What Can You See?

Manuel Fideles[1]

University of Coimbra

Abstract. In this paper, we describe an approach to distinguish between images containing six subjects - *buildings, forest, glacier, mountain, sea,* and *street* -, using different types of Machine Learning algorithms. To accomplish this task, about 17000 50*50 pixels RGB images were used as a dataset. Results obtained suggest that a Convolutional Neural Network is better at detecting images, achieving an accuracy of 0.84.

Keywords: Deepfakes · Convolutional Neural Network · Multi-layer Perceptron

1 Introduction

The present paper constitutes a benchmark on image classification, using a shortened version of the Intel Image Classification dataset. A set of algorithms are evaluated on their ability to detect what subject it represents. The paper is structured as follows: section 2 enumerates the related work researched, section 3 describes the experimental component carried out and the results obtained, section 4 elaborates upon these results and section 5 provides an overview of the project along with proposed future work.

2 Related Work

In [1], the authors propose a CNN architecture for image classification applied to the detection of interstitial lung disease (ILD) from lung image patches. Their results suggest that dropout is key to preventing overfitting.

3 Experimental Work

A dataset containing 17051 images was made available for experimentation purposes. This dataset is divided into two:

– A training dataset, which was used for training and validating the models. It contains 12000 labeled RGB images of size 150-by-150px and is balanced, meaning it features 2000 examples of each of the 6 target classes. A *.npy* version of these images was also made available, with dimensions 50*50.

– A test dataset, containing 5051 unlabeled images, which will be used for submitting models to the Kaggle competition[1], made available by the faculty. A *.npy* version of these images was also made available, with dimensions 50*50.

Still regarding the data, it is relevant to note that data augmentation was performed on the training data, to ensure data diversity. This expands our dataset - which is relatively small for this task - and helps prevent overfitting. [3]

All model parametrization was supported by the Sweeps API, developed by Weights&Biases[2], which automates hyperparameter tuning and provides a quality dashboard to clearly visualize the effects of certain parameters. This tool allowed us to obtain a deeper insight into the necessary changes in layer sizes and topology tweaks the model needed. Furthermore, all the model's training, validation, and testing were carried out in Google Colab[3], which sped up the model development considerably versus running said processes in localhost.

It is important to note that due to limited storage space on Google Colab, we opted to use only the *.npy* versions of the training and testing datasets. We were aware this would probably hamper the performance of the model (*vs.* using the original images as-is) because we are effectively feeding less information to the model - a Numpy representation of an image is 50-by-50px, while an untouched image is 150-by-150px. Despite this, we think the dataset-upload-time-to-performance-loss ratio pays off, as we successfully kept upload times relatively low.

3.1 Evaluation

Regarding the evaluation, a Train-Validation-Test split was performed on the training dataset. After finding the best hyperparameters for each model, these were tested on the test set, and the best-scoring one was tested on the test dataset and later submitted to the competition. The parameterization process consisted of feeding a relatively large parameter space to the Sweeps API and using random search to reduce the size of said parameter space. Then, with a smaller parameter space, we leveraged grid search to exhaust all possible parameter combinations. We found this approach to be an advantageous trade-off between compute/train time and quality of results. The models' performances were evaluated using Accuracy, Area Under ROC Curve (AUC), Precision, and Recall.

3.2 Experimentation

The following sections detail the experimentation process, describing each of the algorithms and their results. After experimentation, two final models (the best model of each one of MLP and CNN) were chosen, and the best-scoring one was selected for submission to the competition.

[1] Kaggle Competition - https://www.kaggle.com/competitions/aml-2023-project2
[2] Sweeps by W&B - https://docs.wandb.ai/guides/sweeps
[3] Google Colab - https://colab.research.google.com/

CNNs

Regarding the CNNs, we started on a basic architecture that consisted of two convolutional layers:

- $16@3 \times 3$ followed by a max pooling layer 2×2
- $32@5 \times 5$ followed by a max pooling layer 2×2

and one fully connected feed-forward neural network (dense layers) with 64 output neurons and a final layer with 6 output neurons - one for each class. We appropriately named this simple architecture SimpleNetV1, and the architectures that proceed also follow this naming convention. In all layers of this architecture and subsequent ones mentioned here, the ReLU activation function is activated, whilst the last layer applies the Softmax function. The convolutional layer uses ”same” padding, and for that matter, all subsequent architectures that will be discussed use ”same” padding. It is also worth mentioning that all architectures were tested on 20 epochs and with a batch size of 512. The loss was measured with *categorical crossentropy.* In parametrization, the optimizer was either set to *sgd* or *adam.*

Table 1. Most relevant CNN architectures

Arch	Optimizer	Learning Rate	# of Params	Loss	Accuracy	AUC	Precision	Recall
Training								
SimpleNetV1	sgd	0.0001	244,036	0.6225	0.7743	0.9601	0.8198	0.7160
SimpleNetV2	adam	0.002	419,674	0.6236	0.7746	0.9595	0.8067	0.7252
SimpleNetV3	adam	0.001	2,627,814	0.6503	0.7777	0.9562	0.8205	0.7215
Validation								
SimpleNetV1	sgd	0.0001	244,036	0.6370	0.7694	0.9585	0.8136	0.7236
SimpleNetV2	adam	0.002	419,674	0.6135	0.7725	0.9612	0.8106	0.7406
SimpleNetV3	adam	0.001	2,627,814	0.5220	0.8114	0.9715	0.8439	0.7856

Figure 1 illustrates the aforementioned architectures. The presented data in Table 1 suggest the presence of very slight overfitting in the first and second architecture, as their performance is marginally better in training *vs.* validation, meaning its fitting well with the data. In the training phase, the third architecture outperforms the others by a small margin, but that margin increases by 4% in the validation set, which suggests that it generalizes better than the rest. The models' performances on the validation set suggest that the randomness introduced by the dropout layers is important to prevent overfitting - in previous iterations of SimpleNetV3, the presence of overfitting dictated the incremental addition of dropout layers. Additionally, stacking convolutional layers together seems to improve the detection of high-level features, as seen in SimpleNetV3's performance in the validation. This implies that this extra convolutional layer allows the model to better generalize its learning. Furthermore, it seems obvious

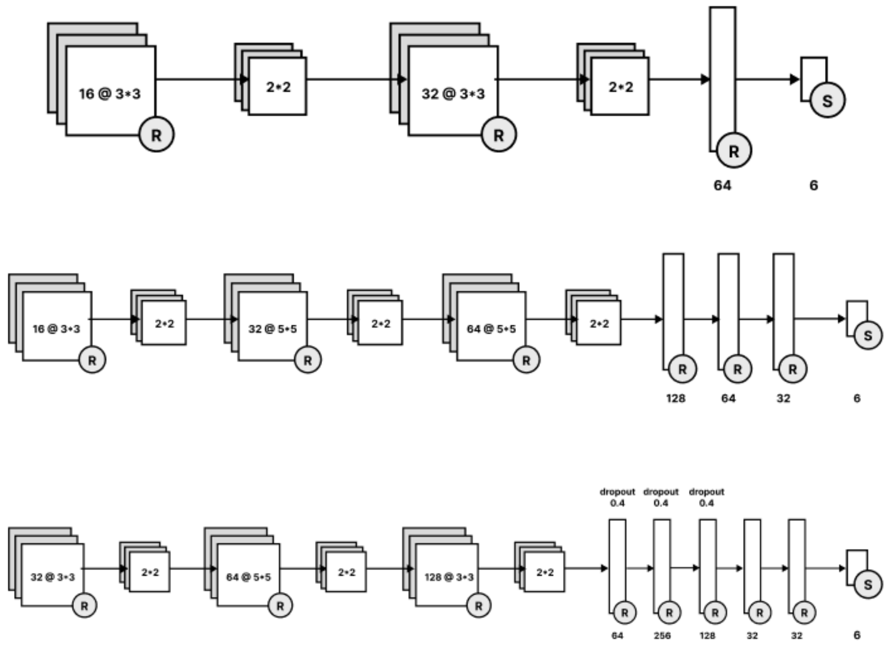

Fig. 1. SimpleNet V1, V2, and V3 topologies - V1 is a simple CNN, V2 adds dense layers to increase the model's complexity. V3 builds on that, and adds dropout to help prevent overfitting.

to conclude that by increasing the size and number of the dense layers, we can obtain superior performance.

As an alternate approach, we decided to leverage transfer learning by using pre-trained models. We had two different development "paths" in mind:

- Using a given pre-trained model as is, which required the least amount of parametrization and implementation effort, but may lack in performance for the task at hand,
- Using a given pre-trained model, but removing the top layer and adding task-specific layers, so that hopefully we can enhance the off-the-shelf performance given by the generic model.

Since the former was not a viable option, due to implementation issues, we opted solely for the latter. With that in mind, the pre-trained models that were tested are listed below:

1. MobileNetV2 - 3.4M trainable parameters
2. VGG19 - 20M trainable parameters
3. ResNet50 - 23M trainable parameters

These models are all available in the *applications* module of the Keras package in Tensorflow[4]. Furthermore, we used the weights inferred from the *ImageNet* dataset, a large visual database designed for use in visual object recognition software research.

For this transfer learning approach, we tested each model with the following topology:

- Base Model + Dense layer with ReLU activation + Dense layer with ReLU activation + Dense layer with Softmax activation

Table 2 lists the results for the best-performing version - i.e. the one with the lowest validation loss - of each one of the previously mentioned pre-trained models, found in parametrization. Columns *fc1_size* and *fc2_size* respectively correspond to the number of output neurons on the first and second dense layers added after the base model (apart from the dense layer with Softmax activation).

The results presented suggest something we had already established: in this specific task, more parameters equals a better performance since a more complex model is able to infer and extrapolate more information. After subjecting all the pre-trained models to the same experimentation process, and failing to surpass the 82%-accuracy mark, we decided to fine-tune the best-performing model to see how big the performance gain would be. Having found the hyperparameters that yielded the highest accuracy and lowest loss (described in 2), we re-trained ResNet50 - i.e. the best-performing model of those tested -, for 60 epochs. Afterward, the fine-tuning phase was comprised of unfreezing[5] the 16 topmost layers

[4] Pretrained modules in Tensorflow - https://www.tensorflow.org/api_docs/python/tf/keras/applications

[5] *Freezing* - Making a model's layer untrainable. Similarly *unfreezing* a layer refers to the process of making it trainable. This is standard procedure when using pre-trained models and/or fine-tuning them.[3]

Table 2. Pre-trained CNNs

Architecture	Optimizer	Learning Rate	*fc1_size*	*fc2_size*	Loss	Accuracy	AUC	Precision	Recall
Training									
MobileNetV2	Adam	0.003	32	64	0.6487	0.7705	0.9569	0.8183	0.7093
VGG19	Adam	0.001	32	128	0.6561	0.7574	0.9559	0.8071	0.7019
ResNet50	Adam	0.001	128	64	0.4444	0.8351	0.9789	0.8576	0.8101
Validation									
MobileNetV2	Adam	0.003	32	128	0.6792	0.7594	0.9534	0.8035	0.7108
VGG19	Adam	0.001	32	128	0.6302	0.7806	0.9594	0.8203	0.7378
ResNet50	Adam	0.001	128	64	0.5019	0.8222	0.9735	0.8429	0.7942

of the model and training it for an additional 30 epochs. The results are depicted in Fig. 2. It is important to note that, following Tensorflow's documentation, the learning rate was divided by 10[6].

The results presented in Fig. 2 suggest that not much value was extracted from the fine-tuning process. After 60 epochs of training, the model wasn't able to further converge in a way that allowed it to surpass the 82% accuracy mark. In fact, validation loss becomes pretty constant after epoch #40. After the start of the fine-tuning process, the increase in overfitting is almost immediate, even after reducing the optimizer's learning rate. Furthermore, the training/validation time increased by a factor of 5[7], and, in exchange, we obtained a 2% increase in accuracy.

MLPs

For the MLPs, we opted to use the same dense layers as the ones used in the CNNs - since those were directly correlated to relatively acceptable performance. As such, the architectures of the MLPs we experimented with are basically described in Fig. 1, but without the convolutional and pooling layers. The MLPs tested follow a similar naming convention, except since these models are simpler, we named them SimplerNet V1, V2, and V3. The results of training and testing are presented in Table 3.

4 Discussion

After finding the hyperparameters for each type of model, these were re-trained on the training set, evaluated on the test set, and yielded the results displayed in Table 4. The results suggest that CNNs are better at detecting the subject matter of a picture in this dataset. As expected, the MLPs performed worse than the more complex models, both "manually" developed and pre-trained.

[6] According to [3], since we are training a much larger model and want to readapt the pre-trained weights, it is important to use a lower learning rate at this stage. Otherwise, the model could overfit very quickly.

[7] Assuming constant time for each epoch: 60 training epochs / 30 fine-tuning epochs = 90 / 20 benchmarking epochs = 4.5

