Discrete Key-Value Bottleneck (Trauble, Goyal et al)

Shuvraneel Mitra

Indian Institute of Technology, Kharagpur

October 23, 2024

Introduction and Context

- Deep neural networks perform well on classification tasks where data streams are i.i.d. and labeled data is abundant. Challenges emerge when the training data stream is non-stationary, i.e. distribution shifts in the training data.
- One approach is to pre-train large encoders on volumes of readily available i.i.d. data and followed by fine-tuning. However this is prone to fall prey to catastrophic forgetting as a large number of weights need to be fine-tuned so information about the previous task is forgotten.
- Building upon this pre-training paradigm, we introduce an approach to distil information into a discrete set of code pairs, each consisting of a coupled key and value code.

Encode, Bottleneck, Decode

- GOAL: Learn a model $f_\theta : \mathcal{X} \to \mathcal{Y}$ from training data $S = ((x_i, y_i)_{i=1}^n)$ that is robust to **strong input distribution changes**.
  Let the model be formulated as

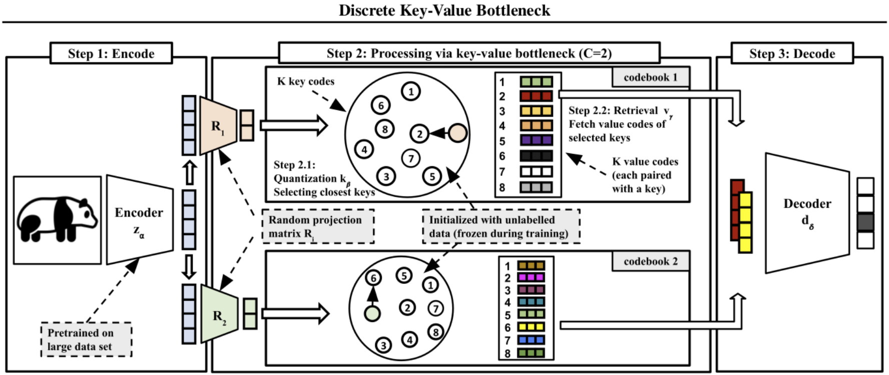
$$f_\theta = d_\delta \circ v_\gamma \circ k_\beta \circ z_\alpha$$

- In the first step an input is fed to the encoder $z_\alpha : \mathcal{X} \to \mathcal{Z} \in \mathbb{R}^m$ which extracts a representational embedding from the high-dimensional observation x.
- We further project this representation into C lower-dimensional feature heads using **fixed** Gaussian Random projection matrices, each of them being passed as input into a *separate head-specific learnable* key-value codebook.

Continued...

- A **KEY-VALUE CODEBOOK** is a bijection that maps each code vector to a different value vector which is learnable.
- Within each codebook, a quantization process k_β selects the closest key to its head-specific input.
- For the purpose of classification the authors propose a simple non-parametric average-pooling to calculate the element-wise average of all the fetched value codes and then apply a soft-max.

Overview Diagram of the proposed model

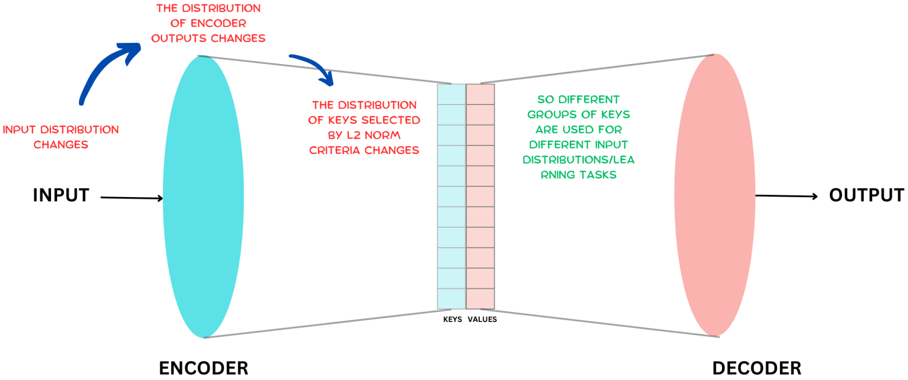

Discrete Key-Value Bottleneck
Step 1: Encode
Encoder
z_α
Pretrained on large data set
R_1
R_2
Random projection matrix R_i
Step 2: Processing via key-value bottleneck (C=2)
K key codes
1
2
3
4
5
6
7
8
Step 2.1: Quantization k_β Selecting closest keys
codebook 1
codebook 2
Step 2.2: Retrieval v_γ Fetch value codes of selected keys
K value codes (each paired with a key)
Initialized with unlabelled data (frozen during training)
Step 3: Decode
Decoder
d_δ

Training under co-variate shifts

Assume we are now given training data $S = ((x_i, y_i))_{i=1}^{n}$, but the data generation process might undergo various shifts of P (X) **without knowing when they occur**.

- Having the encoder and decoder solely connected through the key-value codes, we can now train the model on this data stream by **only updating the value codes**.
- Since we only locally update the actual retrieved values under the input distribution changes, the values from other data domains remain unchanged, thereby enabling *localized, context-dependent model updates*.

Visualisation of the meta-idea behind the Discrete Key-Value Bottleneck


THE DISTRIBUTION OF ENCODER OUTPUTS CHANGES
INPUT DISTRIBUTION CHANGES
THE DISTRIBUTION OF KEYS SELECTED BY L2 NORM CRITERIA CHANGES
SO DIFFERENT GROUPS OF KEYS ARE USED FOR DIFFERENT INPUT DISTRIBUTIONS/LEA RNING TASKS
INPUT
OUTPUT
KEYS
VALUES
ENCODER
DECODER

Potential improvements

1. Explore what happens when we make the random projection matrix learnable.
2. The random projection matrices can be forced to be sparse for faster matrix multiplication and memory savings.
3. Pruning the unused keys should only be done when it is known with 100% probability that all possible classes have been covered in the current incremental-training data stream.

Code implementation

- The code, written from scratch, can be found on GitHub: https://github.com/ShuvraneelMitra/Discrete-Key-Value-Bottleneck.

- The code is written completely in PyTorch, with Jupyter Notebooks being used for implementation of experiments.

A Simple Class-Incremental Learning Setting

- We consider a 2D input feature classification problem for 8 classes, where the training data is not i.i.d. but changes over four stages.
- In each stage, we sample examples of two classes and then move on to two new classes.
- Naive approaches like a vanilla linear probe and a one-layer MLP over-fit on the most recent training data. This behaviour is because we are optimizing all weights on the most recent training data.

Solving the problem using the bottleneck

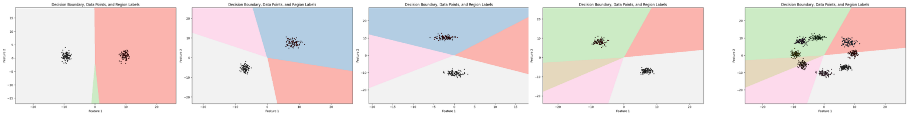
- After keys are initialized, they are frozen, and we train the model on the non-stationary data stream using the same loss and optimizer as before but optimizing the values only.

- The proposed bottleneck mechanism will now solely update the input-dependent value codes. This enables the model to incrementally update its predictions on unseen input domains and minimize interference.

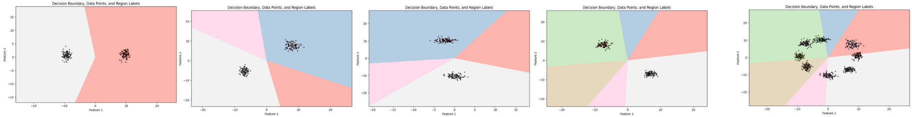
- Finally, we can prune all key-value pairs that were never selected in order to remove any dormant key-value codes.

Continued...

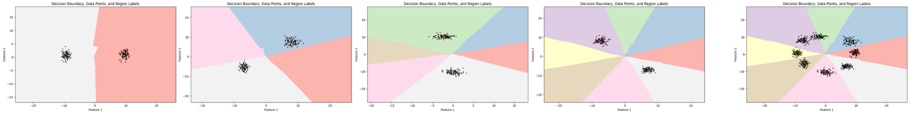
Results from a Linear Probe



Results from a Simple MLP



Results from the Discrete Key-Value Codebook



Real-life data experiments

We reproduce experiments on four publicly available backbones including

1. ResNet50 pre-trained on ImageNet
2. ViT-B/32 pre-trained with CLIP
3. ResNet50w2 pre-trained with SwAV as SSL method
4. ResNet50 pre-trained with DINO as SSL method

Unfortunately, due to lack of compute power the experiments were not able to run to completion. However, it has been ensured that the code runs perfectly without any errors.

Thank You