ML: Lecture 7

Autoencoders, transformers, and attention

Panagiotis Papapetrou, PhD
Professor, Stockholm University

Syllabus

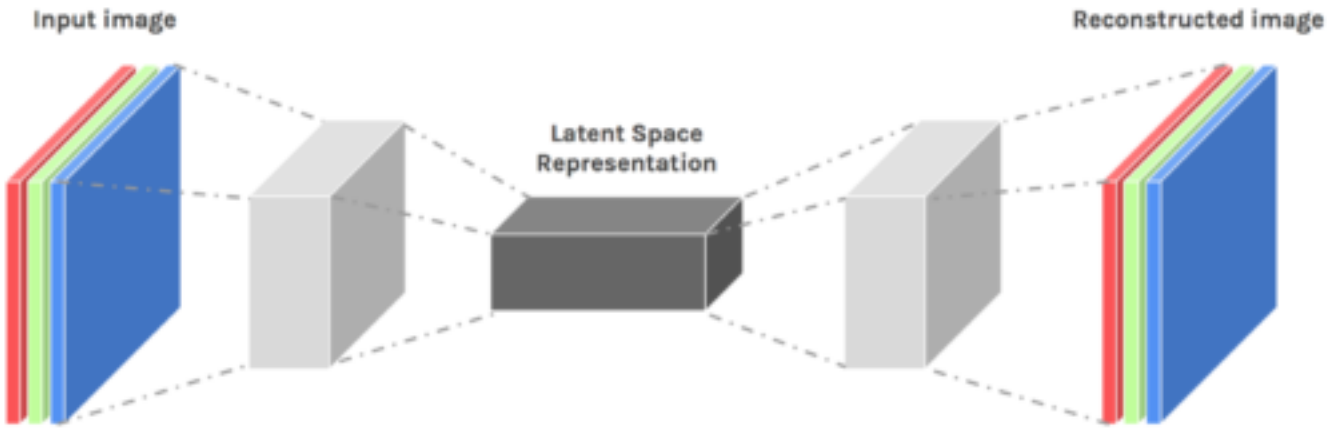
Jan 16	Introduction to machine learning
Jan 18	Regression analysis
Jan 19	Laboratory session 1: numpy and linear regression
Jan 23	Ensemble learning
Jan 25	Deep learning I: Training neural networks
Jan 26	Laboratory session 2: ML pipelines, ensemble learning
Jan 30	Deep learning II: Convolutional neural networks
Feb 1	Laboratory session 3: training NNs and tensorflow
Feb 6	Deep learning III: Recurrent neural networks
Feb 8	Laboratory session 4: CNNs and RNs
Feb 13	**Deep learning IV: Autoencoders, transformers, and attention**
Feb 20	Time series classification

Today

- What are autoencoders?
- How to use autoencoders for image processing?
- What is attention?
- The architecture of transformers

Autoencoders

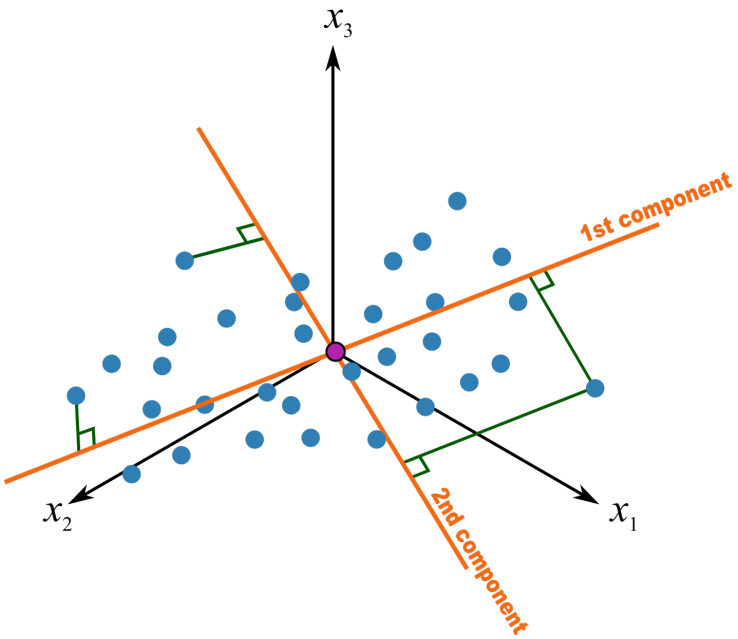
- Autoencoders are neural networks that are designed to learn encodings/feature spaces from data
  - How? By reproducing the input from a learned encoding



https://www.edureka.co/blog/autoencoders-tutorial/

Autoencoders

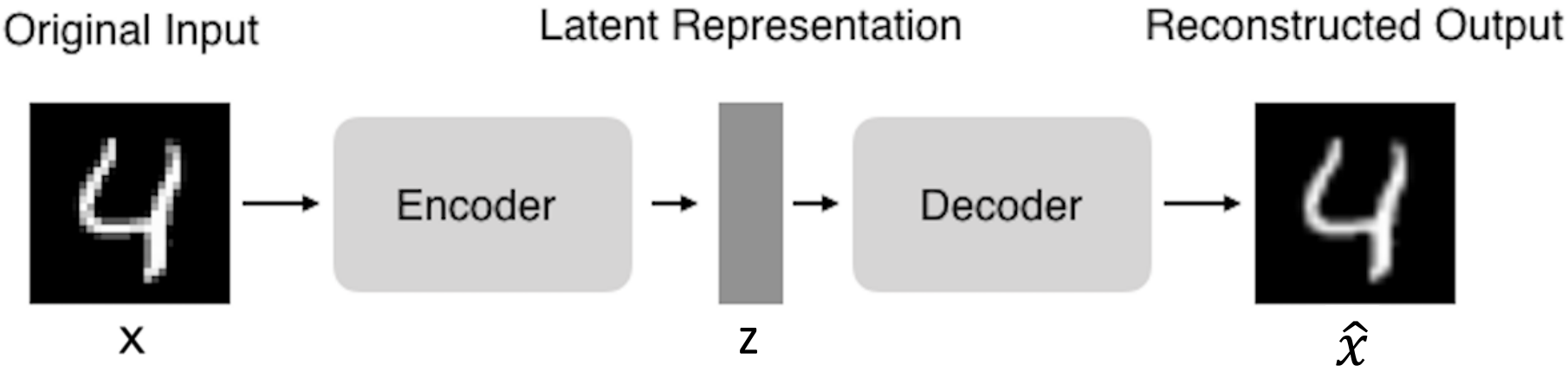
- **Compare PCA/SVD** [Invented by Karl Pearson in 1901]
  - PCA takes a collection of vectors (images) and produces a usually smaller set of vectors that can be used to approximate the input vectors via a linear combination
  - very efficient for certain applications
- **Neural network autoencoders**
  - can learn nonlinear dependencies
  - can use convolutional layers
  - can use transfer learning



https://www.edureka.co/blog/autoencoders-tutorial/

Autoencoders: structure

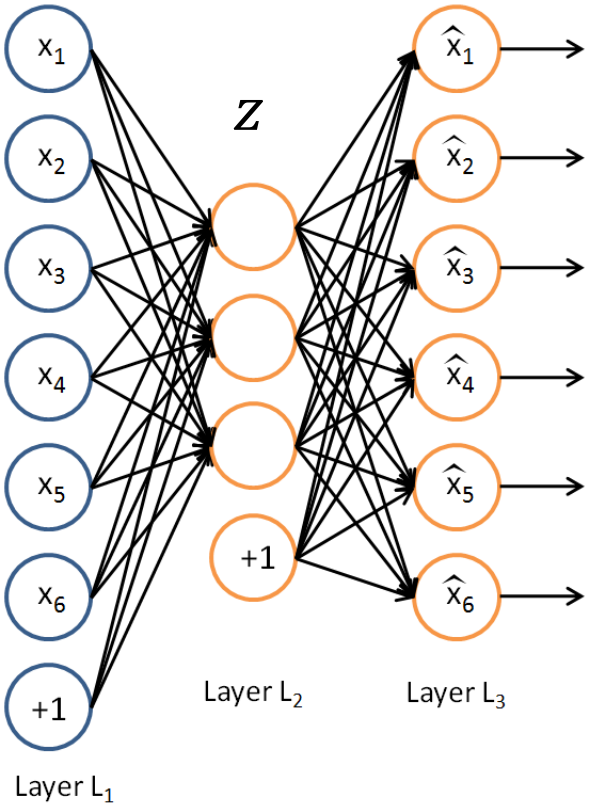
- Encoder: compresses the input into a latent-space of usually smaller dimensionality, ***z*** = *f (x)*
- Decoder: reconstructs the input from the latent space, $\hat{x}$ = g(***z***), with $\hat{x}$ as close to *x* as possible



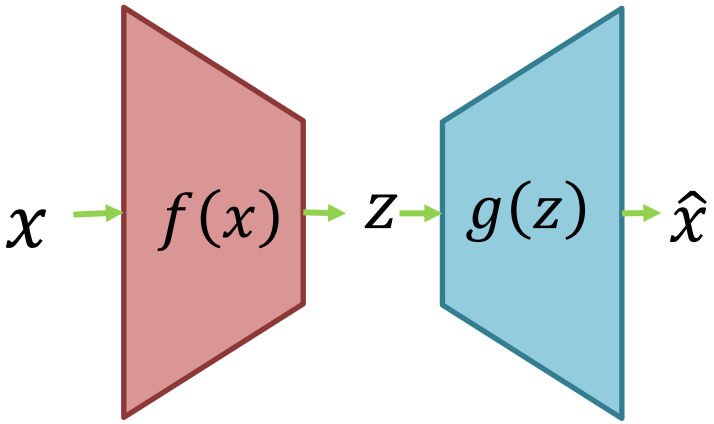
https://towardsdatascience.com/deep-inside-autoencoders-7e41f319999f

Traditional autoencoder

- Unlike **PCA** now we can use activation functions to achieve non-linearity
- It has been shown that an AE without activation functions achieves **PCA** capacity



Simple idea



Given data x (no labels) we would like to learn functions f (**encoder**) and g (**decoder**) where:

$$f(x) = s(wx + b) = z$$

and

$$g(z) = s(w'z + b') = \hat{x}$$

such that: $h(x) = g\big(f(x)\big) = \hat{x}$

where h is an **approximation** of the identity function

z is some **latent** representation or **code** and s is a non-linear function (such as the sigmoid)

$\hat{x}$ is x's reconstruction

Training the autoencoder

Using **Gradient Descent** we can simply train the model as any other fully connected NN using the *traditional squared error* loss function

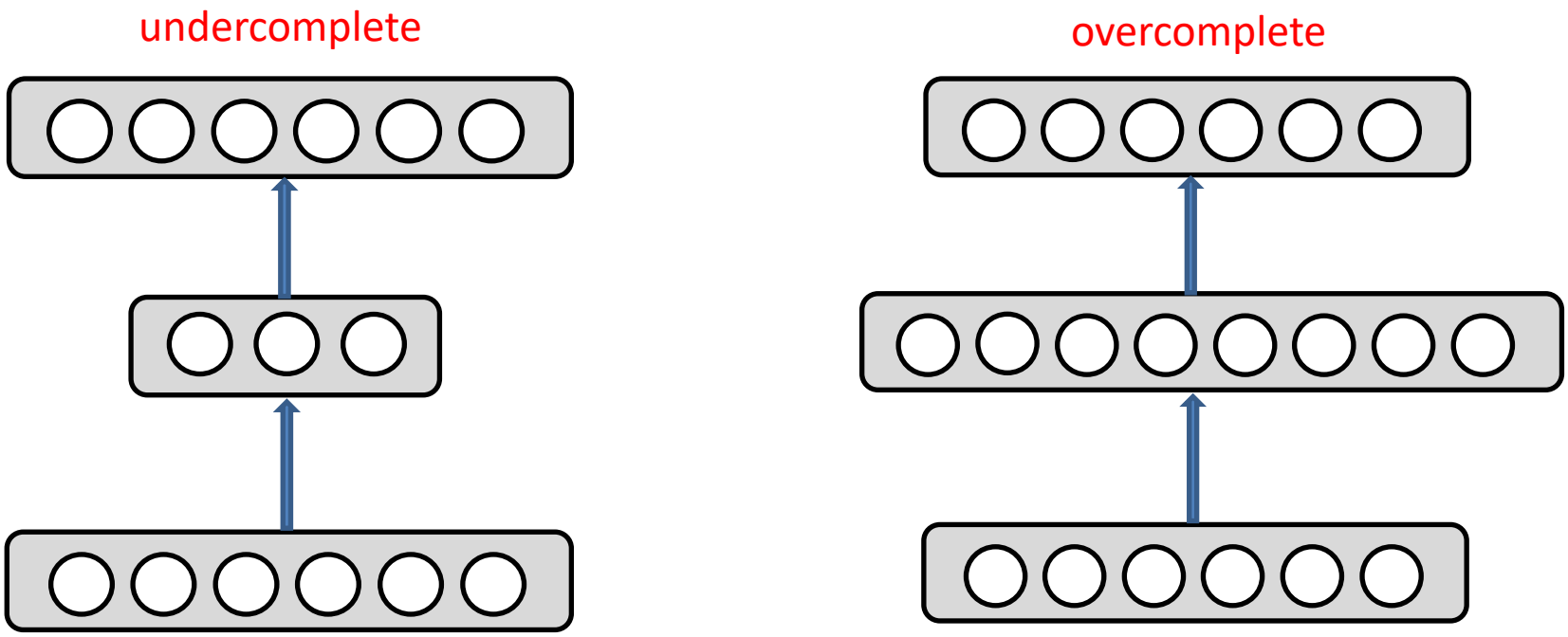
$$L(x, \hat{x}) = \|x - \hat{x}\|^2$$

If our input is interpreted as bit vectors or vectors of bit probabilities, then *cross entropy* can be used

$$\boldsymbol{CE\ Loss} = -\frac{1}{n}\sum_{i=1}^{n}(x_i \cdot \log(\hat{x}_i))$$

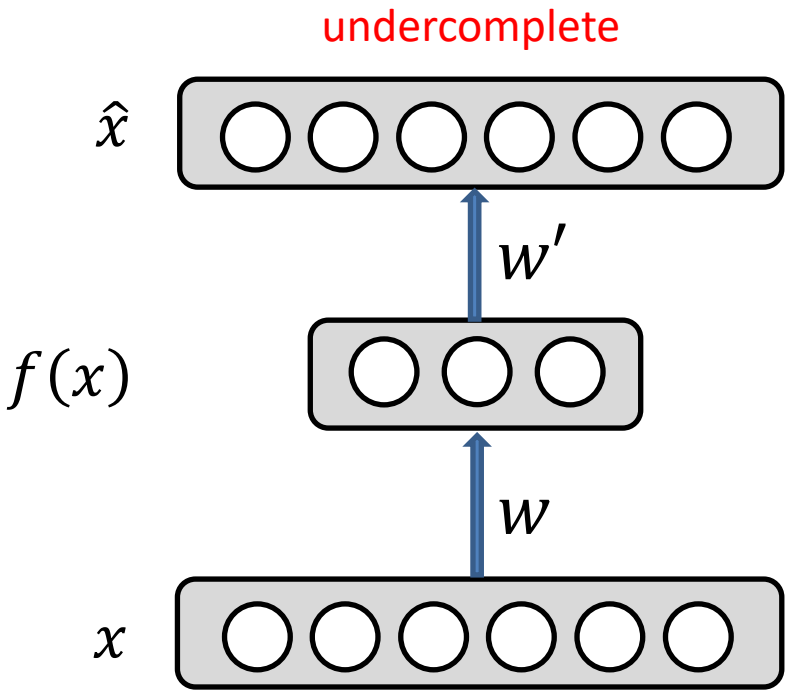
Two types of autoencoders

We distinguish between **two types** of AE structures:



