University of L’Aquila

Department of Information Engineering, Computer Science and Mathematics

Hyper Heursitic Cryptography with Mixed Adversarial Nets

Author

Name :

Aly Shmahell

Signature :

Supervisor

Name :

Prof. Giovanni De Gasperis

Signature :

June 23, 2018

Dissertation License

Project License

Author’s Contact Information

: @AlyShmahell
: @AlyShmahell
in : alyshmahell
✉ : aly.shmahell@gmail.com

Code Repository & Dissertation Publication

A Copy of this dissertation can be found on the author’s Github account.
A Copy of the code is available for review by academic researchers and industry professionals upon request.

Abstract

Chapter 1

Introduction

Definition 1 *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 Thesis Objectives

The objective of this thesis 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 Brain [**?**], which promises to bridge the gap between research and application in the area of Neural Cryptography.

The goal of my research is to provide the following:

- An addition to the variety of the underlying mechanics 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 specter in this area being dominated by authors coming from a mathematical-background angle, my thesis aims to provide:

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

This thesis finally adds the following:

- The introduction of a hybrid neural crypto-system.
- An exploration into adding hyper heuristics to the field.

1.2 Thesis Motivation

The question of motivation behind an idea can be empirically divided into:

- How would the author justify the importance of the idea?
- How would the author justify the viability of the idea?

1.2.1 Justification for the importance of Neural Cryptography

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

It makes sense from an academic perspective that we want neural nets to incorporate an understanding of cryptography, this would propel them closer to achieving general intelligence status, which is a major drive behind research in the field.

It also makes sense from an economic and existential point that we want neural nets to parallel their success in surpassing traditional methods when it comes to cryptography, because cryptography from a traditional sense is static, it always requires mathematicians and computer scientists to come together to patch it and upgrade it, and it is also always under attack, its mathematical models are always being broken and bent with the advancement in computer-power and the incorporation of new mathematical models into software that can break it.
Having neural nets as a dynamic generative model is an opportunity to gain an upper hand on bad actors and put cryptography in a more reactive state to protect our sensitive infrastructure, it would still require research and development, but it would put the neural net as a front line of always devising new ways to mitigate risk and reformulate a cryptographic solution on the fly.

1.2.2 Justification for the viability of Neural Cryptography

This boils down to multiple general factors:

- **Neural Nets are viable general function approximaters:** The incorporation of multiple heuristic methodologies into neural nets has made them tackle a rapidly growing heap of complex tasks, cryptography is just another human invention to be caught up with.
- **Neural Nets are becoming faster:** The increasing successful research into using these heuristics not just to compute a complex task, but to do it quickly without loss in accuracy.
- **Neural Nets are becoming available:** 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 thesis-related factors:

- **Neural Cryptography is viable:** 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.
- **Neural Cryptanalysis is viable:** Having a Mixed Convolutional Net with fully connected layers will teach the network to account for global spatial relations as well, which teaches the net to learn and counter cryptanalysis.
- **Neural Cryptography can be fast:** 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, and this reduces the compute-power required for the whole process compared to other network models.
- **Neural Cryptography is evolved opposite to being patched:** 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 Previous Work

The work being done so far in Neural Cryptography can be divided to old (pre 2016), and new (post 2016). For the old section, this thesis will only list the works and attributions without delving into the details.

1.3.1 Previous Work - Pre 2016 Era

Up until 2010, Neural Nets were a dark alley in the citadel of Artificial Intelligence, mainly due to lack in advancement in back-propagation optimization which made training deep neural nets hard, and the fact that not many people saw the importance of incorporating other areas of Artificial Intelligence into neural nets.
From 2010 until 2016, things started changing for the better, but it was 6 years until the developments allowed for new viable research in Neural Cryptography.
Therefor the works in the Pre 2016 Era were merely academic curiosities which did not aim to make it into industry, but they provided a key stepping stone for those of us who came into the field at this better-equipped stage.

The most notable of these works are:

- The first definition of the Neuro-Cryptography (AI Neural-Cryptography) applied to DES cryptanalysis. [**?**]
- Permutation parity machines for neural synchronization [**?**]
- Permutation parity machines for neural cryptography [**?**]
- Successful attack on permutation-parity-machine-based neural cryptography [**?**]

1.3.2 Previous Work - Post 2016 Era

In the period 2010 - 2016, a surge of new ideas came into play in the field of neural nets, new methods of back-propagation optimization proved to be successful for deep-learning, advancement in initialization and activation solved multitudes of problems like the vanishing gradient (or at least mitigated its effect), ...etc, so it was only natural someone would attempt to take another look at Neural Cryptography, and the most notable works are:

- **Learning to Protect Communications with Adversarial Neural Cryptography** [**?**]:
  This paper was the corner stone for my work, it was the first to realize the importance of applying Generative Adversarial Nets and succeed in its efforts to build a viable cryptosystem.
- **Tensorflow implementation of Adversarial Neural Cryptography** [**?**]:
  Ankesh Anand's Implementation of (Learning to Protect Communications with Adversarial Neural Cryptography) using tensorflow and python is a very informative open-source prototype which I studied before I set on implementing my project.
- **Adversarial Neural Cryptography in Theano** [**?**]:
  Liam Schoneveld made an implementation of (Learning to Protect Communications with Adversarial Neural Cryptography) in Theano and python, his results mirror mine to some extent, and his illustrations and break-down of the process is something to consider going over when delving into Neural Cryptography.

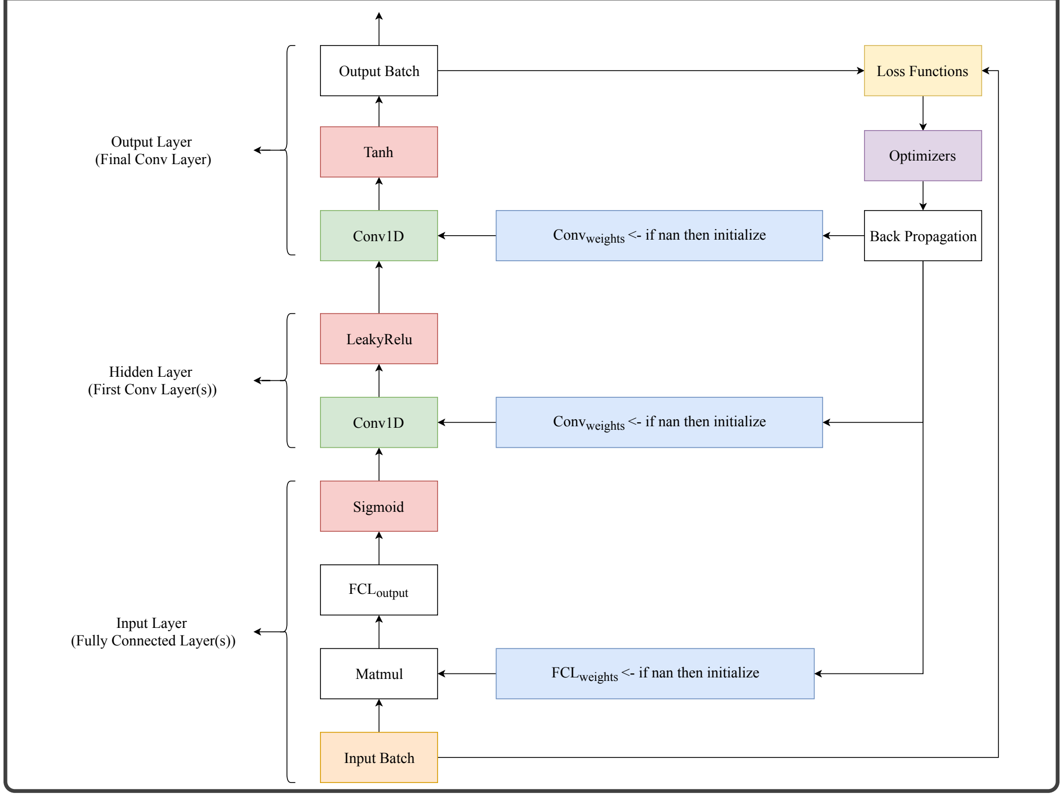
Chapter 2

Design

This chapter deals with the inner-mechanics of how convolutional neural nets work, which are the building blocks for the adversarial crypto-system presented.

As a way to illustrate how a ConvNet should be constructed for our purposes, a simplified dummy example is presented and dissected to explain how its components work.

