

## Purpose of this notebook

This notebook is meant to serve as a *quick reference* of key concepts/notations from the Intro course.

# Notation

To ensure that everyone is up to speed on notation, let's review

- [the notation \(ML Notation.ipynb\)](#) that we used in the "Classical Machine Learning" part of the intro course.
- [additional notation \(Intro to Neural Networks.ipynb\)](#) used in the "Deep Learning" part of the intro course

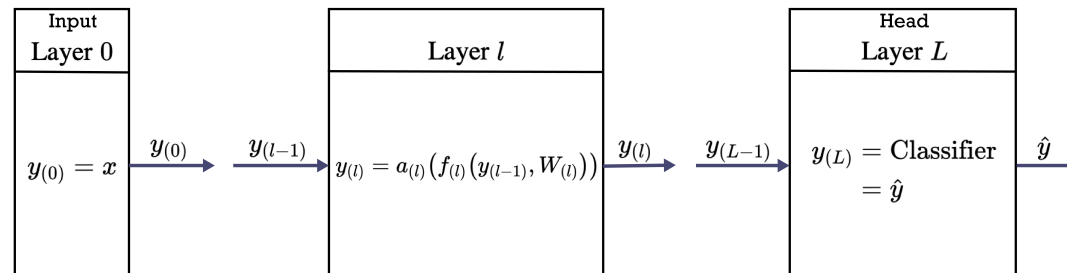
# Representations

A path through a Neural Network can be viewed as a sequence of representation transformations

- transforming *raw features*  $\mathbf{y}_{(0)} = \mathbf{x}$
- into *synthetic features*  $\mathbf{y}_{(l)}$ 
  - varying with layer  $1 \leq l \leq (L - 1)$
- of increasing abstraction

Thus, the output anywhere along the path is an *alternate representation* of the input

## Path through a Neural Network





Shallow features are less abstract: "syntax", "surface"

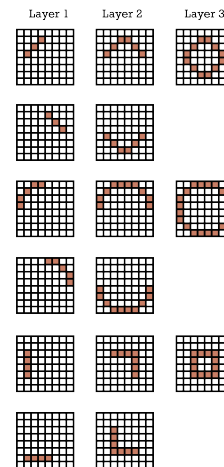
Deeper features are more abstract: "semantics", "concepts"

- We may even interpret the features as "pattern matching" regions or concepts in the raw feature space.

For example, in a CNN

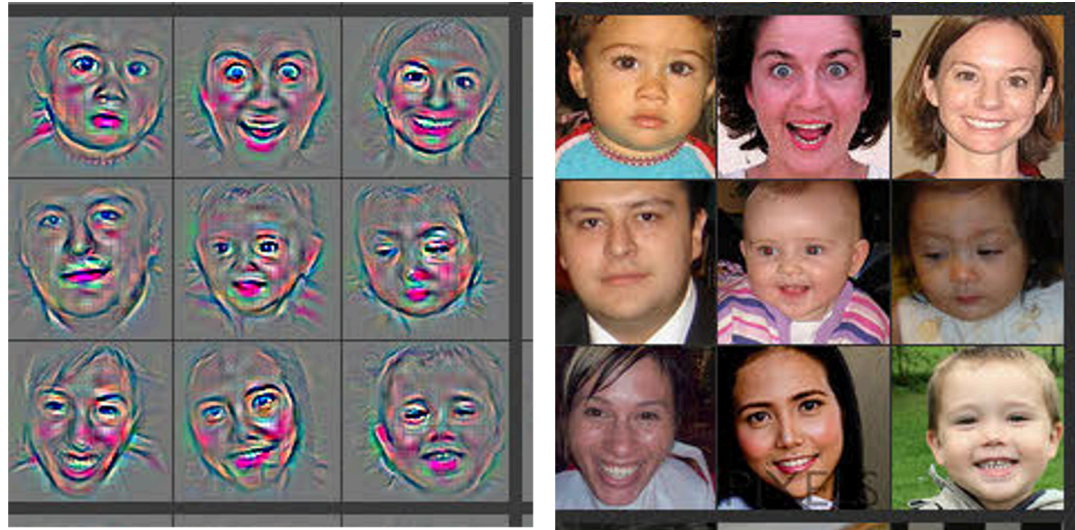
- shallow features are primitive shapes
- deeper features seem to recognize combinations of shallower features

**Input features detected by layer**





Saliency Maps and Corresponding Patches  
Single Layer 5 Feature Map  
On 9 Maximally Activating Input images



Layer 5 ? Feature Map (Row 11, col 1).

Attribution: <https://arxiv.org/abs/1311.2901> (<https://arxiv.org/abs/1311.2901>).

In the simple architectures of the Intro course, we mostly ignored the intermediate representations

$$\mathbf{y}_{(l)} : 1 \leq l \leq (L - 1)$$

The layers were referred to as "hidden" for a reason !

