Deep learning

Course of Machine Learning
Master Degree in Computer Science

University of Rome "Tor Vergata"

a.a. 2019-2020

Giorgio Gambosi

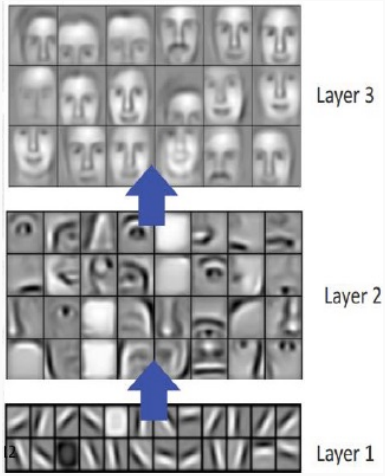
Deep networks

The term **deep** neural networks usually refers to the ones including at least 2 hidden layers (up to several tens).

Model complexity (number of coefficients to be learned) depends from several variables:

- depth (number of layers)
- width (number of units) of each layer
- link topology
- sharing of coefficient values between links

Deep learning



The increase of complexity implies:

- need of larger training sets
- more expensive steps of forward and backpropagation

Many types of deep networks

- **MLP** Multilayer perceptron
- **CNN** Convolutional neural network
- **AE** Auto encoder
- **RNN** Recurrent neural network
- **LSTM** Long-short term memory network
- ...

Learning in deep networks

Learning of coefficients in deep network follows the general approach of defining:

- Loss function
- Optimization method

Moreover, the choice of the activation function has to be considered

Loss functions

Classification

- Hinge: $L_i = \sum_{j \neq y_i} \max\left(0, f_j - f_{y_i} + 1\right)$
- Squared hinge: $L_i = \sum_{j \neq y_i} \max\left(0, f_j - f_{y_i} + 1\right)^2$
- Cross-entropy: $L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$

Loss functions

Regression

- L2 norm: $L_i = \|f - y_i\|_2^2 = \sum_j (f_j - (y_i)_j)^2$
- L1 norm: $L_i = \|f - y_i\|_1 = \sum_j \mid f_j - (y_i)_j \mid$

Regularization

There are several ways of controlling the capacity of Neural Networks to prevent overfitting:

- **L2 regularization**
- **L1 regularization**
- **Max norm constraints**. Enforce an absolute upper bound on the magnitude of the weight by clamping the weight vector w of every neuron to satisfy $||w|| < c$.
- **Dropout**. A neuron is kept active with some probability p (a hyperparameter), or setting it to zero otherwise.

Vanishing gradient

Appying the sigmoid function in a NN hidden units is a critical issue since derivative values

$$\frac{\partial E_i}{\partial w_{jl}^{(k)}}$$

tend to monotonically decrease, in module, during backpropagation.
This is due to the fact that

$$\frac{\partial E_i}{\partial w_{jl}^{(k)}} = \delta_j^{(k)} z_l^{(k-1)}$$

and

$$\delta_j^{(k)} = h_k'(a_j^{(k)}) \sum_{r=1}^{n_{k+1}} \delta_r^{(k+1)} w_{rj}^{(k+1)} = \sigma(a_j^{(k)})(1 - \sigma(a_j^{(k)})) \sum_{r=1}^{n_{k+1}} \delta_r^{(k+1)} w_{rj}^{(k+1)}$$

where, necessarily, it is $\sigma(a_j^{(k)})(1 - \sigma(a_j^{(k)})) < \frac{1}{4}$.

Vanishing gradient

In particular, if the value of $\sigma(a_j^{(k)})$ is near to 0 or to 1 this implies

$$\delta_j^{(k)} \simeq 0$$

with the consequence that

$$\frac{\partial E_i}{\partial w_{jl}^{(k)}} \simeq 0$$

and the effect spreads to all

$$\delta_j^{(k-1)} = \sigma(a_j^{(k-1)})(1 - \sigma(a_j^{(k-1)})) \sum_{r=1}^{n_k} \delta_r^{(k)} w_{rj}^{(k)}$$

Vanishing gradient

Using a different activation function such as **RELU** (Rectified Linear Unit)

$$h(x) = \max(0, x)$$

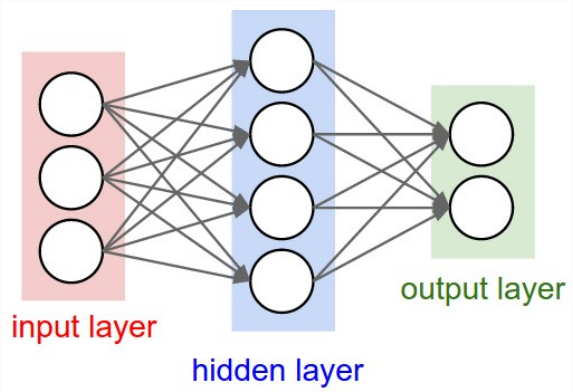
- vanishing gradient tends to be less likely ($h'(x) = 1$ if $x > 0$)
- returns sparser solutions (more null coefficients), since $h'(x) = 0$ if $x < 0$: increased efficiency

Convolutional Neural Networks

Very similar to ordinary Neural Networks

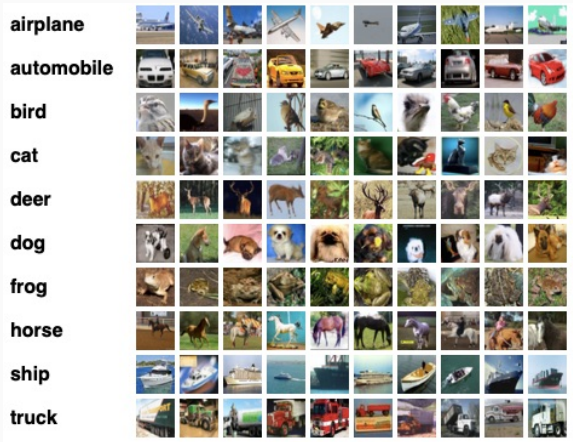
- made up of neurons that have learnable weights and biases
- each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity
- the whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other
- they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks developed for learning regular Neural Networks still apply

Neural networks issue with images



- Neural Networks receive an input (a single vector), and transform it through a series of *hidden layers*
- Each hidden layer is made up of a set of neurons, each neuron fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections
- The last fully-connected layer is called the *output layer* and in classification settings it represents the class scores.

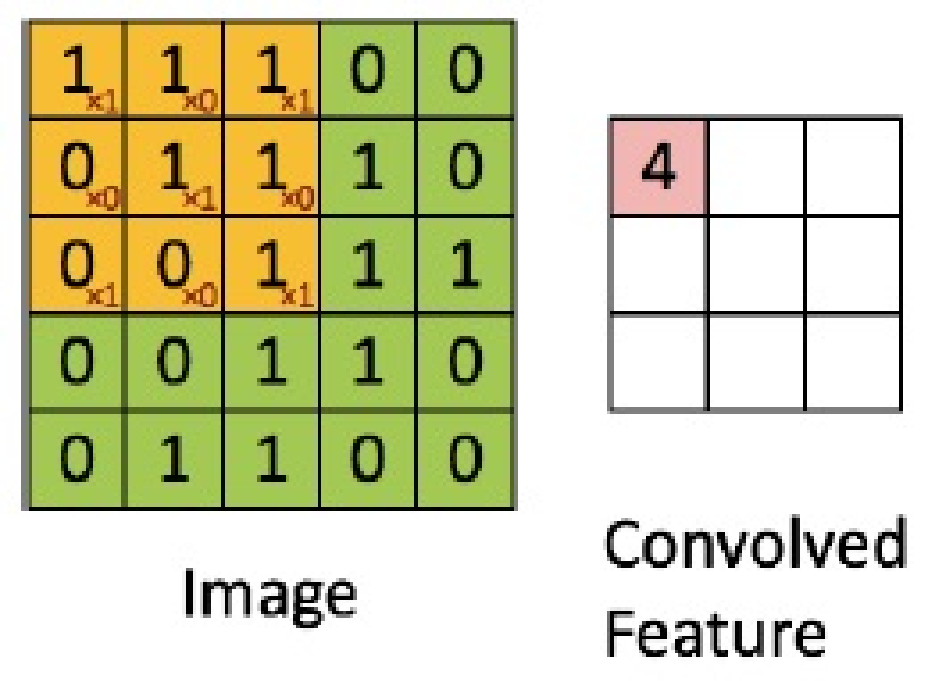
Neural networks issue with images



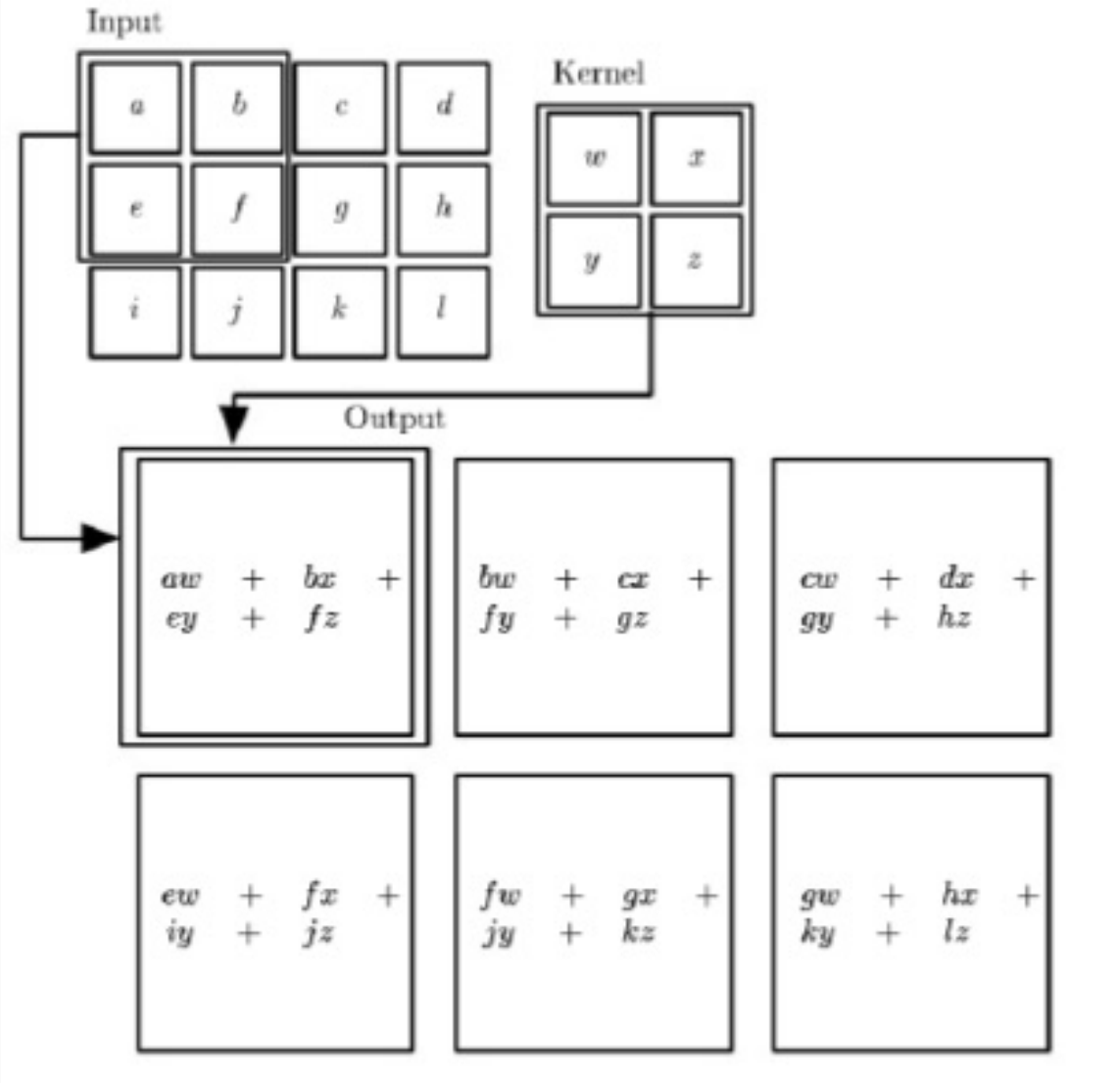
In CIFAR-10, images are only of size 32x32x3 (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer of a regular Neural Network would have 32*32*3 = 3072 weights. But, an image of size 200x200x3 would result into a first layer where each neuron has 120000 weights.
Clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

Convolutional Neural Networks

Usually applied to images. They learn features through a *convolution* operation, with a filter (coefficient matrix) moving on the output of the previous layer

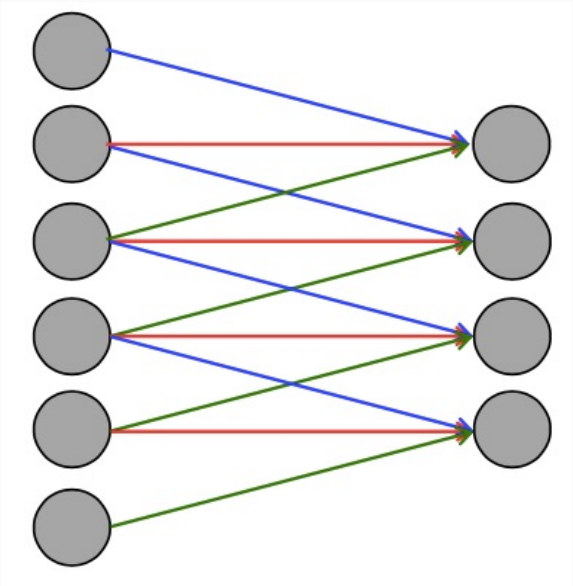


Convolutional Neural Networks



Local connections

Neurons in a layer are connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner.



ConvNet structure

Neurons in a ConvNet are arranged in 3 dimensions: **width, height, depth**.

For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively).
In a ConvNet, units in a layer are (conceptually) organized in a 3d volume

Layers used to build ConvNets

A simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function.
Three main types of layers:

- **Convolutional Layer**,
- **Pooling Layer**
- **RELU Layer**
- **Fully-Connected Layer**

CONV/FC layers perform transformations on the basis of their parameters (weights and biases of neurons), to be learned by gradient descent.
RELU/POOL layers implement a fixed function.

Example

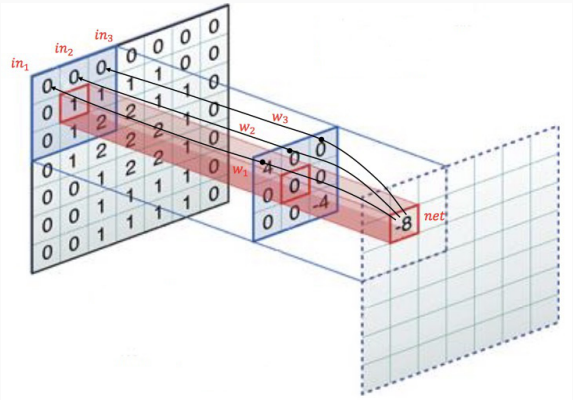
Simple ConvNet for CIFAR-10 classification [INPUT - CONV - RELU - POOL - FC] archietecture.

- INPUT [32x32x3] holds the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- CONV layer computes the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters.
- RELU layer applies an elementwise activation function, such as the $(max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]).
- POOL layer performs a downsampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12].
- FC (i.e. fully-connected) layer computes the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In summary

- ConvNet architecture as list of Layers that transform the image volume into an output volume (e.g. holding the class scores)
- A few distinct types of Layers (e.g. CONV/FC/RELU/POOL)
- Each Layer accepts an input 3D volume and transforms it to an output 3D volume through a differentiable function
- Each Layer may or may not have parameters
- Each Layer may or may not have additional hyperparameters

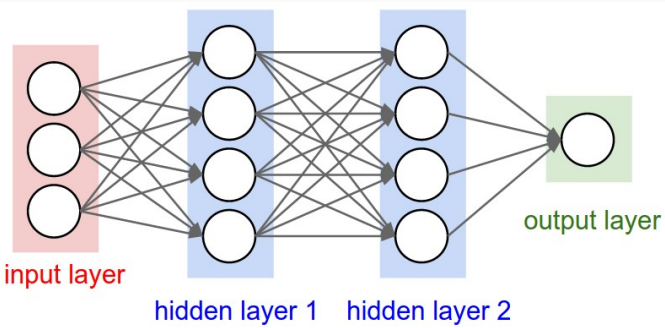
Convolutional Layer

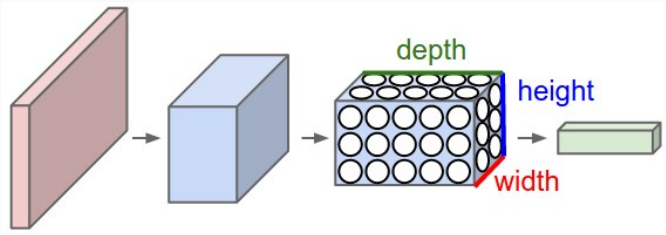


Filter coefficients are common for all layer units: each unit performs the same operations on a different region of the output of the previous layer. This is equivalent to a convolution of the filter and the output.

Convolutional Layer

The r-th convolutional layer arranges its neurons in three dimensions (width, height, depth).
Assume layer r returns as output a volume $w \times h \times d$ ("images" of size $w \times h$, with each "pixel" characterized by d values, related to d different features).





- On each width-height layer the bidimensional structure of the image is reported
- Each layer on the depth dimension corresponds to a different feature computed from the output of the previus network layer

Then, every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations

Convolutional Layer

The structure of layer $r+1$ (and of its output) is characterized by two hyperparameters:

- **Depth**: number of layers (features) to be derived. It corresponds to the number of filters to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. Neurons that are all looking at the same region of the input are denoted as **depth column**

Convolutional Layer

- **Stride**: step of the filter slide over the output of layer r; that is, increment, from a unit to the adjacent one, of the position of the central pixel of the window on which the filter is applied. When the stride is 1 filters are moved one pixel at a time. When the stride is k then the filters jump k pixels at a time as they are slided around. This will produce smaller output volumes spatially.

Depth is fixed as an initial parameter.

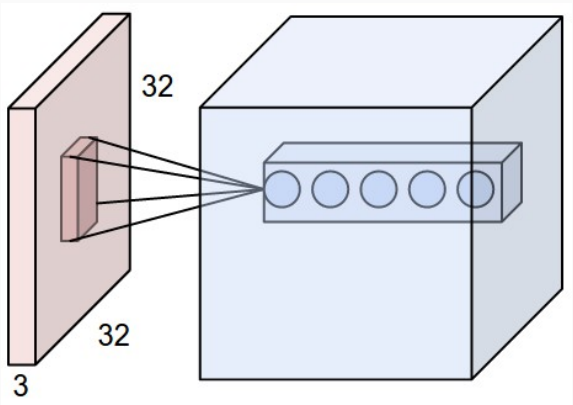
Convolutional Layer

If *stride*$= 1$, then *width* and *height* of layer $r + 1$ are respectively equal to w and h. The case of window on the border can be managed by assuming a suitable "frame" of null values surrounding the image (**zero-padding**) returned by layer r. The size of this *zero-padding* is a hyperparameter.
In the opposite case, they will be given by the ratio of w (and h) with the stride value.

Convolutional Layer

Thus, the spatial size of the output volume is a function of the input volume size (w), the filter size (f), the stride with which they are applied (s), and the amount of zero padding used (p) on the border.
The correct formula for calculating how many neurons "fit" is given by $\frac{w-f+2p}{s} + 1$. For example for a 7x7 input and a 3x3 filter with stride 1 and pad 0 we would get a 5x5 output. With stride 2 we would get a 3x3 output.

Convolutional Layer



Let the input to layer $r+1$ be a volume $32 \times 32 \times 3$.
Assume a filter window of size 3×3: each unit in layer $r+1$ has $3 \times 3 \times 3 + 1 = 28$ values as input, including bias. Assuming a stride $s = 2$, each bidimensional slice at layer $r+1$ has size 16×16.
If 5 features are exctracted the number of units is then $16 \times 16 \times 5 = 1280$.

Convolutional Layer

- Real-world architecture (winner of the ImageNet challenge in 2012). Accepted images of size $[227 \times 227 \times 3$. 1.3 million high-resolution images in the training set; 1000 different classes.
- On the first Convolutional Layer, neurons with $f = 11$, stride $s = 4$ and no zero padding $p = 0$; depth $K = 96$. Since (227 - 11)/4 + 1 = 55, the Conv layer output volume had size $[55 \times 55 \times 96]$.
- Each of the $55 * 55 * 96$ neurons in this volume was connected to a region of size $[11 \times 11 \times 3]$ in the input volume.
- All 96 neurons in each depth column are connected to the same $[11 \times 11 \times 3]$ region of the input, with different weights.

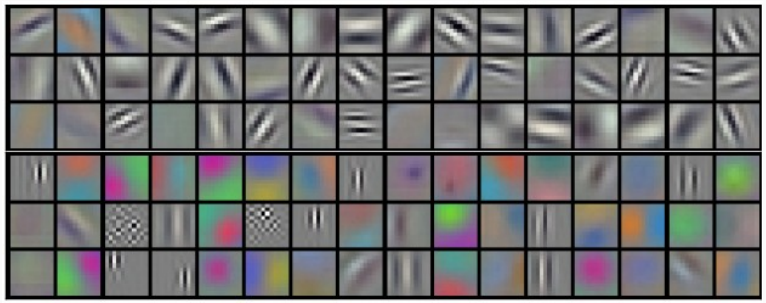
Convolutional Layer

In the real-world example above, there are $55 \times 55 \times 96 = 290.400$ neurons in the first Conv Layer, each with $11 \times 11 \times 3 = 363$ weights plus 1 bias. This adds up to $290400 \times 364 = 105.705.600$ parameters on the first layer of the ConvNet alone. Clearly, this number is very high.

Reasonable assumption: if one feature has a certain usefulness to compute at some spatial position (x, y), then it should have the same usefulness to compute at a different position (x', y').
As a consequence, neurons in a same bi-dimensional layer (**depth slice**) use the same weights and bias. With this parameter sharing scheme, the first Conv Layer in the example would now have only 96 unique set of weights (one for each depth slice), for a total of $96 \times 11 \times 11 \times 3 = 34.848$ unique weights, or 34.944 parameters ($+96$ biases). All 55×55 neurons in each depth slice use the same parameters.

Convolutional Layer

Example filters learned in a real world example



Convolutional Layer

Sometimes the parameter sharing assumption has to be relaxed (**Locally-Connected Layer**).

This is especially the case when we should expect that completely different features should be learned on one side of the image than another. Example: input are faces that have been centered in the image. You might expect that different eye-specific or hair-specific features could (and should) be learned in different spatial locations.

