GNR 650 (Autumn 2024): Assignment 2

Zero Shot Learning for Image Classification on AwA2 Dataset

Soumen Mondal (23m2157)
Siddhant Gole (23m2154)

November 1, 2024

1 Introduction

This assignment focuses on implementing and analyzing zero-shot learning (ZSL) method, specifically following the methods proposed in the Class Normalization (CN) paper [5]. Zero-shot learning is a challenging problem in machine learning, where models are required to generalize to classes that are not present in the training data. This problem is particularly important for applications requiring the model to handle unseen categories, as collecting labeled data for all possible classes can be impractical or impossible. While normalization techniques have proven essential for stable and effective training in supervised learning, they are less explored in ZSL settings, where they can significantly impact model performance. The implementation was tested on AwA2 dataset [7]. Notably, this approach using CN achieved competitive results and faster training speeds.

2 Methodology

This section details the methodology used to implement and train a zero-shot learning (ZSL) model incorporating normalization techniques, particularly Normalize+Scale (NS), Attribute Normalization (AN), and the proposed Class Normalization (CN). We describe the input-output setup, the structure of the model, and how normalization methods are applied, followed by an explanation of the loss function and hyperparameters involved.

2.1 Notation

In a ZSL setup, we have access to labeled datasets of seen and unseen classes:

- $D^s = \{\mathbf{x}_i^s, y_i^s\}_{i=1}^{N_s}$: the labeled dataset of seen classes, where $\mathbf{x}_i^s$ represents an image and y_i^s is its label.
- $D^u = \{\mathbf{x}_i^u, y_i^u\}_{i=1}^{N_u}$: the unlabeled dataset of unseen classes, used during testing.

The original paper used the attribute vector $a_c \in \mathbb{R}^{d_a}$ given in the AwA2 dataset but in this assignment, we do not use the attribute vectors. Instead, each class c is described by its class name m_c and the class names are converted into the attributes $\mathbf{v}_c$ using `word2vec` model [4] with classes divided into:

- $V^s = \{\mathbf{v}_i\}_{i=1}^{K_s}$: class word vectors for seen classes.
- $V^u = \{\mathbf{v}_i\}_{i=1}^{K_u}$: class word vectors for unseen classes.

where N_s and N_u are the number of samples for seen and unseen images, respectively, and K_s and K_u represent the number of seen and unseen classes.

2.2 Model Inputs and Outputs

The input to the model is an image $\mathbf{x}$, which is passed through a visual feature extractor $E : \mathbf{x} \rightarrow \mathbf{z} \in \mathbb{R}^{d_z}$ to obtain a feature representation $\mathbf{z}$ of dimensionality d_z. For each class c, a prototype $\mathbf{p}_c$ is generated by mapping its word vector $\mathbf{v}_c$ through an embedding function $\Phi_\theta : \mathbf{v}_c \rightarrow \mathbf{p}_c \in \mathbb{R}^{d_z}$, where θ denotes the parameters of the embedder. The embedding function projects class vectors $\mathbf{v}_c$ onto feature space $\mathbb{R}^{d_z}$ in such a way that it lies closer to exemplar features $\mathbf{z}_s$ of its class c. During training, the objective is to learn an embedding function Φ_θ that aligns the feature $\mathbf{z}$ with the corresponding class prototype $\mathbf{p}_{y_i}$ based on the label y_i for seen classes. During testing, the model assigns labels to unseen samples by finding the closest unseen class prototype in the embedding space.

- **Visual feature extractor** – Takes an *image* as input and outputs a feature representation $\mathbf{z}$; this component is pretrained. We have used ViT base [6] model as feature extractor from https://huggingface.co/google/vit-base-patch16-224-in21k. The dimension of visual feature vector is set as 768. As an ablation, we will also use pretrained ResNet50 model [2] to extract the visual features which is taken from https://huggingface.co/microsoft/resnet-50.
- **Word vector model** – Takes a *class name* as input and outputs a *class vector* $\mathbf{v}$; this component is pretrained. The `word2vec` model [4] operates at the word level, optimizing each word independently. An alternative variant, the `fasttext` model [1], optimizes at the subword level, which provides an advantage over `word2vec` by avoiding out-of-vocabulary (OOV) errors. Since `fasttext` breaks words into subwords, it can generate embeddings for words not present in the original vocabulary. Thus, `fasttext` can be considered an extension of `word2vec`, as it is trained on the same objective while offering improved coverage for rare or unknown words. We have used the pretrained `word2vec` model (pretrained on Google news corpus) and `fasttext` model (pretrained on Wikipedia corpus) from https://radimrehurek.com/gensim/models/word2vec.html. The dimension of word vector is set as 300.
- **Embedding function** – Takes a *class vector* as input and outputs a *prototype vector*; this component is trainable. We have used a simple feed forward neural network with 1 hidden layer with number of neurons of 1024 as the embedding function. The dimension of class prototypes is set same as the visual feature vector which is 768.

2.3 Normalization Techniques

2.3.1 Normalize+Scale (NS)

The Normalize+Scale (NS) technique modifies the logits calculation to use scaled cosine similarity instead of a standard dot product. Given a feature vector $\mathbf{z}$ and its corresponding output from the embedding function, a class prototype $\mathbf{p}_c$, the logit $\hat{y}_c$ for class c is computed as:

$$\hat{y}_c = \gamma^2 \cdot \frac{\mathbf{z}^\top \mathbf{p}_c}{\|\mathbf{z}\| \|\mathbf{p}_c\|}$$

where γ is a scaling hyperparameter, typically chosen from the interval $[5, 10]$. Scaling with γ^2 enhances the effect of the cosine similarity, making it more discriminative.

2.3.2 Attributes Normalization (AN)

Attributes Normalization (AN) applies L_2 normalization to each class word vector, ensuring they have unit norms:

$$\mathbf{v}_c \leftarrow \frac{\mathbf{v}_c}{\|\mathbf{v}_c\|}$$

This normalization is crucial for maintaining consistent variance across word vectors, facilitating better alignment with the feature space. Although simple, it has been shown to improve performance over traditional normalization methods.

2.3.3 Class Normalization (CN)

To address limitations of NS and AN in deeper architectures, we employ Class Normalization (CN). CN applies a combination of class-wise standardization and normalization to the prototypes $\mathbf{p}_c$ after they are mapped by Φ_θ. During training, the mean and variance of class features (prototypes) are computed for each batch and used to normalize the features. Additionally, a moving average of these statistics is maintained through running mean and running var, which accumulate across batches. Note that the normalization is performed using variance and not the standard deviation for the reasons explained in the paper [5].

During Training (Batch Statistics):

$$\text{result} = \frac{\text{class\_feats} - \text{batch\_mean}}{\text{batch\_var} + \epsilon}$$

During Inference (Accumulated Running Statistics):

$$\text{result} = \frac{\text{class\_feats} - \text{running\_mean}}{\text{running\_var} + \epsilon}$$

2.4 Training Procedure

The training process involves optimizing the parameters of the embedding function Φ_θ such that seen class features are close to their corresponding prototypes. A typical ZSL objective function is used:

$$\mathcal{L} = -\sum_{i=1}^{N_s} \log \frac{\exp(\gamma^2 \cdot \cos(\mathbf{z}_i, \mathbf{p}_{y_i}))}{\sum_{c=1}^{K_s} \exp(\gamma^2 \cdot \cos(\mathbf{z}_i, \mathbf{p}_c))}$$

where $\cos(\mathbf{z}_i, \mathbf{p}_c)$ denotes the cosine similarity between the normalized feature vector $\mathbf{z}_i$ and normalized class prototype $\mathbf{p}_c$, and γ is the scaling factor. This loss function promotes high similarity between features and corresponding prototypes, ensuring correct classification.

2.5 Hyperparameters

The main hyperparameters for our model are:

- **Scaling factor** γ: Controls the influence of cosine similarity in logits computation. A value of 5 is used throughout the experiment.
- **Optimizer**: We have used Adam optimizer [3] to train the model with learning rate of 0.0005 and weight decay of 0.0001. We have used a batch size of 256 and trained the model for 5 number of epochs.

3 Results

3.1 Experiments

The training process involves splitting the dataset classes into seen and unseen sets, following a zero-shot learning setup. We divide the classes alphabetically in a 50:50 ratio: the first 25 classes are designated as seen classes for training, while the remaining 25 classes serve as unseen classes for testing. This approach ensures a clear separation between classes that the model encounters during training and those it must generalize to without direct exposure. The training is conducted across four configurations, combining two feature extractors and two word embedding models:

- **Configuration 1: ResNet-50 + `word2vec`**
  In this configuration, ResNet-50 is used as the feature extractor to obtain image features, while `word2vec` generates the semantic embeddings for each class name. The ResNet-50 model is pretrained and frozen, and the pretrained `word2vec` embeddings provide class vectors for alignment with the visual features.

- **Configuration 2: ResNet-50 + `fasttext`**
  Similar to the first configuration, ResNet-50 is used as the visual feature extractor. However, `fasttext` is used to generate semantic embeddings. Since `fasttext` operates at the subword level, it provides representations even for out-of-vocabulary words, enhancing robustness for classes that may not be directly present in the vocabulary. This is crucial as the class name which has a plus symbol (+) are indeed not present in `word2vec` model. In that case, we have split the word into two subwords separated by + and then took the mean vectors of each of the subwords.

- **Configuration 3: ViT Base + `word2vec`**
  In this configuration, a Vision Transformer (ViT Base) model is employed as the feature extractor. The ViT model, pretrained on a large-scale dataset, extracts high-level features from images, which are aligned with `word2vec`-generated class vectors. ViT's ability to capture spatial dependencies provides an alternative feature representation to the CNN-based ResNet-50.

- **Configuration 4: ViT Base + `fasttext`**
  Here, ViT Base is paired with `fasttext` for generating class embeddings. The subword-level embeddings from `fasttext` allow this configuration to handle any out-of-vocabulary class names, similar to Configuration 2 but with a transformer-based feature extraction approach.

Each configuration is trained on the set of 25 seen classes, and the model's performance is evaluated on the 25 unseen classes. During training, the objective is to align the visual features with the semantic embeddings generated by either `word2vec` or `fasttext`, enabling the model to generalize to new, unseen classes during testing. The configurations allow us to compare the effects of CNN-based (ResNet-50) versus transformer-based (ViT Base) feature extraction, as well as the impact of word-level (`word2vec`) versus subword-level (`fasttext`) embedding models. The training loss is shown in Figure 1.

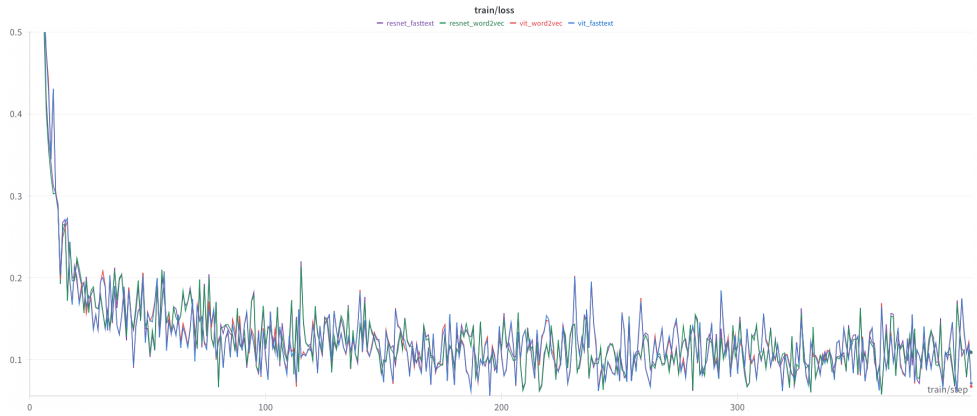

Figure 1: Training loss for all the configurations. Although, the loss did not vary significantly for all the four configurations, the ZSL performance of all the four configuration on the test classes varied significantly due to the three different normalization scheme.

3.2 Performance Evaluation

For zero-shot learning (ZSL), accuracy is calculated as the mean per-class accuracy over all the 25 unseen classes. This approach ensures that each unseen class contributes equally to the overall ZSL accuracy, preventing biases toward classes with more samples. The formula for ZSL accuracy over unseen classes is:

$$\text{ZSL Accuracy} = \frac{1}{|\mathcal{U}|} \sum_{c \in \mathcal{U}} \frac{\text{Number of Correct Predictions in Class } c}{\text{Total Number of Samples in Class } c}$$

where $\mathcal{U}$ represents the set of unseen classes, and each class accuracy is calculated individually. The ZSL accuracy for each configuration is shown in Table 1, allowing us to compare the impact of different feature extractors (ResNet-50 vs. ViT Base) and embedding models (`word2vec` vs. `fasttext`) on the model's zero-shot performance.

Configuration	ZSL Accuracy (%)
ResNet-50 + `word2vec`	19.95
ResNet-50 + `fasttext`	35.55
ViT Base + `word2vec`	23.03
ViT Base + `fasttext`	**40.23**

Table 1: Zero-Shot Learning (ZSL) Accuracy for Different Model Configurations

Table 1 presents the mean per-class ZSL accuracy for different configurations, enabling analysis of CNN-based versus transformer-based feature extraction methods, as well as word-level versus subword-level embeddings in zero-shot scenarios. It is clear that the ViT base model with `fasttext` gives the best ZSL accuracy.

3.3 Relevant Links of Results

The following links provide access to various resources related to the results of our experiments. We have kept all of our codes in GitHub.

- **Code:** The Python files for training, testing and plotting can be found on https://github.com/soumenkm/IITB-GNR650-ADLCV/tree/main/Assignment2. If the repo is not accessible (being a private repo) then one can request for access to 23m2157@iitb.ac.in.
- **Checkpoint:** All of our results are reproducible in less than 1 minute of time. Therefore, we do not provide any checkpoints.

3.4 Explanation of Codes

We have used multiple Python files to simplify the training and testing processes. The functionality of each of the Python files available in the main GitHub repository is explained below:

- `dataset.py`: This script is responsible for managing the dataset. It typically includes functions to load, preprocess, and split the data into seen and unseen classes.
- `main.py`: This is the primary script that coordinates the training and evaluation of the model.
- `text-encoder.py`: This script defines the text encoding or embedding model used for generating class vectors based on class names. Depending on the specified configuration, it might use `word2vec` or `fasttext` to create embeddings that serve as semantic representations of each class.
- `visual-encoder.py`: This script defines the visual encoder, responsible for extracting feature representations from images. Depending on the configuration, it loads a pretrained ResNet-50 or ViT Base model to generate feature vectors for each input image.

4 Conclusion

This assignment investigated the application of zero-shot learning (ZSL) techniques with a focus on normalization methods, model configurations, and performance evaluation. Beginning with an exploration of normalization strategies, the study implemented Normalize+Scale, Attribute Normalization, and Class Normalization (CN), demonstrating their impact on stabilizing model training and enhancing feature distribution across classes. Class Normalization, in particular, showed effectiveness in reducing loss surface irregularities and promoting more consistent inter-class variance, resulting in improved generalization to unseen classes.

The model configurations tested combined ResNet-50 and ViT Base feature extractors with two word embedding models, `word2vec` and `fasttext`, each impacting the model's zero-shot performance. The results showed that ViT Base, when paired with `fasttext`, achieved the highest ZSL accuracy on unseen classes, underscoring the benefits of transformer-based feature extraction and subword-level embeddings for handling vocabulary limitations in ZSL tasks.

In summary, this study demonstrated that a well-designed normalization strategy, combined with advanced feature extraction and embedding techniques, can significantly enhance zero-shot learning performance. Future work may explore additional configurations, such as alternative transformer models or unsupervised embedding techniques, to further advance the field of zero-shot learning.

References

[1] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *arXiv preprint arXiv:1607.04606* (2016).

[2] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[3] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[4] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781.

[5] Ivan Skorokhodov and Mohamed Elhoseiny. "Normalization Matters in Zero-Shot Learning". In: *CoRR* abs/2006.11328 (2020). arXiv: 2006.11328. URL: https://arxiv.org/abs/2006.11328.

[6] Bichen Wu et al. *Visual Transformers: Token-based Image Representation and Processing for Computer Vision*. 2020. arXiv: 2006.03677 [cs.CV].

[7] Yongqin Xian et al. "Zero-Shot Learning—A Comprehensive Evaluation of the Good, the Bad and the Ugly". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 41.9 (2019), pp. 2251–2265. DOI: 10.1109/TPAMI.2018.2857768.