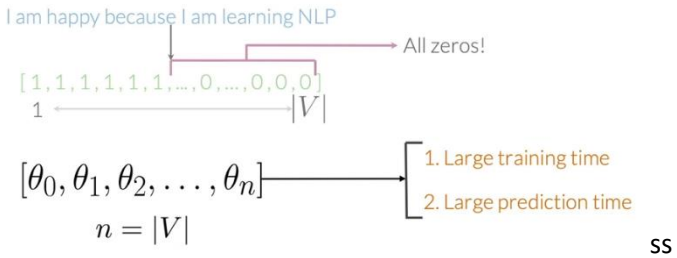
Sentiment Analysis

- Classify positive and negative sentiment
- Process:
    - Logistic Regression classifier to classify the sentiment of a sentence from it's extracted features.

Vocabulary & Feature Extraction



ss

- Vector representation:
    - Sparse representation
- Process:
    - Build a vocabulary
        - Go through all words in tweets
        - Save each unique word encountered
    - Feature extraction:
        - Check the present of vocabulary words in a tweet
            - Assign a value of 1 for presence, 0 for absence.
- Problem:
    - Total features in the vector equal the size of the vocabulary
    - n+1 total parameters to learn, n = |V|
    - Majority of features will likely be 0 for each tweet

Negative and Positive Frequencies

Word frequency in classes

Vocabulary	PosFreq (1)	NegFreq (0)
I	3	3
am	3	3
happy	2	0
because	1	0
learning	1	1
NLP	1	1
sad	0	2
not	0	1

- Simply number of times the words show up in the corresponding class

Feature Extraction with Frequencies

- freqs: dictionary mapping from (word, class) to frequency.

$$X_m = [1, \sum_w freqs(w, 1), \sum_w freqs(w, 0)]$$

Features of tweet m | Bias | Sum Pos. Frequencies | Sum Neg. Frequencies

  -
  - Word refers to unique, no duplicates
- Example:

$$\sum_w freqs(w, 0)$$

I am sad, I am not learning NLP

Vocabulary	PosFreq (1)	NegFreq (0)
I	3	3
am	3	3
happy	2	0
because	1	0
learning	1	1
NLP	1	1
sad	0	2
not	0	1

  -
    - The value of the sum of positive frequencies = 8
    - The value of the sum of negative frequencies = 11
    - X_M = [1,8,11]

Preprocessing

- Stop words and punctuation
  - In a sentencies, remove all stop words like “and”, “has”… and punctuations
  - Example:
    - @YMourri @AndrewYNg tuning GREAT AI model https://deeplearning.ai!!!!

      @YMourri @AndrewYNg tuning GREAT AI model https://deeplearning.ai

Stop words
and
is
a
at
has
for
of

Punctuation
,
.
:
!
“
‘

  - Remove all handles and URLs:
    - ~~@YMourri @AndrewYNg~~ tuning GREAT AI model ~~https://deeplearning.ai~~
- Stemming and lowercasing
  - Stemming:
    - Convert every word to its stem.
    - Tuning → Tun
  - Lowercaseing:
    - GREAT → great
  - Tuning GREAT AI model → tun great ai model

tuning GREAT AI model → tun | tune, tuned, tuning

GREAT, Great, great → great

Preprocessed tweet:
[tun, great, ai, model]

  -

Preprocessing lab:

```
import nltk                                 # Python library for NLP
from nltk.corpus import twitter_samples     # sample Twitter dataset from NLTK
from nltk.corpus import stopwords           # module for stop words that come with NLTK
from nltk.stem import PorterStemmer         # module for stemming
from nltk.tokenize import TweetTokenizer    # module for tokenizing strings
import re                                   # library for regular expression operations
import string                               # for string operations

nltk.download('twitter_samples')

all_positive_tweets = twitter_samples.strings('positive_tweets.json') # a list of string(tweet)
all_negative_tweets = twitter_samples.strings('negative_tweets.json')

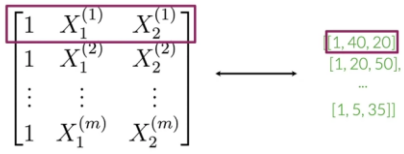
all_positive_tweets[random.randint(0,5000)]
--> "#KFCkitchentours Kitchen is so clean. I'm totally amazed :) @KFC_India"
all_negative_tweets[random.randint(0,5000)]
--> "Athabasca glacier was there in #1948 :-( #athabasca #glacier #jasper #jaspernationalpark #alberta #explorealberta
#… http://t.co/dZZdqmf7Cz"
tweet = all_positive_tweets[2277]

#Preprocess raw text for sentiment analysis
    nltk.download('stopwords')
    tweet2 = re.sub(r'^RT[\s]+', '', tweet)
    tweet2 = re.sub(r'https?://[^\s\n\r]+', '', tweet2)
    tweet2 = re.sub(r'#', '', tweet2)
    print(tweet2)
#- Tokenizing the string && lowercasing
    tokenizer = TweetTokenizer(preserve_case=False, strip_handles=True, reduce_len=True)
    tweet_tokens = tokenizer.tokenize(tweet2)
    #tweet2:
        #My beautiful sunflowers on a sunny Friday morning off :) sunflowers favourites happy Friday off…
    #tweet_tokens:
        #['my', 'beautiful', 'sunflowers', 'on', 'a', 'sunny', 'friday', 'morning', 'off', ':)', 'sunflowers',
'favourites', 'happy', 'friday', 'off', '…']

#- Removing stop words and punctuation
    stopwords_english = stopwords.words('english')
    tweets_clean = []
    for word in tweet_tokens: # Go through every word in your tokens list
        if (word not in stopwords_english and  # remove stopwords
            word not in string.punctuation):  # remove punctuation
            tweets_clean.append(word)
    #tweets_clean (removed stop words and punctuation)
        #['beautiful', 'sunflowers', 'sunny', 'friday', 'morning', ':)', 'sunflowers', 'favourites', 'happy', 'friday',
'…']
#- Stemming
    stemmer = PorterStemmer()
    tweets_stem = []
    for word in tweets_clean:
        stem_word = stemmer.stem(word)  # stemming word
        tweets_stem.append(stem_word)  # append to the list
    #tweets_stem:
        #['beauti', 'sunflow', 'sunni', 'friday', 'morn', ':)', 'sunflow', 'favourit', 'happi', 'friday', '…']

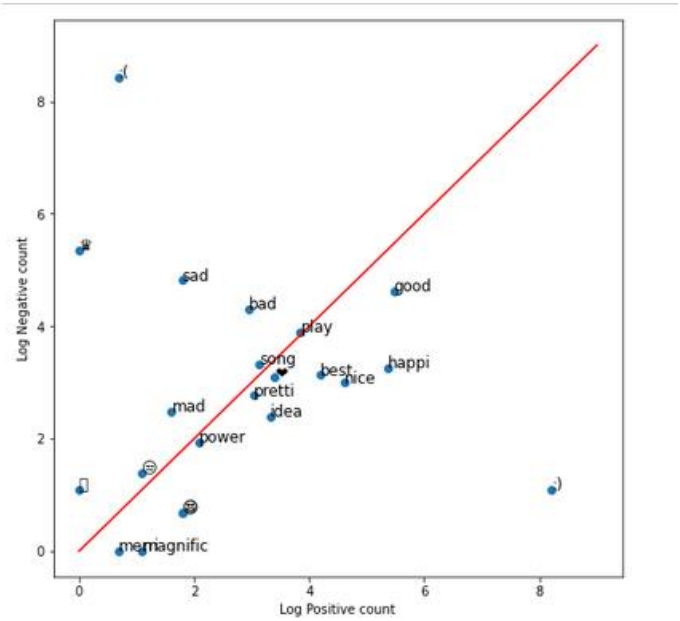
--> make a single function process_tweet that does all above
```

Putting it all together

- Example: I am Happy Because i am learning NLP @deeplearning
  - → Preprocessing: [happy, learn, nlp]
  - → feature extraction:
    - [bias, sum positibe frequencies, sum negative frequencies]
    - → [1,4,2]
- In practice,
  - do preprocessing for a set of m raw tweets one by one
  - extract feature for them → put them together in a X_matrix
  - 

Word frequencies Lab:

```
def build_freqs(tweets, ys):
    freqs = {}
  for y, tweet in zip(yslist, tweets):
        for word in process_tweet(tweet):
            pair = (word, y)
            freqs[pair] = freqs.get(pair, 0) + 1 #example: ('followfriday', 1.0): 25
    return freqs
keys = ['happi', 'merri', 'nice', 'good', 'bad', 'sad', 'mad', '...']
data = []
for word in keys:
    pos = 0
    neg = 0
    if (word, 1) in freqs:
        pos = freqs[(word, 1)]
    if (word, 0) in freqs:
        neg = freqs[(word, 0)]
    data.append([word, pos, neg])
#data: [['happi', 212, 25],[...],['nice', 99, 19]]
```



Logistic Regression

- Training:
  - Decision boundary:
    - $z = \theta * x = 0$
  - $x = [1, pos, neg]$
  - $z(\theta, x) = \theta_0 + \theta_1 * pos + \theta_2 * neg = 0$
  - $neg = (-\theta_0 - \theta_1 * pos)/\theta_2$
- Testing
  - $pred = h(X_{val}, \theta) \geq 0.5$
    - h: sigmoid
- Accuracy:
  - $$\sum_{i=1}^{m} \frac{(pred^{(i)} == y_{val}^{(i)})}{m}$$

Review:

$$h(x)' = \frac{1}{1+e^{-x}}' = \frac{-(1+e^{-x})'}{(1+e^{-x})^2} = \frac{-1' - (e^{-x})'}{(1+e^{-x})^2} = \frac{0 - (-x)'(e^{-x})}{(1+e^{-x})^2} = \frac{-(-1)(e^{-x})}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2}$$
$$= \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}} = h(x)\frac{+1-1+e^{-x}}{1+e^{-x}} = h(x)\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}} = h(x)(1-h(x))$$

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial}{\partial \theta_j}\frac{-1}{m}\sum_{i=1}^{m} y^{(i)} log(h(x^{(i)}, \theta)) + (1-y^{(i)}) log(1-h(x^{(i)}, \theta))$$
$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\frac{\partial}{\partial \theta_j} log(h(x^{(i)}, \theta)) + (1-y^{(i)})\frac{\partial}{\partial \theta_j} log(1-h(x^{(i)}, \theta))$$
$$= -\frac{1}{m}\sum_{i=1}^{m} \frac{y^{(i)}\frac{\partial}{\partial \theta_j} h(x^{(i)}, \theta)}{h(x^{(i)}, \theta)} + \frac{(1-y^{(i)})\frac{\partial}{\partial \theta_j}(1-h(x^{(i)}, \theta))}{1-h(x^{(i)}, \theta)}$$
$$= -\frac{1}{m}\sum_{i=1}^{m} \frac{y^{(i)}\frac{\partial}{\partial \theta_j} h(x^{(i)}, \theta)}{h(x^{(i)}, \theta)} + \frac{(1-y^{(i)})\frac{\partial}{\partial \theta_j}(1-h(x^{(i)}, \theta))}{1-h(x^{(i)}, \theta)}$$
$$= -\frac{1}{m}\sum_{i=1}^{m} \frac{y^{(i)}h(x^{(i)}, \theta)(1-h(x^{(i)}, \theta))\frac{\partial}{\partial \theta_j}\theta^T x^{(i)}}{h(x^{(i)}, \theta)} + \frac{-(1-y^{(i)})h(x^{(i)}, \theta)(1-h(x^{(i)}, \theta))\frac{\partial}{\partial \theta_j}\theta^T x^{(i)}}{1-h(x^{(i)}, \theta)}$$
$$= -\frac{1}{m}\sum_{i=1}^{m} \frac{y^{(i)}h(x^{(i)}, \theta)(1-h(x^{(i)}, \theta))\frac{\partial}{\partial \theta_j}\theta^T x^{(i)}}{h(x^{(i)}, \theta)} - \frac{(1-y^{(i)})h(x^{(i)}, \theta)(1-h(x^{(i)}, \theta))\frac{\partial}{\partial \theta_j}\theta^T x^{(i)}}{1-h(x^{(i)}, \theta))}$$
$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}(1-h(x^{(i)}, \theta))x_j^{(i)} - (1-y^{(i)})h(x^{(i)}, \theta)x_j^{(i)}$$
$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}(1-h(x^{(i)}, \theta)) - (1-y^{(i)})h(x^{(i)}, \theta)x_j^{(i)}$$
$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} - y^{(i)}h(x^{(i)}, \theta) - h(x^{(i)}, \theta) + y^{(i)}h(x^{(i)}, \theta)x_j^{(i)}$$
$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} - h(x^{(i)}, \theta)x_j^{(i)}$$
$$= \frac{1}{m}\sum_{i=1}^{m} h(x^{(i)}, \theta) - y^{(i)}x_j^{(i)}$$

The vectorized version:

$$\nabla J(\theta) = \frac{1}{m} \cdot X^T \cdot (H(X,\theta) - Y)$$

Frequencies to probability:

$P(w_i \mid \text{class})$

word	Pos	Neg
I	3	3
am	3	3
happy	2	1
because	1	0
learning	1	1
NLP	1	1
sad	1	2
not	1	2
Nclass	13	12

word	Pos	Neg
I	0.24	0.25
am	0.24	0.25
happy	0.15	0.08
because	0.08	0.01
learning	0.08	0.08
NLP	0.08	0.08
sad	0.08	0.17
not	0.08	0.17

-
- The columns summed to one

Naïve Bayes:

Tweet: I am happy today; I am learning.

$$\prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)}$$

$$\frac{0.20}{0.20} * \frac{0.20}{0.20} * \frac{0.14}{0.10} * \frac{0.20}{0.20} * \frac{0.20}{0.20} * \frac{0.10}{0.10}$$

word	Pos	Neg
I	0.20	0.20
am	0.20	0.20
happy	0.14	0.10
because	0.10	0.05
learning	0.10	0.10
NLP	0.10	0.10
sad	0.10	0.15
not	0.10	0.15

-
- $\prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)} = \frac{0.14}{0.10} = 1.4 > 1$ → Positive
- Problem:
  - Number of negative and positive class may be different
    → add a prior ratio:

$$\text{ratio}(w_i) = \frac{P(w_i \mid \text{Pos})}{P(w_i \mid \text{Neg})}$$

$$\approx \frac{freq(w_i, 1) + 1}{freq(w_i, 0) + 1}$$

$$\frac{P(pos)}{P(neg)} \prod_{i=1}^{m} \frac{P(w_i|pos)}{P(w_i|neg)}$$

  - Products bring risk of underflow
    → use log-likelihood

$$log \frac{P(tweet|pos)}{P(tweet|neg)} = log(P(tweet|pos)) - log(P(tweet|neg))$$

$$positive = log(P(tweet|pos)) = \sum_{i=0}^{n} log P(W_i|pos)$$

$$negative = log(P(tweet|neg)) = \sum_{i=0}^{n} log P(W_i|neg)$$

Summing the Lambdas

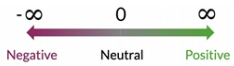
doc: I am happy because I am learning.

$$\lambda(w) = log \frac{P(w|pos)}{P(w|neg)}$$

$$\lambda(\text{happy}) = log \frac{0.09}{0.01} \approx 2.2$$

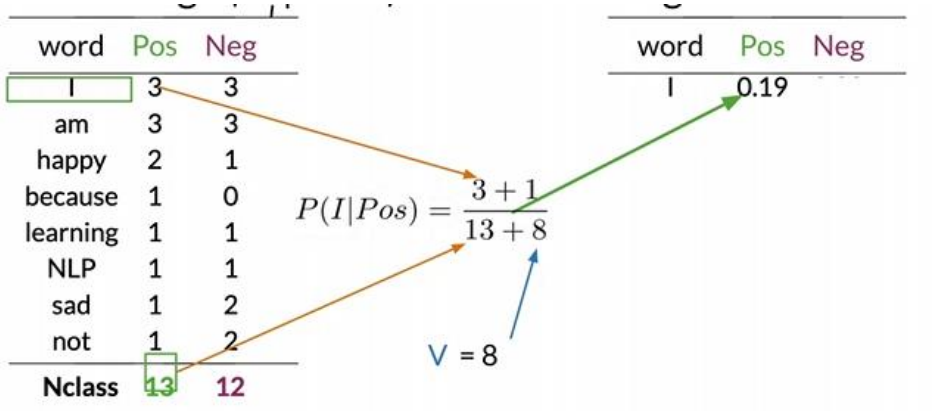
word	Pos	Neg	λ
I	0.05	0.05	0
am	0.04	0.04	0
happy	0.09	0.01	2.2
because	0.01	0.01	0
learning	0.03	0.01	1.1
NLP	0.02	0.02	0
sad	0.01	0.09	-2.2
not	0.02	0.03	-0.4

  - Inference time:



Laplacian Smoothing:

- $$P(w_i \mid class) = \frac{freq(w_i, class)}{N_{class}} \quad class \in \{Positive, Negative\}$$
- $$P(w_i \mid class) = \frac{freq(w_i, class)+1}{(N_{class}+V)}$$
- →



Training Naïve Bayes:

- Get or annotate a dataset with positive and negative tweets
- Preprocess the tweets: process_tweet(tweet) → [w_1, w_2, …]
- Compute freq(w, class)
- Get P(w | pos), P(w | neg)
- Get λ(w)
  - $$\lambda(w) = log\frac{P(w|pos)}{P(w|neg)}$$
- Compute logprior = log(P(pos)/P(neg))

Testing Naïve Bayes on unseen data:

- X_{val} Y_{val} λ *log_prior*
- Score = predict(X_{val}, λ, *logprior*)
- pred = score > 0
- Accuracy:
  - $$\frac{1}{m}\sum_{i=1}^{m}(pred_i == Y_{val_i})$$

Applications of Naïve Bayes

- Author identification
  - $\frac{P(\text{book})}{P(\text{book})}$
- Spam filtering
  - $\frac{P(\text{spam}|\text{email})}{P(\text{nonspam}|\text{email})}$
- Information retrieval
  - $P(\text{document}_k|\text{query}) \propto \prod_{i=0}^{|query|} P(\text{query}_i|\text{document}_k)$
  - Retrieve document if $P(\text{document}_k|\text{query}) > \text{threshold}$
- Word disambiguation
  - $\frac{P(\text{river}|\text{text})}{P(\text{money}|\text{text})}$
  - Bank:  
- This method is usually used as a simple baseline. It is also really fast

Naïve Bayes Assumptions

- Words in a sentence (predictors or features) are independent of each other.
- Example Sentence: "It is sunny and hot in the Sahara desert."
  - Naïve Bayes would treat "sunny" and "hot" as independent when they are often correlated.
  - Could lead to incorrect estimations of conditional probabilities.
  - Naïve Bayes make the independence assumption and is affected by the word frequencies in the corpus. For example:
    -  "It is sunny and hot in the Sahara desert."  "It's always cold and snowy in ___."
  - The naïve model will assign equal weight to the words “spring, summer, fall, winter”.

Reliance on Training Data Distribution

- Naïve Bayes is heavily dependent on the distribution of the training dataset.
- Ideal training datasets should reflect the actual proportions of classes (e.g., positive and negative tweets) found in a random sample.
- Many annotated datasets are artificially balanced, which does not match the real distribution (e.g., the prevalence of positive over negative tweets).

Issues due to Dataset Imbalance

- Artificially balanced datasets → overly optimistic or pessimistic models.
- Reasons for imbalance include platform content bans or user content muting, affecting the negative tweet count.

Analyzing NLP Errors

- Key to error analysis is examining the processed text version.

Common Sources of Errors

- Semantic Loss During Preprocessing:
    - Example: "my beloved grandmother :(" becomes "beloved grandmother" after removing punctuation, altering the sentiment.
- Word Order Significance:
    - The order of words can affect the meaning of a sentence, which may not be captured by some NLP models.
    - Example: "I am not happy because I did not go" can be misinterpreted if the word "not" is overlooked.
- Language Quirks:
    - Nuances like sarcasm, irony, and euphemism, which are naturally understood by humans, can confuse models like Naïve Bayes.
    - Example: "This is a ridiculously powerful movie." might be misinterpreted due to the word "ridiculously."

Preprocessing Considerations

- Importance of preserving sentiment-indicating punctuation and negations.
- Check the results of preprocessing to ensure the model receives accurate data.

Model Limitations & Solutions

- Naïve Bayes classifiers can miss nuances due to their reliance on word frequency and the independence assumption.
- The importance of word order and handling of negations ("not") will be addressed later.

Adversarial Attacks

- Adversarial attacks describe situations where the model is tricked by language phenomena.
- Example: Positive statements with negative words due to sarcasm or emphasis can lead to misclassification.

Naïve Bayes Lab

- Identify the numbe of classes:
  - $$P(D_{pos}) = \frac{D_{pos}}{D}$$
  - $$P(D_{neg}) = \frac{D_{neg}}{D}$$
- Prior and Logprior
  - Prior: the ratio of the probability:
    - $$\frac{P(D_{pos})}{P(D_{neg})}$$
  - Logprior:
    - $$\text{logprior} = log\left(\frac{P(D_{pos})}{P(D_{neg})}\right) = log\left(\frac{D_{pos}}{D_{neg}}\right)$$
    - $$\text{logprior} = \log(P(D_{pos})) - \log(P(D_{neg})) = \log(D_{pos}) - \log(D_{neg})$$
- Positive and Negative Probability of a word:
  - $$P(W_{pos}) = \frac{freq_{pos} + 1}{N_{pos} + V}$$
  - $$P(W_{neg}) = \frac{freq_{neg} + 1}{N_{neg} + V}$$

```
for pair in freqs.keys():
    if pair[1] > 0:
        N_pos += freqs[pair]
    else:
        N_neg += freqs[pair]
```

```
freq_pos = freqs.get((word,1),0)
freq_neg = freqs.get((word,0),0)
p_w_pos = (freq_pos+1)/(N_pos+V)
p_w_neg = (freq_neg+1)/(N_neg+V)
```

- Log-likelihood:
  - $$\log\left(\frac{P(W_{pos})}{P(W_{neg})}\right)$$

```
loglikelihood[word] = np.log(p_w_pos/p_w_neg)
```

- Make prediction:
  - $$p = logprior + \sum_{i}^{N}(loglikelihood_i)$$

```
for word in word_l:
    if word in loglikelihood:
        p += loglikelihood[word]
```

Vector space models

- Usage:
    - identify similar meanings between sentences with different words
        - Use cases: question answering, paraphrasing, and summarization
    - capture dependencies between words (e.g., "buyand "sell")
- Application:
    - Information extraction (“answering who, what, where, how, etc.”)
    - Machine Translation
    - Chatbot

Word-By-Word design

- Number of times two words occur together within a certain distance k
- Example:
    - k = 2
        - 2 words preceding/following it in a sentence will be used to determine co-occurences.
    - “I like simple data”, “I prefer simple raw data”.

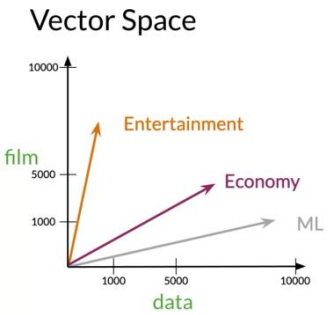
	simple	Raw	like	I
data	2	1	1	0

Word-By-Document Design

- Number of times a word occurs within a certain category

Word/Type	Entertainment	Economy	Machine Learning
data	500	6620	9320
film	7000	4000	1000

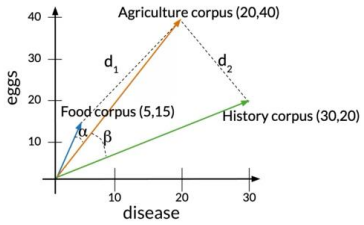
Vector Space



-
- Measures of “similarity” by Angle, Distance
    - Calculate the euclidean distance between the corpus
- Enconomy and ML are more alike than with entertainment

Problem with using Euclidean distance

- Euclidean distance is affected by the size of the document corpora.

