ArrayLSTM

Release 0.0.1

Thijs van Ede

Jul 29, 2022

CONTENTS:

ArrayLSTM provides a pytorch implementation of *Recurrent Memory Array Structures by Kamil M Rocki*. This code was implemented as part of the IEEE S&P 2022 **`DeepCASE: Semi-Supervised Contextual Analysis of Security Events`_** paper. We ask people to cite both works when using the software for academic research papers, see Citing for more information.

Recurrent Memory Array Structures by Kamil M Rocki: https://arxiv.org/abs/1607.03085

IEEE S&P 2022 DeepCASE: Semi-Supervised Contextual Analysis of Security Events paper.: https://vm-thijs.ewi.utwente.nl/static/homepage/papers/deepcase.pdf

CHAPTER ONE

INSTALLATION

The most straigtforward way of installing ArrayLSTM is via pip

```
pip install array-lstm
```

1.1 From source

If you wish to stay up to date with the latest development version, you can instead download the source code. In this case, make sure that you have all the required *dependencies* installed.

Once the dependencies have been installed, run:

```
pip install -e <path/to/directory/containing/arraylstm/setup.py>
```

1.2 Dependencies

ArrayLSTM requires the following python packages to be installed:

- pytorch: https://pytorch.org/

All dependencies should be automatically downloaded if you install ArrayLSTM via pip. However, should you want to install these libraries manually, you can install the dependencies using the requirements.txt file

```
pip install -r requirements.txt
```

Or you can install these libraries yourself

```
pip install -U torch
```

CHAPTER

TWO

USAGE

This section gives a high-level overview of the modules implemented by ArrayLSTM and provides way of extending them. We also include several working examples to guide users through the code. For detailed documentation of individual methods, we refer to the *Reference* guide.

2.1 Overview

This section explains the modular design of ArrayLSTM on a high level. Figure 1 provides an overview of the different modules used in the ArrayLSTM implementation.

All LSTM variants are subclasses of the `nn.Module` class. Therefore, they all implement the `forward(X, hidden)` method. Or alternatively, the class implements the `__call__()` method, which subsequently calls the `forward(X, hidden)` method. Figure 2 gives an overview of how the forward method is implemented. Note that the dashed block `update_hidden()` is only called in the ArrayLSTM variants, not the regular LSTM.

2.2 Code

To use ArrayLSTM into your own project, you can use it as a standalone module. Here we show some simple examples on how to use the ArrayLSTM package in your own python code. For a complete documentation we refer to the *Reference* guide.

2.2.1 Import

To import components from ArrayLSTM simply use the following format

```
from arrayLSTM import <Object>
from arrayLSTM.extensions import <Object>
```

For example, the following code imports the different LSTM objects as found in the *Reference*.

```
# Imports
from arrayLSTM            import LSTM
from arrayLSTM            import ArrayLSTM
from arrayLSTM.extensions import AttentionArrayLSTM
from arrayLSTM.extensions import StochasticArrayLSTM
```

