University of L'Aquila

Department of Information Engineering, Computer Science and Mathematics

Hyper Heursitic Cryptography with Mixed Adversarial Nets

Author :
Aly SHMAHELL

Supervisor :
Prof. Giovanni DE GASPERIS

June 15, 2018

Abstract

Abstract

1 Introduction

Neural cryptography is an interdisciplinary field in Computer Science, combining both Artificial Intelligence and Cryptography, towards the development of stochastic methods, based on artificial neural networks, for use in encryption and cryptanalysis.

1.1 Paper Objective

The objective of this paper is to explore the use of new developments in the field of Neural Networks, mainly Adversarial Neural Networks and Convolutional Neural Networks, as a generative model, to produce a new breed of Crypto-Systems.

The work being done here is based on a new paper released in 2016 from Google's Deep-Mind, which promises to bridge the gap between research and application in this area.

The goal of my research is to provide the following:

- an addition to the variety of methods provided in the original paper.
- an improvement in performance of the models being built.
- an in-depth analysis of how the components work, and the inner details of their mechanics.
- a software documentation that provides a blueprint for a more software-engineering oriented neural-cryptosystem prototype.

With the research paper specter in this area being dominated by authors coming from a mathematical-background angle, my paper aims to provide:

- a Computer Science oriented approach to solving cryptography with neural networks and stochastic methods.

This paper finally adds the following:

- the introduction of hybrid neural crypto-systems.
- an exploration into adding hyper heuristics to the field.

1.2 Paper Motivation

The question of motivation can be empirically divided into:

- Why is the idea of importance?
- Why does the author think it would work?

Why is Neural Cryptography of importance?
The age of intelligent machines is comprised of multiple intricate components, but individually they function narrowly even for the simplest of tasks, but the surge of incorporation of these multiple components into the backbone of neural-nets has put artificial intelligence on a fast track towards competence in multiple complex areas of problem solving, surpassing human intelligence by multitudes on many occasions.

It makes sense from an academic perspective that we want neural nets to incorporate an understanding of cryptography.
It also makes sense from an economic and existential point that we want these backbones of machine intelligence to parallel their success in surpassing human intelligence on cryptography, in order to open the door for new opportunities in that area.

Why would Neural Cryptography work?
This boils down to multiple general factors:

- The incorporation of multiple heuristic methodologies into one unified backbone, which made it tackle a rapidly growing heap of complex tasks, cryptography is just another human invention to be caught up with.
- The increasing successful research into using these heuristics not just to compute a complex task, but to do it quickly without loss in accuracy.
- The introduction of tools, frameworks and libraries of industrial level to the public which propelled the field of neural networks and made it ever so easy to replicate experiments and improve upon them.

Which lead to those paper-related factors:

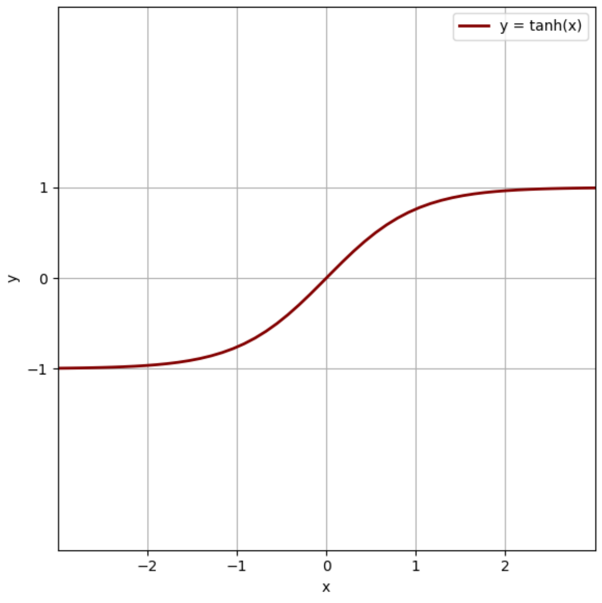
- The introduction of Convolutional networks provides a well tested and understood methodology in reducing problems where local spatial relations in the data matter, which is the case for cryptography and cryptanalysis.
- Having a Mixed Convolutional Net with a fully connected layer will teach the network to account for global spatial relations as well.
- A result of using Convolutions is that the small-sized pattern-finding filter has shared weights (and biases) for all spatial locations which the convolution processes, which reduces the compute-power required for the whole process compared to other network models.
- Adversarial computation has been proven to be effective for years in the form of Genetic Algorithms, and adding adversary as a non-supervised generative model provides a better and easier experiment on how to synthesize a new form of cryptography.

1.3 Xavier Initialization [1]

When we talk about initialization, we mean the initial value which should be given to a certain neuron, this initial value is important for the neuron and for the network in general.