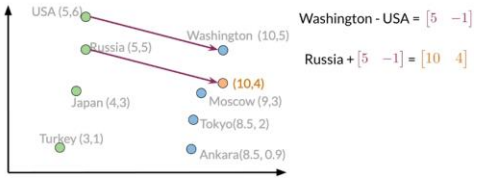


Cosine Similarity:

- Intuition:
  - Use the angle between the documents.
- Equation:
  - $\hat{v} \cdot \hat{w} = \|\hat{v}\|\|\hat{w}\| \cos(\beta)$
  - $\cos(\beta) = \frac{\hat{v} \cdot \hat{w}}{\|\hat{v}\|\|\hat{w}\|}$
  - $= \frac{(20 \times 30) + (40 \times 20)}{\sqrt{20^2 + 40^2} \times \sqrt{30^2 + 20^2}}$
  - β = 90° → cosine similarity = 0
  - β = 0° → cosine similarity = 1

Manipulating words in vector space

- to infer unknown relations among words



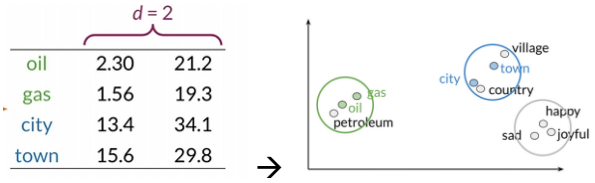
- Predict Russia's capital with the closest one

```
1. capital = vec('France') - vec('Paris')
2. country = vec('Madrid') + capital
3. def find_closest_word(v, k = 1):
4.     diff = embedding.values - v
5.     delta = np.sum(diff * diff, axis=1)
6.     i = np.argmin(delta)
7.     return embedding.iloc[i].name
8. find_closest_word(country)
```

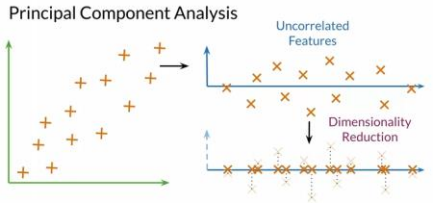
- Problem:
  - Often at very high dimension, unable to visualize clearly
  - d > 2

oil	0.20	...	0.10
gas	2.10	...	3.40
city	9.30	...	52.1
town	6.20	...	34.3

  - → reduce the dimension by PCA



PCA



- How to get uncorrelated features
  - Eigenvector: Uncorrelated features for your data
  - Eigenvalue: the amount of information retained by each feature
- Algorithm:
  - Mean normalize data
  - Get Covariance Matrix
  - Perform SVD on the Covariance matrix → eigenvectors(U) and eigenvalues(S)
    - Eigenvectors give the direction of uncorrelated features
    - Eigenvalues are the variance of the new features
  - Dot product to project data: X' = XU[:,0:2]
    - Gives the projection on uncorrelated features
  - Percentage of retained variance
    - $\frac{\sum_{i=0}^{1} S_{ii}}{\sum_{j=0}^{d} S_{jj}}$

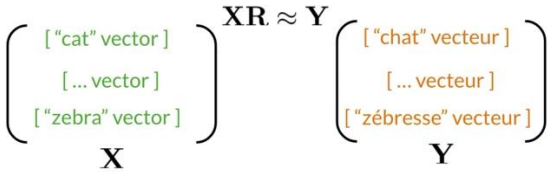
PCA algorithm:

```
1. def compute_pca(X, n_components=2):
2.     X_demeaned = X - np.mean(X, axis=0)
3.     covariance_matrix = np.cov(X_demeaned, rowvar = False)
4.     eigen_vals, eigen_vecs = np.linalg.eigh(covariance_matrix)
5.     idx_sorted = np.argsort(eigen_vals)
6.     idx_sorted_decreasing = idx_sorted[::-1]
7.     eigen_vals_sorted = eigen_vals[idx_sorted_decreasing]
8.     eigen_vecs_sorted = eigen_vecs[:,idx_sorted_decreasing]
9.     eigen_vecs_subset = eigen_vecs_sorted[:,:n_components]
10.    X_reduced = np.dot(eigen_vecs_subset.T,X_demeaned.T).T
11.    return X_reduced
```

Transforming word vectors

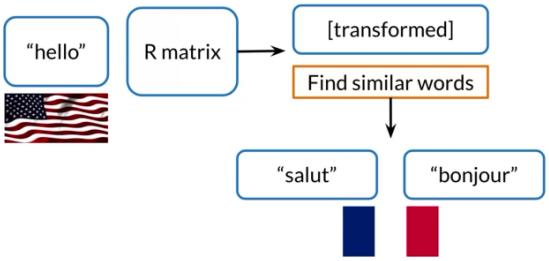
- Machine computes word embeddings for each language and learns to align them.
- Steps:
  - Retrieve the English word embedding (e.g., "cat") [1,0,1].
  - Transform this embedding into the French word vector space.
  - Search for the closest French word vector to find translation candidates.
  - Ideally, the machine will identify "chat" as the French equivalent for "cat".

Align word vectors

- 

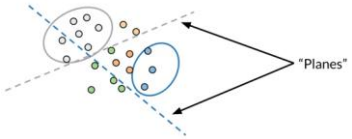
- Finding the transformation matrix R:
  - Initialize R randomly
  - Iteratively:
    - Use Frobenius norm
      - $$\|\mathbf{A}\|_F \equiv \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$
      - ```
        A = np.array([[2,2],[2,2]])
        A_squared = np.square(A)
        A_Frobenious = np.sqrt(np.sum(A_squared))
        A_Frobenious #4.0
        ```
      - Loss = $||XR-Y||_F$
      - Easier to work with frobenius norm squared:
        - $$\|\mathbf{XR}-\mathbf{Y}\|_F^2$$
          $$\mathbf{A}=\begin{pmatrix}2 & 2\\ 2 & 2\end{pmatrix}$$
          $$\|\mathbf{A}\|_F^2=\left(\sqrt{2^2+2^2+2^2+2^2}\right)^2$$
        - $$\|\mathbf{A}\|_F^2=16$$
    - Update R by subtracting the gradient
      - $$g=\frac{d}{dR}Loss$$
        $$R=R-\alpha g$$
      - With frobenious norm squared
        $$Loss=\|\mathbf{XR}-\mathbf{Y}\|_F^2$$
        $$g=\frac{d}{dR}Loss=\frac{2}{m}\left(\mathbf{X}^T(\mathbf{XR}-\mathbf{Y})\right)$$
        ```

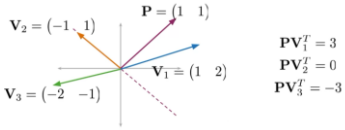
How to find similar word vectors?

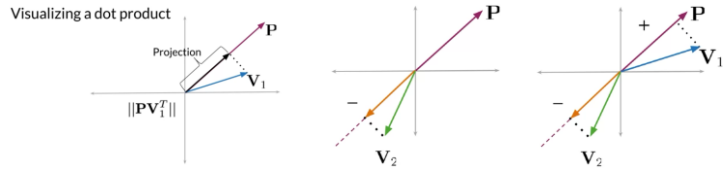


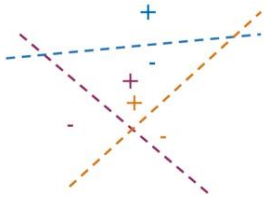
- K-nearest neighbors
  - Key operation to find a matching word.
  - Use locality sensitive hashing

Locality sensitive hashing

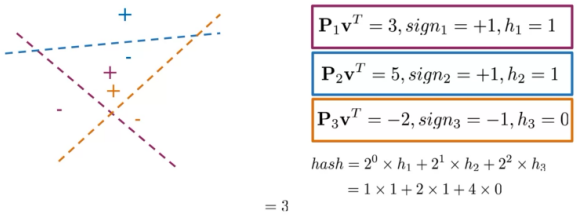
- a hashing method that cares very deeply about assigning items based on where they're located in vector space
- Planes:
  - 

  - 

  - 

- Multiple planes:
  - 

    - → combine all of those signals (-/+) into a single hash value.
      - The hash value defines a particular region within the vector space.
      - $\text{sign}_i \geq 0$, → $h_i = 1$
      - $\text{sign}_i < 0$, → $h_i = 0$
      - $$\text{hash} = \sum_{i}^{H} 2^i \times h_i$$
      - Example:

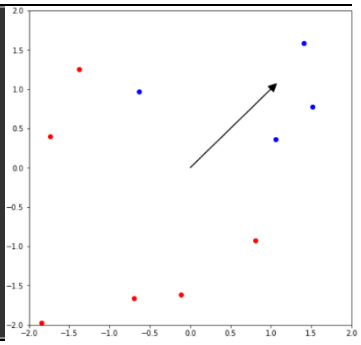


-

Hash function and multiplanes lab

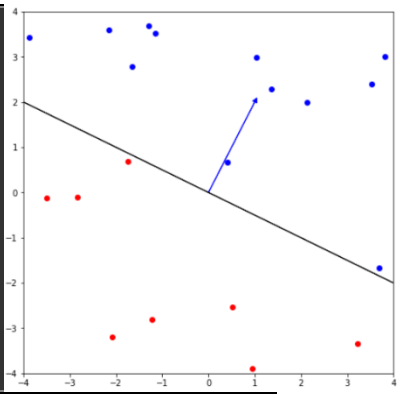
```
P = np.array([[1, 1]]) # Define a single plane.
fig, ax1 = plt.subplots(figsize=(8, 8))

plot_vectors([P], axes=[2, 2], ax=ax1)
for i in range(0, 10):
        v1 = np.array(np.random.uniform(-2, 2, 2))
        side_of_plane = np.sign(np.dot(P, v1.T))
        if side_of_plane == 1:
            ax1.plot([v1[0]], [v1[1]], 'bo')
        else:
            ax1.plot([v1[0]], [v1[1]], 'ro')
plt.show()
```



```
P = np.array([[1, 2]])
PT = np.dot([[0, 1], [-1, 0]], P.T).T

fig, ax1 = plt.subplots(figsize=(8, 8))
plot_vectors([P], colors=['b'], axes=[2, 2], ax=ax1)
plot_vectors([PT*4,PT*-4],colors=['k','k'],axes=[4,4],ax=ax1)
for i in range(0, 20):
        v1 = np.array(np.random.uniform(-4, 4, 2))
        side_of_plane = np.sign(np.dot(P, v1.T))
        if side_of_plane == 1:
            ax1.plot([v1[0]], [v1[1]], 'bo')
        else:
            ax1.plot([v1[0]], [v1[1]], 'ro')
plt.show()
```



```
def side_of_plane(P, v):
    dotproduct = np.dot(P, v.T)
    sign_of_dot_product = np.sign(dotproduct)
    sign_of_dot_product_scalar = sign_of_dot_product.item()
    return sign_of_dot_product_scalar
```

```
def hash_multi_plane(P_l, v):
    hash_value = 0
    for i, P in enumerate(P_l):
        sign = side_of_plane(P,v)
        hash_i = 1 if sign >=0 else 0
        hash_value += 2**i * hash_i
    return hash_value
```

```
def side_of_plane_matrix(P, v):
    dotproduct = np.dot(P, v.T)
    sign_of_dot_product = np.sign(dotproduct)
    return sign_of_dot_product
```

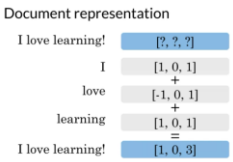
```
def hash_multi_plane_matrix(P, v, num_planes):
    sides_matrix = side_of_plane_matrix(P, v) # Get the side of planes for P and v
    hash_value = 0
    for i in range(num_planes):
        sign = sides_matrix[i].item() # Get the value inside the matrix cell
        hash_i = 1 if sign >=0 else 0
        hash_value += 2**i * hash_i # sum 2^i * hash_i
    return hash_value
```

Approximate nearest neighbors

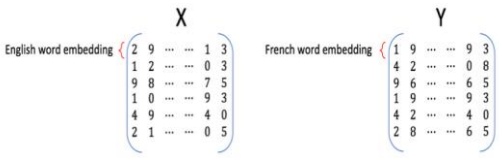
- Vector Space Partitioning
    - Planes as Dividers: A few planes can divide the vector space into distinct regions.
    - Optimal Division Uncertainty: It is unknown which set of planes best divides the vector space.
- Multiple Sets of Random Planes
    - Creating Multiple Sets: Instead of one set of planes, multiple sets of random planes are generated.
- Implementation for ANN
    - Vector Space Example: A vector space is considered with a magenta dot representing a transformed English word into a French word vector.

    - Hash Pockets Assignment: Different sets of random planes are used to assign similar vectors (e.g., French word vectors) to the same hash pockets.
        - One set pairs magenta and green vectors.
        - Another set pairs red and blue vectors.
        - A third set pairs magenta and orange vectors.
    - Robust Searching: Multiple sets of random planes make the search for nearest neighbors more robust by considering various hash pockets.
- Approximation vs. Precision
    - ANN Definition: Searching only a subset of the entire vector space for potential nearest neighbors, leading to an approximate, rather than exact, k-NN.
    - Trade-off: There is a trade-off of some precision for significantly increased efficiency in the search process.

Document representation



Naïve Machine Translation and LSH




```
def get_matrices(en_fr, french_vecs, english_vecs):
    X_l = list()
    Y_l = list()
    english_set = set(english_vecs)
    french_set = set(french_vecs)
    french_words = set(en_fr.values())
    for en_word, fr_word in en_fr.items():
        if fr_word in french_set and en_word in english_set:
            en_vec = english_vecs[en_word]
            fr_vec = french_vecs[fr_word]
            X_l.append(en_vec)
            Y_l.append(fr_vec)
    X = np.vstack(X_l)
    Y = np.vstack(Y_l)
    return X, Y
```

Translation

Problem:

- $$\arg\min_{\mathbf{R}} \|\mathbf{XR} - \mathbf{Y}\|_F$$

Frobenius norm:

- $$\|\mathbf{A}\|_F \equiv \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

Loss function:

- $$\frac{1}{m}\|\mathbf{XR} - \mathbf{Y}\|_F^2$$

Compute loss:

- $$L(X, Y, R) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}\left(a_{ij}\right)^2$$

```
def compute_loss(X, Y, R):
    m = X.shape[0]
    diff = np.dot(X, R) - Y
    diff_squared = np.square(diff)
    sum_diff_squared = np.sum(diff_squared)
    loss = sum_diff_squared / m
    return loss
```

Compute gradient:

```
def compute_gradient(X, Y, R):
    m = X.shape[0]
    gradient = np.dot(X.T,(np.dot(X,R) - Y)) * 2/m
    return gradient
```

Update R:

- $R_{new} = R_{old} - \alpha g$

```
def align_embeddings(X, Y, train_steps=100, learning_rate=0.0003, verbose=True, compute_loss=compute_loss, compute_gradient=compute_gradient):
    np.random.seed(129)
    R = np.random.rand(X.shape[1], X.shape[1])
    for i in range(train_steps):
        if verbose and i % 25 == 0:
            print(f"loss at iteration {i} is: {compute_loss(X, Y, R):.4f}")
        gradient = compute_gradient(X,Y,R)
        R -= learning_rate*gradient
    return R
```

Testing the translation by K-Nearest Neighbor Algorithm

- Use 1-NN with eR as input → search for an embedding f (as a row) in the matrix Y which is the closest to the transformed vector eR.

Cosine Similarity:

- Cosine of the angle between vector u and v.
  - $\cos(u, v) = \frac{u \cdot v}{\|u\| \, \|v\|}$

Nearest_neighbor

```
def nearest_neighbor(v, candidates, k=1, cosine_similarity=cosine_similarity):
    similarity_l = []
    for row in candidates:
        cos_similarity = cosine_similarity(v, row)
        similarity_l.append(cos_similarity)
    sorted_ids = np.argsort(similarity_l)
    sorted_ids = sorted_ids[::-1]
    k_idx = sorted_ids[:k]
    return k_idx
```

Test vocabulary

```
def test_vocabulary(X, Y, R, nearest_neighbor=nearest_neighbor):
    pred = np.dot(X,R)
    num_correct = 0
    for i in range(len(pred)):
        pred_idx = nearest_neighbor(pred[i],Y)
        if pred_idx == i:
            num_correct += 1
    accuracy = num_correct/pred.shape[0]
    return accuracy
```

Get document embedding

```
def get_document_embedding(tweet, en_embeddings, process_tweet=process_tweet):
    doc_embedding = np.zeros(300)
    processed_doc = process_tweet(tweet)
    for word in processed_doc:
        doc_embedding = doc_embedding + en_embeddings.get(word, 0)
    return doc_embedding
```

Get document vecs

- Store all tweet embeddings into a dictionary

```
def get_document_vecs(all_docs, en_embeddings,
get_document_embedding=get_document_embedding):
    ind2Doc_dict = {}
    document_vec_l = []
    for i, doc in enumerate(all_docs):
        doc_embedding = get_document_embedding(doc, en_embeddings)
        ind2Doc_dict[i] = doc_embedding
        document_vec_l.append(doc_embedding)
    document_vec_matrix = np.vstack(document_vec_l)
    return document_vec_matrix, ind2Doc_dict
```

Locality sensitive hashing (LSH)

Let's say your data points are plotted like this:

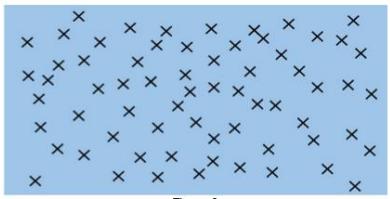


Figure 3

You can divide the vector space into regions and search within one region for nearest neighbors of a given vector.

Figure 4

Hash value of vector

- $$hash = \sum_{i=0}^{N-1} (2^i \times h_i)$$

```
def hash_value_of_vector(v, planes):
    dot_product = np.dot(v, planes)
    sign_of_dot_product = np.sign(dot_product)
    h = sign_of_dot_product >= 0
    h = np.squeeze(h)
    hash_value = 0
    n_planes = planes.shape[1]
    for i in range(n_planes):
        hash_value += np.power(2, i) * h[i]
    hash_value = int(hash_value)
    return hash_value
```

Make hash table

```
def make_hash_table(vecs, planes, hash_value_of_vector=hash_value_of_vector):
    num_of_planes = planes.shape[1]
    num_buckets = 2**num_of_planes
    hash_table = {i: [] for i in range(num_buckets)}
    id_table = {i: [] for i in range(num_buckets)}
    for i, v in enumerate(vecs):
        h = hash_value_of_vector(v, planes)
        hash_table[h].append(v)
        id_table[h].append(i)
    return hash_table, id_table
```

Creating all hash tables

```
def create_hash_id_tables(n_universes):
    hash_tables = []
    id_tables = []
    for universe_id in range(n_universes):  # there are 25 hashes
        print('working on hash universe #:', universe_id)
        planes = planes_l[universe_id]
        hash_table, id_table = make_hash_table(document_vecs, planes)
        hash_tables.append(hash_table)
        id_tables.append(id_table)
    return hash_tables, id_tables
hash_tables, id_tables = create_hash_id_tables(N_UNIVERSES)
```

Approximate KNN

```
def approximate_knn(doc_id, v, planes_l, hash_tables, id_tables, k=1,
num_universes_to_use=25, hash_value_of_vector=hash_value_of_vector):
    vecs_to_consider_l = list()
    ids_to_consider_l = list()
    ids_to_consider_set = set()
    for universe_id in range(num_universes_to_use):
        planes = planes_l[universe_id]
        hash_value = hash_value_of_vector(v, planes)
        hash_table = hash_tables[universe_id]
        document_vectors_l = hash_table[hash_value]
        id_table = id_tables[universe_id]
        new_ids_to_consider = id_table[hash_value]
        for i, new_id in enumerate(new_ids_to_consider):
            if doc_id == new_id:
                continue
            if new_id not in ids_to_consider_set:
                document_vector_at_i = document_vectors_l[i]
                vecs_to_consider_l.append(document_vector_at_i)
                ids_to_consider_l.append(new_id)
                ids_to_consider_set.add(new_id)
    print("Fast considering %d vecs" % len(vecs_to_consider_l))
    vecs_to_consider_arr = np.array(vecs_to_consider_l)
    nearest_neighbor_idx_l = nearest_neighbor(v, vecs_to_consider_arr, k=k)
    nearest_neighbor_ids = [ids_to_consider_l[idx]
                            for idx in nearest_neighbor_idx_l]
    return nearest_neighbor_ids
```

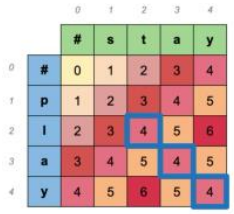
Overview of Autocorrect

- How it works:
    - Identify a misspelled word

```
1. If word not in vocab:
2.     Misspelled = True
```

    - Find strings n edit distance away
        - Number of edit operation away one string from another
        - Insert, delete, replace(a letter), switch (swap 2 adjacent letters)
    - Filter candidates
    - Calculate word probabilities
        - Make a frequence table storing probability for each word
            - Number of times the word appears/total size of the corpus

Minimum edit distance

- Edit cost:
    - Insert(1), Delete(1), Replace(2)
    - Examples:
        - play → stay:
            - p → s: replace(2)
            - l → t: replace(2)
        - total edit distance = 2+2 = 4
- dynamic programming
    - 

		#	s	t	a	y
0	#	0	1	2	3	4
1	p	1	2	3	4	5
2	l	2	3	4	5	6
3	a	3	4	5	4	5
4	y	4	5	6	5	4

    - $D[i,j] =$

$$\min \begin{cases} D[i-1,j] + del\_cost \\ D[i,j-1] + ins\_cost \\ D[i-1,j-1] + \begin{cases} rep\_cost; & \text{if } src[i] \neq tar[j] \\ 0; & \text{if } src[i] = tar[j] \end{cases} \end{cases}$$

Auto Correct Lab

- Edit Distance
  - $$P(c|w) = \frac{P(w|c) \times P(c)}{P(w)}$$
- Data Preprocessing

```
def process_data(file_name):
    words = [] # return this variable correctly
    with open(file_name) as f:
        file_name_data = f.read()
    file_name_data = file_name_data.lower()
    words = re.findall('\w+', file_name_data)
    return words
```

- Get count

```
def get_count(word_l):
    word_count_dict = {}  # fill this with word counts
    word_count_dict = Counter(word_l)
    return word_count_dict
```

- Get probability of word count
  - $$P(w_i) = \frac{C(w_i)}{M}$$

```
def get_probs(word_count_dict):
    probs = {}  # return this variable correctly
    m = sum(word_count_dict.values())
    for key in word_count_dict.keys():
        probs[key] = word_count_dict[key] / m
    return probs
```

- Delete letter

```
def delete_letter(word, verbose=False):
    delete_l = []
    split_l = []
    for c in range(len(word)):
        split_l.append((word[:c], word[c:]))
    for a,b in split_l:
        delete_l.append(a+b[1:])
    if verbose: print(f"input word {word}, \nsplit_l = {split_l}, \ndelete_l = {delete_l}")
    return  delete_l
```

- Switch letter

```
def switch_letter(word, verbose=False):
    switch_l = []
    split_l = []
    word_len = len(word)
    for c in range(word_len):
        split_l.append((word[:c], word[c:]))
    switch_l = [a + b[1] + b[0] + b[2:] for a,b in split_l if len(b) >= 2]
    if verbose: print(f"Input word = {word} \nsplit_l = {split_l} \nswitch_l = {switch_l}")
    return switch_l
```

- Replace

```
def replace_letter(word, verbose=False):
    letters = 'abcdefghijklmnopqrstuvwxyz'
    replace_l = []
    split_l = []
    for c in range(len(word)):
        split_l.append((word[0:c], word[c:]))
    replace_l = [a + l + (b[1:] if len(b) > 1 else '') for a,b in split_l if b for l in letters]
    replace_set = set(replace_l)
    replace_set.remove(word)
    replace_l = sorted(list(replace_set))
    if verbose: print(f"Input word = {word} \nsplit_l = {split_l} \nreplace_l {replace_l}")
    return replace_l
```

- Insert

```
def insert_letter(word, verbose=False):
    letters = 'abcdefghijklmnopqrstuvwxyz'
    insert_l = []
    split_l = []
    for c in range(len(word)+1):
        split_l.append((word[0:c], word[c:]))
    insert_l = [a + l + b for a,b in split_l for l in letters]
    if verbose: print(f"Input word {word} \nsplit_l = {split_l} \ninsert_l = {insert_l}")
    return insert_l
```

- Edit one letter → get all possible edits

```
def edit_one_letter(word, allow_switches = True):
    edit_one_set = set()
    edit_one_set = set(insert_letter(word)+delete_letter(word)+replace_letter(word))
    if(allow_switches):
        edit_one_set = edit_one_set.union(set(switch_letter(word,verbose=False)))
    return set(edit_one_set)
```

- Edit two letters

```
def edit_two_letters(word, allow_switches = True):
    edit_two_set = set()
    edit_one_set = edit_one_letter(word,allow_switches=allow_switches)
    edit_two_set = edit_one_set
    for w in edit_one_set:
        edit_two_set = edit_two_set.union(set(edit_one_letter(w,allow_switches=allow_switches)))
    return set(edit_two_set)
```

- Get correction

```
def get_corrections(word, probs, vocab, n=2, verbose = False):
    suggestions = []
    n_best = []
    temp=[]
    if word in probs.keys():
        temp+=[(word,probs[word])]
    temp+=sorted([(w,probs[w]) for w in edit_one_letter(word) if w in probs],key = lambda x : -x[1])
    temp+=sorted([(w,probs[w]) for w in edit_two_letters(word) if w in probs],key = lambda x: -x[1])
    suggestions = set([sugg[0] for sugg in temp[:n]])
    n_best = [sugg for sugg in temp[:n]]
    if verbose: print("entered word = ", word, "\nsuggestions = ", suggestions)
    return n_best
```

Minimum edit distance

Initialization

$$D[0,0] = 0$$
$$D[i,0] = D[i-1,0] + del\_cost(source[i])$$
$$D[0,j] = D[0,j-1] + ins\_cost(target[j])$$

Per Cell Operations

$$D[i,j] = min \begin{cases} D[i-1,j] + del\_cost \\ D[i,j-1] + ins\_cost \\ D[i-1,j-1] + \begin{cases} rep\_cost; & if\ src[i] \neq tar[j] \\ 0; & if\ src[i] = tar[j] \end{cases} \end{cases}$$

```
def min_edit_distance(source, target, ins_cost = 1, del_cost = 1, rep_cost = 2):
    m = len(source)
    n = len(target)
    D = np.zeros((m+1, n+1), dtype=int)
    for row in range(0,m+1): # Replace None with the proper range
        D[row,0] = row*ins_cost
    for col in range(0,n+1): # Replace None with the proper range
        D[0,col] = col*del_cost
    for row in range(1,m+1):
        for col in range(1,n+1):
            r_cost = rep_cost
            if source[row-1]==target[col-1]:
                r_cost = 0
            D[row,col] = min(D[row-1,col-1]+r_cost , D[row-1,col]+del_cost , D[row,col-1]+ins_cost)
    med = D[m,n]
    return D, med
```

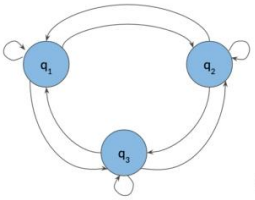
Part of Speech Tagging

lexical term	tag	example
noun	NN	something, nothing
verb	VB	learn, study
determiner	DT	the, a
w-adverb	WRB	why, where
...	...	

-

Markov Chains

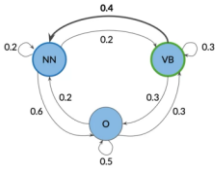
- Stochastic Model: Describes a sequence where each event's probability is dependent only on the state of the previous event.
    - "Stochastic": Indicates randomness within the process.



$Q = \{q_1, q_2, q_3\}$

-
    - Transition Probabilities: Quantifies the likelihood of moving from one state to another.
    - States: Represents the current condition or moment in the chain.

Markov Chains and POS Tags



Why not learn something ?

