MACHINE LEARNING

SEQUENTIAL DATA PROCESSING: RECURRENT AND RECURSIVE NEURAL NETS

Sequential data processing: recurrent and recursive neural nets

Introduction to recurrent neural networks

Example: Natural language inference

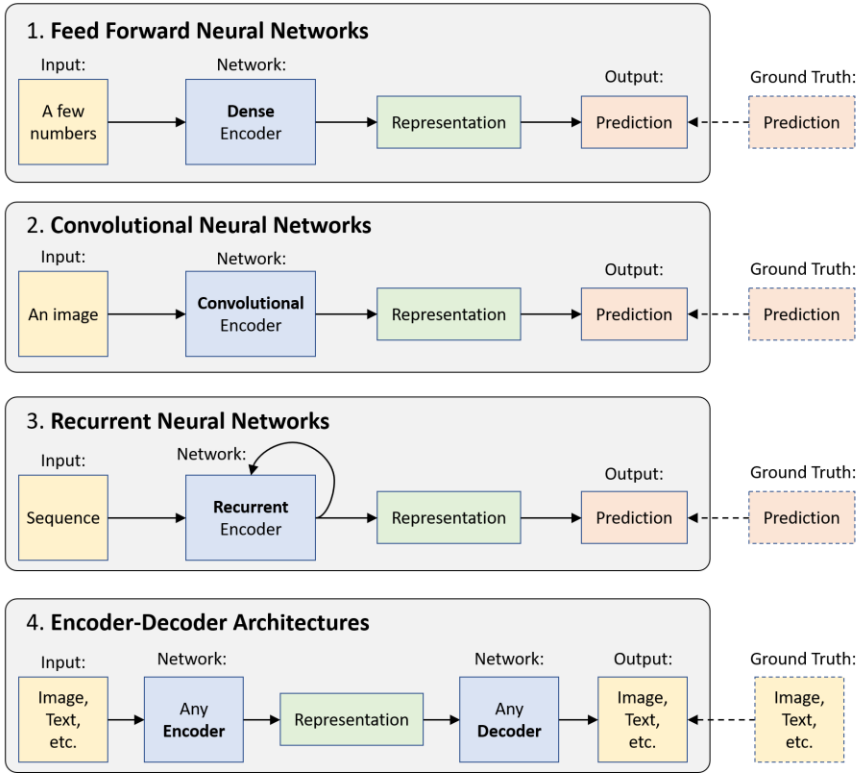
Architecture

Advanced recurrent NN

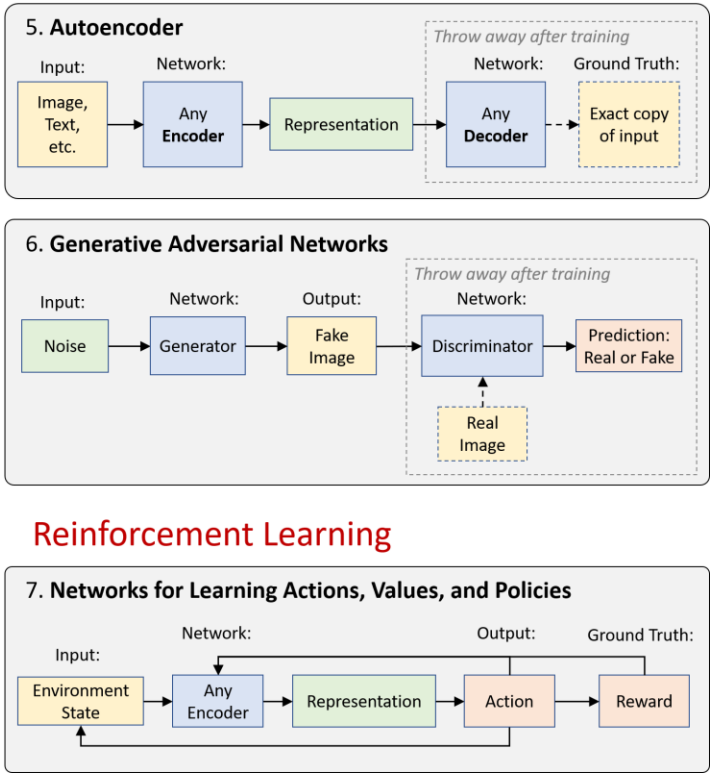
Overview

Universidad Francisco de Vitoria
UFV Madrid

Supervised Learning



Unsupervised Learning



Reinforcement Learning

RNN

- *Introduction to recurrent NN*

Sequential data processing

Universidad Francisco de Vitoria

UFV Madrid

Sequential data: dependency between points in the dataset.

Examples: natural language, time series data, music, sound.

Active areas of research:

- Natural language processing (NLP) and understanding
- Stock market prediction
- Feature Classification
- Sentiment Analysis

Sequential data

Definition: Whenever the points in the dataset are dependent on the other points in the dataset

	x	y
Speech recognition		There's a problem in the surveillance system
Music generation	Folk	
Sentiment Classification	*"This movie is a piece of shit"*	★☆☆☆☆
Language translation	*Я хачу жыць у Лондане*	*"I want to live in London"*
Video activity recognition		*running*
Feature extraction	*The Manchester Guardian* was founded in Manchester in 1821 by cotton merchant John Edward Taylor"	*"John Edward"*

Notation

Former President Donald Trump announces a White House bid for 2024

x	$x^{<1>}$	$x^{<2>}$	$x^{<3>}$	$x^{<4>}$	$x^{<t>}$				$x^{<11>}$	$T_x = 9$
y	0	0	1	1	0	0	0	0	0	
	$y^{<1>}$	$y^{<2>}$	$y^{<3>}$	$y^{<4>}$	$y^{<t>}$				$y^{<11>}$	$T_y = 9$

$x^{(i)<t>}$ → **Element t of sample i**

$y^{(i)<t>}$ → **Element t of label i**

Vector representation

Universidad Francisco de Vitoria
UFV Madrid

Former President Donald Trump announces a White House bid for 2024

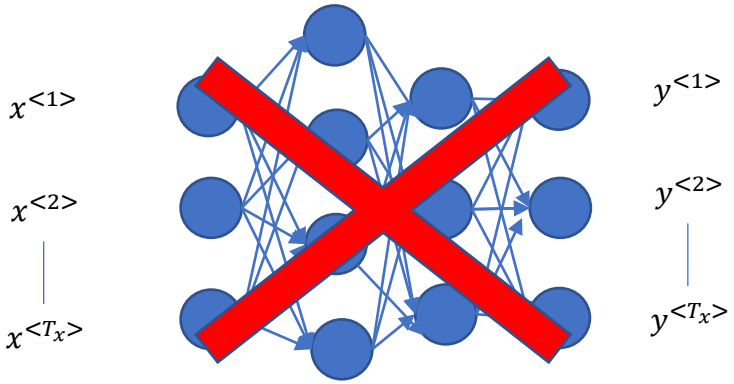
x $x^{<1>}$ $x^{<2>}$ $x^{<3>}$ $x^{<4>}$ $x^{<t>}$ $x^{<11>}$ $T_x = 9$

$$\begin{bmatrix} a \\ aaron \\ and \\ \dots \\ \dots \\ \dots \\ \dots \\ \dots \\ zulu \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dots \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \\ \dots \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dots \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dots \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \dots \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

One-hot vector

Dictionary (30.000 words)

Why not a DNN as learnt so far?



