Predicting Pooches

Dog Breed Classifier using Convolutional Neural Network

Zachary M Lipperd
Dept. Electrical Engineering and Computer Science
University of Missouri, Columbia, Mo
zmlmcb@umsystem.edu

Abstract

Identifying dog breeds can be difficult due to the high similarity between breeds and the number of possible classes. In this study, we aim to develop a deep learning model using a convolutional neural network (CNN) to classify dog breeds. We used a dataset of over 20,000 labeled images from 120 distinct breeds. We analyzed the impact of model complexity, regularization techniques, and learning rate of our custom CNN model's performance. We also explored the use of fine-tuning a pre-trained model like ResNet50V2 for classification. Our best-performing model achieved a 66.92% accuracy, highlighting the potential of CNNs in dog breed classification.

Introduction

Accurate dog breed identification has many practical applications, such as assisting pet owners, supporting animal shelters, as well as in veterinary medicine. Although experts can easily identify dog breeds based on physical characteristics, it can be challenging for some due to the similarities between breeds. With advances in machine learning, there is an opportunity for us to develop an automated dog breed identification system using image recognition.

Convolutional neural networks (CNNs) have become the preferred technique for image classification and have shown to achieve good performance in a wide variety of applications such as object and facial recognition. However, training CNNs requires a large amount of labeled data and computational resources.

In this study, we aim to develop a deep learning model for dog breed classification using CNN. We will investigate the impact of model complexity, regularization techniques, and learning rate on our custom-built model and evaluate its performance. We will also explore the use of pre-trained models, such as ResNet50V2, and fine-tuning them for our specific task. Finally, we will evaluate our models and discuss potential applications and the limitations of our approach.

Dataset and preprocessing

Dataset description

The datasets used in this project were the Dog Breed Identification, however, after testing we changed to using the more popular and well maintained Stanford Dogs Dataset.

Initially, we used the Dog Breed Identification dataset, which contains 355 dog breeds with 25 images per breed, totaling 12,425 images. However, due to low performance in our initial models, we investigated this dataset further. We found that it contained several issues: it had limited data per class, approximately 33% of images were misclassified, and around 15% of the images were irrelevant (e.g., clip art, people, toys, and dog food brands). It appeared that this dataset was not well-curated during its collection.

As a result, we switched to the more reputable and carefully curated Stanford Dogs dataset. This dataset contains images of 120 dog breeds from around the world, built using images and annotations from ImageNet for fine-grained image categorization. It has approximately 170 images per breed, totaling 20,580 images. The increase in number and quality of images improved our model's performance by around 70%.

The images in the Stanford Dogs dataset are in JPEG format and have varying dimensions, with most images around 500 x 400 pixels. Some breeds included are Beagle, Boxer, Chihuahua, Dachshund, Golden Retriever, Labrador Retriever, Poodle, Siberian Husky, and Yorkshire Terrier, among others.

Data Augmentation

Data augmentation was employed to increase the diversity and size of our training dataset, which helps improve model generalization and prevent overfitting. We applied various augmentation techniques, such as image rotation, width and height shifts, shearing, zoom, horizontal flips, and brightness adjustments. We used TensorFlow's ImageDataGenerator for implementing data augmentation.

Using data augmentation leads to improvements in the model's validation accuracy and reduced overfitting, as evidenced by smoother convergence of training and validation loss curves.

Image Preprocessing

Image preprocessing was used in preparing our raw images for input into our model. We applied various preprocessing techniques, such as resizing images to a consistent size of 224x224 pixels, rescaling pixel values to a 0 to 1 range, and normalizing images using the mean and standard deviation. These steps ensured consistent input size, reduced computational complexity, and improved model convergence.

ImageDataGenerator was employed for image preprocessing, ImageDataGenerator lazily applies custom preprocessing functions. Instead of preprocessing the entire dataset beforehand, it applies these steps on-the-fly as images are requested during training. This approach has both pros and cons. The main advantage is reduced memory usage, as preprocessed images aren't stored in memory. However, it also results in increased training time, as each image must undergo preprocessing as it's used.

Train-Validation-Test Split

To assess our model's performance and prevent overfitting, we split the dataset into two distinct sets: training and validation. We employed TensorFlow's ImageDataGenerator for this purpose, utilizing a 20% validation split. This configuration allocated 80% of the data for training and 20% for validation. This approach enables the model to learn from a substantial portion of the data while still reserving a suitable sample for validation purposes.

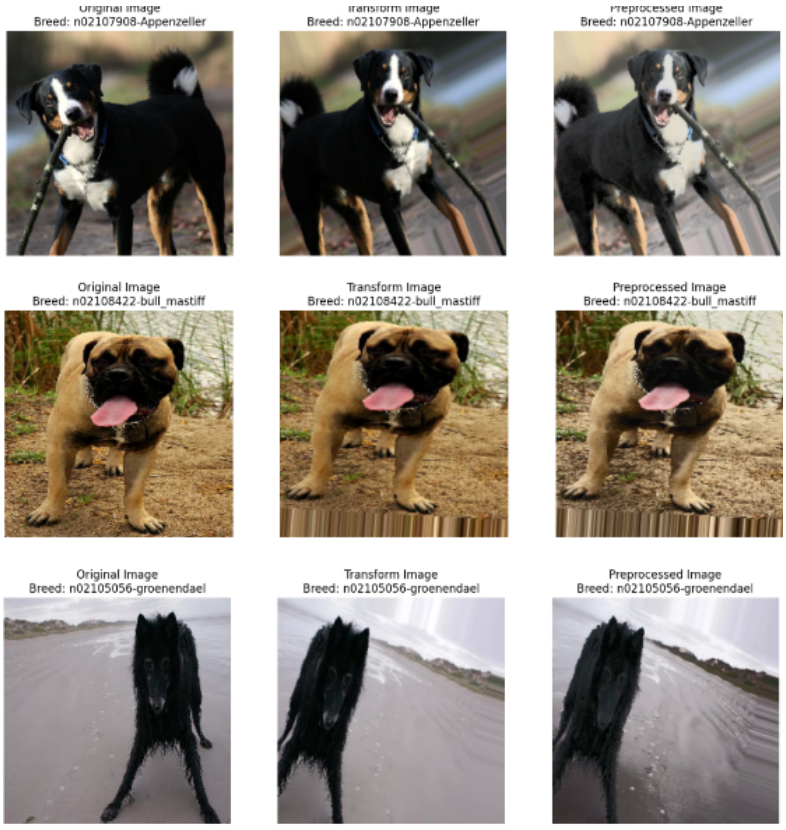

Figure 1: Raw image data and the effects of data augmentation and preprocessing

Methodology

Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a category of deep learning models that have demonstrated great success in image recognition and classification tasks. CNNs are especially adept at handling grid-like data, such as images, due to their capacity to capture local patterns and hierarchical features.

The core components of CNNs are their convolutional layers, which utilize filters to process input data and detect specific patterns or features. These filters learn to identify various low-level features like edges, textures, and colors, which can be combined to create more intricate, high-level features.

Pooling layers serve to decrease the spatial dimensions of the feature maps, effectively summarizing and compressing the information. This reduction aids in enhancing the model's computational efficiency and reducing the likelihood of overfitting.

Our model comprises several convolutional and pooling layers arranged hierarchically, followed by fully connected layers that generate the final class probabilities. We experimented with various architectures and hyperparameters to determine the optimal configuration for our dataset.

Transfer Learning

Transfer learning is an approach in deep learning where a pre-trained model, which has been typically developed for a related task using a substantial dataset, is fine-tuned to perform a specific task. The

premise behind transfer learning is that the pre-trained model has already learned valuable features from the larger dataset, and these features can be adapted to enhance the model's performance on the new task.

For this project in addition to our custom model, we developed a model utilizing transfer learning. We used a pre-trained base model, specifically ResNet50V2, which was originally trained on the ImageNet dataset. ImageNet is a large-scale dataset containing millions of images and thousands of classes.

By harnessing the power of transfer learning, we significantly reduced training time and achieved improved performance compared to our custom CNN.

Model Architecture

Custom CNN Model

Our custom CNN model consists of three main convolutional blocks, followed by a fully connected layer for classification. Each convolutional block contains a series of Conv2D layers with increasing filter sizes, BatchNormalization layers for improved convergence, MaxPooling2D layers for spatial downsampling, and Dropout layers for regularization.

Block 1: The first block consists of two Conv2D layers, each with 64 filters and a kernel size of 3x3. The input shape is set to match the image size and color channels. The activation function is ReLU, and padding is set to 'same' to preserve spatial dimensions. BatchNormalization is applied after each convolutional layer, followed by MaxPooling2D with a pool size of 2x2 and Dropout with a rate of 0.25.
Block 2: The second block is similar to the first, with two Conv2D layers, each having 128 filters and a kernel size of 3x3. BatchNormalization, MaxPooling2D, and Dropout are applied in the same manner as in Block 1.
Block 3: The third block contains two Conv2D layers with 256 filters and a kernel size of 3x3. BatchNormalization, MaxPooling2D, and Dropout are applied as in the previous blocks.

After passing through the three convolutional blocks, the output is flattened and connected to a Dense layer with 512 nodes and ReLU activation. BatchNormalization and Dropout with a rate of 0.5 are applied to prevent overfitting. Finally, a Dense layer with 120 nodes and softmax activation is used to produce the class probabilities.

The combination of Conv2D, BatchNormalization, MaxPooling2D, and Dropout layers allows the model to capture complex patterns and spatial relationships while minimizing overfitting and ensuring efficient training.

Pre-trained ResNet50V2 Model

Our transfer learning model leverages the pre-trained ResNet50V2 model as the base, which has been trained on the ImageNet dataset. This allows our model to take advantage of the learned features from a much larger and more diverse dataset, reducing training time and improving overall performance.

Base model: We loaded the ResNet50V2 model with pre-trained ImageNet weights, excluding the top (classification) layer. The input shape is set to match our dataset's image size and color channels.
Freezing the base model: To preserve the learned features, we set the base model layers as non-trainable, preventing weight updates during training and retaining generalization capabilities.
Custom top layers: We added custom layers to the pre-trained ResNet50V2 base model to adapt it to our dog breed classification task. The added layers include:

- GlobalAveragePooling2D: This layer reduces the spatial dimensions of the output from the base model, converting it into a 1D tensor.

- Dense layer: A fully connected layer with 512 nodes and ReLU activation learns higher-level features specific to our dataset.
- Dropout layer: A dropout layer with a rate of 0.5 is included to prevent overfitting.
- Dense layer: The final dense layer has 120 nodes and softmax activation, corresponding to the 120 dog breed classes.

Utilizing the pre-trained ResNet50V2 model and adding custom top layers allows this model to more efficiently classify dog breeds by leveraging the powerful feature extraction capabilities of the base model trained on a larger and more diverse dataset.

Regularization Techniques

To enhance the generalization capabilities of our models and mitigate overfitting, we implemented several regularization techniques. These techniques help the models to perform better on unseen data by preventing them from fitting too closely to the training data.

Dropout: We included dropout layers in both the custom CNN model and the transfer learning model. Dropout is a regularization technique that randomly sets a fraction of input neurons to zero during training. This prevents the model from relying too heavily on any single neuron and encourages the development of more robust features.

Batch normalization: In the custom CNN model, batch normalization layers are added after each convolutional layer. Batch normalization is a technique that helps to speed up training and improve model performance by normalizing the activations of each layer.

Early stopping: We employed learning rate scheduling to dynamically adjust the learning rate during training. This technique helps the model converge faster at the beginning of training by using a higher learning rate and then gradually reducing the learning rate as training progresses. This allows the model to make finer adjustments to the weights and achieve better performance.

Learning rate scheduling: We employed learning rate scheduling to dynamically adjust the learning rate during training. This technique helps the model converge faster at the beginning of training by using a higher learning rate and then gradually reducing the learning rate as training progresses. This allows the model to make finer adjustments to the weights and achieve better performance.

For both models, our learning rate starts at 0.001. Our scheduling has a patience of three, so if the model does not improve after 3 epochs, the learning rate will be reduced to a factor of 0.1. Our minimum learning rate is set to 0.00001, giving us two total steps down in learning rate. This makes our learning rate go as 0.001 -> 0.0001 -> 0.00001.

Model checkpoint: Checkpointing is a technique that saves the model weights at specific intervals or when certain performance criteria are met. In our case, we saved the model with the best validation loss during training. This ensures that we retain the model with the best generalization performance on the validation set, rather than the last model state which may have started to overfit.

By combining these regularization techniques and model checkpointing, our models are better able to generalize and avoid overfitting, resulting in more accurate predictions.

Experimental Setup

Hardware and Software

Hardware

For this project, we utilized Google Colab, a cloud-based platform that provides a collaborative environment for executing Jupyter notebooks and running machine learning models. Google Colab Pro offers enhanced resources compared to the free version, which includes access to high-performance GPUs, more memory, and longer runtime.

The primary hardware component used for training our models was the NVIDIA Tesla T4 GPU. The Tesla T4 is a high-performance GPU designed for deep learning and AI applications, featuring 16 GB of GDDR6 memory and providing a peak performance of 8.1 TFLOPS for single-precision computations. This GPU allowed us to train and evaluate our models efficiently and effectively while reducing the overall training time.

Software

In this project, Python 3.8 was chosen as the scripting language because it is widely adopted and has extensive libraries and tools available for machine learning and data analysis. TensorFlow 2.6, with its integrated Keras API, was selected as the deep learning framework for its scalability, efficiency, and ease of use.

Keras is a high-level neural network API that was initially developed as a user-friendly interface for building deep learning models. It was integrated into TensorFlow as the official high-level API starting from TensorFlow 2.0. Keras provides a simple and intuitive way to define, train, and evaluate neural networks.

The Jupyter Notebook environment was used to facilitate interactive experimentation and data visualization, enabling us to iteratively develop, test, and refine the models throughout the project. Jupyter Notebook's ability to combine code, visualizations, and narrative text in a single document made it an ideal platform for documenting the project's progress.

Model Evaluation Metrics

To evaluate our models, we used various metrics for accuracy and generalization, calculated during training for both datasets. Monitoring these metrics allowed us to make informed decisions about model architecture, hyperparameter tuning, and training strategies.

Categorical Cross-Entropy Loss: Our primary loss metric, suitable for multi-class classification, measures the discrepancy between predicted probabilities and true labels. Lower values indicate better performance.
Accuracy: A widely-used metric, it represents the proportion of correct predictions out of the total, providing a general understanding of model performance.

Training Procedures

Data Preparation: We preprocess and augment the dataset, resizing images, normalizing pixel values, and applying data augmentation techniques for better generalization.
Model Initialization: We initialized our custom CNN model and the transfer learning model using pre-trained ResNet50V2.
Hyperparameter Selection: We set our hyperparameters like learning rate (0.001), batch size (32), and epochs (50) based on prior knowledge and experiments.
Model Compilation: Both models were compiled using the Adam optimizer, categorical cross-entropy loss, and accuracy.
Training Loop: Models are trained using the prepared dataset, with validation data used to monitor performance. Several callback functions such as early stopping, learning rate scheduling, and model checkpointing.
Model Evaluation: Model performance is assessed for both models. allowing us to determine the model's generalization ability and compare the performance of the custom CNN model and the transfer learning model.

Results and Discussion

Model Performance

In this section, we present the results of our custom CNN model and transfer learning mode.. We compare their performance in terms of training and validation loss, and accuracy.

Custom CNN Model:

- Training Loss: 2.2703
- Validation Loss: 2.4858
- Training Accuracy: 39.24%
- Validation Accuracy: 37.23%
- Early stopping: 45/50 epochs

With our custom CNN, we can see an overall low performance both in loss and accuracy. This was our best performing model, with more complex models performing worse over 50 epochs. This low performance shows a lack in our models ability to generalize and learn.

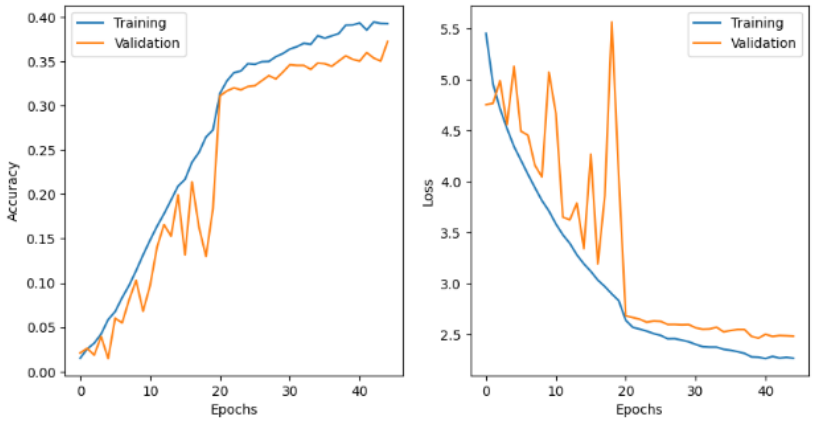

Figure 2. Custom CNN model loss and accuracy over epochs

Transfer Learning Model with ResNet50V2:

- Training Loss: 1.0597
- Validation Loss: 1.1893
- Training Accuracy: 68.60%
- Validation Accuracy: 66.92%
- Early Stopping: 19/50 epochs

Comparing the custom CNN and transfer learning models, the transfer learning model with pre-trained ResNet50V2 performs better in terms of accuracy and loss. This showcases the power of transfer learning in using pre-trained models for feature extraction and classification, even with differing target and source domains.

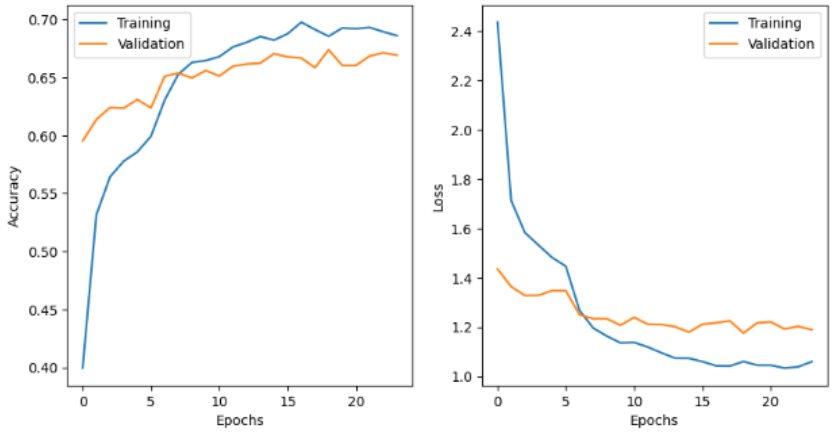

Figure 3. Transfer learning model loss and accuracy over epochs

Performance Analysis

Low Performance of our custom model can be attributed to multiple sources:

Model Complexity: Our custom CNN may lack the necessary depth and complexity to effectively learn dog breed features compared to the more advanced architecture of ResNet50V2. However, we are limited in our computation power and dataset size.
Regularization Techniques: Additional regularization, such as weight decay, could reduce overfitting and improve generalization for the custom model.
Hyperparameter Optimization: Fine-tuning hyperparameters, like learning rate or batch size, may improve training efficiency and convergence.
Data Augmentation Strategies: Advanced augmentation techniques (e.g. random rotations or color jittering) could enhance feature extraction and generalization.

Addressing these limitations may improve the custom model's performance, but we faced constraints in computational power, dataset size, and time for this project.

Our model was limited by the NVIDIA Tesla T4 GPU 16GB provided by Google Colab Pro. Upgrading to a more advanced GPU, like the NVIDIA A100 40GB, would increase costs significantly, and migrating to Google Cloud had comparable expenses. As such our custom model's complexity was restricted by computational resources and time constraints.

The lower performance of our custom CNN model, compared to the ResNet50V2 transfer learning model, might also be due to the limited dataset size. Deep learning models typically require extensive labeled data to achieve optimal performance, and a limited dataset might not provide enough diversity for effective feature representation and generalization.

Transfer learning, however, takes advantage of pre-trained models on large-scale datasets like ImageNet, which have learned robust feature representations. Fine-tuning these pre-trained models on our dataset allows us to leverage these features and achieve better performance, even with smaller amounts of data, which is beneficial in cases like ours.

Conclusion

This project aimed to create a dog breed classification model using convolutional neural networks. We investigated two approaches: constructing a custom CNN from scratch and employing transfer learning with a pre-trained ResNet50V2 model. The transfer learning model significantly outperformed the custom CNN, emphasizing the benefits of using pre-trained models when dealing with limited data and computational resources.

The custom CNN model's performance may have been influenced by factors such as dataset size, data quality, and available computational power. Future work could focus on augmenting the dataset, optimizing the model's architecture, and incorporating additional regularization techniques to enhance performance. Increasing computational resources could also enable more extensive experimentation with model complexity and training parameters.

In conclusion, this project showcased the potential of deep learning techniques in image classification tasks, specifically dog breed classification. It also highlighted the significance of data quality, model selection, and computational power in achieving optimal model performance.

Acknowledgements

I would like to thank Dr. Yunxin Zhao for teaching this course and allowing me to be a part of it. It was one of the most informative and challenging courses I have taken, and I have gained much from it.

Stanford Dogs Dataset, the original data source is found on http://vision.stanford.edu/aditya86/ImageNetDogs/ and contains additional information on the train/test splits and baseline results.

References

1. Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao and Li Fei-Fei. Novel dataset for Fine-Grained Image Categorization. First Workshop on Fine-Grained Visual Categorization (FGVC), IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2011. (Stanford Dogs Dataset)
2. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, ImageNet: A Large-Scale Hierarchical Image Database. IEEE Computer Vision and Pattern Recognition (CVPR), 2009. (Stanford Dogs Dataset)
3. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems (pp. 1097-1105).
4. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.
5. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 1-9).
6. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 770-778).
7. Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 1251-1258).
8. Tan, M., & Le, Q. V. (2019). EfficientNet: Rethinking model scaling for convolutional neural networks. arXiv preprint arXiv:1905.11946.
9. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., ... & Adam, H. (2017). MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.
10. Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In 2009 IEEE Conference on Computer Vision and Pattern Recognition (pp. 248-255). IEEE.
11. Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on Image Data Augmentation for Deep Learning. Journal of Big Data, 6(1), 60.
12. TensorFlow. (n.d.). ImageDataGenerator. Retrieved from https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator