CS/RBE 549 Computer Vision, Fall 2020

Project Report

Team L

Member	Signature	Contribution(%)
Zhuoran Su	Zhuoran Su	45
Joseph Caltabiano	Joseph Caltabiano	45
Yuance Zhang	Yuance Zhang	10

Grading:		
	Approach	_____/15
	Justification	_____/5
	Analysis	_____/15
	Testing & Examples	_____/15
	Documentation	_____/10
	Difficulty	_____/10
	Professionalism	_____/10
	Presentation	_____/20
	Total	_____/100

Contents

Code— `https://github.com/jpcaltabiano/computer_vision_tool_classifier/`

1 Introduction

In this paper, we attempt to recognize tools in images. To challenge ourselves, we wanted to recognize any of three different tools from an image, using the same technique. We chose to use hammers, screw drivers, and wrenches as the three tools we wanted to recognize. Our goal was to be able to pass a model an image, and be able to know which tool was represented in the image.

2 Approach

Convolutional neural networks are well-known for their efficacy in image-related tasks. They are a common tool used in image classification. In this project, we use a convolutional neural network as our base model, and we use the Visual Geometry Network to test our result. We use a few different Keras functions and an edge detector to do the image processing so that we could reduce the overall calculation and increase the volume and complexity of our data.

2.1 Models

2.1.1 Convolutional Neural Network

Convolutional neural networks have been one of the core algorithms in the field of image recognition for a long time, and they have stable performance when the learning data is sufficient [1]. In solving common large-scale image classification problems, convolutional neural networks can be used to construct hierarchical classifiers[2], and CNN can also be used to extract discriminating features of images in fine-grained recognition for other classifiers to learn [3]. In the recognition of fine classification, feature extraction can artificially input different parts of the image into the convolutional neural network [4], or it can be extracted by the convolutional neural network through unsupervised learning [5]. Similarly, CNN can also be used for object recognition. In this project, CNN was the first network we thought of and tried.

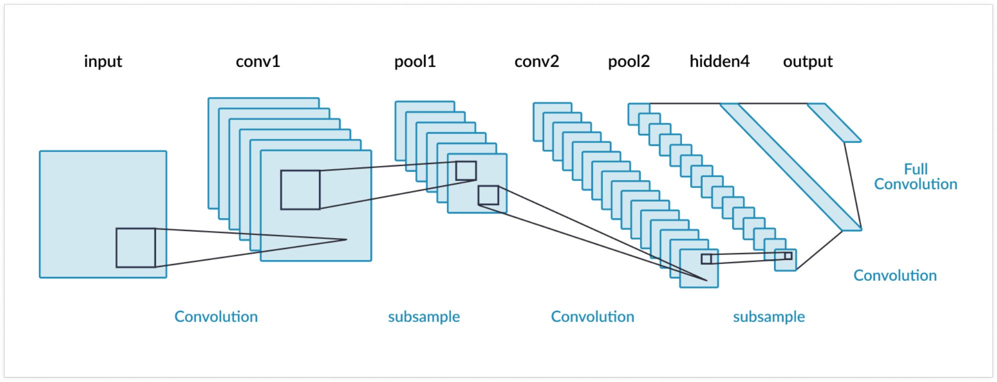


Figure 1: The architecture of CNN

2.1.2 VGG16 and Transfer Learning

We also use Visual Geometry Group Network (VGG16) to classify the data. VGG16 is a very complex cutting-edge model that we could leverage to test results. Figure2 shows the architecture of the VGG16.

The input of the first convolutional layer is a fixed size 224 x 224 RGB image. The image passes through a stack of convolution (transformation) layers, which uses a 3×3 filter.

The convolution stride is fixed at 1 pixel; the converted space is filled. Layer input: For example, the spatial resolution is preserved after convolution. For 3×3 convolutional layers, the padding is 1 pixel. The spatial padding is preserved after convolution. For example, the padding is 1-pixel for 3×3 convolutional layers. Spatial pooling is performed by five maximum convolutional layers, which are followed by some convolutional layers (notice: not all convolutional layers are followed by maximum convolution). The max-pooling is performed on a 2×2 pixel window with stride 2.

Three fully-connected (FC) layers follow a stack of convolutional layers (these layers have different depths in different architectures). The last layer is the soft-max layer. All hidden layers have rectification (ReLU) nonlinear characteristics. It should also be noted that, except for one network, all networks do not contain local response normalization (LRN) [6].

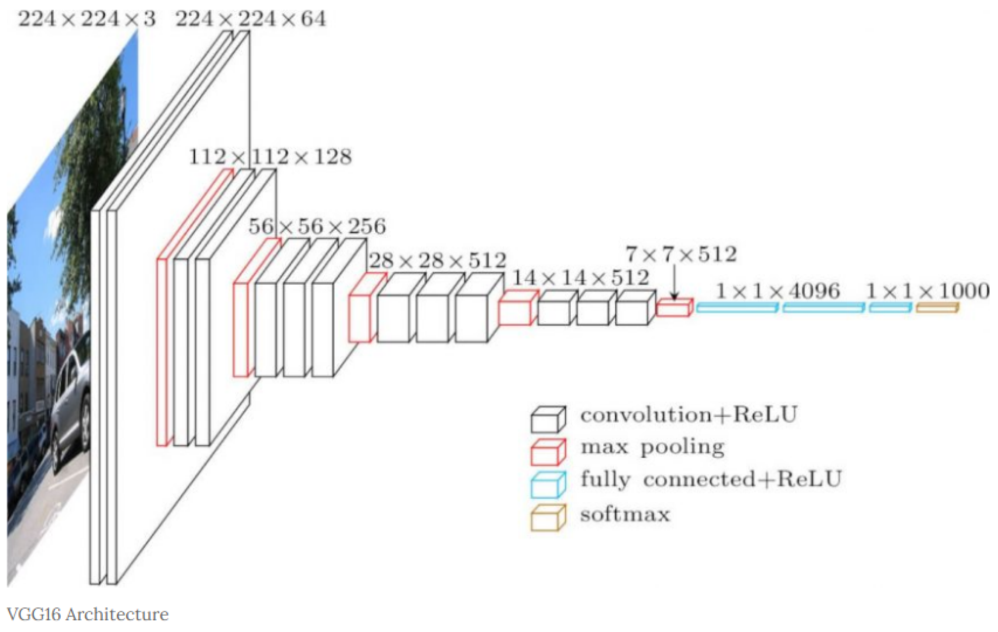


Figure 2: The architecture of VGG16

2.2 Data

2.2.1 Data Selection

Algorithms like the CNN often require a very large volume of data to get a model that is well fit. Especially for image classification tasks, training with more data can often allow for more accurate prediction. However, the quality of the data is just as important. For the scope of this project, we did not have the means to generate this type of data, so we examined several existing image sets of tools and associated items that were curated for use in training predictive models. We chose data from "https://www.kaggle.com/salmaneunus/mechanical-tools-dataset".

2.2.2 Image Formatting and Generation

Image processing can allow us to both generate new examples for training, and format them for input into the model. When the images are loaded in to the program, the first thing that must be done is resize the images into a square. We used a few different Keras functions that automated loading in image data from a directory and applying labels. These functions support resizing, so we passed our desired image dimensions to these functions, which automatically resized the images. Resizing images down to a square grid saves the need to introduce much more complexity to the model. Traditional CNN's like the ones we used (as opposed to fully convolutional networks) flatten the output of the last convolutional layer with a fixed size, so they require all images to be the same size. It is easier to resize every image to a square as resizing every image to the same size rectangle causes the same general loss of data, and can cause unwanted behavior if the fixed size is 'landscape' and you have some 'portrait' images.

We initially resized the images to a 120x120 px array. We later tried resizing to 224x224 px. Using these larger images that contained more data resulted in a better model as will be discussed below. After resizing, we re-scaled the values in the image array. Each pixel in the array can contain values ranging from 0-255 in each of their color channels. Using data with a relatively large range like this can slow the model down as it computes larger and larger values. We scaled all the values down to a range of 0-1, preserving the relative values but compressing the range. Once we had our images formatted to be fed into the model, we applied a few different image operations to the data to create five additional training images for each original image. For each image, we did a horizontal flip, a vertical flip, a random rotation up to 20 degrees, a random zoom up to 20%, and a random shift in brightness in the range 0.2, 1.0.

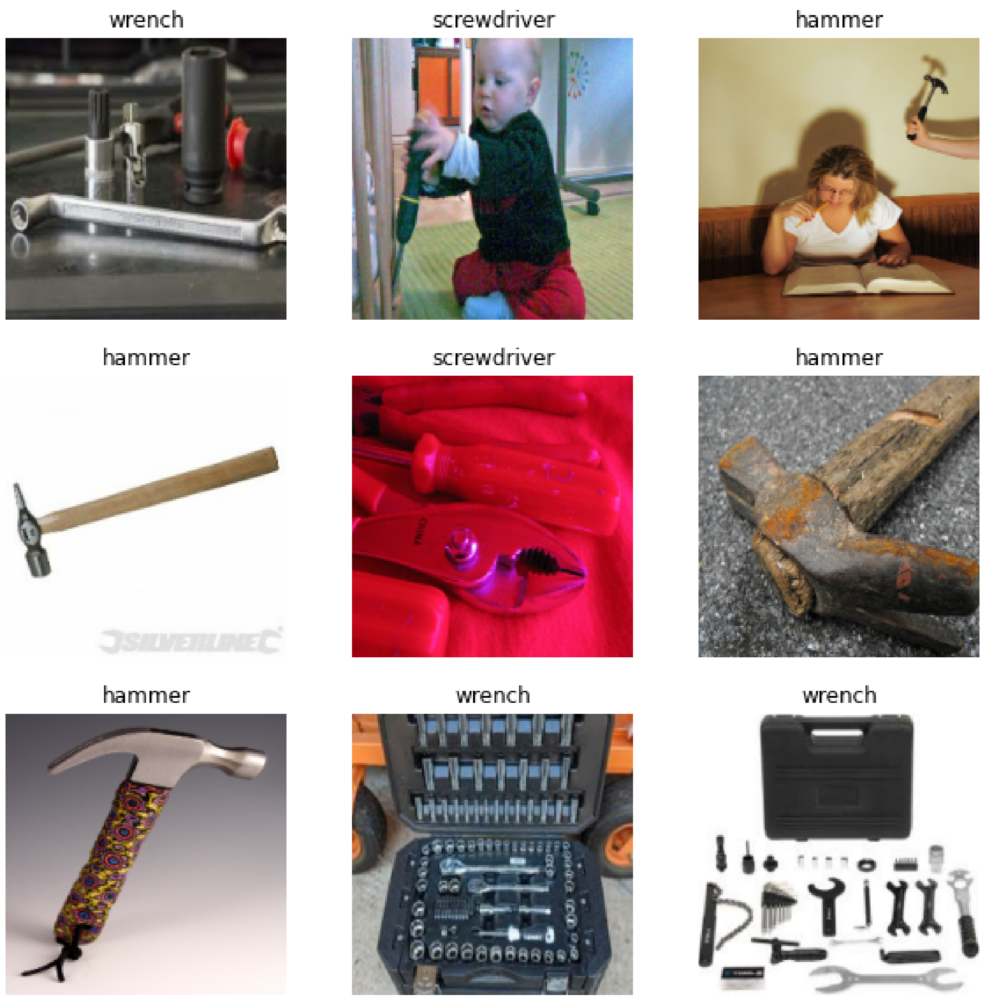


Figure 3: A sample of images from the training data, after resizing to 224x224 px

2.2.3 Edge Detector

Our original images contained a significant amount of "noise" in the form of complex backgrounds. Although many did have a plain white background, many others had people holding the tools, or the tools up against a confusing background such as a toolkit. We hypothesized that being able to isolate the outline of the desired tool and using that as input instead would result in an increase in performance. Although the CNN is able to handle complex backgrounds well, we wanted to see if we could eliminate the need for it entirely by simply removing the backgrounds before the image is passed to the model. We applied OpenCV's Canny edge operation function to the images with a sigma of 0.33. Under ideal circumstances, to obtain the best outline of the target we would be able to customize the parameters in the edge detector for each image. However, not only is this very difficult at scale, it would cause poor performance with real-world test images. 'Snooping' on the data like this would cause overfitting to perfect outlines, and attempting to match the process on the test images would defeat the whole purpose of the model.

As a result of using this one-size-fits-all approach to generating we did see some images that did not have a clean outline of the target tool. They often had the target's outline divided into multiple parts, or in a way that implied it was connected to another part of the image, like the background or a hand holding it.

3 Results

3.1 Baseline

We started by building a fairly simple CNN model that we could use to test how we handled input and output shapes and try to identify areas for potential bugs from the start. We also used this model to generate a baseline that we could compare our results to as we experimented. This model contained 3 CNN layers of 16,32,64 filters, sequentially. They all used square kernels of size 3. Each CNN layers was followed by a max pooling layer. The final three layers were a 20% dropout, an FC dense layer of size 128 using the reLU activation, and a final FC dense layer of size 3 to give us out output in the shape of 3 probabilities, one per class, for each example. This model reached just about of 65% validation accuracy, which was better than a coin flip but there was still much room to improve. We trained this model using the Adam optimizer with no custom parameters, and using sparse categorical cross-entropy loss. We allowed the model to attempt to train for many more epochs than necessary, and used a callback to stop the model from training when the validation loss remained within a certain range for a certain number of epochs, depending on the experiment.

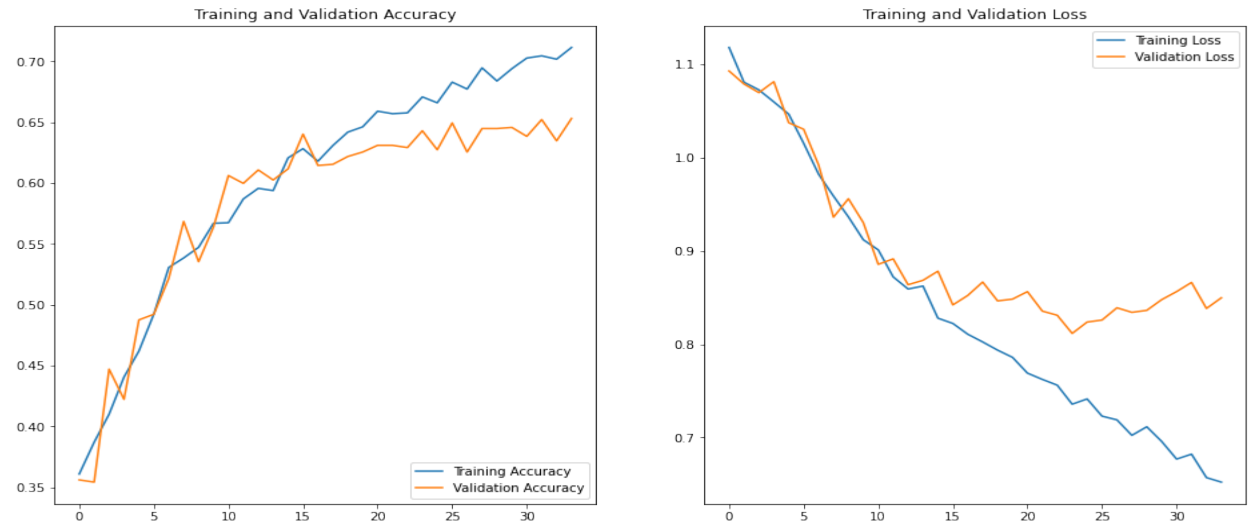


Figure 4: The performance of the baseline model

3.2 Improving Performance

In this section, we will introduce the methodology we used in this project for the improvement of our 10-layer CNN classifier. We will discuss the impact of changing the learning rate, the input image size, and adding the edge detector.

3.2.1 Hyperparameter Tuning

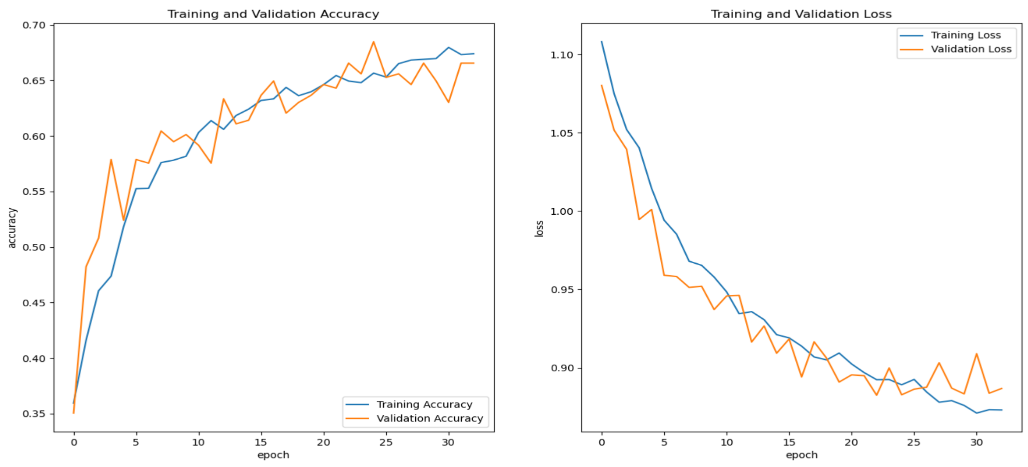


Figure 5: Impact of learning rate 0.001

To solve the overfitting problem, we decided to change the learning rate. We tried 4 typical learning rates and found a larger learning rate like 0.1 did not give us a well trained model. In the training phase, our model stopped from the callback function really fast. Our callback function has a patience of 5 and only saves the model with the highest validation accuracy during the training process. But the validation accuracy is 0.65 which is equal or worse than our baseline implementation.

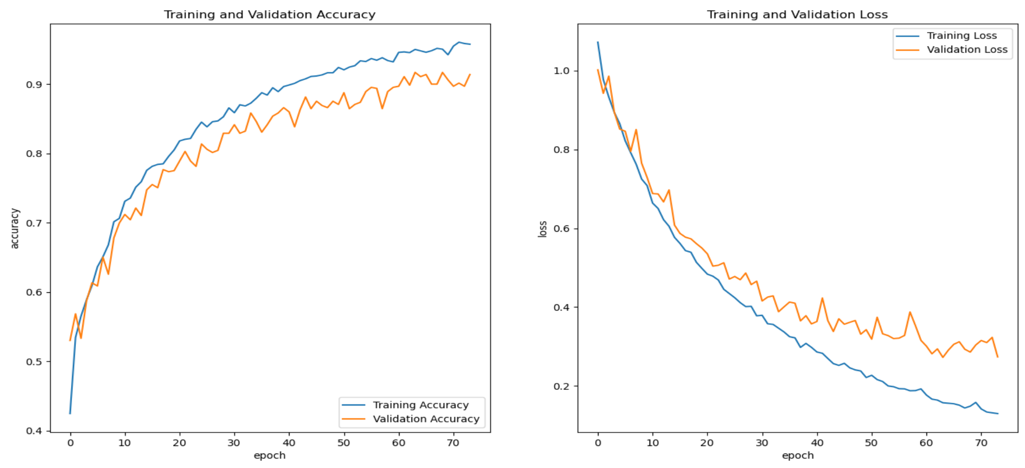


Figure 6: Impact of learning rate 0.0001

In our experiment, the learning rate of 0.0001 tends to give us the lowest validation loss and relative low time consuming. The validation accuracy achieves 90% after training 70 epochs.

