Recurrent Neural Networks

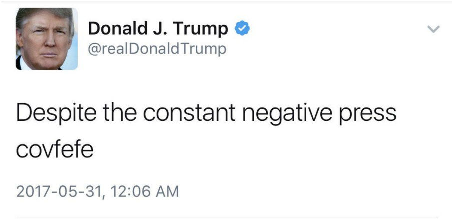
Sequential modeling

ε.δε.μ^2

ΕΠΙΣΤΗΜΗ ΔΕΔΟΜΕΝΩΝ & ΜΗΧΑΝΙΚΗ ΜΑΘΗΣΗ

Sequential modeling

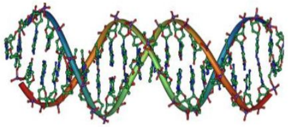
Sequential data: Text, Speech, Music, Movies, Stock prices, DNA, Earthquakes, ...

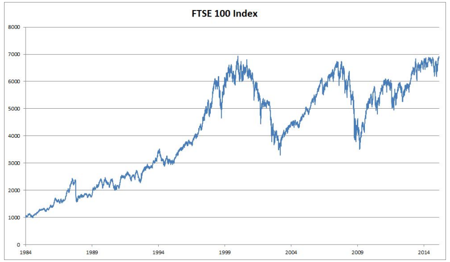


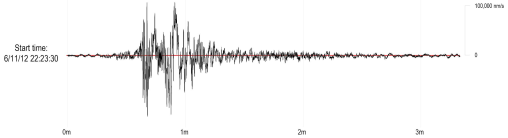
Donald J. Trump
@realDonaldTrump

Despite the constant negative press covfefe

2017-05-31, 12:06 AM







Sequence order is important:

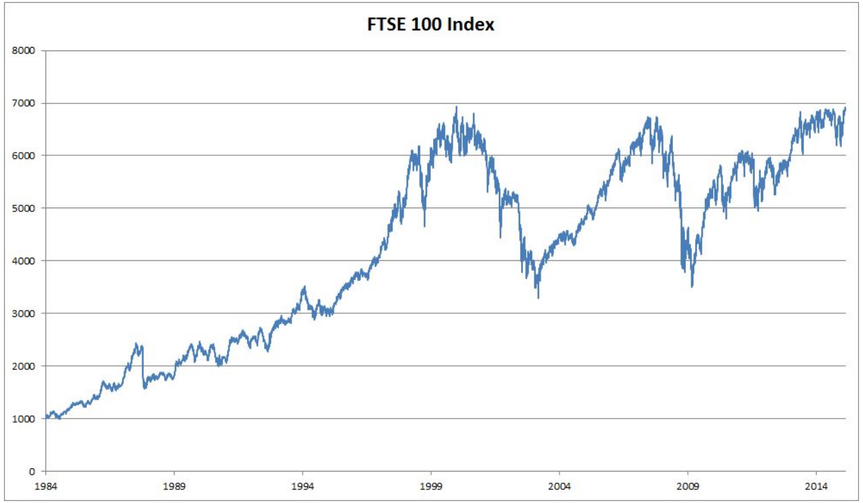
- overall it defines the meaning of the data
- each event in the sequence depends on the previous events

Sequential modeling: prediction

prices $x_t \geq 0$ at time t

For a trader to do well in the stock market on day t he should want to predict x_t via $x_t \sim p(x_t | x_{t-1}, \ldots x_1)$.

Problem: the number of inputs, $x_{t-1}, \ldots x_1$ increases with the amount of data that we encounter



We will need an approximation to make this computationally tractable:

- Autoregressive models: autoregressive models & latent autoregressive models
- Hidden Markov Models

Sequential modeling: Autoregressive models

Two strategies:

Autoregressive (αυτοπαλινδρόμηση): use only T observations x_t~$p(x_t|x_{t-1},\ldots x_{t-T})$. The number of arguments is always the same. Perform regression on themselves. For discrete objects such as words we use a classifier rather than a regressor.

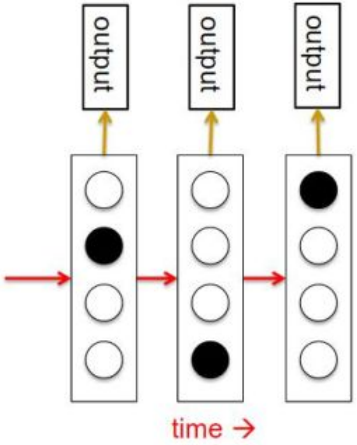
Latent Autoregressive Models: keep some summary h_t of the past observations around and update that in addition to the actual prediction. Estimate $x_t|x_{t-1},h_{t-1}$ and moreover update $h_t=g(h_t,x_t)$. Neural Networks with memoryful hidden layers are LAMs. Moreover the “summary” is the hidden layer status.

Sequential modeling: Hidden Markov Models

Whenever the approximation $p(x_t|x_{t-1},\dots x_{t\text{-}T}) = p(x_t|x_{t-1},\dots x_1)$ is accurate we say that the sequence satisfies a Markov condition. For T=1 we have a first order Markov model.

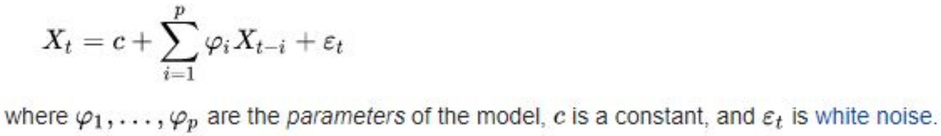
$$p(x_1,\dots x_T) = \prod_{t=1}^{T} p(x_t|x_{t-1}).$$

- Have a discrete one-of-N hidden state.
- Transitions between states are stochastic and controlled by a transition matrix.
- The outputs produced by a state are stochastic.
- Memoryful models, time-cost to infer the hidden state distribution.



Sequential modeling: Autoregressive models

Vanilla AR: the output variable depends linearly on its own previous values and on a stochastic term

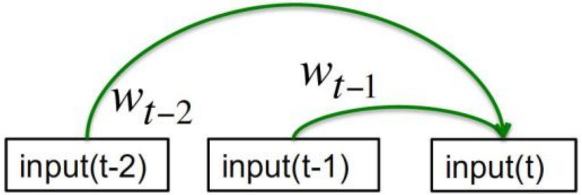


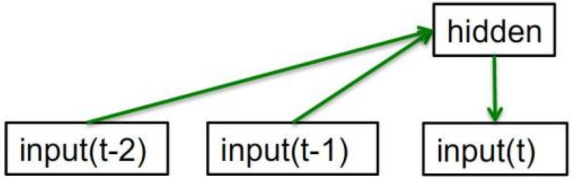
$$X_t = c + \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t$$

where $\varphi_1, \ldots, \varphi_p$ are the *parameters* of the model, c is a constant, and ε_t is white noise.

(Deep) Feed-forward NN AR extension

These generalize AR models by using one or more layers of non-linear hidden units. Memoryless models: limited word-memory window; hidden state cannot be used efficiently

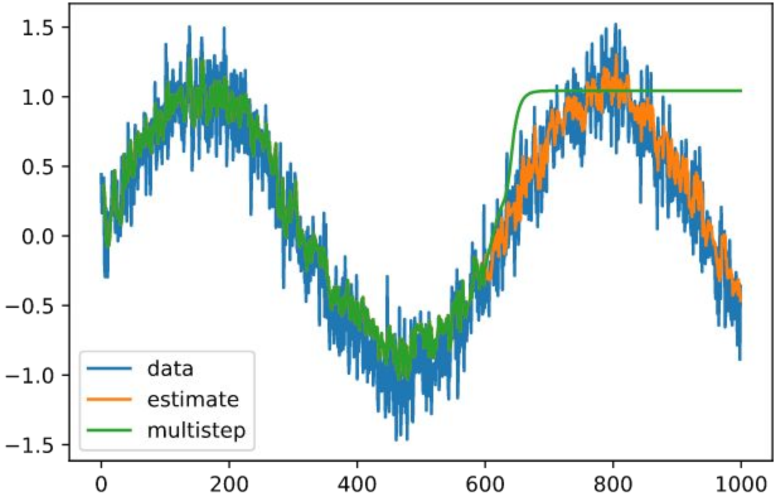


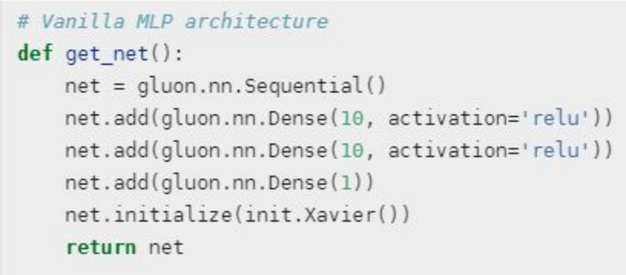


Sequential modeling: Autoregressive FF models

prediction with embedding = 4

$$\begin{aligned} x_{601} &= f(x_{600}, \ldots, x_{597}) \\ x_{602} &= f(x_{601}, \ldots, x_{598}) \\ x_{603} &= f(x_{602}, \ldots, x_{599}) \end{aligned}$$



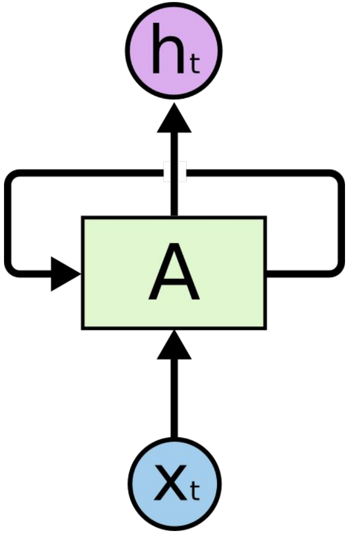


```
# Vanilla MLP architecture
def get_net():
    net = gluon.nn.Sequential()
    net.add(gluon.nn.Dense(10, activation='relu'))
    net.add(gluon.nn.Dense(10, activation='relu'))
    net.add(gluon.nn.Dense(1))
    net.initialize(init.Xavier())
    return net
```

Poor prediction performance. Deep architecture does not help

Reason: we use predictions instead of real data and errors build up

Recurrent Neural Networks



Recurrent NN have hidden layers with loops

This loop “summarizes” the past (memoryful)

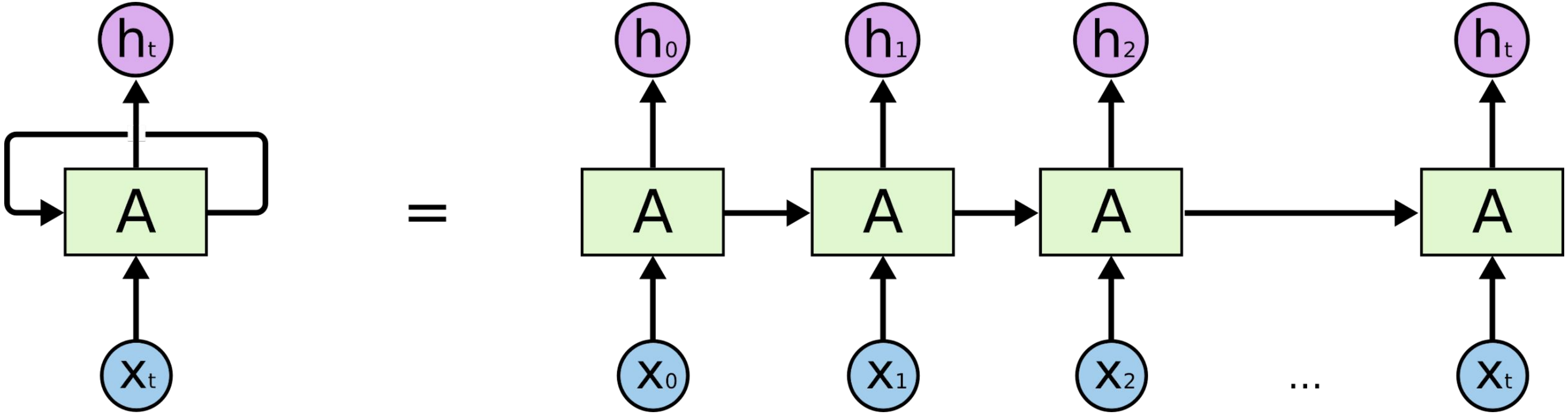
Recurrent -> Ανατροφοδοτούμενα ή Επαναληπτικά

Recurrent NN ≠ Recursive (Αναδρομικά) NN.

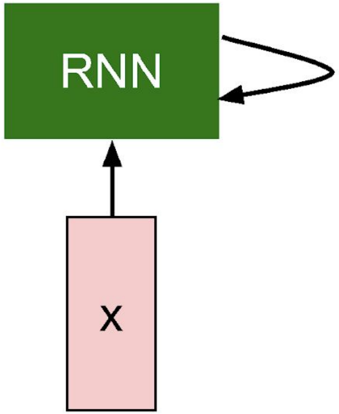
A recursive neural network is more like a hierarchical network where there is really no time aspect to the input sequence but the input has to be processed hierarchically in a tree fashion.

Recurrent Neural Networks

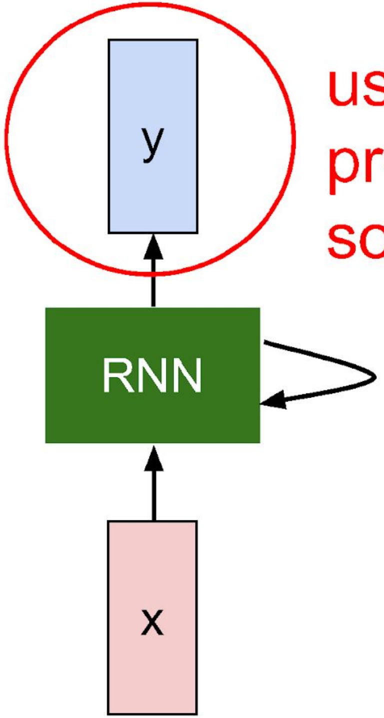
An unrolled Recurrent Neural Network



Recurrent Neural Network



Recurrent Neural Network



usually want to predict a vector at some time steps

Recurrent Neural Network

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

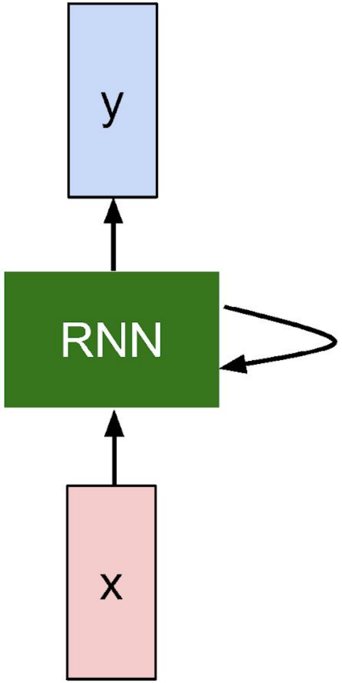
$$\boxed{h_t} = \boxed{f_W}(\boxed{h_{t-1}}, \boxed{x_t})$$

new state

old state

input vector at some time step

some function with parameters W

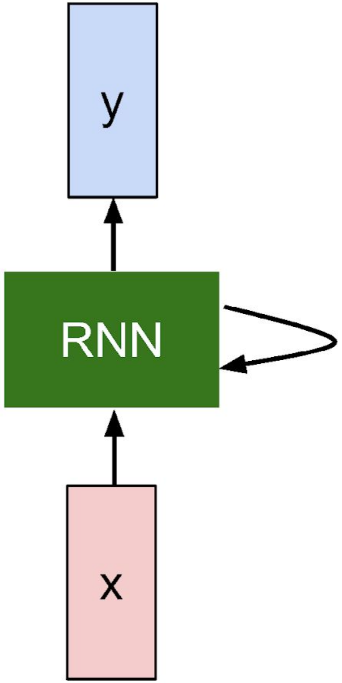


Recurrent Neural Network

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

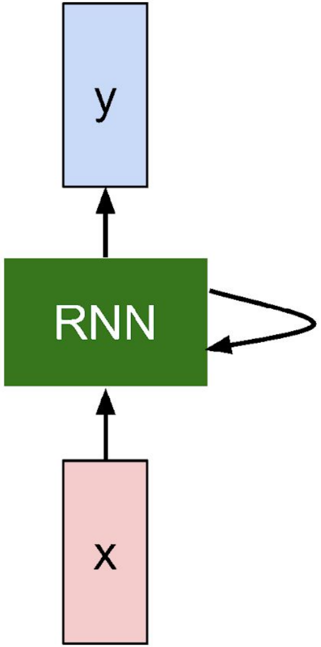
$$h_t = f_W(h_{t-1}, x_t)$$

Notice: the same function and the same set of parameters are used at every time step.



(Vanilla) Recurrent Neural Network

The state consists of a single *“hidden”* vector **h**:



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

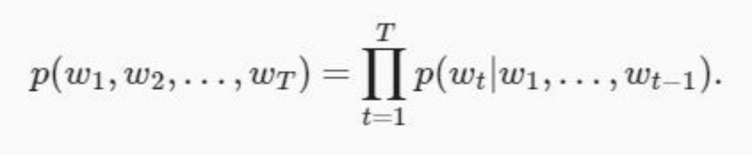
$$\downarrow$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Language Modeling

Assuming the words in a text of length T are in turn $w_1, w_2, \ldots, w_T$. Given such a sequence, the goal of a language model is to estimate the probability $p(w_1, w_2, \ldots, w_T)$.



$$p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t | w_1, \ldots, w_{t-1}).$$

p(Statistics,is,fun,.)=p(Statistics)p(is|Statistics)p(fun|Statistics,is)p(.|Statistics,is,fun).