Lemma 1 *Suppose our net uses the hyperbolic tangent activation function for its neurons:*

Fig. 1: example plot of the hyperbolic tangent



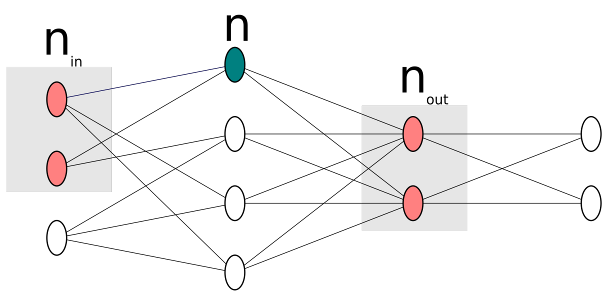
- *If the weights start too small, then the signal shrinks as it passes through each layer until it vanishes, then as it passes deeper in the network with its small values, the layers it enter will become linear, because the output of the hyperbolic tangent is linear with small input values, this means the deeper layers of the net will loose non-linearity.*
- *If the weights start too large, then the signal grows as it passes through each layer until it becomes too large, then as it passes deeper in the network with its large values, the layers it enter will become saturated, as the output of the hyperbolic tangent is flat with large input values, and this flatness will cause the gradient to become zero, and we will get the vanishing gradient problem.*

Lemma 2 *Having a pre-defined net graph: for each neuron we know the number of inputs and the number of outputs, therefor we can calculate a reasonable weight for the neuron in question based on a normal distribution of a zero mean and a 1/n variance.*

Lemma 3 *To achieve initialization while avoiding the two obstacles in Lemma 1, we want the variance to remain the same with each passing layer.*

Suppose we have an input X from a previous layer with n components and a linear neuron with random weights W in the current layer that spits out the same output Y to some neurons in the next layer. The output of the neuron will have the following equation:

Fig. 2: Xavier Initialization on neuron "n"



$$Y = W_1X_1 + W_2X_2 + ... + W_nX_n \tag{1}$$

To calculate the variance of each component:

$$Var(W_iX_i) = E[X_i]^2 Var(W_i) + E[W_i]^2 Var(X_i) + Var(W_i)Var(X_i) \tag{2}$$

Since our inputs and weights come from a normal distribution of zero mean (from Lemma 2):

$$E[X_i]^2 Var(W_i) + E[W_i]^2 Var(X_i) = 0 \implies Var(W_iX_i) = Var(W_i)Var(X_i) \tag{3}$$

Since the neurons in the same previous layer are all independent, we assume that both X_i and W_i are independent and also identically distributed:

$$Var(Y) = Var(W_1X_1 + W_2X_2 + ... + W_nX_n) = nVar(W_i)Var(X_i) \tag{4}$$

In the last equation, we have the variance of the inputs, the variance of the output and the variance of the weights, now we can calculate the variance of the weights from Lemma 3:

$$Var(Y) = Var(X_i) \implies Var(W_i) = \frac{1}{n_{in}} \implies Var(W_i) * n_{in} = 1 \tag{5}$$

Now if we go through the same derivation for back-propagation, we get:

$$Var(W_i) = \frac{1}{n_{out}} \implies Var(W_i) * n_{out} = 1 \tag{6}$$

To keep the variance of the input gradient & the output gradient the same, we combine (5) & (6) and we get:

$$(n_{out} + n_{in}) * Var(W_i) = 2 \implies Var(W_i) = \frac{2}{n_{in} + n_{out}} \tag{7}$$

1.4 1D Convolution on Batch 1D Data with (1,2)-D Filters

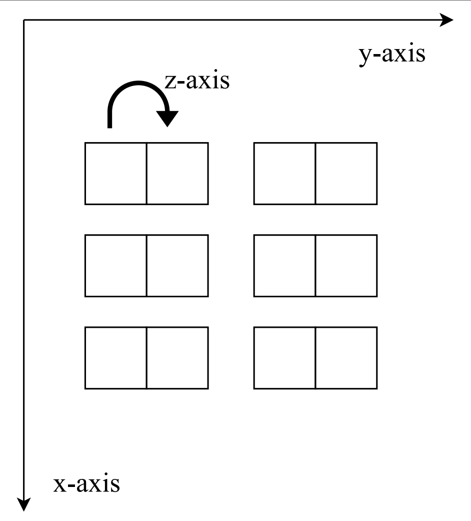
1D Convolution works by sliding along 1 axis, then performing matrix multiplication, therefor in order to perform 1D convolution with a 2D filter, we need to add another axis to our data.

However, if our data is a batch of samples, then each sample has 1D convolution performed on it separately, that calls for adding another axis to both our data and our filter, in order to re-do the convolution on each data sample separately.

Therefor we end up with a 3D tensor for the batch data, and another 3D tensor for the filter.

To illustrate how this works, I've chosen a flat representation of the 3D tensors (representing 3D with 2D), mainly because this is how it's done with Numpy, and this is more practical for practitioners.

Fig. 3: Flat Axis Representation of 3D Tensors - Numpy Style



1D Convolution slides over the y-axis for one convolution on a 1D sample, everything on the same index on the y-axis belongs to that one sample, and the multiplication is performed for that sample independent of the others.

On the other hand, 1D convolution considers the filter as a whole, and slides over the x-axis of the filter when convolving on one data sample.

The multiplication is done by multiplying the z-axis from the data sample by the y-axis from the filter, along the x-axis of the filter.

Then sliding is performed, and the multiplication is applied again.

when the filter cannot multiply anymore (out of reach), then convolution over this sample is over.

Once that happens, the filter is slided to its default position, then we slide over the x-axis to perform 1D convolution again on another data sample from the batch.

The following is an example of 1D convolution over a batch of 1D data with a 2D filter:

The example starts off by generating 1D input strings.

