

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

Ayemen Tlili^{a,1}

^aFaculty 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 repercussions 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 admissible conjecture about the universal approximation theorem benefits 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 moderately 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-incremental 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 constrain 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-incremental 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
 - Subtracting 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 Separate the topological(PH/in-layer distribution) and spectral(eigenvalue/FFT distribution) that allow finetuning to reach in 200-500 steps of the first pair of epochs high performance between Transformer predicted weights and finetuned/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 sparsely 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 separable 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 $Y = [Y_0, Y_1, \dots, Y_9]$ corresponding to the specific Classes C_i from the MNIST Dataset. The following sub-section elaborates how we fit a convolutional neural network model $CNN^{[Y_C]}$ hence having weights $w_{[g,i,e]}^{[l,Y_C]}$ with 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

label	0	1	2	3	4	5	6	7	8	9	gelu	relu	silu	tanh	sigmoid	leakyrelu	weight 0	weight 1	weight 2	bias 2462	bias 2463	Accuracy	epoch
1	[0.1]	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	-0.13655047	-0.32772434	98.64333333333333	36
2	[0.2]	1	0	1	0	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.0496985	-0.061642155	-0.0929891	0.19706774	-0.36267012	98.90666666666668	36
3	[0.3]	1	0	0	1	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.48978	-0.00564362	-0.105482616	0.088529125	-0.2061299	98.37666666666668	36
4	[0.4]	1	0	0	0	1	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	0.29166335	0.5139673	-0.07567718	-0.20371349	-0.28388706	98.91166666666668	36
5	[0.5]	1	0	0	0	0	1	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.24452844	0.029974932	0.52706844	0.08827021	-0.0042240657	98.86833333333334	36
6	[0.6]	1	0	0	0	0	0	1	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.0375004	-0.4844076	-0.27842575	-0.085808694	0.11616701	98.755	36
7	[0.7]	1	0	0	0	0	0	0	1	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.052341357	-0.064843126	0.19656326	-0.237271	-0.04033807	98.84666666666668	36
8	[0.8]	1	0	0	0	0	0	0	0	1	1.0	0.0	0.0	0.0	0.0	0.0	-0.53858924	-0.3099806	-0.19575508	-0.45016676	-0.29142836	98.79	36
9	[0.9]	1	0	0	0	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	0.44759452	0.30329853	0.07768698	-0.14843176	-0.29648837	98.67333333333332	36
10	[1.2]	0	1	1	0	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.059722804	0.360608	0.35631633	-0.07278538	-0.06793964	98.685	36
11	[1.3]	0	1	0	1	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.04179168	-0.21159662	-0.52923477	-0.5364756	-0.000998338	98.55	36
12	[1.4]	0	1	0	0	1	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	0.1668191	0.41012523	-0.24427213	-0.022639548	0.062193777	97.725	36
13	[1.5]	0	1	0	0	0	1	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	0.014314616	-0.08713338	0.21204941	-0.47954503	-0.218975	98.53666666666668	36
14	[1.6]	0	1	0	0	0	0	1	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-1.000273	-0.22925067	-0.32118472	-0.24637736	-0.040949624	98.51166666666668	36
15	[1.7]	0	1	0	0	0	0	0	1	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.21473292	0.23508567	-0.26331523	-0.6218089	-0.33946937	98.85166666666667	36
16	[1.8]	0	1	0	0	0	0	0	0	1	1.0	0.0	0.0	0.0	0.0	0.0	0.23771928	-0.1500995	0.27912033	-0.35365957	-0.37145048	98.52	36
17	[1.9]	0	1	0	0	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.21042645	-0.08642456	0.22186308	0.2798944	-0.19171506	98.90833333333332	36
18	[2.3]	0	0	1	1	0	0	0	0	0	1.0	0.0	0.0	0.0	0.0	0.0	-0.09061946	0.1488133	0.47424597	-0.45934552	-0.5223393	98.695	36

(a) 36468 row/model Kaiming Uniform init zoo early columns and Experience identifier label 1-hot encoding (b) Last columns and accuracy and the epoch identifier

Figure 1. The .csv format opted 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 robustness of the learned models. That being said, we fix the initial 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.

