Python-FHEz

Release 0.1.1.r221.0b50b14

GeorgeRaven

Jan 27, 2022

TABLE OF CONTENTS

Python-FHEz is a privacy-preserving *Fully Homomorphic Encryption* (FHE) and deep learning library.

Interactive graph example of an FHE compatible neural network:

Fully Homomorphic Encryption:

Deep Learning:

This library is capable of both fully homomorphically encrypting data and processing encrypted cyphertexts without the private keys, thus completely privately.

This library also supports:

- advanced serialization of cyphertext objects, for transmission over networks using marshmallow
- *Kernel-Masquerade*ing
- *Hadmard Product* cyphertext branching

CHAPTER

ONE

CITE

Either:

```
@online{fhez,
  author = {George Onoufriou},
  title = {Python-FHEz Source Repository},
  year = {2021},
  url = {https://gitlab.com/DeepCypher/Python-FHEz},
}
```

Or if you do not have @online support:

```
@misc{fhez,
  author = {George Onoufriou},
  title = {Python-FHEz Source Repository},
  howpublished = {Github, GitLab},
  year = {2021},
  note = {\url{https://gitlab.com/DeepCypher/Python-FHEz}},
}
```

1.1 Documentation Variations

This documentation is hosted using different branches/ versions, and to different extents depending on the provider, to ensure maximum availability, and specificity so you are never left without documentation.

Note: We use sphinx autodoc to document our API, however because some base libraries are still new and are only really available in dockerland this auto-documentation is not easily implementable on all providers. If you would like to have this auto-documentation please choose a version of the docs with **autodoc**.

The current variations of the documentation are as follows:

Version	Provider	Link to Documentation
Master	RTD	
Stable	RTD	
Staging	RTD	
Dev	RTD	
Staging (Autodoc)	GitLab Pages	

1.2 Source Code Variations

Python-fhez related source code is officially hosted across multiple locations and possibly unofficially hosted in many locations further to this. This page defines where you can officially attain our free and open source software.

The current repositories are as follows:

Status	Provider	Link to Source
Origin	GitLab	src
Mirror	GitHub	src

1.3 The Open Software License 3.0 (OSL-3.0)

This Open Software License (the "License") applies to any original work of authorship (the "Original Work") whose owner (the "Licensor") has placed the following licensing notice adjacent to the copyright notice for the Original Work:

Licensed under the Open Software License version 3.0

1) Grant of Copyright License. Licensor grants You a worldwide, royalty-free, non-exclusive, sublicensable license, for the duration of the copyright, to do the following:

a) to reproduce the Original Work in copies, either alone or as part of a collective work;

b) to translate, adapt, alter, transform, modify, or arrange the Original Work, thereby creating derivative works ("Derivative Works") based upon the Original Work;

c) to distribute or communicate copies of the Original Work and Derivative Works to the public, with the proviso that copies of Original Work or Derivative Works that You distribute or communicate shall be licensed under this Open Software License;

d) to perform the Original Work publicly; and

e) to display the Original Work publicly.

2) Grant of Patent License. Licensor grants You a worldwide, royalty-free, non-exclusive, sublicensable license, under patent claims owned or controlled by the Licensor that are embodied in the Original Work as furnished by the Licensor, for the duration of the patents, to make, use, sell, offer for sale, have made, and import the Original Work and Derivative Works.

3) Grant of Source Code License. The term "Source Code" means the preferred form of the Original Work for making modifications to it and all available documentation describing how to modify the Original Work. Licensor agrees to provide a machine-readable copy of the Source Code of the Original Work along with each copy of the Original Work that Licensor distributes. Licensor reserves the right to satisfy this obligation by placing a machine-readable copy of the Source Code in an information repository reasonably calculated to permit inexpensive and convenient access by You for as long as Licensor continues to distribute the Original Work.

4) Exclusions From License Grant. Neither the names of Licensor, nor the names of any contributors to the Original Work, nor any of their trademarks or service marks, may be used to endorse or promote products derived from this Original Work without express prior permission of the Licensor. Except as expressly stated herein, nothing in this License grants any license to Licensor's trademarks, copyrights, patents, trade secrets or any other intellectual property. No patent license is granted to make, use, sell, offer for sale, have made, or import embodiments of any patent claims other than the licensed claims defined in Section 2. No license is granted to the trademarks of Licensor even if such marks are included in the Original Work. Nothing in this License shall be interpreted to prohibit Licensor from licensing under terms different from this License any Original Work that Licensor otherwise would have a right to license.

5) External Deployment. The term "External Deployment" means the use, distribution, or communication of the Original Work or Derivative Works in any way such that the Original Work or Derivative Works may be used by anyone other than You, whether those works are distributed or communicated to those persons or made available as an application intended for use over a network. As an express condition for the grants of license hereunder, You must treat any External Deployment by You of the Original Work or a Derivative Work as a distribution under section 1(c).

6) Attribution Rights. You must retain, in the Source Code of any Derivative Works that You create, all copyright, patent, or trademark notices from the Source Code of the Original Work, as well as any notices of licensing and any descriptive text identified therein as an "Attribution Notice." You must cause the Source Code for any Derivative Works that You create to carry a prominent Attribution Notice reasonably calculated to inform recipients that You have modified the Original Work.

7) Warranty of Provenance and Disclaimer of Warranty. Licensor warrants that the copyright in and to the Original Work and the patent rights granted herein by Licensor are owned by the Licensor or are sublicensed to You under the terms of this License with the permission of the contributor(s) of those copyrights and patent rights. Except as expressly stated in the immediately preceding sentence, the Original Work is provided under this License on an "AS IS" BASIS and WITHOUT WARRANTY, either express or implied, including, without limitation, the warranties of non-infringement, merchantability or fitness for a particular purpose. THE ENTIRE RISK AS TO THE QUALITY OF THE ORIGINAL WORK IS WITH YOU. This DISCLAIMER OF WARRANTY constitutes an essential part of this License. No license to the Original Work is granted by this License except under this disclaimer.

8) Limitation of Liability. Under no circumstances and under no legal theory, whether in tort (including negligence), contract, or otherwise, shall the Licensor be liable to anyone for any indirect, special, incidental, or consequential damages of any character arising as a result of this License or the use of the Original Work including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses. This limitation of liability shall not apply to the extent applicable law prohibits such limitation.

9) Acceptance and Termination. If, at any time, You expressly assented to this License, that assent indicates your clear and irrevocable acceptance of this License and all of its terms and conditions. If You distribute or communicate copies of the Original Work or a Derivative Work, You must make a reasonable effort under the circumstances to obtain the express assent of recipients to the terms of this License. This License conditions your rights to undertake the activities listed in Section 1, including your right to create Derivative Works based upon the Original Work, and doing so without honoring these terms and conditions is prohibited by copyright law and international treaty. Nothing in this License is intended to affect copyright exceptions and limitations (including "fair use" or "fair dealing"). This License shall terminate immediately and You may no longer exercise any of the rights granted to You by this License upon your failure to honor the conditions in Section 1(c).

10) Termination for Patent Action. This License shall terminate automatically and You may no longer exercise any of the rights granted to You by this License as of the date You commence an action, including a cross-claim or counterclaim, against Licensor or any licensee alleging that the Original Work infringes a patent. This termination provision shall not apply for an action alleging patent infringement by combinations of the Original Work with other software or hardware.

11) Jurisdiction, Venue and Governing Law. Any action or suit relating to this License may be brought only in the courts of a jurisdiction wherein the Licensor resides or in which Licensor conducts its primary business, and under the laws of that jurisdiction excluding its conflict-of-law provisions. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any use of the Original Work outside the scope of this License or after its termination shall be subject to the requirements and penalties of copyright or patent law in the appropriate jurisdiction. This section shall survive the termination of this License.

12) Attorneys' Fees. In any action to enforce the terms of this License or seeking damages relating thereto, the prevailing party shall be entitled to recover its costs and expenses, including, without limitation, reasonable attorneys' fees and costs incurred in connection with such action, including any appeal of such action. This

section shall survive the termination of this License.

13) Miscellaneous. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable.

14) Definition of "You" in This License. "You" throughout this License, whether in upper or lower case, means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with you. For purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

15) Right to Use. You may use the Original Work in all ways not otherwise restricted or conditioned by this License or by law, and Licensor promises not to interfere with or be responsible for such uses by You.

16) Modification of This License. This License is Copyright © 2005 Lawrence Rosen. Permission is granted to copy, distribute, or communicate this License without modification. Nothing in this License permits You to modify this License as applied to the Original Work or to Derivative Works. However, You may modify the text of this License and copy, distribute or communicate your modified version (the "Modified License") and apply it to other original works of authorship subject to the following conditions: (i) You may not indicate in any way that your Modified License is the "Open Software License" or "OSL" and you may not use those names in the name of your Modified License; (ii) You must replace the notice specified in the first paragraph above with the notice "Licensed under <insert your license name here>" or with a notice of your own that is not confusingly similar to the notice in this License; and (iii) You may not claim that your original works are open source software unless your Modified License has been approved by Open Source Initiative (OSI) and You comply with its license review and certification process.

1.4 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) is the holy-grail of encryption, and the cypherpunks dream. FHE encrypted cyphertexts can be computed on/ used for processing in arbitrary computational-depth calculations while being continuously in cyphertext form, or rather without the ability of decrypting the cyphertexts by the data processor. What this means is it is now possible to process any such encrypted data without any possibility of data leaks, and without any ability to decrypt or discern a users data. Furthermore the answers or predictions given by these FHE compatible data processors is simply a transformation of the input cyphertext, meaning only the original encryptor of the data/ private key holder can decrypt these answers.

Data processors can process FHE cyphertexts, and can own their deep/ machine learning models as a service.

Users (including other industries) can keep their data, and predictions/ calculations completely private, in quantum-decryption resistant form without needing to give anyone else their private key. This includes highly sensitive domains such as diagnosis requiring very personal patient data which is now indecipherable.

Win-Win. This benefits all directly involved. However there are some drawbacks to FHE:

- processing cyphertexts is naturally more (computationally, spatially, and thus monetarily) expensive and slow. Orders of magnitude.
- processing cyphertexts is more complex, requiring that all operations are abelian compatible I.E addition, multiplication, addition of a negative, but not any division or true subtraction.
- FHE is still relatively new, and each implementation is slightly different and could still hold bugs. Most FHE implementations are not FHE yet as they have not implemented bootstrapping thus having a set maximum computational depth.

1.4.1 ReArray

Note: You may see some or no content at all on this page that documents the API. If that is the case please build the documentation locally (*Docker Build*), or view this documentation using a "autodoc"-ed version of this documentation (see: *Documentation Variations*).

ReArray is a second level abstraction that uses some backend like ReSeal and packages it to be able to handle ndimensional operations, batches, and cross batch operations, while conforming to a custom numpy container implementation, making FHE compatible with numpy.

1.4.2 ReSeal

Note: You may see some or no content at all on this page that documents the API. If that is the case please build the documentation locally (*Docker Build*), or view this documentation using a "autodoc"-ed version of this documentation (see: *Documentation Variations*).

The core ReSeal library consists primarily of two components, the cache (ReCache), and the Microsoft Simple Encrypted Arithmetic Library (SEAL) abstraction (ReSeal).

class fhez.recache.**ReCache**(*enable=None*)

Core caching object for ReSeal.

This cache can be enabled (default) to improve/ minimise the need to regenerate the transient properties of ReSeal. That is to say ReSeal only keeps/ stores the minimum required attributes like the keys, the cyphertext and the parameters, as everything else is derived from those at the point of need. Thus this class caches the generated intermediaries so they can be re-used rather than commiting compute power every time they are called. E.G seal.Encryptor and seal.Decryptor are examples of cached objects.

class fhez.rescheme.**ReScheme**(**, only: Optional[Union[Sequence[str], Set[str]]] = None, exclude: Union[Sequence[str], Set[str]] = (), many: bool = False, context: Optional[Dict] = None, load_only: Union[Sequence[str], Set[str]] = (), dump_only: Union[Sequence[str], Set[str]] = (), partial: Union[bool, Sequence[str], Set[str]] = False, unknown: Optional[str] = None*)

Marshmallow serialisation schema.

This schema works with ReSeal to help in transmission of serialised ReSeal objects. This also helps to verify the contents are structured as expected on the recieving end. However since byte strings are encoded as strings there is little further testing that can be done on them.

1.5 Installation

Currently due to a certain amount of awkwardness of the python to Microsoft-SEAL bindings, the only "supported" installation method is by docker container. However we are working on fixing some of these issues in the near future, or at the very least supporting a second pure-python back-end so that installation is easier as a native library if that is desired.

1.5.1 Docker Registry

Dockerised images that you can play around with are available at: https://gitlab.com/deepcypher/python-fhez/container_registry/2063426

E.G, download and run interactively:

```
docker run -it registry.gitlab.com/deepcypher/python-reseal:master
```

1.5.2 Docker Build

If you would rather build this library locally/ for security reasons, you can issue the following build command from inside the projects root directory:

```
docker build -t archer/fhe -f Dockerfile_archlinux .
```

Note: The build process could take several minutes to build, as it will compile, bind, and package the Microsoft-SEAL library in with itself.

Docker Documentation Build

To build the detailed documentation with all the autodoc functionality for the core classes, you can use your now built container to create these docs for you by:

```
docker run -v ${PWD}/docs/build/:/python-fhe/docs/build -it archer/fhe make -C /python-
↪fhe/docs html
```

The docs will then be available in `${PWD}/docs/build/html` for your viewing pleasure

Locally-Built Docker-Image Interactive Usage

To run the now locally built container you can issue the following command to gain interactive access, by selecting the tag (-t) that you named it previously (here it is archer/fhe):

```
docker run -it archer/fhe
```

1.5.3 FHE Modules/ Plugins

This library will support extensions that expose a uniform API which we call "errays" or encrypted-arrays much like numpy custom containers, of which as simple implementation can be achieved by inheriting from our erray class. We hope to add more modules over time and would welcome others implementations too. We ourselves support the following modules:

- Microsoft SEAL (beta)*
- py-fhe (Saroja Erabelli) (alpha)

* Currently built in but being separated.

Note: More instructions to follow on specific installation of modules once implementation is complete for them.

1.5.4 Build The Docs

If you would like to build the documentation manually, for example to auto-doc the API (which is not easy in RTD) then from this directory:

- build the Docker container (docker must be installed, repository must be cloned, and you must be in this directory)

```
sudo docker build -t archer/fhe -f Dockerfile_archlinux .  || exit 1
```

- run the docker container with volume-mount, and trigger documentation build

```
sudo docker run -v ${PWD}/docs/build/:/python-fhe/docs/build -it archer/fhe make -C /
↪python-fhe/docs html
```

- you can then find the documentation in: `${PWD}/docs/build/html/`

The docs will walk you through the rest. Enjoy.

1.6 Examples

Table 1: Example Readiness Status

Category	Name	Docs	IsReady
classification	Fashion-MNIST	*Fashion-MNIST*	
regression	Constelation	*Constelation*	

We have a few in-progress examples, primarily in the examples directory in the python-FHEz repository. These examples use Jupyter Lab to make it as easy and as streamlined as possible for others to use our code/ play around with our code. Further to this you will note there are two dockerfiles in the python-reseal repository, one that expressly states jupyter which is expressly for these examples since they often require more dependencies than the base library itself we bundled things separately. To make it even easier we also included a bash script called run-jupyter which builds and runs the jupyter dockerfile with all our dependencies within, launches the server and gives you a link to the server usually: http://127.0.0.1:8888 + some authentication key and also mounts the examples directory as a volume so you could save any workings beyond the container as long as they are done inside the examples directory.

All of our examples should be played with using this Dockerfile for now as we improve our build system and make it easier to install our very complex library.

1.6.1 Constelation

Interactive Graph

The following **interactive graph** represents the neural network used for generic regression to exemplify this library in use:

1.6.2 Fashion-MNIST

This doubles as an example and as reproducible code to get our results from one of our (soon to be) published papers. This example will download Fashion-MNIST (a drop in more complex replacement for standard MNIST) in CSV format, normalise it, and begin the training process.

Interactive Graph

The following **interactive graph** represents the neural network used for MNIST prediction to exemplify this library in use:

Usage

To run this example please use the generic docker script. Then the fashion-MNIST example is in the file fashion-mnist.ipynb which you can open in the browsers jupyter lab session (use the last of the three links).

1.6.3 Generic Instructions for Example Usage

For any following example they are all accessed from the same container unless specified otherwise, which in theory since it is a Docker container can run on any 64-bit based system.

Step-by-Step

- Install Git - https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
- Install Docker - https://docs.docker.com/engine/install/
- Ensure Docker is running - https://docs.docker.com/config/daemon/
- Download source code - Download from GitHub using command line/ terminal: `git clone https://github.com/dreamingraven/python-reseal` - Or download from GitHub GUI and unzip: https://github.com/DreamingRaven/python-reseal/releases - Or download from GitLab: `git clone https://gitlab.com/georgeraven/python-reseal` - Or download from GitLab GUI and unzip: https://gitlab.com/GeorgeRaven/python-reseal/-/tags
- Build Docker container (this will take some time) - If your system supports bash then you can just run `run-jupyter` from inside the source directory to skip this step - Else from command line/ terminal within the source directory `docker build -t archer/fhe-jupyter -f Dockerfile_archlinux_jupyter .`
- Run Docker container - If your system supports bash then you can just run `run-jupyter` from inside the source directory to skip this step - Else from command line/ terminal within the source directory `docker run -p 127.0.0.1:8888:8888 -v "./examples":"/python-fhe/examples" -it archer/fhe-jupyter`
- Connect to the now running container using jupyter labs port: - Launch a web browser to the local address that the terminal outputs it will look something like: http://127.0.0.1:8888/lab?token=190874369467194612894628462874562 8 where you should now have an interactive jupyter lab view of the code
- Play around with the examples in the "examples" directory to your hearts content

1.7 Neural Network

The computational graph or the neural network similar to many deep learning frameworks is our chosen glue/ executor of individual neurons or nodes. Assuming all nodes follow a predictable format usually using our Node abstraction, we can traverse the graph to execute nodes, or even use the longest path to calculate the minimum encryption parameters necessary for a leveled homomorphic cyphertext to successfully reach the end before the noise limit has been reached.

1.7.1 Neural Network Graph

fhez.nn.nn.**NN**
alias of *fhez.nn.nn.NeuralNetwork*

class fhez.nn.nn.**NeuralNetwork**(*graph=None*)
Multi-Directed Neural Network Graph Handler.

This class handles traversing computational graphs toward some end-state, while computing forward(s), backward(s), and update(s) of the respective components described within.

backward(*gradient*, *current_node*, *end_node*)
Traverse backwards until all nodes processed.

backwards(*gradients*, *current_node*, *end_node*)
Calculate backward pass for multiple examples simultaneously.

forward(*x*, *current_node*, *end_node*)
Traverse and activate nodes until all nodes processed.

forwards(*xs*, *current_node*, *end_node*)
Calculate forward pass for multiple examples simultaneously.

property g
Get computational graph.

probe_shape(*lst: list*)
Get the shape of a list, assuming each sublist is the same length.

This function is recursive, sending the sublists down and terminating once a type error is thrown by the final point being a non-list

update(*current_node*, *end_node*)
Update weights of all nodes using oldest single example gradient.

updates(*current_node*, *end_node*)
Update the weights of all nodes by taking the average gradient.

1.7.2 Node

class fhez.nn.graph.node.**Node**
Abstract class for neural network nodes for traversal/ computation.

abstract backward(*gradient*)
Calculate backward pass for singular example.

backwards(*gradients*)
Calculate backward pass for multiple examples simultaneously.

property cache
Get caching dictionary of auxilary data.

abstract property cost
Get the computational cost per forward example of the node.

disable_cache()
Disable caching.

enable_cache()
Enable caching.

abstract forward(*x*)
Calculate forward pass for singular example.

forwards(*xs*)
Calculate forward pass for multiple examples simultaneously.

property gradients
Get cached input stack.

For neural networks to calculate any given weight update, it needs to remember atleast the last gradient in the case of stocastic descent, or multiple gradients if implementing batch normalised gradient descent. This is a helper method that initialises a stack so that implementation can be offloaded and made-uniform between all subclasses

property inputs
Get cached input stack.

Neural networks backpropogation requires cached inputs to calculate the gradient with respect to x and the weights. This is a utility method that initialises a stack and allows you to easily append or pop off of it so that the computation can occur in FILO.

property is_cache_enabled
Get status of whether or not caching is enabled.

property optimiser
Get optimiser object, E.G Stocastic Gradient Descent.

probe_shape(*lst: list*)
Get the shape of a list, assuming each sublist is the same length.

This function is recursive, sending the sublists down and terminating once a type error is thrown by the final point being a non-list

abstract update()
Update node state/ weights for a single example.

updater(*parm_names: list*, *it=None*)
Private function to universaly update any Node instance.

To simplify the process of updating so and to reduce code duplication, this function serves to derive all the important information given a parameter dictionary. It will then infer from the dictionary the attributes with which to modify.

abstract updates()
Update node state/ weights for multiple examples simultaneously.

1.8 Traversal

Traversal algorithms are the key algorithms that take given graphs and stimulate neurons with some inputs, to achieve some desired state of activation.

1.8.1 Neuronal Firing

Neural network generic firing abstraction.

class fhez.nn.traverse.firing.**Firing**(*graph=None*)
Simple exhaustive neuronal firing calculation.

adaptation()
Correct nodes based on learnt gradient.

correction(*signals*, *receptors*)
Calculate/ learn correction necessary to become closer to our goal.

Parameters

- **signals** – signal to be induced in corresponding receptor
- **receptors** – receptor to be signaled

property graph
Get neuron graph to fire.

harvest(*node_names: list*)
Harvest forward response from neuronal firing, using probes.

This will replay the last node to calculate its output.

probe_shape(*lst: list*)
Get the shape of a list, assuming each sublist is the same length.

This function is recursive, sending the sublists down and terminating once a type error is thrown by the final point being a non-list

stimulate(*neurons: numpy.ndarray*, *signals: numpy.ndarray*, *receptor='forward'*, *debug=False*)
Stimulate a set of receptors with a set of signals for response.

Breadth first stimulation of neurons/ nodes. Note that this is a single simultaneous stimulation and subsequent response. If a neuron does not fire I.E produces no (None) result then that neuron will not be followed until it does produce a result.

Parameters

- **receptors** (*list(str)*) – list of node names to recieve stimulus
- **signals** (*np.ndarray or compatible*) – positional list of signals for the equally positioned receptor
- **receptor** (*str*) – Name of function/ sequence of functions to call of nodes

fhez.nn.traverse.firing.**NeuronalFiring**
alias of *fhez.nn.traverse.firing.Firing*

1.9 FHE Parameterisation

FHE nodes need to be parameterised to be properly capable of computing the source-to-sink computational distance in the network graph. If the FHE parameters do not match or at-least meet the minimum requirements, the cyphertext will be garbled. If no parameters are given plaintext is assumed, so please do parameterise the nodes.

Sources are where data is encrypted. Sinks are where cyphertexts are consumed/ decrypted. Compute nodes are just generic data processors that work with both cyphertexts and plaintexts. Rotation nodes are either explicit bootstrapping nodes, or deliberate key-rotation operations to refresh the keys or reform how the data is encrypted e.g from a single cyphertext to split into many smaller cyphertexts.

To help automate parameterisation we have some useful utilities that you may be interested in:

1.10 Layers

Table 2: Layers Status

Category	Name	Docs	Forward	Backward
nn layer	Artificial (ANN)	*Fully Connected Dense Net (ANN)*		
nn layer	Convolutional (CNN)	*Convolutional Neural Network (CNN)*		
nn layer	Recurrent (RNN)	section_rnn		

1.10.1 Fully Connected Dense Net (ANN)

Here ANN shall mean a fully connected/ dense neuron. Usually these are depicted similar to the following:

We however want to keep using a computational graph style. This computational graph style is not as neat as the traditional style, however we find it much more helpful when doing things like visually inspecting operations that are important to account for in FHE, as each individual operation comes at a cost, and sometimes it can be difficult to see how many steps are involved in the traditional style depictions. Follows is out computational graph variant of the previous neuron:

We can then expand these computational graphs to show en-mass operations. This is even more helpful as now we can see how the data comes in together, and how each multi-dimensional matrix accrues the same operations upon it. This is important as we do not encrypt individual values by themselves. Instead they are encoded into a polynomial and that polynomial is then encrypted. Please keep in mind the *Commuted-Sum*.

ANN Equations

Thankfully there needs not be any approximation in an ANN ignoring the activation function. Thus our ANN can be largely unchanged compared to standard implementations, both being polynomials (excluding g)

ANN

(1.1) ANN: There is little unique about our ANN with the exception of the application of the bias.

Normal ANN equation (not compatible with our representations, where w_0 is actually the bias):

$$a = g(\sum_{i=1}^{T_x}(w_i x_i) + w_0) \tag{1.1}$$

Our ANN implementation (1.2) slightly differs to this (1.1), to handle the *Commuted-Sum* problem is as follows but note how the bias is divided by N which in normal scenarios is simply 1 since it would be a single value, whereas in scenarios where an input x is an un-summable cyphertext holding a multi-dimensional array, b/N serves to counteract broadcasting of values keeping activations in the golden range for our activation function:

$$a^{(i)} = g(\sum_{t=0}^{T_x-1}(w^{<t>}x^{(i)<t>}) + b/N) \tag{1.2}$$

ANN Derivatives

The derivative of an ANN (f) with respect to the bias b:

$$\frac{df}{db} = 1\frac{dg}{dx} \tag{1.3}$$

The derivative of an ANN (f) with respect to the weights w:

$$\frac{df}{dw^{<t>}} = x^{(i)<t>}\frac{dg}{dx} \tag{1.4}$$

The derivative of a ANN (f) with respect to the input x:

$$\frac{df}{dx^{(i)<t>}} = w^{<t>}\frac{dg}{dx} \tag{1.5}$$

Note: where:

- x:
  - $x^{(i)}$; the multidimensional-input array used as the i'th training example / pass of the network. E.G cnn.forward is one whole forward pass.
  - $x_n^{(i)<t>}$; The n'th input value of the multi-dimensional input array $x^{(i)}$. Corresponding to the i'th training example of the network, and branch/ time-step t.
- T_x and t:
  - T_x ; The total number of branches per input array x. No need for $T_x^{(i)}$ as branches should be the same every time.
  - t ; The current (relative)/ t'th timestep/ branch.
- N and n:
  - N; the total number of elements in any individual multi-dimensional input array x
  - n; the n'th input element any individual multi-dimensional input array x, e.g x_n is the n'th value x in the multi-dimensional array x.

- g and a
  - g; some activation function e.g σ_a (see:\sigma_a(x))
  - a; the sum of output / activation of this neural network (if the last network then $a = \hat{y}$)
- y and $\hat{y}$:
  - y; the (normalized) ground-truth / observed outcome
  - $\hat{y}$; the (normalized) prediction of y
- w, k, and b:
  - w; a weight
  - b; a bias
  - k; a kernel that multiplies over some input data, for us this is the *Kernel-Masquerade*

Please also note, this is with respect to each network. One networks output activation a might be another networks input x

---

ANN API

Artificial Neural Network (ANN) as node abstraction.

class fhez.nn.layer.ann.**ANN**(*weights: Optional[numpy.array] = None*, *bias: Optional[int] = None*)

Dense artificial neural network as computational graph.

property b

Shorthand for bias.

backward(*gradient*)

Compute backward pass of neural network.

$$\frac{df}{db} = 1\frac{dg}{dx}$$
$$\frac{df}{dw^{<t>}} = x^{(i)<t>}\frac{dg}{dx}$$
$$\frac{df}{dx^{(i)<t>}} = w^{<t>}\frac{dg}{dx}$$

property bias

Get ANN sum of products bias.

property cost

Get no cost of a this node.

forward(x)

Compute forward pass of neural network.

$$a^{(i)} = \sum_{t=0}^{T_x - 1} (w^{<t>} x^{(i)<t>}) + b$$

update()

Update weights and bias of the network stocastically.

updates()

Update weights and bias as one batch all together.

property w
Shorthand for weights.

property weights
Get the current weights.

1.10.2 Convolutional Neural Network (CNN)

Convolutional neural networks are quite complicated cases with FHE. Since the encrypted cyphertext is the most atomic form of the input data that we can access and we need to be able to multiply subsets of the data by different amounts we use a sparse n-dimensional array with the weights embedded in different positions and the rest zeroed out (see *Kernel-Masquerade* and *Hadmard Product*). This way we can still convolve out filters/ kernels but instead of manipulating x (normally by selecting a slice of x that you were interested in) we manipulate the filter instead generating windows where the filter should be placed, and caching these windows for later use in back-propogation so we know exactly what input x multiplied which weights once x is finally decrypted. Each window becoming a different branch $< t >$ and the total number of windows our total branches T_x.

