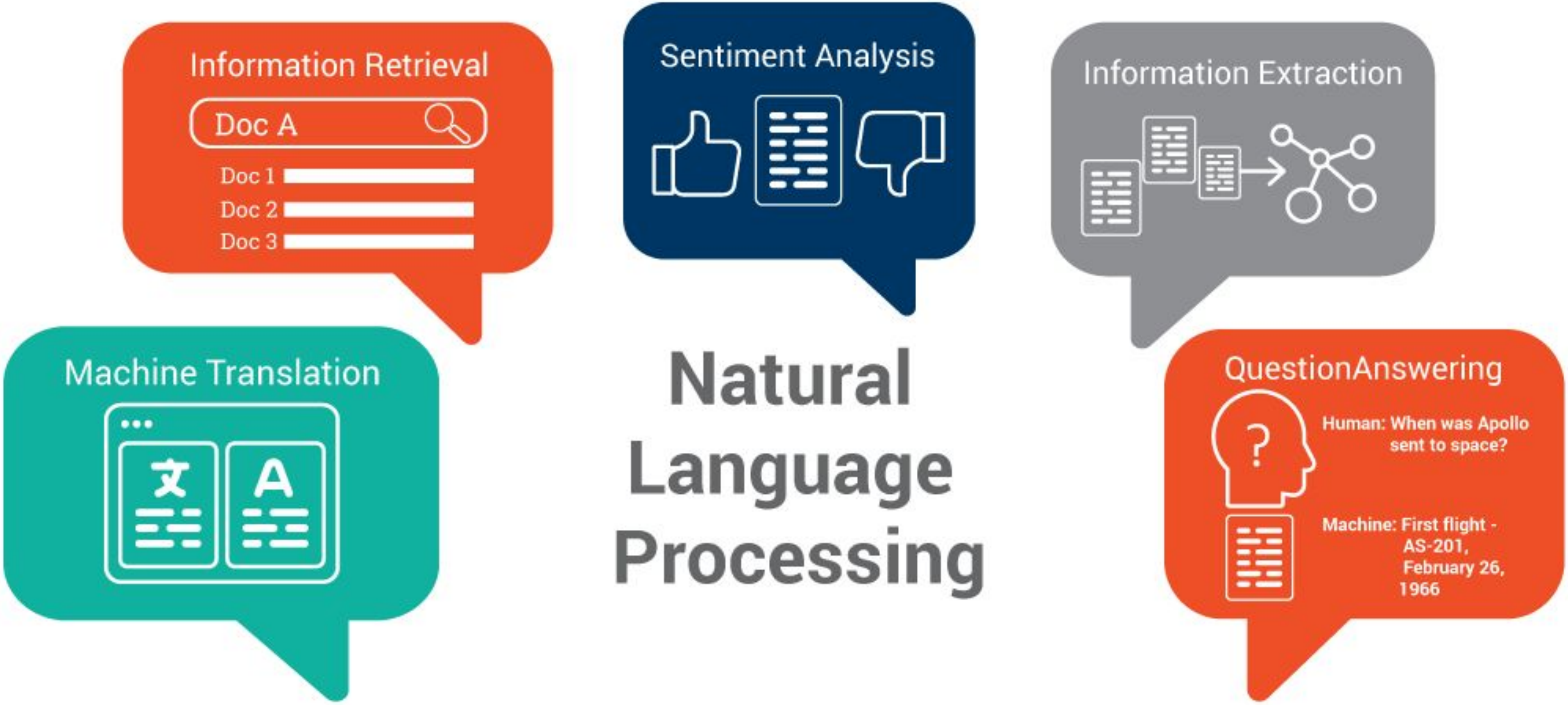
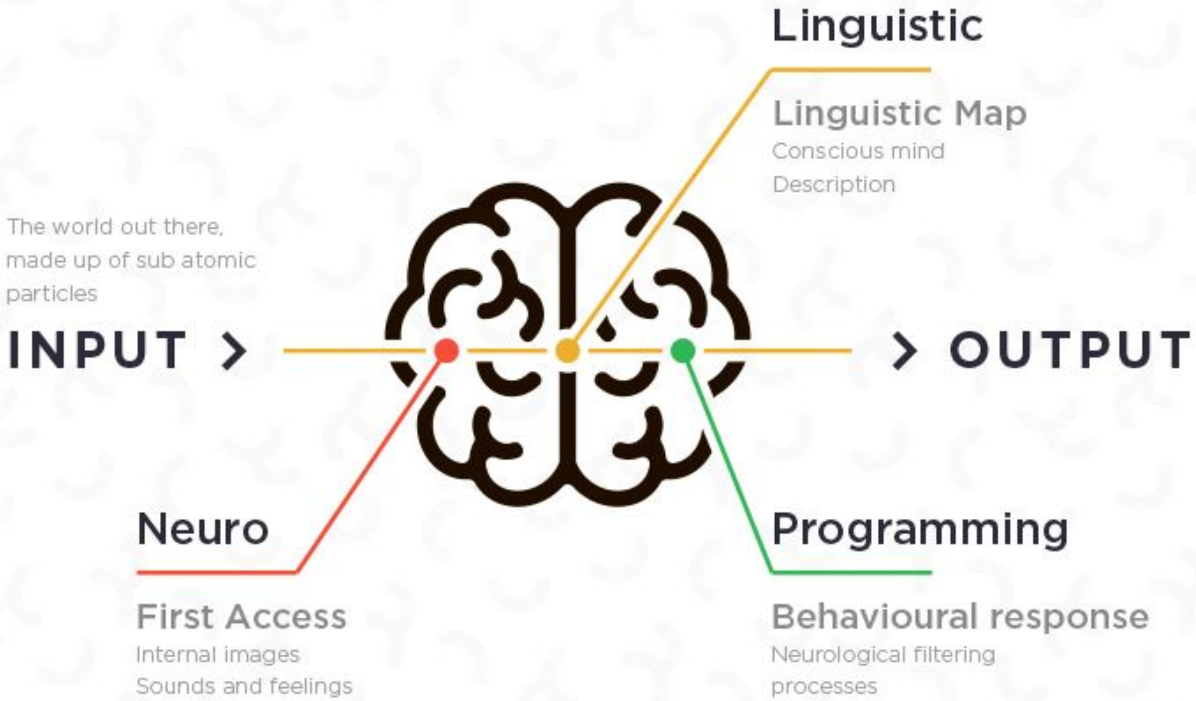
PyTorch

Here we begin the best thing


Information Retrieval
Doc A
Doc 1
Doc 2
Doc 3
Sentiment Analysis
Information Extraction
Machine Translation
文
A
Natural
Language
Processing
QuestionAnswering
?
Human: When was Apollo sent to space?
Machine: First flight - AS-201, February 26, 1966

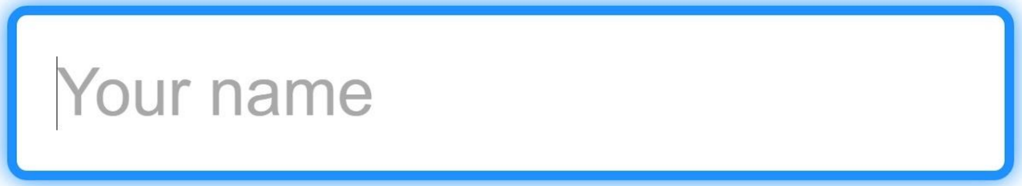

Linguistic
Linguistic Map
Conscious mind
Description
The world out there,
made up of sub atomic
particles
INPUT >
> OUTPUT
Neuro
First Access
Internal images
Sounds and feelings
Programming
Behavioural response
Neurological filtering
processes

Now we have input

Input: ‘i like to eat ice cream’.

Human also read, from left to right, defined by the language.

Machine also read from left to right, defined by the programmer, based on the real language’s rules.



Neuro process,

What human do, we unconsciously combine character-by-character to become a word defined by a space, then word-by-word become a sentence defined by our creativity.

Same goes to computer but not really like what humans do, it requires an initial memory to stack character-by-character until found a space, then initiate another initial memory for next character stack to combine become sentence, ended by human intervention.

Linguistic map

This is still a black box on how humans map thousands of words in just a split of a microseconds. How do we know certain patterns of words got certain fuzziness for certain emotion and sentiment?

We learned word pattern unconsciously since we are young, required years to understand other emotion and sentiment. Our brain keeps iterating the learning both when we awake and sleep.

Linguistic map (cont)

We are trying our best to make machine understand human natural languages by defining finite sets of rules by simply giving also a finite set of benchmarking.

We all agree that machines can understand narrow natural language intelligence. If a machine is trained on sentiment, it would only perform in that specific domain.

Linguistic map (cont)

How does human stored the fuzzy rules in their brain?

If we want to store every word combination in this world, it is an insanely huge amount of static memory required!

Plus, every generation used different combination to define a communication, to define an emotion, sarcasm, etc!

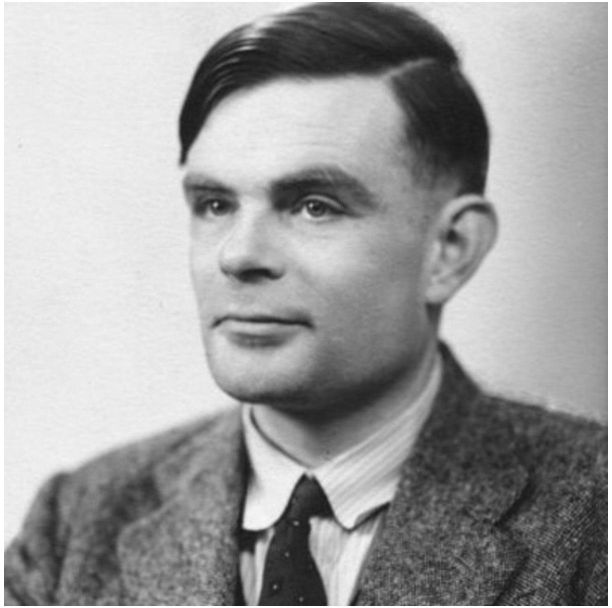
How are we actually storing them? Do we actually store every pattern?

Linguistic map (cont)

This guy (Alan Turing, father of modern computer) proposed ‘Oracle’.

A machine that able to answer any question in this world.

Should we save every single possible combination of question?

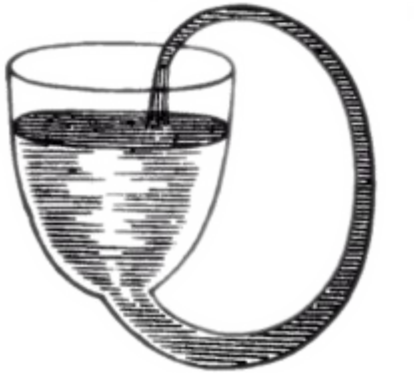


Programming

Computer requires human finite rules to perform certain tasks.

Machines does not perpetual itself to define their own set of rules to perform a task.

