Machine Learning in Finance
Section I3
Lecture 12

Michael G Sotiropoulos

NYU Tandon
Deutsche Bank

26-Apr-2019

Contents

Convolutional Neural Nets: Introduction (I)

Convolutional **N**eural **N**etworks (CNNs) are ANNs specialized for grid data

- regular time series are 1D grid data (financial data, speech)
- images are 2D grid data of pixels (pattern recognition)
- video are 3D grid data of pixels (computer vision)

The features that make CNNs better than generic DNNs for grid data are:

- **sparse weights** (not every node is connected to every upstream node)
  this leads to big computational efficiency
- **parameter sharing** (weights are reused by multiple connections)
  this leads to position independent feature extraction
- **equivariant representations** (invariance to small translations of inputs)
  this leads to robust detection of patterns

CNNs have been inspired by human vision, in particular the visual cortex.
The main structural difference between DNNs and CNNs is that the former use matrix multiplication to apply weights (kernels) to inputs, whereas the latter use **convolution**.

Given two functions $x(t)$ (the input) and $w(t)$ (the kernel), their **convolution** $x * w$ is mathematically defined as

$$(x * w)(t) := \int_{-\infty}^{\infty} x(\tau)\, w(t - \tau)\, d\tau = \int_{-\infty}^{\infty} w(\tau)\, x(t - \tau)\, d\tau \qquad (1)$$

The kernel w is flipped (reflected) around the y-axis, and then it slides along t.
For each t the weighted sum of the input with the kernel is formed.
A closely related definition is **cross-correlation**, convolution without kernel flip

$$(x \star w)(t) := \int_{-\infty}^{\infty} x(\tau)\, w(t + \tau)\, d\tau = \int_{-\infty}^{\infty} x(\tau - t)\, w(\tau)\, d\tau \qquad (2)$$

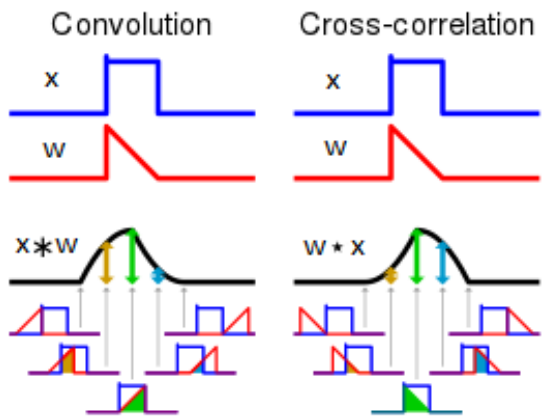


Figure: Convolution and cross-correlation, adapted from Wikipedia.

The discrete analogs of definitions (1) and (2) are

$$(x * w)_i = \sum_{k=-\infty}^{\infty} x_k w_{i-k}, \quad (x \star w)_i = \sum_{k=-\infty}^{\infty} x_k w_{i+k} \tag{3}$$

In two dimensions the operations generalize similarly as

$$(x * w)_{i,j} = \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} x_{k,\ell} w_{i-k,j-\ell}, \ (x \star w) = \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} x_{k,\ell} w_{i+k,j+\ell} \tag{4}$$

- Convolution is symmetric: $x * w = w * x$. Cross-correlation is not.

For neural nets, the symmetry property of convolution is less important, since activation functions that are applied afterwards are themselves not symmetric. CNNs typically implement cross-correlation (i.e. no kernel flip).

In most cases the kernel has smaller size than the input and the preferred "convolution" for CNNs is the one where the running indices are carried by the kernel. For example, in 2D we compute the output signal $s_{i,j}$ as

$$s_{i,j} = (w \star x)_{i,j} = \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} x_{i+k,j+\ell} w_{k,\ell} \tag{5}$$

Receptive field is the number of input nodes that affect a hidden node after the convolution has been applied. The receptive field is initially small (equal to the size of the kernel w) but grows for nodes in deeper convolution layers.

Padding: as the kernel w slides along the input tensor x, it starts rolling off near the end of the input, i.e. it encounters invalid indices. Padding means extending the input past its end (or beginning) by adding zeros. There are three forms of padding (we use a smaller kernel than the input).

1. VALID: no padding. The output tensor will be shorter. If along a given axis the input has size M and the kernel has size K the output has size

$$S_{\text{valid}} = M - K + 1 \tag{6}$$

2. SAME: adding P zeros to the beginning and another P zeros to the end of the input, to maintain the size of the output. The output has size

$$S_{\text{same}} = M - K + 2P + 1 \tag{7}$$

3. CAUSAL: adding zeros to the beginning only and making sure that an output at index t does not depend on input at $t+1, t+2, \ldots$. This is to avoid data snooping into the future, relevant for time series data.

see more in fewer moves

Stride is the number of input indices to skip, as the kernel w is shifted.
In the usual definition of convolution, as in eq. (5), the stride is $S = 1$.
Different input axes may have different strides. For example in 2D

$$s_{i,j} = (w \star x)_{i,j} = \sum_{k=0}^{K} \sum_{\ell=0}^{L} x_{i \cdot s_1 + k,\, j \cdot s_2 + \ell}\, w_{k,\ell}. \tag{8}$$

Strides reduce the size of the output.
With valid padding, the size of the output along an axis with stride S becomes

$$S_{\text{valid}} = \left\lfloor \frac{M - K}{S} + 1 \right\rfloor \tag{9}$$

With same padding of P zeros at the beginning and end of the input

$$S_{\text{same}} = \left\lfloor \frac{M - K + 2P}{S} + 1 \right\rfloor \tag{10}$$

where $\lfloor x \rfloor$ is the integer floor of $x \in \mathbb{R}$ (largest integer not exceeding x).

Feature map is another name for the convolution output (after activation).
Typically several kernels (or **filters**) $w_1, w_2, \ldots w_F$ are applied to the input tensor x, therefore the feature map is a stack $s_{i,j,f} = (w_f \star x)_{i,j}$.

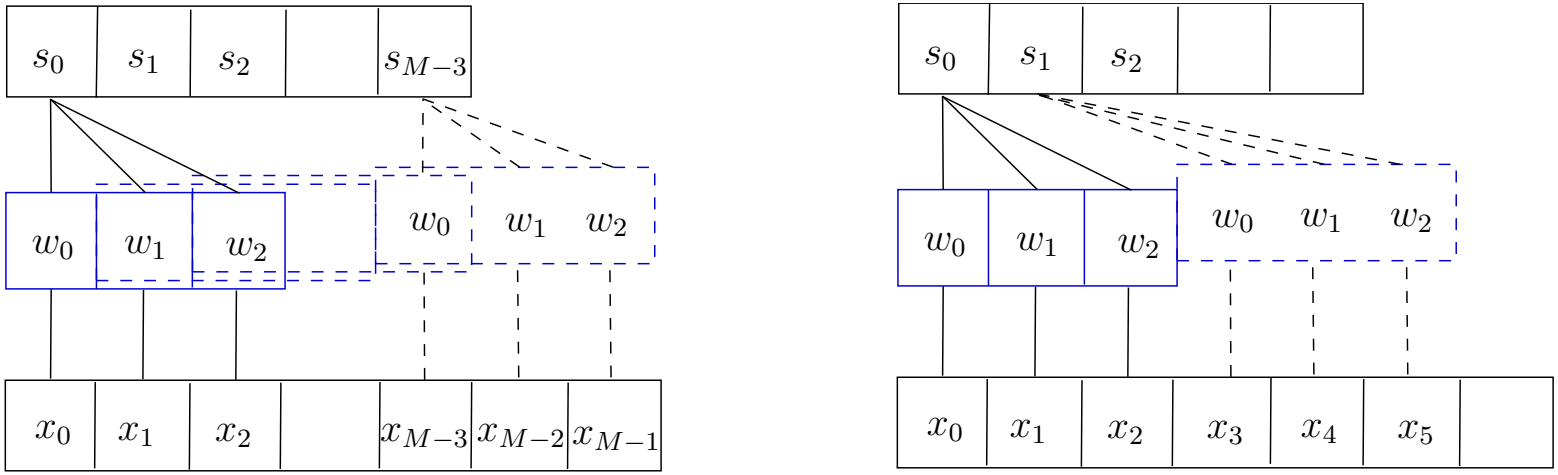


Figure: 1D convolution, *valid* padding (left); valid padding and stride $S = 3$ (right).

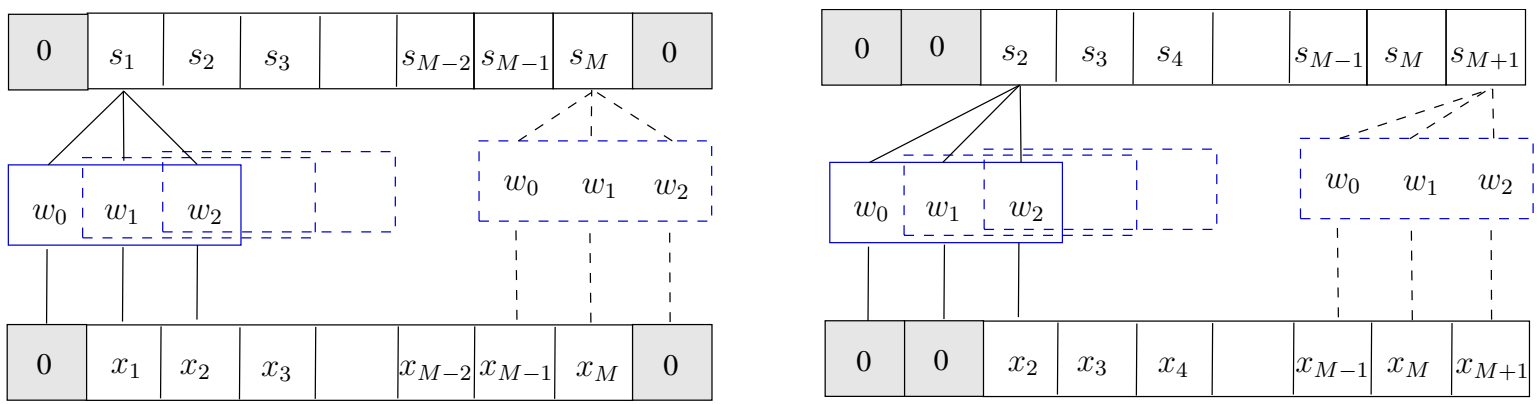


Figure: 1D convolution, *same* padding (left) and *causal* padding (right).

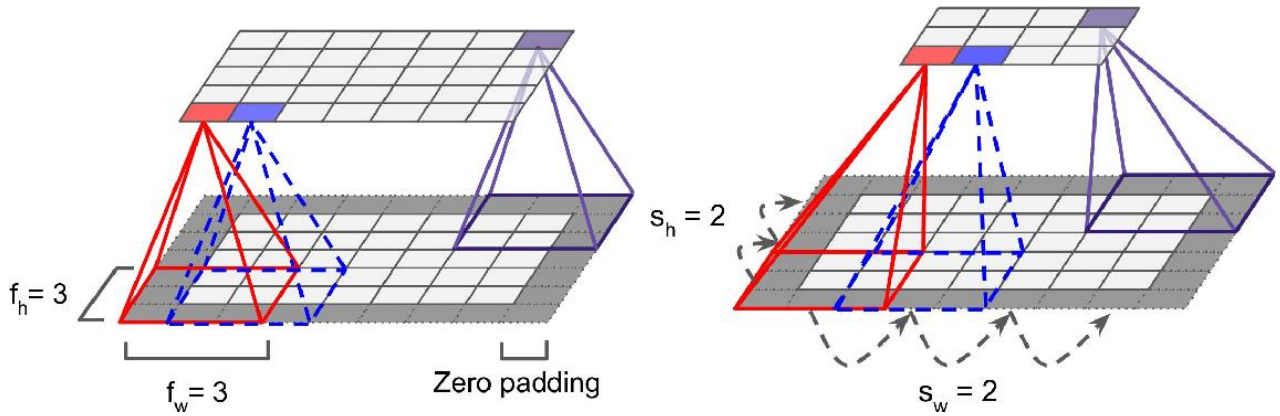


Figure: 2D convolution with a 3×3 kernel and same padding (left). The same convolution but with stride $S = 2$ (right). From [GA]

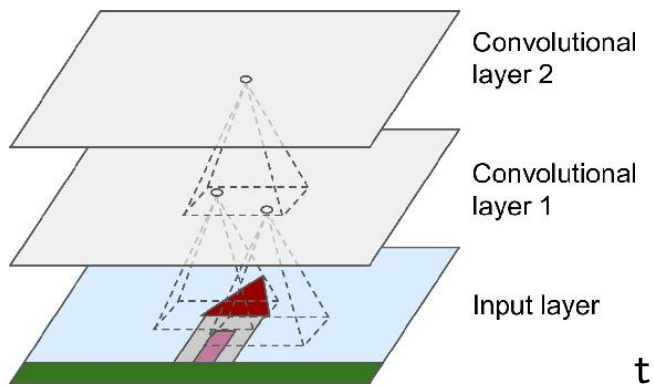


Figure: Two layer 2D convolution using filters of same size. The receptive field increases at deeper layers. From [GA]

Pooling is the operation that downsamples (shrinks) the feature map.
Pooling is very common in computer vision because

- it reduces the computational and memory load
- it leads to equivariant representations (robustness to small shifts/rotations)

A pooling layer does not introduce any new parameters to fit.
Like a convolutional layer, it is defined by size, padding type and stride.

- Max/average pooling outputs the maximum/average of its input nodes.
- Max pooling is the most common in CNNs

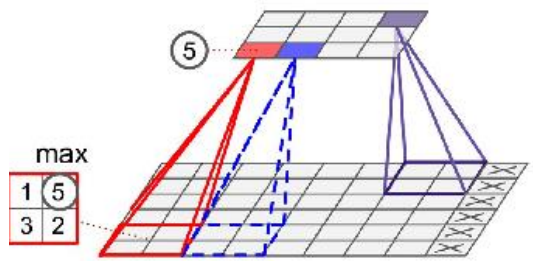


Figure: Max pooling with a 2x2 kernel, stride 2 and valid padding. From [GA].

Finally, **dilation** is the operation of increasing the size (stretching) the filter by a fixed factor. This is done by inserting zeros between the kernel elements. The objective is to increase the receptive field.

Putting it all together, a sequential CNN architecture looks like in the figure.

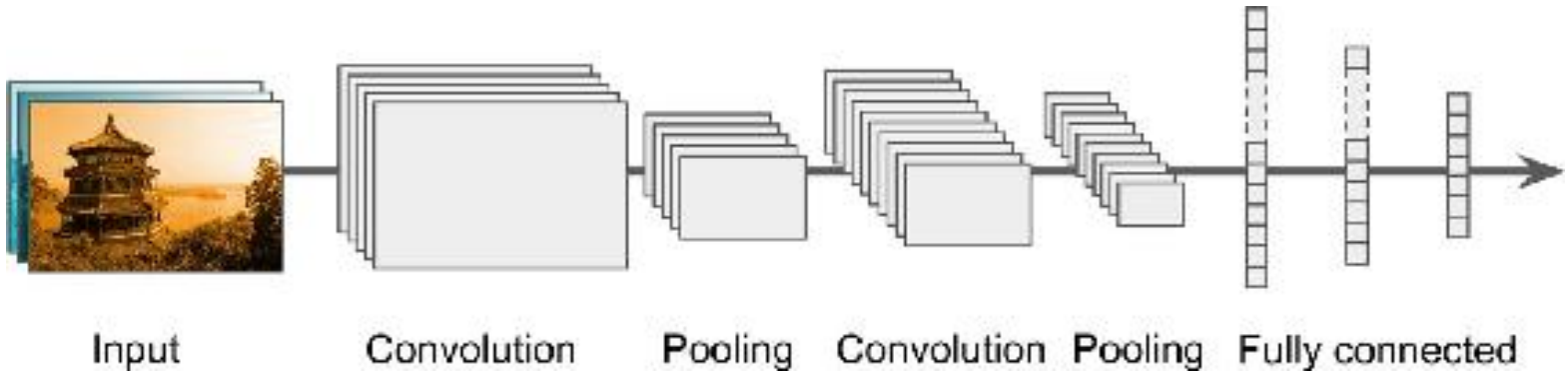


Figure: Sequential CNN architecture. From [GA]

The input is a tensor of 1D indexed data (time series), 2D indexed data (images) or 3D indexed data (volumetric, video).
Besides the axes that define the index, there are axes for the features.
A time series of returns and trading volumes will be a rank 3 tensor $x_{t,p,q}$
A color image, will be a rank-5 (width, height, three color channels).

The feature map is typically flattened near the output, and passed to a DNN that generates the final fitted/forecasted values.

Convolutional Neural Nets: In Keras (I)

1D convolution is implemented in the class `keras.layers.Conv1D`.
Important constructor parameters are

- `filters` number of kernels to apply
- `kernel_size` number of nodes per kernel
- `strides` the stride number
- `padding` "valid" or "same" or "causal"
- `data_format` "channels_last" or "channels_first"
- `dilation_rate` number between 0 and 1.
  Do not change from default 1, if stride is bigger than 1.

2D convolution is implemented in the class `keras.layers.Conv2D`.
Constructor parameters are the same, except the shape of kernel specific parameters changes from single number to tuples of two numbers.
These parameters are `kernel_size`, `strides` and `dilation_rate`

3D convolution is implemented in the class `keras.layers.Conv3D`.

The output of a CNN can be flattened to a single dimension, suitable for passing to a DNN. This can be done using the class `keras.layers.Flatten`.

For details see the notebook **L12-CNN-RNN-Examples.ipynb**

The ANNs we studied so far (DNN or CNN) are **Feed Forward** networks. Given a mini-batch of inputs $\boldsymbol{x}^{(t)}$, they compute hidden layers and output by moving the information forward through the hidden layers from input to output

$$\boldsymbol{h}_1^{(t)} = \phi_1\left(\boldsymbol{x}^{(t)} \cdot \boldsymbol{W}_1^{(t)} + \boldsymbol{b}_1^{(t)}\right), \ldots \hat{\boldsymbol{y}}^{(t)} = \phi_K\left(\boldsymbol{h}_{k-1}^{(t)} \cdot \boldsymbol{W}_k^{(t)} + \boldsymbol{b}_k^{(t)}\right). \tag{11}$$

Then, they fit the weights by minimizing the loss $L\left(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}\right)$. The only communication between mini-batches is via the currently estimated weights.

Inspired by Markov chains and ARMA models, we generalize this concept, so that the output at iteration t also depends on prior outputs

$$\hat{\boldsymbol{y}}^{(t)} = g\left(\boldsymbol{x}^{(t)}, \hat{\boldsymbol{y}}^{(t-1)}; \boldsymbol{W}^{(t)}, \boldsymbol{b}^{(t)}\right) \tag{12}$$

This makes the output $\hat{\boldsymbol{y}}^{(t)}$ depend on the entire prior input history

$$\hat{\boldsymbol{y}}^{(t)} = f\left(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t-2)} \ldots\right) \tag{13}$$

Dependence on prior history can be implemented with hidden layers that retain memory across mini-batches.

- These layers are called **recurrent layers** or **memory cells**.

A **Recurrent Neural Network** (RNN) is an ANN with one or more memory cells.

Practically, an RNN is an ANN with internal loops that feed layer output back to the layer. An RNN can be represented in loop or unfolded format.

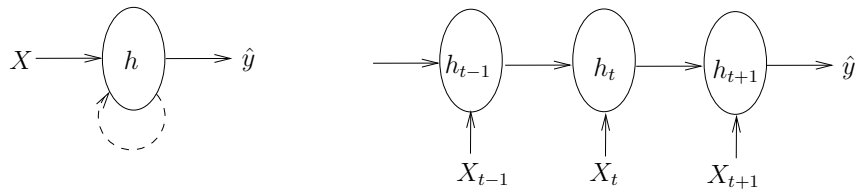


Figure: RNN in loop and unfolded format.

RNNs can also be classified based on the structure of their inputs and outputs

- When both the input and output are sequences (like in time series forecasting) this is a **sequence-to-sequence** RNN
- When the input is a sequence and the output is one vector, i.e. the value at the last step of the sequence, this is a **sequence-to-vector** or **encoder** RNN.
- When the input is a vector and the output is a sequence (like tagging an image by a phrase) this is a **vector-to-sequence** or **decoder** RNN.

There is a wide variety of RNN designs, but they all share the recurrence feature (feedback loop, or memory summarized by a state variable).

In a **simple RNN** the output is fed back to the hidden layer

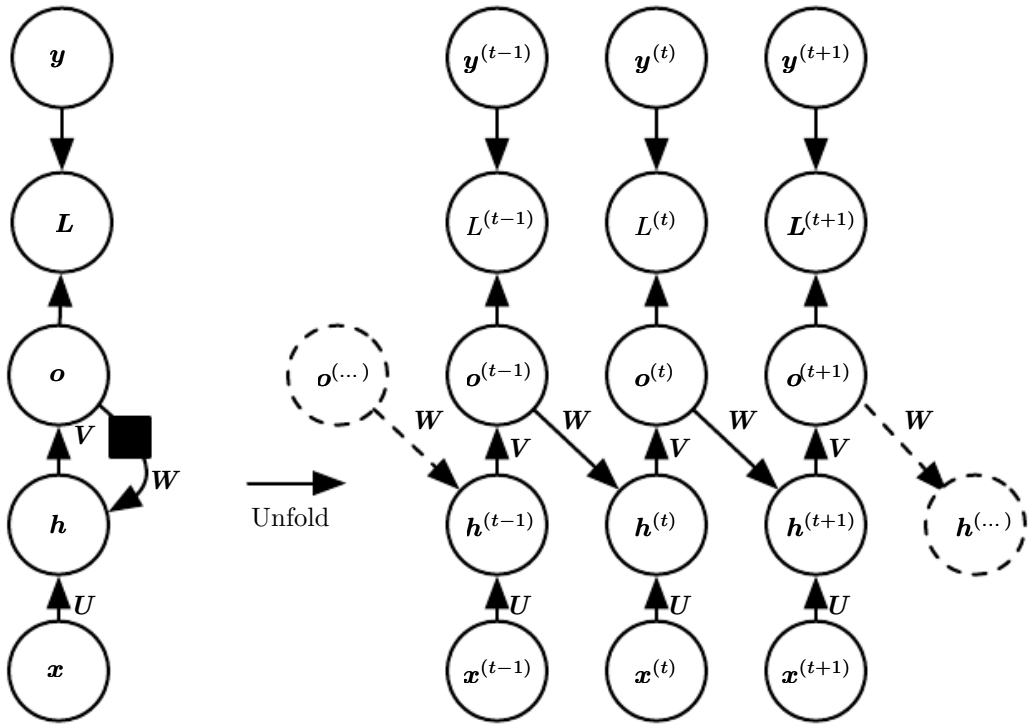


Figure: Simple RNN with output $o^{(t)} := \hat{y}^{(t)}$ fed back to the hidden layer. U are the input-to-hidden weights, V are the hidden-to-output weights and W are the output-to-hidden recurrent weights. From [GBC].

In general, the state fed back to the layer is different from the layer output. The state fed back to layer $h^{(t)}$ is typically denoted by $s^{(t-1)}$.

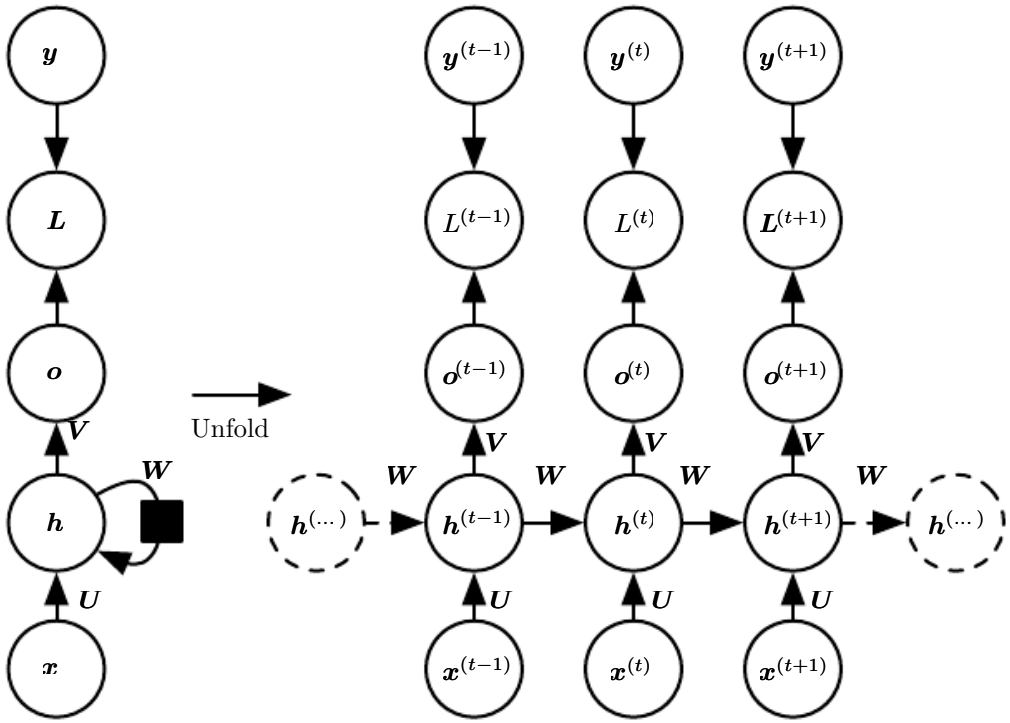
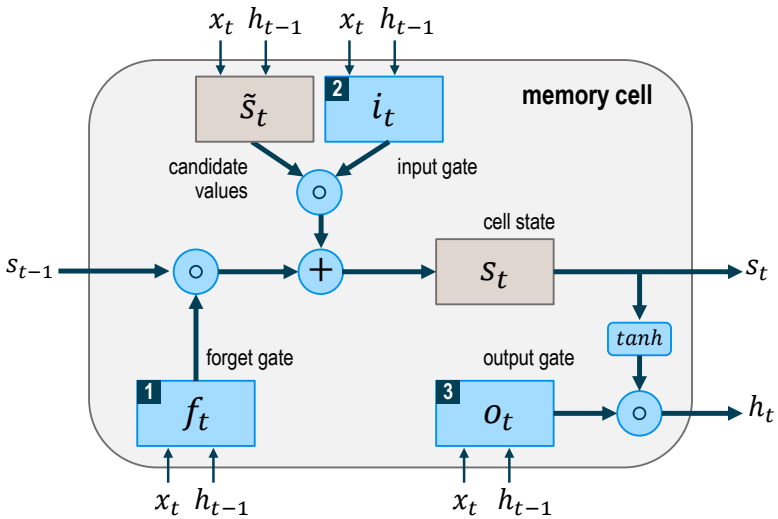


Figure: RNN with state $s^{(t-1)}$ fed back to the hidden layer $h^{(t)}$. U are the input-to-hidden weights, V are the hidden-to-output weights, and W are the state-to-hidden recurrent weights. From [GBC].

The state of the art in memory cells is the Long Short-Term Memory or LSTM cell (Hochreiter 1997).
The LSTM cell contains three "gates", forget (f), input (i) and output (o).

- A gate is a computational node where the current input x_t and the prior layer value h_{t-1} are taken as inputs.
- The cell takes three inputs, x_t, h_{t-1} and the prior state s_{t-1} and computes two outputs s_t and h_t.



1 Forget gate:
Defines which information to remove from the memory (cell state)

2 Input gate:
Defines which information to add to the memory (cell state)

3 Output gate:
Defines which information from the memory (cell state) to use as output

Figure: LSTM Cell Anatomy. The nodes with a circle denote element-wise multiplication and with plus element-wise addition. From [FK].

The update rules are

$$\tilde{s} = \tanh\left(\boldsymbol{x}_t \cdot \boldsymbol{W}^{(x\tilde{s})} + \boldsymbol{h}_{t-1} \cdot \boldsymbol{W}^{(h\tilde{s})} + \boldsymbol{b}^{(\tilde{s})}\right) \tag{14}$$

$$i_t = \sigma\left(\boldsymbol{x}_t \cdot \boldsymbol{W}^{(xi)} + \boldsymbol{h}_{t-1} \cdot \boldsymbol{W}^{(hi)} + \boldsymbol{b}^{(i)}\right) \tag{15}$$

$$f_t = \sigma\left(\boldsymbol{x}_t \cdot \boldsymbol{W}^{(xf)} + \boldsymbol{h}_{t-1} \cdot \boldsymbol{W}^{(hf)} + \boldsymbol{b}^{(f)}\right) \tag{16}$$

$$o_t = \sigma\left(\boldsymbol{x}_t \cdot \boldsymbol{W}^{(xo)} + \boldsymbol{h}_{t-1} \cdot \boldsymbol{W}^{(ho)} + \boldsymbol{b}^{(o)}\right) \tag{17}$$

$$s_t = f_t \otimes s_{t-1} + i_t \otimes \tilde{s}_t \tag{18}$$

$$h_t = o_t \otimes \tanh\left(s_t\right) \tag{19}$$

Parameter count:
An LSTM cell processing M features (length of x_t) and H units per hidden layer (length of h_t) introduces the following number of parameters

$$4MH + 4H^2 + 4H = 4\left(H\left(M+1\right) + H^2\right) \tag{20}$$

The LSTM cell can be used just like a regular RNN layer (no need to look inside the box). It offers the following advantages

1. It recognizes important features (input gate), stores them as long as needed (forget gate) and propagates forward the relevant parts (output gate).
2. It converges faster then standard RNNs.
3. It has been very successful in learning long-term patterns in time series and audio signals.

A simplified version of LSTM is the Gated Recurrent Unit or GRU (Cho et al. 2014).

- Input and forget gates are collapsed into a single control gate. The control outputs 0/1 to close/open the input gate and open/close the forget gate
- There is no separate output gate. The full state vector is propagated forward.

GRU's are gaining popularity because of their simplicity and efficiency.

The simple RNN layer, where output is fed back to the input, is implemented in the class `keras.layers.SimpleRNN`.
Many of the constructor parameters are the same as with `Dense` layer.
New constructor parameters relative to `Dense` are

- `recurrent_initializer`, `recurrent_regularizer`
- `recurrent_activation`, `recurrent_dropout`
- `stateful:` if True, the last state will be reused for the next batch
- `unroll:` if True, the network will be unrolled, suitable only for short sequences because it is memory intensive
- `return_sequences:` if True, it will return the full sequence of outputs, not just the last value

The LSTM layer is implemented in the class `keras.layers.LSTM`
It has all the constructor parameters of `SimpleRNN`.

Finally the GRU is implemented in the class `keras.layers.GRU`

For details see the notebook **L12-CNN-RNN-Examples.ipynb**

Further Reading

1. Read chapters 13 and 14 in [GA] "Convolutional" and "Recurrent Neural Networks". No need to focus on TensorFlow implementation details.
2. Read chapters 9 and 10 in [GBC] for a deeper understanding of the mathematics behind convolutional and recurrent networks.

References

[GA] Géron Aurélien. "Hands-On Machine Learning with Scikit-Learn and TensorFlow". O'Reilly, 2017

[GBC] Goodfellow I., Bengio Y., and Courville A. "Deep Learning". MIT, 2016

[FK] Fischer T. and Krauss C. "Deep learning with long short-term memory networks for financial market predictions". European Journal of Operational Research, 2017.