Very useful

- Generate meaningful text. Draw $w_t \sim p(w_t | w_{t-1}, \ldots w_1)$
- Speech-to-text translation. Evaluate ‘to recognize speech’ vs ‘to wreck a nice beach’
- Document summarization. Evaluate ‘dog bites man’ vs ‘man bites dog’,
- Improve the infamous autocorrector!

Language Modeling

Estimate probabilities using large corpora (e.g. Wikipedia, Project Gutenberg etc.) and keep count of all occurrences and co-occurrences (n-grams)

$$\hat{p}(\text{is}|\text{Statistics}) = \frac{n(\text{Statistics is})}{n(\text{Statistics})}.$$

Many problems

- must store all frequencies
- longer n-grams are much less frequent and probabilities tend to zero. Long word sequences are almost certain to be novel (0 probability)
- we ignore the meaning of words: ‘cookie’ and ‘biscuit’ should occur in related contexts
- Deep learning methods can be good at language modeling

Text preprocessing

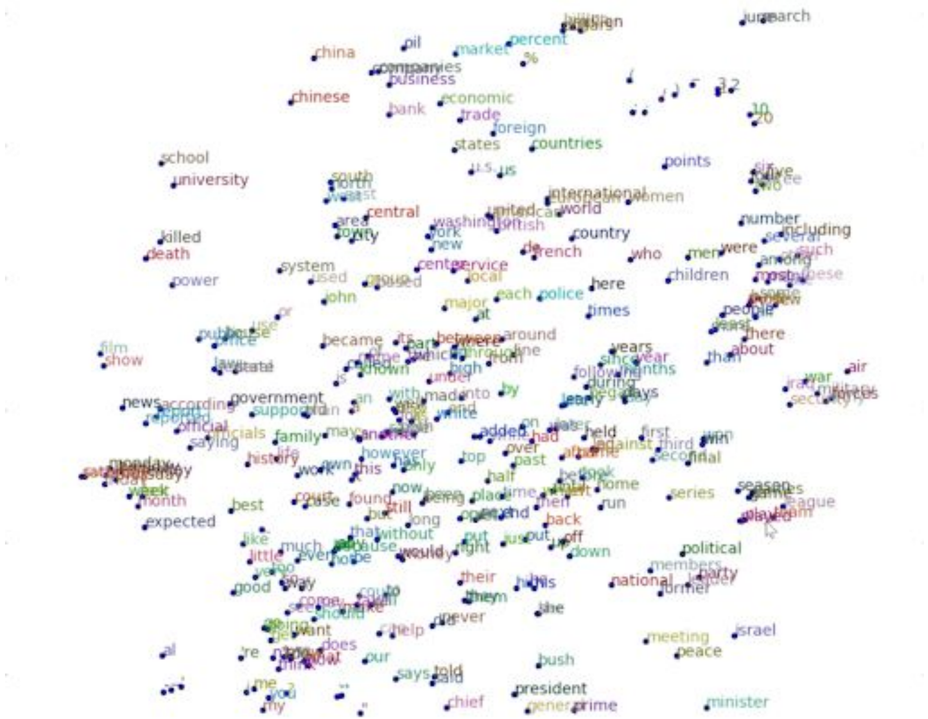
1. Tokenization. “He left!” should be 3 tokens: “He”, “left”, “!” (NLTK tokenizer)
2. Remove infrequent words. For example keep the 7997 most frequent words of the corpus. Reserve one special token UNKNOWN_TOKEN for the rest of words
3. Add special start and end tokens. Prepend a special SENTENCE_START token, and append a special SENTENCE_END token to each sentence (total 8000 tokens)
4. Use One-hot encoding for each sentence

	the	sky	is	blue
the	1	0	0	0
sky	0	1	0	0
is	0	0	1	0
blue	0	0	0	1

Better: use Word2Vec embeddings

* special tokens will be included in the encoding

Κάτι πιο εκφραστικό: Word2Vec embeddings



Traditional n-gram language model

Limitation 1): curse of dimensionality

- Example

- train a 10-gram LM on a corpus of 100.000 unique words
- space: 10-dimensional hypercube where each dimension has 100.000 slots
- model training ↔ assigning a probability to each of the 100.000^{10} slots
- **probability mass vanishes** → more data is needed to fill the huge space
- the more data, the more unique words! → vicious circle
- what about corpuses of 10^6 unique words?

- → in practice, contexts are typically limited to size 2 (trigram model)
  e.g., famous Katz (1987) smoothed trigram model

- → such short context length is a limitation: a lot of information is not captured

Traditional n-gram language model

Limitation 2): word similarity ignorance

- We should assign similar probabilities to `Obama speaks to the media in Illinois` and `the President addresses the press in Chicago`

- This does not happen because of the “one-hot” vector space representation:

$$\left.\begin{array}{r} \text{obama} = [0\ 0\ 0\ 0\ \ \dots\ \ 0\ 1\ 0\ 0] \\ \text{president} = [0\ 0\ 0\ 1\ \ \dots\ \ 0\ 0\ 0\ 0] \end{array}\right\} \overrightarrow{\text{obama}}.\overrightarrow{\text{president}} = \vec{0}$$

$$\left.\begin{array}{r} \text{speaks} = [0\ 0\ 1\ 0\ \ \dots\ \ 0\ 0\ 0\ 0] \\ \text{addresses} = [0\ 0\ 0\ 0\ \ \dots\ \ 0\ 0\ 1\ 0] \end{array}\right\} \overrightarrow{\text{speaks}}.\overrightarrow{\text{addresses}} = \vec{0}$$

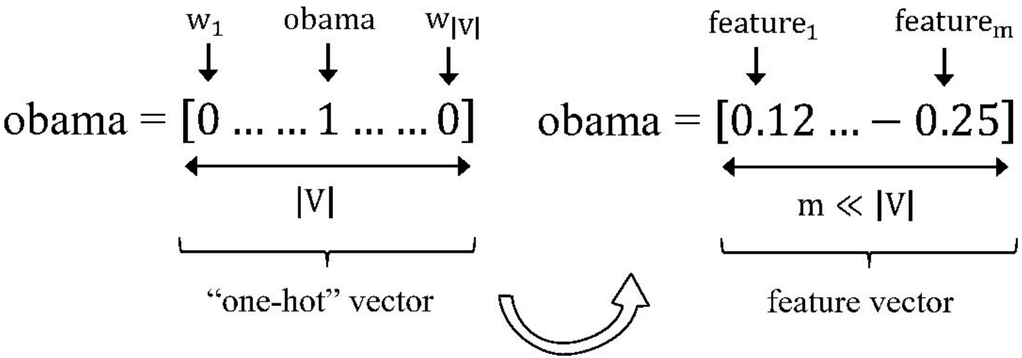
$$\left.\begin{array}{r} \text{illinois} = [1\ 0\ 0\ 0\ \ \dots\ \ 0\ 0\ 0\ 0] \\ \text{chicago} = [0\ 1\ 0\ 0\ \ \dots\ \ 0\ 0\ 0\ 0] \end{array}\right\} \overrightarrow{\text{illinois}}.\overrightarrow{\text{chicago}} = \vec{0}$$

- In each case, word pairs share no similarity
- This is obviously wrong
- We need to encode **word similarity** to be able to **generalize**

Word embeddings: distributed representation of words

- Each unique word is mapped to a point in a real continuous m-dimensional space
- Typically, $|V| > 10^6$, $100 < m < 500$

$$w_i \in V \xrightarrow{\text{mapping } C} \mathbb{R}^{m}$$



w_1 obama $w_{|V|}$

$\text{obama} = [0 \ldots \ldots 1 \ldots \ldots 0]$

$|V|$

“one-hot” vector

feature_1 feature_m

$\text{obama} = [0.12 \ldots - 0.25]$

$m \ll |V|$

feature vector

- Fighting the curse of dimensionality with:
  - **compression** *(dimensionality reduction)*
  - **smoothing** *(discrete to continuous)*
  - **densification** *(sparse to dense)*
- Similar words end up close to each other in the feature space

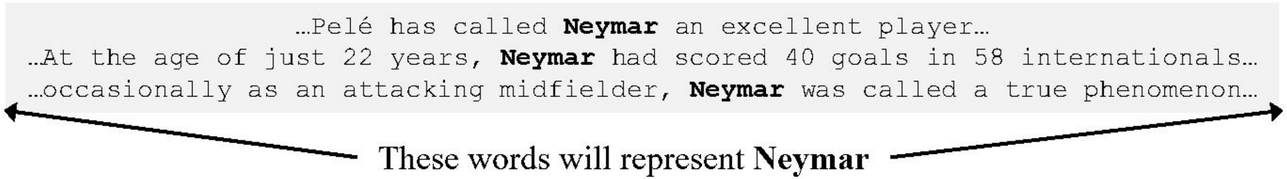
Google's word2vec (Mikolov et al. 2013a)

- Key idea of word2vec: achieve better performance not by using a more complex model (i.e., with more layers), but by allowing a **simpler (shallower) model** to be trained on **much larger amounts of data**

- Two algorithms for learning words vectors:

  - **CBOW**: from context predict target (focus of what follows)
  - **Skip-gram**: from target predict context

- Compared to Bengio et al.'s (2003) NNLM:
  - no hidden layer (leads to 1000X speedup)
  - projection layer is shared (not just the weight matrix)
  - context: words from both **history & future**:
    "You shall know a word by the company it keeps" (John R. Firth 1957:11):



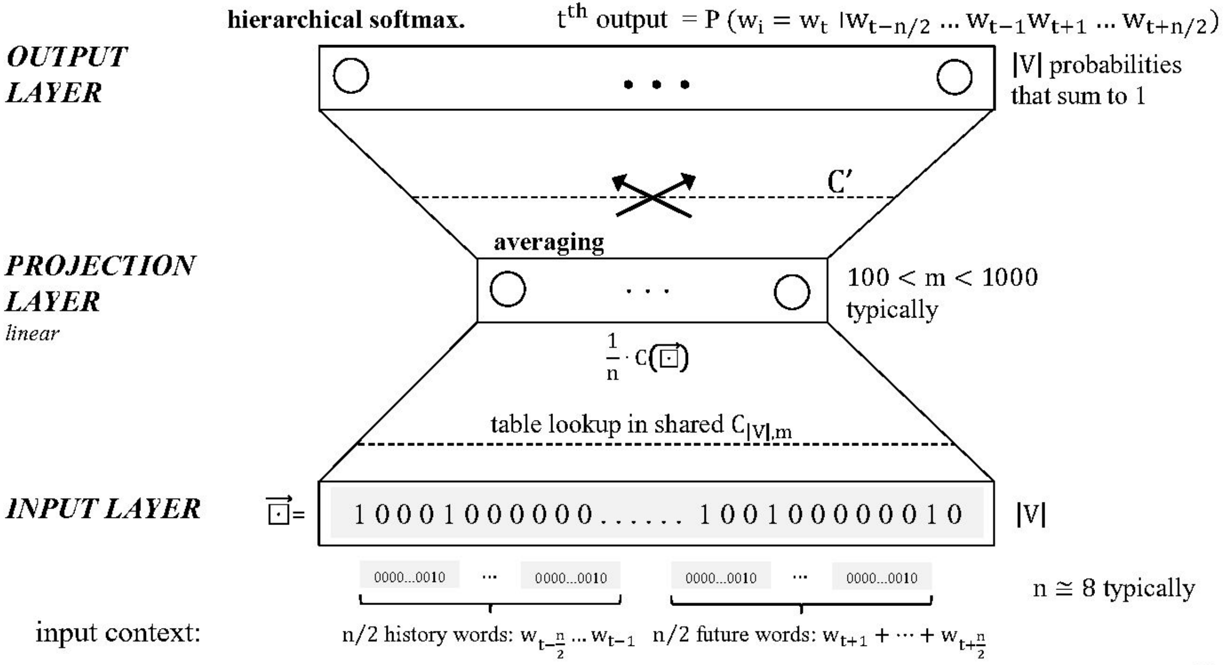
…Pelé has called **Neymar** an excellent player…
…At the age of just 22 years, **Neymar** had scored 40 goals in 58 internationals…
…occasionally as an attacking midfielder, **Neymar** was called a true phenomenon…

These words will represent **Neymar**

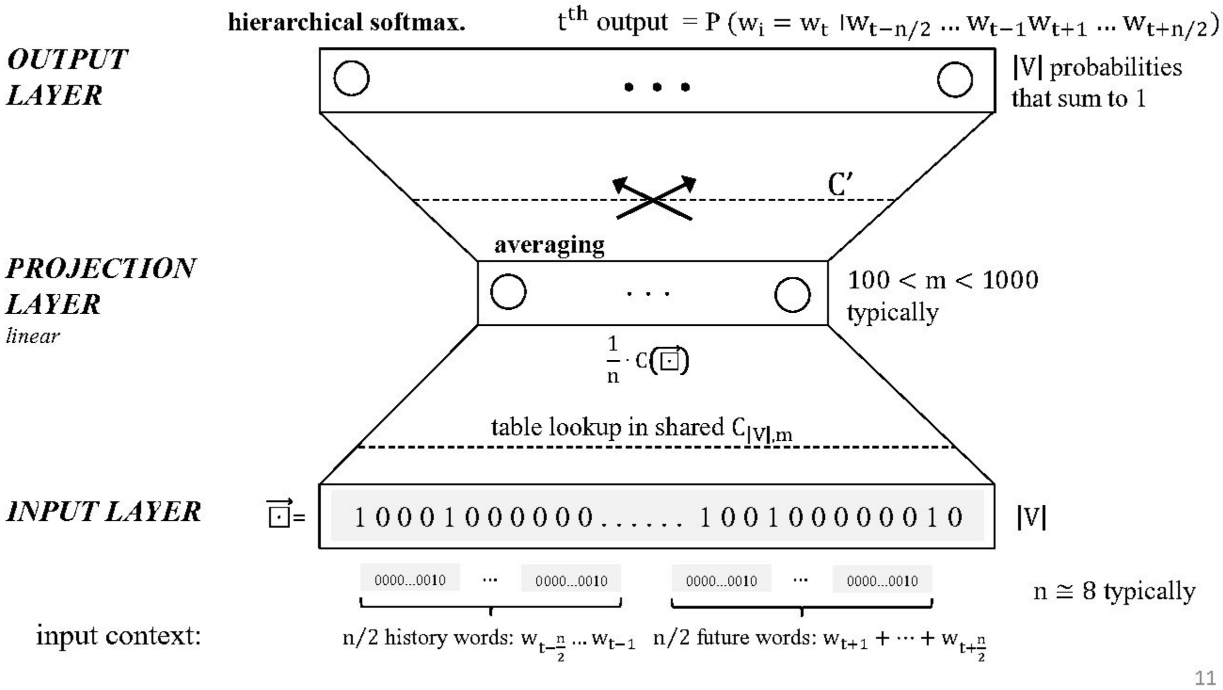
word2vec's Continuous Bag-of-Words (CBOW)

For each training sequence: input = (context, target) pair: $(w_{t-\frac{n}{2}} \dots w_{t-1} w_{t+1} \dots w_{t+\frac{n}{2}}, w_t)$

objective: minimize $E = -\log \widehat{P}\left(w_t \mid w_{t-n/2} \dots w_{t-1} w_{t+1} \dots w_{t+n/2}\right)$



hierarchical softmax.
t^{th} output $= P\,(w_i = w_t \mid w_{t-n/2} \dots w_{t-1} w_{t+1} \dots w_{t+n/2})$
OUTPUT LAYER
|V| probabilities that sum to 1
C'
averaging
PROJECTION LAYER
linear
$100 < m < 1000$ typically
$\frac{1}{n} \cdot C(\vec{\boxdot})$
table lookup in shared $C_{|V|,m}$
INPUT LAYER
$\vec{\boxdot} =$ 1 0 0 0 1 0 0 0 0 0 0 . . . . . . 1 0 0 1 0 0 0 0 0 0 1 0
|V|
0000...0010 ··· 0000...0010 0000...0010 ··· 0000...0010
$n \cong 8$ typically
input context:
n/2 history words: $w_{t-\frac{n}{2}} \dots w_{t-1}$
n/2 future words: $w_{t+1} + \dots + w_{t+\frac{n}{2}}$

