Image Classification using a Convolutional Neural Network

By: Justine Bernshine

For COMP 3190 (Introduction to Artificial Intelligence)

Deep neural networks have taken off over the last few years. With the advent of faster processors and in particular the introduction to the use of Graphics Processing Units for compute purposes, we are seeing neural networking projects becoming more accessible and mainstream. With the introduction of frameworks and libraries to simplify deep learning programming, it has become more and more accessible to developers and data scientists trying to understand their data.

One of these deep learning networks is called a Convolution Neural Networks. It is commonly used for image classification, where images are classified into one or more classes defined by the model. It is one of the most important problems with the area of computer vision, being the basis of other computer vision problems. (Rawat & Wang, 2017) I will explain the basics of how Convolutional Neural Networks work, then try to implement one to see if it can be used to train it to understand cartoon characters.

Introduction to Convolution Neural Networks

Convolutional Neural Networks are inspired by biology. It is an attempt to replicate the visual cortex of the brain. (Rawat & Wang, 2017). The basic architecture of a convolutional neural network includes a convolutional layer, a pooling layer, a fully connected layer, then the output layer where it gives us the most probable classification. The convolutional, pooling and fully connected layers are the hidden layers of the neural network. Since there are multiple hidden layers, at least two, it is called a Deep Convolutional Neural Network. There can be multiple convolutional and pooling layers within the net. It's not just limited to one pair of layers before hitting the fully connected layer. Next, I will go over the specifics of the convolutional and pooling layers and explain how they are determined.

Convolutional Layer

The convolutional layer(s) job is to create feature maps. The convolution process involves taking the image data, which in this case is the pixels, and turning it into a feature map to be used in the later stages. In this project, the feature map is created by using a window of a certain size, sending it over our image data to pick out the strongest feature for that area of the image by pixel, which then becomes the value for that spot on the feature map. It is given by the equation with Y_k is our kth feature map

$$Y_k = f(W_k * x)$$

where the input image is x, the filter of the convolution for the kth feature map, the multiplication sign denotes the 2D convolutional operator used to calculate the inner product of the filter model at each location of our input image. The f(-) function as a whole is the nonlinear activation function, which we have several choices to choose from. Some examples include the sigmoid and hyperbolic tangent. However, the rectified linear units (ReLUs) function has gained popularity and is the activation function I used in this project for convolutions. (Rawat & Wang, 2017)

The activation function can be seen as a neuron firing, or not, depending if the value is above some threshold. It outputs a value to be used in the next layer. This is where the neural part of neural networks comes in. You see these at the end of each layer in the neural network.

Pooling Layer

The Pooling layer takes the feature map created in the convolutional layer and reduces it even further, creating a smaller map. Reducing the dimensions of the convolutional map helps deal with input distortions that can appear at that layer. There are two main types. One is average aggregate pooling, which takes the average value of a convolutional map over a given area.

Lately however there is a move to max pooling, taking the maximum value instead of the average of the values over the area. It is defined by this equation:

$$Y_{kij} = \max_{(p,q)\in Rij} x_{kpq}$$

where the output, Y_{kij}, is the output for the kth feature map. x_{kpq} is an element at location (p, q) in the pooling region of Rij, which is a receptive field around position (i, j) in the feature map. (Rawat & Wang, 2017)

The size of the pooling map depends on the size of the input feature map, along with the size of the pooling filter and the stride of such a filter.

Fully Connected Layers

After going through all the convolutional and pooling layers to find features of our images, we end up at the fully connected layers. These layers take the results of our layers from before and carry out the higher-level reasoning that the mind performs. In the case of classification, the end result is what we use to determine what the image is classified as. Typically, you see a SoftMax activation at the end of the fully connected layers to make a decision. (Rawat & Wang, 2017) In the case of a binary decision where there are only two choices, you can also use a sigmoid function.

