Machine learning

Deep learning

Corso di Laurea Magistrale in Informatica

Università di Roma Tor Vergata

Prof. Giorgio Gambosi

a.a. 2023-2024

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 perceptrons
- **CNN** Convolutional Neural Network
- **AE** Auto Encoders
- **RNN** Recurrent Neural Networks
- **LSTM** Long-Short Term Memory Networks
- **Tranformers** Attention-based Neural Networks
- **Graph Neural Networks** Graph topology-based Neural Networks
- **GAN** Generative Adversarial Networks
- ...

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

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 or the hyperbolic tangent as activation function in a neural network hidden units is a critical issue since derivative values

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

may tend to monotonically decrease, in module, during backpropagation, as k decreases.

This is the problem of vanishing gradient.

Common solution: applying different non linear functions, such as **RELU** (Rectified Linear Unit)

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

Exploding gradient

The opposite problem of gradient unbounded increase may also occur, making the network unstable.

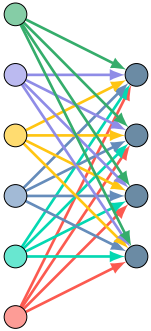
Usually managed through clipping or regularization

Convolutional Neural Networks

Very similar to ordinary Neural Networks

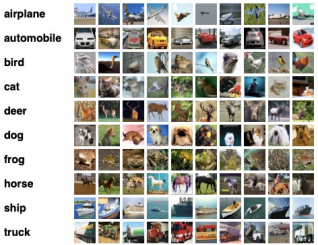
- made up of neurons that have learnable weights and biases
- each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity
- the whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the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

MLP RECAP



- Multilayer Perceptrons 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
- The last fully-connected layer is called the *output layer* and in classification settings it represents the class scores.

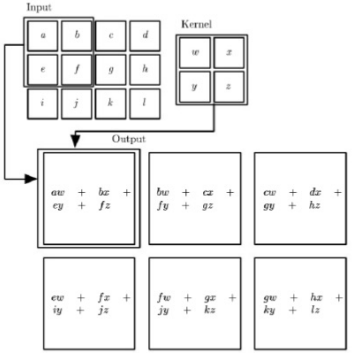
Neural networks issue with images

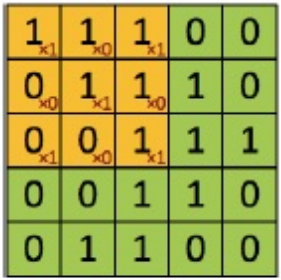


- In the CIFAR-10 dataset, images have size 32x32x3 (32 wide, 32 high, 3 color channels): this results in vectors of 32*32*3 = 3072 values
- a single fully-connected neuron in a first hidden layer of a MLP would be associated to 32*32*3+1 = 3073 weights
- an image of size 200x200x3 would result into a first layer where each neuron is associated 120.001 weights.
- clearly, this full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.
- some type of approach to reduce the overall number of coefficients must be applied

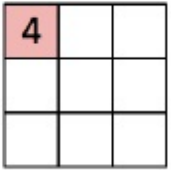
Convolutional Neural Networks

Usually applied to images. Fundamental idea: a kernel (coefficient matrix) “moves” on the output of the previous layer, at each step covering a restricted region and performing a linear combination (convolution) to produce a value. The set of output values corresponds to the set of regions considered





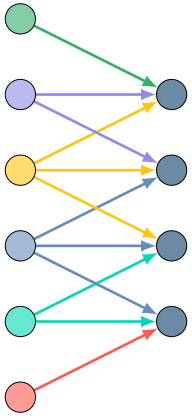
Image



Convolved Feature

Local connections

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



Convolutional network structure

- Units in a layer of a convolutional network are usually arranged in 3 dimensions: width, height, depth.
- For example, an input image in CIFAR-10 is a 3d volume of values (inputs, for the next layer), with dimensions 32x32x3 (width, height, depth respectively).
- Each layer gets a 3d volume of inputs from the preceding one and produces a different 3d volume for the following one:
  - width and height depend from width and height of the previous layer and on the type of layer
  - depth is a project choice

Layers used to build convolutional network

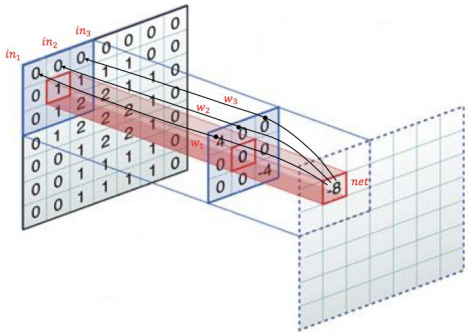
Three main types of layers:

- **Convolutional Layer**: applies a convolution operation to the previous layer, by applying a kernel (matrix of coefficients to be learned) to each region of a set of overlapping ones
- **Pooling Layer**: implements a fixed (no coefficients to be learned) downsampling function to reduce the volume size: a set of regions of the input volume is defined and a single value is computed from each of them
- **RELU Layer**: applies a RELU activation function (no coefficients to be learned) to all input values
- **Fully-Connected Layer**: an MLP-style layer

In summary

- A convolutional network architecture architecture is a list of layers that transform input volumes into output volumes
- 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 hyper-parameters

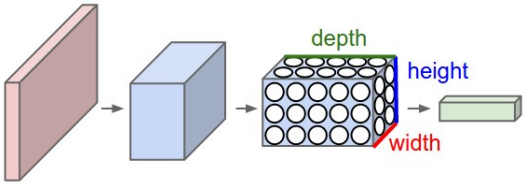
Convolutional Layer



- A convolutional layer of dimensions w, h, d can be seen as applying a set of d different filters on the input volume, each composed of $w \times h$, returning a $w \times h$ matrix of values
- A unique kernel is applied for each filter: its $w \times h$ units apply the kernel to a different region of the input volume, that is, they apply the same operation on different parts of the input.

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 previous network layer
- Then, every layer of a convolutional network 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 hyper-parameters:

- **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 from different filters that are all looking at the same region of the input are denoted as **depth column**

Convolutional Layer

- **Stride**: step of the kernel 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 kernel 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.

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 hyper-parameter.

In the opposite case, they will be given by the ratio of w (and h) with the stride value.

Convolutional Layer

Usually $w = h$ and the kernel is defined as a square matrix of size f. In this case, the spatial size of the filter is a function of w, f, the stride s with which the kernel is moved, and the amount p of zero padding inserted on the border.
It is possible to see that the width of the filter is given by

$$w' = \left\lceil \frac{w - f + 2p}{s} \right\rceil + 1$$

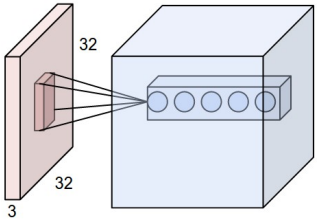
. For example for a 7x7 input and a 3x3 kernel with stride 1 and pad 0 we would get a 5x5 filter. With stride 2 we would get a 3x3 filter.

Convolutional Layer

How many connections are defined for each unit?

- this corresponds to the number of values in the input volume which are considered for computing each output value, plus the bias
- we assume that the input to each unit in the filter is composed by the output values of all units, for all depths, in the corresponding region of the previous layer, that is $f \times f \times d$ values

Convolutional Layer



Let the input to layer $r+1$ be a volume $32 \times 32 \times 3$.

Assume a kernel window of size $f = 3$: 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 filter at layer $r+1$ has width $w' = 16$.

If 5 features are extracted the number of units is then $16 \times 16 \times 5 = 1280$.

Example

- Real-world architecture (winner of the ImageNet challenge in 2012). 1.3×10^6 high-resolution images of size $227 \times 227 \times 3$ 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 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.

Example

- Overall, this results into $55 \times 55 \times 96 = 290.400$ neurons in the first convolutional layer, each with $11 \times 11 \times 3 + 1 = 364$ weights. This adds up to $290400 \times 364 = 105.705.600$ parameters just on the first layer of the convolutional network. 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 filter (**depth slice**) use the same weights and bias. With this parameter sharing scheme, the first layer in the example would now have only 96 unique set of weights (one for each filter), for a total of $96 \times 364 = 34.848$ unique weights.
- All 55×55 neurons in each depth slice use the same parameters.

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, in general a convolutional 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:
  - $w_2 = \left\lceil \frac{w_1 - f + 2p}{s} \right\rceil + 1$
  - $h_2 = \left\lceil \frac{h_1 - f + 2p}{s} \right\rceil + 1$
  - $d_2 = k$
- 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 reduces the spatial size of the representation to decrease the amount of parameters and computation in the network, and hence to also control overfitting.

The aggregation is performed internally to each filter (feature), decreasing its size.

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

A volume of values is produced such that:

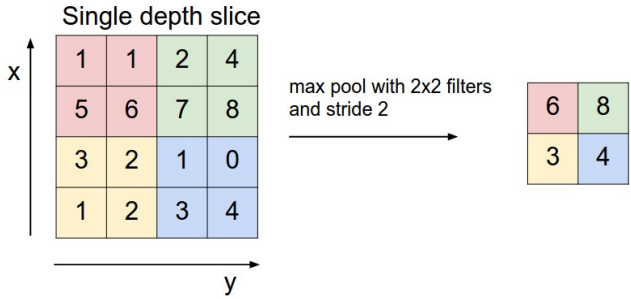
- the number of layers (depth) is not modified
- width and height are equal to the ones of the input volume, divided by the stride.

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



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.

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.

Layer Patterns

Most common form of a convolutional network architecture

Stack of a few convolutional-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`

In practice

One should rarely ever have to train a convolutional network 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 (transfer learning).

