Prof. Dr. Alfred Benedikt Brendel
Chair of Business Information Systems, esp. Intelligent Systems and Services

Data Science: Advanced Analytics
Deep Learning

Dresden // 14.06.2023
Sommersemester 2023

TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR OF BUSINESS INFORMATION SYSTEMS
ESP. INTELLIGENT SYSTEMS AND SERVICES

DRESDEN concept

Prof. Dr. Alfred Benedikt Brendel and Prof. Dr. Kai Heinrich
Chair of Business Information Systems, esp. Intelligent Systems and Services

Deep Learning
ANN Recap and Interpretation Problem

Artificial Neural Networks (ANN)
Idea & Design of ANN

Idea

- Introduce hidden layers that act as artificial features that help to model the data better (do the calculations)
- You can think of the layers as new variables calculated from the input or previous hidden units

Elements

– **Layers:**
  - Input Layer: the features you want to use for prediction go in here
  - Hidden Layer(s): computational units creating artificial features
  - Output Layer: puts out prediction based on computations in hidden layer

– **Units:**
  - Input units: Nodes in the input layer, you have as many nodes as input varibales + a constant (=bias)
  - Hidden units: Nodes in the various hidden layer, can be freely chosen to fit the problem
  - Output units: Nodes representing the various outputs

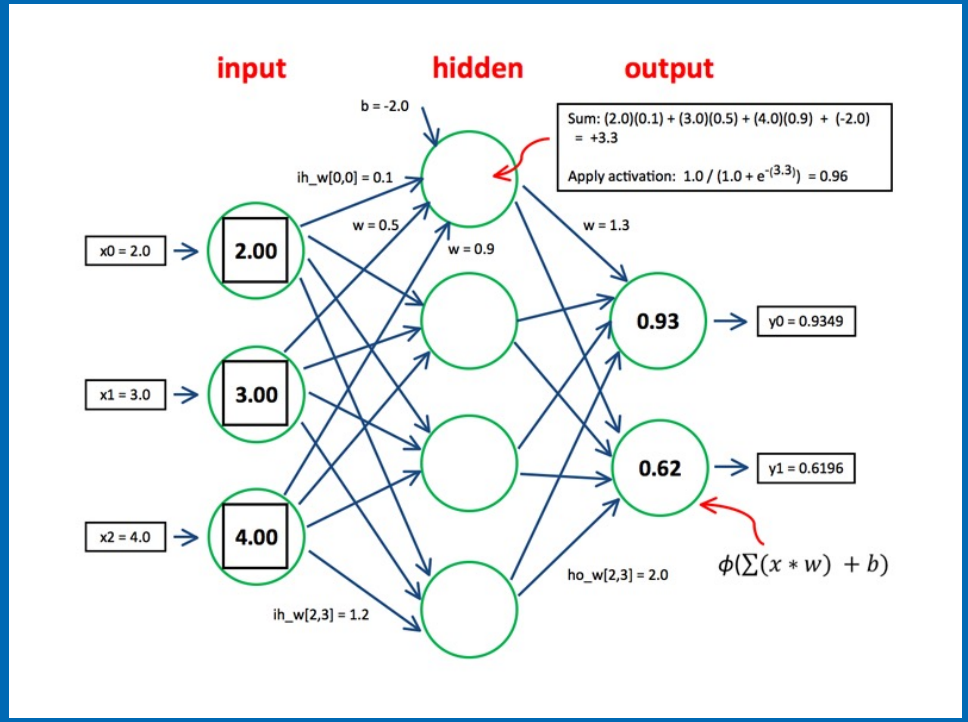
– **Connection:**
  - Each node is connected to the succeeding node (one direction only)
  - Each connection has a weight that need to be learned from the data

Artificial Neural Networks (ANN)

Recap: Classic ANN architecture and example calculation

Example ANN

1 hidden layer, 3 input nodes, 4 hidden nodes, 2 output nodes



Artificial Neural Networks (ANN)
Recap: Example Network description

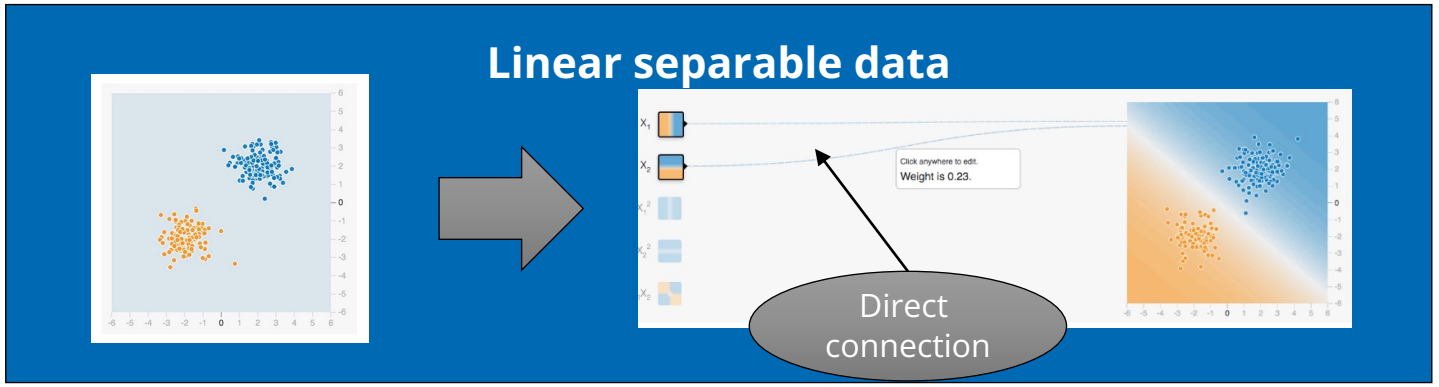
Structure:
- An input layer with 3 input units
- A hidden layer with 4 hidden units
- An output layer with 2 output units

Connections and weights:
- Every unit of a layer is connected to every unit of the following layer
- Those connections are not all of the same strength, instead they are weighted
- Those weights are the parameter of an ANN model and need to be learned

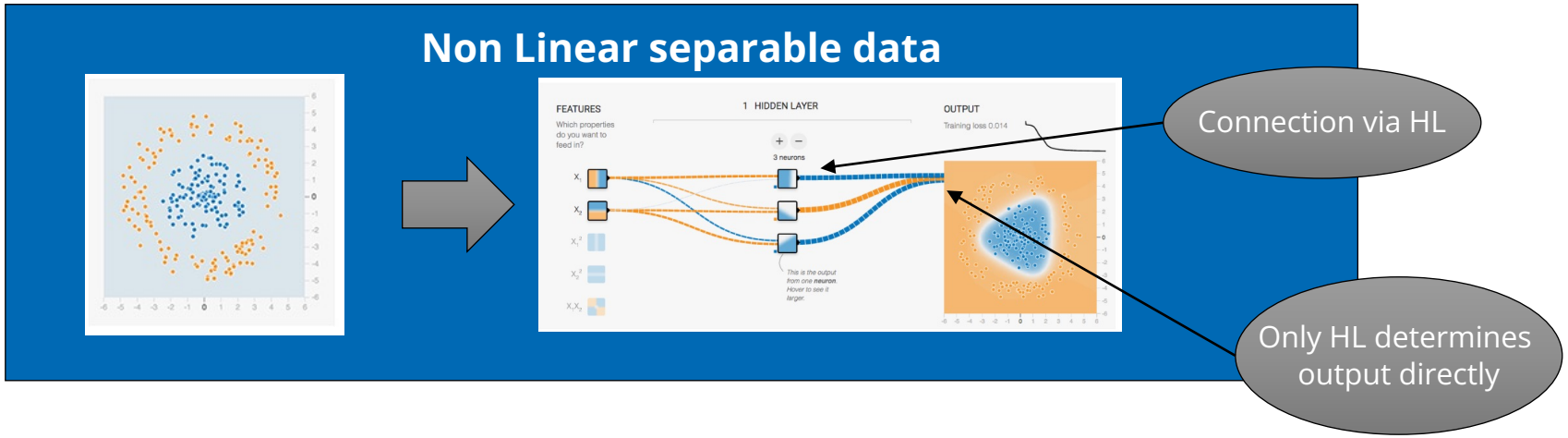
Feed forward run:
- Once the weights are learned, a new input vector can be fed into the ANN and will be propagated through the net until an output is generated
- Every unit (except the input units) are calculated by applying a function (mostly sigmoid or tanh) to the weighted linear combination of all units from the previous layer
- See the picture in the previous slide for an example calculation
- Play with ANN architectures (highly encouraged!!): http://playground.tensorflow.org/

Artificial Neural Networks (ANN)
Weights in ANN: Examples from Tensorflow Playground

Demonstration Example with direct interpretable weights (no HL):



Demonstration Example with HL:



TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR OF BUSINESS INFORMATION SYSTEMS
ESP. INTELLIGENT SYSTEMS AND SERVICES

DRESDEN concept

Prof. Dr. Alfred Benedikt Brendel and Dr. Kai Heinrich
Chair of Business Information Systems, esp. Intelligent Systems and Services

Deep Learning
Convolutional Neural Networks

Task: Pattern recognition

Answer the question: What is on this picture?

Approach (greatly simplified):

1. **Extract features**:
   Straight lines or circles in the image
2. **Classify** the combination of extracted **features**:
   Combination of lines and circles → Face!

Traditionally: Hand crafted feature extractors and trainable classifiers to accomplish pattern recognition

Problem: **Lacking generalization ability in hand crafted algorithms**

Possible solution: Fully connected Feed forward network that learns to extract features

Problems:

- **Large input data**
- **Input variances**

ANN Problems: Large input data

Input: 3 values per pixel (RGB-Channels)
→ 1280 * 1920 = 2.457.600 pixels

First fully connected hidden layer: For example 100 nodes
→ 245.760.000 weights only for the first layer

More layers necessary to complete the task!

Result:

- Large training dataset is needed to prevent overfitting
- Large memory requirement

ANN Problems: Input variances

Assumption: Trained ANN that can recognize the image

Can it also recognize these images?

Normalization required before processing by the ANN:
- Align faces in the middle of the image (same orientation)
- Normalize the size of the images

Complete normalization is not possible for every task

With a large enough dataset (needs to cover all possible variations) a fully connected network could learn to ignore variances

→ Many weights would be identical (or similar) so that same shapes could be recognized across the image
→ **Lots of redundance in the weights**

LeCun, Yann; Bengio, Yoshua: Convolutional Networks for Images, Speech, and Time-Series. In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks* MIT Press (1995).

Solution: Self learning Feature Extractor: CNN

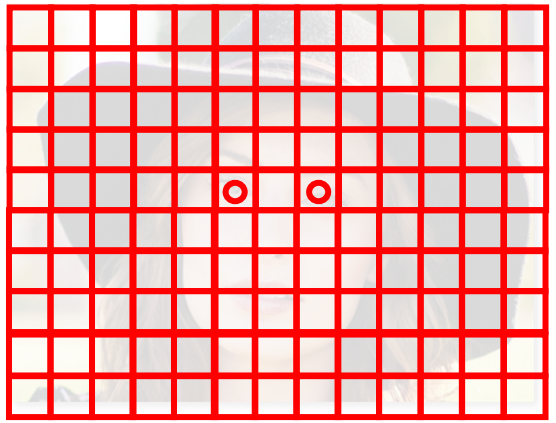
Nodes that look at a small part of the image (for example 3x3 pixels)

Determine if a feature occurs in this region

This is called „**filter**“: One layer that looks for one feature across the image

Slide the filter across the image and repeat this process (**convolution**)
→ Name origin: Convolutional Neural Network

Stack multiple filters into one network



Filter looking for circles in small regions

Small amount of weights in each filter:

- Each pixel has 3 values (RGB)
- Filter scans 9 pixels (in this example)
- Only 27 Weights per filter

No need to manually define filters: **Learning via Backpropagation possible**

LeCun, Yann; Bengio, Yoshua: Convolutional Networks for Images, Speech, and Time-Series. In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks* MIT Press (1995).

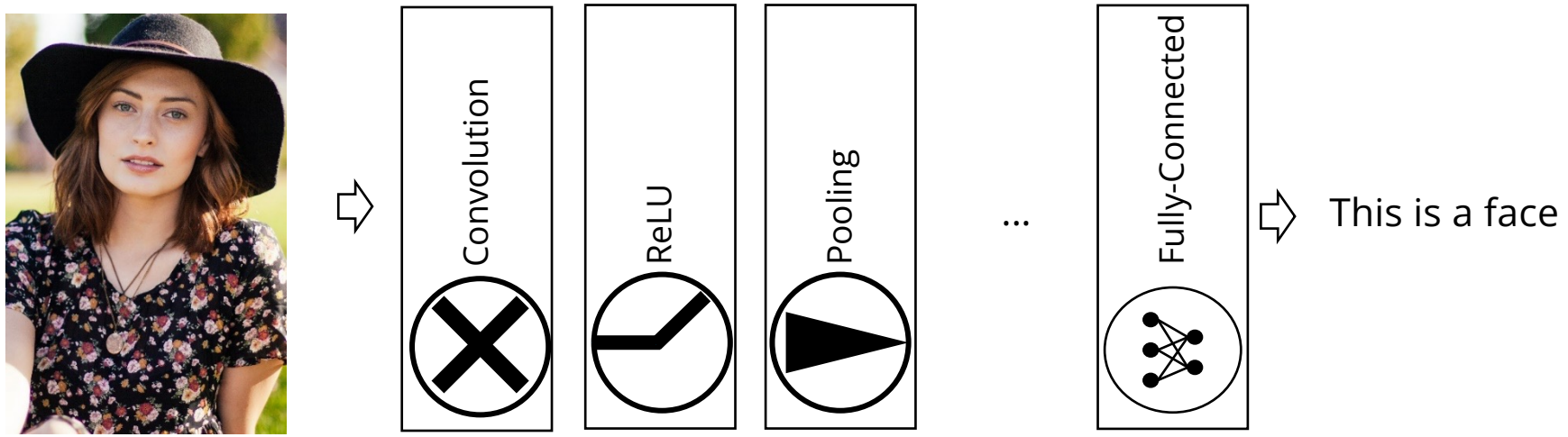
Convolutional Neural Network: Overview

Stack layers to build a complete CNN

One type of layer is the previously mentioned „convolutional layer“

In total there are 4 types of layers:

- **Convolutional Layer** (Extract features)
- **Pooling Layer** (Reduce size)
- **ReLU Layer** (Remove negative values → disregard information about missing features)
- **Fully Connected Layer** (Classify features)

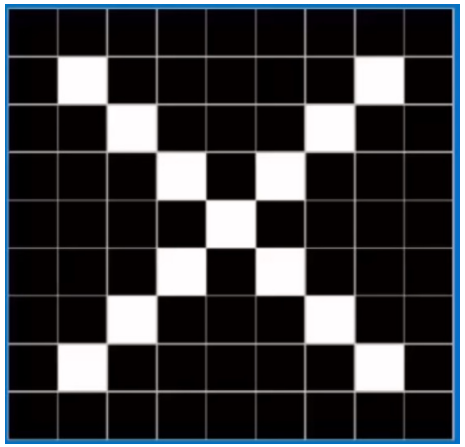


LeCun, Yann; Bengio, Yoshua: Convolutional Networks for Images, Speech, and Time-Series. In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks* MIT Press (1995).

Example to explain layers

We will take a look at the following image

Question: Is this an X or a 0?



-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

The computer can only see the numbers representing the color

Note: This example only has one color channel (black or white) to keep it simple

Images from: Rohrer, Brandon: How Convolutional Neural Networks work. https://www.youtube.com/watch?v=FmpDIaiMIeA

Convolutional Layer

First hyperparameter: **Receptive Field:** Spatial extend of the filter

Apply the filter to a region:

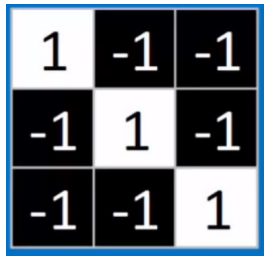
- Multiply filter value by image value
- Calculate average of the (here) 9 fields

Slide the filter across the image by S pixels
(Second hyperparameter: **Stride**)

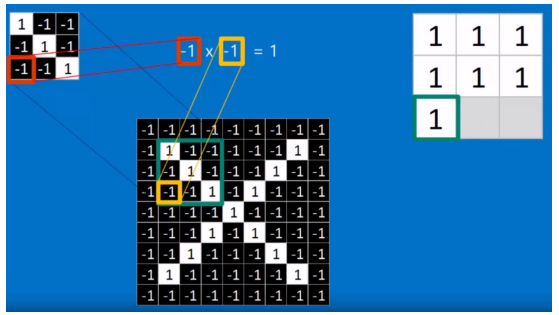
Third hyperparameter: Optional **zero padding**
to allow for more stride and receptive field combinations

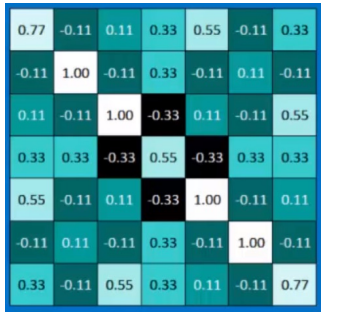
Fourth hyperparameter: **Depth:** Number of filters per layer

Intuitively: Use small filters to extract features in a small region of the image. Slide (convolve) the filter across the image to check for this feature everywhere. Stack multiple filters to detect multiple features.



Example 3x3 filter: diagonal lines


Apply the filter by multiplying filter values by image values



Result of the filter application
(averages of the 9 surrounding cells)

Convolutional Layer

Accepts input of size $\boldsymbol{W_1 \times H_1 \times D_1}$
(width times height times number of color channels for images)

Requires four hyperparameters:

- **Number of filters $\boldsymbol{K}$**
- **Their spatial extent (receptive field) $\boldsymbol{F}$**
- **The stride $\boldsymbol{S}$**
- **The amount of zero padding $\boldsymbol{P}$**

Produces an output of size: $\boldsymbol{W_2 \times H_2 \times D_2}$

- $\boldsymbol{W_2 = \frac{W_1 - F + 2P}{S} + 1}$
- $\boldsymbol{H_2 = \frac{H_1 - F + 2P}{S} + 1}$
- $\boldsymbol{D_2 = K}$

Number of weights: $\boldsymbol{(F * F * D_1) * K}$

Common values for hyperparameters: $F = 3, S = 1, P = 1$

Pooling Layer

Goal: Reduce spatial size of the data
→ **Reduce amount of parameters to control overfitting**

Implementation: Take the highest of N values
(MAX operation)

Common form: Take 2x2 values and keep the highest
→ Size reduction by 75%

During backpropagation, propagate error to the highest value

Intuitively: Condense small regions into single pixels to reduce data and parameters by applying specific mathematical operations. (Mostly MAX)

Pooling Layer

Accepts an input of size $W_1 \times H_1 \times D_1$

Requires **two hyperparameters:**

- **Spatial extend** F
- **Stride** S

Produces output of size $W_2 \times H_2 \times D_2$

- $W_2 = \frac{W_1 - F}{S} + 1$
- $H_2 = \frac{H_1 - F}{S} + 1$
- $D_2 = D_1$

No additional parameters (fixed function)

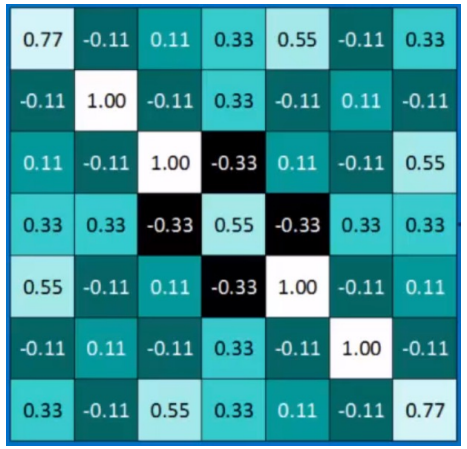
ReLU-Layer

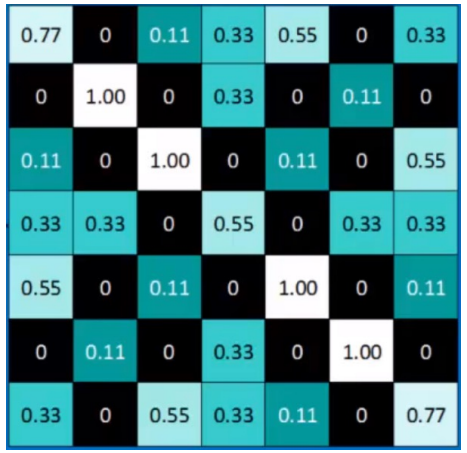
Goal: **Remove negative values**

Implementation:

- Apply Rectified Linear Units (ReLU)
- Function: $\max(0, x)$
  → Set negative values to 0, keep the others

No additional parameters (fixed function)





Intuitively: Remove negative values because we only want information about present features, not about absent features.

Fully-connected Layer

Simple **feed forward layer** that has **connections to all inputs**

Output: probabilities for different classes
(by using the Softmax activation function)

Hyperparameters:

- Number of Fully-connected layers
- Number of nodes in each layer

Intuitively: Learn connection between extracted features and possible classes.

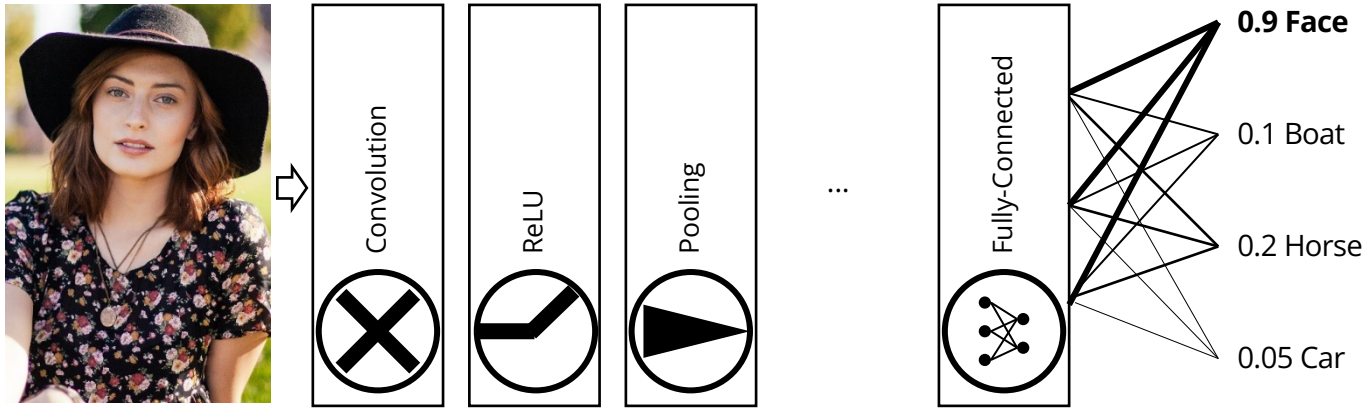
CNN Patterns

Multiple repetition of layers to produce correct output classes

Most common pattern:
INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC

- * Indicates repetition
- ? Indicates an optional layer
- N >= 0 (usually N <= 3), M >= 0, K >= 0 (usually K < 3)

Note: It is often good enough to download a pretrained model and finetune it on your data.



Stanford CS231n Class notes: Convolutional Neural Networks for Visual Recognition: http://cs231n.github.io/convolutional-networks/

Applications of CNNs

Images (matrix of color channel values):

- Detect and interpret road signs
- Identify persons for security (e.g. Face ID)
- Handwriting recognition

Sound (matrix of frequency intensities):

- Speech recognition
- Language classification

Time series (matrix of time related values):

- Predict machine failures from multiple sensor values over time

References & WebLinks

- Goodfellow, I.; Bengio, Y.; Courville, A.: Deep Learning (2016), P. 330ff.
- LeCun, Yann; Bengio, Yoshua: Convolutional Networks for Images, Speech, and Time-Series. In M. A. Arbib (Ed.), *The handbook of brain theory and neural networks* MIT Press (1995).
- Rohrer, Brandon: How Convolutional Neural Networks work. https://www.youtube.com/watch?v=FmpDIaiMIeA
- Stanford CS231n Class notes: Convolutional Neural Networks for Visual Recognition: http://cs231n.github.io/convolutional-networks/

TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR OF BUSINESS INFORMATION SYSTEMS
ESP. INTELLIGENT SYSTEMS AND SERVICES

DRESDEN concept

Prof. Dr. Alfred Benedikt Brendel and Dr. Kai Heinrich
Chair of Business Information Systems, esp. Intelligent Systems and Services

Deep Learning
Convolutional Neural Networks - MNIST

Steps to classify images

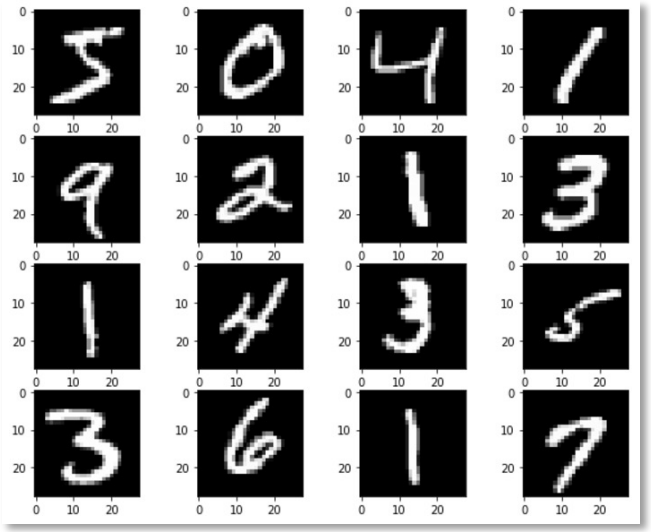
1. Load the dataset
2. Prepare the data for the keras model
3. Build the layers of the keras model
4. Train the model with a training subset of the data
5. Test the trained model with a testing subset of the data

1. Load the data

```
from keras.datasets import mnist
import matplotlib.pyplot as plt
import matplotlib

[...]
(x_train_2d, y_train_nums), (x_test_2d, y_test_nums) = mnist.load_data()

for i in range(16):
    plt.subplot(4, 4, i+1)
    plt.imshow(x_train_2d[i], cmap='gray’)
[...]
```



2. Data preparation – shape

MNIST Dataset (http://yann.lecun.com/exdb/mnist/) contains 60.000 labeled training images and 10.000 labeled test images

Input preparation:

- Each image is 28x28 and has only one color value (black/white) between 0 and 255
  → Each image is only 2D
- The Keras Conv2D Layer needs 3D inputs, to account for multiple channels per pixel
- Values are normalized between 0 and 1

```
x_train = (x_train_2d.reshape(-1, 28, 28, 1) / 255).astype('float32')
x_test = (x_test_2d.reshape(-1, 28, 28, 1) / 255).astype('float32')
```

Output preparation:

- Labels are integers for each image
- We transform these numbers into one-hot vectors: Filled with zeroes and one 1 at the respective index

```
y_train = keras.utils.to_categorical(y_train_nums, 10)
y_test = keras.utils.to_categorical(y_test_nums, 10)
```

3. Build layers

Our CNN Architecture contains multiple layer types:

- Conv2D: CNN Layer with 32 filters, shape 3x3 with ReLU Layer appended
- MaxPooling2D: Pooling layer with shape 3x3
- Dropout: Randomly drop 20% of neurons to prevent overfitting
- Flatten: Prepare filter results for classification
- Dense: Fully connected layers for classification

```
from keras.models import Sequential
from keras.layers import Dense, Flatten, Dropout
from keras.layers import Conv2D, MaxPooling2D

model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=(x_train.shape[1:]), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

4. Train the model

Train the model on the training data

Note: This network needs more resources than previous examples

```
model.fit(x_train, y_train,
          epochs=5,
          verbose=1,
          validation_data=(x_test, y_test))
```

5. Test the model

Compute the loss and accuracy for the test data

- Test loss: 0.029
- Test accuracy: 0.991

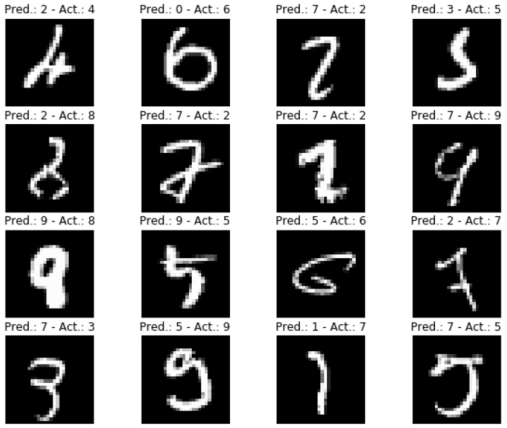
```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

5. Test the model – inspect error cases

Additionally we want to take a look at some error cases

```
predictions = np.argmax(model.predict(x_test), axis=1)
false_predictions = y_test_nums != predictions
error_indices = np.where(false_predictions)[0]

for i in range(16):
    error_index = error_indices[i]
    plt.subplot(4, 4, i+1)
    plt.imshow(x_test_2d[error_index], cmap='gray')
    plt.axis('off')
    plt.title('Pred.: ' + str(predictions[error_index]) + ' - Act.: ' + str(y_test_nums[error_index]))
```



Result: Some of the error cases are barely recognizable by the human eye, others could have been predicted by our model.
→ Longer training time or more layers could help

TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR OF BUSINESS INFORMATION SYSTEMS
ESP. INTELLIGENT SYSTEMS AND SERVICES

DRESDEN concept

Prof. Dr. Alfred Benedikt Brendel and Dr. Kai Heinrich
Chair of Business Information Systems, esp. Intelligent Systems and Services

Deep Learning
Long Short-Term Memory (LSTM)

Challenge: Sequence related problems

Think about the following questions:

- What will the stock price of Nike be tomorrow?
  - You need information from the past: Where was it yesterday? Where was it before that?
- What is the letter after M?
  - In your head you will probably think: „H, I, J, K, L, M, N… It‘s N!“
  - You can see, that your **brain works with sequences of data**.
- How will this sentence end:
  - „I was in France. It was very nice there! I even learned to speak…“
  - As a human, you can **remember** that we were talking about France at the beginning of the sentence, so the language will probably be french.

All these questions are about a sequence of elements, sometimes time related.

How can these sequences be incorporated into an ANN?

Possible Solution: Sliding window

Simple Feed Forward Networks with inputs from the last n periods

Selecting the optimal window size is crucial:

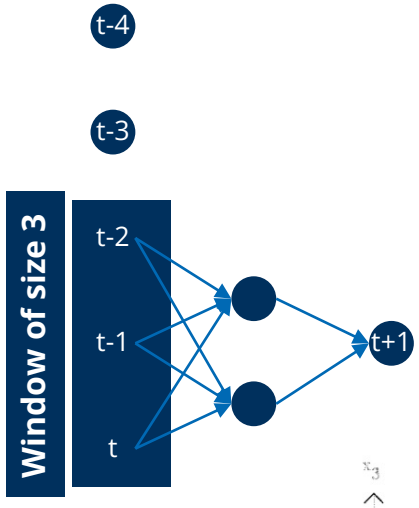
–Too big leads to noise

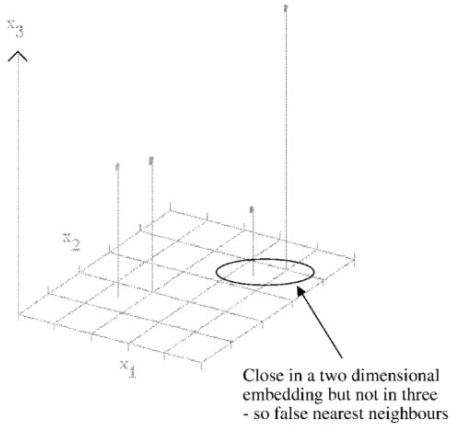
–Too small leads to incorrect results (False nearest Neighbours) as not enough information is captured in the window

Problems

–Values before the window are completly forgotten

–Window size has to be defined by the modeler, the network will not learn the correct size on its own.

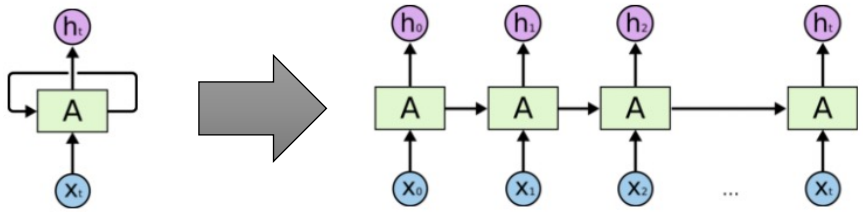




Frank, Ray J. ; Davey, Neil ; Hunt, Stephen P.: Time series prediction and neural networks. In: *Journal of intelligent and robotic systems* Vol. 31 (2001), Nr. 1–3, P. 91–103

Better solution: Recurrent Neural Networks

To incorporate previous results, one can feed the result from the last period as input to the next period. This endless loop is hard to reason about, so we „unroll“ the recursive step to create a chain of networks. As a result, we can receive the output h that is dependant on the current input x and the previous steps from this network. Selecting the optimal window size is crucial:



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

x_t: Input from period t
A: Neural Network
h_t: Output from period t

Note: It is not obligatory to have an ouput after each step. If we want to have one output for a sequence (for example sentiment of a complete sentence), we will only need one output at the end

Recurrent Neural Networks: BPTT

To learn in these Recurrent Networks, the Backpropagation Algorithm is used again

Remember: Goal of Backpropagation is finding the influx that a single weight has on the loss of the current model.

To apply Backpropagation to a RNN, we have to **unfold the network** first. After this, the algorithm can be applied as usual until we reach the start of the sequence.

The application of the Backpropagation Algorithm on the unfolded network is called **Backpropagation Through Time (BPTT)**

This yields the gradients for each weight in the neural network with respect to the total loss, which can be used to adapt the weights (for example with Gradient Descent).

Problems: Vanishing or Exploding Gradients

When using Backpropagation, we are applying the chain rule of derivation recursively until we reach the end of the sequence.

The chain rule is computed by multiplying the derivatives of the activation functions in each node.

As we are passing the same nodes multiple times (BPTT) we are multiplying these derivatives with themselves. This leads to 2 problems: Vanishing Gradients for gradients below 1 and Exploding Gradients for gradients above 1

- $\lim_{n\to\infty} x^n = \infty$ with $x > 1$
- $\lim_{n\to\infty} x^n = 0$ with $|x| < 1$ and $|x| >= 0$

Intuitively this means that long term memories have either

- **too much influence** on the current result because their values are used in every iteration (gradients above 1; happens only rarely)
- **or very little influence** because they are too far in the past (gradients below 1; happens most of the time), for example: Words from 5 sentences before are not taken into account when predicting the next word.

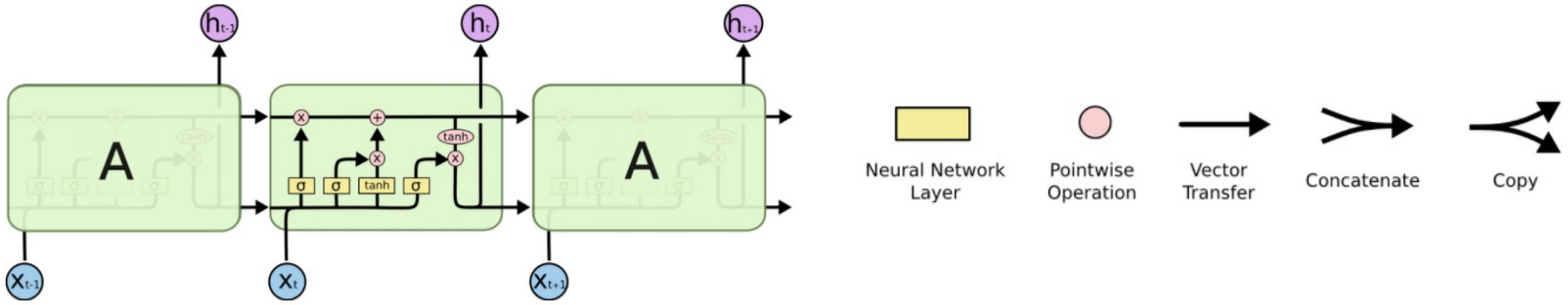
Solution to Vanishing Gradients: LSTM

Long Short Term Memory
Every node has internal state (like a memory cell)

In every iteration, the node decides with the help of different layers:

- Which parts of the state need to be forgotten
- Which parts need to be updated
- Which parts are used when computing the next output

This way, the network can decide on its own, when it should forget old information
→ No more vanishing gradients



Repeating module of an LSTM

The explanation of LSTMs and the images are taken from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM-Concepts: Gates

LSTM heavily uses gates in its architecure

A gate is used to modify values in a vector to only let specific values or adapted values through.

Sigmoid Gate: When using the Sigmoid activation function, the result is a vector of numbers between 0 and 1. When multiplying this vector pointwise with another vector, we can remove or deminish values in the other vector.

$$\begin{pmatrix}5\\4\\3\end{pmatrix} \times \begin{pmatrix}0\\0.25\\1\end{pmatrix} = \begin{pmatrix}0\\1\\3\end{pmatrix}$$

Input Sigmoid output Modified Vector

These gates are used to „filter“ values in vectors.

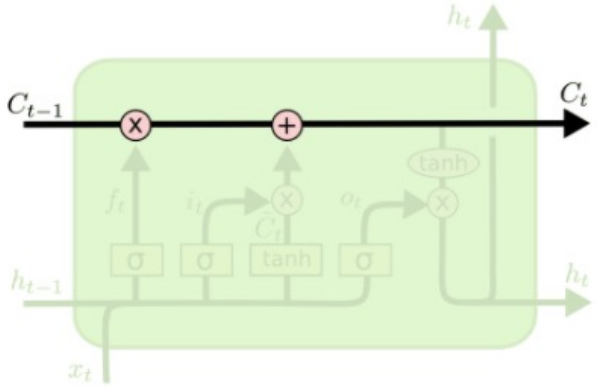
LSTMs use 3 different gate layers to modify the internal state (following slides)

LSTM-Concepts: Internal State

The top lane in the image is representing the internal state of the LSTM module (the „memory“)

This internal state C flows through the 3 gates and influences the output h_t of this sequence step

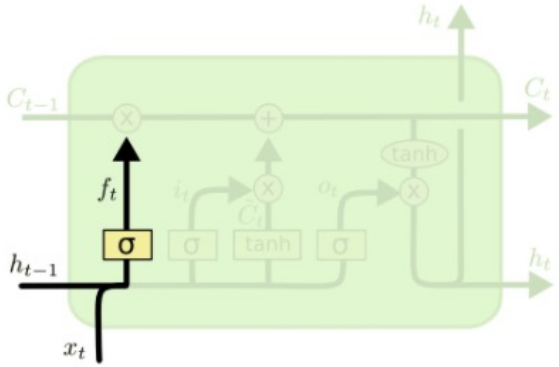
The input x_t and the previous output h_{t-1} is used to determine, which part of the state needs to be forgotten, updated, or used for the current output $h(t)$



LSTM-Concepts: Forget Gate

At first the internal state flows through the forget layer.

This tells the module, which information is no longer needed.
A Sigmoid layer (separate ANN) learns, when some information is no longer needed.
The ouput of this layer is computet according to this formula



$$f_t = \sigma\left(W_f[h_{t-1}, x_t] + b_f\right)$$

with W_f as weights of this layer, b_f as constant bias of this layer and σ as sigmoid activation function

The output f_t is then multiplied pointwise with the internal state to adapt it.

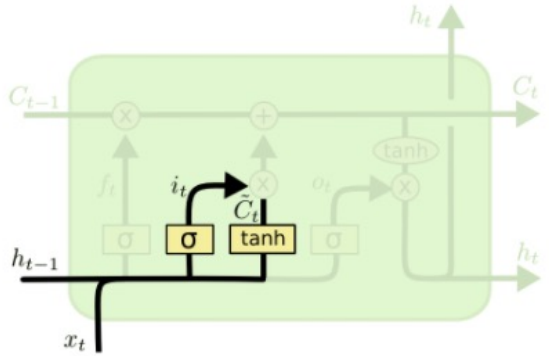
Intuitively: Given some new input and the previous output, this layer knows, what the complete module needs to forget (remove from internal state)

LSTM-Concepts: Input Gate + Candidate Layer

The next gate consists of 2 parts:

The Input Gate is a Sigmoid gate that updates the internal state

The Candidate Layer uses the Tanh activation to create new values that need to be persisted in the state.



The ouput of this layer is computet according to this formula

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$
$$\tilde{C}_t = tanh(W_C[h_{t-1}, x_t] + b_C)$$

with W as weights of this layer, b as constant bias of this layer and σ or $tanh$ as activation functions

The output of the pointwise multiplication of these 2 parts is then added to the internal state to adapt it.

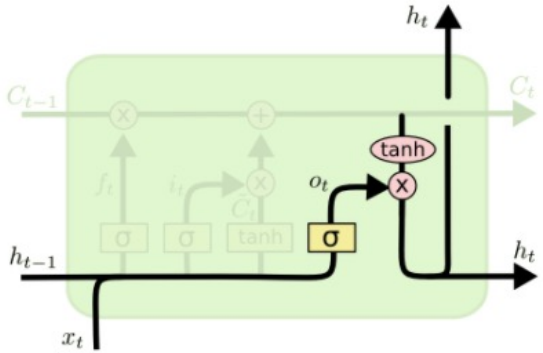
Intuitively: Given some new input and the previous output, this layer knows, what memories need to be adapted and what needs to be remembered for the future.

LSTM-Concepts: Output Gate

Now that we adapted the internal state depending on the current input and the last output, we can decide what needs to be the new output.

First we apply a Sigmoid layer to decide which part of the state will be in the output

This result is multiplied by the Tanh result of the internal state, so that values are between -1 and 1



$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$
$$h_t = o_t \tanh(C_t)$$

with W as weights of this layer, b as constant bias of this layer and σ or $tanh$ as activation functions

Intuitively: Given some new input and the previous output, this layer knows, what memories are important for the current output.

Example: Children‘s book

Imagine a (very) simple children‘s book: It contains sentences in the form of:

- Jack sees Jane.
- Jane sees Bobby.
- Jack sees Bobby.

It only contains 5 different „tokens“: Bobby, Jack, Jane, sees, . (the dot)

The state and the outputs of our LSTM network are indicating the likelyhood for the word beeing the next one in the book, for example:

$$\begin{pmatrix} 0.8 \\ 0.8 \\ -0.9 \\ -0.5 \\ 0 \end{pmatrix}$$ → The next word is either Bobby or Jack

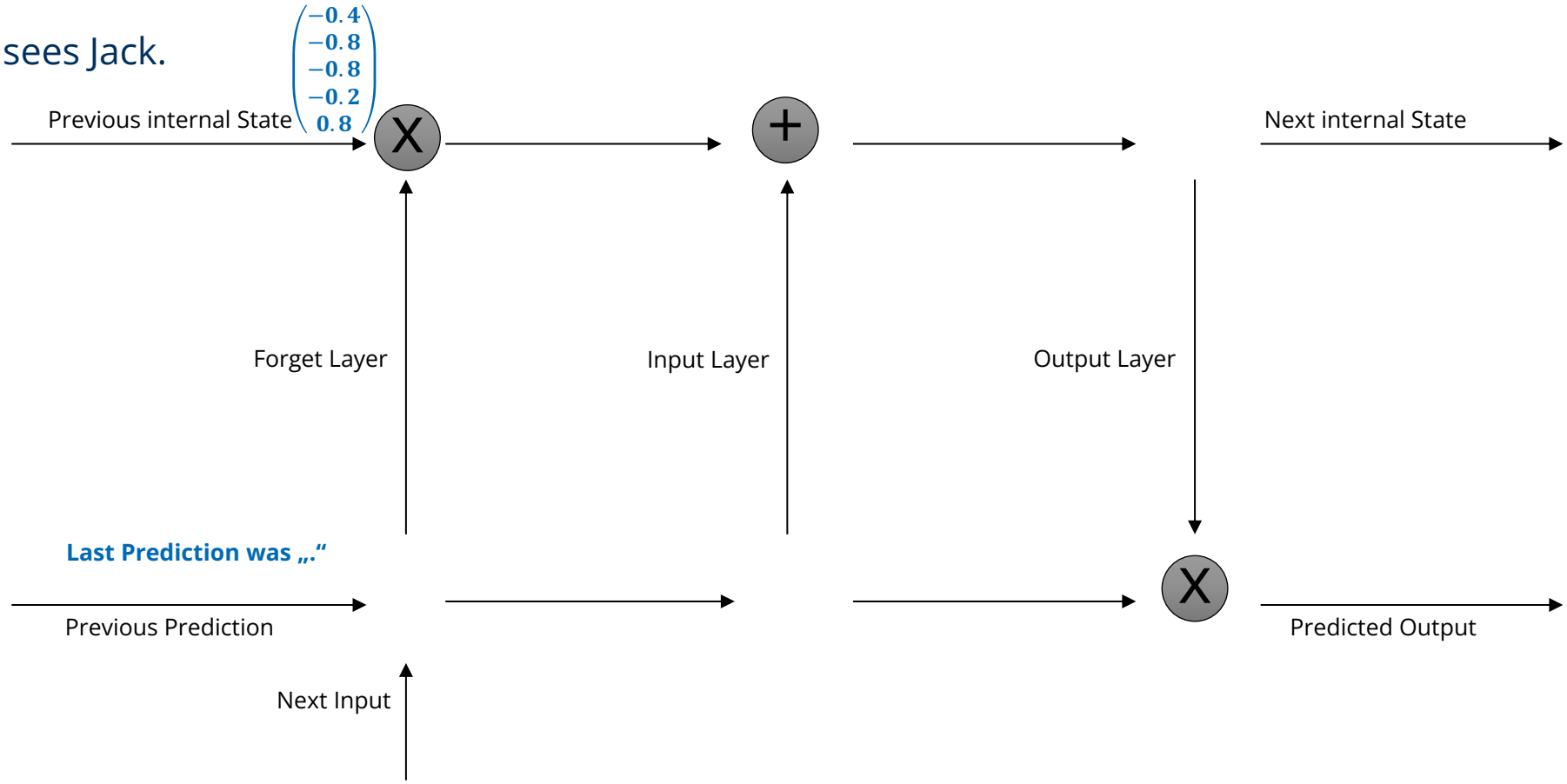
The current step in the sequence is this: Jack sees Jane. Bobby sees Jack

Example idea from : https://www.youtube.com/watch?v=WCUNPb-5EYI

Example: Children‘s book

Jack sees Jane.

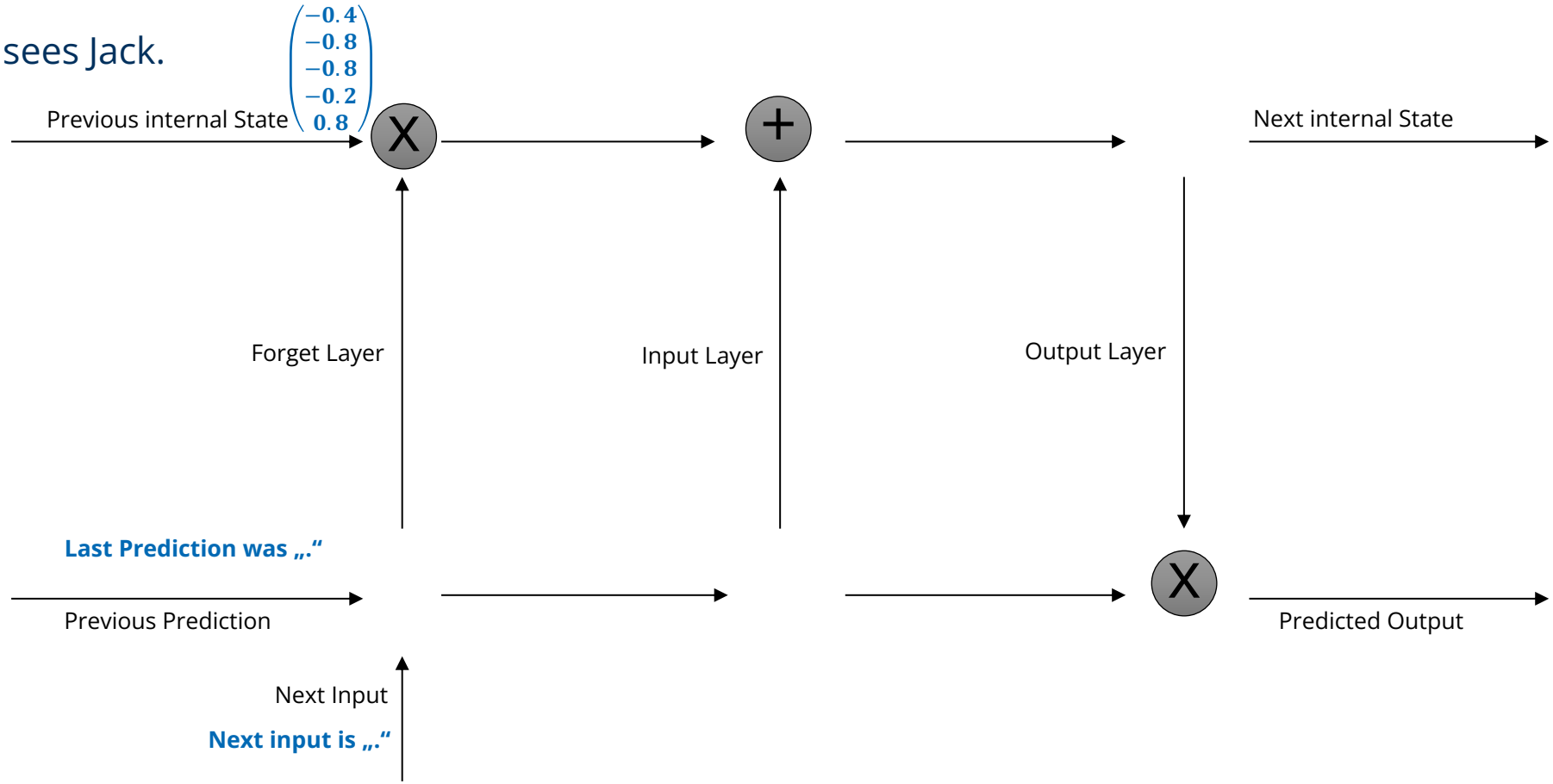
Bobby sees Jack.



Example: Children‘s book

Jack sees Jane.

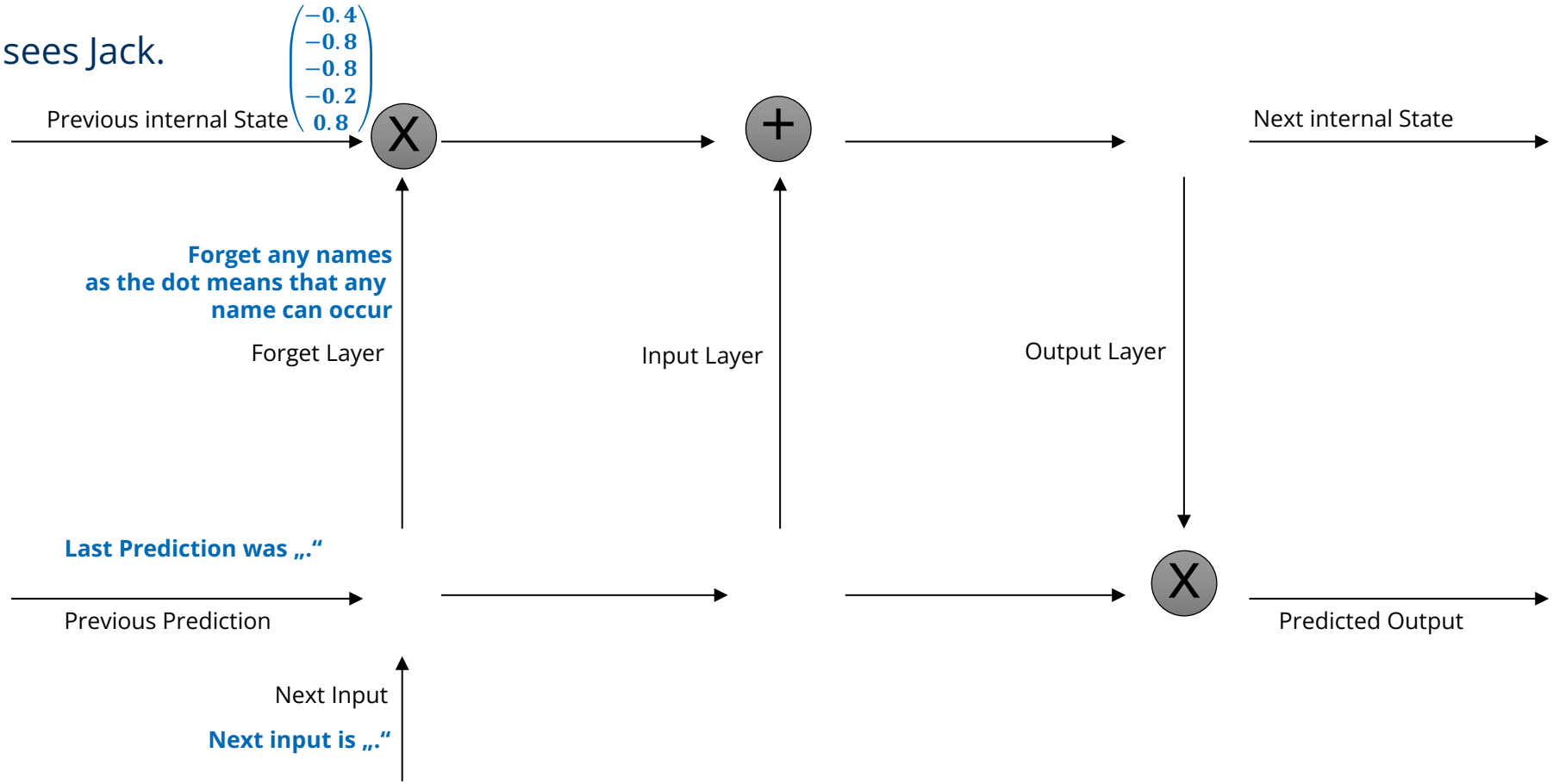
Bobby sees Jack.



Example: Children‘s book

Jack sees Jane.

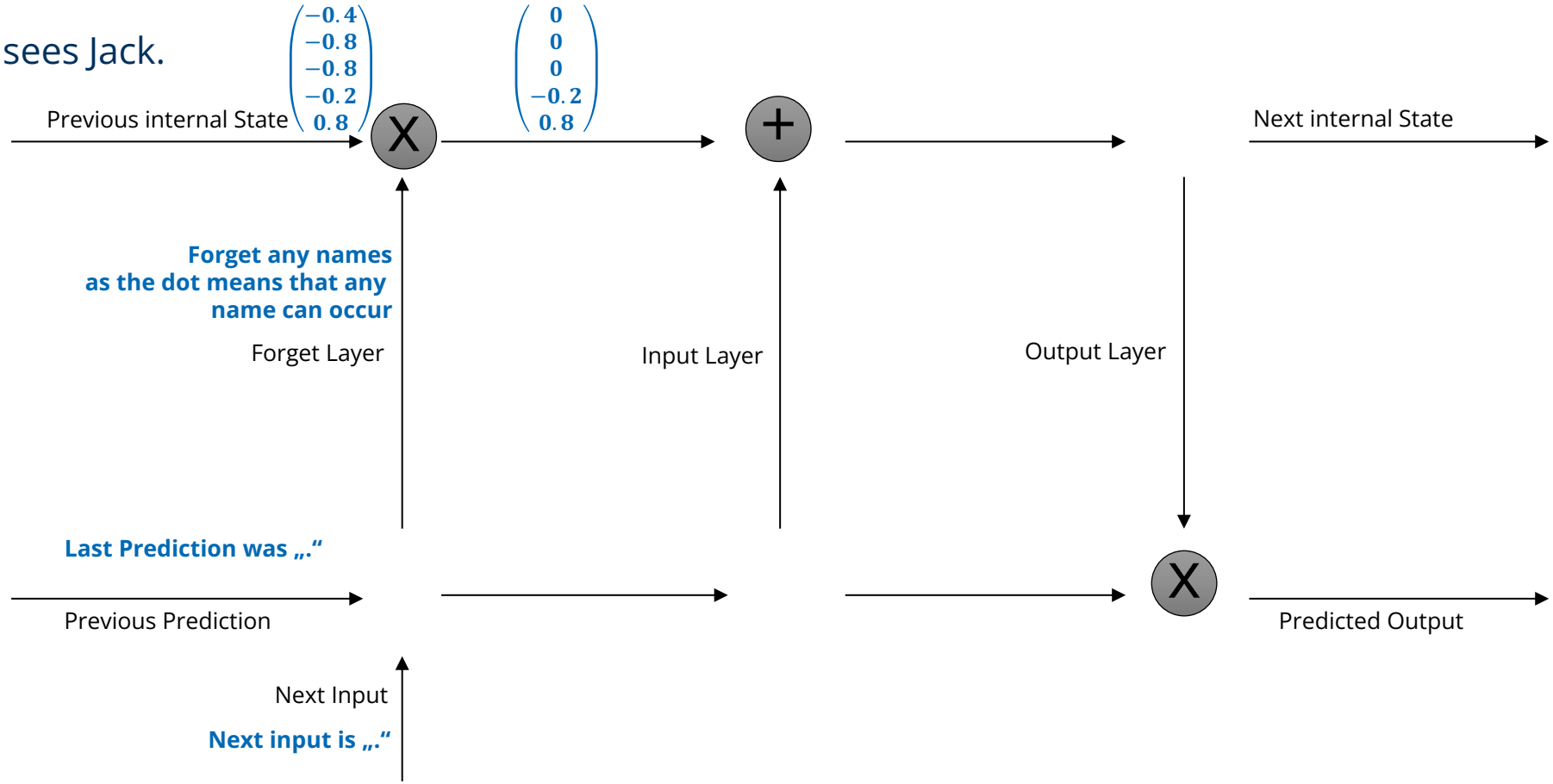
Bobby sees Jack.



Example: Children‘s book

Jack sees Jane.

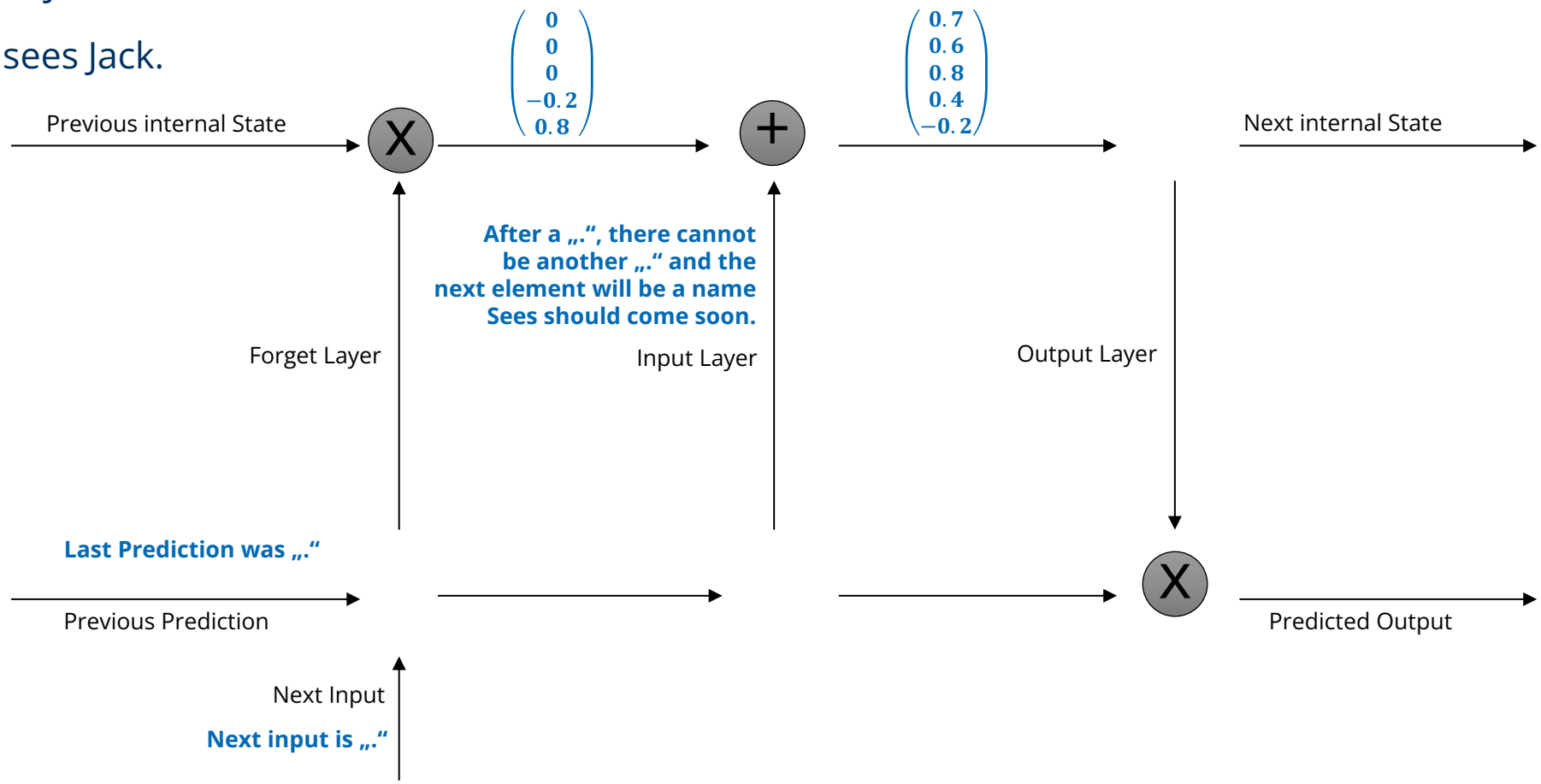
Bobby sees Jack.



Example: Children‘s book

Jack sees Jane.

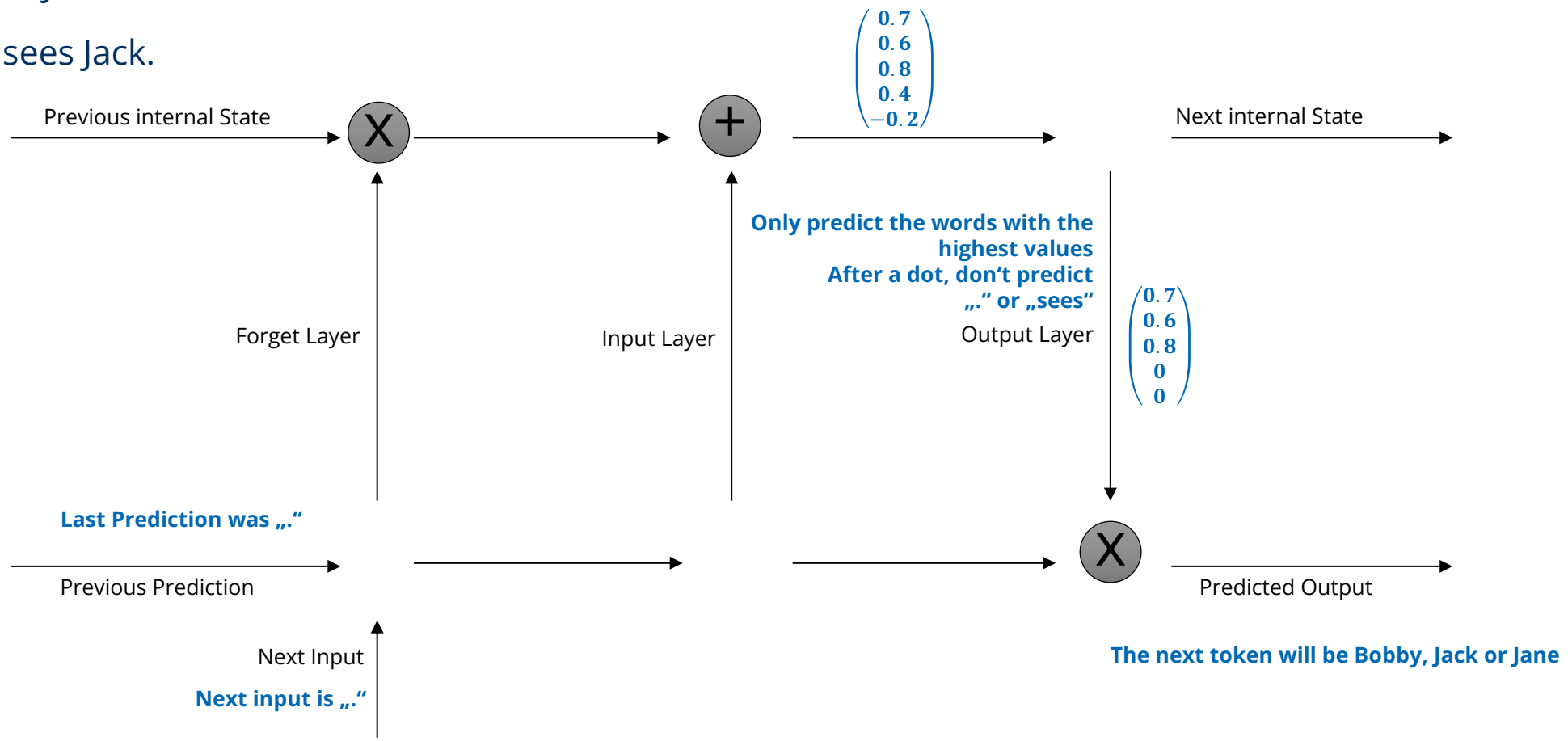
Bobby sees Jack.



Example: Children‘s book

Jack sees Jane.

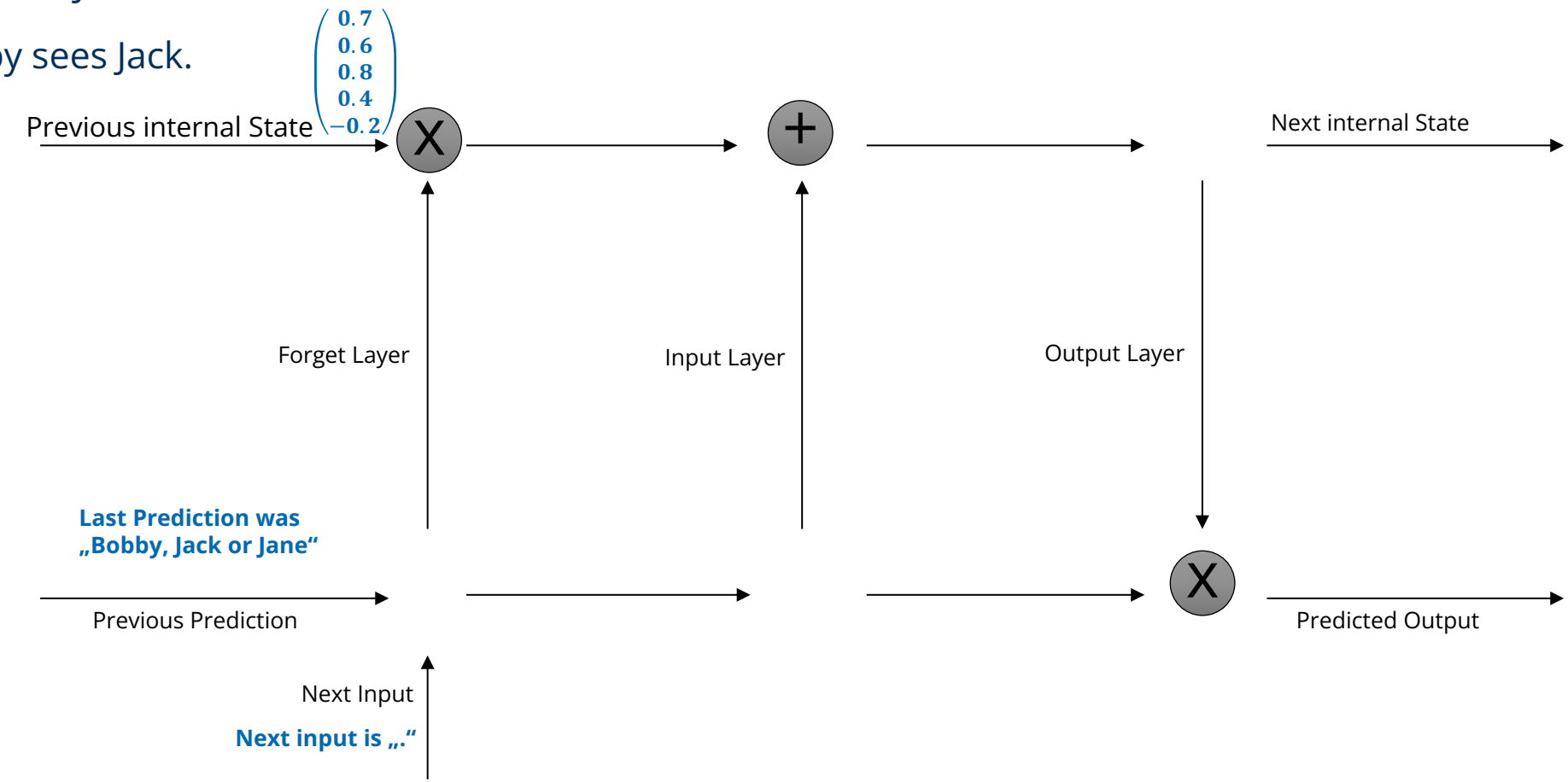
Bobby sees Jack.



Example: Children‘s book

Jack sees Jane.

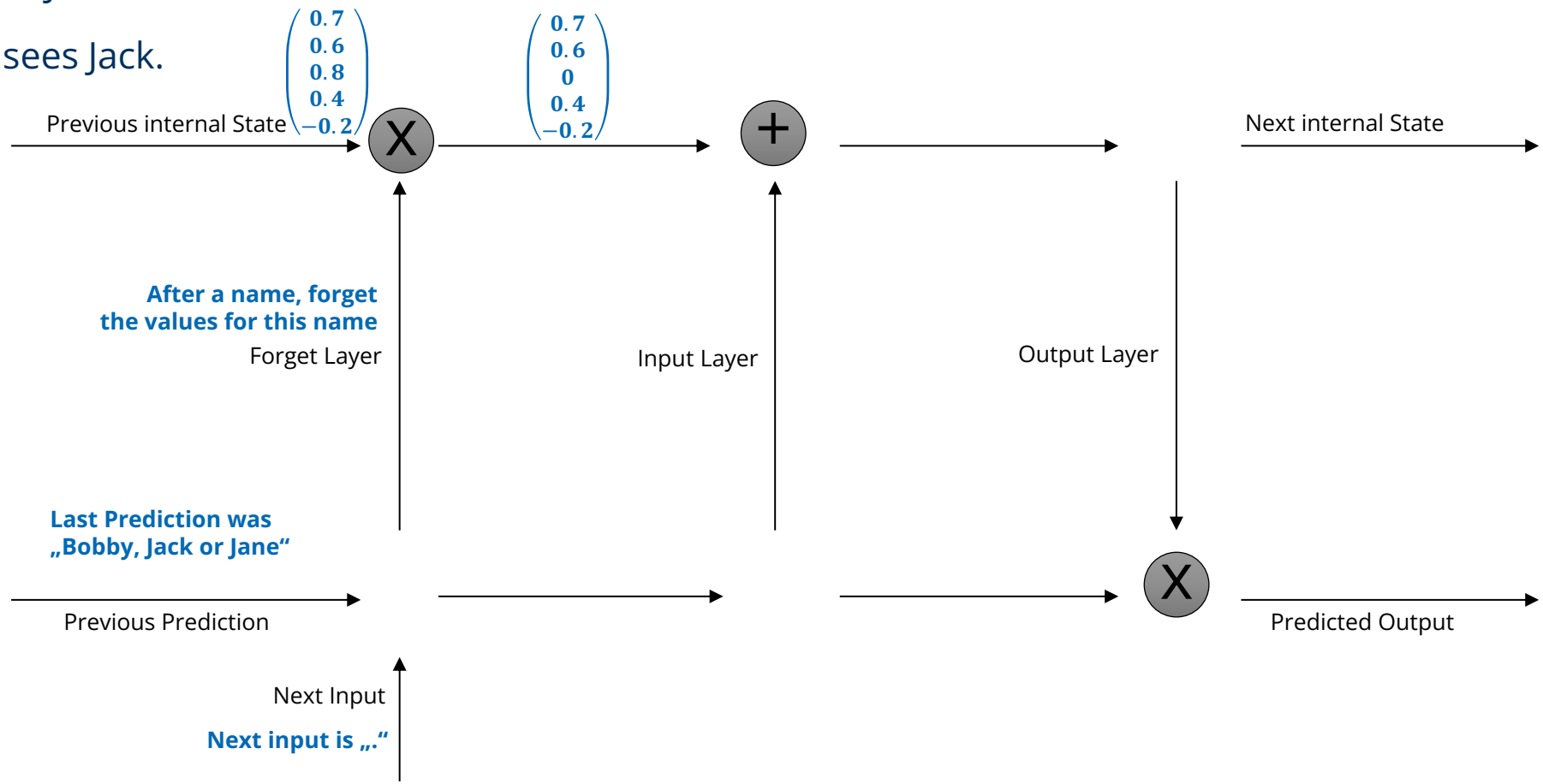
Bobby sees Jack.



Example: Children‘s book

Jack sees Jane.

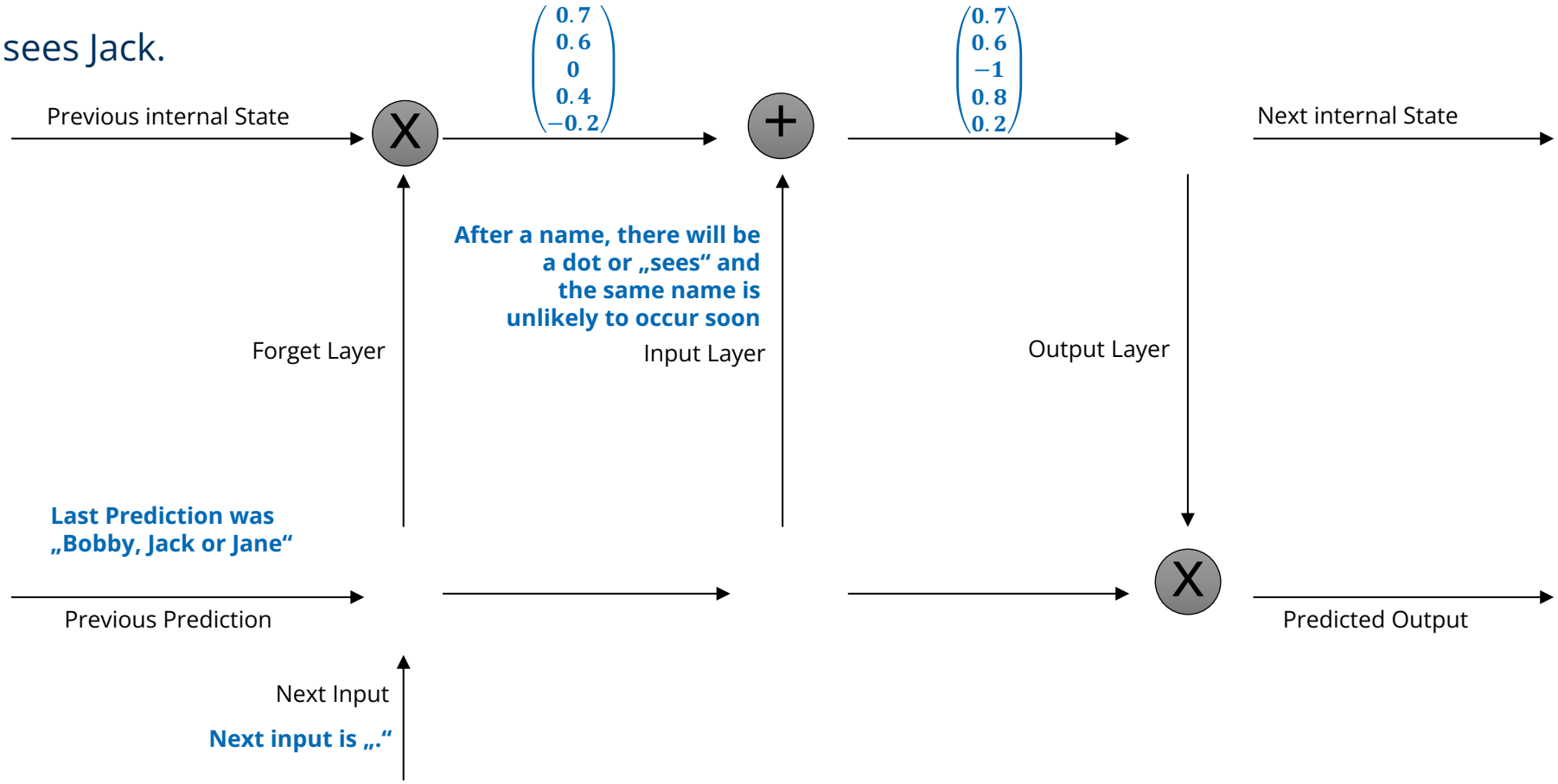
Bobby sees Jack.



Example: Children‘s book

Jack sees Jane.

Bobby sees Jack.

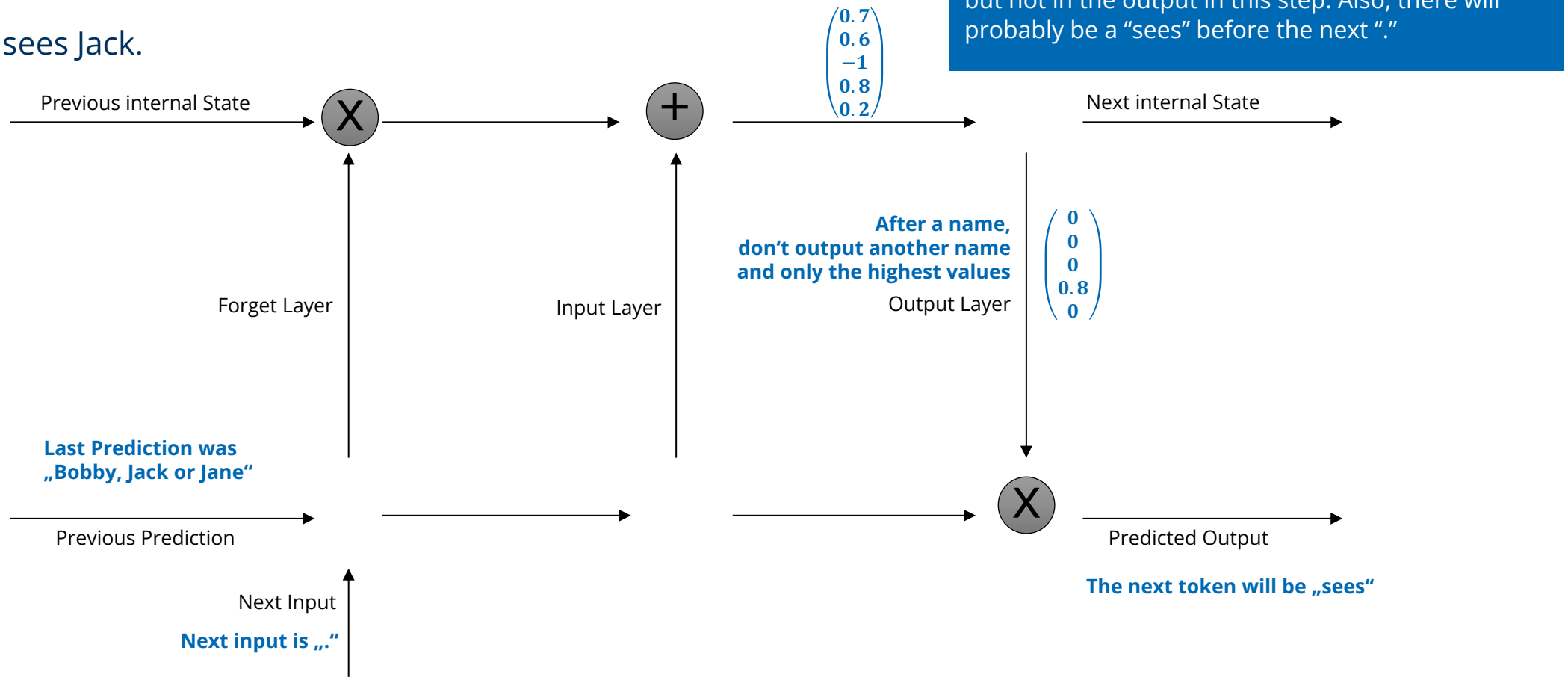


Example: Children‘s book

Jack sees Jane.

Bobby sees Jack.

Note: The cell now knows, that the next name will be Bobby or Jack, not Jane. This will be remembered but not in the output in this step. Also, there will probably be a “sees” before the next “.”



References & WebLinks

HOCHREITER, S.; SCHMIDHUBER, J.: Long Short-Term Memory. In: *Neural Computations* Vo.l 9 (1997), P. 1735-1780

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A.: Deep Learning (2016), P. 373ff.

FRANK, RAY J. ; DAVEY, NEIL ; HUNT, STEPHEN P.: Time series prediction and neural networks. In: *Journal of intelligent and robotic systems* Vol. 31 (2001), Nr. 1–3, P. 91–103

Understanding LSTMs: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

TECHNISCHE UNIVERSITÄT DRESDEN

CHAIR OF BUSINESS INFORMATION SYSTEMS
ESP. INTELLIGENT SYSTEMS AND SERVICES

DRESDEN concept

Prof. Dr. Alfred Benedikt Brendel and Dr. Kai Heinrich
Chair of Business Information Systems, esp. Intelligent Systems and Services

Deep Learning

Long Short-Term Memory (LSTM) - Example

Steps to make predictions on a dataset

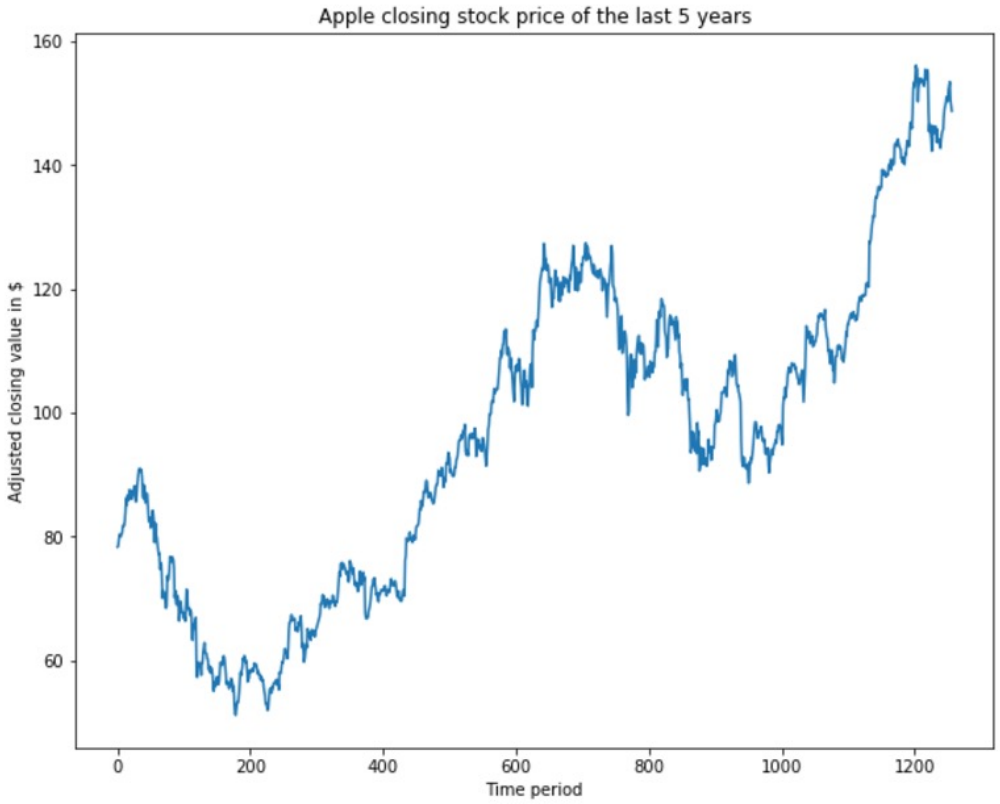
1. Load the data from a file (.txt, .csv, .xlsx)
2. Prepare the data for the keras model
3. Build the layers of the keras model
4. Train the model with a training subset of the data
5. Test the trained model with a testing subset of the data

Complete commented source code in the .ipynb file for this lesson

1. Load the data

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib

[...]
closing_values = np.loadtxt(
    "AAPL.csv",
    delimiter=",",
    skiprows=1,
    usecols=(5)).reshape(-1, 1)
[...]
plt.plot(closing_values)
[...]
```



2. Data preparation – normalization

LSTMs are sensitive to the scale of the data
→ Scale input data to values between 0 and 1

Remeber to reverse this normalization when you want to display predictions

```
from sklearn.preprocessing import MinMaxScaler


scaler = MinMaxScaler(feature_range=(0, 1))
normalized_closing_values = scaler.fit_transform(closing_values)

[...]

predictions = scaler.inverse_transform(normalized_predictions)
```

2. Data preparation – correct shape

The keras LSTM layer expects the input data to be in the correct shape:

3 Dimensions

–Number of inputs for which we want predictions (3)

–Number of timesteps in each input (5)

–Number of values in each timestep (1)

Inputs	Output
$x_t, x_{t+1}, x_{t+2}, x_{t+3}, x_{t+4}$	y_{t+5}
$x_{t+1}, x_{t+2}, x_{t+3}, x_{t+4}, x_{t+5}$	y_{t+6}
$x_{t+2}, x_{t+3}, x_{t+4}, x_{t+5}, x_{t+6}$	y_{t+7}

x and y are vectors of size 1

```
def prepare_sequence_data(values, window_size=2):
    X = []
    y = []

    for i in range(window_size, len(values)):
        X.append(values[i - window_size:i])

        y.append(values[i])


    X = np.asarray(X)
    X.shape = (len(X), window_size, 1)
    y = np.asarray(y)
    y.shape = (len(y),1)

    return X,y
```

3. Build layers

The keras LSTM layers work the same as the simple feed forward layers:

The code example adds 2 layers:

–An LSTM layer with 5 modules

–A feed forward layer with 1 output value: the prediction

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
import keras

model = Sequential()
model.add(LSTM(5, input_shape=(window_size, 1)))
model.add(Dense(1))

model.compile(loss='mean_squared_error', optimizer='adam')
```

4. Train the model

Generate training and test data first

Train the model on the training data

```
window_size = 5
test_count = 100
train_data = normalized_closing_values[:-test_count]
test_data = normalized_closing_values[-test_count:]
X_train, y_train = prepare_sequence_data(train_data, window_size)
X_test, y_test = prepare_sequence_data(test_data, window_size)

[...]

model.fit(X_train, y_train, epochs=100, batch_size=20, verbose=2)
```

5. Test the model

First, make predicted outputs for the test data and the training data

Denormalize this data

Test it by:

–Plotting it

–Calculating the loss

```
import math
from sklearn.metrics import mean_squared_error

train_predictions = scaler.inverse_transform(normalized_train_predictions)
test_predictions = scaler.inverse_transform(normalized_test_predictions)

plt.plot(closing_values)
plt.plot(train_predictions)
plt.plot(range(len(X_train), len(X_train) + len(X_test)), test_predictions)

test_score = math.sqrt(
    mean_squared_error(y_test[:,0], test_predictions[:,0]))
```

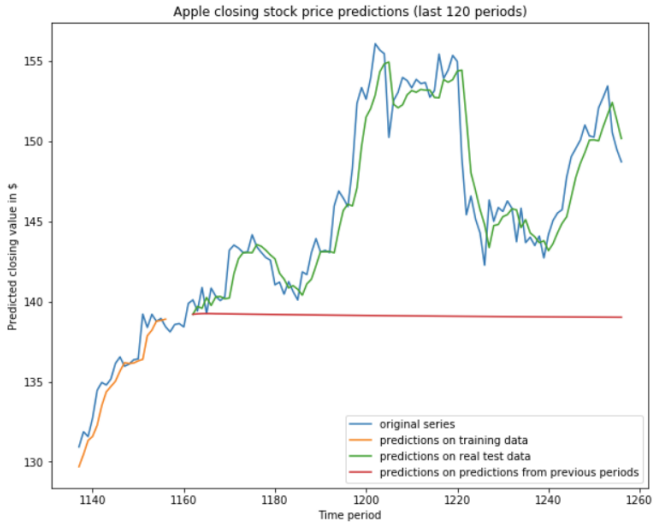
Result

Predictions on the test data are not too far from the original data.

Network does not just output a factor of the last seen value, because the prediction graph is not in the same shape as the original data.

The red line shows, that the network cannot learn all the patterns of the stock price.
→ When feeding its own predictions from previous periods, the line stays more or less the same

- Stock prices in general might not have enough recurring patterns
- Small data sets
- Short training time



Prof. Dr. Alfred Benedikt Brendel and Dr. Kai Heinrich
Chair of Business Information Systems, esp. Intelligent Systems and Services

Thank you for your attention