DNNs don't work in sequence models where data samples are correlated

- Sequence lengths are different, and padding doesn't work as expected
- DNN doesn't share features learnt in different positions of the text (similar problem we had with pictures)
- Sparsity of data needs to reinforce vicinity

Recurrent neural networks (RNNs)

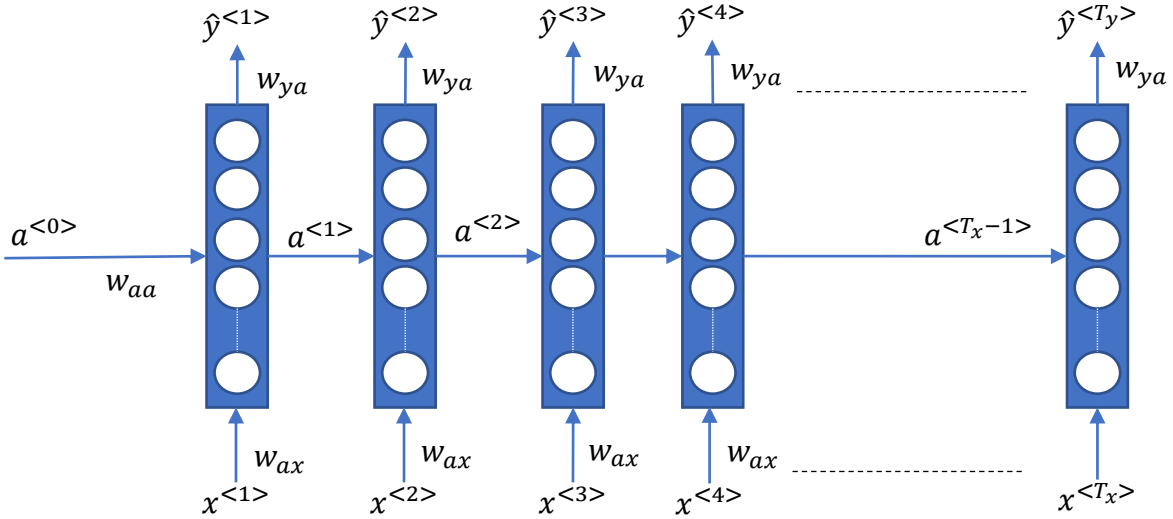
Recurrent neural networks (**RNNs**): de facto implementation for sequential data processing.

RNNs recur through the data holding the information from the previous run and try to find the meaning of the sequence, just like how humans do.

Examples of current RNNs:

- long short-term memory (LSTM)
- gated recurrent units (GRU).

Recurrent Neural Network, the origins



Former President Donald Trump 2024

$$a^{<0>} = 0$$

$$a^{<1>} = g(w_{aa}a^{<0>} + w_{ax}x^{<1>} + b_a)$$

$$\hat{y}^{<1>} = g(w_{ya}a^{<1>} + b_y)$$

The output of the n_{th} element contains a "footprint" from the n-1 layer

The n_{th} only gets informations from the previous element

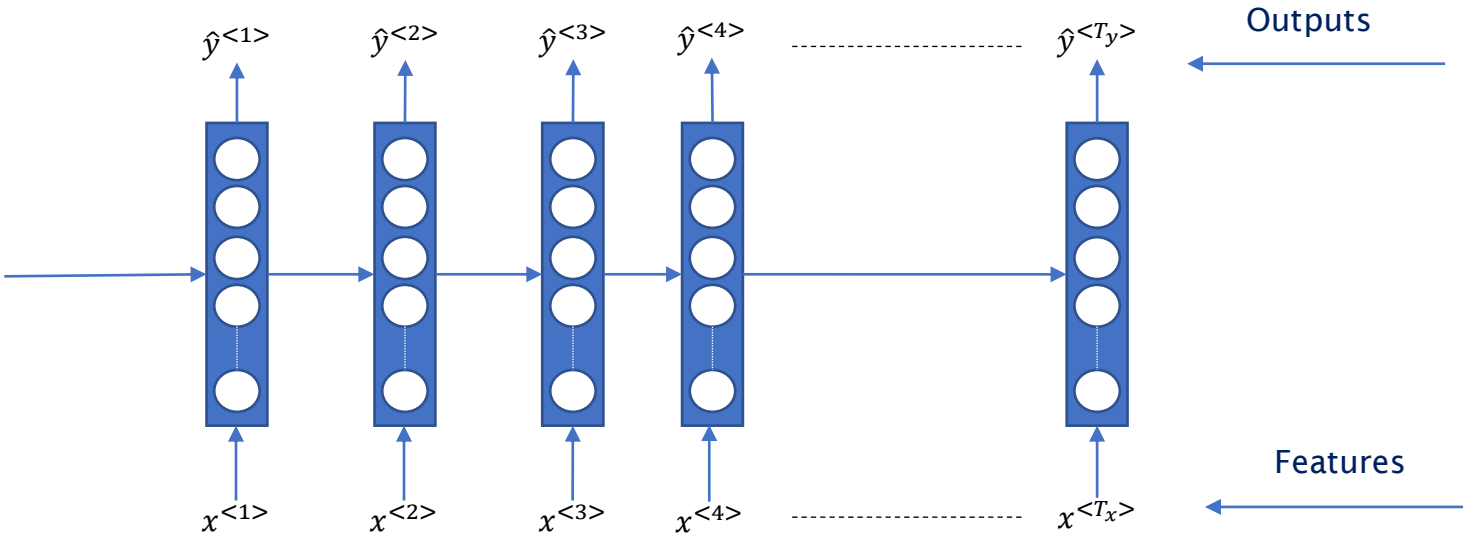
More RNN architectures

Universidad Francisco de Vitoria

UFV Madrid

Many-to-Many: many features returns many outputs

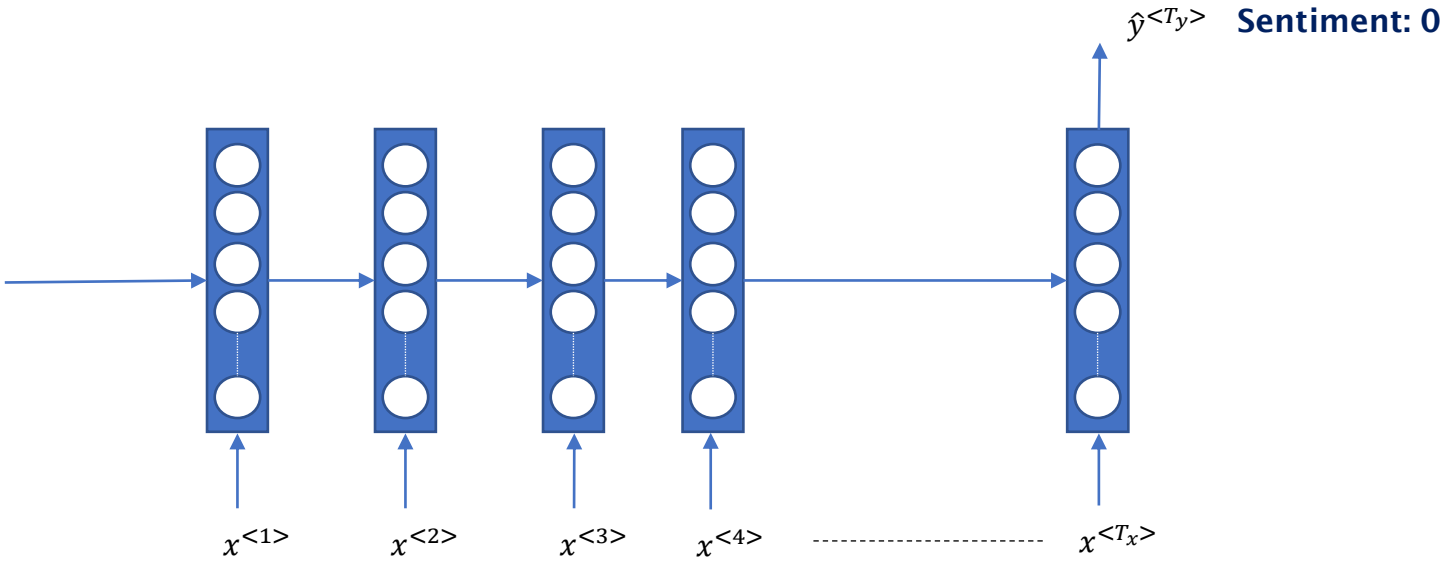
Antiguo presidente Donald Trump 2024



Former President Donald Trump 2024

More RNN architectures

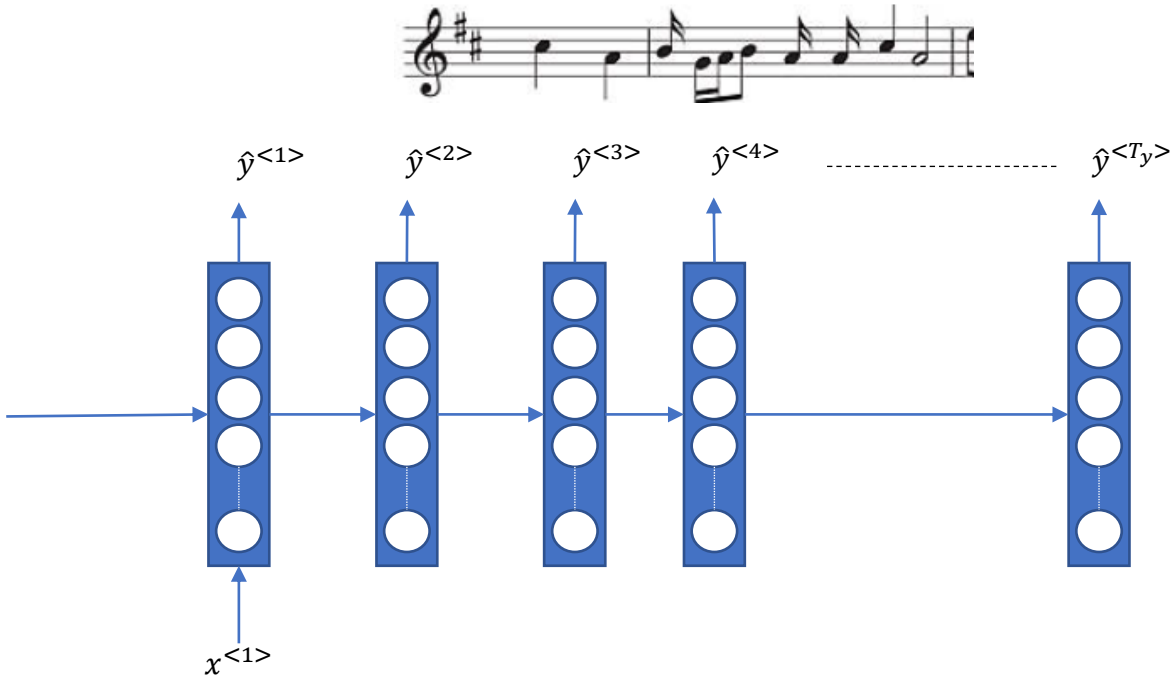
Many-to-One: many features returns one output



Former President Donald Trump 2024

More RNN architectures

One-To-Many: one feature returns many outputs

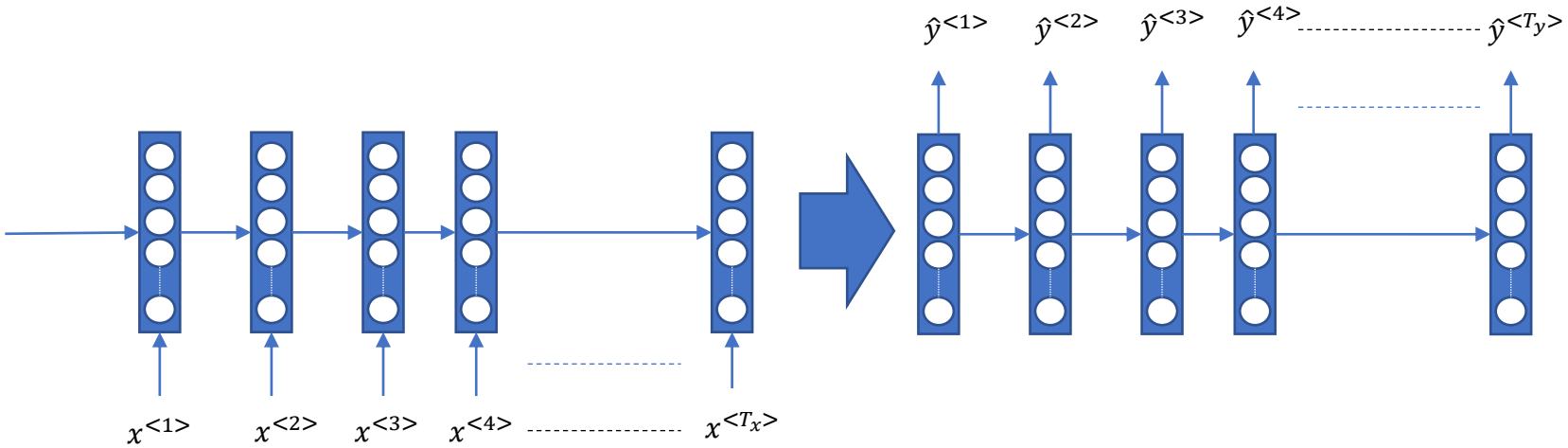


Rock n' Roll

More RNN architectures

Universidad Francisco de Vitoria
UFV Madrid

Encoding – Decoding Architecture



RNN

- *Introduction to recurrent NN*
- *Natural language inference*

Name Classifier

Universidad Francisco de Vitoria
UFV Madrid

We will build a Name Classifier

Given a name, our model will predict its origin

```
In [3]: predict('Pelaez')

> Pelaez
(-0.59) Spanish
(-2.39) French
(-2.54) Dutch
```

```
In [5]: predict('Witek')

> Witek
(-0.33) Polish
(-2.01) Czech
(-3.11) English
```

```
In [7]: predict('Putin')

> Putin
(-1.25) Russian
(-1.70) Vietnamese
(-2.21) English
```

```
In [4]: predict('Lukashenko')

> Lukashenko
(-0.34) Russian
(-1.97) Japanese
(-2.23) Polish
```

```
In [6]: predict('Nienhuis')

> Nienhuis
(-1.29) Arabic
(-1.51) Korean
(-2.24) Dutch
```

