[Notes] Natural Language Processing Specialization @DeepLearning.AI

Xu, Bangyao
*bangyaoxu*0321@*gmail.com*

January 2, 2021

1 NLP with Classification and Vector Spaces

1.1 Sentiment Analysis with Logistic Regression

- Vocabulary (V): the list of unique words from the list of texts
- Feature extraction $\Rightarrow (0, 1)$ sparse representation
- Problems with sparse representation (# parameters = $|V|$):
- 1) large training time 2) large prediction time
- *Frequency dictionary*: word-class pairs $\Rightarrow$ frequencies
- Features of text m: $X_m = [1, \sum_w freqs(w, 1), \sum_w freqs(w, 0)]$
- $\Rightarrow$ [Bias, Sum Pos. Frequencies, Sum Neg. Frequencies] $\in \mathbb{R}^3$
- Preprocessing: stop words, punctuation, handles & URLs
- Stemming: transform any word to its base stem (e.g. tune, tuned, tuning $\Rightarrow$ tun)
- Logistic regression: $h(x^{(i)}; \theta) = \frac{1}{1+e^{-\theta^T x^{(i)}}} \Rightarrow$ gradient descent $\Rightarrow \theta$
- $pred = h(X_{val}, \theta) \geq 0.5 \Rightarrow accuracy = \sum_{i=1}^{m} \frac{(pred^{(i)} == y_{val}^{(i)})}{m}$

1.2 Sentiment Analysis with Naïve Bayes

- Bayes' rule: $\mathbb{P}(X|Y) = \mathbb{P}(Y|X) \times \frac{\mathbb{P}(X)}{\mathbb{P}(Y)}$
- Naïve Bayes assumes features used for classification are independent
- Naïve Bayes inference condition rule for binary classification: $\prod_{i=1}^{m} \frac{\mathbb{P}(w_i|Pos)}{\mathbb{P}(w_i|Neg)}$
- Laplacian Smoothing: $\mathbb{P}(w_i|class) = \frac{freq(w_i,class)+1}{N_{class}+V}$, where N=frequency of all words in class and V=number of unique words in vocabulary $\Rightarrow$ avoid $\mathbb{P}(w_i|class) = 0$
- $ratio(w_i) = \frac{\mathbb{P}(w_i|Pos)}{\mathbb{P}(w_i|Neg)} \Rightarrow$ Positive (>1), Negative (<1), Neutral (=1)
- $\log\left(\frac{\mathbb{P}(Pos)}{\mathbb{P}(Neg)} \prod_{i=1}^{n} \frac{\mathbb{P}(w_i|Pos)}{\mathbb{P}(w_i|Neg)}\right) \Rightarrow \log\left(\frac{\mathbb{P}(Pos)}{\mathbb{P}(Neg)}\right) + \sum_{i=1}^{n} \log\left(\frac{\mathbb{P}(w_i|Pos)}{\mathbb{P}(w_i|Neg)}\right) \Rightarrow$ log prior + log likelihood
- $\lambda(w) = \log\left(\frac{\mathbb{P}(w|Pos)}{\mathbb{P}(w|Neg)}\right) \Rightarrow$ Positive (>0), Negative (<0), Neutral (=0)
- Log likelihood: $\log \prod_{i=1}^{m} ratio(w_i) = \sum_{i=1}^{m} \lambda(w_i)$
- $score = log\ prior + log\ likelihood \Rightarrow$ values don't show up in the table of $\lambda(w)$ are considered neutral and don't contribute to the *score*
- Testing Naïve Bayes: $Accuracy = \frac{1}{m} \sum_{i=1}^{m} (pred_i == Y_{val_i})$
- Applications of Naïve Bayes: 1) Author identification 2) Spam filtering 3) Information retrieval 4) Word disambiguation
- Naïve Bayes relies on the distribution of the training data sets
- Error analysis: punctuations (e.g. **:(**), removing words (e.g. not)
- Adversarial attacks $\Rightarrow$ sarcasm, irony, and euphemism

