Image Classification for Dogs and Cats

Abstract

In this project, our task is to develop an algorithm to classify images of dogs and cats, which is the Dogs vs. Cats competition from Kaggle. We mainly investigated two approaches to address this problem.I used Deep Convolutional Neural Networks (CNN) to learn features of images and trained Back-propagation(BP) Neural Networks

Task Definition

basic task is to create an algorithm to classify whether an image contains a dog or a cat. The input for this task is images of dogs or cats from training dataset, while the output is the classification
accuracy on test dataset.
The given dataset for this competition is the Asirra dataset provided by Microsoft Research. Our training set contains 25,000 images, including 12,500 images of dogs and 12,500 images of cats, while the test dataset contains 12,500 images. The average size for these images is around 350×500.
Our learning task is to learn a classification model to determine the decision boundary for the training
dataset. The whole process is illustrated , from which we can see the input for the learning task is images from the training dataset, while the output is the learned classification model.
Our performance task is to apply the learned classification model to classify images from the test dataset, and then evaluate the classification accuracy the input is images
from the test dataset, and the output is the classification accuracy.

Solution

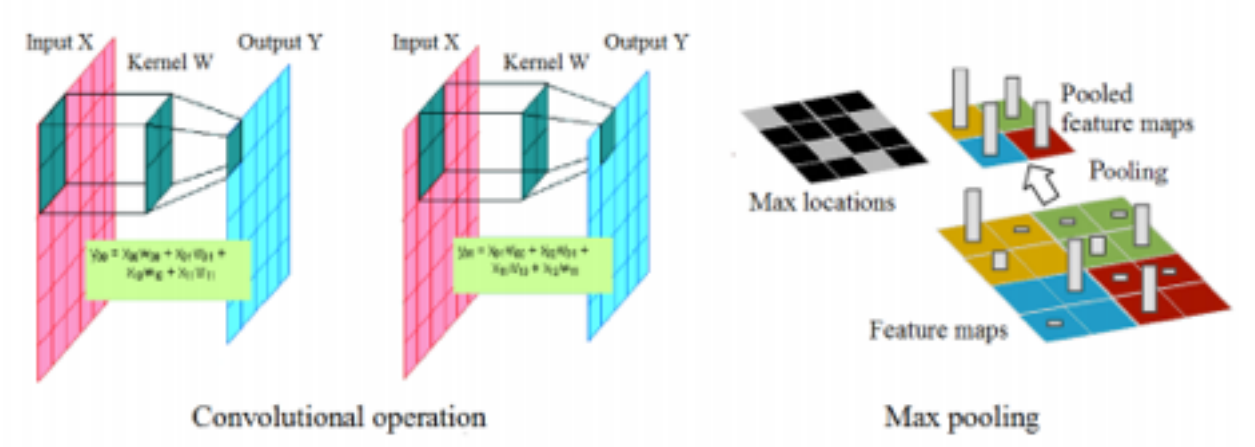
we applied a CNN to learn features. In terms
of classifiers, we mainly chose SVMs and BP Neural Networks, considering the high dimensional feature space for images. We tried various experiments to achieve high accuracy on the test dataset,
with different algorithms and parameter settings.

Using Features learned by Convolutional Neural Network

Our second approach is to learn features by the CNN [5][6] and then use BP Neural Network or SVM classifiers to train these features. Different from human crafted features which are fixed features
directly extracted from images, deep neural networks learn features from images, and discover multiple levels of representation, with higher-level features representing more abstract aspects of the
data

Deep Convolutional Neural Networks

The CNN is a kind of deep architecture which has achieved great performance in tasks like document
recognition and image recognition .
Different from traditional BP Neural Networks, which contains input layer, hidden layers and output layer, the CNN also contains Convolutional layers and Max Pooling layers.
Convolutional layers contain many feature maps, which are two dimensional hidden nodes. Every feature map owns a weight matrix called kernel, and different feature maps owns different kernels. Kernels do convolutional operation with every feature map in previous layer (layer j), then we sum them up and put it into sigmoid function. The output is the pixel value in layer j+1. With different kernels, we can learn different representations of data and the amount of parameters is not increased
exponentially with the number of hidden nodes and layers.
Once a feature has been detected, its exact location becomes less important . Only its approximate
position relative to other features is relevant. Not only is the precise position of each of those features
irrelevant for identifying the pattern, it is also potentially harmful because the positions are likely to vary for different instances of the pattern. Max pooling layers do sub-sampling operation on feature maps. For every four pixels, we only retain the max value, so that the size of feature maps will be half of the original size. Sub-sampling reduces the resolution of feature maps and reduces the sensitivity of the output to shifts and distortions, so that the model will be more robust. Max pooling operation can be incorporated into convolutional layers, and we don't need additional layers to do sub-sampling



Model train and run in tensorflow

Pre-requisites:

I) OpenCV: We use openCV to read images of cats/Dogs so you will have to install it.

ii) Shape function:
can reshape this to a new 2D Tensor of shape[16 128*128*3]= [16 49152].

iii) Softmax: is a function that converts K-dimensional vector ‘x’ containing real values to the same shaped vector of real values in the range of (0,1), whose sum is 1. We shall apply the softmax function to the output of our convolutional neural network in order to, convert the output to the probability for each class.

$$o(x)j=exi∑Nn=1exnforj=1….N$$

Reading inputs:

I have used 37000 images of dogs and cats each from Kaggle dataset but you could use any n image folders on your computer which contain different kinds of objects. Typically, we divide our input data into 3 parts:

Training data: we shall use 80% i.e. 0 images for training.
Validation data: 20% images will be used for validation. These images are taken out of training data to calculate accuracy independently during the training process.
Test set: separate independent data for testing which has around 400 images. Sometimes due to something called Overfitting; after training, neural networks start working very well on the training data(and very similar images) i.e. the cost becomes very small, but they fail to work well for other images. For example, if you are training a classifier between dogs and cats and you get training data from someone who takes all images with white backgrounds. It’s possible that your network works very well on this validation data-set, but if you try to run it on an image with a cluttered background, it will most likely fail. So, that’s why we try to get our test-set from an independent source.
dataset is a class that I have created to read the input data. This is a simple python code that reads images from the provided training and testing data folders.

The objective of our training is to learn the correct values of weights/biases for all the neurone in the network that work to do classification between dog and cat. The Initial value of these weights can be taken anything but it works better if you take normal distributions(with mean zero and small variance). There are other methods to initialise the network but normal distribution is more prevalent. Let’s create functions to create initial weights quickly just by specifying the shape.

Creating network layers:

i) Building convolution layer in TensorFlow:-

tf.nn.conv2d function can be used to build a convolutional layer which takes these inputs:

input= the output(activation) from the previous layer. This should be a 4-D tensor. Typically, in the first convolutional layer, you pass n images of size width*height*num_channels, then this has the size [n width height num_channels]

filter= trainable variables defining the filter. We start with a random normal distribution and learn these weights. It’s a 4D tensor whose specific shape is predefined as part of network design. If your filter is of size filter_size and input fed has num_input_channels and you have num_filters filters in your current layer, then filter will have following shape:

[filter_size filter_size num_input_channels num_filters]

strides= defines how much you move your filter when doing convolution. In this function, it needs to be a Tensor of size>=4 i.e. [batch_stride x_stride y_stride depth_stride]. batch_stride is always 1 as you don’t want to skip images in your batch. x_stride and y_stride are same mostly and the

choice is part of network design and we shall use them as 1 in our example. depth_stride is always set as 1 as you don't skip along the depth.

padding=SAME means we shall 0 pad the input such a way that output x,y dimensions are same as that of input.

After convolution, we add the biases of that neurone, which are also learnable/trainable. Again we start with random normal distribution and learn these values during training.

Now, we apply max-pooling using tf.nn.max_pool function that has a very similar signature as that of conv2d function.

Notice that we are using k_size/filter_size as 2*2 and stride of 2 in both x and y direction. If you use the formula (w2= (w1-f)/S +1; h2=(h1-f)/S +1) mentioned earlier we can see that output is exactly half of input. These are most commonly used values for max pooling.