- Using transition matrix:

$A =$

	NN	VB	O
π (initial)	0.4	0.1	0.5
NN (noun)	0.2	0.2	0.6
VB (verb)	0.4	0.3	0.3
O (other)	0.2	0.3	0.5

    - Written as an actual matrix:

$$A = \begin{pmatrix} 0.4 & 0.1 & 0.5 \\ 0.2 & 0.2 & 0.6 \\ 0.4 & 0.3 & 0.3 \\ 0.2 & 0.3 & 0.5 \end{pmatrix}$$

- Transition matrix A given some states Q:

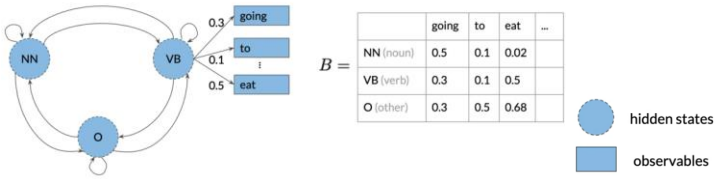
States

$$Q = \{q_1, \ldots, q_N\}$$

Transition matrix

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N+1,1} & \cdots & a_{N+1,N} \end{pmatrix}$$

Hidden Markov Models

- Implies that states are hidden
- Emission probabilities:
  - Describe the transition from the hidden states of the hidden markov
  - 

$B =$

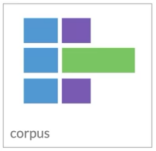
	going	to	eat	...
NN (noun)	0.5	0.1	0.02	
VB (verb)	0.3	0.1	0.5	
O (other)	0.3	0.5	0.68	

hidden states

observables

  - The last row (O) is incorrect, the sum of each row is equal to 1.
    - $\sum_{j=1}^{V} b_{ij} = 1$
  - States $Q = \{q_1, \ldots, q_N\}$ Transition matrix $A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N+1,1} & \cdots & a_{N+1,N} \end{pmatrix}$ Emission matrix $B = \begin{pmatrix} b_{11} & \cdots & b_{1V} \\ \vdots & \ddots & \vdots \\ b_{N1} & \cdots & b_{NV} \end{pmatrix}$

Calculate probability

- Transition probability
  - 

transition probability: ■ + ■ = ⅔
  - Let occurences of tag pairs = $C(t_{i-1}, t_i)$
    - Calculate probabilities using the counts $P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{\sum_{j=1}^{N} C(t_{i-1}, t_j)}$
  - Training:
    - Given a set of corpus, preprocess →
      - <s> in a station of the metro
        <s> the apparition of these faces in the crowd :
        <s> petals on a wet , black bough .
      - Initial probability

Populating the transition matrix

- $A =$

	NN	VB	O
π	1	0	2
NN (noun)	0	0	6
VB (verb)	0	0	0
O (other)	6	0	8

<s> in a station of the metro
<s> the apparition of these faces in the crowd :
<s> petals on a wet , black bough .
Ezra Pound – 1913

- Transition probability:
  - $P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{\sum_{j=1}^{N} C(t_{i-1}, t_j)}$
  - $A =$

	NN	VB	O
π	1+ε	0+ε	2+ε	3+3*ε
NN	0+ε	0+ε	6+ε	6+3*ε
VB	0+ε	0+ε	0+ε	0+3*ε
O	6+ε	0+ε	8+ε	14+3*ε

$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i) + \square}{\sum_{j=1}^{N} C(t_{i-1}, t_j) + N \cdot \square}$

  - with smoothing:
    - $A =$

	NN	VB	O
π	0.3333	0.0003	0.6663
NN	0.0001	0.0001	0.9996
VB	0.3333	0.3333	0.3333
O	0.4285	0.0000	0.5713

    - May not need smooth for initial probability in real world application.

Populating the Emission matrix

- $$P(w_i|t_i) = \frac{C(t_i, w_i) + \epsilon}{\sum_{j=1}^{V} C(t_i, w_j) + N * \epsilon} = \frac{C(t_i, w_i) + \epsilon}{C(t_i) + N * \epsilon}$$
- $B =$

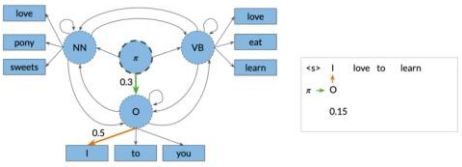
	in	a	...
NN (noun)	0		
VB (verb)	0		
O (other)	2		

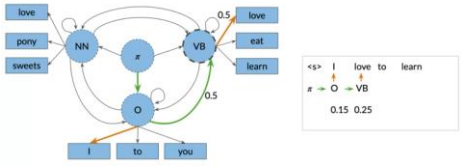
<s> in a station of the metro
<s> the apparition of these faces in the crowd :
<s> petals on a wet , black bough .

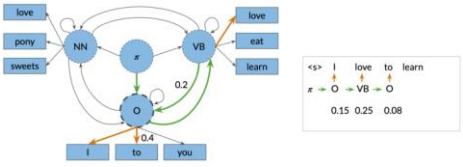
Working with tags

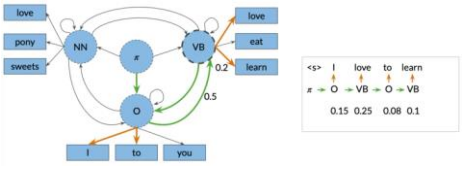
```
tags = ['RB', 'NN', 'TO']
num_tags = len(tags)
transition_matrix = np.zeros((num_tags, num_tags))
transition_matrix
transition_counts = {
    ('NN', 'NN'): 16241,
    ('RB', 'RB'): 2263,
    ('TO', 'TO'): 2,
    ('NN', 'TO'): 5256,
    ('RB', 'TO'): 855,
    ('TO', 'NN'): 734,
    ('NN', 'RB'): 2431,
    ('RB', 'NN'): 358,
    ('TO', 'RB'): 200
}
sorted_tags = sorted(tags)
for i in range(num_tags):
    for j in range(num_tags):
        tag_tuple = (sorted_tags[i], sorted_tags[j])
        transition_matrix[i, j] = transition_counts.get(tag_tuple)
rows_sum = transition_matrix.sum(axis=1, keepdims=True)
transition_matrix = transition_matrix / rows_sum
```

Viterbi Algorithm

- 

- 

- 

- 

- Probability for this sequence of hidden states: 0.15*0.25*0.08*0.1 = 0.0003

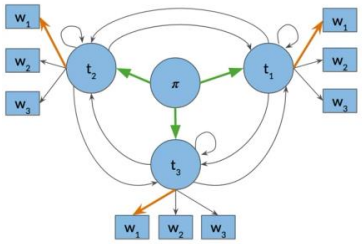
- Procedures:
  - Initialization
  - Forward pass
  - Backward pass
- Two matrices populating:
  - $C =$

	w_1	w_2	...	w_K
t_1				
...				
t_N				

  - $D =$

	w_1	w_2	...	w_K
t_1				
...				
t_N				

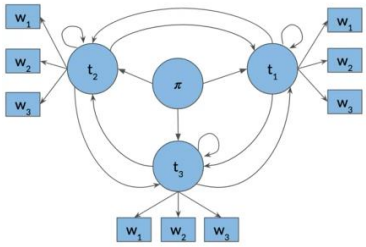
Viterbi Algorithm: initialization

- C matrix: (Transition and Emission Probabilities)
  - 

$C =$

	w_1	w_2	...	w_K
t_1	$c_{1,1}$			
...				
t_N	$c_{N,1}$			

$$c_{i,1} = \pi_i * b_{i,cindex(w_1)}$$
$$= a_{1,i} * b_{i,cindex(w_1)}$$

    - First column: the probability of transitions from startstates to the first tag t_i and w_1.
- D matrix:
  - 

$D =$

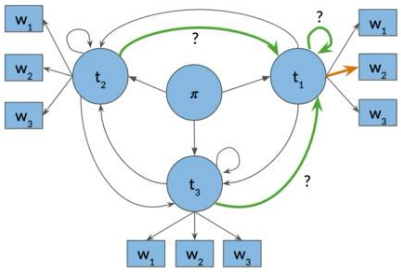
	w_1	w_2	...	w_K
t_1	$d_{1,1}$			
...				
t_N	$d_{N,1}$			

$$d_{i,1} = 0$$

    - Store labels representing the different states traversed while finding the most likely sequence of POS tags for a given seqquence of words, w_1 to w_k.
    - First column: all 0 as there are no preceding POS tags.

Viterbi: Forward Pass

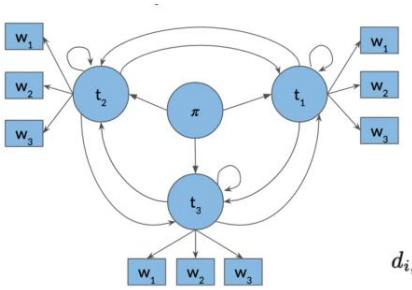
- C matrix:



$$C = \begin{array}{|c|c|c|c|c|} \hline & w_1 & w_2 & \dots & w_K \\ \hline t_1 & c_{1,1} & c_{1,2} & & c_{1,K} \\ \hline \dots & & & & \\ \hline t_N & c_{N,1} & c_{N,2} & & c_{N,K} \\ \hline \end{array}$$

$$c_{1,2} = \max_k c_{k,1} * a_{k,1} * b_{1,cindex(w_2)}$$

  - $b_{1,cindex(w2)}$: the emission probability from tag t_1 to w_2
  - $a_{k,1}$: the transition probability from the POS tag t_k to current tag t_1
  - $c_{k,1}$: the probability for the preceding path you've traversed
- D matrix:



$$D = \begin{array}{|c|c|c|c|c|} \hline & w_1 & w_2 & \dots & w_K \\ \hline t_1 & d_{1,1} & d_{1,2} & & d_{1,K} \\ \hline \dots & & & & \\ \hline t_N & d_{N,1} & d_{N,2} & & d_{N,K} \\ \hline \end{array}$$

$$c_{i,j} = \max_k c_{k,j-1} * a_{k,i} * b_{i,cindex(w_j)}$$

$$d_{i,j} = \operatorname{argmax}_k c_{k,j-1} * a_{k,i} * b_{i,cindex(w_j)}$$

  - Argmax for k maximizing c

Viterbi: Backward Pass

$$C = \begin{array}{|c|c|c|c|c|} \hline & w_1 & w_2 & \dots & w_K \\ \hline t_1 & c_{1,1} & c_{1,2} & & c_{1,K} \\ \hline \dots & & & & \\ \hline t_N & c_{N,1} & c_{N,2} & & c_{N,K} \\ \hline \end{array} \quad D = \begin{array}{|c|c|c|c|c|} \hline & w_1 & w_2 & \dots & w_K \\ \hline t_1 & d_{1,1} & d_{1,2} & & d_{1,K} \\ \hline \dots & & & & \\ \hline t_N & d_{N,1} & d_{N,2} & & d_{N,K} \\ \hline \end{array}$$

$$s = \operatorname{argmax}_i c_{i,K}$$

-
- Example:

$$C = \begin{array}{|c|c|c|c|c|c|} \hline & w_1 & w_2 & w_3 & w_4 & w_5 \\ \hline t_1 & 0.25 & 0.125 & 0.025 & 0.0125 & 0.01 \\ \hline t_2 & 0.1 & 0.025 & 0.05 & 0.01 & 0.003 \\ \hline t_3 & 0.3 & 0.05 & 0.025 & 0.02 & 0.0000 \\ \hline t_4 & 0.2 & 0.1 & 0.000 & 0.0025 & 0.0003 \\ \hline \end{array}$$

$$s = \operatorname{argmax}_i c_{i,K} = 1$$

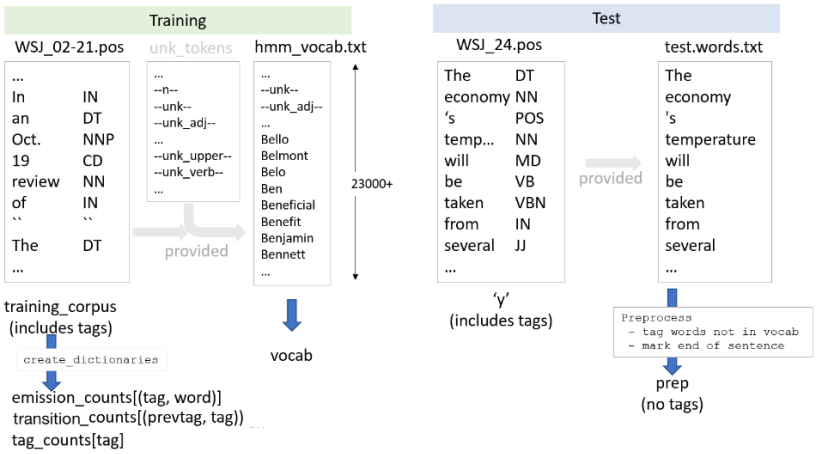
  - $$D = \begin{array}{|c|c|c|c|c|c|} \hline & w_1 & w_2 & w_3 & w_4 & w_5 \\ \hline t_1 & 0 & 1 & 3 & 2 & 3 \\ \hline t_2 & 0 & 2 & 4 & 1 & 3 \\ \hline t_3 & 0 & 2 & 4 & 1 & 4 \\ \hline t_4 & 0 & 4 & 4 & 3 & 1 \\ \hline \end{array}$$

<s> w1 w2 w3 w4 w5

$\pi \leftarrow t_2 \leftarrow t_3 \leftarrow t_1 \leftarrow t_3 \leftarrow t_1$

  -
- Implementation:
  - Use log probabilities instead.

Parts-of-Speech Tagging (POS) lab



-

training_corpus:

- ['In\tIN\n', 'an\tDT\n', 'Oct.\tNNP\n', '19\tCD\n', 'review\tNN\n', ...]

voc_l:

- ['!', '#', '$', '%', '&', """, """", "'40s", ..., 'zip', 'zombie', 'zone', 'zones', 'zoning', '{', '}', '']

vocab (dictionary):

```
:0
!:1
#:2
$:3
%:4
&:5
':6
...
```

y:

- ['The\tDT\n', 'economy\tNN\n', "'s\tPOS\n", 'temperature\tNN\n', ...]

Prep:

- ['The', 'economy', "'s", 'temperature', 'will', 'be', 'taken', 'from', 'several', '--unk--', ...]

Create dictionaries:

```python
def create_dictionaries(training_corpus, vocab, verbose=True):
    emission_counts = defaultdict(int)
    transition_counts = defaultdict(int)
    tag_counts = defaultdict(int)
    prev_tag = '--s--'
    i = 0
    for word_tag in training_corpus:
        i += 1
        if i % 50000 == 0 and verbose:
            print(f"word count = {i}")
        word, tag = get_word_tag(word_tag, vocab)
        transition_counts[(prev_tag, tag)] += 1
        emission_counts[(tag, word)] += 1
        tag_counts[tag] += 1
        prev_tag = tag
    return emission_counts, transition_counts, tag_counts
```

```
transition examples:
(('--s--', 'IN'), 5050)
(('IN', 'DT'), 32364)
(('DT', 'NNP'), 9044)

emission examples:
(('DT', 'any'), 721)
(('NN', 'decrease'), 7)
(('NN', 'insider-trading'), 5)

ambiguous word example:
('RB', 'back') 304
('VB', 'back') 20
('RP', 'back') 84
('JJ', 'back') 25
('NN', 'back') 29
('VBP', 'back') 4
```

```python
print("transition examples: ")
for ex in list(transition_counts.items())[:3]:
    print(ex)
print()
print("emission examples: ")
for ex in list(emission_counts.items())[200:203]:
    print (ex)
print()
print("ambiguous word example: ")
for tup,cnt in emission_counts.items():
    if tup[1] == 'back': print (tup, cnt)
```

```
transition examples:
(('--s--', 'IN'), 5050)
(('IN', 'DT'), 32364)
(('DT', 'NNP'), 9044)

emission examples:
(('DT', 'any'), 721)
(('NN', 'decrease'), 7)
(('NN', 'insider-trading'), 5)

ambiguous word example:
('RB', 'back') 304
('VB', 'back') 20
('RP', 'back') 84
('JJ', 'back') 25
('NN', 'back') 29
('VBP', 'back') 4
```

states:

- Parts-of-speech designations found in the training data
    - ['#', '$', """, '(', ')', ',', '--s--', '.', ':', 'CC', 'CD', 'DT', 'EX', …]
    - https://github.com/clips/MBSP/blob/master/tags.py

Predict POS

```python
def predict_pos(prep, y, emission_counts, vocab, states):
    num_correct = 0
    all_words = set(emission_counts.keys())
    total = 0
    for word, y_tup in zip(prep, y):
        y_tup_l = y_tup.split()
        if len(y_tup_l) == 2:
            true_label = y_tup_l[1]
        else:
            continue
        count_final = 0
        pos_final = ''
        if word in vocab:
            for pos in states:
                key = (pos, word)
                if key in emission_counts:
                    count = emission_counts[key]
                    if count>count_final
                        count_final = count
                        pos_final = pos
            if pos_final == true_label:
                num_correct += 1
        total += 1
    accuracy = num_correct / total
    return accuracy
```

HMM for POS

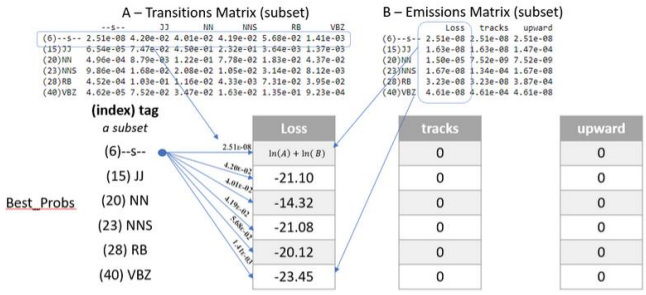
- Constructing “A”: transition probabilities matrix
  - $$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i) + \alpha}{C(t_{i-1}) + \alpha * N}$$

```
def create_transition_matrix(alpha, tag_counts, transition_counts):
    all_tags = sorted(tag_counts.keys())
    num_tags = len(all_tags)
    A = np.zeros((num_tags,num_tags))
    trans_keys = set(transition_counts.keys())
    for i in range(num_tags):
        for j in range(num_tags):
            count = 0
            key = (all_tags[i], all_tags[j
            if key in trans_keys:
                count = transition_counts[key]
            count_prev_tag = tag_counts[key[0]]
            A[i,j] = (count+alpha)/(count_prev_tag+num_tags*(alpha))
    return A
```

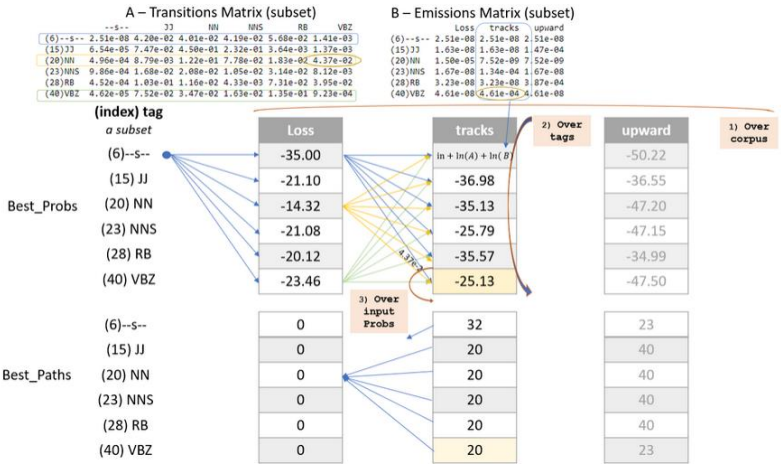
- Constructing “B”: emission probabilities matrix
  - $$P(w_i|t_i) = \frac{C(t_i, word_i) + \alpha}{C(t_i) + \alpha * N}$$

```
def create_emission_matrix(alpha, tag_counts, emission_counts, vocab):
    num_tags = len(tag_counts)
    all_tags = sorted(tag_counts.keys())
    num_words = len(vocab)
    B = np.zeros((num_tags, num_words))
    emis_keys = set(list(emission_counts.keys()))
    for i in range(num_tags):
        for j in range(num_words):
            count = 0
            key = (all_tags[i], vocab[j]) # tuple of form (tag,word)
            if key in emission_counts:
                count = emission_counts[key]
            count_tag = tag_counts[key[0]]
            B[i,j] = (count+alpha)/(count_tag+alpha*num_words)
    return B
```

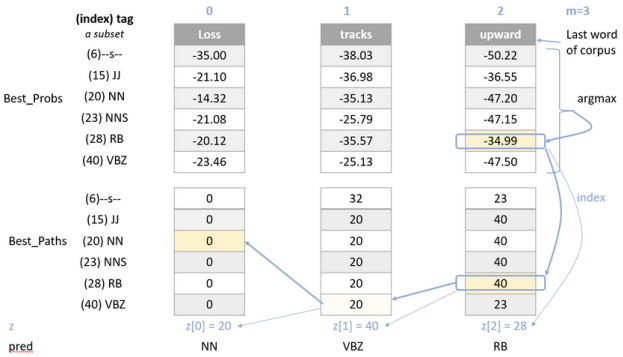
- Viterbi Initalization:
  - Two matrices: best_probs, best_paths(all zeros)
    - $$best\_probs[i,0] = log(A[s_{idx}, i]) + log(B[i, vocab[corpus[0]]$$




```
def initialize(states, tag_counts, A, B, corpus, vocab):
    num_tags = len(tag_counts)
    best_probs = np.zeros((num_tags, len(corpus)))
    best_paths = np.zeros((num_tags, len(corpus)), dtype=int)
    s_idx = states.index("--s--")
    for i in range(num_tags):
        best_probs[i,0] = np.log(A[s_idx,i])+np.log(B[i,vocab[corpus[0]]])
    return best_probs, best_paths
```

- Viterbi Forward:
  - 


```
def viterbi_forward(A, B, test_corpus, best_probs, best_paths, vocab, verbose=True):
num_tags = best_probs.shape[0]
    for i in range(1, len(test_corpus)):
        if i % 5000 == 0 and verbose:
            print("Words processed: {:>8}".format(i))
        for j in range(num_tags):
            best_prob_i = float("-inf")
            best_path_i = None
            for k in range(num_tags):
                prob = best_probs[k, i-1] + math.log(A[k,j]) + math.log(B[j, vocab[test_corpus[i]]])
                if prob > best_prob_i:
                    best_prob_i = prob
                    best_path_i = k
            best_probs[j,i] = best_prob_i
            best_paths[j,i] = best_path_i
    return best_probs, best_paths
```

- Viterbi Backward:
  - 


```
def viterbi_backward(best_probs, best_paths, corpus, states):
    m = best_paths.shape[1]
    z = [None] * m
    num_tags = best_probs.shape[0]
    best_prob_for_last_word = float('-inf')
    pred = [None] * m
    for k in range(num_tags):
        if best_probs[k, -1] > best_prob_for_last_word:
            best_prob_for_last_word = best_probs[k, -1]
            z[m - 1] = k
    pred[m - 1] = states[z[m-1]]
    for i in range(m-1, 0, -1):
        pos_tag_for_word_i = z[i]
        z[i - 1] = best_paths[pos_tag_for_word_i, i]
        pred[i - 1] = states[z[i-1]]
    return pred
```

Compute Accuracy:

```
def compute_accuracy(pred, y):
    num_correct = 0
    total = 0
    for prediction, y in zip(pred, y):
        word_tag_tuple = y.split()
        if len(word_tag_tuple)!=2:
                    continue
        word, tag = word_tag_tuple[0], word_tag_tuple[1]
        if tag==prediction:
                    num_correct += 1
        total += 1
    return num_correct/total
```

AutoComplete: N-Grams

Sequence notation

Corpus: This is great ... teacher drinks tea. $m = 500$

$w_1 \; w_2 \; w_3 \qquad w_{498} \; w_{499} \; w_{500}$

$$w_1^m = w_1 \; w_2 \; \ldots \; w_m$$

$$w_1^3 = w_1 \; w_2 \; w_3$$

$$w_{m-2}^m = w_{m-2} \; w_{m-1} \; w_m$$

-

N-grams

- An N-gram is a sequence of N words (can be characters)
- Example:
  - Corpus: I am happy because I am learning
    - Unigrams: {I, am, happy, …}
      - Probability:

Corpus: I am happy because I am learning

Size of corpus m = 7

$$P(I) = \frac{2}{7} \qquad P(happy) = \frac{1}{7}$$

Probability of unigram: $P(w) = \frac{C(w)}{m}$

    - Bigrams: {I am, am happy, …}
      - Probability:

Corpus: I am happy because I am learning

$$P(am|I) = \frac{C(I\,am)}{C(I)} = \frac{2}{2} = 1 \qquad P(happy|I) = \frac{C(I\,happy)}{C(I)} = \frac{0}{2} = 0$$ ✖ I happy

$$P(learning|am) = \frac{C(am\,learning)}{C(am)} = \frac{1}{2}$$

Probability of a bigram: $P(y|x) = \frac{C(x\;y)}{\sum_w C(x\;w)} = \frac{C(x\;y)}{C(x)}$

    - Trigrams: {I am happy, am happy because, …}
      - Probability:

Corpus: I am happy because I am learning

$$P(happy|I\,am) = \frac{C(I\,am\,happy)}{C(I\,am)} = \frac{1}{2}$$

Probability of a trigram: $P(w_3|w_1^2) = \frac{C(w_1^2\;w_3)}{C(w_1^2)}$

$$C(w_1^2\;w_3) = C(w_1\;w_2\;w_3) = C(w_1^3)$$

    - N-gram probability:;

$$P(w_N|w_1^{N-1}) = \frac{C(w_1^{N-1}\;w_N)}{C(w_1^{N-1})}$$

$$C(w_1^{N-1}\;w_N) = C(w_1^N)$$

      -

Sequence Probability:

Version 1:

- P(I ate an apple) = P(I) * P(ate|I) * P(an|I ate) * P(apple|I ate an)
- Problem:
    - Corpus almost never contains the exact sentence we're interested in or even its longer subsequences.

