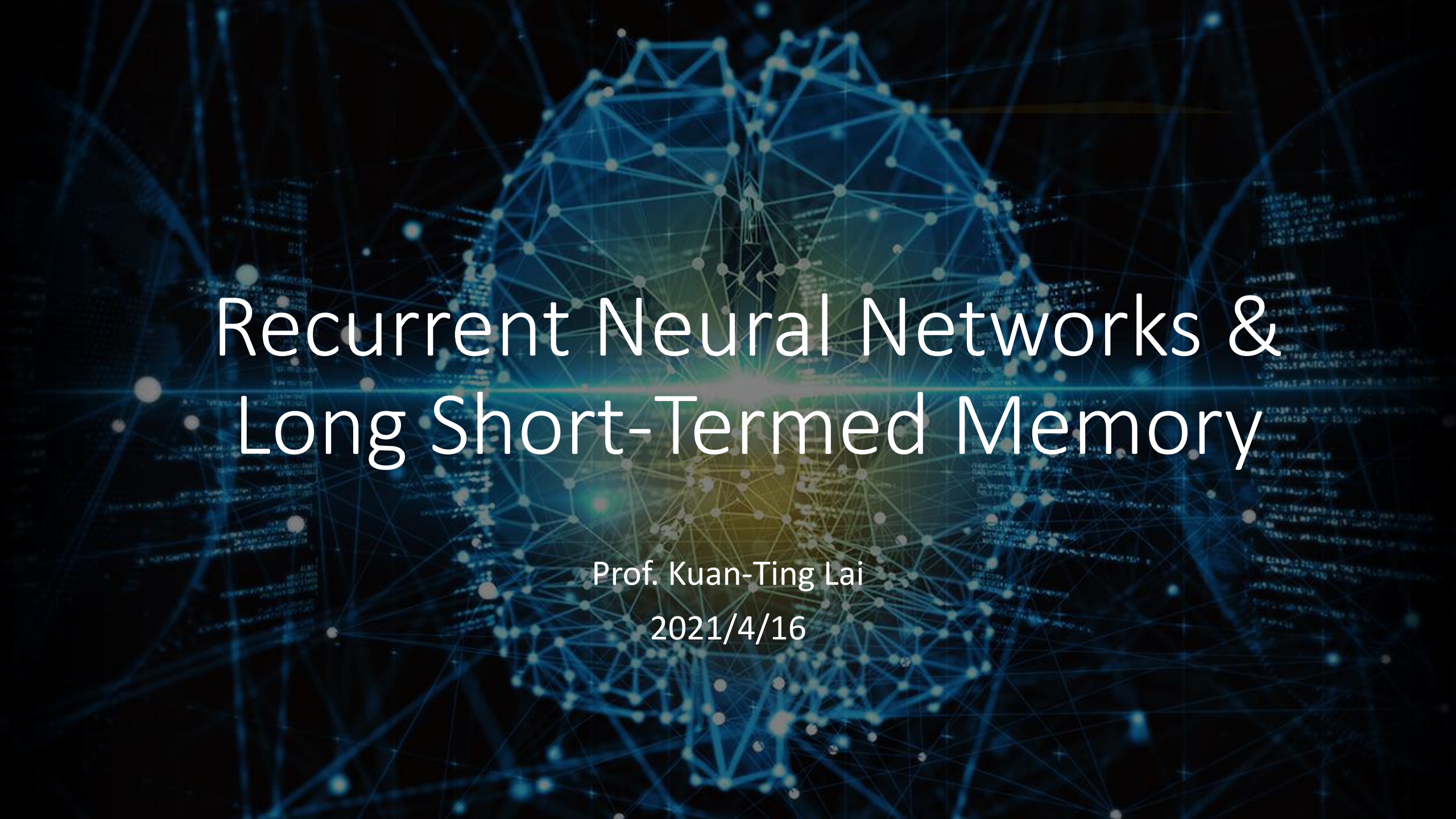


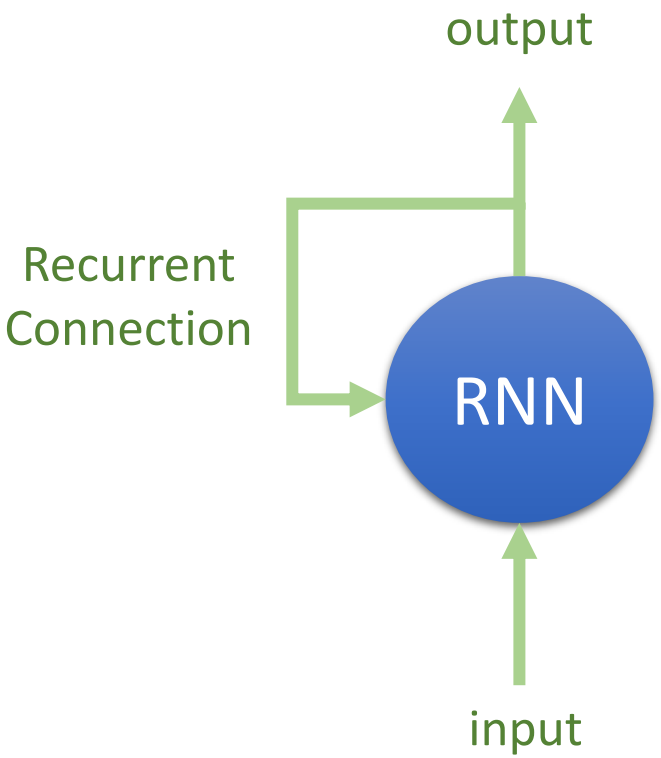
Recurrent Neural Networks & Long Short-Termed Memory

Prof. Kuan-Ting Lai

2021/4/16

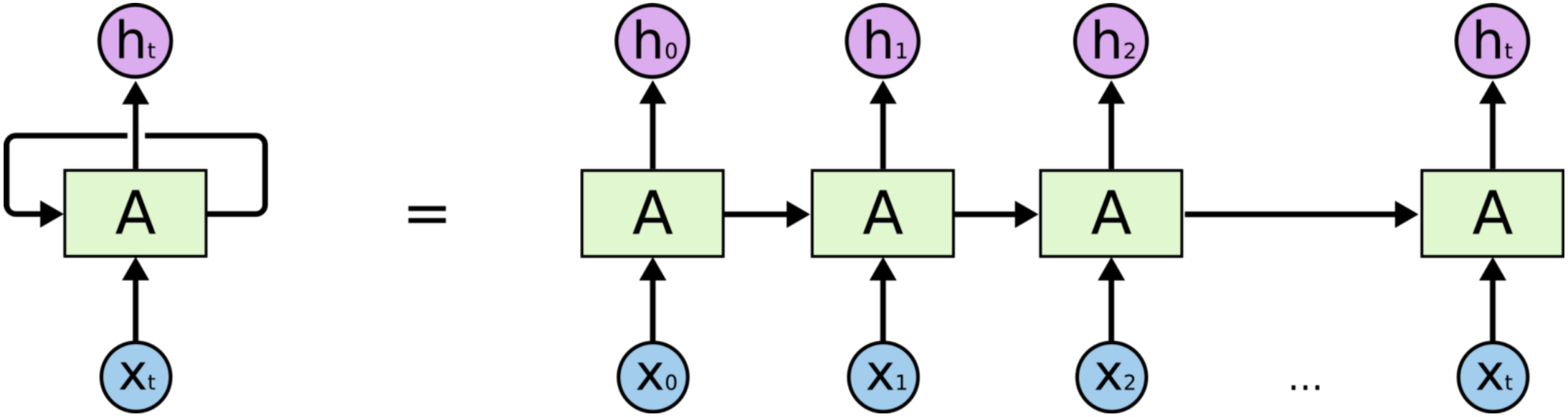
Recurrent Neural Network (RNN)

- Feedforward networks don’t consider temporal states
- RNN has a loop to “memorize” information



Unroll the RNN Loop

- Effective for speech recognition, language modeling, translation



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

Simple Recurrent Networks

- Elman network

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + by)$$

- Jordan network

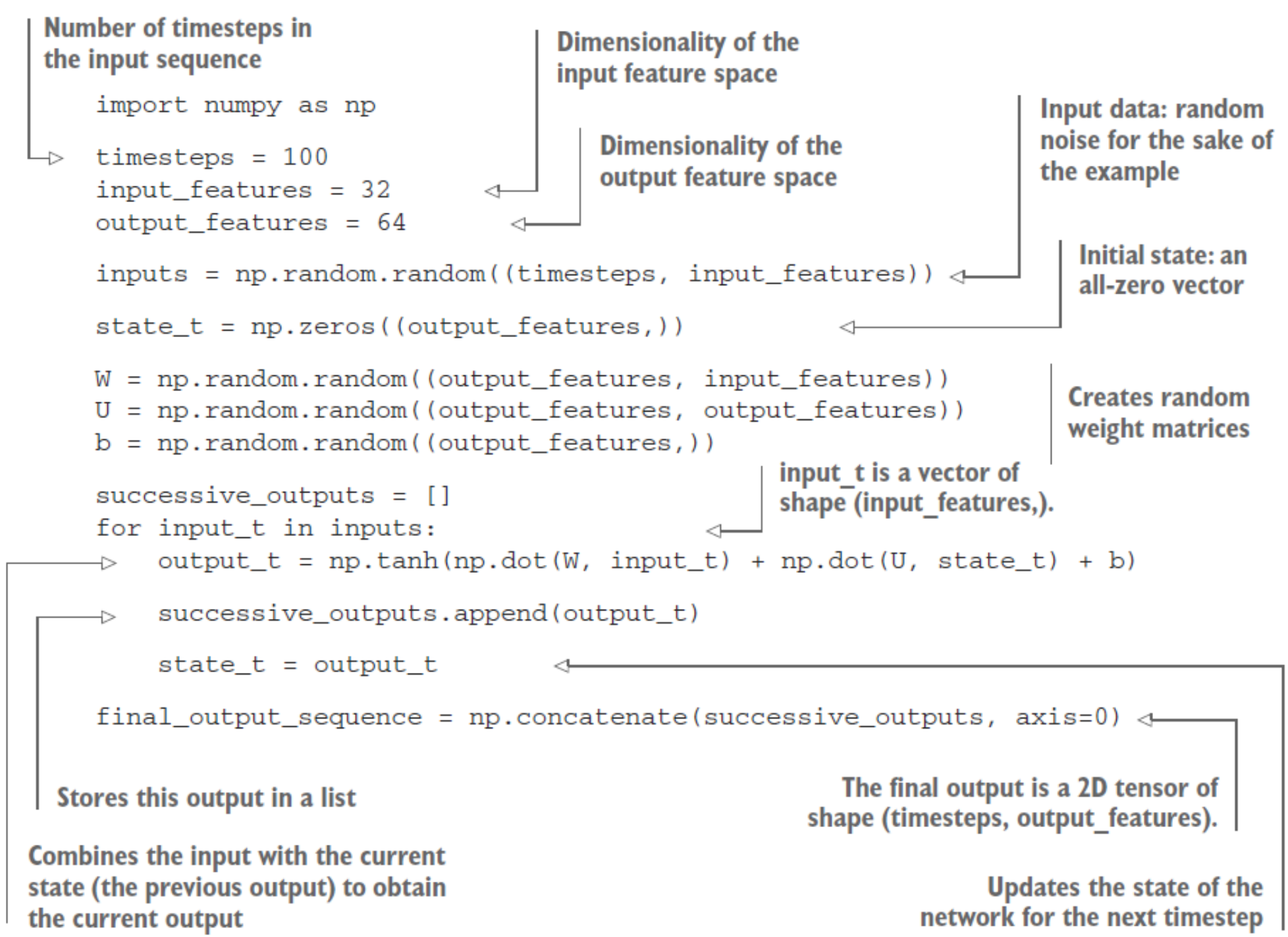
$$h_t = \sigma_h(W_h x_t + U_h y_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + by)$$

$\boldsymbol{x_t}$: input vector
$\boldsymbol{y_t}$: output vector
$\boldsymbol{h_t}$: hidden layer vector
$\boldsymbol{W}$, $\boldsymbol{U}$, $\boldsymbol{b}$: weights and bias
$\boldsymbol{\sigma_h}$, $\boldsymbol{\sigma_y}$: activation functions

Pseudo RNN

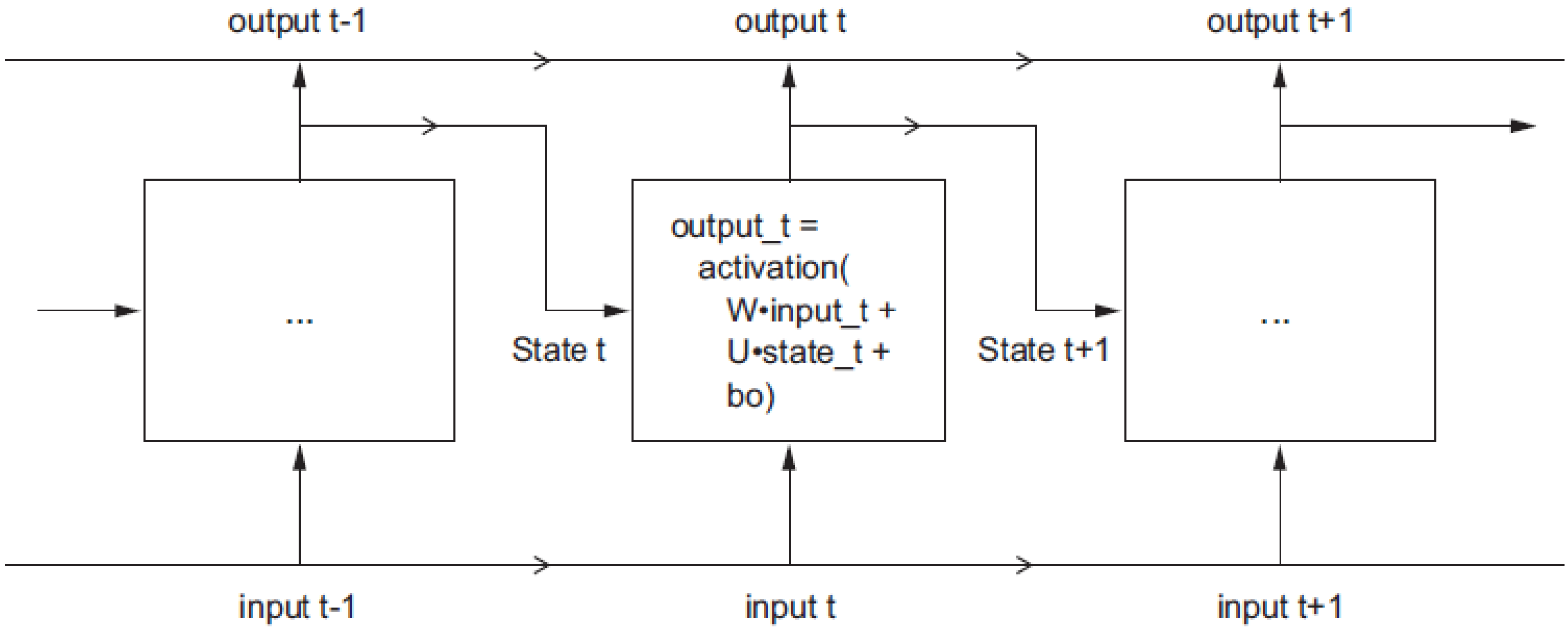
```
# Pseudo RNN
state_t = 0
for input_t in input_sequence:
    output_t = f(input_t, state_t)
    state_t = output_t


# Pseudo RMN with activation function
# y_t = W*x_t + U*S_t + b
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

RNN using NumPy




