Universiteit Hasselt

Machine Learning

Project Report

Authors:
Gwendoline NIJSSEN
Melih DEMIREL

Academic year 2023-2024

1 Dataset

For this project, we were given a dataset with four tool classes: Wrench, CombWrench, Screwdriver, and Hammer, each containing about 50 real-world images. The dataset was noisy, with many images partially obscured by objects or cluttered backgrounds, making the model prone to overfitting due to limited data. We also had a large set of synthetic images for each class.

1.1 Resizing and rescaling

All images were resized to 256 pixels. This choice was initially random, but tweaking the size consistently led us back to 256. We rescaled images by dividing pixel values by 255 to scale them to the [0,1] interval.

1.2 Synthetic data

Synthetic data can expand the training dataset and potentially improve generalization. However, models trained mostly on synthetic data may not perform well on real images due to differences in texture, color, and context. Given that our application requires high accuracy with real-world images, training on real data ensures that the model is exposed to the actual conditions it will encounter during deployment. This is why we omitted the synthetic data in our implementation.

1.3 Data Augmentation

To generate more training images and help the model generalize, we used augmentation with TensorFlow's preprocessing layers: RandomFlip, RandomRotation, RandomZoom, RandomBrightness, RandomTranslation, RandomCrop, and RandomContrast. This improved model accuracy, with a basic CNN initially achieving 30-40% accuracy. However, excessive contrast reduced performance by making images less distinguishable. We optimized the augmentation settings and achieved better results.

Figure 1 shows a couple of examples of possible augmentations on one image, similar to what will be performed on all the training data.

Figure 1: Some examples of the preprocessing layers applied on an image from the training dataset.

1.4 Data split

The distribution of images across the training, validation, and test sets followed general guidelines: 70% for training, 20% for validation, and 10% for testing. We generated different shuffles of this split to test various distributions. One shuffle consistently performed better, likely because the CombWrench training data often only showed parts identical to a normal wrench, causing inconsistencies.

While larger datasets might allow different splits, the 70/20/10 split is effective for our dataset size, ensuring reliable training, validation, and testing. We also tried manually selecting a versatile training set to reduce overfitting, but this did not significantly impact test accuracy.

1.5 Extra testdata

To ensure our model generalizes well to other real-world images, we created a dataset of about 100 Google images per class and tested the models on these images. This helped identify inconsistencies and overfitting to background details present in the provided dataset but not in other real-world images.

2 Architecture of our CNN

2.1 Simple architecture

We started off with a rather simple architecture with only two convolutional layers, followed by pooling layers and a dense layer. This model was very prone to overfitting, so we tried many different set-ups. In this section, we will discuss our final choices and what has led to them.

Figure 2 shows the final set-up of the model layers.

Figure 3 shows the compilation of the model using the optimizer, loss function and metrics described below.

2.2 Chosen layers

The chosen architecture, with multiple convolutional layers followed by pooling and batch normalization, was designed by trial and error, to extract and process features from the small and noisy dataset. By adding additional convolutional layers, we aim to enhance the model's ability to learn robust features, thereby improving its generalization performance. The use of regularization techniques, such as L1 and L2 regularization and dropout, further ensures that the model does not overfit to the limited training data.

2.2.1 Convolutional Layers

The current convolutional layers are the following:

- Conv2D(128, (3, 3), activation='relu')
- Conv2D(64, (3, 3), activation='relu')
- Conv2D(32, (3, 3), activation='relu')
- Conv2D(16, (3, 3), activation='relu')
- Conv2D(32, (3, 3), activation='relu')

Convolutional layers are needed for feature extraction. By stacking multiple convolutional layers, we hoped to better capture these features such as the edges of the objects we are trying to classify, as well as certain details that would distinguish them from the noisy backgrounds. We hoped more convolutional layers would improve the model's ability to generalize despite the small dataset size.

The activation function used here is the ReLu function. We tried out LeakyReLu to see if it would work better on our noisy images but the improvement was not measurable when doing a lot of epochs, which was necessary for our data.

The filter width was chosen to be 3x3 since this yielded the best consistent results on the model we trained for 100-200 epochs. A larger filter (9x9) gave overall better accuracy, but the per-class accuracies were smaller than with smaller filters and a lot more training needed to be done to get better results for all classes.

2.2.2 Pooling Layers

We added MaxPooling2D((2, 2)) after each convolutional layer. Pooling layers reduce the spatial dimensions of the feature maps. This should decrease the computational load and hopefully prevent overfitting. MaxPooling specifically keeps the most prominent features, which ensures that the critical information is preserved while reducing the complexity of the model.

2.2.3 Batch Normalization

Batch normalization standardizes the inputs to a layer for each mini-batch, which should stabilize the learning process and cause faster convergence. We hoped this technique would help to increase the speed the model would learn so our learning rate would not need to change much. Since we are working on basic laptops and usually have limited time, we can't rely on training our models for a very long time with an insane number of inputs. This seemed to indeed work a little bit.

```
# Resizing images is done beforehand
model = Sequential([
    Input(shape=(img_height, img_width, 3)),
    create_preprocessing_layers(),
    tf.keras.layers.Rescaling(1./255),
    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    BatchNormalization(),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    BatchNormalization(),
    Conv2D(32, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    BatchNormalization(),
    Conv2D(16, (3, 3), activation='relu'),
    MaxPooling2D((2, 2)),
    BatchNormalization(),
    Conv2D(32, (3, 3), activation='relu'),
    Flatten(),
    Dense(256,
        activation='relu',
        kernel_regularizer=regularizers.L1L2(l1=1e-4, l2=1e-4),
        bias_regularizer=regularizers.L2(1e-4),
        activity_regularizer=regularizers.L2(1e-5)
    ),
    Dense(128,
        activation='relu',
        kernel_regularizer=regularizers.L1L2(l1=1e-4, l2=1e-4),
        bias_regularizer=regularizers.L2(1e-4),
        activity_regularizer=regularizers.L2(1e-5)
    ),
    Dropout(0.4),
    Dense(64,
        activation='relu',
        kernel_regularizer=regularizers.L1L2(l1=1e-4, l2=1e-4),
        bias_regularizer=regularizers.L2(1e-4),
        activity_regularizer=regularizers.L2(1e-5)
    ),
    Dropout(0.3),
    Dense(4, activation='softmax')
])
```

Figure 2: Model set-up

```
model.compile(
    optimizer = keras.optimizers.Adam(
        learning_rate=1e-4,
        weight_decay=1e-6,
    ),
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=[
        keras.metrics.SparseCategoricalAccuracy(name="sc_accuracy"),
        'accuracy',
    ]
)
```

Figure 3: Compilation of the model

2.3 Optimizer & Learning rate

We chose the Adam optimizer for training the model due to its adaptive learning rate and efficiency with sparse gradients. Adam, short for Adaptive Moment Estimation, combines the benefits of AdaGrad and RMSProp, making it suitable for handling noisy data and achieving quick convergence.

The learning rate, which controls the model's response to error estimates during weight updates, was set to the standard value of 1e-4. A smaller learning rate ensures precise convergence but requires more epochs. Our best models needed 100-200 epochs to reach their full potential, balancing convergence speed with training stability. Other optimizers can achieve similar outcomes, but some converge faster or need different hyperparameter tunings. We stuck with Adam for its consistency and reliability.

We experimented with tuning the Epsilon value, a small number to prevent division by zero, typically set to 1e-8. This adjustment didn't significantly impact our model.

We also tested weight decay, a regularization technique that penalizes large weights to prevent overfitting. In our setup, weight decay didn't always improve performance the way we wanted it to, so we excluded it from the final configuration.

2.4 Loss function

The loss function that we used is Keras' SparseCategoricalCrossentropy. In turn we also used the Sparse Categorical Accuracy, which measures the accuracy of the model by comparing the predicted class with the true class. Sparse in this context refers to the fact that the class names are not one-hot encoded but rather provided as integers which we use as an index to later retrieve the class name.

We found this loss function was the best choice for this project, as we need to be able to provide predictions for images and the certainty for each class.

2.5 Normalization & Regularizers

We added L1 and L2 normalization to reduce overfitting, applying these techniques to different Dense layers in the model. Adding L1 and L2 normalization helped the most in stabilizing the learning process and preventing overfitting. We noticed the validation loss and training loss, as well as the validation accuracy and training accuracy were converging in a much more similar way when using these techniques. It was hard figuring out when to add which normalization technique. The kernel, bias and activity regularizes all were meant to attack different problems. We looked at our model's weights and bias and found the results hard to interpret.

Various sources indicate that biases are crucial in image classification as they allow neurons to activate even when the weighted sum of pixel values is slightly below the activation threshold. This flexibility is important given our limited training data. For activity and kernel regularizers, we tested default values and tweaked them to suit our data.

We tried to fine-tune the parameters to give us the best result possible. Ultimately, we decided upon the configuration that is seen in figure 2. Further fine-tuning is definitely possible in this area.

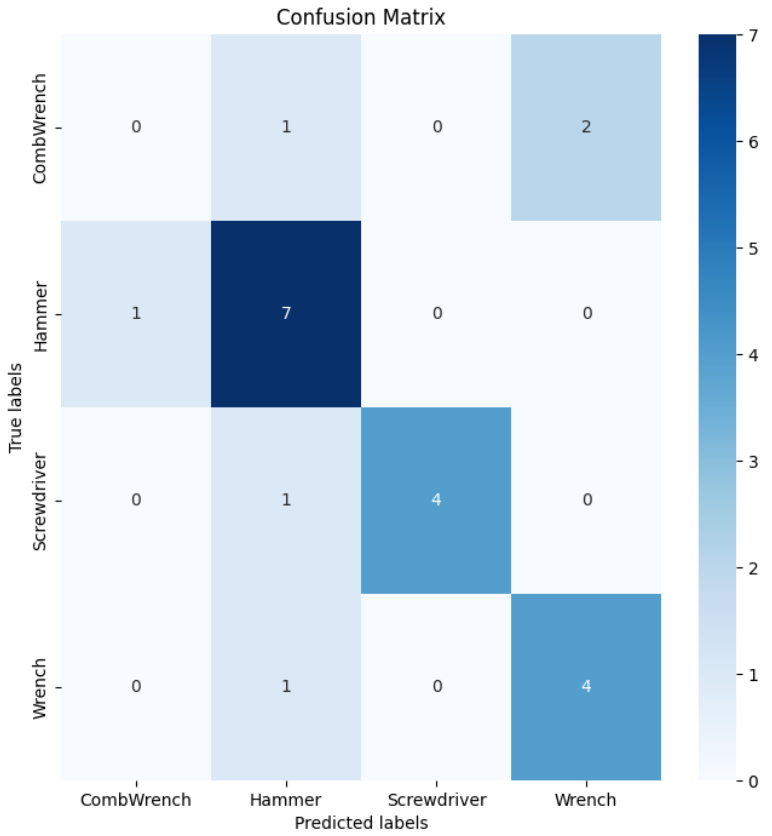

Figure 4: The confusion matrix for the final model visualised in a heatmap.

3 The model

Deciding when to stop training the classifier was not simple. Of course we tried out a lot of different combinations of all the previously mentioned techniques, we have tried early stopping when the validation loss or accuracy stops moving significantly.

Ultimately what helped us decide on a good model was looking at the confusion matrices. In figure 4 we can see that the model performs relatively well for most classes, except for the CombWrench. This makes sense, as the combination wrench is often not very distinguishable between the normal wrench.

In figure 5 we can tell there is perhaps still a bit of overfitting going on. We have tried to combat this by changing the L1 and L2 normalization values and this works, but when training we notice the accruacy for the classes doesn't go much higher than they are now. We decided on this final model because it performed the best on the most amount of classes (test accuracy ¿ 70% for 3/4 classes) compared to a mediocre 50% for all classes in some other cases. We think it's pretty explainable why the test accuracy for CombWrench was nog great: This is because the model was tested on a testsplit where the test images for CombWrench only contained images without the side of the wrench that distinguishes it from a normal wrench. This result makes sense that way.

Figure 7 also shows that the model performs better on the test dataset created from the original dataset, than on a random selection of Google images for each class. This indicates the model can still improve on generalization.

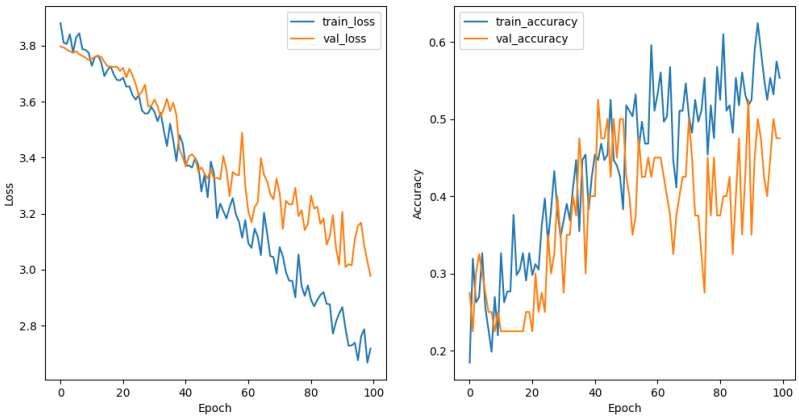

Figure 5: The loss and accuracy for training and validation of the final model.

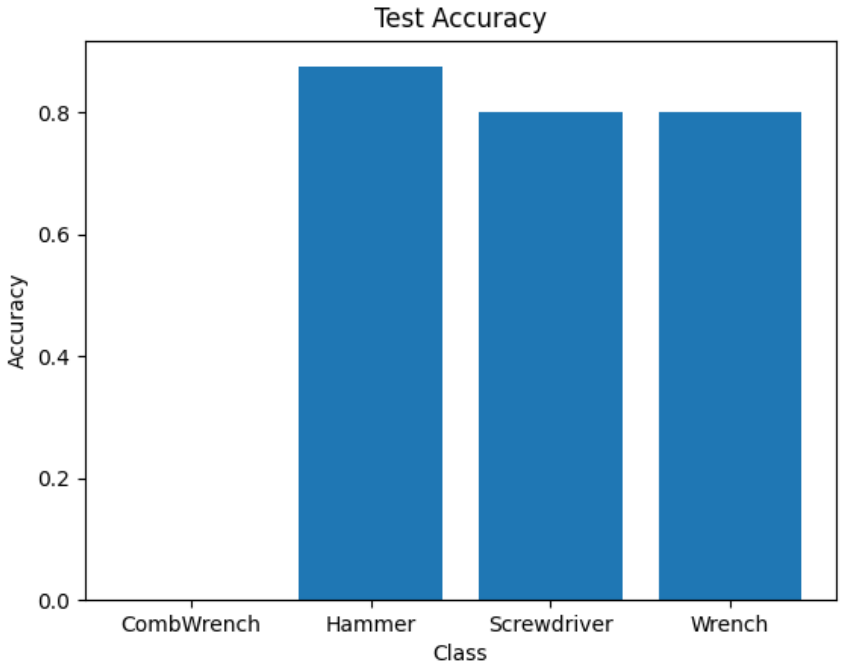

Figure 6: The test accuracy on the test dataset for each class.

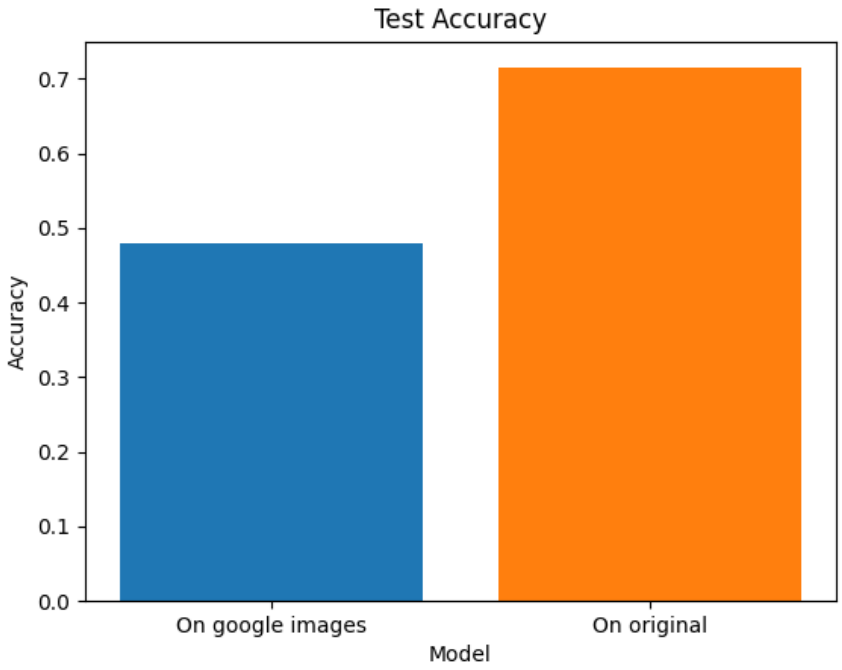

Figure 7: The test accuracy on the test dataset and another test dataset of random google images.

4 Privacy implications

4.1 False positives and false negatives

In a manufacturing process, we assume the safety of these workers is quite essential. An incorrect prediction with large certainty can cause errors in the process and may cause safety issues if people are using incorrect tools. This corresponds to false positives. A false negative would be where the system fails to identify a tool correctly, possibly with lower certainties, which defeats the purpose of the system and will not provide the necessary feedback that the operator is expecting from the assistive tool.

4.2 Misuse

The system could be misused by companies to monitor the operators and gather data on their overall performance, whereabouts, and other sensitive information that is otherwise not tracked. The fact that a camera is used may also indicate that other data can be saved that is not needed for predictions, without the operators knowing.