Fig. 2.1: Sample Dummy Net Graph Diagram



This ConvNet has 6 key components which we will go over as the following:

- Weight Initialization
- 1D Convolution
- Batches
- Activation Functions
- Loss Functions
- Optimization

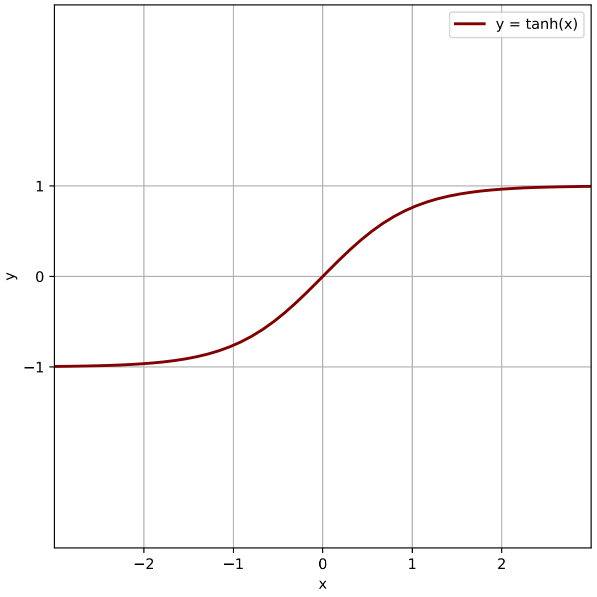
2.1 Weight Initialization

Definition 2 *Initialization is the process in which we give some weight values to some neurons in some layer, the overall process, whether done properly or not means the difference between the network converging on a local/global minima, or never converginf at all.*

2.1.1 Xavier Initialization [?]

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

Fig. 2.2: example plot of the hyperbolic tangent



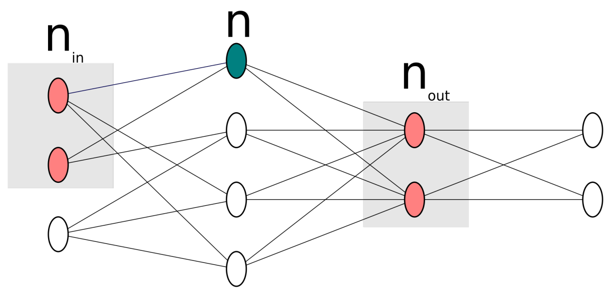
- *If the weights start too small, then the signal shrinks as it passes through each layer until it vanishes [**?**], then as it passes deeper in the network with its small values, the layers it enter will become linear, because the output of the hyperbolic tangent is linear with small input values, 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. 2.3: Xavier Initialization on neuron "n"



$$Y = W_1X_1 + W_2X_2 + ... + W_nX_n \tag{2.1}$$

To calculate the variance of each component:

$$Var(W_iX_i) = E[X_i]^2Var(W_i) + E[W_i]^2Var(X_i) + Var(W_i)Var(X_i) \tag{2.2}$$

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

$$E[X_i]^2Var(W_i) + E[W_i]^2Var(X_i) = 0 \implies Var(W_iX_i) = Var(W_i)Var(X_i) \tag{2.3}$$

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:

$$Var(Y) = Var(W_1X_1 + W_2X_2 + ... + W_nX_n) = nVar(W_i)Var(X_i) \tag{2.4}$$

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:

$$Var(Y) = Var(X_i) \implies Var(W_i) = \frac{1}{n_{in}} \implies Var(W_i) * n_{in} = 1 \tag{2.5}$$

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

$$Var(W_i) = \frac{1}{n_{out}} \implies Var(W_i) * n_{out} = 1 \tag{2.6}$$

To keep the variance of the input gradient & the output gradient the same, we combine (2.5) & (2.6) and we get:

$$(n_{out} + n_{in}) * Var(W_i) = 2 \implies Var(W_i) = \frac{2}{n_{in} + n_{out}} \tag{2.7}$$

2.2 1D Convolution

When looking for examples and literature on Convolution, the vast majority of what is available is on 2D Convolution, and what might be found on 1D convolution is usually done on 1D data with 1D filter [**?**].
Therefor, for the purposes of this study, a more complex example will be presented.

2.3 1D Convolution on Batch 1D Data with (1,2)-D Filters

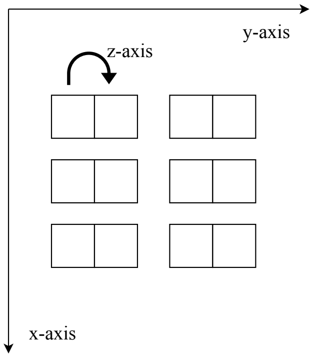
1D Convolution is the process of using a small window to determine local spatial relations over a 1D data sample.
At this moment, the best tool available for representing data samples is **tensors**, and therefor determining local spatial relations amounts to matrix multiplications of the spatial locations inside the data sample tensor by the portion of the filter tensor that fits the location.
In order for the convolution to do these multiplications, it **slides** over one axis of the 1D data sample, the y-axis, the filter also **slides** itself to fit the location.
In order to perform 1D convolution with a 2D filter (having an x-axis and a y-axis), we need to add another axis to both the data and the filter; done by injecting a z-axis into the y-axis, effectively expanding the y-axis over the z-axis. In this case, when the filter *slides* itself, it *slides* over its x-axis.
In this format, the window processing is done by multiplying the z-axis of the sample by the y-axis of the filter, over the x-axis of the filter.
Every *slide* is done over a fixed length called a stride, and every *slide* represents moving the filter window over to a new spatial location in the sample.
If instead of 1 data sample, we have a batch of data samples, we perform convolution on the samples independently. This means the data batch has an extra axis, an x-axis.
This way, each time a convolution over a sample is done, the filter resets its **slided** position to its default, and the convolution **steps** onto the next sample over the x-axis in the batch.
Finally, we end up with a 3D tensor for the batch data, and another 3D tensor for the filter.
After the filter is done processing all the samples in the batch, the result tensor will be a 3D tensor with the following dimension lengths:

- $result.xAxis.length = batch.xAxis.length$
- $result.yAxis.length = filter.xAxis.length$
- $result.zAxis.length = filter.zAxis.length$

Each entry over the x-axis of the result represent a processed sample, and if the z-axis of the result has $length > 1$, then each entry over the y-axis represents a feature map of the sample.

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 Computer Scientists.

Fig. 2.4: Flat 2D Representation of 3D Tensors - Numpy Style



2.3.1 1D Convolution Algorithm

Lemma 4 *The relation between the dimension lengths of the batch and the filter can be described as the following:*

- $filter.xAxis.length >= 1$
- $filter.yAxis.length = batch.zAxis.length$
- $filter.zAxis.length >= 1$

Lemma 5 *if $filter.xAxis.length > batch.yAxis.length$, the filter is offset by an amount of $filter\_offset = filter\_x\_axis\_length - batch\_y\_axis\_length$ throughout the convolution process.*

Algorithm 1: 1D Convolution Pseudo-Code

```
$step = 0$
**if** $filter.xAxis.length > batch.yAxis.length$ **then**
    $filter.offset = filter.xAxis.length - batch.yAxis.length$
**else**
    $filter.offset = 0$
**end if**
**while** $step < batch.xAxis.length$ **do**
    $slide = 0$
    **while** $slide < batch.yAxis.length$ **do**
        $y = slide$
        **while** $y < batch.yAxis.length$ **do**
            $z = 0$
            **while** $z < filter.zAxis.length$ **do**
                $result[step][y][z] = \sum_{x=0}^{x<=filter.xAaxis.length} \Big\{ batch[step][x+slide][y] * filter[x - slide + filter.offset][y][z] \Big\}$
                $z = z + 1$
            **end while**
            $y = y + 1$
        **end while**
        $slide = slide + 1$
    **end while**
    $step = step + 1$
**end while**
```

2.3.2 Simple ConvNet Example with 2 Conv Layers

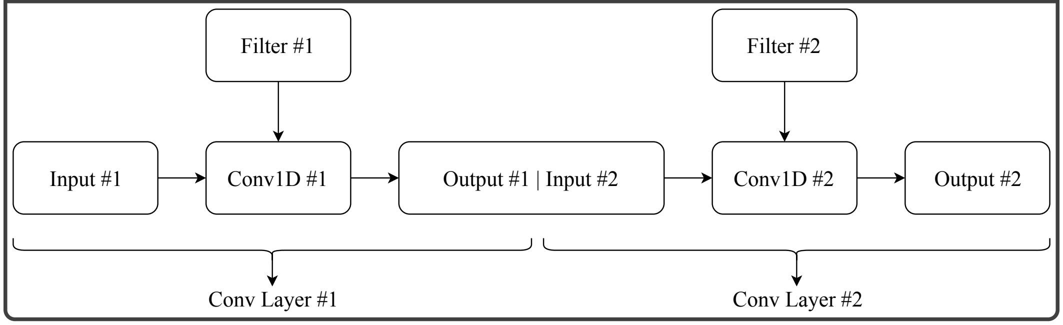
This example illustrates how stacked convolutions work, by feeding one Conv Layer output to the next one.

It also illustrates how 1D convolution works on batch 1D data with 2D (expanded to 3D) filters in each Conv Layer.

For the sake of simplicity:

- We will forsake the use of activation functions between layers.
- We will also use weights initialized by hand, chosen arbitrarily.

Fig. 2.5: Simple ConvNet with 2 Conv Layers



Conv Layer #1

The example starts off by generating 1D input strings.

Fig. 2.6: First Input String

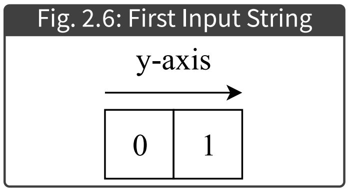


Fig. 2.7: Second Input String

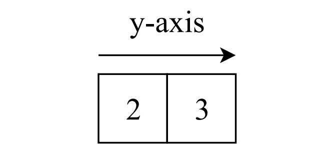
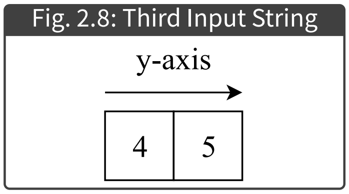
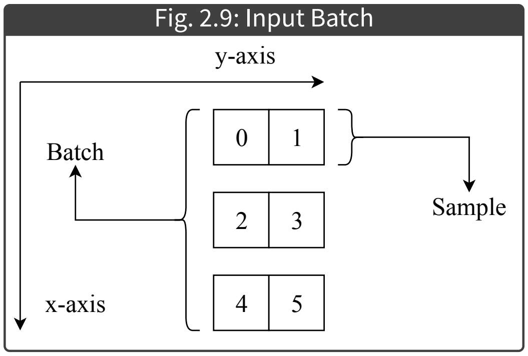


Fig. 2.8: Third Input String



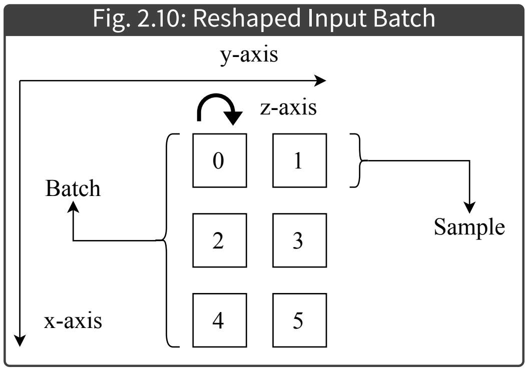
Input strings are then stacked up along the x-axis to form a 2D batch.