```
import numpy as np

timesteps = 100
input_features = 32
output_features = 64

inputs = np.random.random((timesteps, input_features))

state_t = np.zeros((output_features,))

W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))

successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)

    successive_outputs.append(output_t)

    state_t = output_t

final_output_sequence = np.concatenate(successive_outputs, axis=0)
```

Unroll RNN


output t-1
output t
output t+1
...
State t
output_t =
activation(
W•input_t +
U•state_t +
bo)
State t+1
...
input t-1
input t
input t+1

Recurrent Layer in Keras

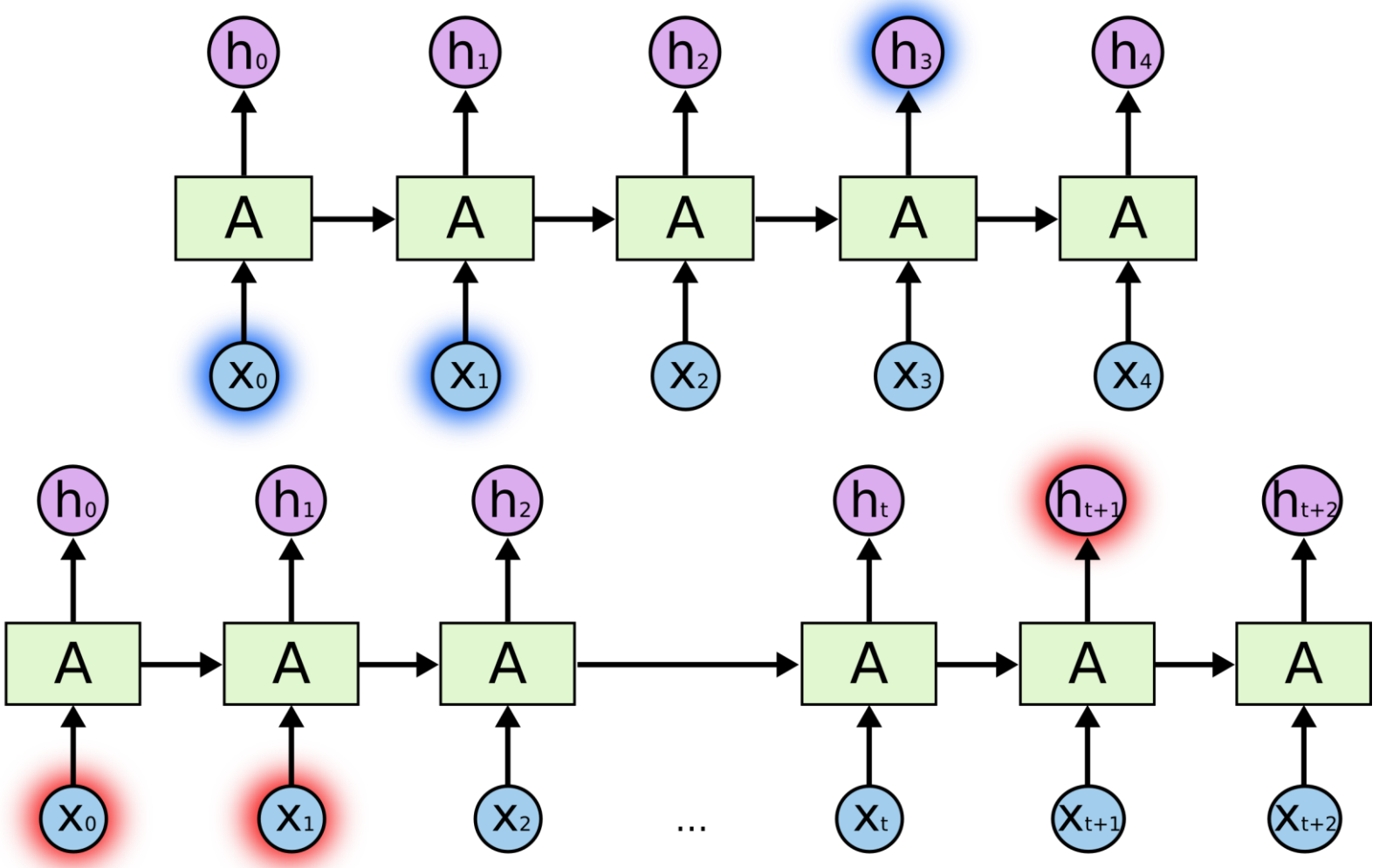
- Simple RNN

```
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN

model = Sequential()
model.add(Embedding(10000, 32))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32, return_sequences=True))
model.add(SimpleRNN(32))
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_24 (Embedding)     (None, None, 32)          320000
_________________________________________________________________
simplernn_12 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_13 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_14 (SimpleRNN)     (None, None, 32)          2080
_________________________________________________________________
simplernn_15 (SimpleRNN)     (None, 32)                2080
=================================================================
Total params: 328,320
Trainable params: 328,320
Non-trainable params: 0
```

Vanishing Gradient Problem

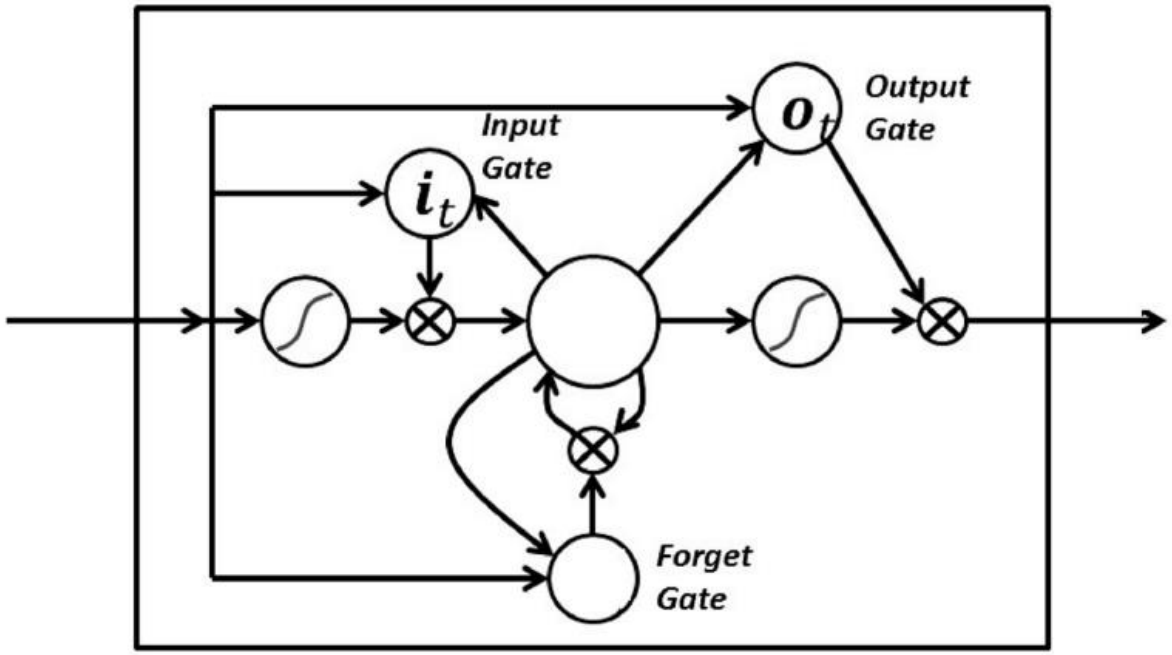
- Hochreiter (1991) [German] and Bengio, et al. (1994)



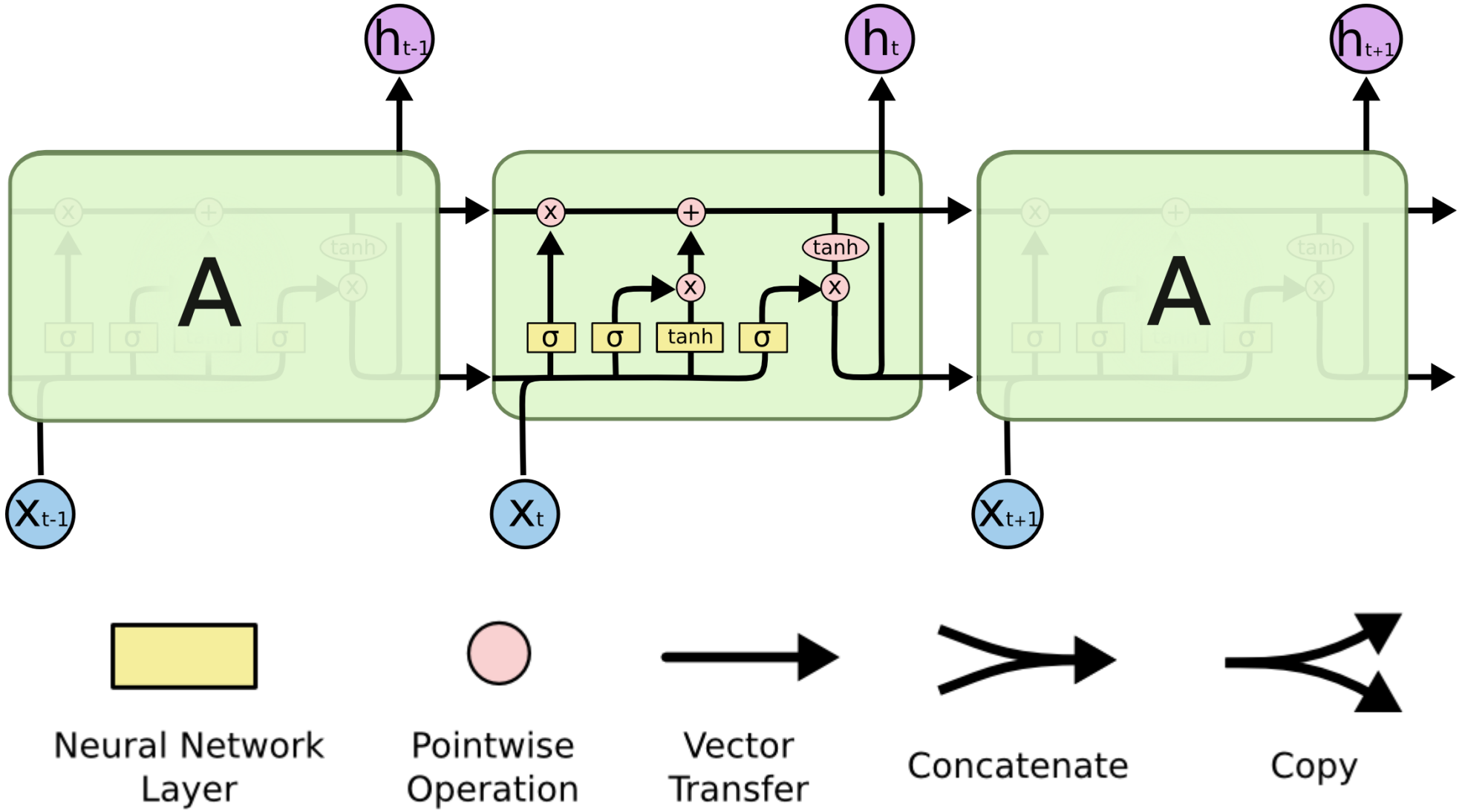
Long Short-Term Memory (LSTM)

- **Input gate**: control when to let new input in
- **Forget gate**: delete the trivial information
- **Output gate**: let the info impact the output at the current time step

Hochreiter & Schmidhuber (1997)

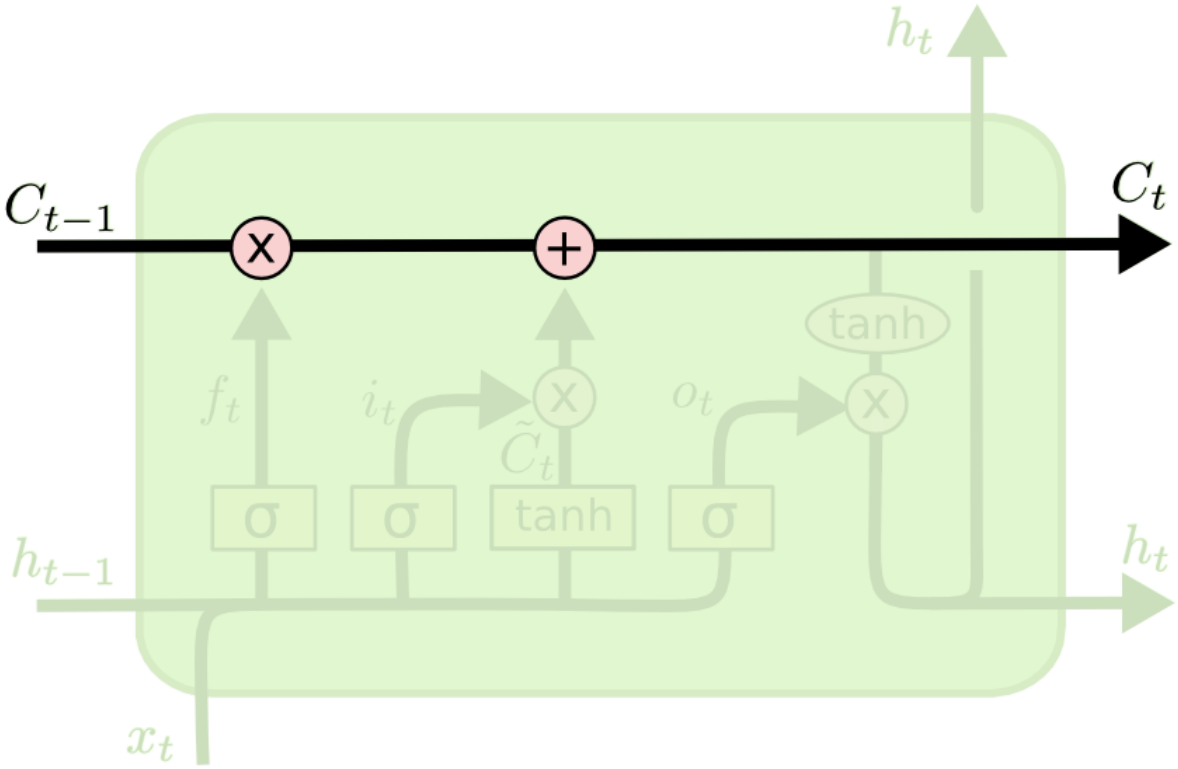


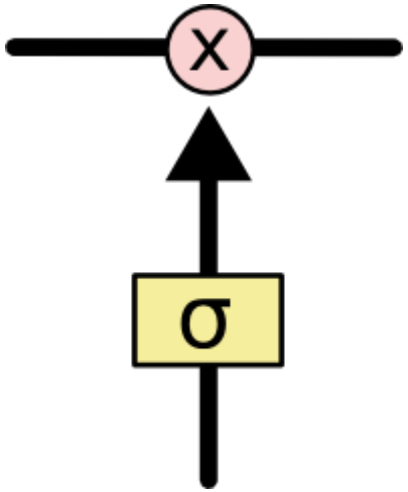
Long Short-Term Memory (LSTM)



Core Idea of LSTM

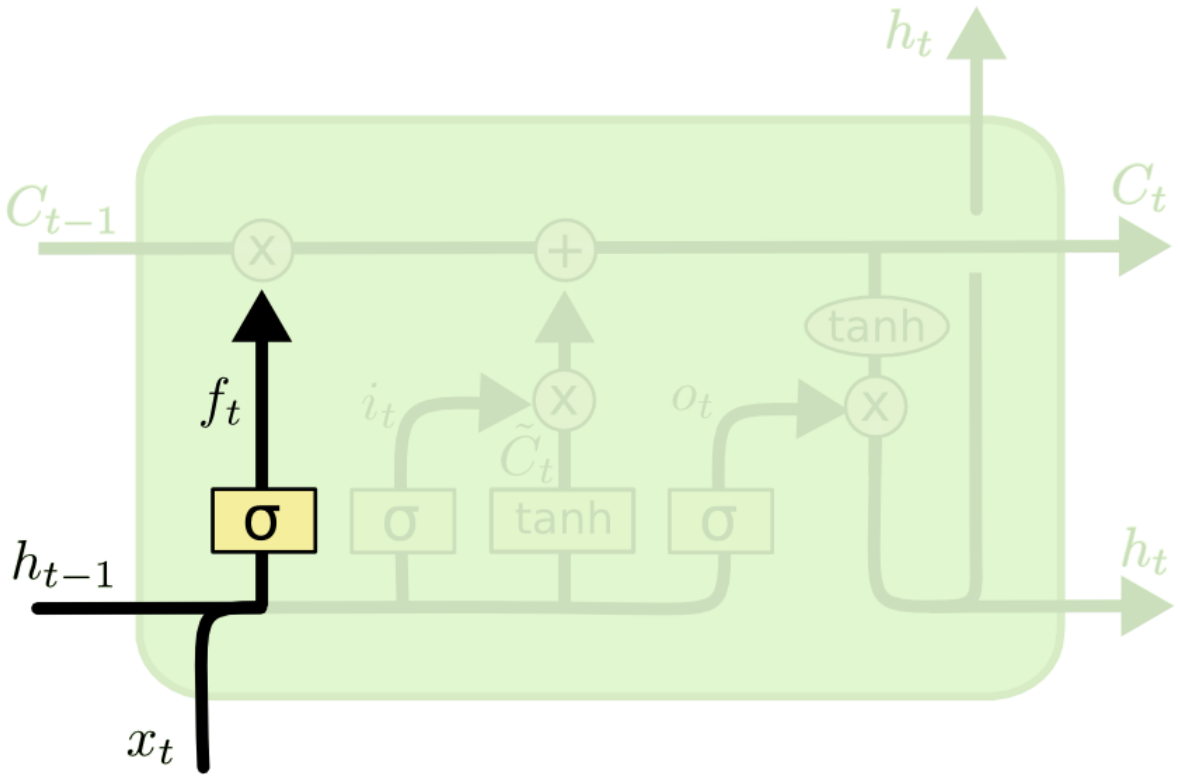
- Cell State C_t : allow information flow unchanged





LSTM Step-by-Step (4-1)

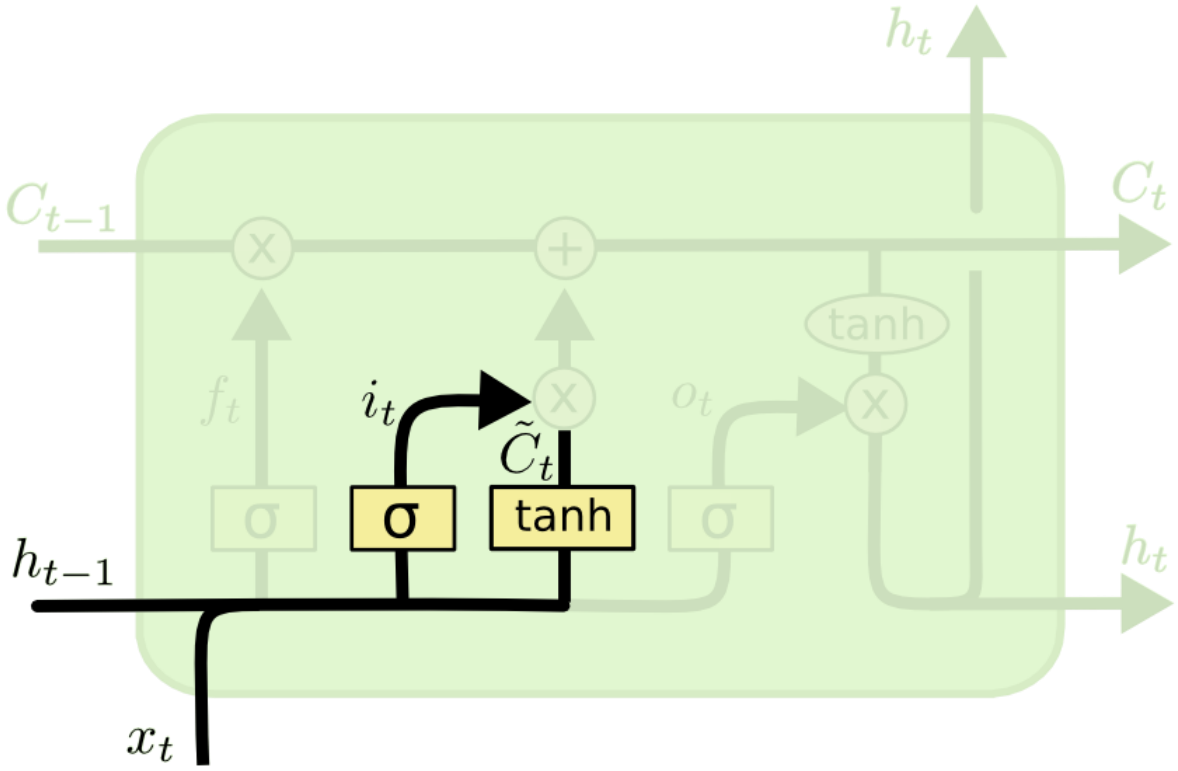
- Decide if to throw away old cell state information C_{t-1}



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

LSTM Step-by-Step (4-2)

- Decide what information to be stored in current cell state C_t

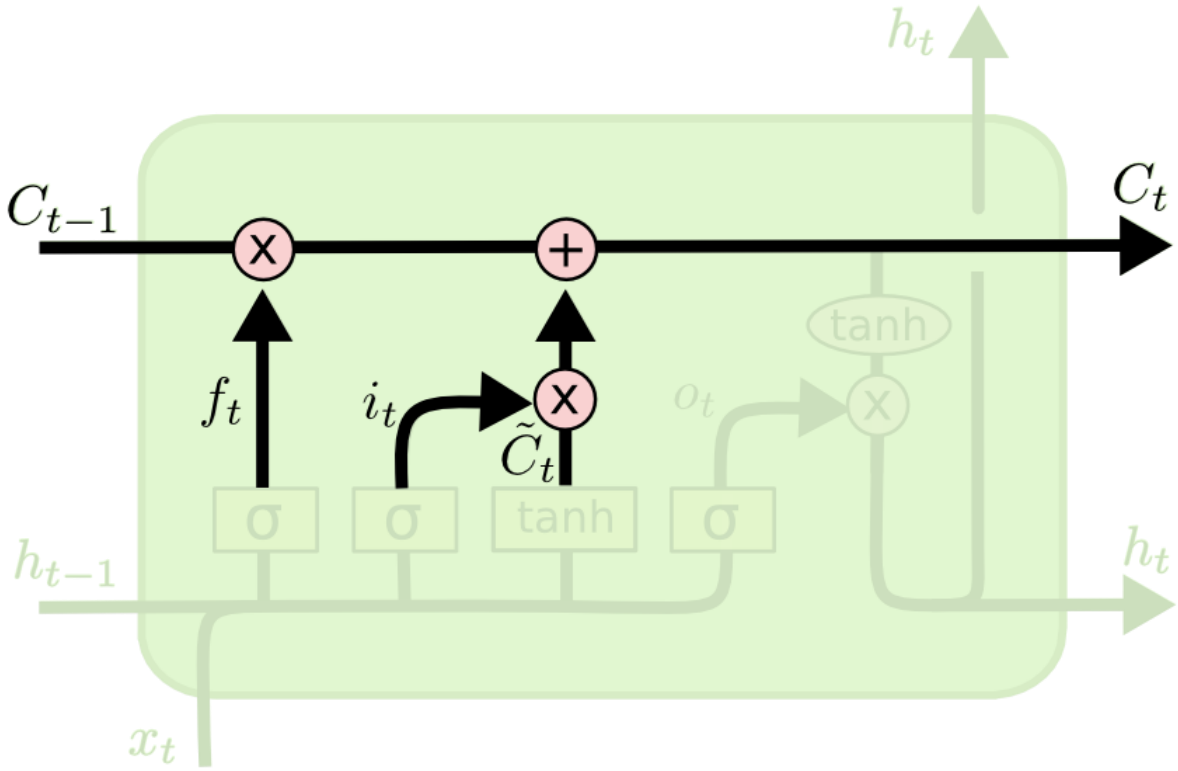


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

LSTM Step-by-Step (4-3)

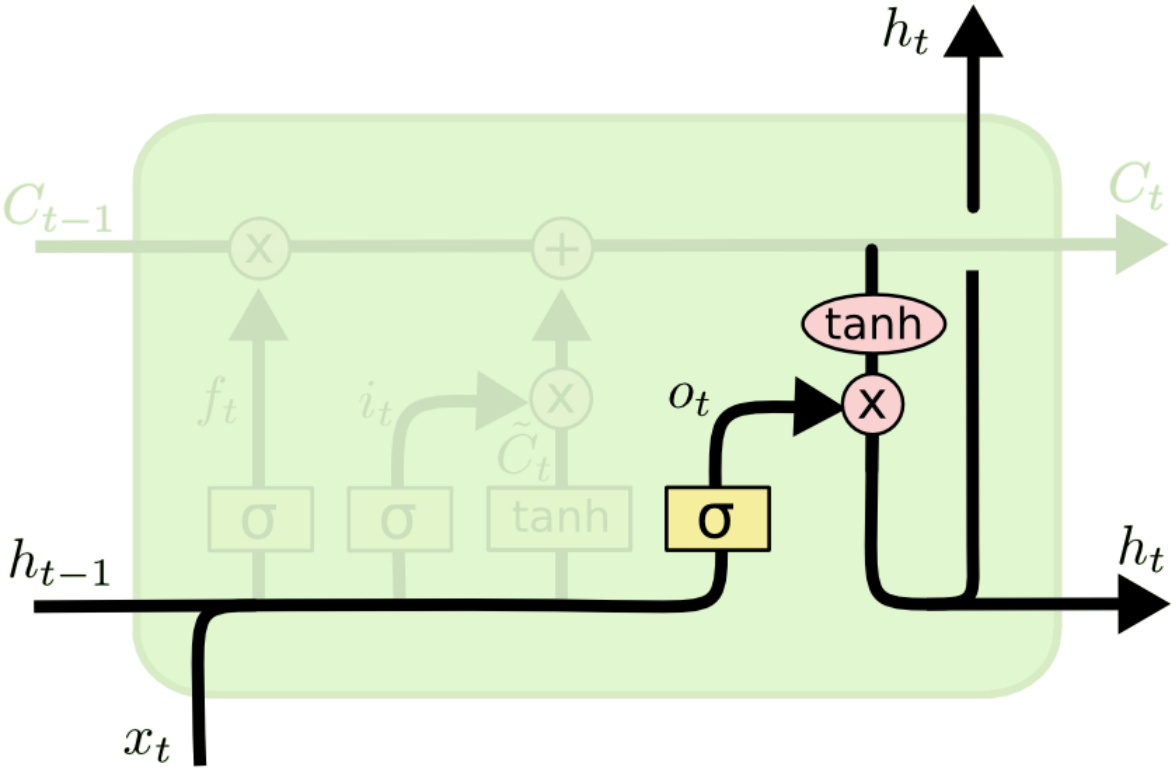
- Update old cell state C_{t-1} into current C_t



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

LSTM Step-by-Step (4-4)

- Decide what to output

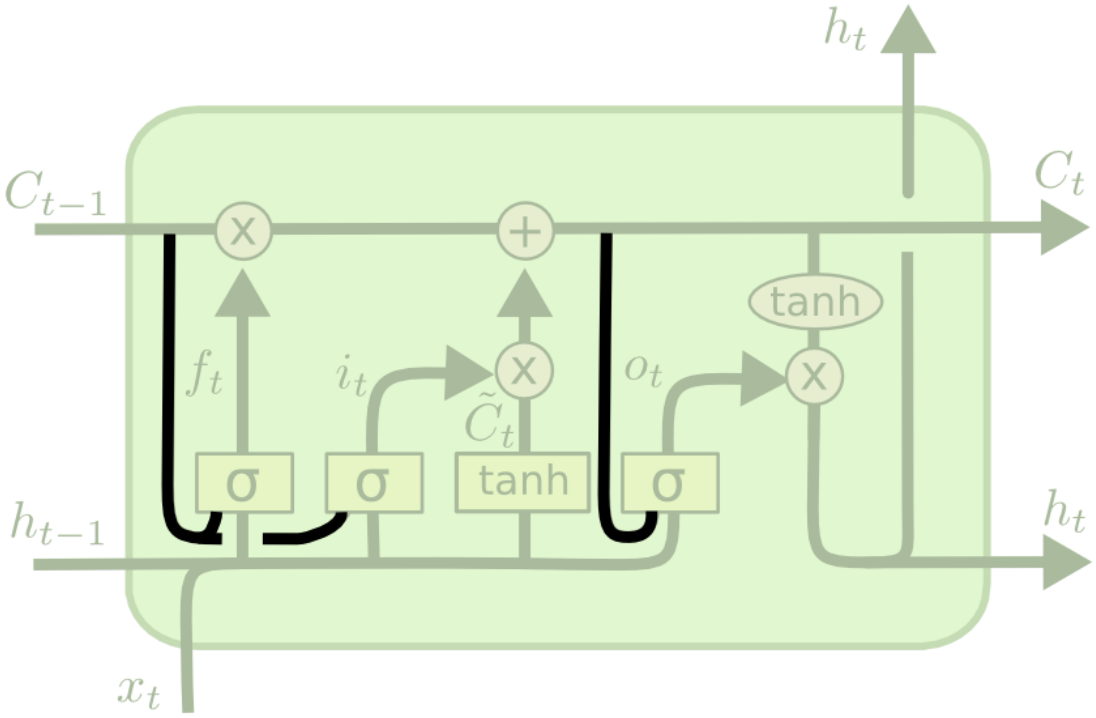


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Variants of LSTM

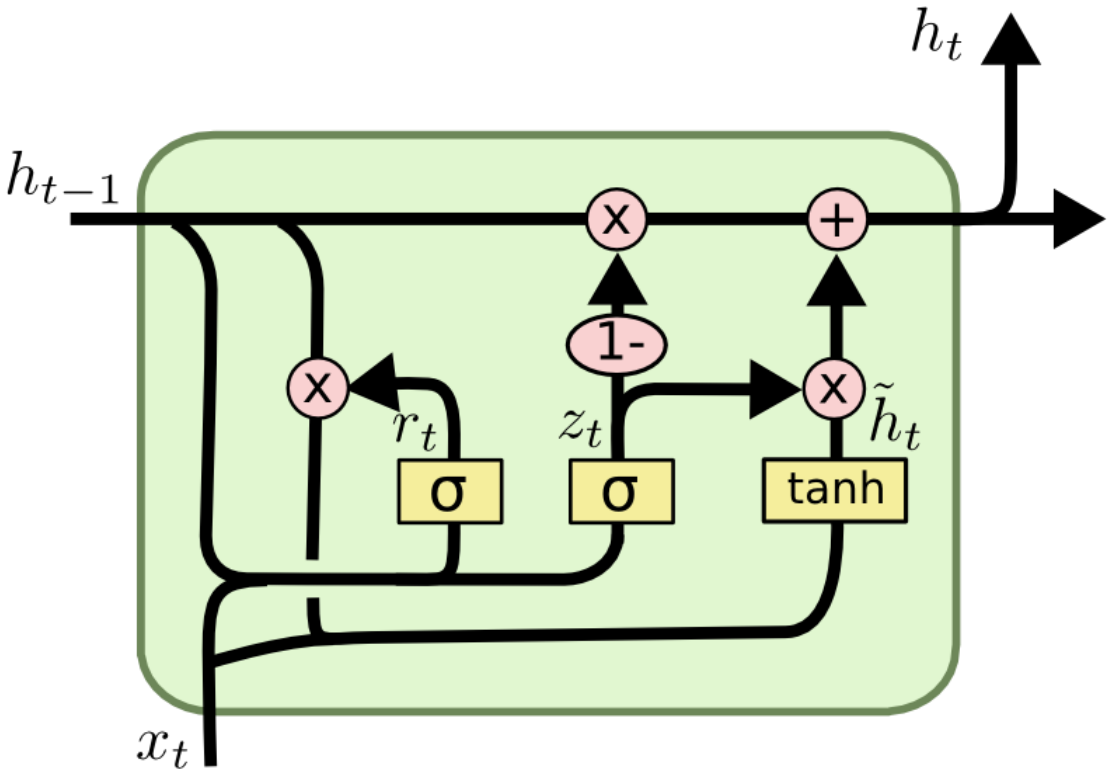
- Gers & Schmidhuber (2000)



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

Gated Recurrent Unit (GRU)

- Cho, et al. (2014)
- Combine the forget and input gates into a single “update gate.”



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Using LSTM in Keras

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_words, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
              epochs=10,
              batch_size=128,
              validation_split=0.2)
```

Advanced Use of RNN

- *Recurrent dropout*
  - Use dropout to fight overfitting in recurrent layers
- *Stacking recurrent layers*
  - This increases the representational power of the network (at the cost of higher computational loads)
- *Bidirectional recurrent layers*
  - These present the same information to a recurrent network in different ways, increasing accuracy and mitigating forgetting issues

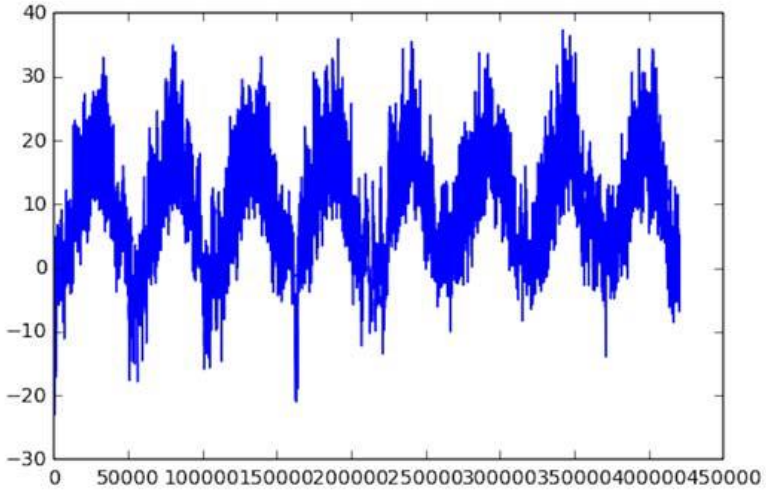
Temperature-forecasting Problem

- Measure 14 features every 10 minutes from 2009 – 2016 in Jena, Germany

["Date Time",
"p (mbar)",
"T (degC)",
"Tpot (K)",
"Tdew (degC)",
"rh (%)",
"VPmax (mbar)",
"VPact (mbar)",
"VPdef (mbar)",
"sh (g/kg)",
"H2OC (mmol/mol)",
"rho (g/m**3)",
"wv (m/s)",
"max. wv (m/s)",
"wd (deg)"]

°C

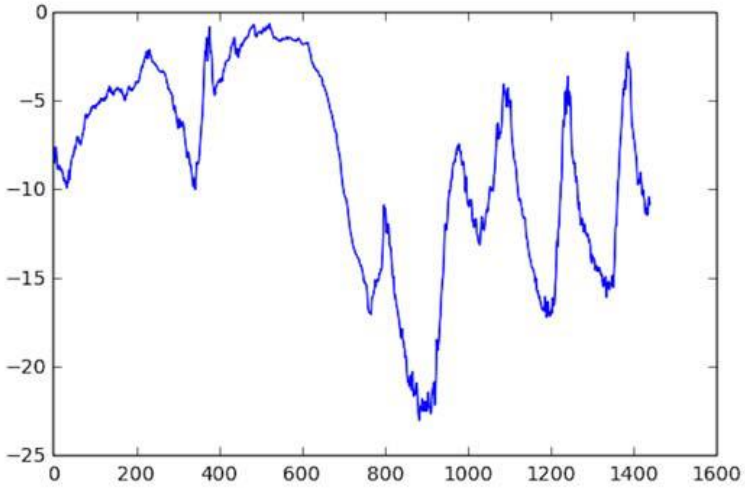
All Time
(2009 – 2016)



10 mins

°C

First 10 days



10 mins

Download Jena Weather Dataset

- AWS
  - wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip

Normalize the Data

- Remember to normalize your data!

```
mean = float_data[:200000].mean(axis=0)
float_data -= mean
std = float_data[:200000].std(axis=0)
float_data /= std
```

Learning Parameters

- lookback = 720
  - Observations will go back 5 days.
- steps = 6
  - Observations will be sampled at one data point per hour.
- delay = 144
  - Targets will be 24 hours in the future.

Design a Data Generator

- data— The normalized data
- lookback—How many timesteps back the input data should go.
- delay—How many timesteps in the future the target should be.
- min_index and max_index—Indices in the data array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another for testing.
- shuffle—Whether to shuffle the samples or draw them in chronological order.
- batch_size—The number of samples per batch.
- step—The period, in timesteps, at which you sample data. You'll set it to 6 in order to draw one data point every hour.

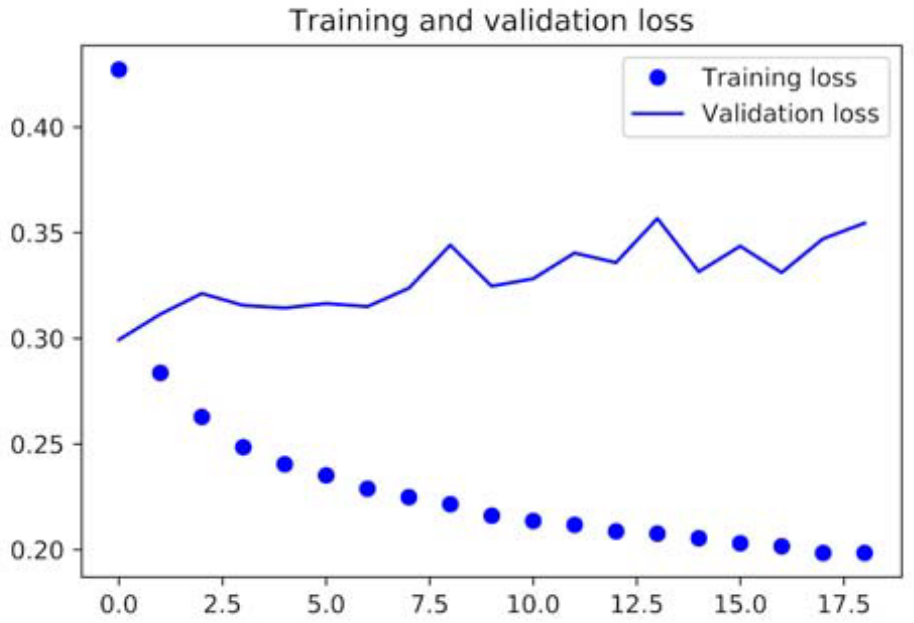
Timeseries Data Generator

```
def generator(data, lookback, delay, min_index, max_index, shuffle=False,
        batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)
        samples = np.zeros((len(rows), lookback // step, data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
            indices = range(rows[j] - lookback, rows[j], step)
            samples[j] = data[indices]
            targets[j] = data[rows[j] + delay][1]
        yield samples, targets
```

Create Baselines

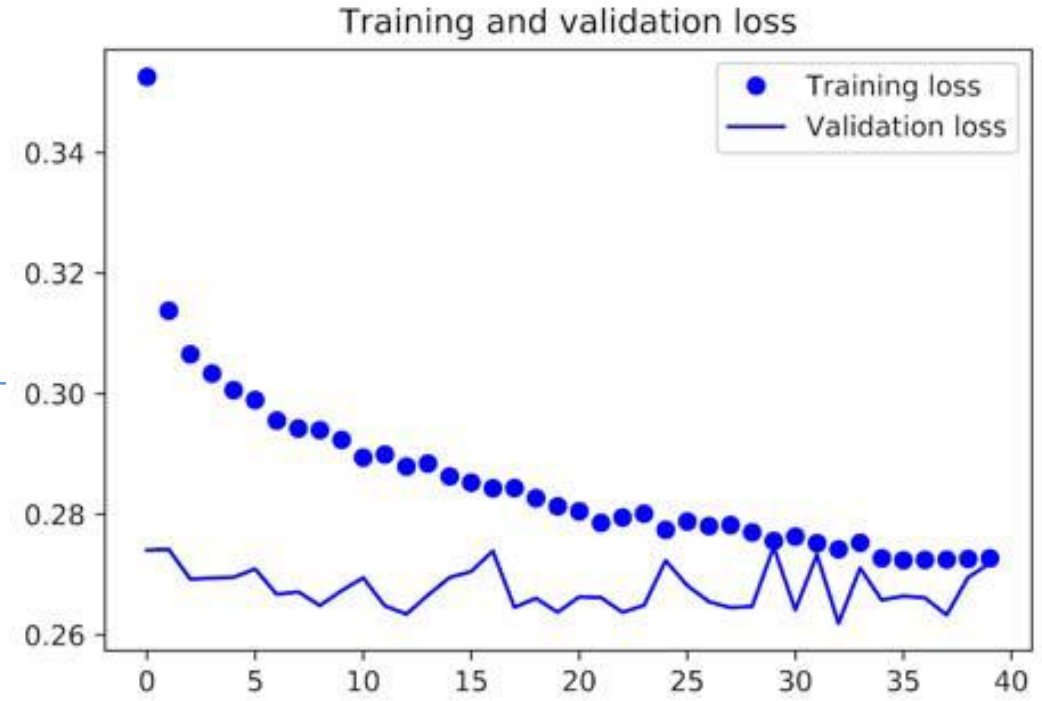
1. Common sense - Simply use last temperature as prediction
   – Mean absolute error 0.29 (2.57°C)
2. Using densely connected network

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step,
                float_data.shape[-1])))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
steps_per_epoch=500, epochs=20, validation_data=val_gen,
validation_steps=val_steps)
```



Using Gated Recurrent Unit (GRU)

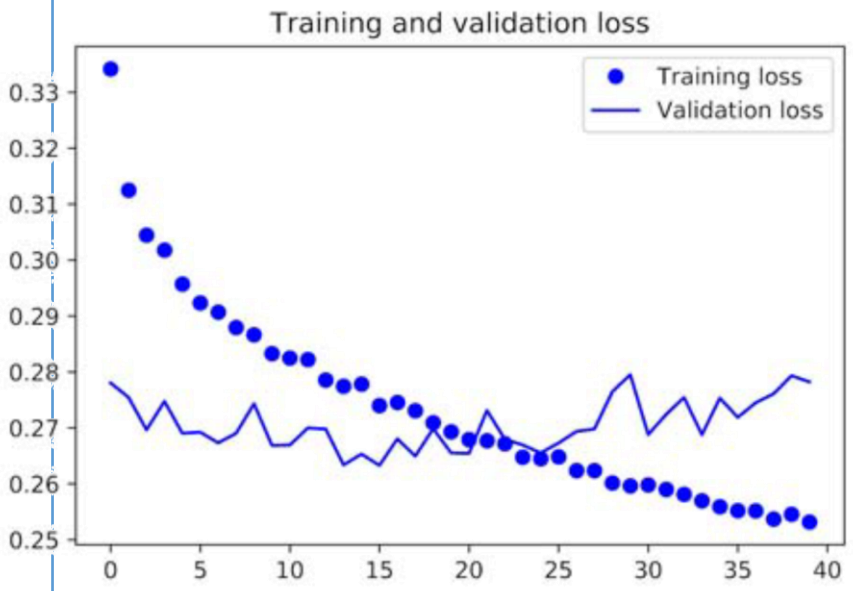
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```



Stacking Recurrent Layers

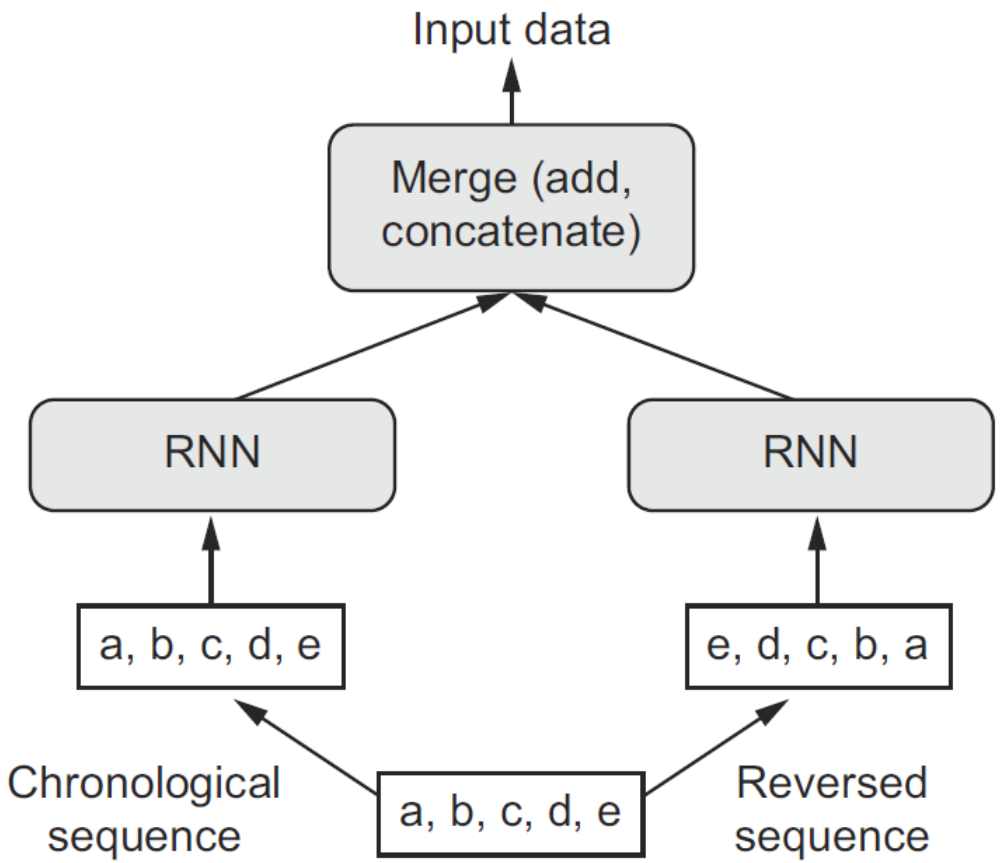
- To stack recurrent layers, all intermediate layers should return their full sequence of outputs (a 3D tensor) (return_sequences=True.)

```
model = Sequential()
model.add(layers.GRU(32,
         dropout=0.1,
         recurrent_dropout=0.5,
         return_sequences=True,
         input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
         dropout=0.1,
         recurrent_dropout=0.5))
         model.add(layers.Dense(1))
         model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
         steps_per_epoch=500,
         epochs=40,
         validation_data=val_gen,
         validation_steps=val_steps)
```



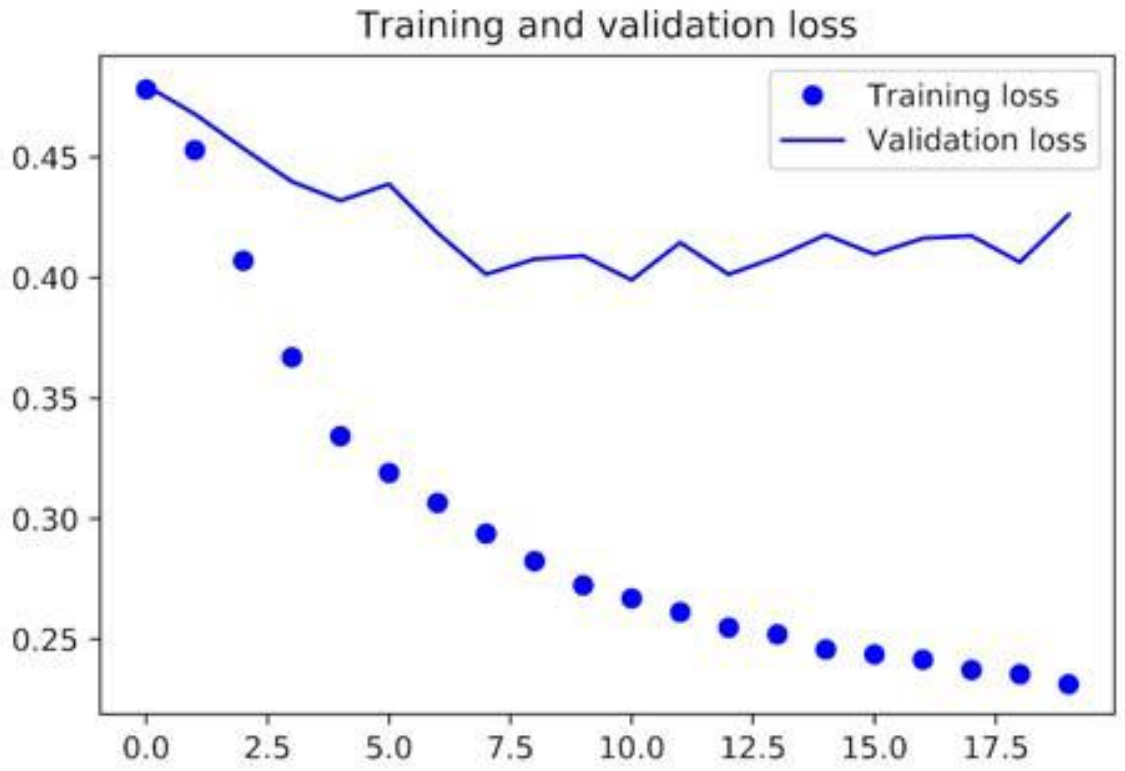
Bidirectional RNN

- A bidirectional RNN exploits the order sensitivity of RNNs
- Commonly used for Natural Language Processing (NLP)



Using Reversed Data for Training

- Perform even worse than the common-sense baseline



Bi-directional GRU for Temperature Prediction

- Get similar performance with regular GRU

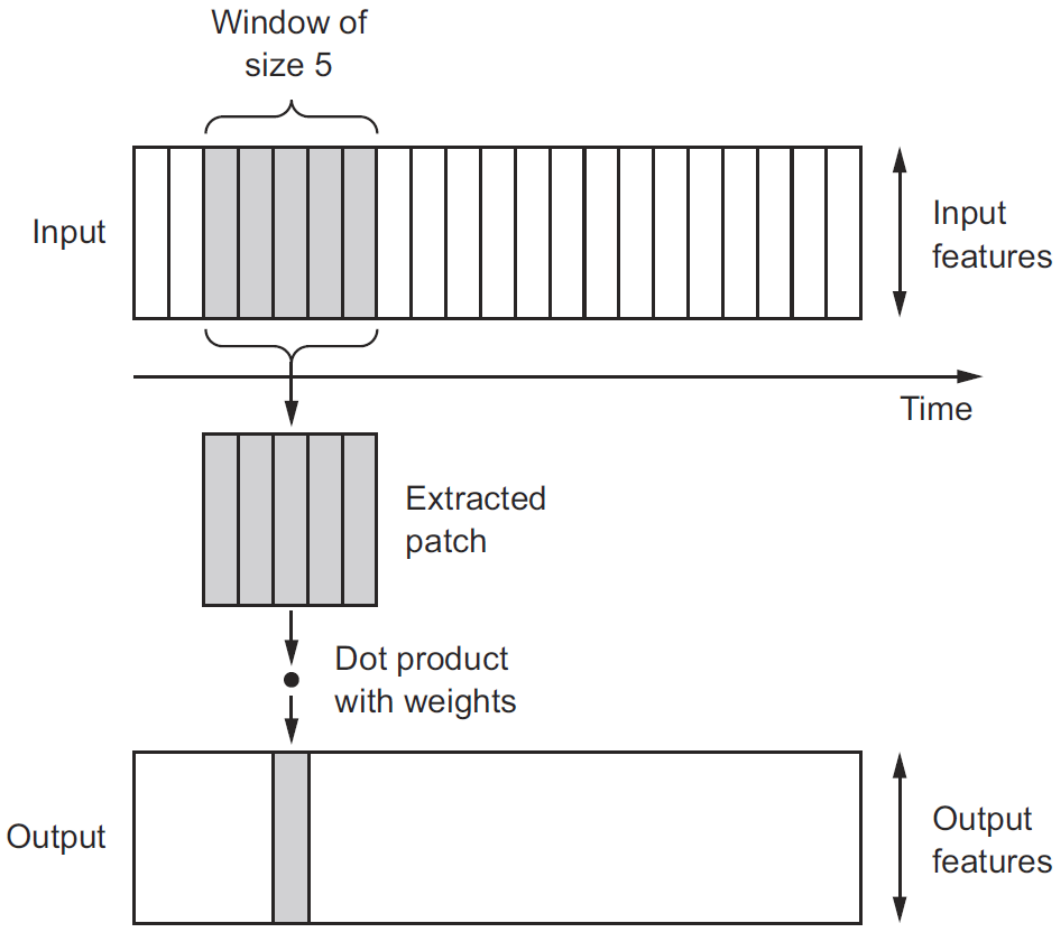
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Bidirectional(
layers.GRU(32), input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))
model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

Going Further

- Adjust the number of units in each recurrent layer in the stacked setup
- Adjust the learning rate used by the RMSprop optimizer
- Try LSTM layers
- Try using a bigger densely connected regressor on top of the recurrent layers
- Don’t forget to eventually run the best-performing models (in terms of validation) on the test set!

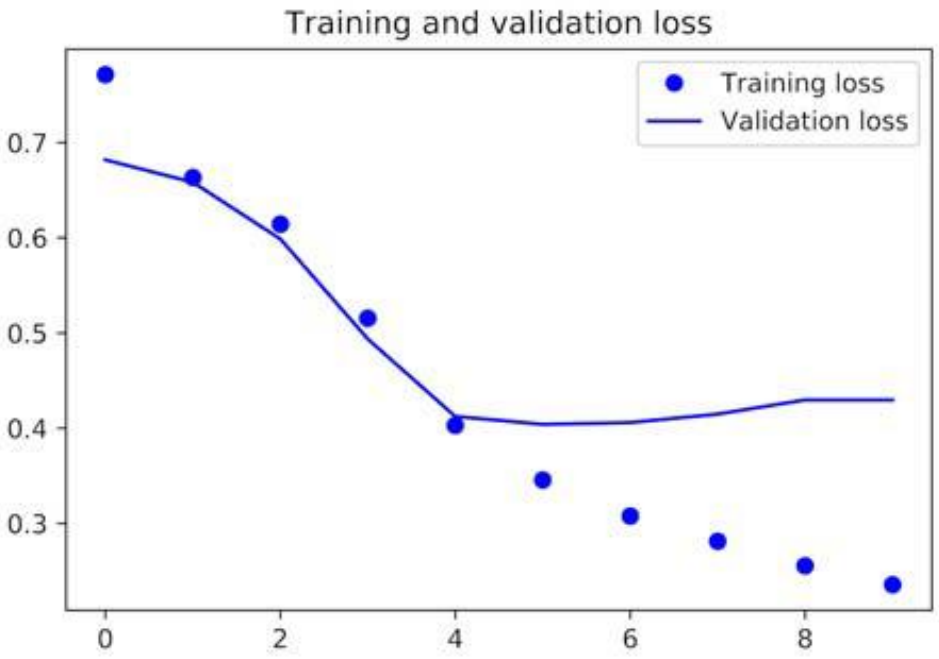
Sequence Processing with ConvNets

- 1-D convolution for sequence data



Building a 1D ConvNet Model for IMDB Dataset

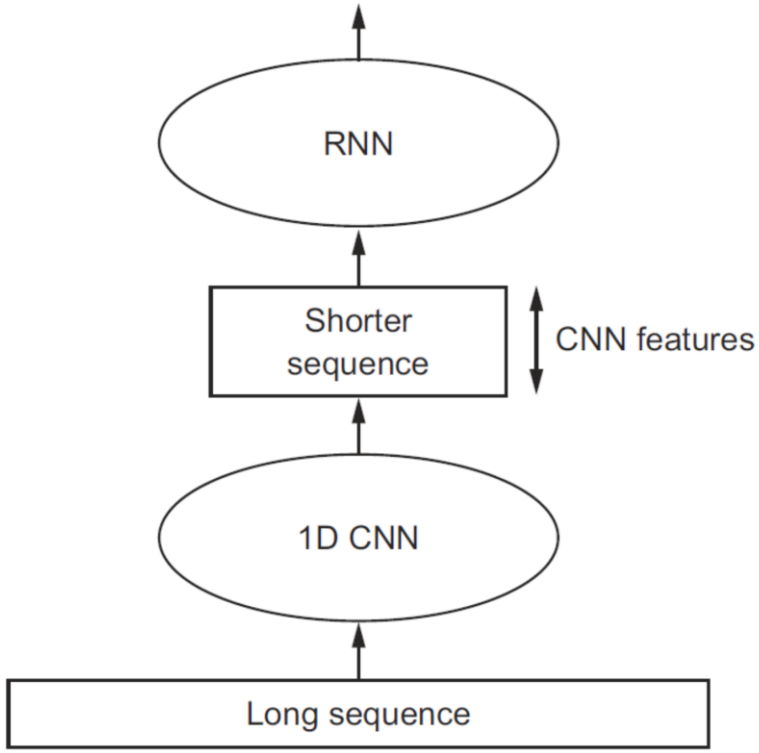
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Embedding(max_features, 128,
input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
model.summary()
model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
              epochs=10,
              batch_size=128,
              validation_split=0.2)
```



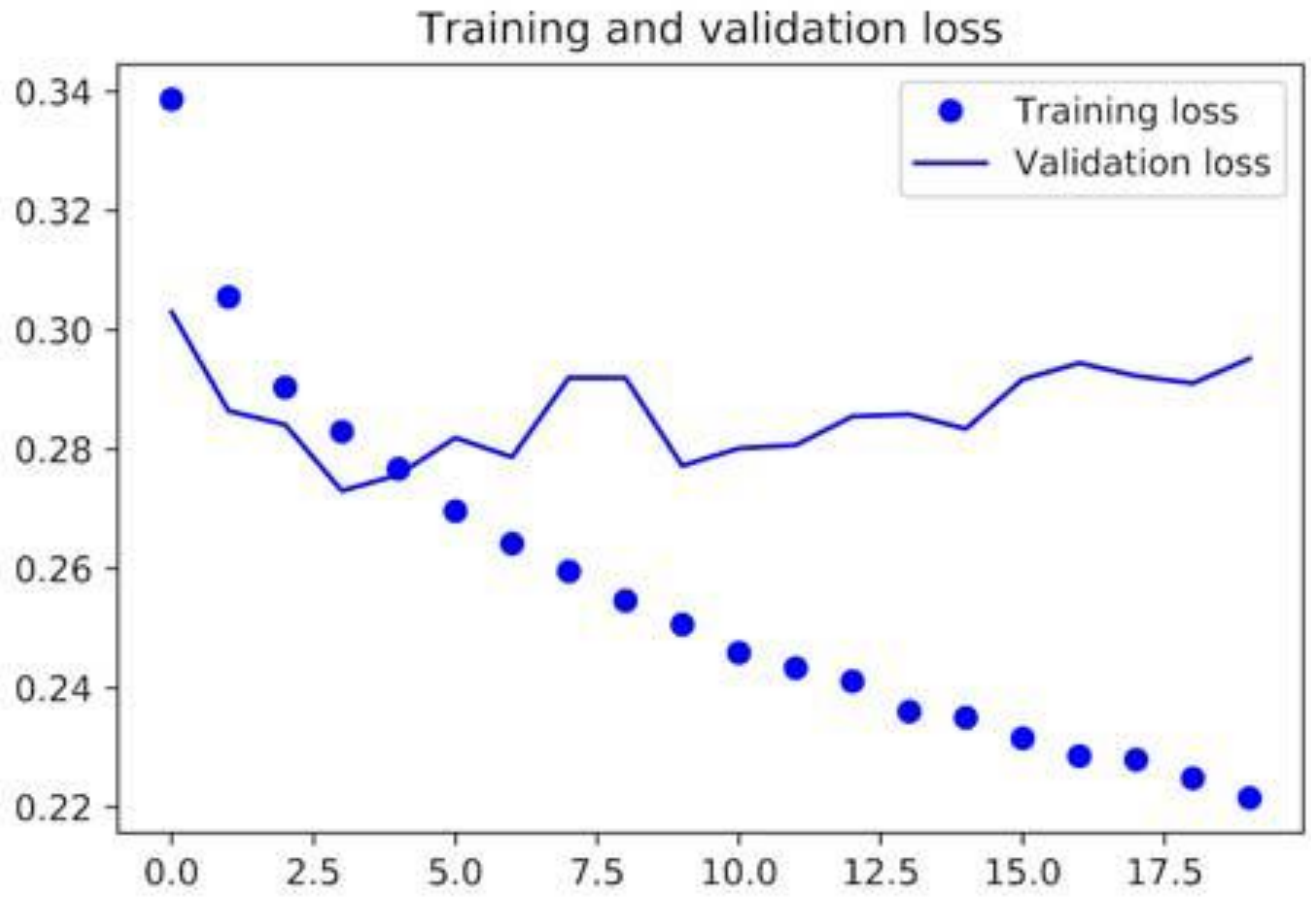
Combining CNN & RNN for Long Sequences

- Prepare a high-resolution data generation and use 1D CNN to shorten the sequence

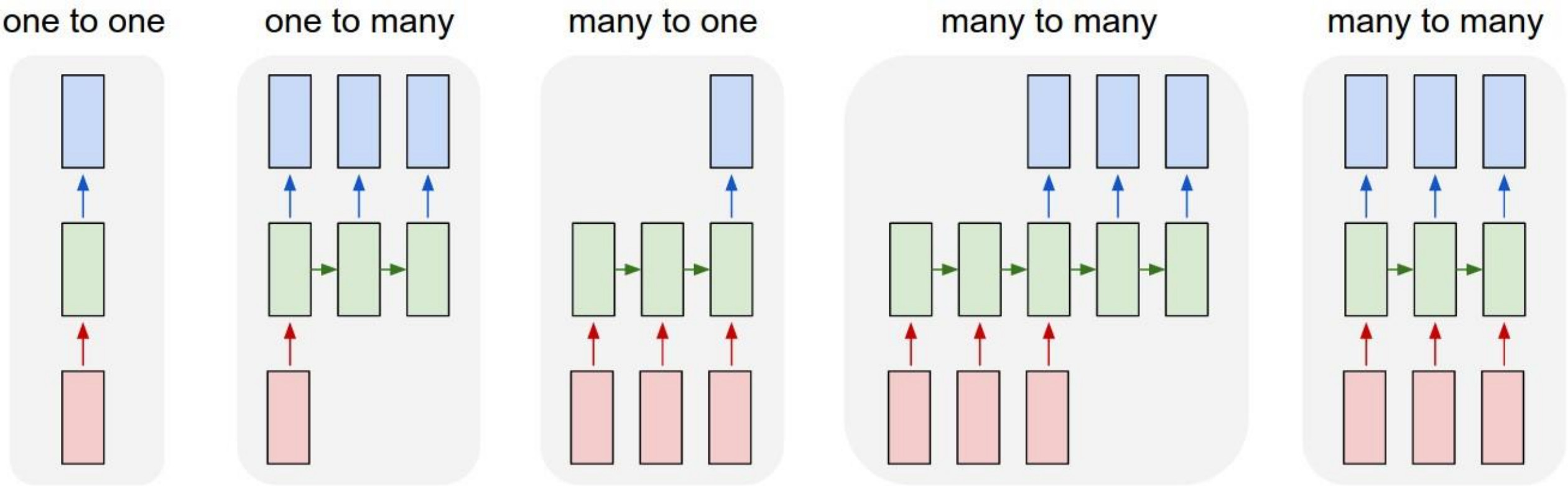
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop
model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GRU(32, dropout=0.1,
recurrent_dropout=0.5))
model.add(layers.Dense(1))
model.summary()
model.compile(optimizer=RMSprop(), loss='mae')
```



Results of 1D ConvNet + RNN on IMDB



Different Types of RNN



http://karpathy.github.io/2015/05/21/rnn-effectiveness/

```
# 1. One-to-one: you could use a Dense layer as you are not processing sequences:
model.add(layers.Dense(output_size, input_shape=input_shape))

# 2. One-to-many: this option is not supported well as chaining models is not very easy
in Keras, so the following version is the easiest one:
model.add(layers.RepeatVector(number_of_times, input_shape=input_shape))
model.add(layers.LSTM(output_size, return_sequences=True))

# 3. Many-to-one: actually, your code snippet is (almost) an example of this approach:
model = Sequential()
model.add(layers.LSTM(1, input_shape=(timesteps, data_dim)))

# 4. Many-to-many: This is the easiest snippet when the length of the input and output
matches the number of recurrent steps:
model = Sequential()
model.add(layers.LSTM(1, input_shape=(timesteps, data_dim), return_sequences=True))

# 5. It's hard to implement in Keras
model = Sequential()
model.add(layers.LSTM(1, input_shape=(timesteps, data_dim), return_sequences=True))
model.add(Lambda(lambda x: x[:, -N:, :]
```

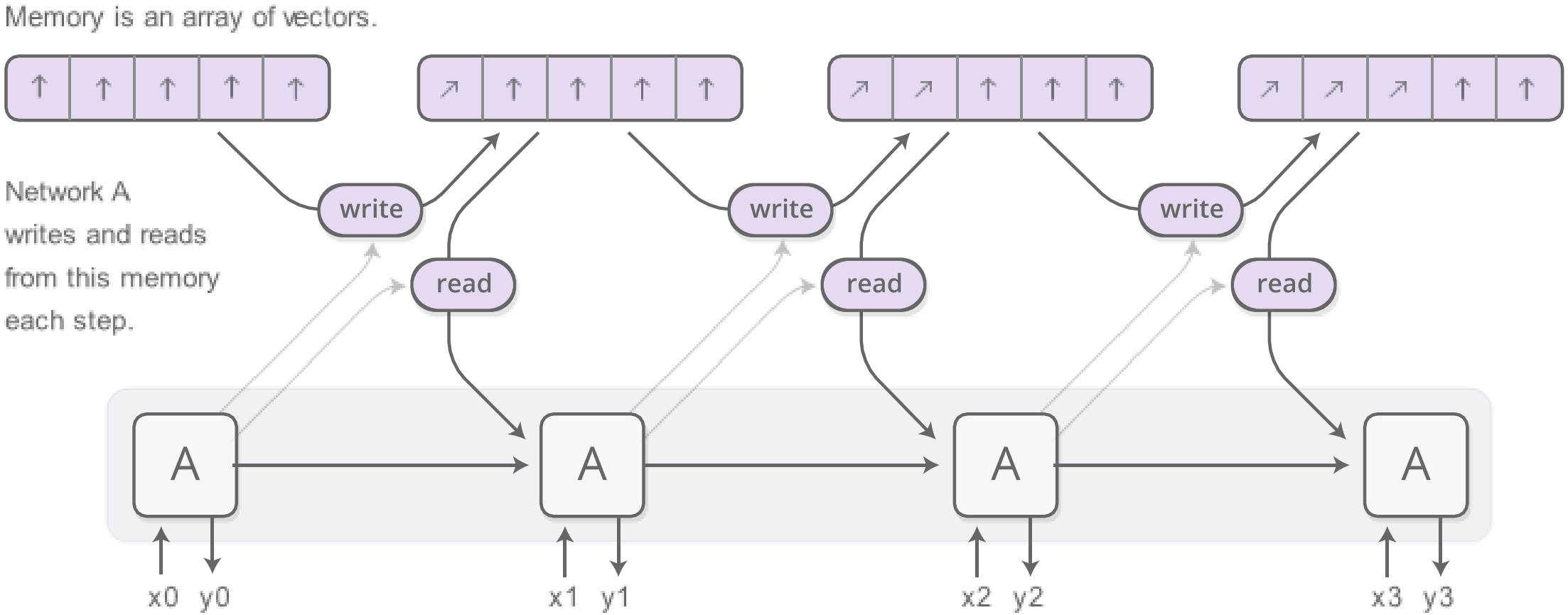
Attention and Augmented RNNs

1. Neural Turning Machine
2. Attention Mechanism
3. Adaptive Computation Time
4. Neural Programmer

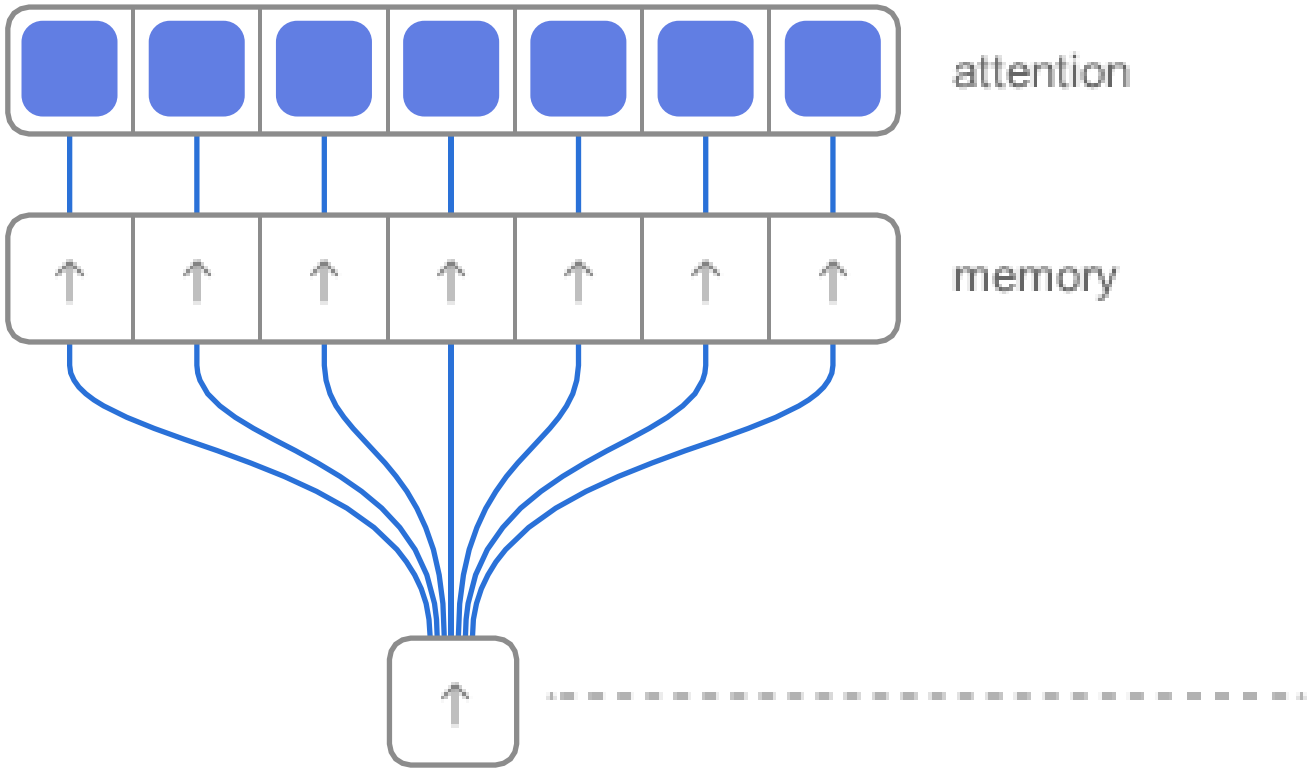
https://distill.pub/2016/augmented-rnns/

Neural Turing Machine

- Combine RNN with external memory bank



Making Memory Read/Write Differentiable

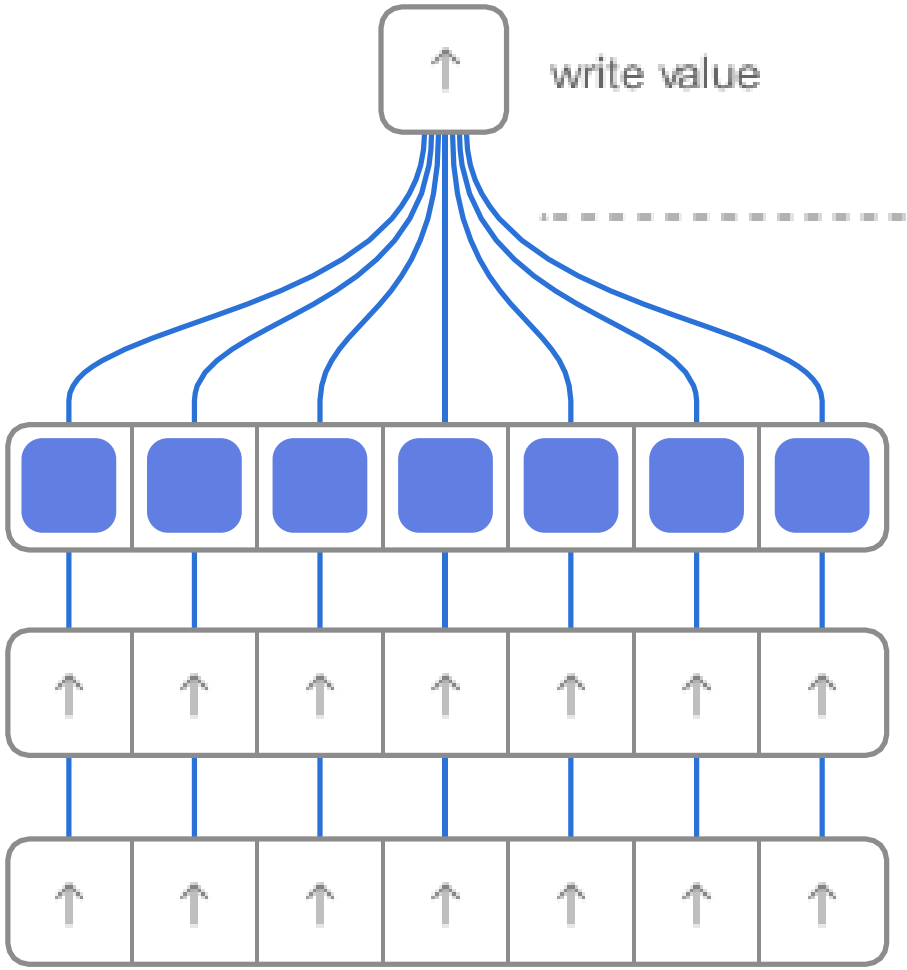


The RNN gives an attention distribution which describe how we spread out the amount we care about different memory positions.

The read result is a weighted sum.

$$r \leftarrow \sum_i a_i M_i$$

NTM Writing

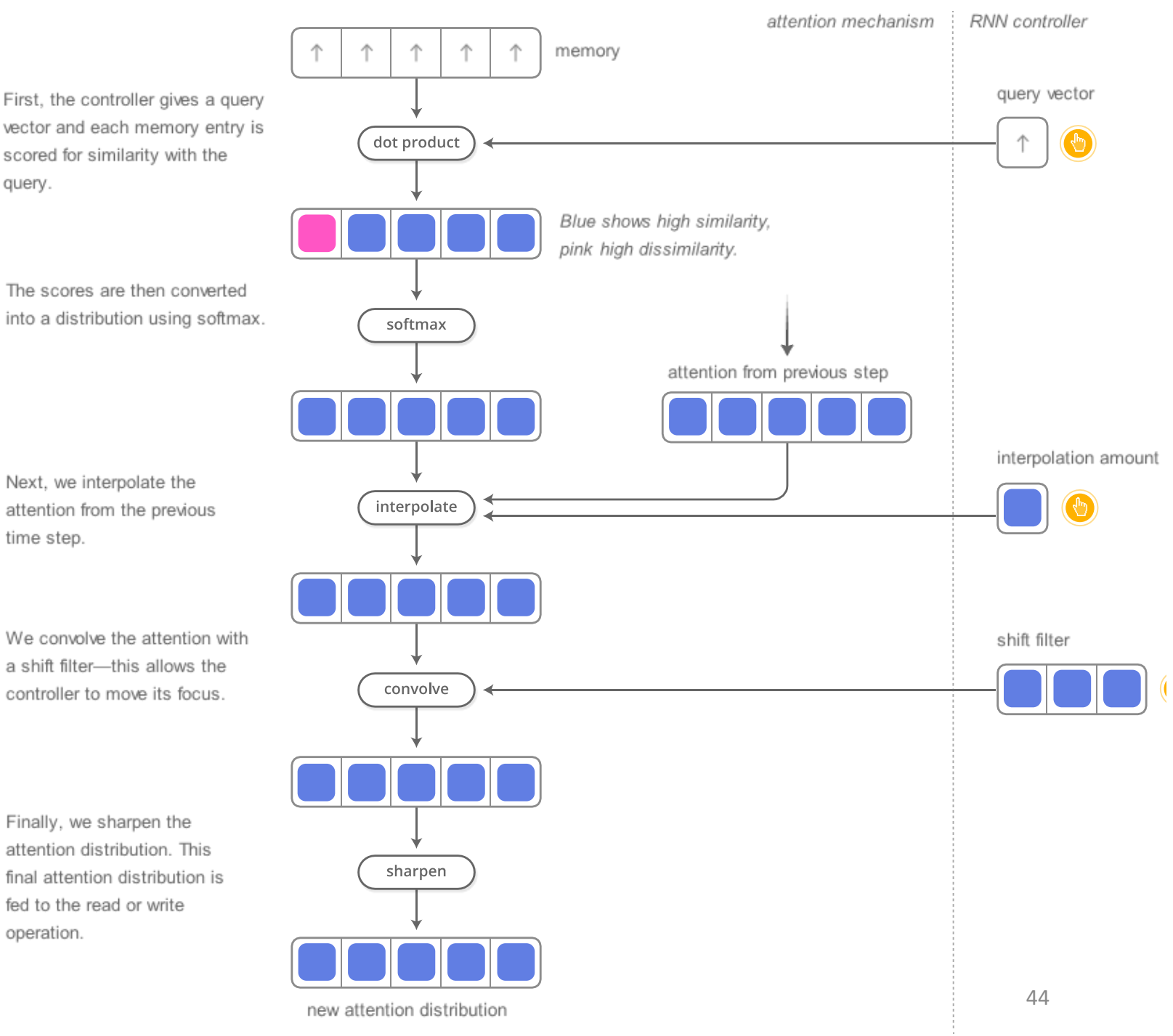


Instead of writing to one location, we write everywhere, just to different extents.

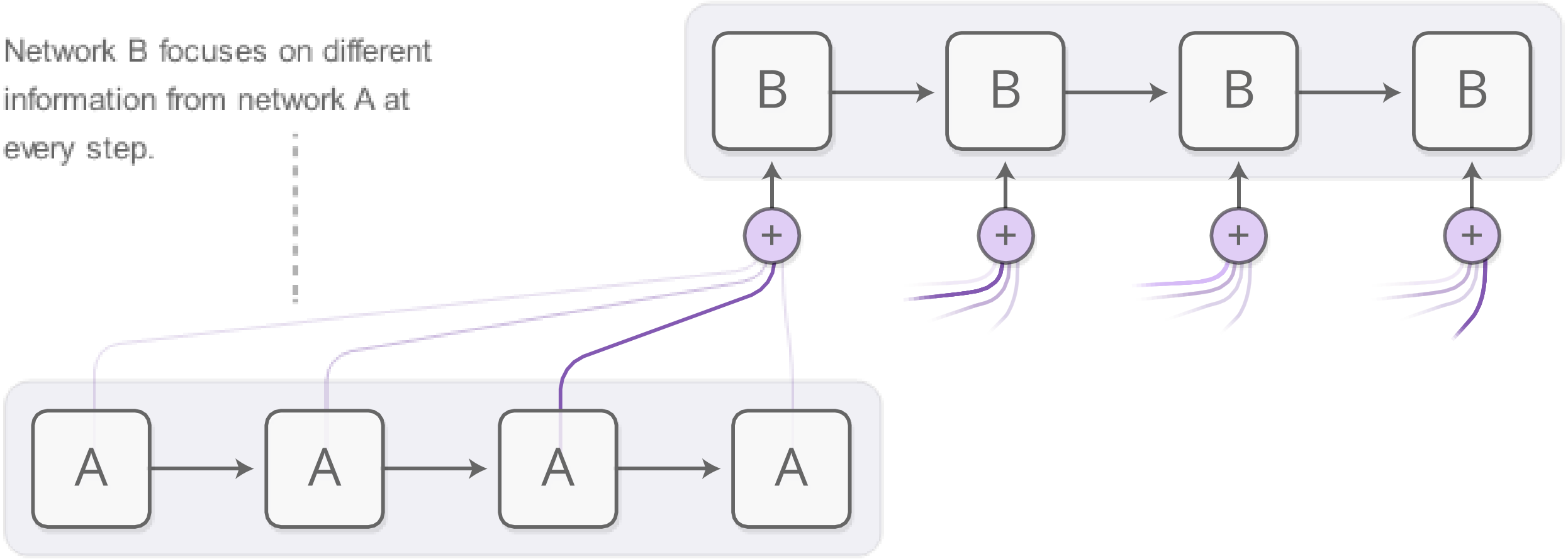
The RNN gives an attention distribution, describing how much we should change each memory position towards the write value.

$$M_i \leftarrow a_i w + (1-a_i) M_i$$

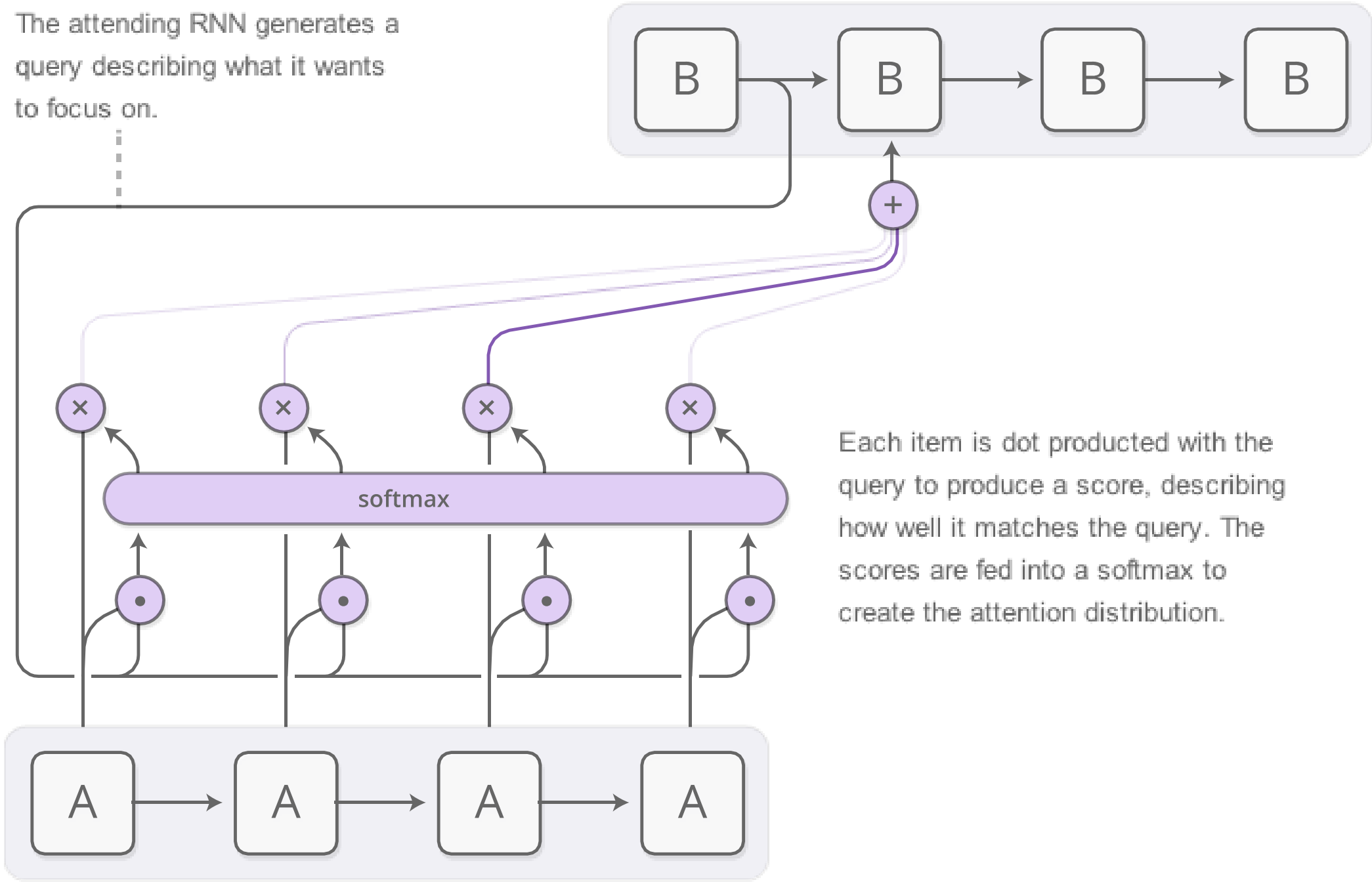
Content-based Attention & Location-based Attention



Attention Mechanism

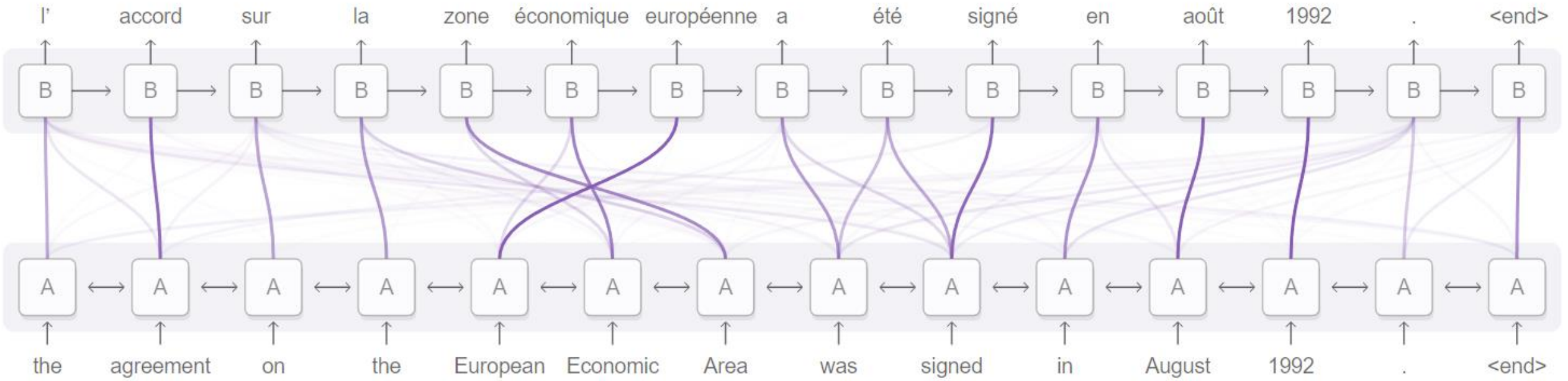


The attending RNN generates a query describing what it wants to focus on.



Each item is dot producted with the query to produce a score, describing how well it matches the query. The scores are fed into a softmax to create the attention distribution.

Applications of Attention



A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.

A stop sign is on a road with a mountain in the background.

An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling

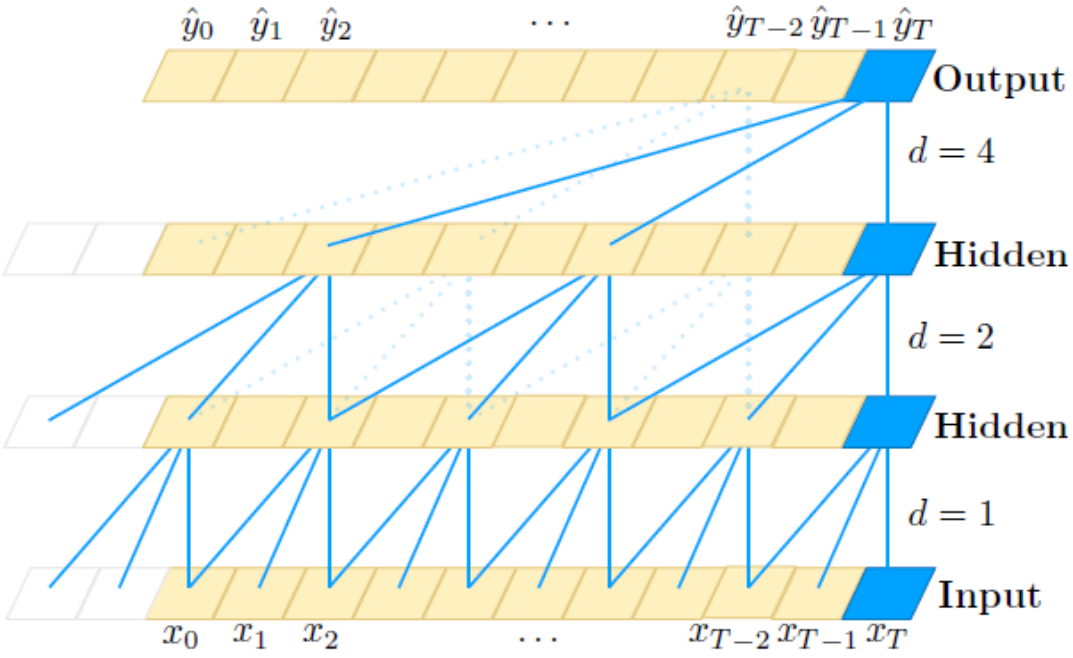
- Shaojie Bai, J. Zico Kolter, Vladlen Koltun (CMU & Intel Labs), April, 2018

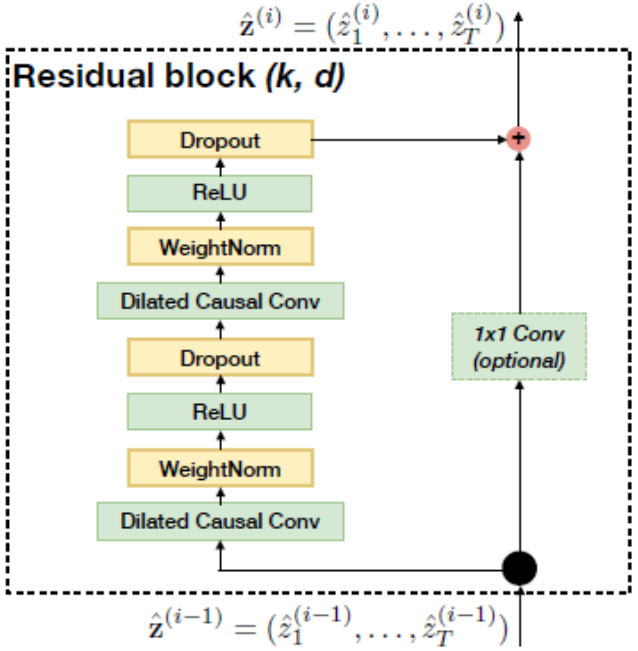
- The models are evaluated on many RNN benchmarks
- A simple temporal CNN model outperforms RNN / LSTMs across a diverse range of tasks and datasets!

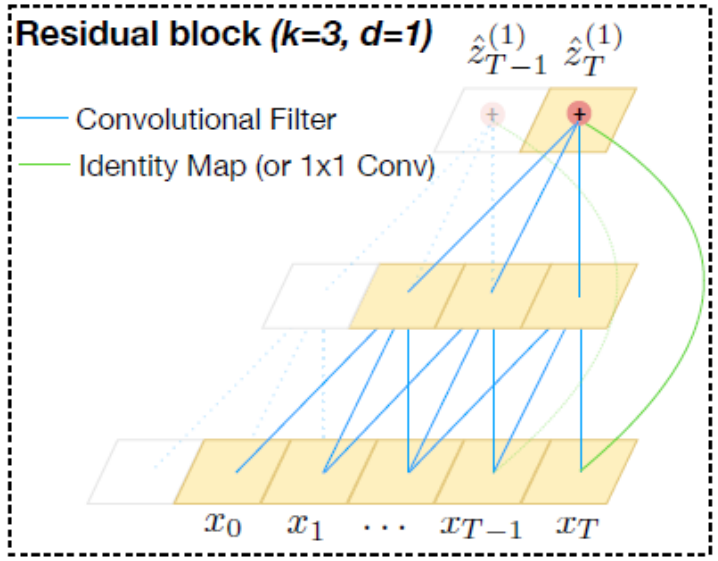

We knew CNN is better than RNN/LSTM for a while.

Temporal Convolutional Network (TCN)

- Dilated causal convolution







Experimental Results

Sequence Modeling Task	Model Size ($\approx$)	Models			
		LSTM	GRU	RNN	TCN
Seq. MNIST (accuracy^h)	70K	87.2	96.2	21.5	**99.0**
Permuted MNIST (accuracy)	70K	85.7	87.3	25.3	**97.2**
Adding problem T=600 (loss$^\ell$)	70K	0.164	**5.3e-5**	0.177	**5.8e-5**
Copy memory T=1000 (loss)	16K	0.0204	0.0197	0.0202	**3.5e-5**
Music JSB Chorales (loss)	300K	8.45	8.43	8.91	**8.10**
Music Nottingham (loss)	1M	3.29	3.46	4.05	**3.07**
Word-level PTB (perplexity$^\ell$)	13M	**78.93**	92.48	114.50	88.68
Word-level Wiki-103 (perplexity)	-	48.4	-	-	**45.19**
Word-level LAMBADA (perplexity)	-	4186	-	14725	**1279**
Char-level PTB (bpc$^\ell$)	3M	1.36	1.37	1.48	**1.31**
Char-level text8 (bpc)	5M	1.50	1.53	1.69	**1.45**

The Fall of RNN/LSTM

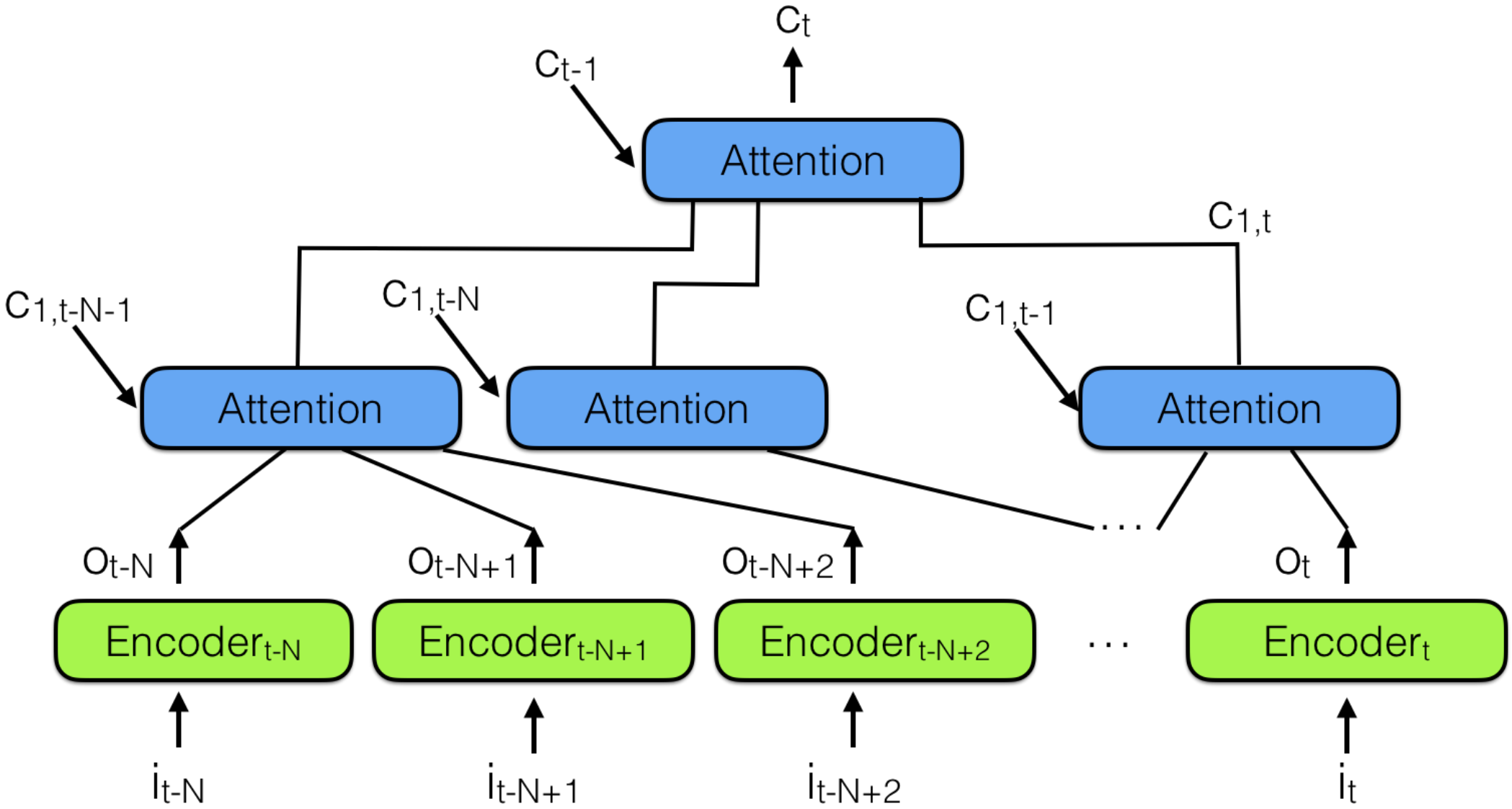
Eugenio Culurciello

Professor of Biomedical Engineering, Purdue University

April 13, 2018

https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0

Hierarchical Neural Attention Encoder




I got a dig bick
You that read wrong
You read that wrong too

Attention is All You Need!

A. Waswani et al., *NIPS*, 2017

Google Brain & University of Toronto

References

- http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- Francois Chollet, “Deep Learning with Python,” Chapter 6
- https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0