Universitat de Barcelona

Jordi Vitrià

Deep Learning

Deep Sequential Models (and NLP)

Deep Sequential Models

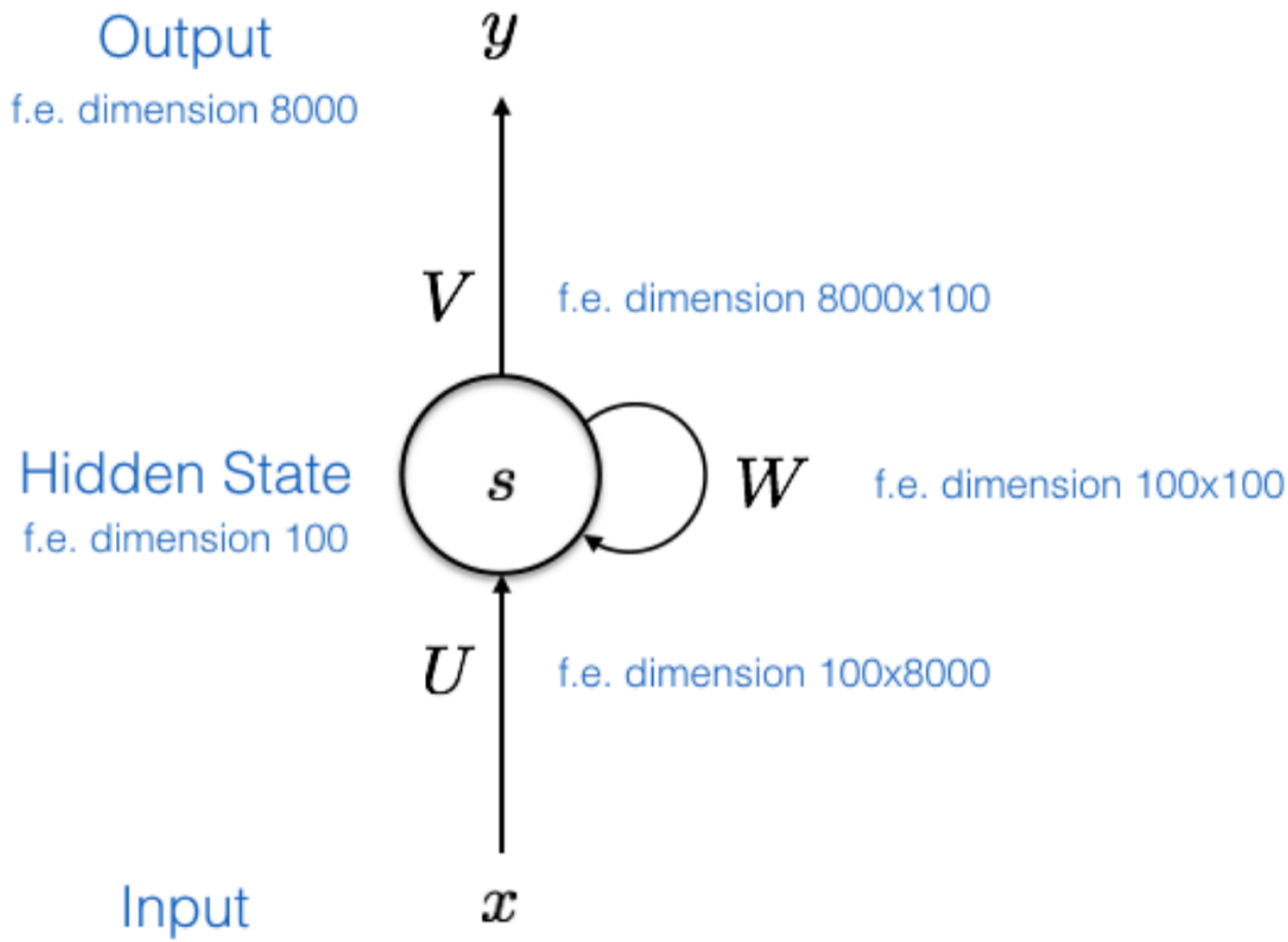
and some NLP

Classical neural networks, including convolutional ones, suffer from two severe limitations:

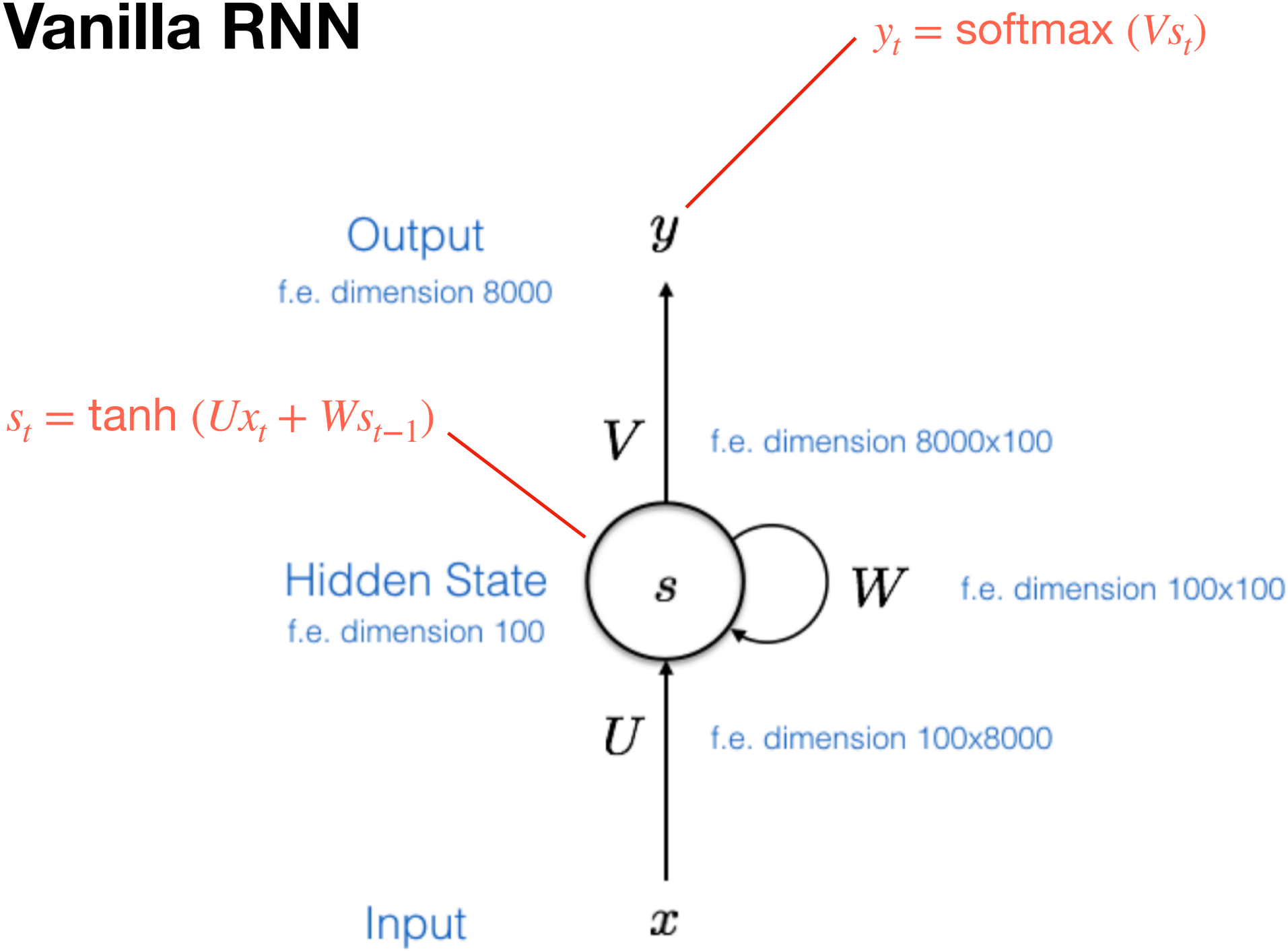
- They only accept a **fixed-sized tensors** as input and produce a fixed-sized tensor as output.
- They do not consider the **sequential nature of some data** (language, video frames, time series, etc.)