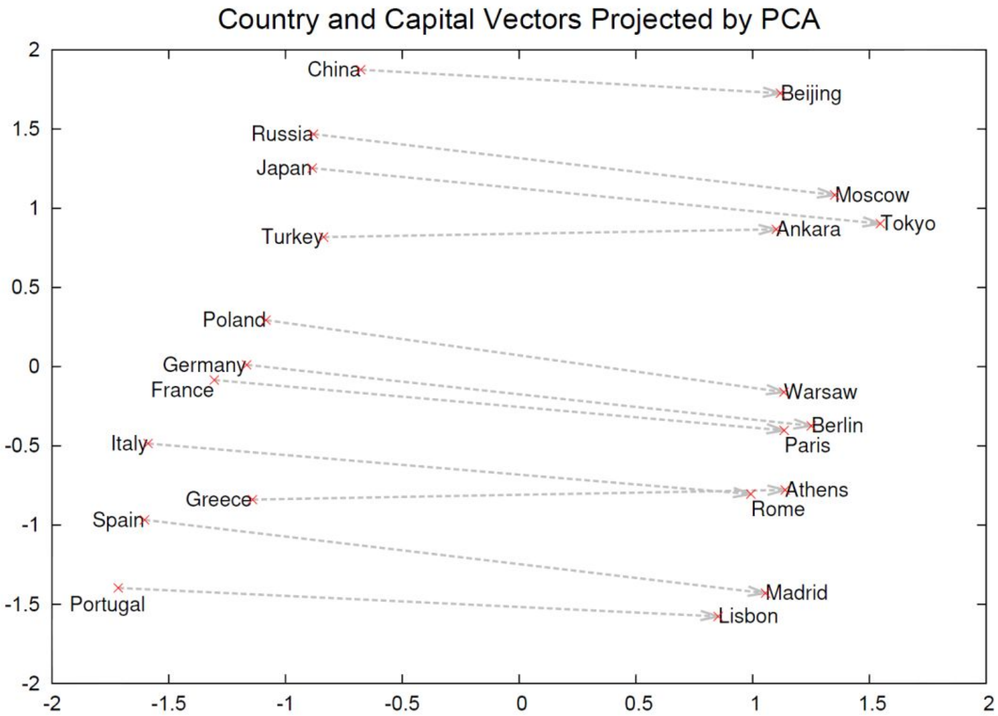

hierarchical softmax.
t^{th} output $= P(w_i = w_t \mid w_{t-n/2} \dots w_{t-1} w_{t+1} \dots w_{t+n/2})$
OUTPUT LAYER
|V| probabilities that sum to 1
C′
averaging
PROJECTION LAYER
linear
100 < m < 1000 typically
$\frac{1}{n} \cdot C(\vec{\boxdot})$
table lookup in shared $C_{|V|,m}$
INPUT LAYER
$\vec{\boxdot}$= 1 0 0 0 1 0 0 0 0 0 0 . . . . . . 1 0 0 1 0 0 0 0 0 0 1 0
|V|
0000...0010 ··· 0000...0010
0000...0010 ··· 0000...0010
n ≅ 8 typically
input context:
n/2 history words: $w_{t-\frac{n}{2}} \dots w_{t-1}$
n/2 future words: $w_{t+1} + \dots + w_{t+\frac{n}{2}}$
11

Ο ανάστροφος πίνακας του C’ έχει διαστάσεις |V| x m

Αν πάρουμε το one-hot διάνυσμα για τη λέξη Neymar (διαστάσεις 1 x |V|) με άσσο μόνο στη λέξη Neymar και τον πολλαπλασιάσουμε με τον προηγούμενο πίνακα θα πάρουμε ένα διάνυσμα 1 x m που είναι πλέον το embedding για τη λέξη Neymar

Όταν στην είσοδο του RNN εμφανίζεται η λέξη Neymar, Αντί του one-hot encoding διαστάσεων |V| με άσσο μόνο στο Neymar, θα εισάγουμε το πυκνό πλέον διάνυσμα 1 x m του embedding

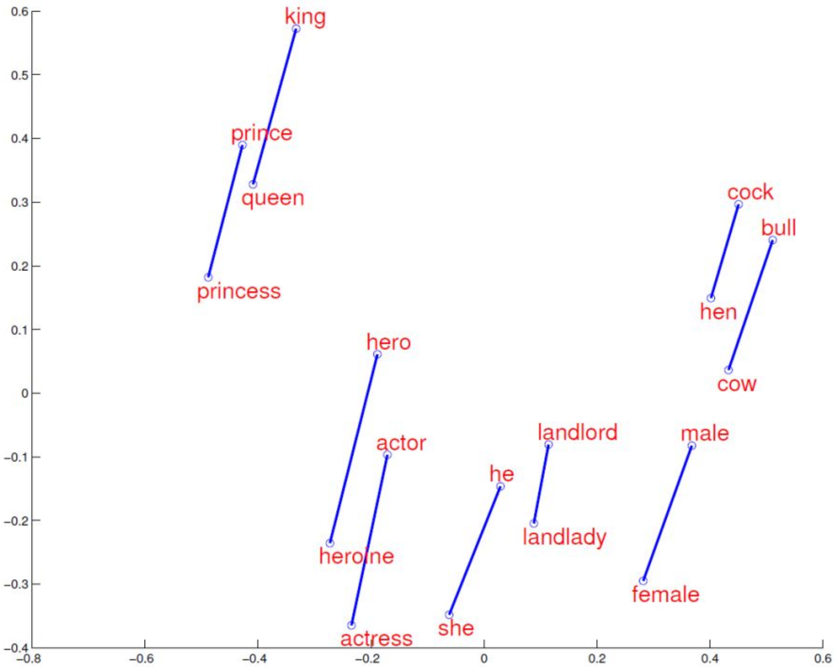
Remarkable properties of word2vec's word vectors



Mikolov et al. (2013b)

regularities between words are encoded in the difference vectors
e.g., there is a constant **country-capital** difference vector

Remarkable properties of word2vec's word vectors



constant **female-male** difference vector

picture taken from http://www.scribd.com/doc/285890694/NIPS-DeepLearningWorkshop-NNforText#scribd

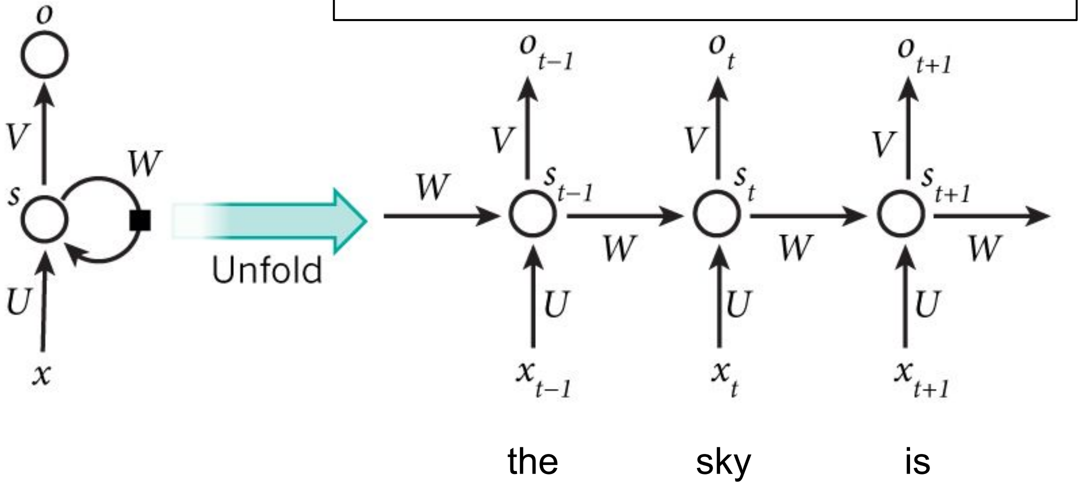
A concrete RNN example

Task: learn a language model e.g. predict next word

The diagram shows an unrolled RNN. By unrolling we simply mean that we write out the network for the complete sequence. For example, if the sequence we care about is a sentence of 5 words, the network would be unrolled into a 5-layer neural network, one layer for each word.

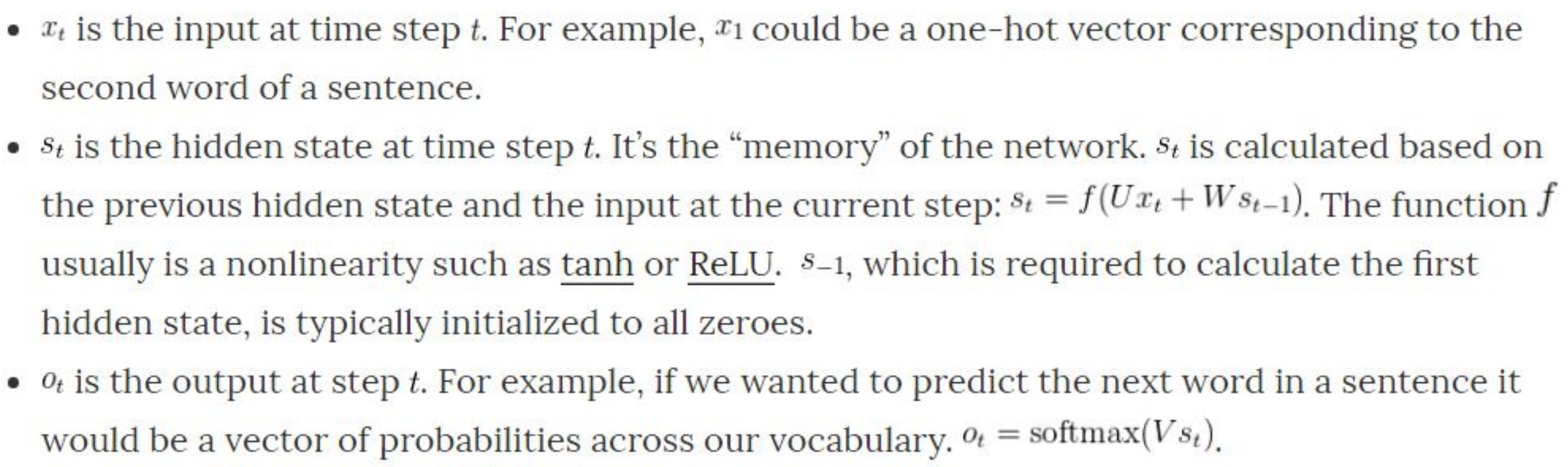
Expected output y (supervised training at each time step)

sky is blue



Since the all the weights (U,V,W) are shared for all time steps we can treat sequences (sentences) of different length

A concrete RNN example

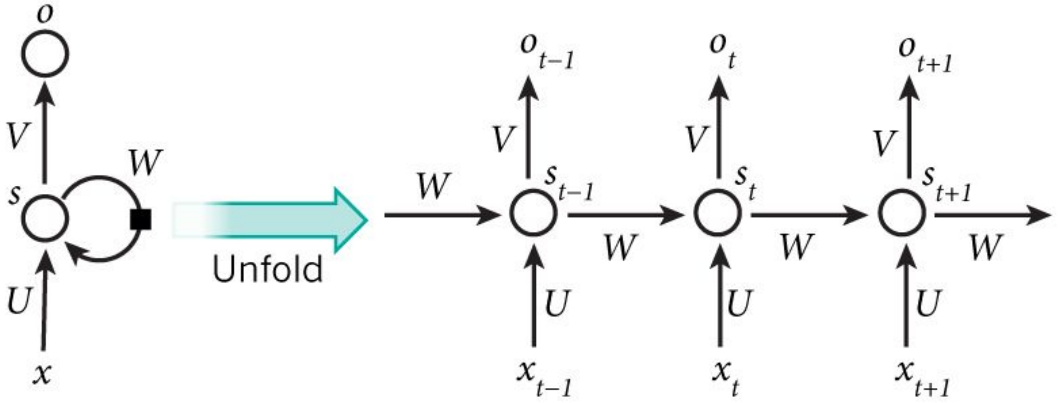


- x_t is the input at time step t. For example, x_1 could be a one-hot vector corresponding to the second word of a sentence.
- s_t is the hidden state at time step t. It's the "memory" of the network. s_t is calculated based on the previous hidden state and the input at the current step: $s_t = f(Ux_t + Ws_{t-1})$. The function f usually is a nonlinearity such as tanh or ReLU. s_{-1}, which is required to calculate the first hidden state, is typically initialized to all zeroes.
- o_t is the output at step t. For example, if we wanted to predict the next word in a sentence it would be a vector of probabilities across our vocabulary. $o_t = \mathrm{softmax}(Vs_t)$.

A concrete RNN example

- Vocabulary size C = 8000
- Hidden layer size H = 100

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$o_t = \mathrm{softmax}(Vs_t)$$



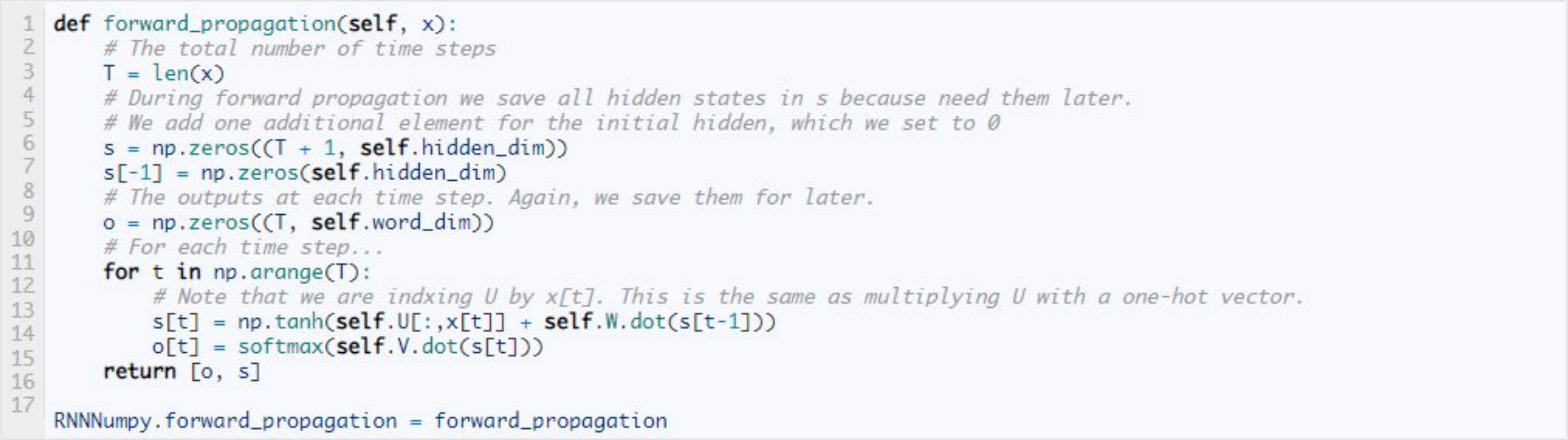
$$x_t \in \mathbb{R}^{8000}$$
$$o_t \in \mathbb{R}^{8000}$$
$$s_t \in \mathbb{R}^{100}$$
$$U \in \mathbb{R}^{100\times 8000}$$
$$V \in \mathbb{R}^{8000\times 100}$$
$$W \in \mathbb{R}^{100\times 100}$$

Total parameters (weights) = 2HC + H^2. In the case of C=8000 and H=100 that’s 1,610,000

Forward pass

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$o_t = \mathrm{softmax}(Vs_t)$$

$$x_t \in \mathbb{R}^{8000}$$
$$o_t \in \mathbb{R}^{8000}$$
$$s_t \in \mathbb{R}^{100}$$
$$U \in \mathbb{R}^{100\times8000}$$
$$V \in \mathbb{R}^{8000\times100}$$
$$W \in \mathbb{R}^{100\times100}$$



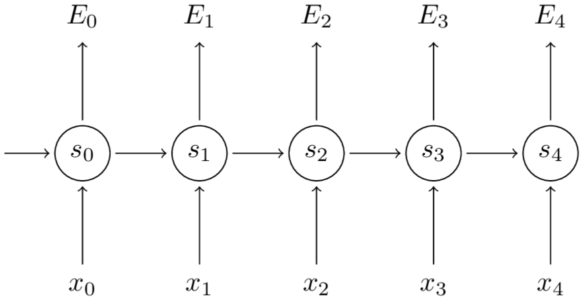
```python
def forward_propagation(self, x):
    # The total number of time steps
    T = len(x)
    # During forward propagation we save all hidden states in s because need them later.
    # We add one additional element for the initial hidden, which we set to 0
    s = np.zeros((T + 1, self.hidden_dim))
    s[-1] = np.zeros(self.hidden_dim)
    # The outputs at each time step. Again, we save them for later.
    o = np.zeros((T, self.word_dim))
    # For each time step...
    for t in np.arange(T):
        # Note that we are indxing U by x[t]. This is the same as multiplying U with a one-hot vector.
        s[t] = np.tanh(self.U[:,x[t]] + self.W.dot(s[t-1]))
        o[t] = softmax(self.V.dot(s[t]))
    return [o, s]

RNNNumpy.forward_propagation = forward_propagation
```

- We store all states s_t => longer sentences need more memory.
- At each step we output probabilities for all words in the vocabulary to be the next word => use softmax to get the most probable at each step

Backpropagation Through Time

$$
\begin{aligned}
s_t &= \tanh(Ux_t + Ws_{t-1}) \\
\hat{y}_t &= \mathrm{softmax}(Vs_t)
\end{aligned}
$$

We typically treat the full sequence (sentence) as one training example, so the total error is just the sum of the errors at each time step (word).



Error (or loss) e.g. cross-entropy (y: true labels)

$$
\begin{aligned}
E_t(y_t, \hat{y}_t) &= -y_t \log \hat{y}_t \\
E(y, \hat{y}) &= \sum_t E_t(y_t, \hat{y}_t) \\
&= -\sum_t y_t \log \hat{y}_t
\end{aligned}
$$

For SGD we need to calculate the gradients for all the weights V,U,W. Like we sum up the errors, we also sum up the gradients at each time step for one training example:

$$
\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}
$$

Backpropagation Through Time

$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \mathrm{softmax}(Vs_t)$$

To calculate these gradients we use the chain rule of differentiation. We will use E_3 as an example.