CNN Equations

Standard neuron equation (not compatible with our representations, where w_0 is actually the bias):

$$a = g(\sum_{i=1}^{N}(w_i x_i) + w_0) \tag{1.6}$$

Kernel-Masquerade, weighted, and biased cross correlation.

$$a^{(i)<t>} = g(\sum_{t=0}^{T_x - 1}(k^{<t>} x^{(i)}) + b/N) \tag{1.7}$$

CNN Derivatives

The derivative of a CNN (f) with respect to the bias b:

$$\frac{df(x)}{db} = T_x \frac{dg}{dx} \tag{1.8}$$

The derivative of a CNN (f) with respect to the weights multi-dimensional array w is the sum of all portions of $x^{(i)}$ unmasked during product calculation:

$$\frac{df(x)}{dw} = \sum_{t=0}^{T_x}(x^{(i)<t>})\frac{dg}{dx} \tag{1.9}$$

The derivative of a CNN (f) with respect to the input x:

$$\frac{df(x)}{dx} = \sum_{t=0}^{T_x}(k^{(i)<t>})\frac{dg}{dx} \tag{1.10}$$

Note: where:

- x:

  - $x^{(i)}$; the multidimensional-input array used as the i'th training example / pass of the network. E.G cnn.forward is one whole forward pass.
  - $x_n^{(i)<t>}$; The n'th input value of the multi-dimensional input array $x^{(i)}$. Corresponding to the i'th training example of the network, and branch/ time-step t.
- T_x and t:
  - T_x ; The total number of branches per input array x. No need for $T_x^{(i)}$ as branches should be the same every time.
  - t ; The current (relative)/ t'th timestep/ branch.
- N and n:
  - N; the total number of elements in any individual multi-dimensional input array x
  - n; the n'th input element any individual multi-dimensional input array x, e.g x_n is the n'th value x in the multi-dimensional array x.
- g and a
  - g; some activation function e.g σ_a (see:*sigma_a(x)*)
  - a; the sum of output / activation of this neural network (if the last network then $a = \hat{y}$)
- y and $\hat{y}$:
  - y; the (normalized) ground-truth / observed outcome
  - $\hat{y}$; the (normalized) prediction of y
- w, k, and b:
  - w; a weight
  - b; a bias
  - k; a kernel that multiplies over some input data, for us this is the *Kernel-Masquerade*

Please also note, this is with respect to each network. One networks output activation a might be another networks input x

---

Kernel-Masquerade

The *Kernel-Masquerade* is the combining of a convolutional kernel and a *mask* to simultaneously calculate the product of any given kernel/ filter over a dataset. This is to *act* as though we were capable of normally slicing some input data which is impossible when this data is embedded in the FHE cyphertext. We deploy kernel weights embedded in a sparse/ zeroed multi-dimensional array, thus ignoring undesired values in the input cyphertext when finding the *Hadmard Product* of the two, and minimising computational depth of cyphertexts in processing.

Hadmard Product

The Hadmard product is simply two equally shaped n-dimensional arrays operated on element wise to produce a third product n-dimensional array of the same size/ shape:

Commuted-Sum

Warning: If you are intending to write your own or extend an additional neural network please pay special attention to the *Commuted-Sum*, it will change the behavior of the networks drastically and in some unexpected ways if it is not accounted for. CNNs and other single input multiple branch networks are the source of commuted-sums.

In our neural networks operating on fully homomorphically encrypted cyphertexts, the cyphertexts encode into themselves multiple values, I.E for us they include **all** the values in a single training example (i), to give a more concrete example, for a CNN one cyphertext is the **whole** of a single image, thats all pixel values of the image. This means computations happen quicker and take up less space as there is less overhead since we do not need different parameters between each pixel value, instead we can encode and encrypted them all using the same parameters, reducing duplication, and allowing us to operate on all of them simultaneously. A consequence to this approach however is that the cyphertext is the most atomic form of the data we have, it cannot be any smaller, it will always be a polynomial with N number of values encoded in it. In practice this means we cannot sum a cyphertext, as it would mean reducing all these values into one single value I.E the sum of the cyphertext, or to put it a different way, we cannot fold the cyphertext in on itself to get this one single number, as this is *homomorphic* encryption I.E structure preserving self similar encryption, the form of what comes in is the form of what comes out.

In the case of CNNs however as you can see in *CNN Equations* there is a sum. As we have stated it is impossible to sum the cyphertext in on itself, thus since almost all our networks *have* to be/ use abelian operations we can *commute* this sum until after we have decrypted as long as we pay special attention to the fact that we have big matrices that should be treated as if they were singular values. The most egregious possible violation is broadcasting of a single addition, as this acts differently on a single value than it does on a matrix of values, thus the need to always divide a bias b by the total number of values being biased N. Assuming our calculus is correct and we succeffully implement a commuted sum until after decryption, this should only ever be a problem in one dimension, all subsequent sums would happen between cyphertexts (since it should have been summed already) meaning there is only ever one commuted sum, we never have to worry about a stack of them.

CNN API

Note: You may see some or no content at all on this page that documents the API. If that is the case please build the documentation locally (*Docker Build*), or view this documentation using a "autodoc"-ed version of this documentation (see: *Documentation Variations*).

CNN has been split into its constituent operations.

See fhez.nn.operations.cnn -> *Cross Correlation (CC)*

1.11 Activations

Neural network activation functions are the primary logic that dictates whether a given neuron should be activated or not, and by how much. Activation functions are best when they introduce some non-linearity to the neuron so that it can better model more complex behaviors.

In our case all activations are children of the Node abstraction to keep things consistent, and reduce the amount of redundant code.

Table 3: Activations Status

Category	Name	Docs	Forward	Backward
activation	Argmax	*Argmax*		
activation	Linear	*Linear*		
activation	ReLU (Approx)	*R_a(x)*		
activation	Sigmoid (Approx)	*\sigma_a(x)*		
activation	Softmax	*Softmax*		

1.11.1 Argmax

Argmax API

class fhez.nn.activation.argmax.**Argmax**

Argmax activation, compute sparse array of highest activation.

backward(*gradient: numpy.ndarray*)

Calculate the argmax derivative with respect to each input.

Argmax.

property cost

Get computational cost of this activation.

forward(*x: numpy.ndarray*)

Calculate the argmax of some input x along its first axis.

Argmax.

update()

Update parameters, so nothing for argmax.

updates()

Update parameters using average of gradients so none for argmax.

1.11.2 Linear

Linear API

Note: You may see some or no content at all on this page that documents the API. If that is the case please build the documentation locally (*Docker Build*), or view this documentation using a "autodoc"-ed version of this documentation (see: *Documentation Variations*).

class fhez.nn.activation.linear.**Linear**(*m=array([1])*, *c=array([0])*, *optimiser={'_beta_2': 0.999, '_beta_1': 0.9, '_alpha': 0.001, '_epsilon': 1e-08}*)

Linear activation function computational graph abstraction.

backward(*gradient*)
Get gradients of backward prop.

property c
Intercept.

property cost
Get the computational cost of this Node.

forward(*x*)
Get linear forward propogation.

property m
Slope.

property optimiser
Get current optimiser object.

property schema
Get Marshmallow schema representation of this class.

Marshmallow schemas allow for easy and trustworthy serialisation and deserialisation of arbitrary objects either to inbulit types or json formats. This is an inherited member of the abstract class Serialise.

Note: Anything not listed here will inevitably be lost, ensure anything important is identified and expressley stated its type and structure.

---

update()
Update any weights and biases for a single example.

updates()
Update any weights and biases based on an avg of all examples.

1.11.3 ReLU & Approximation

Warning: This activation function has an asymptote to y infinity outside of a very small *safe* band of input x values. This **will** cause *nan* and extremely large numbers if you aren't especially careful and keep all values passed into this activation function within the range -q to q which is its *golden* range, which can also be learned with backpropagation. Think especially carefully of your *initial* weights, and whether or not they will exceed this band into the *danger* zone. See: *R_a(x)*

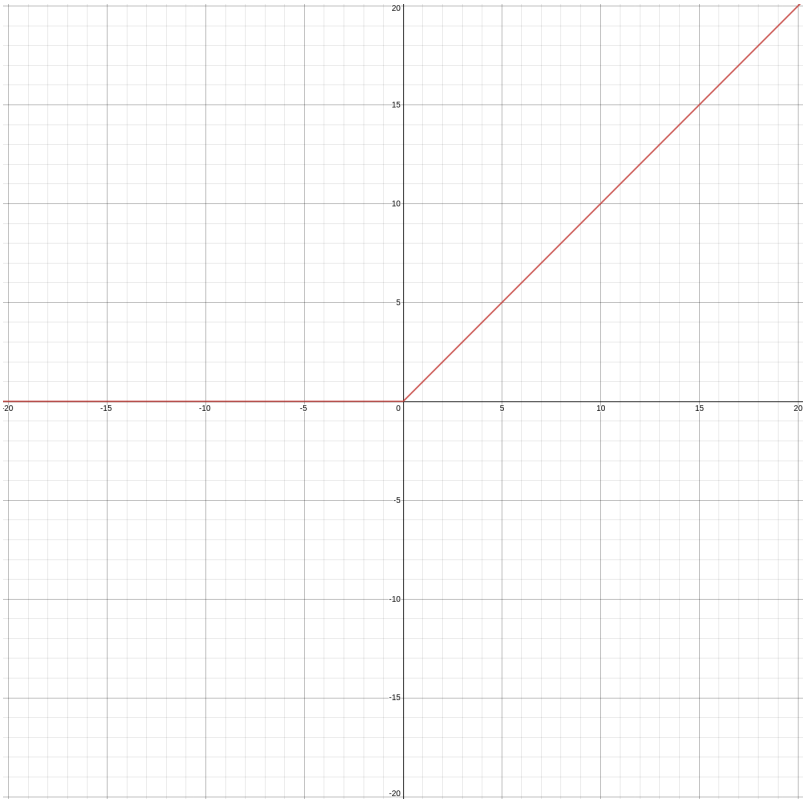
To be able to use (fully homomorphically encrypted) cyphertexts with deep learning we need to ensure our activations functions are abelian compatible operations, polynomials. relu (1.11) is not a polynomial, thus we approximate (1.13). Similarly since we used an approximation for the forward activations we use a derivative of the relu approximation (1.14) for the backward pass to calculate the local gradient in hopes of descending towards the global optimum (gradient descent).

ReLU $R(x)$

$R(x)$

(1.11) Relu

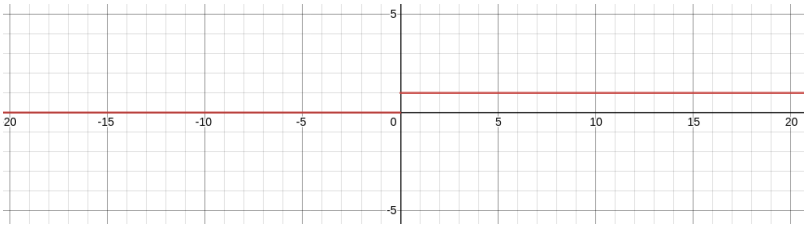
$$R(x) = \max(0, x) \tag{1.11}$$



$\frac{dR(x)}{dx}$

(1.12)

$$\frac{dR(x)}{dx} = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{1.12}$$



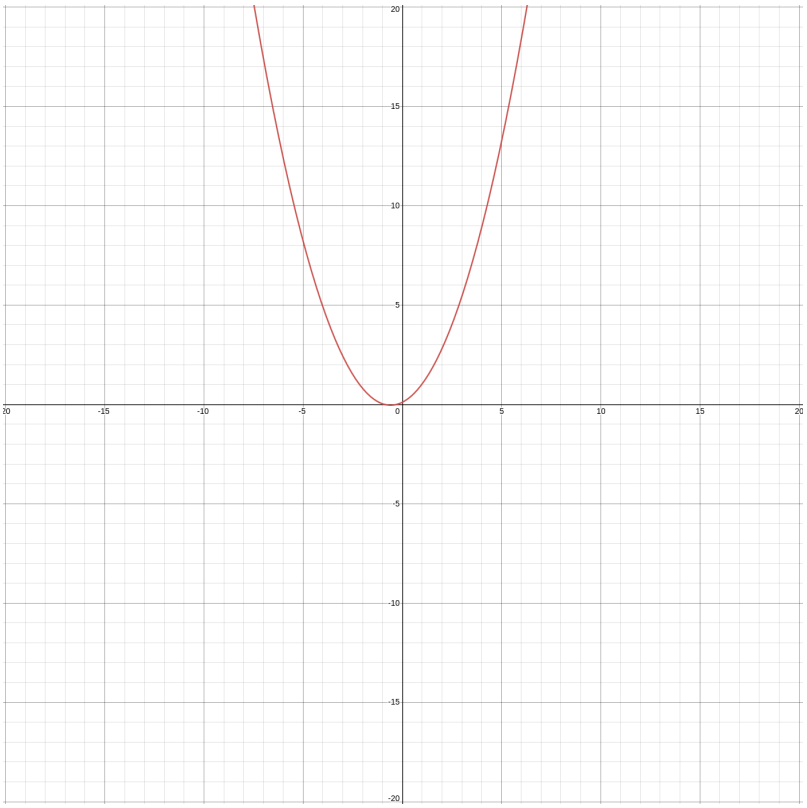
ReLU-Approximation $R_a(x)$

$R_a(x)$

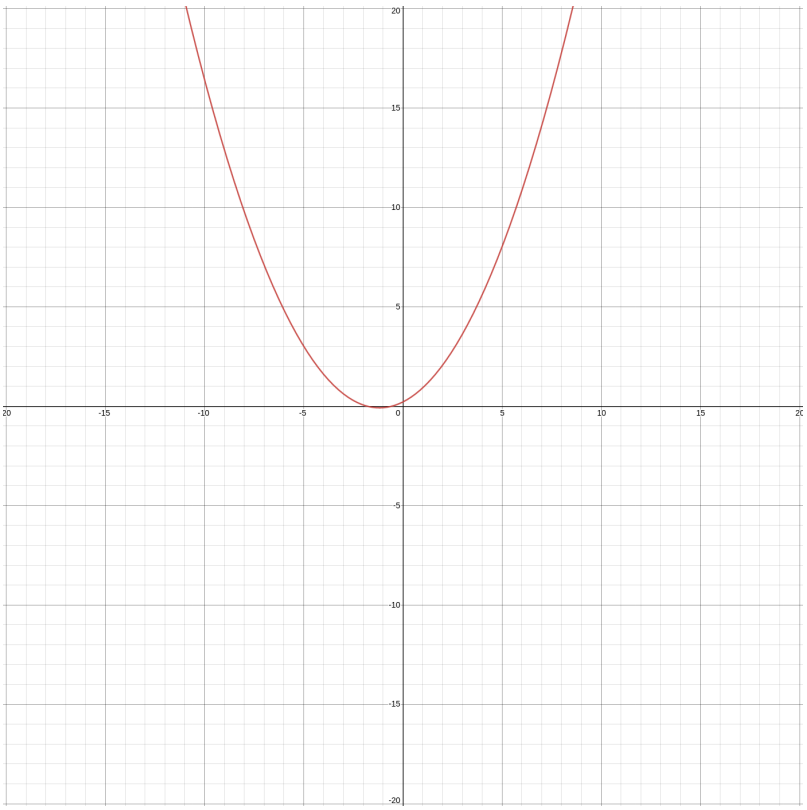
(1.13) relu-approximation

$$R(x) \approx R_a(x) = \frac{4}{3\pi q}x^2 + \frac{1}{2}x + \frac{q}{3\pi}, where\ x \in \{q > x > -q \subset\} \tag{1.13}$$

where q is 1:



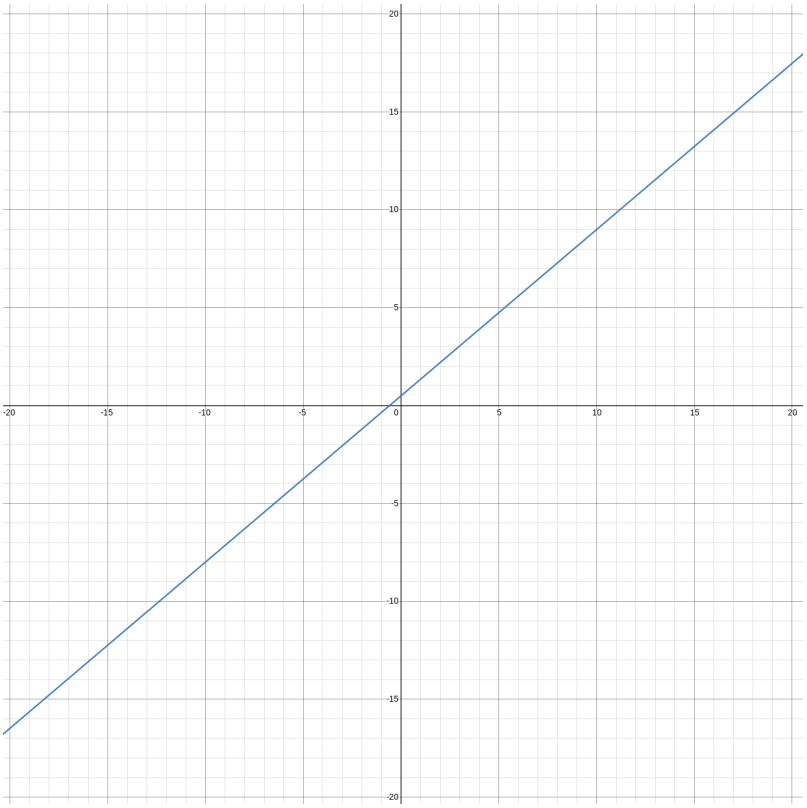
where q is 2



$\frac{dR_a(x)}{dx}$

(1.14) relu-approximation derivative

$$\frac{dR(x)}{dx} \approx \frac{dR_a(x)}{dx} = \frac{8}{3\pi q}x + \frac{1}{2}, where\ x \in \{q > x > -q \subset\} \tag{1.14}$$



ReLU Approximate API

Note: You may see some or no content at all on this page that documents the API. If that is the case please build the documentation locally (*Docker Build*), or view this documentation using a "autodoc"-ed version of this documentation (see: *Documentation Variations*).

---

class fhez.nn.activation.relu.**RELU**(*q=None*)

Rectified Liniar Unit (ReLU) approximation computational graph node.

backward(*gradient*)

Calculate backward pass for singular example.

property cost

Get the computational cost of traversing to this RELU node.

forward(*x*)

Calculate forward pass for singular example.

local_dfdq(*x*, *q*)

Calculate local derivative dfdq.

local_dfdx(*x*, *q*)

Calculate local derivative dfdx.

property q

Get the current ReLU approximation range.

update()

Update node state/ weights for a single example.

updates()
Update node state/ weights for multiple examples simultaneously.

1.11.4 Sigmoid & Approximation

Warning: This activation function has an asymptote to negative y infinity and positive y infinity outside of a very small *safe* band of input x values. This **will** cause *nan* and extremely large numbers if you aren't especially careful and keep all values passed into this activation function within the range -4 to 4 which is its *golden* range. Think especially carefully of your *initial* weights, and whether or not they will exceed this band into the *danger* zone. See: *\sigma_a(x)*

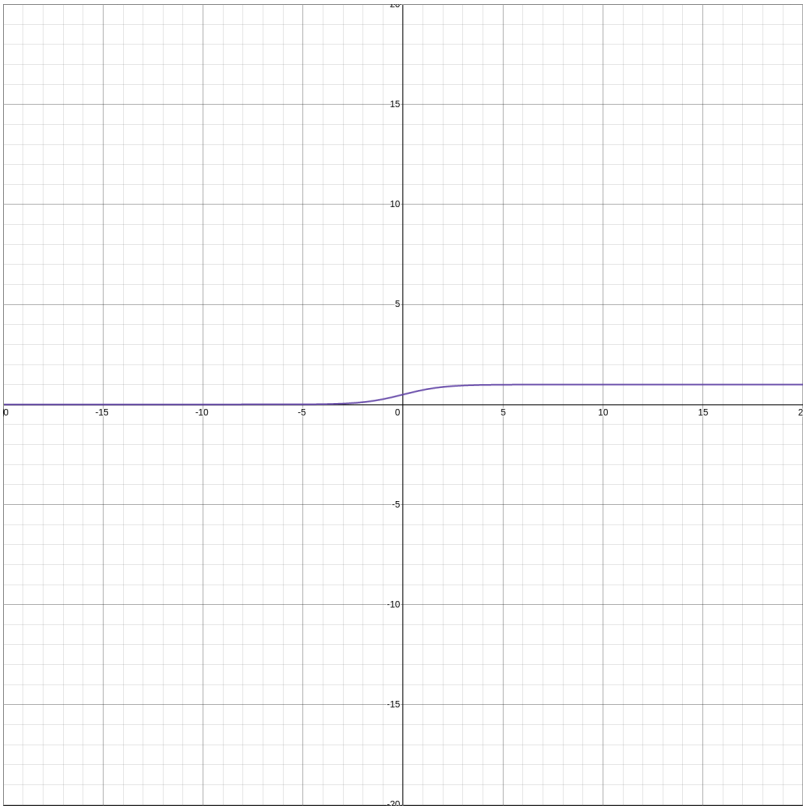
To be able to use (fully homomorphically encrypted) cyphertexts with deep learning we need to ensure our activations functions are abelian compatible operations, polynomials. Sigmoid (1.15) is not a polynomial, thus we approximate (1.17). Similarly since we used an approximation for the forward activations we use a derivative of the sigmoid approximation (1.18) for the backward pass to calculate the local gradient in hopes of descending towards the global optimum (gradient descent).