Approximation of sequence probability

- The product of the probabilities of bigrams.
- P(apple|I ate an) ≈ P(apple | an)

Markov assumption: only last N words matter

- Bigram:
    - $P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1})$
- N-gram:
    - $P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1})$
- Entire sentence with bigram:
    - $P(w_1^n) \approx \prod_{i=1}^{n} P(w_i|w_{i-1})$

Starting and ending sentences

- Start of sentence token <s>
    - For sentence: I ate an apple
    - Bigram: <s>I ate an apple
        - P(<s> I ate an apple) ≈ P(I | <s>) * P(ate | I) * P(an | ate) * P(apple | an)
    - Trigram: <s> <s> I ate an apple
        - $P(w_1 | <s> <s>)P(w_2 | <s>w_1)…P(w_n | w_{n-2}w_{n-1})$
    - N-gram model: add N-1 start tokens <s>
- End of sentence token </s>
    - Motivation:
        - $P(y|x) = \frac{C(x\ y)}{\sum_w C(x\ w)} = \frac{C(x\ y)}{C(x)}$
        - Does not work when:
            - Corpus:
              <s> Lyn drinks chocolate
              <s> John drinks

              $\sum_w C(drinks\ w) = 1$

              $C(drinks) = 2$
            - Corpus:
              <s> yes no
              <s> yes yes
              <s> no no

              Sentences of length 3:
              <s> yes yes yes
              <s> yes yes no
              …
              <s> no no no

              $P(< s >\ \text{yes yes yes}) = \dots$
              $P(< s >\ \text{yes yes no}) = \dots$
              $\dots = \dots$
              $P(< s >\ \text{no no no}) = \dots$

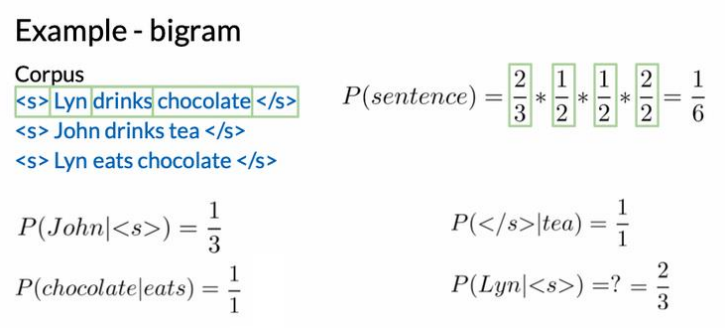
              $\sum_{3\ \text{word}} P(\cdots) = 1$
        - Solution:
            - Bigram

              <s> the teacher drinks tea => <s> the teacher drinks tea </s>

              $P(the|<s>)P(teacher|the)P(drinks|teacher)P(tea|drinks)P(</s>|tea)$
            - Corpus:
              <s> Lyn drinks chocolate </s>
              <s> John drinks </s>

              $\sum_w C(drinks\ w) = 2$

              $C(drinks) = 2$

- 

N-grams Corpus Preprocessing

```
corpus = "Learning% makes 'me' happy. I am happy be-cause I am learning! :)"
corpus = corpus.lower()
corpus = re.sub(r"[^a-zA-Z0-9.?! ]+", "", corpus)
# learning makes me happy. i am happy because i am learning!
input_date="Sat May  9 07:33:35 CEST 2020"
date_parts = input_date.split(" ")
# ['Sat', 'May', '', '9', '07:33:35', 'CEST', '2020']
time_parts = date_parts[4].split(":")
# ['07', '33', '35']
sentence = 'i am happy because i am learning.'
tokenized_sentence = nltk.word_tokenize(sentence)
# ['i', 'am', 'happy', 'because', 'i', 'am', 'learning', '.']
sentence = ['i', 'am', 'happy', 'because', 'i', 'am', 'learning', '.']
word_lengths = [(word, len(word)) for word in sentence]
# [('i', 1), ('am', 2), ('happy', 5), ('because', 7), …, ('.', 1)]
```

Trigram:

```
def sentence_to_trigram(tokenized_sentence):
    for i in range(len(tokenized_sentence) - 3 + 1):
        trigram = tokenized_sentence[i : i + 3]
        print(trigram)
```

The N-gram Language Model

- Count matrix
  - Rows: unique corpus (N-1)-grams
  - Columns: unique corpus words
  - Bigram count matrix

Corpus: <s>I study I learn</s>

	<s>	</s>	I	study	learn
<s>	0	0	1	0	0
</s>	0	0	0	0	0
I	0	0	0	1	1
study	0	0	1	0	0
learn	0	1	0	0	0

"study I" bigram →

- Probability matrix

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}, w_n)}{C(w_{n-N+1}^{n-1})}$$

$$sum(row) = \sum_{w \in V} C(w_{n-N+1}^{n-1}, w) = C(w_{n-N+1}^{n-1})$$

Count matrix (bigram)

	<s>	</s>	I	study	learn	sum
<s>	0	0	1	0	0	1
</s>	0	0	0	0	0	0
I	0	0	0	1	1	2
study	0	0	1	0	0	1
learn	0	1	0	0	0	1

→ Probability matrix

	<s>	</s>	I	study	learn
<s>	0	0	1	0	0
</s>	0	0	0	0	0
I	0	0	0	0.5	0.5
study	0	0	1	0	0
learn	0	1	0	0	0

- Language model

Sentence probability:
<s> I learn </s>

$$P(sentence) = P(I|<s>)P(learn|I)P(</s>|learn) = 1 \times 0.5 \times 1 = 0.5$$

  - Use log probability to prevent numerical underflow.
- Generative Language model:

Corpus:
<s> Lyn drinks chocolate </s>
<s> John drinks tea </s>
<s> Lyn eats chocolate </s>

1. (<s>, Lyn) or (<s>, John)?
2. (Lyn,eats) or (Lyn,drinks) ?
3. (drinks,tea) or (drinks,chocolate)?
4. (tea,</s>) - always

  - Algorithm:
    - Choose sentence start.
    - Choose next bigram starting with previous word.
    - Continue until </s> is picked.

Language Model Evaluation

- Train/val/test split
  - Ratio:
    - For smaller corpora: 80:10:10
    - For larger corpora: 98:1:1
  - Two splitting methods:
    - Split the corpus by choosing longer continuous segments like Wikipedia articles.
    - Randomly choose short sequences of words.
- Perplexity
  - A measure of the complexity in a sample of texts.
  - Used to tell us whether a set of sentences look like they were written by humans rather than by a simple program choosing words at random.

$$PP(W) = P(s_1, s_2, ..., s_m)^{-\frac{1}{m}}$$

W → test set containing **m** sentences **s**
s_i → i-th sentence in the test set, each ending with </s>
m → number of all words in entire test set W including </s> but not including <s>

human: lower, machine: higher

  - Higher language model estimates the probability of the test set → lower

- Perplexity example:

E.g. m=100

$$P(W) = 0.9 => PP(W) = 0.9^{-\frac{1}{100}} = 1.00105416$$

$$P(W) = 10^{-250} => PP(W) = (10^{-250})^{-\frac{1}{100}} \approx 316$$

  - Smaller perplexity = better model
  - Character level models PP< word-based models PP
- Perplexity for bigram models

$$PP(W) = \sqrt[m]{\prod_{i=1}^{m}\prod_{j=1}^{|s_i|}\frac{1}{P(w_j^{(i)}|w_{j-1}^{(i)})}}$$

$w_j^{(i)}$ → j-th word in i-th sentence

- concatenate all sentences in W

$$PP(W) = \sqrt[m]{\prod_{i=1}^{m}\frac{1}{P(w_i|w_{i-1})}}$$

w_i → i-th word in test set

- Log perplexity

$$logPP(W) = -\frac{1}{m}\sum_{i=1}^{m} log_2(P(w_i|w_{i-1}))$$

- Example:

Unigram

Months the my and issue of year foreign new exchange's september were recession exchange new endorsed a acquire to six executives

Bigram

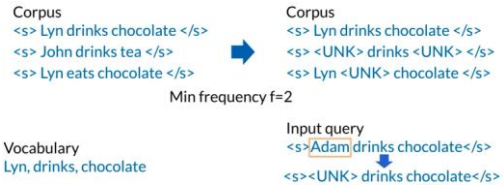
Last December through the way to preserve the Hudson corporation N. B. E. C. Taylor would seem to complete the major central planners one point five percent of U. S. E. has already old M. X. corporation of living on information such as more frequently fishing to keep her

Trigram

They also point to ninety nine point six billion dollars from two hundred four oh six three percent of the rates of interest stores as Mexico and Brazil on market conditions

  - Unigram: 962, Bigram: 170, Trigram: 109

Out of vocabulary words

- Using <UNK> in corpus:
  - Create vocabulary V, replace any word in corpus and not in V by <UNK>, count the probabilties with <UNK> as with any other word.
- Create vocabulary V:
  - Criteria: min word frequency f / Max |V|, including words by frequency
  - Use <UNK> sparingly
  - Perplexity – only compare LMs with the same V
  - 

Smoothing

- Missing N-grams in training corpus
  - Problem: N-grams made of known words still might be missing in the training corpus
    - "John", "eats" in corpus, but "John eats" missing
    - $$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}, w_n)}{C(w_{n-N+1}^{n-1})}$$ ← Can be 0
- Add-one smoothing (Laplacian smoothing)
  - $$P(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n) + 1}{\sum_{w \in V}(C(w_{n-1}, w) + 1)} = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + V}$$
- Add-k smoothing
  - $$P(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n) + k}{\sum_{w \in V}(C(w_{n-1}, w) + k)} = \frac{C(w_{n-1}, w_n) + k}{C(w_{n-1}) + k * V}$$
- Kneser-Ney smoothing
- Good-Turing smoothing

Backoff

- If N-gram missing => use (N-1)-gram
  - Probability discounting e.g. Katz backoff
  - "Stupid" backoff
- Corpus
  <s> Lyn drinks chocolate </s>
  <s> John drinks tea </s>
  <s> Lyn eats chocolate </s>

  $P(chocolate|John\ drinks) = ?$

  ⬇

  $0.4 \times P(chocolate|drinks)$

Interpolation

- Use all (assign different potions to different grams)
- $$\hat{P}(w_n|w_{n-2}\ w_{n-1}) = \lambda_1 \times P(w_n|w_{n-2}\ w_{n-1}) + \lambda_2 \times P(w_n|w_{n-1}) + \lambda_3 \times P(w_n) \qquad \sum_i \lambda_i = 1$$
- Example:
  - $$\hat{P}(chocolate|John\ drinks) = 0.7 \times P(chocolate|John\ drinks) + 0.2 \times P(chocolate|drinks) + 0.1 \times P(chocolate)$$

Building the language model

Count matrix

```
def single_pass_trigram_count_matrix(corpus):
    bigrams = []
    vocabulary = []
    count_matrix_dict = defaultdict(dict)
    for i in range(len(corpus) - 3 + 1):
        trigram = tuple(corpus[i : i + 3])
        bigram = trigram[0 : -1]
        if not bigram in bigrams:
            bigrams.append(bigram)
        last_word = trigram[-1]
        if not last_word in vocabulary:
            vocabulary.append(last_word)
        if (bigram,last_word) not in count_matrix_dict:
            count_matrix_dict[bigram,last_word] = 0
        count_matrix_dict[bigram,last_word] += 1
    count_matrix = np.zeros((len(bigrams), len(vocabulary)))
    for trigram_key, trigam_count in count_matrix_dict.items():
        count_matrix[bigrams.index(trigram_key[0]), \
                     vocabulary.index(trigram_key[1])]\
        = trigam_count
    count_matrix = pd.DataFrame(count_matrix, index=bigrams, columns=vocabulary)
    return bigrams, vocabulary, count_matrix
```

```
corpus = ['i', 'am', 'happy', 'because', 'i', 'am', 'learning', '.']
bigrams, vocabulary, count_matrix = single_pass_trigram_count_matrix(corpus)
print(count_matrix)
```

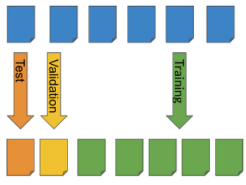
```
                  happy  because    i   am  learning    .
(i, am)             1.0      0.0  0.0  0.0       1.0  0.0
(am, happy)         0.0      1.0  0.0  0.0       0.0  0.0
(happy, because)    0.0      0.0  1.0  0.0       0.0  0.0
(because, i)        0.0      0.0  0.0  1.0       0.0  0.0
(am, learning)      0.0      0.0  0.0  0.0       0.0  1.0
```

Probability matrix

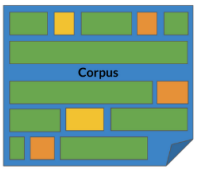
```
row_sums = count_matrix.sum(axis=1)
prob_matrix = count_matrix.div(row_sums, axis=0)
print(prob_matrix)
```

```
                  happy  because    i   am  learning    .
(i, am)             0.5      0.0  0.0  0.0       0.5  0.0
(am, happy)         0.0      1.0  0.0  0.0       0.0  0.0
(happy, because)    0.0      0.0  1.0  0.0       0.0  0.0
(because, i)        0.0      0.0  0.0  1.0       0.0  0.0
(am, learning)      0.0      0.0  0.0  0.0       0.0  1.0
```

- Continuous text



- Random short sequences



LM – AutoComplete

```
import math
import random
import numpy as np
import pandas as pd
import nltk
nltk.download('punkt')
import w3_unittest
nltk.data.path.append('.')
```

- Load and preprocess data
  - Load and tokenize data

```
#data – type: <class: 'str'>
#3335477 number of letters
```

```
def split_to_sentences(data):
    sentences = data.split("\n")
    sentences = [s.strip() for s in sentences]
    sentences = [s for s in sentences if len(s) > 0]
    return sentences
```

```
def tokenize_sentences(sentences):
    tokenized_sentences = []
    for sentence in sentences: # complete this line
        sentence = sentence.lower()
        tokenized = nltk.word_tokenize(sentence)
        tokenized_sentences.append(tokenized)
    return tokenized_sentences
```

```
def get_tokenized_data(data):
    sentences = split_to_sentences(data)
    tokenized_sentences = tokenize_sentences(sentences)
    return tokenized_sentences
```

  - Split the sentences into train and test sets.

```
tokenized_data = get_tokenized_data(data)
random.seed(87)
random.shuffle(tokenized_data)
train_size = int(len(tokenized_data) * 0.8)
train_data = tokenized_data[0:train_size]
test_data = tokenized_data[train_size:]
```

  - Replace words with a low frequency by an unknown marker <UNK>

```
def count_words(tokenized_sentences):
    word_counts = {}
    for sentence in tokenized_sentences: # complete this line
        for token in sentence: # complete this line
            if token not in word_counts.keys():
                word_counts[token] = 1
            else:
                word_counts[token] += 1
    return word_counts
```

```
def get_words_with_nplus_frequency(tokenized_sentences,
count_threshold):
    closed_vocab = []
    word_counts = count_words(tokenized_sentences)
    for word, cnt in word_counts.items(): # complete this line
        if word_counts[word]>=count_threshold:
            closed_vocab.append(word)
    return closed_vocab
```

```
```

```
def replace_oov_words_by_unk(tokenized_sentences, vocabulary, unknown_token="<unk>"):
    vocabulary = set(vocabulary)
    replaced_tokenized_sentences = []
    for sentence in tokenized_sentences:
        replaced_sentence = []
        for token in sentence:
            if token in vocabulary:
                replaced_sentence.append(token)
            else:
                replaced_sentence.append(unknown_token)
        replaced_tokenized_sentences.append(replaced_sentence)
    return replaced_tokenized_sentences
```

```
def preprocess_data(
train_data, test_data, count_threshold, unknown_token="<unk>",
get_words_with_nplus_frequency=get_words_with_nplus_frequency,
replace_oov_words_by_unk=replace_oov_words_by_unk
):
    vocabulary = get_words_with_nplus_frequency(train_data, count_threshold)
    train_data_replaced = replace_oov_words_by_unk(train_data, vocabulary,unknown_token)
    test_data_replaced = replace_oov_words_by_unk(test_data, vocabulary,unknown_token)
    return train_data_replaced, test_data_replaced, vocabulary
```

- Develop N-gram based language models
  - Compute the count of n-grams from a given data set

```
def count_n_grams(data, n, start_token='<s>', end_token = '<e>'):
    n_grams = {}
    for sentence in data: # complete this line
        sentence = [start_token] * n+ sentence + [end_token]
        sentence = tuple(sentence)
        m = len(sentence) if n==1 else len(sentence)-n+1
        for i in range(m): # complete this line
            n_gram = sentence[i:i+n]
            if n_gram in n_grams.keys(): # complete this line
                n_grams[n_gram] += 1
            else:
                n_grams[n_gram] = 1
    return n_grams
```

  - Estimate the conditional probability of a next word with k-smoothing
  - $$\hat{P}(w_t|w_{t-n}\dots w_{t-1}) = \frac{C(w_{t-n}\dots w_{t-1}, w_t) + k}{C(w_{t-n}\dots w_{t-1}) + k|V|}$$

```
def estimate_probability(word, previous_n_gram, n_gram_counts, n_plus1_gram_counts, vocabulary_size, k=1.0):
    previous_n_gram = tuple(previous_n_gram)
    previous_n_gram_count = n_gram_counts[previous_n_gram] if previous_n_gram in n_gram_counts else 0
    denominator = previous_n_gram_count + k*vocabulary_size
    n_plus1_gram = previous_n_gram+(word,)
    n_plus1_gram_count = n_plus1_gram_counts[n_plus1_gram] if n_plus1_gram in n_plus1_gram_counts else 0
    numerator = n_plus1_gram_count + k
    probability = numerator/denominator
    return probability
```

```python
def estimate_probabilities(previous_n_gram, n_gram_counts, n_plus1_gram_counts, vocabulary, end_token='<e>', unknown_token="<unk>",  k=1.0):
    previous_n_gram = tuple(previous_n_gram)
    vocabulary = vocabulary + [end_token, unknown_token]
    vocabulary_size = len(vocabulary)
    probabilities = {}
    for word in vocabulary:
        probability = estimate_probability(word, previous_n_gram,
                                           n_gram_counts, n_plus1_gram_counts,
                                           vocabulary_size, k=k)
        probabilities[word] = probability
    return probabilities
```

```python
def make_count_matrix(n_plus1_gram_counts, vocabulary):
    vocabulary = vocabulary + ["<e>", "<unk>"]
    n_grams = []
    for n_plus1_gram in n_plus1_gram_counts.keys():
        n_gram = n_plus1_gram[0:-1]
        n_grams.append(n_gram)
    n_grams = list(set(n_grams))
    row_index = {n_gram:i for i, n_gram in enumerate(n_grams)}
    col_index = {word:j for j, word in enumerate(vocabulary)}
    nrow = len(n_grams)
    ncol = len(vocabulary)
    count_matrix = np.zeros((nrow, ncol))
    for n_plus1_gram, count in n_plus1_gram_counts.items():
        n_gram = n_plus1_gram[0:-1]
        word = n_plus1_gram[-1]
        if word not in vocabulary:
            continue
        i = row_index[n_gram]
        j = col_index[word]
        count_matrix[i, j] = count
    count_matrix = pd.DataFrame(count_matrix, index=n_grams, columns=vocabulary)
    return count_matrix
```

```python
def make_probability_matrix(n_plus1_gram_counts, vocabulary, k):
    count_matrix = make_count_matrix(n_plus1_gram_counts, unique_words)
    count_matrix += k
    prob_matrix = count_matrix.div(count_matrix.sum(axis=1), axis=0)
    return prob_matrix
```

```python
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))
bigram_counts = count_n_grams(sentences, 2)

print('bigram counts')
display(make_count_matrix(bigram_counts, unique_words))
```

bigram counts

	i	this	is	like	dog	a	cat	<e>	<unk>
(dog,)	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
(<s>,)	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(a,)	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0
(like,)	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0
(cat,)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0
(i,)	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0
(this,)	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0
(is,)	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0

```python
sentences = [['i', 'like', 'a', 'cat'],
             ['this', 'dog', 'is', 'like', 'a', 'cat']]
unique_words = list(set(sentences[0] + sentences[1]))
bigram_counts = count_n_grams(sentences, 2)
print("bigram probabilities")
display(make_probability_matrix(bigram_counts, unique_words, k=1))
```

bigram probabilities

	i	this	is	like	dog	a	cat	<e>	<unk>
(dog,)	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000	0.100000	0.100000
(<s>,)	0.181818	0.181818	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909
(a,)	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.272727	0.090909	0.090909
(like,)	0.090909	0.090909	0.090909	0.090909	0.090909	0.272727	0.090909	0.090909	0.090909
(cat,)	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.090909	0.272727	0.090909
(i,)	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000	0.100000
(this,)	0.100000	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000
(is,)	0.100000	0.100000	0.100000	0.200000	0.100000	0.100000	0.100000	0.100000	0.100000

Evaluate the N-gram models by computing the perplexity score

$$PP(W) = \sqrt[N]{\prod_{t=n}^{N-1} \frac{1}{P(w_t|w_{t-n}\cdots w_{t-1})}}$$

(array indexing starts from 0)

```
def calculate_perplexity(sentence, n_gram_counts, n_plus1_gram_counts,
vocabulary_size, start_token='<s>', end_token = '<e>', k=1.0):
    n = len(list(n_gram_counts.keys())[0])
    sentence = [start_token] * n + sentence + [end_token]
    sentence = tuple(sentence)
    N = len(sentence)
    product_pi = 1.0
    for t in range(n, N):
        n_gram = sentence[t-n:t]
        word = sentence[t]
        probability =
estimate_probability(word,n_gram,n_gram_counts,n_plus1_gram_counts,vocabulary_size,k)
        product_pi *= 1/probability
    perplexity = (product_pi)**(1/N)
    return perplexity
```

Auto-complete System

```
def suggest_a_word(previous_tokens, n_gram_counts, n_plus1_gram_counts,
vocabulary, end_token='<e>', unknown_token="<unk>", k=1.0, start_with=None):
    n = len(list(n_gram_counts.keys())[0])
    previous_tokens = ['<s>'] * n + previous_tokens
    previous_n_gram = previous_tokens[-n:]
    probabilities = estimate_probabilities(previous_n_gram,
                                           n_gram_counts, n_plus1_gram_counts,
                                           vocabulary, k=k)
    suggestion = None
    max_prob = 0
    for word, prob in probabilities.items(): # complete this line
        if start_with!=None: # complete this line with the proper condition
            if not word.startswith(start_with):
                continue
        if prob > max_prob: # complete this line with the proper condition
            suggestion = word
            max_prob = prob
    return suggestion, max_prob
```

```
def get_suggestions(previous_tokens, n_gram_counts_list, vocabulary, k=1.0,
start_with=None):
    model_counts = len(n_gram_counts_list)
    suggestions = []
    for i in range(model_counts-1):
        n_gram_counts = n_gram_counts_list[i]
        n_plus1_gram_counts = n_gram_counts_list[i+1]
        suggestion = suggest_a_word(previous_tokens, n_gram_counts,
                                    n_plus1_gram_counts, vocabulary,
                                    k=k, start_with=start_with)
        suggestions.append(suggestion)
    return suggestions
```

Basic word representation

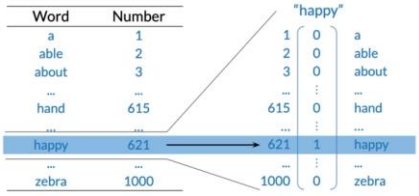
Word	Number
a	1
able	2
about	3
...	...
hand	615
...	...
happy	621
...	...
zebra	1000

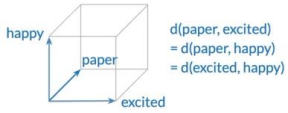
  the ordering doesn't make semantic sense

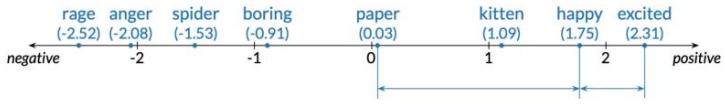
  -

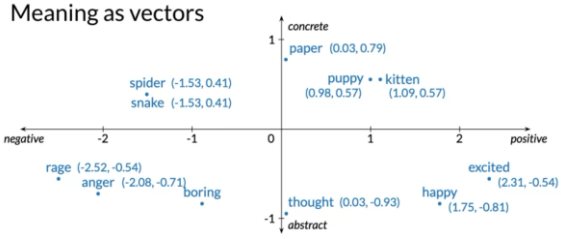
One-hot-vectors

- 

- Advantages:
  - No implied ordering
  - Simple
- Disadvantages:
  - Huge (one million word → one million rows)
  - No embedded meaning
    - 

Word embeddings

- Meaning as vectors
- 

  - Happy is more related to excited than paper
- 

  - Possible for two words to be on the same point (e.g. snake, spider)
- Advantages:
  - Low dimension (~100)
  - Embed meaning
    - Semantic distance
    - Analogies

How to create word embedding

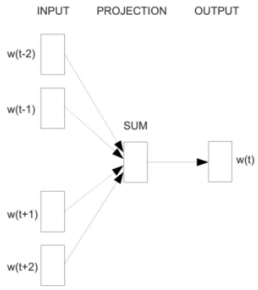
Prepare:

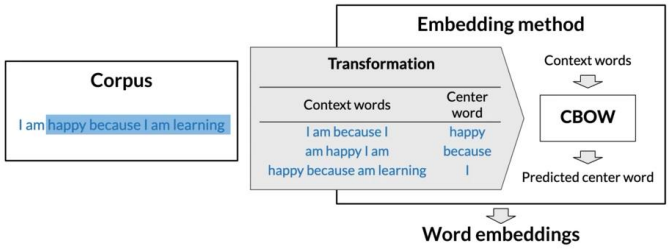
- Corpus:
  - Words in context.
  - Could be general-purpose or specialized.
  - Need to be transformed into machine-readable like vectors before feeding into the model.
- Embedding method:
  - Machine learning model
    - Performs a learning task → define meaning → word embeddings
    - Self-supervised (unsupervised + supervised)
      - The corpus is unlabeled, but it itself provides the necessary context which would ordinarily make up the labels.
      - Dimension of word embedding is one of the hyperparameters.

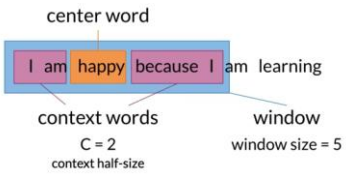
Methods:

- word2vec
  - Continuous bag-of-words (CBOW)
  - Continuous skip-gram / Skip-gram with negative sampling (SGNS)
- Global Vectors (GloVe)
- fastText
  - Supports out-of-vocabulary (OOV) words
- Advanced:
  - Deep learning, contextual embeddings
  - BERT
  - ELMo
  - GPT

Continuous Bag-of-Words (CBOW) Model

- Center word prediction
    - Objective: predict the missing based on the surrounding words
- Rationale:
    - The little _____ is barking
        - Dog, puppy, terrier, …
- Training:
    - 

    - 

    - Example:
        - 

        - C: hyperparameter

Cleaning and Tokenization

- Corpus should be case-insensitive:
  - "The" == "the" == "THE" → *lowercase / upper case*
- Punctuation:
  - , ! . ? →. " ' « » ' " → ∅ ... !! ??? →.
- Numbers:
  - 1 2 3 5 8 → ∅ 3.14159 90210 → *as is/<NUMBER>*
- Special characters:
  - ∇ $ € § ¶ ** →∅
- Special words:
  - Hashtag, emojis

Sliding window of words in python

```
def get_windows(words, C):
    i = C
    while i < len(words) - C:
        center_word = words[i]
        context_words = words[(i-C):i] + words[(i+1):(i+C+1)]
        yield context_words, center_word
        i += 1
```

Transforming words into vectors

- Corpus → vocabulary → one-hot vector
  - Corpus: I am happy because I am learning
  - Vocabulary: am, because, happy, I, learning
  - One-hot vector:

	am	because	happy	I	learning
am	1	0	0	0	0
because	0	1	0	0	0
happy	0	0	1	0	0
I	0	0	0	1	0
learning	0	0	0	0	1

- Take the average of individual one-hot-vectors

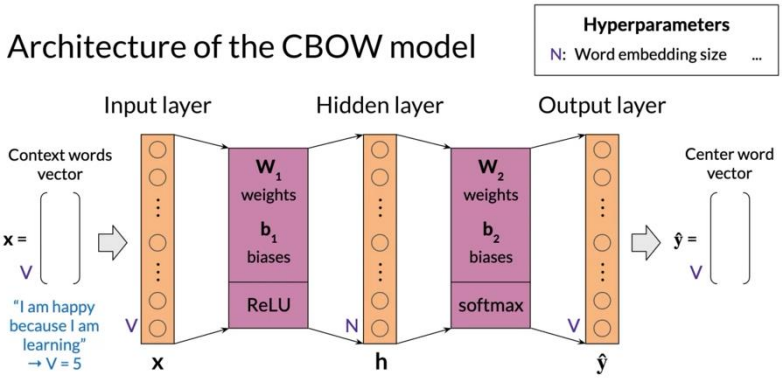
$$\begin{pmatrix} & I & & am & & because & & I & \\ am & 0 & & 1 & & 0 & & 0 & \\ because & 0 & & 0 & & 1 & & 0 & \\ happy & 0 & + & 0 & + & 0 & + & 0 & \Big) / 4 = \begin{matrix} 0.25 \\ 0.25 \\ 0 \\ 0.5 \\ 0 \end{matrix} \\ I & 1 & & 0 & & 0 & & 1 & \\ learning & 0 & & 0 & & 0 & & 0 & \end{pmatrix}$$

- Final prepared training set

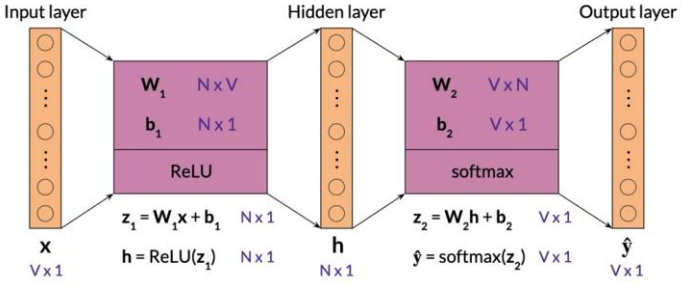
Context words	Context words vector	*Center word*	Center word vector
I am because I	[0.25; 0.25; 0; 0.5; 0]	*happy*	[0; 0; 1; 0; 0]
am happy I am	[0.5; 0; 0.25; 0.25; 0]	*because*	[0; 1; 0; 0; 0]
happy because am learning	[0.25; 0.25; 0.25; 0; 0.25]	*I*	[0; 0; 0; 1; 0]

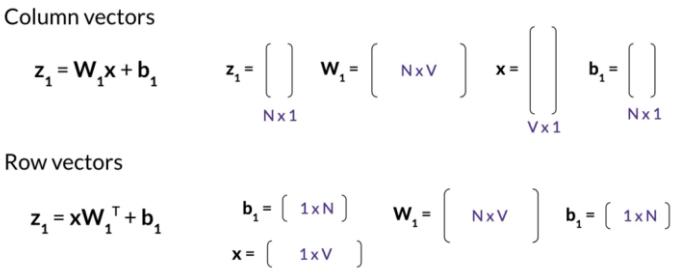
CBOW Model

- Architecture

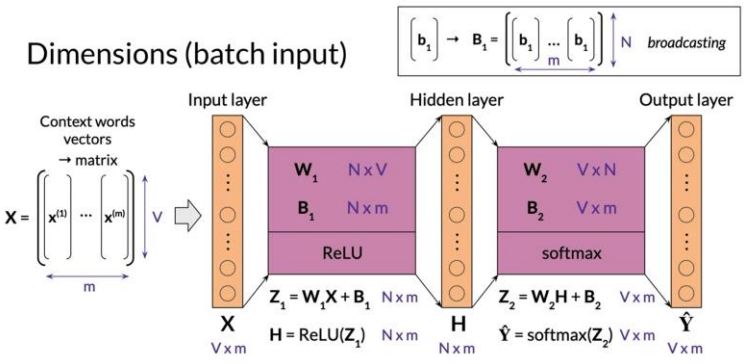


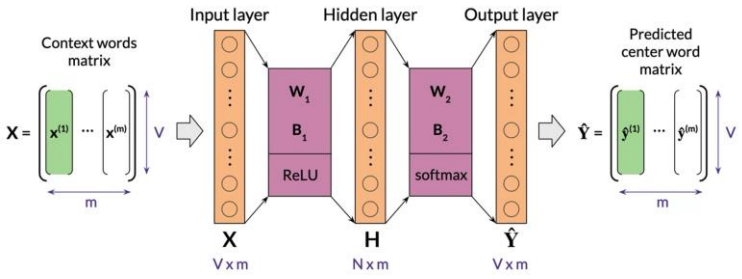
- Dimensions:



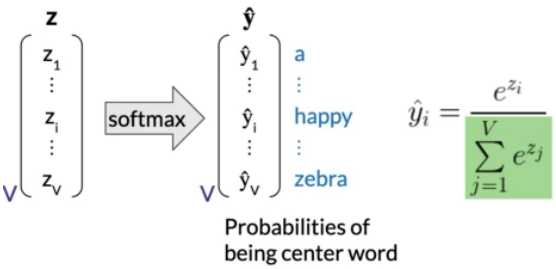


- Dimensions for batch input:





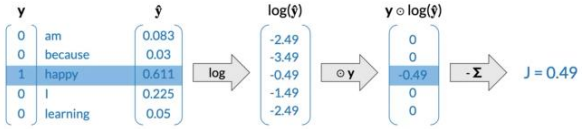
- Activation function:
  - Input layer: ReLU
  - Hidden layer: softmax

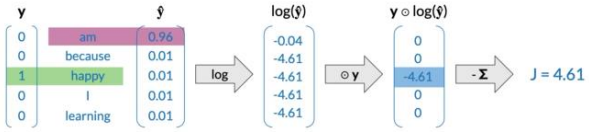


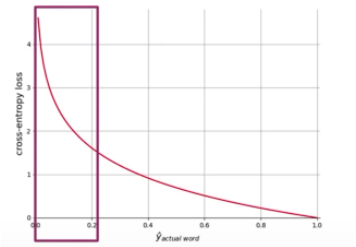
$$\hat{y}_i = \frac{e^{z_i}}{\sum_{j=1}^{V} e^{z_j}}$$

- Cost function:
  - Cross-entropy loss

$$J = -\sum_{k=1}^{V} y_k \log \hat{y}_k$$







- Training process:
  - Forward propagation
  - Cost

$$J_{batch} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{V} y_j^{(i)} \log \hat{y}_j^{(i)}$$

$$J_{batch} = \frac{1}{m}\sum_{i=1}^{m} J^{(i)}$$

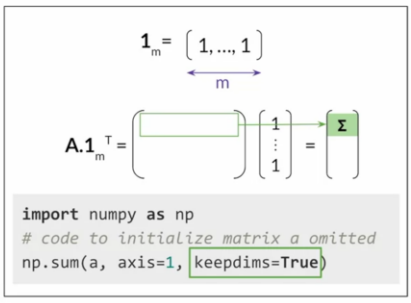
  - Backpropagation and gradient descent

$$\frac{\partial J_{batch}}{\partial \mathbf{W_1}} = \frac{1}{m}(\mathbf{W_2}^\intercal(\hat{\mathbf{Y}} - \mathbf{Y}) \cdot \text{step}(\mathbf{Z_1}))\mathbf{X}^\intercal$$

$$\frac{\partial J_{batch}}{\partial \mathbf{W_2}} = \frac{1}{m}(\hat{\mathbf{Y}} - \mathbf{Y})\mathbf{H}^\intercal$$

$$\frac{\partial J_{batch}}{\partial \mathbf{b_1}} = \frac{1}{m}(\mathbf{W_2}^\intercal(\hat{\mathbf{Y}} - \mathbf{Y}) \cdot \text{step}(\mathbf{Z_1}))\mathbf{1}_m^\intercal$$

$$\frac{\partial J_{batch}}{\partial \mathbf{b_2}} = \frac{1}{m}(\hat{Y} - Y)1_m^T$$



Word Embeddings: Training the CBOW model:

```
# size of the word embedding vectors
N = 3
# size of the vocabulary
V = 5
```

```
# predefined W1 (3 x 5), W2 (5 x 3), b1 (3 x 1), b2 (5 x 1)
# W1: (N x V), W2: (V x N),  b1: (N x 1), b2: (V x 1)
```

```
words = ['i', 'am', 'happy', 'because', 'i', 'am', 'learning']
word2Ind, Ind2word = get_dict(words)
def get_windows(words, C):
    i = C
    while i < len(words) - C:
        center_word = words[i]
        context_words = words[(i - C):i] + words[(i+1):(i+C+1)]
        yield context_words, center_word
        i += 1
def word_to_one_hot_vector(word, word2Ind, V):
    one_hot_vector = np.zeros(V)
    one_hot_vector[word2Ind[word]] = 1
    return one_hot_vector
def context_words_to_vector(context_words, word2Ind, V):
    context_words_vectors = [word_to_one_hot_vector(w, word2Ind, V) for w in context_words]
    context_words_vectors = np.mean(context_words_vectors, axis=0)
    return context_words_vectors
def get_training_example(words, C, word2Ind, V):
    for context_words, center_word in get_windows(words, C):
        yield context_words_to_vector(context_words, word2Ind, V),
word_to_one_hot_vector(center_word, word2Ind, V)
```

```
x = x_array.copy()
x.shape = (V, 1)
y = y_array.copy()
y.shape = (V, 1)
```

```
def relu(z):
    result = z.copy()
    result[result < 0] = 0
    return result
def softmax(z):
    e_z = np.exp(z)
    sum_e_z = np.sum(e_z)
    return e_z / sum_e_z
```

```
z1 = np.dot(W1, x) + b1
h = relu(z1)
z2 = np.dot(W2, h) + b2
y_hat = softmax(z2)
```

```
def cross_entropy_loss(y_predicted, y_actual):
    loss = -np.sum(np.dot(y_actual.T,np.log(y_predicted)))
    return loss
```

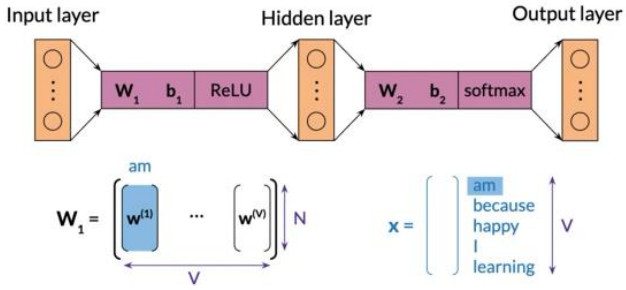
Extracting word embedding vectors

Word Embedding:

```
words = ['i', 'am', 'happy', 'because', 'i', 'am', 'learning']
V = 5
word2Ind, Ind2word = get_dict(words)
W1 = np.array([[ 0.41687358,  0.08854191, -0.23495225,  0.28320538,  0.41800106],
               [ 0.32735501,  0.22795148, -0.23951958,  0.4117634 , -0.23924344],
               [ 0.26637602, -0.23846886, -0.37770863, -0.11399446,  0.34008124]])
W2 = np.array([[-0.22182064, -0.43008631,  0.13310965],
               [ 0.08476603,  0.08123194,  0.1772054 ],
               [ 0.1871551 , -0.06107263, -0.1790735 ],
               [ 0.07055222, -0.02015138,  0.36107434],
               [ 0.33480474, -0.39423389, -0.43959196]])
b1 = np.array([[ 0.09688219],
               [ 0.29239497],
               [-0.27364426]])
```

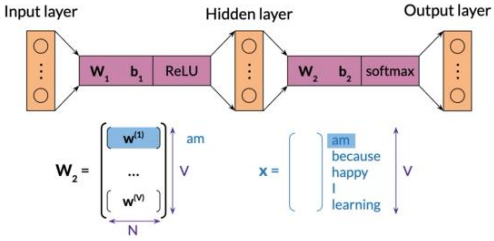
```
b2 = np.array([[ 0.0352008 ],
               [-0.36393384],
               [-0.12775555],
               [-0.34802326],
               [-0.07017815]])
```

- Option 1:



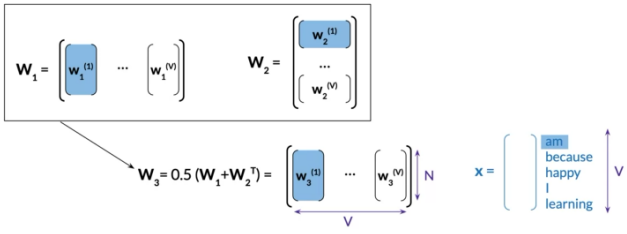

```
for word in word2Ind:
    word_embedding_vector = W1[:, word2Ind[word]]
    print(f'{word}: {word_embedding_vector}')
```

- Option 2:




```
for word in word2Ind:
    word_embedding_vector = W2.T[:, word2Ind[word]]
    print(f'{word}: {word_embedding_vector}')
```

- Option 3:

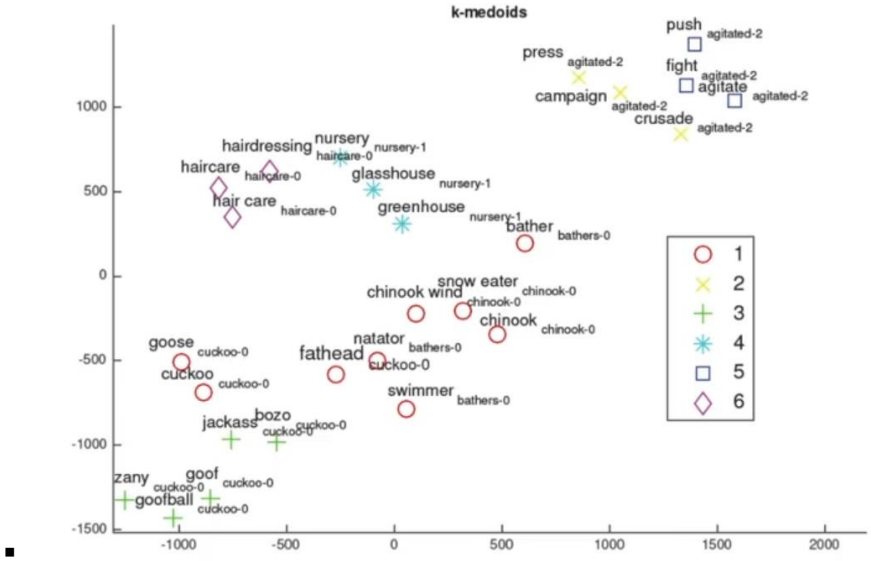



```
W3 = (W1+W2.T)/2
for word in word2Ind:
    word_embedding_vector = W3[:, word2Ind[word]]
    print(f'{word}: {word_embedding_vector}')
```

Evaluating word embeddings: intrinsic evaluation

- Test relationships between words
  - Analogies
    - Semantic analogies:
      - France is to Paris as Italy is to <?>
    - Syntactic analogies:
      - Seen is to saw as been is to <?>
    - There could be ambiguity (can have many different answers)

- Clustering



Evaluating Word Embedding: Extrinsic Evaluation

- Test word embedding on external task
  - E.g. named entity recognition, parts-of-speech tagging
- Advantages:
  - Evaluates actual usefulness of embeddings
- Disadvantages:
  - Time-consuming
  - More difficult t o troubleshoot

Word Embedding lab

Utils:

```
def get_batches(data, word2Ind, V, C, batch_size):
    batch_x = []
    batch_y = []
    for x, y in get_vectors(data, word2Ind, V, C):
        if len(batch_x) < batch_size:
            batch_x.append(x)
            batch_y.append(y)
        else:
            yield np.array(batch_x).T, np.array(batch_y).T
            batch_x = []
            batch_y = []
```

```
def compute_pca(data, n_components=2):
    m, n = data.shape
    data -= data.mean(axis=0)
    R = np.cov(data, rowvar=False)
    evals, evecs = linalg.eigh(R)
    idx = np.argsort(evals)[::-1]
    evecs = evecs[:, idx]
    evals = evals[idx]
    evecs = evecs[:, :n_components]
    return np.dot(evecs.T, data.T).T
```

Load, tokenize and process the data:

```
with open('./data/shakespeare.txt') as f:
    data = f.read()
data = re.sub(r'[,!?;-]', '.',data)
data = nltk.word_tokenize(data)
```

```
5. data = [ ch.lower() for ch in data if ch.isalpha() or ch == '.']
```

Data:

- ['o', 'for', 'a', 'muse', 'of', 'fire', '.', 'that', 'would', 'ascend', 'the', …]

Compute the frequency distribution of the words in the dataset

```
1. fdist = nltk.FreqDist(word for word in data)
```

fdist.most_common(20):

- [('.', 9630), ('the', 1521), ('and', 1394), ('i', 1257), ('to', 1159), …]

Mapping words to indices and viceversa:

```
1. word2Ind, Ind2word = get_dict(data)
2. #word2Ind['king'] = 2745, Ind2word[2745] = 'king'
```

Initialize model

```
1. def initialize_model(N,V, random_seed=1):
2.     np.random.seed(random_seed)
3.     W1 = np.random.rand(N,V)
4.     W2 = np.random.rand(V,N)
5.     b1 = np.random.rand(N,1)
6.     b2 = np.random.rand(V,1)
7.     return W1, W2, b1, b2
```

Softmax:

```
1. def softmax(z):
2.     yhat = np.exp(z)/np.sum(np.exp(z),0) #sum columns
3.     return yhat
```

Forward prop:

- Implement the forward propagation z according to equations (1) to (3).

$$h = W_1 X + b_1 \quad (1)$$
$$h = ReLU(h) \quad (2)$$
$$z = W_2 h + b_2 \quad (3)$$

For that, you will use as activation the Rectified Linear Unit (ReLU) given by:

$$f(h) = \max(0, h) \quad (6)$$

```
1. def forward_prop(x, W1, W2, b1, b2):
2.     h = np.dot(W1, x) + b1
3.     h = np.maximum(0, h)
4.     z = np.dot(W2, h) + b2
5.     return z, h
```

Cost function (cross-entropy)

```
1. def compute_cost(y, yhat, batch_size):
2.     logprobs = np.multiply(np.log(yhat),y)
3.     cost = - 1/batch_size * np.sum(logprobs)
4.     cost = np.squeeze(cost)
5.     return cost
```

Back prop:

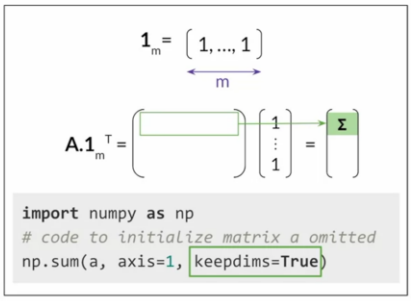
$$\frac{\partial J_{batch}}{\partial \mathbf{W_1}} = \frac{1}{m}(\mathbf{W_2}^\intercal(\hat{\mathbf{Y}} - \mathbf{Y}) \cdot \text{step}(\mathbf{Z_1}))\mathbf{X}^\intercal$$

$$\frac{\partial J_{batch}}{\partial \mathbf{W_2}} = \frac{1}{m}(\hat{\mathbf{Y}} - \mathbf{Y})\mathbf{H}^\intercal$$

$$\frac{\partial J_{batch}}{\partial \mathbf{b_1}} = \frac{1}{m}(\mathbf{W_2}^\intercal(\hat{\mathbf{Y}} - \mathbf{Y}) \cdot \text{step}(\mathbf{Z_1}))\mathbf{1}_m^\intercal$$

$$\frac{\partial J_{batch}}{\partial \mathbf{b_2}}$$

-




```python
def back_prop(x, yhat, y, h, W1, W2, b1, b2, batch_size):
    z1 = np.dot(W1, x) + b1
    l1 = np.dot(W2.T, (yhat-y))
    l1[z1 < 0] = 0 # use "l1" to compute gradients below
    grad_W1 = np.dot(l1, x.T) / batch_size
    grad_W2 = np.dot((yhat-y), h.T) / batch_size
    grad_b1 = np.sum(l1, axis=1, keepdims=True) / batch_size
    grad_b2 = np.sum(yhat-y, axis=1, keepdims = True) / batch_size
    return grad_W1, grad_W2, grad_b1, grad_b2
```

Gradient Descent:

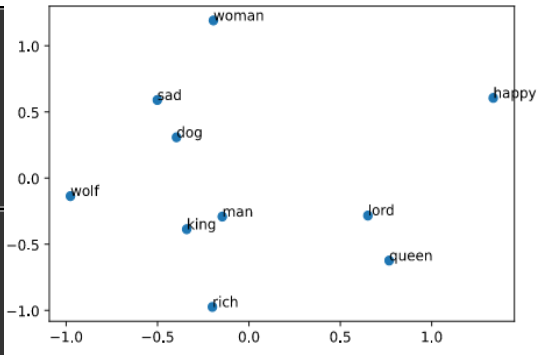
```python
def gradient_descent(data, word2Ind, N, V, num_iters, alpha=0.03,
                     random_seed=282, initialize_model=initialize_model,
                     get_batches=get_batches, forward_prop=forward_prop,
                     softmax=softmax, compute_cost=compute_cost,
                     back_prop=back_prop):
    W1, W2, b1, b2 = initialize_model(N,V, random_seed=random_seed) #W1=(N,V) and W2=(V,N)
    batch_size = 128
    iters = 0
    C = 2
    for x, y in get_batches(data, word2Ind, V, C, batch_size):
        z, h = forward_prop(x,W1,W2,b1,b2)
        yhat = softmax(z)
        cost = compute_cost(y,yhat,batch_size)
        if ( (iters+1) % 10 == 0):
            print(f"iters: {iters + 1} cost: {cost:.6f}")
        grad_W1, grad_W2, grad_b1, grad_b2 = back_prop(x,yhat,y,h,W1,W2,b1,b2,batch_size)
        W1 = W1-alpha*grad_W1
        W2 = W2-alpha*grad_W2
        b1 = b1-alpha*grad_b1
        b2 = b2-alpha*grad_b2
        iters +=1
        if iters == num_iters:
            break
        if iters % 100 == 0:
```

```
            alpha *= 0.66
    return W1, W2, b1, b2
```

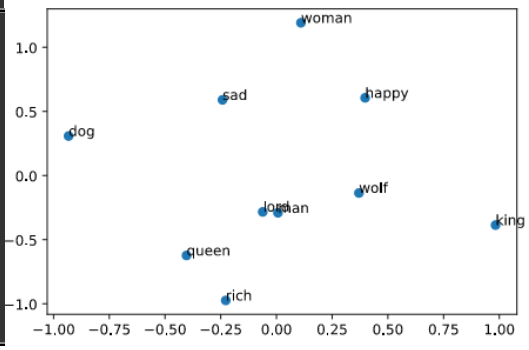
Visualization

```
from matplotlib import pyplot
%config InlineBackend.figure_format = 'svg'
words = ['king', 'queen','lord','man', 'woman','dog',
         'wolf', 'rich','happy','sad']
embs = (W1.T + W2)/2.0
idx = [word2Ind[word] for word in words]
X = embs[idx, :]
```

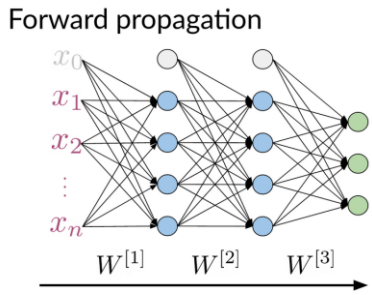
```
result= compute_pca(X, 2)
pyplot.scatter(result[:, 0], result[:, 1])
for i, word in enumerate(words):
    pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```




```
result= compute_pca(X, 4)
pyplot.scatter(result[:, 3], result[:, 1])
for i, word in enumerate(words):
    pyplot.annotate(word, xy=(result[i, 3], result[i, 1]))
pyplot.show()
```



Neural Network:



$a^{[i]}$ Activations ith layer

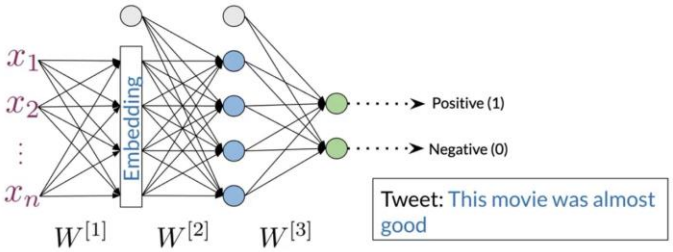
$$a^{[0]} = X$$

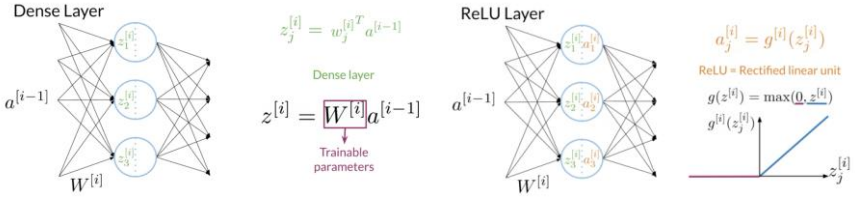
$$z^{[i]} = W^{[i]} a^{[i-1]}$$

$$a^{[i]} = g^{[i]}(z^{[i]})$$

-

Neural Networks for sentiment analysis

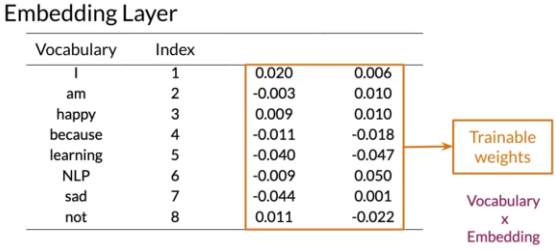


-
- Uses Dense layer and ReLU Layer
  - 

- x:
  - 

Word	Number
a	1
able	2
about	3
...	...
hand	615
...	...
happy	621
...	...
zebra	1000

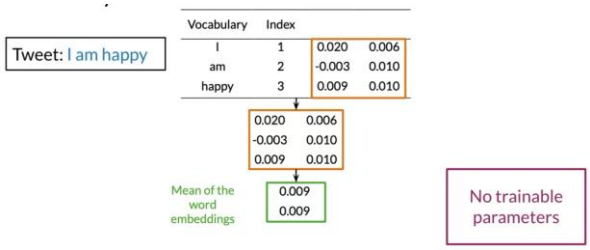
Embedding Layer



-
- **Purpose**: Maps words to a high-dimensional space where words with similar meaning have similar representations.
- **Vocabulary**: Set of unique words from your text data.

- **Functionality**: Takes an index for each word and maps it to a fixed-size dense vector.
- **Dimensionality**: The size of the embedding (e.g., 2) is a hyperparameter.
- **Trainable Parameters**: The embedding weights are learned during training to optimize performance on the NLP task.
- **Benefits**: Allows the model to learn and improve word representations for the specific NLP task.

Mean Layer



-
- **Purpose**: Reduces the dimensionality of the output from the embedding layer.
- **Functionality**: Computes the mean of the features across the word embeddings.
- **Parameters**: No trainable parameters; it simply averages the embeddings.
- **Output**: Returns a fixed-size vector regardless of the sequence length, preserving the number of features as the embedding size.
- **Use Case**: Useful for handling variable-length text inputs with a fixed-size layer downstream.

Programming lab: Sentiment with Deep neural Networks

Import library & data:

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from utils import load_tweets, process_tweet
%matplotlib inline

all_positive_tweets, all_negative_tweets = load_tweets()
```

Process the tweets:

- tokenizing the sentence (splitting to words)
- removing stock market tickers like $GE
- removing old style retweet text "RT"
- removing hyperlinks
- removing hashtags
- lowercasing
- removing stopwords and punctuation
- stemming

```
all_positive_tweets_processed = [process_tweet(tweet) for tweet in all_positive_tweets]
all_negative_tweets_processed = [process_tweet(tweet) for tweet in all_negative_tweets]
```

Split the dataset:

```
val_pos = all_positive_tweets_processed[4000:]
train_pos = all_positive_tweets_processed[:4000]
val_neg = all_negative_tweets_processed[4000:]
train_neg = all_negative_tweets_processed[:4000]
train_x = train_pos + train_neg
val_x  = val_pos + val_neg
train_y = [[1] for _ in train_pos] + [[0] for _ in train_neg]
val_y  = [[1] for _ in val_pos] + [[0] for _ in val_neg]
```

Build the vocabulary:

```
def build_vocabulary(corpus):
    vocab = {'': 0, '[UNK]': 1}
    index = 2
    # For each tweet in the training set
    for tweet in corpus:
        # For each word in the tweet
        for word in tweet:
            # If the word is not in vocabulary yet, add it to vocabulary
            if word not in vocab:
                vocab[word] = index
                index += 1
    return vocab
vocab = build_vocabulary(train_x)
num_words = len(vocab)
```

Find the length of the longest tweet

```
def max_length(training_x, validation_x):
    max_len = max(len(tweet_tokens) for tweet_tokens in training_x + validation_x)
    return max_len
```

Pad the sequence → max_len

```
def padded_sequence(tweet, vocab_dict, max_len, unk_token='[UNK]'):
    unk_ID = vocab_dict[unk_token]
    encoded_tweet = [vocab_dict.get(word, unk_ID) for word in tweet]
    padded_tensor = encoded_tweet + [0] * (max_len - len(encoded_tweet))
    return padded_tensor

train_x_padded = [padded_sequence(x, vocab, max_len) for x in train_x]
val_x_padded = [padded_sequence(x, vocab, max_len) for x in val_x]
```

ReLU

```
def relu(x):
    activation = np.maximum(0,x)
    return activation
```

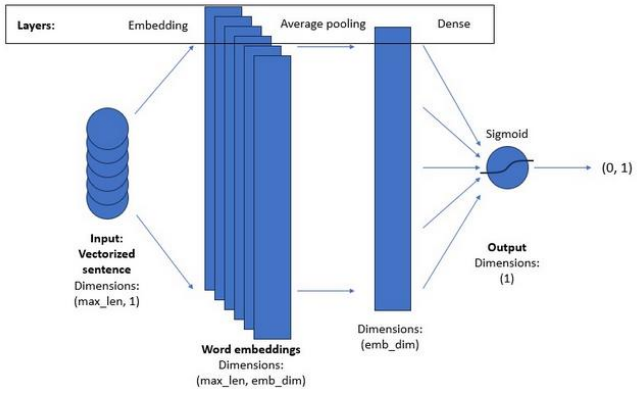
Sigmoid

```
def sigmoid(x):
    activation = 1/(1+np.exp(-x))
    return activation
```

Dense

```
class Dense():
    def __init__(self, n_units, input_shape, activation, stdev=0.1, random_seed=42):
        self.n_units = n_units
        self.random_generator = np.random.default_rng(seed=random_seed)
        self.activation = activation
        w = self.random_generator.normal(scale=stdev, size = (input_shape[-1], self.n_units))
        self.weights = w
    def __call__(self, x):
        return self.forward(x)
    def forward(self, x):
        dense = np.dot(x,self.weights)
        dense = self.activation(dense)
        return dense
```

Model:

- tf.keras.layers.Embedding
    - Turns positive integers (word indices) into vectors of fixed size. You can imagine it as creating one-hot vectors out of indices and then running them through a fully-connected (dense) layer.
- tf.keras.layers.GlobalAveragePooling1D
- tf.keras.layers.Dense
    - Regular fully connected layer
- 

Model Implementation:

```
def create_model(num_words, embedding_dim, max_len):
    tf.random.set_seed(123)
    model = tf.keras.Sequential([
        tf.keras.layers.Embedding(num_words, embedding_dim, input_length=max_len),
        tf.keras.layers.GlobalAveragePooling1D(),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])
    model.compile(loss='binary_crossentropy',
                  optimizer='adam',
                  metrics=['accuracy'])
    return model

model = create_model(num_words=num_words, embedding_dim=16, max_len=max_len)
```

Training model:

```
train_x_prepared = np.array(train_x_padded)
val_x_prepared = np.array(val_x_padded)
train_y_prepared = np.array(train_y)
val_y_prepared = np.array(val_y)
history = model.fit(train_x_prepared, train_y_prepared, epochs=20,
validation_data=(val_x_prepared, val_y_prepared))
```

Evaluating model:

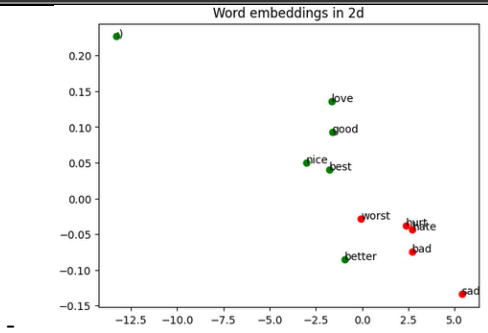
```
def plot_metrics(history, metric):
    plt.plot(history.history[metric])
    plt.plot(history.history[f'val_{metric}'])
    plt.xlabel("Epochs")
    plt.ylabel(metric.title())
    plt.legend([metric, f'val_{metric}'])
    plt.show()

plot_metrics(history, "accuracy")
plot_metrics(history, "loss")
```

Predict the data:

```
example_for_prediction = np.append(val_x_prepared[0:10], val_x_prepared[-10:], axis=0)
model.predict(example_for_prediction)

def get_prediction_from_tweet(tweet, model, vocab, max_len):
    tweet = process_tweet(tweet)
    tweet = padded_sequence(tweet, vocab, max_len)
    tweet = np.array([tweet])
    prediction = model.predict(tweet, verbose=False)
    return prediction[0][0]

unseen_tweet = '@DLAI @NLP_team_dlai OMG!!! what a daaay, wow, wow. This AsSiGnMeNt was gr8.'
prediction_unseen = get_prediction_from_tweet(unseen_tweet, model, vocab, max_len)
```

Word Embeddings

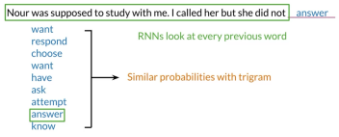
```
embeddings_layer = model.layers[0]
embeddings = embeddings_layer.get_weights()[0]
pca = PCA(n_components=2)
embeddings_2D = pca.fit_transform(embeddings)
neg_words = ['bad', 'hurt', 'sad', 'hate', 'worst']
pos_words = ['best', 'good', 'nice', 'love', 'better', ':)']
neg_n = [vocab[w] for w in neg_words]
pos_n = [vocab[w] for w in pos_words]
plt.figure()
plt.scatter(embeddings_2D[neg_n][:,0], embeddings_2D[neg_n][:,1], color = 'r')
for i, txt in enumerate(neg_words):
    plt.annotate(txt, (embeddings_2D[neg_n][i,0], embeddings_2D[neg_n][i,1]))
plt.scatter(embeddings_2D[pos_n][:,0], embeddings_2D[pos_n][:,1], color = 'g')
for i, txt in enumerate(pos_words):
    plt.annotate(txt,(embeddings_2D[pos_n][i,0], embeddings_2D[pos_n][i,1]))
plt.title('Word embeddings in 2d')
plt.show()
```



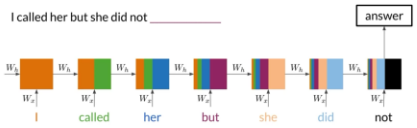
Traditional Language model:

- N-Grams
  - Large N-grams needed to capture dependencies between distant words
  - Need a lot of space and RAM
  - Example: Bigrams:
    - $P(w_1,w_2,w_3) = P(w_1) * P(w_2|w_1) * P(w_3|w_2)$

Recurrent Neural Networks vs N-Grams:

- Example:
  - 

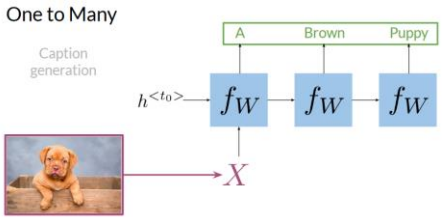
RNNs Basic Structure:

- 

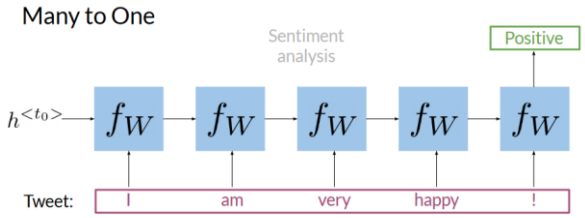
- Learnable parameters:
  - 

Applications of RNNs

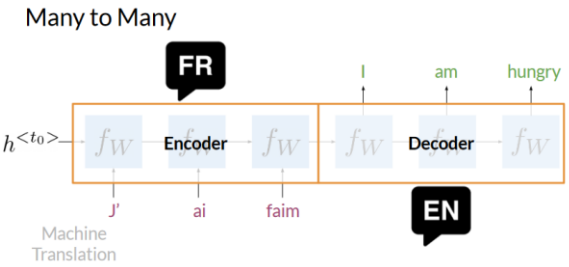
- One to One:
  - A simple model that takes a single input to produce a single output.
  - Example: Predicting the position of a soccer team on the leaderboard based on a list of scores from La Liga Santander.
  - For such tasks, the advantage of RNNs over traditional neural networks is minimal.
  - One to One
    

- One to Many:
  - This architecture takes a single input to generate multiple outputs.
  - Example Use: Image captioning where an RNN takes an image and outputs a series of words to describe it.
  - One to Many
    

- Many to One:
  - Multiple inputs leading to a single output.
  - Example: Sentiment analysis where an RNN analyzes a sequence of words (like a tweet) to determine the overall sentiment.



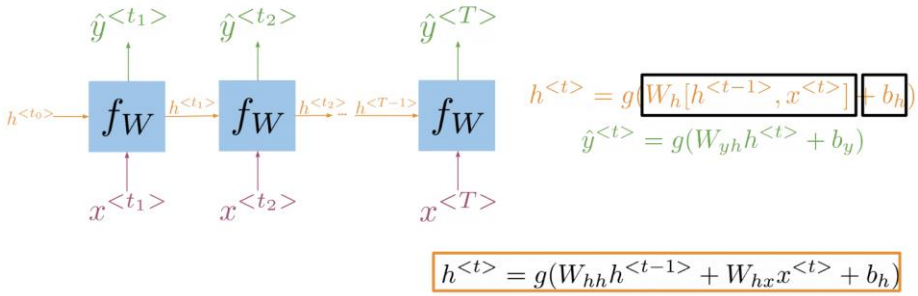
  -
- Many to Many:
  - Involves multiple inputs and multiple outputs.
  - Example: Machine translation where an RNN takes a series of words in one language and translates them into another.
  - Specific Architecture: The encoder-decoder model is commonly used for this task. The encoder processes the input sequence into a single representation capturing the sentence's meaning, which the decoder then translates into another language



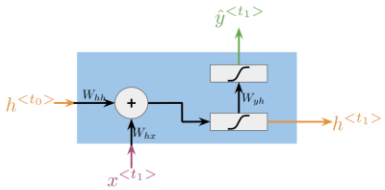
  -

Math in Simple RNNs

- A Vanilla RNN:



  -
  - Look into the first cell:

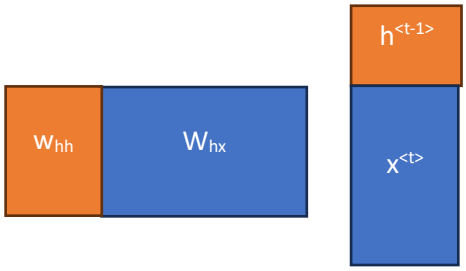


    -
  - Shapes:
    - If $h^{<t>}$ → 4x1, $x^{<t>}$ = 10x1
      - W_h will be 4x14

$W_h = [W_{hh} \mid W_{hx}]$

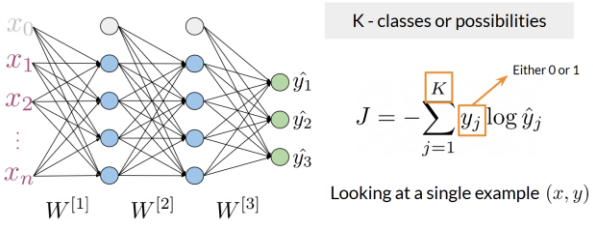
Formula 1: $h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$

Formula 2: $h^{<t>} = g(W_{hh}h^{<t-1>} + W_{hx}x^{<t>} + b_h)$



Formula 1 ⇔ Formula 2:

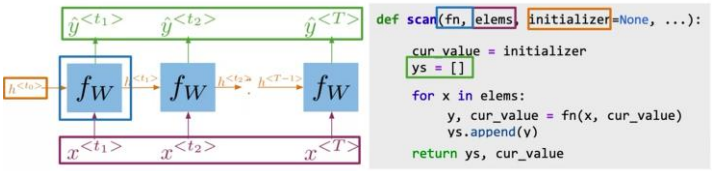
Cross function for RNNs

- Cross Entropy Loss
  - For K – classes or possibilities:
    - 

  - For Vanilla RNN:
    - Average with respect to time
      - there is just one value in every vector $y^{<t>}$ different from 0.
    - $J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{K} y_j^{<t>} \log \hat{y}_j^{<t>}$

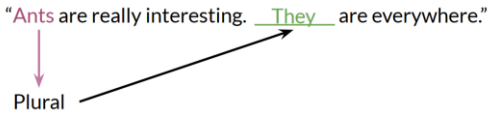
Scan function in Tensorflow:

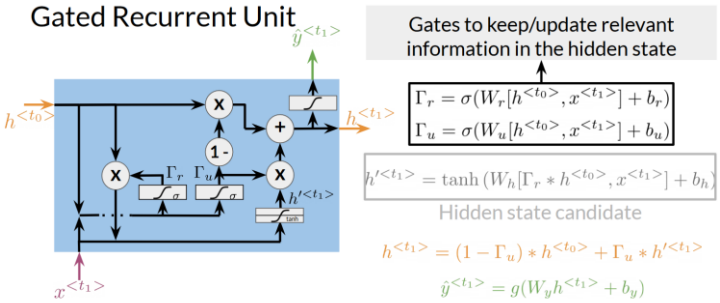
tf.scan() function



- cur_value corresponds to the hidden state in an RNN
- Framwork like Tensorflow need this type of abstraction for parallel computations and GPU usage.

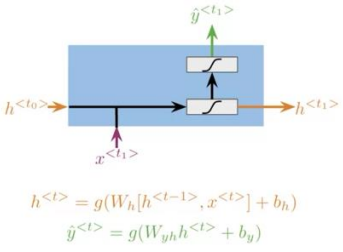
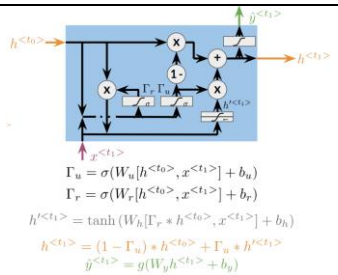
Gated Recurrent Units

- Allows relevant information to be kept in the hidden state over long sequences.
- Example:
  - 

  - Relevance and update gates to remember important prior information.
- Architecture:
  - 

  - Γ_u: Update Gate
    - Calculate how much of past information to keep.
  - Γ_r: Relevance (Reset) Gate:
    - Calculate how much of past information to disregard.
    - How much of the current input should be used to write a new memory.
  - h': Hidden state candidate:
    - Stores all the candidates for information that could override the one contained in the previous hidden state.
    - Selectively forgot past information.
  - Final Hidden state update h:
    - a blend of the old state and the candidate state, as dictated by the update gate.

Vanilla RNN vs GRUs

Vanilla RNN	GRUs
	
Disadvantages: - Updating the hidden state at every time step, for long sequence, the information tends to vanish.	Advantages: - Allow the network to learn what type of information to keep and when to override it.

Programming lab: Vanilla RNN and GRUs

Forward method for Vanillar RNN cell:

$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

```
def forward_V_RNN(inputs, weights):
    x, h_t = inputs
    wh, bh = weights
    h_t = np.dot(wh, np.concatenate([h_t, x])) + bh
    h_t = sigmoid(h_t)
    y = h_t
    return y, h_t
```

Forward method for GRUs:

$$\Gamma_r = \sigma(W_r[h^{<t-1>}, x^{<t>}] + b_r)$$

$$\Gamma_u = \sigma(W_u[h^{<t-1>}, x^{<t>}] + b_u)$$

$$c^{<t>} = \tanh(W_h[\Gamma_r * h^{<t-1>}, x^{<t>}] + b_h)$$

$$h^{<t>} = \Gamma_u * c^{<t>} + (1 - \Gamma_u) * h^{<t-1>}$$

```
def forward_GRU(inputs, weights): # Forward propagation for a single GRU cell
    x, h_t = inputs
    # weights.
    wu, wr, wc, bu, br, bc = weights
    # Update gate
    u = np.dot(wu, np.concatenate([h_t, x])) + bu
    u = sigmoid(u)
    # Relevance gate
    r = np.dot(wr, np.concatenate([h_t, x])) + br
    r = sigmoid(r)
    # Candidate hidden state
    c = np.dot(wc, np.concatenate([r * h_t, x])) + bc
    c = np.tanh(c)
    # New Hidden state h_t
    h_t = u * c + (1 - u) * h_t
    # We avoid implementation of y for clarity
    y = h_t
    return y, h_t

forward_GRU([X[1], h_0], weights_GRU)[0]
```

Scan function:

```
def scan(fn, elems, weights, h_0): # Forward propagation for RNNs
    h_t = h_0
    ys = []
    for x in elems:
        y, h_t = fn([x, h_t], weights)
        ys.append(y)
    return ys, h_t
```

Go through all the elements from a sequence

```
# vanilla RNNs
ys, h_T = scan(forward_V_RNN, X, weights_vanilla, h_0)
# GRUs
ys, h_T = scan(forward_GRU, X, weights_GRU, h_0)
```

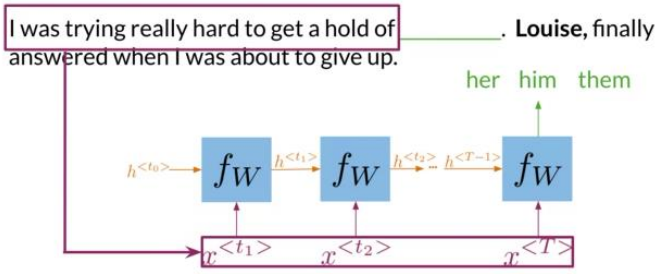
Create a GRU model in tensorflow:

```
model_GRU = tf.keras.Sequential([
    tf.keras.layers.GRU(256, return_sequences=True, name='GRU_1_returns_seq'),
    tf.keras.layers.GRU(128, return_sequences=True, name='GRU_2_returns_seq'),
    tf.keras.layers.GRU(64, name='GRU_3_returns_last_only'),
    tf.keras.layers.Dense(10)
])
batch_size = 60
sequence_length = 50
word_vector_length = 40
input_data = tf.random.normal([batch_size, sequence_length, word_vector_length])
prediction = model_GRU(input_data)
model_GRU.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 GRU_1_returns_seq (GRU)     (60, 50, 256)             228864

 GRU_2_returns_seq (GRU)     (60, 50, 128)             148224

 GRU_3_returns_last_only (G  (60, 64)                  37248
 RU)

 dense (Dense)               (60, 10)                  650

=================================================================
Total params: 414986 (1.58 MB)
Trainable params: 414986 (1.58 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- the model's first GRU layer takes the original 40-dimensional word vectors and transforms them into its own 256-dimensional representations.

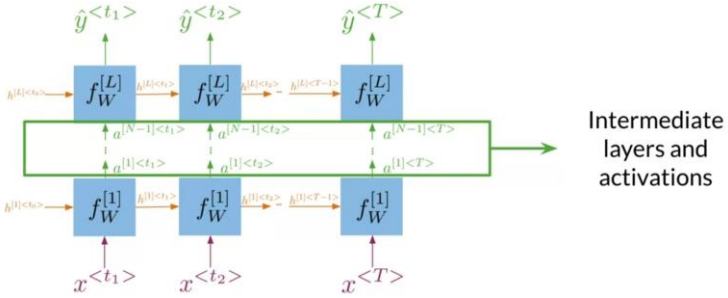
Limitation of simple RNNs:



-

Deep RNNs:

- Just RNNs stack together



-
- Information flow:

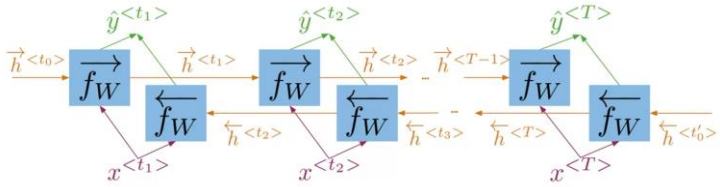
$$h^{[l]<t>} = f^{[l]}(W_h^{[l]}[h^{[l]<t-1>}, a^{[l-1]<t>}] + b_h^{[l]})$$

$$a^{[l]<t>} = f^{[l]}(W_a^{[l]} h^{[l]<t>} + b_a^{[l]})$$

1. Get hidden states for current layer
2. Pass the activations to the next layer

  -

Bi-directional RNNs



-
  - information flows from the past and from the future independently
  - Prediction:

$$\hat{y}^{<t>} = g(W_y[\overrightarrow{h}^{<t>}, \overleftarrow{h}^{<t>}] + b_y)$$

Perplexity:

$$P(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_1, \ldots, w_{i-1})}}$$

-
- Taking the logarithm:

$$= -\frac{1}{N} \sum_{i=1}^{N} \log P(w_i|w_1, \ldots, w_{i-1})$$

  -

Assignment: Deep N-grams

Load library

```
import os
import traceback
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import shutil
import numpy as np
import random as  rnd
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.layers import Input
from termcolor import colored
rnd.seed(32)
```

Load the Data:

```
dirname = 'data/'
filename = 'shakespeare_data.txt'
lines = []
counter = 0
with open(os.path.join(dirname, filename)) as files:
    for line in files:
        pure_line = line.strip()#.lower()
        if pure_line:
            lines.append(pure_line)
```

Create the vocabulary:

```
text = "\n".join(lines)
vocab = sorted(set(text))
vocab.insert(0,"[UNK]")
vocab.insert(1,"")
```

Convert a line to Tensor:

```
line = "Hello world!"
chars = tf.strings.unicode_split(line, input_encoding='UTF-8')
```

chars:

- tf.Tensor([b'H' b'e' b'l' b'l' b'o' b' ' b'w'… b'!'], shape=(12,), dtype=string)

StringLookup:

```
ids = tf.keras.layers.StringLookup(vocabulary=list(vocab), mask_token=None)(chars)
```

Ids:

- tf.Tensor([34 59 66 66 69 4 77 69 72 66 58 5], shape=(12,), dtype=int64)

Line to Tensor:

Takes in a single line and transforms each character into its unicode integer

```
def line_to_tensor(line, vocab):
    chars = tf.strings.unicode_split(line, input_encoding='UTF-8')
    ids = tf.keras.layers.StringLookup(vocabulary=list(vocab), mask_token=None)(chars)
    return ids
```

Ids to Text:

Converting back from ids to text

```
def text_from_ids(ids, vocab):
    chars_from_ids = tf.keras.layers.StringLookup(vocabulary=vocab, invert=True, mask_token=None)
    return tf.strings.reduce_join(chars_from_ids(ids), axis=-1)
```

Predict the next character: Create the input and the output for the model:

```
def split_input_target(sequence):
    input_text = sequence[:-1]
    target_text = sequence[1:]
    return input_text, target_text
```

split_input_target(list("Tensorflow")):

- (['T', 'e', 'n', 's', 'o', 'r', 'f', 'l', 'o'], ['e', 'n', 's', 'o', 'r', 'f', 'l', 'o', 'w'])

Create batch dataset

```
def create_batch_dataset(lines, vocab, seq_length=100, batch_size=64):
    BUFFER_SIZE = 10000
    single_line_data = "\n".join(lines)
    all_ids = line_to_tensor(single_line_data, vocab)
    ids_dataset = tf.data.Dataset.from_tensor_slices(all_ids)
    data_generator = ids_dataset.batch(seq_length + 1, drop_remainder=True)
    dataset_xy = data_generator.map(split_input_target)
    dataset = (
        dataset_xy
        .shuffle(BUFFER_SIZE)
        .batch(batch_size, drop_remainder=True)
        .prefetch(tf.data.experimental.AUTOTUNE)
        )
    return dataset
```

GRU Language Model (GRULM)

```
class GRULM(tf.keras.Model):
    def __init__(self, vocab_size=256, embedding_dim=256, rnn_units=128):
        super().__init__(self)
        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru = tf.keras.layers.GRU(
            units=rnn_units,
            return_sequences=True,
            return_state=True,
        )
        self.dense = tf.keras.layers.Dense(vocab_size, activation=tf.nn.log_softmax)
    def call(self, inputs, states=None, return_state=False, training=False):
        x = inputs
        x = self.embedding(x, training=training)
        if states is None:
            states = self.gru.get_initial_state(x)
        x, states = self.gru(x, initial_state=states, training=training)
        x = self.dense(x, training=training)
        if return_state:
            return x, states
        else:
            return x
```

Using the model to predict the next character using the untrained model:

```
vocab_size = 82
embedding_dim = 256
rnn_units = 512
model = GRULM(
    vocab_size=vocab_size,
    embedding_dim=embedding_dim,
    rnn_units = rnn_units)
```

```
for input_example_batch, target_example_batch in dataset.take(1):
    example_batch_predictions = model(tf.constant([input_example_batch[0].numpy()]))
```

example_batch_prediction.shape:

- (1,100,82): the first sequence generated by the batch generator, predict 100 characters, 82 values for each predicted character.

Choose the next character by getting the index of the element with the highest likelihood:

```
last_character = tf.math.argmax(example_batch_predictions[0][99])
sampled_indices = tf.math.argmax(example_batch_predictions[0], axis=1)

print("Input:\n", text_from_ids(input_example_batch[0], vocab))
print()
print("Next Char Predictions:\n", text_from_ids(sampled_indices, vocab))
```

Compile Model:

```
def compile_model(model):
    loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
    opt = tf.keras.optimizers.Adam(learning_rate=0.00125)
    model.compile(optimizer=opt, loss=loss)
    return model
```

Training:

```
EPOCHS = 10
model = compile_model(model)
history = model.fit(dataset, epochs=EPOCHS)
```

Log_perplexity:

```
def log_perplexity(preds, target):
    PADDING_ID = 1
    log_p = np.sum(preds * tf.one_hot(target, preds.shape[-1]),axis= -1)
    non_pad = 1.0 - np.equal(target, PADDING_ID)
    log_p = log_p * non_pad
    log_ppx = np.sum(log_p, -1) / np.sum(non_pad, -1)
    log_ppx = np.mean(log_ppx)
    return -log_ppx
```

Loading the pretrained model:

```
vocab_size = len(vocab)
embedding_dim = 256
rnn_units = 512
model = GRULM(
    vocab_size=vocab_size,
    embedding_dim=embedding_dim,
    rnn_units = rnn_units)
model.build(input_shape=(100, vocab_size))
model.load_weights('./model/')
```

Calculate the perplexity:

```
eval_text = "\n".join(eval_lines)
eval_ids = line_to_tensor([eval_text], vocab)
input_ids, target_ids = split_input_target(tf.squeeze(eval_ids, axis=0))
preds, status = model(tf.expand_dims(input_ids, 0), training=False, states=None,
return_state=True)
log_ppx = log_perplexity(preds, tf.expand_dims(target_ids, 0))
print(f'The log perplexity and perplexity of your model are {log_ppx} and
{np.exp(log_ppx)} respectively')
```

Output:

- The log perplexity and perplexity of your model are 1.2239635591264044 and 3.400639693684683 respectively

This model in its default form is deterministic and can result in repetitive and monotonous outputs. To make the Language model more dynamic and versatile:

- Introduce an element of randomness into its predictions.
- Employ random sampling, regulated by temperature.

```
def temperature_random_sampling(log_probs, temperature=1.0):
    u = tf.random.uniform(minval=1e-6, maxval=1.0 - 1e-6, shape=log_probs.shape)
    g = -tf.math.log(-tf.math.log(u))
    return tf.math.argmax(log_probs + g * temperature, axis=-1)
```

GenerativeModel:

```
class GenerativeModel(tf.keras.Model):
    def __init__(self, model, vocab, temperature=1.0):
        super().__init__()
        self.temperature = temperature
        self.model = model
        self.vocab = vocab
    @tf.function
    def generate_one_step(self, inputs, states=None):
        input_ids = line_to_tensor(inputs, self.vocab)
        predicted_logits, states = self.model(inputs=input_ids, states=states, return_state=True)
        predicted_logits = predicted_logits[0, -1, :]
        predicted_ids = temperature_random_sampling(predicted_logits, self.temperature)
        predicted_chars = text_from_ids([predicted_ids], self.vocab)
        return tf.expand_dims(predicted_chars, 0), states
    def generate_n_chars(self, num_chars, prefix):
        states = None
        next_char = tf.constant([prefix])
        result = [next_char]
        for n in range(num_chars):
            next_char, states = self.generate_one_step(next_char, states=states)
            result.append(next_char)
        return tf.strings.join(result)[0].numpy().decode('utf-8')
```

Generate a longer text:

```
tf.random.set_seed(np.random.randint(1, 1000))
gen = GenerativeModel(model, vocab, temperature=0.8)
import time
start = time.time()
print(gen.generate_n_chars(1000, "ROMEO "), '\n\n' + '_'*80)
print('\nRun time:', time.time() - start)
```

RNNs and Vanishing Gradients

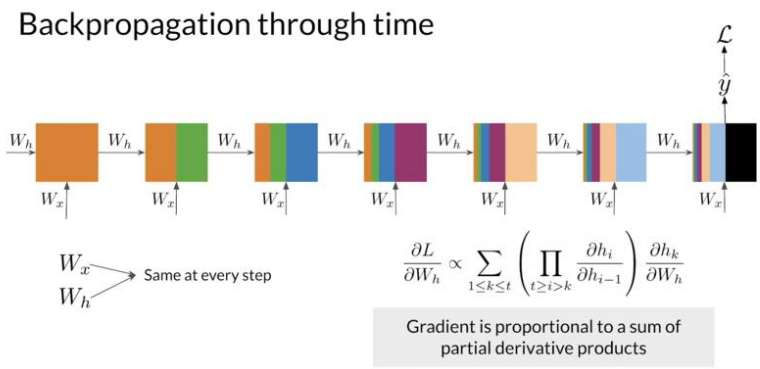
RNNs: Advantages:

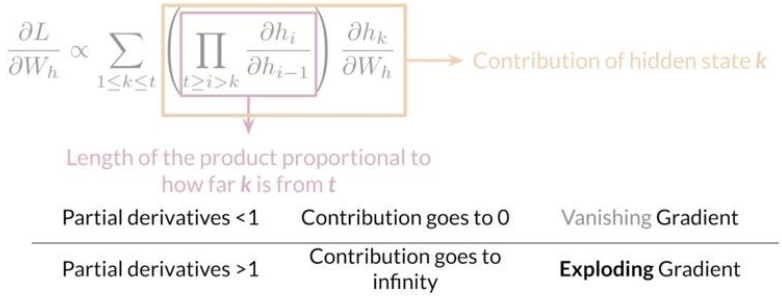
- Captures dependencies within a short range
- Takes up less RAM than other n-gram models

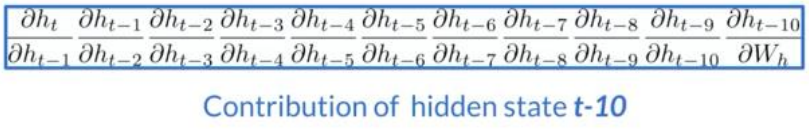
RNNs: Disadvantages:

- Struggles to capture long term dependencies
- Prone to vanishing or exploding gradients

RNNs: Back propagation through time:

- 

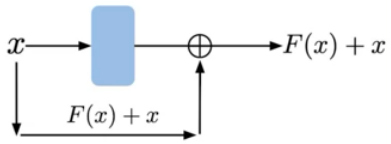
- 

- 

Solving for vanishing gradients problem:

- Identity RNN:
  - Identity RNN with ReLU activation $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ -1 → 0
  -
  - Initalizing the weights to the identity matrix, using ReLU activation.
  - Interpretation:
    - Copy the previous hidden states and information from the current inputs and replace any negative values with zero.
  - Effect:
- Encourage the network to stay close to the values and the identity matrix.

Solving for exploding gradients problem:

- Gradient clipping
- Skip connections:
  - 

Identity Matrix Initialization

In an RNN, the hidden state at time `t` is typically computed as:

```
h_t = activation(W * h_(t-1) + U * x_t + b)
```

where:

- `h_t` is the hidden state at time `t`.
- `h_(t-1)` is the hidden state at time `t-1`.
- `x_t` is the input at time `t`.
- `W` and `U` are the weight matrices for the hidden state and input respectively.
- `b` is a bias vector.
- `activation` is the activation function applied element-wise.

When initializing `W` to the identity matrix (and assuming no bias for simplicity), the recurrent part of the computation simplifies to:

```
h_t = activation(h_(t-1) + U * x_t)
```

If `W` is the identity matrix and `b` is zero, `W * h_(t-1)` results in `h_(t-1)` because multiplying by the identity matrix leaves the vector unchanged. This means that, in the absence of new inputs, `h_t` would be the same as `h_(t-1)`.

ReLU Activation

ReLU, or Rectified Linear Unit, is an activation function defined as:

```
ReLU(x) = max(0, x)
```

The derivative of ReLU is as follows:

- `0` for `x < 0`
- `1` for `x > 0`
- Undefined at `x = 0`, but typically handled as either 0 or 1 in practice.

Because the derivative of ReLU is 1 for positive inputs, when you backpropagate through a ReLU neuron that has a positive activation, the gradient does not diminish. This is particularly useful in combating the vanishing gradient problem because it means that as long as the neuron activates, it will not diminish the gradient during backpropagation.

Mathematical Effect in Identity RNN

By combining the identity matrix with ReLU, the RNN has the following properties:

1. **Preservation of Gradient Magnitude**: During backpropagation, the gradient with respect to `h_(t-1)` is not scaled down by `W` because `W` is an identity matrix. This helps prevent the decay of gradients as they are propagated back through time.
2. **Non-Negativity of Activations**: ReLU ensures that all negative values are set to zero and positive values pass through unchanged. This means that negative data does not contribute to the gradient, which may help in maintaining the stability of the gradient's magnitude over time.
3. **Simplicity and Efficiency**: Since the identity matrix and ReLU are simple and involve minimal computation, this approach can be computationally efficient.

Lab: Vanishing gradients and exploding gradients in RNNs

- For Vanilla RNNS, the gradient with respect to W_h is proportional to:
  - $$\frac{\delta L}{\delta W_h} \propto \sum_{1 \le k \le t} \left(\prod_{t \ge i > k} \frac{\delta h_i}{\delta h_{i-1}} \right) \frac{\delta h_k}{\delta W_h}$$

Activations & Partial Derivative:

- Hidden state:
  - $$h_i = \sigma(W_{hh} h_{i-1} + W_{hx} x_i + b_h)$$

Backpropagation through time (BPTT):

- We want to compute the partial derivative of h_i with respect to h_{i-1}.

By chain rule:

- Compute the derivative of the outer function:
  - $\sigma'(z) = \sigma(z)(1-\sigma(z))$
- Compute the derivatie of the inner function:
  - The inner function is the affine transformation:
    - $W_{hh}h_{i-1} + W_{hx}x_i + b_h$ with respect to h_{i-1}
    - The derivative is simply W_{hh}
- We get:
  - $$\frac{\delta h_i}{\delta h_{i-1}} = \text{diag}(\sigma'(z)) W_{hh}$$
    - The diag function used to ensure that the element-wise derivative of the activation function is applied correctly during matrix multiplication.

Partial derivative:

- $$\frac{\delta h_i}{\delta h_{i-1}} = W_{hh}^T \text{diag}(\sigma'(W_{hh} h_{i-1} + W_{hx} x_i + b_h))$$

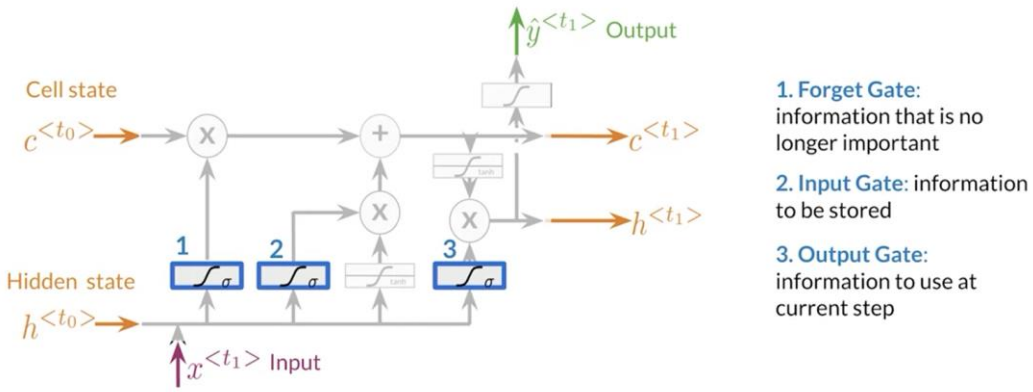
Gradients:

- $$\prod_{t \ge i > k} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{t \ge i > k} W_{hh}^T \text{diag}(\sigma'(W_{hh} h_{i-1} + W_{hx} x_i + b_h))$$
- Approaches 0 → vanishing gradient problems, vice versa.

Introduction to LSTMs

- Learns when to remember and when to forget.

LSTM Overview:

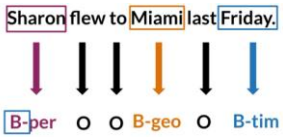


-
- Forget gate:
  - Decides what to keep.
  - $f = \sigma\left(W_f\left[h_t{-}_1; x_t\right] + b_f\right)$
- Input gate:
  - Decides what to add.
  - $i = \sigma\left(W_i\left[h_t{-}_1; x_t\right] + b_i\right)$
  - Candidate values vector: $\tilde{C}_t = \tanh(W_C \bullet [h_{t-1}, x_t] + b_C)$
  - Cell state update: $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
    - Forgetting the things decided by the forget gate & adding the new candidate values scaled by how much we decided to update each state value.
- Output gate:
  - Decides what the next hidden state will be.
  - $o = \sigma\left(W_o\left[h_{t-1}; x_t\right] + b_o\right)$
- Tanh layer:
  - Ensures the values in the network stay numerically stable by squeezing all values between -1 and 1.
- The output of LSTM unit:
  - $h_t = o_t \odot tanh(c_t)$

Application of LSTMs

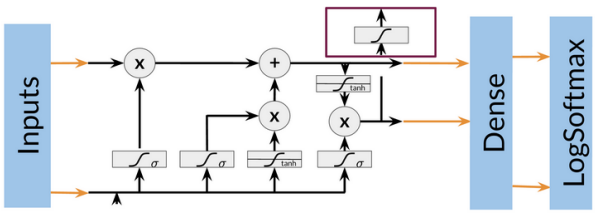
- Next-character prediction
- Image captioning
- Chatbots
- Music composition
- Speech recognition

Name Entity Recognition

- Locates and extracts predefined entities from text.
- Can be places, organizations, names, time and dates.
- Types of Entities:
    - Geographical (e.g. Thailand)
    - Organization (e.g. Google)
    - Geopolitical (e.g. Indian)
    - Time indicator (e.g. December)
    - Artifact (e.g. Egyptian statue)
    - Person (e.g. Barack Obama)
- Example of a labeled sentence:
    - 

- Application:
    - Search engine efficiency
    - Recommendation engines
    - Customer service
    - Automatic trading

Name Entity Recognition(NER) System:

- Training:
    - Create a tensor for each input and its corresponding number
    - Put them in a batch ==> 64, 128, 256, 512 ...
    - Feed it into an LSTM unit
    - Run the output through a dense layer
    - Predict using a log softmax over K classes
- Architecture:
    - 

Computing Accuracy:

- Pass test set through the model
- Get arg max across the prediction array
- Mask padded tokens making the array the same length
    - Padded tokens are not part of the data and are just used to help us keep the same sequence length for more efficient batch processing. We should not include their loss.
- Compare outputs against test labels

Name Entity Recognition Assignment:

Library:

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import numpy as np
import pandas as pd
import tensorflow as tf
tf.keras.utils.set_random_seed(33)
```

Data demo:

```
# display original kaggle data
data = pd.read_csv("data/ner_dataset.csv", encoding = "ISO-8859-1")
train_sents = open('data/small/train/sentences.txt', 'r').readline()
train_labels = open('data/small/train/labels.txt', 'r').readline()
del(data, train_sents, train_labels)
```

```
SENTENCE: Thousands of demonstrators have marched through London to protest
the war in Iraq and demand the withdrawal of British troops from that countr
y .

SENTENCE LABEL: O O O O O O B-geo O O O O O B-geo O O O O O B-gpe O O O O O

ORIGINAL DATA:
     Sentence #           Word  POS Tag
0  Sentence: 1      Thousands  NNS   O
1          NaN             of   IN   O
2          NaN  demonstrators  NNS   O
3          NaN           have  VBP   O
4          NaN        marched  VBN   O
```

Importing the data:

```
def load_data(file_path):
    with open(file_path,'r') as file:
        data = np.array([line.strip() for line in file.readlines()])
    return data

train_sentences = load_data('data/large/train/sentences.txt')
train_labels = load_data('data/large/train/labels.txt')
val_sentences = load_data('data/large/val/sentences.txt')
val_labels = load_data('data/large/val/labels.txt')
test_sentences = load_data('data/large/test/sentences.txt')
test_labels = load_data('data/large/test/labels.txt')
```

Encoding the sentences:

```
def get_sentence_vectorizer(sentences):
    tf.keras.utils.set_random_seed(33) ## Do not change this line.
    sentence_vectorizer = tf.keras.layers.TextVectorization(standardize=None)
    sentence_vectorizer.adapt(sentences)
    vocab = sentence_vectorizer.get_vocabulary()
    return sentence_vectorizer, vocab

sentence_vectorizer, vocab = get_sentence_vectorizer(train_sentences)
```

Encoding the labels

```
def get_tags(labels):
    tag_set = set() # Define an empty set
    for el in labels:
        for tag in el.split(" "):
            tag_set.add(tag)
    tag_list = list(tag_set)
    tag_list.sort()
    return tag_list
tags = get_tags(train_labels)
```

tags:

- ['B-art', 'B-eve', 'B-geo', 'B-gpe', 'B-nat', 'B-org', 'B-per', 'B-tim', ..., 'I-tim', 'O']

Mapping the tags to positive integers:

```
def make_tag_map(tags):
    tag_map = {}
    for i,tag in enumerate(tags):
        tag_map[tag] = i
    return tag_map
tag_map = make_tag_map(tags)
```

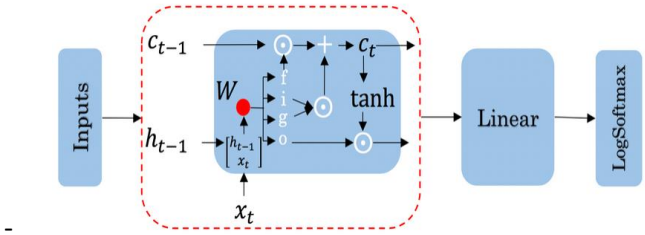
Label vectorizer: inputs a list of labels and a tag mapping and outputs their respective label ids via a tag map lookup.

```
def label_vectorizer(labels, tag_map):
    label_ids = []
    for element in labels:
        tokens = element.split(' ')
        element_ids = [tag_map[token] for token in tokens]
        label_ids.append(element_ids)
    label_ids = tf.keras.utils.pad_sequences(label_ids, padding='post', value=-1)
    return label_ids
```

Building the dataset:

```
def generate_dataset(sentences, labels, sentence_vectorizer, tag_map):
    sentences_ids = sentence_vectorizer(sentences)
    labels_ids = label_vectorizer(labels, tag_map = tag_map)
    dataset = tf.data.Dataset.from_tensor_slices((sentences_ids, labels_ids))
    return dataset

train_dataset = generate_dataset(train_sentences,train_labels, sentence_vectorizer, tag_map)
val_dataset = generate_dataset(val_sentences,val_labels,  sentence_vectorizer, tag_map)
test_dataset = generate_dataset(test_sentences, test_labels,  sentence_vectorizer, tag_map)
```

Model architecture:



-

```python
def NER(len_tags, vocab_size, embedding_dim = 50):
    model = tf.keras.Sequential(name = 'sequential')
    model.add(
        tf.keras.layers.Embedding(
            input_dim=vocab_size + 1,
            output_dim=embedding_dim,
            mask_zero=True
        )
    )
    model.add(
        tf.keras.layers.LSTM(
            units=embedding_dim,
            return_sequences=True
        )
    )
    model.add(
        tf.keras.layers.Dense(
            units=len_tags,
            activation=tf.nn.log_softmax
        )
    )
    return model
```

Masked loss and metrics:

```python
def masked_loss(y_true, y_pred):
    y_true = tf.reshape(y_true, [-1])
    y_pred = tf.reshape(y_pred, [-1, tf.shape(y_pred)[-1]])
    mask = tf.cast(tf.not_equal(y_true, -1), dtype=tf.float32)
    loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
    y_true_masked = tf.boolean_mask(y_true, mask)
    y_pred_masked = tf.boolean_mask(y_pred, mask)
    loss = loss_fn(y_true_masked, y_pred_masked)
    return loss
```

```python
def masked_accuracy(y_true, y_pred):
    y_true = tf.cast(y_true, tf.float32)
    mask = tf.not_equal(y_true, -1.0)
    mask = tf.cast(mask, tf.float32)
    y_pred_class = tf.argmax(y_pred, axis=-1, output_type=tf.int32)
    y_pred_class = tf.cast(y_pred_class, tf.float32)
    matches_true_pred = tf.equal(y_true, y_pred_class)
    matches_true_pred = tf.cast(matches_true_pred, tf.float32)
    matches_true_pred *= mask
    masked_acc = tf.reduce_sum(matches_true_pred) / tf.reduce_sum(mask)
    return masked_acc
```

Experiment:

```python
true_labels = [0,1,2,0]
predicted_logits = [[0.1,0.6,0.3] , [0.2,0.7,0.1], [0.1, 0.5,0.4], [0.4,0.4,0.2]]
print(masked_loss(true_labels, predicted_logits))
```

Model compile:

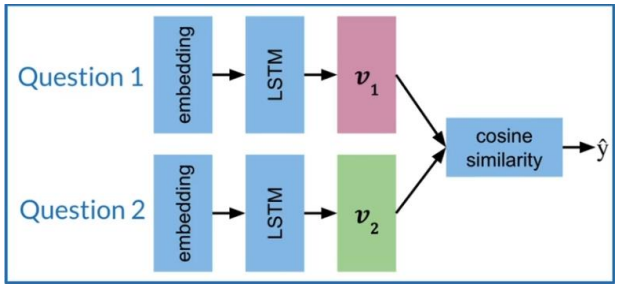
```
model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
              loss = masked_loss,
               metrics = [masked_accuracy])
tf.keras.utils.set_random_seed(33)
BATCH_SIZE = 64
model.fit(train_dataset.batch(BATCH_SIZE),
          validation_data = val_dataset.batch(BATCH_SIZE),
          shuffle=True,
          epochs = 2)
```

Make Prediction:

```
def predict(sentence, model, sentence_vectorizer, tag_map):
    index_to_tag = {index: tag for tag, index in tag_map.items()}
    sentence_vector = sentence_vectorizer([sentence])
    predictions = model(sentence_vector)
    predicted_indices = tf.argmax(predictions, axis=-1)
    predicted_indices = tf.squeeze(predicted_indices)
    pred = [index_to_tag[index] for index in predicted_indices.numpy() if index != -1]
    return pred
```

```
sentence = "Peter Parker , the White House director of trade and manufacturing policy of U.S , said in an interview on Sunday morning that the White House was working to prepare for the possibility of a second wave of the coronavirus in the fall , though he said it wouldn ’t necessarily come"
predictions = predict(sentence, model, sentence_vectorizer, tag_map)
for x,y in zip(sentence.split(' '), predictions):
    if y != 'O':
        print(x,y)
```

Siamese Networks:

- Identify similarity between things
- Example: same:
  - What is your age?
  - How old are you?
- Applications:
  - Handwritten checks
  - Identify question duplicates.
  - Check search engine queries.
- Architecture:
  - 

  - Contains two identical subnetworks (sister networks), merge to produce a similarity score between inputs.
  - The two subnetworks share identical parameters.
  - Process:
    - Transform each question into an embedding.
    - Feed embeddings into an LSTM layer to capture the meaning of each question.
    - Each LSTM outputs a vector, one for each question.
    - Compared the output vectors using cosine similarity → how similar the two questions are.

Siamese Network lab:

```
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras.models import Model, Sequential
from tensorflow import math
import numpy
numpy.random.seed(10)
```

```
vocab_size = 500
model_dimension = 128
LSTM = Sequential()
LSTM.add(layers.Embedding(input_dim=vocab_size, output_dim=model_dimension))
LSTM.add(layers.LSTM(units=model_dimension, return_sequences = True))
LSTM.add(layers.AveragePooling1D())
LSTM.add(layers.Lambda(lambda x: math.l2_normalize(x)))
input1 = layers.Input((None,))
input2 = layers.Input((None,))
conc = layers.Concatenate(axis=1)((LSTM(input1), LSTM(input2)))
Siamese = Model(inputs=(input1, input2), outputs=conc)
```

Loss Function for Siamese Network:

- Anchor: the reference input we compare against
- Positive: A different input that is supposed to be similar to the anchor
- Negative: An input that is supposed to be dissimilar to the anchor
- Example:
  - Anchor: How old are you?
  - Positive: What is your age?
  - Negative: Where are you from?
- Cosine Similarity:

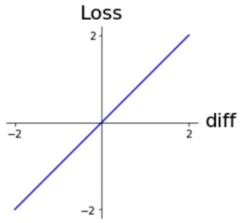
$$\cos(v_1, v_2) = \frac{v_1 \cdot v_2}{||v_1||\,||v_2||}$$

$$s(v_1, v_2)$$

$$s(A, P) \approx 1$$

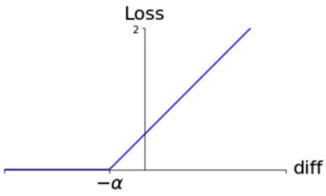
$$s(A, N) \approx -1$$

  - To begin building a loss function: $\text{diff} = s(A, N) - s(A, P)$



    - Problem: can decrease towards negative values, can even go to negative infinity.

Triplets



- Goal: Minimize the difference between similarity of (A,N) and (A,P) with constraint

With alpha margin

$$\mathcal{L} = \begin{cases} 0; & \text{if } diff + \alpha \leq 0 \\ diff + \alpha; & \text{if } diff + \alpha > 0 \end{cases}$$

  - Alpha margin is used to ensure difference between similarities is equal or close to the margin value.
  - Allows you to have some safety.
  - Controls how far cos(A,P) is from cos(A,N)

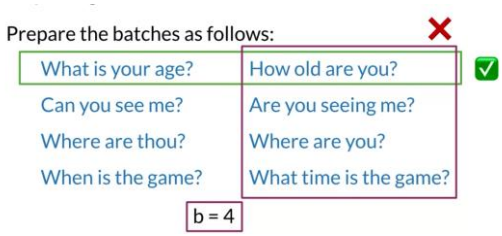
Triplet loss (simplified form):

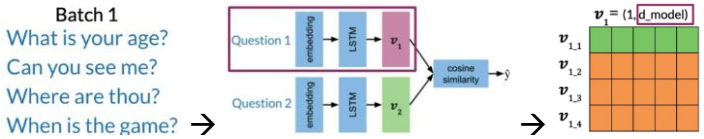
- $\mathcal{L}(A, P, N) = \max(diff + \alpha, 0)$
- $\max(-\cos(A, P) + \cos(A, N) + \alpha, 0)$

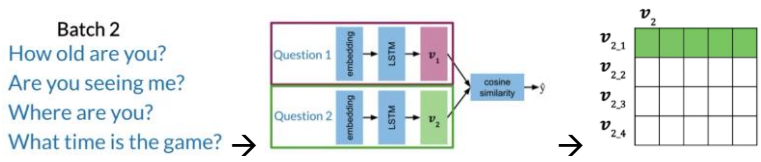
Triplet Selection:

- Objective: Choose challenging triplets to efficiently train the model.
- Selection Process:
  - Select a pair of duplicate questions (A and P).
  - Select a non-duplicate question (N).
- Avoid Random Selection: Leads to trivial cases with loss close to zero.
- Hard Triplets:
  - Select triplets that are difficult for the model to distinguish between P and N.
  - The similarity between A and N is close to but less than between A and P.
  - Focusing on hard triplets accelerates learning and improves model performance.

Computing the cost



- Given batch 1, pass the batch into the model, get the output.
  - 

- Do the same for batch 2:
  - 

- Example of computing the cost:
  - $s(v_1, v_2)$

v_2 \ v_1	_1	_2	_3	_4
_1	0.9	-0.8	0.3	-0.5
_2	-0.8	0.5	0.1	-0.2
_3	0.3	0.1	0.7	-0.8
_4	-0.5	-0.2	-0.8	1.0

  - The diagonal, the values are the similarities for all your positive example (Question duplicates)

$$\mathcal{L}(A, P, N) = \max(diff + \alpha, 0)$$
$$\text{diff} = s(A, N) - s(A, P)$$

$$\mathcal{J} = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$$

Modify the cost:

- Current lost:
  - $s(v_1, v_2)$

v_2 \ v_1	_1	_2	_3	_4
_1	0.9	-0.8	0.3	-0.5
_2	-0.8	0.5	0.1	-0.2
_3	0.3	0.1	0.7	-0.8
_4	-0.5	-0.2	-0.8	1.0

  - Mean negative:
    - Mean of off-diagonal values in each row
  - Cloest negative:
    - Off-diagonal value closest to (but less than) the value on diagonal in each row.
- Hard Negative Mining:
  - $\mathcal{L}_{\text{Original}} = \max(s(A, N) - s(A, P) + \alpha, 0)$ (diff)
  - $\mathcal{L}_1 = \max(mean\_neg - s(A, P) + \alpha, 0)$
  - $\mathcal{L}_2 = \max(closest\_neg - s(A, P) + \alpha, 0)$
    - L1: helps the model converge faster during training by reducing noise by training on just the average of several observations
      - How? Noise is defined to be a small value that comes from a distribution that is centered around zero. The average of several noise values is usually zero.
      - The individual noise from the observations cancelled out.
    - L2: Creates a larger penalty by focusing on the most challenging examples.

Full lost:

- $\mathcal{L}_{\text{Full}}(A, P, N) = \mathcal{L}_1 + \mathcal{L}_2$

$$\mathcal{J} = \sum_{i=1}^{m} \mathcal{L}_{\text{Full}}(A^{(i)}, P^{(i)}, N^{(i)})$$

Tensorflow implementation for Modified Triplet Loss:

```
v1 = np.array([1, 2, 3], dtype=float)
v2 = np.array([1, 2, 3.5], dtype=float)  # notice the 3rd element is offset by 0.5
print("-- Inputs --")
print("v1 :", v1)
print("v2 :", v2, "\n")
def cosine_similarity(v1, v2):
    numerator = tf.math.reduce_sum(v1*v2)
    denominator = tf.math.sqrt(tf.math.reduce_sum(v1*v1) * tf.math.reduce_sum(v2*v2))
    return numerator / denominator
```

```
v1_1 = np.array([1.0, 2.0, 3.0])
v1_2 = np.array([9.0, 8.0, 7.0])
v1_3 = np.array([-1.0, -4.0, -2.0])
v1_4 = np.array([1.0, -7.0, 2.0])
v1 = np.vstack([v1_1, v1_2, v1_3, v1_4])
v2_1 = v1_1 + np.random.normal(0, 2, 3)
v2_2 = v1_2 + np.random.normal(0, 2, 3)
v2_3 = v1_3 + np.random.normal(0, 2, 3)
v2_4 = v1_4 + np.random.normal(0, 2, 3)
v2 = np.vstack([v2_1, v2_2, v2_3, v2_4])
b = len(v1)
sim_1 = np.zeros([b, b])
for row in range(0, sim_1.shape[0]):
    for col in range(0, sim_1.shape[1]):
        sim_1[row, col] = cosine_similarity(v2[row], v1[col]).numpy()
```

```
def norm(x):
    return tf.math.l2_normalize(x, axis=1)
sim_2 = tf.linalg.matmul(norm(v2), norm(v1), transpose_b=True)
```

Hard Negative Mining:

```
sim_hardcoded = np.array(
    [
        [0.9, -0.8, 0.3, -0.5],
        [-0.4, 0.5, 0.1, -0.1],
        [0.3, 0.1, -0.4, -0.8],
        [-0.5, -0.2, -0.7, 0.5],
    ]
)
sim = sim_hardcoded
b = sim.shape[0]
sim_ap = np.diag(sim)
sim_an = sim - np.diag(sim_ap)
mean_neg = np.sum(sim_an, axis=1, keepdims=True) / (b - 1)
mask_1 = np.identity(b) == 1
mask_2 = sim_an > sim_ap.reshape(b, 1)
mask = mask_1 | mask_2
sim_an_masked = np.copy(sim_an)
sim_an_masked[mask] = -2
closest_neg = np.max(sim_an_masked, axis=1, keepdims=True)
```

Implementation in tensorflow:

```
sim_hardcoded = np.array(
    [
        [0.9, -0.8, 0.3, -0.5],
        [-0.4, 0.5, 0.1, -0.1],
        [0.3, 0.1, -0.4, -0.8],
        [-0.5, -0.2, -0.7, 0.5],
    ]
)
sim = sim_hardcoded
b = sim.shape[0]
sim_ap = tf.linalg.diag_part(sim)
sim_an = sim - tf.linalg.diag(sim_ap)
mean_neg = tf.math.reduce_sum(sim_an, axis=1) / (b - 1)
mask_1 = tf.eye(b) == 1
mask_2 = sim_an > tf.expand_dims(sim_ap, 1)
mask = tf.cast(mask_1 | mask_2, tf.float64)
sim_an_masked = sim_an - 2.0*mask
closest_neg = tf.math.reduce_max(sim_an_masked, axis=1)
```

The Loss Functions

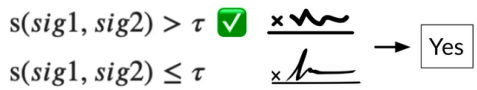
$$\mathcal{L}_1 = \max\left(mean\_neg - s(A, P) + \alpha, 0\right)$$

$$\mathcal{L}_2 = \max\left(closest\_neg - s(A, P) + \alpha, 0\right)$$

- $$\mathcal{L}_{\text{Full}} = \mathcal{L}_1 + \mathcal{L}_2$$

```
alpha = 0.25
l_1 = tf.maximum(mean_neg - sim_ap + alpha, 0)
l_2 = tf.maximum(closest_neg - sim_ap + alpha, 0)
l_full = l_1 + l_2
cost = tf.math.reduce_sum(l_full)
```

One shot learning



- Challenge:
  - Adding new classes necessitates costly model retraining.
- Solution:
  - Recognize a new class from a single example without retraining
- Method:
  - Utilizes a learned similarity function to compare and evaluate signatures based on similarity scores.
- Advantages:
  - Significantly more efficient for dynamic datasets, such as in banking scenarios, where new signatures are regularly introduced.

Training:

- The subnetworks have identical parameters so you only need to train one set of them, which are then shared by both.

Testing:

- Convert each input into an array of numbers.
- Feed arrays into the model.
- Compare v_1, v_2 using cosine similarity.
- Test against a threshold τ.

Evaluating a Siamese model:

Libraries:

```
1. import numpy as np
2. import tensorflow as tf
3. import tensorflow.math as math
4. import tensorflow.linalg as linalg
```

Question vector:

```
1. q1 = np.load('./data/q1.npy') #shape: (512,64)
2. q2 = np.load('./data/q2.npy') #shape: (512,64)
3. y_test = np.load('./data/y_test.npy') #shape: (512,)
4. v1 = np.load('./data/v1.npy') #shape: (512,128)
5. v2 = np.load('./data/v2.npy') #shape: (512,128)
```

Calculating the accuracy:

```
1. accuracy = 0
2. batch_size = 512
3. threshold = 0.7
4. for j in range(batch_size):
5.     d = math.reduce_sum(v1[j]*v2[j])
6.     res = d > threshold
7.     accuracy += tf.cast(y_test[j] == res, tf.int32)
8. accuracy = accuracy / batch_size
```

Question duplicates

Libraries % data:

```
 1. mport os
 2. os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
 3. import os
 4. import numpy as np
 5. import pandas as pd
 6. import random as rnd
 7. import tensorflow as tf
 8. rnd.seed(34)
 9. data = pd.read_csv("questions.csv")
10. N = len(data)
11. N_train = 300000
12. N_test = 10240
13. data_train = data[:N_train]
14. data_test = data[N_train:N_train + N_test]
15. del (data)  # remove to free memory
```

Data split:

```
1. td_index = data_train['is_duplicate'] == 1
2. td_index = [i for i, x in enumerate(td_index) if x]
3. Q1_train = np.array(data_train['question1'][td_index])
4. Q2_train = np.array(data_train['question2'][td_index])
5. Q1_test = np.array(data_test['question1'])
6. Q2_test = np.array(data_test['question2'])
7. y_test  = np.array(data_test['is_duplicate'])
```

- data_train['question1'][5]:
  - Astrology: I am a Capricorn Sun Cap moon and cap rising...what does that say about me?
- data_train['question2'][5]:
  - I'm a triple Capricorn (Sun, Moon and ascendant in Capricorn) What does this say about me?
- data_train['is_duplicate']
  - 1

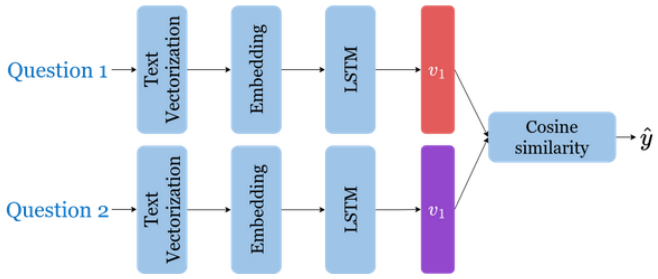
Splitting the data:

```
1. cut_off = int(len(Q1_train) * 0.8)
2. train_Q1, train_Q2 = Q1_train[:cut_off], Q2_train[:cut_off]
3. val_Q1, val_Q2 = Q1_train[cut_off:], Q2_train[cut_off:]
```

Learning question encoding:

```
1. tf.random.set_seed(0)
2. text_vectorization =
tf.keras.layers.TextVectorization(output_mode='int',split='whitespace',
standardize='strip_punctuation')
3. text_vectorization.adapt(np.concatenate((Q1_train,Q2_train)))
```

Defining the Siamese model:



-
- Triplet loss (Distance):
  - $\mathcal{L}(A, P, N) = \max\left(\|f(A) - f(P)\|^2 - \|f(A) - f(N)\|^2 + \alpha, 0\right)$

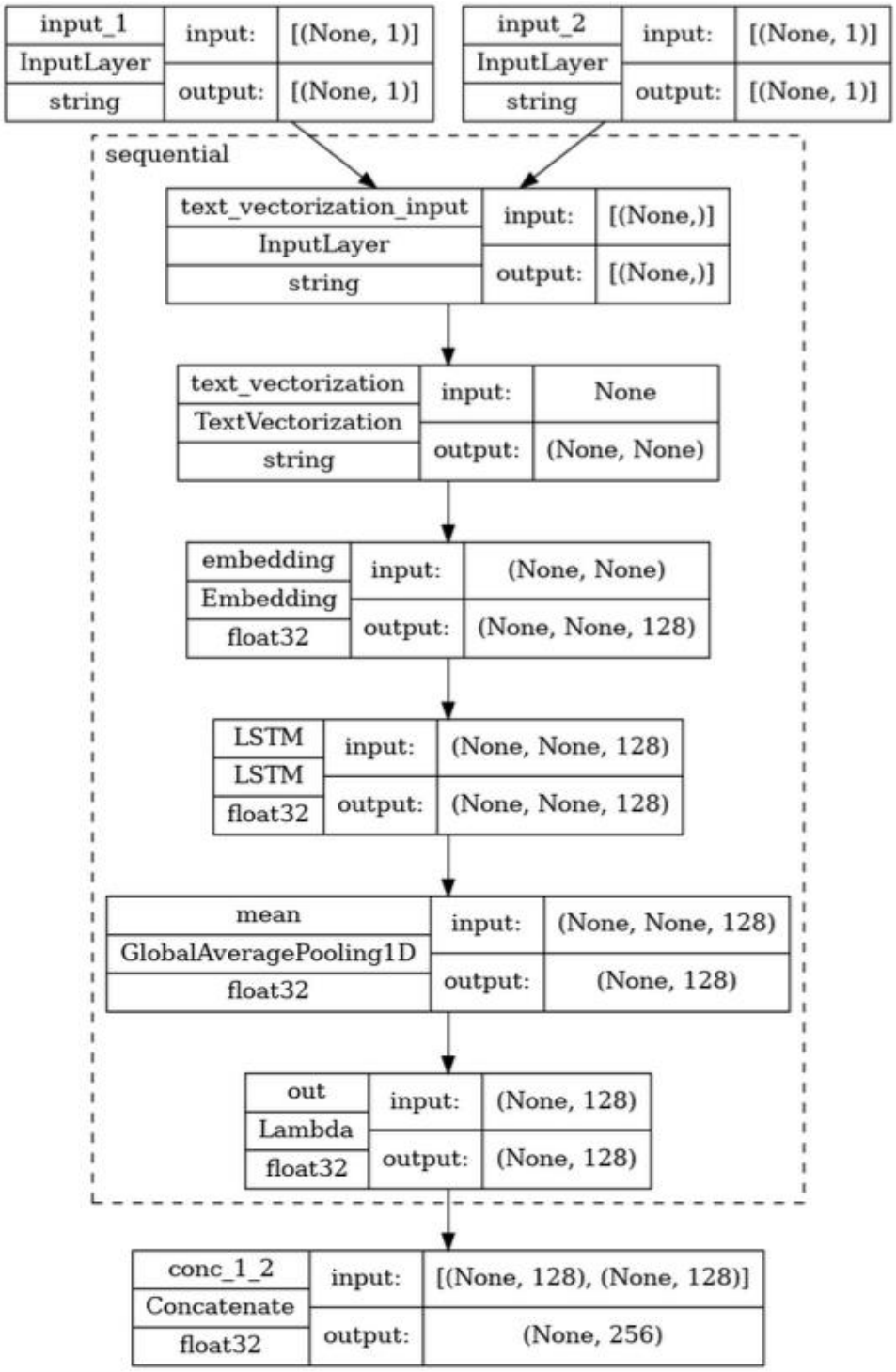
Siamese:

```python
def Siamese(text_vectorizer, vocab_size=36224, d_feature=128):
    branch = tf.keras.models.Sequential(name='sequential')
    branch.add(text_vectorizer)
    branch.add(tf.keras.layers.Embedding(input_dim=vocab_size, output_dim=d_feature, name='embedding'))
    branch.add(tf.keras.layers.LSTM(d_feature, return_sequences=True, name="LSTM"))
    branch.add(tf.keras.layers.GlobalAveragePooling1D(name="mean"))
    branch.add(tf.keras.layers.Lambda(lambda x: tf.math.l2_normalize(x, axis=1), name='out'))
    input1 = tf.keras.Input(shape=(1,), dtype=tf.string, name='input_1')
    input2 = tf.keras.Input(shape=(1,), dtype=tf.string, name='input_2')
    branch1 = branch(input1)
    branch2 = branch(input2)
    conc = tf.keras.layers.Concatenate(axis=-1, name='conc_1_2')([branch1, branch2])
    return tf.keras.models.Model(inputs=[input1, input2], outputs=conc, name="SiameseModel")
```

```python
model = Siamese(text_vectorization, vocab_size=text_vectorization.vocabulary_size())
model.build(input_shape=None)
```

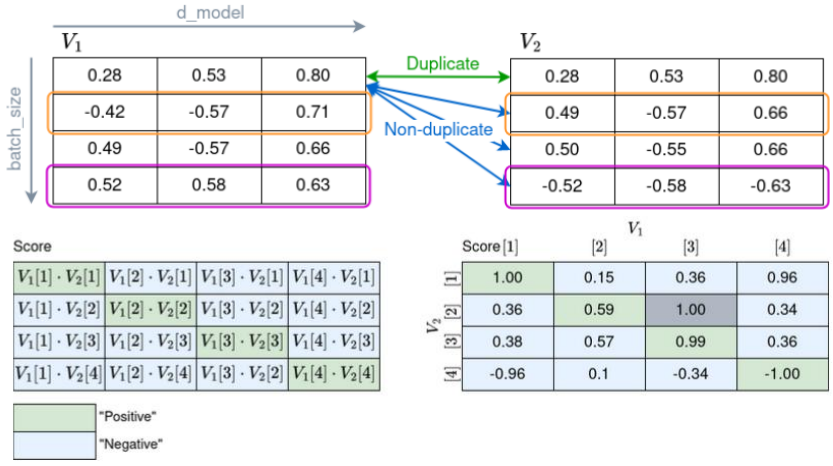
Plot model:

```
tf.keras.utils.plot_model(
    model,
    to_file="model.png",
    show_shapes=True,
    show_dtype=True,
    show_layer_names=True,
    rankdir="TB",
    expand_nested=True)
```



Triplet Loss with Hard Negative Mining:

$$\mathcal{L}oss_1(\mathcal{A}, \mathcal{P}, \mathcal{N}) = \max\left(-cos(A, P) + mean_{neg} + \alpha, 0\right)$$
$$\mathcal{L}oss_2(\mathcal{A}, \mathcal{P}, \mathcal{N}) = \max\left(-cos(A, P) + closest_{neg} + \alpha, 0\right)$$
$$\mathcal{L}oss(\mathcal{A}, \mathcal{P}, \mathcal{N}) = mean(Loss_1 + Loss_2)$$



Triplet Loss Function:

```python
def TripletLossFn(v1, v2,  margin=0.25):
    scores = tf.linalg.matmul(v1, v2, transpose_b = True)
    batch_size = tf.cast(tf.shape(v1)[0], scores.dtype)
    positive = tf.linalg.diag_part(scores)
    negative_zero_on_duplicate = scores - tf.linalg.diag(positive)
    mean_negative = tf.math.reduce_sum(negative_zero_on_duplicate, axis=1) / (batch_size - 1)
    mask_diagonal = tf.eye(batch_size) == 1
    mask_greater_than_positive =  negative_zero_on_duplicate > tf.expand_dims(positive, 1)
    mask_exclude_positives = tf.cast(mask_diagonal|mask_greater_than_positive, scores.dtype)
    negative_without_positive = negative_zero_on_duplicate - mask_exclude_positives * 2.0
    closest_negative = tf.math.reduce_max(negative_without_positive, axis=1)
    triplet_loss1 = tf.maximum(0.0,closest_negative - positive + margin)
    triplet_loss2 = tf.maximum(0.0,mean_negative - positive + margin)
    loss_full = triplet_loss1 + triplet_loss2
    triplet_loss = tf.math.reduce_sum(loss_full)
    return triplet_loss
```

```python
def TripletLoss(labels, out, margin=0.25):
    _, embedding_size = out.shape # get embedding size
    v1 = out[:,:int(embedding_size/2)] # Extract v1 from out
    v2 = out[:,int(embedding_size/2):] # Extract v2 from out
    return TripletLossFn(v1, v2, margin=margin)
```

Model:

```
def train_model(Siamese, TripletLoss, text_vectorizer, train_dataset, val_dataset, d_feature=128, lr=0.01, train_steps=5):
    vocab_size = len(text_vectorizer.get_vocabulary())
    model = Siamese(text_vectorizer,
                    vocab_size = vocab_size,
                    d_feature = d_feature)
    model.compile(loss=TripletLoss,
                  optimizer=tf.keras.optimizers.Adam(lr=lr)
                 )
    history = model.fit(train_dataset,
                        epochs=train_steps,
                        validation_data=val_dataset
                       )
    return model
```

Classify:

```
def classify(test_Q1, test_Q2, y_test, threshold, model, batch_size=64, verbose=True):
    test_gen = tf.data.Dataset.from_tensor_slices(((test_Q1, test_Q2), None)).batch(batch_size=batch_size)
    pred = model.predict(test_gen)
    v1, v2 = np.split(pred, 2, axis=1)
    d = tf.reduce_sum(v1 * v2, axis=1) / (tf.linalg.norm(v1, axis=1) * tf.linalg.norm(v2, axis=1))
    y_pred = tf.cast(d > threshold, tf.float64)
    accuracy = tf.reduce_mean(tf.cast(tf.equal(y_pred, y_test), tf.float64))
    cm = tf.math.confusion_matrix(y_test, y_pred)
    return accuracy, cm
```

Predict

```
def predict(question1, question2, threshold, model, verbose=False):
    generator = tf.data.Dataset.from_tensor_slices(((([question1], [question2]), None)).batch(batch_size=1)
    v1v2 = model.predict(generator)
    v1, v2 = np.split(v1v2, 2, axis=1)
    d = tf.reduce_sum(v1 * v2, axis=1) / (tf.linalg.norm(v1, axis=1) * tf.linalg.norm(v2, axis=1))
    res = d > threshold
    if(verbose):
        print("Q1  = ", question1, "\nQ2  = ", question2)
        print("d   = ", d.numpy())
        print("res = ", res.numpy())
    return res.numpy()[0]

```