Finally, we use a RELU as our activation function which simply takes the output of max_pool and applies RELU using tf.nn.relu

All these operations are done in a single convolution layer. Let's create a function to define a complete convolutional layer.

ii) Flattening layer:-

The Output of a convolutional layer is a multi-dimensional Tensor. We want to convert this into a one-dimensional tensor. This is done in the Flattening layer. We simply use the reshape operation to create a single dimensional tensor

iii) Fully connected layer:-

Now, let's define a function to create a fully connected layer. Just like any other layer, we declare weights and biases as random normal distributions. In fully connected layer, we take all the inputs, do the standard z=wx+b operation on it. Also sometimes you would want to add a non-linearity(RELU) to it. So, let's add a condition that allows the caller to add RELU to the layer.

iv) Placeholders and input:-

Now, let's create a placeholder that will hold the input training images. All the input images are read in dataset.py file and resized to 128 x 128 x 3 size. Input placeholder x is created in the shape of . The first dimension being None means you can pass any number of images to it. For this program, we shall pass images in the batch of 16 i.e. shape will be [16 128 128 3]. Similarly, we create a placeholder y_true for storing the predictions. For each image, we have two outputs i.e. probabilities for each class. Hence y_pred is of the shape (for batch size 16 it will be .

v) Predictions:-

As mentioned above, you can get the probability of each class by applying softmax to the output of fully connected layer.

y_pred = tf.nn.softmax(layer_fc2,name="y_pred")

y_pred contains the predicted probability of each class for each input image. The class having higher probability is the prediction of the network. y_pred_cls = tf.argmax(y_pred, dimension=1)

Now, let's define the cost that will be minimised to reach the optimum value of weights. We will use a simple cost that will be calculated using a Tensorflow function softmax_cross_entropy_with_logits which takes the output of last fully connected layer and actual labels to calculate cross_entropy whose average will give us the cost.

vi) Optimisation:

Tensorflow implements most of the optimisation functions. We shall use AdamOptimizer for gradient calculation and weight optimization. We shall specify that we are trying to minimise cost with a learning rate of 0.0001.

optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimise(cost)

As you know, if we run optimiser operation inside session.run(), in order to calculate the value of cost, the whole network will have to be run and we will pass the training images in a feed_dict(Does that make sense? Think about, what variable would you need to calculate cost and keep going up in the code). Training images are passed in a batch of 16(batch_size) in each iteration.
where next_batch is a simple python function in dataset.py file that returns the next 16 images to be passed for training. Similarly, we pass the validation batch of images independently to in another session.run() call.
Note that in this case, we are passing cost in the session.run() with a batch of validation images as opposed to training images. In order to calculate the cost, the whole network(3 convolution+1 flattening+2 fc layers) will have to be executed to produce layer_fc2(which is required to calculate cross_entropy, hence cost). However, as opposed to training, this time optimization optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(cost) will not be run(as we only have to calculate cost). This is what changes the gradients and weights and is very computationally expensive. We can calculate the accuracy on validataion set using true labels(y_true) and predicted labels(y_pred).
We can calculate the validation accuracy by passing accuracy in session.run() and providing validation images in a feed_dict.

val_acc = session.run(accuracy,feed_dict=feed_dict_validate)

Similarly, we also report the accuracy for the training images.

acc = session.run(accuracy, feed_dict=feed_dict_train)

As, training images along with labels are used for training, so in general training accuracy will be higher than validation. We report training accuracy to know that we are at least moving in the right direction and are at least improving accuracy in the training dataset. After each Epoch, we report the accuracy numbers and save the model using saver object in Tensorflow.

Prediction:

PYTHON LIBRARIES REQUIRED

* Pandas
* Numpy
* Scikit-Learn
* Pylab
* Matplotlib
* tqdm
* cv2
* os
* random

* tflearn

* tensor flow

USEFUL FRAMEWORKS

* SPYDER
* JUPYTER NOTEBOOK

Code

code is given on catvsdog.py file