Fig. 2.9: Input Batch



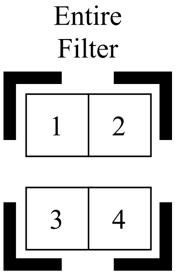
Then the batch gets expanded from 2D to 3D along the y-axis (injecting z-axis into y-axis), which means the final batch shape becomes (3, 2, 1).

Fig. 2.10: Reshaped Input Batch



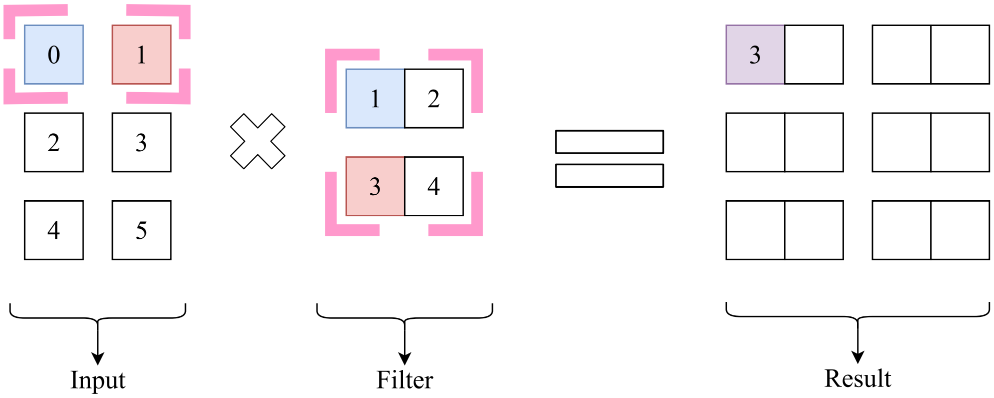
Then finally, a filter of shape (2, 1, 2) is provided for the convolution.

Fig. 2.11: Filter #1 with arbitrary weights



After the input batch tensor and the filter tensor have been generated, the convolution process can begin.

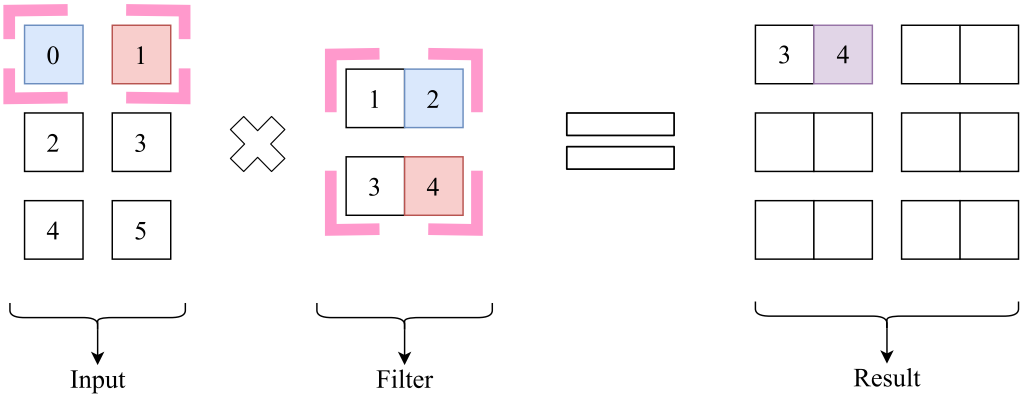
Fig. 2.12: Applying the filter to the first sample in the input batch - generating the first value for the first in the first feature map.



$$Result[0][0][0] = Input[0][0][0] * Filter[0][0][0] + Input[0][1][0] * Filter[1][0][0]$$

$$3 = 0 * 1 + 1 * 3$$

Fig. 2.13: Applying the filter to the first sample in the input batch - generating the second value in the first feature map.



$$Result[0][0][1] = Input[0][0][0] * Filter[0][0][1] + Input[0][1][0] * Filter[1][0][1]$$

$$4 = 0 * 2 + 1 * 4$$

Fig. 2.14: Applying the filter to the first sample in the input batch - generating the second feature map.

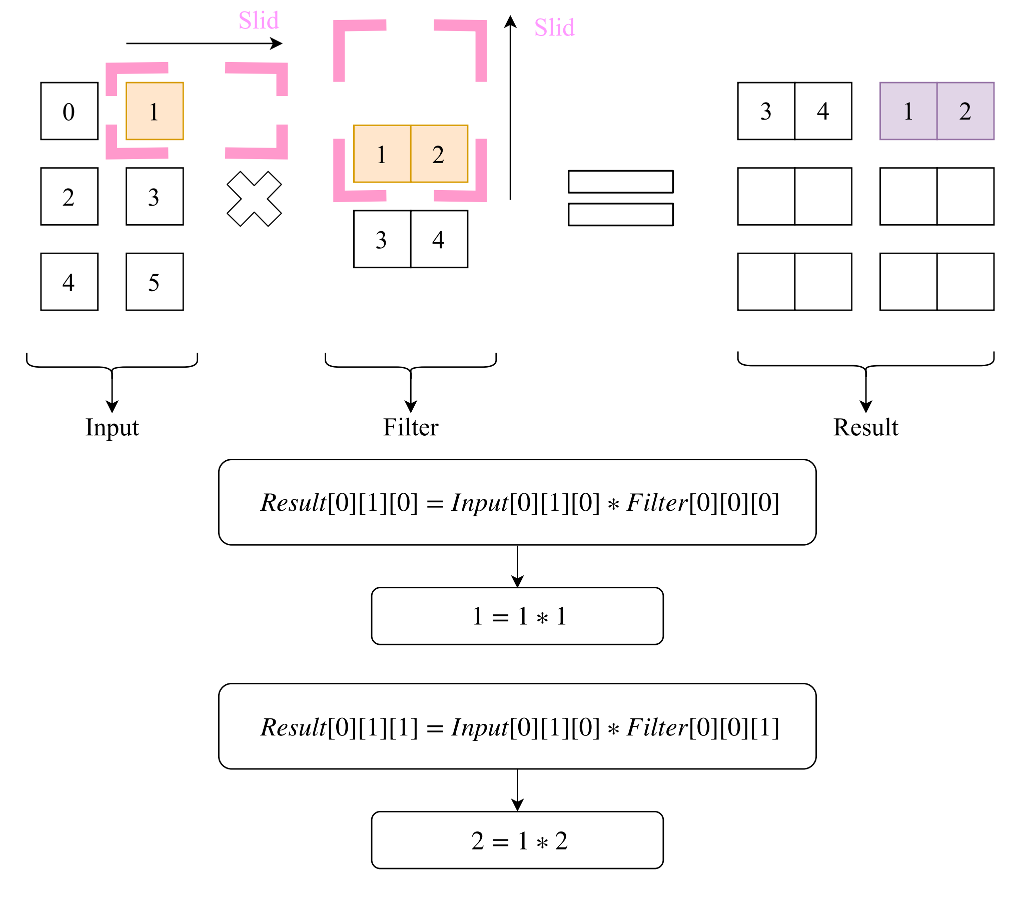
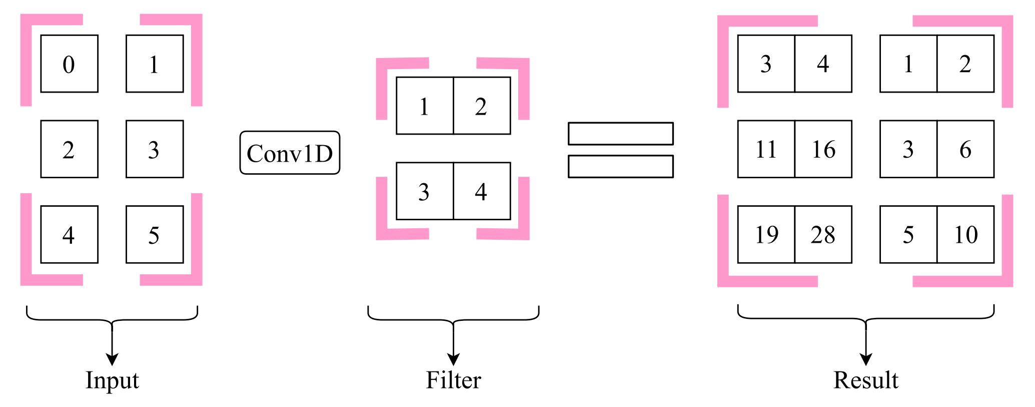

Slid
Slid
0
1
2
3
4
5
1
2
3
4
3
4
1
2
Input
Filter
Result
$Result[0][1][0] = Input[0][1][0] * Filter[0][0][0]$
$1 = 1 * 1$
$Result[0][1][1] = Input[0][1][0] * Filter[0][0][1]$
$2 = 1 * 2$

Fig. 2.15: The entire convolution performed on the batch data.



The resulting 3D tensor is of shape (3, 2, 2), and it contains 2 feature maps.