1.3 Vector Space Models

- Vector space models can identify similarity for a question answering, paraphrasing, summarization & capture dependencies between words
- Vector space models $\Rightarrow$ represent words and documents as vectors $\Rightarrow$ identify the context around each word and capture relative meaning
- Word by word design $\Rightarrow$ number of times they occur together within a certain distance k

- Word by document design ⇒ number of times a word occurs within a certain category
- Euclidean distance: $d(\vec{v}, \vec{w}) = \sqrt{\sum_{i=1}^{n}(v_i - w_i)^2} = ||\vec{v} - \vec{w}||$
- The main advantage of cosine similarity over euclidean distance is that it isn't biased by the size difference between the representations
- Cosine similarity: $\cos(\beta) = \frac{<\hat{v},\hat{w}>}{||\hat{v}||\times||\hat{w}||}$; 0 ⇒ dissimilar; 1 ⇒ similar
- The vectors of the words that occur in similar places in the sentence will be encoded in a similar way ⇒ identify patterns
- Principal Component Analysis (PCA) ⇒ dimension reduction ⇒ visualization to see words relationships in the vector space
- Eigenvector ⇒ uncorrelated features for the data
- Eigenvalue ⇒ amount of information retained by each feature
- PCA algorithm:
- 1) Mean normalize data: $x_i = \frac{x_i - \mu_{x_i}}{\sigma_{x_i}}$;
- 2) Calculate the covariance matrix Σ;
- 3) SVD decomposition ⇒ Eigenvectors (U) & Eigenvalues (S);
- 4) Dot product to project data: $X' = XU[:, 0:k]$;
- 5) Percentage of retained variance: $\frac{\sum_{i=0}^{k} S_{ii}}{\sum_{j=0}^{d} S_{jj}}$

1.4 Machine Translation and Document Search

- Machine Translation: $\mathbf{X} \Rightarrow \mathbf{Y}$, where $\mathbf{X}$ and $\mathbf{Y}$ are vector spaces
- Align word vectors: $\mathbf{XR} \approx \mathbf{Y} \Rightarrow Loss = ||\mathbf{XR} - \mathbf{Y}||_F^2 \Rightarrow$ gradient descent for solving $\mathbf{R}$, where $||\mathbf{A}||_F$ is the Frobenius norm
- Gradient: $g = \frac{d}{d\mathbf{R}} Loss = \frac{2}{m}(\mathbf{X}^T(\mathbf{XR} - \mathbf{Y}))$
- K-nearest neighbors ⇒ translate a word even if its transformation doesn't exactly match the word embedding in the desired language
- Hash function: vectors ⇒ values (bucketing the words into regions)

- Planes: the sign of dot product indicates direction
- Multiple planes: $sign_i \geq 0 \Rightarrow h_i = 1$; $sign_i < 0 \Rightarrow h_i = 0$;
- $hash = \sum_i^H 2^i \times h_i$
- Use multiple sets of random planes for locality-sensitive hashing ⇒ Approximate nearest neighbors ⇒ sacrifice some precision in order to gain efficiency in the search
- Document vectors = sum of each individual word vectors

2 NLP with Probabilistic Models

2.1 Autocorrect

- Autocorrect changes misspelled words into the correct ones
- How Autocorrect works: 1) identify a misspelled word 2) find strings n edit distance away 3) filter candidates 4) calculate word probabilities
- Misspelled word ⇒ can't find it in a dictionary
- Edit ⇒ an operation performed on a string to change it (e.g. Insert, Delete, Switch: swap 2 adjacent letters & Replace)
- $P(w) = \frac{C(w)}{V}$, where $P(w)$ is probability of a word, $C(w)$ is the number of times the word appears and V is total size of the corpus
- Minimum edit distance ⇒ evaluate similarity between 2 strings
- $D[i, j] = source[:i] \Rightarrow target[:j]$; $D[m, n] = source \Rightarrow target$
- Minimum edit distance algorithm (tabular based approach): $src \Rightarrow tar$
- $$D[i,j] = \min \begin{cases} D[i-1,j] + delete\ cost \\ D[i,j-1] + insert\ cost \\ D[i-1,j-1] + \begin{cases} repeat\ cost, & \text{if } src[i] \neq tar[j] \\ 0, & \text{if } src[i] = tar[j] \end{cases} \end{cases}$$
- Levenshtein distance ⇒ insert cost: 1; delete cost: 1; replace cost: 2

- Dynamic programming ⇒ solving the smallest subproblem first and then reusing that result to solve the next biggest subproblem, saving that result, reusing it again and so on

2.2 Part of Speech Tagging and Hidden Markov Models

- Part of Speech (POS) tags in English: noun, verb, adjective, adverb, pronoun, preposition, etc.
- Markov chains ⇒ POS tags as *States* (Q) + *Transition matrix* (A)
- Initial states (π) assign a POS tag to the first word in the sentence
- $Q = \{q_1, ..., q_N\}$; $A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N+1,1} & \cdots & a_{N+1,N} \end{pmatrix}$
- *Emission probabilities* describe the transition from hidden states of the hidden Markov model to the observables or the words of the corpus
- Emission matrix $B = \begin{pmatrix} b_{1,1} & \cdots & b_{1,V} \\ \vdots & \ddots & \vdots \\ b_{N,1} & \cdots & b_{N,V} \end{pmatrix} \Rightarrow \sum_{j=1}^{V} b_{ij} = 1$
- Transition probabilities: 1) Count occurrences of tag pairs $C(t_{i-1}, t_i)$; 2) Calculate probabilities using the counts: $\mathbb{P}(t_i|t_{i-1}) = \frac{C(t_{i-1},t_i)}{\sum_{j=1}^{N} C(t_{i-1},t_j)}$
- Transition matrix ⇒ smoothing: $\mathbb{P}(t_i|t_{i-1}) = \frac{C(t_{i-1},t_i)+\epsilon}{\sum_{j=1}^{N} C(t_{i-1},t_j)+N\times\epsilon}$
- Don't apply smoothing to the initial probabilities in the first row of the transition matrix ⇒ don't allow a sentence to start with any parts of speech tag, including punctuation
- Emission matrix ⇒ $\mathbb{P}(w_i|t_i) = \frac{C(t_i,w_i)+\epsilon}{\sum_{j=1}^{V} C(t_i,w_j)+N\times\epsilon} = \frac{C(t_i,w_i)+\epsilon}{C(t_i)+N\times\epsilon}$
- Viterbi algorithm:
- 1) Initialization: $c_{i,1} = \pi_i \times b_{i,cindex(w_1)} = a_{1,i} \times b_{i,cindex(w_1)}$; $d_{i,1} = 0$;
- 2) Forward pass: $c_{i,j} = \max_k c_{k,j-1} \times a_{k,i} \times b_{i,cindex(w_j)}$; $d_{i,j} = \text{argmax}_k\, c_{k,j-1} \times a_{k,i} \times b_{i,cindex(w_j)} \Rightarrow C$ & D;
- 3) Backward pass: $s = \text{argmax}_i\, c_{i,K} \Rightarrow$ look up the next index in D until arriving the start token ⇒ sequence of $\{t_i\}$

2.3 Autocomplete and Language Models

- The language model (LM) can estimate probability of word sequences and probability of a word following a sequence of words ⇒ *autocomplete a sentence* with most likely suggestions
- N-gram ⇒ a sequence of N words (e.g. unigrams, bigrams, trigrams)
- Sequence notation: corpus ⇒ $w_1, ..., w_m$ (e.g. $w_{m-2}^m = w_{m-2}w_{m-1}w_m$)
- Probability of unigram: $\mathbb{P}(w) = \frac{C(w)}{m}$
- Probability of bigram: $\mathbb{P}(y \mid x) = \frac{C(x\ y)}{\sum_w C(x\ w)} = \frac{C(x\ y)}{C(x)}$
- Probability of trigram: $\mathbb{P}(w_3|w_1^2) = \frac{C(w_1^2 w_3)}{C(w_1^2)} = \frac{C(w_1^3)}{C(w_1^2)}$
- Probability of N-gram: $\mathbb{P}(w_N|w_1^{N-1}) = \frac{C(w_1^{N-1}w_N)}{C(w_1^{N-1})} = \frac{C(w_1^N)}{C(w_1^{N-1})}$
- Probability of a sequence ⇒ conditional probability + chain rule
- As the sentence gets longer, the likelihood that more and more words will occur next to each other in this order becomes smaller and smaller
- Markov assumption ⇒ only last N words matter:
- $\mathbb{P}(w_n|w_1^{n-1}) \approx \mathbb{P}(w_n|w_{n-N+1}^{n-1})$
- Entire sentence modeled with bigram: $\mathbb{P}(w_1^n) \approx \prod_{i=1}^n \mathbb{P}(w_i|w_{i-1})$
- Start of sentence symbols: $< s >$; End of sentence symbols: $< /s >$
- N-gram ⇒ add $(N-1)$ start tokens and just one end token
- Count matrix ⇒ Probability matrix: $\mathbb{P}(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}, w_n)}{C(w_{n-N+1}^{n-1})}$
- Probability matrix ⇒ Language model: 1) Sentence probability; 2) Next word prediction
- All probabilities in calculation are less than or equal to 1 and multiplying them brings risk of underflow ⇒ Log probability
- Generative language model algorithm: 1) Choose sentence start; 2) Choose next bigram starting with previous word; 3) Continue until $< /s >$ is picked

- Test data split method: 1) continuous text; 2) random short sentences
- Perplexity: $PP(W) = \mathbb{P}(s_1, s_2, ..., s_m)^{-\frac{1}{m}} \Rightarrow$ inverse probability of the test sets normalized by the number of words in the test set
- Text written by humans is more likely to have a lower perplexity score
- Perplexity for bigram models: $PP(W) = \left(\prod_{i=1}^{m} \frac{1}{\mathbb{P}(w_i|w_{i-1})} \right)^{\frac{1}{m}} \in [20, 60]$
- Log perplexity: $logPP(W) = -\frac{1}{m} \sum_{i=1}^{m} \log_2(\mathbb{P}(w_i|w_{i-1})) \in [4.3, 5.9]$
- Closed vocabulary (fixed list of words) vs. Open vocabulary (may encounter words from outside the vocabulary e.g. new city)
- Unknown word = Out of vocabulary word (OOV) $\Rightarrow <$ UNK $>$
- Criteria of creating vocabulary V: 1) min word frequency f; 2) max $|V|$, include words by frequency $\Rightarrow$ replace other words with $<$ UNK $>$
- Missing N-grams in training corpus $\Rightarrow$ their counts can't be used for probability estimation $\Rightarrow$ smoothing
- Laplacian smoothing: $\mathbb{P}(w_n|w_{n-1}) = \frac{C(w_{n-1},w_n)+1}{\sum_{w \in V} \left(C(w_{n-1},w)+1 \right)} = \frac{C(w_{n-1},w_n)+1}{C(w_n)+V}$
- Add-k smoothing: $\mathbb{P}(w_n|w_{n-1}) = \frac{C(w_{n-1},w_n)+k}{\sum_{w \in V} \left(C(w_{n-1},w)+k \right)} = \frac{C(w_{n-1},w_n)+k}{C(w_n)+k \times V}$
- Backoff $\Rightarrow$ if N-gram is missing, then use (N-1)-gram probability, and so on until finding non-zero probability (discounting needed)
- Interpolation: e.g. $\hat{\mathbb{P}}(w_n|w_{n-2}\ w_{n-1}) = \lambda_1 \times \mathbb{P}(w_n|w_{n-2}\ w_{n-1}) + \lambda_2 \times \mathbb{P}(w_n|w_{n-1}) + \lambda_3 \times \mathbb{P}(w_n)$, where $\sum_i \lambda_i = 1$ and λ_i's are learned from the validation part of the corpus

2.4 Word embeddings with neural networks

- Integers representation $\Rightarrow$ One-hot vectors:
- Simple & No implied ordering vs. Huge & No embedding meaning
- Word embedding vectors $\Rightarrow$ 1) low dimension 2) embed meaning (e.g. semantic distance, analogies)
- Word embedding process (corpus $\Rightarrow$ word embeddings):

- 1) Corpus (word in context) ⇒ general purpose, specialized;
- 2) Embedding method (meaning) ⇒ machine learning (self-supervised)
- Basic word embedding methods: 1) word2vec (Google); 2) Global Vectors/GloVe (Stanford); 3) fastText (Facebook)
- Advanced word embedding methods: 1) BERT (Google); 2) ELMo (Allen Institute for AI); 3) GPT-2 (OpenAI) ⇒ pre-trained models
- Continuous bag-of-words (CBOW) word embedding ⇒ predict a missing word based on the surrounding words (if two unique words are both frequently surrounded by a similar sets of words when used in various sentences, then those two words tend to be related in their meaning, i.e. related semantically)
- Corpus ⇒ training: Context words ⇒ Center word (sliding window)
- Corpus cleaning: 1) letter case ⇒ lowercase; 2) punctuation ⇒ "."; 3) numbers ⇒< NUMBER >; 4) special characters; 5) special words
- Context words ⇒ vectors: *average* of one-hot vectors of each word
- Architecture of CBOW model: input layer (context words vector $X \in \mathbb{R}^{V \times m}$) ⇒ ReLU (W_1, b_1) ⇒ hidden layer ($H \in \mathbb{R}^{N \times m}$) ⇒ softmax (W_2, b_2) ⇒ output layer (center word vector $\hat{Y} \in \mathbb{R}^{V \times m}$)
- Cross-entropy loss (log-loss): $J = -\sum_{k=1}^{V} y_k \log(\hat{y}_k)$
- Forward propagation of CBOW:
- $Z_1 = W_1 X + B_1; H = \text{ReLU}(Z_1); Z_2 = W_2 H + B_2; \hat{Y} = \text{softmax}(Z_2)$
- Cost of CBOW: $J_{batch} = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{V} y_j^{(i)} \log(\hat{y}_j^{(i)}) = -\frac{1}{m} \sum_{i=1}^{m} J^{(i)}$
- Minimizing the cost ⇒ Back propagation + Gradient descent
- Extracting word embedding vectors:
- 1) W_1; 2) W_2; 3) $W_3 = \frac{1}{2}(W_1 + W_2)$
- Intrinsic evaluation ⇒ Test relationships between words (e.g. analogies, clustering, visualization)
- Extrinsic evaluation ⇒ Test word embeddings on external task (e.g. named entity recognition, parts-of-speech tagging)

3 NLP with Sequence Models

3.1 Neural Networks for Sentiment Analysis

- Zero padding ⇒ ensure all of the vectors have the same size
- Advantages of using frameworks (e.g. Trax based on Tensorflow): 1) run fast on CPUs, GPUs and TPUs; 2) parallel computing; 3) record algebraic computations for gradient evaluation
- Trax ⇒ deep learning library with clear code and speed
- Dense Layer ⇒ $z^{[i]} = W^{[i]}a^{[i-1]}$
- ReLU Layer ⇒ $a^{[i]} = g(z^{[i]}), g(z^{[i]}) = \max(0, z^{[i]})$
- Serial Layer ⇒ a composition of sublayers
- Embedding Layer ⇒ map words to embeddings
- Mean Layer ⇒ average the word embeddings ⇒ vector representation
- Computing gradients in Trax ⇒ $trax.math.grad(f)$
- Training with $grad()$:
- 1) Setup the model: $y = model(x)$;
- 2) Forward and Back-propagation: $grads = grad(y.forward)(y.weights, x)$;
- 3) Gradient descent: $weights- = alpha * grads$

3.2 Recurrent Neural Networks for Language Modeling

- Traditional N-gram language models require a lot of space and memory
- RNNs aren't limited to looking at just the previous n words and propagate information from the beginning of the sentence to the end
- Learn-able shared parameters in plain RNNs ⇒ W_x, W_h, W
- RNNs can be implemented for a variety of NLP tasks (e.g. machine translation and caption generation)
- Math behind a vanilla RNN:
- $h^{<t>} = g\big(W_h[h^{<t-1>}, x^{<t>}] + b_h\big)$; $\hat{y}^{<t>} = g\big(W_y h^{<t>} + b_y\big)$

- Corss entropy loss $\Rightarrow J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{K} y_j^{<t>}\log(\hat{y}_j^{<t>}) \Rightarrow$ the loss function is just an average through time for RNNs
- Tensorflow $tf.scan()$ mimics RNNs $\Rightarrow$ GPUs, parallel computing
- Gated Recurrent Units (GRUs) allow relevant information to be kept in the hidden states, even over long sequences:
- $\Gamma_u = \sigma(W_u[h^{<t_0>}, x^{<t_1>}] + b_u);\ \Gamma_r = \sigma(W_r[h^{<t_0>}, x^{<t_1>}] + b_r);$
- $h'^{<t_1>} = \tanh(W_h[\Gamma_r \times h^{<t_0>}, x^{<t_1>}] + b_h);$
- $h^{<t_1>} = \Gamma_u \times h^{<t_0>} + (1 - \Gamma_u) \times h'^{<t_1>};\ \ \hat{y}^{<t_1>} = g(W_y h^{<t_1>} + b_y)$
- Bi-directional RNNs $\Rightarrow$ information flows from the past and from the future independently $\Rightarrow \hat{y}^{<t>} = g(W_y[\overrightarrow{h}^{<t>}, \overleftarrow{h}^{<t>}] + b_y)$
- Deep RNNs are just RNNs stuck together $\Rightarrow$ intermediate connections pass information through the values of activations: 1) get hidden states for current layer; 2) pass the activations to the next layer

3.3 LSTMs and Named Entity Recognition

- Solving for vanishing or exploding gradients: 1) identify RNN with ReLU activation; 2) gradient clipping; 3) skip connections
- LSTM is a special variety of RNN that was designed to handle entire sequences of data by learning when to remember and when to forget $\Rightarrow$ offer a solution to vanishing gradients
- LSTM applications: 1) next-character prediction; 2) chatbots; 3) music composition; 4) image captioning; 5) speech recognition
- Typical LSTMs have a cell and three gates:
- 1) Forget gate $\Rightarrow$ decides what to keep;
- 2) Input gate $\Rightarrow$ decides what to add;
- 3) Output gate $\Rightarrow$ decides what the next hidden state will be
- Named Entity Recognition (NER) locates and extracts predefined entities from text (e.g. places, organizations, names, time, & dates)
- Applications of NER systems: 1) search engine efficiency; 2) recommendation engines; 3) customer services; 4) automatic trading

- Processing data for NERs: 1) assign each class a number; 2) assign each word a number; 3) token padding (< PAD >) ⇒ same-length numerical arrays
- Training NER: Inputs ⇒ LSTM layer ⇒ Dense layer ⇒ LogSoftmax
- Layer in Trax:
- $model = tl.Serial(tl.Embedding(), tl.LSTM(), tl.Dense(), tl.LogSoftmax())$
- Remember to mask the padding tokens when computing accuracy

3.4 Siamese Networks

- Siamese network is a neural network made up of two identical neural networks which are merged at the end ⇒ identify question duplicates
- Comparing meaning is not as simple as just comparing words
- Siamese network identify similarity or difference between things
- The cosine similarity gives the Siamese networks prediction $\hat{y} \in [-1, 1]$
- Threshold τ: $\hat{y} \leq \tau \Rightarrow$ different; $\hat{y} > \tau \Rightarrow$ same
- Loss function: $diff = s(A, N) - s(A, P)$, where A means Anchor and P and N denote Positive and Negative
- Triplet loss: $\mathcal{L}(A, P, N) = \max(diff + \alpha, 0)$
- Hard triplets (e.g. $s(A, N) \approx s(A, P)$) are better for training
- Matrix similarities $s(v_1, v_2)$ in a batch: diagonal ⇒ positive examples
- Cost function: $\mathcal{J} = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$
- Hard negative mining:
- 1) Mean negative: mean of off-diagonal values in each row;
- 2) Closest negative: off-diagonal value closest to (but less than) the value on diagonal in each row
- $\mathcal{L}_1 = \max(mean\ negative - s(A, P) + \alpha, 0)$
- $\mathcal{L}_2 = \max(closest\ negative - s(A, P) + \alpha, 0)$
- Full cost function: $\mathcal{L}_{Full}(A, P, N) = \mathcal{L}_1 + \mathcal{L}_2$
- One shot learning ⇒ measure similarity between 2 classes

4 NLP with Attention Models

4.1 Neural Machine Translation

- Seq2Seq model maps variable-length sequences to fixed-length memory
- Major limitation of traditional Seq2Seq ⇒ information bottleneck
- Word alignment ⇒ identify relationships among the words in order to make accurate predictions in case the words are out of order or not exact translations ⇒ *Attention layer* which performs a series of calculations that's assigned some inputs, more weights than others
- Attention ⇒ Taking a query, selecting the place where the highest likelihood to look for the key, then finding the key ⇒ $Attention = Softmax(QK^T)V$, where Q, K, V denote Query, Keys and Value score
- The flexible system finds matches even between languages with very different grammatical structures (e.g. English-German)
- Machine translation setup ⇒ state-of-the-art uses pre-trained vectors
- *Teacher forcing* allows the model to 'check its work' at each step ⇒ the actual outputs, or ground-truth, is the input to the decoder for each time step until the end of the sequence is reached
- BLEU (Bilingual Evaluation Understudy) score ⇒ evaluate the quality of machine-translated text by comparing *candidate* text to one or more *reference* translations
- BLEU score = $\frac{\text{sum over unique n-gram counts in the candidate}}{\text{total number of words in candidate}}$
- BLEU doesn't consider semantic meaning and sentence structure
- ROUGE (Recall-Oriented Understudy for Gisting Evaluation) measures precision and recall between generated & human-created texts
- Recall ⇒ How much of the reference text is the system text capturing?
- Recall = $\frac{\text{sum of overlapping unigrams in model and reference}}{\text{total number of words in reference}}$
- Precision ⇒ How much of the model text was relevant?
- Precision = $\frac{\text{sum of overlapping unigrams in model and reference}}{\text{total number of words in model}}$
- ROUGE doesn't take themes or concepts into consideration

- Greedy decoding ⇒ select the most probable word at each step, but the best word at each step may not be the best for longer sequence
- Random sampling ⇒ provide probabilities for each word, and sample accordingly for the next outputs
- In sampling, *temperature* is a parameter allowing for more or less randomness in predictions ⇒ low temperature: more confident, conservative network; high temperature: more excited, random networks
- Beam search decoding selects multiple options for the best input based on conditional probability ⇒ tend to carry more weight than single tokens & can cause translation problem with speech corpus not cleaned
- Minimum Bayes Risk (MBR) ⇒ 1) generate several random samples; 2) compare many samples against one another and assign the similarity scores; 3) select the sample with the highest similarity

4.2 Text Summarization

- Seq2Seq problems: 1) loss of information; 2) vanishing gradient
- Transformer ⇒ Multi-headed attention & Positional encoding
- State of the art transformers: 1) Generative Pre-training for Transformer (GPT-2); 2) Bidirectional Encoder Representations from Transformers; 3) Text-to-text transfer transformer (T5)
- T5 is a powerful multi-task transformer ⇒ 1) translation; 2) classification; 3) Q&A; 4) regression; 5) summarization
- Math behind attention: $K \in \mathbb{R}^{L_K \times D}$, $Q \in \mathbb{R}^{L_Q \times D}$, $V \in \mathbb{R}^{L_K \times D}$;
- $W_A = softmax(QK^T) \in \mathbb{R}^{L_Q \times L_K}$; $Z = W_A V \in \mathbb{R}^{L_Q \times D}$
- Three ways of attention:
- 1) Encoder/decoder attention ⇒ One sentence (decoder) looks at another one (encoder);
- 2) Casual (self) attention ⇒ In one sentence, words look at previous words (used for generation);
- 3) Bi-directional self attention ⇒ In one sentence, words look at both previous and future words

- Casual attention: 1) Queries and keys are words from the same sentence; 2) Queries should only be allowed to look at words before
- Casual attention math $\Rightarrow W_A = softmax(QK^T + M)$, where all values on the diagonal and below of M is 0 and all other values are $-\infty \Rightarrow$ queries are only allowed to search among the past words
- Multi-head attention $\Rightarrow$ each head uses different linear transformations to represent words & different heads can learn different relationships between words
- Math behind multi-head attention:
- $MultiHead(Q, K, V) = Concat(h_1, ..., h_h)W_0$;
- where $h_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$
- Transformer decoder: Input embedding $\Rightarrow$ Positional encoding $\Rightarrow$ Decoder block $\Rightarrow$ Linear layer $\Rightarrow$ Softmax $\Rightarrow$ Output probabilities
- Decoder block structure: Positional input embedding $\Rightarrow$ Multi-head attention $\Rightarrow$ Normalization layer $\Rightarrow$ Feed forward layer (ReLU) $\Rightarrow$ Normalization layer (Repeat N times) $\Rightarrow$ Output
- Transformer for summarization:
- 1) Model input: *Article text* < EOS > *Summary* < EOS > < PAD >;
- 2) Tokenize the model input into numeric vector;
- 3) Define the loss weights as 0 until the first < EOS >, and 1 on the start of the summary;
- 4) Provide *Article text* and generate *Summary* word-by-word, pick the next word by random sampling until the final < EOS >

4.3 Question Answering

- Question answering: context-based vs. closed book
- T5 is trained on Colossal Clean Crawled Corpus (C4) $\approx$ 800GB
- Desirable goals for *transfer learning*:
- 1) reduce training time; 2) improve predictions; 3) small dataset

- Transfer learning approaches: 1) feature-based (pre-training data); 2) fine-tuning (pre-training tasks)
- Continuous Bag of Words (CBOW) uses fixed window ⇒ ELMo uses full context by RNN (bi-directional LSTM) ⇒ Open AI GPT uses decoder (uni-directional) ⇒ BERT uses encoder (bi-directional) ⇒ T5 uses both encoder and decoder (multi-task)
- BERT framework ⇒ pre-training (unlabeled data over pre-training tasks) + fine-tuning (labeled data from downstream tasks)
- Masked language modeling (MLM) ⇒ choose 15% of the tokens at random: mask them 80% of the time; replace them with a random token 10% of the time; or keep as is 10% of the time
- The input for BERT: Position embeddings + Segment embeddings + Token embeddings
- BERT objectives: 1) Multi-mask LM (Cross entropy loss); 2) Next sentence prediction (Binary loss)
- Data training strategies for T5: 1) example-proportional mixing; 2) equal mixing
- *Gradual unfreezing* ⇒ freeze the last layer, then fine-tune using that and keep the others fixed, and so on
- *Adapter layers* ⇒ add a neural network to each feed forward and each block of the transformer & only these new adapter layers and the layer normalization parameters are being updated during fine-tuning
- General Language Understanding Evaluation (GLUE) benchmark ⇒ a collection used to train, evaluate, analyze natural language understanding systems
- Implement Q&A with T5: load a pre-trained model ⇒ process data to get the required inputs and outputs ⇒ fine tune the model on the new task and input ⇒ predict using the new model

4.4 Chatbot

- Long sequence applications in NLP: 1) writing books; 2) chatbots
- Transformer issues:

- 1) Attention on sequence of length L takes L^2 time and memory;
- 2) N layers take N times as much memory
- Local Sensitive Hashing (LSH) attention:
- 1) Hash Q and K;
- 2) Standard attention within same-hash bins;
- 3) Repeat a few times to increase probability of key in the same bin
- LSH is a probabilistic, not deterministic model because of the inherent randomness within the LSH algorithm (the hash can change along the buckets a vector finds itself map to)
- Standard transformer:
- $y_a = x + Attention(x);\ y_b = y_a + FeedFwd(y_a)$
- Reversible layers equations:
- 1) $y_1 = x_1 + Attention(x_2);\ y_2 = x_2 + FeedFwd(y_1);$
- 2) $x_1 = y_1 - Attention(x_2);\ x_2 = y_2 - FeedFwd(y_1)$
- The *reformer* is a transformer model designed to handle context windows of up to 1 million words using LSH attention & reversible layers
- MultiWOZ is a very large datasets of human conversations, covering multiple domains and topics

■