MSE | MASTER OF SCIENCE IN ENGINEERING

Recurrent Neural Networks

Part 1: Simple RNNs, Generative RNNs

Jean Hennebert
Martin Melchior

Overview

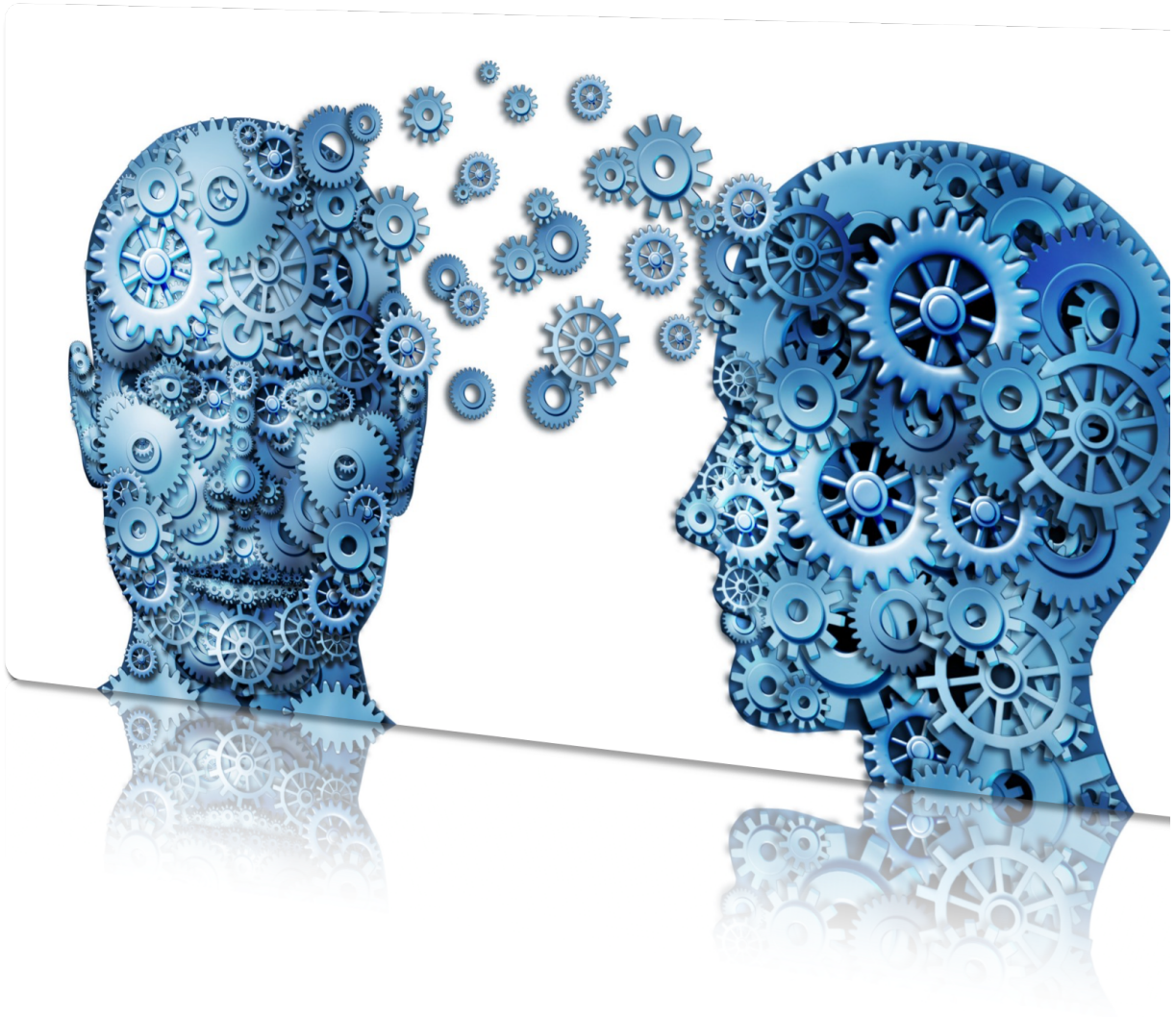
Recap

Why RNNs, Sample Applications

Simple (‘Vanilla’) RNNs

Generative RNNs

CNN3 recaps



Keras Functional API

Transfer Learning

Autoencoders

Leader Board

RECAP - Keras functional API

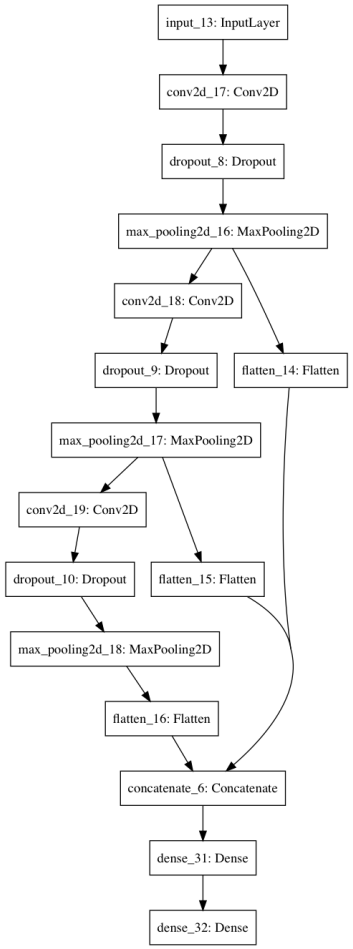
- The **sequential** model corresponds to a regular stack of layers
- The **functional** API is used for non sequential architectures

- The functional API allows for
  - Multiple path in computational graph through layer sharing
  - Multiple inputs, multiple outputs
  - Shared input layer
  - Shared feature extraction layer

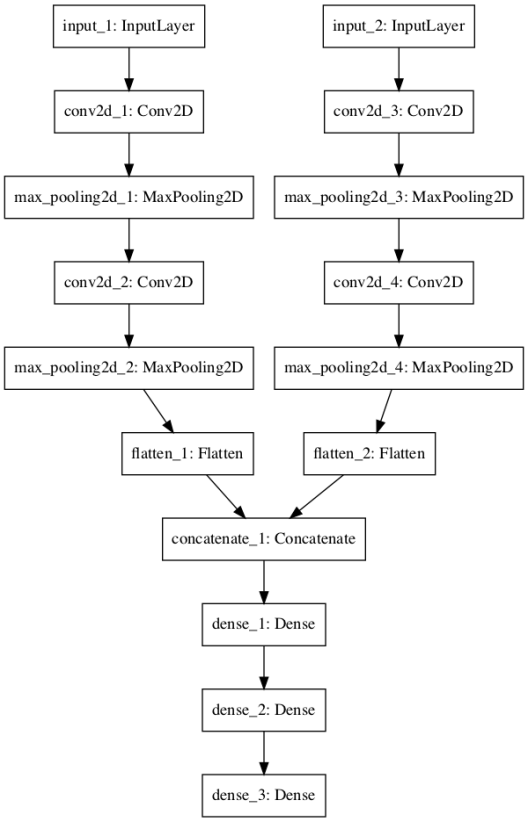
- Objects are **callable** when the class definition includes a `__call__` method. It allows to "call" the object as we would "call" a method with the syntax `obj()`.

RECAP - Keras functional API

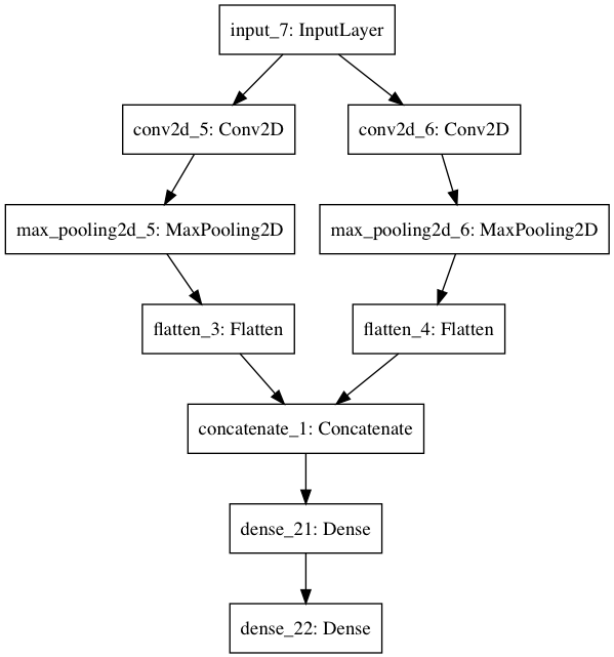
Multiple features



Multiple inputs

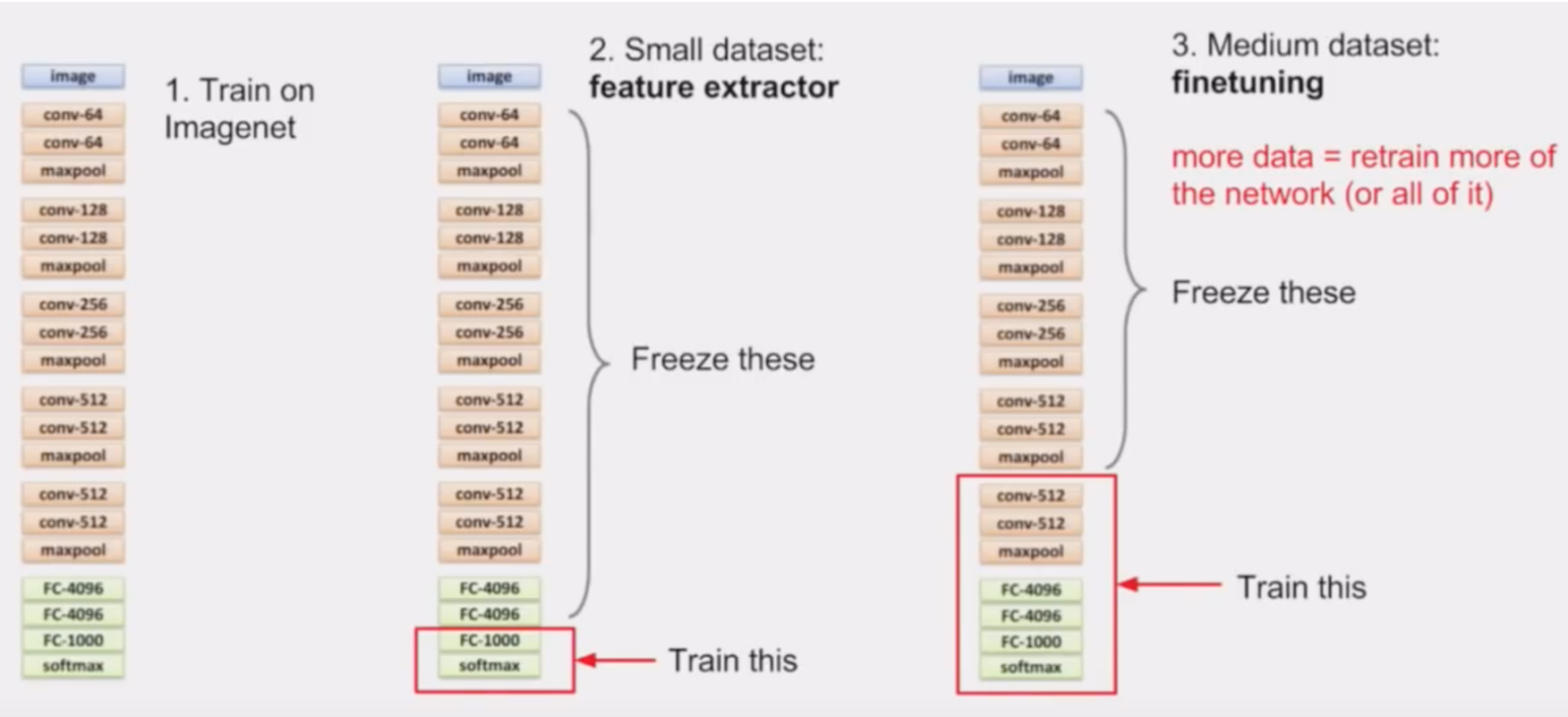


Multiple paths



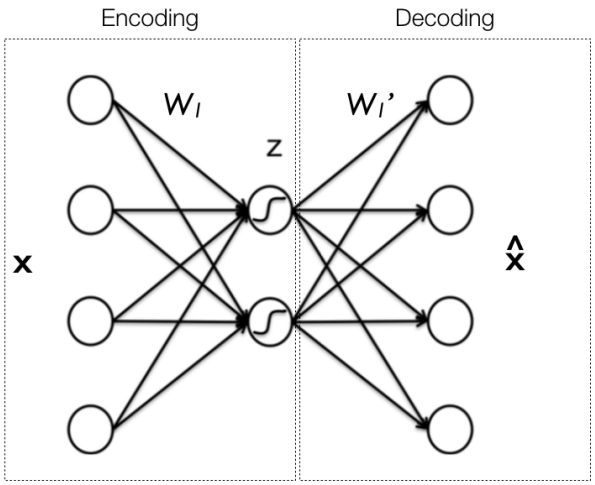
RECAP - Transfer learning

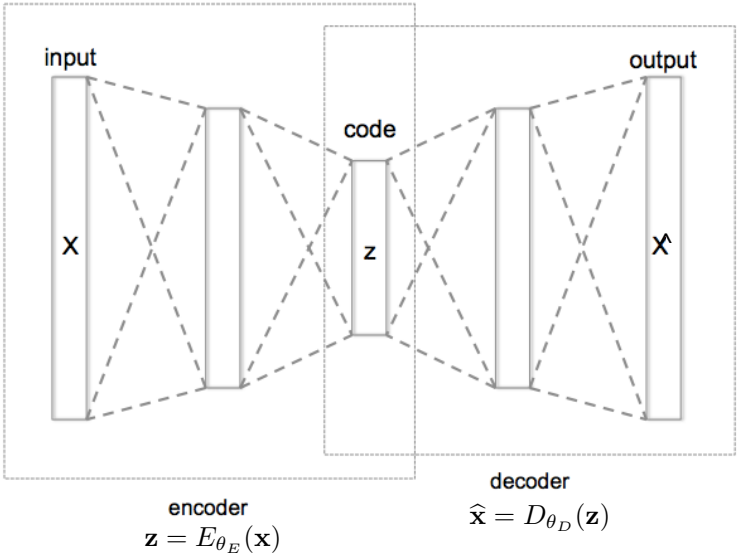
- **Transfer learning** is about using knowledge learned from tasks for which a lot of labelled data is available in settings where only little labelled data is available.



RECAP - Autoencoders

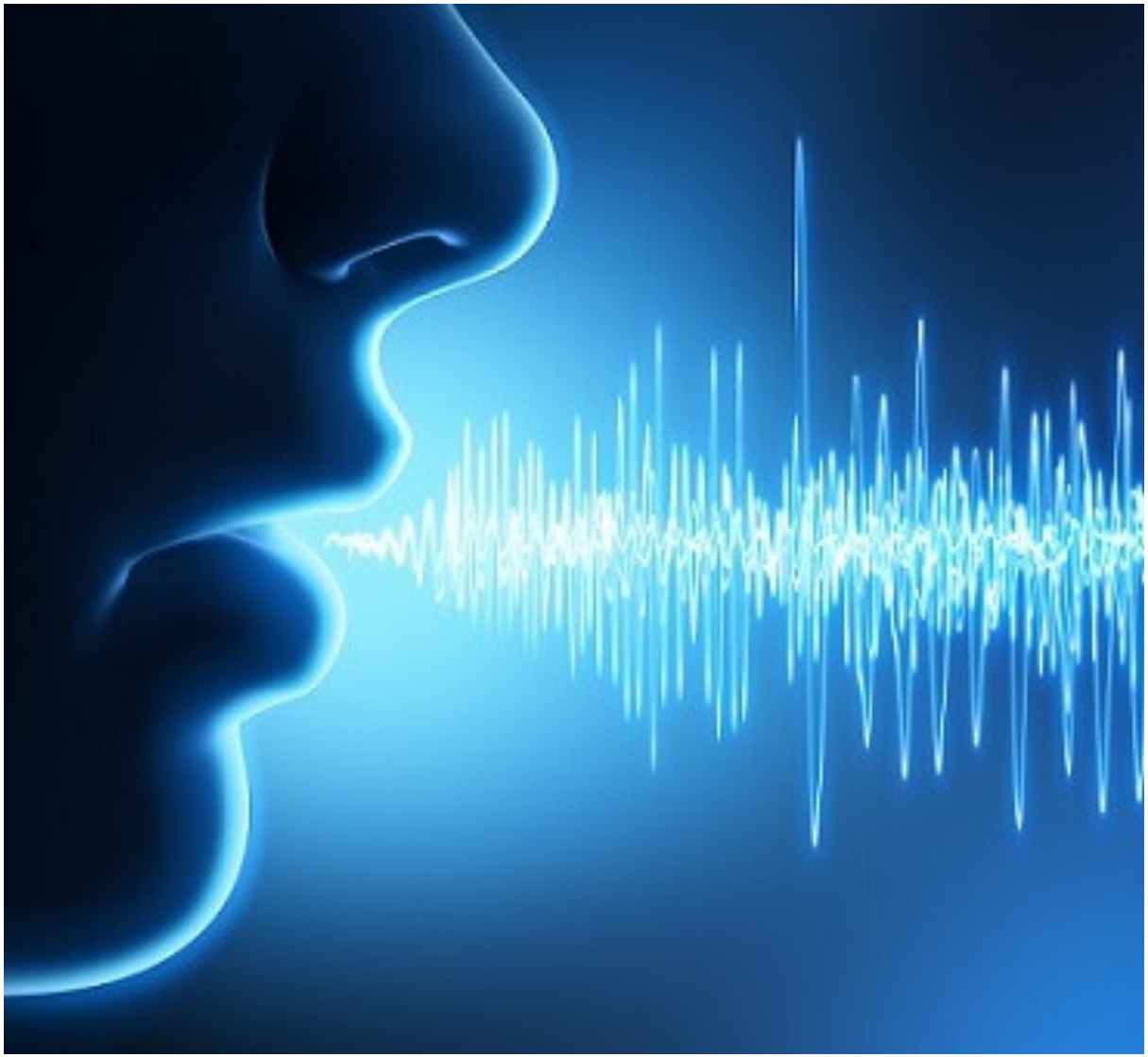
An **autoencoder** is a neural network trained to reproduce its input and able to discover “structures” and “efficient codings” of the input space.



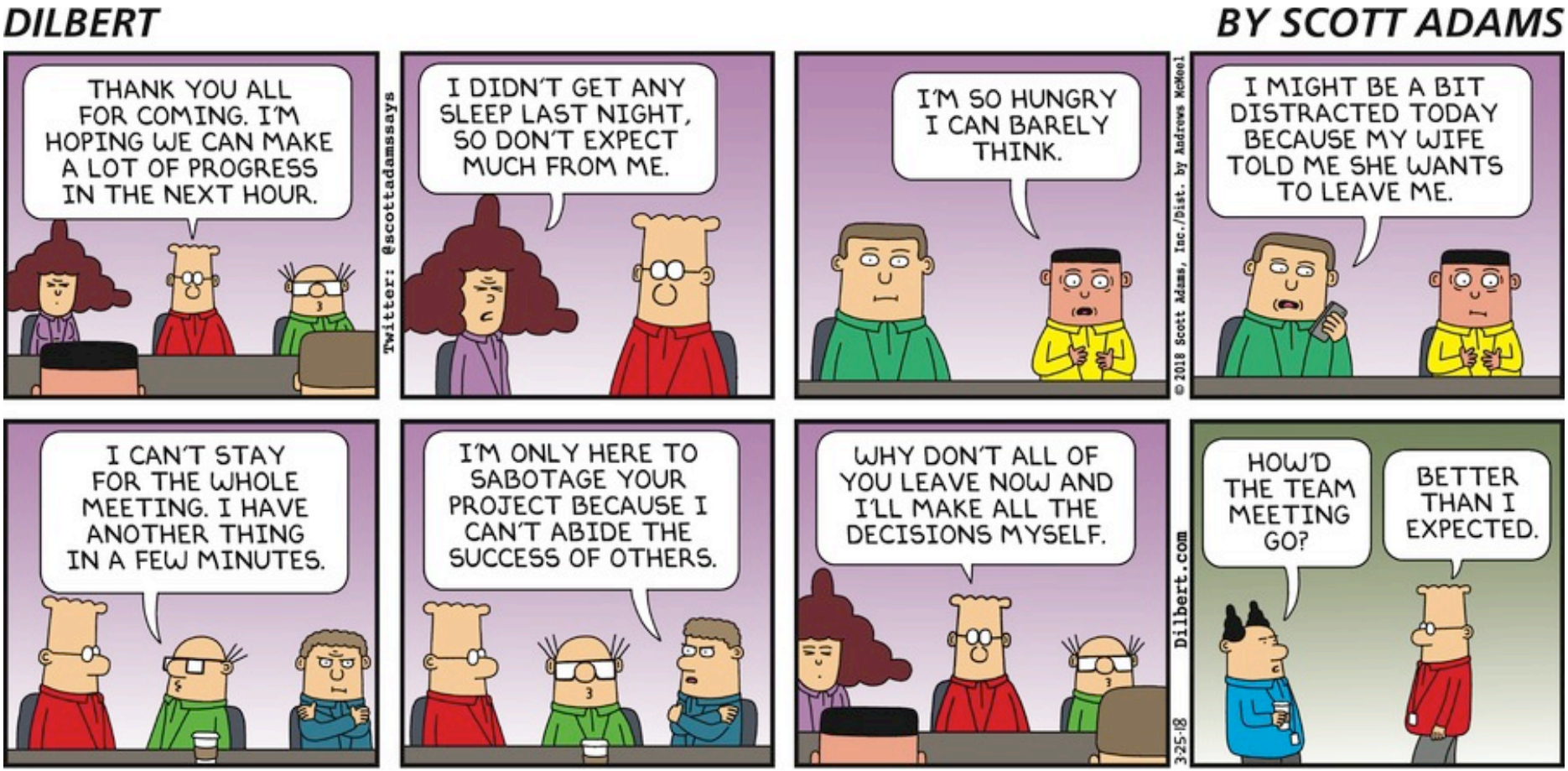


- Discover a good internal representation of the input from which we can make some sense
- Diabolo topology: shallow dense, stacked dense autoencoders or CNN autoencoders
- 5 usages: data compression, feature extraction, de-noising, image in-painting, pre-training of deep network

Why RNNs




HOW'D THE TEAM MEETING GO?
BETTER THAN I EXPECTED.
Dilbert.com
325-10


DILBERT
BY SCOTT ADAMS
THANK YOU ALL FOR COMING. I'M HOPING WE CAN MAKE A LOT OF PROGRESS IN THE NEXT HOUR.
I DIDN'T GET ANY SLEEP LAST NIGHT, SO DON'T EXPECT MUCH FROM ME.
I'M SO HUNGRY I CAN BARELY THINK.
I MIGHT BE A BIT DISTRACTED TODAY BECAUSE MY WIFE TOLD ME SHE WANTS TO LEAVE ME.
I CAN'T STAY FOR THE WHOLE MEETING. I HAVE ANOTHER THING IN A FEW MINUTES.
I'M ONLY HERE TO SABOTAGE YOUR PROJECT BECAUSE I CAN'T ABIDE THE SUCCESS OF OTHERS.
WHY DON'T ALL OF YOU LEAVE NOW AND I'LL MAKE ALL THE DECISIONS MYSELF.
HOW'D THE TEAM MEETING GO?
BETTER THAN I EXPECTED.
Twitter: @scottadamssays
© 2018 Scott Adams, Inc./Dist. by Andrews McMeel
Dilbert.com
3-25-18

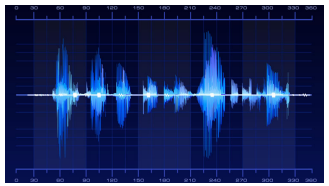
Why RNNs?

- CNN & MLP are good at describing and classifying images. Are they good for describing a scene in a movie?
- The model needs to learn and remember a context, i.e. automatically discover and track useful information (e.g. location, day time, season) — just as a CNN that discovers edges, shapes, and faces.
- RNNs learn to memorise context information that may be important to draw conclusions on observations made later on.
- **RNN helps wherever we need context from the previous input**.

RNN's: Specialised for Processing Sequences

Similar to CNN's that are specialised for processing grids such as images.

Timeseries



Words in sentences

Today, the weather above the sticky fog on **Rigi** is beautiful.
This morning, the train brought us **there**.

Characters in words

w-e-a-t-h-e-r

Successive elements are not independent.

- Local dependencies sticky fog
- Sometimes long-term (non-local) dependencies (e.g. for reasoning)

Rigi … there

Interesting Links

- A. Karpathy
  "The unreasonable effectiveness of Recurrent Neural Networks"
  Very nice and famous blog with cool applications
  (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

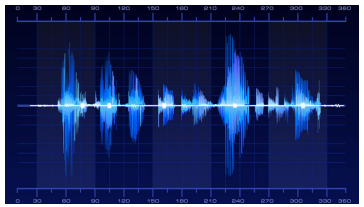
- A. Ng, Deep Learning Course on Coursera (Course on "Sequence Models")

- Fei Fei Li, Justing Johnston, Lecture 10, May 2018:
  https://www.youtube.com/watch?v=6niqTuYFZLQ

Many Applications

- Speech recognition:
  - Assistants (Alexa, Siri, etc).
  - Video captioning, transcription to text
  - etc.

audio signal



→ "Today, the weather on Rigi above the fog is beautiful."

- Sentiment classification:
  - Classification of text: e.g. movie ratings
  - Emojification

"Today, the weather on Rigi above the fog is beautiful." → 😀

- Machine translation
  - DeepL
  - Google Translate

"Today, the weather on Rigi above the fog is beautiful." → "Heute ist das Wetter auf der Rigi oberhalb des Nebels schön."

- Captioning, subtitling
  - images (see e.g. https://arxiv.org/pdf/1411.4555v2.pdf)
  - youtube videos

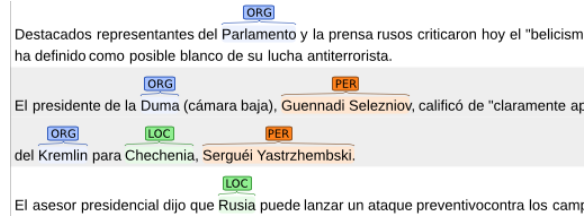
→ "Sun shining above the clouds."

Many Applications

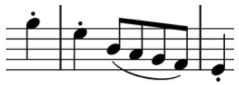
- Chatbots, Question/Answering
  - Siri, Alexa, etc.
  - Google Duplex https://ai.googleblog.com/2018/05/duplex-ai-system-for-natural-conversation.html

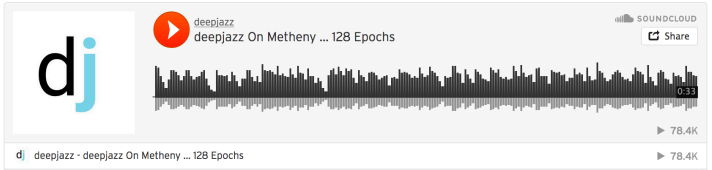
- Named entity recognition (NER)
  - Flag names in sentences

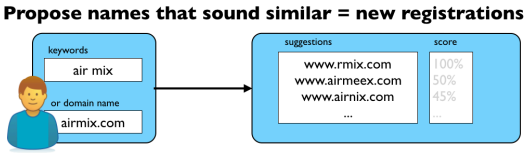
Today, the weather above the fog on Rigi is beautiful.
0 0 0 0 0 0 0 1 0 0

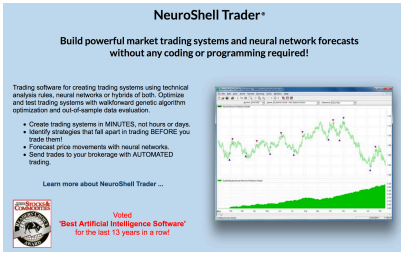


- Word or music generation
  - Music:
    - Magenta: https://magenta.tensorflow.org/
    - https://deepjazz.io/
  - Word:
    - Domain names of week 1
    - See exercises









- Time sequence modelling, prediction

Sequence Models

Input sequence x with length T_x

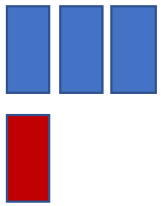
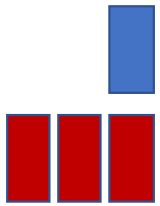
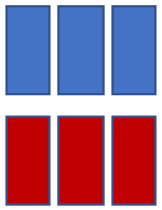
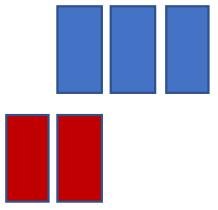
$$x = (x_1, x_2, \ldots, x_{T_x})$$

→

Output sequence y with length T_y

$$y = (y_1, y_2, \ldots, y_{T_y})$$

Typically, the elements of x and y are ***multi-dimensional vectors*** and are ***parametrised*** by an ***index*** *or* ***'time'*** to describe the position in the sequence.

	image tagging, music generation	**sentiment analysis**	**named entity recognition**	**language translation, speech recognition, chatbots**
	one-to-many	**many-to-one**	**many-to-many**	**many-to-many**
out / **in**				
	$T_x = 1, T_y \geq 1$	$T_x \geq 1, T_y = 1$	$T_x \geq 1, T_y \geq 1$ $T_x = T_y$	$T_x \geq 1, T_y \geq 1$ $T_x \neq T_y$

Example

Today, the weather above the fog line is beautiful.

x_1 x_2 x_3 $\cdots$ x_9

$T_x = 9$, word representation vectors

Heute ist das Wetter oberhalb der Nebelgrenze schön.

y_1 y_2 y_3 $\cdots$ y_8

$T_y = 8$, word representation vectors

Modelling Sequences with MLPs or CNNs?

- Discuss in groups of 2-3 how you could design models to process sequences by using MLPs or CNNs.
- What difficulties do you experience?

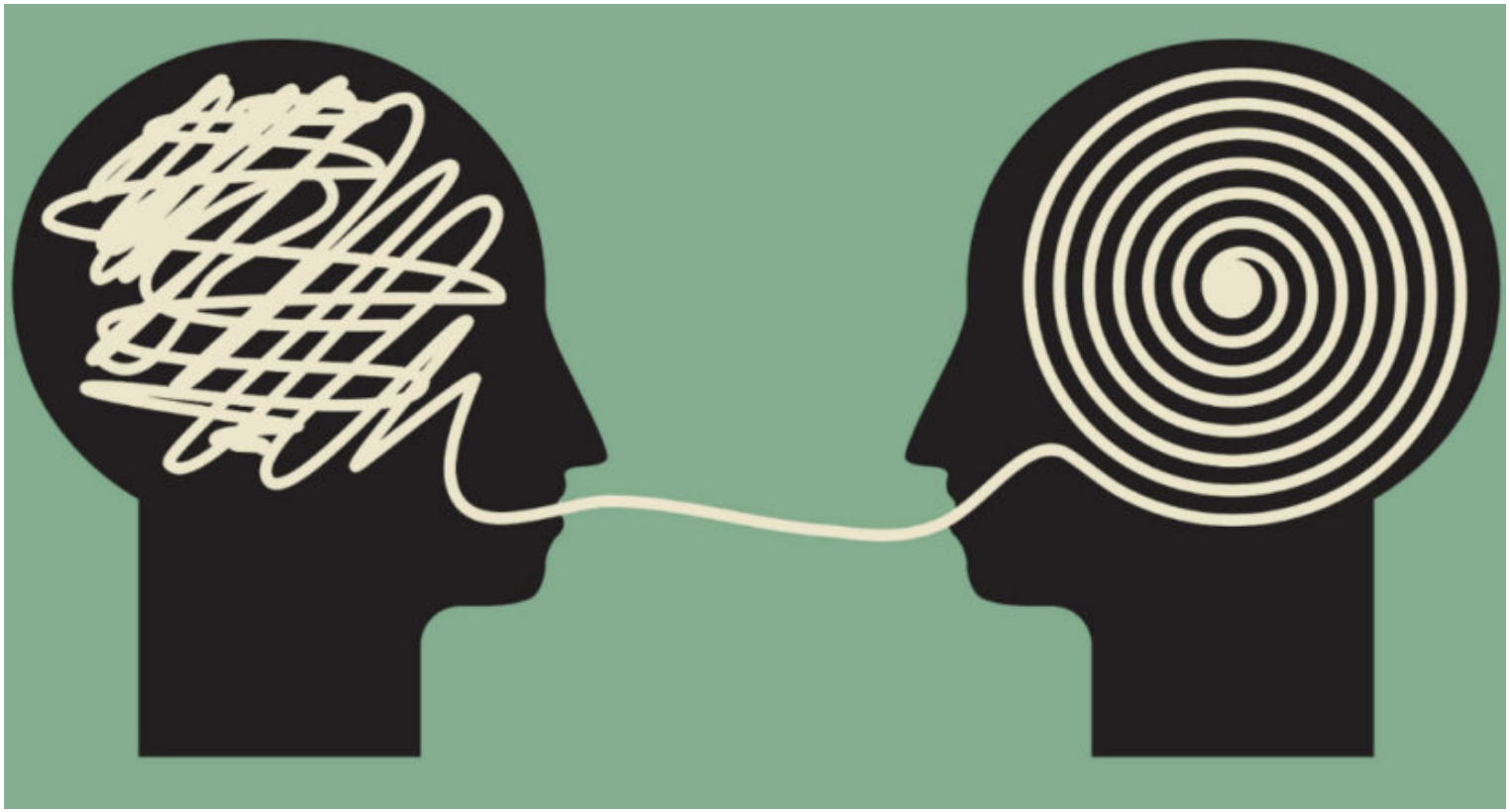
Efficient Representation for Sequences

- Handle sequences of variable length
  - Possibly both, for input and output.
  - However, in practice, a fixed maximum length is set (with suitable padding)

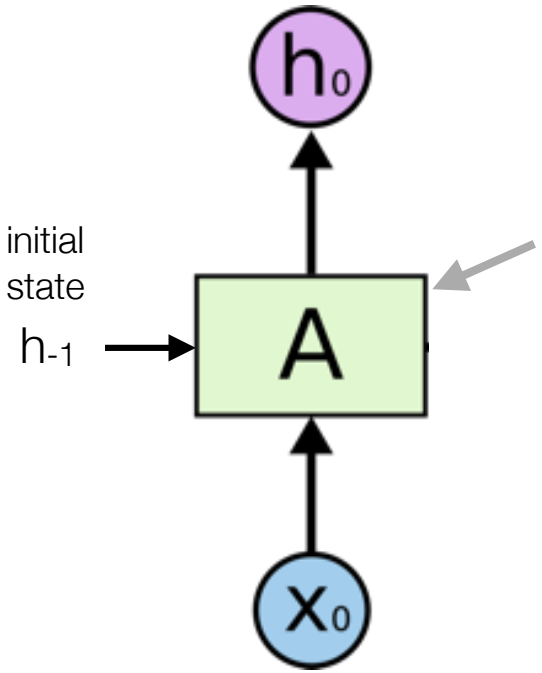
- Represent inherent structure
  - Elements are not independent - similar to the images where a spatial structure is identified (local connectivity) there is also an underlying structure assumed for sequences (e.g. syntactic form or semantics of sentences).

- Gain efficiency by sharing parameters:
  - Use the same parameters across different parts of the sequences — similar to CNNs that share parameters across different parts of images by using filters.
  - Parameter sharing also needed to cover sequences of arbitrary length.

Simple RNNs



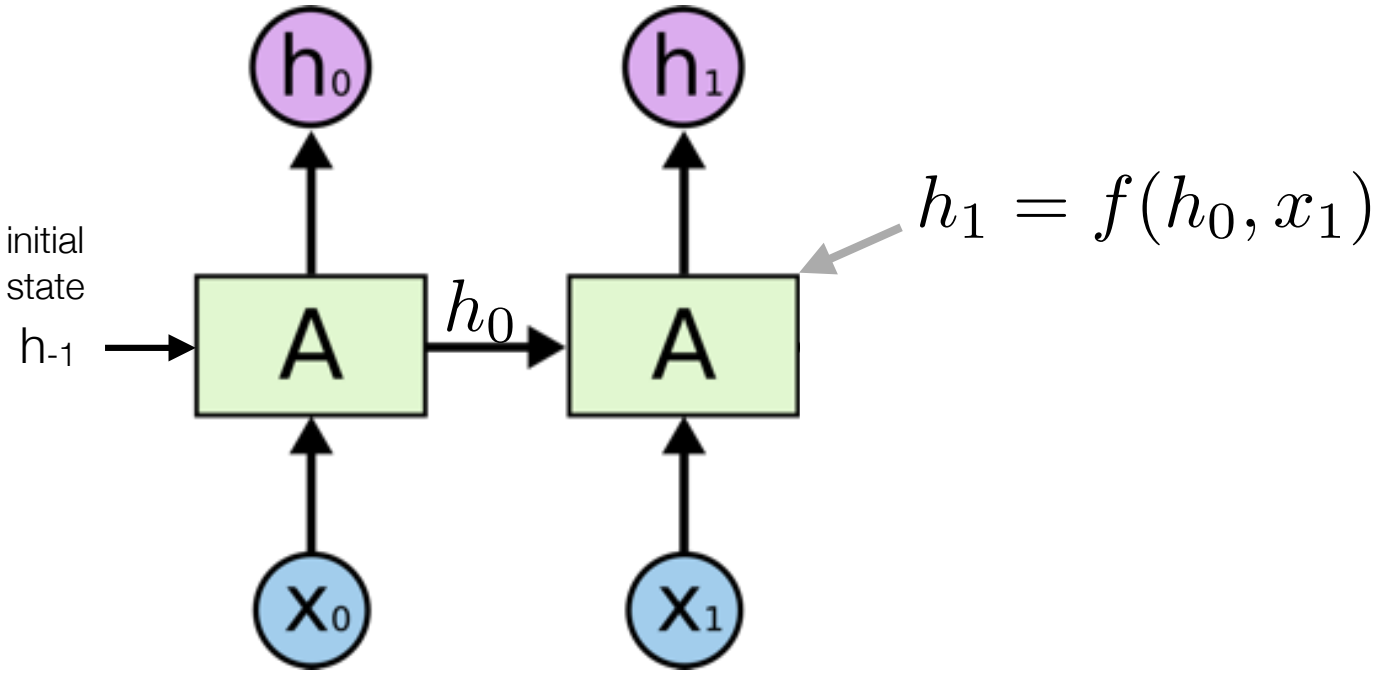
Recurrent Cells



$$h_0 = f(h_{-1}, x_0)$$

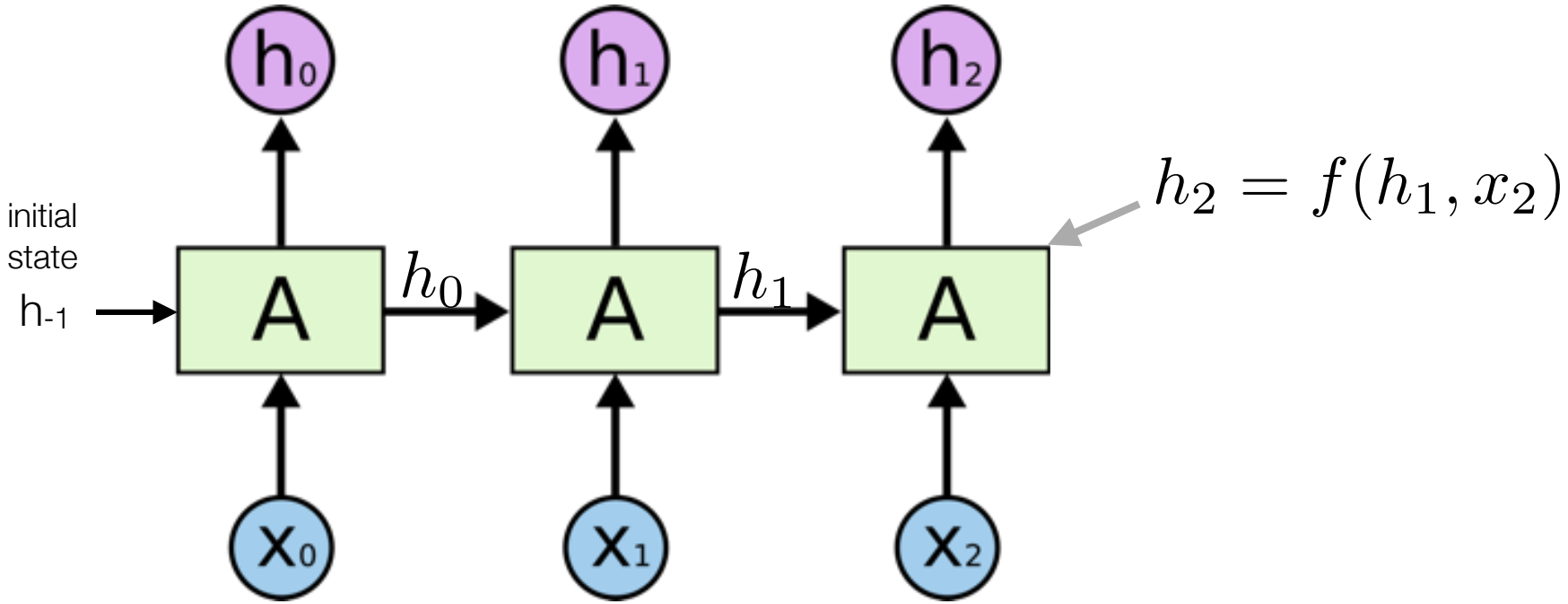
Figures taken from http://colah.github.io/posts/2015-08-Understanding-LSTMs

