**Discrete Key Value Bottleneck**

**Abstract :**

At an introductory level , neural networks perform very well on classification tasks when the data is available all at once (i.i.d i.e independently and identically distributed). However these very do not perform as well when the input data is not available all at once i.e non stationary training data.   
  
There are many common real life cases where the training data changes continuously thus deeming this research topic to be of immense importance. For example : **A medical diagnostic system might be trained incrementally as new diseases are discovered or new data from medical research becomes available.**

An early solution was pre training the encoders on large amounts of readily available data and fine tuning the encoders based on the task at hand. The two main issues faced with this was the sheer **computation capacity** required to update the large number of weights, secondly , since all the weights were being updated based on the new info it gave rise to the issue of **catastrophic forgetting** where the model forgets all previously learned information when trained on new information.

This paper proposes a novel solution by building a discrete bottleneck of key-value pairs which falls in between the encoder and decoder. The overview of this process involves **feeding the input to a pre-trained encoder** after which the output of the encoder is used to **select the nearest keys** and the corresponding values are **fed into the decoder** to finish the task.

The main objective of this paper is to establish Discrete Key Value Bottlenecks as an effective solution to the issue put forward by non stationary data streams.

**Research Question :**

The question which the research paper tries to address is

*How can neural networks effectively learn from non-stationary data streams without suffering from catastrophic forgetting, while minimizing computational costs associated with frequent weight updates?*

Learning is inherently an incremental process. Despite the proficiency of current models in learning from data all at once, the need to relearn everything from scratch whenever the training data changes is a matter of huge concern. To mimic natural learning, it is crucial to develop models capable of learning from new data without forgetting previously acquired knowledge.

**Proposed Solution :**

The paper introduces a novel architecture which consists of three main components:-

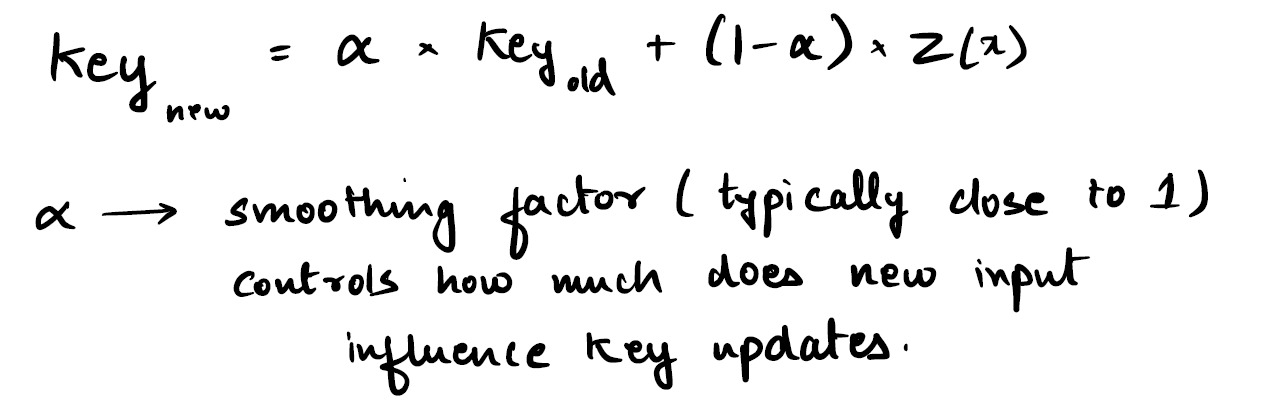
1. The **Encoder** (Pre Trained)
2. **Discrete Key Value Bottleneck**
3. The **Decoder**

The end to end process involves the following steps :

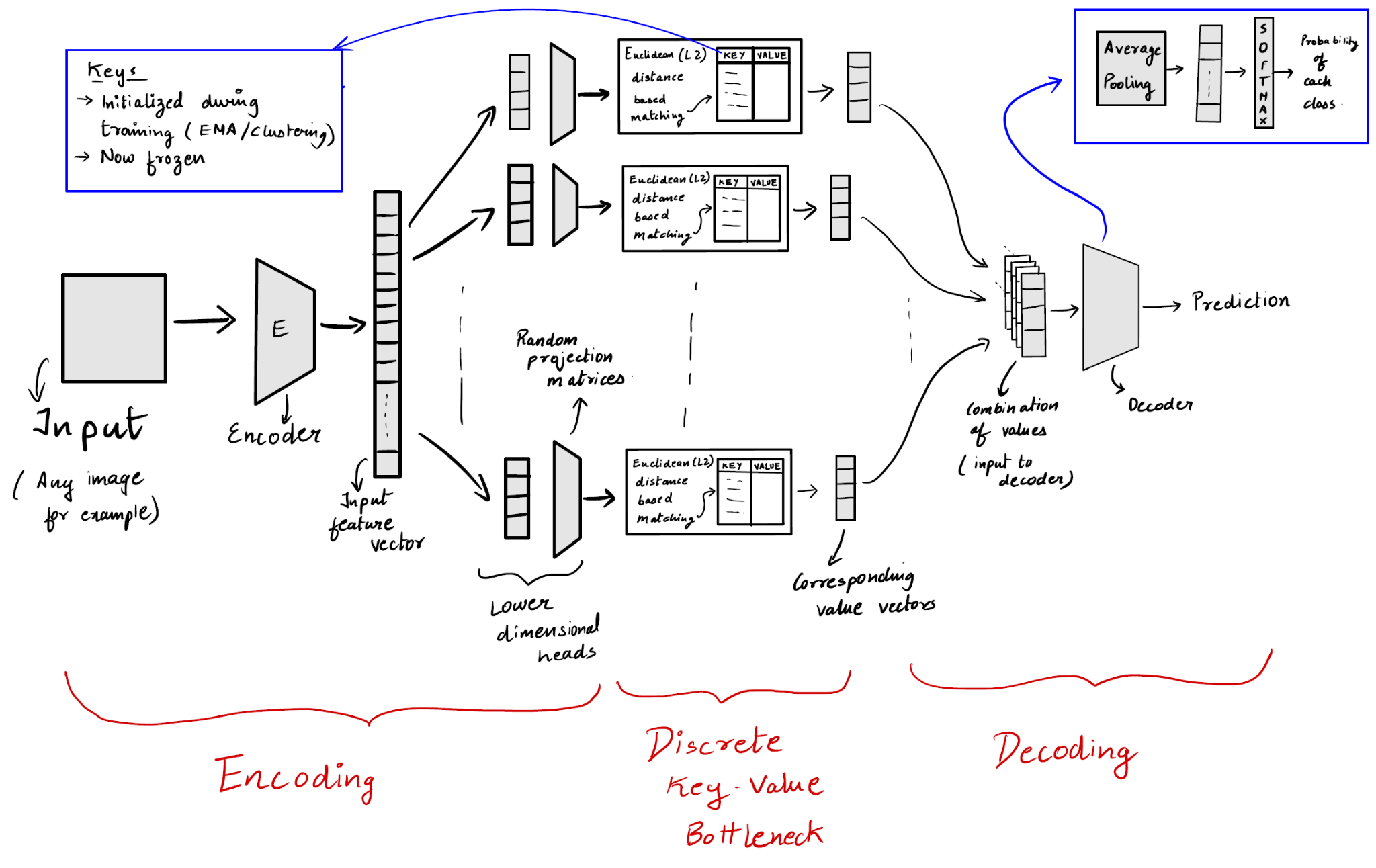
1. The input is fed into the encoder , giving us a **feature vector** for that input, That is then projected onto **X lower dimensional heads** through random projection matrices which are initialized with the help of the Gaussian random distribution. These matrices are used for extracting out particular features of the input vector.   
   For example , different heads are responsible for picking out shape , color etc.
2. Post this encoding process there are pushed into the discrete key value bottleneck where two things take place
   1. Each head is matched to the closest key. This is done by calculating the **L2 distance or Euclidean distance.**
   2. Once the above is done, a simple lookup is implemented in the key value mapping to extract the corresponding value to the matched key.
3. Once the corresponding values are fetched , a prediction is carried out. This prediction is done by the decoder which can work on a variety of functions. For the experiments carried out in the paper , they have used a relatively simple non parametric decoder function which utilizes a combination of average pooling and softmax to the output to give the prediction.

**Important information related to key initialization and key-value pairs**

1. The **keys are frozen** , only the values change during backpropagation. This helps to ensure previously learned info isn’t overwritten and forgotten.
2. The keys could be initialized **randomly** to begin , but this approach would not capture the details of the dataset efficiently. A better approach would be one that is **data driven**. The keys can be initialized using a **separate dataset** (Auxiliary dataset) that is related to the domain of the target data but does not overlap with it.
3. For instance, in the paper, they use **unlabeled data from CIFAR-100** for initializing the keys when working with a class-incremental CIFAR-10 task.
4. The input data from the auxiliary dataset is fed through a **pre-trained encoder** to obtain feature representations. These feature representations provide a more structured view of the input space than random initialization.
5. As feature representations are obtained from the auxiliary dataset, they can be used to update the keys incrementally using an **EMA (Exponential Moving Average) approach**. Given below is the formula for EMA key initialization.



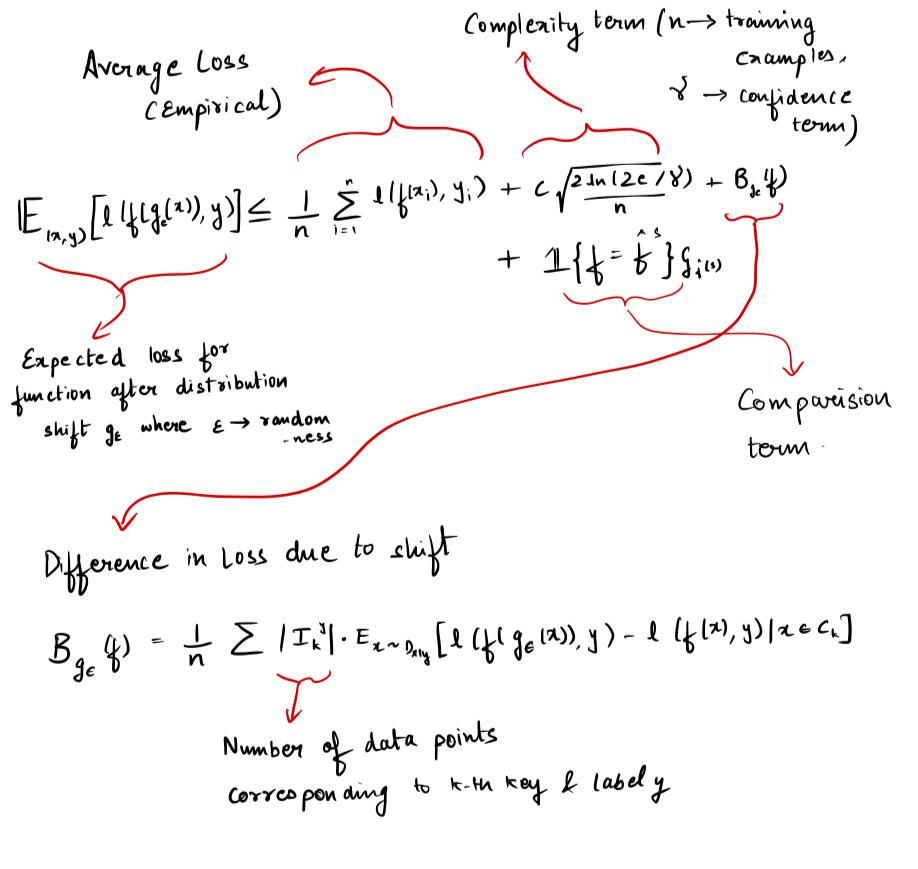
1. Each head has their own set of key value mapping referred to as the codebook.
2. In practice , a combination of the values from different heads is employed to make a prediction.



**Theoretical Analysis :**

This part of the review , deals with proving that this model has the ability to reduce the generalization error of the base model from a theoretical point of view.

This view is substantiated by a theorem whose proof has been given in the appendix of the paper.



Finally a short note on the **parameters** which affect the performance of the bottleneck. These were obtained from the **ablation and sensitivity analysis** carried out in the paper.

1. Dimension of the key codes
2. Number of key value pairs
3. Number of codebooks
4. Key initialization dataset.

**Potential Improvements :**

1. **Dynamic Allocation of key value pairs** 
   1. If the model detects that a particular region of the input space requires more detailed representation, it could allocate additional key-value pairs to that region. Conversely, for simpler regions, it could reduce the number of key-value pairs, thus optimizing memory and computational efficiency.
      1. One possible way of doing this is to track **key usage frequency**. If some keys are rarely used, it may indicate that certain regions of the input space are less important or less populated.
      2. As new tasks or data are introduced, the model can analyze the data distribution to identify regions in the feature space where more detailed representation is needed (e.g., where the data is more dense or complex). One possible way to do this is :
         1. **K-Nearest Neighbors (K-NN) Density Estimation**: For each new data point, calculate the distance to its **K nearest neighbors**. If the distances are small, the data is densely clustered in that region. Thus they might require more key value pairs in that region
2. **Adaptive Key Updates** : In the current Discrete Key Value Bottleneck approach, the keys are frozen once initialized, meaning they remain fixed during the entire training process. Changing the keys by a small factor overtime could hold a solution to improving the working.
   1. An **adaptive key update mechanism** involves gradually updating the keys over time based on the new data encountered during training, rather than keeping them completely frozen. This can help the keys better capture changes in the input distribution without drastic shifts. One way of doing this would be to use a slow moving average i.e a very low learning rate.   
      The condition for key updates would be a topic for discussion. High access frequencies would be an essential factor in my opinion.

**Citations :**

1. [arXiv:2207.11240v3](https://arxiv.org/abs/2207.11240v3)
2. Google

**Declaration** :

All the codes and the reproduction of experiments have been done on my own based on my understanding of the paper. My only sources are the paper and information provided by Google and ChatGPT.

**Chetan Maheshwari**

**4th Year Undergraduate**

**Department of Electronics and Communication**

**National Institute of Technology , Karnataka**

**Pin Code : 575025**

**India**