Sigmoid $\sigma(x)$

$\sigma(x)$

(1.15) Sigmoid

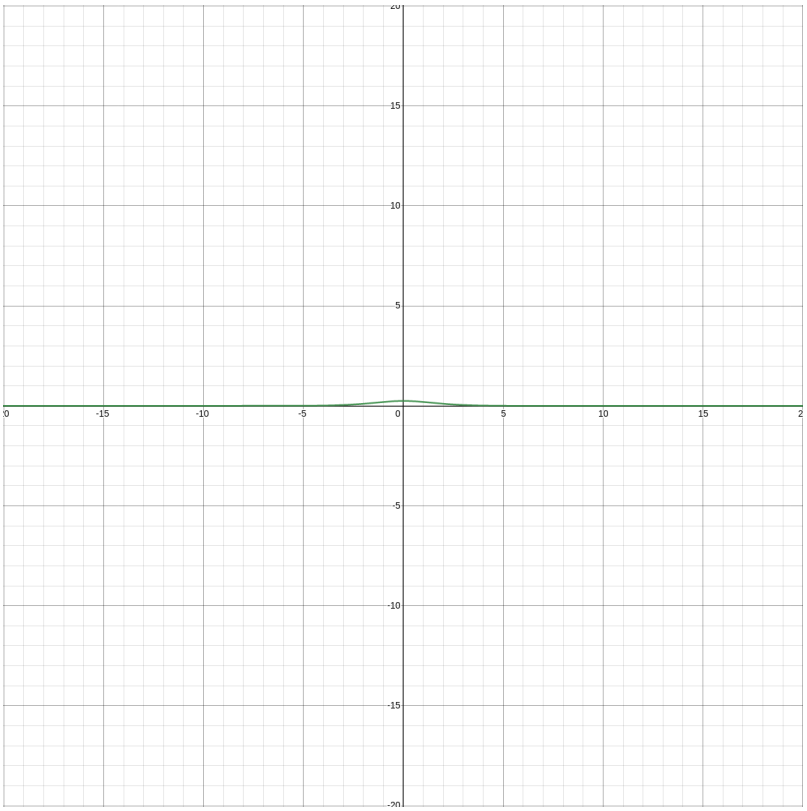
$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.15}$$



$\frac{d\sigma(x)}{dx}$

(1.16) Sigmoid derivative (Andrej Karpathy CS231n lecture)

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = (\frac{1+e^{-x}-1}{1+e^{-x}})(\frac{1}{1+e^{-x}}) = (1-\sigma(x))\sigma(x) \tag{1.16}$$

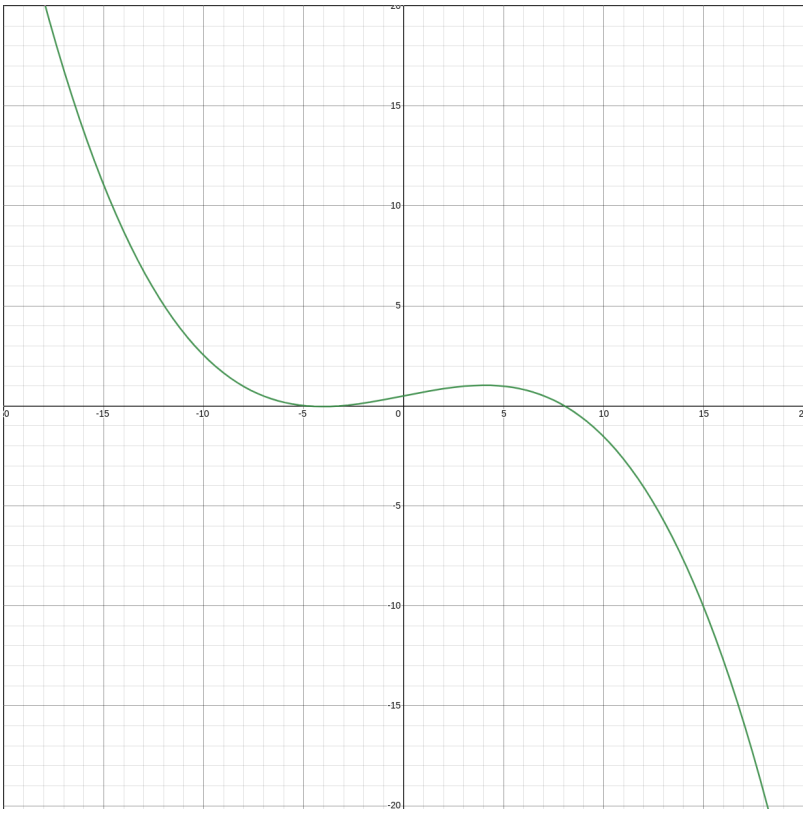


Sigmoid-Approximation $\sigma_a(x)$

$\sigma_a(x)$

(1.17) Sigmoid-approximation

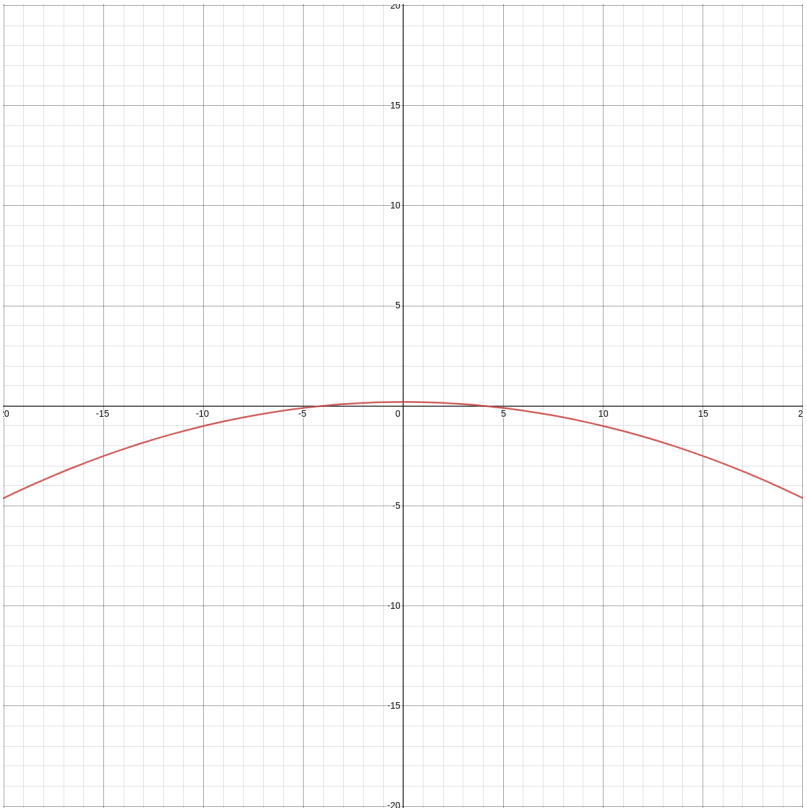
$$\sigma(x) \approx \sigma_a(x) = 0.5 + 0.197x + -0.004x^3, where\ x \in \{4 > x > -4 \subset\} \tag{1.17}$$



$\frac{d\sigma_a(x)}{dx}$

(1.18) Sigmoid-approximation derivative

$$\frac{d\sigma(x)}{dx} \approx \frac{d\sigma_a(x)}{dx} = 0.0 + 0.197 + (-0.004 * 3)x^2 = 0.197 + -0.012x^2, where\ x \in \{4 > x > -4 \subset\} \quad (1.18)$$



Sigmoid API

Note: You may see some or no content at all on this page that documents the API. If that is the case please build the documentation locally (*Docker Build*), or view this documentation using a "autodoc"-ed version of this documentation (see: *Documentation Variations*).

class fhez.nn.activation.sigmoid.**Sigmoid**

Sigmoid approximation.

backward(*gradient: numpy.array*)

Calculate gradient of sigmoid with respect to input x.

property cost

Get computational depth of this node.

forward(*x*)

Calculate sigmoid approximation while minimising depth.

sigmoid(*x*)

Calculate standard sigmoid activation.

update()

Update nothing, as sigmoid has no parameters.

updates()

Update nothing, as sigmoid has no parameters.

1.11.5 Softmax

Example Architecture

This figure shows a generic classification network, and how the softmax is likeley to be used.

API

Softmax activation as node abstraction.

class fhez.nn.activation.softmax.**Softmax**

Softmax activation, normalising sum of inputs to 1, as probability.

backward(*gradient: numpy.ndarray*)

Calculate the soft maximum derivative with respect to each input.

$$\frac{dSMAX(a)}{da_i} = \begin{cases} p(\hat{y}_i)(1 - p(\hat{y}_i)), & \text{if } c = i \\ -p(\hat{y}_c) * p(\hat{y}_i), & \text{otherwise} \end{cases}$$

where: c is the one hot encoded index of the correct/ true classification, and i is the current index for the current classification.

property cost

Get computational cost of this activation.

forward(*x: numpy.ndarray*)

Calculate the soft maximum of some input x.

$p(\hat{y}_i) = \frac{e^{a_i}}{\sum_{j=0}^{C-1} e^{a_j}}$

where: C is the number of classes, and i is the current class being processed.

update()

Update parameters, so nothing for softmax.

updates()

Update parameters using average of gradients so none for softmax.

1.12 Optimisers

Optimisers are responsible for taking a given gradient and the current parameters of a model and ordaining what the next best update to these parameters would be.

Table 4: Optimisers Status

Category	Name	Docs	Update
optimiser	Adam	*Adam*	
optimiser	Gradient Descent	*Gradient Descent*	

1.12.1 Adam

Original algorithm:

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Our implementation API:

class fhez.nn.optimiser.adam.**Adam**(*alpha: float = 0.001*, *beta_1: float = 0.9*, *beta_2: float = 0.999*, *epsilon: float = 1e-08*)

Adaptive moment optimiser abstraction.

