Appendices

A. Query Types Comparison

Type of Queries	mAP [%]
Learnable	88.1
Fixed NLP-based	88.1
Fixed Random	88.1

Table 7. **Comparison of MS-COCO mAP scores for ML-Decoder with different query types**. We used $K = N = 80$.

B. ZSL Group Decoder

We will now describe how to incorporate group-decoding in the ZSL setting in order to improve the ML-Decoder's scalability with respect to the number of classes. Group-decoding cannot be trivially applied for ZSL, since in group-decoding each query is associated with a group of labels, while in the ZSL setting we assume that each query is associated with a specific word embedding. To alleviate this issue, we propose to construct each query by concatenating linear projections of all the word embeddings assigned to its group (See Figure 7). Formally, the k^{th} group query is given by

$$q_k = concat(\{W_a \cdot N_i\}_{i \in \mathcal{G}_j}) \quad (5)$$

where $\mathcal{G}_k$ is the set of labels assigned to the k^{th} group, $\{N_i\}$ is the set of word embeddings ($N_i \in \mathbb{R}^{d_w}$), $W_a \in \mathbb{R}^{\frac{d}{g} \times d_w}$ is the parameter matrix, and $g = \frac{N}{K}$.
In addition, as can be seen in Table 8, the group fully-connected head, even with shared weights, does not generalize well to unseen classes in a group-decoding setting. Therefore, we implemented a different pooling strategy for ZSL group-decoding: we decompose the group fully-connected parameter matrix (W_k from Eq. 3) into two components: $\{N_i\}$ - the set of word embeddings which is label-specific, and $W_b \in \mathbb{R}^{d \times d_w}$ - a learned parameter matrix which is shared for all labels. Formally, the parameter matrix W_k of the k^{th} group-query (Eq. 3) is constructed by the following:

$$W_k = W_b \cdot M_k \quad (6)$$

where $M_k \in \mathbb{R}^{d_w \times g}$ is constructed by stacking the word embedding vectors $\{N_i\}_{i \in \mathcal{G}_k}$ of group k. Inserting Eq. 6 into Eq. 3, we get the output logits for the ZSL group-decoder. (see Fig. 7, and pseudo-code in appendix L). Table 8 shows ablation experiments for the different modifications.

C. MS-COCO Training Details

Unless stated explicitly otherwise, for MS-COCO we used the following training procedure: We trained our models for 40 epochs using Adam optimizer and 1-cycle policy, with maximal learning rate of 2e-4. For regularization, we used Cutout factor of 0.5, True-weight-decay of 1e-4 and auto-augment. We found that the common ImageNet statistics normalization does not improve results, and instead used a simpler normalization - scaling all the RGB channels to be between 0 and 1. Our input resolution was 448. For ML-Decoder, our baseline was full-decoding ($K = N = 80$). Similar to [2], we used Open Images pretraining for our models' backbone. ML-Decoder weights were used with random initialization. The number of token embeddings was $D = 768$. We adjusted the backbone embedding output to D via a 1×1 depth-wise convolution. As a loss function, we used ASL with $\gamma_- = 4$, $\gamma_+ = 0$ and $m = 0.05$. All results were averaged over three seeds for better consistency. TResNet-L model is V2[1].

Query Type	Head Type	mAP [%] (ZSL)
NLP	Learnable	2.4
NLP	Learnable (shared)	2.6
Learnable	NLP	23.4
NLP	NLP	28.7

Table 8. **Comparison of NUS-WIDE mAP scores for ML-Decoder with different query types and head types**. "NLP" signifies using NLP projections to generate query/head parameters, and "Learnable" signifies using regular parameters (as described in Section 2.3.2)

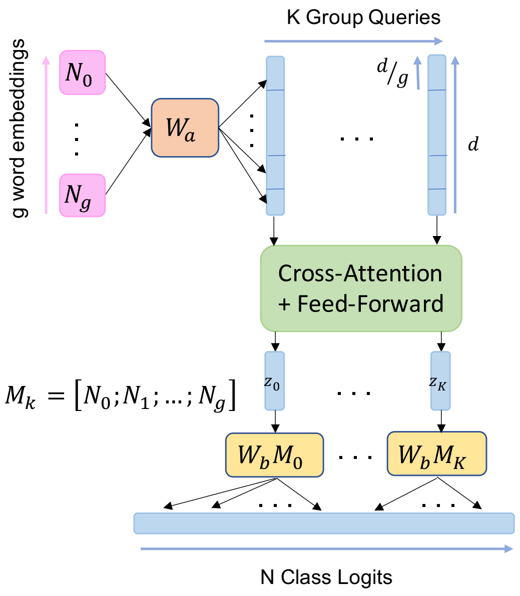

Figure 7. **Group decoding scheme for ZSL.**

[1] see: https://github.com/Alibaba-MIIL/TResNet

Classification Head	Number of Classes	Number of Queries	Training Speed [img/sec]	Inference Speed [img/sec]	Maximal Training Batch Size	Flops [G]
GAP	100	—	706	2915	520	5.7
	1000	—	703	2910	512	5.7
	5000	—	698	2846	504	5.8
Transformer-Decode	100	100	556	2496	424	6.6
	1000	1000	44	916	112	14.2
	5000	5000	2	17	4	61.1
ML-Decoder	100	100	575	2588	464	6.3
	1000	100	568	2563	456	6.3
	5000	100	562	2512	448	6.4

Table 9. **Comparison of throughput indices for different classification heads.** All measurements were done on Nvidia V100 16GB machine, with mixed precision. We used TResNet-M as a backbone, with input resolution 224. Training and inference speed were measured with 80% of maximal batch size.

D. Speed Comparison

In Table 10 we provide flop measurements for ML-Decoder with stacked layers For these measurements wee used a decoder with self-attention, since for additional stacked layer, the attention module is not redundant. We see from Table 10 that the contribution of stacking several layers is negligible. Due to these results, our ML-Decoder based-solution uses a single decoding layer.

Head Type	Decoder Layers	mAP [%]	Flops [G]
GAP		87.02	23.0
ML-Decoder	1	88.09	24.1
	2	88.16	25.0
	3	88.21	25.9

Table 10. Flops vs. accuracy measurements for ML-Decoder with stacked layers.

In Table 11 we present direct direct inference time measurements of different modules of a decoder units.

Num Classes	Module	ML-Decoder cost [ms]	Transformer-Decoder cost [ms]
100	self-attention	None	1.5
	cross-attention	1.4	1.4
	mlp	1.8	1.8
4000	self-attention	None	612.8
	cross-attention	1.4	38.8
	mlp	1.8	85.3

Table 11. Comparing inference time of different modules.

In Table 12 we provide full speed-accuracy measurements for different classification heads. We see that using ML-Decoder on TResNet-M architecture reduces inference speed by 15%, independent with the number of classes, while transformer-decoder classification head scales badly with the number of classes, reducing the inference speed (and other throughput metrices) by orders of magnitudes.

E. MS-COCO Results for Different Input Resolutions

In Table 13 we provide results on MS-COCO dataset for different input resolution - $224, 448$ and 640.

F. NUS-WIDE ZSL Dataset and Training Details

NUS-WIDE is a multi-label ZSL dataset, comprised of nearly $270K$ images with 81 human-annotated categories, in addition to the 925 labels obtained from Flicker user tags. The 925 and 81 labels are used as seen and unseen classes, respectively.

To be compatible with previous works [3], as a loss function we used CE, our backbone was TResNet-M, and our input resolution is 224. Unless stated otherwise, our baseline ML-Decoder for ZSL was full-decoding, with $K = N$, and shared projection matrix, as discussed in Section 2.3.3. Other training details for ZSL NUS-WIDE were similar to the one used for MS-COCO.

G. Pascal-VOC Training Details and Results

Pascal Visual Object Classes Challenge (VOC 2007) is another popular dataset for multi-label recognition. It contains images from 20 object categories, with an average of 2.5 categories per image. Pascal-VOC is divided to a trainval set of 5,011 images and a test set of 4,952 images. As a backbone we used TResNet-L, with input resolution of 448. For ML-Decoder, our baseline was full-decoding ($K = N = 20$). Other training details are similar to the ones used for MS-COCO. Results appear in Table 14.

H. Open-Images Training Details and Results

Open Images (v6) [19] is a large-scale dataset, which consists of 9 million training images, $41,620$ validation

Classification Head	Number of Classes	Number of Queries	Training Speed [img/sec]	Inference Speed [img/sec]	Maximal Training Batch Size	Flops [G]
GAP	100	—	706	2915	520	5.7
	1000	—	703	2910	512	5.7
	5000	—	698	2846	504	5.8
Transformer-Decode	100	100	556	2496	424	6.6
	1000	1000	44	916	112	14.2
	5000	5000	2	17	4	61.1
ML-Decoder	100	100	575	2588	464	6.3
	1000	100	568	2563	456	6.3
	5000	100	562	2512	448	6.4

Table 12. **Comparison of throughput indices for different classification heads.** All measurements were done on Nvidia V100 16GB machine, with mixed precision. We used TResNet-M as a backbone, with input resolution 224. Training and inference speed were measured with 80% of maximal batch size.

Method	Backbone	Input Resolution	Flops [G]	mAP [%]
ML-Decoder	TResNet-L	224x224	9.3	85.5
ML-Decoder	TResNet-L	448x448	36.2	90.0
ML-Decoder	TResNet-L	640x640	73.5	**91.1**

Table 13. **Comparison of MS-COCO mAP scores for different input resolutions**.

Method	mAP [%]
CE [2]	84.8
Focal Loss [2]	84.9
ASL [2]	86.3
ML-Decoder	86.8

Table 15. **Comparison of ML-Decoder to known state-of-the-art results on Open Images dataset.**

Method	mAP [%]
RNN [37]	91.9
FeV+LV [42]	92.0
ML-GCN [8]	94.0
SSGRL [6]	95.0
BMML [21]	95.0
ASL [2]	95.8
Q2L [23]	96.1
ML-Decoder	96.6

Table 14. **Comparison of ML-Decoder to known state-of-the-art models on Pascal-VOC dataset.** For ML-Decoder, we used TResNet-L backbone, input resolution 448.

images and $125,436$ test images. It is partially annotated with human labels and machine-generated labels. For dealing with the partial labeling methodology of Open Images dataset, we set all untagged labels as negative, with reduced weights. Due to the large the number of images, we trained our network for 25 epochs on input resolution of 224. We used TResNet-M as a backbone. Since the level of positive-negative imbalancing is significantly higher than MS-COCO, we increased the level of loss asymmetry: For ASL, we trained with $\gamma_- = 7, \gamma_+ = 0$. For ML-Decoder, our baseline was group-decoding with $K = 100$. Other training details are similar to the ones used for MS-COCO.

I. Single-label Classification with Different Logit Activations

In Table 16 we compare ImageNet classification scores for different logits activations.

Logit Activation	Classification Head	Top1 Acc. [%]
Sigmoid	GAP	79.7
	ML-Decoder	80.3
Softmax	GAP	79.3
	ML-Decoder	80.1

Table 16. **ImageNet classification scores for different classification heads and logits activations**. For ML-Decoder, we used group-decoding with 100 groups. Our training configuration is A2 [39]. Backbone - ResNet50.

I.1. Comparison of ML-Decoder to State-of-the-art Models on Single-label Transfer Learning Datasets

To further test our solution, we compare ML-Decoder based models to known state-of-the-art results from the literature on two prominent and competitive single-label datasets - CIFAR-100[**?**] and Stanford-Cars[**?**]. The comparison is based on [2] and [3].

[2]https://paperswithcode.com/sota/image-classification-on-cifar-100
[3]https://paperswithcode.com/sota/fine-grained-image-classification-on-stanford

Dataset	Model	Top-1 Acc.
CIFAR-100	CvT-W24	94.05
	ViT-H	94.55
	EffNet-L2 (SAM)	**96.08**
	Swin-L + ML-Decoder	95.1
Stanford-Cars	EffNet-L2 (SAM)	95.95
	ALIGN	96.13
	DAT	96.2
	TResNet-L + ML-Deocder	**96.41**

Table 17. **Comparison top of state-of-the-art models**.

We can see from Table 17 that with ML-Decoder, we achieve the 1st and 2nd place on Stanford-Cars and CIFAR-100 datasets respectively.

J. Query Augmentations Illustration

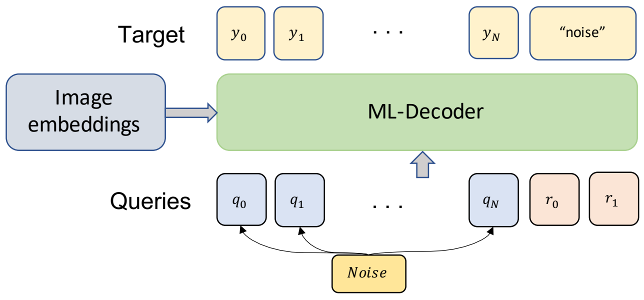

Figure 8. **Query augmentations** *rand-query*: adding random queries which are assigned label "noise". *additive noise*: adding random noise to the input queries.

K. Group Fully-connected Pseudo Code

```
def GroupFullyConnected(G, group_weights, output, num_of_groups):
    '''
    - G is the group queries tensor. G.shape = [groups, embeddings]
    - group_weights are learnable (group) fully connected weights.
      group_weights.shape = [groups, embeddings, classes//groups]
    - output is the interpolated queries tensor. output.shape = [groups, classes//groups]
    '''
    for i in range(num_of_groups):
        g_i = G[i, :] # [1,embeddings]
        w_i = group_weights[i, :, :] # [embeddings, classes//groups]
        output_i = matmul(g_i, w_i) # [1, classes//groups]
        output[i, :] = output_i
    logits = output.flatten(1)[:self.num_classes] # [1, classes]
    return logits
```

Notice that the loop implementation is very efficient in terms of memory consumption during training. Implementing the group-fully-connected in a single vectoric operation (without a loop) is possible, but reduces the possible batch size. Also, the proposed implementation is fully suitable for compile-time acceleration (@torch.jit.script)

L. Group Fully-connected ZSL Pseudo-Code

```
def GroupFullConnectedZSL(G, wordvecs, output, W, num_groups, num_classes):
    '''
    - G is the group queries tensor. G.shape = [num_groups, embeddings_dim]
    - wordvecs is the word-embedding tensor [num_classes, word_embedding_dim]
    - W is a learnable projection matrix [embeddings_dim, word_embedding_dim]
    - output is the interpolated queries tensor. output.shape = [num_groups, num_classes//num_groups]
    '''
    labels_per_group = num_classes // num_groups
    group_weights = zeros(num_groups, embedding_dim, labels_per_group)
    for i in range(num_classes):
        group_weights[i // labels_per_group, :, i % labels_per_group] = W * wordvecs[i, :]

    logits = GroupFullyConnected(G, group_weights, output, num_groups)
    return logits
```