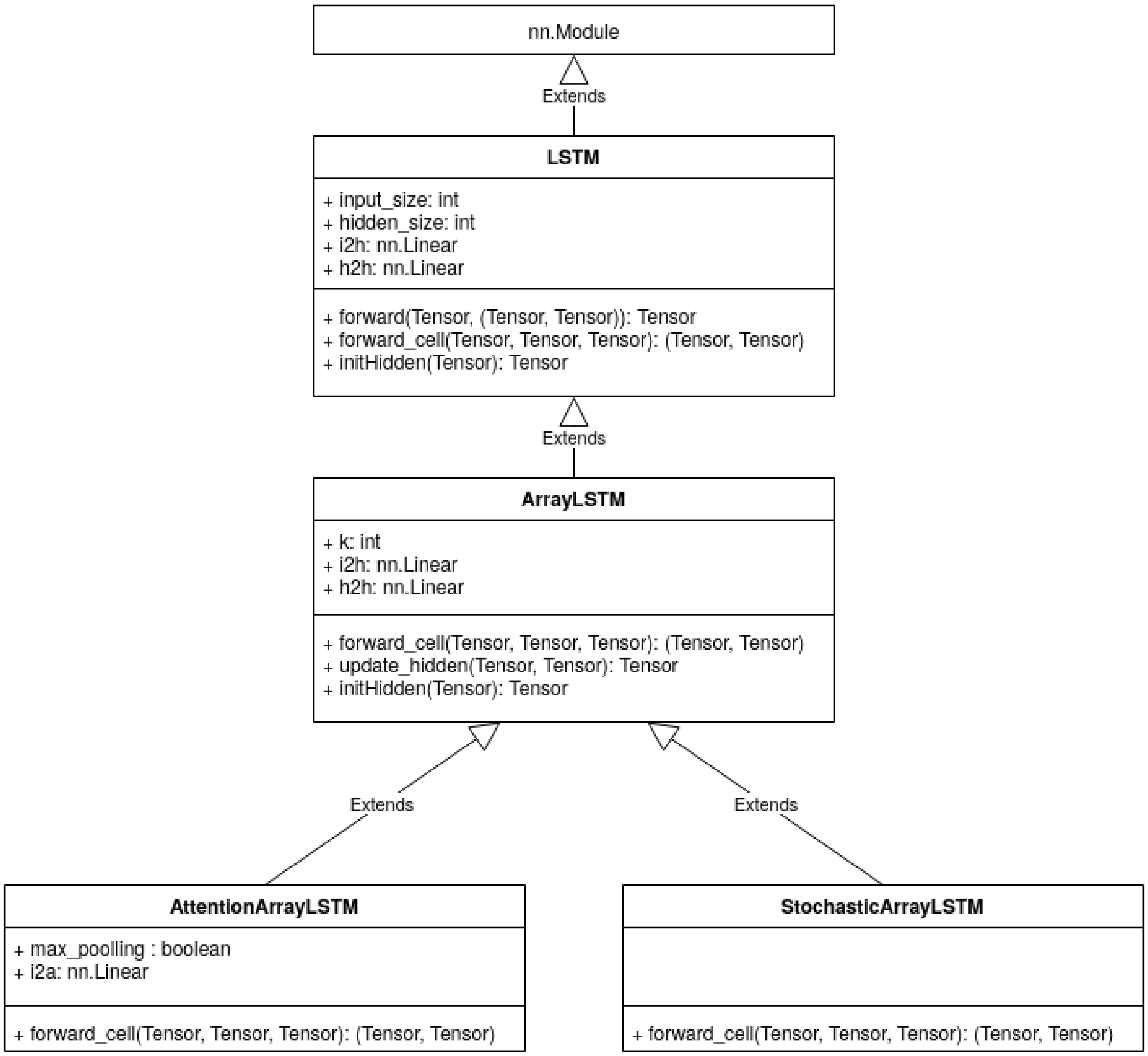
nn.Module
Extends
LSTM
+ input_size: int
+ hidden_size: int
+ i2h: nn.Linear
+ h2h: nn.Linear
+ forward(Tensor, (Tensor, Tensor)): Tensor
+ forward_cell(Tensor, Tensor, Tensor): (Tensor, Tensor)
+ initHidden(Tensor): Tensor
Extends
ArrayLSTM
+ k: int
+ i2h: nn.Linear
+ h2h: nn.Linear
+ forward_cell(Tensor, Tensor, Tensor): (Tensor, Tensor)
+ update_hidden(Tensor, Tensor): Tensor
+ initHidden(Tensor): Tensor
Extends
Extends
AttentionArrayLSTM
+ max_poolling : boolean
+ i2a: nn.Linear
+ forward_cell(Tensor, Tensor, Tensor): (Tensor, Tensor)
StochasticArrayLSTM
+ forward_cell(Tensor, Tensor, Tensor): (Tensor, Tensor)

Fig. 1: Figure 1: Overview of modules in class diagram. Note that this diagram only contains attributes and methods that are overwritten in the respective classes.

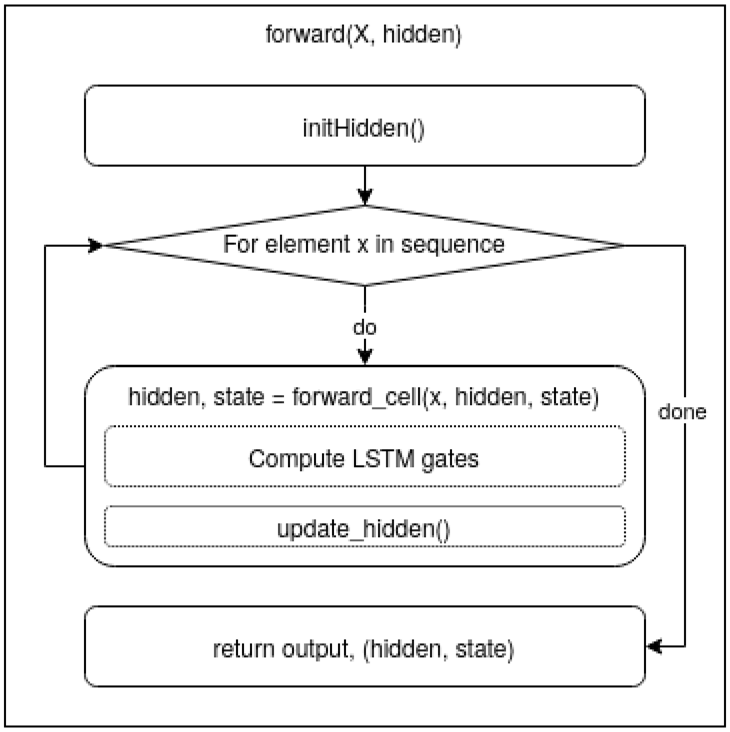
forward(X, hidden)
initHidden()
For element x in sequence
do
hidden, state = forward_cell(x, hidden, state)
Compute LSTM gates
update_hidden()
done
return output, (hidden, state)

Fig. 2: Figure 2: Control flow graph of methods called for each sequence. Note that the update_hidden() method is only called in the ArrayLSTM variants, not the regular LSTM.

2.2.2 Working example

In this example, we import all different LSTM implementations and use it to predict the next item in a sequence. First we import the necessary torch modules and different LSTMs that we want to use.

```
# Torch imports
import torch
import torch.nn as nn
import torch.nn.functional as F

# ArrayLSTM imports
from arrayLSTM            import LSTM
from arrayLSTM            import ArrayLSTM
from arrayLSTM.extensions import AttentionArrayLSTM
from arrayLSTM.extensions import StochasticArrayLSTM
```

Second, we generate some random data

```
# Parameters to use
n_samples   = 1024
seq_length  = 10
size_input  = 10
size_hidden = 128
size_output = 10
k           = 4

```

(continues on next page)

(continued from previous page)

```
# Generate random input data
X = (size_input*torch.rand((n_samples, seq_length))).to(torch.int64)
```

Next, we create a Neural Network with our LSTM of choice. Please note that this is a very simple example in which we show how the StochasticArrayLSTM can be used as a simple module.

```
class MyNetwork(nn.Module):

  def __init__(self, size_input, size_hidden, size_output, k):
    # Call super method
    super().__init__()

    # Set variables
    self.size_input  = size_input
    self.size_hidden = size_hidden
    self.size_output = size_output
    self.k           = k

    # Initialise layers
    self.lstm    = StochasticArrayLSTM(size_input, size_hidden, k) # Use any LSTM of␣
↪your choosing
    self.linear  = nn.Linear(size_hidden, size_output)
    self.softmax = nn.LogSoftmax(dim=1)

  def forward(self, X):
    # One-hot encode input - transforms input into one-hot-encoded input
    encoded = F.one_hot(X, self.size_input).to(torch.float32)

    # Pass through LSTM layer
    out, (hidden, state) = self.lstm(encoded)
    # Take hidden state as output
    hidden = hidden.squeeze(0)

    # Pass through linear layer
    out = self.linear(hidden)
    # Perform softmax and return
    return self.softmax(out)
```

Finally, we can call the network using the data and perform further training, which we leave up to the user.

```
# Create an instance of MyNetwork
net = MyNetwork(size_input, size_hidden, size_output, k)
# Pass the input data X through the network
output = net(X)
```

CHAPTER

THREE

REFERENCE

This is the reference documentation for the classes and methods objects provided by the ArrayLSTM module.

