COMP309 Final Project:
Image Classification

October 29, 2019

Maria DaRocha, StuID#300399718
Trimester Two, 2019

Introduction

Problem description:
Classification of digital images into one of three categories: cherry, strawberry, or tomato.

Approach:
Construct a sequential CNN model using python as a language and keras/tensorflow as model frameworks. Break the problem down into various layers (greater detail on layer types below). Compile and train the model using feature extraction [from images], pooling, regularization, and cross validation. Utilize activation functions, bias, and evaluation metrics to optimize the final model.

Problem Investigation

A **Convolutional Neural Network (CNN)** is a deep learning algorithm/model which takes an input image, learns relative features (including their weights and biases), and is then able to differentiate these objects from one another. CNN's may possess any number of layers, including: convolution, ReLU/tanh/sigmoid, pooling, normalization, dropout, and fully-connected (Dense, MLP).

- **Convolutional layers** slide low dimensional filters across an image, which produces an entire feature map. An example of this would be *convoluting grayscale pixel values*.
- **ReLU, tanh, or sigmoid layers** introduce non-linearity to the network and activate that model's convolution layer. They must be used after every convolution operation. ReLU is an abbreviation of 'Rectified Linear Unit' and acts as a node (or unit) to *activate the convolution layer*.
- **Pooling (sub/down-sampling)** refers to spatial pooling, which reduces the dimensionality of a feature map while retaining the most important information. Pooling can refer to the pooling of averages, sums, differences, minimums, maximums, means, etc. An example of this would be *pooling the RGB values from an image and retaining only the maximums*.
- **Normalization layers** are transformative layers which, as the name implies, normalizes the outputs of the previous layer. They have *minimal efficacy* and therefore haven't been used in this model.
- **Dropout layers** are a regularization technique which *reduces overfitting* by randomly dropping units (and their connections) from the network.
- **Fully-connected (dense) layers** are multi-layer perceptrons which are used to classify an input image into one of various classes.

Why this model is 'Sequential':

Simply put, layers must occur in a particular sequence to construct, compile, and classify images correctly…

> *The penultimate goal of a CNN is for its layers to connect in such a way that the model learns features through convolution, introduces non-linearity through activation functions, reduces dimensionality (while preserving partial invariance) through pooling, produces high-level features, and then uses dense layers to return an image's likelihood (probability) of belonging to a particular class.*

Exploratory Data Analysis (EDA) utilized tensorboard to investigate losses, and ran feature extraction independently of the CNN model to identify error and noise-prone images. Feature extractions for the model include: **Grayscale pixel values**, **Max RGB pixel values**, **Prewitt Vertical Edge Detection**, and **Prewitt Horizontal Edge Detection**. The array outputs of these convolution and pooling layers were explored through a dataframe. Additionally, an instance of an image was initially explored in great detail to gain an understanding of the layers' applications. These functions are well-annotated within train.py (*bw_eda( )*, *full_color_eda( )*, and *edge_detection_eda( )*).

The **preprocessing steps** involved weeding out samples which did not consistently reshape (300x300) correctly. (Required three iterations of reshaping and removing images). Following this, any additional photos which had a grayscale filter already applied to them were removed to prevent errors in "Max_RGB" (color) pooling.

Methodology

Using Keras and Tensorflow to Create a CNN

Images were fed into the model by walking through three separate directories (cherry, strawberry, and tomato) where training images were located. This process consolidated all image file paths into a single csv file. This csv file was then read back into the program, pre-processed using the aforementioned techniques, and then provided an image with its correct class label based on an "*if string contains…*" premise. (i.e. for this model, an image of a strawberry would have the subset 'strawberry' within the full string value, and so on).

The dataset was then shuffled to prevent all test cases from belonging to a single class, and split into X_train (image paths), y_train (class labels), X_test (image paths), and y_test (class labels). This split defined the training and testing directories and their respective labels.

The model constructs itself with two 2D convolution layers that are passed parameters: desired number of nodes (64, then 32 - chosen based on bits), the kernel's input size (3x3 - i.e. the scale of the "sliding" operator), the activation function (ReLU, Rectified Linear Unit), and the expected input shape (300x300x1, where 1 indicates a single grayscale channel instead of three RGB channels).

