ISO 29990:2010

NATIONAL INFOCOMM COMPETENCY FRAMEWORK

NUS National University of Singapore

ISS INSTITUTE OF SYSTEMS SCIENCE

Text Processing using Machine Learning

Introduction: Classic vs Deep NLP

Liling Tan

02 Dec 2019

About Me

Work:

Rakuten Institute of Technology

https://rit.rakuten.co.jp/about/

Education:

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

UNIVERSITÄT DES SAARLANDES

What is this course about?

- **Recent advance techniques for NLP**
  - esp. since the "*deep learning tsunami*"
- Getting a **good grasp of PyTorch for NLP**
  - Similar knowledge can be applied to any tensor libraries for machine/deep learning
- **Understanding the underlying techniques and knowing how to implement them**
  - Hopefully, you'll find them less "magical"

What is this course NOT about?

- **Understanding *everything* in NLP.**
  - There's a lot more to *computational linguistics* than what's in this course
- Using **libraries to achieve results for specific NLP tasks**
  - (e.g. SpaCy, Transformer, Gluon, AllenNLP, TorchText, etc.)
- No, **we won't build chatbots** in this course.

Pardon my quirks…

- **Sometimes I let the code do the talking and I just read out the code** (*Stop me if it's not intuitive enough*)

- **I have more code than math in my slides**

- **Feel free to stop me at any time to ask questions**

Some Questions…

- **How many people learnt deep / machine learning from previous courses (in-class or MOOC)?**

- **How many of from computer science/engineering background? How many from humanities, STEM, non-computer engineering or business background? Others?**

- **How many currently working in a technology field? Industry/Academia?**

Course Schedule

Introduction
- Classic vs Deep NLP
- NN from Scratch

Deep Learning Foundations
- Matrix Calculus for Deep Learning
- Backpropagation

Word Embeddings
- Word2Vec, GloVe, Fasttext, and friends
- Word Embeddings from Scratch

Nuts and Bolts
- Bias - Variance
- Regularization, Loss Functions, Optimizers

Language Models
- N-gram + Neural Language Models
- Recurrent Neural Nets

Memory Networks and Conditional Generation
- Exploding and Vanishing Gradients
- LSTM + GRU and Seq2Seq Models

Attention Networks
- Lots of Different Attentions
- Attention is all you need (aka "Transformer")

Sentence Representation
- Multi-tasks vs Transfer Learning
- Pretrained Language Models (aka "Sesame Street")

Machine Translation (Bonus)
- Phrase-based to Neural MT
- 101 tricks to Neural MT training and quirks

Summary
- Recap
- Ask Me Anything

Course Logistics

For the hands-on later, it'll take a while to download and install, so we do this first while the lesson goes on…

Go to https://www.anaconda.com/download

Download the Python3 version and install

Overview

Lecture

- Classic NLP
- Deep Magic NLP
- Deep Learning Basics

Hands-on

- Environment Setup
- Deep Learning From Scratch

Classic NLP

Vector space model, Frequency, TF-IDF and PPMI

Classic NLP

"***You shall know a word by the company it keeps…***"
– John R. Firth (1957)

Classic NLP

"***You shall know a word by the company it keeps…***"
– John R. Firth (1957)

"***Everyone is about Firth 1957 (you shall know a word…). Somehow we all skipped Firth 1935***"
– Yoav Goldberg (2016)

Classic NLP

"***You shall know a word by the company it keeps…***"
– John R. Firth (1957)

"***Everyone is about Firth 1957 (you shall know a word…). Somehow we all skipped Firth 1935***"
– Yoav Goldberg (2016)

"***… the complete meaning of a word is always contextual, and no study of meaning apart from context can be taken seriously***"
– John R. Firth (1935)

Classic NLP

"***The frequencies of word in a document tend to indicate the relevance of document to a query***"
– Gerard Salton (1975)

"***From frequency to meaning…. Statistical patterns of human word usage can be used to figure out what people mean***"
– Turney and Pantel (2010)

Vector Space Model

- Vector space models are **numerical representation of text**
- Traditionally, computed using **no. of times each word occurs**

Frequency (Count Vector)

```
sent0 = "The quick brown fox jumps over the lazy brown dog ."
sent1 = "Mr brown jumps over the lazy fox ."
```

	brown	dog	fox	jumps	lazy	mr	over	quick	the
sent0	2	1	1	1	1	0	1	1	2
sent1	1	0	1	1	1	1	1	0	1

Term Frequency – Inverse Document Frequency

$$\mathrm{tf}_{t,d} = \begin{cases} 1 + \log_{10}\mathrm{count}(t,d) & \text{if count(t,d)} > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{idf}_i = \log_{10}\frac{N}{\mathrm{df}_i}$$

N ← total no. of sentences

df_i ← no. of sentences that contains word i

tf-idf value for word t in document d :

$$w_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t$$

```
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np
from scipy.sparse.csr import csr_matrix

# The *TfidfVectorizer* from sklearn expects list of strings as input.
sent0 = "The quick brown fox jumps over the lazy brown dog .".lower()
sent1 = "Mr brown jumps over the lazy fox .".lower()

dataset = [sent0, sent1]

vectorizer = TfidfVectorizer(input=dataset, analyzer='word',
                ngram_range=(1,1), min_df = 0, stop_words=None)
tfidf_matrix =  vectorizer.fit_transform(dataset)

# Format the TF-IDF table into the pd.DataFrame format.
vocab = vectorizer.get_feature_names()
documents_tfidf_lol = [{word:tfidf_value
                        for word, tfidf_value in zip(vocab, sent)}
                       for sent in tfidf_matrix.toarray()]
documents_tfidf = pd.DataFrame(documents_tfidf_lol)
documents_tfidf.fillna(0, inplace=True)
```

Term Frequency – Inverse Document Frequency

```
sent0 = "The quick brown fox jumps over the lazy brown dog ."

sent1 = "Mr brown jumps over the lazy fox ."
```

	brown	dog	fox	jumps	lazy	mr	over	quick	the
sent0	0.500	0.351	0.250	0.250	0.250	0.000	0.250	0.351	0.500
sent1	0.354	0.000	0.354	0.354	0.354	0.497	0.354	0.000	0.354

Classification (Train/Test Split)

Train

```
sent0 = "The quick brown fox jumps over the lazy brown dog ."
sent1 = "Mr brown jumps over the lazy fox ."
sent2 = "Roses are red , the chocolates are brown ."
sent3 = "The frank dog jumps through the red roses ."
```

Test

```
sent4 = "Mr Tan jumps on red chocolates ?"
sent5 = "Mr brown likes the lazy dog ."
```

```
# Train sentences.
sent0, label0 = "The quick brown fox jumps over the lazy brown dog .".lower() , False
sent1, label1 = "Mr brown jumps over the lazy fox .".lower(), True
sent2, label2 = "Roses are red , the chocolates are brown .".lower(), False
sent3, label3 = "The frank dog jumps through the red roses .".lower(), False

# Test sentences.
sent4, label4 = "Mr Tan jumps on red chocolates ?".lower(), False
sent5, label5 = "Mr brown likes the lazy dog .".lower(), True

train_documents = [(sent0, label0), (sent1, label1),
                   (sent2, label2), (sent0, label2)]
test_documents = [(sent4, label4), (sent5, label5)]

train_texts, train_labels = zip(*train_documents)
test_texts, test_labels = zip(*test_documents)
```

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the TF-IDF features.
48  vectorizer = TfidfVectorizer(input=train_texts, analyzer='word',
49                               ngram_range=(1,1), # Note this parameter!
50                               min_df = 0, stop_words=None)
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Apply TF-IDF to the train and test set

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the TF-IDF features.
48  vectorizer = TfidfVectorizer(input=train_texts, analyzer='word',
49                               ngram_range=(1,1), # Note this parameter!
50                               min_df = 0, stop_words=None)
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Apply the Machine Learning algorithm

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the TF-IDF features.
48  vectorizer = TfidfVectorizer(input=train_texts, analyzer='word',
49                               ngram_range=(1,1), # Note this parameter!
50                               min_df = 0, stop_words=None)
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Accuracy = 50%

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the TF-IDF features.
48  vectorizer = TfidfVectorizer(input=train_texts, analyzer='word',
49                               ngram_range=(2,2), # Note this parameter!
50                               min_df = 0, stop_words=None)
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Lets use TF-IDF with **bigrams**

Term Frequency – Inverse Document Frequency

```
sent0 = "The quick brown fox jumps over the lazy brown dog ."

sent1 = "Mr brown jumps over the lazy fox ."
```

	brown dog	brown fox	brown jumps	fox jumps	jumps over	...	mr brown	the lazy	the quick
sent0	0.364	0.364	0.000	0.364	0.259	...	0.000	0.259	0.364
sent1	0.000	0.000	0.470	0.000	0.334	...	0.470	0.334	0.000

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the TF-IDF features.
48  vectorizer = TfidfVectorizer(input=train_texts, analyzer='word',
49                               ngram_range=(2,2), # Note this parameter!
50                               min_df = 0, stop_words=None)
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Accuracy = 100%

But we sort of "*cheated*" and we engineered the TD-IDF *features*

Pointwise Mutual Information

Does the w_1 and w_2 co-occur more than if they were independent?

$$\mathrm{PMI}(w_1, w_2) = \log_2 \frac{P(w_1, w_2)}{P(w_1)P(w_2)}$$

Pointwise Mutual Information

Does the w_1 and w_2 co-occur more than if they were independent?

$$\mathrm{PMI}(w_1, w_2) = \log_2 \frac{P(w_1, w_2)}{P(w_1)P(w_2)}$$

$$P_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \quad P_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \quad P_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \quad PMI_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

Co-occurrence Matrix

```
sent1 = "Mr brown jumps over mr fox ."
```

	mr	brown	jumps	over	fox
mr					
brown					
jumps					
over					
fox					

Co-occurrence Matrix

```
sent1 = "Mr brown jumps over mr fox ."
```

	mr	brown	jumps	over	fox
mr		1	1		
brown					
jumps					
over					
fox					

Co-occurrence Matrix

```
sent1 = "Mr brown jumps over mr fox ."
```

	mr	brown	jumps	over	fox
mr		1	1		
brown			1	1	
jumps					
over					
fox					

Co-occurrence Matrix

```
sent1 = "Mr brown jumps over mr fox ."
```

	mr	brown	jumps	over	fox
mr		1	1		
brown			1	1	
jumps	1			1	
over					
fox					

Co-occurrence Matrix

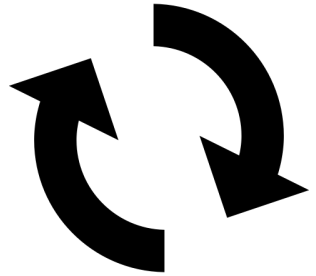
```
sent1 = "Mr brown jumps over mr fox ."
```

	mr	brown	jumps	over	fox
mr		1	1		
brown			1	1	
jumps	1			1	
over	1				1
fox					

Co-occurrence Matrix

```
sent2 = "Fox jumps over mr brown ."
```

	mr	brown	jumps	over	fox
mr		1	1		
brown			1	1	
jumps	1			1	
over	1				1
fox					



Repeat the same process for all sentences and add to the counts in the cells

Pointwise Mutual Information

		$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = 5$
		mr	brown	jumps	over	fox
$i = 1$	mr	0	1	1	0	0
$i = 2$	brown	0	0	1	1	2
$i = 3$	jumps	1	0	0	2	1
$i = 4$	over	2	1	0	0	1
$i = 5$	fox	0	0	1	0	0

Matrix F with
- W rows (words)
- C columns (context)

Pointwise Mutual Information

		$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
		mr	brown	jumps	over	fox
$i=1$	mr	0	1	1	0	0
$i=2$	brown	0	0	1	1	2
$i=3$	jumps	1	0	0	2	1
$i=4$	over	2	1	0	0	1
$i=5$	fox	0	0	1	0	0

Matrix F with
- W rows (words)
- C columns (context)

$$\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij} = 15$$

Pointwise Mutual Information

		$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
		mr	brown	jumps	**over**	fox
$i=1$	mr	0	1	1	0	0
$i=2$	brown	0	0	1	1	2
$i=3$	**jumps**	1	0	0	**2**	1
$i=4$	over	2	1	0	0	1
$i=5$	fox	0	0	1	0	0

Matrix F with
- W rows (words)
- C columns (context)

$$P_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$$\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij} = 15$$

P(jumps, over) = 2 / 15

Pointwise Mutual Information

		$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
		mr	brown	jumps	over	fox
$i=1$	mr	0	1	1	0	0
$i=2$	brown	0	0	1	1	2
$i=3$	**jumps**	**1**	**0**	**0**	**2**	**1**
$i=4$	over	2	1	0	0	1
$i=5$	fox	0	0	1	0	0

Matrix F with
- W rows (words)
- C columns (context)

$$P_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$$\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij} = 15$$

$$\sum_{j=1}^{C} f_{ij} = 4$$

$P(jumps, over) = 2 / 15$

P(jumps) = 4 / 15

Pointwise Mutual Information

		$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
		mr	brown	jumps	over	fox
$i=1$	mr	0	1	1	0	0
$i=2$	brown	0	0	1	1	2
$i=3$	jumps	1	0	0	2	1
$i=4$	over	2	1	0	0	1
$i=5$	fox	0	0	1	0	0

$$\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij} = 15 \qquad \sum_{j=1}^{C} f_{ij} = 4 \qquad \sum_{i=1}^{W} f_{ij} = 3$$

Matrix F with
- W rows (words)
- C columns (context)

$$P_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$P(jumps, over) = 2 / 15$

$P(jumps) = 4 / 15$

P(over) = 3 / 15

Pointwise Mutual Information

	mr	brown	jumps	over	fox
mr	0	1	1	0	0
brown	0	0	1	1	2
jumps	1	0	0	1.321	1
over	2	1	0	0	1
fox	0	0	1	0	0

$$P_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \qquad P_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$$P_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \qquad PMI_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

P(jumps, over) = 2 / 15
P(jumps) = 4 / 15
P(over) = 3 / 15

***PMI(jumps, over)* = log (2/15) / (4/15 * 3/15) = 1.321**

Positive Pointwise Mutual Information

	mr	brown	jumps	over	fox
mr	0	1	1	0	0
brown	0	0	1	1	2
jumps	1	0	0	1.321	1
over	2	1	0	0	1
fox	0	0	1	0	0

P(jumps, over) = 2 / 15
P(jumps) = 4 / 15
P(over) = 3 / 15
PMI(jumps, over) = 1.321

$$P_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \qquad P_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$$P_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \qquad PMI_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

$$PPMI_{ij} = \begin{cases} PMI_{ij} & if \;\; PMI_{ij} > 0 \\ 0 & Otherwise \end{cases}$$

Pointwise Mutual Information

```
>>> import math

>>> math.log2(50/(100*200))
-8.643856189774725
```

$$PMI_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

Pointwise Mutual Information

```
>>> import math

>>> math.log2(50/(100*200))
-8.643856189774725

>>> math.log2(5/(10*20))
-5.321928094887363
```

$$PMI_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

(*higher is better*)

If we simply plug-in words with similar ratio

Pointwise Mutual Information

```
>>> import math

>>> math.log2(50/(100*200))
-8.643856189774725

>>> math.log2(5/(10*20))
-5.321928094887363

>>> math.log2(1/(2*4))
-3.0
```

$$PMI_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

(*higher is better*)

With the same ratio, PMI is biased towards rare words!!