Example

Simple convolutional network for CIFAR-10 classification [INPUT - CONV - RELU - POOL- FC] architecture.

- INPUT [32x32x3] holds the raw pixel values of the image, in this case an image of width 32, height 32, and with 3 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 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.

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

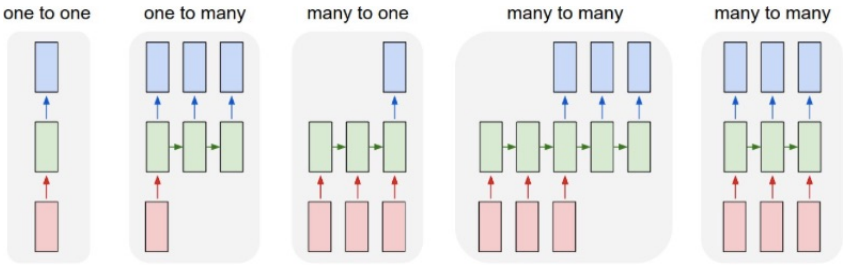
Many real-word problems require processing a timed (or in general indexed) sequence or signal

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

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

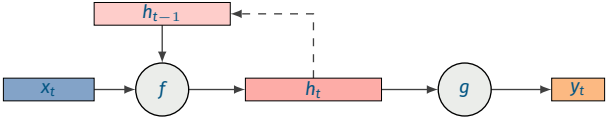


Recurrent neural networks

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

$$h_t = f(x_t, h_{t-1}) = \phi(W_1 x_t + W_2 h_{t-1} + b) \qquad t = 1, \ldots, T(x)$$

W_1, W_2 are weight matrices, learned by backpropagation

ϕ is usually a logistic or a hyperbolic tangent function

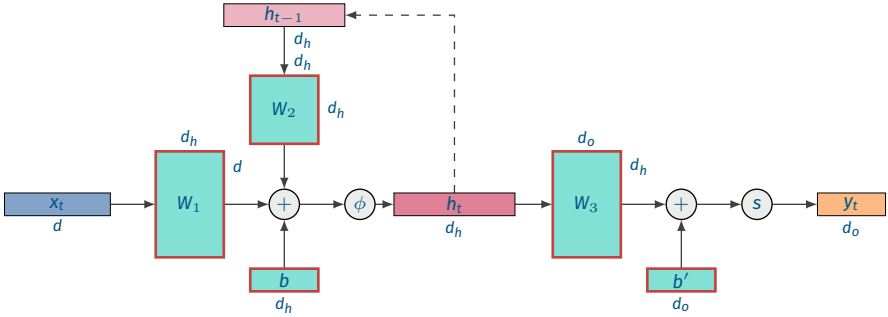
Recurrent neural networks

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

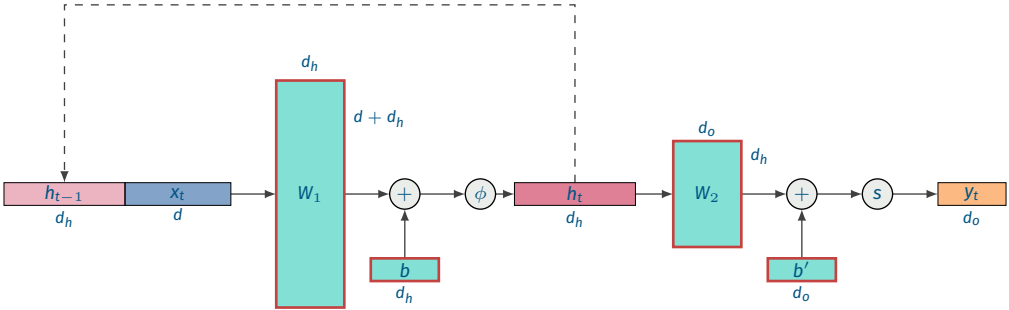
$$y_t = g(h_t) = \psi(W_3 h_t + b')$$

ψ is task-dependent. Usually, a softmax

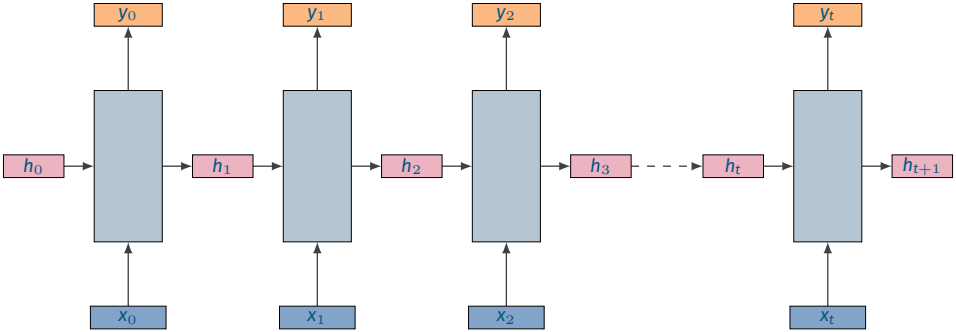
Recurrent neural networks



Recurrent neural networks: alternate view



Recurrent neural networks: unfolded view



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 $t' << t$

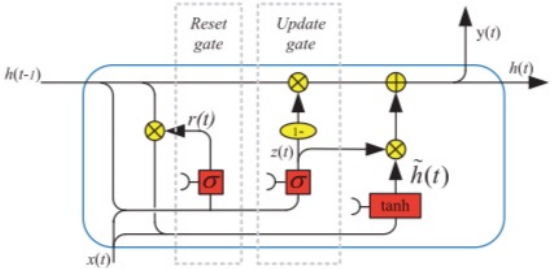
RNN, even if capable in principle, are not able in practice to deal with this problem (vanishing gradient). LSTM specialize the module structure by introducing a set of specialized interacting layers

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

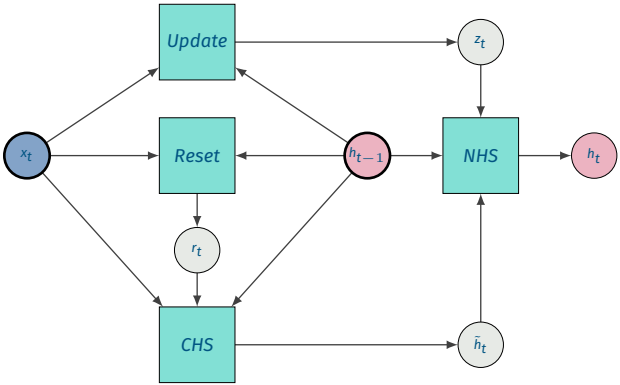
Gated Recurrent Unit

The Gated Recurrent Unit is one of the simplest and common types of LSTM, introduced to limit the number of networks coefficients to be learned wrt more complex LSTMs.

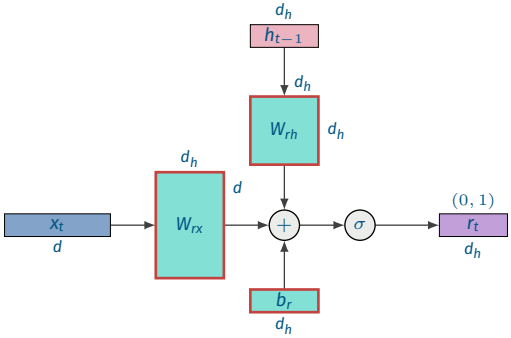


In GRUs, the new hidden state is computed as a combination of the previous state and of a new value, derived from the input by suitably weighing its components.

GRU



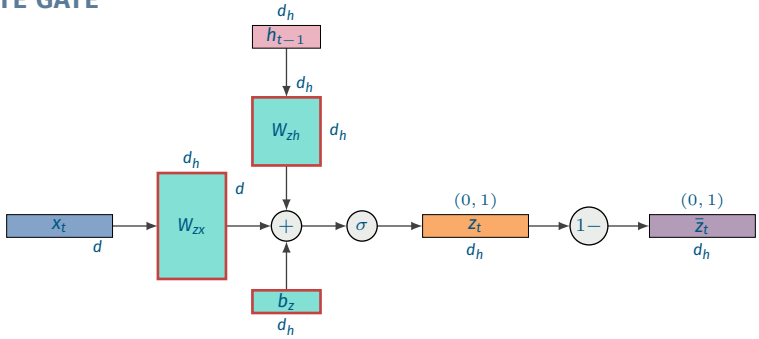
GRU: RESET GATE



$$r_t = \sigma(x_t W_{rx} + h_{t-1} W_{rh} + b_r)$$

specifies for each component of the hidden state how much the current state must be considered to modify the component value

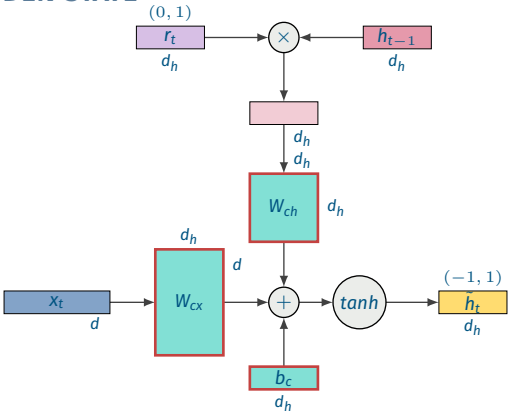
GRU: UPDATE GATE



$$z_t = \sigma(x_t W_{zx} + h_{t-1} W_{zh} + b_z)$$
$$\bar{z}_t = 1 - z_t$$

specifies for each component of the hidden state how much its next value should derive from the current one (lower z_t, higher $\bar{z}_t$) versus how much it must be derived from the current input (higher z_t, lower $\bar{z}_t$)

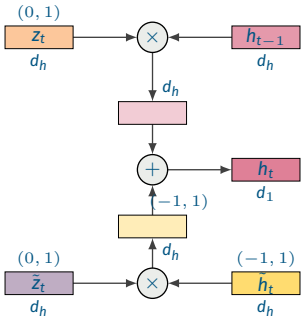
GRU: CANDIDATE HIDDEN STATE



$$\tilde{h}_t = tanh(x_t W_{cx} + h_{t-1} W_{ch} + b_c)$$

derives a new value for each component of the hidden state by combining the current input and the previous state value, weighted by r_t

GRU: NEW HIDDEN STATE

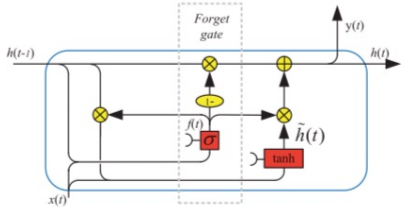


$$h_t = z_t h_{t-1} + \tilde{z}_t \tilde{h}_t = z_t h_{t-1} + (1 - z_t)\tilde{h}_t$$

derives a new value for each component of the hidden state as a convex combination of the old value and the candidate value derived from the input

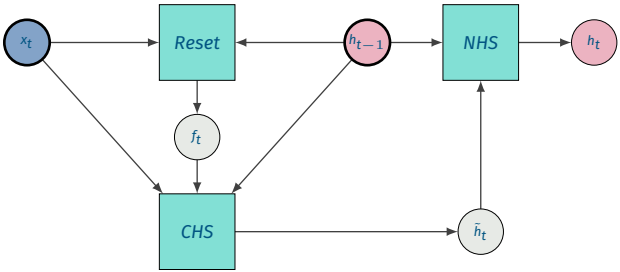
Minimal Gated Unit

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



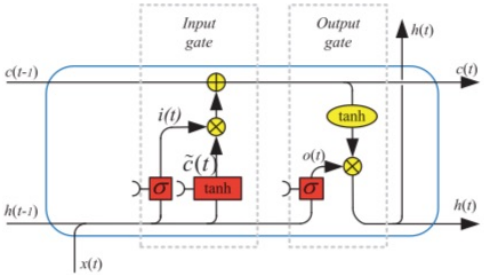
$$f_t = \sigma(x_t W_{fx} + h_{t-1} W_{fh} + b_f)$$
$$\tilde{h}_t = tanh(x_t W_{hx} + (f_t * h_{t-1}) W_{hh} + b_h)$$
$$h_t = (1 - f_t) * h_{t-1} + f_t * \tilde{h}_t$$

Minimal Gated Unit

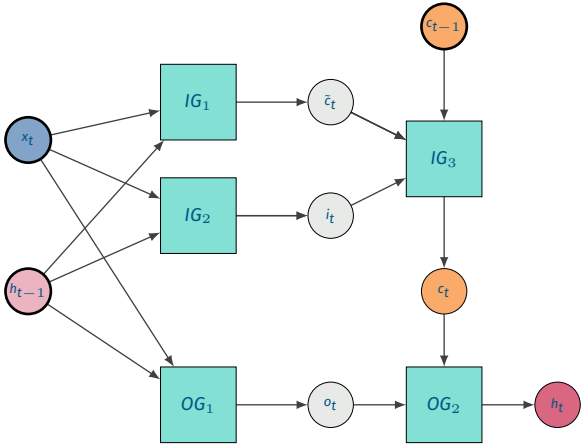


LSTM NETWORKS: FIRST VERSION

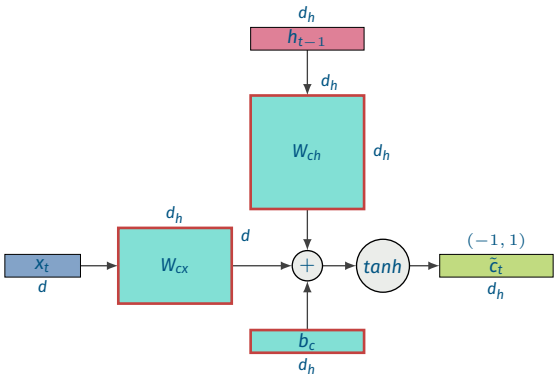
In an LSTM an additional long-term state c_t is maintained. No product, only sum (no forget)



LSTM NETWORK



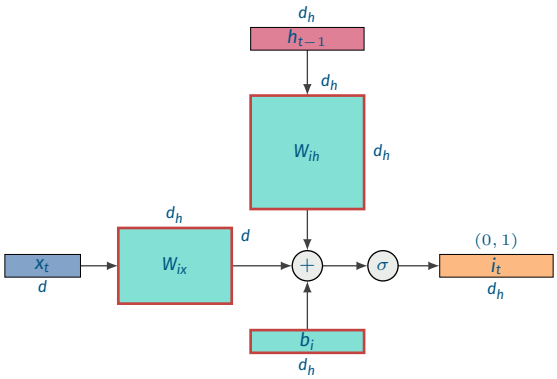
LSTM NETWORKS: INPUT GATE



$\tilde{c}_t$ specifies the increment or decrement of each long term memory component in c_{t-1}

$$\tilde{c}_t = tanh(h_{t-1}W_{ch} + x_tW_{cx} + b_c)$$

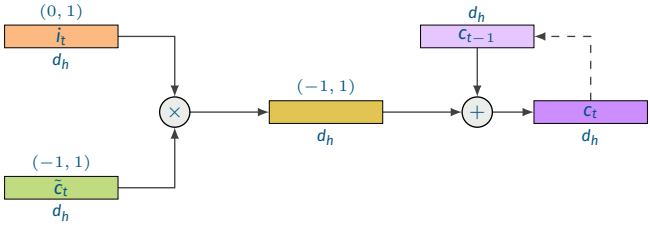
LSTM NETWORKS: INPUT GATE



i_t specifies how much each long term memory cell component in c_{t-1} has to be updated

$$i_t = \sigma(h_{t-1}W_{ih} + x_tW_{ix} + b_i)$$

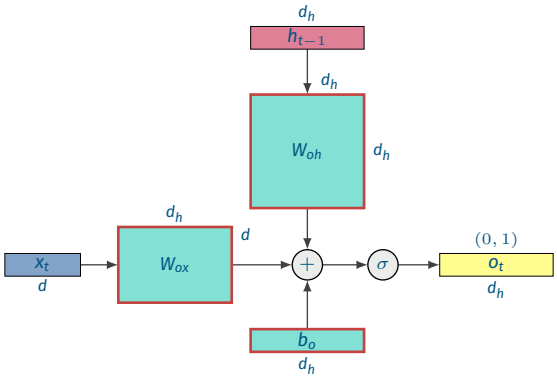
LSTM NETWORKS: INPUT GATE



Each long term memory component in c_{t-1} is updated by adding the value from $\tilde{c}_t$. Only a fraction, as specified by i_t, of the update is performed

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

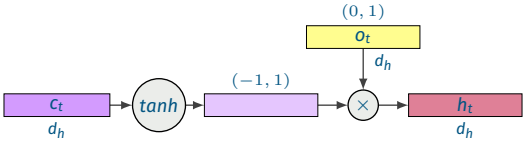
LSTM NETWORKS: OUTPUT GATE



o_t specifies how much each short term memory cell component in h_{t-1} has to be updated by c_t

$$o_t = \sigma(h_{t-1}W_{oh} + x_tW_{ox} + b_o)$$

LSTM NETWORKS: OUTPUT GATE

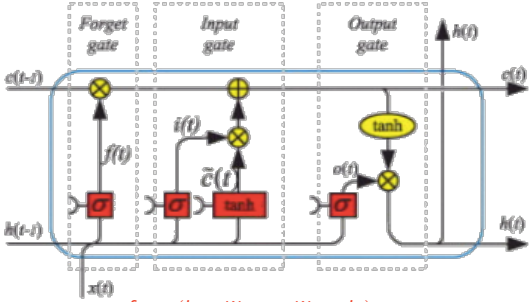


Updates to short term memory components are obtained by squashing long term memory components in $(-1, 1)$. Only a fraction of update is applied, as specified by o_t

$$h_t = o_t tanh(c_t)$$

LSTM NETWORKS WITH FORGET LAYER

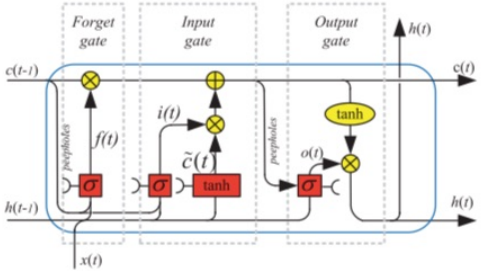
Forget gate layer: determines the decrease of relevance of long term memory components



$$f_t = \sigma(h_{t-1} W_{fh} + x_t W_{fx} + b_f)$$
$$i_t = \sigma(h_{t-1} W_{ih} + x_t W_{ix} + b_i)$$
$$\tilde{c}_t = tanh(h_{t-1} W_{ch} + x_t W_{cx} + b_c)$$
$$c_t = f_t c_{t-1} + i_t \tilde{c}_t$$
$$o_t = \sigma(h_{t-1} W_{oh} + x_t W_{ox} + b_o)$$
$$h_t = o_t tanh(c_t)$$

LSTM NETWORKS WITH PEEPHOLES

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



LSTM NETWORKS WITH PEEPHOLES

$$
\begin{aligned}
f_t &= \sigma(h_{t-1} W_{fh} + x_t W_{fx} + c_{t-1} W_{fc} + b_f) \\
i_t &= \sigma(h_{t-1} W_{ih} + x_t W_{ix} + c_{t-1} W_{ic} + b_i \\
\tilde{c}_t &= tanh(h_{t-1} W_{ch} + x_t W_{cx} + b_c) \\
c_t &= f_t c_{t-1} + i_t \tilde{c}_t \\
o_t &= \sigma(h_{t-1} W_{oh} + x_t W_{ox} + c_{t-1} W_{oc} + b_o) \\
h_t &= o_t \; tanh(c_t)
\end{aligned}
$$

Topologies

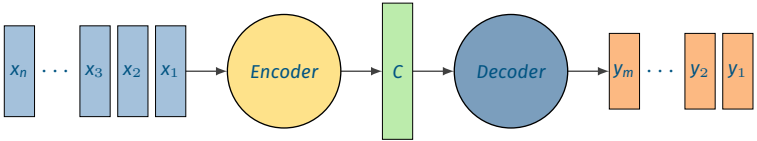
Recurring cells (in particolar LSTM) can be connected according to many different topologies.