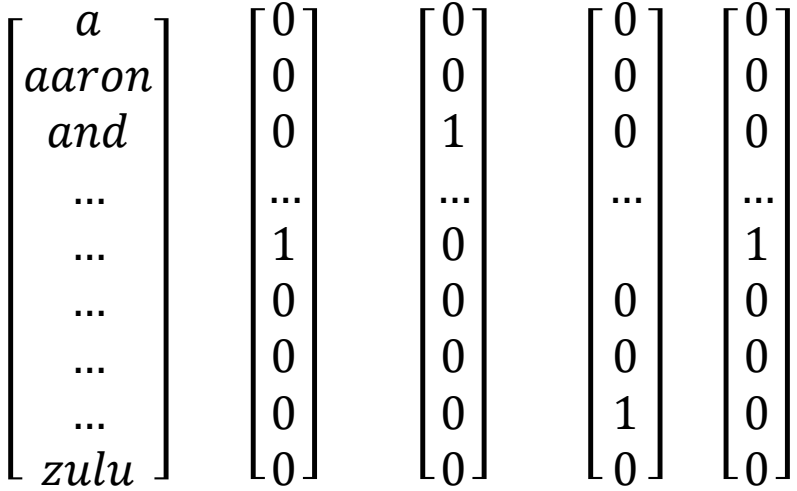
```
In [8]: predict('Biden')

> Biden
(-1.48) Dutch
(-1.70) English
(-1.73) Irish
```

Encoding words into vectors

The canonical approach to using natural language (or any sequences that consist of categorical variables) is to convert each word **to one-hot encoded vectors**.

Problems:

- **size of the vector == size of the vocabulary**
- **one new word in the dictionary requires retraining**



Dictionary (30.000 words)

Word embedding

Universidad Francisco de Vitoria

UFV Madrid

Different strategy: each unique word in the corpus is assigned a corresponding vector (real valued).

Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space.

- The key idea: words with similar meanings often appear in similar contexts. The corresponding embeddings are also trained to be similar.

Examples Tomas Mikolov's Word2vec, Stanford University's GloVe

Name Classifier

Universidad Francisco de Vitoria
UFV Madrid

We convert every name into a bunch of one-hot vectors (one per letter)

1. Set up a list of all letters (ASCII format)

```
all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)
```

```
In [13]: all_letters
Out[13]: "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ .,;'"

In [14]: n_letters
Out[14]: 57
```

2. Convert every letter into a one-hot vector
3. Concatenate all vectors

```
def lineToTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li, letter in enumerate(line):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor
```

Name Classifier

Universidad Francisco de Vitoria

UFV Madrid

‘Mar will be encoded to three vectors; each vector has 1 row and 57 columns)

```
In [23]: lineToTensor('Mar')
Out[23]:
tensor([[[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          0., 0., 0., 0., 0., 0.]],

        [[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          0., 0., 0., 0., 0., 0.]],

        [[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
          0., 0., 0., 0., 0., 0.]]])

In [24]: lineToTensor('Mar').size()
Out[24]: torch.Size([3, 1, 57])
```

CONVOLUTIONS

- *Introduction to recurrent NN*
- *Natural language inference*
- *Architecture*

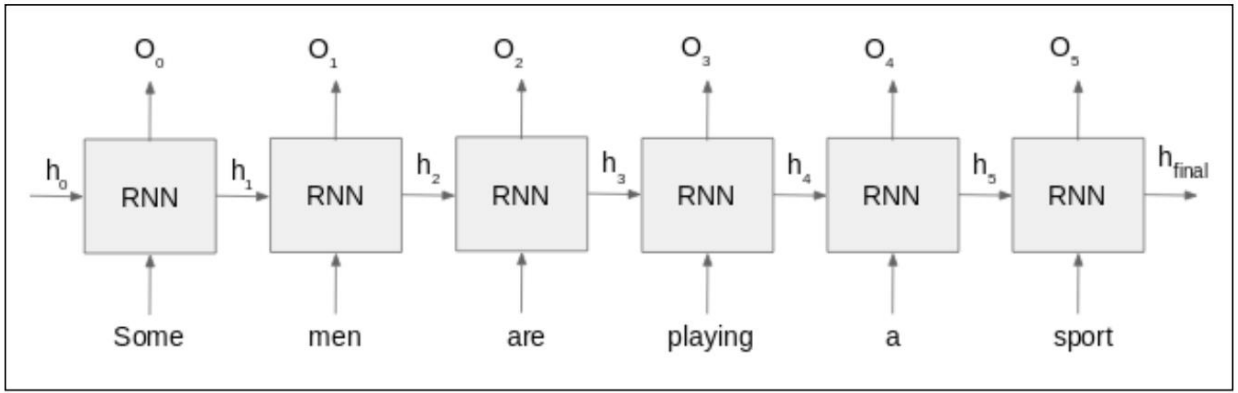
RNN Cell

RNN Cell: foundational building block of a recurrent network.

One RNN cell is capable of processing all the words (inputs) in the sentence (sequence) one by one.

Initially, we pass the first word from our sentence to the cell, which generates an output and an intermediate state.

The output can be trained to predict the next character or word in the sentence (output of the sequence)



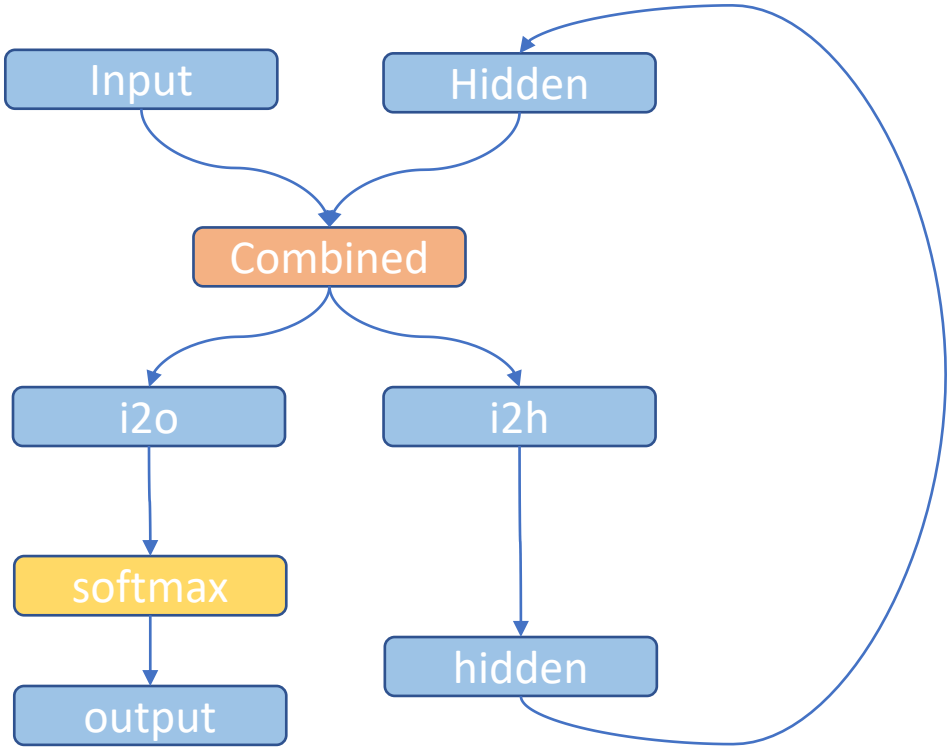
RNN Cell

This state (hidden state) is the running meaning of our sequence.