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the TF-IDF features.
48  vectorizer = TfidfVectorizer(input=train_texts, analyzer='word',
49                               ngram_range=(2,2), # Note this parameter!
50                               min_df = 0, stop_words=None)
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the PPMI features.
48  vectorizer =PPMIVectorizer(input=train_texts, analyzer='word',
49                             context_window=2, stop_words=None)
50
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Replace the TF-IDF features with PPMI*

* There's no PPMIVectorizer in sklearn, you have to write it yourself

```
43  from sklearn.feature_extraction.text import TfidfVectorizer
44  import numpy as np
45  from scipy.sparse.csr import csr_matrix
46
47  # Extract the PPMI features.
48  vectorizer =PPMIVectorizer(input=train_texts, analyzer='word',
49                             context_window=2, stop_words=None)
50
51
52  X_train = vectorizer.fit_transform(train_texts)
53  X_test =  vectorizer.transform(test_texts)
54
55  y_train = train_labels
56  y_test = test_labels
57
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Machine learning block often don't change much

Classic NLP: Feature Engineering

- **Raw frequency is useful**
  - *but frequent words non-content words are not very informative*
- **TF-IDF resolves high freq non-content words issue**
  - *but each word is still somewhat independent of each other*
- **PPMI provides information about whether a word is informative in the context of another word**
  - *but biased towards infrequent events*

Classic NLP: Feature Engineering

- **TF-IDF and PPMI vectors are**
  - long (|V| > 100,000)
  - sparse (lots of zero)

- **Deep learning can create vectors that are**
  - short (often fixed-sized <2000, decided empirically)
  - dense (most are non-zeros)

- **But it's not unlike *'modern'* deep learning based NLP**
  - one model improves upon another often incremental
  - they always come with certain caveats

Deep 'Magic' NLP

Recent advancement in Deep Learning in NLP

Classic NLP

"***The frequencies of word in a document tend to indicate the relevance of document to a query***"
– Gerard Salton (1975)

"***From frequency to meaning…. Statistical patterns of human word usage can be used to figure out what people mean***"
– Turney and Pantel (2010)

Deep 'Magic' NLP

"***The frequencies of word in a document tend to indicate the relevance of document to a query***"
– Gerard Salton (1975)

"***From frequency to meaning…. Statistical patterns of human word usage can be used to figure out what people mean***"
– Turney and Pantel (2010)

"We propose a unified NN architecture by trying to avoid task-specific engineering therefore disregarding a lot of prior knowledge"
– Collobert and Weston (2011)

Definitions (NLP, ML, DL)

- **Natural Language Processing (NLP) is …**
  - making computers understand and produce human languages.

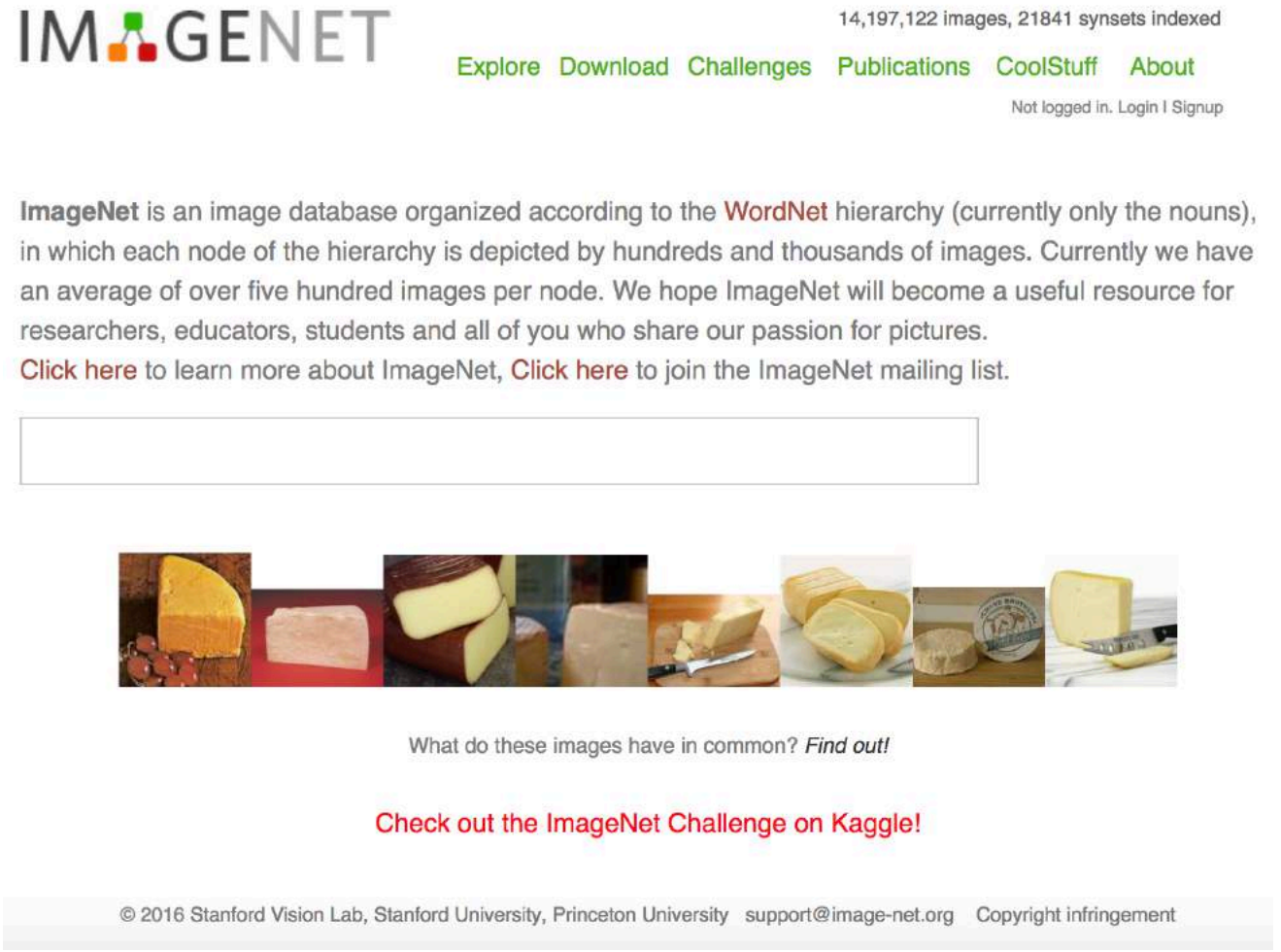
- **Machine Learning is …**
  - optimizing parameters/weights to best make a decision
  - *well-defined counting*

- **Deep Learning, some people say it's …**
  - neural nets
  - stacking multiple layers of "representation learning"
  - something that burns up as much GPUs as Bitcoin mining
  - a subset of methods in machine learning

The "ImageNet" Moment

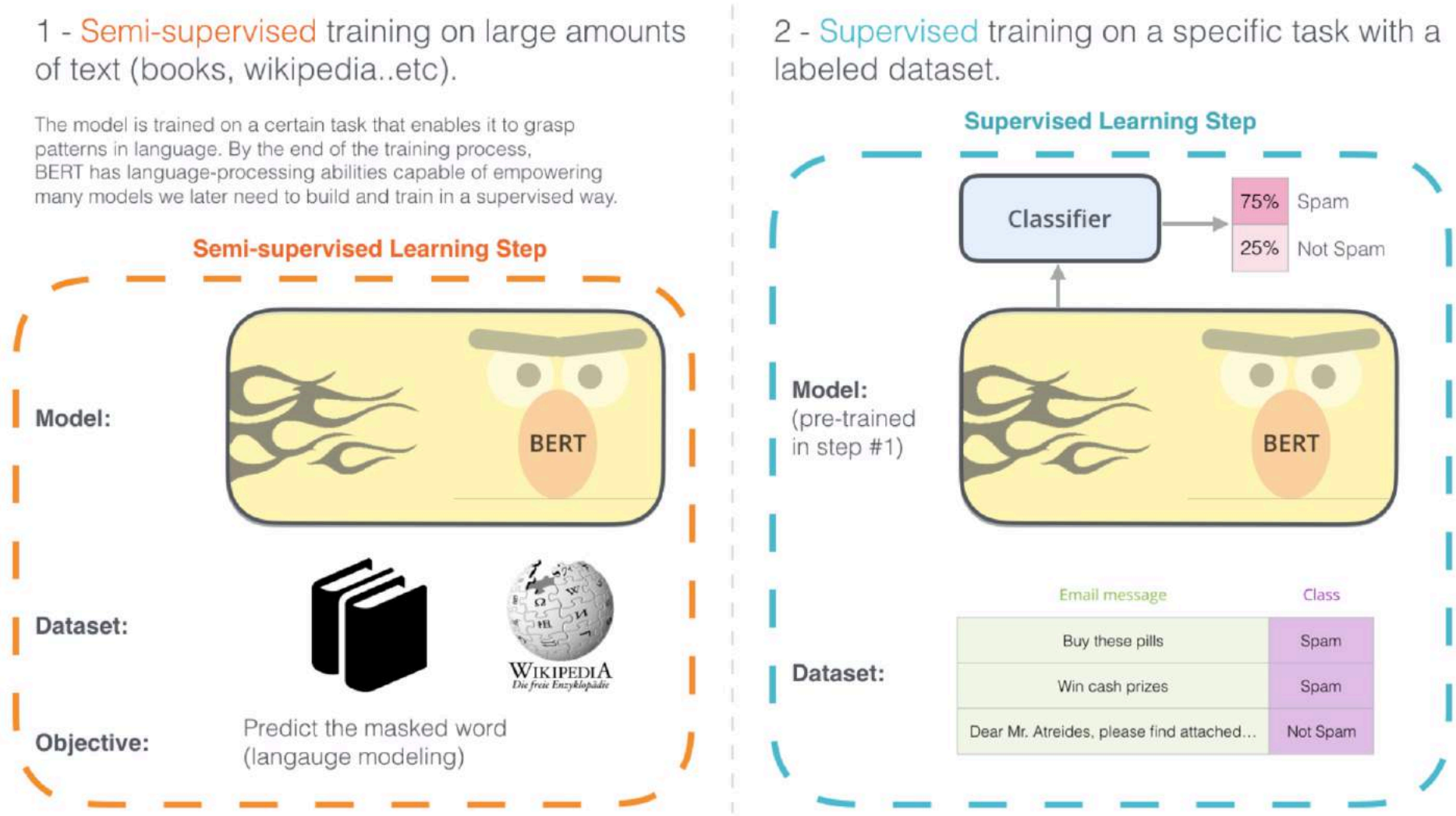
Various Computer Vision Challenges (ImageNet, MS Coco, etc.) started a wave of groups **training models and sharing pre-trained models.**

Fine-tuning / transfer learning for these pre-trained models are faster and *"usually better"* when training new models for other computer vision task.



(*Image from Stanford Vision Lab)

The "ImageNet" Moment for NLP



The two steps of how BERT is developed. You can download the model pre-trained in step 1 (trained on un-annotated data), and only worry about fine-tuning it for step 2. [Source for book icon].

(*Image from Jay Alammar's blog)

The "ImageNet" Moment for NLP

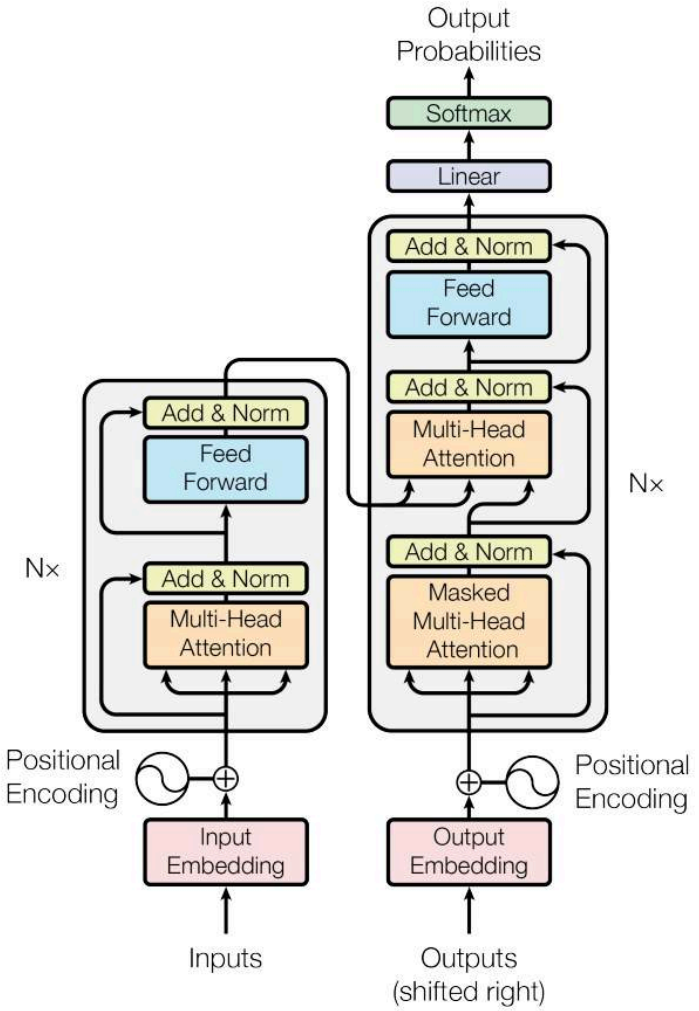


Figure 1: The Transformer - model architecture.

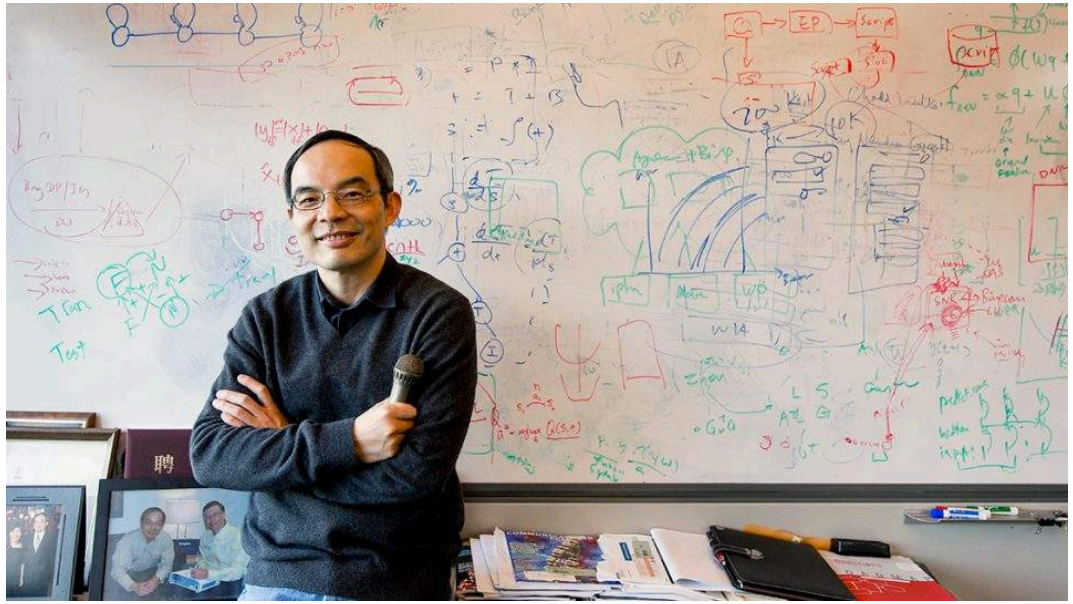
(*Image from Vaswani et al. 2017)

Major breakthrough of BERT came through the **self-attention network architecture, aka. Transformer** (Vaswani et al. 2017)

But do note the caveats with transfer-learning (aka. pre-training and fine-tuning):

- Could get **same results from random initialization** vs pre-training counterparts (He et al. 2018 on "*Rethinking ImageNet*")
- Understanding **why pre-training works in NLP still unclear** (Goldberg, 2018, see also Erhan, 2010)

Machine Translation Achieved Human Parity



(*Image from Microsoft AI Blog)

"Microsoft reaches a historic milestone, using AI to match human performance in translating news from Chinese to English"

- Microsoft AI Blog

Definition of Human Parity

If there is ***no statistically significant difference*** between human quality scores for … machine translation … and the scores for the corresponding human translations then the machine has achieved human parity.

Did MT really achieve Human Parity?

- **Has Machine Translation Achieved Human Parity? A Case for Document-level Evaluation**
  - Document level evaluation is necessary
  - http://aclweb.org/anthology/D18-1512

- **What Level of Quality can Neural Machine Translation Attain on Literary Text?**
  - Only 17-34% of novels can be machine translated
  - https://link.springer.com/chapter/10.1007/978-3-319-91241-7_12

Did MT really achieve Human Parity?

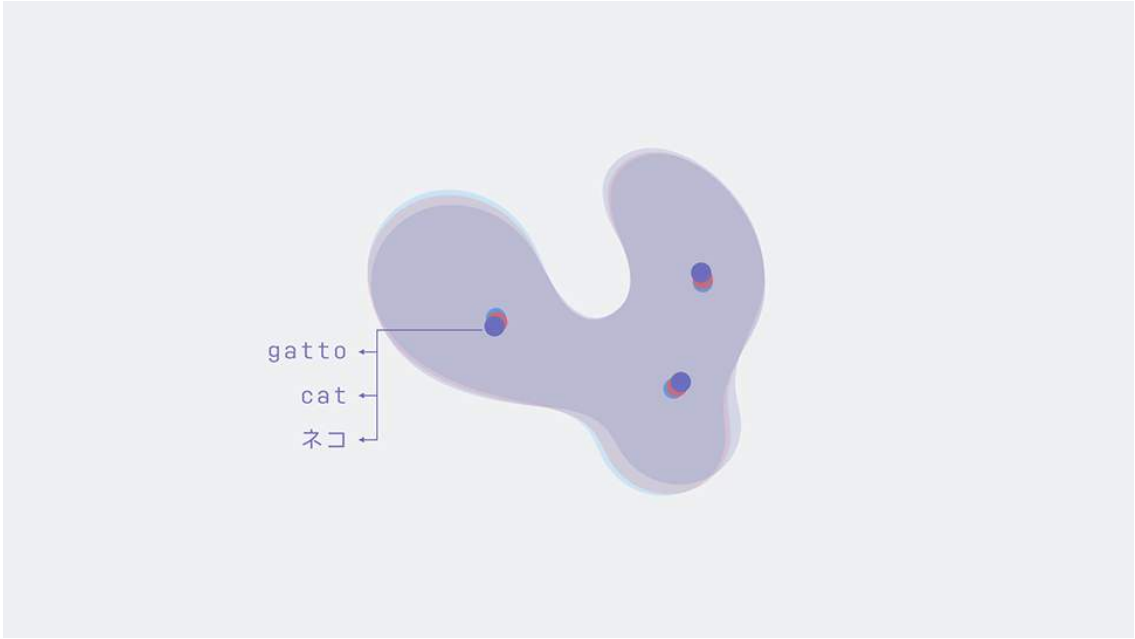
- **Quality expectations of machine translation**
  - *"those of us who have seen many paradigms come and go know that overgilding the lily does none of us any good"*
  - https://arxiv.org/pdf/1803.08409.pdf

- **"Human parity" in machine translation**
  - *"Some aspects of such translations are very good, but the frequent mistakes spoil things."*
  - http://languagelog.ldc.upenn.edu/nll/?p=40602

Machine Translation Achieved Human Parity

Algorithm 1: Unsupervised MT

1 **Language models:** Learn language models P_s and P_t over source and target languages;
2 **Initial translation models:** Leveraging P_s and P_t, learn two initial translation models, one in each direction: $P^{(0)}_{s\to t}$ and $P^{(0)}_{t\to s}$;
3 **for** k=1 **to** N **do**
4 **Back-translation:** Generate source and target sentences using the current translation models, $P^{(k-1)}_{t\to s}$ and $P^{(k-1)}_{s\to t}$, factoring in language models, P_s and P_t;
5 Train new translation models $P^{(k)}_{s\to t}$ and $P^{(k)}_{t\to s}$ using the generated sentences and leveraging P_s and P_t;
6 **end**



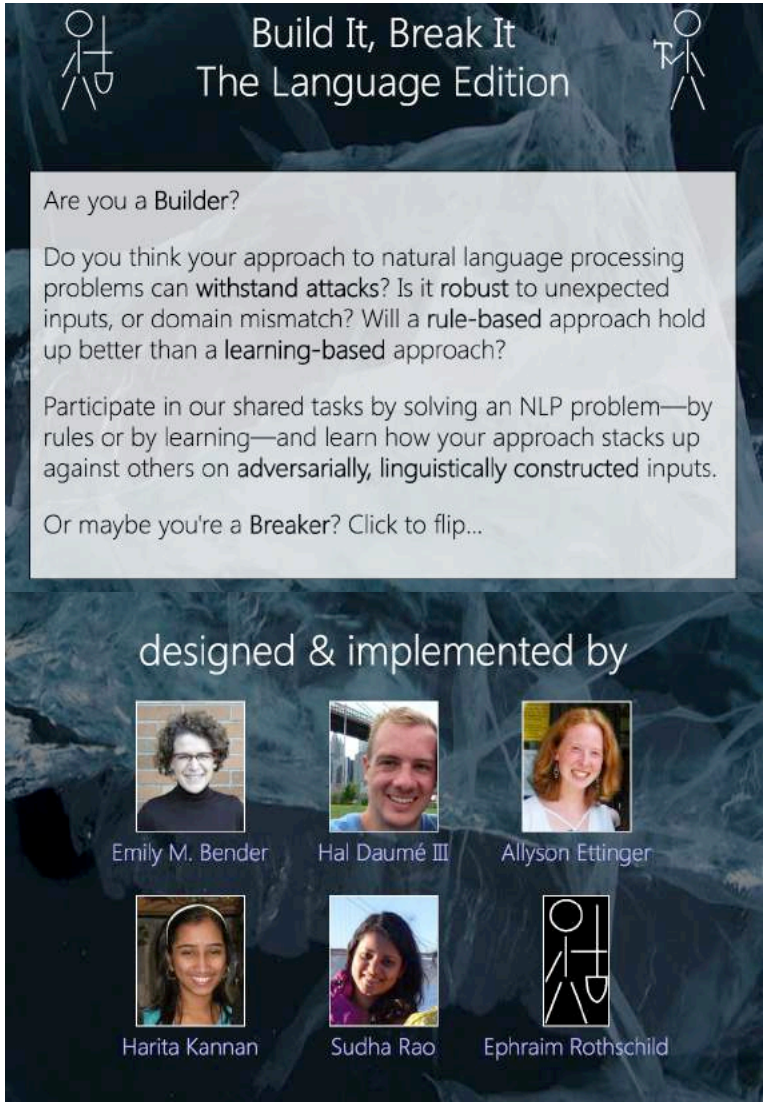
(*Image from Facebook Code Blog)

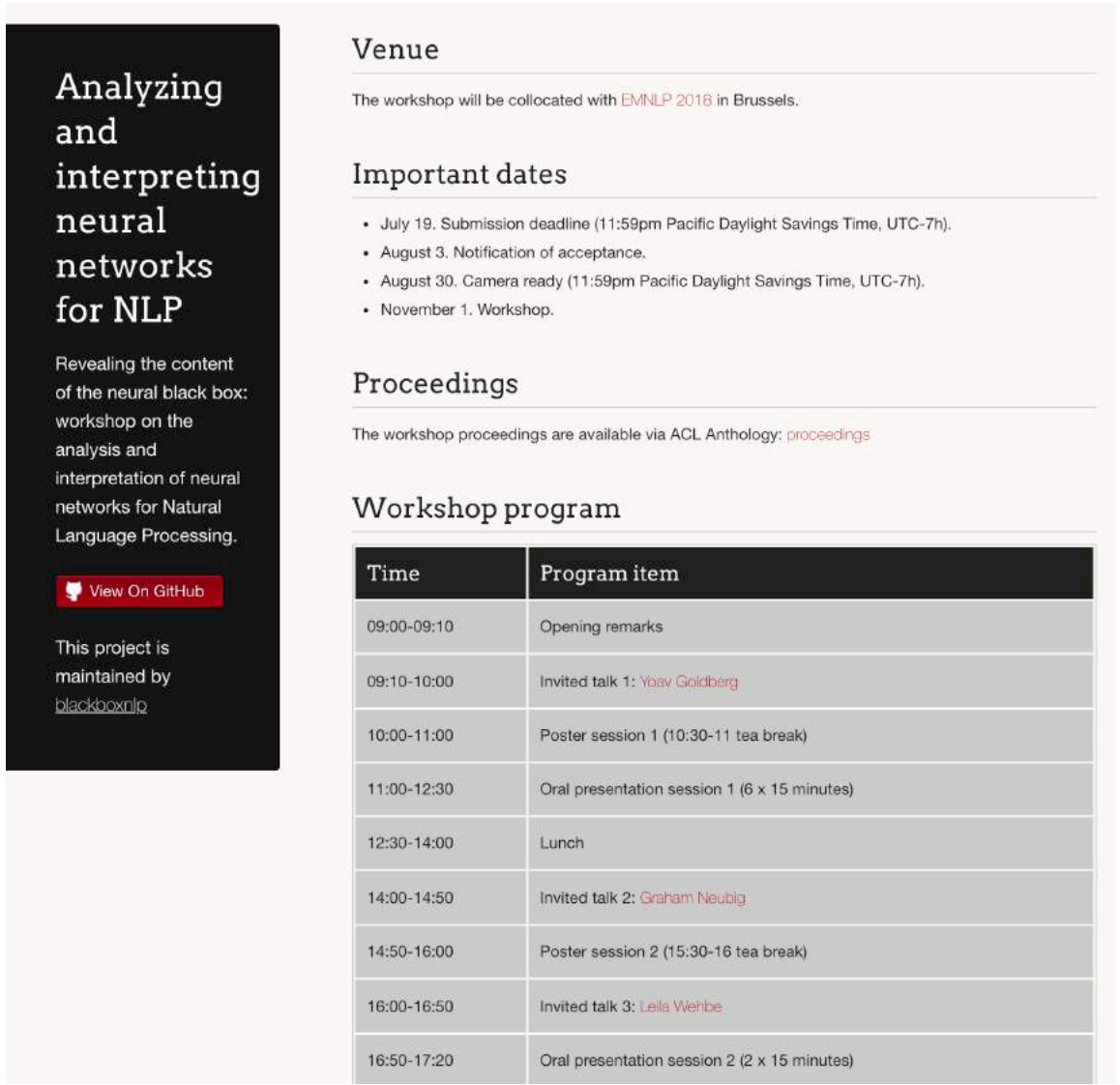
"Two-dimensional word embeddings in two languages can be aligned via a simple rotation. After the rotation, word translation is performed via nearest neighbor search."
– *Facebook Code Blog*

Unsupervised Machine Translation

- **Unsupervised SMT (IXA NLP Group)**
  - Non-neural: http://aclweb.org/anthology/D18-1399
- **Unsupervised NMT (IXA NLP and Cho)**
  - https://arxiv.org/pdf/1710.11041.pdf
- **Unsupervised NMT with weights sharing (UCAS)**
  - http://www.aclweb.org/anthology/P18-1005
- **Lots of unsupervised MT tech at Facebook at EMNLP 2018**
  - https://research.fb.com/facebook-research-at-emnlp/

Robustness and Interpretability





Deep Learning Basics

Recent advancement in Deep Learning in NLP

Perceptron

Perceptron algorithm is a:

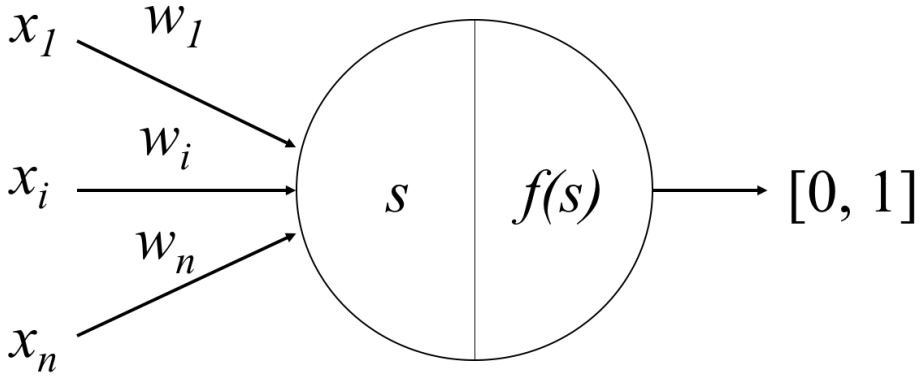
> "*system that depends on **probabilistic** rather than deterministic principles for its operation, gains its reliability from the **properties of statistical measurements obtain from a large population of elements***"
> - Frank Rosenblatt (1957)

```
58  # Pick your poison.
59  from sklearn.linear_model import Perceptron
60  # Initialize your classifier.
61  clf = Perceptron(max_iter=10)
62  # Train the classifier.
63  clf.fit(X_train, y_train)
64
65  print(clf.predict(X_test))
```

Perceptron

Given a **set of inputs x**, perceptron

- learns ***w* vector** to map the inputs to a real-value output between $[0,1]$
- through the **summation of the dot product of the *w*·*x***
- with a **transformation function** (aka. activation function)



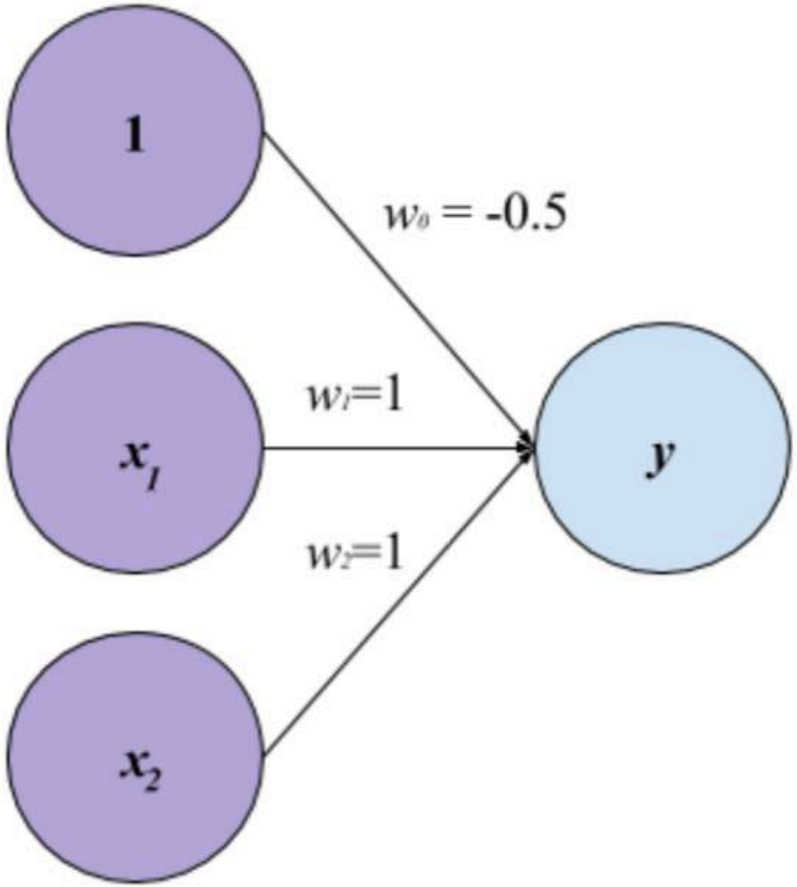
Summation

$$s = \sum w \cdot x$$

Transformation

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

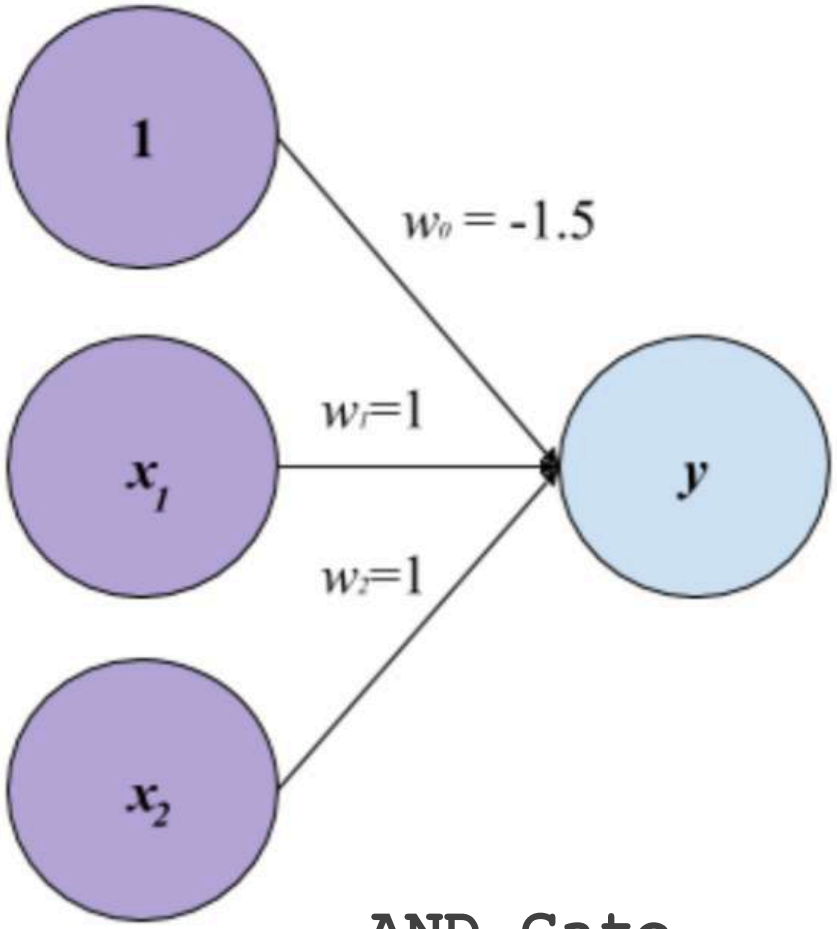
Perceptron



OR Gate

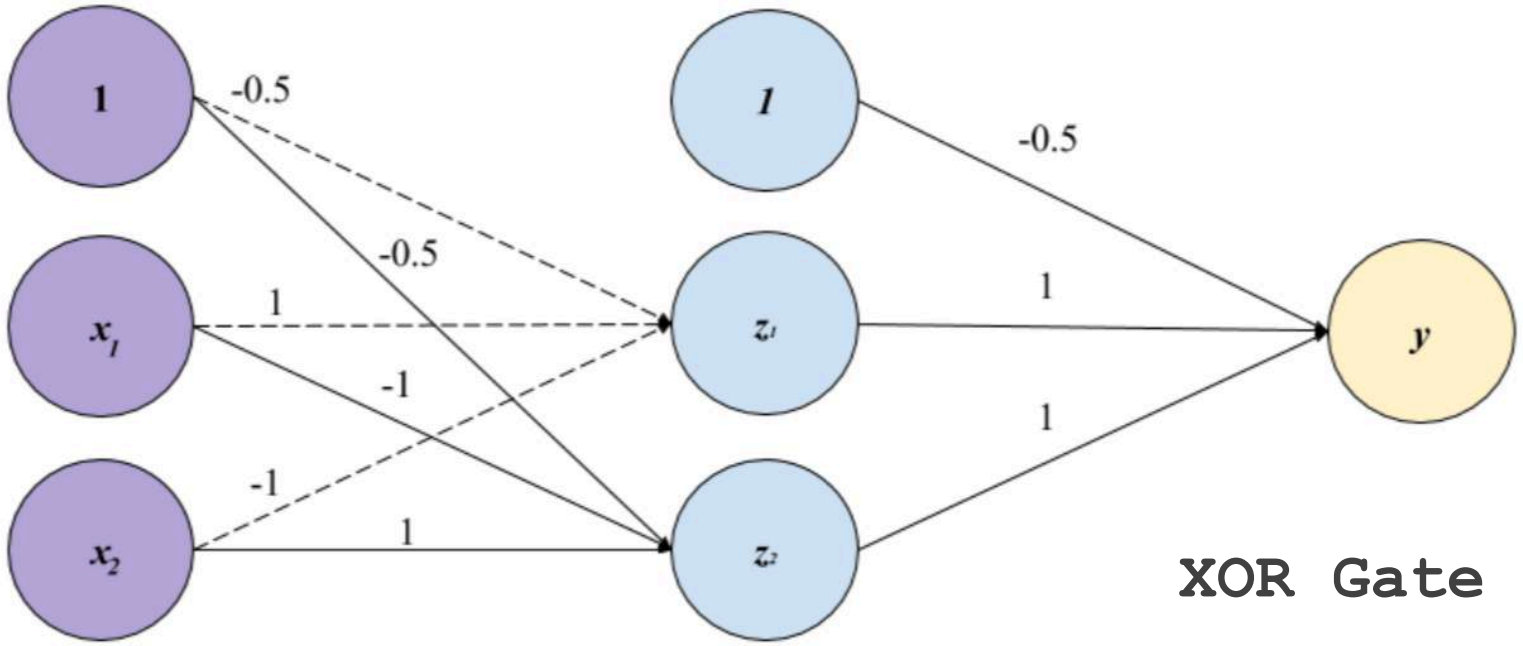
x_1	x_2	$sum = w_0*1 + w_1*x_1 + w_2*x_2$	$y = 1$ if $sum > 0$ $y = 0$ if $sum <= 0$
0	0	-0.5*1 + 1*0 + 1*0 = -0.5	0
0	1	-0.5*1 + 1*0 + 1*1 = 0.5	1
1	0	-0.5*1 + 1*1 + 1*0 = 0.5	1
1	1	-0.5*1 + 1*1 + 1*1 = 1.5	1

Perceptron



AND Gate

x_1	x_2	$sum = w_0*1 + w_1*x_1 + w_2*x_2$	$y = 1$ if $sum > 0$ $y = 0$ if $sum <= 0$
0	0	-1.5*1 + 1*0 + 1*0 = -1.5	0
0	1	-1.5*1 + 1*0 + 1*1 = -0.5	0
1	0	-1.5*1 + 1*1 + 1*0 = -0.5	0
1	1	-1.5*1 + 1*1 + 1*1 = 0.5	1



XOR Gate

x_1	x_2	$sum_1 = w_0*1 + w_1*x_1 + w_2*x_2$	z_1	$sum_2 = w_0*1 + w_1*x_1 + w_2*x_2$	z_2
0	0	-0.5*1 + 1*0 + -1*0 = -0.5	0	-0.5*1 + -1*0 + 1*0 = -0.5	0
0	1	-0.5*1 + 1*0 + -1*1 = -1.5	0	-0.5*1 + -1*0 + 1*1 = 0.5	1
1	0	-0.5*1 + 1*1 + -1*0 = 0.5	1	-0.5*1 + -1*1 + 1*0 = -1.5	0
1	1	-0.5*1 + 1*1 + -1*1 = -0.5	0	-0.5*1 + -1*1 + 1*1 = -0.5	0

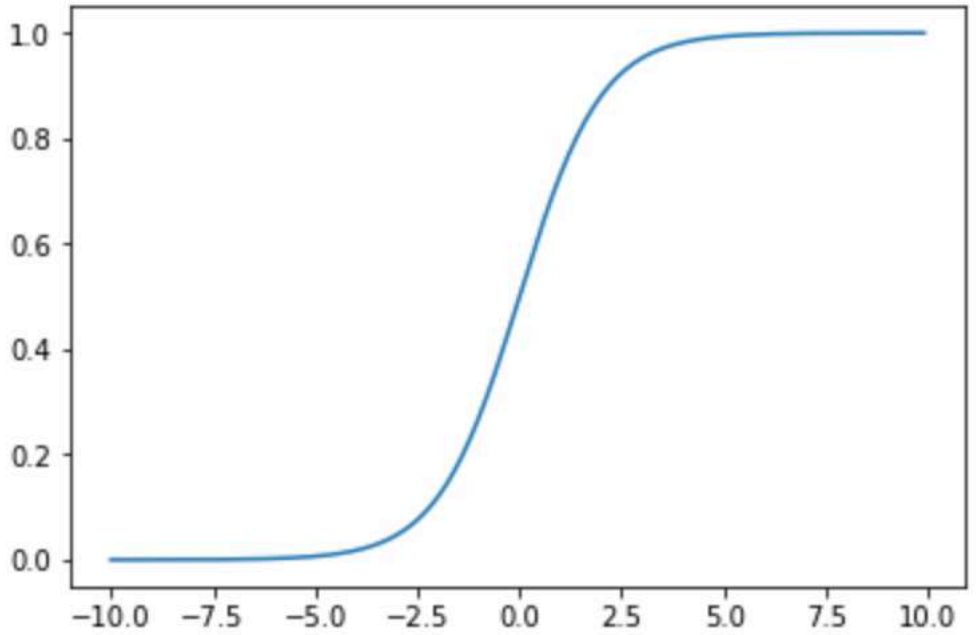
z_1	z_2	$sum_1 = w_0*1 + w_1*x_1 + w_2*x_2$	y
0	0	-0.5*1 + 1*0 + 1*0 = -0.5	0
0	1	-0.5*1 + 1*0 + 1*1 = 0.5	1
1	0	-0.5*1 + 1*1 + 1*0 = 0.5	1
0	0	-0.5*1 + 1*0 + 1*0 = -0.5	0

Figure 2: Emulate an XOR Gate with Feed-forward Network

Multi Layered Perceptron

Activation Function (Sigmoid)

```
3   import numpy as np
4   import matplotlib.pyplot as plt
5
6   def sigmoid(x):
7       return 1/(1+np.exp(-x))
8
9   # Generate points from -10 to +10,
10  # in steps of 0.1
11  x = np.arange(-10, 10, 0.1)
12  y = sigmoid(x)
13
14  # Plot the graph.
15  plt.plot(x, y)
16  plt.show()
```



Loss Function (aka. Criterion)

For regression problems, the simplest loss function is to simply take the difference between the predictions and the truth value, i.e. the L1 Loss / Mean Absolute Error (MAE)

Loss Function (aka. Criterion)

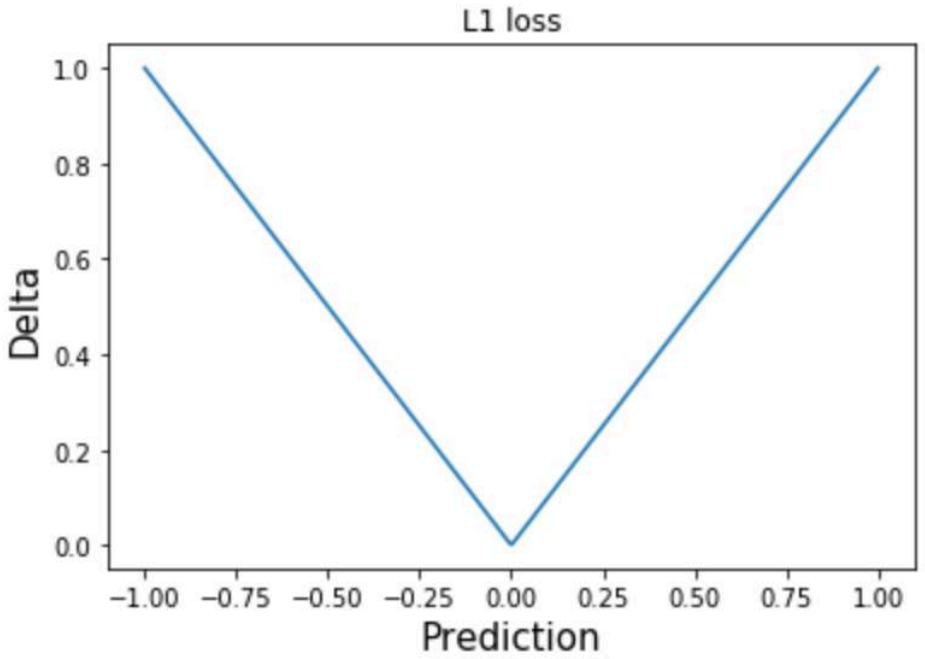
```
38  import numpy as np
39
40  # Create 500 points betwen -1 and 1.
41  predictions = np.linspace(-1, 1., 500)
42  # Set truth to be the constant 0.
43  truth = np.zeros(500)
44  # Calcuate the absolute differences
45  delta = np.abs(truth - predictions)
46
47  # Plotting magic
48  plt.plot(predictions, delta, 'b-', label='L1 loss' )
49  plt.title('L1 loss')
50  plt.xlabel('Prediction', fontsize=15)
51  plt.ylabel('Delta', fontsize=15)
```

For regression problems, the simplest loss function is to take the diff between predictions and the truth value, i.e. the L1 Loss / Mean Absolute Error (MAE)

Loss Function (aka. Criterion)

```
38  import numpy as np
39
40  # Create 500 points betwen -1 and 1.
41  predictions = np.linspace(-1, 1., 500)
42  # Set truth to be the constant 0.
43  truth = np.zeros(500)
44  # Calcuate the absolute differences
45  delta = np.abs(truth - predictions)
46
47  # Plotting magic
48  plt.plot(predictions, delta, 'b-', label='L1 loss' )
49  plt.title('L1 loss')
50  plt.xlabel('Prediction', fontsize=15)
51  plt.ylabel('Delta', fontsize=15)
```

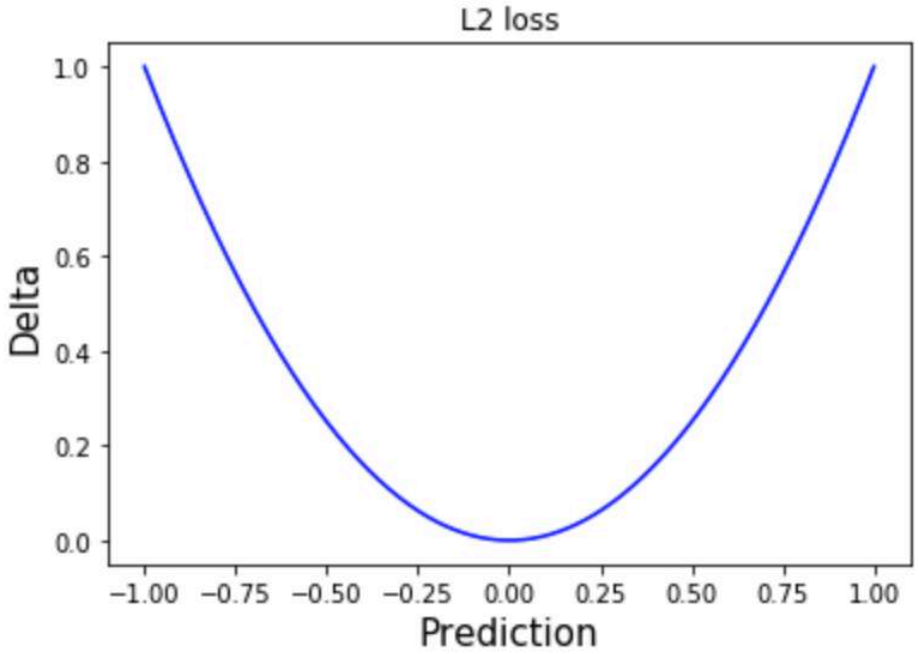
$$L_{mae} = \frac{1}{n} \sum_{i=1}^{n} |y - \hat{y}|$$



Loss Function (aka. Criterion)

```
38  import numpy as np
39
40  # Create 500 points betwen -1 and 1.
41  predictions = np.linspace(-1, 1., 500)
42  # Set truth to be the constant 0.
43  truth = np.zeros(500)
44  # Calcuate the absolute differences
45  delta = np.abs((truth - predictions)**2)
46
47  # Plotting magic
48  plt.plot(predictions, delta, 'b-', label='L2 loss' )
49  plt.title('L2 loss')
50  plt.xlabel('Prediction', fontsize=15)
51  plt.ylabel('Delta', fontsize=15)
```

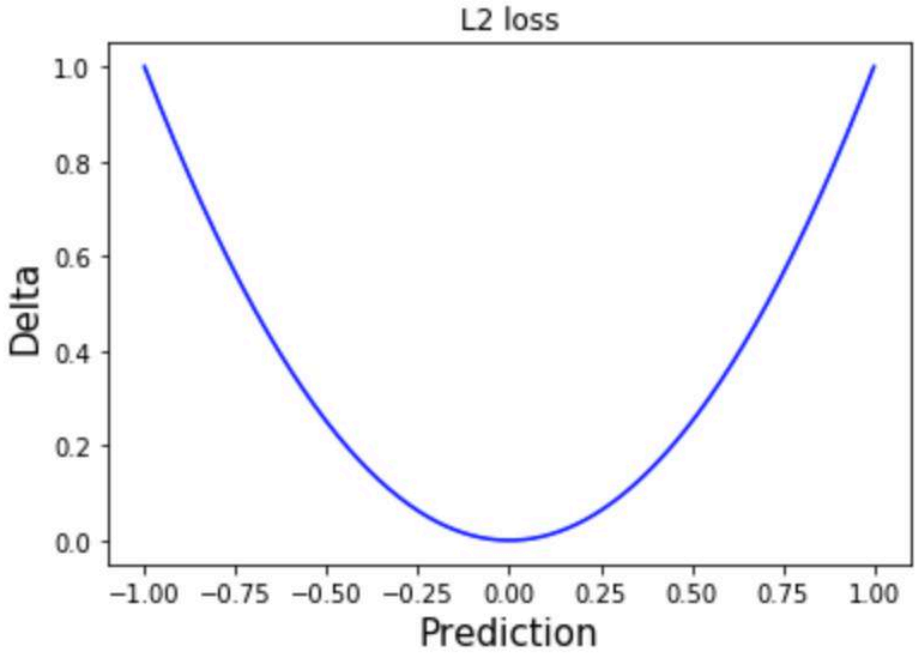
$$L_{mse} = \frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2$$



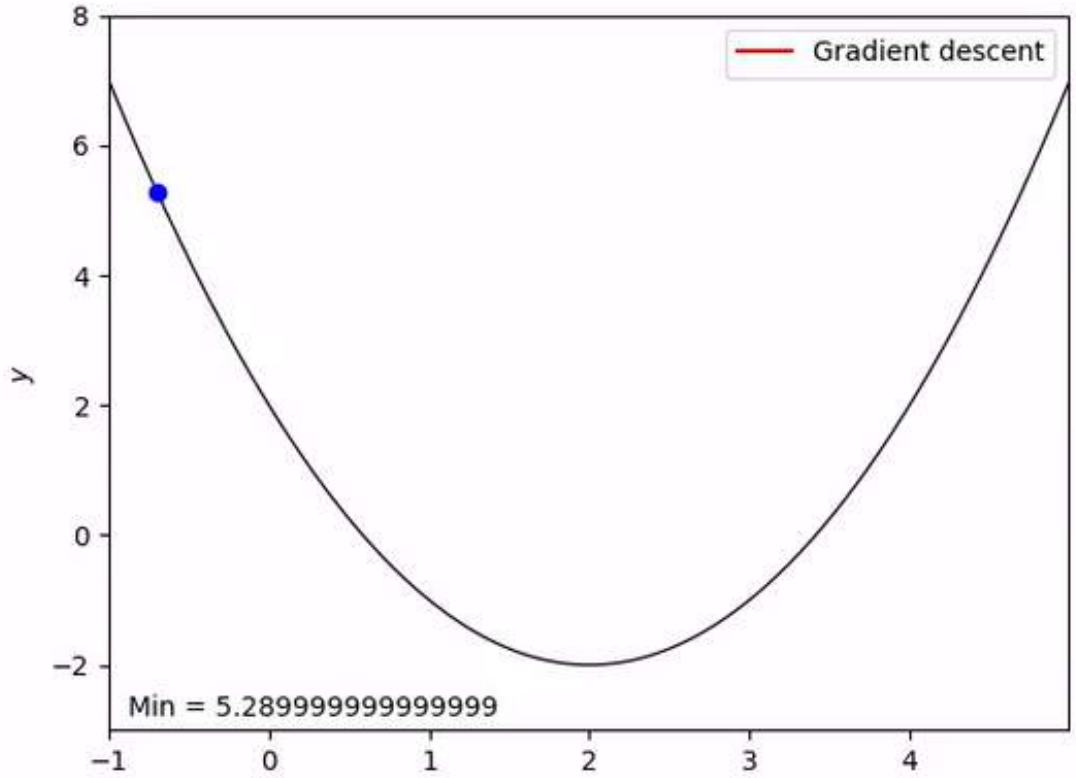
Loss Function (aka. Criterion)

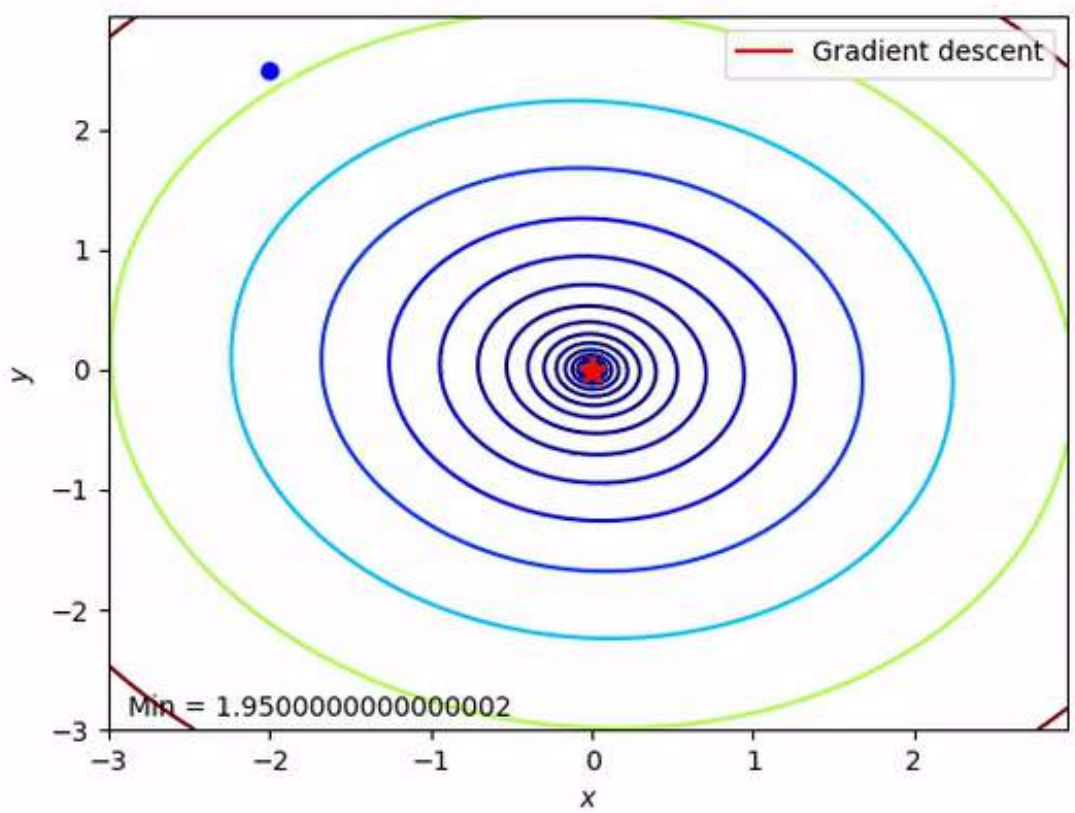
```
38  import numpy as np
39
40  # Create 500 points betwen -1 and 1.
41  predictions = np.linspace(-1, 1., 500)
42  # Set truth to be the constant 0.
43  truth = np.zeros(500)
44  # Calcuate the absolute differences
45  delta = np.abs((truth - predictions)**2)
46
47  # Plotting magic
48  plt.plot(predictions, delta, 'b-', label='L2 loss' )
49  plt.title('L2 loss')
50  plt.xlabel('Prediction', fontsize=15)
51  plt.ylabel('Delta', fontsize=15)
```

$$L_{mse} = \frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2$$



Optimization (Gradient Descent)





(Images from https://jed-ai.github.io/py1_gd_animation/)
It has some cool code to generate the GD pictures

Training Routine

Typically, process performs the following 4 steps iteratively.

Initialization

1. Initialize weights vector

Forward Propagation

2a. Multiply the weights vector with the inputs, sum the products.
2b. Put the sum through the activation function, e.g. sigmoid

Training Routine

Back Propagation

3a. Compute the errors, i.e. difference between expected output and predictions
3b. Multiply the error with the derivatives to get the delta
3c. Multiply the delta vector with the inputs, sum the product

Optimizer takes a step

4. Multiply the learning rate with the output of step 3c

Repeat 1-4 until *desired*

Summary

Summary

- **Classic NLP == Lots of feature engineering**
  - Frequency, TF-IDF, PPMI
- **Deep 'Magic' NLP**
  - Transfer Learning moment for NLP
  - NMT achieving human parity & Unsupervised MT
  - Robustness and Interpretability
- **Deep Learning Basics**
  - Perceptron and Multi-layered Perceptron
  - Activation function, Loss function, Gradient Descent
  - Training routine

Environment Setup

Open Anaconda Navigator.

Go to the PyTorch installation page, copy the command as per configuration: https://pytorch.org/get-started/locally/

Fire up the terminal in Anaconda Navigator.

Start a Jupyter Notebook.

Download https://goo.gl/fyZv3m

Import the .ipynb to the Jupyter Notebook

Fin

Neural Nets from Scratch

(with some numpy magic)

In [1]:
```
import sys
sys.executable
```

Out[1]:
```
'/usr/local/bin/python3'
```

In [2]:
```
!which pip3
!which python3
!ls -lah /usr/local/bin/python3
```

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/pip3
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3
lrwxr-xr-x  1 liling.tan  admin    69B Dec 14  2017 /usr/local/bin/python3 -> ../../../Librar
y/Frameworks/Python.framework/Versions/3.6/bin/python3
```

In [3]:
```
!pip3 install torch==1.0.0
```

```
Requirement already satisfied: torch==1.0.0 in /Library/Frameworks/Python.framework/Versions/
3.6/lib/python3.6/site-packages (1.0.0)
```

In [4]:
```
import torch
torch.cuda.is_available()
```

Out[4]:
```
False
```

These are just some IPython functions for showing image and making things pretty

In [1]:
```
# IPython candies...
from IPython.display import Image
from IPython.core.display import HTML

from IPython.display import clear_output
```

In [2]:
```
%%html
<style> table {float:left} </style>
```

Perceptron

Perceptron algorithm is a:

> "*system that depends on **probabilistic** rather than deterministic principles for its operation, gains its reliability from the **properties of statistical measurements obtain from a large population of elements***" - Frank Rosenblatt (1957)

Then the news:

> "*[Perceptron is an] **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.***" - The New York Times (1958)

News quote cite from Olazaran (1996)

Perceptron in Bullets

- Perceptron learns to classify any linearly separable set of inputs.
- Some nice graphics for perceptron with Go https://appliedgo.net/perceptron/

If you've got some spare time:

- There's a whole book just on perceptron: https://mitpress.mit.edu/books/perceptrons
- For watercooler gossips on perceptron in the early days, read Olazaran (1996)

Perceptron in Math

Given a set of inputs x, the perceptron

- learns w vector to map the inputs to a real-value output between $[0, 1]$
- through the summation of the dot product of the $w \cdot x$ with a transformation function

Perceptron as a Workflow Diagram

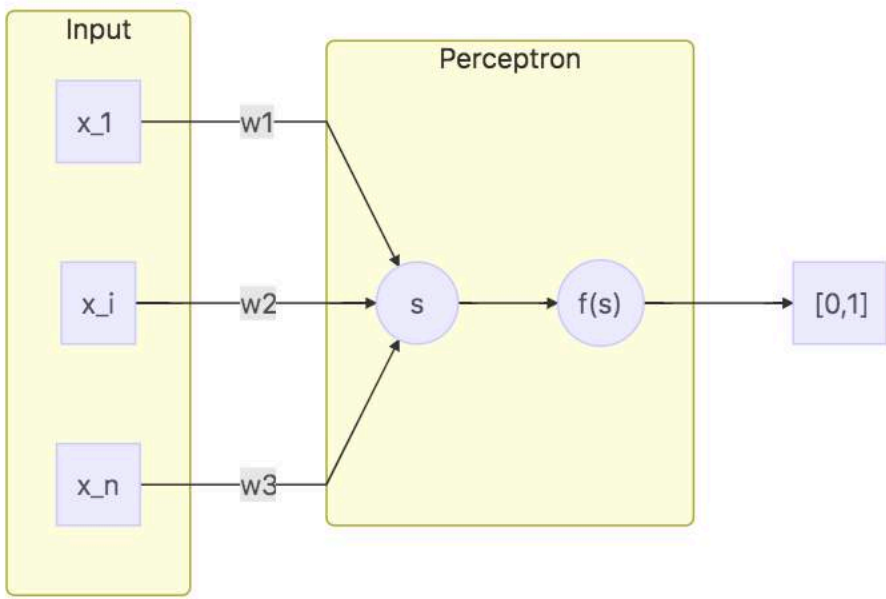
If you're familiar with mermaid flowchart

```
.. mermaid::

    graph LR
        x_1 -->|w1| n1((s))
        x_i -->|w2| n1((s))
        x_n -->|w3| n1((s))
        subgraph Perceptron
            n1 --> n2(("f(s)"))
        end
        n2 --> y["[0,1]"]
```

In [6]:
```
##Image(url="perceptron-mermaid.svg", width=500)
Image(url="https://svgshare.com/i/AbJ.svg", width=500)
```

Out[6]:



Optimization Process

To learn the weights, w, we use an **optimizer** to find the best-fit (optimal) values for w such that the inputs correct maps to the outputs.

Typically, process performs the following 4 steps iteratively.

Initialization

- **Step 1**: Initialize weights vector

Forward Propagation

- **Step 2a**: Multiply the weights vector with the inputs, sum the products, i.e. `s`
- **Step 2b**: Put the sum through the sigmoid, i.e. `f()`

Back Propagation

- **Step 3a**: Compute the errors, i.e. difference between expected output and predictions
- **Step 3b**: Multiply the error with the **derivatives** to get the delta
- **Step 3c**: Multiply the delta vector with the inputs, sum the product

Optimizer takes a step

- **Step 4**: Multiply the learning rate with the output of Step 3c.

In [62]:
```
import math
import numpy as np
np.random.seed(0)

def sigmoid(x): # Returns values that sums to one.
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(sx):
    # See https://math.stackexchange.com/a/1225116
    # Hint: let sx = sigmoid(x)
    return ???
```

Fill in the blanks =)

In [9]:
```
sigmoid(np.array([2.5, 0.32, -1.42]))          # [out]: array([0.92414182, 0.57932425, 0.19466158])
```

Out[9]:
```
array([0.92414182, 0.57932425, 0.19466158])
```

In [10]:
```
sigmoid_derivative(np.array([2.5, 0.32, -1.42]))  # [out]: array([0.07010372, 0.24370766, 0.15676845])
```

Out[10]:
```
array([0.07010372, 0.24370766, 0.15676845])
```

In [62]:
```
import math
import numpy as np
np.random.seed(0)
```

```
def sigmoid(x): # Returns values that sums to one.
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(sx):
    # See https://math.stackexchange.com/a/1225116
    # Hint: let sx = sigmoid(x)
    return sx * (1 -sx)
```

In [9]:
```
sigmoid(np.array([2.5, 0.32, -1.42]))          # [out]: array([0.92414182, 0.57932425, 0.19466158])
```

Out[9]:
```
array([0.92414182, 0.57932425, 0.19466158])
```

In [10]:
```
sigmoid_derivative(np.array([2.5, 0.32, -1.42]))  # [out]: array([0.07010372, 0.24370766, 0.15676845])
```

Out[10]:
```
array([0.07010372, 0.24370766, 0.15676845])
```

```
def cost(predicted, truth):
    return np.abs(truth - predicted)
```

```
gold = np.array([0.5, 1.2, 9.8])
pred = np.array([0.6, 1.0, 10.0])
cost(pred, gold)
```

```
array([0.1, 0.2, 0.2])
```

```
gold = np.array([0.5, 1.2, 9.8])
pred = np.array([9.3, 4.0, 99.0])
cost(pred, gold)
```

```
array([ 8.8,  2.8, 89.2])
```

Representing OR Boolean

Lets consider the problem of the OR boolean and apply the perceptron with simple gradient descent.

x2	x3	y
0	0	0
0	1	1
1	0	1
1	1	1

```
X = or_input = np.array([[0,0], [0,1], [1,0], [1,1]])
Y = or_output = np.array([[0,1,1,1]]).T
```

```
or_input
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

```
or_output
```

```
array([[0],
       [1],
       [1],
       [1]])
```

```
# Define the shape of the weight vector.
num_data, input_dim = ???
# Define the shape of the output vector.
output_dim = len(or_output.T)
```

Fill in the blanks =)

```
print('Inputs\n======')
print('no. of rows =', num_data)
print('no. of cols =', input_dim)
print('\n')
print('Outputs\n=======')
print('no. of cols =', output_dim)
```

```
Inputs
======
no. of rows = 4
no. of cols = 2


Outputs
=======
no. of cols = 1
```

```
# Initialize weights between the input layers and the perceptron
W = np.random.random((input_dim, output_dim))
W
```

```
array([[0.5488135 ],
       [0.71518937]])
```

```
# Define the shape of the weight vector.
num_data, input_dim = or_input.shape
# Define the shape of the output vector.
output_dim = len(or_output.T)
```

```
print('Inputs\n======')
print('no. of rows =', num_data)
print('no. of cols =', input_dim)
print('\n')
print('Outputs\n=======')
print('no. of cols =', output_dim)
```

```
Inputs
======
no. of rows = 4
no. of cols = 2


Outputs
=======
no. of cols = 1
```

```
# Initialize weights between the input layers and the perceptron
W = np.random.random((input_dim, output_dim))
W
```

```
array([[0.5488135 ],
       [0.71518937]])
```

Step 2a: Multiply the weights vector with the inputs, sum the products

To get the output of step 2a,

- Itrate through each row of the data, `X`
- For each column in each row, find the product of the value and the respective weights
- For each row, compute the sum of the products

```
# If we write it imperatively:
summation = []
for row in X:
    sum_wx = 0
    for feature, weight in zip(row, W):
        sum_wx += ???
    summation.append(???)
print(np.array(summation))
```

Fill in the blanks =)

```
[[0.        ]
 [0.38344152]
 [0.96366276]
 [1.34710428]]
```

Step 2a: Multiply the weights vector with the inputs, sum the products

To get the output of step 2a,

- Itrate through each row of the data, `X`
- For each column in each row, find the product of the value and the respective weights
- For each row, compute the sum of the products

```
# If we write it imperatively:
summation = []
for row in X:
    sum_wx = 0
    for feature, weight in zip(row, W):
        sum_wx += feature*  weight
    summation.append(sum_wx)
print(np.array(summation))
```

```
[[0.        ]
 [0.38344152]
 [0.96366276]
 [1.34710428]]
```

Step 2a: Multiply the weights vector with the inputs, sum the products

To get the output of step 2a,

- Itrate through each row of the data, `X`
- For each column in each row, find the product of the value and the respective weights
- For each row, compute the sum of the products

```
# If we write it imperatively:
summation = []
for row in X:
    sum_wx = 0
    for feature, weight in zip(row, W):
        sum_wx += feature*  weight
    summation.append(sum_wx)
print(np.array(summation))
```

```
[[0.        ]
 [0.38344152]
 [0.96366276]
 [1.34710428]]
```

```
# If we vectorize the process and use numpy.
np.dot(X, W)
```

```
array([[0.        ],
       [0.38344152],
       [0.96366276],
       [1.34710428]])
```

Train the Single-Layer Model

```
num_epochs = 10000 # No. of times to iterate.
learning_rate = 0.03 # How large a step to take per iteration.

# Lets standardize and call our inputs X and outputs Y
X = or_input
Y = or_output

for _ in range(num_epochs):
    layer0 = X

    # Step 2a: Multiply the weights vector with the inputs, sum the products, i.e. s
    # Step 2b: Put the sum through the sigmoid, i.e. f()
    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(X, W))

    # Back propagation.
    # Step 3a: Compute the errors, i.e. difference between expected output and predictions
    # How much did we miss?
    layer1_error = cost(layer1, Y)

    # Step 3b: Multiply the error with the derivatives to get the delta
    # multiply how much we missed by the slope of the sigmoid at the values in layer1
    layer1_delta = layer1_error * sigmoid_derivative(layer1)

    # Step 3c: Multiply the delta vector with the inputs, sum the product (use np.dot)
    # Step 4: Multiply the learning rate with the output of Step 3c.
    W +=  learning_rate * np.dot(layer0.T, layer1_delta)
```

```
layer1
```

```
array([[0.5       ],
       [0.95603077],
       [0.95677329],
       [0.99792643]])
```

```
# Expected output.
Y
```

```
array([[0],
       [1],
       [1],
       [1]])
```

```
# On the training data
[[int(prediction > 0.5)] for prediction in layer1]
```

```
[[0], [1], [1], [1]]
```

Lets try the XOR Boolean

Lets consider the problem of the OR boolean and apply the perceptron with simple gradient descent.

x2	x3	y
0	0	0
0	1	1
1	0	1
1	1	0

```
X = xor_input = np.array([[0,0], [0,1], [1,0], [1,1]])
Y = xor_output = np.array([[0,1,1,0]]).T
```

```
xor_input
```

```
array([[0, 0],
       [0, 1],
       [1, 0],
       [1, 1]])
```

```
xor_output
```

```
array([[0],
       [1],
       [1],
       [0]])
```

Fill in the blanks =)

```
num_epochs = 10000 # No. of times to iterate.
learning_rate = 0.003 # How large a step to take per iteration.

# Lets drop the last row of data and use that as unseen test.
X = xor_input
Y = xor_output

# Define the shape of the weight vector.
num_data, input_dim = ???
# Define the shape of the output vector.
output_dim = len(Y.T)
# Initialize weights between the input layers and the perceptron
W = np.random.random((input_dim, output_dim))

for _ in range(num_epochs):
    layer0 = X
    # Forward propagation.
    # Inside the perceptron, Step 2.
    layer1 = ???

    # How much did we miss?
    layer1_error = ???

    # Back propagation.
    # multiply how much we missed by the slope of the sigmoid at the values in layer1
    layer1_delta = ???

    # update weights
    W +=  ???
```

```
num_epochs = 10000 # No. of times to iterate.
learning_rate = 0.003 # How large a step to take per iteration.

# Lets drop the last row of data and use that as unseen test.
X = xor_input
Y = xor_output

# Define the shape of the weight vector.
num_data, input_dim = X.shape
# Define the shape of the output vector.
output_dim = len(Y.T)
# Initialize weights between the input layers and the perceptron
W = np.random.random((input_dim, output_dim))

for _ in range(num_epochs):
    layer0 = X
    # Forward propagation.
    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(X, W))

    # How much did we miss?
    layer1_error = cost(layer1, Y)

    # Back propagation.
    # multiply how much we missed by the slope of the sigmoid at the values in layer1
    layer1_delta = sigmoid_derivative(layer1) * layer1_error

    # update weights
    W +=  learning_rate * np.dot(layer0.T, layer1_delta)
```

```
# Expected output.
Y
```

```
array([[0],
       [1],
       [1],
       [0]])
```

```
# On the training data
[int(prediction > 0.5) for prediction in layer1] # All correct.
```

```
[0, 1, 1, 1]
```

Wrong!!!

You can't represent XOR with simple perceptron !!!

No matter how you change the hyperparameters or data, the XOR function can't be represented by a single perceptron layer.

There's no way you can get all four data points to get the correct outputs for the XOR boolean operation.

Solving XOR (Add more layers)

```
from itertools import chain
import numpy as np
np.random.seed(0)

def sigmoid(x): # Returns values that sums to one.
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(sx):
    # See https://math.stackexchange.com/a/1225116
    return sx * (1 - sx)

# Cost functions.
def cost(predicted, truth):
    return truth - predicted

xor_input = np.array([[0,0], [0,1], [1,0], [1,1]])
xor_output = np.array([[0,1,1,0]]).T

# Lets drop the last row of data and use that as unseen test.
X = xor_input_shuff
Y = xor_output_shuff

# Define the shape of the weight vector.
num_data, input_dim = X.shape
# Lets set the dimensions for the intermediate layer.
hidden_dim = 5
```

Fill in the blanks =)

```
hidden_dim = 5
# Initialize weights between the input layers and the hidden layer.
W1 = np.random.random((input_dim, hidden_dim))

# Define the shape of the output vector.
output_dim = len(Y.T)
# Initialize weights between the hidden layers and the output layer.
W2 = ???

# Initialize weigh
num_epochs = 10000
learning_rate = 0.03

for epoch_n in range(num_epochs):
    layer0 = X
    # Forward propagation.

    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(layer0, W1))
    layer2 = ???

    # Back propagation (Y -> layer2)

    # How much did we miss in the predictions?
    layer2_error = cost(layer2, Y)
    # In what direction is the target value?
    # Were we really close? If so, don't change too much.
    layer2_delta = layer2_error * sigmoid_derivative(layer2)


    # Back propagation (layer2 -> layer1)
    # How much did each layer1 value contribute to the layer2 error (according to the weights)?
    layer1_error = np.dot(layer2_delta, W2.T)
    layer1_delta = ???

    # update weights
    W2 +=  learning_rate * np.dot(layer1.T, layer2_delta)
    W1 +=  ???
```

```
hidden_dim = 5
# Initialize weights between the input layers and the hidden layer.
W1 = np.random.random((input_dim, hidden_dim))

# Define the shape of the output vector.
output_dim = len(Y.T)
# Initialize weights between the hidden layers and the output layer.
W2 = np.random.random((hidden_dim, output_dim))

# Initialize weigh
num_epochs = 10000
learning_rate = 0.03

for epoch_n in range(num_epochs):
    layer0 = X
    # Forward propagation.

    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(layer0, W1))
    layer2 = sigmoid(np.dot(layer1, W2))

    # Back propagation (Y -> layer2)

    # How much did we miss in the predictions?
    layer2_error = cost(layer2, Y)
    # In what direction is the target value?
    # Were we really close? If so, don't change too much.
    layer2_delta = layer2_error * sigmoid_derivative(layer2)


    # Back propagation (layer2 -> layer1)
    # How much did each layer1 value contribute to the layer2 error (according to the weights)?
    layer1_error = np.dot(layer2_delta, W2.T)
    layer1_delta = layer1_error * sigmoid_derivative(layer1)

    # update weights
    W2 +=  learning_rate * np.dot(layer1.T, layer2_delta)
    W1 +=  learning_rate * np.dot(layer0.T, layer1_delta)
```

```
# Training input.
X
```

```
array([[1, 1],
       [0, 1],
       [0, 0],
       [1, 0]])
```

```
# Expected output.
Y
```

```
array([[0],
       [1],
       [0],
       [1]])
```

```
layer2 # Our output layer
```

```
array([[0.46427804],
       [0.6213127 ],
       [0.31284349],
       [0.62323891]])
```

```
# On the training data
[int(prediction > 0.5) for prediction in layer2]
```

```
[0, 1, 0, 1]
```

Now try adding another layer

Use the same process:

1. Initialize
2. Forward Propagate
3. Back Propagate
4. Update (aka step)

```
from itertools import chain
import numpy as np
np.random.seed(0)

def sigmoid(x): # Returns values that sums to one.
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(sx):
    # See https://math.stackexchange.com/a/1225116
    return sx * (1 - sx)

# Cost functions.
def cost(predicted, truth):
    return truth - predicted

xor_input = np.array([[0,0], [0,1], [1,0], [1,1]])
xor_output = np.array([[0,1,1,0]]).T

# Lets drop the last row of data and use that as unseen test.
X = xor_input_shuff
Y = xor_output_shuff
```

Fill in the blanks =)

```
layer0to1_hidden_dim = 5
layer1to2_hidden_dim = 5

# Initialize weights between the input layers 0 ->  layer 1
W1 = np.random.random((input_dim, ???))

# Initialize weights between the layer 1 -> layer 2
W2 = np.random.random((???, ???))

# Define the shape of the output vector.
output_dim = len(Y.T)
# Initialize weights between the layer 2 -> layer 3
W3 = np.random.random((???, output_dim))

# Initialize weigh
num_epochs = 10000
learning_rate = 1.0

for epoch_n in range(num_epochs):
    layer0 = X
    # Forward propagation.

    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(layer0, W1))
    layer2 = ???
    layer3 = ???

    # Back propagation (Y -> layer2)
    # How much did we miss in the predictions?
    layer3_error = ???
    # In what direction is the target value?
    # Were we really close? If so, don't change too much.
    layer3_delta = ???

    # Back propagation (layer2 -> layer1)
    # How much did each layer1 value contribute to the layer3 error (according to the weights)?
    layer2_error = ???
    layer2_delta = ???

    # Back propagation (layer2 -> layer1)
    # How much did each layer1 value contribute to the layer2 error (according to the weights)?
    layer1_error = ???
    layer1_delta = layer1_error * sigmoid_derivative(layer1)

    # update weights
    W3 +=  learning_rate * np.dot(layer2.T, layer3_delta)
    W2 +=  ???
    W1 +=  ???
```

```
layer0to1_hidden_dim = 5
layer1to2_hidden_dim = 5

# Initialize weights between the input layers 0 ->  layer 1
W1 = np.random.random((input_dim, layer0to1_hidden_dim))

# Initialize weights between the layer 1 -> layer 2
W2 = np.random.random((layer0to1_hidden_dim, layer1to2_hidden_dim))

# Define the shape of the output vector.
output_dim = len(Y.T)
# Initialize weights between the layer 2 -> layer 3
W3 = np.random.random((layer1to2_hidden_dim, output_dim))

# Initialize weigh
num_epochs = 10000
learning_rate = 1.0

for epoch_n in range(num_epochs):
    layer0 = X
    # Forward propagation.

    # Inside the perceptron, Step 2.
    layer1 = sigmoid(np.dot(layer0, W1))
    layer2 = sigmoid(np.dot(layer1, W2))
    layer3 = sigmoid(np.dot(layer2, W3))

    # Back propagation (Y -> layer2)
    # How much did we miss in the predictions?
    layer3_error = cost(layer3, Y)
    # In what direction is the target value?
    # Were we really close? If so, don't change too much.
    layer3_delta = layer3_error * sigmoid_derivative(layer3)

    # Back propagation (layer2 -> layer1)
    # How much did each layer1 value contribute to the layer3 error (according to the weights)?
    layer2_error = np.dot(layer3_delta, W3.T)
    layer2_delta = layer3_error * sigmoid_derivative(layer2)

    # Back propagation (layer2 -> layer1)
    # How much did each layer1 value contribute to the layer2 error (according to the weights)?
    layer1_error = np.dot(layer2_delta, W2.T)
    layer1_delta = layer1_error * sigmoid_derivative(layer1)

    # update weights
    W3 +=  learning_rate * np.dot(layer2.T, layer3_delta)
    W2 +=  learning_rate * np.dot(layer1.T, layer2_delta)
    W1 +=  learning_rate * np.dot(layer0.T, layer1_delta)
```

Now, lets do it with PyTorch

First lets try a single perceptron and see that we can't train a model that can represent XOR.

```
from tqdm import tqdm

import torch
from torch import nn
from torch.autograd import Variable
from torch import FloatTensor
from torch import optim
use_cuda = torch.cuda.is_available()
```

```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

sns.set_style("darkgrid")
sns.set(rc={'figure.figsize':(15, 10)})
```

```
X # Original XOR X input in numpy array data structure.
```

```
array([[0, 1],
       [1, 1],
       [1, 0],
       [0, 0]])
```

```
Y # Original XOR Y output in numpy array data structure.
```

```
array([[1],
       [0],
       [1],
       [0]])
```

```
device = 'gpu' if torch.cuda.is_available()  else 'cpu'
# Converting the X to PyTorch-able data structure.
X_pt = torch.tensor(X).float()
X_pt = X_pt.to(device)
# Converting the Y to PyTorch-able data structure.
Y_pt = torch.tensor(Y, requires_grad=False).float()
Y_pt = Y_pt.to(device)
print(X_pt)
print(Y_pt)
```

```
tensor([[0., 1.],
        [1., 1.],
        [1., 0.],
        [0., 0.]])
tensor([[1.],
        [0.],
        [1.],
        [0.]])
```

```
# Use tensor.shape to get the shape of the matrix/tensor.
num_data, input_dim = X_pt.shape
print('Inputs Dim:', input_dim)

num_data, output_dim = Y_pt.shape
print('Output Dim:', output_dim)
```

```
Inputs Dim: 2
Output Dim: 1
```

```
# Use Sequential to define a simple feed-forward network.
model = nn.Sequential(
            nn.Linear(input_dim, output_dim), # Use nn.Linear to get our simple perceptron
            nn.Sigmoid()                      # Use nn.Sigmoid to get our sigmoid non-linearity
        )
model
```

```
Sequential(
  (0): Linear(in_features=2, out_features=1, bias=True)
  (1): Sigmoid()
)
```

```
# Remember we define as: cost = truth - predicted
# If we take the absolute of cost, i.e.: cost = |truth - predicted|
# we get the L1 loss function.
criterion = nn.L1Loss()
learning_rate = 0.03
```

```
# The simple weights/parameters update processes we did before
# is call the gradient descent. SGD is the sochastic variant of
# gradient descent.
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

Training a PyTorch model

To train a model using PyTorch, we simply iterate through the no. of epochs and imperatively state the computations we want to perform.

Remember the steps?

1. Initialize
2. Forward Propagation
3. Backward Propagation
4. Update Optimizer

```
num_epochs = 7000
```

```
# Step 1: Initialization.
# Note: When using PyTorch a lot of the manual weights
#       initialization is done automatically when we define
#       the model (aka architecture)
model = nn.Sequential(
            nn.Linear(input_dim, output_dim),
            nn.Sigmoid())
criterion = nn.L1Loss()
learning_rate = 1.0
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 10000

losses = []
```

```
# Step 1: Initialization.
# Note: When using PyTorch a lot of the manual weights
#       initialization is done automatically when we define
#       the model (aka architecture)
model = nn.Sequential(
            nn.Linear(input_dim, output_dim),
            nn.Sigmoid())
criterion = nn.MSELoss()
learning_rate = 1.0
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 10000

losses = []

for i in tqdm(range(num_epochs)):
    # Reset the gradient after every epoch.
    optimizer.zero_grad()
    # Step 2: Foward Propagation
    predictions = model(X_pt)

    # Step 3: Back Propagation
    # Calculate the cost between the predictions and the truth.
    loss_this_epoch = criterion(predictions, Y_pt)
    # Note: The neat thing about PyTorch is it does the
    #       auto-gradient computation, no more manually defining
    #       derivative of functions and manually propagating
    #       the errors layer by layer.
    loss_this_epoch.backward()

    # Step 4: Optimizer take a step.
    # Note: Previously, we have to manually update the
    #       weights of each layer individually according to the
    #       learning rate and the layer delta.
    #       PyTorch does that automatically =)
    optimizer.step()

    # Log the loss value as we proceed through the epochs.
    losses.append(loss_this_epoch.data.item())

# Visualize the losses
plt.plot(losses)
plt.show()
```

```
for _x, _y in zip(X_pt, Y_pt):
    prediction = model(_x)
    print('Input:\t', list(map(int, _x)))
    print('Pred:\t', int(prediction))
    print('Ouput:\t', int(_y))
    print('######')
```

```
Input:   [0, 1]
Pred:    0
Ouput:   1
######
Input:   [1, 1]
Pred:    1
Ouput:   0
######
Input:   [0, 0]
Pred:    0
Ouput:   0
######
Input:   [1, 0]
Pred:    0
Ouput:   1
######
```

Now, try again with 2 layers using PyTorch

Fill in the blanks =)

```
%%time

hidden_dim = 5
num_data, input_dim = X_pt.shape
num_data, output_dim = Y_pt.shape

model = nn.Sequential(nn.Linear(???, ???),
                      ???,
                      nn.Linear(???, ???),
                      nn.Sigmoid())

criterion = nn.MSELoss()
learning_rate = 0.3
optimizer = ???
num_epochs = 5000

losses = []

for _ in tqdm(range(num_epochs)):
    optimizer.zero_grad()
    predictions = ???
    loss_this_epoch = ???
    loss_this_epoch.???
    optimizer.???
    losses.append(loss_this_epoch.data.item())
    ##print([float(_pred) for _pred in predictions], list(map(int, Y_pt)), loss_this_epoch.data[0])

# Visualize the losses
plt.plot(losses)
plt.show()
```

Now, try again with 2 layers using PyTorch

```
%%time

hidden_dim = 5
num_data, input_dim = X_pt.shape
num_data, output_dim = Y_pt.shape

model = nn.Sequential(nn.Linear(input_dim, hidden_dim),
                      nn.Sigmoid(),
                      nn.Linear(hidden_dim, output_dim),
                      nn.Sigmoid())

criterion = nn.MSELoss()
learning_rate = 0.3
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 5000

losses = []

for _ in tqdm(range(num_epochs)):
    optimizer.zero_grad()
    predictions = model(X_pt)
    loss_this_epoch = criterion(predictions, Y_pt)
    loss_this_epoch.backward()
    optimizer.step()
    losses.append(loss_this_epoch.data.item())
    ##print([float(_pred) for _pred in predictions], list(map(int, Y_pt)), loss_this_epoch.data[0])

# Visualize the losses
plt.plot(losses)
plt.show()
```

```
for _x, _y in zip(X_pt, Y_pt):
    prediction = model(_x)
    print('Input:\t', list(map(int, _x)))
    print('Pred:\t', int(prediction > 0.5))
    print('Ouput:\t', int(_y))
    print('######')
```

```
Input:   [1, 1]
Pred:    0
Ouput:   0
######
Input:   [0, 1]
Pred:    1
Ouput:   1
######
Input:   [0, 0]
Pred:    0
Ouput:   0
######
Input:   [1, 0]
Pred:    1
Ouput:   1
######
```

MNIST

The "Hello World" of Neural Nets

MNIST: The "Hello World" of Neural Nets

Like any deep learning class, we ***must*** do the MNIST.

The MNIST dataset is

- is made up of handwritten digits
- 60,000 examples training set
- 10,000 examples test set

```
# We're going to install tensorflow here because their dataset access is simpler =)
!pip3 install tensorflow
```

```
Requirement already satisfied: tensorflow in /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-pac
kages (1.8.0)
Requirement already satisfied: termcolor>=1.1.0 in /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/si
te-packages (from tensorflow) (1.1.0)
```

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
# Visualization Candies
import matplotlib.pyplot as plt

def show_image(mnist_x_vector, mnist_y_vector):
    pixels = mnist_x_vector.reshape((28, 28))
    label = np.where(mnist_y_vector == 1)[0][0]
    plt.title('Label is {}'.format(label))
    plt.imshow(pixels, cmap='gray')
    plt.show()
```

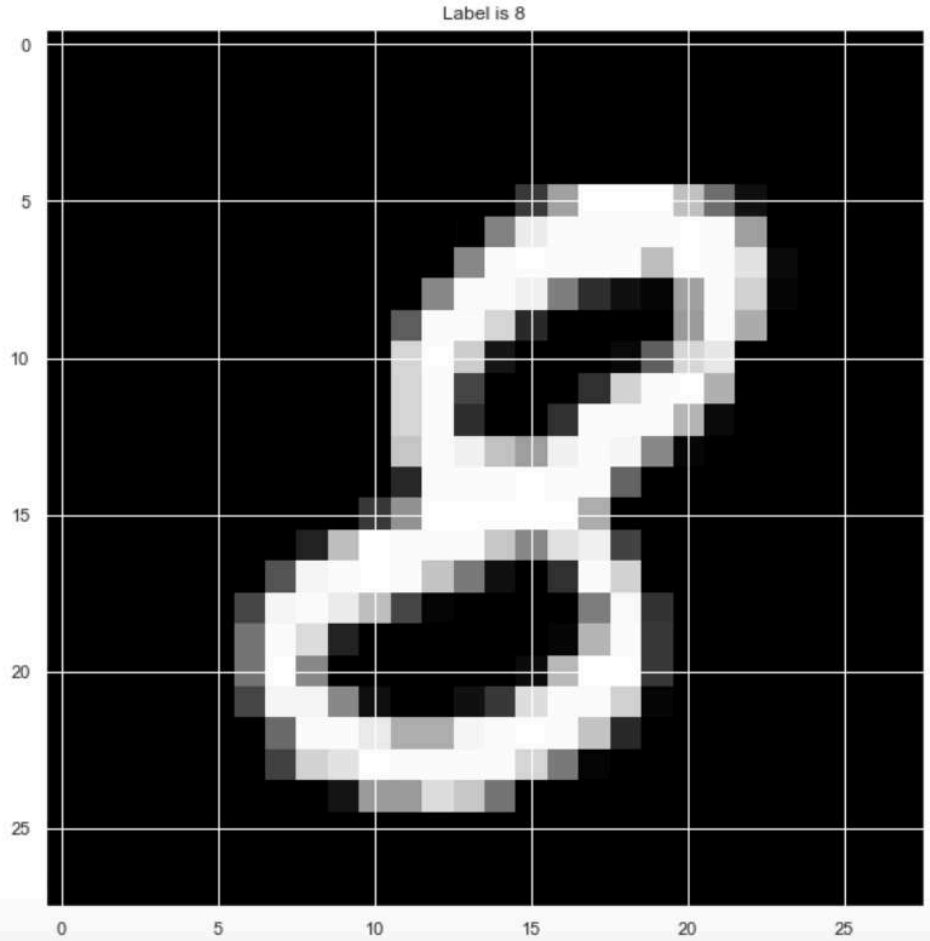
```
# Fifth image and label.
show_image(mnist.train.images[5], mnist.train.labels[5])
```

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

```
# Visualization Candies
import matplotlib.pyplot as plt

def show_image(mnist_x_vector, mnist_y_vector):
    pixels = mnist_x_vector.reshape((28, 28))
    label = np.where(mnist_y_vector == 1)[0][0]
    plt.title('Label is {}'.format(label))
    plt.imshow(pixels, cmap='gray')
    plt.show()
```

```
# Fifth image and label.
show_image(mnist.train.images[5], mnist.train.labels[5])
```



Fill in the blanks =)

```
X_mnist = torch.tensor(mnist.train.images).float()
Y_mnist = torch.tensor(mnist.train.labels).float()

X_mnist_test = torch.tensor(mnist.test.images).float()
Y_mnist_test = torch.tensor(mnist.test.labels).float()
```

```
# Use FloatTensor.shape to get the shape of the matrix/tensor.
num_data, input_dim = X_mnist.shape
print('Inputs Dim:', input_dim)

num_data, output_dim = ???
print('Output Dim:', output_dim)
```

```
Inputs Dim: 784
Output Dim: 10
```

```
hidden_dim = 500

model = nn.Sequential(nn.Linear(???, ???),
                      nn.???())

criterion = nn.???()
learning_rate = 1.0
optimizer = optim.???(model.parameters(), lr=learning_rate)
num_epochs = 100


losses = []
plt.ion()

for _e in tqdm(range(num_epochs)):
    optimizer.zero_grad()
    predictions = ???
    loss_this_epoch = ???
    loss_this_epoch.???()
    optimizer.???()
    ##print([float(_pred) for _pred in predictions], list(map(int, Y_pt)), loss_this_epoch.data[0])
    losses.append(loss_this_epoch.data.item())

    clear_output(wait=True)
    plt.plot(losses)
    plt.pause(0.05)
```

```
X_mnist = torch.tensor(mnist.train.images).float()
Y_mnist = torch.tensor(mnist.train.labels).float()

X_mnist_test = torch.tensor(mnist.test.images).float()
Y_mnist_test = torch.tensor(mnist.test.labels).float()
```

```
# Use FloatTensor.shape to get the shape of the matrix/tensor.
num_data, input_dim = X_mnist.shape
print('Inputs Dim:', input_dim)

num_data, output_dim = Y_mnist.shape
print('Output Dim:', output_dim)
```

```
Inputs Dim: 784
Output Dim: 10
```

```
hidden_dim = 500

model = nn.Sequential(nn.Linear(input_dim, output_dim),
                      nn.Sigmoid())

criterion = nn.MSELoss()
learning_rate = 1.0
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 100


losses = []
plt.ion()

for _e in tqdm(range(num_epochs)):
    optimizer.zero_grad()
    predictions = model(X_mnist)
    loss_this_epoch = criterion(predictions, Y_mnist)
    loss_this_epoch.backward()
    optimizer.step()
    ##print([float(_pred) for _pred in predictions], list(map(int, Y_pt)), loss_this_epoch.data[0])
    losses.append(loss_this_epoch.data.item())

    clear_output(wait=True)
    plt.plot(losses)
    plt.pause(0.05)
```

```
X_mnist = torch.tensor(mnist.train.images).float()
Y_mnist = torch.tensor(mnist.train.labels).float()

X_mnist_test = torch.tensor(mnist.test.images).float()
Y_mnist_test = torch.tensor(mnist.test.labels).float()
```

```
# Use FloatTensor.shape to get the shape of the matrix/tensor.
num_data, input_dim = X_mnist.shape
print('Inputs Dim:', input_dim)

num_data, output_dim = Y_mnist.shape
print('Output Dim:', output_dim)
```

```
Inputs Dim: 784
Output Dim: 10
```

```
hidden_dim = 500

model = nn.Sequential(nn.Linear(input_dim, output_dim),
                      nn.Sigmoid())

criterion = nn.MSELoss()
learning_rate = 1.0
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 100


losses = []
plt.ion()

for _e in tqdm(range(num_epochs)):
    optimizer.zero_grad()
    predictions = model(X_mnist)
    loss_this_epoch = criterion(predictions, Y_mnist)
    loss_this_epoch.backward()
    optimizer.step()
    ##print([float(_pred) for _pred in predictions], list(map(int, Y_pt)), loss_this_epoch.data[0])
    losses.append(loss_this_epoch.data.item())

    clear_output(wait=True)
    plt.plot(losses)
    plt.pause(0.05)
```

```
predictions = model(X_mnist_test)
predictions
```

```
tensor([[0.1220, 0.0558, 0.1120,  ..., 0.8124, 0.1212, 0.2739],
        [0.1745, 0.1066, 0.4185,  ..., 0.0175, 0.1121, 0.0340],
        [0.0963, 0.7832, 0.2290,  ..., 0.1862, 0.1943, 0.1618],
        ...,
        [0.0073, 0.0296, 0.0250,  ..., 0.1103, 0.0635, 0.1627],
        [0.0748, 0.1572, 0.0525,  ..., 0.0713, 0.1085, 0.0661],
        [0.1785, 0.0087, 0.1558,  ..., 0.0084, 0.0274, 0.0240]],
       grad_fn=<SigmoidBackward>)
```

```
pred = np.array([np.argmax(_p) for _p in predictions.data.numpy()])
pred
```

```
array([7, 2, 1, ..., 4, 1, 6])
```

```
truth = np.array([np.argmax(_p) for _p in Y_mnist_test.data.numpy()])
truth
```

```
array([7, 2, 1, ..., 4, 5, 6])
```

```
(pred == truth).sum() / len(pred)
```

```
0.7895
```

```
hidden_dim = 500

model = nn.Sequential(nn.Linear(input_dim, hidden_dim),
                      nn.Sigmoid(),
                     nn.Linear(hidden_dim, output_dim),
                      nn.Sigmoid())

criterion = nn.MSELoss()
learning_rate = 1.0
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
num_epochs = 1000


losses = []
plt.ion()

for _e in tqdm(range(num_epochs)):
    optimizer.zero_grad()
    predictions = model(X_mnist)
    loss_this_epoch = criterion(predictions, Y_mnist)
    loss_this_epoch.backward()
    optimizer.step()
    ##print([float(_pred) for _pred in predictions], list(map(int, Y_pt)), loss_this_epoch.data[0])
    losses.append(loss_this_epoch.data.item())

plt.plot(losses)
```

```
  2%|▌          | 21/1000 [00:35<27:24,  1.68s/it]
```