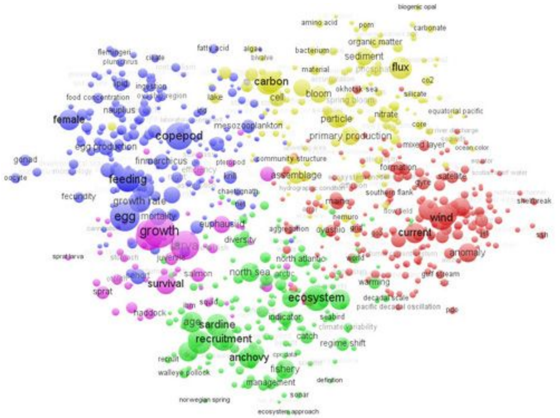
The weights of word defined by our judgements. We do accept the approximation values from the machine, because there is no exact hypothesis able to define the best solution for a problem.



Example of NLP

1- Easy

- Spell checking
- Keyword find
- Finding synonyms



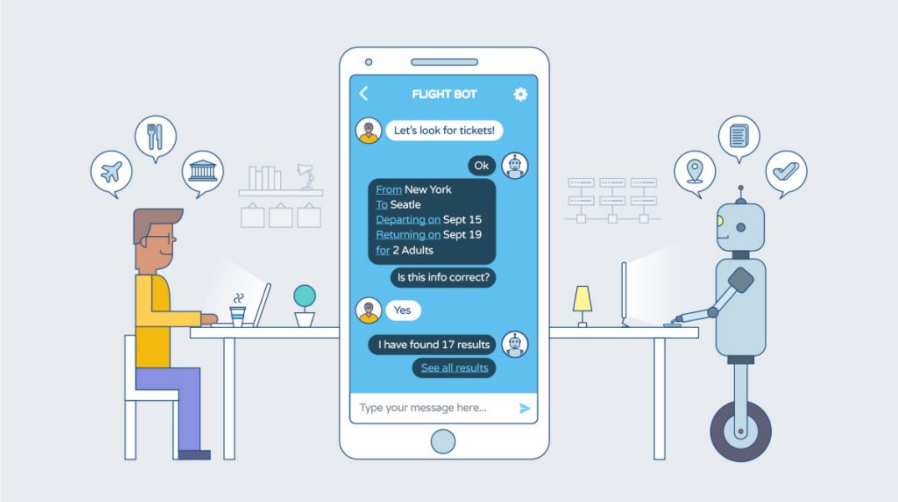
2- Medium

- Parsing information from text / document. Or simply we call text mining.

Example of NLP (cont)

3- Hard

- Machine translation, e.g (dialect Bahasa to English)
- Semantic Analysis (meaning of sentence)
- Reference (what does ‘he’ pointed to)
- Question - Answer
- Chatbot



But,

Most important is,

How we represent words as input for our NLP tasks.

But,

Most important is,

How we represent meaning of words as input for our NLP tasks.

Vectorization Technique

The process we going to change words into vector representation, called vectorizing.

Commonly used:

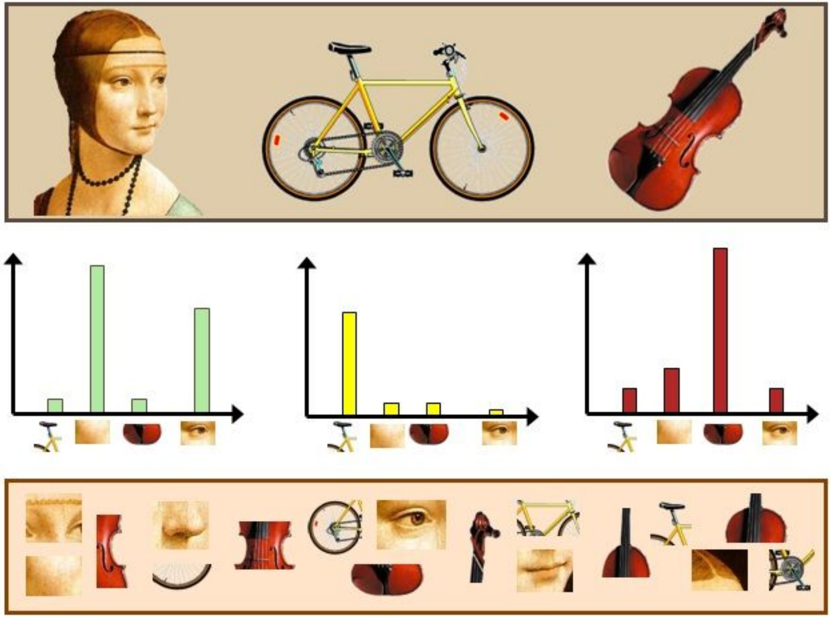
1- Bag-Of-Word / Unigram

2- N-Gram

3- TF-IDF

4- Timestamp based on dictionary position

5- Word Vector based on Skip-gram model



Bag-Of-Word / Uni-Gram / 1-Gram

Accumulated unique words in a sentence. Frequency unique words in sentences in a document.

```
In [3]: import numpy as np
        sentence = 'the dog is on the table'
        unique_words = list(set(sentence.split()))
        bow = np.zeros((len(unique_words)))
        for i in sentence.split():
            bow[unique_words.index(i)] += 1.0

        bow
Out[3]: array([ 1.,  1.,  2.,  1.,  1.])
```

That is means, ‘the dog is on the table’ brings 1 magnitude towards x-axis, 1 magnitude towards y-axis, 2 magnitude towards z-axis, 1 magnitude towards a-axis, 1 magnitude towards b-axis, 1 magnitude towards c-axis, based on BOW.

N-Gram

sequence of N adjacent elements from a string of tokens (the way how you splitted it).

```
import numpy as np
sentence = 'the dog is on the table'
sentence_splitted = sentence.split()
grams = []
N = 2
for i in range(len(sentence_splitted)-N):
    grams.append(' '.join(sentence_splitted[i:i+N]))
grams = list(set(grams))
print(grams)
gram_2 = np.zeros((len(grams)))
for i in range(len(sentence_splitted)-N):
    gram_2[grams.index(' '.join(sentence_splitted[i:i+N]))] += 1
gram_2
```

```
['is on', 'dog is', 'on the', 'the dog']
array([ 1.,  1.,  1.,  1.])
```

1 x-axis, 1 y-axis, 1 z-axis, 1 a-axis based on Bi-Gram

```
sentence = 'the dog is on the table'
sentence_splitted = sentence.split()
grams = []
N = 3
for i in range(len(sentence_splitted)-N):
    grams.append(' '.join(sentence_splitted[i:i+N]))
grams = list(set(grams))
print(grams)
gram_3 = np.zeros((len(grams)))
for i in range(len(sentence_splitted)-N):
    gram_3[grams.index(' '.join(sentence_splitted[i:i+N]))] += 1
gram_3
```

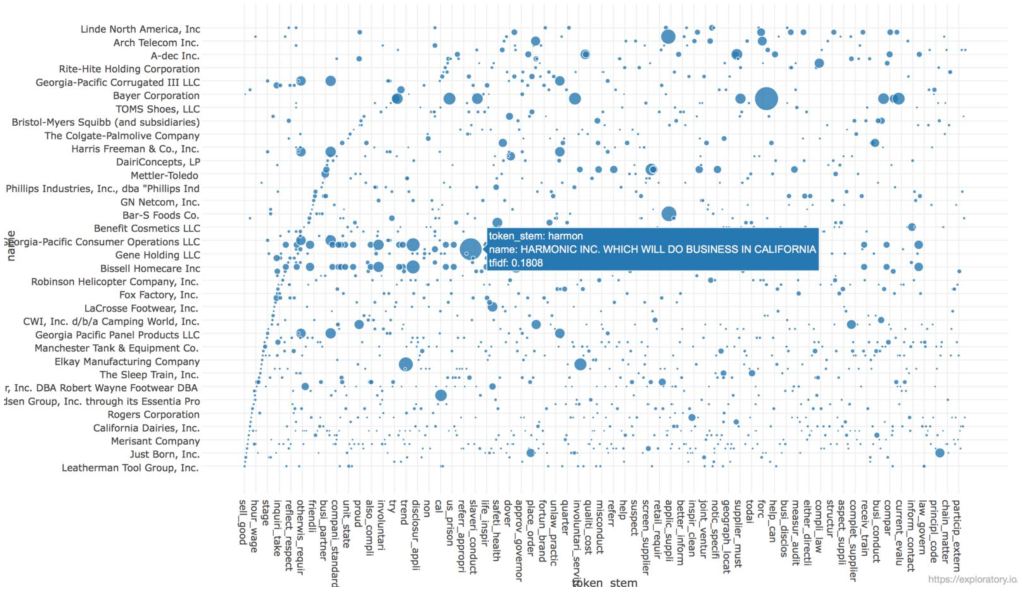
```
['the dog is', 'dog is on', 'is on the']
array([ 1.,  1.,  1.])
```

1 x-axis, 1 y-axis, 1 z-axis based on Tri-Gram

TF-IDF

Term Frequency * Inverse Document Frequency

based on the frequency method but it is different to the count vectorization in the sense that it takes into account not just the occurrence of a word in a single document but in the entire corpus.



TF-IDF (cont)

this	1
is	1
really	2
love	3

this	1
is	1
really	1
hate	1

TF = (Number of times term t appears in a document)/(Number of terms in the document)

IDF = log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.

TF(love, left) = 3 / 7, IDF(love) = log(2 / 1), TF * IDF = 3 / 7 * 0.3 = 9 / 70

TF-IDF (cont)

this	1
is	1
really	2
love	3

this	1
is	1
really	1
hate	1

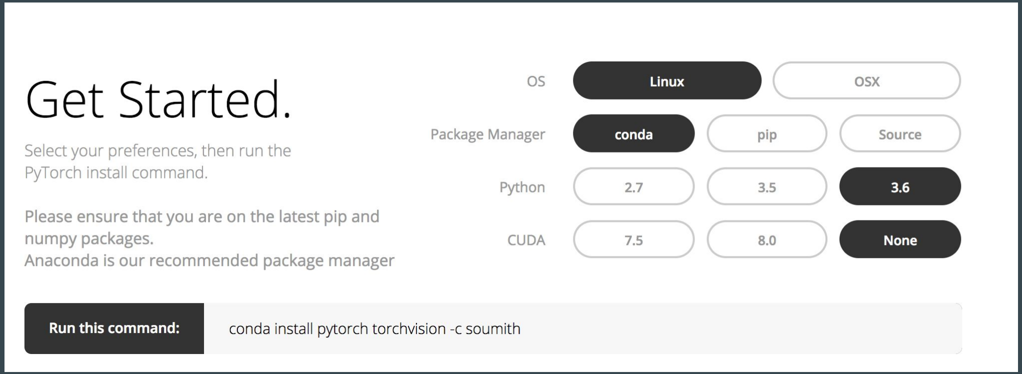
TF(this, left) = 1 / 7, IDF(this) = log(2 / 2), TF * IDF = 1 / 7 * 0 = 0

TF-IDF method heavily penalises the word ‘this’ but assigns greater weight to ‘love’. So, this may be understood as ‘love’ is an important word for Left from the context of the entire corpus.

