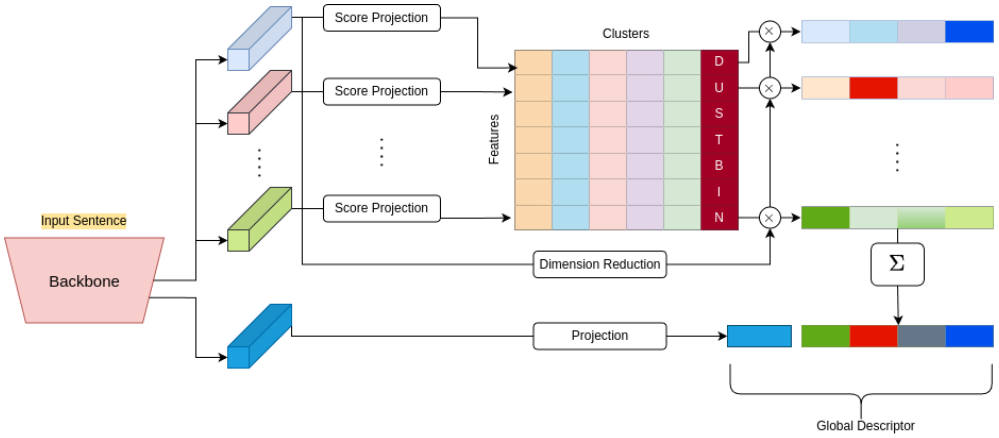
Final Report

Gnana Prakash Punnavajhala	Vaibhav Agrawal	Keshav Gupta
2021111027	2021101019	2021101018

Brief Recap - Improving Global Descriptors of Text Encoders

- For various applications, one might require a global representation of the entire sentence.
- Usually, representations such as the CLS token of BERT or the <EOT> token of CLIP are used for this purpose. However, these global descriptors are limited by their size (dimension) and the task on which they were trained upon, and therefore might not be able to capture enough information [1], [2].
- Our hypothesis is that the global information in a sentence can be captured in a better manner if we use other global aggregation schemes, which are popular in Computer Vision research [3], [4], [5].

Architecture



Backbones

- As part of the submission, we have coded up the following backbones:
  - BERT
  - RoBERTa
  - CLIP Textual Encoder
- We have also processed the sentences to these models using their corresponding tokenizers for optimal performance of each of these models
- To align with our final training pipeline, we have coded up these backbones in a very modular manner, so that they serve as drop-in to pass features to the aggregator
- Additionally, to enable efficient hyperparameter tuning further down the pipeline, we have made it very accessible to change the configurations of the model for the attributes we want to ablate on, namely,
  - Number of trainable blocks in the encoder - This is to understand the extent to which the model has to be finetuned to align the model with the dataset
  - To return the global token (CLS token for the BERT family of models) to the aggregator or not - To understand the extent to which these global tokens represent the sentences
- Below is a sample code snippet for the CLIP Textual Encoder:

```
class CLIP(nn.Module):
    """
    Class for the CLIP Text Encoder module.
    """
    def __init__(
            self,
            model_name="openai/clip-vit-base-patch32",
            num_trainable_blocks=0,
            return_global_token=True,
            cache_dir=None
        ):
        super().__init__()

        self.tokenizer = CLIPTokenizer.from_pretrained(model_name, cache_dir=cache_
dir)
        self.model = CLIPTextModel.from_pretrained(model_name, cache_dir=cache_dir)
        self.num_trainable_blocks = num_trainable_blocks
        self.return_global_token = return_global_token

        self.model.eval()
```

```
        for param in self.model.parameters():
            param.requires_grad = False

        if num_trainable_blocks > 0:
            for layer in self.model.text_model.encoder.layers[-self.num_trainable_b
locks:]:
                for param in layer.parameters():
                    param.requires_grad = True
```

Aggregator

- Our aggregator architecture is inspired from [6]
- Once we have the hidden layer features and (optionally) the global token of the sentence from the backbone, the aggregator first passes each of the hidden layer embeddings through projection layers to obtain their corresponding features across all clusters and the corresponding score associated with them using which the final global descriptor is created

```
x, t = x # Extract features and token

f = self.cluster_features(x).permute(0, 2, 1) # [B, l, n]
p = self.score(x).permute(0, 2, 1) # [B, m, n]
t = self.token_features(t) # [B, c]
```

- The Sinkhorn algorithm is then used to efficiently transform this data distribution to include an additional dustbin cluster, whose features are discarded so as to prevent gradients from propagating to the bad features

```
p = log_optimal_transport(p, self.dust_bin, 3) # [B, m + 1, n] -> +1 is from the du
stbin cluster
p = torch.exp(p)
p = p[:, :-1, :] # Remove dustbin cluster
```

- Once, we have the cluster features and the corresponding scores, we aggregate these descriptors using weighted sum coupled with intra normalization (within each cluster)
- Additionally, we concatenate this set of cluster descriptors with the projected global token embedding and perform inter normalization to finally obtain our global descriptor for the sentence

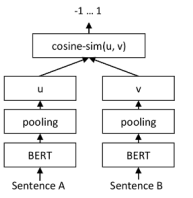
```
f = torch.cat([
    nn.functional.normalize(t, p=2, dim=-1),

    nn.functional.normalize((f * p).sum(dim=-1), p=2, dim=1).flatten(1)
], dim=-1)

return nn.functional.normalize(f, p=2, dim=-1)
```

Losses

Cosine Similarity Loss

- We refer to this loss from here
- The documentation for the same is as follows:

CosineSimilarityLoss



For each sentence pair, we pass sentence A and sentence B through our network which yields the embeddings *u* und *v*. The similarity of these embeddings is computed using cosine similarity and the result is compared to the gold similarity score.

This allows our network to be fine-tuned to recognize the similarity of sentences.

class `sentence_transformers.losses.CosineSimilarityLoss`*(model: sentence_transformers.SentenceTransformer.SentenceTransformer, loss_fct: torch.nn.modules.module.Module = MSELoss(), cos_score_transformation: torch.nn.modules.module.Module = Identity())* [source]

CosineSimilarityLoss expects that the InputExamples consists of two texts and a float label. It computes the vectors `u = model(sentence_A)` and `v = model(sentence_B)` and measures the cosine-similarity between the two. By default, it minimizes the following loss: `||input_label - cos_score_transformation(cosine_sim(u,v))||_2`.

Parameters:
- **model** – SentenceTransformer model
- **loss_fct** – Which pytorch loss function should be used to compare the `cosine_similarity(u, v)` with the input_label? By default, MSE is used: `||input_label - cosine_sim(u, v)||_2`
- **cos_score_transformation** – The cos_score_transformation function is applied on top of cosine_similarity. By default, the identify function is used (i.e. no change).

- This basically computes the MSE loss between the ground truth scores (in the range of [0, 1]) and the predicted scores, which are obtained by computing the cosine similarity score between the global embeddings of the two sentences

Binary Cross Entropy Loss

- Loss Formulation

$$-\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1-y_i) \cdot log(1-p(y_i))$$

- We use this loss, when we have a binary label to denote the relation between a pair of sentences - they are either similar or they are not
- We use the cosine similarity score between the embeddings of the two sentences as the predicted score (or probability) and the ground truth flag as the label

---

Datasets

SemEval24 Dataset Task A (Semantic Textual Similarity)

- This dataset provides sentence pairs along with their similarity scores in the range 0 to 1
- We use the cosine similarity loss to train the model on this dataset as we have scores for each pair

MTEB Dataset (Semantic textual similarity)

- Similar to the SemEval dataset, this dataset comprises of sentence pairs along with their corresponding similarity scores ranging from 0 to 5
- Hence, we directly employ the cosine similarity loss to train the model on this dataset
  - To ensure validity of the loss, we scale the scores to the range of 0 to 1

Quora Question Pairs (QQP) Dataset (Semantic Textual Similarity)

- This dataset provides a pair of sentences along with a flag indicating whether these questions are duplicates of each other or not
  - This implies that the flag will be 1 if they are duplicates, i.e., they are similar, else 0

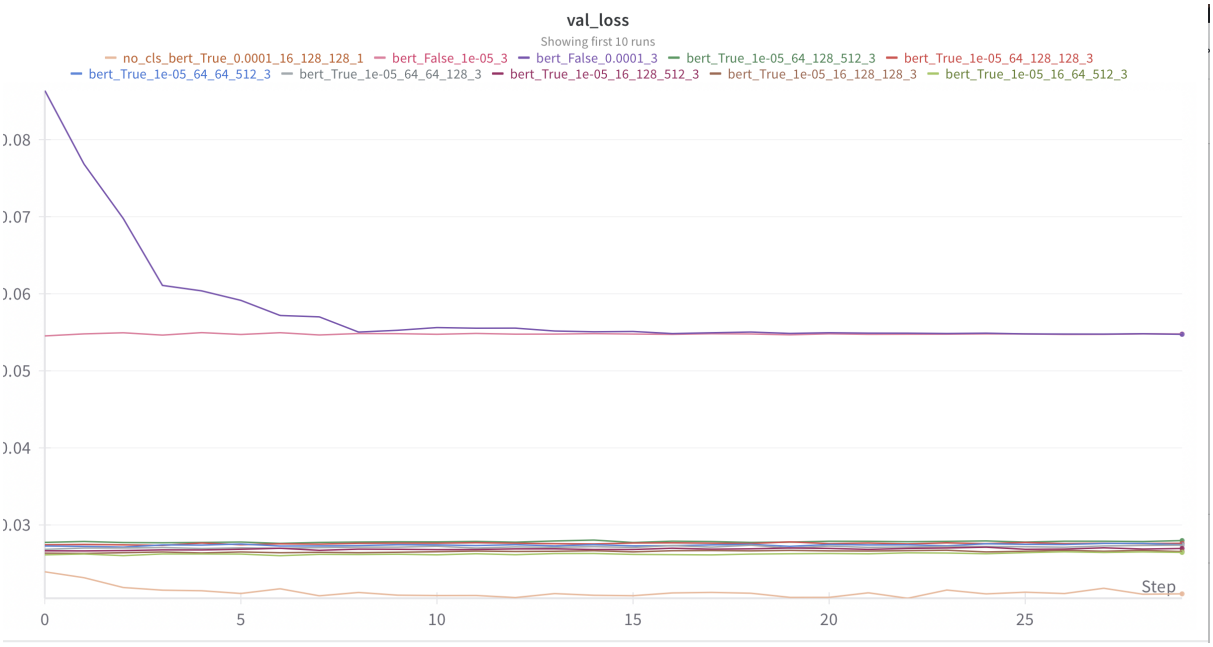
- Since we have a binary flag to denote the similarity, we use the binary cross entropy loss to train the model on this dataset
- This dataset has not been provided with a train-val-test split, so we split it in the ratio of 80-10-10 for the train-val-test datasets

```
def split_data(data):
    random.shuffle(data)

    total_size = len(data)
    train_size = int(total_size * 0.8)
    val_size   = int(total_size * 0.1)

    train_data = data[:train_size]
    val_data   = data[train_size:train_size + val_size]
    test_data  = data[train_size + val_size:]

    return train_data, val_data, test_data
```

Experiments

Hyperparameter Tuning

```
sweep_configuration = {
    "method": "grid",
    "metric": {
        "goal": "minimize",
        "name": "val_loss"
    },
    "parameters": {
        "lr": {
            "values": [1e-4, 1e-5]
        },
        "model": {
            "values": ["bert"]
        },
        "num_trainable_blocks": {
            "values": [1, 3]
        },
        "num_clusters": {
                "values": [16, 64]
        },
        "cluster_dim": {
                "values": [64, 128]
        },
        "token_dim": {
                "values": [128, 512]
        }
    }
}
```



- As we can see, there is visible difference in validation loss between using our aggregator and training just the baseline

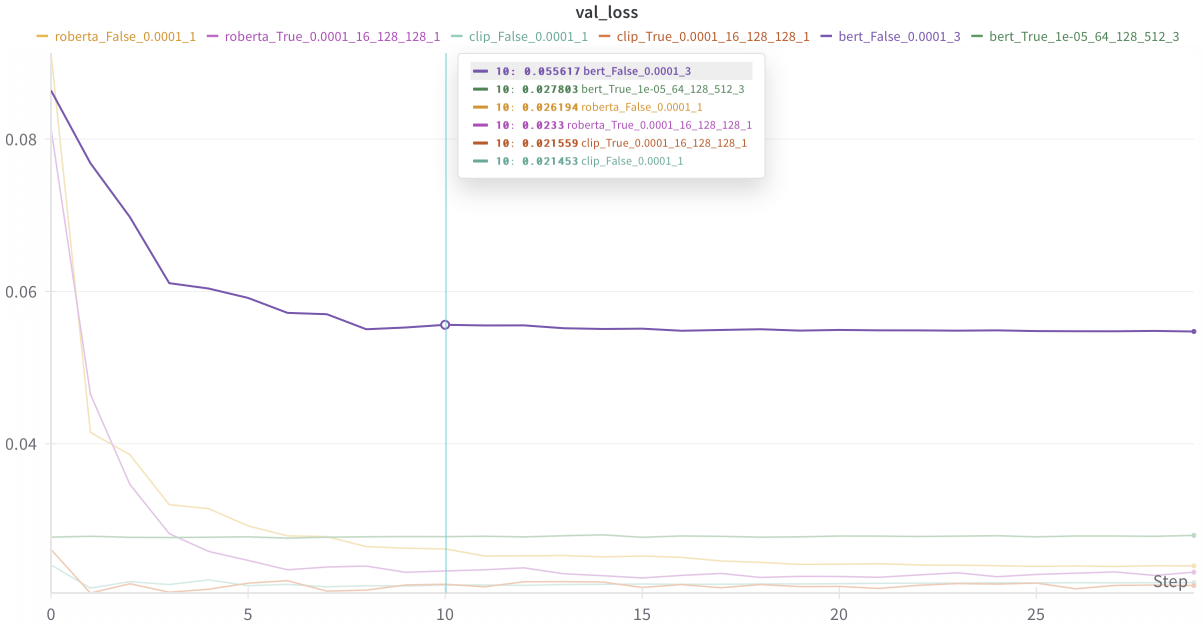
Best Params

```
lr: 1e-4
num_trainable_blocks: 1
num_clusters: 16
cluster_dim: 128
token_dim: 128
```