For V, $\frac{\partial E_3}{\partial V}$ only depends on the values at the current time step $\hat{y}_3, y_3, s_3$ ($z_3=Vs_3$)

$$\begin{aligned}\frac{\partial E_3}{\partial V} &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V}\\ &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V}\\ &= (\hat{y}_3 - y_3) \otimes s_3\end{aligned}$$

For W (and U) the calculation depends on the previous steps $\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial W}$ but $s_3 = \tanh(Ux_t + Ws_2)$

We need to apply the chain rule again $\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$. $\frac{\partial s_3}{\partial s_k}$ is a chain rule in itself $\frac{\partial s_3}{\partial s_1} = \frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}$ so we can rewrite it as

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial s_3}\left(\prod_{j=k+1}^{3} \frac{\partial s_j}{\partial s_{j-1}}\right)\frac{\partial s_k}{\partial W}$$

Backpropagation Through Time

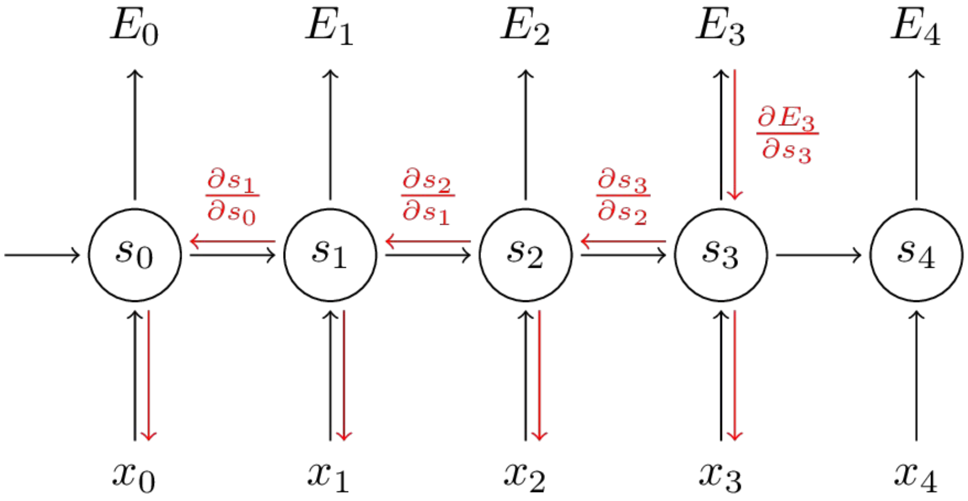
$$s_t = \tanh(Ux_t + Ws_{t-1})$$
$$\hat{y}_t = \mathrm{softmax}(Vs_t)$$

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

BPTT is just a standard backpropagation on an unrolled RNN with the only difference is that we sum up the gradients for W at each time step.

Because W is used in every step up to the output we care about, we need to backpropagate gradients from t=3 through the network all the way to t=0

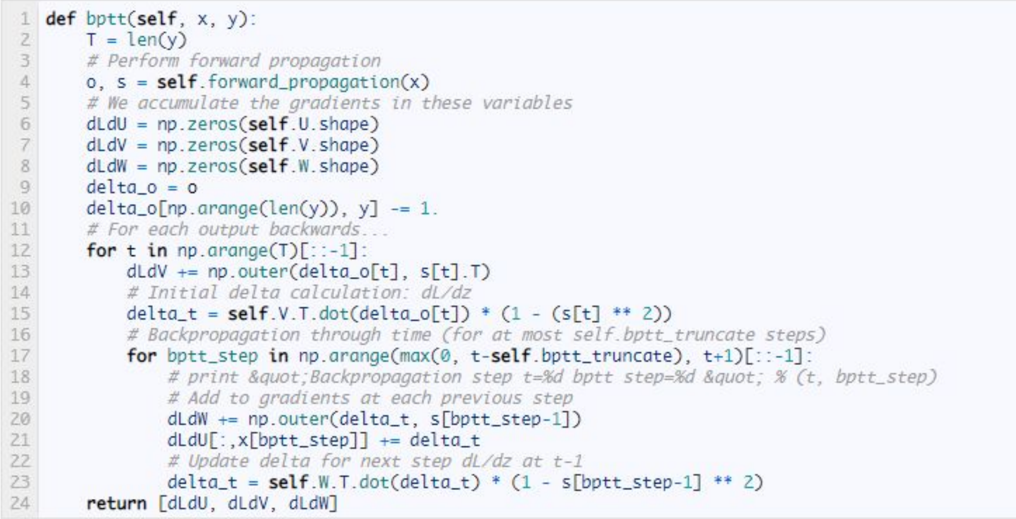
[Williams & Zipser 1995]



E_0
E_1
E_2
E_3
E_4
$\frac{\partial E_3}{\partial s_3}$
$\frac{\partial s_1}{\partial s_0}$
$\frac{\partial s_2}{\partial s_1}$
$\frac{\partial s_3}{\partial s_2}$
s_0
s_1
s_2
s_3
s_4
x_0
x_1
x_2
x_3
x_4

Backpropagation Through Time

- We use gradient accumulators for V,W,U
- Bishops delta rule
- In practice many people truncate the backpropagation to a few steps.



```python
def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o, s = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = o
    delta_o[np.arange(len(y)), y] -= 1.
    # For each output backwards...
    for t in np.arange(T)[::-1]:
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation: dL/dz
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-1]:
            # print &quot;Backpropagation step t=%d bptt step=%d &quot; % (t, bptt_step)
            # Add to gradients at each previous step
            dLdW += np.outer(delta_t, s[bptt_step-1])
            dLdU[:,x[bptt_step]] += delta_t
            # Update delta for next step dL/dz at t-1
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]
```

Limitations of RNNs

- In principle, recurrent networks are capable of learning long distance dependencies.
- In practice, standard gradient-based learning algorithms do not perform very well.
  - Bengio et al. (1994) – the ‘vanishing gradient’ problem.
  - Mikolov & Bengio (2013) - the ‘exploding gradient’ problem. “Clipping” solution
  - The gradient is a product of Jacobian matrices, each associated with a step in the forward computation. This can become very small or very large quickly.
- Nevertheless, the repeating cell structure is powerful
- Today, there are several methods available for training recurrent neural networks that avoids these problems.
  - LSTMs, optimisation with small gradients, careful weight initialisations, …

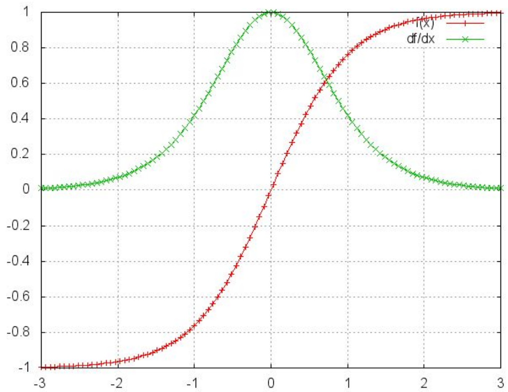
The Vanishing Gradient problem

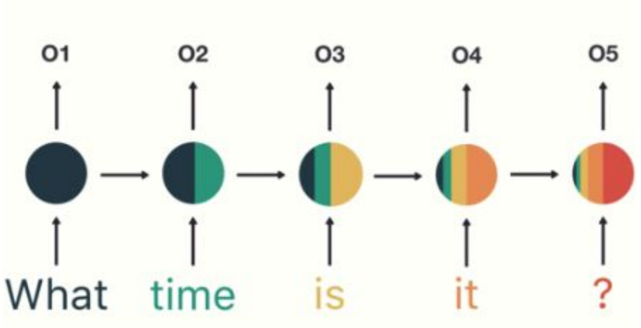
The tanh and sigmoid functions have derivatives of 0 at both ends.

Thus, with small values in the matrix and multiple matrix multiplications the gradient values are shrinking exponentially fast, eventually vanishing completely after a few time steps.

Gradient contributions from “far away” steps become zero, and the state at those steps doesn’t contribute to what you are learning: You end up not learning long-range dependencies.

Vanishing gradients aren’t exclusive to RNNs. They also happen in deep Feedforward Neural Networks. It’s just that RNNs tend to be very deep (as deep as the sentence length in our case), which makes the problem a lot more common.





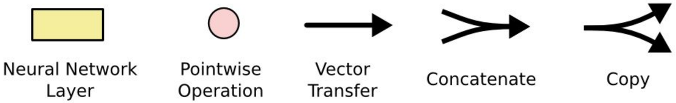
Long Short Term Memory networks (LSTM)

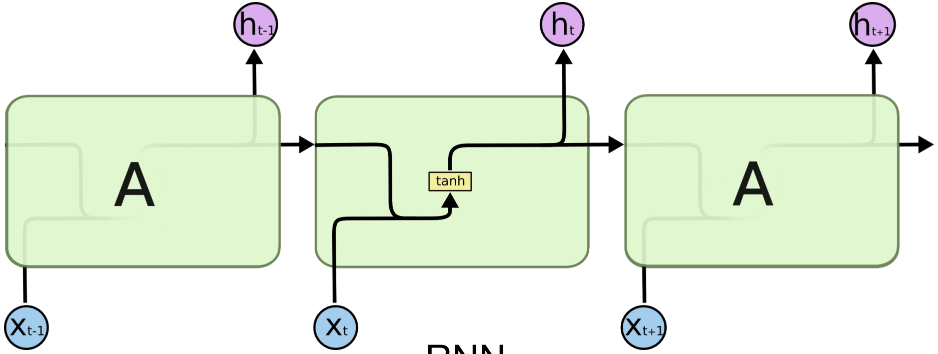
Hochreiter & Schmidhuber (1997)

All recurrent neural networks have the form of a chain of repeating modules of neural network.

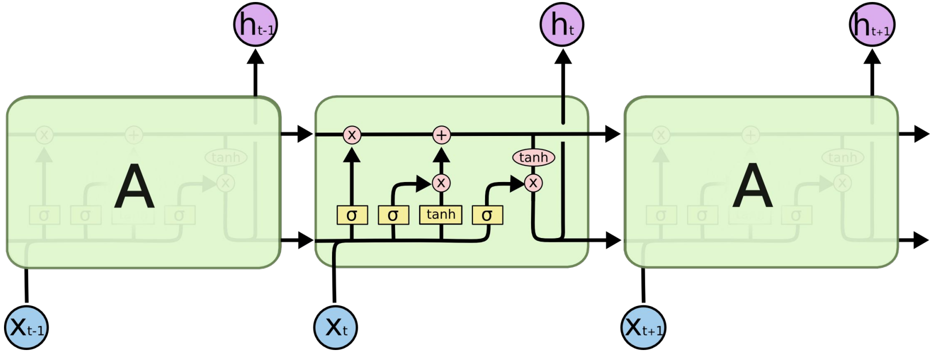
In standard RNNs, this repeating module is a single tanh layer.

In LSTMs repeating module has four layers instead of one.





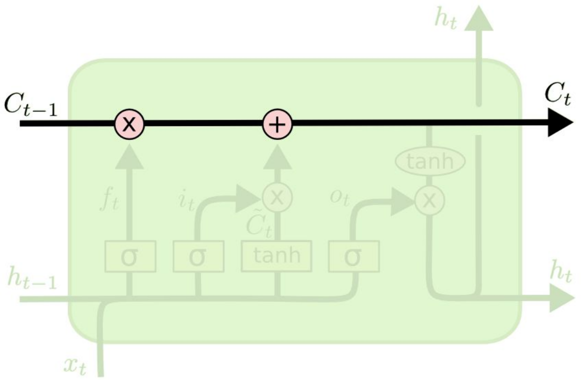
RNN



LSTM

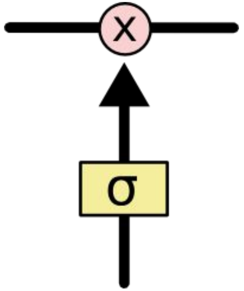
The cell state and three gates

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



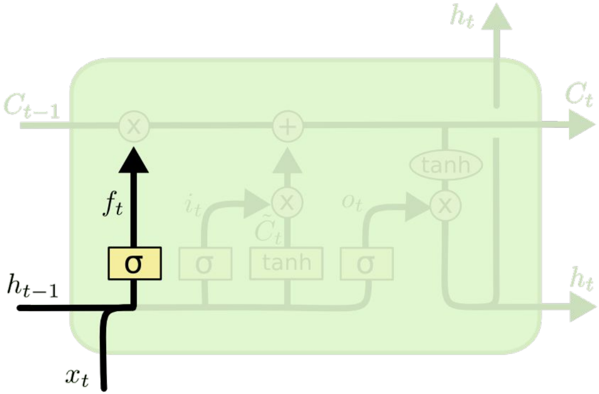
The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!"



An LSTM has three of these gates, to protect and control the cell state.

Forget gate



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

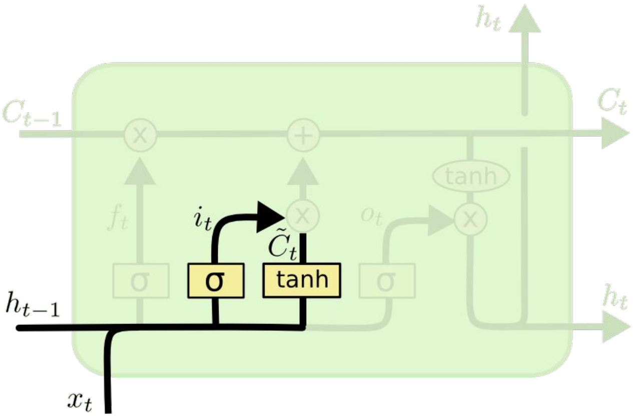
The first step is to decide what information we’re going to throw away from the cell state.

This decision is made by a sigmoid layer called the “forget gate layer.”

It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}.

A 1 represents “keep this” while a 0 represents “completely get rid of this.”

Input gate



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

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

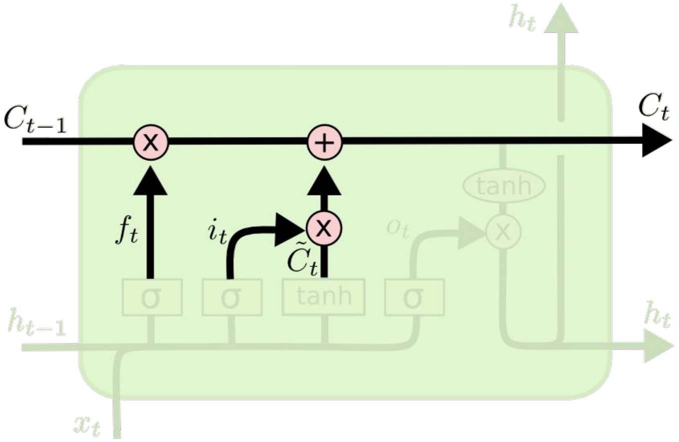
The next step is to decide what new information to store in the cell state. This has two parts:

First, a sigmoid layer called the “input gate layer” decides which values will be updated.

Next, a tanh layer creates a vector of new candidate values, $C\sim_t$, that could be added to the state.

In the next step, we combine these two to create an update to the state.

Cell state update



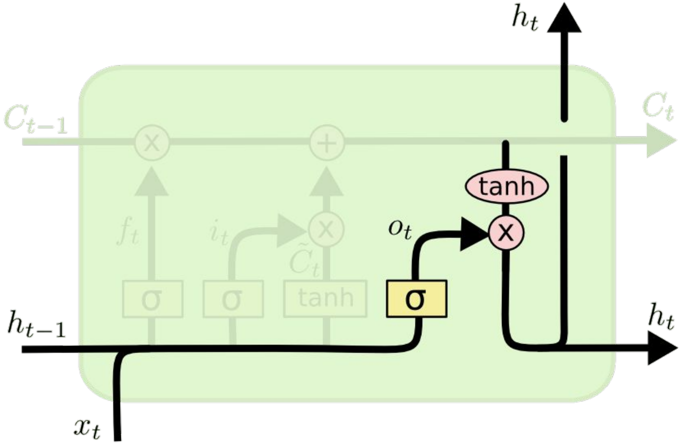
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

We now update the old cell state, C_{t-1}, into the new cell state C_t.

We multiply the old state by f_t, forgetting the things we decided to forget earlier.

Then we add i_t∗$C\sim_t$. This is the new candidate values, scaled by how much we decided to update each state value.

Output



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

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

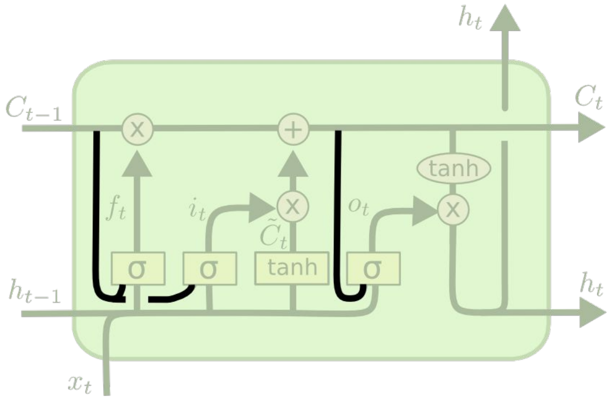
Finally, we decide what we're going to output. This output will be based on our cell state, but will be a filtered version.

First, we run a sigmoid layer over h_{t-1} which decides what parts of the cell state we're going to output.

Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

Overall weights to learn: W_f, W_i, W_c, W_o

Variants: “Peephole” LSTMs

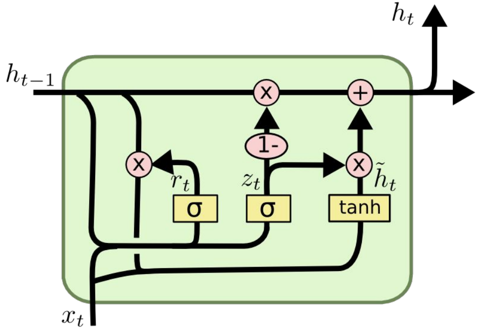


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \; + \; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \; + \; b_o\right)$$

We add “peephole connections.” This means that we let the gate layers look at the cell state.

[Gers & Schmidhuber (2000)]

Variants: Gated Recurrent Unit (GRU)



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

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

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

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

GRUs combine the forget and input gates into a single “update gate.”

It also merges the cell state and hidden state.

The resulting model is simpler than standard LSTM models, and has been growing increasingly popular.

[Cho, et al. (2014)]

- Many more LSTM variants, many hyperparameters, empirical evaluation

Deep RNNs and LSTMs

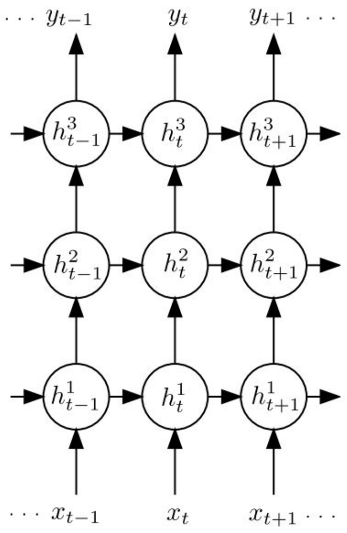
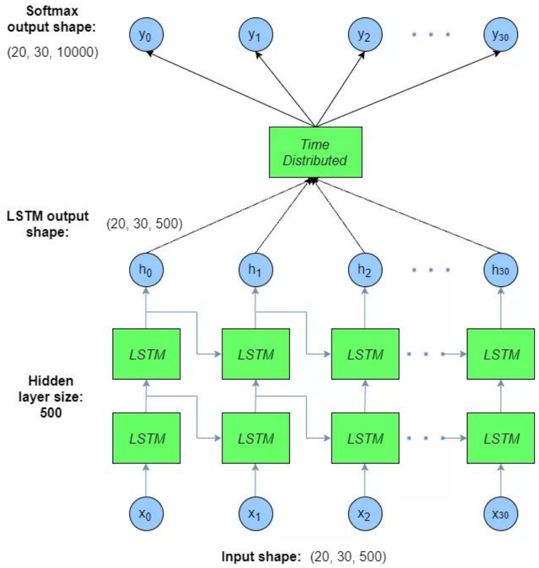


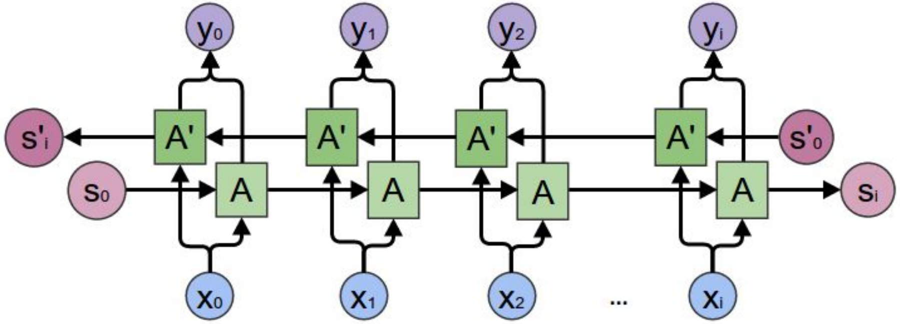
Fig. 3. Deep Recurrent Neural Network

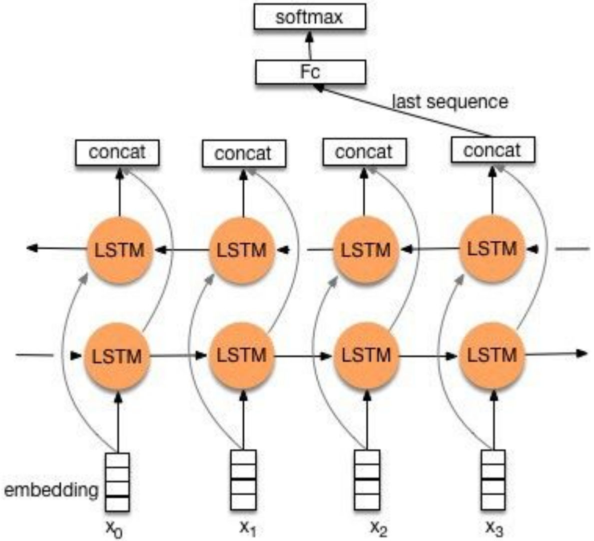


Deep LSTM

- Each layer learns higher level features
- For recurrent architectures the depth is usually low (2-4)

Bidirectional RNNs and LSTMs



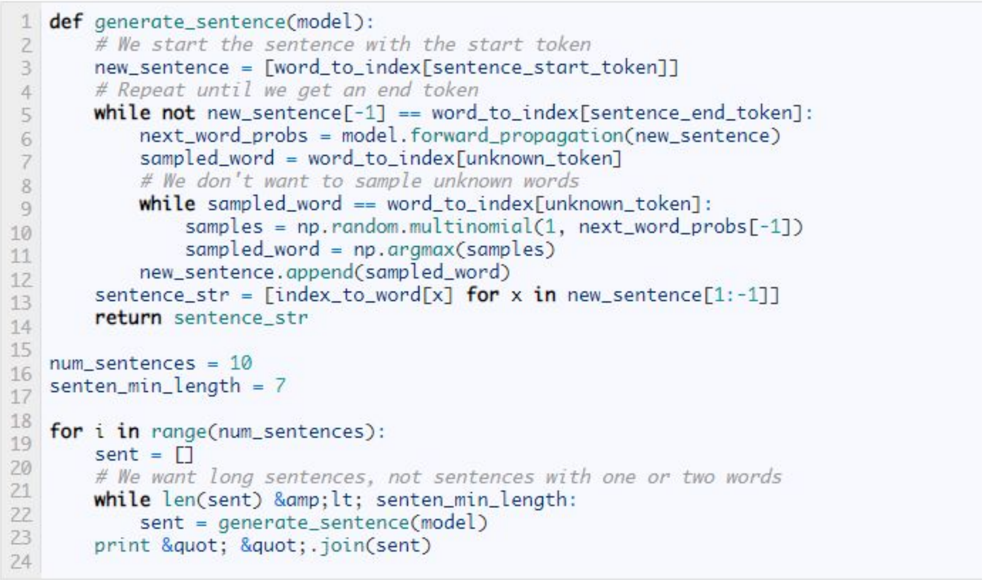


- We split the input and train two networks in reverse order
- We concatenate the hidden layers outputs at each step to calculate outputs

[Graves 2005]

Applications of sequence modeling: Text Generation

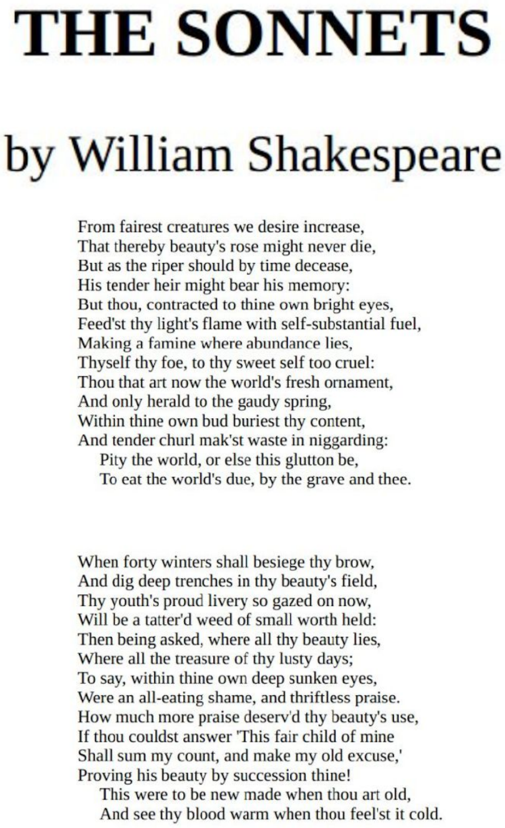
- Input sequence is of same type (words) as output sequence.
- We take a sample of next words and decide with argmax or we take top-k probable words and select one at random or perform random multinomial experiments with the respective probabilities
- Repeat for next word



```
def generate_sentence(model):
    # We start the sentence with the start token
    new_sentence = [word_to_index[sentence_start_token]]
    # Repeat until we get an end token
    while not new_sentence[-1] == word_to_index[sentence_end_token]:
        next_word_probs = model.forward_propagation(new_sentence)
        sampled_word = word_to_index[unknown_token]
        # We don't want to sample unknown words
        while sampled_word == word_to_index[unknown_token]:
            samples = np.random.multinomial(1, next_word_probs[-1])
            sampled_word = np.argmax(samples)
        new_sentence.append(sampled_word)
    sentence_str = [index_to_word[x] for x in new_sentence[1:-1]]
    return sentence_str

num_sentences = 10
senten_min_length = 7

for i in range(num_sentences):
    sent = []
    # We want long sentences, not sentences with one or two words
    while len(sent) &amp;lt; senten_min_length:
        sent = generate_sentence(model)
    print &quot; &quot;.join(sent)
```

- the RNN hidden layers learn and store the language model

Karpathy: The Unreasonable Effectiveness of Recurrent Neural Networks

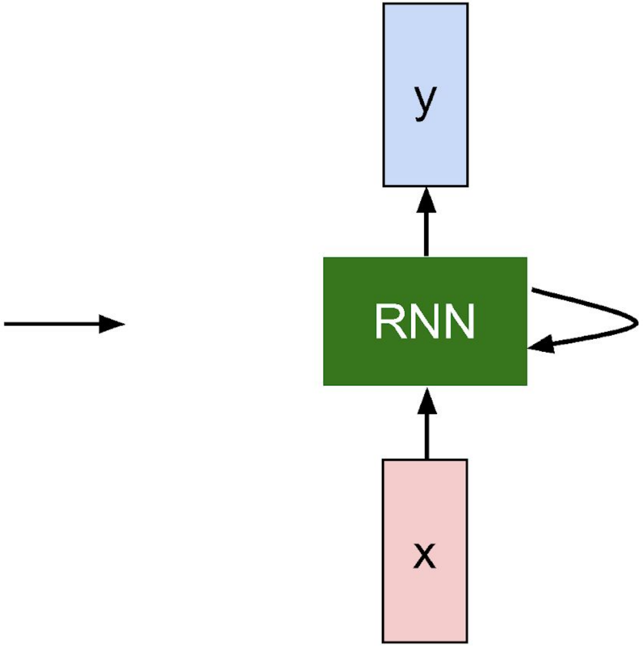


THE SONNETS

by William Shakespeare

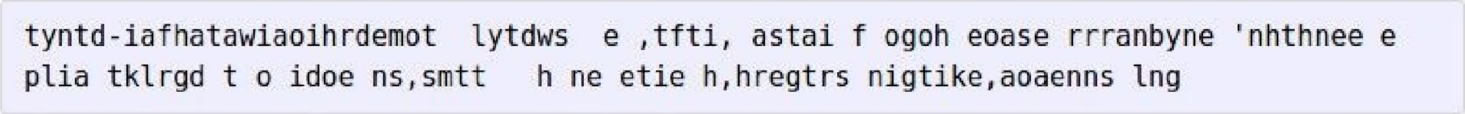
From fairest creatures we desire increase,
That thereby beauty's rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou, contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thyself 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.



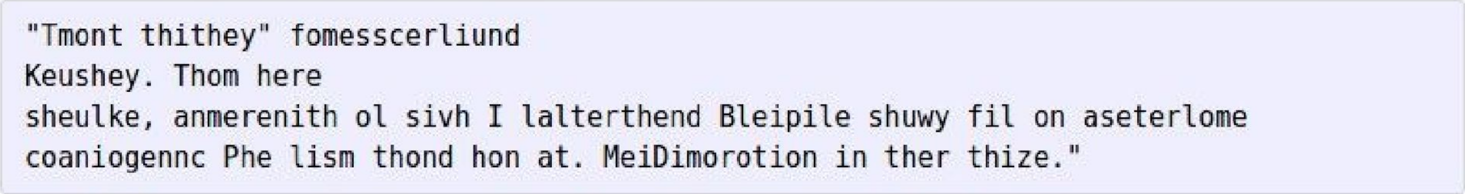
- character - level generation

at first:



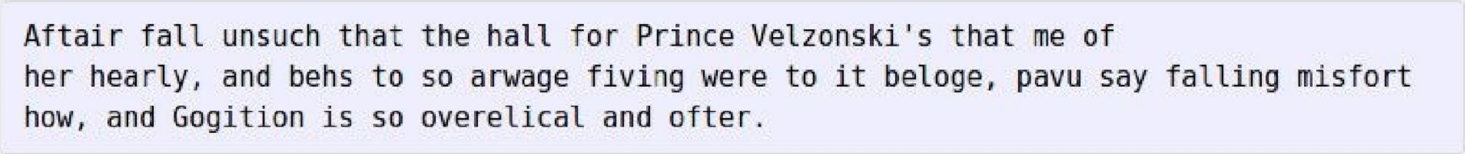
```
tyntd-iafhatawiaoihrdemot  lytdws  e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e
plia tklrgd t o idoe ns,smtt   h ne etie h,hregtrs nigtike,aoaenns lng
```

train more



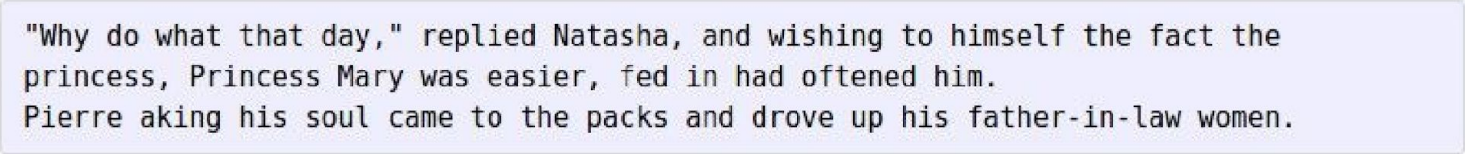
```
"Tmont thithey" fomesscerliund
Keushey. Thom here
sheulke, anmerenith ol sivh I lalterthend Bleipile shuwy fil on aseterlome
coaniogennc Phe lism thond hon at. MeiDimorotion in ther thize."
```

train more



```
Aftair fall unsuch that the hall for Prince Velzonski's that me of
her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort
how, and Gogition is so overelical and ofter.
```

train more



```
"Why do what that day," replied Natasha, and wishing to himself the fact the
princess, Princess Mary was easier, fed in had oftened him.
Pierre aking his soul came to the packs and drove up his father-in-law women.
```

Applications of sequence modeling

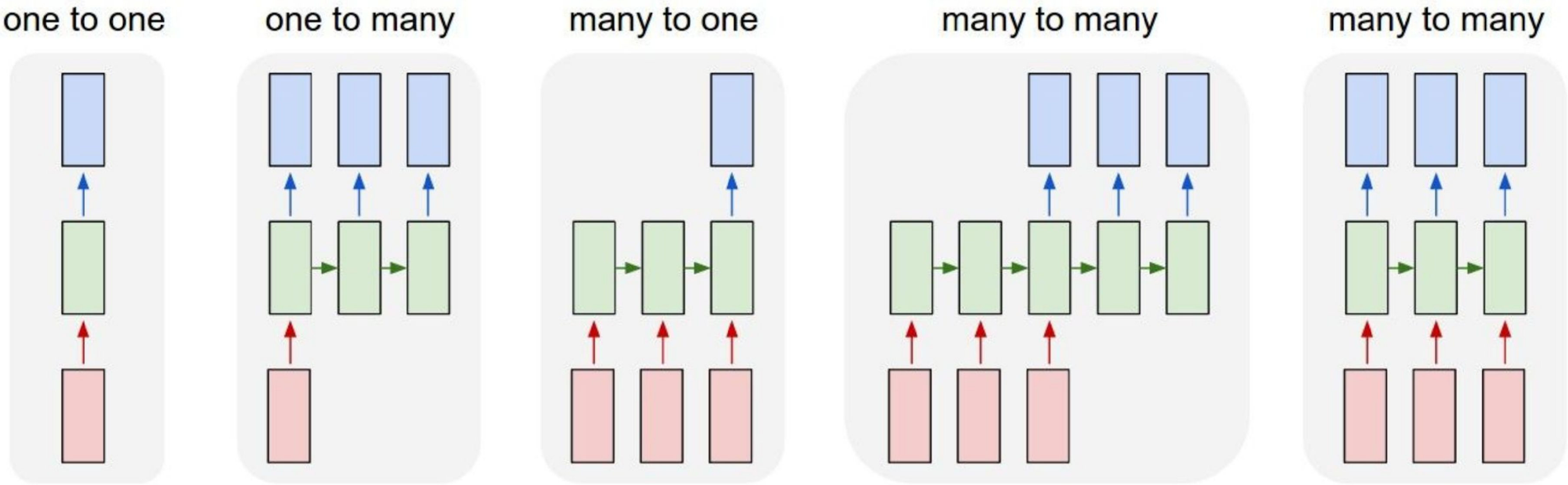
Input can be different modality than the output.

Encode input in hidden state, decode in output

One to many, many to many, many to one schemes

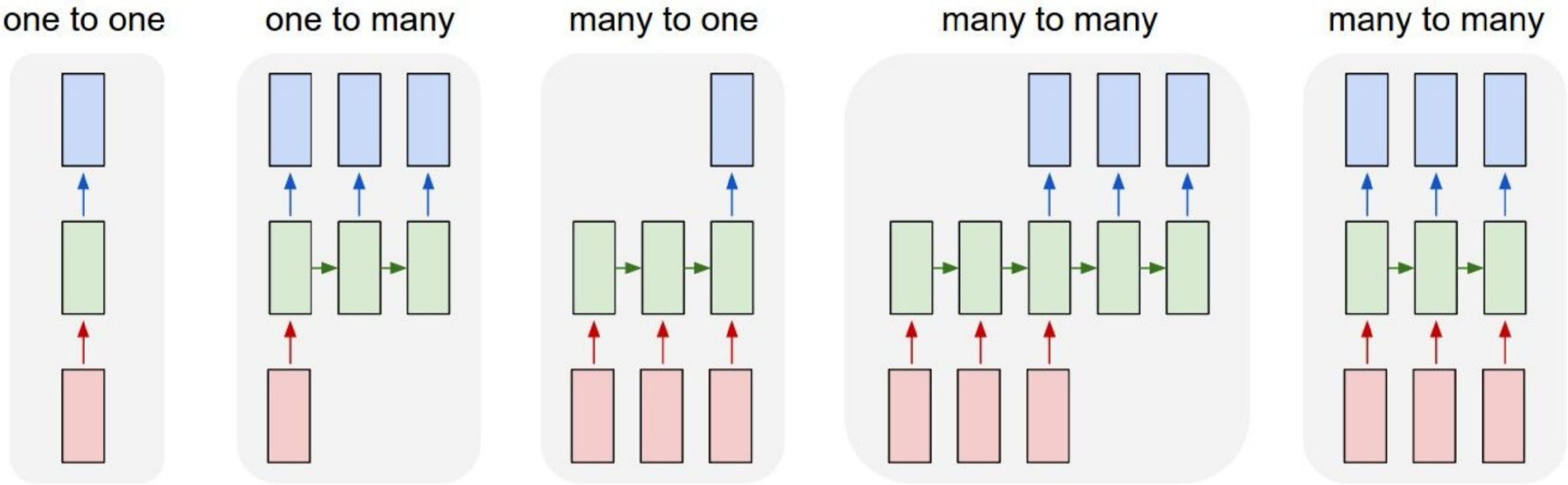
- Image captioning
- Map unsegmented connected handwriting to strings.
- Map sequences of acoustic signals to sequences of phonemes.
- Translate sentences from one language into another one.

Recurrent Neural Networks: Process Sequences



e.g. **Image Captioning**
image -> sequence of words

Recurrent Neural Networks: Process Sequences



one to one

one to many

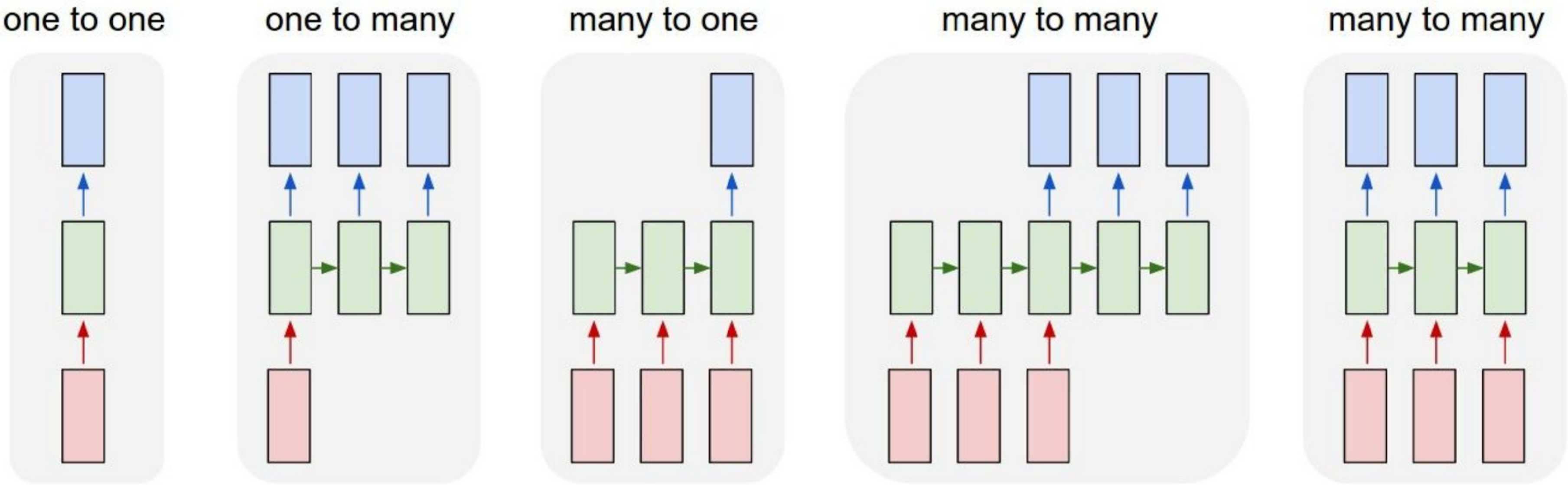
many to one

many to many

many to many

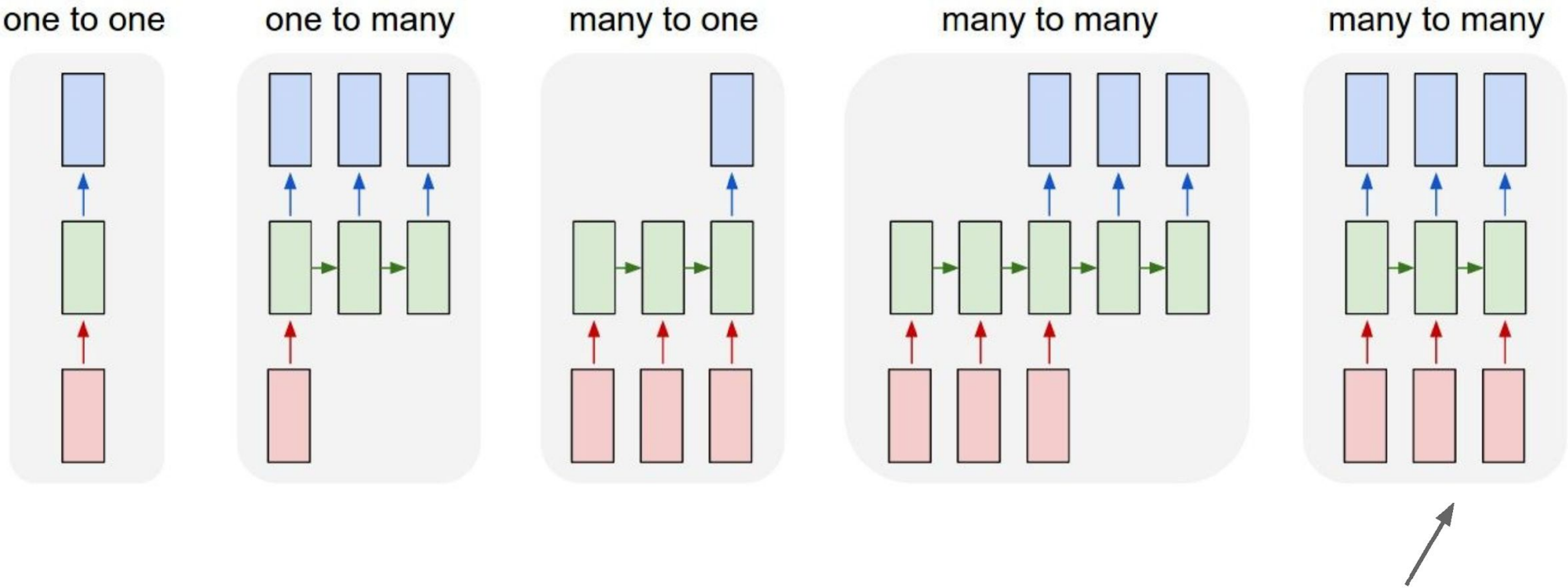
e.g. **Sentiment Classification**
sequence of words -> sentiment

Recurrent Neural Networks: Process Sequences



e.g. **Machine Translation**
seq of words -> seq of words

Recurrent Neural Networks: Process Sequences



e.g. **Video classification on frame level**

Image Captioning

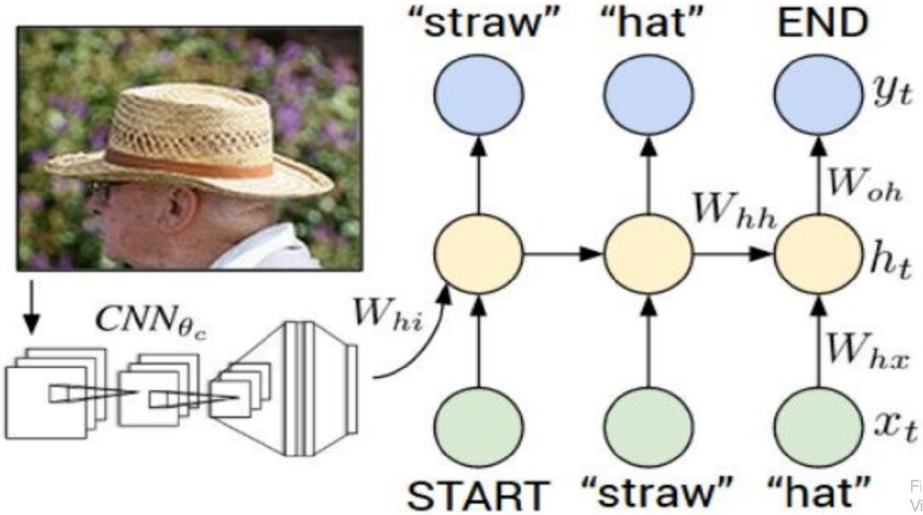
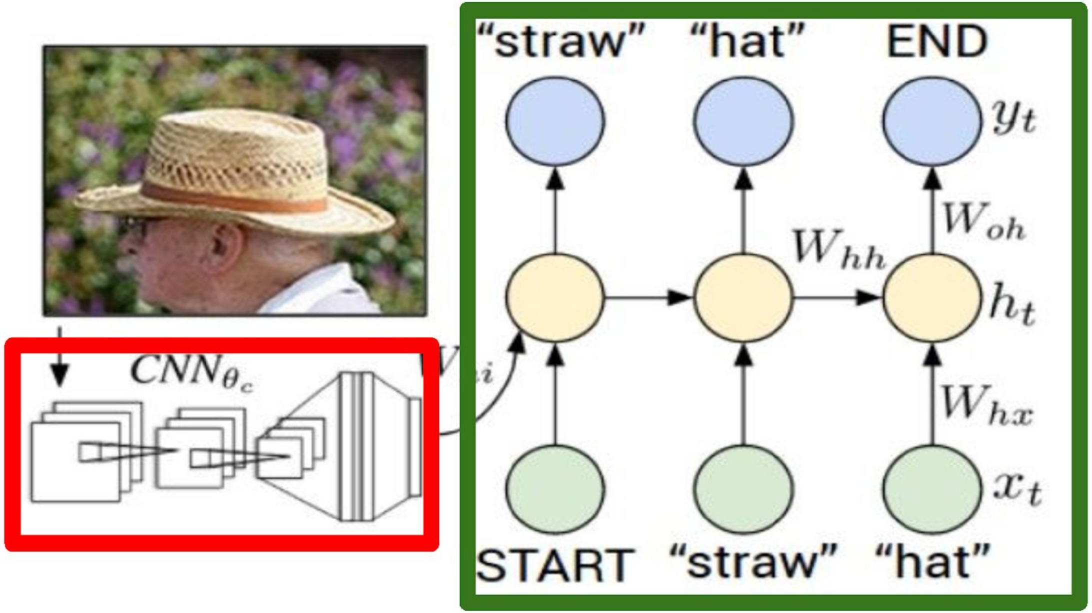


Figure from Karpathy et a, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015; figure copyright IEEE, 2015.
Reproduced for educational purposes.

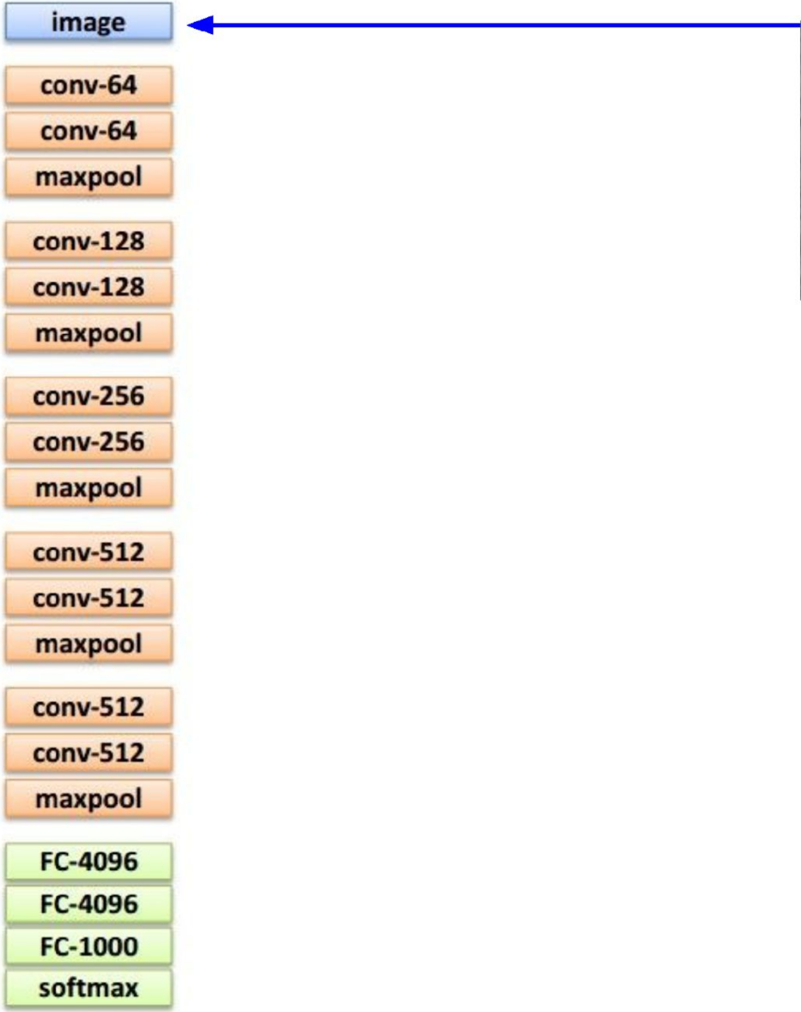
Explain Images with Multimodal Recurrent Neural Networks, Mao et al.
Deep Visual-Semantic Alignments for Generating Image Descriptions, Karpathy and Fei-Fei
Show and Tell: A Neural Image Caption Generator, Vinyals et al.
Long-term Recurrent Convolutional Networks for Visual Recognition and Description, Donahue et al.
Learning a Recurrent Visual Representation for Image Caption Generation, Chen and Zitnick

Recurrent Neural Network



Convolutional Neural Network

test image



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

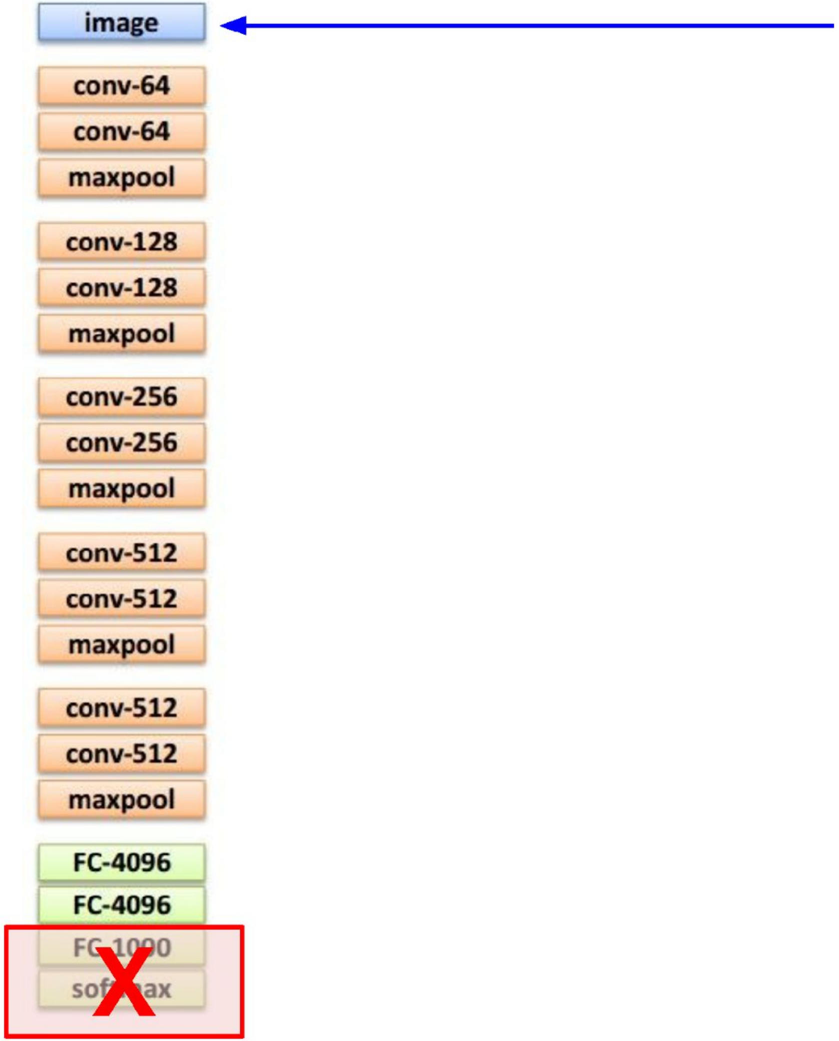
FC-4096

FC-4096

FC-1000

softmax

test image



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

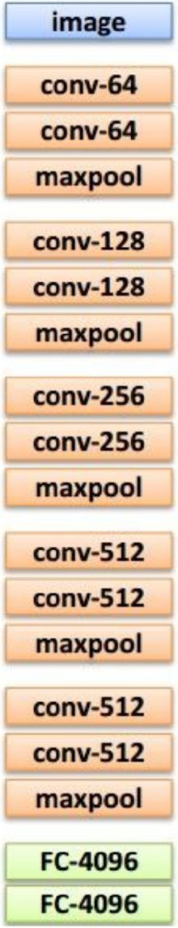
FC-4096

FC-4096

~~FC-1000~~

~~softmax~~

test image



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

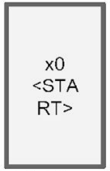
conv-512

maxpool

FC-4096

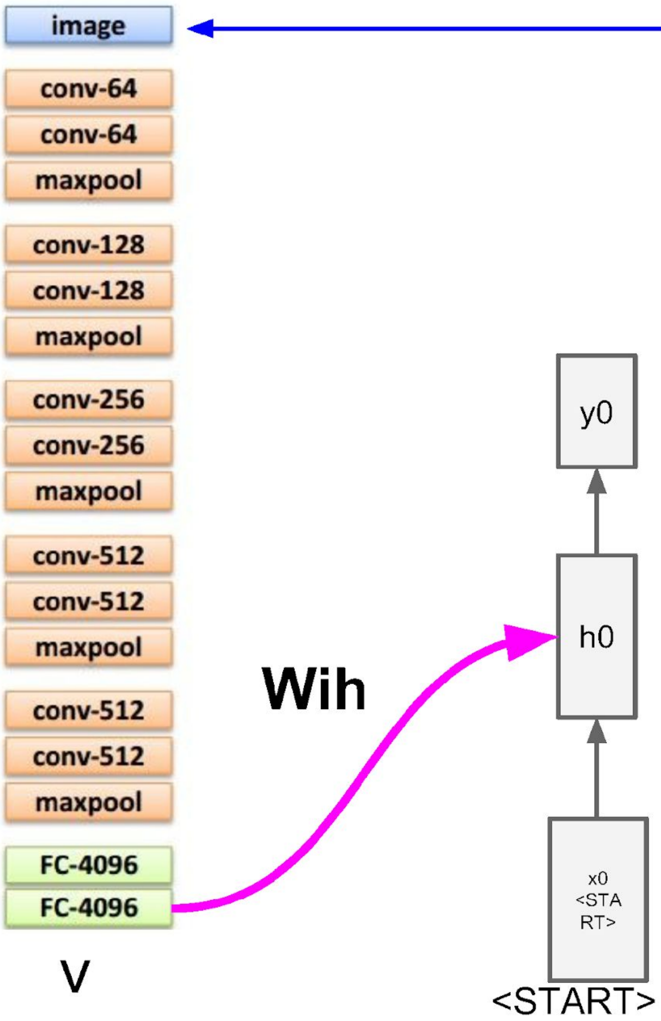
FC-4096

test image



x0
<START>

<START>



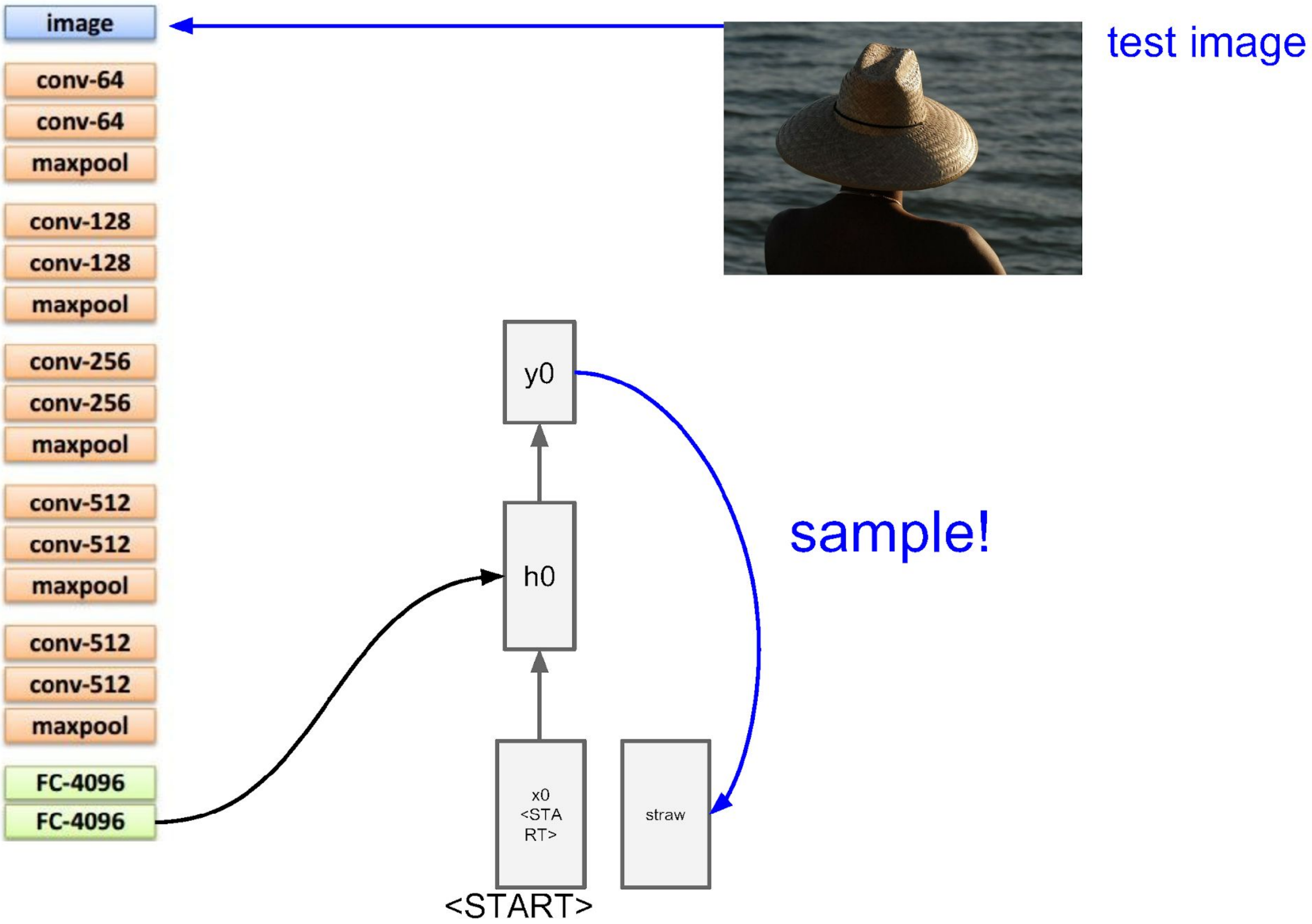
test image

before:

$$h = tanh(Wxh * x + Whh * h)$$

now:

$$h = tanh(Wxh * x + Whh * h + \mathbf{Wih * v})$$



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

test image

y0

h0

x0 <START>

straw

sample!

<START>

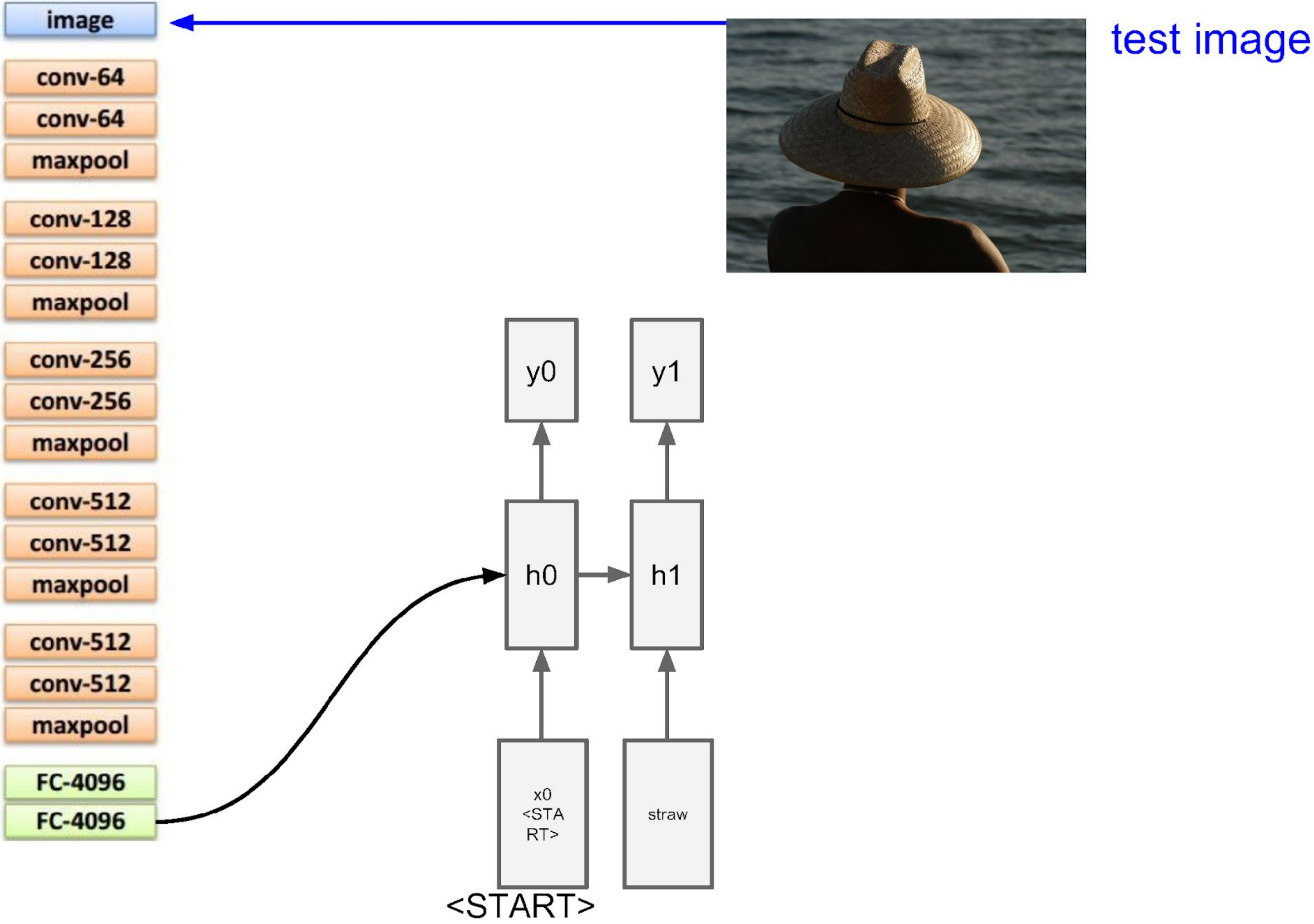
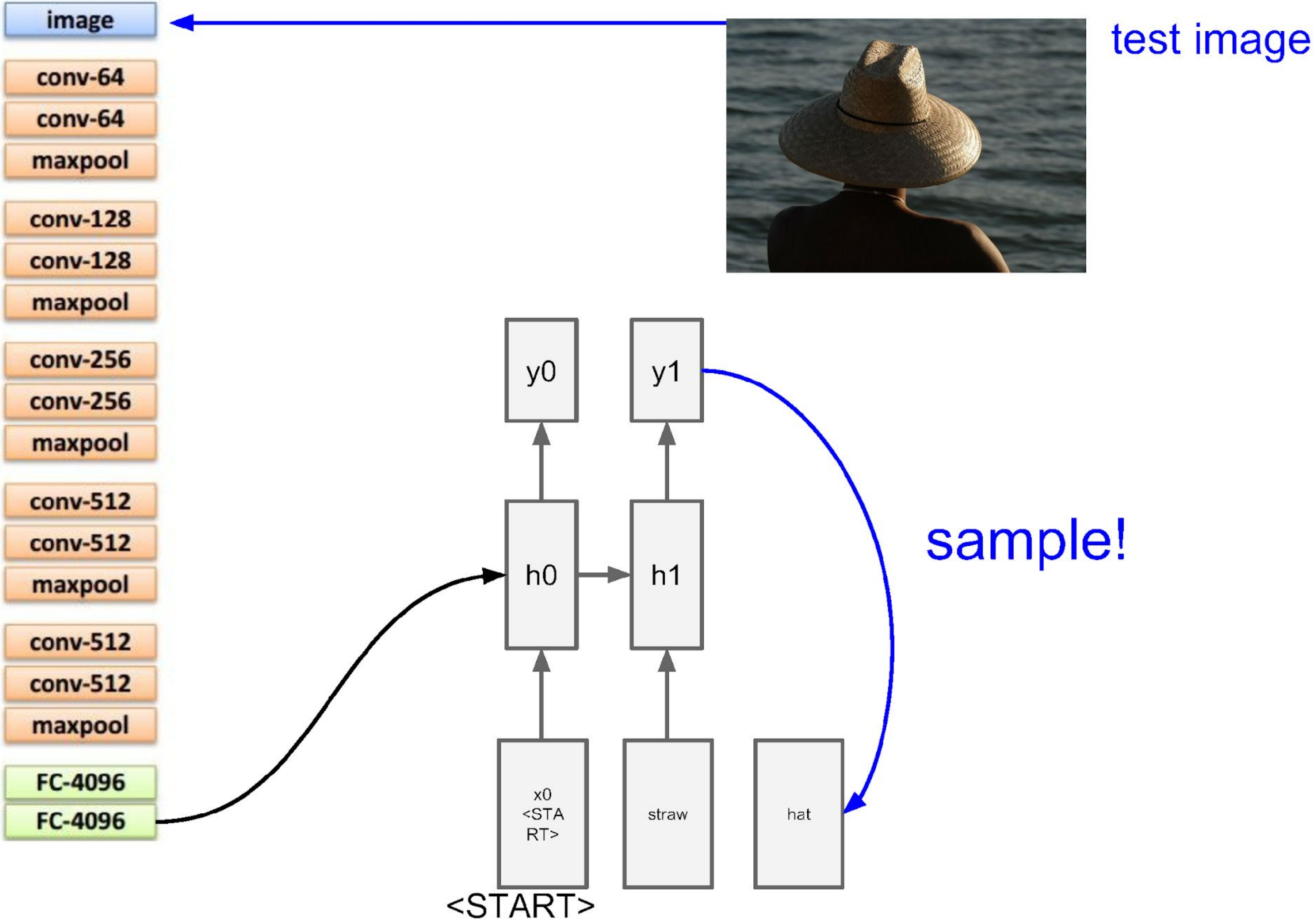


image
conv-64
conv-64
maxpool
conv-128
conv-128
maxpool
conv-256
conv-256
maxpool
conv-512
conv-512
maxpool
conv-512
conv-512
maxpool
FC-4096
FC-4096
test image
y0
y1
h0
h1
x0
<STA
RT>
straw
<START>



image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

test image

y0

y1

h0

h1

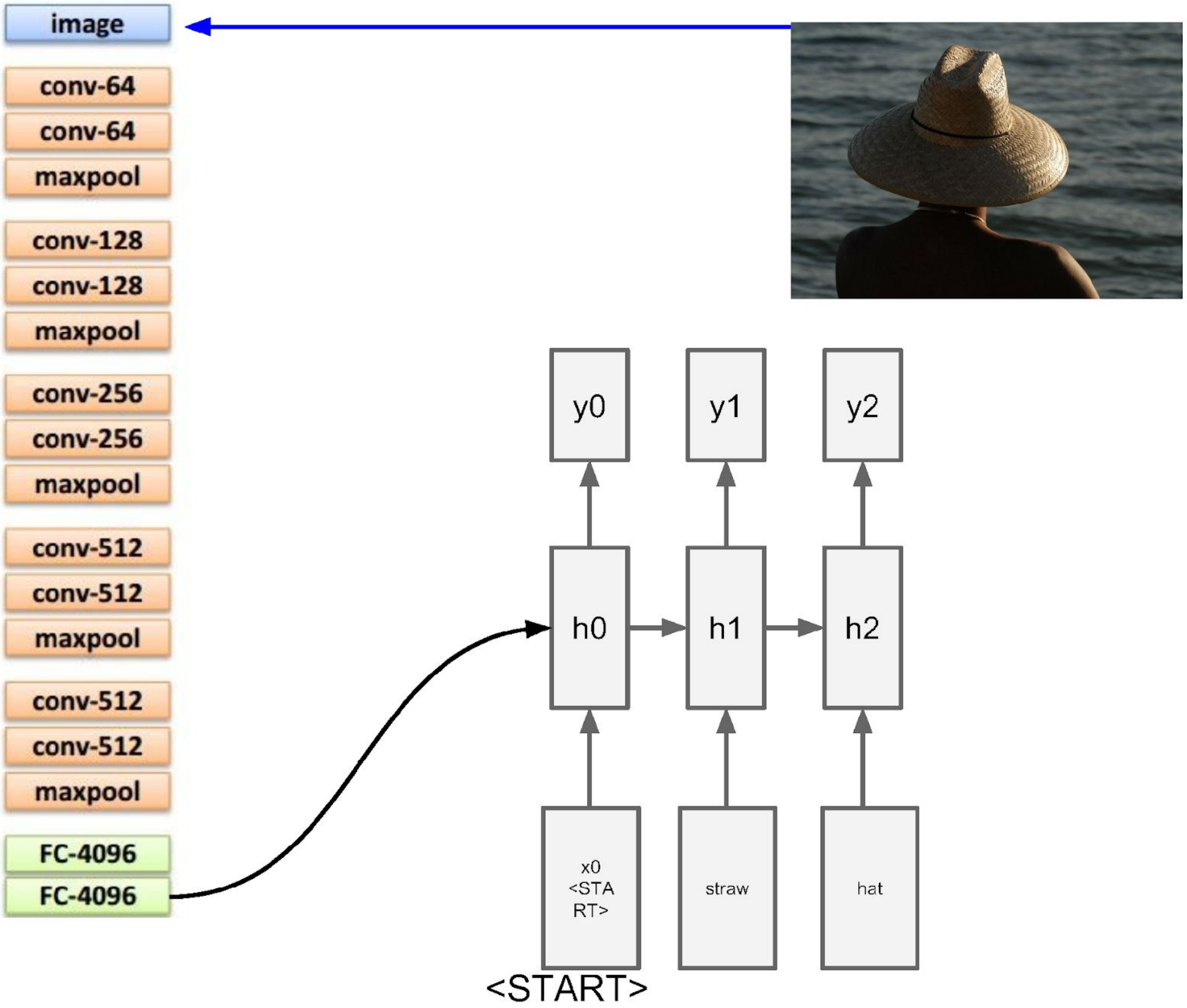
x0 <START>

straw

hat

<START>

sample!



test image

image

conv-64

conv-64

maxpool

conv-128

conv-128

maxpool

conv-256

conv-256

maxpool

conv-512

conv-512

maxpool

conv-512

conv-512

maxpool

FC-4096

FC-4096

y0 y1 y2

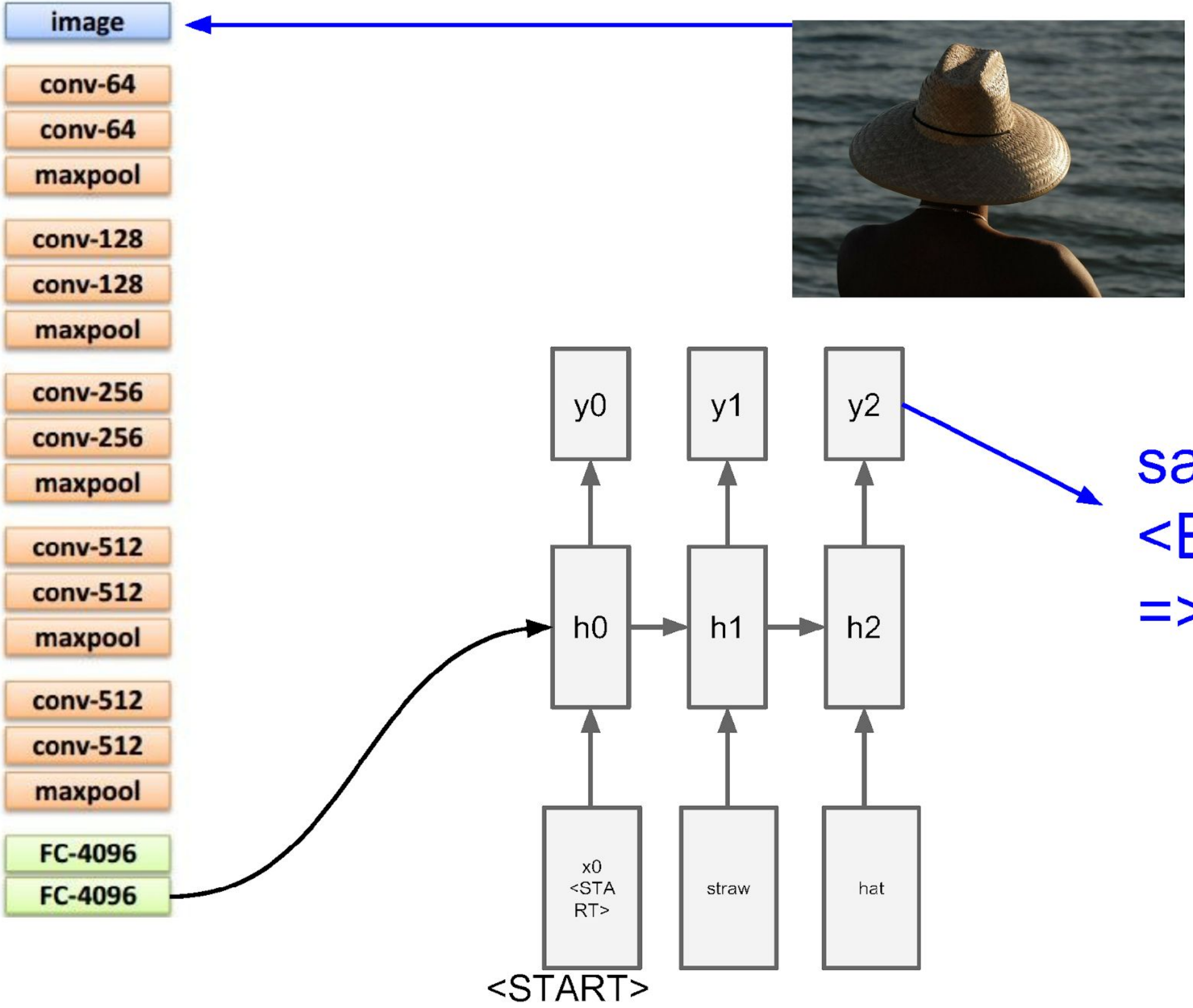
h0 h1 h2

x0 <STA RT>

straw

hat

<START>

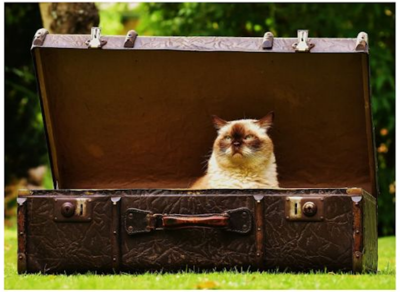


test image

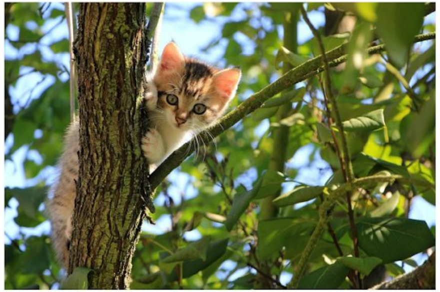
sample
<END> token
=> finish.

Image Captioning: Example Results

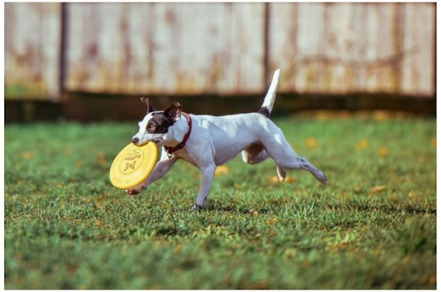
Captions generated using neuraltalk2
All images are CC0 Public domain:
cat suitcase, cat tree, dog, bear,
surfers, tennis, giraffe, motorcycle



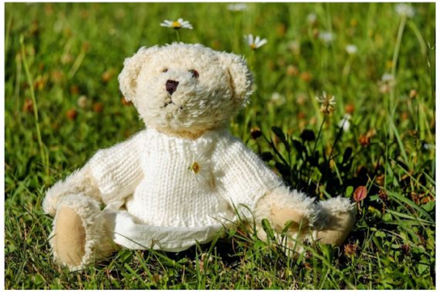
A cat sitting on a suitcase on the floor



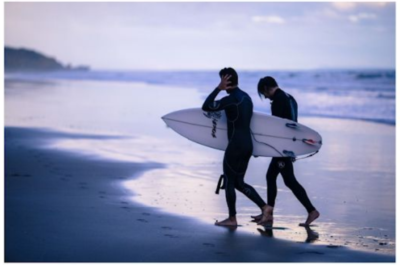
A cat is sitting on a tree branch



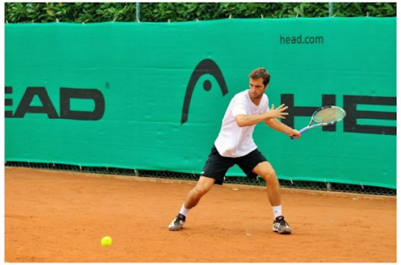
A dog is running in the grass with a frisbee



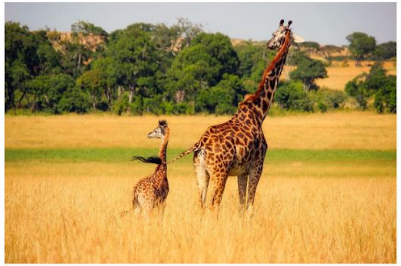
A white teddy bear sitting in the grass



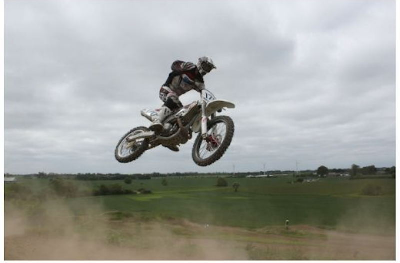
Two people walking on the beach with surfboards



A tennis player in action on the court



Two giraffes standing in a grassy field



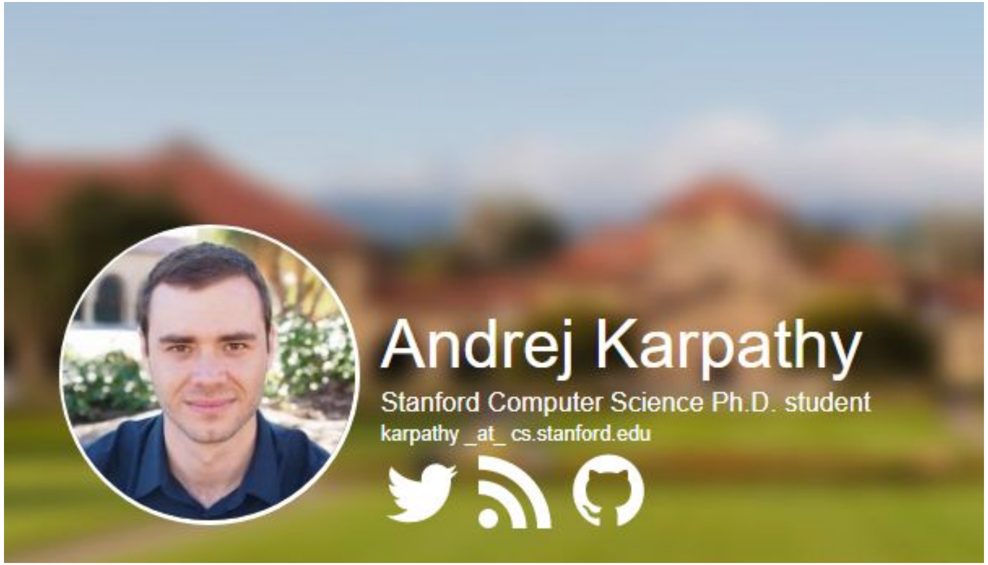
A man riding a dirt bike on a dirt track

Mikolov - Karpathy

Tomas Mikolov

Research Scientist

Facebook AI Research. Previously Google Brain



Andrej Karpathy

Stanford Computer Science Ph.D. student

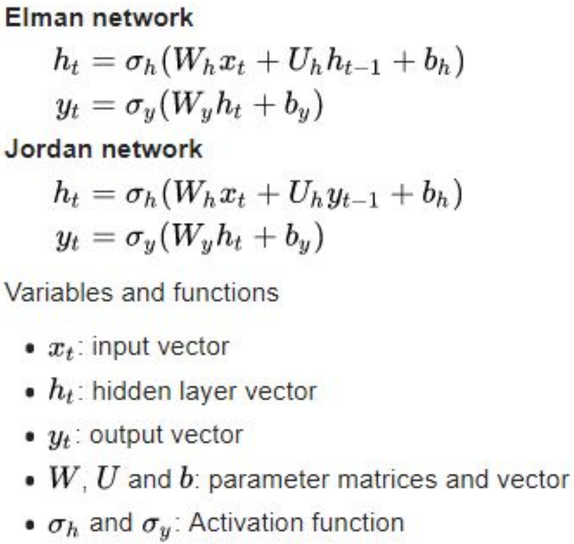
karpathy _at_ cs.stanford.edu

Director of AI at Tesla Neural Networks for the Autopilot

Historical notice

"Simple Recurrent Networks" (SRN) are old

- Elman networks [1990]
- Jordan networks [1997]



Elman network

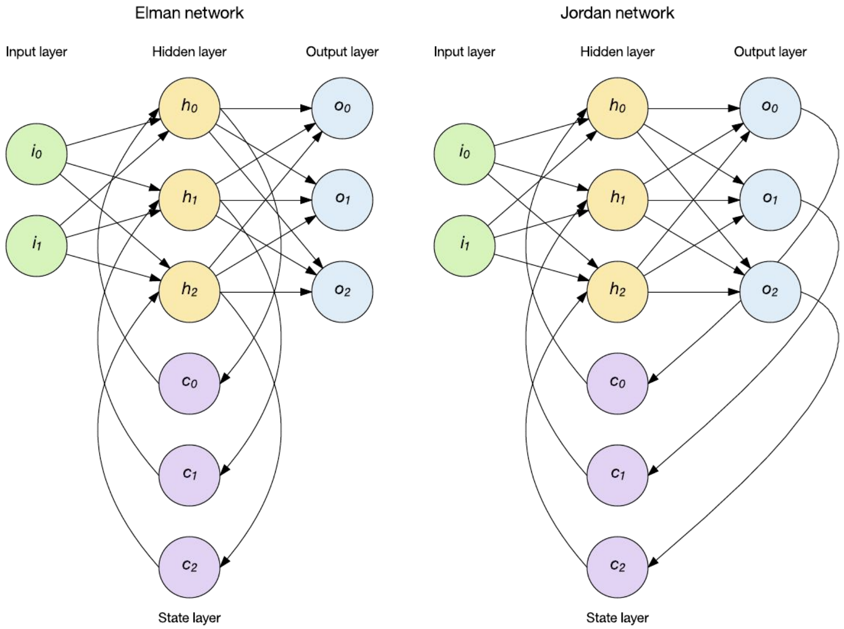
$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

Jordan network

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

Variables and functions

- x_t: input vector
- h_t: hidden layer vector
- y_t: output vector
- W, U and b: parameter matrices and vector
- σ_h and σ_y: Activation function



Bibliography

- Elman, J. L. (1990). Finding structure in time. *Cognitive science*, *14*(2), 179-211.
- Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157-166.
- Williams, R. J., & Zipser, D. (1995). Gradient-based learning algorithms for recurrent. *Backpropagation: Theory, architectures, and applications*, *433*.
- Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. In *Advances in psychology* (Vol. 121, pp. 471-495). North-Holland.
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735-1780.
- Gers, F. A., & Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium* (Vol. 3, pp. 189-194). IEEE.
- Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, *18*(5-6), 602-610.
- Fernández, S., Graves, A., & Schmidhuber, J. (2007, September). An application of recurrent neural networks to discriminative keyword spotting. In *International Conference on Artificial Neural Networks* (pp. 220-229). Springer, Berlin, Heidelberg.
- Pascanu, R., Mikolov, T., & Bengio, Y. (2013, February). On the difficulty of training recurrent neural networks. In *International conference on machine learning* (pp. 1310-1318).
- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.
- Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.
- Karpathy, A., & Fei-Fei, L. (2015). Deep visual-semantic alignments for generating image descriptions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 3128-3137).