Deep Learning Image Project

deepbois Project Report

Niclas Joswig
niclas.joswig@ad.helsinki.fi

Jonas De Paolis
depaolis@ad.helsinki.fi

UNIVERSITY OF HELSINKI

1 Data Exploration and Preparation

Given a training set of 20 000 images and 14 classes, this project aims to develop a classification pipeline for multi-label image classification using deep convolutional neural networks. Besides the obvious work upfront such as generating the multi-label encoding, we performed two significant pre-processing steps before feeding the image data into our neural network.

As the provided data involves both **RGB- and grayscale-encoded images**, we first need to cope with different numbers of color channels. 1 458 of 20 000 images in the initial dataset are RGB-gray, meaning that with every pixel all 3 color channels share the same value and therefore could technically be represented by 1 single channel. To prevent the model from learning the differences in encoding as a feature, we converted all RGB images to grayscale. Although this will inevitably result in information loss, we decided that working on a consistent and fully usable data set was of greater importance.

With good generalisation performance in mind, we chose to use **data augmentation** as a means of regularisation to facilitate robustness in our models. Based on the 20 000 data points given as the initial training set, we therefore generated 2 variations of each image by rotating, shifting and flipping them to a random extent, ultimately resulting in 60 000 training samples.

Further looking into the data, we noticed that there are **substantial imbalances regarding class distribution**. Dealing with 95 occurences for the smallest class *baby* in contrast to 6 403 for the largest class *people*, we cannot expect the outputs to be probabilistic and therefore need to find a way to define useful thresholds. We do so by finding the threshold that maximises our micro F1 metric for each class individually, setting the other classes' thresholds to a default value. We also tried including class weights to put more emphasis on underrepresented classes in our training data but did not notice any improvements.

2 Optimisation Approach

In order to both efficiently search for viable solutions and still understand the model intrinsics, we designed the network optimisation process to involve two phases that sequentially narrow down an initially defined hyperparameter space. The first step is to identify reasonable ranges of parameter settings and to use these ranges as a parameter grid for the scikit-learn **random search** implementation[1]. In contrast to the typical application of random search in deep learning models that is rather superficial and limited to few hyperparameters, we decided to optimise the entire network architecture in the sense that each of the layers and its attributes are generated independently. Out of all presented parameter combinations, random search aims to find an approximation of the best possible solution by exploring a pre-defined number of options. We discarded our initial idea of using grid search to solve our problem as we figured that exploring the *entire* solution space would come at the high price of computational inefficiency. Having retrieved the best parameters, the second step is to analyse trends in the documented results and try to further **improve model performance manually**.

This approach as outlined above is highly useful since we are able to randomly explore large parts of the solution space and gain insight into the effects of different parameters, yet for the most part we avoid the time-consuming and at times intransparent challenge of manual optimisation. As k-fold **cross-validation** is a built-in part of the scikit-learn random search implementation, we are able to prevent our model from overfitting on the training data by averaging over multiple train-test-splits and therefore

[1] More information on random search implementation at https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html

reducing variance of the test error estimate. During manual optimisation, we used a traditional train-test-split of 80:20 for model validation.

3 Results and Analysis

3.1 Random Search

Although it is highly unlikely to find the global best solution using random search, we establish a controllable environment that fulfils our requirements of an efficient and insightful approach. We used extensive combinations of potentially relevant parameters. Running 30 iterations led us to our baseline model scoring at 43% micro F1 without threshold optimisation. The corresponding architecture is depicted below.

```
conv1d_1:
32 kernels, (3,3) kernel size, (1,1) stride, ReLU activation, batchnorm, (2,2) max pooling
conv1d_2:
64 kernels, (3,3) kernel size, (1,1) stride, ReLU activation, batchnorm, (2,2) max pooling
conv1d_3:
32 kernels, (3,3) kernel size, (1,1) stride, ReLU activation, batchnorm, (2,2) max pooling
dense_1:
128 units, 0.5 dropout, ReLU activation, truncated_normal initialisation
dense_2:
14 units, sigmoid activation
___
adam optimiser, binary_crossentropy loss, 128 batch size
```

Looking into the 3 best models we were able to identify trends in effects that particular parameters seemed to have on model performance. Key takeaway is that networks with more conv2d layers and less dense layers perform better. This leads to comparatively deep models of about 3 convolutional layers and 1 dense layer. The number of parameters per conv2d layer stays rather small with the amount of kernels ranging from 32 to 64 and a kernel size of about (3,3). Also, per default we used batch normalisation and max pooling with every conv2d layer. The dense layer ideally has 128 neurons which, given the initial image size, is relatively small. Compared to tanh, ReLU as activation function seemed to deliver the best results in both types of layers. The dropout rate was best at 0.5. Best performing optimiser was Adam and a good batch size seems to lie at around 128 samples.

It should be noted that the random search approach could potentially deliver very good results if one decided to explore a great amount of hyperparameter combinations. Due to time restrictions, however, we performed no more than 30 iterations and used the best result as prototypical architecture for further manual improvements.

3.2 Manual optimisation

Based on the observations presented in section 3.1, we were able to develop an intuition that, over the course of multiple iterations, led us to achieve further improvements with the micro F1 score of our class predictions. All measures described below are modifications we made with the prototypical architecture. We set the number of epochs to 50 for all runs.

1st iteration

We started off by adding one more conv2d layer and setting the amount of kernels for each respective convolutional layer to 32, 64, 128 and 256 while the dense layer remained unchanged with 128 neurons. We removed the max pooling layer in the last convolutional layer to prevent unnecessary information loss. This led to 50% micro F1 (cp. fig. 1). Adding another conv2d layer did not cause any improvement in performance and we therefore desisted from further experimenting with the convolutional layers.

2nd iteration

Next, we tried a different size for the dense layer which caused the micro F1 performance to decrease drastically (cp. fig. 2). As an amount of 128 neurons seemed fairly small in view of the initial image size, we scaled it up to 1 024 neurons which gave us exactly 0% micro F1. We attribute this effect to the great amount of trainable parameters resulting in an overly high model complexity and the inability to adequately model our problem. This model solely predicts values of 0 for each class.

3rd iteration

In the final model, we scaled down again the number of neurons in the dense layer to 256 which seems to be a good middle ground. These hyperparameter settings gave us an initial micro F1 score of 57% (cp. fig. 3), using a default threshold of 0.2 likewise for all classes. After threshold optimisation as described in section 1, we achieved our best result of 60% micro F1. In short, our final architecture can be described as depicted below.

```
conv1d_1:
32 kernels, (3,3) kernel size, (1,1) stride, ReLU activation, batchnorm, (2,2) max pooling
conv1d_2:
64 kernels, (3,3) kernel size, (1,1) stride, ReLU activation, batchnorm, (2,2) max pooling
conv1d_3:
128 kernels, (3,3) kernel size, (1,1) stride, ReLU activation, batchnorm, (2,2) max pooling
conv1d_4:
256 kernels, (3,3) kernel size, (1,1) stride, ReLU activation, batchnorm
dense_1:
256 units, 0.5 dropout, ReLU activation, truncated_normal initialisation
dense_2:
14 units, sigmoid activation
___
adam optimiser, binary_crossentropy loss, 128 batch size
```

4 Conclusion

Structuring the project into both automated and manual solution stages, we were able to achieve good classification results in a comparatively short amount of time while collecting enough metadata to infer suitable hyperparameter settings for further manual optimisation. This approach is especially useful when there is enough computational power at hand as was the case with GPU support in *Google Colaboratory*[2]. With regard to computational efficiency, an additional measure to improve our pipeline could be to implement bayesian optimisation as strategic search for promising hyperparameter combinations considering search history and its probabilistic evaluation.

We faced a minor but important problem that was related to the F1 metric being difficult to implement as a callback in Keras and scikit-learn. For this reason, although we were able to monitor the F1 score

[2] More information on Google Colaboratory at https://colab.research.google.com/

with each epoch, the evaluation and model ranking in random search was based on accuracy – which in our case does not make a lot of sense. We are sure that there is an easy fix, but unfortunately ran out of time to resolve this issue. Still, the best architecture based on accuracy gave us decent results with F1.

As we started off with random search, our generated baseline model achieved a micro F1 score of 43% on our validation data and, eventually, manual optimisation led to 60%. This goes to show that automated optimisation as the driving force behind our classification pipeline arguably saved us some time and effort. Judging by our evaluation results, we are confident that our final model will generalise well with the test dataset.

Appendix

Explanation: we computed the F1 score every 5 epochs, so a value of 2 on the horizontal axis, for instance, stands for 10 training epochs.

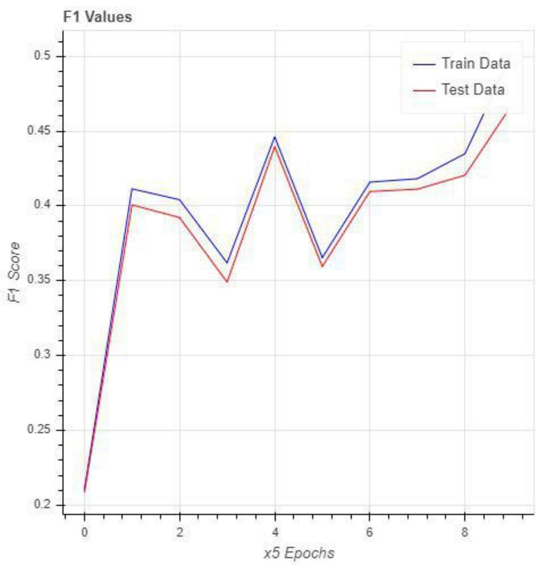

Fig 1: 1st iteration F1 performance

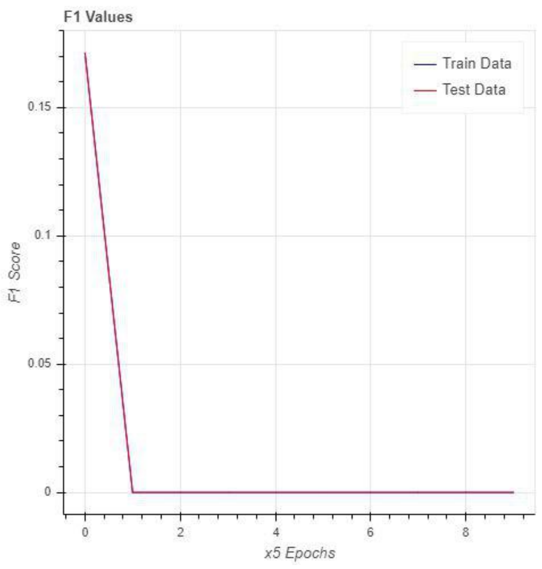

Fig 2: 2nd iteration F1 performance

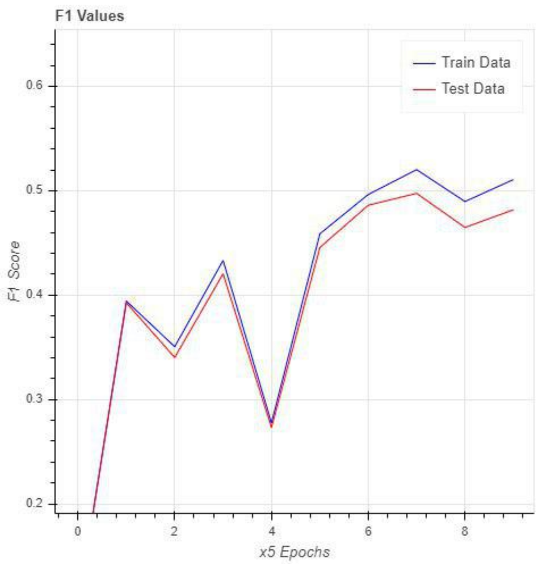

Fig 3: 3rd iteration F1 performance