**20/04/2025 Report**

# Technical Report for TransformerAE Training Process for weight/-task merging

**Aymen Tlili**[a,1]

[a]*Faculty of Sciences of Sfax*

Supervising Professor:Bassem Ben Hamed

**Abstract—**In This report, I elaborate the methodology used to train our Transformer Architecture for Task Merging via weight regression or sequence-to-sequence modeling.as well as observations and specificities of using this task in the context of continual learning with the ultimate aim being uncovering what constitutes learning in a neural net and what's benign noise .

**Keywords—***LaTeX Computer Vision, Hyper-representation, AutoML, MNIST, Continual learning , Sequence modeling*

## 1. Introduction

Eminating from needs based on continuous learning and fast DevOps operations that require adapting models to an ever-flowing amount of data: The task of keeping a model up-to-date with current patterns and trends without losing previous knowledge and generalization power is a major challenge as the hardware cost ,time investment,expertise needed and the environmental reprocussions of training good-fit models that are able to perform both in and out of the learning dataset distribution .

Based on existing hypotheses that neural networks are just a compact format of the representations possible for a dataset and admissable conjecture about the universal approximation theorem benifits from repeating certain challenging examples that lie on the decision boundary more than easy-to-fit ones, We opt to focus on a subclass of deep neural networks known as convolutional NNs .These models are moderatly adept at resolving tasks like classification/regression or unsupervised context ones and with such diversity we fix a singular architecture and vary the weights and biases are that supposed carriers of information about the geometry of the learned manifold.

Previous Hyper-representation works focused on denoising parameters and strategically sampling a hidden representation of these CNNs from a meta-learner Transformer and then finetuning them into functional learners but used a set of CNNs that are denoted 'model zoo' which were only exposed to data superficially hence underfitting caused bad Accuracy metric results on classification tasks.The latent representations in the hidden neural network are usually the most spectrally compact and filled with informative features critical to the output's quality.In the transformer AutoEncoder used this was the (bottle-)neck represented as a dense Layer relating a downsampling encoder fed poor quality CNNs and it's mirroring upsampling decoder which produced the series of positionally tokenized denoised CNN parameters.

This model zoo is ill-fit for our task of Merging knowledge between 2 CNNs to produce a 3rd CNN with same number of parameters as that zoo's models are no better than coin flippers whereas Continual-learning assumes the input models have a task-incrimental or class-incremental setting or even same tasks but with concept drifting features i.e time-varying distribution making old models useless against new tasks.This is quickly remedied by finetuning with fresh/new data but the compromise with that is sacrificing knowledge from earlier model version and expensive-to-store older models and giving rise to the need to resort to one or hybrids of 3 categories of strategies to combat catastrophic forgetting and maintain old representations in new models :

- Regularization methods: L1 constain new models' parameters to not stray too far away from their counterparts in older models.
- Replay methods :save Information-rich/Class-representative examples or even GANs to produce old data on demand
- Architecture-incrimental methods :Keep an adapter/task-specific layer stored

## 2. Contributions:

- Show the effect of the loss function choice on the latent representations performance just after Inference and after finetuning [continuation of Layer-wise-Loss normalization proposed in hyperrepresentation paper]
- Propose a new CNNs Zoo best-fit for Class-incremental learning with proficient learners with high accuracy ceilings [unlike hyper-representation paper , have good accuracy and trained for 40 epochs with early stopping vs 5-10-25 epoch weak learning rate models]
  - Merging knowledge
  - Substracting knowledge(model unlearning)
  - minimize the reliance on real data (Federated Learning application)
- Showcase the **topological/spectral differences** in the presence or absence of certain classes/class combinations and derive optimizations methods from them (currently using loss terms)
- Clearly Seperate the topological(PH/in-layer distribution) and spectral(eigenvalue/FFT distribution) that allow fintuning to reach in 200-500 steps of the first pair of epochs high performance between Transformer predicted weights and fintuned/ground-truth models.
- Find What properties are maintained between the ground truth and finetuned models that we consider intrinsic for performance
- See what eventual combination or singular Loss best expresses what the meta learner should select in a sequence of weights
- Eventually sparsly feed only necessary weights to achieve a certain task and maintain generalisability across out-of-distribution classes
- Eventually use KANs to explicitly derive the formula between predicted and finetuned models

Good Performance is rated across multiple facets for what constitutes Learning

- CNN accuracy percentage after prediction IID and OOD
- CNN accuracy percentage after finetuning IID and OOD
- How fast the predicted CNN catches up to it's **parents'** accuracy
- How stable the predicted CNNs Accuracy is after each epoch of training (no deterioration as it continues to train)
- Transformer Loss value and how well it's correlated to the CNN performance
- How Clearly seperable are Classes [do we need to introduce something like diffusion-aware bernoulli mask to avoid the model giving the same output regardless of input? if this prediction's collapse to a singular model is benign , how good is that model ? ]
- Continual learning metrics like forgetting(loss of acc on old) and forward-learning(how older classes [presence/Absence or order of appearance] contributed to making future classes easier to learn

## 3. Dataset of weights (name pending):

### 3.1. Setting:

We define the images $I_C$ and their respective labels $Y_C$ with $Y = [Y_0, Y_1, .., Y_9]$ corresponding to the specific Classes $C_i$ from the MNIST Dataset. The following sub-section elaborates how we fit a convolutional neural network model $CNN^{[Y_C]}_{[g,i,e]}$ hence having weights $w^{[l,Y_C]}_{[g,i,e]}$ with $\mathbf{l}$ being the index of the Layer in our 3-convolution Layers 2-dense Layers fixed architecture model. We limit this work temporarily to working with kaiming uniform initialization zoo as it had a good median of results and least outliers during training

| Possible values | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| activation A | gelu | relu | silu | leakyrelu | sigmoid | tanh |
| checkpoint Epoch e | 11 | 16 | 21 | 26 | 31 | 36 |
| initialization i | xavier uniform | xavier normal | uniform | normal | kaiming normal | kaiming uniform |

meta.ipynb | Merged zoo.csv | wslayn 1.0.0.ipynb

Delimiter: ,

| | label | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | gelu | relu | silu | tanh | sigmoid | leakyrelu | weight 0 | weight 1 | weight 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | [0, 1] | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.094684735 | -0.7905626 | -0.4464047 |
| 2 | [0, 2] | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.0496985 | -0.061642155 | -0.09269891 |
| 3 | [0, 3] | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.48978 | -0.005564362 | -0.105482616 |
| 4 | [0, 4] | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.29166335 | 0.5139673 | -0.07567718 |
| 5 | [0, 5] | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.24452844 | 0.029974932 | 0.52706844 |
| 6 | [0, 6] | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.0375004 | -0.4844076 | -0.27842575 |
| 7 | [0, 7] | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.052341357 | -0.064843126 | 0.19656326 |
| 8 | [0, 8] | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.53858924 | -0.33099806 | -0.19575508 |
| 9 | [0, 9] | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.44759452 | 0.30329853 | 0.07768698 |
| 10 | [1, 2] | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.059722804 | 0.360608 | 0.35631633 |
| 11 | [1, 3] | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.04179168 | -0.21159662 | -0.52923477 |
| 12 | [1, 4] | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1668181 | 0.41012523 | -0.24427313 |
| 13 | [1, 5] | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.014314616 | -0.08713338 | 0.21204941 |
| 14 | [1, 6] | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -1.000273 | -0.22925067 | -0.32118472 |
| 15 | [1, 7] | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.21473292 | 0.23508567 | -0.26331523 |
| 16 | [1, 8] | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.23771928 | -0.1590095 | 0.27912033 |
| 17 | [1, 9] | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.21042645 | -0.08642456 | 0.22186308 |
| 18 | [2, 3] | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -0.09061946 | 0.1488133 | 0.47424597 |

**(a)** 36468 row/model Kaiming Uniform init zoo early columns and Experience identifier label 1-hot encoding

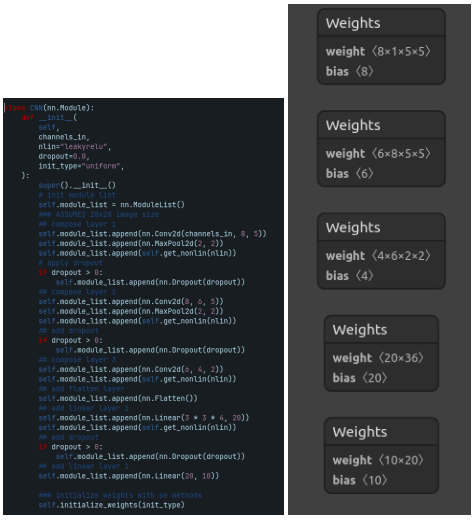
| bias 2462 | bias 2463 | Accuracy | epoch |
|---|---|---|---|
| -0.13655047 | -0.32772434 | 98.64333333333336 | 36 |
| 0.19706774 | -0.36267012 | 98.90666666666668 | 36 |
| 0.088529125 | -0.2061299 | 98.37666666666668 | 36 |
| -0.20371349 | -0.28388706 | 98.91166666666666 | 36 |
| 0.08827021 | -0.0042240657 | 98.86833333333334 | 36 |
| -0.085808694 | 0.11616701 | 98.755 | 36 |
| -0.237271 | -0.04033807 | 98.84666666666668 | 36 |
| -0.45016676 | -0.29142836 | 98.79 | 36 |
| -0.14843176 | -0.29648837 | 98.67333333333332 | 36 |
| -0.07278538 | -0.06793964 | 98.685 | 36 |
| -0.5364756 | -0.0009983338 | 98.55 | 36 |
| -0.022639548 | 0.062193777 | 97.725 | 36 |
| -0.47954503 | -0.216975 | 98.53666666666666 | 36 |
| -0.24637736 | -0.040949624 | 98.51166666666668 | 36 |
| -0.6218089 | -0.33946937 | 98.85166666666667 | 36 |
| -0.35365957 | -0.37145048 | 98.52 | 36 |
| 0.2798944 | -0.19171506 | 98.90833333333332 | 36 |
| -0.45934552 | -0.5223393 | 98.695 | 36 |

**(b)** Last columns and accuracy and the epoch identifier

**Figure 1.** The .csv format opfted for in our zoo

### 3.2. Training:

We train every subcombination possible of CNNs classification on 10-label MNIST Dataset to clearly target the effect of the presence of a class or it's absence i.e models operating on 2-10 classes.We vary the initialization distribution and activation used in the weights for diversity and robusteness of the learned models.That being said , we fix the intial seed, making models sharing the same init have the same exact sampling first origin checkpoint which will vary eventually on classes it's exposed to.




**(a)** Class CNN declaration[1]

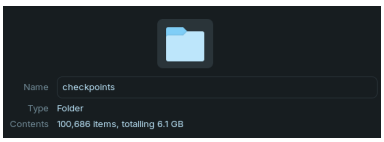
**(b)** Layer declaration Weights and channels

**Figure 2.** Showcasing the dataset checkpoints before building .csv format [1].

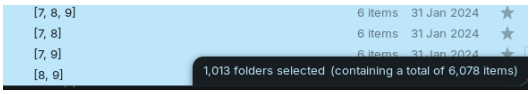
The file 'Silu.py' shows an example of a script we let run on the same remote server.on other machines we change the activation in the activation list.The training is done until convergence and satisfaction conditions are met.

- 40 epochs of training are originally declared
- checkpointing happens only every 5 epochs after the 10th and only if Validation-set accuracy for that **Experience** (set of classes chosen).
- Early stopping rules :A stagnation counter increments until it meets a patience parameters. If model validation accuracy 'stagnates' for 3 epochs. stagnation is defined as staying within a range of a Margin 0.05% accuracy from the last epoch's accuracy.
- The early stopping means we can have CNNs scattered over different bins of epochs.The exact epoch is saved along the checkpoint but for the formatting of the 'Merged Zoo.csv' we save the model weights along it's closest predecessor bin representer. *For example :* a model exiting on epoch 19 is saved along models in the epoch 15 bin and is not trained or logged further than 19 epochs

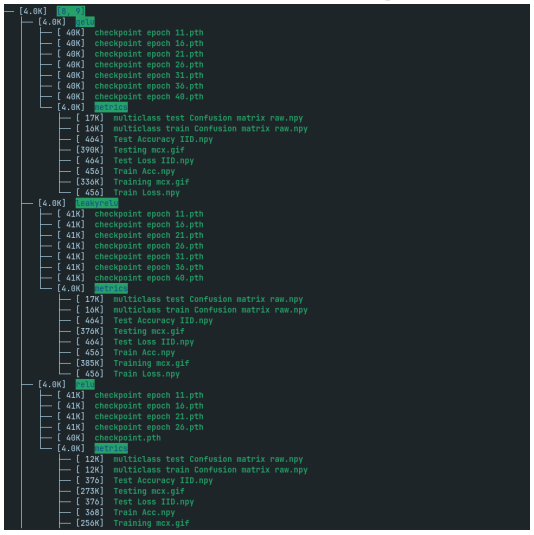
  




**(a)** Total size of subcombinations of MNIST checkpoints for inits and activations




**(b)** Example Nomenclature of an Experience




**(c)** Example of Tree view of data stored about CNN checkpoint

**Figure 3.** Showcasing the the Dataset checkpoints before building .csv format [1].

The architecture is the 1-channel architecture used in the hyper-representation paper.We log the accuracy and loss of every single CNN in numpy arrays along of the multiclass confusion matrix and create a gif out of them.

[details about optimizer Adam+scheduler CyclicLR choice/learning rate/usage of avalanche to create SplitMnist and save each class' images in a seperate folder can be explained later but as per standard cross-entropy was the loss and no data augmentation was used ]

### 3.3. Declaring Scenarios:

As explained above , we have early stopping for the CNNs meaning that some don't make it to the final 40 epochs ,hence a sub-scenario is defined with

- Activation **A**
- Set of Experiences **E**
- a parameter of the number of allowed overlapping classes **m**

The 'Create training scenarios.ipynb' elaborates how we created 2 sets of scenarios

1. sub-scneraio type 1:from [0-9] with 0,1,2 overlapping classes
2. sub-scneraio type 2:from [0-5] with up to 4 overlapping classes

A training Scenario is finally decided by taking S=(Sb,epoch) with all initializations included and that is used to declare the run title in WeightsAndBiases.com (Wandb) our logging tool and using our CustomDataset Class.

The idea to have overlapping classes was to see if we can identify a topological or spectral denominator or to see if it would make learning Weights that DID NOT require finetuning easier. For now all experiments are done with 0 overlap and so the fusion of knowledge happens across pairs and by varying the the pairs themselves we force the Transformer to learn to Fuse independently of the weight input.

### 3.4. Statistical Analysis:

every layer has it's own mean and std of values

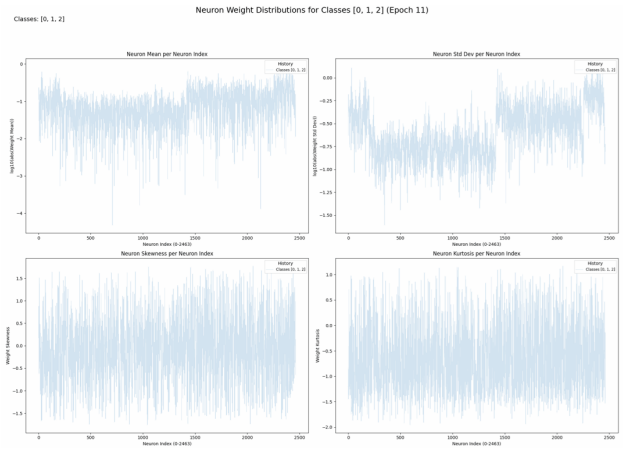
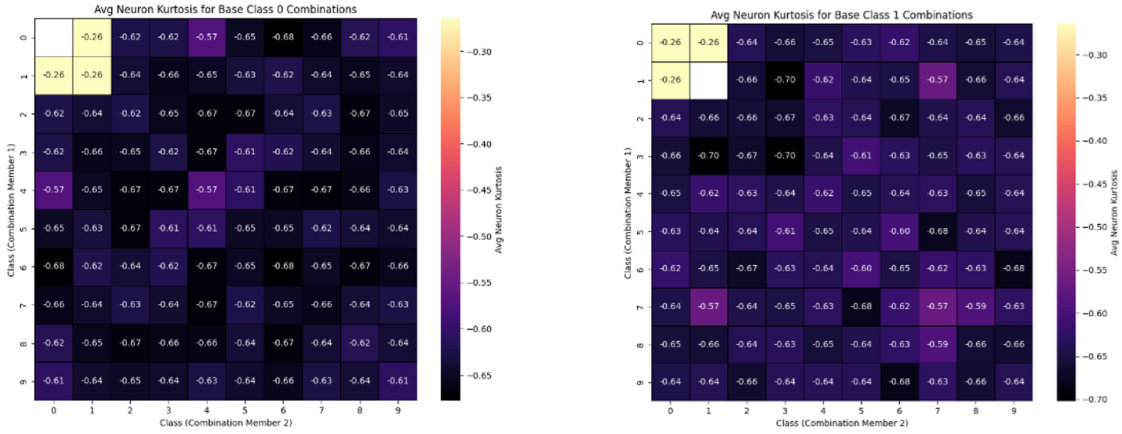
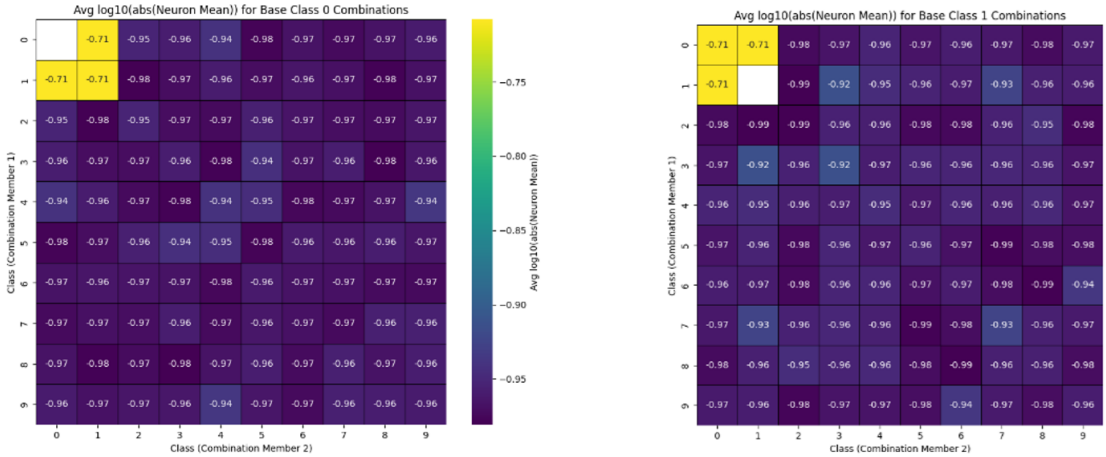



**Figure 4.** Statistics on models [0,1,2]

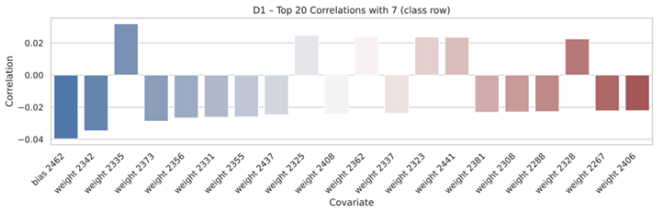



**(a)** 0 1 kurto

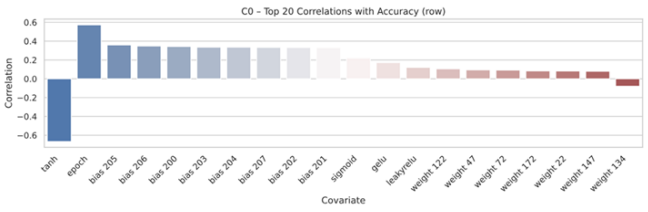



**(b)** 0 1 log10

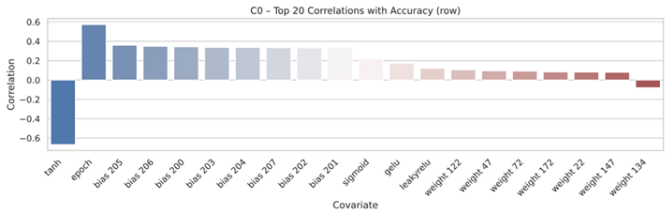
**Figure 5.** Statistical visualizations - Part 1



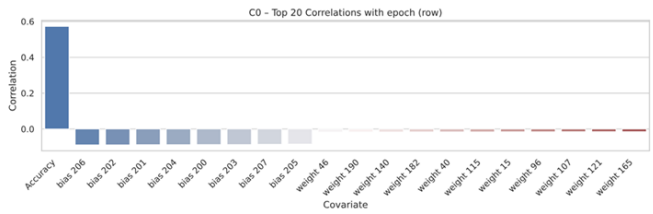

**(a)** C0 correlation matrix




**(b)** D1 correlation matrix



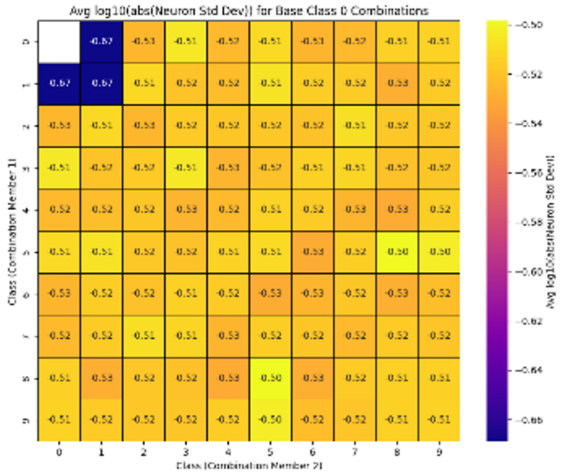

**(c)** C0 correlation matrix (zoomed detail)



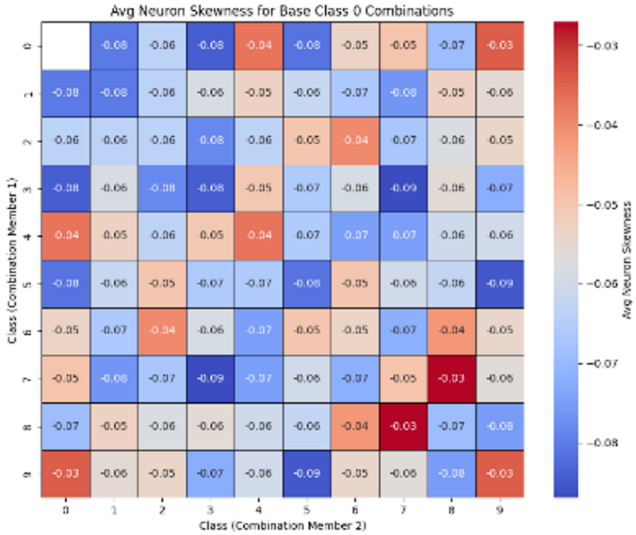

**(d)** D1 correlation matrix (zoomed detail)

**Figure 6.** Correlation matrices comparison – layout for enhanced readability

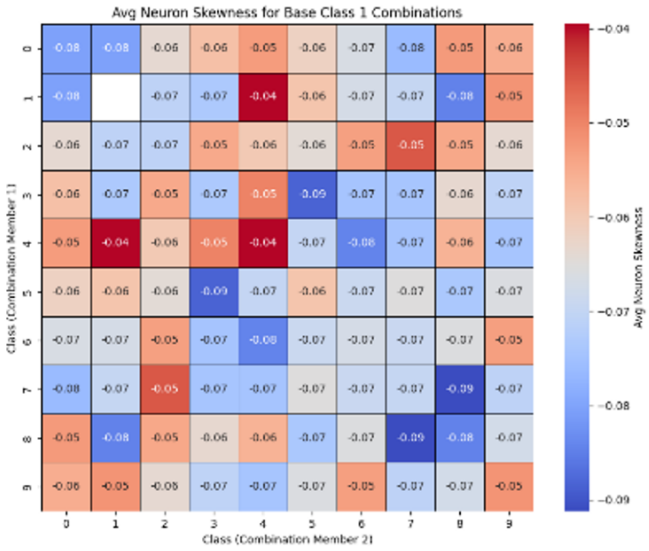
  




**(a)** 0 std

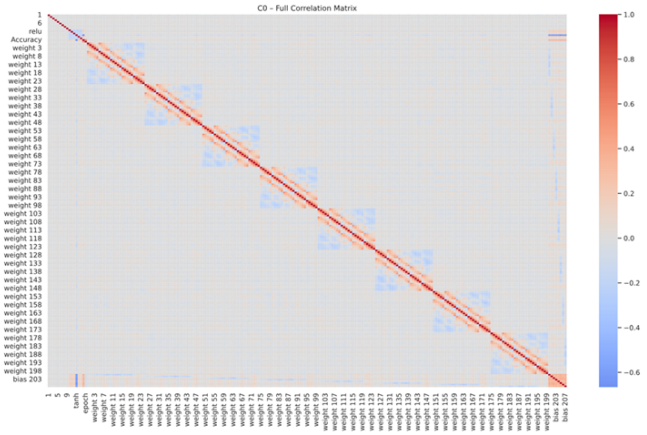



**(b)** 0 skewness

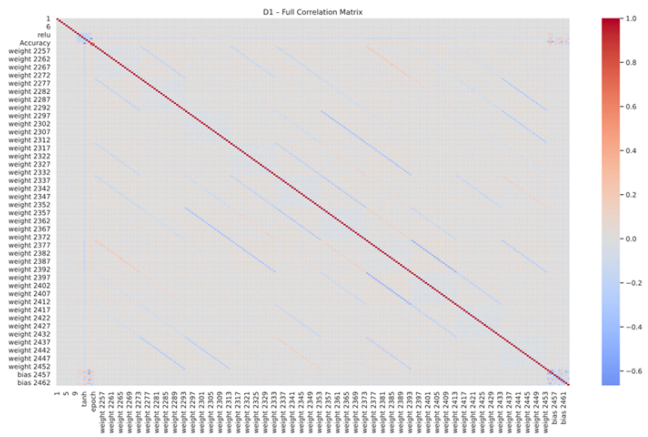



**(c)** 1 skewness

**Figure 7.** Statistical visualizations - Part 2




**(a)** C0 correlation matrix

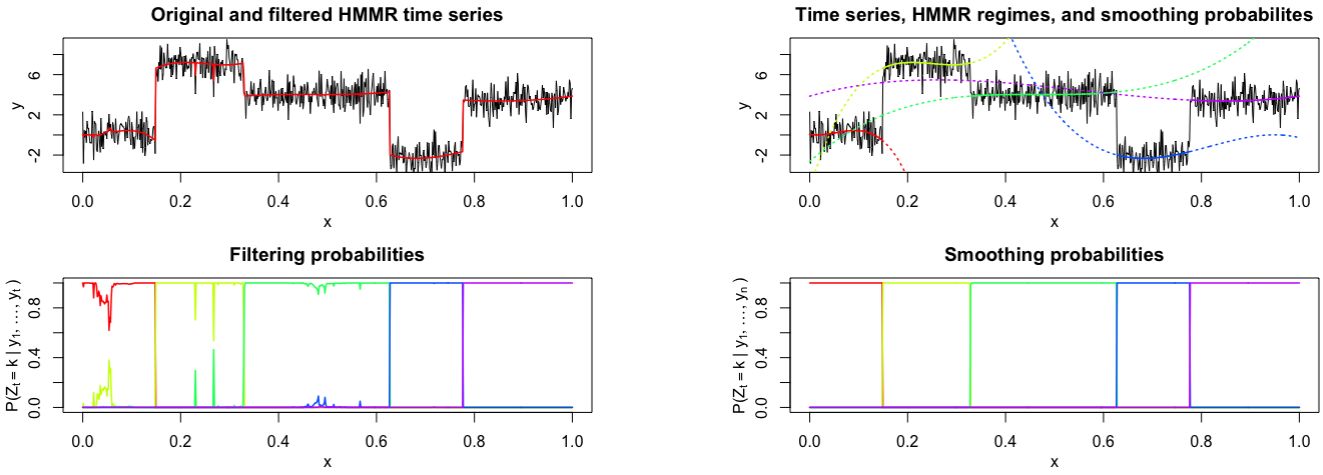



**(b)** D1 correlation matrix

**Figure 8.** Correlation matrices comparison

Some activations are better early and stay better as training continues

https://github.com/fchamroukhi/HMMR_r is an idea . regression on each layer one by one is another (would need multiple models or 1-sequence model with input being the length of the largest CNN layer and we pad the rest of the sequence for smaller layers -but with what-)




**(a)** C0 correlation matrix

**(b)** D1 correlation matrix

**Figure 9.** Correlation matrices comparison

## 4. Meta-Learner Methodology:

### 4.1. Architecture:

| model Arguement | description | Our experiments | Hyper-representation paper |
|---|---|---|---|
| – | number of encoder | 2 | 1 |
| d_model | Dimensionality of model embeddings (input/output tokens). | 960 | 972 |
| N | Number of stacked encoder-decoder layers (blocks). | 4 | 4 |
| heads | Number of attention heads in each MultiHeadAttention layer. | 4 | 12 |
| d_ff | Hidden dimension in FeedForward networks (typically 4 d_model). | 960 | 1140 |
| neck | Latent/bottleneck dimension  output of encoder, input to decoder. | 512 | 700 |
| dropout | Dropout probability applied after attention and FF layers. | 0.07 | 0.01 |
| max_seq_len | Max sequence length for positional encoding | 50 | 50 |
| epochs | Training epochs | 800 | 1750 |
| batch_size | batch size | 36 | 500 |
| – | data augmentation with neurone symmetry permuteation | no | 25000 |

### 4.2. Methodology:

we define the following :

- $\tilde{y_C}$ : is the ground truth label by a CNN
- $\hat{y_C}$ : is the predicted label by a CNN
- $\tilde{w}^{[l,Y_C]}_{[g,i,e]}$: are the ground truth weights of model from our zoo on $I_C$
- $\hat{w}^{[l,Y_C]}_{[g,i,e]}$: are the weights for a predicted CNN by the Transformer
- $\bar{w}^{[l,Y_C]}_{[g,i,e]}$: are the weights for a predicted CNN by the Transformer after finetuning on $I_C$
- $\hat{z}^{neck-1}_t$: is the latent representation after Encoder index $t \in [1,2]$.
- $\hat{z}^{neck}$: is the latent representation after the neck dense layer.

The target task is to predict:

$$\text{CNN3}^{[Y_{C3}]}_{[g,i,e]}(\tilde{w}^{[l,Y_{C3}]}_{[g,i,e]} \,\big|\, \tilde{w}^{[l,Y_{C1}]}_{[g,i,e]}) \cup \tilde{w}^{[l,Y_{C2}]}_{[g,i,e]})\big)$$

with weights obtained via:

$$\hat{w}^{[l,Y_{C3}]}_{[g,i,e]} = \text{Decoder}(\hat{z}^{neck}) = \textit{Decoder}\big(\text{Dense}(\textit{Encoder}1(\hat{z}^{neck-1}_1) \,\|\, \text{Encoder2}(\hat{z}^{neck-1}_2))\big) = \textit{Decoder}\big(\text{Dense}(\,\text{Encoder1}(\tilde{w}^{[l,Y_{C1}]}_{[g,i,e]}) \,\|\, \text{Encoder2}(\tilde{w}^{[l,Y_{C2}]}_{[g,i,e]}))\big),$$

where $\|$ denotes concatenation, and $\tilde{w}^{[l,Y_{C1}]}_{[g,i,e]}$, $\tilde{w}^{[l,Y_{C2}]}_{[g,i,e]}$ are the ground truth weights of CNN1 and CNN2, respectively.

Given the pair:

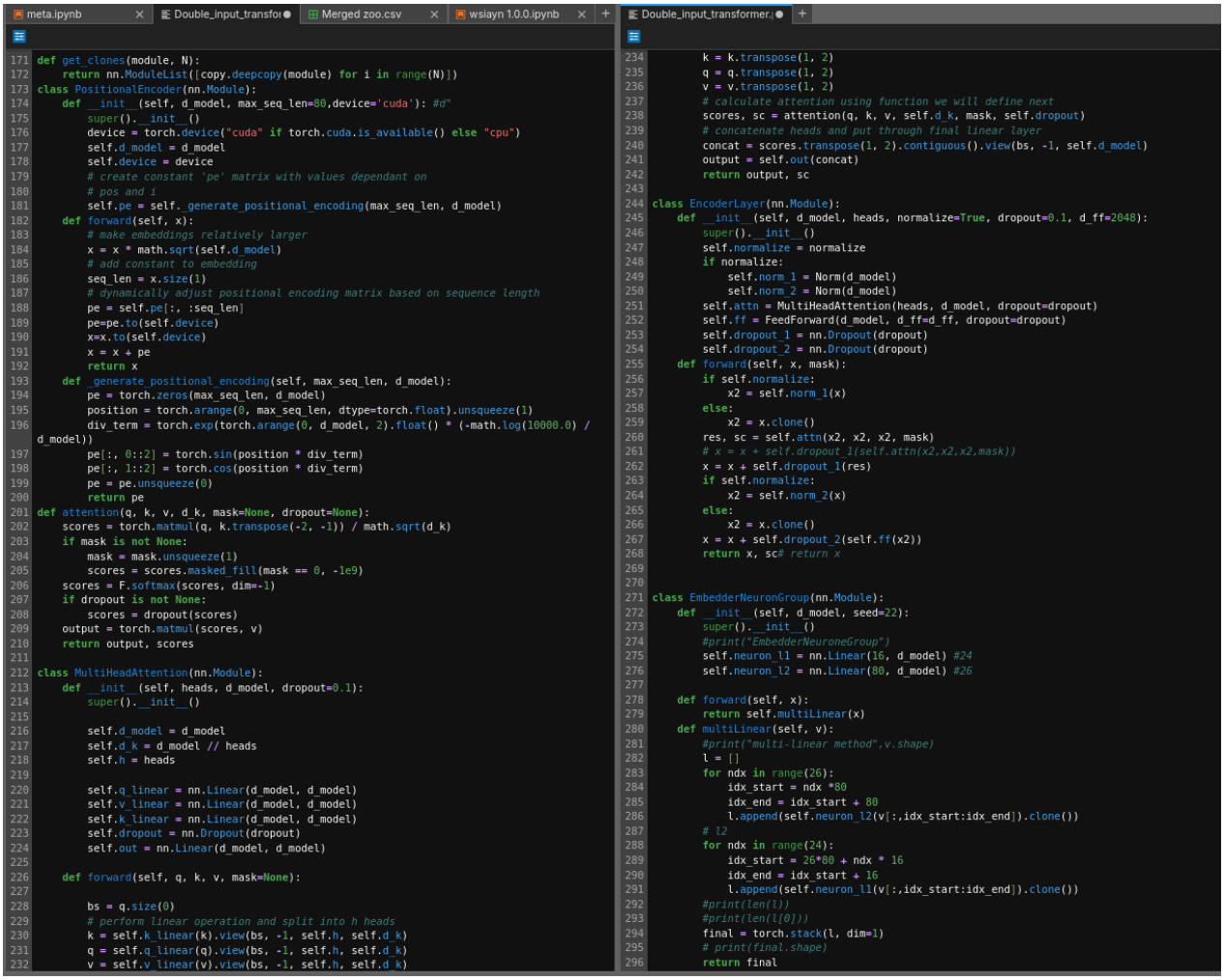
- $\text{CNN1}^{[Y_{C1}]}_{[g,i,e]}([Y_{C1}] \,\big|\, I_{C1})$
- $\text{CNN2}^{[Y_{C2}]}_{[g,i,e]}([Y_{C2}] \,\big|\, I_{C2})$

the combined experience uses implicitly input union $I_{C1} \cup I_{C2}$ and outputs CNNs to be performance-checked on concatenation $[Y_{C1}, Y_{C2}] = Y_{C3}$ for their in-distribution metrics and [0..9]-$Y_{C3}$ for out of distribution ones .

### 4.3. Tokenizing Numerical Values:

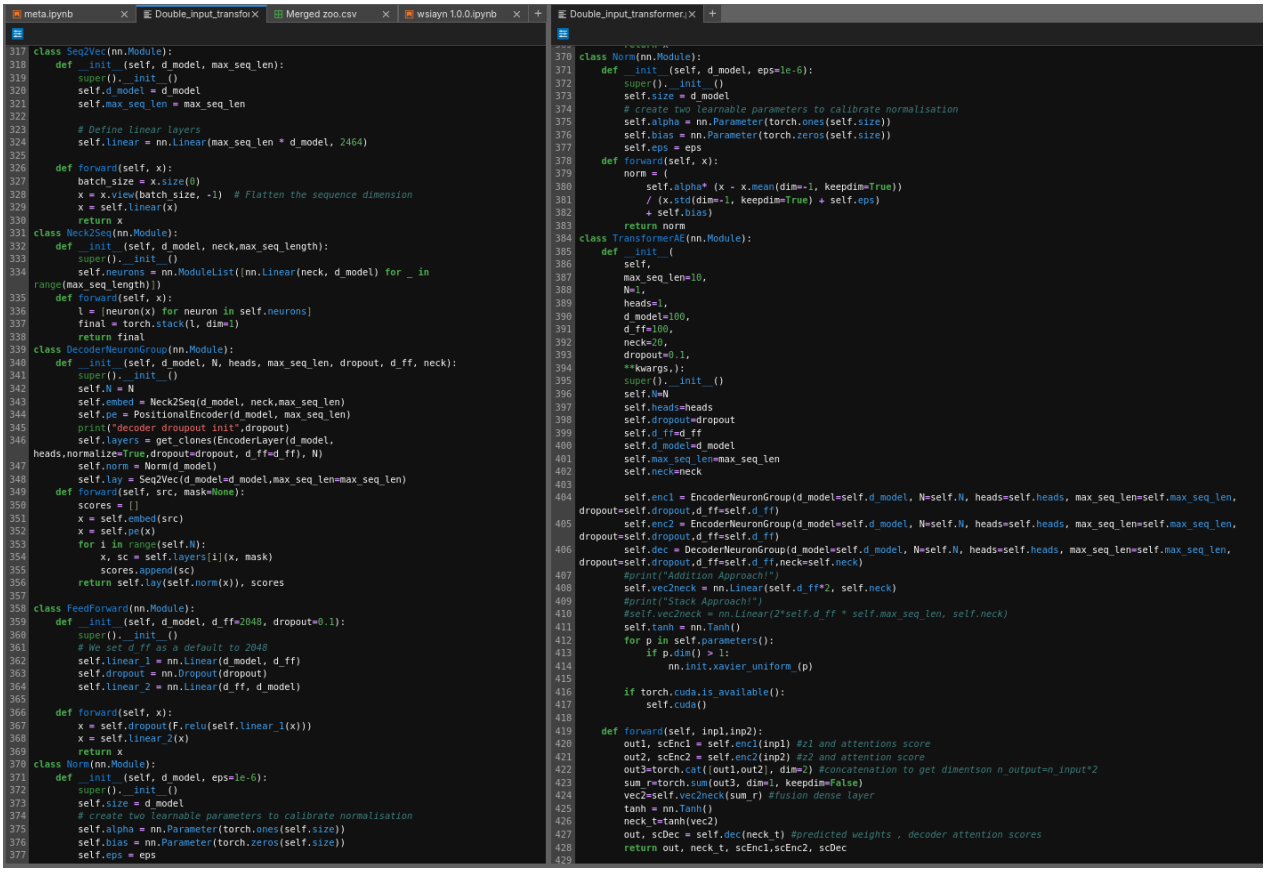
is done via a fixed dense layer ‘EmbedderNeuronGroup‘



```python
def get_clones(module, N):
    return nn.ModuleList([copy.deepcopy(module) for i in range(N)])
class PositionalEncoder(nn.Module):
    def __init__(self, d_model, max_seq_len=80,device='cuda'): #d"
        super().__init__()
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.d_model = d_model
        self.device = device
        # create constant 'pe' matrix with values dependant on
        # pos and i
        self.pe = self._generate_positional_encoding(max_seq_len, d_model)
    def forward(self, x):
        # make embeddings relatively larger
        x = x * math.sqrt(self.d_model)
        # add constant to embedding
        seq_len = x.size(1)
        # dynamically adjust positional encoding matrix based on sequence length
        pe = self.pe[:, :seq_len]
        pe=pe.to(self.device)
        x=x.to(self.device)
        x = x + pe
        return x
    def _generate_positional_encoding(self, max_seq_len, d_model):
        pe = torch.zeros(max_seq_len, d_model)
        position = torch.arange(0, max_seq_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) /
d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        return pe
def attention(q, k, v, d_k, mask=None, dropout=None):
    scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        mask = mask.unsqueeze(1)
        scores = scores.masked_fill(mask == 0, -1e9)
    scores = F.softmax(scores, dim=-1)
    if dropout is not None:
        scores = dropout(scores)
    output = torch.matmul(scores, v)
    return output, scores

class MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, dropout=0.1):
        super().__init__()

        self.d_model = d_model
        self.d_k = d_model // heads
        self.h = heads

        self.q_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)
        self.out = nn.Linear(d_model, d_model)

    def forward(self, q, k, v, mask=None):

        bs = q.size(0)
        # perform linear operation and split into h heads
        k = self.k_linear(k).view(bs, -1, self.h, self.d_k)
        q = self.q_linear(q).view(bs, -1, self.h, self.d_k)
        v = self.v_linear(v).view(bs, -1, self.h, self.d_k)
        k = k.transpose(1, 2)
        q = q.transpose(1, 2)
        v = v.transpose(1, 2)
        # calculate attention using function we will define next
        scores, sc = attention(q, k, v, self.d_k, mask, self.dropout)
        # concatenate heads and put through final linear layer
        concat = scores.transpose(1, 2).contiguous().view(bs, -1, self.d_model)
        output = self.out(concat)
        return output, sc

class EncoderLayer(nn.Module):
    def __init__(self, d_model, heads, normalize=True, dropout=0.1, d_ff=2048):
        super().__init__()
        self.normalize = normalize
        if normalize:
            self.norm_1 = Norm(d_model)
            self.norm_2 = Norm(d_model)
        self.attn = MultiHeadAttention(heads, d_model, dropout=dropout)
        self.ff = FeedForward(d_model, d_ff=d_ff, dropout=dropout)
        self.dropout_1 = nn.Dropout(dropout)
        self.dropout_2 = nn.Dropout(dropout)
    def forward(self, x, mask):
        if self.normalize:
            x2 = self.norm_1(x)
        else:
            x2 = x.clone()
        res, sc = self.attn(x2, x2, x2, mask)
        # x = x + self.dropout_1(self.attn(x2,x2,x2,mask))
        x = x + self.dropout_1(res)
        if self.normalize:
            x2 = self.norm_2(x)
        else:
            x2 = x.clone()
        x = x + self.dropout_2(self.ff(x2))
        return x, sc# return x


class EmbedderNeuronGroup(nn.Module):
    def __init__(self, d_model, seed=22):
        super().__init__()
        #print("EmbedderNeuroneGroup")
        self.neuron_l1 = nn.Linear(16, d_model) #24
        self.neuron_l2 = nn.Linear(80, d_model) #26

    def forward(self, x):
        return self.multiLinear(x)
    def multiLinear(self, v):
        #print("multi-linear method",v.shape)
        l = []
        for ndx in range(26):
            idx_start = ndx *80
            idx_end = idx_start + 80
            l.append(self.neuron_l2(v[:,idx_start:idx_end]).clone())
        # l2
        for ndx in range(24):
            idx_start = 26*80 + ndx * 16
            idx_end = idx_start + 16
            l.append(self.neuron_l1(v[:,idx_start:idx_end]).clone())
        #print(len(l))
        #print(len(l[0]))
        final = torch.stack(l, dim=1)
        # print(final.shape)
        return final
```

**(a)** Helper functions and modules for Tranformer



```python
class Seq2Vec(nn.Module):
    def __init__(self, d_model, max_seq_len):
        super().__init__()
        self.d_model = d_model
        self.max_seq_len = max_seq_len

        # Define linear layers
        self.linear = nn.Linear(max_seq_len * d_model, 2464)

    def forward(self, x):
        batch_size = x.size(0)
        x = x.view(batch_size, -1)  # Flatten the sequence dimension
        x = self.linear(x)
        return x
class Neck2Seq(nn.Module):
    def __init__(self, d_model, neck,max_seq_length):
        super().__init__()
        self.neurons = nn.ModuleList([nn.Linear(neck, d_model) for _ in
range(max_seq_length)])
    def forward(self, x):
        l = [neuron(x) for neuron in self.neurons]
        final = torch.stack(l, dim=1)
        return final
class DecoderNeuronGroup(nn.Module):
    def __init__(self, d_model, N, heads, max_seq_len, dropout, d_ff, neck):
        super().__init__()
        self.N = N
        self.embed = Neck2Seq(d_model, neck,max_seq_len)
        self.pe = PositionalEncoder(d_model, max_seq_len)
        print("decoder droupout init",dropout)
        self.layers = get_clones(EncoderLayer(d_model,
heads,normalize=True,dropout=dropout, d_ff=d_ff), N)
        self.norm = Norm(d_model)
        self.lay = Seq2Vec(d_model=d_model,max_seq_len=max_seq_len)
    def forward(self, src, mask=None):
        scores = []
        x = self.embed(src)
        x = self.pe(x)
        for i in range(self.N):
            x, sc = self.layers[i](x, mask)
            scores.append(sc)
        return self.lay(self.norm(x)), scores

class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff=2048, dropout=0.1):
        super().__init__()
        # We set d_ff as a default to 2048
        self.linear_1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        x = self.dropout(F.relu(self.linear_1(x)))
        x = self.linear_2(x)
        return x
class Norm(nn.Module):
    def __init__(self, d_model, eps=1e-6):
        super().__init__()
        self.size = d_model
        # create two learnable parameters to calibrate normalisation
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))
        self.eps = eps
```

```python
class Norm(nn.Module):
    def __init__(self, d_model, eps=1e-6):
        super().__init__()
        self.size = d_model
        # create two learnable parameters to calibrate normalisation
        self.alpha = nn.Parameter(torch.ones(self.size))
        self.bias = nn.Parameter(torch.zeros(self.size))
        self.eps = eps
    def forward(self, x):
        norm = (
            self.alpha* (x - x.mean(dim=-1, keepdim=True))
            / (x.std(dim=-1, keepdim=True) + self.eps)
            + self.bias)
        return norm
class TransformerAE(nn.Module):
    def __init__(
        self,
        max_seq_len=10,
        N=1,
        heads=1,
        d_model=100,
        d_ff=100,
        neck=20,
        dropout=0.1,
        **kwargs,):
        super().__init__()
        self.N=N
        self.heads=heads
        self.dropout=dropout
        self.d_ff=d_ff
        self.d_model=d_model
        self.max_seq_len=max_seq_len
        self.neck=neck

        self.enc1 = EncoderNeuronGroup(d_model=self.d_model, N=self.N, heads=self.heads, max_seq_len=self.max_seq_len,
dropout=self.dropout,d_ff=self.d_ff)
        self.enc2 = EncoderNeuronGroup(d_model=self.d_model, N=self.N, heads=self.heads, max_seq_len=self.max_seq_len,
dropout=self.dropout,d_ff=self.d_ff)
        self.dec = DecoderNeuronGroup(d_model=self.d_model, N=self.N, heads=self.heads, max_seq_len=self.max_seq_len,
dropout=self.dropout,d_ff=self.d_ff,neck=self.neck)
        #print("Addition Approach!")
        self.vec2neck = nn.Linear(self.d_ff*2, self.neck)
        #print("Stack Approach!")
        #self.vec2neck = nn.Linear(2*self.d_ff * self.max_seq_len, self.neck)
        self.tanh = nn.Tanh()
        for p in self.parameters():
            if p.dim() > 1:
                nn.init.xavier_uniform_(p)

        if torch.cuda.is_available():
            self.cuda()

    def forward(self, inp1,inp2):
        out1, scEnc1 = self.enc1(inp1) #z1 and attentions score
        out2, scEnc2 = self.enc2(inp2) #z2 and attention score
        out3=torch.cat([out1,out2], dim=2) #concatenation to get dimentson n_output=n_input*2
        sum_r=torch.sum(out3, dim=1, keepdim=False)
        vec2=self.vec2neck(sum_r) #fusion dense layer
        tanh = nn.Tanh()
        neck_t=tanh(vec2)
        out, scDec = self.dec(neck_t) #predicted weights , decoder attention scores
        return out, neck_t, scEnc1,scEnc2, scDec
```

**(b)** Decoder Layer and Tranformer declaration

**Figure 10.** Transformer Architecture

```
encoder droupout init 0.07
encoder droupout init 0.07
decoder droupout init 0.07
==========================================================================
Layer (type:depth-idx)                                  Param #
==========================================================================
TransformerAE                                           --
├─EncoderNeuronGroup: 1-1                               --
│    └─EmbedderNeuronGroup: 2-1                         --
│    │    └─Linear: 3-1                                 16,320
│    │    └─Linear: 3-2                                 77,760
│    └─PositionalEncoder: 2-2                           --
│    └─ModuleList: 2-3                                  --
│    │    └─EncoderLayer: 3-3                           5,539,200
│    │    └─EncoderLayer: 3-4                           5,539,200
│    │    └─EncoderLayer: 3-5                           5,539,200
│    │    └─EncoderLayer: 3-6                           5,539,200
│    └─Norm: 2-4                                        1,920
├─EncoderNeuronGroup: 1-2                               --
│    └─EmbedderNeuronGroup: 2-5                         --
│    │    └─Linear: 3-7                                 16,320
│    │    └─Linear: 3-8                                 77,760
│    └─PositionalEncoder: 2-6                           --
│    └─ModuleList: 2-7                                  --
│    │    └─EncoderLayer: 3-9                           5,539,200
│    │    └─EncoderLayer: 3-10                          5,539,200
│    │    └─EncoderLayer: 3-11                          5,539,200
│    │    └─EncoderLayer: 3-12                          5,539,200
│    └─Norm: 2-8                                        1,920
├─DecoderNeuronGroup: 1-3                               --
│    └─Neck2Seq: 2-9                                    --
│    │    └─ModuleList: 3-13                            24,624,000
│    └─PositionalEncoder: 2-10                          --
│    └─ModuleList: 2-11                                 --
│    │    └─EncoderLayer: 3-14                          5,539,200
│    │    └─EncoderLayer: 3-15                          5,539,200
│    │    └─EncoderLayer: 3-16                          5,539,200
│    │    └─EncoderLayer: 3-17                          5,539,200
│    └─Norm: 2-12                                       1,920
│    └─Seq2Vec: 2-13                                    --
│    │    └─Linear: 3-18                                118,274,464
├─Linear: 1-4                                           983,552
├─Tanh: 1-5                                             --
==========================================================================
Total params: 210,546,336
Trainable params: 210,546,336
Non-trainable params: 0
==========================================================================

Tracking run with wandb version 0.22.1
```

**Figure 11.** number of parameters per layer

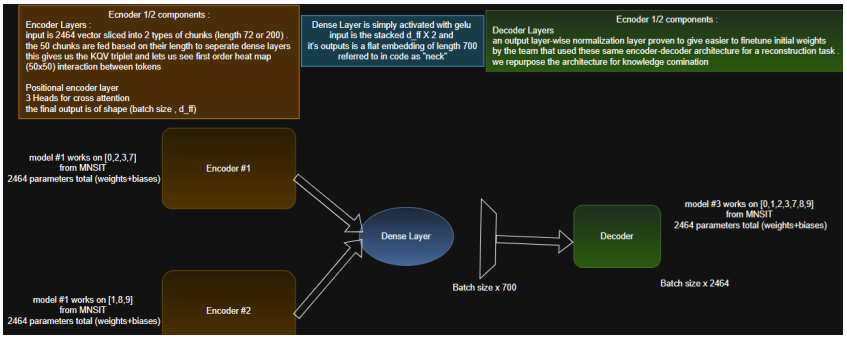



**Figure 12.** Your caption here (optional)

### 4.4. Training Loop Algorithm:

**Input:** Weights of two source CNNs $(W_1, W_2)$, target CNN weights $W_t$, pretrained double-encoder Transformer $T_\theta$
**Output:** Predicted target weights $\hat{W}_t$, performance diagnostics, and logged metrics

**Initialization:**
Initialize $track \leftarrow 0$, gradients $\nabla \leftarrow 0$
Create mixed-precision `Accelerator` with bf16 precision
Load list of scenarios $\{S_1, \ldots, S_n\}$ from `./data/Scenario/`
**foreach** *scenario* $S_t$ **do**
  Load training, validation, and test pairs:
  train_pair2, val_pair2, test_pair2 $\leftarrow$ np.load($S_t$)
  Convert each pair to list of tensors $(W_1, W_2, W_t)$

  **Model setup:**
  Initialize encoders $E_1, E_2$, transformer module $T_\theta$, and decoder $D$
  Initialize optimizers for encoder, decoder, and transformer
  Move models to accelerator device and set them to training mode

  **Training Loop:**
  **for** *epoch* $= 1$ **to** $N_{epochs}$ **do**
    Shuffle train_pair2
    **for** *each batch* $(W_1^b, W_2^b, W_t^b)$ **do**
      Encode: $z_1 \leftarrow E_1(W_1^b)$, $z_2 \leftarrow E_2(W_2^b)$
      Fuse latent codes: $z_f \leftarrow T_\theta(z_1, z_2)$
      Decode: $\hat{W}_t^b \leftarrow D(z_f)$
      Compute reconstruction loss $\mathcal{L}_{pred} = \|\hat{W}_t^b - W_t^b\|^2$
      Compute auxiliary metrics (distances, eigenvalues, persistent homology graphs)
      Combine metrics into total loss $\mathcal{L}_{total}$
      Zero optimizer gradients
      Backpropagate $\mathcal{L}_{total}$ and update model parameters
      Clip gradients by norm to 1

      **Logging:**
      Log losses and metrics to Weights & Biases:
      - $\mathcal{L}_{pred}$, $\mathcal{L}_{total}$
      - Fisher / Wasserstein distances
      - Eigenvalue spectra
      - Mapper graph statistics and persistence diagrams
      - Histograms of predicted vs. target weights
    **end**
    Optionally perform evaluation on validation pairs and save checkpoints
  **end**

  Compute test set predictions $\hat{W}_t$ and log final distances and graphs
**end**
**Output:** trained model $T_\theta$ and full W&B experiment log

**Algorithm 1:** Training Procedure for Double-Encoder Transformer for CNN Weight Prediction

### 4.5. Loss functions used and effects:

#### 4.5.1. Q-quantile loss:

Measures distributional discrepancy at specific quantilesrobust to outliers, sensitive to tail behavior.

$\mathcal{L}_Q(\mathbf{w}^{(1)}, \mathbf{w}^{(2)}) = \frac{1}{N}\sum_{i=1}^{N} \rho_Q\left(w_i^{(1)} - w_i^{(2)}\right), \quad \rho_Q(u) = u\left(Q - \mathbb{I}_{\{u<0\}}\right),$

This loss is minimized when the Q -th quantile of the difference distribution is zero, i.e., the two CNNs align at that quantile.

#### 4.5.2. Frobenius Norm Jacobian (Loss):

Measures overall sensitivity discrepancy between two models via Jacobian difference.

$\mathcal{L}_{\text{Frob}} = \left\|\mathbf{J}^{(1)} - \mathbf{J}^{(2)}\right\|_F = \sqrt{\sum_{i=1}^{C}\sum_{j=1}^{D}\left(J_{ij}^{(1)} - J_{ij}^{(2)}\right)^2},$

Minimizing this loss encourages the two networks to have similar local inputoutput geometry.

#### 4.5.3. Fisher Information Difference:

Measures intrinsic statistical dissimilarity between models via curvature of log-likelihood.

Compute the Fisher Information Distance (FID) between the two CNNs by treating their parameter vectors as points on a statistical manifold endowed with the Fisher information metric. This captures how differently the models encode informationuseful for comparing learning trajectories, robustness, or generalization.

$\mathcal{D}_{\text{Fisher}}(\theta^{(1)}, \theta^{(2)}) = \inf_\gamma \int_0^1 \sqrt{\dot{\gamma}(t)^\top \mathcal{I}(\gamma(t))\,\dot{\gamma}(t)}\, dt,$

#### 4.5.4. Contractive loss:

Penalizes sensitivity of hidden units to inputencourages robust, invariant representations.

$\mathcal{L}_{\text{contractive}} = \lambda \sum_{i=1}^{H}\left(h_i(1-h_i)\right)^2 \|\mathbf{w}_i\|_2^2 = \lambda \sum_{i=1}^{H}\left(h_i(1-h_i)\right)^2 \sum_{j=1}^{D} W_{ij}^2\,,$

#### 4.5.5. Wasserstein Distance/Geomloss:

Measures geometric discrepancy between distributions with smooth, differentiable approximation.

$\mathcal{W}_{2,\varepsilon}^2(\mu,\nu) = \min_{\mathbf{P}\in\Pi(\mu,\nu)}\left\{\sum_{i,j} P_{ij}\,\|x_i - y_j\|^2 \;-\; \varepsilon\, H(\mathbf{P})\right\},$

where

#### 4.5.6. Difference in Norm of the vector:

Measures global magnitude discrepancysimple, scale-sensitive, ignores direction.

$\mathcal{L}_{\|\cdot\|} = \left| \|\mathbf{w}^{(1)}\|_2 - \|\mathbf{w}^{(2)}\|_2 \right| = \left| \sqrt{\sum_{i=1}^{2464} \left(w_i^{(1)}\right)^2} - \sqrt{\sum_{i=1}^{2464} \left(w_i^{(2)}\right)^2} \right|,$

Minimizing this loss aligns the overall energy or scale of the two models, which can be a useful auxiliary objective in model compression, distillation, or weight-space interpolation.

#### 4.5.7. Auto-regressive Loss:

Enforces sequential fidelitylater chunks penalized relative to earlier prediction accuracy.

$\mathcal{L}_{\text{AR-MSE}} = \mathcal{L}_1 + \sum_{k=2}^{K} \lambda_{k-1} \frac{\mathcal{L}_k}{\mathcal{L}_{k-1}+\varepsilon}, \quad \mathcal{L}_k = \frac{1}{B\,|\mathcal{I}_k|} \sum_{b=1}^{B} \sum_{i \in \mathcal{I}_k} \left(\hat{y}_i^{(b)} - y_i^{(b)}\right)^2,$

#### 4.5.8. MAPE Loss:

$\text{MAPE}(\mathbf{w}^{(1)}, \mathbf{w}^{(2)}) = \frac{100\%}{N} \sum_{i=1}^{N} \frac{\left|w_i^{(1)} - w_i^{(2)}\right|}{|w_i^{(2)}| + \varepsilon}, \quad N = 2464,$

#### 4.5.9. Layer-wise-loss Normalization:

$\mathcal{L}_{\text{LW-Norm}} = \sum_{\ell=1}^{L} \frac{1}{|\Theta_\ell|} \sum_{\theta \in \Theta_\ell} \ell\left(\theta^{(1)}, \theta^{(2)}\right) = \sum_{\ell=1}^{L} \frac{1}{|\Theta_\ell|} \left\|\boldsymbol{\theta}_\ell^{(1)} - \boldsymbol{\theta}_\ell^{(2)}\right\|_2^2,$

#### 4.5.10. Jensen-Shannon Loss:

$\mathcal{L}_{\text{JS}}(\mathbf{p}, \mathbf{q}) = \sqrt{\text{JSD}(\mathbf{p} \,\|\, \mathbf{q})} = \sqrt{\frac{1}{2} D_{\text{KL}}(\mathbf{p} \,\|\, \mathbf{m}) + \frac{1}{2} D_{\text{KL}}(\mathbf{q} \,\|\, \mathbf{m})}, \quad \mathbf{m} = \frac{\mathbf{p}+\mathbf{q}}{2},$

#### 4.5.11. Fourrier transform difference in Norm:

$\mathcal{L}_{\text{FFT}}(\mathbf{w}^{(1)}, \mathbf{w}^{(2)}) = \frac{1}{N} \sum_{k=0}^{N-1} \left|\hat{w}_k^{(1)} - \hat{w}_k^{(2)}\right|, \quad \hat{\mathbf{w}} = \mathcal{F}(\mathbf{w}), \quad N = 2464,$

#### 4.5.12. Mel spectrogram Difference:

$\mathcal{L}_{\text{Mel}}(\mathbf{w}^{(1)}, \mathbf{w}^{(2)}) = \left\|\mathbf{M}^{(1)} - \mathbf{M}^{(2)}\right\|_F = \sqrt{\sum_{t=1}^{T} \sum_{m=1}^{M} \left(M_{t,m}^{(1)} - M_{t,m}^{(2)}\right)^2},$

#### 4.5.13. Mel Spectrogram Frechet inception distance:

$\text{MS-FID} = \left\|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\right\|_2^2 + \text{Tr}\left(\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2 - 2\left(\boldsymbol{\Sigma}_1^{1/2} \boldsymbol{\Sigma}_2 \boldsymbol{\Sigma}_1^{1/2}\right)^{1/2}\right),$

#### 4.5.14. Gromov-wasserstein loss:

$\text{GW}_p^p(\mu, \nu) = \min_{\mathbf{T} \in \Pi(\mu,\nu)} \sum_{i,j=1}^{N} \sum_{k,\ell=1}^{N} \left|d_X(x_i, x_k) - d_Y(y_j, y_\ell)\right|^p T_{ij}\, T_{k\ell},$

$\text{GW}_{p,\varepsilon} = \min_{\mathbf{T} \in \Pi(\mu,\nu)} \langle \mathbf{L}, \mathbf{T} \otimes \mathbf{T} \rangle - \varepsilon H(\mathbf{T}),$

Why it matters for CNN weights:

Invariant to permutation of neurons/channels (no need for explicit matching). Captures global structure: e.g., whether both models have a few large-magnitude weights surrounded by small ones. Suitable for comparing spectral embeddings or topological summaries derived from weight vectors.

#### 4.5.15. Bottleneck distance loss:

Measures similarity of topological features (persistence diagrams) via worstcase matching cost.

$d_{\text{B}}(\mathcal{D}_1, \mathcal{D}_2) = \inf_{\gamma \in \Gamma(\mathcal{D}_1, \mathcal{D}_2)} \sup_{p \in \mathcal{D}_1} \|p - \gamma(p)\|_\infty,$

#### 4.5.16. Latent:

Measures consistency of latent geometry before and after fusion via a shared dense bottleneck.

Explanation of the Loss

Intra-branch alignment: The predictions (z1_pred, z2_pred) are pulled toward the target embedding (z1_tg, z2_tg) via MSE.

This encourages:

Each encoder to faithfully encode its input, The prediction (from a combined or intermediate representation) to be closer to the target than the raw inputs, The merged bottleneck to align with a desired fused representation.

$\mathcal{L}_{\text{total}} = \sum_{k=1}^{2} \mathcal{L}_Z^{(k)} + \sum_{k=1}^{2} \frac{2\,\mathcal{L}_{\text{pred}}^{(k)}}{\mathcal{L}_{x_1}^{(k)} + \mathcal{L}_{x_2}^{(k)}}.$

Why Its Useful

Latent consistency: Ensures that the geometry of embeddings is preserved through the fusion operation. Relative supervision: The ratio term prevents trivial collapseif inputs already match the target well, the model isnt forced to overfit the prediction. Dual-branch validation: By enforcing the same target output[1] for both merged branches, you encourage cross-branch agreement in the final representation.

This design is well-suited for self-supervised fusion, multi-view learning, or cross-modal alignment where you want to verify that merging latent codes doesnt distort semantic content.

#### 4.5.17. multipersistence L2 :

#### 4.5.18. Time-Series segmentation guided normalisation loss :

### 4.6. Other Types of Attention

Differences Between This Implementation and Classical Multi-Head Attention

The MultiHeadAttention class in this code is a close implementation of the classical multi-head attention mechanism as described in the original Transformer paper ("Attention Is All You Need" by Vaswani et al., 2017). It computes scaled dot-product attention in parallel across multiple heads, concatenates the results, and applies a final linear projection. However, there are a few minor differences in design, functionality, and return values. I'll break them down step by step:

Core Computation (Scaled Dot-Product Attention):

Classical: The attention for each head is computed as $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$, where $Q, K, V$ are linearly projected versions of the inputs, and $d_k$ is the dimension per head (typically $d_{model}/h$, with $h$ being the number of heads). The softmax is applied along the last dimension.

This Implementation: Identical. The attention function computes exactly this: scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt($d_k$), followed by masking (if provided), softmax, optional dropout, and torch.matmul(scores, v). This is the standard scaled dot-product.

Linear Projections:

Classical: Separate linear layers project the input to queries (Q), keys (K), and values (V), each from $d_{\text{model}}$ to $h \times d_k$ (i.e., the full multi-head projection). After per-head attention, outputs are concatenated and projected back via another linear layer from $h \times d_k$ to $d_{\text{model}}$.

This Implementation: Matches exactly. It uses $q_linear$, $k_linear$, and $v_linear$ (each $d_{model} \rightarrow d_{model}$), reshapes to split into heads (via .view(bs, -1, self.h, self.$d_k$) and transpose), computes attention per head implicitly in parallel, concatenates with .transpose(1, 2).contiguous().view(bs, -1, self.$d_model$), and applies self.out (another linear $d_{model} \rightarrow d_{model}$).

Masking:

Classical: Supports optional masking (e.g., for padding or future tokens in decoders) by setting masked positions to a large negative value before softmax.

This Implementation: Identical. If a mask is provided, it's unsqueezed and used in masked–fill(mask == 0, -1e9) of ths cores variable, which is the standard approach.

Dropout:

Classical: Dropout is typically applied after the softmax (on the attention probabilities) and also in other parts of the Transformer (e.g., feed–forward layers). The original paper uses a dropout rate of 0.1 by default.

This Implementation: Matches. Dropout is applied post-softmax in the attention function (scores = dropout(scores) if provided). The class initializes with dropout=0.1, and it's passed to the attention function.

Return Values:

Classical: Most standard implementations (e.g., in PyTorch's nn.MultiheadAttention) return only the attention output (the weighted values). Attention weights (probabilities) are optional and often require a flag like 'return attn weights'=True.

This Implementation: A key difference–it always returns both the output and the attention weights (return output, sc, where sc is the post-softmax scores from the attention function). This is useful for interpretability or visualization (e.g., in the EncoderLayer, which propagates these scores). Classical implementations often omit weights by default to save computation/memory, but this is a minor extension, not a fundamental change.

Input Flexibility:

Classical: Designed to handle queries (Q), keys (K), and values (V) that can be from different sources (e.g., self–attention when Q=K=V, or cross-attention when Q differs).

This Implementation: Fully general–accepts separate q, k, v tensors, allowing for both self- and cross–attention. In the code's usage (e.g., in EncoderLayer), it's called with self.attn(x2, x2, x2, mask) for self–attention, which is standard.

Efficiency and Assumptions:

Classical: Assumes batched inputs with shape ('batch size', 'seq len', 'd model'). Head splitting is done via reshaping, which is efficient on GPUs.

This Implementation: Identical assumptions and efficiency. It uses .view() and .transpose(1, 2) for head-parallel computation, which is the optimized PyTorch way.

Initialization and Other Details:

Classical: No specific initialization mentioned in the paper, but common practice is Xavier uniform for weights.

This Implementation: In the parent TransformerAE, it applies 'nn.init.xavier uniform (p)' for all parameters with >1 dimension, which aligns with best practices but isn't a difference from classical (many libs do this).

Overall Summary of Differences: This is essentially the classical multi-head attention with one small addition–always returning the attention weights (sc) alongside the output. This makes it slightly more verbose for debugging or analysis but doesn't change the math or behavior. If you strip out the sc return, it's identical to reference implementations like Harvard's Annotated Transformer or PyTorch's official module (minus some optional features like bias terms or in-proj optimizations in PyTorch's version).

Comparison to Other Types of Attention Mechanisms

Attention mechanisms have evolved significantly since the original additive attention in seq2seq models. Below, I'll compare the multi-head attention (MHA) from this code (which is classical scaled dot-product MHA) to other major variants. I'll categorize them and highlight key differences in computation, use cases, efficiency, and limitations. This is not exhaustive but covers the most influential types.

1. Single-Head vs. Multi-Head Attention

Single-Head: Computes attention once over the full embedding dimension (e.g., basic scaled dot-product without head splitting).

Comparison to This MHA: Single-head is a special case of MHA with heads=1. MHA (as here) allows learning diverse relationships by projecting into multiple subspaces, improving representation power (e.g., capturing different syntactic/semantic aspects). This implementation supports arbitrary heads, making it more flexible. Single-head is simpler/faster but often underperforms on complex tasks like NLP or vision.

2. Additive (Bahdanau) Attention

Description: From Bahdanau et al. (2014) for seq2seq. Scores are computed as $score = v^T \tanh(W_1 q + W_2 k)$, an additive (MLP-based) alignment rather than dot-product.

Comparison to This MHA: Additive is slower ($O(d)$ per pair due to the MLP) and less scalable than dot-product ($O(1)$ per pair after matmul). This MHA uses efficient scaled dot-product, which parallelizes better on GPUs. Additive works well for smaller models/sequences but is largely replaced by dot-product in Transformers due to speed. No multi-head in classic additive.

3. Dot-Product Attention (Without Scaling)

Description: Similar to scaled dot-product but without $/\sqrt{d_k}$ (e.g., early versions in Luong et al., 2015).

Comparison to This MHA: This implementation includes scaling to prevent softmax saturation for large $d_k$, as in the Transformer paper. Unscaled can lead to vanishing gradients; scaling stabilizes training. Otherwise, very similar–both rely on $QK^T$.

4. Self-Attention vs. Cross-Attention

Self-Attention: Q, K, V from the same sequence (e.g., modeling intra-sequence dependencies).

Cross-Attention: Q from one sequence (e.g., decoder), K/V from another (e.g., encoder outputs).

Comparison to This MHA: This class supports both (via separate q/k/v args), but in the code, it's used as self-attention (q=k=v). Cross-attention is identical in math but applied differently in architectures like decoder layers. No structural difference here.

5. Local/Sliding Window Attention

Description: Restricts attention to a fixed window around each token (e.g., in Longformer or Image Transformer) to reduce $O(n^2)$ complexity to $O(n \times w)$, where w is window size.

Comparison to This MHA: This is global (full $O(n^2)$ over sequence length), attending to all positions. Local is more efficient for long sequences but loses global context. This implementation could be adapted with a custom mask for locality, but it's not built-in.

6. Sparse Attention (e.g., Sparse Transformer, Reformer)

Description: Uses sparsity patterns (e.g., fixed strides or hashing) to attend only to a subset of positions, achieving $O(n \log n)$ or better complexity.

Comparison to This MHA: This is dense/full attention (computes all pairs). Sparse is more scalable for very long inputs (e.g., >10k tokens) but may miss some interactions. Reformer's LSH (locality-sensitive hashing) approximates attention; this exact computation is more accurate but quadratic.

7. Linear/Efficient Attention (e.g., Linformer, Performer)

Description: Approximates full attention with low-rank projections or kernel tricks (e.g., Performer uses random Fourier features for softmax approximation), reducing complexity from $O(n^2)$ to $O(n)$.

Comparison to This MHA: This is exact quadratic attention, which is precise but memory-intensive. Linear variants trade a bit of accuracy for massive speedups on long sequences. For example, Linformer projects K/V to lower dims before matmul; this doesn't.

8. Grouped Query Attention (GQA) or Multi-Query Attention (MQA)

Description: In large models like LLaMA or GPT, shares K/V projections across groups of queries (GQA) or all heads (MQA) to reduce KV cache size in inference.

Comparison to This MHA: This uses separate projections per head for Q/K/V (full MHA). GQA/MQA are optimizations for deployment–faster inference with similar quality but less parallel diversity in heads. Not applicable here since this is a small custom model.

9. Relative Positional Attention (e.g., Transformer-XL, DeBERTa)

Description: Adds relative position biases to scores (e.g., $QK^T + QR^T$, where R is relative positions) for better handling of long-range dependencies.

Comparison to This MHA: This uses absolute positional encodings (via PositionalEncoder) but no relative biases in attention. Relative is better for extrapolation to longer sequences; this might struggle with position shifts.

10. Flash Attention or Optimized Variants

Description: Hardware-aware optimizations (e.g., DAO's FlashAttention) fuse softmax/matmul/kernel ops for faster training/inference without changing math.

Comparison to This MHA: This is a naive PyTorch implementation (separate matmuls), which is correct but slower on GPUs due to memory access. Flash is drop-in compatible but requires custom kernels–not a functional difference, just efficiency.

General Trade-offs Across Types:

Efficiency: This classical MHA is $O(n^2 d)$ time/space–great for short sequences (like here, 'max seq len'=80) but scales poorly. Efficient variants (linear/sparse) are for long inputs.

Expressivity: Exact dot-product (as here) is highly expressive; approximations (e.g., Performer) may lose nuance.

Use Cases: This suits standard Transformers for seq modeling. Additive for older RNNs; sparse/linear for genomics/videos; GQA for LLMs.

When to Choose This: If you need vanilla Transformer behavior with attention weights exposed, this is perfect. For production, consider optimized libs like Hugging Face Transformers.

### 4.7. Other Types of Transformers :

- **Mamba**: Linear scaling with sequence length, constant memory, 5x faster throughput than Transformers, handles million-token contexts via selective state space models (SSMs) without full attention.
- **Jamba**: Hybrid Transformer-Mamba MoE model with 52B parameters, processes 256k tokens on a single GPU using 4GB KV cache, balances efficiency and reasoning.
- **cosFormer**: Replaces softmax with cosine-based reweighting for linear attention, achieves 10x memory reduction and 2-22x speed-up while retaining 92-97
- **Performer**: Approximates softmax attention using random feature maps, reduces complexity to linear time, 4,000x faster on long sequences with 92-97
- **Linformer**: Projects attention keys/values to lower dimension for linear complexity, 76
- **FNet**: Substitutes self-attention with Fourier Transforms for token mixing, 80
- **Reformer**: Employs locality-sensitive hashing (LSH) and reversible layers, 10x memory reduction, processes up to 64k tokens efficiently.
- **BigBird**: Sparse attention with random, window, and global tokens, linear complexity, handles 8x longer sequences than BERT with theoretical completeness.
- **Longformer**: Uses sliding window attention plus global tokens, scales to 4,096 tokens linearly, drop-in replacement for BERT.
- **Switch Transformer**: Mixture-of-Experts (MoE) with top-1 routing, scales to 1.6T parameters with 7x training speedup and constant compute.
- **Griffin**: Hybrid of gated linear recurrences (Hawk-like) and local attention, combines recurrence for efficiency with attention for reasoning.
- **Mamba-Transformer Hybrids**: Integrates Mamba's state space efficiency with Transformer's attention for reasoning, achieves comparable quality with better memory utilization.

# 5. Appendix:

### 5.1. Continual Learning Primer

### 5.2. Hyper representation Current results

hyper former + Diffu + konstanstin

Accelerating training and creating new optimizer

### 5.3. Optimal Transport Primer

$\det(D^2u(x)) = \frac{f(x)}{g(\nabla u(x))}, \quad x \in \Omega$

$D^2u(x)$: Hessian of $u$ (must be positive definite for strict convexity). $\nabla u(x)$: Gradient (the transport map is $\nabla u$). $f(x)$: Source density. $g(y)$: Target density (evaluated at $y = \nabla u(x)$). The equation enforces the Jacobian condition: the pushforward of $f$ via $\nabla u$ equals $g$.

Optimal Transport (OT) is a mathematical framework fundamentally concerned with defining the most efficient way to move mass from a starting distribution to a target distribution, given a defined cost function for transportation. The sources describe OT through its historical formulations, its relaxed version, the resulting distance metrics, and its crucial applications in modern computational problems.

Part 1: Detailed Explanation of Optimal Transport 1. Historical Foundations (Monge and Kantorovich) Transportation theory, the study of optimal transportation and allocation of resources, was first formalized by the French mathematician Gaspard Monge in 1781. The core challenge was to find an optimal plan for moving resources (like dirt or ore) from multiple supply points (mines) to multiple demand points (factories). Major theoretical advances were later made by the Soviet mathematician and economist Leonid Kantorovich in the 1940s. Consequently, the problem is often known as the Monge Kantorovich transportation problem.

2. Monge's Formulation (MP) Monge's original problem attempts to find a one-to-one measure-preserving map $T$ (or $s$) from a source space $X$ with measure $\alpha$ to a target space $Y$ with measure $\beta$ The goal is to minimize the total transportation cost

$\inf_T \{\int_X c(x, T(x))d\alpha(x) \quad T_{\#}\alpha = \beta\}$

where c(x, T(x)) is the cost of moving a unit of mass from location $x$ to $T(x)$, and $T_{\#}\alpha = \beta$ means that $T$ pushes the mass distribution $\alpha$ forward exactly onto $\beta$. Limitation: The Monge formulation can be ill-posed because an optimal map $T$ might fail to exist. This happens, for example, if the source measure $\alpha$ is a single Dirac mass but the target measure $\beta$ is not, as a single point cannot be deterministically mapped to multiple locations.

3. Kantorovich's Formulation (MKP): The Relaxation Kantorovich introduced a relaxed, probabilistic version (MKP) which avoids the need for a deterministic map $T$. Instead of finding a single mapping, MKP seeks an optimal coupling, $\pi$, which is a joint probability measure on $X \times Y$. The key idea is that mass at a source point $x$ can be "split" and potentially dispatched across several target locations $y$. This flexibility is captured by $\pi(x, y)$, which describes the amount of mass flowing from $x$ to $y$. The objective is to minimize the generalized transport cost $\int_{X\times Y} c(x, y)d\pi(x, y)$, subject to constraints ensuring that $\pi$'s marginals are the original distributions $\alpha$ and $\beta$.

Advantages: This relaxation solves the issues of existence and symmetry inherent in Monge's problem. The continuous Kantorovich problem always has a solution on compact domains. Critically, this formulation constitutes an infinite-dimensional linear program.

Probabilistic Interpretation: If $(X, Y) \sim \pi$, the marginal constraint means that $X \sim \alpha$ and $Y \sim \beta$. The problem minimizes the expected cost $\mathbb{E}(c(X, Y))$ over all joint distributions $\pi$ with the correct marginals.

4. The Resulting Metric: Wasserstein Distance ($W_p$) When the cost function $c(x, y)$ is defined as a power of the distance between points, $c(x, y) = d(x, y)^p$, the minimum cost attained by the Kantorovich problem defines the $p$-Wasserstein distance ($W_p(\alpha, \beta)$). This metric quantifies the cost of transforming one distribution ($\alpha$) into another ($\beta$). $W_p(\alpha, \beta) = \left(\min_{\pi\in U(\alpha,\beta)} \int_{X\times Y} d(x, y)^p d\pi(x, y)\right)^{1/p}$ The Wasserstein distance (also known as the optimal transport distance or Kantorovich Rubinstein norm) is a metric (it is symmetric, positive, and satisfies the triangle inequality). It is particularly useful because it is a weak distance, meaning it can compare singular distributions (like discrete point clouds) and quantify spatial shift between their supports.

5. Duality and Monge-Kantorovich Equivalence The Kantorovich problem, being a linear program, possesses a dual formulation that maximizes a term involving Kantorovich potentials ($f$ and $g$). A fundamental result linking the two formulations is Brenier's Theorem. For the quadratic cost ($c(x, y) = ||x-y||^2$), and assuming the source measure $\alpha$ has a density, the unique optimal transport map $T$ (Monge's solution) exists and is the gradient of a convex function, $\phi$ ($T = \nabla\phi$). This implies that the Monge and Kantorovich costs are the same, and the optimal coupling $\pi$ is induced by this deterministic map: $\pi = (\mathrm{Id}, T)_{\#}\alpha$.

6. Computation and Regularization (Sinkhorn) Since the exact OT problem is often computationally expensive, especially in high dimensions, a common approach is Entropic Regularization. This involves modifying the Kantorovich objective function by adding a scaled Kullback-Leibler (KL) divergence term (entropy) as a regularizer:

$L_\epsilon^c(\alpha, \beta) := \min_{\pi\in U(\alpha,\beta)} \int_{X\times Y} c(x, y)d\pi(x, y) + \epsilon\mathrm{KL}(\pi|\alpha \otimes \beta)$ This makes the problem strictly convex, guaranteeing a unique, smooth solution $P_\epsilon$. The solution $P_\epsilon$ can be efficiently computed using Sinkhorn's algorithm, which is derived from the dual regularized problem.

Part 2: Concepts to Retain from the Sources The sources emphasize that Optimal Transport is a crucial concept in contemporary machine learning for introducing geometric knowledge and defining robust metrics, particularly when dealing with structured data like neural network parameters or topological descriptors. 1. The Wasserstein Distance as a Topological Metric

The Wasserstein distance ($W_q$) is a fundamental metric used to compare persistence diagrams (PDs) in Topological Data Analysis (TDA). This distance leverages optimal transport theory to calculate the cost of matching features (points) between two diagrams.

When used in optimization pipelines, the Sliced Wasserstein Distance (SWD) or related kernelized versions are often preferred because the full Wasserstein distance involves combinatorial matching and is generally non-differentiable.

The Optimal Transport distance is used to define distances between discrete signed measures, such as the Hilbert decomposition signed measure derived from multiparameter persistence, enabling the creation of differentiable topological losses. 2. Optimal Transport for Neural Network Structure (OTMANN)

The concept of optimal transport is adapted to create the OTMANN (Optimal Transport Metrics for Architectures of Neural Networks) distance.

OTMANN is formulated as an Optimal Transport program that minimizes a matching scheme to quantify the dissimilarity between two network architectures based on structural position and layer properties.

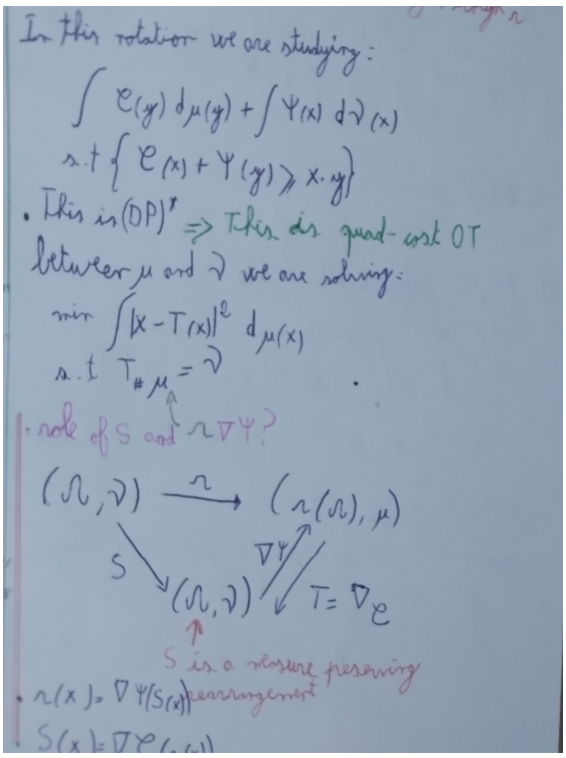
This metric is used within the NASBOT (Neural Architecture Search with Bayesian Optimisation and Optimal Transport) framework, where a measure of similarity (a kernel) is derived from the OTMANN distance to facilitate Bayesian optimization.

Although OTMANN uses an OT formulation, it is specifically noted that it is not a Wasserstein distance because the underlying mass supports and cost matrices change based on the specific networks being compared. 3. Role in Loss Functions and Continual Learning

Optimal Transport provides a robust way to compare distributions of complex objects. This is highly relevant to my objective, as you noted that the Wasserstein distance (WD) decreases meaningfully during fine-tuning of weight hyper-representations, suggesting it captures functional similarity better than simple $L_2$ losses.

Optimal Transport is seen as a key component for designing loss functions that can preserve both the shape (Topological manifold) and content of data, making it a prime candidate for optimizing knowledge combination in generative models like the one you are developing.

OT finds applications in analyzing generative models via Flow Matching, where an optimal transport map is sought to transform a reference distribution (like Gaussian noise) to a target data distribution.




**(a)** Class CNN declaration[1]

**(b)** Showcasing the dataset checkpoints before building .csv format [1].

### 5.4. Persistent Homology Primer

#### *5.4.1. Topological Data Analysis (TDA) Background*

**Persistent Homology and Classical TDA** Topological Data Analysis (TDA) is a field of algebraic topology aimed at providing quantifiable, comparable, robust, and concise summaries of the shape of data. PH is the core tool of TDA, analyzing the evolution of topological features across different scales. The process begins by representing the data space as a simplicial complex. A filtration is then constructed : a nested sequence of increasing subsets of the simplicial complex. Homology provides the mathematical language for counting the holes, or cavities, in a topological space. The dimension of the $i$-th homology group, $H_i(K)$, is the $i$-th Betti number ($\beta_i$), which counts the number of independent $i$-dimensional holes (e.g., $\beta_0$ counts connected components, $\beta_1$ counts loops/tunnels). Persistent Homology tracks how these homology groups evolve (or persist) across the filtration. Features born at one scale and dying at a later scale are considered persistent and are deemed more likely to represent true topological structure rather than noise.

One-Parameter Persistence and Barcodes/Diagrams In the classical one-parameter setting (indexed by a single scale value, $t \in \mathbb{R}$), there exists a canonical and complete discrete invariant for PH: the persistence barcode.

Barcode: A multiset of intervals $[b, d)$, where $b$ is the filtration value (scale) at which a topological feature is born (appears) and $d$ is the value at which it dies (is filled or disappears).

Persistence Diagram (PD): An equivalent representation where each interval $[b, d)$ is plotted as a point $(b, d)$ in the extended plane. The persistence (lifetime) of the feature is $d - b$, which is the vertical distance from the point to the diagonal $\Delta = (x, x)$, where noise typically lies.

- **(**Persistence Diagram)

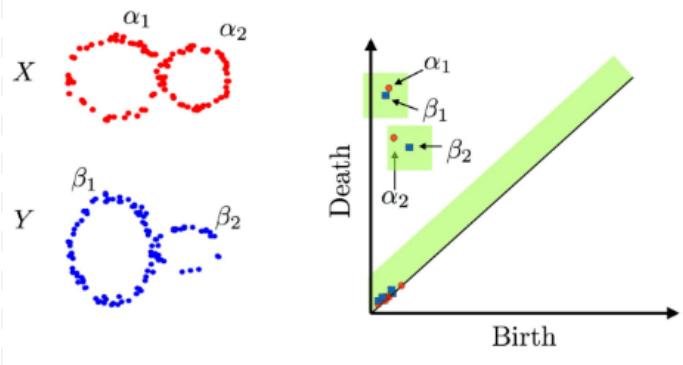



**Figure 14.** Your caption here (optional)

- **(**Barcodes)

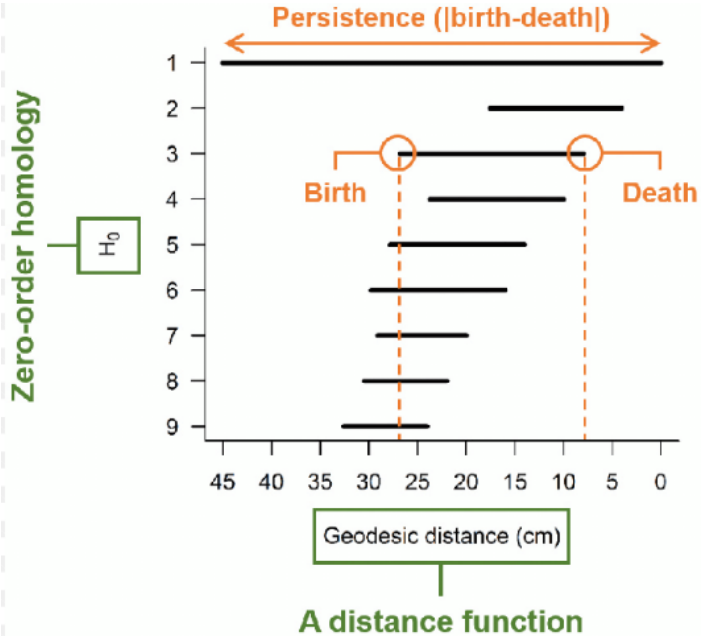
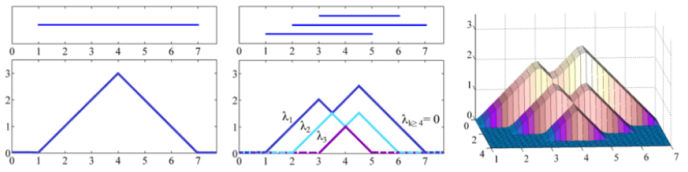



**Figure 15.** Your caption here (optional)

- (Persistence Landscape)



Construction of a persistence landscape: (left) from an interval to the auxiliary function; (middle) from a barcode to a persistence landscape; (right) 3D visualization of the persistence landscape.

**Figure 16.** Your caption here (optional)

- (Persistence Image)
- (Heat Kernel / Image)
- (Betti Numbers / Curves)

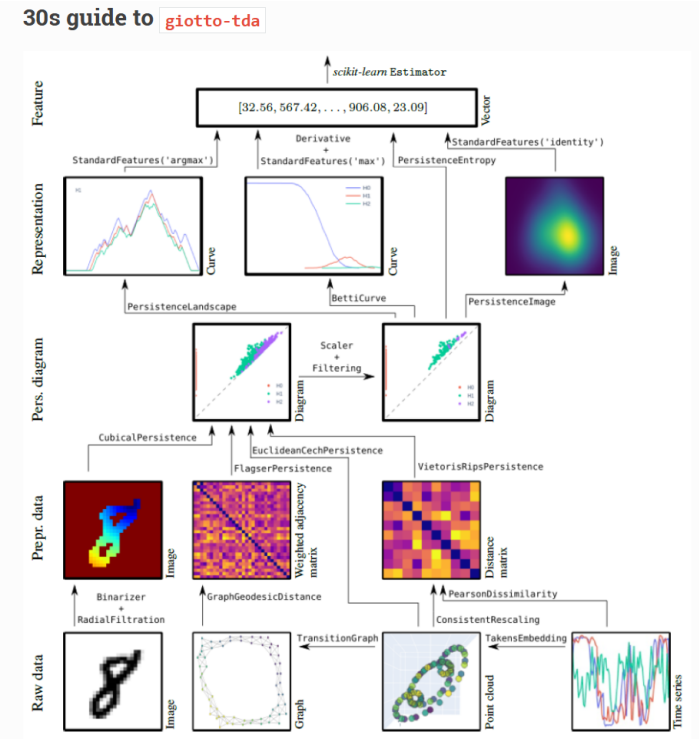



**Figure 17.** Your caption here (optional)

- (Bottleneck Distance)
- (Wasserstein Distance)
- (Sliced Wasserstein Distance)
- (Multiparameter Persistence)

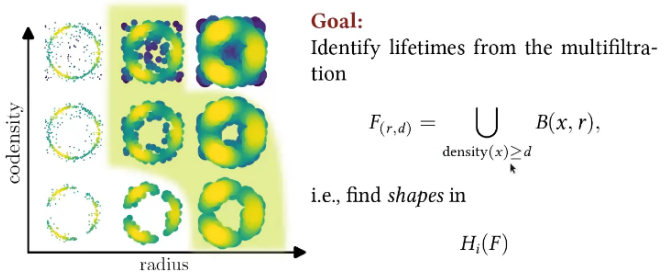



**Figure 18.** Your caption here (optional)

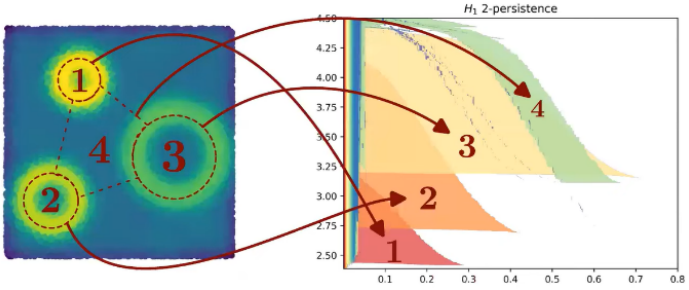



**Figure 19.** Your caption here (optional)

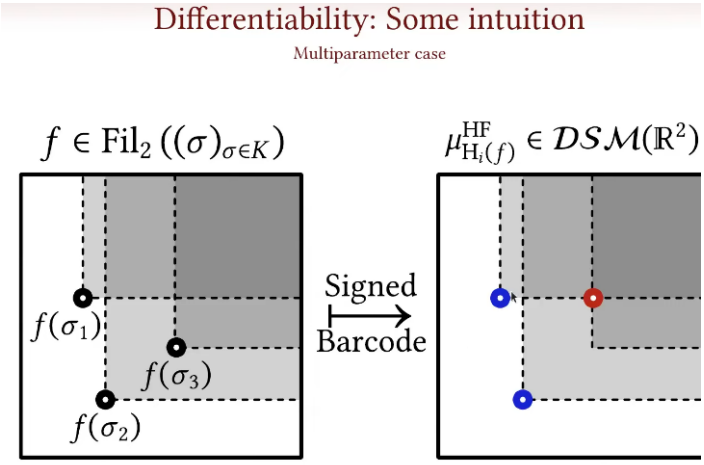



**Figure 20.** Your caption here (optional)

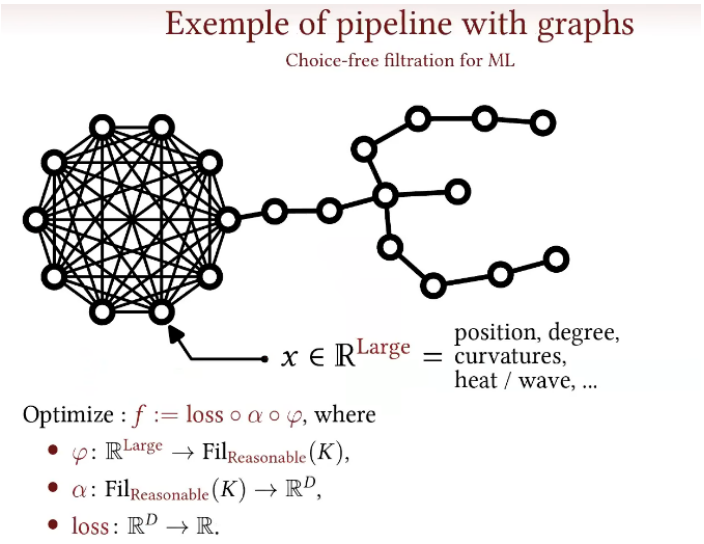



**Figure 21.** Your caption here (optional)

- (Mapper Algorithm / Graph)

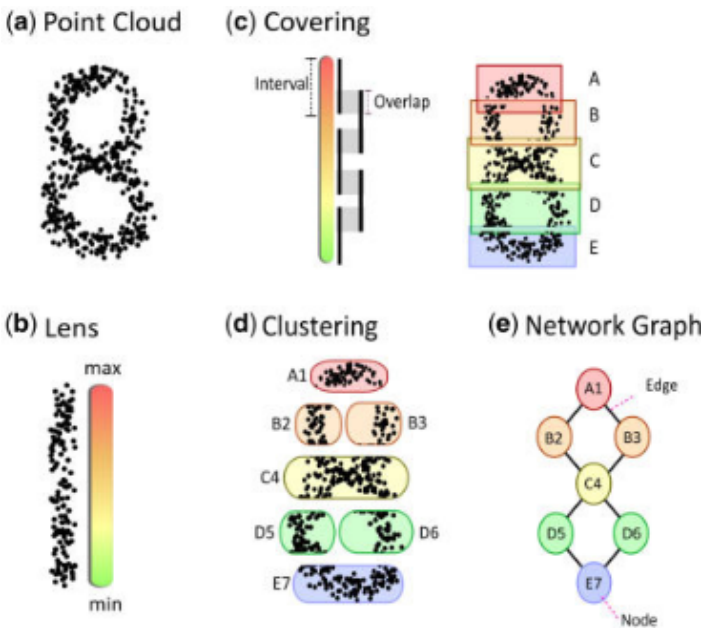
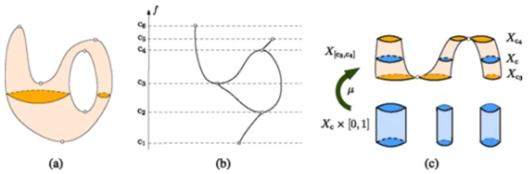



**Figure 22.** Your caption here (optional)

- (Reeb Graph)



(a) X is a solid torus and its Reeb graph w.r.t. the height function $f$ is shown in (b). (c) $f$ is a level-set-tame function w.r.t. discrete values $\{c_1,\ldots,c_6\}$. There is a continuous map $\mu : X_c \times \allowbreak[0,1] \to X_{[c_3,c_4]}$ whose restriction to the open set $X_c \times (0,1)$ is a homeomorphism. In the *top row*, there are three disjoint interval-components in $X_{(c_3,c_4)}$ whose closures may intersect in level sets $X_{c_3}$ and $X_{c_4}$

**Figure 23.** Your caption here (optional)

- (Contour Trees)

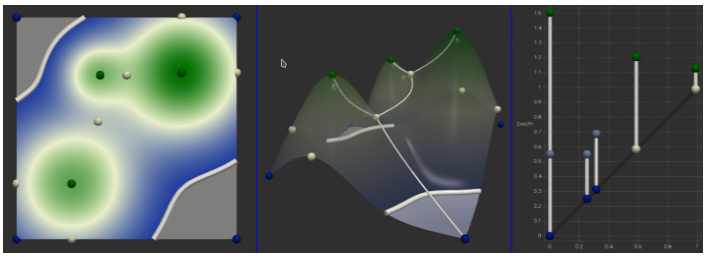


**Figure 24.** Your caption here (optional)

- (Merge Trees)

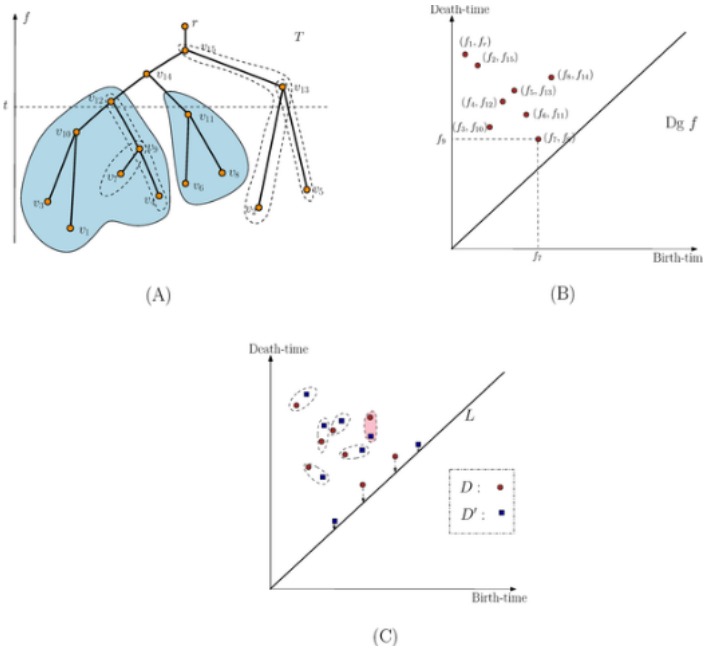



**Figure 25.** Your caption here (optional)

- (Morse–Smale Complex)
- (Topological Regularizer)
- (Local Cohomology)
- (Vietoris–Rips Complex)
- (Alpha Complex)
- (Cubical Complex)

| TDA Feature | multipers | Gudhi | giotto-tda | Perslay | TTK | Ripser | Persim | Kepler Mapper | TopologyLayer | Torch Topological | Differentiable? |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Persistence Diagram | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | Partial |
| Barcodes | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | Partial |
| Persistence Landscape | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ | Yes |
| Persistence Image | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | Yes |
| Heat Kernel/Image | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | Yes |
| Betti Numbers/Curves | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | Yes |
| Bottleneck Distance | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | No |
| Wasserstein Distance | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ | Yes |
| Sliced Wasserstein Distance | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✓ | Yes |
| Multiparameter Persistence | ✓ | Partial | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | Yes |
| Mapper Algorithm/Graph | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | No |
| Reeb Graph | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | No |
| Contour Trees | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | No |
| Merge Trees | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | No |
| Morse-Smale Complex | ✗ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | No |
| Topological Regularizer | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✓ | Yes |
| Local Cohomology | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | No |
| Vietoris-Rips Complex | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | No |
| Alpha Complex | ✗ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | No |
| Cubical Complex | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ | ✓ | No |

✓ Fully Supported Partial Partially Supported ✗ Not Supported **Differentiable?** Indicates gradient descent compatibility for ML integration

**Figure 26.** Your caption here (optional)

3. The Mapper Algorithm

**Input:** D with $|D| = m$, filter function $f : D \to \mathbb{R}^d$, finite cover $\mathcal{U} = \{U_i\}_{i \in I}$ of $\mathrm{Im}(f) \subseteq \mathbb{R}^d$, clustering algorithm $\mathcal{C}$.
**Output:** Simplicial complex $S_D$.
$S_D \leftarrow \emptyset$;
$D_i \leftarrow f^{-1}(U_i)$ for all $i \in I$;
**for** *all* $i \in I$ **do**
  $\{C_i^1, \dots, C_i^{k_i}\} \leftarrow \mathcal{C}(D_i)$ `// Apply the clustering algorithm to` $D_i$`: the output are the clusters`
  $S_D \leftarrow S_D \cup \{C_i^1, \dots, C_i^{k_i}\}$ `// Add the clusters found as vertices`
**end**
**for** *all* $\{C_1, \dots, C_t\} \in \mathcal{P} \setminus S \bigcup_{i \in I} \{C_i^1, \dots, C_i^{k_i}\}$ `// for all possible subsets of found clusters` **do**
  **if** $\bigcap_{j=1}^{t} C_j \neq \emptyset$ **then**
    $S_D \leftarrow S_D \cup \{\{C_1, \dots, C_t\}\}$ `// We add the simplex` $\{C_1, \dots, C_t\}$
  **end**
**end**
**return** $S_D$;

**Algorithm 2:** Mapper algorithm

The Mapper algorithm (introduced by Singh et al., 2007) is a tool for visualizing and extracting structural descriptions from high-dimensional datasets, typically outputting a graph or simplicial complex. Mapper uses four primary ingredients: 1. A dataset $D$. 2. A filter function ($f$): A scalar (usually $\mathbb{R}$ or $\mathbb{R}^2$-valued) function defined on the dataset $D$ that projects the high-dimensional data down to a manageable domain. 3. A finite, overlapping cover of the image of the filter function $f$. 4. A clustering algorithm (like DBSCAN) applied to the preimages of the cover elements. The result is a graph where nodes represent clusters of points that are close in the data space and map close to each other via the filter function, and edges connect nodes if their corresponding clusters overlap. Mapper is highly valuable for exploratory data analysis and visualization, offering interpretability, but the complex construction typically renders the graph structure non-differentiable for end-to-end optimization. 4. Transition to Multidimensional Persistence In many applications, data is complex and may be filtered along multiple non-linear dimensions simultaneously (e.g., scale and density), leading to a multifiltration. The critical transition occurs when $n > 1$ parameters index the filtration (e.g., a bifiltration parameterized by radius $\epsilon$ and curvature $\kappa$). Gunnar Carlsson and Afra Zomorodian demonstrated that, unlike the one-parameter case, no similar complete discrete invariant (like the barcode) exists for multiparameter persistence modules due to fundamental algebraic reasons. Instead, they proposed the rank invariant ($\rho_{X,i}$) as a robust discrete invariant for multifiltrations.

Rank Invariant: Defined as $\rho_{X,i}(u, v) = \mathrm{rank}(H_i(X_u, k) \to H_i(X_v, k))$, this invariant measures the rank of the structure maps between homology groups indexed by parameters $u \leq v$.

While the rank invariant is mathematically equivalent to the persistence barcode in one dimension (where it is complete), it extends to higher dimensions as a useful, though incomplete, invariant for robust estimation of Betti numbers and detection of persistent features. To address the lack of a canonical discrete descriptor, modern research focuses on developing effective and often differentiable incomplete descriptors for multiparameter persistence, such as the multiparameter persistence landscape, multiparameter persistence image, and signed barcodes as measures.

Recommended Loss Function and Topological Descriptors To rival diffusion-based methods like D2NWG, which prioritize reconstruction accuracy(often $L_2$ based), my Transformer model must optimize for qualities conducive to fine-tuning and fast learning–qualities often missed by simple MSE on weights. Choosing the Optimal Loss Function Since my Transformer uses an autoencoder structure for weight regression, the loss should leverage known functional losses for hyper-representations combined with topologically robust metrics:

1. Layer–Wise Loss Normalization (LWLN): You should adopt the LWLN MSE loss. This loss prevents layers with small weight magnitudes from being disregarded, which is critical for generating functional models (as shown in work related to hyper–representations). $L_{\overline{MSE}} = \frac{1}{MN} \sum_{i=1}^{M} \sum_{l=1}^{L} \frac{|\hat{w}_i^{(l)} - w_i^{(l)}|_2^2}{\sigma_l^2}$

2. Contrastive Loss ($L_c$): The original hyper-representation work combines $L_{\overline{MSE}}$ with a contrastive loss $L_c$ to impose structure and regularity on the latent space. This practice should be retained: $L = \beta L_{\overline{MSE}} + (1 - \beta) L_c$.

3. Topological Regularizer (Differentiable WD): To enforce manifold properties and bridge the gap to high–performing weights efficiently, incorporating the topological distance that showed success in my convergence experiments :the Wasserstein Distance (WD) is essential. The WD metric on persistence diagrams is the large–scale version of optimal transport distance. Since full WD is often non–differentiable, you should employ the Sliced Wasserstein Distance (SWD), which is known to be differentiable. $L_{Topological} = \alpha \cdot \mathrm{SWD}(\mathrm{PH}(\hat{W}), \mathrm{PH}(W_{target}))$ Where $\hat{W}$ are the predicted weights and $W_{target}$ are the ground truth/optimal weights.

4. Weight Importance Regularization (Fisher Information): Given my work's focus on continual learning and efficient representations, a term promoting finetunable weights is necessary. The Fisher Information Matrix (F) provides information about weight importance. Adopting an Elastic Weight Consolidation (EWC) style regularization term, where parameters are penalized based on their importance (F) to previous tasks, aligns perfectly with penalizing deviation from desired weight structures. $R = \sum_i \frac{\gamma}{2} F_i (\hat{w}_i - w_i)^2$ Where $w_i$ are the important weights/embeddings from the encoder stage (sub–partitions) and $F_i$ is the

corresponding diagonal Fisher information value. Recommended Composite Loss: A mixture that targets both precise reconstruction and topological/functional relevance: $L_{Total} = \beta L_{\overline{MSE}} + (1-\beta)L_c + \alpha L_{Topological} + R$

Summary of Differentiable Topological Descriptors :

The complexity of TDA features often requires turning the persistence diagram (PD) into a vectorization or using specific differentiable measures to integrate into gradient-based learning pipelines. The following table summarizes key descriptors and their suitability for Transformer training (where differentiability is crucial): Table 1: Topological Descriptors and Suitability for Transformer Training

**Table 1.** Topological Descriptors and Suitability for Transformer Training

| TDA Feature | Description | Differentiable? | Notes/Context |
|---|---|---|---|
| Persistence Diagram/Barcode | Multiset of birth/death points. | Partial. Computation involves non-smooth operations. Requires implicit differentiation libraries (e.g., Torchph). | Core descriptor, often used as input for differentiable vectorizations. |
| Persistence Landscape | Continuous, piecewise linear function summarizing PD features. | Yes. Allows for averaging and is useful for statistical analysis. | One of the most computationally tractable differentiable descriptors. |
| Persistence Image (Heat Kernel) | Grid-based vectorization of PD using Gaussian kernels. | Yes. Provides a smooth, fixed-size vector output highly valued in ML. | Implemented for differentiability in libraries like Perslay and giotto-tda. |
| (Sliced) Wasserstein Distance | Metric based on optimal transport between PDs. | Yes (Sliced and kernelized versions). Full bottleneck distance is generally non-differentiable. | Crucial for manifold learning and metric preservation. |
| Signed Barcodes/Hilbert Decomposition | Discrete signed measure characterizing the Hilbert function of a filtration. | Yes (as measures). | These multiparameter descriptors are shown to be semilinearly determined on grids, allowing for explicit Clarke subdifferentiability. |
| Multiparameter Persistence | Persistence indexed by $n > 1$ parameters (e.g., density + scale). | Yes (via specific descriptors/libraries like multipers). | Generally challenging, but differentiable invariants are emerging. |
| Topological Regularizer | Topological distance used as a loss function component. | Yes. | Directly designed for integration into gradient descent pipelines. |

Comparison of Simplicial Complexes The choice of simplicial complex construction determines the fidelity of the topological approximation and computational cost. For geometric data (like weights visualized as point clouds), Vietoris–Rips and related complexes are common. Table 2: Comparison of TDA Simplicial Complexes

**Table 2.** Topological Descriptors and Suitability for Transformer Training

| Complex Type | Input Data | Key Construction Rule | Advantages | Disadvantages |
|---|---|---|---|---|
| Vietoris–Rips (RIPS) | Point cloud + distance matrix in any metric space. | Clique complex: A $k$-simplex for every subset of $k+1$ points where all pairwise distances are $\leq$ radius $r$. | Easy to define; widely applicable; fast implementations exist (e.g., Ripser). | Computationally expensive; number of simplices grows combinatorially/exponentially with points. |
| Alpha Complex | Point cloud in $\mathbb{R}^d$. | Subcomplex of the Delaunay triangulation restricted by a radius $\alpha$. | Captures the true homotopy type of the union of balls; sparse; uses exact geometry. | Requires full Delaunay tessellation (even in 3D); restricted to Euclidean space. |
| ?ech Complex | Point cloud in $\mathbb{R}^d$. | A $k$-simplex if the intersection of $k+1$ balls of radius $r$ is non-empty. | Theoretically sound (Nerve Lemma); captures the true homotopy type of the union of balls. | Computationally expensive; difficult to materialize high-dimensional nerve structures. |
| Witness Complex | A large point cloud and a set of Landmarks $L$. | Approximates the Rips filtration by only considering proximity to landmarks. | Ultra-fast; scalable and memory efficient for billion-point clouds. | Coarse topological features; sensitive to the selection of landmarks. |
| Cubical Complex | Regular grid/voxel data (e.g., images). | Cells are hypercubes (pixels, voxels) based on scalar values. | Extremely efficient for structured data; ideal for medical imaging/CNN analysis on grids. | Only for structured grid data. |

Recent Works in TDA for Neural Network Analysis The field of TDA for Neural Networks focuses on four broad domains: 1. Architecture, 2. Input/Output Spaces, 3. Internal Representations (Weights/Activations), and 4. Training Dynamics/Loss Functions. My project primarily relates to domains 3 and 4

The following table, similar to the survey provided, summarizes relevant recent contributions: Table 3: Summaries of Recent Works and Contributions

| Title (Reference) | Summary / Contribution | Key Area / Cat. | Apps. |
|---|---|---|---|
| DIFFUSION-BASED NEURAL NETWORK WEIGHTS GENERATION (D2NWG) | Introduces a latent diffusion model leveraging knowledge from pretrained weights to efficiently generate task-conditioned, high-performing weights for transfer learning/initialization. | Generative Hyper-Rep. | - |
| Differentiability and Optimization of Multiparameter Persistent Homology | Developed a general theoretical framework proving that multiparameter persistence descriptors (like signed barcodes and persistence landscapes) are differentiable (semilinearly determined on grids), enabling their use in gradient descent pipelines. | TDA Theory / Optimization | - |

| Title (Reference) | Summary / Contribution | Key Area / Cat. | Apps. |
|---|---|---|---|
| Stable Vectorization of Multiparameter Persistent Homology using Signed Barcodes as Measures | Introduced the Hilbert decomposition signed measure ($\mu_{Hil}$) and rank decomposition signed measure as differentiable vectorizations of multiparameter persistence modules, offering stability guarantees. | Multiparameter TDA | - |
| Intrinsic dimension, persistent homology and generalization in neural networks | Showed that the persistent homology dimension (a geometric fractal dimension derived from PDs) of the space of training weights correlates significantly with the generalization capacity of NNs (lower dimension = better generalization). | Training Dynamics (4) | - |
| A topological regularizer for regularizers via persistent homology (Chen et al., 2019) | Proposed one of the first topological regularization terms for classifiers, using differentiable PH to remove weak connected components from the decision boundary, reducing overfitting. Regularization of neural networks by modifying their decision regions using differentiable persistent homology. | Loss Functions (4), (2) | (1) |
| Neural persistence: A complexity measure for deep neural networks using algebraic topology (Rieck et al., 2019) | Defined Neural Persistence as a measure of topological complexity derived from the weights of FCFNNs, finding that higher persistence correlates with better generalization, usable as an early stopping criterion. Topological complexity measure for neural networks based on their weights. | Internal Rep. (3) | (1) |
| Representation topology divergence: A method for comparing neural network representations | Proposed RTD, a measure to compare two Vietoris–Rips filtrations (representations) by counting how features merge, providing a dissimilarity measure between data representations. | Internal Rep. (3) | - |
| Topological dynamics of functional neural network graphs during reinforcement learning | Studied the evolution of Betti numbers derived from complexes induced by functional neural network graphs during RL training/inference. | Internal Rep. (3) | - |
| On the complexity of neural network classifiers: A comparison between shallow and deep architectures (Bianchini and Scarselli, 2014) | Bounds on the Betti numbers of the positive decision region generated by binary classification neural networks with Pfaffian activations. | (2) | - |
| Topological approaches to deep learning (Carlsson and Bru?el Gabrielsson, 2020) | Topological analysis of the weights of convolutional neural networks and generalization of convolutional neural networks. | (3) | - |
| Topological data analysis of decision boundaries with application to model selection (Ramamurthy et al., 2019) | Study and approximation of the topology of network decision boundaries. | (2) | (5) |
| Geometry Score: A method for comparing generative adversarial networks (Khrulkov and Oseledets, 2018) | Measurement of GAN quality using persistent homology. | (2) | (7) |
| What does it mean to learn in deep networks? And, how does one detect adversarial attacks? (Corneanu et al., 2019) | Study of generalization in terms of the persistent homology of neuron activations and their correlations. | (3) | (1, 3) |
| On connected sublevel sets in deep learning (Nguyen, 2019) | Study of the connectivity, boundedness, and local minima of sublevel sets of convex losses for overparameterised neural networks with piecewise linear activation functions. | (4) | - |
| Characterizing the shape of activation space in deep neural networks (Gebhart et al., 2019) | Topological characterization of the neuron activations given a sample. | (3) | (3) |
| Exposition and interpretation of the topology of neural networks (Bru?el Gabrielsson and Carlsson, 2019) | Topological analysis of the weights of convolutional neural networks and connection to generalization capacity of models. | (3) | - |
| Path homologies of deep feedforward networks (Chowdhury et al., 2019) | Analysis of path and directed flag homology groups of MLPs' directed graph. | (1) | - |
| Topology of deep neural networks (Naitzat et al., 2020) | Study of the topology of the data through layer transformations. | (3) | - |
| Finding the homology of decision boundaries with active learning (Li et al., 2020) | Use of active learning to improve the methods of Topological data analysis of decision boundaries with application to model selection. | (2) | (5) |
| Computing the testing error without a testing set (Corneanu et al., 2020) | Regression of test accuracy using topological vectorizations of persistence diagrams. | (3) | (6) |
| Topological detection of trojaned neural networks (Zheng et al., 2021) | Study of trojaned networks in terms of the persistent homology of neuron activations and their correlations. | (3) | (4) |
| Experimental stability analysis of neural networks in classification problems with confidence sets for persistence diagrams (Akai et al., 2021) | Study of persistence diagrams of last hidden layer activations and its connection with generalization. | (3) | - |
| PHom-GeM: Persistent homology for generative models (Charlier et al., 2019) | Comparison between persistence diagrams of real and generated manifolds using generative models. | (2) | (7) |
| TopoAct: Visually exploring the shape of activations in deep learning (Rathore et al., 2021) | Analysis of the Mapper graph of neuron activations for each layer. | (3) | - |
| Deep neural network pruning using persistent homology (Watanabe and Yamana, 2020) | Pruning of neural networks using persistent homology. | (3) | (2) |
| Intrinsic dimension, persistent homology and generalization in neural networks (Birdal et al., 2021) | Generalization error bounds using persistent homology dimension on the training weights. | (3, 4) | (1) |

| Title (Reference) | Summary / Contribution | Key Area / Cat. | Apps. |
|---|---|---|---|
| Activation landscapes as a topological summary of neural network performance (Wheeler et al., 2021) | Study of the topology of the data through layer transformations and its connection with training accuracy. | (3) | - |
| Topology of learning in feedforward neural networks (Gabella, 2021) | Analysis of the evolution of the weights of a neural network during training using Mapper. | (3) | - |
| Topological uncertainty: Monitoring trained neural networks through persistence of activation graphs (Lacombe et al., 2021) | Uncertainty measurement of neural network predictions using the topology of neuron activations. | (3) | (3, 5) |
| Topological measurement of deep neural networks using persistent homology (Watanabe and Yamana, 2022b) | Computation of persistence diagrams using neuron path relevance and connection with network expressivity and problem difficulty. | (3) | - |
| Evaluating the disentanglement of deep generative models with manifold topology (Zhou et al., 2021) | Measurement of disentanglement of generative neural networks. | (2) | - |
| Quantitative performance assessment of CNN units via topological entropy calculation (Zhao and Zhang, 2022) | Measurement of quality of convolutions in a neural network using persistent homology. | (3) | - |
| Overfitting measurement of deep neural networks using no data (Watanabe and Yamana, 2022a) | Study of overfitting by analysing the points near the diagonal of a persistence diagram generated from the weights of a neural network. | (3) | - |
| An adversarial robustness perspective on the topology of neural networks (Goibert et al., 2022) | Analysis of adversarial examples by means of the topology of the subgraph of under-optimized edges of neural networks. | (3) | (3) |
| Representation topology divergence: A method for comparing neural network representations (Barannikov et al., 2022) | Definition of similarity between data representations using persistent homology. | (3) | (7) |
| On the topological expressive power of neural networks (Petri and Leitao, 2020) | Measurement of expressivity of network architectures. | (2) | - |
| Generalization bounds using data-dependent fractal dimensions (Dupuis et al., 2023) | Data-dependent generalization error bounds using persistent homology dimension on the training weights. | (3, 4) | - |
| TopoBERT: Exploring the topology of fine-tuned word representations (Rathore et al., 2023) | Mapper graph of transformer-based models with applications in finetuning of language models. | (3) | - |
| Experimental observations of the topology of convolutional neural network activations (Purvine et al., 2023) | Analysis of the Mapper graph of neuron activations and definition of a similarity function between layers using sliced Wasserstein distances between persistence diagrams generated from them. | (3) | - |
| ReLU neural networks, polyhedral decompositions, and persistent homology (Liu et al., 2023b) | Detection of homological features in manifolds embedded in the input space of a neural network using the polyhedra decomposition induced by a ReLU feedforward neural network. | (2) | - |
| Caveats of neural persistence in deep neural networks (Girrbach et al., 2023) | Connection between neural persistence and variance measures of neural network weights. | (3) | - |
| On the use of persistent homology to control the generalization capacity of a neural network (Barbara et al., 2024) | Regression of generalization gap using the average persistence of zero dimensional persistence diagrams. | (3) | (6) |
| Visualizing and analyzing the topology of neuron activations in deep adversarial training (Zhou et al., 2023) | Analysis of the Mapper graph of neuron activations of normally and adversarially trained neural networks. | (3) | - |
| TopP&R: Robust Support Estimation Approach for Evaluating Fidelity and Diversity in Generative Models (Kim et al., 2023) | Approximation of the precision and recall scores for generative models using a persistent homology-based approach. | (2) | (7) |
| Topological structure of complex predictions (Liu et al., 2023a) | Analysis of the GTDA Reeb network of output spaces of neural networks. | (2) | - |
| Topological dynamics of functional neural network graphs during reinforcement learning (Muller et al., 2024) | Study of reinforcement learning neural networks during training and inference using homology. | (3) | - |
| Topological dynamics of functional neural network graphs during reinforcement learning (Muller et al., 2024) | Study of reinforcement learning neural networks during training and inference using homology. | (3) | - |
| Fast Witness Persistence for MRI Volumes via Hybrid Landmarking (Jorge Ruiz Williams 2025)<br><br>A Framework for Fast and Stable Representations of Multiparameter Persistent Homology Decompositions (David Loiseaux et al 2023 ) | The paper introduces a scalable GPU-accelerated method for computing persistent homology on full-brain MRI volumes using hybrid density-aware landmark selection and witness complexes, achieving runtime under 10 seconds while preserving topological features that traditional methods would take hours to compute.<br>- | -<br><br>- | -<br><br>- |
| Multi-parameter Module Approximation: an efficient and interpretable invariant for multi-parameter persistence modules with guarantees (David Loiseaux et al 2024) | - | - | - |
| Stable Vectorization of Multiparameter Persistent Homology using Signed Barcodes as Measures (David Loiseaux et al 2023) | - | - | - |

| Title (Reference) | Summary / Contribution | Key Area / Cat. | Apps. |
|---|---|---|---|
| Differentiability and Optimization of Multiparameter Persistent Homology (Luis Scoccola et al 2024) | - | - | - |

Table 1: Table containing the published articles reviewed in this survey ordered by year of publi- cation. Each row corresponds to an article. The first column contains the title; the second column contains a one-line summary; the third column contains the categories introduced in Section 1 as- sociated to each article; and the last column contains the applications introduced in Section 2.4. The four possible categories are 1. Structure of the neural network; 2. Input and output spaces; 3. Internal representations and activations; 4. Training dynamics and loss functions. The seven possible applications are 1. Regularization; 2. Pruning of neural networks; 3. Detection of adversar- ial, out-of-distribution and shifted examples; 4. Detection of trojaned networks; 5. Model selection; 6. Prediction of accuracy; 7. Quality assessment of generative models.

The Mapper algorithm

The breakthrough in utilizing two-dimensional (2D), or more generally multiparameter, persistent homology (PH) stems fundamentally from overcoming a major theoretical limitation of classical one-parameter PH, and subsequently developing differentiable descriptors that allow these rich invariants to be integrated into machine learning optimization tasks .

1. The Core Limitation of 1D Persistence In classical persistent homology (often called one-parameter persistence), data is filtered along a single dimension, such as increasing radius (?) or density (?) .

This results in a sequence of nested spaces, known as a filtration. The key theoretical breakthrough for 1D persistence was the establishment of a complete discrete invariant, known as the persistence barcode (or persistence diagram) .

This barcode uniquely and completely encodes all the topological information within the filtration, up to isomorphism. However, many real-world structures are defined by multiple interacting parameters, leading to a multifiltration (e.g., parameterized along curvature ? and radius ?, or scale and density) .

If one attempts to use 1D PH on such a structure, one must fix the value of the other parameters, which risks losing valuable information or requiring arbitrary parameter choices. 2. The Negative Result (The Theoretical Wall) The first step toward the 2D PH breakthrough was the critical realization by Gunnar Carlsson and Afra Zomorodian (2009) that the simplicity and completeness of the 1D barcode do not generalize to higher dimensions.

No Complete Discrete Invariant: Due to fundamental algebraic reasons, no similar canonical complete discrete invariant (like the barcode) exists for n–valued filtering functions when n>1 (e.g., for 2D persistence). The classification of multigraded modules (the algebraic structure corresponding to multifiltrations) involves both discrete and continuous portions, and the continuous portion has no precise, field-independent parameterization.

Field Dependence: The classification in higher dimensions can be dependent on the choice of the coefficient field (k), meaning changing the field changes the classification and the target of the classification, preventing the existence of a complete discrete invariant. This negative result meant that new mathematical tools were needed to capture persistent topological features in n–parameter settings. 3. The Shift to Incomplete, but Useful, Invariants Given the impossibility of a complete discrete invariant, the breakthrough shifted focus to developing invariants that are computable, compact, and effective for extracting persistent features–even if they are incomplete. The proposed alternatives measure the relationships (homomorphisms) between the filtered spaces, rather than just the birth/death times.

The Rank Invariant:

Carlsson and Zomorodian proposed the rank invariant (?X,i), a discrete invariant defined as the rank of the structure maps between homology groups.

Utility: The rank invariant extends naturally to higher dimensions, remaining a homeomorphism invariant of the multifiltered space. In 1D, it is mathematically equivalent to the barcode, proving its completeness in that case. In 2D and higher, it is used to identify persistent features by looking for points far from the diagonal with a neighborhood of constant value.

Limitation: Unlike the barcode, the rank invariant is incomplete in higher dimensions. B. Emergence of Differentiable Descriptors A subsequent major practical breakthrough was developing methods to translate multiparameter PH structures into vectorizations that are differentiable for gradient-based optimization. This is crucial for integrating TDA topological insights into machine learning loss functions or layers.

1. Signed Barcodes as Measures (Hilbert Decomposition Signed Measure): These generalizations of the 1D barcode allow for representation in vector space. Works like Loiseaux et al. (2023b) developed techniques for stable vectorization of signed barcodes. 2. Multiparameter Persistence Landscapes (MPPL): This is a functional descriptor extending the 1D persistence landscape. 3. Multiparameter Persistence Image (MPI): This descriptor, which uses a convolution of a signed measure with a Gaussian kernel, is specifically designed for machine learning. 4. Differentiability and Optimization: The Final Breakthrough The definitive achievement that made 2D/multiparameter PH widely applicable in machine learning was providing a general, unified framework for differentiability and optimization.

Semilinear Determination on Grids: The paper "Differentiability and Optimization of Multiparameter Persistent Homology" (Scoccola et al., 2024) introduced a framework proving that a broad class of descriptors–those characterized as being semilinearly determined on grids–are semilinear maps with an explicit Clarke subdifferential.

This category encompasses powerful multiparameter invariants like signed barcodes as measures and the multiparameter persistence landscape.

Gradient Convergence Guarantees: This theoretical result generalizes the existing optimization framework for 1D barcodes and provides the sound mathematical foundation necessary for optimizing multi-parameter topological terms via gradient descent.

Improved Performance: Numerical experiments show that optimizing multiparameter homological descriptors tends to outperform their one-parameter counterparts in machine learning tasks. For instance, using a function-Rips bifiltration (metric + density) in optimization experiments prevents point clouds from diverging and preserves density-dependent structures, which single-parameter Rips optimization cannot achieve.

In summary, the breakthrough of using 2D persistent homology (multiparameter PH) was a two-step process: first, accepting the algebraic necessity of using incomplete invariants (like the rank invariant) instead of a complete barcode; and second, developing differentiable vectorizations and a comprehensive theoretical framework that allows these complex topological structures to be reliably and robustly optimized using standard gradient-based methods in deep learning.

### 5.5. Random Matrix Theory:

Random Matrix Theory (RMT) is a field of algebraic probability and mathematical physics that studies the properties of matrices whose entries are random variables, especially as the matrix dimensions become large . It is fundamentally a statistical approach: instead of analyzing the detailed laws of a complex system, RMT replaces deterministic matrices with random ones and calculates averages and other statistical properties of their eigenvalues and eigenvectors

1. Definition and Core Focus A random matrix is simply a matrix whose elements are random variables

RMT focuses on analyzing the statistical properties of these matrices, primarily their eigenvalues and eigenvectors . RMT is essential when dealing with systems too complicated for detailed study, such as the Hamiltonian of a heavy nucleus or complex financial data . Instead of attempting a deterministic prediction, RMT provides a statistical description . 2. Historical Context and Applications RMT gained attention primarily in nuclear physics . Wigner's Contribution: Eugene Wigner introduced random matrices to model the nuclei of heavy atoms . Wigner postulated that the spacing between energy levels (eigenvalues of the Hamiltonian) in complex systems should resemble the spacing between the eigenvalues of a random matrix, depending only on the symmetry class of the system . Wigner's Semicircle Law: Wigner showed that for large as $N \to \infty$, the average spectral density $\rho(x)$ of many such matrices converges to the universal semicircle distribution . Wishart Matrices: John Wishart introduced random matrices to estimate covariance matrices in multivariate statistics, predating Wigner's physics applications . These matrices form the Wishart-Laguerre ensemble . RMT has wide applications, including: financial data analysis , quantum chaos (the BGS conjecture) , computational neuroscience (modeling synaptic connections) , and high-dimensional statistics .

3. Key Concepts in RMT A. Ensembles and Symmetry Classes Random matrices are grouped into ensembles based on the distribution of their entries and symmetry requirements . The most commonly studied are the Gaussian ensembles (GOE, GUE, GSE), categorized by the Dyson index ($\beta$) : Gaussian Orthogonal Ensemble (GOE): Real symmetric matrices ($\beta = 1$) . Invariant under orthogonal conjugation . Models Hamiltonians with time-reversal symmetry . Gaussian Unitary Ensemble (GUE): Complex hermitian matrices ($\beta = 2$) . Invariant under unitary conjugation . Models Hamiltonians lacking time-reversal symmetry . Gaussian Symplectic Ensemble (GSE): Hermitian quaternionic matrices ($\beta = 4$) . Invariant under symplectic conjugation .

The Dyson index $\beta$ is equal to the number of real variables needed to specify one entry of the matrix .

**B. The Joint Probability Density Function (JPDF) of Eigenvalues**

For rotationally invariant ensembles, the JPDF of the eigenvalues, $\rho(x_1, \ldots, x_N)$, contains two competing terms: **confinement** and **repulsion**.

$$\rho(x_1, \ldots, x_N) = \frac{1}{Z}\left(\sum_{i=1}^{N} x_i^2\right)\prod_{j<k}|x_j - x_k|^{\beta} \qquad \text{(beta model)}$$

**Confinement**: The Gaussian exponential factor $\exp(-\frac{1}{2}\sum x_i^2)$ pushes eigenvalues toward the origin.

**Repulsion**: The Vandermonde determinant term $\prod_{j<k}|x_j - x_k|^{\beta}$ enforces level repulsion, meaning the probability of two eigenvalues being very close ($\delta \to 0$) is very small. This repulsion is raised to the power $\beta$, confirming that the strength of repulsion depends on the symmetry class.

**C. Spectral Density and Limiting Laws**

The spectral density $S\rho(x)$ (or average density of states) is the histogram profile of eigenvalues as N $\to \infty$ This is the marginal density of the JPDF $S\rho(x) = \int \rho(x, x_2, \ldots, x_N)\, dx_2 \cdots dx_N$.

- **Wigner's Semicircle Law**: For large Gaussian (Wigner) matrices, the rescaled spectral density converges to $S\rho(x) = \frac{4}{\pi}\sqrt{1 - x^2}$ over the interval $[-2, 2]$.
- **Marchenko–Pastur (MP) Law**: For Wishart-Laguerre matrices (rectangular), the limiting spectral density is the MP distribution $S\rho(y)$ with law characterized by the eigenvalue behavior of covariance matrices when $N$ and $M$ both tend to infinity while their aspect ratio $c = N/M \leq 1$ remains fixed.

D. The Spectrum Method in Neural Networks In the context of transformer networks, RMT is used through the spectrum method . This involves: 1. Treating layers (or groups of functionally similar layers, like query layers) as rectangular random matrices . 2. Computing the covariance matrix $\Sigma = W^T W/n$ of the layer weight matrix $W$ and extracting its eigenvalues (or singular values) . 3. Comparing the resulting empirical spectrum to the theoretical Marchenko-Pastur (MP) distribution . 4.Interpretation: Eigenvalues lying within the MP bounds are interpreted as noise, suggesting their corresponding principal components carry little meaningful information, and thus the layer has lower importance .

Part 2: Concepts to Retain Given my thesis focus on weight regression and continual learning using Transformers, the most important RMT concepts to retain are those connecting matrix properties to information content and generalization: 1. Marchenko-Pastur Distribution as the Null Hypothesis The Marchenko-Pastur (MP) distribution is the theoretical cornerstone for analyzing layer weights because they are typically represented as rectangular matrices or covariance matrices ($\Sigma = W^T W/n$) .

Concept: The MP distribution defines the expected distribution of eigenvalues for a purely random covariance matrix .

Retention: Any eigenvalue of a weight matrix $W$ that falls within the bounds predicted by the MP law ($\lambda \in [\lambda_-, \lambda_+]$) is considered consistent with noise . Conversely, eigenvalues outside these bounds represent signal or genuine structure, which is the information learned by the neural network . This provides a powerful metric for distinguishing useful model capacity from useless noise. 2. Eigenvalue Repulsion vs. Feature Clustering RMT highlights that structured matrices exhibit level repulsion ($\prod|x_j - x_k|^{\beta}$), meaning eigenvalues are regularly spaced and avoid clustering .

Concept: In the context of neural network weights, features that are highly correlated (or functional units that are functionally redundant) would lead to clustered eigenvalues or zero eigenvalues in the covariance matrix. The strong repulsion observed in standard RMT ensembles suggests a fundamental structuring principle for matrices defined by independent entries (like GOE/GUE) .

Retention: Analyzing the eigenvalue spacing of my predicted weights could serve as a proxy for the structural robustness of the resulting model. Weights that are "too random" might exhibit the characteristic spacing of RMT models, whereas functional, learned weights (which often exhibit redundancy or compression) might deviate from these universality classes . 3. The Resolvent Method and Spectral Density The resolvent or Stieltjes transform $G(z)$ is a key function used to analytically calculate the average spectral density $\rho(x)$ for large $N$, avoiding complex integral calculations .

Concept: The spectral density $\rho(x)$ can be recovered from the imaginary part of the resolvent $G(z)$ using the Sokhotski-Plemelj formula . For large $N$, the resolvent often satisfies a simple algebraic equation .

Retention: The analytic approach used in RMT (like solving for the resolvent via the Coulomb gas technique or matrix integration) demonstrates how global spectral properties (the shape of the $\rho(x)$) are derived from the local statistical properties of the matrix entries . This methodology could inspire techniques for analytically approximating the spectral distribution of the Transformer-generated weight matrices, thus giving insight into the capacity and scale of the generated hyper-representations . 4. Eigenvector Localization (IPR) and Mobility Edges The concept of eigenvector localization is measured by the Inverse Participation Ratio (IPR), $I_{N,j}$ . Concept: IPR quantifies how "spread out" an eigenvector is: $I_{N,j} \approx 1/N$ means the state is extended (all components are relevant), while $I_{N,j} \approx 1/s$ (for small $s$) means the state is localized (only a few components are active)

Retention: The localization properties of the eigenvectors associated with the weights could correlate strongly with the redundancy, efficiency, or modularity of the generated CNNs. A low IPR for the principal components (extended states) might indicate efficient use of capacity, whereas highly localized states might suggest specific, compartmentalized knowledge within the network, potentially useful for understanding which weights are finetunable . The concept of a mobility edge (the critical eigenvalue separating extended/localized states) is particularly valuable for this analysis .

### 5.6. (

kolmogorov arnold networks KAN)

Since the diffusion paper shows that finetuning is needed for CNNs to have good performance . we decide to keep weights before and after finetuning and since it's aprocess that happens extremly fast we assume the underlying function is simple. simple enough to be approximated by Kalmagrov arnold networks . The theory supporting my approach–approximating the fast, functionally simple weight transformation during fine-tuning using Kolmogorov-Arnold Networks (KANs)–rests on the Kolmogorov-Arnold Representation Theorem (KAT), the demonstrated efficiency of hyper-representation models like D2NWG and Hyper–Representations, and insights into the simplicity and manifold structure of neural network solutions. Here is a detailed explanation of the theoretical basis that could make this work, drawing specifically on the sources provided.

The Theoretical Foundation: KANs for Weight Transformation My core assumption is:

1. The performance gap between predicted weights ($W_{\text{predicted}}$) and high–performing fine–tuned weights ($W_{\text{finetuned}}$) is bridged quickly during fine-tuning (e.g., in one epoch). 2. This fast transformation implies the underlying function mapping$W_{\text{predicted}}$ to $W_{\text{finetuned}}$ is relatively simple (or low-dimensional). 3. This simple transformation function can be effectively approximated by a Kolmogorov-Arnold Network (KAN). 1. Kolmogorov-Arnold Representation Theorem (KAT) The primary mathematical theory underpinning the KAN architecture is the Kolmogorov-Arnold Representation Theorem (KAT). The KAN Architecture: The KAN architecture directly embodies this theorem, using learnable activation functions (often parametrized by B-splines) on the edges, rather than fixed activation functions (like ReLU) on the nodes as in a traditional Multi-Layer Perceptron (MLP). The functions ?q,p(xp) and ?q are the 1D activation functions learned on edges. Simplicity and Interpretability: KANs are proposed as alternatives to MLPs, empirically showing they can achieve better accuracy, faster numerical scaling laws, and greater interpretability in terms of accuracy and interpretability trade-offs on various science tasks. For problems where the underlying function is structurally simple (e.g., following a known symbolic formula), KANs can discover this exact structure automatically.

2. Application to Weight Transformation You hypothesize that the function mapping the generated weights (W) to the high-performing, fine-tuned weights is simple. This simplicity is evidenced by the fast convergence observed when fine-tuning generated hyper-representations: Fast Fine-Tuning: Weights generated by models like D2NWG or SKDE often accelerate convergence, achieving superior performance in a few epochs (e.g., 25 epochs) compared to training from random initialization (e.g., 50 epochs). The need for only a couple hundred steps (essentially a single epoch) to reach desirable performance confirms the speed of this transformation. Simplicity Implication: This suggests that the generated weights $W_{\text{predicted}}$ are already located on or very close to the low-dimensional

manifold of high-performing weights. The quick fine-tuning step is merely a short, simple trajectory that moves the weights from the predicted point on the manifold to a nearby optimal local minima W?.

KAN as the Simple Map: Since the required correction is rapid and localized, the functional difference F($W_{\text{predicted}}$)=$W_{\text{finetuned}}$ might be simple enough to be effectively captured by a KAN. KANs excel at fitting functions where the intrinsic dimensionality is low or the function has a compact compositional structure. If the fine–tuning trajectory represents a simple, low–complexity change in the high-dimensional weight space, a KAN could serve as a model for the fine–tuning operator OFT.

3. Connection to Loss Landscape and Manifold Structure The success of this approximation relies on the idea that the "simple function" F exists because the weights lie on a structured manifold. Weight Manifold: Neural network models populate a low dimensional manifold. Weights generated by hyper–representation models like D2NWG are hypothesized to lie on the same manifold as the original and fine-tuned weights, enabling the method to produce optimized weights without needing extensive fine–tuning. Loss Flatness and Simple Curves: The underlying weight transformation during fine–tuning likely follows simple curves connecting different optima. The final predicted weights $W_{\text{predicted}}$ are already placed in a region of the loss surface that leads to faster learning and better solutions. Your previous observation confirmed that MSE increases post-fine-tuning, but the Wasserstein Distance (WD) decreases meaningfully, suggesting the fine-tuning jump is geometrically effective, moving from a sub-optimal point $W_{\text{predicted}}$ to a functionally equivalent point $W_{\text{finetuned}}$ that resides on a flat loss region. KAN for Gradient-Free/Interpretable Layers: If the KAN is used to model the dense layer merging the embeddings or as a post–processor to refine the predicted weights, it introduces interpretable, non-gradient–based layers with fewer parameters. This aligns with your overall goal of creating a model that is more explainable and efficient.

4. Technical Feasibility (Differentiability) While KANs were initially presented as being interpretable and efficient for function fitting, their integration into a deep learning pipeline requires differentiability, which is supported by their implementation: Differentiable Operations: The KAN architecture, particularly the standard composition of L layers KAN(x)=?L?1????0x, is built on basic operations (summation, multiplication) and learnable activation functions parametrized by B-splines. Since all the operations are differentiable, KANs can be trained via backpropagation. This makes them theoretically suitable for integration into your Transformer's weight regression loss calculation, potentially as a differentiable refinement stage.

## References

[1] PFGPlots, *A latex package to create plots.* [Online]. Available: https://pgfplots.sourceforge.net/