Supervised Learning: Classification, Regression
Unsupervised: Clustering, Association (Recco System)

Feature Engineering:
1) Dimension Reduction using PCA/LDA

PCA: PC (nm) = Actual Data (nm) x Eigenvectors of Cov(X) (mm)

- Eigenvector is the unit vector of the PC
- Eigenvalue is the SS distances/variation of data captured
- If plot, called scree plot
- PC1 highest eigenvalue => highest VAR captured
- Loses interpretability, but a huge reduction in data

LDA: New axis are created by $max\frac{\sum d_i^2}{\sum s_i^2}$
Maximise the distance of each category centroid to the overall centroid.
If 2 category it is just the differences between the 2 means
while minimising variation within each category.

- LD1 does the best, LD2 second best, ortho to each other

2) Feature Selection using Regularization
β: coefficients of the features. In NN, they will be weights
These are penalty terms added to the loss function scaled by λ
L1 reg $\|\beta\|_1$: Mahattan norm, $\sum|\beta_i|$
L2 reg $\|\beta\|_2^2$: Elucidean norm squared, $\sum\beta_i^2$

LASSO: **L1** driven by the number of params
Drives unnecessary ones to 0. Multiple solutions

Ridge Regression: **L2** reg driven by the size of the params
Drives unnecessary ones to near 0, unique solution

Net Elastic: **L1** + **L2** reg

Confusion Matrix (Of Classification Model):
Precision: TP / (TP+FP)
TPR/ Recall/ Sensitivity : TP/ (TP+FN)
TNR/ Specificity: TN/ (TN+FP), -ve for Recall
FNR/ Type II: FN/ (FN+TP)
FPR/ Type I: FP/ (TN+FP)
Accuracy: (TP+TN) / (All)
F1 Score:
2*Prec*Recall/(Prec+Recall)
- Harmonic mean,
if any is low, F1 will be low

AUC Curve:
TPR, Recall/ Sensitivity against **FPR, 1 - Specificity**,
- Find the Pareto optimal threshold
- Greater the AUC better the model
- Increase t, lower TPR & FPR

If minority is +ve, very easy to cover all +ve actual and produce good results, AUC is bad for imbalanced data with few TP.

AUPRC: Places **emphasis on +ve minority cases**

If multiclass: agg of the metrics for each class
e.g. mAP in obj detection, cross-entropy loss etc

Variance: Variability of the model's prediction, the mean squared deviation of the predicted points, **$E(pred^2) - E(pred)^2$**

Bias: Difference between the average prediction of our model and the correct value, **E(pred) - actual**

Err(x) = Var(X) + Bias(X)^2 + Var(e), Irreducible Error

- The **more flexibile/complex** a model can be:

1. **Bias decreases**. However when overtrained or too complex, the bias will plateau
2. **Variance increases** as the model fits the data better
3. **Test Err(x) decreases** first as the model fits generally better. However it increases after an optimal point of training or complexity as it overfits the training data
4. **Train Err(x) decreases** as the model fits the data better
5. **Irreducible Error never changes**, independent of model only dependent on data

Underfitting: Increase size model, train more, reduce reg
Overfitting: Add more data, increase reg, dropout, early stopping, feature selection to reduce model complexity/size

To add in:
- Types of losses to use for each class of problem
- Data Imputation: When to use what

Discrete Variables Handling E.g. Categorical (Red, Blue, Green)

- **1) 1 hot encoding:**
  No. of features = No. of classes - 1
  (2 0/1 columns needed minimally)
- **2) Label encoding:**
  Each class given a index number (1, 2, 3)
- **3) Target encoding:**
  Use the target label column calculate weighted mean for each category (e.g. 0.37, 0.29, 0.57)
  Using target encoding can result in data leakage, as we are using our targets to affect the predictors, resulting in overfitting
- **4) k-fold Target encoding:**
  Split the data into folds, for each fold use the rest of the data for their target encoding.