Recurrent neural networks overcome these limitations by allowing to operate over sequences of vectors (in the input, in the output, or both).

Vanilla RNN


Output
f.e. dimension 8000
y
V
f.e. dimension 8000x100
Hidden State
f.e. dimension 100
s
W
f.e. dimension 100x100
U
f.e. dimension 100x8000
Input
f.e. dimension 8000
x
TOTAL: 1,610,000 parameters

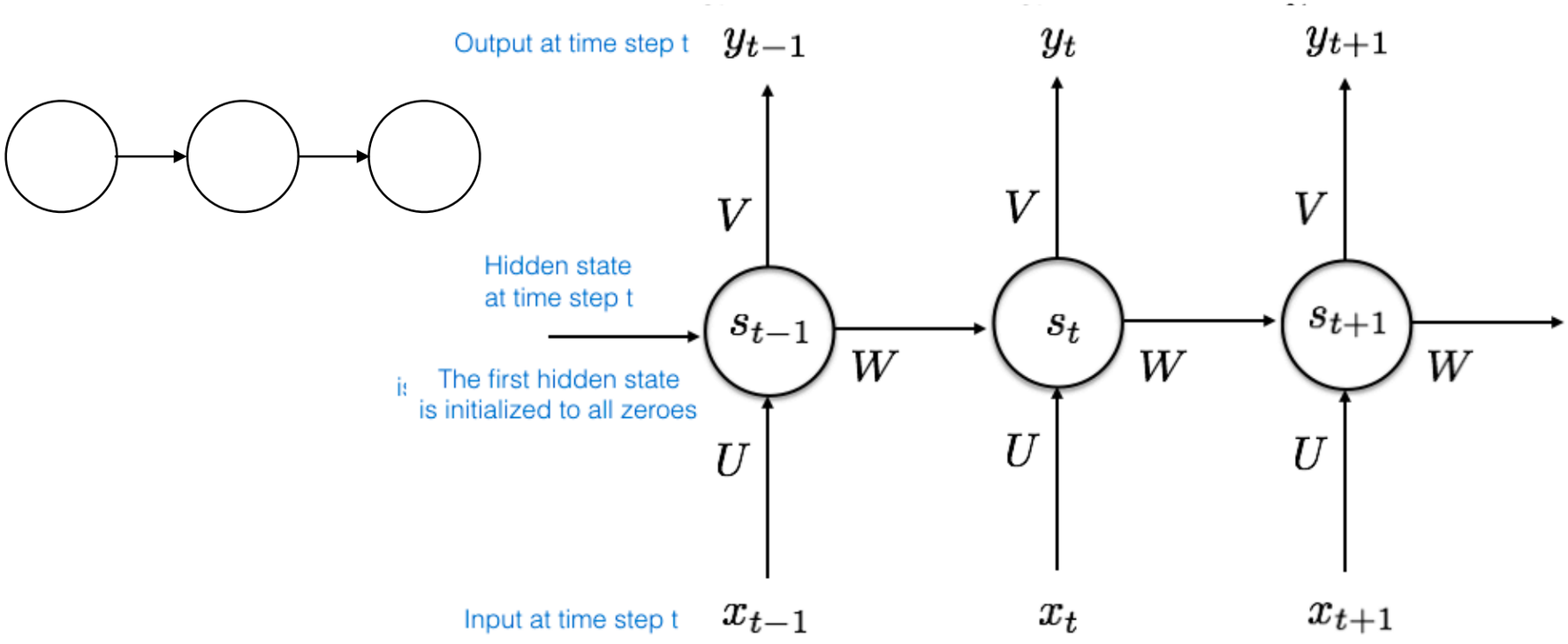
Vanilla RNN


$y_t = \text{softmax}\,(Vs_t)$
Output
f.e. dimension 8000
y
$s_t = \tanh\,(Ux_t + Ws_{t-1})$
V
f.e. dimension 8000x100
Hidden State
f.e. dimension 100
s
W
f.e. dimension 100x100
U
f.e. dimension 100x8000
Input
f.e. dimension 8000
x
TOTAL: 1,610,000 parameters

Vanilla RNN

Instead of imagining that hidden state is being recurrently fed back into the network, it's easier to visualize the process if we **unroll** the operation into a computational graph that is composed to many time steps.

By unrolling we mean that we write out the network for the complete sequence.



Vanilla RNN

- We can think of the **hidden state** s_t as a memory of the network that captures information about the previous steps.
- The RNN **shares the parameters** U, V, W across all time steps.
- It is not necessary to have outputs y_t at each time step.