Fig. 4: First Input String

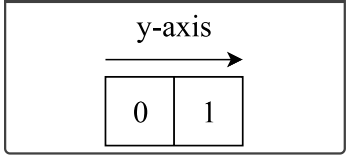


Fig. 5: Second Input String

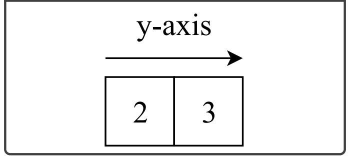
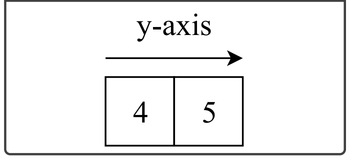
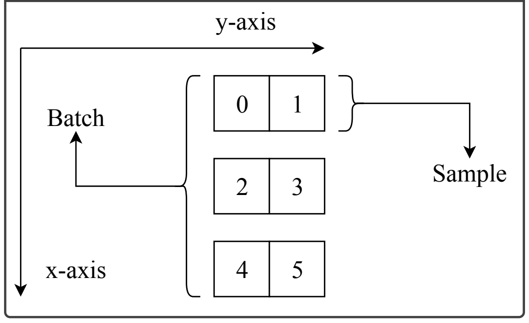


Fig. 6: Third Input String



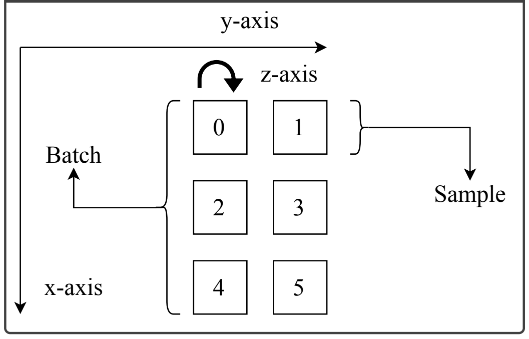
Input strings are then stacked up along the x-axis to form a 2D batch.

Fig. 7: Input Batch



Then the batch gets expanded from 2D to 3D along the y-axis (injecting z-axis into y-axis), which means the final batch shape becomes (3, 2, 1).

Fig. 8: Reshaped Input Batch



Then finally, a filter of shape (2, 1, 2) is provided for the convolution.

Fig. 9: Filter #1 with arbitrary weights

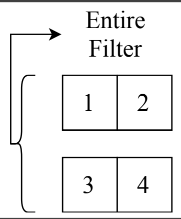
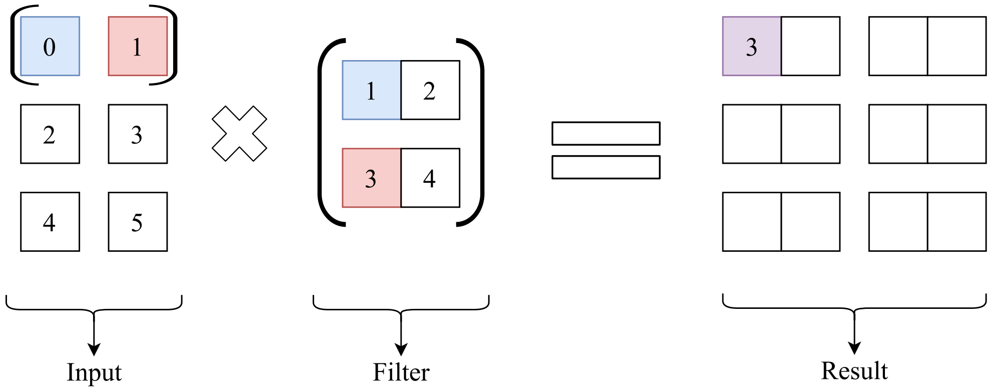


Fig. 10: Applying the filter to the first sample in the input batch - generating the first value for the first in the first feature map.

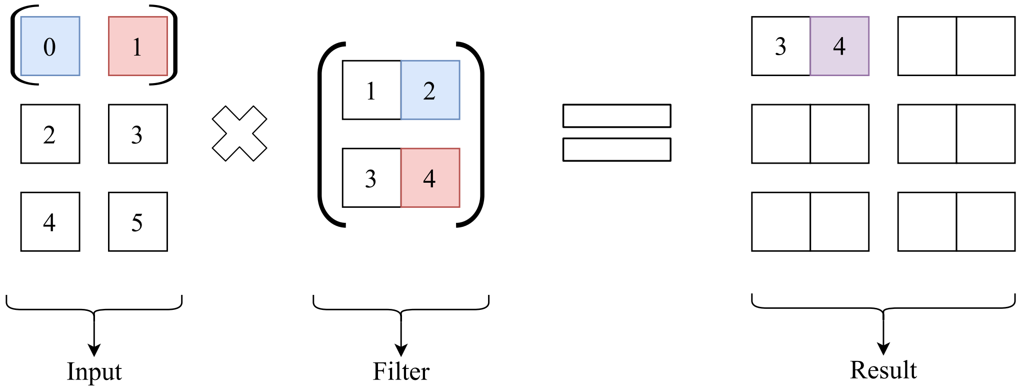


$$\boxed{\begin{aligned} \mathit{Result}[0][0][0] &= \mathit{Input}[0][0][0] * \mathit{Filter}[0][0][0] \\ &+ \mathit{Input}[0][1][0] * \mathit{Filter}[1][0][0] \end{aligned}}$$

$$\downarrow$$

$$\boxed{3 = 0 * 1 + 1 * 3}$$

Fig. 11: Applying the filter to the first sample in the input batch - generating the second value in the first feature map.



$$\boxed{\begin{aligned} \mathit{Result}[0][0][1] &= \mathit{Input}[0][0][0] * \mathit{Filter}[0][0][1] \\ &+ \mathit{Input}[0][1][0] * \mathit{Filter}[1][0][1] \end{aligned}}$$

$$\downarrow$$

$$\boxed{4 = 0 * 2 + 1 * 4}$$

Fig. 12: Applying the filter to the first sample in the input batch - generating the second feature map.

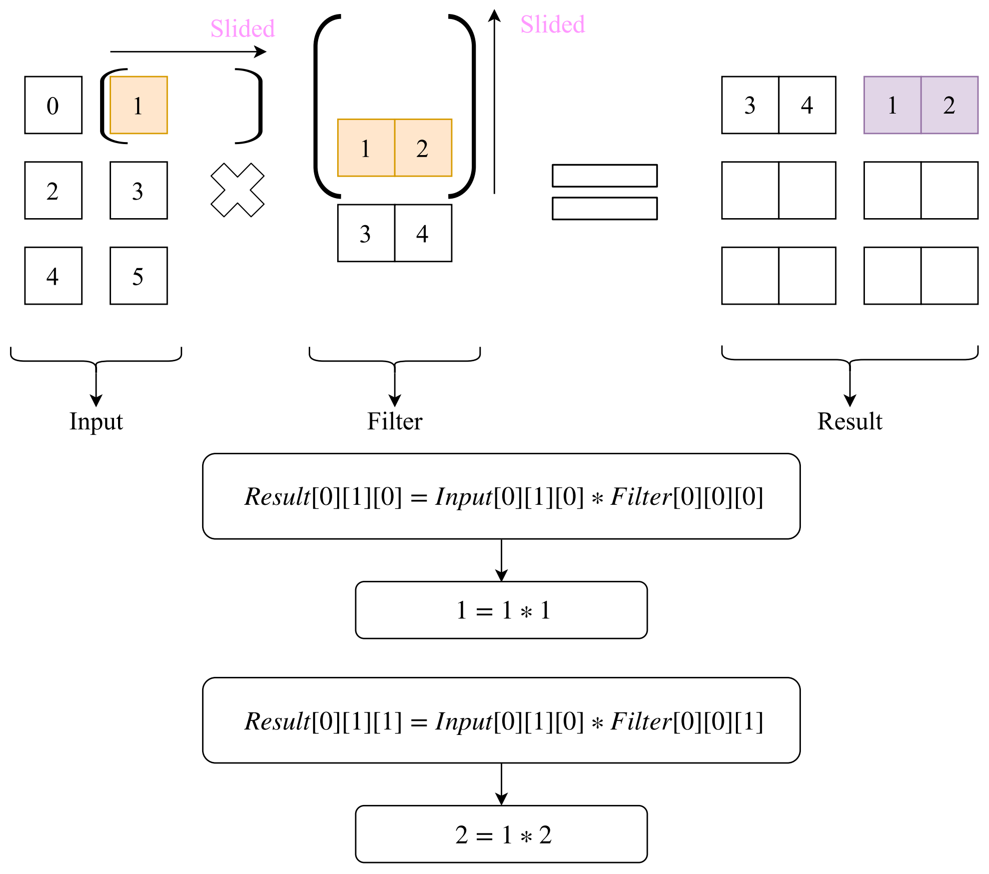
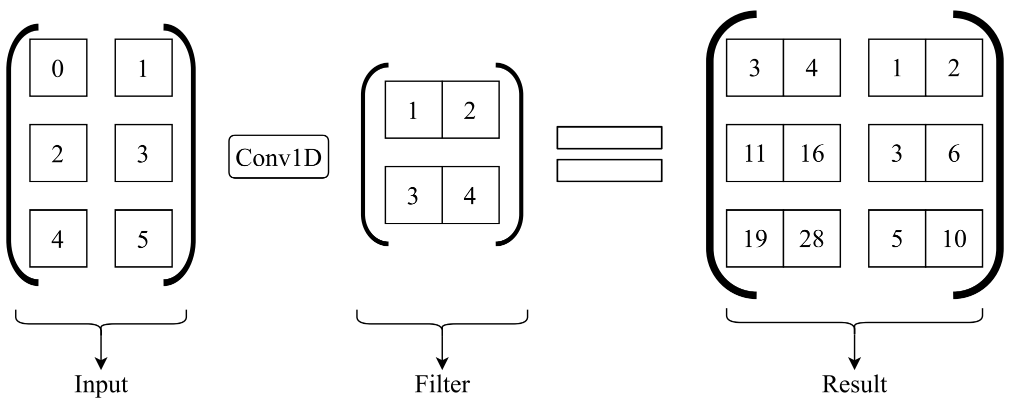

Slided
Slided
0
1
2
3
4
5
1
2
3
4
3
4
1
2
Input
Filter
Result
$Result[0][1][0] = Input[0][1][0] * Filter[0][0][0]$
$1 = 1 * 1$
$Result[0][1][1] = Input[0][1][0] * Filter[0][0][1]$
$2 = 1 * 2$

Fig. 13: The entire convolution performed on the batch data.



