Data Augmentation and Convolutional Neural Networks

Archit Gupta, M.Sc. Mathematics

Probability and Mathematical Statistics

June 29, 2019

Introduction

This is my humble attempt to build a CNN network to classify images. The sole purpose of writing this report is for me to learn thoroughly, to share with other people and also put together information from different sources on the internet at one place. My work is heavily influenced and motivated by the following webpage/s:

1. Building powerful image classification models using very little data.

Chapter 1

Introduction to the dataset

While building an image classifier, we usually face the issue of not having enough training examples. In such cases we might be able to use what is called *data augmentation*. In this article, I will discuss implementation of the technique and compare the accuracy of the model to when we do not apply data augmentation. For this purpose I work with *Dogs vs Cats* dataset available on Kaggle. The original dataset contains 12500 images of cats and dogs each. While I use the entire dataset to build a model when I don't use data augmentation, I consider only 1000 images for each class for building a model with data augmentation and use 400 images each for validation set.

In the following sections, I will briefly discuss implementation of CNN without data augmentation - for gray-scale images and color images both. These models will act as a benchmark before I move on to discuss a model with augmented data. I will also consider different model architecture for color images and study the relative performance. For implementation of this model, please follow the script *catsDogsOriginal.py* at my github repository GitHub.

1.1 Model 1

M1 is the model where I consider only gray-scale images. Following was the model architecture.

Layer	Output Shape	Param #
conv2d (Conv2D)	(None, 48,48, 64)	640
activation (Activation)	(None, 48,48, 64)	
max_pooling2d (MaxPooling2D)	(None, 24,24,64)	
conv2d_1 (Conv2D)	(None, 22,22,64)	36928
activation_1 (Activation)	(None, 22,22, 64)	
max_pooling2d_1 (MaxPooling2D)	(None, 11,11,64)	
flatten (Flatten)	(None, 7744)	0
dense (Dense)	(None, 64)	495680
dense_1 (Dense)	(None, 1)	65
activation_2 (Activation)	(None,1)	

The Total and Trainable parameters are 533 and 313 respectively.

The validation set accuracy of the model with 10 epochs with batch_size = 32 was 79.16%. Following were observed numbers:

Epoch	Loss	Accuracy	Validation Loss	Validation accuracy
1	0.6026	66.88	0.5469	72.87
2	0.5085	75.14	0.5051	75.39
3	0.4702	78.12	0.4691	78.36
⋮	⋮	⋮	⋮	⋮
8	0.3455	84.83	0.4431	79.64
9	0.3309	85.56	0.4619	79.96
10	0.3140	86.35	0.4562	79.16

1.2 Model 2

M2 is the model where I consider color images. Following was the model architecture.

Layer	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 48,48, 64)	1792
activation_3 (Activation)	(None, 48,48, 64)	
max_pooling2d_2 (MaxPooling2D)	(None, 24,24,64)	
conv2d_3 (Conv2D)	(None, 22,22,64)	36928
activation_4 (Activation)	(None, 22,22, 64)	
max_pooling2d_3 (MaxPooling2D)	(None, 11,11,64)	
flatten_1 (Flatten)	(None, 7744)	0
dense_2 (Dense)	(None, 64)	495680
dense_3 (Dense)	(None, 1)	65
activation_5 (Activation)	(None,1)	

The Total and Trainable parameters are 534 and 465 respectively. The validation set accuracy of the model with 10 epochs with batch_size = 32 was 78.60%. Following were observed numbers:

Epoch	Loss	Accuracy	Validation Loss	Validation accuracy
1	0.6331	63.48	0.5550	72.22
2	0.5242	74.09	0.4998	76.35
3	0.4818	76.82	0.4801	77.84
⋮	⋮	⋮	⋮	⋮
8	0.3665	83.55	0.4635	79.56
9	0.3460	84.76	0.4607	79.72
10	0.3245	85.72	0.4747	78.60

Chapter 2

Imge classification model using very little data

2.1 Data pre-processing and data augmentation

In order to make the most out of small number of training examples, I will 'augment' them via a number of random transformations, so that my model would never see twice the same picture. This helps prevent overfitting and helps the model generalize better. In Keras this can be done using keras.preprocessing.image.ImageDataGenerator class. Let us start generating some pictures and save them to a temporary directory.

```
from keras.preprocessing.image import
                                    ImageDataGenerator
datagen = ImageDataGenerator(
            rotation_range=40,
            width_shift_range = 0.2,
            height_shift_range = 0.2,
            rescale = 1./255,
            shear_range = 0.2,
            zoom_range = 0.2,
            horizontal_flip = True,
            fill_mode = 'nearest')
# upload a PIL image
img = load_img('Cat/0.jpg')
```

```
x = img_to_array(img) # numpy array (3,150,150)
x = x.reshape((1,) + x.shape) # numpy (1,3,150,150)

# the .flow() command generates batches of randomly
# transformed images and saves the results to the
# 'preview/' directory.

for batch in datagen.flow(x, batch_size = 1,
                          save_to_dir = 'preview',
                          save_prefix = 'cat',
                          save_format = 'jpeg')
    i+= 1
    if i >20:
        break
```

The above code will take on image, in this case *Cat/0.jpg* and with .flow command we create 21 images that are randomly transformed and saved in directory Preview.

2.2 Basic model with augmented data

2.2.1 Data augmentation

I train a CNN with same architecture as in Chapter 1 but with augmented data and on a much smaller set. As discussed in Chapter 1, I will only use 1000 images for each category to train on and 400 each for validation. In order to prepare the dataset, I have written a script to prepare the subset of data that we will train on. To follow please use the script *subsetImagesFromFolder.py* at my github repository GitHub. Note that in order to use the script without much changes on your computer, you must have the following directory structure on your computer. Let's say you folders *Cat* and *Dog* are located in director *data*. Then we will create following structure of directories for our purpose:

- data
  - data1000

    - train
        - cats
        - dogs
    - test
        - cats
        - dogs

Now that we have prepared our data, we will augment the data. For this purpose, we call the ImageDataGenerator as discussed above although we will use a slightly different generator in this case study.

2.2.2 Data augmentation and model architecture

In order to stay consistent, I first consider a model architecture as the one considered in Chapter 1 however now we train on a smaller dataset and on augmented data. Please open the script *dataAugmentationModel3.py* on my github repository GitHub to follow along. In this case, we are only building a model for **color** images.

Data augmentation

We will use different generators for train and test samples as follows.

```
batch_size = 16
# Use the following data augmentation configuration
train_datagen = ImageDataGenerator(
        rescale = 1./255,
        shear_range=0.2,
        zoom_range =0.2,
        horizontal_flip = True)

# For testing we use the following augmentation
test_datagen = ImageDataGenerator(rescale = 1./255)

# The following is generator that will read pictures
# found in the sub-folders 'data1000/train' and
# indefinitely generate batches of augmented image
# data.
```

```
train_generator = train_datagen.flow_from_directory
        ("data1000/train",
        target_size = (50,50),
        batch_size = batch_size,
        class_mode = "binary")

# similar generator for validation data
validation_generator =
        test_datagen.flow_from_directory
        ("data1000/test",
        target_size = (50,50),
        batch_size = batch_size,
        class_mode= "binary")
# we now use the generator to train our model
model3.fit_generator(
        train_generator,
        steps_per_epoch = 2000//batch_size,
        epochs = 10,
        validation_data = validation_generator,
        validation_steps = 8000//batch_size)
model3.save_weights('first_try.h5')
```

The architecture of the model, same as before, was as follows. The Total

Layer	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 48,48, 64)	1792
activation_9 (Activation)	(None, 48,48, 64)	
max_pooling2d_6 (MaxPooling2D)	(None, 24,24,64)	
conv2d_7 (Conv2D)	(None, 22,22,64)	36928
activation_10 (Activation)	(None, 22,22, 64)	
max_pooling2d_7 (MaxPooling2D)	(None, 11,11,64)	
flatten_3 (Flatten)	(None, 7744)	0
dense_6 (Dense)	(None, 64)	495680
dense_7 (Dense)	(None, 1)	65
activation_5 (Activation)	(None,1)	

and Trainable parameters are 534 and 465 respectively.

The validation set accuracy of the model with 10 epochs with batch_size = 32 was 68.25%. Following were observed numbers:

Epoch	Loss	Accuracy	Validation Loss	Validation accuracy
1	0.7006	50.80	0.6884	55.50
2	0.6844	56.40	0.6807	56.38
3	0.6789	57.60	0.6823	57.50
⋮	⋮	⋮	⋮	⋮
8	0.6362	63.10	0.7114	63.75
9	0.6178	67.05	0.6242	65.87
10	0.5986	67.20	0.6298	68.25

2.3 Training a small convnet

Data augmentation is one way to avoid overfitting, but it isn't enough since our augmented samples are still highly correlated. Main focus to avoid overfitting should be the entropic capacity of the model.

There are different ways to modulate entropic capacity. The main one is the choice of number of parameters in your model i.e. the number of layers and the size of each layer. Another way is to use weight regularization, such as L1 or L2 regularization.

In this case, we use small convnet with few layers and few filters per layer, alongside data augmentation and dropout. *Dropout* helps reduce overfitting by preventing a layer from seeing twice the exact same pattern, thus acting in a way analogous to data augmentation.

We create the first model as simple stack of 3 layers with ReLU activation and followed by max-pooling layers.