The model then applies a flattening layer to connect the convolution layers and the dense layers. This reshapes the tensor to match the number of elements it contains.

The final two layers attempt to extract high-level classifications from the model. The first specifies an output dimension of 64, uses ReLU again as an activation function, and expects an input dimension of 100. The second dense layer outputs a dimension of 10 and uses a "*softmax*" activation function that translates the outputs into probabilities by summing them to 1 (ex. *Strawberry => 78%*)

In keras, all CNN's possess three requirements to successfully compile: an **optimizer**, a **loss function**, and **specified evaluation metrics**.

Optimization: Stochastic Gradient Descent (SGD)

The optimizer used was stochastic gradient descent (SGD). It computes a finite approximation of the gradient descent operation.

In a gradient descent algorithm, the system feeds forward the present pattern at the input layer (propagates the forwards activations) for the nodes, then calculates the error for the output neurons and propagates backwards, calculates the partial derivatives, repeats for all patterns, and then sums. ***In stochastic gradient descent***, the gradient is repeatedly approximated by a single example at the respective learning rate until the algorithm converges.

(Expanded:) It is a technique used to minimize the error function. This approach uses derivatives to adjust input values. Because several weights need adjustment, partial derivatives are applied, (which is when one derives with respect to a single variable while holding the others constant). "True" stochastic gradient descent does not make use of vectorization and thus iterates over each step separately. This can be temporally expensive, and so a common remedy is to use batches.

- $dy/dx > 0$ implies that y increases as x increases...
  Thus to find the minimum y, we reduce x.
- $dy/dx < 0$ implies that y decreases as x increases...
  Thus to find the minimum y, we increase x
- $dy/dx = 0$ implies presently at a local minimum or maximum.

Loss: Categorical Cross-Entropy

The loss function used was categorical cross-entropy (sometimes known as *log loss*, given that they resolve the same way in machine learning). Cross entropy loss produces an output between 0 and 1, where a perfect model's log loss would be 0. Loss increases as the predicted value diverges from the actual value.

- $Cross\ Entropy\ (Log\ Loss,\ M = 2) = -(y * log(p) + (1 - y) * log(1 - p))$
- $Cross\ Entropy\ (Log\ Loss,\ M > 2) = -\sum_{c=1}^{M} y_{o,c} * log(p_{o,c})$

Where…

- ➔ M = number of classes
  - ◆ If M > 2, calculate a separate loss for each label
- ➔ *log* refers to the natural log
- ➔ y = binary indicator for correct or incorrect label
- ➔ p = predicted probability observation is of class *c*

Evaluation: Accuracy, Categorical Accuracy, & MAE

The desired evaluation metrics were accuracy, mean absolute error (MAE), and categorical accuracy.

Accuracy refers to a straight-forward evaluation: comparing the model's number of correct classifications to its number of incorrect classifications in a raw, uniform way.
It is a bad performance indicator if there is a large class imbalance (in favor of one majority class). This is because accuracy scores may not penalize misclassification harshly enough to reflect a model's true performance, and because recall instances may be more detrimental in "real world" applications of the model than its false negative counterparts.
However, in the current model the classes are well-balanced and recall penalties should be treated with the same weight as other misclassifications (i.e. - classifying a cherry as a strawberry is no more detrimental than the reverse). These points offer assurance of accuracy's reliability as a performance metric in this scenario.

- ***In keras***, passing *'accuracy'* as a performance metric will have the program automatically ascertain if the output metric should be expressed as accuracy or categorical accuracy (in the case of multi-class classification). Multi-class classification is used to calculate accuracy per class.

Mean Absolute Error (MAE) takes the average magnitude of errors over a sample while disregarding their direction. As a result, all of the differences are equally weighted. It is the average of the absolute differences between predicted and observed values for a sample set. It is only undefined when the predicted values and the observed values match exactly.

- $MAE = \frac{1}{n}\sum_{k=1}^{n}\left|y_k - \hat{y}_k\right|$

Ideally, there would have followed implementations which fine-tuned the model. Examples of this would include applying filters (e.g. - salt & pepper) which cause to moderate-low image interference. This approach offers a way to decrease the likelihood of model over-fitting.
Fine tuning also would have involved the investigation of activation functions to better assess how they compare to one another (e.g. - sigmoid or tanh).
For example, a sigmoid activation function differs from a SGD activation function.
If we apply a standard logistic function (sigmoid activation function) to gain the output of neuron *i* for pattern *p,* we attain a more "S"-shaped curve and neurons are updated using the generalized delta rule:

$$\Delta w_{ij} = \eta\, \delta_i^p\, x_{ij}$$
$$\delta_i^p = (t_i^p - o_i^p) f'(u_i)$$

and account for hidden nodes with

$$\delta_i^p = \sum_k w_{ki}\, \delta_k f'(u_i)$$

Fine-tuning can also be done by updating the biases of nodes (i.e. investigating optimum values and adjusting their weights accordingly).

Manual Feature Extraction (Demonstration of Convolution & Pooling)
I began by using a filewalker which utilized the os and csv libraries to walk along training data file paths and produce a csv file containing the file paths to all training data images.
Following this, the images were read into the program as a list of strings. I then produced an "answer key" of the pre-processed data as a means of generating a class label set.

The first feature extraction was a grayscale value extraction, an example of convolution techniques. This extracted pixel values from an image in its black and white form. The computational time for this step was relatively short, given information was being consolidated from only one channel.

The second feature extraction was a pooling process involving feature extraction from three channels to gather respective color values (red, green, and blue). Instead of processing a 3D matrix of 270,000RGB values per image, only the maximum of the three values was retained for each pixel. This process reduces dimensionality to 90,000RGB values per image, and by extension the required computation time.

The third and fourth feature extractions were also convolutional techniques that applied a Prewitt operator on the pixel matrix to identify vertical and horizontal edges. This process uses images in grayscale, and thereby one channel (making it convolution, as opposed to pooling). Edge detection is done by Identifying drastic changes in pixel values. Manually, this would be done by taking the difference between a selected pixel value and its adjacent pixel values. Instead, we are able to use scikit's Prewitt kernel ([-1,0,1],[-1,0,1],[-1,0,1]). By multiplying the values surrounding the selected pixel against the kernel, adding the values becomes equivalent to taking the difference.

-1	0	1
-1	0	1
-1	0	1

Prewitt Operator:
The Prewitt operator slides along the full matrix of pixels to identify drastic changes and detects vertical and horizontal edges.

Results

This model was unable to fully compile and reach the results phase. However, it is certain that its performance and computation time would surpass that of the baseline method multi-layer perceptron (MLP). The process of abstracting features and transforming them into higher-level information is done explicitly to enhance the processing speed and accuracy of a CNN, making it more favorable to use for complex deep learning. Within the code, an attempt was made to use a basic MLP to produce the same classifications but this model had not finished executing after 30 minutes of running, and was halted. This MLP was built using a similar approach to the CNN. It collated feature data into arrays, and then organized the extracted information into a dataframe. This serves as an example of how a well-defined MLP may fail to out-perform a simple CNN.

Conclusions

The conclusions drawn from this work are that CNN's are effective algorithms for complex deep learning problems. As previously mentioned, they offer *one solution* to complex classification problems, which is a massive feat in the world of machine learning. However, there are drawbacks that lie in their complexity, as well. They require time and resources to train extensively with fine tuning. It is also of note that visibility decreases as information abstractions reach higher levels through the layers of a network. Abstraction allows for easier classification, but also equates to information loss. An easy oversight for this project would also be the drawback of extensive data pre-processing that is often required to build CNN's. Fortunately, this project provided a large quantity of training samples that were reasonably clean, but it is much more likely that rigorous cleaning and exploration would be needed for a "real world" application of a CNN model.

Proposals for future work built on this include: expanding the classifications beyond the three classes that are already present. It would be possible to add more fruit classes (of the same or different colors), or classes that contain different categories altogether (e.g. - broccoli). Another possibility would be to use the whole image and classifications of objects within it to automate captions (e.g. full color image output with the sub-text "a tomato on a vine").

References

https://medium.com/human-in-a-machine-world/mae-and-rmse-which-metric-is-better-e60ac3bde13d

https://towardsdatascience.com/how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-1-regrression-metrics-3606e25beae0

https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html