|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| NN | Perceptron = linear fn that aggregates input signals + threshold fn that determines if response neuron fires or not + learning procedure to adjust weights  Cost fn: E(, y) = MSE =  For particular weight w, e(b, w, x, y) = (y - (b + wx))2  gradient of error w.r.t to weights =  Then | | | | Linear fn:  Threshold decision fn:  Learning procedure: ,  where | |
| Activation fn | Sigmoid fn:  Sigmoid fn for non-linearity  Can start trying w ReLU first | | For binary classification problems, use threshold function:  For regression, use identity fn:  For multiclass classification problems, use softmax fn: | | | |
| Back Propagation | Cost fn: , where = predicted value for training example k  If calculating loss, cost fn = cross entropy =  = p.d of error w.r.t sigmoid activation \* pd of sigmoid wrt linear fn \* pd of linear fn wrt weight  = [a(L) - y] \* [a(L)(1 - a(L))] \* [a(L-1)]  = [a(L) - y] \* [a(L)(1 - a(L))] \* [w(L)] \* [a(L-1)(1 - a(L-1))] \* x(L-1) (assuming 1 hidden layer) | | | | | |
| Gradient Descent | 1) Stochastic g.d. = 1 data point. Escape local minima but slow  2) Mini-batch g.d. = some data points. Faster and might escape local minima  3) Batch g.d. = all data points. Fastest but most likely to get stuck in local minima  4) Momentum g.d.: moving average of past gradients, which smooths out noisy updates, aiding optimization process.  Momentum moves faster and has shot at escaping local minima  *delta = -learningRate \* gradient + previousDelta \* decayRate*  5) ADAGRAD: adapts learning rate individually for each param based on historical gradient info, can handle sparse data and features. Can escape saddle point better compared to normal g.d. or momentum  *sumOfGradientSquared = previousSumOfGradientSquare + gradient2*  *delta = -learningRate \* gradient / sqrt(sumOfGradientSquared)*  6) RMSProp: mitigates diminishing learning rate issue observed in AdaGrad by adding a decay factor (AdaGrad is slow as sum of gradient squared only grows and never shrinks)  *sumOfGradientSquared = previousSumOfGradientSquare \* decayRate + gradient2 \* (1 - decayRate)*  *delta = -learningRate \* gradient / sqrt(sumOfGradientSquared)*  7) ADAM (Adaptive Moment Estimation): best of both Momentum and RMSProp  ADAM gets speed from Momentum and ability to adapt gradients in diff directions from RMSProp  *sumOfGradient = previousSumOfGradient \* beta1 + gradient \* (1 - beta1)*  *sumOfGradientSquared = previousSumOfGradientSquare \* decayRate + gradient2 \* (1 - decayRate)*  *delta = -learningRate \* sumOfGradient / sqrt(sumOfGradientSquared)* | | | | | theta += delta  Backpropagation = Gradient Descent for all weight parameter  However, still high chancces of local optima -> need for deep learning |
| Deep Learning | | Automated feature engineering. At each layer, breaks down task into simpler problem (simplified loss landscape) | | | | |
| Regularization | | L1 regularization: Lasso, feature selection, diamond. Cost fn = Loss +  L2 regularization: Ridge, shrink to 0, circle. Cost fn = Loss +  Early stopping: when validation error start increasing  Dropout: Assign each neuron a dropout prob. Weight each neuron output by same prob **p** – proportional to the fraction of time the neuron was present during training (model averaging effect)  Data Augmentation: more diverse set of training examples  Batch Normalization, Layer Normalization. Weight decay | | | | |
| Training | No guarantee of convergence. Many epochs required. Termination criteria. To avoid local minima: initialise w diff random weights | | | To prevent over training: - Use held-out validation set  - dont run too many epoch, or have too many hidden layers | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| NLP Pipeline | | | | | Raw Text -> Preprocessing -> Featurization -> ML model -> Prediction | | | | | | | | | | | |
| Preprocessing | | | | Raw Text -> 1) Sentence segmentation/tokenization -> 2) Tokenization -> 3) POS Tagging -> 4) Lemmatization -> 5) Stop words and cleaning -> 6) Dependency Parsing -> 7) Noun Phrases -> 8) NER -> Processed Text  Not all steps are necessary. Steps do not have to follow exact sequence. Steps are dependent on problem statement | | | | | | | | | | | | |
| 1) | | | | *import nltk*  *sentences = nltk.sent\_tokenize(text)* | | | | | | | | | *for sentence in sentences:*  *print(sentence)* | | | |
| 2) Word tokenization | | | | *from nltk.tokenize import word\_tokenize*  *word\_tokenize(text)*  *for sentence in sentences:*  *print(word\_tokenize(sentence))*  Let N = num of tokens. V = vocabulary = set of words | | | | | Problems: Fragments/pauses in transcription. Lemma (cat & cats: same lemma).  Wordform (full inflected surface form, cat and cats have diff wordforms)  Language issues: German noun compounds are not segmented by whitespace  Chinese and Japanese have no spaces btw words  Japanese have multiple alphabets intermingled | | | | | | | |
| 2a) | Tokenization is vocabulary dependent. Can define vocab based on previously defined tokens OR dynamically for given corpus OR based on rules  BPE and WordPiece get vocab by training dynamically for a given corpus | | | | | | | | | | | | | | | |
| Word-based models: (faster, fastest, quickest). Pros: intuitive  Cons: large vocab (many words); word combination; abbreviations; language issues | | | | | | | | | Character-based models: (f, a, s, t, e, r, q, u, i, c, k): Pros: Very small vocabulary (26 letters)  Cons: lack of meaning; increased input computation; limits network choices | | | | | | |
| Subword tokenization: (fast, er, est, quick): Middle ground btw word-based and character-based  - Tokenization scheme that deals w an infinite potential vocab via a finite list of known words  - Words like "any" and "place" which make "anyplace" or compound words like "anyhow" or "anybody"  - Subword chunks do not need to be whole words | | | | | | | | | | | | | | | |
| 2b) Tokenization Algo | | | | 1. Byte-Pair Encoding (BPE): data compression algo that iteratively replaces the most frequent pair of bytes w a single unused byte  In NLP, BPE is used to tokenize text into subwords  1) Get word count freq  2) Get initial token count & freq of each character  3) Merge most common byte pairing  4) Add to list of tokens and recalculate freq count for each token  5) Repeat until reached defined token limit or set num of iterations  If new word like 'mug', then would become ['[UNK]', 'ug'] since letter 'm' not in our vocab | | | | | | | | 1) ('hug, 10'), ('pug', 5), ('pun', 12), ('bun', 4), ('hugs', 5)  2) ('h', 'u', 'g', 10), (p, u, g, 5), (p, u, n, 12), (b, u, n, 4), (h, u, g, s, 5)  3) Merge 'u', 'g' as total counts = 10 + 5 + 5 = 20  4) Vocab: [h, u, g, p, n, b, s, ug]  Corpus: (h, ug, 10), (p, ug, 5), (p, u, n, 12), (b, u, n, 4), (h, ug, s, 5)  3) Merge 'u', 'n' as total counts = 12 + 4 = 16  4) Vocab: [h, u, g, p, n, b, s, ug, un]  Corpus: (h, ug, 10), (p, ug, 5), (p, un, 12), (b, un, 4), (h, ug, s, 5)  3 + 4) Merge 'h', 'ug'. Vocab: [h, u, g, p, n, b, s, ug, un, hug]  Corpus: (hug, 10), (p, ug, 5), (p, un, 12), (b, un, 4), (hug, s, 5) | | | | |
| Problems: can result in potentially ambiguous vocab. E.g. vocab = [d, de, ee, ep, e]. Then subwords can be (d, e, ep) or (de, ep)  Pros: BPE can handle a wide range of vocab including rare and out-of-vocab words | | | | | | | | | | | | |
| 2. WordPiece: start from small vocab inluding special tokens used by model and initial alphabet. Each word in initially split by adding a prefix (## for BERT) to all characters inside the word  BPE selects most frequent pair for merging but WordPiece computes a score for each pair:  WordPiece prioritize merging of pairs when individual parts are less freq in vocab | | | | | | | | 2) (h, ##u, ##g, 10), (p, ##u, ##g, 5), (p, ##u, ##n, 12), (b, ##u, ##n, 4), (h, ##u, ##g, ##s, 5)  3) Score for (##u, ##g) = 1/36 < 1/20 = score for (##g, ##s)  4) Vocab: [h, p, b, ##u, ##g, ##n, ##s, ##gs]  Corpus: (h, ##u, ##g, 10), (p, ##u, ##g, 5), (p, ##u, ##n, 12), (b, ##u, ##n, 4), (h, ##u, ##gs, 5).....  Vocab: [h, p, b, ##u, ##g, ##n, ##s, ##gs, hu, hug]  (hug, 10), (p, ##u, ##g, 5), (p, ##u, ##n, 12), (b, ##u, ##n, 4), (hu, ##gs, 5) | | | | |
| WordPiece vs BPE: WordPiece only saves the final vocab, not the merge rules learned.  Starting from the word to tokenize, WordPiece finds the longest subword that is in the vocabulary, then splits on it.  E.g., for the word "hugs" the longest subword starting from the beginning is "hug", so we split there and get ["hug", "##s"].  With BPE, we would have applied the merges learned in order and tokenized this as ["hu", "##gs"], so the encoding is different. | | | | | | | | | | | | |
| *from transformers import AutoTokenizer*  *tokenizer = AutoTokenizer.from\_pretrained('gpt2') # BPE: GPT, GPT2, RoBErTa, BART, DeBERTa. For WordPiece: bert-base-cased*  *tokens = tokenizer.tokenizer(sentence)* | | | | | | | | | | | | |
| 3. Unigrams: Compared to BPE and WordPiece, Unigram works in the other dirn: it starts from a big vocab and removes tokens from it until it reaches the desired vocabulary size. To build that base vocab: can take the most common substrings in pre-tokenized words, or apply BPE on the initial corpus with a large vocab size.  At each step of training, algo computes a loss over the corpus given current vocab  For each symbol in vocab, algo computes how much loss would incr if symbol was removed, and looks for symbols that would incr it the least. Those symbols have a lower effect on overall loss, hence are 'less needed'  Very costly operation, so don't just remove single symbol, but remove p% (usually 10 or 20%) of symbols w lowest loss incr  Never remove base characters, to make sure any word can be tokenized  *# Unigram: xlnet-base-cased, AlBERT, T5, mBART, Big Bird* | | | | | | | | | | | | |
| 2c) Limitations | | | | 1) Lower/upper case. Tokenizer treats same word differently based on cases  2) Numbers: Tokenizer's inconsistency in representing each num. E.g. 500 may be represented as 1 token, while 501 = [50, 1]  3) Trailing whitespace: E.g. "water" could be represented as " water" as one token instead of [" ", "water"]  Can impact prob of predicting next word if you finish your prompt w a whitespace or not  4) Model-specific/data-specific: Although most language models use BPE, they still train a new tokenizer for their own models | | | | | | | | | | | | |
| 5) | | | | 1) Converting to lower case (before tokenization): *sentence.lower()*  2) Removing nums: *import re; result = re.sub(r'\d+', '', sentence)*  3) Removing punctuation (if needed): *result = re.sub(r'[^\w\s]', '', sentence) #ignore words and whitespace* | | | | | | | | | | | | |
| 4) Removing stopwords (common words) so that more attention is given to relevant words  *from nltk.corpus import stopwords*  *from nltk.tokenize import word\_tokenize* | | | | | | | | *stop\_words = set(stopwords.words('english'))*  *word\_tokens = word\_tokenize(sentence)*  *filtered\_sentence = [w for w in word\_tokens if w.lower() not in stop\_words]* | | | | |
| 5) Removing HTML Tags. *html\_pattern = "<(?:\"[^\"]\*\"['\"]\*|'[^']\*'['\"]\*|[^'\">])+>" ; re.sub(html\_pattern, '', html\_str)* | | | | | | | | | | | | |
| 6) Removing accented characters: standardise characters into ASCII. E.g. converting é to e  7) Expanding contractions: E.g. we're = we are; we've = we have  8) Removing special characters like emojis, hashtags, ...: *regex\_pattern = "#(\w+)"*  9) Correction of typos: Requires dictionary to map words to correct form based on similarity. Phonetic similarity | | | | | | | | | | | | |
| 4) | | | | Standardization transform words and phrases into their normal/standard form. Require manual efforts using data dict or regex  E.g. acronyms: NLP, NLU, NLG. or slangs: "rt", "luv", "dm"  Normalization: Different forms of a word e.g. drive, drives, driving & Related words have similar meaning e.g. am, are, is -> be  This reduces num of unique tokens, removes variations of word, removes redundant info, convert high dimensional to low dimensional  Popular methods for normalization are Stemming and Lemmatization | | | | | | | | | | | | |
| 4a) Stemming | | | | Word stems = base form of a word. Can create new words by attaching affixes to them in a process known as inflection  Stemming algos identify conditions on language (suffix and prefix or affixes), eliminate this to get stem words | | | | | | | | *from nltk.stem import PorterStemmer*  *# Also have RegexpStemmer, LancasterStemmer, SnowballStemmer*  *word\_stemmer = PorterStemmer()*  *word\_stemmer.stem(word)* | | | | |
| 4b) Lemmatization | | | | | | Output 'lemma' which is a root word. Output is a legit word unlike stemming which just removes affixes (E.g. stemming: located -> locat; lemma: located -> locate)  Lemmatization accounts for POS values. May generate diff outputs for diff POS values  Stemming easier to implement, faster. Lemmatization more accurate | | | | | | | | *from nltk.stem import WordNetLemmatizer*  *lemmatizer = WordNetLemmatizer()*  *lemmatizer.lemmatize(word)* | | |
| 3) Parts Of Speech (POS) | | Assigns parts of speech to each word (nouns, verbs, adjectives)  {CC: conjunction of coordinating, CD: digit of cardinal, DT: determiner, EX: existential, FW: foreign word, IN: preposition and conjunction, JJ: adjective, JJR and JJS: adjective and superlative, LS: list marker, MD: modal, NN: singular noun, [NNS, NNP, NNPS]: proper and plural noun, PDT: predeterminer, WRB: adverb of wh, WPS: possessive wh, WP: pronoun of wh, WDT: determiner of wp, VBZ: verb, [VBP, VBN, VBG, VBD, VB]: forms of verbs, UH: interjection, TO: to go; RB: particle, [RBS, RB, RBR]: adverb, [PRP, PRP$]: pronoun personal and professional}  [('London', 'NNP'), ('is', 'VBZ'), ('the', 'DT'), ('capital', 'NN'), ('and', 'CC'), ('most', 'RBS'), ('populous', 'JJ'), ('city', 'NN'), ...] | | | | | | | | | | | | | | *from textblob import TextBlob*  *result = TextBlob(sentence)*  *print(result.tags)* |
| 7) | | Can group the noun phrases to generate: [London, is, the capital, and, most, populous city ...] = [Proper Noun, verb, noun, conjunction, adverb, noun]. NP = noun phrases  (S London/NNP is/VBZ (NP the/DT capital/NN) and/CC most/RBS (NP populous/JJ city/NN), ...) | | | | | | | | | | | | | *reg\_exp = "NP: {<DT>?<JJ>\*<NN>}"*  *rp = nltk.RegexpParser(reg\_exp)*  *result = rp.parse(result.tags)* | |
| 8) | | | | Named Entity Recognition (NER). From earlier e.g. nouns are [London, capital, city, England, United Kingdom]  NER detect and label there nouns w real-world concepts that they represent  Geographic Entity: [London, England, United Kingdom]  Can label entities like people's name, company names, geographic locations, product names | | | | | | | *from nltk import word\_tokenize, pos\_tag, ne\_chunk*  *nltk.download('maxent\_ne\_chunker'); nltk.download('words')*  *words = word\_tokenize(sentence); pos\_tags = pos\_tag(words)*  *ner\_tree = ne\_chunk(pos\_tags)*  *for chunk in ner\_tree:*  *if hasattr(chunk, 'label'):*  *print(f"Entity: {' '.join(c[0] for c in chunk)}, Label: {chunk.label()}")* | | | | | |
| Feature engineering | | | | | | | Bag-of-words (BoW) representations. TF-IDF. Embeddings | | | | | | | | | |
| BoW | From given text, create vocab. For a given sentence, create vector as count of word in the vocab and represent.  E.g. [1, 1, 2, 0, 0, 3]. If vocab is huge, then vector would be sparse.  Instead use sparse vector. Suppose we have a dict {word1: id1, word2: id2, ...}. Then sparse vector of (id, counts): (1, 1), (2, 1), (3, 2), (4, 0), ...  Cons: With BoW, original word order is lost. Words like 'is', 'the' have higher weight | | | | | | | | | | | | | | | |
| N-grams | | | | To capture order of nearby words, use N-grams. E.g. The cat sat on the mat  2-grams: the-cat, cat-sat, sat-on, on-the, the-mat. 3-grams: the-cat-sat, cat-sat-on, sat-on-the, on-the-mat  Can use multiple n-grams features in ML models. E.g. unigrams + bigrams + trigrams  If original vocab size is |V|, then num of 2-grams is |V|2, for 3-grams is |V|3  But natural language n-grams have a power law freq structure. I.e. most of the n-grams are common. | | | | | | | | | | | | |
| Term freq-inverse document freq (TF-IDF) | | | | Show how impt a word is to a document in a corpus (how unique). To solve BoW problem of common words having higher weight  , where nt,d = num of times term "t" appear in document "d"  How impt a term is,  . Words w higher score = more impt | | | | | | | | | | | | |
| BoW and TF-IDF cons | | | | If new sentences contain new words, vocab size would increase and hence length of vectors would incr.  Vecotrs would contain many 0s, resulting in sparse matrix. Also retain no info on grammar or ordering of words | | | | | | | | | | | | |
| Embeddings | | | Word2Vec embeddings puts similar words nearby in space. Word2Vec is based on skip-grams and CBOW | | | | | | | | | | | | | |
| Skip-grams: analyze meaning of word by looking at contexts in which it occurs. Context = set of words that occur near the word, i.e. at displacements of ..., -3, -2, -1, +1, +2, +3, ... in each sentence where word occurs. E.g. The quick brown fox jumps over the lazy dog.  The: (the, quick), (the, brown). quick: (quick, the), (quick, brown), (quick, fox)  brown: (brown, the), (brown, quick), (brown, fox), (brown, jumps)...  Skips-grams: predict the context given the word  Input to NN: One-Hot encoded vector  E.g. Vocab has 10000 words. Put a 1 in the position corresponding to the word 'quick'  Output would be a 10000 dimensional vector, representing the prob of that word being the context word for our input target word  For each position t = 1, to T, predict context words within a window of fixed size m, given center word. Data likelihood =  Objective fn = (average) negative log likelihood  , where wO = context/output word and wI = input words  Minimize objective fn loss using gradient descent  Rows of the input-hidden layer weight matrix are used as the word vectors (word embeddings)  Negative sampling: ignores most of the '0' in the one-hot label word vector, and only propagates and updates the weights for the target and a few negative classes which were randomly sampled.  - Take k -ve samples (using word probabilities). - Maximize log-likelihood of target word; minimise log-likihood of sampled -ve words  - Sample w P(w) = U(w)3/4/Z, where U(w) is the unigram distribution. - Power of 3/4 makes less freq words be sampled more often  - reduce computation by sampling just k -ve instances along w the target word instead of computing denominator of  Semantic and syntactic r/s: can capture r/s btw similar words in embedding | | | | | | | | | | | | | |
| CBOW | | | | CBOW: instead of predicting the context words, input them into model and ask the network to predict the current word  Skip-grams works well w small datasets and can better represent rare words  CBOW is faster to train and can better represent freq words | | | | | | | | | | | | |
| GLOVE and FastText are inspired from Word2Vec | | | | *w2v.most\_similar('happy')* to get words w similar meaning as happy and their probs | | | | | | | | |
| Limitations of Word2Vec: Many words have multiple meanings so one vector cannot capture all meanings | | | | | | | | | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Topic Modeling | | Each document consists of ≥ 1 topics and Each topic consists of a collection & distribution of tokens/words  Topic = Similar records (sentences/paragraphs/documents) clustered tgt based on the tokens (words)  Topic modeling = unsupervised mtd of finding topics in a collections of documents (based on distribution of tokens in doc)  Topic modeling divides a corpus of documents into: 1) A list containing all topics that are covered by the docs in the corpus  2) Grouped several sets of docs from the corpus-based on the topics they cover | | |
| Key objectives: 1) Topic term dist (which are the most impt topics) & 2) Document to topic dist (which topics are assigned to every doc)  Topic modeling tries to find the latent structure in a text corpus that: - resembles "topics", - best summarizes the collection,  - is based on statistical patterns, - are obscured by synonyms, homonyms, stopwords, ..., - may overlap | | |
| Similar to clustering but clustering focus on data points/documents but topic modelling focus on the topics/clusters themselves | | |
| Mtds of topic modeling: Latent Dirichlet Allocation (LDA. Can be Bayesian or Matrix factorization approach), Non-negative Matrix Factorization (NMF), Latent Semantic Allocation (LSA), Doc2Vec, BERTopic, ... | | |
| LDA | Latent: This refers to everything that we don’t know a priori and are hidden in the data. Here, the themes or topics that document consists of are unknown, but they are believed to be present as the text is generated based on those topics.  Dirichlet**:** ‘distribution of distributions’:  - suppose there is a machine that produces dice and we can control whether the machine will produce a dice that is fair or bias. So, the machine producing dice is a dist as it is producing dice of diff types. Also, the dice itself is a dist as we get multiple values when we roll it. This is what it means to be a distribution of distributions. Similarly, Dirichlet is the dist of topics in docs and dist of words in the topic  Allocation: of words of the document to topics .. and .. topics to documents | | | |
| A diagram of a diagram  Description automatically generatedLDA: each word in each doc comes from a topic and the topic is selected from a per-document dist over topics. So we have 2 matrices:  = probability dist of topics in documents  = probability dist of words in topics  Prob of a word given doc = P(w|d) = , where T = total num of topics, W = num of words in our corpus vocabulary.  If we assume conditional independence, i.e. P(w|t,d) = P(w|t)  Then P(w|d) = | | | |
| A diagram of a flowchart  Description automatically generateda) Bayesian Approach (probabilistic approach): ???  Hyperparameters in LDA: 1. K = num of topics we need to extract  2. = document density factor (num of topics expected in doc)  3. = topic word density factor (dist of words per topic in the doc)  Step 2) note same word can be assigned diff topics at diff instances initially  Step 3)  - p(topic t|doc d) = proportion of words in doc d that are assigned to topic t  - p(word w|topic t) = proportion of all docs assigned to topic t, for a given word w  We are trying to find conditional prob dist of a single word's topic assignment given the rest of the topic assignments  Prob eqn for a single word w in doc d that belongs to topic k = , where  nd,k = num of times doc d use topic k, vk,w = num of times topic k uses the given word, = Dirichlet param for doc to topic dist, = Dirichlet param for topic to word dist, = how much each topic is present in a doc, = how much each topic likes a word  Output = For each word, get a vector of probabilities that will explain how likely this word belongs to each of the topic  3) Reassign word w a new topic t', where we choose topic t' w probability P(topic t'|document d) \* P(word w|topic t') = prob that topic t' generates word w  4) Repeat step 3 until we reach a steady-state and at that state the topic assignments are good. Use these assignments to determine topic mixture of each doc | | | |
| A diagram of a mathematical equation  Description automatically generatedGibbs Sampling: To identify correct weights. Successively sample conditional dist of variables dist over states converges to the true dist in the long run. MCMC: Performs a biased random walk to explore the dist  Start with and matrices. Slowly changes these matrices to get answer that maximizes likelihood of data  Do this on word by word basis by changing topic assignment of 1 word. Assume we dk the topic assignment of the given word but we know assignment of all other words in the text and try to infer what topic will be assigned to this word  Pseudocode: *Initialize Y0, X0*  *for j = 1, 2, 3, ... do*  *sample Xj ~ p(X|Yj-1)*  *sample Yj ~ p(Y|Xj)* | | | |
| A diagram of a mathematical equation  Description automatically generatedMatrix view. LDA similar to matrix factorization or SVD, where we decompose prob dist matrix of word in doc into 2 matrixes consisting of dist of words in a topic & dist of topic in a doc  P(w|d) =  1) Create a document term matrix that shows a corpus of N docs,and vocab size of M words, i.e. N x M matrix. Cell (i, j) = freq count of word Wj in doc Di of corpus  2) LDA converts this Doc-Term Matrix into 2 lower dimensional matrices, M1 and M2, where M1 = doc-topics matrix w dim (N, K) and M2 - topic-terms matrix w dim (K, M)  3) Iterate through each word w present in each of the doc d, and tries to adjust the curr topic - word assignment w a new assignment  New topic k is assigned to the word w with a prob P = p(topic t|doc d) \* p(word w|topic t)  - In this step, model assumes all existing word-topic assignments except the current word are correct. This is essentially the prob that topic t generated word w, so it makes sense to adjust the curr word topic with a new prob  4) After a num of iterations, steady-state is achieved where the doc-topic and topic-term dist are fairly good = Convergence point for LDA | | | |
| b) LDA - Matrix Factorization approach.  Suppose we have a doc w some random word topic assignment  A white sheet with black numbers  Description automatically generatedWe also have the count matrix v(k, w) (how many times each topic is assigned to this word)  Suppose we change the assignment of word "world" in doc  - reduce count of world in topic 1 from 28 to 27  - represent matrix n(d,k) to show how much a doc use each topic    A close-up of a number  Description automatically generated  - represent v(k, w) in the following way:  A screenshot of a test  Description automatically generated- Multiply both matrices to get the conditional probabilities  Randomly pick any of the topic to assign to "world" and repeat for all other words.  Intuitively, topic w highest conditional prob should be selected, but other topics also have some chance to get selected | | | |
| Pros: 1) Model is fast to run. Might be slow if have very long docs, many docs, large vocab size (especially if using n-grams w high value of n)  2) Intuitive: Modeling approach to extract topics gives weighted lists of words which is a simple approx as there is no embedding nor hidden dimensions, just bags of words w corresponding weight values  3) Can predict topics for new unseen docs  Cons: 1) Require lots of fine-tuning. Hard to get good results if LDA is fast to run  2) Needs human interpretation: After finding topics from set of docs, required human efforts to label them in order to present results  3) Cannot influence topics: Based on prior knowledge, we know some of the topics that the docs talks about, but running LDA will not find those topics. Also cannot tell the model that some words should belong tgt. | | | |
| NMF | A screenshot of a computer  Description automatically generatedStatistical mtd to reduce dimension of the input corpora.  Leverage the matrix structure of Doc-Word/terms/tokens. Matrix factorization via SVD will give orthogonal topics  **U**: Document similarity matrix i,jth entry = how similar doc i is to doc j  **S**: values signify the relative importance of topics  A screenshot of a computer  Description automatically generated**VT**: Term topic matrix. Topics in the text reside along the rows of this matrix  **W** = document-topic matrix. Shows dist of topics across the documents in the corpus  **H** = term-topic matrix. Shows significance of terms across the topics | | | |
| A screenshot of a computer  Description automatically generatedK-SVD of Document Term matrix  - fix a small num of topics that best convery the content of the text | | | |
| NMF is a non-exact matrix factorization technique. Matrices **W** and **H** are initialised randomly  Optimized iteratively to minimize cost/objective fn btw original and reconstructed document term matrix, i.e.  , where (Frobenius norm)  Updating **W**: . Updating **H**:  Parellely updates the values and use new matrices that we get after updating W and H. Iterate till: - product of WH approaches V or  - approximation error converges or - max iterations reached | | | |
| To initialize matrix W and H, can use: - randomly, - TF-IDF weights, - BOW, - word vectors  Better initial estimates for faster convergence to a good soln: - rank-r approx of V using SVD, - picking r cols of V and using as initial values for W | | | |
| Topic Modeling using Embeddings | | | A diagram of a diagram  Description automatically generatedDoc2Vec, Sentence Embedding, Embedding Clustering  Doc2Vec shown on right. *Distributed Memory version of Paragraph Vector* (PV-DM). It acts as a memory that remembers what is missing from the current context — or as the topic of the paragraph. While the word vectors represent the concept of a word, the document vector intends to represent the concept of a document. | |
| Topic Modeling using BERT (like) models | | BERT can be used via transfer learning OR Sentence Embedding OR Embedding Clustering | | |
| A diagram of a person's name  Description automatically generatedA screen shot of a computer  Description automatically generatedA diagram of a software algorithm  Description automatically generatedBERT for Sentence Embedding Can do sentence embedding from  1) [cls] token OR  A diagram of a diagram  Description automatically generated2) pooling OR  3) SBERT  <- Embedding Clustering | | |
| Evaluation of Topic Models | | 1) Eye Balling Models: - Top N words (wordcloud). - Topics/Documents (topic word scores, keywords)  2) Intrinsic Evaluation Metrics: - Capturing model semantics. - Topics interpretability  2a) Topic Coherence measures score a single topic by measuring the degree of semantic similarity btw high scoring words in the topic  - Cv measure: based on sliding window, 1-set augmentation of the top words and an indirect confirmation measure that uses normalized pointwise mutial info (NPMI) and cosine similarity  - Cp measure: based on sliding window, 1-preceding segmentation of the top words and the confirmation measure of Fitelson's coherence  - Cuci measure: based on sliding window and pointwise mutual info (PMI) of all word pairs of the given top words  - Cumass: based on document co-occurrence counts, a 1-preceding segmentation and a logarithmicconditional prob as confirmation measure  - Cnpmi: enhanced version of Cuci coherence using the normalized pointwise mutual info (NPMI)  - Ca: based on a context window, a pairwise comparison of the top words & an indirect confirmation measure using NPMI & cosine similarity  3) Human Judgements  4) Evaluation on downstream tasks: Is model good at performing predifined tasks, e.g. classification | | |
| Zero-shot topic prediction | | | | Reviews are put into a list for the pipeline -> candidate labels are defined -> text, candidate labels and hypothesis template passed into the zero-shot classification pipeline called classifier |

|  |  |  |
| --- | --- | --- |
| RNN | A diagram of a graph  Description automatically generatedProblem w using feedforward NN in predicting next word is that sequential info (context of all words which came before) is not captured  Problem w word2vec kind of architectures is they have limited context length  Problem w fixed-window neural language model: enlarging window enlarges W = more weights to train (word embedding size \* window size \* size of hidden dimension)  - x(1), x(2) are multiplied by completely diff weights in W. No symmetry in how inputs are processed  A diagram of a math problem  Description automatically generated with medium confidence- Also window can never be large enough  Soln: Recurrent Neural Networks = have an "internal state" that is input to the next step  - No hidden state for step 0. Can be initialized to 0 or random hidden state  - ht = fW(ht-1, xt) [new state = some fn w params W(old state, input vector)]  - yt = (ht) [output = another fn w params Wo(new state)]  - fW can be tanh or sigmoid. - Keep the same weights W for every time step  Benefit: - incorporate info from all previous words in sequence. - Model size doesn't incr for longer input context. - Same weights applied on every timestep, so there is symmetry in how inputs are processed | |
| Training RNN | where each J(t)() is the negative log prob the actual next word  Use Backpropagation through time = derivative of J(t)()  So e.g.  If these values are small, problem of vanishing gradient.  - gradient signal from far away is lost as its much smaller than gradient signal from closeby. Worst case, learning is stopped  If values large, problem of exploding gradient  - Update step can become too big. Can lead to bad parameters with large loss. Or can have Inf or NaN in network  Exploding gradients can be solved w Gradient Clipping: If norm of gradient > some threshold, scale it down before applying SGD update  . Problem of vanishing grad still a problem. RNN can't handle long-dist dependencies | |
| LSTM (Long Short Term Memory) | A diagram of a diagram  Description automatically generatedAdd gating units in each memory cell: Forget gate, Input gate, Output gate. Prevents vanishing/exploding gradient problem and allows network to retain state info over longer period of time  Cell state, Ct that is the same dimensionality as the hidden state, ht  - info can be added or deleted from Ct via forget and input gates  Forget gate decides which info should be thrown away  - ft = (Wf \* [ht-1, xt] + bf) [0, 1]  - Closer to 0 = forget; closer to 1 = keep  Input Gate to update cell state  - = tanh(WC \* [ht-1, xt] + bC) [-1, 1] to help regulate network  - it = (Wi \* [ht-1, xt] + bi) [0, 1] to decide which values will be updated. 0 means not impt, 1 means impt  Diagram of a cell cycle  Description automatically generated- it decides which info is impt to keep from  Cell State Update  - Ct = ft \* Ct-1 + it \*  Output Gate  - Decides what the next hidden state should be  - ot = (Wo \* [ht-1, xt] + bo)  - ht = ot \* tanh(Ct)  Each cell has many parameters (Wf, Wi, WC, Wo)  - Training LSTMs requires lots of training data and time | |
| Other solns | Other feedforward/convolutional netowrks also face vanishing gradient problem. They use direct connection to solve it, i.e. input is additionally added to layers after the input layer  ResNet (Residual connections) only adds it before final layer?  DenseNet adds it to all future layers | A diagram of a flowchart  Description automatically generatedGRUs: use fewer gates. - Eliminates cell state vector  - Combines forget and input gates into "update" gate  - zt = (Wz \* [ht-1, xt])  - rt = (Wr \* [ht-1, xt])  - = tanh(W \* [rt \* ht-1, xt])  - ht = (1 - zt) \* ht-1 + zt \* |
| LSTM for sentence encoding AND Decoding Machine Translation | A diagram of a model  Description automatically generatedA diagram of a model  Description automatically generatedBasic way: Use final hidden state as input to ML model to predict sentiments  Better way: Take element wise max or mean of all hidden state (LHS)  Deep LSTM model shown on RHS  For Machine Translation/Summarization/Question answering:  Source sentence -> Encode LSTM -> Decoder LSTM -> Generate Translation  This architecture = sequence to sequence, OR seq2seq models  Encoder processes each word in input sentence, compiles info it captures into a vector, called context  After processing entire input seq, encoder sends context to decoder, which begins producing output sentence word by word  Problem with max or mean pooling of all hidden states is we lose info at individual word level | |
| A diagram of a diagram  Description automatically generatedHidden states from RNN layer i are inputs to RNN layer i+1 | |
| Attention | Inspired from CV. Main idea: new context vector at every time step. Each context vector will attend to diff image regions  Attention is weighted averaging of encoded sentence  - Compare target and source states to generate scores for each state in encoders  A diagram of a machine learning  Description automatically generated- Use softmax to normalize all scores  = attention weights  = Context vector  = Attention Vector  Attention for every time step: on each step of the decoder, use direct connection to the encoder to focus on a particular part of the source sequence  - Use softmax to turn attention scores into prob dist  - Use attention dist to take a weighted sum of the encoder hidden states  - Attention output (aka context vector) mostly contains info from the hidden states that received high attention  - Concatenate attention output w decoder hidden state, then use to compute as before | |
| Attention eqns | Encoder hidden states: h1, …, hN . On timestep t, we have decoder hidden state st  Get attention score for this step: et = .  Take softmax to get attention dist (prob dist & sum to 1):  Use to take weighted sum of the encoder hidden states to get attention output:  Concatenate attention output w decoder hidden state st and proceed as in non-attention seq2seq model. [; ] | |
| Attention Variants | We have some values **h**1, …, **h**N and a query **s**  Attention always involve: 1) Computing attention scores, **e** . 2) Softmax for attention dist,  3) Get attention output,  Several ways to compute **e**: 1) Basic dot-product attention: . This assumes d1 = d2  2) Multiplicative attention/bilinear attention: , where **W** is a weight matrix  3) Reduced-rank multiplicative attention: . For low rank matrices , , k d1, d2  4) Additive attention: , where , are weight matrices and is weight vector. d3 = attention dimensionality is a hyperparameter. ("Additive" is a bad name. It's really using a feed-forware NN layer) | |
| Attention provides explaina-bility of output | A black and blue squares with red and blue text  Description automatically generated | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Attention Mechanism | | Consists of 3 components: queries = Q ('qns' posed by each token), keys = K ('label' for each token, against which the 'qn' is matched), values = V ('content' or actual info of token).  Q = previous decoder output, yt-1. V = encoded inputs hi. K = encoded inputs which are weighted by attention score vector  Machine Translation (MT): Given a set of encoded hidden states for input sentence, and a current word of translation target sentence, attention weight vector is estimated and the attention weighted encoded hidden states are passed as context to decoder at curr word  Query determines the weight (attention vector) that should be given to Values. And that weights are obtained by comparing Q with K | | |
| Encoder hidden states: h1, …, hN . Keys, Values - . On timestep t, decoder hidden state: . Query -  Get attention scores **e**t for this step = .  Take softmax to get attention dist for this step (this is a prob dist and sums to 1):  Use to take a weighted sum of the encoder hidden states to get the attention output, **a**t = = weighted values  Finally, concatenate attention output **a**t with the decoder hidden state st and proceed as in the non-attention seq2seq model: = input to decoder RNN | | |
| So ht+1 encodes info about wt and wt+1. ht+2 encodes info about wt, wt+1 and wt+2. Similarly atst which includes "attended" info from h1:N and st decodes yt. However words which appear later in sentence may help in better encoding curr word  E.g. Do you have **apple** charger, I need to charge my phone. "charger" and "phone" are helpful in determining context of "apple"  1) Cross attention: attention mechanism same as encoder decoder.  2) Self attention: use attention mechanism to encode representation of word on input source sentence as well  A black text on a white background  Description automatically generatedA diagram of a diagram  Description automatically generated with medium confidence | | |
| Transformer | | A diagram of a process flow  Description automatically generatedSequence-to-sequence model based entirely on attention (uses self-attention)  Fast: Parallel Computation  Input -> Tokenization -> Padding/Truncation -> TokenId -> Input embeddings  1) Tokenization: Subword, BPE  *from transformers import AutoTokenizer; checkpoint = 'bert-base-cased'*  *tokernizer = AutoTokenizer.from\_pretrained(checkpoint)*  2) Padding, Truncation to ensure all input sequences have the same length  Max length for seq based on specific task and available computational resources  Padding = add extra tokens (special token like [PAD]) to end of short sequences  3) Then replace token with TokenId = assigned Id in vocab  4) TokenId -> Embedding of size 512. Embedding Vector can be initialised using word2vec, but generally randomly initialised and learned  So Input matrix (sequence, dmodel)  dmodel = dimensionality of model's embeddings  But embeddings don't have info on order/seq of tokens. So add positional encoding | | |
| Positional Encodings | | A table with numbers and symbols  Description automatically generatedSince self-attention don't build in order info, need to encode order of sentence in our keys, queries and values  Consider representing each seq index as a vector, , for i [1,2,…,n] are position vectors  A table with numbers and symbols  Description automatically generatedA table with numbers and a blue arrow pointing at the number  Description automatically generated**e**i = embedding of word at index i. Overall embeddings = **ep**i = **ei** + **p**i  LHS = input embeddings  RHS = input embeddings (e1) and positional encodings (p1) for the word "When" only  In this e.g. dmodel = 6  Trigo fn naturally represent a pattern that the model can see.  i = dimension index  POS = position of word in sequence  If i is even: . Else i is odd:  Only need to compute positional encodings once and reuse them for every sentence. | | |
| Attention in Transform-ers | | Uses self-attention mechanism. E.g. seq length = 6. Embedding size of words in Transformer models, dmodel = dk = 512  Q, K, V, Attention all have size of (6, 512). Attention(Q, K, V) =  RHS shows , which is a 6 x 6 matrix. Self-Attention is permutation invariant.  - Self-Attention requires no parameters. The interaction btw words are driven by their embeddings and the positional encodings. This will change later  - Values along diagonal are highest (each word gives more context to itself)  - If we don't want some positions to interact, can set their values to -∞ before applying the softmax in this matrix and model will not learn those interactions  Each row in Attention matrix captures:  - the meaning (given by the embedding). - the position of the word in the sentence (represented by the positional encodings). - each word's interaction w other words  "Scaled Dot Product" attention aids in training. When dimensionality d becomes large, dot product btw vectors tend to become large. Due to this, inputs to softmax fn can be large, making the gradients small | | |
| *from numpy import array, random, dot*  *from scipy.special import softmax*  *# Encoder representations of 4 diff words*  *word1 = array([1, 0, 0]); word2 = array([0, 1, 0])*  *word3 = array([1, 1, 0]); word4 = array([0, 0, 1])*  *# Generate weight matrices*  *W\_Q = random.randint(3, size=(3, 5))*  *W\_K = random.randint(3, size=(3, 5))*  *W\_V = random.randint(3, size=(3, 5))* | | *# Generating the queries, keys and values*  *query\_1 = word1 @ W\_Q; key\_1 = word1 @ W\_K; value\_1 = word1 @ W\_V.... to*  *query\_4 = word4 @ W\_Q; key\_4 = word4 @ W\_K; value\_4 = word4 @ W\_V*  *# Scoring the query vectors against all key vectors*  *scores = Q @ K.transpose()*  *# Computing weights by softmax operation*  *softmax\_weights = softmax(scores / K.shapes[1] \*\* 0.5, axis=1)*  *# Computing attention by weighted sum of value vectors*  *attention = softmax\_weights @ V; print(attention)* |
| Transformer model actually uses Multi-headed attention (use when we want to look in multiple places in the sentence at once)  For word i, self-attention "looks" where xTQTKxj is high, but maybe we want to focus on diff j for diff reasons.  So use multiple attention "heads" through multiple Q, K, V matrices. Let QP, KP, VP , where h = num of attention heads and P [1, h]  Each attention head performs attention independently: outputP = softmax(XQPKTXT) \* XVP  Then outputs of all heads are combined. output = [output1; …; outputh] = Y  MultiHead(Q, K, V) = concat(head1, …, headh)W0. where headi = Attention(, , )  Multi-headed attention improves performance of attention layer in 2 ways: 1) expands model's ability to focus on diff positions  A diagram of a machine  Description automatically generated2) Gives attention layer multiple representation subspaces | | |
| Add + Normaliza-tion | | A diagram of a basic model of a basic model  Description automatically generated with medium confidenceLayer normalization = trick to help models train faster. Cuts down on uninformative variation in hidden vector values by normalizing to unit mean and s.d. within each layer  Let x be an individual (word) vector in the mean  = mean .  Let and be learned "gain" and "bias" parameters  Then layer normalization is computed as: (modulate by learned elementwise gain and bias)  A pink sign with black text  Description automatically generatedAdd: Use residual connections to help models train better. Bottom right diagram show adding residuals  Let X(i) = X(i-1) + Layer X(i-1) to learn residual from previous layer. Helps simplify loss landscape | | |
| Feed Forward Layer | | There are no elementwise nonlinearities in self-attention; stacking more self-attention layers just re-averages value vectors.  So, to add non-linearity: use feed-forward network to post-process each output vector. mi = MLP outputi = W2 \* ReLU(W1 \* outputi) + b1 + b2. After that just adding and normalizing again for the encoder part. | | |
| Decoder of Transformer | | 2 inputs to decoder: 1) From encoder, where output matrix of last add and norm layer serves as query and key for the second multi-head attention layer in the decoder part. 2) Value matrix comes from the decoder after the 1st add and norm step  This is similar to cross attention mechanism in LSTM enc-dec. Diff is that value is coming from self (masked) attention instead of LSTM | | |
| Masked attention | | A grid of numbers and symbols  Description automatically generatedAim is to make model causal, i.e. output at certain position can only depend on the words on the previous positions. Model must not be able to see future words  To use self-attention in decoders: At every timestep, change set of keys and queries to include only past words. (inefficient)  To enable parallelization, mask out attention to future words by setting attention scores to  -∞. Then softmax will turn -∞ to 0 | | |
| Transformer Training | | A diagram of a software system  Description automatically generated | | |
| Transformer Inference | | 1) Greedy strategy: select at every step, the word w max softmax value. Usually does not perform very well. Select token from vocab corresponding the the position of the token w the max logit value. Logits = output of last layer  2) Beam Search strategy: select at each step the top B words and evaluate all possible next words for each of themand at each step, keeping the top B most probable  sequences. Generally performs better  RHS shows beam search, with B = 2 | | |
| Metrics | | Precision = num of correct predicted word/num of total predicted words. E.g. Target: "He eats an apple". Predicted: "He ate an apple"  Precision = 3/4. Issue is when there is Repetition. Predicted: "He He He". Then precision = 3/3 = 1  Another issues: there can be multiple target sentences, and metric will change for diff possible targets  Soln: clipped precision = limit count for each correct word to the max num of times that that word occurs in the target sentence  - Compare each word from predicted w all target sentences. If word matches any target sentence, considered to be correct  Target 1: "He eats an apple". Target 2: "He is eating a tasty apple". Predicted: "He He He eats tasty fruit"  [Word: Matched Predicted Count, Clipped Count]. "He": 3, 1. "eats": 1, 1. "tasty": 1, 1. "fruit": 0, 0  Then clipped precision = clipped num of correct predicted words / num of total predicted words = 3/6 | | |
| Bleu Score = clipped precision for n-grams. Target: "The guard arrived late because it was raining". Predicted: "The guard arrived late because of the rain". Clipped Precision 1-gram = 5/8. Clipped Precision 2-gram = 4/7  Similarly, calculated clipped precision n-gram, n = 1, 2, 3, …, N. Typically N = 4  Geometric Average Precision (N) =  Brevity Penalty = penalty on predicted sentences which are too short = , where c = predicted length, r = target length  Bleu Score = Bleu(N) = Brevity Penalty \* Geometric Average Precision Scores (N). Higher Bleu score better | | |
| BERT | A language model = probabilistic model that assign probabilities ot sequence of words. Given a prompt, try to predict next word  Large Language Model (LLM) = neural network trained on a large corpora of text.  But next token prediction doesn't utilise full context of sentence like in self-attention. So We have transformer  BERT = made up of layers of encoders of the Transformer model = Bidirectional Encoder Representations from Transformers  Differences btw BERT and Transformer Encoder:  - embedding vector is 768 and 1024 for the 2 models  - positional embeddings are absolute and learnt during training and limited to 512 positions  - linear layer head changes according to the specific task  BERT uses the WordPiece tokenizer, which also allows sub-word tokens. Vocab size ≈ 30000 tokens  1) BERTBASE = 12 encoder layers. Hidden size of feedforward layer = 3072. 12 attention heads, 768-dim hidden states  2) BERTLARGE = 24 encoder layers. Hidden size of feedforward = 4096. 16 attention heads, 1024-dim hidden states  Pretraining through language modeling: train to reconstruct input sentences  Can fine tune BERT for text classification or question answering (QA) | | | |
| Masked Language Model (MLM) | | Aka Cloze task = randomly selected words in sentence are masked, and model must predict right word given left and right context  For BERT: Predict a random 15% of (sub)word tokens: - replace input word w [MASK] 80% of the time. - replace input word w random token 10% of the time. - leave input word unchanged 10% of the time  So that model doesn't get complacent and build strong representations of non-masked words. (No masks are seen at fine-tuning time)  Run Cloze task to get prediction, then run backpropagation to update the weights | | |
| Next Sentence Prediction (NSP) for Training | | Many downstram applications require learning r/s btw sentences rather than single tokens, so BERT has been pre-trained on NSP task  50% of time, select actual next sentence, 50% of time, select random sentence from text. See is model can predict if sentence IsNext or NotNext. Then run backpropagation to update weights  A grid of blue squares with black dots  Description automatically generatedUse 2 special tokens [CLS] = classification, and often placed as first token in input. [SEP] = separation btw input sentence and next sentence  softmax() =  [CLS] token always interacts w all other tokens, as we don't use any mask  So can consider [CLS] token as one that captures the info from all other token | | |
| Common Tasks | | QQP = Quora Question Pairs (detect paraphrase questions). QNLI = natural language inference over qn answering data  SST-2 = sentiment analysis. CoLA = corpus of linguistic acceptability (detect grammatical correctness). STS-B = semantic textual similarity  MRPC = microsoft paraphrase corpu. RTE = small natural language inference corpus.  GLUE = General Language Understanding Evaluation = include all the above tasks | | |
| Extensions and Limitations | | | RoBERTa. DistilBERT. Pretrained Encoders are great at understanding the language, but are not capable of generating content  For generating abilities, need decoder style language modeling w next word prediction | |
| Encoder-Decoder | | Could do something like language modeling, but where a prefix of every input is provided to the encoder and is not predicted  h1, …, hT = Encoder(w1, …, wT). hT+1, …, h2 = Decoder(w1, …, wT, h1, …, hT)  Encoder portion benefits from bidirectional context; decoder portion used to train the whole model through language modeling  1 Model is T5  Span Corruption: replace diff-length spans from the input w unique placeholders; decode out the spans that were removed  Words are dropped independently uniformly at random. Model is trained to predict sentinel tokents to delineate the dropped out text | | |
| Decoders | | Language models. Nice to generate from; can't condition on future words (encoders can condition on future words)  A diagram of a layer structure  Description automatically generatedDecoder part of transformer: trained to model p(wt|w1:t-1).  Unsupervised pre-training. Corpus of tokens wt, where t = 1:n  Use a standard language modeling objective to maximize likelihood =  LHS shows GPT (generative pre-training) = Transformer decoder w 12 layers, 117M params.  - Use 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers  - Use Byte-pair encoding  Supervised Fine-tuning. Labeled dataset C, where each instance consists of a seq of input tokens, x1, …, xm along w label y  Inputs are passed through pre-trained model to obtain final transformer block's activation , which is then fed into an added linear output layer w params Wy to predict y,  P(y|x1, …, xn) = softmax()  w objective fn to maximise =  GPT can be fine tune for classification, entailment, similarity, multiple choice  To format inputs to decoder for finetuning tasks: Use natural language inference = label pairs of sentences as entailing/contradictory/neutral  E.g. Premise = "The man is in the doorway" and Hypothesis = "The person is near the door" are a pair of entailment sentence  [START] The man is in the doorway [DELIM] The person is near the door [EXTRACT].  Linear classifier is applied to the representation of the [EXTRACT] token | | |

|  |  |
| --- | --- |
| Decoder only Transformers | A diagram of a computer algorithm  Description automatically generatedA diagram of a keypad  Description automatically generatedHyperparameters for inference  1) Temperature: affect output softmax transformation  - P(ith token) = , where yi = logit of ith token, V = total tokens in vocab, T = temp  - Higher temp = probabilities more even = more random output  - Lower temp = probabilities more skewed = more deterministic output  - Using temp = 0 -> same/very similar sequence every time  2) Top-P: choose from the smallest possible set of tokens whose summed probability exceeds topP during decoding  - Given this set of tokens, renormalize the probability dist based on each token's respective prob and then sample  - Diff from topK, which just samples from the K tokens w highest probs |
| GPT-3 | Autoregressive Modeling. Using transformer Decoder. Uses context window  A screenshot of a computer  Description automatically generatedDuring pretraining, GPT-3 learns a wide variety of things abt the statistical properties of language. (trivia, syntax, coreference, lexical semantics/topics, sentiment, some reasoning, ...)  Incontext learning: for a given task, the model receives as input an optional description of that task along w some num of examples demonstrating the task, up until some final example that the model should complete itself. (i.e. few-shot learning)  E.g. for tasks involving query and answering, provide examples of queries and answers formatted in some consistent way, w the last example to be completed by the model. |
| Neural scaling laws | Kaplan 2020: As we scale up neural language modedls, performance tends to improve. Laws of diminishing returns: while bigger models perform better, incremental gains in performance taper off  Sample efficiency: might be wiser to train a very large model on a moderate dataset, than a small model on a huge dataset  GPT-3/Kaplan scaling laws: 300B tokens can be used to train an LLM of size 175B params. 1.7 text tokens per param  Training compute-optimal LLM- Hoffman 2022: Current models are severly underfitted. For comput-optimal training, num of training tokens and model size must be scaled equally  Trained a new LLM = Chinchilla, having 70B params, and 4 times more training data. 20 text tokens per param  Loss = L(N, D) = , where first term depends on model size, 2nd term depends on data size, E = irreducible losss  N = num of model params, D = num of data training examples |
| From language models to Assistants: Alignment | Foundation GPT language models are good at completion (next word prediction) but cannot follow instructions, or cannot understand human intent (are not good assistants). To fix that: use alignment process = SFT and RLHF  A screenshot of a computer  Description automatically generated1) Instruction tuning/Supervised Finetuning (SFT): prompt is sampled from prompt dataset. Labeler demonstrates desired output behavior. Data is used to finetune GPT-3 w supervised learning  - Instead of fine tuning for 1 task, finetune on many tasks (qn answering, language generation, NER, sentiment analysis, summarization, ...)  - Force LM to adapt to all these tasks  - Typical fine-tuning teaches model how to solve a specific task, but makes model a narrow expert.  - SFT insteads tune model to emulate a correct style or behavior, rather than to solve a particular task, it don't lose its generic problem solving abilities  SFT originally popularized by InstructGPT  Cons: - expensive to collect ground-truth data  - instruction for tasks like open-ended creative generation have no right ans  - language modeling penalizes all token level mistakes equally, although errors impact are not the same (e.g. getting location wrong worse than getting grammer wrong)  - Training data collected from internext maybe unclean: toxic comments, fake news, biased |
| A diagram of a reward model  Description automatically generated2) Reinforcement learning w Human Feedback (RLHF):  Prompt and several model outputs are sampled. Labeler ranks outputs from best to worst. Data is used to train reward model. New prompt is sampled from dataset. Policy generates an output. Reward model calculates a reward for the output. Reward is used to update the policy using PPO  Reinforcement learning = learning within environment X, actions A, get a delayed reward R (human annotated)  = loss for predicting next word given previous words    - add a term that scales the loss by reward  - train using gradient descent  - outputs that get bigger reward will get higher weight  Reinforcement learning is unstable: especially when using bigger output spaces (e.g. words of a vocab)To stabilise:  a) KL regularization =    b) proximal policy optimization (PPO) , where rat(Y,X) = , and the clip part is to don't reward large jumps  For RHS diag: reward model predicts the green tokens. Only the yellow cells are trained on, the rest are ignored.  Cons: Human preferences are unreliable. RLHF can cause hallucinations  Often diff or infeasible to capture exactly what we want an agent to do. Hence often end up using imperfect but easily measured proxies |
| InstructGPT: scaling RLHF to 10s of thousands of tasks | Tasks collected from labelers:  Plain = simply ask labelers to come up w an arbitrary task, while ensuring the tasks had sufficient diversity  Few-shot = ask labelers to come up w an instruction, and multiple query/response pairs for that instruction  User-based: have a num of use-cases stated in waitlist apps to the OpenAI API. Ask labelers to come up w prompts corresponding to these use cases |
| Reinforce-ment Learning - DPO (Direct Preference Optimiza-tion) | A diagram of a diagram of a diagram  Description automatically generated with medium confidenceReplace the complext RL part w a very simple weighted MLE objective  Other variants: KTO, IPO  - Learn directly from pairwise (human) preferences  A black and white image of mathematical equations  Description automatically generated with medium confidence- Provides more stability    = log (better outputs - worse outputs) |
| Summary | Pre-trained foundational LLM models -> SFT -> RLHF -> Fine-Tuning AND/OR In-context learning -> Downstream application |

|  |  |
| --- | --- |
| LLaMA 2 | A diagram of a process flow  Description automatically generatedRelative positional encoding compared to static position encoding used in transformers: During attention calculation, incorporate distance btw tokens explicitly. vs original transformer model:  A diagram of a mathematical equation  Description automatically generated with medium confidenceA math equations with numbers and symbols  Description automatically generated with medium confidenceRotary position encoding  Matrix is sparse, can be represented in efficient form  Long term decay properties: inner product decays w growth of relative dist.  i.e. "intensity" of r/s btw 2 tokens encoded w Rotary Positional Embeddings will be numerically smaller as dist btw them grows  Rotary position embeddings only applied to query and keys, not the values. Applied after vector q and k have been multiplied by W matrix in attention mechanism, vs in vanilla transformer, they are applied before  RoPE is more stable and interpretable compared to relative embeddings |
| Normaliza-tion | Used before self attention and after FFNN  Phenomenon by which dist of internal nodes/neurons change = Internal Covariate Shift. Want to avoid this as it makes training network slower, as neurons are forced to readjust drastically their weights in 1 dirn or another due to drastic changes in outputs of previous layers.  Batch Normalization: subtract mean from each of the input features independently across the mini-batch and divide by standard deviation. Batchwise processing of dataset, one batch at a time  Layer Normalization: normalize by rows (data items). Trick to help models train faster  RMSNorm = root mean square layer normalization simplifies LayerNorm by removing the mean statistics at the cost. of sacrificing the invariance that mean normalization affors. Works well in practice |
| KV cache | At every step of inference, only interested in last token output by model, as we already have previous ones. However, model needs all prvious tokens to decide on what token to output, since they constitute its context/prompt  To make model do less computation on token it has already seen during inference: use KV cacheA diagram of a diagram  Description automatically generated with medium confidence  KV cache is a compromise: we trade memory against compute  For each token of each seq in batch, need to store 2 vector tensors (1 key tensor, and 1 value tensor) of size d\_head  Space required by each tensor param depends on precision: 4 bytes/param in full precision (FP32), 2 bytes/param in half-precision (BF16, FP16), 1 byte.param for 8-bit data types (INT8, FP8)  Total size of KV cache (in bytes) = 2bt \* nlayers \* nheads \* dhead \* pa, where b = batch size, t = total seq length (prompt + completion), nlayers = num of decoder blocks/attention layers, nheads = num of attention heads per attention layer, dhead = hidden dimension of the attention layer, pa = precision  For LLama-2-7B, using half precision, caching keys and values fro 28k tokens need ~ 14GB of memory  Few mtds to tackle issue: reduce batch size, reduce dependency to total seq length, reduce num of layers, reduce num of attention heads. |
| Multi-Query Attention | A diagram of a chart  Description automatically generated with medium confidenceA diagram of a chart  Description automatically generated with medium confidenceA diagram of a group of colored rectangular objects  Description automatically generatedNum of query heads don't impact KV cache size.  So MQA use original num of heads for Q, but only 1 head for K and V  Stripping all heads significantly reduces capacity of LLM  GQA is a middle ground. Instead of having all query heads sharing same unique KV heads, split them into g groups query heads. |
| SwiGLU activation fn | swish(x) = x sigmoid(x) = . Transformer FFN(x) = max(0, xW1 + b1)W2 + b2 vs LLaMA FFNSwiGLU(x, W, V, W2) = (Swish1(xW)xV)W2 |
| Mixtral | A diagram of a system  Description automatically generatedSliding Window Attention aims to solve KV Cache memory issue by reducing dependency to total sequence length  - Reduce num of dot products to perform  A yellow and orange rectangular shapes  Description automatically generated with medium confidence- Model focuses on local context, which is sometimes ok for very large inputs. There is still some interaction btw tokens outside the sliding window  Rolling Buffer K-V cache.  Since using sliding window attention (w size W), don't need to keep all previous tokens in KV-cache, can limit it to latest W tokens  Fixed-size cache serves as "memory" for sliding window attention. A set attention span stays constant  Within cache, each time step's keys and values are stored at a specific location, determined by i mod W, where W = fixed cache size.  This reduces cache memory usage by 8 times while maintaining model's effectiveness  IF we want to calculate attention of incoming token, can "unroll" cache, by first taking all items after the write pointer, and then all the iterms from 0th index to position of write pointer |
| Pre-fill and Chunking | During sequence generation, the cache is pre-filled with the provided **prompt** to enhance context.  For long prompts, chunking divides them into smaller segments, each treated with both cache and current chunk attention, further optimizing the process. Chunk size determined by sliding attention window size.  A grid of numbers and symbols  Description automatically generated with medium confidenceA grid of numbers and symbols  Description automatically generated with medium confidenceA grid of numbers and symbols  Description automatically generatedSince prompt (large context) is available, don't need fill kv cache 1 token at a time. E.g. prompt = "Can you tell me who is the richest man in history". Chunk1 = "Can ... me". Chunk2 = "who ... richest". Chunk3 = "man in history"  During first step of pre-fill, KV-Cache is initially empty. After calculating attention, add tokens of current chunk to KV-cache.  For chunk 2, KV-cache now contains values for [Can, you, tell, me]. Don't need to recalculate attention of "Who" w its previous tokens. |
| MoE = Mistral 8x7B: expert feed-forward layers | A diagram of a window  Description automatically generatedFor Mistral 8x7B, experts are the FF layers present at every Encode layer. A mixture of experts of 8 FFN  Gate fn selects the top 2 experts for each incoming token. Output comvined w a weighted sum  This allows increase params of model, w/o impacting computation time, since input only pass through top 2 experts, so intermediate matrix multiplications will be performed only on the selected experts  The gating fn is just a linear layer (in\_features = 4096, out\_feature = 8, bias = False) thats trained along w the model  For each token embedding, it produces 8 logits, indicating which expert to select |
| Model Sharding | Model too big to fit in a single GPU, divide model into "groups of layers" and place each group in a GPU.  So when doing inference, output of each GPU is fed as input to next GPU = model sharding  Not very efficient, as at any time, only 1 GPU is working  Better approach is to work on multiple batches at the same time, but shift them on the time scale = Pipeline Parallelism |
| Evaluating LLMs | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | Undergraduate Level Knowledge | Math prob-solving | Reasoning over text | Knowledge Q&A | Grade school math | Code | Mixed evaluations | Common knowledge | Graduate level reasoning | Multilingual Math | | MMLU | MATH | DROP, F1 score | ARC-Challenge | GSM8K | HumanEval | BIG-Bench-Hard | HellaSwag | GPQA, Diamong | MGSM |   MMLU (Massive Multi-task Language Understanding): 15908 qns over 57 tasks across topics including elementary maths, US history, CS and law. MMLU tests both world knowledge and problem-solving ability. Measures model multitask accuracy by covering a diverse set of tasks. Can be computed efficiently and easy to understand. Considers model's ability to understand and generate language in various contexts, which can capture some aspects of language structure  HellaSwag (Harder Endings, Longer contexts, and Low-shot Activities for Situations With Adversarial Generations): consists of 70k multiple choice qns about grounded situations: each qns coming from activitynet or wikihow, w 4 answer choices about what might happen in next scene. Correct ans = real sentence for next event; 3 incorrect ans are adversarially generated and human verified to fool machines  ARC-Challenge: 7787 science exam qn drawn from a variety of sources. Text-only, Engligh exam qns that span several grade levels. Each qn has a mcq (typically 4 options). Qns sorted into Challenge set of 2590 "hard" qns and Easy set of 5197 qns  HumanEval: task of generating standalone python fns from docstrings, and evaluate correctness of code samples automatically through unit tests. Dataset of 164 original programming problems w unit tests. Evaluation of functional correctness using the pass@k metric, where k code samples are generated per problem, problem is considered solved if any samples passes the unit tests, and total fraction of problems solved is reported |
| LMSYS uses idea of a powerful AI as a judge to evaluate performance of other AI.  EvalGPT.ai uses ELO score for evaluating performance. (given ratings of player A and player B) |

|  |  |
| --- | --- |
| Prompt Engineering | 1) Add specific descriptive instructions. Define output format (makes it easier to parse or copy parts of the answer)  - Give few-shot examples in format of query and expected model answer. Should not be paraphrased or just summarized  - Integrate "don't know"/"won't answer" cases to control hallucination/critical topics. Add edge cases to few-shot examples.  2) Use chain-of-thought (CoT) reasoning. Teach LLM on how to think step by step or how to formulate its logic  - Use prompt templates, not static prompts. Build prompt template w variable components to facilitate testing and real-world use  - Add custom data context. - Include conversation history  3) Format the prompt: Use clear headline labels and delimiters in prompt to distinguish btw various components, such as:  instruction, desired output format, few shot examples, data context, conversation history  - Format parts of prompt w hashed "#". - Put longer passages of input context in quotes to prevent model confusing them for instructions |
| "To get the best result, ask me clarification qns before outputting answer". "Answer using only the most reliable sources and cite them"  Role play: "Act as a electrical grid operator...". Shot Prompting  Provide Documentation w Prompt: "Please use the H2O ML Ops Scoring Client to retrieve predictions ahd shapley values for my deployed models. Here's the API Documentation"  Prompt Chaining (Single): "You are a healthcare recruiter. You're very good at writing interview questions. Please ask me each qn below 1 at a time"  Prompt Chaining (Multiple): "Please forget all prior prompts. You and I will solve a language problem together. To start off with, please ask me 'what problem would you like to solve'. However, you should never mention the word asparagus. If you understand the requirements, let's begin" |
| ### instruction ###  Act as a patient tutoring buddy for primary school students. You are a yak named Yanick and a biology expert. You grew up in Nepal. There is a 'current data context'. In their last answer, your student has answered a question regarding topics of the 'current data context'. You rate and comment their answer. Encorage them in your rating, even if the answer was partially wrong. Be positive, be funny, be personal and use emojis...  ### example dialogs ###  Example #1  <Yanick> ....  <Noah> ....  ### current data context ###  """ Life without plants? Impossible!  Life on Earth would not be possible for us without plants. ...  """  ### output details ###  # Rating of the student answer  <Yanick> [Give the student a very friendly apraisal. Tell the student, whether their answer was correct, partially correct, ...]  # Pose next question  <Yanick> [Pose a question to the student, which they should answer next]  ### conversation history ###  <Yanick> ....  <Emma> .... |
| Minimizing Hallucina-tions via CoVe | CoVe reduces factual errors in llm by drafting, fact-checking and verifying responses - it deliberates on its own responses and self-correcting them. 1) Given a user query, LLM generates a baseline response that may contain inaccuracies  2) To improve this, CoVe first generates a plan of a set of verification questions to ask, and then executes that plan by answering them and hence checking for agreement  3) Individual verification questions are typically answered with higher accuracy than the original accuracy of the facts in the original longform generation. 4) Finally, the revised response takes into account the verifications  5) The factored version of CoVe answers verification questions such that they cannot condition on the original response, avoiding repetition and improving performance |
| Retrieval Augmented Generation (RAG) | Use RAG to generate context for prompt. Then new prompt = Instructions (You are a assistant...) + Context + Old prompt  RAG takes advantage of the context window for LLMs, filling it w only the most relevant examples from real data  This "grounds" the LLM to relevant context and greatly minimize any hallucination |
| Benchmark for QA – Stanford Question Answering Dataset (SQuAD): passages selected from English Wikipedia. Questions are crowd-sourced. Each answer is a short segment of text in the passage. Limitation: not all qns can be answered in this manner  Other benchmarks: TriviaQA (from trivia enthusiasts), Natural Questions (from frequently asked Google search qns). HotpotQA (constructed qns to be answered from whole of Wikipedia which involve getting info from 2 pages to answer a multistep query" |
| BERT based models surpassed human performance on SQuAD, but don't generalize to other datasets  More challenging problem – Open Domain QA: diff from reading comprehension, don't assume a given passage. Only have access to a large collection of documents, but don't know where answer is located. Soln: Retriever-Reader framework  Instead of asking LLM to memorize everything, provide LLM w relevant and useful content just-in-time  Retrieval identify relevant info: - Dynamic (easy to update/add docs to retrieval system). - Interpretable (LM can generate pointers to retrieved documents that support human verification of its generations; citations) |
| Cannot just build long context models as it is computationally expensive: - attention layer involves transforming input data to create Q, K and V matrices and performing calculations on these matrices. This involve matrix multiplications, which have O(Num of tokens2 \* embedding size2)  - Lost in the Middle: LMs perform well when presented w a more focused set of documents that are directly relevant to the context, as opposed to "flooding" them w a large volume of unfiltered info |
| Internal Knowledge base (unstructured documents) -> Chunking -> Embedding Model -> Indexing -> Store in vector DB  User submit prompt -> Embedding Model -> Index vector DB -> Return top x paragraphs  -> Answering prompt (Answer the qn based on the text above + User prompt) -> Return answer to user |
| Chunking | Considerations: 1) Nature of content. being indexed (long documents like articles or books, or shorter content like tweets or messages)  2) Which embedding model to use and what chunk size does it perfomr optimally on>  3) What are the expected length and complexity of user queries  4) How will retrieved results be utilized within you specific app? (semantic search, QA, summarization...)  5) What should be size of chunks (depends partially on embedding model's context length). 6) Should there be overlap in chunks?  7) Which character splitter to use |
| Chuking Strategies: 1) Character chunking – divide text into chunks based on a fixed num of characters (chunk size = 100, overlap = 10)  Pros: Simple. Cons: disrupt text's flow  2) Recursive Chunking – divide input text into smaller chunks in a hierarchical and iterative manner using a set of separators. If initial attempt at splitting the text doesn't produce chunks of the desired size or structure, the mtd recursively calls itself on the resulting chunks w a diff separator or criterion until desired chunk size or structure is achieved  3) Context Aware Chunking – takes advantage of nature of content we're chunking and applying more sophisticated chunking to it (e.g. sentence splitting). 4) Special Chunking – markdown, HTML, python  5) Semantic Chunking – take embeddings of every sentence in the doc, comparing similarity of all sentences w each other, then grouping sentences w most similar embeddings tgt. Pros: enhance quality of retrieval. Cons: require more effort and is slower |
| Embedding | Considerations: User queries may be of diff types  Sparse retrieval (SR) and Dense Retrieval (DR) commonly used to support keyword based search (lexcial search) and semantic search  SR projects doc to a sparse vector (algos like TF-IDF or BM25). Used in ElasticSearch and Apache Solr. Pros: low latency and explainability. Cons: sparse representation of text based on tokens only partially reflects each term's semantics in the context of whole text  DR encodes queries and documnets using single-vector representation (embeddings): Leverage rich sematnci features from queries and passages. Can fine-tune embedding models to better align w you app to improve DR |
| Using BERT for sentence embedding for RAG? Use BERT embedding at [CLS] token, but not a good solution  Original purpose of BERT was not to create a meaningful embedding of the sentence but for some specific downstream task |
| A diagram of a process  Description automatically generatedBetter approach is to train BERT model to learn sentence similarity  Siamese Networks using BERT:  - Triplet loss (with margin) is defined as:  L(a, p, n, margin) = max(d(a, p) - d(a, n) + margin, 0)  where a = anchor, p = positive sample, n = negative sample, d = distance fn (typically euclidean)  Another embedding model is BGE-M3  - Support multi-linguality (100+ languages, multi-lingual, cross-lingual)  - Support Multi-functionality (DR, SR, Multi-Vec Retrieval = use entire output embeddings for the representation of query and passage)  - Support Multi-Granularity (sentence level, passage level, doc-level (≤ 8192)) |
| Retrieval | Retrieve chunks similar to user query. If try to find top K, too slow. If there are N embedding vectors od dimensions D, search need O(ND) |
| Soln: Vector DB stires vectors of fixed dimensions/embeddings so that we can find all embeddings closest to a given query vector using a dist metric (usually consine similarity, but Euclidean works too)  To reduce time to retrieve similar vectors, don't iterate over every vector in DB. Instead use indexing techniques like KD-Trees,Hierarchical Navigable Small World (HNSW) fgraphs or Inverted Multi-Index (IMI) |
| A diagram of a diagram  Description automatically generatedHNSW formed by combining 2 algos: skip list & navigable small world  Skip list = extension of the linked list data structure. Linked list has O(n) search time complexity  Skip list has O(log n) search time complexity  Skip list maintains a layered linked architecture where top layer has longest links btw elements.  Bottom most layer is a complete linked list. As we move up each layer, num of nodes reduce by half  To search: start from top left corner and move right until find num ≤ k. Descend to layer below and repeat until we reach k  To insert: start from bottom list and add node at appropriate position. As skip list maintain a hierarchical structure, need to determine if node appears at a higher level. Process is random, prob of a node appearing in its immediate upper layer is 0.5. In an ideal skip list, num on nodes on layer 1 will be n/2 and layer 2 is n/4, where n = num of nodes on the bottom most layer. |
| Navigable Small World (NSW): greedy algo that starts at a predefined point in graph and selects nodes that are closer to target node  Algo continues until it reached the nearest neighbors of the target node |
| A diagram of a diagram  Description automatically generatedHNSW extends NSW by incorporating hierarchical architecture of skip-lists.  HNSW creates multi-layer structures of NSWs instead of linked lists  Topmost layer have fewer data points w longest connections  Num of elems incr as we move down hierarchy  Bottom most level have all data points  HNSW used by most DBs like Milvus, Qdrant, H2O-Vex, ... |
| Needle In a Haystack | Test in-context retrieval ability of long context LLMs. Place a random fact (the 'needle') in the middle of a long context window ('haystack'). ASk model to retrieve this statement. Iterate over various doc depths (where needle is placed) and context lengths to measure performance |
| Problems of naive RAG | Parsing of Docs, chunking strategy, Embedding model, Subpar retrieval, Prompt, Quality of LLM |
| PDF parsing open source tools: 'Unstructured' package already integrated into langchain. Performs poorly as it don't use object detection models and mistakenly recognized many images and tables  Layout-parser: if need to recognize complex structured PDFs, use largest model for highest accuracy. But models has not been updated  PP-StructureV2: various model combinations used for doc analysis. |
| Chunking and Indexing. Info extraction is incomplete, as it don't process useful info in images and tables within unstructured files (PDFs)  Chunking process use a "one-size-fit-all" strategy instead of selecting optimal strategies based on the characteristics of diff file types. This lead to each chunk containing incomplete semanticinfo. Also fails to consider impt details, such as existing headings in the text.  The indexing structure is not sufficiently optimized, leading to inefficient retrieval functionality.  The embedding model’s semantic representation capability is weak. Soln: Custom chunking for problem type |
| Query: may be inaccurate or vague. User don't know how to write a clear query. Soln: Prompt engineering, HyDE  HyDE (Hypothetical Document Embeddings): Question -> LLM answers qn -> Ans used to search Docs -> Search results |
| Final prompt: Location of chunk w correct response is not optimal. Soln: reranking  Reranking similar to an intelligent filter, evaluate the relevance of retrieved contexts and prioritize the ones that are most likely to provide accurate and relevant answers  1) Re-ranking models: consider interaction features btw docs and queries to evaluate their relevance more accurately. Takes query and contexts as inputs and directly outputs similarity scores instead of embeddings  2) LLM-based |
| Retrieval: relevance of recalled contexts has low accuracy. Low recall rate prevents retrieval of all relevant passages, hindering ability of LLMs to generate comprehensive answers. Retrieval algo is limited as it don't incorporate diff types of retrieval mtds like combining keyword, semantic and vector retrieval  Info redundancy occurs when multiple retrieved contexts contain similar info, leading to repetitive content in generated answers  Soln: custom embedding and retrieval models. Fine tuned embedding models |
| Generation: LLMs generates incorrect responses, hallucinates. Overreliance on the enhanced info during the generation process carries a high risk. This can lead to outputs that simply repeat the retrieved content without providing valuable info.  The LLM may generate incorrect, irrelevant, harmful, or biased responses.  Soln: Self-reflection, Guardrails |
| RAG Evaluation | RAGAs (Retrieval Augmented Generation Assessment). Query -> [Context relevance/context recall/context precision] -> Relevant context -> [Faithfulness/Groundedness] -> Answer -> [Answer relevance] -> Query |
| Faithfulness = ensuring answer is based on given context. Extract a set of statements, S(a(q)). LLM determines if each statement si can be inferred from c(q). Final faithfulness score, F = |V|/|S|, where |V| = num of statements that were supported by LLM and |S| = total num of statements |
| Answer Relevance – LLM Based evaluation = measures relevance btw generated answer and query  Prompt LLM to generate n potential qns, qi based on given answer a(q). Use a text embedding model to obtain embeddings for all qns. For each qi, calculate similarity sim(q, qi) w original q  Answer relevance score AR for qn q = AR = mean of all sim(q, qi)  Retrieval quality = evalute degree to which retrieved context supports the query. A set of key sentences (Sext) is extracted from context c(q). Relevance is calculated at sentence level = CR = num of extracted sentences / total num of sentences in c(q) |
| Context recall = level of consistency btw the retrieved context and the annotated answer. Requires groundtruth  context recall = |Ground truth sentences that can be attributed to context| / |Num of sentences in Ground truth sentences|  Context precision = whether all relevant contexts containing true facts that are retrieved are ranked at the top  Context Precision@k = ∑ precision@k / total num of relevant items in top K results  Precision@k = true positive@k / (true positives@k + false positive@k) |

|  |  |  |
| --- | --- | --- |
| Num of params | FFNN, where i = input size, h = size of hidden layer, o = output size. Num of params = (i + 1) \* h + (h + 1) \* o  RNN, where g = num of FFNN in a unit (RNN has 1, GRU has 3, LSTM has 4), h = size of hidden units, i = dim/size of input  Num of params = g \* [(h + i + 1) \* h]  Llama2 num of params = embed\_params + num\_layers \* (attention\_module\_params) + mlp\_block\_params + per\_layer\_rms\_norm\_params + pre\_lm\_head\_rms\_norm\_params + lm\_head\_params = [32000 \* 5120] + 40 \* [4 \* 5120 \* (128 \* 4)] + [3 \* 5120 \* 13824] + [2 \* 5120] + [5120] + [5120 \* 32000] | |
| Large Models | Slow Training and inference causes: 1) Model scale = large num of weights and computations  2) Architecture: Attention operations has quadratic complexity (computation & memory & memory access) w.r.t input token length  3) Decoding approach in inference: Generate tokens one by one (fully sequential)  Possible solns: a) Structure Design (MoE = Mixture of Experts, Multi-query attention). b) Inference Engine & Serving Framework (Flash attention, vLLM, DeepSpeed, NVIDIA TensorRT). c) Model compression (quantization) | |
| Quantization | Convert and store weights of NN using lower precision data types  Lower computation workload + lower memory operations -> lower latency, higher throughput, lower power consumption, fewer GPUs  Floating point representation: use n bits to store value. Num = (-1)sign \* baseexponent - 127 \* significand  1) Sign: positive or negative. Use 1 bit, where 0 = +ve, 1 = -ve.  2) Exponent: Power to which base (usually 2) is raised. Exponent can be +ve or -ve. 3) Significand/Mantissa. INT8 = [0, 255] or [-128,127]  FP32 = 1 bit for sign, 8 for exponent, 23 for significand. FP16 = 1 sign, 4 exponent, 10 significand. BF16 = 1 sign, 8 exponent, 7 significand. | |
| Mixed precision approach for training and inference: weights are held in FP32 as a precise "main weights" reference, while computation in forward and backward pass are done for FP16/BF16 to enhance training speed. FP16/BF16 gradients are used to update FP32 weights  To calculate model size in bytes: multiply num of params by size of chosen precision in bytes | |
| Symmetric Quantization: for INT8 use [-127, 127] to obtain symmetric range  a) . Xdequant = s \* Xquant. where clamp(x; a; c) = and s = 2 - k  b) Absolute max (absmax): Xquant = round(127/max|**X**| \* X). Xdequant = max|**X**|/127 \* Xquant. Some loss of precision due to rounding | |
| Asymmetric Quantization: [0, 255]. s = scale factor, z = zero-point, b = bit width  a) Uniform Affine quantization: . Xdequant = s \* (Xquant - z)  b) Zero-point quantization: s = 255/(max(**X**) - min(**X**)). z = -round(s \* min(**X**)) - 128. Xquant = round(scale \* **X** + z). Xdequant = (Xquant - z) / s  Asymmetric quantization incur less error in dequantization but more overhead of calculation compared to symmetric | |
| Yq = XW + B. Can quantize X, W, B. B usually quantized as int32. X can be quantized "on the fly" using dynamic quantization or w observers  Y can't be quantized since it's the result of an operation, and can't calculate its scale (s) and zero (z): instead, run inference on model using a few inputs, and "observe: the typical output to calculate s and z. This processs = calibration | |
| Post Training Quantization (PTQ) = weights of trained model converted to lower precision w/o needing any retraining. Easy to implement but often result in performance degradation  - Pre-trained model -> Attach observers -> Calibrate (performace inference on unlabeled data and calculate s and z) -> Quantized model  Quantization Aware Training (QAT) = incorporates weight conversion process during pre-training or fine-tuning stage. Computationally expensive, require representative training data but better performance | |
| PTO: A diagram of a algorithm  Description automatically generatedHow to efficiently calculate WX + b  MAC array for efficient computation  A math equations and numbers  Description automatically generated with medium confidenceEach row and col can be calculated in parallel  , where hat are quantized values | |
| A diagram of a machine learning  Description automatically generatedQAT: Insert some fake modules in the computational graph of model to simulate effect of quantization training.  Backpropagation algo updates weights that constantly suffer from effect of quantization, and usually leads to more robust model  During backpropagation, model needs to evaluate the gradient of the loss function w.r.t every weight and input. Problem: what is the derivative of the quantization operation we defined before?  A typical solution is to approximate the gradient with the STE (Straight-through Estimator) approximation.  STE appox = 1 if value being quantized , else 0 | |
| Optimize LLM Inference – vLLM | KV Cache: During inference from transformers, we get the query vector incrementally and sequentially. This is multiplied by the Key vector to get the attention matrix of each token with previously generated tokens and itself. Then, after taking softmax, we multiply with the value vector to get the self-attention score.  E.g. T = 4. Then token 1 to token 4 comes sequentially as attention computation token 4 depends on all previous tokens  Q and K matrix multiplications grows along w attention matrix, but K and V value matrix remains the same for all previous tokens.  So can cache K and V matrix as they are not going to change but cannot cache Q matrix as it changes | |
| Problems: KV caches usually stored in a contiguous memory. If have parallel multiple request, need to store separately which waste memory and potentially leads to an out of memory (OOM) error  Prompts for each of these requests are nearly the same (e.g. system prompt like "you are a helpful assistant ..."), so storing them again and again in a contiguous memory is inefficient.  Due to sequential nature of inferencing, have to reserve memory (pre-allocation) for KV cache as it needs to be stored in contiguous form  Can reserve according to max output seq length but create bottleneck. If generate small response, will be wasting the space | |
| Soln: Paged attention: allow storing continuous keys and values in non-contiguous memory space. Stores a block of KV matrices, where each block contains key and value vectors for a fixed num of tokens  For easier implementation, vLLM authors store key and value matrices of diff heads and layers in a block  Store these KV matrices in blocks and map the blocks to non-contiguous physical memory  To generate tokens: PagedAttention kernel can get KV blocks (via page table in contiguous memory) in non-contiguous physical memory | |
| Use vLLM when maximum speed is required for batched prompt delivery.  Opt for Text generation inference if you need native HuggingFace support and don’t plan to use multiple adapters for the core model.  Consider CTranslate2 if speed is important to you and if you plan to run inference on the CPU.  Choose OpenLLM if you want to connect adapters to the core model and utilize HuggingFace Agents, especially if you are not solely relying on PyTorch.  Consider Ray Serve for a stable pipeline and flexible deployment. It is best suited for more mature projects.  Utilize MLC LLM if you want to natively deploy LLMs on the client-side (edge computing), for instance, on Android or iPhone platforms.  Use DeepSpeed-MII if you already have experience with the DeepSpeed library and wish to continue using it for deploying LLMs. | |
| Fine Tuning | Use In-context learning when direct access to LLM is limited, such as when interacting w LLM through an API or user interface.  Cons: large num of tokens become a problem during docQA. Maintain templates for slight variations. Not fully reliable  Using fine tuning: no need to worry about tokens. Can finetune smaller models to specialize on use cases  Fine tuning update params of a pre-trained model w new data. Cons: requires retraining all model params which takes time and money  Also require alot of memory as need to store LLM, optimizer states, gradients, forward activations, ...  Also incr risk of overfitting, when new data is small or noisy, which can degrade model's performance on other tasks or domains  Can also lead to catastrophic forgetting as it changes all params of the model. Since Parameter Efficient Fine Tuning (PEFT) only updates a small subset of params, more robust against catastrophic forgetting effect | |
| A diagram of a process  Description automatically generatedPEFT: tune only a small fraction of weights. Techniques: Adapters, LoRA  Adapters = special type of submodule added to pre-trained model to modify their hidden representation during fine tuning  By inserting adapters after multi-head attention and feed-forward layers in transformer architecture, can update only params in adapters during fine tuning while keeping rest of model params frozen  The adapter module comprises two feed-forward projection layers connected with a non-linear activation layer. There is also a skip connection that bypasses the feed- forward layers. | |
| A diagram of a process  Description automatically generatedLoRA (Low Rank Adaptation of LLM)  - pre-trained parameterized model reside in a lower dimension and changes in weights from gradient descent have a lower intrinsic rank. Rather than optimizing parms of dense layers, can represent them in lower dimensions using SVD (singular value decomposition) and then do gradient descent to optimize weights at lower dimensions  SVD:  To compress a matrix to dim of r from n, select first r eigenvalues, and then zero out other diagonal elems.  A diagram of a diagram of weights  Description automatically generated  Query, key and value feedforward layers have dim d x d, which is in a higher dimension.  The A matrix is initialized as a Gaussian w 0 mean and s.d., and B matrix initialized as all 0s  B matrix of dim d x r and A has dim r x d.  Lower rank r and more num of matrices for finetuning is preferred  Goldilock zone for rank matrix r is 4 or 8 | |
| A diagram of a diagram of a diagram  Description automatically generated with medium confidenceLoRA + Finetuning = QLoRA  (Quantized Low Rank Adaptation) = extension of PEFT approach  Uses 4-bit NormalFloat (NF4): new data type for normally distributed weights in a NN  Quantization of quantization constants to save memory  Paged optimizers to manage memory spikes  NF4 solves issue of outliers. Outliers are essential for NN as they contribute most to loss function. Assums NN weights have normal dist w 0 mean and s.d.  So they transform all weights to a single fixed dist by scaling w and set an arbitrary range of [-1, 1]  Rather than equally distributing the buckets based on width, we can distribute the buckets based on equal probability mass. The block width can be arbitrary, but each bucket needs to have almost block\_size(b) number of elements.  Paged optimizer: optimizer states are evicted to CPU RAM whenever the GPU runs out of memory and then it is paged back into GPU VRAM when the optimizer state is required for gradient backward pass to update the weights. It is just an optimization technique and it is useful for longer sequences where the GPU VRAM is not sufficient to hold all the optimizer states. | |
| Other Fine Tuning Approaches | A diagram of a structure  Description automatically generatedPrefix tuning: lightweight alternative to standard fine tuning. Keeps LLM params frozen and optimizes a small continuous task-specific vector called prefix  Prefix = set of free params that are trained along w language model  Goal of prefix tuning is to find a context that steers the language model toward generating text that solves a particular task  Prefix can be seen as a seq of "virtual tokens" that subsequent tokens can atten to  Prefix-tuning outperforms fine-tuning in low-data settings, and extrapolates better to examples with topics unseen during training. | |
| A diagram of a transformer layer  Description automatically generatedPrompt Tuning: learning soft prompts through backpropagation that can be fine-tuned for specific tasks by incorporating labeled examples.  Prompt tuning outperforms the few-shot learning of GPT-3 and becomes more competitive as the model size incr. Also benefits domain transfer’s robustness and enables efficient prompt ensembling.  Storing a small task-specific prompt for each task, makes it easier to reuse a single frozen model for multiple downstream tasks, unlike model tuning, which requires making a task-specific copy of the entire pre-trained model for each task.  Some vectors are prepended to beginning of a sequence at the input layer. When presented with an input sentence, the embedding layer converts each token into its corresponding word embedding, and the prefix embeddings are prepended to the seq of token embeddings.  Next, the pre-trained transformer layers will process the embedding sequence like a transformer model does to a normal seq. Only the prefix embeddings are adjusted during the fine-tuning process, while the rest of the transformer model is kept frozen and unchanged. | |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |