AAIT – Homework 2

Andrei Dugăeșescu IA2

Leaderboard username: Andrei

Repository: https://github.com/Andrei-d6/AAIT

Accuracy (task1/task2): 82.24/83

The following chapters will describe the approaches used for tackling the two image classification tasks – how all data aspects (and challenges) were handled, the models used throughout experimentation, the training procedure and any other relevant detail for replicating the work conducted for this assignment. The source code for both tasks can be found in this repository alongside detailed instructions for setting up the environment and its dependencies.

Task 1

The first task involved the problem of classifying images when some of them have missing labels and the core idea behind the proposed solution is the use of pseudo labeling. Pseudo labeling, in short, is the process of training a model on labeled data and then using that model in order to predict labels for the unlabeled samples. Afterwards, the pseudo-labeled data can be used in conjunction with the initial train, labeled data in order to form a bigger (and hopefully more informative) training set and retrain the model. The network architecture chosen for both tasks is a ResNet50. This decision was made after careful experimentation (and consideration) with other architectures such as VGG16, ResNet34, ResNext50 etc. Fine tuning a pretrained ResNet50 seemed to offer the highest levels of performance in terms of validation accuracy within a reasonably small number of training epochs.

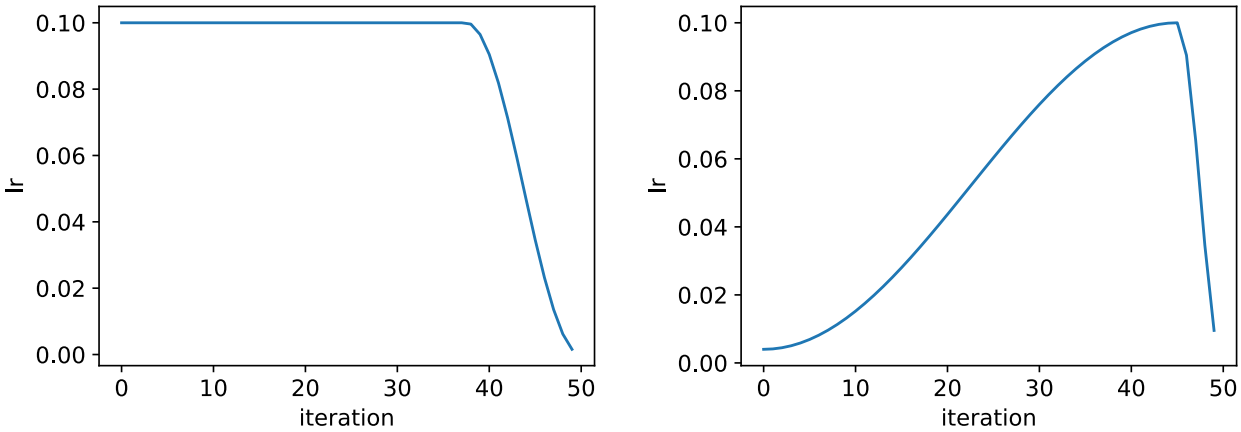
The entirety of the training procedure, alongside the results obtained on a per-epoch basis can be seen in the **`Task1.ipynb`** notebook, available in the previously mentioned public repository. A brief overview of the training setup is as follows. The first step is represented by training the ResNet50 classifier on the training data that is labeled – more exactly, fine-tuning a pre-trained ResNet50 on the labeled data. For this training methodology, the labeled samples were split 80/20% between train and validation. The validation set consisting of 20% from the initially labeled data is then used throughout the entirety of the training procedure. The idea in this case was to have a constant reference point for evaluating the evolution of model performance. After finishing the first fine-tuning loop, the trained model is then used in order to predict the classes for the unlabeled samples which are then added to the training set (again, the validation set remains unchanged). The second step is then represented by training a classifier on the dataset consisting of both the labeled and pseudo-labeled images and then using it to re-label the unlabeled data in the hopes that it would predict better classes than the model trained only on the initially labeled images. Lastly, a third and final ResNet50 classifier is trained and used for creating the predictions on the test set – and therefore the final submission.

The methodology presented above covers the basic steps used for developing a solution for this task. That being said, there are a number of significantly impactful details that are the subject of the following paragraphs. Although the core idea remains unchanged (pseudo-labeling for two iterations), the following details managed to improve the performance, on the validation set, by up to 20% in terms of accuracy.

A very important aspect is how the data was handled before being passed to the model for training. Early experimentation revealed that training on the original size of the images – 64x64, produced considerably lower levels of accuracy when compared to when they are upsampled to the size of 128x128 or even 224x224 pixels and therefore, some level of upsampling was employed throughout the training procedure (as it will be described shortly). Since the classifier uses a pre-trained ResNet50 backbone, naturally, the images are normalized using ImageNet statistics (mean and standard deviation). As for the data augmentation part, the images are randomly resized, cropped and flipped horizontally (random crops of a given size are created, more details here).

The CNN classifier is trained using an Adam optimizer, with cross-entropy as the objective function using mixed precision training (training partially in half-precision floating point). The reason for using mixed precision training is twofold. Firstly, it sped up the training time and significantly reduced GPU memory usage and secondly, also seemed to improve the model's generalizational capabilities. A short rundown of what mixed precision training entails is the following: first compute the output of the model with half floating-point precision (FP16) and then back-propagate the gradients in half-precision. Afterwards, the gradients are copied in FP32 precision and are updated in the master model (in FP32 precision). A more detailed explanation is presented in [1].

Another important addition to the training setup is represented by training using progressive resizing. Progressive resizing in this case consists of training the model with images at a lower resolution before conducting the last fine-tuning at a higher resolution. For the final version of the solution proposed for this task, each model is trained with images resized to 128x128 and then is fine-tuned again, only this time with images resized to 224x224. This might sound a little excessive especially given how small the training data is (64x64 pixels) but proved to notably improve the final test accuracy (~3%). The addition of progressive resizing is essentially a way of squeezing out a bit more performance out of the network. Having said this, the final training procedure is the following: initial train at 128x128 followed by another fine-tune with 224x224 images (the starting weights for the 224x224 part are the final weights from the 128x128 training). Use the resulting model to create labels for the unlabeled data and then re-train. Repeat the pseudo-labeling and retraining again and then use the last model in order to create the predictions on the test data (the submission). The last aspect worth discussing is how exactly training (at each resolution) looks like. One such training instance is as follows: the pre-trained ResNet50 backbone is initially frozen and only the linear, classification layers are trained following a cosine annealing policy called flat cos as presented in Figure 1.



a) flat cos lr scheduling – start % = 0.75 b) one cycle lr scheduling – start % = 0.9

Figure 1 – Learning rate scheduling policies – examples for a maximum learning rate of 0.1 over 5 epochs (10 iterations each)

After training the linear layers, the backbone is unfrozen and the whole model is trained with a much lower learning rate following the one cycle learning rate scheduling policy shown in Figure 1. This methodology of training with a frozen backbone followed by a little more fine-tuning on the whole network represents a training block. One such block is used for every resolution – one block of training on images of size 128x128 and one block on 224x224 data. Two such blocks make up a training iteration. After the initial train, which starts off the pseudo-labeling process, there are two more training iterations. For the exact number of epochs, and learning rate used please see **`Trained/v2/Task1.ipynb`** from the previously mentioned repository. Moreover, all the setup configurations can be seen in the **`configs/`** directory. The learning rate scheduling policies do make up a drastic difference in the final performance of the model. Whilst the fit flat cos approach helps with the convergence of the linear layers in the initial part of one block of training, the one cycle policy covers the final part of the fine-tuning. Although these policies were chosen initially based on intuition, experimentation revealed that they can have a big positive impact of the final performance of the model.

The last training aspect, which also concerns Task 2, is the checkpointing storage. During each training step, the best model in terms of validation loss is saved alongside the model after completing the number of training epochs (ideally these two should be the same). Therefore, each training call will result in two models being saved – the best one and the last one. Referring back to the "terminology" previously introduced, one training block (same model trained on two different sizes for the input data) will generally produce at most 8 models (if everything is wanted to be saved and not overwritten) – two for training with the frozen backbone, two for fine-tuning the entire network (again, two meaning the best and the last which might be different), repeated again for the second image resolution. The same checkpointing approach is also employed for the second task. If the number of saved models seems too high and takes too much space, one could simply overwrite models from one step to another, or only keep the best one after each training call – the code allows for a reasonably high degree of flexibility in this regard (checkpointing).

Going further one aspect worth experimenting with is what labels are actually kept for annotating the unlabeled data. In this current version, predictions are generated for all the unlabeled samples regardless of model confidence or any other quality related score for the prediction. A simple approach would be to only include predictions that are over a certain level of confidence e.g., 95%, but smarter approaches could be more beneficial. From an intuitive point of view, the model performance is heavily impacted by the quality of the data that it is trained on. Therefore, ensuring that only high-quality predictions (in terms of correctness) get used should improve the final test accuracy.

Task 2

The second image classification task from this homework assignment involved the use of noisy labels in the training data. The main concern that tried to be addressed was developing a classifier that is robust enough to still manage enough generalization even when confronted with significant amount of noise in the input labels. Generalization was also an issue for the first task, where many experiments led to a very low training loss in comparison to validation (and therefore poor performance). For the first task, data augmentation, in conjunction with the chosen network architecture seemed to alleviate most of those issues of overfitting, the test accuracy being extremely close (if not a little higher) to the performance exhibited by the model on the validation set – which needless to say was a fortunate sight. That being said, a number of decisions to further address the problem of overfitting and generalization went into the solution proposed for this second task and are

the subject of the following paragraphs. The main idea behind this task was as follows: fine-tune a classifier on the noisy data and then use it in order to derive a number of outliers (hopefully most of them). Outliers in this case refers to predictions for a certain class that are far off from all the other samples predicted for that class. Using this fine-tuned model, the training data is cleaned – only keeping those samples that tend to cluster together. Another model is trained afterwards on the cleaned dataset and is then used as the final predictor for the test set.

Regarding the changes in the training procedure for addressing the problem of noisy data: firstly, the objective function was changed from cross-entropy to cross-entropy loss with label smoothing. Secondly, MixUp was introduced in the training procedure, at each step. Progressive resizing was no longer used for this task since the focus was on finding the most relevant connections between samples belonging to the same class instead of trying to squeeze as much information as possible for each sample. The last change from the methodology presented in the first task is the inclusion of random erasing as part of the data augmentation – random patches in the image are replaced with gaussian noise (more details can be found here).

The next aspect worth discussing is the cleaning process that takes place after training on the noisy data. The gist of the approach would be the following: un unnormalized representation of the joint distribution between the predicted label and the given label from the dataset is used in order to algorithmically flag the candidates which are likely to be error labels. This joint distribution called the *confident joint* is estimated by counting all the examples with noisy label and a high probability of actually belonging to the target label. A far more comprehensive explanation of this approach is presented in [2] [3]. As from a more practical standpoint, the cleaning logic is implemented using functionality from the CleanLab framework [4] for confident learning.

Having completed the cleaning process, using the previously mentioned approach, another model is then trained using the clean version of the dataset which contains roughly 80% of its original size (approximately 20% of the labels were considered noisy and were subsequently removed). It should be stated that both the model used for removing the noisy samples, as well as the one trained on the cleaned data start from the set of weights obtained at the end of the training process from Task 1. This was done because the datasets from the two tasks share some structural similarity and because experimentation revealed this approach to provide slightly better results. Similar to the training setup from Task 1, the initial model (the one used for cleaning) is first trained with a frozen ResNet50 backbone using the flat cos learning rate scheduling followed by a little more fine-tuning on the entire network using the one cycle policy. The final classifier is trained exclusively using the flat cos policy, simply because it could be trained in that fashion for longer without struggling to converge.

For any other detail regarding the training setup, such as the exact number of epochs or the learning rates used, they can be found in the **`Trained/v2/Task2.ipynb`** notebook as well as in the **`configs/`** directory. In order to reproduce the results simply follow the instructions presented in **`Task1.ipynb`** (for the first task) and **`Task2.ipynb`** (for the second task) which essentially amount to running the entire notebook from top to bottom. All the configurations regarding loading the dataset, previous checkpoints or any hyperparameter can be changed/updated by modifying the **`.yml`** files from the **`configs/`** directory (and updating the path to the YAML file in the training notebook, if necessary).

References

[1] Fast.ai, "Mixed precision training," 2020. [Online].
Available: https://docs.fast.ai/callback.fp16.html. [Accessed 20 January 2023].

[2] C. G. Northcutt, A. Athalye and J. Mueller, "Pervasive Label Errors in Test Sets Destabilize Machine Learning Benchmarks," 8 April 2021. [Online].
Available: https://arxiv.org/abs/2103.14749. [Accessed 20 January 2023].

[3] C. G. Northcutt, L. Jiang and I. L. Chuang, "Confident Learning: Estimating Uncertainty in Dataset Labels," 21 August 2022. [Online]. Available: https://arxiv.org/abs/1911.00068. [Accessed 20 January 2023].

[4] C. Inc, "cleanlab: The standard package for data-centric AI, machine learning with label errors, and automatically finding and fixing dataset issues in Python.," PyPI, [Online]. Available: https://pypi.org/project/cleanlab/. [Accessed 20 January 2023].