<pre> class CNN(nn.Module): def __init__(self): self.channels_in = 1 self.n_filters = 16 self.init_type = "uniform" self.dropout = 0.5 self.__init__() self.module_list = nn.ModuleList() self.module_list.append(nn.Conv2d(1, 16, 3, padding=1)) self.module_list.append(nn.ReLU(inplace=True)) self.module_list.append(nn.MaxPool2d(2, 2)) self.module_list.append(nn.Dropout(self.dropout)) self.module_list.append(nn.Conv2d(16, 16, 3, padding=1)) self.module_list.append(nn.ReLU(inplace=True)) self.module_list.append(nn.MaxPool2d(2, 2)) self.module_list.append(nn.Dropout(self.dropout)) self.module_list.append(nn.Conv2d(16, 16, 3, padding=1)) self.module_list.append(nn.ReLU(inplace=True)) self.module_list.append(nn.MaxPool2d(2, 2)) self.module_list.append(nn.Dropout(self.dropout)) self.module_list.append(nn.Flatten()) self.module_list.append(nn.Linear(16 * 3 * 3, 10)) self.module_list.append(nn.Dropout(self.dropout)) self.module_list.append(nn.Linear(10, 10)) self.initialize_weights(self.init_type) </pre>	<p>Weights</p> <p>weight (8×1×5×5)</p> <p>bias (8)</p> <p>Weights</p> <p>weight (6×8×5×5)</p> <p>bias (6)</p> <p>Weights</p> <p>weight (4×6×2×2)</p> <p>bias (4)</p> <p>Weights</p> <p>weight (20×36)</p> <p>bias (20)</p> <p>Weights</p> <p>weight (10×20)</p> <p>bias (10)</p>
---	---

(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

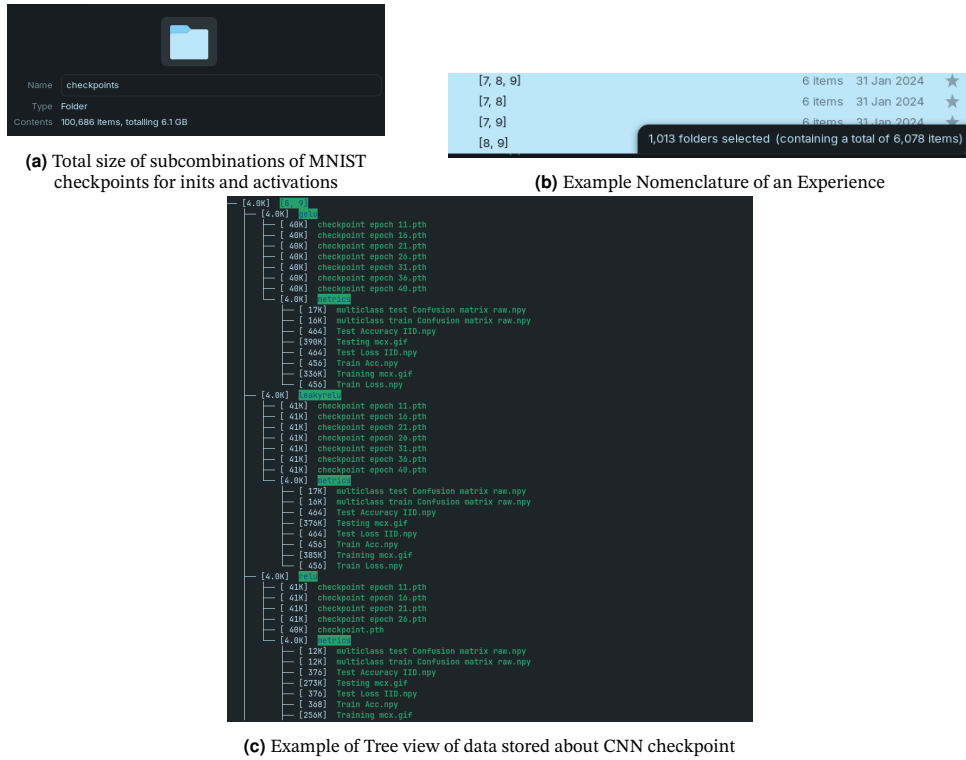


Figure 3. Showcasing the the Dataset checkpoints before building .csv format [1].

