Cojocariu Sebastian
507 - AI

Challenge description

Introduction

After analysing the dataset provided, several convolutional neural networks were chosen to tackle the challenge:

- Simple Linear (non-convolutional): a custom linear model that uses as features the concatenation of patches from the initial image (using AveragePooling2D and MaxPooling2D with various filters). These are then forwarded to a sequence of Fully Connected layers to solve the task.
- VGG variant: sequential blocks of Conv2D + MaxPooling.
- InceptionNet variant: the output of each block consists of stacked filters coming from different Conv2D-sized kernels. Additionally, it stacks filters coming from MaxPooling / AveragePooling too.
- ResNet variant: blocks that contain, apart from the residual/skip connection, BatchNormalization and Conv2D 1 x 1 that acts as a complexity reduction layer before applying the computational expensive Conv2D k x k. If the number of filters do not match, it applied a Conv2D 1 x 1 to the initial input.
- DenseNet variant: sequential blocks of convolutions (similar to *ResNet* blocks), with the difference that instead of addition, it concatenates the output of the previous layers before passing the result forward. These blocks usually consist of multiple convolutional layers blocks, followed by a transition layer (that shrinks both the dimension and the number of filters).
- MobileNetV2 variant: similar to *ResNet* blocks, but uses a special type of convolution called DepthwiseSeparable to reduce the number of operations, as well as the number of learnable parameters. Because of the limited time available for this challenge, this architecture was not tested (although its implementation exists in the provided source code). From prior experience this is a great architecture to work with, especially in constraint/limited environments like phones, tablets, and any embedded devices.

Observations:

- Squeeze and Excitation Layers were added to reweight features' importance and to offer a boost performance.
- To improve the overall prediction rate as well as overall robustness, we trained a *meta-learner* over the predictions of multiple models. For this task, *XGBoost* and *Majority Vote* were chosen as suitable candidates. Surprisingly, experiments showed that the latter achieved a better performance.

Cojocariu Sebastian
507 - AI

Data Preprocessing

Looking at the corpus, we understood that the class is determined by the number of horizontal lines. In order to prepare the data in the suitable format for training, several techniques were applied:

- Normalization / Standardization: dividing the input by 255 to fit into [0, 1]. The reason for applying this is to better guide the training and speed up the process. Normalization between [-1, 1] was also used, but the results were poorer and so this approach was rapidly discarded.
- Adaptive thresholding to convert the image into black and white (to reduce noise).
- Manually designed filters from Computer Vision Field to reduce the noise (erosion, dilation, opening and closing method). This approach was not really successful though (prior knowledge about the corpus was needed).
- Extracted features from patches using AveragePooling2D and MaxPooling2D operations.
- Data Augmentation: applied vertical + horizontal flips, as well as rotation by 180 degrees (other rotations would cause the horizontal lines to become vertical, which would have not been a correct augmentation technique). ImageDataGenerator from TensorFlow was also tried, but with no improvement whatsoever.
- Adding Gaussian Noise as part of the model (to avoid overfitting and to offer a wider range of slightly modified samples during training).
- **Note:** Several combinations of these techniques were used during the development of the models.

For each training, the available corpus was split into two parts, randomly: train (80%) and validation (20%). Additionally, the augmentation step is done after the split, and not before (such that no flipped image should be in both train and validation datasets). The metric watched was the model's accuracy against the validation set.

Hyperparameters

Architecture

Taking into consideration the reduced dimension of the images (128 x 55), we are forced to use at most 3-4 pooling layers or strided convolutions. When compared directly, AveragePooling layer offer better performance than MaxPooling (a reason for this is because it can better distinguish the pixels that compose lines). Even so, experimental results show that using strided convolutions instead of pooling layers offers a substantial increase in the prediction rate, and thus they were preferred.

As for the kernel sizes used in Conv2D, we saw an increase in performance when using horizontal kernels at the beginning of the network, imposing a new constraint to fine-tune. The choices were among [3, 5] x [3, 5, 7]. As per the number of convolutional blocks used (we define by block the operations between the strided convolutions), we saw that four such blocks produced the most consistent results: a higher number would just slow the training, while less blocks would obtain poorer results against both train and validation datasets. Usually, the main differences between architectures consists of how these blocks are implemented.

Normally in any convolutional network, the number of filters is progressively increased as the height and weight decrease (because of downsampling operations). The range of testing was between 64 and 1024.

The output of the last convolutional block is then passed to a GlobalAveragePooling (AveragePooling or MaxPooling followed by Flatten were also tested), then followed by a sequence of Fully Connected Layers. Additionally, Dropout layers are used at this level to reduce overfitting.

Optimizers

- SGD: learning rate selected in the range of [5e-4, 1e-6]. An immediate problem is that fine-tuning the learning rate is very difficult, and thus we did not focus much on this optimiser (although it is known that, when correctly fine-tuned, can produce better results than other stable algorithms).
- Adam: learning rate selected in the range [1e-3, 1e-4] with default betas.
- AdamW: improves on Adam by penalising large weights. Learning rate selected in the range [1e-3, 1e-4]: bigger learning rates would case spikes in validation loss, while smaller ones would make the weights not update. Also, weight decay in the range [5e-4, 1e-6] was added.
- Note: After each epoch the learning rate is multiplied by a constant factor using a Learning Rate Scheduler (usually in the range 0.98-0.995, depending on the number of epochs to train). There are also two limits: for minimum and maximum; if LR is smaller or higher than them, it is clipped. The code uses a minimum of 1e-4 and a maximum of 1e-1.

Loss Functions

The models were trained having as task either classification or regression. For the latter, an accuracy function was implemented.

1) *Regression*

- Mean Absolute Error (MAE): the derivate is constant no matter the error. Good when the values to predict are quite similar.
- Mean Squared Error (MSE): it penalises errors larger than 1 more. Good when the values to predict are large/sparse.
- Huber Loss*:* a combination of both worlds.

- Margin Loss: max(0, (label – prediction) ^ 2 – threshold ^ 2). When setting threshold to 0.5, it does not penalise predictions that are within 0.5 distance from the true label. The threshold needs to be fine-tuned.
- Among all these, the best results were achieved using **MAE Loss.** An intuition for this is due to the fact that the values to predict are relatively small and discrete (1 to 5).

2) *Classification*

- Categorical Cross Entropy: from information theory. For this to work, BatchNormalization layers must be enabled (otherwise it the loss does not substantially change across each epoch).

Callbacks

- Learning Rate Scheduler: multiply the learning rate after each epoch with a constant factor (usually between 0.98-0.995). Dependent on the number of training epochs set. There are also two limits: for minimum and maximum; if LR is smaller or higher than them, it is clipped. The code uses a minimum of 1e-4 and a maximum of 1e-1.
- Early Stopping: exit the training once a specific metric is no longer improving (in our use-case, accuracy on the validation set).
- Model Checkpoint: used after each epoch to save the best model so far with respect to the metric watched.

Regularization techniques

- **Dropout** after each Dense/Fully-Connected Layer: the most suitable one is in the range [0.4, 0.7] (smaller values creates an overfitting regime, while larger ones make the model doesn't train at all - underfitting).
- **GaussianNoise** applied to the original input, before the convolutional blocks: uses a standard deviation of around 0.01. This value was used taking into consideration that the pixels are divided by 255. Thus, 0.01 equals around +/- 3 in the original image. Dependent on the preprocessing step.
- **Weight Decay** for **Adam Optimizer** in the range of [5e-4, 1e-6].
- **Augmentation** of the initial dataset with the corresponding flipped versions (horizontal + vertical) + 180 degrees rotations. ImageDataGenerator from TensorFlow was also tried, but with no improvement whatsoever.

Configuration file

- Type: train or cross_validation.
- Model_type: random / linear / vgg / inception / resnet / densenet.

- Loss_name: the name of the loss function: mae / mse / margin_loss / categorical_crossentropy
- Epochs: the number of epochs to train.
- Batch size: chosen in the interval [64, 128]. Most consistent results with 96. For larger values, learning rate should be increased, while for smaller values, it should be decreased.
- Enable_batchnorm: true / false.
- Task_type: classification / regression.
- Learning_rate: initial learning rate.
- Learning_rate_decay: used in the Learning Rate Scheduler to update the LR.
- Weight_decay: used to penalize large weights (to counter overfitting).

General observations

- Pixel Normalization in [0, 1] instead of [-1, 1]. The latter generated spikes in the validation loss.
- Even though the task itself seems to suggest a classification approach, regression was preferred due to better overall performance. Additionally, experimental results showed that disabling the BN layers actually increased the prediction rate. An intuition why the regression is better is because of the nature of the task (which counts the number of horizontal lines and thus it imposes a relative ordering – something that classification miss to spot).
- In order for classification to work, BN must be enabled (otherwise the model does not successfully train). By using this approach, we observed that the validation loss tends to generate spikes from one epoch to another. This effect was mildly overcome by adding BN after the activation, and not before it in the convolutional blocks (as many papers suggest), although the spikes still happen (but rarely).

Hyperparameter Tunning

In order to design the best model for our task, we also recorded how modifying some hyperparameters affected the overall performance of each architecture and summarized these findings into the table below:

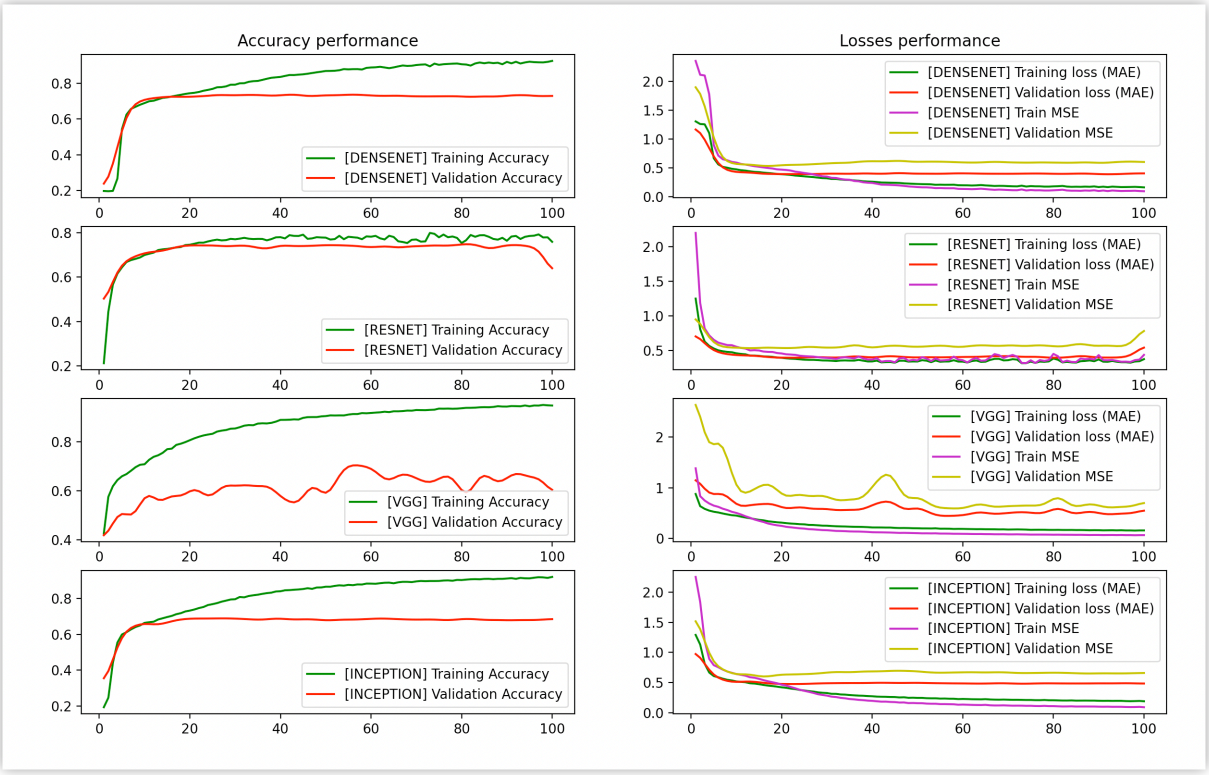
Model	# Conv Blocks	Batch Size	LR	MAE train	MAE val	MSE train	MSE val	ACC train	ACC val
VGG	3	64	1E-04	0.21741	0.45746	0.15832	0.64071	0.79189	0.68595
VGG	4	96	5E-04	0.16057	**0.42919**	0.09492	**0.59286**	**0.80781**	**0.70282**
VGG	5	128	1E-03	0.16255	0.42975	0.08616	0.61970	0.81294	0.69811
INCEPTION	3	64	1E-04	0.21958	0.49981	0.18189	0.75016	0.83012	0.70197

INCEPTION	4	96	5E-04	0.24579	**0.49205**	0.18575	**0.71562**	**0.80985**	**0.71146**
INCEPTION	5	128	1E-03	0.22051	0.50116	0.19182	0.73551	0.79713	0.70911
RESNET	3	64	1E-04	0.25588	0.37671	0.30955	0.52759	0.82912	0.76021
RESNET	4	96	5E-04	0.27602	**0.36035**	0.30531	**0.51802**	**0.83019**	**0.76198**
RESNET	5	128	1E-03	0.27517	0.36132	0.28571	0.51401	0.83951	0.74673
DENSENET	3	64	1E-04	0.27186	0.34338	0.34775	0.49387	0.84051	0.75844
DENSENET	4	96	5E-04	0.29623	**0.33987**	0.35891	**0.48298**	**0.85192**	**0.77837**
DENSENET	5	128	1E-03	0.26672	0.34501	0.33086	0.49557	0.84373	0.76429

Note: For VGG, BN was used, while for the others, it was disabled (for performance reasons).

So far, it seems like all the architectures prefer 4 convolutional blocks instead of 3 or 5, with a batch size of 96 and a learning rate of 5e-4.

Below are the plots related to the metrics watched (MAE, MSE and Accuracy) during the training of each type of architecture. As we can see, all the models expect for VGG manage to enter an optima around epoch 10 (validation accuracy), while the training accuracy keeps increasing. VGG on the other hand do show an ascending trend, but with spikes. The losses remain slightly stable across the training period (with a slight improvement for the train dataset, which is expected).



Cojocariu Sebastian
507 - AI

Experimental Results

To design the most suitable architecture for the task at hand, a cross validation method with 5 folds was implemented. The metrics watched were Mean Absolute Error (**MAE**) and Mean Squared Error (**MSE**) and Accuracy. The experimental results can be seen in the tables below for the foregoing metrics. Each architecture except for *DenseNet* was trained for 75 epochs per fold, while DenseNet only against 50 epochs (because of the internal complexity of the architecture).

MAE

Model Type	**Fold-1**	**Fold-2**	**Fold-3**	**Fold-4**	**Fold-5**	**Mean**	**Standard Deviation**
Random	1.59967	1.61258	1.56967	1.59096	1.63129	1.6008	0.0206
SimpleLinear	1.01271	1.27512	1.11856	1.08192	1.38619	1.1749	0.1362
VGG	0.58715	0.77501	0.47670	0.65553	0.79811	0.6585	0.1193
Inception	0.48487	0.55436	0.54206	0.49614	0.54277	0.5240	0.0279
ResNet	0.41484	0.45558	0.51762	0.46472	0.45185	0.4609	0.0330
DenseNet	0.37466	0.44714	0.43745	0.40317	0.47511	**0.4275**	0.0350

MSE

Model Type	**Fold-1**	**Fold-2**	**Fold-3**	**Fold-4**	**Fold-5**	**Mean**	**Standard Deviation**
Random	3.95838	4.11451	3.86129	3.9664	4.18612	4.0173	0.1169
SimpleLinear	1.61829	1.71172	1.68257	1.61293	1.59875	1.6448	0.0441
VGG	0.85089	1.12911	0.64534	1.06590	1.07848	0.9538	0.1814
Inception	0.64848	0.80668	0.82516	0.68376	0.78604	0.7500	0.0705
ResNet	0.57778	0.68330	0.83513	0.67618	0.66686	0.6878	0.0829
DenseNet	0.51621	0.65012	0.52812	0.58421	0.51080	**0.5578**	0.0529

Cojocariu Sebastian
507 - AI

Accuracy

Model Type	Fold-1	Fold-2	Fold-3	Fold-4	Fold-5	Mean	Standard Deviation
Random	0.21012	0.20761	0.22016	0.20176	0.19972	0.2078	0.0072
SimpleLinear	0.27017	0.29182	0.28754	0.29712	0.28564	0.2864	0.0090
VGG	0.65671	0.67095	0.69764	0.68152	0.64981	0.6713	0.0171
Inception	0.66841	0.69518	0.70814	0.68591	0.68105	0.6877	0.0133
ResNet	0.70791	0.71517	0.69581	0.71952	0.72597	0.7128	0.0103
DenseNet	0.73153	0.70783	0.72486	0.73632	0.72918	**0.7259**	0.0097

For the Ensemble approach, we trained multiple models with different architectures and hyperparameters, each against a different validation dataset, using random seeds to split the initial corpus (to make the meta-learner more robust to overfitting).

The best model for each architecture, as well as the ensemble's performance against the validation dataset are summarized below.

Model Type	MAE Train	MAE Val	MSE Train	MSE Val	Accuracy
Random	-	-	-	-	21,2%
SimpleLinear	0.78901	1.08192	0.98947	1.35911	29,7%
VGG	0.24579	0.49205	0.18575	0.71562	70,2%
Inception	0.16055	0.42915	0.09491	0.59289	71,1%
ResNet	0.27602	0.36035	0.30531	0.51800	76,1%
DenseNet	0.29623	**0.33987**	0.35891	**0.48298**	77,8%
Ensemble	-	-	-	-	**79,8%**

Conclusions

The best single architecture tested is DenseNet, with an accuracy of up to 77,8%, while the best overall model is the ensemble one using as features the predictions of several trained models (VGG, InceptionNet, ResNet and DenseNet), with an increase of 2%. Even so, the ensemble model comes with a considerable overhead that should be considered by the engineer designing models to run on real tasks, especially on embedded devices such as phones with limited resources available.