The Project

For this project, I took two Simpson's characters from a Kaggle data set and sent them through the neural network for this project. (alexattia, n.d.) I specifically picked Homer Simpson and Ned Flanders, since they have the most training images in the set. Since Homer had much more training images than Ned, I had to delete a number of images in excess of Ned. This is to keep the training balanced. If you have more of one classification than another, the model will be trained to pick that one over another.

The code itself was inspired from a tutorial I saw about Tensorflow and Keras. (Kinsley, 2018) Kinsley's tutorial focused on classifying cats and dogs. I wondered if it could also classify

cartoon characters. I picked The Simpsons data set because it was publicly available and was easy to handle in this context.
The first step was to collect our image data. Then using the OpenCV library I converted the 3-dimentional RGB data into a single dimensional grayscale. I also reduced the pixels in the image to keep it standard, since each picture can have slightly different sizes. This also simplifies our data and requires less processing resources since I can only run this on a CPU and not a graphics card.

The project consists of two Python scripts along with training and testing data. The testing.py script loads in the training images, creates the neural network, and runs it. It then saves the model into a file that can be loaded and ran over and over again for testing without having to retrain. The code mostly follows the tutorial, with a few changes such as the increased image size and the handling of files.

The neural network is created with the assistance of Keras, a library for Tensorflow that simplifies the creation of neural networks. Libraries such as Numpy and OpenCV is used to process the data, to get it into a format Tensorflow will use. The neural network consists of two convolutional layers, each of which is followed by a pooling layer. After this, the data from these layers is flatten, then combined until it gets to a single value, which is then used to determine if the image is Homer or Ned. The model is saved to a file.

The testing script, testing.py, takes the model file created earlier and applies it to the testing data of Homer and Ned images. These images are separate; they were not included in the training data. Each image is loaded, preprocessed for testing, then predicted. This makes it the ultimate test of its effectiveness. It then prints out the results and statistics of testing for each character individually and then as a whole.

Results

At first, I was expecting that this would work. After all, it was shown to work with cats and dogs. Unfortunately, it kept predicting that almost every image was of Homer and not Ned. Every time I retrained the model, I kept getting the same result.

I want to speculate on why this might have been the case. One of the reasons I suspect is that The Simpsons images uses standard colours throughout. Since it is a cartoon, it uses less colour then it would be if it is real life. This reduces the diversity of possible RGB values, which in turn would affect the grayscale values. There is just less diversity of colour in The Simpsons, causing the model to be ineffective.

Another reason on top of that could be the number of training images. The cats and dogs dataset had thousands of examples for each, giving the neural network lots of data to process. Meanwhile The Simpsons dataset only has around 1500 images each. I also had to delete a bunch of Homer images from training to keep it balanced. Without more data, the model can't learn effectively.

The other reason I suspect is that the Convolutional Neural Network doesn't do face detection. It works on the entire image. This means that irrelevant data is being used for training and testing. It's the whole image being trained and tested, not the face in it.

Combined with my reasons put together, they could be the cause why it failed. In the future, I would like to get this working with the other characters of The Simpsons. I've heard it is possible to assign weights to the classifications. This would make it possible to use training data that is unbalanced. I also would like to use colour data instead of grayscale. There would be more diversity for the neural network. It wouldn't solve the colour diversity problem that all cartoons have, but it would make it slightly better.

Bibliography

alexattia, n.d. *The Simpsons Characters Data.* [Online]

Available at: https://www.kaggle.com/alexattia/the-simpsons-characters-dataset

[Accessed 20 December 2018].

Kinsley, H., 2018. *Introduction to Deep Learning - Deep Learning basics with Python, TensorFlow and Keras.* [Online]

Available at: https://pythonprogramming.net/introduction-deep-learning-python-tensorflow-keras/

[Accessed 20 December 2018].

Rawat, W. & Wang, Z., 2017. Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. *Neural Computation,* 29(9), pp. 2352-2449.