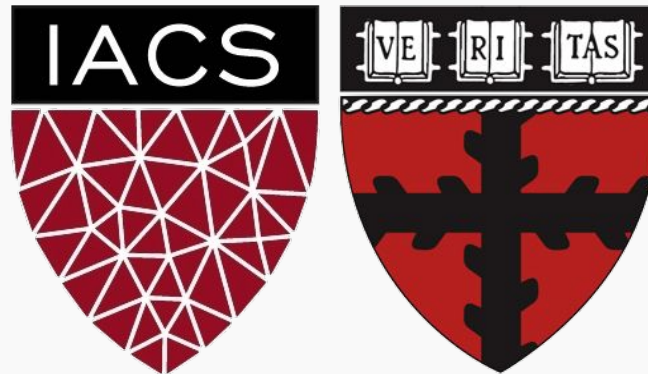


# Lecture 17: Language Modelling 2

## CS109B Data Science 2

Pavlos Protopapas, Mark Glickman, and Chris Tanner



# Outline

---

- Seq2Seq +Attention
- Transformers +BERT
- Embeddings

# Embedding of “stick” in “Let’s stick to” - Step #2

1- Concatenate hidden layers



2- Multiply each vector by a weight based on the task

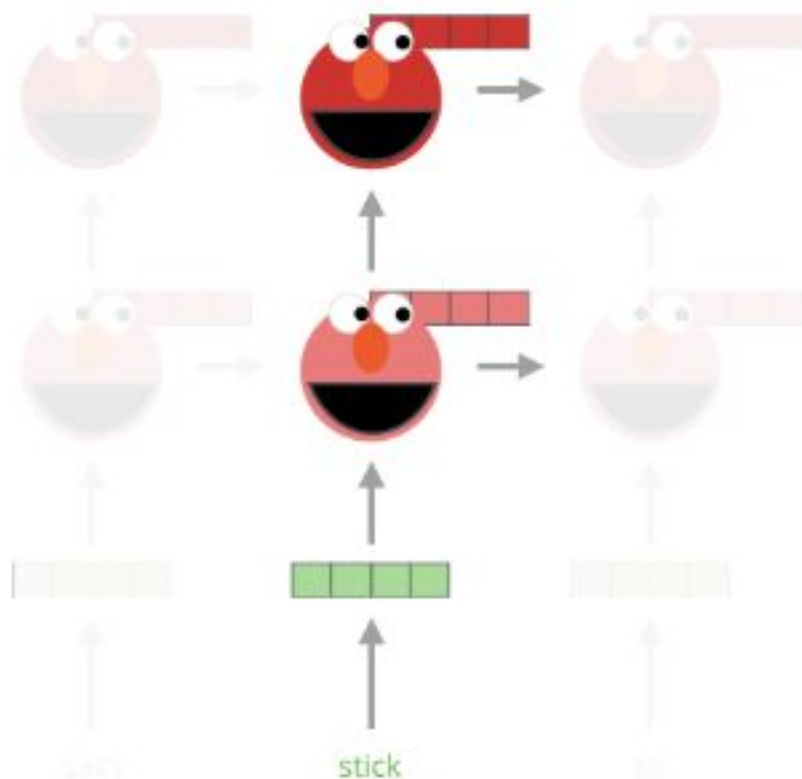


3- Sum the (now weighted) vectors

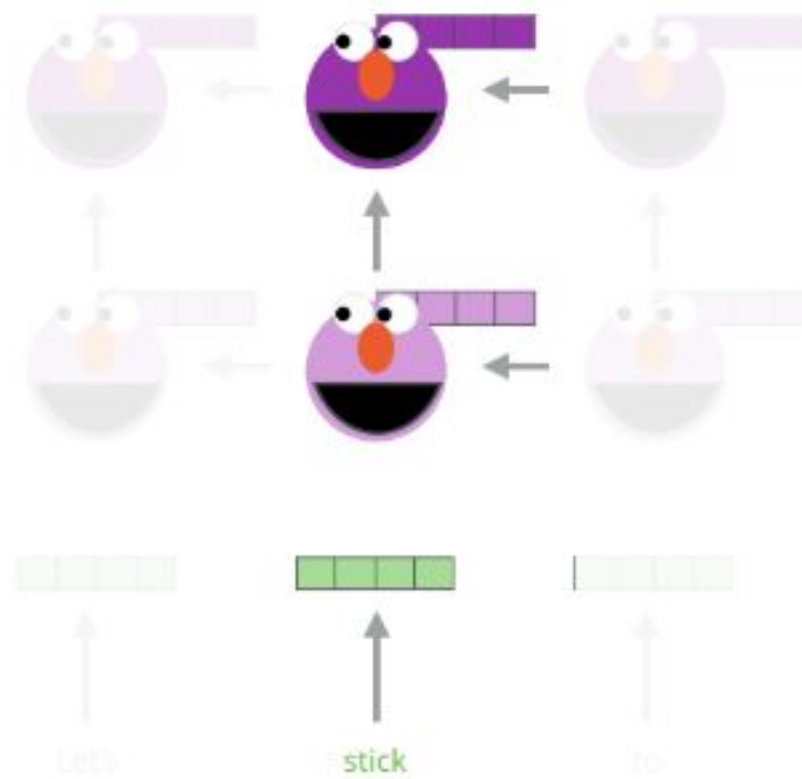


ELMo embedding of “stick” for this task in this context

Forward Language Model



Backward Language Model



# ELMo: Stacked Bi-directional LSTMs

---

- ELMo yielded incredibly good word embeddings, which yielded state-of-the-art results when applied to many NLP tasks.
- **Main ELMo takeaway:** given enough training data, having tons of explicit connections between your vectors is useful (system can determine how to best use context)

## REFLECTION

So far, for all of our sequential modelling, we have been concerned with emitting 1 output per input datum.

Sometimes, a *sequence* is the smallest granularity we care about though (e.g., an English sentence)



# Outline

---

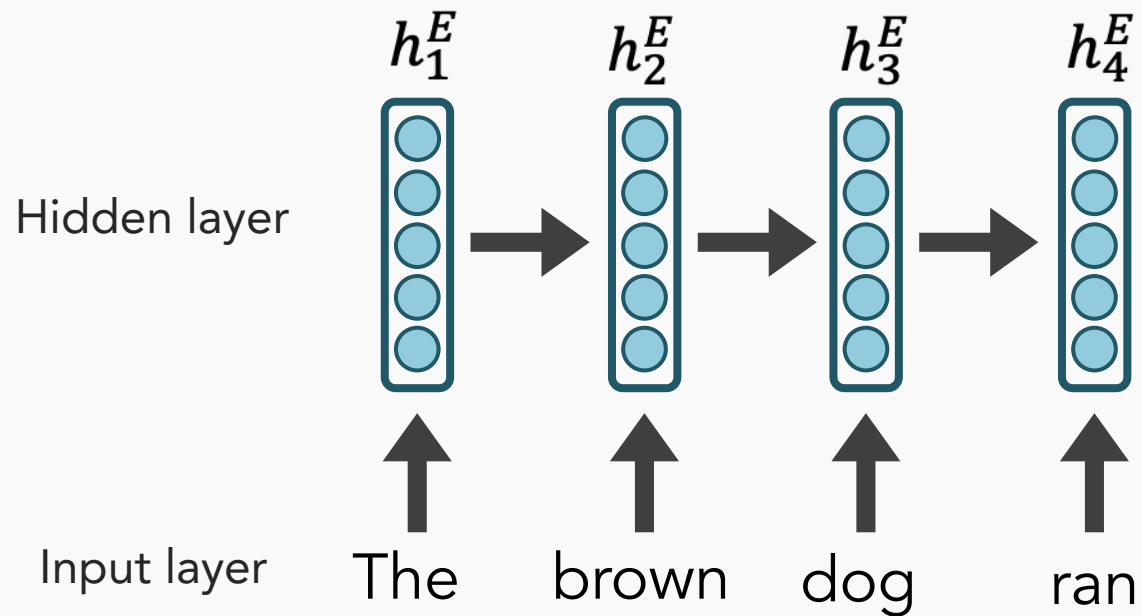
- **Seq2Seq +Attention**
- Transformers +BERT
- Embeddings

# Sequence-to-Sequence (seq2seq)

- If our input is a sentence in **Language A**, and we wish to translate it to **Language B**, it is clearly sub-optimal to translate word by word (like our current models are suited to do).
- Instead, let a **sequence** of tokens be the unit that we ultimately wish to work with (a sequence of length **N** may emit a sequences of length **M**)
- **Seq2seq** models are comprised of **2 RNNs**: 1 encoder, 1 decoder



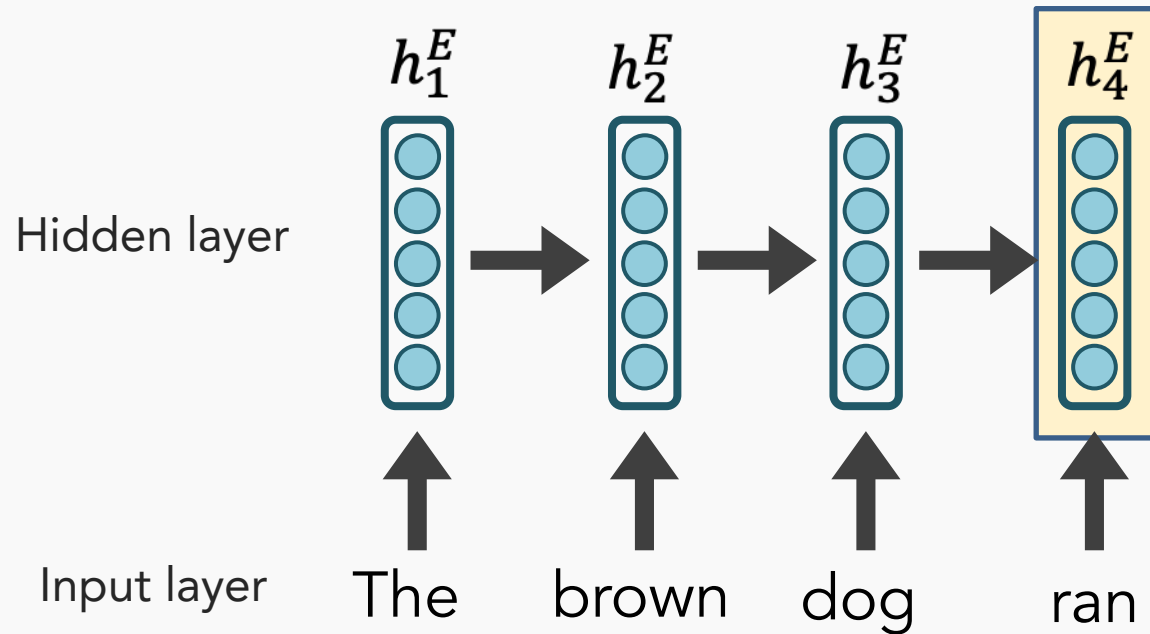
# Sequence-to-Sequence (seq2seq)



ENCODER RNN

# Sequence-to-Sequence (seq2seq)

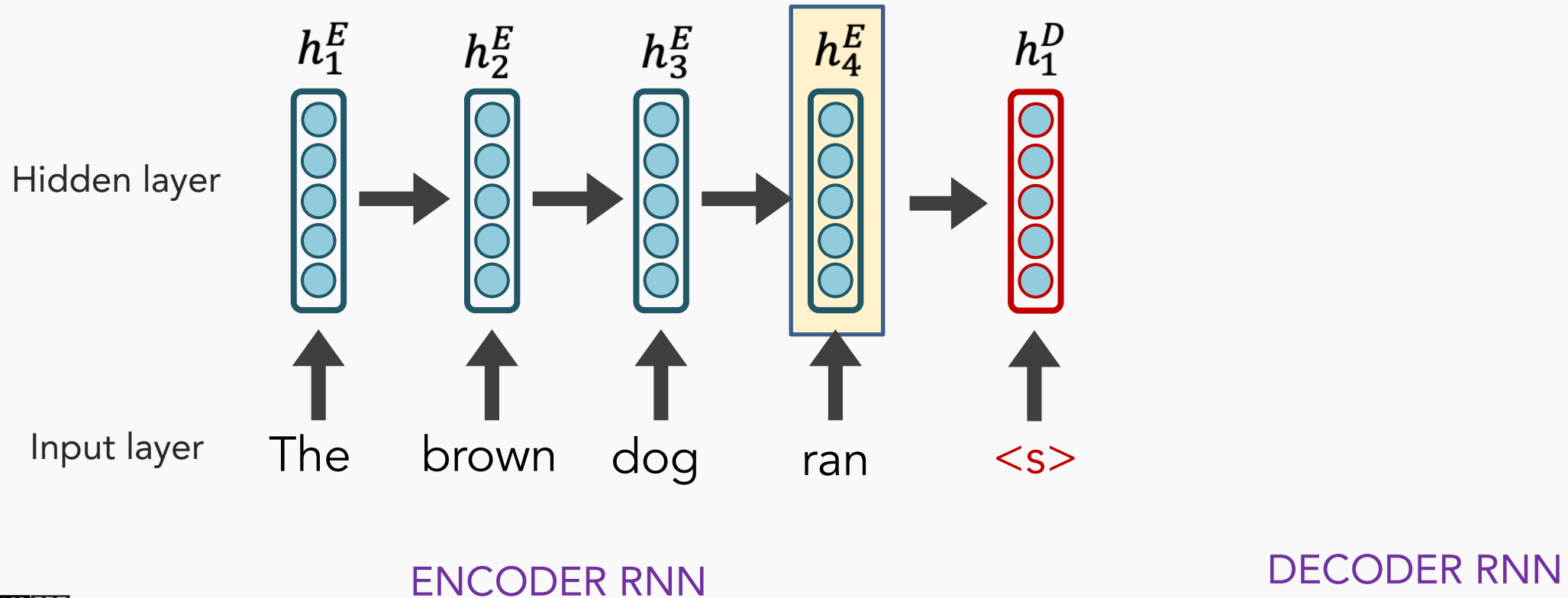
The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



ENCODER RNN

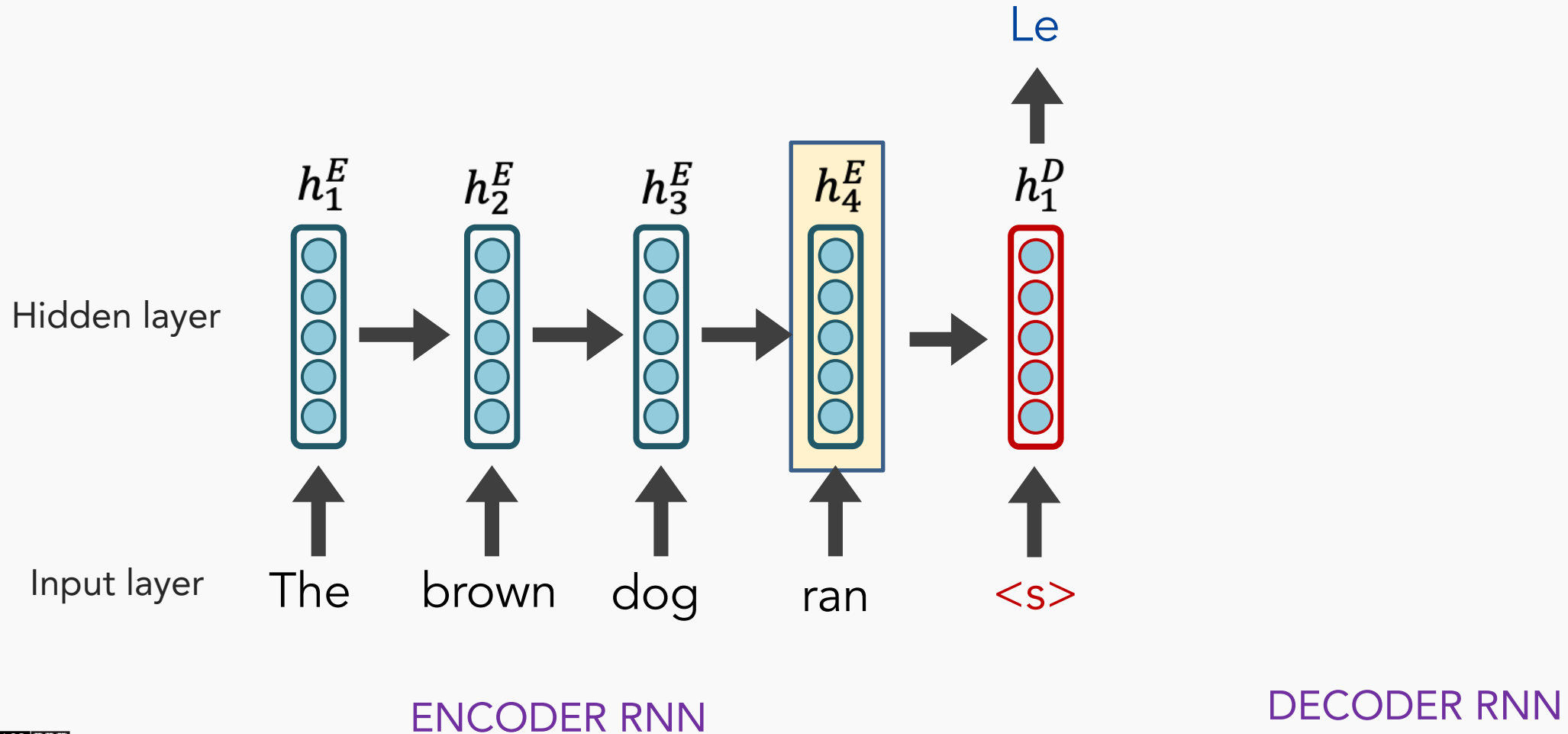
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



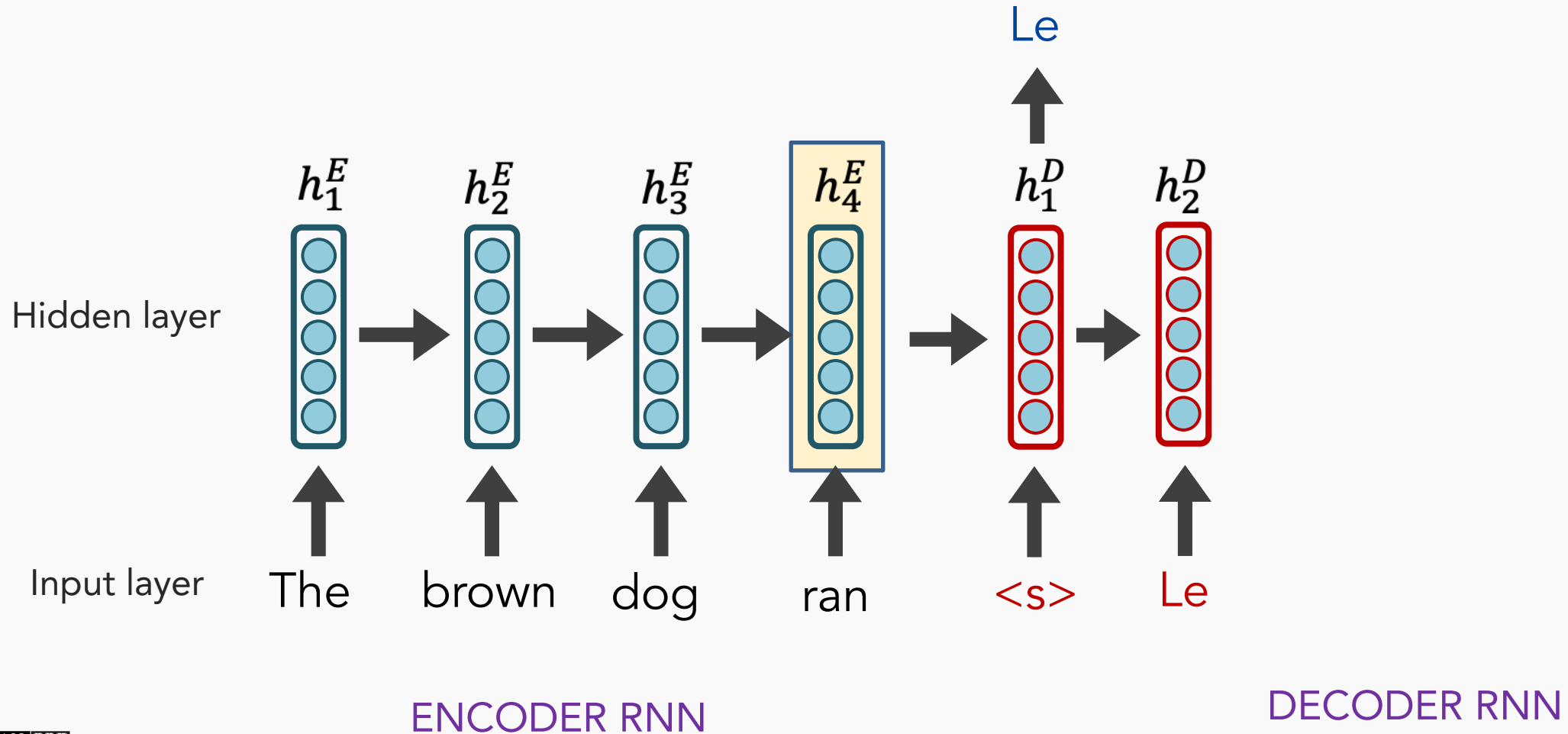
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



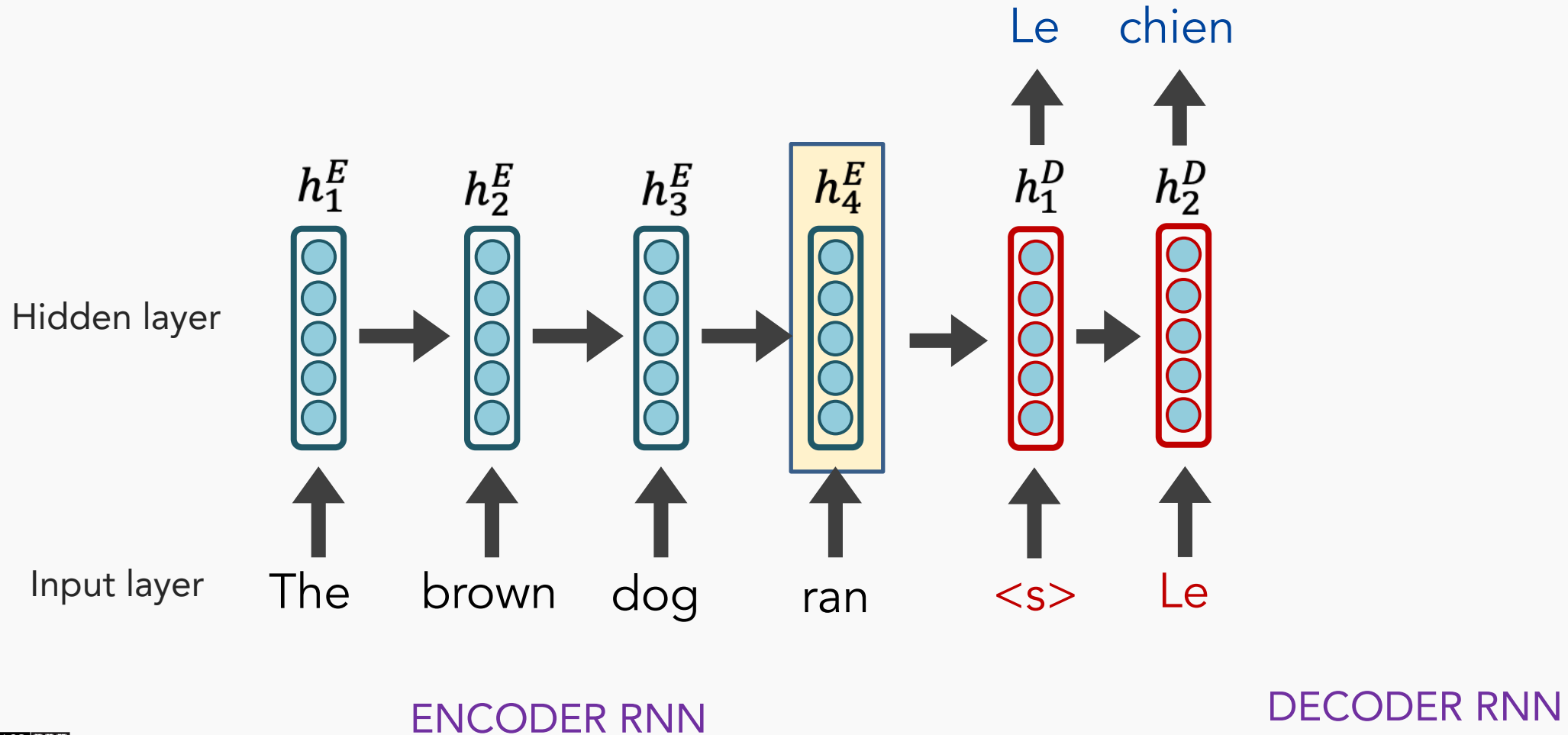
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



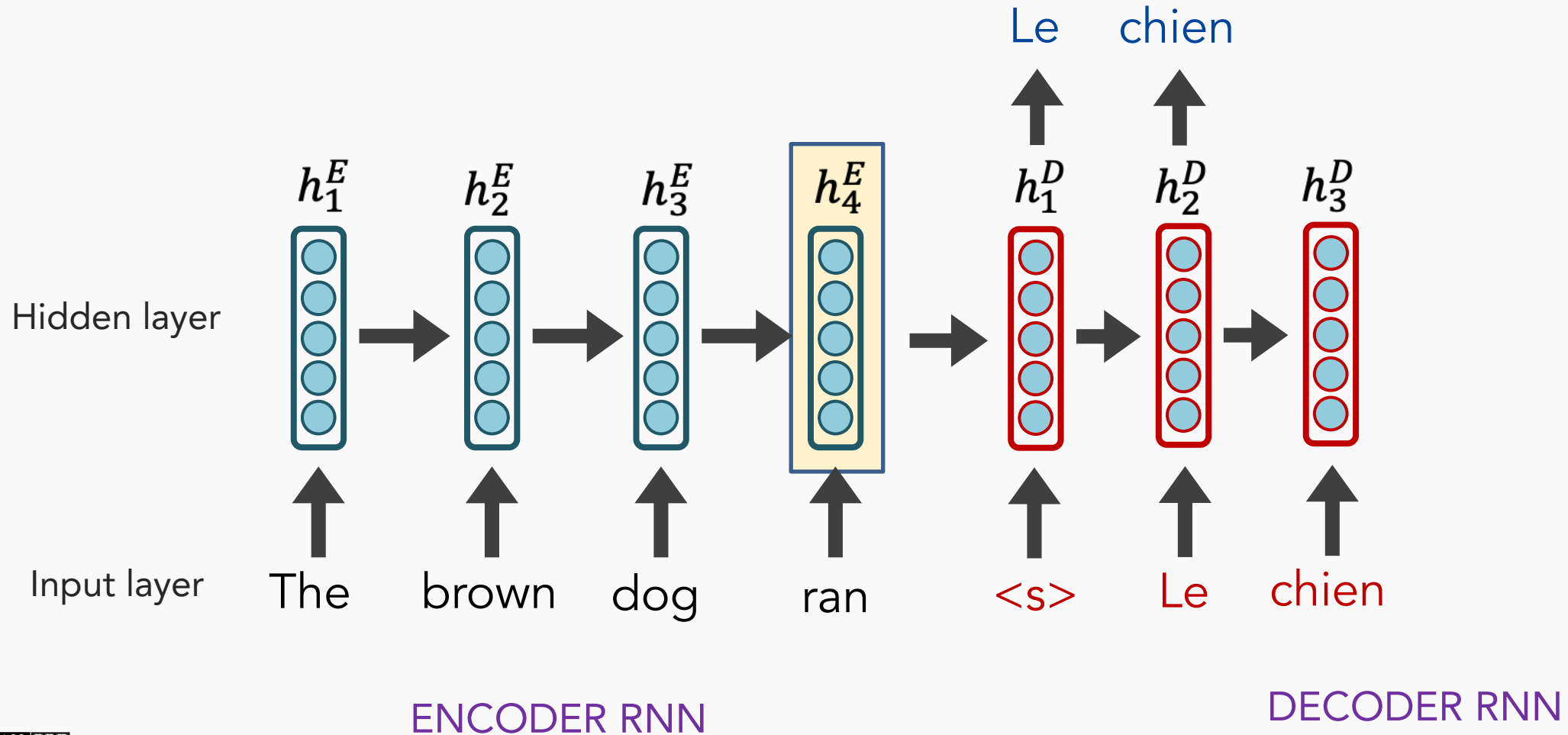
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



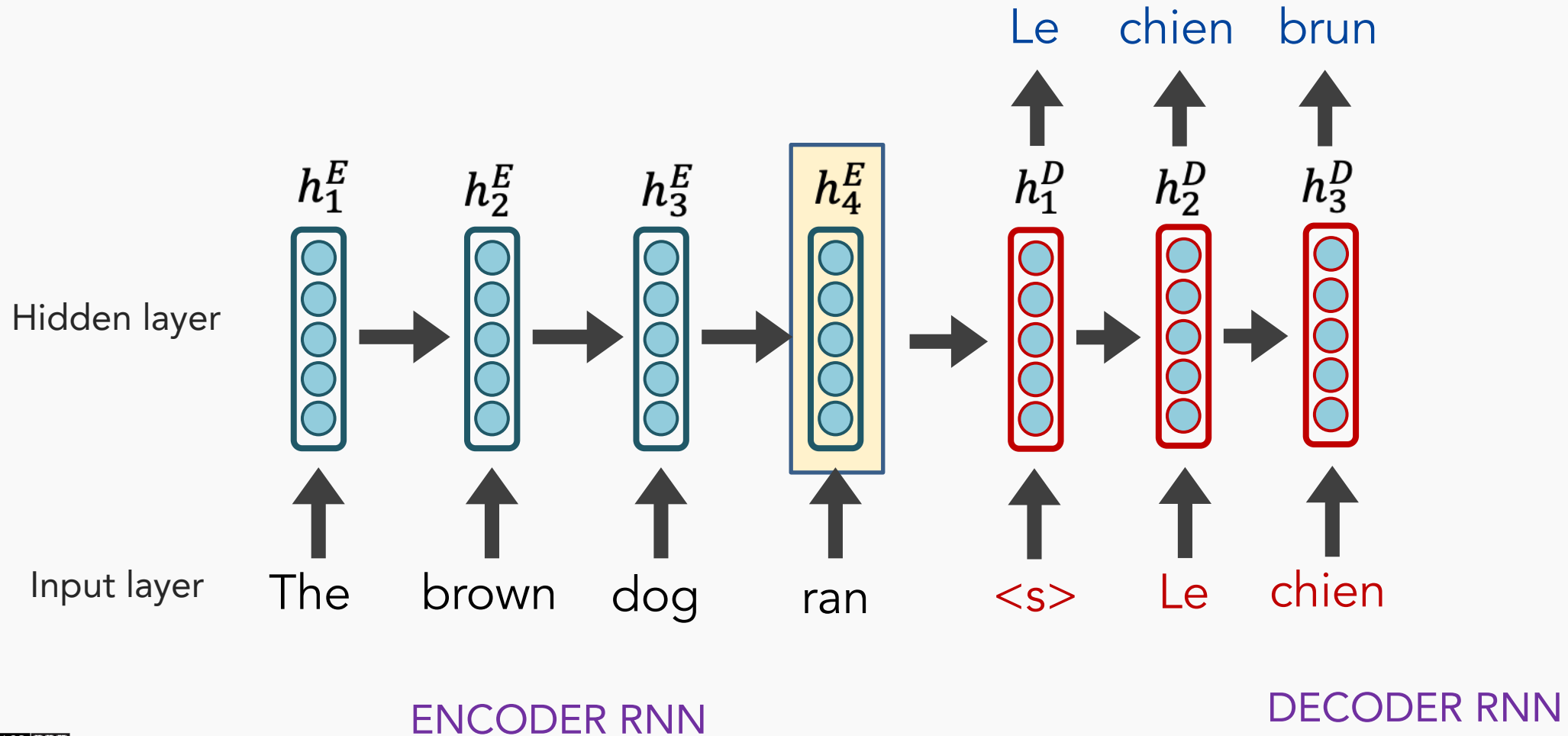
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



# Sequence-to-Sequence (seq2seq)

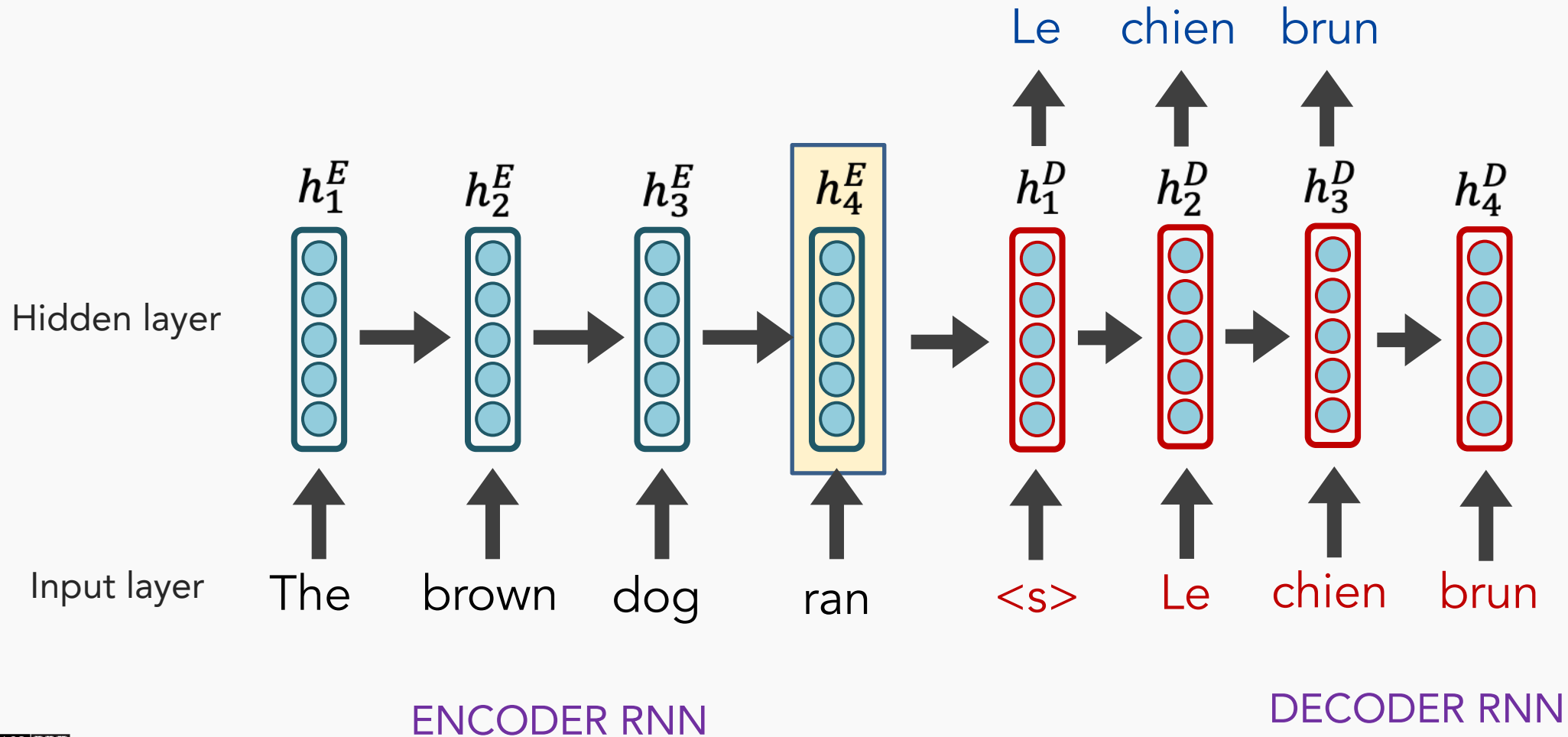
The final hidden state of the encoder RNN  
is the initial state of the decoder RNN