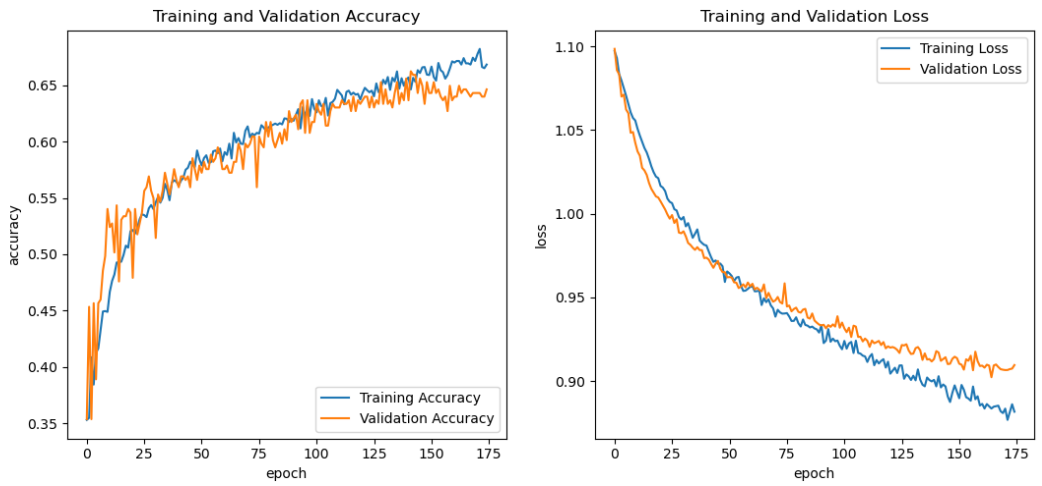


Figure 7: Impact of learning rate 0.00001

We also tried learning rates smaller than 0.0001. Figure 7 shows the training progress of the CNN with a learning rate of 0.00001. The validation loss of this model converged extremely slow compared with that of a model with a larger learning rate. When the training phase was stopped by a callback function, the validation loss was still about 0.85. Our experiment shows that large learning rates result in unstable training and tiny learning rates converge slowly or even do not converge and result in a failure to train.

3.2.2 Impact of Image Size and Data Augmentation

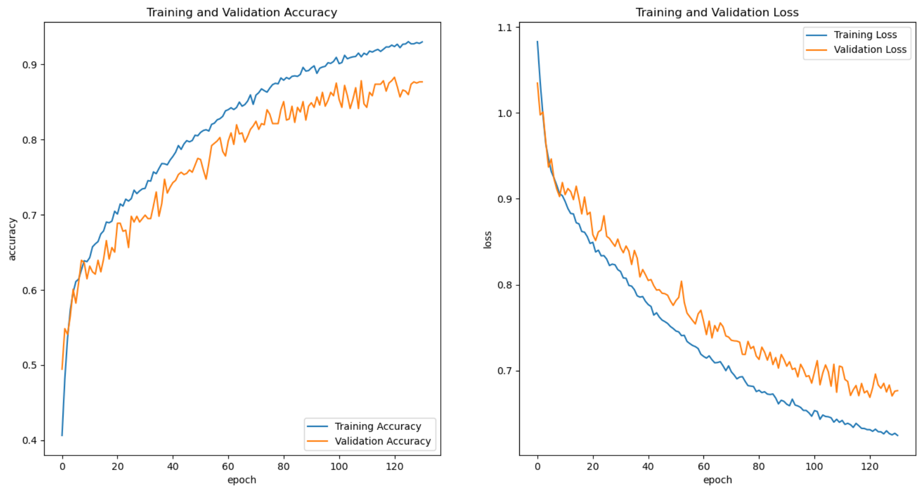


Figure 8: Training Phase after applying Data Augmentation and Resizing Image

We resized images to 224 by 224 pixels to keep more patterns of the image. After applying more image augmentation options like brightness shift and scale, we have a larger training image set and the validation accuracy achieves 92%. As shown in Figure 5, the model has higher accuracy on the validation set than the training set at the beginning of training process. We did some research and found out it is a consequence of data augmentation because we only augmented the training set but did not do anything to the validation set, which means our validation set consists of "easier" examples than the training set. It is also due in part to have a high amount of dropout in the training phase. During validation, the dropout layers are not used to calculate the prediction.

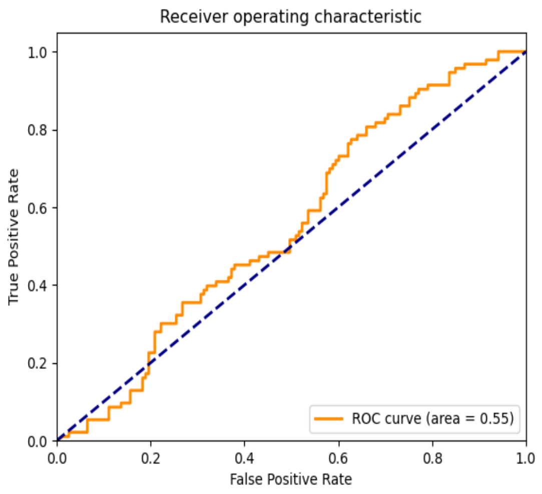
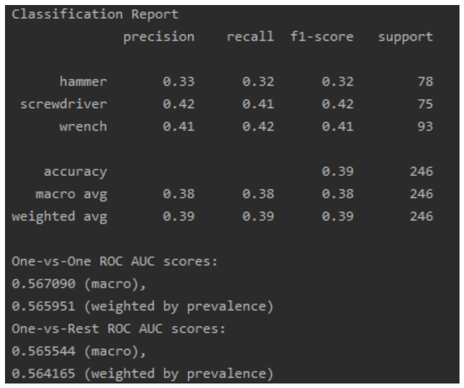


Figure 9: ROC of model with Data Augmentation



```
Classification Report
              precision    recall  f1-score   support

      hammer       0.33      0.32      0.32        78
 screwdriver       0.42      0.41      0.42        75
      wrench       0.41      0.42      0.41        93

    accuracy                           0.39       246
   macro avg       0.38      0.38      0.38       246
weighted avg       0.39      0.39      0.39       246

One-vs-One ROC AUC scores:
0.567090 (macro),
0.565951 (weighted by prevalence)
One-vs-Rest ROC AUC scores:
0.565544 (macro),
0.564165 (weighted by prevalence)
```

Figure 10: Accuracy metrics and AUC

The overall performance of this model is poor. It is difficult for this model to classify the three tools we selected even after we solved the overfitting problem. The ROC curve in 9 is quite flat, and as seen in 10, the f1 scores for every class are very low.

3.2.3 Canny Edge Detector

To improve the accuracy, we applied a Gaussian filter to blur the image then input the edged image to Canny edge detector. Canny edge detector is commonly used for edge detection. The edge detector will compute the gradient intensity and detect edges. The threshold of the Canny edge detector is chosen by computing the median of the single-channel pixel intensities. There is also a parameter sigma can be used to vary the percentage of thresholds.

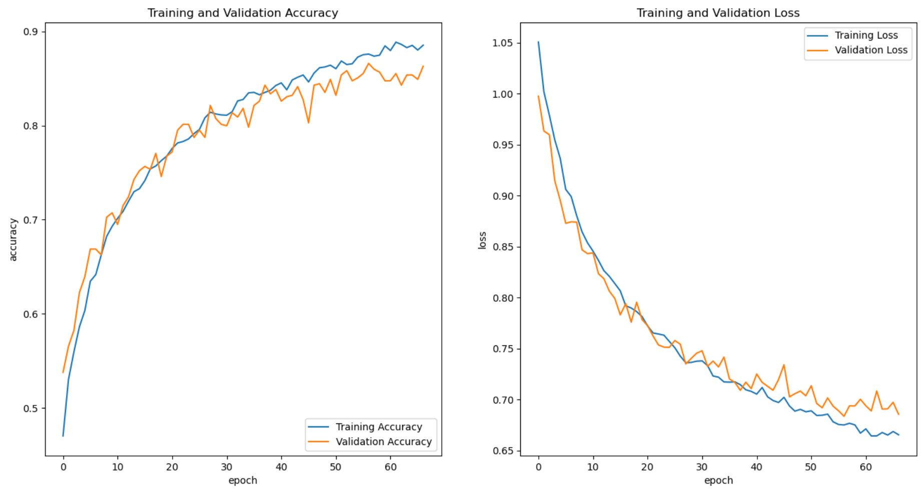


Figure 11: Training Phase after applying Canny edge detector

Figure 11 shows the training process after applying the edge detector. The cost of time on the training phase was reduced, which means the validation converged earlier compared with the model without Canny edge detector. However, the validation loss was similar to previous result, which was about 0.65.

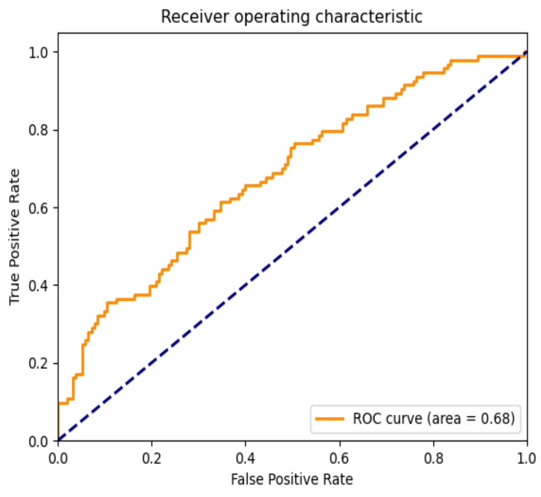
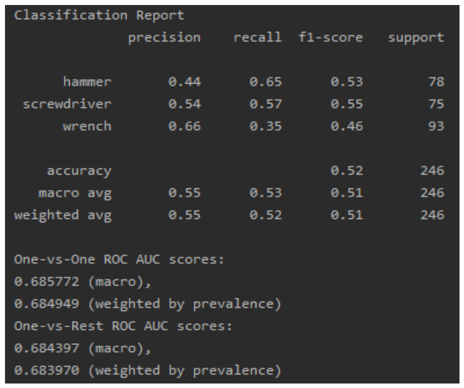


Figure 12: ROC of the model with the Canny Edge Detector



```
Classification Report
              precision    recall  f1-score   support

      hammer       0.44      0.65      0.53        78
 screwdriver       0.54      0.57      0.55        75
      wrench       0.66      0.35      0.46        93

    accuracy                           0.52       246
   macro avg       0.55      0.53      0.51       246
weighted avg       0.55      0.52      0.51       246

One-vs-One ROC AUC scores:
0.685772 (macro),
0.684949 (weighted by prevalence)
One-vs-Rest ROC AUC scores:
0.684397 (macro),
0.683970 (weighted by prevalence)
```

Figure 13: Classification report after applying the Canny Edge Detector

As shown in Figure 12. The ROC score was 0.68 after using edged images. Figure 13 shows that the accuracy on test set was increased by about 0.15. However, the overall performance still did not achieve our design goal. As we talked about in 2.2.3, the edge detector was not effective at outlining the tool due to the complex background and multiple tools in the image.

3.3 VGG16

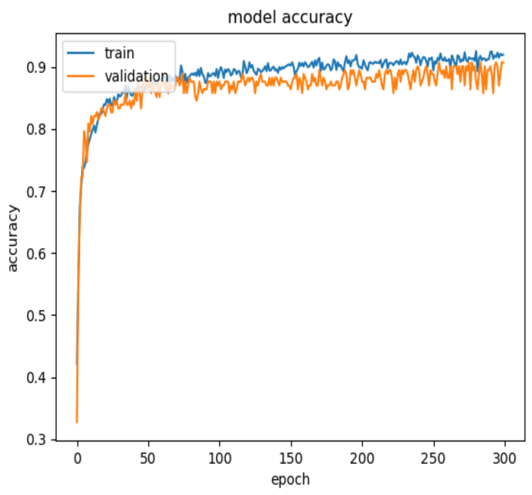


Figure 14: Accuracy

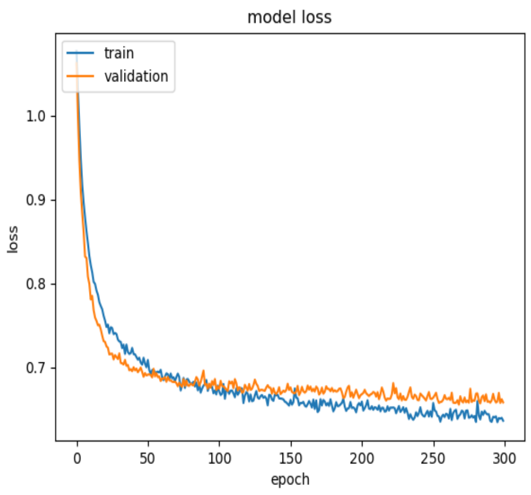


Figure 15: Loss

Although we have more than 5000 images for training, our training set is so complex that this number of images is not enough to extract useful features for classification. We turned to transfer learning to solve this problem. We used a pre-trained model as a feature extractor and constructed our fully connected layers as the classifier. In this experiment, a pre-trained VGG16 serves as the feature extractor. We kept the weight of VGG16 unchanged during training phase, and we have 14 million non-trainable parameters and about half a million trainable parameters. The validation accuracy is 90%, which is the same as our previous result but the performance on the test set is much better.

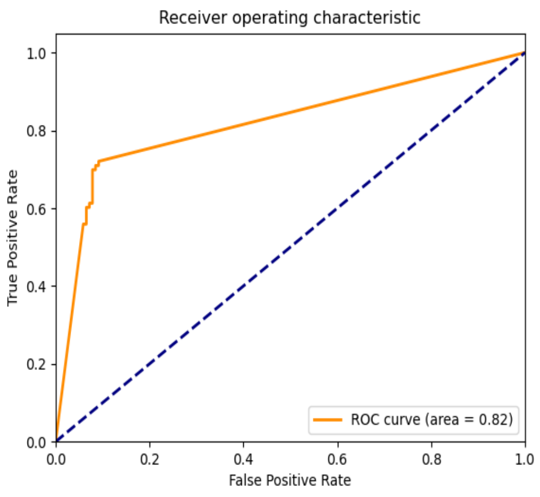
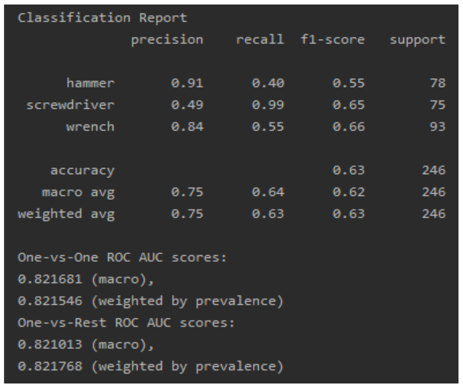


Figure 16: ROC curve



```
Classification Report
              precision    recall  f1-score   support

      hammer       0.91      0.40      0.55        78
 screwdriver       0.49      0.99      0.65        75
      wrench       0.84      0.55      0.66        93

    accuracy                           0.63       246
   macro avg       0.75      0.64      0.62       246
weighted avg       0.75      0.63      0.63       246

One-vs-One ROC AUC scores:
0.821681 (macro),
0.821546 (weighted by prevalence)
One-vs-Rest ROC AUC scores:
0.821013 (macro),
0.821768 (weighted by prevalence)
```

Figure 17: Classification report

The ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters: True Positive Rate and False Positive Rate. The One-vs-One AUC (Area Under the ROC Curve) score is 0.82, and the One-vs-Rest AUC score is also 0.82. The accuracy of the hammer is 0.91. The accuracy of the screwdriver is 0.49, and the accuracy of the wrench is 0.84. Our model is not effective at recognizing screwdrivers, so the overall accuracy of this model is 0.75.

4 Conclusion

In conclusion, we implemented our 10-layer CNN but met the overfitting and the early stop problem. We did experiments on parameter tuning and adjusted the learning rate to find the best parameters for the CNN model. Although we managed to increase the image size to 224 by 224 pixels, and leveraged image augmentation with multiple options such as brightness shift and zoom on the training set, our improved baseline implementation gave us 0.39 accuracy on the test set. The accuracy of this model was not as good as we expected. To improve the accuracy of the classifier on the test data, we decided to add a Canny edge detector, which is commonly used in computer vision applications. The accuracy was increased by 0.15. The complexity of background and objects in our training images limited the performance of the Canny edge detector. We turned to implement a transfer learning model with VGG16, which gave us a chance to retrieve more features with pre-trained weights. This VGG16-based model can successfully recognize hammer and wrench with an accuracy of 0.91 and 0.84 respectively, but it failed to classify the screwdriver with an accuracy of 0.49. On average, this model gives us an overall accuracy of 0.75.

In this project, we learned how to design and implement a CNN classifier with in the Python language. We also improved our programming skills on the deep learning API, Keras, and successfully ran our training phase on the GPU via CUDA. We realized that hyperparameter tuning could be an important step for a model to achieve better performance. Data augmentation played a crucial role in our training phase and the Canny Edge Detector effectively improved the accuracy of our classifier.

For future work, we plan to expand the image data set with more photos and more classes to make sure our training set has the diversity and richness for a more successful classifier. In addition, we will also try to add reasonably more layers and more hyperparameter tuning options to increase the depth of our neural network. As for the transfer learning part, we are planning to try more well-designed networks like ResNet50, Xception, and YOLO v3. If we can get a large enough data set, we can train all weights of our model instead of loading a pre-trained model as a feature extractor. We believe that our classifier will be improved with these approaches.

References

[1] M. Egmont-Petersen, D. de Ridder, and H. Handels, "Image processing with neural networks—a review," *Pattern recognition*, vol. 35, no. 10, pp. 2279–2301, 2002.

[2] N. Srivastava and R. R. Salakhutdinov, "Discriminative transfer learning with tree-based priors," in *Advances in neural information processing systems*, 2013, pp. 2094–2102.

[3] Z. Wang, X. Wang, and G. Wang, "Learning fine-grained features via a cnn tree for large-scale classification," *Neurocomputing*, vol. 275, pp. 1231–1240, 2018.

[4] S. Branson, G. Van Horn, S. Belongie, and P. Perona, "Bird species categorization using pose normalized deep convolutional nets," *arXiv preprint arXiv:1406.2952*, 2014.

[5] J. Krause, T. Gebru, J. Deng, L. Li, and L. Fei-Fei, "Learning features and parts for fine-grained recognition," in *2014 22nd International Conference on Pattern Recognition*, 2014, pp. 26–33.

[6] X. Zhang, J. Zou, K. He, and J. Sun, "Accelerating very deep convolutional networks for classification and detection," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 10, pp. 1943–1955, 2015.

[7] S. I. Eunus, "Mechanical tools classification dataset." [Online]. Available: https://www.kaggle.com/salmaneunus/mechanical-tools-dataset

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[9] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[10] "Anaconda software distribution," 2020. [Online]. Available: https://docs.anaconda.com/

[11] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[12] TensorFlow, "Imagedatagenerator python tutorial." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGeneratort