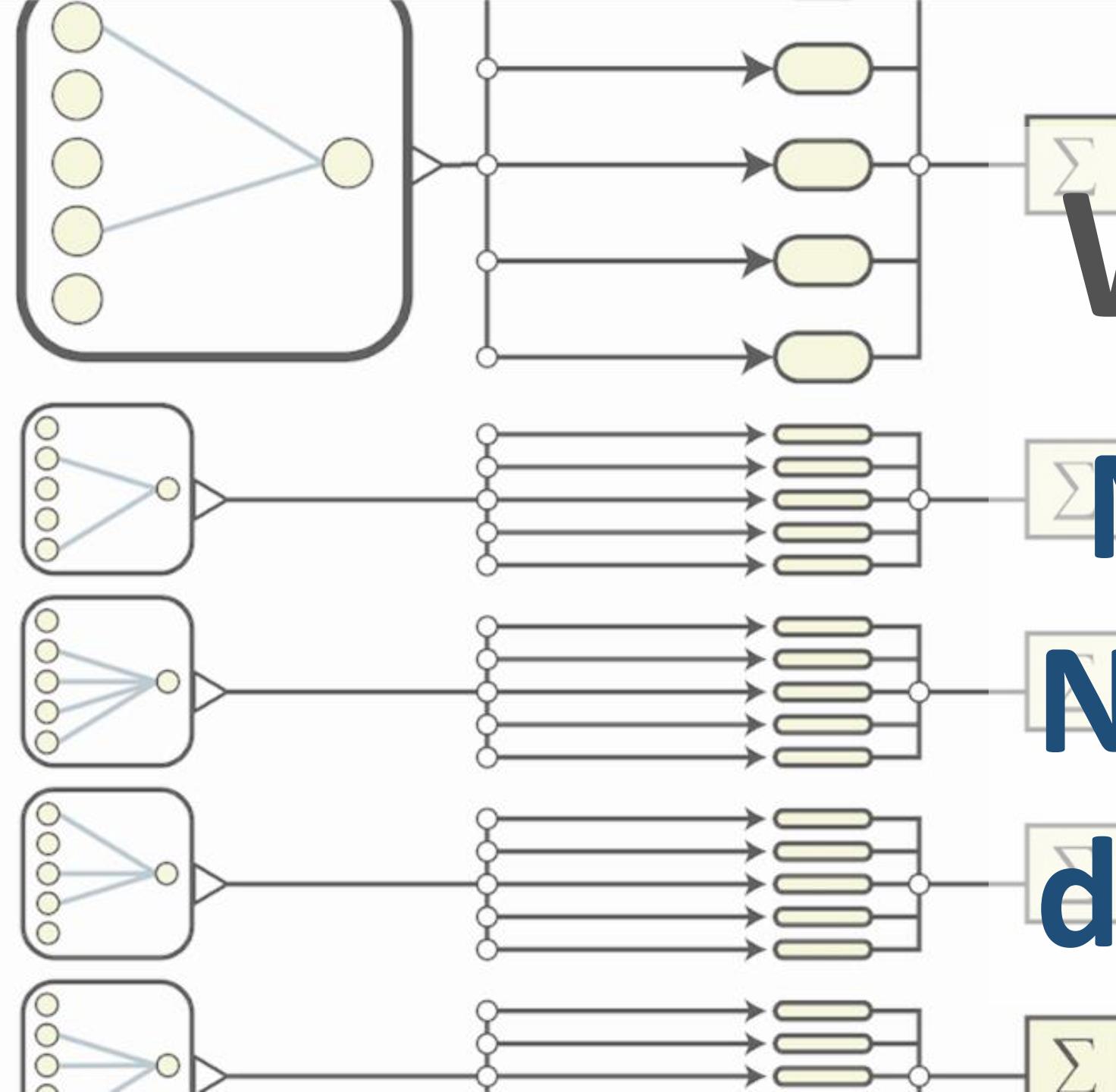


Wahlfach Neuronale Netze - KI in der Medizin



Themen des Wahlfachs

So., 10.12.2023

Crashkurs Python

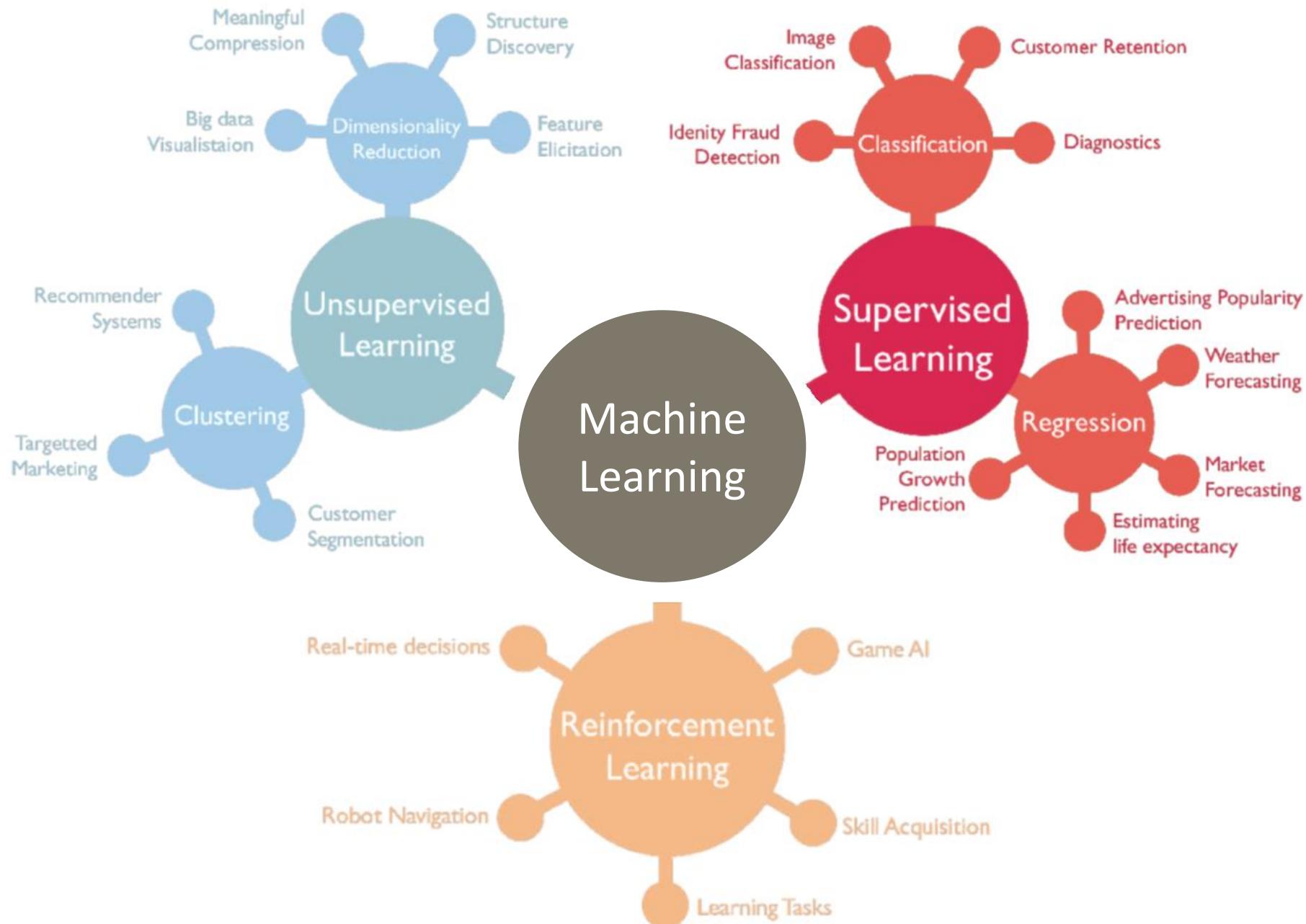
Sa., 16.12.2023

Crashkurs Machine Learning

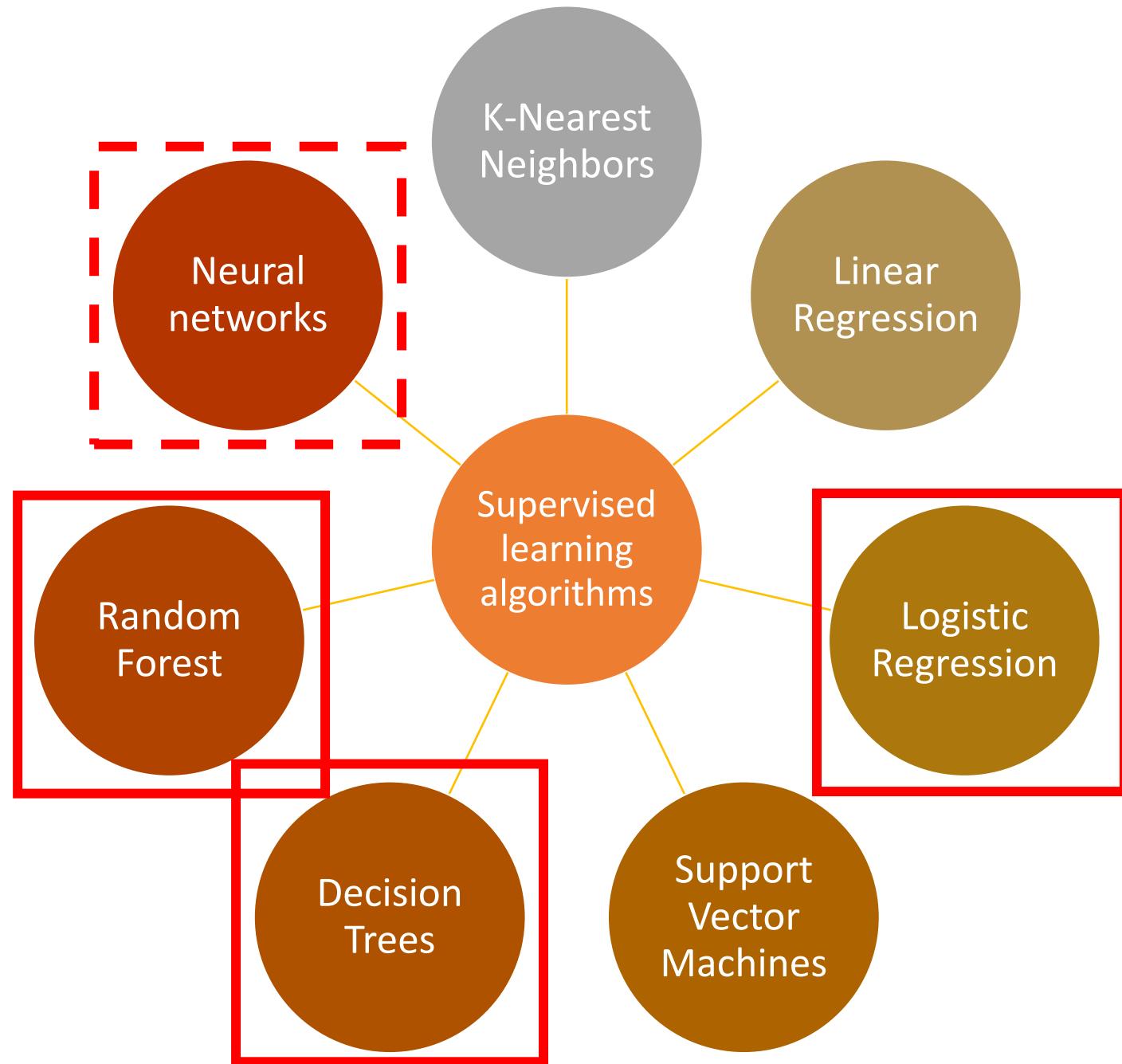
So., 17.12.2023

Crashkurs Deep Learning

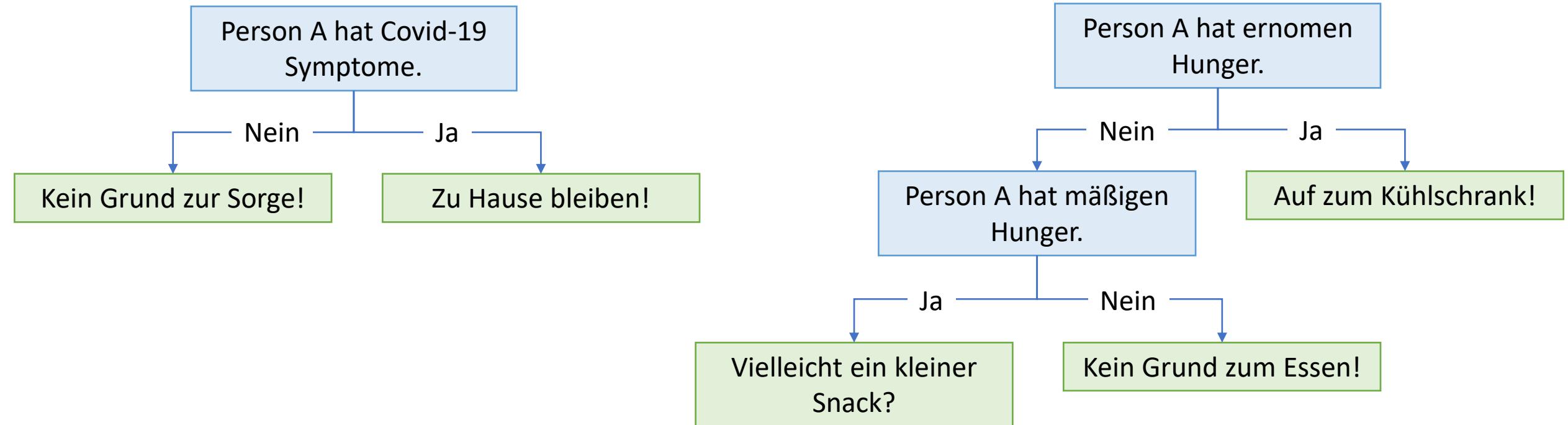
Welches Machine Learning System kommt am besten in Frage?



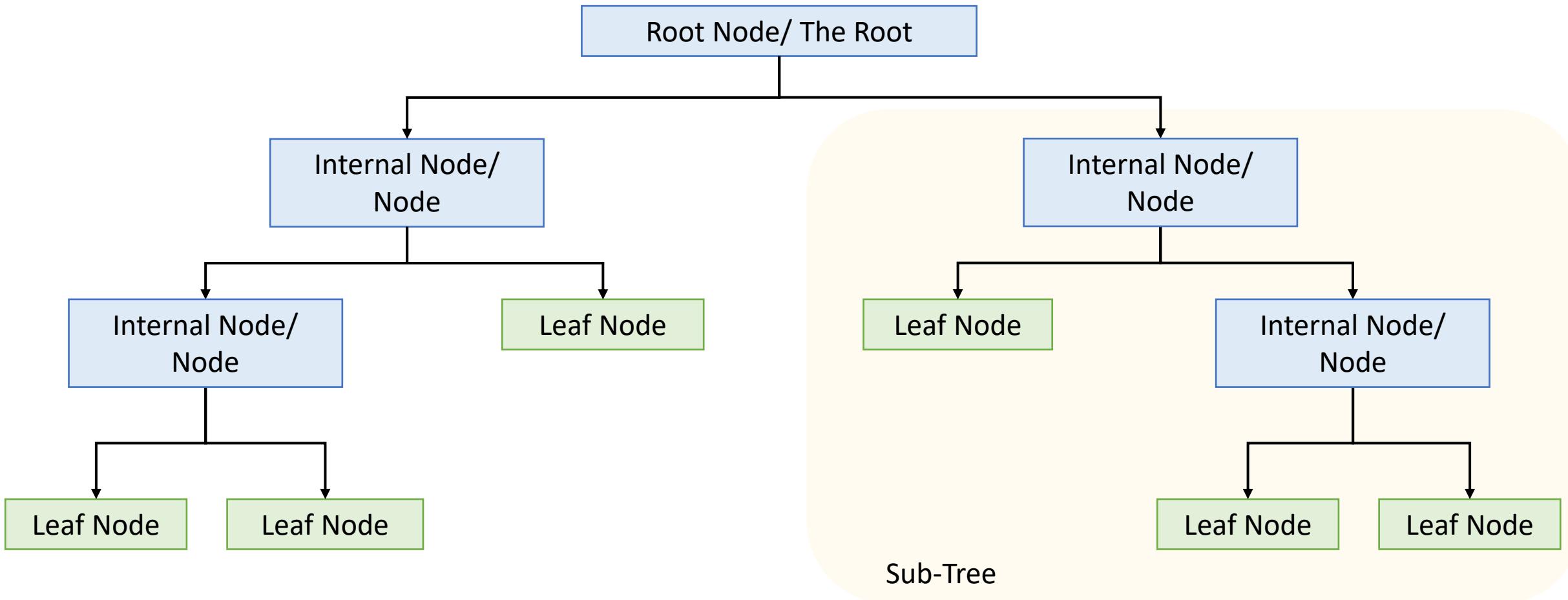
Die Qual der Wahl...



Im Auge des Sturms: Decision Trees

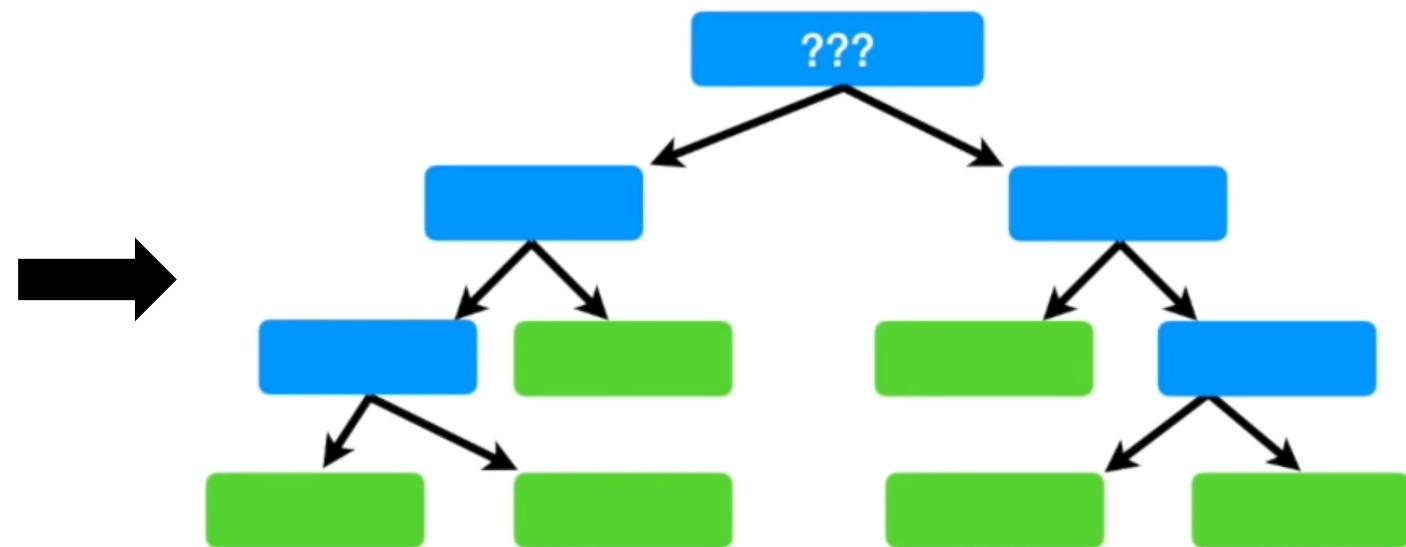


Im Auge des Sturms: Decision Trees



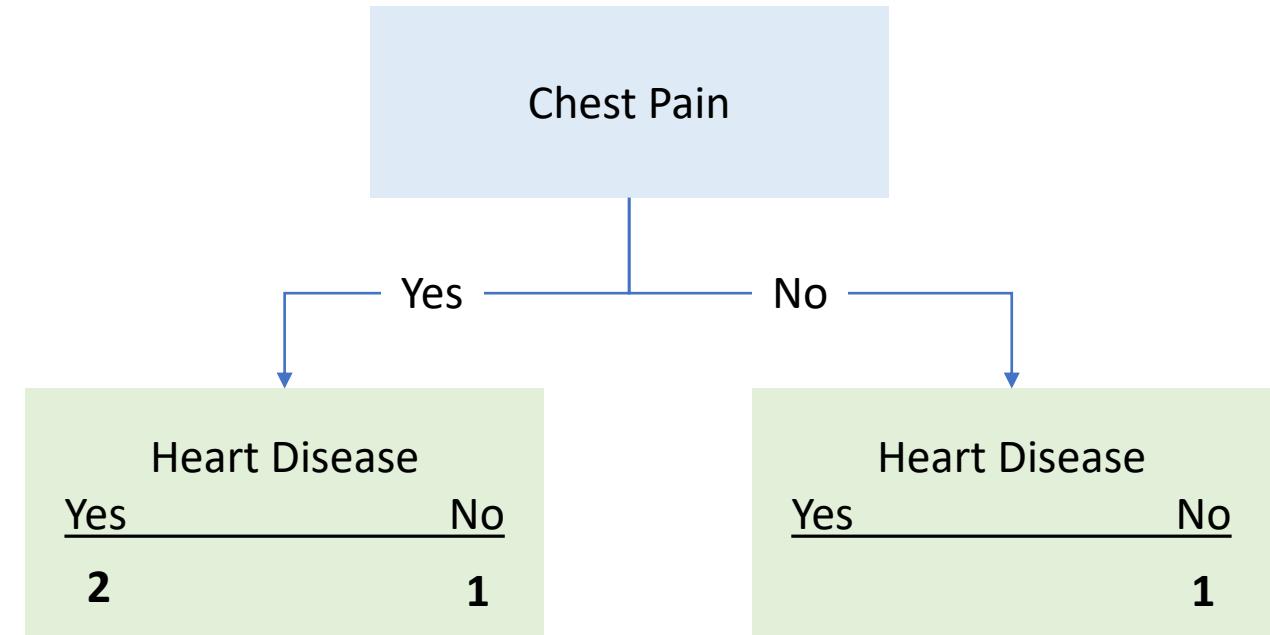
Im Auge des Sturms: Decision Trees

Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	??	Yes
...



Im Auge des Sturms: Decision Trees

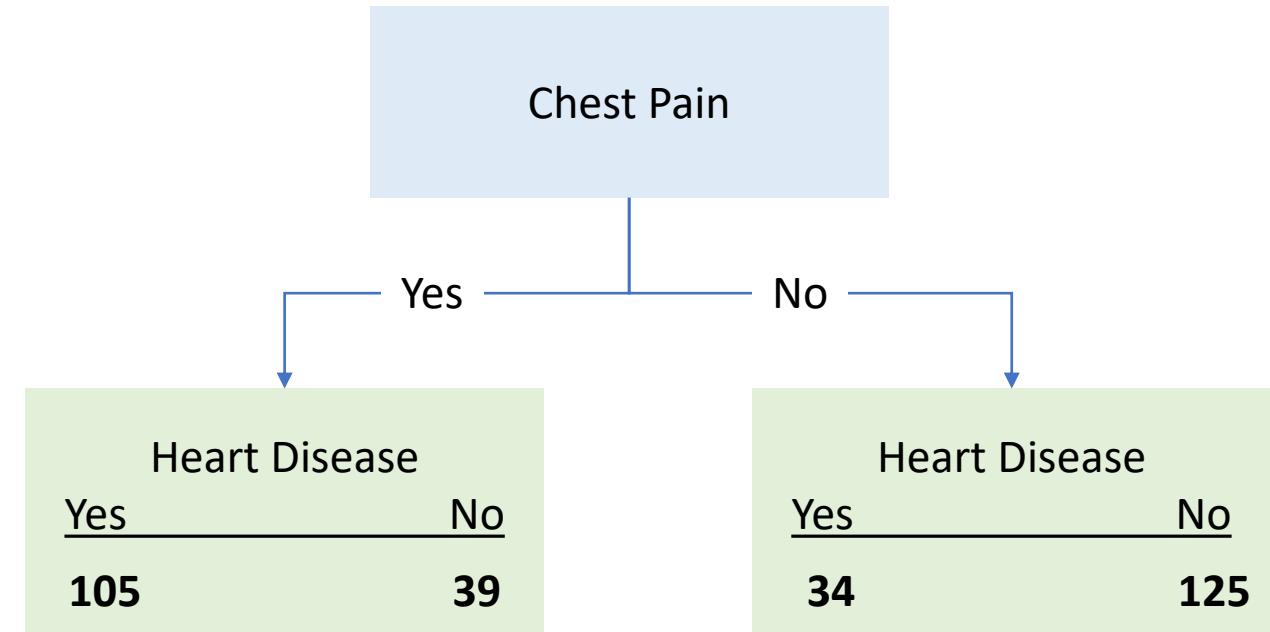
Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	??	Yes
...



Im Auge des Sturms: Decision Trees

Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	??	Yes
...

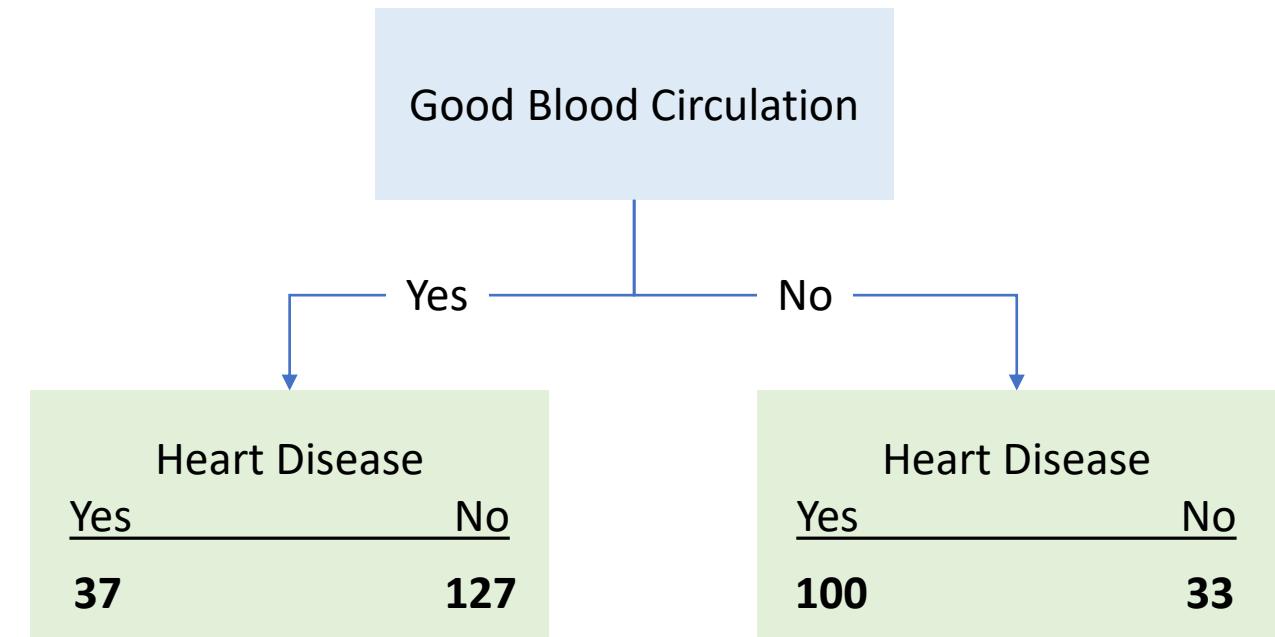
Total: 303 Patienten



Im Auge des Sturms: Decision Trees

Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	??	Yes
...

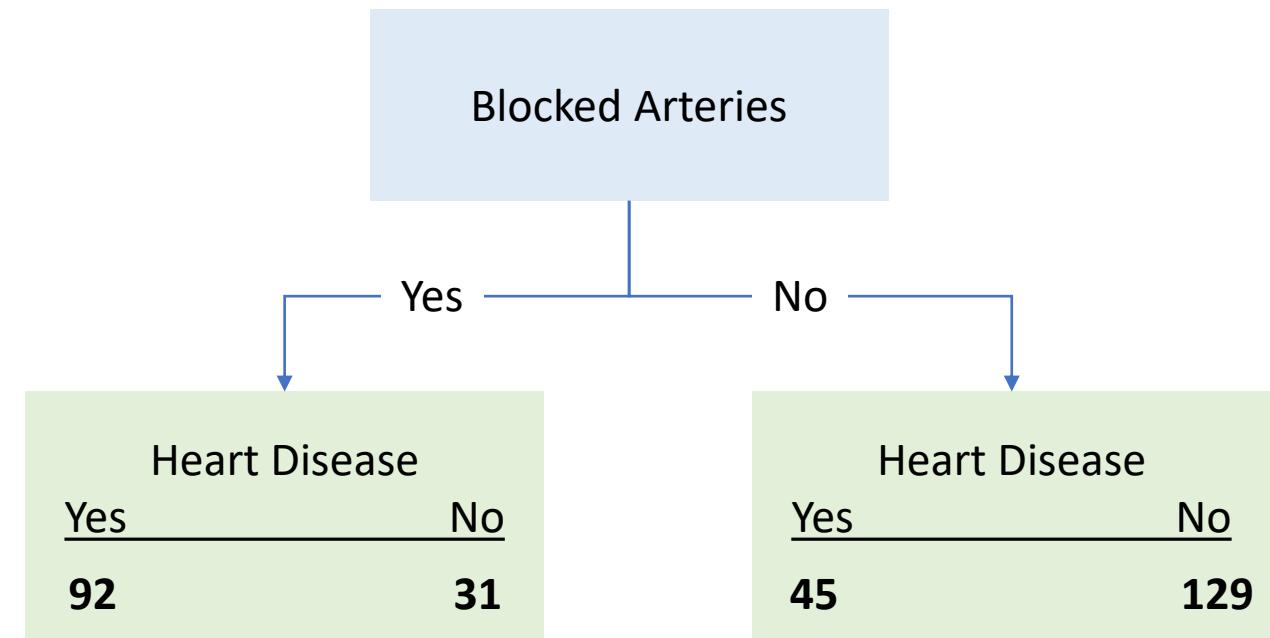
Total: 303 Patienten



Im Auge des Sturms: Decision Trees

Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	??	Yes
...

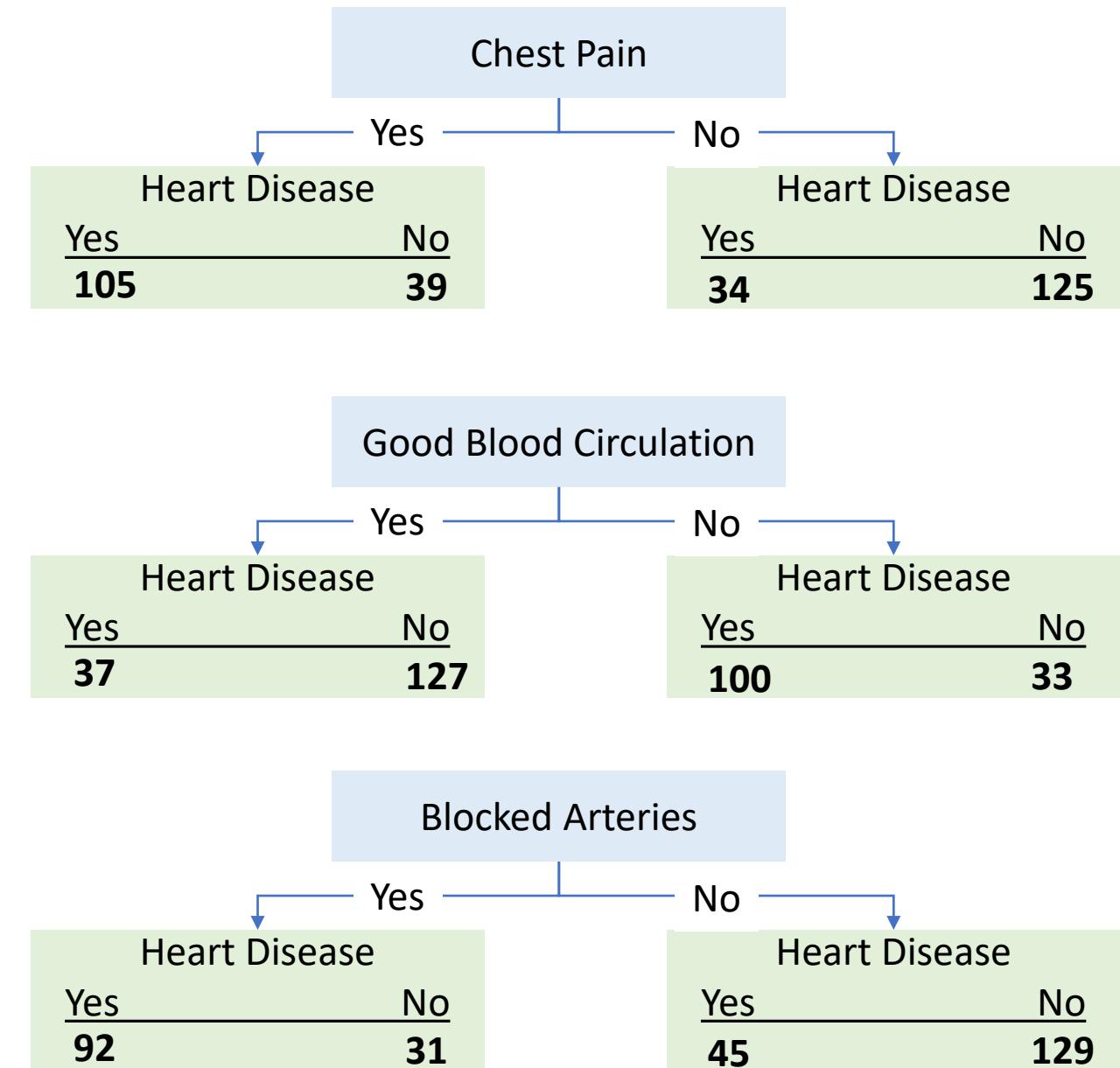
Total: 303 Patienten



Im Auge des Sturms: Decision Trees

Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	??	Yes
...

Total: 303 Patienten



Wir sehen alle unserer Parameter erweisen sich als „impure“ (= unrein), d.h. keiner wäre als „Root Node“ perfekt geeignet.



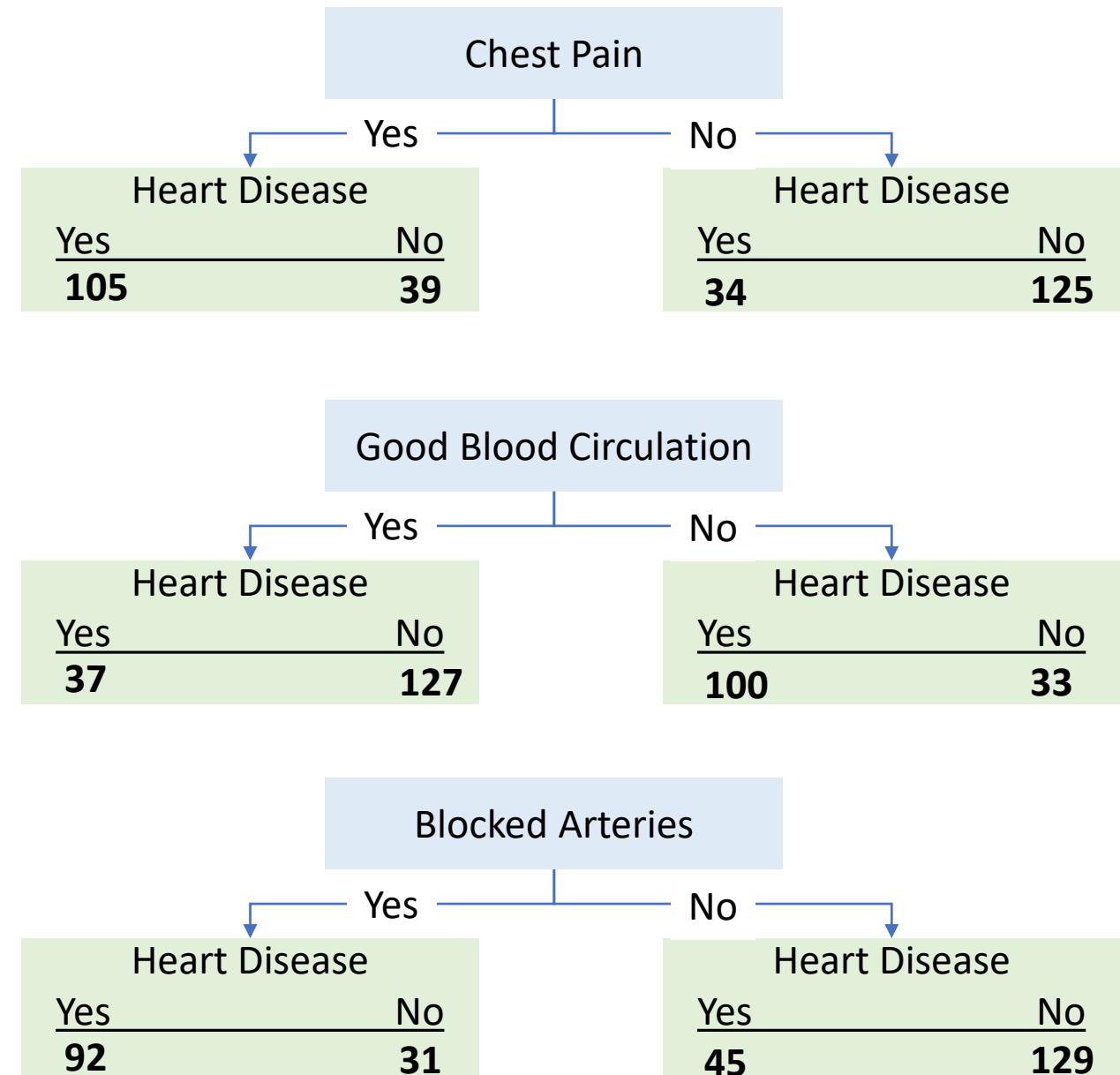
Wir machen einfach das Beste daraus und bestimmen den Reinsten aller Unreinen.



Aber wie misst man „**impurity**“?



$$\text{Gini impurity} = 1 - (\text{probability of yes})^2 - (\text{probability of no})^2$$



Im Auge des Sturms: Decision Trees

Gini-impurity des linken „Leaf“:

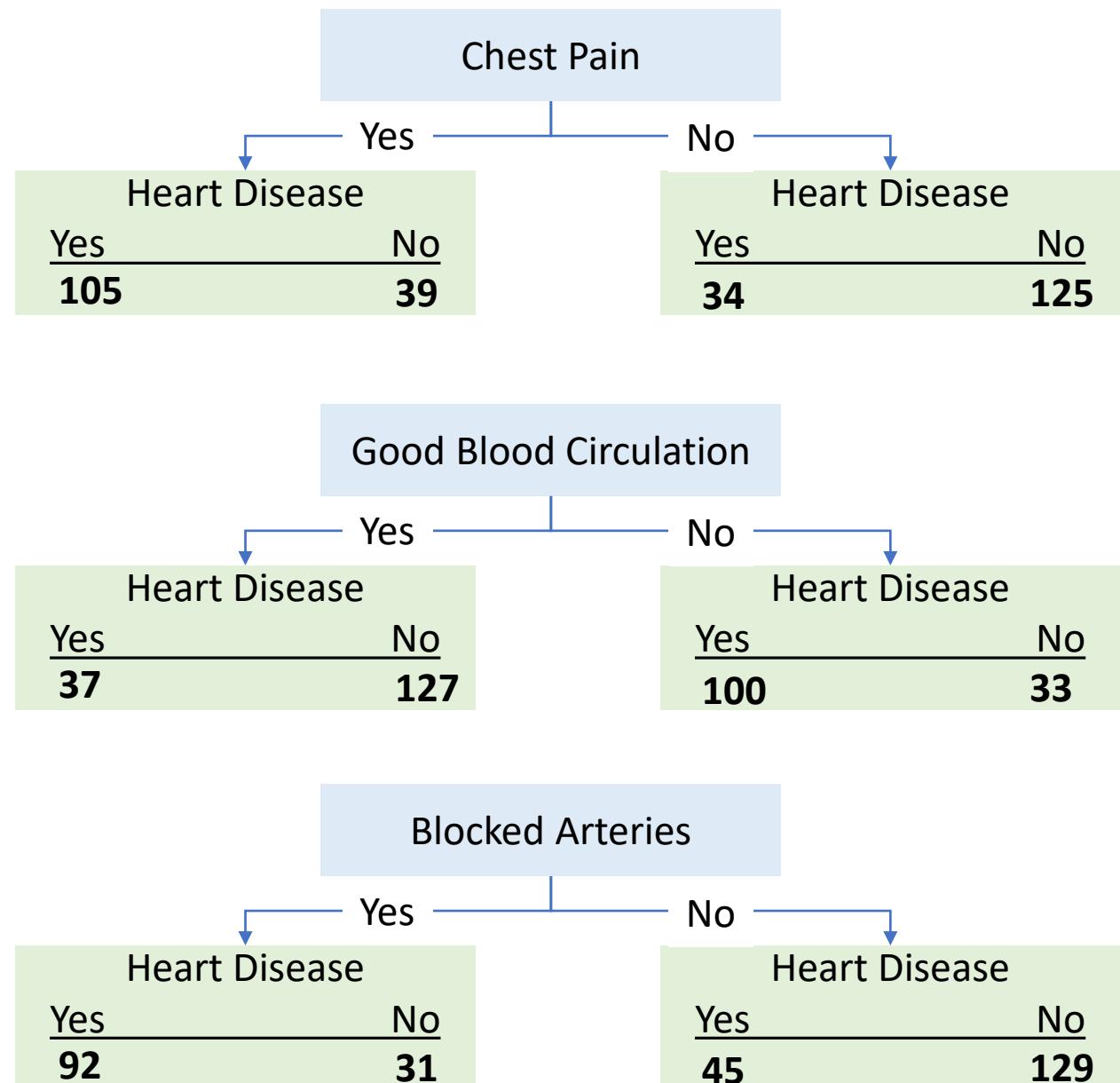
$$1 - \left(\frac{105}{105 + 39} \right)^2 - \left(\frac{39}{105 + 39} \right)^2 = 0.395$$

Gini-impurity des rechten „Leaf“:

$$1 - \left(\frac{34}{34 + 125} \right)^2 - \left(\frac{125}{34 + 125} \right)^2 = 0.336$$

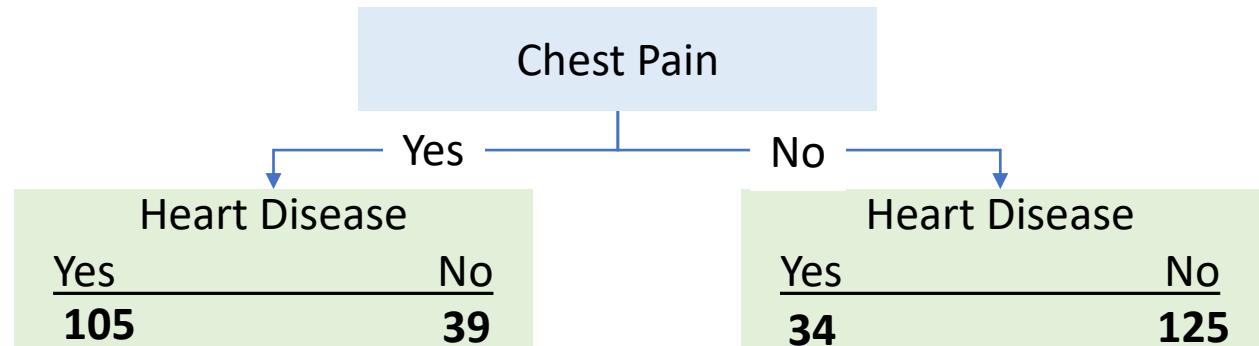
Gewichtete Gini-impurity für „Chest Pain“:

$$\left(\frac{144}{144 + 159} \right) * 0,395 + \left(\frac{159}{144 + 159} \right) * 0,336 = 0,364$$



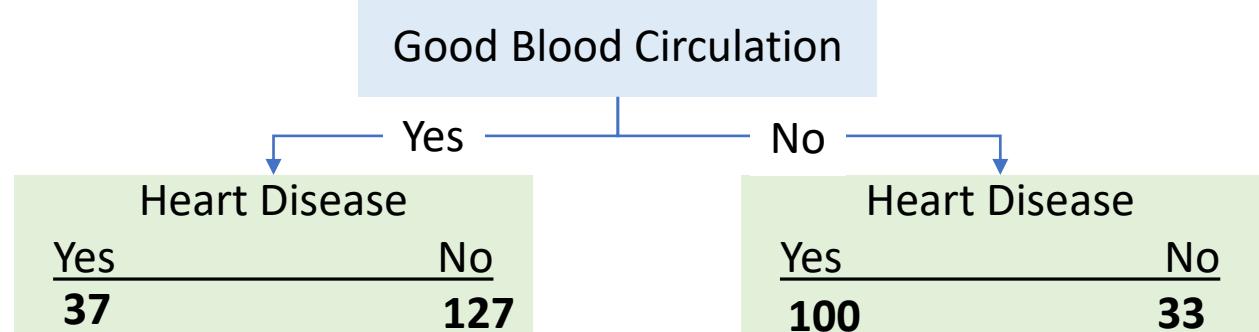
Im Auge des Sturms: Decision Trees

Gini-impurity für „Chest Pain“ = 0,364

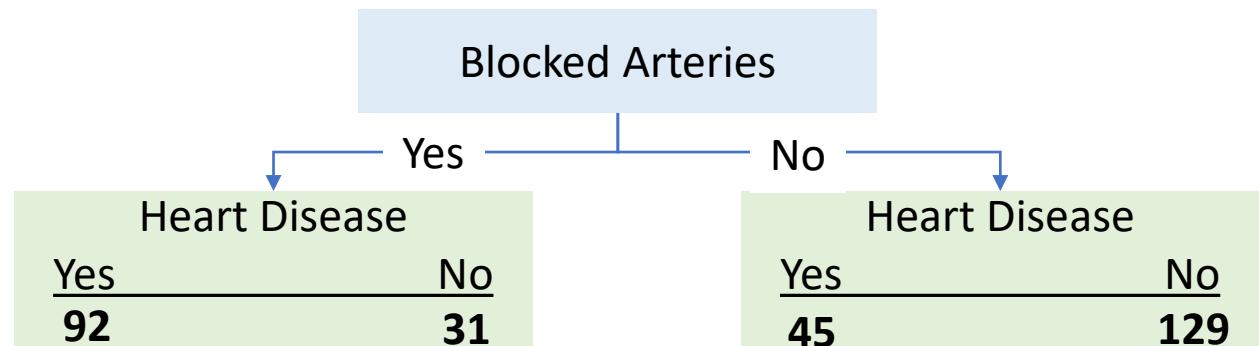


Gini-impurity für „Good Blood Circulation“ = 0,360

Good Blood Circulation hat folglich die geringste „impurity“, sodass es als bestmöglicher Root Node in Frage kommt!



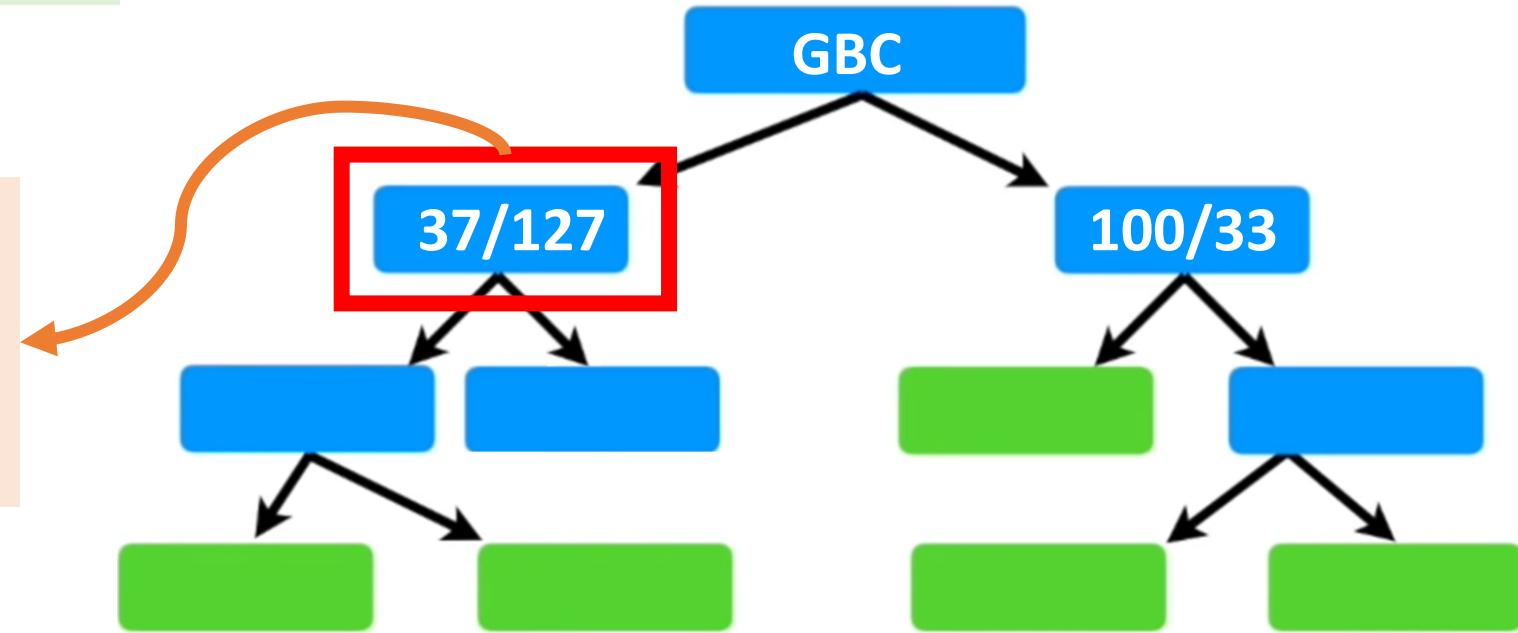
Gini-impurity für „Blocked Arteries“ = 0,381



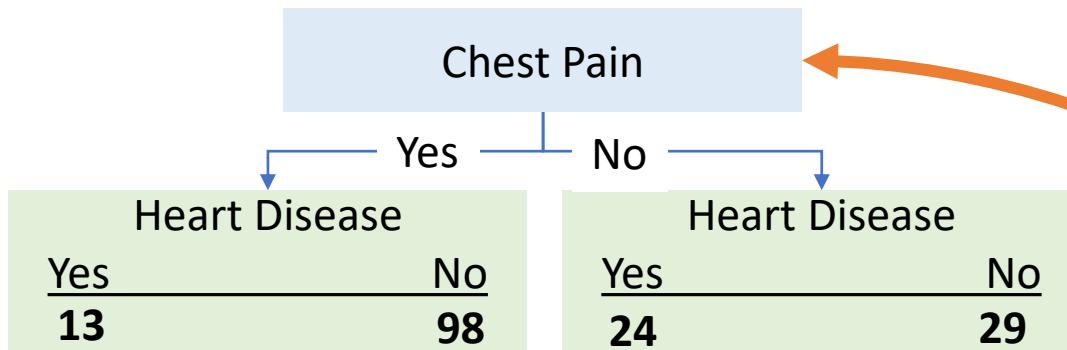
Im Auge des Sturms: Decision Trees

Good Blood Circulation			
Heart Disease		Heart Disease	
Yes	No	Yes	No
37	127	100	33

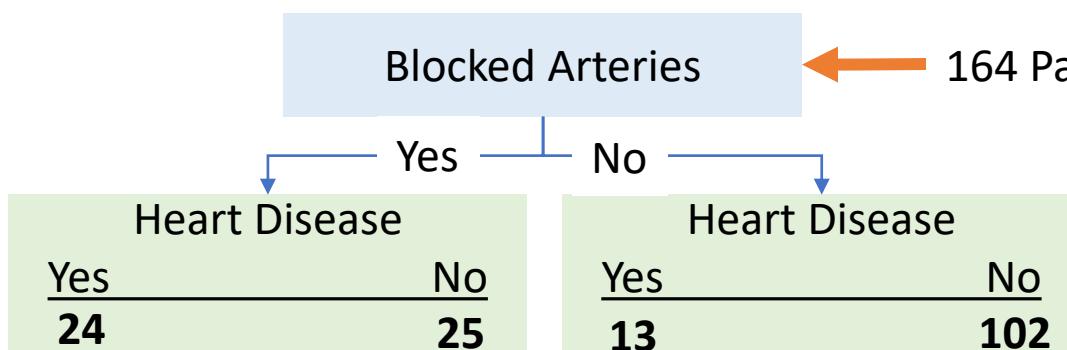
Aber wie fahren wir mit diesen 164 Patientendaten fort?
Welchen Parameter nehmen wir als nächsten Internal Node?



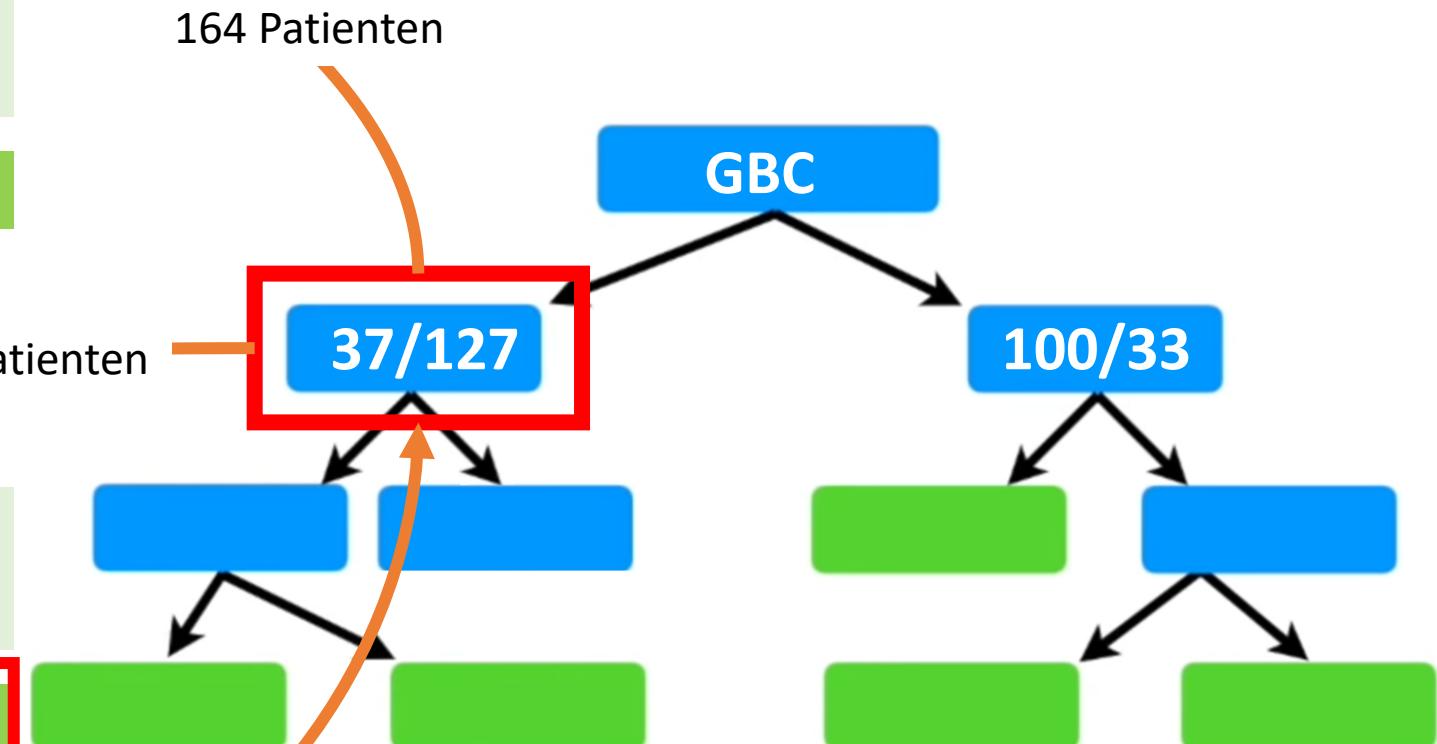
Im Auge des Sturms: Decision Trees



Gini impurity für Chest Pain = 0,3



Gini impurity für Blocked Arteries = 0,290



→ Blocked Arteries hat folglich die geringste „impurity“, sodass es als bestmöglicher Internal Node in Frage kommt!

Im Auge des Sturms: Decision Trees

Blocked Arteries

Yes

Heart Disease

24

No

Heart Disease

25

Gini impurity für Blocked Arteries = 0,520

Chest Pain

Yes

Heart Disease

Yes

17

No

3

Yes

7

Heart Disease

Yes

22

No

22

Gini impurity für Chest Pain = 0,218



Zwar nicht ganz perfekt, aber besser als wenn wir es beim letzten Internal Node gelassen hätten.

GBC

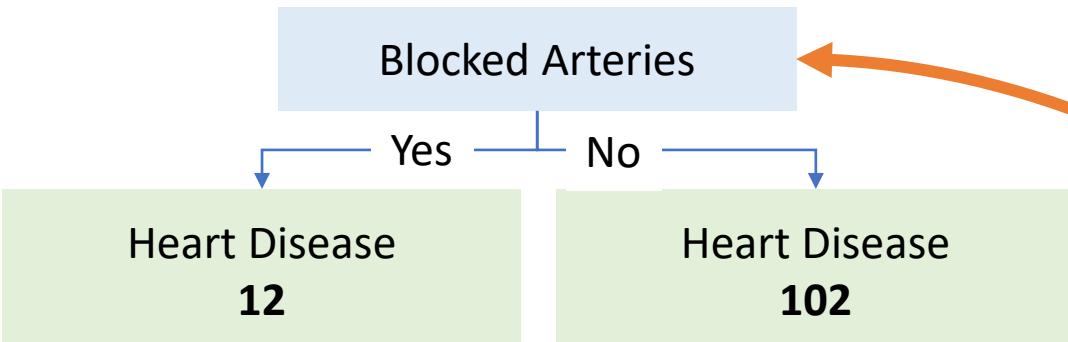
Blocked

24/25

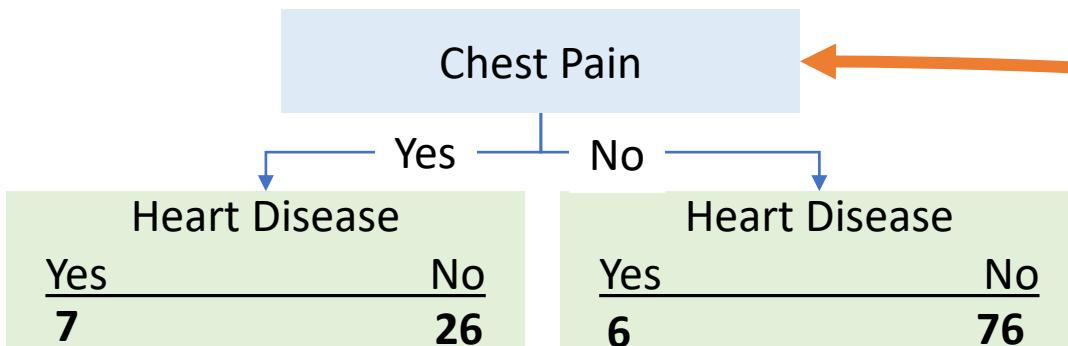
13/102

100/33

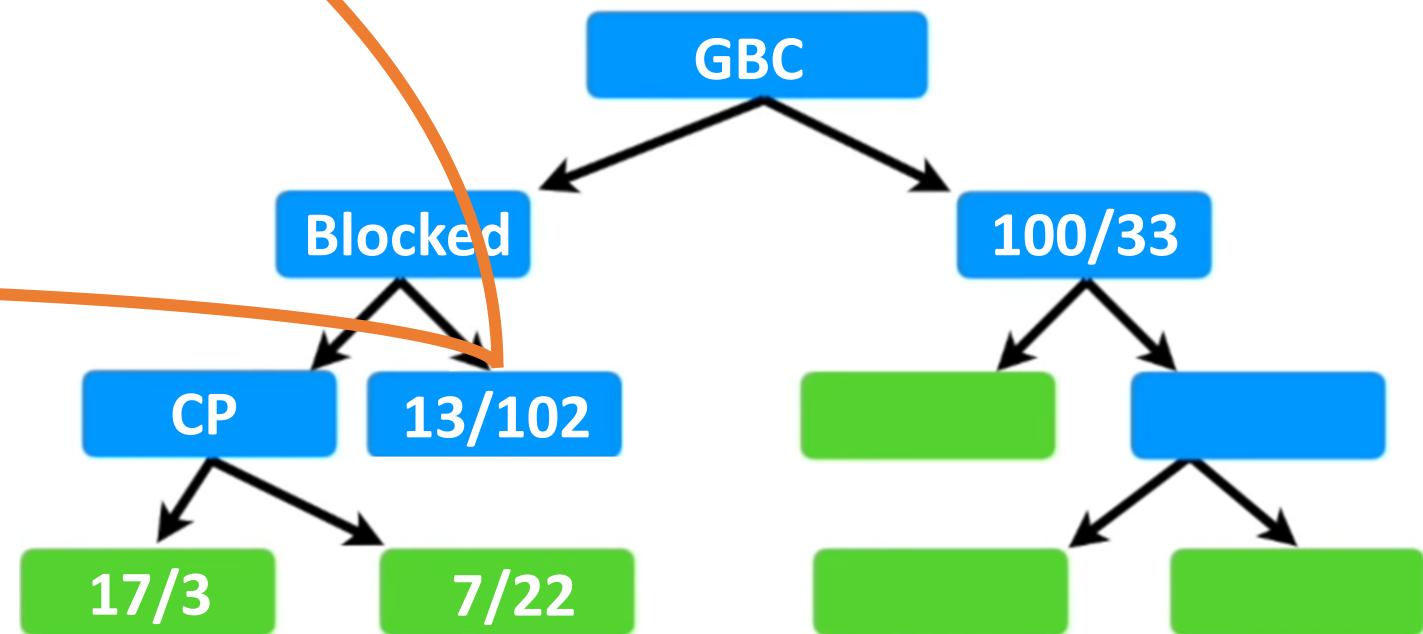
Im Auge des Sturms: Decision Trees



Gini impurity für Blocked Arteries = 0,2



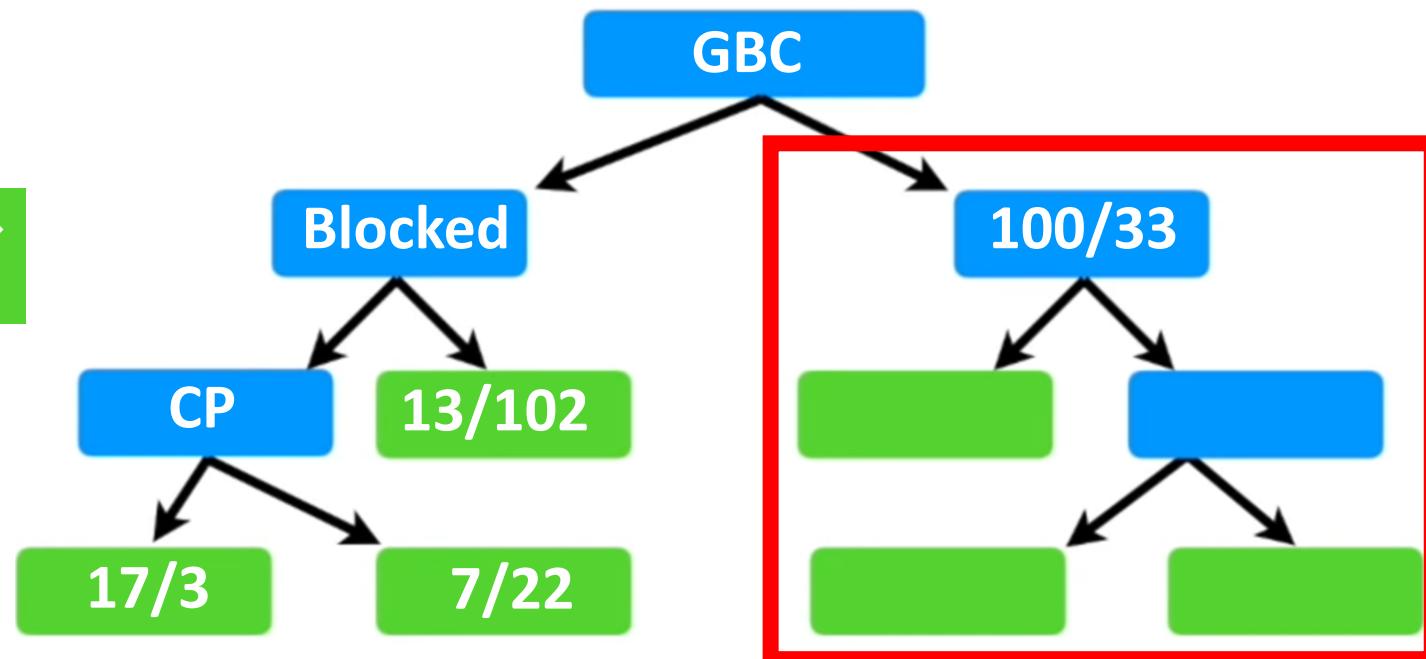
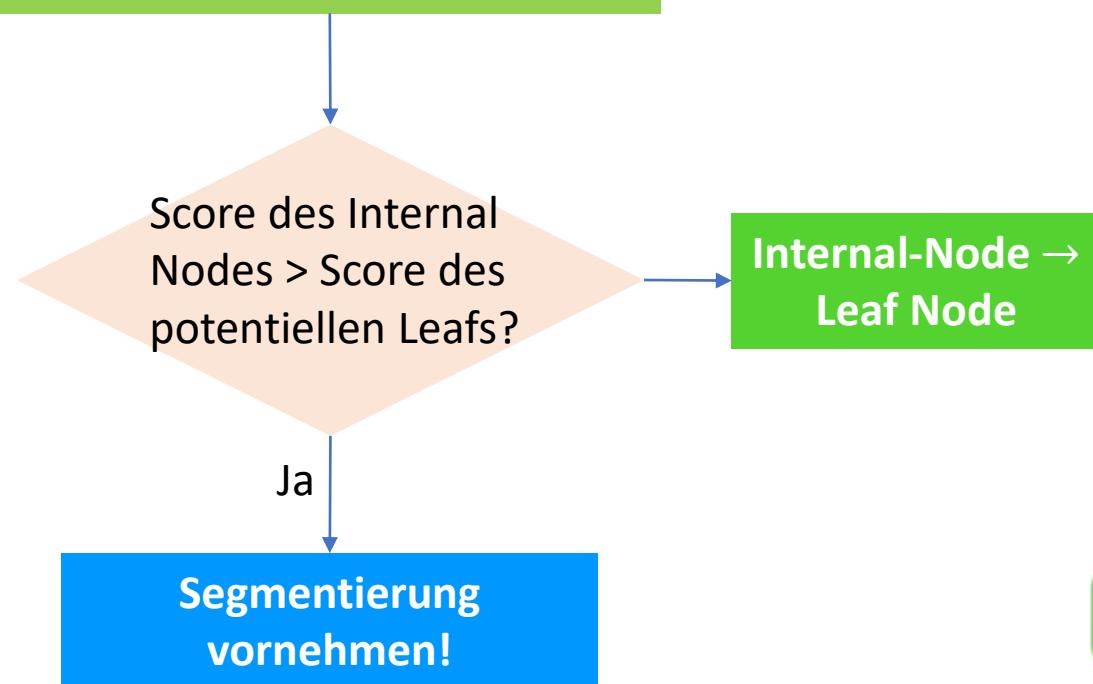
Gini impurity für Chest Pain = 0,29



Die Impurity wird schlechter, wenn wir Chest Pain miteinbeziehen. Wir fügen hier daher keinen neuen Node ein!

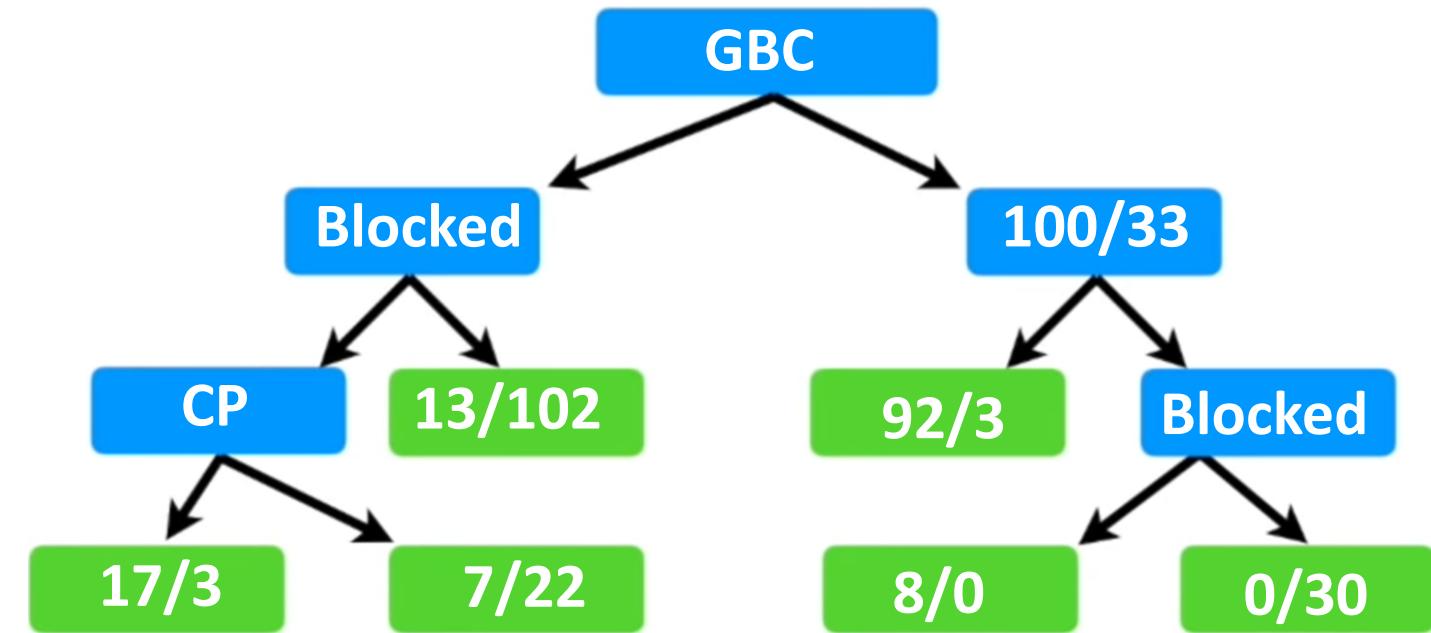
Nun fehlt nur noch die rechte Seite, wo wir nach dem gleichen Schema fortfahren:

Gini impurity scores berechnen.

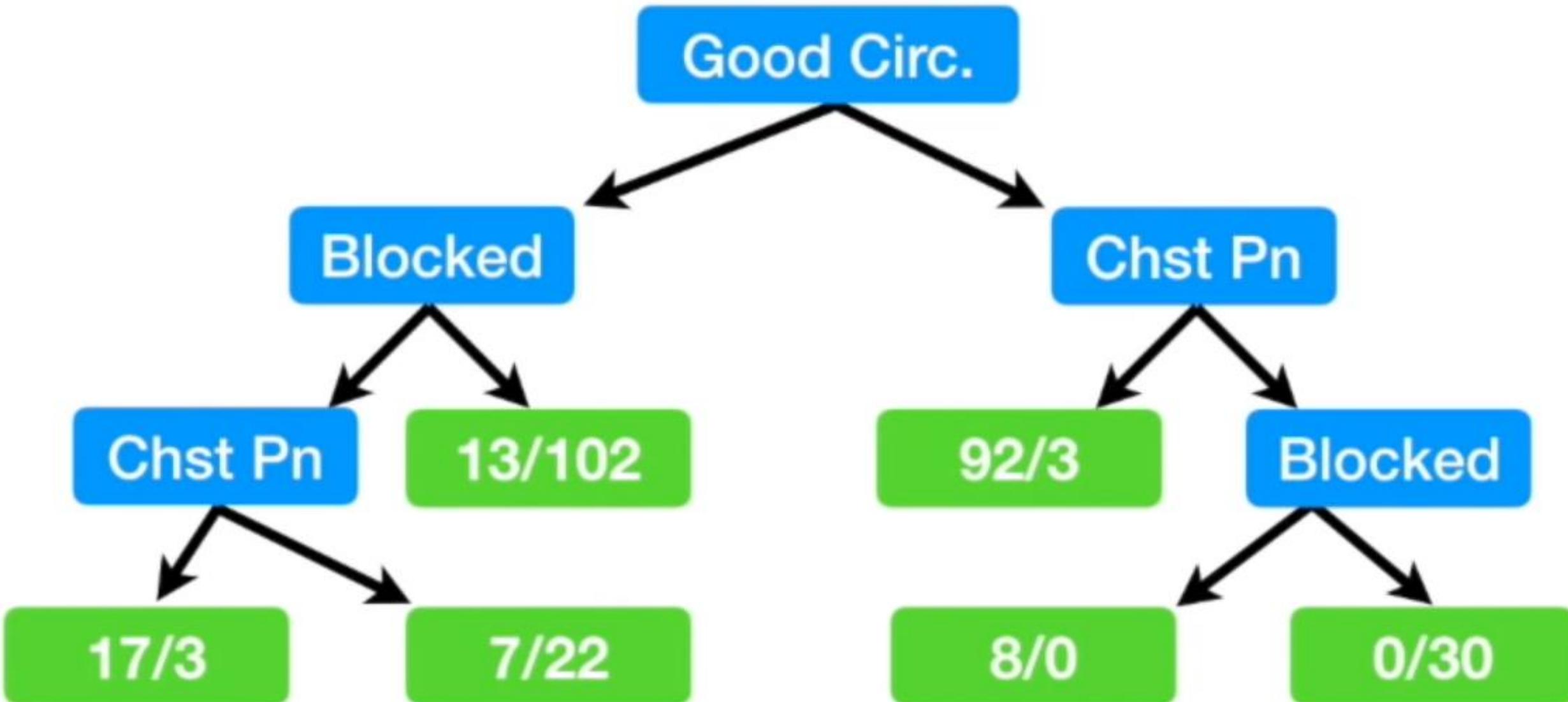


Im Auge des Sturms: Decision Trees

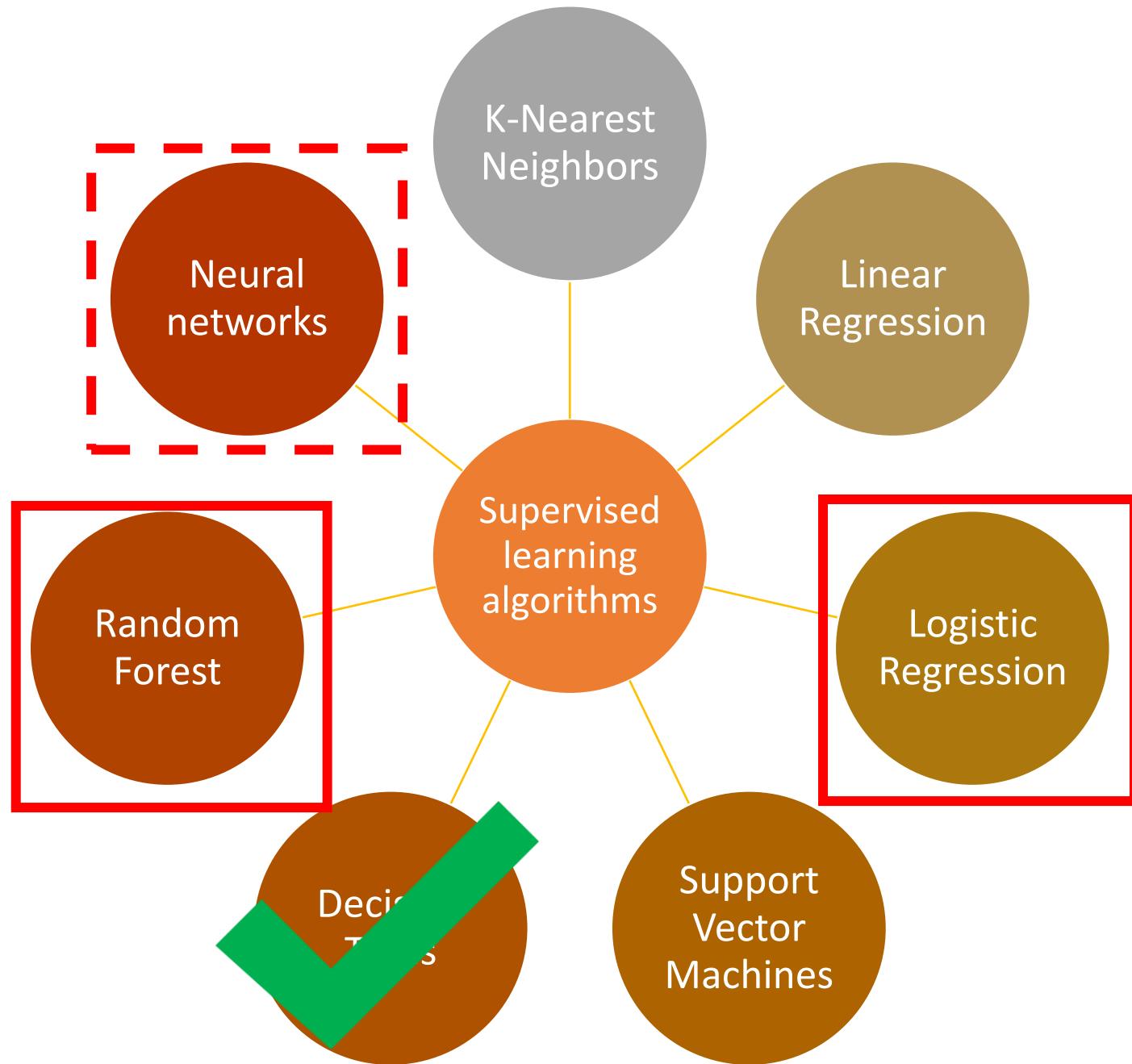
Chest Pain	Good Blood Circulation	Blocked Arteries	Heart Disease
No	No	No	No
Yes	Yes	Yes	Yes
Yes	Yes	No	No
Yes	No	??	Yes
...



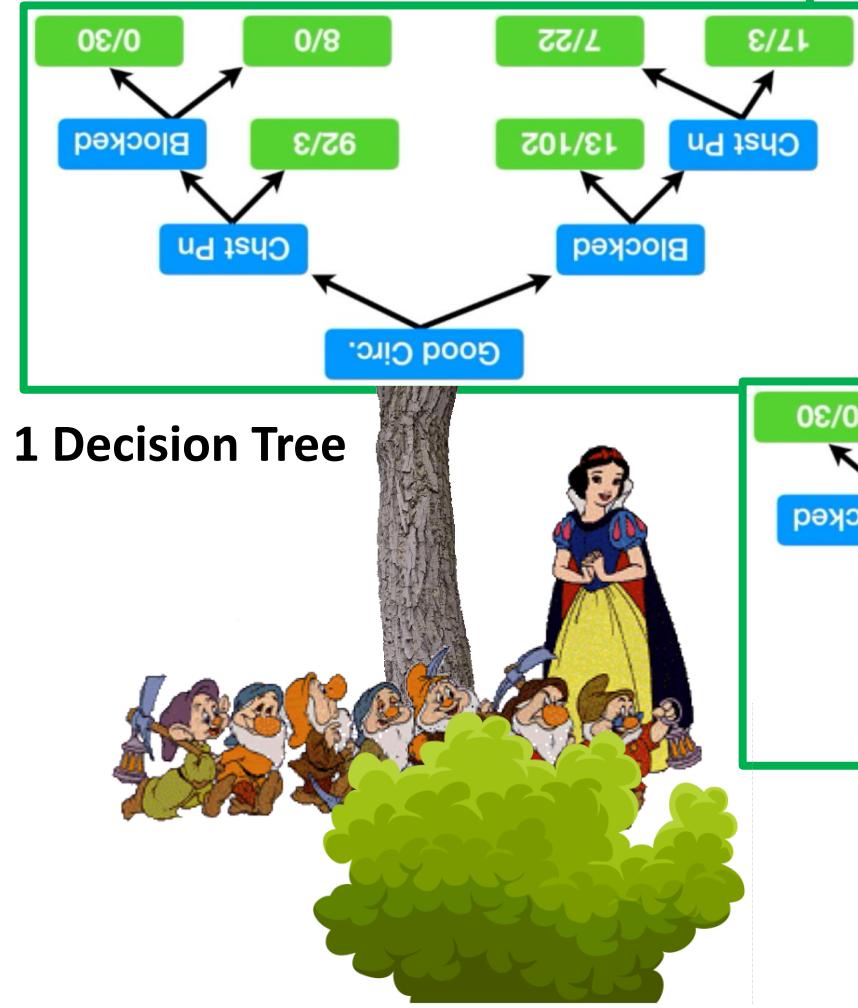
Unsere Künstliche Intelligenz zur Klassifikation:



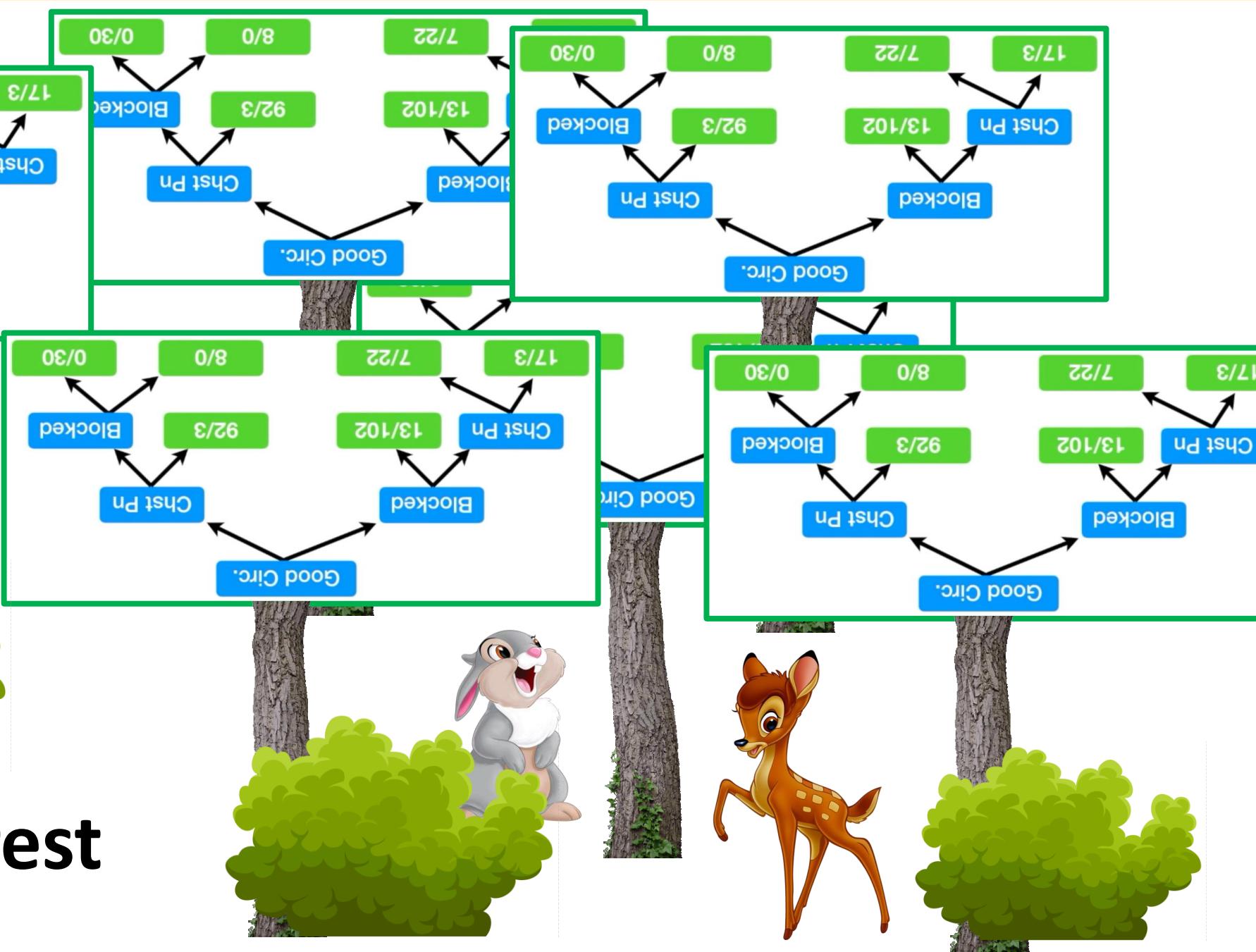
Die Qual der Wahl...



Im Auge des Sturms: Random Forest



1 Random Forest



Im Auge des Sturms: Random Forest

Schritt 1: Wir kreieren einen „bootstrapped dataset“, indem wir

- a) zufällig Zeilen vom originalen Dataset auswählen
- b) dabei auch eine Zeile mehrmals nehmen -> random!

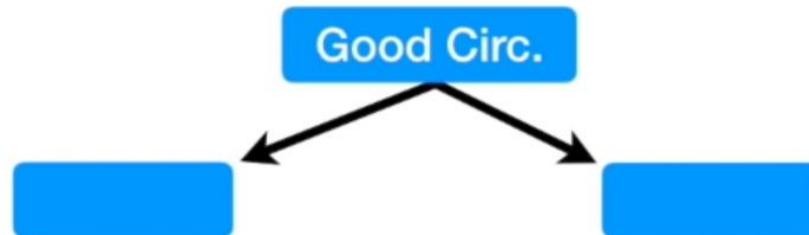
Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease		Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No		Yes	Yes	Yes	180	Yes
Yes	Yes	Yes	180	Yes		No	No	No	125	No
Yes	Yes	No	210	No		Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes		Yes	No	Yes	167	Yes

The diagram illustrates the process of creating a bootstrapped dataset. It shows two tables side-by-side. The left table represents the original dataset with 5 rows. The right table represents the bootstrapped dataset with 4 rows. Blue arrows point from specific rows in the original dataset to the corresponding positions in the bootstrapped dataset, indicating which rows are selected. For example, the first row ('No', 'No', 'No', 125, 'No') is selected for the first row of the bootstrapped dataset ('Yes', 'Yes', 'Yes', 180, 'Yes'). The second row ('Yes', 'Yes', 'Yes', 180, 'Yes') is selected for the second row of the bootstrapped dataset ('No', 'No', 'No', 125, 'No'). The third row ('Yes', 'Yes', 'No', 210, 'No') is selected for the third row of the bootstrapped dataset ('Yes', 'No', 'Yes', 167, 'Yes'). The fourth row ('Yes', 'No', 'Yes', 167, 'Yes') is selected for the fourth row of the bootstrapped dataset ('Yes', 'No', 'Yes', 167, 'Yes'). This visualizes how random sampling with replacement creates a subset of the original data.

Im Auge des Sturms: Random Forest

Schritt 2: Wir kreieren ein Decision Tree mit Hilfe unseres „bootstrapped“ Datensatzes. Dabei verwenden wir bei jedem Schritt aber nur zufällig ausgewählte Teile unseres gesamten Datensatzes, z.B. nur 2 Spalten.

Schritt 2a: Um den Root Node zu bestimmen, nehmen wir mal zufällig nur GBC und Blocked Arteries in Betracht:

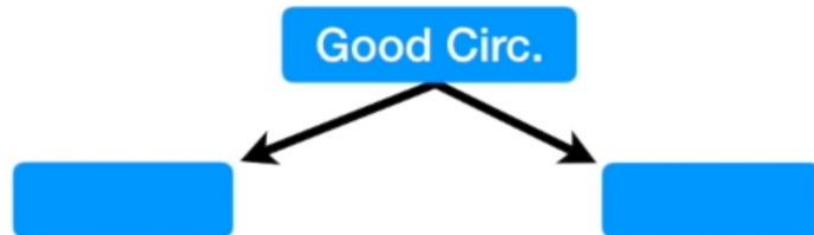


Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
Yes	Yes	Yes	180	Yes
No	No	No	125	No
Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes

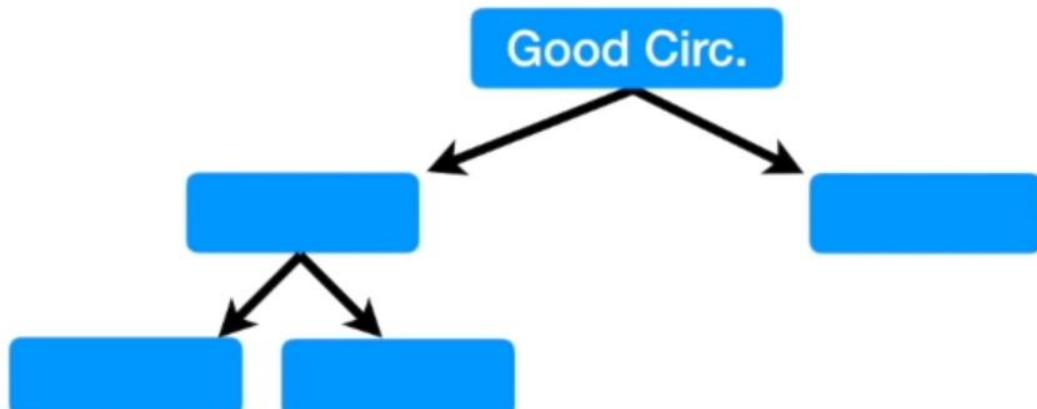
Im Auge des Sturms: Random Forest

Schritt 2: Wir kreieren ein Decision Tree mit Hilfe unsere „bootstrapped“ Datensatzes. Dabei verwenden wir bei jedem Schritt aber nur zufällig ausgewählte Teile unseres gesamten Datensatzes, z.B. nur 2 Spalten.

Schritt 2a: Um den Root Node zu bestimmen, nehmen wir mal zufällig nur GBC und Blocked Arteries in Betracht:



Schritt 2a: Um den nächsten Internal Node zu bestimmen, wählen wir 2 weitere Spalten zufällig aus.:



Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
Yes	Yes	Yes	180	Yes
No	No	No	125	No
Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes

Schritt 1: Wir kreieren einen „**bootstrapped dataset**“, indem wir

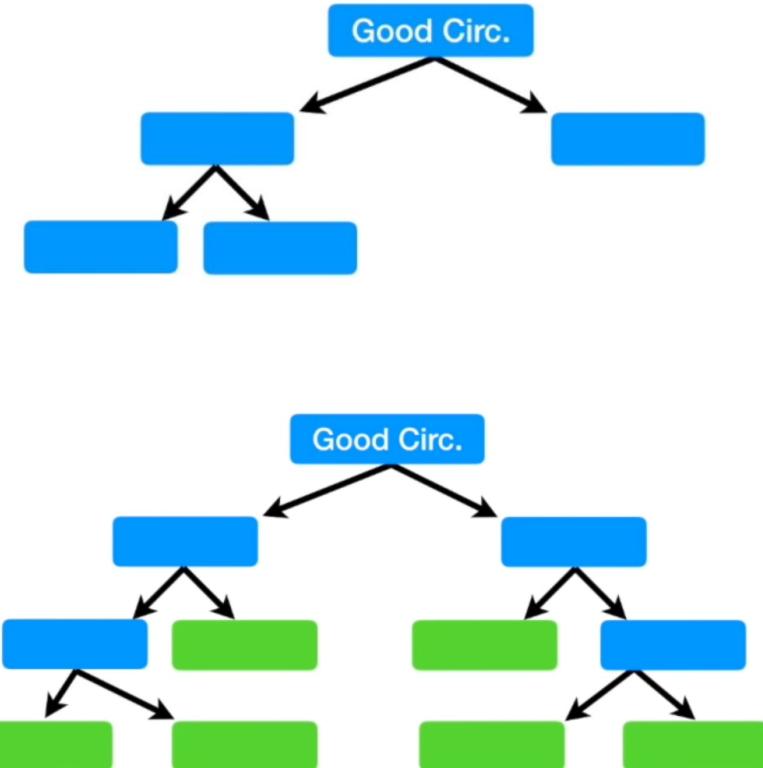
- a) zufällig Zeilen vom originalen Dataset auswählen
- b) dabei auch eine Zeile mehrmals nehmen dürfen -> random!

Schritt 2: Wir kreieren ein Decision Tree mit Hilfe unsere „bootstrapped“ Datensatzes. Dabei verwenden wir bei jedem Schritt aber nur zufällig ausgewählte Teile unseres gesamten Datensatzes, z.B. nur 2 Spalten.

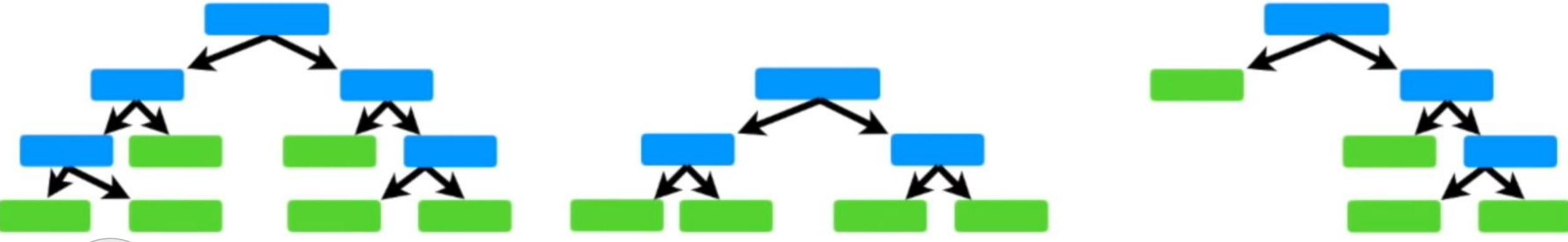
Schritt 2a: Um den Root Node zu bestimmen, nehmen wir mal zufällig nur GBC und Blocked Arteries in Betracht:

Schritt 2b: Um den nächsten Internal Node zu bestimmen, wählen wir 2 weitere Spalten zufällig aus.

Schritt 2c: Dies führen wir fort, bis wir einen vollständigen Decision Tree erhalten.

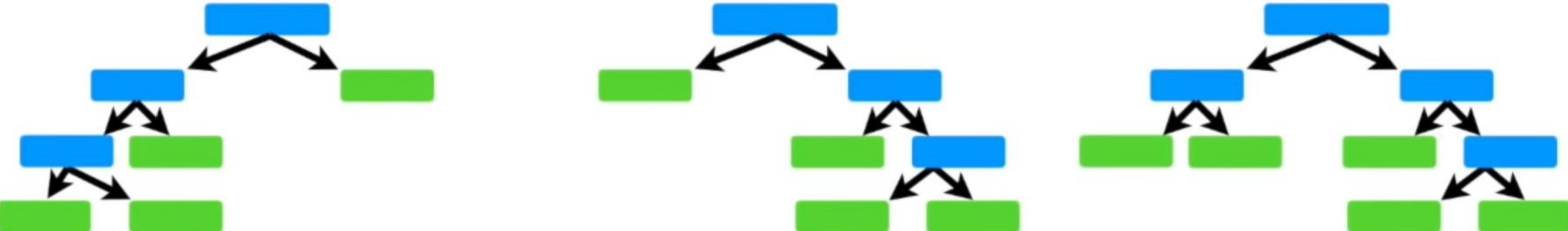


Diesen Prozess gehen wir also 100 male durch und kreieren hunderte verschiedenster Decision Trees, also einen Random Forest!

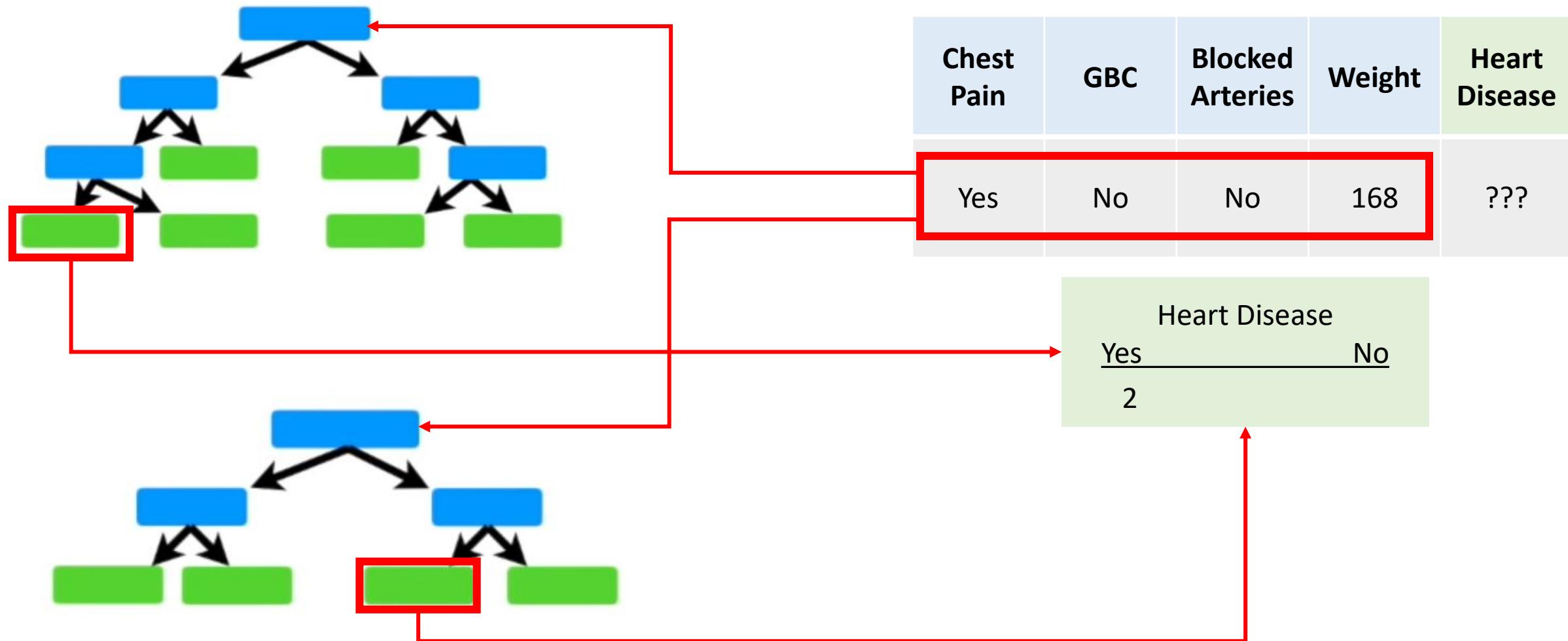


ACHTUNG
FLACH

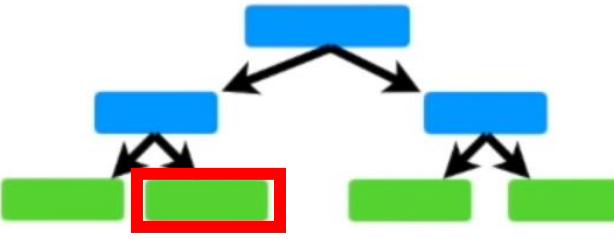
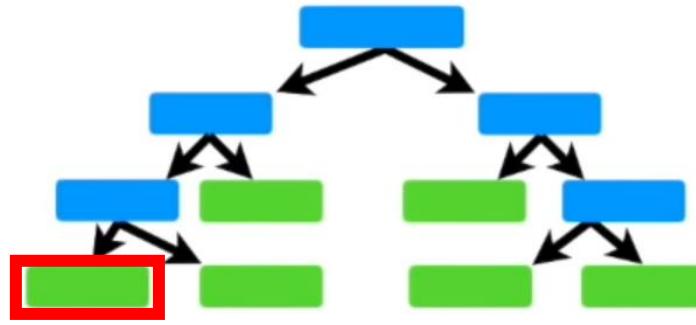
Aber was machen jetzt, wo wir „den Wald vor lauter Bäumen nicht sehen“?



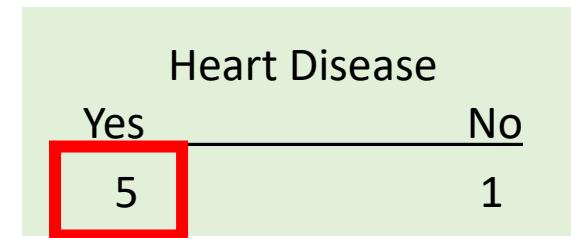
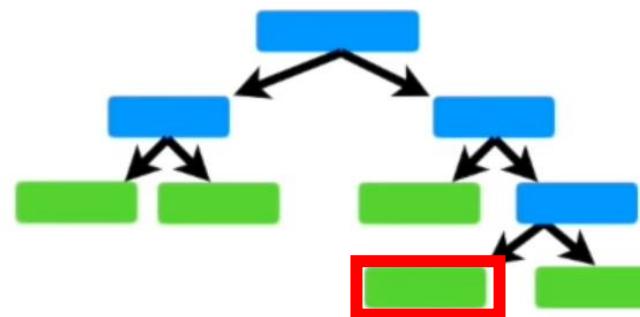
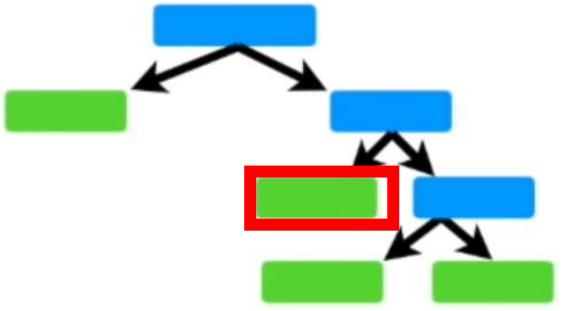
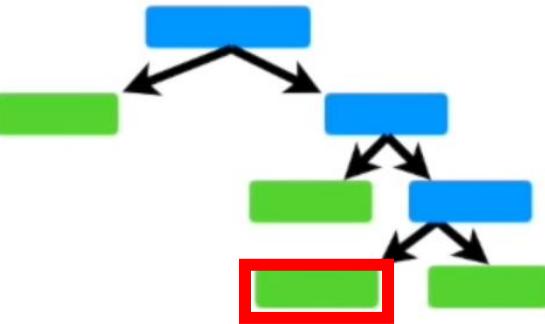
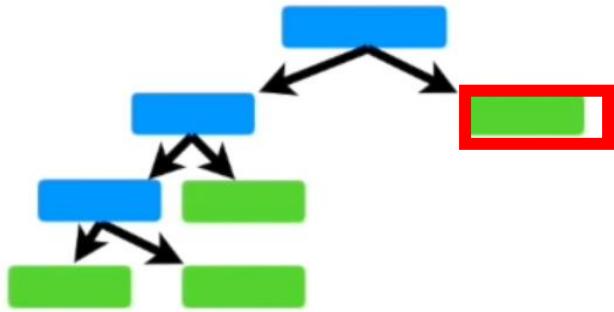
Im Auge des Sturms: Random Forest



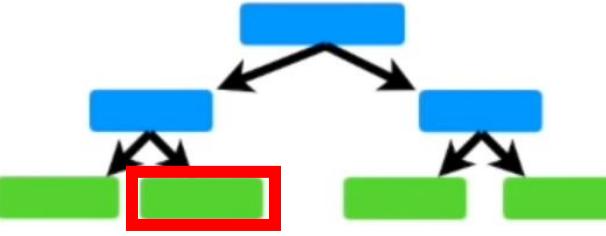
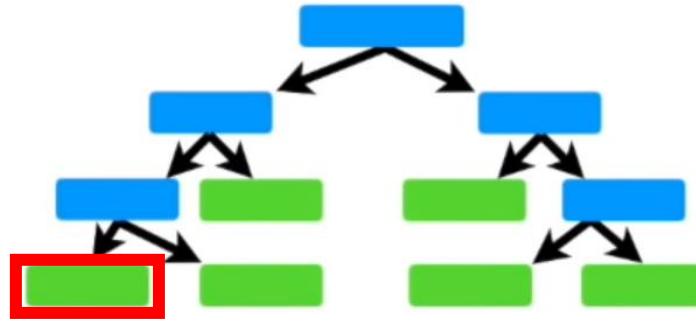
Im Auge des Sturms: Random Forest



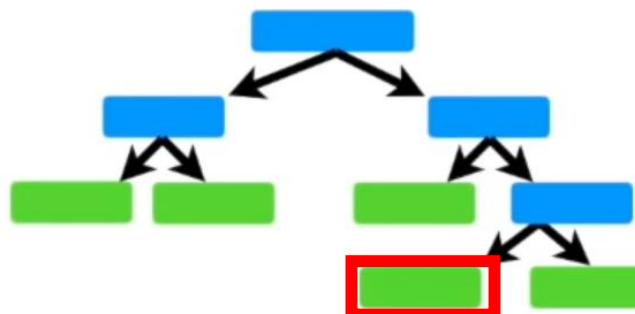
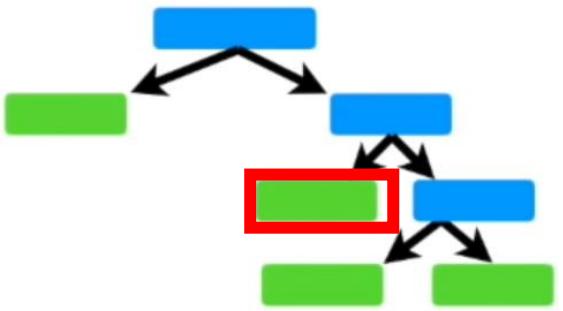
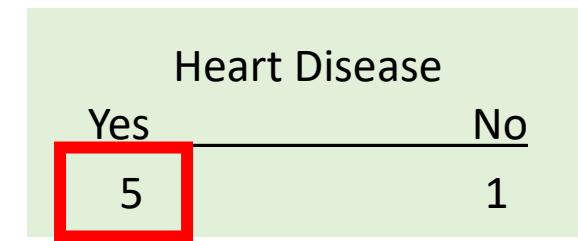
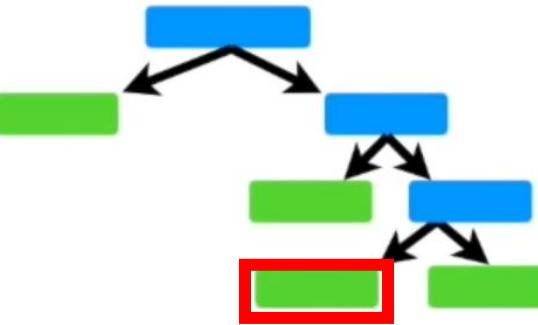
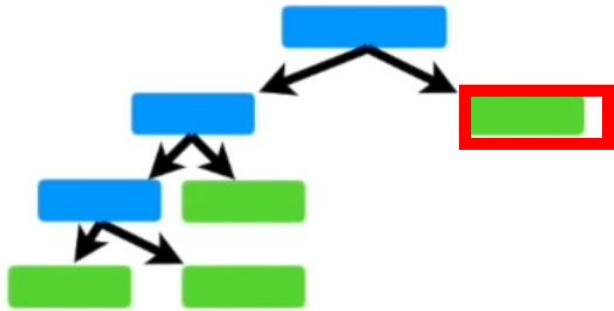
Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
Yes	No	No	168	???



Wie geht Random Forest nun bei einem neuen Datensatz vor?



Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
Yes	No	No	168	YES



Bootstrapping
+
Aggregating
=

Bagging

Im Auge des Sturms: Random Forest

Woher weiß ich denn, dass mein Random Forest auch wirklich gut ist?

Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease		Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No		Yes	Yes	Yes	180	Yes
Yes	Yes	Yes	180	Yes		No	No	No	125	No
Yes	Yes	No	210	No		Yes	No	Yes	167	Yes
Yes	No	Yes	167	Yes		Yes	No	Yes	167	Yes

```

graph LR
    R1[Row 1] --> R5[Row 5]
    R2[Row 2] --> R5
    R3[Row 3] --> R5
    R4[Row 4] --> R5
  
```

Im Auge des Sturms: Random Forest

Woher weiß ich denn, dass mein Random Forest auch wirklich gut ist?

Wir erstellen ein „Out-Of-Bag-Dataset“!

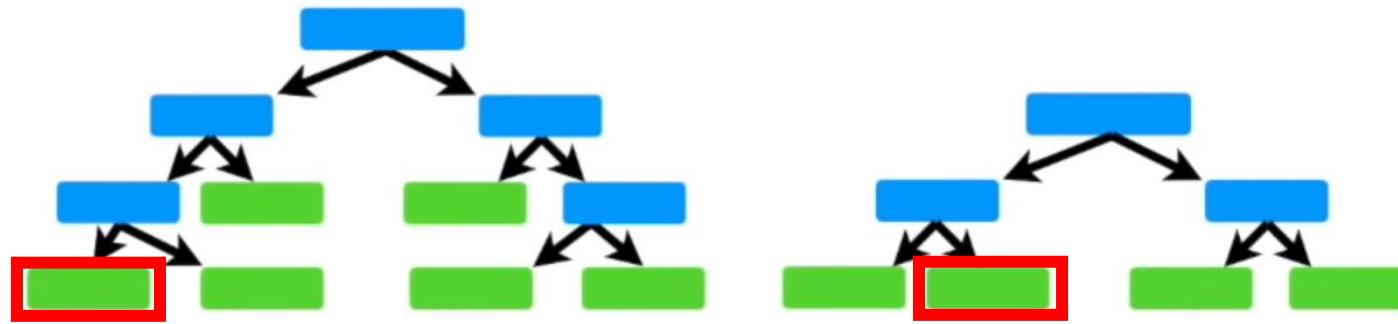
Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
No	No	No	125	No
Yes	Yes	Yes	180	Yes
Yes	Yes	No	210	No
Yes	No	Yes	167	Yes

Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
Yes	Yes	No	210	No
...



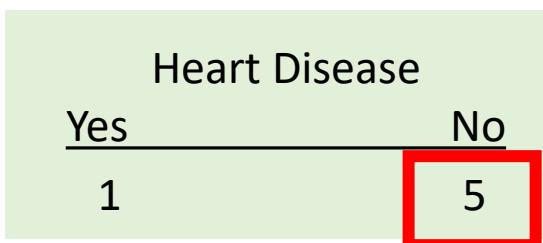
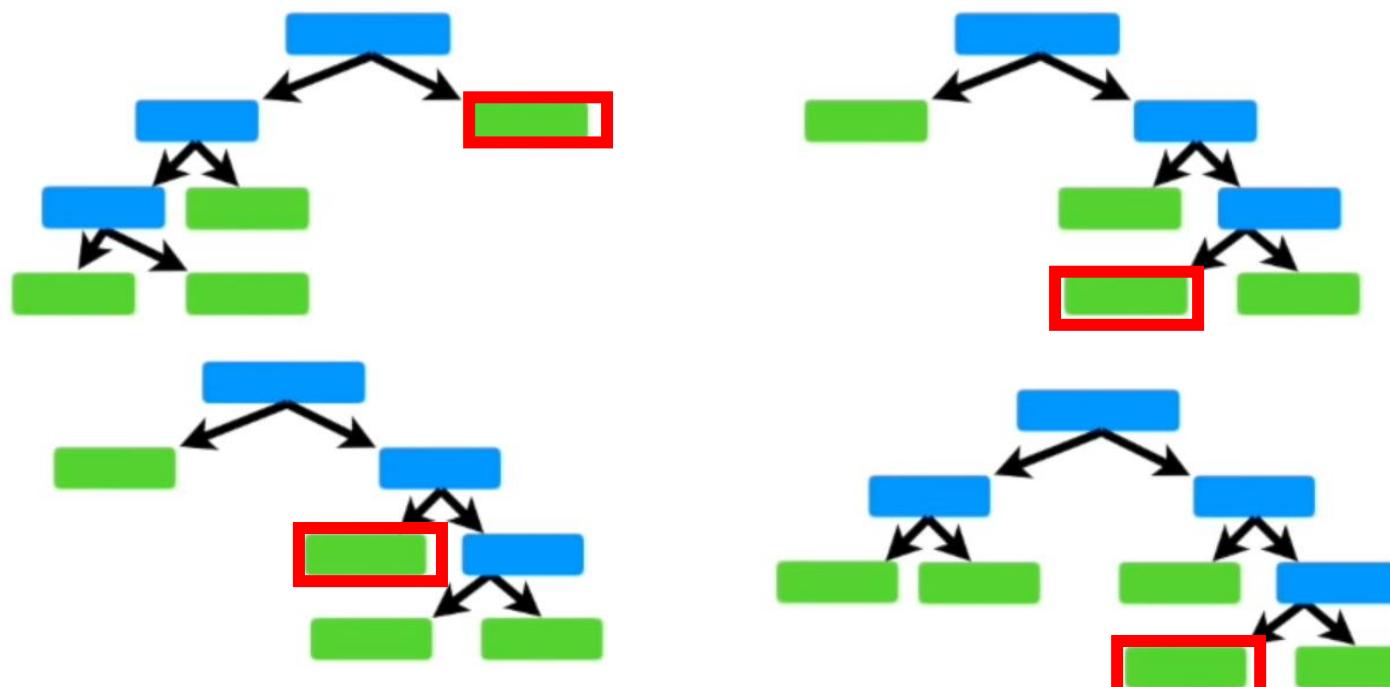
Im Auge des Sturms: Random Forest

Woher weiß ich denn, dass mein Random Forest auch wirklich gut ist?



Wir erstellen ein „Out-Of-Bag-Dataset“!

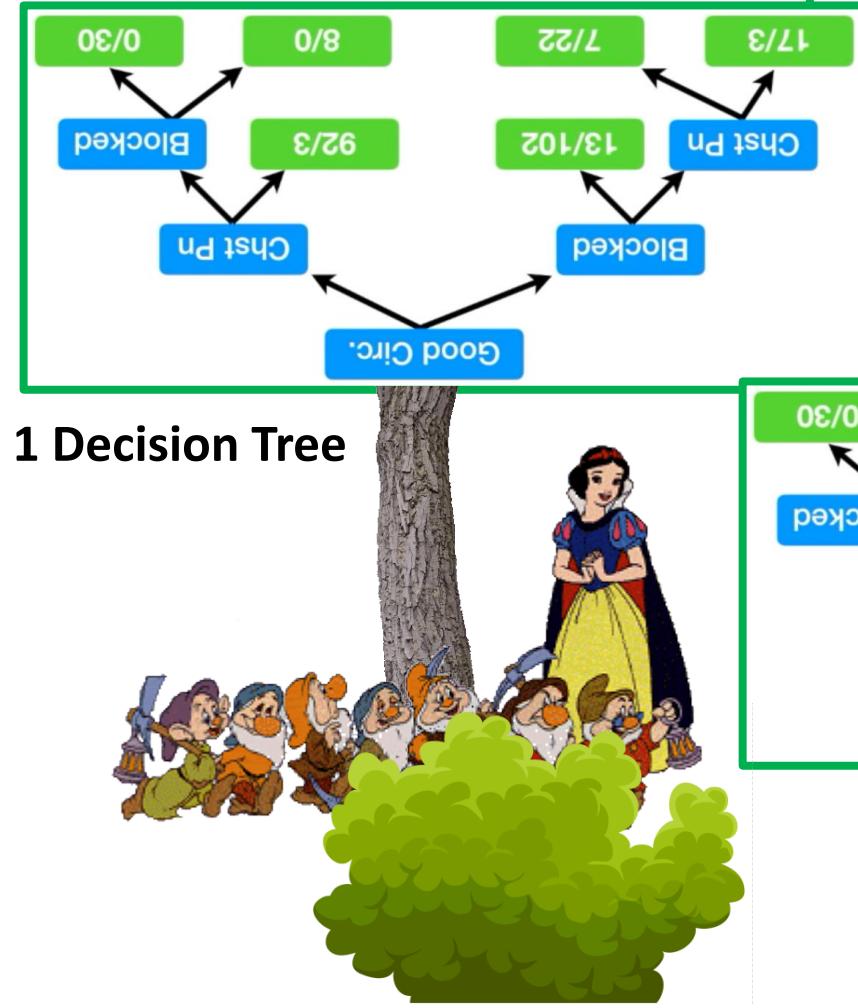
Chest Pain	GBC	Blocked Arteries	Weight	Heart Disease
Yes	Yes	No	210	No



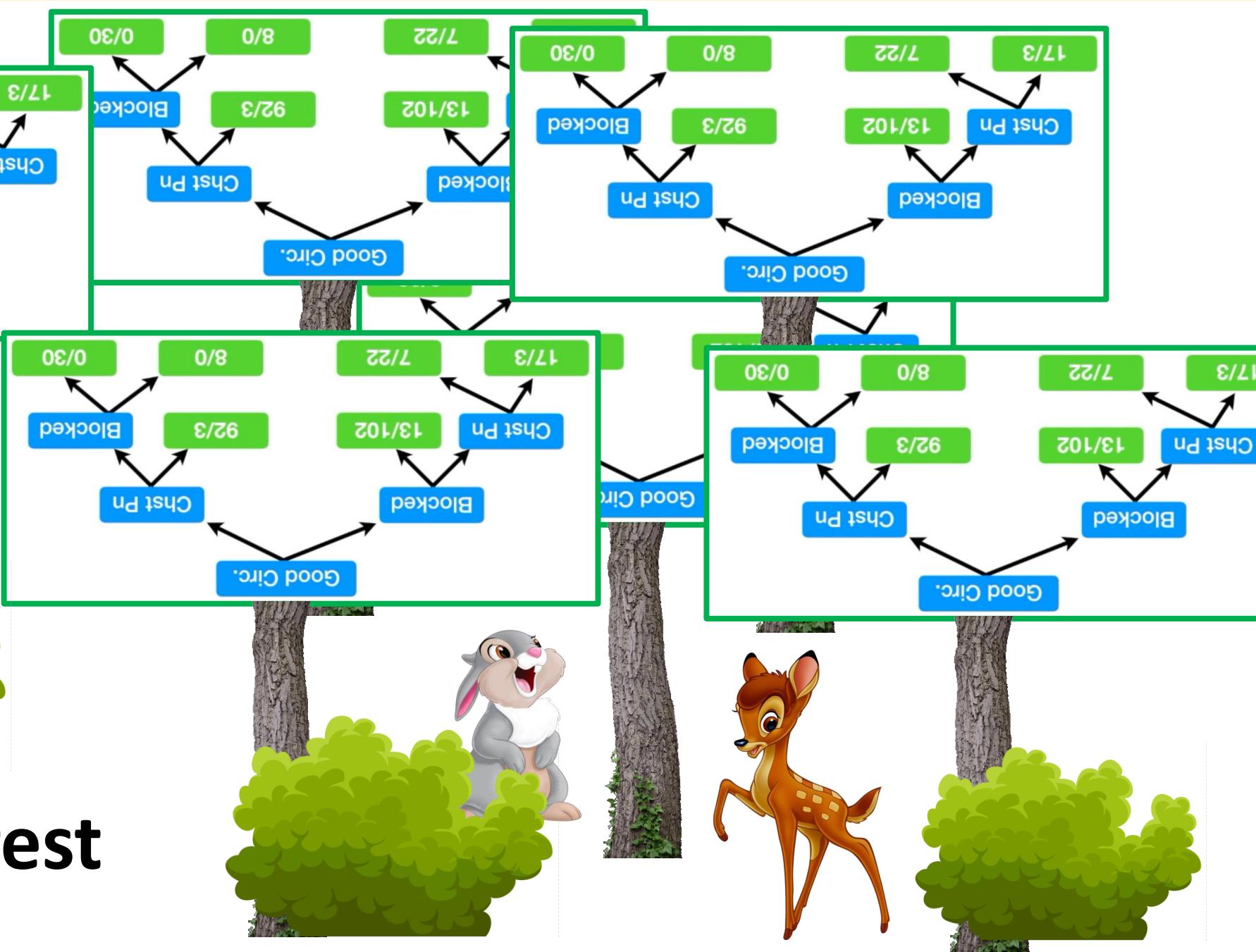
Die Genauigkeit unseres Random Forest kann mithilfe der korrekt klassifizierten „Out-Of-Bag“-Patienten bestimmt werden!

Falls ungenau: Neuer Random-Forest-Lauf!

Im Auge des Sturms: Random Forest



1 Random Forest



Classification-tree in scikit-learn

```
# Import DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
# Import train_test_split
from sklearn.model_selection import train_test_split
# Import accuracy_score
from sklearn.metrics import accuracy_score
# Split the dataset into 80% train, 20% test
X_train, X_test, y_train, y_test= train_test_split(X, y,
                                                    test_size=0.2,
                                                    stratify=y,
                                                    random_state=1)
# Instantiate dt
dt = DecisionTreeClassifier(max_depth=2, random_state=1)
```

```
# Fit dt to the training set
dt.fit(X_train,y_train)

# Predict the test set labels
y_pred = dt.predict(X_test)
# Evaluate the test-set accuracy
accuracy_score(y_test, y_pred)
```

0.90350877192982459

Random Forests: Classification & Regression

Classification:

- Aggregates predictions by majority voting
- `RandomForestClassifier` in scikit-learn

Regression:

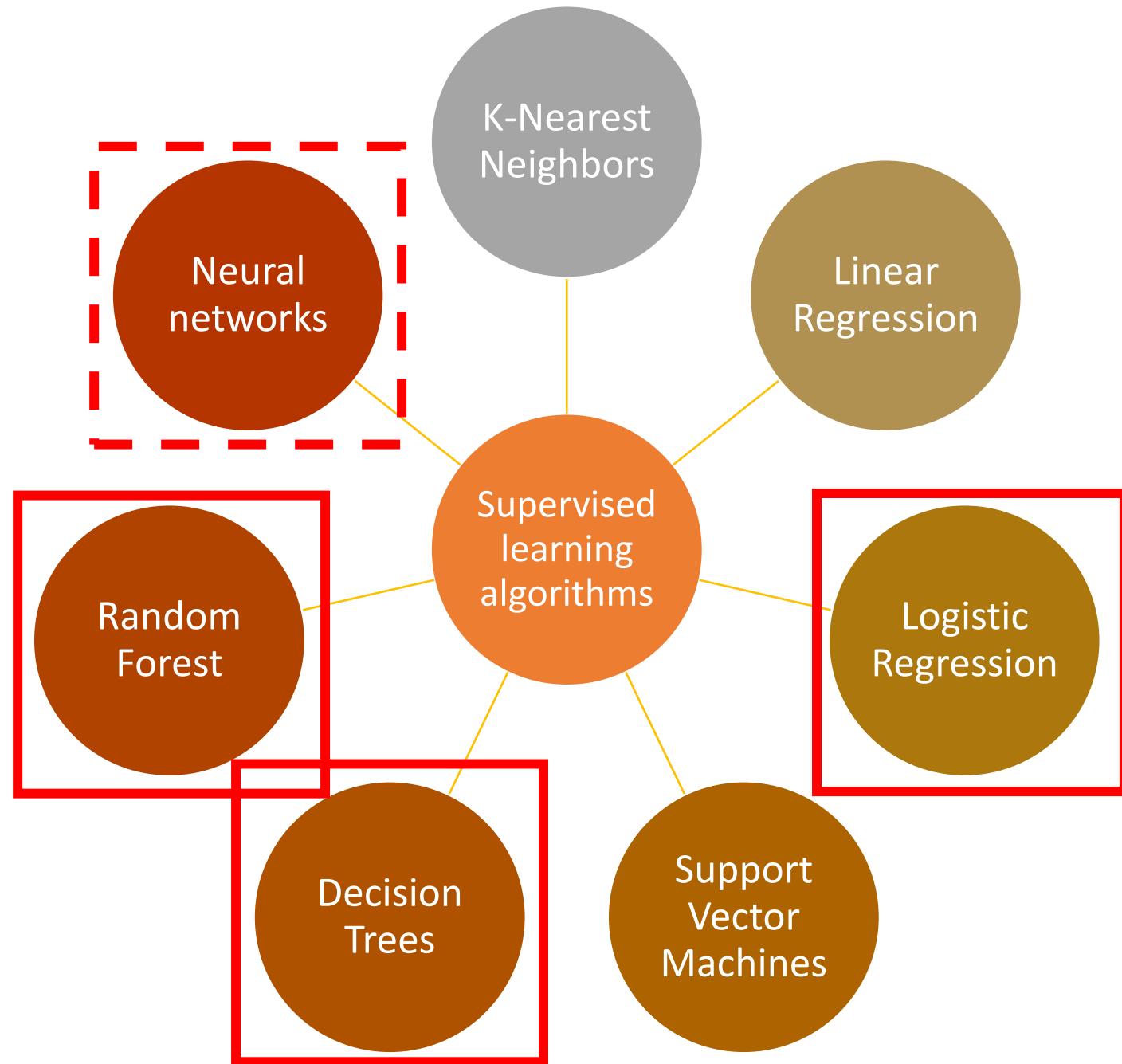
- Aggregates predictions through averaging
- `RandomForestRegressor` in scikit-learn



Auf zu Colab!

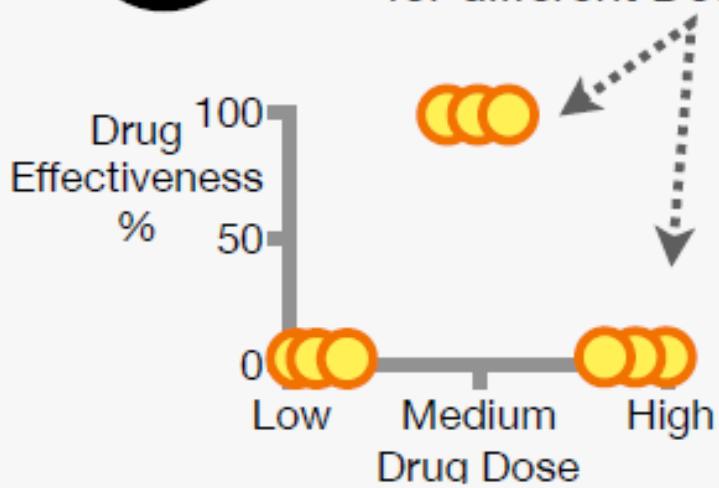
Supervised Learning with scikit-learn

Die Qual der Wahl...

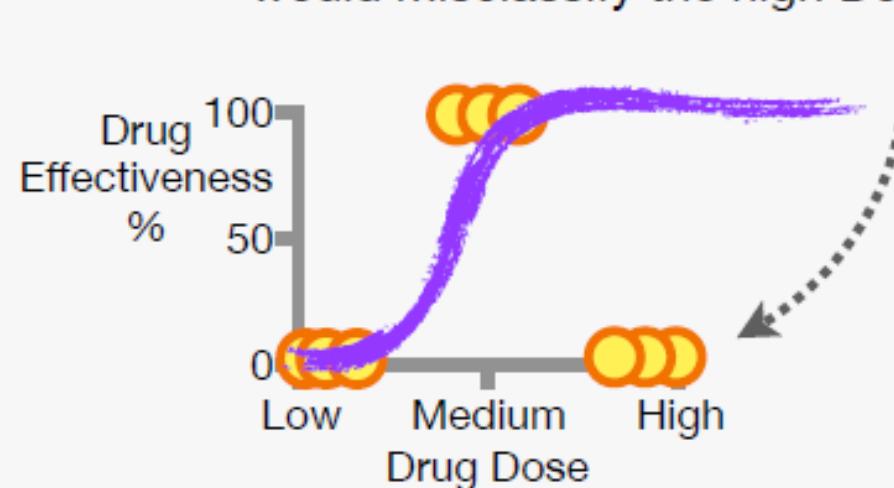


1

The Problem: We have data that show the Effectiveness of a drug for different Doses....



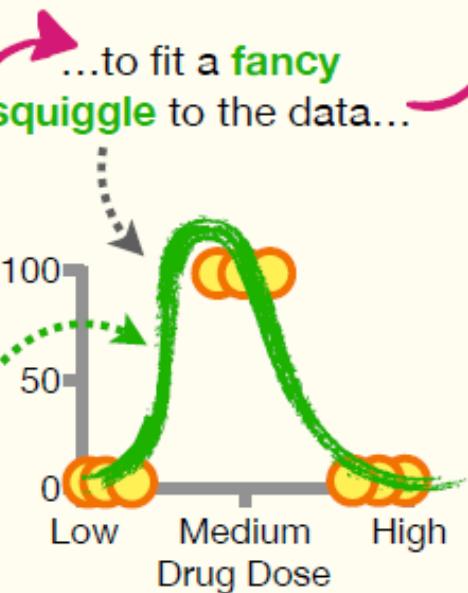
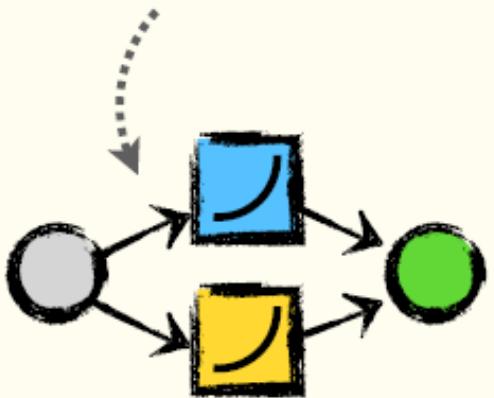
...and the **s-shaped squiggle** that **Logistic Regression** uses would not make a good fit. In this example, it would misclassify the high Doses.



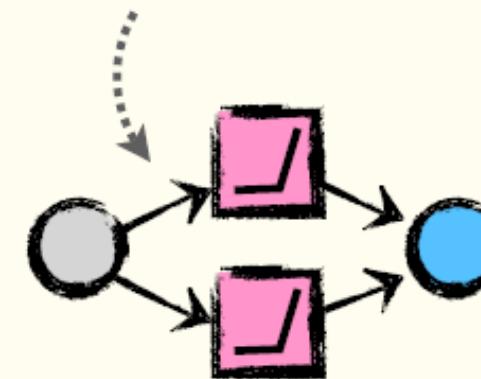
2

A Solution: Although they sound super intimidating, all **Neural Networks** do is fit **fancy squiggles** or **bent shapes** to data. And like **Decision Trees** and **SVMs**, **Neural Networks** do fine with any relationship among the variables.

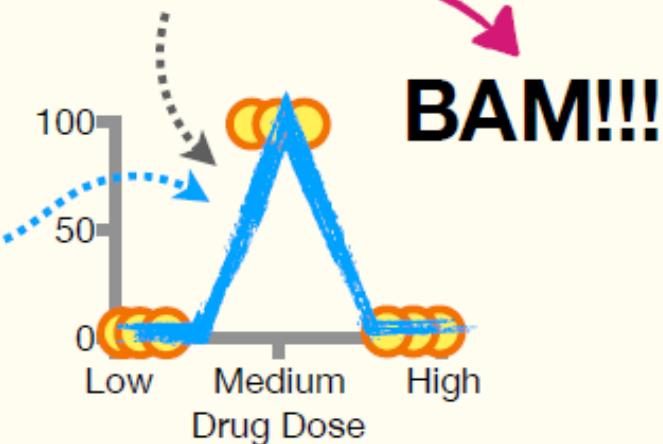
For example, we could get this **Neural Network**...



...or we could get this **Neural Network**...

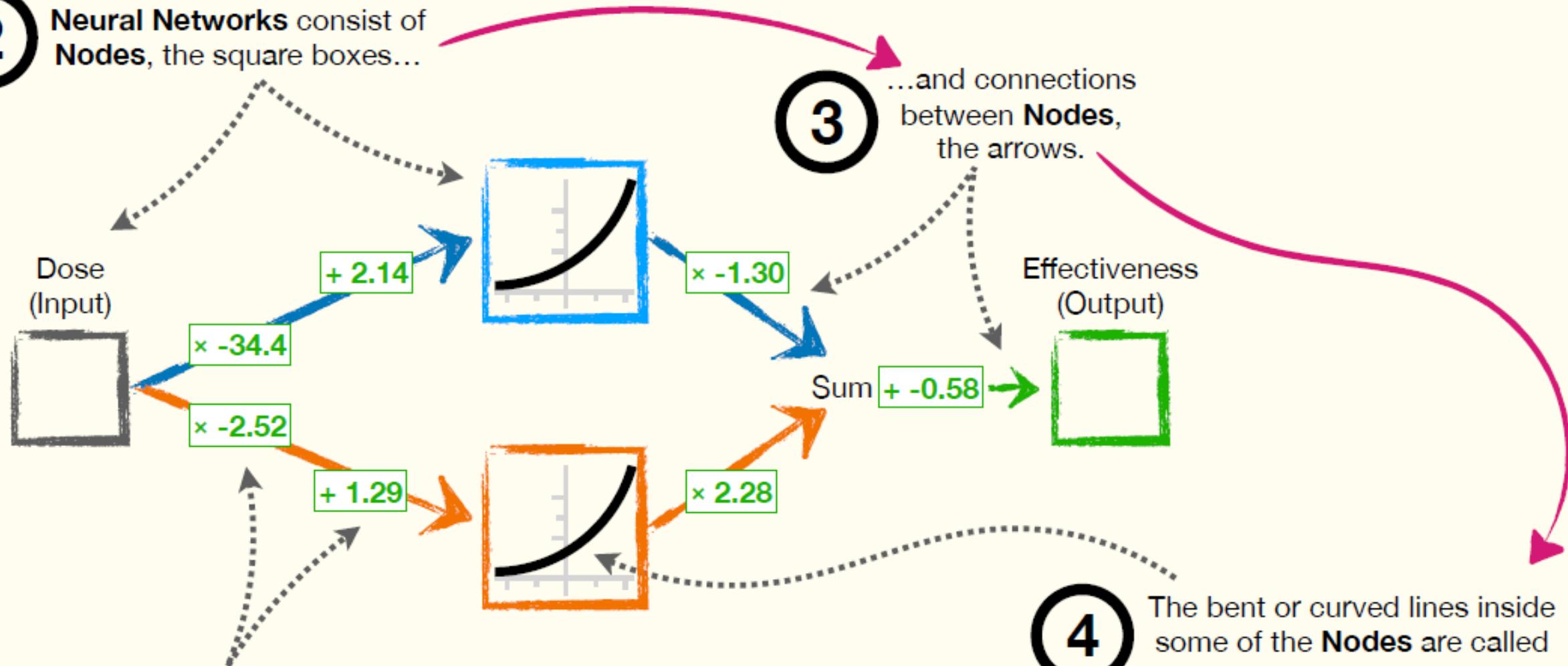


...to fit a **bent shape** to the data.



2

Neural Networks consist of Nodes, the square boxes...



3

...and connections between Nodes, the arrows.

4

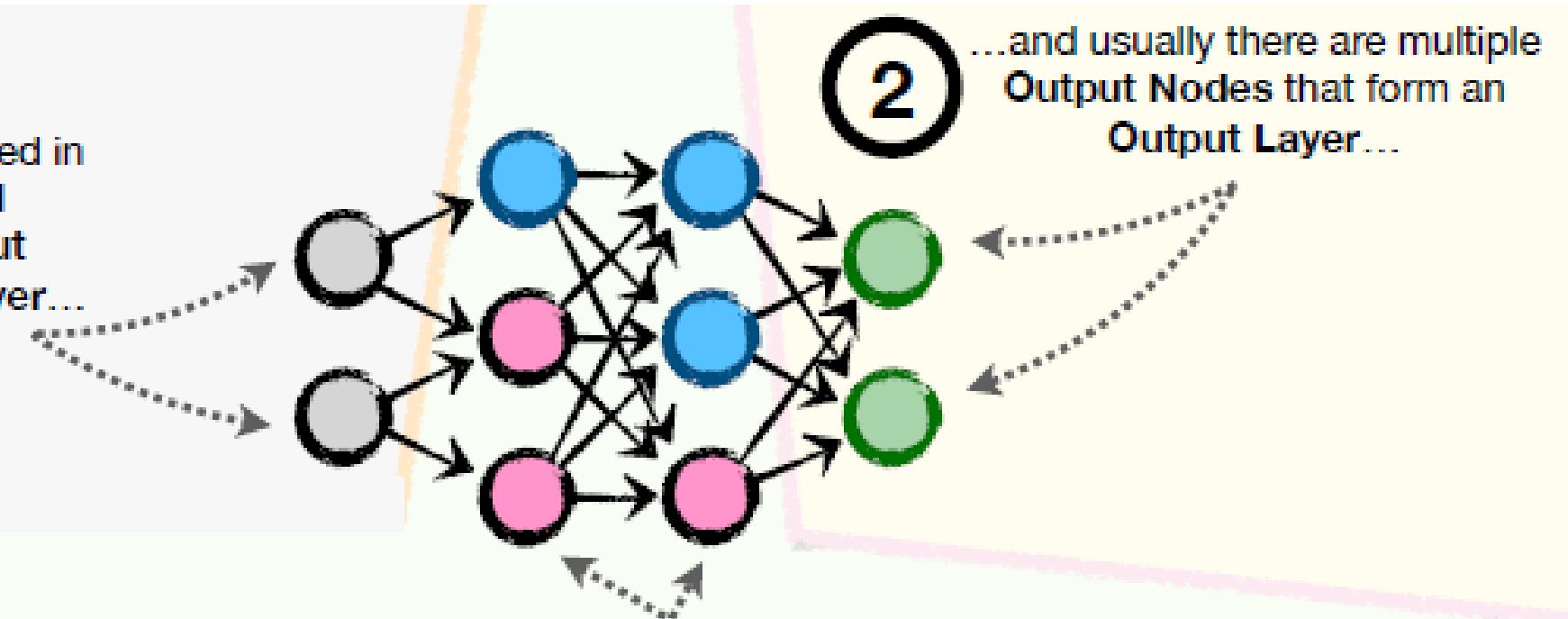
The bent or curved lines inside some of the Nodes are called Activation Functions, and they make Neural Networks flexible and able to fit just about any data.

5

The numbers along the connections represent parameter values that were estimated when this Neural Network was fit to data using a process called Backpropagation. In this chapter, we'll see exactly what the parameters do and how they're estimated, step-by-step.

1

Neural Networks are organized in **Layers**. Usually, a Neural Network has multiple **Input Nodes** that form an **Input Layer**...



2

...and usually there are multiple **Output Nodes** that form an **Output Layer**...

3

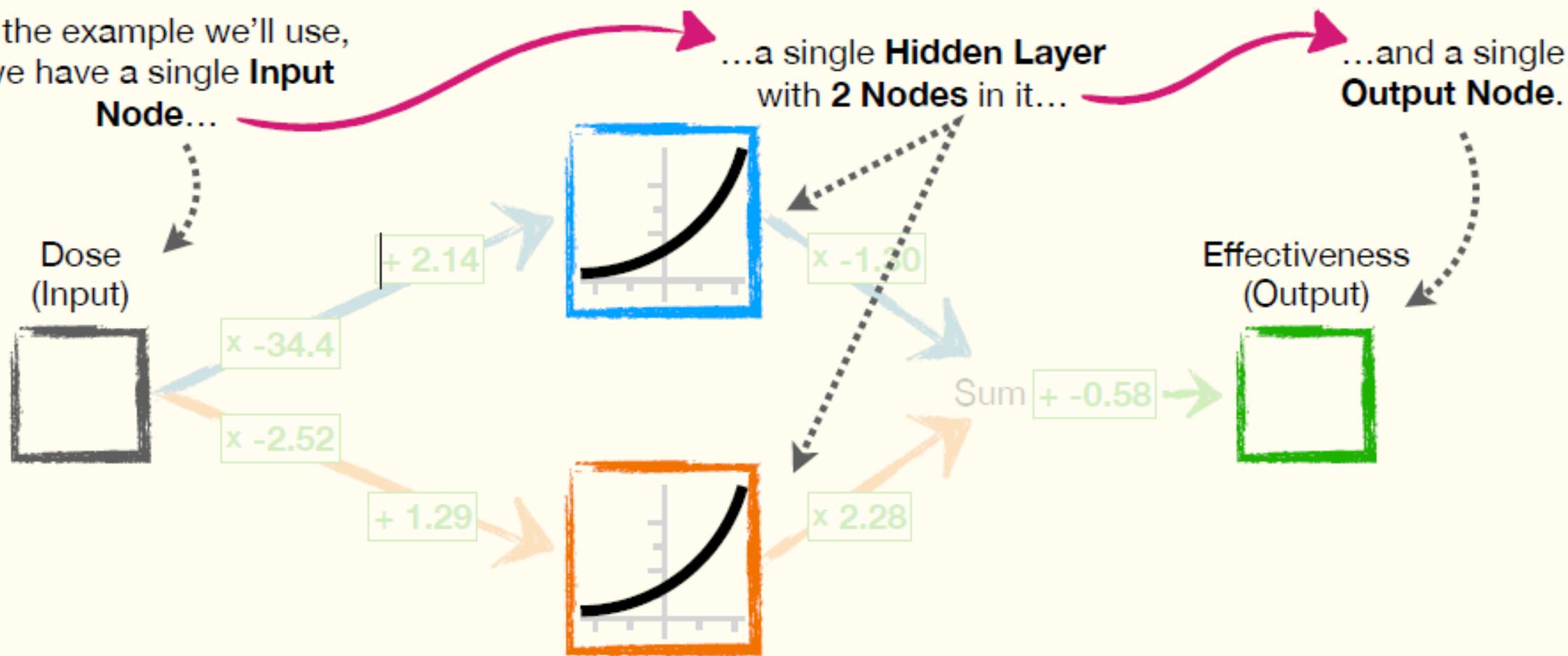
...and the **Layers of Nodes** between the **Input** and **Output Layers** are called **Hidden Layers**. Part of the art of **Neural Networks** is deciding how many **Hidden Layers** to use and how many **Nodes** should be in each one. Generally speaking, the more **Layers** and **Nodes**, the more complicated the shape that can be fit to the data.

4

In the example we'll use,
we have a single **Input
Node**...

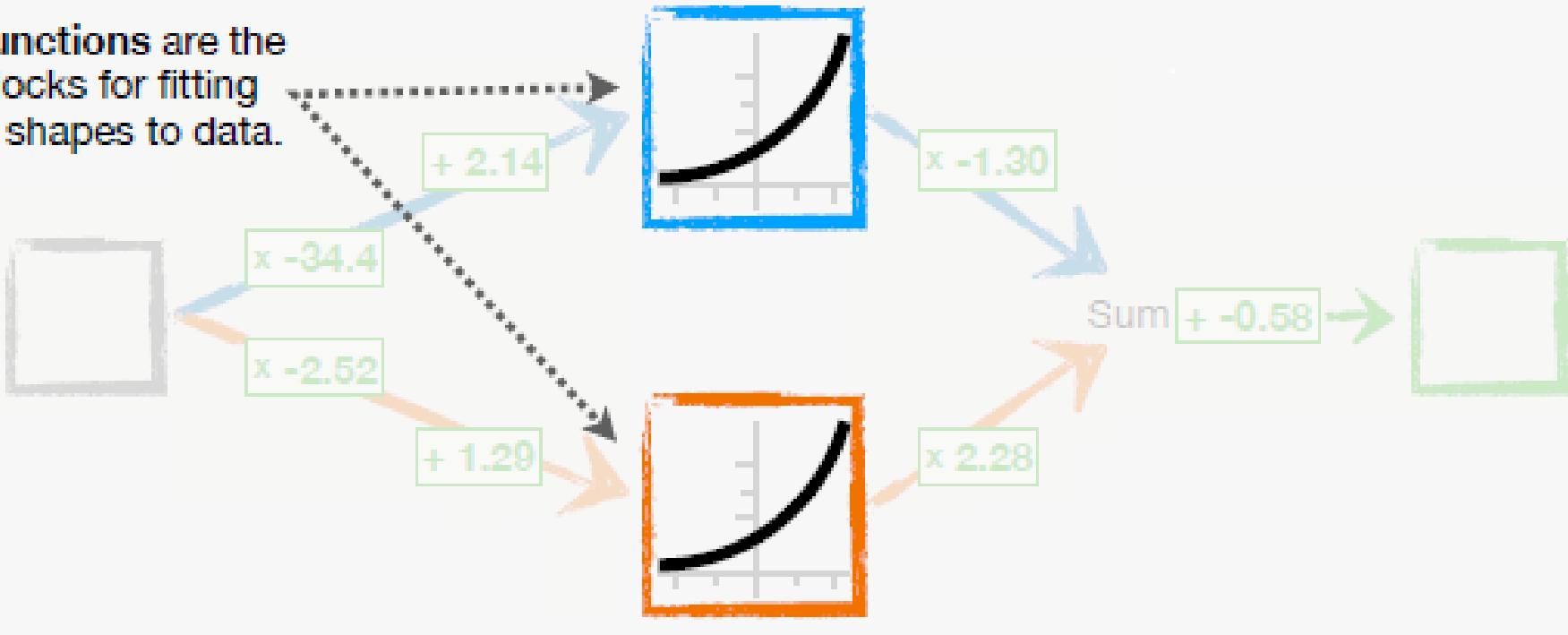
...a single **Hidden Layer**
with **2 Nodes** in it...

...and a single
Output Node.



1

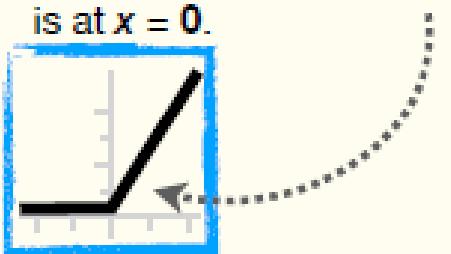
The Activation Functions are the basic building blocks for fitting squiggles or bent shapes to data.



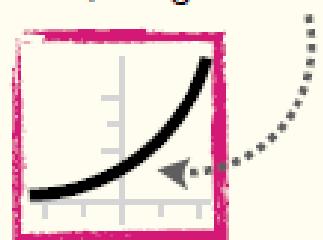
2

There are lots of different Activation Functions. Here are three that are commonly used:

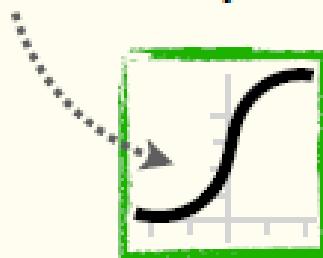
ReLU, which is short for Rectified Linear Unit and sounds like the name of a robot, is probably the most commonly used Activation Function with large Neural Networks. It's a Bent Line, and the bend is at $x = 0$.



SoftPlus, which sounds like a brand of toilet paper, is a modified form of the ReLU Activation Function. The big difference is that instead of the line being bent at 0, we get a nice Curve.

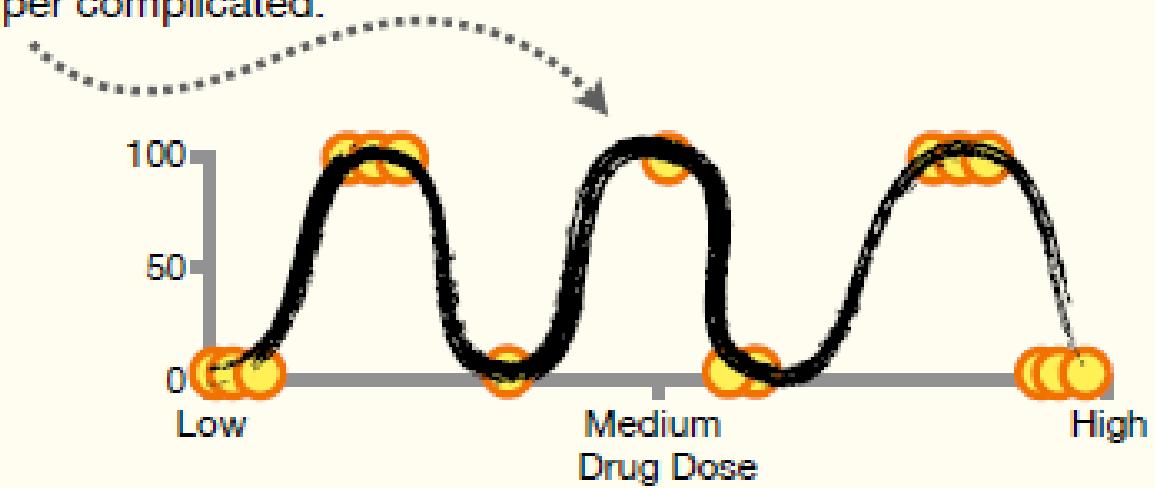
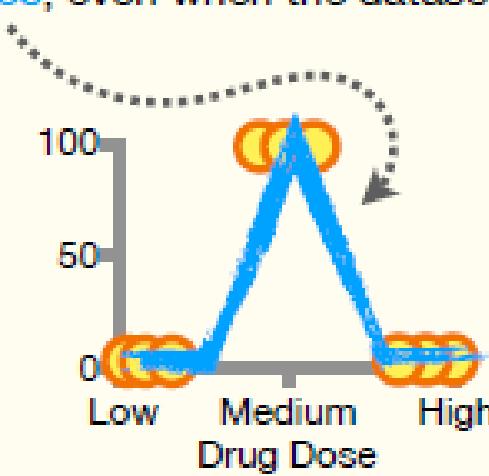
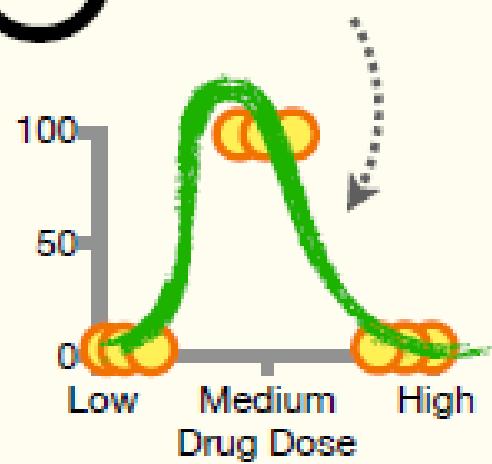


Lastly, the Sigmoid Activation Function is an s-shaped squiggle that's frequently used when people teach Neural Networks but is rarely used in practice.



1

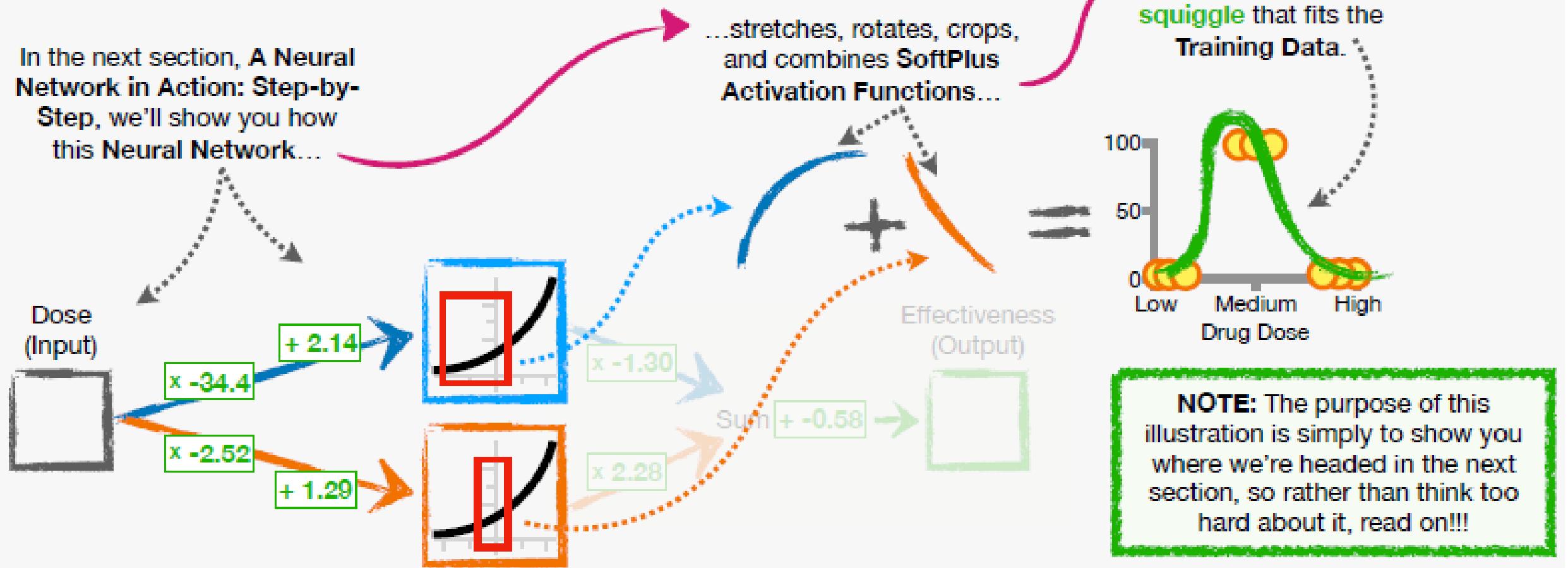
The Problem: We need to create new, exciting shapes that can fit any dataset with a **squiggle** or **bent lines**, even when the dataset is super complicated.



2

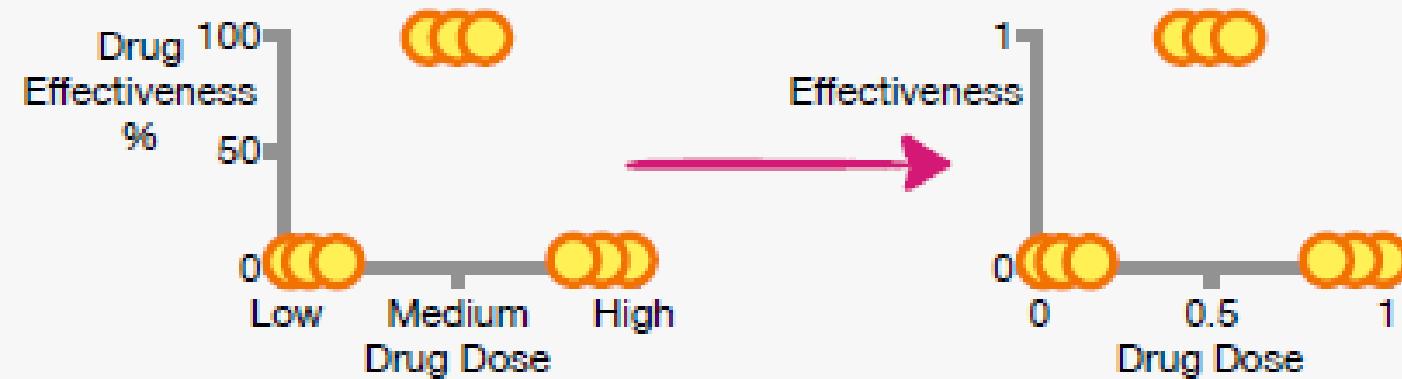
The Solution: Neural Networks stretch, rotate, crop, and combine Activation Functions to create new, exciting shapes that can fit anything!!!

In the next section, A Neural Network in Action: Step-by-Step, we'll show you how this Neural Network...

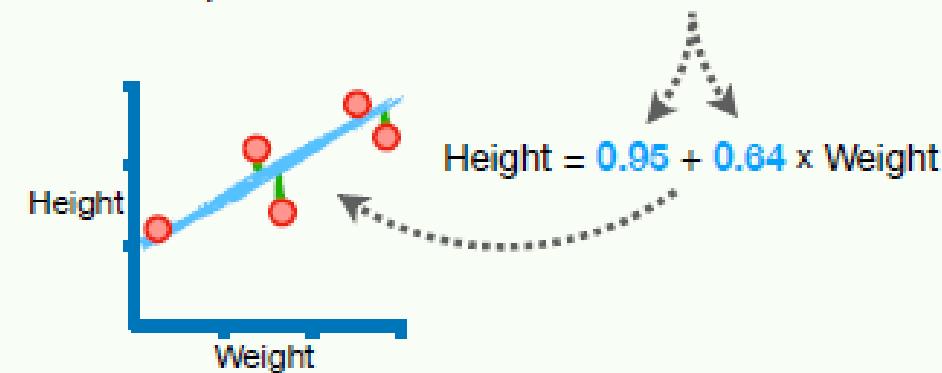


2

NOTE: To keep the math in this section simple, let's scale both the x- and y-axes so that they go from 0, for *low* values, to 1, for *high* values.

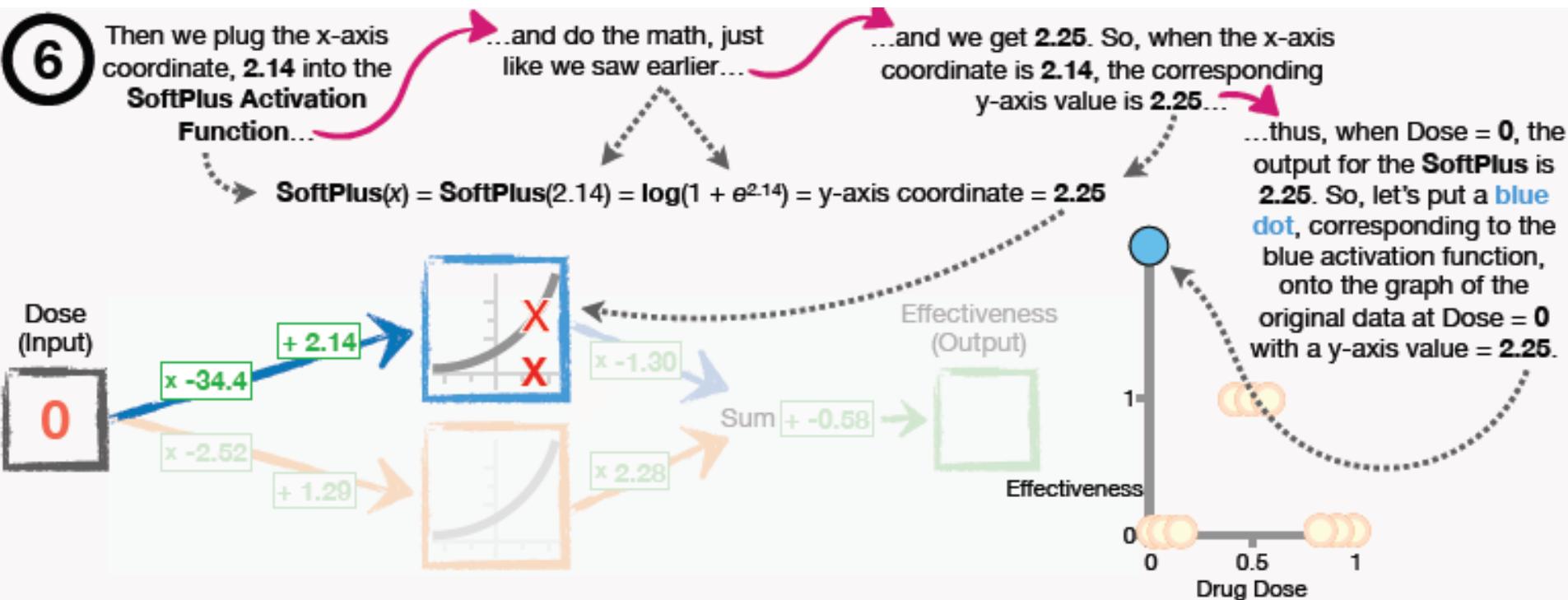
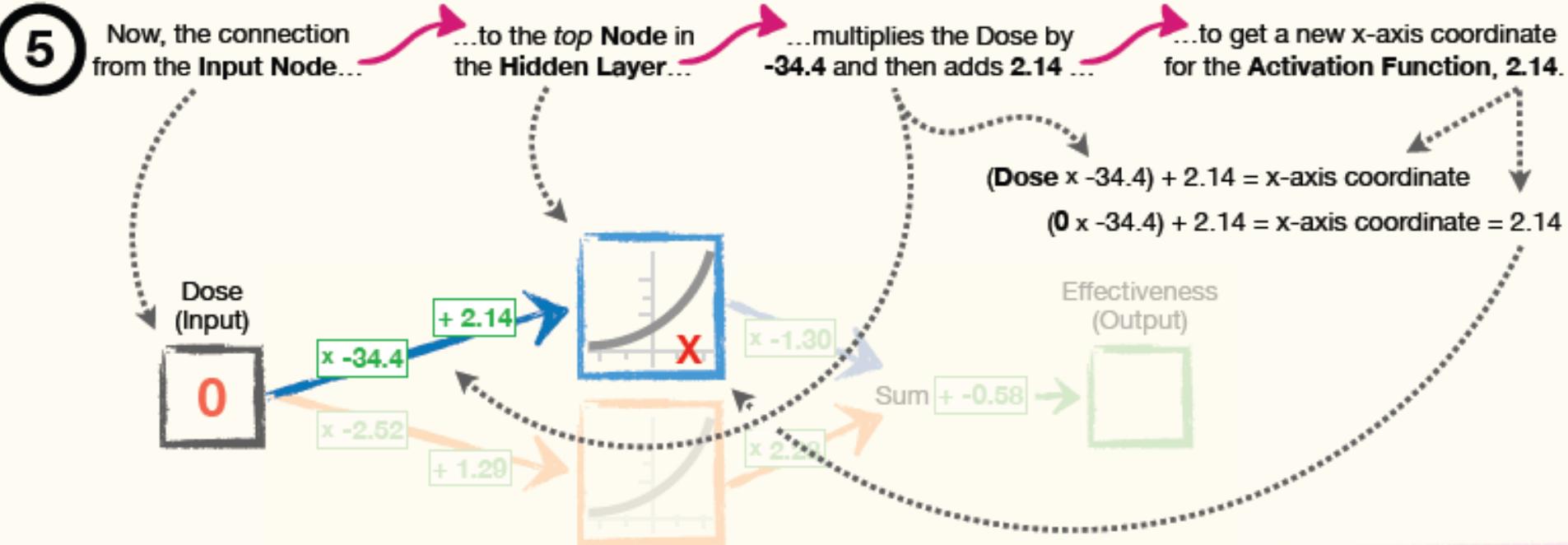
**3**

ALSO NOTE: These numbers are parameters that are estimated using a method called **Backpropagation**, and we'll talk about how that works, step-by-step, later. For now, just assume that these values have already been optimized, much like the **slope** and **intercept** are optimized when we fit a line to data.

**4**

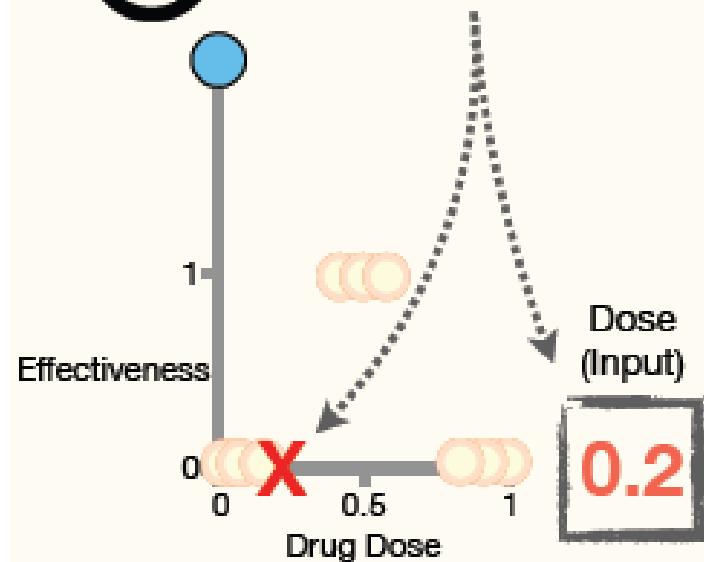
The first thing we do is plug the lowest Dose, 0, into the **Neural Network**.





7

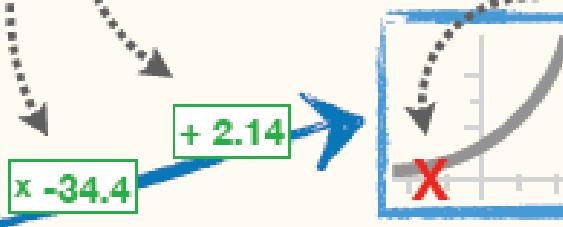
Now let's increase the Dose to 0.2...



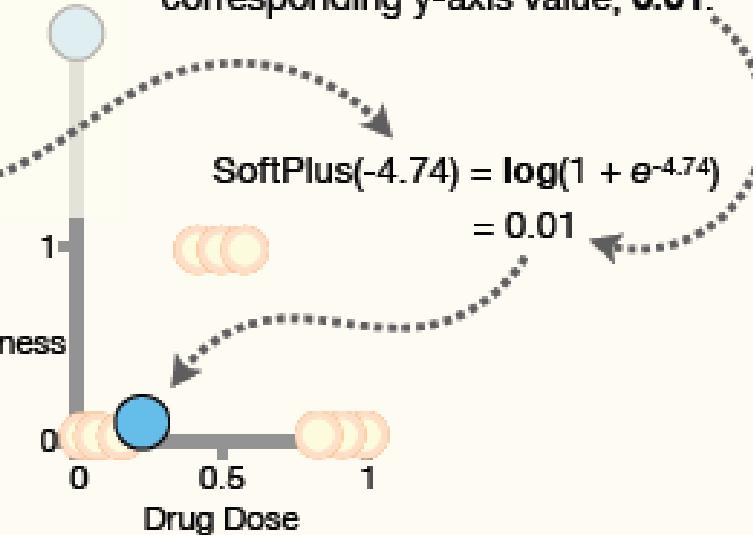
...and calculate the new x-axis coordinate by multiplying the Dose by -34.4 and adding 2.14 to get -4.74...

$$(Dose \times -34.4) + 2.14$$

$$(0.2 \times -34.4) + 2.14 = -4.74$$

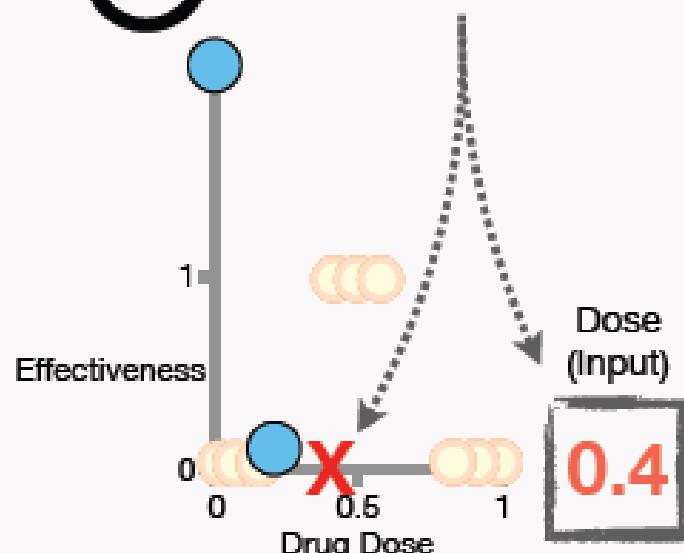


...and plug -4.74 into the SoftPlus Activation Function to get the corresponding y-axis value, 0.01.



8

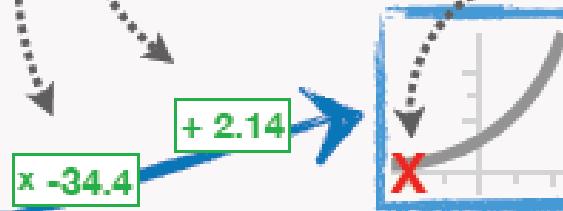
Now let's increase the Dose to 0.4...



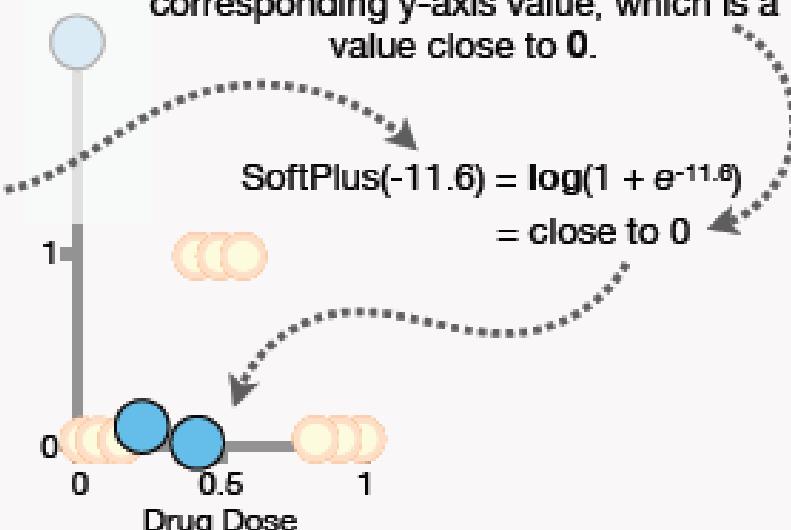
...and calculate the new x-axis coordinate by multiplying the Dose by -34.4 and adding 2.14 to get -11.6...

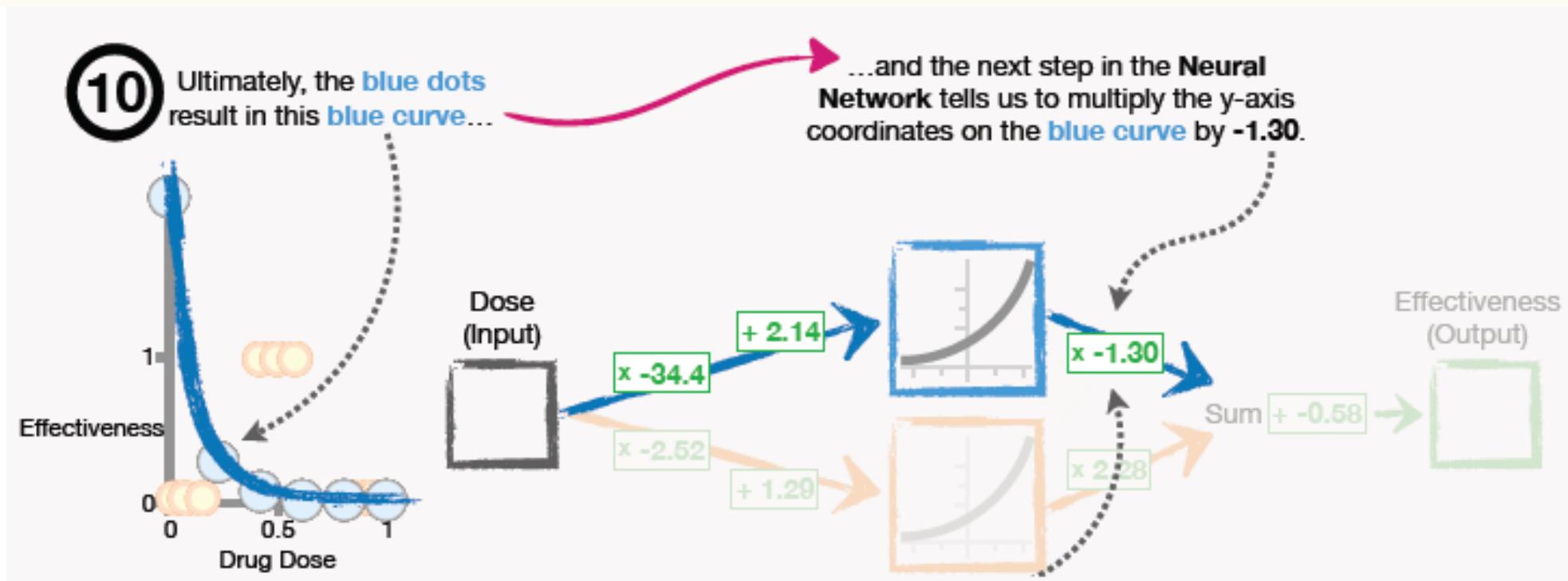
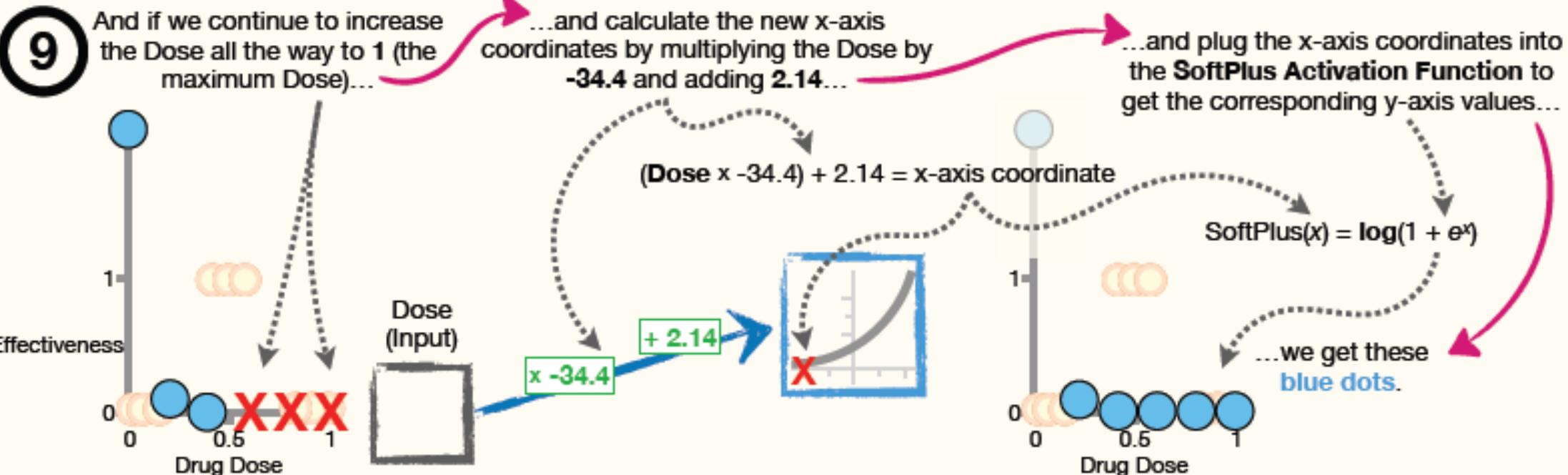
$$(Dose \times -34.4) + 2.14$$

$$(0.4 \times -34.4) + 2.14 = -11.6$$



...and plug -11.6 into the SoftPlus Activation Function to get the corresponding y-axis value, which is a value close to 0.

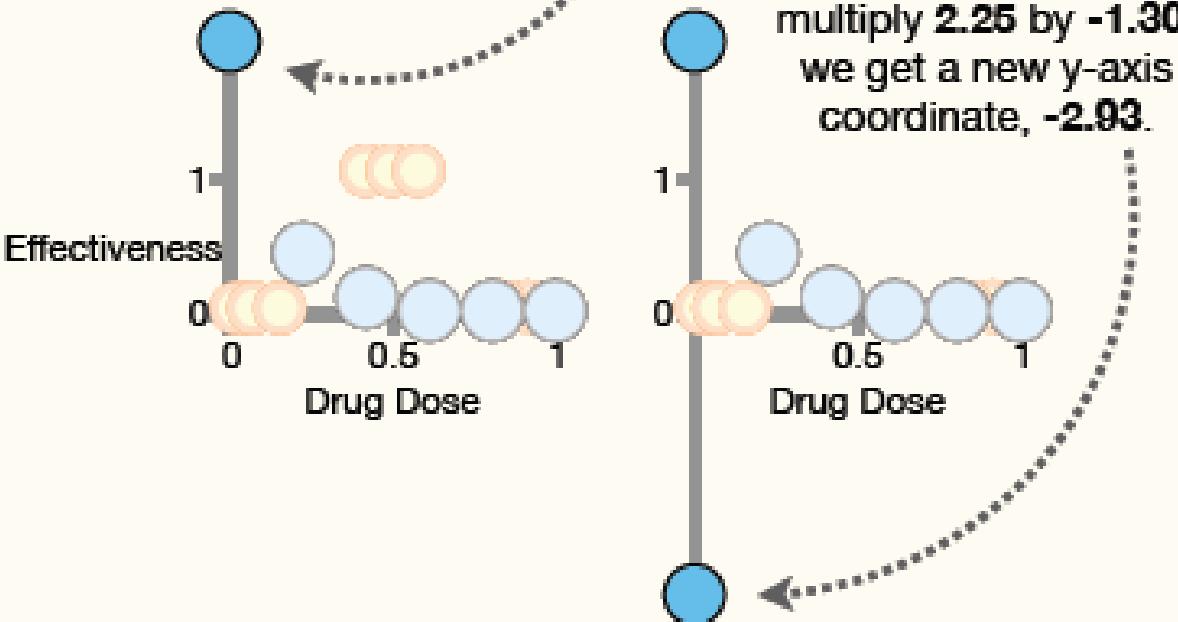




11

For example, when Dose = 0, the current y-axis coordinate on the **blue curve** is **2.25**...

...and when we multiply **2.25** by **-1.30**, we get a new y-axis coordinate, **-2.93**.



12

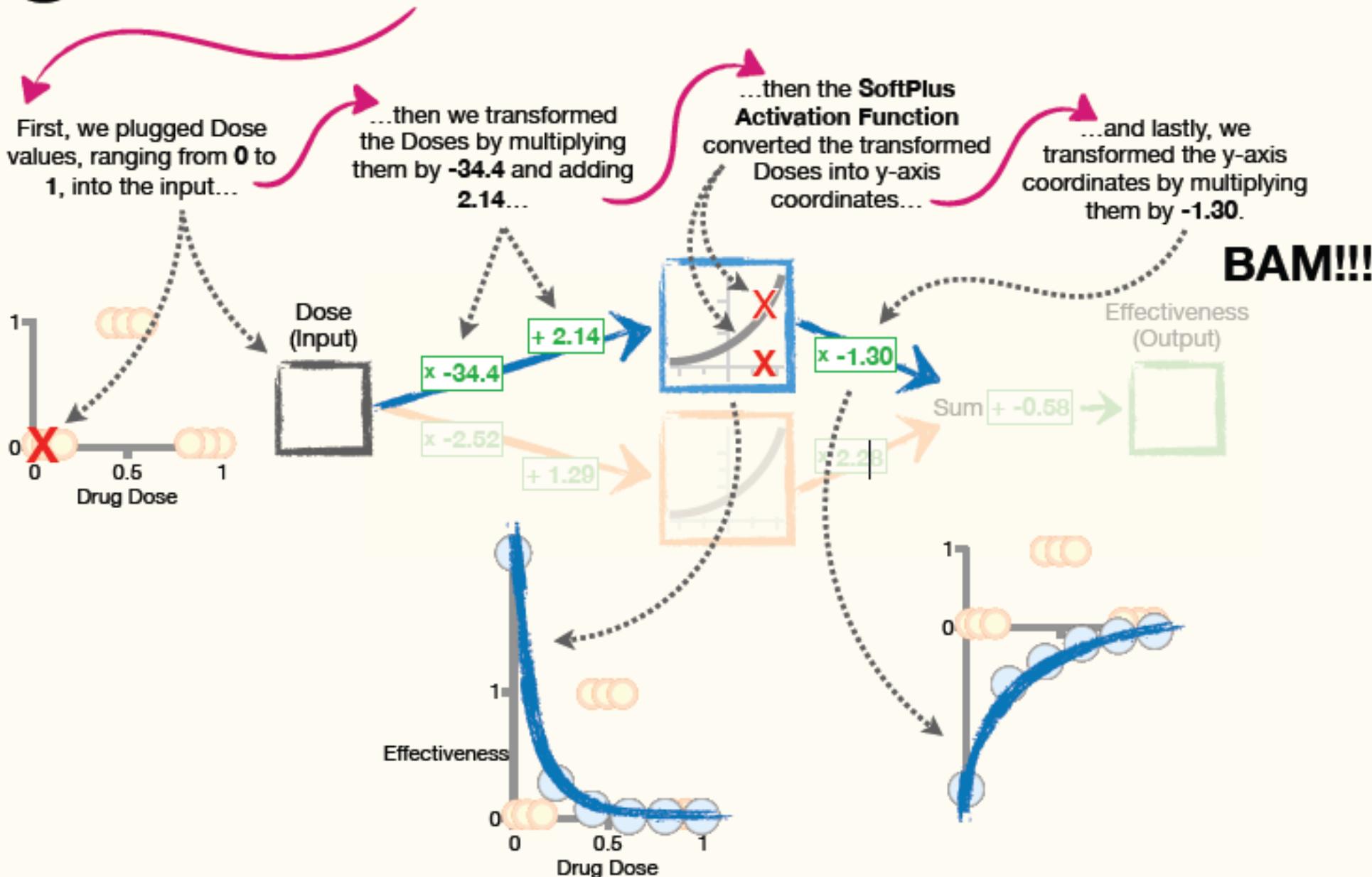
Likewise, when we multiply all of the y-axis coordinates on the **blue curve** by **-1.30**...

...we end up flipping and stretching the *original blue curve* to get a *new blue curve*.



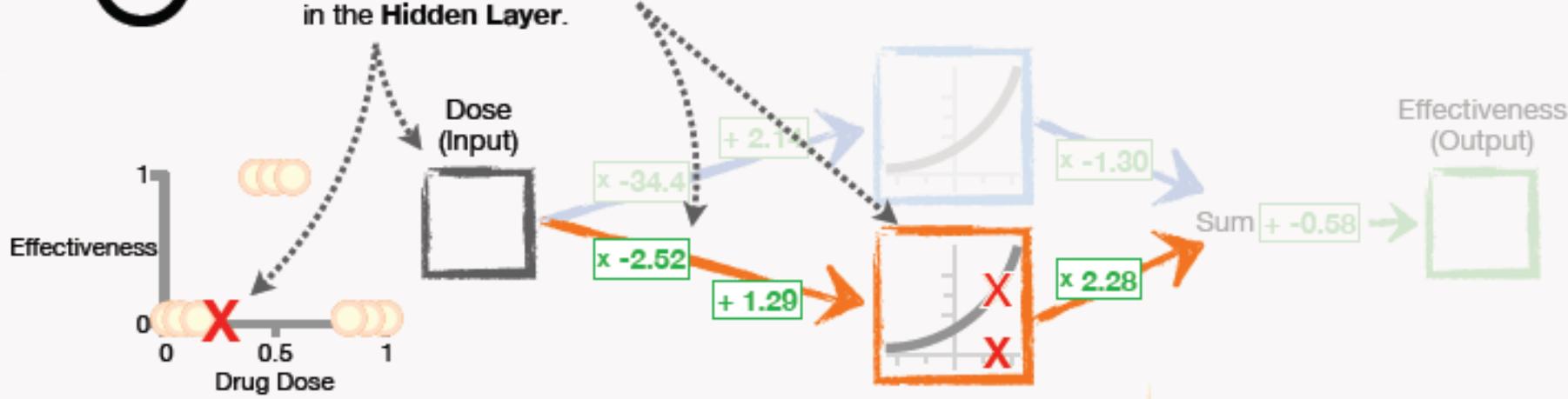
13

Okay, we just finished a major step, so this is a good time to review what we've done so far.



14

Now we run Dose values through the connection to the *bottom Node* in the **Hidden Layer**.

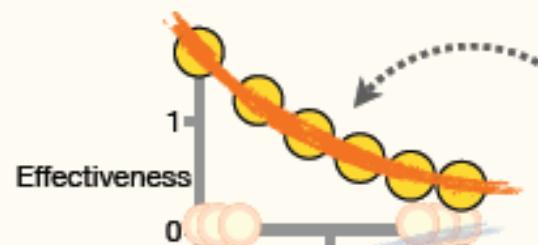


15

The good news is that the only difference between what we just did and what we're doing now is that now we multiply the Doses by **-2.52** and add **1.29**...

...before using the **SoftPlus Activation Function** to convert the transformed Doses into y-axis coordinates...

...and then we stretch the y-axis coordinates by multiplying them by **2.28** to get this **orange curve**.

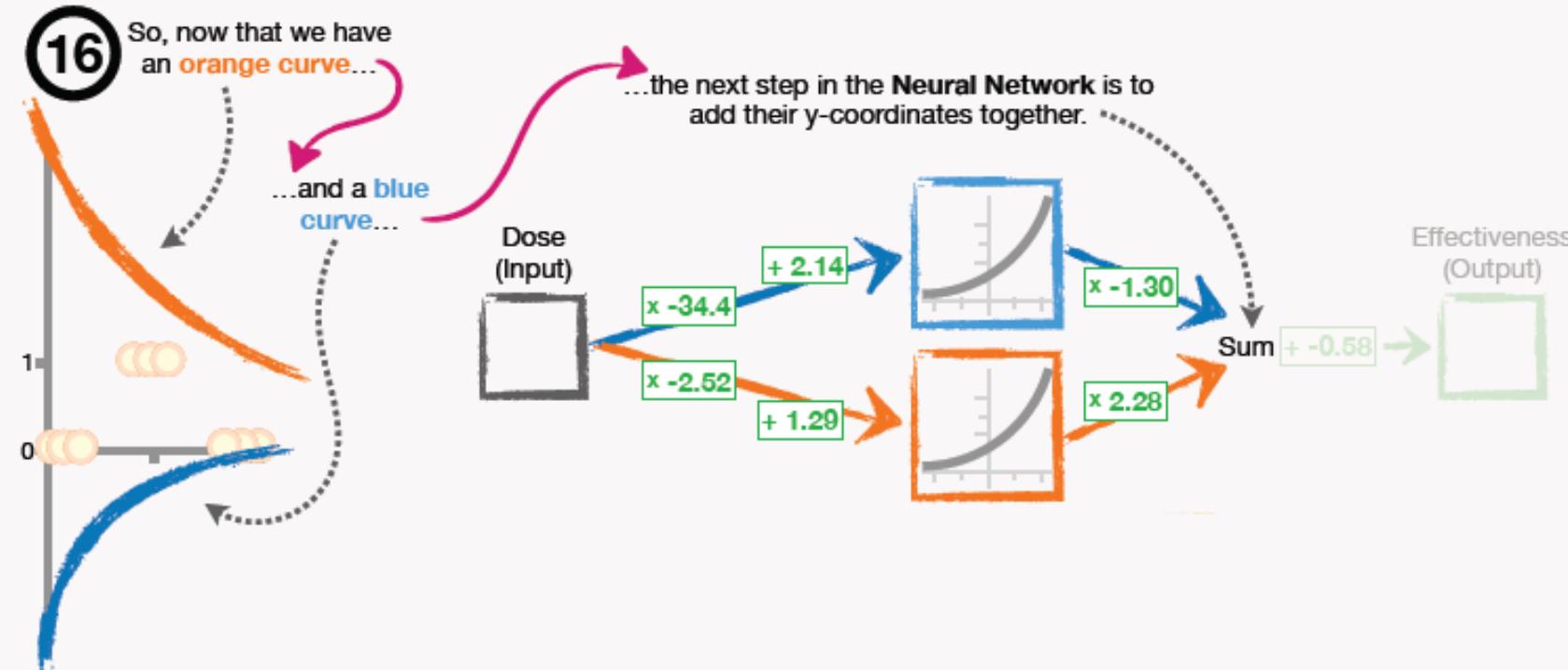


16

So, now that we have
an orange curve...

...and a blue
curve...

...the next step in the **Neural Network** is to
add their y-coordinates together.

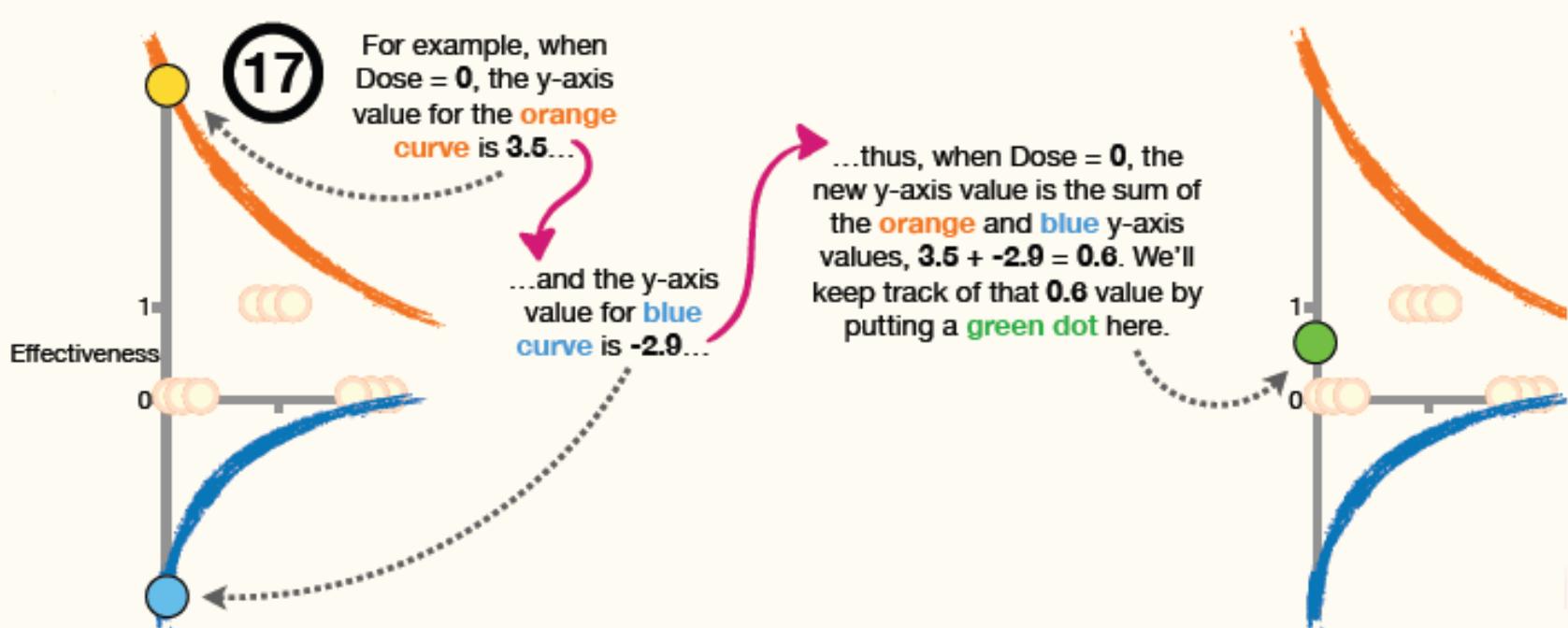


17

For example, when
Dose = 0, the y-axis
value for the orange
curve is 3.5...

...and the y-axis
value for blue
curve is -2.9...

...thus, when Dose = 0, the
new y-axis value is the sum of
the orange and blue y-axis
values, $3.5 + -2.9 = 0.6$. We'll
keep track of that 0.6 value by
putting a green dot here.

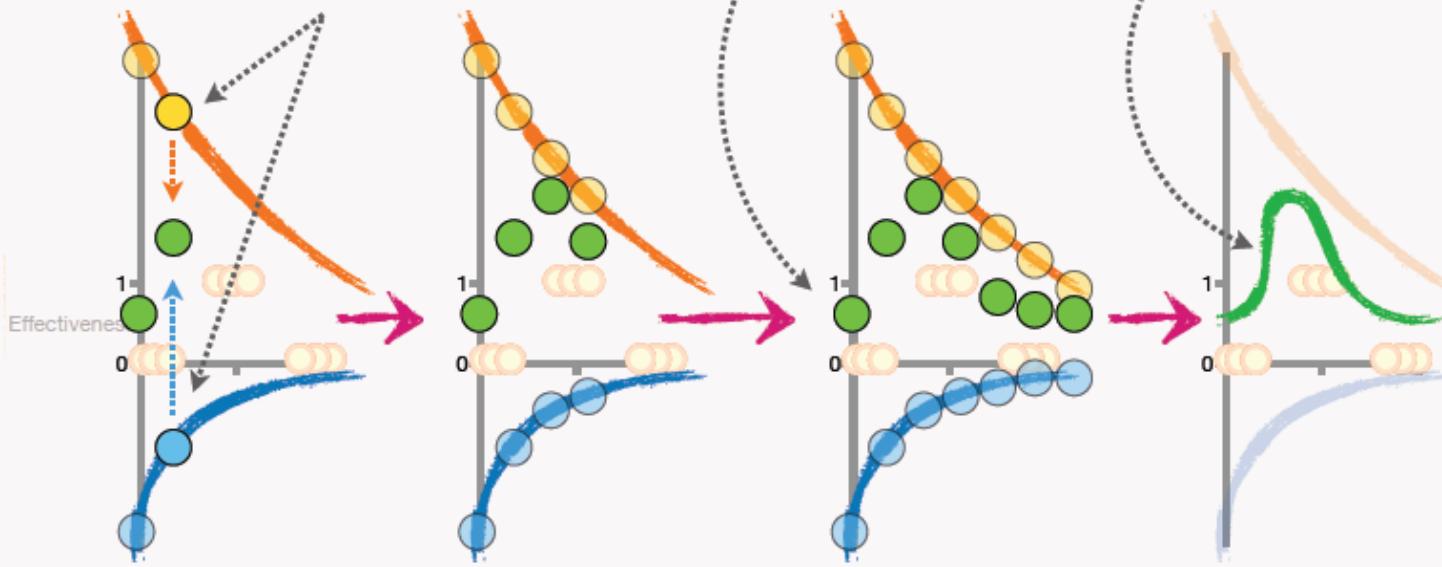


18

Then, for the remaining Dose values, we just add the y-axis coordinates of the blue and orange curves...

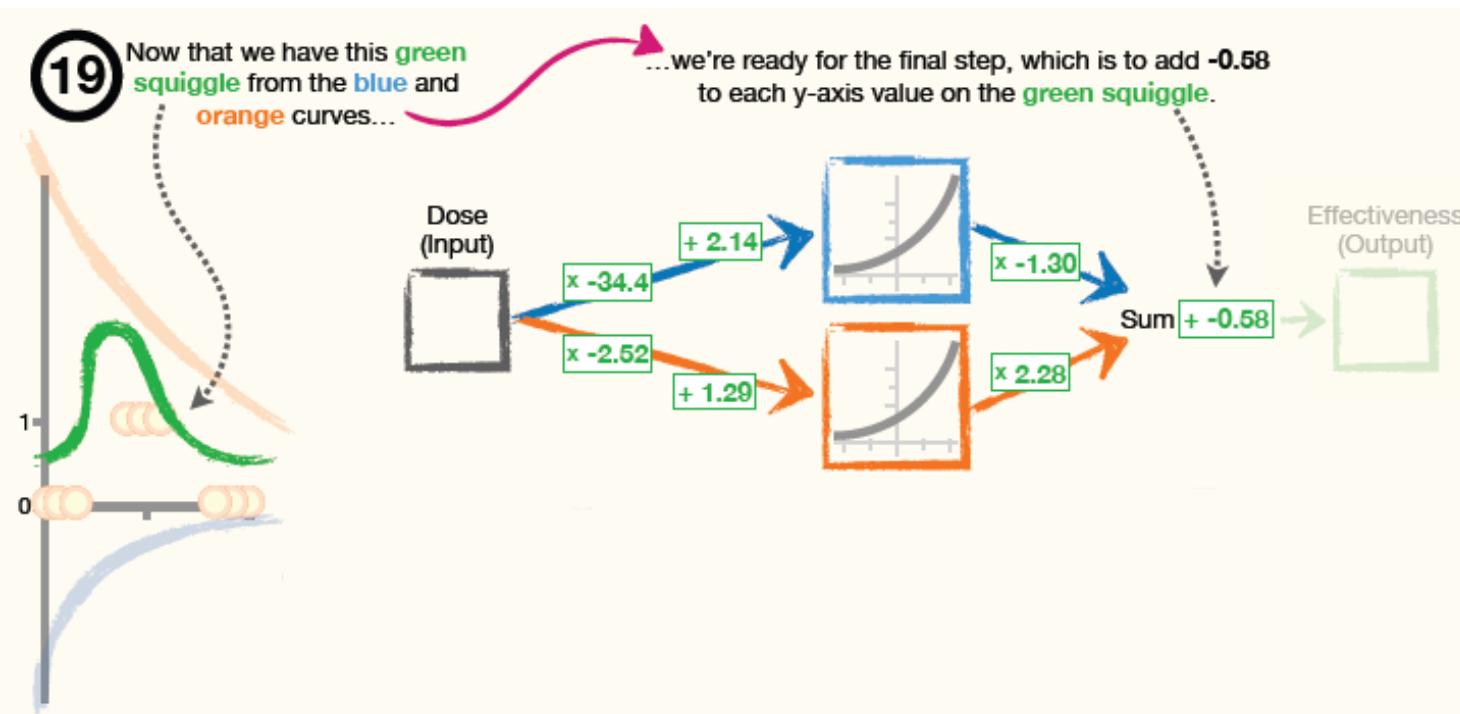
...plot the resulting green dot values...

...and, after connecting the dots, we ultimately end up with this green squiggle.

**19**

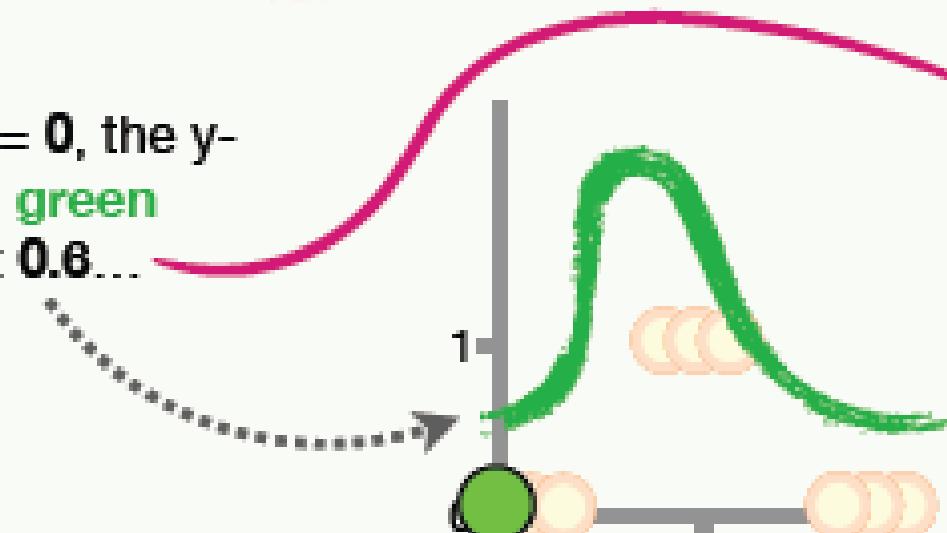
Now that we have this green squiggle from the blue and orange curves...

...we're ready for the final step, which is to add -0.58 to each y-axis value on the green squiggle.



20

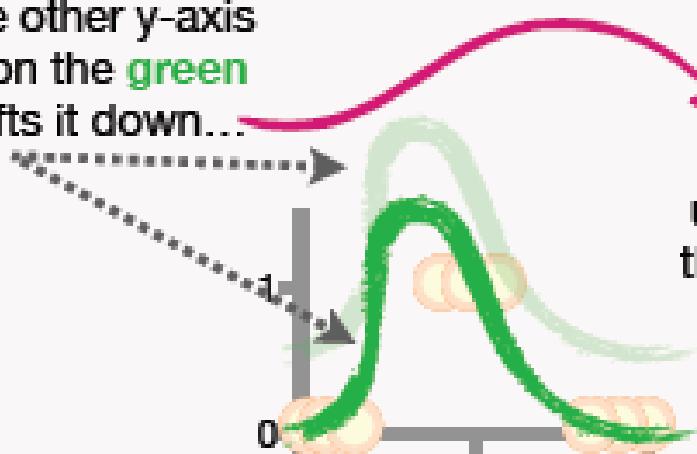
For example, when Dose = 0, the y-axis coordinate on the **green squiggle** starts out at **0.6**...



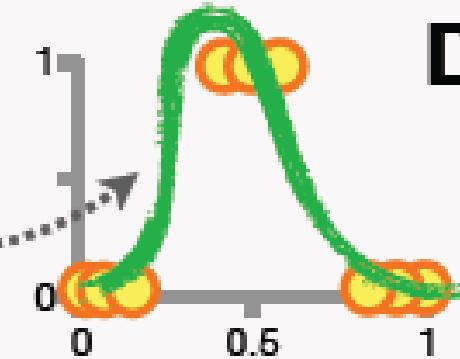
...but after subtracting **0.58**, the new y-axis coordinate for when Dose = **0** is **0.0** (rounded to the nearest tenth).

21

Likewise, subtracting **0.58** from all of the other y-axis coordinates on the **green squiggle** shifts it down...



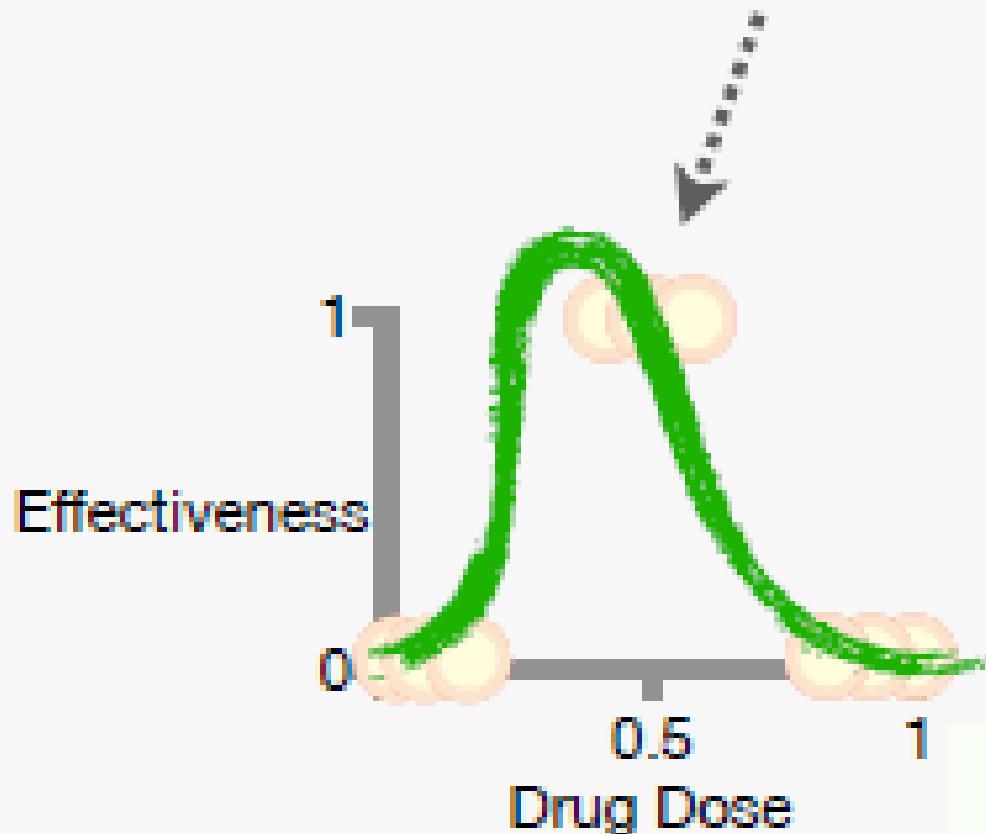
...and, ultimately, we end up with a **green squiggle** that fits the **Training Data**.



**DOUBLE
BAM!!!**

23

Now, if we're wondering if a medium Dose of 0.5 will be effective, we can look at the green squiggle and see that the output from the Neural Network will be 1, and thus, Dose = 0.5 will be effective.

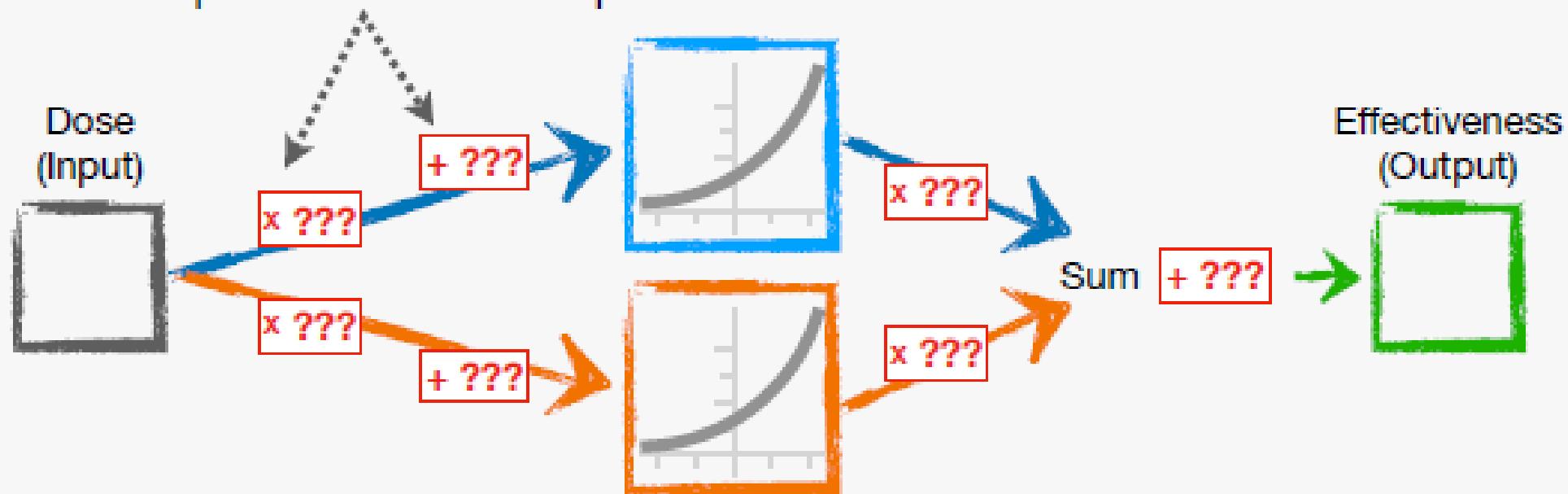


24

Alternatively, if we plug Dose = 0.5 into the Neural Network and do the math, we get 1, and thus, Dose = 0.5 will be effective.

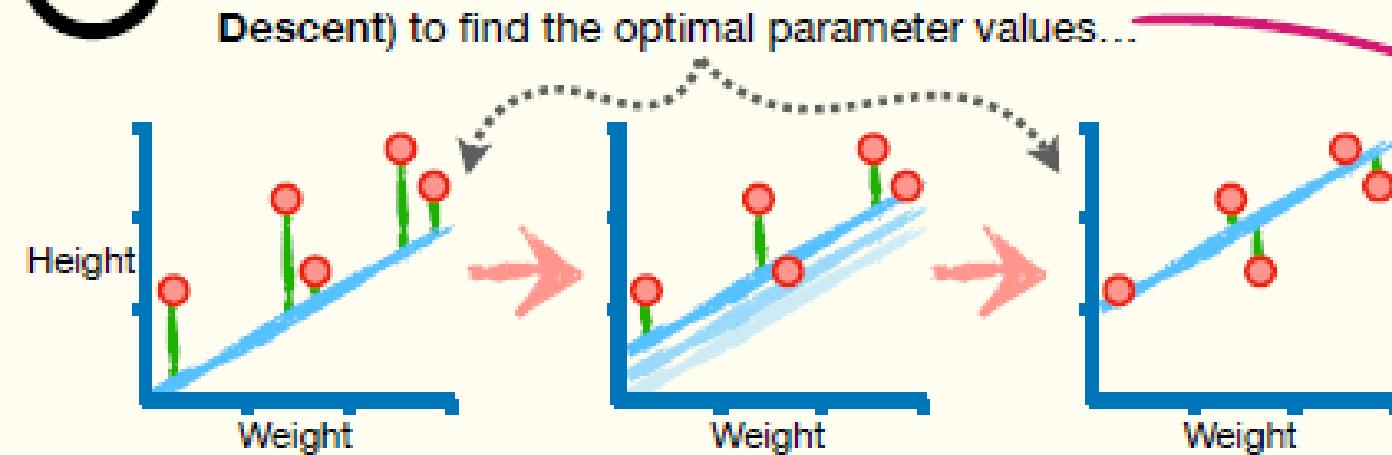
1

The Problem: Just like for **Linear Regression**, **Neural Networks** have parameters that we need to optimize to fit a squiggle or bent shape to data.
How do we find the optimal values for these parameters?



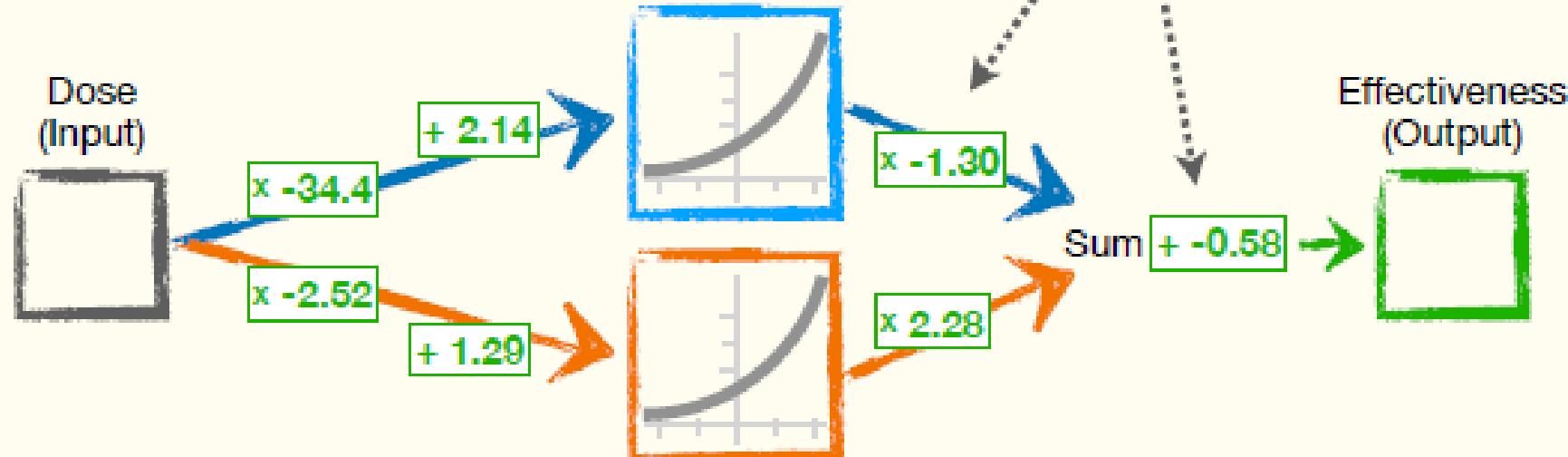
2

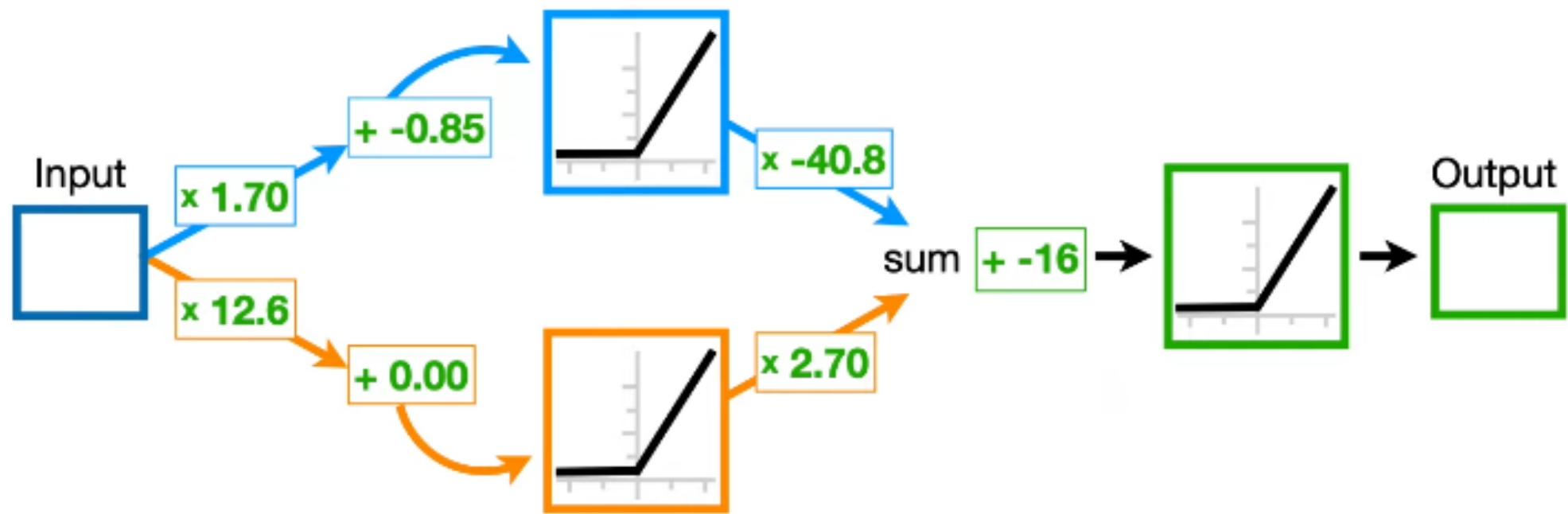
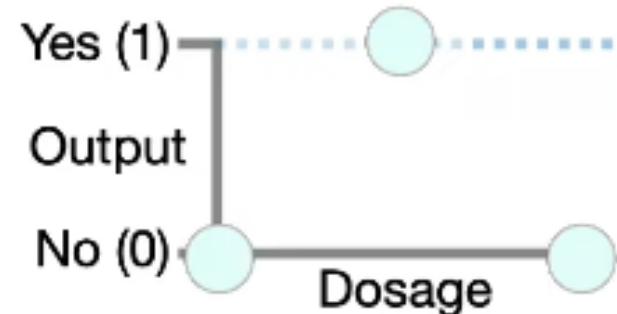
A Solution: Just like for Linear Regression, we can use Gradient Descent (or Stochastic Gradient Descent) to find the optimal parameter values...

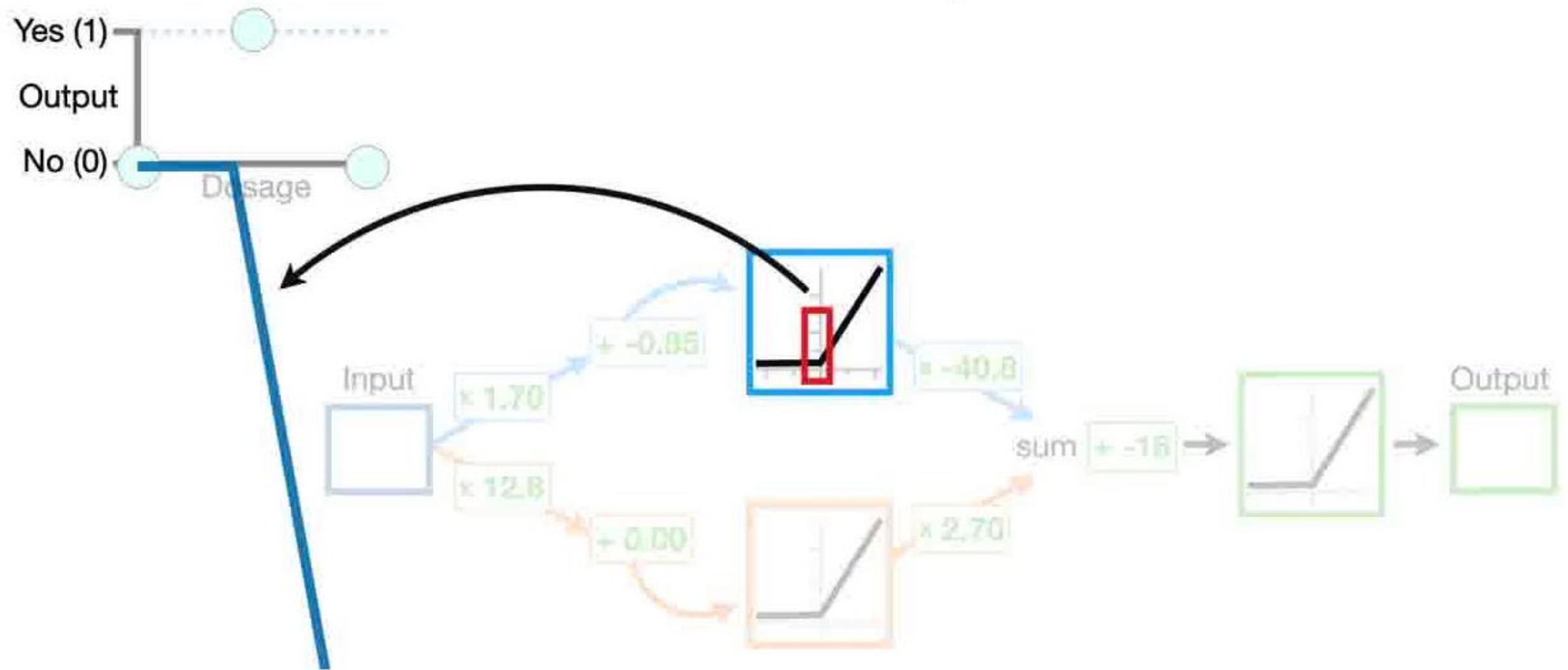


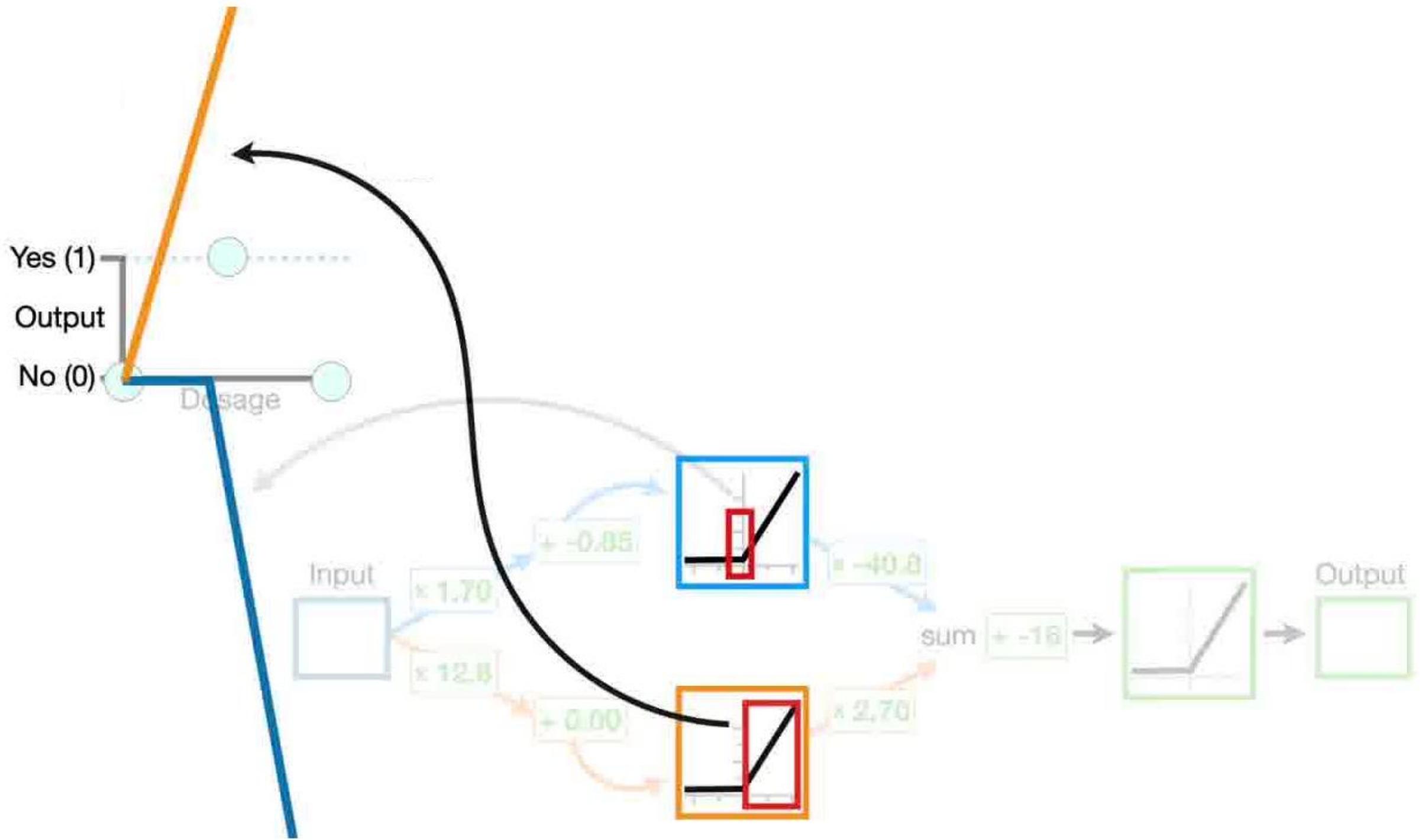
...however, we don't call it Gradient Descent. That would be too easy. Instead, because of how the derivatives are found for each parameter in a Neural Network (from the back to the front), we call it Backpropagation.

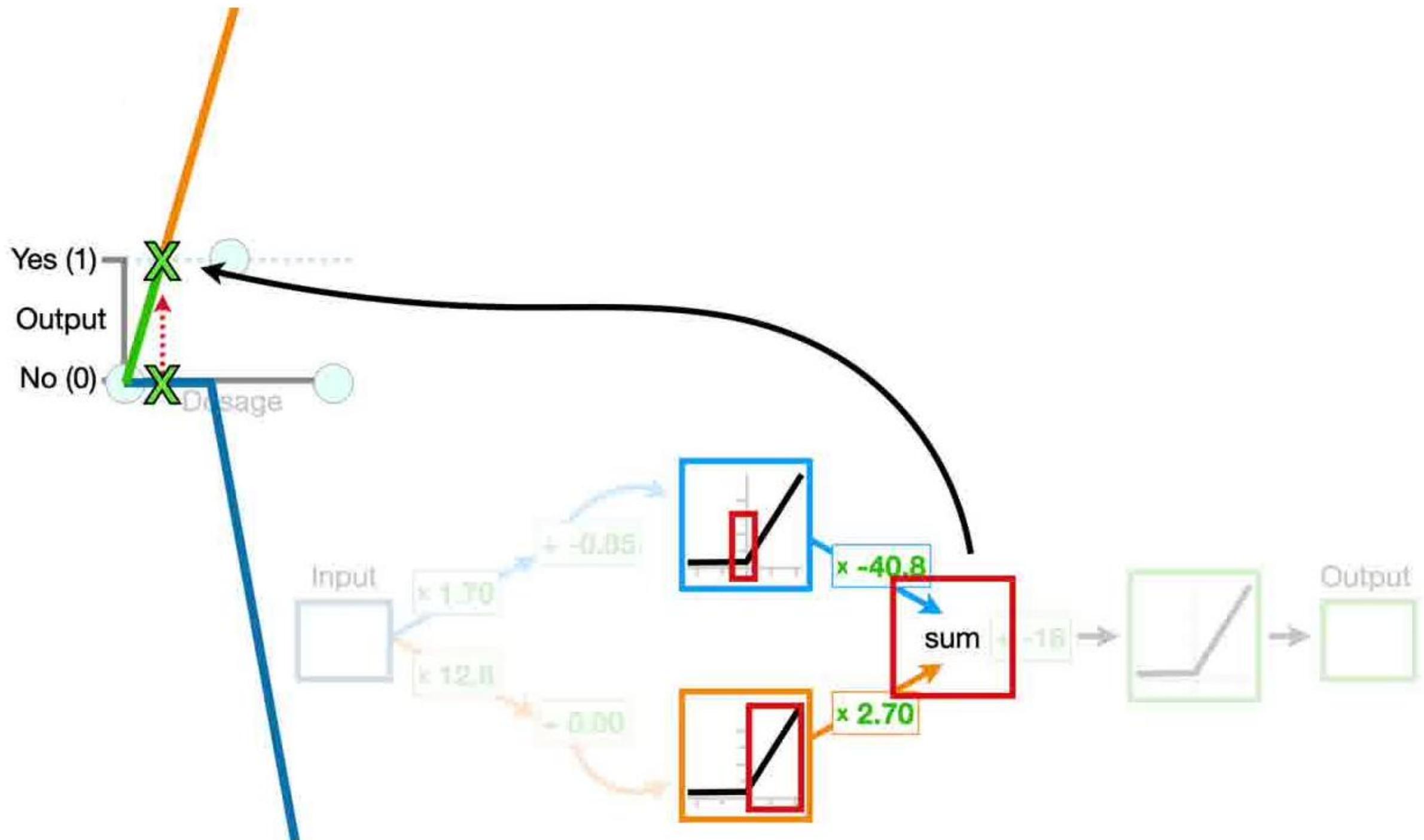
BAM!!!

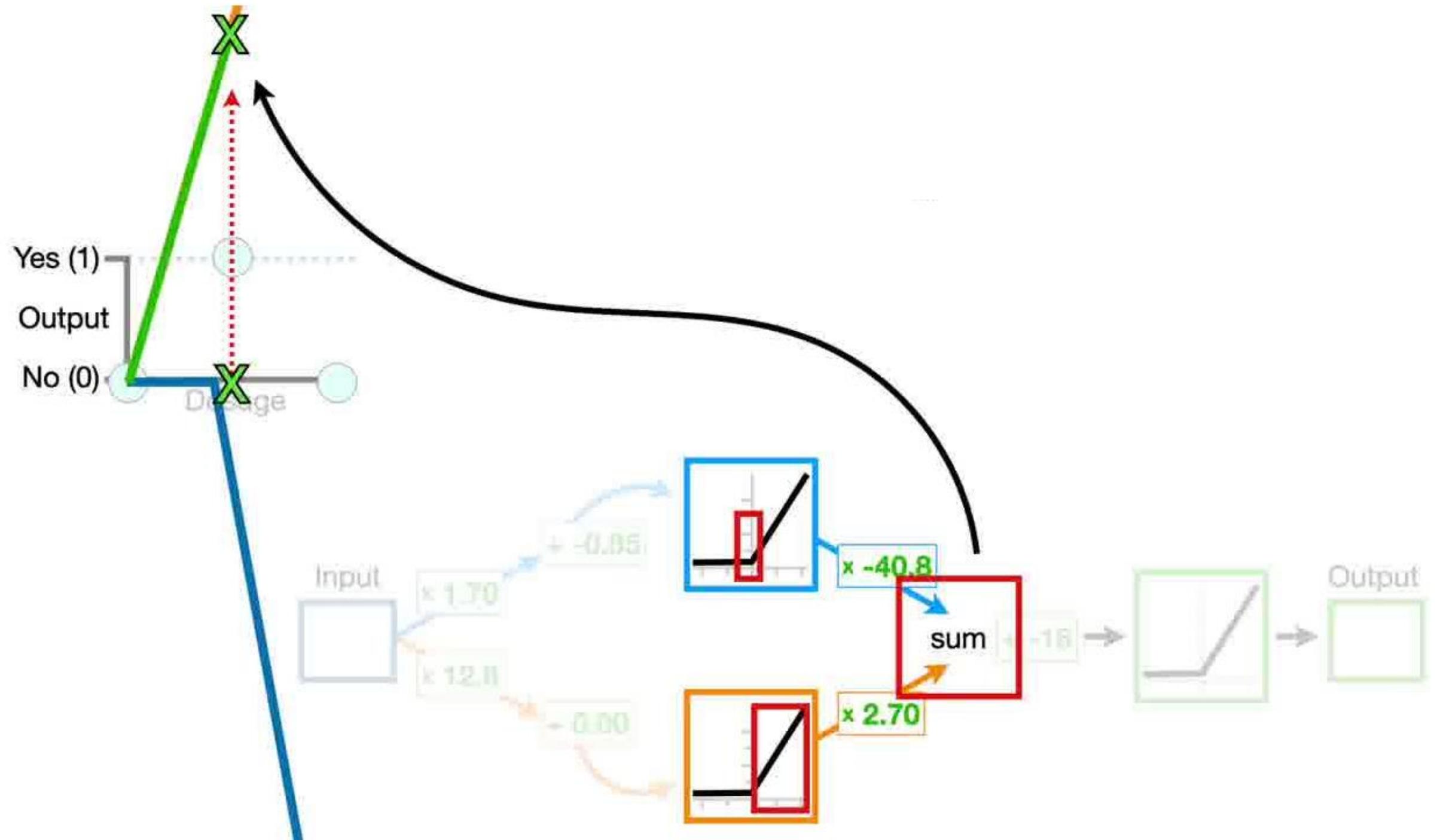


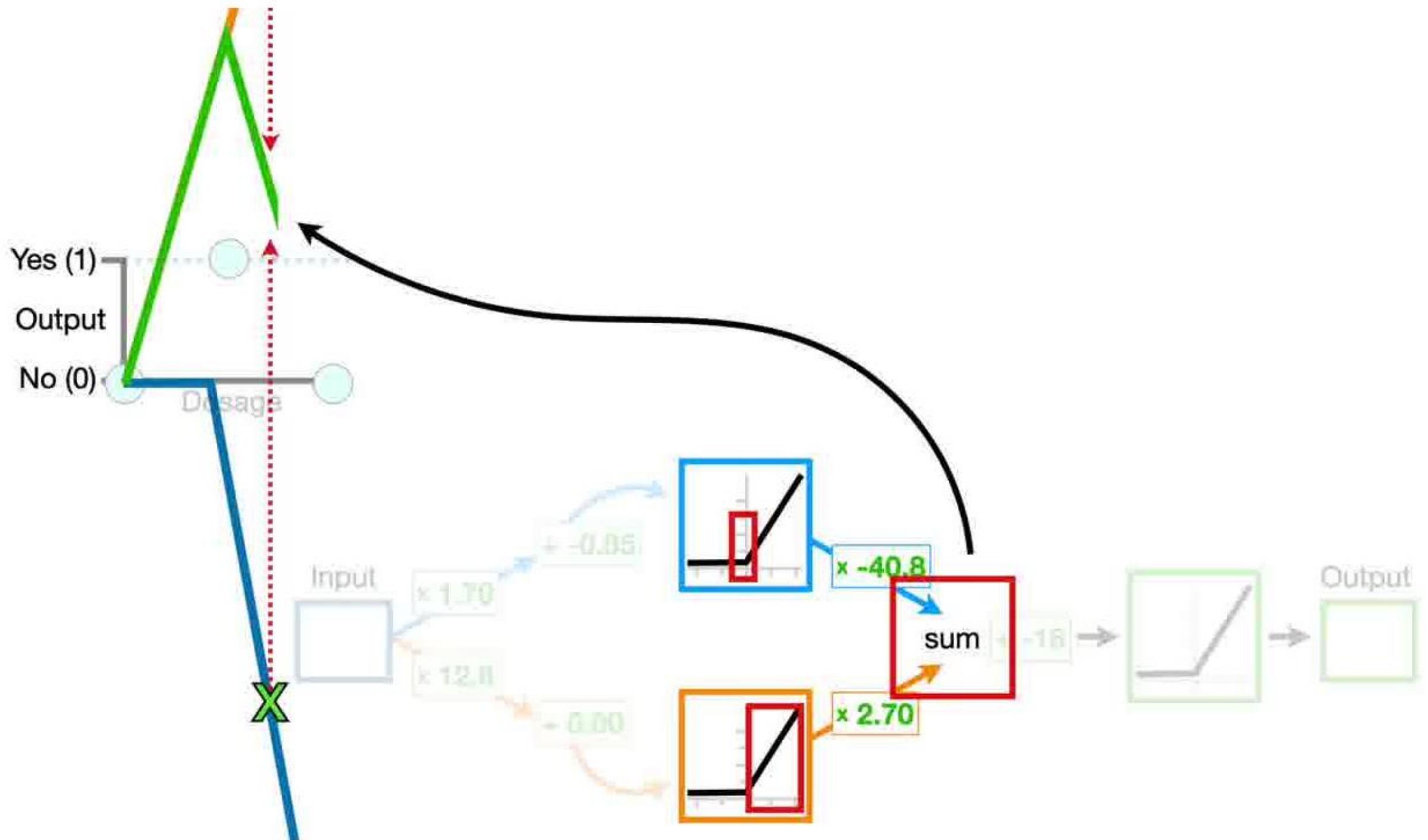


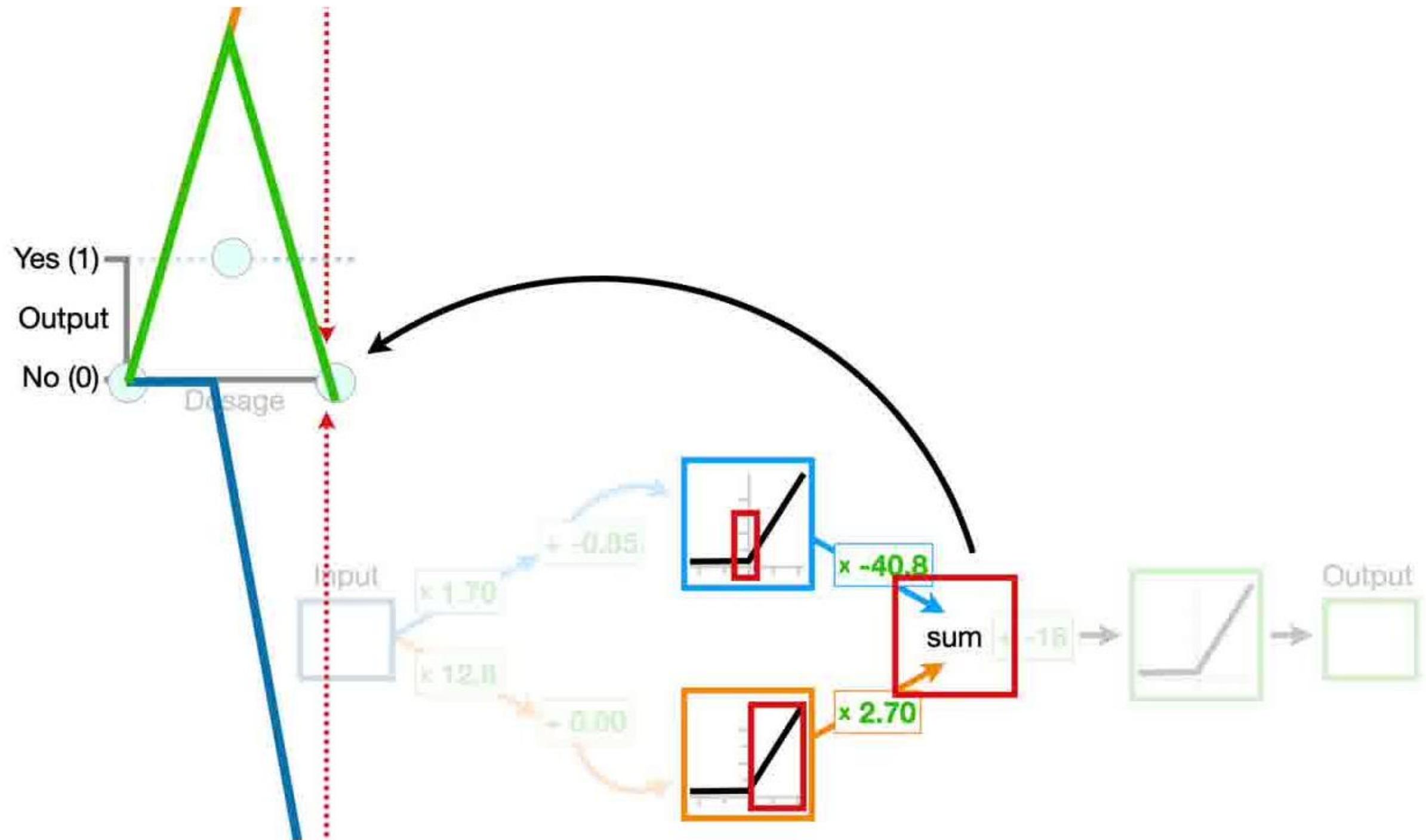


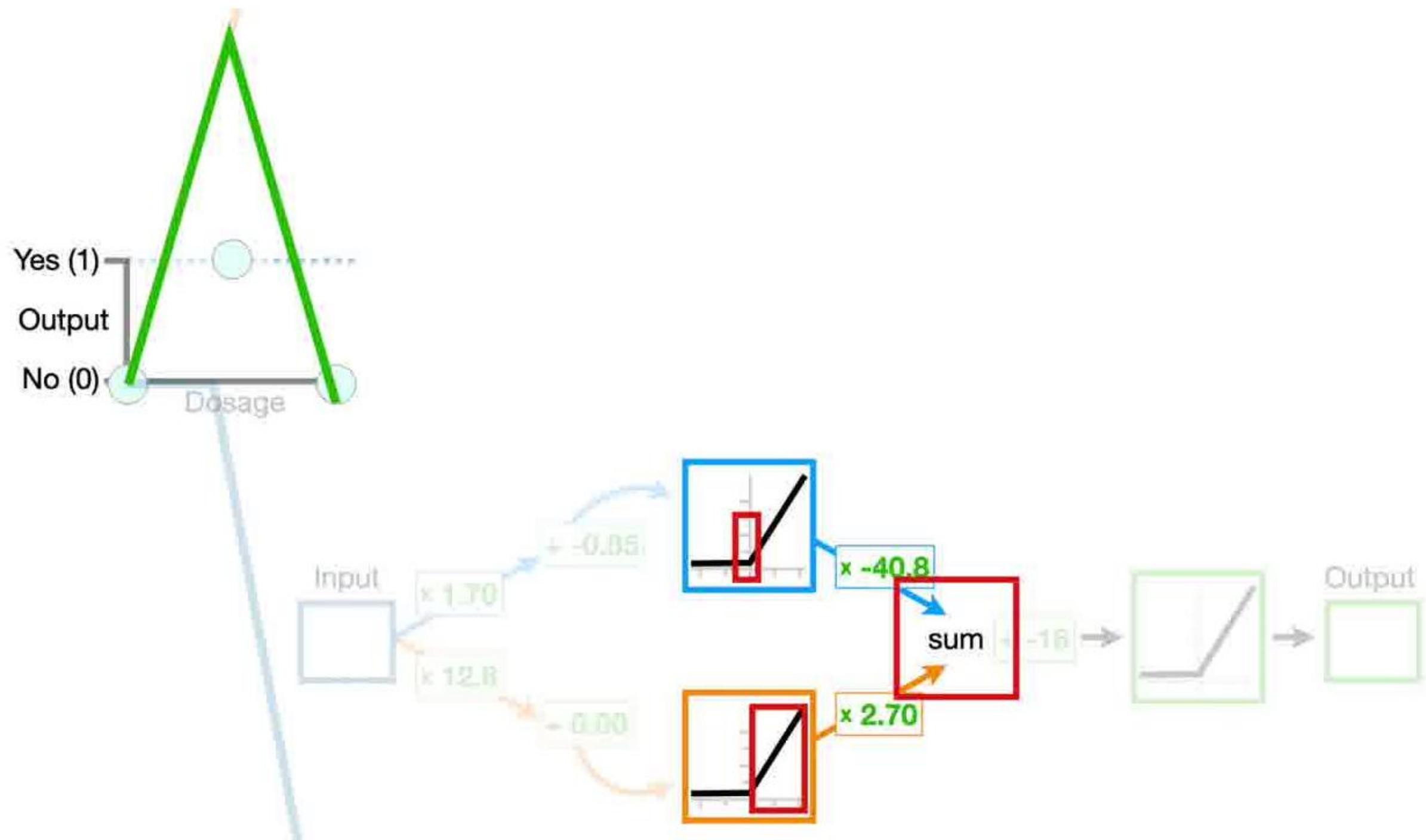


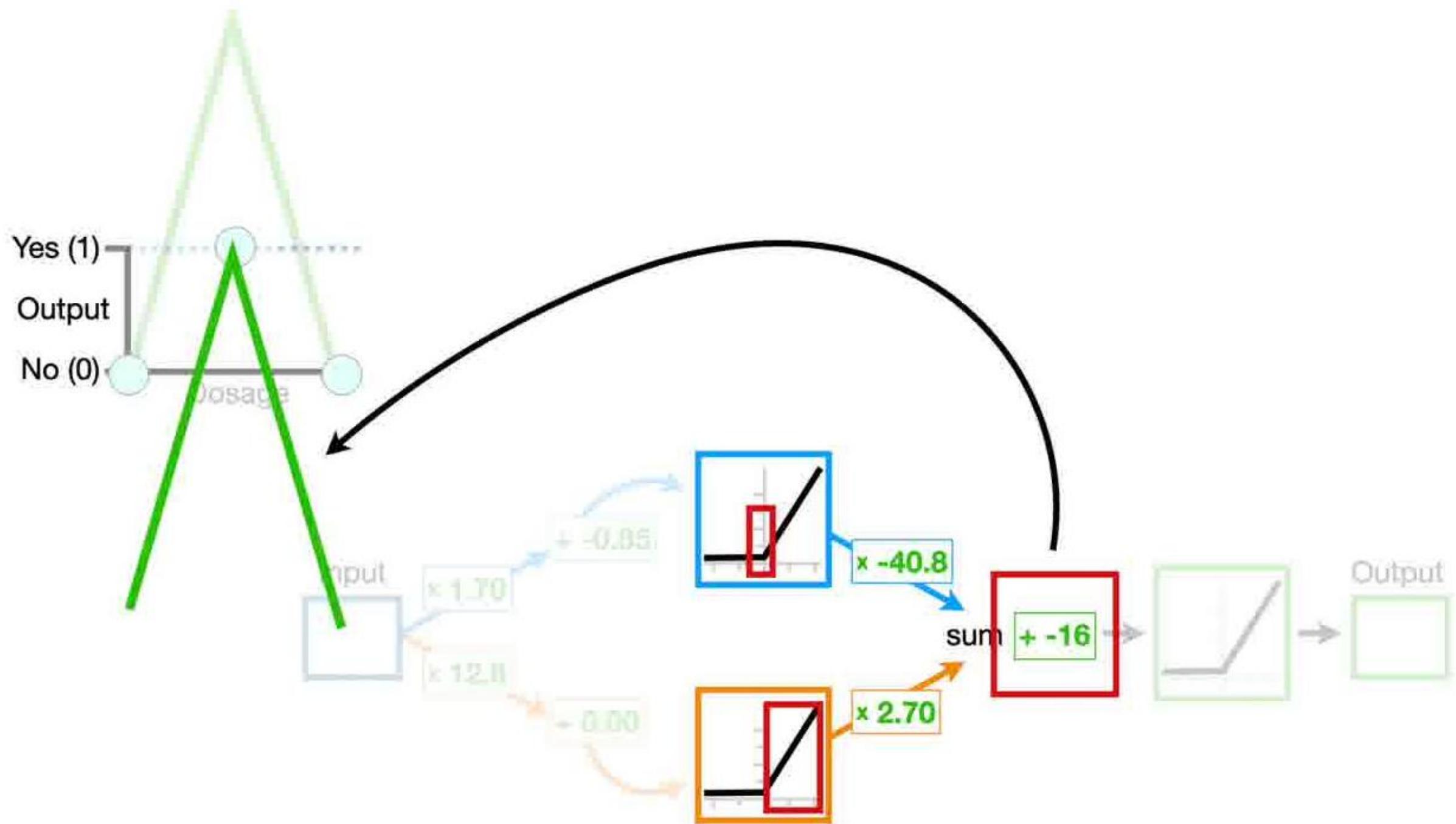


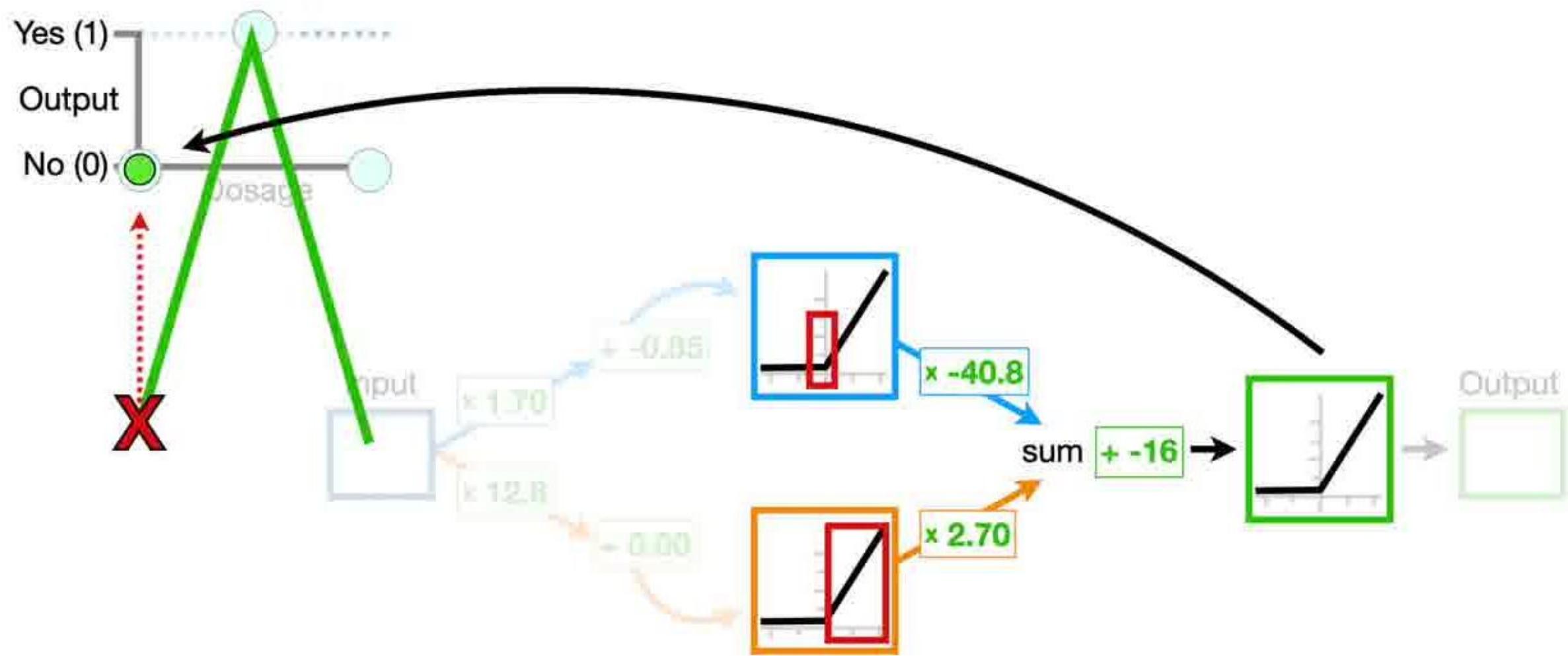


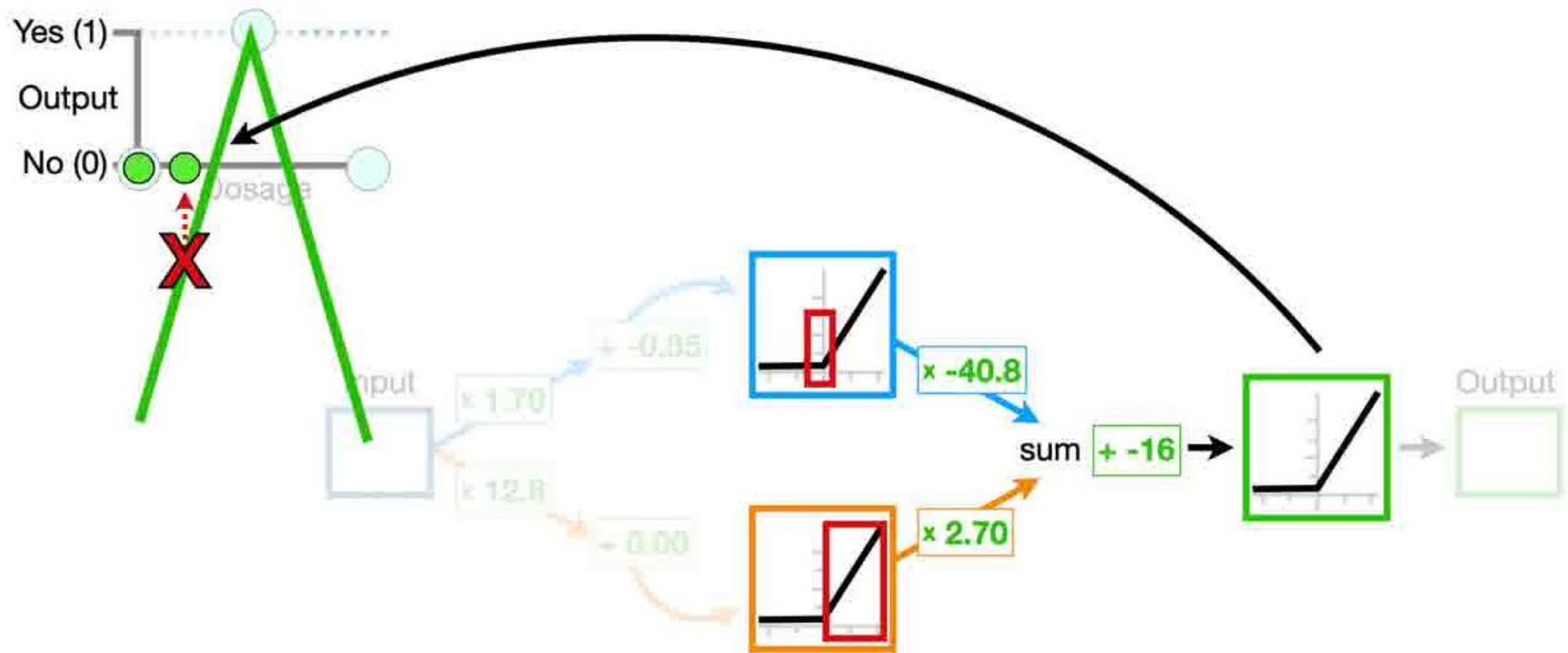


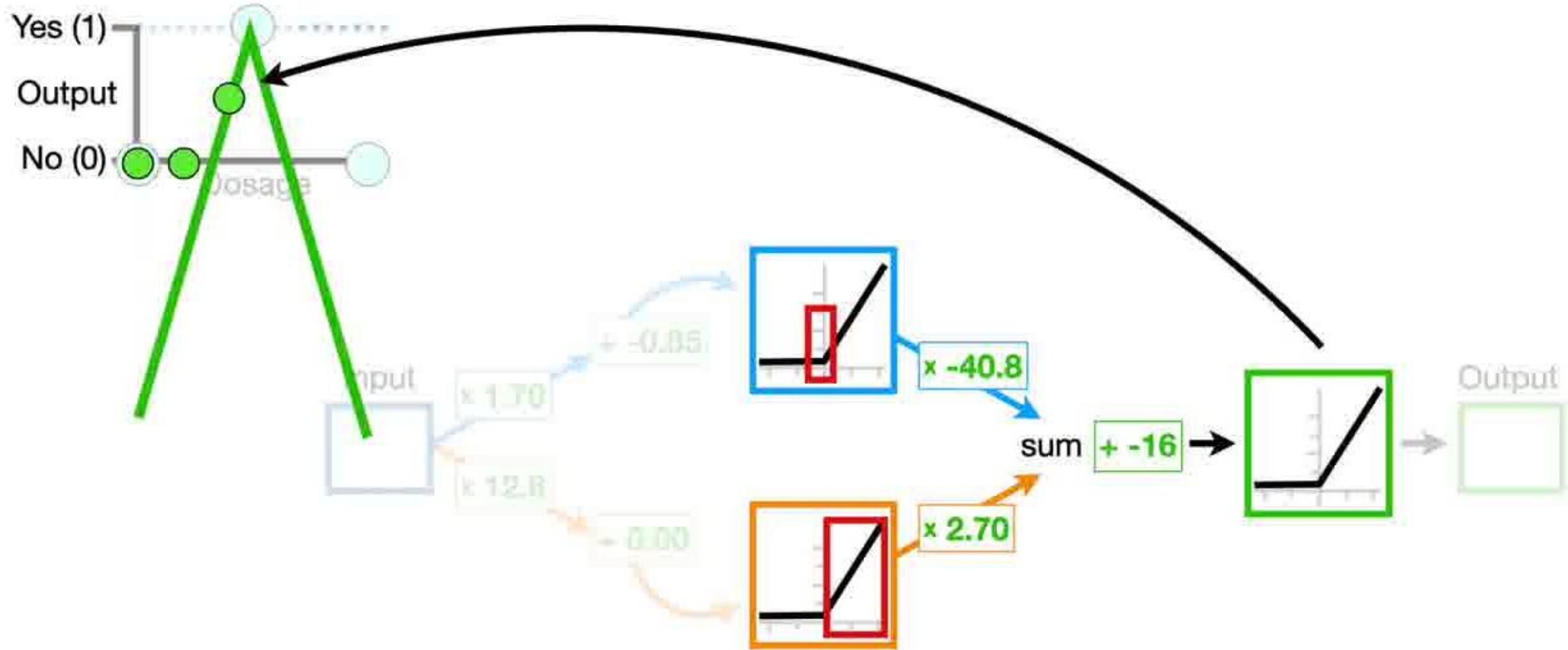


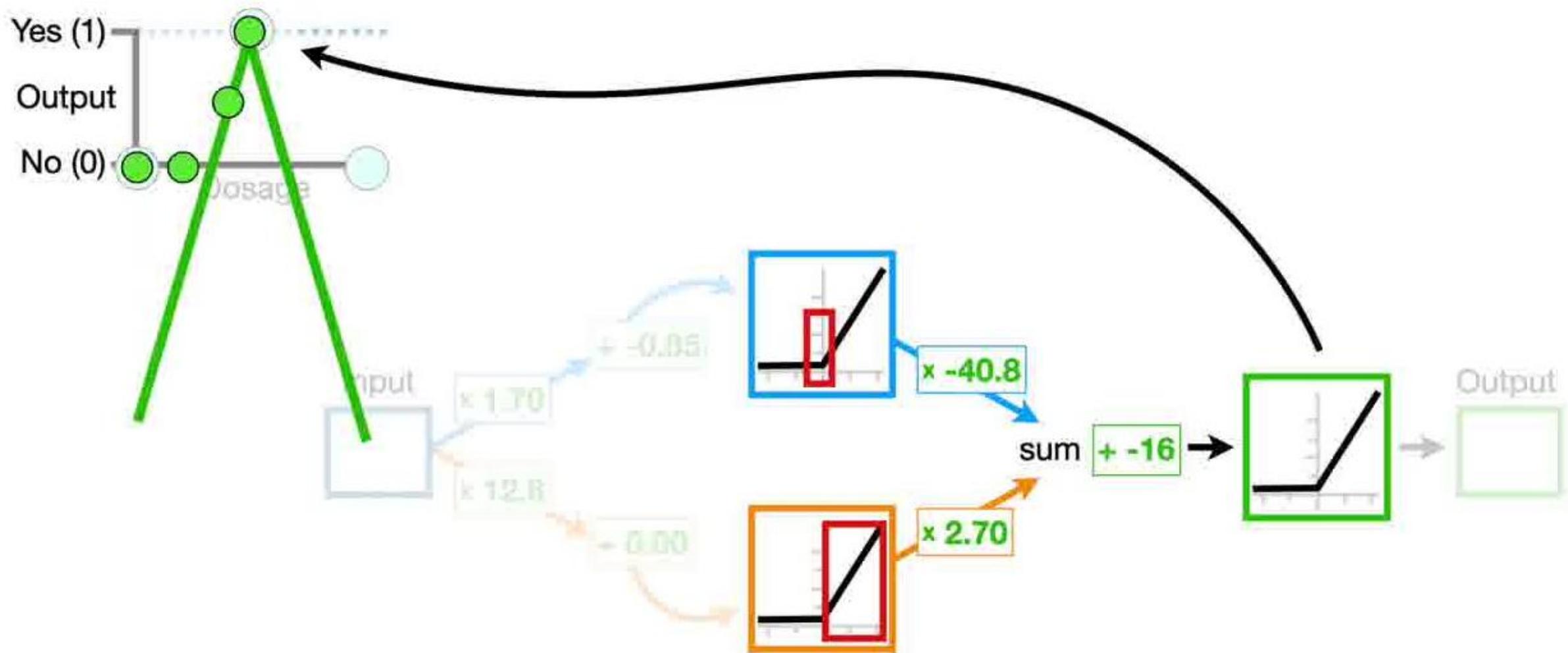


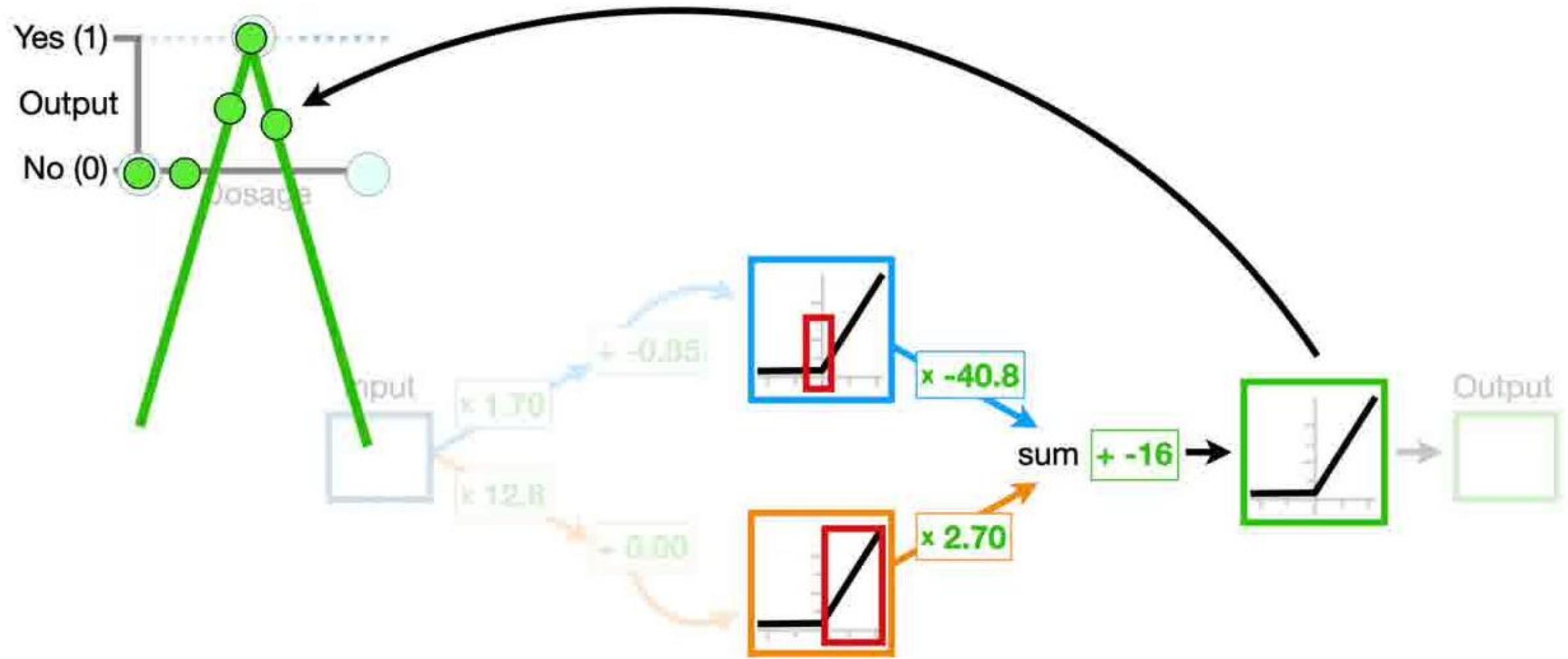


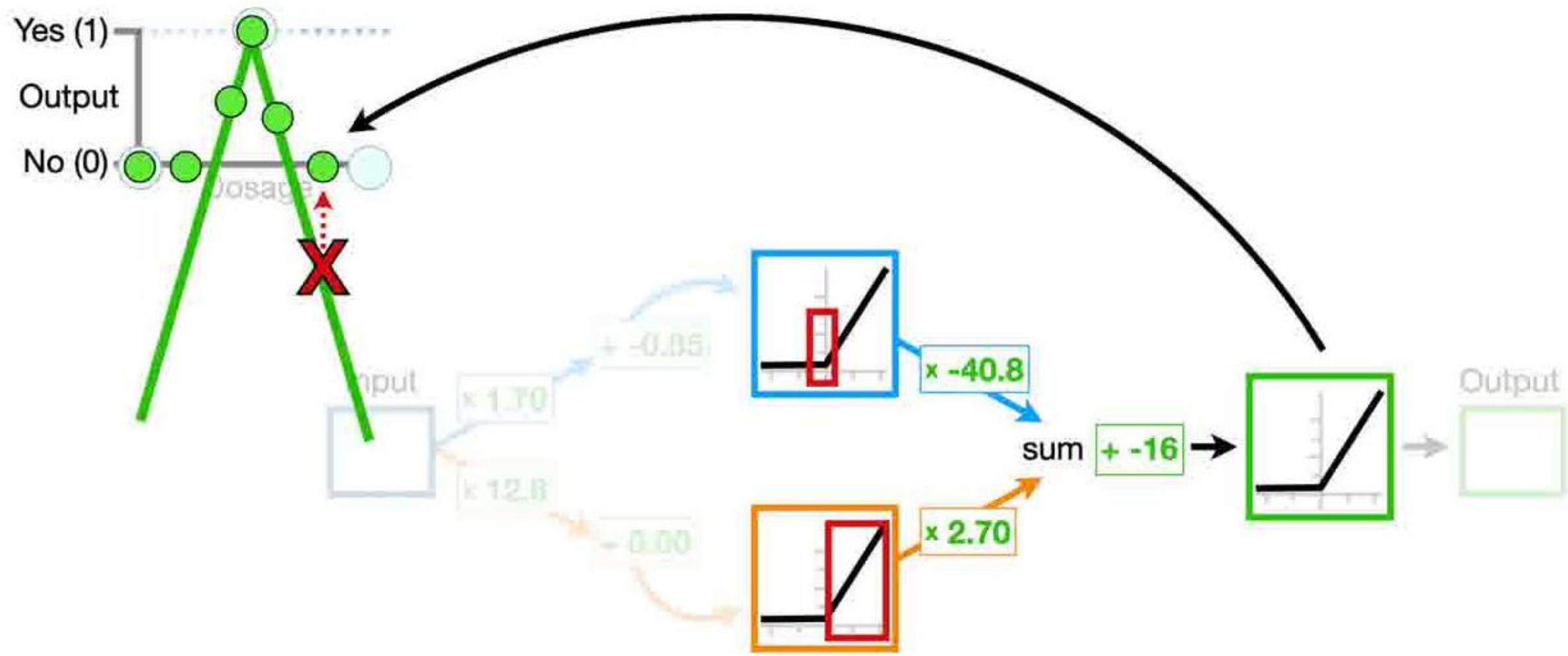


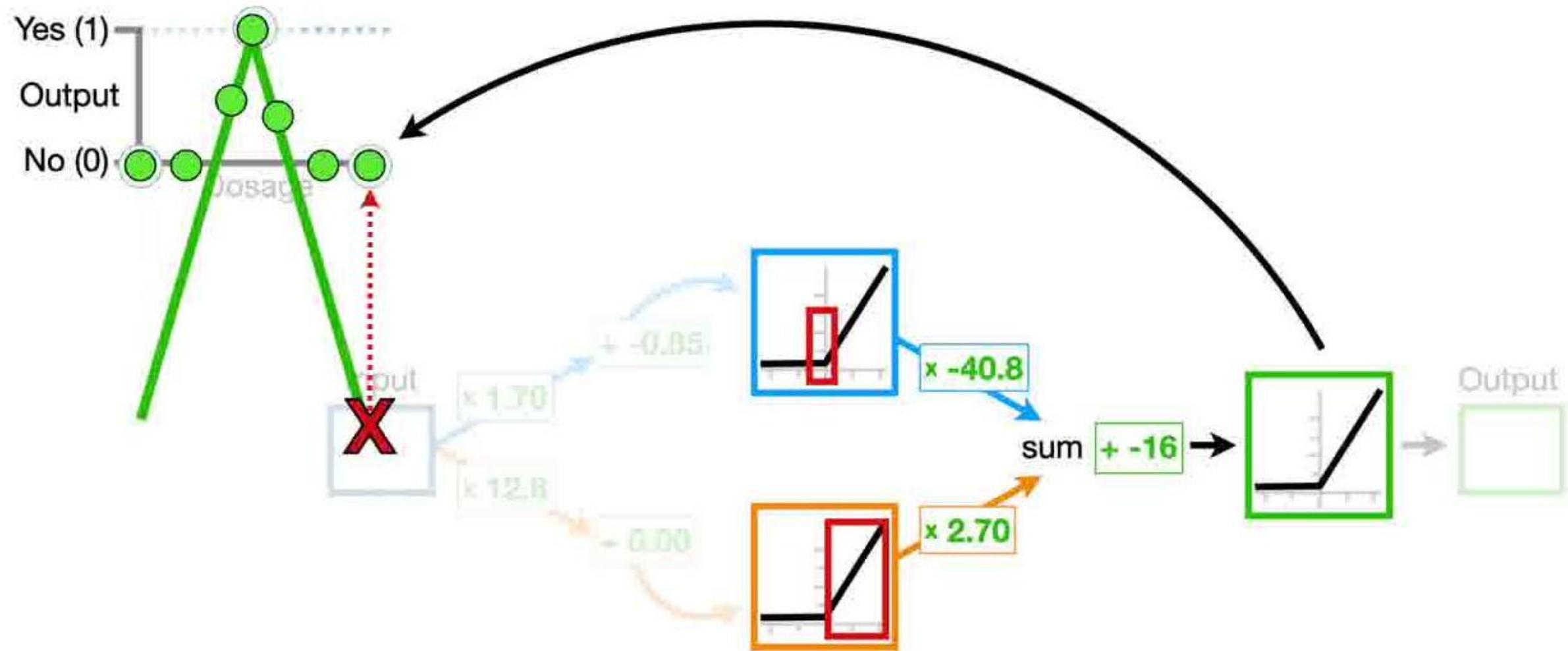


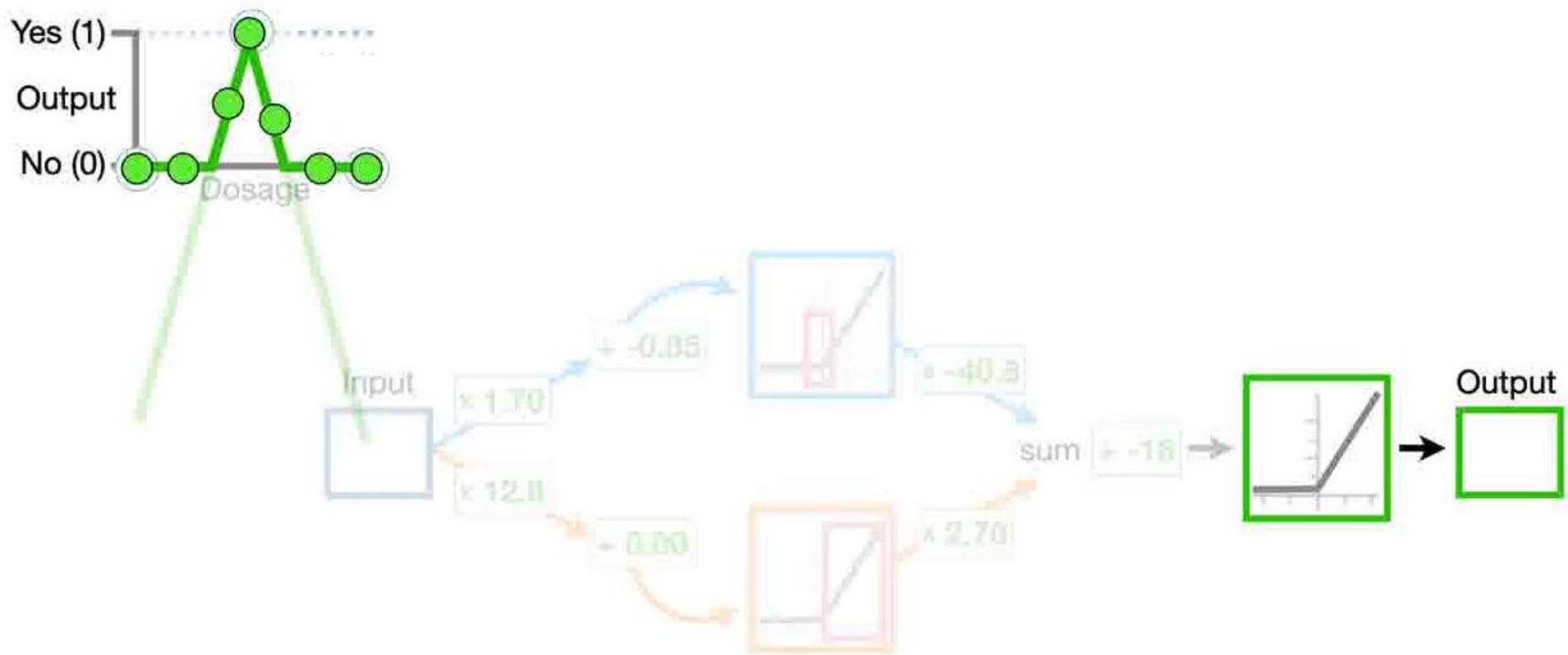


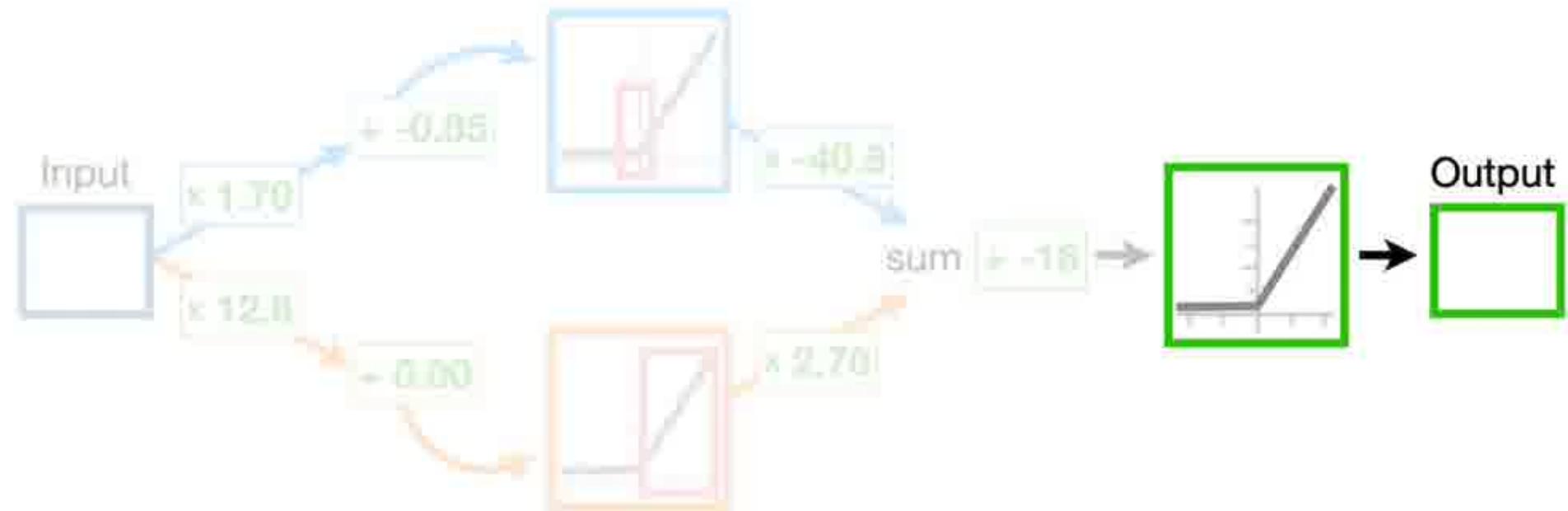
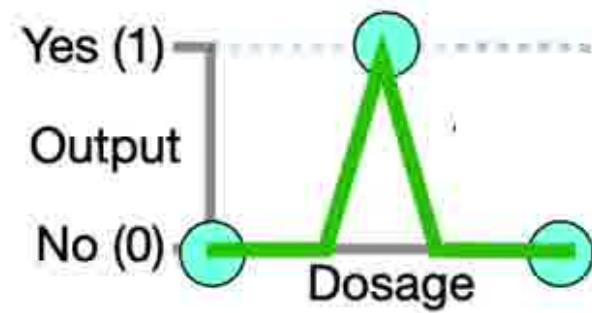










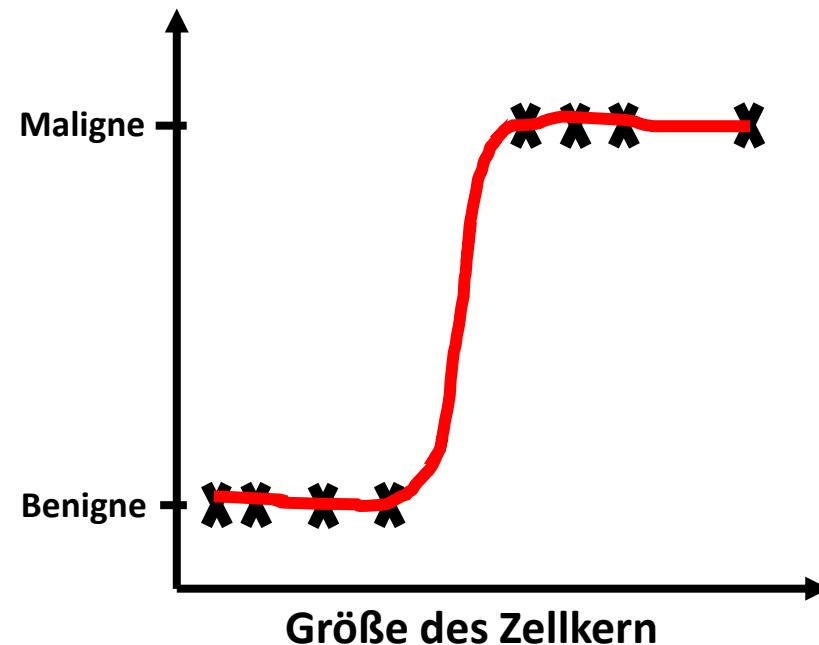


Schritt 1: Man füttere eine KI mit einem gewaltigen Trainingsdatensatz.

Schritt 2: KI versucht die Daten notfalls sogar in einem mehrdimensionalen Koordinatensystem darzustellen.

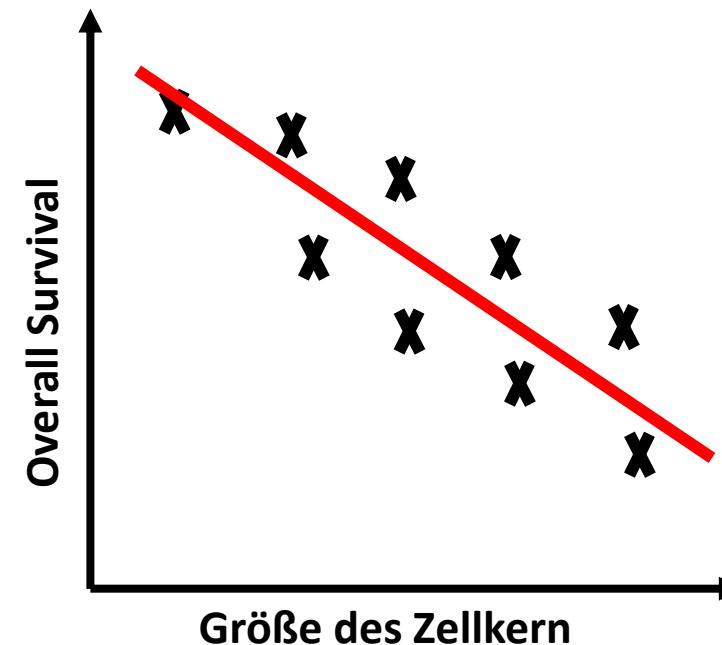
Schritt 3: KI versucht eine Funktion in die Datenpunkte zu erzwingen

bei Logistic Regression...



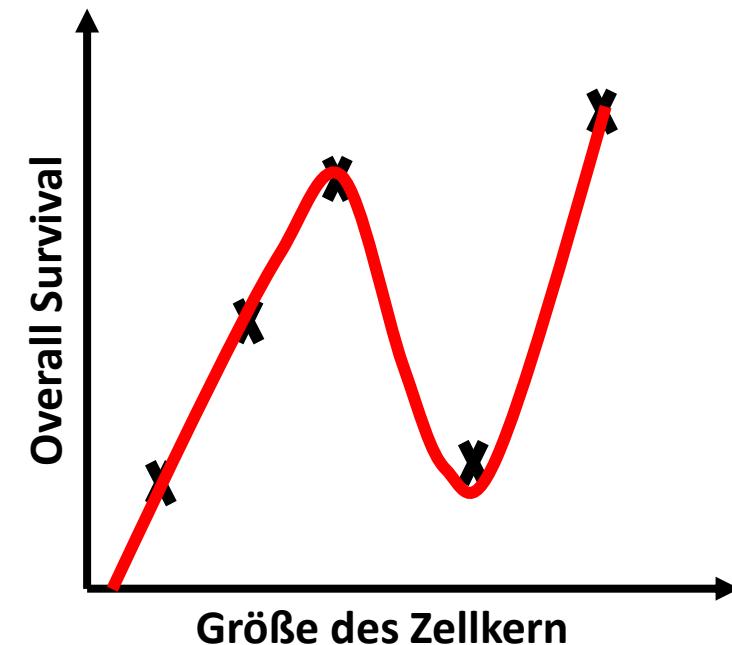
... eine sigmoide Funktion

bei Linear Regression...



... eine lineare Funktion

bei neuronalen Netzen ...



... eine komplexere Funktion

Schritt 4: Abgleich der modellierten Funktion mit einem Trainingsdatensatz -> Berechnung der Accuracy...

Und wie kann man ein Neuronales
Netz programmieren?

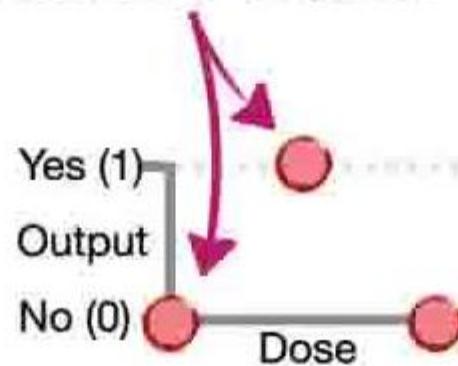
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD

import matplotlib.pyplot as plt
import seaborn as sns
```

The first thing we do is
import the **Python** modules
that we will use.

```
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from torch.optim import SGD  
  
import matplotlib.pyplot as plt  
import seaborn as sns
```

We will use **torch** to create **tensors** to store all of the numerical values, including the raw data...



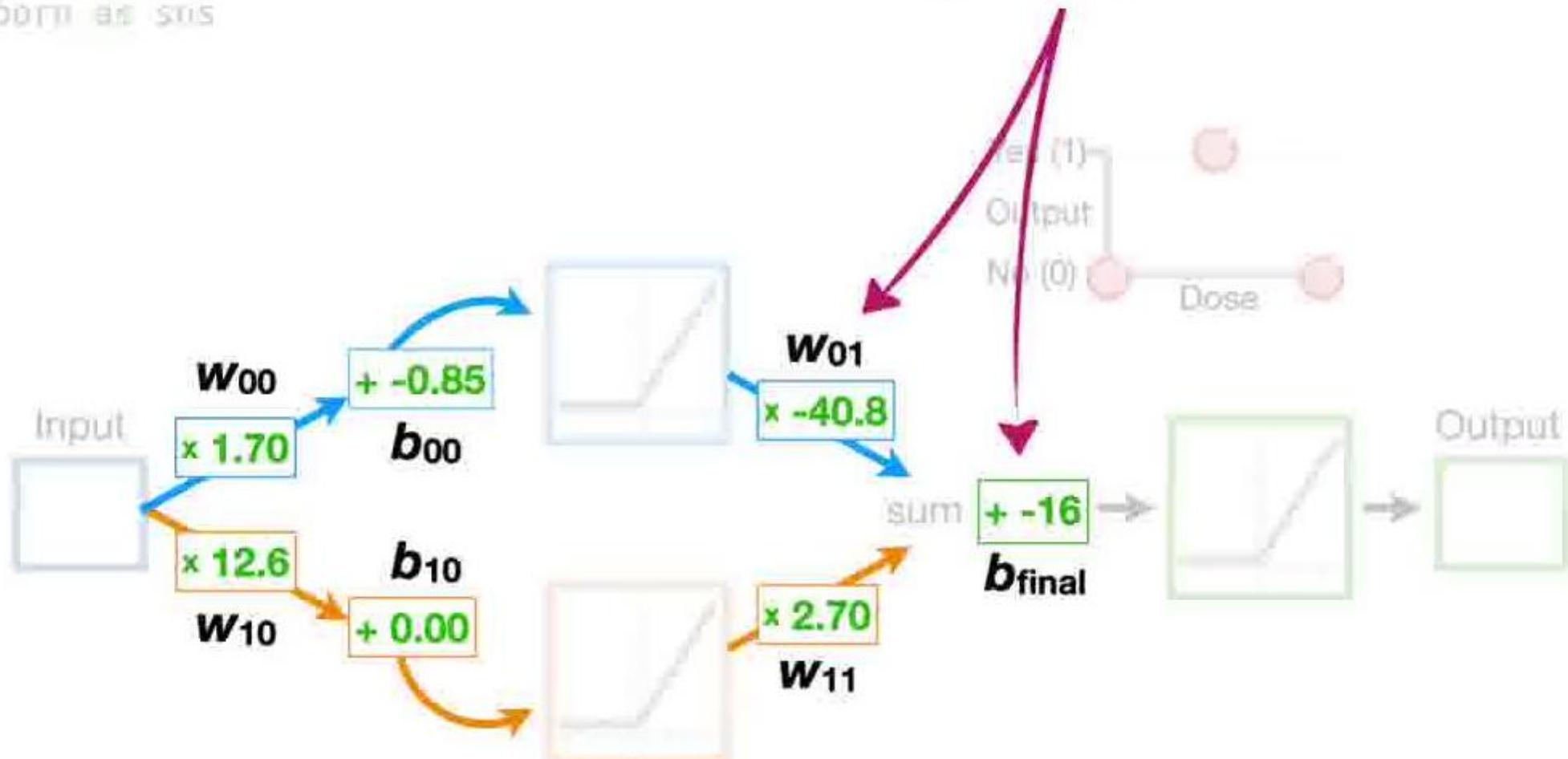
```

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD

import matplotlib.pyplot as plt
import seaborn as sns

```

...and the values for each
weight and bias.



```
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from torch.optim import SGD  
  
import matplotlib.pyplot as plt  
import seaborn as sns
```

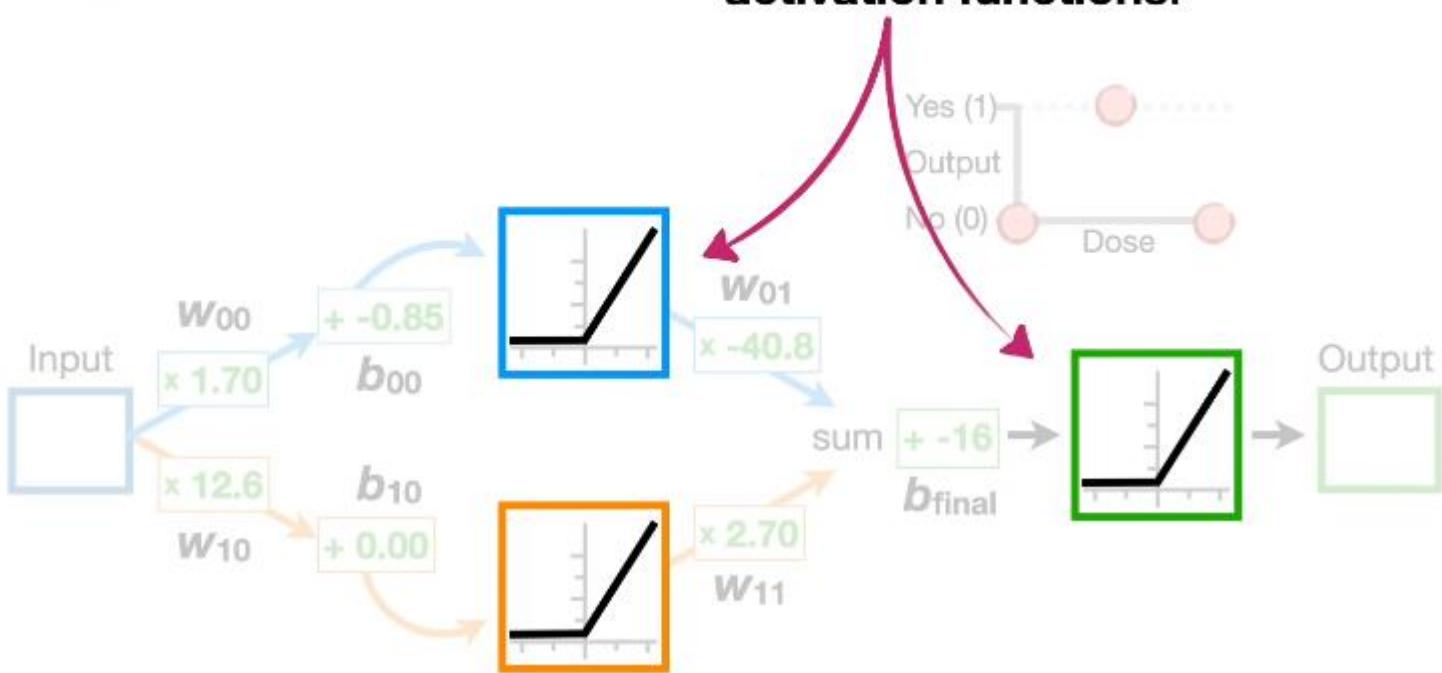
Then we import **torch.nn**, which we will use to make the **weight** and **bias tensors** part of the neural network.



```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD

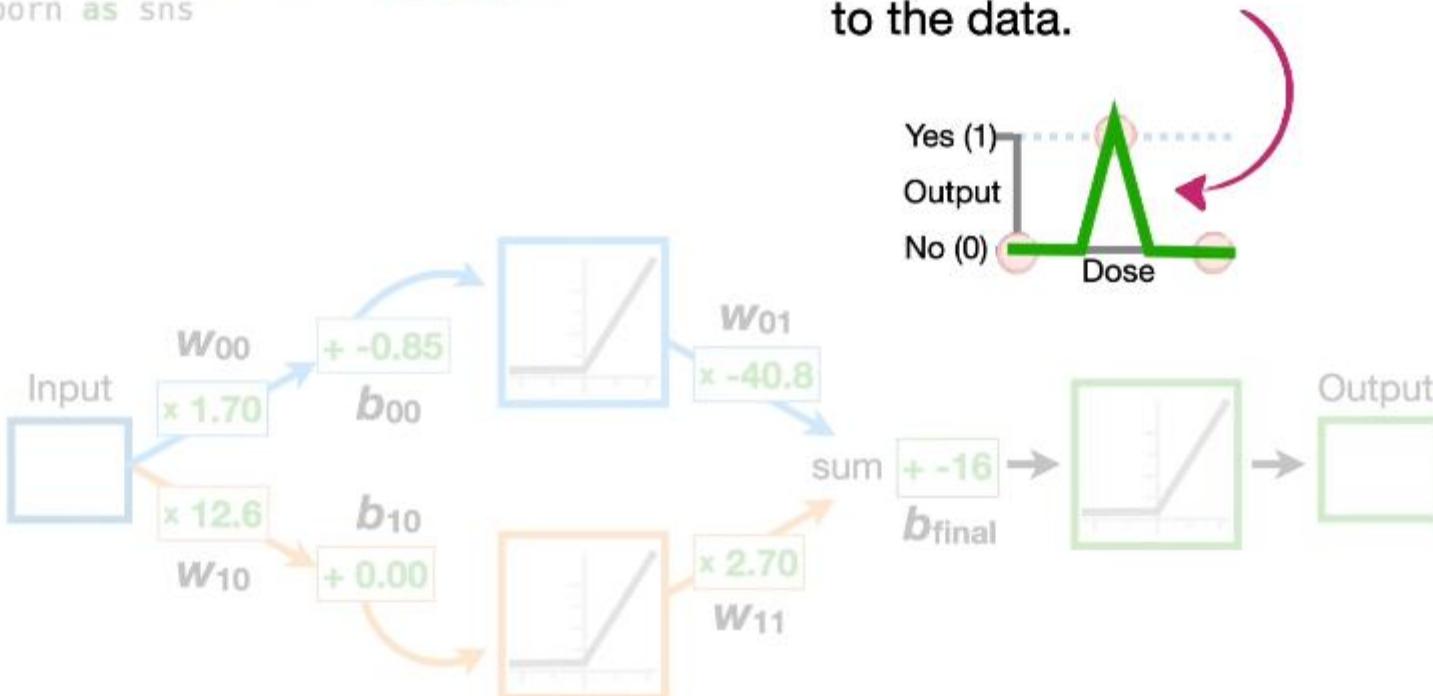
import matplotlib.pyplot as plt
import seaborn as sns
```

Then we import
torch.nn.functional,
which gives us the
activation functions.



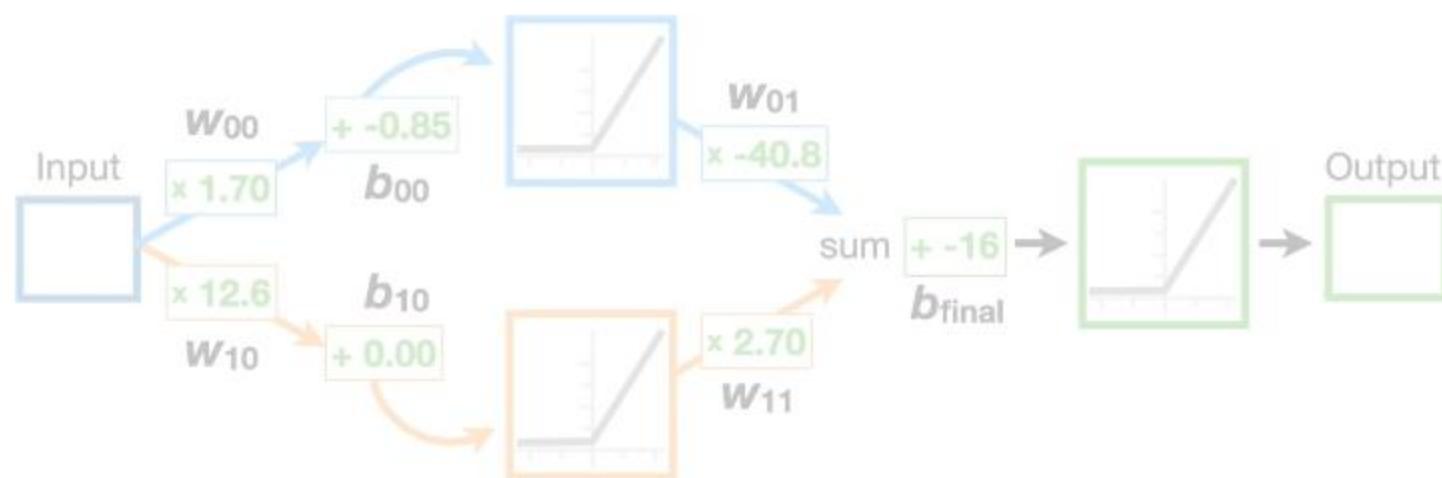
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.optim import SGD
import matplotlib.pyplot as plt
import seaborn as sns
```

Then we import **SGD**, which is short for **Stochastic Gradient Descent**, to fit the neural network to the data.



```
class BasicNN(nn.Module):
```

With **PyTorch**, creating a new neural network means creating a new **class**.



```
class BasicNN(nn.Module):
```

So we start by
creating a new
class...

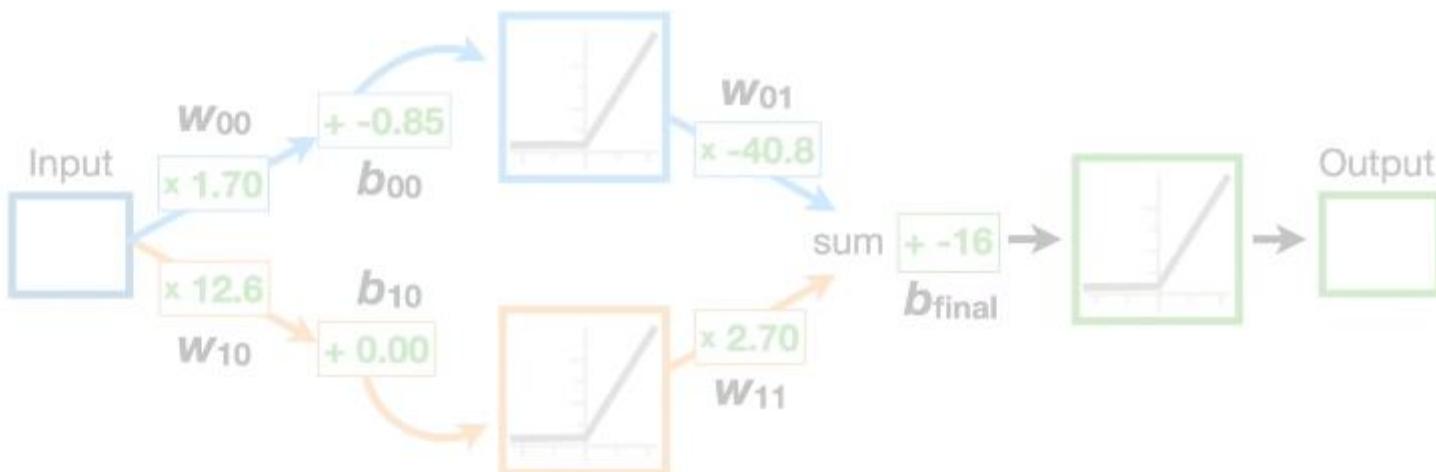
...that, in this
example, we call
BasicNN...

...and **BasicNN** will
inherit from a **PyTorch**
class called **Module**.



```
class BasicNN(nn.Module):  
    def __init__(self):
```

Now we create an
initialization method for the
new class...



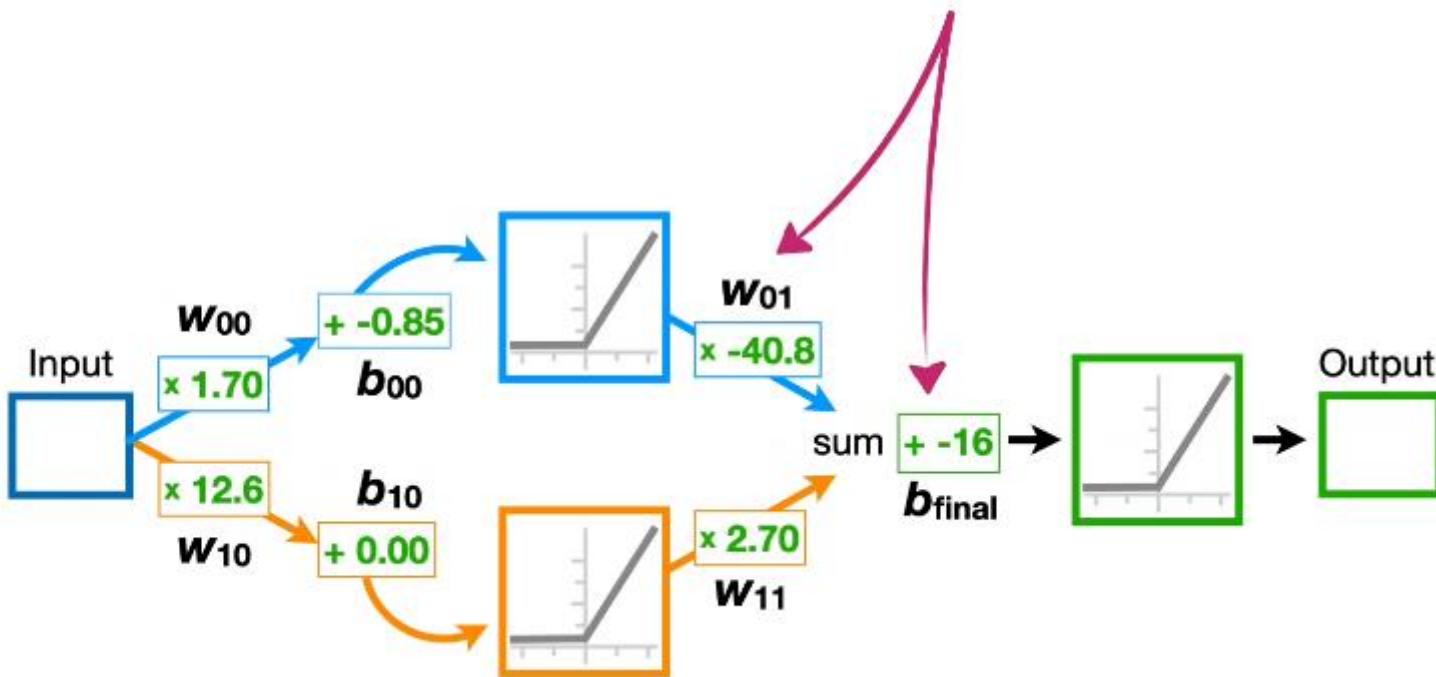
```
class BasicNN(nn.Module):  
  
    def __init__(self):  
        super().__init__()
```

...and the first thing we do
is call the initialization
method for the parent
class, **nn.Module**.



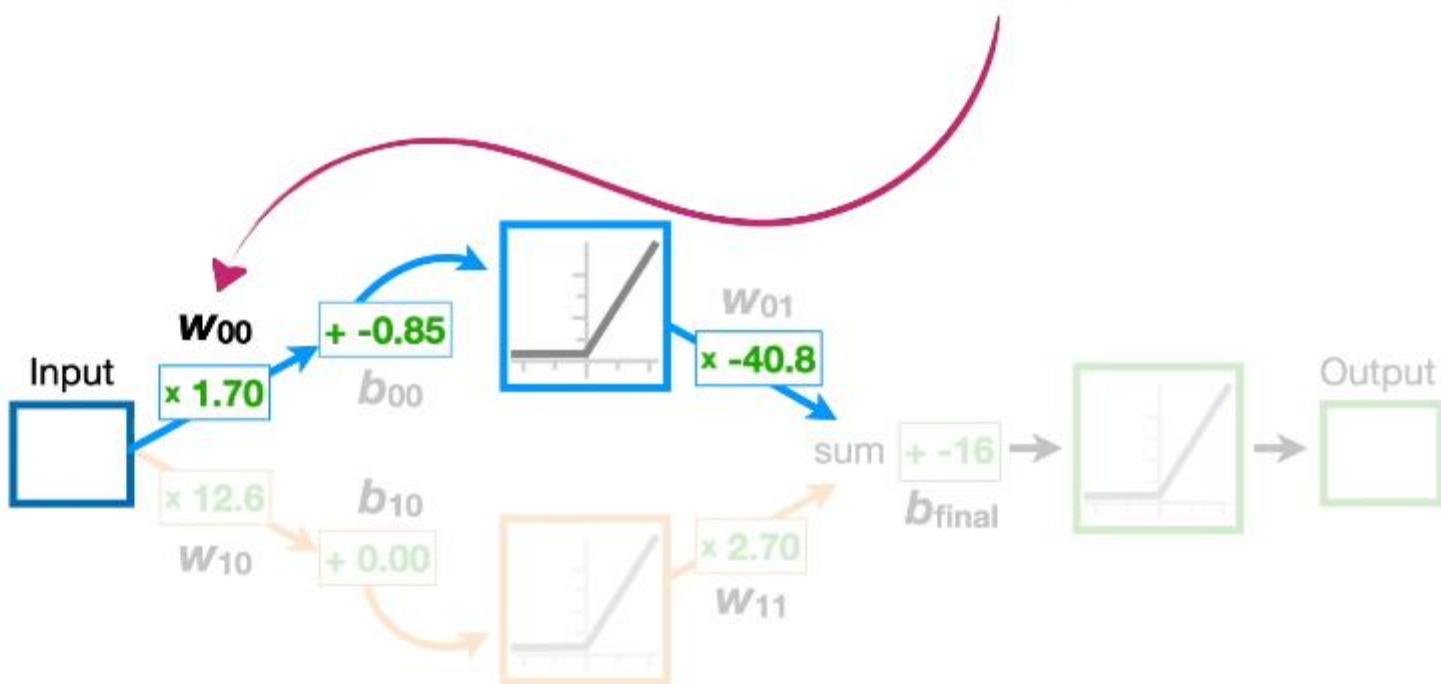
```
class BasicNN(nn.Module):  
  
    def __init__(self):  
        super().__init__()
```

Then we initialize the **weights** and **biases** in our neural network.



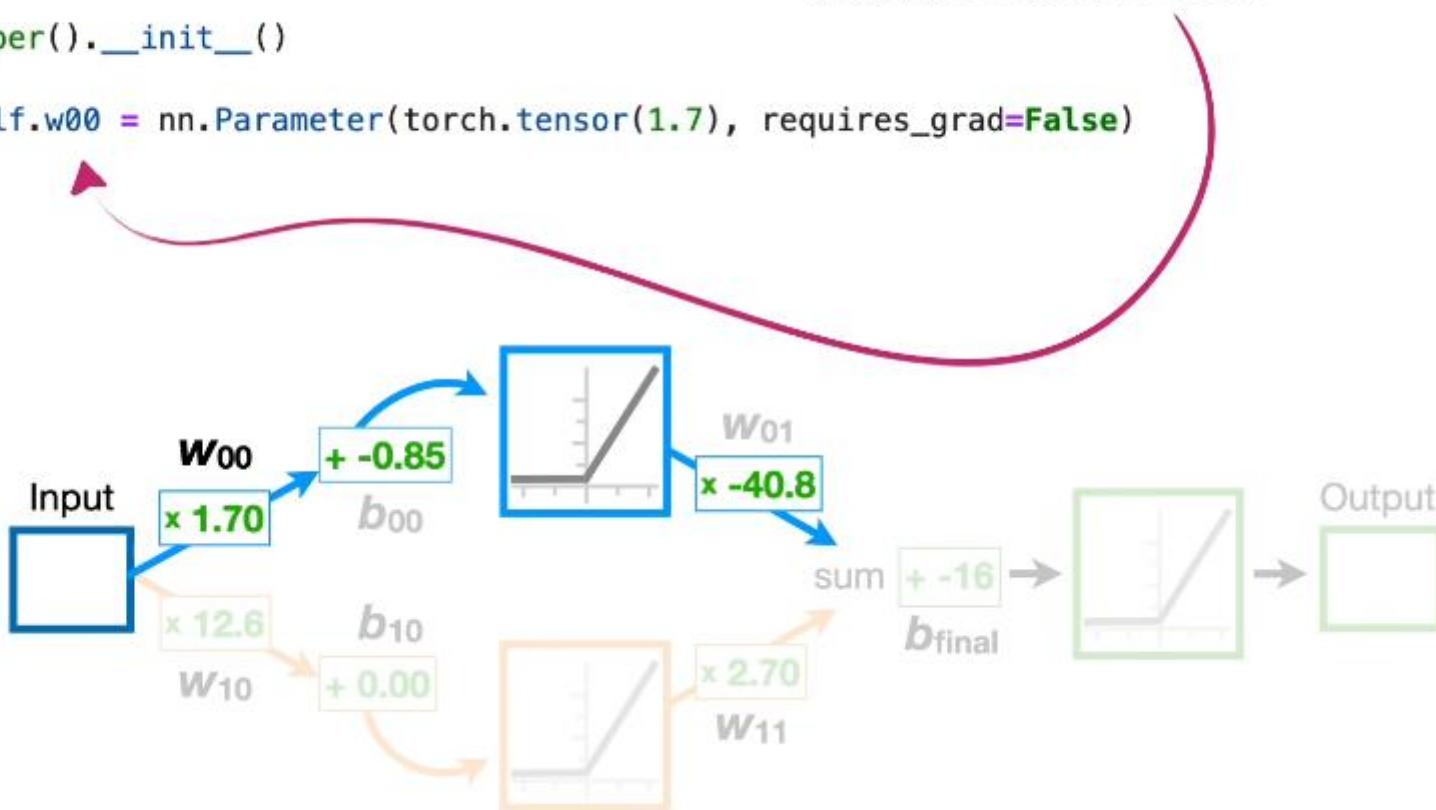
```
class BasicNN(nn.Module):  
    def __init__(self):  
        super().__init__()
```

We'll start with
weight w_{00} , which is
set to **1.70.**



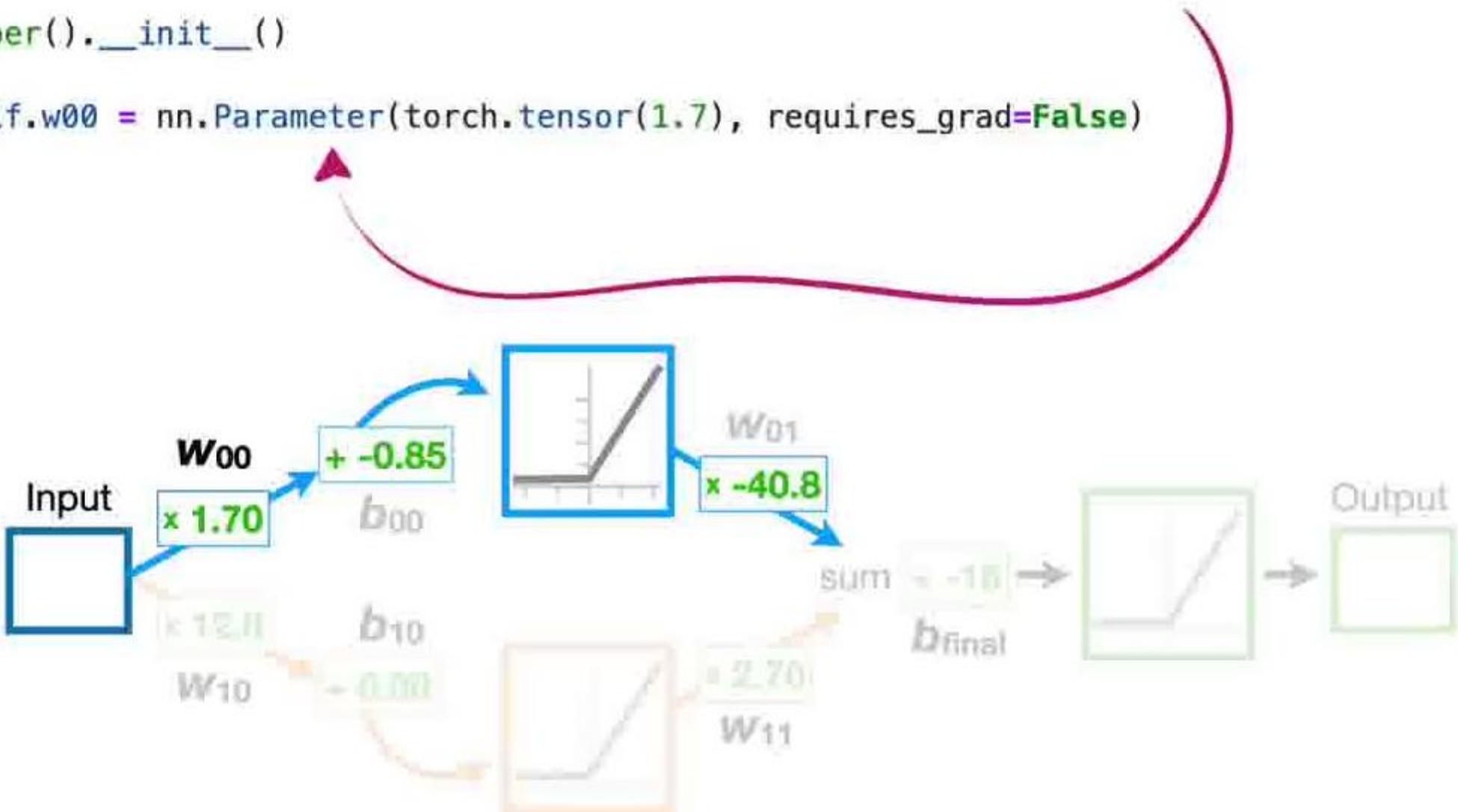
```
class BasicNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

So we create a new variable called **w00**...



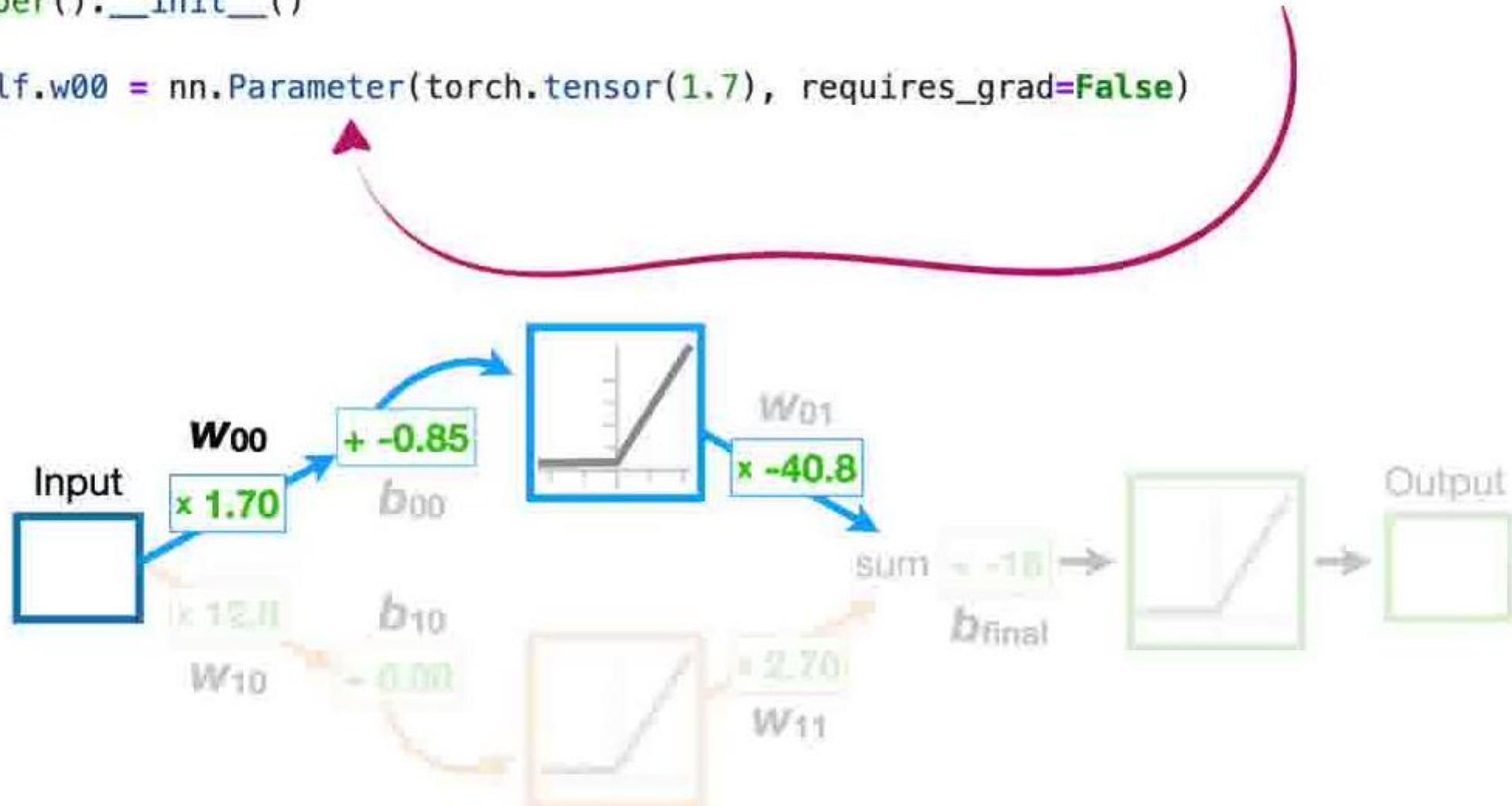
```
class BasicNN(nn.Module):  
    def __init__(self):  
        super().__init__()  
  
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

...and make it a neural network **Parameter**.



```
class BasicNN(nn.Module):  
    def __init__(self):  
        super().__init__()  
  
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

Making this **weight** a parameter
for the neural network gives us
the option to optimize it.



```

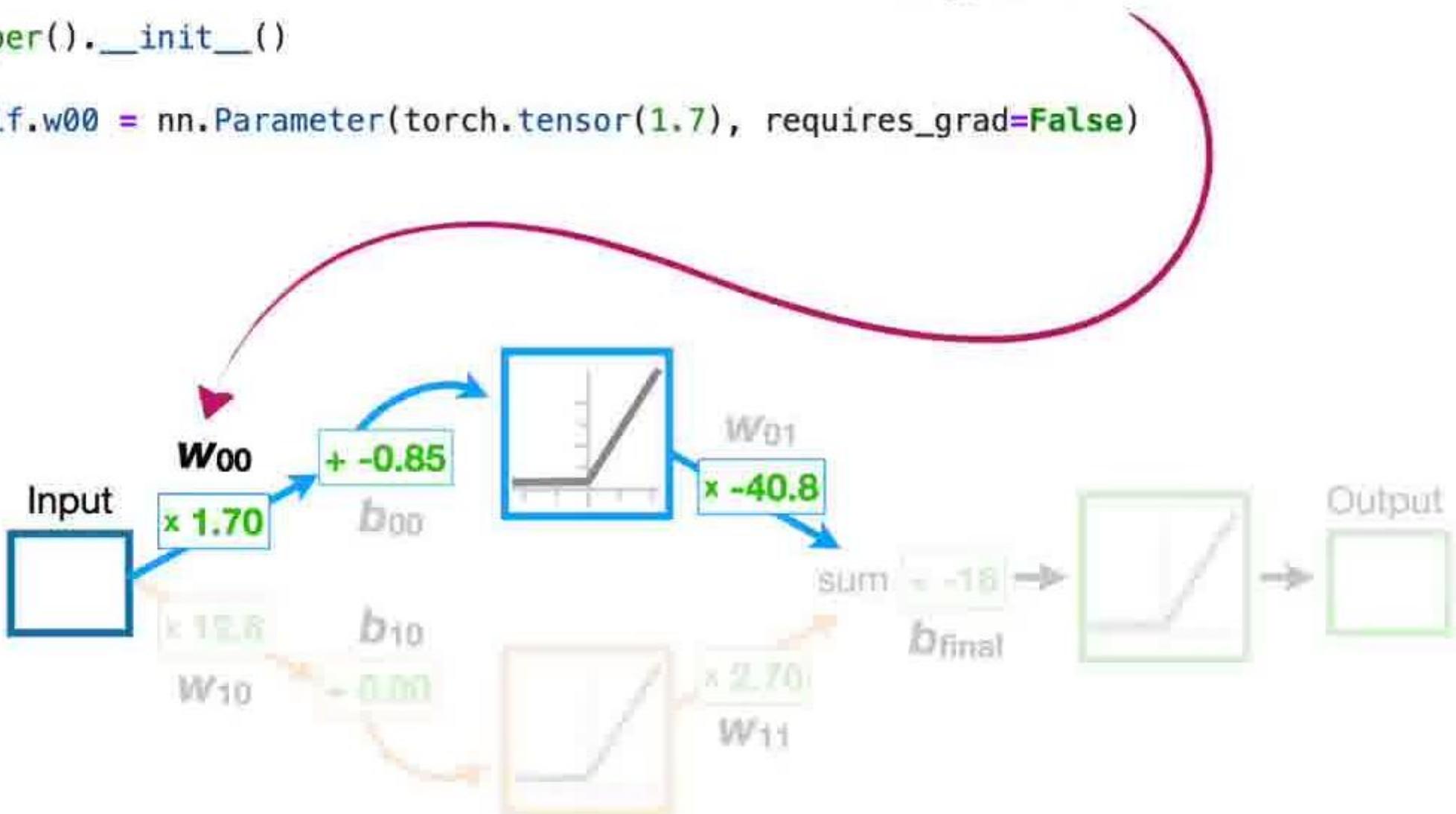
class BasicNN(nn.Module):

    def __init__(self):
        super().__init__()

        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)

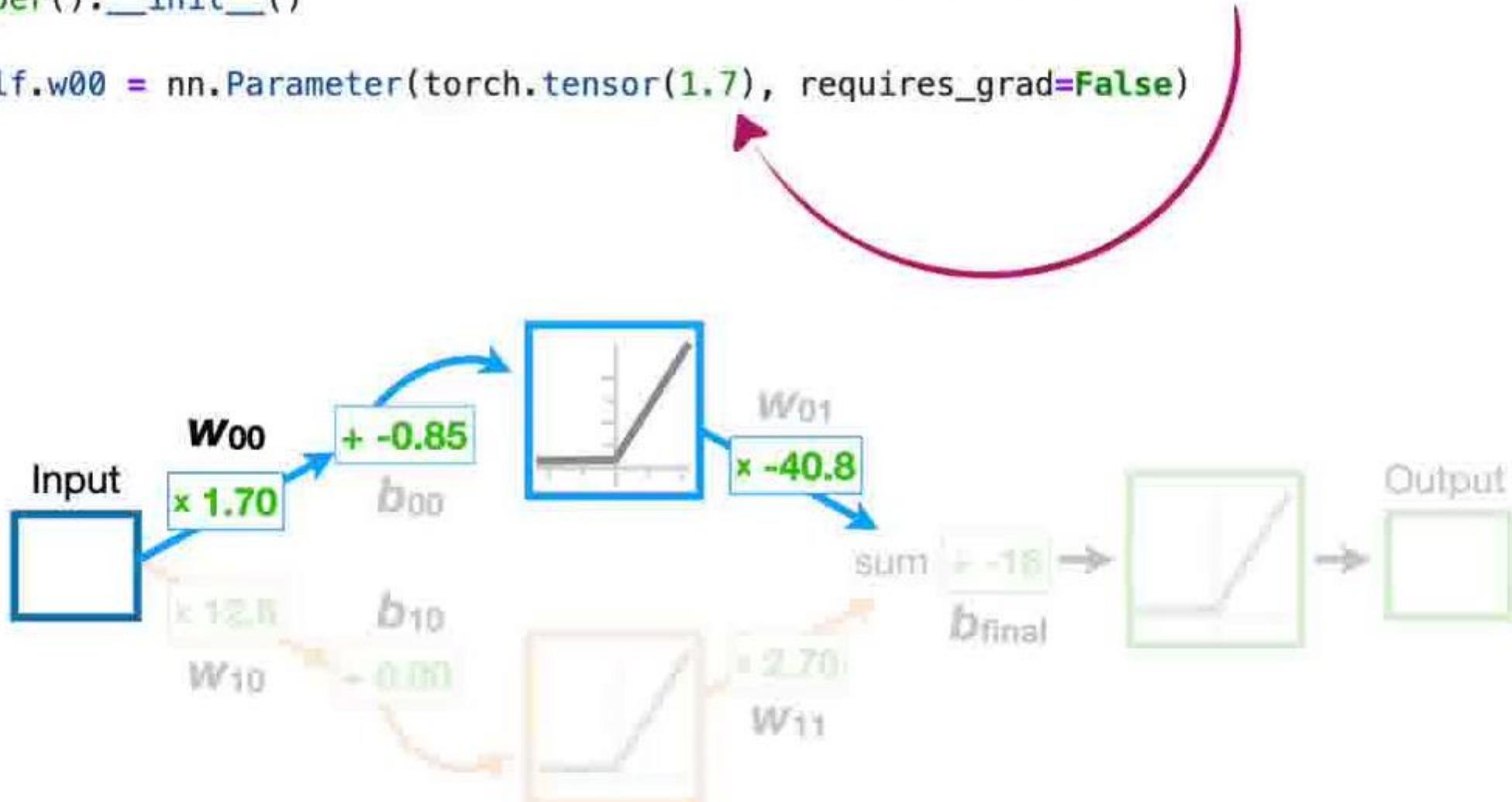
```

Now, since **weight**
w00 is 1.70...



```
class BasicNN(nn.Module):  
    def __init__(self):  
        super().__init__()  
  
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

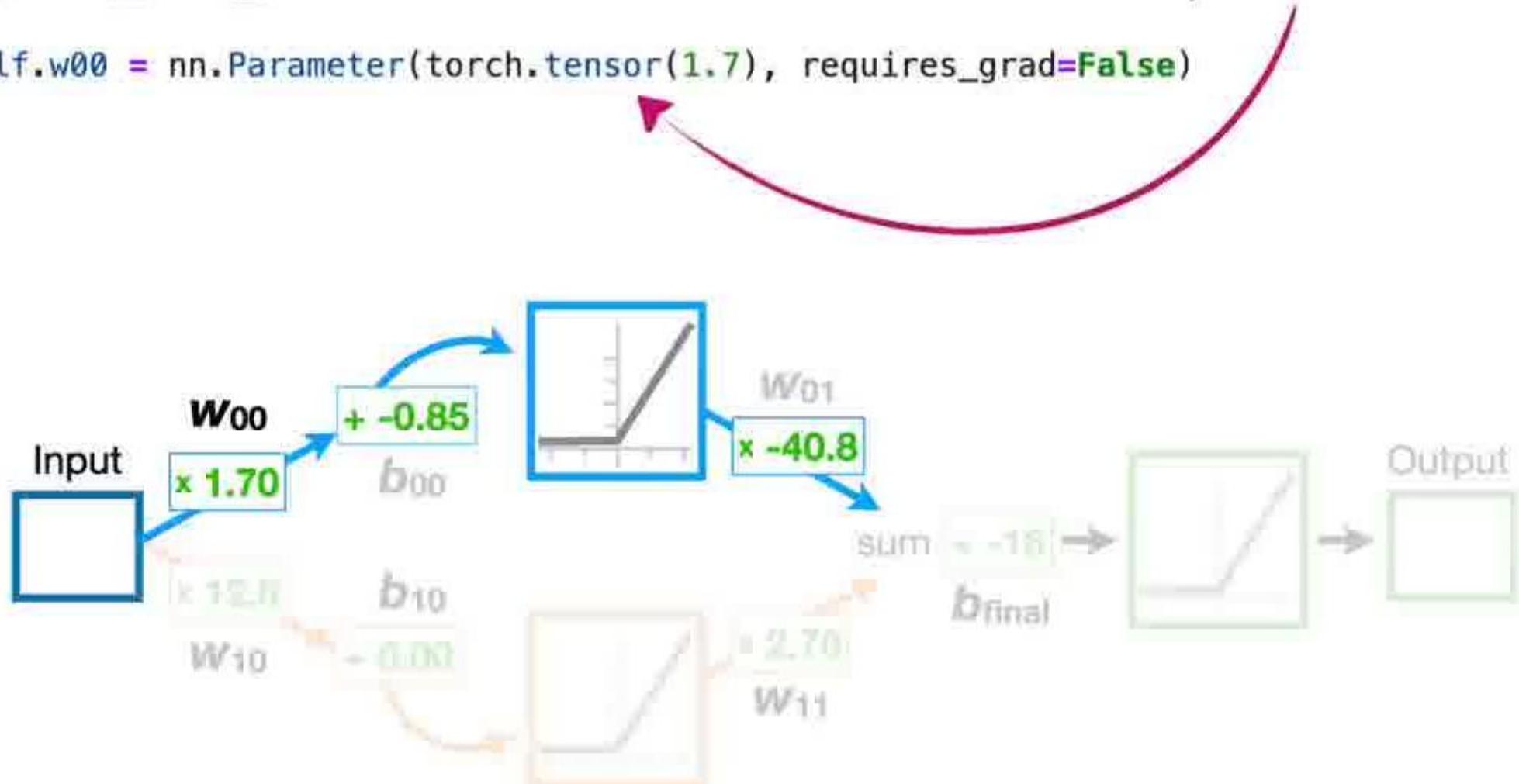
...we initialize the new parameter with a **tensor** set to 1.7.



```
class BasicNN(nn.Module):  
    def __init__(self):  
        super().__init__()
```

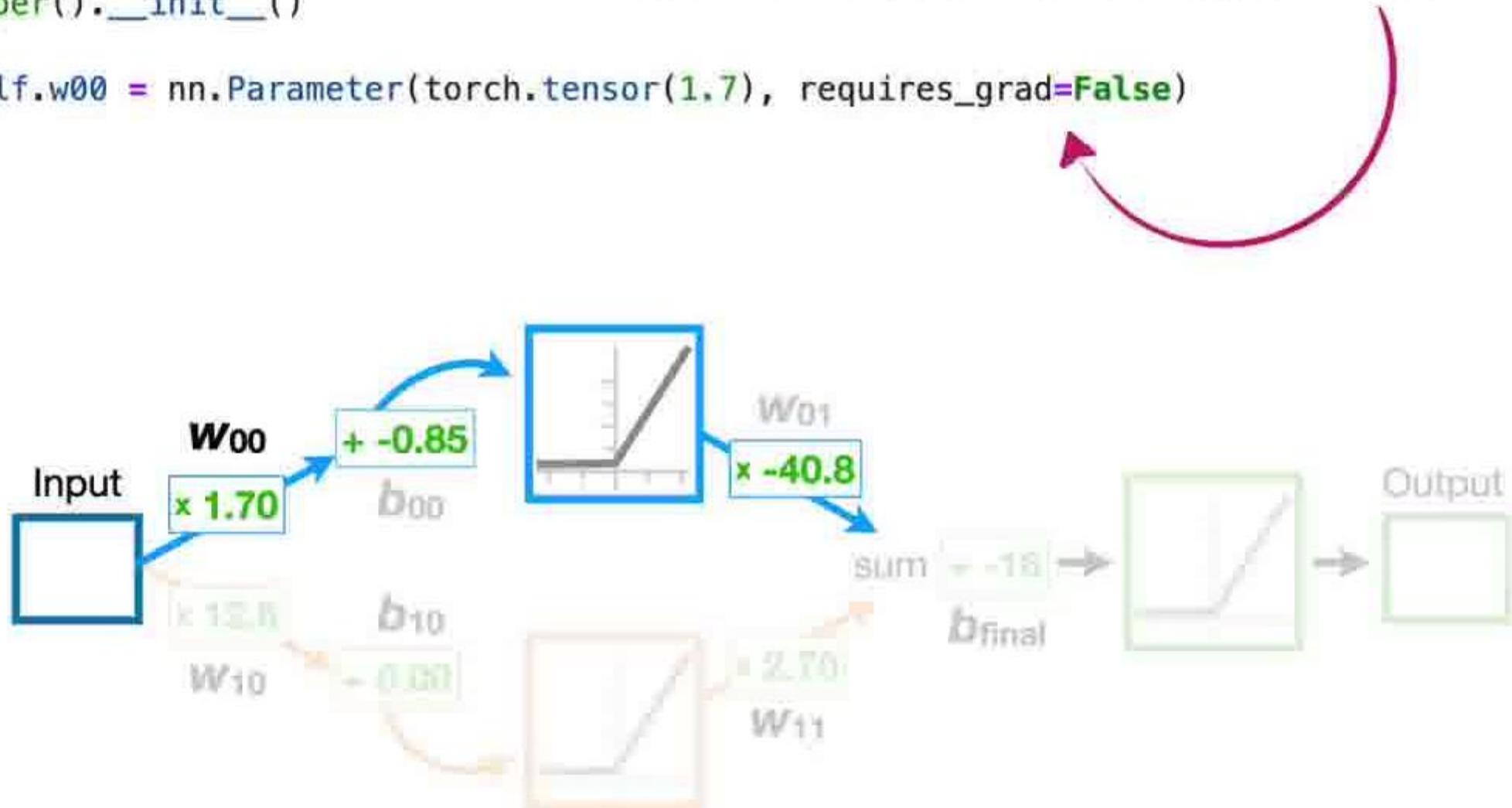
```
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

NOTE: Since this is a **tensor**, the neural network can take advantage of the accelerated arithmetic and automatic differentiation that it provides.



```
class BasicNN(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

Lastly, because we don't need to optimize this **weight**, we'll set **requires_grad**, which is short for **requires gradient**, to **False**.



```

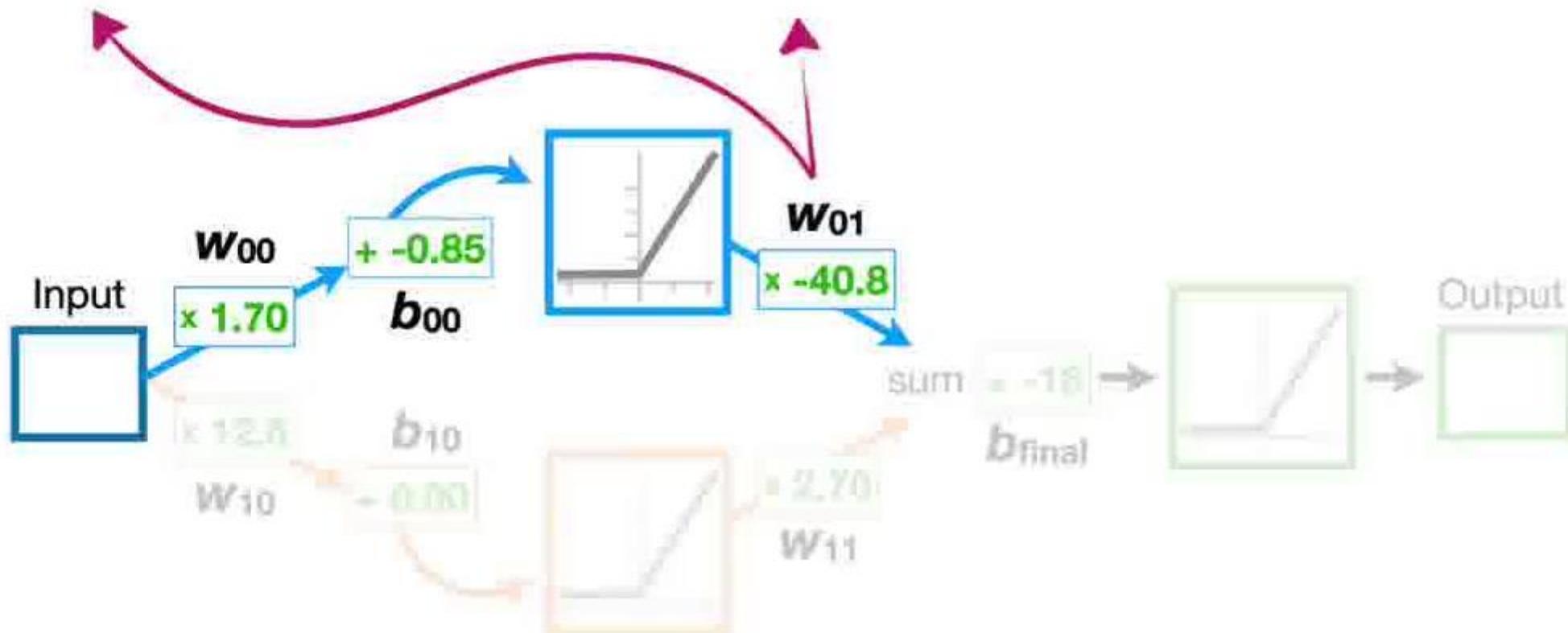
class BasicNN(nn.Module):

    def __init__(self):
        super().__init__()

        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

```

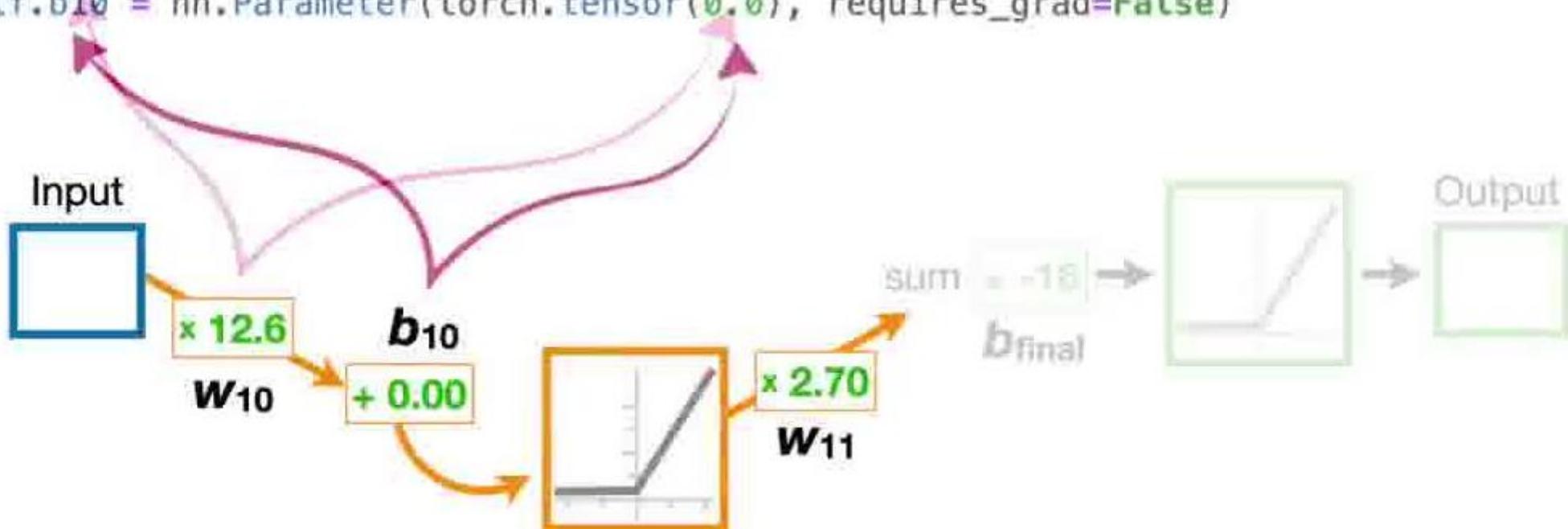
Likewise, we create new variables for the **bias b_{00}** and the **weight w_{01}** .



```
class BasicNN(nn.Module):  
  
    def __init__(self):  
  
        super().__init__()  
  
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)  
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)  
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)
```

```
        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)  
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
```

And then we create variables
for the remaining **weights**
and **biases**.



```

class BasicNN(nn.Module):

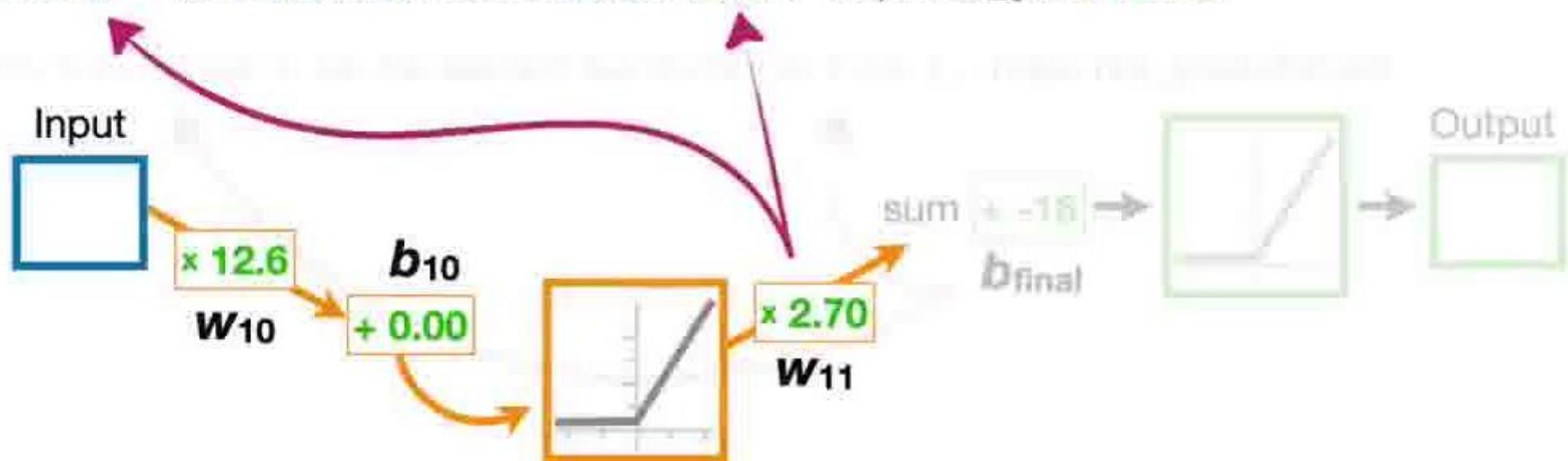
    def __init__(self):
        super().__init__()

        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

```

And then we create variables
for the remaining **weights**
and **biases**.



```

class BasicNN(nn.Module):

    def __init__(self):
        super().__init__()

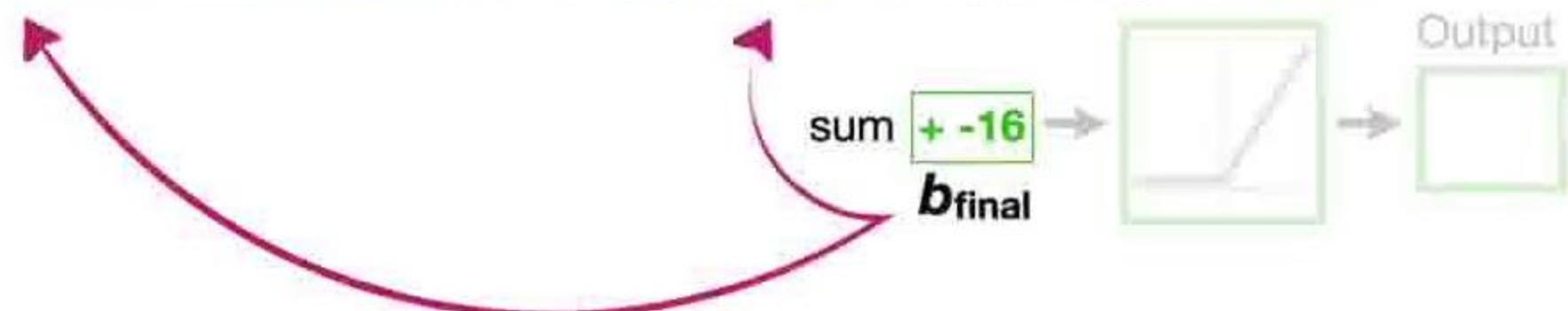
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)

```

And then we create variables
for the remaining **weights**
and **biases**.



```

class BasicNN(nn.Module):

    def __init__(self):
        super().__init__()

        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)

```

Bam! We have created neural network parameters for each **weight** and **bias**.



```
class BasicNN(nn.Module):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

```
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
```

```
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)
```

```
        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
```

```
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
```

```
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)
```

```
        self.final_bias = nn.Parameter(torch.tensor(-16.1), requires_grad=False)
```



Now we need to connect them to the input...

```
class BasicNN(nn.Module):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

```
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
```

```
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)
```

```
        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
```

```
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
```

```
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)
```

```
        self.final_bias = nn.Parameter(torch.tensor(-16.1), requires_grad=False)
```

...the activation functions...



```

class BasicNN(nn.Module):
    def __init__(self):
        super().__init__()

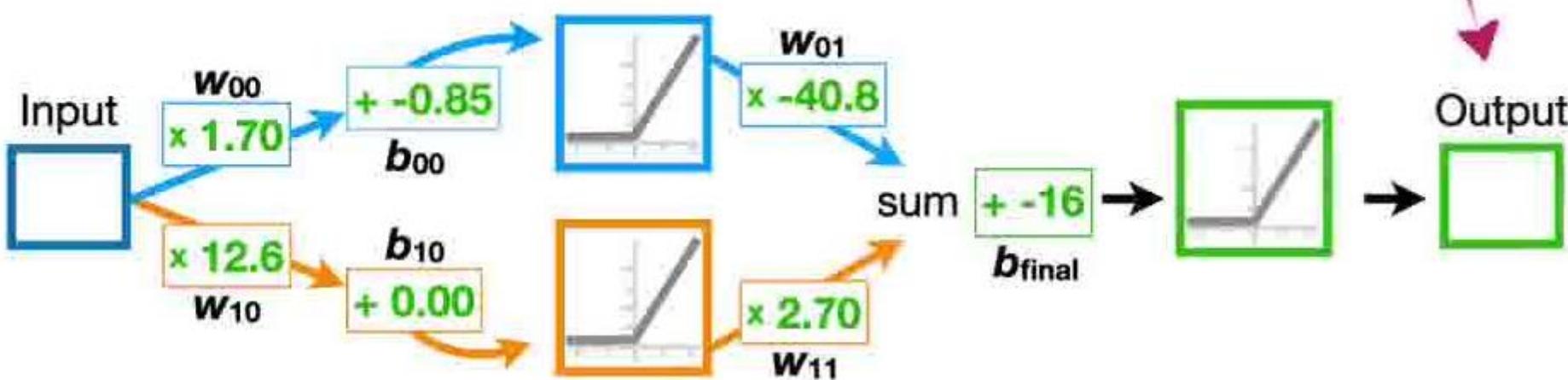
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-0.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        self.final_bias = nn.Parameter(torch.tensor(-16.1), requires_grad=False)

```

...and, ultimately,
to the output.



```
class BasicNN(nn.Module):
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
```

```
        self.b00 = nn.Parameter(torch.tensor(-0.05), requires_grad=False)
```

```
        self.w01 = nn.Parameter(torch.tensor(-0.0), requires_grad=False)
```

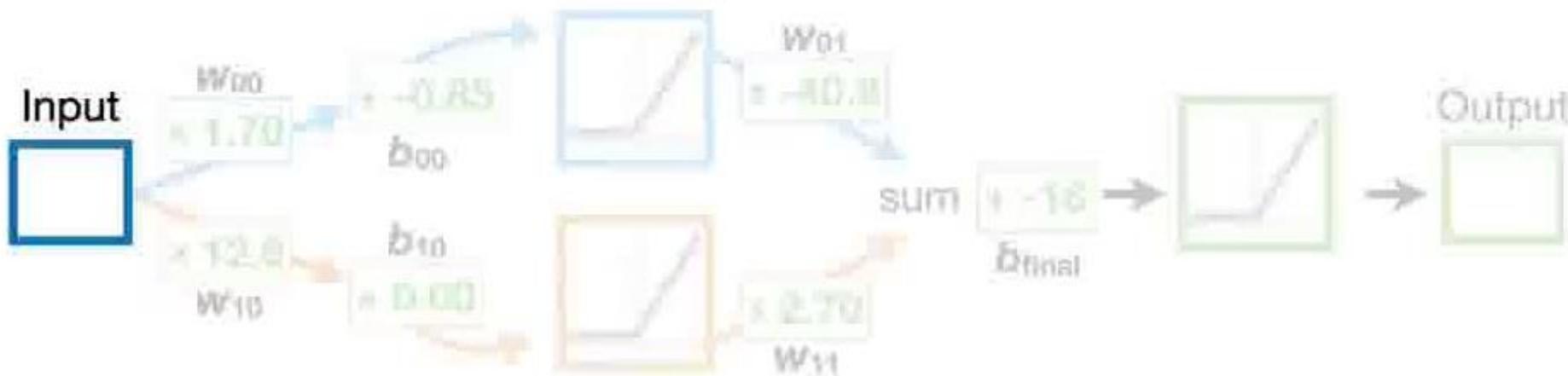
```
        self.w10 = nn.Parameter(torch.tensor(12.0), requires_grad=False)
```

```
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
```

```
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)
```

```
        self.final_bias = nn.Parameter(torch.tensor(-16.1), requires_grad=False)
```

In other words, we need a way to make a **forward pass** through the neural network that uses the **weights** and **biases** that we just initialized.



```
class BasicNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-0.5), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        self.final_bias = nn.Parameter(torch.tensor(-16.1), requires_grad=False)

    def forward(self, input):
```

So we do that by creating a second method inside **BasicNN** called **forward()**.

```
class BasicNN(nn.Module):
    def __init__(self):
        super().__init__()

        self.w00 = nn.Parameter(torch.tensor(1, 7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40, 8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12, 6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0, 6), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2, 7), requires_grad=False)

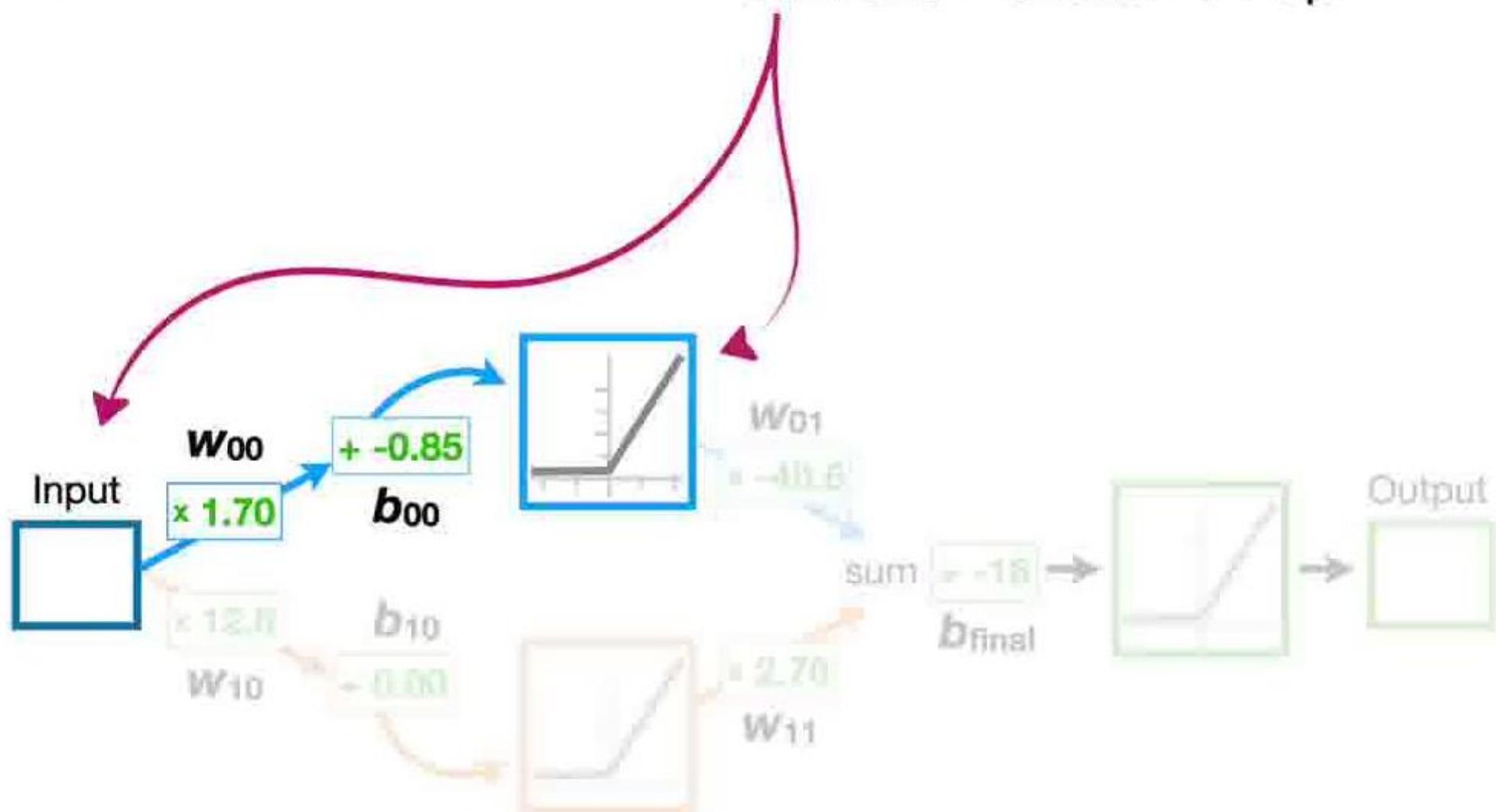
        self.final_bias = nn.Parameter(torch.tensor(-16, 1), requires_grad=False)

    def forward(self, input):
```

So we can see what's going on, let's move the code for **forward()** to the top of the screen.

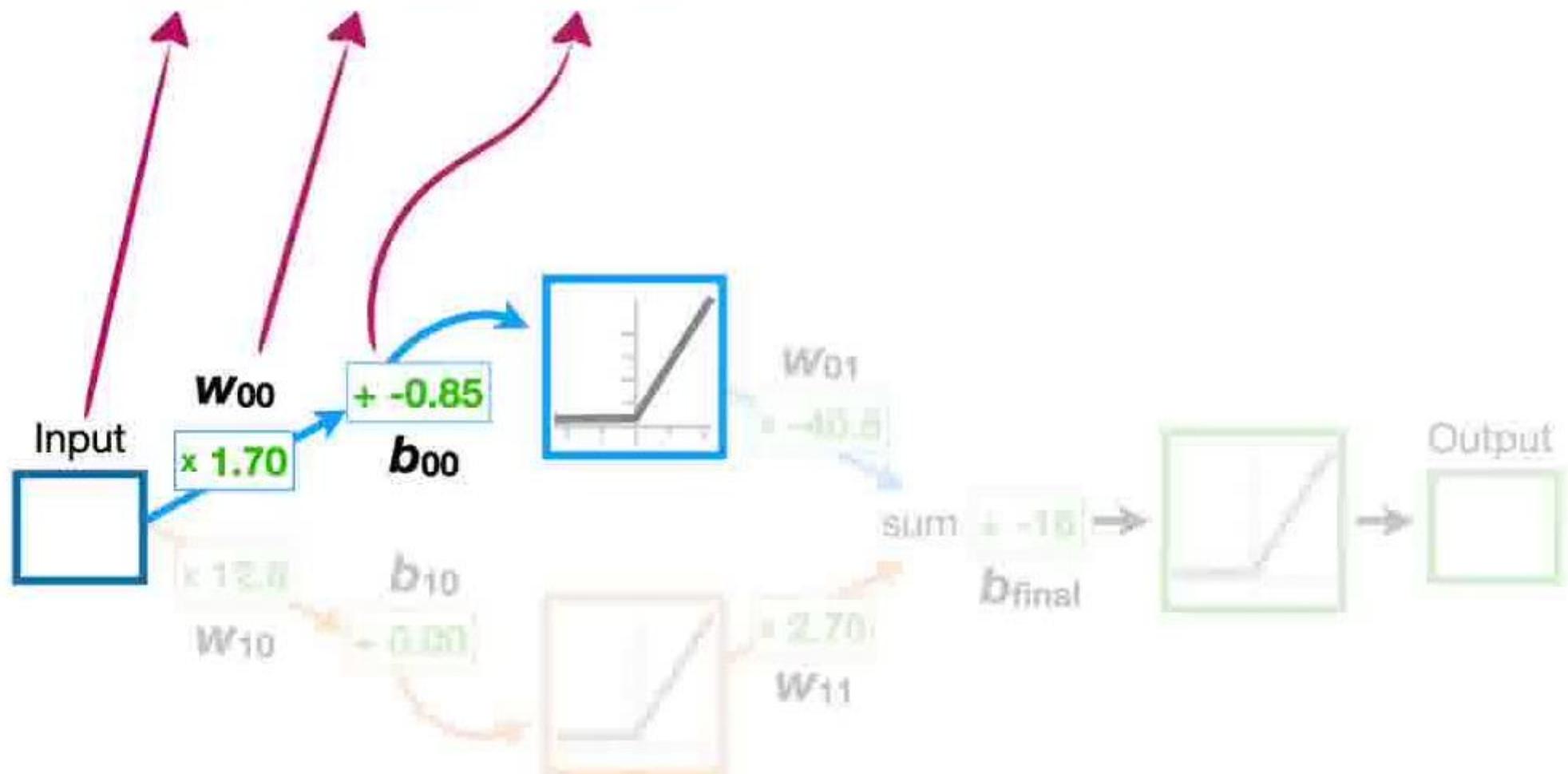
```
def forward(self, input):
```

Now, the first thing we want to do is connect the input to the **activation function** on top.



```
def forward(self, input):  
    input_to_top_relu = input * self.w00 + self.b00
```

So we create a new variable,
input_to_top_relu, that is
equal to...



Then we pass `input_to_top_relu` to the **ReLU** activation function with `F.relu()`.

```
def forward(self, input):  
  
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)
```

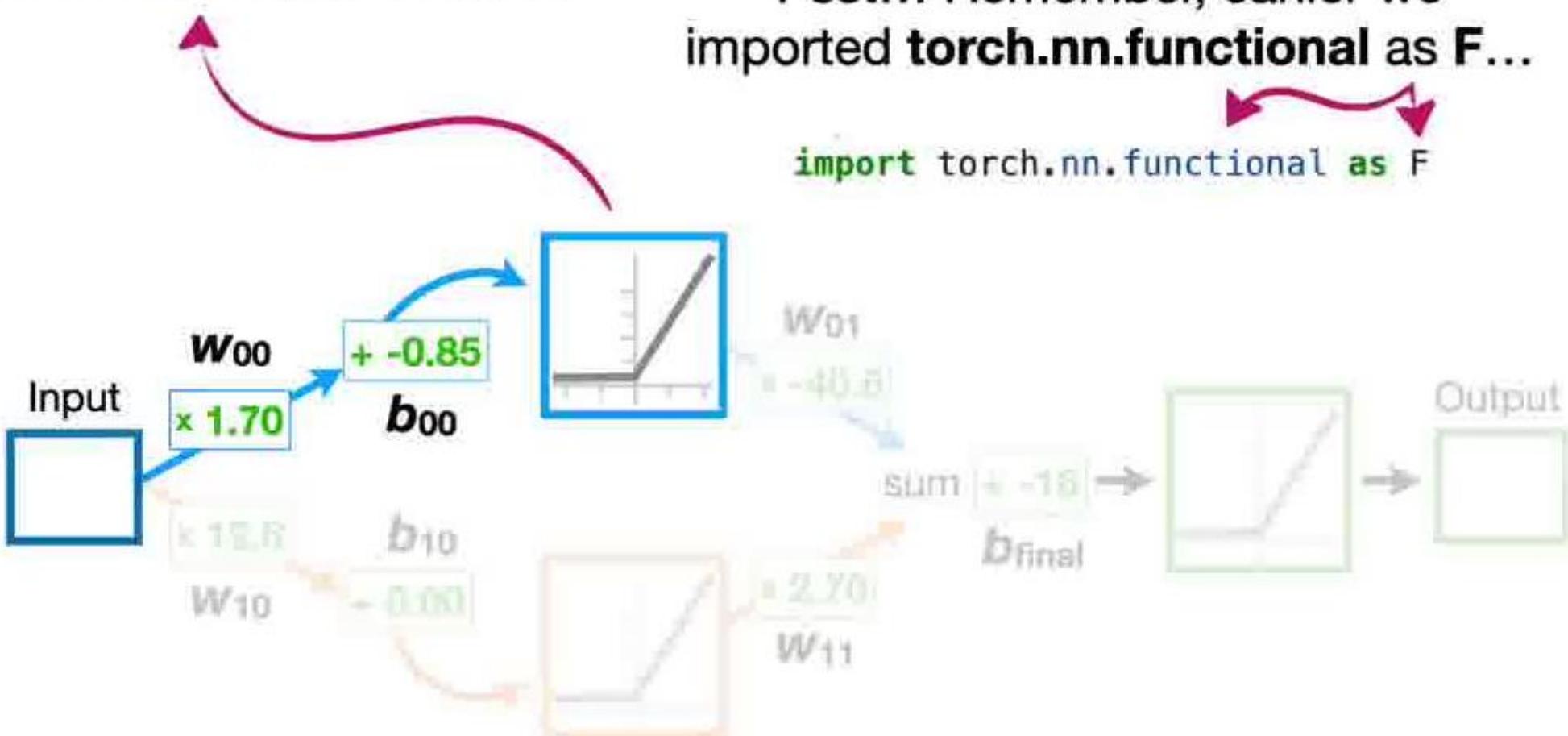


```
def forward(self, input):
```

```
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)
```

Then we pass `input_to_top_relu` to the **ReLU** activation function with `F.relu()`.

Psst... Remember, earlier we imported `torch.nn.functional` as `F`...



```
def forward(self, input):
```

```
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)
```

Then we pass `input_to_top_relu` to the **ReLU** activation function with `F.relu()`.

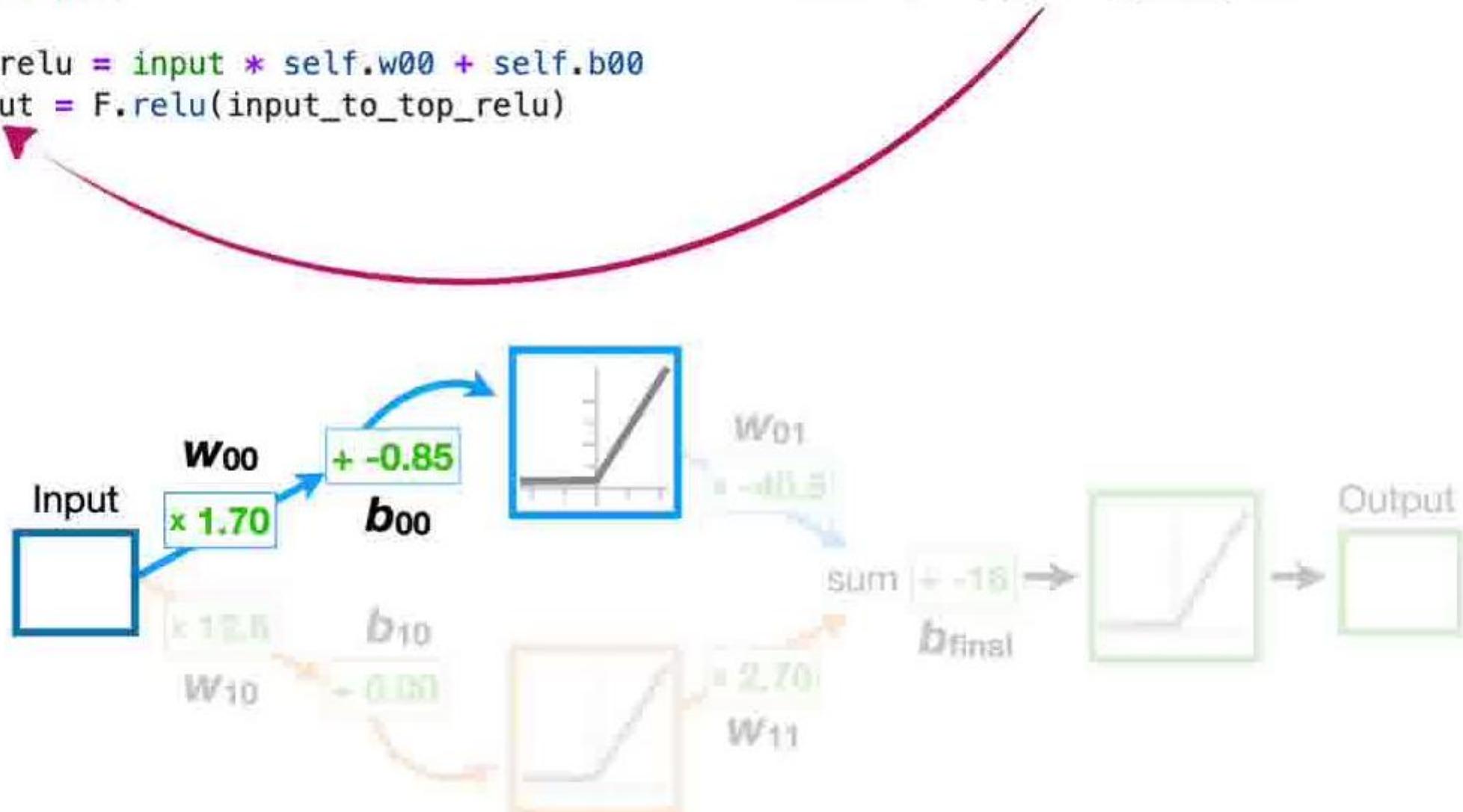
...so that is where the `relu()` comes from.



Then we save the output of the ReLU in `top_relu_output`.

```
def forward(self, input):
```

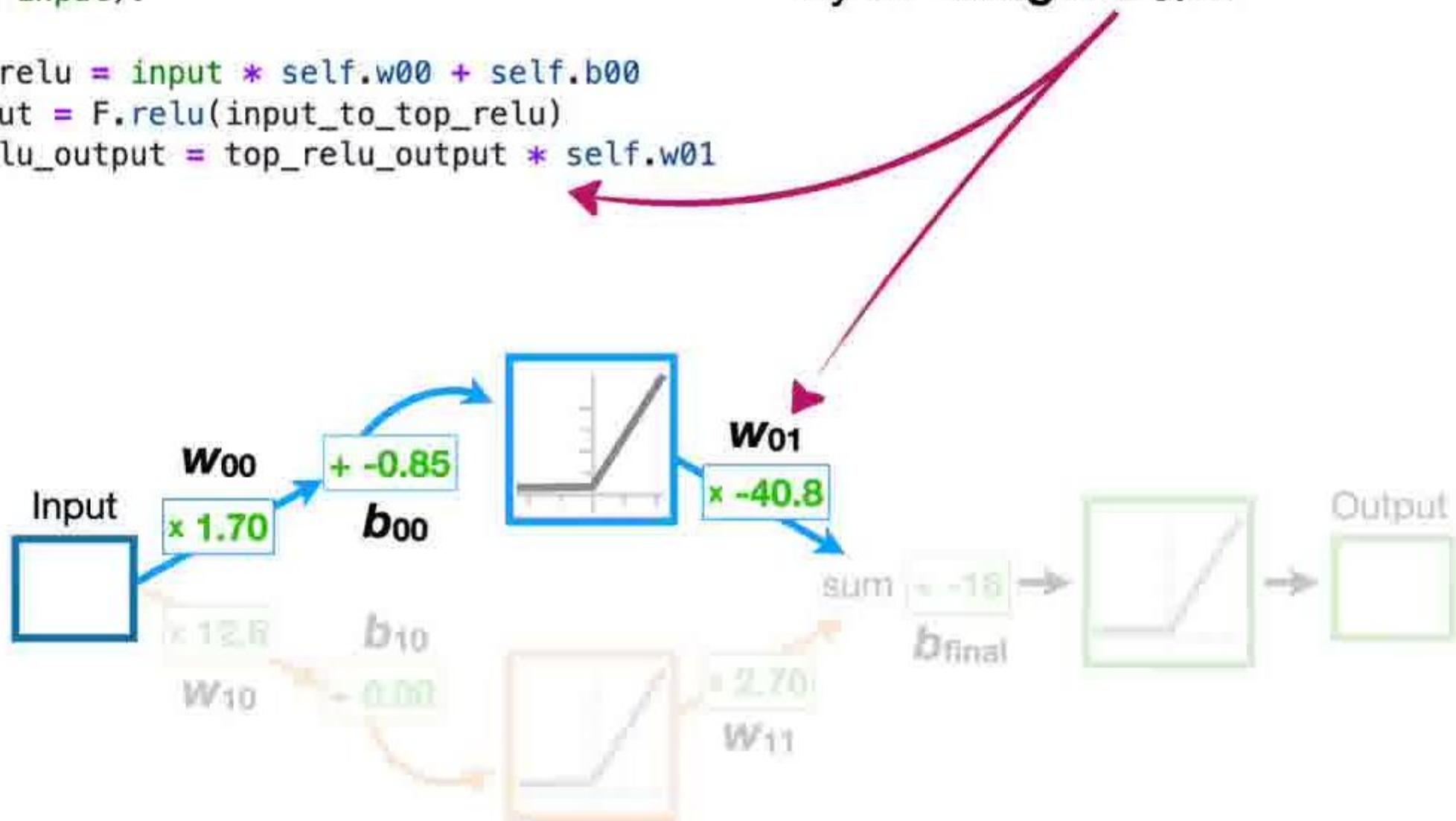
```
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)
```



Now we scale top_relu_output by the weight **w₀₁**...

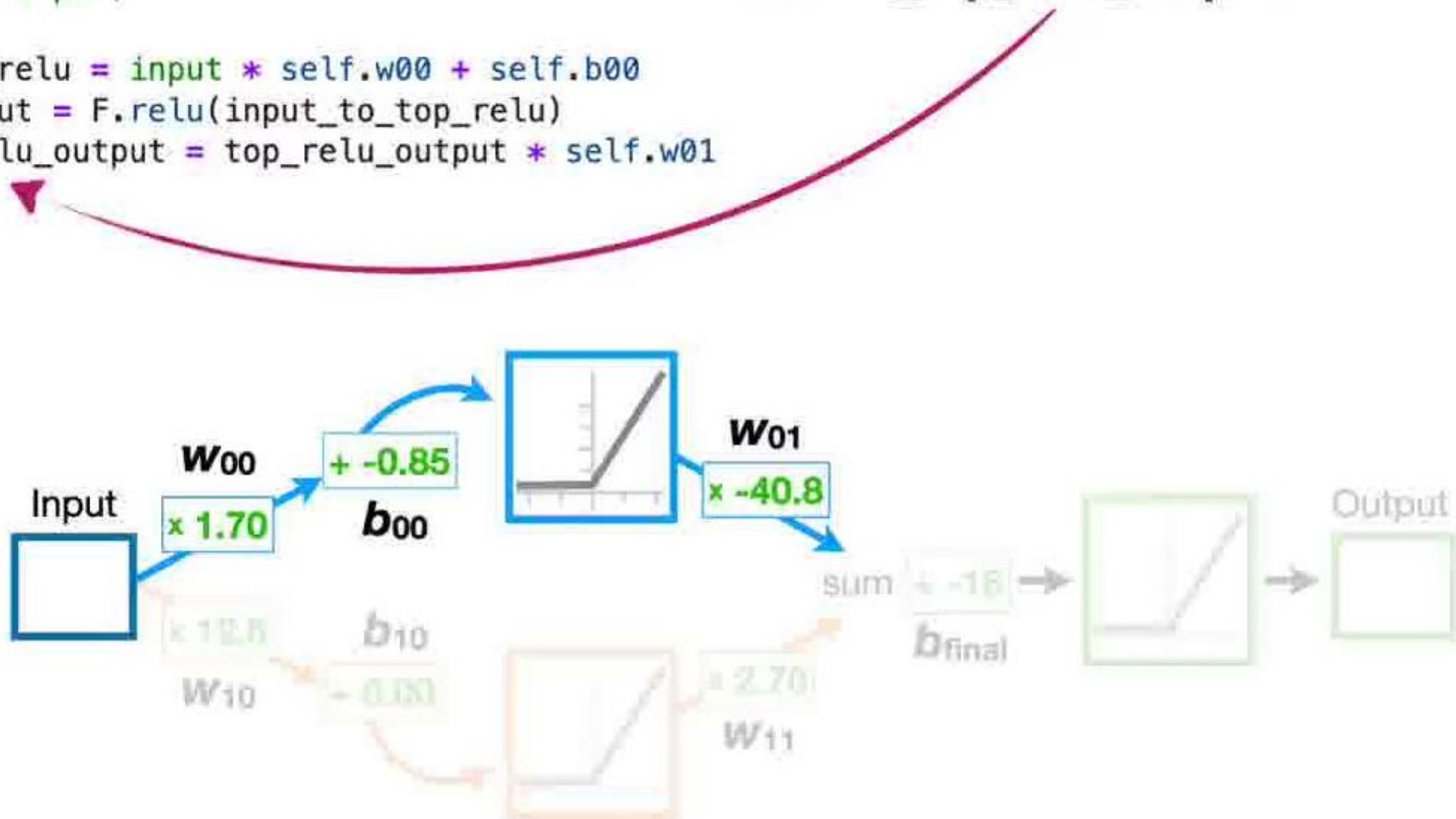
```
def forward(self, input):
```

```
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)  
    scaled_top_relu_output = top_relu_output * self.w01
```



...and save the result in
scaled_top_relu_output.

```
def forward(self, input):  
  
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)  
    scaled_top_relu_output = top_relu_output * self.w01
```

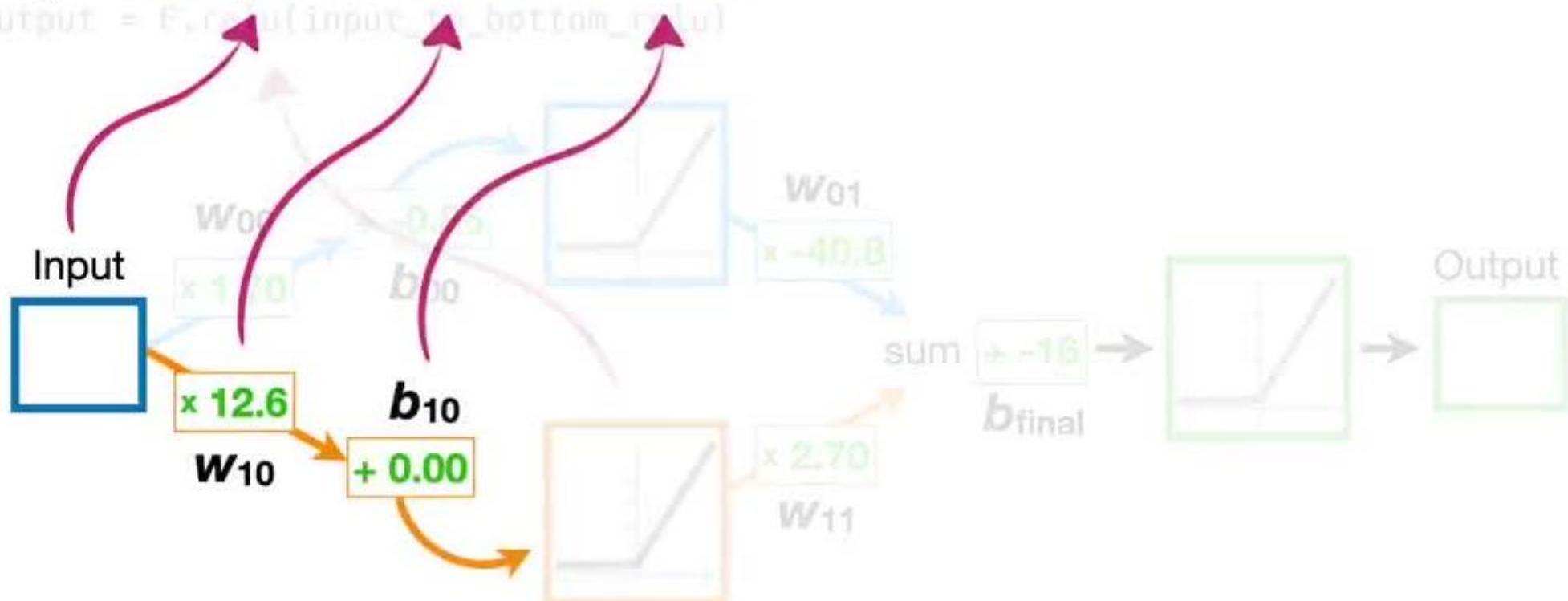


```
def forward(self, input):
```

```
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)  
    scaled_top_relu_output = top_relu_output * self.w01
```

```
    input_to_bottom_relu = input * self.w10 + self.b10  
    bottom_relu_output = F.relu(input_to_bottom_relu)
```

Likewise, we connect the input to the bottom **ReLU** and scale the activation function's output.



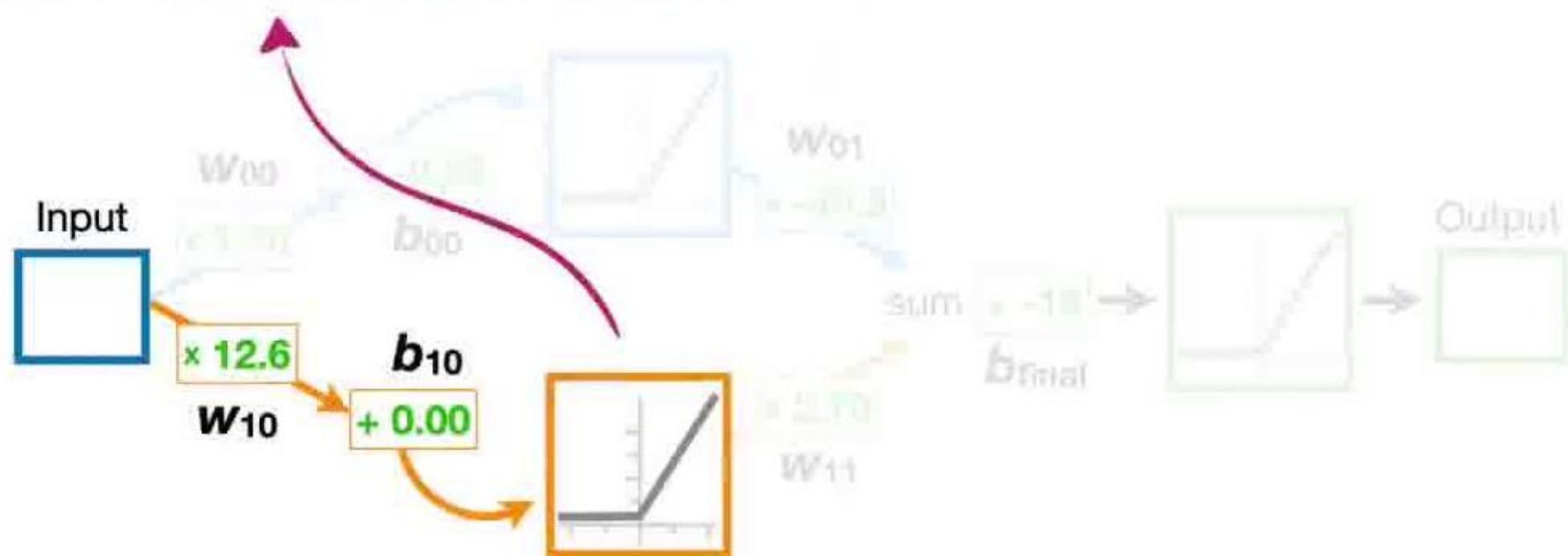
```

def forward(self, input):
    input_to_top_relu = input * self.w00 * self.b00
    top_relu_output = F.relu(input_to_top_relu)
    scaled_top_relu_output = top_relu_output * self.w01

    input_to_bottom_relu = input * self.w10 + self.b10
    bottom_relu_output = F.relu(input_to_bottom_relu)

```

Likewise, we connect the input to the bottom **ReLU** and scale the activation function's output.



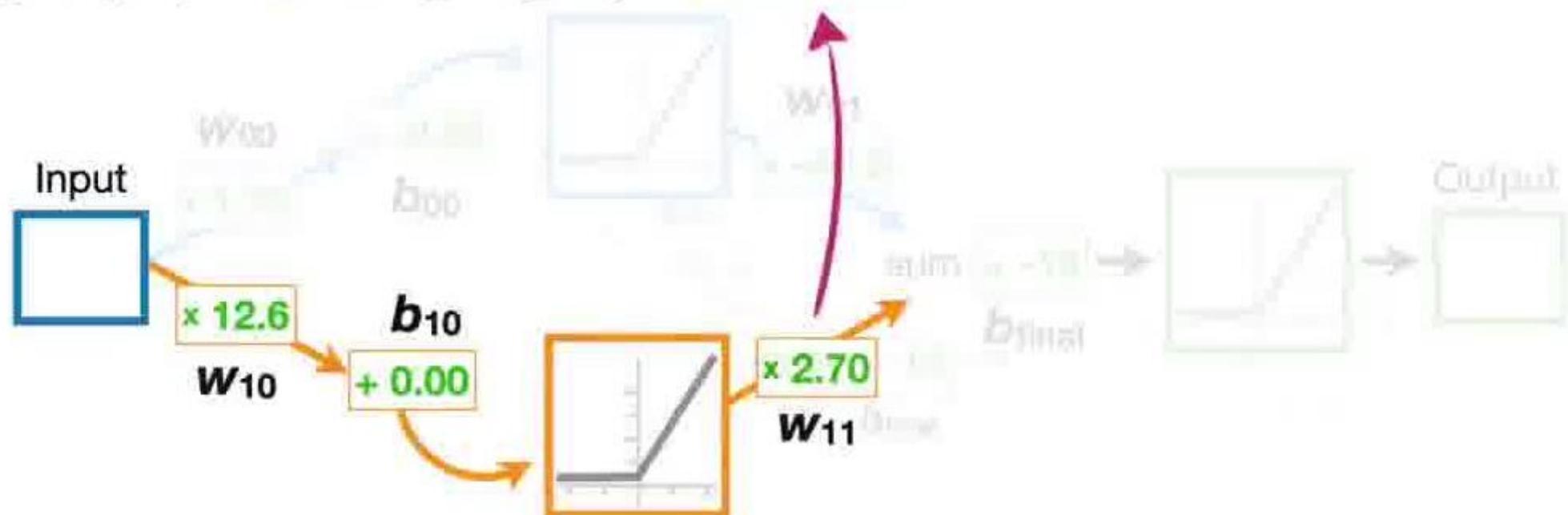
```

def forward(self, input):
    input_to_top_relu = input * self.w00 * self.b00
    top_relu_output = F.relu(input_to_top_relu)
    scaled_top_relu_output = top_relu_output * self.w01

    input_to_bottom_relu = input * self.w10 + self.b10
    bottom_relu_output = F.relu(input_to_bottom_relu)
    scaled_bottom_relu_output = bottom_relu_output * self.w11

```

Likewise, we connect the input to the bottom **ReLU** and scale the activation function's output.

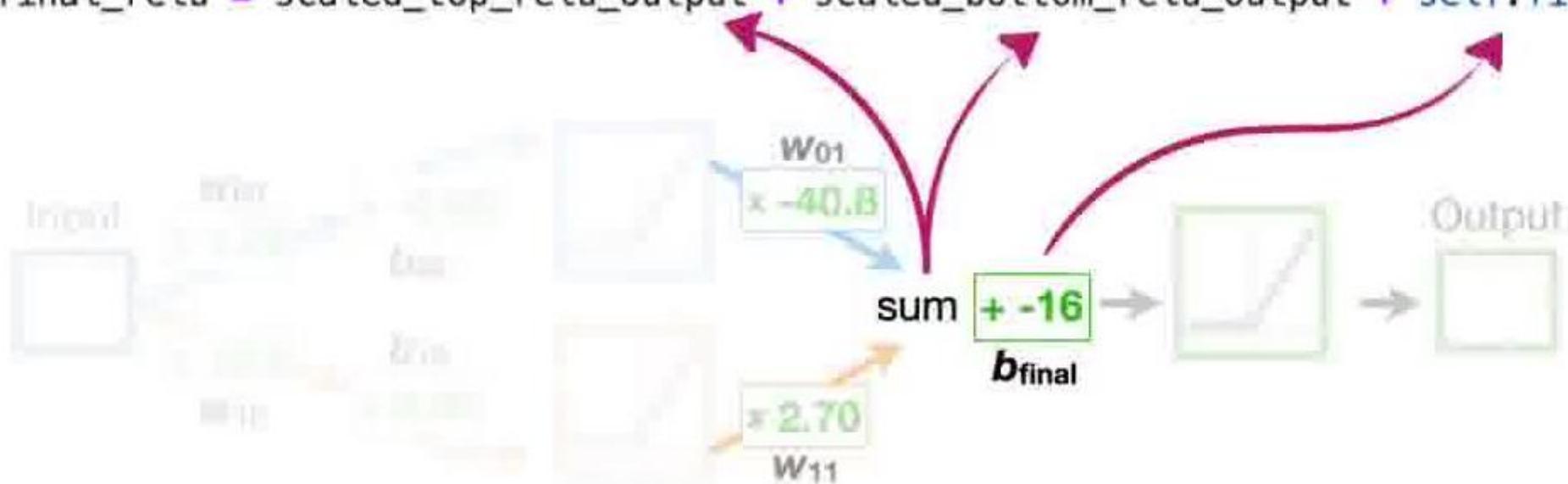


Then we add the top and bottom scaled values to the final **bias**...

```
def forward(self, input):
    input_to_top_relu = input * self.w00 + self.b00
    top_relu_output = F.relu(input_to_top_relu)
    scaled_top_relu_output = top_relu_output * self.w01

    input_to_bottom_relu = input * self.w10 + self.b10
    bottom_relu_output = F.relu(input_to_bottom_relu)
    scaled_bottom_relu_output = bottom_relu_output * self.w11

    input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias
```



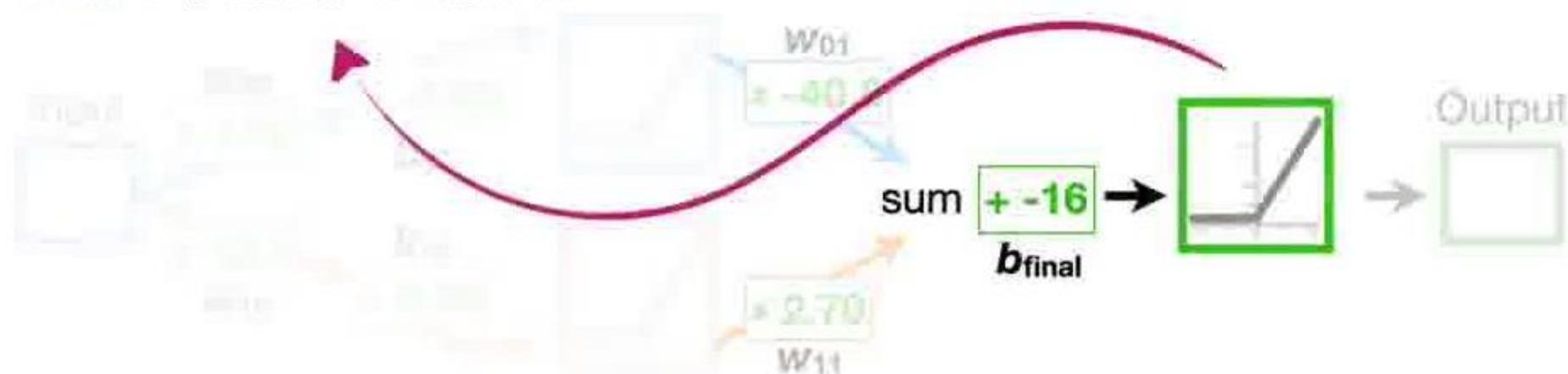
...and use the sum as the input to the final **ReLU** to get the output value.

```
def forward(self, input):
    input_to_top_relu = input * self.w00 * self.b00
    top_relu_output = F.relu(input_to_top_relu)
    scaled_top_relu_output = top_relu_output * self.w01

    input_to_bottom_relu = input * self.w10 + self.b10
    bottom_relu_output = F.relu(input_to_bottom_relu)
    scaled_bottom_relu_output = bottom_relu_output * self.w11

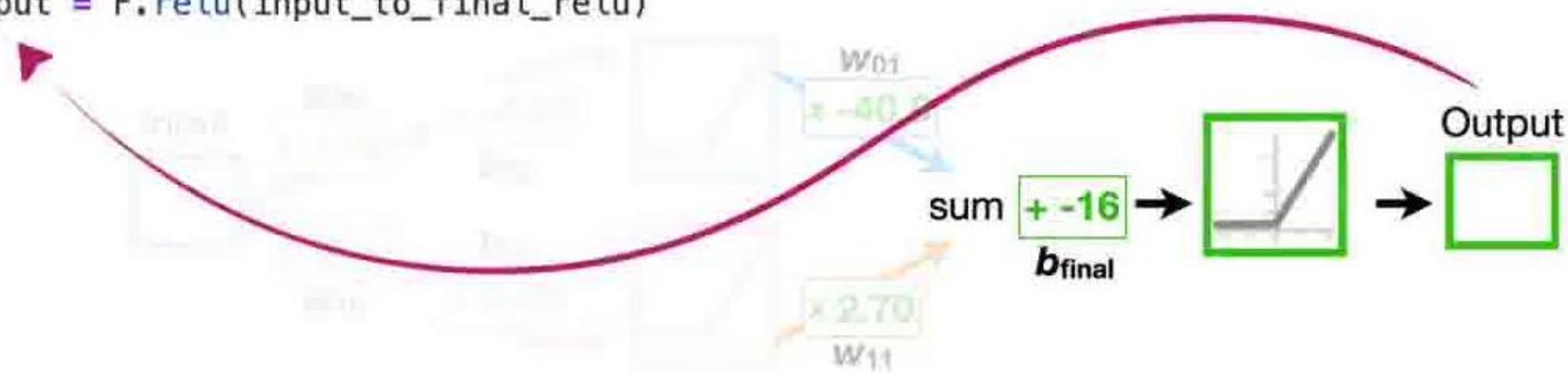
    input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias

    output = F.relu(input_to_final_relu)
```



...and use the sum as the input to the final **ReLU** to get the output value.

```
def forward(self, input):  
  
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)  
    scaled_top_relu_output = top_relu_output * self.w11  
  
    input_to_bottom_relu = input * self.w10 + self.b10  
    bottom_relu_output = F.relu(input_to_bottom_relu)  
    scaled_bottom_relu_output = bottom_relu_output * self.w11  
  
    input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias  
  
    output = F.relu(input_to_final_relu)
```



Lastly, the **forward()** function returns the **output**.

```
def forward(self, input):  
  
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)  
    scaled_top_relu_output = top_relu_output * self.w01  
  
    input_to_bottom_relu = input * self.w10 + self.b10  
    bottom_relu_output = F.relu(input_to_bottom_relu)  
    scaled_bottom_relu_output = bottom_relu_output * self.w11  
  
    input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias  
  
    output = F.relu(input_to_final_relu)  
  
    return output
```



```
class BasicNN(nn.Module):
    def __init__(self):
        super().__init__()

        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)

    def forward(self, input):
        input_to_top_relu = input * self.w00 + self.b00
        top_relu_output = F.relu(input_to_top_relu)
        scaled_top_relu_output = top_relu_output * self.w01

        input_to_bottom_relu = input * self.w10 + self.b10
        bottom_relu_output = F.relu(input_to_bottom_relu)
        scaled_bottom_relu_output = bottom_relu_output * self.w11

        input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias
        output = F.relu(input_to_final_relu)

        return output
```

Now, if we look at the entire class that we created, **BasicNN**...

```
class BasicNN(nn.Module):
    def __init__(self):
        super().__init__()

        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)

        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)

        self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)

    def forward(self, input):
        input_to_top_relu = input * self.w00 + self.b00
        top_relu_output = F.relu(input_to_top_relu)
        scaled_top_relu_output = top_relu_output * self.w01

        input_to_bottom_relu = input * self.w10 + self.b10
        bottom_relu_output = F.relu(input_to_bottom_relu)
        scaled_bottom_relu_output = bottom_relu_output * self.w11

        input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias

        output = F.relu(input_to_final_relu)

        return output
```

...we see two methods,
`__init__()` and
`forward()`.

**`__init__()` creates
and initializes the
weights and
biases...**

```
class BasicNN(nn.Module):  
  
    def __init__(self):  
        super().__init__()  
  
        self.w00 = nn.Parameter(torch.tensor(1.7), requires_grad=False)  
        self.b00 = nn.Parameter(torch.tensor(-0.85), requires_grad=False)  
        self.w01 = nn.Parameter(torch.tensor(-40.8), requires_grad=False)  
  
        self.w10 = nn.Parameter(torch.tensor(12.6), requires_grad=False)  
        self.b10 = nn.Parameter(torch.tensor(0.0), requires_grad=False)  
        self.w11 = nn.Parameter(torch.tensor(2.7), requires_grad=False)  
  
        self.final_bias = nn.Parameter(torch.tensor(-16.), requires_grad=False)
```

`def forward(self, input):`

```
    input_to_top_relu = input * self.w00 + self.b00  
    top_relu_output = F.relu(input_to_top_relu)  
    scaled_top_relu_output = top_relu_output * self.w10  
  
    input_to_bottom_relu = input * self.w01 + self.b01  
    bottom_relu_output = F.relu(input_to_bottom_relu)  
    scaled_bottom_relu_output = bottom_relu_output * self.w11  
  
    input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias  
  
    output = F.relu(input_to_final_relu)  
  
    return output
```

```
class BasicNN(nn.Module):
```

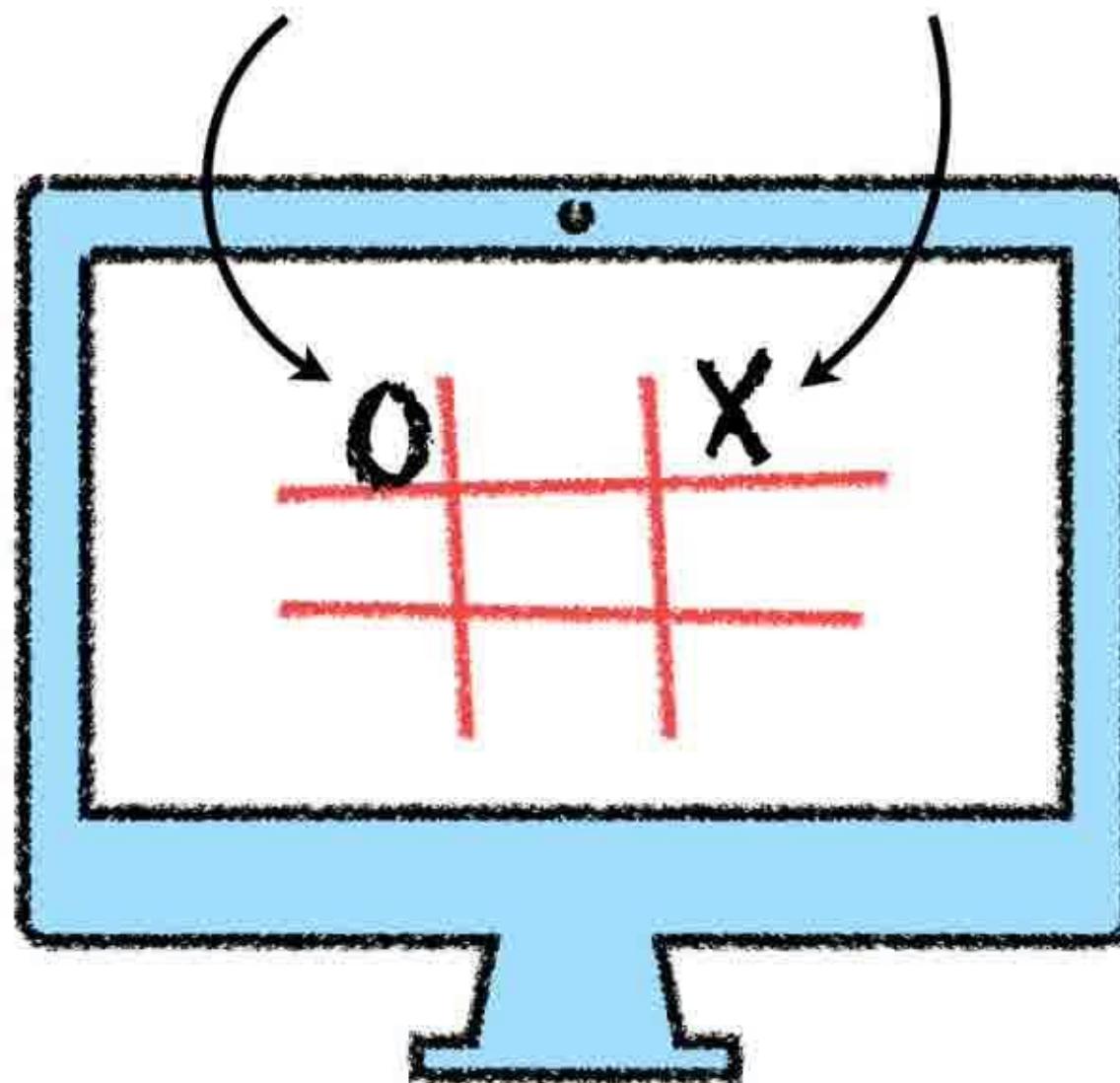
```
    def __init__(self):  
        super().__init__()  
  
        self.w00 = nn.Parameter(torch.FloatTensor(1), requires_grad=False)  
        self.b00 = nn.Parameter(torch.FloatTensor(1), requires_grad=False)  
        self.w01 = nn.Parameter(torch.FloatTensor(1), requires_grad=False)  
  
        self.w10 = nn.Parameter(torch.FloatTensor(1), requires_grad=False)  
        self.b10 = nn.Parameter(torch.FloatTensor(1), requires_grad=False)  
        self.w11 = nn.Parameter(torch.FloatTensor(1), requires_grad=False)  
  
        self.final_bias = nn.Parameter(torch.FloatTensor(1), requires_grad=False)
```

```
    def forward(self, input):
```

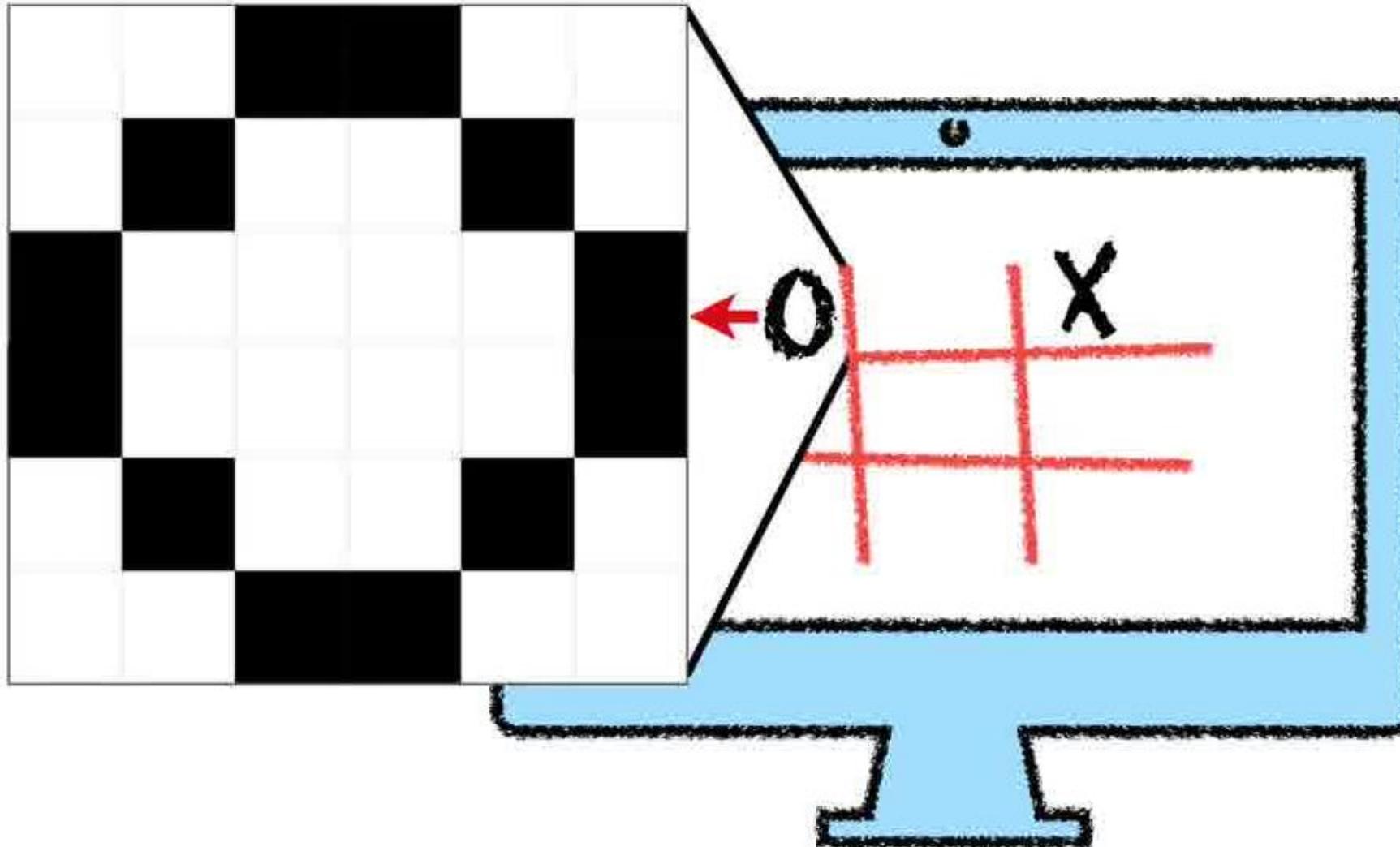
```
        input_to_top_relu = input * self.w00 + self.b00  
        top_relu_output = F.relu(input_to_top_relu)  
        scaled_top_relu_output = top_relu_output * self.w01  
  
        input_to_bottom_relu = input * self.w10 + self.b10  
        bottom_relu_output = F.relu(input_to_bottom_relu)  
        scaled_bottom_relu_output = bottom_relu_output * self.w11  
  
        input_to_final_relu = scaled_top_relu_output + scaled_bottom_relu_output + self.final_bias  
  
        output = F.relu(input_to_final_relu)  
  
        return output
```

...and **forward()** does a **forward pass** through the neural network by taking an input value and calculating the output value with the **weights, biases** and **activation functions**.

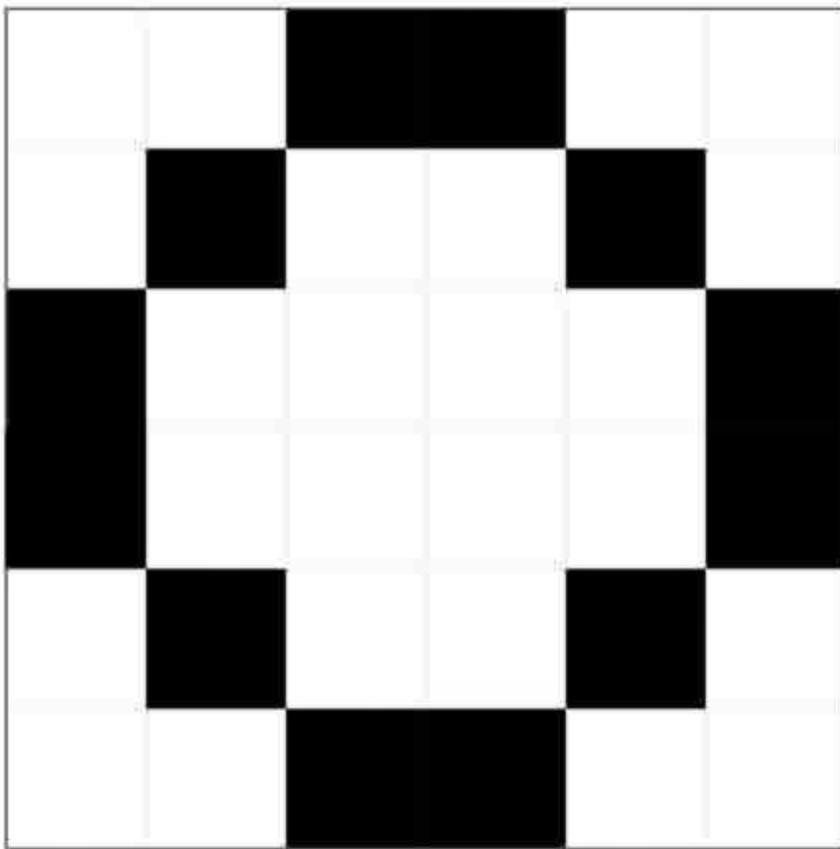
Wie geht ein Neuronales Netz bei
Bilddaten vor?



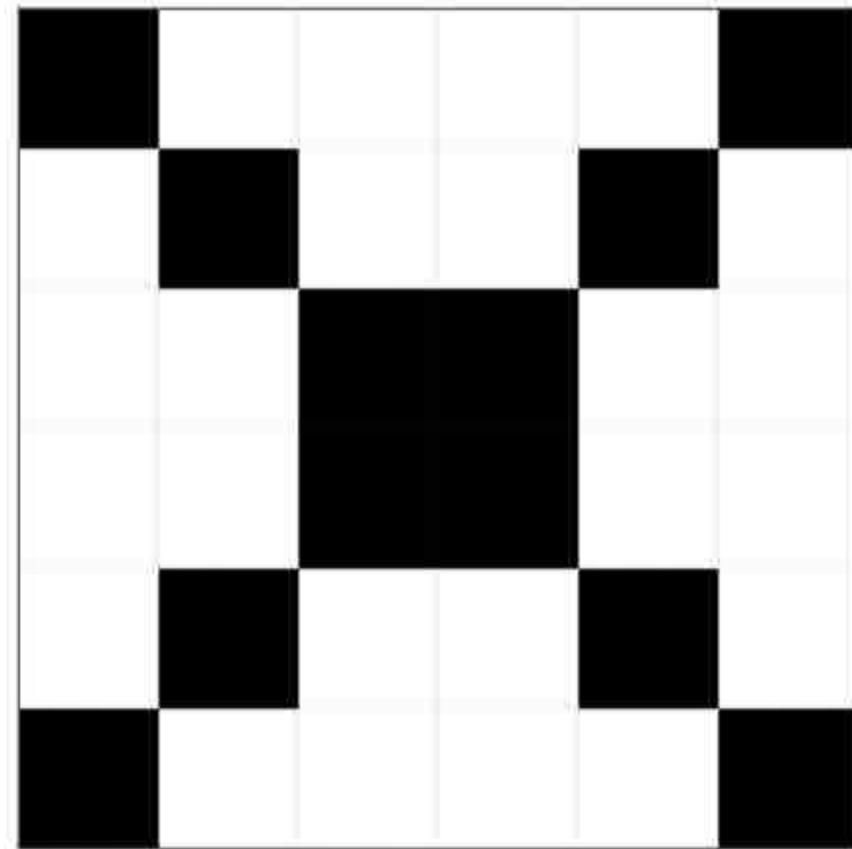
The letter "O", zoomed in.



The letter “O”, zoomed in.



The letter “X”, zoomed in.

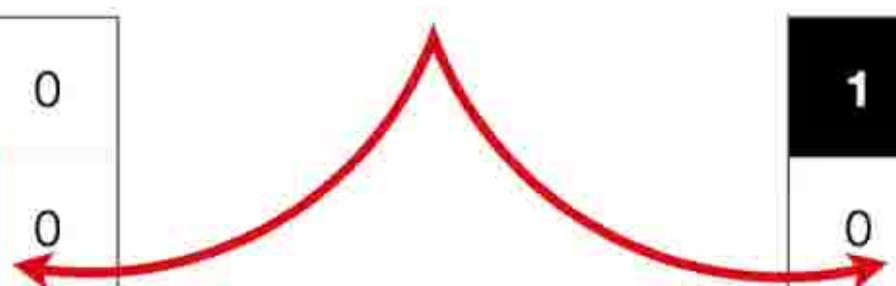


The letter “O”, zoomed in.

0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

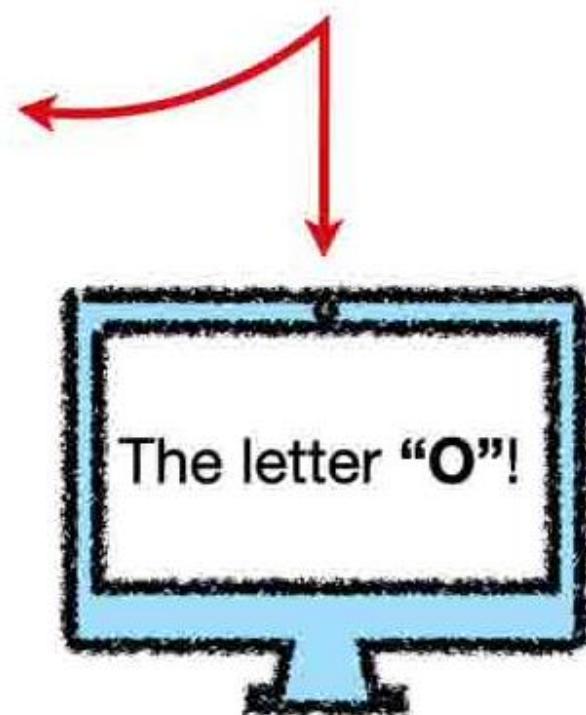
The letter “X”, zoomed in.

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1



The letter “O”, zoomed in.

0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0



The letter “X”, zoomed in.

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1

A 6x6 grid of binary values (0 or 1) representing a pixel image. The grid is labeled "6 pixels wide" with a horizontal pink arrow at the top. The values are:

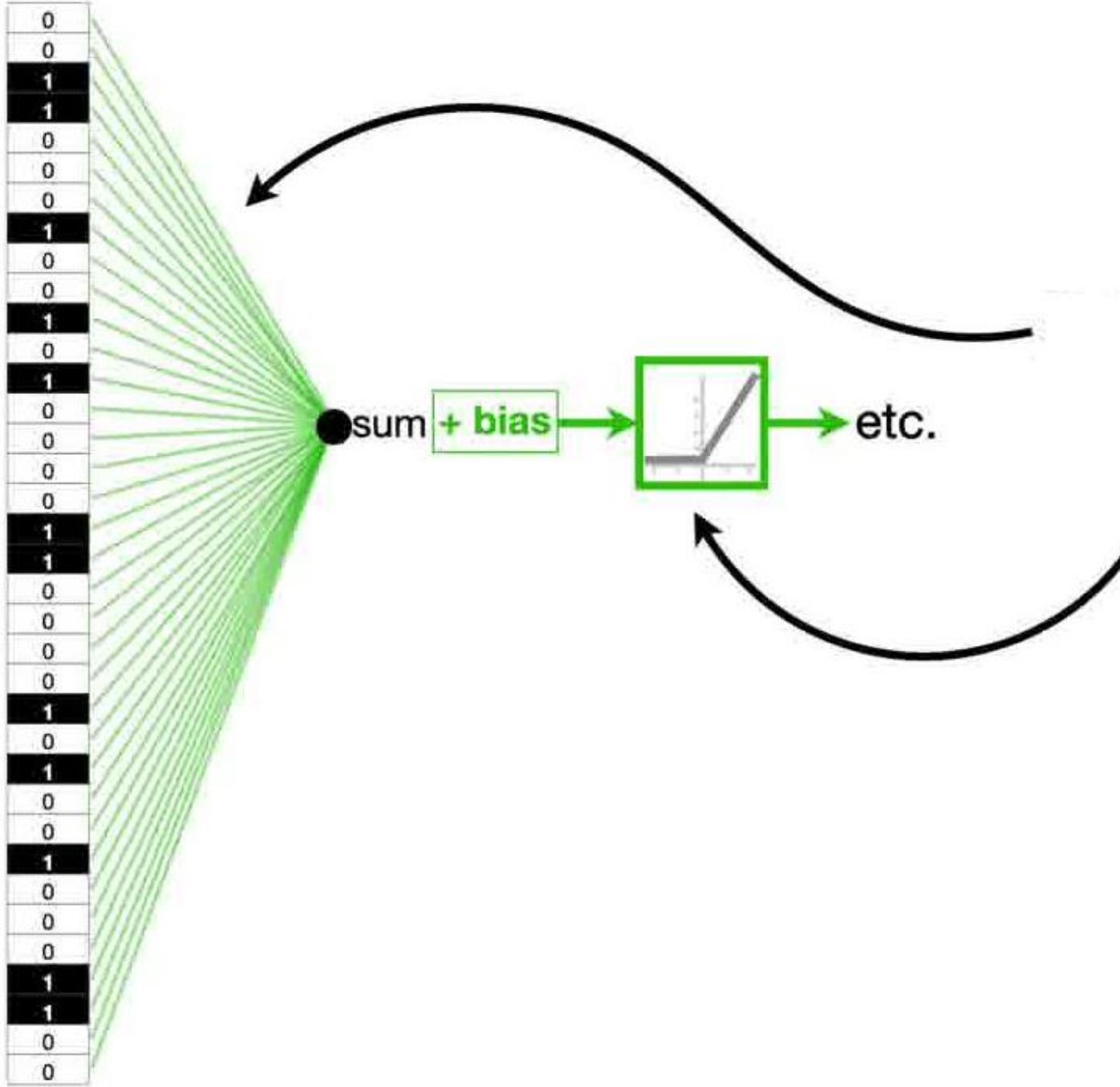
0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

6 pixels tall

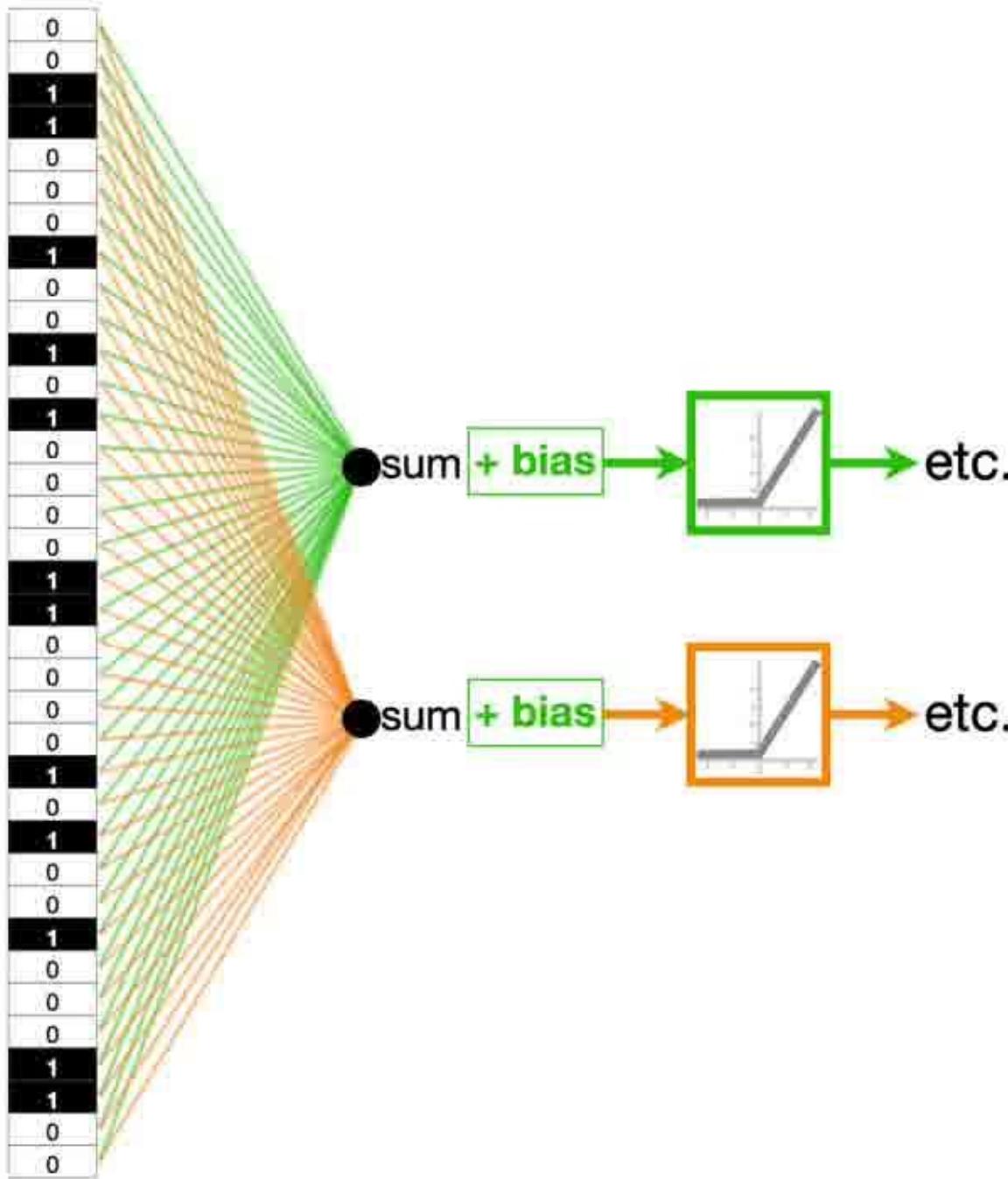
0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

0
0
1
1
0
0
0
1
0
0
1
0
0
0
1
0
0
0
1
1
0
0
1
0
0
0
1
0
0
1
1
0
0

0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0



0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0



Bei großen Bilddaten und komplexeren Neuronalen Netzen wäre ein solches Verfahren viel zu aufwändig!

Lösung:
CONVOLUTIONAL NEURAL NETWORK



Input Image

0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

Filter (aka Kernel)

0	0	1
0	1	0
1	0	0



Ein CNN wendet zunächst einen **Filter** auf das Input Image an.

Input Image

0	0	1		1	0	0
0	1	0		1	0	0
1	0	0		0	0	1
1	0	0		0	0	1
0	1	0		1	0	0
0	0	1	1	0	0	

Filter (aka Kernel)

0	0	1	
0	1	0	
1	0	0	

$$\begin{aligned} & (0 \times 0) + (0 \times 0) + (1 \times 1) \\ & + (0 \times 0) + (1 \times 1) + (0 \times 0) \\ & + (1 \times 1) + (0 \times 0) + (0 \times 0) \end{aligned}$$

= 3

Durch das **Skalarprodukt** zwischen Input und Filter, wird der Filter mit dem Input „convolved“ (zusammengefaltet).

Input Image

0	0	1		1	0	0
0	1	0		0	-1	0
1	0	0		0	0	1
1	0	0		0	0	0
0	1	0		1	0	0
0	0	1	1	0	0	0

Filter (aka Kernel)

0	0	1	
0	1	0	
1	0	0	

Feature Map

1

$$\begin{aligned} & (0 \times 0) + (0 \times 0) + (1 \times 1) \\ & + (0 \times 0) + (1 \times 1) + (0 \times 0) \\ & + (1 \times 1) + (0 \times 0) + (0 \times 0) \end{aligned}$$

$$= 3$$

Input Image

0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

Filter (aka Kernel)

0	0	1
0	1	0
1	0	0

$$+ -2$$

Feature Map

1	-1

$$(0 \times 0) + (1 \times 0) + (1 \times 1)$$

$$+ (1 \times 0) + (0 \times 1) + (0 \times 0)$$

$$+ (0 \times 1) + (0 \times 0) + (0 \times 0)$$

$$= 1$$

Input Image

0	0	1	0	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

Filter (aka Kernel)

0	0	1
0	1	0
1	0	0

$$+ -2$$

Feature Map

1	-1	-2

$$(1 \times 0) + (1 \times 0) + (0 \times 1)$$

$$+ (0 \times 0) + (0 \times 1) + (1 \times 0)$$

$$+ (0 \times 1) + (0 \times 0) + (0 \times 0)$$

$$= 0$$

Input Image

0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

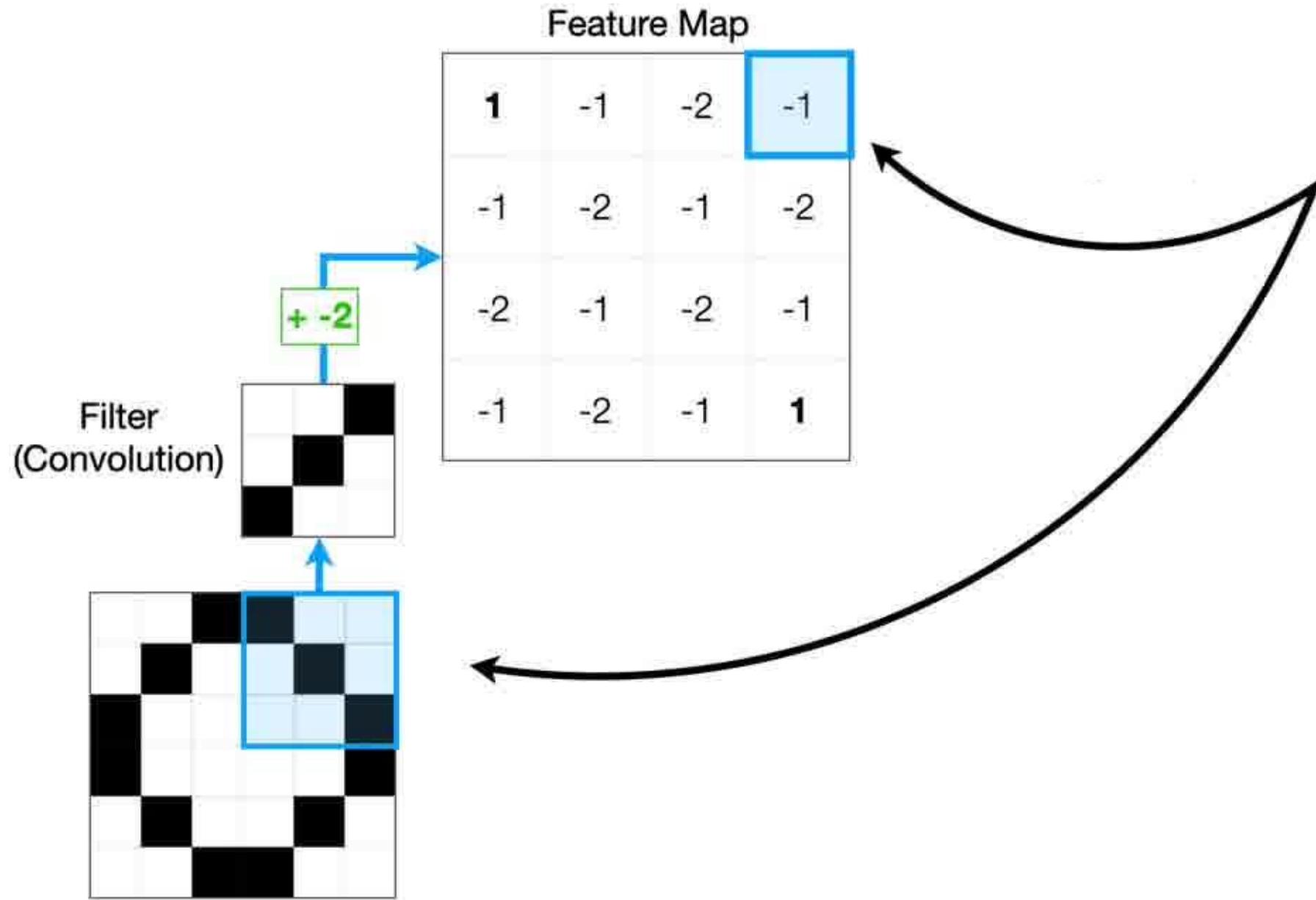
Filter (aka Kernel)

0	0	1
0	1	0
1	0	0

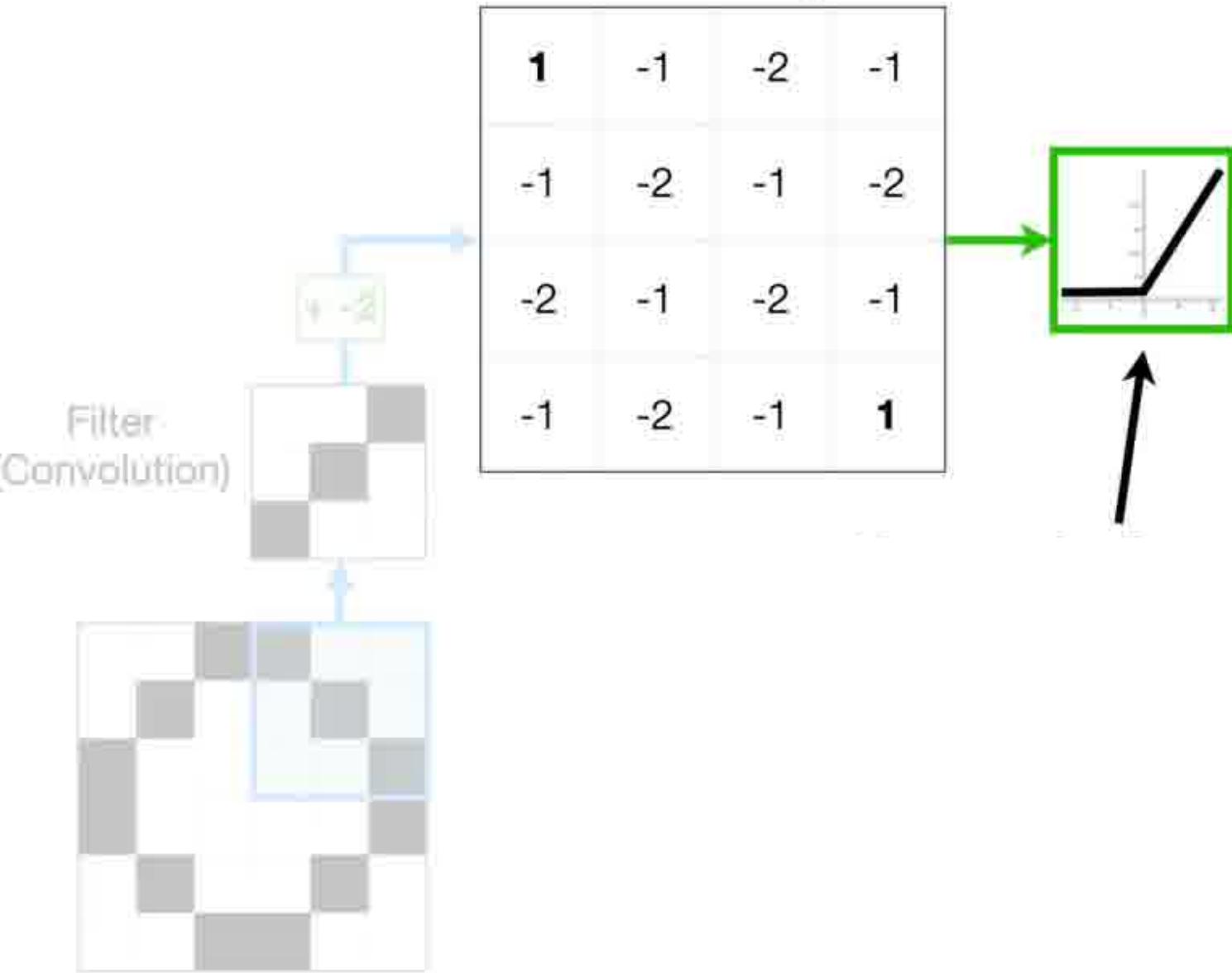
$$+ -2$$

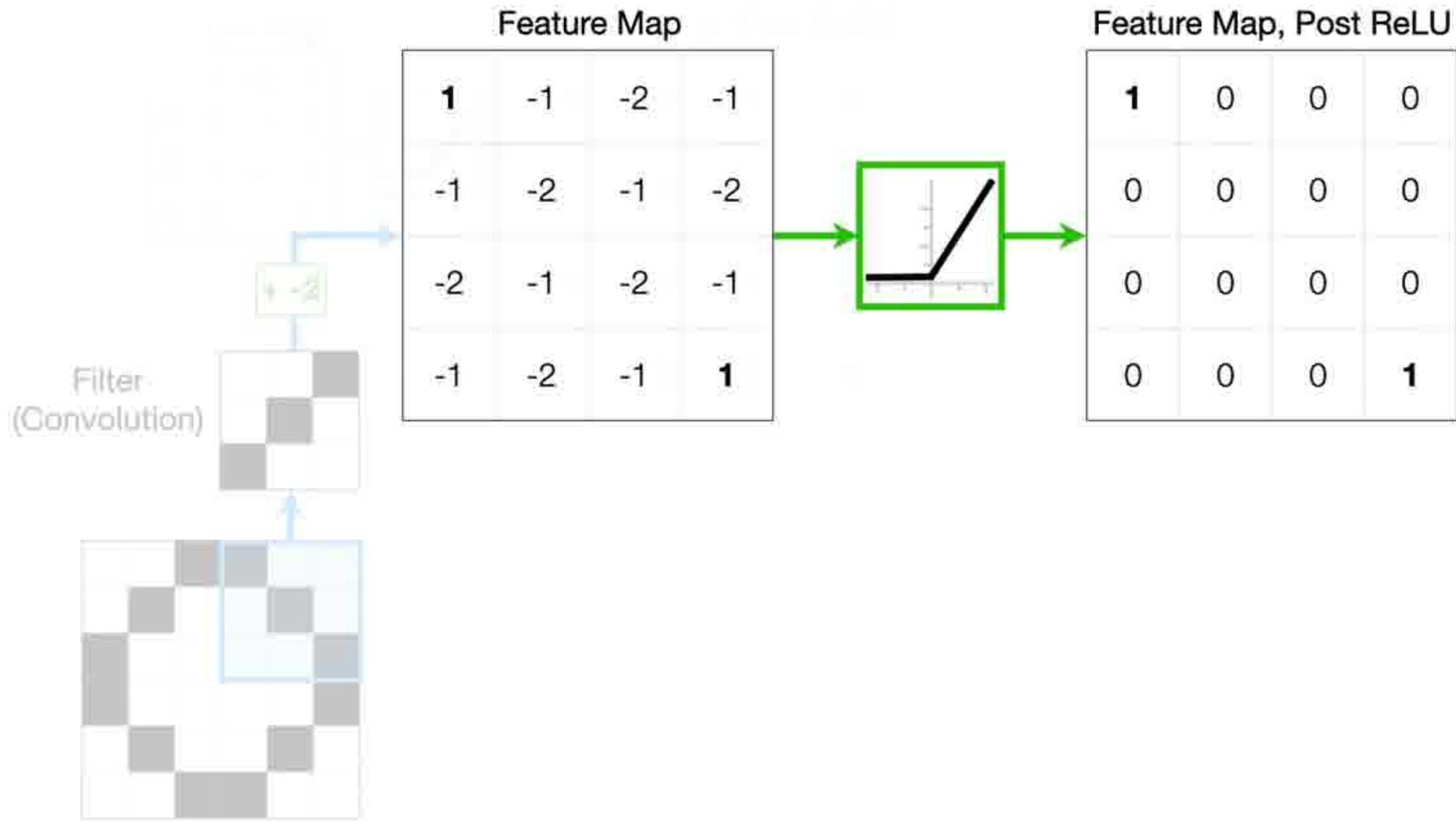
Feature Map

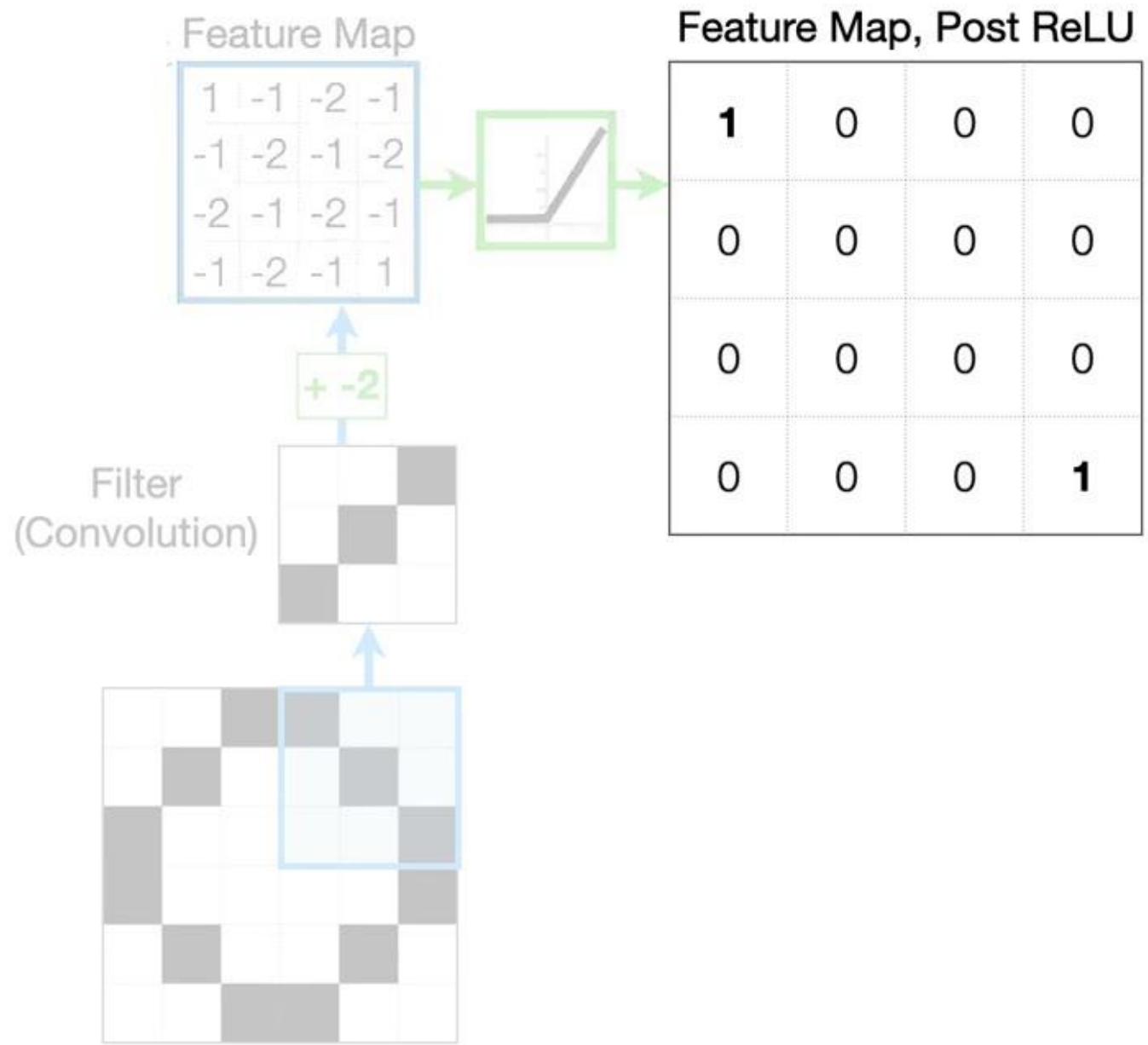
1	-1	-2	-1
-1	-2	-1	-2
-2	-1	-2	-1
-1	-2	-1	1

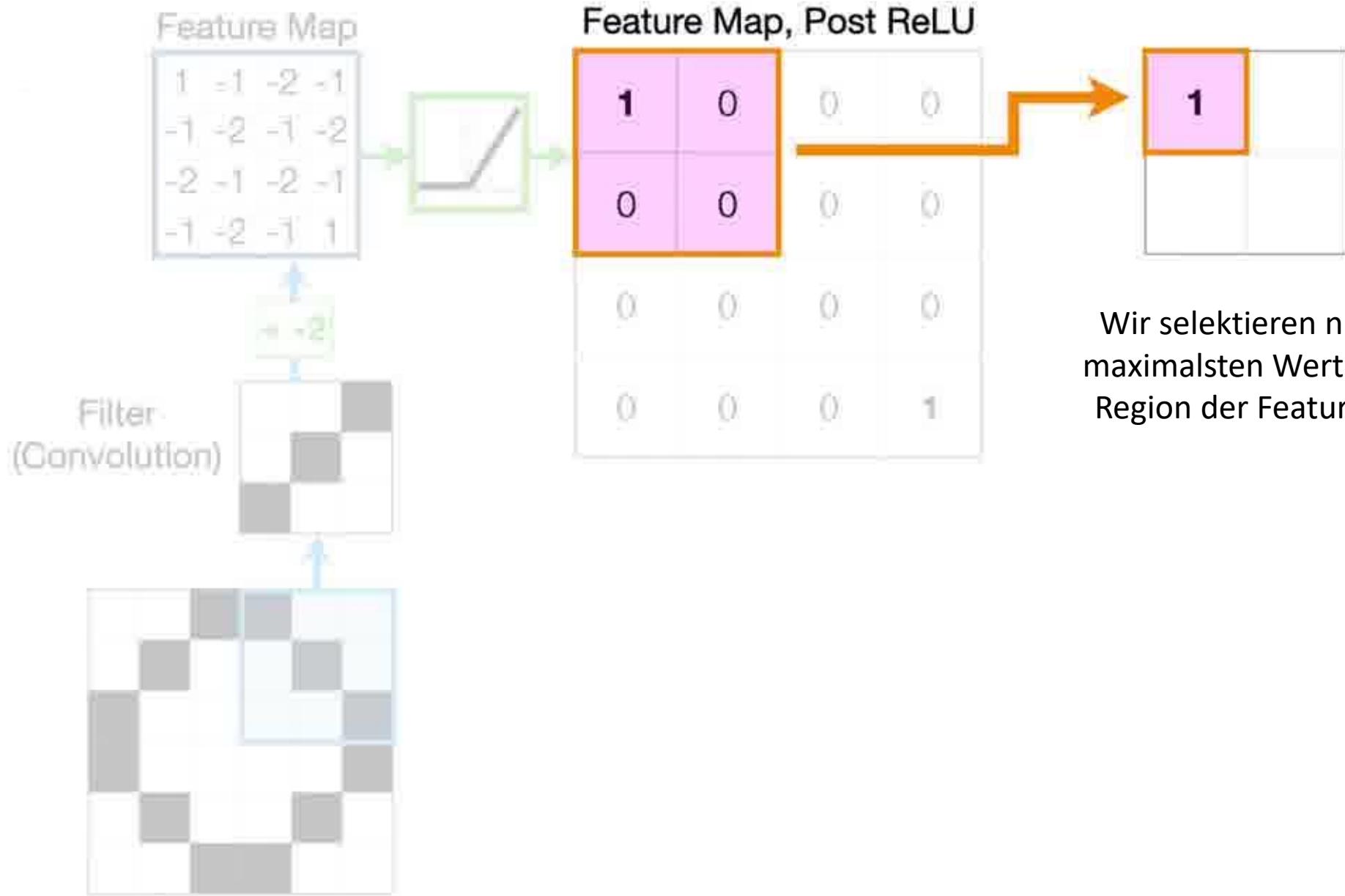


Feature Map



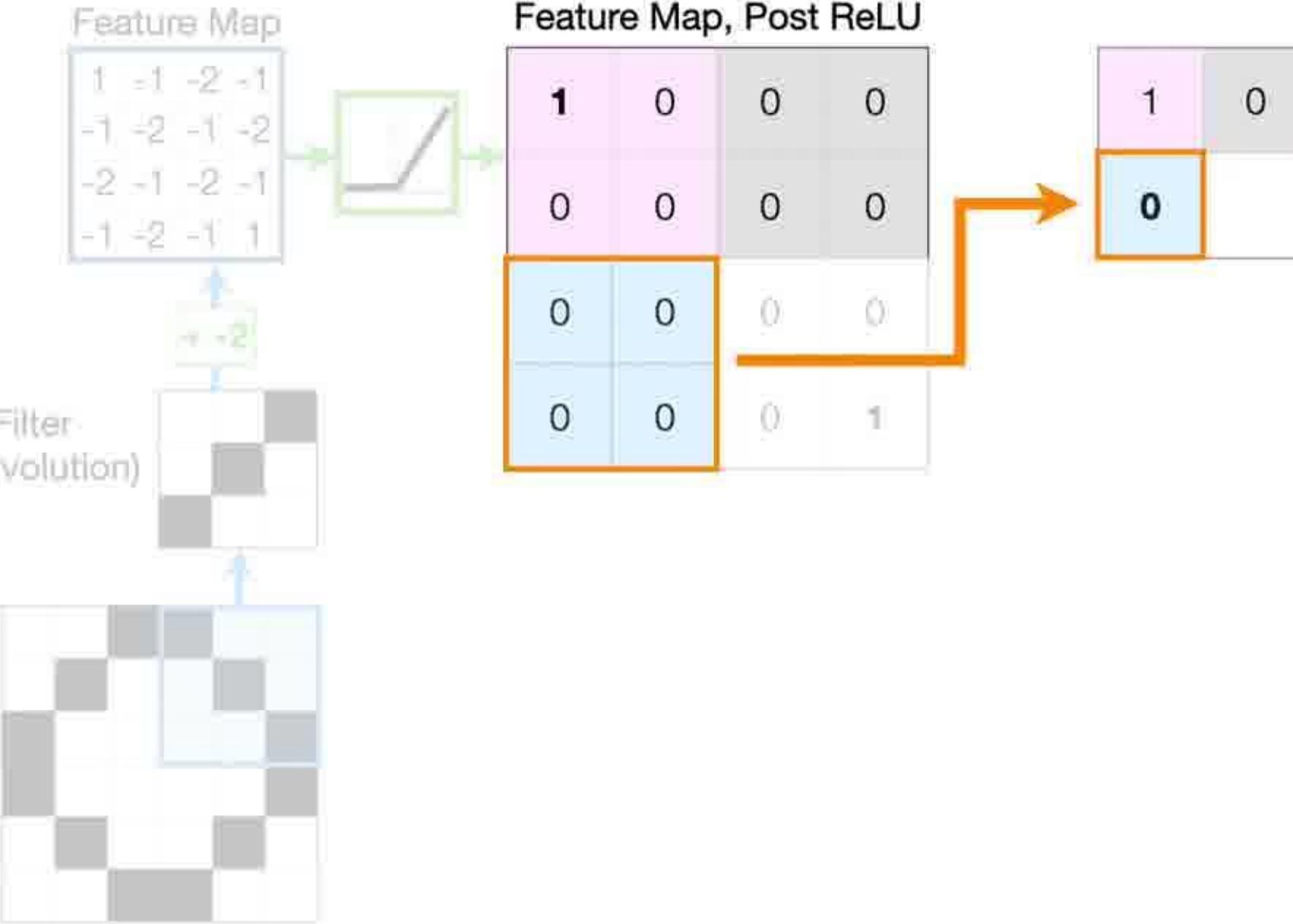


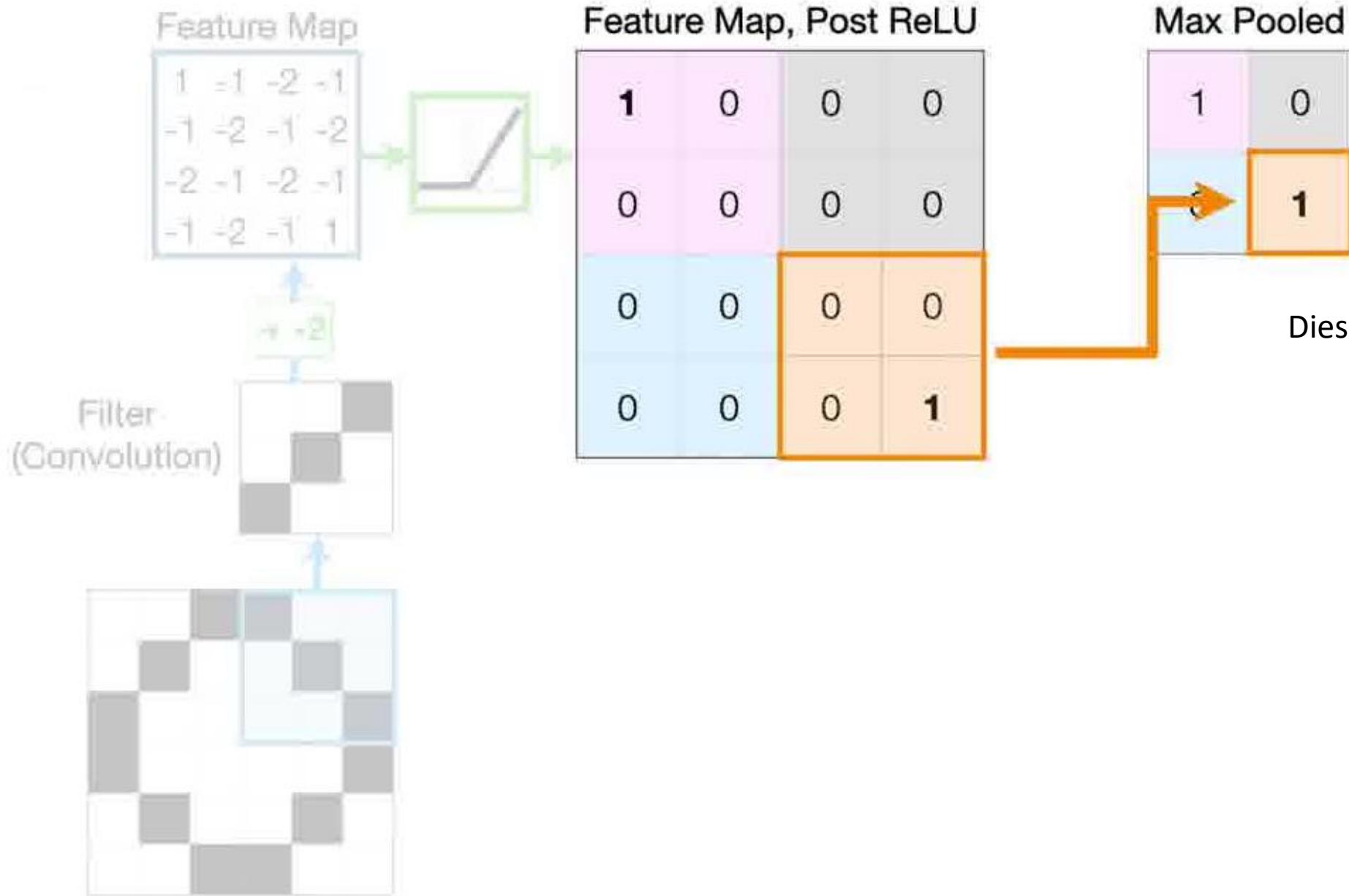




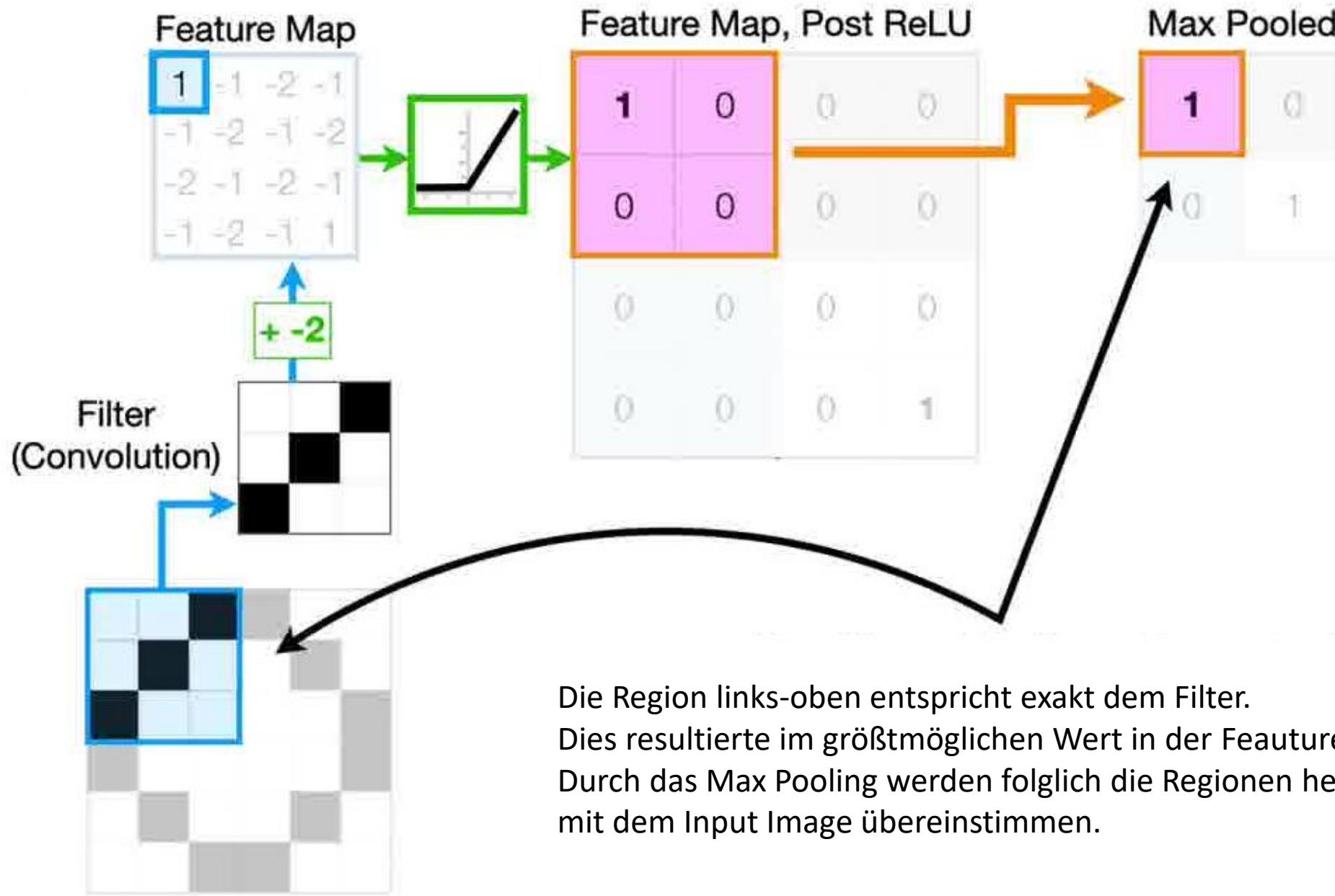
Wir selektieren nun den maximalsten Wert in einer Region der Feature Map.



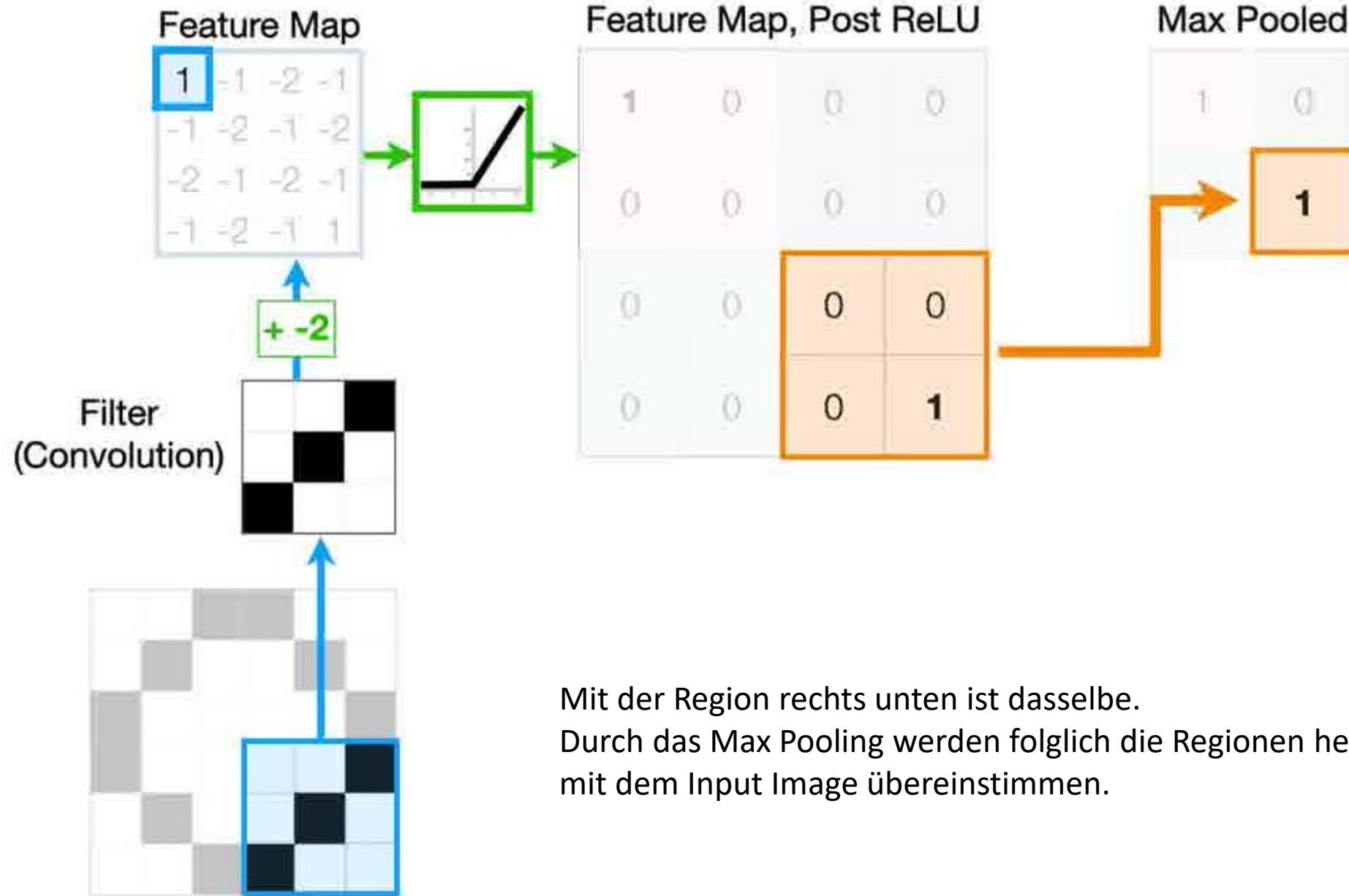


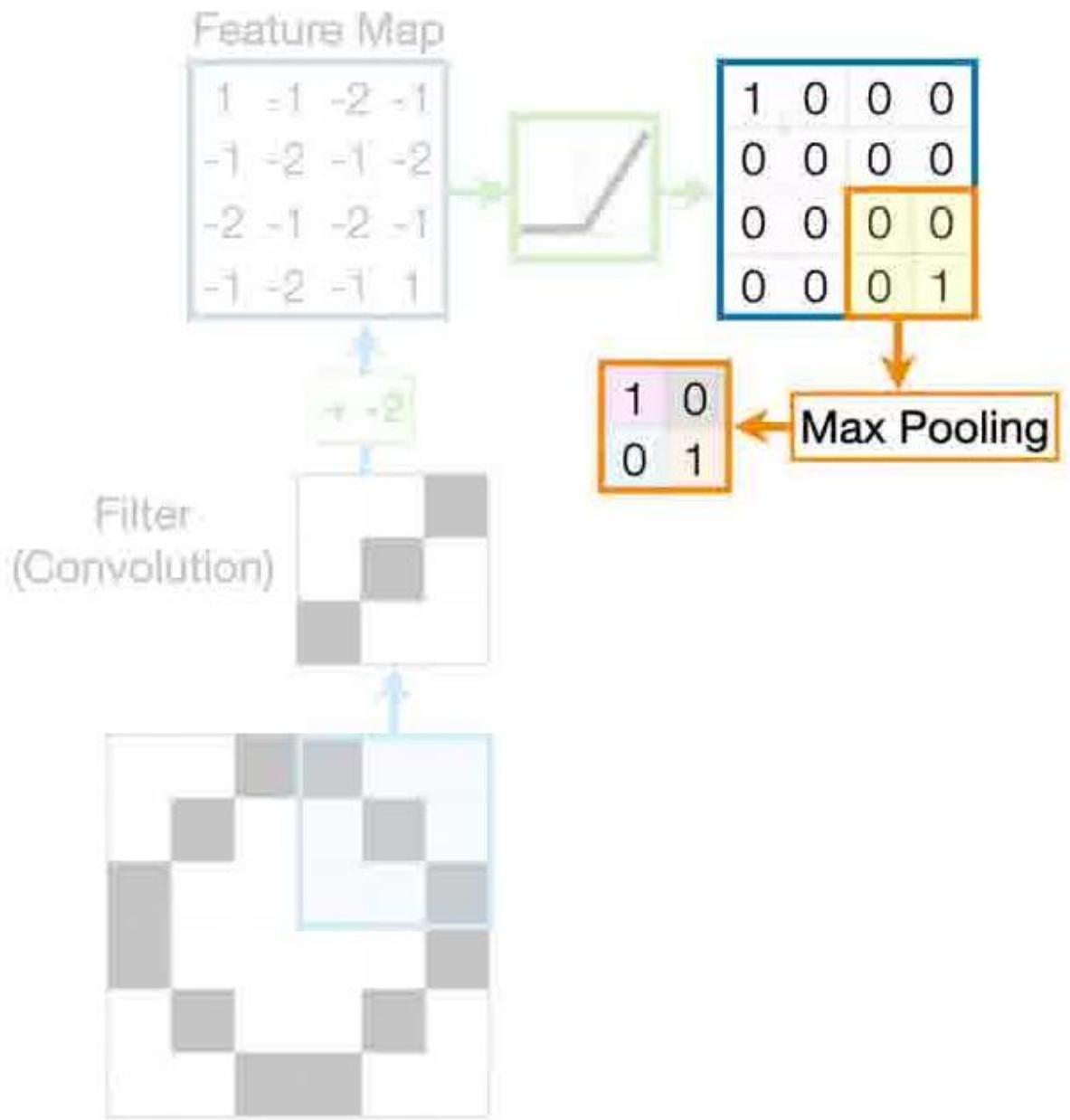


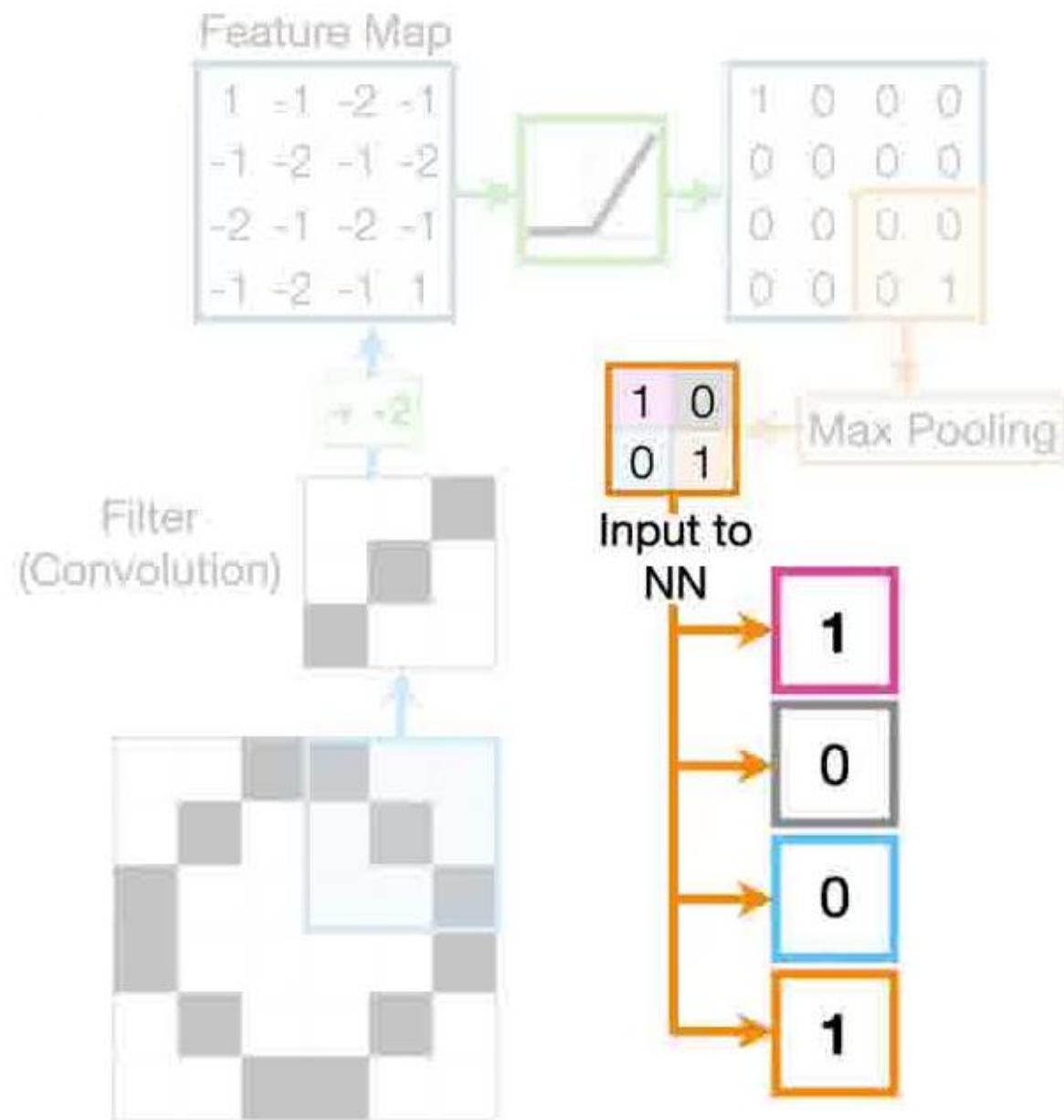
Dies bezeichnet man als
Max Pooling.



Die Region links-oben entspricht exakt dem Filter.
 Dies resultierte im größtmöglichen Wert in der Feauture Map.
 Durch das Max Pooling werden folglich die Regionen hervorgehoben, die am besten mit dem Input Image übereinstimmen.

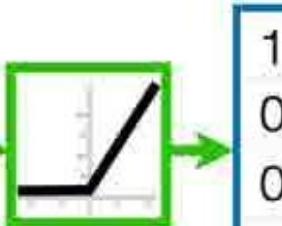






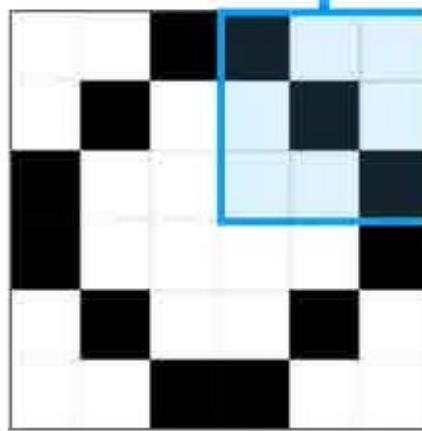
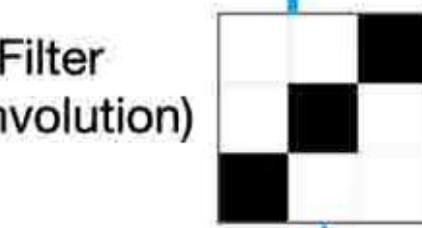
Feature Map

1	-1	-2	-1
-1	-2	-1	-2
-2	-1	-2	-1
-1	-2	-1	1



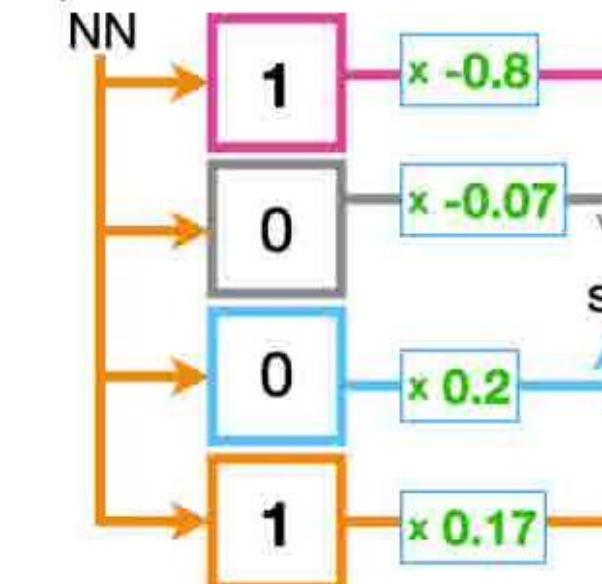
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	1

Filter
(Convolution)



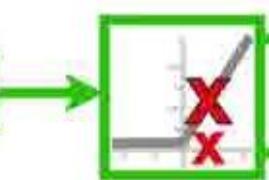
Max Pooling

Input to NN



sum

+ 0.97

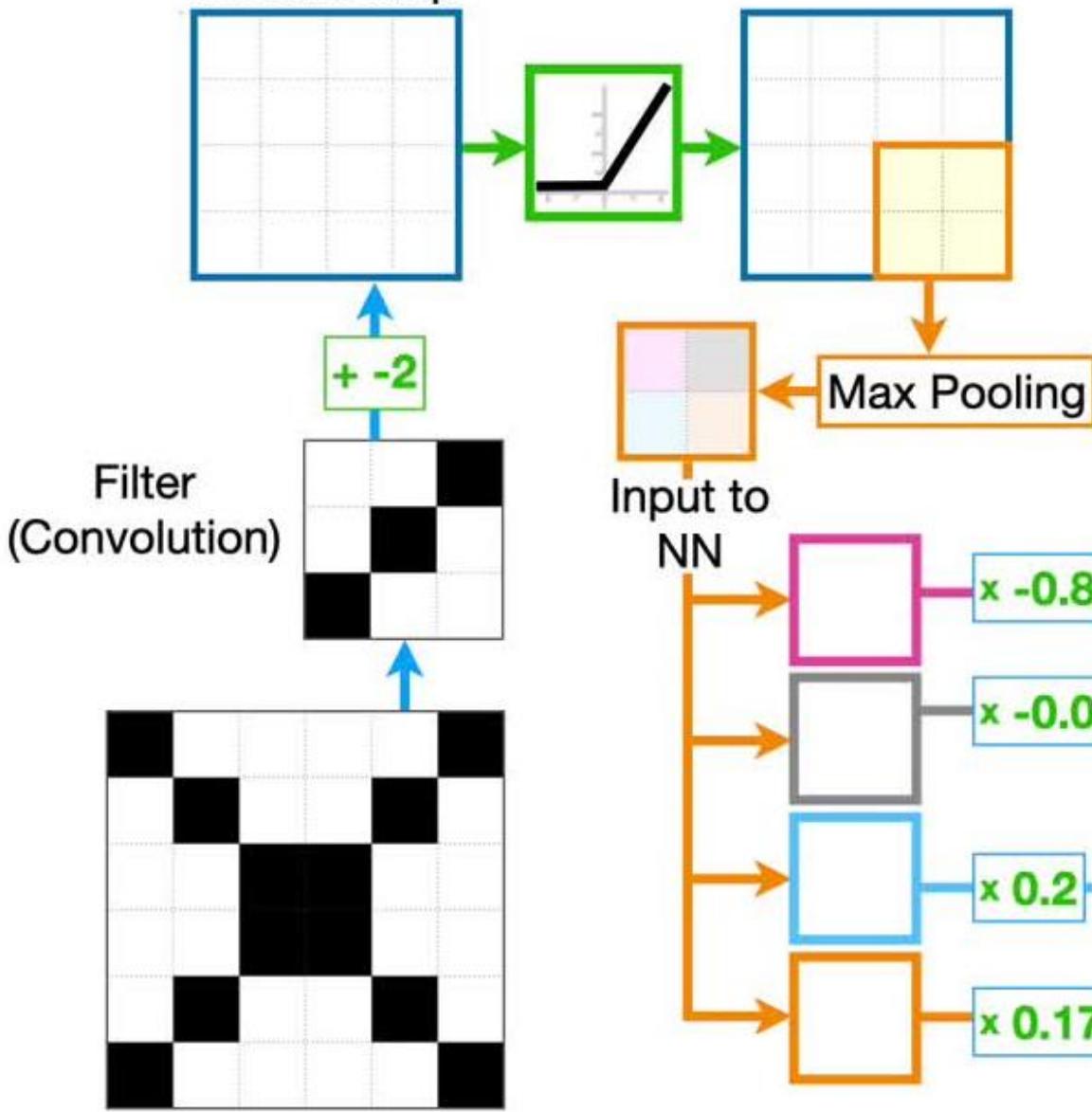


$$+ 1.45 = 1$$

$$+ -0.45 = 0$$



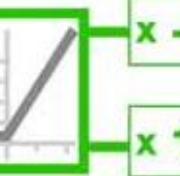
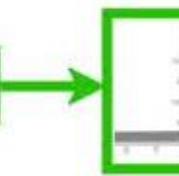
Feature Map



Input to
NN

sum

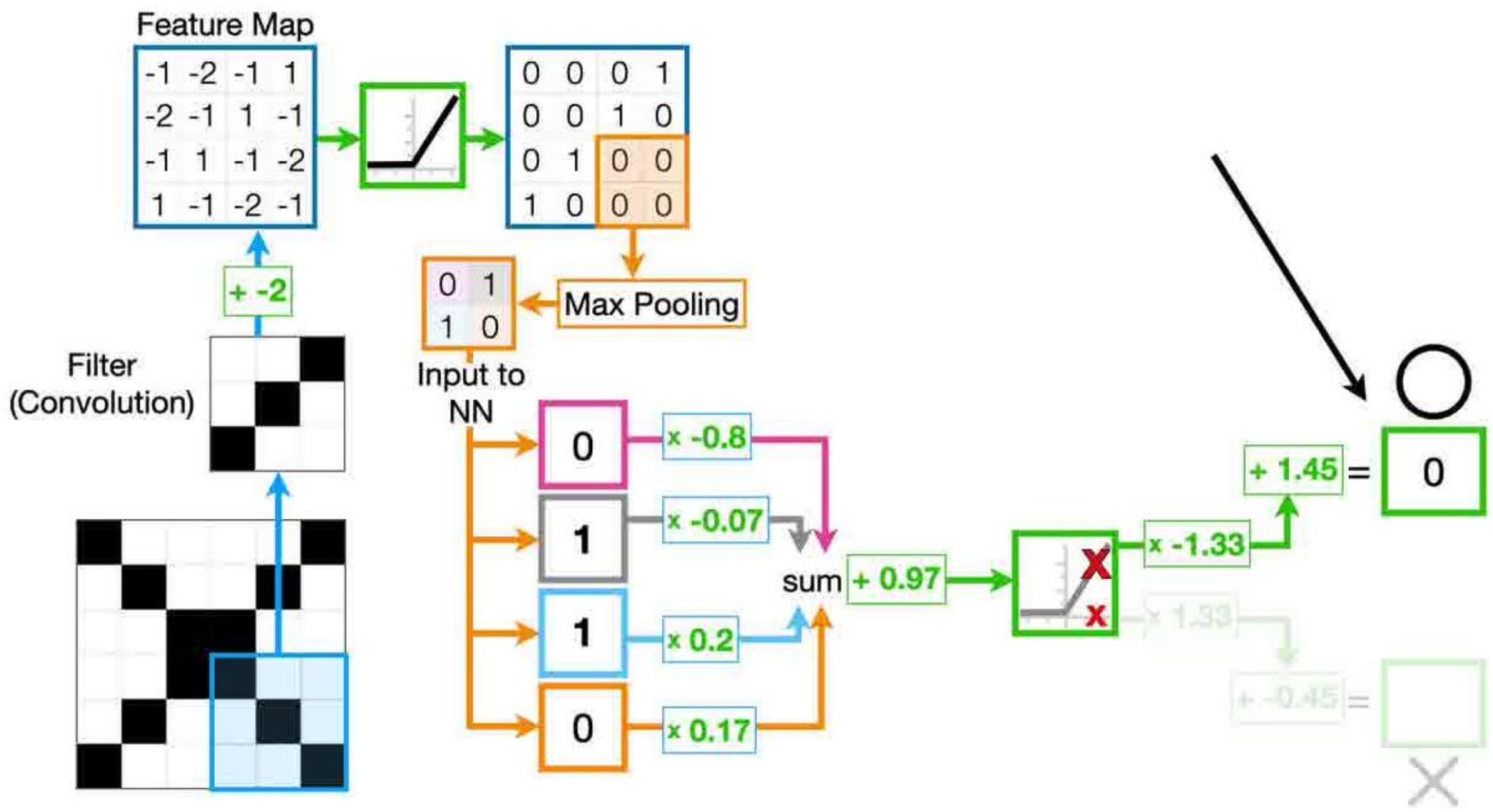
$$+ 0.97$$



$$+ 1.45 =$$

$$+ -0.45 =$$

\times



Feature Map

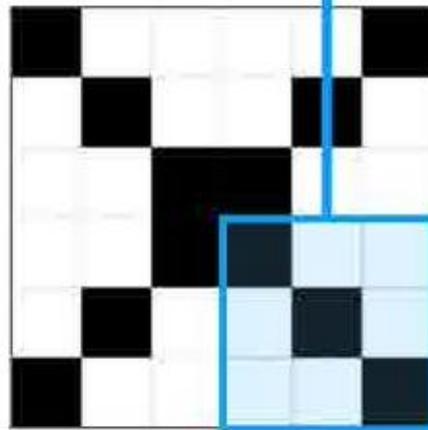
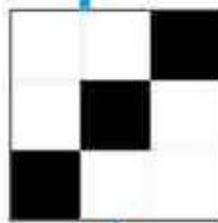
-1	-2	-1	1
-2	-1	1	-1
-1	1	-1	-2
1	-1	-2	-1



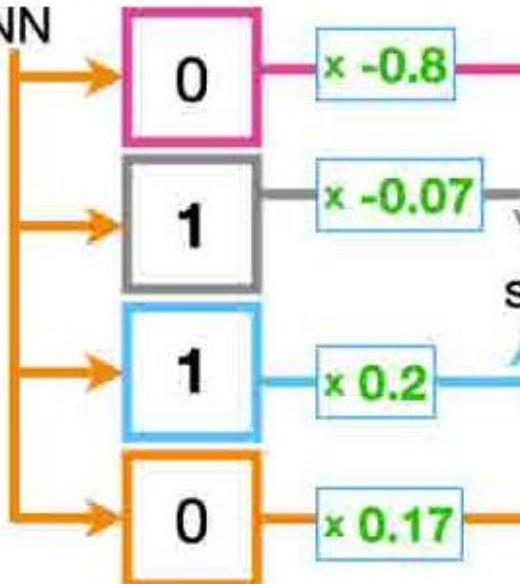
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0



Filter
(Convolution)

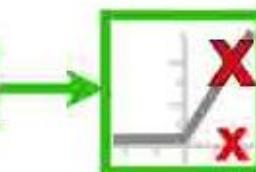


Input to
NN



sum

$$+ 0.97$$



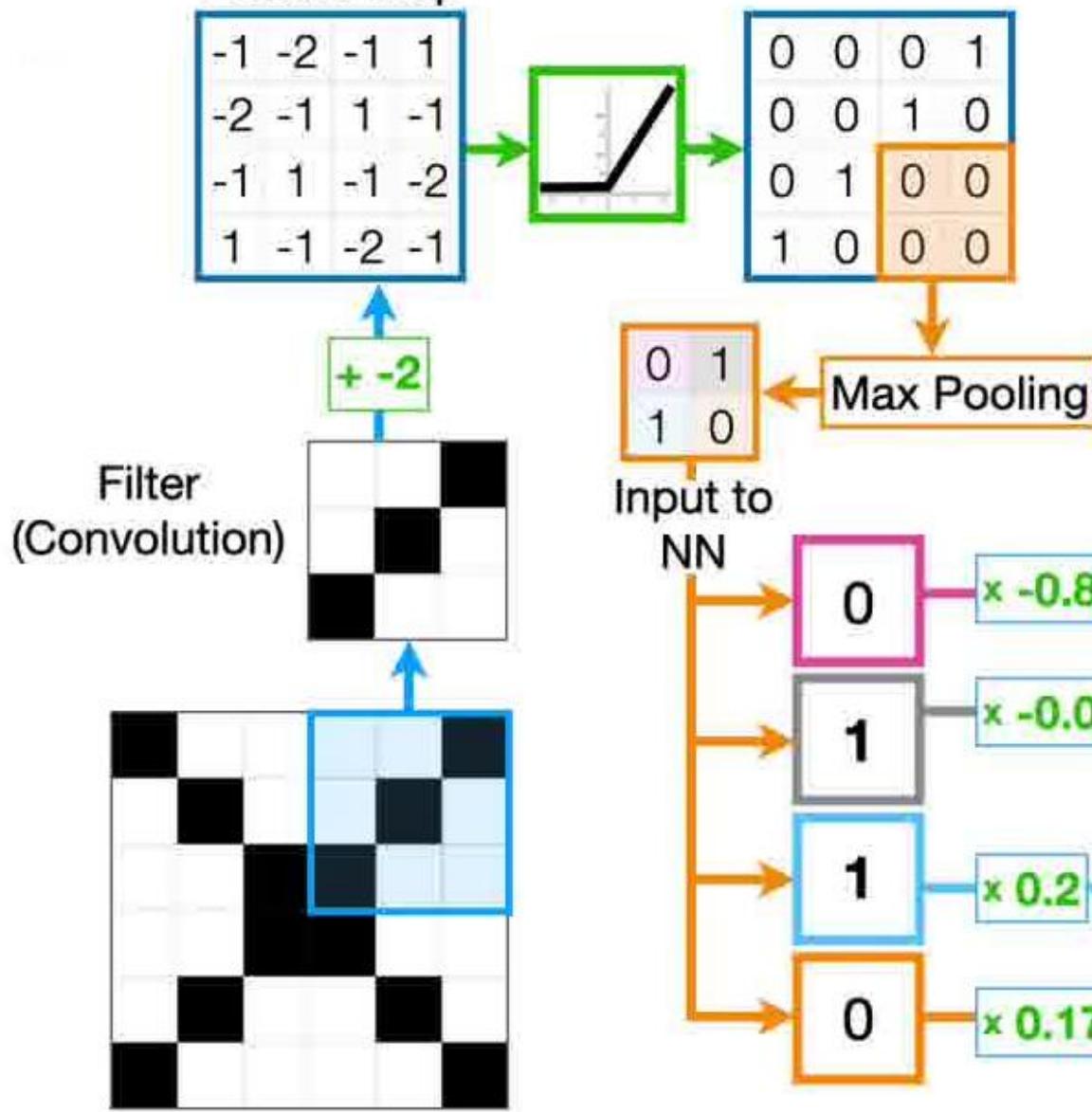
$$\times 1.33$$

$$+ -0.45 = 1 \times$$

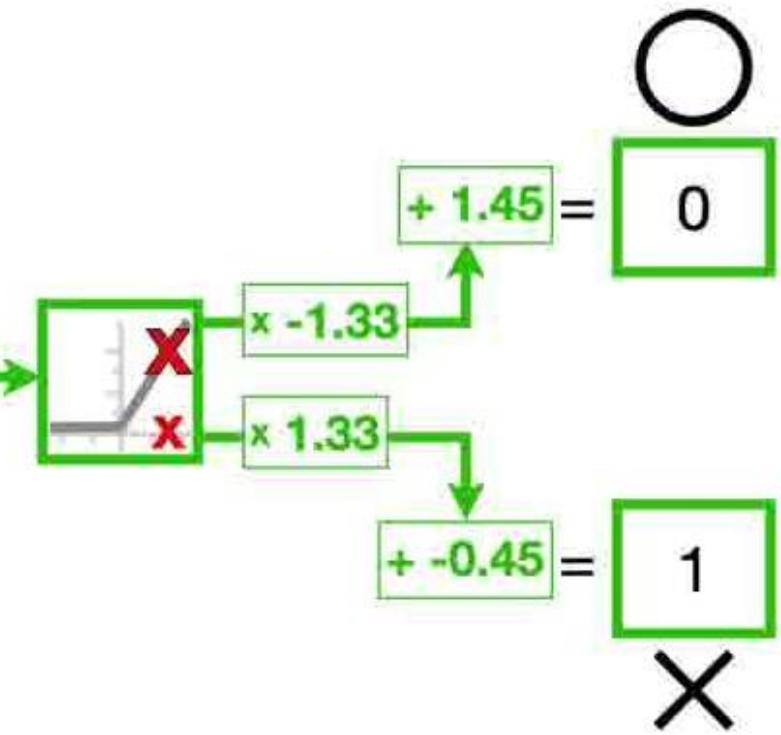


$$0$$

Feature Map

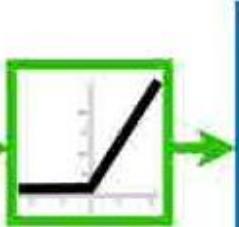


Die Genialität von CNNs besteht folglich darin, große Bilddaten auf kleinste Datenpunkte zu schrumpfen („convolve“), mit dem ein Neuronales Netz effizient arbeiten kann ohne groß Informationen des ursprünglichen Inputs zu verlieren!



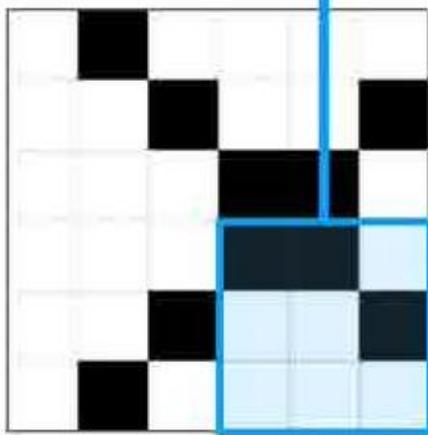
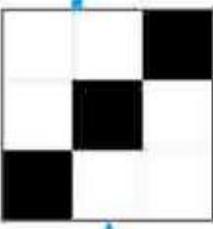
Feature Map

-2	-1	-2	-1
-1	-2	-1	1
-2	-1	1	-1
-2	1	-1	-2



0	0	0	0
0	0	0	1
0	0	1	0
0	1	0	0

Filter
(Convolution)



Max Pooling

Input to NN

NN

0

1

1

1

$\times -0.8$

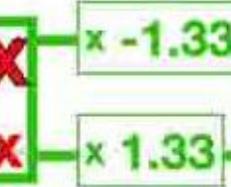
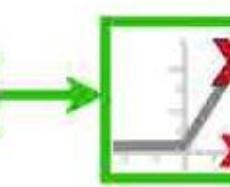
$\times -0.07$

$\times 0.2$

$\times 0.17$

sum

+ 0.97



+ 1.45 = -0.2

+ -0.45 = 1.23





```

import torch
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):

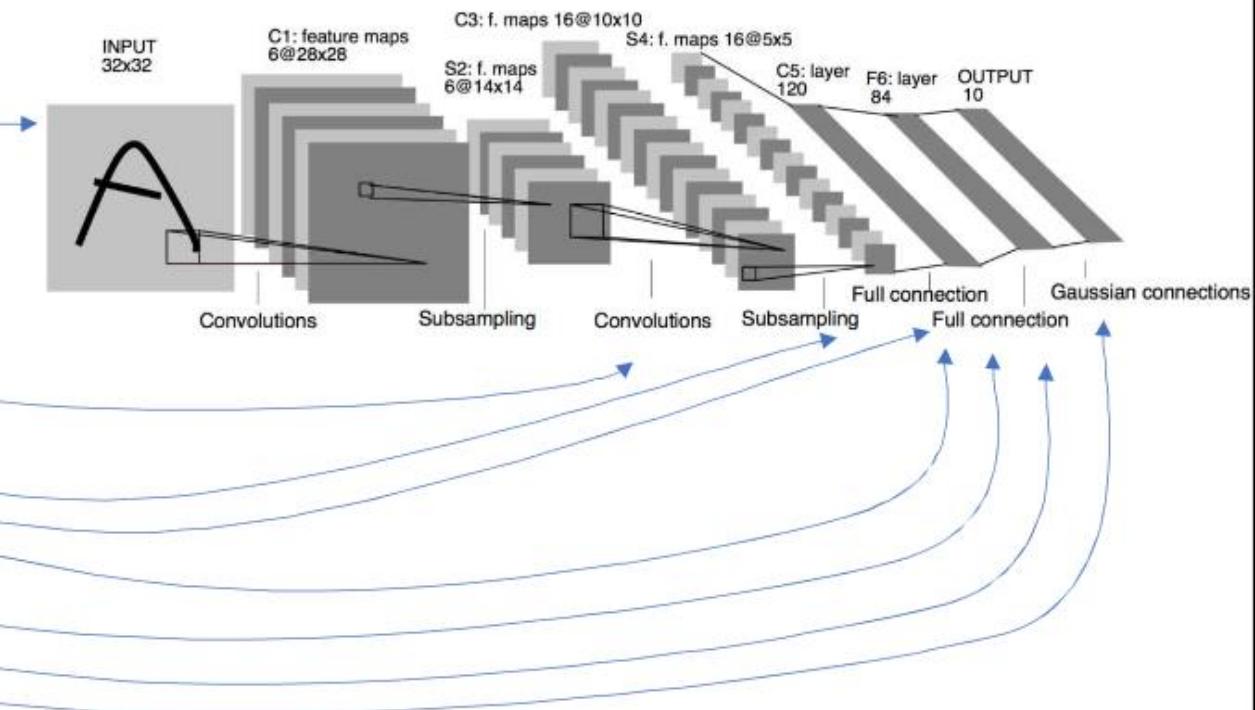
    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120) # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:] # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = Net()
print(net)

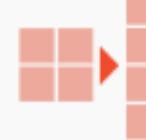
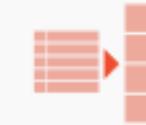
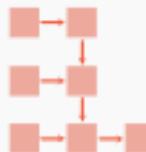
```



https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

– Bernhard Kainz

Layer

- | | | | |
|---|---|--|--|
|  | nn.Linear(m, n): Fully Connected Layer (oder Dense Layer) von m auf n Neuronen |  | nn.ConvXd(m, n, s): X-dimensionaler Convolution Layer von m auf n Kanäle mit Kernelgröße s; $X \in \{1, 2, 3\}$ |
|  | nn.Flatten(): Reduziert die Dimensionen eines Tensors auf 1 |  | nn.MaxPoolXd(s): X-dimensionaler Pooling Layer mit Kernelgröße s; $X \in \{1, 2, 3\}$ |
|  | nn.Dropout(p=0.5): Setzt während des Trainings zufällig Eingaben auf 0; hilft Overfitting zu vermeiden |  | nn.BatchNormXd(n): Normalisiert einen X-dimensionalen Eingabebatch mit n Features; $X \in \{1, 2, 3\}$ |
|  | nn.Embedding(m, n): Bildet Indizes aus Verzeichnis mit Größe m auf Vektoren der Dimension n ab |  | nn.RNN/LSTM/GRU: Recurrent Networks verbinden Neuronen einer Schicht mit Neuronen derselben oder vorheriger Schichten |

torch.nn bietet zudem viele weitere Bausteine.

"State of the art"-Architekturen gibt es unter <https://paperswithcode.com/sota>

Daten laden

Ein Datensatz wird durch eine Klasse repräsentiert, die von **Dataset** erbt (Liste von (Features, Label)-Tupeln).

Mit **DataLoader** erfolgt das Laden strukturunabhängig in Batches.

Üblicherweise wird der Datensatz in Trainings- (z.B. 80%) und Testdaten (z.B. 20%) aufgeteilt.

```
1 from torch.utils.data
2 import Dataset, TensorDataset,
3         DataLoader, random_split
4
5 train_data, test_data =
6     random_split(
7         TensorDataset(inps, tgts),
8         [train_size,test_size]
9     )
10
11 train_loader =
12     DataLoader(
13         dataset=train_data,
14         batch_size=16,
15         shuffle=True)
```

Aktivierungsfunktionen

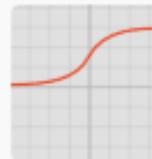
Zu den häufigsten Aktivierungsfunktionen zählen **ReLU**, **Sigmoid** und **Tanh**. Daneben existieren noch eine Reihe weiterer Aktivierungsfunktionen.

nn.ReLU() erzeugt ein **nn.Module** etwa für Sequential Modelle. **F.relu()** ist nur Aufruf der ReLU Funktion etwa für eine Verwendung in der forward Methode.



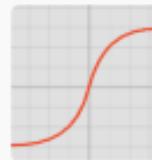
nn.ReLU() oder **F.relu()**

Ausgabe zwischen 0 und ∞ ,
häufigste Aktivierungsfunktion



nn.Sigmoid() oder **F.sigmoid()**

Ausgabe zwischen 0 und 1,
oft für Wahrscheinlichkeiten verwendet



nn.Tanh() oder **F.tanh()**

Ausgabe zwischen -1 und 1,
oft für zwei Klassen verwendet

Modelldefinition

Es gibt verschiedene Wege ein neuronales Netz in PyTorch zu definieren, z.B. mit **nn.Sequential** (a), als Klasse (b) oder auch als Kombination aus beidem.

```
model = nn.Sequential(  
    nn.Conv2D(■, ■, ■)  
    nn.ReLU()  
    nn.MaxPool2D(■)  
    nn.Flatten()  
    nn.Linear(■, ■)  
)
```

a

```
class Net(nn.Module):  
    def __init__(self):  
        super(Net, self).__init__()  
  
        self.conv = nn.Conv2D(■, ■, ■)  
        self.pool = nn.MaxPool2D(■)  
        self.fc = nn.Linear(■, ■)  
  
    def forward(self, x):  
        x = self.pool(F.relu(self.conv(x)))  
        x = x.view(-1, ■)  
        x = self.fc(x)  
        return x  
  
model = Net()
```

b

Training

VERLUSTFUNKTION

PyTorch bietet bereits verschiedene Möglichkeiten den Fehler zu berechnen, u.a.:

nn.L1Loss	Mittlerer absoluter Fehler
nn.MSELoss	Mittlere quadratische Abweichung (L2Loss)
nn.CrossEntropyLoss	Kreuzentropie, u.a. für Single-Label, unausgewogene Trainingsdaten
nn.BCELoss	Binäre Kreuzentropie, u.a. für Multi-Label oder Autoencoder

OPTIMIERUNGsalGORITHMEN (torch.optim)

Optimierungsalgorithmen werden genutzt, um im Gradientenabstiegsverfahren Gewichte zu aktualisieren und die Lernrate dynamisch anzupassen, u.a.:

optim.SGD	Stochastic Gradient Descent
optim.Adam	Adaptive Moment Estimation
optim.Adagrad	Adaptive Gradient
optim.RMSProp	Root Mean Square Prop



Auf zu Colab!

Supervised Learning with scikit-learn

Rückblick: Was haben wir alles in
diesem Wahlfach gelernt?

Grundlagen von Python:

- Grundlegendes Verständnis der Python Syntax
- lists, dictionaries, functions, if/else, for-loops
- matplotlib zur Darstellung Graphen
- Pandas DataFrames

Supervised Learning mit scikit-learn:

- Classification: k-nearest-neighbors
- Regression: linear regression, lasso
- Statistik: overfitting/underfitting, cross-validation, AUROC, Confusion-Matrix

Classification und Regression Trees:

- Decision Trees
- Random Forest

Grundlagen zu Neuronalen Netzen

FAQ: Woher habe ich die ganzen
Datensätze?

Tutorials

Python

- <https://geeksforgeeks.org/python-basics/>
- <https://coursera.org/specializations/python>
- <https://codecademy.com/learn/learn-python>

Machine Learning and Deep Learning

- <https://coursera.org/learn/machine-learning>
- <https://coursera.org/specializations/deep-learning>
- <https://pytorch.org/tutorials/>
- <https://youtube.com/playlist?list=PLTuOKAhkBkunnUQIOYD0PtxC7EZk2TQMv>
- <https://course.fast.ai>
- <https://nvidia.com/en-us/deep-learning-ai/education/>
- <https://d2l.ai/>
- <https://explained.ai/>
- <https://deeplearningbook.org/>
- <https://lelon.io/blog/2018/02/08/pytorch-with-baby-steps>

codecademy

datacamp

CS4MS - S23

This repository contains the exercises - provided in form of Jupyter notebooks - for the course CS4MS taught in the summer semester of 2023.

How to Run

Refer to the [colab_setup](#) for running the notebooks on Google Colaboratory.

Python and machine learning tutorials

More information and additional material can be found here [Summary of Tutorials](#).





szmuenchen ✅

⋮



Markus Söder

Ministerpräsident

„Die TU ist eine der geilsten Unis, die es
überhaupt gibt, echt. Die LMU ist
größer, ist auch super, aber die TUM ist
schon ein sehr schnittiges Schiffchen.“

Foto: Robert Haas

Süddeutsche Zeitung

Bitte evaluieren!