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

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

Encoder-decoder architecture

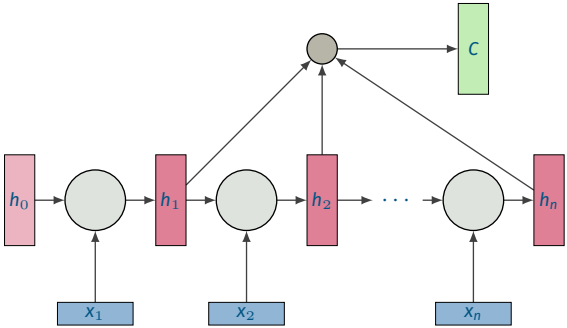
Applied for sequence-to-sequence translation



- The encoder produces a context from the whole input sequence, which provides a suitable overall coding of the sequence
- The decoder produces the output sequence step by step, from the context
- Both the encoder and the decoder can be implemented as RNN

Encoder

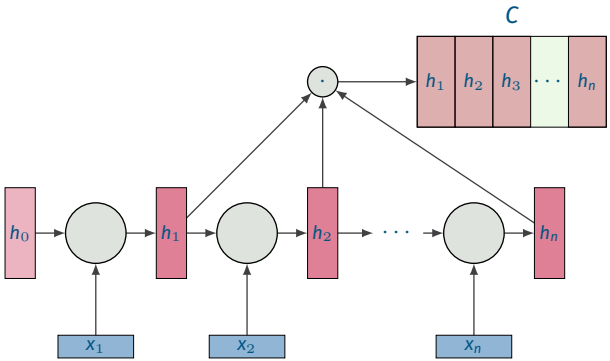
The context derives from the sequence of hidden states traversed by the network



$$h_t = f(x_t, h_{t-1})$$
$$C = q(h_1, \ldots, h_n)$$

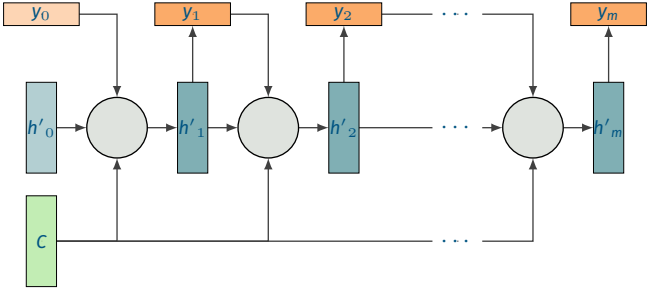
Encoder

A possible context is just the juxtaposition of all hidden states traversed



Decoder

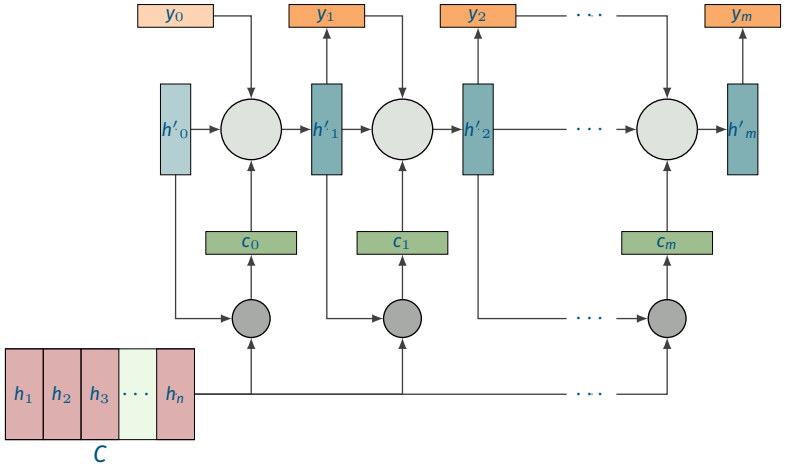
At each step both the output and the new hidden state derive from the current hidden state and the context encoding the input sequence (usually, also from the previous output)



$$h'_t = f(h'_{t-1}, y_{t-1}, c)$$
$$y_t = g(h'_t)$$

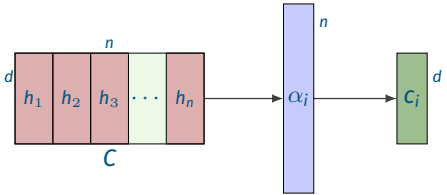
Attention

The attention mechanism computes different contexts for each decoder step by comparing the current state h' and the context C.



Attention

Each context c_i is usually computed as a non negative linear combination of the encoder hidden states, which are gathered in the context.



$$c_i = C \cdot \alpha_i$$

where

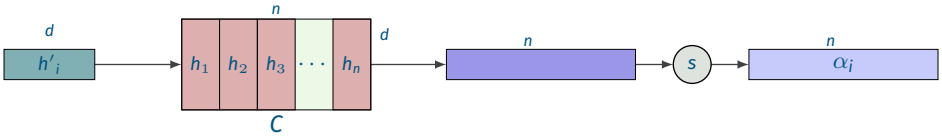
$$C = \begin{bmatrix} | & | & \cdots & | \\ h_1 & h_2 & \cdots & h_n \\ | & | & \cdots & | \end{bmatrix}$$

hence

$$c_i = \sum_{j=1}^{n} \alpha_{ij} h_j$$

Attention

The weights of the linear combination are derived by comparing the current decoder state h'_i with each encoder hidden state from the context.



$$\alpha_i = \text{softmax}(h'_i C)$$

Attention

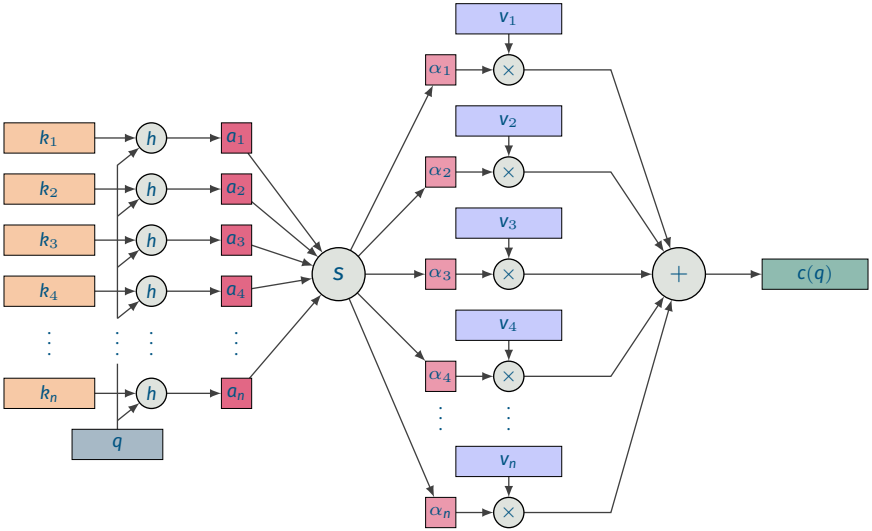
We may abstract the attention mechanism described above as follows:

- given three domains (vector spaces) Q (queries), K (keys) and V (values)
- given a function $h : Q \times K \mapsto \mathbb{R}$ or in particular $h : Q \times K \mapsto (0, 1)$
- let q be a query, a set of n keys $k_1, \ldots, k_n$ be n keys and $v_1, \ldots, v_n$ be n values
- a set of weights $\alpha_1, \ldots, \alpha_n$ can be computed as $\alpha_i = h(q, k_i)$
- the attention $a(q)$ associated to query q is the linear combination of $v_1, \ldots, v_n$ weighted by $\alpha_1, \ldots, \alpha_n$

$$a(q) = \sum_{i=1}^{n} \alpha_i v_i = \sum_{i=1}^{n} h(q, k_i) v_i$$

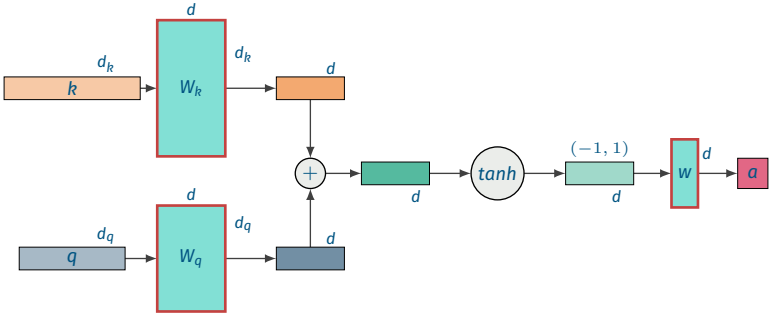
In the above approach, q is the current decoder state h'_i, while both K and V are the encoder states $h_1, \ldots, h_n$