Both the output and hidden state have their own purpose.

- The output can be trained to predict the next character or word in the sentence or label in the sequence. For example, stock price prediction
- The hidden state, at the end, can capture the meaning (as in natural language inference)

Cell flow diagram



Notice we are using the same RNN cell for each word.

Reduced number of parameters

Different wiring mechanisms have been tried to design the RNN cell to get the most efficient output.

We will use the most basic one, which consists of two fully connected layers and a softmax layer.

RNNCell module


```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

torch.cat: concatenates the given sequence of seq tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension).

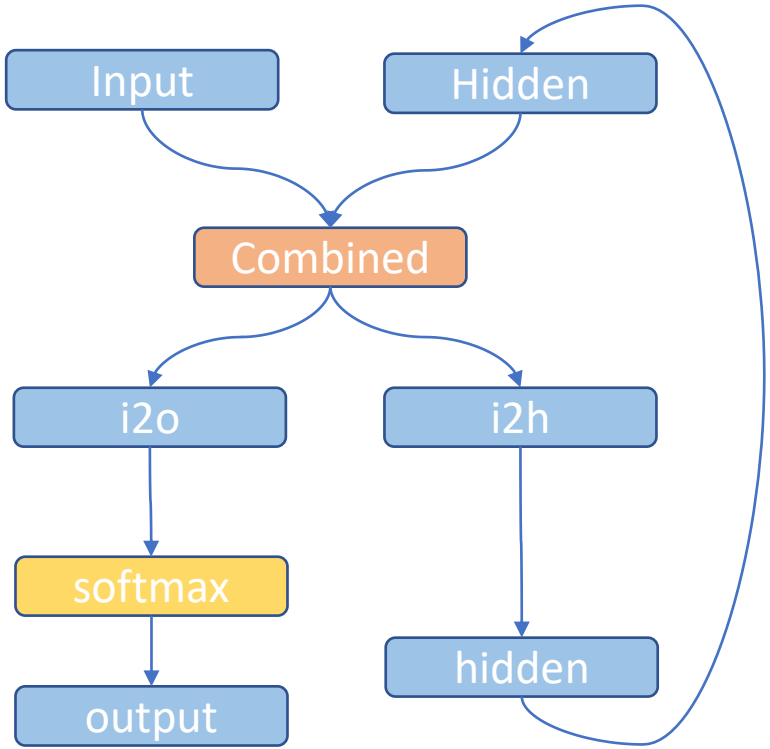
init_hidden: kept for generating the first hidden state. Must be called before starting to loop through the sequence.

RNN Cell Module

Hidden vector can be as large as you want

(new hyperparamenter)

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size

        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```



Forward Pass

Universidad Francisco de Vitoria

UFV Madrid

The model admits one vector at once

```
In [35]: input = lineToTensor('Mar')
    ...: hidden = torch.zeros(1, n_hidden)
    ...:
    ...: output, next_hidden = rnn(input[0], hidden)
    ...: print(output)
tensor([[-3.0372, -2.7794, -2.7529, -2.9034, -2.4730, -2.7569, -2.6195, -3.8863,
         -3.0832, -3.5388, -3.1988, -2.2534, -3.0925, -3.4337, -2.9468, -3.0830,
         -3.0711, -2.4405]], grad_fn=<LogSoftmaxBackward0>)

In [36]: input = lineToTensor('Mar')
    ...: hidden = torch.zeros(1, n_hidden)
    ...:
    ...: output, next_hidden = rnn(input[1], hidden)
    ...: print(output)
tensor([[-2.9724, -3.5310, -1.9685, -4.1221, -3.9900, -4.3898, -4.2428, -4.4723,
         -3.9834, -2.9250, -2.1964, -2.9757, -3.0389, -2.0915, -4.1408, -4.4784,
         -1.5073, -2.9506]], grad_fn=<LogSoftmaxBackward0>)

In [37]: input = lineToTensor('Mar')
    ...: hidden = torch.zeros(1, n_hidden)
    ...:
    ...: output, next_hidden = rnn(input[2], hidden)
    ...: print(output)
tensor([[-2.7616, -3.4449, -2.5595, -2.7443, -2.4087, -2.4313, -1.2580, -4.8432,
         -3.7138, -4.2340, -4.0958, -2.9873, -4.2377, -4.1996, -3.8172, -2.5261,
         -4.0603, -3.2513]], grad_fn=<LogSoftmaxBackward0>)
```

```
In [39]: for num,value in enumerate(all_categories):
    ...:     print(num,value)
    ...:
0 Arabic
1 Chinese
2 Czech
3 Dutch
4 English
5 French
6 German
7 Greek
8 Irish
9 Italian
10 Japanese
11 Korean
12 Polish
13 Portuguese
14 Russian
15 Scottish
16 Spanish
17 Vietnamese
```

For every single letter, the model returns the likelihood of being in one category, the highest the better

Training the model with one name

The model admits one vector at once

```
def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning rate
    for p in rnn.parameters():
        p.data.add_(p.grad.data, alpha=-learning_rate)

    return output, loss.item()
```

1. We feed the model with all the letters of a given name one by one
2. The model calculate the loss for the resulting output compared with the ground truth (category tensor)
3. Backward pass…
4. Optimization Algorithm

Loss Function: Negative Logarithmic Likelihood

Universidad Francisco de Vitoria

UFV Madrid

```
criterion = nn.NLLLoss()
```

Negative log likelihood loss

NLLLOSS

CLASS torch.nn.NLLLoss(*weight=None, size_average=None, ignore_index=- 100, reduce=None, reduction='mean'*) [SOURCE]

The negative log likelihood loss. It is useful to train a classification problem with *C* classes.

If provided, the optional argument `weight` should be a 1D Tensor assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* given through a forward call is expected to contain log-probabilities of each class. *input* has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case. The latter is useful for higher dimension inputs, such as computing NLL loss per-pixel for 2D images.

Obtaining log-probabilities in a neural network is easily achieved by adding a *LogSoftmax* layer in the last layer of your network. You may use *CrossEntropyLoss* instead, if you prefer not to add an extra layer.

The *target* that this loss expects should be a class index in the range $[0, C-1]$ where *C = number of classes*; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range).

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot 1\{c \neq \text{ignore\_index}\},$$

where x is the input, y is the target, w is the weight, and N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} l_n, & \text{if reduction} = \text{'mean'}; \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'}. \end{cases}$$

Training for more than one Name

We have trained the model with just one name

We should choose randomly more names and train the model with them

```
n_iters = 100000
print_every = 5000
plot_every = 1000


for iter in range(1, n_iters + 1):
    category, line, category_tensor, line_tensor = randomTrainingExample()
    output, loss = train(category_tensor, line_tensor)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100, timeSince(start), loss, line, guess, correct))

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses.append(current_loss / plot_every)
        current_loss = 0
```

We use the randomTrainingExample() helper to randomly choose a Name

The helper function...

randomTrainingExample() provides us all the necessary to train the model with ONE name

```
def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    category_tensor = torch.tensor([all_categories.index(category)], dtype=torch.long)
    line_tensor = lineToTensor(line)
    return category, line, category_tensor, line_tensor
```

```
IPdb [7]: category
Out  [7]: 'Dutch'
```

```
line
'Slootmaekers'
```

```
IPdb [11]: category_tensor
Out  [11]: tensor([3])
```

```
IPdb [10]: all_categories
Out  [10]: ['Arabic', 'Chinese', 'Czech', 'Dutch', 'English', 'French', 'German', 'Greek',
'Irish', 'Italian', 'Japanese', 'Korean', 'Polish', 'Portuguese', 'Russian', 'Scottish',
'Spanish', 'Vietnamese']
```

How the model runs

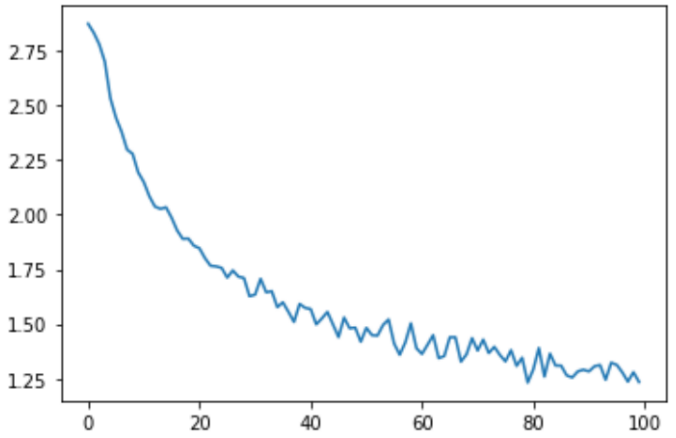
Universidad Francisco de Vitoria

UFV Madrid

We've shown 100.000 names to the model.

```
5000 5% (3m 35s) 2.5979 Mudra / Japanese ✗ (Czech)
10000 10% (3m 46s) 1.9801 Renov / Russian ✓
15000 15% (3m 58s) 1.7048 Couture / French ✓
20000 20% (4m 10s) 2.4612 Palzewicz / German ✗ (Czech)
25000 25% (4m 22s) 0.8419 Kokoris / Greek ✓
30000 30% (4m 34s) 1.9458 Donnell / Scottish ✗ (Irish)
35000 35% (4m 46s) 1.3227 Germano / Italian ✓
40000 40% (4m 58s) 0.7845 Yi / Korean ✓
45000 45% (5m 10s) 2.2371 Lim  / Japanese ✗ (Korean)
50000 50% (5m 22s) 1.8893 Kasai / Arabic ✗ (Japanese)
55000 55% (5m 35s) 3.9308 Komon / Dutch ✗ (Japanese)
60000 60% (5m 49s) 2.8555 Dale / Irish ✗ (Dutch)
65000 65% (6m 2s) 1.1437 Mateus / Arabic ✗ (Portuguese)
70000 70% (6m 15s) 1.1443 Pinho / Portuguese ✓
75000 75% (6m 28s) 0.5560 Amari / Arabic ✓
80000 80% (6m 41s) 2.4466 Bazzi / Polish ✗ (Arabic)
85000 85% (6m 54s) 0.3413 Arakawa / Japanese ✓
90000 90% (7m 7s) 0.2252 Vuu / Vietnamese ✓
95000 95% (7m 19s) 1.3841 Yun / Chinese ✗ (Korean)
100000 100% (7m 33s) 0.1946 Vinh / Vietnamese ✓
```

Loss chart shows a learning curve



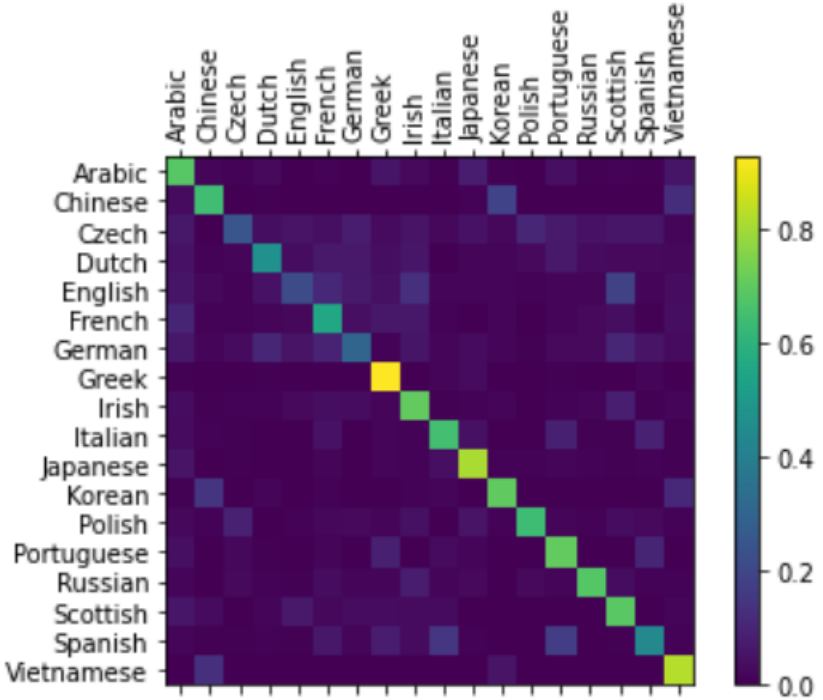
Confusion matrix

- To see how well the network performs on different categories,
- We will create a confusion matrix, indicating for every actual language (rows) which language the network guesses (columns).
- To calculate the confusion matrix, a bunch of samples are run through the network with evaluate()
- evaluate() is the same as train() minus the backprop.

```
IPdb [15]: confusion.shape
Out  [15]: torch.Size([18, 18])
```

```
# Keep track of correct guesses in a confusion matrix
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 10000

# Just return an output given a line
def evaluate(line_tensor):
    hidden = rnn.initHidden()

    for i in range(line_tensor.size()[0]):
        output, hidden = rnn(line_tensor[i], hidden)

    return output

# Go through a bunch of examples and record which are correctly guessed
for i in range(n_confusion):
    category, line, category_tensor, line_tensor = randomTrainingExample(
    output = evaluate(line_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = all_categories.index(category)
    confusion[category_i][guess_i] += 1

# Normalize by dividing every row by its sum
for i in range(n_categories):
    confusion[i] = confusion[i] / confusion[i].sum()
```

Model Evaluation



- It seems that our model:
- Works nices for Greek names
- Works also nice for Spanish, Chinese and Arabic names
- Works really bad for English names

Recurrent Neural Networks

- *Introduction to recurrent NN*
- *Natural language inference*
- *Architecture*
- *Advanced recurrent NN*

Advanced RNN

State-of-the-art models are developments/variations of two advanced RNN designs:

- long short-term memory (LSTM)
- gated recurrent units (GRU).

These architectures are generalized as gated recurrent networks

- they have gates for handling the flow of inputs/gradients through the network.

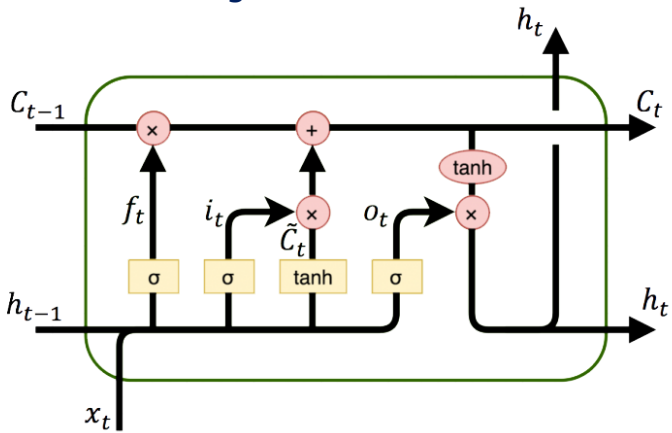
Gates are layers with activations, such as sigmoid, that are combined to decide the amount of data to flow through (to the state and the output)

Another development is attention which helps the network to focus on an important part of the input instead of searching the whole input and trying to find the answer (out of our scope)

LSTM, basic concept

Four layers that interact with one another in a way to produce the output of that cell (hidden state)

along with the cell state.

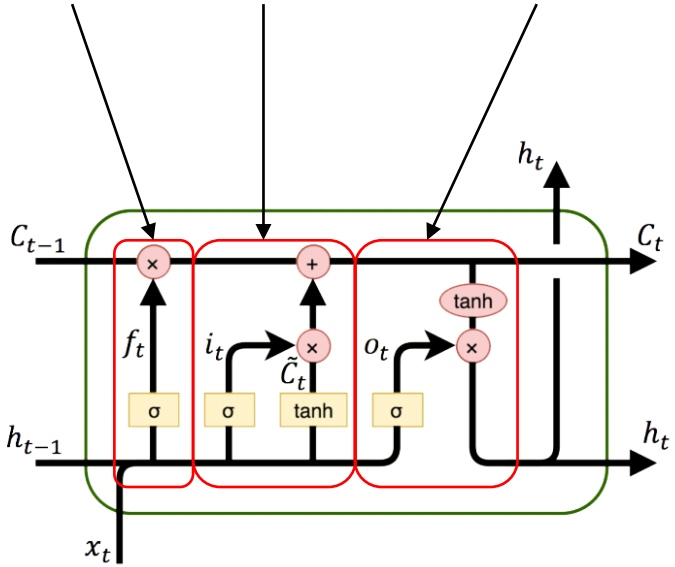


Both **hidden state** and **cell state** are passed to the recursion.

Each LSTM cell has three inputs $h_{\{t-1\}}$,$C_{\{t-1\}}$ and x_t and two outputs h_t and C_t . For a given time t, h_t is the hidden state, C_t is the cell state or memory, x_t is the current data point or input.

Inside the LSTM cell

A typical LSTM **cell** has a **forget gate, input gate**, an **output gate**.



The cell remembers values over arbitrary time intervals and the three *gates* regulate the flow of information into and out of the cell.

LSTM: forget gate

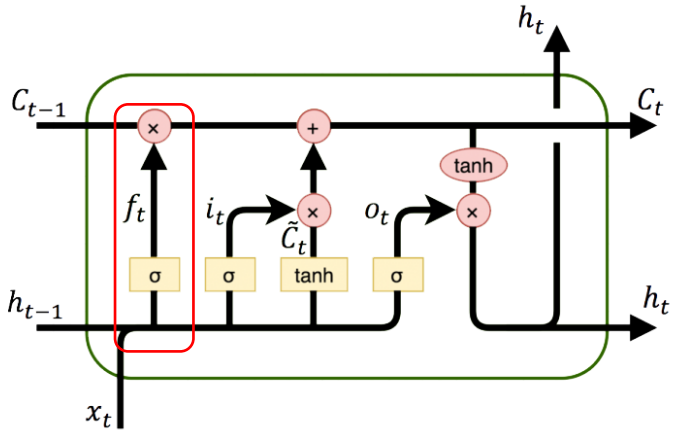
The first sigmoid layer has two inputs–$h_{\{t-1\}}$ and x_t where $h_{\{t-1\}}$ is the hidden state of the previous cell.

It is known as the forget gate as its output selects the amount of information of the previous cell to be included.

$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$

The output is a number in [0,1] which is multiplied (point-wise) with the previous cell state $C_{\{t-1\}}$.

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$



- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0,1)^h$: forget gate's activation vector
- $i_t \in (0,1)^h$: input/update gate's activation vector
- $o_t \in (0,1)^h$: output gate's activation vector
- $h_t \in (-1,1)^h$: hidden state vector , output
- $\tilde{c}_t \in (-1,1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices

LSTM: information gate

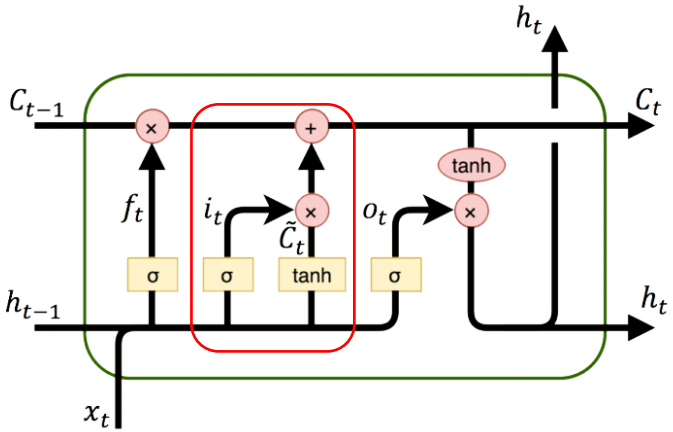
The second sigmoid layer is the input gate that decides what new information is to be added to the cell.

It takes two inputs $h_{\{t-1\}}$ and x_t. There is an information vector.

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$

and a *tanh* layer that creates a vector C_t of the new candidate values.

$$\tilde{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$



- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0,1)^h$: forget gate's activation vector
- $i_t \in (0,1)^h$: input/update gate's activation vector
- $o_t \in (0,1)^h$: output gate's activation vector
- $h_t \in (-1,1)^h$: hidden state vector , output
- $\tilde{c}_t \in (-1,1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices

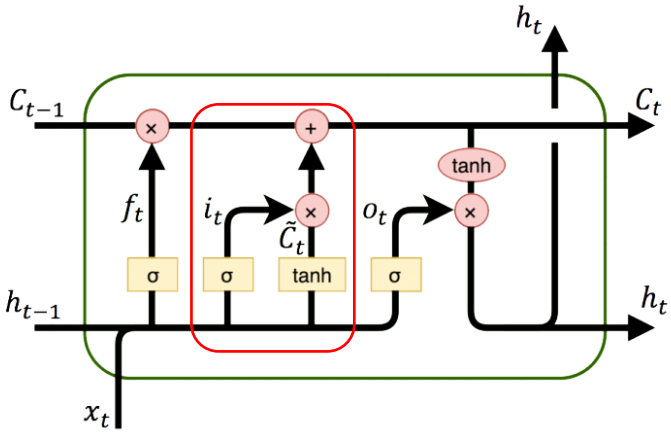
LSTM: information gate

Their point-wise multiplication ($i_t \times C_t$) tells us the amount of information to be added to the cell state.

$$f_t \circ c_{t-1}$$

The result is then added with the result of the forget gate multiplied with previous cell state ($f_t \circ C_{\{t-1\}}$) to produce the current cell state C_t . ° <; −{)}

$$c_t = f_t \circ c_{t-1} + i_t \circ \tilde{c}_t$$



- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0,1)^h$: forget gate's activation vector
- $i_t \in (0,1)^h$: input/update gate's activation vector
- $o_t \in (0,1)^h$: output gate's activation vector
- $h_t \in (-1,1)^h$: hidden state vector also known as out
- $\tilde{c}_t \in (-1,1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices

LSTM: output gate

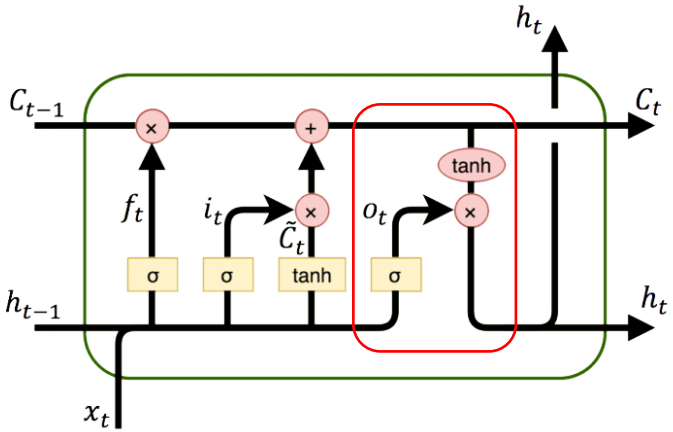
Next, the output gate activation vector is calculated with a sigmoid activation function

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$

and output of the cell h_t is calculated using o_t and a tanh layer on c_t.

$$h_t = o_t \circ \sigma_h(c_t)$$

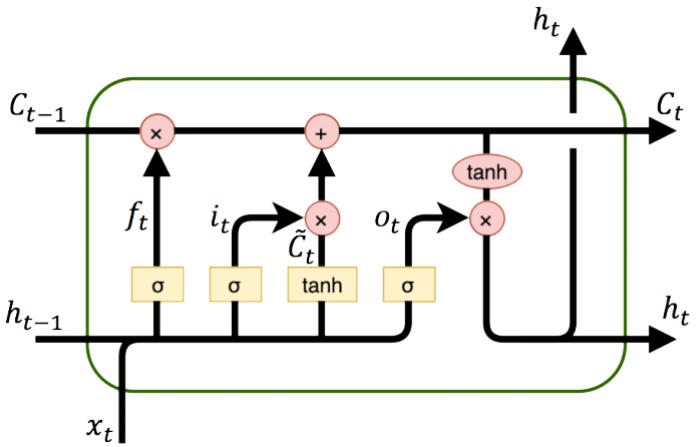
The sigmoid layer o_t decides which part of the cell state will be present in the output whereas tanh layer shifts the output in the range of [-1,1]. The results of the two layers undergo point-wise multiplication to produce the output h_t of the cell.



- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0,1)^h$: forget gate's activation vector
- $i_t \in (0,1)^h$: input/update gate's activation vector
- $o_t \in (0,1)^h$: output gate's activation vector
- $h_t \in (-1,1)^h$: hidden state vector , output
- $\tilde{c}_t \in (-1,1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices

LSTM, summary

$$
\begin{aligned}
f_t &= \sigma_g(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma_g(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma_g(W_o x_t + U_o h_{t-1} + b_o) \\
\tilde{c}_t &= \sigma_c(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \sigma_h(c_t)
\end{aligned}
$$



- $x_t \in \mathbb{R}^d$: input vector to the LSTM unit
- $f_t \in (0,1)^h$: forget gate's activation vector
- $i_t \in (0,1)^h$: input/update gate's activation vector
- $o_t \in (0,1)^h$: output gate's activation vector
- $h_t \in (-1,1)^h$: hidden state vector also known as output vector of the LSTM unit
- $\tilde{c}_t \in (-1,1)^h$: cell input activation vector
- $c_t \in \mathbb{R}^h$: cell state vector
- $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$: weight matrices and bias vector parameters which need to be learned during training

Architecture in PyTorch

Architecture, like our “simple” RNN, but RNNCell is replaced with LSTM or GRU cells.

PyTorch has functional APIs available for using an LSTM cell or GRU cell as the smallest unit of the recurrent network.

As seen earlier dynamic graph capability, looping through the sequence and calling the cell is completely possible with PyTorch.

Using the high-level API

While PyTorch gives access to granular LSTMCell and GRUCell APIs, the torch.nn module has higher-level APIs for LSTM and GRU nets, which wrap LSTMCell and GRUCell and implement the efficient execution using **cuDNN** (**CUDA Deep Neural Network**) LSTMs and cuDNN GRUs.

LSTMs and GRUs accept more parameters than the cell counterpart:

- num_layers
- dropout
- bidirectional

Using higher- level APIs like LSTM eliminates the need for a Python loop and accepts the complete sequence at a time as input.

Using the high-level API

While PyTorch gives access to granular LSTMCell and GRUCell APIs, the torch.nn module has higher-level APIs for LSTM and GRU nets, which wrap LSTMCell and GRUCell and implement the efficient execution using **cuDNN** (**CUDA Deep Neural Network**) LSTMs and cuDNN GRUs.

LSTMs and GRUs accept more parameters than the cell counterpart:

- num_layers
- dropout
- bidirectional

Using higher- level APIs like LSTM eliminates the need for a Python loop and accepts the complete sequence at a time as input.

LSTM example

Universidad Francisco de Vitoria
UFV Madrid

```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM, self).__init__()

        self.hidden_size = hidden_size

        self.lstm = nn.LSTM(input_size = input_size + hidden_size, hidden_size = hidden_size)
        self.i2o  = nn.Linear(hidden_size,output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        output = self.lstm(combined)
        output = self.i2o(output[0])
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

LSTM Cell

Linear Regression

Activation

LSTM Example, confusion matrix