|  |
| --- |
| NN History: [1958 (Perceptron, Rosenblatt), 1969 (XOR problem, Minsky & Papert), 1974 (Backpropagation, Werbos), 1986 (MLP, Rummelhart, Hilton & Williams, RNN, Rummelhart), 2020 (GPT-3, OpenAI)] Neural networks are **directed / sequential** (one neuron to another, i/p -> hidden -> o/p layer); Directed Acyclic Graph (DAG): Directed (one-way); Acyclic (no cycles or closed loops); Perceptron: simplest form of NN model, for binary classification tasks, single-layer n/w, input x weights, perceptron cannot solve XOR (⊕) as **XOR** require non-linear boundaries; uses step function / Heaviside function as activation function; trained using supervised learning, adjusts weights via backpropagation (n(ytrue - ypred)xi); **n = learning rate**; **Chan rule**: Each layer o/p depends on previous layer, applies **chain rule of calculus**, Activation functions: [*Purpose*: Non-linearity, O/p control, Gradient propagation]; **ReLU** (hidden layer only, only +ve , fast, doesn’t saturate, efficient computation, few neurons activated at same time, faster convergence); **Sigmoid** (hidden or o/p layer, 0 to 1, on/off style, binary classification, independent outputs, vanishing gradient problem, not zero-centered, exp() is computation intensive); **Tanh** (hidden layer only, -1 to 1, both +ve & -ve, zero-centered, train faster); **softmax** (output layer, multi-class classification problems but not multi-label classification, 0 to 1, sum adds to 1, dependent outputs); Issues with activation function: **Saturation**: o/p is pushed to extreme value (e.g. close to 0 or 1 in sigmoid), result is near-zero gradient; learning is impacted; **Vanishing gradient**: gradients of loss function become extremely small, preventing weights to be updated in backpropagation (especially in DNN), sigmoid & Tanh suffer from this, ReLU mitigates vanishing gradients; **Logical AND**: both i/p needs to be 1, step-function; **Logical OR**: either of them is 1 or both are 1, step-function; **XOR**: either i/p is 1, both can’t be 1, step-function; single-perceptron (AND, OR) can handle only linear boundaries; XOR requires non-linear boundary; Network (hidden layer) can solve XOR; **Perceptron learning rule**: Initialization (randomly assigned) -> Prediction (weighted sum + activation func) -> Weight update (adjust weights & bias based on learning rate); **MLP parts**: I/p layer, Hidden layer, Activation func, O/p layer, Weights & Biases, Loss func, Backpropagation; **DNN**: Hidden layer (2 or more), Increased capacity (more intricate/non-linear relationships in data), Feature abstraction (each hidden layer learns more abstract representation of i/p data, more hidden is more complex representation), training w/ backpropagation (determine gradients for each weight & bias, *uses activation function to calculate gradients*); **GPUs**: Parallel processing (1000s of cores for vector calculations), High throughput (large data in parallel), Versatility (wide range of DNN frameworks), Memory bandwidth (high memory allows fast data transfer b/w memory and processor); TPUs: large-scale ML, offered thru CSP; **Loss function**: Cost func, Objective func, diff b/w predicted and actual value, feedback to adjust weights & biases; **Epochs**: one pass thru entire training dataset, ML training has multiple epochs; **Pass**: one forward and one backward propagation, subset of dataset; **mini-batch**: small, random-select subset of training dataset in single iteration of training, computationally beneficial; **Size of batches**: 32, 64, 128 etc. is a decision on tradeoff b/w computation & better uncovering of patterns in data; **Forward pass**: model computes predictions for each mini-batch; **Loss calc**: Computed on all samples in mini-batch; **Backward pass**: gradient calc: gradients of loss w.r.t parameters (weights & bias) for mini-batch; Loss calc for numeric: MSE, MAE; **Huber loss**: combine MSE & MAE, balancing sensitivity to errors & robustness; **Log-Cosh loss**: Alternative to MSE, robust to outliers; **Quantile Loss**: For uncertainty; Loss calc for classification: **Binary Cross-Entropy**: Logarithm loss, probability b/w 0 & 1, uses sigmoid in general, High cross-entropy means more misclassification, only 2 classes, used in spam detection, disease diagnosis; **Multiclass classification with DNN**: Uses softmax activation func, **categorical cross-entropy loss func**, O/p layer has one neuron per class, one-hot encoded labels, used in image recognition, sentiment analysis; **Minimizing loss**: Adjusting model parameters θ, move in negative gradient direction of loss func to reach minimum; **Stochastic Gradient Descent (SGD)**: updates θ for every observation in entire network, computationally expensive, instead run Batch SGD that will update θ after batch (entire dataset, per epoch) is processed. Weight update alternatives / Optimizers: SGD w/ Momentum (accelerate convergence, when loss surface is not well-conditioned, gradients oscillate in a direction), RMSprop (Root Mean Squared Propagation, exponentially decaying average of squared gradients to adjust learning rates, prevents learning rate shrinking, used in RNN), Adam (combines momentum and RMSprop, compute adaptive learning rates for each parameter, accelerate convergence), Nadam (variant of Adam, Nesterov momentum, improves Adam performance, look-ahead mechanism of NAG); Learning rate: **Too high** (large updates / initial convergence, risk of overshooting, fail to converge, terminate early, poor perf, oscillates); **Too low** (small updates, slow convergence, excessive epochs, slow perf / stuck early or saddle points); **Number of parameters**: *(Nbr of I/ps x Nbr of neurons) + Nbr of neurons*; If zero i/ps, then zero params; *Nbr of neurons = Nbr of bias*; **CNN**: mimics how humans classify images by recognizing features or patterns to object class; low-level features combined to form high-level objects, uses filters or kernels to detect patterns, filters are learned during training (e.g. vertical edge, horizontal edge), multiple filters in each layer; **CNN architecture**: I/p -> *convolve* (filter, kernels to extract features) -> *pool* (reduce dimensions, retain imp features) -> *flatten* (reshaping to 1D, simplify computation); **Max pooling**: summarize each non-overlapping block of pixels in image using max value in block; **Average Pooling**: takes average, preserving all info but smoothing it; **Global pooling**: reduces entire feature map into single value; **L2 pooling**: magnitude of values; **Mixed pooling**: add randomness with both max and average pooling; **RNN**: I/p is a sequence, Schmidhuber & Hochreiter in 1997, order & closeness of certain words in a sentence convey semantic meaning, maintain memory of previous inputs, ideal for order & context of data such as time series prediction, NLP, speech recognition; **RNN applications**: Documents, Time series, Handwriting, etc.; **RNN memory**: Hidden state (summary of i/p), Shared Weights (same set of weights across all time steps), Feedback loops (previous time step to influence current step, model dependencies over time); **DNN variants**: Long Short-Term Memory (**LSTM**): kind of RNN that overcome vanishing gradient problem, have gates to control flow of info, remember long-term dependencies; Gated Recurrent Unit (**GRU**): Variant of LSTM, solve vanishing gradient problem, simple architecture/few gates, computationally cheaper; Bidirectional RNN: Info flows forward & backward. Access to future data & past data when predicting; **RNN steps**: Initialize weights & hidden state -> Feed i/p sequence step-by-step -> update hidden state & compute o/p at each step -> Accumulate loss across sequence -> BPTT backpropagate through time to calculate gradients -> update weights based on gradients -> repeat over multiple epochs for model training; **Time Series Data**: sequence of data points over time. E.g. daily temperatures, stock prices; **Smoothing**: Remove noise, identify trends, make predictions easier; *Smoothing methods*: SMA **(Simple Moving Average)**: replace each data point with average; **WMA (Weighted Moving Average)**: more weight to recent points; **Exponential smoothing**: recent data points are exponentially more weight; **Seasonality:** patterns, frequency, intervals; **Trend lines**: Linear (steady rise or fall), Exponential (accelerating or decelerating), Polynomial (complex pattern w/ fluctuations), Logarithmic (growth patterns that level off over time), Moving average (data is noisy to reveal core trend); *Stationary time series*: constant means, variance, & autocorrelation (e.g. random fluctuations around horizontal line without upward/downward trend); *Non-stationary time series*: Trend (long-term increase or decrease), Seasonality (repeat patterns or cycles), Changing variance (spread of data changes over time); **Reduce trend**: Differencing, Detrending, Logarithm transformation, moving average smoothing; Stationarity helps in model assumptions, predictability, simplifies analysis; STL (Seasonal & Trend Decomposition): Time series analysis method to break down series into Trend, Seasonality, Residual -> understand patterns & random variations in data; Additive & Multiplicative; ARIMA: **Auto Regressive (AR)**: relationship b/w time series & its past values (lags); **parameter (p)**: how many previous time steps to include in model; **Integrated (I)**: differencing time series to make it stationary (remove trends/seasonality), **parameter (d)**: how many times to difference the data to remove trends; **Moving Average (MA)**: models relationship between time series and past forecast errors; **parameter (q)**: Nbr of lagged forecast errors included in model; **ACF**: correlation between time series & its labs (to determine **q**); **PACF plot**: to determine **p**. |
| Protocol must be well-known by each party involved for successful encryption; **Perfect Secrecy**: Cipher text provides no additional info about plaintext to adversary, even if unlimited compute power; **Kerckhoff’s principle**: rely on secrecy of key, not the algorithm or design; **Base64**: Convert binary into text formats, divides data into 3 bytes (24 bits), then split into 4 groups of 6 bits. **A-Z a-z 0-9** **+** **/**. Padded with = characters to align with 3 bytes. **Stream cipher**: encrypt one bit at a time, when data arrives continuously, where low latency is required, suitable for real-time apps (voice, video); RC4, ChaCha20, Salsa20; Shared key used to initialize the cipher. **IV**: random, non-repeating value used along w/ secret key to ensure randomness in encryption when same key is used for each session. IVs are not secret, sent along with ciphertext. **LFSR (Linear Feedback Shift Register)**: Linear function (XOR), shift bits w/ each clock cycle, used to generate pseudorandom bit streams for encrypting plaintext. Generated keystream is XORed (**⊕**) w/ plaintext to produce ciphertext, PRNGs, Error detection & correction; **One Time Pad (OTP)**: key as long as message, truly random key, single use, perfect secrecy; **Block cipher**: Divides plaintext into blocks before encrypting each block independently w/ key, block size: 64 bits, 128 bits, AES uses 128-bit blocks, *Deterministic for a given key*: For a same key, and sample plaintext block, always leads to same ciphertext block; **ECB**: Simple, encrypt each block independently, repeated patterns (plaintext & ciphertext combination); **CBC**: Each block is XORed with previous ciphertext block before encryption. IV added at beginning for randomness; CFB & OFB; CTR; CBC used in TLS, IPSec, data in DB, File encryption; **AES**: key sizes: 128-bit [10 rounds], 192-bit [12 rounds], 256-bit [14 rounds], block cipher (block size always 128 bits), replaced DES; **AES Rounds**: Substitution (uses table & predefined cipher), ShiftRows (data rows by 1, except 1st ), MixColumns (Hill cipher), AddRoundKey; Decryption: Inverse SubBytes, Inverse ShiftRows, Inverse MixColumns, AddRoundKey; **Hashing**: Ensures confidentiality & integrity of message; Collision resistant: Two different i/p cannot produce same hash; **Salting**: adding random chunks of bits to end of password before hashing; save salt along with hash value; **Asymmetric Enc**: Python’s random module uses PRNG; Python’s **os.urandom**: Random bytes from OS’s entropy pool, derived from h/w noise, and other unpredictable system activities; Python’s secrets: high-level interface for security-sensitive applications, built on os.urandom; **Primes**: vital to security of encryption schemes; **Prime factorization or integer factorization** (used to secure public-key encryption schemes, large semiprime numbers that are results of multiplication of two prime numbers), factoring is computationally difficult; **Hardness in computation**: ease to compute value but infeasible hardness to reverse the process; Large composite number = multiple (**two large prime numbers), factoring in RSA**. Y = gx mod p (hard to determine x, even if g, y, p are known -> **discrete logarithmic problem**, used in **Diffie-Hellman & ECC**; RSA solves symmetric key distribution problem; **ECC**: elliptic curves; Scalar multiplication is computationally efficient but reversing it (finding K, private key, given P, public key) is extremely hard, called **ECDLP (Elliptic Curve Discrete Logarithm Problem)**. **ZKP**: prover to convince verifier they know a value without revealing any info about value itself; Privacy, Blockchain, Authentication; **Homomorphic Enc**: Computations to be performed on enc data without dec it first; **HE workflow**: Key gen -> Enc -> Computation -> Decrpt; **TenSEAL**: Lib supports Full HE, Cloud Ops, & Privacy Preserving ML; **crypto libs**: *import cryptography*; *import hashlib*; *from passlib.hash import bcrypt*; |
| TCP/IP: App -> Host-to-Host -> Internet -> N/w access; **TCP**: SYN, SYN-ACK, ACK, Unicast: One-to-one, Packet order, error checking, flow control; **UDP**: Request, response, Broadcast: One-to-all, Multicast: One-to-several, Real time communications, Gaming, IoT, DNS; **Nodes**: Devices that connect to network (smartphones, servers, computers); Firewalls, IDS, IPS, NAC, VPN; **IPv4 private**: 10.0.0.0 to 10.255.255.255 (A); 172.16.0.0 to 172.31.255.255 (B); 192.168.0.0 to 192.168.255.255 (C); CIDR (Network and Host space); IP header, TCP header, NAT, IPv6 (128-bit, /64 prefix) vs IPv4 (32-bit); ARIN (North America), LACNIC (South America), AFRINIC (Africa), RIPENICC (Russia), APNIC (Asia); Segmentation; **ZTA**: **Control plane** (PDP: [Policy engine(make decision to grant/deny), Policy admin (establish & terminate connections b/w user/devices & resources)]) & **Data plane**: (Subject, system, PEP (Control access to resource based on decision from policy engine, like firewall, agents), Enterprise resource); Never trust, always verify; ensure secure access & resource control by enforcing stringent verification process; NIST principles: Least priv access, Continuous verification, **Dynamic Policy enforcement** (adapted by context, behavior, threats); DoS, DDoS, MITM (Sniffing, Scanning/Eaves dropping, Spoofing); **Scapy**: *from scapy.all import \**; *packet = IP(dst="8.8.8.8")/TCP()/Raw("Hello")*; **send()** at layer 3, **sendp()** at layer 2, **sr()** send and wait for response, returns tuple of answered & unanswered packets; *packets = sniff(filter="tcp", count=10); wrpcap("captured\_packets.pcap", packets)*;  **Offensive security**: Proactive approach, uncover vuln, faults, threats before bad actors do, think & act like adversaries, fix vuln, stay ahead of attackers, analyze ability to responds to incidents, Python is effective (many libs, integrations w/ Metasploit & Zap); Offensive includes: Penetration testing, Red teaming (overall security resilience), Vuln research (responsible disclosure, support in patches), CTF (red & blue team contest, real-world simulation, inspire competitors, take advantage of them); Kill chain: Reconnaissance (conference info, get email IDs), Weaponization (exploit w/ backdoor into delivery payload), Delivery (bundled into email, web, usb), Exploitation (execute code/exploit vuln), Installation (put malware on asset), C2 (remote manipulation channel), Action on Objectives (hands-on-keyboard access to accomplish goals); Ethics for pen test: Legal authz, Define scope, Disclose vuln (w/ guidance), maintain confidentiality; **Threat actors**: Cybercriminals, nation-state, insiders, hacktivist; **Attack vectors**: method or pathways used to compromise system (phishing, malware, social engg); **Vuln**: weakness in systems/apps; **Impact**: Damage from attack (financial, reputational, operations disruption); |
| **Cyber env**: 3 layers / frameworks: **Physical** (tangible, h/w-based, physical devices, infra, physical connections for comm e.g. PCs, routers, servers, IoT, Fiber-optic, data centers; *Threats*: Physical destruction, tampering, theft, power outages); **Logical** (digital infra, flow & process of data, protocols, apps, logical conn b/w systems, Network protocols TCP/IP, HTTP, Software apps, OS, VMs, Cloud, Virtual n/w; *Threats*: Malware, ransomware, DDoS, hacking); **Human** (human & cyber env like users, administrators, policies, decision making process; End-users, IT admins, Org staff, policies, training, gov frameworks, Social engg, human-error vuln; *Threats*: Phishing, insider threats); APT: Advanced (Adv techniques, multimethod approach/tool, vuln discover); Persistent (targeted diligence, long-term target access, dormant activity); Threat (defined obj, malicious actors); Offensive security lifecycle phases: Planning (**Whitebox vs Blackbox**: [difference in knowledge required (code vs functionality), focus (code accuracy/logic vs user exp & req), performed by (developers vs QA testers/end users), tools (debuggers/profilers/code analyzer vs automation tools/test mgmt. tools), use cases (unit test/code optimize vs functional test/regression test)]); Recon (gather info like passive recon like public data, active recon via scanning target, OSINT, profiling target like employees, assets); Enumeration (get clear picture / detailed info of security landscape via n/w scan, service version details, vuln scan); Exploitation (use of known exploits, priv esc, payloads [*in pen tests, no damage or disruption shall be done*], Kali linux: Metasploit, BeEF, Sqlmap); Maintaining Access (*Evade detection*: low-noise exploits, no bruteforce, encrypt payloads to evade from IDS, covert channels for C2, clear logs; *Backdoors & rootkits*: hide malicious activities from tools); Reporting (Completes after notifying results of accomplishing objectives: *Report*: Summary of findings, vuln discovered, exploit details, recommendations; *Impact analysis*: assessment of impact & critical vuln; maintain CIA of systems/data; *Recommendations*: roadmaps to address issues, program effectives to decrease repeat issues); MITRE ATT&CK: kb of adversary tactics & techniques based on real-world observations; **Tactic**: “why”, goals & obj of attacker, like Initial access; **Techniques**: “how”, actions or methods used by attacker, detail desc of steps/process to achieve goals, like phishing to get initial access; *List of tactics*: Initial access, Execution, Persistence, Priv Esc, Defense evasion, Credential access, Discovery, Lateral movement, Collection, C2, Exfil, Impact; **Metasploit**: Exploits, Paylaods, Aux modules, Post-exp modules, Encoders, Nops; **Defensive security**: CTI: Gathering & using intel to predict & counter threats; **IoC**: Identifying signs of breach; **Strategic CTI**: high-level, long-term insights for decision makers; **Tactical CTI**: Details about TTPs used by attackers; **Operational CTI**: Specific & imminent threats or incidents; **Technical CTI**: low-level technical data for security tools; **Threat actor profile**: Motivation, TTPs, Skill level, Tools & resources, Attack complexity, Resources & org, Targeting & Victimology, Persistence & Adaptability, Geographic & cultural indicators, Public exposure; SIEM: Log collect, Parse & Normalize, Storage, Correlation engine, Alerts, Dashboards & Reports; SOAR: Playbook mgmt., EDR, Security Operations Automation, Vuln mgmt., Case mgmt. based on IR, Threat intel; **SOAR components of playbook**: Trigger (alert from SIEM, initiate playbook), Tasks (actions executed in workflow; isolate machine, notify admin), Decision Points (Conditional steps based on previous action; escalations), Outputs (Mitigation of threat; report, ticket closure); **Vuln mgmt.**: Patch mgmt., Pen testing (simulate attacks), Attack surface analysis (exposed entry points), Zero-day vuln mgmt. (unknown threats); IR process: NIST 4-phases: Preparation (plan to prevent & respond; ), Detection & analysis (Whether incident occurred, severity, type); Containment & eradication (halt effects); Post-incident recovery (lesson learnt, involve parties ); **Structured Threat Information Expression (STIX)**: language for represent & desc CTI to be shared, stored, analyzed in consistent manner, IoC, attack pattern, WHAT is share (TTPs), JSON; **TAXII (Trusted Automated eXchange of Intel Information)**: transport protocol to exchange threat intel in standard format between systems; automation of sharing structured threat intel in STIX format, HOW to share, Rest API; STIX components: Observables, Indicators, Incidents, Adversary TTP, Exploit targets, Courses of action, Campaigns, Threat actors, Reports; Threat feeds: MISP, AlienVault, IBM X-Force, Threat Connect; Defensive Deception: strategy when org deliberately presents false info or decoy systems to mislead attackers, diverting attention away from critical assets, gaining valuable insights into TTPs. Pros: Early threat detect, Reduce false +ve, Threat Intel gather, Adv threat mitigation; Cons: Increase mgmt. overhead, legal issues, easily defeated by adv adversaries; Types of deception: **Honeypot** (decoy computer systems designed to be probed, attacked, compromised); Honeytokens (bait resources like accounts, user files, DB entries, passwords); **Camouflaging** (Hiding real org’s systems/data via obfuscation, renaming, altering system attributes); Moving target defense (technique to change targets); Breadcrumbs (pieces of info to lure attackers towards decoy); BC/DR: **BCP**: Time-sensitive (hours or minutes); Maintain operational continuity during disruption, scope: all org aspects: people, process, technology; metrics: RTO, max allowable downtime for critical process, RPO: how much data loss is acceptable for buss ops ; **DR**: Can take long time; Restore IT systems/data after disruption, scope: technical recovery: servers, DBs, apps; metrics: RTO, ensure IT systems are back up within a specific time, RPO: Data is recovered to specific point in time before disruption; |
| NLP: Computers to understand, interpret, generate human language that is meaningful & useful; **NLP components**: Tokenization (Break text into units like words, phrases to analyze individually); Part-of-speech tagging (identify grammar role: noun, verb, adjective); Named Entity Recognition (NER) (Identity & classify entities like people, places, org, dates); Sentiment Analysis (sentiment or emotion behind text); Text analysis (predefined labels or classes, spam/ham); Dependency Parsing (analyze grammar structure of sentence by relationships b/w words); Machine translation (one language to another); Text summarization (Condense large pieces of text while preserving key info); **NLP application**: Chatbots, Virtual assistants, Sentiment Analysis, Language translation, Speech recognition, Text summarization; Cybersecurity use cases for NLP: Threat Intel & analysis (from blogs, news, forums, dark web discussions), Phishing email detection (linguistic cues, URL analysis, tone analysis), Malicious document detection, Incident response, Vuln mgmt., User behavior analysis, Social engg detection, Compliance monitoring, Log analysis, Dark web monitor; **NLP timeline**: Turing test to assess machine’s ability to exhibit intelligent behavior equal to that of human; 2010s: Deep learning, relationship between words (BERT, Word2Vec); 2020s (LLMs, GPT series); **Tokenization**: Word (by space or punctuation), Sentence (for translation), Subword (**Byte-Pair Encoding** [BPE: merge most-frequent pairs, chars or words], WordPiece [BERT uses it], Unigram Language Model [Google’s SentencePiece, most-likely subwords by probabilistic model]), Character (individual chars, misspelled words); Benefits of NLP: Model input, **Vocab mgmt**., Handle complex language, Ambiguity, Language differences; NLP tokenization libs: **NTLK** (easy but slow, not for large scale), SpaCy, **Hugging Face** (for transformer models), Stanford NLP, Genism, SentencePiece, **Regex**; **TensorFlow Tokenization**: tf.keras.preprocessing.text; Convert text into numerical format (integers); Word frequency-based indexing: assign higher value to words that appear less frequently, and low values to more frequent words; Handling Out-of-vocabulary (OOV) words: OOV token can be used to define those; Text sequences to integer sequences: After tokenizing & encoding, transform entire text into integer sequences; Also pad/truncate to ensure same length; **Embeddings**: Convert tokens into dense fixed-size vectors w/ semantic relationships between tokens (pre-training step); Output is a dense vectors of floating-point arrays representing the position of each token in embedding space; **Bag of words**: (not embedding) After tokenization, the ‘bag of words’ model takes tokens and counts occurrences of each token in the document. Order of token doesn’t matter, just frequency or presence. BoW represents entire document as vector of word frequencies; BoW vector puts 1 when it has the word in that sentence, else 0 in that vector array; Term frequency / Inverse Document Frequency: (not embeddings) importance of word in a set of documents; **TF-IDF** is a vector based on word frequency, and how unique or rate the word is across all docs; TF(t) = Nbr of times term t appears in doc / total nbr of terms in doc; IDF (t) = log (total nbr of docs / Nbr of docs containing term t); TF-IDF (t) = TF (t) x IDF (t); **Word2Vec**: Not an algorithm; Family of model architectures & optimization to learn word embeddings from datasets; 2 methods in W2V: **Continuous bag-of-words** (predict middle word based on context words, sg=0), **Continuous skip-gram** (sg=1, predicts words within certain range before & after current word in same sentence); **Cosine Similarity**: used in W2V to measure similarity between word embeddings; Cosine Similarity(A,B) = A.B / ||A|| ||B|| (dot product and || is magnitudes of vectors), ranges from -1 to 1; 1 = two vectors are identical; 0 = orthogonal (no similarity); -1 = two vectors are diametrically opposite; **GloVe**: Global Vectors for word representation; Unsupervised algo from Stanford; Computationally efficient, Interpretability (embeddings can be analyzed to understand relationships between words, *interpretability*); GloVe provides word vectors, dimensionality, pre-trained models; **Transformers**: NLP, Computer vision, Time series forecasting; **Self-attention**: focus on different parts of i/p sequence when making predictions; model understands relationships between elements at different positions in sequence; each token interacts with other token to produce a representation that captures context meaning; computes 3 vectors for each token: **Query, Key, Value**; These calculate attention scores; Query vector (Q): word being processed; Key vector (K): words in sequence compared against; Value vector (v): words being compared to compute final weighted representation; **Steps**: Each word transformed into Q, K, V -> Attention scores w/ dot product of Q and all K values -> Scores are scaled and passed thru Softmax to get attention weights -> Weights are used to compute weighted sum of value vectors; **Encoder**: Input pre-processing; **Decoder**: Output pre-processing & Output post-processing; **Multiheaded attention**: Instead of single attention, uses multiple heads; each head independently computes attention scores; **BERT**: Bidirectional Encoder Representations from Transformers; Google in 2018; Transformers to understand context of words; to understand meaning of sentence by taking left & right context of word; powerful than previous models; BERT Applications: Question answering (SQuAD), Text classification, Named Entity Recognition (NER), Sentence similarity, Translation; BERT mask token, next sentence prediction, sentence pair classification; **BERT Steps**: Install transformer lib -> Load pre-trained BERT model & tokenizer -> Tokenize input data -> Pass tokenized data into model for embeddings -> optionally, fine tune model for specific tasks like classification -> use model for inference or continue training; Hugging face; Architecture blocks; **Multithreading vs. Multiprocessing** in python; |
| **EDA steps**: df.head() //first 5 rows; df.shape //rows & columns; df.columns //get column names; df.dtypes //data types of each column; df.isnull().sum() //check for null values; df.describe() //statistics of numerical columns; df.describe(include='all') //statistics of all columns including non-numerical; df.nunique() //for unique values in each column; df['column\_name'].unique() //unique values of specific column; df.duplicated().sum() //duplicate rows; df.drop\_duplicates(inplace=True) //drop duplicate rows; df.corr() //get correlation matrix; df.corrwith(df['target\_column']) //get pairwide correlation w/ specific column; df.isnull().mean() \* 100 //nbr of missing values as percentage; df.dropna(axis=1, thresh=int(0.8 \* len(df)), inplace=True) //drop columns with too many missing values; df.dropna(inplace=True) //drop rows with missing values; df.fillna(df.mean(), inplace=True) //fill missing values with mean; df.fillna(0, inplace=True) //fill missing values with 0; df['column\_name'].value\_counts() //value counts of categorical column; df['column\_name'].value\_counts(normalize=True) //normalize value counts to get percentage; df.rename(columns={'old\_name': 'new\_name'}, inplace=True) //rename column; df.sort\_values(by='column\_name', ascending=False, inplace=True) //sort values by a column; df[df['column\_name'] > 100] //filter rows based on condition; df.groupby('group\_column').mean() //group by column and calculate mean; df.quantile([0.25, 0.5, 0.75]) //check for outliers using quantile; df.drop('column\_name', axis=1, inplace=True) //drop a column; df.info() // dataset info; |
| import os; import hashlib; import base64; salt = os.urandom(32); result = hashlib.sha256(salt + user\_input.encode('utf-8')); hash = result.digest(); Zero Knowledge Proof (**ZKP**): prover to prove verifier statement is true without revealing info (used in crypto for privacy & security in systems; resistance to credential leaks, secure MFFA); import requests; from bs4 import BeautifulSoup; response = requests.get(‘https://scrape.com’); soup = BeautifulSoup(response.content, "html.parser"); quote = soup.find\_all("span", class\_="text"); print(quote.get\_text()); key=os.getenv("API\_KEY"); header = {"X-API-KEY": key}; resp = requests.get(url, headers=header) |