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

A training Scenario is finally decided by taking $S=(S_b, \text{epoch})$ 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 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 [I'll upload images later]

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-)

4. Meta-Learner Methodology:

4.1. Architecture:

model Argument	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

```

171 def get_clones(module, N):
172     return nn.ModuleList([copy.deepcopy(module) for i in range(N)])
173 class PositionalEncoder(nn.Module):
174     def __init__(self, d_model, max_seq_len=80, device='cuda'): #d"
175         super().__init__()
176         device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
177         self.d_model = d_model
178         self.device = device
179         # create constant 'pe' matrix with values dependant on
180         # pos and i
181         self.pe = self.generate_positional_encoding(max_seq_len, d_model)
182     def forward(self, x):
183         # make embeddings relatively larger
184         x = x * math.sqrt(self.d_model)
185         # add constant to embedding
186         seq_len = x.size(1)
187         # dynamically adjust positional encoding matrix based on sequence length
188         pe = self.pe[:, :seq_len]
189         x += pe.to(self.device)
190         x = x + pe
191     def generate_positional_encoding(self, max_seq_len, d_model):
192         pe = torch.zeros(max_seq_len, d_model)
193         position = torch.arange(0, max_seq_len, dtype=torch.float).unsqueeze(1)
194         div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) /
195         d_model))
196         pe[:, 0::2] = torch.sin(position * div_term)
197         pe[:, 1::2] = torch.cos(position * div_term)
198         pe = pe.unsqueeze(0)
199         return pe
200 def attention(q, k, v, d_k, mask=None, dropout=None):
201     scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)
202     if mask is not None:
203         mask = mask.unsqueeze(1)
204         scores = scores.masked_fill(mask == 0, -1e9)
205     scores = F.softmax(scores, dim=-1)
206     if dropout is not None:
207         scores = dropout(scores)
208     output = torch.matmul(scores, v)
209     return output, scores
210 class MultiHeadAttention(nn.Module):
211     def __init__(self, heads, d_model, dropout=0.1):
212         super().__init__()
213         self.d_model = d_model
214         self.d_k = d_model // heads
215         self.h = heads
216         self.q_linear = nn.Linear(d_model, d_model)
217         self.v_linear = nn.Linear(d_model, d_model)
218         self.k_linear = nn.Linear(d_model, d_model)
219         self.dropout = nn.Dropout(dropout)
220         self.out = nn.Linear(d_model, d_model)
221     def forward(self, q, k, v, mask=None):
222         bs = q.size(0)
223         # perform linear operation and split into h heads
224         k = self.k_linear(k).view(bs, -1, self.h, self.d_k)
225         q = self.q_linear(q).view(bs, -1, self.h, self.d_k)
226         v = self.v_linear(v).view(bs, -1, self.h, self.d_k)
227
228         k = k.transpose(1, 2)
229         q = q.transpose(1, 2)
230         v = v.transpose(1, 2)
231         scores, sc = attention(q, k, v, self.d_k, mask, self.dropout)
232         output, sc = self.out(scores)
233         return output, sc
234 class EncoderLayer(nn.Module):
235     def __init__(self, d_model, heads, normalize=True, dropout=0.1, d_ff=2048):
236         super().__init__()
237         self.normalize = normalize
238         self.norm1 = Norm(d_model)
239         self.norm2 = Norm(d_model)
240         self.attn = MultiHeadAttention(heads, d_model, dropout=dropout)
241         self.ff = FeedForward(d_model, d_ff=d_ff, dropout=dropout)
242         self.dropout1 = nn.Dropout(dropout)
243         self.dropout2 = nn.Dropout(dropout)
244     def forward(self, x, mask):
245         if self.normalize:
246             x2 = self.norm1(x)
247         else:
248             x2 = x.clone()
249         res, sc = self.attn(x2, x2, x2, mask)
250         # x = x + self.dropout_1(self.attn(x2, x2, x2, mask))
251         x = x + self.dropout1(res)
252         if self.normalize:
253             x2 = self.norm2(x)
254         else:
255             x2 = x.clone()
256         x = x + self.dropout2(self.ff(x2))
257         return x, sc
258 class EmbedderNeuronGroup(nn.Module):
259     def __init__(self, d_model, seed=22):
260         super().__init__()
261         #print("EmbedderNeuronGroup")
262         self.neuron_l1 = nn.Linear(16, d_model) #24
263         self.neuron_l2 = nn.Linear(80, d_model) #26
264     def forward(self, x):
265         return self.multiLinear(x)
266     def multiLinear(self, v):
267         #print("Multi-linear method", v.shape)
268         l = []
269         for ndx in range(26):
270             idx_start = ndx * 80
271             idx_end = idx_start + 80
272             l.append(self.neuron_l2(v[:, idx_start:idx_end]).clone())
273         # l2
274         for ndx in range(24):
275             idx_start = 26 * 80 + ndx * 16
276             idx_end = idx_start + 16
277             l.append(self.neuron_l1(v[:, idx_start:idx_end]).clone())
278         #print(len(l))
279         #print(len(l[0]))
280         final = torch.stack(l, dim=1)
281         # print(final.shape)
282         return final

```

(a) Helper functions and modules for Tranformer

```

317 class SeqVec(nn.Module):
318     def __init__(self, d_model, max_seq_len):
319         super().__init__()
320         self.d_model = d_model
321         self.max_seq_len = max_seq_len
322         # Define linear layers
323         self.linear = nn.Linear(max_seq_len * d_model, 2464)
324     def forward(self, x):
325         batch_size = x.size(0)
326         x = x.view(batch_size, -1) # Flatten the sequence dimension
327         x = self.linear(x)
328         return x
329 class Neck2Seq(nn.Module):
330     def __init__(self, d_model, neck, max_seq_length):
331         super().__init__()
332         self.neurons = nn.ModuleList([nn.Linear(neck, d_model) for _ in
333         range(max_seq_length)])
334     def forward(self, x):
335         l = [neuron(x) for neuron in self.neurons]
336         final = torch.stack(l, dim=1)
337         return final
338 class DecoderNeuronGroup(nn.Module):
339     def __init__(self, d_model, N, heads, max_seq_len, dropout, d_ff, neck):
340         super().__init__()
341         self.N = N
342         self.embed = Neck2Seq(d_model, neck, max_seq_len)
343         self.pe = PositionalEncoder(d_model, max_seq_len)
344         #print("decoder drouous init, dropout")
345         self.layers = get_clones(EncoderLayer(d_model,
346         heads, normalize=True, dropout=dropout, d_ff=d_ff), N)
347         self.norm = Norm(d_model)
348         self.lay = SeqVec(d_model=d_model, max_seq_len=max_seq_len)
349     def forward(self, src, mask=None):
350         scores = []
351         x = self.embed(src)
352         x = self.pe(x)
353         for i in range(self.N):
354             x, sc = self.layers[i](x, mask)
355             scores.append(sc)
356         return self.lay(self.norm(x)), scores
357 class FeedForward(nn.Module):
358     def __init__(self, d_model, d_ff=2048, dropout=0.1):
359         super().__init__()
360         # We set d_ff as a default to 2048
361         self.linear_1 = nn.Linear(d_model, d_ff)
362         self.dropout = nn.Dropout(dropout)
363         self.linear_2 = nn.Linear(d_ff, d_model)
364     def forward(self, x):
365         x = self.dropout(F.relu(self.linear_1(x)))
366         x = self.linear_2(x)
367         return x
368 class Norm(nn.Module):
369     def __init__(self, d_model, eps=1e-6):
370         super().__init__()
371         self.size = d_model
372         # create two learnable parameters to calibrate normalisation
373         self.alpha = nn.Parameter(torch.ones(self.size))
374         self.bias = nn.Parameter(torch.zeros(self.size))
375         self.eps = eps
376     def forward(self, x):
377         norm = (
378             self.alpha * (x - x.mean(dim=-1, keepdim=True))
379             / (x.std(dim=-1, keepdim=True) + self.eps)
380             + self.bias)
381         return norm
382 class TransformerAE(nn.Module):
383     def __init__(self):
384         self.max_seq_len=10,
385         N=1,
386         heads=1,
387         d_model=100,
388         d_ff=100,
389         neck=20,
390         dropout=0.1,
391         **kwargs):
392         super().__init__()
393         self.N=N
394         self.heads=heads
395         self.dropout=dropout
396         self.d_ff=d_ff
397         self.d_model=d_model
398         self.max_seq_len=max_seq_len
399         self.neck=neck
400         self.enc1 = EncoderNeuronGroup(d_model=self.d_model, N=self.N, heads=self.heads, max_seq_len=self.max_seq_len,
401         dropout=self.dropout, d_ff=self.d_ff)
402         self.enc2 = EncoderNeuronGroup(d_model=self.d_model, N=self.N, heads=self.heads, max_seq_len=self.max_seq_len,
403         dropout=self.dropout, d_ff=self.d_ff)
404         self.dec = DecoderNeuronGroup(d_model=self.d_model, N=self.N, heads=self.heads, max_seq_len=self.max_seq_len,
405         dropout=self.dropout, d_ff=self.d_ff, neck=self.neck)
406         #print("Addition Approach")
407         self.vec2neck = nn.Linear(self.d_ff*2, self.neck)
408         #print("Stack Approach")
409         self.vec2neck = nn.Linear(2*self.d_ff * self.max_seq_len, self.neck)
410         self.tanh = nn.Tanh()
411         for p in self.parameters():
412             if p.dim() > 1:
413                 nn.init.xavier_uniform_(p)
414         if torch.cuda.is_available():
415             self.cuda()
416     def forward(self, inp1, inp2):
417         out1, scEnc1 = self.enc1(inp1) #21 and attentions score
418         out2, scEnc2 = self.enc2(inp2) #22 and attention score
419         out3=torch.cat([out1,out2], dim=2) #concatenation to get dimenson n_output=n_input*2
420         sum=torch.sum(out3, dim=1, keepdim=False)
421         vec2=self.vec2neck(sum_r) #fusion dense layer
422         tanh = nn.Tanh()
423         neck_t=tanh(vec2)
424         out, scDec = self.dec(neck_t) #predicted weights , decoder attention scores
425         return out, neck_t, scEnc1,scEnc2, scDec

```

(b) Decoder Layer and Tranformer declaration

Figure 4. Transformer Architecture

```

encoder dropout init 0.07
encoder dropout init 0.07
decoder dropout 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 5. number of parameters per layer

4.2. Methodology:

we define the following :

- y_C : is the ground truth label by a CNN
- \hat{y}_C : is the predicted label by a CNN
- $\tilde{w}_{[g,i,e]}^{[l,Y_C]}$: are the ground truth weights of model from our zoo on I_C
- $\tilde{w}_{[g,i,e]}^{[l,Y_C]}$: are the weights for a predicted CNN by the Transformer
- $\tilde{w}_{[g,i,e]}^{[l,Y_C]}$: are the weights for a predicted CNN by the Transformer after finetuning on I_C
- \hat{z}_t^{neck-1} : 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:

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

with weights obtained via:

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

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

Given the pair:

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

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’

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_θ

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 $\{S_1, \dots, S_n\}$ 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_θ , 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_θ 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 quantiles robust to outliers, sensitive to tail behavior.

Compute the Qquantile loss between the two flattened weight (or gradient) vectors to assess how their distributions differ at quantile Q (e.g., median $Q=0.5$, or extreme tails $Q=0.01, 0.99$). This reveals whether one CNNs parameters are systematically larger/smaller in certain regions of the distribution, which can indicate differences in learning dynamics, sparsity, or robustness.

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

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.

Compute the Frobenius norm of the difference between the Jacobians of the two CNNs (w.r.t. inputs or parameters). This quantifies how differently the models respond locally useful for alignment, distillation, or stability analysis.

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

Minimizing this loss encourages the two networks to have similar local input/output 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 information useful for comparing learning trajectories, robustness, or generalization.

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

4.5.4. Contractive loss:

Penalizes sensitivity of hidden units to input encourages robust, invariant representations.

The contractive loss regularizes an autoencoder by minimizing the Frobenius norm of the encoders Jacobian w.r.t. input. Given weight matrix W and hidden activations $h=(Wx+b)$, it approximates $h \otimes h^T$ as $i(h_i(1-h_i))2Wi, :2$. Use it to compare two CNNs by computing this loss on their encoder-like layers or feature extractors lower values indicate smoother, more stable feature maps.

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

This loss approximates xhF2, promoting insensitivity to small input perturbations ideal for comparing stability of learned representations across models

4.5.5. Wasserstein Distance/Geomloss:

Measures geometric discrepancy between distributions with smooth, differentiable approximation.

Useful Implementation: Use the Sinkhorn (entropic-regularized) Wasserstein-2 distance to compare the two 1D distributions of flattened CNN weights or gradients. Despite being 1D, this formulation remains valid and differentiable, enabling use as a loss for aligning weight distributions, enforcing smoothness, or matching latent statistics during training. The entropic regularization (controlled by ϵ) ensures computational efficiency via the Sinkhorn algorithm.

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

where

$N_i = \sum_j P_{ij}$, $N_j = \sum_i P_{ij}$ are empirical measures from the two flattened vectors \mathbf{x}, \mathbf{y} , \mathcal{C} is the set of couplings (joint distributions with marginals μ, ν), $H(\mathbf{P}) = -\sum_{i,j} P_{ij} \log P_{ij}$ is the entropy of the transport plan, $\epsilon > 0$ controls regularization strength. The minimizer \mathbf{P} is obtained efficiently via the Sinkhorn iterations.

In 1D, the unregularized W_2 has a closed form ($W_2^2 = N \sum_{i=1}^N (x_{(i)} - y_{(i)})^2$, with sorted samples), but the Sinkhorn version is preferred when differentiability through the sorting operation is undesirable or when integrating into end-to-end training pipelines.

Thus, minimizing $W_2(\mathbf{x}, \mathbf{y})$ encourages the two CNNs to have statistically and geometrically aligned weight/gradient distributions.

4.5.6. Difference in Norm of the vector:

Measures global magnitude discrepancy, simple, scale-sensitive, ignores direction.

Useful Implementation: Compute the absolute difference between the norms of the two flattened weight (or gradient) vectors. This scalar loss captures whether one CNN has systematically larger or smaller overall parameter magnitude useful for detecting norm drift, regularization effects, or initialization bias.

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

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 fidelity, later chunks penalized relative to earlier prediction accuracy.

This autoregressive MSE loss treats the 2464-dimensional output as a sequence of chunks (e.g., layers or time steps). It computes per-chunk MSEs and then penalizes each subsequent chunk proportionally to its error relative to the previous chunks error, scaled by learnable or predefined weights. This encourages the model to maintain or improve accuracy over the sequence useful when comparing two CNNs weight trajectories or layer-wise reconstructions, where early layers should be reliably predicted before later ones.

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

Minimizing LAR-MSE aligns models not just in absolute error but in progressive consistency across structured segments of the weight vector ideal for layerwise or stagewise model comparison.

4.5.8. MAPE Loss:

Measures relative prediction error, scale-invariant, sensitive to small true values.

Useful Implementation: Compute the Mean Absolute Percentage Error (MAPE) between two flattened 2464-dimensional vectors (e.g., predicted vs. true weights or gradients). MAPE expresses average error as a percentage of the true magnitude, making it useful for comparing models across different scales or detecting systematic bias in parameter estimation. Avoid using it when true values contain zeros or near-zeros (add a small ϵ for stability).

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

A low MAPE indicates that the two CNNs agree closely in relative terms useful for model compression, weight transfer, or monitoring convergence in log-scale parameter spaces.

4.5.9. Layer-wise-loss Normalization:

Normalizes loss per layer to balance contribution across heterogeneous layer sizes.

Useful Implementation: When comparing two CNNs via a reconstruction, distillation, or weight-matching loss over their flattened 2464-dimensional parameter vectors, split the vector into per-layer segments (e.g., conv weights, biases, BN params). Compute a base loss (e.g., MSE) per layer, then normalize each by its layers parameter count (or another scale like Frobenius norm). Sum the normalized losses this prevents large layers from dominating and ensures fair layerwise alignment.

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

4.5.10. Jensen-Shannon Loss:

Symmetrized, smoothed measure of divergence between two probability distributions.

Treat the flattened weight (or gradient) vectors of the two CNNs as empirical probability distributions (e.g., by applying softmax or histogram binning). The Jensen-Shannon loss computes the square root of the Jensen-Shannon divergence (JSD) between the two distributions, a bounded, differentiable metric that captures both global and local distributional mismatches. Use it to align weight spectra, activation histograms, or gradient distributions in a way that is more stable than raw KL divergence.

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

Minimizing this loss encourages the two CNNs to have statistically similar weight or gradient distributions useful for model matching, ensemble diversity control, or comparing spectral densities in a probabilistic framework.

4.5.11. Fourier transform difference in Norm:

Compares global frequency content sensitive to periodic structure and long-range correlations.

Apply the FFT loss to the flattened 2464-dimensional weight or gradient vectors of two CNNs. By measuring the mean absolute difference of their discrete Fourier transforms, this loss captures discrepancies in spectral (frequency domain) structure e.g., smoothness, oscillation patterns, or alignment of large-scale parameter trends. Useful when weight distributions exhibit structured correlations (e.g., across layers or channels) that are invisible in pointwise norms.

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

Because the DFT is a unitary linear transform (up to scaling), this loss is equivalent to an l_1 norm in the frequency domain and emphasizes differences in global structure rather than local pointwise errors. Minimizing LFFT aligns the spectral signatures of the two CNNs valuable for analyzing topological regularities, enforcing smoothness, or matching weight-space dynamics in frequency.

4.5.12. Mel spectrogram Difference:

Compares timefrequency energy patterns robust to phase, emphasizes perceptually relevant structure.

Treat the flattened 2464dimensional weight or gradient vector as a 1D signal (e.g., layerwise concatenated weights). Compute its Mel spectrogram smoothed, perceptually motivated timefrequency representation using a shorttime Fourier transform (STFT) followed by Melscale filterbank integration. Then compute the L₂ (Frobenius) norm of the difference between the two Mel spectrograms. This loss captures discrepancies in structured spectral energy (e.g., clusters of large/small weights across time or layer index), which is more informative than raw FFT or pointwise norms when weight patterns exhibit localized bursts or smooth trends.

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

Minimizing L_{Mel} aligns the energy distribution across scales and positions of the two CNNs parameter vectors useful for comparing topological regularities, enforcing smoothness in weight evolution, or matching spectral textures in model distillation or generative weight modeling

4.5.13. Mel Spectrogram Fréchet inception distance:

Compares distributions of timefrequency features via Gaussianapproximated Fréchet distance captures both mean and covariance of Mel spectral statistics.

Useful Implementation: Treat each flattened 2464dimensional weight/gradient vector as a 1D signal, compute its Mel spectrogram, and then extract a feature vector (e.g., by flattening or using a pretrained encoder). Assuming these features are approximately multivariate Gaussian, the MelSpectrogram Fréchet Inception Distance (MSFID) is the Fréchet distance between the two Gaussian distributions estimated from two sets of CNNs (or two models). This provides a single scalar that accounts for both mean shifts (e.g., overall energy differences) and covariance mismatches (e.g., changes in spectral correlation structure) ideal for evaluating generative models of weights, comparing training trajectories, or assessing layerwise spectral similarity.

$$\text{MS-FID} = \|\mu_1 - \mu_2\|_2^2 + \text{Tr}(\Sigma_1 + \Sigma_2 - 2(\Sigma_1^{1/2} \Sigma_2 \Sigma_1^{1/2})^{1/2}),$$

Practical notes:

If only one sample per model is available (e.g., a single weight vector), estimate μ and Σ from subsegments (e.g., sliding windows over the Mel spectrogram) or use a pretrained spectral encoder to produce multiple patch embeddings. The metric is differentiable if the Mel spectrogram and feature extraction are implemented with differentiable ops (e.g., torchaudio.transforms.MelSpectrogram), enabling use as a training loss.

Minimizing MSFID encourages two CNNs (or a generator and a target) to produce weight/gradient signals with statistically indistinguishable timefrequency characteristics, making it powerful for spectral model comparison beyond simple L₂ or MAPE.

4.5.14. Gromov-wasserstein loss:

Compares intrinsic geometry of two distributions alignment-invariant, structure-aware.

Useful Implementation: Treat the two flattened 2464dimensional weight (or gradient) vectors as point clouds $X = \{x_i\}_{i=1}^N$ and $Y = \{y_j\}_{j=1}^N$ (with $N=2464$). Instead of comparing values directly (as in Wasserstein), the GromovWasserstein (GW) distance compares the pairwise relational structures e.g., distances or inner products within each point cloud. This is ideal when the absolute scale or labeling of parameters differs (e.g., permuted channels or layers), but you care about preserving topological or spectral similarity (e.g., clustering of large/small weights, smoothness patterns). Use GW as a loss to align CNNs up to relabeling or reordering of parameters.

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

In practice, one often uses the entropic-regularized version (SinkhornGromovWasserstein) for differentiability and efficiency:

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

with $L(i,j),(k,\ell) = d_X(x_i, x_k) d_Y(y_j, y_\ell)^p$.

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.

Minimizing the GromovWasserstein distance aligns the shape of the two weight distributions in a geometrically faithful way ideal for model comparison, compression, or generative modeling when parameter correspondence is unknown or irrelevant.

4.5.15. Bottleneck distance loss:

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

Useful Implementation: Compute persistence diagrams (e.g., from sublevel/superlevel filtrations) of the two 2464dimensional weight or gradient vectorstreating them as scalar fields over a 1D domain (e.g., layer index or parameter index). The bottleneck distance then quantifies the maximum displacement needed to match topological features (connected components, peaks, valleys) between the two diagrams. This captures differences in topological structure (e.g., number and scale of weight clusters, robust extrema) while being robust to small perturbations. Use it as a loss or evaluation metric when comparing CNNs based on their topological signatures.

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

Minimizing (or comparing via) the bottleneck distance ensures that the essential topological shapesuch as the number and prominence of weight clusters or gradient peaksis preserved between two CNNs, making it ideal for topological model comparison, pruning analysis, or generative modeling of neural parameters.

4.5.16. Latent:

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

Explanation of the Loss

You have two parallel encoders (enc1, enc2) that embed inputs (vec1, vec2, output[0], tg) into latent spaces z1,z2RBCELED. For each branch:

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

To normalize this pull, you relativize the prediction loss by the average reconstruction error of the two inputs: $Lx1 + Lx22Lpred$

This autoregressive-style term ensures the model only pays a high penalty if it fails to improve over the baseline inputs.

Post-merge consistency: Each target embedding is duplicated, summed, passed through a dense layer (vec2neck), and squashed with tanh to produce a merged bottleneck representation Z1tg,Z2tgRBCELED. These are compared to a shared target output[1] (presumably a ground-truth fused code) via MSE: $LZ1, LZ2$

Total loss: $L_{\text{total}} = (LZ1 + LZ2) + Lx1(1) + Lx2(1)2Lpred(1) + Lx1(2) + Lx2(2)2Lpred(2)$

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.

$$\mathcal{L}_{\text{total}} = \sum_{k=1}^2 \mathcal{L}_Z^{(k)} + \sum_{k=1}^2 \frac{2 \mathcal{L}_{\text{pred}}^{(k)}}{\mathcal{L}_{x1}^{(k)} + \mathcal{L}_{x2}^{(k)}}.$$

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 doesn't distort semantic content.

4.5.17. multipersistence L2 :

5. Appendix:

- 5.1. Optimal Transport Primer
- 5.2. Persistent Homology Primer
- 5.3. Random Matrix Theory:
- 5.4. earlier-work results: