IDL 2021 Final Project: Image Classification

Team DataKiller: Ville Puurunen, Lassi Virtanen, Sahin Dursun

1 Introduction

Drawing inference from real-world data has become an increasingly complex task. Modern advances in data science, computer science and software development strategies have unlocked a wide range of unforeseen potential for automated inference, decision making, and reasoning. From these applications emerges, amongst many others, a need for reliable classification of multidimensional data. One such task is finding labels for multi-labeled image data, a problem that has been applied to demanding data from *e.g.* oncology, robotics, and autonomous transportation systems, to name a few.

In comparison to a simplified, binary classification tasks, multi-label classification problems are demanding to satisfy. Computing, for example losses and accuracies in such situations requires broader consideration of labeling of the data as well as making sure the model optimizes for multiple labels per observation, instead of one. In binary classification tasks, one does not usually consider such questions.

In this report, we discuss two artificial neural networks suitable for multi-label classification problems. We also present our results for a particular multi-label classification task. In addition to architectural considerations, we give an overview of a handful of different loss functions and optimizers used in measuring the models' performance. The final, putative results of the task are submitted as a separate csv-file for the instructors to evaluate. However, to give an initial approximation of our models' performance, we present the models' performance in this report on validation data, as measured by unweighted as well as weighted F1-score.

1.1 Multiclass problem

There are quite a few challenges when implementing multiclass classifcation for multilabel images. Definition of accuracy, dataloading procedures, oversampling and undersampling.
The dataset consists of 20000 images with up to 14 labels with a heavy amount of class imbalance. To counter this, zero label images (approximately 50% of the dataset) were removed.

2 Data wrangling

2.1 The 12 data wranglings of Heracles

Our file layout underwent a lot of changes before we managed to find a good solution. First we tried to split the images into folders by label like in Homework 2 and then feed the images to Pytorch's Imagefolder to get a labeled dataset, but it was only after we had already done this that we realized that we were dealing with a multi-label classification problem. We then started over and split the data into two folders, training and validation. We looked at the images and found that they seemed to already be in a mostly random order, so we didn't think it was necessary to shuffle the images before splitting them. We found that we could make a custom Pytorch Dataset that we could use to feed the images and labels to a DataLoadewr object, which required quite a bit more wrangling. This implementation was largely copied from [?]. Our troubles didn't end there, as we discovered that it was beneficial to delete all images that didn't have any labels from the dataset, which required our thankfully final bit of data wrangling.

Some of these were done manually i.e. by dragging and dropping images into folders and then using git to get them to Puhti, but most of the files we used for wrangling can be found in the Wranglings folder of the repository. Keep in mind that the original location of the files was different, so they won't work if run in that location.

2.2 Data transformations

As a form of regularisation, we applied transformations to the training set. We used randomHorizontalFlip(), RandomResizeCrop and RandomRotation. Of course, we also had to resize images for transfer learning.

3 Model selection

3.1 Architecture

The network architectures tested were focused on primarily two options.

3.1.1 ResNet34

We implemented transfer learning by using a pretrained ResNet34 network where the last fully connected layer maps to the 14 class outputs with pretrained weights. The ResNet consists of 34 convolutional layers with varying convolution kernel dimensions (3x3 to 7x7). The key advantage of transfer learning is that we can use weights trained on massive datasets (ie. the ImageNet that is trained on 1 million images). It is possible to freeze the network and restrict updates to the last layer and simply extracting features from the pretrained net. Another option is to tune the entire network without freezing gradients. ResNet and other pretrained nets require data in a standardized format (224x224) PIL-images. []

3.1.2 CNN

Our initial efforts to implement a multi-label classifier applied a CNN architecture with three convolutional *Conv2d* layers, and ReLUs as activation functions. We implemented a MaxPool filter between the first and second convolutional and two *BathcNorm2d* functions to reduce internal covariate shift after second and third convolutional layers. At the end of the network, signals were squeezed through a linear layer and flattened with sigmoidal function to produce an output.

To experiment further, we decided to leave one convolutional layer out from our second CNN architecture. This resulted, as expected, in a decreased level of accuracy, thus guiding us to bring back the third convolutional layer. However, after implementing a ResNet architecture discussed above, our initial CNN architectures turned out to be of inferior accuracy, and we decided to continue with only ResNet architectures as our main classifiers.

3.2 Optimizers

3.2.1 Adam

We tried Adam with a lower than default learning rate (1e-4) and the default learning rate.

3.2.2 RMSprop

RMSProp was used with it's recommended settings (learning rate 0.001, alpha 0.9)

3.2.3 Adadelta

We used the adaptive learning rate algorithm Adadelta with default learning rates.

3.2.4 SGD with scheduled learning rate updates

Another alternative would be to use Stochastic Gradient Descent with learning rate and momentum updates. There are numerous scheduling routines available.

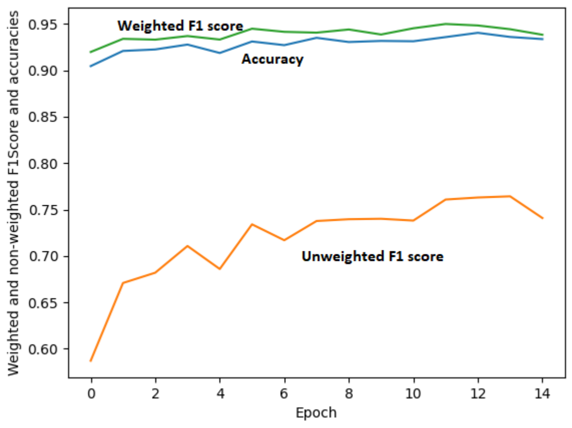
3.2.5 AdamW

AdamW with default values was used for both the CNN and ResNet architectures. It had relatively good results on the CNN but didn't perform as well as SGD or Adadelta on the Resnet.

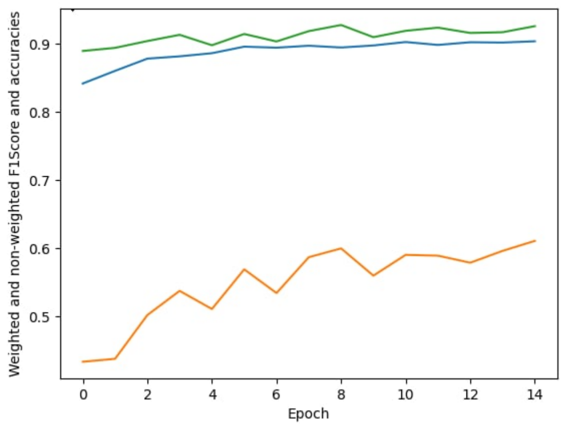
4 Training and validation

After we managed to wrangle our dataset into a desirable form, we got to work on testing different optimizers and loss functions. We first tried Adam, which was able to slowly improve its Fscore with a learning rate of 1e-4. Using the default learning rate of 1e-3, the loss and Fscore of the network jumped around a lot more, but was able to reach relatively good results quite early. These results would naturally require checkpointing to be utilized however. RMSprop with default and lower learning rates seemed to behave similarly, with the results bouncing between 0.2 and 0.4 Fscore. AdamW performed fairly well on the CNN but more poorly on the ResNet architecture. SGD and Adadelta did well on both architectures.

We also tested ResNet for feature extraction (that is, freezing all layers except the last). The results from this were unconclusive as the number of epochs might not have been sufficient, but generally performance was found to be inferior to fully optimizing the network.



(a) ResNet Training with full network tuning



(b) ResNet: Learning weights for last layer only

4.1 Loss functions and loss

Due to its suitability for multi-label classification problems, we used mainly *Binary cross-entropy with logits* as our primary loss function. However, to directly optimize for F1-score, we experimented with implementing a continuously differentiable F1-loss function *s.t.* during backpropagation the optimizer would increase the model's F1 score. This turned out to not work as intended, since the function seemed to optimize for accuracy rather than F1 score, leaving the model to weight negative classifications more than true positive ones.

4.2 Validation

Accuracy was difficult to determine at first, as we didn't really know whether to count accuracy as the portion of completely correctly labeled images or simply the portion of all correct labels. When we got the information to use F1-Score as the main metric:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R + \varepsilon}$$

Where the small constant ε was used to avoid division by zero and P, R are defined as

$$P = \frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fp}}$$
$$R = \frac{\mathrm{tp}}{\mathrm{tp} + \mathrm{fn}}$$

Accuracy, as discussed above, was defined as:

$$acc = \frac{\mathrm{tp} + \mathrm{tn}}{\mathrm{tp} + \mathrm{tn} + \mathrm{fp} + \mathrm{fn}}$$

For output $\mathcal{O}$, we used $\mathcal{O} > 0.5$ (after sigmoidal squeezing) as a threshold for *True* classification for each class. Such threshold expressed the output of the model as a binary tensor suitable for computing F1-scores. However, when computing weighted F1-scores, we did not manually adjust classification thresholds, since having label probabilities as weights seemed sufficient for satisfactory results.

5 Conclusions

CNN and ResNet networks were used for the task. We learned that the CNN architecture implemented was too small to obtain meaningful results and that a bigger Net was required. With CNN, F1-scores during training were in the range $[0.40, 0.45]$. We then moved on to using a pretrained ResNet34 without any frozen layers.

Improvements could be achieved by hyperparameter tuning and implementing a SGD optimizer with scheduler LR updates using ie. simulated annealing, utilizing a robust sampling technique for training and validation sets (ie.oversampling and undersampling as mentioned by the lecturer), using weighted F1-score as the loss function.

All in all, the project took a lot of effort and the instructions were quite minimalistic, but we learned an incredible amount particularly about the options available from Pytorch and the challenges posed by multi-label classification problems. We are quite satisfied with the results overall.

6 References

1. Example of custom Dataset class implementation and customized DataLoader `https://github.com/jiangqy/Customized-DataLoader-pytorch/blob/master/multi_label_classifier.py`

2. F1-loss implementation `https://gist.github.com/SuperShinyEyes/dcc68a08ff8b615442e3bc6a9b55a354`

3. Pytorch - Transfer learning tutorial: `https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`