The resulting 3D tensor is of shape (3, 2, 2), and evident by its x-axis, it contains 2 feature maps.

To continue with the example, the resulting tensor from the previous convolution can be run through another convolution, for this purpose we need a new filter that conforms to the rules of matrix multiplication (taking into account the sliding rule), that is its y-axis has the same length as the z-axis from the tensor we're convolving on.

Also, the x-axis of the filter should be larger or equal to the y-axis of the tensor being convolved on.

And for the sake of making the example closer to real-life usage, we will make the resulting tensor from this new convolution have the same shape as the original input tensor by making the z-axis of the new filter of length 1, which will reduce the number of feature maps from 2 to 1, then by performing dimensionality reduction (squeezing of the z-axis onto the y-axis).

The shape of the newly constructed filter would be: (var, 2, 1), where $var \quad = \quad 2$, in our example var = 3.

Fig. 14: The output from the first conv layer becomes input to the second conv layer

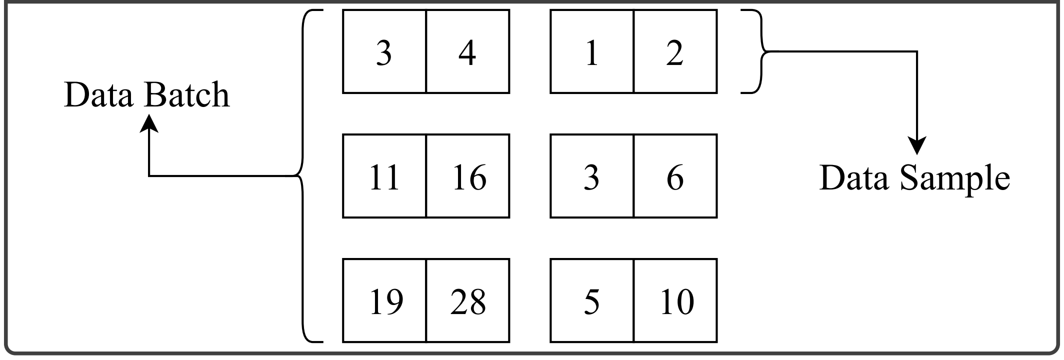


Fig. 15: Filter #2 with arbitrary weights

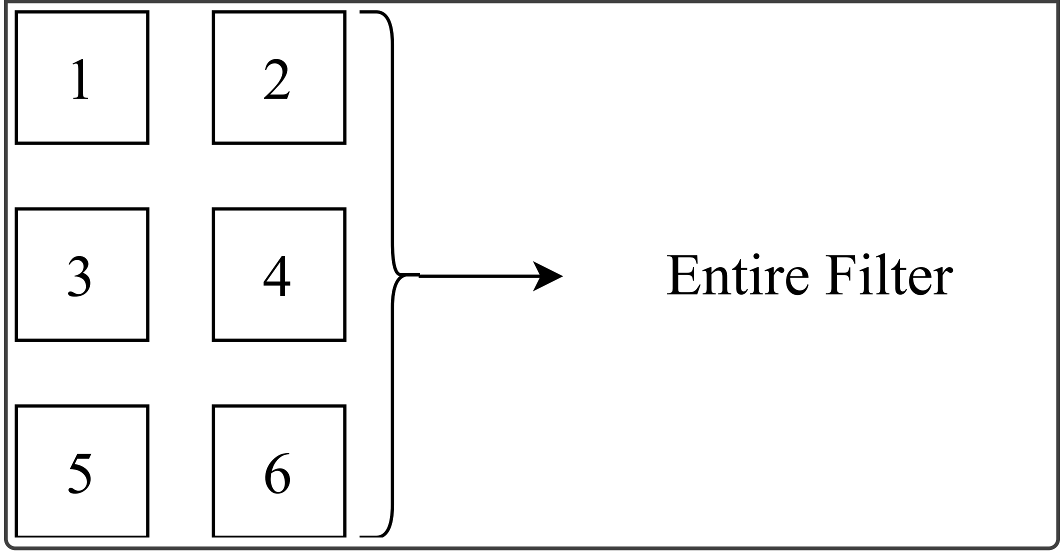
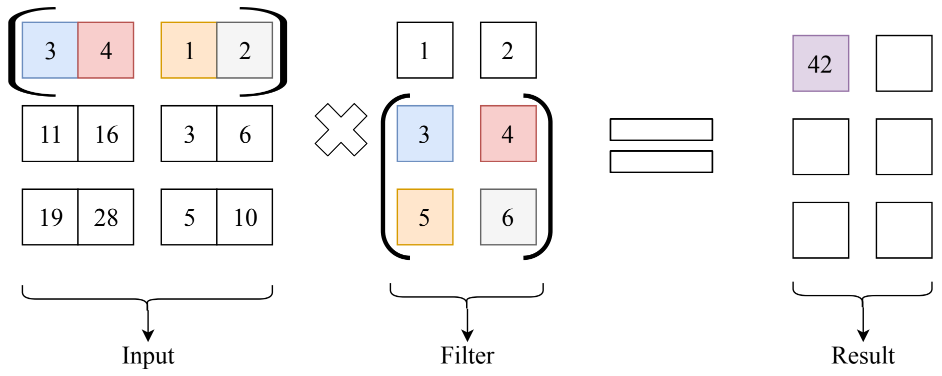


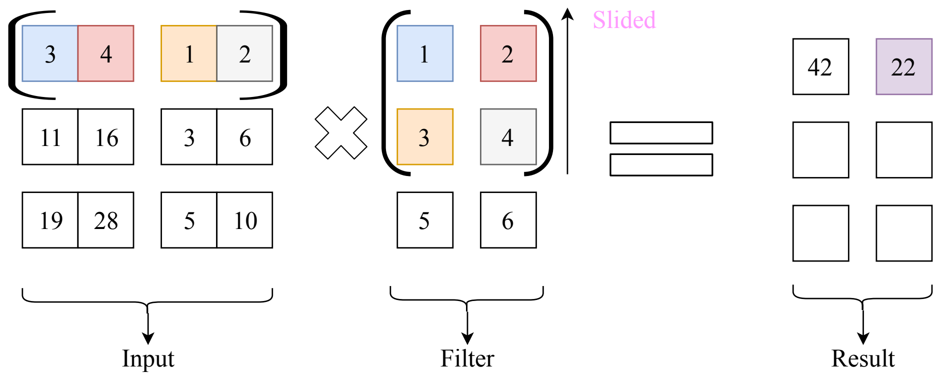
Fig. 16: Extracting the first value from the first sample in the data tensor to the feature map.



$$\begin{aligned}Result[0][0][0] = {} & Input[0][0][0] * Filter[1][0][0] \\ & + Input[0][0][1] * Filter[1][1][0] \\ & + Input[0][1][0] * Filter[2][0][0] \\ & + Input[0][1][1] * Filter[2][1][0]\end{aligned}$$

$$42 = 3 * 3 + 4 * 4 + 1 * 5 + 2 * 6$$

Fig. 17: Extracting the second value from the first sample in the data tensor to the feature map.



$$\begin{aligned}Result[0][1][0] = {} & Input[0][0][0] * Filter[0][0][0] \\ & + Input[0][0][1] * Filter[0][1][0] \\ & + Input[0][1][0] * Filter[1][0][0] \\ & + Input[0][1][1] * Filter[1][1][0]\end{aligned}$$

$$42 = 3 * 1 + 4 * 2 + 1 * 3 + 2 * 4$$

Fig. 18: The entire convolution performed on the batch data.

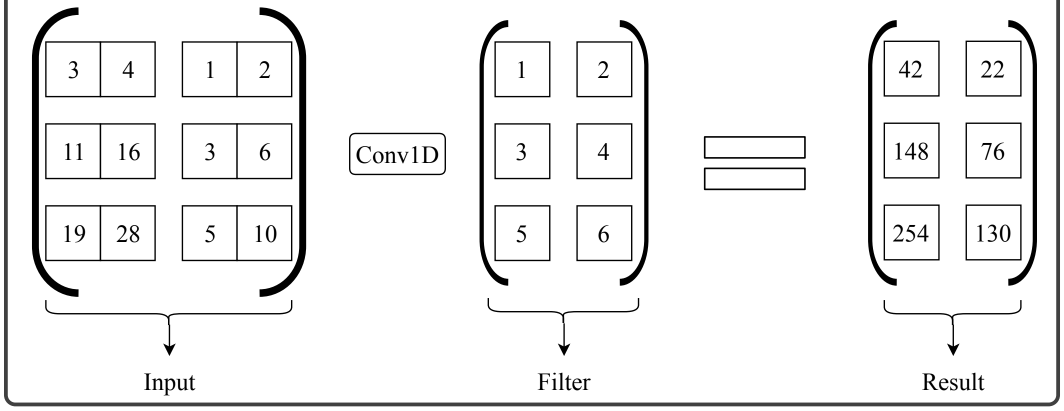


Fig. 19: squeezing of the z-axis onto the y-axis.

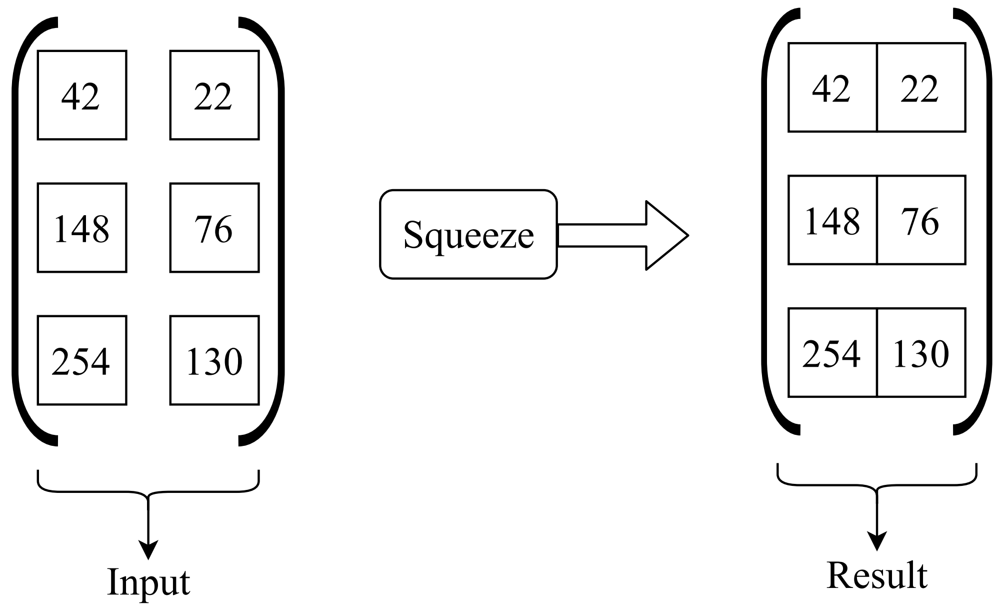


Fig. 20: Net Graph Diagram - Alice

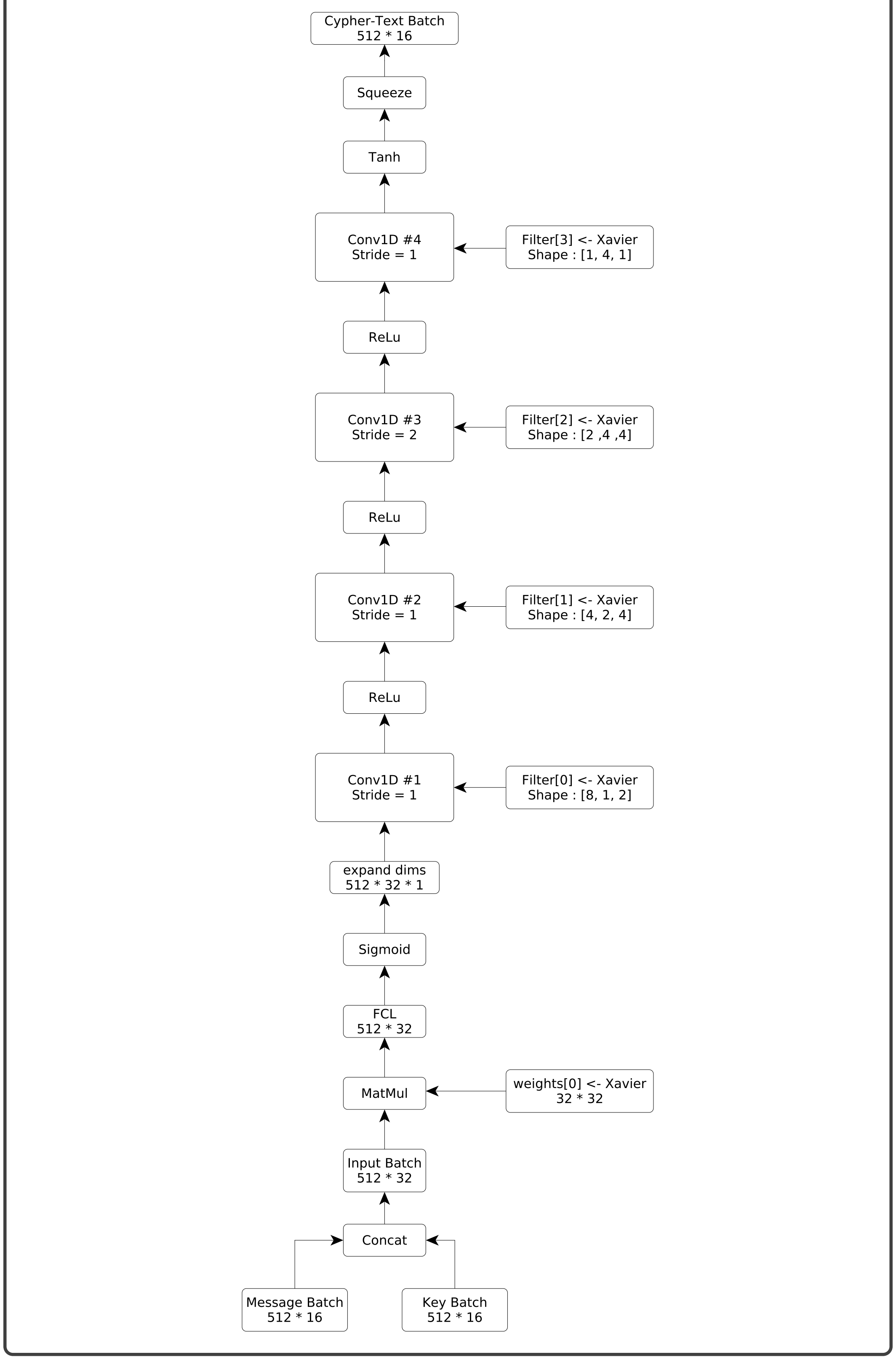


Fig. 21: Net Graph Diagram - Bob

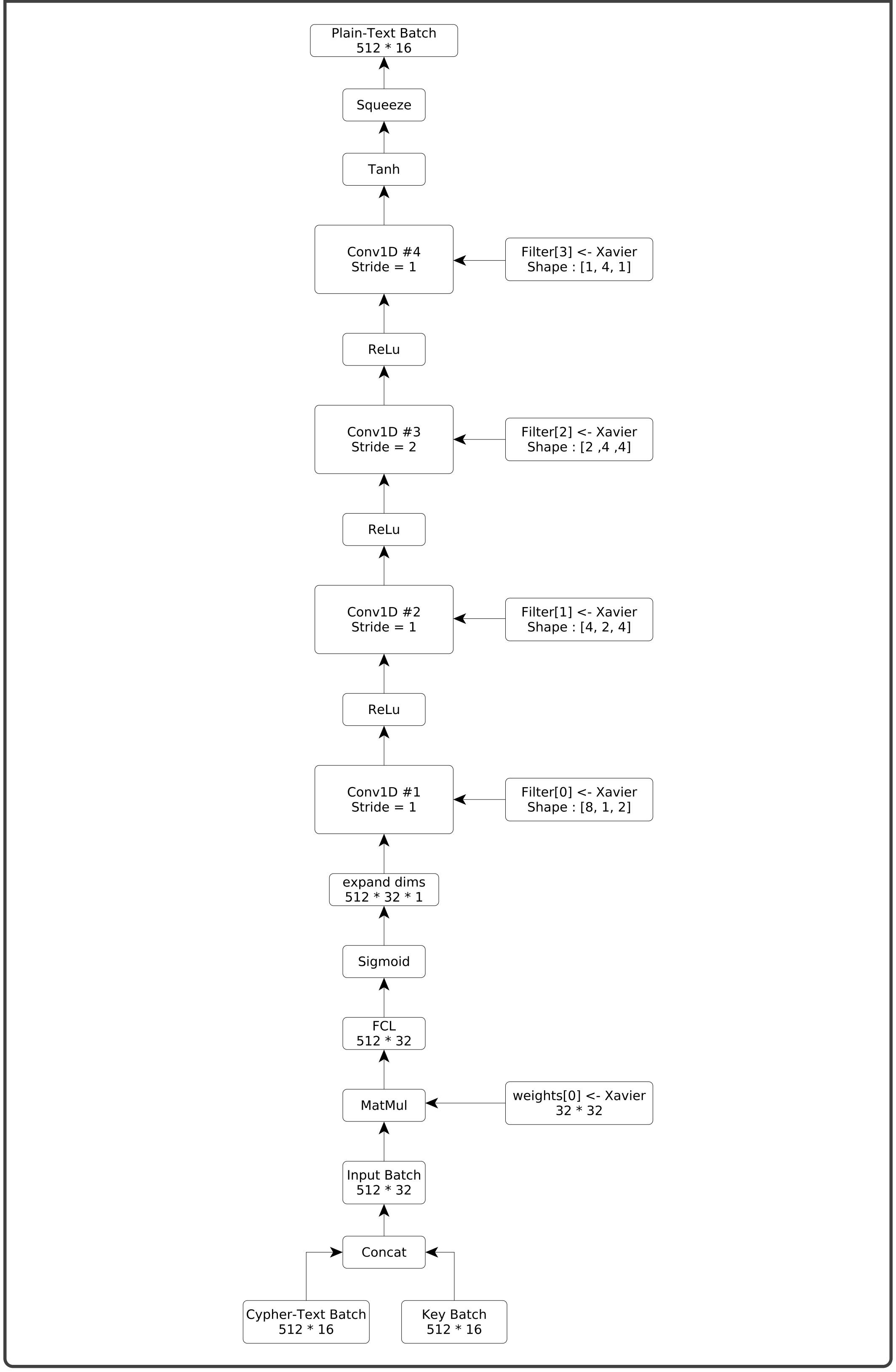
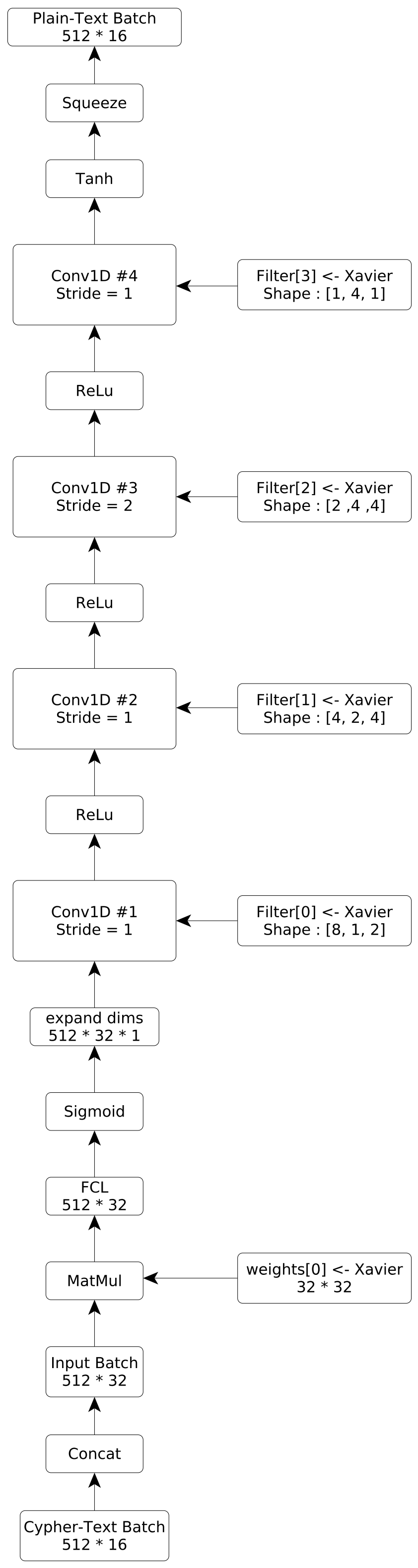


Fig. 22: Net Graph Diagram - Eve/Alan



2 Design

3 Implementation

4 Conclusion

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, pages 249–256, 2010.

Appendix