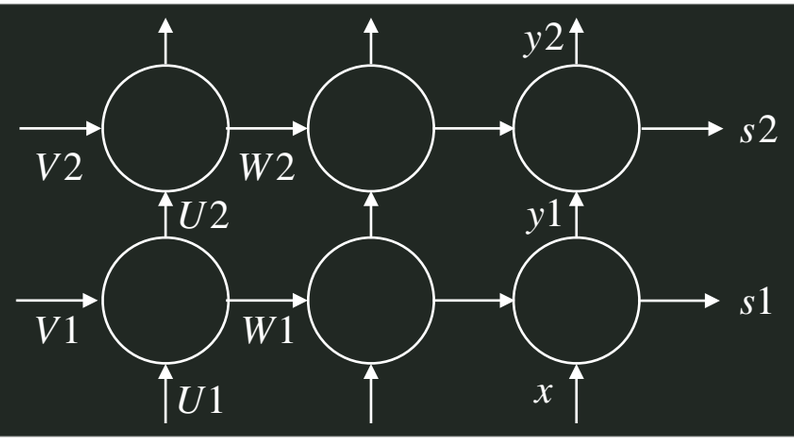
Vanilla RNN

```
def RNN_step(x):
    s = np.tanh(np.dot(W, s) + np.dot(U, x))
    y = np.dot(V, h)
    return y
```

This is heavy model because of the matrices!

We can go deep by stacking RNNs:

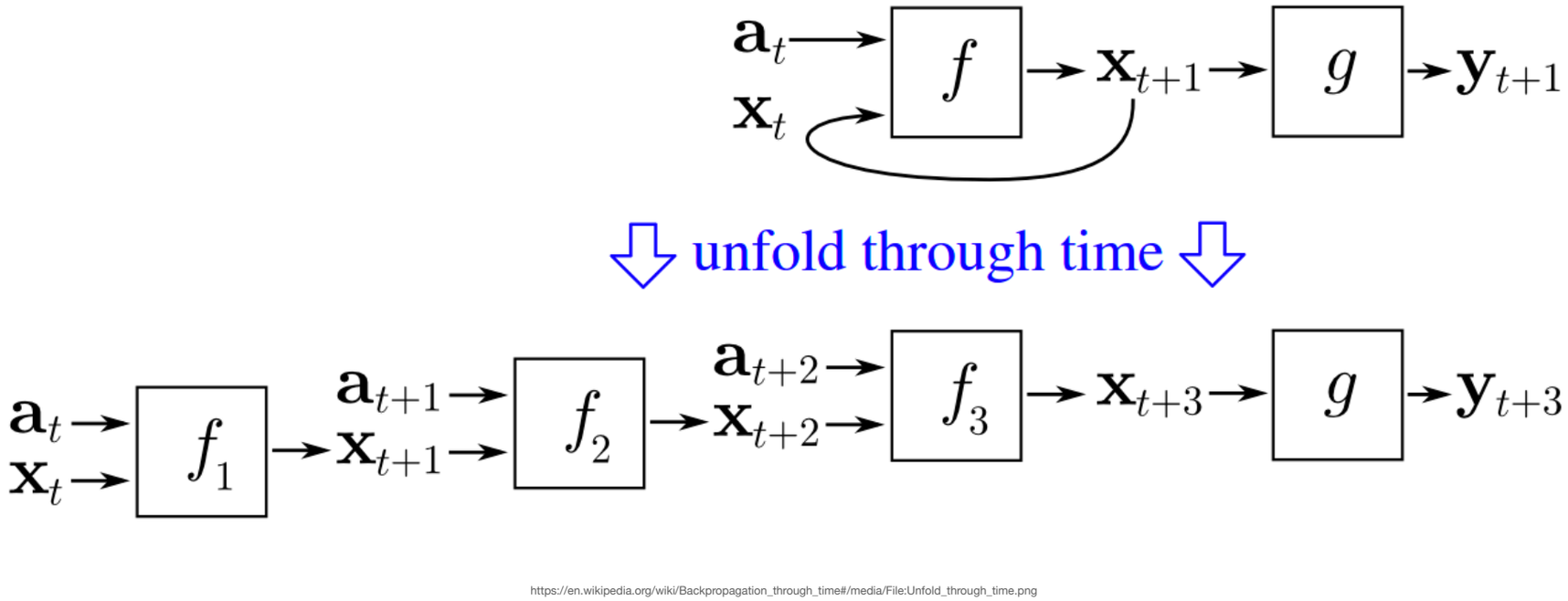
```
y1 = RNN.step(x)
y2 = RNN.step(y1)
```



Vanilla RNN

Training a RNN is similar to training a traditional NN, but some modifications.

The main reason is that parameters are shared by all time steps: in order to compute the gradient at $t = 4$, we need to propagate 3 steps and sum up the gradients. This is called **Backpropagation through time (BPTT)**.



https://en.wikipedia.org/wiki/Backpropagation_through_time#/media/File:Unfold_through_time.png

Vanilla RNN

Let's suppose we are classifying a **series of words**:

$$x_1, \ldots, x_{t-1}, x_t, x_{t+1}, \ldots x_T \, .$$

x_i are the word vectors corresponding to a corpus with T **symbols**.

Then, the relationship to compute the hidden layer output features at each time-step t is $h_t = \sigma(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$, where:

- $x_t \in \mathbb{R}^d$ is input word vector at time t.
- $W^{(hx)} \in \mathbb{R}^{D_h \times d}$ is the weights matrix used to condition the input word vector, x_t.
- $W^{(hh)} \in \mathbb{R}^{D_h \times D_h}$ is the weights matrix used to condition the output of the previous time-step, h_{t-1}.
- $h_{t-1} \in \mathbb{R}^{D_h}$ is the output of the non-linear function at the previous time-step, $t-1$.
- $h_0 \in \mathbb{R}^{D_h}$ is an initialization vector for the hidden layer at time-step $t = 0$.
- $\sigma()$ is the non-linearity function (normally, $tanh$).

We can compute the **output probability distribution over the vocabulary** at each time-step t as $\hat{y}_t = softmax(W^{(hy)} h_t)$. Essentially, $\hat{y}_t$ is the next predicted word given the document context score so far (i.e. h_{t-1}) and the last observed word vector $x^{(t)}$. Here, $W^{(S)} \in \mathbb{R}^{|V| \times D_h}$ and $\hat{y} \in \mathbb{R}^{|V|}$ where $|V|$ is the vocabulary.