Sources:

- https://arxiv.org/abs/1412.6980
- https://openreview.net/pdf?id=ryQu7f-RZ
- https://www.youtube.com/watch?v=JXQT_vxqwIs&t=276s
- https://keras.io/api/optimizers/adam/
- https://www.geeksforgeeks.org/intuition-of-adam-optimizer/

property alpha

Get learning rate hyperparameter.

Returns alpha α, defaults to 0.001

Return type float

property beta_1

Get first order moment exponential decay rate.

Returns beta_1 β_1, defaults to 0.9

Return type float

property beta_2

Get second order moment exponential decay rate.

Returns beta_2 β_2, defaults to 0.999

Return type float

property cache

Cache of iteration specific values.

This cache is a dictionary of keys (the parameter name) and values (the parameter specific variables). For example in this cache you can expect to get the previous iterations moment, and number of iterations.

property epsilon

Get epsilon.

Returns epsilon ϵ (not ε), defaults to $1e^{-8}$

Return type float

momentum(*gradient: float*, *param_name: str*, *ord: int = 1*)

Calculate momentum, of a single parameter-category/ name.

This function can calculate either 1st order momentum or 2nd order momentum (rmsprop) since they are both almost identical.

where moment is 1 (I.E first order):

- current moment $m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t$
- decayed moment $\hat{m}_t = \frac{m_t}{1^{\wedge}\beta_1^t}$

where moment is 2 (I.E second order/ RMSprop):

- current moment $v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * g_t^2$
- decayed moment $\hat{v}_t = \frac{v_t}{1^{\wedge}\beta_2^t}$

Steps taken:

- retrieve previous momentum from cache dictionary using key (param_name) and number of iterations
- calculate current momentum using previous momentum:
- Save current momentum into cache dictionary using key
- calculate current momentum correction/ decay:
- return decayed momentum

Parameters

- **gradient** (*float*) – gradient at current timestep, usually minibatch
- **param_name** (*str*) – key used to look up parameters in m_t dictionary
- **ord** (*int*) – the order of momentum to calculate defaults to 1

Returns $\hat{m}_t$ corrected/ averaged momentum of order ord

Return type float

Example Adam().momentum(gradient=100, param_name="w", ord=1)

optimise(*parms: dict*, *grads: dict*)

Update given params based on gradients using Adam.

Params and grads keys are expected to be *x* and *dfdx* respectiveley. They should match although the x in this case should re replaced by any uniquely identifying string sequence.

Parameters

- **parms** (*dict[str, float]*) – Dictionary of keys (param name), values (param value)
- **grads** (*dict[str, float]*) – Dictionary of keys (param name), values (param gradient)

Returns Dictionary of keys (param name), values (proposed new value)

Return type dict[str, float]

Example Adam().optimise({"b": 1},{"dfdb": 200})

`rmsprop`(*gradient: float*, *param_name: str*)
Get second order momentum.

`property schema`
Get Marshmallow schema representation of this class.

Marshmallow schemas allow for easy and trustworthy serialisation and deserialisation of arbitrary objects either to inbulit types or json formats. This is an inherited member of the abstract class Serialise.

Note: Anything not listed here will inevitably be lost, ensure anything important is identified and expressley stated its type and structure.

1.12.2 Gradient Descent

`class fhez.nn.optimiser.gd.GD`
Most basic no-momentum gradient descent.

1.13 Loss Functions

Loss functions are responsible for calculating some measure of wrongness. We usually use these to inform the neural network and us, just how good or bad our predictions $\hat{y}$ are compared to the ground truth y.

Table 5: Loss Functions Status

Category	Name	Docs	Forward	Backward
loss	Categorical Cross Entropy (CCE)	*Categorical Cross Entropy (CCE)*		
loss	Mean Absolute Error (MAE)	*Mean Absolute Error (MAE)*		
loss	Mean Squared Error (MSE)	*Mean Squared Error (MSE)*		

1.13.1 Categorical Cross Entropy (CCE)

Example Architecture

This figure shows a generic classification network, and how the CCE is likeley to be used.

Graph

The Graph here shows categorical cross entropy plotted on x and y axis. Where green is the target value, orange is the predicted value, red is the output of CCE, and blue is the local gradient of CCE.

See https://www.desmos.com/calculator/q2dwniwjsp for an interactive version.

API

Categorical Cross Entropy (CCE) as node abstraction.

class fhez.nn.loss.cce.**CCE**

Categorical cross entropy for **multi-class** classification.

This is also known as **softmax loss**, since it is mostly used with softmax activation function.

Not to be confused with binary cross-entropy/ log loss, which is instead for multi-label classification, and is instead used with the sigmoid activation function.

CCE Graph: https://www.desmos.com/calculator/q2dwniwjsp

backward(*gradient: numpy.ndarray*)

Calculate gradient of loss with respect to $\hat{y}$.

$$\frac{dCCE(p(\hat{y}))}{dp(\hat{y}_i)} = \frac{-1}{p(\hat{y}_i)} p(y_i)$$

property cost

Get 0 cost of plaintext loss calculation.

forward(*signal=None*, *y: Optional[numpy.ndarray] = None*, *y_hat: Optional[numpy.ndarray] = None*, *check=False*)

Calculate cross entropy and save its state for backprop.

Can either be given a network signal with both y_hat and y stacked, or you can explicitly define y and y_hat.

loss(*y: numpy.ndarray*, *y_hat: numpy.ndarray*)

Calculate the categorical cross entryopy statelessley.

$$CCE(p(\hat{y})) = -\sum_{i=0}^{C-1} y_i * \log_e(\hat{y_i})$$

where:

$$\sum_{i=0}^{C-1} p(\hat{y}_i) = 1$$
$$\sum_{i=0}^{C-1} p(y_i) = 1$$

fhez.nn.loss.cce.**CategoricalCrossEntropy**

alias of *fhez.nn.loss.cce.CCE*

1.13.2 Mean Absolute Error (MAE)

class fhez.nn.loss.mae.**MAE**

Loss function to node wrapper.

backward(*gradient*)

Calculate MAE gradient with respect to $\hat{y}$.

This will take a single gradient value, and return the average gradient with respect to $\hat{y}$

$$\frac{d}{d\hat{y}}(\text{MAE}) = \begin{cases} +1, & \hat{y} > y \\ 0, & \hat{y} = y \\ -1, & \hat{y} < y \end{cases}$$

cost()
Get computational cost of the forward function.

forward(*y: numpy.ndarray*, *y_hat: numpy.ndarray*)
Calculate the loss of the output given the ground truth.

This will take multiple values for both y and $\hat{y}$, and return a single value that is the mean of their absolute difference.

$\text{MAE} = \frac{\sum_{i=0}^{N-1} \|y - \hat{y}\|}{N}$

update()
Do nothing as there are no parameters to update.

updates()
Do nothing as there are no parameters to update.

1.13.3 Mean Squared Error (MSE)

class fhez.nn.loss.mse.**MSE**
Loss function to node wrapper.

backward(*gradient*)
Calculate MSE gradient with respect to $\hat{y}$.

This will take a single gradient value, and return the average gradient with respect to $\hat{y}$. If $\hat{y}$ is more than 1 dim it will return a multidimensional array of values which are the average gradients in those dims.

$\frac{d}{d\hat{y}}(\text{MSE}) = \sum_{i=0}^{N-1} -2(y - \hat{y})$

property cost
Get computational cost of the forward function.

forward(*signal=None*, *y: Optional[numpy.ndarray] = None*, *y_hat: Optional[numpy.ndarray] = None*)
Calculate the loss of the output given the ground truth.

This will take multiple values for both y and $\hat{y}$, and return a single value that is the mean of their absolute difference.

$\text{MSE} = \frac{\sum_{i=0}^{N-1} (y - \hat{y})^2}{N}$

update()
Do nothing as there are no parameters to update.

updates()
Do nothing as there are no parameters to update.

1.13.4 Loss Abstraction

Neural Network Loss Functions.

class fhez.nn.loss.loss.**Loss**
Abstract loss class to unify loss function format.

abstract backward(*gradient: numpy.ndarray*)
Calculate gradient of loss with respect to $\hat{y}$.

property cache
Get caching dictionary of auxilary data.

disable_cache()
Disable caching.

enable_cache()
Enable caching.

abstract forward(*signal: numpy.ndarray*, *y: numpy.ndarray*, *y_hat: numpy.ndarray*)
Calculate loss(es) given one or more truths.

property inputs
Get cached input stack.

Neural networks backpropogation requires cached inputs to calculate the gradient with respect to x and the weights. This is a utility method that initialises a stack and allows you to easily append or pop off of it so that the computation can occur in FILO.

property is_cache_enabled
Get status of whether or not caching is enabled.

update()
Loss funcs have no params so do nothing.

updates()
Loss funcs have no params so do nothing.

1.14 Operations

Operations are low level math operations such as summation, and cross correlation.

Table 6: Operations Status

Category	Name	Docs	Forward	Backward
operation	Cross Correlation	*Cross Correlation (CC)*		
operation	Sum	*Maths Sum \sum*		
operation	Enqueue	*Enqueue*		
operation	Dequeue	*Dequeue*		

1.14.1 Cross Correlation (CC)

Our implementation API:

Convolutional Neural Network (CNN) as node abstraction.

class fhez.nn.operations.cc.**CC**(*weights: Optional[numpy.ndarray] = None*, *stride: Optional[numpy.ndarray] = None*, *bias: Optional[numpy.ndarray] = None*, *optimiser=None*)
Convolutional Neural Network.

property b
Shorthand for bias.

backward(*gradient: numpy.ndarray*)
Compute computational filter gradient and input gradient.

property bias
Get sum of products bias coefficient.

property cost
Get computational cost of this CNN node.

forward(*x: numpy.ndarray*)
Compute convolutional filter forward pass and sums.

probe_shape(*lst: list*)
Get the shape of a list, assuming each sublist is the same length.

This function is recursive, sending the sublists down and terminating once a type error is thrown by the final point being a non-list

property schema
Get Marshmallow schema representation of this class.

Marshmallow schemas allow for easy and trustworthy serialisation and deserialisation of arbitrary objects either to inbulit types or json formats. This is an inherited member of the abstract class Serialise.

Note: Anything not listed here will inevitably be lost, ensure anything important is identified and expressley stated its type and structure.

---

property stride
Get stride over convolutions.

update()
Update node state/ weights for a single example.

updates()
Update node state/ weights for multiple examples simultaneously.

property w
Shorthand for weights.

property weights
Get cross convolved filter n-dimensional weights or error.

windex(*data: list*, *filter: list*, *stride: list*, *dimension: int = 0*, *partial: list = []*)
Recursive window index or Windex.

This function takes 3 lists; data, filter, and stride. Data is a regular multidimensional list, so in the case of a 32x32 pixel image you would expect a list of shape (32,32,3) 3 being the RGB channels. Filter is the convolutional filter which we seek to find all windows of inside the data. So for data (32,32,3) a standard filter could be applied of shape (3,3,3). Stride is a 1 dimensional list representing the strides for each dimension, so a stride list such as [1,2,3] on data (32,32,3) and filter (3,3,3), would move the window 1 in the first 32 dimension, 2 in the second 32 dim, and 3 in the 3 dimension.

This function returns a 1D list of all windows, which are themselves lists. These windows are the same length as the number of dimensions, and each dimension consists of indexes with which to slice the original data to create the matrix with which to convolve (cross correlate). An example given: data.shape=(4,4), filter.shape=(2,2), stride=[1,1]

```
list_of_window_indexes = [
    [[0, 1], [0, 1]], # 0th window
    [[0, 1], [1, 2]], # 1st window
    [[0, 1], [2, 3]], # ...
    [[1, 2], [0, 1]],
    [[1, 2], [1, 2]],
    [[1, 2], [2, 3]],
```

(continues on next page)

(continued from previous page)

```
    [[2, 3], [0, 1]],
    [[2, 3], [1, 2]],
    [[2, 3], [2, 3]], # T_x-1 window
]
```

We get the indexes rather than the actual data for two reasons:

- we want to be able to cache this calculation and use it for homogenus data that could be streaming into a convolutional neural networks, cutting the time per epoch down.
- we want to use pure list slicing so that we can work with non- standard data, E.G Fully Homomorphically Encrypted lists.

windex_to_slice(*window*)
Convert x sides of window expression into slices to slice np.

property windows
Get current list of windows for cross correlation.

1.14.2 One-Hot Decode

One hot decoder as computational node.

class fhez.nn.operations.one_hot_decode.**OneHotDecode**
Encode value in one-hot encoded sparse array.

backward(*gradient: numpy.ndarray*)
Map only the gradient of the encoded backward.

property cost
Get non-cost of this node.

forward(*x: numpy.ndarray*)
Encode input to sparse matrix.

property schema
Get marshmallow serialisation schema.

update()
Do nothing as nothing to update.

updates()
Do nothing as nothing to update.

1.14.3 Decrypt

Generic decryptor as computational graph node.

class fhez.nn.operations.decrypt.**Decrypt**
Generic decryptor of inputs.

backward(*gradient*)
Pass gradients back unmodified.

property cost
Return no depth/ cost/ **0** of decryption.

forward(*x*)
Decrypt cyphertext using numpy ufunc API.

update()
Do nothing as decryption has no deep-learning parameterisation.

updates()
Do nothing as decryption has no deep-learning parameterisation.

1.14.4 Dequeue

Dequeue as computational node.

Dequeue as a node, forward is dequeue, backward is dequeue, the exact inverse of the dequeue node as we name it for the forward pass.

See: Comp-sci queues

class fhez.nn.operations.dequeue.**Dequeue**(*length=None*)
Stack multiple arrays together until a given shape is achieved.

backward(*gradient: numpy.ndarray*)
Accumulate inputs into a single queue, then return when full.

property cost
Get **0** cost for dequeueing arrays.

forward(*x*)
Distribute input to respective outputs in order via yield.

Effectiveley backward is a dequeue but for gradients.

> **Warning:** This **YIELDS** gradients unlike most nodes, requiring special logic by a network traverser, only getting one input but results in many outputs.

property length
Get the desired length of the final dequeue.

property queue
Get the current queue.

update()
Update nothing as dequeueing is not parameterisable.

updates()
Update nothing as dequeueing is not parameterisable.

1.14.5 One-Hot Encode

One hot encoder as computational node.

class fhez.nn.operations.one_hot_encode.**OneHotEncode**(*length=None*)
Encode value in one-hot encoded sparse array.

backward(*gradient: numpy.ndarray*)
Map only the gradient of the encoded backward.

property cost
Get non-cost of this node.

forward(*x: numpy.ndarray*)
Encode input to sparse matrix.

property length
Get length of sparse matrix to be generated.

property schema
Get marshmallow serialisation schema.

update()
Do nothing as nothing to update.

updates()
Do nothing as nothing to update.

1.14.6 Encrypt

Generic encryptor as computational graph node.

class fhez.nn.operations.encrypt.**Encrypt**(*provider=None*, ***kwargs*)
Generic encryptor of inputs.

backward(*gradient*)
Pass gradients back unmodified.

property cost
Return no depth/ cost/ **0** of encryption.

forward(*x*)
Encrypt cyphertext using configured FHE provider.

property parameters
Get parameters to for the encryption provider.

property provider
Get encryption provider to be parameterised for encryption.

update()
Do nothing as encryption has no deep-learning parameterisation.

updates()
Do nothing as encryption has no deep-learning parameterisation.

1.14.7 Enqueue

Enqueue as computational node.

Enqueueing is the process of taking two or more arrays and stacking them together using some meta container that encapsulates both arrays, and enqueues->dequeue in first in first out manner (FIFO).

While the queue is still being enqueued this node will return nothing. Once the queue has reached the desired length, it will return the queue as a list. This will map gradients again in FIFO manner using a dequeue.

Enqueue as a node, forward is enqueue, backward is dequeue, the exact inverse of the Dequeue node as we name it for the forward pass.

See: Comp-sci queues

class fhez.nn.operations.enqueue.**Enqueue**(*length=None*)
Stack multiple arrays together until a given shape is achieved.

backward(*gradient*)
Distribute gradient to respective inputs in order via yield.

Effectiveley backward is a dequeue but for gradients.

> **Warning:** This **YIELDS** gradients unlike most nodes, requiring special logic by a network traverser, only getting one input but results in many outputs.

property cost
Get **0** cost for enqueueing arrays.

forward(*x*)
Accumulate inputs into a single queue, then return when full.

property length
Get the desired length of the final enqueue.

property queue
Get the current queue.

update()
Update nothing as enqueueing is not parameterisable.

updates()
Update nothing as enqueueing is not parameterisable.

1.14.8 Rotate

Rotate nodes have 3 modes of operation, an encryptor, decryptor, or rotator. Rotate.forward() can, depending on configuration, transform data plaintext->cyphertext, cyphertext->cyphertext, cyphertext->plaintext, plaintext->plaintext. Rotate always keeps the original data shape, but can change which axis is encrypted. For instance a single cyphertext of shape (10,32,32) can come out as a list of (10,) cyphertexts of shape (32,32).

- node provider/ encryptor non-configured: input will be converted to plaintext, thus cyphertext->plaintext, and plaintext->plaintext mode
- node provider/ encryptor configured: input will be encrypted with new keys based on given axis, thus plaintext->cyphertext, cyphertext->cyphertext2

This node can thus be used as a replacement for both encrypt and decrypt nodes. The only caveat being those nodes being more specialised give you slightly more warnings. For instance if an encryption node if left unconfigured it makes sense to warn you that it will be operating in plaintext. However if this node is left unconfigured it may be the case that you want it to be a decryption node, thus we don't warn you.

Rotate API

Generic cyphertext key rotation as abstract node.

class fhez.nn.operations.rotate.**Rotate**(*axis=None*, *encryptor=None*, *provider=None*, *sum_axis=None*, *flatten=None*, ***kwargs*)

Generic cyphertext key rotation abstraction.

property axis
Get axis of key rotation.

backward(*gradient*)
Pass gradients back unmodified.

property cost
Return no depth/ cost/ **0** of encryption.

property encryptor
Encryption parameterised object.

property flatten
Get if intermediary should be flattened to 1D flag.

forward(*x*)
Rotate keys using encryption provider on desired axis.

This function has 3 modes:

- Encryptor: given some provider or encryptor will encrypt x and return the cyphertext or list of cyphertexts (if axis is not 0)
- Decryptor: given neither provider nor encryptor will just turn x into a numpy array, which for our numpyapi implementation means turning cyphertexts into plaintexts and plaintexts stay plaintexts
- Rotator: given a cyphertext-x and a provider or encryptor will decrypt x and re-encrypt using the new provider on the desired axis.

property parameters
Get parameters to for the encryption provider.

property provider
Get encryption provider to be parameterised for encryption.

property sum_axis
Get the desired axis to be summed between rotation.

update()
Do nothing as encryption has no deep-learning parameterisation.

updates()
Do nothing as encryption has no deep-learning parameterisation.

1.14.9 Maths Sum $\sum$

Our implementation API:

Maths sum as computational node.

class fhez.nn.operations.sum.**Sum**

Sum inputs and distribute backprop.

First dim of inputs shape e.g (64,32,32,3) to sum are summed along axis=0 resulting in an output of (32,32,3) and returning gradients of (64,)

backward(*gradient: numpy.ndarray*)

Distribute gradient to inputs.

property cost

Get computational cost of this node.

forward(*x: numpy.ndarray*)

Sum inputs together assuming first dim is inputs.

update()

Do nothing since sum is not parameterisable.

updates()

Do nothing since sum is not parameterisable.

CHAPTER

TWO

ALSO SEE

This is part of a larger body of related together work. We are building similar tools for go (DarkLantern). We are also building fully open-source infastructure to run FHE using the server-client model, as well as offering this as a service to others (DeepCypher GitLab, DeepCypher.me)

PYTHON MODULE INDEX

f

INDEX

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

U

W