US Patent & Trademark Office
Patent Public Search | Text View

United States Patent 12393679
Kind Code B2
Date of Patent August 19, 2025
Inventor(s) Marson; Mark Evan et al.

Protection of neural networks by obfuscation of neural network operations and architecture

Abstract

Aspects of the present disclosure involve implementations that may be used to protect neural network models against adversarial attacks by obfuscating neural network operations and architecture. Obfuscation techniques include obfuscating weights and biases of neural network nodes, obfuscating activation functions used by neural networks, as well as obfuscating neural network architecture by introducing dummy operations, dummy nodes, and dummy layers into the neural networks.

Inventors: **Marson; Mark Evan (Carlsbad, CA), Hamburg; Michael Alexander ('s-Hertogenbosch, NL), Handschuh; Helena (Palo Alto, CA)**

Applicant: **CRYPTOGRAPHY RESEARCH, INC.** (San Jose, CA)

Family ID: **1000008763681**

Assignee: **CRYPTOGRAPHY RESEARCH, INC. (San Jose, CA)**

Appl. No.: **18/267773**

Filed (or PCT Filed): **December 16, 2021**

PCT No.: **PCT/US2021/063880**

PCT Pub. No.: **WO2022/140163**

PCT Pub. Date: June 30, 2022

Prior Publication Data

Document Identifier	Publication Date
US 20240078308 A1	Mar. 07, 2024

Related U.S. Application Data

us-provisional-application US 63199363 20201221
us-provisional-application US 63199365 20201221
us-provisional-application US 63199364 20201221

Publication Classification

Int. Cl.: **G06F21/55** (20130101); G06F21/14 (20130101)

U.S. Cl.:

CPC **G06F21/55** (20130101); G06F21/14 (20130101)

Field of Classification Search

CPC: G06F (21/55); G06F (21/14)

USPC: 726/22

References Cited

U.S. PATENT DOCUMENTS

Patent No.	Issued Date	Patentee Name	U.S. Cl.	CPC
10733292	12/2019	Araujo et al.	N/A	N/A
11604973	12/2022	Sather	N/A	G06F 17/16
12061981	12/2023	Sather	N/A	G06N 3/084
2010/0180346	12/2009	Nicolson	726/26	G06F 21/14
2019/0130110	12/2018	Lee	N/A	G06N 3/08
2019/0244081	12/2018	Franca-Neto	N/A	G06N 3/084
2020/0160978	12/2019	Prosky	N/A	G06N 5/04
2020/0311540	12/2019	Chakraborty et al.	N/A	N/A
2020/0364545	12/2019	Shattil	N/A	G06N 3/08
2022/0180148	12/2021	Kubosawa	N/A	G06N 3/045
2022/0269928	12/2021	Esmaeilzadeh	N/A	G06N 3/044
2025/0021670	12/2024	Yang	N/A	G06F 21/72

OTHER PUBLICATIONS

Batina, Lejla et al., “CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel”, Proceedings of the 28th USENIX Security Symposium, Aug. 14-16, 2019, Santa Clara, CA. 19 pages. cited by applicant

Carlini, Nicholas et al., “Cryptanalytic Extraction of Neural Network Models”, In: Micciancio D., Ristenpart T. (eds) Advances in Cryptology—CRYPTO 2020. CRYPTO 2020. Lecture Notes in Computer Science, vol. 12172. Springer, Cham. https://doi.org/10.1007/978-3-030-56877-1_7, International Association for Cryptologic Research 2020. 30 pages. cited by applicant

Dubey, Anuj et al., “MaskedNet: The First Hardware Inference Engine Aiming Power Slide-Channel Protection”, Proceedings of the 2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), May 4-7, 2020, San Jose, CA. 13 pages. cited by applicant

Hua, Weizhe et al., “Reverse Engineering Convolutional Neural Networks Through Side-Channel Information Leaks”, DAC '18: Proceedings of the 55th Annual Design Automation Conference, Jun. 2018 https://doi.org/10.1145/3195970.3196105. 6 Pages. cited by applicant

Notification of Transmittal of the International Search Report and the Written Opinion of the International Searching Authority, or the Declaration with Mail Date May 6, 2022 re: Int'l Appln. No. PCT/US2021/063880. 20 pages. cited by applicant

Oh, Seong Joon et al., “Towards Reverse-Engineering Black-Box Neural Networks”, In: Samek W., Montavon G., Vedaldi A., Hansen L., Müller KR. (eds) Explainable AI: Interpreting, Explaining and Visualizing Deep Learning. Lecture Notes in Computer Science, vol. 11700. Springer, Cham. https://doi.org/10.1007/978-3-030-28954-6_7. ICLR Feb. 14, 2018. 20 pages. cited by applicant

EP Extended European Search Report with Mail Date Dec. 3, 2024 re: EP Appln. No. 21911922.9. 15 pages. cited by applicant

Sun, Zhichuang et al., “ShadowNet: A Secure and Efficient System for On-device Model Inference”, arxiv.org, Cornell University Library, 201 Olin Library Cornell University Ithaca, NY 14853, Nov. 11, 2020, XP081811707, arXiv:2011.05905v1, 30 pages. cited by applicant

---

Primary Examiner: Lemma; Samson B

Attorney, Agent or Firm: Lowenstein Sandler LLP

Background/Summary

RELATED APPLICATIONS (1) This application is a 371 application of International Application No. PCT/US2021/063880, filed Dec. 16, 2021, which claims the benefit of U.S. Provisional Patent Application No. 63/199,363, filed Dec. 21, 2020, U.S. Provisional Patent Application No. 63/199,364, filed Dec. 21, 2020, and U.S. Provisional Patent Application No. 63/199,365, filed Dec. 21, 2020, which is incorporated herein by reference.

TECHNICAL FIELD
(1) The disclosure pertains to neural network computing applications, more specifically to protecting neural network models against adversarial attacks.

Description

BRIEF DESCRIPTION OF THE DRAWINGS
(1) The present disclosure will be understood more fully from the detailed description given below and from the accompanying drawings of various implementations of the disclosure.
(2) FIG. **1** illustrates an example network architecture in which various implementations of the present disclosure may operate.
(3) FIG. **2**A illustrates example operations of protection of neural network operations using weight and bias obfuscation, in accordance with one or more aspects of the present disclosure.
(4) FIG. **2**B depicts schematically matrix operations that may be performed as part of example operations of FIG. **2**A, in accordance with one or more aspects of the present disclosure.
(5) FIG. **3**A illustrates example operations of protection of a neural network by obfuscating an activation function of a neural network node, in accordance with one or more aspects of the present disclosure.
(6) FIG. **3**B illustrates example operations of protection of a neural network by using multiple obfuscating activation functions of a neural network node, in accordance with one or more aspects of the present disclosure.
(7) FIG. **4** depicts a flow diagram of an example method of protecting neural network operations using obfuscation of weights and biases, in accordance with one or more aspects of the present disclosure.
(8) FIG. **5** depicts a flow diagram of an example method of protecting neural network operations using activation function obfuscation, in accordance with one or more aspects of the present disclosure.
(9) FIGS. **6**A-E illustrate example operations of protection of neural network architecture by obfuscation using dummy operations, in accordance with one or more aspects of the present disclosure. FIG. **6**A illustrates an example implementation of a constant-output node that may be used for obfuscation of neural network operations. FIG. **6**B illustrates an example implementation of a pass-through node that may be used for obfuscation of neural network operations. FIG. **6**C illustrates an example implementation of a pass-through cluster of nodes that may be used for obfuscation of neural network operations. FIG. **6**D illustrates an example implementation of dummy (inconsequential) outputs from a node that may be used for obfuscation of neural network operations. FIG. **6**E illustrates an example implementation of node clusters having cancelling outputs that may be used for obfuscation of neural network operations.
(10) FIG. **7** depicts a flow diagram of an example method of protecting neural network operations using obfuscation of neural network architecture, in accordance with one or more aspects of the present disclosure.
(11) FIG. **8** depicts a block diagram of an example computer system operating in accordance with one or more aspects of the present disclosure.
DETAILED DESCRIPTION
(12) Aspects of the present disclosure are directed to protection of neural networks (neural network models) against adversarial attacks and other unauthorized accesses. More specifically, aspects of the present disclosure are directed to obfuscating weights, biases, activation functions, as well as topology of neural networks and computations performed by neural networks, using a variety of obfuscation techniques.
(13) An artificial neural network (NN) is a collection of computational operations that emulate how a biological NN operates and that may be used in a variety of applications, such as object and pattern recognition, voice recognition, text recognition, robotics, decision making, game playing, behavior modeling, and numerous other tasks. ANN often may be mapped as a graph that includes a collection of nodes and edges, where computations are performed within nodes and the data (inputs and outputs of the nodes) flows along various edges connecting the nodes. Nodes may be arranged in layers, with input layer receiving input data (e.g., a digital representation of an image) and an output layer delivering an output (e.g., image classification) of the NN. Depending on a domain-specific problem solved by the NN, any number of hidden layers may be positioned between the input layer and the output layer. Various NN architectures may include feed-forward NNs, recurrent NNs, convolutional NNs, long/short term memory NNs,

Boltzmann machines, Hopfield NNs, Markov NNs, and many other types of NNs.

(14) A node of a NN may receive a plurality of input values {x.sub.i} (e.g., from other nodes of the same NN or from an outside input agent, such as an image digitizing device). The node may be associated with a respective plurality of weights {w.sub.i} that weigh the input values and may further include a bias value b. In particular, the inputs into the node may be weighted and biased,

(15) $z = \sum_{i=1}^{N} w_i x_i + b$,

to produce a weighted input z for the node. The weighted input z may then be input into an activation function (AF) to obtain the output of the node

$y=f(z)$.

The activation function may be selected from a variety of functions, such as a step function (e.g., Heaviside function), rectified linear function (which may be a leaky rectified linear function), sigmoid function, softmax function, and the like. The output value y may be provided as an input into one or more nodes of the next layer (or the same layer or, in some instances, into the same node, and so on). Each node may have its own set of weights {w.sub.i}, bias b, and activation function f(z), referred herein as node parameters. While each node may, potentially, output different values y to different nodes of the next layer (e.g., by having a first set of node parameters to generate output value y.sub.1 to serve as an input into node 1 and output value y.sub.2 to serve as an input into node 2), a situation of N outputs from a given node may, equivalently, be represented via N distinct nodes each having the same output into all downstream nodes to which the respective node is connected. Accordingly, for conciseness, it shall be assumed herein that a node has the same output into all its downstream nodes (however, as described in more detail below, in some implementations of the present disclosure, the output received by a single downstream node may include multiple output values y.)

(16) Specific node parameters are determined during NN training (e.g., using training input and target outputs) and may represent a trade secret that a developer of the NN may wish to keep secret even when NN is published or is made available to a customer or another user. A user may run the NN and benefit from its output but may have no access to the actual NN parameters or NN architecture, such as the number of edges leading to/from various nodes, the number of layers, the number of nodes in each layer (including input and output layers), and so on. In some instances, an adversarial (e.g., side-channel) attack may be attempted against the NN to reveal the NN parameters and architecture. For example, a side-channel attack may be performed by monitoring emissions (signals) produced by electronic circuits (e.g. processor, memory, etc.) when the circuits are executing operations of the NN. Such signals may be electrical, acoustic, electromagnetic, optical, thermal, and so on. By recording emissions, a hardware trojan and/or malicious software may be capable of correlating specific processor (and/or memory) activity with operations carried out by the processor/memory. For example, an attacker employing a trojan may be able to detect emissions corresponding to multiple multiplication operations where different inputs are processed using the same NN parameters. As a result, by analyzing (e.g., using methods of statistical analysis) hardware emissions of the processing device, the attacker may be able to determine the values of the weights and biases, the type of AFs used, the numbers of nodes/connections/layers, and so on.