Vanilla RNN

The loss function used in RNNs is often the cross-entropy error:

$$L^{(t)}(W) = -\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j})$$

The cross entropy error over a corpus of size T is:

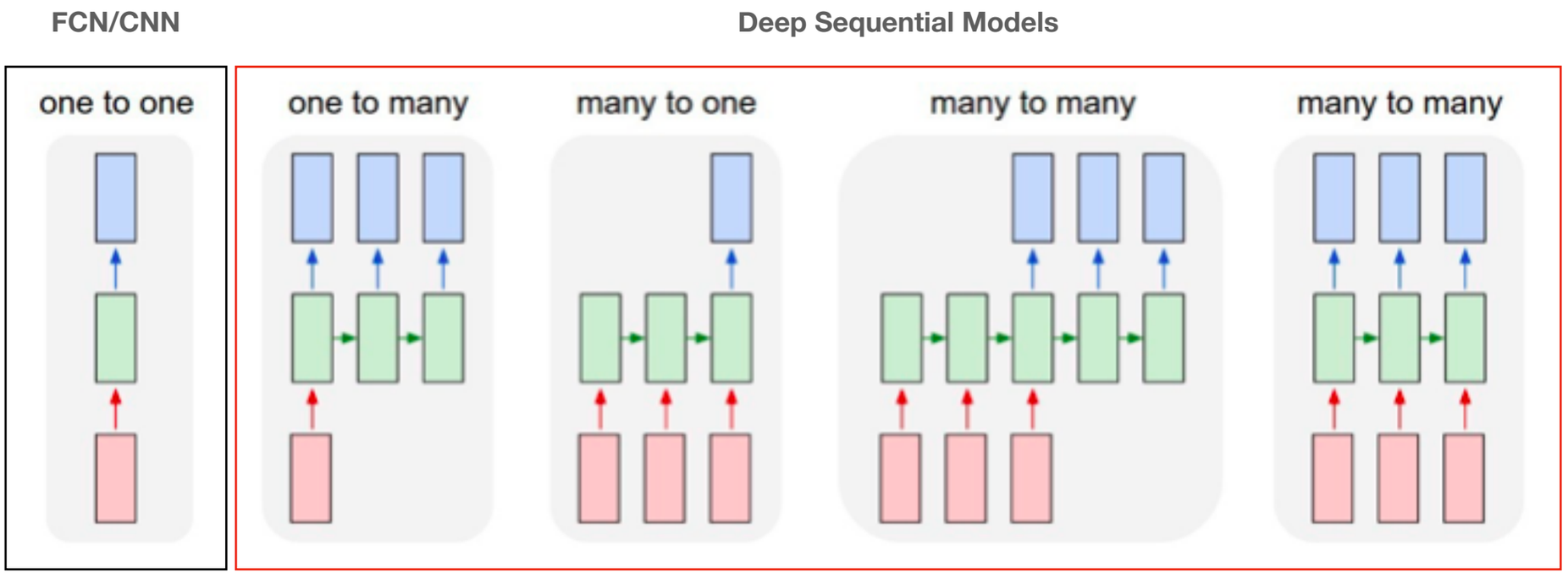
$$L = \frac{1}{T}\sum_{t=1}^{T} L^{(t)}(W) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j})$$

In the case of classifying a series of symbols/words, the **perplexity** measure can be used to assess the goodness of our model. It is basically 2 to the power of the negative log probability of the cross entropy error function:

$$Perplexity = 2^{L}$$

Perplexity is a measure of confusion where lower values imply more confidence in predicting the next word in the sequence (compared to the ground truth outcome).

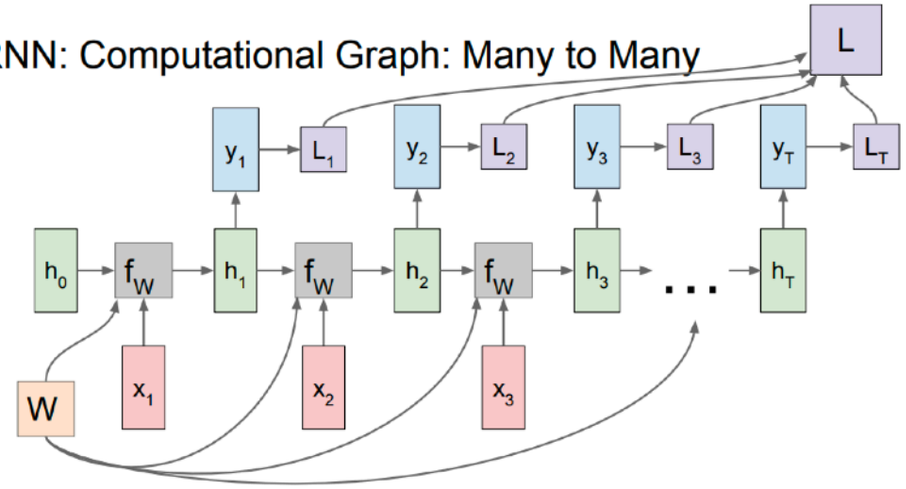
Deep Sequential Models



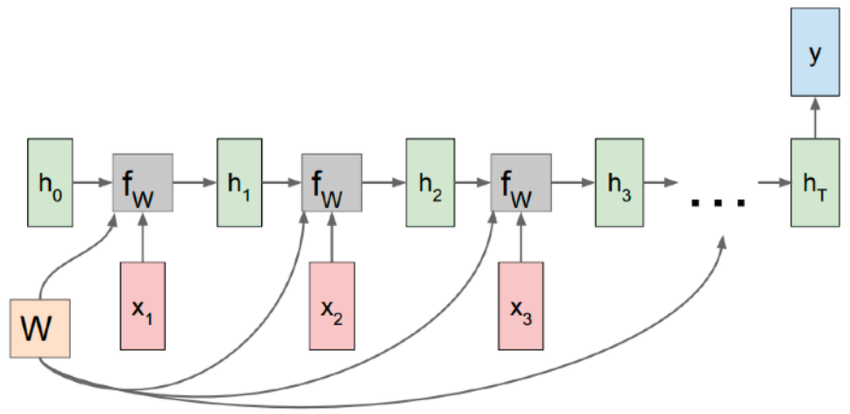
Source: http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Deep Sequential Models

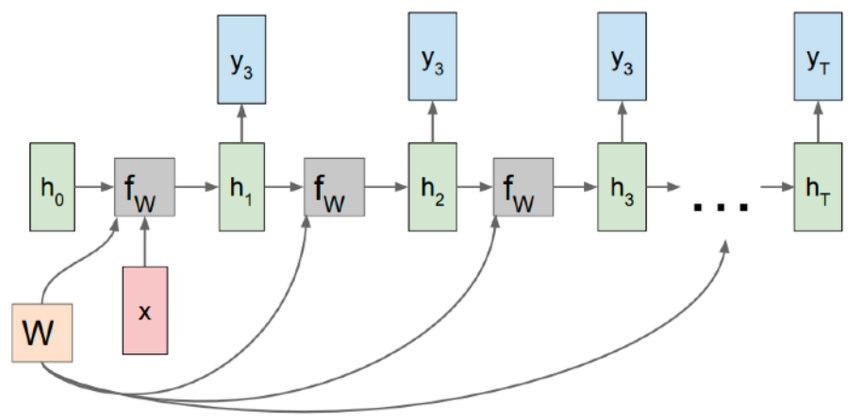
RNN: Computational Graph: Many to Many



RNN: Computational Graph: Many to One



RNN: Computational Graph: One to Many



https://calvinfeng.gitbook.io/machine-learning-notebook/supervised-learning/recurrent-neural-network/recurrent_neural_networks

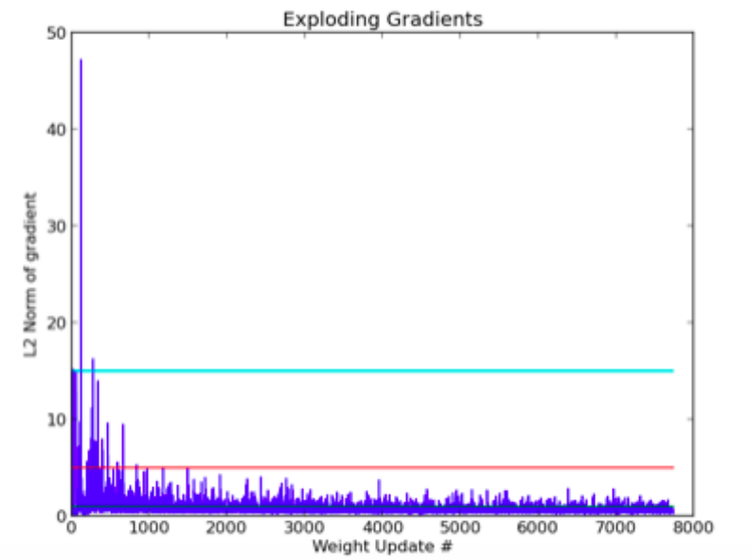
RNN Training Tricks

Recurrent neural networks propagate weight matrices from one time-step to the next. Recall the goal of a RNN implementation is to enable propagating context information through faraway time-steps. When these propagation results in a **long series of matrix multiplications**, weights can vanish or explode.

Once the gradient value grows extremely large, it causes an overflow (i.e. NaN) which is easily detectable at runtime; this issue is called the **Gradient Explosion Problem**.

When the gradient value goes to zero, however, it can go undetected while drastically reducing the learning quality of the model for far-away words in the corpus; this issue is called the **Vanishing Gradient Problem**.

To solve the problem of exploding gradients, Thomas Mikolov first introduced a simple heuristic solution that clips gradients to a small number whenever they explode. That is, whenever they reach a certain threshold, they are set back to a small number.



RNN Training Tricks

To solve the problem of vanishing gradients, instead of initializing $W^{(hh)}$ randomly, starting off from random orthogonal matrices works better, i.e., a square matrix W for which $W^T W = I$.

There are two properties of orthogonal matrices that are useful for training deep neural networks:

- they are norm-preserving, i.e., $\|Wx\|^2 = \|x\|^2$, and
- their columns (and rows) are all orthonormal to one another.

At least at the start of training, the first of these should help to keep the norm of the input constant throughout the network, which can help with the problem of exploding/vanishing gradients.

Similarly, an intuitive understanding of the second is that having orthonormal weight vectors encourages the weights to learn different input features.

RNN Training Tricks

You can obtain a random $n \times n$ orthogonal matrix W, (uniformly distributed) by performing a QR factorization of a $n \times n$ matrix with elements i.i.d. Gaussian random variables of mean 0 and variance 1.

```python
import numpy as np
from scipy.linalg import qr

n = 3
H = np.random.randn(n, n)
print(H)
print('\n')

Q, R = qr(H)

print (Q.dot(Q.T))
```

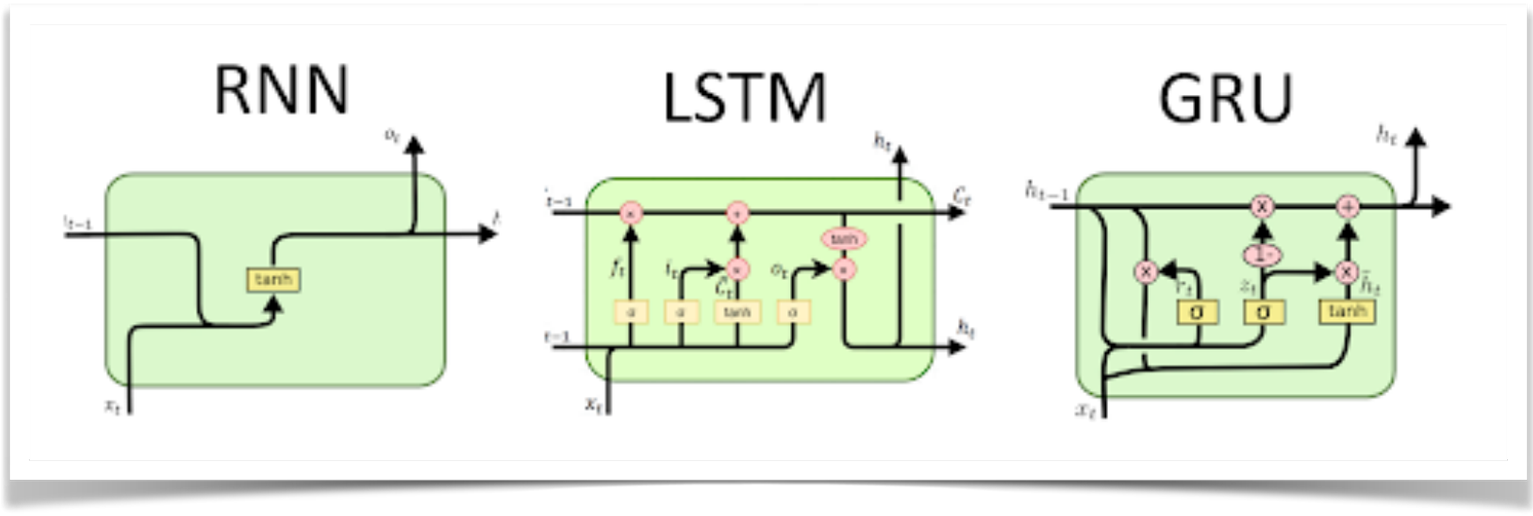
```
[[ 0.06351849  0.75175046 -0.10405964]
 [ 1.11485701  1.78923717 -0.9788983 ]
 [-0.08515308 -1.16475846  0.0640979 ]]


[[ 1.00000000e+00  5.49819565e-16 -1.29542567e-16]
 [ 5.49819565e-16  1.00000000e+00  6.99257794e-17]
 [-1.29542567e-16  6.99257794e-17  1.00000000e+00]]
```

Gated Units

The most important types of **gated RNN**s are:

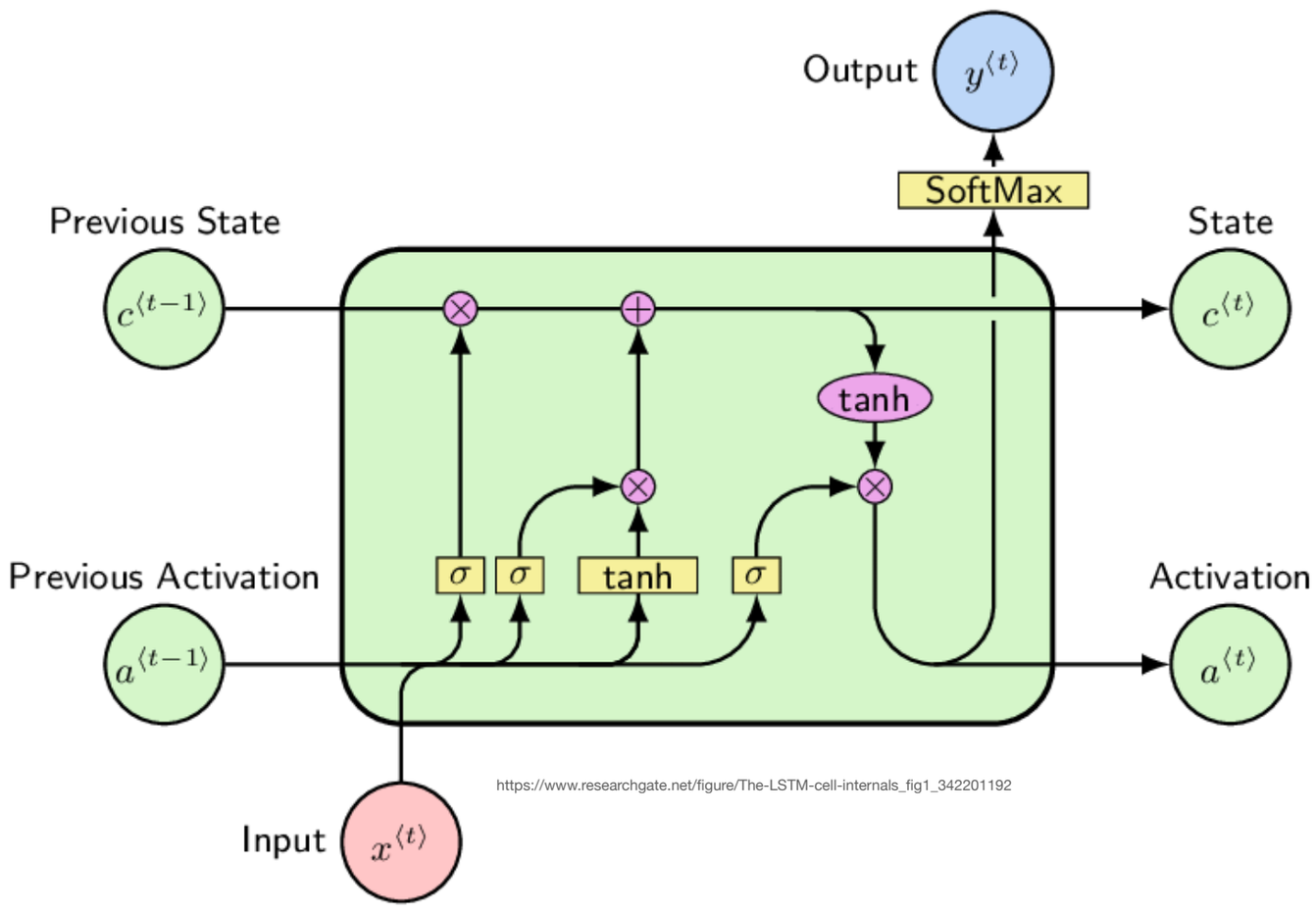
- **Long Short Term Memories** (LSTM). It was introduced by S.Hochreiter and J.Schmidhuber in 1997 and is widely used. LSTM is very good in the long run due to its high complexity.
- **Gated Recurrent Units** (GRU). It was recently introduced by K.Cho. It is simpler than LSTM, fasters and optimizes quicker.



LSTM

The key idea of LSTMs is the cell state C, the horizontal line running through the top of the diagram.

The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.

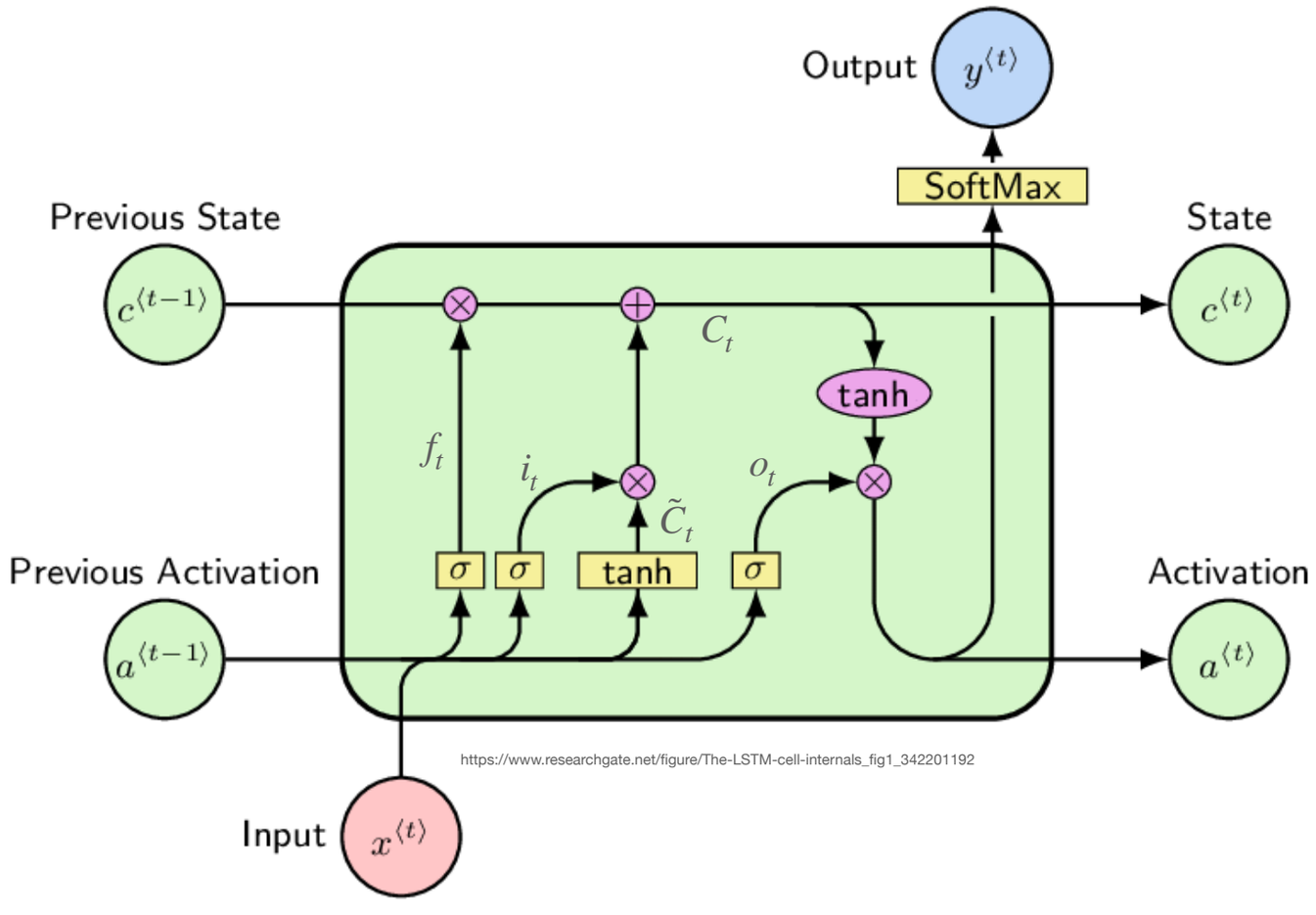


https://www.researchgate.net/figure/The-LSTM-cell-internals_fig1_342201192

LSTM

LSTM has the ability to **remove** or **add** information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a **sigmoid** neural net layer and a pointwise multiplication operation.



https://www.researchgate.net/figure/The-LSTM-cell-internals_fig1_342201192

$f_t = \sigma(W_f \cdot [a_{t-1}, x_t])$ (Forget gate)

$i_t = \sigma(W_i \cdot [a_{t-1}, x_t])$ (Input gate)

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

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

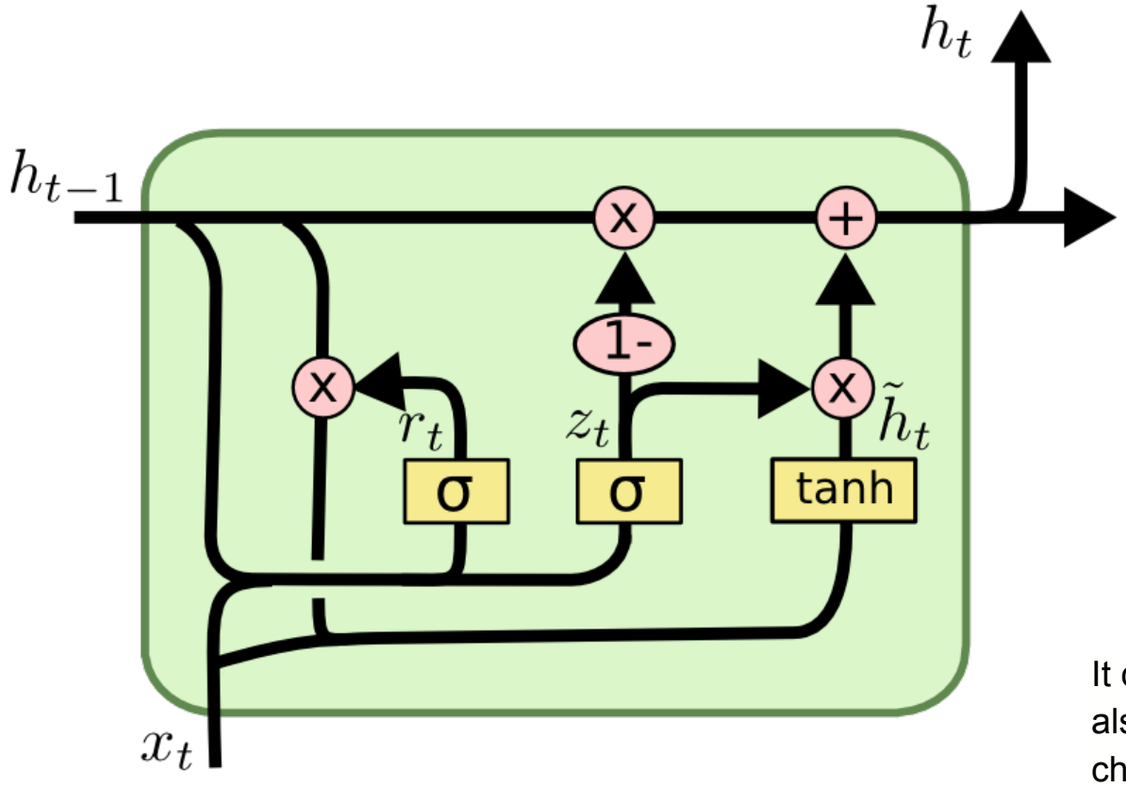
$o_t = \sigma(W_o \cdot [a_{t-1}, x_t])$

$h_t = o_t * \tanh(C_t)$ (Output gate)

GRU

The transition from hidden state h_{t-1} to h_t in vanilla RNN is defined by using an affine transformation and a point-wise nonlinearity.

Although RNNs can theoretically capture long-term dependencies, they are very hard to actually train to do this. Gated recurrent units are designed in a manner to have more persistent memory thereby making it easier for RNNs to capture long-term dependencies.



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

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

$\tilde{h}_t = \tanh(r_t \cdot [x_t, r_t \circ h_{t-1}])$ (New memory)

$h_t = (1 - z_t) \circ \tilde{h}_{t-1} + z_t \circ h_t$ (Hidden state)

It combines the forget and input gates into a single “update gate.” It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models.

Example in Keras

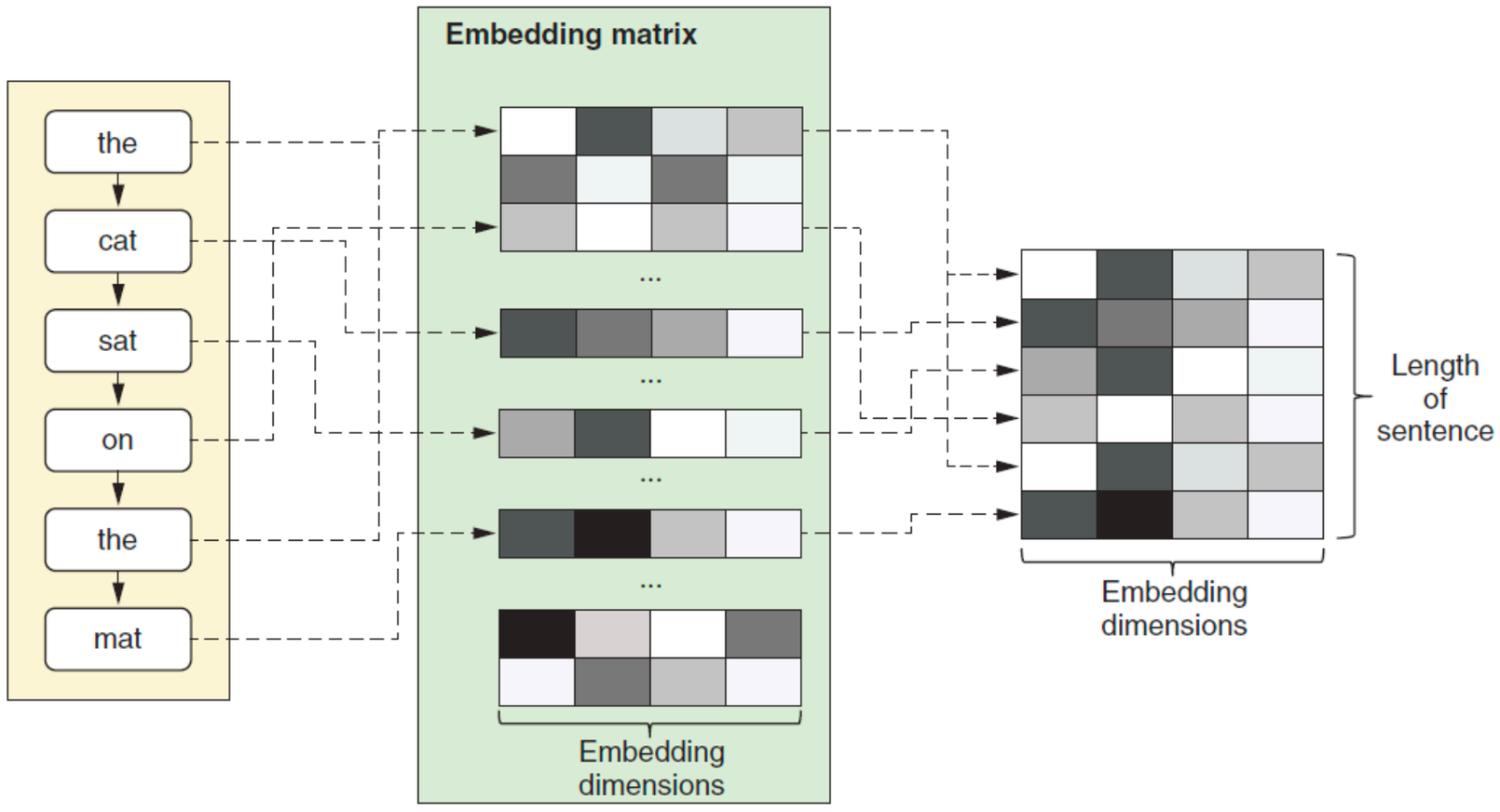
```
1 model = Sequential()
2
3 model.add(Embedding(vocab_size, embedding_dim))
4 model.add(Dropout(0.5))
5 model.add(LSTM(embedding_dim,return_sequences=True))
6 model.add(LSTM(embedding_dim))
7 model.add(Dense(6, activation='softmax'))
8
9 model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 64)          320000
_________________________________________________________________
dropout (Dropout)            (None, None, 64)          0
_________________________________________________________________
lstm (LSTM)                  (None, None, 64)          33024
_________________________________________________________________
lstm_1 (LSTM)                (None, 64)                33024
_________________________________________________________________
dense (Dense)                (None, 6)                 390
=================================================================
Total params: 386,438
Trainable params: 386,438
Non-trainable params: 0
```

Embedding Layers

The model begins with an **embedding layer** which turns the input integer indices into the corresponding word vectors.

Word embedding is a way to represent a word as a vector. Word embeddings allow the value of the vector's element to be trained. After training, words with similar meanings often have the similar vectors.



https://djajafer.medium.com/multi-class-text-classification-with-keras-and-lstm-4c5525bef592