**Building a Cat vs. Dog Classifier Using Keras and TensorFlow**

**Overview**

Building a deep learning model to classify images of cats and dogs. By uses the **TensorFlow** and **Keras** libraries, specifically leveraging **transfer learning** with the **ResNet50** model, a powerful convolutional neural network (CNN) pre-trained on the ImageNet dataset.

**Prerequisites**

**Deep Learning Concepts**: Layers, activation functions, CNNs, and transfer learning.

**Python Programming**: Familiarity with Python syntax and basic libraries.

**Section 1: Importing Required Libraries and Modules**

A screen shot of a computer

Description automatically generated

Each library and module here serves a specific purpose in building, training, and evaluating our model:

**General Libraries**:

* os: Helps manage file paths and directories, useful for organizing data.
* matplotlib and seaborn: Provide tools for plotting images and data visualizations, helping us understand model performance.

**TensorFlow & Keras**:

**TensorFlow**: The main deep learning framework we’re using. TensorFlow provides efficient ways to manage and run neural networks.

**Keras Layers** (layers, Dense, Dropout, Conv2D, etc.): These are the building blocks of neural networks. We use different layers to construct our model, including convolutional layers for feature extraction and dense layers for classification.

**Optimizers and Callbacks**: Modules like optimizers and ReduceLROnPlateau are crucial for improving the model’s training efficiency.

**Pre-trained Models** (VGG16, ResNet50): We use **ResNet50**, a pre-trained model, for transfer learning. Pre-trained models on ImageNet allow us to start with learned weights, significantly improving training time and performance with less data.

**TensorFlow and Keras**: These frameworks are optimized for deep learning, making it easy to build, train, and scale models. Keras, in particular, provides a simple API for defining layers and models.

**Transfer Learning with Pre-trained Models**: Using models like ResNet50 enables us to leverage existing knowledge, improving model accuracy without needing extensive training from scratch.

**Section 2: Data Preparation and Organization**

**2.1 Unzipping and Organizing Dataset**

A computer screen with text

Description automatically generated

Organizes images into 'dogs' and 'cats' folders within the training directory. This structure allows image\_dataset\_from\_directory to easily infer class labels from directory names, simplifying data loading.

**Section 3: Loading and Preparing the Dataset**

**3.1 Creating the Training Dataset**

A screenshot of a computer

Description automatically generated

Loads the training images into a format ready for training.

•image\_size: Resizes all images to 256x256 pixels.

•batch\_size: Sets batch size to 64 images, balancing memory efficiency with processing speed.

•shuffle=True: Randomizes images each epoch, preventing overfitting by changing the order of data presented to the model.

**3.2 Creating the Validation and Test Splits**

A screen shot of a computer program

Description automatically generated

Splits 20% of the data for validation, leaving the remaining 80% for training. This split allows us to monitor the model’s performance on unseen data during training.

A screen shot of a computer

Description automatically generated

calculates the number of batches in the validation\_ds dataset and store the number of batches in validation\_ds

takes the first 20% of the validation\_ds dataset and assigns it to test\_df. 20% of the validation data is now being used as a test set

assigns the remaining 80% to validation\_df, which will be used as the actual validation set. will be used during training to monitor the model’s performance

**3.3 Optimizing Dataset Performance**

**A screen shot of a computer

Description automatically generated**

Caches and prefetches batches for faster data loading, reducing waiting times during training.

**Caching**: Saves the dataset in memory after its first access, making it faster to load in subsequent epochs.

**Prefetching**: Loads the next batch of data while the current batch is being processed, reducing latency and keeping the GPU busy.

**Why Cache and Prefetch?**

Optimize data loading and processing, ensuring the model can access data quickly without bottlenecks, leading to more efficient training and evaluation.

**Section 4: Data Augmentation**

**A screen shot of a computer program

Description automatically generated**

Applies random transformations to each image, such as rotation, flipping, zooming, contrast, and brightness adjustments. Data augmentation enhances model robustness by exposing it to various transformations, which helps prevent overfitting.

add a RandomRotation layer that randomly rotates each image by up to 30% (0.3) of 360 degrees.

Add a RandomFlip layer that randomly flips images both horizontally and vertically.

add RandomZoom layer that randomly zooms in on the image by up to 10% (0.1).

Add a RandomContrast layer that randomly changes the contrast of the image by up to 20%.

Add a RandomBrightness layer that randomly adjusts the brightness of the image by up to 20%.

Each augmented photo looks a bit different, doing the augmentation helps the model become better at recognizing cats and dogs in all kinds of real-world situations.

**Section 5: Building the Model (ResNet50 Architecture)**

**A screen shot of a computer program

Description automatically generated**

Defines the size of each input image as 256x256 pixels. A standard size across all images, and 256x256 is large enough to retain meaningful image details for classification without being overly computationally intensive.

Adds the color channel dimension, setting image\_shape to (256, 256, 3). The 3 represents the RGB channels, specifying that the input images are in color. This is essential for the model to understand the input format.

Sets up a preprocessing function specific to ResNet50, which normalizes input images as expected by this model. ResNet50, trained on ImageNet, expects images in a specific format (pixel values normalized to a range suitable for ResNet). Using preprocess\_input ensures the images are standardized for optimal performance.

Loads the pre-trained ResNet50 model with:

•input\_shape=image\_shape: Accepts images of shape (256, 256, 3).

•include\_top=False: Excludes the original fully connected layers that classify ImageNet’s 1,000 classes.

•weights='imagenet': Initializes with weights learned from the ImageNet dataset.

Using pre-trained weights lets us leverage this knowledge, reducing the need for a large dataset or long training time. By excluding the top layers, we can add our own output layers tailored for the binary classification task (cats vs. dogs).

Sets nclass to the number of classes (2 in this case, for cats and dogs). Defines the number of neurons in the output layer, ensuring it matches the number of categories we want to classify.

Adds a global average pooling layer.

**Dimensionality Reduction**: Global average pooling reduces each feature map to a single value by averaging, resulting in a smaller, more generalized feature vector.

**Less Overfitting**: Compared to fully connected layers, this pooling approach has fewer parameters, which reduces the risk of overfitting.

**Interpretability**: Global average pooling tends to produce features that are less sensitive to small image transformations.

Adds a dense layer with nclass neurons (2, one for each class) and softmax activation to output class probabilities. This layer interprets the feature vector output by ResNet50, assigning probabilities to each class (cat or dog). Softmax is ensure the output values sum to 1, making them interpretable as probabilities.

Defines the input layer for the model, specifying the expected shape (256, 256, 3). Sets up the model’s starting point, clearly defining the input size and structure.

Applies the ResNet-specific preprocessing to the input images, normalizing pixel values. Standardizing the images helps ensure they match the input format ResNet50 was trained on, improving the transfer learning effectiveness.

Applies data augmentation (random rotations, flips, zooms, etc.) to the images. Augmentation increases the variety of images seen during training, making the model more robust and less prone to overfitting.

Passes the preprocessed, augmented data through the ResNet50 base model. This is where the core feature extraction happens, with ResNet50 identifying patterns and features within each image.

Applies global average pooling to the output of ResNet50, reducing each feature map to a single averaged value. It condenses the complex feature maps from ResNet50 into a smaller, manageable feature vector for the dense layer to interpret.

Produces the final output layer, which assigns a probability to each class (cat or dog) based on the features extracted. Converts the feature vector into class predictions, making the output ready for interpretation

Creates the complete model by connecting the input layer, all intermediate layers, and the output layer. defines the end-to-end structure of the model and ready to train!!!!!!!!!!!!!!!!!

A screenshot of a computer program

Description automatically generated

**input\_layer\_1**:**Type**: InputLayer. **Shape**: (None, 256, 256, 3)

**Preprocessing Steps (GetItem, Stack, Add)**

These layers manipulate and prepare the input data for further processing by the model, such as stacking the channels or manipulating the image data in a particular way.

**Data Augmentation Layer**

sequential: This layer refers to the data augmentation pipeline that was defined earlier. It includes random rotations, flips, zooms, and other transformations to the input images.

**Base Model (ResNet50)**

**Type**: Functional **Output Shape**: (None, 8, 8, 2048) It transforms the input image into a feature map of size (8, 8, 2048).

**Global Average Pooling Layer**

This layer reduces the spatial dimensions (from (8, 8, 2048) to just (2048) by taking the average of each feature map. This step reduces the number of parameters while still retaining important information from the feature maps.

**Dense Layer**

This is the final classification layer with 2 units (for 2 classes). It uses **softmax activation** to output probabilities for each class.

**Total Parameters**: 23,591,810

This includes all parameters in the model (both trainable and non-trainable).

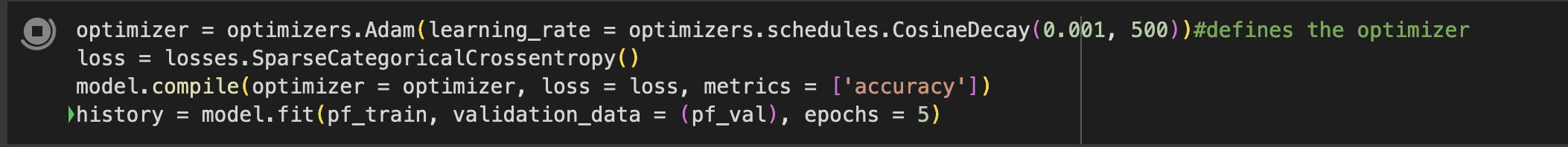
**Trainable Parameters**: 23,538,690

These are the parameters that will be updated during training. Most of them come from the ResNet50 model.

**Non-Trainable Parameters**: 53,120

These are parameters that won’t be updated during training. They might come from certain layers in ResNet50 that are frozen or from batch normalization layers.

**Section 6: Compiling and Training the Model**



Adam **Optimizer**: The Adam (Adaptive Moment Estimation) optimizer is a commonly used optimization algorithm in deep learning. It adapts the learning rate for each parameter during training, which typically results in faster convergence and better performance.

**Cosine Decay**: This applies a **learning rate schedule** that decreases the learning rate following a cosine curve over the course of training.

**Initial Learning Rate (**0.001**)**: The initial learning rate is set to 0.001, which is a common starting value for Adam.

**Decay Steps (**500**)**: The learning rate decreases over 500 steps (or epochs, depending on how you define it). The cosine decay reduces the learning rate more gently compared to traditional step decay, helping the model train more smoothly and potentially achieve better performance.

**Why Use Cosine Decay?**

As training progresses, lowering the learning rate allows the model to fine-tune more precisely without overshooting. Cosine decay smooths this process by reducing the learning rate gradually.

We’re starting with a learning rate of 0.001, which gradually decreases following a “cosine” curve over 500 steps. A gradually decreasing learning rate helps the model “fine-tune” itself as training progresses.

**loss = losses.SparseCategoricalCrossentropy()**

defines the **loss function**, which measures how well the model’s predictions match the actual labels during training. classification tasks with multiple classes (cat and dog), It’s appropriate when the labels are provided as integers (e.g., 0 for “cat” and 1 for “dog”) rather than one-hot encoded vectors.

In regular CategoricalCrossentropy, labels are one-hot encoded vectors. In “sparse” format, the labels are single integers, and TensorFlow automatically handles the conversion internally.

Model Compile

compiles the model, specifying the optimizer, loss function, and performance metrics to track during training. Compiling the model tells TensorFlow which optimization algorithm (optimizer), loss function, and evaluation metrics to use during training. During training and validation, the model will display the accuracy to provide feedback on how well it is performing.

**history = model.fit(pf\_train, validation\_data=pf\_val, epochs=5)**

trains the model using the training dataset and validates it using the validation dataset for a specified number of epochs.

pf\_train: This is the preprocessed and prefetched training dataset.

validation\_data=pf\_val: The model will evaluate its performance on the validation set (pf\_val) after each epoch to track how well it generalizes to unseen data.

epochs=numbers: The model will go through the entire training dataset how many times during training.

A screenshot of a math equation

Description automatically generated

**Final Training Accuracy**: 83.97%

**Final Validation Accuracy**: 83.91%

**Section 7: Evaluating Predictions**

A computer screen shot of a program code

Description automatically generated

A screenshot of a dog

Description automatically generatedwrong case !