Increasing Negative Examples for ITC

Our current approach utilizes Distributed Data Parallel (DDP) [1] with a batch size of 256 per GPU. With two GPUs, the combined batch size is 512, and image/text features are gathered from all devices to increase the number of negative examples, as described in VLMo [2]. This method, as demonstrated in experiments with the supervised teacher (SHRe), improves performance.

However, implementations of image-text models that leverage contrastive learning typically use much larger batch sizes, and therefore have much more negative examples available. For instance, CLIP uses a batch size of 32,768 [3]. Achieving such large batch sizes would require more GPUs, as batch sizes exceeding 256 (per device) lead to out-of-memory (OOM) errors on the GPU (NVIDIA RTX 4090).

Although adding more GPUs to our setup is costly, the training time will be reduced proportionally to the number of GPUs used. For example, using two GPUs instead of one halves the training time. This is because DDP shardes the whole training dataset between all devices, such that each device processes a different part of the dataset [1]. From a cost perspective, this is acceptable, however, we decide against this approach, because we want to keep the number of GPUs manageable. Moreover, going with more GPUs would make the success of our approach dependent on the number of GPUs one has available, which is why we search for alternatives that allow us to increase the number of negative samples without requiring more GPUs.

One intuitive approach to increasing the number of negative examples is to use a FIFO queue-based memory bank. The queue stores image and text embeddings from previous batches. With each new batch, the memory bank is updated by adding the current batch embeddings and discarding the oldest (batch). These stored embeddings, combined with the embeddings of the current batch, are then used as negative examples.

Given a batch size of N and a memory bank size of M, we can achieve $N - 1 + M$ negative samples. For instance, with a batch size of 256 and a memory bank size of 768, we can attain 1023 negative samples. This configuration effectively simulates a contrastive loss with a batch size of 1024, and is comparable to four GPUs with DDP and a batch size of 256 per GPU. However, it is important to note that this "simulation" of larger batch sizes with a memory bank only applies to the number of negative examples. The actual gradients are still computed using the effective batch size of 512 (256 for two devices/GPUs).

For an implementation the training setup remains the same, we are merely required to maintain four distinct memory banks. This is because we apply the contrastive loss to both the intermediate and output cls token `[T_CLS]`/`[I_CLS]` of the shared Transformer block, as illustrated in (TODO: cite model figure), and we need one memory bank for negative image and text examples, respectively.

No.	Modality	Layer	Stores negatives for:
1	Image	FFN 1	`[T_CLS]` of FFN 1
2	Text	FFN 1	`[I_CLS]` of FFN 1
3	Image	FFN 2	`[T_CLS]` of FFN 2
4	Text	FFN 2	`[I_CLS]` of FFN 2

Table 1: Each memory bank stores the representations of one modality, created by a layer, to use as negative examples for the represenations of the other modality, created by the same layer.

Illustrated in Table 2, we observed a significant drop in performance. This suggests that simply increasing the number of negative examples via a memory bank does not help in learning richer represenations.

Model	ImageNet-1K Accuracy	Retrieval MSCOCO	Retrieval Flickr30K	Batch Size	ITC # Negative Examples
CLIP [3]	**72.6**	**66.73**	**90.1**	32,768	32,767
$\text{EMKUM}_{\text{DDP}}$	26.1	66.3	42.4	512	511
EMKUM_{MB}	17.8	54.4	30.3	256	511

Table 2: CLIP

We suspect this drop in performance originates because of the following reasons: When using an actual batch size of 512, so the same approach we used before ($\text{EMKUM}_{\text{DDP}}$ in Table 2), all negative examples come from the same model with the same weights, as they are all generated during the same step. Therefore, the representations share the same latent space and are consistent with each other. A similarity measure, in our case the cosine similarity, can then provide the model with a meaning of distance between the representations, in our case an image and text.

However, when using a memory bank, the negative examples come from previous batches that were stored in the memory bank. At previous batches, or steps respectively, the model's weights were different, and therefore the information encoded in the representations. As the model's weights constantly change, especially at the beginning of training, there is a continous shift in the representation space. This shift is so pronounced that even representations from the immediate previous steps differ significantly from the current representations, and a similarity measure will not provide meaningful information to the model.

This can be thought of as a less extreme case of comparing the representations of an image-only and text-only model, which are not associated with each other. In the beginning of our experiments, we tested image-text retrieval with the Data2Vec2 image and text model, and observed that this approach is ineffective for image-text retrieval. The representations produced by both models do not have any relationship with each other, and therefore the cosine similarity does not provide any meaningful information to the model.

With a memory bank, this effect is less pronounced, as the representations are still generated by the same model, but the shift in the model's weights is still significant enough to make the representations inconsistent with each other.

Relation to Initial Memory Bank Approach

A memory bank was initially introduced by [4] as a mapping of the complete training dataset. The embedding of each sample in the dataset is stored in the memory bank (illustrated in Figure 1). For each batch, K samples are randomly drawn from the memory bank to be used as negative examples. The representations of samples that are currently in the batch are then updated in the memory bank [4]. This approach is similar to ours, but faces the same problem: The representations come from an older variant of the model with different weights. Even worse, the representation of an example in the dataset is updated when it was last seen in a batch, which can be a long time ago for large datasets. However, the authors mitigate this issue by using proximal regularization, as shown in Equation 1.

$$-\log h(i, \boldsymbol{v}_i^{t-1}) + \lambda * \|\boldsymbol{v}_i^t - \boldsymbol{v}_i^{t-1}\|_2^2 \tag{1}$$

While the term $-\log h(i, \boldsymbol{v}_i^{t-1})$ can be ignored for now, as it just denotes a form of contrastive loss, the other term $\lambda * \|\boldsymbol{v}_i^t - \boldsymbol{v}_i^{t-1}\|_2^2$ serves as the proximal regularization term. It describes the mean squared error between the representation $\boldsymbol{v}_i^t$ of a training example i, e.g. an image, which is part of the current batch, and the representation of the same training example $\boldsymbol{v}_i^{t-1}$ stored in the memory bank and updated when it was last seen, denoted as time step $t-1$.

This term enforces that the representation of a training example does not change too rapidly between updates, and allows for more stable negative examples, depending on the value of weight λ. The authors report improved results with a value of $\lambda = 30$ [4], meaning that the proximal regularization term is 30 times more important than the contrastive loss term.

This forces the model to keep the representations of the training examples in the memory bank consistent with the current batch, so that a similarity measure can provide meaningful learning signals to the model. Our approach does not take this into account.

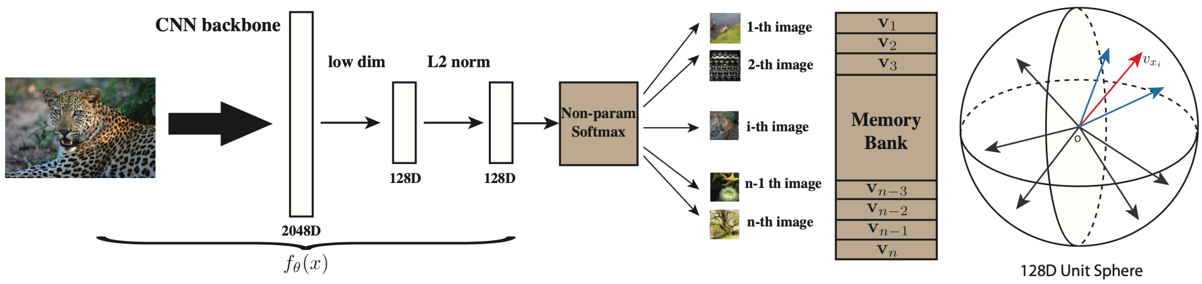


Figure 1:

Momentum Encoder

- problem also identified by authors of MoCo [5]
- authors experiments, displayed in Figure 2, show that even a memory bank that maps all training examples performs significantly worse than using the actual batch size it tries to mimic
- especially with lower batch sizes
- this type of memory bank need to sample up to 65,536 ($K = 65, 536$) negative examples to match actual, comparatively small, batch size of 1024

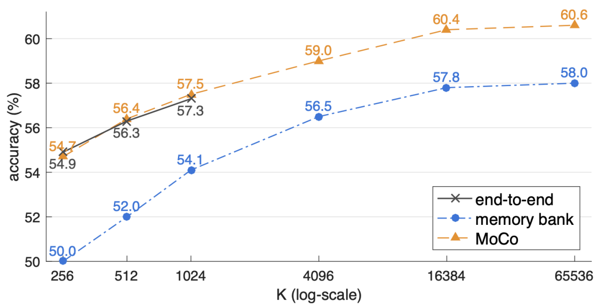


Figure 2:

- we can't use the memory bank style of [4], since we have 3,264,868 training examples
  - each embedding has a size of 768, considering storing them at full precision, so float32, we would need $3, 264, 868 * 768 * 4B > 10e^9 B = 10$ GB of additional GPU memory
- we suspect that using a proximal regularization term, as in Equation 1, might help to stabilize our memory bank approach
- however, we cannot use it -> the term is based on the representation of the same training example at the previous time step
- but our memory bank is significantly smaller than the training dataset, and older samples are dequeued, so the same training example will never be in the memory bank and at the same time in the current batch
- authors of MOCO propose a different approach to solving this problem

- they use a queue based memory bank, much smaller than the training dataset (cite MoCo v2), so same approach as we use
- but the negative examples in the memory bank are not updated by the model that is trained, but instead by a momentum encoder, which is a moving average of the actual model weights
- weight update of the momentum encoder is given by

$$\theta_k = m * \theta_k + (1 - m) * \theta_q \tag{2}$$

- with θ_k being the momentum encoder weights, θ_q the actual model weights, and m the momentum factor
- set to to $m = 0.999$, meaning the momentum encoder is updated very slowly
- that way negative examples in the memory bank are updated by a model with similar weights as the model that is trained
- makes them more stable and consistent
- can be seen as keeping the model consistent that produces the negative examples, instead of making the negative examples themselves consistent through an additional regularization term (Equation 1)
- disadvantage is that a copy of the same model that is being trained has to be kept in memory
- even though no gradients are needed for the momentum encoder, it still requires additional GPU memory

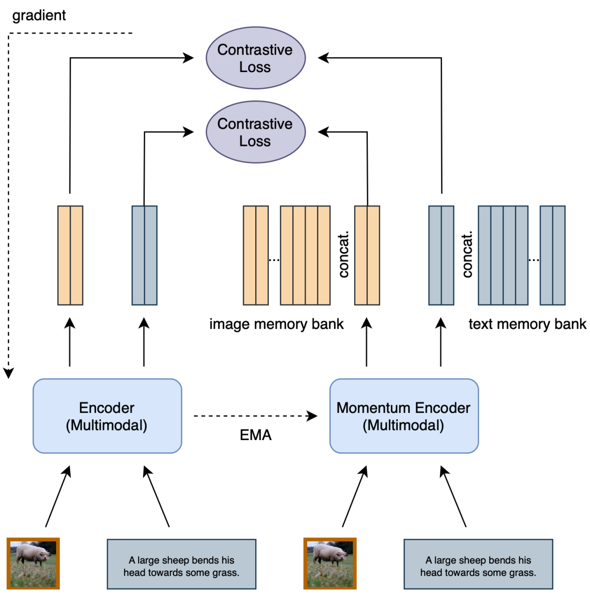


Figure 3: Figure inspired by MoCo v2 [6]

- in conclusion:

1. can't use FIFO queue based memory bank, as representations of negative examples are inconsistent
2. can't use memory bank that maps all training examples, as it would require too much additional GPU memory
3. proximal regularization not applicable to a memory bank that is smaller than the training dataset

- only alternative is to use a momentum encoder as in MoCo

- therefore, we opt for this approach
- experimental setup remains the same, but we add a momentum encoder, which is a copy of our (student) model that is trained
- oriented on ALBEF [7], we use a memory bank of size 65,536 and a momentum factor of 0.995
  - both are hyperparameters that also lead to good results in MoCo, where this approach was first introduced [5]
- however, we get an OOM error
- this is not surprising, as we have a very large memory bank of size 65,536
- as seen in Figure 3, we can't just have one memory bank, as done in unimodal models like MoCo [5], because we need one memory bank for storing negative text examples and one for storing negative image examples
- also recall that we do contrastive loss on both FFN layers of the shared Transformer layer
- so we need two memory banks for each FFN layer
- so we need four memory banks of size 65,536 each
- we would like to keep the size of the memory bank, as this is important for performance (seen in Figure 2)
- therefore, we perform two optimizations:
  - we only do the contrastive loss on the last FFN layer of the shared Transformer layer, so on its cls token output
  - will reduce the GPU memory by a margin, as we only need two memory banks of size 65,536 each
  - also, while the memory bank of the last FFN layer stores embeddings of size 768, because this is the hidden dim used for our model, the memory bank of the first FFN layer stores embeddings of size 3072
    - this is because in Transformer layers the first FFN layer of the MLP expands the hidden dim by a factor of four ($4 * 768 = 3072$)
  - just removing this memory bank will save us 1.6 GB of GPU memory
  - reason: assuming embeddings are in full precision, so float32, we would need $3072 * 4B = 12,288B$, so approximately 12.3 KB per embedding
  - for one memory bank, this means $12,288 * 65,536 = 805,8 \text{ MB}$ of GPU memory is required
  - for two memory banks, this means $805,8 \text{ MB} * 2 \approx 1.6 \text{ GB}$ of GPU memory is required
- this is still not enought to prevent OOM errors, and we identify a further optimization
- usually, forward pass of momentum encoder is done after the forward pass of the model that is trained
- this approach is used in MoCo [5] and ALBEF [7]
- that means that during the forward pass of the momentum encoder, where GPU memory is required to store the results, all activations of the actual model are also stored on the GPU, as they are needed for the backward pass later
- because we did not encoder OOM errors before using the momentum encoder, and we observed that the GPU memory in previous experiments was almost fully utilized, even without a momentum encoder, we suspect that the forward pass of the momentum encoder is the reason

- we therefore perform the forward pass of the momentum encoder before any work is done in one step of the training loop
- so momentum update and forward pass of the momentum encoder is done first
- this way, activations of momentum encoder are freed before the forward pass of the model that is trained
- performance stays the same, as the work that is done remains the same
- with this strategy, we avoid OOM errors
- time per step remains the same, as the work that is done remains the same

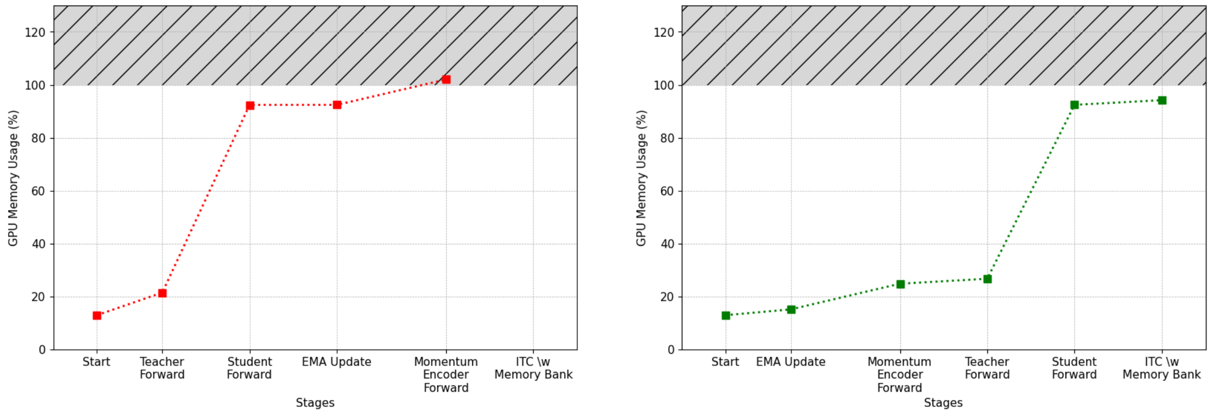
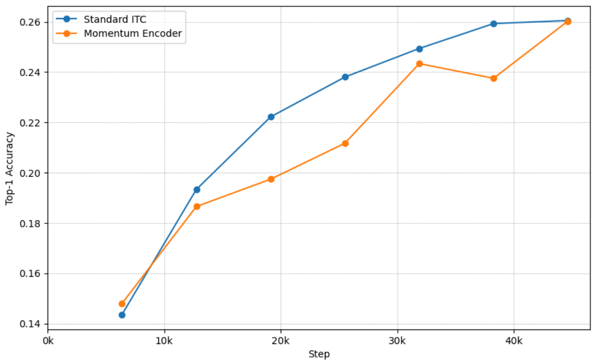


Figure 4: Doing the forward pass of the momentum encoder after the forward pass of the model, when the model's activations are already stored on the GPU memory (left), leads to a cuda OOM error. This can be avoided by reorderind the operations, so that the momentum encoder (EMA) update and forward pass are performed before the forward pass of the model (right). Results are based on NVIDIA RTX 4090 with 24 GB of GPU memory.

- compared to normal DDP, the memory bank approach with momentum encoder adds on average 5 minutes per epoch, so the overhead is marginal
- in Figure 5 we show the results of this approach
- we see that the performance does not exceed that of the the standard gathering from all devices with just 511 negative examples (effective batch size of 512)
- while we can observe that the zero-shot accuracy on ImageNet-1K exceeds that of the vanilla ITC approach towards the end of training, we consider such small improvement as worthwile, considering the amount of additional complexity required to achieve it
- even though the memory bank approach looks more promising to achieve a higher accuracy than the previous approach, as the latter appears to saturate towards the end of training, but this would require longer training times
- with a batch size of 256, the model is already trained for almost a combined 90k steps, and we consider longer training times as both out of scope for this work, and too long to consider our approach as efficient
  - (the actual step size in training is around 45k (Figure 5 left), as we divide the whole dataset between both GPUs, meaning the batch size is doubled to 512)
- another reason is that we are going to introduce a new definition of similarity between image and text, which will be infeasible in combination with a memory bank



Model	ImageNet-1K Accuracy	Retrieval MSCOCO	Retrieval Flickr30K
Standard ITC	26.1	66.3	42.4
Momentum Encoder	26.0	256	30.3

Figure 5: Comparison of the Standard ITC approach with momentum encoder and a memory bank of size 65,536. The momentum encoder approach does not exceed the performance of the standard ITC approach (right), even though it shows a promising trend towards the end of training (left).

…

- best approach would be to use no contrastive loss -> all the work we did before was because contrastive learning requires large number of negative examples
- BEiT-3 give good reason to discard it

Bibliography

[1] S. Li *et al.*, “PyTorch distributed: experiences on accelerating data parallel training,” *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3005–3018, Aug. 2020, doi: 10.14778/3415478.3415530.

[2] H. Bao *et al.*, “VLMo: Unified Vision-Language Pre-Training with Mixture-of-Modality-Experts,” in *Advances in Neural Information Processing Systems*, 2022. [Online]. Available: https://openreview.net/forum?id=bydKs84JEyw

[3] A. Radford *et al.*, “Learning transferable visual models from natural language supervision,” in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, M. Meila and T. Zhang, Eds., in Proceedings of Machine Learning Research, vol. 139. PMLR, 2021, pp. 8748–8763.

[4] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin, “Unsupervised Feature Learning via Non-parametric Instance Discrimination,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 3733–3742. doi: 10.1109/CVPR.2018.00393.

[5] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, “Momentum Contrast for Unsupervised Visual Representation Learning,” *arXiv preprint arXiv:1911.05722*, 2019.

[6] X. Chen, H. Fan, R. Girshick, and K. He, “Improved Baselines with Momentum Contrastive Learning,” *arXiv preprint arXiv:2003.04297*, 2020.

[7] J. Li, R. R. Selvaraju, A. D. Gotmare, S. Joty, C. Xiong, and S. Hoi, “Align before Fuse: Vision and Language Representation Learning with Momentum Distillation,” in *NeurIPS*, 2021.