We will discover uses for intermediate representations and show how to build a "feature extractor" to obtain them from a given architecture.



# Recurrent Neural Networks

We use subscripted parenthesis to index into a sequence

$$\mathbf{y}_{(t)}$$

is element  $t$  of the sequence  $\mathbf{y}$ .

Consider the task of

- constructing the *next* element  $\mathbf{y}_{(t)}$  of sequence  $\mathbf{y}$
- conditioned on some input sequence  $\mathbf{x} = \mathbf{x}_{(1)} \dots \mathbf{x}_{(t')}$

$$p(\mathbf{y}_{(t)} | \mathbf{x}_{(1)} \dots \mathbf{x}_{(t)})$$

## Examples

- Translate sequence of tokens
  - $\mathbf{x}$  in one language
  - into  $\mathbf{y}$ , the translation into a second language
- Large Language Model: predict the next token conditioned on all *previously generated* tokens
  - in this case, we condition on

$$\mathbf{x} = (\mathbf{y}_{(0)} \dots \mathbf{x}_{(t-1)})$$

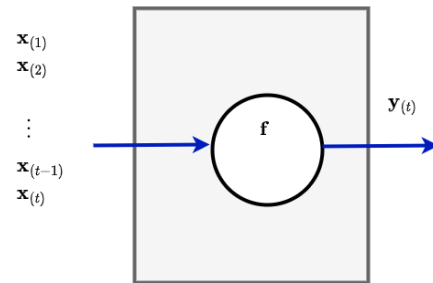
How should be implement the prediction of  $\mathbf{y}_{(t)}$  ?

Some choices

- Predict  $\mathbf{y}_{(t)}$  as a direct function of the prefix of  $\mathbf{x}$  of length  $t$ :

$$p(\mathbf{y}_{(t)} | \mathbf{x}_{(1)} \dots \mathbf{x}_{(t)})$$

**Direct function**



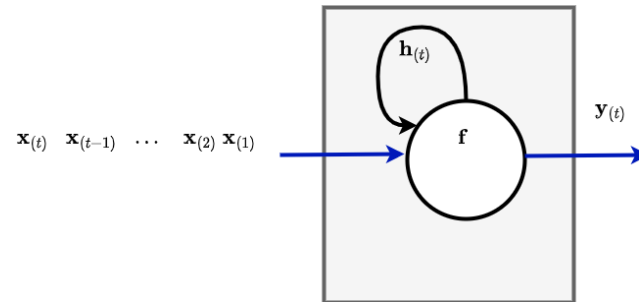
- Loop

- Uses a "latent state" that is updated with each element of the sequence, then predict the output

$p(\mathbf{h}_{(t)}|\mathbf{x}_{(t)}, \mathbf{h}_{(t-1)})$  latent variable  $\mathbf{h}_{(t)}$  encodes  $[\mathbf{x}_{(1)} \dots \mathbf{x}_{(t)}]$

$p(\mathbf{y}_{(t)}|\mathbf{h}_{(t)})$  prediction contingent on latent variable

Loop with latent state



## Latent state

The *latent state*  $\mathbf{h}_{(t)}$  is a kind of memory that acts as a *summary* of the prefix of sequence  $\mathbf{x}$  through time step  $t$ :

$$\mathbf{h}_{(t)} = \text{summary}(\mathbf{x}_{([1:t])})$$

Note that  $\mathbf{h}_{(t)}$  is a *vector* of fixed length.

Thus, it is a *fixed length* representation of the key aspects of a sequence  $\mathbf{x}$  of potentially *unbounded* length.

## Example

Let's use an RNN to compute the sum of a sequence numbers

- the latent state  $\mathbf{h}_{(t)}$  can be maintained as

$$\mathbf{h}_{(t)} = \text{summary}(\mathbf{x}_{([1:t])}) = \sum_{t'=1}^t \mathbf{x}_{(t')}$$

- by updating  $\mathbf{h}_{(t)}$  in the loop

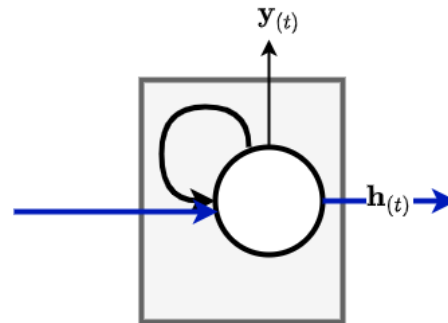
$$\mathbf{h}_{(t)} = \mathbf{h}_{(t-1)} + \mathbf{x}_{(t)}$$

Let's make this concrete with an example: a sequence of words

RNN

---

Machine Learning is easy not hard



$\mathbf{h}_{(t)}$  is a **fixed length** vector that "summarizes" the prefix of sequence  $\mathbf{x}$  up to element  $t$ .

The sequence is processed element by element, so order matters.

$$\mathbf{h}_{(0)} = \text{summary}([\text{Machine}])$$

$$\mathbf{h}_{(1)} = \text{summary}([\text{Machine}, \text{Learning}])$$

$$\vdots$$

$$\mathbf{h}_{(t)} = \text{summary}([\mathbf{x}_{(0)}, \dots, \mathbf{x}_{(t)}])$$

$$\vdots$$

$$\mathbf{h}_{(5)} = \text{summary}([\text{Machine}, \text{Learning}, \text{is}, \text{easy}, \text{not}, \text{hard}])$$



The importance of  $\mathbf{h}_{(t)}$  being *fixed length*

- can be used as input to other types of Neural Network layers
- which *don't* process sequences.

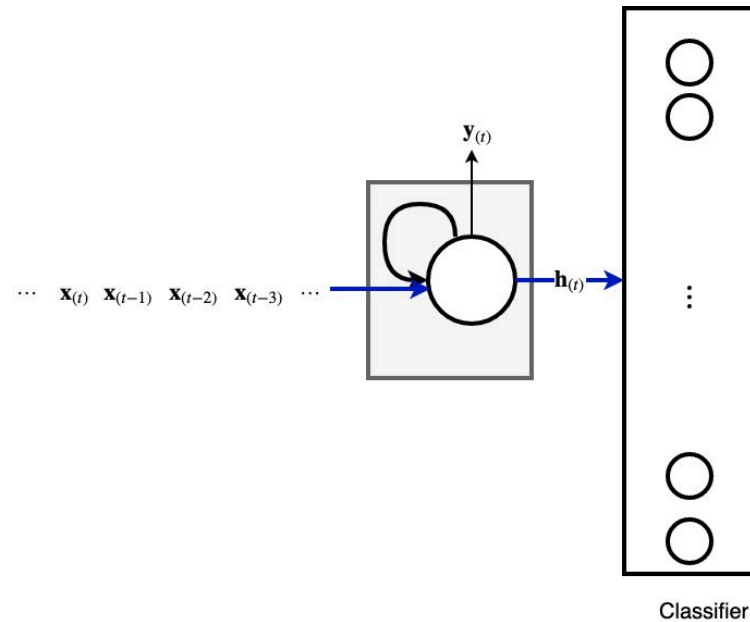
A typical example is a model for text classification (sentiment)

- Using an RNN to create a fixed length encoding of a variable length sequence
- A Head Layer that is a Binary Classifier

---

RNN Many to one; followed by classifier

---





## Output $\hat{\mathbf{y}}_{(t)}$ of an RNN

According to our pseudo-code and diagram

$$\hat{\mathbf{y}}_{(t)} = \mathbf{h}_{(t)}$$

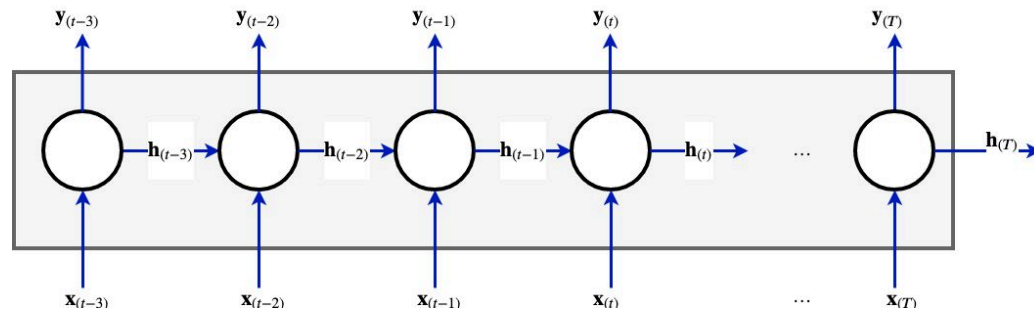
That is: the output is the same as the latent state.

It is easy to add another NN to transform  $\mathbf{h}_{(t)}$  into a  $\hat{\mathbf{y}}_{(t)}$  that is different

- we will omit this additional layer for clarity

# Unrolled RNN diagram

RNN many to many API



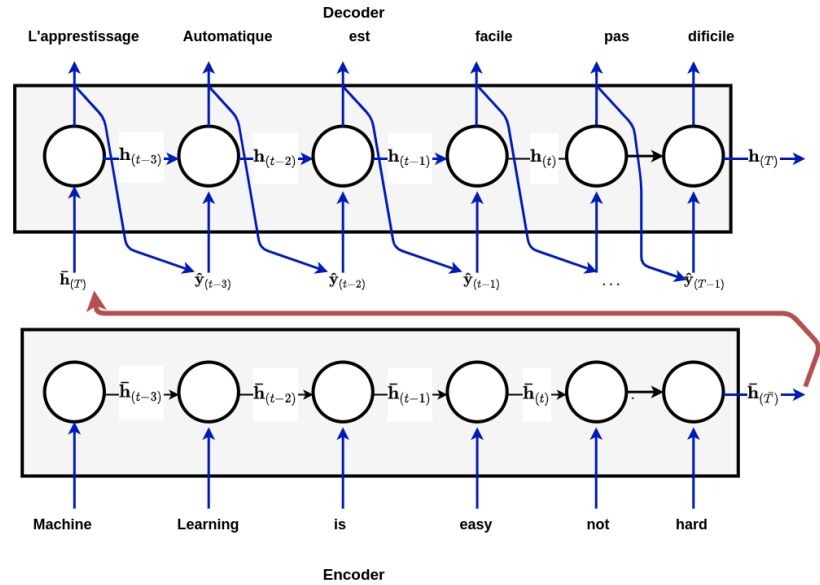
## Encoder-Decoder architecture; Auto-regressive

A very common architecture pairs two RNN's

- an Encoder, which summarizes the input sequence  $\mathbf{x}_{([1:\bar{T}])}$  via final latent state  $\bar{\mathbf{h}}_{(\bar{T})}$
- a Decoder, which takes the input summary  $\bar{\mathbf{h}}_{(\bar{T})}$  and outputs sequence  $\hat{\mathbf{y}}_{([1:T])}$

It is used for *Sequence to Sequence* tasks where both the input and output are sequences.

# Encoder-Decoder for language translation

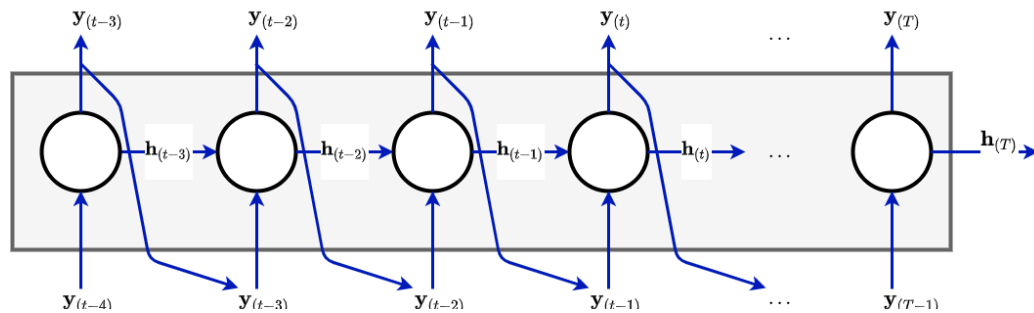


Notice that the Decoder output  $\hat{\mathbf{y}}_{(t-1)}$  at position  $(t - 1)$  is fed back as *input* for position  $t$ .

This is called *Autoregressive* behavior.

It is typical behavior for Generative tasks.

Test time: no forcing





# Language Models

The *Language Model* training objective

- given some text
  - sequence of *tokens*
- predict a word that could be the next word in the sequence

We sometimes refer to this as the "predict the next" task.

Clearly, we need to train a model on the "predict the next" objective with labeled examples.

But this is sometimes called Semi-Supervised or Unsupervised because text is not inherently labeled.

Yet we can easily create  $T$  labeled examples from a text string  $s[1 : T]$ . Example  $t$

- feature:  $s[1 : t - 1]$
- label:  $s[t]$

$$\mathbf{s} = \mathbf{s}_{(1)}, \dots, \mathbf{s}_{(T)}$$

$i$	$\mathbf{x}^{(i)}$	$\mathbf{y}^{(i)}$
1	$\mathbf{s}_{(1)}$	$\mathbf{s}_{(2)}$
2	$\mathbf{s}_{(1),(2)}$	$\mathbf{s}_{(3)}$
$\vdots$		
$i$	$\mathbf{s}_{(1),\dots,(i)}$	$\mathbf{s}_{(i+1)}$
$\vdots$		
$(T - 1)$	$\mathbf{s}_{(1),\dots,(T-1)}$	$\mathbf{s}_{(T)}$



The *Unsupervised Pre-Trained Model + Supervised Fine-Tuning* paradigm is

- a way of adapting a model trained on the Language Modeling objective
- to perform another task

Pre-training refers to training a model on the Language Modeling objective with *lots* of data

- this is called Unsupervised because text is not inherently labeled
- we can easily create a labeled example from a text string  $s[1 : T]$ 
  - feature:  $s[1 : t - 1]$
  - label:  $s[t]$
- Pre-training
  - Train a model with *lots* of data
  - On the

In [2]: `print("Done")`

Done