VALIDATION:

1. **Validation set approach: (but underfit)**
   Split data evenly and roughly
2. **Leave one out Cross v: (but overfit)**
   Leave out 1 obs, train on rest
3. **k fold cross v: (balanced data)**
   Divide data in k folds, iterate leave out 1 to test, train on rest, average the error
4. **Stratitfied k fold cross v: (unbalanced data)**
   Split the data in k folds with similar distribution in each fold

Clustering
1. K-means Clustering
Puts the data into the specified k numebr of clusters

- First selects k random central points.
- Then assign each observation to nearest central point
- Then calculate mean of each cluster as new central point, then repeat until cluster no change
- Then find the variation within the clusters
- Repeat the entire process with different starting points and return the one with best results overall

How select k: Elbow method to find k just before the SS distances does not reduce much

2. Hierarchical Clustering
Finds pairwise what two observation are most similar. If need k clusters, cut the graph at where number of nodes = number of clusters

3. K-Nearest Neighbour for Classification

- Given a data point and k neighbours.
- k nearest points are selected using a specified distace measurement.
- Category with highest votes, will be used to classify the data point

Approx. Nearest Neighbour Variation
Searches only using a subset of candidate points variation of KNN, using a specified algorithm

In fact for ELK semantic search they use ANN with **Hierarchical Navigable Small World (HNSW)**

4. Support Vector Machines:
Cross v. finds the best soft margin
- **Soft margin classifier** = **Support vector classifier**
- **Support** = Point on the **Support vector hyperplane**
- Points within soft margin are missclassified

But with many misclassification (Non-linear data), we can apply a non-linear transformation (**higher dimension space)** to a feature, then find a **Support vector classifier**. However it is computationally extensive.

We extend to use **Support vector machines**
- In the **new higher dimension vector space**, we find the relationship **(dot product) between each pair of points** which is the **kernel function** and use that relationship value to determine the **Support vector classifier** (when brought down to current dimension space, is a **non-linear hyperplane**)
- Note we did not do any transformation/calculation of the data to any higher dimension

(CART) DECISION TREE: Partition predictor space

a) Regression Tree (Predict numeric data)
Prediction is given by the mean of the response values
Recursive binary splitting: Consider all predictors & select the cut with the lowest RSS

b) Classification Tree (Classify)
Given by the most commonly occurring class
Uses a measure of **impurity** instead of RSS
Lowest impurity feature will be closer to the top of the tree

- **Gini Impurity of Leaf: [0:0.5]**
  $= 1 - \sum_{k=\{classes\}}(p_k^2)$
- Total Gini = Weighted avg of the Gini Impurities,
  weight = fraction of observations/residuals captured
- For cont. features, calculate the avg value for each sorted adjacent value pairs
  Calculate the gini impurity for each pair to determine best split
- **Entropy: [0:1]**
  $-\sum_{k=\{classes\}} p_k log(p_k)$
- Taken from concept of surprise: $log(1/prob(x))$
  High probability = Low Surprise

GOOD: Good interpretability and visual, handles continuous variables
BAD: Not robust and accurate. Builds complex tree with **high variance** if no pruning

Cost Complexity Pruning/ Weakest Link Pruning:

$$min(\text{tree score}) = \min(\text{Total RSS/impurity} + \alpha|T, \text{No. of terminal nodes}|)$$

- Using kfold cv, for given fold, produce subtrees with a range of **α (complex. param)**
- Select the best subtree for each fold, then pick the value of **α** with the lowest validation RSS/impurity

Ensemble Learning: Combining weak learners

1) Bagging
TO REDUCE VAR, OVERFITTING
Generate B different **bootstrapped** training datasets
Avg the prediction (OOB prediction), for OOB MSE

Bootstrapping: Repeatedly sample observation from original **WITH** replacement until the original size

BUT if there are few strong predictors, most bagged trees will use similar predictors at the top of the split, trees are still highly correlated, hence variance is not reduced much

2) Random Forest: (Less interpretable)
TO REDUCE VAR, OVERFITTING MORE
At each split, use a random subset of predictors to build a forest of trees

3) Boosting
TO REDUCE BIAS, UNDERFITTING
The model is sequentially improved by minimizing errors made from the previous iteration. Aggregated prediction

Generate m subsets of training data random **WITHOUT** replacement**,** where each subsequent training data also includes the misclassified data points

4) XGBoost

- For each binary recursive split of a predictor space, starting from 0.5
- Calculate **similarity scores** for all nodes, with regularization parameter λ
  Reg: $\frac{\sum(\text{residuals/obs})^2}{\sum(1)+\lambda} = \frac{\sum(\text{residuals/obs})^2}{\text{No. of residuals}+\lambda}$
  Class: $\frac{\sum(\text{residuals})^2}{\sum[\text{prev prob}_i \cdot (1-\text{prev prob}_i)]+\lambda}$
  where the term to the left of λ is called the **cover, sum of hessians**
- **Calculate gain** to determine how to split the data
  $\text{Left}_{simi} + \text{Right}_{simi} - \text{Root}_{simi}$
- Higher the **gain** the better the split is at splitting residuals into a group of similar values
- If $\text{Gain} \leq \gamma$, **tree complexity parameter**, then prune it
  γ = minimum reduction in loss/impurity or gain in similarity scores

- Gradient boosting concept (Underfitting)

- The process of additively generating weak models is formalized as a GDA over an objective function
- **Simple example after mathematical operations**
- For cont. target label, use the mean as starting point
- Find the $\text{pseudo residuals} = \text{actual weight} - \text{starting point}$
- Build 1st tree to classify the pseudo residuals
  $\text{Leaf value} = \text{Average of pseudo residuals}$
- $\text{Prediction} = \text{Starting point} + \text{Leaf Value} \times \text{Learning rate}$
- With new pseudo residuals, repeat and each iteration the pseudo residuals should decrease

- Cache awareness access (Memory efficiency)

- **Gradients and Hessians** are cached to output similarity scores faster

- Parallel tree building (Speed efficiency)

- Utilizes **weighted quantile sketch**, which splits the large data & calculates the quantile in parallel to give an approximate quantile representation of data. Each bin will have same **sum of weights/sum of hessians**
- Uses it for each iteration, to find the best split among all features that results in largest gain in loss objective function
- Hence uses **approximate greedy tree learning** instead of level wise expansion at each split
- If dataset is small, just defaults to **greedy tree learning**, very fast

- Regularization (Overfitting)

- **L1/L2 reg** on the leaf scores
- **Learning rate** on GDA
- **Gamma** for pruning

- Handles missing + sparse data (Imperfect data)

- Utilizes **sparsity-aware split finding** by splitting the dataset into missing and non-missing values. During gain calculation, it places the non-missing values to the left and the right and selects option with higher gain

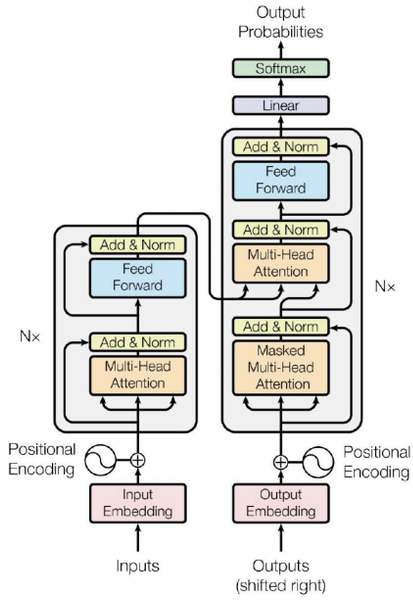
Softmax
$p_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$

Argmax
$\text{argmax}(p) = \underset{i}{\text{argmax}}\ p_i$

Common Activation Function

- **Sigmoid:**
  Sigmoid nonlinearity squashes real numbers to range between [0,1]
  $\frac{1}{1+e^{-x}}$
- **Tanh:**
  Tanh nonlinearity squashes real numbers to range between [-1,1]
  $tanh(x)$
- **Rectified Linear (ReLU):**
  0 when x < 0 and then linear with slope 1 when x > 0
  $max(0, x)$
- **Leaky ReLU:**
  0 or have a small negative slope (of 0.01, or so) when x < 0, then linear with slope 1 when x > 0
  $max(0.1x, x)$

Transformers



1. Tokenization into Input Embeddings + Positional Encoding

Inputs are tokenized into a vector space. Then the input vectors are mapped onto an embedding matrix that captures the semantics of the input vector.

E.g. Input sentence "Hi how are you?" with word tokenization into input vector [1,44, 23, 24] gets mapped onto an embedding matrix to [[0.213, 0.32, 0.34], [0.423, 0.234. 0.456], [0.69, 0.954, 0.43]] of dimension 3.

The semantics are based on its unique location in the vector space.

The embedding vector and the positional encoding are summed. By adding the positional encoding, the word order information is preserved and thus the relevance of the word's position in the sentence is maintained.

2. Self Attention

LLMs assign attention weights to different words based on their relevance to the current word being processed.

- Multiheaded Attention:

The Attention module splits its Query, Key, and Value parameters N-ways and passes each split independently through a separate Head of linear layers.

Note that there is still a single Q, K, V matrix, just that each head takes a logical section of the matrices.

Then the Attention scores are calculated using the Q, K and V matrices, which are then combined into a final Attention Score

Hence intuitively, each head can encode a specific relationship or nuances for the input.

3. Feed Forward NN to Single Linear Layer:

Then it passes through a simple NN, before compressing down to a final single layer.

4. Softmax

Then to get the final output, we have a softmax layer to convert the linear layer into a probability distribution on potential subsequent words or phrases, considering a series of input words

- **Greedy vs Random(weighted) sampling:**
  - Token with highest prob selected
  - Token selected using random-weighted strategy across the probabilities of token (E.g. 20% chance of this token)
- **top-k:** Select an output from the top-k results after applying the random weighted sampling
- **top-p:** Select an output from the top results with cumulative probability <=p
- **temperature:** The hotter the temperature, the flatter the probability distribution

Quantization: Summary

	Bits	Exponent	Fraction	Memory needed to store one value
FP32	32	8	23	4 bytes
FP16	16	5	10	2 bytes
BFLOAT16	16	8	7	2 bytes
INT8	8	-/-	7	1 byte

FLAN T5

- Reduce required memory to store and train models
- Projects original 32-bit floating point numbers into lower precision spaces
- Quantization-aware training (QAT) learns the quantization scaling factors during training
- BFLOAT16 is a popular choice

Prompt Engineering

- 0/Few Shot/Dynamic Inference inside the system prompt

Finetuning

- Datasets of q&a pairs
- But if on 1 task, will lead to catastrophic forgetting and reduction in ability in other tasks

- Overcome using **multi-task instruction fine-tuning**
- Many different system prompts, but requires many examples for training.E.g. FLAN model family

LLM Evaluation Metrics:

1. ROUGE:

- Used for text summarization
- Compares output to golden reference answer

ROGUE-1 Recall: unigram matches/unigram in ref
ROGUE-1 Precision: unigram matches/unigram in output
ROGUE-2: Bigrams
ROGUE-L: Numerator uses No. Longest Common Subsequence, Denominator uses unigram

Clipping: Unique matches only, else if match same words more will boost ROGUE

2. BLEU Score

- Used for text translation
-Avg precision across a range of n-gram sizes

Benchmarks:

Frameworks of tasks and metrics:

- **SuperGLUE** benchmark
- Holistic Evaluation of Language Models (**HELM**)
  - 7 metrics: Accuracy, Calibration, Robustness,
  - Fairness, Bias, Toxicity, Efficiency

Parameter Efficient Finetuning (PEFT)

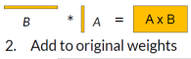
Full fine-tuning for each task creates multiple copies of the large LLM models, not efficient

1. Selective: Select a subset of initial parameters to fine-tune
2. Reparameterization: Repara. model weights using a lower-rank representation (LoRA, Q-LoRA, Ia3)
3. Additive: Add trainable layers to the model (Prefix, Prompt Tuning)

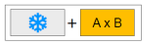
Low Rank Adaption of LLMs (LORA): Q is quantized version

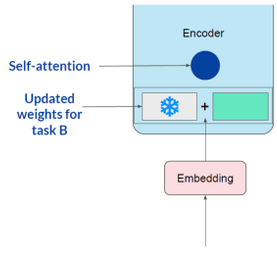
1. Freeze most of the original LLM weights.
2. Inject 2 **rank decomposition matrices**
3. Train the weights of the smaller matrices

