Assignment 2: Convolutional Neural Networks and AutoML

Max van den Hoven 1750461   Dean Bakens 1026726   Alwin Verbeeten 1223096   Johan de Langen 0995417

9 June 2023

Introduction

Wefabricate, a Dutch manufacturing company, aims to build a fully autonomous factory that enables affordable make-to-order for single products. As part of this plan, Wefabricate is investigating applications of deep learning to automate part of the visual inspection process of produced parts. More specifically, they want to identify defective products using images taken along the production line. The goal of this assignment is therefore to investigate how deep learning techniques can be used to build a binary image classifier for defective products, and how automatic machine learning techniques can be used to tune the hyperparameters of such a model.

For the purposes of this assignment, Wefabricate has collected a balanced dataset of 170 labeled images. The images are taken by hand and show a front plate designed for an industrial plug. This front plate is manufactured using injection molding, which can introduce various defects such as holes, scratches, burns, or missing chunks. To accurately classify images as defective or not, we train an image classifier to recognize common defects in the images, according to the labels assigned by Wefabricate.

The remainder of this report is structured as follows. First, we motivate why convolutional neural networks are a good choice for this task. Then, we discuss our initial model architecture and its performance on the dataset. Following this, we discuss which hyperparameters can be tuned, and which two algorithms we used to optimize these hyperparameters. Finally, we compare the model found by hyperparameter optimization to our original model and reflect on the entire process.

Convolutional Neural Network Description

In this section we motivate why convolutional neural networks (CNNs) are a good choice for solving the proposed image classification task. An important consideration, which quickly becomes apparent when investigating the provided images, is that defects can occur anywhere on the product. As such, we would like our model to be invariant to translations, meaning that the appearance of a defect will always cause our model to classify a product as defective, regardless of where the defect occurs in the image.

Considering the aforementioned symmetries in the data, a fully connected network would likely be ill-suited for this task, as it would have to learn to recognize different types of defects at every part of the image. While technically possible due to the universal approximation theorem, this would result in poor data and parameter efficiency, which is undesirable given the relatively small size of the provided dataset.

Instead, we choose to build our classifier using a CNN architecture, which consists of alternating convolutional and pooling layers. Convolutional layers are inherently equivariant to translation, as learnable filters are applied to the image at every position. As a result, defect patterns can be recognized using relatively few parameters, improving the efficiency of the model. Pooling layers are used to downsample intermediate representations, which reduces the computational complexity of the model and acts as a form of regularization. Furthermore, pooling introduces invariance to small shifts in the input, which, when applied multiple times throughout a CNN, ensures that the whole model is approximately invariant to translation.

Model Architecture Before Optimization

In this section we describe our proposed model architecture. We train this model using a set of default hyperparameters and show its performance on the train, validation, and test sets[1]. Using the provided support code, we first resize the input images to a resolution of 60×30, which results in an input tensor of shape $60 \times 30 \times 3$, as we are working with RGB images.

In our model, the input is fed through a number of convolutional blocks comprising a convolutional layer, an activation function, and a max-pooling layer. The convolutional layer in the first block uses NUM_BASE_CHANNELS filters to increase the number of channels on the image, where NUM_BASE_CHANNELS is a hyperparameter. Subsequent convolutional blocks scale the number of channels by a factor of 2, such that, e.g., the second convolutional block outputs $2\times$NUM_BASE_CHANNELS channels.

The stride of all convolutional layers is set to one and the padding is set such that the spatial dimensions are preserved[2]. All pooling layers have a stride of two, such that the spatial dimensions are halved. The kernel sizes of all convolutional and pooling layers are controlled using the hyperparameters KERNEL_SIZE_CONV and KERNEL_SIZE_POOL respectively. The number of convolutional blocks in the model is controlled by the hyperparameter NUM_CONV_BLOCKS.

After passing through all convolutional blocks, the resulting tensor is flattened and passed through a fully connected layer with one output neuron and sigmoid activation. This clamps the output value to the range $(0, 1)$, where a value of 0 represents a defective product and a value of 1 represents a non-defective product. Finally, we use binary cross-entropy loss to evaluate the model during training.

The motivation behind this CNN architecture is that it is relatively simple, in accordance with the relatively small image resolution, but still flexible enough to be defined by more than five hyperparameters. The domains and default values for all hyperparameters are shown in Table 1. It is important to note that not all possible values have been included in the domains, in order to reduce the complexity of the search space for this assignment.

Hyperparameter	Default value	Domain
NUM_BASE_CHANNELS	16	$\{4, 8, 16, 32\}$
NUM_CONV_BLOCKS	2	$\{2, 3\}$
KERNEL_SIZE_CONV	3	$\{3, 5, 7\}$
KERNEL_SIZE_POOL	2	$\{2, 3\}$
ACTIVATION	ReLU	{TanH, ReLU, LeakyReLU}
OPTIMIZER	Adam	{SGD, RMSProp, Adam}
LEARNING_RATE	0.001	$(0.0001, 0.01)$

Table 1: Hyperparameters

We train a baseline model using 5-fold cross-validation with the default parameter values from Table 1 (see Footnote 1). We use early stopping to terminate the training loop when the validation loss stops improving for three epochs, with a maximum of 30 epochs. Note that we checkpoint the model whenever the best validation loss is improved, meaning that the final epochs without improvement do not affect the final model. We plot the learning curves of the model with the lowest overall validation loss in Figure 1.

As can be seen from Figure 1, both the loss and accuracy metrics show decent improvement. There are noticeable spikes in both learning curves, which may indicate unstable training due to the relatively small training sets resulting from cross-validation (again, see Footnote 1). Evaluating the model on the test set results in a test loss of 0.55 and a test accuracy of 62%, indicating that additional performance may be achieved by further optimizing the hyperparameters.

[1]We perform cross-validation for this part as well, since we set up our code to always use a validation set during training. This was due to a discrepancy between the requirements listed in the work structure and submission details sections. We are aware that in practice, you would train the model on the entire training set and test it on the testing set afterwards. All plots show the learning curves of the best model resulting from cross-validation.

[2]We use the formula $P = \frac{1}{2}(K - 1)$ to compute the correct amount of padding P based on the kernel size K.

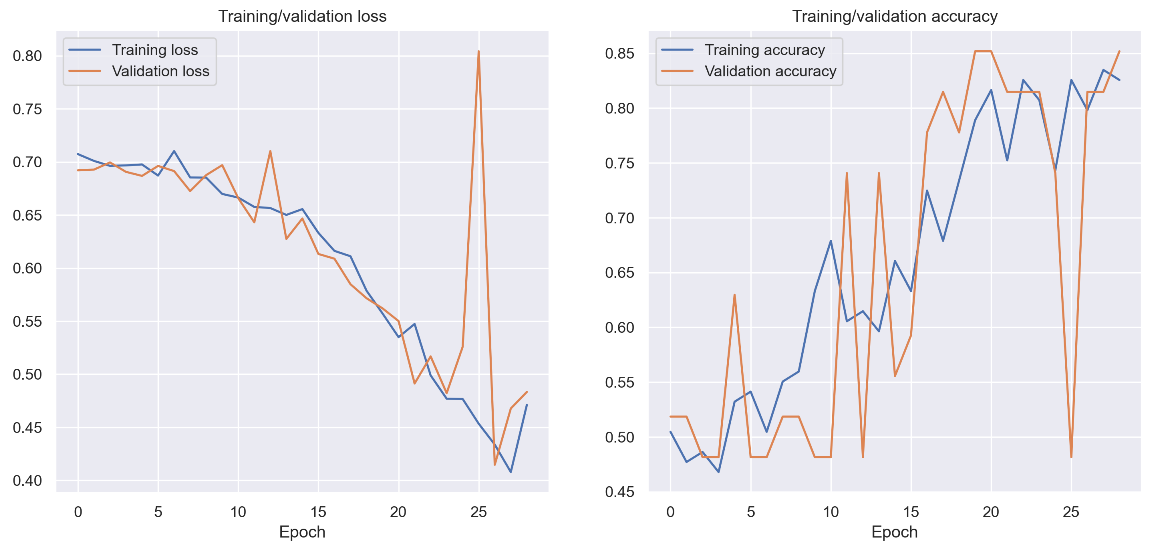

Figure 1: Training and validation loss and accuracy over epochs

Hyperparameter Optimization

In this section we cover how automatic machine learning techniques can be used to tune the hyperparameters of our model architecture. In particular, we investigate two optimization methods: random search and Tree Parzen Estimation (TPE). Random search is a simple algorithm that generates and evaluates random configurations of hyperparameters, whereas TPE is a Bayesian optimisation algorithm that tries to improve upon configurations that performed well in the past.

We implement both methods using the Hyperopt library, which readily provides implementations for both algorithms. We optimize our model using the FMIN function with the average validation loss obtained from 5-fold cross-validation as the optimization target. The search space is set to the cross product of the individual hyperparameter domains shown in Table 1. We run both algorithms for 50 iterations and record the best hyperparameter configuration in Tables 2 and 3.

Parameter	Value
NUM_BASE_CHANNELS	8
NUM_CONV_BLOCKS	2
KERNEL_SIZE_CONV	3
KERNEL_SIZE_POOL	2
ACTIVATION	LeakyReLU
OPTIMIZER	Adam
LEARNING_RATE	0.006
Average training loss	0.73
Average validation loss	0.40
Average training accuracy	65%
Average validation accuracy	72%

Table 2: Best parameters (random)

Parameter	Value
NUM_BASE_CHANNELS	32
NUM_CONV_BLOCKS	3
KERNEL_SIZE_CONV	7
KERNEL_SIZE_POOL	2
ACTIVATION	LeakyReLU
OPTIMIZER	Adam
LEARNING_RATE	0.001
Average training loss	0.63
Average validation loss	0.35
Average training accuracy	66%
Average validation accuracy	83%

Table 3: Best parameters (TPE)

It is clear from Tables 2 and 3 that both algorithms manage to outperform the baseline model, where TPE managed to find a better configuration than random search. Both algorithms seem to converge towards the same values for the pooling kernel size, activation function, and optimizer, indicating that these parameters could be fixed to their best values. It is notable that besides these parameters, random search and TPE seem to have found very different models, as random search finds one of the smallest models, whereas TPE finds one of the largest models. This, combined with the relatively unstable training curves and low overall performance, indicates that there is not enough data, or that our chosen architecture contains some fundamental flaws. We lean towards the second conclusion however, as the main focus of our architecture was a good parameterization for tuning as opposed to absolute performance.

Model Architecture After Optimization

Finally, we use the best hyperparameters from Table 3 to train our final model (see Footnote 1). A graph of the training and validation metrics can be seen in Figure 2. Evaluating this model on the test set results in a test loss of 0.31 and a test accuracy of 88%, which significantly outperforms the baseline model's test score.

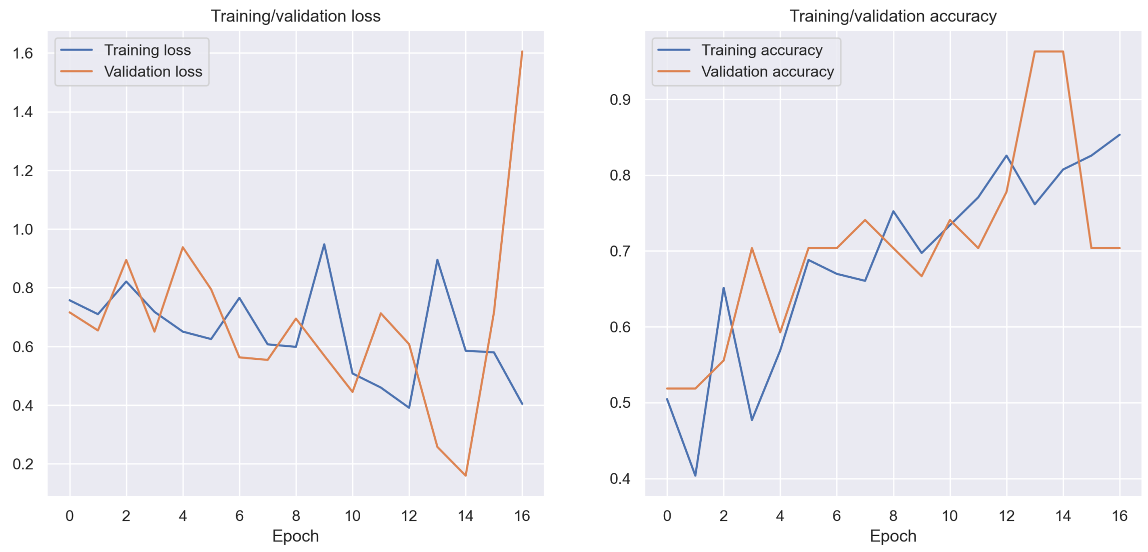

Figure 2: Training and validation loss and accuracy over epochs

Conclusion

In conclusion we find that we were able to train a binary image classifier to identify defects in products, and that we were able to effectively apply automatic machine learning techniques to tune the hyperparameters of the model. We recognize that our final model is not directly applicable in real-world applications, due to the relatively low performance and unstable learning curves.

As future work, we might be able to improve our model by experimenting with different architectures and hyperparameterizations. Another potential area of improvement lies in data preprocessing, as the input resolution was kept very low for computational reasons. A higher image resolution may provide the model with more detail in order to make a more accurate assessment of the product. Due to time constraints, we also did not experiment with data augmentation techniques such as random flipping, rotation, or cropping, which also have the potential of increasing performance without requiring more data to be gathered.