3.1 LSTM

As a basis, we provide a pure pytorch implementation of the LSTM module. This extends the regular torch.nn.Module interface.

class `lstm.LSTM`(**args: Any*, ***kwargs: Any*)

LSTM implementation in pytorch

Note: This is a *batch_first=True* implementation, hence the *forward()* method expect inputs of *shape=(batch, seq_len, input_size)*.

`input_size`

Size of input dimension

Type
int

`hidden_size`

Size of hidden dimension

Type
int

`i2h`

Linear layer transforming input to hidden state

Type
nn.Linear

`h2h`

Linear layer updating hidden state to hidden state

Type
nn.Linear

3.1.1 Initialization

`LSTM.`**`__init__`**(*input_size*, *hidden_size*)

LSTM implementation in pytorch

Note: This is a *batch_first=True* implementation, hence the *forward()* method expect inputs of *shape=(batch, seq_len, input_size)*.

Parameters

- **`input_size`** (*`int`*) – Size of input dimension
- **`hidden_size`** (*`int`*) – Size of hidden dimension

3.1.2 Forward

As all nn.Module objects, the LSTM implements a `forward()` method. This method forwards all sequences in x through the `forward_cell()` method.

`LSTM.`**`forward`**(*x*, *hidden=None*)

Forward all sequences through the network.

Parameters

- **`x`** (*`torch.Tensor of shape=(batch, seq_len, input_size)`*) – Tensor to pass through network
- **`hidden`** (*`tuple`*) – Tuple consisting of (hidden, state) to use as initial vector. If None is given, both hidden and state vectors will be initialised as the 0 vector.

  ***hidden* torch.Tensor of shape (batch, input_size), default=0 vector**
  Tensor containing the hidden state

  ***state* torch.Tensor of shape (batch, input_size), default=0 vector**
  Tensor containing the cell state

Returns

- **outputs** (*torch.Tensor of shape=(batch, seq_len, hidden_size)*) – Outputs for each input of sequence
- **hidden** (*tuple*) – Tuple consisting of (hidden, state) of final output.

  ***hidden* torch.Tensor of shape (batch, output)**
  Tensor containing the hidden state

  ***state* torch.Tensor of shape (batch, output)**
  Tensor containing the cell state

A single LSTM cell is implemented by the `forward_cell()` method. Note that this method is also overwritten by subclasses to implement their custom forward methods.

`LSTM.`**`forward_cell`**(*x*, *hidden*, *state*)

Perform a single forward pass through the network.

Parameters

- **`x`** (*`torch.Tensor of shape=(batch, input_size)`*) – Tensor to pass through network

- **hidden** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the hidden state
- **state** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the cell state

Returns

- **hidden** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the next hidden state
- **state** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the next cell state

3.1.3 Hidden state

The LSTM provides a method for initializing the hidden state and cell state. Note that this method is also overwritten by subclasses to implement their custom cell initializations.

`LSTM.`**`initHidden`**(*x*)

Initialise hidden layer

3.2 ArrayLSTM

The ArrayLSTM implements the basic ArrayLSTM of Rocki's Recurrent Memory Array Structures. It module is build as an extension of the normal *LSTM* implementation.

`class` `arraylstm.`**`ArrayLSTM`**(**args: Any*, ***kwargs: Any*)

Implementation of ArrayLSTM

From Recurrent Memory Array Structures by Kamil Rocki

Note: This is a *batch_first=True* implementation, hence the *forward()* method expect inputs of *shape=(batch, seq_len, input_size)*.

`input_size`

Size of input dimension

Type

int

`hidden_size`

Size of hidden dimension

Type

int

`k`

Number of parallel memory structures, i.e. cell states to use

Type

int

`i2h`

Linear layer transforming input to hidden state

Type

nn.Linear

`h2h`

Linear layer updating hidden state to hidden state

Type

nn.Linear

3.2.1 Initialization

`ArrayLSTM.__init__`(*input_size*, *hidden_size*, *k*)

Implementation of ArrayLSTM

Note: This is a *batch_first=True* implementation, hence the *forward()* method expect inputs of *shape=(batch, seq_len, input_size)*.

Parameters

- **`input_size`** (*int*) – Size of input dimension
- **`hidden_size`** (*int*) – Size of hidden dimension
- **`k`** (*int*) – Number of parallel memory structures, i.e. cell states to use

3.2.2 Forward

A single ArrayLSTM cell is implemented by the `forward_cell()` method. This method overwrites its *LSTM* superclass.

`ArrayLSTM.forward_cell`(*x*, *hidden*, *state*)

Perform a single forward pass through the network.

Parameters

- **`x`** (*torch.Tensor of shape=(batch, input_size)*) – Tensor to pass through network
- **`hidden`** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the hidden state
- **`state`** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the cell state

Returns

- **hidden** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the next hidden state
- **state** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the next cell state

As variations of the ArrayLSTM update their hidden state differently, we also add a method `forward_cell()`. This method can be overwritten by subclasses to update the hidden state in different ways.

`ArrayLSTM.update_hidden`(*outputs*, *states*)

Default hidden state as sum of outputs and cells

Parameters

- **`outputs`** (*torch.Tensor of shape=(k, batch_size, hidden_size)*) – Tensor containing the result of output gates o

- **states** (*torch.Tensor of shape=(k, batch_size, hidden_size)*) – Tensor containing the cell states

Returns
hidden – Hidden tensor as computed from outputs and states

Return type
torch.Tensor of shape=(1, batch_size, hidden_size)

3.2.3 Hidden state

The ArrayLSTM requires multiple cell states instead of a single one, therefore it overwrites it super method from *LSTM*.

`ArrayLSTM.`**`initHidden`**`(`*x*`)`
Initialise hidden layer

3.3 AttentionArrayLSTM

The AttentionArrayLSTM implements an ArrayLSTM with Deterministic Array-LSTM extension "Lane selection: Soft Attention" of Rocki's Recurrent Memory Array Structures. It module is build as an extension of the basic *ArrayLSTM* implementation.

`class` `extensions.`**`AttentionArrayLSTM`**`(`**args: Any*, ***kwargs: Any*`)`
Implementation of ArrayLSTM with Lane selection: Soft attention

From Recurrent Memory Array Structures by Kamil Rocki

Note: This is a *batch_first=True* implementation, hence the *forward()* method expect inputs of *shape=(batch, seq_len, input_size)*.

`input_size`
Size of input dimension

Type
int

`hidden_size`
Size of hidden dimension

Type
int

`k`
Number of parallel memory structures, i.e. cell states to use

Type
int

`max_pooling`
If True, uses max pooling for attention instead

Type
boolean, default=False

`i2h`

Linear layer transforming input to hidden state

Type
nn.Linear

`h2h`

Linear layer updating hidden state to hidden state

Type
nn.Linear

3.3.1 Initialization

`AttentionArrayLSTM.`**`__init__`**`(`*input_size*, *hidden_size*, *k*, *max_pooling=False*`)`

Implementation of ArrayLSTM with Lane selection: Soft attention

Note: This is a *batch_first=True* implementation, hence the *forward()* method expect inputs of *shape=(batch, seq_len, input_size)*.

Parameters

- **`input_size`** (*`int`*) – Size of input dimension
- **`hidden_size`** (*`int`*) – Size of hidden dimension
- **`k`** (*`int`*) – Number of parallel memory structures, i.e. cell states to use
- **`max_pooling`** (*`boolean, default=False`*) – If True, uses max pooling for attention instead

3.3.2 Forward

The AttentionArrayLSTM overwrites ArrayLSTM's `forward_cell()` method to include an attention mechanism. The API is equivalent to that of *ArrayLSTM*, but the implementations differ.

`AttentionArrayLSTM.`**`forward_cell`**`(`*x*, *hidden*, *state*`)`

Perform a single forward pass through the network.

Parameters

- **`x`** (*`torch.Tensor of shape=(batch, input_size)`*) – Tensor to pass through network
- **`hidden`** (*`torch.Tensor of shape (batch, input_size)`*) – Tensor containing the hidden state
- **`state`** (*`torch.Tensor of shape (batch, input_size)`*) – Tensor containing the cell state

Returns

- **hidden** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the next hidden state
- **state** (*torch.Tensor of shape (batch, input_size)*) – Tensor containing the next cell state

3.4 StochasticArrayLSTM

The StochasticArrayLSTM implements an ArrayLSTM with Non-deterministic Array-LSTM extension "Stochastic Output Pooling" of Rocki's Recurrent Memory Array Structures. It module is build as an extension of the basic *ArrayLSTM* implementation.

class extensions.**StochasticArrayLSTM**(**args: Any*, ***kwargs: Any*)

Implementation of ArrayLSTM with Stochastic Output Pooling

From Recurrent Memory Array Structures by Kamil Rocki

Note: This is a *batch_first=True* implementation, hence the *forward()* method expect inputs of *shape=(batch, seq_len, input_size)*.

input_size

Size of input dimension

Type
int

hidden_size

Size of hidden dimension

Type
int

k

Number of parallel memory structures, i.e. cell states to use

Type
int

i2h

Linear layer transforming input to hidden state

Type
nn.Linear

h2h

Linear layer updating hidden state to hidden state

Type
nn.Linear

3.4.1 Initialization

StochasticArrayLSTM.**__init__**(**args: Any*, ***kwargs: Any*) → None

3.4.2 Forward

The StochasticArrayLSTM overwrites ArrayLSTM's `update_hidden()` method to update the hidden state using stochastic output pooling. The API is equivalent to that of *ArrayLSTM*, but the implementations differ.

`StochasticArrayLSTM.`**`update_hidden`**(*outputs*, *states*)

Update hidden state based on most likely output

Parameters

- **`outputs`** (*`torch.Tensor of shape=(k, batch_size, hidden_size)`*) – Tensor containing the result of output gates o
- **`states`** (*`torch.Tensor of shape=(k, batch_size, hidden_size)`*) – Tensor containing the cell states

Returns

hidden – Hidden tensor as computed from outputs and states

Return type

torch.Tensor of shape=(1, batch_size, hidden_size)

CHAPTER

FOUR

CONTRIBUTORS

This page lists all the contributors to this project. If you want to be involved in maintaining code or adding new features, please email t(dot)s(dot)vanede(at)utwente(dot)nl.

4.1 Code

- Thijs van Ede

4.2 Academic Contributors

- Thijs van Ede
- Hojjat Aghakhani
- Noah Spahn
- Riccardo Bortolameotti
- Marco Cova
- Andrea Continella
- Maarten van Steen
- Andreas Peter
- Christopher Kruegel
- Giovanni Vigna

CHAPTER

FIVE

LICENSE

MIT License

Copyright (c) 2020 Thijs van Ede

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER SIX

CITING

To cite ArrayLSTM please use the following publications:

van Ede, T., Aghakhani, H., Spahn, N., Bortolameotti, R., Cova, M., Continella, A., van Steen, M., Peter, A., Kruegel, C. & Vigna, G. (2022, May). DeepCASE: Semi-Supervised Contextual Analysis of Security Events. In 2022 Proceedings of the IEEE Symposium on Security and Privacy (S&P). IEEE. [PDF DeepCASE]

Rocki, K.M. (2016). Recurrent memory array structures. In arXiv preprint arXiv:1607.03085. [PDF ArrayLSTM]

6.1 Bibtex

6.1.1 DeepCASE

```
@inproceedings{vanede2020deepcase,
  title={{DeepCASE: Semi-Supervised Contextual Analysis of Security Events}},
  author={van Ede, Thijs and Aghakhani, Hojjat and Spahn, Noah and Bortolameotti,␣
↪Riccardo and Cova, Marco and Continella, Andrea and van Steen, Maarten and Peter,␣
↪Andreas and Kruegel, Christopher and Vigna, Giovanni},
  booktitle={Proceedings of the IEEE Symposium on Security and Privacy (S&P)},
  year={2022},
  organization={IEEE}
}
```

6.1.2 ArrayLSTM

```
@article{rocki2016recurrent,
  title={Recurrent memory array structures},
  author={Rocki, Kamil},
  journal={arXiv preprint arXiv:1607.03085},
  year={2016}
}
```

INDEX