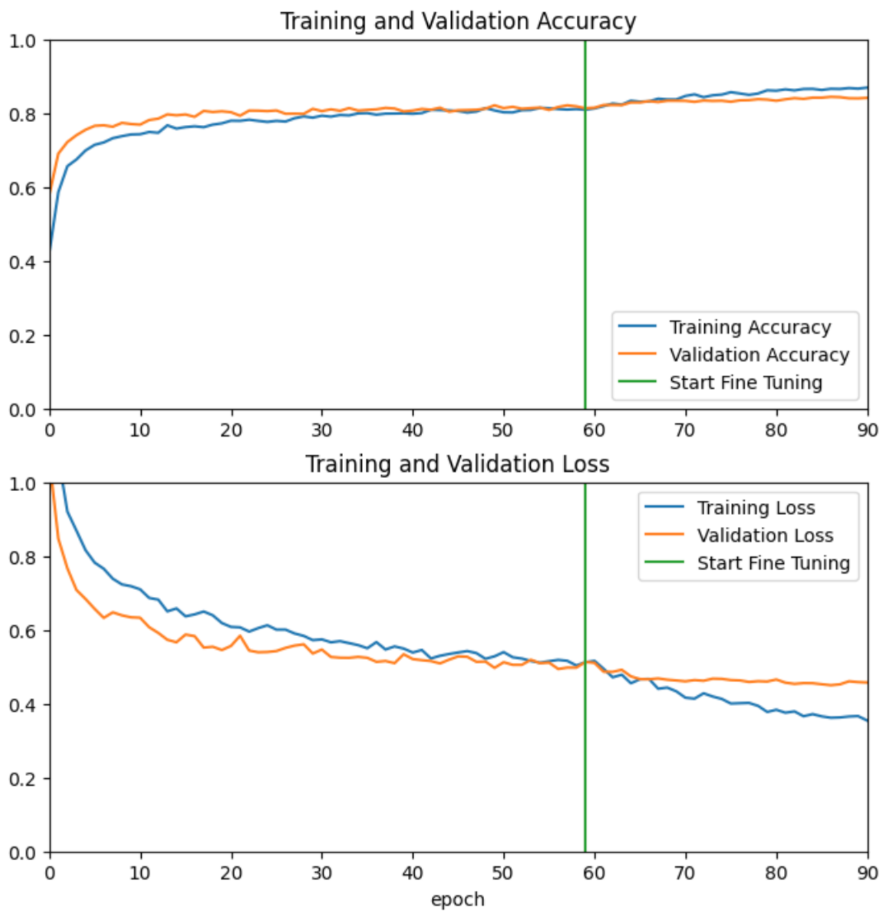

Fig. 2. ResNet50 fine-tuning: The data depicted to the left of the green line shows the training/validation process with a totally frozen base model (plus the two previously discussed dense layers). To the right of the green line, the topmost 16 layers of the said model were unfrozen to enable fine-tuning. As we can see, some overfitting is present, but we have effectively broken the 82% accuracy mark

Table 3. Most relevant MLP architectures

Arch	Optimizer	Learning Rate	# of Params	Loss	Accuracy	AUC	Precision	Recall
Training								
SimplerNetV1	sgd	0.0001	375,356	5.8666	0.3465	0.6704	0.5028	0.1787
SimplerNetV2	adam	0.002	1,134,530	1.7103	0.2725	0.6347	0.7770	0.0892
SimplerNetV3	adam	0.001	4,015,270		3.8271	0.2440	0.5822	0.2822
0.1473								
Validation								
SimplerNetV1	sgd	0.0001	375,356	4.8169	0.3142	0.6556	0.4793	0.1572
SimplerNetV2	adam	0.002	1,134,530	1.7372	0.2739	0.6289	0.7690	0.0842
SimplerNetV3	adam	0.001	4,015,270	1.7375	0.2869	0.6440	0.5386	0.0775

Table 4. Performance on test set

Model	Accuracy	AUC	Precision	Recall
MLP	0.3265	0.6256	0.4593	0.1372
SimpleNetV3	0.8177	0.9562	0.8205	0.7615
ResNet50	0.8402	0.9789	0.0.8298	0.8101

5 Conclusions

Regarding this paper, we believe we've met some conclusions about the models studied, and about how they work.

In general, a more complex model - i.e. one with a higher number of parameters -, yields superior performance, but this was not always true, and the marginal performance difference between MobileNetV2 and VGG19 illustrates that very clearly. The latter has around 6 times more parameters than the former, yet the increase in accuracy is minimal.

The performance ceiling we hit during our experimentation can also be a consequence of only using the data present in the *.npy* files. In our opinion, this may have been the biggest constraint on all the models' performances, as is explained in 3.

The use of transfer learning allowed us to iterate faster on a more relevant solution for the problem at hand. "Manually" developing a CNN is hard and time-consuming. Nevertheless, both approaches obtained similar results when compared in the same experimentation environment, which also corroborates the data problem mentioned above. Still on transfer learning, unfreezing more layers of the base model might have been beneficial, and yielded better overall performance, although it would increase the training/validation time considerably. Testing this hypothesis was made impossible by the biggest constraint of all, time, as we started experimenting with transfer learning relatively close to this project's deadline.

Furthermore, we could have been more thorough when designing the experimental topologies. For instance, experimenting with two or more convolutional layers in a row, without any pooling between them. Since pooling is a destructive operation, stacking several convolutional layers in a row could have been beneficial for high-level feature extraction.

References

1. Li, Q., Cai, W., Wang, X., Zhou, Y., Feng, D. D., & Chen, M. (2014, December). Medical image classification with convolutional neural network. In 2014 13th international conference on control automation robotics & vision (ICARCV) (pp. 844-848). IEEE.
2. Tensorflow Core. "Transfer Learning and Fine-Tuning" TensorFlow, Google, https://www.tensorflow.org/tutorials/images/transfer_learning#use_data_augmentation. Accessed 7 May 2023.
3. Tensorflow Core. "Transfer Learning and Fine-Tuning" TensorFlow, Google, www.tensorflow.org/tutorials/images/transfer_learning#fine_tuning. Accessed 7 May 2023.