal{L}}{\partial a_1^{(2)}} = \frac{\partial}{\partial a_1^{(2)}}\left(\frac{1}{2}\left(y - a_1^{(2)}\right)^2\right)$$
$$= a_1^{(2)} - y.$$

Recall y is the label for the current training example, and we compute $a_1^{(2)}$ during the forward pass as the output of the network. So, when working through the BP algorithm, you can replace $\frac{\partial \mathcal{L}}{\partial a_1^{(2)}}$ with $\left(a_1^{(2)} - y\right)$ in the relevant expressions, making the calculations more straightforward. This simplification is a common step in the BP process.

Output layer:

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(2)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}}\frac{\partial a_1^{(2)}}{\partial w_{11}^{(2)}} = \left(a_1^{(2)} - y\right)\left(a_1^{(1)}\right),$$
$$\frac{\partial \mathcal{L}}{\partial w_{12}^{(2)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}}\frac{\partial a_1^{(2)}}{\partial w_{12}^{(2)}} = \left(a_1^{(2)} - y\right)\left(a_2^{(1)}\right),$$
$$\frac{\partial \mathcal{L}}{\partial w_{10}^{(2)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}}\frac{\partial a_1^{(2)}}{\partial w_{10}^{(2)}} = \left(a_1^{(2)} - y\right)(1).$$

First neuron in hidden layer:

$$\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}}\frac{\partial a_1^{(2)}}{\partial a_1^{(1)}}\frac{\partial a_1^{(1)}}{\partial z_1^{(1)}}\frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} = \left(a_1^{(2)} - y\right)\left(w_{11}^{(2)}\right)\left(\left(1 - a_1^{(1)}\right)a_1^{(1)}\right)\left(a_1^{(0)}\right),$$
$$\frac{\partial \mathcal{L}}{\partial w_{12}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}}\frac{\partial a_1^{(2)}}{\partial a_1^{(1)}}\frac{\partial a_1^{(1)}}{\partial z_1^{(1)}}\frac{\partial z_1^{(1)}}{\partial w_{12}^{(1)}} = \left(a_1^{(2)} - y\right)\left(w_{11}^{(2)}\right)\left(\left(1 - a_1^{(1)}\right)a_1^{(1)}\right)\left(a_2^{(0)}\right),$$

$$\frac{\partial \mathcal{L}}{\partial w_{10}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{10}^{(1)}} = \left(a_1^{(2)} - y\right)\left(w_{11}^{(2)}\right)\left(\left(1 - a_1^{(1)}\right)a_1^{(1)}\right)(1).$$

Second neuron in hidden layer:

$$\frac{\partial \mathcal{L}}{\partial w_{21}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{21}^{(1)}} = \left(a_1^{(2)} - y\right)\left(w_{12}^{(2)}\right)\left(\left(1 - a_2^{(1)}\right)a_2^{(1)}\right)\left(a_1^{(0)}\right),$$

$$\frac{\partial \mathcal{L}}{\partial w_{22}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{22}^{(1)}} = \left(a_1^{(2)} - y\right)\left(w_{12}^{(2)}\right)\left(\left(1 - a_2^{(1)}\right)a_2^{(1)}\right)\left(a_2^{(0)}\right),$$

$$\frac{\partial \mathcal{L}}{\partial w_{20}^{(1)}} = \frac{\partial \mathcal{L}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{20}^{(1)}} = \left(a_1^{(2)} - y\right)\left(w_{12}^{(2)}\right)\left(\left(1 - a_2^{(1)}\right)a_2^{(1)}\right)(1),$$

where we use $\frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} = \left(1 - \sigma\left(z_1^{(1)}\right)\right)\sigma\left(z_1^{(1)}\right) = \left(1 - a_1^{(1)}\right)a_1^{(1)}$ and $\frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} = \left(1 - a_2^{(1)}\right)a_2^{(1)}$.

In the context of updating the bias term ($w_{10}^{(2)}$), the update rule for each epoch can be written as follows:

$$w_{10}^{(2)} = w_{10}^{(2)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left.\frac{\partial \mathcal{L}}{\partial w_{10}^{(2)}}\right|_{\mathbf{a}_i^{(0)}}.$$

Here: $w_{10}^{(2)}$ is the current value of the bias term. α is the learning rate. m is the number of samples in the training set. $\mathcal{L}$ is the loss function. $\left.\frac{\partial \mathcal{L}}{\partial w_{10}^{(2)}}\right|_{\mathbf{a}_i^{(0)}}$ is the partial derivative of the loss with respect to $w_{10}^{(2)}$ evaluated for the i-th

training sample. The summation term is the accumulation of the partial derivatives of the loss with respect to $w_{10}^{(2)}$ over all training samples. This accumulation represents the overall contribution of the bias term to the loss across the entire training set. The division by m is to take the average contribution over all samples, and the entire term $\frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}}{\partial w_{10}^{(2)}}\Big|_{\mathbf{a}_i^{(0)}}$ is the amount by which the bias term is adjusted during the update step. This process is repeated for each parameter in the NN (weights and biases) during each epoch to gradually improve the model's performance. The learning rate (α) controls the size of the step taken in the direction of the steepest decrease in the loss landscape.

Output layer:

$$w_{11}^{(2)} = w_{11}^{(2)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}}{\partial w_{11}^{(2)}}\Big|_{\mathbf{a}_i^{(0)}} = w_{11}^{(2)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\left((a_1^{(2)} - y)(a_1^{(1)})\right)\Big|_{\mathbf{a}_i^{(0)}},$$

$$w_{12}^{(2)} = w_{12}^{(2)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}}{\partial w_{12}^{(2)}}\Big|_{\mathbf{a}_i^{(0)}} = w_{12}^{(2)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\left((a_1^{(2)} - y)(a_2^{(1)})\right)\Big|_{\mathbf{a}_i^{(0)}},$$

$$w_{10}^{(2)} = w_{10}^{(2)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}}{\partial w_{10}^{(2)}}\Big|_{\mathbf{a}_i^{(0)}} = w_{10}^{(2)} - \alpha\frac{1}{m}\sum_{i=1}^{m}(a_1^{(2)} - y)\Big|_{\mathbf{a}_i^{(0)}}.$$

First neuron in hidden layer:

$$w_{11}^{(1)} = w_{11}^{(1)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}}{\partial w_{11}^{(1)}}\Big|_{\mathbf{a}_i^{(0)}} = w_{11}^{(1)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\left((a_1^{(2)} - y)(w_{11}^{(2)})\left((1 - a_1^{(1)})a_1^{(1)}\right)(a_1^{(0)})\right)\Big|_{\mathbf{a}_i^{(0)}},$$

$$w_{12}^{(1)} = w_{12}^{(1)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}}{\partial w_{12}^{(1)}}\Big|_{\mathbf{a}_i^{(0)}} = w_{12}^{(1)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\left((a_1^{(2)} - y)(w_{11}^{(2)})\left((1 - a_1^{(1)})a_1^{(1)}\right)(a_2^{(0)})\right)\Big|_{\mathbf{a}_i^{(0)}},$$

$$w_{10}^{(1)} = w_{10}^{(1)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}}{\partial w_{10}^{(1)}}\Big|_{\mathbf{a}_i^{(0)}} = w_{10}^{(1)} - \alpha\frac{1}{m}\sum_{i=1}^{m}\left((a_1^{(2)} - y)(w_{11}^{(2)})\left((1 - a_1^{(1)})a_1^{(1)}\right)\right)\Big|_{\mathbf{a}_i^{(0)}}.$$

Second neuron in hidden layer:

$$w_{21}^{(1)} = w_{21}^{(1)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left. \frac{\partial \mathcal{L}}{\partial w_{21}^{(1)}} \right|_{\mathbf{a}_i^{(0)}} = w_{21}^{(1)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left. \left((a_1^{(2)} - y)(w_{12}^{(2)}) \left((1 - a_2^{(1)}) a_2^{(1)} \right) (a_1^{(0)}) \right) \right|_{\mathbf{a}_i^{(0)}},$$

$$w_{22}^{(1)} = w_{22}^{(1)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left. \frac{\partial \mathcal{L}}{\partial w_{22}^{(1)}} \right|_{\mathbf{a}_i^{(0)}} = w_{22}^{(1)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left. \left((a_1^{(2)} - y)(w_{12}^{(2)}) \left((1 - a_2^{(1)}) a_2^{(1)} \right) (a_2^{(0)}) \right) \right|_{\mathbf{a}_i^{(0)}},$$

$$w_{20}^{(1)} = w_{20}^{(1)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left. \frac{\partial \mathcal{L}}{\partial w_{20}^{(1)}} \right|_{\mathbf{a}_i^{(0)}} = w_{20}^{(1)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \left. \left((a_1^{(2)} - y)(w_{12}^{(2)}) \left((1 - a_2^{(1)}) a_2^{(1)} \right) \right) \right|_{\mathbf{a}_i^{(0)}}.$$

3.6 The Four Fundamental Equations Behind Backpropagation

The key idea behind BP is to update the weights and biases of the network in the direction that reduces the error or cost function. To achieve this, the algorithm computes the partial derivatives of the cost function with respect to the weights ($\partial \mathcal{L}/\partial w_{jk}^{(l)}$) and biases ($\partial \mathcal{L}/\partial w_{j0}^{(l)}$) in each layer of the NN. The derivatives $\partial \mathcal{L}/\partial w_{jk}^{(l)}$ and $\partial \mathcal{L}/\partial w_{j0}^{(l)}$, represent how much the cost function changes with respect to the weights and biases, respectively.

BP is indeed based on four fundamental equations that help in computing the error and the gradient of the cost function. These equations are derived using the chain rule and are applied during the backward pass of the training process. The auxiliary variable $\boldsymbol{\delta}^{(l)}$ is commonly used in BP to represent the derivative of the loss ($\mathcal{L}$) with respect to the weighted sum of inputs $\mathbf{z}^{(l)}$ for layer l, i.e., the auxiliary variable $\boldsymbol{\delta}^{(l)}$ represents the rate of change of the loss with respect to the pre-activation output of layer l. Here, l can take values from 0 to the final layer L. In mathematical terms:

$$\boldsymbol{\delta}^{(l)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}}. \tag{3.61}$$

Let us revisit and update our notations, opting to reintegrate $\mathbf{b}^{(l)}$ as the biases for layer l, $\mathbf{z}^{(l)}$ as the weighted input to layer l, $\mathbf{a}^{(l)}$ as the output of layer l after applying the AF, $\mathcal{L}$ as the cost function, $\mathbf{W}^{(l)}$ as the weights for layer l, as commonly done in most literature. Our strategy involves initially presenting the equations with a straightforward example to demonstrate their application in a spatial case, Example 3.2. Following this, we will provide a comprehensive proof for the general case. The four fundamental equations are as follows:

1. Error in the Output Layer $\boldsymbol{\delta}^{(L)}$:

The components of $\delta^{(L)}$ are given by

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial a_j^{(L)}} \sigma'\left(z_j^{(L)}\right). \tag{3.62}$$

The first term on the right, $\partial \mathcal{L}/\partial a_j^{(L)}$, just measures how fast the cost is changing as a function of the j-th output activation. The second term on the right, $\sigma'\left(z_j^{(L)}\right)$, measures how fast the AF σ is changing at $z_j^{(L)}$. Notice that everything in (3.62) is easily computed. In particular, we compute $z_j^{(L)}$ while computing the behaviour of the network, and it is easy to compute $\sigma'\left(z_j^{(L)}\right)$. The exact form of $\partial \mathcal{L}/\partial a_j^{(L)}$ will, of course, depend on the form of the cost function. However, provided the cost function is known there should be little trouble computing $\partial \mathcal{L}/\partial a_i^{(L)}$. For example, if

3.7.1 Full-Batch Gradient Descent (Gradient Descent)

How can we apply Full-Batch Gradient Descent (FBGD) (traditional GD) to learn in a NN? The idea is to use GD to find the weights $\mathbf{W}^{(l)}$ and biases $\mathbf{b}^{(l)}$ which minimize the cost function. In GD, one tries to minimize the cost function of the NN by moving the parameters along the negative direction of the gradient of the loss over all points, because this is the direction of the steepest descent. Most machine learning problems can be recast as optimization problems over a linearly additive sum of the loss functions on the individual training data points. However, the loss over the entire data set is really defined as the sum (average) of the losses over individual training points. One can write the loss function of a NN as the sum of point-specific losses:

$$\mathcal{L} = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}_{\mathbf{x}_i}, \tag{3.87}$$

where m is the number of all training inputs $\{\mathbf{x}_i\}$, $i \in 1,..m$, and $\mathcal{L}_{\mathbf{x}_i}$ is the loss contributed by the ith training point. To understand what the problem is, let's look back at the quadratic cost function. Notice that this cost function is an average over costs $\mathcal{L}_{\mathbf{x}_i} \equiv \frac{(y_i - a^{(L)})^2}{2}$ for individual training examples. In practice, to compute the gradient $\nabla\mathcal{L}$ we need to compute the gradients $\nabla\mathcal{L}_{\mathbf{x}_i}$ separately for each training input, i, and then average them,

$$\nabla\mathcal{L} = \frac{1}{m}\sum_{i=1}^{m} \nabla\mathcal{L}_{\mathbf{x}_i}. \tag{3.88}$$

Therefore, in traditional GD, one would try to perform GD steps (in vector notation) such as the following:

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \alpha \frac{\partial\mathcal{L}}{\partial\mathbf{W}^{(l)}}, \tag{3.89.1}$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha \frac{\partial\mathcal{L}}{\partial\mathbf{b}^{(l)}}. \tag{3.89.2}$$

Correspondingly, it is easy to show that the true update of the NN should be the following:

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}_{\mathbf{x}_i}}{\partial \mathbf{W}^{(l)}}, \tag{3.90.1}$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}_{\mathbf{x}_i}}{\partial \mathbf{b}^{(l)}}. \tag{3.90.2}$$

By repeatedly applying this update rule we can "roll down the hill", and hopefully find a minimum of the cost function.

In other words, the NN is designed to be a batch algorithm. All of the training examples are presented to the NN, and the average sum of the cost functions of all training examples is then computed, and this is used to update the weights. Thus, there is only one set of weight updates for each epoch (pass through all the training examples). This means that we only update the weights once for each iteration of the algorithm, which means that the weights are moved in the direction that most of the inputs want them to move, rather than being pulled around by each input individually.

The basic intuition behind FBGD can be illustrated by a hypothetical scenario. A person is stuck in the mountains and is trying to get down (i.e., trying to find the global minimum). There is heavy fog such that visibility is extremely low. Therefore, the path down the mountain is not visible, so he must use local information to find the minimum. He can use the method of FBGD, which involves looking at the steepness of the hill at its current position, and then proceeding in the direction with the steepest descent (i.e., downhill).

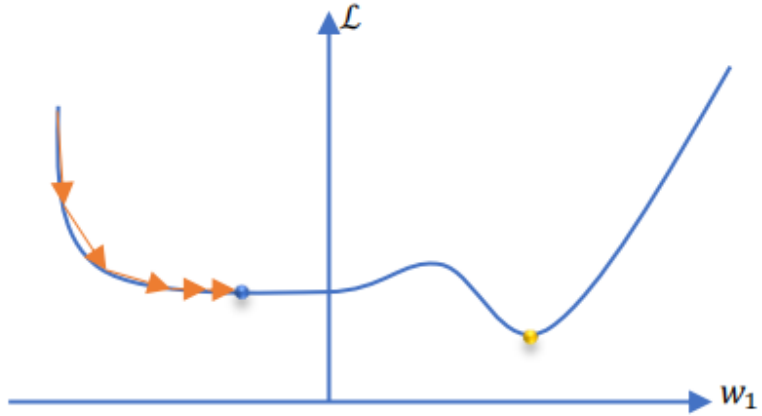

Figure 3.17. FBGD is sensitive to saddle points which can lead to premature convergence.

- Envision the mountain landscape as the cost function surface, where the goal is to find the lowest point, representing the minimum of the cost function. The mountains symbolize the high-dimensional space of possible parameter values in a NN model. Each point in this space corresponds to a set of parameters that define the model.
- The person in the mountains represents the optimization algorithm, specifically FBGD in this context.
- The low visibility indicates that the person (algorithm) can't see the entire cost function surface. Instead, he relies on local information obtained from a subset of the landscape. In machine learning, this local information corresponds to the gradient of the cost function at the current parameter values. The gradient provides information about the slope or steepness of the terrain at the current location.
- In FBGD, the person doesn't just evaluate the steepness of the hill at one point but considers the entire landscape. This corresponds to using the entire dataset (a batch of data) to compute the gradient.
- The person assesses the steepest descent by considering the average steepness of the entire landscape in the batch. In FBGD, the algorithm calculates the average gradient of the cost function with respect to the model parameters using all data points in the batch.
- The person then moves downhill based on the average gradient, aiming to reach lower elevations. In FBGD, the algorithm updates the model parameters in the direction opposite to the average gradient, seeking to decrease the average value of the cost function over the entire batch.

- FBGD continues this process iteratively, considering the entire dataset in each iteration.
- Through iterations, FBGD aims to converge to the global minimum of the cost function, taking advantage of the comprehensive information provided by the entire dataset in each step.

Unfortunately, when the number of training inputs is very large this can take a long time, and learning thus occurs slowly. Note that the intermediate activations and derivatives for each training instance would need to be maintained simultaneously over hundreds of thousands of NN nodes. This can be exceedingly large in most practical settings. It is, therefore, impractical to simultaneously run all examples through the network to compute the gradient with respect to the entire data set in one shot. As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long. If we are using FBGD (which looks at all the data), we take steps leading us in the correct direction. But each step is expensive, so we can only take a few steps.

Moreover, for a simple quadratic error surface (cost function surface), FBGD works quite well. But in most cases, our error surface may be a lot more complicated. Let us consider the scenario in Figure 3.17. We have only a single weight, and we use random initialization and FBGD to find its optimal setting. The error surface, however, has a flat region (also known as saddle point in high-dimensional spaces), and if we get unlucky, we might find ourselves getting stuck while performing FBGD.

Remarks:

- Since FBGD uses the entire dataset to compute the gradient, it requires storing the entire dataset in memory. This might be a limitation for very large datasets.
- FBGD assumes that the cost function is smooth and continuous. If the cost function has many local minima, FBGD might get stuck in a suboptimal solution.

- FBGD continues iterating until certain convergence criteria are met. This could be a predefined number of iterations or a threshold for the change in the cost function. Commonly, practitioners monitor the cost function and stop training when the change is smaller than a predefined tolerance.
- The key advantage of FBGD is that it utilizes the entire dataset to compute the gradient in each iteration, which leads to a more stable convergence.
- One drawback is that it may be slow when dealing with large datasets or when the model is complex. To address this, variations such as stochastic gradient descent and mini-batch stochastic gradient descent are often used, where the gradient is computed using only a subset of the data in each iteration. These methods provide a compromise between the stability of FBGD and the computational efficiency of GD with respect to individual data points.

3.7.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an extension of the GD algorithm, sometimes also called iterative or on-line GD. A recurring problem in machine learning is that large training sets are necessary for good generalization, but large training sets are also more computationally expensive. One rarely uses all the points at a single time, and one might stochastically select a point in order to update the parameters to reduce that point-specific loss. In a NN, this process is natural because the simple methods we have introduced so far process the points one at a time in forwards and backwards propagation in order to iteratively minimize point-specific losses. In SGD, all updates to the weights of a NN are performed in point-specific fashion. The point-at-a-time update introduced so far is really a practical approximation of the true update by updating with the use of $\mathcal{L}_{\mathbf{x}_j}$ rather than $\mathcal{L}$:

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \alpha \frac{\partial \mathcal{L}_{\mathbf{x}_i}}{\partial \mathbf{W}^{(l)}}, \tag{3.91.1}$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha \frac{\partial \mathcal{L}_{\mathbf{x}_i}}{\partial \mathbf{b}^{(l)}}. \tag{3.91.2}$$

Assuming a random ordering of points, each update can be viewed as a probabilistic approximation of the true update. The main advantages of SGD are that it is fast and memory efficient, albeit at the expense of accuracy. The main issue with SGD is that the point-at-a-time approach can sometimes behave in an unstable way, because individual points in the training data might be mislabeled or have other errors.

Remarks:

- The SGD algorithm is the sequential algorithm, where the errors are computed and the weights updated after each input. This is not guaranteed to be as efficient in learning, but it is simpler to program when using loops, and it is therefore much more common.
- When we use SGD, we perform m updates per epoch, so we get more updates or steps for a fixed number of epochs. But because of the stochastic or random behavior of using just one data point for each update, the steps we take are noisy. They don't always head in the correct direction. But the larger total number of steps eventually gets us closer to the answer.
- Since each gradient is calculated based on a single training example, the error surface is noisier than in FBGD, which can also have the advantage that SGD can escape shallow local minima more readily. The SGD approach is illustrated by Figure 3.18, where instead of a single static error surface, our error surface is dynamic. As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions.
- In the SGD algorithm, the order of the weight updates can matter, which is why the algorithm includes a suggestion about randomizing the order of the input vectors at each iteration (which is why we want to shuffle the training set for every epoch to prevent cycles). This can significantly improve the speed with which the algorithm learns.
- In SGD implementations, the fixed learning rate α is often replaced by an adaptive learning rate that decreases over time.

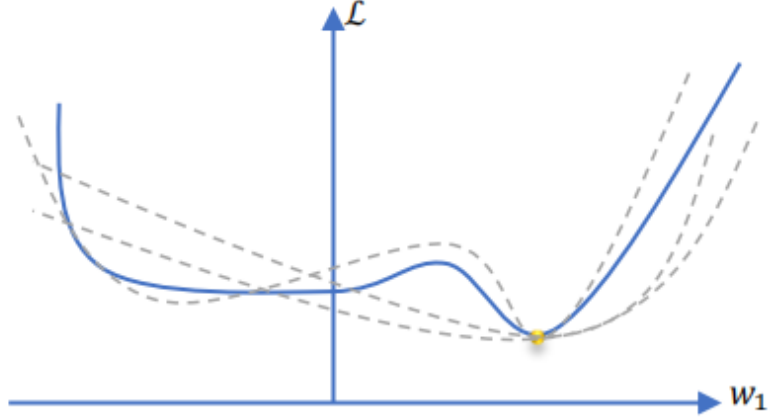

Figure 3.18. The SGD error surface fluctuates with respect to the batch error surface, enabling saddle point avoidance.

- Another advantage of SGD is that we can use it for online learning. In online learning, our model is trained on-the-fly as new training data arrives. This is especially useful if we are accumulating large amounts of data—for example, customer data in typical web applications. Using online learning, the system can immediately adapt to changes and the training data can be discarded after updating the model if storage space in an issue.
- The frequent updates allow an easy check on how the model learning is going. (You don't have to wait until all the datasets have been considered.)

The basic intuition behind SGD can be illustrated by a hypothetical scenario.

- Imagine our person in the mountains facing unpredictable weather changes. Sometimes the fog lifts momentarily, allowing for clearer visibility, but it can also return suddenly. This uncertainty corresponds to the stochastic nature of SGD.
- In SGD, visibility is limited even more. The person now relies on brief moments of clarity (randomly selected individual data points) to gather information about the steepness of the terrain.
- The information obtained from a single data point is noisier compared to the average gradient computed over the entire batch. It might provide a good estimate or introduce some randomness.

- When visibility improves, the person quickly assesses the steepness and takes a step downhill. However, this step is now based on information from only one data point.
- SGD updates model parameters more frequently but with a higher level of uncertainty compared to FBGD.
- Due to the stochastic nature, the person takes rapid, less precise steps downhill. This enables the algorithm to explore the landscape more dynamically and escape potential local minima.
- SGD might not consistently move directly toward the global minimum because of the noisy information obtained from individual data points. Instead, it meanders around, allowing for exploration.
- The person adapts to the changing weather conditions by adjusting the step size and direction more frequently. Similarly, SGD adjusts its learning rate dynamically based on the noisiness of individual data points.
- SGD trades off some accuracy for increased efficiency. It may not precisely follow the steepest descent, but the frequent, quick updates make it computationally less expensive, especially with large datasets.

3.7.3 Mini-Batch Stochastic Gradient Descent

The insight of mini-batch stochastic gradient descent (MBSGD) is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples. The MBSGD can be used to speed up learning. The idea is to estimate the gradient $\nabla\mathcal{L}$ by computing $\mathcal{L}_{\mathbf{x}_j}$ for a small sample of randomly chosen training inputs. By averaging over this small sample, it turns out that we can quickly get a good estimate of the true gradient $\nabla\mathcal{L}$, and this helps speed up gradient descent, and thus learning.

Specifically, on each step of the algorithm, we can sample a minibatch of examples $B = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{m'}\}$ drawn uniformly from the training set. The minibatch size m' is typically chosen to be a relatively small number of examples,

ranging from one to a few hundred. Crucially, m' is usually held fixed as the training set size m grows. We may fit a training set with billions of examples using updates computed on only a hundred examples. Provided the sample size m' is large enough we expect that the average value of the $\nabla\mathcal{L}_{\mathbf{x}_j}$ will be roughly equal to the average over all $\nabla\mathcal{L}$, that is,

$$\frac{1}{m'}\sum_{j=1}^{m'}\nabla\mathcal{L}_{\mathbf{x}_j} \approx \frac{1}{m}\sum_{i=1}^{m}\nabla\mathcal{L}_i = \nabla\mathcal{L}, \tag{3.92}$$

where the second sum is over the entire set of training data. Swapping sides we get

$$\nabla\mathcal{L} \approx \frac{1}{m'}\sum_{j=1}^{m'}\nabla\mathcal{L}_{\mathbf{x}_j}, \tag{3.93}$$

confirming that we can estimate the overall gradient by computing gradients just for the randomly chosen mini-batch.

To connect this explicitly to learning in NNs, suppose $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ denote the weights and biases in our NN. Then MBSGD works by picking out a randomly chosen mini-batch of training inputs, and training with those,

$$\mathbf{W}^{(l)} = \mathbf{W}^{(l)} - \alpha\frac{1}{m'}\sum_{j=1}^{m'}\frac{\partial\mathcal{L}_{\mathbf{x}_j}}{\partial\mathbf{W}^{(l)}}, \tag{3.94.1}$$

$$\mathbf{b}^{(l)} = \mathbf{b}^{(l)} - \alpha\frac{1}{m'}\sum_{j=1}^{m'}\frac{\partial\mathcal{L}_{\mathbf{x}_j}}{\partial\mathbf{b}^{(l)}}, \tag{3.94.2}$$

where the sums are over all the training examples $\mathbf{x}_j$ in the current mini-batch. Then we pick out another randomly chosen mini-batch and train with those. And so on, until we have exhausted the training inputs, which is said to complete an epoch of training. At that point, we start over with a new training epoch.

Remarks:

- The idea of a MBSGD is to find some happy middle ground between the FBGD and SGD, by splitting the training set into random batches, estimating the gradient based on one of the subsets of the training set, performing a weight update, and then using the next subset to estimate a new gradient and using that for the weight update, until all of the training set have been used. The training set is then randomly shuffled into new batches and the next iteration takes place. If the batches are small, then there is often a reasonable degree of error in the gradient estimate, and so the optimization has the chance to escape from local minima, albeit at the cost of heading in the wrong direction.
- Incidentally, it's worth noting that conventions vary about the scaling of the cost function and of mini-batch updates to the weights and biases. In (3.87) we scaled the overall cost function by a factor $1/m$. People sometimes omit the $1/m$, summing over the costs of individual training examples instead of averaging. This is particularly useful when the total number of training examples isn't known in advance. This can occur if more training data is being generated in real-time, for instance. And, similarly, the mini-batch update rules (3.94.1) and (3.94.2) sometimes omit the $1/m'$ term out the front of the sums. Conceptually this makes little difference since it's equivalent to rescaling the learning rate α. But when doing detailed comparisons of different work it's worth watching out for.
- We can think of MBSGD as being like political polling: it's much easier to sample a small mini-batch than it is to apply GD to the full batch, just as carrying out a poll is easier than running a full election. For example, if we have a training set of size $n = 60{,}000$, and choose a mini-batch size of (say) $m' = 10$, this means we'll get a factor of 6,000 speedup in estimating the gradient! Of course, the estimate won't be perfect – there will be statistical fluctuations – but it doesn't need to be perfect: all we really care about is moving in a general direction that will help decrease $\mathcal{L}$, and that means we don't need an exact computation of the gradient. In practice, MBSGD is a commonly used and powerful technique for learning in NNs.

- The data used in machine learning problems often have a high level of redundancy in terms of the knowledge captured by different training points, and therefore the gradient obtained from a sample of points is usually quite accurate. Furthermore, the weights are often incorrect to such a degree at the beginning of the learning process that even a small sample of points can be used to create an excellent estimate of the gradient. This observation provides a practical foundation for the success of MBSGD, which often exhibits the best trade-off between stability, speed, and memory requirements.
- From an implementation perspective, a mini-batch can be represented by a matrix because each individual training example is an array of inputs and an array of arrays becomes a matrix. Similarly, the weights for a single neuron can be arranged as an array, and we can arrange the weights for all neurons in a layer as a matrix. Computing the inputs to all activation functions for all neurons in the layer for all input examples in the mini-batch is then reduced to a single matrix-matrix multiplication. In other words, when using MBSGD, the outputs of each layer are matrices instead of vectors, and forward propagation requires the multiplication of the weight matrix with the activation matrix (3.84). The same is true for backward propagation in which matrices of gradients are maintained. Hence, mini-batch learning allows us to replace the for-loop over the training samples in SGD by vectorized operations (specifically matrix–matrix multiplications), which can further improve the computational efficiency of our learning algorithm. In this scenario, we also have the option of sending the vectorized computations to GPUs if they are present. The GPUs do a good job of computing a mini-batch in parallel.
- Note that, the SGD is a more extreme version of the MBSGD. The idea is to use just one piece of data to estimate the gradient, and to pick that piece of data uniformly at random from the training set. So, a single input vector is chosen from the training set, and the output and hence the error for that one vector computed, and this is used to estimate the gradient and so update the weights. A new random input vector (which could be the same as the previous one) is then chosen and the process is repeated.
- The advantage over FBGD is that convergence is reached faster via mini-batches because of the more frequent weight updates. Applying MBSGD has also been shown to lead to smoother convergence because the gradient is computed at each step it uses more training examples to compute the gradient.

- As the mini-batch size increases the gradient computed is closer to the "true" gradient of the entire training set. This also gives us the advantage of better computational efficiency. Ideally, each mini-batch trained on should contain an example of each class to reduce the sampling error when estimating the gradient for the entire training set.
- The use of MBSGD introduces a new hyperparameter that must be tuned: the batch size (number of observations in the mini-batch).
- The relationship between how fast our algorithm can learn the model is typically U-shaped (batch size versus training speed). This means that initially as the batch size becomes larger, the training time will decrease. Eventually, we'll see the training time begin to increase when we exceed a certain batch size that is too large.
- The model update frequency is higher than with FBGD but lower than SGD. Therefore, allows for a more robust convergence.
- Finally, don't forget that the number of parameter updates per epoch is reduced when we increase the mini-batch size (where an epoch refers to one full pass of the training data). The number of parameter updates per epoch is just the total number of examples in our training set divided by the mini-batch size. For example, if 20 arbitrary data points are selected out of 100 training data points, and the MBSGD is applied to the training data, in this case, a total of five weight adjustments are performed to complete the training process for all the data points ($5 = 100/20$). Figure 3.19 shows how the mini-batch scheme selects training data and calculates the weight update. In the FBGD, the number of training cycles of the NN equals an epoch, as shown in Figure 3.19. This makes perfect sense because the batch method utilizes all of the data for one training process. When we have m training data points in total, the number of training processes per epoch is greater than one, which corresponds to the MBSGD, and equal to m, which corresponds to the SGD.
- The MBSGD, when it selects an appropriate number of data points, obtains the benefits from both methods: speed and the local-minima avoidance afforded by the SGD and stability from the FBGD. For this reason, it is often utilized in deep learning, which manipulates a significant amount of data.

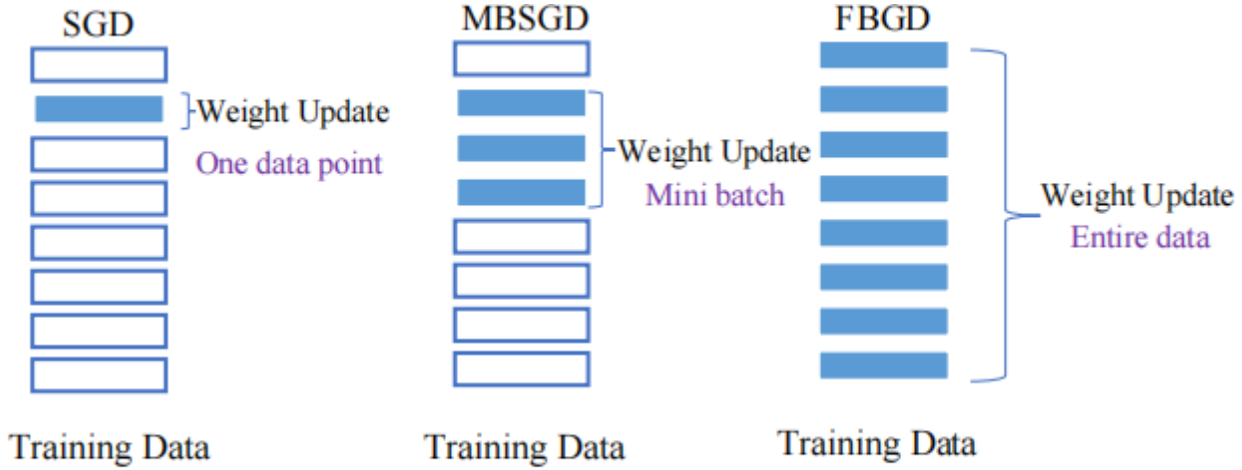

Figure 3.19. Comparison of SGD, MBSGD, and FBGD algorithms based on batch size.

In principle (with appropriate tuning), a NN can learn with any minibatch size. In practice, we need to choose a mini-batch size that balances the following: Memory requirements, computational efficiency, and optimization efficiency. Regarding computational efficiency, modern deep learning libraries parallelize learning at the level of mathematical operations such as matrix multiplications and vector operations (additions, element-wise multiplications, etc.). This means that too small a mini-batch size results in poor hardware utilization (especially on GPUs), and too large a mini-batch size can be inefficient—again, we average gradients over all examples in the mini-batch. For performance (this is most important in the case of GPUs), we should use a multiple of 32 for the batch size, or multiples of 16, 8, 4, or 2 if multiples of 32 can't be used. The reason for this is simple: memory access and hardware design are better optimized for operating on arrays with dimensions that are powers of two, compared to other sizes. We should also consider the powers of two when setting our layer sizes. For example, we should use a layer size of 128 over size 125, or size 256 over size 250, and so on. Regarding optimization efficiency, it's sufficient to note that we cannot choose the mini-batch size totally in isolation from the other hyperparameters such as learning rate—a larger mini-batch size means smoother gradients (i.e., more accurate/consistent gradients), which, in conjunction with appropriate tuning, allow for faster learning for a given number of parameter updates. The trade-off of course is that each parameter update will take longer to compute. Using a larger mini-batch size might help our network to learn in some difficult cases, such as for noisy or imbalanced datasets.

In summary, given the dataset contains 1280 points and each mini-batch contains 128 points, we can calculate the number of mini-batches needed to cover the entire dataset: Number of mini-batches = Total number of data points / Mini-batch size= 1280 /128= 10. So, in each epoch of training, there are 10 mini-batches, with each mini-batch containing 128 data points. This setup ensures that every data point is seen exactly once in each epoch.