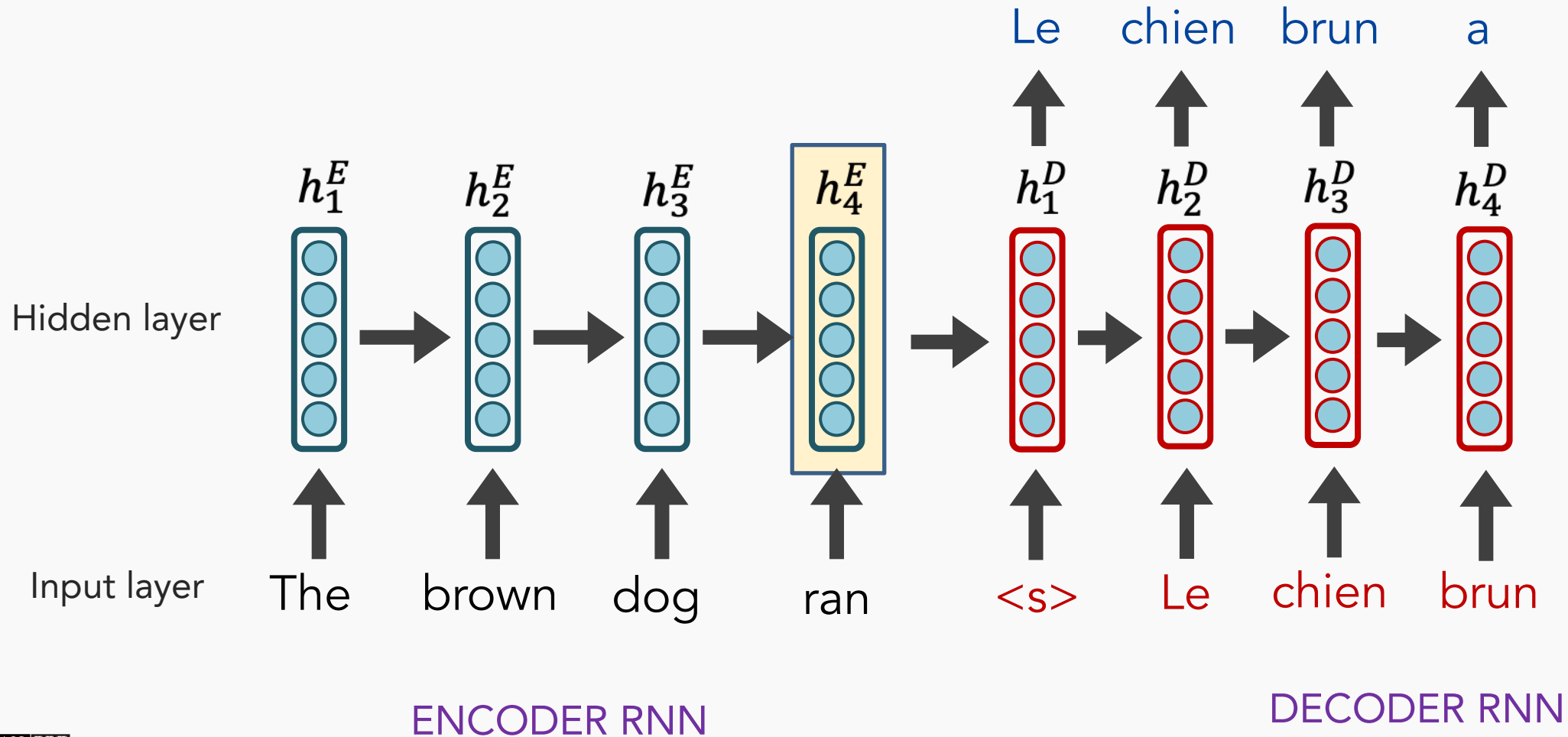
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



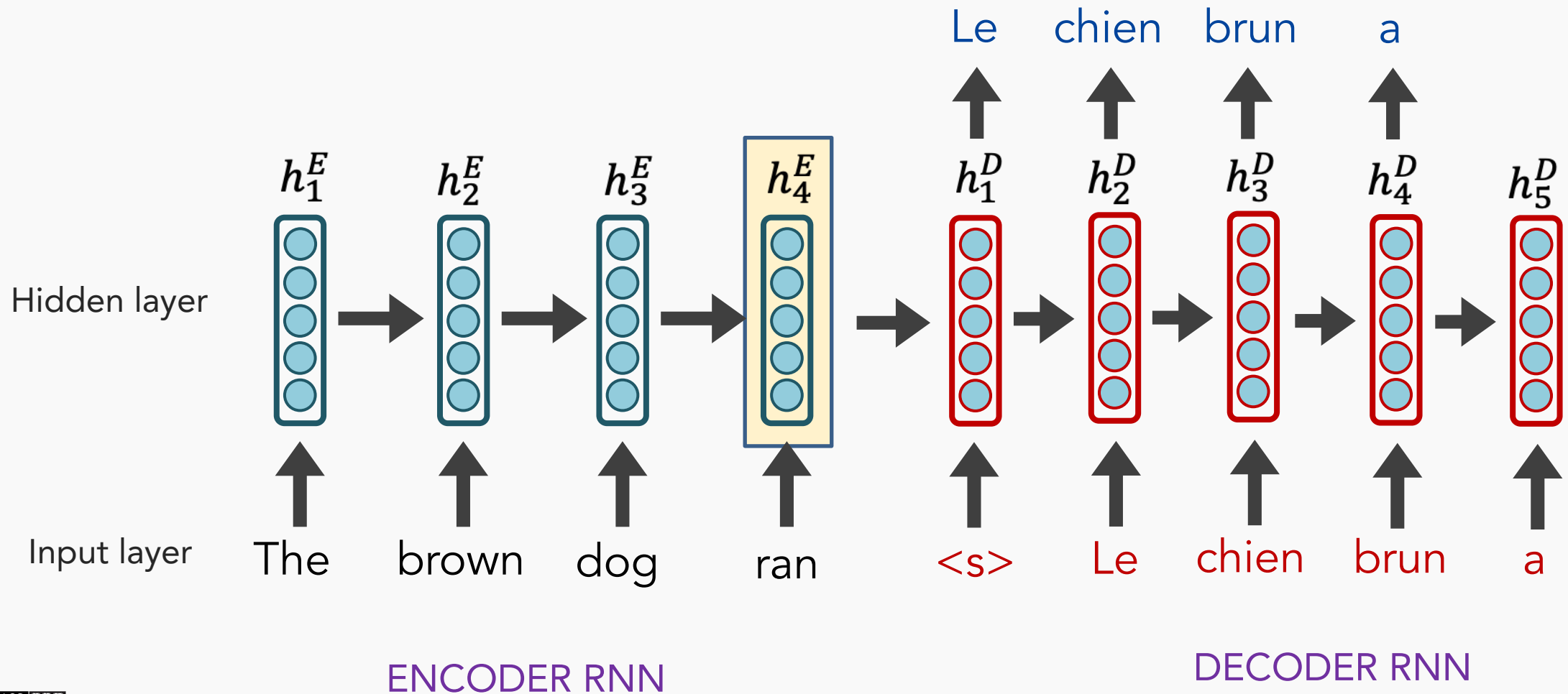
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



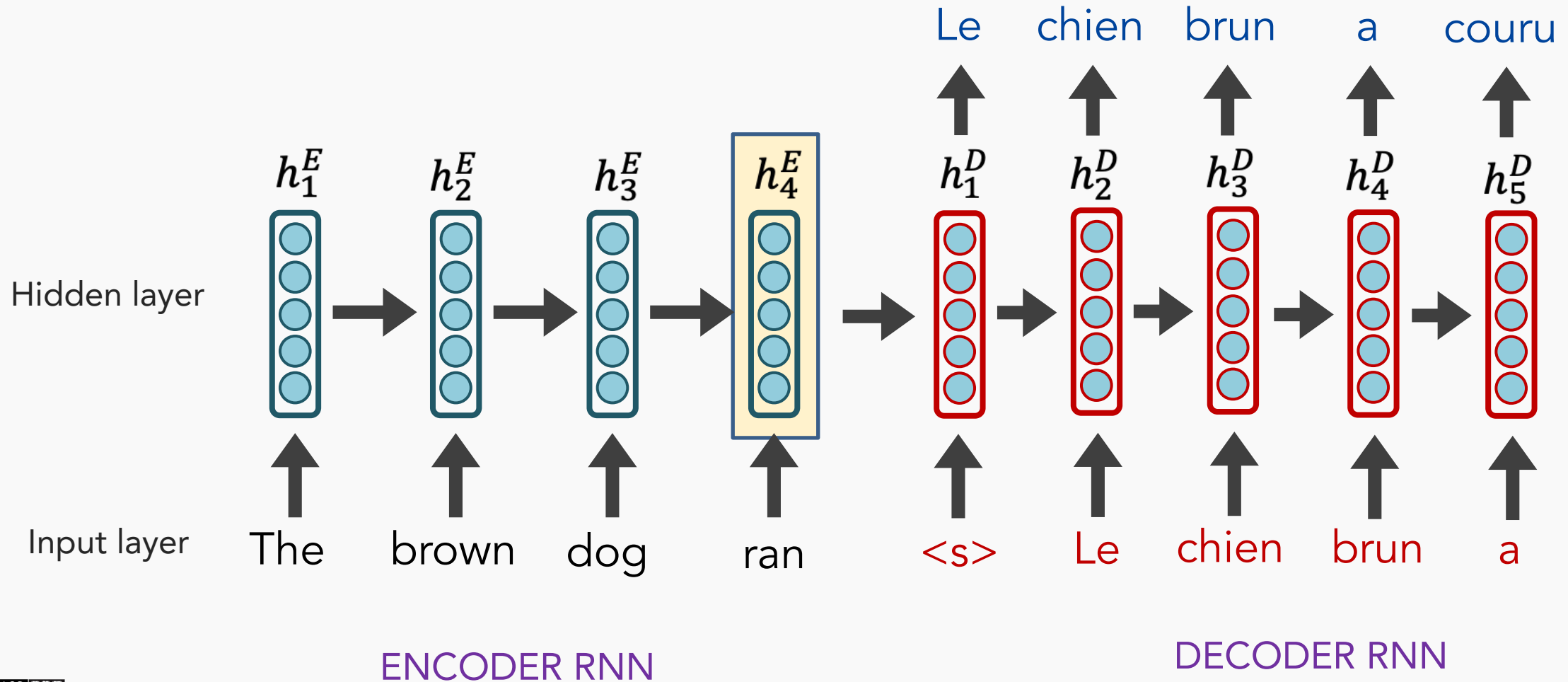
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



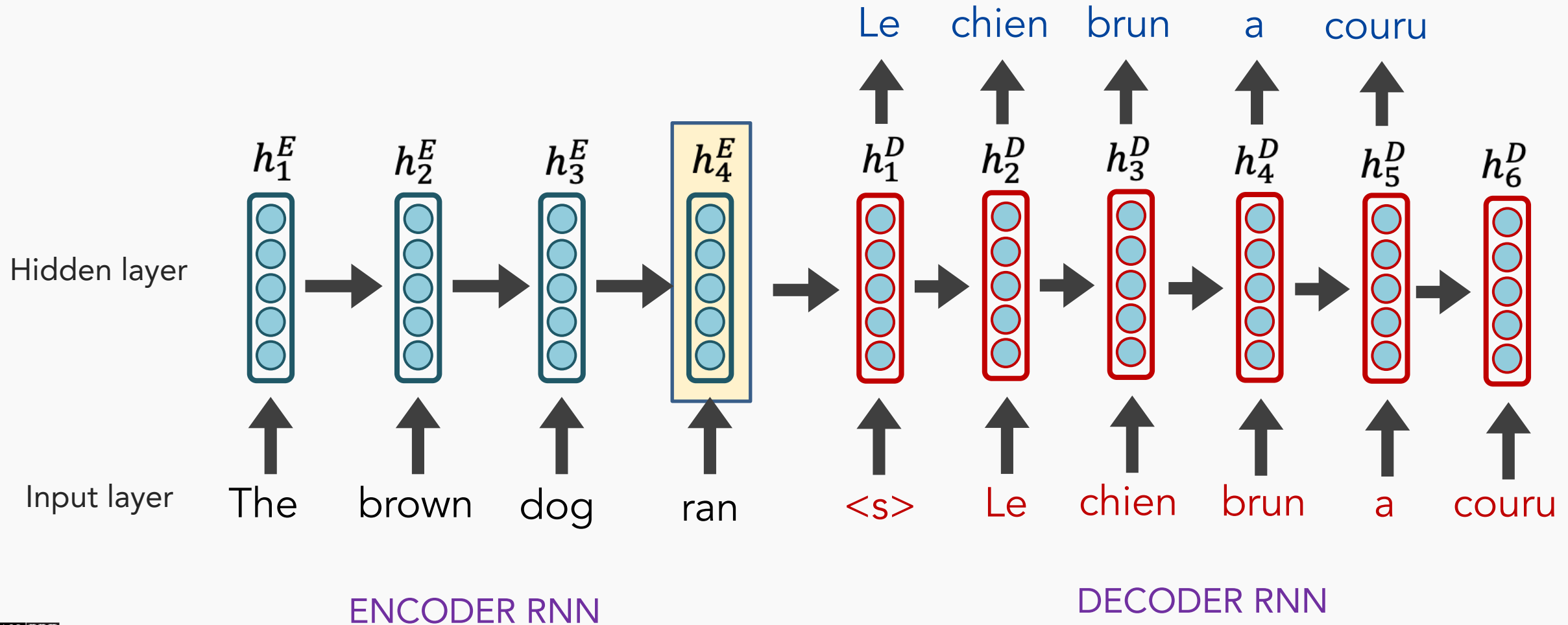
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



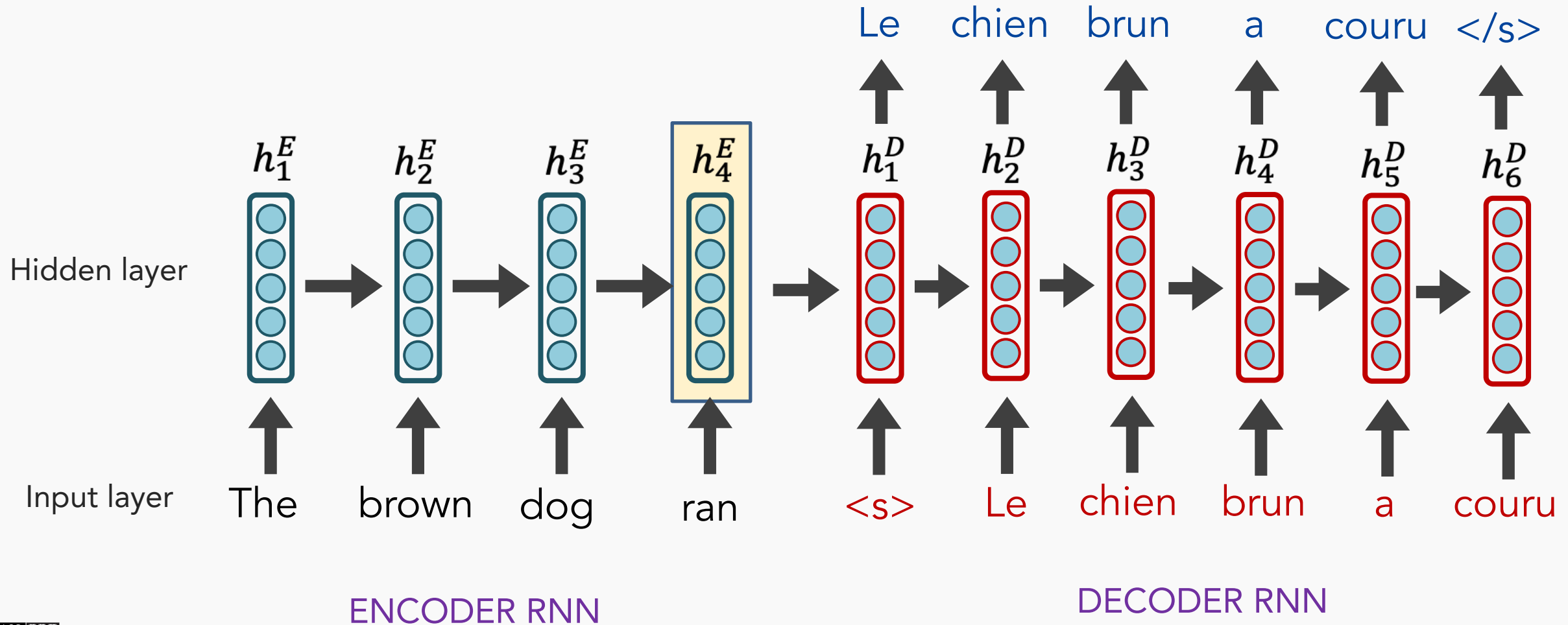
# Sequence-to-Sequence (seq2seq)

The final hidden state of the encoder RNN  
is the initial state of the decoder RNN

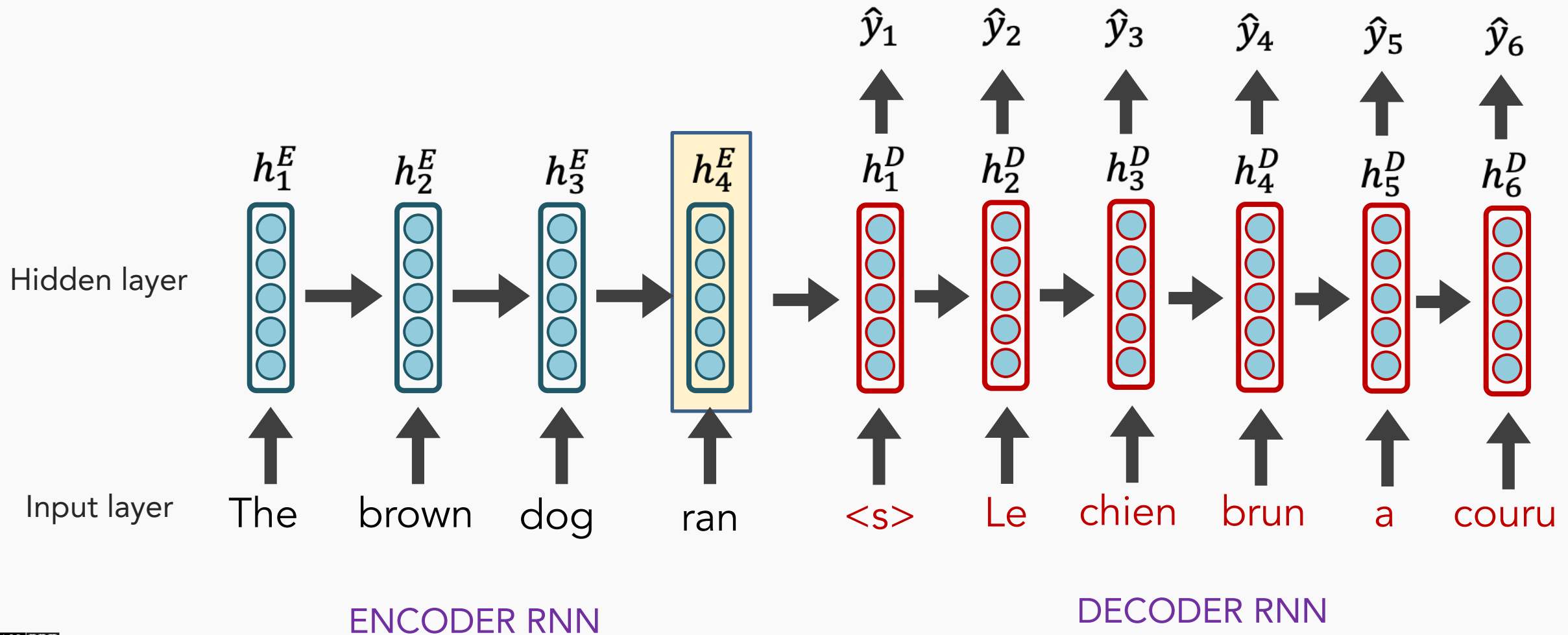


# Sequence-to-Sequence (seq2seq)

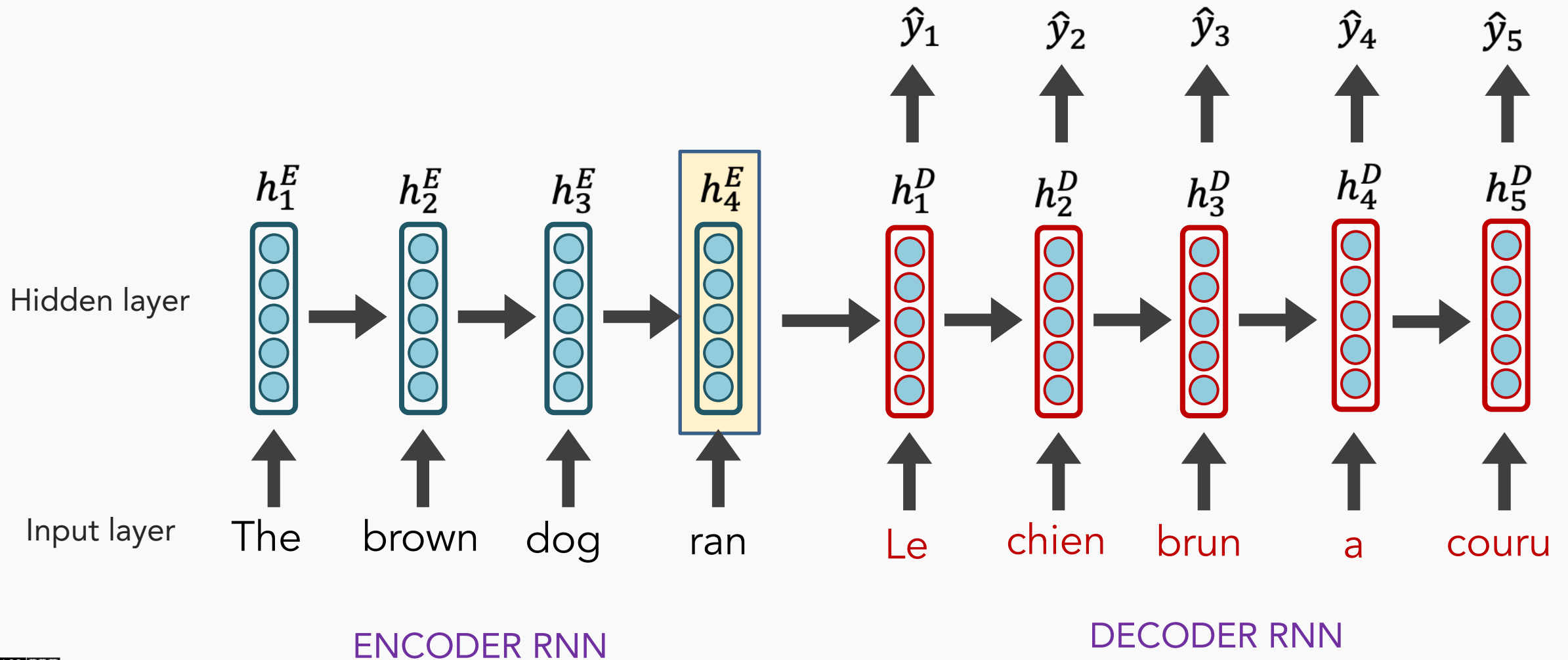
The final hidden state of the encoder RNN  
is the initial state of the decoder RNN



# Sequence-to-Sequence (seq2seq)



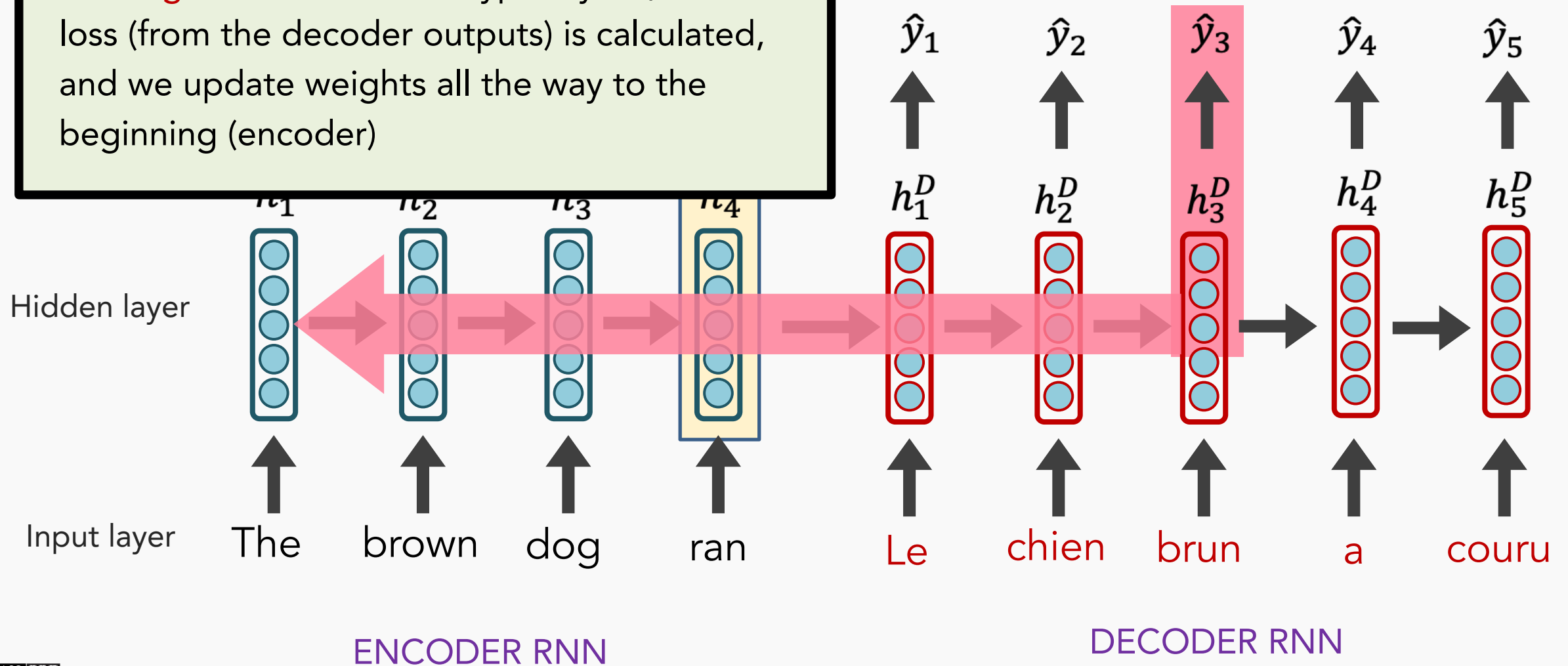
# Sequence-to-Sequence (seq2seq)





# Sequence-to-Sequence (seq2seq)

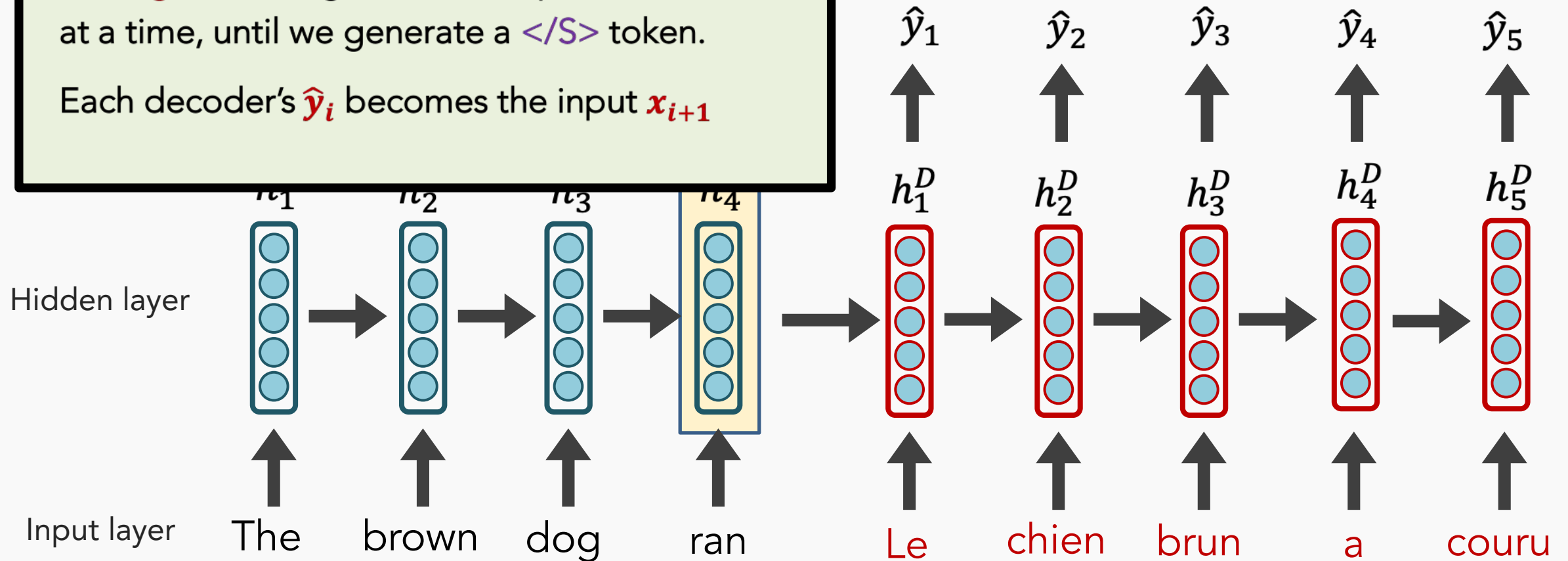
**Training** occurs like RNNs typically do; the loss (from the decoder outputs) is calculated, and we update weights all the way to the beginning (encoder)



# Sequence-to-Sequence (seq2seq)

**Testing** decoder generates outputs one word at a time, until we generate a  $\langle /S \rangle$  token.

Each decoder's  $\hat{y}_i$  becomes the input  $x_{i+1}$



ENCODER RNN

DECODER RNN

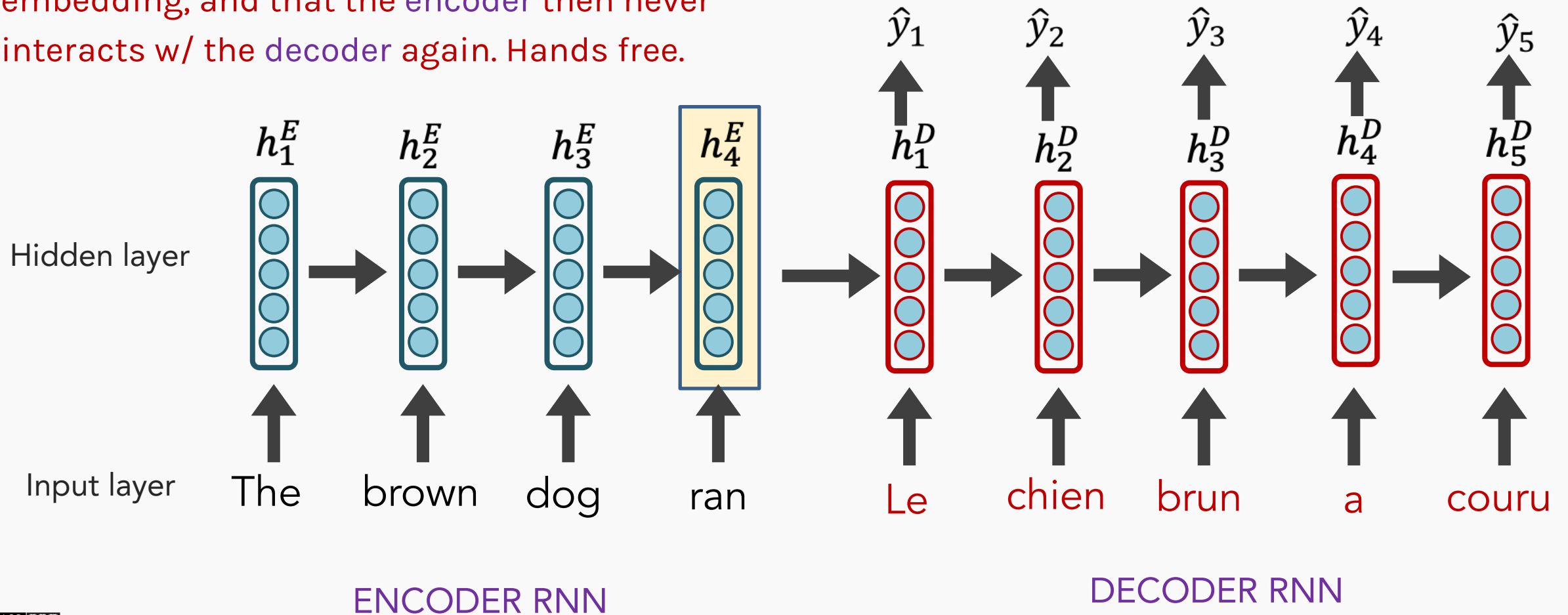
# Sequence-to-Sequence (seq2seq)

---

See any issues with this traditional **seq2seq** paradigm?

# Sequence-to-Sequence (seq2seq)

It's crazy that the entire “meaning” of the 1<sup>st</sup> sequence is expected to be packed into this one embedding, and that the encoder then never interacts w/ the decoder again. Hands free.



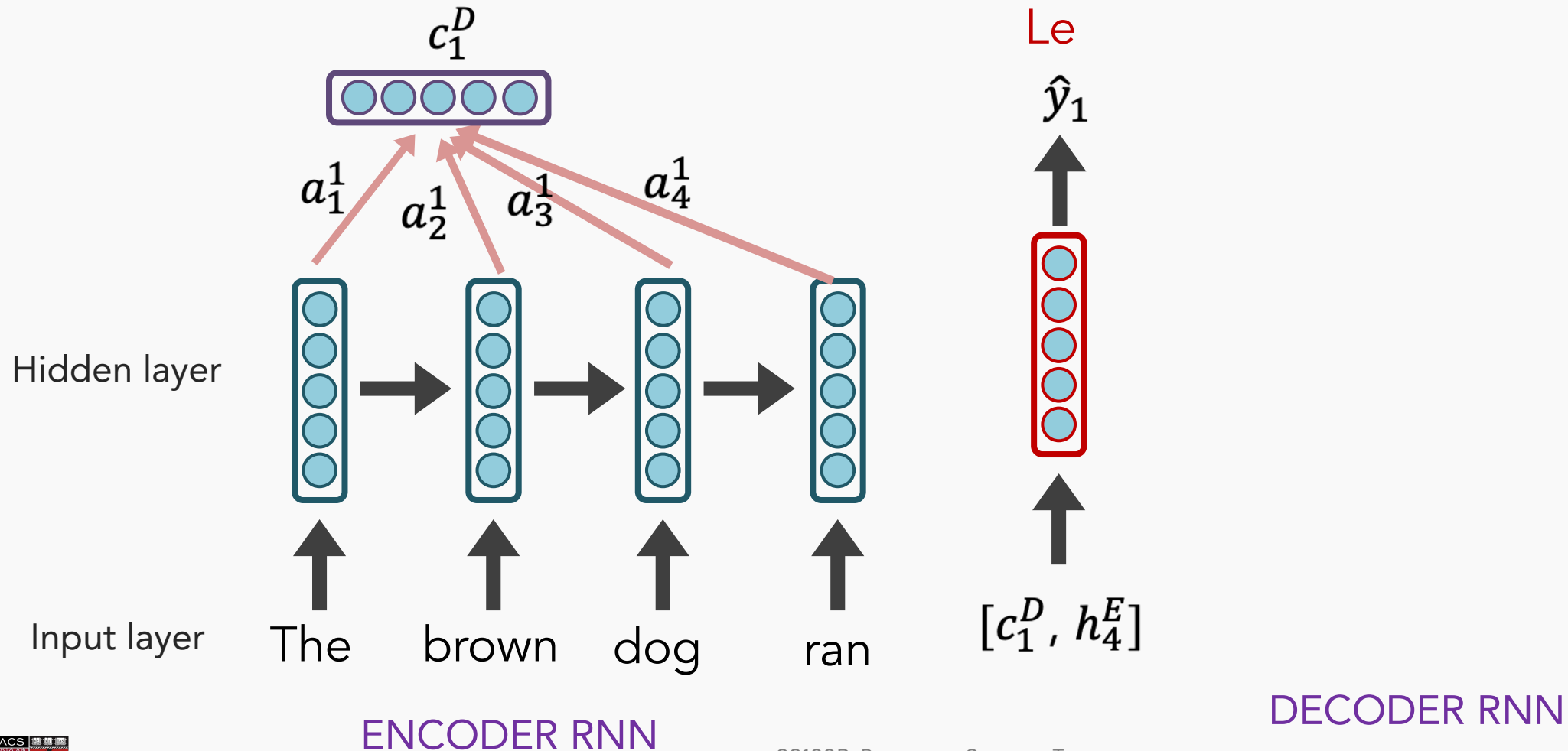
# Sequence-to-Sequence (seq2seq)

---

Instead, what if the decoder, at each step, pays **attention** to a *distribution* of all of the encoder's hidden states?

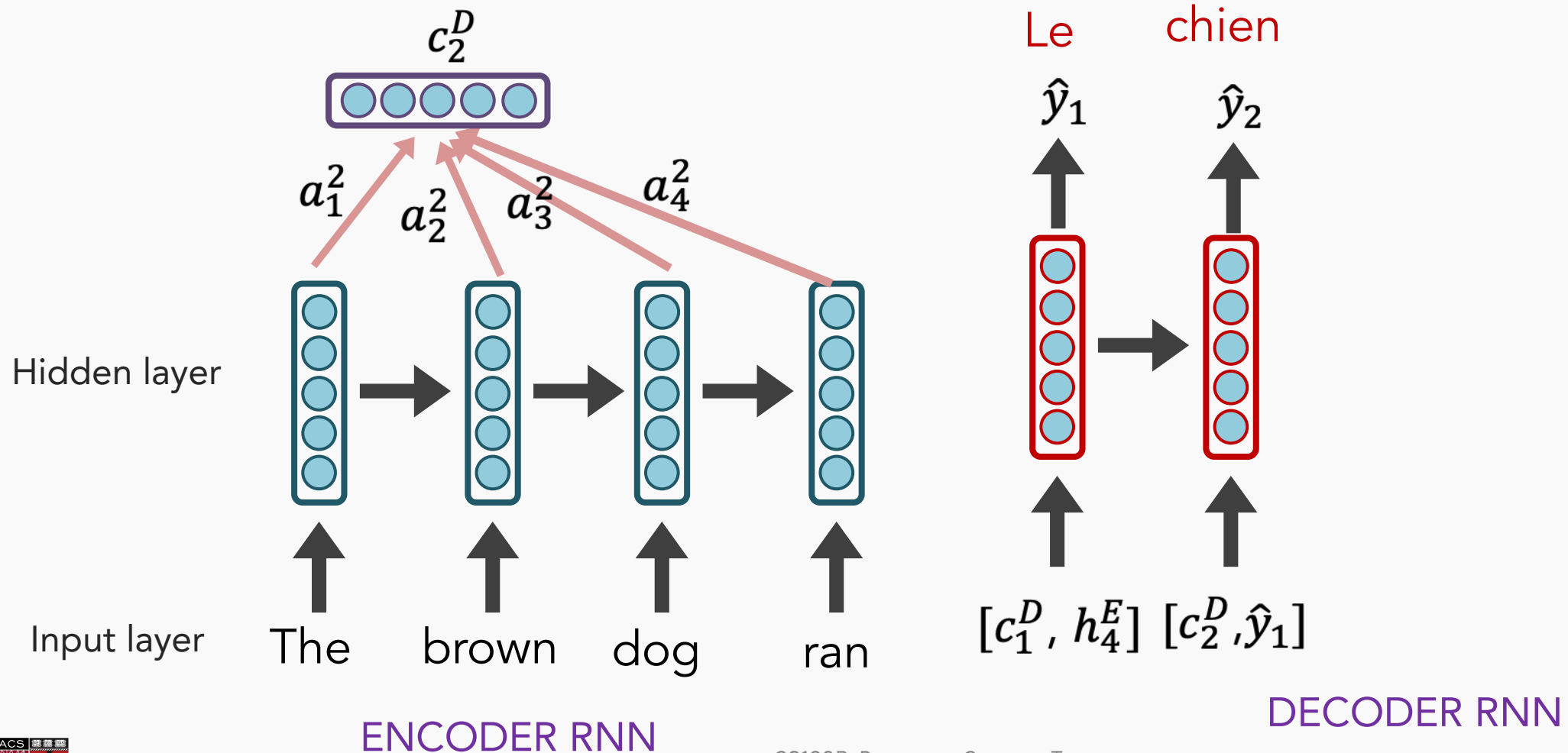
# seq2seq + Attention

**NOTE:** each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



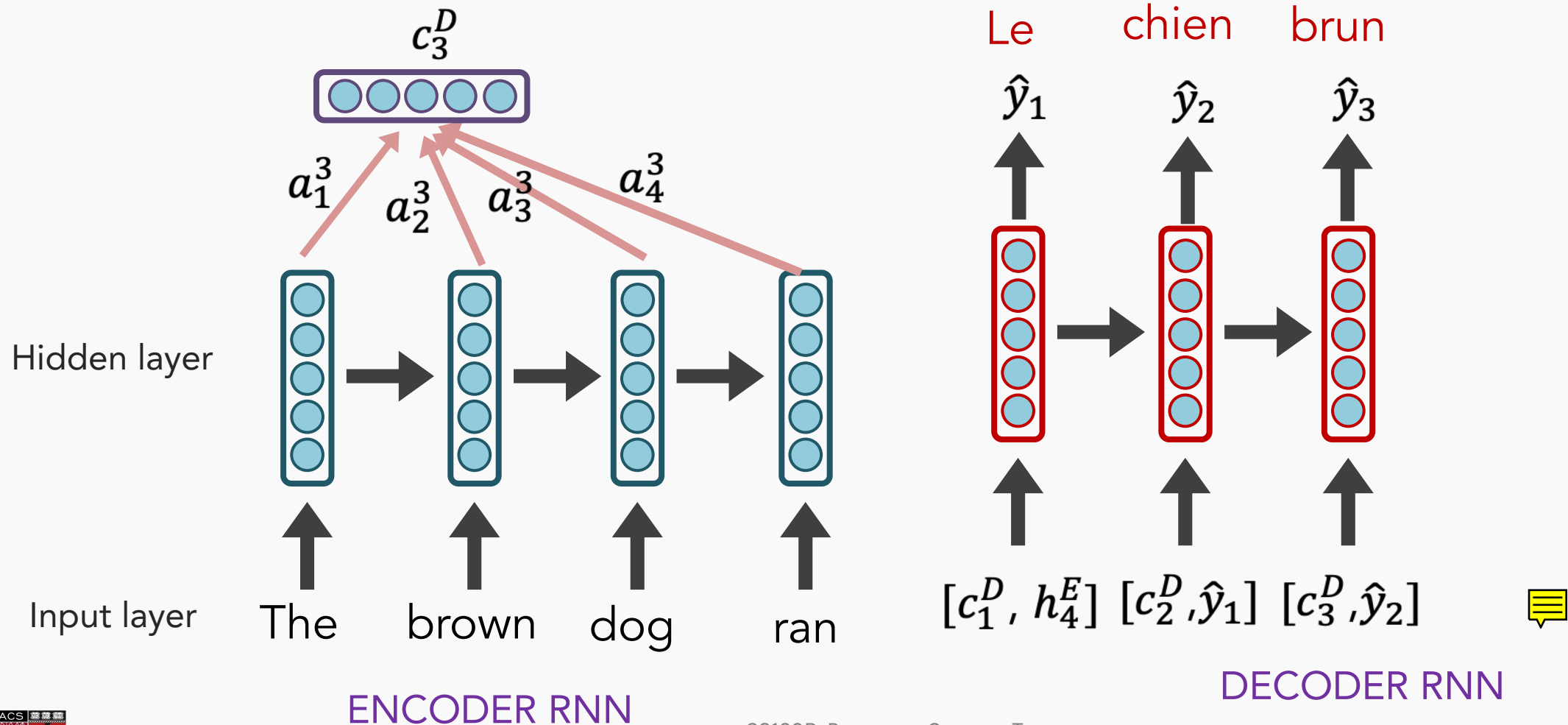
# seq2seq + Attention

**NOTE:** each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



# seq2seq + Attention

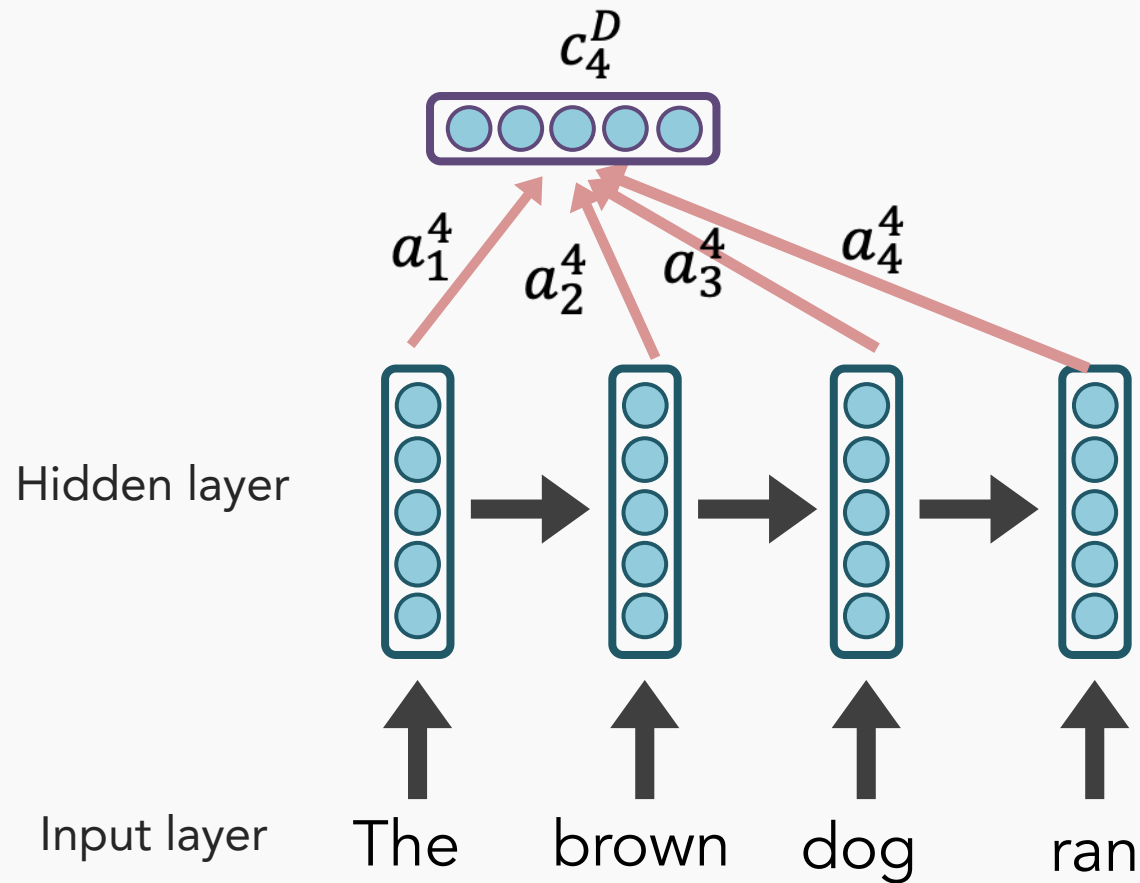
**NOTE:** each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



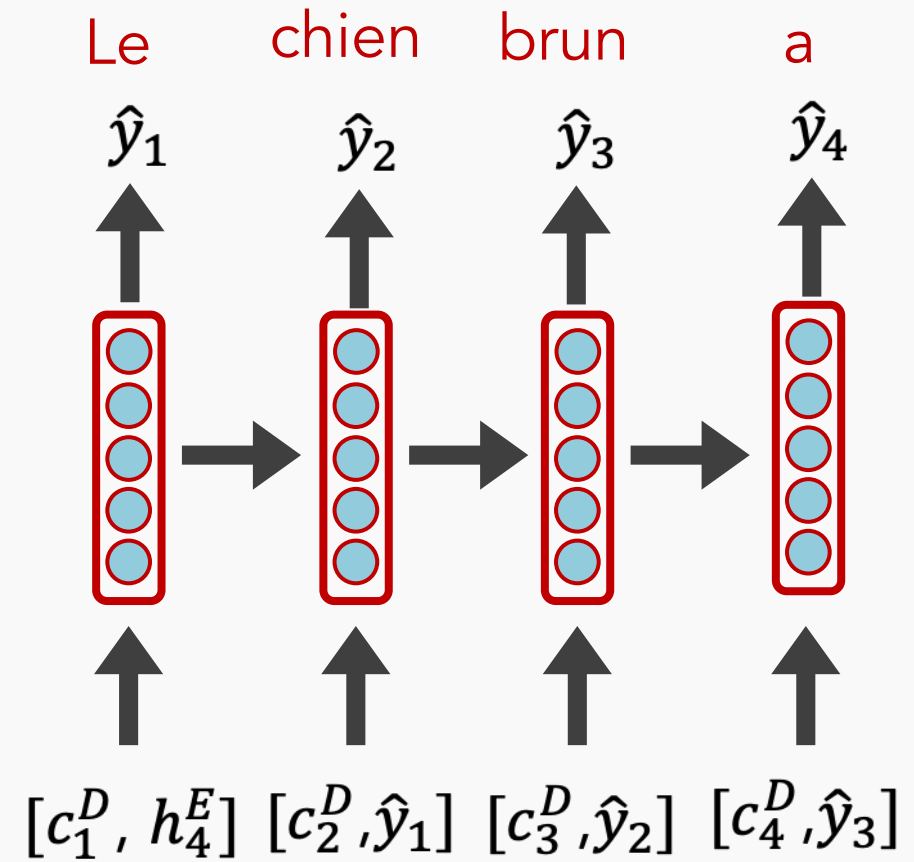


# seq2seq + Attention

**NOTE:** each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



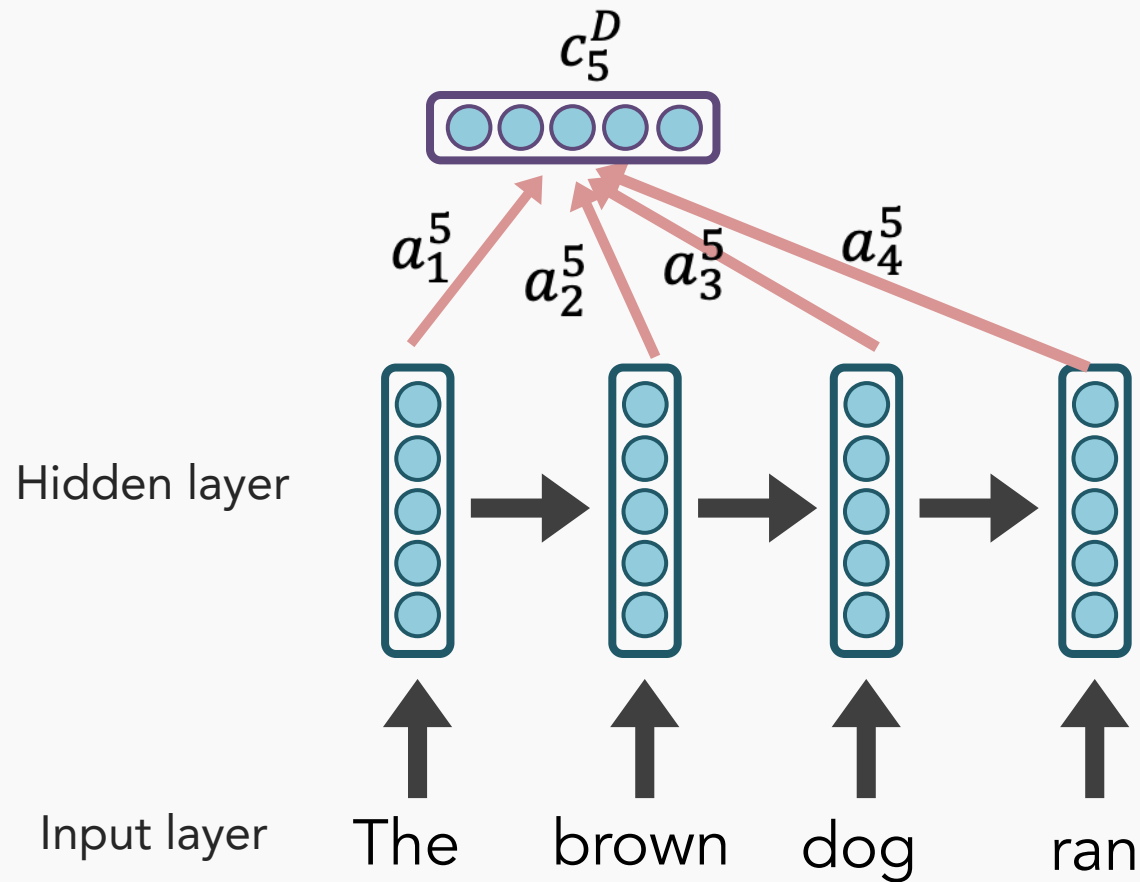
ENCODER RNN



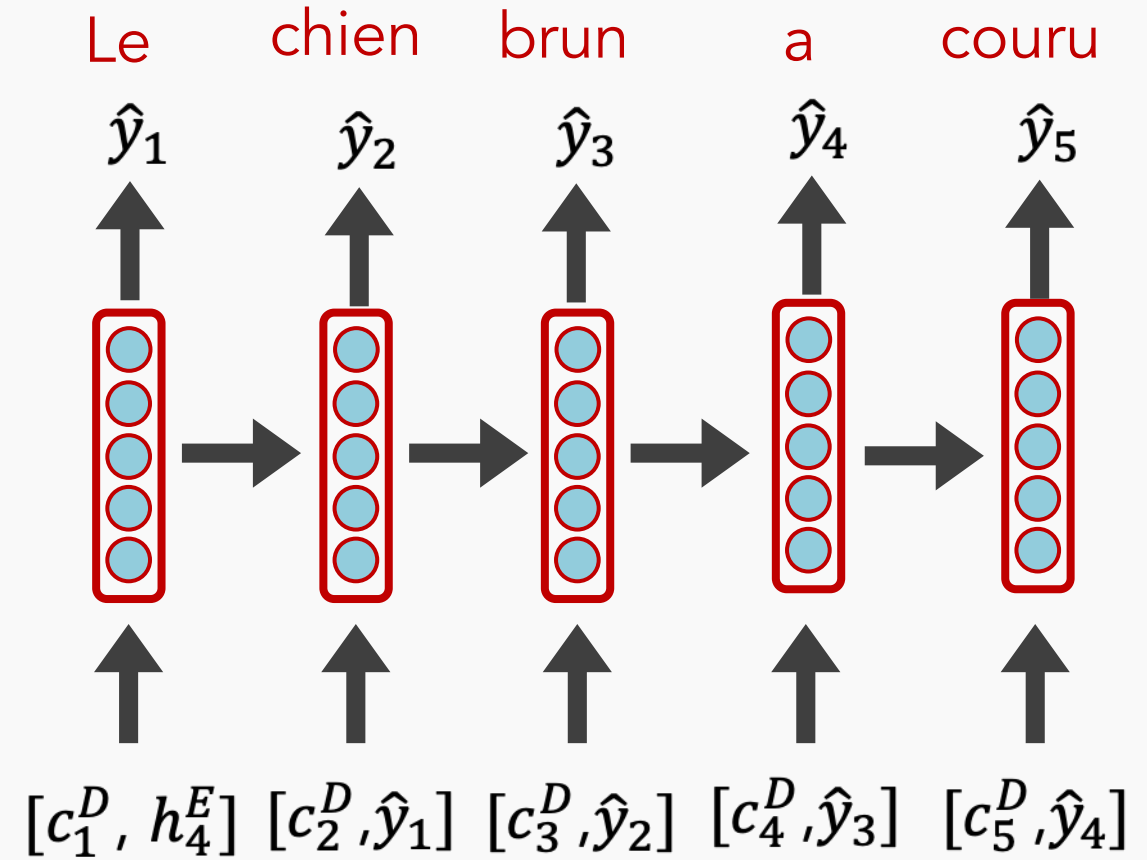
DECODER RNN

# seq2seq + Attention

**NOTE:** each attention weight  $a_i^j$  is based on the decoder's current hidden state, too.



ENCODER RNN

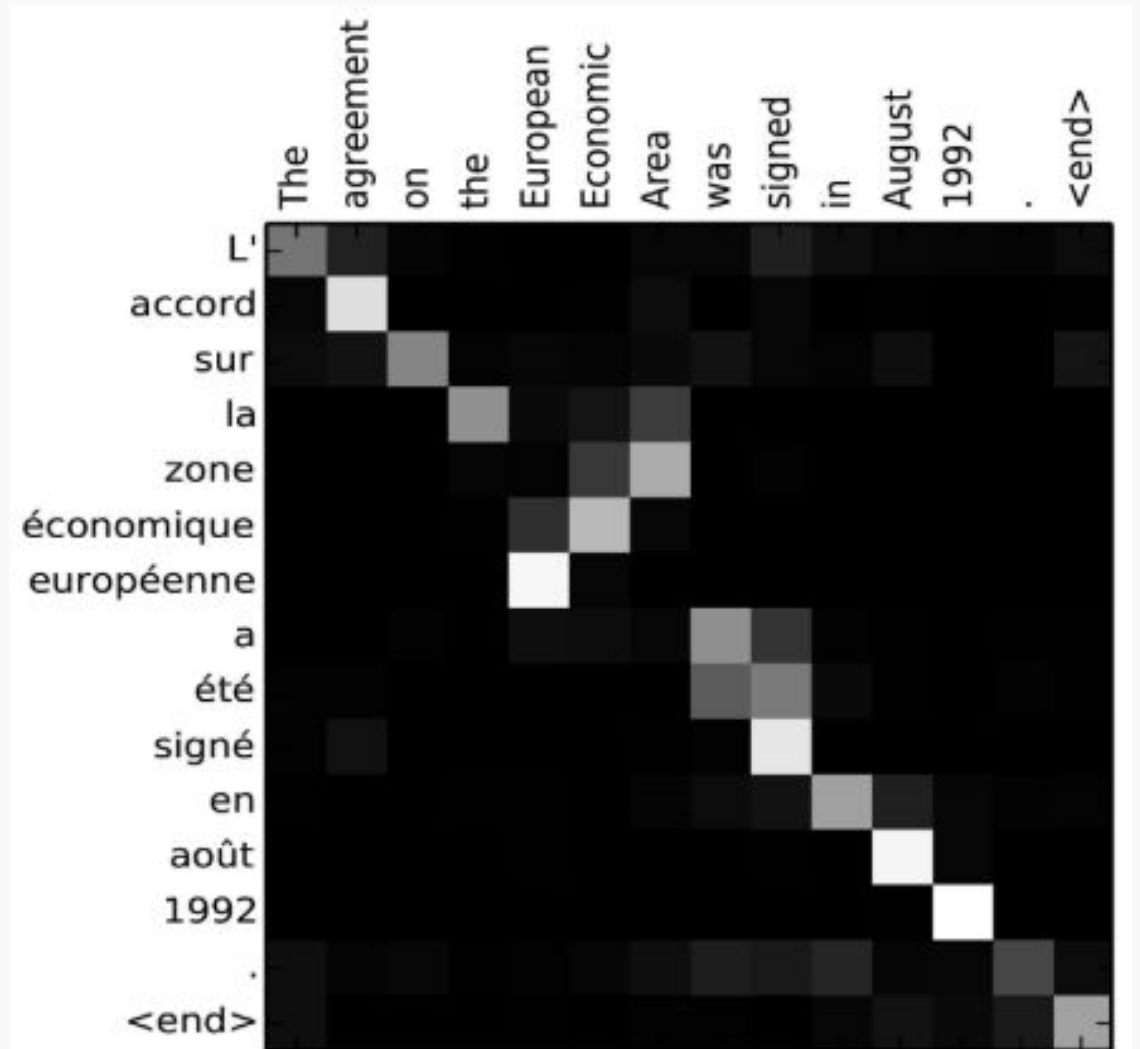


DECODER RNN

# seq2seq + Attention

## Attention:

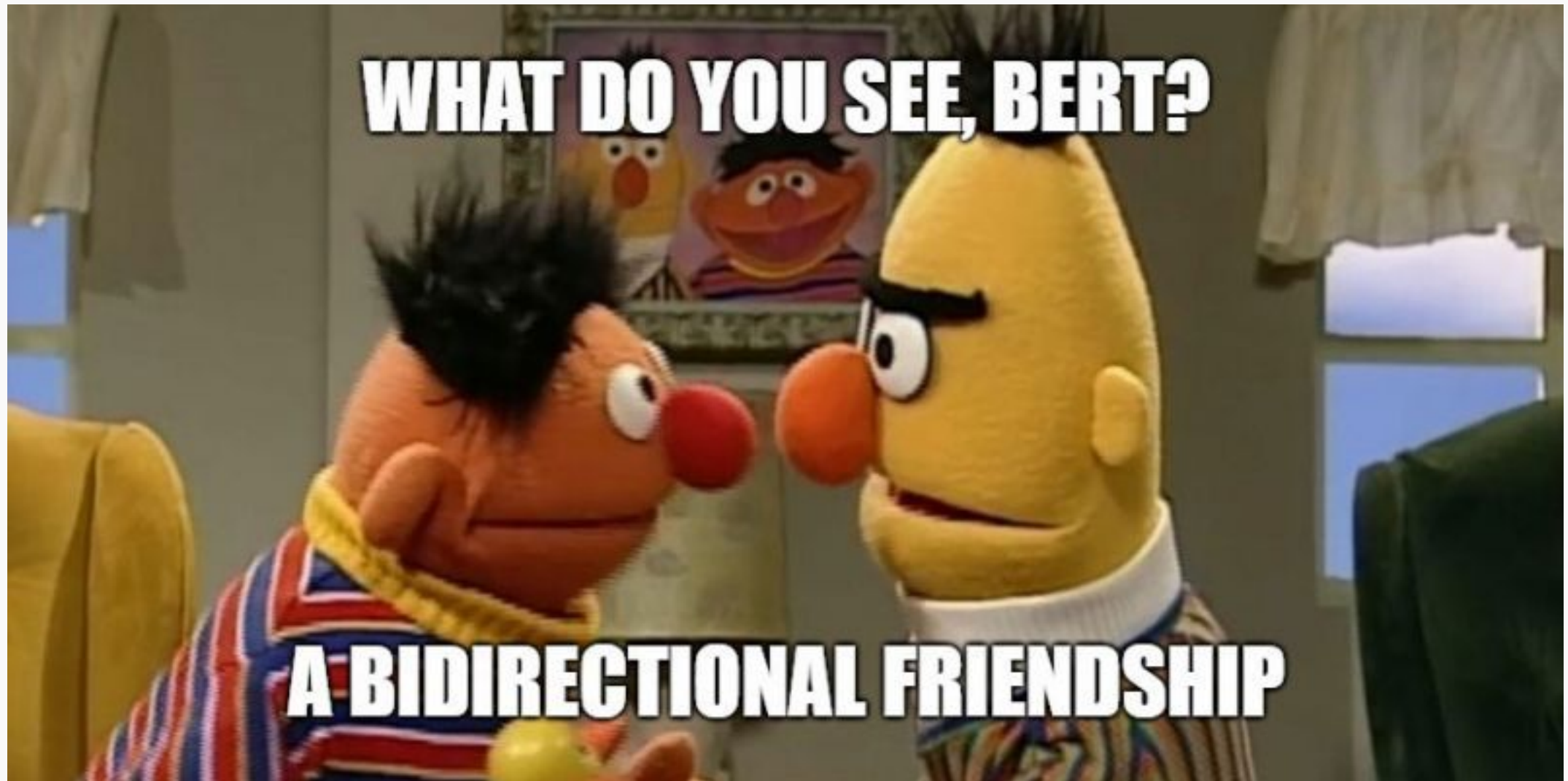
- greatly improves seq2seq results
- allows us to visualize the contribution each word gave during each step of the decoder



# Outline

---

- Seq2Seq +Attention
- **Transformers +BERT**
- Embeddings



# Self-Attention

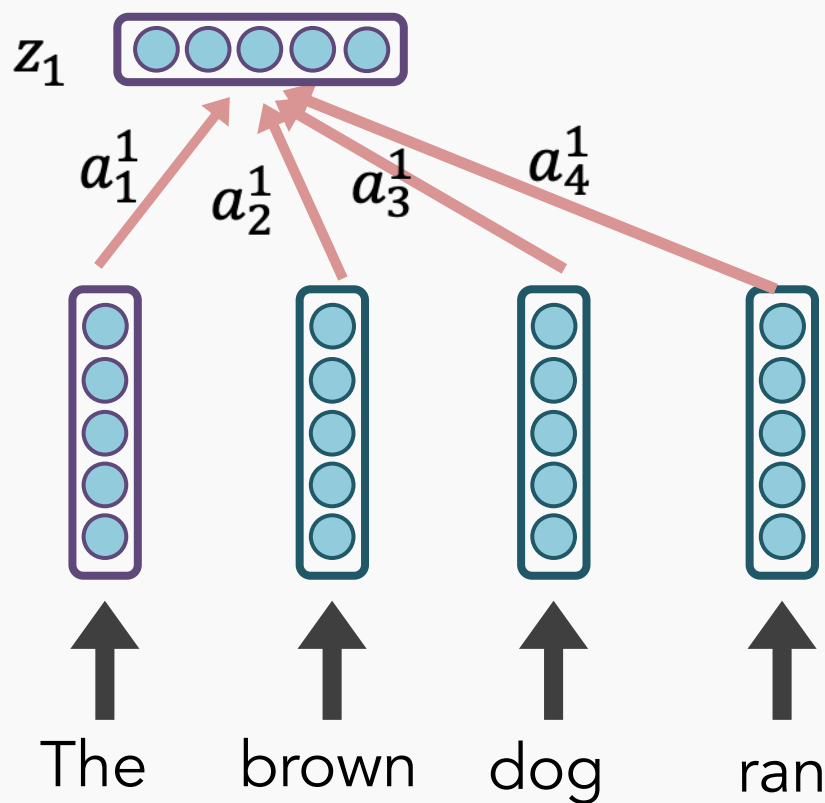
---

- Models direct relationships between all words in a given sequence (e.g., sentence)
- Does not concern a seq2seq (i.e., encoder-decoder RNN) framework
- Each word in a sequence can be transformed into an abstract **representation** (embedding) based on the weighted sums of the other words in the same sequence

# Self-Attention

Output  
representation

Input vectors



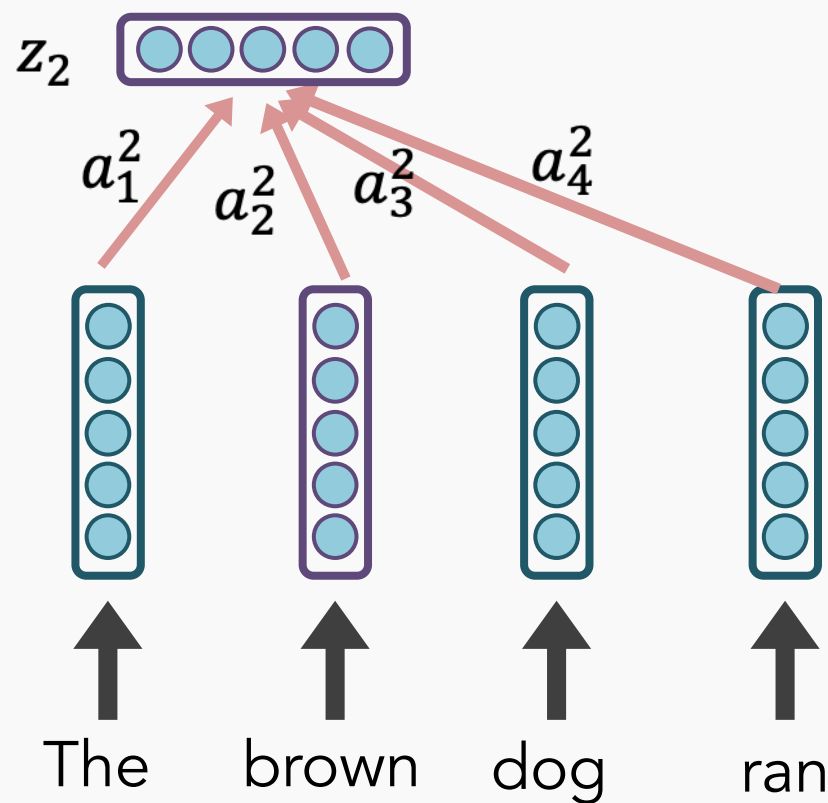
This is a large simplification.

The representations are created from using **Query**, **Key**, and **Value** vectors, produced from learned weight matrices during Training

# Self-Attention

Output  
representation

Input vectors



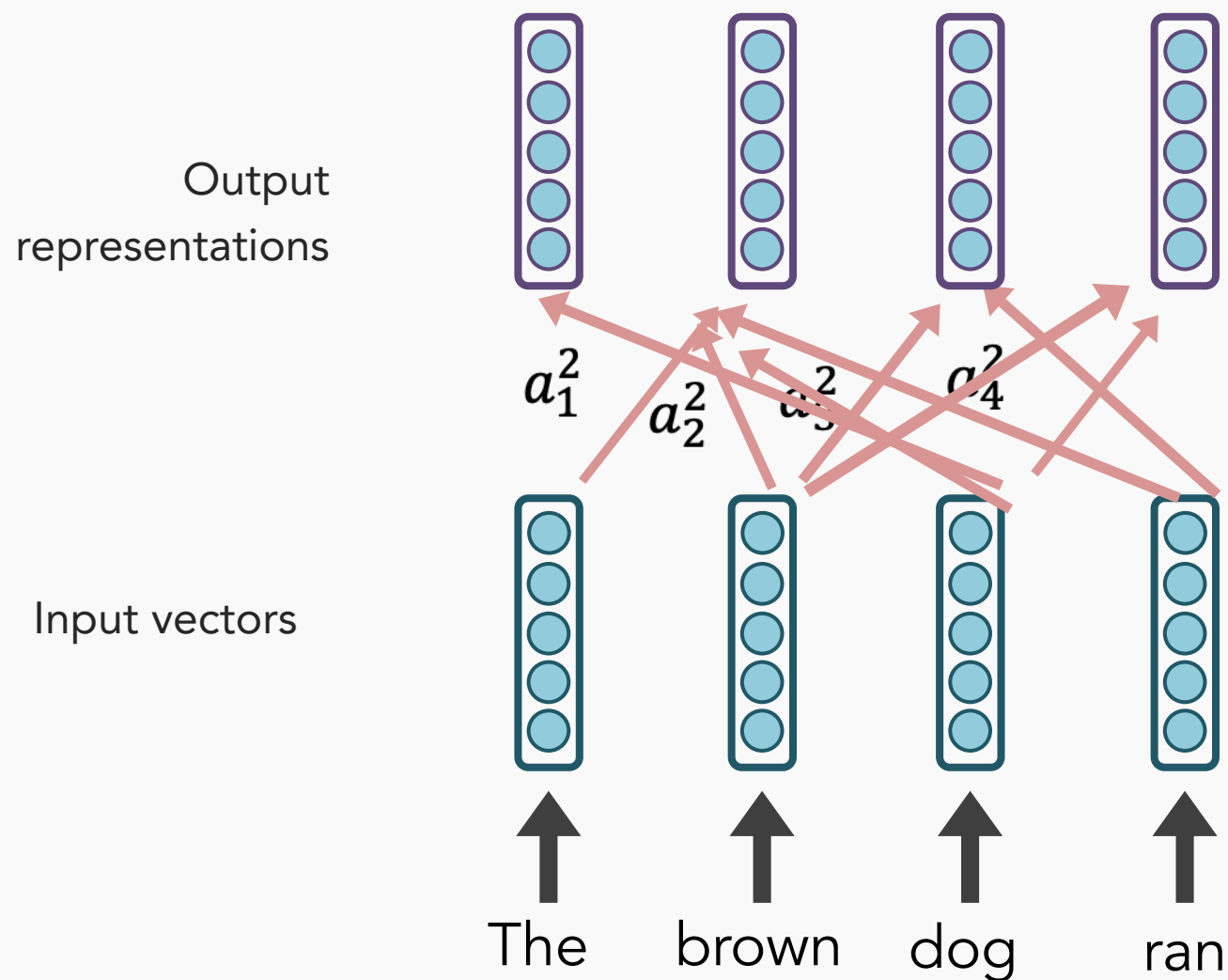
This is a large simplification.