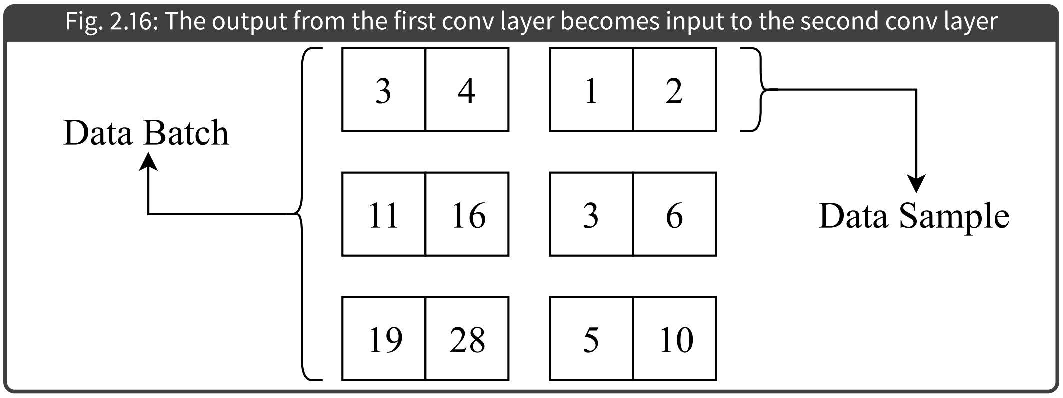
Conv Layer #2

To continue with the example, the resulting tensor from **Conv Layer #1** will be fed as input to **Conv Layer #2**, 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.

And for the sake of making the example closer to real-life usage, we will make the resulting tensor from *Conv Layer #2* 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).

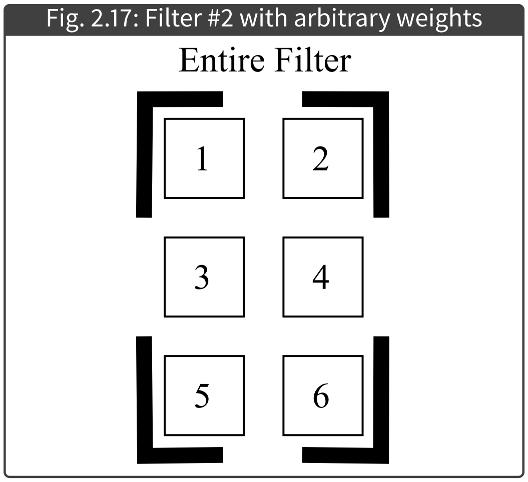
The shape of the newly constructed filter would be: (var, 2, 1), where $var >= 1$, in our example var = 3.

Fig. 2.16: The output from the first conv layer becomes input to the second conv layer



A filter of shape (3, 2, 1) is provided for the convolution.

Fig. 2.17: Filter #2 with arbitrary weights



After the input batch tensor has been provided the filter tensor has been generated, the convolution process can begin.

Fig. 2.18: Extracting the first value from the first sample in the data tensor to the feature map.

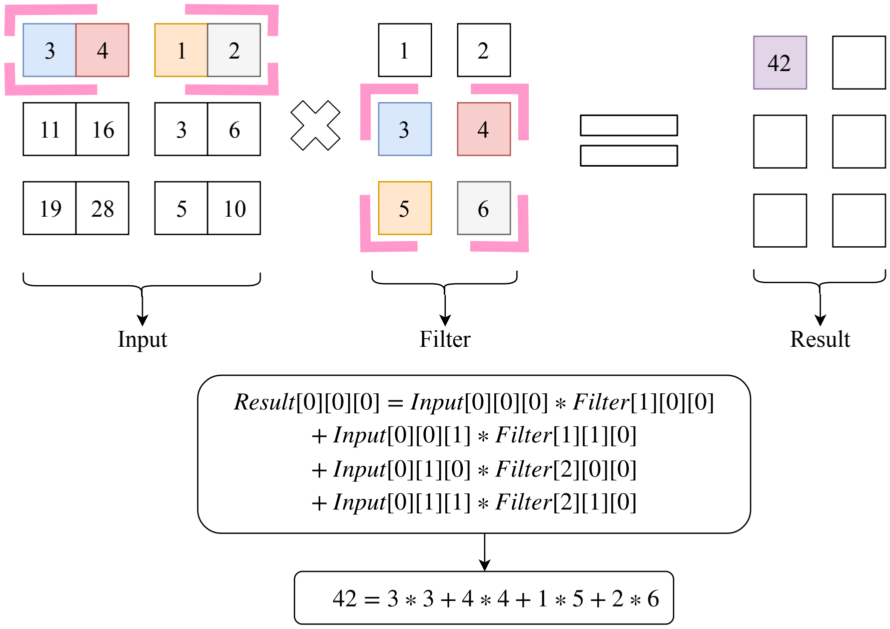

3 4 1 2
11 16 3 6
19 28 5 10
1 2
3 4
5 6
42
Input
Filter
Result
$Result[0][0][0] = Input[0][0][0] * Filter[1][0][0]$
$+ Input[0][0][1] * Filter[1][1][0]$
$+ Input[0][1][0] * Filter[2][0][0]$
$+ Input[0][1][1] * Filter[2][1][0]$
$42 = 3 * 3 + 4 * 4 + 1 * 5 + 2 * 6$

Fig. 2.19: Extracting the second value from the first sample in the data tensor to the feature map.

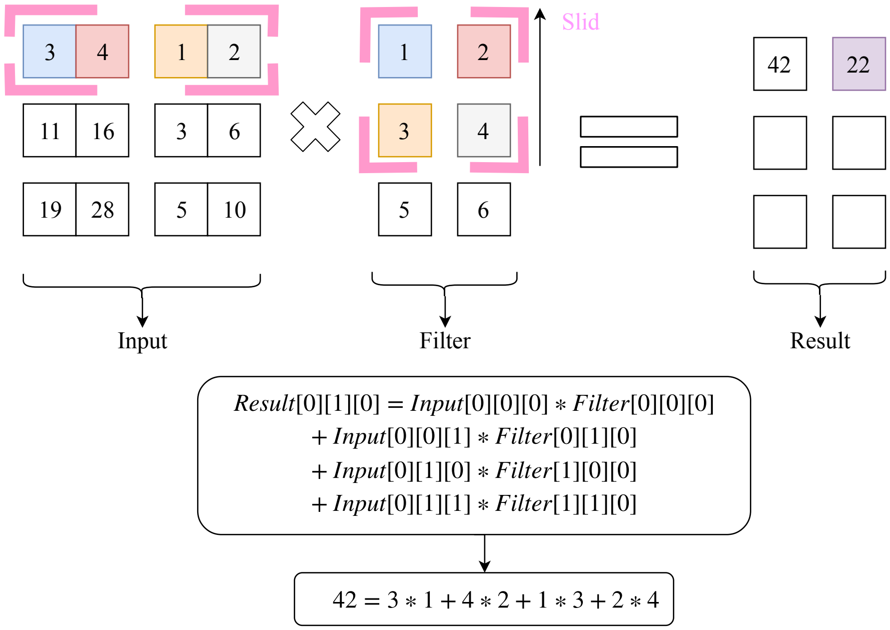

3 4 1 2
11 16 3 6
19 28 5 10
1 2
3 4
5 6
Slid
42 22
Input
Filter
Result
$Result[0][1][0] = Input[0][0][0] * Filter[0][0][0]$
$+ Input[0][0][1] * Filter[0][1][0]$
$+ Input[0][1][0] * Filter[1][0][0]$
$+ Input[0][1][1] * Filter[1][1][0]$
$42 = 3 * 1 + 4 * 2 + 1 * 3 + 2 * 4$

Fig. 2.20: The entire convolution performed on the batch data.

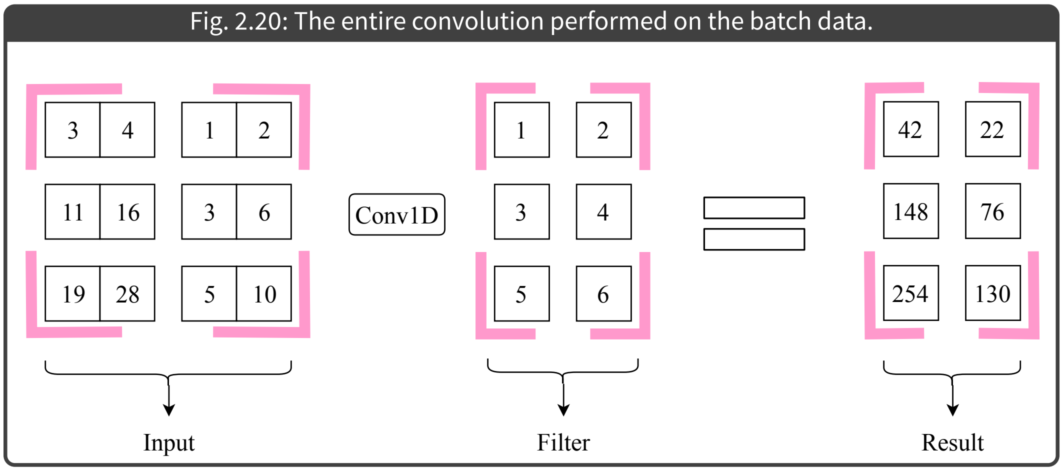
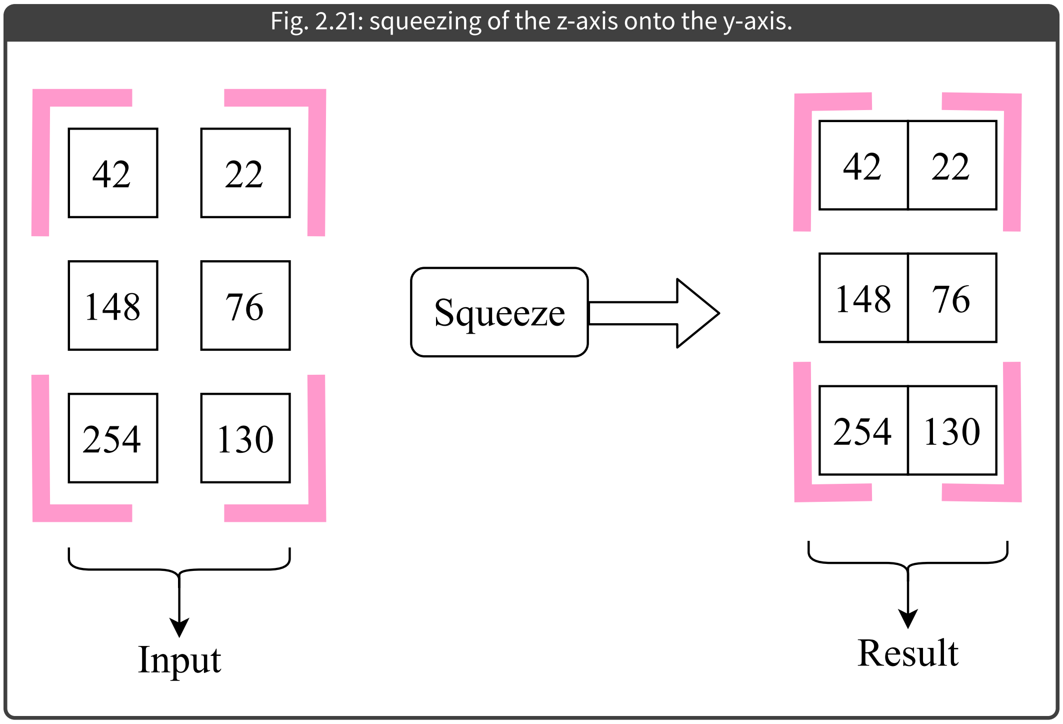


Fig. 2.21: squeezing of the z-axis onto the y-axis.



The output 3D tensor is of shape (3, 1, 1), and it contains 1 feature maps.
The output tensor is similar in shape to the input tensor, as inferred in the example.

2.4 Batches

As seen from section 2.2, when performing convolutions: we perform each convolution on each data sample separately.
Then the question arises: Why is a batch of samples is needed? The short answer is simply because a batch allows for a good experiment to take place from a statistical point of view.

When calculating the training error, each net performs mean reduction on each sample error, and a large sample is required for mean reduction (estimation), because the standard error of the mean (SEM) can be expressed as:

$$SEM = \frac{\sigma}{\sqrt{n}}$$

Where:

σ : the standard deviation of the batch
n : the size of the batch

The larger the batch size is, the less the SEM value is, the more confidence is placed in the accuracy of the mean reduction (estimation) of training errors.

2.5 Activation Functions

To understand which combination of activation functions to choose, a basic understanding of the information being processed and fed to each activation function is needed.
If we take a look at Fig. 2, we deduce the following:

- The combination of activation functions chosen is: $Sigmoid \rightarrow LeakyRealu \rightarrow Tanh$.
- Each function is fed the result of matrix multiplication operations (involving float weight values), therefor each function receives float values as input.
- Sigmoid and LeakyRelu results are then fed to next layers, therefor their results are multiplied by a distribution of float values.
- Tanh processes the results of the output layer, therefor its output remains unchanged.

There are two other alternatives to the choice of function combination which will be examined:

- $Sigmoid \rightarrow LeakyRealu \rightarrow Sigmoid$.
- $Tanh \rightarrow LeakyRealu \rightarrow Tanh$.

To test the efficacy of these combinations we will perform the following numerical analyses, considering a float distribution over an input x-axis, the distribution of possible result values over the y-axis can be described as the following:

- For $Sigmoid \rightarrow LeakyRealu \rightarrow Sigmoid$:
  $\forall x \in X, X = [-1, +1] : Y = sigmoid(leakyRelu(sigmoid(X) * x) * x)$
- For $Tanh \rightarrow LeakyRealu \rightarrow Tanh$:
  $\forall x \in X, X = [-1, +1] : Y = tanh(leakyRelu(tanh(X) * x) * x)$
- For $Sigmoid \rightarrow LeakyRealu \rightarrow Tanh$:
  $\forall x \in X, X = [-1, +1] : Y = tanh(leakyRelu(sigmoid(X) * x) * x)$

Where X is a line-space over the x-axis and Y is a line-space over the y-axis.

Fig. 2.22: $\forall x \in X, X = [-1, +1] : Y = tanh(leakyRelu(tanh(X) * x) * x)$

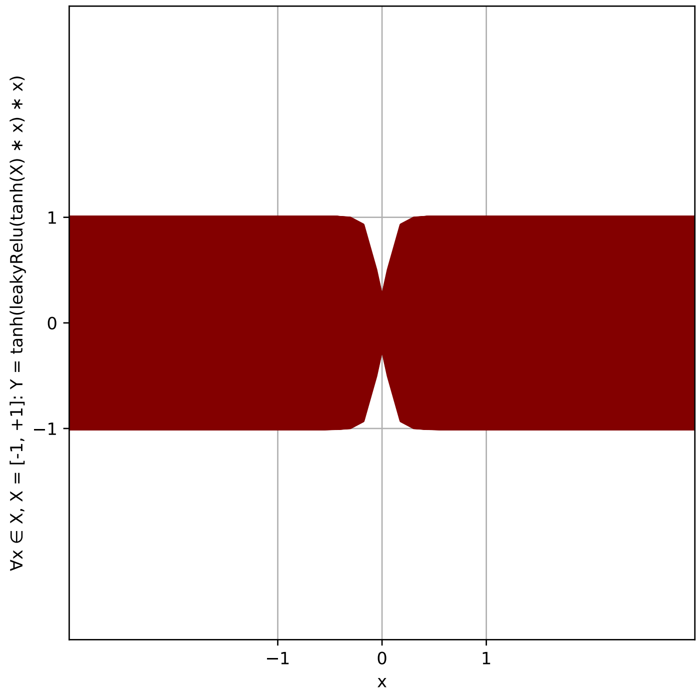


Fig. 2.23: $\forall x \in X, X = [-1, +1] : Y = sigmoid(leakyRelu(sigmoid(X) * x) * x)$

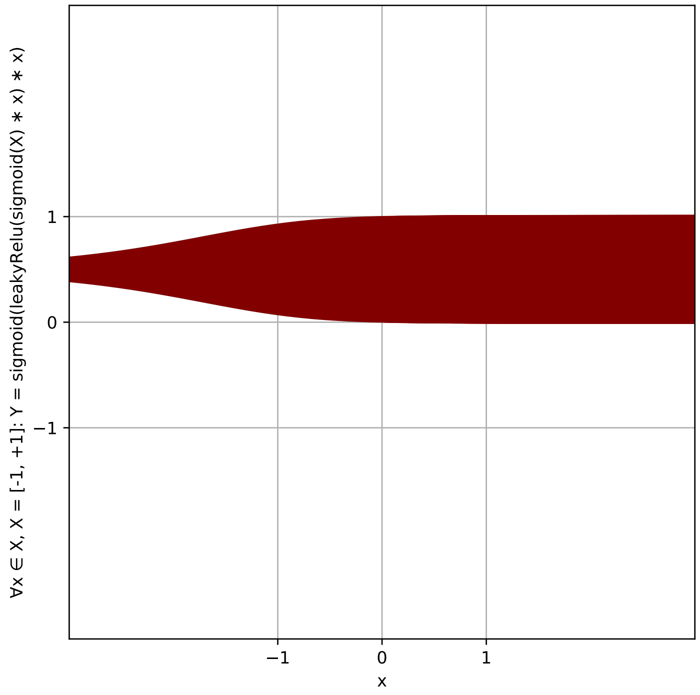
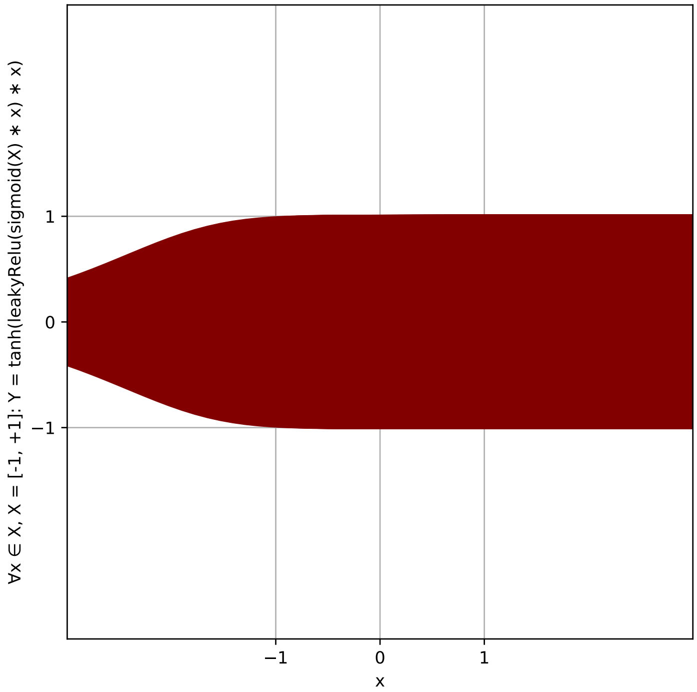


Fig. 2.24: $\forall x \in X, X = [-1, +1] : Y = tanh(leakyRelu(sigmoid(X) * x) * x)$



To illustrate how $Sigmoid \rightarrow LeakyRealu \rightarrow Tanh$ makes a better option than the presented alternatives, the net can be considered as an information channel, and Shannon's theory [**?**] applies to it, specially his entropy (on discrete values):

$$H(V) = -\Sigma P(v_i) * log_b(P(v_i))$$

Since linespaces can be converted to discrete values, If applied to the linespaces above, we get:

$$\forall x \in X, X = [-1, +1] : H(Y) = -\Sigma P(y_i) * log_b(P(y_i))$$

When comparing $Sigmoid \rightarrow LeakyRealu \rightarrow Tanh$ to $Sigmoid \rightarrow LeakyRealu \rightarrow Sigmoid$, $Sigmoid \rightarrow LeakyRealu \rightarrow Tanh$ is a winner, because both are equiprobable as we move along the x-axis, but $Sigmoid \rightarrow LeakyRealu \rightarrow Tanh$ has more information capacity and therefor has a larger $H(Y)$.
When comparing $Sigmoid \rightarrow LeakyRealu \rightarrow Tanh$ to $Tanh \rightarrow LeakyRealu \rightarrow Tanh$, $Tanh \rightarrow LeakyRealu \rightarrow Tanh$ begins to become non-linearly inequiprobable as we move towards 0 along the x-axis, this not only results in less information capacity and a lower $H(Y)$, but because each neuron in a layer is independent of the other, therefor each value probability along the x-axis is independent of the other, and the net could be training on a mix of values randomly chosen along the x-axis, this will lead to non-uniformality of information capacity and entropy across the trainable process (e.g. Conv1D), which will lead to inconsistent results for that process, and the conversion will jump up and down, and probably not get fully achieved. Therefor: $Sigmoid \rightarrow LeakyRealu \rightarrow Tanh$ is the winner combination.

2.6 Loss Functions

Definition 3 *Loss functions are functions that map out the error of the network results compared to the desired output.*

The desired output for our nets are the following:

- For an Encryptor-Decryptor combination, loss has positive correlation with its own loss, and negative correlation with the Eavesdropper loss:
  $Loss_{positive} = mean(Decryptor_{output} - Encryptor_{input})$
  $Loss_{negative} = (1 - mean(Eavesdropper_{output} - Encryptor_{input}))^2$
  $Loss = Loss_{positive} + Loss_{negative}$
- For an Encryptor-Eavesdropper combination, loss has positive correlation with its own loss, but is intended not to be aware of the loss of the Decryptor:
  $Loss = mean(Eavesdropper_{output} - Encryptor_{input})$

2.7 Optimizers

Definition 4 *An Optimizer is an algorithm or a methodology used to optimize the weights of the network using backpropagation, more often by utilizing stochastic techniques, to reduce the error produced by the loss function(s).*

The state of the art at this moment is the **Adam Optimizer** [**?**], which is out of the scope of this dissertation, mainly because it is a full topic in its own right, and it is well documented elsewhere, especially where it was proposed in its original paper.

Chapter 3

Implementation

3.1 Net Structures

Fig. 3.1: Net Graph Diagram - Alice

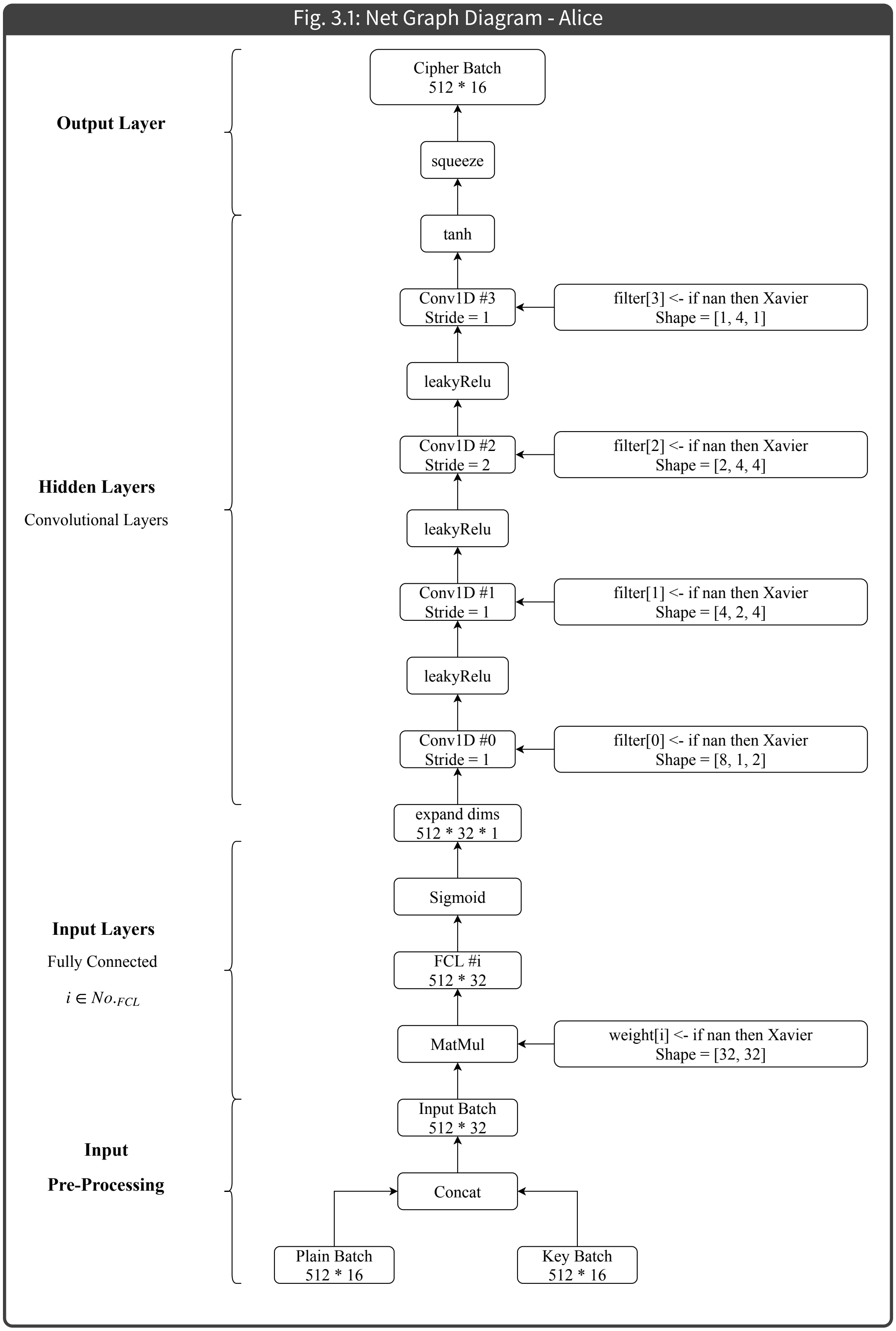


Fig. 3.2: Net Graph Diagram - Bob

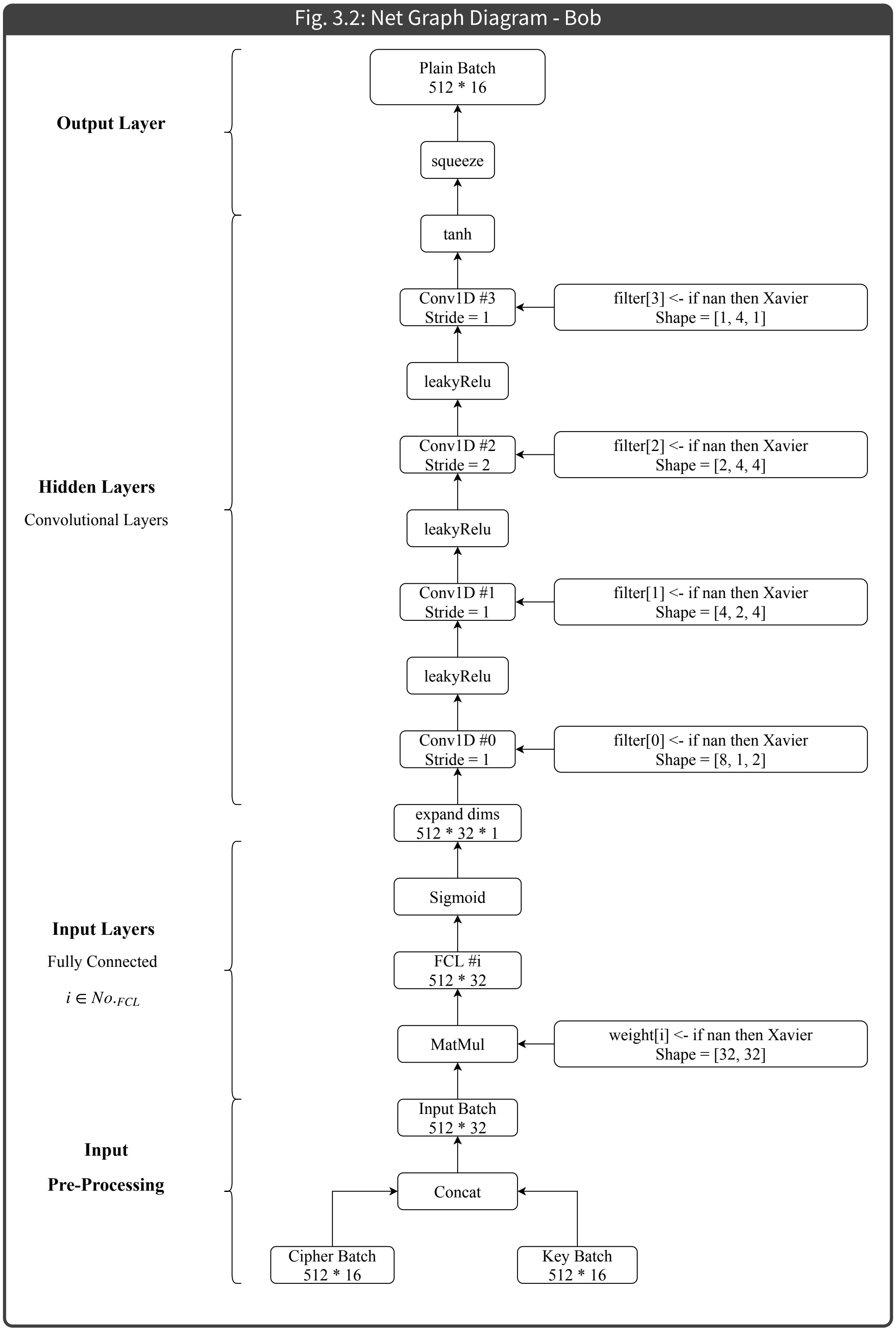
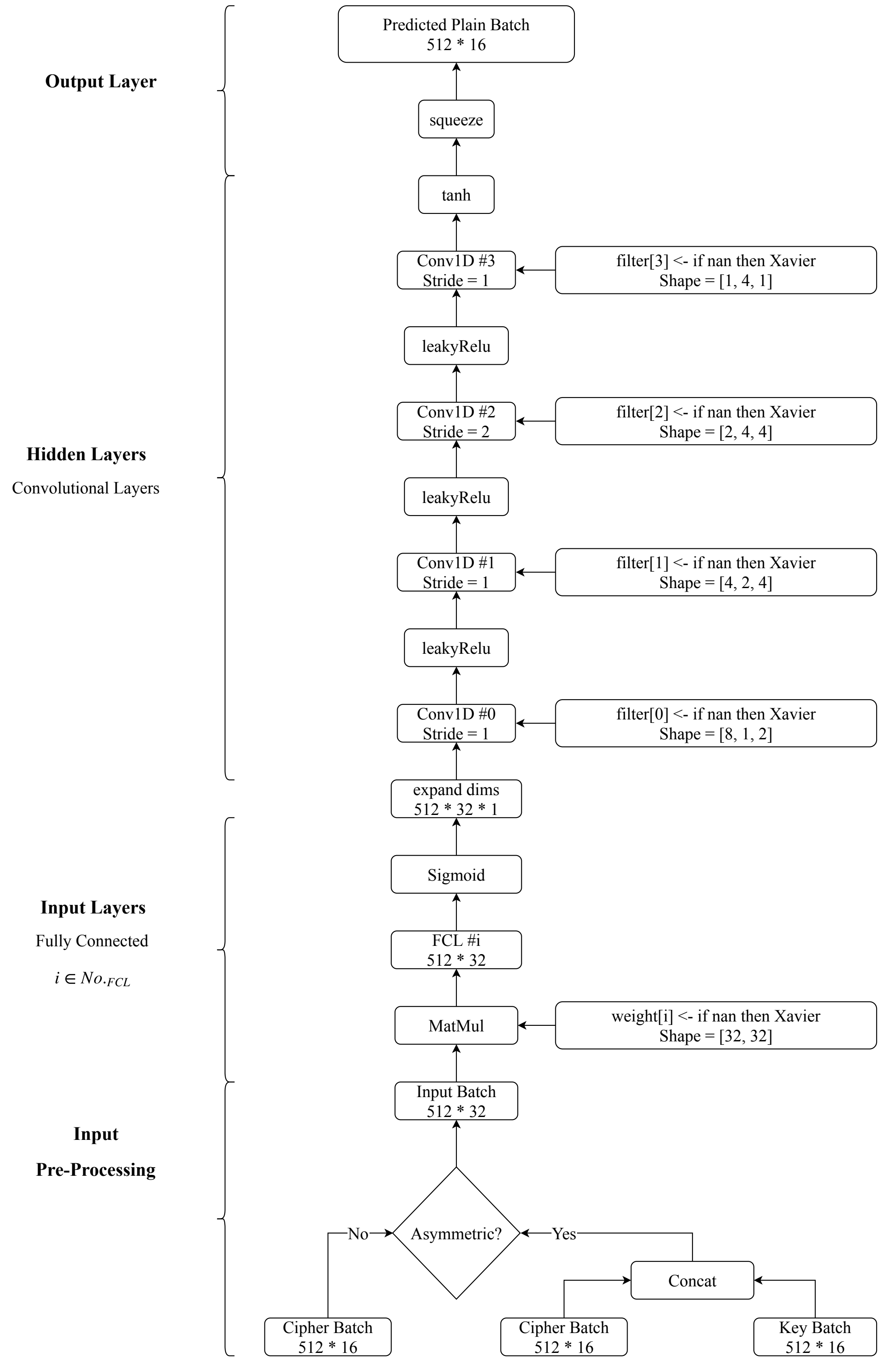


Fig. 3.3: Net Graph Diagram - Eve/Alan



3.2 Scheme Structures

Fig. 3.4: Symmetric Scheme

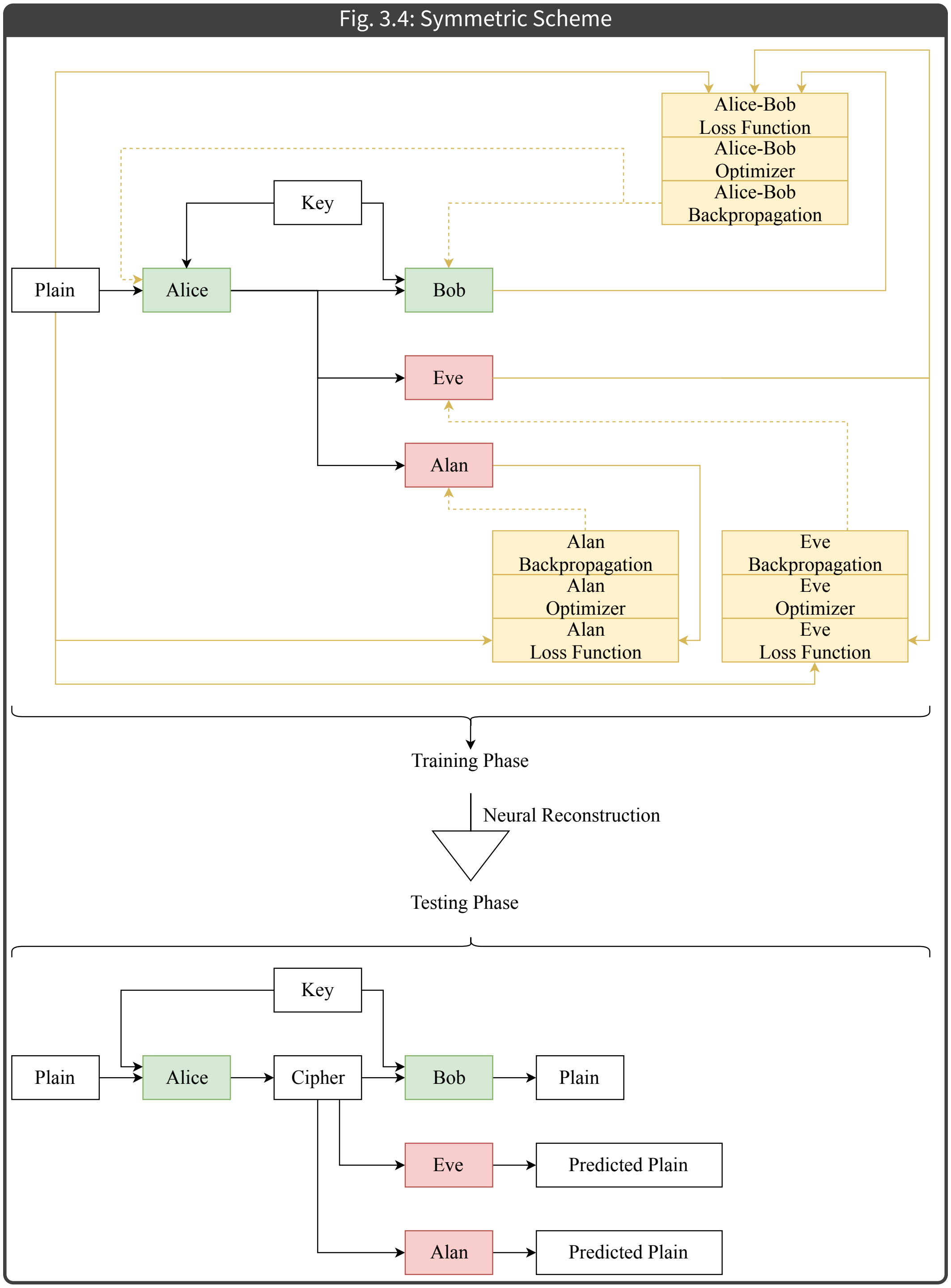


Fig. 3.5: Asymmetric Scheme

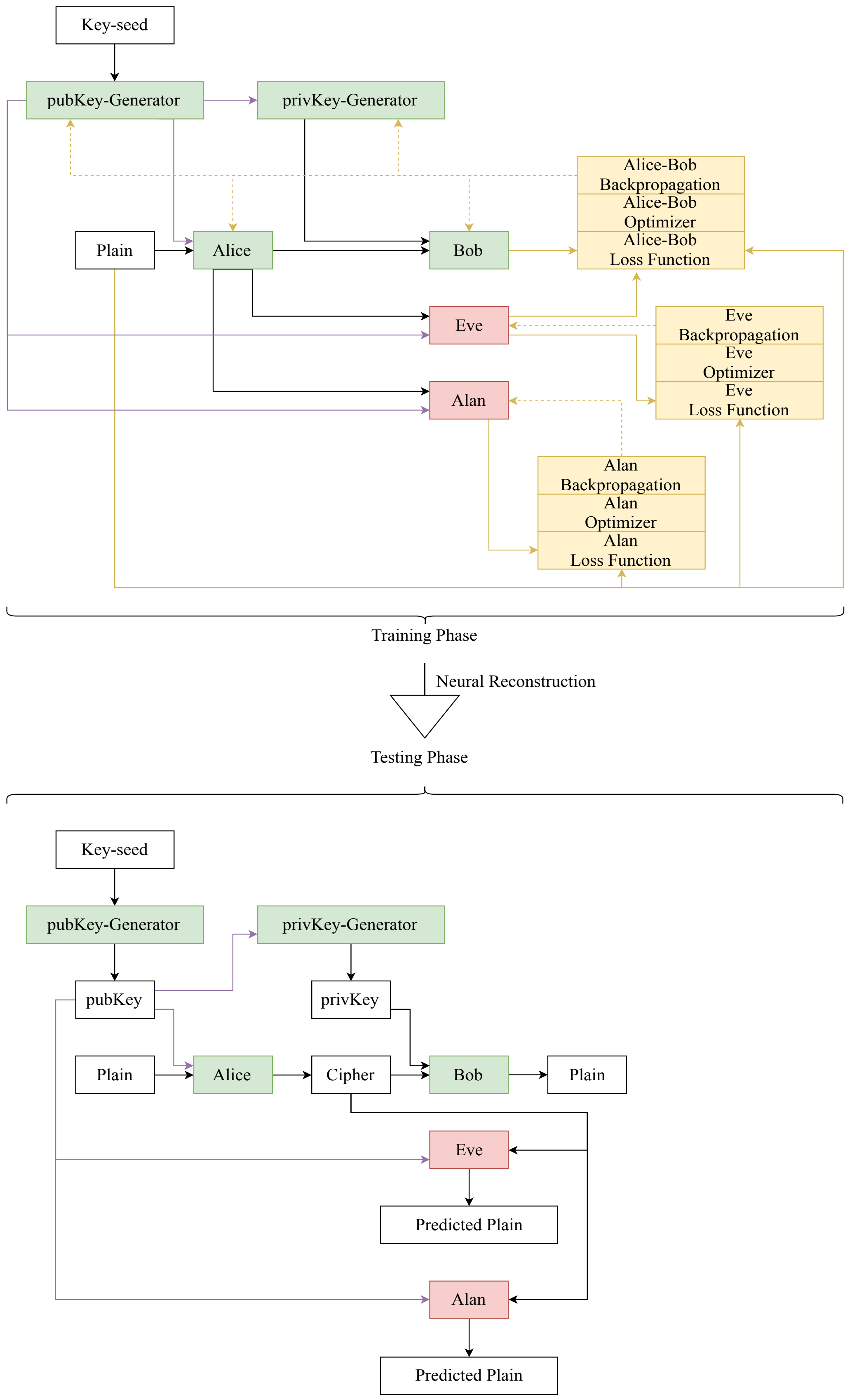


Fig. 3.6: Hybrid Scheme

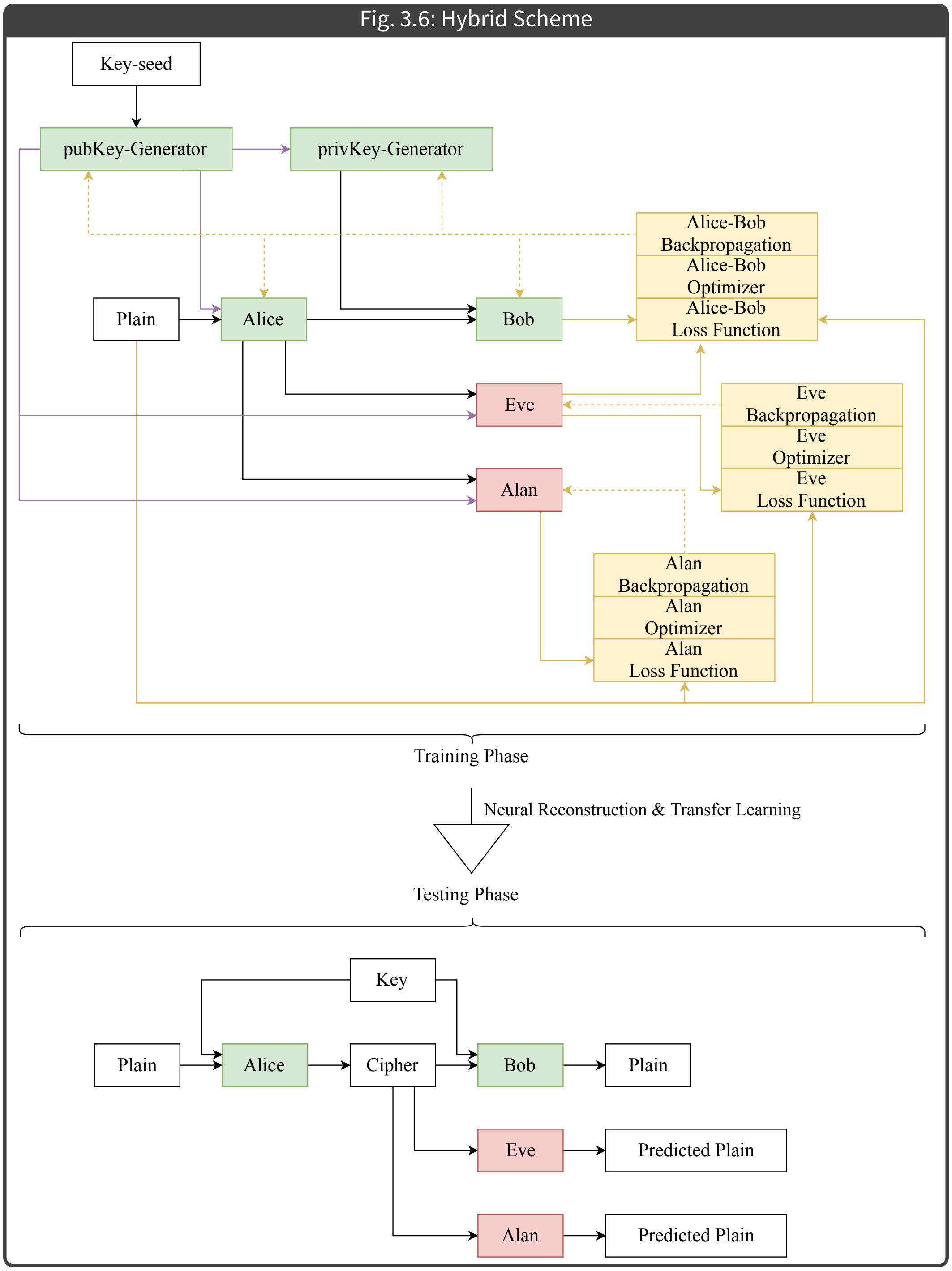
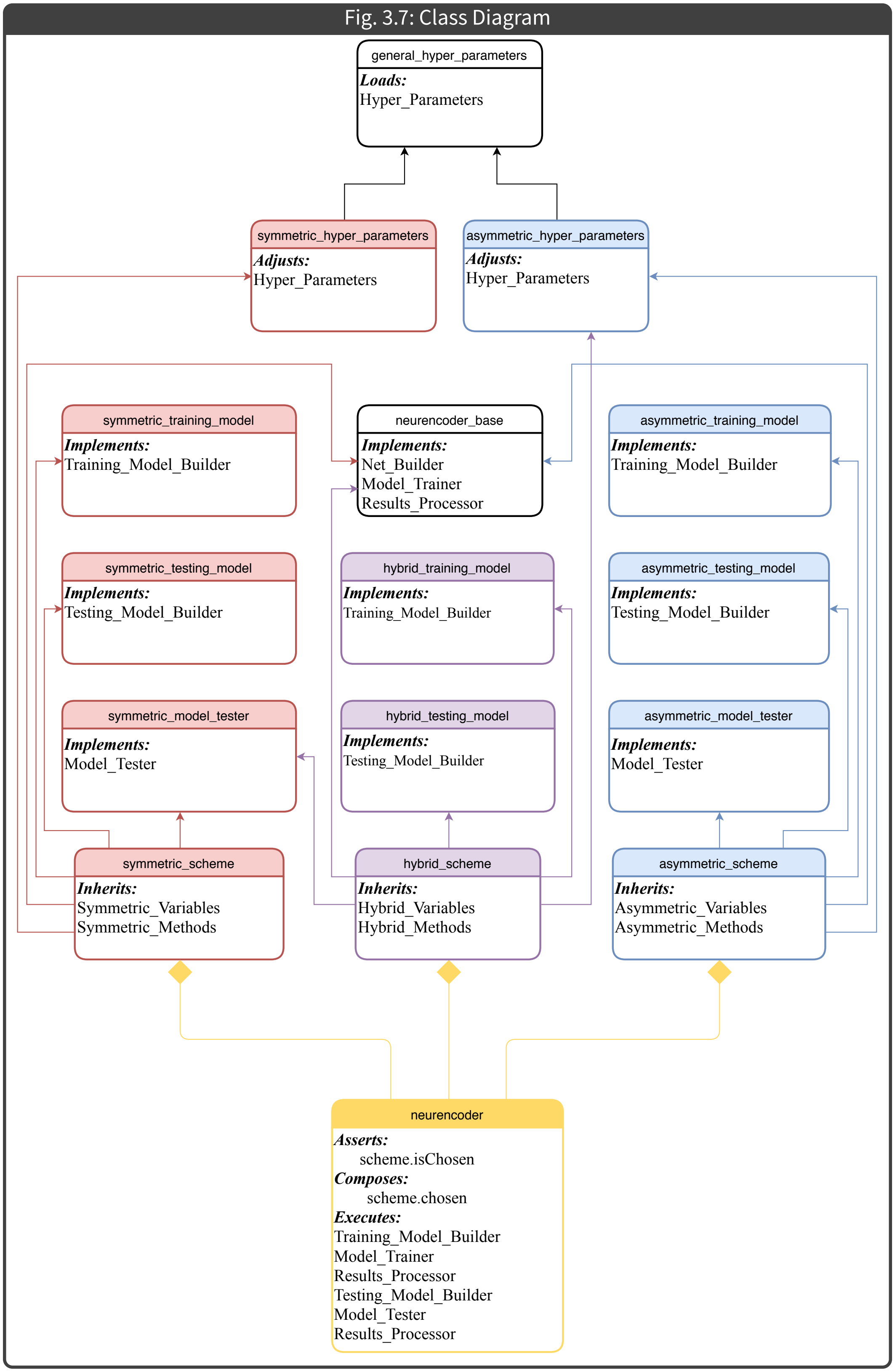


Fig. 3.7: Class Diagram



Chapter 4

Results

Chapter 5

Conclusion

Appendix