Recurrent Cells



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

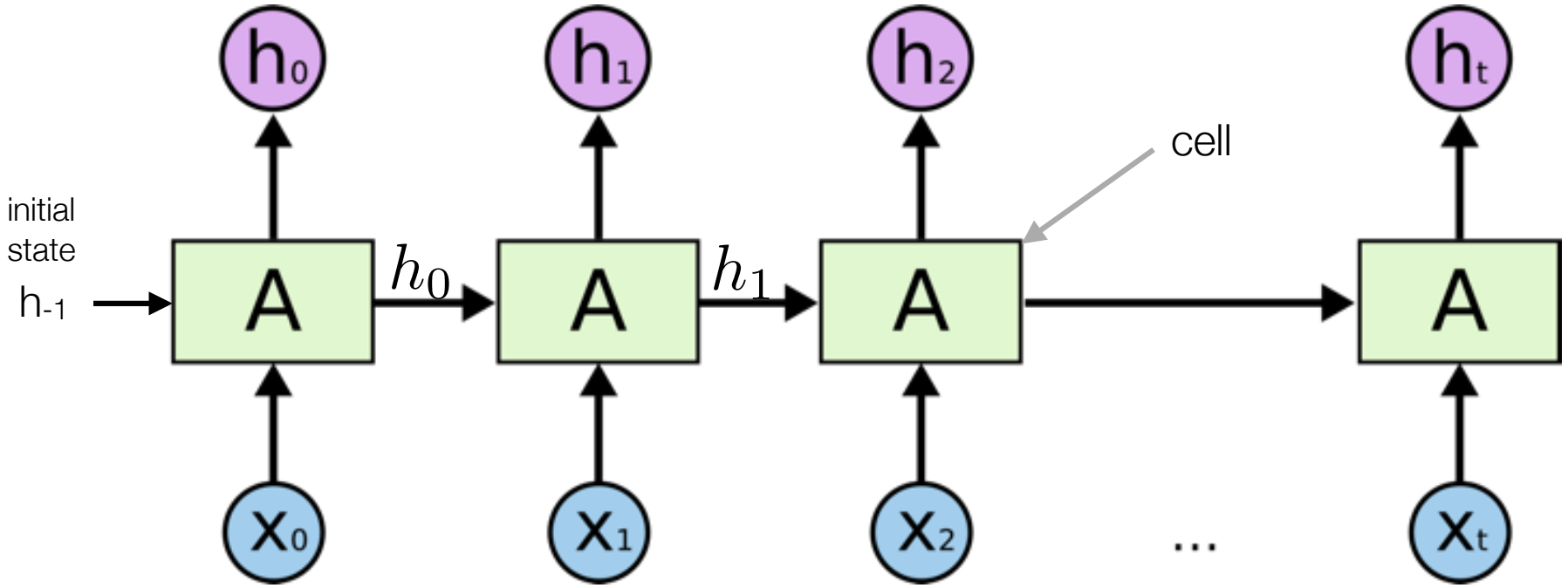
Recurrent Cells



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

RNN: Computational Graph

computational graph un-rolled



$$h_t = f(h_{t-1}, x_t)$$

State updated over a sequence of steps, consuming elements of an input sequence.

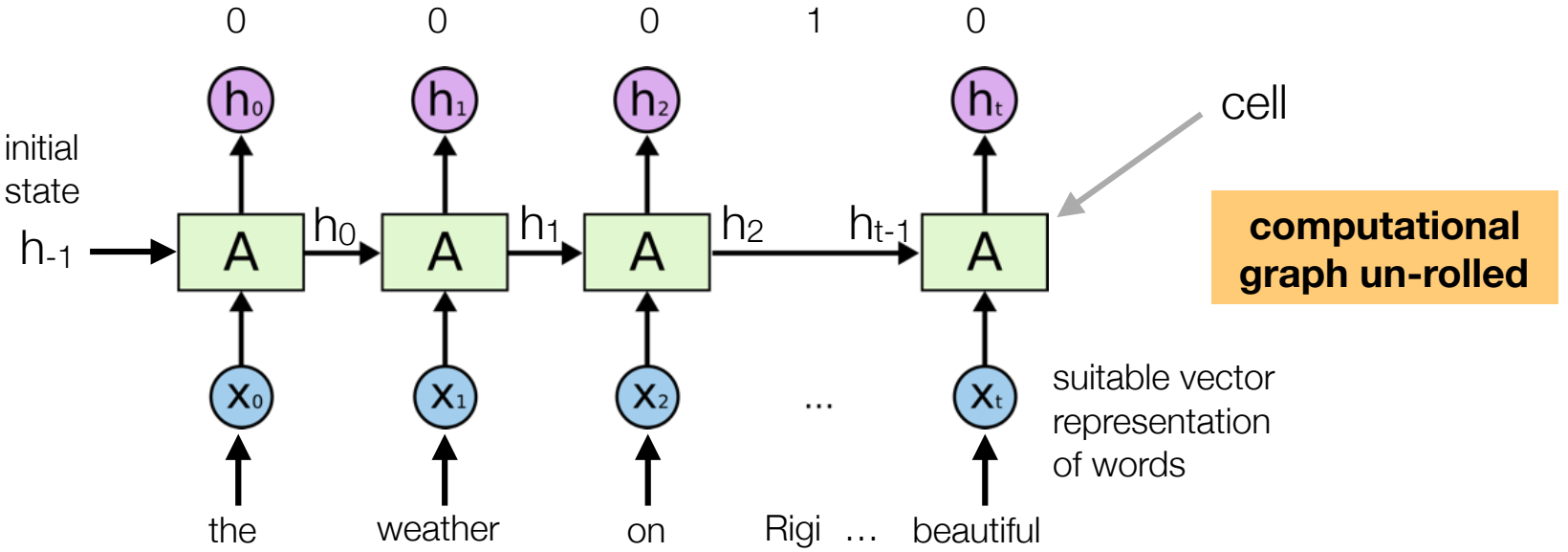
Figures taken from http://colah.github.io/posts/2015-08-Understanding-LSTMs

Recurrent Cells

In this example:

$T_x = T_y$

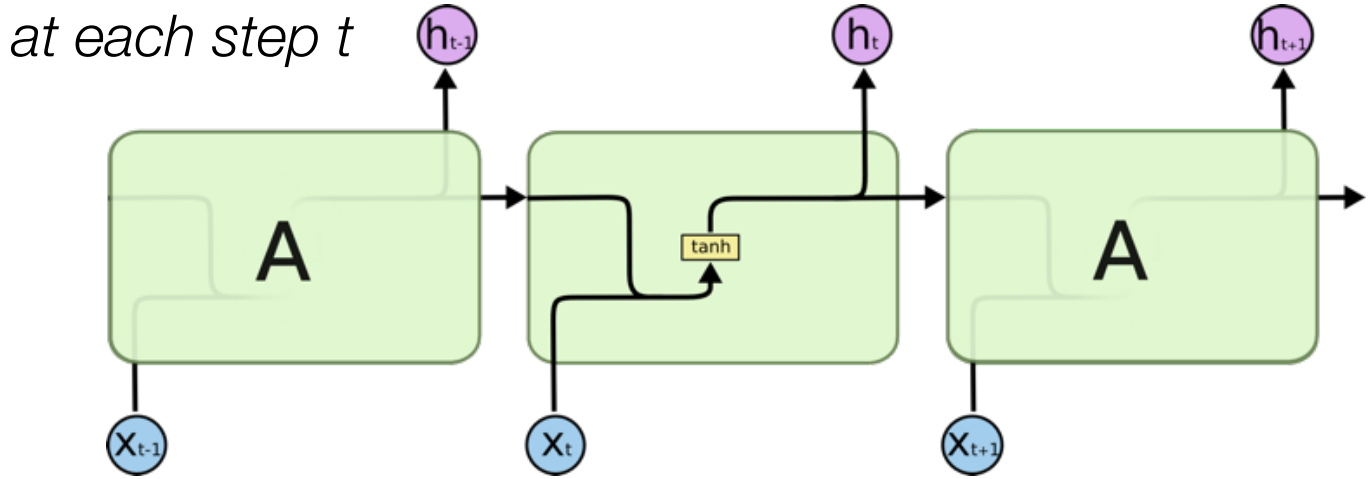
(like for NER)



- **State** h is updated through a succession of steps. The state **memorises** (part of) the "history".
- At update *t*, it consumes the next element of the input sequence (if available) and the previous state.
- State is passed on to
  - the next cell associated with step t+1 or
  - possibly to another layer (e.g. output layer)
- Before the first step, the state is suitably initialised: typically with zero values.
- Succession of cells forms a layer.
- All the cells of a layer share the parameters.

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

Computation for *Simple* RNN-Cells



Computation for forward propagation:

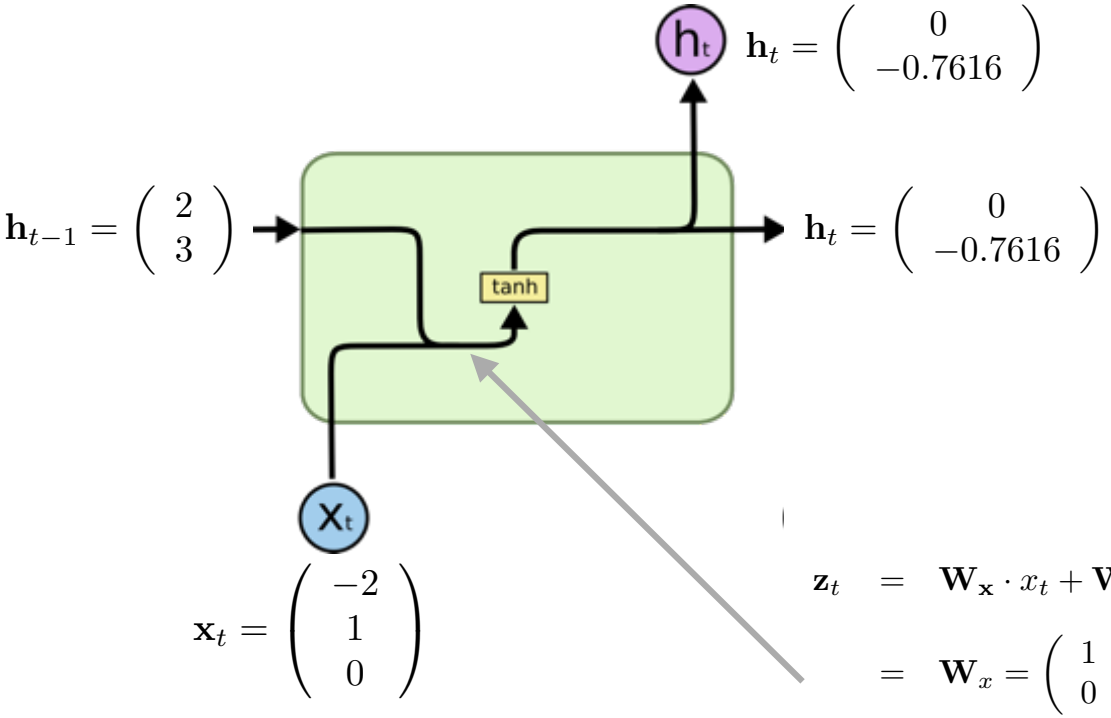
- affine transformation of the inputs (as seen for MLPs)
- non-linear activation function, e.g. *tanh*

$$h_t = g(\mathbf{W_x} \cdot x_t + \mathbf{W_h} \cdot h_{t-1} + b_h)$$

$n_h \times n_x$ $\quad$ $n_h \times n_h$

Parameters **W**'s and **b** are the same for all time step

Example Computation for Single Step



$$\mathbf{W}_x = \begin{pmatrix} 1 & -1 & -1 \\ 0 & 2 & 1 \end{pmatrix}$$

$$\mathbf{W}_h = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

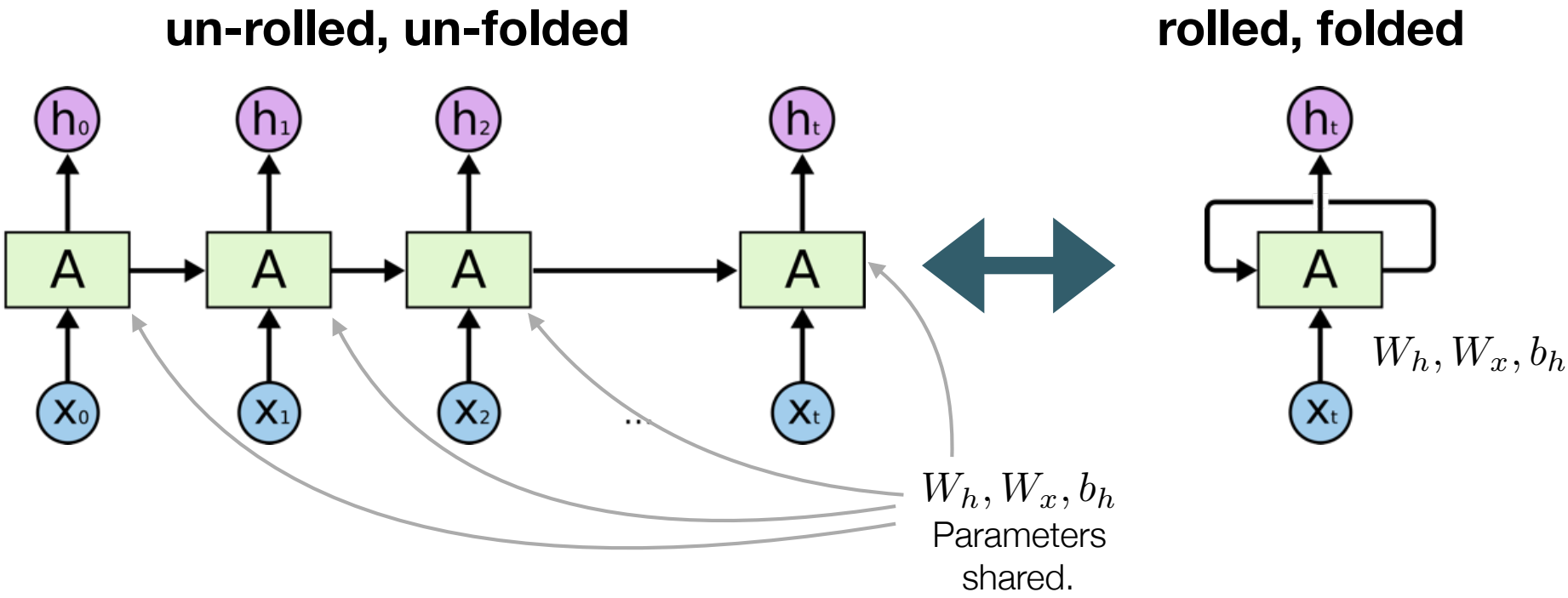
$$\mathbf{b}_h = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Parameters to be trained independent of *t* (same of all cells in the layer)

$$\begin{aligned} \mathbf{z}_t &= \mathbf{W_x} \cdot x_t + \mathbf{W_h} \cdot h_{t-1} + b_h \\ &= \mathbf{W}_x = \begin{pmatrix} 1 & -1 & -1 \\ 0 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} -2 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ -1 \end{pmatrix} \\ \mathbf{h}_t &= \tanh\left(\begin{pmatrix} 0 \\ -1 \end{pmatrix}\right) = \begin{pmatrix} 0 \\ -0.7616 \end{pmatrix} \end{aligned}$$

In typical NLP applications, word embeddings are taken as inputs, i.e. vectors of 100's of dimensions. Similarly, the hidden states vectors have a dimensionality that go into the 100's.

Two Equivalent Representations

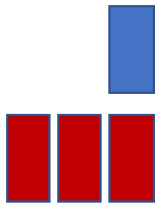


Represents one layer of a recurrent network, as such is expressed as single object in `keras` (similarly in tensorflow, pytorch, etc):

```
from keras.layers import SimpleRNN
model = Sequential()
model.add(SimpleRNN(units=…, input_shape=…, …))
```

Example: Name Classification

	name	is_male
0	Marie	False
1	Kelley	False
2	Frederik	True
3	Martino	True
4	Dian	False
5	Kristal	False
6	Elihu	True
7	Bobbie	True
8	Zackariah	True
9	Hayes	True
10	Nisa	False
11	Ansell	True
12	Bee	False
13	Ainslee	False

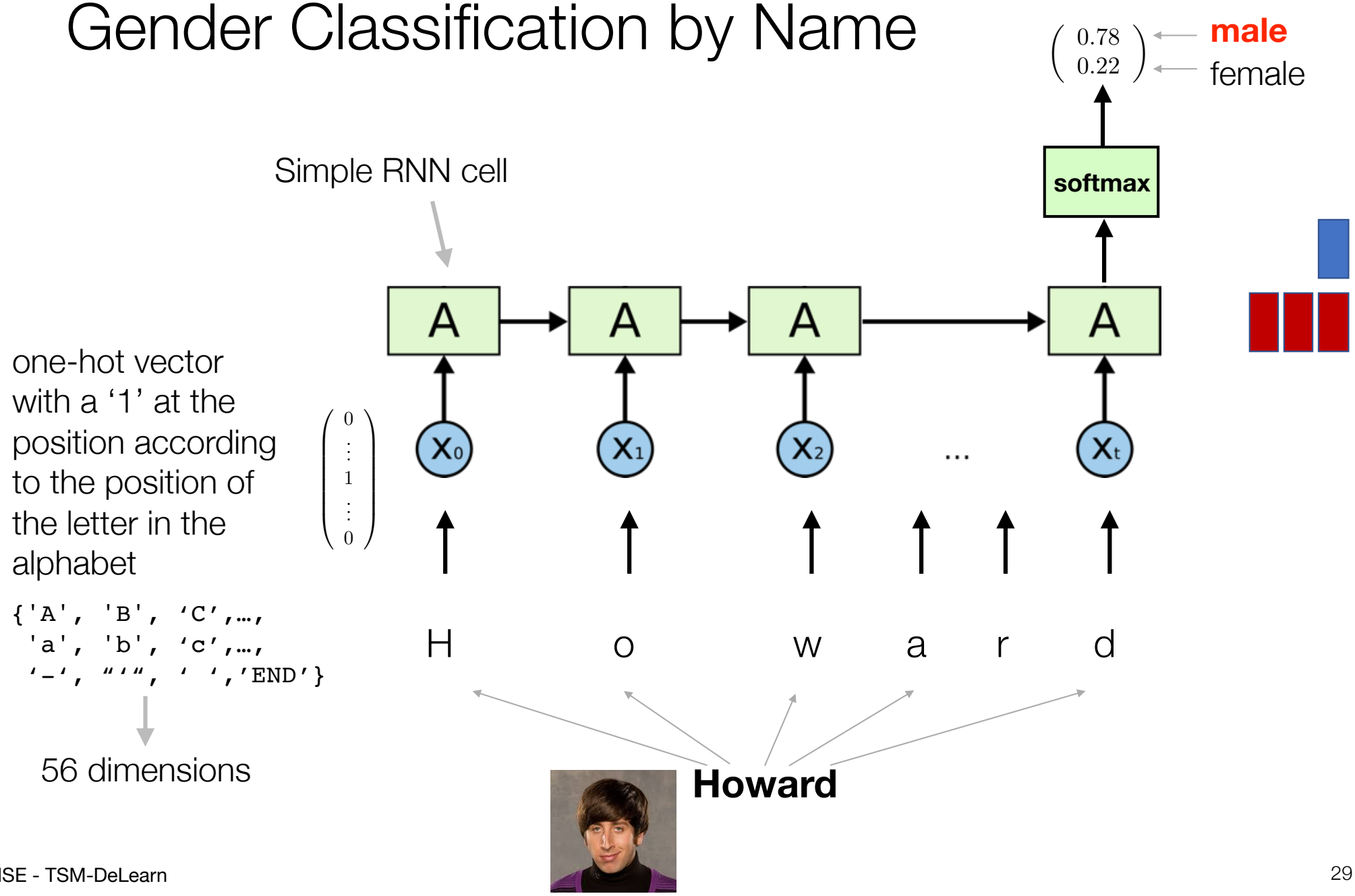


- Classification Task:
  - Predict gender from first name.
  - Predict home country from first name.
- Model:
  - Input: Name treated as sequence of characters.
  - Output: Class as a single element (sequence of length 1)
- Numeric representation of the input/output data:
  - Input: One-hot representation of the characters in the alphabet. *Character-based model.*
  - Output: One hot representation of classes (labels), vector of scores for the individual classes (model output).

Is gender or home country encoded in the structure of the words?

See https://github.com/prdeepakbabu/Python/tree/master/Deep%20learning%20gender
Data: http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/nlp/corpora/names/

Gender Classification by Name



Define the Model in `Keras`

We again use a **`Sequential`** model since we will define it as a simple succession of layers.

Simplest form of an RNN layer:

- **`units`:** specifies the dimension of the hidden state variable.
- **`return_sequences=False`:** that no sequence is returned, i.e. just the hidden state at the last step.
- **`input_shape`:** length of input sequence (`maxlen`) and dimension of an element of the input sequence (`len_vocab`)

We add a softmax layer for the binary classification.

```
model = Sequential()
model.add(SimpleRNN(units=64, return_sequences=False, \
                    input_shape=(maxlen,len_vocab)))
model.add(Dense(2, activation='softmax'))

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_3 (SimpleRNN)     (None, 64)                7744
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 130
=================================================================
Total params: 7,874
Trainable params: 7,874
Non-trainable params: 0
_________________________________________________________________
```

$$\begin{aligned} \mathbf{W}_h: &\quad 64 \times 64 &&= 4096 \\ \mathbf{W}_x: &\quad 64 \times 56 &&= 3584 \\ \mathbf{b}_h: &\quad 64 \times 1 &&= 64 \end{aligned} \Big\} \; 7744$$

`len_vocab=56`

An according MLP would have 56'834 parameters

$$2 \times 64 + 2 = 130$$

Transform Input Data

Prepare the alphabet (in NLP terms the "vocabulary") that is used for generating the one-hot representation.
We add some special characters that can occur in names ("-","'",""").
Furthermore, "END" is added for marking the END of a name. This is also used for padding the sequence so that sequences of equal length can be constructed.

```
import string
alphabet = set(string.ascii_letters+" -'")
alphabet.add('END')
len_alphabet = len(alphabet)
char_index = dict((c, i) for i, c in enumerate(alphabet))
print("Size of alphabet: ",len_alphabet)
print(alphabet)
```

```
Size of alphabet:  56
{'x', 'S', 'N', 'h', 'U', 'a', 'n', 'R', 'c', 'k', 'F', 'Y', ' ', "'
", 'A', 'L', 'K', 'V', 'P', 'r', 'v', 'J', 'Z', 'END', 'd', 'f', 't'
, 'u', '-', 'w', 'y', 'X', 'C', 'e', 'E', 'H', 'Q', 'z', 'I', 'g', '
T', 'M', 'b', 'i', 'G', 'm', 'j', 's', 'B', 'l', 'O', 'q', 'p', 'o',
'W', 'D'}
```

Here, the representation for the names as a sequence of one-hot vectors is constructed.
Note that the `maxlen` parameter is set to make all the sequences equal in length - if necessary by truncating the sequences or padding them with 'END'.

```
def onehot(i, length):
    v = np.zeros(length);
    v[i] = 1
    return v

def name_representation(name, maxlen):
    name_trunc = str(name)[0:maxlen]
    size = len(char_index)
    vector = [onehot(char_index[j], size) for j in str(name)]
    # fill the rest with
    for k in range(0,maxlen - len(str(name))):
        vector.append(onehot(char_index['END'], size))
    return vector

def gender_representation(ismale):
    return np.array([1,0]) if ismale else np.array([0,1])
```

No need to specify the length of the sequence statically in frameworks with dynamic computational graph (e.g. in pytorch) - but training is much more efficient if specified statically,

Load Data

```
def load_from_file(path):
    names = pd.read_csv(path, header=None)
    names.columns=['name','is_male']
    return names

path = "names_by_gender.csv"
names = load_from_file(path)
names.head(5)
```

	name	is_male
0	Tomas	True
1	Armstrong	True
2	Dino	True
3	Sybyl	False
4	Milissent	False

```
maxlen = np.max([len(name) for name in names.name])
print("Maximum name length: ", maxlen)
names.groupby('is_male')['name'].count()
```

```
Maximum name length:  15

is_male
False    5001
True     2943
Name: name, dtype: int64
```

Roughly 8000 samples.
Imbalance towards
female names (5/8 : 3/8)

Shuffle the dataset.

Split into a training and test set:
- 80% training data
- 20% test data

```
msk = np.random.rand(len(names)) < 0.8
train = names[msk]
test = names[~msk]
```

```
X_train = []
Y_train = []
for name in train.name:
    X_train.append(name_representation(name,maxlen))
for ismale in train.is_male:
    Y_train.append(gender_representation(ismale))

X_train = np.asarray(X_train)
Y_train = np.asarray(Y_train)
print(X_train.shape,Y_train.shape)
```

```
(6371, 15, 56) (6371, 2)
```

… and similarly for the test set …

Train Model

The well known `fit` method is used for training the model.
Here, we train for 30 epochs with a mini-batch size of 200 and see that probably 15 epochs would have been sufficient.

With this simple model we obtain roughly 80% accuracy.

Quite a gap between train and test accuracy:
More data would be nice — but here, we cannot expect to get more.
Data augmentation is not possible either.
Would a richer model + regularisation help?

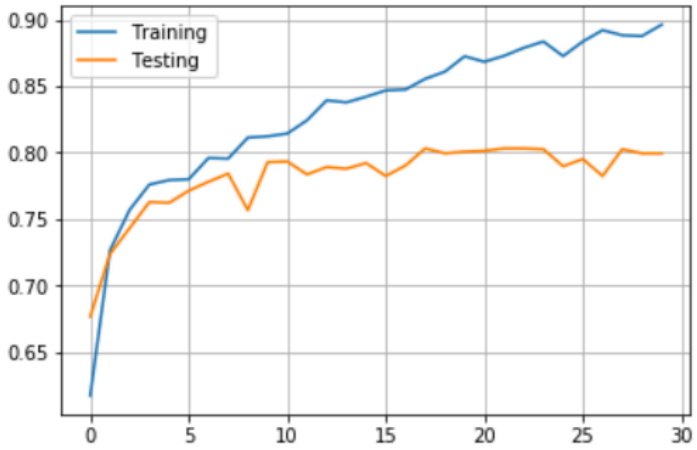
```
batch_size=200
log = model.fit(X_train, Y_train, batch_size=batch_size, \
                epochs=30, validation_data=(X_test, Y_test))
```

```
Train on 6371 samples, validate on 1573 samples
Epoch 1/30
6371/6371 [==============================] - 1s 84us/step - loss
0.6929
Epoch 2/30
6371/6371 [==============================] - 0s 36us/step - loss
0.7254
Epoch 3/30
```

```
score, acc = model.evaluate(X_test, Y_test)
print('Test score:', score)
print('Test accuracy:', acc)
```

```
1635/1635 [==============================] - 0s 61us/step
Test score: 0.4873464875082722
Test accuracy: 0.7993883792048929
```

```
plt.plot(log.history['acc'], label='Training')
plt.plot(log.history['val_acc'], label='Testing')
plt.legend()
plt.grid()
```



Results (for this class)

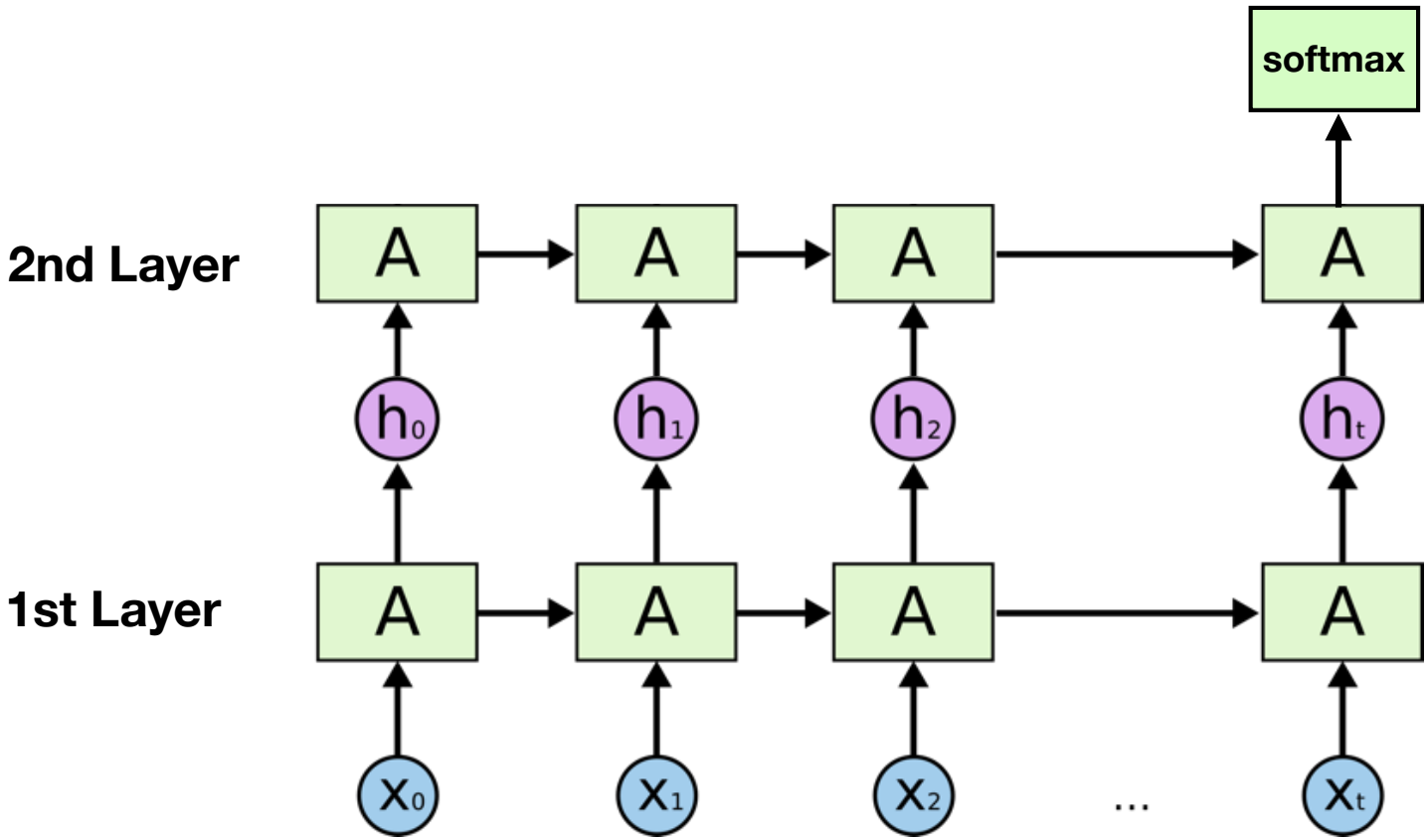
name	is_male	pred	name	is_male	pred	name	is_male	pred
Adrian	TRUE	FALSE	Gregory	TRUE	TRUE	Reto	TRUE	TRUE
Alexander	TRUE	TRUE	Jean	TRUE	TRUE	Roger	TRUE	TRUE
Anastasia	FALSE	FALSE	Joel	TRUE	FALSE	Samuel	TRUE	TRUE
Andreas	TRUE	FALSE	Jonas	TRUE	TRUE	Sandro	TRUE	TRUE
Armin	TRUE	TRUE	Juan	TRUE	TRUE	Severin	TRUE	TRUE
Benjamin	TRUE	TRUE	Julien	TRUE	TRUE	Siro	TRUE	TRUE
Bjoern	TRUE	TRUE	Linda	FALSE	FALSE	Stefan	TRUE	TRUE
Cedric	TRUE	TRUE	Lorenz	TRUE	FALSE	Stefano	TRUE	TRUE
Christian	TRUE	FALSE	Lukas	TRUE	TRUE	Steven	TRUE	TRUE
Christophe	TRUE	TRUE	Manuel	TRUE	TRUE	Thomas	TRUE	TRUE
Damian	TRUE	FALSE	Marc	TRUE	TRUE	Timan	TRUE	FALSE
Daniel	TRUE	FALSE	Marcel	TRUE	TRUE	Tobias	TRUE	TRUE
Daniele	TRUE	FALSE	Markus	TRUE	TRUE	Tobias	TRUE	TRUE
Dario	TRUE	FALSE	Martin	TRUE	TRUE	Yanick	TRUE	TRUE
David	TRUE	TRUE	Marvin	TRUE	TRUE	Yannick	TRUE	TRUE
Dominic	TRUE	TRUE	Matthias	TRUE	FALSE	Ying	FALSE	TRUE
Dominik	TRUE	FALSE	Michael	TRUE	TRUE			
Etienne	TRUE	FALSE	Parijat	FALSE	FALSE			
Florian	TRUE	FALSE	Pascal	TRUE	TRUE			
Glenn	TRUE	FALSE	Patrick	TRUE	TRUE			

52 male
4 female

16 names wrongly classified in this class - too many boys classified as girls.
Accuracy ~72% which is 8% smaller than obtained on the test set.

Is this acceptable? Or why could this happen?

Stacking RNNs



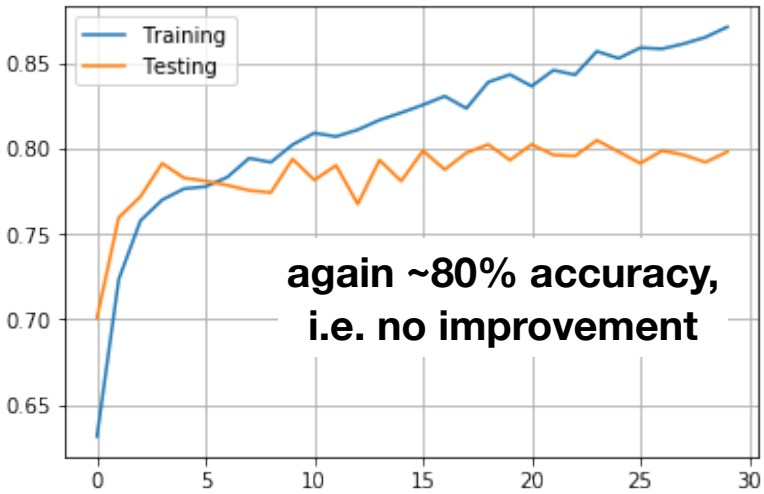
Stacking RNNs

First RNN layer with dropout.
Note that
return_sequences=True
must to set so that a sequence is past to the next layer.

Second RNN layer again with dropout.

Finally, again the softmax layer for the binary classification.

```
model = Sequential()
model.add(SimpleRNN(units=64, return_sequences=True, \
                    input_shape=(maxlen,len_alphabet)))
model.add(Dropout(0.2))
model.add(SimpleRNN(units=64, return_sequences=False, \
                    input_shape=(maxlen,len_alphabet)))
model.add(Dropout(0.2))
model.add(Dense(2, activation='softmax'))

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

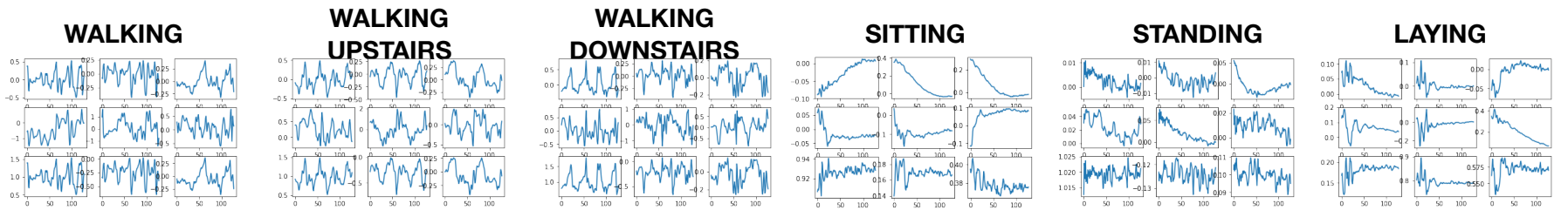



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_19 (SimpleRNN)    (None, 15, 64)            7744
_________________________________________________________________
dropout_8 (Dropout)          (None, 15, 64)            0
_________________________________________________________________
simple_rnn_20 (SimpleRNN)    (None, 64)                8256
_________________________________________________________________
dropout_9 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_16 (Dense)             (None, 2)                 130
=================================================================
Total params: 16,130
Trainable params: 16,130
Non-trainable params: 0
_________________________________________________________________
```

Human Activity Recognition

- Recognise activity based on motion activity recorded by smartphones:
  - Six different classes: WALKING, WALKING_UPSTAIRS, WALKING_DOWNSTAIRS, SITTING, STANDING, LAYING.

- Smartphones Data Set from the "UC Irvine Machine Learning Repository" (see http://archive.ics.uci.edu/ml/datasets/Human+Activity+Recognition+Using+Smartphones#)
  - Sensor signals from accelerometer and gyroscope, including gravitational and body motion components suitably pre-processed to time series with 128 steps and 9 values per step.
  - 7352 training and 2947 test samples.



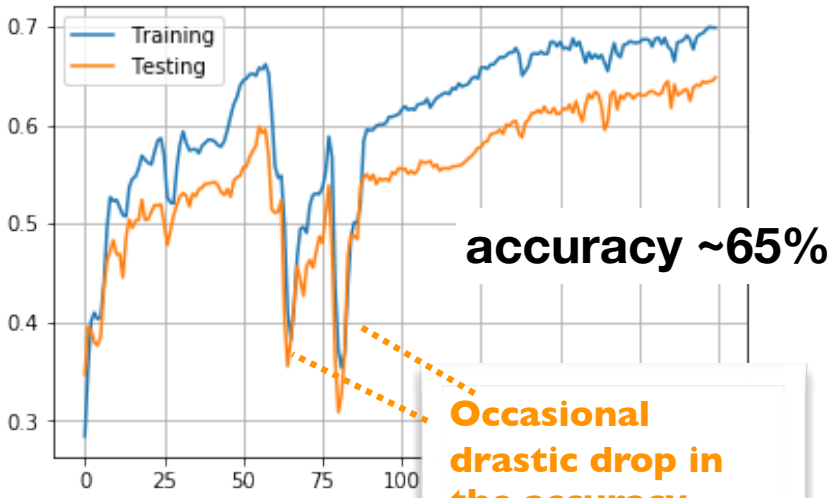
https://youtu.be/XOEN9W05_4A

- Traditional ML approach:
  - A lot of feature engineering required — signal processing approach combined with classical data science techniques.

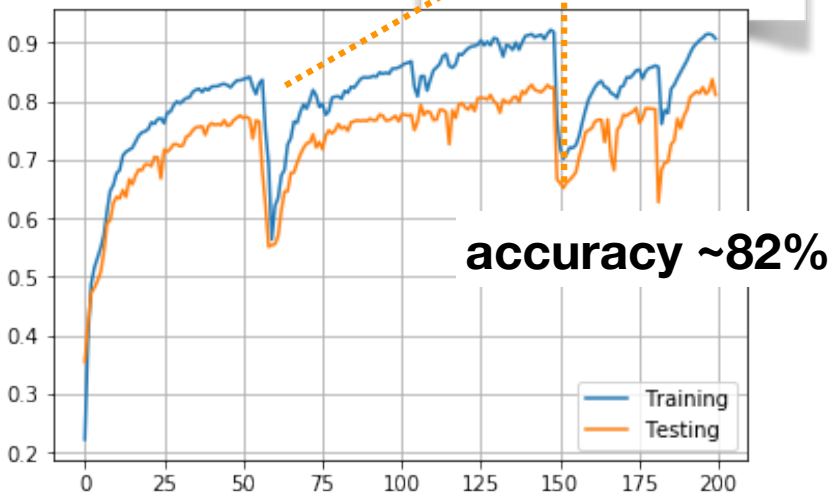
- RNN approach:
  - Only very little manual feature engineering needed (see Guillaume-Chevalier: https://github.com/guillaume-chevalier/LSTM-Human-Activity-Recognition)

Human Activity Recognition

- One layer with simple RNN
- 32 hidden units
- L2-Regularisation (lambda=0.005) (this regularisation also chosen by Guillaume-Chevalier)
- Batch size 1500, 100 epochs

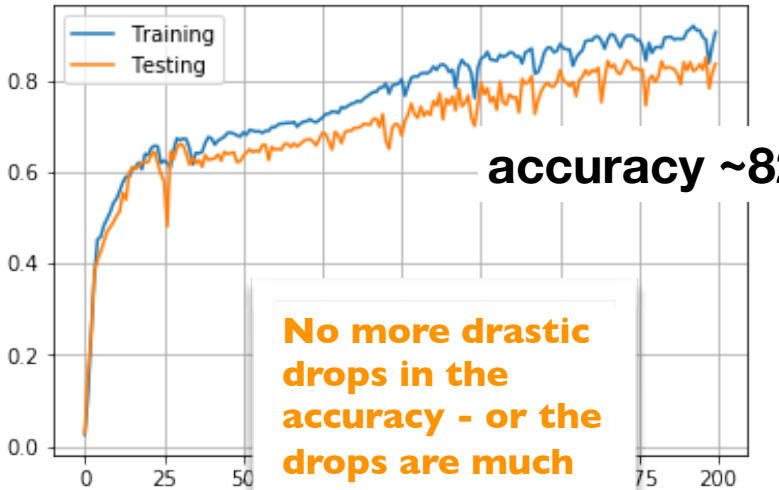


- Two stacked layers with Simple RNN
- 32 hidden units each
- L2-Regularisation (lambda=0.005) (this regularisation also chosen by Guillaume-Chevalier)
- Batch size 1500, 100 epochs



Results including Gradient Clipping

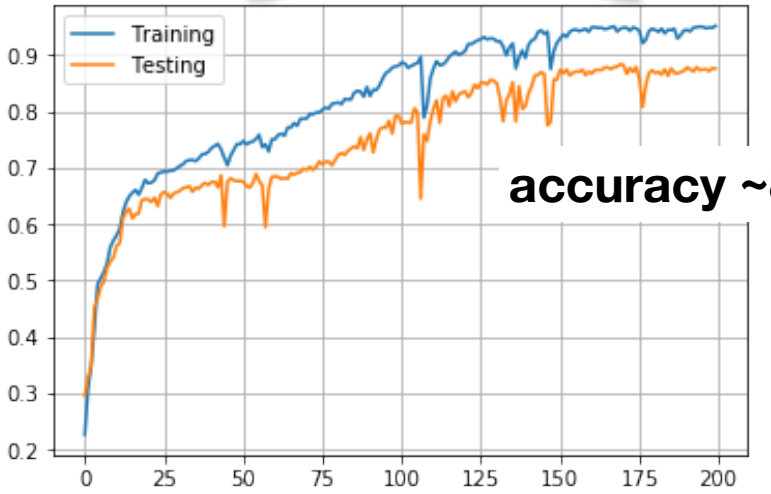
- One layer with simple RNN
- 32 hidden units
- L2-Regularisation (lambda=0.005)
- Batch size 1500, 100 epochs
- Clipping gradient by value (each component of the gradient restricted to in [-0.5,05]



accuracy ~82%

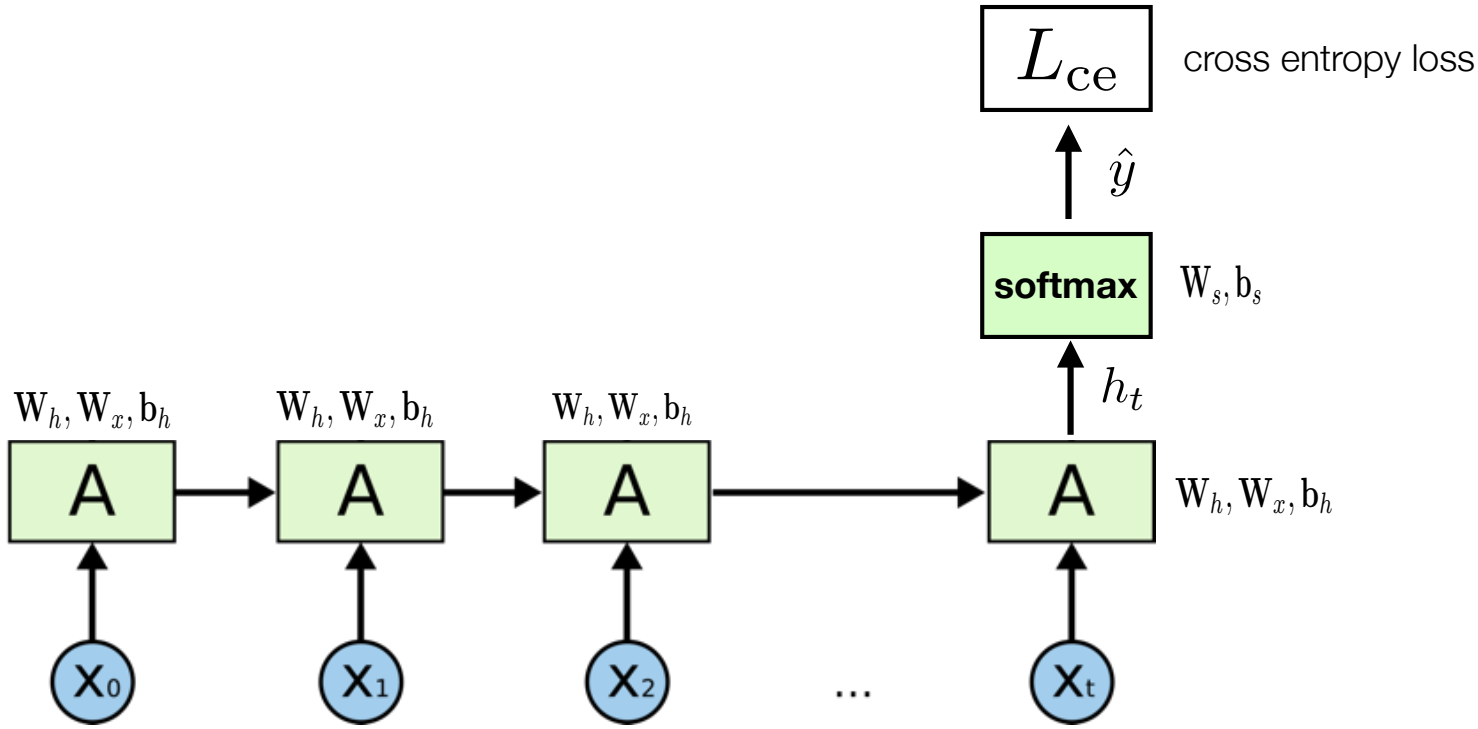
No more drastic drops in the accuracy - or the drops are much reduced.

- Two stacked layers with Simple RNN
- 32 hidden units each
- L2-Regularisation (lambda=0.005)
- Batch size 1500, 100 epochs
- Clipping gradient by value (each component of the gradient restricted to in [-0.5,05]



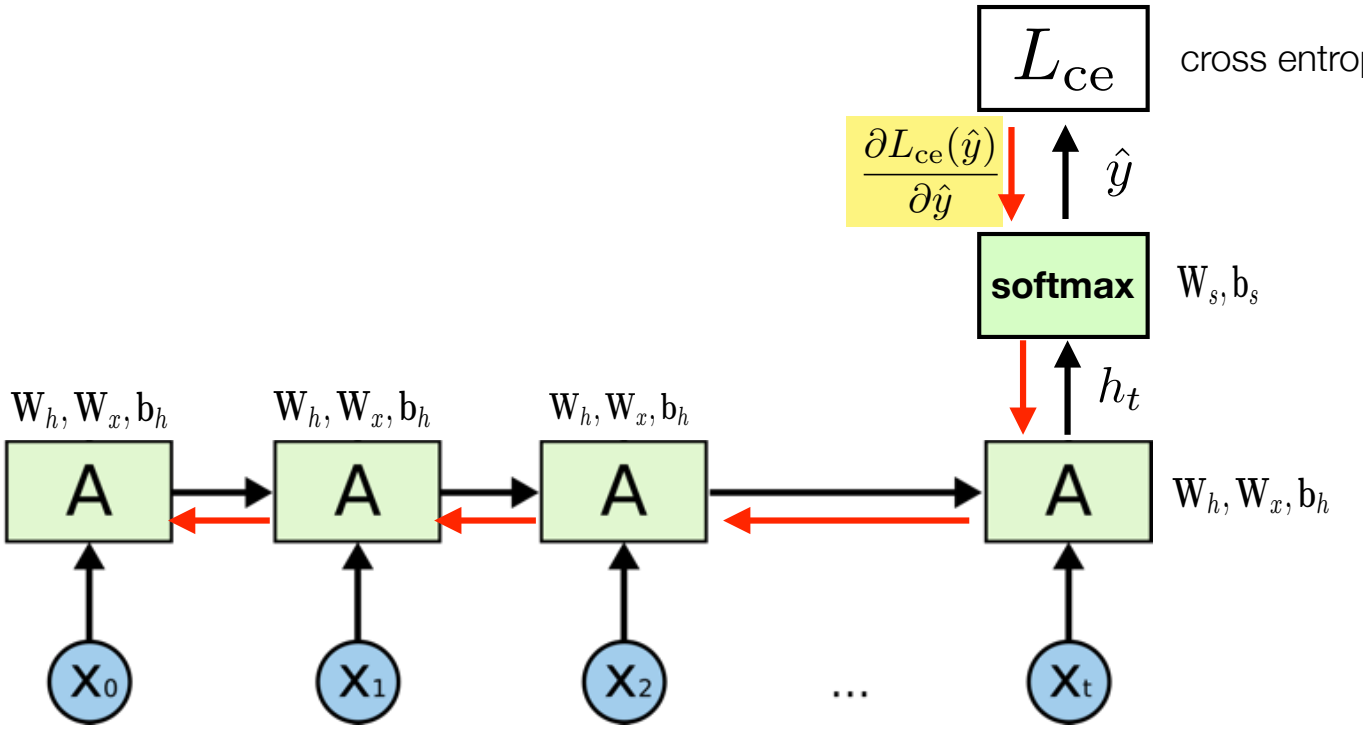
accuracy ~88%

Backprop with a Single-Layer Simple RNN



A $h_t = g(z_t), z_t = \mathbf{W_x} \cdot x_t + \mathbf{W_h} \cdot h_{t-1} + b_h$

Backprop with a Single-Layer Simple RNN

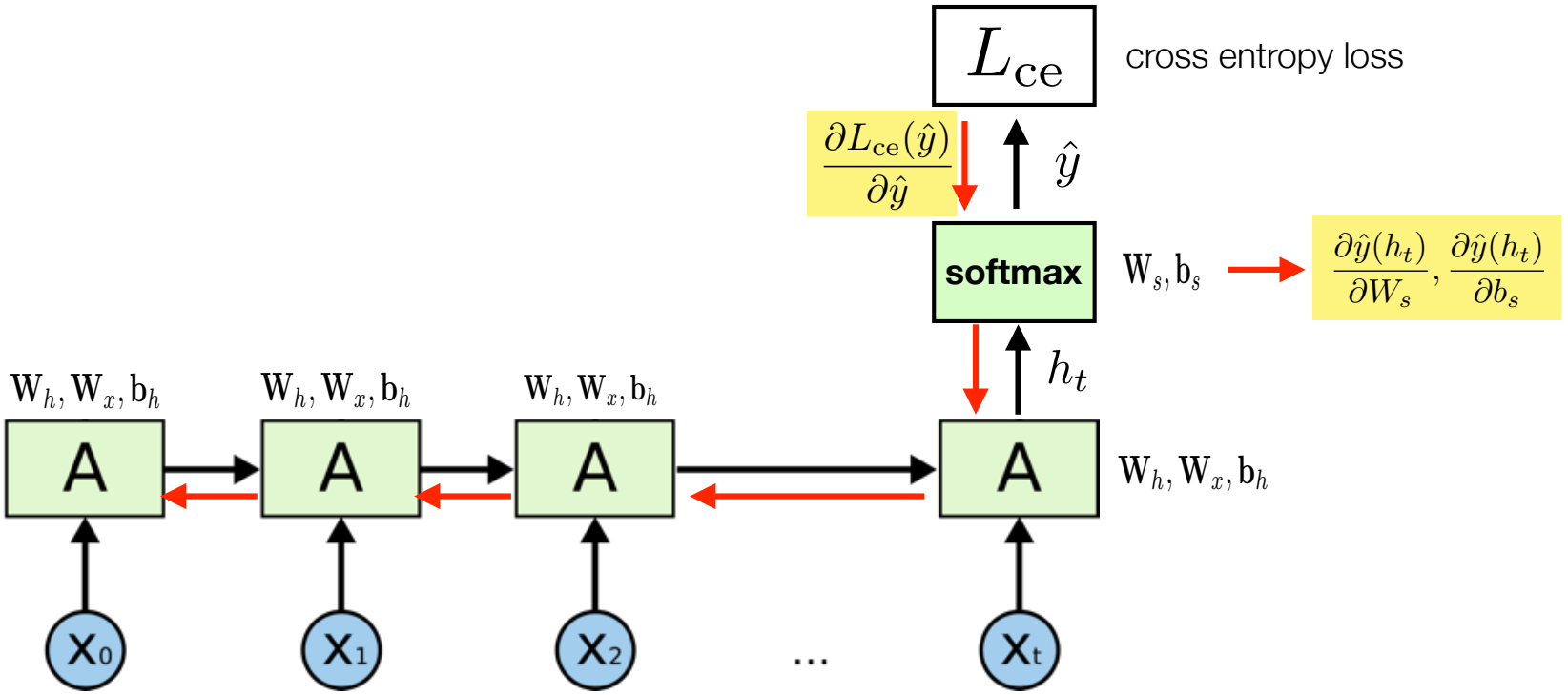


A $h_t = g(z_t), z_t = \mathbf{W_x} \cdot x_t + \mathbf{W_h} \cdot h_{t-1} + b_h$

Backpropagation

Partial derivates — are vectors or matrices; straightforward to compute: e.g. $\frac{\partial h_s}{\partial h_{s-1}} = W_h^T \cdot g'(z_s)$

Backprop with a Single-Layer Simple RNN

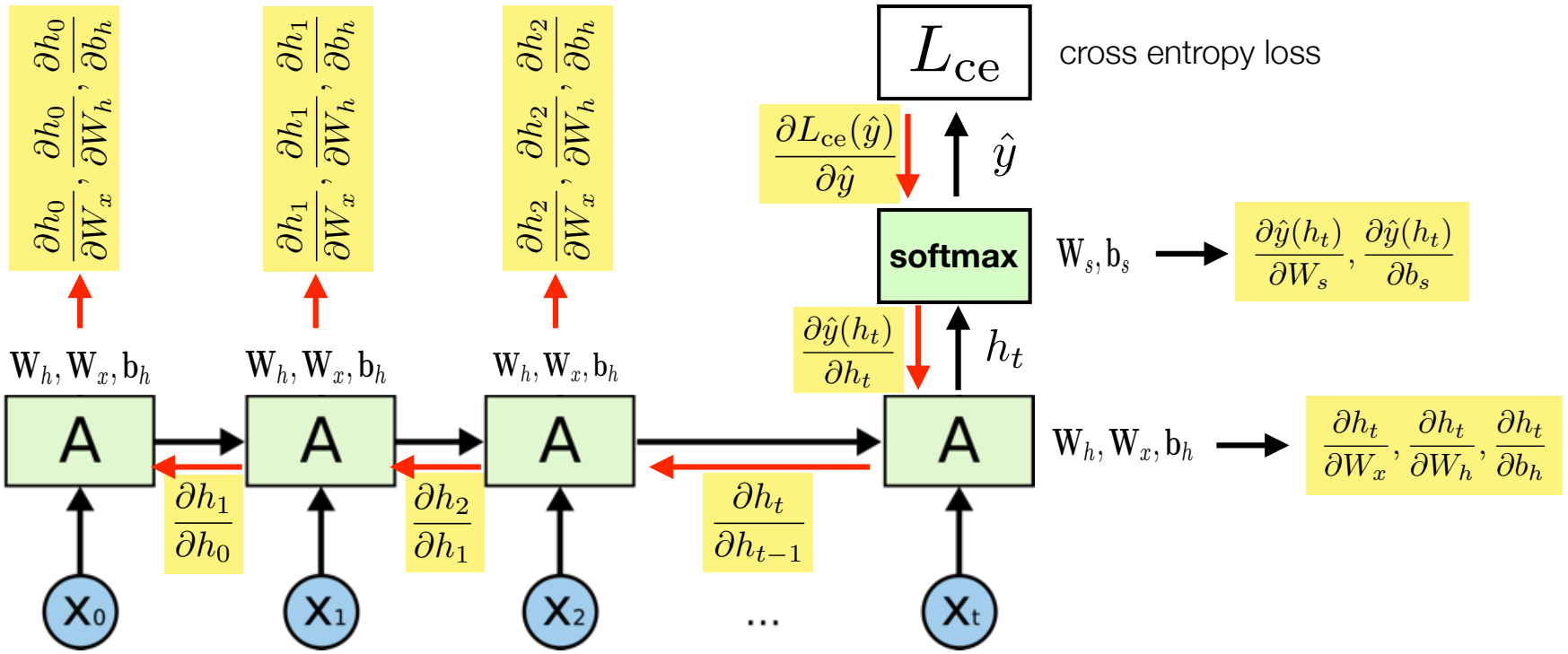


A $h_t = g(z_t), z_t = \mathbf{W_x} \cdot x_t + \mathbf{W_h} \cdot h_{t-1} + b_h$

Backpropagation

Partial derivates — are vectors or matrices; straightforward to compute: e.g. $\frac{\partial h_s}{\partial h_{s-1}} = W_h^T \cdot g'(z_s)$

Backprop with a Single-Layer Simple RNN



A $h_t = g(z_t), z_t = \mathbf{W_x} \cdot x_t + \mathbf{W_h} \cdot h_{t-1} + b_h$

→ Backpropagation

Partial derivates — are vectors or matrices; straightforward to compute: e.g. $\frac{\partial h_s}{\partial h_{s-1}} = W_h^T \cdot g'(z_s)$

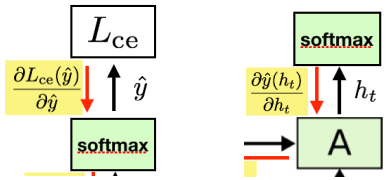
Backprop with a Single-Layer Simple RNN

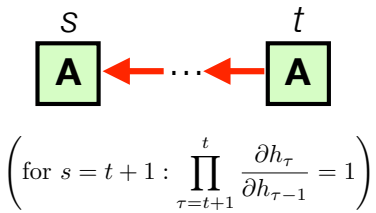
- Note that in general the partial derivative terms are tensors (i.e. the inputs *x* and the hidden state *h* have more than one dimension).
- Finally, we are interested in the partial derivatives of the loss w.r.t. to the parameters. Since the same parameters are shared across the different cells, many terms contribute.

Backprop through time

$$\frac{\partial L_{\text{ce}}}{\partial W_h} = \frac{\partial L_{\text{ce}}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_t} \sum_{s=1}^{t+1} \left(\prod_{\tau=s}^{t} \frac{\partial h_\tau}{\partial h_{\tau-1}} \right) \frac{\partial h_{s-1}}{\partial W_h}$$

needed for gradient descent





Product with many factors, each of the form

$$\mathbf{W}_h^T \cdot g'(\mathbf{z}_\tau)$$

can become arbitrarily small or large.

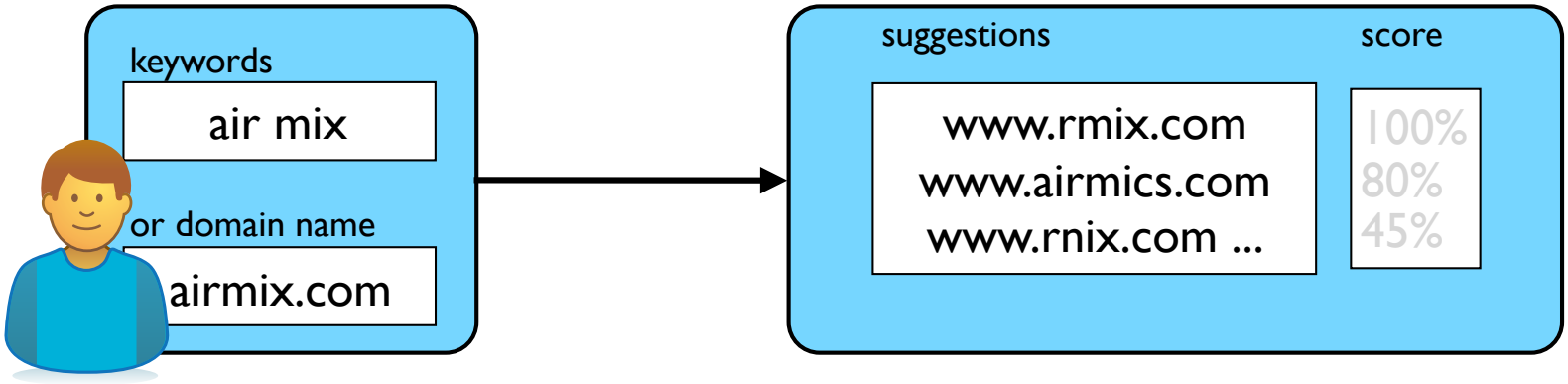
Here, we ignore the complexity introduced by dealing with tensors.

Generative RNNs

- use case

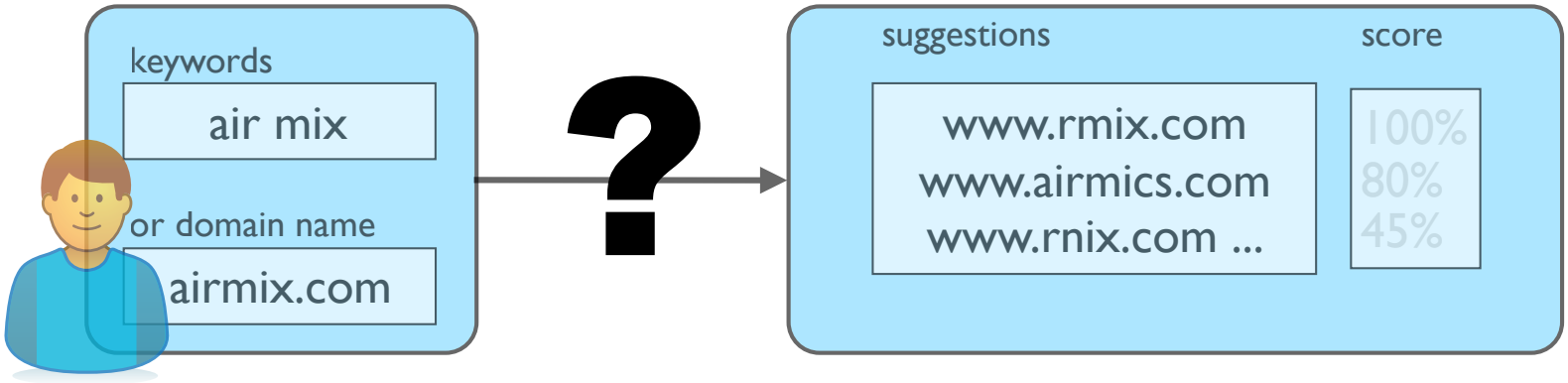
Domain name suggestion

Similar sounding domain name suggestion



- airmix.com is taken. I would like to know what are the similar sounding domain names that are available.
- I decide to book rmix.com that shows a 100% phonetic matching with what I initially wanted.

Similar sounding domain name suggestion



- Challenges:

1. Transform input into phoneme sequences

   invented words, brand names, family names, etc.

2. Find the phonetic patterns

3. Replace phonetic patterns with similar sounding ones

4. Transform back into sequence of characters

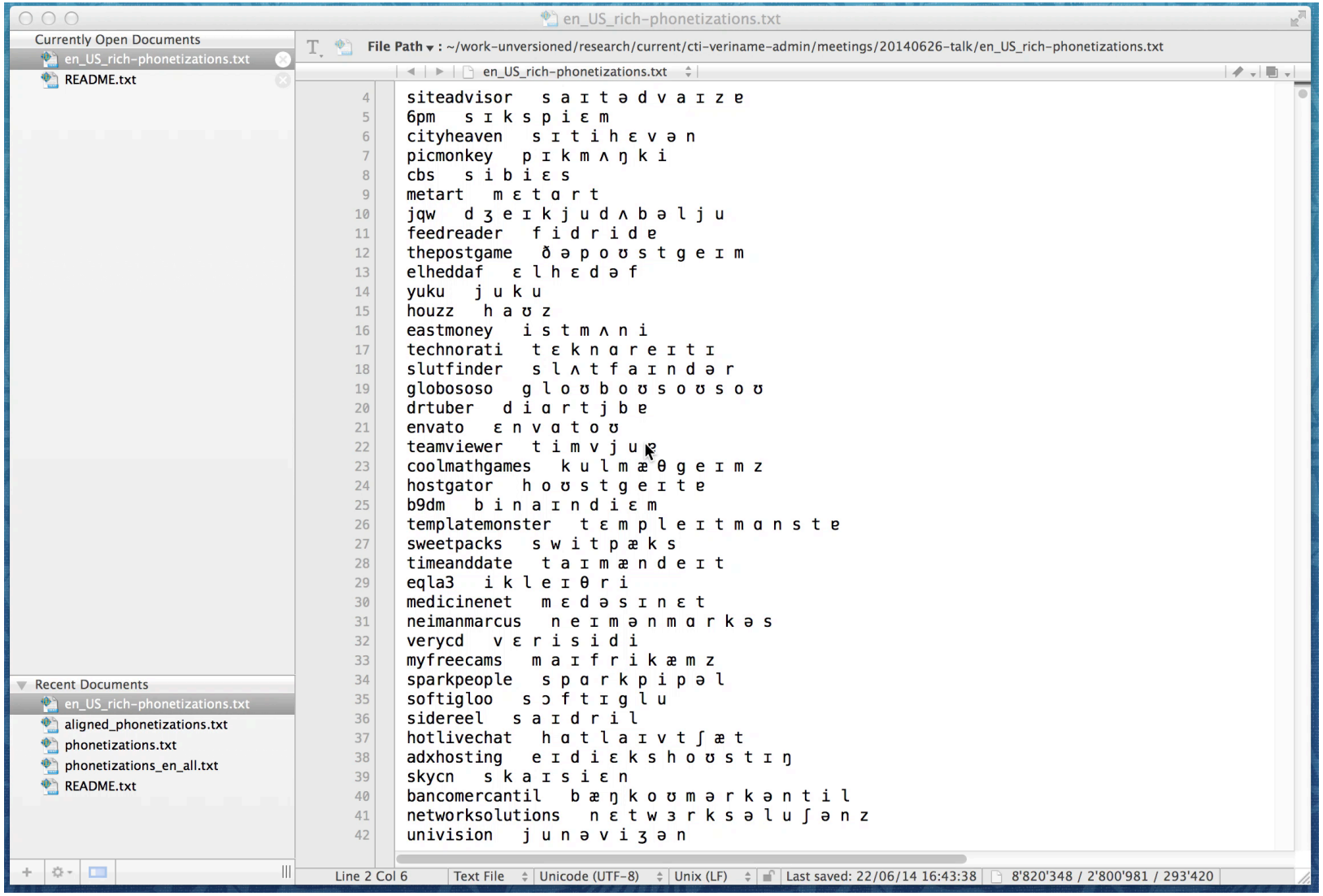
Additional constraints

We need to scale to as many language as possible !

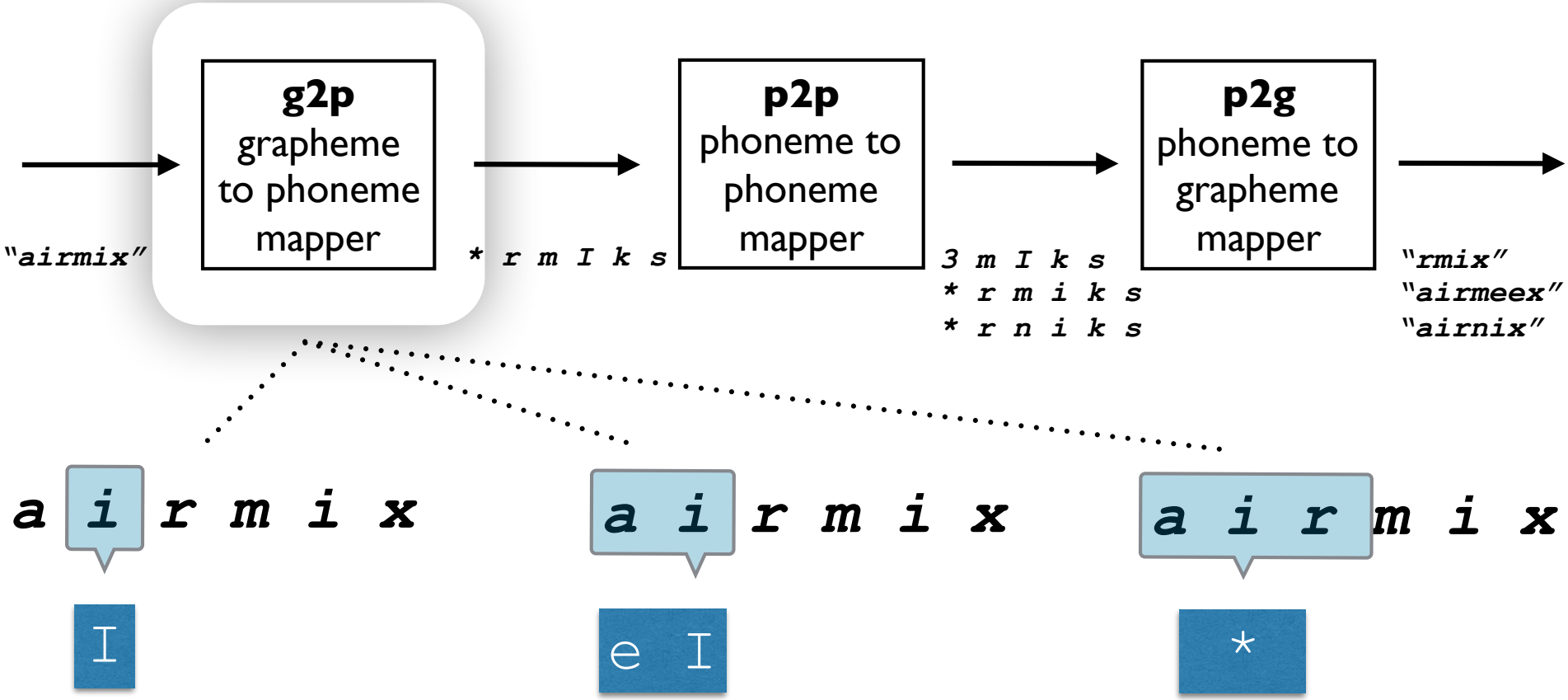
The algorithms should be efficient if we want to deploy !

What Languages Sound Like To Foreigners - http://www.youtube.com/watch?v=ybcvlxivscw

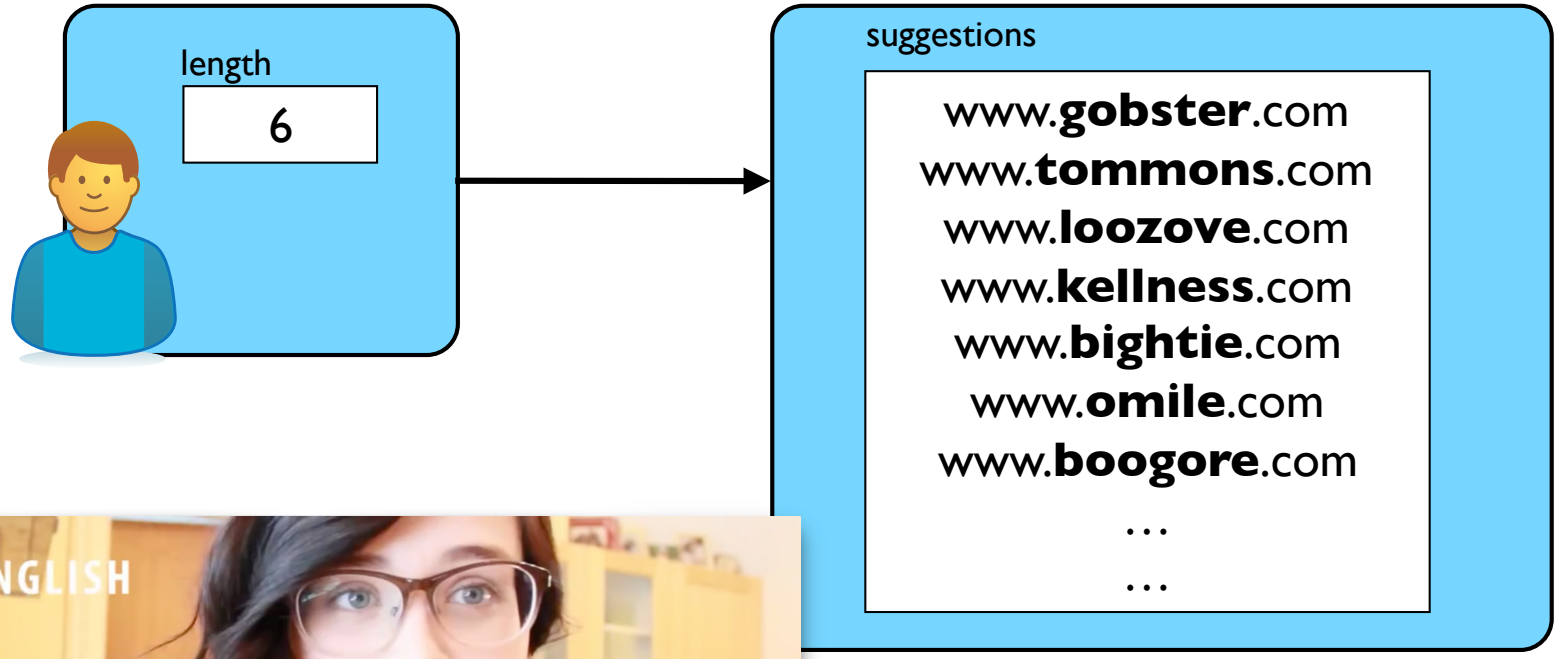
Phonetic structures



Similar sounding domain name suggestion



Random generation of (good) sounding domain names



Supported Languages

- English, French, German, Spanish, Portuguese, Italian, Russian
- In development:
  - Turkish, Korean, Japanese, Hindi, Arabic, Chinese

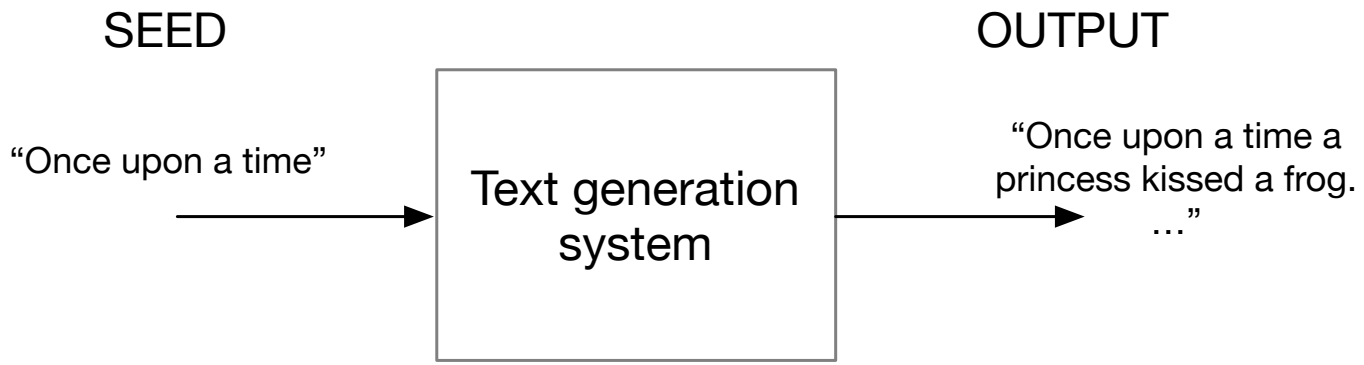
Generative RNNs

- Principles
- Two approaches
- Many to one
  - Example on text generation
- Many to many
  - Time distributed back-propagation
  - Example on text generation

Principle of data generation

A **generative system** is a system able to generate new consistent data from a **seed**. By consistent, we mean respecting temporal or spatial "structures" that have been learned from the input space.

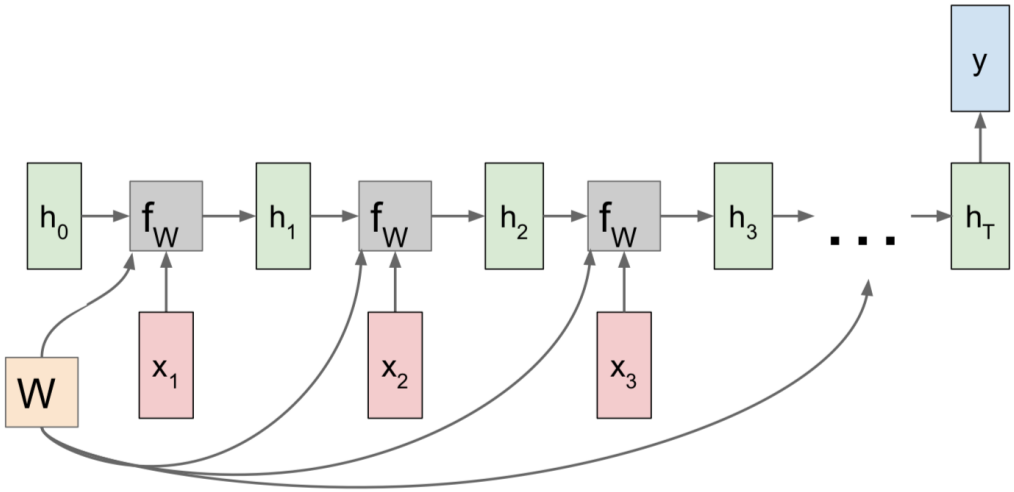
- Here we deal with *sequential* data that can be related to space (image), time (speech, music) or sequence of symbols (text, DNA).



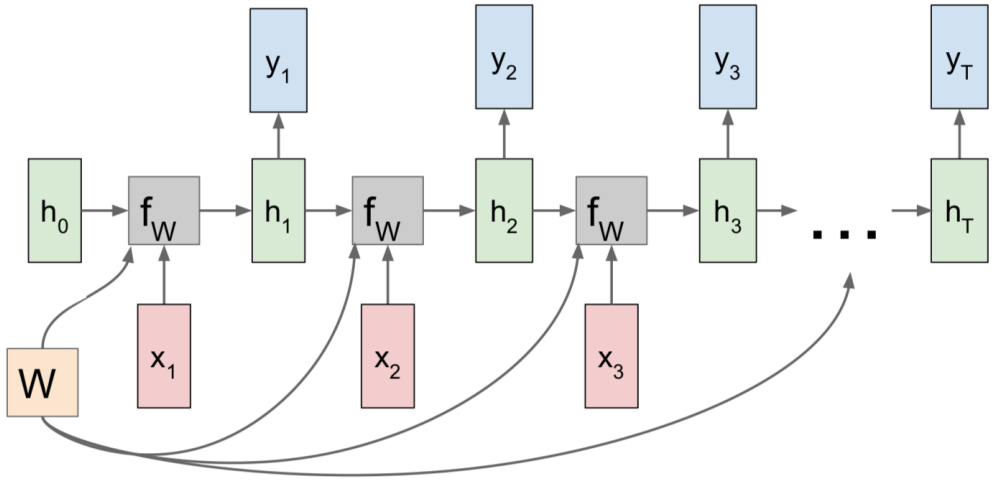
How to generate new sequential data with RNNs ?

- 2 approaches

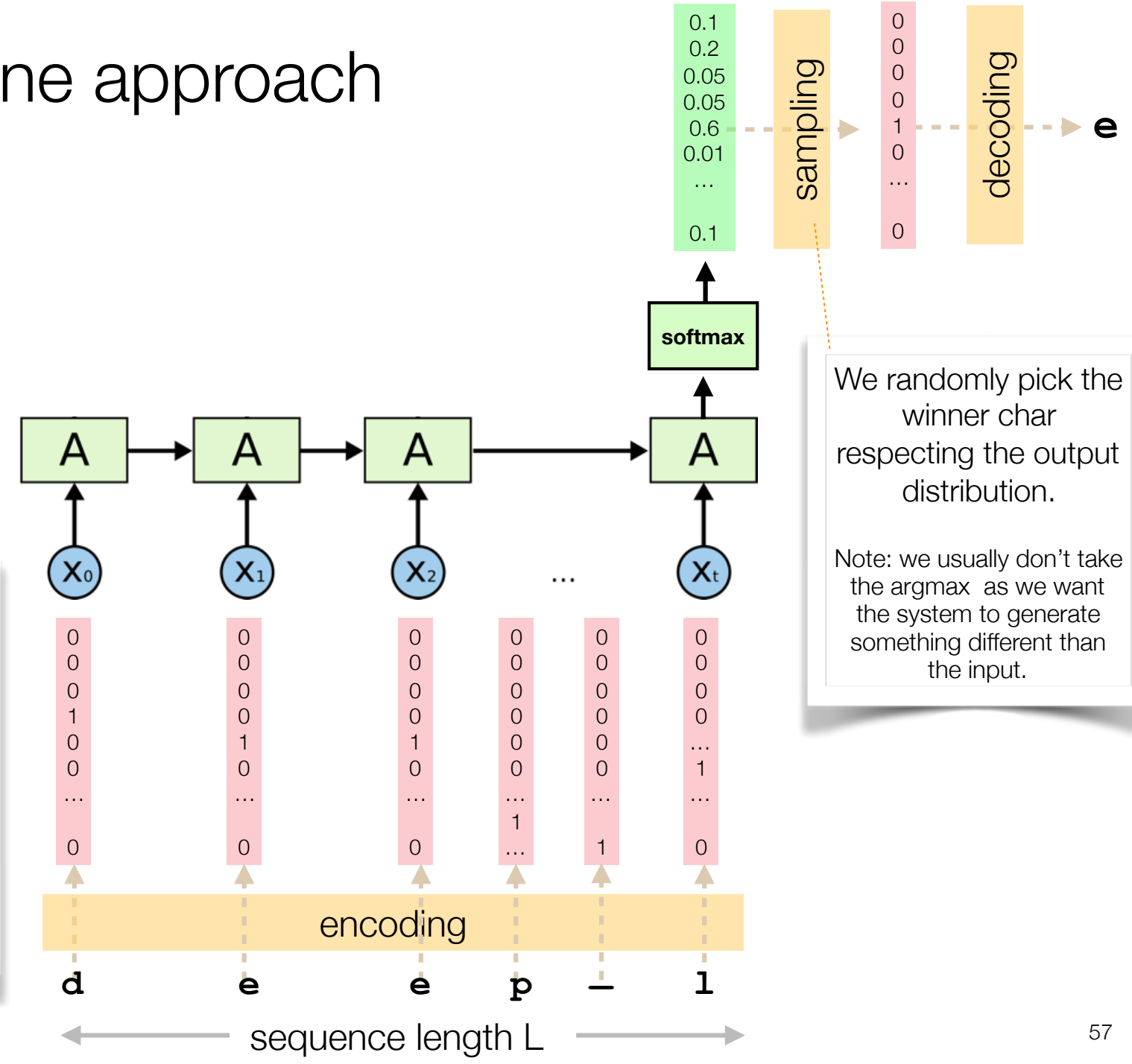
Many to one



Many to many



Many to one approach



One-hot vector with a '1' at the position according to the position of the letter in the alphabet size D

```
{'a', 'b', 'c',…,
 'A', 'B', 'C',…,
 '-', "'",
'_','END'}
```

We randomly pick the winner char respecting the output distribution.

Note: we usually don't take the argmax as we want the system to generate something different than the input.

Example on Shakespeare sonnets

- Training set:
  - 94K character of the sonnets of Shakespeare.
  - 62 different chars

sonnets.txt

~/SWITCHdrive/teaching/mse-tsm-dl/teaching/week11/examples-in-class/sonnets.txt

```
From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou, contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thy self thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And tender churl mak'st waste in niggarding:
  Pity the world, or else this glutton be,
  To eat the world's due, by the grave and thee.

When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a tatter'd weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
  This were to be new made when thou art old,
  And see thy blood warm when thou feel'st it cold.
```

Preliminary discussion - Shakespeare sonnets

- What can we expect out of the RNN?
  - Are the generated phrases going to “make sense”?
  - Are the words going to be valid words?
- How can we evaluate a generator?
  - At training time
  - At testing time

Define the many to one model in `Keras`

The model is very similar to what is used for the name classification. The only difference is that here we have D=62 categories as output.

SimpleRNN layer:

- **`units`:** specifies the dimension of the hidden state variable.
- **`return_sequences=False`:** that no sequence is returned, i.e. just the hidden state at the last step.
- **`input_shape`:** length of input sequence (`sentence_length`) and dimension of an element of the input sequence (`num_char`)

We add a softmax layer for the binary classification.

```
# Define our model
model = Sequential()
model.add(SimpleRNN(256, input_shape=(sentence_length, num_chars),
                    return_sequences=False))
model.add(Dense(num_chars))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_2 (SimpleRNN)     (None, 256)               81664
_________________________________________________________________
dense_2 (Dense)              (None, 62)                15934
_________________________________________________________________
activation_2 (Activation)    (None, 62)                0
=================================================================
Total params: 97,598
Trainable params: 97,598
Non-trainable params: 0
_________________________________________________________________
```

$$\mathbf{w}_h : 256 \times 256 = 65536$$
$$\mathbf{w}_x : 256 \times 62 = 15872$$
$$\mathbf{b}_h : 256 \times 1 = 256$$

81664

$$256 \times 62 + 62 = 15934$$

`# unique char = 62`

Load the data

The corpus has a sequence length of 94652 chars.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation, SimpleRNN
from keras.callbacks import ModelCheckpoint
from random import randint
from matplotlib import pyplot as pl
```

```
Using TensorFlow backend.
```

```
with open("sonnets.txt") as corpus_file:
    corpus = corpus_file.read()
    corpus_length = len(corpus)
print("Loaded a corpus of {0} characters".format(corpus_length))
```

```
Loaded a corpus of 94652 characters
```

Get a list of unique chars : 62

Define en encoding (char to int) and decoding dictionary (int to char).

```
# Get a unique identifier for each char in the corpus,
# then make some dicts to ease encoding and decoding
chars = sorted(list(set(corpus)))
num_chars = len(chars)
encoding = {c: i for i, c in enumerate(chars)}
decoding = {i: c for i, c in enumerate(chars)}
print("Our corpus contains {0} unique characters.".format(num_chars))
```

```
Our corpus contains 62 unique characters.
```

Prepare the inputs and outputs

`sentence_length` is the length of the sequence and `skip` defines the stride we do on the input sequence

Chop the corpus into sentences of length 20. Take the 21st char as target.

A given input sequence is encoded into the corresponding char index defined in the encoding dictionary.

An example of x and y.

```
# chop up our data into X and y, slice into roughly
# (num_chars / skip) overlapping 'sentences' of length
# sentence_length, and encode the chars
sentence_length = 20
skip = 1
X_data = []
y_data = []
for i in range (0, len(corpus) - sentence_length, skip):
    sentence = corpus[i:i + sentence_length]
    next_char = corpus[i + sentence_length]
    X_data.append([encoding[char] for char in sentence])
    y_data.append(encoding[next_char])

num_sentences = len(X_data)
print("Sliced our corpus into {0} sentences of length {1}"
      .format(num_sentences, sentence_length))
```

```
Sliced our corpus into 94632 sentences of length 20
```

```
print(X_data[1])
```

```
[17, 52, 49, 47, 1, 40, 35, 43, 52, 39, 53, 54, 1, 37, 52, 39,
35, 54, 55, 52, 39, 53, 1, 57, 39, 1, 38, 39, 53, 43, 52, 39, 1
, 43, 48, 37, 52, 39, 35, 53, 39, 6, 0, 30, 42, 35, 54, 1, 54,
42]
```

```
print([decoding[idx] for idx in X_data[1]])
print(decoding[y_data[1]])
```

```
['F', 'r', 'o', 'm', ' ', 'f', 'a', 'i', 'r', 'e', 's', 't', '
', 'c', 'r', 'e', 'a', 't', 'u', 'r']
e
```

Prepare the inputs and outputs and train

Vectorise the data and target into one-hot representation.

```
# Vectorize our data and labels. We want everything in one-hot.
print("Vectorizing X and y...")
X = np.zeros((num_sentences, sentence_length, num_chars), dtype=np.bool)
y = np.zeros((num_sentences, num_chars), dtype=np.bool)
for i, sentence in enumerate(X_data):
    for t, encoded_char in enumerate(sentence):
        X[i, t, encoded_char] = 1
    y[i, y_data[i]] = 1
```

```
Vectorizing X and y...
```

Sanity check on the tensor sizes.

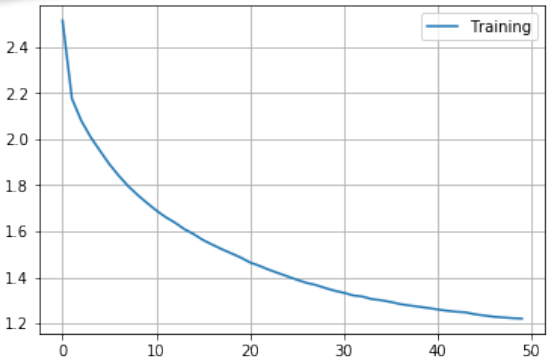
```
# Double check our vectorized data before we sink hours into fitting a model
print("Sanity check y. Dimension: {0} # Sentences: {1} Characters in corpus: {2}"
      .format(y.shape, num_sentences, len(chars)))
print("Sanity check X. Dimension: {0} Sentence length: {1}"
      .format(X.shape, sentence_length))
```

```
Sanity check y. Dimension: (94632, 62) # Sentences: 94632 Characters in corpus: 62
Sanity check X. Dimension: (94632, 20, 62) Sentence length: 20
```

Train the network

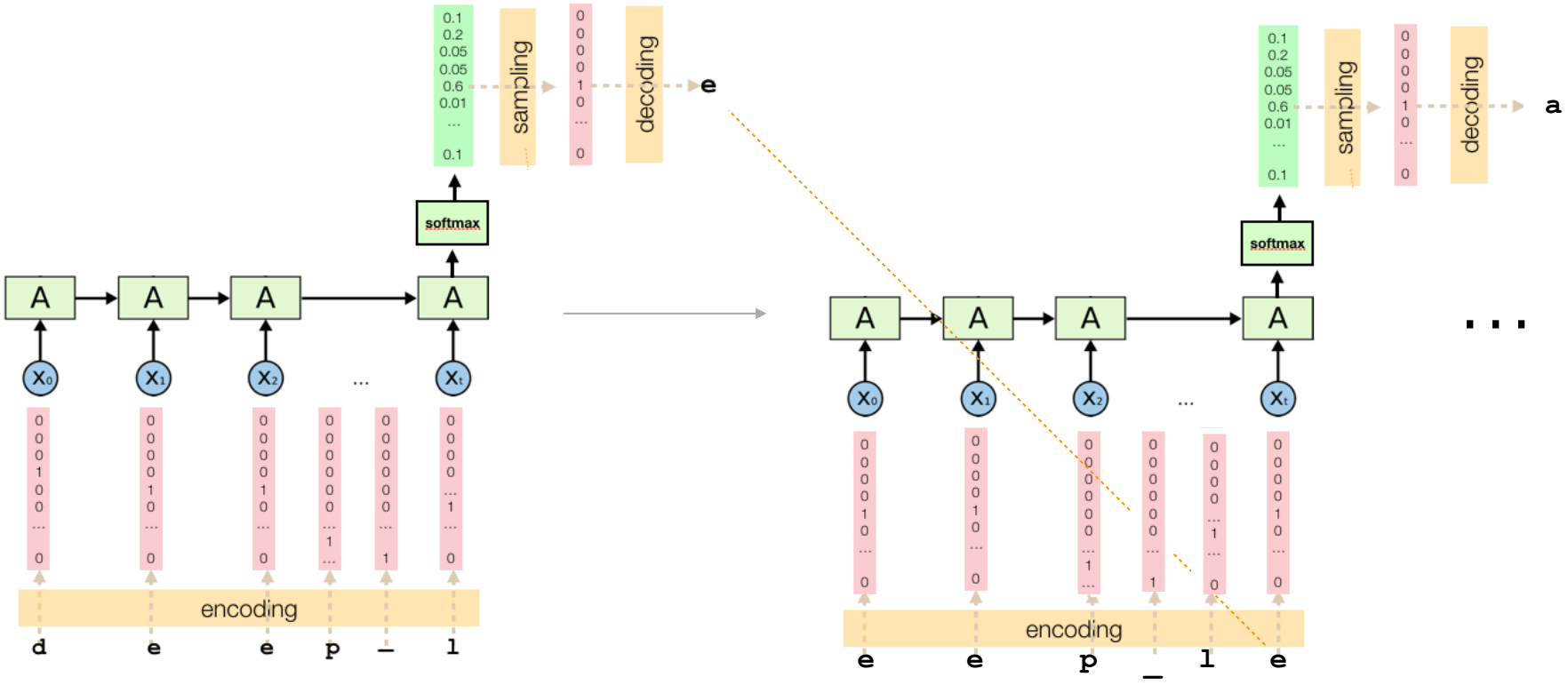
```
#training time
log = model.fit(X, y, epochs=50, batch_size=128)
```

```
Epoch 1/50
94632/94632 [==============================] - 14s 148us/step - loss: 2.5147 - acc: 0.3067
Epoch 2/50
94632/94632 [==============================] - 15s 162us/step - loss: 2.1761 - acc: 0.3698
Epoch 3/50
94632/94632 [==============================] - 16s 166us/step - loss: 2.0797 - acc: 0.3913
Epoch 4/50
94632/94632 [==============================] - 19s 198us/step - loss: 2.0086 - acc: 0.4085
Epoch 5/50
94632/94632 [==============================] - 18s 194us/step - loss: 1.9491 - acc: 0.4222
Epoch 6/50
94632/94632 [==============================] - 17s 181us/step - loss: 1.8909 - acc: 0.4372
Epoch 7/50
94632/94632 [==============================] - 19s 196us/step - loss: 1.8411 - acc: 0.4488
Epoch 8/50
```



Using the trained network

1. Define the seed [d, e, e, p, , l]
2. Predict the next char: e
3. Shift the input [e, e, p, , l, e], go back to 2



Using the trained network

```
def make_seed(seed_phrase=""):
        if seed_phrase:
            phrase_length = len(seed_phrase)
            pattern = ""
            for i in range (0, sentence_length):
                pattern += seed_phrase[i % phrase_length]
        else:
            seed = randint(0, corpus_length - sentence_length)
            pattern = corpus[seed:seed + sentence_length]

        return pattern
```

```
seed_pattern = make_seed("In the early morning, the flower is shining")
print(seed_pattern)
```

```
In the early morning
```

```
X = np.zeros((1, sentence_length, num_chars), dtype=np.bool)
for i, character in enumerate(seed_pattern):
    X[0, i, encoding[character]] = 1

generated_text = ""
for i in range(500):
    #prediction = np.argmax(model.predict(X, verbose=0))
    output_prob = model.predict(X, verbose=0)[0]
    prediction = np.random.choice(num_chars, p = output_prob )
    generated_text += decoding[prediction]
    activations = np.zeros((1, 1, num_chars), dtype=np.bool)
    activations[0, 0, prediction] = 1
    #now remove first char and glue the predicted one
    X = np.concatenate((X[:, 1:, :], activations), axis=1)
```

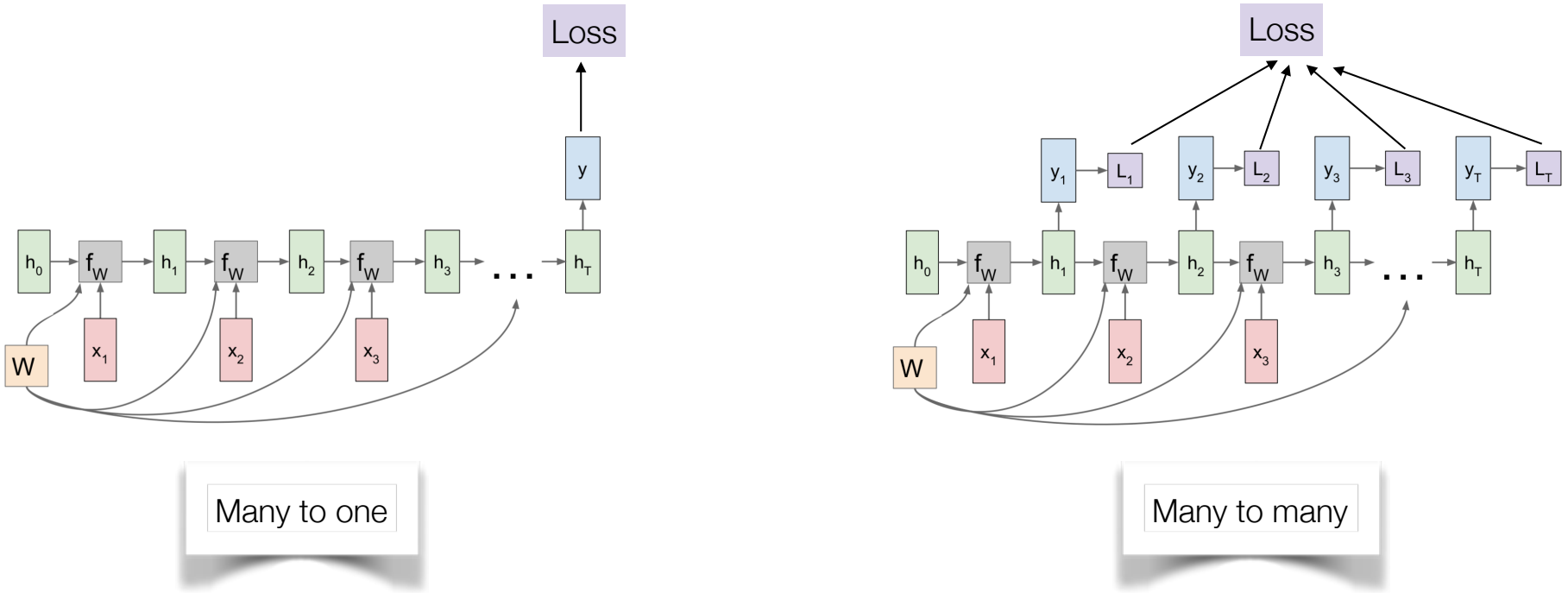
```
print(generated_text)
```

```
 Time strongly than visth canth, dayt  on doul shad's,
Which powhing me,
Rescrech's sumbly imiss of townt
A'd you, worses to can facr well knowl and sack far ade, be bedes
s to enceems.
Thou hast the dane:
Then love, to wilties mad, and earth, mine eyes.

How nith unus of her aspee
When I sumbe days
No lays face is fouldangly aresoner.
Hertran whency, ketwanty byid?
Make say wset acking thy pirtucts mads,
And this buck of mustress'd the spyit doth gend
Thoughts thines nother,
Htweres behred out,
```

We randomly pick the winner char respecting the output distribution.

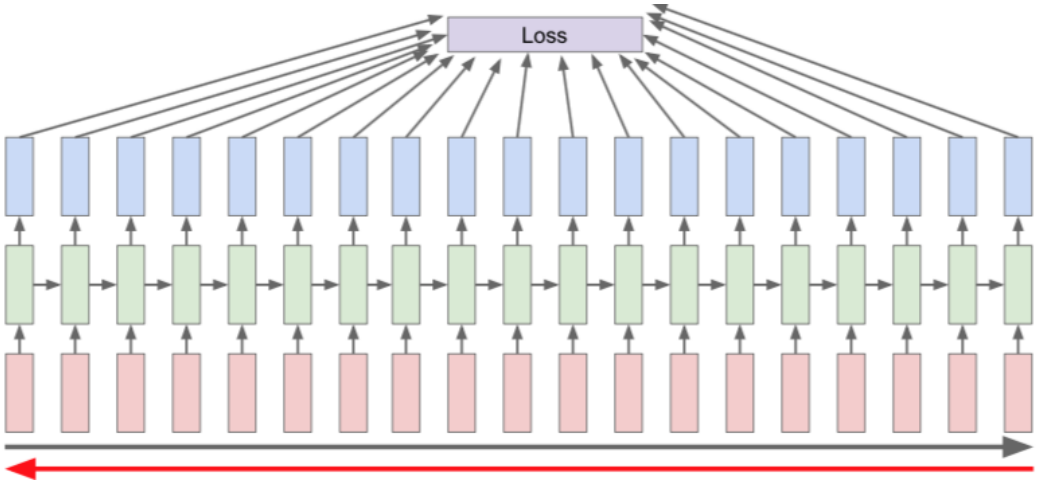
Shift the input with the new generated char and begin again.

Many to many approach

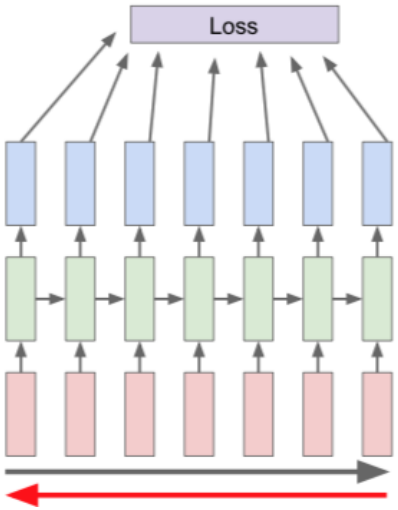


- The difference is mainly at training time where the RNN cells are emitting an output at each step.
- The loss is computed from all the outputs, versus 1 output on the many to one approach
- The output layer is “time distributed” to the whole sequence

Truncated back propagation through time



In theory: forward through the entire sequence to compute loss, then backward through the entire sequence to compute gradient. But not really feasible if the sequence is long.



In practice: run forward and backward through chunks of the sequence instead of whole sequence

Define the many-to-many model in `Keras`

SimpleRNN layer:

- **`return_sequences=True`:** to tell the network to give an output at each step of the sequence.

We wrap the Dense output layer with a TimeDistributed layer to "distribute the output over each step of the sequence".

```
# Define our model
model = Sequential()
model.add(SimpleRNN(256, input_shape=(sentence_length, num_chars),
                    return_sequences=True))
model.add(TimeDistributed(Dense(num_chars, activation='softmax')))
model.compile(loss='categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (None, 20, 256)           81664
_________________________________________________________________
time_distributed_1 (TimeDist (None, 20, 62)            15934
=================================================================
Total params: 97,598
Trainable params: 97,598
Non-trainable params: 0
_________________________________________________________________
```

$$\mathbf{w}_h : 256 \times 256 = 65536$$
$$\mathbf{w}_x : 256 \times 62 = 15872$$
$$\mathbf{b}_h : 256 \times 1 = 256$$

81664

$$256 \times 62 + 62 = 15934$$

`# unique char = 62`

Data preparation is a bit different for y

For each x char input, the y is the char at position + 1

```
# chop up our data into X and y, slice into roughly
# (num_chars / skip) overlapping 'sentences' of length
# sentence_length, and encode the chars
sentence_length = 20
skip = 1
X_data = []
y_data = []
for i in range (0, len(corpus) - sentence_length, skip):
    sentence = corpus[i:i + sentence_length]
    next_char = corpus[i+1:i+1 + sentence_length]
    X_data.append([encoding[char] for char in sentence])
    y_data.append([encoding[char] for char in next_char])

num_sentences = len(X_data)
print("Sliced our corpus into {0} sentences of length {1}"
      .format(num_sentences, sentence_length))
```

```
Sliced our corpus into 94632 sentences of length 20
```

```
print(X_data[1])
```

```
[61, 17, 52, 49, 47, 1, 40, 35, 43, 52, 39, 53, 54, 1, 37, 52, 39, 35,
54, 55]
```

```
print([decoding[idx] for idx in X_data[1]])
print([decoding[idx] for idx in y_data[1]])
```

```
['F', 'r', 'o', 'm', ' ', 'f', 'a', 'i', 'r', 'e', 's', 't', ' ', 'c',
'r', 'e', 'a', 't', 'u', 'r']
['r', 'o', 'm', ' ', 'f', 'a', 'i', 'r', 'e', 's', 't', ' ', 'c', 'r',
'e', 'a', 't', 'u', 'r', 'e']
```

y is then a tensor of the same dimensions as x: (94632, 20, 62)

```
# Vectorize our data and labels. We want everything in one-hot.
print("Vectorizing X and y...")
X = np.zeros((num_sentences, sentence_length, num_chars), dtype=np.bool)
y = np.zeros((num_sentences, sentence_length, num_chars), dtype=np.bool)
for i, sentence in enumerate(X_data):
    for t, encoded_char in enumerate(sentence):
        X[i, t, encoded_char] = 1
for i, sentence in enumerate(y_data):
    for t, encoded_char in enumerate(sentence):
        y[i, t, encoded_char] = 1
```

Using the trained network

```
def make_seed(seed_phrase=""):
        if seed_phrase:
            phrase_length = len(seed_phrase)
            pattern = ""
            for i in range (0, sentence_length):
                pattern += seed_phrase[i % phrase_length]
        else:
            seed = randint(0, corpus_length - sentence_length)
            pattern = corpus[seed:seed + sentence_length]

        return pattern
```

```
seed_pattern = make_seed("In the early morning, the flower is shining")
print(seed_pattern)
```

```
In the early morning
```

```
X = np.zeros((1, sentence_length, num_chars), dtype=np.bool)
for i, character in enumerate(seed_pattern):
    X[0, i, encoding[character]] = 1

generated_text = ""
for i in range(500):
    #prediction = np.argmax(model.predict(X, verbose=0))
    output_prob = model.predict(X, verbose=0)[0]
    #print(output_prob.shape)
    prediction = np.random.choice(num_chars, p = output_prob[-1] )
    generated_text += decoding[prediction]
    activations = np.zeros((1, 1, num_chars), dtype=np.bool)
    activations[0, 0, prediction] = 1
    #now remove first char and glue the predicted one
    X = np.concatenate((X[:, 1:, :], activations), axis=1)
```

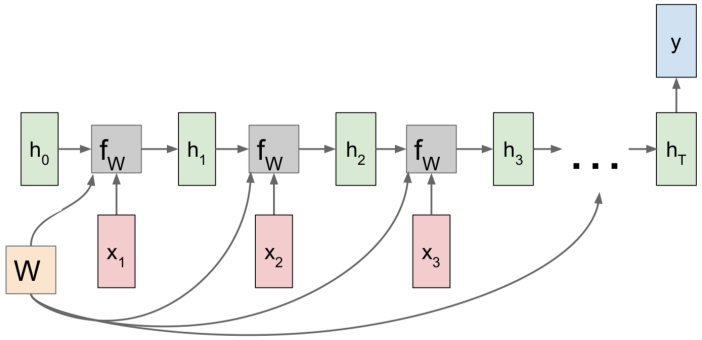
```
print(generated_text)
```

```
 hast thy spiencies deyis, in eyes, or nanks to age:
The dearing,
Why showers wime,
For then love's sour rruelful dlace,
Behat old, une to be fear any morrany.
Diviest my mind,
When hearder vaste:
Bug, betchange chedirs are foather where true beauty sad you lifter pl
ear,
And reases anst;
But not dratch, whereing eweres brad
That to his amisy
Make tran.
  No wan might thou better nor.

The ads his treasure,
  Fart, hell in luste doth desert know
Than higns yet to may:
Itatuch thou dost ride.

O!
```

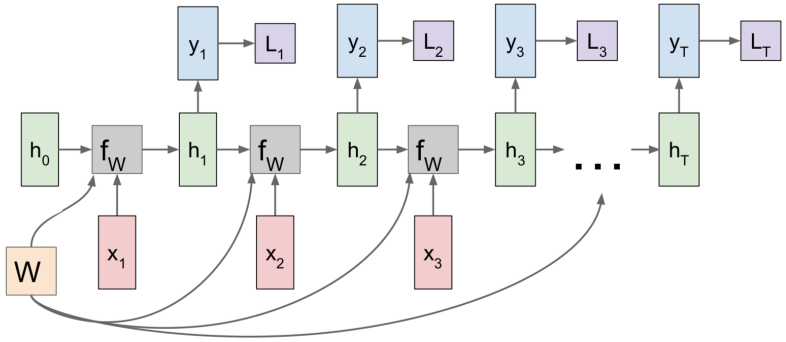
We randomly pick the winner char respecting the output distribution.

Shift the input with the new generated char and begin again.

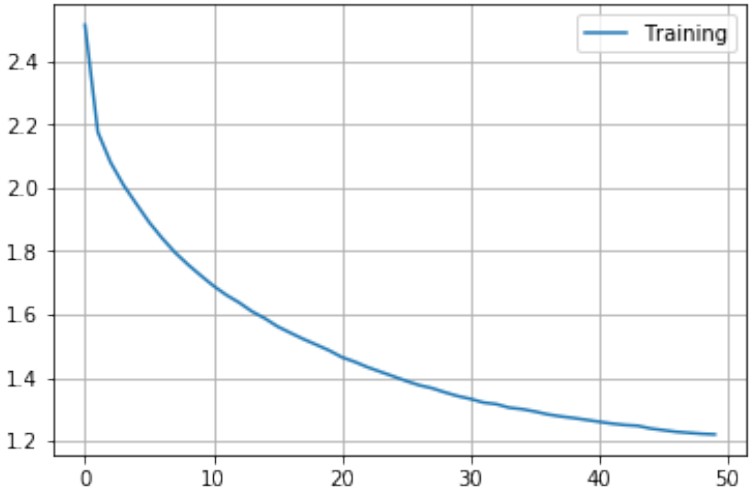
Loss evolution comparison

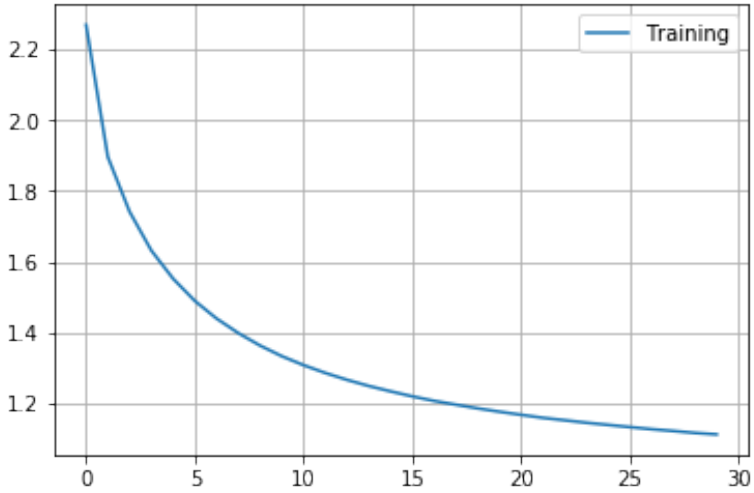


Many to one



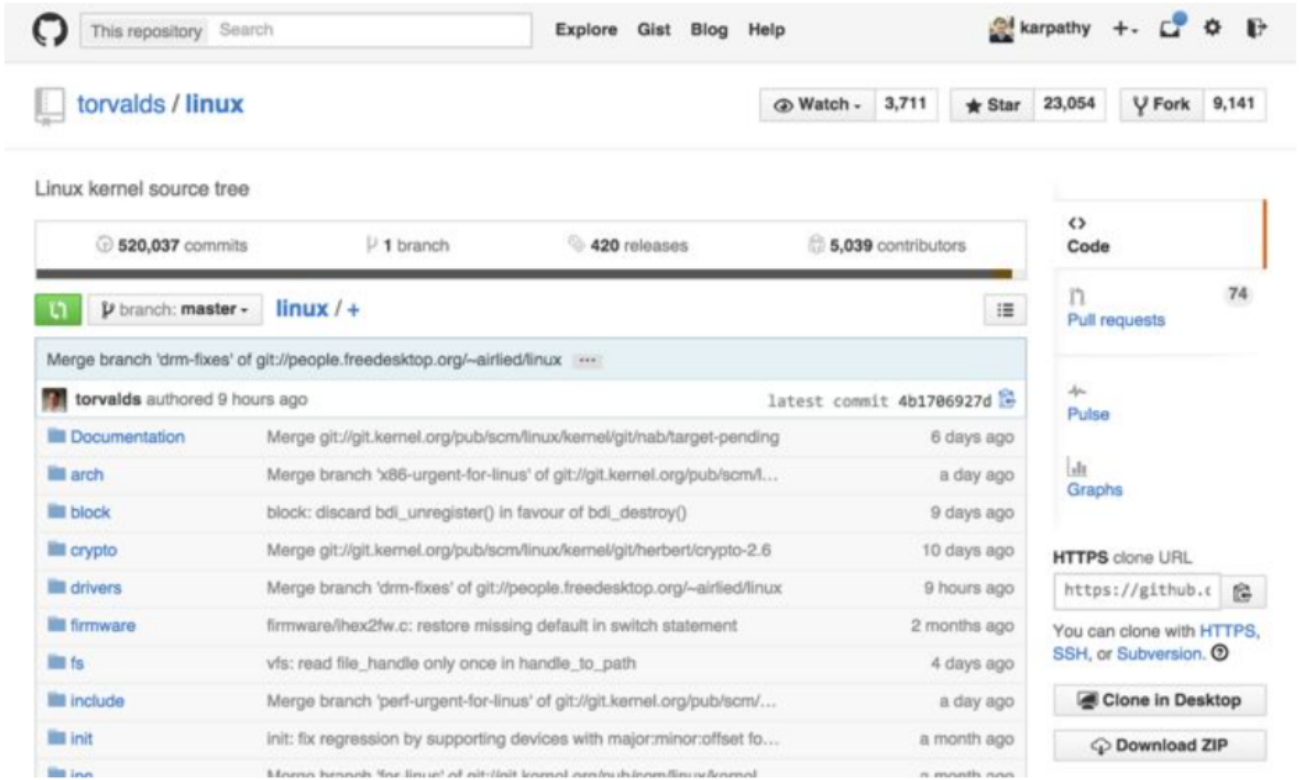
Many to many





Other examples - C code generation

- Train a RNN on the source code of Linux kernel



Other examples - C code generation

```
/*
 *  Copyright (c) 2006-2010, Intel Mobile Communications.  All rights reserved.
 *
 *   This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 as published by
 * the Free Software Foundation.
 *
 *        This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *
 *  GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *    along with this program; if not, write to the Free Software Foundation,
 *  Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

#include <linux/kexec.h>
#include <linux/errno.h>
#include <linux/io.h>
#include <linux/platform_device.h>
#include <linux/multi.h>
#include <linux/ckevent.h>

#include <asm/io.h>
#include <asm/prom.h>
#include <asm/e820.h>
#include <asm/system_info.h>
#include <asm/setew.h>
#include <asm/pgproto.h>
```

```
static void do_command(struct seq_file *m, void *v)
{
  int column = 32 << (cmd[2] & 0x80);
  if (state)
    cmd = (int)(int_state ^ (in_8(&ch->ch_flags) & Cmd) ? 2 : 1);
  else
    seq = 1;
  for (i = 0; i < 16; i++) {
    if (k & (1 << 1))
      pipe = (in_use & UMXTHREAD_UNCCA) +
        ((count & 0x00000000fffffff8) & 0x000000f) << 8;
    if (count == 0)
      sub(pid, ppc_md.kexec_handle, 0x20000000);
    pipe_set_bytes(i, 0);
  }
  /* Free our user pages pointer to place camera if all dash */
  subsystem_info = &of_changes[PAGE_SIZE];
  rek_controls(offset, idx, &soffset);
  /* Now we want to deliberately put it to device */
  control_check_polarity(&context, val, 0);
  for (i = 0; i < COUNTER; i++)
    seq_puts(s, "policy ");
}
```

Wrap-up

- **Why RNNs?**
  - RNN helps wherever we need context from the previous input
  - Many applications: Speech recognition, video captioning, machine translation, chatbots, text generation
  - One -to-many, many-to-one, many-to-many approaches
- **Simple RNNs**
  - Notion of rolled vs un-rolled representations, notion of time-step in the processing, notion of state h of the cell that memories the context of past inputs
  - Gradient backpropagation is done as usual, e.g. on the un-rolled computational graph
    - Potential problems of vanishing or exploding gradient for which we need regularisation, e.g. gradient clipping
- **Generative RNNs** are able to generate new consistent data from a seed.
  - By consistent, we mean respecting temporal or spatial “structures” that have been learned from the input space.are neural networks trained to reproduce its input
  - Two approaches: many-to-one or many-to-many
  - The many-to-many approach needs a distribution of the output through time and a truncated back-propagation through time