(17) Aspects and implementations of the present disclosure address these and other problems of the existing technology by disclosing systems and methods of protecting NNs against adversarial attacks, reverse engineering of the NNs, and other unauthorized operations. More specifically, disclosed is a method of protecting NNs by obfuscating weights and biases of various nodes of a NN by expanding the number of weights and biases, e.g., by using dummy weights and biases that do not affect the ultimate output of the NN but present an attacker with a wider range of possible values to be determined, as the number of dummy weights and biases may be more (in some implementations, much more) that the number of actual weights and biases of the NN. In some implementations, dummy weights and biases may be randomly selected, thus increasing the challenges to an attacker who attempts to use an adversarial attack against the NN. Weights and biases may further be masked using various linear or reversible non-linear transformations. In some implementations, actual AFs may be masked by dummy AFs deployed for at least some of the nodes. In such implementations, a node may provide multiple output values to another node, each output computed using a different activation function. In some implementations, only one of the outputs may be the actual output of the node whereas other outputs may be dummy outputs intended to obfuscate the actual output. In some implementations, none of the outputs may be an actual output of the node. Instead, the actual output may be a certain combination of some or all of the NN's outputs (and of the underlying AFs), which combination may not be known to the attacker. Additionally, security of the NN may be further enhanced by having at least some of the nodes performing dummy operations that do not affect the actual output of the NN, but whose purpose is to make it more difficult for the attacker to focus on the actual computations and the actual data flows of the NN. Such dummy operations may include dummy computations involving real inputs and real nodes, computations involving dummy nodes, computations involving splitting of computations across multiple layers, computations involving whole dummy layers, and so on.

(18) FIG. **1** is a block diagram illustrating an example computer system **100** in which various implementations of the

present disclosure may operate. The example computer system **100** may be a desktop computer, a tablet, a smartphone, a server (local or remote), a thin/lean client, and the like. The example computer system **100** may be a system dedicated to one or more domain-specific applications **110** (e.g., object recognition application, decision-making application), and so on. The example computer system **100** may include, but not be limited to, a computer device **102** having one or more processors (e.g., capable of executing binary instructions) such as central processing units (CPUs) **120**, one or more graphics processing units (GPUs) **122**, and one or more system memory **130** devices. "Processor" may further refer to any device capable of executing instructions encoding arithmetic, logical, or I/O operations. In one illustrative example, a processor may follow Von Neumann architectural model and may include an arithmetic logic unit (ALU), a control unit, and a plurality of registers.

(19) Computer device **102** may further include an input/output (I/O) interface **104** to facilitate connection of the computer device **102** to peripheral hardware devices **106** such as card readers, terminals, printers, scanners, internet-of-things devices, and the like. The computer device **102** may further include a network interface **108** to facilitate connection to a variety of networks (Internet, wireless local area networks (WLAN), personal area networks (PAN), public networks, private networks, etc.), and may include a radio front end module and other devices (amplifiers, digital-to-analog and analog-to-digital converters, dedicated logic units, etc.) to implement data transfer to/from computer device **102**. Various hardware components of computer device **102** may be connected via a bus **112** which may have its own logic circuits, e.g., a bus interface logic unit.

(20) Computer device **102** may support one or more domain-specific applications **110**, including any application that uses neural networks. Applications **110** may be instantiated on the same computer device **102**, e.g., by an operating system executed by CPU **120** and residing in the system memory **130**. Alternatively, application **110** may be instantiated by a guest operating system supported by a virtual machine monitor (hypervisor) executed by the CPU **120**. In some implementations, applications **110** may reside on a remote access client device or a remote server (not shown), with computer device **102** providing computational support for the client device and/or the remote server.

(21) CPU **120** may include one or more processor cores having access to a single or multi-level cache and one or more hardware registers. In implementations, each processor core may execute instructions to run a number of hardware threads, also known as logical processors. Various logical processors (or processor cores) may be assigned to one or more applications **110**, although more than one processor core (or a logical processor) may be assigned to a single application **110** for parallel processing. A multi-core CPU **120** may simultaneously execute multiple instructions. A single-core CPU **120** may typically execute one instruction at a time (or process a single pipeline of instructions). CPU **120** may be implemented as a single integrated circuit, two or more integrated circuits, or may be a component of a multi-chip module.

(22) GPU **122** may include multiple cores each capable of executing multiple threads (e.g., in parallel) using GPU memory **124**. In some implementations, GPU **122** may execute at least some operations of NNs. For example, various GPU **122** threads may execute various operations (e.g., computation of weighed inputs and applying activation functions to the weighted inputs of the same layer NN nodes) in parallel. GPU **122** may include a scheduler (not shown) and a dispatch unit to distribute execution of computational tasks among different threads of GPU cores.

(23) System memory **130** may refer to a volatile or non-volatile memory and may include a read-only memory (ROM) **132**, a random-access memory (RAM) **134**, as well as (not shown) electrically-erasable programmable read-only memory (EEPROM), flash memory, flip-flop memory, or any other device capable of storing data. The RAM **134** may be a dynamic random-access memory (DRAM), synchronous DRAM (SDRAM), a static memory, such as static random-access memory (SRAM), and the like.

(24) System memory **130** may be used to store inputs into NNs (e.g., images received from applications **110**), outputs generated by NNs (e.g., identifications of objects captured by the images and classifications of such objects), parameters of the NNs, obfuscation data, masking data, and/or any other data. System memory **130** may include one or more registers **136**, which may be used to store some of the data related to execution of NNs. In some implementations, registers **136** may be implemented as part of RAM **134**. In some implementations, some or all of registers **136** may be implemented separately from RAM **134**. Some or all registers **136** may be implemented as part of the hardware registers of CPU **120**. Some inputs or outputs of nodes of NNs and/or intermediate values (e.g., weighted inputs) may be stored in GPU memory **124**.

(25) Computer device **102** may include a neural network engine **103** to execute one or more NNs. NN engine **103** may be located within system memory **130** or in a separate memory device. NN engine **103** may further include a neural network obfuscation engine (NOE) **105** to implement obfuscation of NN parameters, topology, and operations, as described in the present disclosure. NOE **105** may be implemented in software and/or hardware, firmware, or in any combination thereof.

(26) FIG. **2**A illustrates example operations **200** of protection of neural network operations using weight and bias obfuscation, in accordance with one or more aspects of the present disclosure. Example operations **200** may be implemented by NN engine **103** in conjunction with NN obfuscation engine **105**. Example operations **200** are illustrated for a single node of a NN being used, but similar operations may be performed for any number (or all)

nodes of the NN. FIG. **2**B depicts schematically matrix operations **250** that may be performed as part of example operations **200** of FIG. **2**A, in accordance with one or more aspects of the present disclosure.

(27) A node input **202** may include p input values {x.sub.i} henceforth denoted as a row vector {right arrow over (x)}=(x.sub.1, . . . , x.sub.p). Stored weights **204** {w.sub.i}, similarly combined into a vector {right arrow over (w)}>=(w.sub.1, . . . , w.sub.p), together with stored bias **206** *b* may determine the target output y=*f*({right arrow over (w)}.Math.{right arrow over (x)}.sup.T+b) of the node. To obfuscate operations leading to the output y, NOE **105** may generate n−1 additional (dummy) weight vectors and expand (block **208**) the weight vector {right arrow over (w)} into an n×p weight matrix

{right arrow over (w)}.fwdarw.Ŵ,Ŵ=[{right arrow over (w)}.sub.1, . . . ,*{right arrow over (w)}*.sub.n],

where, in one implementation, one of the vectors {right arrow over (w)}.sub.j may be the vector {right arrow over (w)} of the actual inputs whereas other vectors may be dummy weight vectors containing obfuscation weights. For the sake of specificity only, in some illustrations it may be assumed that vector {right arrow over (w)}.sub.1 may be vector {right arrow over (w)} of the actual weights while other vectors {right arrow over (w)}.sub.j≠1 are dummy weight vectors.

(28) Furthermore, NOE **105** may generate n−1 additional (dummy) biases to expand (block **210**) bias b into an n-component bias vector

b.fwdarw.{right arrow over (B)},{right arrow over (B)}=(*b*.sub.1, . . . ,*b*.sub.n),

where, in one implementation, one of the components b.sub.j may be the actual bias b whereas other components may be dummy biases. For the sake of specificity only, in some illustrations, it may be assumed that the component b.sub.1 of the vector may be the actual bias b while other biases b.sub.j≠1 are dummy biases. Dummy weights and dummy biases may be generated randomly, chosen from a previously prepared list of values (which may be periodically updated), or selected in any other manner.

(29) Using expanded weights and biases, the weighted input z into the node may be computed by first computing the n-component column vector Ŵ.Math.{right arrow over (x)}.sup.T+{right arrow over (B)}.sup.T, and then selecting an appropriate component for the weighted input z. For example, in an illustration where real weights and biases correspond to the first row of matrix Ŵ and the first component of vector {right arrow over (B)}, respectively, the first component of vector Ŵ.Math.{right arrow over (x)}.sup.T+{right arrow over (B)}.sup.T yields the actual weighted input into the activation function of the node (whereas other components of the vector obfuscate the actual weighted input).

(30) In some implementations, prior to computing the vector of weighted inputs, NOE **105** may perform additional masking of the weights and biases. In the following description, the concept of masking is first illustrated using linear masking examples, but it should be understood (see also a discussion below) that any non-linear masking in which unmasked values can be recovered from their masked values may also be used. In one example implementation, masking may be performed using a masking matrix {circumflex over (M)} (block **212**), which may be a k×n matrix, with k that may be different from p and/or n. Specifically, masking matrix {circumflex over (M)} may be used to mask the expanded weights by determining (at block **214**) the matrix product Ŵ.sub.M={circumflex over (M)}.Math.Ŵ to obtain a k×n matrix of masked weights Ŵ.sub.M. Similarly, NOE **105** may perform additional masking of expanded biases by determining (at block **216**) a k-component vector of masked biases {right arrow over (B)}.sub.M.sup.T={circumflex over (M)}.Math.{right arrow over (B)}.sup.T. The weighted masked input vector **218** (likewise, a k-component vector) may now be computed as the sum:

{right arrow over (Z)}.sub.M=*Ŵ*.sub.M.*Math.{right arrow over (x)}*.sup.T+*{right arrow over (B)}*.sub.M.sup.T.

To obtain a correct weighted input z into the activation function of the node (to obtain the node output y=*f*(z)), the masked input may be processed using an unmasking vector {right arrow over (U)}, e.g., z={right arrow over (U)}.Math.{right arrow over (Z)}.sub.M. The unmasking vector (block **220**) may be determined based on the inverse matrix {circumflex over (M)}.sup.−1. For example, in an illustration where real weights and biases correspond to the first row of matrix Ŵ and the first component of vector {right arrow over (B)}, respectively, the unmasking vector may be determined as

{right arrow over (U)}=(1,0, . . . ,0).Math.*{circumflex over (M)}*.sup.−1,

Where the n-component sampling vector {right arrow over (S)}=(1, 0, . . . , 0) extracts the first row of the n×k inverse masking matrix {circumflex over (M)}.sup.−1.

(31) To avoid performing unmasking operation (multiplication by {right arrow over (U)}) after computing the weighted masked input **218** (and thus exposing the weighted input z to a potential attacker), the unmasking operation may be combined with computation of the activation function *f*(z) **222**. More specifically, instead of computing the node output value **226** by applying the activation function *f*(z) to the unmasked weighted input

y=f(z)=*f*(*{right arrow over (U)}.Math.{right arrow over (Z)}*.sub.M),

NOE **105** may first determine (block **224**) a composite activation function *f*°{right arrow over (U)} that operates on the weighted masked input {right arrow over (Z)}.sub.M directly and produces the same result as the activation function *f* acting on the unmasked weighted input z:

(*f*°*{right arrow over (U)}*)(*{right arrow over (Z)}*.sub.M)≡*f*(*{right arrow over (U)}.Math.{right arrow over*

(Z)}.sub.M).
As a result, composite activation function **224** (*f*°{right arrow over (U)}), determined based on activation function **222** (*f*) and unmasking vector **220** ({right arrow over (U)}), and being applied to weighted masked input **218** ({right arrow over (Z)}.sub.M), generates the correct output value **226** (y).
(32) In various implementations, any non-linear masking (in which information about the actual weights and biases is not lost) may be used instead of linear masking. For example, expanded weight matrix $\hat{W}$ may be nonlinearly masked using a first (non-linear) masking transformation, such that an element ($\hat{W}$.sub.M).sub.ij of the masked weight matrix $\hat{W}$.sub.M is a function of one or more (or even all) elements of the expanded weight matrix $\hat{W}$: ($\hat{W}$.sub.M).sub.ij=F.sub.ij($\hat{W}$). The first transformation F.sub.ij may be reversible, in a sense that there may exist an inverse (unmasking) transformation that determines an element $\hat{W}$.sub.ij based on of one or more elements of the masked weight matrix $\hat{W}$.sub.M:$\hat{W}$.sub.ij=F.sup.−1.sub.ij($\hat{W}$.sub.M). Similarly, expanded bias vector {right arrow over (B)} may be nonlinearly masked using a second transformation, such that an element ({right arrow over (B)}.sub.M).sub.i of the masked bias vector {right arrow over (B)}.sub.M is a function of one or more (or even all) elements of the expanded bias vector {right arrow over (B)}:({right arrow over (B)}.sub.M).sub.i=G.sub.i({right arrow over (B)}). The second transformation G.sub.i may also be reversible, in a sense that there may exist an inverse (unmasking) transformation that determines an element ({right arrow over (B)}).sub.i based on of one or more elements of the masked bias vector {right arrow over (B)}.sub.M: ({right arrow over (B)}).sub.i=G.sup.−1.sub.i({right arrow over (B)}.sub.M). The masked weight matrix $\hat{W}$.sub.M and the masked bias vector {right arrow over (B)}.sub.M may be used to compute the masked weighted inputs. An activation function *f*(z) may be composed with the first masking transformation F.sup.−1 and the second masking transformation G.sup.−1, and may further be composed with the sampling vector {right arrow over (S)} to extract correct elements of the masked weighted inputs. The composed activation function may act directly on the masked weighted inputs without unmasking actual weighted inputs, substantially as described in relation to linear masking.
(33) Periodically, masking update module **228** of NOE **105** may update (as depicted schematically with dashed arrows) masking matrix **212** ({circumflex over (M)}), weight masking **214** ($\hat{W}$.sub.M), bias masking **216** ({right arrow over (B)}.sub.M), unmasking vector **220** ({right arrow over (U)}), and composite activation function **224** (*f*°{right arrow over (U)}) without unmasking the weights, biases or activation function. Masking update may be performed for every instance of a NN execution, after a certain number of NN executions is performed, after a certain time has passed, and so on. In some implementations, updated weights $\hat{W}$.sub.M′ and biases {right arrow over (B)}.sub.M′ may be obtained by generating a new k×n masking matrix {circumflex over (M)}′ and computing masked weights $\hat{W}$.sub.M′={circumflex over (M)}′.Math.$\hat{W}$ and biases {right arrow over (B)}.sub.M.sup.T′={circumflex over (M)}′.Math.{right arrow over (B)}.sup.T musing expanded weights $\hat{W}$ and expanded biases {right arrow over (B)}.sup.T that include actual weights {right arrow over (w)}, and actual bias b. In some implementations, to protect weights and biases from added exposure to adversarial attacks, updated weights and biases may be determined sequentially, based on previous (e.g. most recent) weights and biases, by using a square k×k masking matrix {circumflex over (M)}′, as follows,
$\hat{W}$.sub.M′*={circumflex over (M)}′.Math.$\hat{W}$*.sub.M≡*{circumflex over (M)}′.Math.{circumflex over (M)}.Math.$\hat{W}$,{right arrow over (B)}*.sub.M.sup.T′*={circumflex over (M)}′.Math.{right arrow over (B)}*.sub.M.sup.T≡*{circumflex over (M)}′.Math.{circumflex over (M)}.Math.{right arrow over (B)}*.sup.T.
The updated composite activation function, (*f*°{right arrow over (U)})°{right arrow over (U)}′, may similarly be obtained based on a recent (e.g., the most recent) activation function, *f*°{right arrow over (U)}. In some implementations, matrix {circumflex over (M)}′ need not be a square matrix, but may be a rectangular k′×k matrix, so that the number of elements in $\hat{W}$.sub.M′ and biases {right arrow over (B)}.sub.M′ is different than the number of elements in $\hat{W}$.sub.M and biases {right arrow over (B)}.sub.M, for additional protection.
Nonlinear masking operations can also be updated in a similar manner.
(34) In some implementations, the weighted inputs may additionally be masked (e.g., multiplicatively and/or additively) before the weighted inputs are input into the node's activation function. Even though the obfuscation implementations described in relation to FIGS. **2**A and **2**B involve linear transformations, in some implementations non-linear obfuscation and masking may be used. For example, an invertible non-linear transformation may be used instead of masking matrix {circumflex over (M)}, with the inverse of the non-linear transformation composed with the activation function acting on the output of the non-linear transformation.
(35) As a result of the obfuscation operations disclosed above (or other similar operations), the weighted input into the node's activation function may be multiplicatively (with mask m.sub.1) and additively (with mask m.sub.2) masked compared with the target weighted input:
z.fwdarw.z=m*.sub.1.*Math.z+m*.sub.2.
Corresponding obfuscation operations for various activation functions are illustrated below. It should be understood that the list of activation functions that may be used in various implementations consistent with the present disclosure is practically unlimited and that activation functions described below are intended as illustrations only.
(36) In some implementations, the Heaviside function Θ(z) (step function) may be used as the activation function,

y=Θ(m.sub.1.Math.z+m.sub.2). In such implementations, masking obfuscates the point z=−m.sub.2/m.sub.1 which separates the input values z for which the output is zero (z<−m.sub.2/m.sub.1) from the input values for which the output is non-zero (z<−m.sub.2/m.sub.1). Because the masked output y.sub.M=Θ(m.sub.1.Math.z+m.sub.2)=Θ(z+m.sub.2/m.sub.1) is different from the correct output y=Θ(z), an additional adjustment operation may subsequently be performed to determine the correct output y.sub.M.fwdarw.y, e.g., using y=y.sub.M.fwdarw.y−sign(m.sub.2/m.sub.1). Θ(−z(z+m.sub.2/m.sub.1)). Such an unmasking operation may be performed as part of computations associated with the next (downstream) node (or multiple nodes), into which the masked output value y.sub.M is provided. To prevent an attacker from identifying the location of the point (−m.sub.2/m.sub.1) where the output values become non-zero, NOE **105** may use additional techniques, such as shifting the step function into a same-sign domain (e.g., positive or negative) of the output values. For example, NOE **105** may select additional masks, e.g., m.sub.3 and m.sub.4, to determine outputs that are positive (or negative) for both positive and negative inputs z, for example: y.sub.M=m.sub.3.Math.Θ(m.sub.1.Math.z+m.sub.2)+m.sub.4.Math.Θ(−m.sub.1.Math.z−m.sub.2). In some implementations, an order of evaluation of the conditions for the sign of the argument of the step functions may be randomized. For example, during evaluation of the step function, NOE **105** may randomly choose to evaluate which condition takes place, m.sub.1.Math.z+m.sub.2>0 or m.sub.1.Math.z+m.sub.2<0, to further obfuscate where the location of the point −m.sub.2/m.sub.1 actually is. Adjustment for masks m.sub.3 and m.sub.4 may then be done (e.g., at the next node(s)) similarly to how adjustment for masks m.sub.1 and m.sub.2 may be performed. In some implementations, all masks m.sub.1, m.sub.2, m.sub.3 and m.sub.4 may be deployed. In some implementations, any of the masks m.sub.1, m.sub.2, m.sub.3 and m.sub.4 may be omitted. Similar techniques may be used to obfuscate a point of discontinuity of an activation function that is not a step function but some other discontinuous function.

(37) In some implementations, an activation function may be a rectified linear function (with or without a leak) that has a knee at some input value, which will be set to zero for the sake of conciseness (although a generalization to the knee located at arbitrary value of z is possible), e.g., y=α.Math.z.Math.Θ(z)+β.Math.z.Math.Θ(−z). When a masked value, e.g., z.sub.M=z+m.sub.1, is input into such an activation function, NOE **105** may generate an additional mask m.sub.2 and compute sums and differences of the two masks m.sub.±=m.sub.2±m.sub.1 and then compute two additional input values:

z.sub.+=z.sub.M+*m*.sub.−≡*z*+*m*.sub.2*;* z.sub.−=z.sub.M−*m*.sub.+≡*z*−*m*.sub.2.

Subsequently, NOE **105** may compare the values of z.sub.+ and z.sub.− to identify the sign of the actual input value z without unmasking z. For example, if a positive mask m.sub.2 is selected and it is determined that |z.sub.+|>|z.sub.−|, NOE **105** may identify that z>0 and apply the positive part (α-part) of the activation function: y.sub.M=α.Math.z.sub.M. The adjustment for the mask m.sub.1 may then be performed as part of the next node's computations, e.g., y=y.sub.M−α.Math.m.sub.1. Alternatively, the adjustment for the mask m.sub.1 may performed, e.g., by adjusting weights and biases of that next node to compensate for the extra value α.Math.m.sub.1 contained in y.sub.M. Similarly, if a positive mask m.sub.2 is selected and it is determined that |z.sub.+|<|z.sub.−|, NOE **105** may identify that z<0 and apply the negative part (β-part) of the activation function: y.sub.M=β.Math.z.sub.M. The adjustment for the mask m.sub.1 may be performed as part of the next node's computations, e.g., y=y.sub.M−β.Math.m.sub.1, or, alternatively, the adjustment for the mask m.sub.1 may be performed by adjusting weights and biases of that next node to compensate for the extra value β.Math.m.sub.1 contained in y.sub.M. Similar techniques may be used to obfuscate a point of discontinuity of an activation function that is not a rectified linear function but some other function that has a discontinuous derivative.

(38) Additional obfuscation may be achieved by masking (rescaling) the slopes α and β, shifting the location of the knee from z=0 to a different point, and the like. Such additional obfuscation may be performed similarly to rescaling and shifting described above in relation to the Heaviside function.

(39) In some implementations, a rectified linear activation function may be used that has β=0 and α=1 (by deploying rescaling, a may also be given any other value): $f(x)=\mathrm{relu}(x)=(x+|x|)/2$. In some implementations, an input z into the rectified linear function may be a sum of the masked weighted input z.sub.m and a masking value m: y=relu(z.sub.m+m). The actual input z=z.sub.m+m may further be masked with a product of two additional masks, m.sub.1 and m.sub.2, using the following masking procedure:

|*m*.sub.1.*Math.m*.sub.2|relu(*z*.sub.m+*m*)=½.Math.|*m*.sub.1|.Math.

(|*m*.sub.2|.*Math.z*.sub.m+|*m*.sub.2|.*Math.m*)+½.Math.|*m*.sub.1.*Math.m*.sub.2.*Math.z*.sub.m+*m*.sub.1.*Math.m*.sub.2.*Math.m*|

Consequently, relu(z.sub.m+m) is masked by m.sub.1 and m.sub.2 without separately revealing the value z=z.sub.m+m. To further mask where the knee of the function relu(z.sub.m+m) is located, NOE **105** may introduce additional masking knees during computation of the masked value |m.sub.1.Math.m.sub.2|.Math.relu(z.sub.m+m). In particular, since z≡relu(z)−relu(−z) (or more generally, z≡relu(z−a)−relu(−z+a)+a), a masking knee at some (e.g., randomly chosen) point z=a may be introduced, e.g., the rectified linear function may be computed alternatively, as follows,

|*m*.sub.1.*Math.m*.sub.2|.Math.relu(*z*.sub.m+*m*)=½.Math.|*m*.sub.1|.Math.

(|*m*.sub.2|.Math.relu(*z*.sub.m+*a*)+|*m*.sub.2|.*Math.m*)+½.*Math.*|*m*.sub.1.*Math.m*.sub.2.*Math.z*+*m*.sub.1.*Math.m*.sub.2.*Math.m*

−½.*Math.*|*m*.sub.1|.Math.(|*m*.sub.2|.Math.relu(−z.sub.m−*a*)−|*m*.sub.2|.*Math.m*),
where the masking knee at z.sub.m=−a is introduced in a way that obfuscates intermediate computations but does not affect the ultimate result. Additional masking knees may be introduced in a similar manner, up to a target number of knees.
(40) In some implementations, where activation function **222** is a linear function, an unmasking operation (e.g., multiplication by vector {right arrow over (U)} prior) may commute with an application of the activation function: *f*({right arrow over (U)}.Math.{right arrow over (Z)}.sub.M)={right arrow over (U)}.Math.*f*({right arrow over (Z)}.sub.M). Accordingly, activation function *f* may be applied directly to weighted masked input {right arrow over (Z)}.sub.M while the unmasking operation (e.g., composite with the application of weights and biases) may be applied during computations of the next node. In some implementations, additional masking may be performed by a partial reduction of weighted masked input {right arrow over (Z)}.sub.M (which is an n-component vector) to an m-component vector. More specifically, in addition to the unmasking vector (which unmasks and selects a correct weighted output value z from {right arrow over (Z)}.sub.M),
{right arrow over (U)}=(1,0,0,0,0, . . . ,0,0).Math.*{circumflex over (M)}*.sup.−1,
another vector {right arrow over (U)}′ may be defined such that, when applied to vector of weighted inputs {right arrow over (Z)}.sub.M, it selects and sums various specific (e.g., randomly selected, in number and position) elements of the vector of the weighted inputs {right arrow over (Z)}.sub.M:
{right arrow over (U)}′=(0,1,1,0,0 . . . ,0,1).Math.*M*.sup.−1.
For example, the row vector {right arrow over (U)}′ illustrated here selects and adds together the second, the third, and the last components (for a total of n−1 components) of vector {circumflex over (M)}.sup.−1.Math.{right arrow over (Z)}.sub.M. A linear activation function applied to the partially reduced weighted masked input {right arrow over (Z)}.sub.M may now be represented as,
f(({*right arrow over (U)}*+{*right arrow over (U)}′*).Math.*{right arrow over (Z)}*.sub.M)=*f*({*right arrow over (U)}.Math.{right arrow over (Z)}*.sub.M+{*right arrow over (U)}′.Math.{right arrow over (Z)}*.sub.M)=*f*({*right arrow over (U)}.Math.{right arrow over (Z)}*.sub.M)+*f*({*right arrow over (U)}′.Math.{right arrow over (Z)}*.sub.M)={*right arrow over (U)}*.Math.*f*({*right arrow over (Z)}*.sub.M)+{*right arrow over (U)}′*.Math.*f*({*right arrow over (Z)}*.sub.M),
where the term {right arrow over (U)}.Math.*f*({right arrow over (Z)}.sub.M) represents an actual output of the node and the term {right arrow over (U)}′.Math.*f*({right arrow over (Z)}.sub.M) corresponds to a masking data. Accordingly, in one implementation, NOE **105** may input the total value ({right arrow over (U)}+{right arrow over (U)}′).Math.{right arrow over (Z)}.sub.M to the activation function and subtract the masking value {right arrow over (U)}′.Math.*f*({right arrow over (Z)}.sub.M) to determine the actual output of the node:
y≡{*right arrow over (U)}*.Math.*f*({*right arrow over (Z)}*.sub.M)=*f*(({*right arrow over (U)}*+{*right arrow over (U)}*′).Math.{*right arrow over (Z)}*.sub.M)−{*right arrow over (U)}*′.Math.*f*({*right arrow over (Z)}*.sub.M).
In some implementations, subtraction of the masking value {right arrow over (U)}′.Math.*f*({right arrow over (Z)}.sub.M) may be performed as part of the next node's computations (e.g., by using inputs and weights composite with the masking value).
(41) In some implementations, the sigmoid function may be used as an activation function. To determine the output value of the sigmoid function y=S(z)=[1+exp(−z)].sup.−1, based on the masked input z+m.sub.1 without unmasking the actual input z, the following procedure may be implemented. An additional (optional) masking value m.sub.2 may be selected (e.g., randomly or from an existing list, which may be periodically updated) and the exponential values may be computed:
f.sub.1(*z*)=exp(*z*+*m*.sub.1+*m*.sub.2), *f*.sub.2(*z*)=exp(*z*+*m*.sub.1), *f*.sub.3=exp(*m*.sub.1).
Using the computed values, the output value may be determined as
(42) $y_M = \frac{f_1(z)}{f_2(z)+f_3} = \frac{\exp(z+m_1+m_2)}{\exp(z+m_1)+\exp(m_1)} = y$ .Math. $\exp(m_2)$
Therefore, the output value y.sub.M (multiplicatively masked by exp(m.sub.2)) is determined without revealing the actual input z (which is handled in combination z+m.sub.1). Unmasking of the actual output y may then be performed using multiplication of y.sub.M by exp(−m.sub.2). Alternatively, adjustment for the masking may be performed as part of the next node's computations (e.g., using inputs and weights composite with the unmasking value exp(−m.sub.2)).
(43) Similarly, masking of the softmax activation function,
(44) $y = \text{softmax}(z_1, z_2) = \frac{\exp(z_1)}{\exp(z_1)+\exp(z_2)}$,
may be performed by computing
(45) $y_M = \frac{\exp(z_1+m_1+m_3)+\exp(z_1+m_2+m_3)+\exp(z_2+m_2+m_3)}{\exp(z_1+m_3)+\exp(z_2+m_3)} = y$ .Math. $\exp(m_1)+\exp(m_2)$.
As in the case of the sigmoid function, the output value y.sub.M (multiplicatively masked by exp(m.sub.1) and additively coupled by exp(m.sub.2)) is determined without revealing the inputs z.sub.1 and z.sub.2 (which are handled in respective combinations z.sub.1+m.sub.3 and z.sub.2+m.sub.3). Unmasking of the actual output y may be performed by using y=y.sub.M.Math.exp(−m.sub.1)−exp(m.sub.2−m.sub.1)), separately or in composition with

computations of the next node(s).

(46) FIG. **3**A illustrates example operations **300** of protection of a neural network by obfuscating an activation function of a neural network node, in accordance with one or more aspects of the present disclosure. Example operations **300** may be implemented by NN engine **103** in conjunction with NN obfuscation engine **105**. Example operations **300** are illustrated for a pair of nodes of a NN, e.g., a first node **301** and a second node **303**. Although operations **300** are shown, for conciseness, as being performed without obfuscation of weights and biases, it should be understood that operations **300** may also be combined with such obfuscation operations (e.g., as disclosed above in relation to FIGS. **2**A and **2**B).

(47) As shown in FIG. **3**A, a first node input **302**, e.g., a row vector {right arrow over (x)}=(x.sub.1, . . . , x.sub.p), may be weighted using weights **304**, e.g., {right arrow over (w)}=(w.sub.1, . . . , w.sub.p), and biased with bias **306**, e.g., b, to determine a weighted input **318**, e.g., z={right arrow over (w)}.Math.{right arrow over (x)}.sup.T+b. Under normal (without obfuscation) node operations, weighted input **318** may be provided to activation function $f(z)$ to generate a node output y=f({right arrow over (w)}.Math.{right arrow over (x)}.sup.T+b) of the node. In some implementations, such normal operations may be modified to protect against adversarial attacks. More specifically, operations **300** may include obfuscation of activation function **322** to protect the nature (e.g., type) and specific form (e.g., parameters of the activation function) using an obfuscation function **312**, which may be randomly generated by NOE **105**, selected from a database of obfuscation functions, and so on. Obfuscation function **312**, e.g., s=g(f), may be an invertible function (such that there exists a unique value f=g.sup.−1(s) for a range of inputs f that may be output by activation function **322**). Obfuscation function **312** may be applied to the node output y and may produce obfuscated output O=g (f(z)). In order to obfuscate activation function $f(z)$ **322**, NOE **105** may compute the obfuscated output O without revealing $f(z)$. In particular, NOE **105** may obtain a composite activation function g°f **324** that applies to the weighted input **318** directly,

$$O \equiv g(f(z))=(g \circ f)(z),$$

computing the obfuscated output O without performing the intermediate step of computing $f(z)$. In one non-limiting example, activation function **322** may be a sigmoid function $f(z)=[1+\exp(-z)]$.sup.−1. NOE **105** may obfuscate activation function **322** by selecting the natural logarithm obfuscation function **312**, g(f)=ln f. The composite activation function **324** may, therefore, be first selected as g°$f(z)=z-\ln[\exp(z)+1]$.

(48) The obfuscated output O may then be provided (or made available) to second node **303** as (obfuscated) input **330**. Additionally, a de-obfuscation function **332** may be provided to second node **303**. De-obfuscation function **332** may be an inverse f=g.sup.−1(s) to obfuscation function **312** s=g(f). For example, if obfuscation function **312** is selected to be g(f)=exp(f)−1, de-obfuscation function **332** may be g.sup.−1(s)=ln(1+s).

(49) De-obfuscation function **332** may be used to identify an actual output value y of first node **301** without revealing the output value y. For example, a respective weight w (to weigh the input from first node **301**) of the weights **334** of second node **303** may be combined with de-obfuscation function **332** to form a respective composite weight, e.g., w′=wg.sup.−1, of the composite weights **335** of second node **303**. In some implementations, all inputs (e.g., from all upstream nodes) into second node **303** may similarly be obfuscated with respective obfuscation functions **312** and de-obfuscated using respective de-obfuscation functions **332**, e.g., by forming respective composite weights for each of the (upstream) nodes that provide inputs into second node **303**. Composite weights **335**, together with bias **336** may then be used to obtain a weighed input **338** into an activation function **340** of second node **303**. Composite weights **335** and bias **336** may further be masked using implementations described in relation to FIGS. **2**A and **2**B.

(50) FIG. **3**B illustrates example operations **350** of protection of a neural network by using multiple obfuscating activation functions of a neural network node, in accordance with one or more aspects of the present disclosure. Example operations **350** may be implemented by NN engine **103** in conjunction with NN obfuscation engine **105**. Example operations **350** may further protect the neural network by generating multiple activation functions **322**, such as f.sub.1(z), f.sub.2(z), . . . , to obfuscate the actual activation function $f(z)$ of first node **301**. Each activation function f.sub.j(z) may also be obfuscated with a respective obfuscation function **312**, e.g., g.sub.j(f), which may be an invertible function. Multiple obfuscation functions **312** may be applied to the node output y and may produce multiple obfuscated outputs O.sub.j=g.sub.j(f.sub.j(z)). NOE **105** may compute the obfuscated outputs O.sub.j without revealing f.sub.j(z). In particular, NOE **105** may obtain multiple composite activation functions g.sub.j°f.sub.j **324** that apply to the weighted input **318** directly,

$$O.\text{sub.j}=(g.\text{sub.j} \circ f.\text{sub.j})(z),$$

and compute the obfuscated output O.sub.j without performing intermediate steps of computing f.sub.j(z).

(51) The obfuscated output O.sub.j may then be provided to second node **303** as a vector obfuscated input **330**, e.g. {right arrow over (O)}=(O.sub.1, O.sub.2, . . .). (In some implementations, each upstream node that provides inputs into second node **303** may provide its own vector of obfuscated inputs **330**, each vector having a number of components that may be different from a number of components provided by other upstream nodes.) Additionally, to obfuscate which of the activation functions f.sub.1(z), f.sub.2(z), . . . , is an actual activation function and which functions are dummy functions, NOE **105** may perform mixing of outputs among the output components O.sub.j.

(52) As a non-limiting example intended only to illustrate such a mixing of outputs, activation function f.sub.1(z) may be the actual activation function whereas activation functions f.sub.2(z) and f.sub.3(z) may be dummy activation functions. Each activation function may be obfuscated with a respective obfuscation function g.sub.1, g.sub.2, or g.sub.3, as disclosed above. For conciseness of notations, such an obfuscation is henceforth implied but not stated explicitly. A masking matrix {circumflex over (M)} may be constructed that transforms a vector of activation functions {right arrow over (f)}=(f.sub.1, f.sub.2, . . .) into a vector of output values: {right arrow over (O)}={circumflex over (M)}.Math.{right arrow over (f)}. For example, a masking matrix

$$(53)\quad \hat{M} = \begin{pmatrix} m_1 & m_2 & m_3 \\ 0 & m_4 & 0 \\ 0 & 0 & m_5 \end{pmatrix}$$

generates the following vector of output values: O.sub.1=m.sub.1f.sub.1+m.sub.2f.sub.2+m.sub.3f.sub.3, O.sub.2=m.sub.2f.sub.2, O.sub.3=m.sub.3f.sub.3. Correspondingly, the following unmasking vector

$$(54)\quad \overset{\text{.fwdarw.}}{U} = \left(\frac{1}{m_1}, -\frac{m_2}{m_1 m_4}, -\frac{m_3}{m_1 m_5}\right)$$

may be used to unmask the correct activation function using the vector of output values, {right arrow over (U)}.Math.{right arrow over (O)}=f.sub.1(z)≡y. The unmasking of the actual output y may be performed in composition with computations of the composite weights **335** of second node **303**.

(55) FIG. **4** depicts a flow diagram of an example method **400** of protecting neural network operations using obfuscation of weights and biases, in accordance with one or more aspects of the present disclosure. Method **400**, as well as method **500** disclosed below, and/or each of their individual functions, routines, subroutines, or operations may be performed by one or more processing units of the computing system implementing the methods, e.g., CPU **120** and/or GPU **122** or some other processing device (an arithmetic logic unit, an FPGA, and the like, or any processing logic, hardware or software or a combination thereof). In certain implementations, each of methods **400** and **500** may be performed by a single processing thread. Alternatively, each of methods **400** and **500** may be performed by two or more processing threads, each thread executing one or more individual functions, routines, subroutines, or operations of the method. In an illustrative example, the processing threads implementing each of methods **400** and **500** may be synchronized (e.g., using semaphores, critical sections, and/or other thread synchronization mechanisms). Alternatively, the processing threads implementing each of methods **400** and **500** may be executed asynchronously with respect to each other. Various operations of each of methods **400** and **500** may be performed in a different order compared to the order shown in FIGS. **4** and **5**. Some blocks may be performed concurrently with other blocks. Some blocks may be optional. Some or all of the blocks of each of methods **400** and **500** may be performed by NOE **105**.

(56) Method **400** may be implemented to protect execution of a neural network from adversarial attacks attempting to identify proprietary information about the neural network. Method **400** may involve a processing device obtaining, at block **410**, a vector of input values (e.g., √{right arrow over (x)}) for a first node of the plurality of nodes of the neural network. The terms “first” and “second” are used herein as mere identifiers and not in any limiting way. For example, a “first node” (and, similarly, a “second node”) may be an arbitrary node of the neural network and should not be understood as a node that is executed before other nodes of the network. Any node of the network, e.g., the first node may be associated with a plurality of parameters that map the vector of input values (e.g., {right arrow over (x)}) onto a target weighted input value (e.g., z) of the first node. For example, parameters of the first node may include weights of the first node (e.g., {right arrow over (w)}), a bias of the first node (e.g., b), and the like. Parameters of the first node may represent a part of proprietary information that is being protected against possible attacks.

(57) At block **420**, the processing device implementing method **400** may perform a first transformation of the plurality of parameters to obtain an expanded plurality of parameters for the first node. At least some of the expanded plurality of parameters may be obfuscation parameters. In some implementations, block **420** may include operations depicted by the pertinent blowout section of FIG. **4**. For example, as shown by block **422**, the first transformation may include obtaining an expanded weight matrix (e.g., $\hat{W}$) that may include the one or more (actual) weights for the first node weight (e.g., {right arrow over (w)}) and a plurality of obfuscation (dummy) weights. Additionally, as shown by block **424**, the first transformation may include obtaining an expanded bias vector that includes the (actual) bias value for the first node (e.g., b) and a plurality of obfuscation biases (e.g., {right arrow over (B)}).

(58) At block **430**, method **400** may include determining, based on the vector of input values (e.g., and the expanded plurality of parameters (e.g., $\hat{W}$ and {right arrow over (B)}), one or more weighted input values for the first node. For example, in some implementations, the one or more weighted input values may be $\hat{W}$.Math.{right arrow over (x)}.sup.T+{right arrow over (B)}. In some implementations, block **430** may include operations depicted by the pertinent blowout section of FIG. **4**. For example, as shown by block **432**, method **400** may include performing a first masking transformation to obtain a masked weight matrix from the expanded weight matrix. In one specific non-limiting example, when linear masking is used, NOE **105** may multiply the expanded weight matrix (e.g., $\hat{W}$) by

a masking matrix (e.g., {circumflex over (M)}), to obtain a masked weight matrix (e.g., $\hat{W}$.Math.{circumflex over (M)}). At block **434**, method **400** may continue with performing a second masking transformation to obtain a masked bias vector from the expanded bias vector. In one specific non-limiting example, when linear masking is used, NOE **105** may multiply the expanded bias vector (e.g., {right arrow over (B)}) by the masking matrix to obtain a masked bias vector (e.g., $\hat{W}$.Math.{right arrow over (B)}). At block **436**, method **400** may further include adding the masked bias vector (e.g., $\hat{W}$.Math.{right arrow over (B)}) to a product of the masked weight matrix (e.g., $\hat{W}$.Math.{circumflex over (M)}) and the vector of input values (e.g., {right arrow over (x)}) e.g., computing {right arrow over (Z)}.sub.M=$\hat{W}$.Math.{circumflex over (M)}.Math.{right arrow over (x)}.sup.T+$\hat{W}$.Math.{right arrow over (B)}. The determined one or more weighted input values (e.g., vector {right arrow over (Z)}.sub.M) may be different from a target weighed input value (e.g., z), but may include information that is sufficient to recover the target weighted input value. More specifically, the target weighted input value may be obtainable from the one or more weighted input values using a second transformation, e.g., using multiplication of {right arrow over (Z)}.sub.M by the unmasking vector: z={right arrow over (U)}.Math.{right arrow over (Z)}.sub.M. (Although the second transformation does not have to be performed explicitly, and may be instead composed with the activation function, as described below.)

(59) At block **440**, method **400** may continue with the processing device determining a composite activation function formed by the activation function and the second transformation. For example, a composite activation function may be (f°{right arrow over (U)})({right arrow over (Z)}.sub.M), where f represents the activation function and {right arrow over (U)} represents the second transformation that recovers z from {right arrow over (Z)}.sub.M. At block **450**, method **400** may continue with the processing device applying the composite activation function, (f°{right arrow over (U)}), to the one or more weighted input values (e.g., vector {right arrow over (Z)}.sub.M) for the first node to obtain an output value for the first node. In some implementations, the output value may be the actual target output value y that equal to a value of the activation function applied to the target weighted input value (e.g., y=f(z)). In some implementations, however, the output may be representative of the target output value f(z), but may be different from the target output value f(z), e.g., be an obfuscated target output value.

(60) At optional block **460**, method **400** may continue with the processing device updating at least one of the first masking transformation or the second masking transformation. In one non-limiting example, where linear masking is used, the processing device may obtain an updated masked weight matrix (e.g., {circumflex over (M)}′.Math.$\hat{W}$) by multiplying the masked weight matrix (e.g., $\hat{W}$) by a second masking matrix (e.g., {circumflex over (M)}′). In some implementations, the second masking matrix may be selected randomly. The processing device may further obtain an updated masked bias vector (e.g., {circumflex over (M)}′.Math.{circumflex over (B)}) by multiplying the masked bias vector by the second masking matrix.

(61) FIG. **5** depicts a flow diagram of an example method **500** of protecting neural network operations using activation function obfuscation, in accordance with one or more aspects of the present disclosure. Method **500** may be implemented by CPU **120**, GPU **122**, or some other processing device (an arithmetic logic unit, an FPGA, and the like, or any processing logic, hardware or software or a combination thereof). At block **510**, processing device implementing method **500** may determine, based on parameters (e.g., weights and biases) of a first node of a neural network, a weighted input (e.g., z or {right arrow over (Z)}.sub.M) into an activation function (e.g., f(z)) for the first node. In some implementations, the weighted input into the first node may be obtained using a plurality of masked weights of the first node, and other obfuscation techniques described in conjunction with method **400** of FIG. **4**.

(62) At block **520**, method **500** may continue with the processing device selecting an obfuscation function (e.g., g(f)) for the first node. In some implementations, the obfuscation function may be an invertible function. At block **530**, method **500** may continue with the processing device determining a first composite activation function (e.g., (g°f)(z)) for the first node, wherein the composite activation function is formed by the activation function for the first node (e.g., f(z)) and the obfuscation function for the first node (e.g., g(f)). In some implementations, block **530** may include operations depicted by the pertinent blowout section of FIG. **5**. For example, the first composite activation function may be one of a plurality of composite activation functions defined for the first node, each of the plurality of composite activation functions (e.g., {g.sub.j°f.sub.j}) being based on a respective activation function of a plurality of activation functions (e.g., {f.sub.j}) for the first node and a respective obfuscation function of a plurality of obfuscation functions (e.g., {g.sub.j}) for the first node.

(63) At block **540**, method **500** may continue with the processing device applying the first composite activation function to the weighted input (e.g., z or {right arrow over (Z)}.sub.M) to compute an obfuscated output (e.g., O) of the first node. In some implementations, block **540** may include operations depicted by the pertinent blowout section of FIG. **5**. For example, at block **542**, processing device implementing method **500** may apply each of the plurality of composite activation functions (e.g., {g.sub.j°f.sub.j}) to the weighted input to compute a respective obfuscated output (e.g., O.sub.j) of a plurality of obfuscated outputs (e.g., {O.sub.j} or, equivalently, {right arrow over (O)}) of the first node. At block **544**, processing device implementing method **500** may mask the plurality of obfuscated outputs (e.g., apply a masking matrix to the obfuscated outputs: {right arrow over (O)}.sub.masked={circumflex over (M)}.Math.{right arrow over (O)}).

(64) At block **550**, method **500** may continue with the processing device providing an obfuscated output (e.g., O) of the first node to a second node of the plurality of nodes of the neural network. In some implementations, block **550** may include operations depicted by the pertinent blowout section of FIG. **5**. For example, in those implementations where multiple obfuscated outputs (e.g., {O.sub.j}) are generated, the processing device implementing method **500** may, at block **552** provide each of the plurality of obfuscated outputs {O.sub.j} to the second node.
(65) At block **560**, method **500** may continue with the processing device determining a weighted input into an activation function of the second node. For example, the processing device may apply, to the provided obfuscated output (e.g., O) of the first node, a weight of the second node (e.g., {w°g.sup.−1}) composite with a de-obfuscation function (e.g., g.sup.−1). In some implementations, the weighted input into the activation function of the second node may be obtained using a plurality of masked weights of the second node, e.g., as described in conjunction with method **400** of FIG. **4**. In some implementations, block **560** may include operations depicted by the pertinent blowout section of FIG. **5**. For example, in those implementations where obfuscated outputs are additionally masked (at block **544**), the processing device performing method **500** may unmask the masked plurality of obfuscated outputs (e.g., using an unmasking vector {right arrow over (U)} as described in conjunction with FIG. **3**B). In some implementations, the unmasking may be composite with determining a weighted input into the second node. In some implementations, the unmasking may further be composite with one or more de-obfuscation functions.
(66) FIGS. **6**A-E illustrate example operations **600** of protection of neural network architecture by obfuscation using dummy operations, in accordance with one or more aspects of the present disclosure. Various implementations illustrated enable to obfuscate the actual architecture of a neural network by performing operations that are inconsequential (do not affect the outcome of the neural network execution) while at the same time presenting a potential attacker with a wider range of operations to track and analyze. Additionally, the inconsequential operations can further be (at regular or random times) changed in any manner that keeps such operations inconsequential, further hindering collection of meaningful statistics for the attacker. In some implementations, inconsequential operations may extend to an entire node ("dummy node") that does not change the flow of relevant (consequential) data. In some implementations, inconsequential operations may extend to an entire layer of dummy nodes. Dummy operations, dummy nodes, and dummy layers may not only make it more difficult for an attacker to identify parameters of nodes, but also obfuscate the topology (number of nodes, layers, edges) of the neural network.
(67) FIG. **6**A illustrates an example implementation of a constant-output node that may be used for obfuscation of neural network operations. A constant-output node **602** may be configured to output a constant value (e.g., y.sub.0) regardless of the input values (e.g., {x.sub.i}). For example, a constant-output node may have an activation function, e.g., a Heaviside step function composite with an exponential function, Θ(exp(Σ.sub.iw.sub.ix.sub.i)), or composite with a sigmoid function, or any other activation function or a combination of activation functions that outputs a constant value y.sub.0 (indicated by the dashed arrow). The value y.sub.0 may be positive, negative, or zero. Because inputs {x.sub.i} do not affect value y.sub.0, the inputs may be outputs of real nodes or dummy nodes, may be obfuscated or unobfuscated, may be inputs from any layer of the neural network, from multiple layers, and so on. Constant output y.sub.0 may be provided to one or more nodes, such as a constant-adjusting node **604** that are configured to handle such constant outputs in a way that does not affect the actual flow of data within the network. For example, constant-adjusting node **604** may weigh the output y.sub.0 from constant-output node **602** with a weight w.sub.0 and also adjust a bias value from a desired (target) value b.sub.T.fwdarw.b to the value b=b.sub.T−w.sub.0y.sub.0. Upon such an adjustment, constant-adjusting node **604** may be capable of handling outputs from other nodes (solid arrows) without affecting actual computations performed by the neural network (e.g., by constant-adjusting node **604** and various other downstream nodes). In some implementations, e.g., where a constant-output node outputs zero value y.sub.0=0 (zero-output node), no adjustment needs to be performed by constant-adjusting node **604** or other downstream nodes. In some implementations, the constant-output character of the node may be obfuscated using multiple output values produced by multiple activation functions, as disclosed above in conjunction with FIG. **3**B.
(68) FIG. **6**B illustrates an example implementation of a pass-through node that may be used for obfuscation of neural network operations. A pass-through node **610** may have an input (or a plurality of inputs) x.sub.1 that is passed-through without modification even though another input (or a plurality of inputs) x.sub.2 is being processed by the same node. In some implementations, output of pass-through node **610** is independent of a specific value (or values) x.sub.2. For example, operations of pass-through node **610** may include weighting input values using at least one non-linear weight function, e.g., z=w.sub.1(x.sub.1)+w.sub.2(x.sub.2)+b. The weight function w.sub.2(x.sub.2) for the input x.sub.2 may be an obfuscation function that performs computations that ultimately do not change the weighted input z into the node but involve a sequence of operations (multiplications, additions, exponentiations, etc.) whose overall effect is null. For example, a non-linear weight function may be w.sub.2(x)=ln(e.sup.x+1)−x−ln(1+e.sup.−x). Determining the weighted input z=w.sub.1(x.sub.1)+w.sub.2(x.sub.2)+b may involve mixing (scrambling) computations of different steps (terms) of the weight function w.sub.2(x.sub.2) with steps in computation of w.sub.1(x.sub.1)+b, for additional protection against adversarial attacks. The weighted input z=w.sub.1(x.sub.1)+b may then be processed by an activation

function f(z) of pass-through node **610** that restores the input x.sub.1, e.g., f(w.sub.1(x.sub.1)+b)=x.sub.1. The pass-through character of such a node may be further obfuscated with various techniques described in relation to FIGS. **2**A-**2**B and **3**A-**3**B.

(69) FIG. **6**C illustrates an example implementation of a pass-through cluster of nodes that may be used for obfuscation of neural network operations. Depicted schematically is a cluster that includes three nodes **621**, **622**, and **623**, but any number of nodes may be arranged in a pass-through cluster. Input nodes **621** and **622** may have inputs (or a plurality of inputs) x.sub.1 that are passed-through while another input (or a plurality of inputs) x.sub.2 is processed for the purpose of obfuscation. First node **621** may output a first function of x.sub.1 and x.sub.2 (shown, for illustration purposes only is an even combination of x.sub.1 and x.sub.2) whereas second node **622** may output a second function of x.sub.1 and x.sub.2 (shown, for illustration, is an odd combination of x.sub.1 and x.sub.2). Subsequently, when third node **623** processes the first function and the second function, third node **623** may output a value, e.g., y(x.sub.1) that is determined by first input x.sub.1 only, but not by second input x.sub.2. In some implementations, the output of third node **623** may be the same as first input x.sub.1 (a pure pass-through cluster). In some implementations, the output of third node **623** may be masked, e.g., y(x.sub.1)=m.sub.1.Math.x.sub.1+m.sub.2 (a masking pass-through cluster). In some implementations, the cluster of nodes may additionally perform substantive computations meaning such that y(x.sub.1) is not equal to x.sub.1 or its masked representation. Outputs of nodes **621** and **623** may additionally be input into one or more dummy nodes, such as various zero-output and constant output-nodes to make it more difficult for a potential attacker to identify the pass-through character of the cluster.

(70) FIG. **6**D illustrates an example implementation of dummy (inconsequential) outputs from a node that may be used for obfuscation of neural network operations. Shown is a node **630** that outputs multiple output values {O.sub.i} into nodes **631** and **632**. Node **631** may use output values {O.sub.i} for real computations whereas node **632** may use the same output values for dummy computations. Accordingly, outputs {O.sub.i} are consequential inputs **633** when input into node **631** (the inputs affect the output of the neural network execution) but are inconsequential inputs **634** when input into node **632** (do not affect the output of the neural network execution). In some implementations, producing outputs that may be used as both consequential and inconsequential inputs into other nodes may include receiving node **630** inputs {x.sub.i} and determining node **630** weighted input value z (which, in some implementations, may be obfuscated using techniques disclosed above). Node **630** may deploy multiple activation functions {f.sub.i} with functions f.sub.2, f.sub.3, . . . serving to obfuscate an actual activation function f.sub.1 of node **630**. To further obfuscate which one of the activation functions {f.sub.i} is the actual activation function, output values {O.sub.i} may be obtained using a transformation of {f.sub.i}. In some implementations, a linear obfuscating transformation may be deployed, e.g., using a masking matrix {circumflex over (M)} to transform the vector of values {right arrow over (f)}(z) (having components equal to various functions of the set {f.sub.i}) into a vector of output values {right arrow over (O)}:{right arrow over (O)}={circumflex over (M)}.Math.{right arrow over (f)}(z). Vector {right arrow over (O)} received by node **631** may be processed by a set (vector) of weights {right arrow over (w)}.sub.1 to obtain a weighted node **630** input, e.g., {right arrow over (w)}.sub.1.Math.{circumflex over (M)}.Math.{right arrow over (f)}(z). For this weighted input to be equal to the actual output f.sub.1(z) of node **630**, weights {right arrow over (w)}.sub.1 of node **631** may be selected such that {right arrow over (w)}.sub.1.Math.{circumflex over (M)} is the vector with a first component equal to 1 and all other components equal to 0. For example, the weights may be {right arrow over (w)}.sub.1=(1, 0, 0, . . .).Math.{circumflex over (M)}.sup.−1. Accordingly, node **631** may be capable of identifying (and processing) correct inputs among consequential inputs **633**. Additionally, node **631** may receive other inputs {F.sub.i} from other nodes and process inputs {F.sub.i} together with the identified input f.sub.1(z) of node **630**. Consequently, output of node **631** may be a function of inputs f.sub.1 and {F.sub.i}: y.sub.1(f.sub.1, {F.sub.i}):

(71) When inconsequential inputs **634** (which may be the same values {O.sub.i} or, equivalently, the same vector {right arrow over (O)}) are provided to node **632**, node **632** may apply a different set (vector) of weights {right arrow over (w)}.sub.2 to obtain a weighted node **630** input, e.g., {right arrow over (w)}.sub.2.Math.{circumflex over (M)}.Math.{right arrow over (f)}(z)≡{right arrow over (v)}. {right arrow over (f)}(z), where {right arrow over (v)}={right arrow over (w)}.sub.2.Math.{circumflex over (M)}. To ensure that inputs **634** are inconsequential, weights {right arrow over (w)}.sub.2 may be selected to make the product {right arrow over (v)}.Math.{right arrow over (f)}(z) vanish for all values z. For example, component v.sub.1 may be zero: v.sub.1=0. Vanishing of the remaining inputs may be achieved by selecting the obfuscation functions to be linearly-dependent, e.g., such that there exists a relationship,

(72) $f_2(z) = -\frac{1}{v_2}(v_3 f_3(z) +$ .Math.)

Under such a condition (or similar conditions), and provided that the weights {right arrow over (w)}.sub.2 and the masking matrix {circumflex over (M)} are chosen to obey the condition, {right arrow over (v)}={right arrow over (w)}.sub.2.Math.{circumflex over (M)}, output of node **632** may be independent of inconsequential inputs **634**. Additionally, mode **631** may receive other inputs {G.sub.i} from other nodes and process inputs {G.sub.i} together

with inconsequential inputs **634**. As a result, output of node **632** may be a function of inputs {G.sub.i}:y.sub.2({G.sub.i}). In various implementations, the number of obfuscating functions f.sub.2(z), f.sub.3(z), . . . may differ. In some implementations, only two functions f.sub.2(z) and f.sub.3(z) may be used. Even though the two functions f.sub.2(z) and f.sub.3(z) may essentially amount to the same function (up to multiplication by a constant factor), NOE **105** may vary how the respective functions are computed so that the similarity of the two (or more) functions is obfuscated to protect against an adversarial attack. For example, one of the functions may be
f.sub.2(z)=(z.sup.2+1).sup.2,
whereas the other function may be
f.sub.3(z)=½.Math.(2.Math.z.sup.2−1).sup.2+3/2+6.Math.z.sup.2.
Even though f.sub.3 (z)=2.Math.f.sub.2(z), this fact may not be easily determinable by an attacker, especially if operations of computing different obfuscation functions are scrambled (e.g., with the processing device computing different functions not fully sequentially, but rather in batches, with batches of computations related to various functions interspersed with batches of computations related to other functions. More than two functions may be used for added protection. In some implementations, functions (both in number and specific form) may be changed at certain time intervals, periodically, or after a certain number of NN operations.

(73) Although linear transformations from {f.sub.i} to output values {O.sub.i} have been used (for conciseness and ease of notations) to illustrate operations performed in conjunction with FIG. **6**D, various non-linear (invertible) transformations may be used for this purpose instead.

(74) FIG. **6**E illustrates an example implementation of node clusters having cancelling outputs that may be used for obfuscation of neural network operations. Depicted schematically is a cluster that includes four nodes **641**, **642**, **643**, and **644**, but any number of nodes may be similarly arranged. Input nodes **641**, **642**, and **643** may have different sets (vectors) of inputs {right arrow over (x)}.sub.1, {right arrow over (x)}.sub.2 and {right arrow over (x)}.sub.3. In some implementations, the vectors of inputs may be arbitrary and independent of each other. Each of the nodes may output one or more activation functions, e.g., node **641** may output activation functions f.sub.1 and f'.sub.1, node **642** may output activation functions f.sub.2 and f'.sub.2, and node **643** may output activation functions f.sub.3 and f'.sub.3. Activation functions in respective pairs may be different from each other but may be correlated, e.g., function f.sub.1 may be the sigmoid function f.sub.1=S(z.sub.1) of a (weighted) input into node **641**, whereas respective function f'.sub.1 may be f'.sub.1=S(−z.sub.1). For any value z.sub.1, the two outputs of node **641** may add up to a constant, e.g., f.sub.1+f'.sub.1=1 in this example. Similar conditions may be satisfied by other pairs of outputs, e.g., functions f.sub.2 and f'.sub.2 may be the Heaviside functions, f.sub.2=Θ(z.sub.2) and f'.sub.2=Θ(−z.sub.2), and so on. Node **644** may combine activation functions from the three input nodes and output a constant value. For example, a weighted input value into an activation function of node **644** may be z.sub.4=f.sub.1+f'.sub.3+f'.sub.1+f.sub.2+f.sub.3+f'.sub.2. The order of summation may be mixed to obfuscate the fact that pairs of respective outputs add up to a constant (e.g., 3). An arbitrary activation function f.sub.4 of node **644** may then be applied to a (constant) weighted input value to produce a constant output, f.sub.4(z.sub.4). In some implementations, a bias value of node **644** may be chosen to ensure that the output of the cancelling cluster, f.sub.4(z.sub.4+b), is always zero, although any other constant value may be output. Node **644** may have additional inputs (shown by arrows) whose contributions into the output of node **644** do not cancel. Accordingly, node **644** may serve both as an obfuscating node and an active node that contributes to NN operations.

(75) Various implementations of dummy operations disclosed in conjunction with FIGS. **6**A-E may be combined with each other. For example, a pass-through node of FIG. **6**B may be combined with a canceling node of FIG. **6**E. Any and all operations disclosed in conjunction with FIGS. **6**A-E may be obfuscated using any of the techniques disclosed in relation to FIGS. **2**A-B, FIGS. **3**A-B, FIG. **4**, and FIG. **5**, including (but not limited to) obfuscation by weight and bias expansion, obfuscation by masking, activation function obfuscation, by splitting of activation functions, and the like.

(76) Various implementations of dummy operations disclosed in conjunction with FIGS. **6**A-E are intended for illustration purposes only. A practically unlimited number of other implementations, variations, and combinations of dummy operations and dummy nodes are possible.

(77) In some implementations, multiple nodes having inconsequential inputs, which may include pass-through nodes, constant-output nodes, canceling nodes, or any other nodes described in relation to FIGS. **6**A-E may be joined into multi-node clusters. In some implementations, an entire dummy layer of pass-through nodes may be formed to obfuscate a total number of layers in a NN and other features of the architecture of the NN. In some implementations, dummy nodes may be used to obfuscate a number of nodes in various layers of the NN. In some implementations, dummy nodes may be used to mask a nature of the NN, e.g. to present (to an attacker) a convolutional network (or a convolutional sub-network of a larger network) as a deconvolutional NN or a NN of fully-connected layers. For example, a portion of a layer may be made of dummy nodes (e.g. constant-output nodes), with the nodes of the next layer connected to the dummy nodes adjusting or canceling the inputs from the dummy nodes. The pass-through nature of dummy nodes and dummy layers may be obfuscated with some or all obfuscation techniques disclosed above in relation to FIGS. **2**A-B, FIGS. **3**A-B, FIG. **4**, and FIG. **5**.

(78) FIG. **7** depicts a flow diagram of an example method **700** of protecting neural network operations using obfuscation of neural network architecture, in accordance with one or more aspects of the present disclosure. Method **700** and/or each of its individual functions, routines, subroutines, or operations may be performed by one or more processing units of the computing system implementing the methods, e.g., CPU **120** and/or GPU **122**. In certain implementations, method **700** may be performed by a single processing thread. Alternatively, method **700** may be performed by two or more processing threads, each thread executing one or more individual functions, routines, subroutines, or operations of the method. In an illustrative example, the processing threads implementing method **700** may be synchronized (e.g., using semaphores, critical sections, and/or other thread synchronization mechanisms). Alternatively, the processing threads implementing method **700** may be executed asynchronously with respect to each other. Various operations of method **700** may be performed in a different order compared to the order shown in FIG. **7**. Some blocks may be performed concurrently with other blocks. Some blocks may be optional. Some or all of the blocks of each of method **700** may be performed by NOE **105**.
(79) Method **700** may be implemented to protect execution of a neural network (NN) model from adversarial attacks attempting to identify proprietary information about the NN model. Multiple NN models **702** may be protected using method **700**. Method **700** may involve a processing device identifying, at block **710**, a specific neural (NN) model to be protected. The identified NN model may be configured to generate, based on an input into the NN model, a target output of the NN model. For example, the input into the NN model may be a digital representation of an image of a document and the target output may be a recognized text contained in the document. In another example, the input may be a digital representation of a human voice and the target output may be an identified owner of the voice. The identified model may include multiple layers of nodes, each layer having multiple nodes. Each node may be associated with multiple weights (to weight input values from various input nodes), a bias, and one or more activation functions, and may output computed values to any number of downstream nodes.
(80) At block **720**, method **700** may continue with the processing device obtaining a modified NN model **730**. Modification of the NN model may include obfuscation (e.g., by expansion, masking, randomization, etc.) of the existing operations of the NN model, such as configuring existing nodes to modify weights, biases, activation functions, generating additional inputs/outputs into/from various nodes, and so on. Modifications of the NN model may be performed in such a way that the modified NN model **730** is configured to output the same target output(s) based on the same input(s) as the identified NN model is configured to output. Correspondingly, the modified NN model **730** may function similarly to the unmodified NN model, but operations of the modified NN model **730** may be protected against adversarial attacks.
(81) In some implementations, obtaining the modified model **730** may include configuring a first node to provide one or more output values to a second node, wherein the one or more output values provided by the first node may include one or more inconsequential input values into the second node (block **740**). For example, with reference to FIG. **6**D, first node **630** may provide inconsequential input values **634** to second node **632**. The inconsequential input values may have no effect on the output of the modified neural network model **730**. For example, the second node (e.g., node **634**) may be configured to perform one or more operations to compensate for the inconsequential input values. For example, the processing device performing method **700** may change weights and biases of the second node to cancel out the inconsequential input values. In some implementations, compensation for the inconsequential input values may be performed not by operations of the second node, but by other nodes within a downstream (relative to the second node) portion of the modified NN model **730**. Downstream portion may include the second node and any other nodes whose computations are performed after computations of the second node and are connected to the second node by one or more edges. Downstream portion may be configured to compensate for the one or more inconsequential input values into the second node at some point prior to (or concurrently with) producing the target output of the modified NN model. The terms “first” and “second” are used herein as mere identifiers and may identify arbitrary nodes of the NN model.
(82) In some implementations, the modified NN model **730** may include a variety of operations depicted by blocks **740**-**772** showing a number of illustrative non-limiting examples. In some implementations, one or more inconsequential input values into the second node may include at least one constant input value (block **740**) output by the first node, e.g., node **602** of FIG. **6**A. The first node may output the constant value for multiple inputs into the first node. In some implementations, the first node may output the same constant value for all inputs into the first node. In some implementations, the constant input value may be zero. Accordingly, compensation for the constant input value into the second node may be performed automatically when the constant zero input is multiplied by a respective weight of the second node that weights the input values provided by the first node. In some implementations, the constant input value may be non-zero and the second node may be configured to compensate for the constant input value. For example, the second node may be constant-adjusting node **604** described in conjunction with FIG. **6**A.
(83) In some implementations, the second node may be a dummy node and may perform only computations (dummy computations) that do not affect the ultimate output of the modified NN model. In some implementations, however, the second node may perform both dummy computations and actual (consequential) computations. For example, the

second node may be node **632** of FIG. **6**D, which may receive additional input values (e.g., {G.sub.i}) from one or more additional nodes of the modified NN model (block **750**). The additional input values may be consequential input values for the second node and may be a part of the actual computations carried out by the NN model.

(84) In some implementations, the modified NN model may include a third node (block **760**). The first node may provide the one or more input values into the third node. For example, the third node may be node **631** of FIG. **6**D and may receive the same inputs from the first node (node **630**) as received by the second node **632**. The input values received by the third node may be consequential input values for the third node and may be a part of the actual computations carried out by the NN model. In some implementations, the one or more output values of the first node, provided as input values to the third node, may be obfuscated (block **762**) using any of the techniques described in the present disclosure. Subsequently, operations of the third node may include application of a de-obfuscation transformation to de-obfuscate the one or more values output by the first node.

(85) In some implementations, the second node may be a pass-through node (block **770**) configured to receive one or more additional input values from a third node. An output of the second node may be based on the one or more additional input values and may be independent of the one or more inconsequential input values into the second node. For example, a pass-through node may be node **610** illustrated in FIG. **6**B. The inconsequential input (e.g., x.sub.2) may be received from the first node (along the edge indicated by the dashed arrow) and the additional input (e.g., x.sub.1) may be received from the third node (along the edge indicated by the solid arrow). The additional input values may be consequential input values for the second node and may be a part of the actual computations carried out by the NN model. In some implementations, the output of the second node may be the same as the one or more additional input values received from the third node, such as y=x.sub.1 (a pure pass-through node). In some implementations, the output of the second node may include an obfuscated representation (block **772**) of the one or more additional input values, e.g., a masked representation, y=m.sub.1.Math.x.sub.1+m.sub.2 (a masking pass-through node). Various downstream nodes receiving outputs from the second node may then adjust to compensate for the masking.

(86) In some implementations, the second (pass-through) node may be one of multiple pass-through nodes (block **774**) added to the modified NN model. Pass-through nodes may be randomly interspersed with nodes performing actual computations of the NN model. In some implementations, pass-through nodes may constitute portions of various (including multiple) layers of nodes added to the modified NN model. In some implementations, pass-through nodes of the modified NN model may form one or more full layers of pass-through nodes to obfuscate topology of the modified NN model.

(87) FIG. **8** depicts a block diagram of an example computer system **800** operating in accordance with one or more aspects of the present disclosure. In various illustrative examples, computer system **800** may represent the computer device **102**, illustrated in FIGS. **1**A-B.

(88) Example computer system **800** may be connected to other computer systems in a LAN, an intranet, an extranet, and/or the Internet. Computer system **800** may operate in the capacity of a server in a client-server network environment. Computer system **800** may be a personal computer (PC), a set-top box (STB), a server, a network router, switch or bridge, or any device capable of executing a set of instructions (sequential or otherwise) that specify actions to be taken by that device. Further, while only a single example computer system is illustrated, the term "computer" shall also be taken to include any collection of computers that individually or jointly execute a set (or multiple sets) of instructions to perform any one or more of the methods discussed herein.

(89) Example computer system **800** may include a processing device **802** (also referred to as a processor or CPU), which may include processing logic **827**, a main memory **804** (e.g., read-only memory (ROM), flash memory, dynamic random access memory (DRAM) such as synchronous DRAM (SDRAM), etc.), a static memory **806** (e.g., flash memory, static random access memory (SRAM), etc.), and a secondary memory (e.g., a data storage device **818**), which may communicate with each other via a bus **830**.

(90) Processing device **802** represents one or more general-purpose processing devices such as a microprocessor, central processing unit, or the like. More particularly, processing device **802** may be a complex instruction set computing (CISC) microprocessor, reduced instruction set computing (RISC) microprocessor, very long instruction word (VLIW) microprocessor, processor implementing other instruction sets, or processors implementing a combination of instruction sets. Processing device **802** may also be one or more special-purpose processing devices such as an application specific integrated circuit (ASIC), a field programmable gate array (FPGA), a digital signal processor (DSP), network processor, or the like. In accordance with one or more aspects of the present disclosure, processing device **802** may be configured to execute instructions implementing method **400** of protecting neural network operations using obfuscation of weights and biases, method **500** of protecting neural network operations using activation function obfuscation, and method **700** of protecting neural network operations using obfuscation of neural network architecture.

(91) Example computer system **800** may further comprise a network interface device **808**, which may be communicatively coupled to a network **820**. Example computer system **800** may further comprise a video display **810** (e.g., a liquid crystal display (LCD), a touch screen, or a cathode ray tube (CRT)), an alphanumeric input device

812 (e.g., a keyboard), a cursor control device **814** (e.g., a mouse), and an acoustic signal generation device **816** (e.g., a speaker).

(92) Data storage device **818** may include a computer-readable storage medium (or, more specifically, a non-transitory computer-readable storage medium) **828** on which is stored one or more sets of executable instructions **822**. In accordance with one or more aspects of the present disclosure, executable instructions **822** may comprise executable instructions implementing method **400** of protecting neural network operations using obfuscation of weights and biases, method **500** of protecting neural network operations using activation function obfuscation, and method **700** of protecting neural network operations using obfuscation of neural network architecture.

(93) Executable instructions **822** may also reside, completely or at least partially, within main memory **804** and/or within processing device **802** during execution thereof by example computer system **800**, main memory **804** and processing device **802** also constituting computer-readable storage media. Executable instructions **822** may further be transmitted or received over a network via network interface device **808**.

(94) While the computer-readable storage medium **828** is shown in FIG. **8** as a single medium, the term "computer-readable storage medium" should be taken to include a single medium or multiple media (e.g., a centralized or distributed database, and/or associated caches and servers) that store the one or more sets of operating instructions. The term "computer-readable storage medium" shall also be taken to include any medium that is capable of storing or encoding a set of instructions for execution by the machine that cause the machine to perform any one or more of the methods described herein. The term "computer-readable storage medium" shall accordingly be taken to include, but not be limited to, solid-state memories, and optical and magnetic media.

(95) Some portions of the detailed descriptions above are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

(96) It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise, as apparent from the following discussion, it is appreciated that throughout the description, discussions utilizing terms such as "identifying," "determining," "storing," "adjusting," "causing," "returning," "comparing," "creating," "stopping," "loading," "copying," "throwing," "replacing," "performing," or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

(97) Examples of the present disclosure also relate to an apparatus for performing the methods described herein. This apparatus may be specially constructed for the required purposes, or it may be a general purpose computer system selectively programmed by a computer program stored in the computer system. Such a computer program may be stored in a computer readable storage medium, such as, but not limited to, any type of disk including optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), random access memories (RAMs), EPROMs, EEPROMs, magnetic disk storage media, optical storage media, flash memory devices, other type of machine-accessible storage media, or any type of media suitable for storing electronic instructions, each coupled to a computer system bus.

(98) The methods and displays presented herein are not inherently related to any particular computer or other apparatus. Various general purpose systems may be used with programs in accordance with the teachings herein, or it may prove convenient to construct a more specialized apparatus to perform the required method steps. The required structure for a variety of these systems will appear as set forth in the description below. In addition, the scope of the present disclosure is not limited to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the present disclosure.

(99) It is to be understood that the above description is intended to be illustrative, and not restrictive. Many other implementation examples will be apparent to those of skill in the art upon reading and understanding the above description. Although the present disclosure describes specific examples, it will be recognized that the systems and methods of the present disclosure are not limited to the examples described herein, but may be practiced with modifications within the scope of the appended claims. Accordingly, the specification and drawings are to be regarded in an illustrative sense rather than a restrictive sense. The scope of the present disclosure should, therefore,

be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

Claims

1. A computer-implemented method to execute using at least one or more hardware processors, a neural network model that has a plurality of nodes, the method comprising: obtaining, using the one or more hardware processors, a vector of input values for a first node of the plurality of nodes, wherein the first node is associated with a plurality of parameters that map the vector of input values to a target weighted input value of the first node, the plurality of parameters comprising one or more weights for the first node; performing, using the one or more hardware processors, a first transformation of the plurality of parameters to obtain an expanded plurality of parameters for the first node, wherein performing the first transformation comprises: obtaining an expanded weight matrix comprising the one or more weights for the first node and a plurality of obfuscation weights; and determining, using the one or more hardware processors, based on the vector of input values and the expanded plurality of parameters, one or more weighted input values for the first node, wherein the target weighted input value is obtainable from the one or more weighted input values using a second transformation, and wherein determining the one or more weighted input values for the first node is based, at least in part, on the expanded weight matrix.
2. The method of claim 1, wherein the plurality of parameters further comprises a bias value for the first node, and wherein performing the first transformation further comprises: obtaining an expanded bias vector comprising the bias value for the first node and a plurality of obfuscation biases; and wherein determining the one or more weighted input values for the first node comprises: performing a first masking transformation to obtain a masked weight matrix from the expanded weight matrix; and performing a second masking transformation to obtain a masked bias vector from the expanded bias vector.
3. The method of claim 2, wherein performing the first masking transformation comprises multiplying the expanded weight matrix by a masking matrix, and wherein performing the second masking transformation comprises multiplying the expanded bias vector by the masking matrix.
4. The method of claim 3, wherein determining the one or more weighted input values for the first node further comprises: adding the masked bias vector to a product of the masked weight matrix and the vector of input values.
5. The method of claim 2, further comprising: updating at least one of the first masking transformation or the second masking transformation.
6. The method of claim 1, wherein the first node is further associated with an activation function, the method further comprising: determining, using the one or more hardware processors, a composite activation function formed by the activation function and the second transformation.
7. The method of claim 6, further comprising: applying the composite activation function to the one or more weighted input values for the first node to obtain an output value for the first node.
8. The method of claim 7, wherein the obtained output value is representative of a target output value, wherein the target output value is equal to a value of the activation function applied to the target weighted input value.
9. The method of claim 7, wherein the activation function is a function that is discontinuous or a function that has a discontinuous derivative, and wherein applying the composite activation function further comprises: obfuscating a location of a point of discontinuity.
10. The method of claim 9, wherein the composite activation function is a step function and wherein applying the composite activation function further comprises shifting the step function into a same-sign domain of output values.
11. The method of claim 7, wherein the activation function is a sigmoid function, and wherein the one or more weighted input values input into the composite activation function are additively masked and an output value of the composite activation function is multiplicatively masked.
12. A computer-implemented method performed using one or more hardware processors, the method comprising: identifying a neural network (NN) model to be protected against adversarial attacks, wherein the NN model includes a plurality of nodes and is configured to generate, based on an input into the NN model, a target output of the NN model; and obtaining a modified NN model configured to output the same target output based on the same input, wherein obtaining the modified NN model comprises: obtaining, using the one or more hardware processors, a vector of input values for a first node of the plurality of nodes, wherein the first node is associated with a plurality of parameters that map the vector of input values to a target weighted input value of the first node, the plurality of parameters comprising one or more weights for the first node; performing, using the one or more hardware processors, a first transformation of the plurality of parameters to obtain an expanded plurality of parameters for the first node, wherein performing the first transformation comprises; obtaining an expanded weight matrix comprising the one or more weights for the first node and a plurality of obfuscation weights; and determining, using the one or more hardware processors, based on the vector of input values and the expanded plurality of parameters, one or more weighted input values for the first node, wherein the target weighted input value is obtainable from the one or more

weighted input values using a second transformation, and wherein determining the one or more weighted input values for the first node is based, at least in part, on the expanded weight matrix.

13. A system to execute a neural network model that has a plurality of nodes, the system comprising: a memory device; and a processing device communicatively coupled to the memory device, the processing device to: obtain a vector of input values for a first node of the plurality of nodes, wherein the first node is associated with a plurality of parameters that map the vector of input values to a target weighted input value of the first node, the plurality of parameters comprising one or more weights for the first node; perform a first transformation of the plurality of parameters to obtain an expanded plurality of parameters for the first node, wherein to perform the first transformation, the processing device is to: obtain an expanded weight matrix comprising the one or more weights for the first node and a plurality of obfuscation weights; and determine, based on the vector of input values and the expanded plurality of parameters, one or more weighted input values for the first node, wherein the target weighted input value is obtainable from the one or more weighted input values using a second transformation, and wherein to determine the one or more weighted input values for the first node, the processing device is to use the expanded weight matrix.

14. The system of claim 13, wherein the plurality of parameters further comprises a bias value for the first node, and wherein to perform the first transformation, the processing device is further to: obtain an expanded bias vector comprising the bias value for the first node and a plurality of obfuscation biases; and wherein to determine the one or more weighted input values for the first node, the processing device is to: perform a first masking transformation to obtain a masked weight matrix from the expanded weight matrix; and perform a second masking transformation to obtain a masked bias vector from the expanded bias vector.

15. The system of claim 14, wherein to perform the first masking transformation the processing device is to multiply the expanded weight matrix by a masking matrix, and wherein to perform the second masking transformation the processing device is to multiply the expanded bias vector by the masking matrix.

16. The system of claim 15, wherein to determine the one or more weighted input values for the first node the processing device is further to: add the masked bias vector to a product of the masked weight matrix and the vector of input values.

17. The system of claim 14, wherein the processing device is further to: update at least one of the first masking transformation or the second masking transformation.

18. The system of claim 13, wherein the first node is further associated with an activation function, and wherein the processing device is further to: determine a composite activation function formed by the activation function and the second transformation.

19. The system of claim 18, wherein the processing device is further to: apply the composite activation function to the one or more weighted input values for the first node to obtain an output value for the first node.

20. A non-transitory computer-readable medium storing instructions thereon, wherein the instructions, when executed by a processing device, cause the processing device to: obtain a vector of input values for a first node of a plurality of nodes of a neural network, wherein the first node is associated with a plurality of parameters that map the vector of input values to a target weighted input value of the first node, the plurality of parameters comprising one or more weights for the first node; perform a first transformation of the plurality of parameters to obtain an expanded plurality of parameters for the first node, wherein to perform the first transformation, the processing device is to: obtain an expanded weight matrix comprising the one or more weights for the first node and a plurality of obfuscation weights; and determine, based on the vector of input values and the expanded plurality of parameters, one or more weighted input values for the first node, wherein the target weighted input value is obtainable from the one or more weighted input values using a second transformation, and wherein to determine the one or more weighted input values for the first node, the processing device is to use the expanded weight matrix.