Assessing Methods for In-Database Machine Learning: An Analysis of Kläbe et al.’s Research

Clara

OUTLOOK

1. **Introduction**
2. **Different Approaches**
3. **Architectures for Neural Networks**
   a. **Perceptrons and Feed Forward Networks**
   b. **Recurrent Networks**
   c. **Convolutional Neural Networks**
4. **AI4DB vs. DB4AI**
5. **ML-TO-SQL Design**
6. **Native MODELJOIN Operator**
7. **Conclusion**

1. Introduction

Why do we need Machine Learning inside DB`s?

-> growing amount of data that is created, stored, processed, consumed

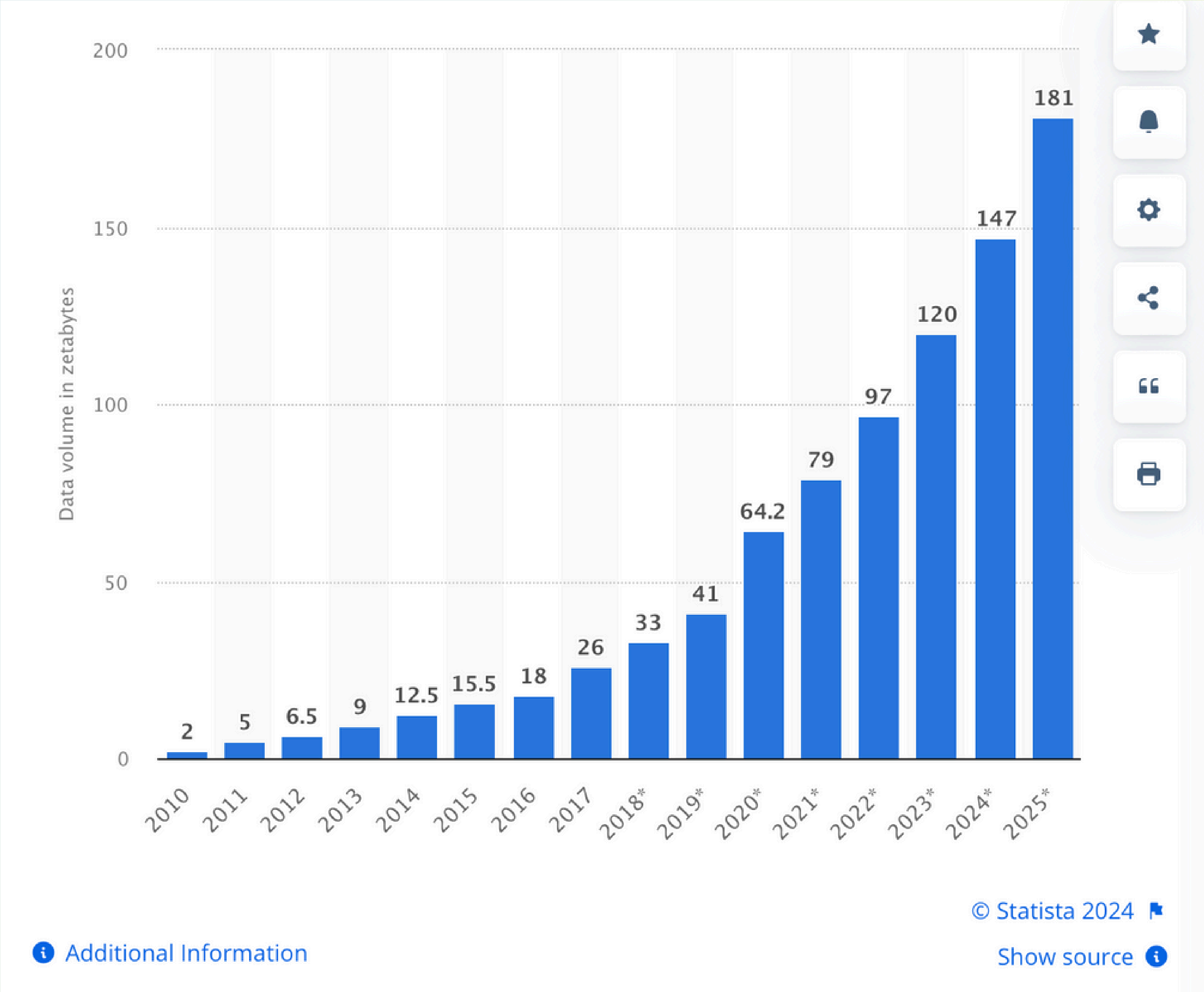


Image by: https://www.statista.com/statistics/871513/worldwide-data-created/

1. **Introduction**

Advantages of ML in DB`s:

- reduced data transfer
- exploiting server hardware
- scalability
- protection of sensitive data

2. Different Approaches

Kläbe et al. identify four different approaches:

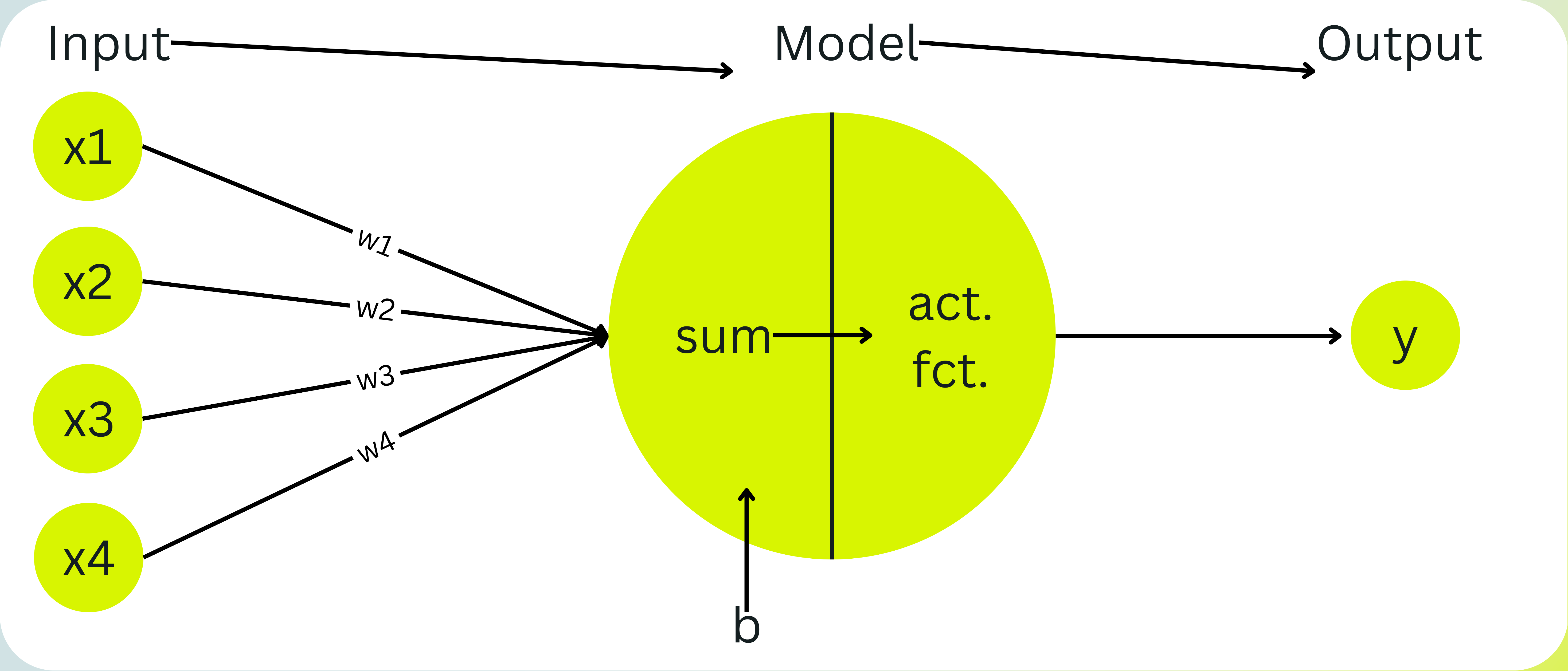
1. Python UDFs
2. Native APIs of ML systems
3. SQL
4. Native operators

3. Architectures for Neural Networks

1. **Perceptron and Feed Forward Network**
2. **Recurrent Networks**
3. **Convolutional Neural Networks**

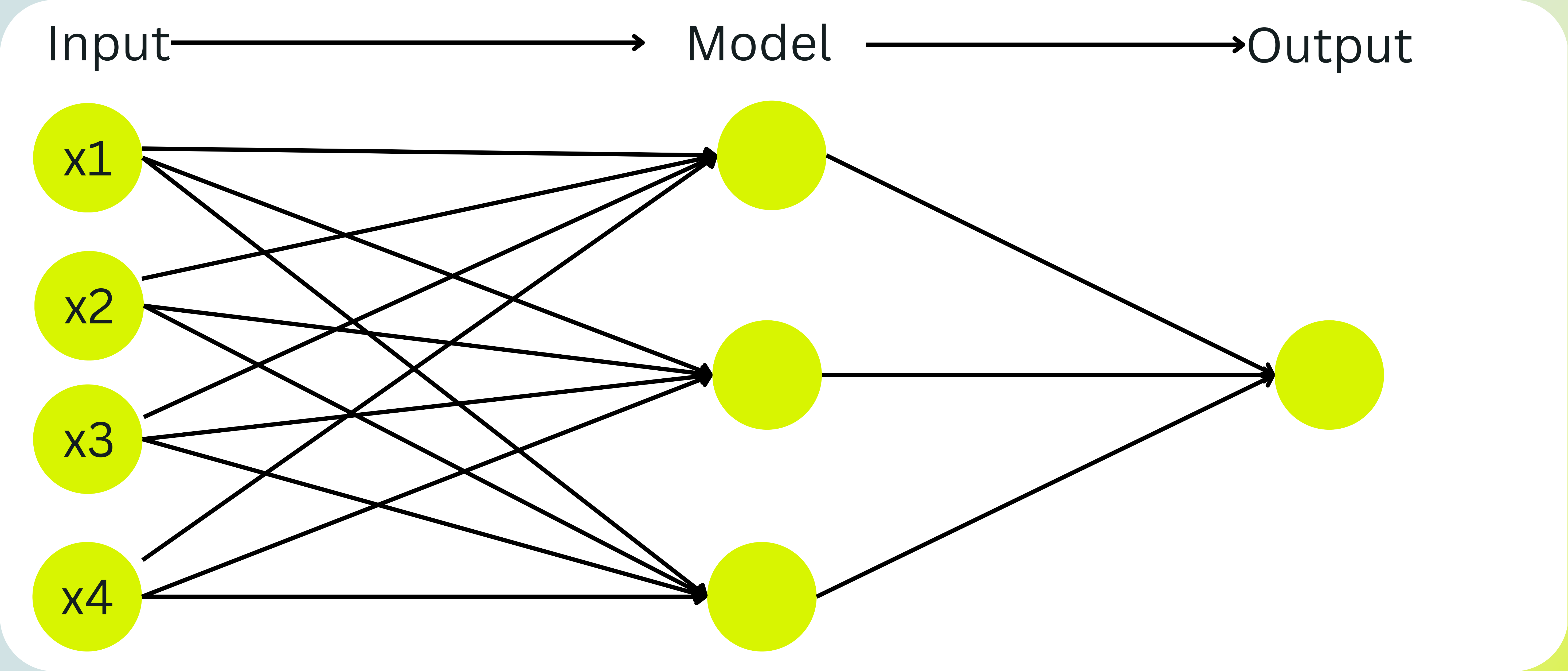
Perceptron

- simplest model of a neural network
- classification with two classes
- one activation function



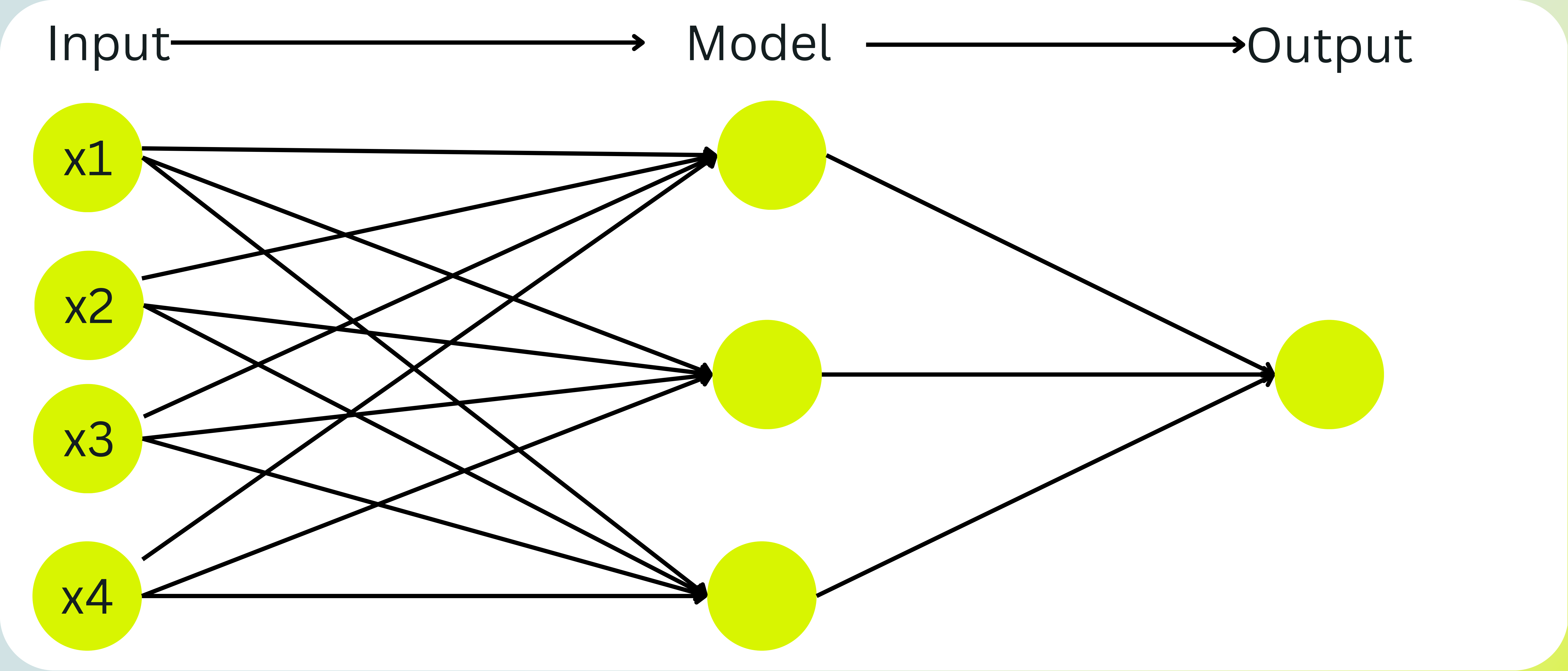
Multi-Layer Perceptron

- adds hidden layers
- solves limitations of Perceptron



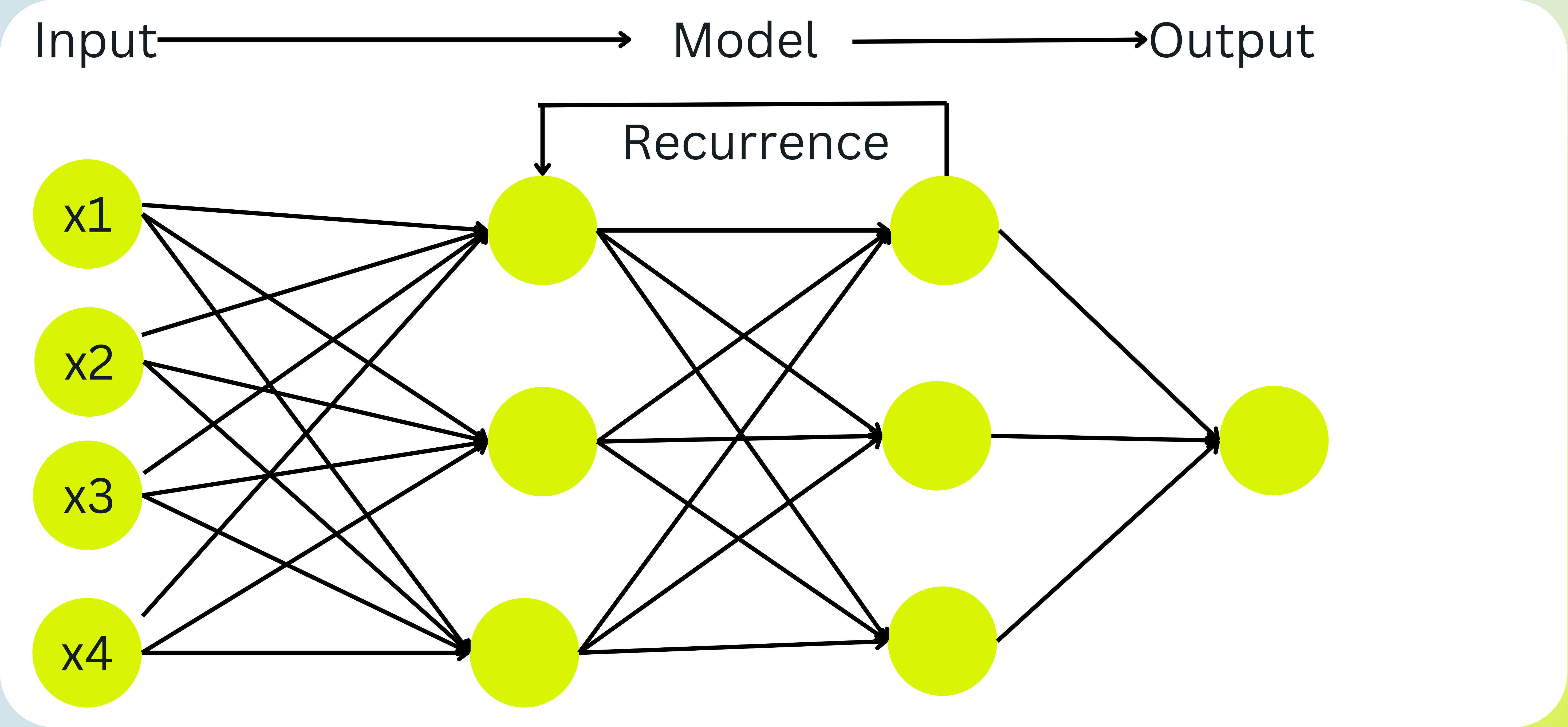
Multi-Layer Perceptron

- = **Feed Forward Network**
- dense layers



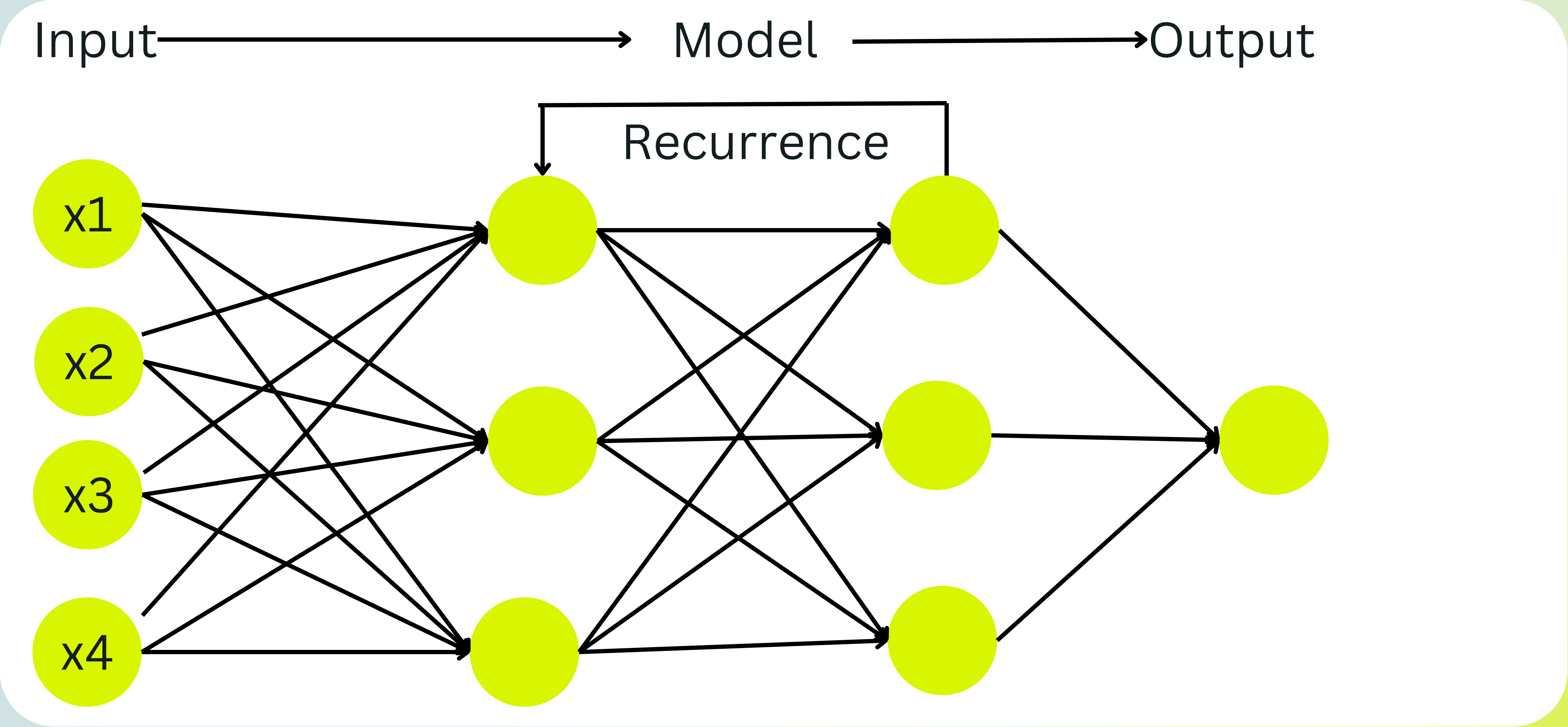
Recurrent Network

- enables temporal dependencies
- output can be used as input over multiple layers



Recurrent Network

- used e.g. in translation, in general for everything with timely dependencies
- only short time memory



Long short-Term Memory

- allows for longer-time memory
- complex structure including gates
- Form of RNN

gates:
Input Gate
Forget Gate
Output Gate
Candidate Cell State

New component:
Cell state -> Long-Term memory

Long short-Term Memory

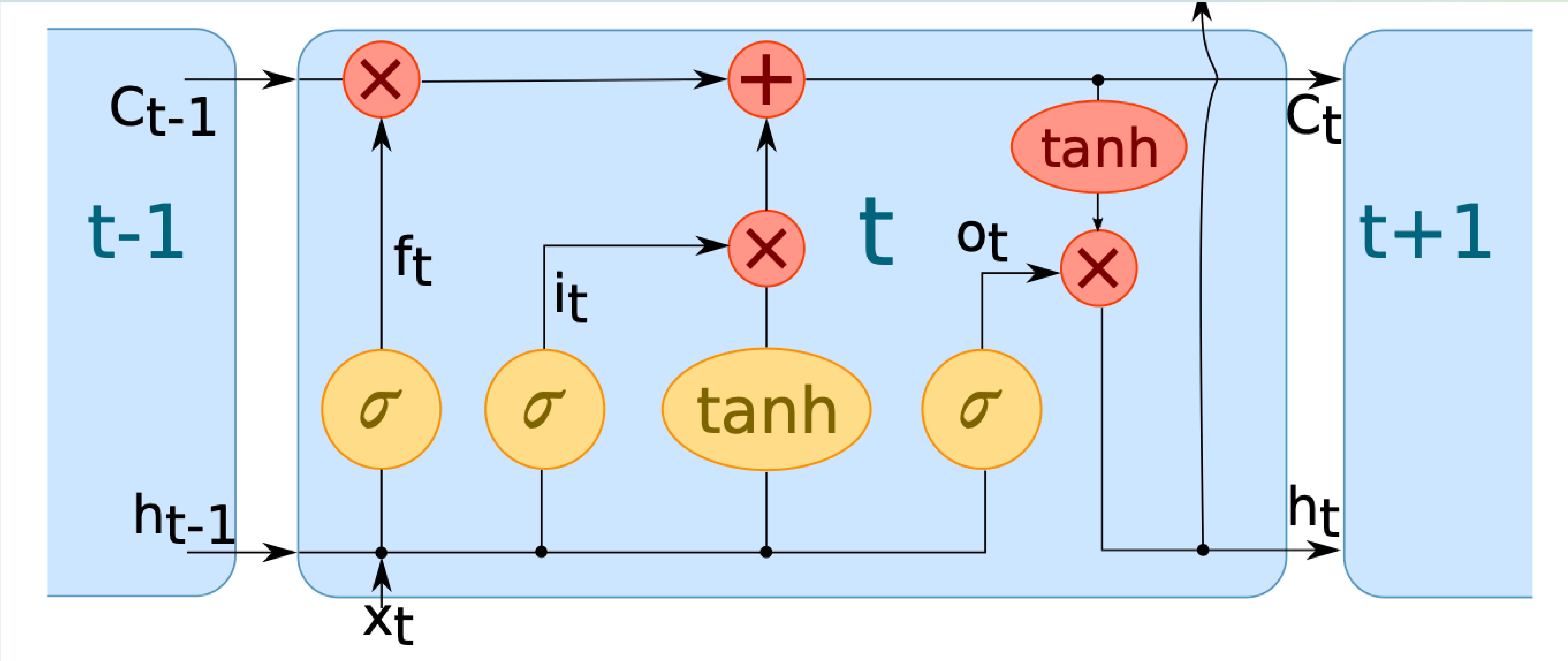


Figure by: Kläbe et al.

Convolutional Neural Network

Differences to FFN and RNN:

- not only dense layers
- neurons only consider input from within their receptive field
- mostly used for image and audio processing

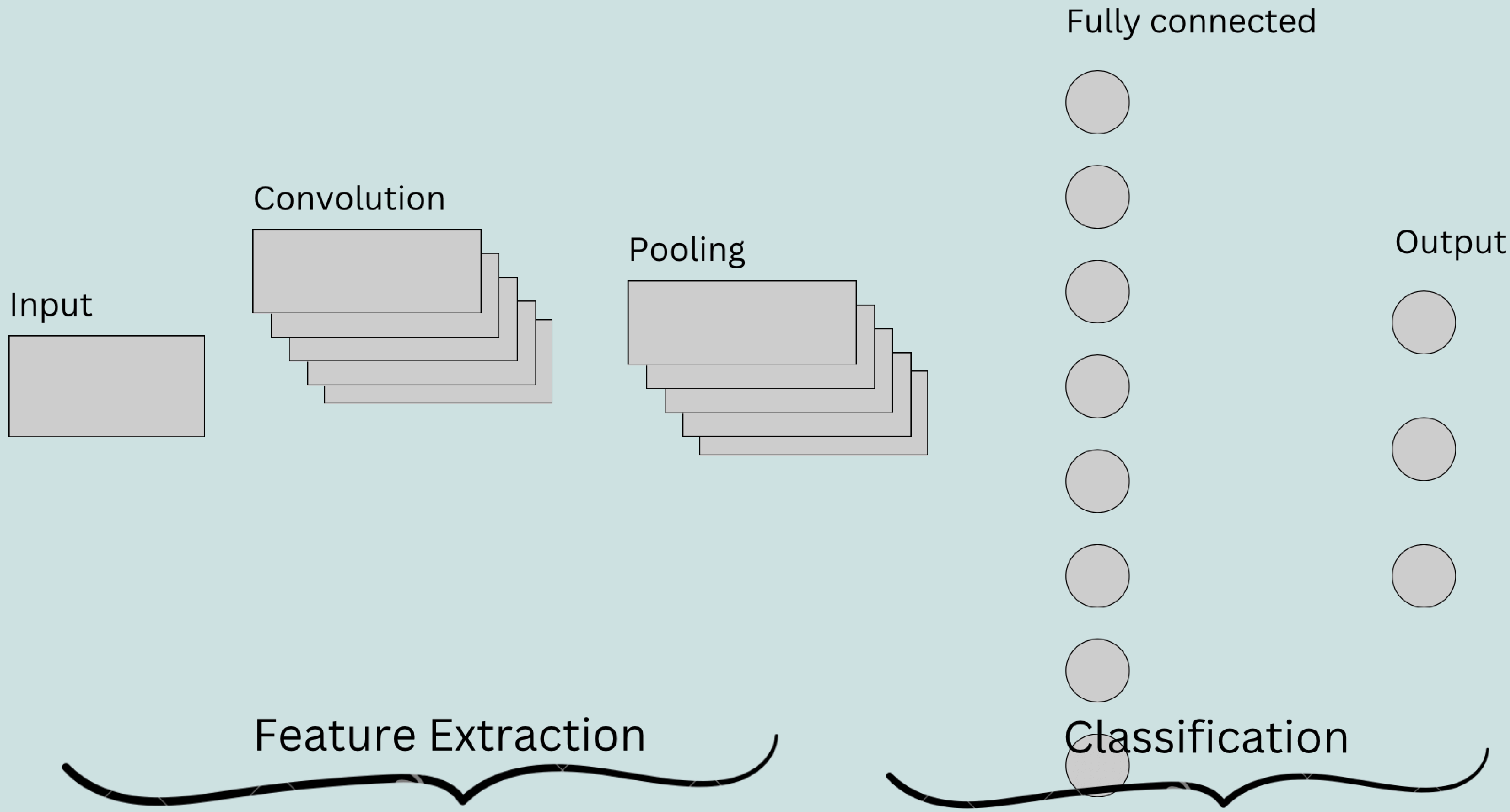


Figure inspired by: https://www.researchgate.net/publication/369963460_A_Lightweight_CORONA-NET_for_COVID-19_detection_in_X-ray_images

AI4DB vs. DB4AI

AI4DB:

- leverage db components with AI

DB4AI:

- uses DBSM`s for leveraging AI on existing data

ML-TO-SQL DESIGN

- highly portable
- using a ML-To-SQL Python Framework
- for interaction between data and the neural network we need a relative representation
- model is stored in a table

Relational model

Node: unique pair (Layer, Node)

Edge: (Layer_in, Node_in, Layer, Node)

for each edge:

Kernel weights Wi, Wf, Wc, Wo

Recurrent Kernel Weights: Ki, Kf, Kc, Ko

bias weights: bi, bf, bc, bo

=> model is represented in a table with 16 columns

The code

Definitions

```
import time
from ml2sql import ML2SQL

# PostgreSQL
import psycopg2 as pg
con = pg.connect(CONNECTIONSTRING)
backend = "postgres"
```

```
def run_query(query, con, should_print = True):
    cursor = con.cursor()
    cursor.execute(query)
    rs = cursor.fetchall()
    if not rs:
        print("Query result is empty")
    colnames = [desc[0] for desc in cursor.description]
    if should_print:
        print(colnames)
        for res in rs:
            print(res)
    cursor.close()

def run_update_query(query, con):
    cursor = con.cursor()
    cursor.execute(query)
    con.commit()
    cursor.close()
```

Code by: https://github.com/dbis-ilm/ML-To-SQL/blob/main/classification_example.ipynb

The code

Example model

```
import tensorflow as tf

model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(64, input_shape=(4,),activation='linear', bias_initializer=tf.keras.initializers.Random
    tf.keras.layers.Dense(8, input_shape=(4,),activation='relu', bias_initializer=tf.keras.initializers.RandomNor
    tf.keras.layers.Dense(2, input_shape=(4,),activation='sigmoid', bias_initializer=tf.keras.initializers.Random
    tf.keras.layers.Dense(1, activation='linear', bias_initializer=tf.keras.initializers.RandomNormal())
])

model.compile(loss='categorical_crossentropy')
```

Initialize ml2sql

```
translator = ML2SQL(con, backend, model)
```

Model import

```
model_table_name = "iris_model"
start_time = time.time()
queries = translator.model_to_relation(model_table_name)
for q in queries:
    run_update_query(q, con)
print("--- %s seconds ---" % (time.time() - start_time))
q = f"select * from {model_table_name}"
#run_query(q, con)
```

Code by: https://github.com/dbis-ilm/ML-To-SQL/blob/main/classification_example.ipynb

The code

Model join

```
# Table has to exist in database
tablename = "iris"
id_col_name = "id"
col_names = ["sepal_length", "sepal_width", "petal_length", "petal_width"]
output_col_name = "prediction"

# Build MJ query
input_query = f"Select * from {tablename}"
mj_query = translator.model_join_query(input_query, id_col_name, col_names, model_table_name, output_col_name)
# Run MJ
start_time = time.time()
run_query(mj_query, con, False)
print("--- %s seconds ---" % (time.time() - start_time))
```

Code by: https://github.com/dbis-ilm/ML-To-SQL/blob/main/classification_example.ipynb

Native ModelJoin Operator

requires changes in the db engine

implemented in Actinan`s Vextor x100 analytical query engine

typical two-phase join operation (e.g. hash join)

based on the Volcano iterator model

-> open(), next(), close()

Native ModelJoin Operator

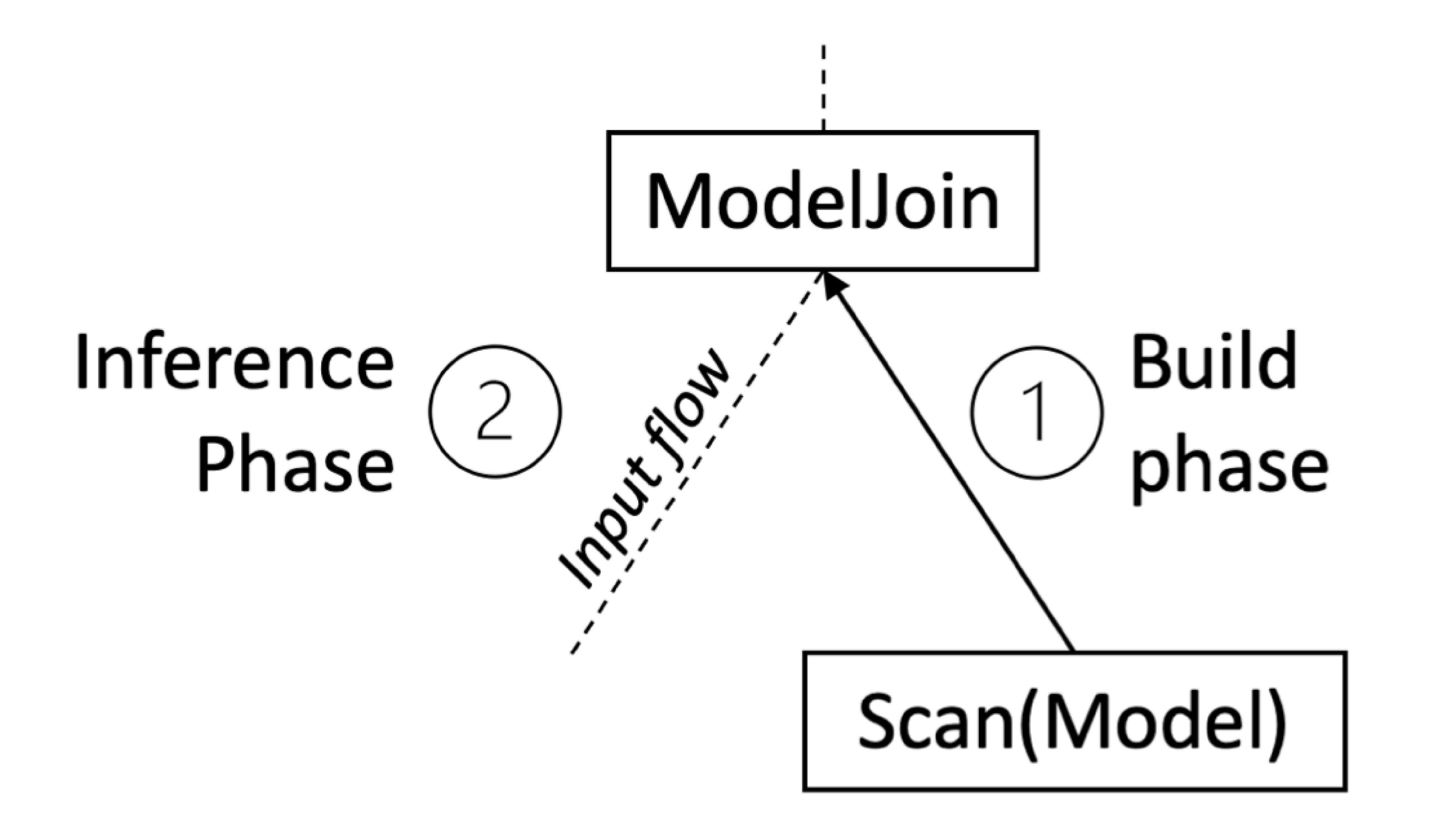


Figure by: Kläbe et al.

Native ModelJoin Operator

Columnar input is translated into input-matrix

Inference steps can be done for multiple inputs on the same time

Iterate over model layers and perform specific activation and layer forward functions

Native ModelJoin Operator

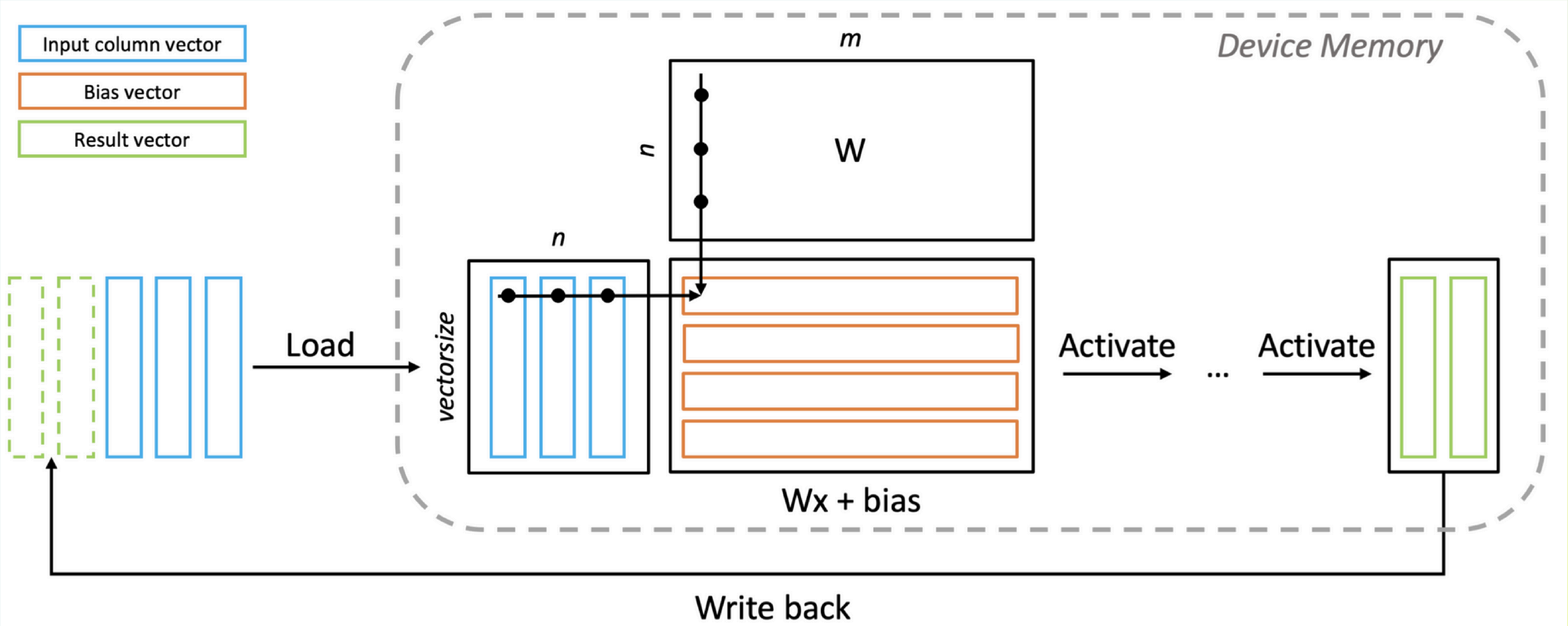


Figure by: Kläbe et al.

Conclusion

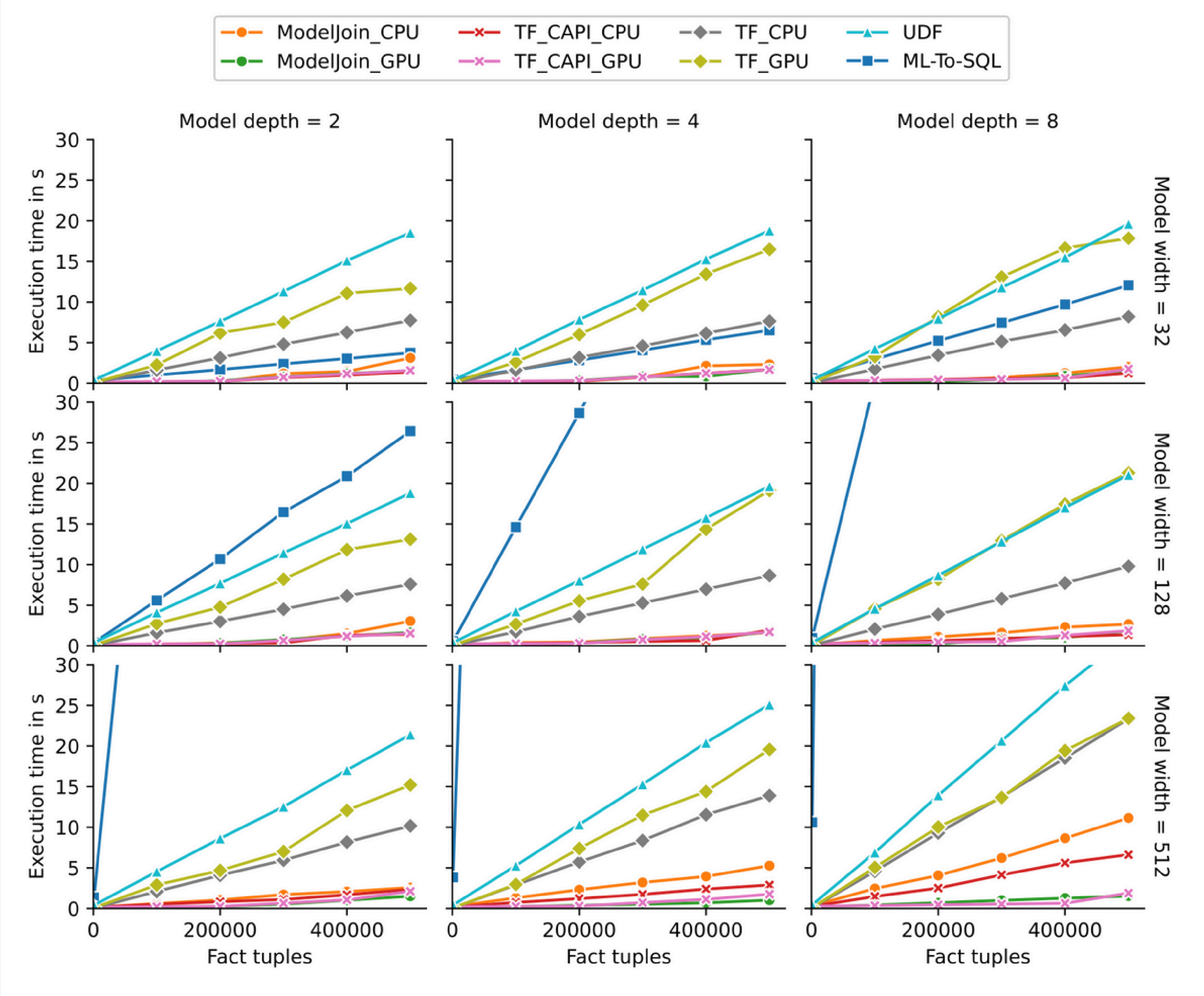


Figure by: Kläbe et al.

Conclusion

	ML-To-SQL	Native ModelJoin	TF(Python)	TF(C-API)	UDF
Performance (Small Models)	Good	Good	Medium	Good	Medium
Performance (Large Models)	Bad	Good	Bad	Good	Bad
Memory Consumption	Medium	Good	Bad	Good	Bad
Portability	Good	Bad	Good	Bad	Medium
Generalizability	Bad	Bad	Good	Good	Good

Figure by: Kläbe et al.

Conclusion

ML-TO-SQL:
high portability, easy-to-use API
very bad scalability
good for small datasets

Natural MODELJOIN operator:
high scalability,
small memory usage

References

Kläbe, Steffen; Hagedorn, Stefan; Sattler, Kai-Uwe
Exploration of approaches for in-database ML. - Konstanz : University of Konstanz. - 1 Online-Ressource (Seite 311-322)Online-Ausgabe: Advances in Database Technology - Volume 26 : proceedings of the 26th International Conference on Extending Database Technology (EDBT), 28th March-31st March, 2023, ISBN 978-3-89318-093-6