NLP Basics

Before we begin

- Anaconda Python3.5
- Install PyTorch



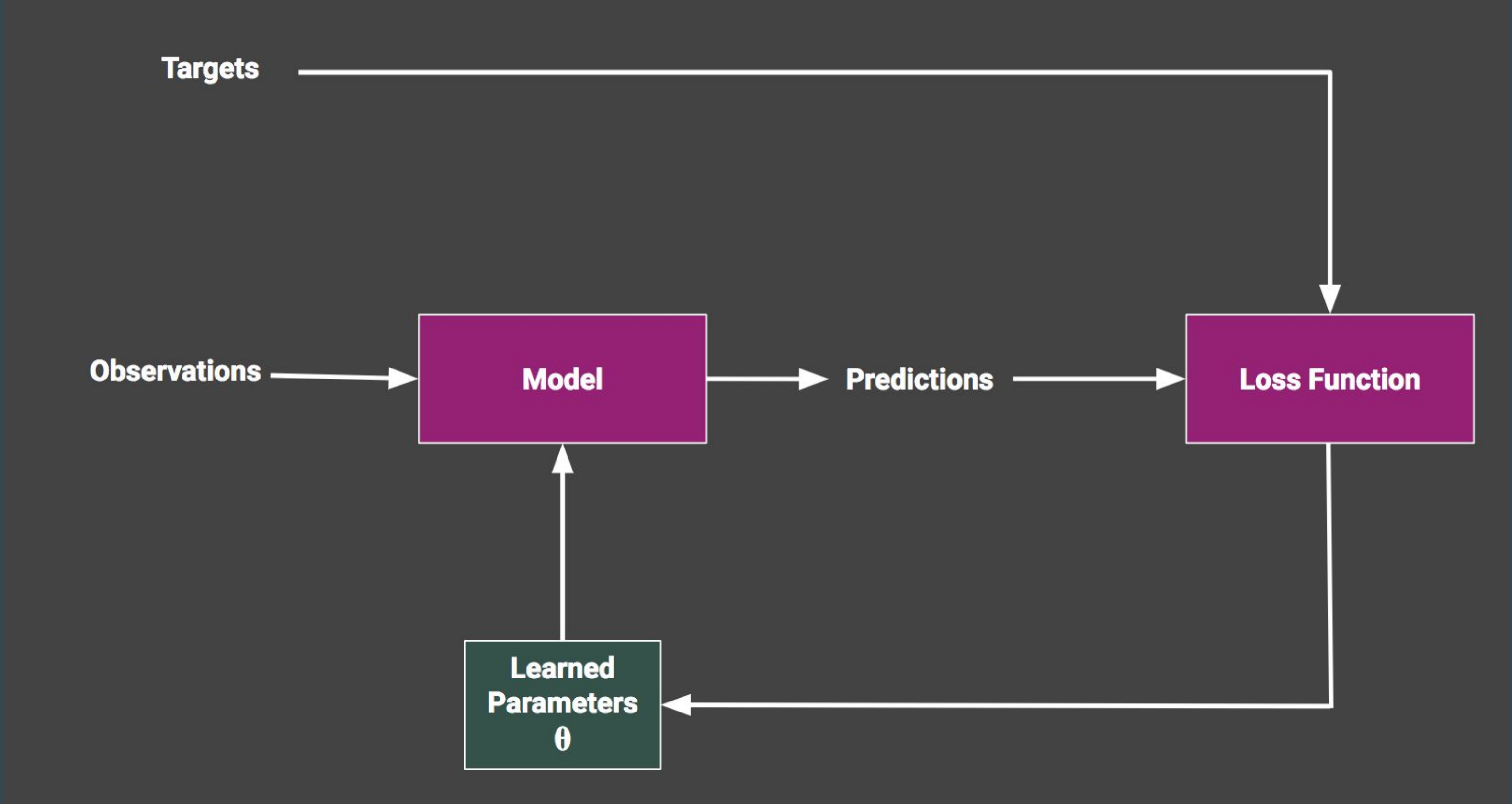
- Install requirements
- Download data

What is PyTorch

It’s a Python based package for serving as a replacement of Numpy and to provide flexibility as a Deep Learning Development Platform.

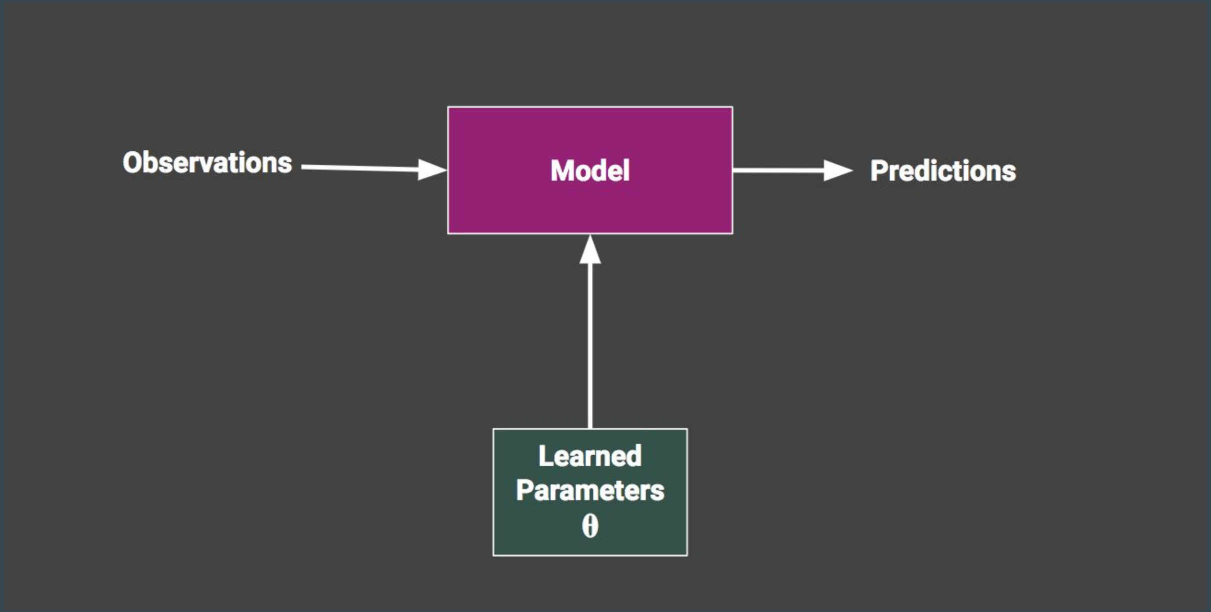
(Supervised) Machine Learning 101

Training

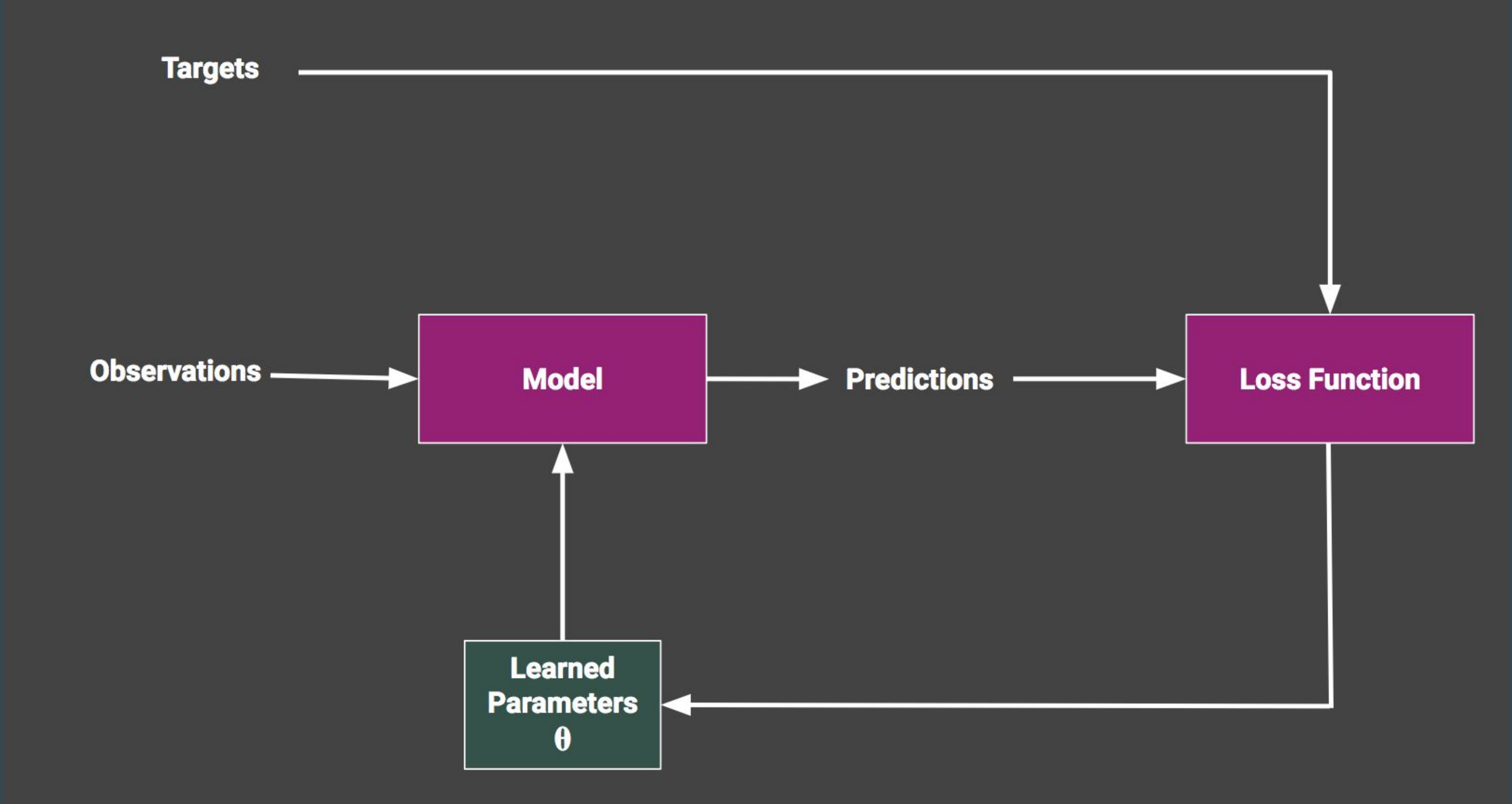


Targets
Observations
Model
Predictions
Loss Function
Learned
Parameters
θ

Inference



Training


Targets
Observations
Model
Predictions
Loss Function
Learned
Parameters
θ

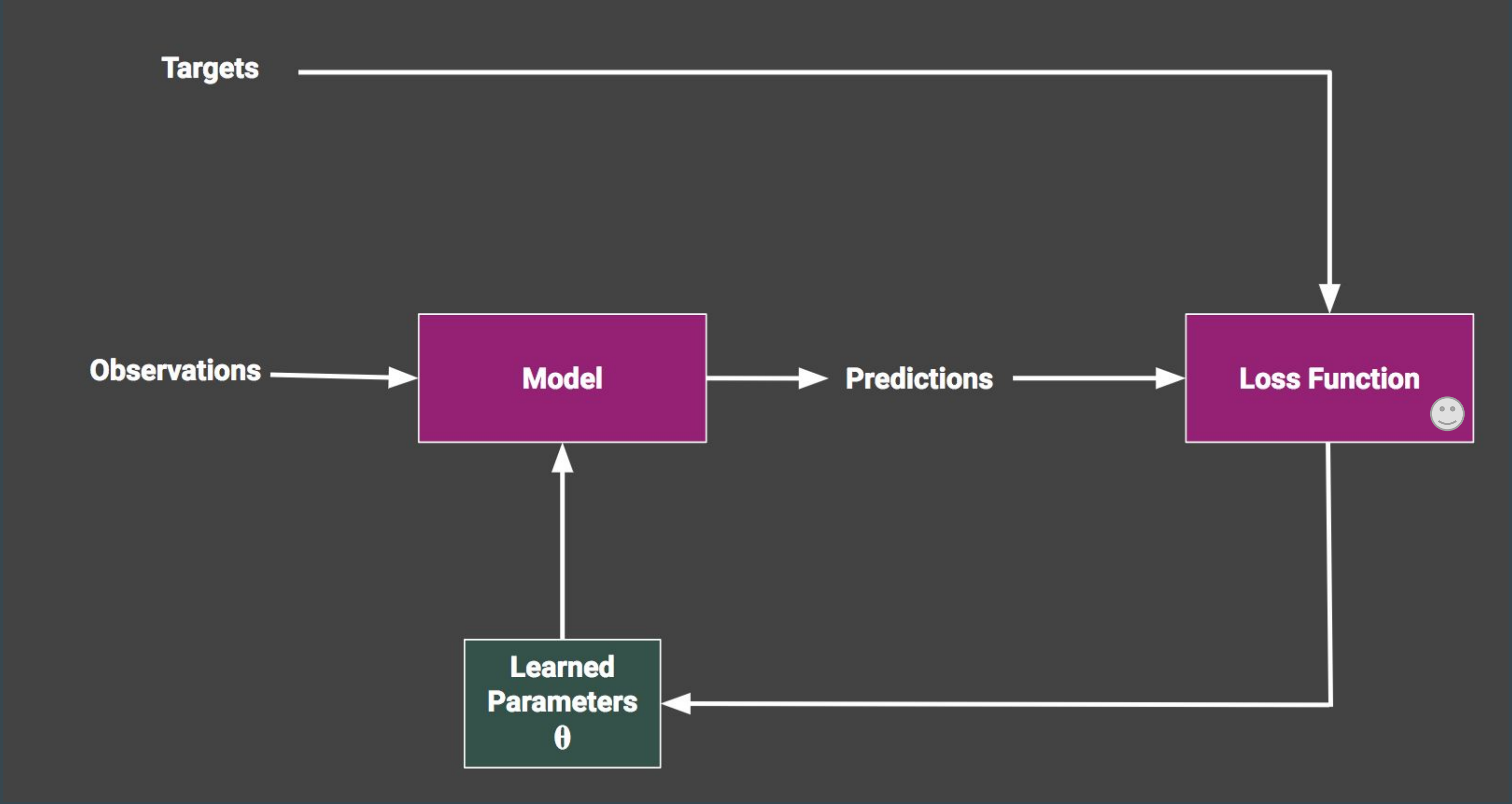
Loss function

- A function that maps a target (y) and its prediction (y) to a real value
- Higher the value, the worse off the prediction is from target
- Other names: Objective Function, Risk function

Examples:

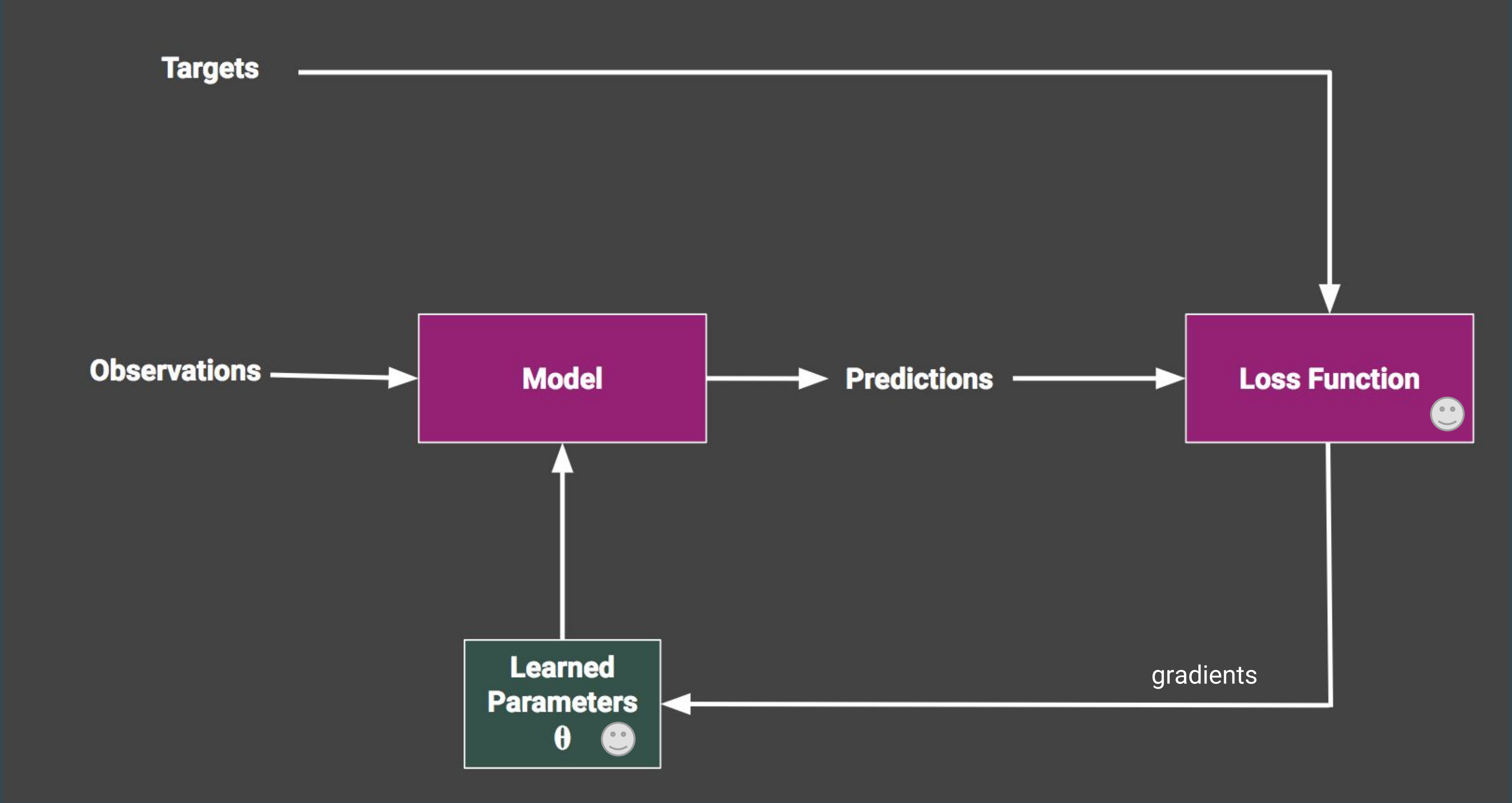
L1 Loss	$loss(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n} \lvert y_i - \hat{y}_i \rvert$
L2 loss	$loss(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$
CrossEntropy	$loss(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{n}\sum_{i=1}^{n} [y_i \ log \ \hat{y}_i + \hat{y}_i \ log \ y_i]$
NLL loss	$loss(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{n}\sum_{i=1}^{n} log \ \hat{y}_i$

Training



Targets
Observations
Model
Predictions
Loss Function
Learned
Parameters
θ

Training


Targets
Observations
Model
Predictions
Loss Function
Learned Parameters
θ
gradients

Model

"learn a function f(x, ***W***), parameterized by ***W***"

Two components here:

1. Structure of the function *f*
2. Parameters (or weights) ***W***

Say *f()* was a linear model, then

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{d} w_i x_i + w_0$$

Model

“learn a function f(x, ***W***), parameterized by ***W***”

Two components here:

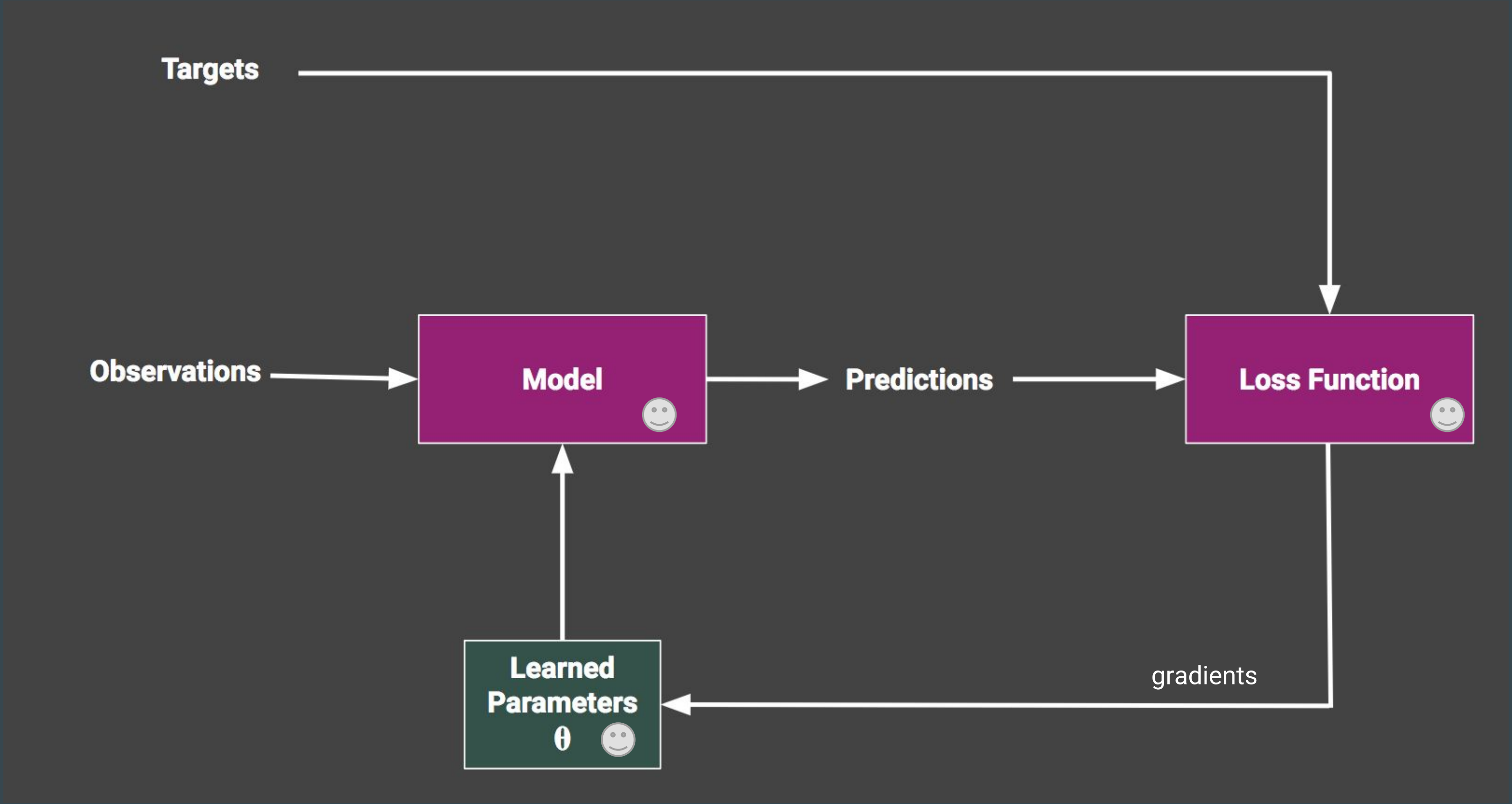
1. Structure of the function *f*
2. Parameters (or weights) ***W***

Say *f()* was a linear model, then

$$f(\mathbf{x}, \mathbf{w}) = \sum_{i=1}^{d} w_i x_i + w_0$$

But *f()* can be arbitrarily complex

Training



Targets
Observations
Model
Predictions
Loss Function
Learned
Parameters
θ
gradients

Computational Graph

Nodes: Operations

Edges: arguments to the operation

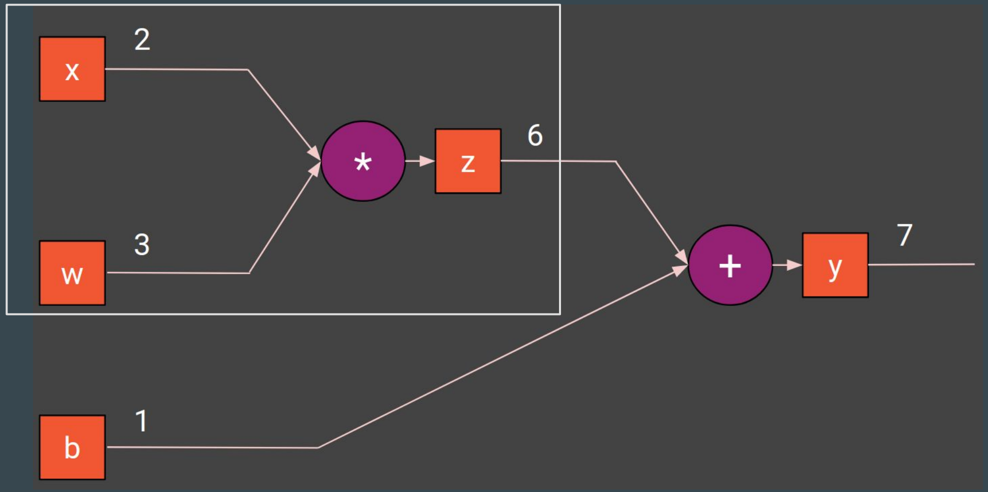
$y = wx + b$

Computational Graph

Nodes: Operations

Edges: arguments to the operation

$y = \boxed{wx} + b$

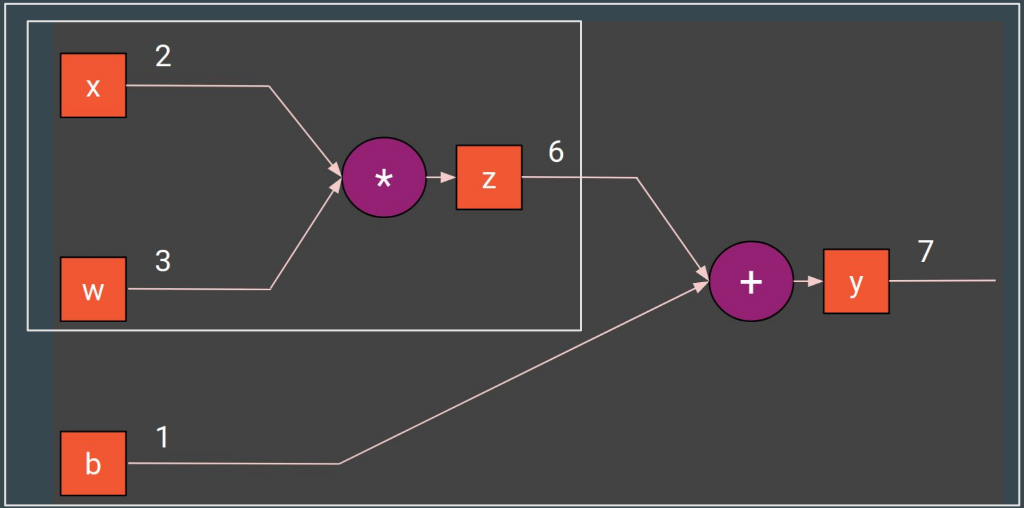


Computational Graph

Nodes: Operations

Edges: arguments to the operation

y = wx + b



Computational Graph

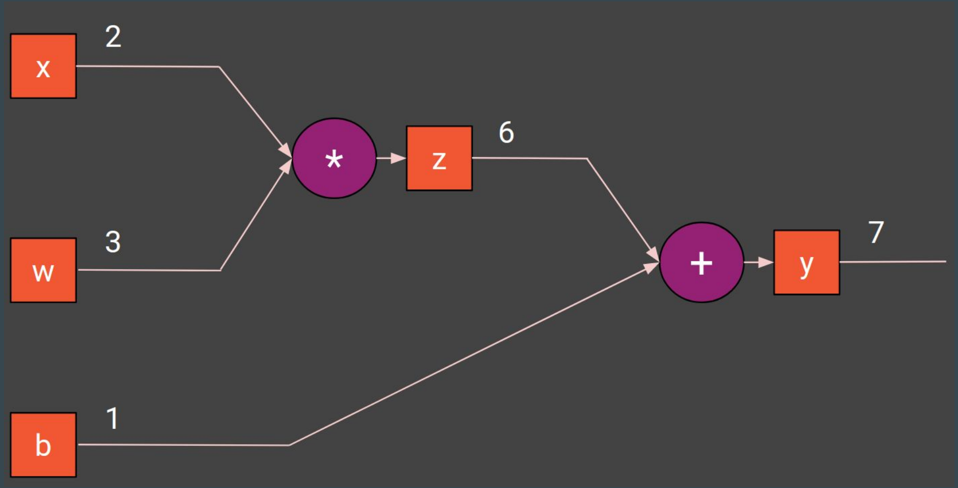
Nodes: Operations

Edges: inputs and outputs of the operation

y = wx + b

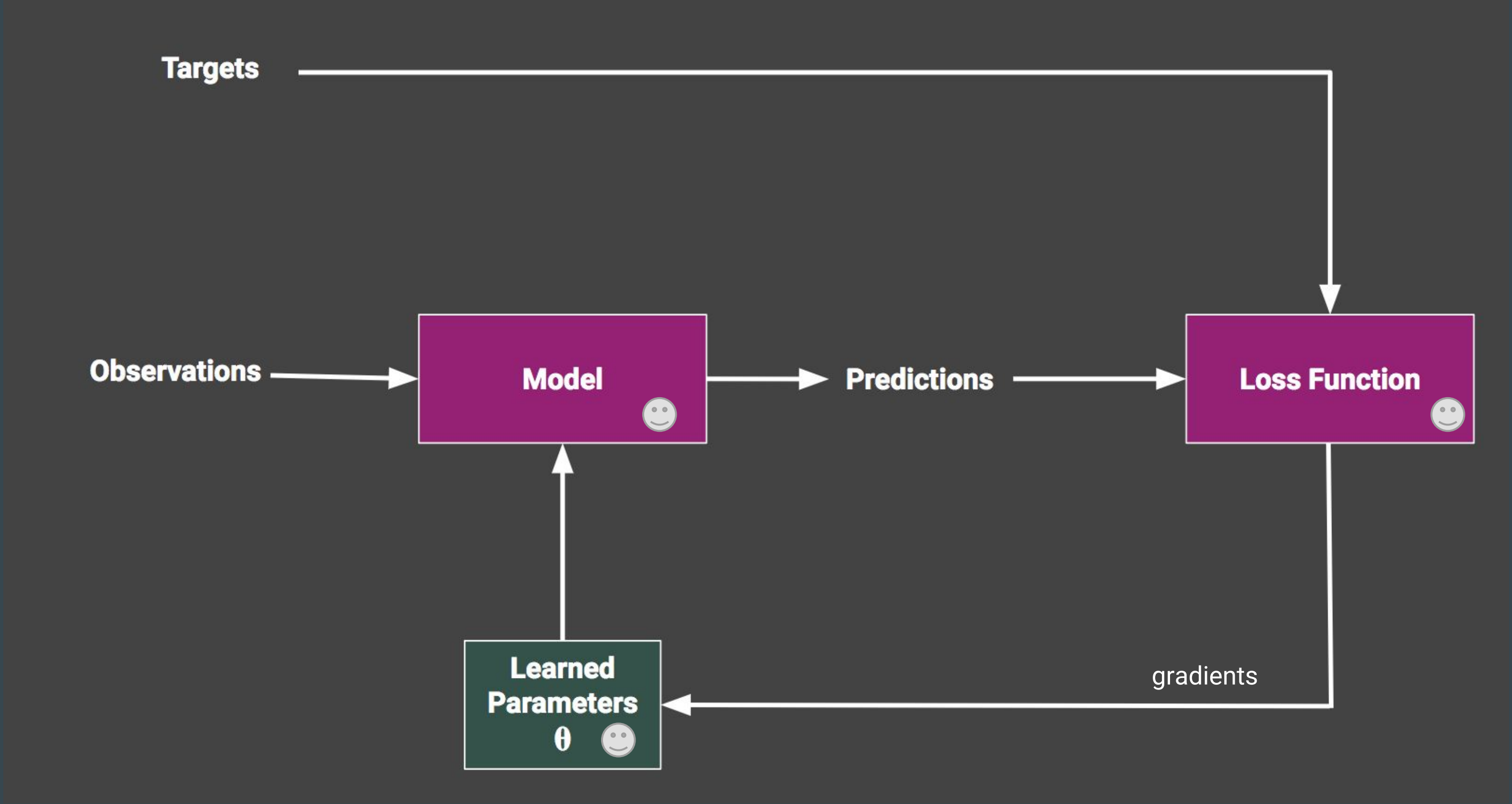
Saving the model:

- Save the weights
- Save the graph structure



Why use Computational Graphs?

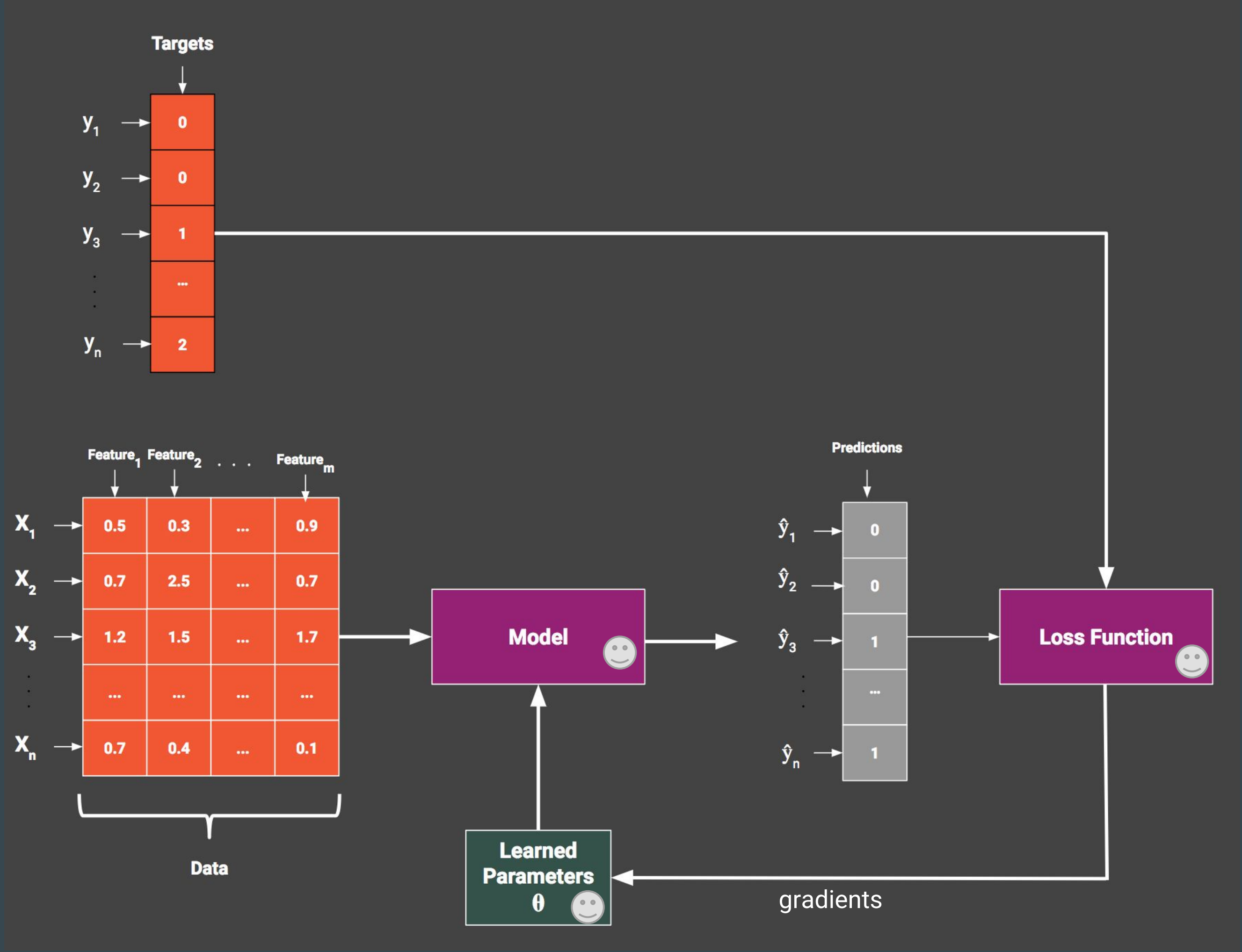
Training



Targets
Observations
Model
Predictions
Loss Function
Learned
Parameters
θ
gradients

Label Encoding

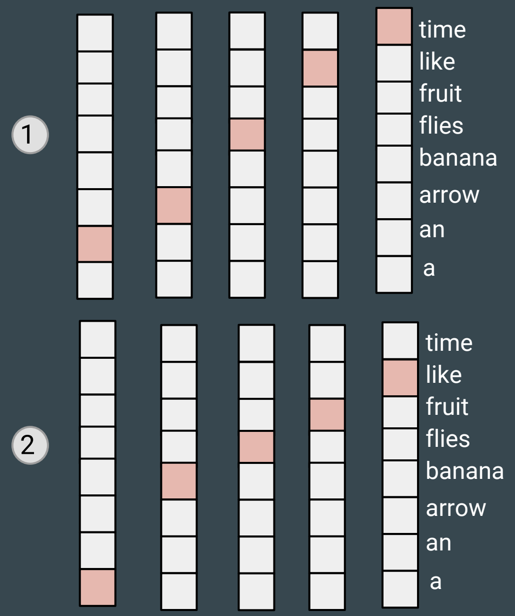
Input Encoding



Input Encoding

One-Hot

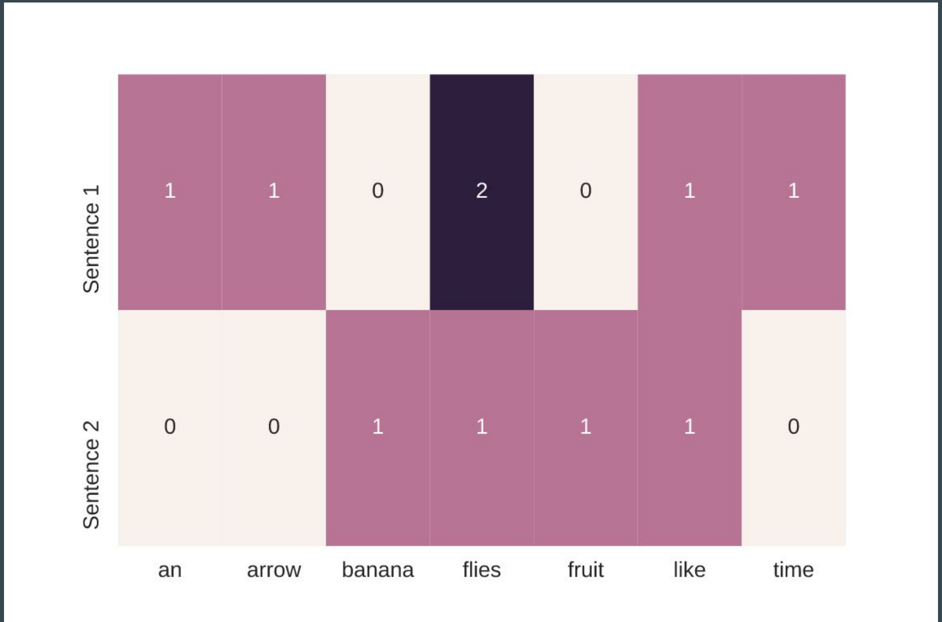
1. time flies like an arrow
2. fruit flies like a banana



Input Encoding

Term-Frequency

```
time flies like an arrow
fruit flies like a banana
```



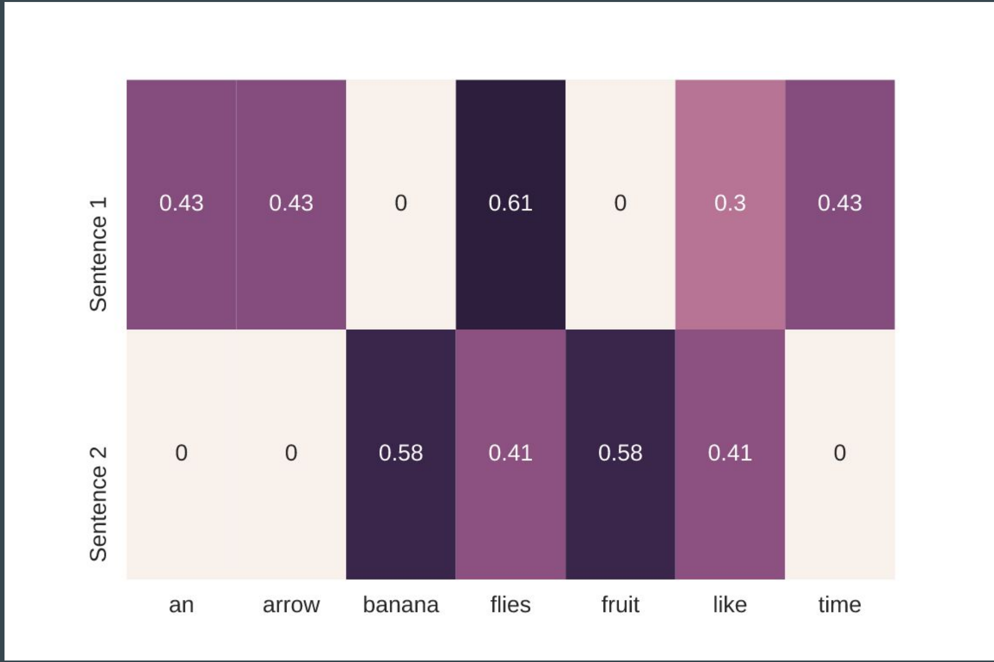

```
from sklearn.feature_extraction.text import CountVectorizer
```

Input Encoding

TF-IDF

```
time flies like an arrow
fruit flies like a banana
```

$$f_{t,d} \cdot \log \frac{N}{n_t}$$

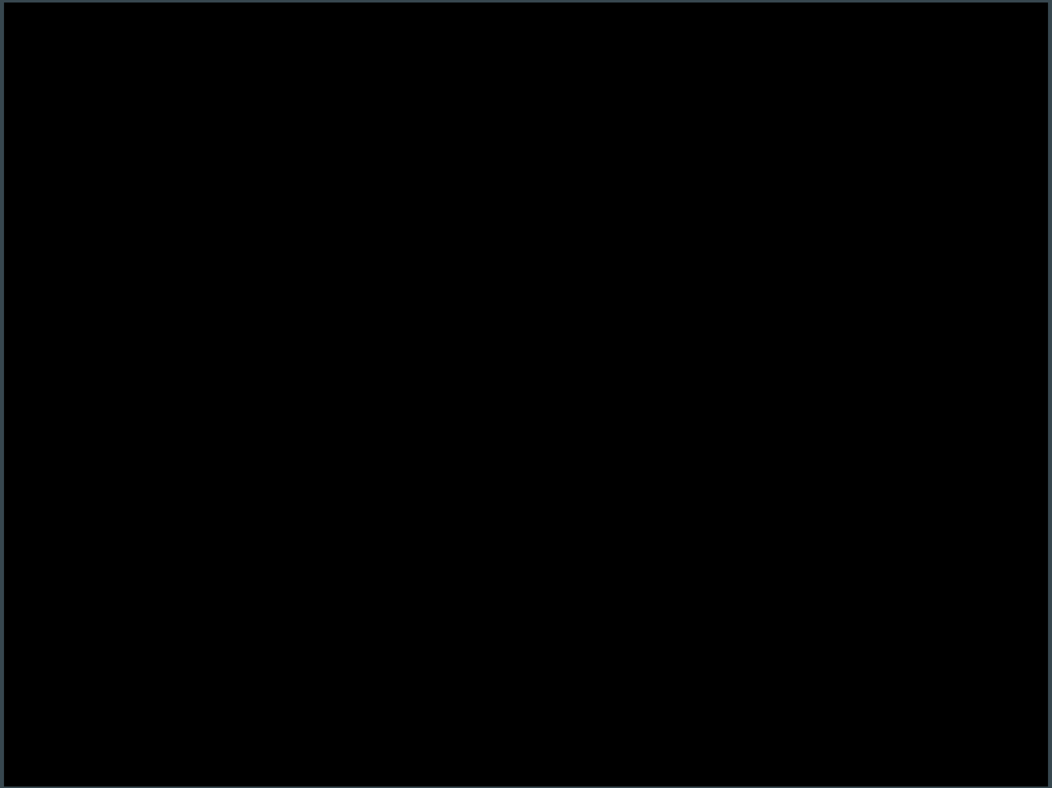



```
from sklearn.feature_extraction.text import TfidfVectorizer
```

Input Encoding

Other forms

- LSA / SVD



d_1 : *Romeo and Juliet.*

d_2 : *Juliet: O happy dagger!*

d_3 : *Romeo died by dagger.*

d_4 : *"Live free or die", that's the New-Hampshire's motto.*

d_5 : *Did you know, New-Hampshire is in New-England.*

Courtesy: http://webhome.cs.uvic.ca/~thomo/svd.pdf

Word Embeddings

- Pre-trained Embeddings
  - Word2Vec-derived
  - Glove
- Task-specific (supervised) embeddings

Distributional Representations: A long history in NLP

“A word is known the company it keeps” -- Firth, 1957

Using Pretrained Embeddings (Notebook)

What can be encoded?

- Words
- Positions of words
- Part-of-speech tags
- Padding
- Unknown words
- Word shapes
- Multiword expressions
- Named entities

... practically everything

Ways to combine dense representations

green apple

- Concat

"green apple"

- Continuous Bag of Words (CBOW)

"green apple"

- WCBOW

"green apple"

- Nonlinear → A neural network!

Deep Learning Frameworks
(computational graph frameworks)

Deep Learning Frameworks

K

Chainer

theano

PYTORCH

Caffe

TensorFlow

Deep Learning Frameworks

K

Chainer

theano

PYTORCH

Caffe

TensorFlow

Deep Learning Frameworks

Deep Learning Frameworks

DYNAMIC FRAMEWORKS!

What’s “Dynamic”

Define and Run

vs.

Define by Run

Define and Run

1. Define the neural network
2. “Compile” step
3. Feed data to the network (“Run”)

Define and Run

1. Define the neural network
2. "Compile" step
3. Feed data to the network ("Run")
4. Limiting as the network should know the actual operation (programming logic)

Define by Run

1. Define the network "on the fly" by running code.
2. Possible because history of computation is preserved
3. Can define really complex network topologies easily

Define and Run

1. Define the neural network
2. "Compile" step
3. Feed data to the network ("Run")
4. Limiting as the network should know the actual operation (programming logic)

```
import tensorflow as tf

sess = tf.InteractiveSession()
i = tf.constant(0)
c = lambda i: check_something(i)
b = lambda i: do_something(i)
r = tf.while_loop(c, b, [i])

sess.run(tf.initialize_all_variables())
```

TensorFlow

Define by Run

1. Define the network "on the fly" by running code.
2. Possible because history of computation is preserved
3. Can define really complex network topologies easily

```
import torch

i = 0
while(check_condition(i)):
    i = do_something(i)
```

PYTORCH

Define and Run

1. Define the neural network
2. "Compile" step
3. Feed data to the network ("Run")
4. Limiting as the network should know the actual operation (programming logic)

theano

TensorFlow

PYTORCH

Define by Run

1. Define the network "on the fly" by running code.
2. Possible because history of computation is preserved
3. Can define really complex network topologies easily

Chainer

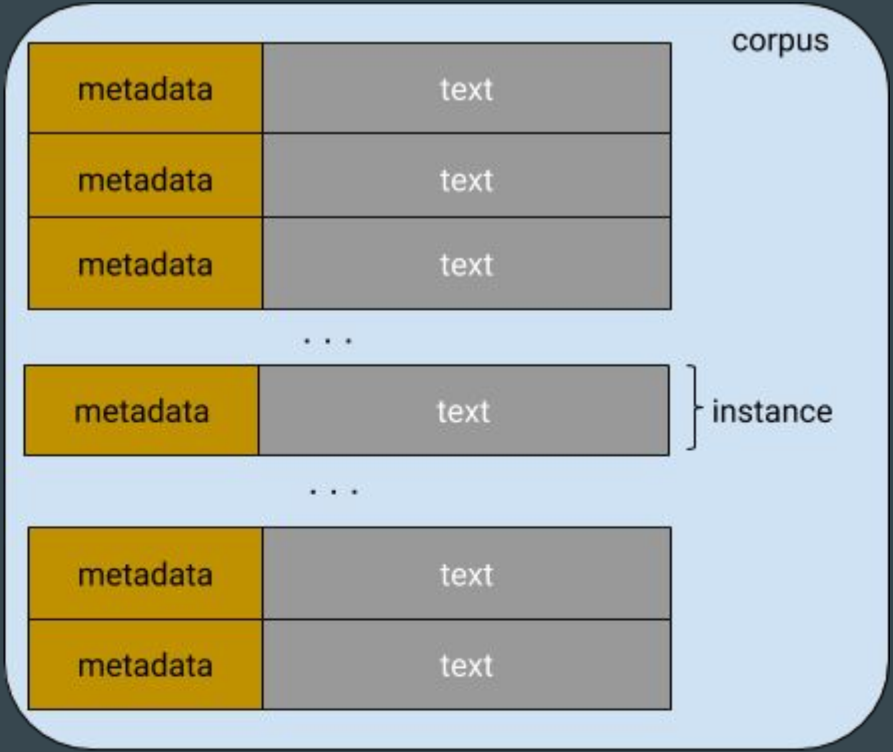
Why Dynamic Frameworks

- Variable length inputs
- Complex structured outputs

PyTorch Basics

Corpora, Tokens, Types

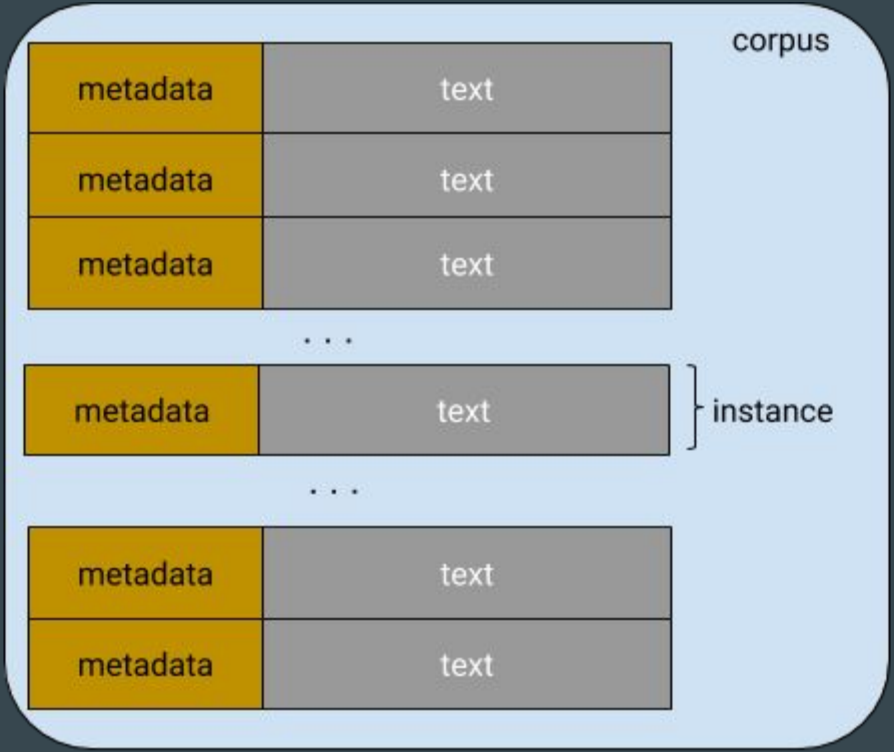
```
Peter Piper picked a peck of pickled peppers.
Did Peter Piper pick a peck of pickled peppers?
If Peter Piper Picked a peck of pickled peppers,
Where's the peck of pickled peppers Peter Piper picked?
```



Corpora, Tokens, Types

```
Peter Piper picked a peck of pickled peppers.
Did Peter Piper pick a peck of pickled peppers?
If Peter Piper Picked a peck of pickled peppers,
Where's the peck of pickled peppers Peter Piper picked?
```

```
3 a
1 did
1 if
4 of
4 peck
4 peppers
4 peter
1 pick
3 picked
4 pickled
4 piper
1 's
1 the
1 where
```



Tokenization: Not always splitting on spaces

Turkish	English
kork(-mak)	(to) fear
korku	fear
korkusuz	fearless
korkusuzlaş (-mak)	(to) become fearless
korkusuzlaşmış	One who has become fearless
korkusuzlaştır(-mak)	(to) make one fearless
korkusuzlaştırıl(-mak)	(to) be made fearless
korkusuzlaştırılmış	One who has been made fearless
korkusuzlaştırılabil(-mek)	(to) be able to be made fearless
korkusuzlaştırılabilecek	One who will be able to be made fearless
korkusuzlaştırabileceklerimiz	Ones who we can make fearless
korkusuzlaştırabileceklerimizden	From the ones who we can make fearless
korkusuzlaştırabileceklerimizdenmiş	I gather that one is one of those we can make fearless
korkusuzlaştırabileceklerimizdenmişçesine	As if that one is one of those we can make fearless
korkusuzlaştırabileceklerimizdenmişçesineyken	when it seems like that one is one of those we can make fearless

http://i.imgur.com/yaTxPoI.jpg

Tokens -> Lemmas and Stems

- Lemma: canonical (dictionary) form of a word/token
  ran -> run
  running -> run
  runs -> run
- Process of deriving lemma: lemmatization
- Stem: approximation for lemmas
  universal -> univers
  university -> univers
  universe -> univers
- Part of a larger art in NLP called “feature engineering”

Claim

*all of NLP is

Structured (output)

Prediction

*well, almost

NLP (aka Structure prediction) Tasks

- Categorizing words: Part of Speech Tagging

```
Mary - PROPN
slapped - VERB
the - DET
green - ADJ
witch - NOUN
. - PUNCT
```

- Categorizing multi-word units:
  - Shallow parsing or “chunking”

    [NP Mary] [VP slapped] [NP the green witch] .

  - Named Entity Recognition

    John PERSON was born in Chicken GPE , Alaska GPE , and studies at Cranberry Lemon University ORG .

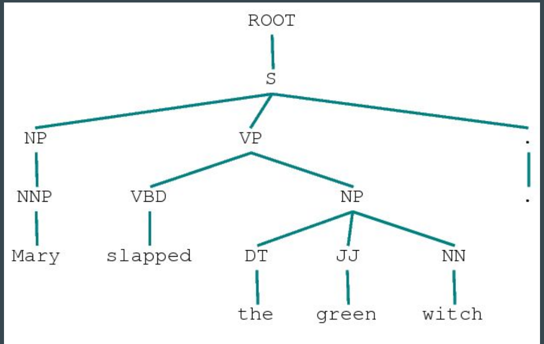
- Morphology: Sub-word parsing

"noncombatants" ⇒ (Stem (Prefix non-] (Root combat] (Prefix -ant]] + Inflexional Suffix -s]

NLP (aka Structure prediction) Tasks

- Constituent Parsing: Extracting sentence structure
  - Terminal nodes
  - Non-terminal nodes

- Dependency Parsing: Extracting Relationships between word units



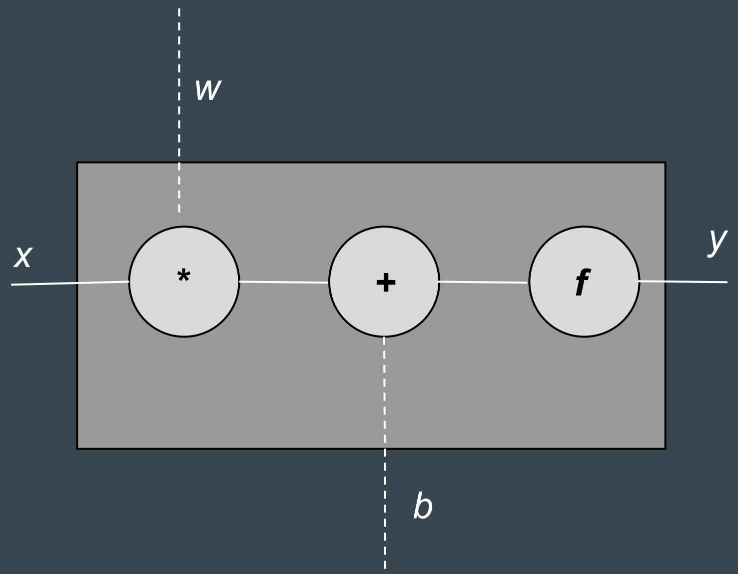
Multi Layer Perceptrons

Perceptron (Rosenblatt, 1958)

“the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.”

-- New York Times, 1958

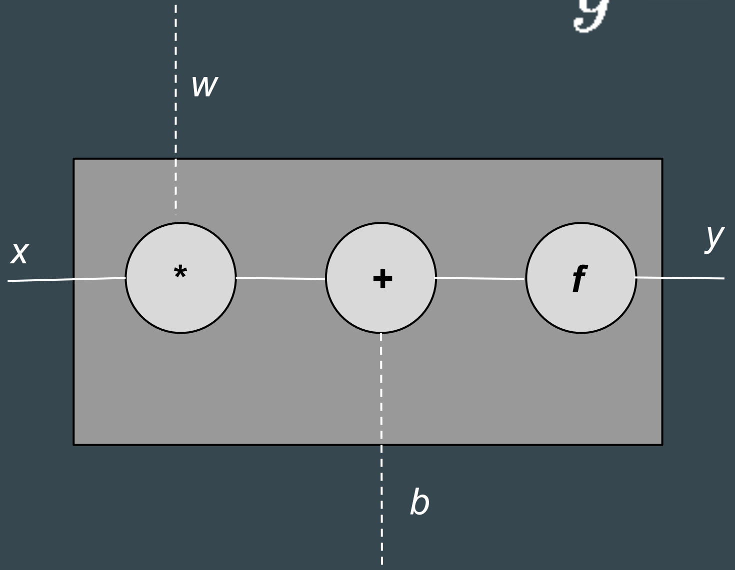
Perceptron as a computational graph



Perceptron as a computational graph

$$y = f(wx + b)$$

"Activation" function

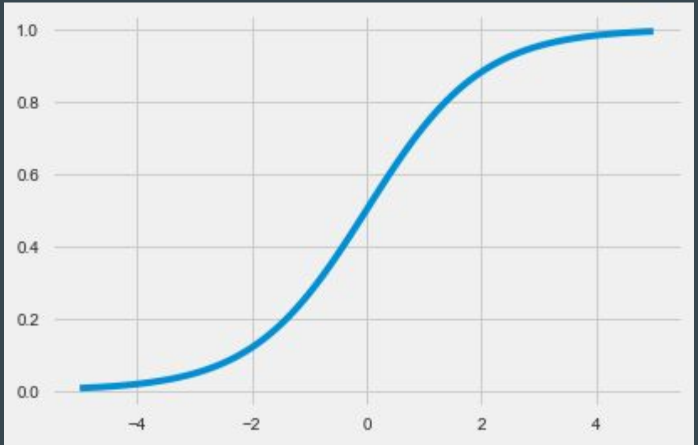


Activation function: Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$

```
import torch
import matplotlib as plt

x   torch range( 5.  5.  0.1)
y   torch sigmoid(x)
plt plot(x numpy()  y numpy())
plt show()
```

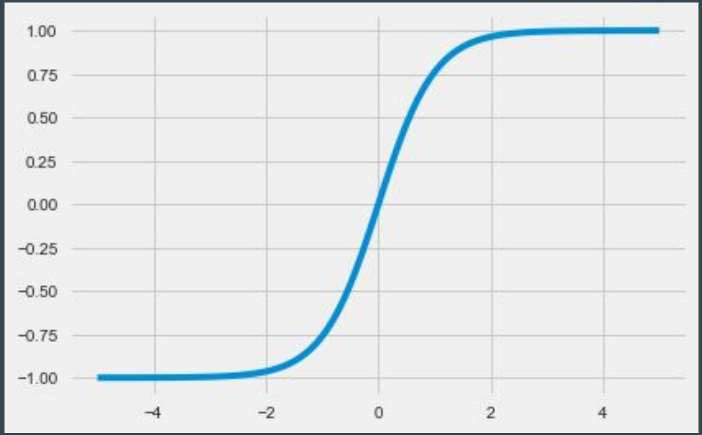


Activation function: tanh

$$f(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

```
import torch
import matplotlib as plt

x   torch range( 5.  5.  0.1)
y   torch tanh(x)
plt plot(x numpy()  y numpy())
plt show()
```



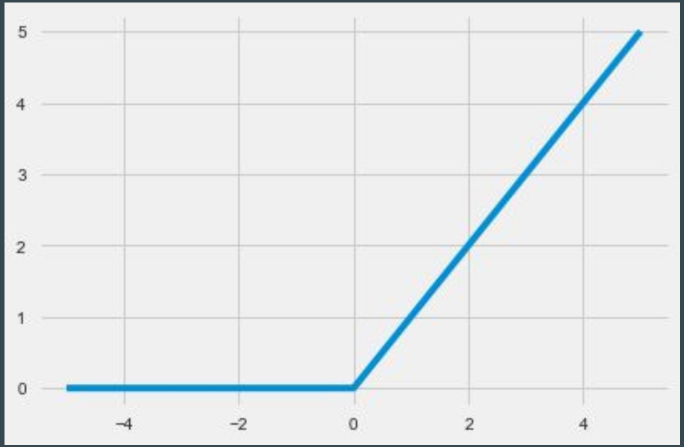
Activation function: ReLU

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

```
import torch
import torch.nn as nn
import matplotlib as plt

relu   nn ReLU()
x   torch range( 5.  5.  0.1)
y   relu(Variable(x))

plt plot(x numpy()  y data numpy())
plt show()
```



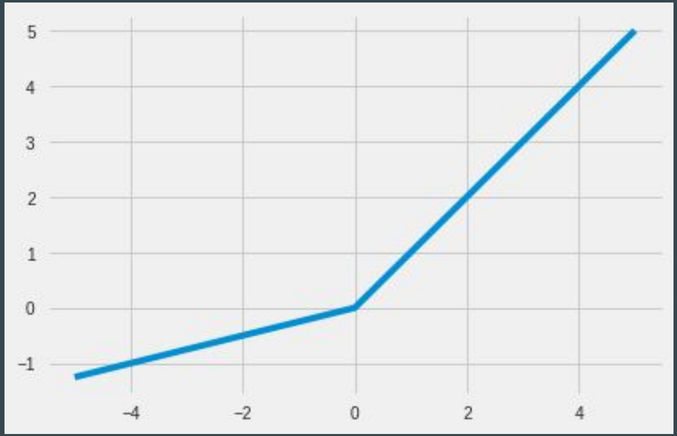
Activation function: PReLU

$$f(x) = \max(x, ax)$$

```
import torch
import torch.nn as nn
import matplotlib as plt

prelu   nn PReLU(num_parameters 1)
x   torch range( 5.   5.   0.1)
y   relu(Variable(x))

plt plot(x numpy()  y data numpy())
plt show()
```



Activation functions

- Many many more.
  - As of Jun 26, 2017: At least 22 activation functions are defined in PyTorch
  - But, only a few of those are heavily used in NLP work
  - See PyTorch documentation for more!
- Which activation function should I use?

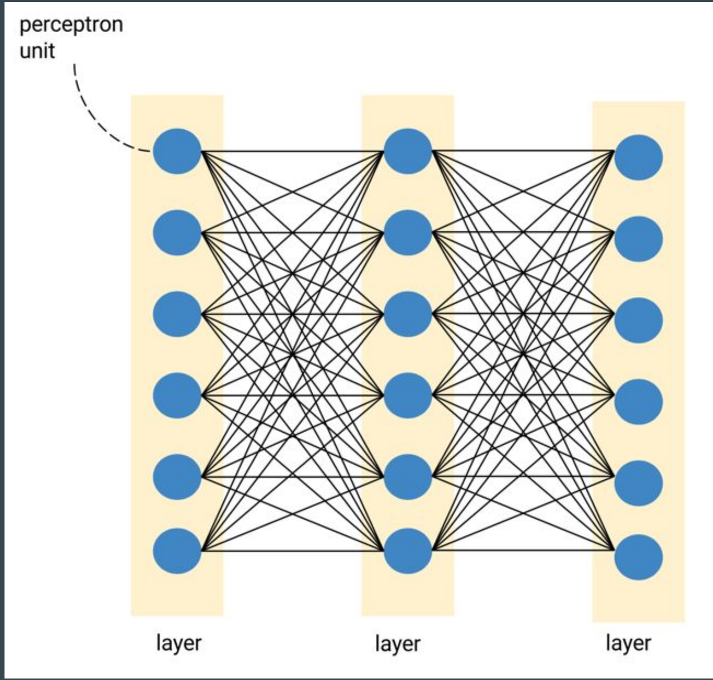
Activation functions

- Many many more.
  - As of Jun 26, 2017: At least 22 activation functions are defined in PyTorch
  - But, only a few of those are heavily used in NLP work
  - See PyTorch documentation for more!
- Which activation function should I use?

Activation functions

- Many many more.
  - As of Jun 26, 2017: At least 22 activation functions are defined in PyTorch
  - But, only a few of those are heavily used in NLP work
  - See PyTorch documentation for more!
- Which activation function should I use?
- Understand how the function "shapes" the output
- "Don't be a hero" : Follow best practices in literature
  (best practices == what actually works :)

From Perceptrons to MultiLayer Perceptrons (MLP)



Notebook

Convolutional Networks for Text

In some NLP problems sub-sequences are predictive

5/15/2014

Cesar was twice scheduled to come out and template. Both times he stood me up. First time he had some dramatic excuse. Second time I never heard back. Total flake. Proceed at your own risk unless you don't value your own time.

10/18/2013

Cesar over at Bay Countertops is Great to work with and he bent over backwards to get our counters installed. Great price and the workmanship is very good, we had Silestone Quartz Lagoon installed with the full bullnose edges, and it looks beautiful. You can't go wrong with Bay and by the way his crew is the Best!

In some NLP problems sub-sequences are predictive

$$\underbrace{\begin{array}{l}\texttt{cellulase}\\ \texttt{amylase}\\ \texttt{lipase}\\ \texttt{ribonuclease}\end{array}}_{\text{Enzyme}}$$

In some NLP problems sub-sequences are predictive

cellulase

amylase

lipase

ribonuclease

Enzyme

radium

palladium

cadmium

thorium

Element

In some NLP problems sub-sequences are predictive

cellulase
amylase
lipase
ribonuclease

Enzyme

radium
palladium
cadmium
thorium

Element

cellulose
sucralose
glucose
fructose

Carbohydrate

If you want to remember one thing about CNNs

CNNs extract meaningful task-specific substructures

If you want to remember one more thing about CNNs

Convolutional Layers can be used a basic blocks in other networks