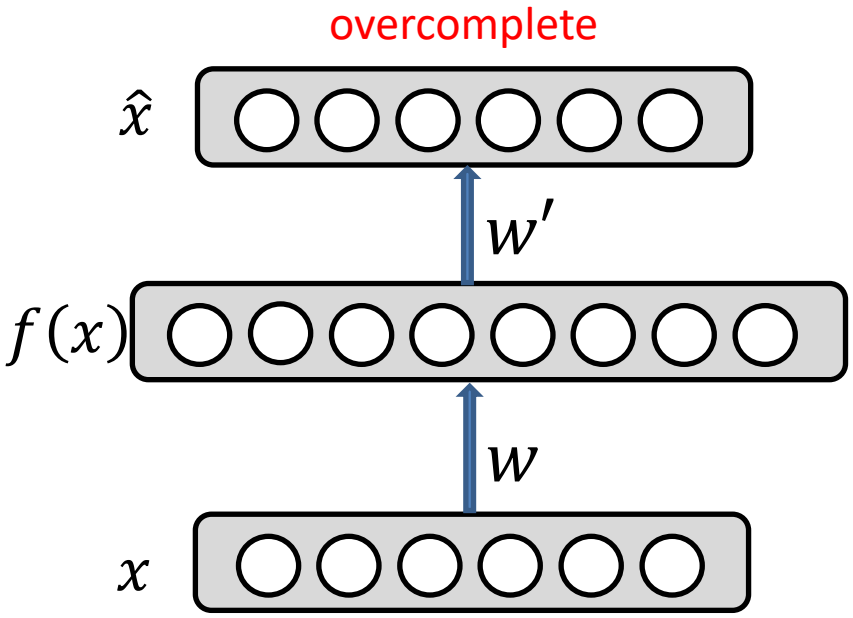
Undercomplete autoencoders

- Hidden layer is **undercomplete**, i.e., smaller than the input layer
  - compresses the input
  - hidden nodes capture good features of the training set distribution

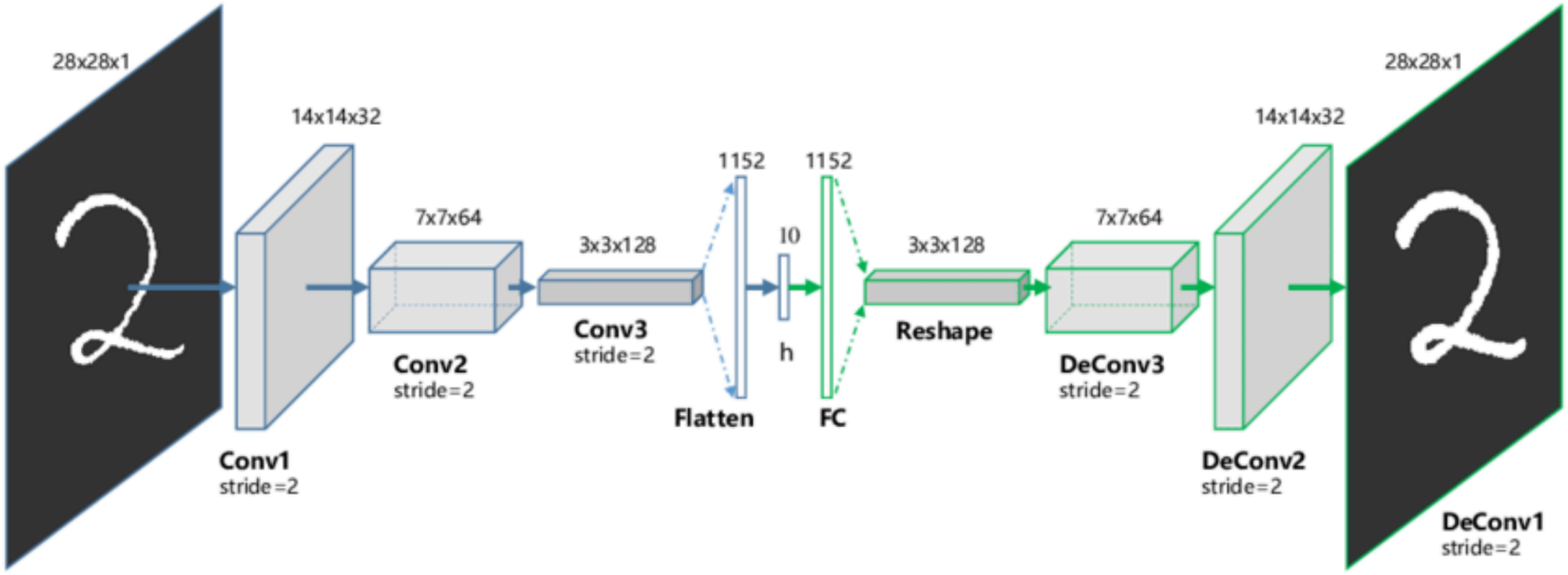


Overcomplete autoencoders

- Hidden layer is **overcomplete**, i.e., greater than the input layer
  - no compression in hidden layer
  - **no guarantee** that the hidden units will extract a meaningful structure
  - a latent space of **higher dimensionality** models a more complex distribution



Training an autoencoder

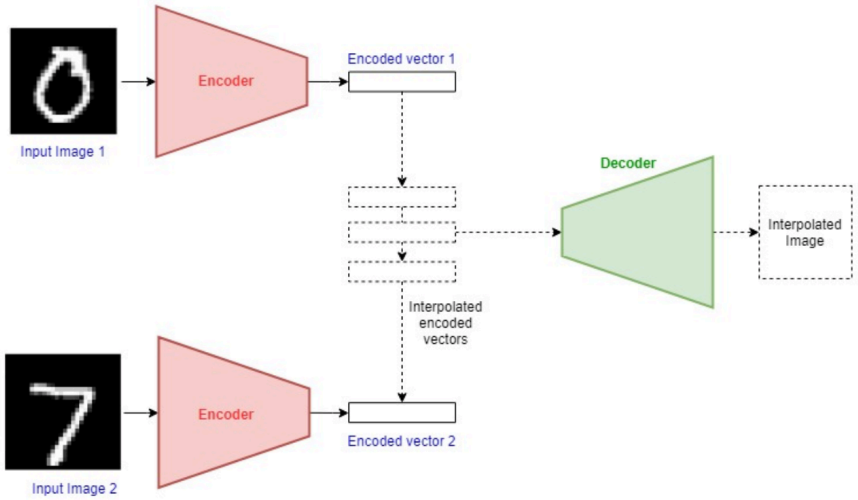


The auto-encoder model is trained to generate the same image it is fed with

Once the auto-encoder is trained as a whole, we use the encoder and decoder networks independently

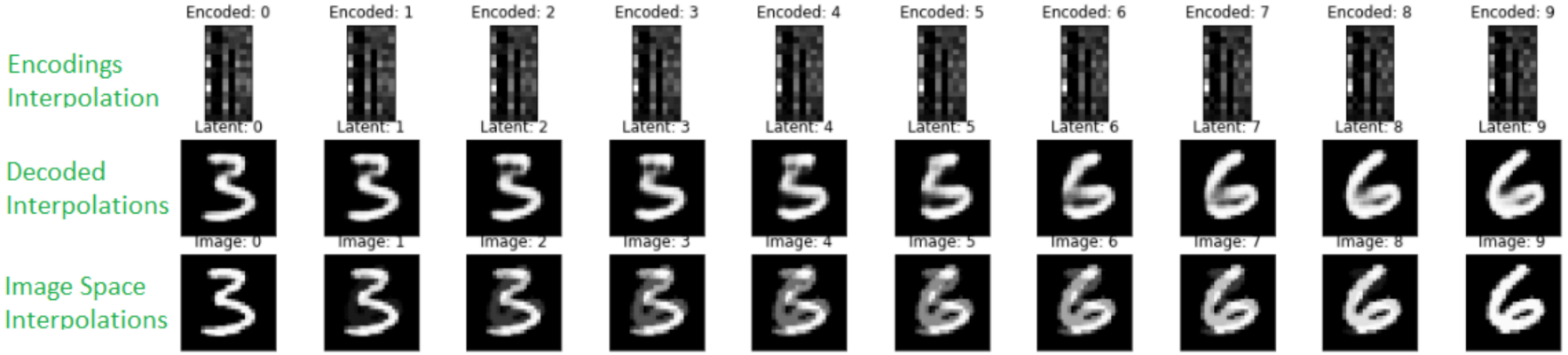
Image fading

- Take two random images (a zero and a seven) and generate the encodings by passing them through the encoder
- The encodings are flat vectors of length 128 each

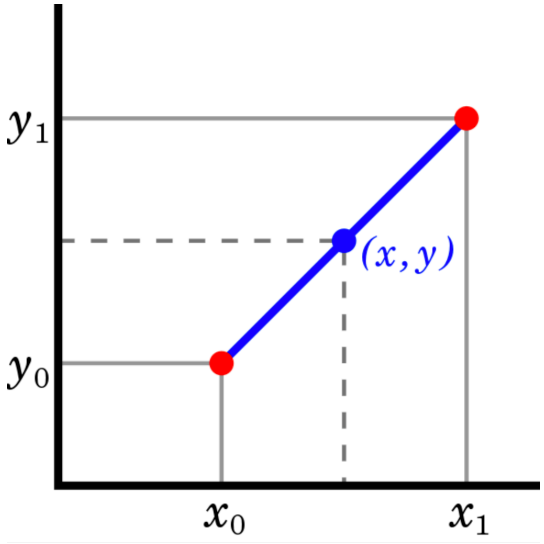


Generate interpolations (e.g., 10) between the encodings

Generate images for each of these interpolations by passing them through the decoder

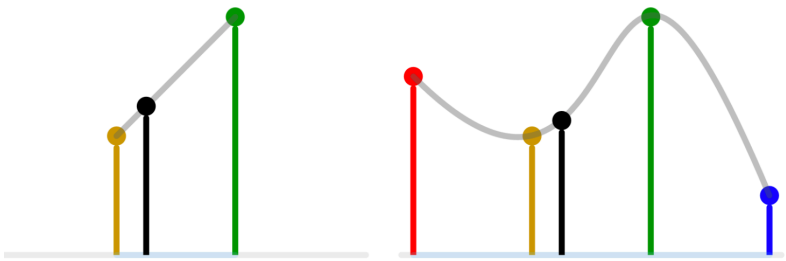


Data interpolation

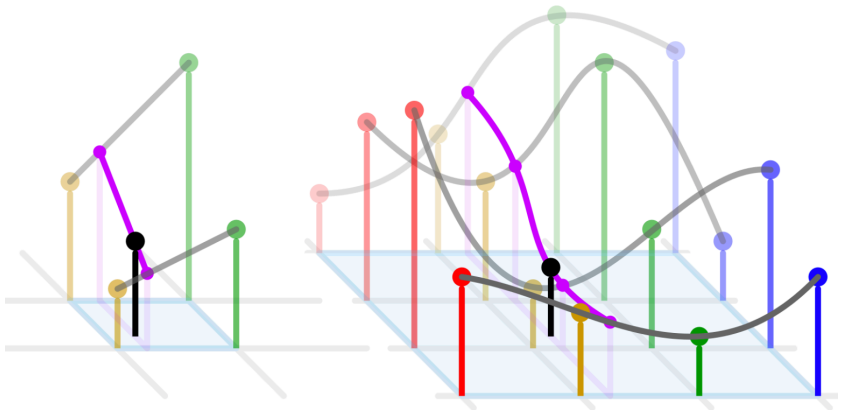


$$\frac{y - y_0}{x - x_0} = \frac{y_1 - y_0}{x_1 - x_0},$$

$$\begin{aligned} y &= y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0} \\ &= \frac{y_0(x_1 - x_0)}{x_1 - x_0} + \frac{y_1(x - x_0) - y_0(x - x_0)}{x_1 - x_0} \\ &= \frac{y_1 x - y_1 x_0 - y_0 x + y_0 x_0 + y_0 x_1 - y_0 x_0}{x_1 - x_0} \\ &= \frac{y_0(x_1 - x) + y_1(x - x_0)}{x_1 - x_0}, \end{aligned}$$



Linear Cubic



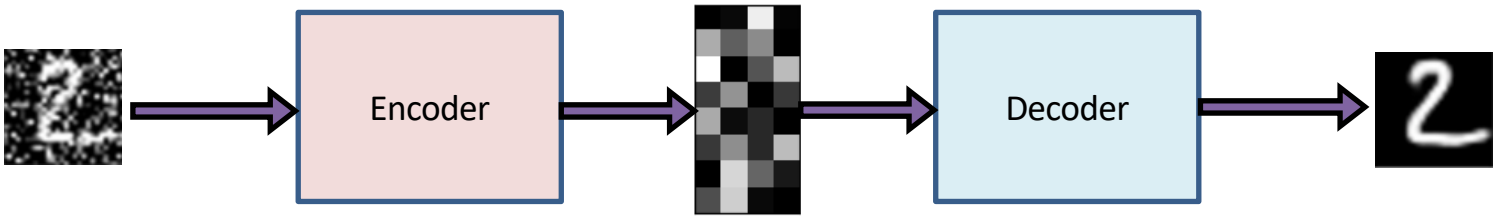
Bilinear Bicubic

Denoising autoencoders

Intuition:

- Encode the input and do **not** mimic the identity function
- Undo the effect of the ***corruption*** process applied to the input

A more robust model



Noisy Input

Latent space representation

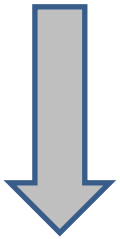
Denoised Input

Denoising autoencoders

Instead of trying to *mimic* the identity function by minimizing:

$$\boldsymbol{L}\left(\boldsymbol{x}, \boldsymbol{g}\left(\boldsymbol{f}(\boldsymbol{x})\right)\right)$$

where $\boldsymbol{L}$ is some loss function



A **DAE** instead minimizes:

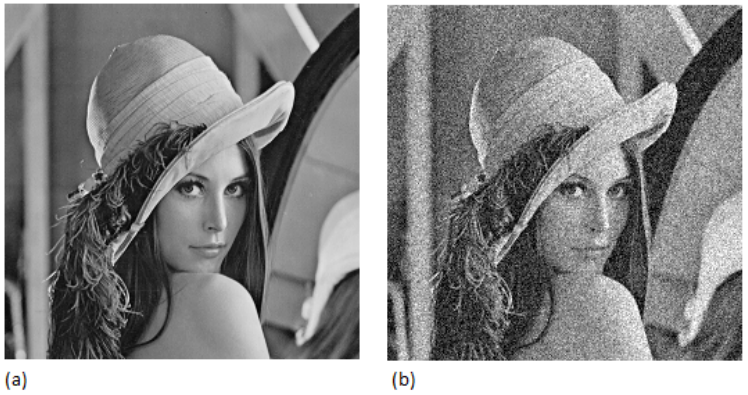
$$\boldsymbol{L}\left(\boldsymbol{x}, \boldsymbol{g}\left(\boldsymbol{f}(\tilde{\boldsymbol{x}})\right)\right)$$

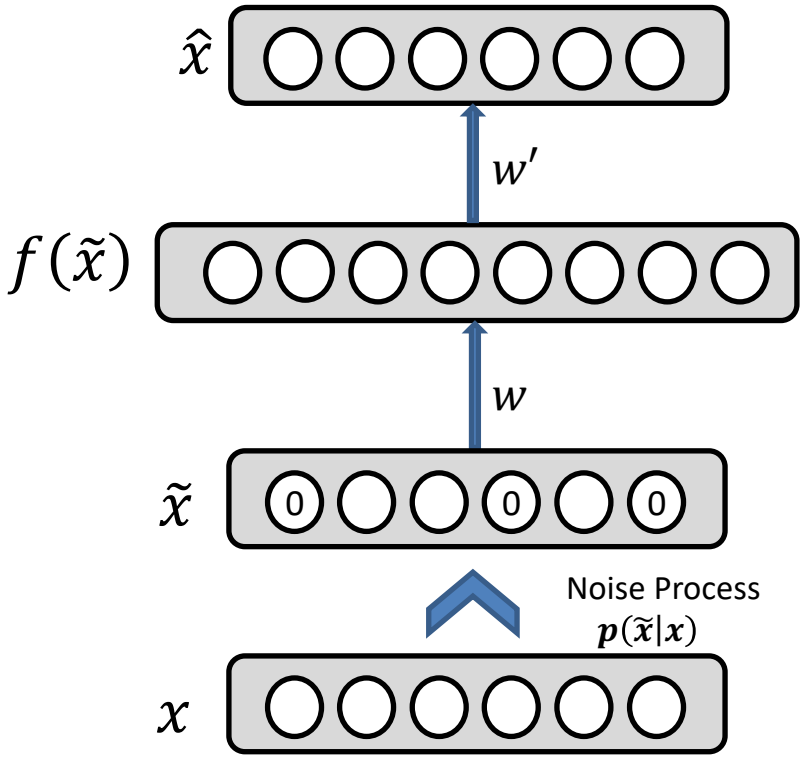
where $\tilde{\boldsymbol{x}}$ is a copy of $\boldsymbol{x}$ that has been corrupted by some form of noise

Denoising autoencoders

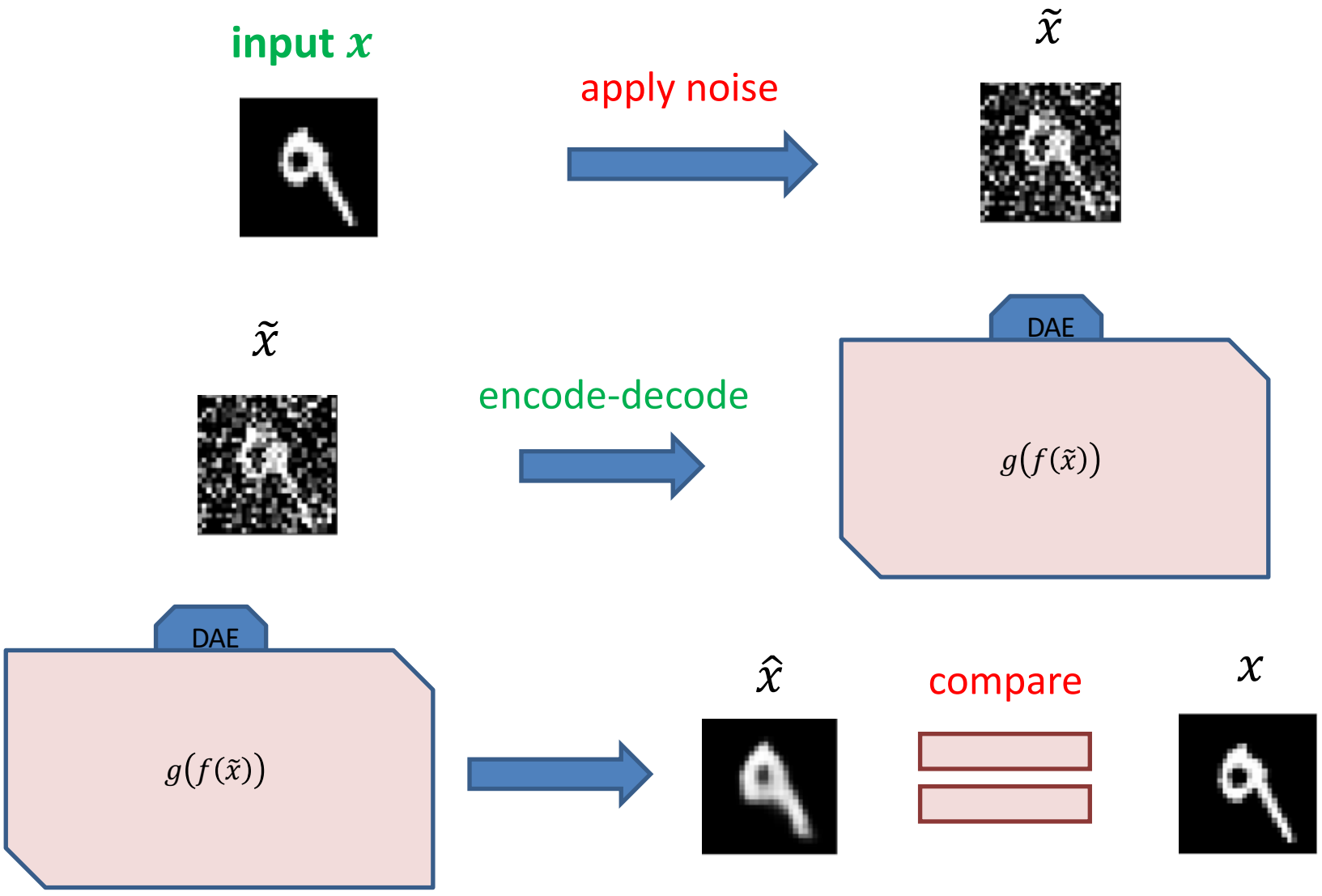
A robust representation against noise:

- random assignment of subset of inputs to 0, with probability v
- gaussian additive noise





Denoising autoencoders - process



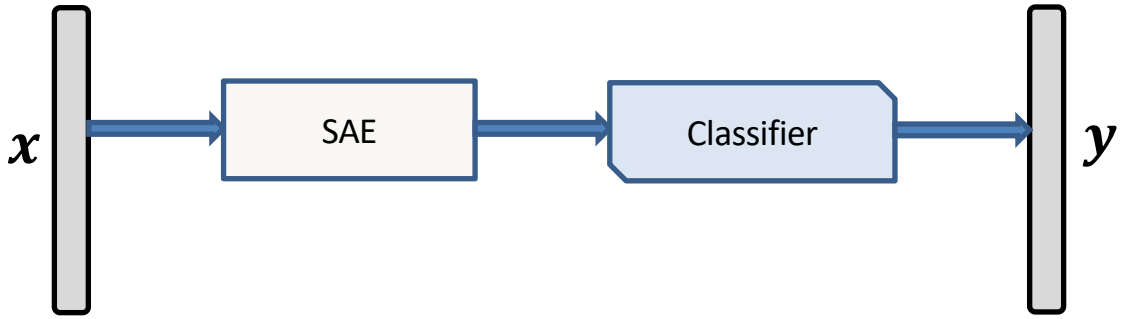
Stacked autoencoders

Motivation:

- We want to harness the feature extraction quality of an AE
- **For example:** we can build a deep classifier, whose input is the output of a SAE
- **The benefit:** our deep model's ***W*** are not randomly initialized but are rather "**smartly**" selected

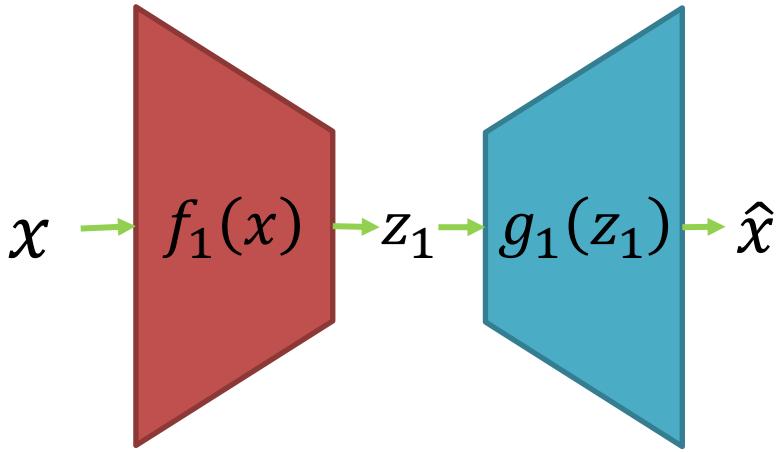
Building a SAE consists of two phases:

- Train each AE layer one after the other
- Connect any classifier (SVM / FC NN layer etc.)



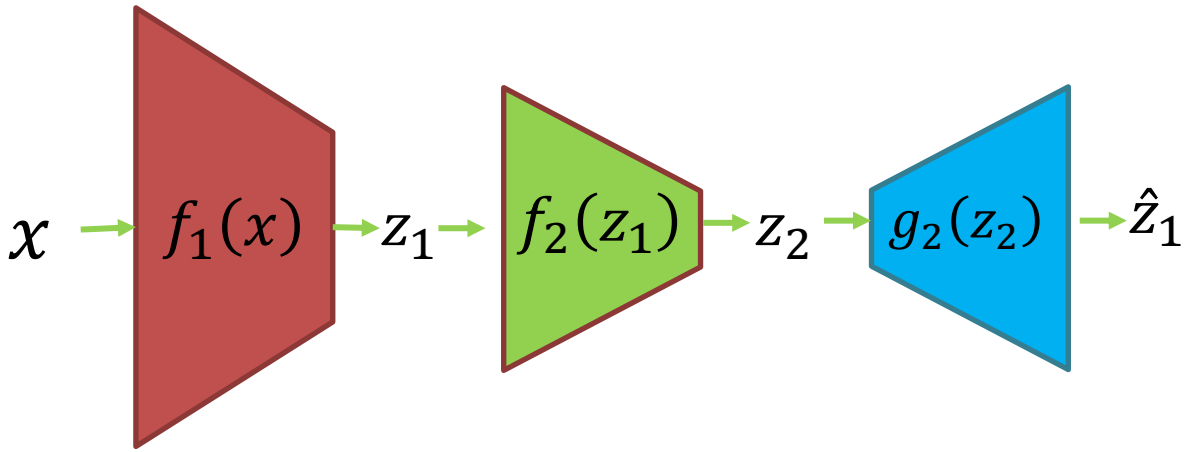
Stacked autoencoders

first layer training (AE 1)



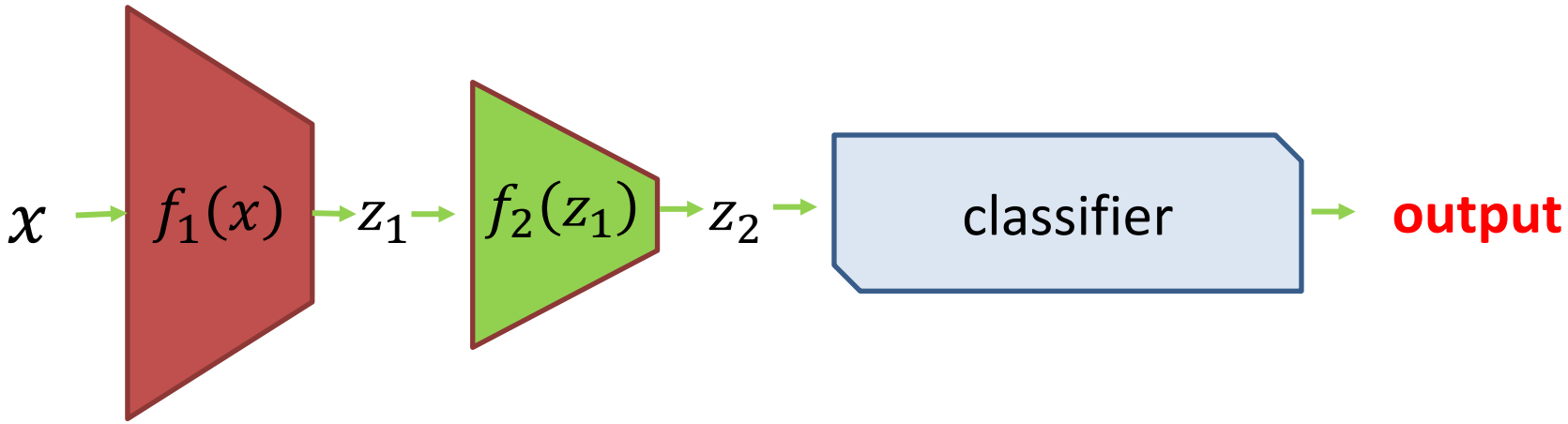
Stacked autoencoders

second layer training (AE 2)



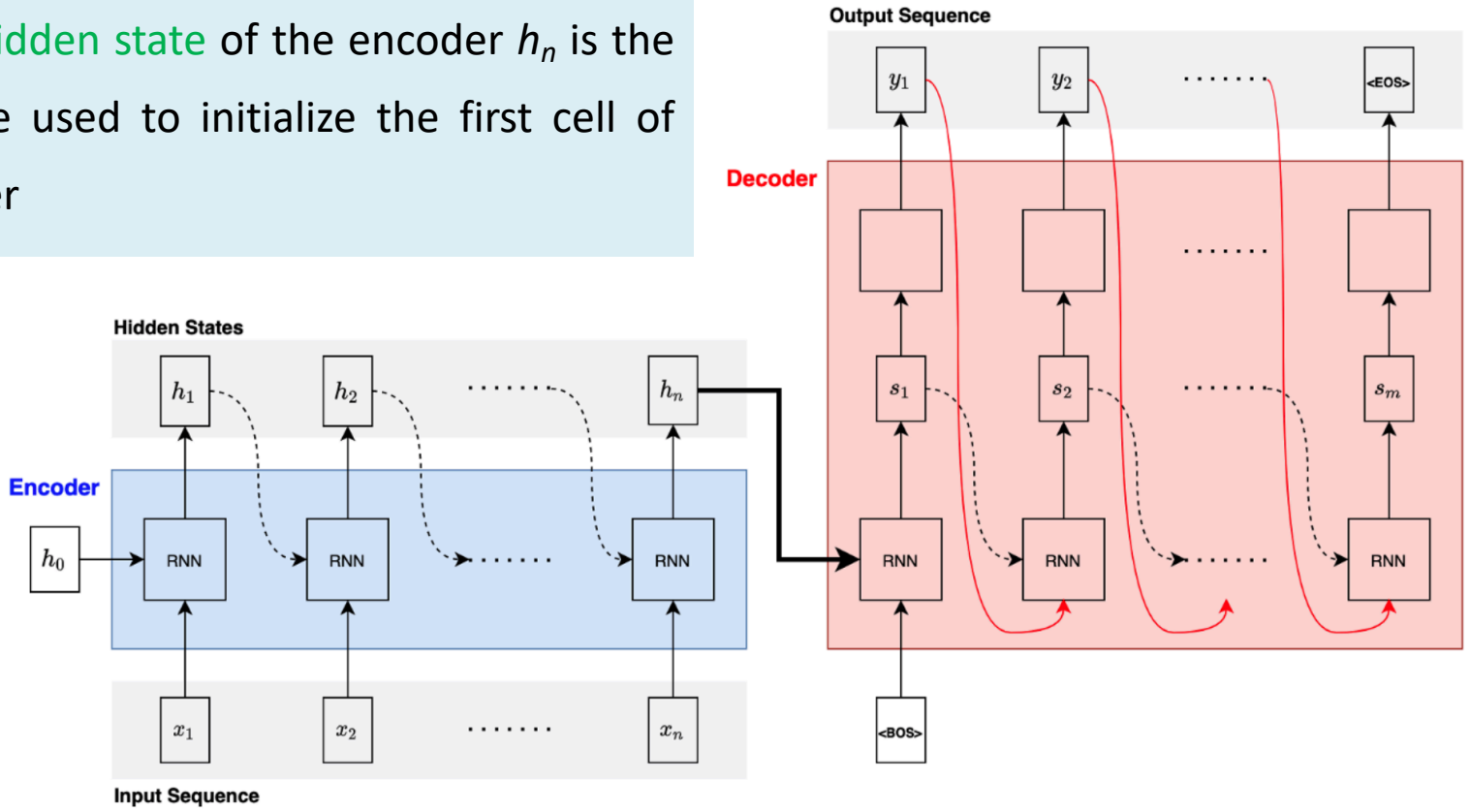
Stacked autoencoders

add any classifier

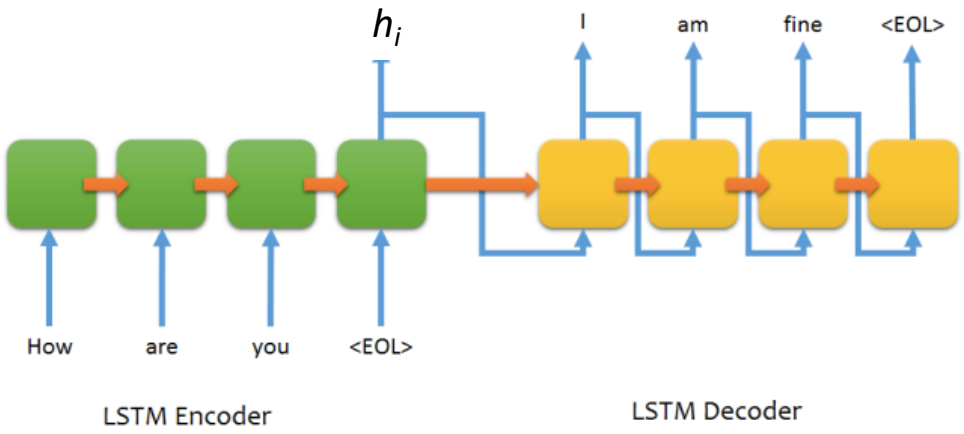


Autoencoder with RNNs

- **LSTMs** for the encoder and decoder networks
- Can use bi-LSTMs for the encoder
- The final hidden state of the encoder h_n is the latent code used to initialize the first cell of the decoder



Autoencoder with RNNs – question answering



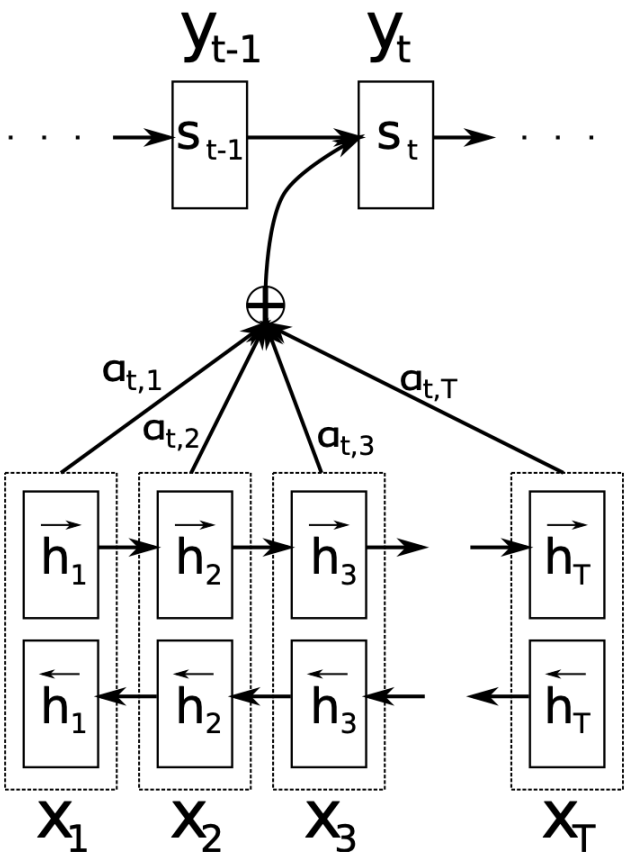
- Reconstruction loss (cross-entropy)
- Output h_i of every decoder cell *i* is sent to *softmax*
- Predict probabilities of words in a given word vocabulary
- **At training time:** the decoder receives the **ground-truth** previous word as input at each time step
- **At inference time:** the decoder receives the **predicted** word at the previous time step

Limitation of autoencoders

- **Language translation:** encoder + decoder learn the translation task together
- **Encoder:** reads and encodes a source sentence into a fixed-length vector
- **Decoder:** outputs a translation from the encoded vector
- A neural network needs to be able to compress all the necessary information of a source sentence into a fixed-length vector
- This may make it difficult for the neural network to cope with long sentences, especially those that are longer than the sentences in the training corpus [Bahdanau 2015]
- Solution?
  - attention…

Attention mechanism [Bahdanau 2015]

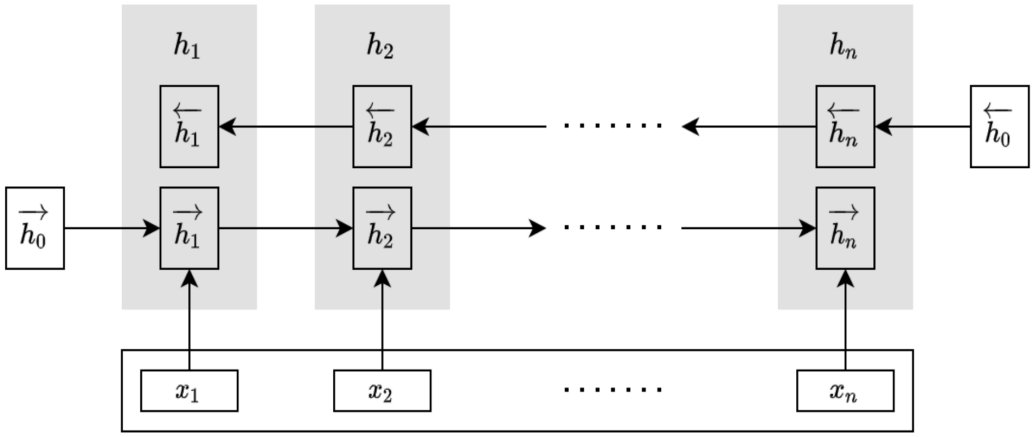
- **For each word prediction** the model searches for a set of positions in the source sentence, where the most relevant information is concentrated
- The model predicts a target word based on the context vectors associated with these source positions and all previously generated target words



- This AE does not encode a whole input sentence into a single fixed-length vector
- It encodes the input sentence into a sequence of vectors and chooses a subset of these vectors adaptively, while decoding the translation

Attention mechanism [Bahdanau 2015]

- **Decoder:** implements an attention mechanism by "soft"-searching through **annotations** generated by the encoder
- **Encoder:** uses a BiRNN (Bidirectional RNN) to capture richer context from the input sentence by producing **annotations** that are concatenations of the forward and backward hidden states at each step



- Each annotation contains information from the whole input sentence thanks to the bidirectional RNN
- Each annotation has a strong focus on the parts surrounding it

Attention mechanism

- How does the decoder learn to pay attention to relevant annotations?
- It uses an alignment model with a trainable feed-forward network to score how relevant each annotation is for the decoder's hidden state:

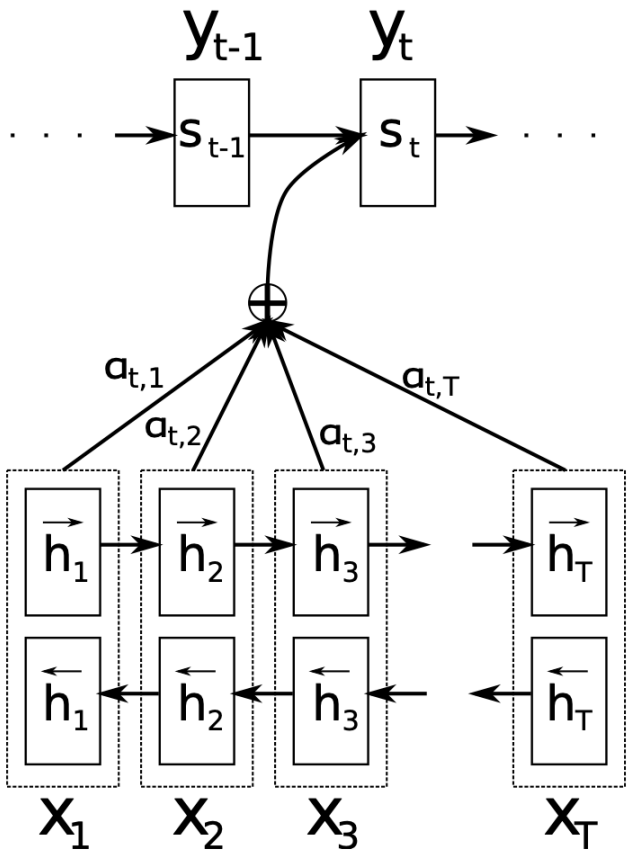
decoder encoder

$$e_{ij} = a(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j)$$

- It calculates the weights of each annotation using the alignment scores:

(**note:** *n* = *T*)

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})}$$



- The decoder receives as input at *time point* ***i*** a context vector that is a weighted sum of the annotations

$$\boldsymbol{c}_i = \sum_{j=1}^{n} \alpha_{ij} \boldsymbol{h}_j$$

Attention mechanism

Decoder at step *i:*

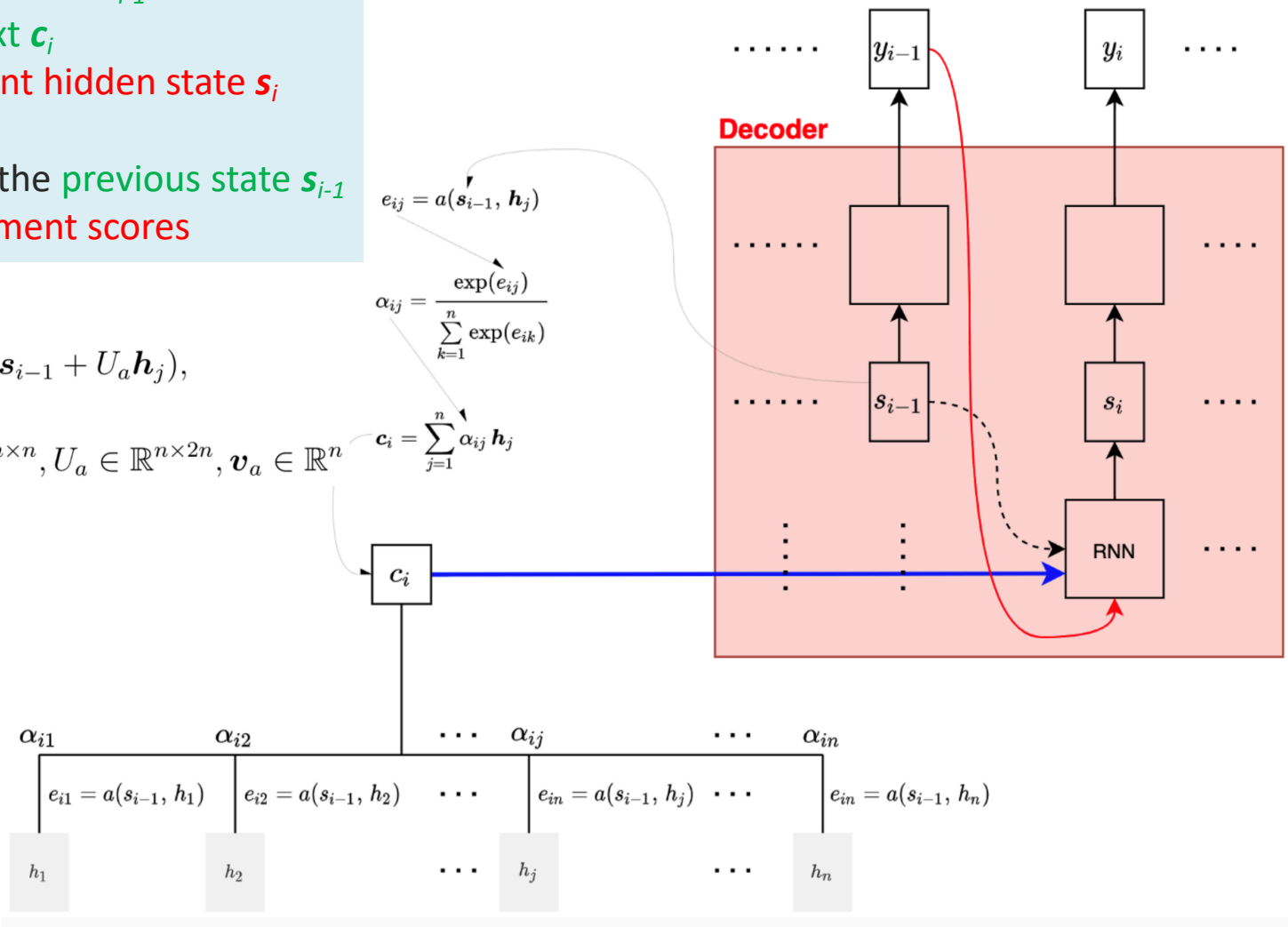
- takes the previous state s_{i-1}
- the current context c_i

to generate the current hidden state s_i

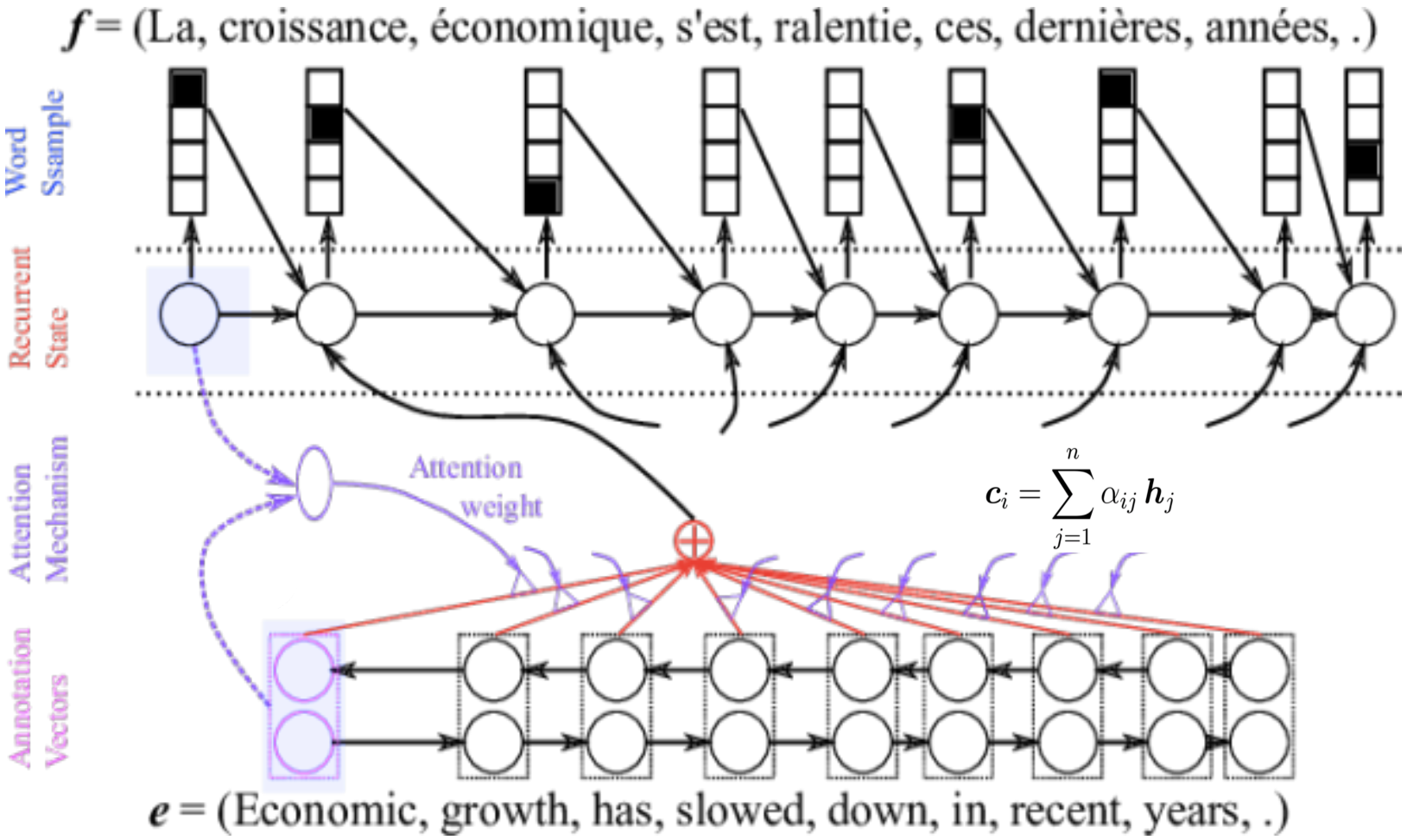
which is why we use the previous state s_{i-1} to calculate the alignment scores

$$a(\boldsymbol{s}_{i-1}, \boldsymbol{h}_j) = \boldsymbol{v}_a^\top \tanh(W_a \boldsymbol{s}_{i-1} + U_a \boldsymbol{h}_j),$$

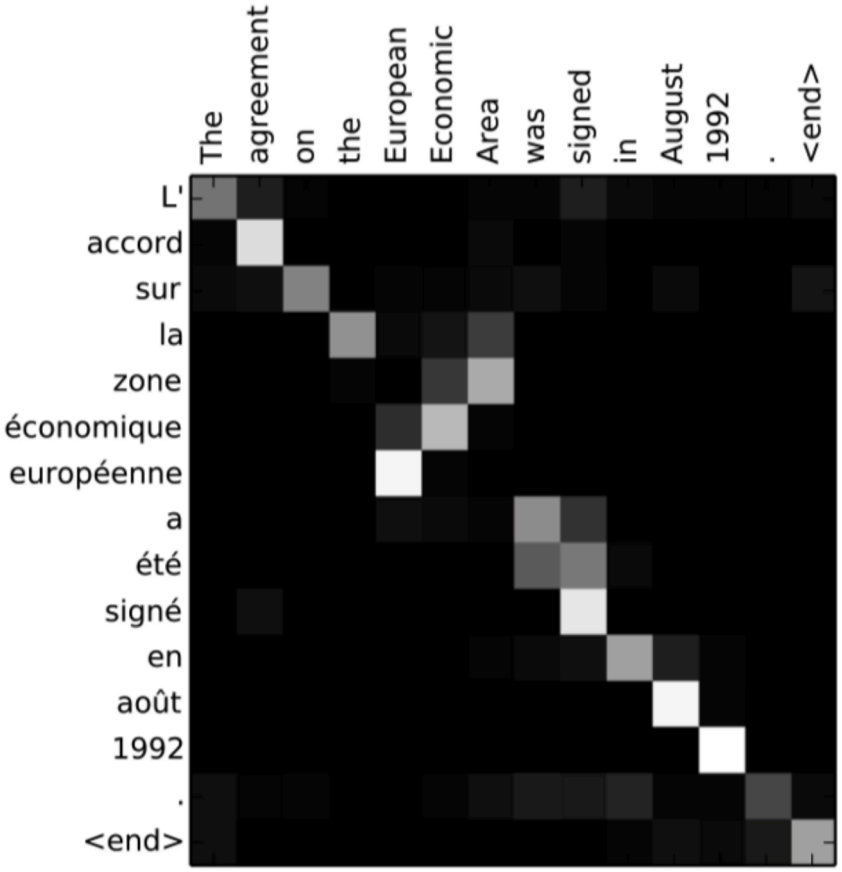
$$\text{where } W_a \in \mathbb{R}^{n \times n}, U_a \in \mathbb{R}^{n \times 2n}, \boldsymbol{v}_a \in \mathbb{R}^n$$

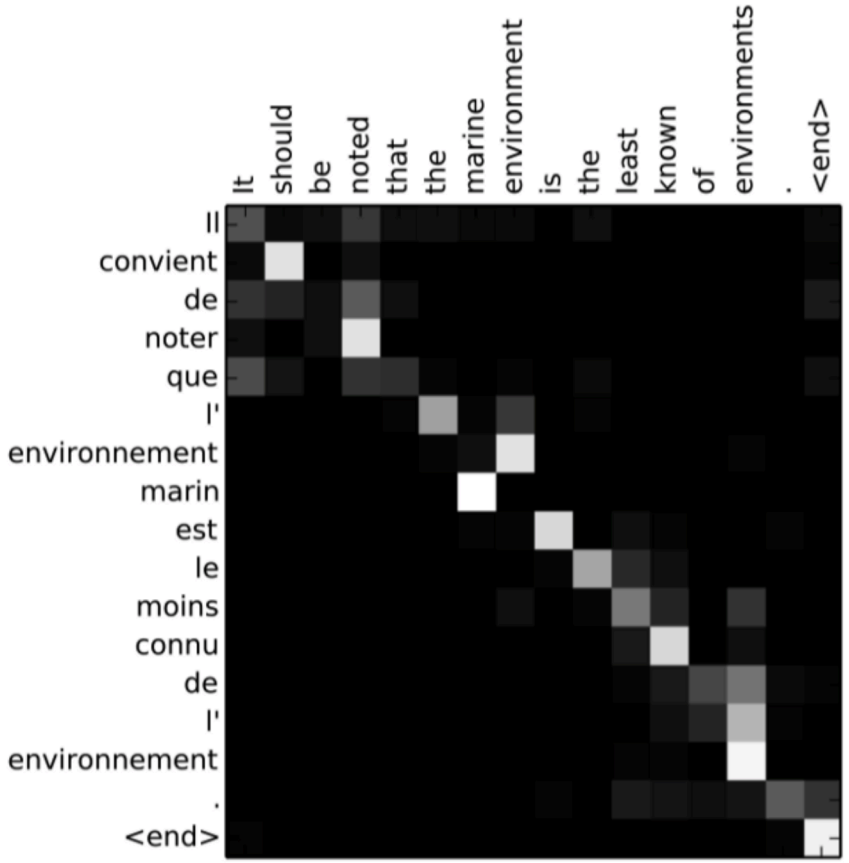


Encoder-Decoder machine translation



Encoder-Decoder machine translation

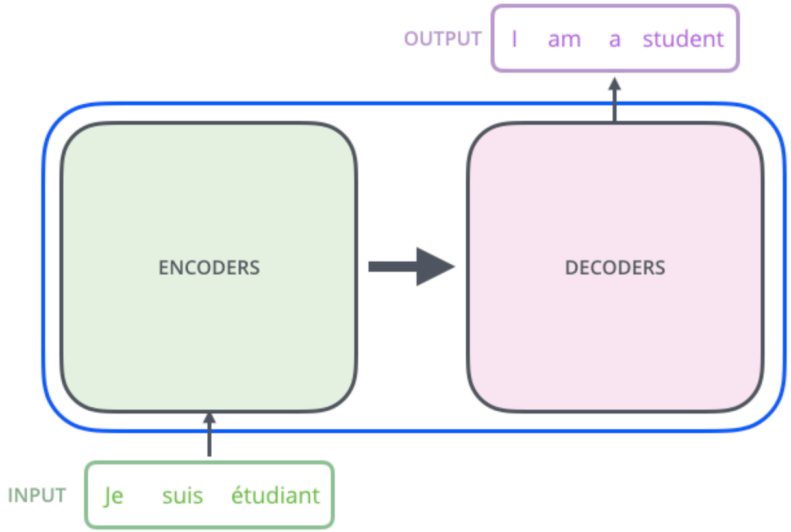




Two examples of sample alignments and their alignment scores α_{ij}

Transformers

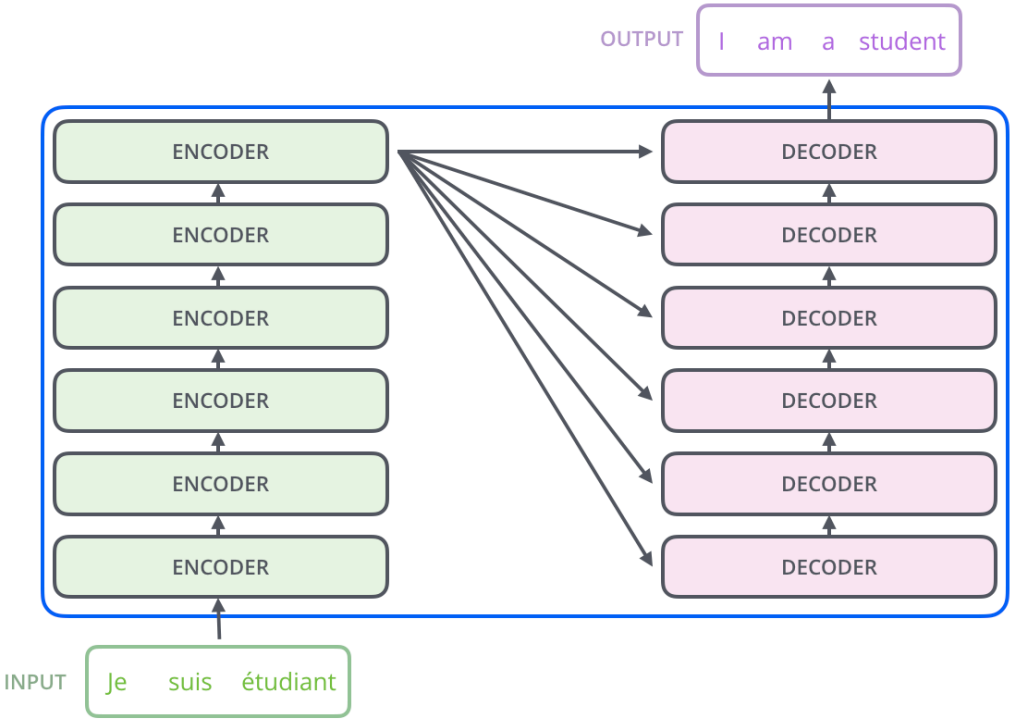
- The transformer is the first model relying entirely on self-attention to compute the representations of its input and output without using any RNN or CNN units



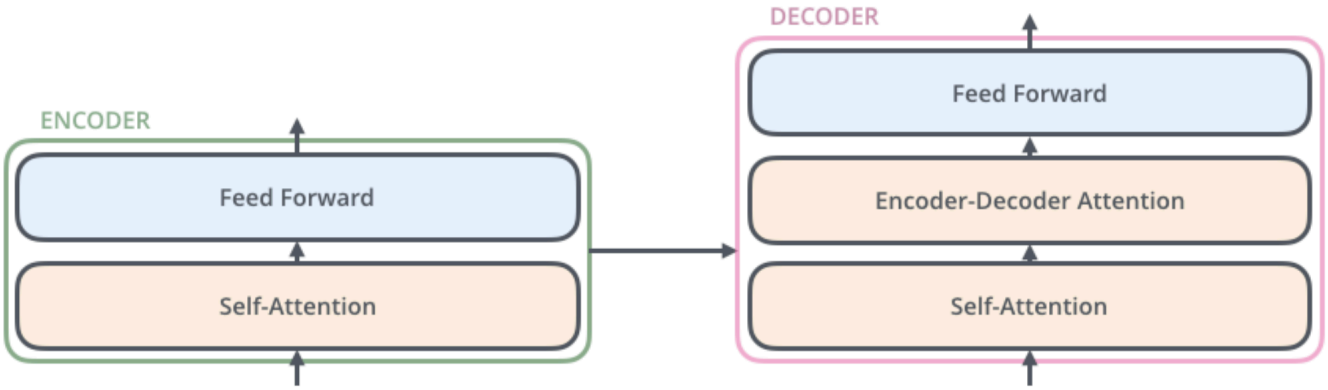
Transformers

Encoding component: a stack of encoders

Decoding component: a stack of decoders of the same number

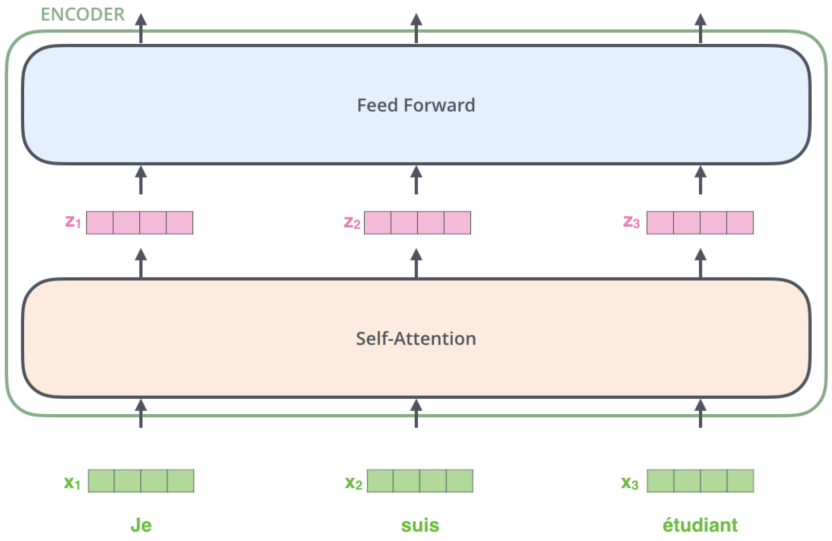


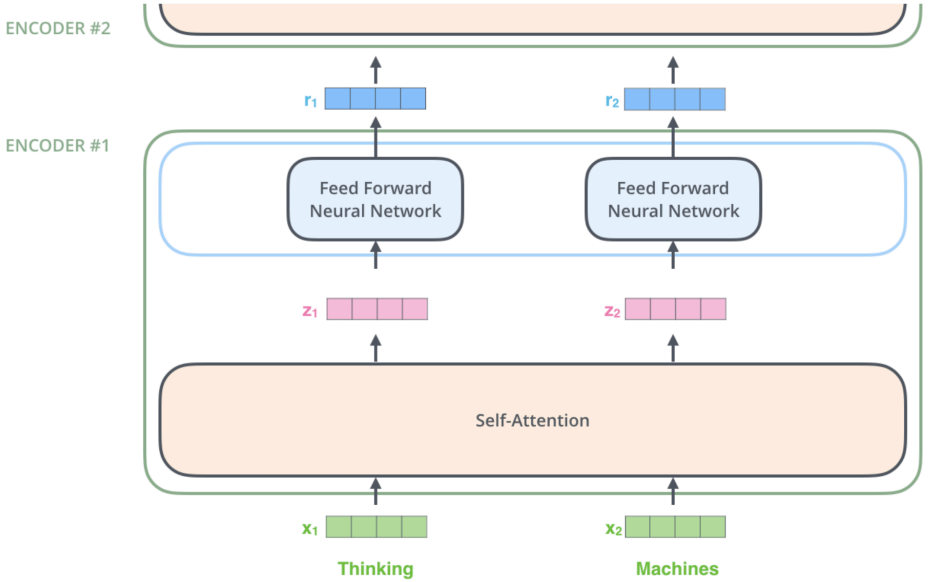
Transformers – Encoder/Decoder



- **Encoders:** all are identical in structure, but they do not share the same weights
  - inputs flow through a self-attention layer: a layer that helps the encoder look at other words in the input sentence as it encodes a specific word
  - outputs of the self-attention layer are fed to a feed-forward neural network
  - the exact same feed-forward network is independently applied to each position
- **Decoder:** has both those layers, but between them there is an attention layer that helps the decoder focus on relevant parts of the input sentence

Encoding





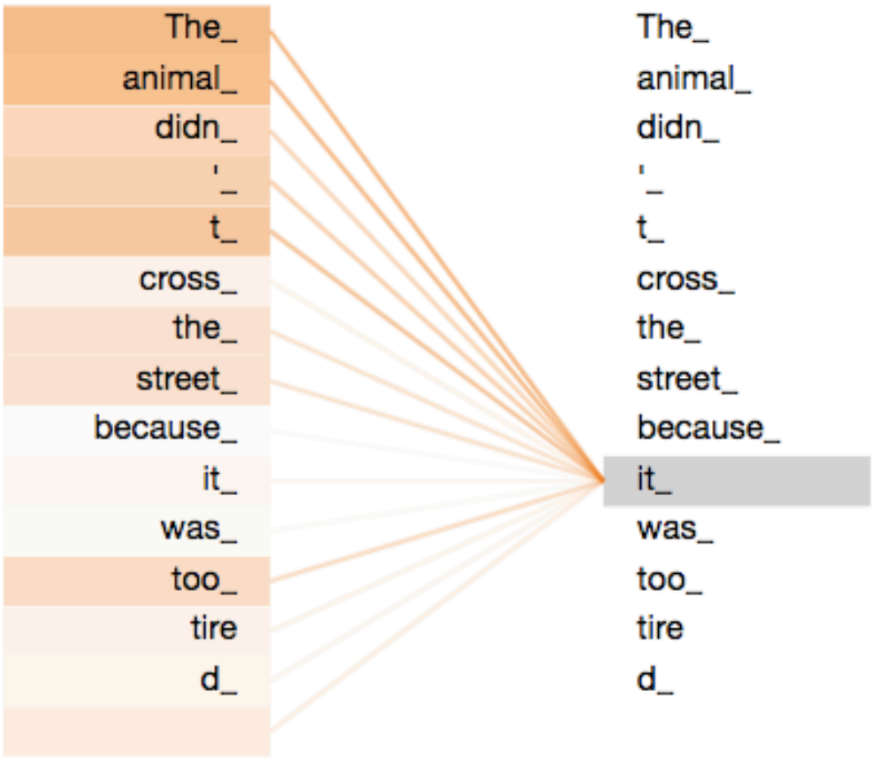
- Each encoder receives a list of vectors, each of 512 bits (word embeddings)
- Each embedding flows through each of the two layers of the encoder
- These paths have dependncies when in the self-attention layer but not when through the feed-forward NN, which is the same for all input vectors
- Hence, the latter can be executed in parallel

Self-attention

- Consider the sentence:

  ”The animal didn't cross the street because it was too tired”

- What does “it” refer to?
  - **simple answer** for humans but complex for an algorithm
  - self-attention solves this by associating “it” with “animal”



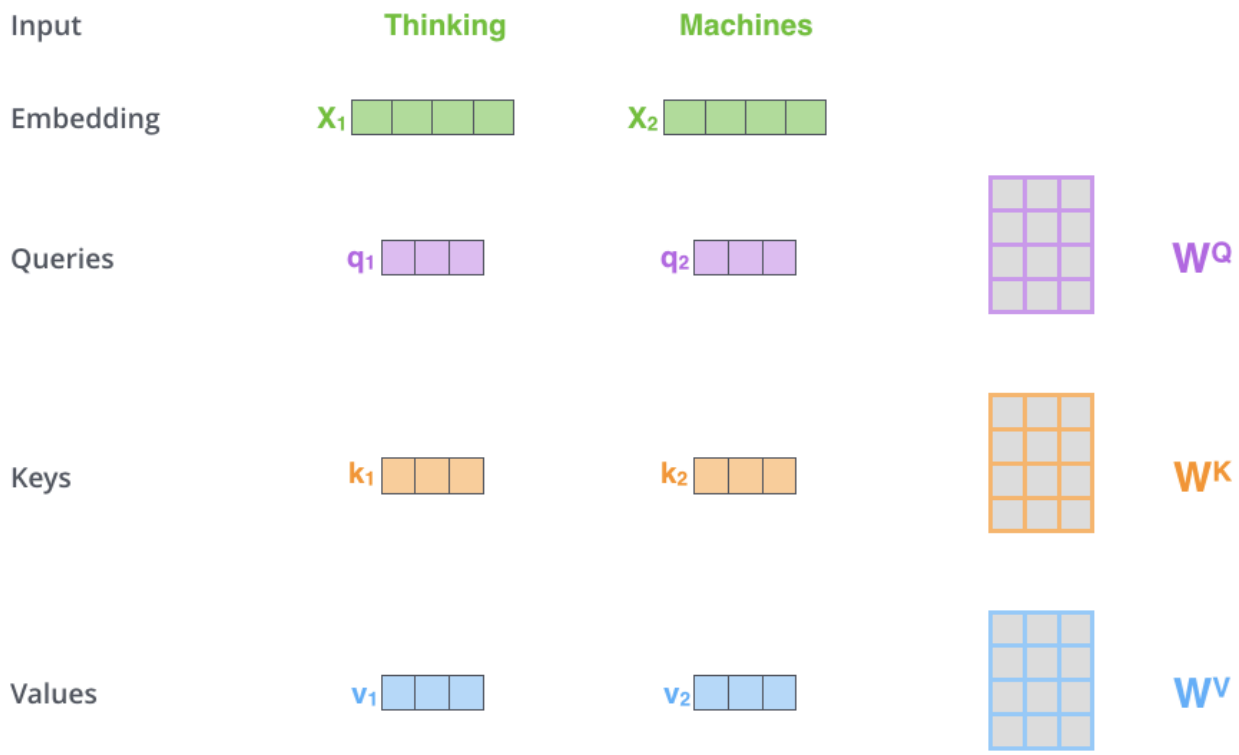
Step I: define three vectors

For each input embedding x_i, we create three vectors by multiplying x_i with three matrices (already trained)

Query vector: $q_i = x_i W^Q$

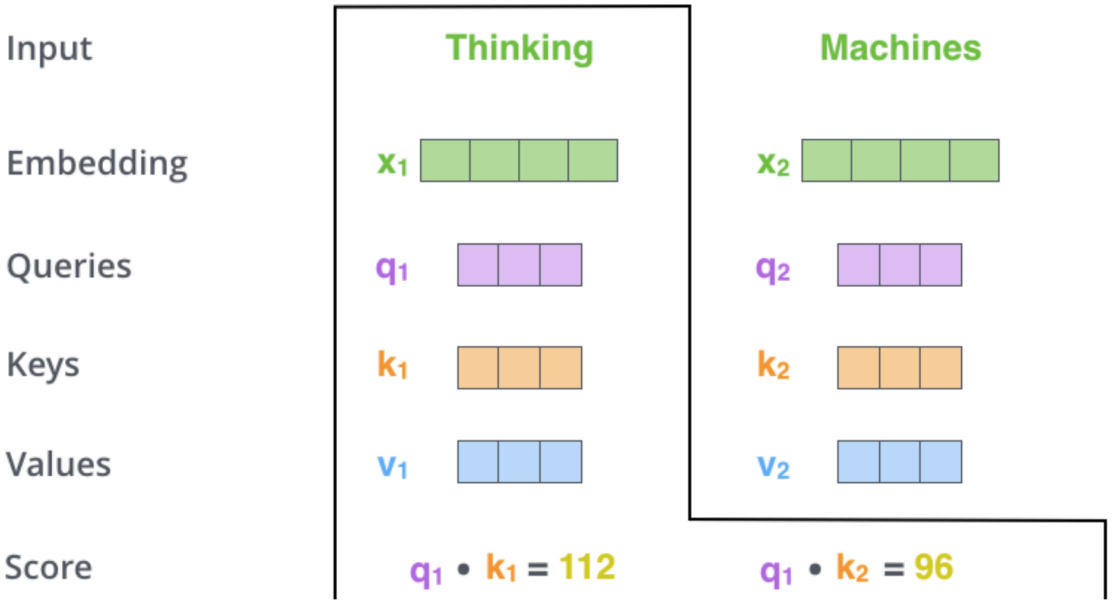
Key vector: $k_i = x_i W^K$

Value vector: $v_i = x_i W^V$



- The default dimensionality of these new vectors is 64 (smaller than the embedding which is 512)

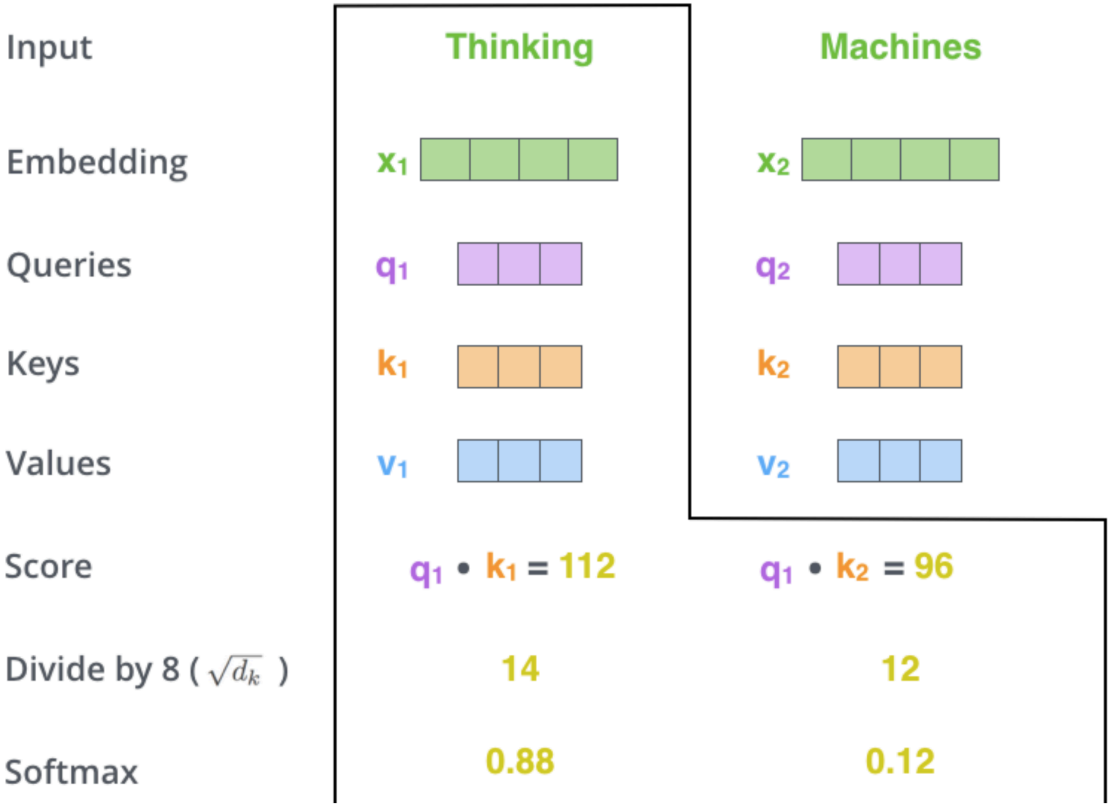
Step II: self-attention score

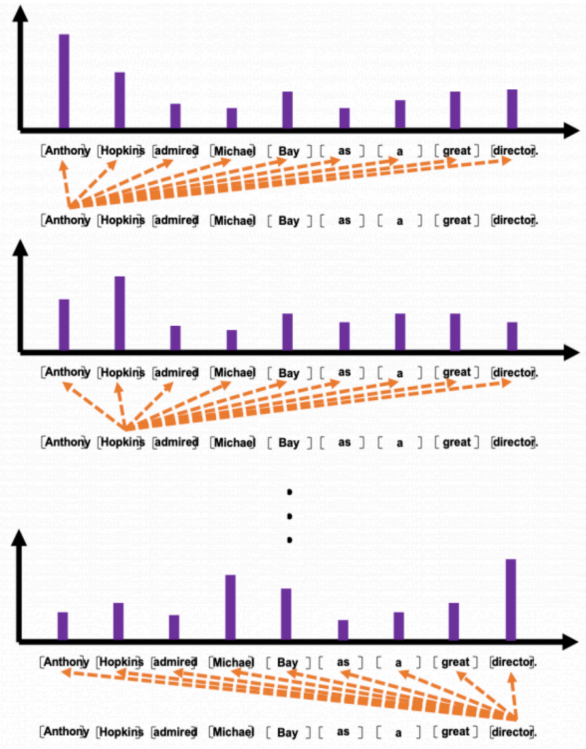


- **Self-attention score:** the dot product of the query vector with the key vector of the respective word
- If processing self-attention for the word in position #1, the first score is q1*k1, the second score q1*k2, etc…

- We score each word of the input sentence against each word (for example, Thinking)
- The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position

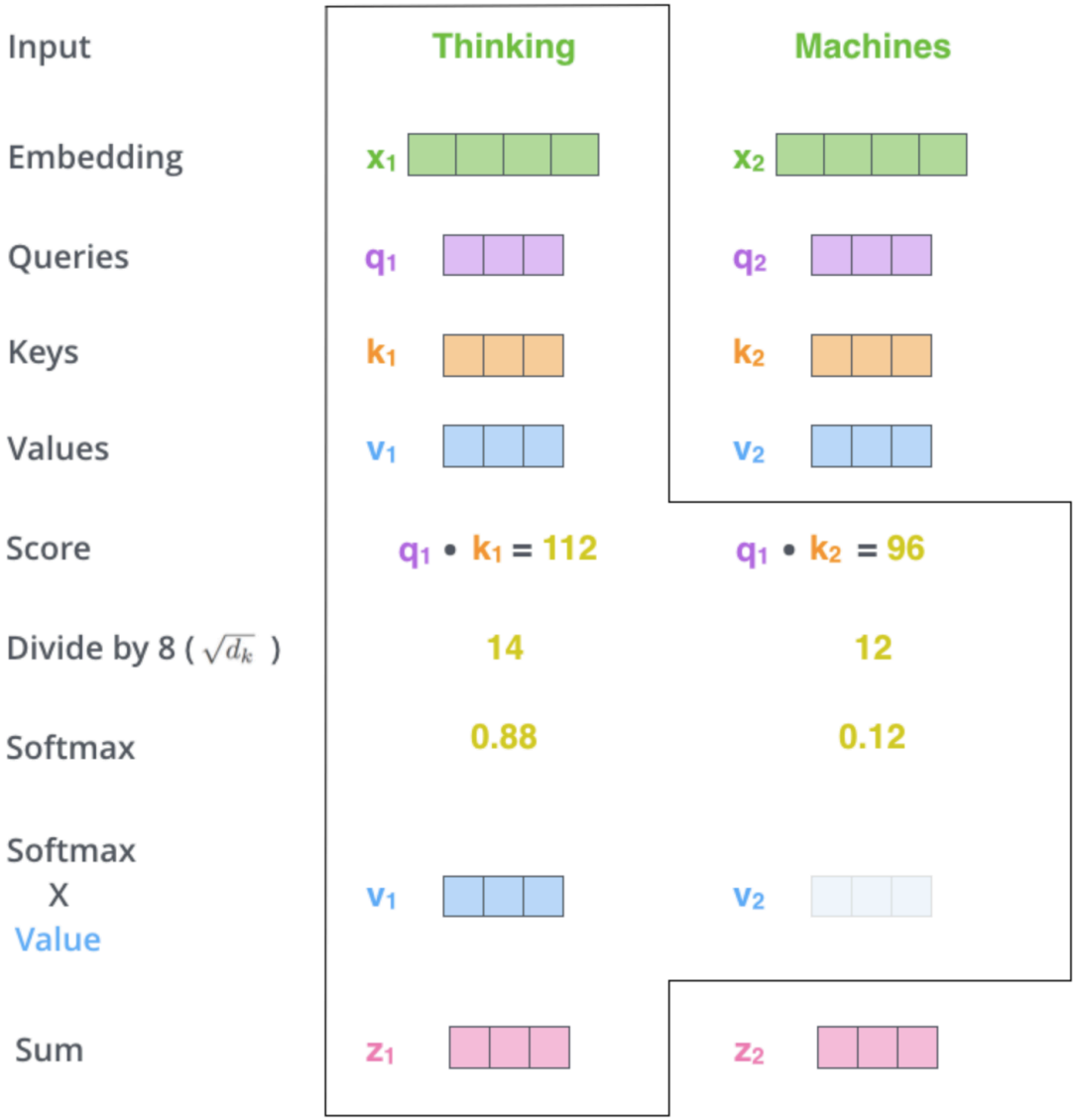
Steps III-IV: normalization





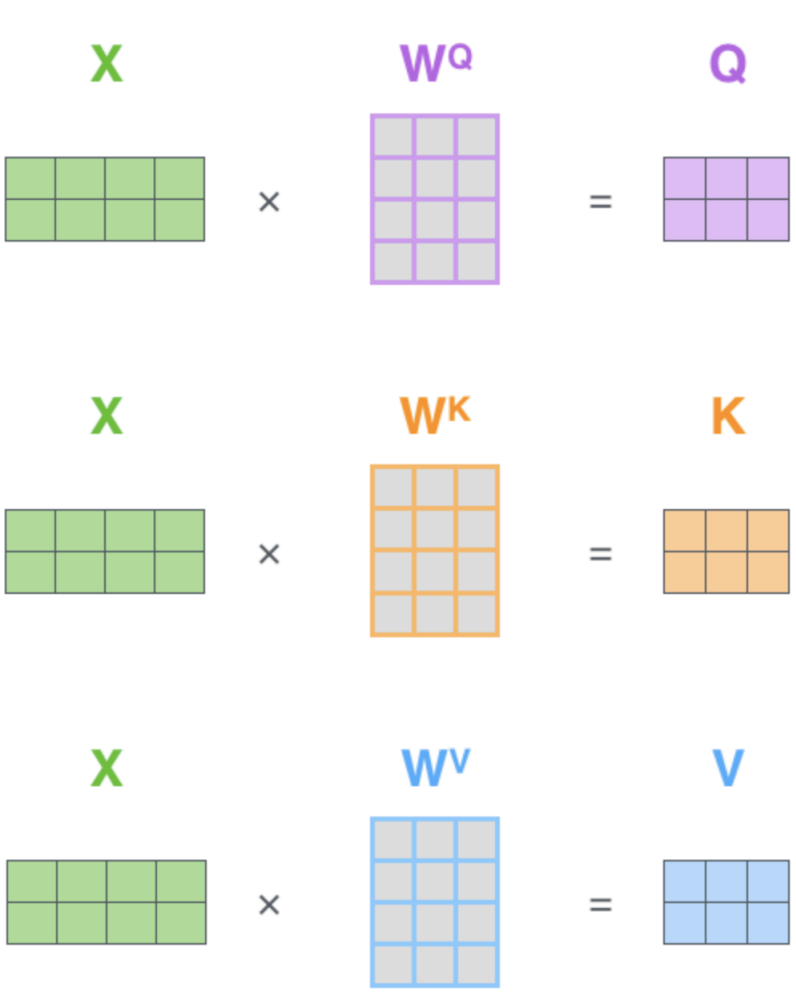
- Divide the scores by 8 (the square root of the dimension of the key vectors, i.e. 64)
- This leads to having more stable gradients
- Then pass the result through a softmax operation (normalization of scores)

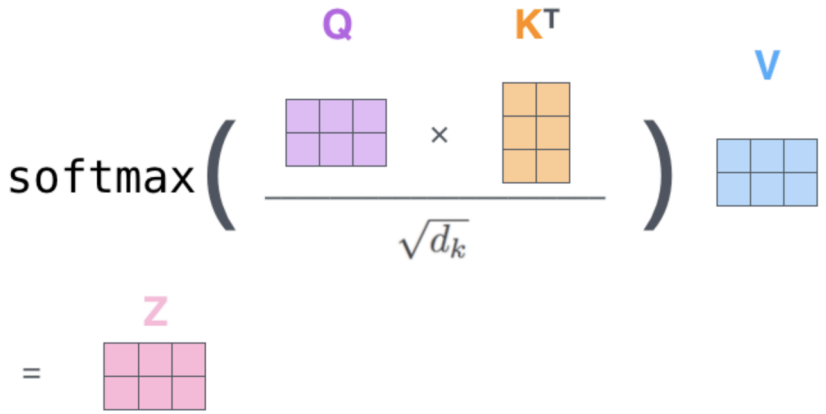
Steps V-VI: scaling and summation



- Multiply each value vector by the softmax score
- Intuition: maintain the values of the word(s) we want to focus on, and drown-out irrelevant words (multiplying them by tiny values)
- Output of the self-attention layer at each position: sum of the weighted value vectors

Self-attention: matrix calculation

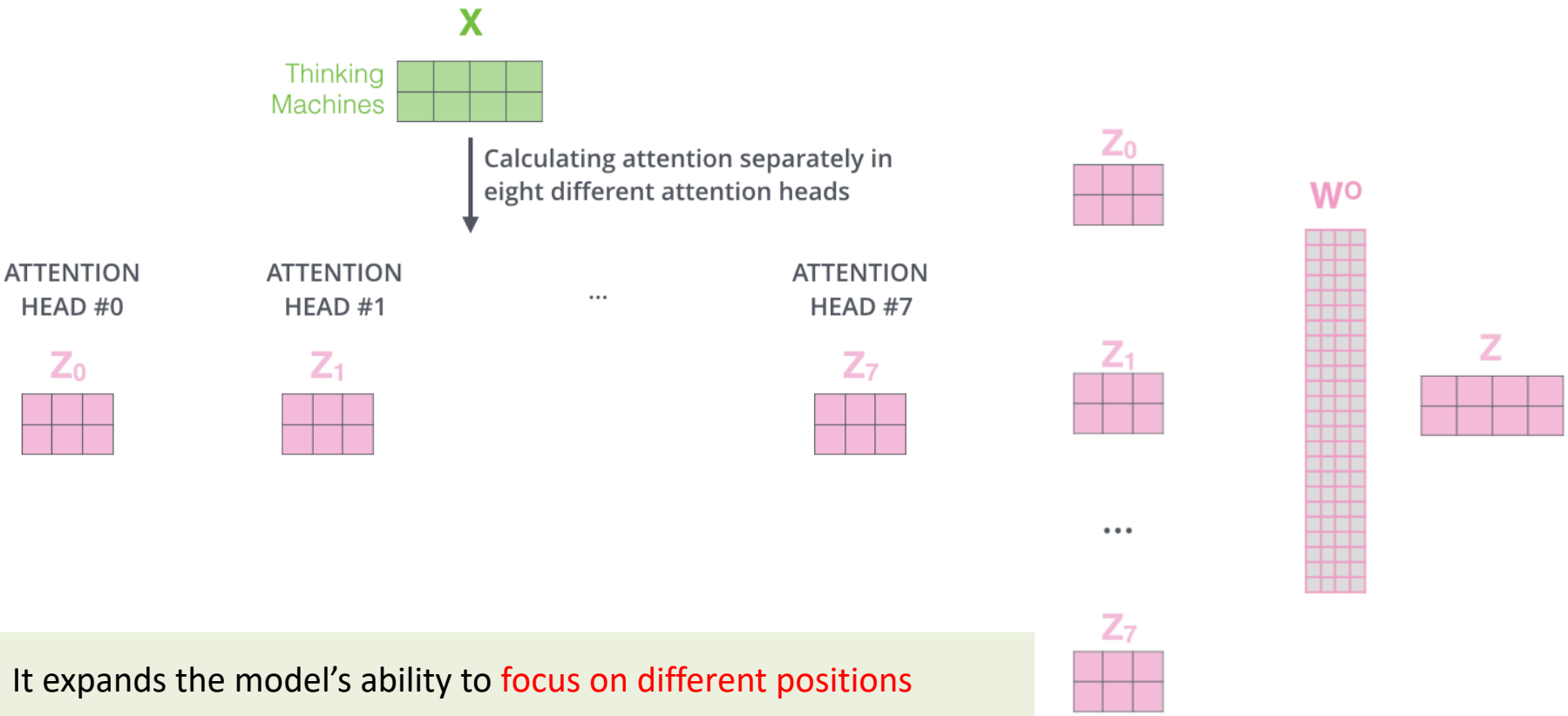




$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\text{T}}}{\sqrt{d_k}}\right)V$$

dimensionality of the key vectors

Multiple heads



- It expands the model's ability to focus on different positions
- It offers the attention layer multiple "representation subspaces"
- It tries to capture different definitions of "relevance"

Multiple heads

1) This is our input sentence*

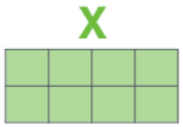
2) We embed each word*

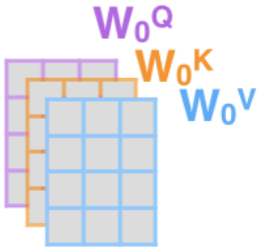
3) Split into 8 heads. We multiply X or R with weight matrices

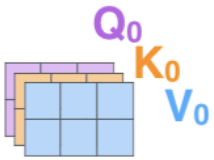
4) Calculate attention using the resulting Q/K/V matrices

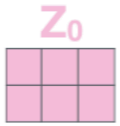
5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

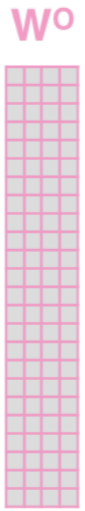
Thinking Machines

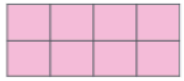




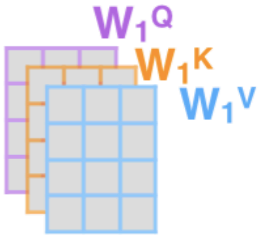


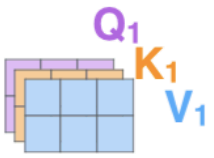


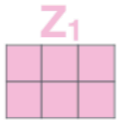




* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one



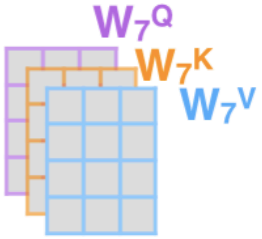


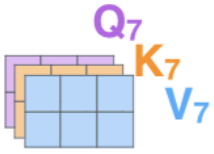


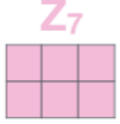
...

...

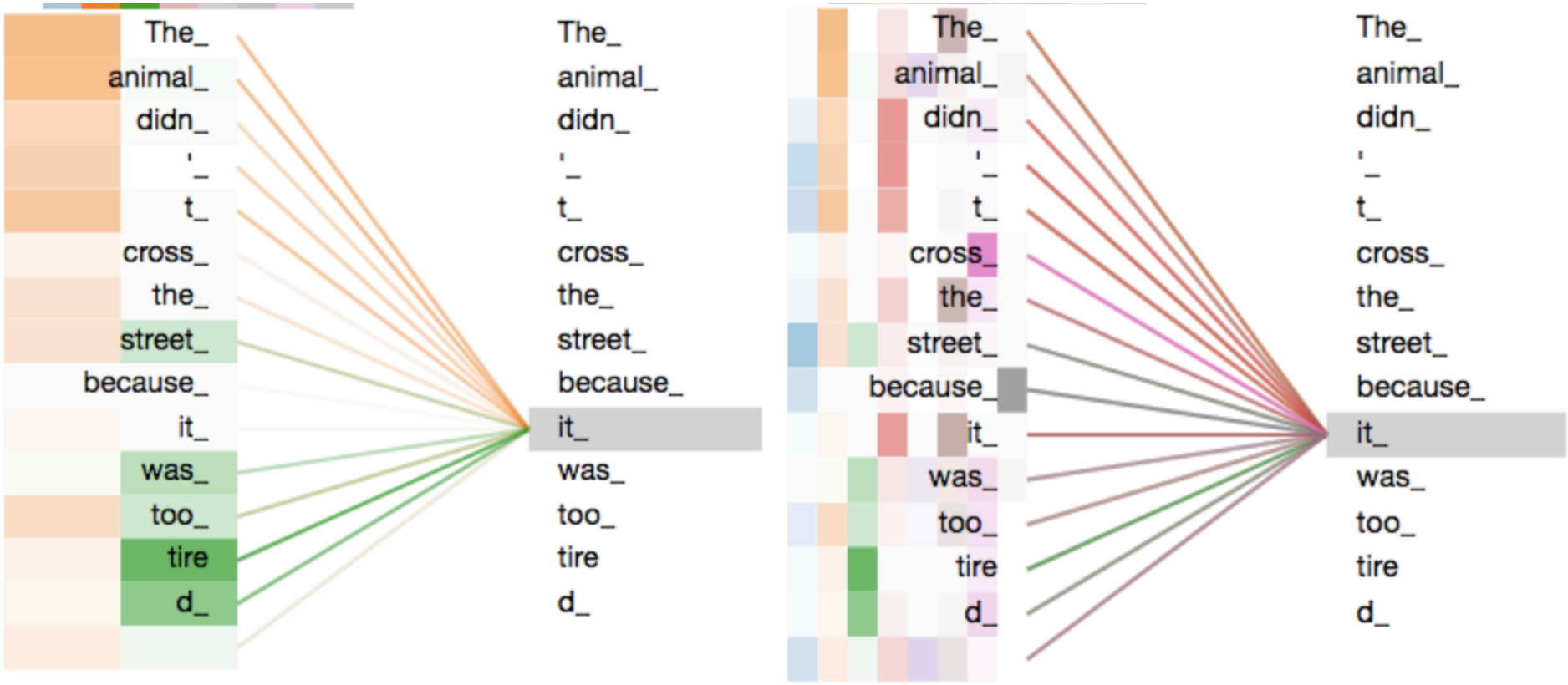
...







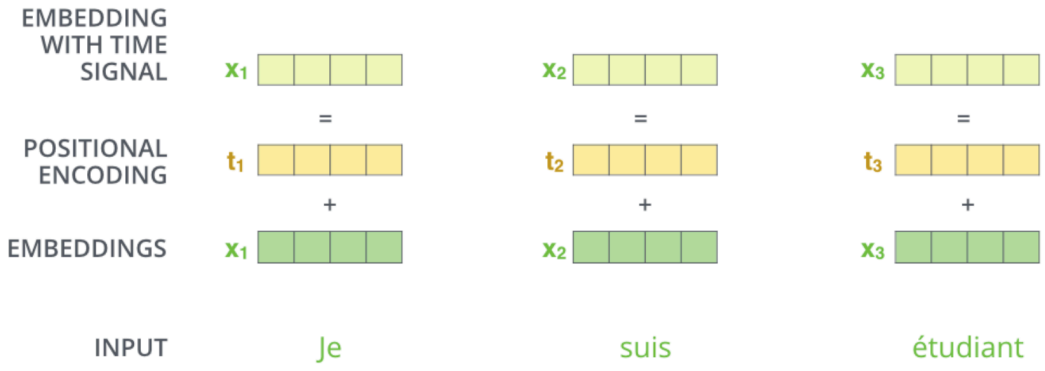
Multiple heads: semantics



- **Two attention heads:** as we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired"
- If we add all the attention heads to the picture, things can be harder to interpret

Positional encoding

- How do we account for the order of the words?
  - the transformer adds a vector to each input embedding
  - these vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence
  - Intuition: adding these values to the embeddings provides meaningful distances between the vectors once they are projected into Q/K/V and during dot-product attention

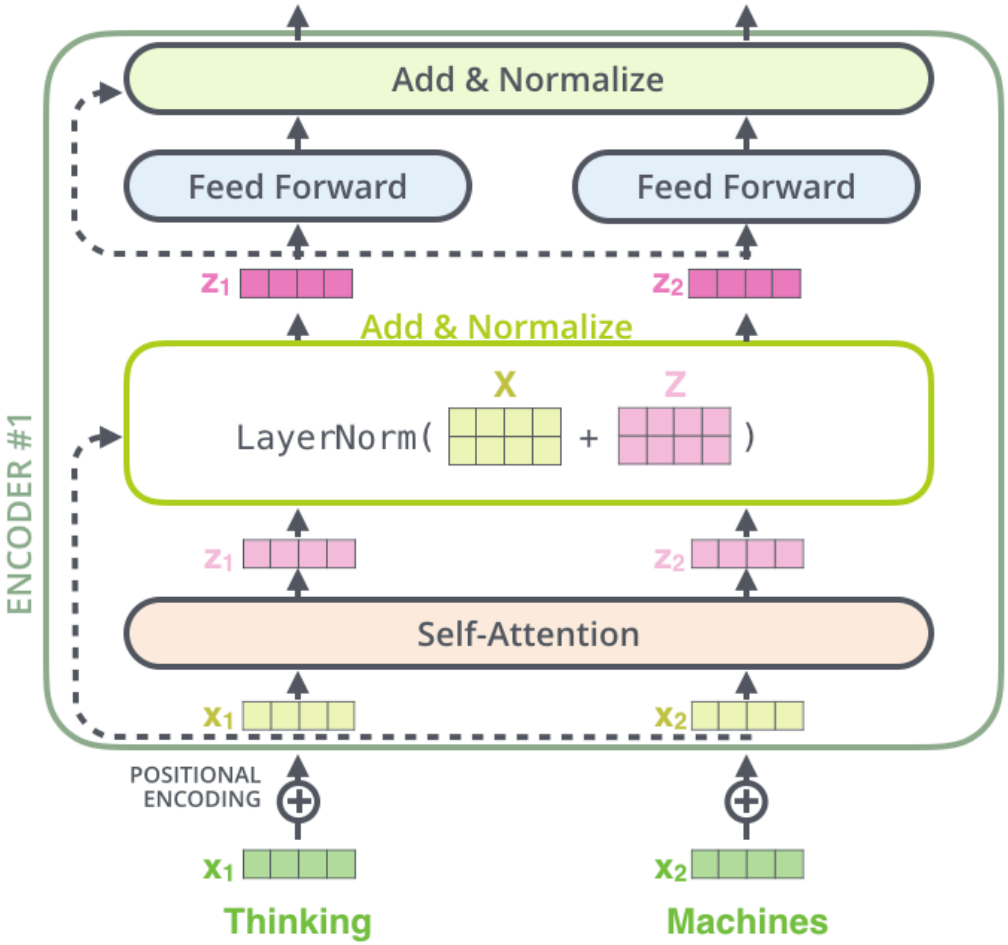


$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Layer normalization

Regulate the computation:

- **normalization layer** so that each feature (column) has the same average and standard deviation
- normalizes the inputs across the features

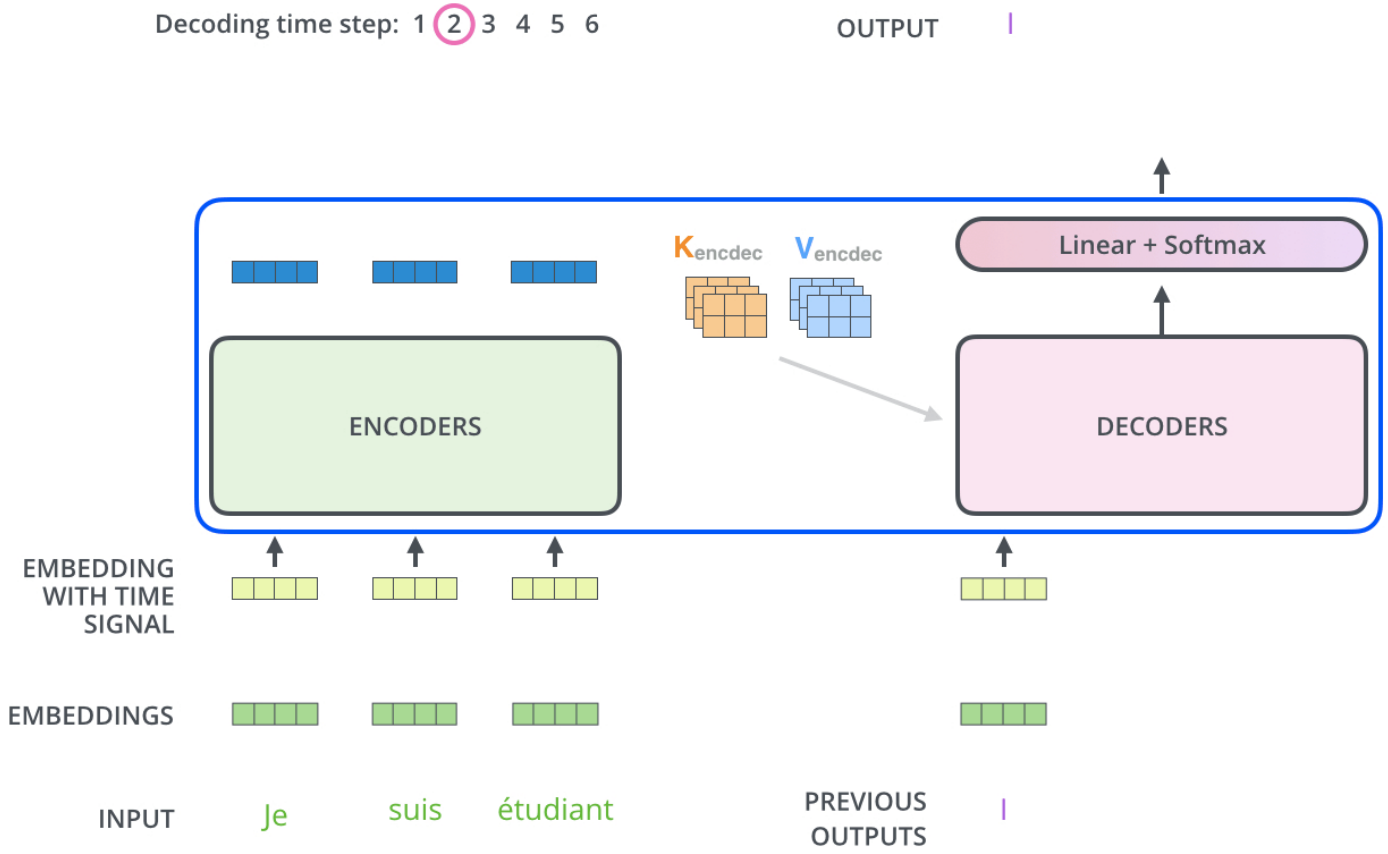


The decoder

- The encoder starts by processing the input sequence
- The output of the top encoder is then transformed into a set of attention vectors K and V
- These are to be used by each decoder in its "encoder-decoder attention" layer helping the decoder focus on appropriate places in the input sequence
- "Encoder-Decoder Attention" layer: works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack
- Self-attention layer: only allowed to attend to earlier positions in the output sequence by masking future positions (using "-inf") before the softmax step in the self-attention calculation

The decoder

- The encoder starts by processing the input sequence
- The output of the top encoder is then transformed into a set of attention vectors K and V

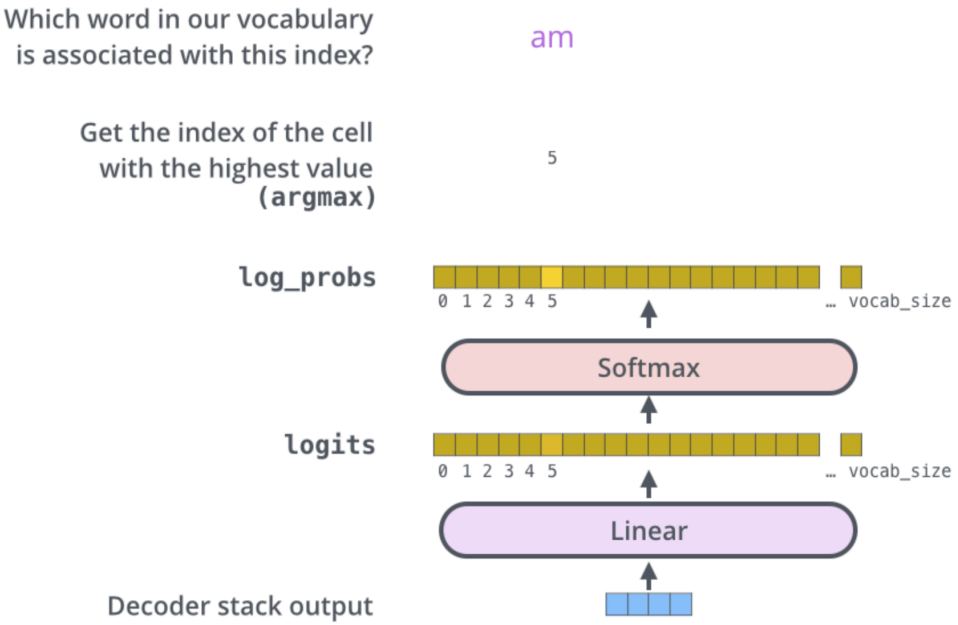


The final layer

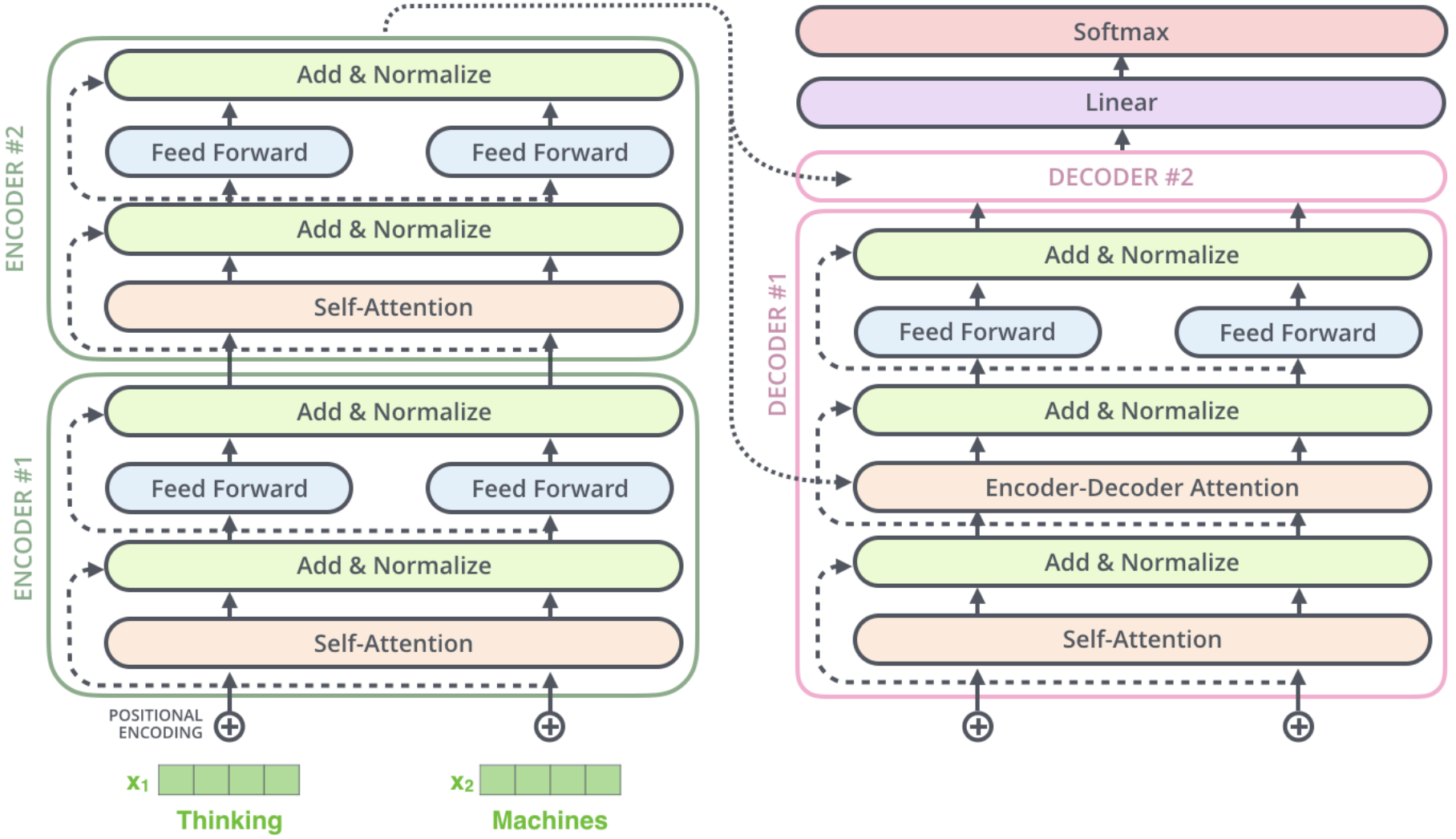
- The decoder stack outputs a vector of floats
- How do we turn that into a word?
  - final Linear layer, followed by a Softmax Layer
- **Linear layer:** a simple fully connected neural network that projects the vector of the stack of decoders into a larger vector called logits vector

- Assume the model knows 10,000 unique words learned from the training set
- **Logits vector:** 10,000 cells
- The softmax layer then turns those scores into probabilities

The cell with the highest probability is chosen, and the word associated with it is produced as output for this time step



The complete transformer



Loss function

- **Simple task:** translate “merci” to “thanks”

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word “am”

Use one-hot encoding to represent the output vocabulary

Untrained Model Output

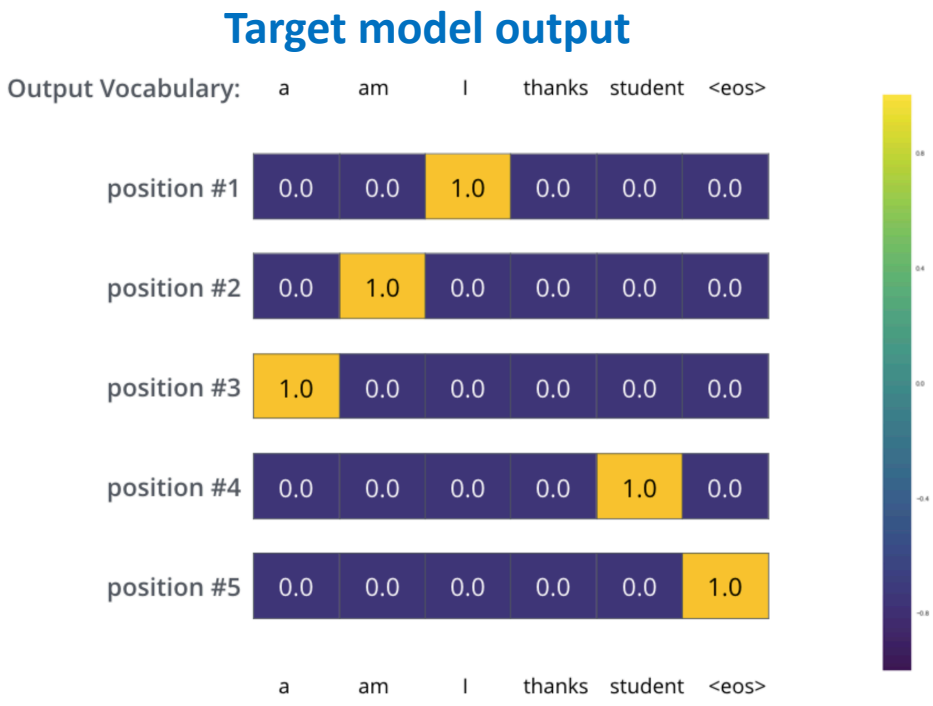
Correct and desired output

a am I thanks student <eos>

Cross-entropy loss or Kullback–Leibler divergence

More realistically speaking

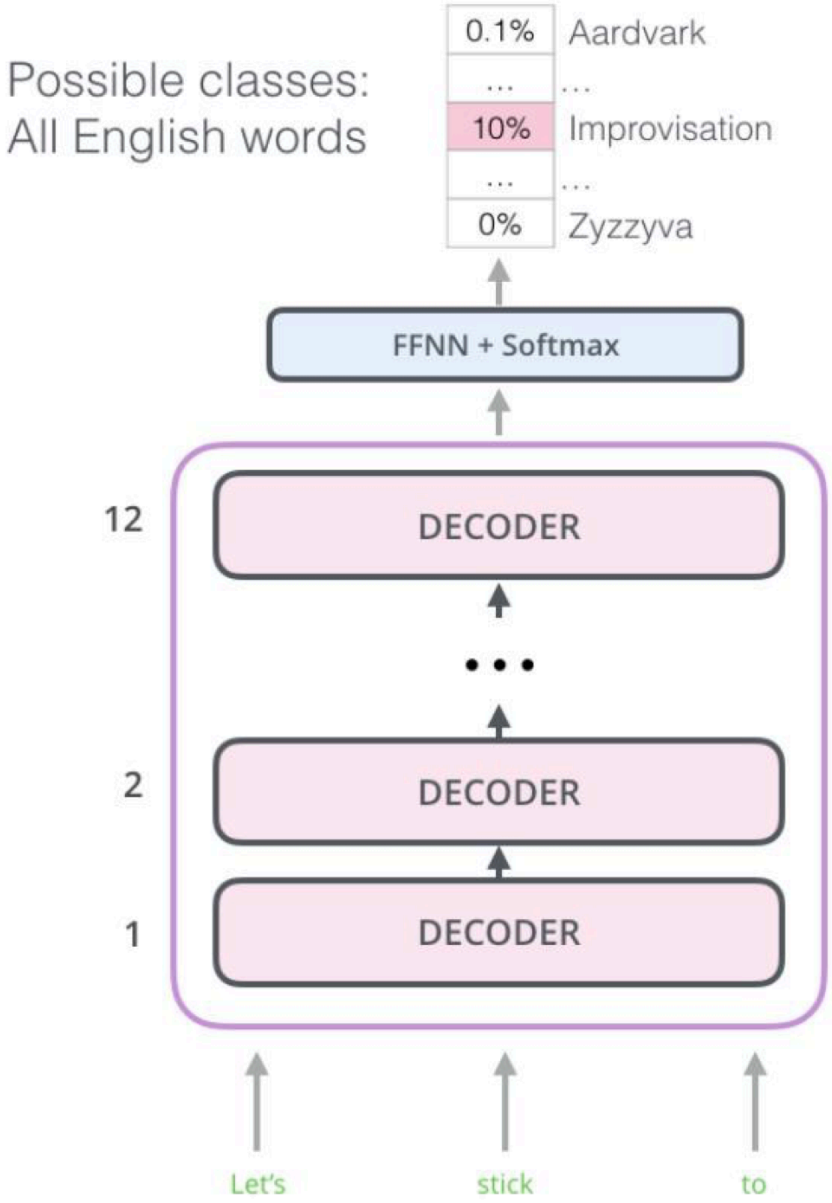
- **Realisic task:** translate “je suis étudiant” to “i am a student”
- **Realistic vocabulary size:** 30,000 or 50,000



Transformers for classification

- How do we use **transformers** for **classification**?

- **Simple:** add an FFNN and a softmax layer on top!



TODOs

- **Reading:**
  - The Deep Learning Book: Chapter 20
  - Additional material: online
- **Homework 2**
  - Due: Feb 23