# Building and Training a Simple CNN for Image Classification

Kaiser Hamid

December 16, 2024

## Contents

# 1 Introduction

In this article, we will guide you through the process of loading a small image dataset and training a simple Convolutional Neural Network (CNN) for image classification. We will break the code into multiple chunks, each followed by an explanation to ensure clarity.

Before we begin, let's assume your dataset is organized as follows:

```
dataset/
    train/
        dog/
            dog001.jpg
            dog002.jpg
            ...
        cat/
            cat001.jpg
            cat002.jpg
            ...
    val/
        dog/
            dog101.jpg
            dog102.jpg
            ...
        cat/
            cat101.jpg
            cat102.jpg
            ...
```

In this example, we have two classes: `dog` and `cat`. The `train/` directory holds training images, and the `val/` directory holds validation images. The number of images in `val/` determines the size of your validation set. For example, if `val/` contains 200 images total, your validation set size is 200.

# 2 Loading and Preprocessing the Data

We will use Keras' `ImageDataGenerator` to load images from the directories. This utility will:

- Resize images to a uniform size (here, $150 \times 150$ pixels).
- Normalize pixel values to a common range.

- Apply data augmentation for the training set to help the model generalize.
- Produce images in batches, making training memory-efficient.

## Chunk 1: Imports and Paths

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import
   ImageDataGenerator
import os

# Set paths for training and validation datasets
train_dir = 'dataset/train'
val_dir = 'dataset/val'
```

**Explanation:**

Here we import TensorFlow and Keras components. We also set the directory paths where our training and validation images are located.

## Chunk 2: Image Loading Parameters

```
# Define parameters for image loading
img_height = 150
img_width = 150
batch_size = 32
```

**Explanation:**

All images will be resized to $150 \times 150$ pixels. We choose a batch size of 32 to feed images to the model in small groups.

## Chunk 3: Data Generators

```
# Create ImageDataGenerators for training and validation
    sets
train_datagen = ImageDataGenerator(
    rescale=1./255,      # Normalize pixel values to
       [0,1]
    rotation_range=20,   # Random rotations up to 20
       degrees
    width_shift_range=0.2,
```

```
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

val_datagen = ImageDataGenerator(rescale=1./255)
```

**Explanation:**

For the training set, we apply data augmentation (rotations, shifts, flips, zoom) to help the model learn robust features. For the validation set, we only rescale pixel values, ensuring validation images are a stable measure of performance (no augmentation).

## Chunk 4: Creating Iterators

```
train_generator = train_datagen.flow_from_directory(
    train_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='binary' # 'binary' since we have two
       classes (dog and cat)
)

val_generator = val_datagen.flow_from_directory(
    val_dir,
    target_size=(img_height, img_width),
    batch_size=batch_size,
    class_mode='binary'
)
```

**Explanation:**

The `flow_from_directory` method scans the given directories and infers class labels from the folder names. It then generates batches of image data and corresponding labels.

The validation set size corresponds to the total number of images in the `val/` directory. For example, if there are 200 images in `val/`, the validation set size is 200 images.

# 3 Defining and Compiling the Model

We define a simple CNN that includes convolutional layers to extract features, pooling layers to reduce spatial dimensions, and a fully connected layer to perform classification.

## Chunk 5: Defining the CNN Architecture

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3,3), activation='relu',
                        input_shape=(img_height,
                           img_width, 3)))
model.add(layers.MaxPooling2D((2,2)))

model.add(layers.Conv2D(64, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))

model.add(layers.Conv2D(128, (3,3), activation='relu'))
model.add(layers.MaxPooling2D((2,2)))

model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

**Explanation:**
We start with a small CNN:

- The first layer uses 32 filters and a $3 \times 3$ kernel.
- Max pooling reduces the size of the feature maps.
- We add more convolutional layers with increasing filter counts (64, then 128).
- After flattening, we use a dense (fully connected) layer with 128 neurons.
- The final layer has 1 output neuron with a sigmoid activation, which gives a probability of one class versus the other.

## Chunk 6: Compiling the Model

```
model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
```

**Explanation:**

We use binary crossentropy loss for a two-class problem, the Adam optimizer for parameter updates, and track accuracy as our metric.

# 4 Training the Model

## Chunk 7: Training the Model

```
epochs = 10
history = model.fit(
    train_generator,
    epochs=epochs,
    validation_data=val_generator
)
```

**Explanation:**

We train the model for 10 epochs. After each epoch, the model is evaluated on the validation set, allowing us to track overfitting or improvements in generalization.

# 5 Evaluating and Saving the Model

## Chunk 8: Evaluating and Saving the Model

```
acc = history.history['accuracy'][-1]
val_acc = history.history['val_accuracy'][-1]
print(f"Final Training Accuracy: {acc:.2f}, Validation
   Accuracy: {val_acc:.2f}")

model.save('simple_cnn_dog_cat.h5')
```

**Explanation:**

We print the final training and validation accuracies. Saving the model allows us to load it later for inference or further training.

# 6 Summary of the Process

- We organized the dataset into a directory structure with separate folders for each class under `train/` and `val/`.
- We used `ImageDataGenerator` to load and preprocess images, applying data augmentation to the training set.
- We defined a simple CNN and compiled it with a suitable loss, optimizer, and metric.
- We trained the model while monitoring validation performance.
- The validation size is determined by how many images are placed in the `val/` directory.
- Finally, we evaluated, printed results, and saved the trained model.

This approach provides a foundational understanding of building and training a CNN for image classification tasks.