Steps to update model for inference:
1. Matrix multiply the low rank matrices

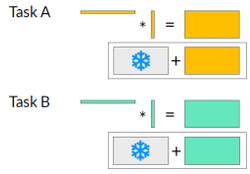


2. Add to original weights

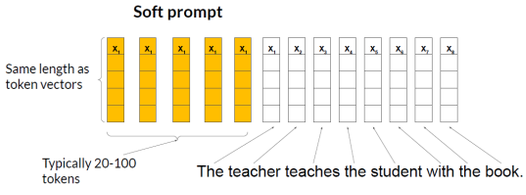




1. Train different rank decomposition matrices for different tasks
2. Update weights before inference



Prompt Tuning:



Prefix Tuning

Adding embedding layers to every layer that are trainable.

Encoder (auto-encoder, masked LM):
BERT, ROBERTA
Encodes the inputs with contextual understanding and produces 1 vector per input token

Objective: Reconstruct the text ("denoising)
Bidirectional context

The teacher <MASK> the student
The teacher {teaches} the student

- Sentiment analysis, NER

Decoder (auto-regressive/ casual LM):
GPT, Bloom
Accepts input tokens, to generate new tokens

The teacher ?
The teacher {teaches}

Objective: Predict the next token
Unidirectional context
- Text generation

Encoder-Decoder (Seq2Seq):
T5 Flan, Bart

The teacher <X> student
<X> teaches the

Objective: Reconstruct a span of tokens
- Translation, Text Summarization, Q&A

IA3 (Infused Adapter by Inhibiting and Amplifying Inner Activations)

- Employs learned vectors to rescale inner activations in a transformer-based model that are integrated into attention and feedforward modules.

This reduces the number of trainable parameters significantly compared to approaches like LoRA.

Embedding Matrix:
- Each column vector represents a token in the vocabulary
- Number of rows gives us the dimensions of the token

E.g. GPT 3: 50,257 tokens with 12,288 dimensions

Unembedding Matrix:
- Each row for each token in the vocabulary
- Number of columns gives us the dimensions of the token
- Transpose of the embedding matrix with regards to shape

Probability Distribution:
Due to matrix multiplication, the values of the final vector is not between 0 and 1.

Softmax: Takes the exponential of each value, to make all numbers positive, then divide by the total of the exponents

This normalizes the vectors, such that it sums to 1, a valid probablity distribution

Throwing in a constant T (temperature) to the denominator of the values before applying the exponent, results in a sharper probability distribution for larger T

The values before applying exponent is what MLs call logits

Dot Product:
Measure of how close two vectors are, positive if pointing in same direction, vice versa. 0 if perpendicular

Transformers Network:
Context Size: How many vectors the model can take

Idea:
Suppose we are at the last token to predict. The token before it, it's values have to be updated by all the attention blocks, encoding all the information from the full context window relavant to predicting the next word.

For example, if the token was the word "was", then it's values have to have been updated (direction of vector changed in the vector space) to better predict the next word.

Single Head Attention (Self Attention Head):

Initial embedding of each token, encodes its meaning of the token (word) and it's position (positional encoding)

Next for each i^{th} token, suppose we are producing this new vector called q_i. It is called the query vector, intuitively it is asking a question about the token. For example: "Any adjectives are there before me?". This Q vector space will have a much smaller dimension, such as 128.

To transform it to q_i, we need to multiple the i^{th} token vector by a weight query matrix, W_q. The weight query matrix, W_q is learned through data.

Then there is another sequence of vectors called the key vectors, k_i.
Initutively, for each token we can think of it as answering the queries. For example, "I am an adjective here".

If a pair of key and query vector match well (dot product very positive), we can think of it as the key answers the query very well. For example if token A key's vector has a high dot product value with token B query's vector. We say in ML terms that token A attend to the embedding of B

As a result we have a matrix which is a dot product of every key and query pair, which gives us a score of how each token (word) row is used to update the meaning of every other token column. It a single word, we are finding the relevance.

Since we are going to use these scores as weights, we want the matrix to be normalized. This is called an attention pattern.

In ML papers:

$$\text{Attention}(Q, K) = \text{softmax}\left(\frac{QK^T}{\text{Numerical stability factor,}\sqrt{d_k}}\right)$$

where d_k is the dimesion of the key query space

out-of-vocabulary (OOV) tokens:

1. **BERT (Bidirectional Encoder Representations from Transformers)**:
    - **Handling**: BERT uses subword tokenization, specifically WordPiece tokenization, which allows it to represent OOV words by breaking them down into smaller subword units.
    - **Example**: If an OOV word is "unrecognizable", BERT might tokenize it into ["un", "##recognizable"] where "##" denotes a continuation of a subword token.

2. **GloVe (Global Vectors for Word Representation)**:
    - **Handling**: Models like GloVe typically assign a zero vector to OOV tokens.
    - **Example**: If an OOV word is "onomatopoeia", GloVe might assign a zero vector to it.

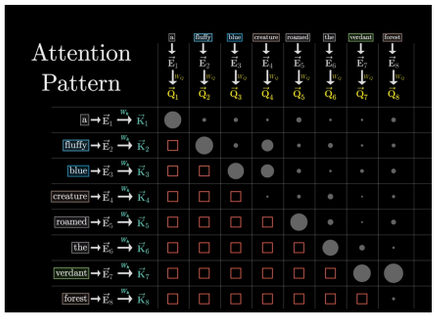
3. **Word2Vec**:
    - **Handling**: Word2Vec often assigns a special token like `<unk>` to represent OOV tokens.
    - **Example**: If an OOV word is "onomatopoeia", Word2Vec might represent it as `<unk>`.

4. **FastText**:
    - **Handling**: FastText also uses subword tokenization, specifically the n-gram approach, which can handle OOV words by breaking them down into character-level n-grams.
    - **Example**: If an OOV word is "onomatopoeia", FastText might represent it as ["ono", "nom", "oma", "top", "ope", "ia"].

5. **ELMo (Embeddings from Language Models)**:
    - **Handling**: ELMo uses character-level embeddings as part of its modeling approach, which allows it to handle OOV words by constructing embeddings based on character-level information.
    - **Example**: If an OOV word is "onomatopoeia", ELMo might construct its embedding based on characters "o", "n", "o", "m", "a", "t", "o", "p", "o", "e", "i", "a".

6. **Transformer-based models (e.g., GPT, GPT-2, GPT-3)**:
    - **Handling**: These models can handle OOV tokens in various ways, such as using a special token like `<unk>`, or by assigning a random initialization vector.
    - **Example**: If an OOV word is "onomatopoeia", a transformer model might replace it with a special token `<unk>`.

Each of these models has its own way of dealing with OOV tokens, which is typically specified in their documentation or implementation details. These approaches ensure that the model can still process and tokenize input text even if it contains words that were not seen during training.



where d_k is the dimesion of the key query space

Then the size of QK^T will be the square of the context window

The models during training simultaneously predict every possible next token following each initial subsequence of tokens.

For example, if the model is currently training to predict "What is the next ____?" is effectively acting as many training examples, as it also predicts every subsequence of tokens: "What ___, What is ____, What is the ____, etc".

However this implies that later words should never influence the weights of earlier tokens. Hence in our attention pattern matrix, we need the subdiagonal values (bottom half) to be 0. To do so, before we do softmax, we force the values to negative infinity as $e^{-\infty} = 0$, this is called masking. In some cases, we don't mask, but for GPT-3 we do this masking.

Next we need to update the embeddings of the tokens to cause a change in the embedding values of other tokens. For example the token: fluffy should influence the token: creature to "move"/transform the vector of creature towards a place in the embedding space that better represents a fluffy creature.

Of course, the straightforward way is using another matrix multiplication which another matrix, V or the value matrix.

For example: "blue fluffy creature". The position of creature will get shifted by adding the resulting vectors of $W_V\text{fluffy}^T$ and $W_V\text{creature}^T$

Then for instance from our atttention pattern, we know words fluffy and blue have high values inside the QK_T matrix of their respective Q and K dot product to the word creature.

We just scale each value of that by the resulting vectors of $W_V\text{fluffy}^T$ and $W_V\text{creature}^T$. Then sum up the weights for creature, which gives us the resulting shift for the creature word. Do that for the entire sequence, we get 1 head of attention

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\text{Numerical stability factor,}\sqrt{d_k}}\right)V$$

So parameters wise, now we have
- $K_{\text{key dimension by context window}}$, $K_{\text{128 by 12,288}}$
- $Q_{\text{key dimension by context window}}$, $Q_{\text{128 by 12,288}}$

Then by right it could be:
- $V_{\text{context window by context window}}$, $V_{\text{12,288 by 12,288}}$
but that is too many parameters, so weuse a low rank adoption with 2 lower rank matrices:
- $V_{\text{key dimension by context window}}$, $V_{\text{128 by 12,288}}$
- $V_{\text{context window by key dimension}}$, $V_{\text{12,288 by 128}}$

For cross attention, the models involves process of 2 distinct types of data, e.g. 2 languages. Then for Q and K, each corresponds to each langauge. Since there is no notion of later tokens affecting earlier ones, there is no need for masking.

Now we know that there any many context to capture, for example associations/attention maps like grammar, movie names etc

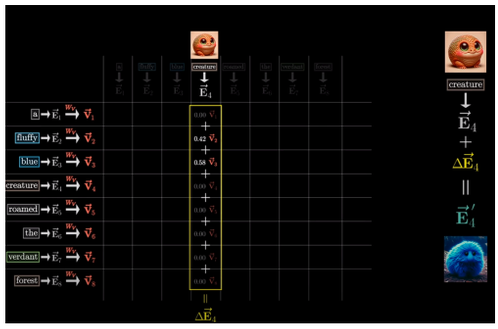
The contextual updating really depends on the task at hand, and we really cannot intepret them.

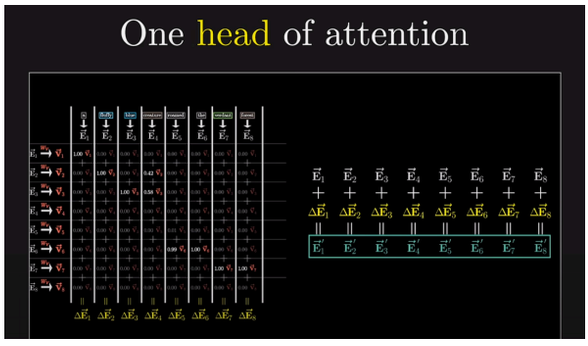
A full attention head contains multiple distinct key-query and value maps running in paralllel: e.g. GPT-3 has 96 attention heads inside this multi-headed attention block.

Actually, the value matrix which we split into 2 is different in real implementation.

These $V_{\text{context window by key dimension}}$, $V_{\text{12,288 by 128}}$ for each attention head is actually stiched together to a large ouput matrix, O associated with the entire attention block.

Now in a transformers there are multiple blocks, for example for GPT-3 we have 96 blocks also.





STATISTICS: t-test if population variance not known, else z-test

The choice of 1 or 2-sided test depends on the **PROBLEM, NOT the sample data**

i) z test: Sample from a normal distribution.
If the sample size is large, CLT ensures normality (else use a normal QQ plot to determine)

ii) t test: Sample deviation is known, n-1 degree of freedom

- **Type I Error**: Reject H_{null} when its true in reality, α
- **Type II Error**: Fail to reject H_{null} when it's false in reality, β (1-β = Power)
- **p-value**: Strength of evidence against H_{null}
  Probability of observing the value of the test statistic given H_{null} is true
  Reject H_{null} if **p-value <= alpha**
- **CI:** Range of plausible values for the **parameter** based on the sample data such that we do not reject the H_{null}

AB TESTING

1. Problem Statement:
Question the interviewer on the problem

2. Hypothesis Testing:
State null and alternative hypothesis

Success metrics must be **measurable**, **sensitive** and **timely**

H: appines, how satisfied are users
E: gangement, how engaged are they in the product
A: cqusition, how many new users
R: etention, what are daily active users
T: ask Success, how long to complete a task in the product

3. State values (To determine sample size)

- **Alpha:** 0.05, P(type I error|given ...): Threshold for rejecting H_{null} given H_{null} is true
- **Power:** 0.80, Sensitivity of the test to detect an effect exists given H_{alt} is true
- **MDE:** 1-5% lift/ prac sig, min effect size (diff in parameters) as detection of a difference between control and treatment group

Lower alpha (more stringent), stronger evidence required, **larger sample size**

To meet a **greater power**, **larger sample size** needed to meet the increased sensitivity

Greater MDE implies lower minimum effect size, hence **smaller sample size** needed

p-value not significant, but there must be an effect, so increase power to increase sensitivity

4. Run the experiment

Duration of experiment: Based on the domain

Ensure the pipeline runs through the entire duration and avoid peeking at the p values, for statistical integrity. As some might stop at desired p-value, each check is technically another test. Hence the probability of finding a false positive increase.

5. Validity Checks:
Ensure no bias
- From instrument effect
- External factors (e.g. holiday, competitors)
- Selection Bias (Determined using A/A test)
- Sample ratio mismatch (Determined using chi-squared test)
- Novelty effects (Initial excitement or curiosity from the treatment group, hence maybe segment new from old users)

6. Interpret and Launch
Talk with business users, about cost vs other metric tradeoffs

Process of Designing Red Flags:

- **Data Analysis:**
  Analyze historical transactional data to identify patterns and potential risk scenarios.
- **Risk Identification:**
  Identify specific risk scenarios that may indicate compliance issues.
- **Hypothesis Generation:**
  Formulate hypotheses on red flags that can detect these risk scenarios.
- **Implementation:**
  Implement the red flags into the monitoring system.
- **Testing and Validation:**
  Conduct rigorous testing and validation, including A/B testing, to evaluate red flag performance.
- **Iteration and Optimization:**
  Refine and optimize the red flags based on test results and feedback.

How to improve 100k data mining

1. Data Cleaning:

- Detect and handle outliers
  (E.g. data outside 1.5 times IQR range, using Box whisker plot)
- Utilize data validation
  (E.g. sweetviz automated EDA in python, Data governance platforms like Collibra with data attributes)
- Checking back with the business user

2. Feature Selection:

- Feature importance techniques built into models like Gradient boosting
- Using regularized logistic regression (LASSO, L1)
- Using PCA

3. Sampling:

- Stratified sampling, to ensure balanced representation from different groups
- Cluster sampling or even bootstrapping (bagging) sampling

4. Parallel Processing:

- Utilize distributed computing frameworks such as Apache Spark, pyspark wrapper in Python which has inbuilt parallelization
- Store as HDFS
  - **Scalability**:
    HDFS can handle large volumes of data and easily accommodate dataset growth without performance degradation.
  - **Fault tolerance**:
    HDFS replicates data across multiple nodes, ensuring data availability and reducing the risk of data loss.
  - **Data locality**:
    HDFS stores data in a distributed manner, minimizing data transfer over the network and improving performance by maximizing data locality.
  - **Integration with big data ecosystem**:
    HDFS seamlessly integrates with other big data tools and frameworks, allowing for the use of scalable and distributed data mining algorithms.

5. Tools

- Utilize workflow management tools like Apache Airflow, MLFlow, Luigi to automate, schedule processes in batches
- Even workspaces like CDSW
- Utilize proper data pipeline frameworks like Kedro for instance

6. Documentation and Reproducibility

Engagement Metrics:	**Revenue**	**User Experience Metrics:**
- Click-through Rate (CTR)	- Return on Investment	- User Satisfaction
- Engagement Rate	- Customer Lifetime Value	- Response Time
- App Usage Frequency	- Incremental Revenue	- Personalization Effectiveness
- App Session Duration		- Opt-out Rate
- Time-on-Page	**App Store Ratings**	- Message Deliverability
- Message Interactivity	- App Rating Improvement	- Device Type Effectiveness
- In-App Purchase Rate	- Customer Acquisition Cost	- Personal Data Consent Rate
- Social Media Mentions		- Notification Opt-in Rate
	Retention Metrics:	
Conversion Metrics:	- Retention Rate	**Response Metrics:**
- Conversion Rate	- Churn Rate	- Response Rate
	- Uninstall Rate	- Time-to-Action
	- Subscriber Growth Rate	- Geographic Targeting Efn.