Deep Learning Course: Multi-Label Image Classification

Johanna Männistö **Vilhelm Hovland**
johanna.mannisto@helsinki.fivilhelm.hovland@helsinki.fi

1 Introduction

Image classification is a classic deep learning problem, and this multilabel image classification task poses a challenging version of this problem, with 14 different unevenly distributed categories that each image must be labeled in terms of. Our approach mainly consisted of attempting to utilize finetuned pretrained models, using the pytorch library.

The code for our project can be found at github.com/jmannisto/Multi-Label-Image-Classification

2 Data and Data Processing

The data for this task was initially stored in two folders. The first folder contained 14 different *.txt files, each named for a label used for this multi-label classification task. Each text file had line-separated image names for the images with that particular label. Image names may show up in more than one file (i.e. images may have several labels). The second folder contained 20,000 images, each of which was assigned between 0 and 5 labels. Most images only had one label assigned. Notably, the distribution of labels within the dataset was not uniform. As can be seen in Figure 1, the label 'people' had by far the most appearances within the dataset. To prepare the data for training a model, the images and their corresponding labels had to be combined. To do so, each of the image names listed amongst the 14 text files was gathered to create a dataframe of multi-hot encodings. Each image name was associated with multi-hot encoding where 1 under a category indicated the image name's presence in the corresponding text file. This can be found in the data_extract.py script[1]. After combining all pieces of information regarding the data, it was prepared to be loaded into a model.

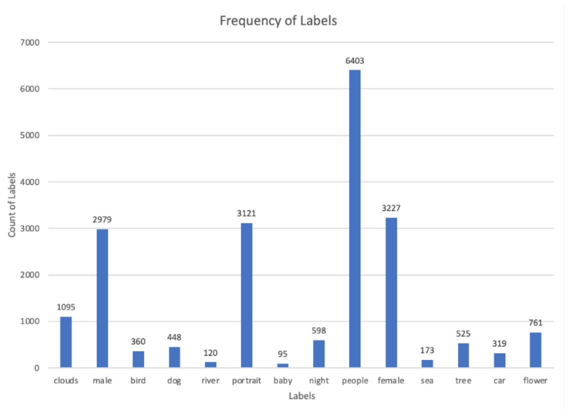

Figure 1: Distribution of Image Labels

Each image was transformed into a 224x224 image instead of keeping the 128x128 shape to match the input the pretrained models expected.

The complete dataset was split into train, validate, and test sets at an 0.8/0.1/0.1 ratio. This meant our train set had 16,000 images, and our validate and test sets had 2,000 images each. The entire dataset was shuffled with a random seed of 42 before splitting the data. Details of this split and the dataloader can be found in the data_load.py script. Most of the images were in RGB format,

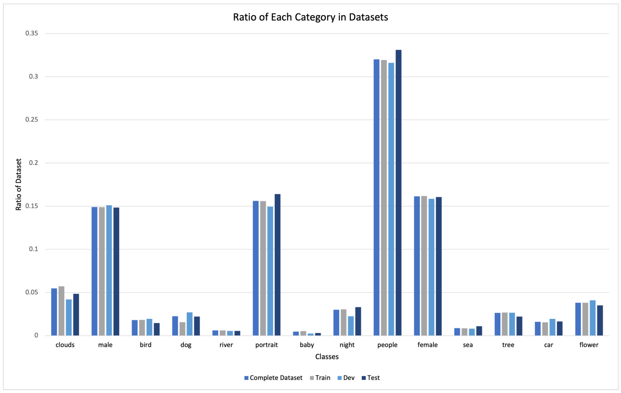

Figure 2: Label Distribution by Data Set

although there were a few which appeared to be grayscale. We decided, in order to pass as much

[1]See github.com/jmannisto/Multi-Label-Image-Classification for project code

information to the model as possible, to keep the images as RGB and pass 3 channels to the models. For images that only had one channel, this channel was duplicated twice to create a total of 3 channels for the image.

3 Our Approach

To achieve the highest accuracy possible, we decided to finetune pretrained image classification models. Finetuning a model comes with multiple benefits: we can reduce our computation costs by avoiding training a model from scratch (and thus reducing our carbon footprint), and it also allows us to leverage the knowledge of the model trained on much larger datasets and more complex tasks than the one at hand. PyTorch has an assortment of pretrained models that can easily be finetuned, either by updating all the parameters within the model, or by using "feature extraction" where only the final layer weights are updated. Unless otherwise stated, all pretrained models discussed below use their default weights, SGD as an optimizer with a learning rate of 0.001. All models used BCEWithLogitsLoss as a loss function. Whenever pos_weights were used, values of [17.26484018, 5.713662303, 54.55555556, 43.64285714, 165.6666667, 5.408202499, 209.5263158, 32.44481605, 2.123535843, 5.197706848, 114.6069364, 37.0952381, 61.69592476, 25.28120894] were used which corresponded to the count of negative samples/positive samples for each class (see Section 3.3 for more information).

The details of training each model, their hyperparameters, and results are detailed in sections 3.1 - 3.5.

3.1 AlexNet

The first pretrained model chosen was PyTorch's pretrained AlexNet[2], a model with 61.1M parameters, where only the final layer was finetuned. At first glance, this model appeared to perform very well, achieving 92% accuracy. However, it seemed to be stuck at 92% accuracy despite efforts to train for longer and leverage different image transformations. Examining the predictions of the model made it immediately evident that the model had learned to predict only 0s to minimize the loss.

Examining the loss between the dev set and train set indicate that there wasn't a large amount of variance between the datasets. This pattern of low variance appeared across all of our models. The low variance indicated to us that we needed to consider different optimizations. In scenarios of high bias, training longer, a bigger network, or using different optimizations would be needed.

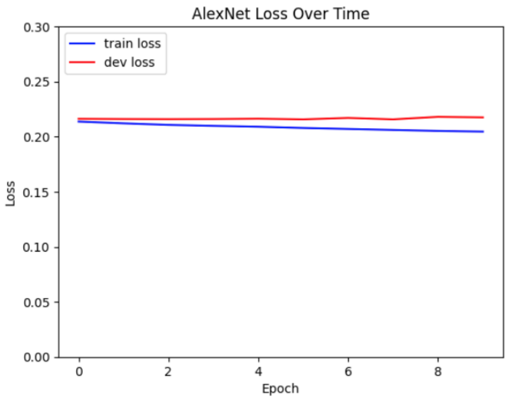

Figure 3: AlexNet Loss by Epoch

We also tested the pretrained model where no additional finetuning was done outside of reshaping the output layer to match the number of labels our task required. This resulted in a model that had a fairly high recall and higher precision than the other models, although the overall accuracy was lower. It was evident from this not-finetuned model that the model training shifted the model to default to predicting 0s.

The first attempt to break this habit of the model was to use different image transformations to see if that convinced the model. However, image transformations didn't yield any benefits to the model (RandomRotation(25) and ColorJitter were tested). Using other optimizers such as Adam and RMSprop were also detrimental to model performance.

Next, finetuning the entire model instead of the final layer was done to see if influencing the features learned in the CNN would improve results. There was some minimal improvement although nothing significant. The results of this though led us to focus primarily on doing full finetunes instead of tuning only the final layer of pretrained models. Table 1 summarizes findings of AlexNet models and their hyperparameters and results.

Later we started applying weights to the loss function, discussed further in 3.3 DenseNet121, which seemed to encourage the model to avoid solely predicting 0s. This however led to a reduction in overall accuracy.

[2]See PyTorch docs for details

Model	Pretrain Type	Optim	Epochs	Weighted?	Accuracy	Precision/Recall
1	N/A	N/A	N/A	No	50.94%	0.07/0.51
2	Feature Extract	SGD	10	No	92.71%	0.03/0.00
3	Finetune	SGD	15	No	92.75%	0.04/0.00
4	Finetune	SGD	15	pos_weight	69.11%	0.07/0.31
5	Finetune	SGD	30	pos_weight	84.35%	0.07/0.12
6	Finetune	SGD	40	pos_weight	85.75%	0.07/0.11
7	Finetune	Adam	30	pos_weight	34.64%	0.07/0.71

Table 1: AlexNet Model Hyperparameters and Results

3.2 VGG

VGG[3], introduced in the paper Very Deep Convolutional Networks for Large-Scale Image Recognition, is another pretrained model using deep convolutional networks of varying dimensions. The paper suggests, and pytorch offers, versions of this network of weight layer depths varying from 11 to 19 layers and from 133 to 144 million parameters. We tested the networks of depth 11, 13, and 16, only adapting and updating the final layer. We found that the performance barely differed between these versions on our dataset, achieving accuracies of 92.8, but, as the precision and recall scores suggest, ultimately outputting negative predictions for all labels except "people", which was the label with the most support. Introducing pos_weights to the loss function significantly reduced the performance; the precision and recall scores show that more positive predictions are outputted, but very few of them are correct. An additional approach that was experimented with for the VGG models was introducing a "prediction threshold" parameter, instead of rounding the outputs (which would be equivalent to a prediction threshold of 0.5). These experiments were not very successful however, usually only improving recall but not precision, at the expense of accuracy. The best scores achieved with these models (in terms of precision and recall) used the following hyperparameters: 24 epochs, finetuning all parameters, positive weighting, and a learning rate of 0.002. Table 2 summarizes the scores using different prediction thresholds.

3.3 DenseNet121

AlexNet is a model with 61.1M parameters, although only the final output layer's weights were updated in the process, we wanted to compare the different approaches to transfer learning available with PyTorch. DenseNet121 has 8M parameters, much smaller than AlexNet. We chose this model with significantly fewer parameters for quicker training as well as to compare the capabilities of transfer learning between two different architectures.

Two finetuned models were trained using the pretrained DenseNet121 model. The first model was trained with a feature extraction approach, i.e. only the final layer of the model was updated while the weights for all other parameters were kept frozen. The second model was trained such that all parameters were updated during training.

Following suggestions from the PyTorch, we added weights, specifically 'pos_weight' values to the BCEWithLogitsLoss loss function. When training with pos_weight for a class being greater than 1.0, the network should produce more positive predictions, in our case, more predictions of 1.0 than 0. The drawback is that these predictions may actually be false positives, essentially increasing the recall of our model but decreasing its precision.

There was a significant amount of trial and error when testing out different weights. At first, the weights were defined as $\frac{\#ofnegativesamples}{\#ofpositivesamples}$. These weights were adjusted after training and examining the classification matrix on the sets. Table 3, summarizes various different hyperparameters used with the DenseNet architecture.

Figure 4 shows the validation accuracy of the fine tuned and feature extracted models throughout training. It appears that both models immediately achieve a little over 60% accuracy on the validation set but quickly after, the model where all parameters are updated began outperforming the Feature Extraction model where only the final layer is updated. Making updates on all layers in a CNN is likely very beneficial as the model can update its filters and the features it's learned to correspond to the categories for the task at hand rather than what it was trained on which may be more complex or

[3]See Pytorch docs

Threshold	Accuracy	Macro Avg. Precision	Macro Avg. Recall
0.5	82.2%	0.07	0.14
0.4	80.4%	0.07	0.16
0.3	78.3%	0.07	0.19
0.2	76.2%	0.08	0.24
0.1	74.0%	0.07	0.26

Table 2: Thresholds and Results

Model	Pretrain Type	Optim	Epochs	Weighted?	Accuracy	Precision/Recall
1	N/A	N/A	N/A	No	57.08%	0.07/0.48
2	Feature Extract	SGD	10	No	92.71%	0.03/0.00
3	Feature Extract	SGD	15	pos_weight	54.81%	0.07/0.45
4	Feature Extract	SGD	30	pos_weight	58.16%	0.07/0.42
5	Finetune	SGD	15	No	92.73%	0.02/0.00
6	Finetune	SGD	15	pos_weight	77.60%	0.08/0.20
7	Finetune	SGD	30	pos_weight	77.80%	0.07/0.19
8	Finetune + RandomRotation(25)	SGD	30	pos_weight	68.04%	0.07/0.30
8	Finetune + ColorJitter)	SGD	30	pos_weight	77.61%	0.07/0.21

Table 3: DenseNet Model Hyperparameters and Results

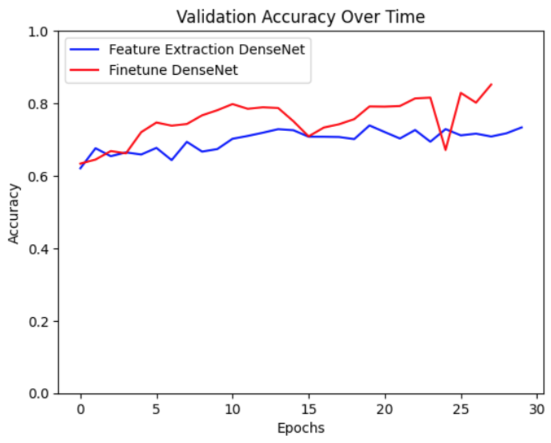

Figure 4: DenseNet Validation Accuracy by Epoch

simple than needed. Adding weights added motivation to the model to make positive predictions of labels, but as a result, due to the sparseness of our dataset it also resulted in lower accuracy.

3.4 SqueezeNet

Squeezenet[4], introduced in the paper SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size, introduces a significantly smaller model architecture than Alexnet and VGG, and advertises equivalent performance. It did not avoid the pitfalls the earlier approaches proved vulnerable to however, and did in fact start predicting all negative labels very quickly, after only a couple epochs of training. This could be due to the much smaller number of parameters in this model compared to the others.

3.5 Basic CNN

For a proper comparison, we also decided to train a CNN from scratch to see a model's performance without benefitting from transfer learning. The CNN used Adam as an optimizer with a learning rate of 0.001 and weight decay of 0.0001. Like all other models in this paper, the model used BCEWithLogitsLoss as the criterion. The CNN had 3 layers, with a dropout of 0.2, ReLU activation function, and MaxPool of size 2x2 and stride of 2 on each layer. Layer 1 was convolved with 16 filters of size 3x3 with a padding of 1, and layer 2 was convolved with 32 filters of size 3x3. Two fully connected layers then complete the CNN, the first fully connected layer outputs 1000 units while the final fully connected layer outputs the number of labels - 14.

This model immediately had an accuracy rate of 92% and did not see any improvement throughout training.

[4]See PyTorch docs

4 Evaluation

For evaluating our models, we are focusing on label prediction accuracy, precision, and recall.

We used the classification_report from SciKitLearn to identify precision and recall on a label by label basis on the test and development sets when and after training the models. The classification reports consistently revealed that labels with more support, primarily the "people" label with approximately twice the support of the second most frequent label, were much more likely to receive positive predictions by the models; certain models, such as the VGG model with Feature Extraction, did not successfully predict any other label than "people" as revealed by the classification report. It only reached a precision of 0.30 and recall of 0.11 for this label, however. This pattern of higher precision than recall was common to all models trained using non-weighted loss functions, implying that positive predictions tend to be very uncommon. With high positive weights for the loss function, the recall would usually be greatly positively affected but the precision only slightly improved, if at all. This in turn implies an increased quantity of positives, but that are largely inaccurate (poor quality). This corresponds with the BCEWithLogitsLoss documentation on PyTorch which states that pos_weights greater than one increase recall while pos_weights less than one increase precision.

We calculated accuracy by counting correct predictions per label per sample. This led to misleadingly high accuracy values; seeing as the average number of positive labels in the data being fairly low, the baseline accuracy of a model outputting exclusively negative predictions would be at around 90; therefore, an accuracy of for example 92.8 is not actually very promising.

5 Conclusion

The final model chosen was a DenseNet model trained with pos_weights for 15 epochs. It had the highest precision of 0.08 on our test set. In general it also had an acceptable recall for the results we were finding and accuracy of around 77%. While this doesn't match the 92% accuracy models could achieve by predicting solely 0s, we believe that this model has learned something a bit more than simply guessing 0s.

This was a very challenging task. It is no small feat to convince a model which is designed to optimize to pick a more challenging route and to learn features. Implementing weights and experimenting with prediction thresholds allowed us to modify the task for the models and resulted in more positive predictions, although precision remained low.

With more time and resources we would like to try ensembling models and testing out one vs. rest approaches to see if these setups encourage the model to learn features from the dataset and predict more than simply 0s. Adding additional tasks for the model to optimize for, could help the model use shared parameters to learn relevant features rather than optimizing solely for a task where predicting no labels results in low loss.