Convolutional Layer

To summarize, a Conv Layer:

- Accepts a volume of size $w_1 \times h_1 \times d_1$
- Requires four hyperparameters:
  - Number of filters k
  - their spatial extent f
  - the stride s
  - the amount of zero padding p
- Produces a volume of size $w_2 \times h_2 \times d_2$ where:
  - $w_2 = \frac{w_1 - f + 2p}{s} + 1$
  - $h_2 = \frac{h_1 - f + 2p}{s} + 1$
  - $d_2 = k$
- With parameter sharing, it introduces $f \times f \times d_1 + 1$ weights per filter
- In the output volume, the i-th depth slice (of size $w_2 \times h_2$) is the result of performing a valid convolution of the i-th filter over the input volume with stride s, and then offset by i-th bias.

A common setting of the hyperparameters is $f = 3, s = 1, p = 1$. There are common conventions and rules of thumb that motivate these hyperparameters.

Pooling Layer

A pooling layer aggregates output data from the preceding layer. It progressively reduces the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting.

The aggregation is performed internally to each layer (feature), decreasing its dimension.

A simple function (such as maximum or mean) is applied to all values in a window.

A volume of values is produced such that:

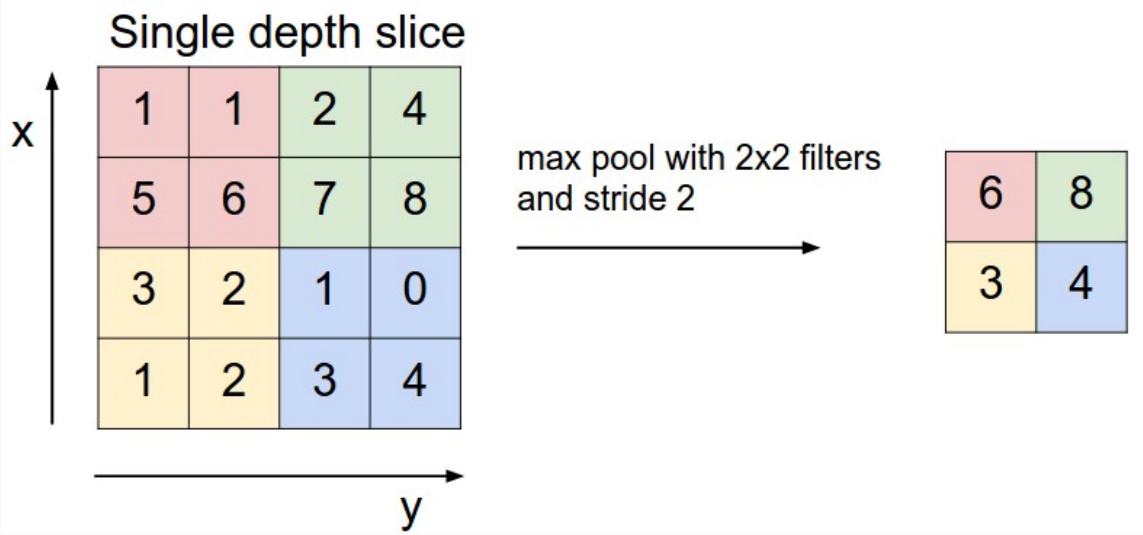
- the number of layers (depth) is not modified
- width and height are equal to the ones of the input volume, divided by the stride.

Pooling Layer

- Accepts a volume of size $w_1 \times h_1 \times d_1$
- Requires two hyperparameters: spatial extent f and stride s
- Produces a volume of size $w_2 \times h_2 \times d_2$ where:
- $w_2 = (w_1 - f)/s + 1$
- $h_2 = (h_1 - f)/s + 1$
- $d_2 = d_1$
- Introduces zero parameters since it computes a fixed function of the input (mean, maximum)
- Usually, no padding

Only two schemes commonly found in practice: pooling layer with $f = 3, s = 2$, and more commonly $f = 2, s = 2$. Pooling sizes with larger receptive fields are too destructive.

Pooling layer



Fully-connected layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks.
The only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical.
It is possible to convert between FC and CONV layers.

RELU layer

This is just a way of representing the application of a non-linear activation function, such as *RELU*, to the results of the dot products performed at layers of different types.

Layer Patterns

Most common form of a ConvNet architecture

Stack of a few CONV-RELU layers, followed by POOL layers. Pattern repeated until the image has been merged spatially to a small size. At some point, transition to fully-connected layers. The last fully-connected layer holds the output, such as the class scores.

`INPUT -> [[CONV -> RELU]*N -> POOL?]*M -> [FC -> RELU]*K -> FC`

But more complex scheme exist (Google Inception, Microsoft ResNets).

In practice

One should rarely ever have to train a ConvNet from scratch or design one from scratch. Better looking at whatever architecture currently works best in a general case (ImageNet), download a pretrained model and finetune it on our data.

Case studies

Some architectures in the field of Convolutional Networks that have a name. The most common are:

- **LeNet**. One of the first successful applications of Convolutional Networks (1998)
- **AlexNet**. (2012) Very similar architecture to LeNet, but deeper, bigger, and featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).
- **ZF Net**. (2013) an improvement on AlexNet obtained by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller.
- **GoogLeNet**. (2014) introduced an *Inception Module* that dramatically reduced the number of parameters in the network (4M, compared to AlexNet with 60M).
- **VGGNet**. (2014) contains 16 CONV/FC layers; only performs 3x3 convolutions and 2x2 pooling from the beginning to the end.
- **ResNet**. (2015) very common choice for using ConvNets in practice.

Recursive neural networks

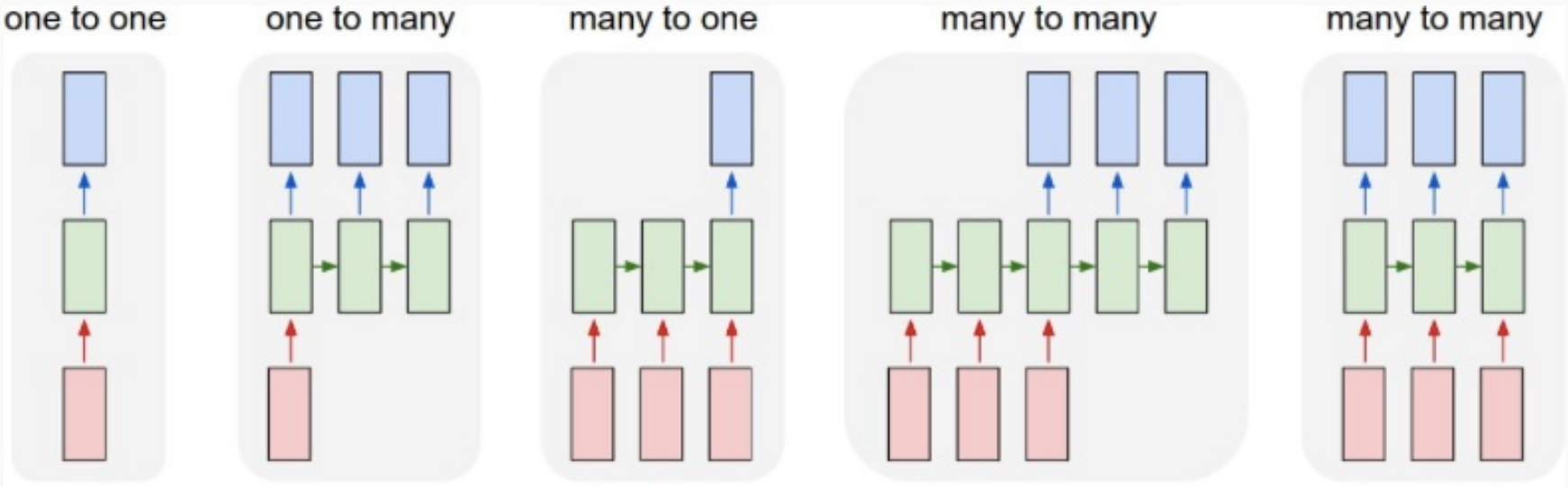
Many real-word problems require processing a timed (or in general indexed) sequence or signal

- *Sequence classification*: sentiment analysis, DNA sequence classification, action recognition
- *Sequence synthesis*: text, music
- *Sequence-to-sequence translation*: speech recognition, text translation, PoS tagging

Modeling sequential data

- Sample data sequences from a distribution $p(x_1, \ldots, x_T)$
- Generate data sequences to describe an image $p(y_1, \ldots, y_T | I)$
- Activity recognition from a sequence $p(y | x_1, \ldots, x_T)$
- Speech recognition; Object tracking $p(y_1, \ldots, y_T | x_1, \ldots, x_T)$
- Generate sentences to describe a video; language translation $p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T)$

Recurrent neural networks

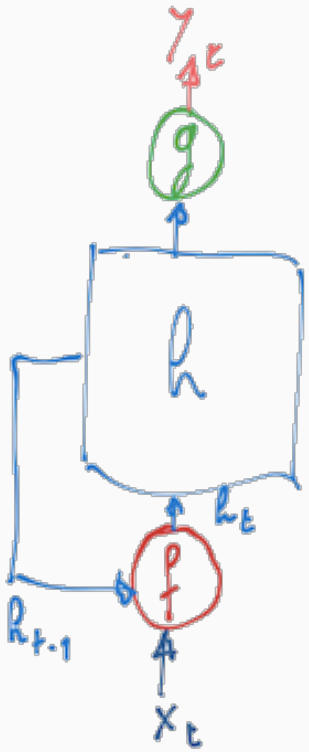


Recurrent neural networks

A RNN, differently from MLP and CNN, is applied to sequences of input vectors.

At each new input vector, an output is returned which is a function of such vector and of all past ones.

This is obtained by using the current output as (part of the) input for the next step.



Recurrent neural networks

Given a sequence x and an initial state h_0, the model iteratively computes the sequence of recurrent states

$$h_t = f(x_t, h_{t-1}) = \phi(w_1 x_t + w_2 h_{t-1} + b) \qquad t = 1, \dots, T(x)$$

w_1, w_2 weight matrices, learned by backpropagation

ϕ is usually a logistic function

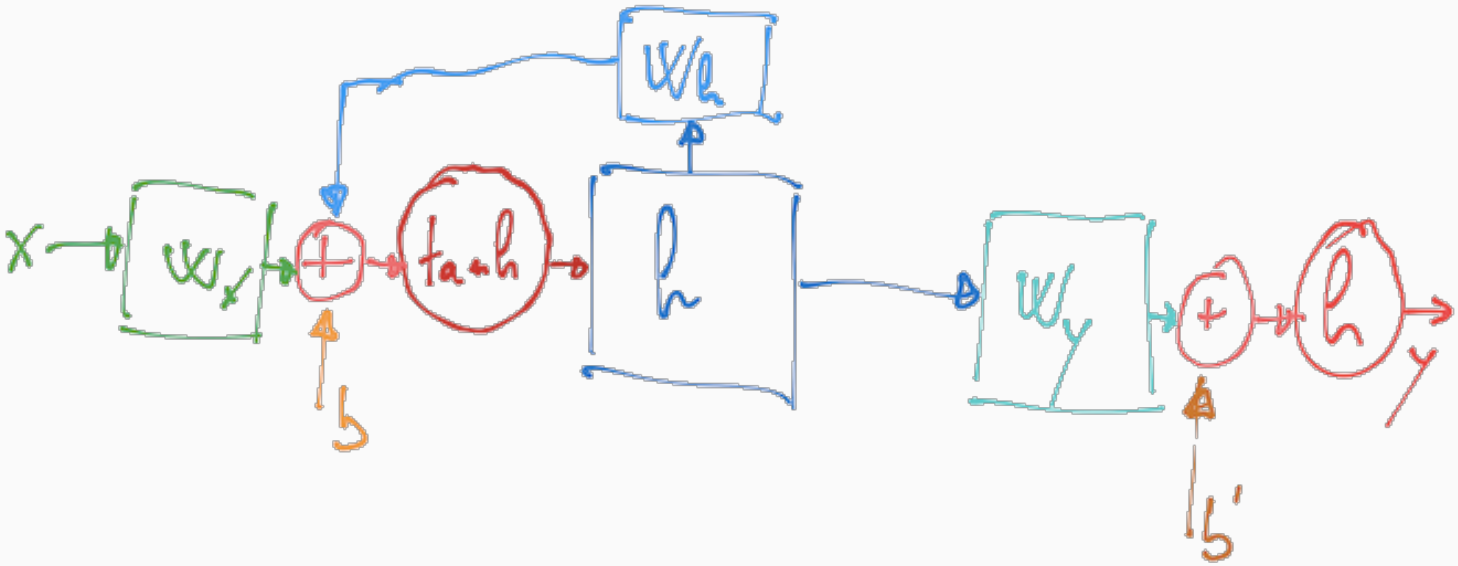
Recurrent neural networks

A prediction can be computed at any time step from the recurrent state

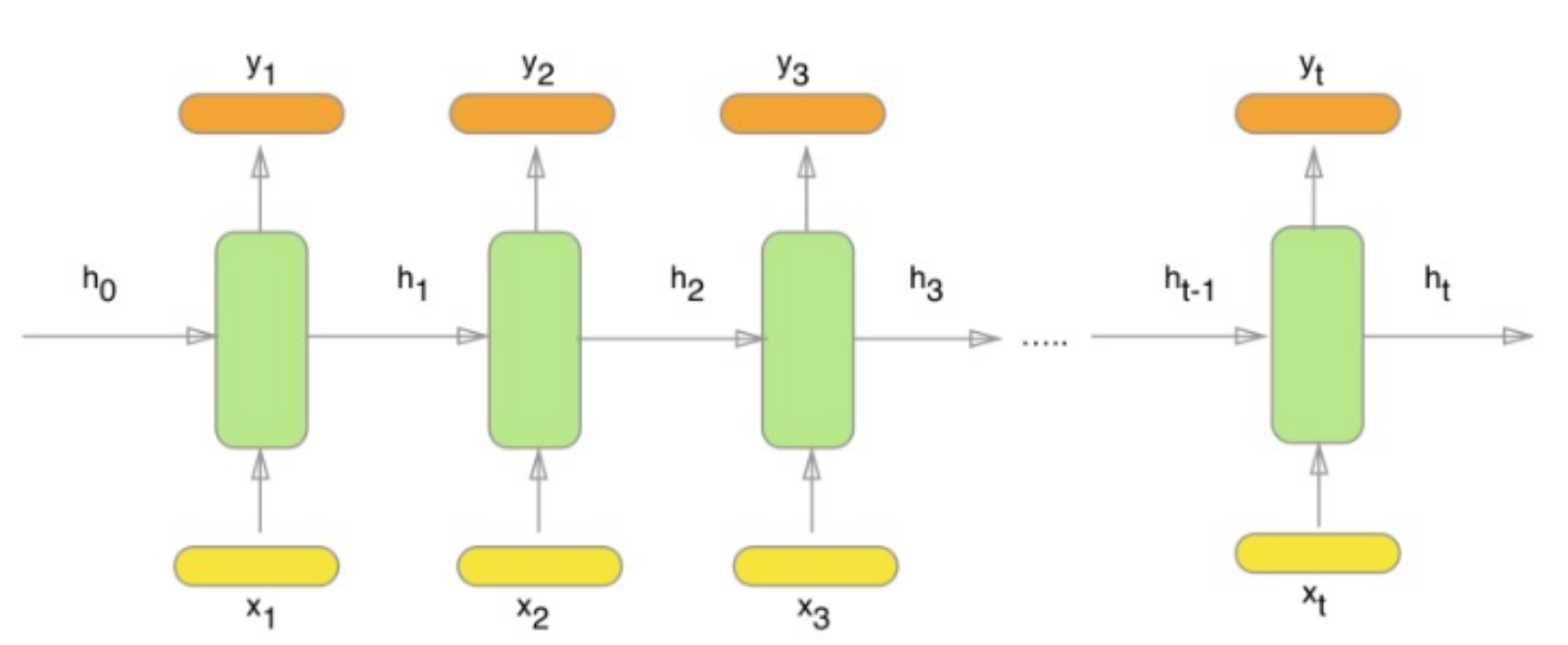
$$y_t = g(h_t; w) = \psi(w_3 h_t + b')$$

ψ is task-dependent. Usually, a softmax

Recurrent neural networks



Recurrent neural networks



Network can be seen as a sequence of modules sharing the same coefficients.
Backpropagation through time (*BPTT*)

LSTM networks

Long Short Term Memory networks – usually just called *LSTM* – are a special kind of RNN, capable of learning long-term dependencies, that is, situations when y_t is strongly dependent from $x_{t'}$ with $t' << t$

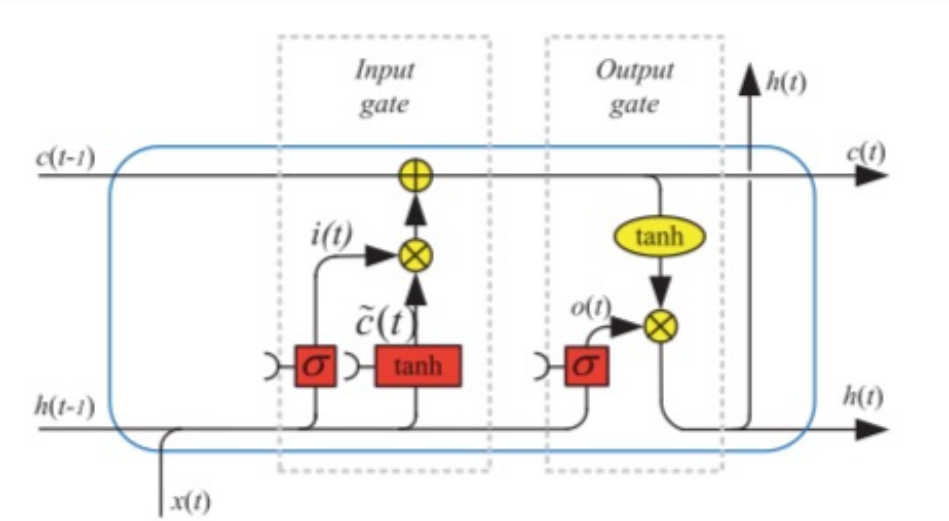
RNN, even if capable in principle, are not able in practice to deal with this problem (vanishing gradient). LSTM specialize the module structure by introducing a set of specialized interacting layers

Cell state: stores long term information; its value can be modified by suitable operations

- a product (with a vector a values in $(0, 1)$) decreasing the weights of components
- a sum (with a vector a values in $(0, 1)$) increasing the weights of components

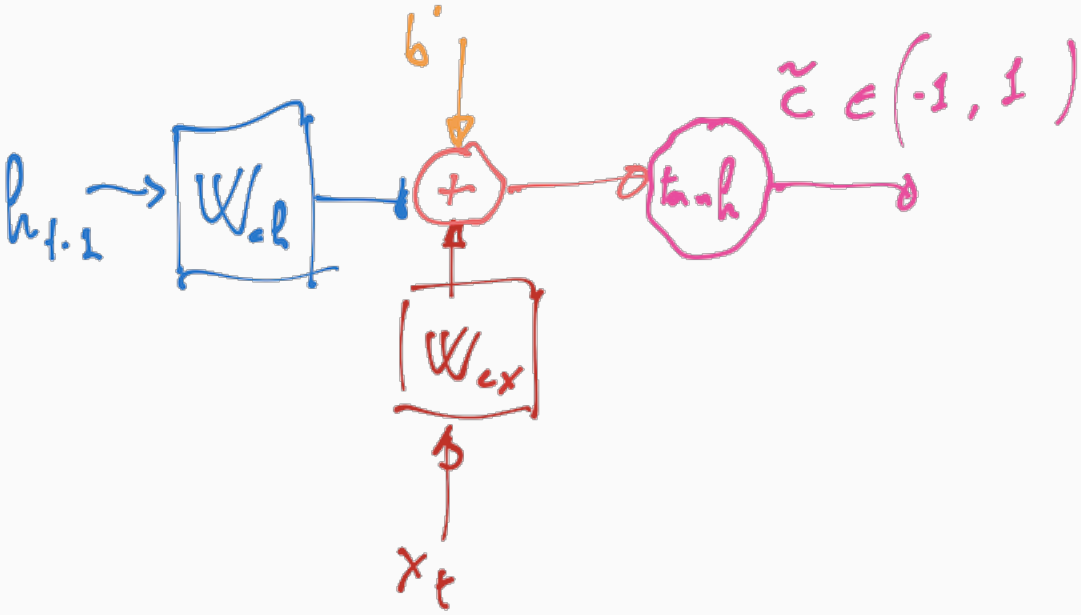
LSTM networks: first version

No product, only sum (no forget)



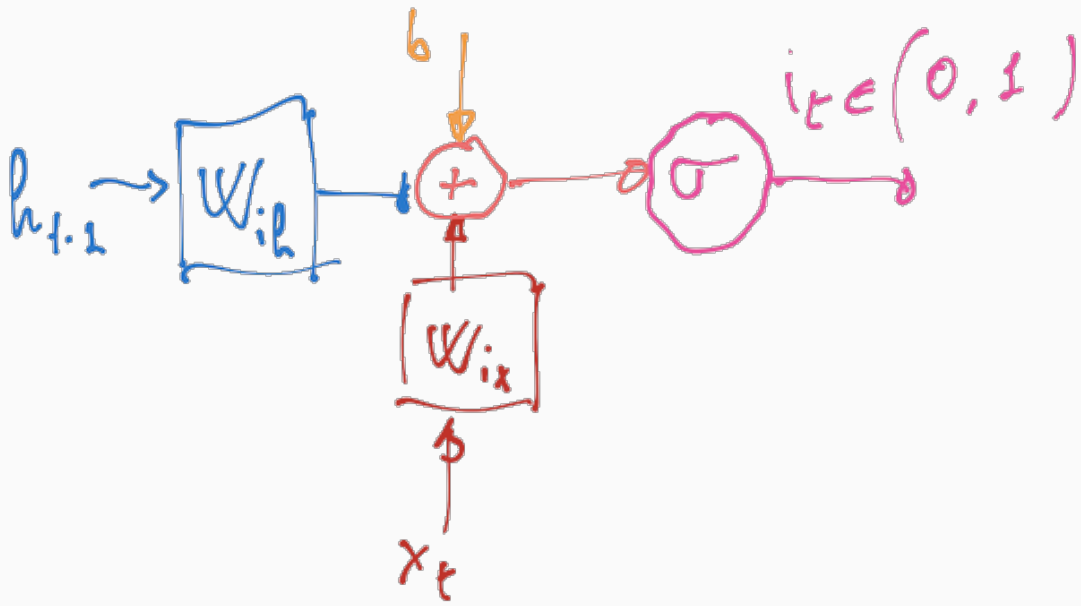
$$i_t = \sigma(w_{ih} \cdot h_{t-1} + w_{ix} \cdot x_t + b_i)$$
$$\tilde{C}_t = \tanh(w_{ch} \cdot h_{t-1} + w_{cx} \cdot x_t + b_c)$$
$$C_t = C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma(w_{oh} \cdot h_{t-1} + w_{ox} \cdot x_t + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

LSTM networks: input layer



Determines increment or decrement of long term memory cell components

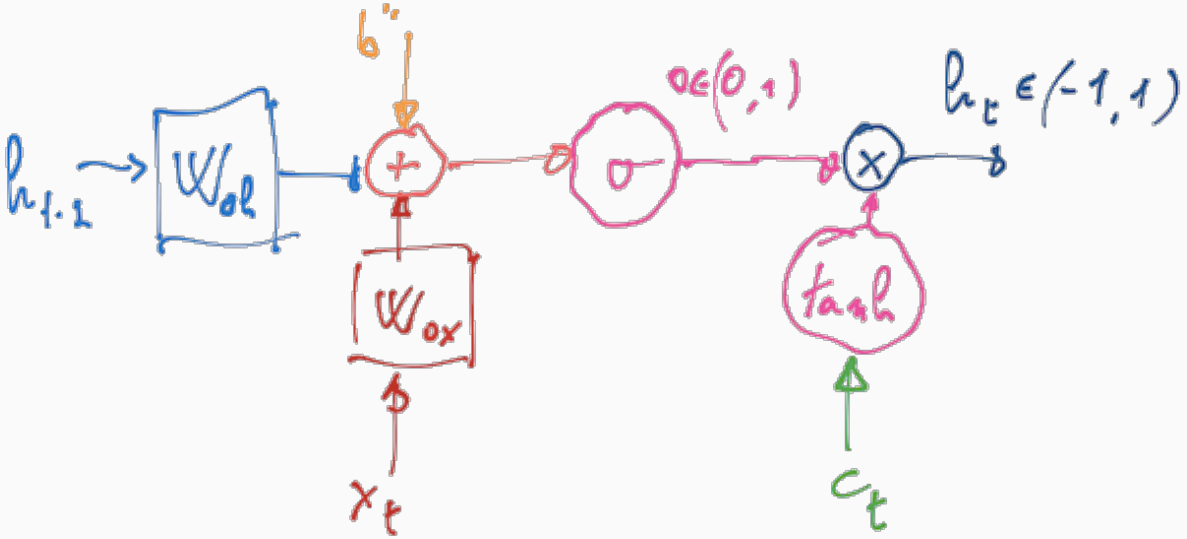
LSTM networks: input layer



Additionally determines the increment or decrement of long term memory cell components, by assign a weight to each component update

New values of long term memory components are derived by increasing them by the values in $\tilde{c}$, weighted by the values in i

LSTM networks: output layer

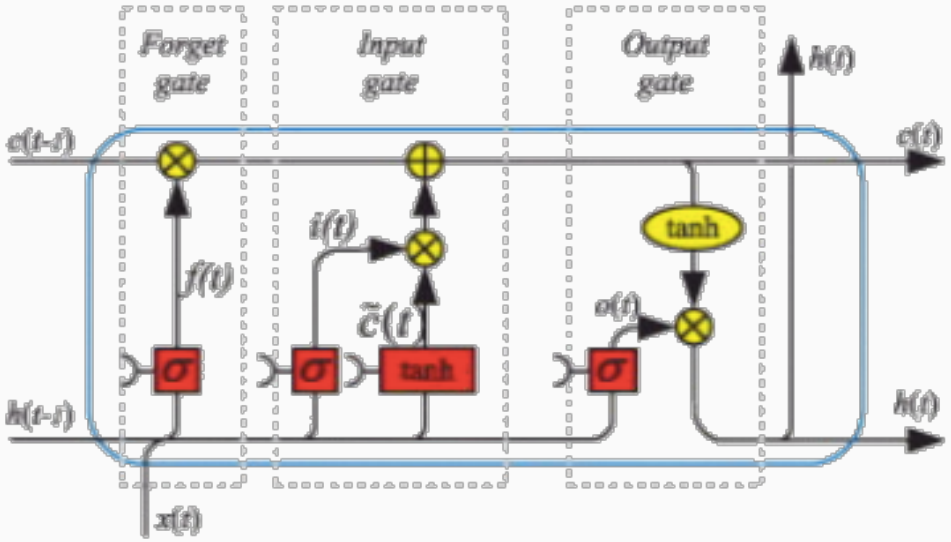


The values of long term memory components are squashed to be in $(-1, 1)$

Input and current state determine how much each component is maintained in the new state (and returned as output)

LSTM networks with forget layer

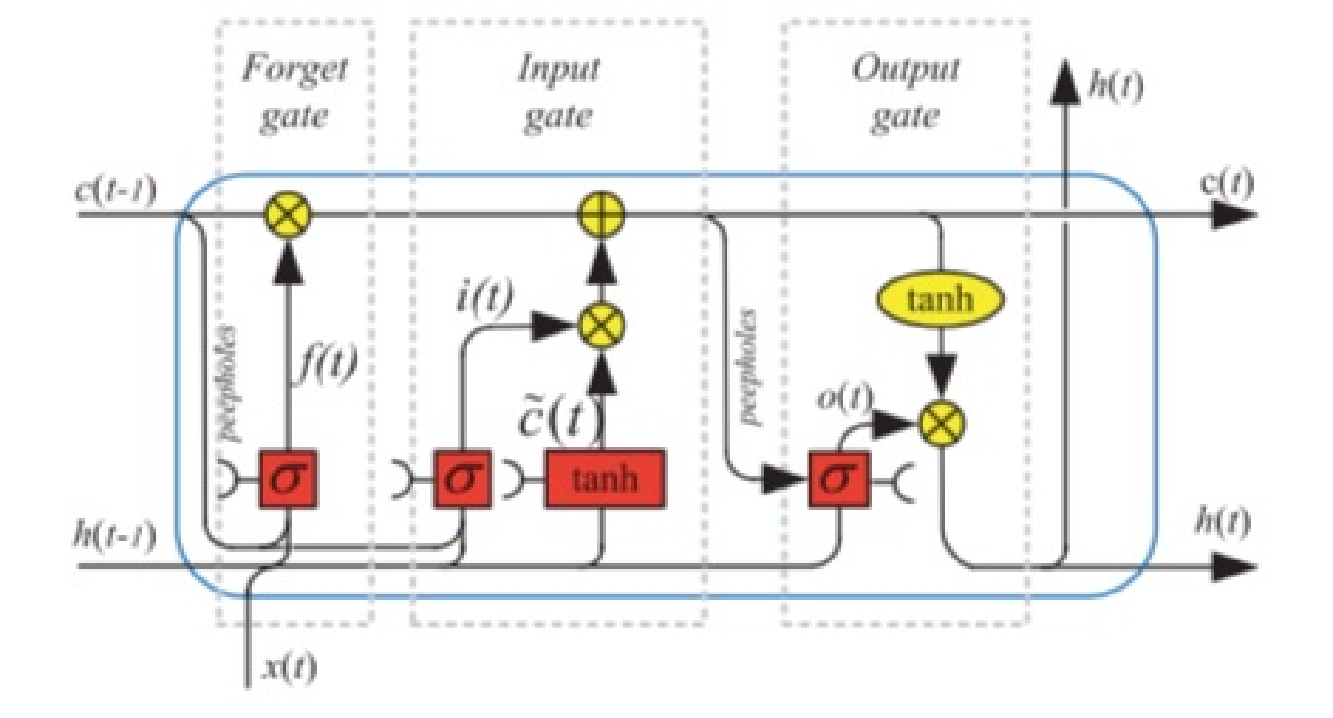
Forget gate layer: determines the decrease of relevance of cell state components as a function of input and history



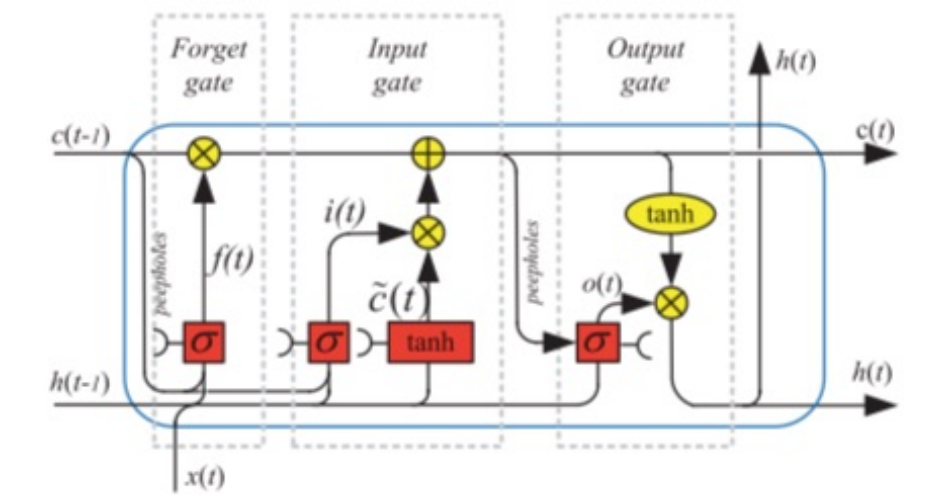
$$f_t = \sigma(w_{fh} \cdot h_{t-1} + w_{fx} \cdot x_t + b_f)$$
$$i_t = \sigma(w_{ih} \cdot h_{t-1} + w_{ix} \cdot x_t + b_i)$$
$$\tilde{C}_t = \tanh(w_{ch} \cdot h_{t-1} + w_{cx} \cdot x_t + b_c)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma(w_{oh} \cdot h_{t-1} + w_{ox} \cdot x_t + b_o)$$
$$h_t = o_t * \tanh(C_t)$$

LSTM networks variants

In LSTM above there is no direct connection between cell state and gates: this lack of information at gates may spoil network performance. To handle this issue, additional connections (peephole) are defined.



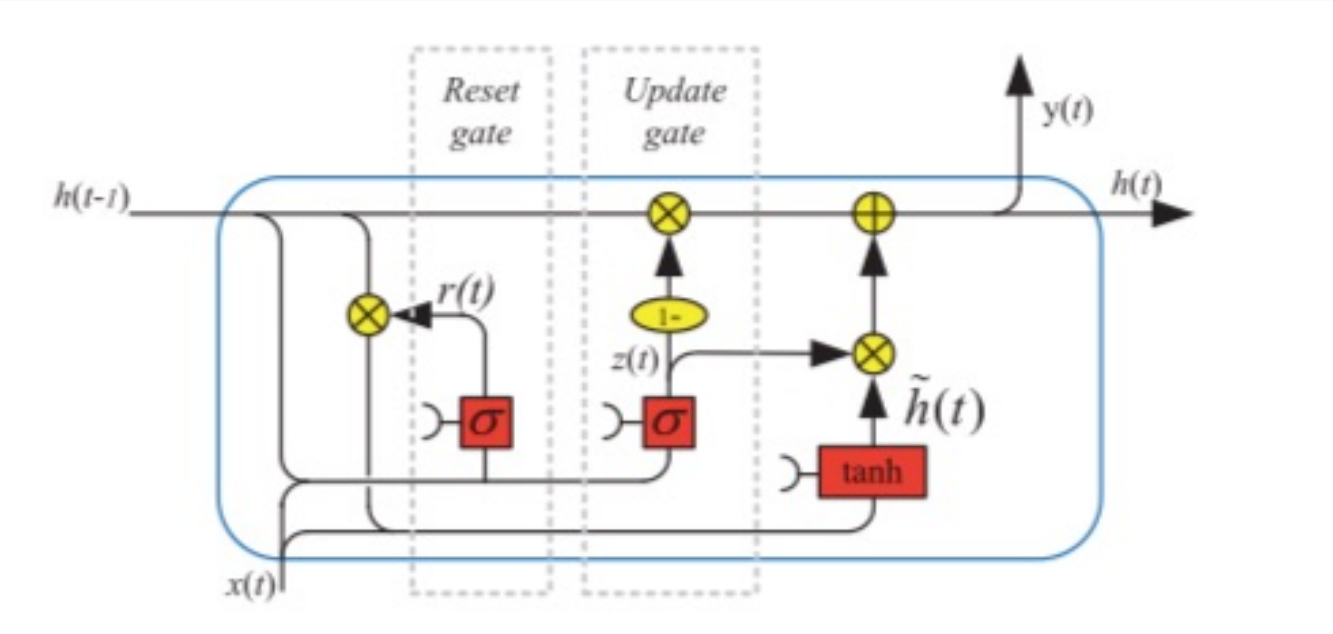
LSTM networks variants



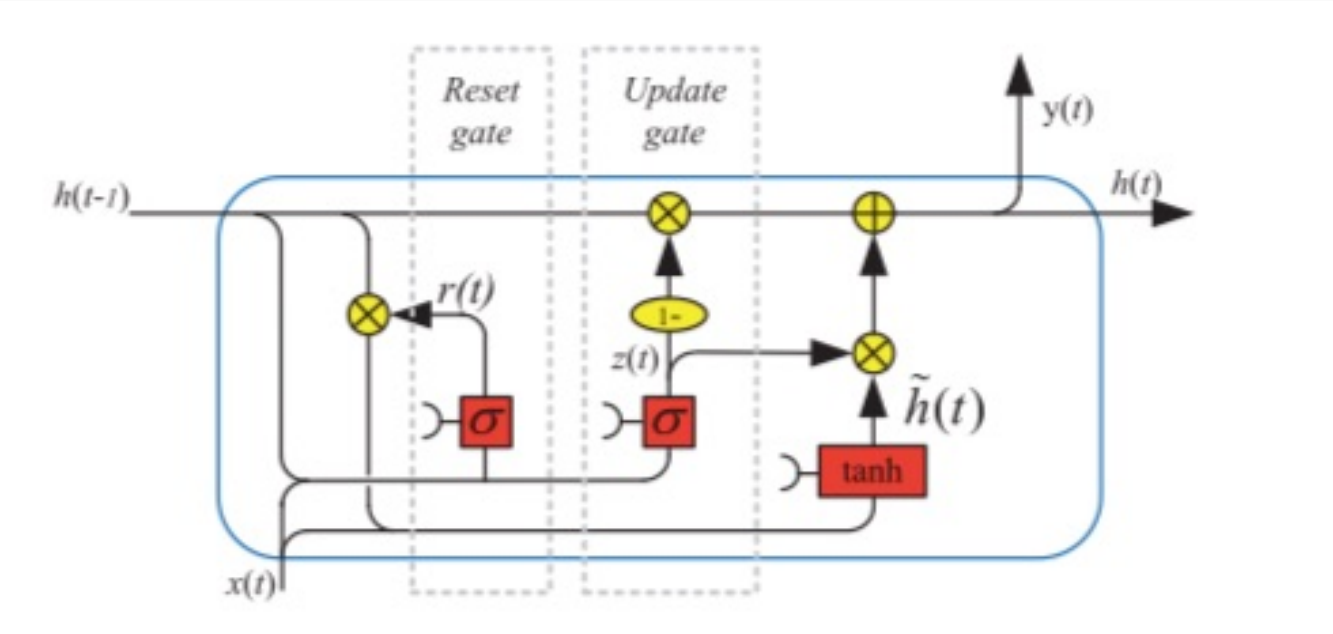
- $f_t = \sigma(w_f^{(1)^T} h_{t-1} + w_f^{(2)^T} x_t + P_f^T c_{t-1} + b_f)$
- $i_t = \sigma(w_i^{(1)^T} h_{t-1} + w_i^{(2)^T} x_t + P_i^T c_{t-1} + b_i)$
- $\tilde{C}_t = \mathsf{tanh}(w_c^{(1)^T} h_{t-1} + w_c^{(2)^T} x_t + b_c)$
- $C_t = C_{t-1} + i_t * \tilde{C}_t$
- $o_t = \sigma(w_o^{(1)^T} h_{t-1} + w_o^{(2)^T} x_t + b_o)$
- $h_t = o_t * \mathsf{tanh}(C_t)$

LSTM networks variants

The Gated Recurrent Unit was introduced to limit the number of networks coefficients to be learned. The input and forget gates are integrated in a unique update gate. The networks now has two gates, total: an update gate and a reset gate.



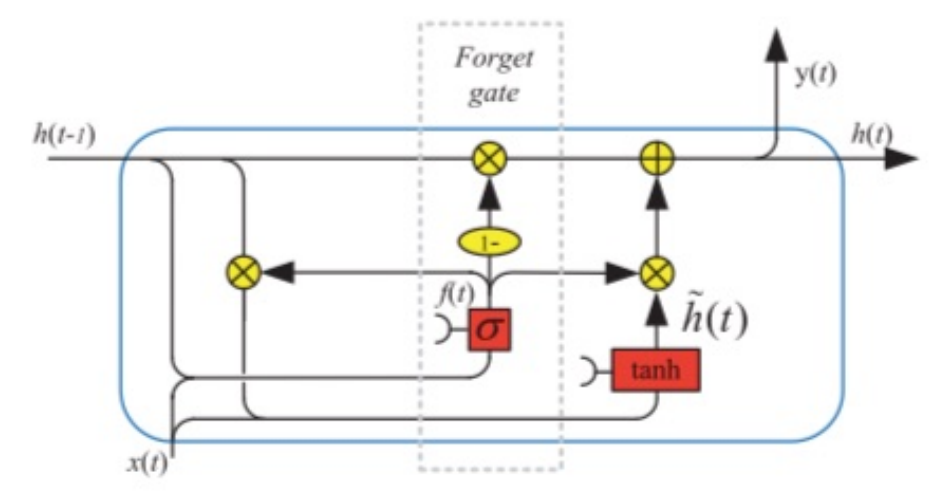
LSTM networks variants



- $r_t = \sigma(w_r^{(1)^T} h_{t-1} + w_r^{(2)^T} x_t + b_r)$
- $z_t = \sigma(w_z^{(1)^T} h_{t-1} + w_z^{(2)^T} x_t + b_z)$
- $\tilde{h}_t = \tanh(w_h^{(1)^T} (r_t * h_{t-1}) + w_h^{(2)^T} x_t + b_z)$
- $h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$

LSTM networks variants

A Minimal Gated Unit further limits the number of coefficients, by using a single gate



- $f_t = \sigma(w_f^{(1)^T} h_{t-1} + w_f^{(2)^T} x_t + b_f)$
- $\tilde{h}_t = \tanh(w_h^{(1)^T} (f_t * h_{t-1}) + w_h^{(2)^T} x_t + b_h)$
- $h_t = (1 - f_t) * h_{t-1} + f_t * \tilde{h}_t$

Topologies

Recurring cells (in particolar LSTM) can be connected according to many different topologies.

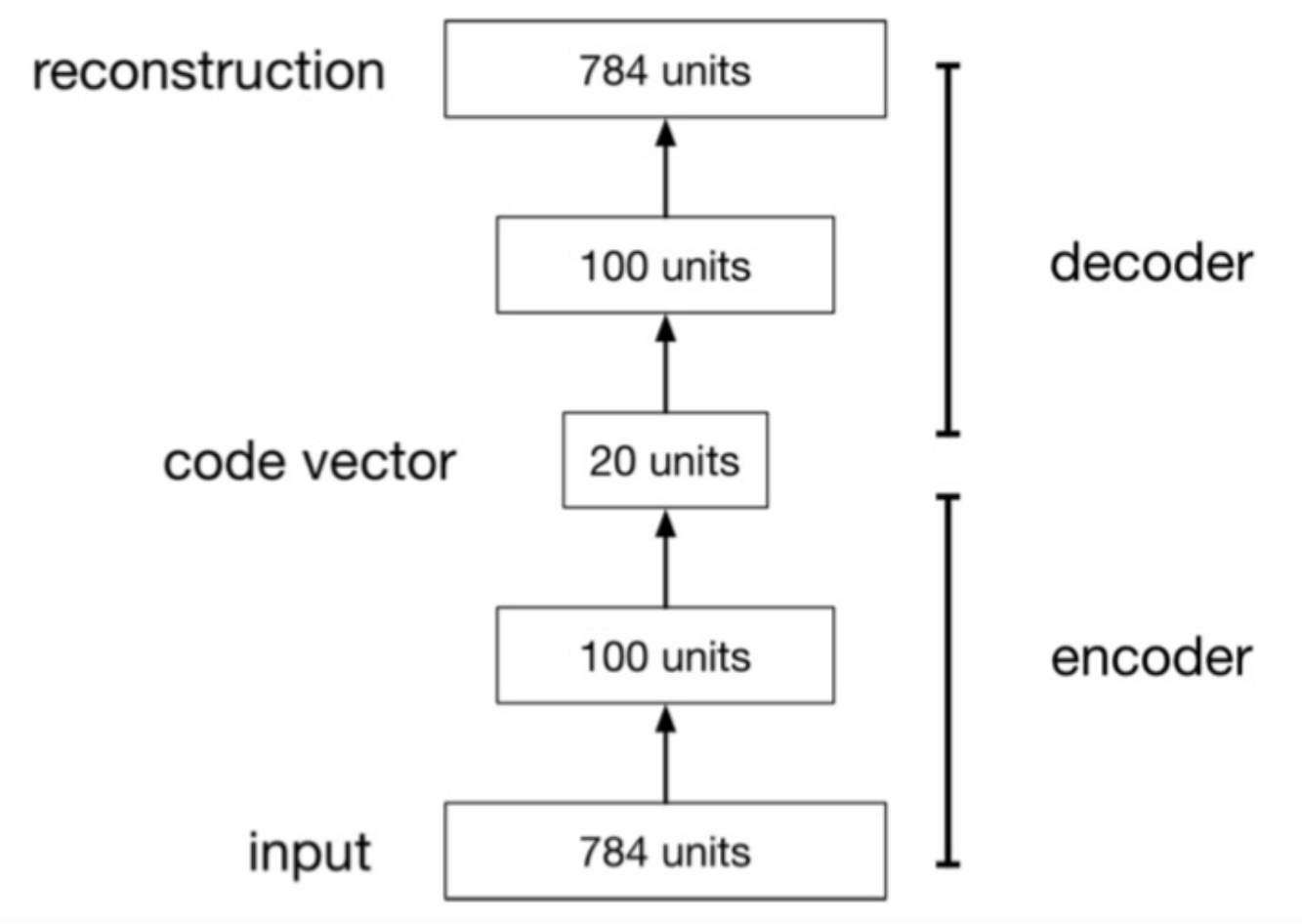
- Stacked networks
- Bidirectional networks
- Multidimensional networks
- Graph networks

Moreover, recurring structures (layer) can be integrated within more complex networks.

Autoencoders

- An autoencoder is a feed-forward neural net whose job it is to take an input x and predict x.
- To make this non-trivial, we need to add a bottleneck layer whose dimension is much smaller than the input.

Autoencoders

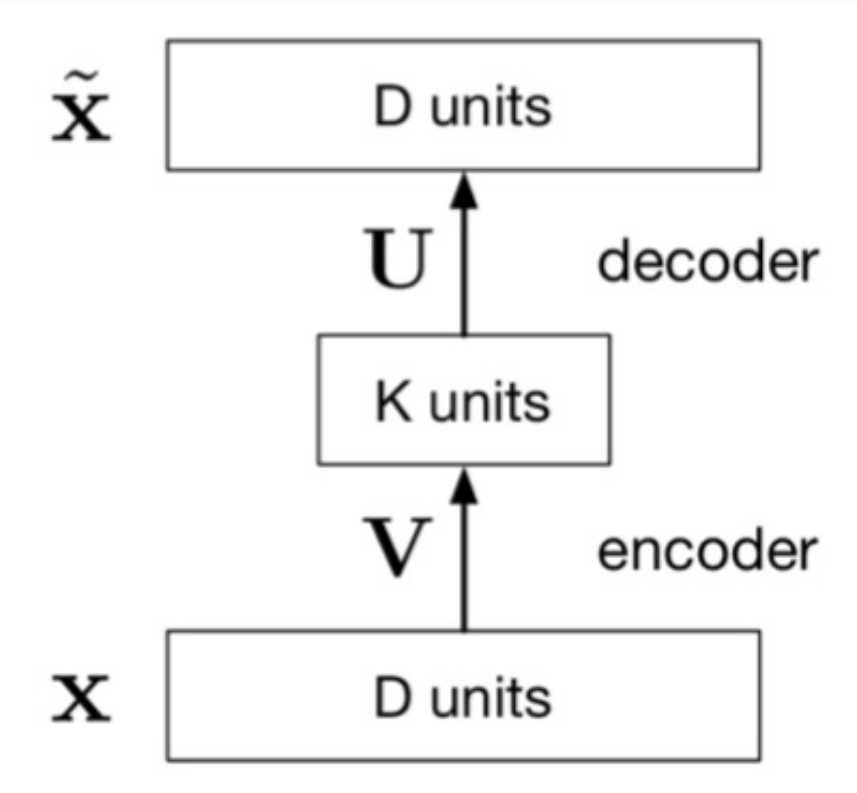


Why autoencoders?

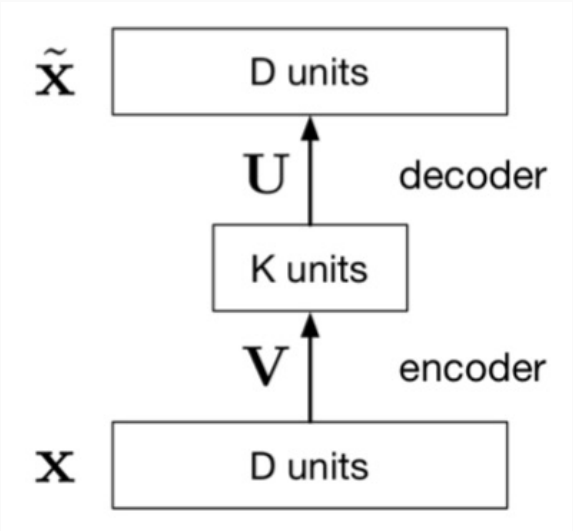
- Map high-dimensional data to two dimensions for visualization
- Compression (i.e. reducing the file size)
- Learn abstract features in an unsupervised way so you can apply them to a supervised task

PCA

- The simplest kind of autoencoder has one hidden layer, linear activations, and squared error loss $L(x_1, x_2) = || x_1 - x_2 ||^2$



PCA



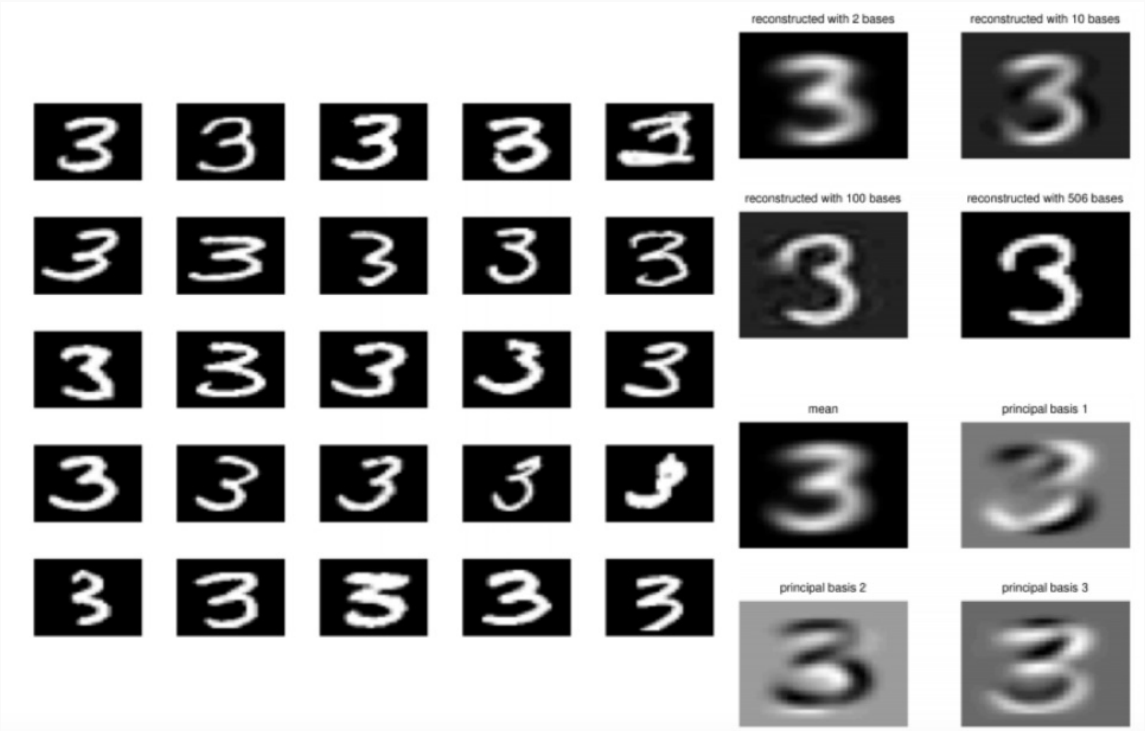
- This network computes $\tilde{x} = UVx$, a linear function. If $K \geq D$, U and V can be chosen such that $UV = I$ is the identity. This is not very interesting
- If $K < D$
  - V maps x from a D dimensional space to a K dimensional space, defined by the column space of U. So it is doing dimensionality reduction
  - U maps Vx from a K dimensional space to the D dimensional space

PCA

- What is the best possible mapping to choose?
- It is well known that the best mapping corresponds to the column of U being the eigenvectors of the scatter matrix $S = \frac{1}{n}\sum_{i=1}^{n}(x_i - m)(x_i - m)^T$ corresponding to the K largest eigenvalues

PCA for faces

PCA for digits

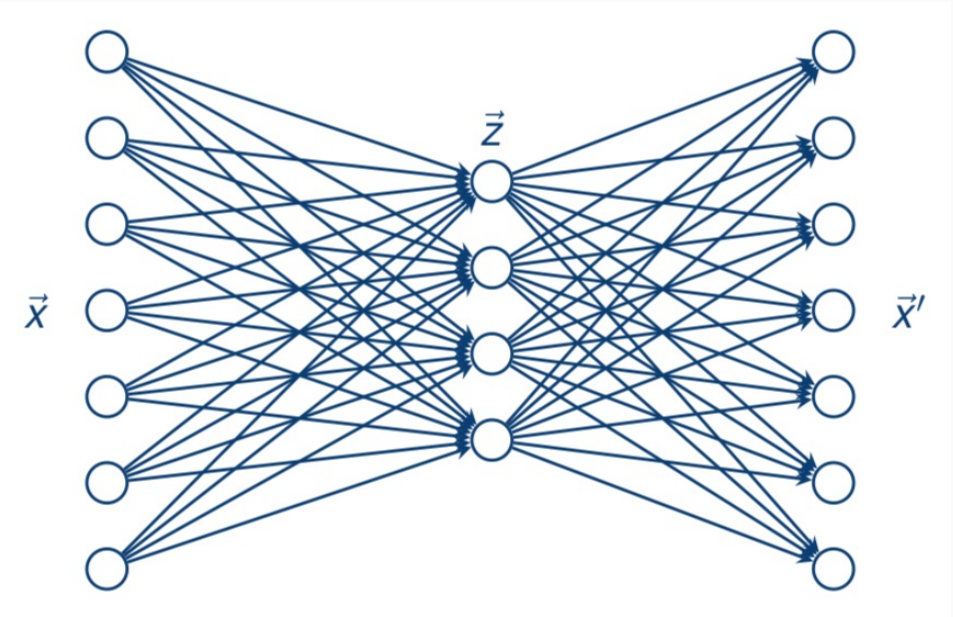


Extension to nonlinear mappings

- PCA relies on linear mapping: projection is done into a linear subspace
- lower information loss if mapping on nonlinear manifold is allowed
- neural networks can be applied

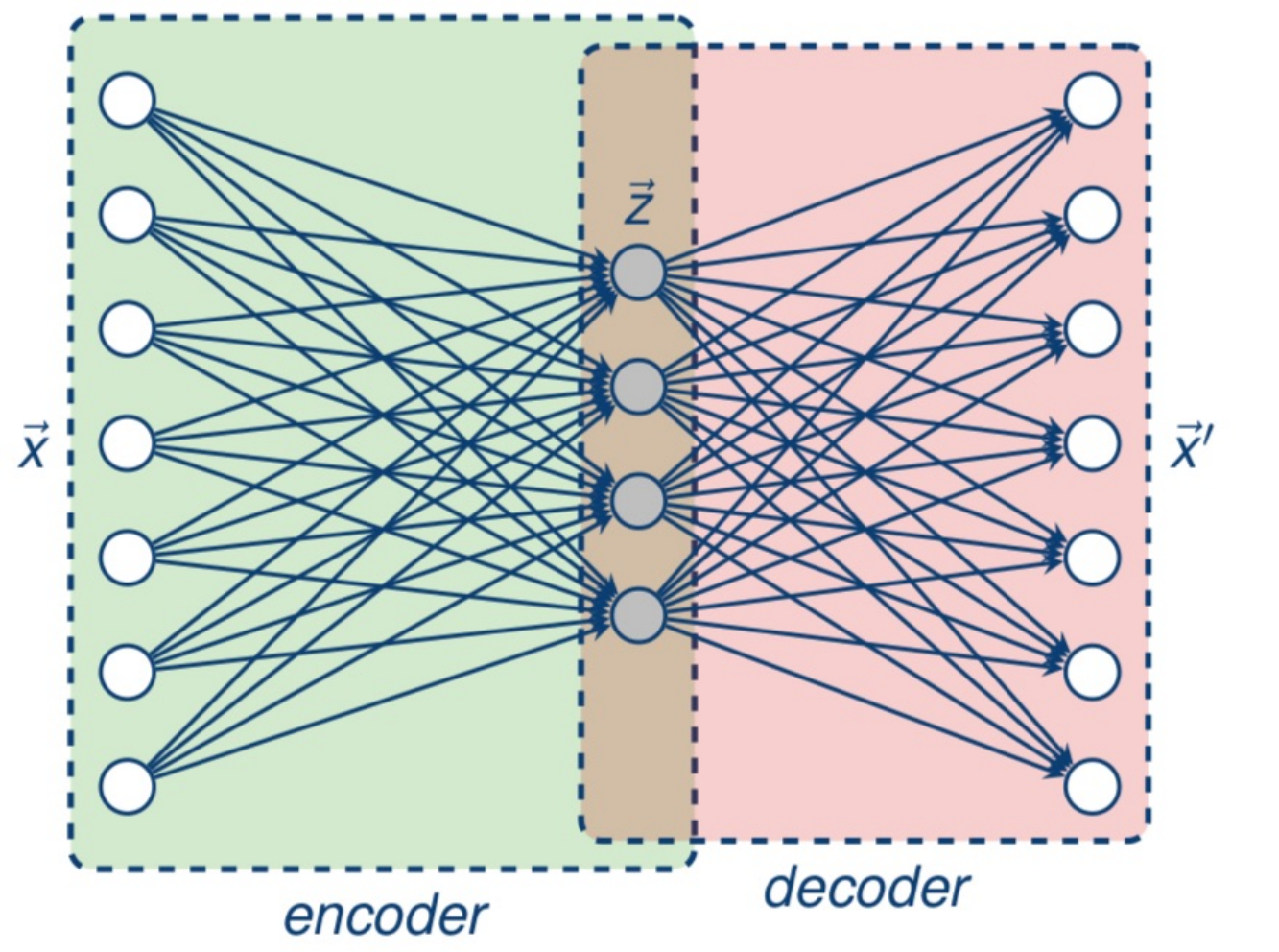
Alternative perspective

- It should be simple to relate the operations of PCA to those of a two-layer fully-connected neural network without activations
- This would allow us to work in exactly the same scenario, but derive mappings using backpropagation



$$z = Vx$$
$$x = Uz$$

Alternative perspective



So what?

- Thus far, our (linear) autoencoder is only capable of the same kind of expressivity as PCA. Indeed, it is simply a rephrasing of PCA
- More so, it was doing so inefficiently compared to PCA. It applied backpropagation to approximately derive the optimum solution, which is easily computable directly
- However, enabling training by backpropagation means that we can now introduce depth and nonlinearity into the model
- This should allow us to capture complex non linear manifolds more accurately

Autoencoders

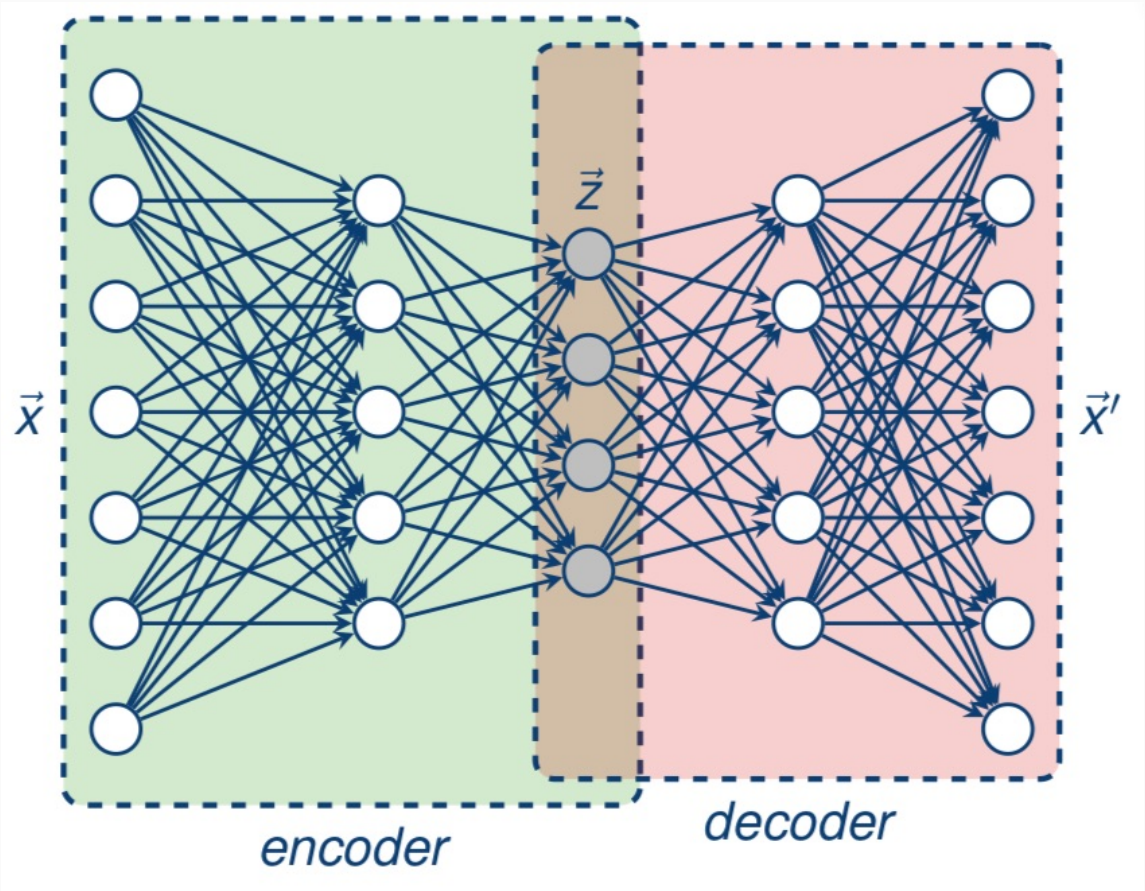
- Encoder: $z = \sigma(Vx)$
- Decoder: $x' = g(Uz) = g(U\sigma(Vx))$
- g could be the identity if x is defined on real values or σ if they are binary $\{0, 1\}$
- Loss functions vary accordingly from squared loss $|| x - x' ||^2$ to cross entropy $-(x \log x' + (1 - x) \log(1 - x'))$

Deep autoencoders

- We introduce additional non linear layers of r (ReLU) activations

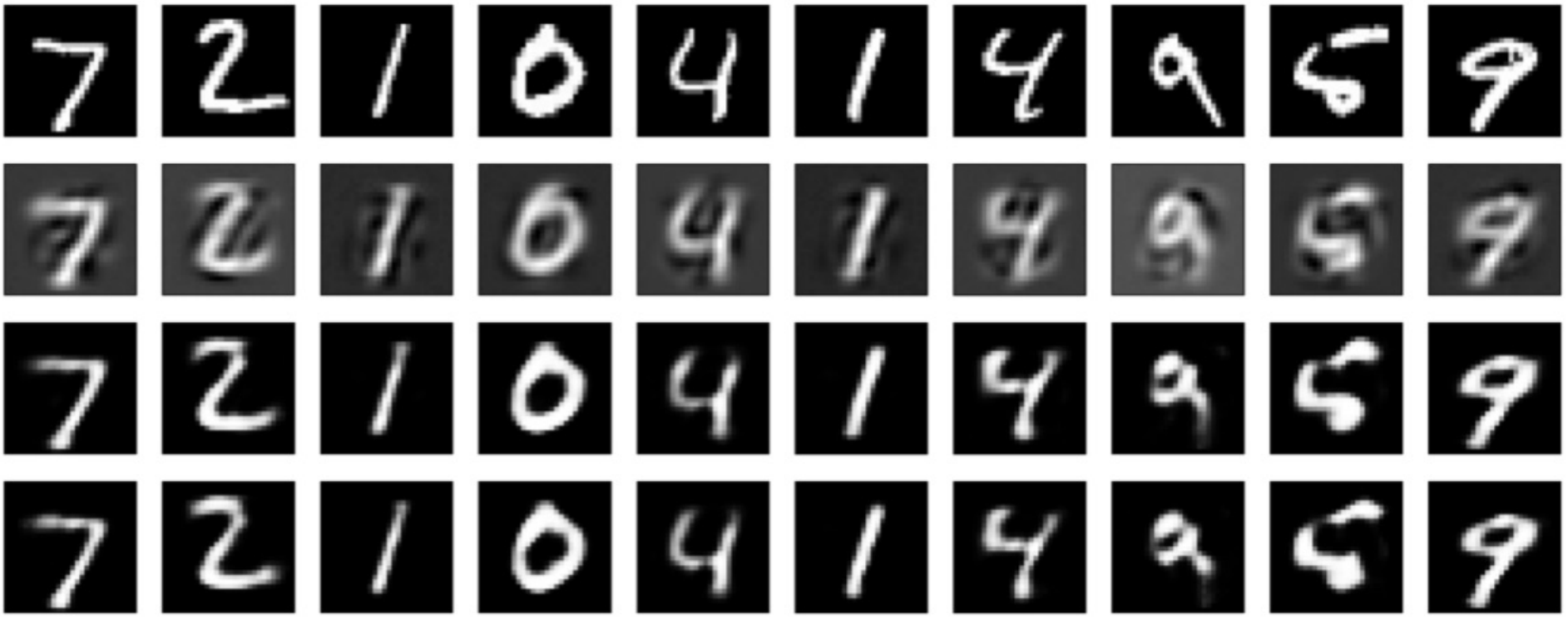
$$z = r(W_2 r(W_1 x))$$
$$x = g(W_4 r(W_3 z))$$

Deep autoencoders



Autoencoders on MNIST images

Line by line: original data, reconstruction with 30d PCA, reconstruction with 30d nonlinear autoencoder, reconstruction with 30d deep autoencoder



Extensions

- The autoencoder may be extended in several ways, such as
  - Sparse autoencoders
  - Denoising autoencoders
  - Variational autoencoders

Sparse autoencoder

A sparse autoencoder is one that is regularized to not only minimize loss, but to also incorporate sparse features.

If $z = h(x)$ and $x' = g(z)$, then the sparse encoder has the following loss:

$$L(x, x') + \lambda \sum_i \mid z_i \mid$$

where z_i is the i-th element of z.

This is intuitive, as we know L1-regularization introduces sparsity.

Denoising autoencoder

- An autoencoder that is robust to noise.
- Assume an additional noise ε, so that $\tilde{x} = x + \varepsilon$.
- Then, the loss function of the autoencoder could be defined as

$$L(x, g(h(\tilde{x})))$$

and it would learn to denoise $\tilde{x}$ to reproduce x.

Variational autoencoder

- A VAE, instead of learning $h()$ and $g()$, learns distributions of the features given the input and of the input given the activations, i.e., probabilistic versions of $h()$ and $g()$.
- That is, a VAE will learn:
  - $p(z|x)$: the distribution of features given the input
  - $p(x|z)$: the distribution of input given the features
- That is, a VAE learns the distribution of observed variables x conditioned on latent variables z and vice versa

Why learn distributions?

- Often, data is noisy, and a model of them, in the form distribution of a probability distribution is more useful for a given application.
- Making inference may result easier if the relationship between x and z is complex and non linear
- The VAE is a generative model. Given the observed data distribution $p(x)$, it is possible to:
  - sample $\hat{x}$ from $p(x)$
  - sample $\hat{z}$ from $p(z|\hat{x})$
  - sample x' from $p(x|\hat{z})$
- This enables the generation of data that has similar statistics wrt the input
- In practice, a function $g(z)$ is often used

Generative models

Use of latent variables models (representations) to generate data which "looks like" the one provided in datasets

Fake face/text/music generation

- VAE: (approximate) modeling of input distribution
- GAN (Generative Adversarial Network): generator network + discriminator network
  - generator trained to provide fake data which is accepted by discriminator
  - discriminator trained to effectively discriminate real data (from dataset) from fakes