- The training data is shuffled to introduce randomness, and then divided into mini-batches, each containing 128 data points. This step ensures that the model sees different subsets of data in each epoch, aiding generalization. The sampling is done without replacement, meaning that at the end of an epoch each point data has been seen by the algorithm only once.
- Each mini-batch is passed through the network using forward propagation. The input data is processed layer by layer until the output layer produces predicted values, denoted as $\hat{y}$.
- The predicted values ($\hat{y}$) are compared against the true labels (y) using a cost function. This function evaluates how well the model is performing on the current mini-batch.
- With the cost calculated, the model updates its parameters (weights and biases) using GD. The gradients of the parameters with respect to the cost are computed through BP. The adjustments made to the parameters are scaled by the learning rate hyperparameter (α).
- After completing training on all mini-batches once (i.e., after 10 cycles in this case), the first epoch of training concludes. The training process then proceeds to the next epoch, where the entire dataset is replenished, shuffled, and divided into mini-batches again.
- Training continues for multiple epochs until the desired number is reached. Each epoch involves iterating through the entire dataset with multiple mini-batches and updating the model parameters accordingly. This iterative process allows the model to gradually improve its performance over successive epochs.

3.8 Linear Activation Function

Before concluding this chapter, we must ask an important question: why do NNs work so well? One of the fundamental reasons for the effectiveness of NNs is their ability to introduce non-linearity. Non-linearity is essential because it allows NNs to model complex patterns and relationships within data. Without non-linearity, NNs would be limited to modeling linear functions, severely restricting their expressive power. In this section, we will delve into this point in detail. In the next section, we will explore other reasons for the success of NNs.

The linear AF, also known as the identity AF, is one of the simplest AFs used in NNs. It is defined as: $\sigma_{\text{Linear}}(x) = c\,x$, for $c = 1$, i.e., identity function. The function is infinitely smooth, but all derivatives beyond the second derivative are zero. The range of the function is $[-\infty, \infty]$.

At its most basic level, a NN is a computational graph that performs compositions of simpler functions to provide a more complex function. Much of the power of deep learning arises from the fact that the repeated composition of functions has significant expressive power. However, not all base functions are equally good at achieving this goal. In fact, the nonlinear squashing functions used in NNs are not arbitrarily chosen but are carefully designed because of certain types of properties. For example, imagine a situation in which the identity AF is used in each layer, so that only linear functions are computed. In such a case, the resulting NN is no stronger than a single-layer, linear NN.

Theorem 3.1: A multi-layer NN that uses only the identity AF in all its layers reduces to a single-layer NN.

Proof:

Consider a NN containing L hidden layers, and therefore contains a total of $(L + 1)$ computational layers (including the output layer). The corresponding $(L + 1)$ weight matrices between successive layers are denoted by $\mathbf{W}^{(1)} \ldots \mathbf{W}^{(L+1)}$. Let $\mathbf{x}$ be the d-dimensional column vector corresponding to the input, $\mathbf{a}^{(1)} \ldots \mathbf{a}^{(L)}$ be the post-activation column vectors corresponding to the hidden layers, and $\mathbf{0}$ be the m-dimensional column vector corresponding to the output.

Case 1: (without biases)

We have the following recurrence conditions for multi-layer NNs:

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}^{(1)} \cdot \mathbf{x}\right) = \mathbf{W}^{(1)} \cdot \mathbf{x},$$

$$\mathbf{a}^{(p+1)} = \sigma\left(\mathbf{W}^{(p+1)} \cdot \mathbf{a}^{(p)}\right) = \mathbf{W}^{(p+1)} \cdot \mathbf{a}^{(p)} \quad \forall\, p \in \{1 \dots L-1\},$$

$$\mathbf{0} = \sigma\left(\mathbf{W}^{(L+1)} \cdot \mathbf{a}^{(L)}\right) = \mathbf{W}^{(L+1)} \cdot \mathbf{a}^{(L)}.$$

In all the cases above, the AF $\sigma(\cdot)$ has been set to the identity function. Then, by eliminating the hidden layer variables, we obtain the following:

$$\begin{aligned}
\mathbf{0} &= \sigma\left(\mathbf{W}^{(L+1)} \cdot \mathbf{a}^{(L)}\right) \\
&= \mathbf{W}^{(L+1)} \cdot \mathbf{a}^{(L)} \\
&= \mathbf{W}^{(L+1)} \cdot \left(\mathbf{W}^{(L)} \cdot \mathbf{a}^{(L-1)}\right) \\
&= \mathbf{W}^{(L+1)} \cdot \left(\mathbf{W}^{(L)} \cdot \left(\mathbf{W}^{(L-1)} \cdot \mathbf{a}^{(L-2)}\right)\right) \\
&= \mathbf{W}^{(L+1)} \cdot \left(\mathbf{W}^{(L)} \cdot \left(\mathbf{W}^{(L-1)} \cdot \left(\mathbf{W}^{(L-2)} \cdot \mathbf{a}^{(L-3)}\right)\right)\right) \\
&= \mathbf{W}^{(L+1)} \cdot \left(\mathbf{W}^{(L)} \cdot \left(\mathbf{W}^{(L-1)} \cdot \left(\mathbf{W}^{(L-2)} \cdot \left(\dots \left(\mathbf{W}^{(2)} \cdot \left(\mathbf{W}^{(1)} \cdot \mathbf{x}\right)\right)\right)\right)\right)\right) \\
&= \mathbf{W}^{(L+1)} \cdot \mathbf{W}^{(L)} \cdot \mathbf{W}^{(L-1)} \cdot \dots \cdot \mathbf{W}^{(2)} \cdot \mathbf{W}^{(1)} \cdot \mathbf{x} \\
&= \left(\prod_{i=1}^{L+1} \mathbf{W}^{(i)}\right) \cdot \mathbf{x}
\end{aligned}$$

Let

$$\bar{\mathbf{W}} = \prod_{i=1}^{L+1} \mathbf{W}^{(i)}.$$

Finally, we can express this as a single-layer NN:

$$\mathbf{O} = \bar{\mathbf{W}} \cdot \mathbf{x}$$

Case 2: (with biases)

We have the following recurrence conditions for multi-layer NNs:

$$\mathbf{a}^{(1)} = \sigma\left(\mathbf{W}^{(1)} \cdot \mathbf{x} + \mathbf{b}^{(1)}\right) = \mathbf{W}^{(1)} \cdot \mathbf{x} + \mathbf{b}^{(1)},$$

$$\mathbf{a}^{(p+1)} = \sigma\left(\mathbf{W}^{(p+1)} \cdot \mathbf{a}^{(p)} + \mathbf{b}^{(p+1)}\right) = \mathbf{W}^{(p+1)} \cdot \mathbf{a}^{(p)} + \mathbf{b}^{(p+1)} \quad \forall\, p \in \{1 \dots L-1\},$$

$$\mathbf{O} = \sigma\left(\mathbf{W}^{(L+1)} \cdot \mathbf{a}^{(L)} + \mathbf{b}^{(L+1)}\right) = \mathbf{W}^{(L+1)} \cdot \mathbf{a}^{(L)} + \mathbf{b}^{(L+1)}.$$

In all the cases above, the AF $\sigma(\cdot)$ has been set to the identity function. Then, by eliminating the hidden layer variables, we obtain the following:

$$\begin{aligned}
\mathbf{O} &= \sigma\left(\mathbf{W}^{(L+1)} \cdot \mathbf{a}^{(L)} + \mathbf{b}^{(L+1)}\right) \\
&= \mathbf{W}^{(L+1)} \cdot \mathbf{a}^{(L)} + \mathbf{b}^{(L+1)} \\
&= \mathbf{W}^{(L+1)} \cdot \left(\mathbf{W}^{(L)} \cdot \mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}\right) + \mathbf{b}^{(L+1)} \\
&= \mathbf{W}^{(L+1)} \cdot \left(\mathbf{W}^{(L)} \cdot \left(\mathbf{W}^{(L-1)} \cdot \mathbf{a}^{(L-2)} + \mathbf{b}^{(L-1)}\right) + \mathbf{b}^{(L)}\right) + \mathbf{b}^{(L+1)} \\
&= \mathbf{W}^{(L+1)} \cdot \left(\mathbf{W}^{(L)} \cdot \left(\mathbf{W}^{(L-1)} \cdot \left(\mathbf{W}^{(L-2)} \cdot \mathbf{a}^{(L-3)} + \mathbf{b}^{(L-2)}\right) + \mathbf{b}^{(L-1)}\right) + \mathbf{b}^{(L)}\right) + \mathbf{b}^{(L+1)} \\
&= \mathbf{W}^{(L+1)} \cdot \left(\mathbf{W}^{(L)} \cdot \left(\mathbf{W}^{(L-1)} \cdot \left(\mathbf{W}^{(L-2)} \cdot \left(\dots\left(\mathbf{W}^{(2)} \cdot \left(\mathbf{W}^{(1)} \cdot \mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right)\dots\right) + \mathbf{b}^{(L-2)}\right) + \mathbf{b}^{(L-1)}\right) + \mathbf{b}^{(L)}\right) \\
&\qquad + \mathbf{b}^{(L+1)} \\
&= \mathbf{b}^{(L+1)} + \mathbf{W}^{(L+1)} \cdot \mathbf{b}^{(L)} + \mathbf{W}^{(L+1)}\mathbf{W}^{(L)} \cdot \mathbf{b}^{(L-1)} + \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdot \mathbf{b}^{(L-2)} + \dots + \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdot \dots \\
&\qquad \cdot \mathbf{W}^{(2)} \cdot \mathbf{b}^{(1)} + \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdot \dots \cdot \mathbf{W}^{(2)} \cdot \mathbf{W}^{(1)} \cdot \mathbf{x}
\end{aligned}$$

$$= \mathbf{b}^{(L+1)} + \mathbf{W}^{(L+1)} \cdot \mathbf{b}^{(L)} + \mathbf{W}^{(L+1)}\mathbf{W}^{(L)} \cdot \mathbf{b}^{(L-1)} + \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdot \mathbf{b}^{(L-2)} + \cdots + \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdot \ldots$$
$$\cdot \mathbf{W}^{(2)} \cdot \mathbf{b}^{(1)} + \mathbf{W}^{(L+1)}\mathbf{W}^{(L)}\mathbf{W}^{(L-1)} \cdot \ldots \cdot \mathbf{W}^{(2)} \cdot \mathbf{W}^{(1)} \cdot \mathbf{x}$$

$$= \left(\prod_{i=1}^{L+1} \mathbf{W}^{(i)}\right) \cdot \mathbf{x} + \sum_{i=1}^{L} \left(\left(\prod_{j=i+1}^{L+1} \mathbf{W}^{(j)}\right) \cdot \mathbf{b}^{(i)}\right) + \mathbf{b}^{(L+1)}.$$

Here, $\prod_{i=1}^{L+1} \mathbf{W}^{(i)}$ represents the product of all weight matrices from $i = 1$ to $L + 1$ (i.e., new weight matrix), and $\sum_{i=1}^{L} \left(\left(\prod_{j=i+1}^{L+1} \mathbf{W}^{(j)}\right) \cdot \mathbf{b}^{(i)}\right) + \mathbf{b}^{(L+1)}$ represents the sum of all the bias terms (i.e., new bias). Let

$$\bar{\mathbf{W}} = \prod_{i=1}^{L+1} \mathbf{W}^{(i)},$$

$$\bar{\mathbf{b}} = \sum_{i=1}^{L} \left(\left(\prod_{j=i+1}^{L+1} \mathbf{W}^{(j)}\right) \cdot \mathbf{b}^{(i)}\right) + \mathbf{b}^{(L+1)}.$$

Finally, we can express this as a single-layer NN:

$$\mathbf{O} = \bar{\mathbf{W}} \cdot \mathbf{x} + \bar{\mathbf{b}}.$$

So, using only the identity AF in all layers collapses the entire NN into a single-layer NN with a combined weight matrix and bias vector.

■