📣 We used these configurations for *all* models across *all* datasets

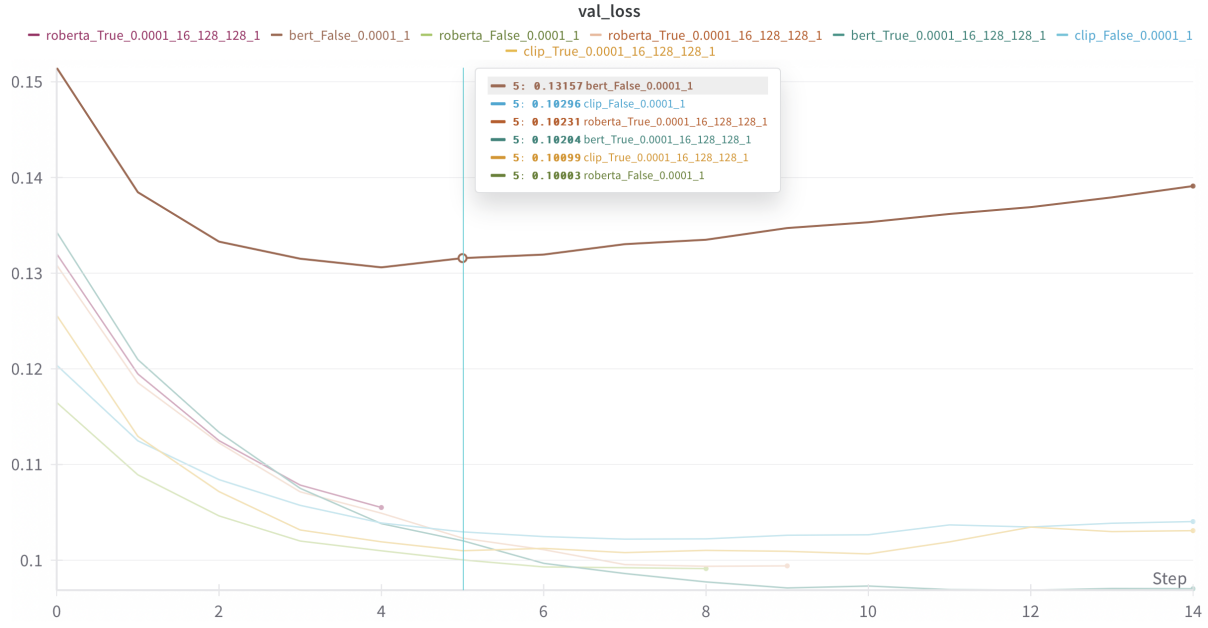
Performance on SemEval24 Dataset

Backbone	Aggregator	Val Loss	Test Loss
BERT	Used	0.268	0.272
BERT	Not used	0.534	0.529
Roberta	Used	0.238	0.235
Roberta	Not used	0.245	0.247
CLIP	Used	0.267	0.263
CLIP	Not used	0.269	0.269



Performance on QQP Dataset

Backbone	Aggregator	Val Loss	Test Loss
BERT	Used	**0.100**	**0.097**
BERT	Not used	0.138	0.132
Roberta	Used	0.092	0.102
Roberta	Not used	0.099	0.093
CLIP	Used	0.102	0.109
CLIP	Not used	0.103	0.108



Performance on MTEB Dataset

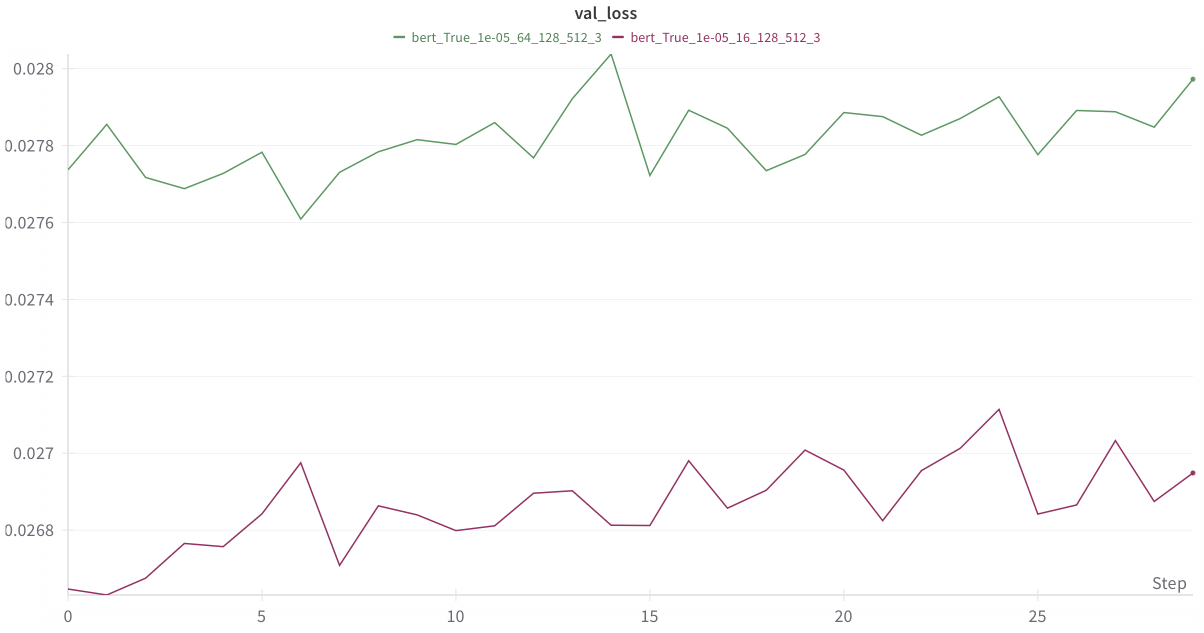
Backbone	Aggregator	Val Loss	Test Loss
BERT	Used	4.152	4.163
BERT	Not used	4.152	4.142
Roberta	Used	4.158	4.188
Roberta	Not used	4.157	4.172
CLIP	Used	4.144	4.141
CLIP	Not used	4.146	4.143

Analysis of Results

Hyperparameter Analysis

Number of clusters

- We have varied the number of clusters with `16` and `64`
- We get better performance by using lesser number of clusters, which can be interpreted as overfitting when there are larger number of clusters
- This choice might play a major part in the degradation of performance in CLIP because it deals with only `77` tokens, while BERT and RoBERTa deal with `512` tokens per sentence. This means that the same number of clusters is not the best choice for another model with such different configurations, as the number of clusters needed to represent the information contained in the tokens of CLIP might vary

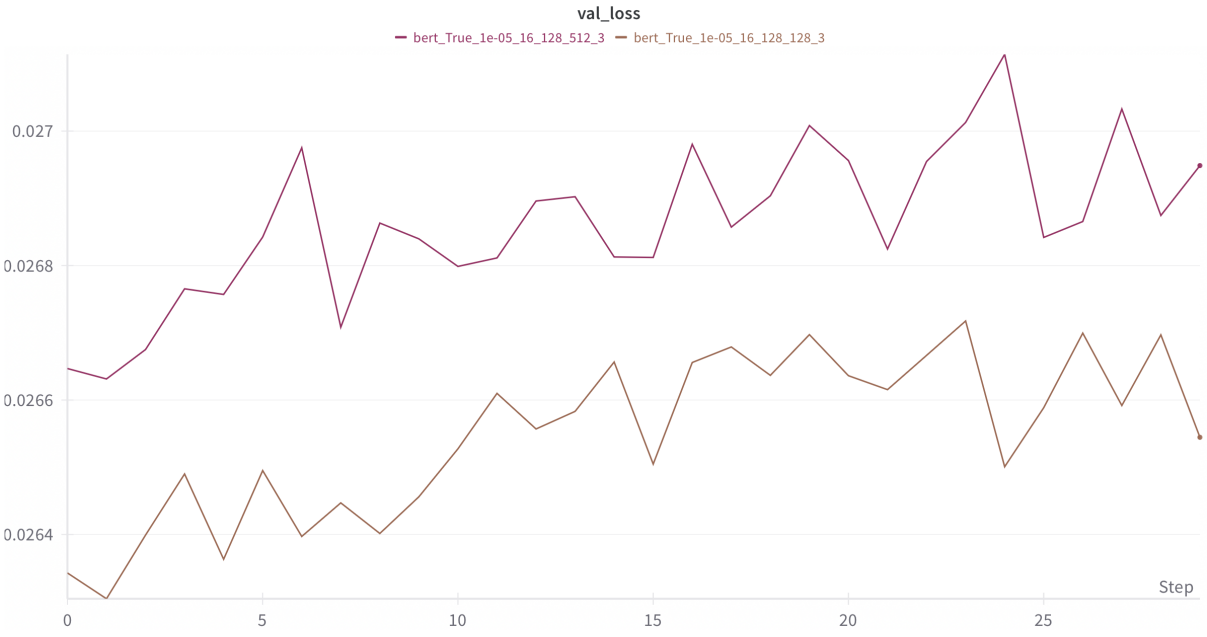


Cluster dimension

- We have varied the feature dimensions for each cluster with `64` and `128`
- We get better performance with `128` dimensions as this will help better represent the sentence, as it results in a bigger global descriptor, but at the same time not making it too big, when there is a choice for lesser number of clusters

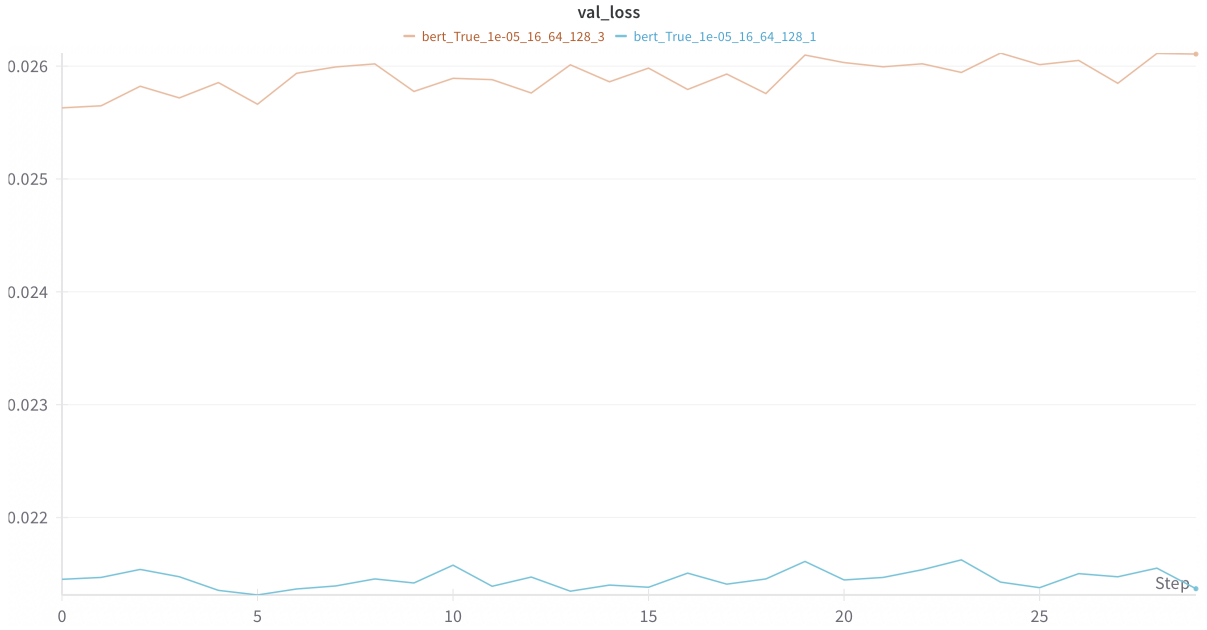
Token dimension

- This is the dimension of the projected CLS/EOT token of the models
- We have tested with `128` and `512`, which is the original size of the CLS/EOT token for all of these models, which is obtained after passing through linear layer projections
- This shows that, coupled with hidden state embeddings, not all of the information contained by the CLS/EOT tokens is actually informative, and we can get away with smaller feature vector contribution from these tokens



Number of trainable blocks

- This refers to the number of layers we unfreeze while training both the baselines and also when we add the aggregator
- The possible values for this parameter were `1` and `3`
- We can observe that there is better performance when lesser number of layers are trained, because there are more number of parameters to be trained when more layers are unfreezed, and the SemEval dataset is rather small.
    - Sadly, we could not perform more extensive hyperparameter tuning on a larger dataset due to compute limitations



Learning rate

- We chose the values of `1e-4` and `1e-5` for tuning
- Empirically, we found out `1e-4` to give better performace

Losses

- We have tried using the binary cross entropy loss for our training pipeline but faced unstable initialization in the beginning. We believe that the process was unstable due to the need for logarithmic computation in the Sinkhorn formulation
  - This logarithmic implementation of Sinkhorn is needed for its numerical stability. But, this is not suitable for the binary cross entropy loss formulation as it leads to nan values after some iterations
  - The training procedure is extremely unpredictable, and runs with the same configurations fail for some runs and finish successfully for some other runs, as can be observed below
  - So, we shifted to the cosine similarity loss formulation for more stable training

Overall analysis of the aggregator

- As the BERT model was tuned to find the best parameters on the SemEval24 dataset, we observe the expected trend that the BERT model with aggregator performs better than plain baseline finetuning.

- Note that the hyperparameters were chosen specifically based on the best performing BERT model, which does not necessarily mean that these transfer in a similar manner to other backbones like RoBERTa and CLIP as they have different architectures and underwent different training regimes.
    - Although the positive trend continued for the QQP dataset, there was again no significant improvement observed for the MTEB dataset, for which we hypothesize the difference in distributions across the datasets to be the reason
    - So, major improvements were not observed for these models with these hyperparameters
    - But, we can conclude from our hyperparameter study that given enough compute, we could find the optimal hyperparameters for each model-dataset configuration

---

Conclusion

We firmly believe that there is potential in exploring this approach, and given enough compute, we are confident that this formulation of globally aggregated descriptors can be viable to create informative descriptors that can be used as global embeddings for robust language understanding rather than using simple CLS/EOT tokens that might not be relevant for downstream tasks

---