Attention


k_1
k_2
k_3
k_4
k_n
q
h
h
h
h
h
a_1
a_2
a_3
a_4
a_n
s
v_1
v_2
v_3
v_4
v_n
α_1
α_2
α_3
α_4
α_n
$\times$
$+$
$c(q)$

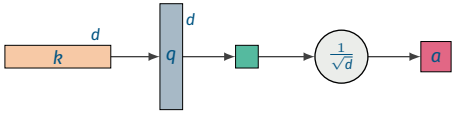
Additive attention

A possible tunable implementation of h. W_k, W_q and w provide $d(d_k + d_q + 1)$ learnable coefficients



Scaled down attention

A non tunable, simple implementation of h when queries and keys have the same size



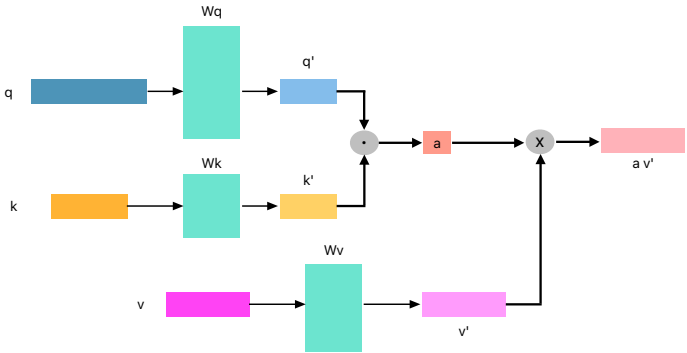
Let $\mathbf{Q} \in \mathbb{R}^{n \times d}$, $\mathbf{K} \in \mathbb{R}^{r \times d}$, $\mathbf{V} \in \mathbb{R}^{r \times m}$: that is assume there are n queries of size d, and r key-value pairs, with keys of same size d than queries and values of arbitrary size m. Then applying scaled down attention results into n vectors α_i of size v, computed as the columns of

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}$$

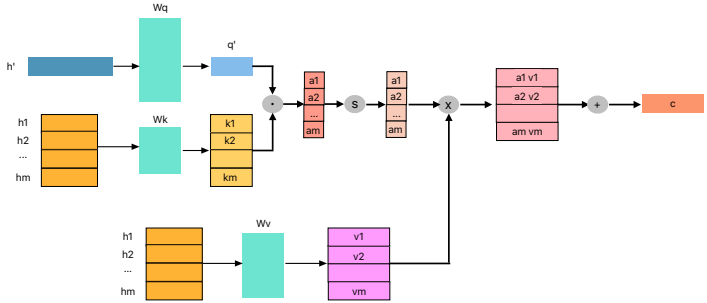
Usually, $r = n$, which results in the method returning a context of size m for each input vector.

MAKING SCALED DOWN TUNABLE

Again, introducing matrices of parameters makes the method tunable

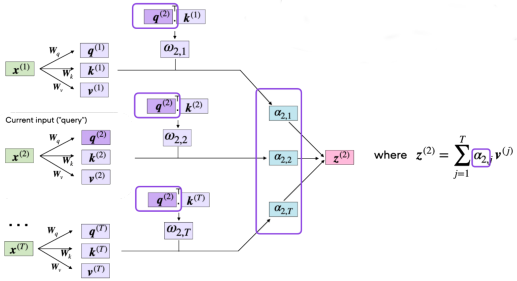


Scaled down in context derivation

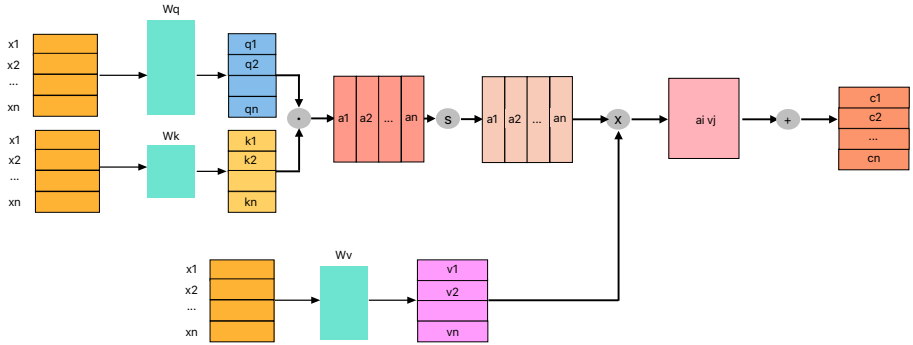


Self-attention

In self-attention, the attention mechanism is applied as shown above, but without referring to hidden states. Instead, queries $q_1, \ldots, q_n$, keys $k_1, \ldots, k_n$, and values $v_1, \ldots, v_n$ are all derived from input tokens $x_1, \ldots, x_n$

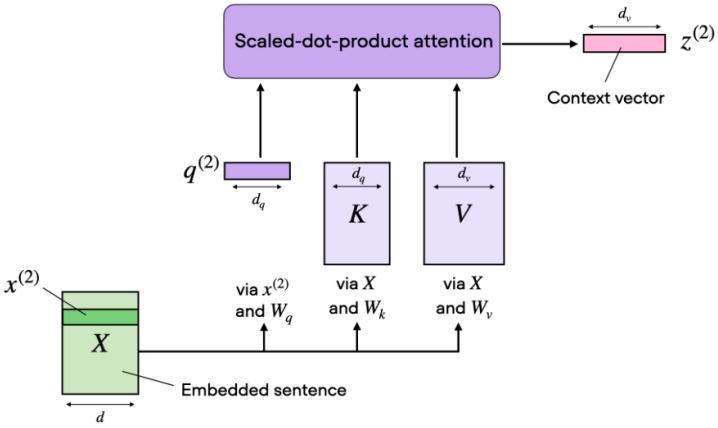


Self-attention



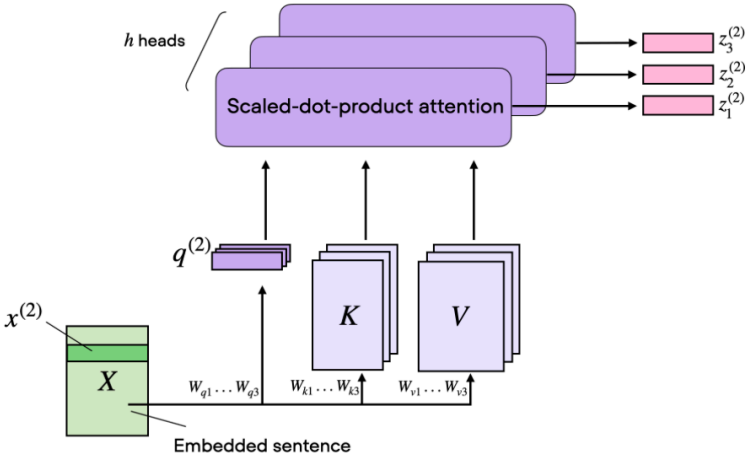
MULTI HEAD

Single head:



Multi head

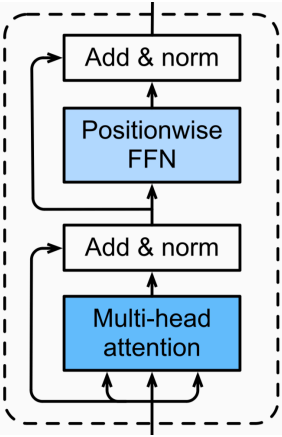
The same operations are done in parallel (with different W_q, W_k, W_v triples), deriving K contexts for each token.



Transformer: encoder

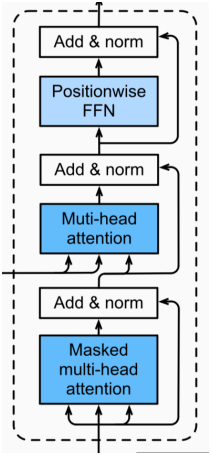
Both encoder and decoder are composed of sequences of blocks.

Encoder block:

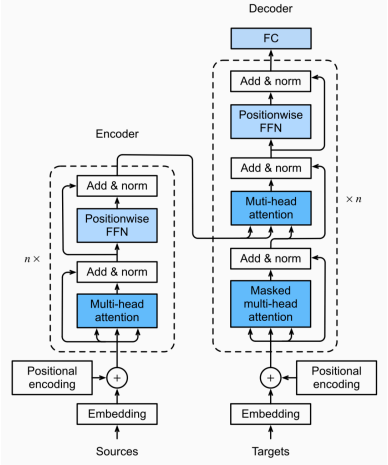


TRANSFORMER: DECODER

Decoder block:



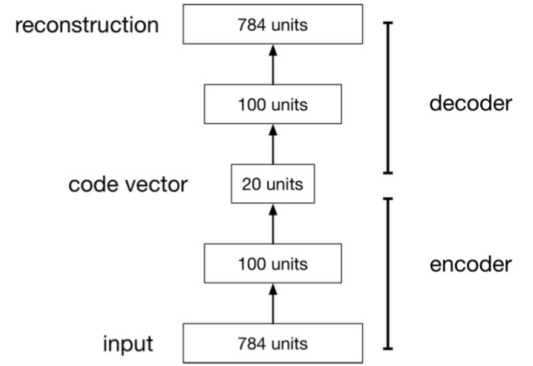
Transformer architecture



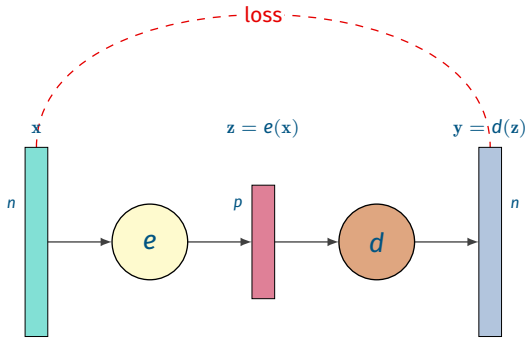
Autoencoders

- 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



Autoencoders

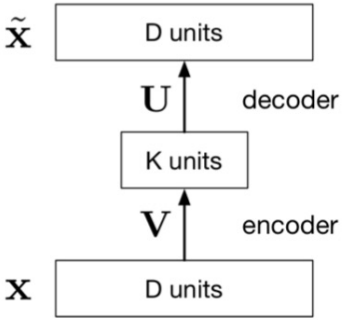


Why autoencoders?

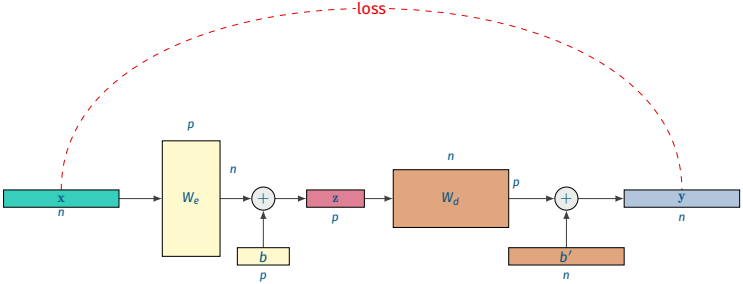
- Map high-dimensional data to two dimensions for visualization
- Compression (i.e. reducing the file size)
- Learn abstract features in an unsupervised way so you can apply them to a supervised task

Linear autoencoder

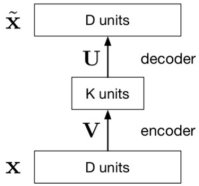
- The simplest kind of autoencoder has one hidden layer, linear activations, and squared error loss $L(x_1, x_2) = || x_1 - x_2 ||^2$



Linear autoencoder



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

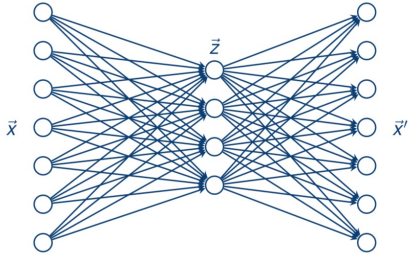
- What is the best possible mapping to choose?
- It is well known that the best mapping corresponds to the column of U being the eigenvectors of the scatter matrix $S = \frac{1}{n} \sum_{i=1}^{n} (x_i - m)(x_i - m)^T$ corresponding to the K largest eigenvalues

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

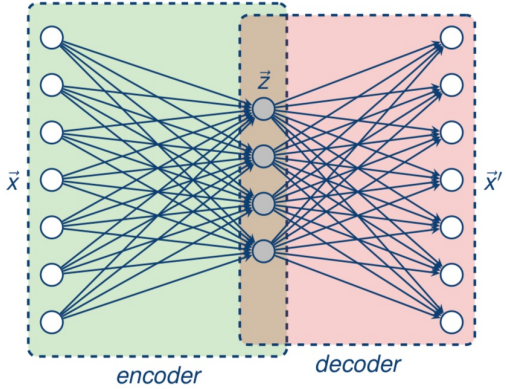
Alternative perspective

- 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

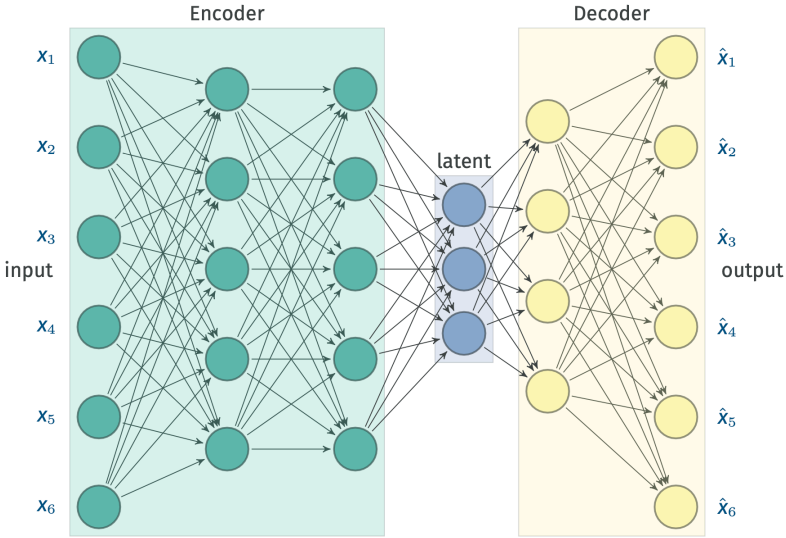
- Encoder: $z = \sigma(Vx)$
- Decoder: $x' = g(Uz) = g(U\sigma(Vx))$
- g could be the identity if x is defined on real values or σ if they are binary $\{0, 1\}$
- Loss functions vary accordingly from squared loss $|| x - x' ||^2$ to cross entropy $-(x \log x' + (1 - x) \log(1 - x'))$

Deep autoencoders

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

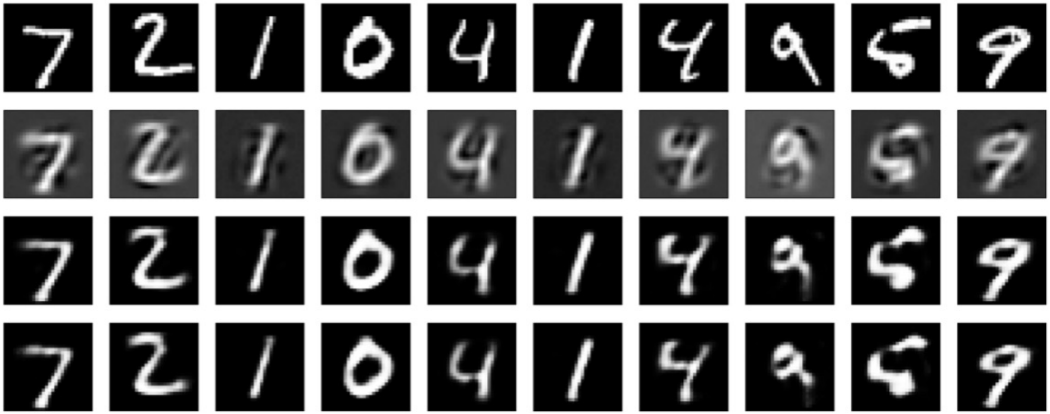
$$z = r(W_2 r(W_1 x))$$
$$x = g(W_4 r(W_3 z))$$

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.
One way to deal with this aim is to enforce sparsity on the hidden activations. This can be added on top of the bottleneck enforcement, or instead of it.
The first way to do so, is to apply L_1 regularization, which is known to induce sparsity. Thus, the autoencoder optimization objective becomes:

$$\underset{\boldsymbol{\theta},\boldsymbol{\phi}}{\operatorname{argmin}} \left(\mathbb{E}_{\mathbf{x}} \left[\mathcal{L}(\mathbf{x}, \boldsymbol{d}(\boldsymbol{e}(\mathbf{x};\boldsymbol{\theta}),\boldsymbol{\phi})) \right] + \lambda \sum_i |a_i| \right)$$

where a_i is the activation at the ith hidden layer and i iterates over all the hiddens activations.

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

Sparse autoencoder

Another way to do so, is to use the KL-divergence.
Instead of tweaking the λ parameter as in the L_1 regularization, we can assume the activation of each neuron acts as a Bernoulli variable with probability π and tweak that probability.
At each batch, the actual probability is then measured, and the difference is calculated and applied as a regularization factor.
Let s_l be the size of the l-th hidden layer; the calculated empirical probability for the j-th neuron in this layer is then $\hat{\pi}_j^{(l)} = \frac{1}{n} \sum_{i=1}^{n} a_j^{(l)}(\mathbf{x}_i)$, where $a_j^{(l)}(\mathbf{x}_i)$ is the value of the activation of neuron j in layer l when the input is $\mathbf{x}_i$.
Then, we wish the empirical probability of activation of this neuron to be approximately equal to a small constant probability π, known also as sparsity parameter, that is

$$\hat{\pi}_j^{(l)} \approx \pi$$

Sparse autoencoder

This constraint is achieved by adding a penalty term into the loss function, related to the difference (measured by the KL-divergence) between a Bernoulli distribution with probability π and another Bernoulli distribution with probability $\hat{\pi}_j^{(l)}$. The overall loss function includes then a penalty term which is the sum of the penalties for each neuron (here L denotes the overall number of layers).

$$\underset{\boldsymbol{\theta},\boldsymbol{\phi}}{\operatorname{argmin}} \left(\mathbb{E}_{\mathbf{x}} \left[\mathcal{L}(\mathbf{x}, \boldsymbol{d}(\boldsymbol{e}(\mathbf{x};\boldsymbol{\theta}),\boldsymbol{\phi})) \right] + \sum_{l=1}^{L} \sum_{j=1}^{s_l} D_{KL} \left(\pi || \hat{\pi}_j^{(l)} \right) \right)$$

KL DIVERGENCE

The Kullback-Leibler divergence between $p_1(z)$ and $p_2(z)$, measures the difference between the two distributions.

$$D_{KL}\left(p_1(\mathbf{z})||p_2(\mathbf{z})\right) = \mathbb{E}_{\mathbf{z}\sim p_1(\mathbf{z})}\left[\log \frac{p_1(\mathbf{z})}{p_2(\mathbf{z})}\right] = \int_{\mathcal{Z}} p_1(\mathbf{z}) \log \frac{p_1(\mathbf{z})}{p_2(\mathbf{z})} d\mathbf{z}$$

The following basic properties of *KL* divergence are relevant

$$D_{KL}\left(p_1(\mathbf{z})||p_2(\mathbf{z})\right) \geq 0$$
$$D_{KL}\left(p_1(\mathbf{z})||p_2(\mathbf{z})\right) = 0 \quad \text{iff} \quad p_1(\mathbf{z}) = p_2(\mathbf{z}) \forall \mathbf{z}$$

Denoising autoencoder

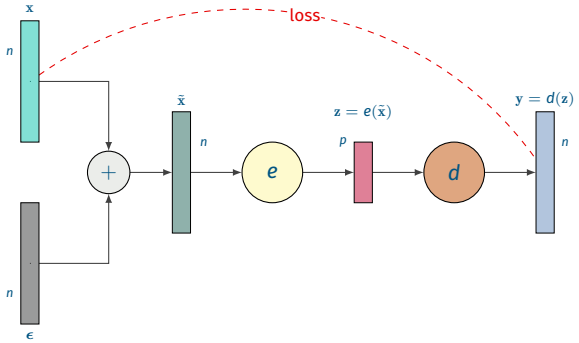
- An autoencoder that is robust to noise.
- Assume an additional noise ε, so that $\tilde{x} = x + \varepsilon$.
- Then, the loss function of the autoencoder could be defined as

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

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

the input $\mathbf{x}$ is partially corrupted to $\tilde{\mathbf{x}}$. This is obtained either by adding Gaussian noise to or by masking some values of the input vector in a stochastic manner, for example according to a Bernoulli distribution.
Note that in both cases $\tilde{\mathbf{x}}$ is a random variable derived by composing $\mathbf{x}$ and a random noise.
The autoencoder is expected to reconstruct the clean version $\mathbf{x}$ of the input from the corrupted value $\tilde{\mathbf{x}}$

Denoising autoencoder



Observe that the model is trained to recover the original input, not the corrupt one. That is, the loss function compares the input value $\mathbf{x}$ and the output value $d(e(\tilde{\mathbf{x}}))$ as follows

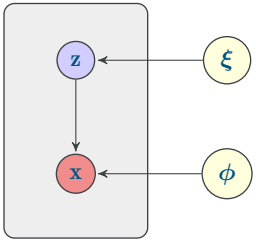
$$(\boldsymbol{\theta}^*, \boldsymbol{\phi}^*) = \underset{\boldsymbol{\theta},\boldsymbol{\phi}}{\operatorname{argmin}} \; \mathbb{E}_{\mathbf{x}} \left[\mathcal{L}(\mathbf{x}, d(e(\tilde{\mathbf{x}}; \boldsymbol{\theta}), \boldsymbol{\phi})) \right]$$

VARIATIONAL AUTOENCODER

The idea of a variational autoencoder derives from that of latent variable models, where observed data items are assumed distributed according to the model

$$\mathbf{z} \sim p(\mathbf{z}; \boldsymbol{\xi})$$
$$\mathbf{x} \sim p(\mathbf{x}|\mathbf{z}; \boldsymbol{\phi})$$

or, in graphical form,



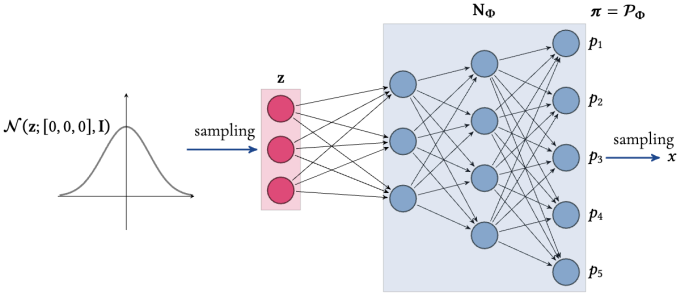
Deep latent variable models

In a deep latent variable model, we assume that:

1. the latent variable $\mathbf{z}$ has a very simple distribution, such as for example $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
2. the dependance of the observed variable from the latent one, modeled by the distribution $p(\mathbf{x}|\mathbf{z}; \phi)$ can be fairly complex, thus making its marginal distribution $p(\mathbf{x}; \phi) = p(\mathbf{z})p(\mathbf{x}|\mathbf{z}; \phi)$ arbitrarily complex
3. in particular, we assume that $p(\mathbf{x}|\mathbf{z}; \phi) = \mathcal{P}(\mathbf{x}; f_{\Phi}(\mathbf{z}))$, where $f_{\Phi}(\mathbf{z})$ is a complex parametric function and $\mathcal{P}$ is a parametric distribution
4. finally, we assumed that $f_{\Phi}(\mathbf{z})$ is implemented by a deep neural network N_{Φ} denoted as decoder

Deep latent variable models

Example of gaussian-categorical DLVM with $\mathbf{z} \in \mathbb{R}^3$ and $x \in \{1, \dots, 5\}$

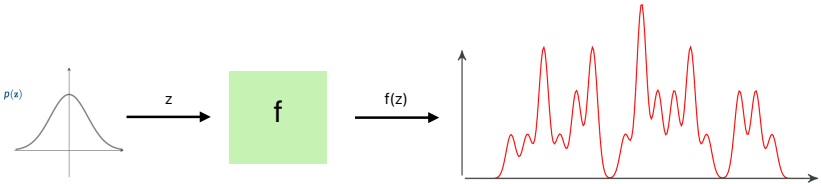


Deep latent variable models

The role of the decoder N_{Φ} is:

- to transform the code $\mathbf{z}$ into a specific distribution on the domain of $\mathbf{x}$ by computing its parameters as $f_{\Phi}(\mathbf{z})$
- the decoder parameters Φ are learned from data

Deep latent variable models



Learning the parameters of a latent variable model

Given an observation $\overline{\mathbf{x}}$, we learn a parameter value ϕ^* by maximizing the log-likelihood

$$\log p(\overline{\mathbf{x}}; \phi)$$

where

$$p(\overline{\mathbf{x}}; \phi) = \int p(\overline{\mathbf{x}}|\mathbf{z}; \phi) p(\mathbf{z}) d\mathbf{z}$$

Unfortunately, computing this integral is usually intractable, making $p(\overline{\mathbf{x}}; \phi)$ practically impossible to evaluate. Moreover, $p(\mathbf{z}|\overline{\mathbf{x}}; \phi)$ is intractable too.

Variational inference

Since $p(\mathbf{x};\phi)$ cannot be computed, variational inference approximately maximizes the log-likelihood by maximizing the evidence lower bound

$$\text{ELBO}(\phi, q, \mathbf{x}) = \mathbb{E}_{\mathbf{z}\sim q}\left[\log \frac{p(\mathbf{x},\mathbf{z};\phi)}{q(\mathbf{z})}\right] = \log p(\mathbf{x};\phi) - D_{KL}\left(q(\mathbf{z})||p(\mathbf{z}|\mathbf{x};\phi)\right) \leq \log p(\mathbf{x};\phi)$$

where $q(\mathbf{z})$ is any distribution and the gap between the ELBO and the log-likelihood is smaller for functions more similar to $p(\mathbf{z}|\mathbf{x};\phi)$.
In order to maximize the ELBO, a sequence of two-phases steps is performed:

1. for a given ϕ, find the distribution $q(\mathbf{z})$ which better approximates $p(\mathbf{z}|\mathbf{x};\phi)$, thus making the gap between the ELBO and the log-likelihood as small as possible
2. for the given $q(\mathbf{z})$, maximize the ELBO wrt ϕ

Variational inference

We have to define a set of distributions $\mathcal{Q}$ within which we may search for the best q for any ϕ.

Usually, this is a parametric class of distributions $\mathcal{Q}(\mathbf{z}; \psi)$, which makes the ELBO a function of the parameters

$$\text{ELBO}(\phi, \psi, \mathbf{x}) = \mathbb{E}_{\mathbf{z} \sim \mathcal{Q}(\mathbf{z}; \psi)} \left[\log \frac{p(\mathbf{x}, \mathbf{z}; \phi)}{\mathcal{Q}(\mathbf{z}; \psi)} \right] = \log p(\mathbf{x}; \phi) - D_{KL} \left(\mathcal{Q}(\mathbf{z}; \psi) || p(\mathbf{z}|\mathbf{x}; \phi) \right) \leq \log p(\mathbf{x}; \phi)$$

Maximizing the log-likelihood of $\mathbf{x}$ then results into finding values ϕ^*, ψ^* which make

$\text{ELBO}(\phi^*, \psi^*, \mathbf{x})$ as large as possible, hence:

- $\mathcal{Q}(\mathbf{z}; \psi^*)$ is as similar as possible to $p(\mathbf{z}|\mathbf{x}; \phi^*)$
- $p(\mathbf{x}; \phi^*)$ is as large as possible

Variational inference

Given a set $\mathbf{X}$ of observations, the ELBO is

$$\text{ELBO}(\boldsymbol{\phi}, \langle \boldsymbol{\psi}_1, \dots, \boldsymbol{\psi}_n \rangle, \mathbf{X}) = \sum_{i=1}^{n} \mathbb{E}_{\mathbf{z} \sim \mathcal{Q}(\mathbf{z}; \boldsymbol{\psi}_i)} \left[\log \frac{p(\mathbf{x}_i, \mathbf{z}; \boldsymbol{\phi})}{\mathcal{Q}(\mathbf{z}; \boldsymbol{\psi}_i)} \right] \leq \sum_{i=1}^{n} \log p(\mathbf{x}_i; \boldsymbol{\phi})$$

The procedure above would result into:

1. for a given $\boldsymbol{\phi}$ and for each $\mathbf{x}_i$, find the value $\boldsymbol{\psi}_i$ such that $\mathcal{Q}(\mathbf{z}; \boldsymbol{\psi}_i)$ better approximates $p(\mathbf{z}|\mathbf{x}_i; \boldsymbol{\phi})$, thus making the gap between the ELBO and the contribution of $\mathbf{x}_i$ to the log-likelihood as small as possible
2. for the given set of values $\boldsymbol{\psi}_1, \dots, \boldsymbol{\psi}_n$, maximize the $\text{ELBO}(\boldsymbol{\phi}, \langle \boldsymbol{\psi}_1, \dots, \boldsymbol{v}_n \rangle, \mathbf{X})$ wrt $\boldsymbol{\phi}$

this then results into finding a different distribution $\mathcal{Q}(\mathbf{z}; \boldsymbol{\psi}_i)$ for each $\mathbf{x}_i$ for each step, which is costly for large datasets.

Amortized variational inference

Since the role of $\mathcal{Q}(\mathbf{z};\psi_i)$ is to provide a good (the best, ideally, within the class of distributions considered) approximation of $p(\mathbf{z}|\mathbf{x}_i;\phi)$, we may, in order to save computing time, define a function g_Ψ such that $\psi_i = g_\Psi(\mathbf{x}_i)$, thus making the ELBO defined as

$$\begin{aligned}
\text{ELBO}(\Phi,\Psi,\mathbf{X}) &= \sum_{i=1}^{n} \mathbb{E}_{\mathbf{z}\sim\mathcal{Q}(\mathbf{z};g_\Psi(\mathbf{x}_i))}\left[\log \frac{\mathcal{P}(\mathbf{x}_i;f_\Phi(\mathbf{z}))p(\mathbf{z})}{\mathcal{Q}(\mathbf{z};g_\Psi(\mathbf{x}_i))}\right] \\
&= \sum_{i=1}^{n} \left(\mathbb{E}_{\mathbf{z}\sim\mathcal{Q}(\mathbf{z};g_\Psi(\mathbf{x}_i))}\left[\log \mathcal{P}(\mathbf{x}_i;f_\Phi(\mathbf{z}))\right] - D_{KL}\left(\mathcal{Q}(\mathbf{z};g_\Psi(\mathbf{x}_i))||p(\mathbf{z})\right)\right) \\
&\leq \sum_{i=1}^{n} \log p(\mathbf{x}_i;\phi)
\end{aligned}$$

VARIATIONAL AUTOENCODER

In summary:

- In a variational autoencoder given an input $\mathbf{x}$ we do not want to map it into a latent vector $\mathbf{z} = e(\mathbf{x}; \boldsymbol{\theta})$ but, instead, into a distribution $p(\mathbf{z}; \boldsymbol{\psi})$, parameterized by $\boldsymbol{\psi} = E_{\Psi}(\mathbf{x})$, on a predefined latent space.
- At the same time, given a latent value $\mathbf{z}$ we do not want to map it into a vector $\mathbf{x} = d(\mathbf{z}; \boldsymbol{\theta})$ but, instead, into a distribution $p(\mathbf{x}; \boldsymbol{\phi})$, parameterized by $\boldsymbol{\phi} = D_{\Phi}(\mathbf{z})$
- A VAE, instead of learning the parameters of functions $h()$ and $g()$ mapping values into values, learns the parameters of functions E and D mapping values into distributions
- This results into a probabilistic version of an autoencoder

Variational autoencoder

An example:

- $\mathcal{P}$ is the set of gaussian distributions: $E_{\boldsymbol{\Phi}}(\mathbf{x})$ provides the parameters $\boldsymbol{\mu}(\mathbf{x}), \boldsymbol{\Sigma}(\mathbf{x})$ of a distribution $\mathcal{N}(\mathbf{z}; \boldsymbol{\mu}(\mathbf{x}), \boldsymbol{\Sigma}(\mathbf{x}))$ on latent space
- $\mathcal{Q}$ is the set of categorical distributions: $D_{\boldsymbol{\Psi}}(\mathbf{z})$ provides the parameters $\boldsymbol{\pi}(\mathbf{z})$ of a categorical distribution $\mathcal{C}(\mathbf{x}; \boldsymbol{\pi}(\mathbf{z}))$ on observed space

The ELBO results:

$$\begin{aligned}\text{ELBO}(\boldsymbol{\Phi}, \boldsymbol{\Psi}, \mathbf{X}) &= \sum_{i=1}^{n} \mathbb{E}_{\mathbf{z}\sim\mathcal{N}(\mathbf{z};\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i)} \left[\log \frac{\mathcal{C}(\mathbf{x}; \boldsymbol{\pi}_i)}{\mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)}\right] \\ &= \sum_{i=1}^{n} \left(\mathbb{E}_{\mathbf{z}\sim\mathcal{N}(\mathbf{z};\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i)} \left[\log \mathcal{C}(\mathbf{x}; \boldsymbol{\pi}_i)\right] - \mathbb{E}_{\mathbf{z}\sim\mathcal{N}(\mathbf{z};\boldsymbol{\mu}_i,\boldsymbol{\Sigma}_i)} \left[\mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)\right]\right)\end{aligned}$$

where $\boldsymbol{\mu}_i = \boldsymbol{\mu}_{\boldsymbol{\Psi}}(\mathbf{x}_i)$, $\boldsymbol{\Sigma}_i = \boldsymbol{\Sigma}_{\boldsymbol{\Psi}}(\mathbf{x}_i)$ and $\boldsymbol{\pi}_i = \boldsymbol{\pi}(\mathbf{z})$.

Variational autoencoder

The ELBO can be evaluated by iterating on all $\mathbf{x}_i$ and, at each iteration:

1. compute $\boldsymbol{\mu}_i$ and $\boldsymbol{\Sigma}_i$ through the encoder
2. approximating the expectations, by means of a sequence of K samples of $\mathbf{z}$ from $\mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$
3. for each sampled $\mathbf{z}$, compute $\boldsymbol{\pi}$ through the decoder
4. compute the expectations of $\mathcal{N}(\mathbf{z}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i)$ and $\mathcal{C}(\mathbf{x}; \boldsymbol{\pi}_i)$

Loss function

The loss function, to be minimized is defined as the opposite of the ELBO, that is,

$$L = L_{rec} + L_{KL} = -\log p(\mathbf{x}; \mathbf{\Phi}, \mathbf{\Psi}) + D_{KL}\left(\mathcal{Q}(\mathbf{z}; g_{\mathbf{\Psi}}(\mathbf{x}))||p(\mathbf{z}|\mathbf{x}; \boldsymbol{\phi})\right)$$

where $L_{rec} = -\sum_{i=1}^{n} \log p(\mathbf{x}_i; \mathbf{\Phi}, \mathbf{\Psi})$ is the reconstruction loss, related to the probability of producing the original dataset $\mathbf{X}$, and $L_{KL} = \sum_{i=1}^{n} D_{KL}\left(\mathcal{Q}(\mathbf{z}; g_{\mathbf{\Psi}}(\mathbf{x}_i))||p(\mathbf{z}|\mathbf{x}_i; \boldsymbol{\phi})\right)$ is the sum of the differences between the exact posterior distribution $p(\mathbf{z}|\mathbf{x}_i; \boldsymbol{\phi})$ and its approximation $\mathcal{Q}(\mathbf{z}; g_{\mathbf{\Psi}}(\mathbf{x}_i))$.

As shown above,

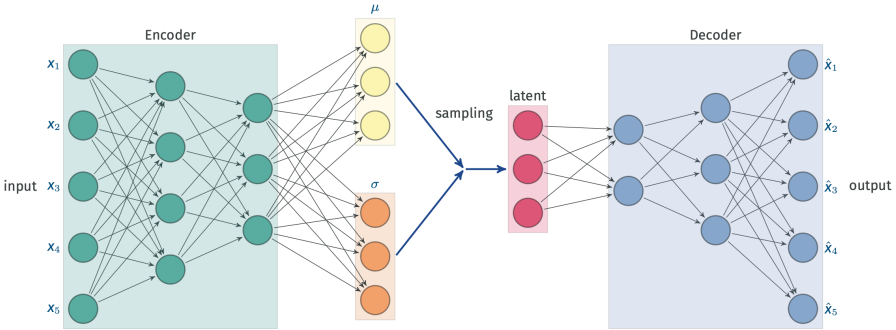
$$L = -\mathbb{E}_{\mathbf{z}\sim\mathcal{Q}(\mathbf{z}; g_{\mathbf{\Psi}}(\mathbf{x}))}\left[\log \mathcal{P}(\mathbf{x}; f_{\mathbf{\Phi}}(\mathbf{z}))\right] + D_{KL}\left(\mathcal{Q}(\mathbf{z}; g_{\mathbf{\Psi}}(\mathbf{x}))||p(\mathbf{z})\right)$$

which can be interpreted as the expected reconstruction provided by the model plus a regularization term which penalizes approximations of the posterior distribution which are different from the prior $p(\mathbf{z})$

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

VARIATIONAL ENCODER AS NEURAL NETWORK



Generative models

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

Fake face/text/music generation

- 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
- Diffusion models
- Large language models