The representations are created from using **Query**, **Key**, and **Value** vectors, produced from learned weight matrices during Training





# Self-Attention



This is a large simplification.

The representations are created from using **Query**, **Key**, and **Value** vectors, produced from learned weight matrices during Training

# Self-Attention

---

To recap:

- **Attention** determines which pieces of **sequence A** are most relevant w.r.t. **sequence B**
- **Self-attention** determines which pieces of **sequence A** are most relevant w.r.t. **sequence A**
- A **Transformer** combines both; it has an encoder-decoder component, yet also uses **self-attention** to refine each respective sequence's representation

# Self-Attention

---

- Transformers yield the best results for machine translation (seq2seq)
- Transformers handle long-range dependencies better than LSTMs
- BERT is an example of a Transformer

# BERT

---

- BERT is like the encoder portion of a Transformer
- Uses self-attention
- Uses bi-directional conditioning to perform language modelling
- Yet, it doesn't see its own words because it cleverly masks 15% of its words
- Fine-tunes on a sentence/entailment task
- BERT provides generalized contextual embeddings which can be fine-tuned toward other classification tasks (e.g., sentiment classification)

# Conclusion

---

- There has been significant progress in the past few years.
- Some of the complex models are incredible, but rely on having a lot of data and computational resources (e.g., Transformers)
- With all data science and machine learning, it's best to understand your data and your task very well, then clean the data, and start with a simple model (instead of jumping to the most complex model)

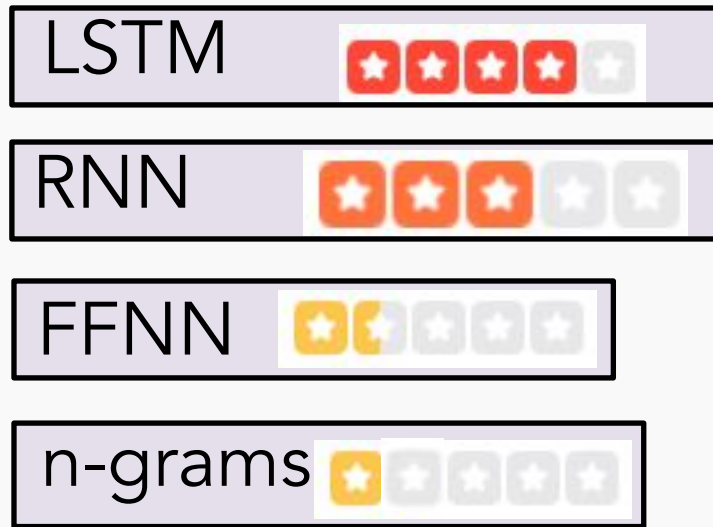
# Conclusion

---

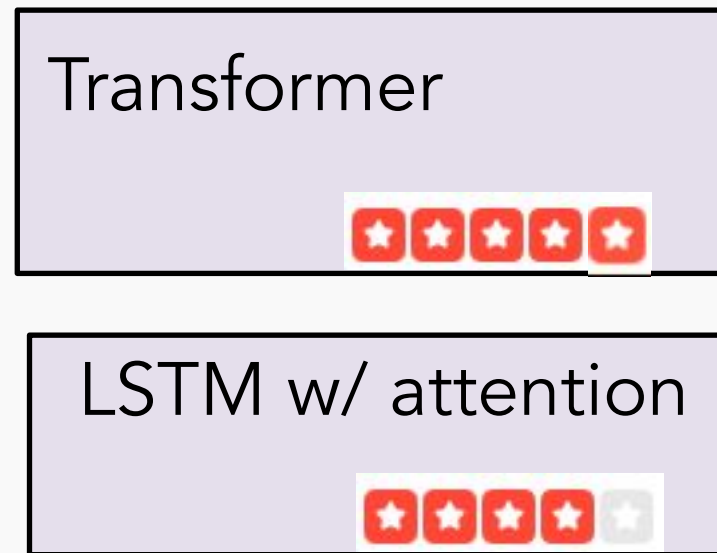
## Models

- **N-gram**: count statistics; **elementary** sequence modelling
- **FFNN**: fixed-length context window; basic sequence modelling
- **(Vanilla) RNN**: uses context; fair sequence modelling
- **LSTM**: great contextual usage; great sequence modelling
- **Seq2Seq**: maps 1 sequence to another
- **Attention**: determines which elements in **sequence A** pertain to **sequence B**
- **Self-Attention**: determines great representations for items in **a sequence**
- **Transformers**: learns excellent representation, via a **seq2seq** framework and **self-attention**

# Sequential Modelling



Sequence Modelling  
(1-to-1 mapping)



seq2seq

## Credit & further resources:

- Backprop: <http://cs231n.github.io/optimization-2/>
- Abigail See's lectures: <http://web.stanford.edu/class/cs224n/index.html>
- Illustrated BERT: <http://jalammar.github.io/illustrated-bert/>
- Andrew Ng (Attention): <https://www.youtube.com/watch?v=quoGRI-1l0A>
- Illustrated Transformer: <https://jalammar.github.io/illustrated-transformer/>
- Google (Transformers): <https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html>
- ELMo paper: <https://arxiv.org/pdf/1802.05365.pdf>



# Outline

---

- Seq2Seq +Attention
- Transformers +BERT
- **Embeddings**

# The basics idea

	Trait #1	Trait #2	Trait #3	Trait #4	Trait #5
Jay	-0.4	0.8	0.5	-0.2	0.3
Person #1	-0.3	0.2	0.3	-0.4	0.9
Person #2	-0.5	-0.4	-0.2	0.7	-0.1

- Observe a bunch of people
- **Infer** personality traits from them
- Vector of traits is called an **Embedding**
- Who is more similar? Jay and who?
- Use Cosine Similarity of the vectors

$\text{cosine\_similarity}(\text{Jay}, \text{Person \#1}) = 0.66$  ✓

$\text{cosine\_similarity}(\text{Jay}, \text{Person \#2}) = -0.37$

# Categorical Data

---

## *Example:*

**Rossmann Kaggle Competition:** Rossmann is a 3000 store European Drug Store Chain. The idea is to predict sales 6 weeks in advance.

Consider `store_id` as an example. This is a **categorical** predictor, i.e. values come from a finite set.

We usually **one-hot encode** this: a single store is a length 3000 bit-vector with one bit flipped on.

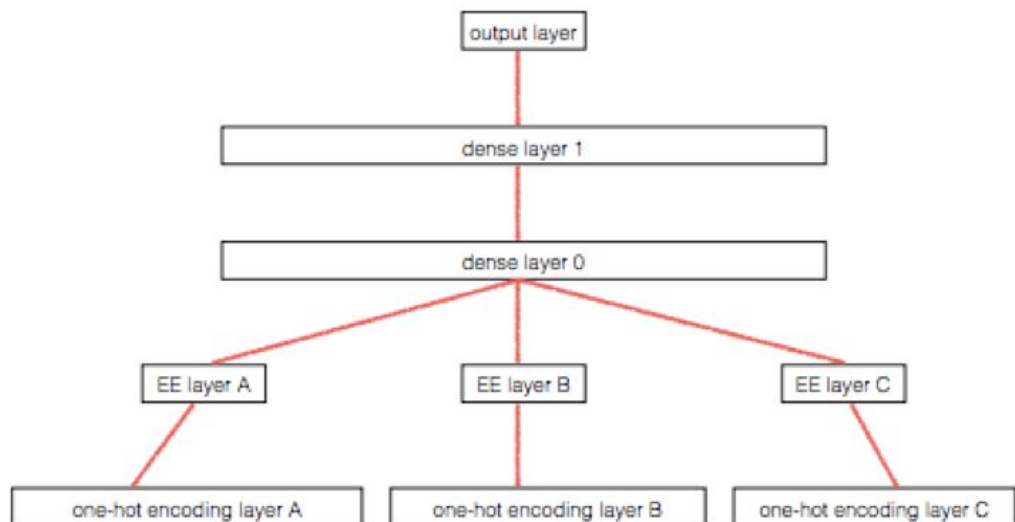
# Categorical Data

---

## What is the problem with this?

- The 3000 stores have commonalities, but the one-hot encoding does not represent this.
- Indeed the dot-product (cosine similarity) of any two 1-hot bitmaps must be 0.
- Would be useful to learn a lower-dimensional **embedding** for the purpose of sales prediction.
- These store "personalities" could then be used in other models (different from the model used to learn the embedding) for sales prediction.
- The embedding can be also used for other **tasks**, such as employee turnover prediction.

# Training an Embedding



- Normally you would do a linear or MLP regression with sales as the target, and both continuous and categorical features.
- The game is to replace the 1-hot encoded categorical features by "**lower-width**" embedding features, for each categorical predictor.
- This is equivalent to considering a neural network with the output of an additional **Embedding Layer** concatenated in.
- The Embedding layer is simply a linear regression.

# Training an Embedding (cont)

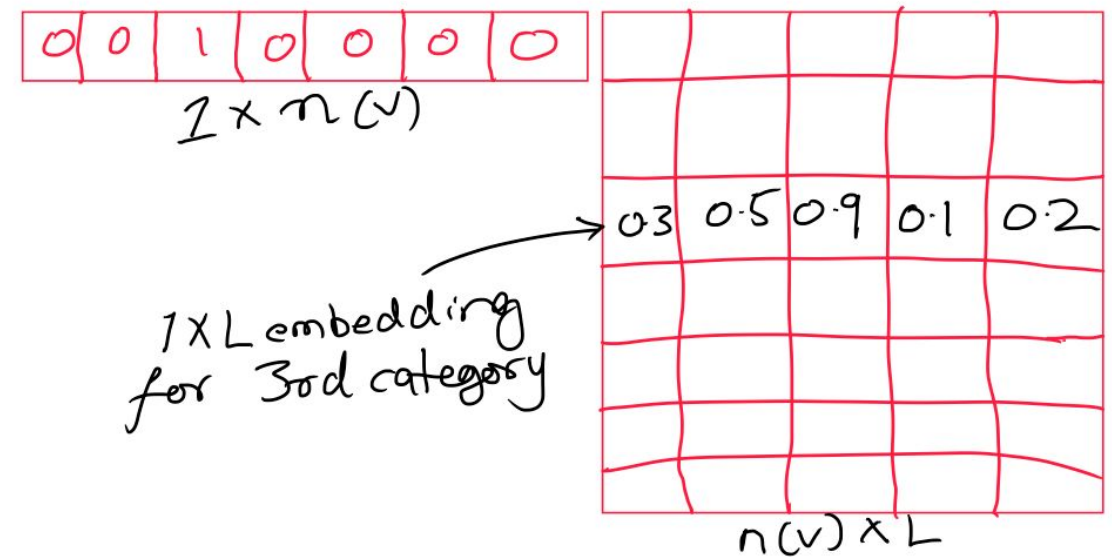
A 1-hot vector for a categorical variable with cardinality can be written using the Kronecker Delta symbol as:

$$v_k = \delta_{jk}, j \in \{1, \dots, N(v)\}$$

Then an embedding of width (dimension)  $L$  is just a  $N(v) \times L$  matrix of weights  $W_{ij}$  such that multiplying the  $k^{\text{th}}$  1-hot vector by this weight matrix picks out the  $k^{\text{th}}$  row of weights (see right).

But how do we find these weights?

We fit for them with the rest of the weights in the MLP!



# Training an Embedding (cont)

```
def build_keras_model():
    input_cat = []
    output_embeddings = []
    for k in cat_vars+ncols_cat: #categoricals plus NA booleans
        input_1d = Input(shape=(1,))
        output_1d = Embedding(input_cardinality[k], embedding_cardinality[k], name='{}_embedding'.format(k))(input_1d)
        output = Reshape(target_shape=(embedding_cardinality[k],))(output_1d)
        input_cat.append(input_1d)
        output_embeddings.append(output)

    main_input = Input(shape=(len(cont_vars),), name='main_input')
    output_model = Concatenate()([main_input, *output_embeddings])
    output_model = Dense(1000, kernel_initializer="uniform")(output_model)
    output_model = Activation('relu')(output_model)
    output_model = Dense(500, kernel_initializer="uniform")(output_model)
    output_model = Activation('relu')(output_model)
    output_model = Dense(1)(output_model)

    kmodel = KerasModel(
        inputs=[*input_cat, main_input],
        outputs=output_model
    )
    kmodel.compile(loss='mean_squared_error', optimizer='adam')
    return kmodel

def fitmodel(kmodel, Xtr, ytr, Xval, yval, epochs, bs):
    h = kmodel.fit(Xtr, ytr, validation_data=(Xval, yval),
                   epochs=epochs, batch_size=bs)

    return h
```

# Embedding is just a linear regression

---

So why are we giving it another name?

- it is usually to lower the dimensional space
- traditionally we have done linear dimensional reduction through PCA and truncation, but sparsity can throw a spanner into the works.
- we train the weights of the embedding regression using gradient descent (or stochastic gradient descent), along with the weights of the downstream task (here predicting the sales 6 weeks in advanced).
- the embedding can be used for alternate tasks, such as finding the similarity of users.

See how Spotify does all this.



```
def embedding_input(emb_name, n_items, n_fact=20, l2regularizer=1e-4):
    inp = Input(shape=(1,), dtype='int64', name=emb_name)
    return inp, Embedding(n_items, n_fact, input_length=1, embeddings_regularizer=l2(l2regularizer))(inp)
```

```
usr_inp, usr_emb = embedding_input('user_in', n_users, n_fact=50, l2regularizer=1e-4)
mov_inp, mov_emb = embedding_input('movie_in', n_movies, n_fact=50, l2regularizer=1e-4)
```

```
def create_bias(inp, n_items):
    x = Embedding(n_items, 1, input_length=1)(inp)
    return Flatten()(x)
```

```
usr_bias = create_bias(usr_inp, n_users)
mov_bias = create_bias(mov_inp, n_movies)
```

```
def build_dp_bias_recommender(u_in, m_in, u_emb, m_emb, u_bias, m_bias):
    x = dot([u_emb, m_emb], axes=(2,2))
    x = Flatten()(x)
    x = add([x, u_bias])
    x = add([x, m_bias])
    bias_model = Model([u_in, m_in], x)
    bias_model.compile(Adam(0.001), loss='mse')
    return bias_model
```

```
bias_model = build_dp_bias_recommender(usr_inp, mov_inp, usr_emb, mov_emb, usr_bias, mov_bias)
```

# Word Embeddings

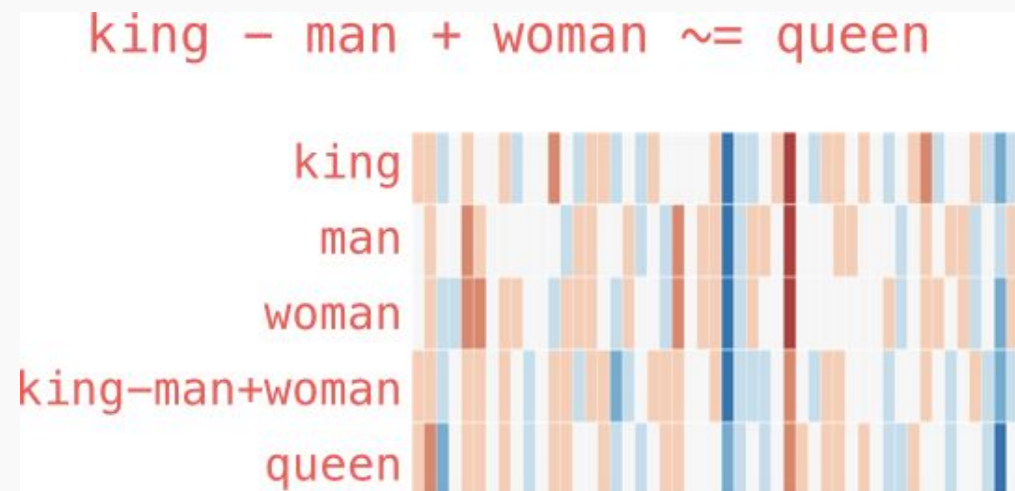
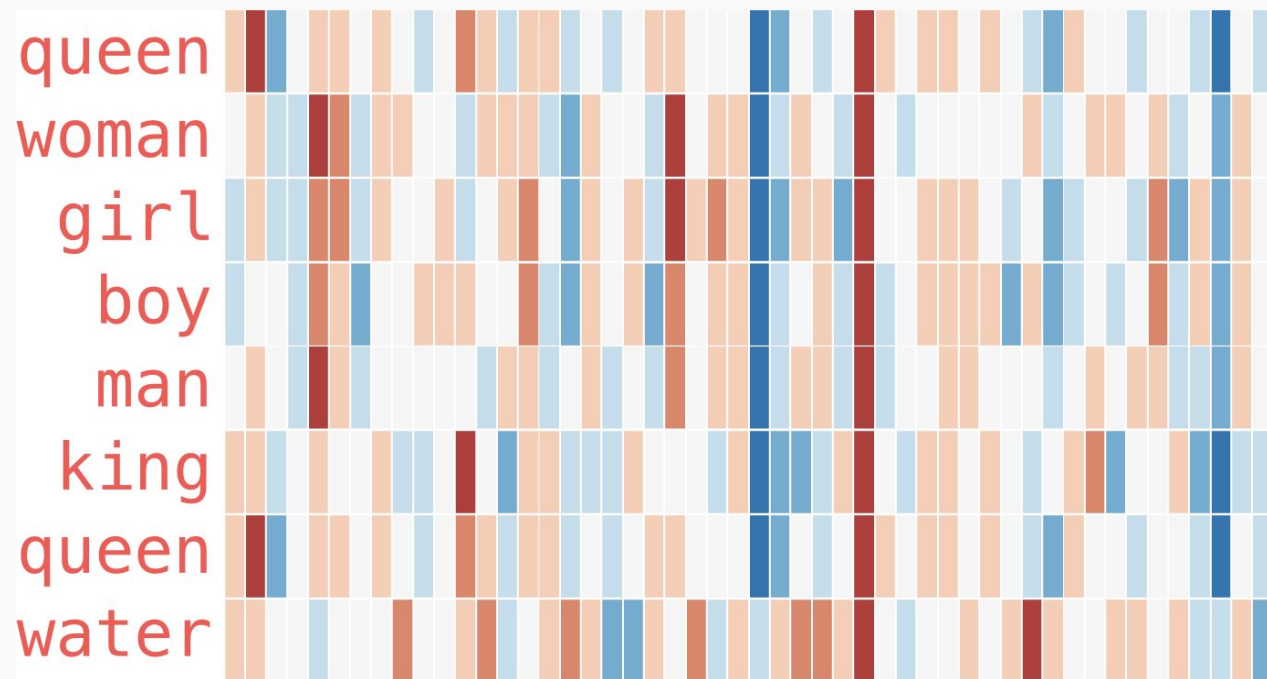
---

- the vocabulary  $V$  of a corpus (large swath of text) can have 10,000 and maybe more words.
- a 1-hot encoding is huge, moreover, similarities between words cannot be established.
- we map words to a smaller dimensional latent space of size  $L$  by considering some downstream task to train on.
- we hope that the embeddings learned are useful for other tasks.



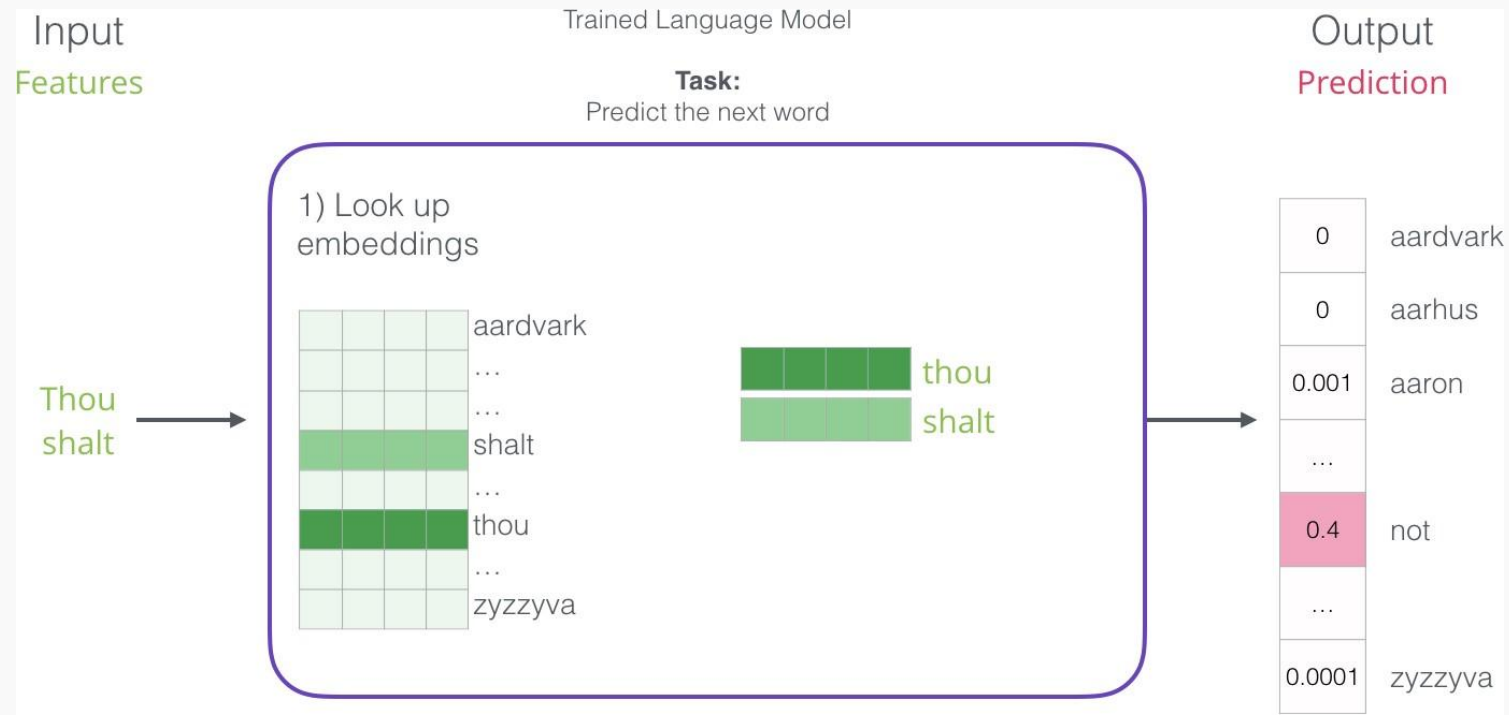
# Obligatory example

See man→boy as woman→girl, similarities of king and queen, for e.g. These are lower dimensional GloVe embedding vector.



# How do we train word embeddings?

- We need to choose a downstream task.
- We could choose **Language Modeling**: predict the next word.
- We'll start with random "weights" for the embeddings and other parameters and start learning.
- A trained model+embeddings would look like this:



# How do we set up a training set?

Thou shalt not make **a machine** in the likeness of a human mind

Sliding window across running text

thou	shalt	not	make	a	machine	in	the	...
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	
thou	shalt	not	make	a	machine	in	the	

Dataset

input 1	input 2	output
thou	shalt	not
shalt	not	make
not	make	a
make	a	machine
a	machine	in

Why not look both ways? This leads to the Skip-Gram and CBOW architectures..

# SKIP-GRAM: Predict Surrounding Words

Choose a window size (here 4) and construct a dataset by sliding a window across.

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

input word	target word
not	thou
not	shalt
not	make
not	a

Thou shalt not make a machine in the likeness of a human mind

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

thou	shalt	not	make	a	machine	in	the	...
------	-------	-----	------	---	---------	----	-----	-----

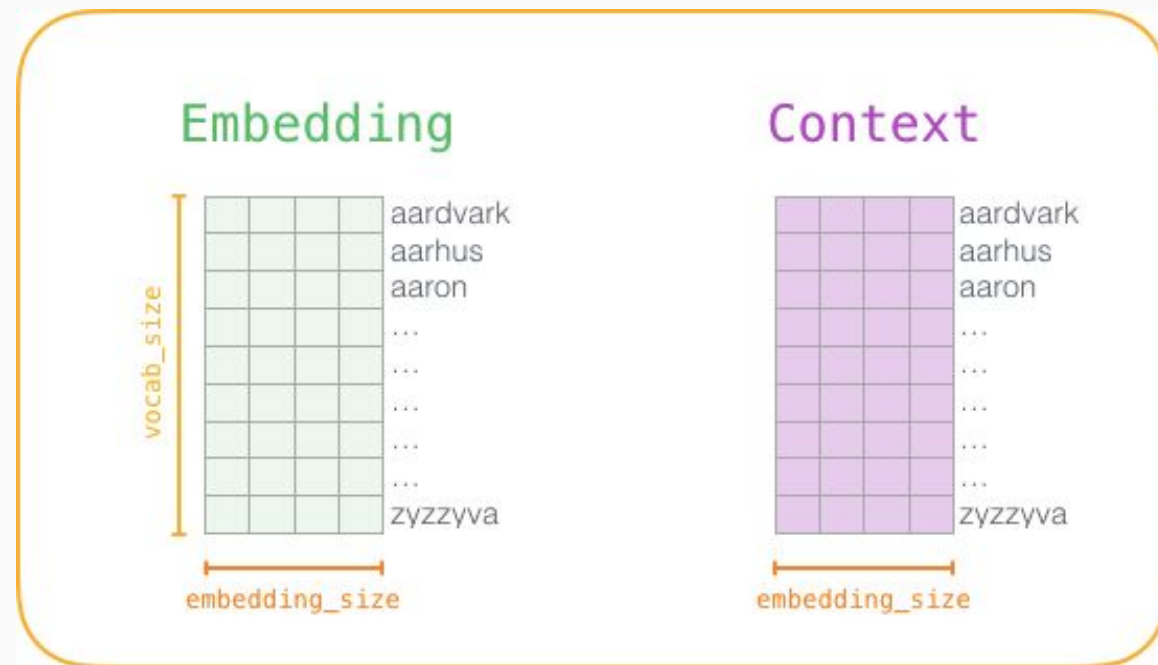
input word	target word
not	thou
not	shalt
not	make
not	a
make	shalt
make	not
make	a
make	machine



# SKIP-GRAM: Details

We assume that, Naive Bayes style, the joint probability of all **context** words in a window conditioned on the central word ( $w_c$ ) is the product of the individual conditional probabilities:

$$P(\{w_o\} \mid w_c) = \prod_{i \in \text{window}} p(w_{oi} \mid w_c)$$



Now assume that each word is represented as 2 embeddings, an **input** embedding ( $v_c$ ) when we talk about the central word and a context embedding ( $u_o$ ) when we talk about the surrounding window.

The probability of an output word, given a central word, is assumed to be given by a softmax of the dot product of the embeddings.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)},$$

input word	output word	target	input • output
not 	thou 	1	0.2

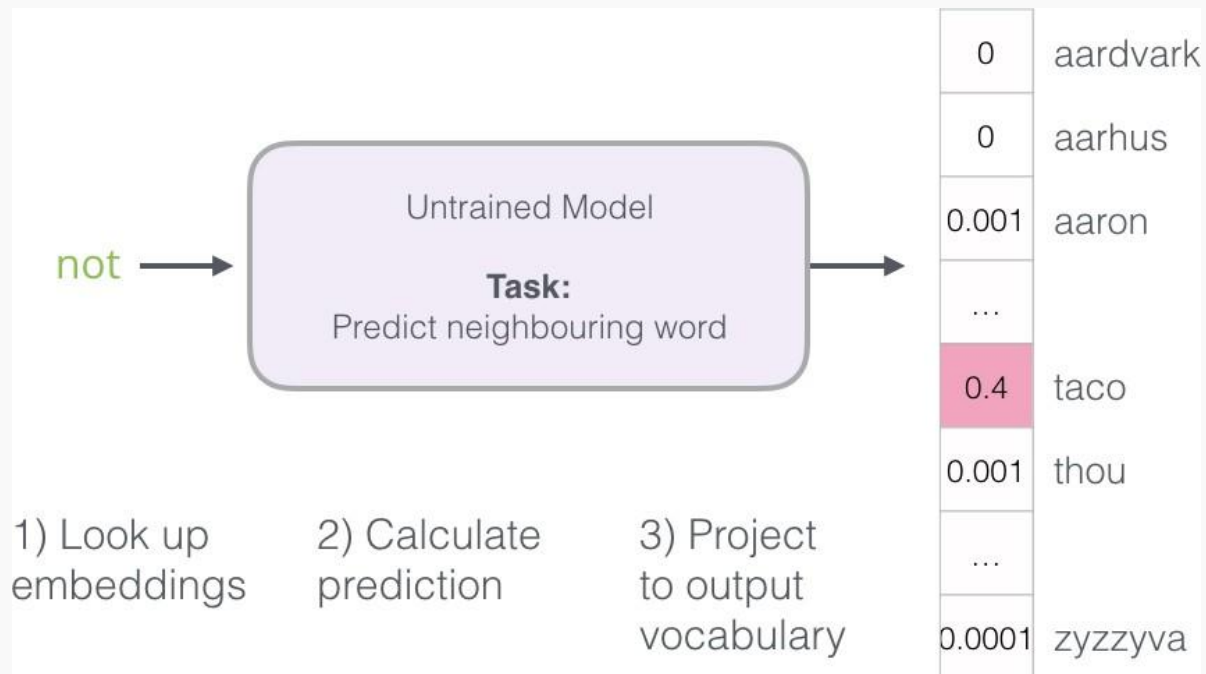
Then, assuming a text sequence of length  $T$  and window size  $m$ , the likelihood function is:

$$\mathcal{L} = \prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)}).$$

We'll use the Negative Log Likelihood as loss (NLL)



# Prediction



With random initial weights, we make a prediction for surrounding words, and calculate the NLL for the prediction. We then backpropagate the NLL's gradients to find new weights and repeat

Consider two sentences: "*I am running.*" and "*I am writing.*". "I" and "am" targets will backprop to same **input** embedding and so, after some training, "writing" and "running" will be highly correlated. Appropriate correlations will emerge as corpus size increases.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)},$$

- in the forward mode, the calculation of softmax requires a sum over the entire vocabulary
- in the backward mode, the gradients need this sum too. For example:

$$\frac{\partial \log P(w_o \mid w_c)}{\partial \mathbf{v}_c} = \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c) \mathbf{u}_j.$$

For large vocabularies, this is very expensive!

# Changing Tasks

Change Task from



To:



Changing from predicting neighbors to "*are we neighbors?*" changes model from neural net to logistic regression.

# Changing Tasks (cont)

We'll now thus choose  $P(D = 1 | w_c, w_o) = \sigma(u_o^T v_c)$  and will maximize the likelihood:

$$\mathcal{L} = \prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}).$$

But the response variable in the dataset changes to all 1's and a trivial classifier always returning 1 will give the best score. Not good (this is equivalent to all embeddings being equal and infinite)!

# Negative Sampling

input word	target word		input word	output word	target
not	thou		not	thou	1
not	shalt		not	shalt	1
not	make		not	make	1
not	a		not	a	1
make	shalt		make	shalt	1
make	not		make	not	1
make	a		make	a	1
make	machine		make	machine	1

To fix we randomly choose words from our vocabulary and label them with 0.

input word	output word	target		Word	Count	Probability
not	thou	1		aardvark		
not	aaron	0		aarhus		
not	taco	0		aaron		
not	shalt	1		taco		
				thou		
				zyzzyva		
not	make	1				

Pick randomly from vocabulary (random sampling)

# Likelihood model

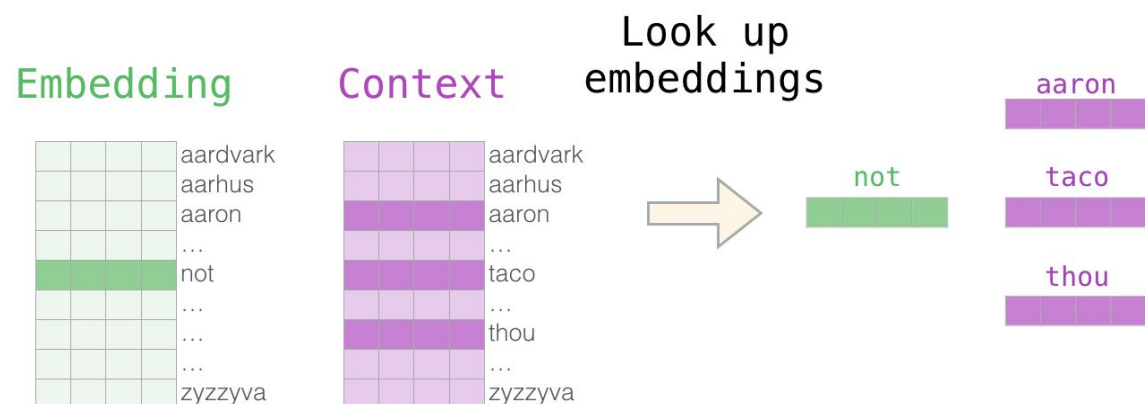
We go back to the old likelihood:  $\mathcal{L} = \prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)}).$

But now, the probability is approximated using negative sampling as:

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1, w_k \sim P(w)}^K P(D = 0 \mid w^{(t)}, w_k).$$

The NLL now has a sum over a  $K$ --sized window, rather than the full vocabulary.

# Training the model



- The negative sampling probabilities are now sigmoids subtracted from 1, whereas the positives are simply sigmoids.
- We now compute the loss, and repeat over training examples in our batch.
- And backpropagate to obtain gradients and change the embeddings and weights some, for each batch, in each epoch

input word	output word	target	input • output	sigmoid()
not	thou	1	0.2	0.55
not	aaron	0	-1.11	0.25
not	taco	0	0.74	0.68

```

def make_model(vector_dim, vocab_size, learn_rate):
    stddev = 1.0 / vector_dim
    initializer = RandomNormal(mean=0.0, stddev=stddev, seed=None)

    word_input = Input(shape=(1,), name="word_input")
    word = Embedding(input_dim=vocab_size, output_dim=vector_dim, input_length=1,
                     name="word_embedding", embeddings_initializer=initializer)(word_input)

    context_input = Input(shape=(1,), name="context_input")
    context = Embedding(input_dim=vocab_size, output_dim=vector_dim, input_length=1,
                       name="context_embedding", embeddings_initializer=initializer)(context_input)

    merged = dot([word, context], axes=2, normalize=False, name="dot")
    merged = Flatten()(merged)
    output = Dense(1, activation='sigmoid', name="output")(merged)

    optimizer = TFOptimizer(tf.train.AdagradOptimizer(learn_rate))
    model = Model(inputs=[word_input, context_input], outputs=output)
    model.compile(loss="binary_crossentropy", optimizer=optimizer)
    self.model = model

```



```
def train(model, sequence, window_size, negative_samples, batch_size):
    """ Trains the word2vec model """

    # in order to balance out more negative samples than positive
    negative_weight = 1.0 / negative_samples
    class_weight = {1: 1.0, 0: negative_weight}
    sequence_length = len(sequence)
    approx_steps_per_epoch = (sequence_length * (
        window_size * 2.0) + sequence_length * negative_samples) / batch_size
    batch_iterator = skip_gram.batch_iterator(sequence, window_size, negative_samples, batch_size)

    model.fit_generator(batch_iterator,
                        steps_per_epoch=approx_steps_per_epoch,
                        epochs=epochs,
                        verbose=verbose,
                        class_weight=class_weight,
                        max_queue_size=100)
```

# The result

---

- We discard the Context matrix, and **save the embeddings matrix**.
- We can use the embeddings matrix for our next task (perhaps a sentiment classifier).
- We could have trained embeddings along with that particular task to make the embeddings sentiment specific. There is always a tension between domain/task specific embeddings and generic ones.
- This tension is usually resolved in favor of using generic embeddings since task specific datasets seem to be smaller
- We can still unfreeze pre-trained embedding layers to modify them for domain specific tasks via transfer learning.

# Usage of word2vec

---

- the pre-trained word2vec and other embeddings (such as GloVe) are used everywhere in NLP today.
- the ideas have been used elsewhere as well. **AirBnB** and **Anghami** model sequences of listings and songs using word2vec like techniques.
- **Alibaba** and **Facebook** use word2vec and graph embeddings for recommendations and social network analysis.